A Requirement-Based Socket API for a Transition to Future Internet Architectures

Abbas Ali Siddiqui, Paul Mueller \{siddiqui,pmueller\}@cs.uni-kl.de
University of Kaiserslautern, Germany

Abstract—The existing application programming interface (API) between applications and the network architecture is one reason that it is hard to deploy novel protocols into the network architecture. Coupling between applications and underlying protocols makes it almost impossible to change one without changing the other. Coupling can be loosened or resolved by not involving applications in protocols implementation details but, only in functionality necessary to establish a communication. This way underlying network can deploy novel or updated implementations of a functionality without needing to change the applications. Using intermediate abstraction layers is an approach to break the dependency between applications and network protocols. One of the major goals in future internet architectures is to be flexible enough to adapt to application’s requirements. In this paper, a requirement-based API is presented as an abstraction layer to make applications independent of network mechanisms, which also helps in the transition to future internet architectures.

Index Terms—API, Network Architecture, Requirements, Sockets, Object-Oriented Methodology

I. INTRODUCTION

The existing API was made with the assumption that the Internet supports a limited number of protocols and relies on applications to specify the exact protocol to use. The current API hinders the deployment of new transport protocols such as SCTP [13], DCCP [15] and new addressing schemes as an application is obliged to specify the address type (IPv4 [10] or IPv6 [14]). Stipulation of protocol by applications fosters tightly bound coupling, which forces the network stack to use that exact protocol rather than an improved version of a protocol or one that is more suitable with respect to network conditions. In order to deploy a new protocol in the Internet architecture, it is not enough just to change the application but it also requires modifying the API or using another API so that a user can unveil its intention to use the newly deployed protocol.

An application needs to be modified if it wants to use TCP [11] or UDP [12]. Peer addressing and address-resolution are part of an application, which makes an application address type dependent. Thus an application has to select a particular addressing type such as IPv4 or IPv6. Different addressing types require different socket so that there is a difference between IPv4 TCP socket structure and IPv6 TCP socket structure and same is true for UDP.

The call of “setsockopt” is an example of another dependency where protocol specific options such as TCP_NODELAY can be turned on or off, as options are specific to a protocol so that it is must for an application to know details about a protocol.

Currently, there are multiple existing APIs each developed for different transport protocols. If an application needs to switch transport protocol, it is not enough just to adjust socket options or to change addressing family but it is also required to use a particular API given for a particular transport protocol.

Abstraction is used for hiding complexity and to encourage flexibility. In our approach, we propose an API by which an application can send its requirements in an abstract form to the underlying network such that applications do not need to rely on specific protocols; process of selection would rather be handled by the network architecture. This triggers the requirement for the network architecture to be able to handle those abstract requirements from applications. Abstract requirements from applications also help to create an unified API so that a single API can be used for multiple transport protocols.

After motivating for the requirement-based API, next section describes related work, which is followed by the design considerations that is further divided into sub-sections. Section V describes the function signatures used in the proposed API, and later sections cover conclusion and future work.

II. MOTIVATION

The transition from the contemporary to a novel network architecture may not be done all at once, but step by step. Adaptation of the current network API arguably can be seen as the first step towards the transition, as the API is the only point of contact for an application to reveal its communication requirements to a network architecture. An API by which an application can send its requirements to a network architecture without being concerned which kind of code or protocol stack will be used promotes loose-coupling between an application and the underlying network architecture. On the one hand it eases the burden of an application by having least knowledge about underlying protocols and on the other hand, it makes applications adaptable to different kinds of network architectures without triggering any need of modification. From the provider (i.e. network architecture) perspective, it is easy to deploy novel protocols without being concerned about legacy support for the consumers. Aforementioned arguments are the motivation to work on a socket based API that to flexible enough to accommodate futuristic demands and still able to support a smooth transition.

Requirement-Based API Current applications are tightly coupled with the given protocols, though they only care about its connectivity demands are fulfilled. A Requirements-based
API will alleviate the developer from choosing a protocol or even a protocol specification. Instead, requirements will be communicated to the underlying network architecture. Using these requirements, explicit connection characteristics are requested for the new communication relationship. Requirements are specified in terms of effects/capabilities, an effect is a visible outcome of a functionality such as flow control functionality provides effect of transmission rate adaptation between two parties.

