Using Transport Services instead of specific Transport Protocols

Paul Müller, Bernd Reuther, Michael Schneider and Andreas Schulz

304/00

Using Transport Services instead of specific Transport Protocols

Paul Müller, Bernd Reuther, Michael Schneider and Andreas Schulz
Computer Science Department, University of Kaiserslautern,
P.O. Box 3049, 67653 Kaiserslautern (Germany)

Abstract: For most applications the used transport service providers are predetermined during the development of the application. This makes it difficult to consider the application communication requirements and to exploit specific features of the network technology. Specialized protocols that are more efficient and offer a qualitative improved service are typically not supported by most applications because they are not commonly available. In this paper we propose a concept for the realization of protocol independent transport services. Only a transport service is predetermined during the development of the application and an appropriate transport service provider is dynamically selected at run time. This enables to exploit specialized protocols if possible, but standard protocols could still be used if necessary. The main focus of this paper is how a transport service could provide a new transport service provider transparently to existing applications. A prototype is presented that maps TCP/IP based applications to an ATM specific transport service provider which offers a reliable and unreliable transport service like TCP/IP.

1. Introduction

Today data networks are used for various communication tasks. The same network is used for non time critical transport of masses of data as well as for time critical voice communication using low bandwidths. For these and many more different communication tasks there are specialized applications available. The user must choose an application which offers an appropriate communication service from his point of view. Further the user may use different settings for the same application depending on his actual requirements. Different applications and different settings lead to various requirements to the transport services. For example applications for file transfer, for accessing web sites and for video conferencing have different requirements to the transport service. Further the selection of a specific video codec or quality level for the video conferencing application will change the requirements of this application.

Additionally the same communication tasks are performed under different environmental conditions. Here we use the term communication environment to describe the properties and status of the endsystems and network which influence the transport of data. The endsystems operating system, the available system services and hardware and the actual resource utilization within the endsystem is part of the communication environment. Further the network properties and the available network resources between the communication partners are also part of the communication environment. Therefore the service that could be provided by a transport service depends on the communication environment, because it provides the basis functionality for the transport services but may also limit the transport services. For example a transport service may provide some Quality of Service (QoS) only if the underlying network offers this functionality like ATM or RSVP. The processing delay within the transport service typically depends on the operating systems scheduling strategy and on the actual CPU load.

Therefore an ideal transport service should try to fulfill the specific application requirements under the given conditions of the communication environment. But the reality today is that the application requirements and the communication environment are considered only slightly. Most applications are using the same transport protocols independently from the application requirements and the communication environment. The reason for this is that the protocols which implement the services are chosen during the development of the application[1], i.e. when there is only limited knowledge about the application requirements and often no knowledge about the communication environment. Therefore protocols must be used which do not depend on specific environment properties and which are commonly available.

[1] In some cases the protocols are not choosen by the application developers, but by the developers of libraries or middleware software like CORBA.

The necessity for new protocol architectures to achieve a higher degree of flexibility has been addressed by several different approaches e.g. [ClTe90], [Haas90], [MaPe92], [ZiStTa93], [GeRö97]. The approach presented here increases flexibility by moving the determination of a transport service provider (i.e. a protocol stack offering a transport service) from the time of the application development to the run time of the application. The predetermination of protocols limits the service that could be offered to the application because protocols hide technical details of lower layers, but therefore transport service users (i.e. applications or protocol Layers 5-7) cannot benefit from features which are not supported by the protocols. For example an application that is aware of TCP/IP only, is not able to request any bandwidth reservation since standard TCP/IP does not offer this feature even if the underlying network provides this service. Further the predetermination of common available protocols prohibits the usage of specialized protocols which may be very suitable and efficient for specific tasks or network technologies, but may be applicable in some communication environments only. It is important to note that most applications do not depend on the predetermination of a transport service provider in order to function properly, they depend on the service which is implemented by the protocols only. For example a Web client will use TCP/IP to communicate with a Web server, whereby ‚http' is used as an application layer protocol. But neither the client and server nor the ‚http' protocol requires to run on top of TCP/IP. Only a connection oriented and reliable transport service is required.

A transport service user is therefore conceptually independent of a concrete implementation of the transport service, i.e. of the transport service provider (TSP). This makes it possible to choose the most appropriate TSP regarding the application requirements and the communication environment which will lead to more efficient and qualitative improved communication. Since most knowledge about the application requirements and the communication environment is available at run time, the TSP should be chosen during run time also. But it cannot be expected from application developers to handle lots of different protocols and their specific parameters, because this would be much too complicated. Therefore we propose to offer protocol independent transport services for application developers which should be used instead of predetermined protocols. The services in turn will select an appropriate TSP during the run time transparently for the application. To realize such services two major questions have to be answered:

1. Which properties should interfaces for protocol independent transport services have?
2. How to choose a concrete transport service provider at run time?

