TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: January 3, 2019
July 02, 2018

P. Tiesel
T. Enghardt
TU Berlin

A Socket Intents Prototype for the BSD Socket API - Experiences, Lessons Learned and Considerations

draft-tiesel-taps-socketintents-bsdsockets-02

Abstract

This document describes a prototype implementation of Socket Intents [I-D.tiesel-taps-socketintents] for the BSD Socket API as an illustrative example how Socket Intents could be implemented. It described the experiences made with the prototype and lessons learned from trying to extend the BSD Socket API.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

With the proliferation of devices that have multiple paths to the
internet and an increasing number of transport protocols available,
the number of transport options to serve a communication unit
explodes. Implementing a heuristic or strategy for choosing from
this overwhelming set of transport options by each application puts a
huge burden on the application developer. Thus, the decisions
regarding all transport options mentioned so far should be supported

and, if requested by the application, automated within the transport layer.

Socket Intents [I-D.tiesel-taps-socketintents] allow an application to express what it knows, assumes, expects or wants to prioritize regarding its own network communication. This information can than be used by the OS to perform destination selection, path selection and transport protocol stack instance selection.

Our Socket Intents prototype for the BSD Socket API is a first attempt to automate transport option selection within the OS. It is primarily targeted at path and destination address selection and tries to be as close as possible to the semantics of the BSD Socket API. The prototype mostly excludes the problem of transport protocol stack instance selection, which is more closely discussed in [I-D.tiesel-taps-communitgrany].

We implemented the prototype as a wrapper for the BSD Socket API that communicates to a central Multiple Access Manager that makes the actual decisions and can optimize across applications. The whole implementation was done in about 15k lines of C code. The code is available at Github [1] under BSD License.

This document describes our Socket Intents prototype for the BSD Socket API. It details important aspects of the implementation and the API variants we developed over time based on lessons learned. Finally, it summarizes these lessons and points out why the BSD Socket API is not particularly well suited to integrate automated transport protocol stack instance selection. Furthermore, it describes the limitations for destination address and path selection within the BSD Socket API.

2. Prototype Architecture

The Socket Intents prototype consists of the following components, also shown in Figure 1:

o The Socket Intents API, a BSD Socket API wrapper for applications to use, including a representation of the actual Socket Intents.

o The Socket Intents Library which implements the Socket Intents API. It sends requests to the Multiple Access Manager, e.g. before establishing a connection, and gets back a response regarding what interface to use.

o The Multiple Access Manager (MAM), a daemon which gets informed about all application requests and has knowledge of the available network interfaces.

- The Policy, a dynamically loaded library hosted by the MAM. It chooses which of the available interfaces to use based on the available knowledge about them and the Socket Intents.

- Data collectors that that reside inside the MAM and that provide information like bandwidth usage, smoothed RTT estimate and RSSI for wireless links to the policy.

```
    +-------------------------+
    |      Application        |
    |                         |                  +-------------------+
    +-{ Socket Intents API }-+  (MAM Request)    |  Multiple Access  |
    |                         | ----------------> |      Manager      |
    |     Socket Intents      |  (MAM Response)   | +---------------+ |
    |        Library          | <---------------- | |    Policy     | |
    +-------------------------+                   | +---------------+ |
    |      BSD Sockets        |                   | |Data Collectors| |
    +-------------------------+                   +-+---------------+-+
```

Figure 1: Components of the Socket Intents Prototype

3. Multiple Access Manager

The Multiple Access Manager (MAM) is the central transport option selection instance on a host. It is realized as a daemon that runs in userspace and receives requests from each application that uses the Socket Intents Library.

The MAM hosts the Policy, which is the actual decision making component, e.g., deciding which source address and therefore which source interface to use. Upon events, such as an application requesting to resolve a name or to connect a socket (see Section 5 for details), the Socket Intents Library issues a MAM request and the MAM invokes a callback to the policy - see Section 3.1 for details - which can either communicate its decision right away or defer its decision, e.g., when it has to wait for the results of name resolution. The results and decisions are communicated back to the Socket Intents Library through the MAM response, where they are applied to the actual socket, see also Figure 1.

To support the policy, the MAM maintains a list of IP prefixes that are configured on the local interfaces and available for outgoing communications. As destination address selection and path selection are highly dependent on each other, the MAM integrates DNS resolution and maintains separate resolver configurations per prefix (see [ANRW17-MH] for further discussion on multiple PvDs and DNS resolution). Furthermore, the MAM includes data collectors which

periodically gather statistics on the available paths, see Section 3.2 for details.