III. RELATED WORK

Developing concepts and techniques for more flexible networks have been a research topic for many years now, but with varying focuses. The current API is seen as one of the first hurdles on the way to a flexible network architecture. [6] presents an approach to extend the generic nature of the socket API by introducing new generic structures (e.g. addrinfo) and function calls. However poor programming practices and protocol-dependent supporting APIs have been indicated as major problems of the current API. Unlike our approach, in [6] protocols are still part of the structure in addition to socket addresses which have been expanded to accommodate wider varieties of addresses. In our approach an application doesn’t have information about protocols and doesn’t carry the responsibility of address resolution, but uses abstract addressing scheme (e.g. URIs [1]), which can be dealt by various network architectures.

Name-based sockets [17] focus on relieving applications from IP address management responsibilities by shifting responsibilities such as address discovery, address selection, middlebox traversal and address updates to the underlying implemented functions. In our approach, URIs are used to replace specific addressing, which can be further resolved by underlying functionality.

Protocol-Independent API [7] proposes a common API to support multiple transport protocols, a service abstraction layer is made between applications and protocols providers which is expressed by service numbers so that an application uses service numbers instead of using a particular transport protocol. The service numbers are derived from functionality provided by transport protocols if more than one protocol provides the similar service then service number is mapped to more than single transport protocol such as service number 1 is associated to TCP and SCTP.

IV. DESIGN CONSIDERATIONS

The following sub-sections define design considerations which are taken into account while proposing the approach.

A. Abstraction Level

The existing API deals with an abstraction level where implementation details are revealed to outside. Which makes harder to replace underlying functionality without having a similar implementation. The PIAPI [7] approach takes it to next level and express functionality by service numbers, same service can be covered by multiple protocols which makes it flexible enough to switch between different protocol stacks.

The following text compares the abstraction levels covered by PIAPI [7] approach (i.e. Service numbering) and the presented approach (i.e. requirements-based).

Inclusion & Exclusion of a Functionality: Requirements-based selection of protocols has benefit over service number based selection, if it happens in the near future that new feature(s) (i.e. in PIAPI, service is defined by features it covers) is added in to a transport protocol, it will invalidate many of the service numbers as old application will still hope to get only capabilities which have been represented earlier by this service number rather than having the additive functionality. Same thing will happen if some currently provided functionality will be obsoleted in future so that an application will not have expected service any more. An example scenario could be, assume an application desires to use current TCP/IP protocol features which is mapped to service number 1, and if in near future it happens that TCP/IP implementation includes authentication functionality which can not be turned off. Additional feature will trigger the necessity of inclusion of a new feature column in the service table, besides it will invalidate the service number 1 because, in addition to, providing the requested functionality it also gives functionality which is not demanded or desired by an application. In other scenario, where a feature is excluded, developer needs to traverse through list to find out an exact number of a service which supports all the desired features and then update the application. On the other hand, in requirements based selection, as asked service is forwarded in terms of requirements so that only a transport protocol will be selected which is best suitable for the purpose. And an application will not need to update requested communication characteristics in case of inclusion or exclusion of functionality from a transport protocol.

Know-How about a Functionality: In the service numbering abstraction it is must to know about the capabilities to map with a specific service number, well but it is also true in the presented approach as an application must describe desired capabilities nevertheless, user doesn’t care about service number and its mapping to service. So that, user doesn’t need to go through list to find out appropriate service number himself but this task will be left to the underlying network.

Inclusion of QoS: Parameters, which usually influences the quality of entire communication such as delay, bandwidth, jitter as shown in fig. 2. Using the requirement based abstraction makes it possible to integrate QoS parameters which is not part of the PIAPI abstraction.

Modification: Every time some transport protocol will be updated it will trigger the requirements to update the service-mapping table, but in requirements based approach translator will take care to select the best suitable transport protocol for the given requirements.

Setup Time: Service number based abstraction has an advantage in terms of time required to setup a connection as selection of a transport protocol is straight forward by directly accessing the transport protocol by given service number. But it also increases the work of a developer by giving him the responsibility to look through service table to find an appropriate service number with matching required and offered
features. If list of features augments in the future, it might not be efficient or even practical to identify a service number manually, it may require automation.

B. Support For Transition

Inflexibility of current Internet architecture has encouraged the researchers to come up with new paradigms for Future Internet architectures. Various approaches have been proposed to introduce a layerless architecture such as ANA [5], SONATE [9], SILO [4], RBA [2]. Support for transition from current Internet to future internet architecture(s) is one of the major goals of the proposed API.

Fig. 1. Network Architecture Interfaces

As shown in the figure 1, this API proposal enables multiple architectures to work in parallel without need of adapting an application with respect to an architecture. Nevertheless it stimulates the need of an additional component in the current Internet, which will translate the incoming requirements (i.e. represented as capabilities) to match with an appropriate protocol stack (e.g. TCP/IP, UDP/IP). The abstraction of requirements in terms of capabilities enables to use any of underlying architecture without changing the application however architecture must have comprehension of the incoming requirements.

