Towards a Protocol-Independent Internet Transport API

Michael Welzl
University of Oslo
Email: michawe@ifi.uio.no

Stefan Jörer
University of Innsbruck
Email: Stefan.Joerer@student.uibk.ac.at

Stein Gjessing
University of Oslo
Email: steing@ifi.uio.no

***Abstract*—The conjoint API of TCP, UDP, UDP-Lite, SCTP and DCCP has a large number of choices and is quite complex. By requiring an application to specify the name of the protocol to use, it also constrains the implementation of the Internet's transport layer. We propose to provide a common API that only offers the services that application programmers need to see, and we show how its design could be undertaken, ending up with an API that we believe is much simpler than the alternatives in use today. We also argue that this would be a very good way to get new transport protocols widely deployed.**

I. Introduction

Let us assume that UDP-Lite, SCTP and DCCP get deployed as common transport protocols, right next to TCP and UDP, available to everyone. Then, consider the choices that application programmers face: SCTP with partial reliability, using multiple streams or only one stream across an association? Or no reliability at all? But this is also what DCCP provides – however, DCCP also has various forms of congestion control. Should we pick TCP-like or TFRC congestion control? If we pick TCP-like congestion control, is it better to use DCCP or SCTP? DCCP has receiver feedback with various drop codes, and ACK congestion control. SCTP has partial reliability with a timer. What is more important for us? Do we want a checksum for our data? If not, would it be even better to use UDP-Lite and take care of congestion control in the application?

Given that these protocols are new and experimental and might often not even be available from one Internet endpoint to the other, is it really likely that this vast set of complex choices, reminiscent of fast-food restaurants, will convince application programmers to use them?

The example given above is no exaggeration: DCCP does not have a standardized API yet, and at the time of writing, the most recent Internet-draft specifying the SCTP API [1] has version number 23 and is 98 pages long. It seems clear that we should at least try to simplify access to these protocol's mechanisms if we ever want to see them broadly used. As a part of this simplification, and as an element of abstraction that has the potential to enable gradual deployment of new and innovative transport mechanisms, we should first of all hide the protocols themselves: why should an application programmer be even confronted with a decision between, e.g., SCTP and TCP, as opposed to having a system that automatically makes the best possible choice from the services that it has available?

Specifying such an API is a potentially endless task, as the design space is huge, and so are the possible debates that may ensue among designers who would each like to tackle the problem at their favorite layer of abstraction. As we will see later, there are also many unresolved issues regarding the internals of the "system" underneath such an API – "automatically making the best possible choice from the available services" is easier said then done.

A gradual approach is perhaps the only solution. We can start by defining an API that is not very abstract at all and just simplifies the currently available APIs, removing or combining functions that are not necessary for an application to see with rigorous arguments. We can sketch what kind of functionalities should be placed underneath the API without requiring their implementation: adding them later will only improve the performance of the overall system, i.e. better provide the service that the application asked for. This is possible if, and only if, we stay away from QoS requirements at this stage and rely on the best-effort models that SCTP, DCCP, TCP, UDP and UDP-Lite build upon.

In the next section we have a closer look at currently available transport APIs as well as some proposed future transport APIs. Then we will sketch our proposed design of a common transport API in section III. It is only natural that it cannot be perfect, and some of our arguments may be met with heavy disagreement by some colleagues. Even then, we have a discussion that is based on a concrete proposal – a starting point. We complement our arguments with some details about a possible implementation in section IV, and section V concludes.

II. Related work

The socket API has been extended for SCTP [1], and it also supports DCCP on some operating systems. The XTI API, which was designed for the ISO/OSI model, also gives transport users considerable independence from the underlying transport provider [2]. Both the socket API (with all its options) and XTI are quite complex and do not give the user a simplified abstraction of the transport services; hence, they are not uniform over the available transport protocols.

The name-based sockets focus on the problems that arise from using host mobility, multi-homing, firewalls and NATs [3]. In addition they enable a simplification of the socket API by removing the distinction between connection-oriented and

978-1-61284-955-3/11/$26.00 ©2011 IEEE

	ACE	DANCE	Name-based Sockets	QSocket	Socket	SST	XTI
unify connection-oriented and connection-less	x	x	x	x		(x)	
support of new transport protocols			(x)		x		(x)
configurability of features		(x)	(x)	(x)	x		(x)
support of QoS		x		x		(x)	

TABLE I
OVERVIEW OF TRANSPORT APIS. X-MARKS IN BRACKETS MEAN THAT THE API PARTIALLY SUPPORTS THIS FEATURE

connection-less sockets and propose a uniform API across transport protocols. The addressed problems are important and interesting, but mainly orthogonal to our research.