In the next section some general aspects concerning these question are discussed. In the remainder of this paper we want to consider the questions mainly with regard to existing applications and systems. In the third section a concept for the realization of protocol independent transport services is proposed. Section 4 presents the prototype maps TCP/IP based applications to ATM specific transport providers. The throughput performance of that prototype is compared to TCP/IP in section 5. The last section draws a conclusion and describes further prospects.

2. Aspects of protocol independent transport services

First it should be considered how transport services are provided for the applications today. Applications access these services by using standardized or system specific APIs, for example BSD sockets [Ste90], X/Open Transport Interface (XTI) [XNS97] or Winsock [WS2]. On the one hand some specifications claim that the interface is protocol independent (e.g. [XNS97]), but on the other hand there are protocol specific extensions for each supported transport service provider. Most functions offered by the APIs are indeed protocol independent like *read* and *write* which could be used the same way for all protocols. Only few if any functions are protocol specific like the *t_addleaf* function of XTI which is defined for ATM only. Anyhow the APIs cannot provide protocol independent services, because several protocol specific parameters have to be used. There are parameters to select the transport service provider when creating a service access point. The application must provide protocol specific addresses, for example IP addresses or port numbers which are meaningful for protocols of the TCP/IP stack only. Further the application may influence the behavior of the protocols by using several protocol specific parameters like „TCP_NODELAY". Therefore the parameters used by an application determine which protocols are used.

The interfaces of protocol independent transport services may provide similar functions as provided by typical transport services today, but the parameters must be protocol independent also. This could be achieved by using protocol independent names instead of addresses[2] and by describing the application requirements instead of accessing protocol internal parameters. For example an application should request a property like „low delay for data transmission" instead of setting the „TCP_NODELAY" flag. It is the task of the service to translate the application requirements to appropriate protocol parameters after the protocols were selected. This means that he predetermined selection of an interface - which is necessary to build an application - equals to the selection of a service only and does not imply the selection of a protocol.

At run time a transport service could select a transport protocol by applying rules to the available information about the communication environment and the known application requirements. Since the application should provide its requirements to the transport service the main problem for choosing a specific transport service provider at run time is to find out enough information about the environment to be able to make a decision. These parameters may be automatically detected by the services or other system instances or may be configured by an administrator. Then rules based on these parameters could be used to choose an appropriate transport service provider. It is assumed to use existing complete transport service providers, it is not the task of the service to build new protocol stacks. For example rules may decide like this „if the application requires QoS and if ATM is available end to end and if specific ATM transport protocols are available then to use these specific protocols". For this decision the following parameters must be available:

- QoS parameters (given by the application at run time)
- Native ATM access of the local host (configured by an administrator)
- Native ATM access of the remote host (may be detected by requesting an ATM address of the remote system by using a name service like ANS [ANS96])
- Native ATM connectivity end to end (may be detected by setting up a connection)
- Availability of the specific transport protocols on the remote host (may be detected by addressing the specific protocol on the remote host during connection setup)

3. A protocol independent service concept

For the realization of a protocol independent transport service we propose a concept as shown in Figure 1. There are several transport service providers which offer similar or different transport services. Instead of using a TSP the application will use a protocol independent transport service provider (PI-TSP). The PI-TSP offers also a transport service but it dynamically selects an appropriate transport service provider at run time. The decision for a specific TSP is done on the client side where the communication is initiated and the PI-TSP of the server must be able to accept data from any of its supported TSPs. The PI-TSPs have to perform a mapping between the application parameters and the protocol specific parameters of the selected TSP. In some cases it might be necessary to map some functions also. Note that one transport service provider could be accessed by more than one PI-TSP. This concept enables to predetermine a transport service during the development of an application without implicitly defining a concrete transport service provider at the same time.

