A Protocol-Independent Internet Transport API

master thesis in computer science

by

Stefan Jörer

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of Master of Science

supervisors: Dr. Michael Welzl, Department of Informatics
University of Oslo
Dr. Radu Prodan, Institute of Computer Science
University of Innsbruck

Innsbruck, 16 December 2010

Certificate of authorship/originality

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Stefan Jörer, Innsbruck on the 16 December 2010

Abstract

The conjoint API of TCP, UDP, UDP-Lite, SCTP and DCCP is complicated. By requiring an application to specify the protocol to use, it also restrains the flexibility of implementing the Internet's transport layer. Therefore, a common API is proposed that only offers the services that application programmers need to see. Moreover, this thesis shows how its design could be undertaken, ending up with an API which is much simpler than the alternatives in use today. Furthermore, one can argue that this might be the only way to get new transport protocols widely deployed.

Acknowledgments

I would like to express my appreciation to my supervisors: Dr. Michael Welzl at the University of Oslo and Dr. Radu Prodan at the University of Innsbruck. I want to thank Dr. Radu Prodan for giving me the possibility to join the Networks and Distributed Systems group at the University of Oslo for outstanding research. Special thanks to Dr. Michael Welzl who was never tired in discussing details and was always convinced of the idea behind this thesis.

Moreover, I want to thank the institutions which have made this research work at the University of Oslo financially possible; the Erasmus program and the Julius-Raab-Stiftung.

My gratitude goes to my parents who have always supported my studies and especially for standing behind me at the end of writing this thesis. I also want to thank my sister Elisabeth as well as my brother Martin who both helped me wherever they could.

Finally, I want to say thanks to Jolanda and Theresa for proof-reading this master thesis.

Contents

Chapter 1

Introduction

Currently we have a peculiar situation in the network application development area. A lot of research has been carried out to develop protocols that match today's application requirements. These efforts have led to three new transport protocols, which have already been standardized by the *Internet Engineering Task Force* (IETF), and even more new features for these and existing protocols are on the way to become a standard. Although there exist several advantages in using them, they hardly get deployed in today's Internet. This complex situation is described as a **"transport tussle"** in [1].

1.1 Transport tussle

The tussle consists of the following three participating parties: application designers/developers, operating system (OS) developers and designers of middleboxes or firewalls.

In general, the first group (**application designers/developers**) tries to get the best performance for their applications with minimal programming effort. Hence, it does not make sense for them to try a new transport protocol which would fit their requirements better as long as they do not see a good chance that it will succeed. The needed extra amount of effort is not worth trying something new facing a risk of failure. But what is this needed extra amount of work? The developers would have to build some kind of fallback mechanisms which are used in the case of a failure. Therefore, they would have to develop parts of the application twice: one time for the new technology, realizing the attainment of a possible performance advantage, and another time as backup solution that ensures reliability. Naturally the willingness for investing extra effort grows with the probability that new protocols will work successfully. Nevertheless, only very few application programmers would like to invest the double amount of work for a possible small advantage.

The **OS developers** have their focus on minimizing the risk that new technology could lead to. Additionally, they want to deliver the best performance within the given risk constraint. Since it is possible to develop “safe” code for new protocols, there is no need to hinder the deployment of new transport protocols. Despite this fact some OS manufacturers still block new technology as long as it does not have a wide-spread usage (e.g. Microsoft still does not support SCTP by default).

The **designers of middleboxes/firewalls** mainly pay attention to security issues as well as maintainability. Hence, they are very restrictive with new protocols and technologies, because something unknown is a potential security risk. In general the restrictions lead to a behavior of *“whatever is unknown is blocked”*. Nevertheless, they also want to deliver good performance, because otherwise the *Internet service providers* (ISPs) are not able to serve their costumers in an appropriate way.

The tussle starts when looking at all three stakeholder parties. They are simply harming the deployment of new technology by following their own interests. The first group will not develop applications which use new technologies as long as they do not see that the advantages prevail. On the other hand, new technology is blocked by the designers of middleboxes as long as there is no real need for supporting it, which would exist if applications used new technology. The third party, the OS developers, has to support new protocols and often consists of designers of firewalls as well. They do not harm the deployment directly, but they do also not use the possibility of improving the situation for the other two parties, which will be explored in the following.

1.1.1 How to solve the transport tussle?

The solution can be approached in two steps: The first one is called “Beneficial Transparent Deployment” [1]. This step involves only actions for the OS developers which seem to be most flexible party of all. The transparent deployment should be enabled by providing an automatic fallback mechanism for new technology. Hence, no disadvantage can be exploited by the application developers, because the fallback mechanism is built-in transparently and the only disadvantage of using a new protocol could be the loss of the performance benefit. This may already lead to a more wide-spread usage of new transport protocols and therefore give the designers of middleboxes a reason not to block new technology anymore, as blocking would lead to a performance disadvantage for some applications. By facilitating the usage of new protocols even more application designers will consider the usage of a more suitable protocol for their application and which will lead to a constant improvement of the situation.

The second step “A new *Application Programming Interface* (API)” [1] could help to speed up step one by offering the application developers an easier usage of new technology. The existing transport APIs are rather complex and even not standardized for new transport protocols. This has led to an unsustainable situation for network application developers, where they have to carry out a lot of analysis when developing applications for different operating systems. The work that has been carried out during this thesis is the analysis of current transport APIs as well as a proposal of a new protocol-independent transport API which is needed to provide application developers with a sustainable utility to use new technology in a transparent and uniform way.

1.2 Motivating a protocol-independent transport API

The preceding section has shown the tussle of today's Internet. Here, we want to stress the need for a new transport API further. The first argument leads to the conclusion that a new transport API is needed whereas the second one yields an explicit argumentation for a protocol-independent API and the third one strengthens the need for a protocol-independent transport API. Finally, an insight in today's network application development cycle which obviously confirms the necessity for a protocol-independent transport API is given.

1.2.1 Complex proposed standards/no standards of new protocols

As already shown before, we face a situation where the Internet-Draft "Sockets API Extensions for Stream Control Transmission Protocol (SCTP)" [2] is already in version 24, has 99 pages and seems to be quite complex. Another aspect is that as long as it is only a draft and not a standard, there exist quite a few differences between this document andreal implementations. Actually the implementation which is closest to the draft has quite a few differences too. Hence, it would be an improvement to introduce a new transport API which is standardized in an operating system independent manner.

1.2.2 Wealth of choices offered by new protocols

The development of new transport protocols has led to a wealth of transport services which are more suitable for today's Internet application needs. Since the application developers have to decide on these features, they need a lot of knowledge of how they can profit from using them. This knowledge constraint combined with the current protocol dependent usage of features makes it hard to figure out the right choice for an application. Therefore, we suggest a protocol-independent API that minimizes the complexity of choices to a manageable amount and removes the protocol dependence from network applications by automatically making the best possible choice from the services that are available.

1.2.3 Gradual deployment of new protocols

The removal of protocol dependence of the transport API also enables the gradual deployment of new protocols by providing a fallback mechanism underneath the API for new transport technologies. The fallback mechanism is essential since otherwise new transport protocols have a problem for getting widely deployed (already mentioned in Section 1.1).

1.2.4 Network development cycle

Now we have a closer look at the "Network Application Development" cycle which consists of the same phases as the simplified and generalized "Iterative Development" cycle for software, but needs some special considerations concerning the network communication.

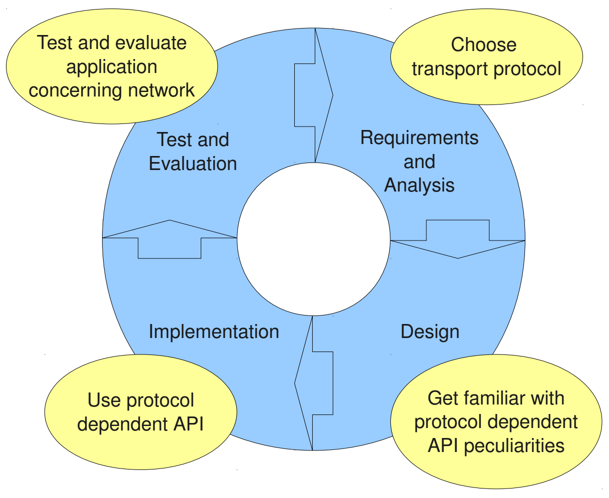
Test and evaluate application concerning network
Choose transport protocol
Test and Evaluation
Requirements and Analysis
Implementation
Design
Use protocol dependent API
Get familiar with protocol dependent API peculiarities

Figure 1.1: Typical iterative network application development cycle.

The cycle is shown in Figure 1.1 and can be explained as follows:

1. **Requirements and Analysis**: After the requirements for the application have been specified, the application developer has to choose a specific

transport protocol. This choice may even result in more than one protocol, because different communication needs may arise within one application.

2. **Design**: In order to make the right design decisions, the application developer has to get familiar with the protocol dependent API peculiarities and therefore often needs deep knowledge of the transport protocols.

3. **Implementation**: A protocol dependent API is used for implementing the network application.

4. **Test and Evaluation**: The test and evaluation phase needs to have a close look at the performance concerning the network communication.

The cycle is completed when the test and evaluation phase is satisfactory and all application features have been implemented. A problem may arise when an additional iteration is needed and this new iteration has new requirements which lead to a choice of a different or additional transport protocol, then the same efforts are needed again. In the past this was not likely to happen because the decision between the *Transmission Control Protocol* (TCP) and the *User Datagram Protocol* (UDP) was in most cases quite clear. Either an application needed the features of TCP or it did not. However, in the presence of five transport protocols with more features the decision gets tougher and hence more often wrong decisions would be made in the development of an application.

Therefore, it would be advantageous to have an API which abstracts the protocol peculiarities. The user should be able to make a decision with basic knowledge of transport layer characteristics rather than knowing a lot of implementation details of transport protocols which are even different among several operating systems. Then a user would just have to choose a service and if the requirements change later, only switch to a different service.

1.3 Overview

In Chapter 2 an insight of today's Internet transport protocols which have been standardized by the IETF is given. This yields a better understanding of the aforementioned wealth of choices and in addition helps us to carry out an analysis and intuitive reductions of available transport services in Chapter 4. Chapter 3 investigates available transport APIs as well as possible future transport APIs and hence leads to the decision on which the API is built upon in Chapter 5. Finally Chapter 6 concludes our proposal for a protocol-independent transport API and summarizes the open tasks which have to be investigated further.

Chapter 2

Background

This chapter is going to describe the background of the new proposed transport API. First we are focusing on the features of currently available transport layer protocols like TCP, UDP, UDP-Lite, SCTP and DCCP. At the end protocol support negotiation is going to be discussed, because it should be integrated in a future version of the new transport API for querying which transport protocols can be utilized.

2.1 Transport layer protocols

In the following the currently available transport protocols for the Internet protocol suite are described in detail. For the two well-known protocols (TCP and UDP) a rough overview is only given, whereas the two emerging protocols (SCTP and DCCP) get more attention due to their missing deployment until now.

2.1.1 TCP

The *Transmission Control Protocol* (TCP) provides reliable and therefore connection-oriented end-to-end transmission between two hosts over the unreliable *Internet Protocol* (IP) which operates at layer 3. In 1981 TCP has been described in principle in RFC 793 [3], but during the evolution a lot of features and changes were proposed. The first significant change was the incorporation of congestion control which has been seriously introduced by Van Jacobson in 1988 [4] after the first congestion collapse in 1986. The recommended algorithms of Van Jacobson and others are incorporated in RFC 5681 [5] which deals with TCP congestion control in detail. Moreover, TCP supports features like flow control, error detection and in-order delivery that will be described on the next pages.

Functionality

Connection establishment TCP establishes connections by the so called *three-way handshake*, which means that TCP uses 3 segments between the two participants. First the initiator *host A* sends a SYN segment with its initial sequence number J to the other side. Then the responder *host B* has to acknowledge the requested connection by sending a SYN containing its own initial sequence number K to the host A. Since the host B also has to acknowledge the SYN J, it combines its SYN K with the ACK $J + 1$ in one segment. Therefore, this segment is often called SYN ACK. The final step in the three-way handshake obviously is the acknowledgment of host B by sending ACK $K + 1$.

Congestion control has become a complex task for TCP. Nowadays a lot of different TCP versions (e.g. TCP Tahoe, Reno, Vegas, New Reno, FAST TCP, Cubic,...) exist and still a lot of research is going on in this area. Since this work does not deal with congestion control in detail here only a rough overview about the general concept is given. Commonly it is divided into two phases: *Slow Start* and *Congestion Avoidance*.

In the slow start phase the sender initializes the congestion window with 1. So it is able to transfer one segment of the *maximum segment size* (MSS). For each received *acknowledgment* (ACK) the sender increases the congestion window by one. In addition to check for the congestion window the sender has to take the minimum of congestion window and receiver window (the minimum is called current window), which is explained underneath, before he attempts to send more data.

After exploring the first loss due to congestion or overloaded network the congestion avoidance phase is utilized. If congestion is detected by duplicate ACKs or the retransmission timer expired for one segment, the congestion window is reduced to one half of the current window size. If congestion is indicated by a timeout, the window size is immediately set to 1 and the slow start phase starts again. The explained behavior in congestion avoidance is also called *Additive Increase/Multiplicative Decrease* (AIMD).

Reliability is one of the main features of TCP. TCP uses a byte-by-byte *sequence number* in combination with acknowledgments for ensuring reliability. When a host sends a segment, the sequence number of the first byte is written into the sequence number field and the receiver replies with an acknowledgment that contains the next expected sequence number. In general cumulative acknowledgment is utilized which means that the receiver acknowledges all bytes until the specified sequence number in the ACK segment.

Flow control is used for ensuring that the sender does not transmit the data too fast to the receiver. The receiver tells the sender in each segment in the receive window field how much bytes it can accept from the sender. The sender can only send up to that amount data. Then it has to wait until it receives an acknowledgment which specifies a new receive window which is big enough for the next segment(s).

Error detection is carried out by the use of a 16-bit end-to-end checksum. The term end-to-end checksum indicates that the data, contained in a segment, is protected against failures from the sender to the receiver.

In-order delivery The use of sequence numbers enables the receiver to reassemble the original data stream of the sender. Therefore the receiver blocks delivery of data until all preceding bytes are received correctly, called *head-of-line blocking* (HOL).

2.1.2 UDP

The *User Datagram Protocol* (UDP) enables a transport user to transmit data in an unreliable and unordered manner. Another point that distinguishes UDP from TCP is its *connection-less service*. In 1980 UDP has been described in RFC 768 [6] and didn't experience major changes. Instead of changes to UDP, other transport protocols such as UDP-Lite and DCCP, that are going to be described later, have been developed.

Functionality

Message-boundary preservation is carried out by the protocol itself by delivering only correctly reassembled datagrams.

Error detection is ensured by utilizing a checksum which is the 16-bit one's complement of the one's complement sum of the pseudo header information from the IP-header, the UDP header and the payload padded to be a multiple of two octets.

Unreliability Since UDP is an unreliable transport protocol, it does not care whether a datagram has been dropped in the network due to congestion or due to corruption of data. Therefore, the transport user has to remember to implement sufficient mechanisms for the needed communication reliability.

Lack of flow and congestion control Moreover, UDP does not support these two important transport features, because of its connection-less behavior. If UDP is used for long-lived data transmission, these features also have to be implemented at application level for ensuring network stability and application needs.

2.1.3 UDP-Lite

The *Lightweight User Datagram Protocol* (UDP-Lite) has been introduced because UDP cannot deliver partially corrupted payload. The so-called checksum coverage solves this problem by allowing an application to specify the coverage of the checksum. UDP-Lite has been proposed in RFC 3828 [7] and behaves like UDP if the checksum covers the entire datagram.

2.1.4 DCCP

Applications, including streaming audio, Internet telephony, and multiplayer on-line games, share a preference for timeliness over reliability. This fact leads to the growth of long-lived non-congestion-controlled traffic like UDP and UDP-Lite and causes a real threat to the overall health of the Internet [8].

For this reason the *Datagram Congestion Control Protocol* (DCCP) has been introduced in RFC 4340 [9] in 2006 for providing an unreliable, but congestion controlled transport service in order to fit the needs for these applications. Unlike TCP, which normally takes as much as bandwidth that is available as long as it is fair to others - called *TCP-like*, DCCP utilizes different kinds of congestion control methods due to the possibility of different requirements of applications. The currently available congestion control modules (*Congestion Control Identifiers*, CCIDs) are described in detail below. Hence, applications which benefit from controlling the trade-off between reliability and timeliness are well suited for the use of DCCP.

Congestion control identifiers

DCCP uses CCIDs to specify the congestion control mechanism in use on a half-connection. Therefore, one can employ different congestion control mechanisms on both directions of the data flow. In addition to the currently specified CCIDs, Section 4.3.4 is going to describe the usage of two possible new congestion control mechanisms for DCCP.

CCID 2 - TCP like congestion control This congestion control mechanism behaves like TCP congestion control with its AIMD behavior. CCID 2 closely follows the proposed *Selective acknowledgment* (SACK) TCP with the additional usage of *Explicit Congestion Notification* (ECN), but has the following notable differences:

- Congestion control is applied to acknowledgments instead on data packets.
- The units for congestion control are the number of packets instead of number of bytes, because DCCP is a datagram protocol.
- Retransmission is not facilitated since DCCP is an unreliable protocol.

CCID 2 should be chosen if one would like to take advantage of the available bandwidth in an environment with rapidly changing conditions [10].

CCID 3 - TCP-friendly rate control (TFRC) TFRC is a receiver-based congestion control mechanism that provides a TCP-friendly sending rate while minimizing the abrupt rate changes characteristic of TCP or of TCP-like congestion control [11].

TFRC uses a throughput equation to determine the sending rate, which depends on the loss-event rate, *round-trip time* (RTT) and packet size. Loss events are one or more packets lost or marked by ECN. A general description of TFRC's congestion control mechanism looks like as follows [12]:

- The receiver measures the loss event rate and feeds this information back to the sender.
- The sender also uses these feedback messages to measure the RTT.
- The loss event rate and RTT are then fed into TFRC's throughput equation, and the resulting sending rate is limited to at most twice the receive rate to give the allowed transmit rate X.
- The sender then adjusts its sending rate to match the allowed transmit rate X.

If the sender does not receive a feedback message within two RTTs, the sending rate is halved immediately. The sender has also to calculate the mean packet size and to measure the RTT.

CCID 4 - TFRC for small packets This congestion control mechanism was designed to support applications that send small packets with a similiar behavior as TFRC provides for greater packets. The specification in RFC 5622 [13] is experimental and is not ment to be widespread. Therefore, only a rough description of the main differences to CCID 3 follows:

- The transmit rate is reduced by factor that accounts for the header size.

- A maximum sending rate is ensured by requiring a minimum interval of 10 milliseconds between two data packets.
- The nominal segment size is set to 1460 bytes.

Functionality

The features of DCCP make use of several different packet types. Every packet type consists of a generic header and additional fields, depending on the packet type. The specific packet type header may be followed by options and they conclude the DCCP header. Figure 2.1 gives an overview of the used packet types during an example connection, where host A refers to the client and host B to the server.