3.1. Policy

In the Socket Intents prototype, the Policy implements the decision logic for selecting among available transport options. In our current implementation, only one policy can be active at a given time. We implement different interchangeable policies as dynamically loaded libraries, which are hosted by the Multi Access Manager (MAM), see Figure 1. When launching the MAM, the user has to choose a policy and supply a policy configuration, which can contain additional information to configure the policy.

Examples of policy configuration include:

- o A list of IP prefixes configured on local interfaces to consider as source for the communication
- o Name server(s) to use for each of the IP prefixes
- o Preferences to instrument the policy, e.g., default prefix to use

The policy is initialized with this configuration and then waits for the callback of an incoming MAM request.

Upon a callback, the policy can use information from the MAM request, such as Socket Intents, and information available within the MAM, such as recently measured path characteristics (see Section 3.2), to make decisions.

Policy decisions can include:

- o The source address(es) used for name resolution
- o How to order the results of name resolution (i.e., preferring certain IP addresses over others)
- o Picking an IP protocol version
- o Picking a transport protocol (Note that in our current implementation, we are constrained by the Socket API, so our policy cannot override the transport protocol chosen by an application.)
- o Setting socket options (e.g., disable TCP Nagle)
- o Choosing a source address for the outgoing communication

- Reusing a socket from a given socket set (only for the API variant described in Section 5.3)

Note that in our current implementation, the policy is a piece of code which can in principle execute arbitrary instructions. We assume this is acceptable for an experimental platform but would prefer an abstract description like a domain-specific language for a production system.

3.2. Path characteristics data collectors

The data collectors are implemented as a component of the MAM, within a callback that is executed periodically, e.g., every 100 ms. When this callback is invoked, the MAM passively gathers statistics about the current usage and properties of the available local interfaces and stores them in per-interface or per-network prefix data structures.

Measured properties include:

- Minimum Smoothed Round Trip Time (SRTT) of current TCP connections using a network prefix, as an estimate for last-mile latency

- Median SRTT of current TCP connections using a network prefix, as an alternate estimate for last-mile latency

- Median of Round Trip Time variations within connections

- Median variation of Smoothed Round Trip Times across connections

- Median of percentage of segments deemed lost of all transmitted segments of current TCP connections, as an estimate of upstream packet loss

- Maximum transmitted and received bytes per second over an interface within the last 5 minutes, as an estimate for maximum available bandwidth

- On 802.11 interfaces, the Received Signal Strength Indicator (RSSI) of the last received frame on that interface, as an estimate for reception strength

- On 802.11 interfaces, the modulation rate of the last received and the last transmitted unicast data frame on that interface, as an estimate for the available data transmission rate on the first hop

o On 802.11 interfaces, the latest Channel Utilization as parsed from a Beacon frame, as an estimate of congestion on the wireless medium

See [ANRW18-Metrics] for more discussion of the gathered metrics.

When a policy callback is invoked, the policy can use the latest measured properties to guide its decisions, see Section 3.1.

Note that we do not perform active measurements from within the MAM to avoid overhead.

4. Socket Intents Representation

As described in [I-D.tiesel-taps-socketintents], Socket Intents are pieces of information about upcoming traffic. An application can share the information that it has available through the Socket Intents API.

In our implementation, Socket Intents are represented as socket options for get/setsockopt on its own socket option level (SOL_INTENTS).

For some of the API variants, we had to introduce socket option lists, i.e., data structures that can hold multiple socket options and therefore multiple Socket Intents.

Which of these variants is actually used depends on the API variant, see Section 5.

5. The Socket Intents API Variants

The Socket Intents API is a wrapper around the BSD Socket API. It sends requests to the Multiple Access Manager (MAM) at certain events, e.g., before a connection is established, and applies the suggestions that it gets from the MAM, e.g., to bind to a certain local interface or to set a certain socket option.

There exist different variants of this API, see Section 5, that try to fit different concepts:

o The Classic API with muacc_context, see Section 5.1, was attempting to stick as close as possible to the call sequence of BSD Sockets.

o The second variant of the classic API does all transport option selection in "getaddrinfo", see Section 5.2. This variant tries to simplify the implementation without deriving too much from the

usage of BSD Sockets. It minimizes the changes to the BSD Socket API, but adds additional overhead to the application.