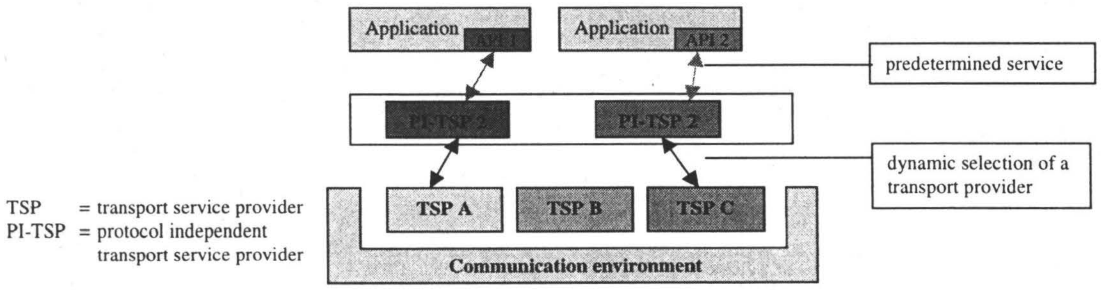
Application
API 1
Application
API 2
predetermined service
PI-TSP 2
PI-TSP 2
dynamic selection of a transport provider
TSP A
TSP B
TSP C
Communication environment
TSP = transport service provider
PI-TSP = protocol independent transport service provider

Figure 1: protocol independent transport service concept

In this concept transport services should offer a basic service which is defined by only few essential properties. If any service could be provided then these properties are guaranteed and cannot be changed on request. Further a service offers parameters to enable the request of optional properties. The service should try to fulfil these requirements, but this is not guaranteed. The basic service is needed by an application to work at all, for example the provision of reliable connection oriented communication or unreliable connection less

[2] A discussion of this topic is included in [RFC1900].

communication. The optional properties describe what is needed by an application to perform a task with sufficient quality, for example the 'http' protocol may request a "fast connection setup" because it typically setups and releases connections quickly. Having only few essential properties reduces the number of services, defining most properties to be optional avoids to build specific transport services which are applicable in few communication environments only.

The introduced concept described that a PI-TSP should offer a new API without protocol specific parameters. But only applications which are aware of such a new API could benefit from the protocol independent transport service. In order to support other applications also, a protocol independent transport service could provide its service transparently by defining an API which is backward compatible to a commonly used API. The API may offer additional functions which provide access to optional properties, for example to request QoS. Then only minor changes in existing applications are necessary to enable them to benefit from QoS capable networks if available. A PI-TSP that offers such a backward compatible API could be provided in parallel to a PI-TSP which offers a totally new API. A PI-TSP that emulates the API of an existing TSP has to map protocol specific parameters from that TSP to any other supported TSP. In this paper a mapping is presented between the BSD socket API and a proprietary ATM specific TSP.

Further the dynamic selection of a TSP must be transparent for remote systems that do not support protocol independent transport services and will typically use only one specific TSP. The local PI-TSPs therefore have to select the same TSP in order to provide a transport service at all. If the TSP used by the remote system is not known or cannot be determined, it suggests itself to use a commonly available TSP as a default, for example the TCP/IP suite. So it is possible to assure that at least the services of the default TSP could be provided. This is an improvement compared to use exactly one common available TSP which is done by most applications today.

4. Description of the prototype

The aim of the development of the prototype was to offer a new transport service provider transparently to the applications. Two services were implemented offering the basic services of reliable connection oriented communication and unreliable connection less communication which equals to the services of TCP/IP and UDP/IP. The new TSPs are especially adapted to ATM and could be used instead of the TCP/IP and UDP/IP in most cases when ATM is available end to end. Figure 2 shows the components of the prototype. A mapping has been implemented which offers an API that is compatible to the BSD socket API for TCP/IP. Yet there is no dynamic selection of the TSPs, so that the ATM specific TSPs are used in all cases. The dynamic selection would assure that a basic transport service could be provided even if the ATM specific TSPs cannot be used. The remainder of this section describes the two ATM specific TSPs, the BSD socket API and mapping of the BSD socket API parameters to the ATM specific TSPs.