Connection establishment Even DCCP uses a three-way handshake for connection establishment shown in line 1 - 3 in Figure 2.1. In the case of DCCP the first packet type transmitted by host A is called DCCP-Request which is the equivalent to a SYN packet for TCP. It is allowed to add already an application request as well as feature negotiations to the DCCP-Request and any subsequent packet - called *piggyback*. Then host B transmits a DCCP-Response to host A if it is willing to communicate with the host A. This packet may include feature negotiation responses and requests as well as an *Init Cookie* that has to be returned by host A for the connection to complete. Finally, host A has to acknowledge the DCCP-Response packet by an DCCP-Ack packet which confirms the initial sequence number of host B and responds to any Init Cookie.

Reliable acknowledgments During the data transmission phase DCCP-Data, DCCP-Ack and DCCP-DataAck packets are exchanged. DCCP-Data pack-

```
    Client                                     Server
    ------                                     ------
0.  [CLOSED]                                 [LISTEN]
1.  DCCP-Request -->
2.                                 <-- DCCP-Response
3.  DCCP-Ack -->
4.  DCCP-Data, DCCP-Ack, DCCP-DataAck -->
             <-- DCCP-Data, DCCP-Ack, DCCP-DataAck
5.                                 <-- DCCP-CloseReq
6.  DCCP-Close -->
7.                                    <-- DCCP-Reset
8.  [TIMEWAIT]
```

Figure 2.1: Example connection of DCCP [9].

ets carry only data, DCCP-Ack packets only acknowledge these data, whereas DCCP-DataAck contains data with piggybacked acknowledgments. In general, reliable acknowledgments rely on the particular chosen congestion control mechanism, but are normally achieved by *acks-of-acks.* This means ACKs are transmitted by the receiver until they are acknowledged by sender of the original datagram.

Connection teardown The connection teardown may be initiated by both hosts, but in Figure 2.1 the one initiated by host B is depicted. First the host B sends a DCCP-CloseReq packet requesting a close. Host A replies with a DCCP-Close packet acknowledging the close. Host B sends a DCCP-Reset packet with Reset Code 1 and clears its connection state information. When host A receives this packet it holds its state for *2 maximum segment lifetimes* (2MSL), because older packets could arrive during this time.

Resynchronization after bursts of loss is done via sending a DCCP-Sync packet and is immediately acknowledged with a DCCP-SyncAck packet by the other host. The resynchronization is needed for maintaining the **Loss Window**. DCCP uses the loss window to define the valid sequence number range. This will generally be set larger than TCP's receive window, as the goal is to cover a number of RTTs of packets in the window. Unlike TCP's receive window, it always moves up so that the greatest valid sequence number received is somewhere in the middle of the window.

Reliable negotiation of options The negotiation of options in DCCP is reliable and includes also the negotiation of a suitable congestion control mechanism. In general, the reliability of negotiation is achieved by the use of *Change* and *Confirm* messages exchanged by the two endpoints. Changes of options can be requested by both hosts. It does not matter whether the option has to be changed locally (L) or remote (R). For example a $Change(R, X, V)$ tells the other host: "Change option X to value V". Whereas $Change(L, X, V)$ means: "I want to change option X to value V". The same meaning applies to confirm messages which acknowledge the change of an option.

Flow control DCCP does not have an explicit reveiver window due to its datagram-based sending rate. However, it allows the receiver to tell the sender to reduce the sending rate in two ways: The *Slow Receiver* option means; "Do not increase your sending rate/window for one RTT." The Data Dropped (Drop Code 2) states; "Reduce your sending rate/window by the equivalent of one packet per RTT for every packet marked as Drop Code 2".

2.1.5 SCTP

The *Stream Control Transmission Protocol* (SCTP) is a connection-oriented, reliable transport protocol operating on top of a connection-less packet network such as IP. It has been introduced in 2000 by RFC 2960 [14] and has been updated later by RFC 4960 [15] in 2007. Originally SCTP was designed to transport Public Switched Telephone Network (PSTN) signaling messages, but it is also capable to support the needs of Internet applications. For being open to a wider spectrum of applications the support of partial reliable data transmission has been added in RFC 3758 [16].

Functionality

Currently SCTP is the transport protocol that offers most features compared to others and it is highly configurable. To achieve the flexibility that is offered by SCTP, it uses a combination of a thin common header followed by one or more chunks instead of inflating the header with many fields. Some of these chunk types are presented throughout the functionality description in the following paragraphs. For a complete list, please refer to RFC 4960 [15].

Association setup Since SCTP may use multiple IP-addresses to represent one host the term connection between host A (initiator) and host B has been replaced by *association* for affirming this fact. The startup procedure relies on a four-way handshake which is shown in Figure 2.2. Host A sends an `INIT` chunk with its verification tag to host B. After receiving the `INIT` chunk host B replies immediately with an `INIT ACK` chunk that contains the receive verification tag of host A, a new created verification tag of host B and the state cookie of B. So host B must not keep any state after this step, because otherwise it would be vulnerable against denial-of-service attacks.

In the third step host A returns the received cookie of host B in a `COOKIE ECHO` chunk which might be already bundled with pending `DATA` chunks. The last step is performed by host B with returning a `COOKIE ACK` chunk which can also be bundled with `DATA` and/or `SACK` chunks. The `COOKIE ACK` chunk must be the first one, because host A has to complete association initialization by receiving it and moving to the `ESTABLISHED` state.

Error detection The error detection in SCTP is done with a 32 bit CRC32c checksum which is calculated over the entire SCTP packet (including the common header as well as all contained chunks).

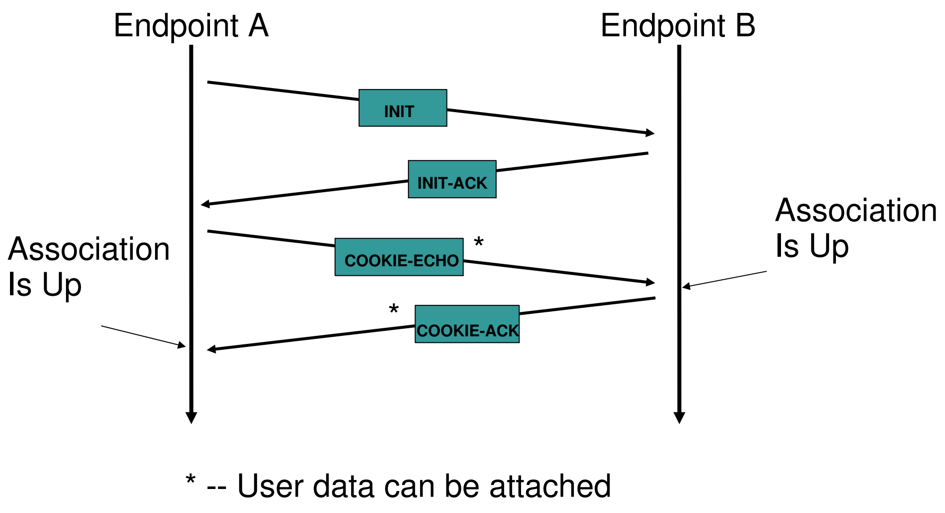

Figure 2.2: The association setup procedure in SCTP [17].

Reliable, partial-reliable and unreliable Transmission Originally, SCTP has been designed as a reliable transport protocol, but with RFC 3758 [16] partial reliability has been added. The partial reliability must be negotiated on association startup by a parameter. Currently only a timely partial-reliable transmission is standardized. This service allows the transport user to indicate a limit on the duration of time that the sender should try to transmit/retransmit the message [16]. The user is able to specify on a per message basis how persistent the message should be. Therefore, the user is also able to force a completely unreliable transmission.

Preservation of message boundaries SCTP preserves the boundaries of messages sent by a transport endpoint to the other. The preservation is provided with two features: fragmentation and bundling. Fragmentation is done when a message contained by one packet would exceed the current *Path Maximum Transportation Unit* (PMTU). A message is delivered on the receiver side when it has been resembled correctly.

Application protocol data unit bundling (PDU) The application PDU bundling is always done in congestion-like situations where as many chunks as possible are bundled together in order to come as close as possible to the PMTU. Bundling is optional in other situations and must be configurable if available for the sender side, but the receiver side has to support it anyway.

Congestion control SCTP uses the congestion control mechanisms defined in RFC 2581 [18] as well as *Selective acknowledgment* (SACK) and optional *Explicit Congestion Notification* (ECN), but has some differences to the TCP implementation of these features. Congestion control is done in per endpoint manner and not in per association manner. The variables for congestion control, such as congestion window and slow start threshold, are calculated for each endpoint separately.

Another important fact to keep in mind is that SCTP uses a *Transmission Sequence Number* (TSN) which numbers all transmitted data chunk. In general retransmissions are timer-controlled, where the time is derived from continuous measuring the RTT. When a retransmission timer expires, all unacknowledged data chunks are sent again and the timer is started with doubled initial duration. If the receiver detects a gap in the sequence of data chunks (at least one TSN is missing), each incoming packet is still acknowledged, but also all existing gaps are reported to the sender (negative selective acknowledgment). When the sender receives four consecutive SACKs reporting the same missing data chunk, this data chunk is retransmitted immediately (fast retransmit).

Flow control The flow control is similar to the one used in TCP. The receiver can control the rate at which the sender is allowed to send its data by specifying the receiver window size in SACK chunks.

Multi-streaming Two SCTP endpoints may use multiple streams within one association for communication. A stream is an unidirectional logical data flow that is negotiated during association setup. Each stream has its own *Stream Sequence Number* (SSN) for determining the sequence of delivery.

On the receiving side, SCTP ensures that messages are delivered to the SCTP user in sequence within a given stream. However, while one stream may be blocked waiting for the next in-sequence user message, delivery from other streams may proceed [15]. Therefore, the head-of-line blocking problem does not exist in the case of utilizing multiple streams.

In SCTP the number of streams has to be defined when setting up an association: "The SCTP user can specify at association startup time the number of streams to be supported by the association. This number is negotiated with the remote end. [15]"

Ordered and unordered delivery SCTP normally performs in-order delivery per stream, but there exist a mechanism to bypass the sequenced delivery and make received messages as soon as they are received available to the transport user. The unordered delivery is achieved by setting the flag `U` to 1 in the DATA chunk. If this flag is set, the SSN is simply ignored by the receiver.

Association takedown SCTP has two possibilities to end an association. The first one is a so called graceful close (shutdown of an association), which is initiated by one of the two hosts (lets say host A) by sending a `SHUTDOWN` chunk after any locally queued data has been acknowledged. SCTP does not accept any new data from the transport user after the shutdown primitive has been triggered by it. If host B does not reply to the `SHUTDOWN` chunk within a specified time, the `SHUTDOWN` chunk should be resend. When host B receives the `SHUTDOWN` chunk it stops accepting new data from its transport user, but transmits the already queued DATA chunks. After finishing the data transmission host B replies with a `SHUTDOWN ACK` chunk. When host A receives the `SHUTDOWN ACK` chunk it removes the state of the association and replies finally with a `SHUTDOWN COMPLETE` chunk which triggers upon reception at host B the clearance of state.

The second possibility is an ungraceful close (abort of an association). When one association endpoint wants to abort an association, it sends an `ABORT` chunk which must not be bundled with any `DATA` chunk. The `ABORT` chunk must contain the verification tag. The receiver of an `ABORT` chunk verifies the tag and after doing this successfully removes its state of the association.

Multi-homing SCTP provides the possibility that one association endpoint is represented by one or more destination transport addresses. A destination transport address consist of an IP-address and a port address whereas multiple transport addresses share the same port within multi-homing.

The SCTP path management function chooses the destination transport address for each outgoing SCTP packet based on the SCTP user's instructions and the currently perceived reachability status of the eligible destination set [15]. The monitoring to observe the reachability of the different addresses is called *heartbeat mechanism* which uses `HEARBEAT` chunks. This mechanism is facilitated if other packet traffic is inadequate to deliver the necessary information. At association setup the set of addresses is exchanged and a primary path is defined and afterwards used normally.

2.2 Protocol support negotiation

In order to know which transport protocols are supported by the other side, there are several possibilities to achieve this goal. It is not a trivial process, because the protocol has to be known before a connection is established and especially the costs for connection setup which can be determined by the number of needed *round-trip times* (RTTs) should be as low as possible. Moreover, we present two different approaches which address this problem and focus on keeping the costs in an acceptable range.

2.2.1 Cross-layer negotiation protocol

In principle the extension of the Internet, hence inventions of new technologies involves often introducing new layers. A key missing ingredient, however, is a mechanism to decide efficiently which protocols implementing which layers to use between a given pair of hosts [19]. For this reason an efficient method for *Cross-Layer Negotiation* [19] has been proposed. The negotiation of which protocol to use at which layer is done in a 3-way handshake by agreeing on a common supported protocol graph:

First, the initiator proposes its protocol graph which reflects all its supported or willing to support protocol combinations to the responder. In the second step the responder revises the proposed graph by pruning the unsupported or undesired protocol paths and sends it back to the initiator. The final step of the 3-way handshake is the acknowledgment of the final selected protocol path by the initiator to the responder.

The protocol design is a rough sketch and includes thoughts on *Negotiation Context*, *Encoding the Graph* and *Negotiation Message Transport*. The negotiation context must be well-defined. This means that the communication endpoints and the raw delivery channel are identified uniquely. For example if one uses UDP the negotiation context is defined uniquely by the 5-tuple (2 IP addresses, IP protocol and UDP port numbers).

The description of a node in the graph consists of a *Node ID*, a *Protocol ID* and a *Node Descriptor*. The node ID is fixed for one negotiation but is chosen arbitrary at the negotiation start. The protocol ID defines a specific protocol and the node descriptor contains of zero or more *Child Node ID*(s). A whole negotiation message is just a sequence of those node descriptors.

For negotiation message transport they consider that it might happen that a message exceeds the *Path Maximum Transport Unit* (PMTU). Therefore, they proposed to use the *chunk* segmentation as SCTP specified in RFC 4960 [15] does.

Evaluation

The cross-layer Negotiation Protocol does not only focus on solving the transport protocol negotiation problem, but also presents a suitable solution for negotiating whole protocol stacks. However, such an all-in-one solution may not provide the best performance for special applications like the transport protocol negotiation. Another point of criticism is that the chosen protocols by the endpoints need not to be supported along the entire path. Therefore, an additional check is needed for ensuring this fact.

2.2.2 Happy Eyeballs

Happy Eyeballs proposes a protocol negotiation in context of HTTP for the protocols IPv4 and IPv6 as well as TCP and SCTP. During their decision procedure they utilize two variables: `SWAIT` and `PREF`. `SWAIT` specifies the application-wide wait time for an SCTP association attempt to complete [20] whereas `PREF` defines a possible preference (possible values are `TCP`, `SCTP` and `BOTH`). The HTTP client starts one or more threads according to the preference:

- **Thread 1** attempts to connect using SCTP (started for `BOTH` and `SCTP`).
- **Thread 2** attempts to connect using SCTP over UDP (started for `BOTH` and `SCTP` when using IPv4).
- **Thread 3** attempts to connect using TCP (started for `BOTH` and `TCP`).

If an SCTP association attempt was made by a thread, the HTTP client waits for at least K ms, where $K = max($`SWAIT`, time taken for the TCP connection to complete) [20]. If the SCTP association is established during the time K, then this one is used and the TCP connection is abandoned. After this procedure the per-destination preference `PREF` is set to the currently used protocol.

Evaluation

Happy Eyeballs suggests a procedure for protocol negotiation for transport protocols, but this procedure is not explained for a general usage scenario. Although the focus lies on HTTP, the general idea may be adapted for a general transport layer protocol negotiation. This idea comes with a significant overhead since multiple transport protocols are tried to be instantiated at the same time. Therefore, we agree with the opinion of Bryan Ford; Speculatively opening alternative connections in parallel wastes host and network resources, and these costs compound with additional alternatives [19].

Chapter 3

Investigation

This chapter investigates "higher-level", QoS, existing and possible future Internet transport APIs as a starting point for the design of the protocol-independent API. After the evaluation of the different transport APIs, the most popular one – the socket API – with all its extensions has been chosen for creating the new protocol-independent API. The new protocol-independent socket API does not concentrate on specific protocols but rather tries to facilitate using the services provided by currently available transport protocols.

3.1 Higher-level transport APIs

Two higher-level transport APIs are going to be discussed in the following section. Although these approaches do not exactly match the motivation mentioned in Section 1.2, a closer look has been taken at them for the sake of completeness and to draw some conclusions for the design of a new API.

3.1.1 BEEP

The *Blocks Extensible Exchange Protocol* (BEEP) is defined in [21] as; "a network application framework protocol that enables network designers to spend more time in their own protocols rather than solving over and over again the same problem". BEEP gives a network application developer the possibility to design her own application protocol. It is using MIME which normally is provided in a textual form in XML. Therefore, it is well-suited for applications which have a need for an application protocol. One could easily realise HTTP, SMTP and many more application protocols with it.

However, despite its advantages, BEEP is not used much. One reason for this might be its additional protocol overhead. Since an efficient method for accessing the services provided by the transport layer should be developed, this protocol is not suitable, because it adds a kind of session instantiation which may not be practical for all types of network application programs. Thus BEEP is not able to fulfill the requirements for building a protocol-independent Internet transport API on top of it.

3.1.2 ACE

The *ADAPTIVE Communication Environment* (ACE) is an object-oriented (OO) toolkit that implements fundamental design patterns for communication software [22]. ACE has been designed in a way that the code, which is written for it, is portable to different operating systems and machines. It does not only specify common data types and methods for network applications, but is also applicable in the areas of interprocess communication, thread management and memory management. ACE utilizes several design patterns and tries to keep the programming effort as low as possible.

ACE provides a set of *Connector* and *Acceptor* components that decouple the active and passive initialization roles [22]. It tries to give a common access mode to non-object-oriented transport APIs by using abstraction and inheritance with common design patterns like the facade, wrapper and factory pattern.

ACE identified the following essential design goals by having the problems, mentioned underneath, in mind [22]:

- shield applications from error-prone details,
- combine several operations to form a single operation,
- parameterize IPC mechanisms into applications,
- enhance code sharing.

The structure of ACE is not explained in detail, because this is beyond the scope of this work and is not relevant. Therefore, only problems which have been identified with the socket API and the *X/Open Transport Interface* (XTI) are mentioned [23]:

- **Lack of type-safety**: The I/O handles are not strongly typed and cannot be checked at compile time.
- **Steep learning curve**: The functions have quite complex semantics. Especially the fact that multiple protocol families (Internet, Unix, OSI,...) and infrequently used features (broad/multicasting, asynchronous I/O, urgent data delivery,...) are supported makes the beginning of network programming quite complicated.
- **Poorly structured**: The steep learning curve is also caused by the poor structure of existing APIs. The structure is linear, and there are a lot of

different methods for sending/receiving data (shown for the Socket API in Section 3.3.3).

- **Portability**: The existence of multiple standards like the socket API and TLI/XTI makes portability hard. There are also problems when porting a socket-based UNIX application to Windows, because the Winsock API has some slight differences in header files, error numbers, shutdown semantics and socket options.

