Network Working Group
Internet-Draft
Intended status: Experimental
Expires: September 5, 2010

P. Sarolahti
HIIT/ICSI
March 4, 2010

Multi-address Interface in the Socket API
draft-sarolahti-mptcp-af-multipath-01.txt

Abstract

This document specifies a new address family to be used for sockets that are bound to more than one IP address, as motivated by the Multipath TCP work in the IETF. The goal is to use the same set of function calls as traditionally, but by new address family make it possible for them to express multiple addresses to connect or bind to. The document gives a high-level definition of the behavior of the traditional function calls, but a detailed specification of the API syntax is not in the scope of this document.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

The socket API is designed as a generic protocol-independent interface that includes a compact set of operations for network applications to communicate to the network. Despite the small number of basic operations, the socket API has proved to be enough powerful to carry out various different kinds of tasks, arguably because of the generic enough interface definitions. In addition to Internet communication, the socket API is also used, for example, in the Unix domain sockets that operate on Unix filenames as communication identifiers, and various control tasks between the applications and the communication stack, for example in IPsec key management sockets [RFC2367], or so called netlink or routing sockets that interact with the routing table in the network stack. The exact semantics of the socket API methods are not defined by the API itself, but depend on the parameters given when a new socket is created.

The socket API is designed to expose the addresses used for the communication to the applications. Consistent with the generic design of the API, the addresses are passed in a generic type-length encoded structure, that is interpreted based on the address family given in the beginning of the structure. In practice, primarily two address families are currently used for Internet communication: IPv4 (also known as AF_INET) or IPv6 (AF_INET6). Perhaps confusingly, traditionally the same constants used to indicate the address family are often also used to indicate the protocol family for creating socket. This dependency is discussed in more detail in Section 5.1.

This document proposes a new way to use the socket API to better support protocols that use multiple IP addresses at either end of a connection. The primary motivation for this specification is the ongoing work on Multipath TCP (e.g., [I-D.ietf-mptcp-architecture], [I-D.ford-mptcp-multiaddressed]), but the same API can be used with any other protocol that runs multiple addresses on a single socket. One of the design goals in this specification is to enable support for multiple addresses in a socket without changing the binary function call API at the operation system interface, or the set of networking function calls available in the system. This design also aims to maintain unchanged semantics with the previously familiar operations to the extent it is possible, while avoiding backwards compatibility issues by explicitly using a new address family.

Using Multipath TCP with a traditional single-homed socket API can be problematic, as discussed in the API considerations document [I-D.scharf-mptcp-api]. The socket API was designed with an assumption that a socket is using just one address, with this address being explicitly visible to the applications. When the API is used with a protocol that uses multiple addresses for communication,

defining the semantics of existing function calls that directly refer to one IP address becomes problematic, possibly making the existing applications behave defectively when using the legacy socket API with Multipath TCP. While the motivation of Multipath TCP to operate on unmodified legacy APIs is well understandable, eventually a more expressive API is needed to better manage connections using multiple addresses at either end.

This document specifies a new multipath-compatible address family to be used with the familiar socket operations, called AF_MULTIPATH. This address family is composed as a sequence of one or multiple elements that are each structured in the same way as one of the existing address families supported by the system, such as AF_INET or AF_INET6. At the same time, this lets the application indicate if it supports the use of multiple addresses for the socket, for example using multipath TCP. One advantage of the Multipath Address Family is that it supports using different address families, such as IPv4 or IPv6, in the same address set, thereby enabling dual-stack functionality between both IPv4 and IPv6 interfaces (although we note that IPv4 addresses can be expressed as a AF_INET6 structure).

The AF_MULTIPATH address family could be used also with other protocols capable of multihoming, for example SCTP [RFC4960]. It may possibly be applicable also to shim-layer approaches to multihoming such as SHIM6 [RFC5533] or HIP [RFC5206], although these are based on a different philosophy of splitting locators (IP addresses) from the host identity. Different API extensions for multihomed protocols have been specified (or are being worked on), for example one using a set of socket options [I-D.ietf-shim6-multihome-shim-api], and another extending the set of socket operations in the socket API [I-D.ietf-tsvwg-sctpsocket]. This document deliberately proposes a new approach as an alternative to these, and discusses the benefits and disadvantages of different approaches in Section 5.2

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. The Multipath Address Family

Multipath address family (AF_MULTIPATH) is composed of a sequence of addresses, each expressed using one of the existing address family formats supported in the system. A desirable behavior would be that in a system that supports the multipath address family, opening a

socket using one of the traditional single-address families should be taken as an indication that multiple addresses should not be used for that socket. However, to allow migration period for legacy applications that are not converted to use the new address family, but would benefit from multipath communication, an additional option switch may be needed to control the behavior on traditional single-address families.