Structured Stream Transport (SST) enhances the traditional stream abstraction with a hereditary structure, allowing applications to create lightweight child streams from any existing stream [4]. SST, however, requires a new transport protocol which is not standardized.

Other APIs are mainly more high level or consider quality of service as well. A notable API for object oriented programming environments is the *Adaptive Communication Environment* (ACE) [5], and, for QoS, *QSockets* [6] and DANCE [7].

Table I gives an overview of the mentioned APIs. All APIs except Socket and XTI are uniform for connection-oriented and connection-less protocols. The socket API, which truly supports all currently available transport protocols, is for that reason the only one which supports full configurability of the features. QoS is supported by the two mentioned QoS-like APIs – DANCE and QSocket. SST could be seen as offering some kind of QoS too, by allowing the user to prioritize streams.

All of these above mentioned APIs are rather sophisticated, and the more abstract ones are generally more complex. The thesis of our paper is that it is possible to simplify and unify the API over the current Internet transport protocols based on the socket API. We have decided to build upon this API because it is the most popular one and currently the only one that supports all transport protocols.

III. DESIGN

We start our design by identifying which protocol features or services SCTP, DCCP and UDP-Lite provide. In case of SCTP, which provides the largest variety of services to its users, there are several tutorial-style papers that list its capabilities in tables, comparing it against TCP and UDP. We decided to rely on [8] because it seems to have one of the longest lists and is co-authored by one of the protocol's designers. For DCCP, our source is RFC 4340 [9], and for UDP-Lite it is RFC 3828 [10]. We then approached our simplification task in two steps.

A. Step 1: Analysis of transport services

We started out with the combined list of all services the three protocols provide. Then we removed entries where we believed that it is easy to argue that these are not really services from an application's point of view. They are:

- **Full duplex**: Since all current transport protocols support this feature, there is no choice to be made for the application.
- **ECN capable**: ECN is a part of congestion control and can be used by all protocols that have this function, irrespective of the form of congestion control used.
- **Selective ACKs**: The usage of selective acknowledgements is also a part of the implementation details of congestion control.
- **Path MTU discovery (PMTUD)**: Although [8] mentions that UDP itself does not provide this feature, PMTUD can be used with any packetization layer as specified in RFC 4821 [11]. For example, the current Linux implementation performs PMTUD for UDP and does not allow a sender to send larger datagrams than the current MTU [12]. DCCP, SCTP and TCP support this feature. Hence we do not need to distinguish services based on this; the current PMTU can always be retrieved through the API.
- **Application PDU fragmentation**: Fragmentation of one application PDU into smaller packets can be performed for all transport protocols. Hence, there is no need for the user to specify this.
- **Protection against SYN flooding attacks**: This makes the protocol robust against such kinds of attacks, but does not add a distinguishable service for a transport user (i.e. it is hard to imagine that an application designer would say "no, I do not want that").
- **Allows half-closed connections**: This is a protocol-internal detail, and a historical remainder rather than a transport service, hence 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 included in multi-homing.
- **Pseudo-header for checksum**: This is a protocol internal mechanism which does not add any kind of service.
- **Time wait state**: This is also a protocol internal detail which does not add any kind of service.

This resulted in Table II, which, in order to identify all available unique transport services, we expanded into a list with a line for every possible combination of transport features. This list, which we have not included due to space constraints, has 43 lines.[1] 32 are bound to SCTP, and only 3 services were derived from TCP and UDP together. The fact that over 93 % of all services are created by the more recently introduced transport protocols DCCP, SCTP and UDP-Lite, supports our assumption that in the future a new transport API is needed to facilitate usage of all these different services.

A network programmer looking up a service in this large list will need significant knowledge about the different features of these transport protocols in order to make the right choice. Therefore, an API that allows the transport user to specify a

[1]This list is provided in [13] along with further details of our design.