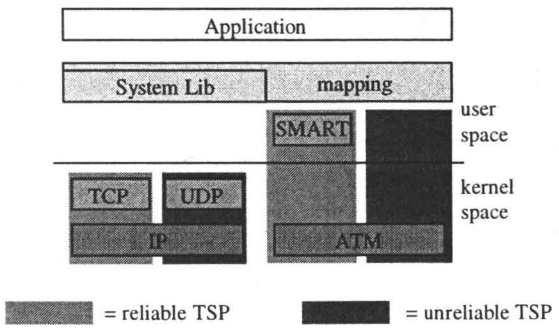

Figure 2: prototype architecture

4.1. ATM specific transport service providers

The new TSPs where realized on top of ATM which offers an unreliable connection oriented transport service. In order to map the connection less service to the connection oriented service offered by ATM, we implemented a connection management. The connection management establishes and releases connections, so that the connection oriented property of ATM is hidden for the application. In comparison to the service offered by UDP/IP some restrictions have been made. The unreliable TSP can only handle 1:1 communication relations. The 1:n and n:1 communication relations which are additionally supported by UDP/IP require a more complex connection management or a central server as implemented by the LAN-Emulation [LANE95], but many applications using UDP/IP typically build up 1:1 communication relations only. The management functionality is hidden in the mapping, since no additional protocol is required for this. The following description differentiates between the sender and the corresponding receiver. For the communication between client and server one ATM connection is set up and used in both directions to exchange data between client and server. If the sender wants to transfer data using the unreliable service for the first time, an attempt to set up an ATM connection to the desired peer entity is made and if the connection setup was successful, the data is transferred. In both cases the successful and the unsuccessful connection setup, the return value for the local application indicates a successful data transfer to rebuild the correct semantics of an unreliable service. If the receiver wants to get some data from the network and calls the receive function for the first time, the function is blocked until a connection request arrives. After the new connection is established, the receiver waits for incoming data and passes the received data to the calling application.

To provide reliable communication a retransmission protocol had to be used on top of ATM. For this the SMART (Simple Method to Aid Retransmission) [KeMo97] retransmission strategy was selected because it exploits some properties of ATM and therefore works efficiently over ATM. SMART uses heuristics which enable the fast retransmission of lost PDUs under the precondition that the ordering of PDUs is not changed during transmission which is guaranteed by ATM. Further SMART needs only one timer per connection which is used as a last resort to detect lost PDUs. This is an advantage since it is difficult to find appropriate values for such timers especially when high speed networks should be supported [Zah86]. SMART combines the best features of the traditional Go-back-N and selective-ack strategies. That is only lost PDUs will be retransmitted as for selective-ack, whereby Go-back-N may cause unnecessary retransmissions. But the exchanged status information is nearly as simple as for Go-back-N. For further details about SMART and its performance the reader is referred to [AKS96] and [KeMo97]. Here SMART is carried out as an independent process running in the user space (the Comprocess) on both participating machines, because it has to monitor the incoming and outgoing data streams always and asynchronously to the application[3]. When the Comprocess is started, the communication between the mapping and the Comprocess takes place by using the Unix pipe mechanism. The creation of the Comprocess at both communicating peers is done in the moment the connection is established, i.e. the call of the *accept* command at server site and the execution of the *connect* command at client site. TCPs sliding window mechanism for flow control was adopted to our implementation of SMART to achieve better performance and higher throughput.

4.2. BSD Socket Interface

This section gives a short introduction to the BSD Socket interface, an application programming interface used by a lot of applications on Unix systems to get network access. The BSD Socket API provides a set of commands that offer an easy way to get network access and the file descriptor semantics of the Unix file system. The common commands for file I/O are applicable to sockets. Additionally network I/O has to consider more details than file I/O and to deal with this there are some more commands to affect sockets.

The BSD Socket API offers sockets as service access points to the transport service user. To create a new socket both the client and the server of the distributed application use the *socket* command and obtain a descriptor to access the socket. The client and the server are able to associate an address to the new socket by calling the *bind* command. This is optional and could be done implicitly by executing the following commands (*listen* or *connect*). If the application uses UDP, the socket is in an active state and able to send and receive data after calling the *bind* command. But if TCP is used, the server has to do some more work to activate the socket. The call of the *listen* command moves the (server) socket in a state where it is able to accept requests for new connections. In order to get the next connection request, the server calls the *accept* command and gets the first entry of the connection request queue. At that time the new connection is established. In order to set up a new connection, the client has to send a connection request to the server by invoking the *connect* command. If the server accepts the incoming call, they are both able to send and receive data. To close a socket both the client and the server have to execute the Unix system call *close*. Information about and manipulation of the state of a socket is performed by several functions. To get the address of the local or the remote socket there are the two API calls *getsockname* and *getpeername*. In order to get infor-

[3] An implementation as a thread is not transparent to the application.

mation about the state of the socket the application uses the *getsockopt* call and to change the state *setsockopt*. The API provides the functions *send*, *sendto*, *sendmsg* and the Unix system call *write* to send data. Receiving data is possible by calling the commands *recv*, *recvfrom*, *recvmsg* or the system call *read*. Finally there are additional Unix system calls to affect sockets. The system calls *select* and *poll* look at descriptors and indicate the change of state of the descriptors. If the descriptor references a socket *select* and *poll* show new connection requests or the arrival of new data. Finally there are the system calls *dup* and *dup2* serving to duplicate a given descriptor. For further details about the BSD Socket API and Unix I/O control the reader is referred to [Ste90] and [Ste93].