The address family is structured according to the generic sockaddress structure as follows. The fields are given in network byte order.

- Length (8 bits)
- Address family (=AF_MULTIPATH) (8 bits)
- Number of addresses (8 bits)
- Address 1
- Address 2
- ....
- Address N

Each of the address records above takes the generic sockaddress format, i.e.:

- Length (8 bits)
- Address family (8 bits)
- Address (...i.e., the rest of sockaddr structure as defined by the address family...)

In a dynamic multipath connection the set of address fields can change over time: new addresses may be added and earlier addresses can be removed. This characteristic, that the socket address content can change during a connection, differs from traditional behavior. However, this just reflects the changed behavior of multipath TCP compared to traditional TCP, which uses the same pair of addresses through the connection life cycle.

In today's systems the address family is commonly either AF_INET or AF_INET6, although also other address families can exist. Depending on the address family, the address would typically be structured as sockaddr_in or sockaddr_in6 structure, with the length set appropriately. Different address families can be combined in a

single AF_MULTIPATH record.

4. Behaviour with Different Networking Functions

This section defines the intended behavior of commonly used network operations when used with AF_MULTIPATH address family. The section gives a high-level definition of the operations to be applied as appropriate in different application environments.

4.1. Bind

An AF_MULTIPATH socket can bind to several addresses using a single call. It is possible to use wildcard ("Any") address in some of entries of the address set. Technically, multiple "Any" addresses could allow binding several ports to the same socket, although it is unclear if there is any viable reason for doing so. AF_MULTIPATH can also contain just one address entry, in which case the behavior is similar to traditional single-homed bind. On return, the function call should indicate how many addresses were successfully bound, and use failure response to indicate that binding failed to all addresses. "Get Local Address" operation (getsockname in Posix) can be used to investigate which addresses were successfully bound.

Differing from its past use, bind can be called multiple times for the same socket, to allow the application dynamically change the set of local addresses. When a subsequent bind call does not include an address that is currently in use, it indicates to the protocol that this address should not be used anymore in a connection. When a subsequent bind call includes addresses that are not currently in use, it indicates that these addresses should be added to the connection. The protocol implementation may change the set of used addresses dynamically without a trigger from application. Before the bind call the set of currently used addresses can be obtained using the "Get Local Address" (getsockname) call as described in Section 4.4.

4.2. Connect

An AF_MULTIPATH socket can give multiple addresses to connect, assuming the addresses belong to the same host. The underlying protocol may need to activate these connections one at a time, if the protocol logic does not permit connecting to multiple addresses simultaneously. On return, the function call should indicate how many of the addresses were successfully connected, or an error code. It is expected that commonly this call is used together with name resolution, as described below. "Get Remote Address" (getpeername) operation can be used to investigate which addresses were

successfully connected to.

Similarly to bind operation, connect can be called multiple times for the same socket, to allow the application dynamically change the set of remote addresses. When a subsequent connect call does not include an address that is currently in use, it indicates to the protocol that this address should not be used anymore in a connection. When a subsequent bind call includes addresses that are not currently in use, it indicates that these addresses should be added to the connection (in practice triggering this from application can be unusual for remote addresses). The protocol implementation may change the set of used addresses dynamically without a trigger from application. Before the connect call the set of currently used addresses can be obtained using the "Get Remote Address" (getpeername) call as described in Section 4.4.

An application may give multiple addresses that seem reachable, but belong to different hosts. The underlying protocol that supports AF_MULTIPATH API MUST be able to detect such situation, and prevent connections to multiple hosts. Often there are sufficient protocol mechanisms (such as connection tokens in multipath TCP) and other protocol state that cause such connections to fail automatically.

4.3. Name resolution

In a typical usage of a name resolver, multiple addresses may be returned from a name server, and a client cycles through the given addresses until connection is successfully established. This is useful, for example, in dual-stack IPv4/IPv6 hosts. A client may need to try connecting separately to IPv6 addresses and IPv4 addresses, when it is not certain whether IPv6 is supported on the path.

When an AF_MULTIPATH-enabled name resolver is called, it returns the available address records as separate entries in a single AF_MULTIPATH structure. This would mean that the call returns a single AF_MULTIPATH host entry that may contain multiple addresses as specified in the AF_MULTIPATH format. An application may directly place the returned AF_MULTIPATH structure as a parameter of a connect call, indicating that a multipath protocol should try these addresses as subflows of the multipath connection. When a name resolver receives an AF_MULTIPATH-enabled call in a host that supports both IPv6 and IPv4, it is useful to invoke name server queries for both IPv6 and IPv4 addresses. If available, records of both types are returned to the application, that can pass them to connect call, in attempt to invoke a multipath connection over both IPv4 and IPv6 paths. This may be useful feature supporting transition from IPv4 to IPv6.

