Technische Universität Berlin

Fakultät IV
Fachgebiet Internet Network Architectures

Masterarbeit in Informatik

Socket Intents: Extending the Socket API to Express Application Needs

Theresa Enghardt

July 26, 2013

Aufgabenstellerin:	Prof. Anja Feldmann, Ph. D.
Betreuer/innen:	Philipp S. Schmidt, Dipl.-Inf.
	Prof. Anja Feldmann, Ph. D.
Abgabedatum:	26.07.2013

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.
Berlin, den 26.07.2013

Unterschrift

Abstract

In today's internet, almost all end devices have multiple interfaces built in, which enable them to switch between different access networks or even use them simultaneously. Having different characteristics, some of them may be more suitable for certain kinds of traffic, and therefore better meet the requirements of certain applications. To account for this, current solutions rely on static policies or reactive approaches to choosing between interfaces.

This thesis proposes a proactive, application informed approach, *Socket Intents*. Socket Intents augment the socket interface to enable the application to express its communication preferences. This information can then be used by proactive policies to choose the appropriate interface, tune the network parameters, or even combine multiple interfaces. In this work, a prototype implementation of Socket Intents and a framework supporting them is presented. Furthermore, a first evaluation of the Intents and its benefits is conducted. We find that Socket Intents may improve the possibilities to take advantage of one's multiple interfaces.

Zusammenfassung

Heutzutage haben nahezu alle internetfähigen Geräte mehrere Netzwerkschnittstellen, die es ihnen ermöglichen, zwischen unterschiedlichen Zugangsnetzen zu wechseln oder sogar mehrere gleichzeitig zu benutzen. Zugangsnetze haben unterschiedliche Eigenschaften und Applikationen haben unterschiedliche Anforderungen, weshalb einige Zugangsnetze unter Umständen für bestimmte Applikationen besser geeignet sind als andere. Es gilt, unter mehreren verfügbaren Netzwerkschnittstellen eine passende auszuwählen. Hierfür setzen viele aktuelle Lösungen auf statische, pauschale Regeln, ohne die Bedürfnisse der individuellen Applikationen zu berücksichtigen. Andere Lösungen reagieren auf sie, indem sie versuchen, einer bestehenden Verbindung zu entnehmen, ob Verbesserungen möglich sind, z.B. durch Messung.

Die vorliegende Arbeit verfolgt stattdessen einen proaktiven Ansatz, der über die Anforderungen der Applikation im Vornherein informiert ist und diese berücksichtigt: *Socket Intents.* Socket Intents erweitern die Socket-Schnittstelle, um es der Applikation zu ermöglichen, Präferenzen hinsichtlich ihrer Netzwerkkommunikation auszudrücken. Diese Information kann dann proaktiv für Entscheidungen genutzt werden, um die passende Netzwerkschnittstelle auszuwählen, die Netzwerkparameter anzupassen oder sogar mehrere Verbindungen aufzubauen und zu kombinieren. Diese Arbeit präsentiert einen Prototyp der Socket Intents und ein Framework, das diese unterstützt. Außerdem wird eine erste Evaluation der Intents und ihres Nutzens durchgeführt. Wir stellen fest, dass Socket Intents es einem System ermöglichen können, die Möglichkeiten mehrerer Netzwerkschnittstellen besser auszunutzen.

Submitted and Joint Work

Parts of this thesis were written in collaboration and submitted as a conference paper:

Philipp S. Schmidt, Theresa Enghardt, Ramin Khalili and Anja Feldmann
Socket Intents: Leveraging Application Awareness for Multi-Access Connectivitiy
(short paper)
For CoNEXT'13, December 2013, Santa Barbara, CA, USA

Socket Intents were incorporated into the Multi Access Framework originally developed by Philipp S. Schmidt and further expanded in collaboration. It has been extended to handle the Intent Socket Options. Furthermore, several policies that handle Intents were written and evaluated within this thesis.

Contents

Glossary

2G 2nd Generation of cellular mobile telecommunications technology.

3G 3rd Generation of cellular mobile telecommunications technology.

4G 4th Generation of cellular mobile telecommunications technology.

API Application Program Interface.

BSD Berkeley Software Distribution, Unix operating system.

DiffServ Differentiated Services.

DNS Domain Name System.

DSL Digital Subscriber Line.

GPRS General Packet Radio Service, data service of 2G.

HTTP HyperText Transfer Protocol.

IEEE Institute of Electrical and Electronics Engineers, standardization body.

IETF Internet Engineering TaskForce, informal standards organization.

IntServ Integrated Services.

IP Internet Protocol.

IPv4 Internet Protocol version 4.