4.3. Mapping of functions and parameters

The mapping is performed within the user space (see Figure 2) because of implementation reasons only. To use the mapping functionality it is therefore necessary to extend the source code of the application by an include statement and to recompile the code. In future the modification and recompilation of the code could be avoided by shifting the mapping into the kernel. The included file defines a set of macros which implement the mapping. Several system calls are modified by these macros. In the following the main focus will be the mapping of parameters since this is the intrinsic challenge for realizing a protocol independent transport service. Most functions could be mapped more or less straight to their counterpart of the ATM API or the SMART implementation respectively.

When a service access point is created by calling the *socket* command it must be detected which service is requested by the application. Only if the transport service of TCP or UDP is requested a mapping is performed, otherwise the normal system call is made (e.g. when opening a Unix socket). An application identifies the transport service provider by using the domain parameter AF_INET and the socket type SOCK_STREAM for TCP or SOCK_DGRAM for UDP respectively. According to these parameters a service access point of the reliable or unreliable ATM specific TSP is created.

A central problem for the realization of protocol independent transport service is the handling of addresses which are protocol specific. The BSD socket interface requires the specification of an address structure which is forwarded to the TSP without modification. An application may internally use protocol independent names instead of addresses, but these names must be translated by a separate name service to protocol specific addresses before it is used with the BSD socket interface. Additionally to the address of the remote host an address of the remote service or application must be provided. TCP and UDP are using a 32 bit IP number to address a remote host and a 16 bit port number to address the remote service/application. This address tuple must be mapped to adequate parameters of the ATM specific TSP.

ATM uses a more complex concept for addressing hosts and services. Each endsystem has at least one ATM address of 20 octets, whereby the first 19 octets identify the endsystem uniquely. The last octet (selector) may be used to address a service of the endsystem. Here the selector is treated as part of the endsystem address. Further the endsystems may[4] exchange two sorts of "Broadband Low Layer Information" (BLLI) and a "Broadband High Layer Information" (BHLI) [Q2931], each has a length of 64 bit. An additional type field specifies how to interpret these bits (e.g. for BHLI it is distinguished between ISO, vendor or user specific BHLI values). The BLLI specifies which Layer 2 (BLLI-2) and Layer 3 (BLLI-3) protocols are used, the definition of these values is not considered by the mapping. The BHLI is used to address a service or an application and is therefore a counterpart to the port numbers of the TCP/IP suite.

Therefore a mapping is provided between IP numbers and ATM addresses and between port numbers and the BHLI. Since IP and ATM addresses are assigned and used independently no direct mapping could be found. The addresses are therefore mapped to names by using a separate name service first. Then the protocol independent name is mapped to the other address. This method predicts that a name service is available and that the same name is bound to different address types. The mapping utilizes the domain name service (DNS) to map IP numbers to host names. The mapping of ATM addresses could be done by an extension of the DNS as specified in [ANS96]. Since this functionality was not available the mapping is performed by using an associative list, which is given by a well defined local file[5]. For each ATM connection the mapping stores information about IP and ATM addresses and host names, so that a subsequent address mapping could

[4] The usage of the BLLI and BHLI is optional.

[5] Like the standard Unix "/etc/hosts" file a implementation of FORE-SYSTEMS is available using a "/etc/xti_hosts" file for the mapping of host names to ATM addresses.

be performed faster. The proposed mapping between IP and ATM addresses makes it possible for the applications to use IP numbers furthermore. The actual used address format becomes transparent for the application. The mapping between the 16 bit port number and the 64 bit BHLI could be done directly. A draft document of the ATM-Forum specifies such a mapping [OCAG97] which is employed in this work (see Figure 3). The first four octets are constant values consisting of the ATM Forums OUI and the value 0x01. The fifth octet carries an identifier of a transport protocol, this is necessary because the service that is addressed by a port number depends on the transport protocol. Only two values for the protocol field are allowed here, 0x06 for the reliable TSP and 0x11 for the unreliable TSP. The same numbers are used by IP to identify the protocols TCP and UDP [IANA]. The octets 6+7 simply contain the 16 bit port number. The last octet is not used, in this implementation it is always set to zero in order to have a unique BHLI value of the full length. Note that the protocol field is not used to address a protocol it is needed only as a context to the port number. Also according to the draft specification of the ATM-Forum the BHLI type is set to "vendor-specific".

octet 1	2	3	4	5	6	7	8
AF OUI = 0x00A03E			0x01	Protocol	Port number		0x00

1-3:	Organisational Unique Identifer (OUI) of the ATM-Forum
4:	Native ATM Servcie API-Codepoint
5:	Protocol number ~ IANA protocol ID
6-7:	Port number as used by the TCP/IP suite
8:	not used, set to 0x00 to be unique

Figure 3: Mapping port numbers into BHLI [OCAG97]

Further protocol specific parameters like TCP_NODELAY[6], are not mapped to the ATM TSPs since there is no appropriate counter part for them. In order to emulate a correct behavior protocol specific parameters are stored, so that later requests could be answered with appropriate values. In order to be able to implement the *getpeername* function which returns the IP and port number of the peer application these numbers are exchanged during the connection setup.