It is not uncommon that a DNS name is associated to multiple hosts, for example to perform DNS-based load balancing. As discussed above with the connect call, the underlying protocol implementation should be able to prevent connect attempts to separate hosts.

4.4. Get Local Address / Get Remote Address

The basic operation of these calls happens as before: they return a sockaddress structure either at the local or the remote end. AF_MULTIPATH address family is returned if it has been used earlier with the same socket. The set of local or remote addresses SHOULD be up-to-date with the currently active set in the protocol implementation. When the underlying protocol is able to change the active address set during the connection, this implies that subsequent calls to these functions can return different sets of addresses. Along with the current addresses, an application learns also about the connection identifier with this call. The connection identifier returned with "Get Remote Address" MUST be the same as the identifier returned with "Get Local Address", since it only has local meaning, and therefore the other end of the connection typically gets entirely different connection identifier for the connection.

5. Discussion

5.1. Address Family or Protocol Family?

The socket API defines separately the protocol family, that is used to define the semantics of the socket behavior when a socket is created, and the address family that is used to distinguish different socket address formats [RFC2553]. By convention, in many systems AF_INET is defined to have an equal value to PF_INET, and AF_INET6 is defined to have an equal value to PF_INET6, allowing the pairs of definitions to be used interchangeably in implementations.

Defining separate protocol families for IPv6 and IPv4 has some unfortunate consequences: in principle, it needs to be decided at socket creation time whether to use IPv6 or IPv4. In practice, IPv6 defines a specific address range, IPv4-mapped addresses, to "virtually" represent IPv4 addresses in IPv6 address space. Implementations can use this feature to signal, that IPv4 should be used on the wire. An alternative design choice could have been to use same protocol family for both IPv4 and IPv6, and distinguish use of IPv4 and IPv6 with the address family, that can be selected also after the socket is created. After all, the transport protocol implementation, and its semantics, are the same in both cases.

In case of AF_MULTIPATH, the format of the socket address structure

changes, while for example in the case of multipath TCP, the semantics of the socket calls are intended to remain unchanged. This would suggest that when AF_MULTIPATH address family is used with socket address structures, the protocol family definition should remain at its typical value. Here an argument can be made that AF_MULTIPATH changes the behavior of bind and connect calls, but on the other hand, it is correct to say that from the applications point of view the outcome of a successful completion -- choosing a local access point, or successfully establishing connection with the peer -- is exactly the same as with the traditional use of the API. It is unclear at this point, whether the pairwise definitions of protocol family and address family has lead to dependencies in implementations that would hinder the implementation of the AF_MULTIPATH address family, or whether such dependencies would be difficult to be fixed.

5.2. Comparison to Alternative Design Options

Replacing the current socket API with a "connect-by-name" API has been proposed. Different proposals have slightly different abstraction levels, but commonly in these APIs application passes a DNS name with the connect call. The benefit of such API is that the application does not need to handle the network addresses, that arguably shouldn't be application's concern in most cases, and it thus avoids a separate name resolution step. In a long term this seems a promising direction to take in API design, but involves inter-operation between the name resolver that is often implemented in user space in current systems, and may need changes in operating system kernel interface. This proposal intentionally has taken the traditional, more short term approach, to expose the network addresses to applications.

Additions to the set of calls in the socket API has been proposed, for example, alternative operations for multi-address bind and connect [I-D.ietf-tsvwg-sctpsocket]. In the beginning this document lauded the elegance of simple, generic socket API with a small basic set of operations, and addition of new purpose-specific operations would be a deviation from this design principle. In addition adding operations to socket API would cause changes to the operating system kernel function interface, that could cause interoperability and maintenance issues. One way to implement such extended operations would be through an user-space library that maps the operations to the existing socket calls in the kernel implementation. In mapping the additional operations to existing kernel interface, such library might leverage a mechanism similar to what presented in this document, or use, for example socket options or ioctl calls.

Socket options can be used tune parameters affecting the protocol behavior. The extensibility of the socket option format can make it

appealing to use this interface for more significant run-time tasks, such as adding or deleting addresses in a multi-address session, as done in [I-D.ietf-shim6-multihome-shim-api]. In this approach the semantics of the traditional single-homed operations still need to be specified. It can also be questioned whether it is appropriate to use socket options to trigger actions that can be seen to fall beyond the scope of the original meaning of "socket option".

In summary, introducing a new address family as proposed in this document allows keeping the existing set of socket operations in the API, which the author believes to be a useful property, for example concerning the maintenance of the interfaces between the operation system and applications. If an operating system does not support AF_MULTIPATH, it can gracefully return an appropriate error code to an application, that can then revert to use the traditional single-homed address family, if desired. There are no backwards compatibility issues with old applications, because applications explicitly signal their support of this address family with the connect or bind calls.