Evaluation

This high-level approach limits its usage to object-oriented programs written in C++ and is therefore not considered as a low level library which can be used by other programming languages. However, ACE documentation outlines the problems with the socket API and other transport APIs, and some of them will be addressed as long as this is possible with the low level API that should be introduced. In addition to the high-level difficulty there are two more problems: ACE cannot provide tuning of the options of different transport protocols for the user and support of new transport protocols (SCTP and DCCP) is missing. Hence, the authors of ACE may have assumed that configurability of features is not very important because the used protocols (TCP and UDP) do not provide as many features as new protocols do (see Section 2.1).

3.2 QoS transport API proposals

There were some efforts for introducing service based APIs instead of the currently used protocol based APIs. All of them were influenced by the developement of Quality of Service (QoS) and tried to enable QoS towards the end user. Hereafter we present only one representative approach which tried to introduce end-to-end QoS as an extension to the existing socket API, although several different approaches exist [24], [25], [26], [27], and [28]. The second project focuses much more on a service-oriented view of the API than on enabling end-to-end QoS.

3.2.1 QSockets

The QSocket API enables applications to easily create and manage connections with QoS requirements [29] by extending the socket API. The main focus of the API is to enable applications to use end-to-end QoS in an easy way. The authors of QSockets considered that extending the well-known socket API by associating a socket with QoS information would be sufficient to facilitate end-to-end QoS. The new functionality should be provided with a minimum amount of modifications to existing applications.

The usage of scheduling functionality which is already available in GNU/Linux is essential for this API and allows to place application specific QoS modifications of packets at later points in the network path (e.g. the queue before the packet is passed to the network interface). Since we do not want to enforce QoS towards the end-user, we do not take a closer look at the traffic controller.

For realizing and specifying the QoS requirements, a new QSocket library has been built which operates quite similar to the socket API. The transport user uses a transparent library interface, but in kernel space the calls are multiplexed on two different modules. The first one passes normal connect/send/receive calls onto the inner socket layer and the other one is responsible for taking QoS specifications and parameterize the aforementioned Linux traffic controller.

Although the QSocket promised to make the new functionality available with as little as possible effort, the application examples mentioned in [29] seem rather complex. The QSocket library needs several new functions: `qsockinit()`, `qattach()`, `qchange()` and `qinfo()`. However, it has been shown that these function calls do not add significant overhead for the transport user.

Summary and evaluation

QSockets have their main focus on introducing end-to-end QoS and therefore do not consider protocol independence as necessary. Nowadays, QSockets must be filed under one of the end-to-end QoS approaches which never succeeded in getting deployed. The high-level abstraction of QoS specifications and the needed amount of effort to introduce QoS applications are probably the two most important reasons.

3.2.2 Dynamic application oriented network services (DANCE)

The DANCE project introduces a model which provides a service-oriented view to a communication subsystem [30] and tries to remove the dependence on transport protocols by introducing an abstraction. The approach for a protocol independent transport API, which was part of the DANCE project, connects several ideas. In the following section the ideas for the unified, protocol-independent API for connection-oriented and connection-less protocols [31] will be explained.

Names instead of numbers

Currently, name resolution is provided by DNS and must be done by the transport user itself. The usage of DNS helps transport users to solve several problems like portability of services to new hosts, load sharing and mobility. Therefore the approach presented by DANCE is to move the name resolution behind the API, just like in [32] (discussed in Section 3.4.2). DANCE promises additional advantages from doing so, because also technologies like IPv6 and ATM for example could then be deployed in an easy fashion. This can be achieved because the network application has not to deal with any address structures anymore. Moreover, DANCE also wants to offer the possibility to specify the services via "service names" instead of port numbers.

Unifying access to sockets

Currently the socket API distinguishes between the so-called connection-oriented protocols and connection-less protocols (explained in detail in Section 3.3.3). However the goal of a unified, protocol independent transport API can only be reached if this difference is removed. DANCE solves this problem by providing three kinds of entities: *Listener*, *Flows* and the *ServiceProvider*. A listener waits for incoming requests, and when a request is received creates a flow with the help of the service provider. The communication initiator uses the service provider to directly create a flow and connect to the listener.

Although DANCE wants to unify the access to connection-oriented and connection-less protocols, it provides a simpler possibility to send a single datagram. This possibility damages the unification and does not even allow a better performance, because the complex procedure (described above) is hidden from the transport user, but nevertheless performed in the background.

Specification of communication services

For the specification of communication service requests the properties are divided into two groups. They are called *inherent* and *qualitative* properties. Inherent properties specify a fixed needed behavior of the transport provider (e.g. reliable transport, secure transmission), whereas qualitative properties describe desired but optional service properties [30], like loss rate, quality of decryption and delay.

Summary and evaluation

The DANCE project introduces a service-oriented and hence protocol-independent view towards a new transport API. The analysis of transport properties has not been carried out very thoroughly and therefore it is described very vaguely by the qualitative properties. There is no link between properties mentioned above and current transport protocols. Contrary to our approach DANCE was more oriented towards designing what would be interesting to have in a service-oriented API instead of analysing the existing transport protocols and their features.

3.3 Current transport APIs

Currently, there are a few different standardized APIs for the transport layer, namely: *Socket*, *Winsock* and *X/Open Transport Interface* (XTI), and newer ones such as *Sockets Extensions for SCTP*. All of them have their own history and evolution and were also influenced by different transport layer protocols. Although they are very closely related, we will have a look at their development and also at their current state and usage. The mentioned APIs are categorized into two parts; there is the XTI API on the one hand and the socket API with all its related APIs on the other hand.

3.3.1 Evolution of transport APIs

In 1983, the socket interface was introduced by the University of California, Berkley, but the principle idea of sockets had already been standardized in RFC 147 [33] by the IETF in 1971. Therefore the socket interface is often referenced as *Berkley socket* or *Internet socket*. After the socket API has become the standard API for facilitating the TCP/IP stack, Microsoft also decided to develop a socket-like interface called *Winsock*. The inventors of Winsock used the Berkley socket as a base and added features like "non-blocking" and "asynchronous" sockets. The decision for sticking to a similar API was also enforced by the fact that migration of programs will be easier in this way.

The evolution, mentioned above, is closely related to the TCP/IP stack, but the development of the ISO/OSI architecture also led to a new API. In the beginning, the name was *Transport Layer Interface* (TLI), but after its standardization by the OpenGroup it has been called XTI. It also provides the use of a TCP/IP stack, although its initial design was built in general for the ISO/OSI model. Nevertheless, the two APIs (XTI and sockets) are not interchangeable [34], but they may communicate with each other due to the standardization of transport protocols.

The newest API is the sockets extensions API for SCTP, which is necessary since the ordinary socket API would not be able to support all the features of SCTP. The extensions to the standard socket API are described in an Internet-

Draft [2], but have not been standardized by the OpenGroup or POSIX yet. Now the socket API also supports DCCP on some operating systems, but this API is very closely related to the TCP style socket API and has not yet been proposed for standardization.

3.3.2 XTI API

The XTI specification defines an independent transport-service interface that allows multiple users to communicate at the transport level of the OSI reference model. While XTI gives transport users considerable independence from the underlying transport provider, the differences between providers are not entirely hidden [35].

In order to understand the API and its functionality only the core part of XTI is explained in the following. For a detailed description please refer to the technical standard of *Network Services Issue 5* of the OpenGroup available at [35].

Terms

The following terms, which might not be intuitive, are used within the XTI specification and therefore explained here.

Transport endpoint Locally a *transport endpoint* is identified by an integer called file descriptor (`fd`) and specifies the communication path between a transport user and a transport provider. A transport endpoint is created by the transport user with the `t_open()` function by specifying the transport provider. In addition to the created file descriptor the `t_open()` function also returns protocol specific information.

Initiator The transport user which actively initiates the conversation.

Responder The transport user which is passively waiting for another transport user to request a connection.

Connection-oriented mode

This mode consist of four phases of communication [35]:

- Initialisation/De-initialisation
- Connection establishment

- Data transfer
- Connection release

In the *Initialisation/De-initialisation* phase the transport endpoint is created and afterwards bound to a specific transport address (normally called port) with the `t_bind()` function.

During the *connection establishment* phase one has to distinguish between the initiator and the responder of a connection. The initiator calls the `t_connect()` function for establishing a connection. This may be done in a synchronous or asynchronous manner, but when the asynchronous way is chosen the transport user has to call the `t_rcvconnect()` function for retrieving the status of the connect request. On the other side, the responder has to listen to the bound address by the `t_listen()` function after its initialisation phase.

The *data transfer* phase consists of two functions: `t_snd()` and `t_rcv()`. Both functions can send/receive normal and expedited data. These functions may also be used in an asynchronous manner.

The *connection release* phase of XTI normally supports only an abortive release, but due to the fact that some transport protocols (e.g. TCP) also support an orderly release, many XTI implementations also provide this release possibility.

Connection-less mode

The connection-less mode only needs two phases: *Initialisation/De-initialisation* and *data transfer*. After initialization of the transport endpoints, data can be sent/received immediately with the functions `t_sndudata()` and `t_rcvudata()`. These functions may be utilized in a synchronous as well as in an asynchronous way. In addition to these two functions the function `t_recvuderr()` gives the opportunity to retrieve error information associated with a previously sent data unit.

States and events

The transport endpoint can be in different states, and, depending on the current state, various events can occur and the allowable sequence of functions can be determined. Table 3.1 shows the possible states of a transport interface. A transport interface used in connection-less mode will only have transitions within the first three states (`T_UNINIT`, `T_UNBND` and `T_IDLE`). A connection-oriented transport interface will have transitions within all states depending on whether it supports orderly release or not. In the latter case the last two states are not included.

State	Description
T_UNINIT	uninitialised - initial and final state
T_UNBND	unbound
T_IDLE	no connection established
T_OUTCON	outgoing connection pending for active users
T_INCON	incoming connection pending for passive users
T_DATAXFER	data transfer
T_OUTREL	outgoing orderly release
T_INREL	incoming orderly release

Table 3.1: Transport interface states [35].

The events are divided into two subclasses: incoming and outgoing events. The outgoing events correspond to returns (successful or error) from the outgoing user-level transport functions (e.g. `t_connect()` and `t_send()`). The incoming events respond to the incoming user-level transport functions, like `t_listen()` and `t_rcv()`. The full transition table of XTI states is shown in [35].

Option management

There are general XTI options and those that are specific for each transport provider. Some options allow the user to tailor her communication needs for instance by asking for high throughput or low delay. Others allow the fine-tuning of the protocol behavior so that communication with unusual characteristics can be handled more effectively. Other options are for debugging purposes or to request that certain operations should be performed [35].

A differentiation is done between options that have an end-to-end significance (they need to be negotiated with the opposite transport endpoint) and those which are only used by the local transport endpoint.

Notes on portability of XTI

Since the options in XTI are often tightly coupled to the transport protocols making use of them harms the portability of network applications using XTI. Another aspect for portability is the possibility that different system platforms (even operating systems from different vendors) may implement only a subset of options which suits their needs.

Evaluation of XTI

Although XTI seems to be well prepared to support our needs the missing popularity is a lack which can not be overruled by its clear design and greater sake for transport protocol independence. The missing popularity can be seen by the fact that XTI was a mandatory part of the *Network Services Issue 4* and *Network Services Issue 5*, but already in version *Network Services Issue 5.2* it is considered as optional. The *Single Unix System Specification 3* does not contain XTI either.

The attempt to create a transport protocol-independent API was maybe not successful due to too many restrictions for the options. The general XTI options of course must be supported by any implementation, but since these are only a few, a portable XTI network application cannot use any specific transport protocol features.

3.3.3 Socket API

The socket API does not only provide access to functionality of the transport layer, but also allows the direct use of the IP layer. Since we focus on transport APIs we are not having a closer look at non-transport relevant socket details.

The socket API consists of two standard procedures. There is one for the so-called "server" and there is another one for the "client". The server provides a service by listening to a specific port and the client requests the offered service from the server by connecting to it. There are several differences in the send/receive procedures for each supported transport protocol. Therefore, we are going into detail for each transport protocol separately, but leave some common aspects (socket options, socket instantiation, connection-oriented vs connection-less and the comparison of I/O functions) out for a joint description at the end.

TCP socket

First we describe the standard procedure when using a TCP socket which is shown in Figure 3.1. After creating a socket with the `socket()` method (described underneath) the server has to bind its newly instantiated socket to a well-known port (`bind()`), whereas the client connects to this port with the `connect()` method. The server has to complete its initialization phase first by getting its newly created socket from the `CLOSE` state to the `LISTEN` state, which is done with the `listen()` method, and afterwards wait for new connections with the `accept()` method. The `accept()` method blocks until a client tries to connect to the server. After accepting a new connection, which is the first part of establishing a TCP connection, the server completes the TCP three-way handshake with the client.

Then the client and the server communicate with their application specific defined protocol through several `read()` and `write()` method calls. After they

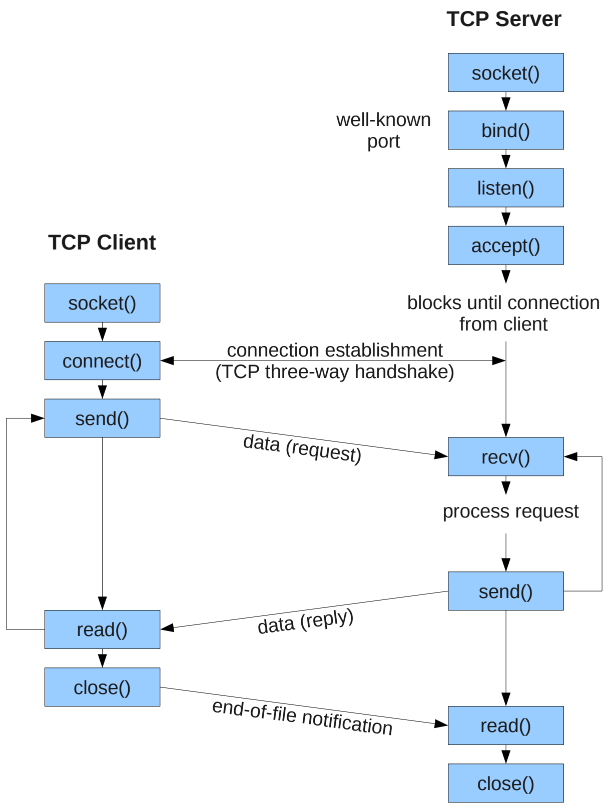

Figure 3.1: Socket functions for elementary TCP client/server [34].

have finished their communication they use the `close()` method for terminating the connection. At this point it is important to know that TCP sockets also support the so-called *half-closed connections* when only one of both closes its side of communication. This side is then not able to send data anymore, but may still receive data from the other one.

UDP socket

For using a UDP socket, both the server and client have to create a socket with the `socket()` method. As with TCP, the UDP server has to bind to a well-known port with `bind()`, but afterwards it uses the `recvfrom()` method to block until it receives a datagram from the client. The socket functions for an elementary UDP communication are shown in Figure 3.2.

After the creation of the socket the client has to call the `sendto()` method for signaling the server a data request in an application specific format. The UDP server replies to a request with a `sendto()` and afterwards blocks again

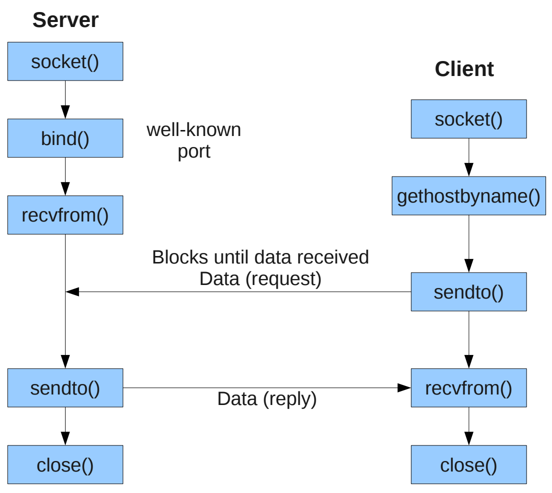

Figure 3.2: Socket functions for elementary UDP client/server [34].

using the `recvfrom()` method awaiting a new request. The UDP client normally closes the socket by calling `close()` after it received the reply from the server in the blocking method `recvfrom()`.

Connected UDP sockets can be created by calling the `connect()` function after the socket creation. There is no real connection establishment but rather a simple reachability check of the requested host. A connected UDP socket normally does not use `sendto()` and `recvfrom()` and makes use of `write()` and `read()` instead, because the destination has already been explicitly specified during "connection setup".

SCTP socket

The sockets API extensions [2] for SCTP define a mapping of SCTP to the well-known sockets API. Its operation is divided into two different parts. The first one is called *one-to-one style* and the second one *one-to-many style*.

One-to-one style The one-to-one style was developed to alleviate the porting of existing TCP applications to SCTP [34]. Therefore, there are no appreciable differences in use of the one-to-one style interface and the standard TCP socket. The use of `read()` and `write()` is replaced by `recv()` and `send()`, but may, for portability reasons, also use the former ones. The one-to-one style supports the same stream-based behavior as TCP does, although SCTP itself works in a message-oriented way.

One-to-many style The one-to-many style interface differs from the normal TCP socket as well as from the UDP socket. It makes use of the connection-oriented communication as TCP does, but communicates in a message-based way rather than using the stream-based way which TCP uses. The benefit of using the one-to-many style socket is the possibility for one SCTP socket to receive messages from multiple clients in a similar way as UDP sockets.

A typical server using a one-to-many style socket will do a `socket()` call, followed by a `listen()` and `recvfrom()` or `recvmsg()`. A typical client will just use `sendto()` or `sendmsg()` to the server of its choice. Note that the `connect()` and `accept()` calls are not needed, although the `connect()` call may be used for initiating an association without sending data. Another difference is that the data transfer methods mentioned above need to specify which association should be used, because this socket type supports more than one SCTP association at a time.

Comparison of one-to-one and one-to-many Randall R. Stewart, one of the SCTP developers, wrote in reply to the question which feature of SCTP is not supported by the one-to-one style socket [36]:

> The big issue (at least I know in KAME-BSD) is that a one-to-one style socket is not allowed to send data until the connect() or accept() are completed. This translates into not being able to piggyback data on the third or fourth leg of the 4-way handshake. The one-to-many uses implicit connection setup so that when you do a send(data, new-addr); it will start the connection setup and piggyback the data with the cookie-echo.

Nowadays, this cannot be considered true any longer, because several implementations of the sockets extensions for SCTP do implement an implicit connection setup on a `sendmsg()` on a non-connected one-to-one style socket (e.g. Linux Kernel implementation of SCTP [37]). One of the reasons why both styles will remain in the future is the fact that the Internet-Draft [2] which specifies the socket extensions for SCTP does not explicitly require the behavior mentioned above.

DCCP socket

Although the socket API for DCCP is already implemented in the Linux Kernel as a loadable module no standardization for using and implementing the DCCP socket API currently exist.

The currently implemented DCCP socket in the Linux Kernel used the TCP socket as implementation model. Its authors identified the code in the TCP implementation that could be made generic and shared with other transport level implementations [38]. The shared parts are called "minisockets". The usage is more or less the same as for a TCP socket; the implementation of DCCP is still in an experimental stage and may still change a bit. For these two reasons we do not explain further details on DCCP sockets here.