The mapping described so far offers the same transport services than TCP/IP and UDP/IP only. Since ATM is used as a network layer here and ATM offers the feature to reserve an individual amount of bandwidth for each connection, a simple extension could provide this feature to the applications. For example by offering an additional *connect* function which accept parameters for the bandwidth reservation. Then the mapping may use the parameters if the selected TSP offers the feature of bandwidth reservation or it could be ignored otherwise.

5. Performance evaluation

Even though the main reason for the usage of specialized TSPs is to offer new or improved service features, their performance should be at least as good as that of the TCP/IP suite. Here we present throughput measurements with the netperf application [Jon] comparing the performance of the ATM specific TSPs and TCP/IP over LANE. Netperf was able to use the ATM specific TSPs just by recompiling it after adding an include statement to the source code.

The tests were performed on two Ultra 1 (140 MHz) Sparc machines running Solaris 2.6 with Fore ATM Adapters SBA-200 using driver version 5.0.0. Both machines were connected to an ATM switch with 155 Mbit/s. To avoid the influence of caching mechanisms, the application SDU was modified between two send operations. In order to achieve good throughput results for TCP/IP and UDP/IP the TCP_NODELAY flag was set for TCP tests and the socket buffer was set to 64 KB (larger than any application SDU used) for TCP and UDP. The measurements were done using different SDU sizes whereby each test lasted 60 seconds. Beside a "idealistic" test - where no additional application or network load influences the test - throughput measurements were made, when another process competes for the CPU resource and when there is additional traffic on the network.

Figure 4 compares the throughput results between the ATM specific TSPs and TCP/IP over LANE in the best case. The ATM specific TSP implementation achieves always higher throughput results than TCP/IP over LANE. Depending on the SDU size the reliable ATM TSP achieves up to 60% more throughput than

[6] More exactly: TCP_NODELAY is an API specific parameter which could be applied when using a specifc TSP only.

TCP/IP over LANE. Further tests have shown that for SDUs smaller than 512 bytes TCP will perform better without the TCP_NODELAY flag, but the results are still lower than that of the ATM specific TSP. For the unreliable service the data rate of the sender and the throughput (receiver side) is distinguished, the difference between both is the loss rate. Remarkable are the very good throughput results of the ATM specific TSP for packages larger than 4096 bytes. A throughput up to 133 Mbit/s is achieved, which comes close to the theoretical limit of ~135,7 Mbit/s[7].