- The "socketconnect" API, see Section 5.3, tries to automate as much functionality as possible and adds support for automating connection caching. It replaces the usual sequence of BSD Socket API calls with a single call.

5.1. Classic API / muacc_context

In the first variant, we add a parameter called "muacc_context" to the BSD Socket API calls and to getaddrinfo. This parameter holds properties provided by the socket calls and retains them across function calls to enable automation of the connection properties by our Socket Intents Prototype. The shadow data structures behind the "muacc_context" parameter are initialized by API wrapper at the time of the first call (which we assume to be muacc_getaddrinfo most of the time) with most of its fields empty. Then within each call to our modified Socket API, it is filled with data.

Properties include:

- Socket file descriptor
- API calls that were already performed on this context
- domain, type, and protocol of the socket
- remote hostname
- remote address
- hints for resolving the remote address
- local address to bind to that the application requested
- local address to bind to that the MAM suggested
- current socket options that were set
- socket options suggested by MAM

5.1.1. muacc_getaddrinfo()

This function resolves a host name or service to an addrinfo data structure, usually containing an IP address or port. Internally, the Socket Intents prototype sends a "getaddrinfo" request to the MAM, which should do the name resolution. It can, e.g., resolve the name

over multiple available interfaces at the same time, and then order the results according to a policy decision, or only return results obtained over a specific interface.

SIGNATURE:

```
int muacc_getaddrinfo(muacc_context_t *ctx, const char *hostname,
const char *servname, const struct addrinfo *hints, struct addrinfo
**res)
```

ARGUMENTS:

ctx: Context that can contain properties of this socket/connection and retains them across function calls. This function is mostly called with an empty context, which is then filled within the function.

hostname: Remote host name to be resolved

servname: Remote service to be resolved

hints: Hints for resolving the name

res: Data structure for result of name resolution

RETURN VALUE:

Returns 0 on success, or an error code as provided by getaddrinfo().

5.1.2. muacc_socket()

This function creates a socket file descriptor just like the regular socket call.

SIGNATURE:

```
int muacc_socket(muacc_context_t *ctx, int domain, int type, int
protocol)
```

ARGUMENTS:

ctx: Context that can contain properties of this socket/connection and retains them across function calls. This function is mostly called after muacc_getaddrinfo(), since domain, type, and protocol can depend on the type of resolved address.

domain: Domain of the socket

type: Type of the socket

protocol: Protocol of the socket

RETURN VALUE:

Returns a file descriptor of the new socket on success, or -1 on failure.

5.1.3. muacc_setsockopt()

This call allows to set socket options (including Socket Intents). For Socket Intents, this function can be called on a valid "muacc_context" and an invalided file descriptor (-1) to provide assertional hints to "muacc_getaddrinfo()".

SIGNATURE:

```
int muacc_setsockopt(muacc_context_t *ctx, int socket, int level, int
option_name, const void *option_value, socklen_t option_len)
```

ARGUMENTS:

ctx: Context that can contain properties of this socket/connection and retains them across function calls. This function is mostly called to set Intents as socket options within the context.

socket: Socket file descriptor

level: Level of the socket option to set

option_name: Name of the socket option to set

option_value: Value of the socket option to set

option_len: Length of the socket option to set

RETURN VALUE:

Returns 0 on success, or -1 on failure.

5.1.4. muacc_connect()

Like the regular connect call, but also binds to the source address selected by the Socket Intents Policy and applies socket options suggested by the Socket Intents Policy.

SIGNATURE:

int muacc_connect(muacc_context_t *ctx, int socket, const struct sockaddr *address, socklen_t address_len)

ARGUMENTS:

ctx: Context that can contain properties of this socket/connection and retains them across function calls. This function is mostly called after all Socket Intents for this connection have been set via muacc_setsockopt().

socket: Socket file descriptor

address: Remote address to connect to

address_len: Length of the remote address

RETURN VALUE:

Returns 0 on success, or -1 on failure.

5.1.5. muacc_close()

Like regular close, but also cleans up state held in shadow structures behind "muacc_context"

SIGNATURE:

int muacc_close(muacc_context_t *ctx, int socket)

ARGUMENTS:

ctx: Context that can contain properties of this socket/connection and retains them across function calls. This function deinitializes and releases the context.

socket: Socket file descriptor

RETURN VALUE:

Returns 0 on success, or -1 on failure.