Socket options

The functions `setsockopt()` and `getsockopt()` are the functions in the socket API for manipulating and reading socket options of the following categories:

- generic
- IPv4
- IPv6
- TCP
- SCTP
- DCCP

Due to the fact that we are only interested in transport layer sockets we do not mention any further details about IPv4 and IPv6 socket options. In the list above, it can also be seen that UDP sockets do not need any special socket options and only some of the generic options are supported by it.

Generic socket options In Table 3.2 we mention only generic socket options which have an influence on the features offered by transport protocols. The explanation of the socket options is not examined here in detail, because there are a lot of special behaviors for the different protocols. The following description should only outline the concept and the given possibilities. For a detailed explanation one can refer to [34].

The generic socket options include also options which are not supported by each socket type. For example the `SO_BROADCAST` socket option is supported

Socket Option	Description	TCP	UDP	SCTP	DCCP
SO_BROADCAST	enables or disables the possibility to send broadcast messages.		x		
SO_KEEPALIVE	is used to send a keep-alive probe to the peer if for more than two hours no data has been exchanged.	x			
SO_LINGER	specifies how the `close()` works for connection-oriented protocols.	x		x	x
SO_RCVBUF	specifies the size of the receive buffer.	x	(x)	x	x
SO_SNDBUF	specifies the size of the send buffer.	x	x	x	x
SO_RCVTIMEO	specifies the timeout for receive functions.	x	x	x	x
SO_RCVTIMEO	specifies the timeout for send functions.	x	x	x	x
SO_REUSEADDR	specifies whether the address can be reused or not.	x	x	x	x
SO_REUSEPORT	specifies whether the port can be reused or not.	x	x	x	x

Table 3.2: Overview of generic socket options which influence the behavior of current transport protocols.

only by datagram sockets and only on networks which support the concept of a broadcast message (e.g., Ethernet, token ring, etc.) [34].

`SO_KEEPALIVE` is not supported by SCTP although it has a similar heartbeat-mechanism. The one of SCTP is configured via `SCTP_SET_PEER_ADDR_PARAMS`, because this mechanism is controlled for each address separately.

The receive buffer size influences the flow control of TCP, SCTP and DCCP. However, if the receive buffer of a UDP recipient does not have enough space for a new datagram, the datagram is discarded.

The two `SO_REUSE` options state that the address/port can be reused but in detail these options are quite complicated when these combinations of address/-port reuse are allowed.

TCP options Since a lot of TCP options are handled by general socket options only two remain as dedicated TCP options. `TCP_MAXSEG` allows to fetch or set the MSS of a TCP connection. The MSS is related to the path MTU; therefore, the MSS can change if path MTU discovery delivers a new value(e.g. due the usage of a new route).

`TCP_NODELAY` enables or disables the Nagle algorithm of TCP. The Nagle algorithm reduces the number of small packets by coalescing. A packet is small if it is smaller then the MSS. The algorithm waits before sending a small packet if there is an unacknowledged packet.

SCTP options SCTP itself is the most configurable of all currently available transport protocols. Therefore, it provides a lot of options, which also requires data to be passed to the kernel. The normal `setsockopt()` and `getsockopt()` functions normally allow to pass data into and out of the kernel. However, not all current socket implementations do so. Therefore, the SCTP API provides its own function `sctp_opt_info()` for this. The two previously mentioned TCP options also exist for SCTP.

Another interesting option is the `SCTP_DEFAULT_SEND_PARAM` which allows the user to specify different parameters which normally are specified on a per-message basis. For multipath support the primary address can be set using `SCTP_PRIMARY_ADDR`. There are many more options specified in the Internet-Draft [2] but these are not explained in detail in this thesis, because they are not relevant for the later investigated features of SCTP.

DCCP options Since there are currently no standardizations of the DCCP socket API, here we only mention the options which are in our interest. There are two features of DCCP which must be supported by socket options: The first one is the choice of the CCID which is done via `DCCP_SOCKOPT_CCID` for the sending and receiving half connection, `DCCP_SOCKOPT_TX_CCID` for the sending half connection and `DCCP_SOCKOPT_RX_CCID` for the receiving half connection. The second one specifies the coverage of the checksum by `DCCP_SOCKOPT_SEND_CSCOV` for creating the checksum and `DCCP_SOCKOPT_RECV_CSCOV` for verifying a received checksum.

Socket instantiation

The protocol decisions are made at the very beginning when a socket is instantiated by calling the `socket()` method. There are 3 parameters when creating a socket: *protocol family*, *type of socket* and the *protocol* itself.

Currently, there are a lot of different protocol families but since we are focusing on a new transport layer API only the following two are relevant:

- `PF_INET`: protocol family for IPv4 protocols
- `PF_INET6`: protocol family for IPv6 protocols

The socket API including the extensions for SCTP currently supports the following combinations of protocol families, types and protocols which are shown in Table 3.3.

In Table 3.3 it can be seen that there are no differences between IPv4 and IPv6 in the context of protocol support. Another aspect which worth pointing out is that SCTP is the only protocol which supports the delivery of sequential

	PF_INET	PF_INET6
SOCK_STREAM	TCP/SCTP	TCP/SCTP
SOCK_DGRAM	UDP	UDP
SOCK_SEQPACKET	SCTP	SCTP
SOCK_RAW	IPv4	IPv6

Table 3.3: Supported combinations of protocol families, types and protocols [34].

packets (`SOCK_SEQPACKET`) so far. The delivery of sequential packets provides a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length [39]. The differences of `SOCK_SEQPACKET` and `SOCK_STREAM` are described in [39] as follows:

> The `SOCK_SEQPACKET` socket type is similar to the `SOCK_STREAM` type, and is also connection-oriented. The only difference between these types is that record boundaries are maintained using the `SOCK_SEQPACKET` type. A record can be sent using one or more output operations and received using one or more input operations, but a single operation never transfers parts of more than one record. Record boundaries are visible to the receiver via the `MSG_EOR` flag in the received message flags returned by the `recvmsg()` function. It is protocol-specific whether a maximum record size is imposed.

When a socket of type `SOCK_SEQPACKET` with the protocol `IPPROTO_SCTP` is created, this results in a one-to-many style socket. If the combination of `SOCK_STREAM` and `IPPROTO_SCTP` is used, the result is a one-to-one style socket.

The instantiation of the DCCP protocol is not mentioned in Table 3.3 because it does not fit in the used and standardized scheme. Currently, in the Linux Kernel implementation [38] the combination of `SOCK_DCCP` and `IPPROTO_DCCP` is suggested for instantiating a new DCCP socket.

Connection-oriented vs connection-less sockets

So far we have seen a lot of different socket APIs that allow an application developer to access diverse transport protocols. The most important differentiation of sockets is the one belonging to their connection mode. A socket can run either in connection-oriented mode like TCP, DCCP and SCTP sockets, or in connection-less mode like UDP and SCTP sockets. Although this can be seen as the primary distinction between connection-oriented and connection-less, this is no hard boundary, because also a UDP socket, which is normally considered as connection-less, may be transformed into a connected one (already mentioned in

I/O function	connection-oriented	connection-less
`read`	not advised	not advised
`write`	not advised	not advised
`readv`	not advised	not advised
`writev`	not advised	not advised
`recv`	ok	normally not used
`send`	ok	normally not used
`recvfrom`	normally not used	ok
`sendto`	destination ignored	ok
`recvmsg`	ok	ok
`sendmsg`	destination ignored	ok

Table 3.4: Overview of I/O functions for connection-oriented and connection-less sockets.

the description of the UDP socket section). Another aspect which confirms the soft boundaries is that the SCTP socket can be both. Additionally a connection-less SCTP socket can be peeled off by the `sctp_peeloff()` function and turned into a connection-oriented SCTP socket.

Comparison of I/O functions

Before having a closer look at the real I/O functions (send/receive) we are looking at the `connect()` function which is not mandatory for all socket types since not all underlying transport protocols are connection-oriented (although also the connection-less protocols like UDP can make use of the `connect()` function for a rudimentary check of reachability of a server). For both SCTP socket styles the connect is optional. This means that, if a message is sent to a not already established association, the initialization is done automatically. Note that there are some limitations for the one-to-one style interface.

As already mentioned earlier there are more than just `read()` and `write()` functions (which are inherited from standard I/O mechanisms): `recv()` and `send()`, which allow a fourth argument that contains flags from the process to the kernel, `readv()` and `writev()` which lets a user specify a vector of buffers to read from or write to and `recvmsg()` and `sendmsg()`, which combine all the features of the other I/O functions along with the new capability of receiving and sending ancillary data [34].

The two I/O functions `recvmsg()` and `sendmsg()` are the most general ones and indeed all calls of `read()`, `readv()`, `recv()` and `recvfrom()` could be replaced by calls to `recvmsg()`. Similarly all calls to the various output functions could be replaced with calls to `sendmsg()` [34].

Table 3.4 shows a detailed overview of the supported I/O functions of connection-oriented and connection-less sockets. In general, all I/O functions may be used by both types but the usage is normally suggested for only one of them. On the one hand, the `recv()` and `send()` are normally used by connected sockets only, because a socket in connection-less mode does not have a specific address set (except if it does this implicitly). On the other hand, the `recvfrom()` and `sendto()` functions are utilized by the connection-less sockets. If the `sendto()` function is called on a connection-oriented socket the destination address in the call will be ignored. The same happens for a `sendmsg()` call. The `read()` and `write()` functions should not be utilized because they are only inherited from the standard I/O functions of file descriptors.

Blocking vs non-blocking mode

The socket API supports blocking as well as non-blocking I/O. The user has the possibility to switch to the non-blocking I/O mode via the `O_NONBLOCK` flag using the function `fctnl()`. This leads in general to a behavior where all operations that would block will return with `EAGAIN`.

The specification of the socket API states the following for send and recv mechanisms belonging the mode [40]:

> If space is not available at the sending socket to hold the message to be transmitted, and the socket file descriptor does not have `O_NONBLOCK` set `send()` shall block until space is available. If space is not available at the sending socket to hold the message to be transmitted, and the socket file descriptor does have `O_NONBLOCK` set, `send()` shall fail. The `select()` and `poll()` functions can be used to determine when it is possible to send more data. If no messages are available at the socket and `O_NONBLOCK` is not set on the socket's file descriptor, `recv()` shall block until a message arrives. If no messages are available at the socket and `O_NONBLOCK` is set on the socket's file descriptor, `recv()` shall fail and set `errno` to `[EAGAIN]` or `[EWOULDBLOCK]`.

Evaluation and summary

Nowadays, the socket API is the most popular and best-known transport API. However, throughout its evolution it has been modified and influenced by many protocols, not only the Internet protocols. This has led to quite a complex usage of it, where a network application developer has to use the different features carefully. Additionally one always has to bear in mind that functions are possibly interpreted differently for each protocol.

Especially the support of SCTP by the socket API has resulted in a complex design, because the API has not been designed to handle protocols that may be utilized in the many different ways that are possible with SCTP. As we have seen, the distinction of one-to-one and one-to-many style sockets is unnecessary from a programmer's point of view, because the same functionality can be exploited by both. Another negative aspect of the socket API is the lack of standardization for DCCP, which therefore is currently not consistent for different implementations. This seems to be an issue for other socket types as well (especially SCTP).

The problems especially with new transport protocols clearly show that there is a need for a protocol-independent Internet transport API, which stays close to the semantic of the current socket API, but allows to access new features in an easier way and with less necessary knowledge about the API and transport protocols.

3.4 Proposed future transport APIs

3.4.1 Structured streams

Structured Stream Transport (SST) enhances the traditional stream abstraction with a hierarchical hereditary structure, allowing applications to create lightweight *child streams* from any existing stream [41]. Structured streams have been proposed for solving several problems which exist nowadays when using the transport APIs:

- The delays of 3 and 4-way handshakes for TCP, SCTP and DCCP and the TIME-WAIT period on close can be omitted by creating child streams instead of new connections.
- The usage of structured streams provides the possibility to prioritize in a dynamic way through out-of-band signaling.
- The "large datagram problem" which states that the loss probability of a datagram increases exponentially with the fragment count, can be solved by unifying datagrams and streams semantically.

Design and benefits of SST for transport users

SST extends the socket API by the following three new operations for creating substreams: An application first calls `listen_substream()` which prepares

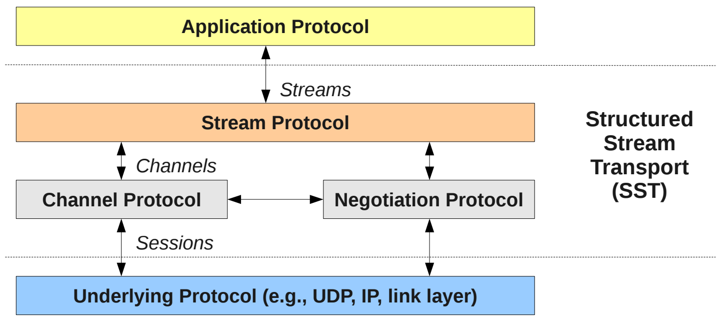

Figure 3.3: The SST protocol architecture [41].

the socket to be a server socket and finally waits with `accpet_substream()` for an incoming request for a substream. The client side needs to call `create_substream()` to create a new substream. The semantics of these three operations are very close to the *listen, accept* and *connect* procedure utilized in the socket API explained in Section 3.3.3.

For sending datagrams SST provides a `send_datagram()` operation that creates a temporary child stream, sends the data on it and closes it immediately afterwards. The operation `receive_datagram()` first waits for accepting a shorthand substream, receives the data and also closes the substream. Since the datagram utilizes just an ephemeral substream, `receive_datagram()` can also be realised by an `accept_substream()` and therefore SST claims that datagrams and streams have been semantically unified. SST can use an efficient and stateless delivery as UDP would, but the receiving application does not note anything about this. Therefore, SST is free to use this optimization only under appropriate network conditions and hence solve the "large datagram problem".

Design of SST protocol suite

SST consists of three dependent protocols shown in Figure 3.3. The *negotiation protocol* has already been described in Section 2.2 as cross-layer negotiation protocol. The *channel protocol* is a connection-oriented best-effort delivery service [41] that provides services which are shared for the same pair of hosts. The services are packet sequencing, integrity and privacy protection, selective acknowledgment and congestion control [41]. In general all streams between the same pair of hosts are mapped to one channel at any point in time. The multiplexing of several streams onto one channel is done by the *stream protocol*.

Summary and evaluation

SST tries to solve many problems (listed above) at a time. This fact has led to a quite complex design consisting of three different protocols operating on two different levels of abstraction (streams and channels). SST proclaims itself more as a complete solution for a new transport layer rather than using existing transport protocols (TCP, SCTP and DCCP) and giving the possibility to access their features in an easy fashionable way. Therefore, SST is not suitable as a new transport API in the light of future concerns of the IETF, because it should support existing transport protocols. However, it provides some good concepts like the negotiation protocol which can be used in a new transport API.

3.4.2 Name-based sockets

The name-based sockets focus on solving the problems which arise when using host mobility, multi-homing, firewalls and *network address translators* (NAT). A name-based evolution of the existing sockets interface is proposed to relieve applications from IP address management responsibilities [32].

For their design, the authors of name-based sockets discovered three additional requirements [32]:

- **Backwards compatibility**: It must be possible for legacy applications to communicate without using the name-based sockets interface.
- **Security**: It must be ensured that domain names cannot be faked. The authentication of the initiator's domain name must be enabled.
- **Deployment incentives**: The adoption of the name-based sockets interface should provide a guaranteed benefit, at acceptable costs for all stakeholders.

Design and benefits of name-based sockets for transport users

The new API needs new procedures for initiating and receiving communication sessions. Another design decision is that the resuage of a socket (like it can be done for UDP sockets, explained in Section 3.3.3) is omitted, because session-specific state is required in the case of name-based socket. This results in a clear one-to-one coupling between one socket and a communication session and therefore unifies the socket API across transport protocols. The authors propose the use of the following six functions which are very close to the semantics of a TCP socket:

- `listen()` creates a socket for incoming sessions and makes it reachable by instructing firewalls and NATs. It has the two optional arguments, domain name and port which specify how this socket can be reached.

- `accept()` waits for incoming sessions and upon receiving one creates a new socket and returns it with the remote domain name which has been accepted.

- `open()` creates a socket and initiates an outgoing session to a remote host by giving the remote domain name and port as well as specifying the local domain name (from which the session is initiated) and a transport type (either "stream-oriented" or "datagram-oriented").

- `write()` sends the given data to the specified socket.

- `read()` reads data from the specified socket and returns the received data.

- `close()` closes the session and removes any session specific state.

Another important feature in the design is security. Especially the domain names must not be faked. Therefore, impersonation is prevented by a forward lookup in the DNS. If the retrieved IP address by the DNS is within the used set of IP addresses of the host then it can be assumed that the host is legitimate. Thus the security of name-based sockets relies on the security of DNS which is not really strong, but there are initiatives by the IETF to deploy secure DNS (DNSSEC) soon.

Summary and evaluation

The name-based socket API tries to raise the abstraction level of the socket API by using domain names instead of IP addresses and therefore enable mobility and multi-homing in a transparent manner towards the transport user. This approach gives a good opportunity to overcome future problems concerning the usage of multiple IP addresses during one communication session. Moreover, it gives a unified transport API over several transport protocols. However, the transparent choice of the transport protocol is determined by only one parameter (either "stream-oriented" or "datagram-oriented") which seems not be sufficient in the presence of 5 transport protocols with a variety of features. Therefore, no special features of them can be utilized and hence no advantage can be achieved.

3.5 Evaluation of transport APIs and future approaches

As a summary an overview of some important properties of the presented transport APIs in Table 3.5 is shown. The first row shows that the API is identical for protocols with different connection modes. It can be seen that only the two old-established APIs – socket and XTI – are different for connection-oriented and connection-less protocols. Another interesting fact is the level of programming languages which may be used to utilize an API. Of course there are a lot of high-level approaches. For example nearly every higher-level programming language has its own API for facilitating the socket API. Among the mentioned APIs only ACE does exclusively provide its functionality to higher-level languages. In addition the approaches by BEEP and DANCE seem to be more suitable for higher-level languages.

	ACE	BEEP	DANCE	Name-based Sockets	QSocket	Socket	SST	XTI
unifying connection-oriented and connection-less	x	x	x	x	x		x	
support of only high-level languages	x	(x)	(x)					
support of new transport protocols				(x)		x		(x)
configurability of transport features			(x)	(x)	(x)	x		(x)
support of QoS			x		x		(x)	

Table 3.5: Overview of Internet transport APIs (x-marks in brackets mean that the API partially supports this feature).

New transport protocols (SCTP and DCCP) are only supported by the socket API, but a standardization is still missing. The name-based socket API is maybe able to support new protocols, but this fact has not been confirmed yet. There is an XTI implementation available for SCTP but not for DCCP. The configurability of different transport features is also restricted by many transport APIs. The socket API facilitates full configurability of all features, although there are also some problems (mentioned in Section 3.3.3). DANCE and QSockets try to abstract from protocol features via a QoS requirement specification. The name-based socket API will probably support a similar access to features as the normal socket API if it is accordingly extended, and XTI allows to configure features, but not all transport protocols are supported. The last comparison criterion is the support of QoS which is clearly provided by the QoS oriented APIs DANCE and QSockets. Nevertheless, SST also offers some kind of QoS by assigning different priorites to streams with out-of-band signaling.