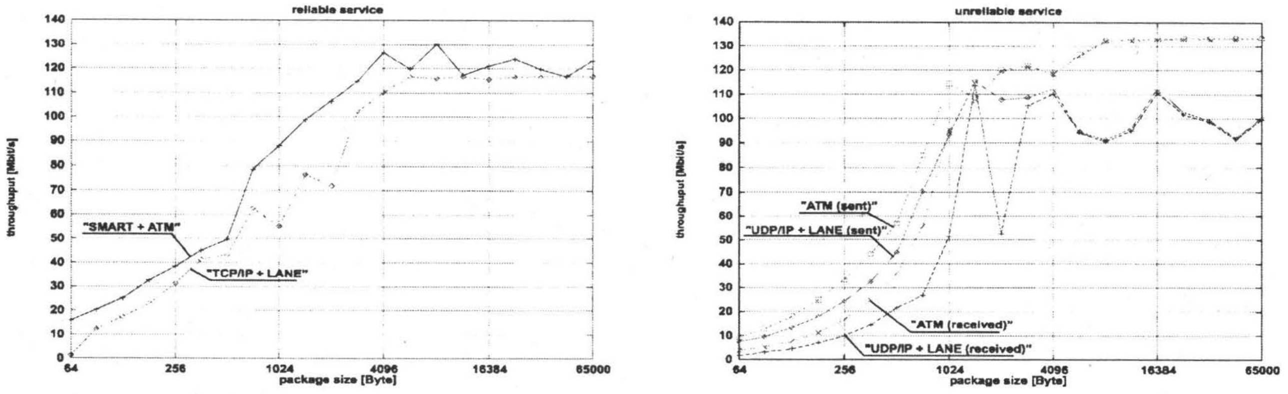

Figure 4: Throughput comparison between ATM specific TSPs and TCP/IP over LANE

Figure 5 shows the throughput results when an additional process at the endsystems competes for the CPU resource. The competing process tries to occupy 50% of the available CPU time. Considering the reliable services Figure 5 shows that our implementation of SMART over ATM achieves remarkable better throughput than TCP/IP+LANE over ATM for small SDUs and that the throughput of the ATM specific TSP as well as of TCP are limited almost at the same level for larger packages. Note that SMART implementation was running in user space, better results could be expected if implemented in kernel space. The throughput of the unreliable ATM based service is decreased by the disturbing processes while the loss rate is on a relative low level. For UDP over LANE considerable fluctuations of the throughput and loss rate are observed in our tests for SDUs equal and greater than 2048 bytes.

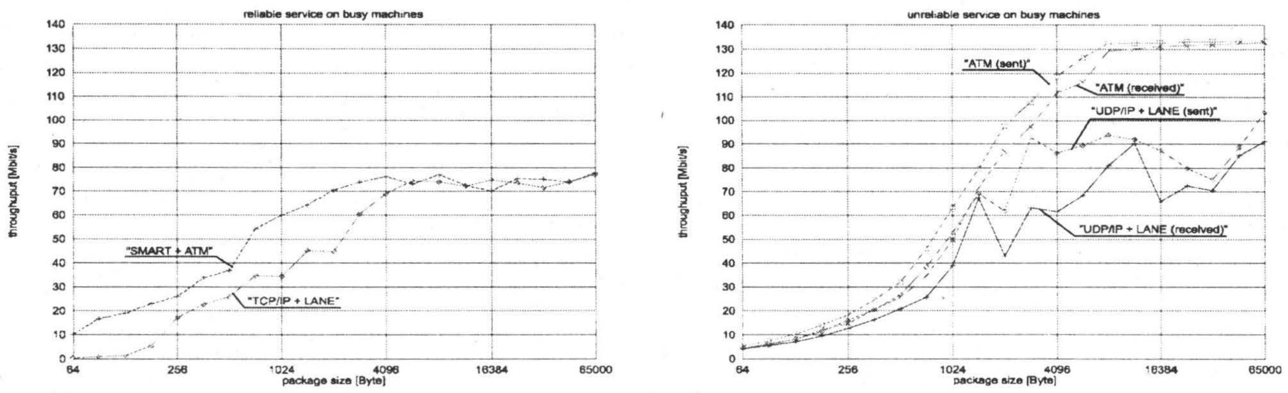

Figure 5: Throughput comparison with additional CPU consuming processes

In Figure 6 measurements with additional network traffic are presented. Both reliable services achieved almost the same throughput which is limited by the constant network load. The unreliable LANE based service performs bad when fragmentation becomes necessary for SDUs larger than the MTU of the emulated Ethernet. In this case the early packet discard mechanism of the switches is no longer able to recognize SDUs of the application when LANE has to perform fragmentation. Therefore larger SDUs are discarded only partially which reduces the available bandwidth and results in higher loss rates.

[7] 155.52 Mbit/s line speed - ~3.7 % SDH overhead ≈ 149.77 Mbit/s - ~9.4% Cell Header overhead ≈ 135.7 Mbit/s