IPv6 Internet Protocol version 6.

ISO International Standards Organization.

ISP Internet Service Provider.

LTE Long Term Evolution, implementation of a 4G network.

MAM Multi Access Manager.

MPTCP MultiPath Transmission Control Protocol.

MUACC MUltiple ACCess.

OS Operating System.

POSIX Portable Operating Systems Interface.

QoE Quality of Experience.

QoS Quality of Service.

RFC Request For Comments, publication of the IETF.

RSSI Received Signal Strength Indication.

RTT Round Trip Time.

SCTP Stream Control Transmission Protocol.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

WiFi Wireless Fidelity, synonym for WLAN.

WLAN Wireless Local Area Network.

1. Introduction

Ten years ago, most clients had just a single way to connect to the internet (typically Ethernet for laptops and workstations, and GPRS for smartphones). Today, almost all devices have multiple interfaces built in. For example, smartphones have built-in 3G/4G as well as WiFi interfaces, see Figure 1, while most laptops have Ethernet, WiFi, and can use 3G in addition. Moreover, they can seamlessly switch between using one or the other interface or even use multiple of them at the same time.

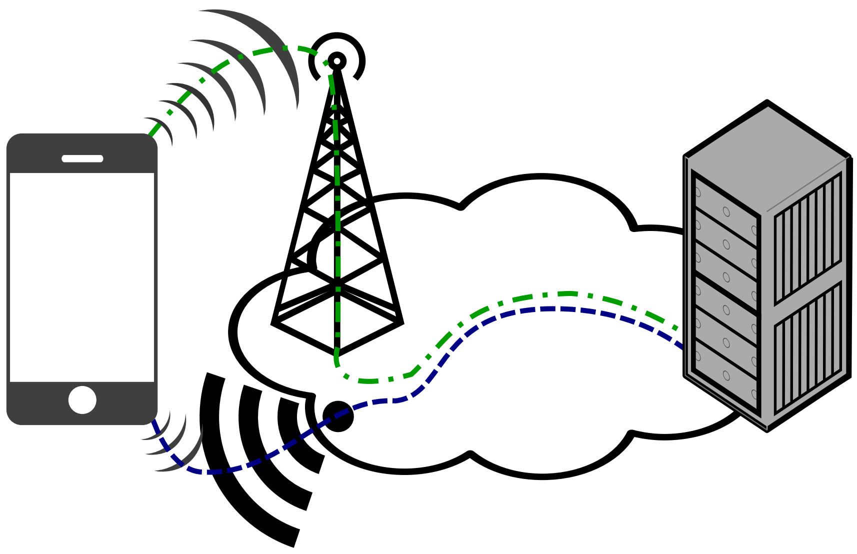


Figure 1: Multi-Access Host: The majority of today's hosts can connect to the internet through multiple interfaces at the same time.

The performance that each individual network technology provides differs, e.g., in terms of bandwidth, delay, availability, congestion, cost per byte, and energy cost. For example, while WiFi if uncongested provides higher bandwidth than 3G and therefore may be preferable, the 3G network may be preferable if the WiFi network is congested. Moreover, 3G and 4G work even if the user is no longer in reach of their WiFi access points. Of course 4G currently offers higher peak bandwidth than even most DSL lines but it is a shared medium and thus susceptible to congestion. Thus, there is a lot of optimization potential on the devices with respect to choosing or combining the appropriate interfaces.

For this, current devices often rely on static configuration policies, trying to find a general solution for all connections regardless of their individual characteristics. If they take application needs into account, their approach is reactive rather than proactive. This means that they do not attempt to optimize their decisions before the application establishes the connection, but try to assess and possibly improve their service while communication is in progress.

In this thesis, we propose a proactive, application informed approach: *Socket Intents*. Instead of trying to infer the characteristics of an application's traffic from an ongoing connection, the application is given the possibility to express its needs up front. In this way, they can immediately be incorporated into decisions, even before a connection is established. For example, if the user of the smart phone of Figure 1 wants to access a website that consists of many small objects, the connection will benefit from low delay. However, if they are downloading a big file, high bandwidth is more important. If this information is communicated by the application before establishing the connection, it is possible to choose a suitable source interface in either case.

One of the key problems though is that such decisions are often made within the operating system running on the host, where by default, only little information about the application's needs is available. For instance, it typically has no idea of the size of the downloaded data nor its relevance to the user. However, this information is often available to the application. Therefore, Socket Intents augment the Socket API to enable the application to express its communication preferences. Here, preferences can refer to high bandwidth, low delay, connection resilience, etc. This information can then be used by our dynamic proactive policies to choose the appropriate interface, tune the network parameters, or even combine multiple interfaces. We implement a framework for making such policy decisions and applying them, aiming to provide better service according to application needs. Different policies for different settings and use cases were implemented and evaluated.