C. Why Socket API

Two distinct approaches exist for the creation of a Future Internet API: A classical, procedural API that resembles the traditional BSD socket API, and an object-oriented approach. Both approaches form an interface to the application or the network via two distinct parameters: First, a naming parameter, that is used to either connect to a service on the network or to provide it, and the second parameter to specify communication characteristics.

The former tries to keep as much of the traditional semantics as possible, without adhering to the principles that are likely to be changed in a Future Internet architecture like IP addressing or the explicit choice of a protocol. This approach rather borrows the original function names for connection establishment, “listening” to provide a service and the actual communication. This means that open, listen, read, etc. will still be named that way and roughly perform the same operation, however, they won’t take a packet socket address (i.e., created with inet_aton).

Communication is done by means of a file descriptor, following the standard UNIX approach. Here, the descriptor itself cannot be used to draw conclusions of the underlying system. The descriptor, which itself is pointing to an entry in the descriptor table, could as well be pointing to an open file on a local file system. This way, the new API will at the point not intrude the general OS’s behavior, but instead still act in the same way as the other UNIX I/O subsystems do. This also underlines the fact that although a new network communication stack is being created, the internal workings of it are hidden to the application developer thus following the goal to alleviate the developer from the need posses knowledge about a network’s interna.

The traditional semantics are also kept in respect to the general workings of the input/output subsystem. Calls to read, write and accept are still blocking, unless the file descriptor or socket as marked as being non-blocking via fcntl or during the initial call to socket. Via polling or the use of the select system call, the availability of data or incoming connections can be checked.

This approach clearly favors simplicity over versatility. Adhering to the traditional functions naming scheme will aid in the transition to a Future Internet architecture: The function names are already known to most programmers since the Berkeley Socket API is the current de-facto standard in this regard. Also, the functions still perform the same general operations: socket is used to create a new socket, connect() connects to another server or remote service, and so on. Also, the return values will follow the same semantics. Like the BSD API, a negative return value for the socket descriptor or the number of bytes read or written will indicate an error. The exact reason for the failure can still be obtained by means of checking a error number variable as it is done today when including the errno.h header file and checking against the errno variable. However, the error codes required here cannot be mapped to the POSIX error codes. They need to inform the application developer about things that were not part of the original POSIX specification, such as when the network stack is not able to fulfill the requested requirements.

However, the major drawback of this simple, yet elegant approach is that controls for more advanced means of communication which are already possible today are missing. Examples include (but are not limited to) multicasting [3] and streaming controls. These controls could be layered on top of it. But this would impose additional pressure on the underlying implementation, as it would not only have to be completely error-free in regards to the means of communication it was originally intended for, but also be carefully implemented with keeping the possible semantics and workings of more advanced concepts in mind during the implementation.

As this design is prone to errors, we advise to carefully extend this basic API with additional functions. These functions should take care of the work that goes beyond the scope
of this simplistic API, however, where possible, transparent overwriting of functions is suggested. For stream control, for example, the `read` and `write` functions should still be exposed, but not work in the traditional, connection-oriented way. Instead, where not applicable, they should return −1. The API itself should ensure that it is not “misused,” however, using it in the intended way is still a burden left to the developer: If a application developer is designing a streaming application, he or she will know — e.g., by reading the appropriate API documentation — that the `read` and `write` controls are disabled and the appropriate stream control functions — e.g., `play`, `pause`, etc. — should be used.

For a reference of the Berkeley sockets API, please consult [16] or the standard unix man pages, beginning with the `socket(7)` manpage. The different core functions are also described in the corresponding subsection later.

D. Address Resolution

A task that is not necessarily specific to an API approach is the task of how to name a particular service and how the naming resolution is to be done. Traditionally, socket addresses would be packed using a `struct sockaddr` C structure and the `inet_ntoa` function call. This approach is likely to be obsoleted in a Future Internet architecture, as IPv4 or IPv6 addresses will not remain the only (de-facto) standard addressing scheme. Keeping in mind that the network stack should now also take tasks upon itself which were not initially included in the Berkeley Socket API but instead needed to be built on top of it — e.g., streaming controls — the naming scheme of today that focuses on protocol addresses does not work anymore.