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	x	2/3/4		x/p1		m	u		
SCTP	x	x	x	0/1	x	t/p2	m	o/u	0/1	0/1

TABLE II

FEATURE OVERVIEW OF CURRENTLY AVAILABLE TRANSPORT PROTOCOLS AFTER THE ANALYSIS OF TRANSPORT SERVICES. EMPTY CELLS MEAN THAT A SERVICE IS NOT PROVIDED, X MEANS THAT IT IS ALWAYS PROVIDED. 0/1 MEANS A SERVICE CAN BE TURNED ON OR OFF. 2/3/4 MEANS THAT THERE IS A CHOICE BETWEEN CCIDS 2, 3 AND 4. P1 MEANS PARTIAL ERROR DETECTION, T MEANS TOTAL AND P2 MEANS PARTIAL RELIABILITY, S MEANS STREAM, M MEANS MESSAGE, O MEANS ORDERED AND U MEANS UNORDERED DELIVERY OF MESSAGES.

service rather than first choosing a protocol and afterwards tuning its features would significantly improve the work situation for network programmers. Moreover, by hiding the transport protocol from the application, the performance can gradually be improved by merely changing transport system underneath the API (in the OS), with zero programming effort for the application developer.

B. Step 2: Obvious / intuitive reductions

The next step was to reduce our list based on some very simple deductions. These reductions were achieved by removing unnecessary features or merging duplicate features and services. We define a feature to be "unnecessary" when there is no clear way for an application to decide whether it should use this feature or not. For instance, mechanisms which are only meant to improve the performance if they are used in the right way, but do not depend on application requirements, were removed.

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, i.e. the number of services provided by the transport protocols remains the same after the column "connection oriented" is removed. This simplification leads to a more uniform API. 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.

2) Removing Delivery Type: TCP is the only protocol that uses streams as delivery type. Except for the different delivery types, the two services provided by TCP in our list can also be realized by SCTP. We do not take the delivery type into account as a service because a message-based SCTP flow can easily imitate a stream-based flow towards the transport user; this is not a matter of the protocol's operation, but purely a matter of how the API exposes it. The delivery type is chosen via the used I/O function (message-based vs non-message-based I/O functions).

3) Removing Flow Control: The flow control feature is removed, because this and congestion control coincide in all current protocols. There is no service which has congestion control but no flow control. Whenever congestion control is specified, flow control will be applied as well. If a protocol is later developed that provides one without the other, we might want to reintroduce flow control.

4) Renaming Congestion Control: We do not distinguish between variants of congestion control for TCP-like flows because these variants do not change the service for the transport user (they just improve the performance under certain conditions). In order to show more clearly that Congestion Control and Flow Control always coincide, we introduce the term *Flow Characteristic* for this combined service. This should also emphasize that we 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 III in detail, are available.

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

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

Instead, the implementation of the new API should reuse already existing connections and automatically employ multi-streaming whenever this is possible. The new API works on a per-application-stream basis, where each socket represents exactly one stream. The transport user should not need to care about whether this stream is mapped onto a new connection (for example, for a requested service that needs TCP), or if a stream is used within an already existing SCTP association.

6) Merging duplicate services: Our list (expanded from Table II) shows that some services are duplicates in the sense that they are offered by more than one transport protocol – e.g., the flow characteristics of TCP and SCTP are the same. In this case, merging resulted in one service which can be

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 [14].	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 [14].
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 [15].	Therefore it is made more suitable for applications such as streaming media where a relatively smooth sending rate is of importance [15]. TFRC congestion control is appropriate for flows that would prefer to minimize abrupt changes in the sending rate [16].
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 [17].	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 [17].

TABLE III
OVERVIEW OF CURRENT FLOW CHARACTERISTICS, QUOTING FROM THE RFCS DESRIBING THEM.

service no.	*supported by transport protocol(s)*	flow characteristic	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/-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 IV
OVERVIEW OF REDUCED SET OF TRANSPORT SERVICES.
THE SECOND COLUMN IS NOT EXPOSED BY OUR API.
THE SYMBOLS IN THE TABLE ARE DEFINED IN THE CAPTION OF TABLE II.

provided by both of the two transport protocols.

Another reduction of two services was possible because, if UDP-Lite uses full coverage of the checksum, the provided service is exactly the same as the one provided by UDP.

Finally, if 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.

As a summary, Table IV shows the final set of transport services after applying the aforementioned reductions and simplifications. The number of services is reduced to 23. In addition, the number of features of a service is now only 6. *This means that the transport user now only has to decide on these* 6 *features instead of first choosing a transport protocol and afterwards deciding on various options.* Notice that the second column, the transport protocol name, is hidden in the implementation of the API.

The presented table could also be a starting point for the development of new transport protocols, because needed but currently not provided services can easily be identified. In total, there are 192 theoretical combinations of these 6 features, but only 23 of them are provided by today's transport protocols. The two mature protocols (TCP and UDP) offer only 3 services together.