5.2. Classic API / getaddrinfo

In this variant, Socket Intents are passed directly to "getaddrinfo()" as part of the "hints" parameter. The name resolution is done by the MAM, which makes all decisions and stores them in the "result" data structure as list of options ordered by preference. Subsequently, applications can use this information for

calls to the unmodified BSD Socket API or other APIs. We provide helpers to apply all socket options from the "result" data structure.

All relevant infos are stored in our addrinfo struct (see Figure 2)

SIGNATURE:

```
int muacc_ai_getaddrinfo(const char * hostname, const char * service,
const struct muacc_addrinfo * hints, struct muacc_addrinfo ** result)
```

ARGUMENTS:

hostname: Remote host name to be resolved

service: Remote service to be resolved

hints: Hints for resolving the name. Contents include family, socket type, protocol, socket options (including Socket Intents for this socket/connection), local address to bind to.

result: Data structure for result of name resolution

RETURN VALUE:

Returns 0 on success, or an error code as provided by getaddrinfo().

```
/** Extended version of the standard library's struct addrinfo
 *
 * This is used both as hint and as result from the
 * muacc_ai_getaddrinfo * function. This structure
 * differs from struct addrinfo only in the three members
 * ai_bindaddrlen, ai_bindaddr and ai_socketopt.
 */
struct muacc_addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;

    /** Not included in struct addrinfo. Purpose:
      * 1. If the structure is given to muacc_ai_getaddrinfo
      *    as hints, you set socket intents that influence MAM's
      *    source and destination as well as transport protocol
      *    selection
      * 2. The recommended socket options MAM will be returned
      *    through this attribute.
      */
    struct socketopt *ai_sockopts;

    int ai_addrlen;
    struct sockaddr *ai_addr;
    char *ai_canonname;

    /** Not included in struct addrinfo.
      * Length of ai_bindaddr.
      */
    int ai_bindaddrlen;
    /** Not included in struct addrinfo.
      * Contains the address, which the MAM recommends us to bind to.
      */
    struct sockaddr *ai_bindaddr;

    struct muacc_addrinfo *ai_next;
};
```

Figure 2: Definition of the muacc_addrinfo struct

Appendix A.2 shows an example usage of the classic API with most functionality in getaddrinfo.

5.3. Socketconnect API

In this API variant, we move the functionality of resolving a hostname and connecting to the resulting address into one function called "socketconnect()". This API makes it possible to call socketconnect not only for each connection, but also to multiplex messages across multiple existing sockets.

This function returns a file descriptor of a connected socket for the application to use. This socket can either be a newly created one or a socket that existed previously and is now being reused. Furthermore, a socket can belong to a socket set of sockets with common destination and service. These sockets may, e.g., be bound to different local addresses, but are treated as interchangeable by the API implementation. So if the application passes a socket file descriptor to this function, it may get back a different file descriptor to a socket from the same set, e.g., to use the connection over a different local interface for its following communication.

SIGNATURE:

int socketconnect(int *socket, const char *host, size_t hostlen, const char *serv, size_t servlen, struct socketopt *sockopts, int domain, int type, int proto)

ARGUMENTS:

socket: Existing socket file descriptor as representant to a socket set, "-1" to create a new socket, or "0" to automatically try to find a suitable socket set

host: Remote hostname to be resolved

hostlen: Length of remote hostname

serv: Remote service or port

servlen: Length of remote service

socketopts: List of socket options, including Socket Intents

domain: Domain of the socket

type: Type of the socket

proto: Protocol of the socket

RETURN VALUE:

Returns 0 on success if socket is from an existing socket set, 1 on success if socket was newly created, or -1 on fail.

Appendix A.3 shows an example usage of the Socketconnect API.

6. API Implementation Experiences & Lessons Learned

While designing and implementing the different parts of the system as described in this document, we faced several challenges. In the Multiple Access Manager discovering the currently available paths and statistics about their performance turned out to be quite complex and had to be implemented in a partially platform-dependent way. However, the most challenging parts were the Socket Intents API and Library, on which we focus in the following sections.

6.1. The Missing Link to Name Resolution

Transport option selection is most useful if crucial information, such as Socket Intents or other socket options, is available as early as possible, i.e., for name resolution. The primary problem here is the order of the function calls that are involved in name resolution, destination selection, protocol, and path selection, and how they are linked.