Instead, a more general naming scheme must be developed and advised. We propose a hierarchical naming scheme that is loosely based on URIs [1]. Here, IP addresses can still be named using a resource locator like `ip4://172.19.32.1`. However, the goal of the new naming scheme is to name not addresses, machines, or multicast groups, but in general establish a communication relationship to a resource that is accessible through the network. As such, video streams can as well be named via `video://some-video`, even by supplying additional parameters like a certain timestamp in the video.

This will of course make the introduction of an address resolution system an hard requirement for a Future Internet architecture using this naming scheme. But a current operating system installation already comes with several name resolution systems installed and enabled by default. The most widely known is DNS, the Domain Name System [8]. A UNIX system carries other means of resolution which can be transparently enabled or disabled using the `/etc/nsswitch.conf` configuration files. They are either plain text files, simple databases, access directory servers or make use of different means of resolution altogether. Also, Windows hosts employ similar means of name resolution.

Thus, the question of how to resolve names is already answered on any of today’s operating systems. Also considerations of where this resolution takes place — i.e., kernel space vs. user space — are also solved. The same design principles and considerations apply to name resolution in a Future Internet architecture. The difference lies only in the fact that name resolution is now no longer distinct part in an ecosphere of mechanisms used in conjunction, but integrated into the network stack and its usage hidden from the application developer.

E. Types of Requirements

Before identifying the types of requirements it worth to describe what requirement real is so that Requirements are one of two parameters an API approach needs. While the naming parameter (explained in the preceding subsection) specifies what service is accessed, the requirements parameter specifies how. Such requirements can be abstract, like “ciphering,” or specific, like “delay lower than 200ms.”

Requirements consist of three parts: An effect, an operator and an attribute. Effects denote certain characteristics, e.g. delay. It is important to emphasize that these characteristics are neither “good” nor “bad” but denote the outcome of an operation. Effect is thus a neutral term. “Delay” is an effect just as “packet loss” or “encryption” is an effect. To quantify an effect, attributes are used. An attribute is a numeric value like “200ms.” Effects and attributes are linked via operators, such as “equals,” “greater than,” etc.

One way to identify the basic requirements list is to review contemporary transport protocol stacks and their provided functionality. In PIAP [7], analyzing of transport protocols such as SCTP, TCP, DCCP, UDP has been carried out. Provided list of PIAP can be seen as an initial step towards finding appropriate requirements for the purpose, however, proposed names in the list are function oriented on the other hand we see the requirements in terms of effect (i.e. outcome/resultant of a functionality) which can be asked from an application. Following are the provided capabilities from the PIAPI list.

1) Flow Characteristic: Combined service of flow and congestion control
2) Application PDU bundling: Waiting for more messages and to send them all together
3) Error Detection: Mechanism to detect and correct errors
4) Reliability: Data delivery with no loss
5) Delivery Order: To ensure to receive messages in an exact same order in which they have been sent
6) Multi-Homing: To support multiple network interfaces at a single machine

Granularity of capabilities is not a trivial challenge and it may vary from domain to domain. Why should congestion control and flow should be defined in a single effect? it is a valid argument that in all current implementation they have been placed together and what if in the future that the new protocols will be developed where those effects will be implemented independent of each other, it would simply outdate the defined diction for the effects. Best possible granularity will be where implementation of effects can be independent of each other, thus an effect can be implemented without being dependent on other protocols’ implementation. If incongruous granularity will trigger the need to modify the capabilities list very often so that it may trigger to accommodate the outdated
applications too as they might be using the wrong diction for the given effect or it would require to have redundancy (i.e. simply adding new effects without deleting or altering the older ones) of terms with composed and atomic effects.

In fig. 2, list of minimum set of effects is presented. Provided effects in the list are viewed from a consumer (e.g. an application, a user) perspective, an application should not be concerned if an underlying network architecture uses delivery type either packet or stream, but it should concerns about if data can be securely or error free transmitted. Following is the explanation of some of the given effects.

Fig. 2. Effects: Requested from Application/User

1) Authentication: It is requested from an application or a user, in case it needed to authenticate the communicating partner. It should be clear that there are two aspects of authentication, one to authenticate oneself and one to authenticate the communicating partner. It is highly unlikely that a request will be made to authenticate oneself. So that only authentication of a communication partner is being addressed here.

2) Packet Boundary Preservation: An Application or a user doesn't care about that if data is delivered in datagrams or in a stream, but it would be required from some applications that packet boundary is preserved and recovered at the other end. If no boundary is defined then it can behave like a stream-oriented communication.

3) Error Ratio: It defines the error ratio even after correcting the error, as it is not possible to assure an entirely error free transmission. So that the name is chosen to give the freedom that how much error is corrected.