IV. IMPLEMENTATION CONSIDERATIONS

We take the well-known socket API as a starting point for our API and adapt it with functions and options where needed.

Some of the reductions in step 2 require quite a sophisticated transport entity to be running underneath the API. In particular, the automatic usage of SCTP's multi-streaming has not yet been amply investigated. We have done some first steps in [18], but it is clear that there is still a significant amount of work to be done here. There also need to be rules for deciding which protocol to choose when more than one protocols realize a service, e.g. for services 1, 2 and 5 in table IV. In the following, we show some needed adaptions of the socket API and have a closer look at feature negotiation, which has to be solved by the API too.

A. Socket adaptions

Our API is based on the connection-oriented socket API, and also uses some ideas from the Internet-draft specifying the SCTP API [1]. The user has to pass an identification of the service needed as shown in Table IV. Alternatively the user could have passed the values of the six arguments, but we decided against that since it would be rather complicated to pass up to 6 parameters.

The creation of a socket looks like this:

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

We suggest using the prefix `PI_` (Protocol Independent) when identifying a service. Then, for example, services 1, 2 and 3 in Table IV could be `PI_TCPLIKE_NODELAY`, `PI_TCPLIKE` and `PI_NO_CC_UNRELIABLE`.

As mentioned in the first design step, 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 and `PI_API` as level argument.

Based on the capabilities that the underlying protocols have today, one feature in Table IV is ***static***, meaning that it is

decided upon at socket creation time and cannot be changed. Other features are also chosen at socket 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 used ***dynamically***, which leads to a service transition for a socket. All service characteristics are retrieved through `getsockopt()`. Moreover, the `setsockopt()` function enables the user to set the parameters of configurable features and change the dynamic features. As level argument, `PI_API` must always be used. Naturally all options of the socket level (`SOL_SOCKET`) are still accessible and of course affect the behaviour of our API.

*1) **Static feature**:*

Flow characteristic is chosen at socket creation time and cannot be changed afterwards. Hence all packets sent via the same socket use the same flow characteristic. The currently available flow characteristics listed in Table III do not have any parameters, and therefore this is a static feature.

*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 fully or only partially detect errors. 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.

Reliability is also chosen upon socket creation, with the possibility of changing parameters 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 [19]. We take only this form of partial reliability into account. 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 must also be chosen when a socket is created. The configuration of this feature is similar to the one used by the SCTP socket [1]. We offer the function `int pi_bindx(int sd, struct sockaddr *addrs, int addrcnt, int flags);` for maintaining the server side addresses. The details are not mentioned here due to space restrictions and can be found in [1] when looking at the `sctp_bindx()` function. With the option `PI_SET_PEER_PRIMARY_ADDR` one address can be marked as primary address.

*3) **Dynamic 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 the single messages, and that is why the user must have the possibility to turn it on/off. This feature can be turned on/off by setting the socket option `PI_NODELAY`.

Delivery order is a feature which can be defined on a per-message basis. Either the delivery order is ordered and hence sequential or it is unordered by setting the value of `PI_DELIVERY_ORDER` via the socket option.

Note that the modification of these features must be legal, i.e. the requested service must be available in our API; otherwise, the modification leads to an error. For example, if service 7 of Table IV 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.

Since we want to simplify the socket API, our API offers two send/receive mechanisms instead of the four different methods in the standardized socket API [2]. Our send/receive functions are shown in Figure 1; they are identical to the current version of the standardized socket API.

```
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);
```

Fig. 1. Send/receive funtions of our API. The first two functions are corresponding to message-based, and the others to stream-based send/receive mechanisms.

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 IV. The message-based delivery is usable for all services and allows the user to tune the delivery order via the `flags` field. 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 [20].

B. Feature negotiation

If a host wants to begin communicating with another host using a certain transport protocol, it must first be clear how the other host is informed about this intention. With TCP and UDP, it suffices to use a well-known port; if there is an application listening on the port in question, the communication can be established. With more protocols, this procedure raises several new questions: should servers listen on the same well-known ports for multiple protocols? What if host A wants to use protocol X, but host B has a preference for protocol Y? What if that the protocol of choice does not pass through middleboxes along the path from A to B?

At least two proposals for addressing this issue have been brought forward: the authors of [21] have attempted to answer these questions with a dedicated negotiation protocol, which is extremely flexible, but leaves the question of what happens if a protocol's packets are dropped along the path unanswered. One straightforward, but maybe less flexible possibility, is to

simply try using multiple protocols in parallel with TCP as a fallback, as described in [22] for HTTP; this requires sending an initialization packet, and waiting for a response, for each protocol. Each method has its pro's and con's. Since it is not our goal to discuss this aspect of using other protocols than TCP and UDP, it suffices to note that it seems possible to solve this problem.