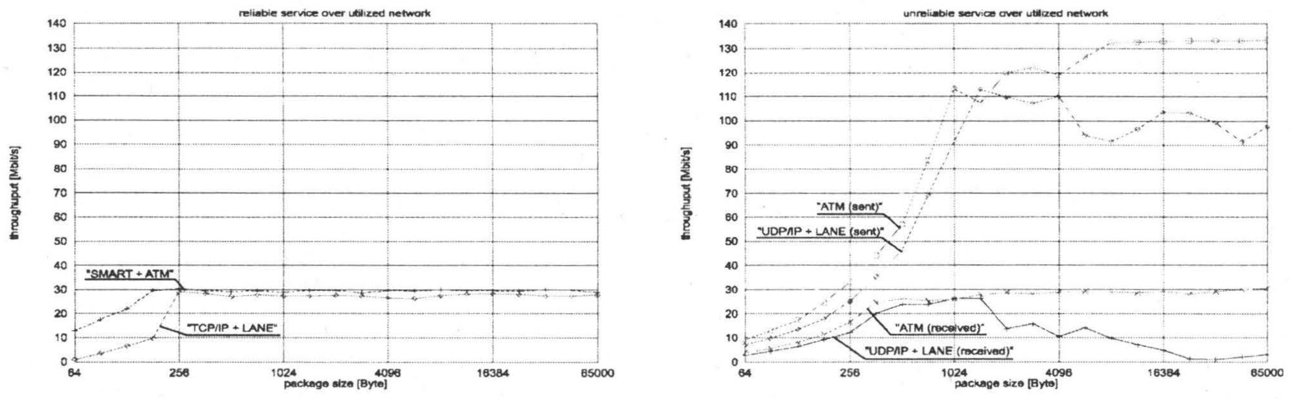

Figure 6: Throughput comparison with constant network traffic

6. Conclusion and Prospects

In this work a general concept for the realization of protocol independent transport services was introduced. A prototype was presented which enables the usage of specialized proprietary transport service providers by applications which were designed with the intention to use TCP/IP. Since broadcasts and unicasts are not realized not all UDP/IP bases applications could be supported. Nevertheless the prototype demonstrates that it is possible to separate the decision for a transport service and for a transport service provider which is essential for the realization of the proposed concept. The performance evaluation shows that even applications which use a basic transport service may benefit from specialized transport service providers because of their improved efficiency.

More work has to be done to realize the proposed concept completely. The dynamic selection of a concrete transport service provider has to be implemented, which will be easier when all transport service providers are implemented within the kernel. Further mechanisms may gather information in order to support the decision for a transport service provider. Also an interface should be defined which uses protocol independent parameters only.

References

[AKS96] R. Ahuja, S. Keshav and H. Saran: "Design, Implementation and Performance of a Native-Mode ATM Transport Layer", IEEE Proceedings Infocom 96, March 1996.

[ANS96] ATM Forum: "ATM Names Service", af-saa-0069.000, November 1996

[ClTe90] David D. Clark, David L. Tennenhouse: "Architectureal Considerations for a New Generation of Protocols", Computer Communications Review, Vol. 20, No. 4, Sept. 1990

[GeRö97] B. Geppert, F. Rößler: "Generic Engineering of Communication Protocols - Current Experience and Future Issues", 1st IEEE International Conference on Formal Engineering Methods, ICFEM'97, Hiroshima, Japan, 1997

[Haas90] Zygmunt Hass: "A Communication Architcture for High-Speed Networking", INFOCOM'90, Vol. 2, IEEE Comp. Soc, Wash. D.C., June 1990, pp 433-441

[IANA] Internet Assigned Numbers Authority, www.iana.org

[Jon] Rick A. Jones: netperf a tool for network benchmarking, www.netperf.org

[KeMo97] S. Keshav and S.P. Morgan: "SMART Retransmission: Performance with Overload and Random Losses", IEEE Proceedings Infocom 97, April 1997

[LANE95] ATM Forum: "LAN Emulation over ATM 1.0" af-lane-0021.000, January 1995

[MaPe92] Sean W. O'Malley, Larry L. Peterson: "A Dynamic Network Architekture", ACM Transactions on Computer Systems, Vol. 10, No. 2, May 1992, pp 110-143

[OCAG97] ATM Forum: "ATM Forum OUI Codepoint Allocation Guidelines", ATM Forum/BTD-SAA-API-01.02, July 1997

[Q2931] ITU-T Q.2931: "Broadband Integrated Services Digital Network (B-ISDN) - Digital subscriber signalling system No. 2 (DSS 2) - user-network interface (UNI) Layer 3 specification for basic call/connection control", 1996

[RFC1900] B. Carpenter, Y. Rekhter: " Renumbering Needs Work", RFC 1900, February 1996

[Ste90] W. Richard Stevens: "Unix Network Programming", 3rd Edition, Prentice Hall, 1990

[Ste93] W. Richard Stevens: "Advanced Programming in the UNIX Environmant", Addison-Wesley, 1993

[WS2] Windows Sockets 2 Application Programmer Interface, Revision 2.2.1, May 1997

[XNS97] The Open Group: "Specification Network Services (XNS) Issue 5", X/Open Document Number C523, 1997

[Zah86] L.Zahng: "Why TCP Timers Don't Work Well", Proceeding ACM SIGCOMM, August 1986

[ZiStTa93] Martina Zitterbart, Burkhard Stiller, Ahmed N. Tantawy: "A Model for Flexible High-Performance Communication Subsystems", IEEE Journal on Selected Areas in Communication, Vol. 11, No. 4, Mai 1993, pp 507-518