This thesis is structured as follows: First, some background information on the socket interface and its implementation is provided, as it is the basis for our enhanced Socket API. Then there is a survey of related work. Afterwards we describe our design and implementation of the Socket Intents framework in detail and explain the practical aspects of how to use it. We then provide a first evaluation of the potential benefits. Finally, we conclude the thesis with an outlook and possible future work.

2. Background

The following section provides an introduction of how network communication on end hosts works. It first presents sockets as the central concept at the intersection of applications and the networking implementation within the Operating System (OS). Then, the process of establishing a connection is explained, the relevant functions are discussed in detail and an overview of the structure of an OS's networking implementation is given. Concluding this section, the most promising points where modification is possible are highlighted.

2.1. Definition of a Socket

A socket is an abstraction of communication between processes, i.e. applications running on computing devices such as PCs or smartphones. They allow for communication with other applications, either running on the same host or on another one. Thus, they facilitate bidirectional interprocess communication across multiple hosts and possibly using multiple interfaces, as shown in Figure 2.

To be able to deliver data to the correct socket, sockets must be referenced by socket addresses. The socket address serves as an identifier of the communication endpoint, and may consist of several components, e.g. for the machine and for the application on the machine. A common analogy is the way in which the endpoint of a telephone line is identified by a telephone number and extension. There are different kinds of socket addresses, called address families, which may differ in content and form of the identifiers they store. Which address family is used for a certain socket depends on its domain of communication. For instance, there are local sockets for communication within a single host and network sockets which use a network protocol to communicate with remote hosts. The former are called Unix Domain Sockets and use file descriptors as addresses, i.e. paths to a file on the host. The latter vary depending on the available network protocols, but most commonly they use IP, the Internet Protocol, and are thus called Internet sockets.

In the case of IPv4, the address consists of a numeric IP address to identify the host that the application is running on, and a port number. This is a number between 1 and 65535 which helps distinguish between applications within a single host. Port numbers up to 1024 are "well-known" ports reserved for certain applications, e.g. port 80 is most often used by web servers.

Sockets can have a local socket address on the host they are running on, and they can connect to a remote socket address on another host. Furthermore, they need to specify a transport protocol to use. The most common examples of transport protocols are TCP and UDP of the Internet Protocol suite. Within a host, the combination of local socket address and transport protocol that a socket uses must be unique. That means that only one process may use a given port number with a given IP address and a given transport protocol at a time. While this process is running, others must either choose another port, another transport protocol or another local IP address. Two sockets are connected when one socket's local address is the other socket's remote address and vice versa, and they use the same transport protocol. The combination of local and remote socket address and transport protocol is called association, which is an important concept for setting up a connection.

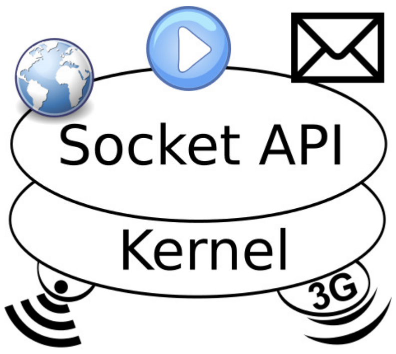


Figure 2: Within host: Applications/network interfaces.

2.2. The Socket API

The Socket API consists of a number of functions and data structures provided by OSes in standard libraries. It provides a programming interface that allows applications to use sockets. Applications can expect them to be present on most systems by default.

This API, also called the Berkeley Socket API, has been standardized in a joint Open Group/IEEE/ISO standard for Portable Operating Systems Interface (POSIX) [1]. Many modern OSes are mostly POSIX-compliant and implement the Socket API in the form presented here. This is especially true for any Unix-based OS such as Linux or Mac OS, since the POSIX Socket API evolved from BSD, a Unix derivate. Also, the Windows Socket API is modeled after the presented Berkeley Socket API.

After standardization, many extensions have been proposed, e.g. by the IETF in RFCs to support IPv6 [8, 19]. This is one of the important extensions that has also been implemented in most OSes.

In the following sections, the functions of the Socket API are introduced in detail.

2.2.1. Connection and Operation

Let's start with the example of an application that wants to communicate over a network using the Socket API. Assuming that it wants to connect to a remote host, exchange some data with it and then terminate the connection, it has to perform a number of steps. The necessary actions and the resulting status of its local socket are shown in Figure 3.