V. Conclusion

With the introduction of new transport protocols like UDP-Lite, DCCP and SCTP, the socket interface has become more complex. Other efforts to create APIs for transport services mainly add functionality, and introduce even more options.

In this paper we opened up for a discussion about going in the other direction. We have made a first attempt to show that it might be possible to reduce the complexity of the API to the transport services. The application programmer should not have to decide which protocol to use, only which services the application needs.

Since our proposed API stays within the boundaries of the Internet's "best effort" service model, implementing its functionality can be done gradually – it is even possible to map all services to TCP or UDP in a first step, at the cost of having suboptimal performance. This might even be the only way to get the new transport protocols widely deployed: with a service oriented API, applications do not use the new protocols explicitly, they simply gain a benefit when a new protocol becomes available. Our API is therefore also not restrained to the protocols that we have mentioned: if a new protocol can do a better job at providing a requested service, it can simply be used underneath the API. If, however, a new service is provided, the API must be accordingly extended; what we have then achieved a more application-oriented way of thinking about protocol design. Maybe we can see a glimpse of a future where our system developer will find it much easier to order transport services for her application program than order a meal from a wall size menu in a fast-food restaurant.

Future Work

There is still a significant amount of work to be done: the automatic multi-streaming behaviour should be investigated; we also need to think in detail about events and notifications for our protocol independent socket API which we just have started to implement. Another interesting element for further research is to consider combinations of the six identified features (Table IV) 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.

Our proposal also has some not-so-obvious limitations that need to be investigated – for instance, given that the code that is running underneath the API will have to take care of a number of sophisticated functions, will this prevent it from being used in embedded systems? Hiding protocol details may also be unacceptable for some applications that are tuned for high performance (e.g. interactive online games) – e.g., hiding whether TCP or SCTP is used will not work well for applications that care about the chosen congestion control mechanism, or performance differences between the implementations of these two protocols. In conclusion, we do not expect that all applications would immediately happily switch over to our new API – but we believe that offering it as an additional choice to programmers of new applications is already a significant step towards increased flexibility of the Internet's transport layer.

References

[1] R. Stewart, K. Poon, M. Tuexen, V. Yasevich, and P. Lei, "Sockets API Extensions for Stream Control Transmission Protocol (SCTP)," Internet-draft draft-ietf-tsvwg-sctpsocket-25 (work in progress), January 2011.

[2] "Networking Services, Issue 5," February 2010, http://www.opengroup.org/bookstore/ catalog/c523.htm.

[3] 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.

[4] B. Ford, "Structured streams: a new transport abstraction," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 361–372, 2007.

[5] D. C. Schmidt, "The ADAPTIVE Communication Environment - An Object-Oriented Network Programming Toolkit for Developing Communication Software," May 2010, http://www.cs.wustl.edu/ schmidt/PDF/SUG-94.pdf.

[6] H. Abbasi, C. Poellabauer, K. Schwan, G. Losik, and Richard, "A Quality-of-Service Enhanced Socket API in GNU/Linux," in *4th Real-Time Linux Workshop*, 2002.

[7] B. Reuther, D. Henrici, and M. Hillenbrand, "DANCE: dynamic application oriented network services," in *Proceedings of the 30th EUROMICRO Conference*, Washington, DC, USA, 2004, pp. 298–305.

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

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

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

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

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

[13] S. Jörer, "A Protocol-Independent Internet Transport API," Master's thesis, University of Innsbruck, 2010. [Online]. Available: http://heim.ifi.uio.no/michawe/research/projects/new-transport/

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

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

[16] 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), Mar. 2006.

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

[18] F. Niederbacher, "Beneficial gradual deployment of SCTP," Master's thesis, University of Innsbruck, 2010. [Online]. Available: http://heim.ifi.uio.no/michawe/research/projects/new-transport/

[19] 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.

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

[21] B. Ford and J. Iyengar, "Efficient cross-layer negotiation," in *Eighth ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, New York City, NY, October 2008.

[22] D. Wing, A. Yourtchenko, and P. Natarajan, "Happy Eyeballs: Trending Towards Success (IPv6 and SCTP)," Internet-draft draft-wing-http-new-tech-01 (work in progress), August 2010.