4) Traffic Prioritization: This functionality is asked by an application which is dealing with more than one kind of traffic, it is important to make it clear that an application doesn't have any influence on external traffic (i.e. traffic from other applications or users), so that the application may specify which kind of its own traffic it wants to prioritize.

5) Loss Ratio: This functionality covers reliability provided by TCP. The name has been chosen because of introduction of partial_reliability from SCTP protocol stack, where number of attempts or time-out period can be specified before packet is stopped to be retransmitted. As to say, it is unlikely to be possible to have full reliability, thus loss_ratio is used to define the ratio of the data loss in the communication.

6) Data Duplication Avoidance: This functionality is used by the application which is not able to deal with duplicated data itself.

Classification, shown in fig. 3, encourages the separation of concern. It is unlikely that an application or user will be interested in the functionality such as data reduction, connection management, loop avoidance. The shown functionalities in fig. 3 are of no importance for an application but many of them are essential for a communication such as addressing or connection management. Hence extra policies or requirements from an administrator or a network are needed, classification helps to identify the related functionalities for an administrator and a network.

Fig. 3. Effects: Application/User should not care about

V. FUNCTION SIGNATURES

This section will introduce the actual function/method signatures. They are denoted in a C-like syntax to ease reading for people proficient with Java, C, and other languages with a similar syntax.

The API can be broken down into three distinct parts, which denote the different phases of the communication flow. First, the client part that is responsible for connecting to a service and closing the connection at the end. Second, the former’s counterpart on the server side, registration of a service and the deregistration of a service, along with the ability to accept incoming communication requests. Third, the functions that are responsible for the actual data exchange and transfer.

```c
int socket()
```

Creates a new socket, suitable for both creating a new communication relationship and for providing a service. It serves as a transition function in as such that it does not take the same parameters the old BSD Socket API did (i.e., domain, type and protocol).

```c
int connect(int socket, char* address, req_t* requirements)
```
Creates a new communication relationship with a service. It takes the socket file descriptor created with the socket call, the address (i.e. which can be name of a service or traditional IP address, further resolution is the task of an underlying network) and a list of requirements. On an error, connect returns -1. If requirements could not be fulfilled, the requirements list parameters is modified as to include only the requirements that failed.

```c
int bind(int socket, char *service)
Binds a socket to a particular service.
```

```c
int listen(int socket, int backlog)
```
Announces a service to be available to the network.

```c
int accept(int socket)
Accepts the next incoming communication relationship request and returns a file descriptor for it. This call will block until a new request comes in (i.e., adhere to traditional semantics). If that is not desired, the poll function has to be used to check for incoming connections.
```

```c
int read(int fd, void *buf, size_t count)
Reads data from a file descriptor created by either accept or connect. Works as the standard read(2) system call.
```

```c
int write(int fd, void *buf, size_t count)
The opposite of read. Writes an amount of bytes to a file descriptor. Works just as UNIX’s write(2) call.
```

```c
int close(int fd)
Closes a file descriptor denoting a communication relationship.
```

VI. CONCLUSION

A requirements based API is proposed here to decouple applications from underlying network architectures. It also encourages to seamlessly switch between different network architectures which will help to deploy novel network architectures without having the need to change the applications or the currently used API. Requirements are an abstraction which can be dealt by various diverse architectures. Set of capabilities have been identified and classified for the separation of concern, this list can be further extended if new capabilities will be included in the future. QoS parameters can also be described besides functional requirements by using the same requirement description structure. Address resolution is not seen as a responsibility of an application. An application will rather use homogeneous URI addressing, further address resolution is taken care by the underlying networks architectures, resolution of a address may differ from one architecture to another. This way addressing is no more used as part of infrastructure rather like a mechanism which can be replaced by better one if necessary. The socket API semantics have been kept, being well known it can be relatively faster adapted by the developers.

VII. FUTURE WORK

The requirements section briefly outlined how requirements are used to specify and request certain specification of the communication relationship. Nevertheless, it is necessary to develop a translator to map those requirements with a protocol stack in the current Internet architecture or a service in a future internet architecture. The functional units will also have to be described along with their relationships and dependencies. This will lead to the creation of a Domain Specific Language. Also, the URI-like hierarchical scheme that is used for naming needs to be specified in more detail.

Finally, this paper presents the interface between an application and the network subsystem of an operating system. However, more components will want to interact with the OS’s network subsystem, like the system administrator or the network hardware. Thus, a System Programming Interface will also be needed in the future.

VIII. ACKNOWLEDGEMENTS

This paper has been created as part of the German Lab research project, founded by the Federal Ministry of Research, Germany (BMBF).

REFERENCES