In the classic BSD Socket API, most functions either take a socket file descriptor as argument or return it, and thus link different function calls to the same flow. However, "getaddrinfo()" is not linked to a socket file descriptor, and it is typically called before the socket is created. At this point, it is not yet possible to set a socket option, because the socket does not exist yet.

Consequently, across BSD Socket API calls, several choices are being made before it is possible to set a Socket Intent: A call to "getaddrinfo()" returns a linked list of "addrinfo" structs, where each entry contains an "ai_family" (IP version), the pair of "ai_socktype" and "ai_protocol" (transport protocol), and a "sockaddr" struct containing an IP address and port to connect to. Then a socket of the given family, type, and protocol is created. Only after this has been done, socket options can be set on the socket, but at this point destination, IP version, and transport protocol are already fixed. Before calling "connect()", only the path to be used (i.e., the local address to bind to) can still be chosen, but the available paths and which one to prefer may be constrained by the choice of destination.

The three variants described in Section 5 work around this problem in different ways:

- The approach in Section 5.2 places the whole automation of transport option selection into the "getaddrinfo()" function. The results are returned in an extended "addrinfo" struct and have to be applied manually by the application, including binding to a source address representing the selected path and applying all socket options provided in a list, for each connection attempt.

- The approach in Section 5.1 adds a context to all socket- and name resolution-related API calls.

- The approach in Section 5.3 puts all functionality into one call.

All of these approaches add the missing link between name resolution and the other parts of the API, but add a lot of state keeping either to the API, which the application developer has to manage, or to the Socket Intents library.

6.2. File Descriptors Considered Harmful

When using BSD sockets, file descriptors are the abstraction for network flows. Depending on the transport protocol used, their semantics changes and these file handles represent streams (SOCK_STREAM), associations (SOCK_DRAM) or network interfaces (SOCK_RAW). This does not provide a unified API, but is merely an artifact of squeezing networking into the "Everything is a file" UNIX philosophy.

File descriptors make no good abstraction for automated protocol stack instance selection as applications have to adopt to changed semantics, e.g., whether message boundaries are preserved, depending on the transport protocol chosen.

File descriptors make no good abstraction for destination instance selection and path selection either. Once a socket has been created, its protocol stack instance is fixed, so selecting a path by binding to a local address and connecting to a destination instance is now only possible using this protocol stack instance. If such a connection attempt fails, it is possible to retry using another path and destination, but changing the protocol stack instance requires creating a new socket with a different file descriptor.

For further discussion of other asynchronous I/O weirdness with file descriptors see end of Section 6.3.

6.3. Asynchronous API Anarchy

Network I/O is asynchronous, but asynchronous I/O within the POSIX filesystem API is hard to use. There are at least three different asynchronous I/O APIs for each operating system.

To implement asynchronous I/O for our Socket Intents prototype, we wrapped one of the asynchronous I/O APIs that is available on most platforms: "select()". To make Socket Intents accessible to more applications and on more platforms, a production-grade system would need to wrap all asynchronous I/O APIs and implement most of the socket creation logic, path selection and connection logic within these wrappers. However, mixing asynchronous I/O and multithreading may lead to unintuitive behavior, e.g., calling our prototype's select() from different threads could lead to anything from deadlocks to busy waiting.

Another issue is that we use Unix domain sockets to communicate between our Multiple Access Manager and the Socket Intents API library called by the application, so we need to make sure that the application does not block on communication with the Multiple Access Manager.

Also the problems with using file descriptors get even worse. If a Socket API call should return immediately, it needs to provide the application with a reference to a flow that has not yet been fully set up, i.e., a reference to a "future" socket. An implementation of such an asynchronous API has to return an unconnected socket file descriptor, on which the application then calls, e.g., "select()", and starts using it once it becomes readable and writable. If the destination, path and transport protocol have not been chosen yet at this point, the file descriptor returned by the implementation might not yet have the final family and transport protocol. When the implementation later creates the final socket of the right type, it can re-bind it to the file-id of the originally returned file descriptor using "dup2". This procedure can easily lead to time-of-check to time-of-use confusion. To make things even worse, the application can copy the "future" file descriptor using "dup", which is rarely useful for sockets, but in combination with file descriptors used as "future" it leads to unexpected behavior.