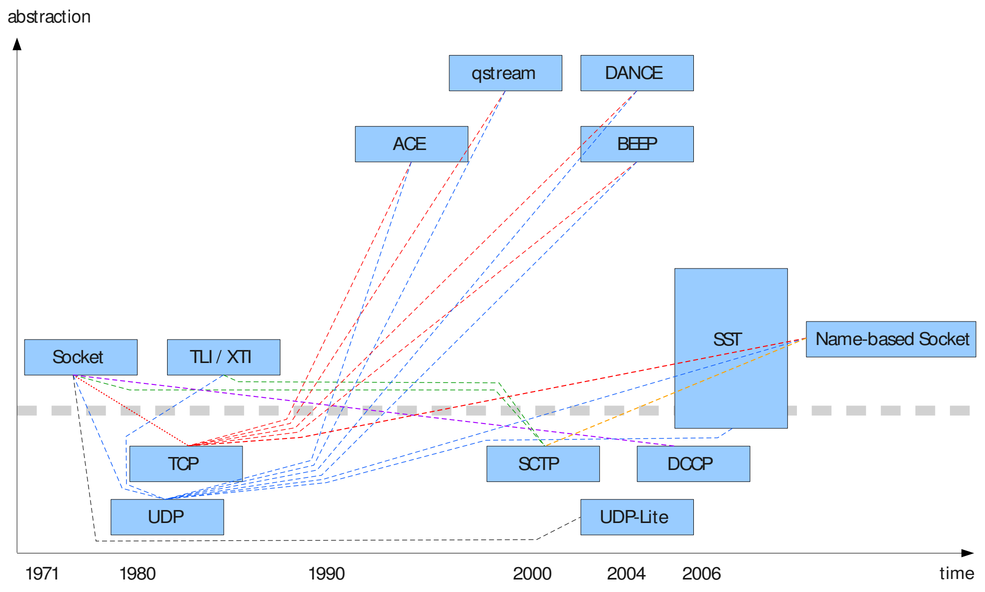

Figure 3.4: Overview of the history of transport APIs and protocols.

3.5.1 Old supports new, but new uses old?

Figure 3.4 shows a joint history of the transport protocols described in Section 2.1 and the transport APIs described in this chapter. The dashed lines from an API to a protocol show the support of the protocol by the API. It can be seen that something strange has happend: the old transport APIs like Sockets and XTI, but especially the socket API, have been adapted to support new protocols like SCTP and DCCP. However, the new transport APIs tend to only use older transport protocols. Especially the APIs which offer a higher degree of abstraction rely on older protocols. Therefore, the need of a new transport API which supports new protocols and is a bit abstracter than the existing well-established socket API, is clear.

3.5.2 Final conclusion of the investigation

Although there are some troubles with the socket API especially for the support of new transport protocols, it is still the most utilized transport API. Therefore, the socket API is considered as a good starting point for designing the protocol-independent Internet transport API. The analysis of this section have been taken into account for designing the new API. An overview about the design process is given in Chapter 4, and finally the entire API is presented in Chapter 5.

Chapter 4

Design

The investigation of existing transport APIs showed in Section 3.3 that the obvious starting point for a new API should be the well-known socket API. Before a possible implementation of the new transport API is described in Chapter 5, we are going to identify the transport services in detail. This identification and classification is divided into two steps: The first one is an analysis of current transport protocols and their features which leads to a huge number of available transport services. In the second step their number is reduced by simplifying them in a reasonable manner. In addition, some implementation hints and restrictions as well as two possible future transport services are mentioned.

4.1 Step 1: Analysis of transport services

In Chapter 2 the basic functionalities of currently available transport protocols have already been described. For a general description and protocol specific implementation details of features please refer to the according sections (TCP 2.1.1, UDP 2.1.2, UDP-Lite 2.1.3, DCCP 2.1.4 and SCTP 2.1.5) in the mentioned chapter. In the following section we are going to give a condensed overview first, followed by a detailed list of all available services of the transport protocols.

4.1.1 Feature overview transport services

Table 4.1 summarizes the services offered by currently available transport protocols standardized by the IETF. Since we concentrate on the Internet protocol stack, others such as the *Xpress Transport Protocol* (XTP) are not considered here because they have not been standardized by the IETF. The *Reliable Datagram Protocol* (RDP) is also kept out of the analysis because its specification in RFC 908 [42] and RFC 1151 [43] is experimental and it has never been in widespread use.

The main source for evaluating the transport services is [44] and has been extended for DCCP by evaluating its services with RFC 4340 [9]. In the following

transport protocol	connection-oriented	flow control	congestion control	app. PDU bundling	error detection	reliability	delivery type	delivery order	multi streaming	multi homing
TCP	x	x	x	0/1	x	t	s	o		
UDP					x		m	u		
UDP-Lite					x/p1		m	u		
DCCP	x	x1/x2	2/3/4		x/p1		m	u		
SCTP	x	x	x	0/1	x	t/p2	m	o/u	0/1	0/1

Table 4.1: Feature overview of currently available transport protocols.

two lists we are going to explain which of the features mentioned in [44] are considered as useful services towards a transport user and which are not.

Included transport features

The following features are taken into account as services and are indicated by the mentioned abbrevations in the tables of this chapter:

- **Connection oriented**: Although the connection-oriented protocols have different connection establishment and teardown procedures, they share "connection orientation" as a feature. Hence the service "connection-oriented" is either provided (indicated by an **x**) or not (always indicated by an empty cell).
- **Flow control**: The protocols TCP and SCTP simply support flow control by the "receiver window" but also DCCP offers a flow control: the slow receiver option, the usage of the data dropped (drop code 2) option and the additional limited sending rate when CCID 4 is used.
- **Congestion control**: SCTP and especially TCP can support different congestion control procedures (selective acknowledgment, ECN,...), but all of them follow the same goal: Transfering as much data as possible in as short a time as possible [10] with regard to prevention of congestion collapse, and providing some degree of fairness [45]. This type of congestion control support is indicated by an **x**. DCCP has different means in congestion control. DCCP provides a service by allowing the end users to choose between a variety of congestion control identifiers. Therefore, we consider the currently available CCIDS **2**, **3** and **4** as services towards a transport user.
- **Application Protocol Data Unit (PDU) bundling**: The bundling of multiple messages (PDUs) is useful for reducing the transmission over-

head by headers at the cost of a possible higher latency. For the stream-oriented TCP protocol the Nagle algorithm provides this feature. SCTP implementations which currently follow the Internet-Draft "Sockets API Extensions for Stream Control Transmission Protocol" [2] support the bundling via the so called Nagle-like algorithms, which is controlled via the `SCTP_NODELAY` option. Transport protocols that support the configuration of application PDU bundling are tunable (**0/1**) whereas others do not provide this service.

- **Error detection**: Error detection is either fully supported by a checksum which covers the entire packet, or it may be adjustable by specifying the range of the checksum. The usage of a partial checksum of course leads to possible delivery of corrupt data. If partial error detection is used this is indicated by **p1**, whereas the total one is denoted by **x**.
- **Reliability**: TCP only supports total reliability (**t**). In contrast SCTP gives also the possibility to define different partial reliability mechanisms (**p2**).
- **Delivery type**: There are only two types of delivery: message-based (**m**) and stream-based (**s**). The message-based delivery refers to the feature "preservation of message boundaries" in [44].
- **Delivery order**: The order of delivery is obviously either ordered (**o**) or unordered (**u**) and hence of course a service towards the transport user.
- **Multi-streaming**: This feature is tunable (**0/1**, currently only supported by SCTP).
- **Multi-homing**: This feature is tunable (**0/1**, currently only supported by SCTP).

Excluded transport features

The following features from [44] do not add a service for a transport user and are therefore not taken into account in the further analysis:

- **Full duplex**: Since all current transport protocols support this feature, there is no need to distinguish.
- **ECN capable**: ECN is a part of congestion control and therefore controlled via the congestion control feature.
- **Selective ACKs**: The usage of selective acknowledgments is a part of congestion control.

- **Path MTU discovery (PMTUD)**: Although the article [44] mentions that UDP itself does not provide this feature, PMTUD can be used with any packetization layer as specified in RFC 4821 [46]. For example the current Linux implementation performs PMTUD for UDP and does not allow a sender to send larger datagrams than the current MTU by indicating an error [47]. DCCP clearly supports this feature as mentioned in RFC 4340 [9] just like SCTP and TCP. Hence, we do not distinguish our services by this feature, but the current PMTU can always be retrieved through our API.

- **Application PDU fragmentation**: The fragmentation of one application PDU into smaller packets can be performed for all transport protocols. Therefore, we also do not have to take care about this feature.

- **Protection against SYN flooding attacks**: This makes the protocol robust against such kinds of attacks, but does not add any service for a transport user.

- **Allows half-closed connections**: This seems to be a historical remainder rather than a service for a transport user, and therefore it is ignored.

- **Reachability check**: Normally a reachability check is only necessary in multi-homed protocols to check whether an endpoint address is reachable or not. Consequently, this feature is clearly included in the multi-homing feature.

- **Pseudo-header for checksum**: This is a protocol internal mechanism which does not add any kind of service towards a transport user.

- **Time wait state**: This is also a protocol internal detail which does not add any kind of service towards a transport user.

4.1.2 Result of transport service analysis

In Table 4.2 all currently available transport services are shown. This list has been constructed by expanding Table 4.1 and listing all possible combinations of transport features as separate transport services. It resulted in 32 services created by SCTP and only 3 services derived from TCP and UDP together. The fact that over 93 % of services are created by recently introduced transport protocols, DCCP, SCTP and UDP-Lite, supports our proposal that in the future a new transport API is needed to facilitate the usage of all these different services. A network programmer will soon need more knowledge about the purposes of the variety of transport protocols. Therefore, the usage of an API

service no.	transport protocol	connection-oriented	flow control	congestion control	app. PDU bundling	error detection	reliability	delivery type	delivery order	multi streaming	multi homing
1	TCP	x	x	x		x	t	s	o		
2	TCP	x	x	x	x	x	t	s	o		
3	UDP					x		m	u		
4	UDP-Lite					x		m	u		
5	UDP-Lite					p1		m	u		
6	DCCP	x	x	CC 2		x		m	u		
7	DCCP	x	x	CC 2		p1		m	u		
8	DCCP	x	x	CC 3		x		m	u		
9	DCCP	x	x	CC 3		p1		m	u		
10	DCCP	x	x	CC 4		x		m	u		
11	DCCP	x	x	CC 4		p1		m	u		
12	SCTP	x	x	x		x	t	m	o		
13	SCTP	x	x	x		x	t	m	o		x
14	SCTP	x	x	x		x	t	m	o	x	
15	SCTP	x	x	x		x	t	m	o	x	x
16	SCTP	x	x	x		x	t	m	u		
17	SCTP	x	x	x		x	t	m	u		x
18	SCTP	x	x	x		x	t	m	u	x	
19	SCTP	x	x	x		x	t	m	u	x	x
20	SCTP	x	x	x		x	p2	m	o		
21	SCTP	x	x	x		x	p2	m	o		x
22	SCTP	x	x	x		x	p2	m	o	x	
23	SCTP	x	x	x		x	p2	m	o	x	x
24	SCTP	x	x	x		x	p2	m	u		
25	SCTP	x	x	x		x	p2	m	u		x
26	SCTP	x	x	x		x	p2	m	u	x	
27	SCTP	x	x	x		x	p2	m	u	x	x
28	SCTP	x	x	x	x	x	t	m	o		
29	SCTP	x	x	x	x	x	t	m	o		x
30	SCTP	x	x	x	x	x	t	m	o	x	
31	SCTP	x	x	x	x	x	t	m	o	x	x
32	SCTP	x	x	x	x	x	t	m	u		
33	SCTP	x	x	x	x	x	t	m	u		x
34	SCTP	x	x	x	x	x	t	m	u	x	
35	SCTP	x	x	x	x	x	t	m	u	x	x
36	SCTP	x	x	x	x	x	p2	m	o		
37	SCTP	x	x	x	x	x	p2	m	o		x
38	SCTP	x	x	x	x	x	p2	m	o	x	
39	SCTP	x	x	x	x	x	p2	m	o	x	x
40	SCTP	x	x	x	x	x	p2	m	u		
41	SCTP	x	x	x	x	x	p2	m	u		x
42	SCTP	x	x	x	x	x	p2	m	u	x	
43	SCTP	x	x	x	x	x	p2	m	u	x	x

Table 4.2: Overview of currently available transport services.

which allows the transport user to specify a service rather than first choosing a protocol and afterwards tuning the features would dramatically reduce the necessary knowledge of transport protocols.

4.2 Step 2: Obvious/intuitive reductions of transport services

The result of Step 1, Table 4.2, presents a summary of all currently available transport services. Step 2 takes this as input for obvious reductions which can be done by simple conclusions. The reductions, which are achieved by removing unnecessary features or merging duplicate features and services, are explained in the following. At the end of this step we present the reduced set of transport services as well as an outlook on possible future services.

4.2.1 Removing connection orientation

The distinction between connection-oriented and connection-less can be removed because this reduction does not restrict the total diversity of services. Therefore, after removing the column "connection-oriented", we still have the same number of services. This simplification leads to a more uniform API. As mentioned in Section 3.5, all recently introduced transport APIs are unified over the connection mode.

The usage of our API always resembles traditional connection-oriented communication (sockets are created and can be reused until they are closed), but there is a clear difference on the wire: a non-congestion controlled unreliable socket will always use UDP or UDP-Lite to communicate. This distinction is static, and clear by documentation.

4.2.2 Removing delivery type

TCP is the only protocol that uses streams as delivery type. The two services (numbers 1 and 2 shown in Table 4.2) provided by TCP can also be realized by SCTP services (12 and 28). Therefore, we do not take the delivery type further into account as a service because a message-based SCTP flow can easily imitate a stream-based flow towards the transport user. The choice of the delivery type does not affect the type of service because the delivery type is just chosen by the used I/O function (message-based vs non-message-based I/O functions).

This reduction results in merging the services 1 and 12 as well as 2 and 28 into two services (shown in Table 4.6 as services 1 and 2) which can either be realized by using TCP or SCTP.

4.2.3 Renaming congestion control

Since we do not distinguish between all available variants of congestion control for TCP-like flows, because these variants do not change the service for the transport user, we introduce the term of *Flow Characteristic.* This term should affirm that we are focusing on the implied characteristics by applying a certain kind of congestion control including flow control and do not care about which congestion control procedures are employed to reach the desired characteristics. Currently, only three flow characteristics, which are described in Table 4.3 in detail, are available:

Name	Description	Usage
TCP-like	TCP-like Congestion Control is appropriate for flows that would like to receive as much bandwidth as possible over the long term, consistent with the use of end-to-end congestion control. Flows must also tolerate the large sending rate variations characteristic of AIMD congestion control, including halving of the congestion window in response to a congestion event [10].	It should be used by senders who would like to take advantage of the available bandwidth in an environment with rapidly changing conditions, and who are able to adapt to the abrupt changes in the congestion window typical of TCP's Additive Increase Multiplicative Decrease (AIMD) congestion control [10].
Smooth	It is reasonably fair when competing for bandwidth with TCP flows, but has a much lower variation of throughput over time compared with TCP [12].	Therefore it is made more suitable for applications such as streaming media where a relatively smooth sending rate is of importance [12]. TFRC congestion control is appropriate for flows that would prefer to minimize abrupt changes in the sending rate [11].
Smooth for Small Packets (Smooth-SP)	Flows using TFRC-SP compete reasonably fairly with large-packet TCP and TFRC flows in environments where large-packet flows and small-packet flows experience similar packet drop rates [48].	The Small-Packet (SP) variant of TFRC is designed for applications that send small packets to achieve the same bandwidth in bps (bits per second) as a TCP flow using packets of up to 1500 bytes [48].

Table 4.3: Overview of current flow characteristics.

We have the mentioned three flow characteristics, but after Step 1 we had four different types of congestion control. That is fine because the TCP-like congestion control of TCP and SCTP can obviously be merged with the TCP-like congestion control of DCCP. Therefore, it can be unified under the "TCP-like" flow characteristic.

The small packet variant for the smooth flow characteristic should be chosen automatically by the new transport API when the packets are in a reasonable range (size between 1 and 1500 bytes). Since the API does not know which size the packets have in future, this decision is not trivial. Therefore, we still distinguish between the normal and the small packet variant of the smooth flow characteristic.

4.2.4 Removing of flow control

The flow control feature is removed, because the employment of it and congestion control coincide. There is no service which has congestion control but no flow control. Therefore there is no need to distinguish between flow control and flow characteristic. Whenever a flow characteristic is specified, flow control will be applied as well.

4.2.5 Removing multi-streaming

In our opinion the service multi-streaming should not be visible to a transport user because this makes it difficult for the programmer to distinguish between multi-streaming services and non-multi-streaming services. The transport user would always have to keep in mind whether she wants to use this feature or not. The recoding effort would be significant if the decision changes later.

Instead the new API should reuse already existing connections and automatically employ multi-streaming for services where this is possible. In particular the mapping onto streams of the same connection is only allowed if *non conflicting service requirements* (described in the paragraph below) are provided. Therefore, the new socket API works on a per-stream basis, where each socket represents exactly one stream. The transport user should not need to take care about whether this stream is mapped on a new connection (for example, for a requested service that needs TCP), or if a stream is used within an already existing SCTP association.

Non-conflicting service requirements

For making use of multi-streaming, if provided by a protocol, the service requirements must not conflict.

Two sets of service requirements of service A and service B do not conflict iff for all requirements one of the following rules applies:

- both services have exactly the same requirement
- the requirement can be defined on a per-stream basis.

If the services A and B, for example, are required as shown in Table 4.4, then service B can be realized as a stream if SCTP is used for service A, because the only difference between the services is the delivery order. This is possible because SCTP allows to decide on the delivery order for every message and hence also on a stream based granularity.

	flow charac-teristic	app. PDU bundling	error detection	reliability	delivery order	multi-homing
Service A	x	x	x	x	o	
Service B	x	x	x	x	u	

Table 4.4: Service requirements of two non conflicting services.

An example for conflicting service requirements is shown in Table 4.5. The requirements for service B remain the same, whereas service A additionally requires multi-homing. Since multi-homing is currently only supported by SCTP and multi-homing is in use for the whole association or not, it is not possible to tune this requirement on a per-stream basis.

	flow charac-teristic	app. PDU bundling	error detection	reliability	delivery order	multi-homing
Service A	x	x	x	x	o	x
Service B	x	x	x	x	u	

Table 4.5: Service requirements of two conflicting services.

The term “non-conflicting service requirements” is very important for the designers and implementors of the new API, but it need not to be understood by the transport user, because this complexity is hidden underneath the API. We do not give an overview of non-conflicting services here, because the decision whether a feature can be controlled on a per-stream basis or not, depends on the transport protocol implementation and not just on the protocol specificiation.

4.2.6 Merging duplicate services

Table 4.2 clearly shows that some services are duplicates in the sense that they are offered by more than one transport protocol. We have already discovered such a merging of duplicate services when the two services, created by TCP, were merged with two services of SCTP in Section 4.2.2. This merging resulted in one service which can be provided by two transport protocols.

Another reduction of two services (number 3 and 4) is possible because, if UDP-Lite uses full coverage of the checksum, the provided service is exactly the same as the one provided by UDP. Therefore, we can omit service number 4.