Presenting the multiple addresses associated with the connection using the socket addresses seems a natural and native way of expressing this fairly new kind of property to applications. The recent discussion on this idea showed, however, that the use of more dynamic socket addresses can be perceived as unconventional and can raise doubts, for example regarding the possible assumptions about the persistence of the address structure in the implementations. It is unclear how actual this concern is, given that the underlying protocol that dynamically maintains the set of addresses is a fairly new feature, compared to decades of past use of the end-to-end transport.

5.3. Open Issues

Below are listed some potentially open issues that need to be taken in to account in follow-up discussion on this document.

o Are there any constraints on the generic format of the socket address structure, that would conflict with what proposed above? It appears that for example BSD and Linux use different formats for this structure, so that the Linux structure follows the "old BSD" convention, without a common length field (sa_len). Are there reasons to follow one of these conventions, or would it be possible, for example, to specify a 16-bit length field that could be useful for large sets of IPv6 addresses?

o Is there a need to expose pairwise source-destination subflow associations, instead of just a group of source and destination

addresses? Currently no such reason can be seen: the socket should represent one logical connection between a source host and a destination host, that in this case may have multiple IP addresses in use for the connection.

- Is a separate connection identifier field needed, as in the earlier version of this document? Currently the author cannot identify such need.

6. Security Considerations

No additional security threats are known because of the multipath address family. This document referred to the possibility that dynamic end host multihoming may enable new ways to maliciously transfer a connection to another host. A multi-address interface may open this possibility to applications, but ultimately the multihomed protocol should have mechanisms to protect from such behavior.

7. Acknowledgments

The author is thankful to Alan Ford for pointing out specific technical issues to be addressed, and to the people who have participated the discussion on the MPTCP mailing list.

The author is supported by Finland-ICSI Center for Novel Internet Architectures (FICNIA) and the Finnish Funding Agency for Technology and Innovation (TEKES).

8. References

8.1. Normative References

[I-D.ietf-mptcp-architecture]
Ford, A., Raiciu, C., Barre, S., and J. Iyengar, "Architectural Guidelines for Multipath TCP Development", draft-ietf-mptcp-architecture-00 (work in progress), March 2010.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

8.2. Informative References

[I-D.ford-mptcp-multiaddressed]
Ford, A., Raiciu, C., and M. Handley, "TCP Extensions for

Multipath Operation with Multiple Addresses", draft-ford-mptcp-multiaddressed-02 (work in progress), October 2009.

[I-D.ietf-shim6-multihome-shim-api]
Komu, M., Bagnulo, M., Slavov, K., and S. Sugimoto, "Socket Application Program Interface (API) for Multihoming Shim", draft-ietf-shim6-multihome-shim-api-13 (work in progress), February 2010.

[I-D.ietf-tsvwg-sctpsocket]
Stewart, R., Poon, K., Tuexen, M., Yasevich, V., and P. Lei, "Sockets API Extensions for Stream Control Transmission Protocol (SCTP)", draft-ietf-tsvwg-sctpsocket-21 (work in progress), February 2010.

[I-D.scharf-mptcp-api]
Scharf, M. and A. Ford, "MPTCP Application Interface Considerations", draft-scharf-mptcp-api-00 (work in progress), October 2009.

[RFC2367] McDonald, D., Metz, C., and B. Phan, "PF_KEY Key Management API, Version 2", RFC 2367, July 1998.

[RFC2553] Gilligan, R., Thomson, S., Bound, J., and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC 2553, March 1999.

[RFC4960] Stewart, R., "Stream Control Transmission Protocol", RFC 4960, September 2007.

[RFC5206] Nikander, P., Henderson, T., Vogt, C., and J. Arkko, "End-Host Mobility and Multihoming with the Host Identity Protocol", RFC 5206, April 2008.

[RFC5533] Nordmark, E. and M. Bagnulo, "Shim6: Level 3 Multihoming Shim Protocol for IPv6", RFC 5533, June 2009.

Appendix A. Change log

Changes from version -00 to -01

- o Added more background to the Introduction

- o Added section to discuss protocol family/address family issues, and comparison to related API designs

- Removed the connection identifier from the address structure, and changed related descriptions related to socket calls.

- Added discussion about open issues in Section 3

Author's Address

Pasi Sarolahti
HIIT/ICSI
1947 Center Street (Suite 600)
Berkeley, CA 94704
USA

Phone: +1 (510) 409 - 9972
Email: pasi.sarolahti@iki.fi
URI: http://www.iki.fi/pasi.sarolahti/