6.4. Here Be Dragons hiding in Shadow Structures

The API variants described in Section 5.3 and Section 5.1 need to keep a lot of state in shadow structures that cannot be passed between the Socket API calls otherwise. This state needs to be cleaned up when the last copy of the file descriptor is closed or the

last socket held for reuse has timed out. In addition, access to these shadow structures has to be thread-safe.

Implementing both has turned out to be extremely error-prone and there is a high amount of unspecified behavior and platform-dependent extensions in the system library. These issues guarantee that an implementation of transport option selection that nicely integrates with BSD Sockets will come with lots of limitations and will not be portable across POSIX-compliant operating systems.

7. Conclusion

Adding transport option selection to BSD Sockets is hard, as the API calls are not designed to defer making and applying choices to a moment where all information needed for transport option selection is available.

After all, if limiting transport option selection to the granularity BSD Sockets typically provide today (TCP connections and UDP associations), the API variant described in Section 5.2 seems to be a good compromise, even if it forces the application to try all candidates itself (either in a sequential or partial parallel fashion). This option is easily deployable, but does not include automation of techniques like connection caching or HTTP pipelining.

The most versatile API variant described in Section 5.3 implements connection caching on the transport layer. This comes at the cost of heavily modifying existing applications. If feasible, given the unnecessary complexity of the file I/O integration of BSD sockets, it seems easier to move to a totally different system like [I-D.trammell-taps-post-sockets].

8. Acknowledgments

The API variant described in Section 5.2 was originally drafted and implemented by Tobias Kaiser mail@tb-kaiser.de [2] as part of his BA thesis.

This work has been supported by Leibniz Prize project funds of DFG - German Research Foundation: Gottfried Wilhelm Leibniz-Preis 2011 (FKZ FE 570/4-1).

9. References

9.1. Informative References

[ANRW17-MH]
Tiesel, P., May, B., and A. Feldmann, "Multi-Homed on a Single Link", Proceedings of the 2016 workshop on Applied Networking Research Workshop - ANRW 16, DOI 10.1145/2959424.2959434, 2016.

[ANRW18-Metrics]
"Metrics for access network selection (ANRW 2018)", n.d..

[I-D.tiesel-taps-communitgrany]
Tiesel, P. and T. Enghardt, "Communication Units Granularity Considerations for Multi-Path Aware Transport Selection", draft-tiesel-taps-communitgrany-02 (work in progress), May 2018.

[I-D.tiesel-taps-socketintents]
Tiesel, P., Enghardt, T., and A. Feldmann, "Socket Intents", draft-tiesel-taps-socketintents-01 (work in progress), October 2017.

[I-D.trammell-taps-post-sockets]
Trammell, B., Perkins, C., Pauly, T., Kuehlewind, M., and C. Wood, "Post Sockets, An Abstract Programming Interface for the Transport Layer", draft-trammell-taps-post-sockets-03 (work in progress), October 2017.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <https://www.rfc-editor.org/info/rfc6824>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <https://www.rfc-editor.org/info/rfc7413>.

[RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <https://www.rfc-editor.org/info/rfc7556>.

9.2. URIs

[1] https://github.com/fg-inet/socket-intents/

[2] mailto:mail@tb-kaiser.de

Appendix A. API Usage Examples

A.1. Usage Example of the Classic / muacc_context API

In this example, a client application sets up a connection to a remote host and sends data to it. It specifies two Socket Intents on this connection: The Category of Bulk Transfer and the File Size of 1 MB.

```
#define LENGTH_OF_DATA 1048576

// Create and initialize a context to retain information across function
// calls
muacc_context_t ctx;
muacc_init_context(&ctx);

int socket = -1;

struct addrinfo *result = NULL;

// Initialize a buffer of data to send later.
char buf[LENGTH_OF_DATA];
memset(&buf, 0, LENGTH_OF_DATA);

// Set Socket Intents for this connection. Note that the "socket" is
// still invalid, but it does not yet need to exist at this time. The
// Socket Intents prototype just sets the Intent within the
// muacc_context data structure.

enum intent_category category = INTENT_BULKTRANSFER;
muacc_setsockopt(&ctx, socket, SOL_INTENTS,
    INTENT_CATEGORY, &category, sizeof(enum intent_category));

int filesize = LENGTH_OF_DATA;
muacc_setsockopt(&ctx, socket, SOL_INTENTS,
    INTENT_FILESIZE, &filesize, sizeof(int));


// Resolve a host name. This involves a request to the MAM, which can
// automatically choose a suitable local interface or other parameters
// for the DNS request and set other parameters, such as preferred
// address family or transport protocol.
```

```
muacc_getaddrinfo(&ctx, "example.org", NULL, NULL, &result);

// Create the socket with the address family, type, and protocol
// obtained by getaddrinfo.
socket = muacc_socket(&ctx, result->ai_family, result->ai_socktype,
    result->ai_protocol);

// Connect the socket to the remote endpoint as determined by
// getaddrinfo.  This involves another request to MAM, which may at this
// point, e.g., choose to bind the socket to a local IP address before
// connecting it.
muacc_connect(&ctx, socket, result->ai_addr, result->ai_addrlen);

// Send data to the remote host over the socket.
write(socket, &buf, LENGTH_OF_DATA);

// Close the socket. This de-initializes any data that was stored within
// the muacc_context.
muacc_close(&ctx, socket);
```