The last merging of duplicate services affects service numbers 6 and 24. When DCCP is used with TCP-like flow characteristic and full error detection, this service is equal to the one provided by SCTP using no application PDU bundling, partial reliability (in this case no reliablility at all) and unordered delivery.

4.2.7 Result of reducing the transport services

As a summary, Table 4.6 shows the final set of transport services after applying the aforementioned reductions and simplifications. The number of services could be reduced to 23. In addition, the number of features of a service is now only 6. Therefore, the transport user has to decide only on these 6 features instead of first choosing a transport protocol and afterwards deciding on a lot of options. This makes the use of new transport protocols easier because they are highly configurable and therefore challenge the network application developer. In order to get a better overview about the offered services, we have restructered Table 4.6. The column of the *supported transport protocols* is hidden by the protocol-independent API, because this is the result of chosing a service. Moreover, the services have been ordered according to their features that means that the first services are listed do not use a feature.

Table 4.6 does not only give an overview of the transport services provided by our API, but also gives the application developer the possibility to see what kinds of services will work and to look up the service number. The presented table could also be the starting point for the development of new transport protocols, because needed but currently not provided services can be easily identified. In total there are 144 service combinations by these 6 features, but only 23 of them are provided by today's transport protocols. The two mature protocols (TCP and UDP) offer only 4 services together.

4.3 Additional design considerations

One feature in Table 4.6 is ***static***, meaning that it is decided at socket creation time and can not be changed. Other features are also chosen at creation time, but they are ***configurable***. This means they cannot be turned on/off dynamically, but their parameters may be changed during usage. The last group of features can be decided ***dynamically*** and hence lead to a service transition for a socket.

service no.	*supported by transport protocol(s)*	flow charac-teristic	app. PDU bundling	error detection	reliability	delivery order	multi-homing
1	*TCP/SCTP*	TCP-like		x	t	o	
2	*TCP/SCTP*	TCP-like	x	x	t	o	
3	*UDP/UDP-Lite*			x		u	
4	*UDP-Lite*			p1		u	
5	*DCCP/SCTP*	TCP-like		x	[p2]	u	
6	*DCCP*	TCP-like		p1		u	
7	*DCCP*	Smooth		x		u	
8	*DCCP*	Smooth		p1		u	
9	*DCCP*	Smooth-SP		x		u	
10	*DCCP*	Smooth-SP		p1		u	
11	*SCTP*	TCP-like		x	t	o	x
12	*SCTP*	TCP-like		x	t	u	
13	*SCTP*	TCP-like		x	t	u	x
14	*SCTP*	TCP-like		x	p2	o	
15	*SCTP*	TCP-like		x	p2	o	x
16	*SCTP*	TCP-like		x	p2	u	x
17	*SCTP*	TCP-like	x	x	t	o	x
18	*SCTP*	TCP-like	x	x	t	u	
19	*SCTP*	TCP-like	x	x	t	u	x
20	*SCTP*	TCP-like	x	x	p2	o	
21	*SCTP*	TCP-like	x	x	p2	o	x
22	*SCTP*	TCP-like	x	x	p2	u	
23	*SCTP*	TCP-like	x	x	p2	u	x

Table 4.6: Overview of reduced set of transport services.

4.3.1 Static features

Flow characteristic

is clearly a property which belongs to one socket. It is not possible to change the flow characteristic for a socket. Hence all packets sent via the same socket use the same flow characteristic. The current available flow characteristics listed in Table 4.3 do not have any parameters and therefore this is a static feature.

4.3.2 Configurable features

Error detection

is chosen upon creation of a socket. This means that the user decides at the beginning whether the socket should detect errors fully or only partially by chosing a corresponding service. When partial error detection has been chosen, the user has the possibility to specify the checksum coverage. The checksum coverage can be in the range from 0 (which means no checksum at all) to the length of the total message.

Reliability

is also chosen at socket creation and only the parameters can be changed later. Currently only services offered by SCTP support partial reliability and the only standardized form of partial reliability is timed reliability. This feature allows the user to indicate a limit on the duration of time that the sender should try to transmit/retransmit the message [16]. We take only this form of partial reliability into account.

Multi-homing

must also be chosen when a socket is created. The configuration of this feature is similar to the one used by the SCTP socket [2].

4.3.3 Dynamic features

The modification of these features must be legal; i.e. the requested service must be available by the API, otherwise the modification leads to an error. For example if service 7 of Table 4.6 is used, it is forbidden to turn on application PDU bundling or set the delivery order to ordered, because currently there exists no service with this combination of features.

Application PDU bundling

can be utilized when small messages should be sent and bundled together in one or more packets (called the Nagle algorithm for TCP). This procedure may affect the delay of single messages, and that is why the user must have the possibility to turn it on/off.

Delivery order

is a feature which can be defined on a per-message basis and hence must be a dynamic feature. Either the delivery is ordered and hence sequential, or it is unordered.

4.3.4 Future flow characteristics

This section provides a small outlook on possible future services offered by the protocol-independent Internet transport API concerning new flow characteristics. Currently there are two new flow characteristics in development in the IETF: *Low Extra Delay Background Transport* (LEDBAT) and *MulTFRC*. Both react to congestion, but have a different focus in their flow characteristic. MulTFRC is smooth and adjustable via its aggression parameter N, whereas LEDBAT provides a scavenger service for long-lived transfers.

The description and the usage of both new flow characteristics shown in Table 4.7 have been taken from their current Internet-Drafts LEDBAT [49] and MulTFRC [50].

If the two flow characteristics become standardized, we would introduce a parameter for the flow characteristic: N is an element of the natural numbers for the TCP-like flow characteristic, whereas N can be an element of the real numbers for smooth flows. In general this parameter should determine the aggressiveness compared to a standard TCP flow. If one chooses $N = 1$, the resulting service will behave like TCP congestion control in the case of TCP-like flow characteristic and like TFRC for the smooth flow characteristic. For $0 <= N < 1$ the service will have a scavenger characteristic which is implemented by LEDBAT. In the case of, $N > 1$ MulTFRC is chosen and parameterized with the given N to enable the smooth flow characteristic. If a TCP-like flow characteristic should be used with $N > 1$, MulTCP, as proposed in [51] and [52], can be utilized. The combinations of flow characteristic and the parameter N are summarized in Table 4.8.

Name	Description	Usage
Adjustable Smooth	MulTFRC emulates a number of TFRC flows with more flexibility than what would be practical or even possible using multiple real TFRC flows [50].	Like TFRC, MulTFRC is suitable for applications that require a smoother sending rate than standard TCP. Possible reasons to use MulTFRC include the assignment of priorities according to user preferences, increased efficiency with $N > 1$, the implementation of low-priority "scavenger" services, and resource pooling [50]. A very important point when using MulTFRC is that the parameter N is fixed and therefore set at the beginning of a transfer.
Scavenger	LEDBAT is designed to allow to keep the latency across the congested bottleneck low even as it is saturated. LEDBAT takes advantage of delay measurements and backs off before loss occurs. It thus implements an end-to-end version of scavenger service [49].	This allows applications that send large amounts of data, particularly upstream on home connections, such as peer-to-peer applications, to operate without destroying the user experience in interactive applications [49].

Table 4.7: Overview of current flow characteristics.

Range of N	smooth flow characteristic	TCP-like flow characteristic
$N = 0$	scavenger(LEDBAT)	scavenger(LEDBAT)
$0 < N < 1$	MulTFRC	not possible
$N = 1$	TFRC	TCP
$N > 1$	MulTFRC	MulTCP

Table 4.8: Overview of the parameterized flow characteristics and their realizing technology.

Analysis of future flow characteristics

The flow characterisitic and the flow characteristic parameter N is not changeable during transmission. The reasons that N must not be changed for MulTFRC, while a transfer is ongoing, are [50]:

> The effects of changing N during the lifetime of a MulTFRC session on the dynamics of the mechanism are yet to be investigated; in particular, it is unclear how often N could safely be changed, and how "safely" should be defined in this context. Further research is required to answer these questions.

In general, the modification of the flow characteristic during a session has not been checked so far. Therefore, we propose the flow characteristic as well as the parameter N of it as not changeable during a transfer.

4.4 Design conclusion

We have 23 services that our API currently offers to a transport user. Hence, the complexity could nearly be reduced by half when looking at the number of services (the analysis revealed 43 services). Moreover, we have exploited a large potential for simplifying the usage of the Internet transport API. For example, the one-to-one mapping of sockets and streams facilitates an easier usage of advanced features like multi-streaming.

The careful considerations about future flow characteristics clearly show that our API is adaptable for future congestion control mechanisms. Furthermore, new transport protocol mechanisms can be introduced by additional transport features or by extending the range of current transport features. New resulting services just get a new service number. These thoughts justify that our API is adaptable for future features or adaptions which will certainly occur.

In Chapter 5 we build upon the result of the design process to show that it is really possible to develop a simpler and more flexible API which in addition enables the gradual deployment of new transport protocols and features.

Chapter 5

Implementation

This chapter shows in detail how an implementation of the protocol-independent transport API can be accomplished. The investigation of available transport APIs in Chapter 3 has shown that the well-known socket API is the best starting point for the *protocol-independent API* (PI_API). Therefore, the decision is clear to rely on the socket API and adapt it with functions and options where needed. Based on the socket adaptions a proof-of-concept implementation has been built which shows that it is possible to obtain the same performance with and without the PI_API. Finally, we give an outlook about future challenges that must be taken into account when implementing the whole protocol-independent transport system.

5.1 Socket adaptions

The PI_API is based on the connection-oriented socket API, and also uses some ideas from the Internet-Draft specifying the SCTP API [2]. The prefix `PI_` (protocol-independent) is used for all newly introduced API mechanisms.

5.1.1 Socket creation

The user has to pass the identification of the service (*service number*) as defined in Table 4.6. Alternatively the user could have passed the values of the six features. But since it would be rather complicated to pass up to six parameters and also would harm the flexibility to introduce new features, the decision was taken against this procedure. The creation of a socket is shown in Figure 5.1.

```
int socket(int domain, int service);
```

Figure 5.1: Socket creation of the protocol-independent transport API.

The domain may either be `PF_INET` or `PF_INET6`. While the first one refers to the protocol family of IPv4, the second one refers to IPv6. The second parameter service represents the aforementioned service number which may be defined by passing a number or using a surrogate name. For example, the services that are often used, i.e. services 1, 2 and 3 in Table 4.6 may receive the following surrogate names: PI_TCPLIKE_NODELAY, PI_TCPLIKE and PI_NO_FC_UNRELIABLE. Others may be introduced on demand.

5.1.2 Socket options

All service characteristics can be retrieved and set through the two functions shown in Figure 5.2. The setsockopt() function enables the user to set the parameters of configurable features and change the dynamic features. In general, all socket options which are utilized by the protocol-independent socket API are retrieved and set with the level argument PI_API.

```
int getsockopt(int socket, int level, int option_name,
               void *restrict option_value,
               socklen_t *restrict option_len);

int setsockopt(int socket, int level, int option_name,
               const void *option_value, socklen_t option_len);
```

Figure 5.2: Functions for retrieving and setting the socket options.

Naturally all options of the socket level SOL_SOCKET (mentioned as "Generic socket options" in Section 3.3.3) are still accessible and of course affect the behavior of the PI_API. For example, the adjustments of SO_RCVBUF and SO_SNDBUF have an impact on the corresponding buffers of a socket.

As mentioned in the first design step (see Section 4.2.2), the user has the ability to query the maximum message size which is related to the current path MTU. This is done via the function getsockopt() by passing PI_MAXMSG as option_name.

Configurable features

The configurable features (mentioned in Section 4.3.2) are fixed in principle, but their parameters may be changed during the usage of a socket.

Error detection When partial error detection has been chosen, the user has the possibility to specify the checksum coverage via the socket option with PI_CSCOV. The checksum coverage can be in the range from 0 (which means no checksum at all) to the length of the total message.

Reliablity Currently only timed partial reliablity is available if partial reliablity has been chosen by the service number. The user can tune this feature by setting the time via the `PI_TIME_RELIABLE` socket option. Future partial reliability mechanisms could be made accessible through additional `PI_` socket options.

Multi-homing If multi-homing is available for the current service, the configuration of this feature is similar to the one used by the SCTP socket [2]. The function `pi_bindx()`, shown in Figure 5.3, is offered for maintaining the server side addresses. This function offers the same functionality as the `sctp_bindx()` function does. The addresses associated with a socket are the eligible transport addresses for the endpoint to send and receive data [2]. The endpoint will also present these addresses to its peers during the association initialization process [15]. Nevertheless only one local port can be associated with the socket through the normal bind function.

```
int pi_bindx(int socket, struct sockaddr *addrs,
             int addrcnt, int flags);
```

Figure 5.3: Function for enabling multihoming.

The function `pi_bindx()` allows the user to add or delete specific addresses as described in RFC 5061 [53]. The addresses specified in parameter `*addrs` are either added with `PI_BINDX_ADD_ADDR` or removed with `PI_BINDX_REM_ADDR`. These two `PI_BINDX_XXX` parameters are passed via the parameter `flags`.

In addition the user can mark one of the addresses as primary address by using the option `PI_SET_PEER_PRIMARY_ADDR`. Further details about multi-homing can be found in the Internet-Draft for socket extensions for SCTP [2].

Dynamic features

As mentioned in Section 4.3.3 the modification of the features must be legal. In the case of an illegal modification the error `EINVAL` is returned by the function `setsockopt()`. `EINVAL` means that the specified option is invalid at the specified socket level or the socket has been shut down [40]. The setting of a socket option affects all subsequent messages which are sent via the given socket.

Application PDU bundling This feature which resembles the Nagle algorithm for TCP can be turned on/off depending on the socket option `PI_NODELAY`.

Delivery order The delivery order is either ordered (PI_DELIVERY_ORDERED) and hence sequential or it is unordered (PI_DELIVERY_UNORDERED) by setting the value of the socket option PI_DELIVERY_ORDER.

5.1.3 Send/receive mechanisms

Since one goal is simplifying the socket API, the PI_API offers two send/receive mechanisms instead of the 4 different methods of the standardized socket API mentioned in Section 3.3.3. The send/receive functions are shown in Figure 5.4. They are identical to the current version of the standardized socket API (see [40]).

```
ssize_t sendmsg(int socket, const struct msghdr *message, int flags);

ssize_t recvmsg(int socket, struct msghdr *message, int flags);

ssize_t send(int socket, const void *buffer, size_t length, int flags);

ssize_t recv(int socket, void *buffer, size_t length, int flags);
```

Figure 5.4: Send/receive funtions of the protocol-independent Internet transport API. The first two functions are corresponding to *message-based* send/receive mechanisms and the others to *stream-based.*

The **stream-based** send/receive mechanism is only usable for services which do not conflict with the properties of stream-based delivery. Hence, it can be exploited for services 1, 2, 11, and 17 from Table 4.6, because all other services make use of partial reliability, partial error detection or unordered delivery. The user has to provide a `buffer` and specify its `length` and the function returns the received length, which is less or equal to the specified `length`.

The **message-based** delivery is usable for all services. The user has to specify the message within the structure `msghdr` and pass this structure and the `flags` to the function. The user can tune the delivery order via the `flags` field by bitwise ORing with either PI_DELIVERY_UNORDERED or PI_DELIVERY_ORDERED. Thus, the user can switch on a per-message basis within the following service pairs 1/12, 2/18, 5/14, 11/13, 15/16, 17/19, 20/22, and 21/23. The usage of the per-message adjustment of the delivery type is only valid for this `sendmsg()` call. For changing the service for all subsequent messages the socket option PI_DELIVERY_ORDER must be utilized (explained in Section 5.1.2).

Additionally, the user can specify the parameters for error detection and reliability on a per-message basis via the `cmsghdr` as proposed in RFC 3542 [54]. The level argument of the `cmsghdr`, `cmsg_level` must be set to PI_API and the type `cmsg_type` to `PI_TIME_RELIABLE` or `PI_CSCOV`. Finally, the values for time reliability and/or the checksum coverage are specified in the `cmsg_data` as natural numbers. Further details on the configuration of these two parameters can be found in Section 5.1.2.

5.1.4 Blocking vs non-blocking mode

The protocol-independent socket API supports blocking as well as non-blocking I/O similar to the standard socket API (as described in Section 3.3.3). For simplifying the PI_API every created socket works in blocking mode per default. The user can switch to the non-blocking I/O mode via the `O_NONBLOCK` flag using the function `fctnl()`.

5.2 Proof of concept implementation

The proof of concept implementation shows that it is possible to build the API as it has been designed and specified. Moreover, it demonstrates that the PI_API achieves the same performance, but with less programming effort and transport protocol knowledge. The rudimentary implementation of one exemplary service is a user-space implementation which currently supports only the needed service.

5.2.1 Implementing the protocol-independent Internet transport API

Figure 5.5 presents the headers of the PI_API to give an overview of the entire API. It can be seen that only the `socket()` call has been altered to meet the requirements of the PI_API. All other socket calls remain the same as in the Single Unix System Specification [40]. Naturally, the type definitions are identical to the Single Unix System Specification. For a clear distinction between `PI_API` calls and normal socket calls, the prefix `pi_` has been introduced for all functions and as already mentioned the prefix `PI_` for all options.

The API implementation internally builds a state for every created socket. In particular, the service features are deduced from the service number and stored with the service number in a structure. Moreover, the state information contains the utilized protocol as well as the destination address of the socket after it has been "connected". The collection of states is currently organized as a linked list, but this has to be improved to be competitive when using several sockets concurrently.

```
int pi_socket(int domain, int service_no);

int pi_bind(int sockfd, const struct sockaddr *address,
            socklen_t address_len);

int pi_listen(int sockfd, int backlog);

int pi_accept(int sockfd, struct sockaddr *address,
              socklen_t *address_len);

int pi_connect(int sockfd, const struct sockaddr *address,
               socklen_t address_len);

ssize_t pi_sendmsg(int socket, const struct msghdr *message, int flags);

ssize_t pi_send(int sockfd, const void *buffer, size_t length,
                int flags);

ssize_t pi_recvmsg(int socket, struct msghdr *message, int flags);

ssize_t pi_recv(int sockfd, void *buffer, size_t length, int flags);

int pi_close(int fildes);

int pi_getsockopt(int sockfd, int level, int optname, void *optval,
                  socklen_t *optlen);

int pi_setsockopt(int sockfd, int level, int optname,
                  const void *optval, socklen_t optlen);
```

Figure 5.5: piSocket.h: The functions of the protocol-independent Internet transport API are defined.

5.2.2 Application scenario

The application scenario measures the time for transferring a message between two hosts (A and B). This is called one-way delay and in this special case it could be called *one-way application to application delay* (app-to-app delay), because it really measures the time from the application on host A to the application on host B. Hence, the time also includes the processing time of the transport API and all underlying layers as well as the transmission time itself.

Since the protocol-independent transport API should be compared with the existing socket API, the implementation has to rely on the client-server model which the socket API is based upon. It does not matter whether the client or the server is the sending part of the application scenario. Therefore, an arbitrary decision was taken that uses the client application as sending part and the server application as receiving and hence logging part.