A.2. Usage Example of the Classic / getaddrinfo API

As in Appendix A.1, the application sets the Intents "Category" and "File Size".

```
#define LENGTH_OF_DATA 1048576

// Define Intents to be set later
enum intent_category category = INTENT_BULKTRANSFER;
int filesize = LENGTH_OF_DATA;

struct socketopt intents = { .level = SOL_INTENTS,
    .optname = INTENT_CATEGORY, .optval = &category, .next = NULL};
struct socketopt filesize_intent = { .level = SOL_INTENTS,
    .optname = INTENT_FILESIZE, .optval = &filesize, .next = NULL};

intents.next = &filesize_intent;

// Initialize a buffer of data to send later.
char buf[LENGTH_OF_DATA];
memset(&buf, 0, LENGTH_OF_DATA);

struct muacc_addrinfo intent_hints = { .ai_flags = 0,
    .ai_family = AF_INET, .ai_socktype = SOCK_STREAM, .ai_protocol = 0,
    .ai_sockopts = &intents, .ai_addr = NULL, .ai_addrlen = 0,
    .ai_bindaddr = NULL, .ai_bindaddrlen = 0, .ai_next = NULL };

struct muacc_addrinfo *result = NULL;

muacc_ai_getaddrinfo("example.org", NULL, &intent_hints,
    &result);

// Create and connect the socket, using the information obtained through
// getaddrinfo
int fd;
fd = socket(result->ai_family, result->ai_socktype,
    result->ai_protocol);
muacc_ai_simple_connect(fd, result);

// Send data to the remote host over the socket, then close it.
write(fd, &buf, LENGTH_OF_DATA);
close(fd);

muacc_ai_freeaddrinfo(result);
```

A.3. Usage Example of the Socketconnect API

As in Appendix A.1, the application sets the Intents "Category" and "File Size". As we provide "-1" as socket, no we do not reuse existing connections.

```
   #define LENGTH_OF_DATA 1048576

   // Define Intents to be set later
   enum intent_category category = INTENT_BULKTRANSFER;
   int filesize = LENGTH_OF_DATA;

   struct socketopt intents = { .level = SOL_INTENTS,
       .optname = INTENT_CATEGORY, .optval = &category, .next = NULL};
   struct socketopt filesize_intent = { .level = SOL_INTENTS,
       .optname = INTENT_FILESIZE, .optval = &filesize, .next = NULL};

   intents.next = &filesize_intent;

   // Initialize a buffer of data to send later.
   char buf[LENGTH_OF_DATA];
   memset(&buf, 0, LENGTH_OF_DATA);

   int socket = -1;

   // Get a socket that is connected to the given host and service,
   // with the given Intents
   socketconnect(&socket, "example.org", 11, "80", 2, &intents, AF_INET,
       SOCK_STREAM, 0);

   // Send data to the remote host over the socket.
   write(socket, &buf, LENGTH_OF_DATA);

   // Close the socket and tear down the data structure kept for it
   // in the library
   socketclose(socket);
```

Appendix B. Changes

B.1. Since -01

- Updated list of gathered path characteristics

- Reordered start of Policy section to make it clearer

B.2. Since -00

- Fixed Author's affiliations and funding

- Fixed acknowledgments

Authors' Addresses

Philipp S. Tiesel
TU Berlin
Marchstr. 23
Berlin
Germany

Email: philipp@inet.tu-berlin.de

Theresa Enghardt
TU Berlin
Marchstr. 23
Berlin
Germany

Email: theresa@inet.tu-berlin.de