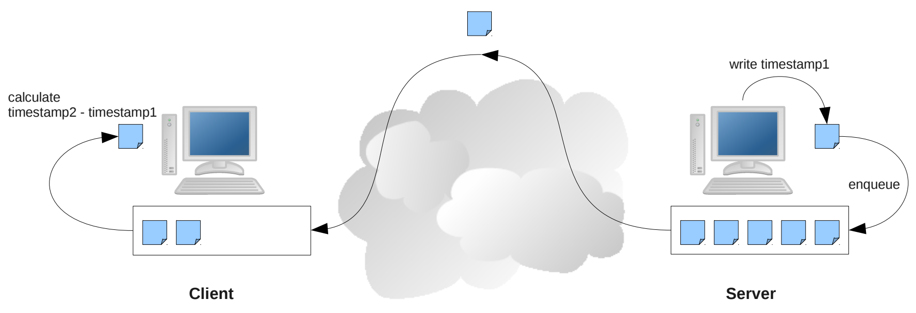

Figure 5.6: Overview of the application scenario.

In Figure 5.6 the basic application scenario is depicted. The client produces messages with the current timestamp of its machine and enqueues them in the socket buffer as long there is space for new messages. Therefore, the size of the send buffer has a direct impact on the one-way app-to-app message delay. Then the messages are transmitted over an arbitrary network to the server application which consumes these as fast as possible and calculates the difference between the current timestamp on this machine and the timestamp of the message. Naturally, this does not result in the real delay between the two hosts, because their clocks are not perfectly synchronized. Since all message delays encounter the same clock de-synchronization this fact does not need to be taken into account further.

Hypothesis

This simple application scenario proofs that *the protocol-independent Internet transport API does not add a significant delay to the overall delay and therefore preserves the more significant benefits that new transport protocols can yield.*

In addition, we claim that *the protocol-independent Internet transport API makes network application developement easier by simplifying access to transport protocol features.* This step involes the programming effort as well as the knowledge required to develop efficient network applications.

In particular, service 12 in Table 4.6 has been amply investigated. The service number 12 has the following service features: it uses TCP-like congestion control, full error detection and total reliability, but does not use application PDU bundling and multihoming, and its delivery order is unordered. This service is supported only by SCTP because no other transport protocol has the same properties.

SCTP implementation

In the current Internet-Draft for the socket extensions for SCTP [2], there are two possibilities described to enable an SCTP socket to send data unordered. However, only one of the described variants worked for the example implementation on the test system (as described in the appendix). The first possibility is to pass the information with the `cmsghdr` within the `sctp_sndrcvinfo` structure in the `sendmsg()` call, and the second one is a newly defined `sctp_send()` call.

```
struct sctp_sndrcvinfo *si;
struct cmsghdr *cmsg;
char cbuf[sizeof (*cmsg) + sizeof (*si)];
size_t cmsglen = sizeof (*cmsg) + sizeof (*si);

cmsg = (struct cmsghdr *)cbuf;
cmsg->cmsg_level=IPPROTO_SCTP;
cmsg->cmsg_type= SCTP_SNDRCV;
si = (struct sctp_sndrcvinfo *)(cmsg + 1);
si->sinfo_stream = 1;
si->sinfo_flags = SCTP_UNORDERED;

msg.msg_control = cbuf;
msg.msg_controllen = cmsglen;

sendmsg(sockfd, &msg, 0);
```

Figure 5.7: Solution 1 for using unordered data delivery for SCTP sockets.

Figure 5.7 shows the first possibility to use unordered data delivery for SCTP sockets. It is obvious that it is a rather complex task to specify that a message should be sent without regard to its order. This method did not work on the test system even after investing extensive efforts into it. Therefore, it has been decided to give up on using this solution in the test system. The reasons for the dysfunctionality could not be exploited, but might be a bug in the current SCTP socket API implementation on the test system.

In Figure 5.8 first the prototype of the `sctp_sendmsg()` function is shown. Already the prototype shows clearly that this function has been overloaded with functionality. Hence, there is no doubt that a network application developer can become confused by the overwhelming choice of 10 parameters. Normally, the `sendmsg()` calls imply that the message is passed via the `struct msghdr`.

```
int sctp_sendmsg(int sd, const void *msg, size_t len,
                 const struct sockaddr *to, socklen_t tolen,
                 uint32_t ppid, uint32_t flags, uint16_t stream_no,
                 uint32_t pr_value, uint32_t context);

sctp_sendmsg(sockfd, textMsg, msgLength, NULL, 0, 0,
             SCTP_UNORDERED, 1, 0, 0);
```

Figure 5.8: Solution 2 for using unordered data delivery for SCTP sockets.

Therefore, either the function name has been poorly chosen or the parameter should be changed to the structure type. Finally, it can be seen in Figure 5.8 that for using the unordered message delivery most of the parameters are ignored by setting them to 0 or `NULL`.

In addition, the flag `SCTP_NODELAY` has to be set via the function `setsockopt()` in order to ensure the requested service towards the application. The network application developer is even confronted with the decision whether a one-to-one style socket or a one-to-many style socket should be used (the differences are described in Section 3.3.3). The usage of a one-to-many style socket does not make any sense in the context of the application scenario, but nevertheless the decision has to be made when creating a socket. Hence, the `socket()` call has the following form: `socket(PF_INET, SOCK_STREAM, IPPROTO_SCTP)`. All remaining socket calls are standard socket calls and SCTP does not require to change anything for them.

PI_API implementation

The PI_API implementation of the application scenario is straightforward: The socket is created by the adapted `pi_socket()` call that specifies nothing more than the protocol family `PF_INET` and the service number 12. After creating the socket with this function it is clear by definition that it already has the requested properties. Therefore, the `pi_sendmsg()` call is the same as it would be when using a standard TCP socket. No additional parameters have to be specified, because all necessary information has already been passed via the service number during the socket creation. Figure 5.9 shows the only differences to the SCTP implementation. All remaining socket calls are used in the standard way, with the only difference that for all calls the corresponding `pi_` function must be used. Note that no flags have to be set via a `pi_setsockopt()` call in this case.

```
pi_socket(PF_INET, 12);
pi_sendmsg(sockfd, &msg, 0);
```

Figure 5.9: Solution for using service 12 with the PI socket.

5.2.3 Test results

The claimed hypothesis has been proven by practical tests, where the following test constraints have been applied:

- **Bandwidth 10MBit/s** seems to get a standard for future Internet access. Several ISPs already provide higher maximum bandwidth rates, but they are rarely reached in practice. Therefore, this bandwidth was emulated during the tests.
- **Delay 100ms** is the average of some experiences made in today's Internet.
- **Message size 1448** is the MSS for TCP in the test network, because the timestamp option was enabled.
- **Test size 10000** is a reasonably large test size which reflects the number of messages sent within one test scenario.
- **Optional loss of 1%** was introduced by emulation and can be regarded as loss which could occur in reality caused by wireless networks for example.

For performing tests to compare TCP, SCTP and the PI_API by fair means, one would have to adapt several options of TCP, e.g. the congestion control algorithm should be the same. Since the app-to-app performance of SCTP and TCP should be compared, no adjustments are made and the "reality" is measured. Therefore, all protocols are used with their standard settings of the test system. The details of the protocol parameters of the test system can be found in Appendix A.3. Only the socket send and receive buffers have been adapted, because they have a direct impact on the app-to-app delay measurements. All send and receive buffers have been set to the *bandwidth delay product* (BDP).

Test case 1

This test case evaluates the behavior of SCTP and TCP with the first four test constraints, i.e. without loss. Normally, both should perform quite similarly in this test scenario, but in Figure 5.10 the cumulative distribution function clearly shows that SCTP performs better than TCP. The reason for this behavior has not been discovered, but the result shown in Figure 5.10 has been confirmed by

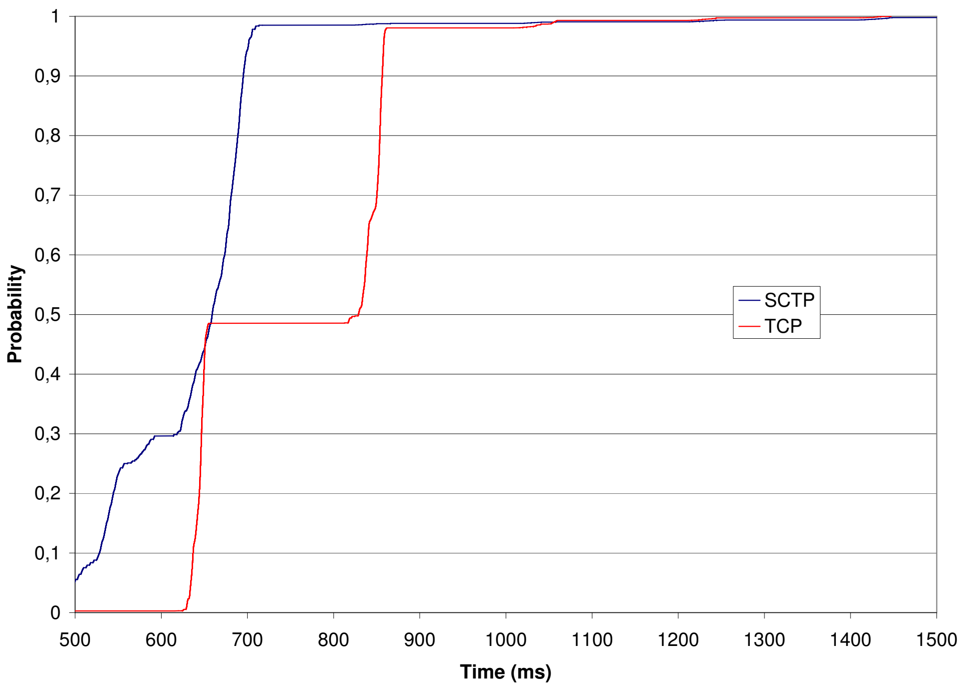

Figure 5.10: Cumulative distribution function of the app-to-app delay: comparing the real world delay of SCTP and TCP without loss.

several independent tests. This testcase has revealed an unexpected behavior for the application scenario using SCTP and TCP.

Test case 2

Test case 2 compares SCTP, the service 12 of the PI_API and TCP under the aforementioned test constraints with 1% loss. Figure 5.11 demonstrates that SCTP performs significantly better than TCP. Even when the results are compared with the preceding test case without loss, a further improvement of SCTP's performance can be seen. Additionally, the service 12 of the PI_API performs a little worse than SCTP on small delays, which might be the case because the PI_API adds a small processing delay. Nevertheless, SCTP and the service 12 of the PI_API achieve more or less the same performance. There are slight differences between SCTP and the service 12 of the PI_API because the random loss sometimes affect the transmit rate worse than at other times. In particular, the time between two losses affects the transmit rate because the congestion control algorithms interpret loss as congestion. Hence, the performance hypothesis that the PI_API does not add a significant delay to the overall delay has been proven by this test case.

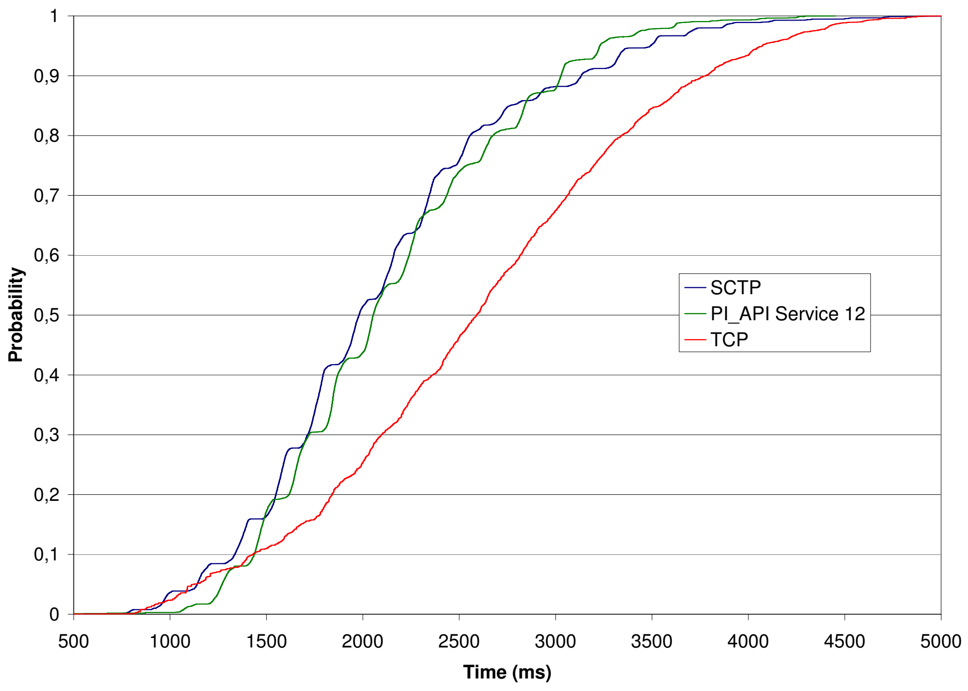

Figure 5.11: Cumulative distribution function of the app-to-app delay: comparing the real world delay of SCTP, PI_API using service 12 and TCP with 1% loss.

5.2.4 Conclusion

In general the test results in the preceding section confirm the hypothesis. Moreover, it has been shown that the protocol-independent Internet transport API does not only achieve the same performance as the specialized transport protocol SCTP, but in addition facilitates the usage of a specialized transport service. The SCTP implementation has revealed how complex the usage of a simple specialized transport service is nowadays. The challenges of using a new transport protocol could be reduced to a minimum by introducing the protocol-independent Internet transport API.

5.3 Future implementation work

Currently the proof of concept implementation is rudimentary and only shows that it is possible to implement the proposed transport API. Hence, there is a large amount of implementation work to be done until all identified transport services can be provided in a satisfactory way. Some of the services can be implemented in a rather simple way, whereas others may require sophisticated techniques to reach the goal. The challenges, that have already been identified and must be taken into account, are mentioned in the following.

5.3.1 Implementing the API in the "kernel"

An in-kernel implementation is superior to a user-space library for several reasons. For example, only a kernel implementation can solve some of the ensuing challenges and in most cases it is obvious that a kernel implementation will help to gain performance too.

5.3.2 Fallback mechanism

The motivation of this thesis has shown in Section 1.2.3 that a fallback mechanism is essential for getting new transport technologies widely deployed. Since it is not clear whether the mentioned fallback mechanism is already available when the entire API is implemented or not, the implementation of such a fallback mechanism should be considered too.

The fallback mechanism works on a best-effort basis. This means that all services are emulated towards the transport user even if not all the needed transport protocols are available on both hosts or on the path between them. Since such an emulation cannot be perfect, it is only natural that this does not result in perfect quality. For example, if service 12 (unordered delivery) is emulated by the PI_API, it may be the case that TCP is used, and hence the delivery is in fact ordered. This will not pose a semantical problem for applications because it is never *required* that messages arrive out-of-order – it is just a network behavior that an application can *allow* for the sake of better performance.

The fallback behavior reveals the true power of the PI_API: application programmers will never see a disadvantage from using the new API, but if a protocol like SCTP or DCCP becomes available, some desired services could be better realized "under the hood". This makes it possible to get these protocols deployed on a gradual basis.

5.3.3 Automatic multi-streaming

In Section 4.2.5 it has already been discovered in detail why it is advantageous to hide multi-streaming underneath the protocol-independent API. Hence, the advantages of multi-streaming (i.e. the removal of the HOL problem) can be exploited without requiring the network application developer to invest extra effort. Some first steps towards exploiting multi-streaming automatically were done by Florian Niederbacher in his diploma thesis [55]. He faced one big challenge due to the current usage of multi-streaming through the SCTP socket API: the socket API does not provide a separate message queue for each stream. This fact has led him to build a complex user-space implementation from scratch, which creates a message queue for each stream as well as a single reader thread which reads all messages from the SCTP socket. Such a complex user-space implementation could be avoided by the PI_API, because it has a clear one-to-one relation between sockets and streams.

5.3.4 Protocol negotiation

The protocol negotiation is essential for the PI_API, because it needs to know which protocols and features are working for the two hosts A and B as well as on the path between them. Two possible solutions have been presented in Section 2.2. In the context of building a protocol-independent Internet transport API the "Happy Eyeballs" proposal might be the most adequate one, because it really tries to establish a connection and hence also checks the protocol's availability along the path from host A to B and vice versa.

5.3.5 Making the best possible choice

The PI_API provides only a low level abstraction, because it stays within the boundaries of the best-effort model of TCP, UDP, UDP-Lite, SCTP and DCCP. Nevertheless, if there is a choice between protocol A and B for a service X, the API should choose the best possible one. In addition, the API should have the capability to emulate a service over another protocol if the specialized protocol is not available on both hosts or on the path.

This component might be the most complex one, because in general "making the best possible choice" is easier said than done. For realizing this feature the protocol negotiation component is also needed, because based on the knowledge gained through the negotiation phase, the best possible choice can be made. Therefore, the information provided by the protocol negotiation component directly influences the outcome of this component.

5.3.6 Events and notifications

Events and notifications are a part of the socket API, but have not been covered by the investigation in this thesis so far. Here, additional efforts are needed for implementing them in a useful and uniform way for the protocol-independent transport API.

Chapter 6

Conclusion

This final chapter summarizes the whole thesis and draws logical conclusions from the results of the preceding chapters. Additionally, it outlines the standardization procedures for the defined system. In the end an outlook shows the steps that must be taken into account for reaching the long-term goal of a standardized protocol-independent Internet transport API.

6.1 Conclusion

After having found the motivation for a protocol-independent API in Chapter 1, the current transport API situation has been investigated in detail in Chapter 3. The investigation has revealed that there are a lot of approaches concerning transport API problems. Nevertheless, none of these approaches is able to provide the needed requirements. For example, only the socket API supports the whole set of today's available transport protocols, but in a rather complex way. Other approaches like ACE, BEEP, DANCE, Name-based Sockets, QSockets and SST mainly focus on abstraction without making use of new technologies. That is why we ended up asking why it seems that; "Old supports new, but new uses old?". Hence, the conclusion has been drawn that the new API has to be simplified and unified over the current Internet transport protocols.

With the knowledge gained by the investigation and the revealed features of today's transport protocols in Chapter 2, the design of a new API has been accomplished in Chapter 4. The design has been started by identifying which protocol features TCP, UDP, SCTP, DCCP and UDP-Lite provide. This resulted in a list of 43 services, where only 3 of them are offered by the nowadays used transport protocols TCP and UDP. Step 2 has reduced the number of services via obvious and intuitive reductions, by removing unnecessary features or merging duplicate features and services. This resulted in 23 services which are offered by today's Internet transport protocols. This is the main result of this master thesis and shown in Table 4.6.

Finally, considerations of an implementation have been made and a proof of concept implementation has shown that the goals of the new API have been achieved in Chapter 5. Based on the investigation in Chapter 3 the decision to build upon the socket API is clear. Therefore, the needed socket adaptions are listed there. They mainly concern the service-based view of the new transport API and are hence inevitable. The proof of concept implementation shows with a simple application scenario that the protocol-independent Internet transport API does not add a significant delay and additionally makes network application development easier by simplifying access to specialized transport protocol features.

6.2 Standardization of Internet transport APIs

First, the question "Who is responsible for writing/achieving standards?" has to be answered. Therefore, we have a look at two different parties/institutions who might be responsible for standardization concerning an Internet transport API.

The IETF describes its own goals of the Internet Standards Process as follows [56]:

- technical excellence
- prior implementation and testing
- clear, concise, and easily understood documentation
- openness and fairness
- timeliness

Although these goals do not conflict with the standardization of an API, the IETF has not specified APIs for a long time. Nevertheless, there already exists an exception: the *Generic Security Services Application Program Interface* (GSS-API). Several RFCs (e.g. RFC 2744 [57]) specify this API and adapt it to the current needs in the form of "Proposed Standards". On the other hand other APIs, for example extensions to the socket API (e.g. RFC 3542 [54], RFC 4584 [58]) have never reached the standard track. Hence, it is legitimate to ask the following question to the IETF: "Why are there no standards for Internet transport APIs?"

The current socket API as well as the XTI has not been standardized for a long time, but after becoming popular both have been standardized by the OpenGroup. Currently, only the socket API is included in the latest Single

UNIX Specification [40]. The Austin Group (a part of the OpenGroup) defines the current set of specifications which is simultaneously POSIX.1-2008, ISO/IEC 9945, IEEE Std 1003.1 and the core of the Single UNIX Specification.

Newer Internet transport APIs like the socket extensions for SCTP have not been standardized yet, neither by the IETF nor by the OpenGroup. Both institutions have the possibility to accomplish a standard for a new transport API, but they seem to hinder themselves in achieving that, because it would be against their cultural bias. The IETF does not see its main task in specifying APIs and the OpenGroup does not specify something that is not really utilized nowadays. Hence, the following conclusion can be drawn: If the IETF wants to see their developed technologies widely deployed, they should move their cultural bias in a way where they allow themselves to standardize APIs. This conclusion can be strengthened by the number of 8 current Internet-Drafts concerning the extensions of the socket API for new transport technologies:

- SCTP Socket API Extensions for Concurrent Multipath Transfer
- Sender Queue Info Option for the SCTP Socket API
- Basic Socket Interface Extensions for Host Identity Protocol (HIP)
- Socket Application Program Interface (API) for Multihoming Shim
- Sockets API Extensions for Stream Control Transmission Protocol (SCTP)
- Socket API Extension for MIF Host
- MPTCP Application Interface Considerations
- TCP Cookie Transactions (TCPCT) Sockets Application Program Interface (API)

6.3 Outlook

The next step towards the protocol-independent Internet transport API clearly is the implementation of the entire API. This step already involves big challenges like automatic multi-streaming and making the best possible choice (both mentioned in Section 5.3). The implementation needs to perform identically to the current transport API for currently used transport services (1, 2 and 3) and needs to show performance advantages for other services in their specific application area. In addition, the API has to provide the fallback mechanism mentioned in Section 5.3.2 and show a zero-cost performance gain, when the fallback mechanism of the API is not used anymore. For demonstrating the future adaption ability the implementation should contain all available and truly necessary transport services at the time of developement. Hence, the current number of 23 services may already grow in the first entire implementation.

Another interesting element for further research is to consider combinations of the six identified features (Table 4.6) which are currently not offered by the available transport protocols. There might be services which may be useful for some applications and hence lead to the development of new protocols or protocol extensions.

Finally, the achievement of a standard would complete the goals of the protocol-independent Internet transport API. Nevertheless, the further development of transport technologies continues and the Internet transport API must be developed along as well.

Appendix A

Testsystem

The testsystem consists of two hosts which were connected directly via a network cable. The network parameters (e.g. bandwidth, delay and loss) were emulated by the mentioned tool in the software specification.

A.1 Hardware

The hardware shown in Table A.1 and Table A.2 was used for acquiring the test results.

Linux host A (Client)	Hardware specification
CPU	Intel®Core^{TM}2 Duo processor SL9400
	1.86GHz, 6MB L2, 1066 MHz FSB
Memory	3GB 1066MHz DDR3 memory
Harddrive	Serial ATA HITACHI HTS72323 320GB 5400rpm
Network card	Intel Corporation 82567LM Gigabit Network

Table A.1: Hardware specification of the test client.

Linux host B (Server)	Hardware specification
CPU	Intel®Core^{TM}2 processor T5500
	1.66GHz, 4MB L2, 667 MHz FSB
Memory	1GB 667MHz DDR2 memory
Harddrive	Serial ATA FUJITSU MHV2080B 60GB 5400rpm
Network card	Broadcom Corporation BCM4401-B0 100Base-TX

Table A.2: Hardware specification of the test server.

A.2 Software

Both test hosts had the same software installed when running the application scenario and measuring the app-to-app delay during the tests. The main characteristics of the used software are shown in Table A.3. The protocol parameters were not adapted and therefore the general settings are listed in Table A.4, for TCP in Table A.5 and for SCTP in Table A.6. Note that the receiver and send buffer were adapted to the BDP in the application scenario.

The network emulation was applied by emulating the needed network behavior on both hosts on the going packets with dummynet [59]. The code that was executed on both hosts is shown in Figure A.1.

Operating system	Ubuntu 9.10 (Karmic Koala)
Kernel version	2.6.31-22
Network emulation	dummynet [59]

Table A.3: Software specification of the test hosts.

A.3 Protocol parameters

System variable	Value
net.core.message_burst	10
net.core.message_cost	5
net.core.rmem_default	114688
net.core.rmem_max	131071
net.core.wmem_default	114688
net.core.wmem_max	131071

Table A.4: System variables general network settings.

```
ipfw add pipe 2 out
ipfw pipe 2 config bw 10000Kbit/s delay 100ms
ipfw pipe 2 config bw 10000Kbit/s delay 100ms plr 0.01
```

Figure A.1: Utilized network emulation commands.

System variable	Value
net.ipv4.tcp_abc	0
net.ipv4.tcp_abort_on_overflow	0
net.ipv4.tcp_adv_win_scale	2
net.ipv4.tcp_allowed_congestion_control	cubic reno
net.ipv4.tcp_app_win	31
net.ipv4.tcp_available_congestion_control	cubic reno
net.ipv4.tcp_base_mss	512
net.ipv4.tcp_congestion_control	cubic
net.ipv4.tcp_dma_copybreak	4096
net.ipv4.tcp_dsack	1
net.ipv4.tcp_ecn	2
net.ipv4.tcp_fack	1
net.ipv4.tcp_fin_timeout	60
net.ipv4.tcp_frto	2
net.ipv4.tcp_frto_response	0
net.ipv4.tcp_keepalive_intvl	75
net.ipv4.tcp_keepalive_probes	9
net.ipv4.tcp_keepalive_time	7200
net.ipv4.tcp_low_latency	0
net.ipv4.tcp_max_orphans	32768
net.ipv4.tcp_max_ssthresh	0
net.ipv4.tcp_max_syn_backlog	1024
net.ipv4.tcp_max_tw_buckets	180000
net.ipv4.tcp_mem	79872 106496 159744
net.ipv4.tcp_moderate_rcvbuf	1
net.ipv4.tcp_mtu_probing	0
net.ipv4.tcp_no_metrics_save	0
net.ipv4.tcp_orphan_retries	0
net.ipv4.tcp_reordering	3
net.ipv4.tcp_retrans_collapse	1
net.ipv4.tcp_retries1	3
net.ipv4.tcp_retries2	15
net.ipv4.tcp_rfc1337	0
net.ipv4.tcp_rmem	4096 87380 3407872
net.ipv4.tcp_sack	1
net.ipv4.tcp_slow_start_after_idle	1
net.ipv4.tcp_stdurg	0
net.ipv4.tcp_synack_retries	5
net.ipv4.tcp_syncookies	1
net.ipv4.tcp_syn_retries	5
net.ipv4.tcp_timestamps	1
net.ipv4.tcp_tso_win_divisor	3
net.ipv4.tcp_tw_recycle	0
net.ipv4.tcp_tw_reuse	0
net.ipv4.tcp_window_scaling	1
net.ipv4.tcp_wmem	4096 16384 3407872
net.ipv4.tcp_workaround_signed_windows	0

Table A.5: System variables TCP settings.

System variable	Value
net.sctp.addip_enable	0
net.sctp.addip_noauth_enable	0
net.sctp.association_max_retrans	10
net.sctp.auth_enable	0
net.sctp.cookie_preserve_enable	1
net.sctp.hb_interval	30000
net.sctp.max_burst	4
net.sctp.max_init_retransmits	8
net.sctp.path_max_retrans	5
net.sctp.prsctp_enable	1
net.sctp.rcvbuf_policy	0
net.sctp.rto_alpha_exp_divisor	3
net.sctp.rto_beta_exp_divisor	2
net.sctp.rto_initial	3000
net.sctp.rto_max	60000
net.sctp.rto_min	1000
net.sctp.sack_timeout	200
net.sctp.sctp_mem	80640 107520 161280
net.sctp.sctp_rmem	4096 289500 3440640
net.sctp.sctp_wmem	4096 16384 3440640
net.sctp.sndbuf_policy	0
net.sctp.valid_cookie_life	60000

Table A.6: System variables SCTP settings.

List of Figures

List of Tables

Bibliography

[1] M. Welzl. How to truly improve the Internet's transport layer , September 2010. `http://heim.ifi.uio.no/michawe/research/publications/unitn-sep10.pdf`.

[2] R. Stewart, K. Poon, M. Tuexen, V. Yasevich, and P. Lei. Sockets API Extensions for Stream Control Transmission Protocol (SCTP). Internet-Draft (work in progress), July 2010. `http://tools.ietf.org/html/draft-ietf-tsvwg-sctpsocket-23`.

[3] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168.

[4] V. Jacobson. Congestion avoidance and control. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 314–329, New York, NY, USA, 1988. ACM.

[5] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), September 2009.

[6] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[7] L-A. Larzon, M. Degermark, S. Pink, L-E. Jonsson, and G. Fairhurst. The Lightweight User Datagram Protocol (UDP-Lite). RFC 3828 (Proposed Standard), July 2004.

[8] S. Floyd, M. Handley, and E. Kohler. Problem Statement for the Datagram Congestion Control Protocol (DCCP). RFC 4336 (Informational), March 2006.

[9] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), March 2006.

[10] S. Floyd and E. Kohler. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control. RFC 4341 (Proposed Standard), March 2006.

[11] S. Floyd, E. Kohler, and J. Padhye. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC). RFC 4342 (Proposed Standard), March 2006. Updated by RFC 5348.

[12] S. Floyd, M. Handley, J. Padhye, and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 5348, September 2008.

[13] S. Floyd and E. Kohler. Profile for Datagram Congestion Control Protocol (DCCP). Congestion ID 4: TCP-Friendly Rate Control for Small Packets (TFRC-SP). RFC 5622 (Draft Standard), August 2009.

[14] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960 (Proposed Standard), October 2000. Obsoleted by RFC 4960, updated by RFC 3309.

[15] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), September 2007.

[16] R. Stewart, M. Ramalho, Q. Xie, M. Tuexen, and P. Conrad. Stream Control Transmission Protocol (SCTP) Partial Reliability Extension. RFC 3758 (Proposed Standard), May 2004.

[17] R. Stewart, K. Poon, M. Tuexen, V. Yasevich, and P. Lei. SCTP: An Overview, Part 2: Protocol Details, April 2010. `http://www.sctp.org/SCTPOttawaPart2v1.0.ppt`.

[18] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581 (Proposed Standard), April 1999. Updated by RFC 3390.

[19] B. Ford and J. Iyengar. Efficient Cross-Layer Negotiation. In *HotNets-VIII: Eighth ACM Workshop on Hot Topics in Networks*, New York, NY, USA, October 2009.

[20] D. Wing, A. Yourtchenko, and P. Natarajan. Happy Eyeballs: New Tech to HTTP. Internet-Draft (work in progress), March 2010. `http://tools.ietf.org/html/draft-wing-http-new-tech-00`.

[21] Marshall T. Rose. *BEEP: The Definitive Guide*. O'Reilly Media, 2002.

[22] D.C. Schmidt. The ADAPTIVE Communication Environment: an object-oriented network programming toolkit for developing communication software. In *12th Sun Users Group Conference*, 1994. `http://www.cs.wustl.edu/~schmidt/PDF/SUG-94.pdf`.

[23] Douglas C. Schmidt and Chris Gill. Developing Efficient and Portable Communication Software with ACE and C++, May 2010. `http://www.cs.wustl.edu/~schmidt/PDF/ACE-examples4.pdf`.

[24] P. G. S. Florissi, Y. Yemini, and D. Florissi. QoSockets: a new extension to the sockets API for end-to-end application QoS management. *Computer Networks*, 35(1):57–76, 2001. Selected Topics in Network and Systems Management.

[25] James Won-Ki Hong and Robert Weihmayer. The networked planet: Management beyond 2000. In *7th IEEE/IFIP Network Operations and Management Symposium*, Honolulu, HI, USA, 2000. IEEE.

[26] Vitor Jesus, Susana Sargento, Miguel Almeida, Daniel Corujo, Rui L. Aguiar, Janusz Gozdecki, Gustavo Carneiro, Albert Banchs, and Pablo Yá nez-Mingot. Integration of mobility and QoS in 4g scenarios. In *Q2SWinet '07: Proceedings of the 3rd ACM workshop on QoS and security for wireless and mobile networks*, pages 47–54, New York, NY, USA, 2007. ACM.

[27] M. El-Gendy, A. Bose, S.-T. Park, and K.G. Shin. Paving the first mile for QoS-dependent applications and appliances. In *Quality of Service, 2004. IWQOS 2004. Twelfth IEEE International Workshop on Quality of Service*, pages 245 – 254, Montreal, Canada, June 2004.

[28] T. w. Chen, Chen Krzyzanowski, M. R. Lyu, C. Sreenan, and J. Trotter. A VC-based API for renegotiable QoS in wireless ATM networks. In *ICUPC 97 - 6th International Conference on Universal Personal Communications*, San Diego, CA , USA, October 1997.

[29] Hasan Abbasi, Christian Poellabauer, Karsten Schwan, Gregory Losik, and Richard. A Quality-of-Service enhanced socket API in GNU/Linux. In *4th Real-Time Linux Workshop*, Boston, Massachusetts, USA, December 2002.

[30] Bernd Reuther, Dirk Henrici, and Markus Hillenbrand. DANCE: Dynamic Application Oriented Network Services. In *EUROMICRO '04: Proceedings of the 30th EUROMICRO Conference*, pages 298–305, Washington, DC, USA, 2004. IEEE Computer Society.

[31] Bernd Reuther and Dirk Henrici. A unified, protocol independent API for connection-oriented and connection-less protocols. In *8th World Multi-Conference on Systemics, Cybernetics, and Informatics*, pages 298–305, Orlando, Florida, USA, July 2004.

[32] C. Vogt. Simplifying Internet Applications Development - With A Name-Based Sockets Interface, February 2010. http://christianvogt.mailup.net/pub/vogt-2009-name-based-sockets.pdf.

[33] J.M. Winett. Definition of a socket. RFC 147, May 1971.

[34] W. Stevens, B. Fenner, and A. Rudoff. *UNIX Network Programming - The Sockets Networking API.* Addison-Wesley, 2004.

[35] Networking Services, Issue 5. The Open Group. Technical Standard, February 1997. http://www.opengroup.org/bookstore/catalog/c523.htm.

[36] R. Stewart. Re: [Tsvwg] one-to-many/one-to-one socket API extension for SCTP, February 2010. http://www.ietf.org/mail-archive/web/tsvwg/current/msg04475.html.

[37] SCTP for the Linux Kernel, February 2010. http://lksctp.sourceforge.net/index.html.

[38] DCCP implementation for Linux, March 2010. http://www.linuxfoundation.org/collaborate/workgroups/networking/dccp.

[39] Standard for information technology - portable operating system interface (POSIX). Shell and utilities. In *IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Shell and Utilities*, 2004. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1309816.

[40] Portable Operating System Interface (POSIX) Base Specifications, Issue 7. The Open Group. Technical Standard, December 2008. http://www.opengroup.org/bookstore/catalog/c082.htm.

[41] Bryan Ford. Structured streams: a new transport abstraction. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07, pages 361–372, New York, NY, USA, 2007. ACM.

[42] D. Velten, R.M. Hinden, and J. Sax. Reliable Data Protocol. RFC 908 (Experimental), July 1984. Updated by RFC 1151.

[43] C. Partridge and R.M. Hinden. Version 2 of the Reliable Data Protocol (RDP). RFC 1151 (Experimental), April 1990.

[44] R. Stewart and P. Amer. Why is SCTP needed given TCP and UDP are widely available?, March 2010. http://www.isoc.org/briefings/017/.

[45] L. Eggert and G. Fairhurst. Unicast UDP Usage Guidelines for Application Designers. RFC 5405 (Best Current Practice), November 2008.

[46] M. Mathis and J. Heffner. Packetization Layer Path MTU Discovery. RFC 4821 (Proposed Standard), March 2007.

[47] Linux Programmer's Manual, March 2010. http://www.kernel.org/doc/man-pages/online/pages/man7/udp.7.html.

[48] S. Floyd and E. Kohler. TCP Friendly Rate Control (TFRC): The Small-Packet (SP) Variant. RFC 4828 (Experimental), April 2007.

[49] S. Shalunov, G. Hazel, and J. Iyengar. Low Extra Delay Background Transport (LEDBAT). Internet-Draft (work in progress), October 2010. http://tools.ietf.org/html/draft-ietf-ledbat-congestion-03.

[50] M. Welzl, D. Damjanovic, and S. Gjessing. MulTFRC: TFRC with weighted fairness. Internet-Draft (work in progress), July 2010. http://tools.ietf.org/html/draft-irtf-iccrg-multfrc-01.

[51] Jon Crowcroft and Philippe Oechslin. Differentiated end-to-end internet services using a weighted proportional fair sharing tcp. *SIGCOMM Comput. Commun. Rev.*, 28(3):53–69, 1998.

[52] Fang-Chun Kuo and Xiaoming Fu. Probe-Aided MulTCP: an aggregate congestion control mechanism. *SIGCOMM Comput. Commun. Rev.*, 38(1):17–28, 2008.

[53] R. Stewart, Q. Xie, M. Tuexen, S. Maruyama, and M. Kozuka. Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration. RFC 5061 (Proposed Standard), September 2007.

[54] W. Stevens, M. Thomas, E. Nordmark, and T. Jinmei. Advanced Sockets Application Program Interface (API) for IPv6. RFC 3542 (Informational), May 2003.

[55] Florian Niederbacher. Beneficial gradual deployment of SCTP . Master thesis at University of Innsbruck, February 2010. http://www.flori.bz.it/sctp/download/thesis.pdf.

[56] S. Bradner. The Internet Standards Process – Revision 3. RFC 2026 (Best Current Practice), October 1996. Updated by RFCs 3667, 3668, 3932, 3979, 3978, 5378.

[57] J. Wray. Generic Security Service API Version 2 : C-bindings. RFC 2744 (Proposed Standard), January 2000.

[58] S. Chakrabarti and E. Nordmark. Extension to Sockets API for Mobile IPv6. RFC 4584 (Informational), July 2006.

[59] Marta Carbone and Luigi Rizzo. Dummynet revisited. *SIGCOMM Comput. Commun. Rev.*, 40:12–20, April 2010.