At first, the application has to create a socket and choose a communication domain, a type, and a transport protocol. For creating the socket, the `socket` function is called. The socket type that is chosen depends on the characteristics of data delivery that the application needs, most commonly used are stream sockets and datagram sockets. The former provide the reliable delivery of a stream of bytes, which is guaranteed to arrive at the other side in the same order. The latter delivers packets of data on a best-effort basis, which means that their arrival is not guaranteed and can generally not be confirmed. The combination of address family and socket type designates a number of transport protocols to choose from. Applications can either set one explicitly or instruct the OS to use the default, e.g. TCP for stream sockets and UDP for datagram sockets.

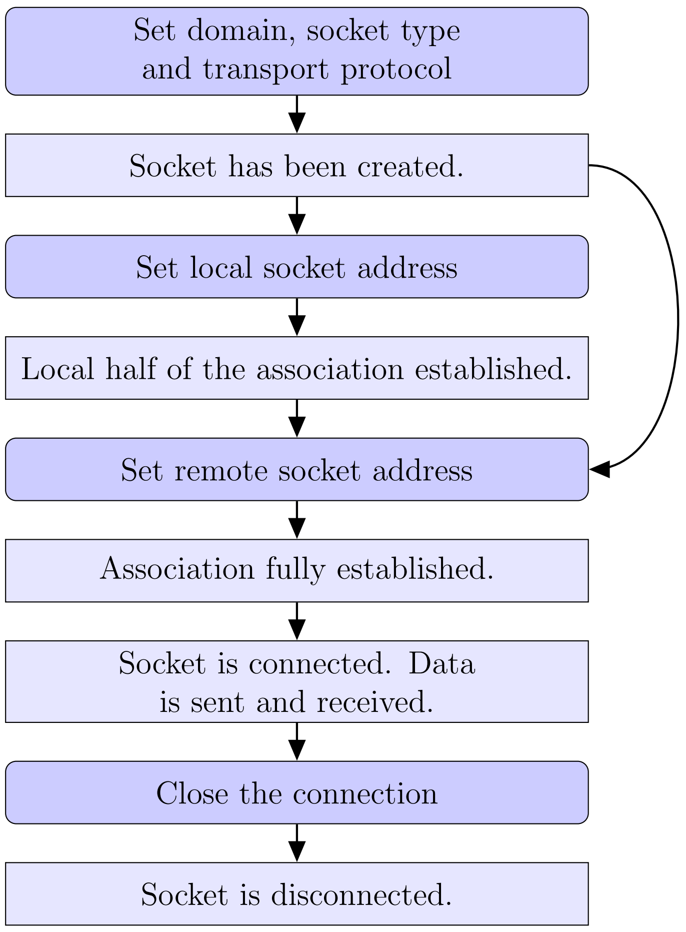

Set domain, socket type and transport protocol
Socket has been created.
Set local socket address
Local half of the association established.
Set remote socket address
Association fully established.
Socket is connected. Data is sent and received.
Close the connection
Socket is disconnected.

Figure 3: Steps to establish, use and terminate a connection on the client side

Note that datagram sockets are not connection oriented, which means that an application is not required to establish an association on them and can simply start sending or receiving. In this case, it needs to specify the destination address within the send operation.

For connection oriented sockets, an association has to be established first. There are two possibilities: Applications can either proactively connect to a remote address, or they can passively wait for other applications to connect to it. The first case corresponds to the "client", the second case to the "server" in the client-server model for distributed applications.

Applications that proactively connect to a remote application can use the `bind` function for their local half of the socket's association and must use `connect` for the remote part. Both of these functions take a socket address and its length as an argument.

If the application omits the step of setting a local socket address, the OS chooses one of its available local socket addresses on its own.

Applications that passively wait for incoming connections from other applications, also called peers, must use the same `bind` function mentioned above to set its local socket address. This is important, since peers need to know this socket address to be able to connect to it. Waiting for them, the application must call `listen` to set up a queue for incoming connections on its half-associated socket. To handle connection requests, the `accept` function must then be called. Once a peer connects to the application's local socket address, it creates a new socket with a complete association, including the remote socket address of the peer. The original socket remains unconnected with only half an association, waiting for more peers to connect to it.

Depending on the used transport protocol, a handshake between the two sockets may be started once the association is complete, which means that messages are exchanged to set up the connection. A well-known example for this is the TCP handshake.

If the association has been established successfully, data can be exchanged between the local and the remote socket. This can be done using the functions `send` and `recv`.

If the socket is not connection-oriented, e.g. in the case of a datagram socket communicating over UDP, association is not mandatory. Alternatively, the destination address has to be specified as part of the `sendto` function, which takes a socket address as parameter. This allows an application to send data to multiple destinations on the same socket. Receiving data on an unconnected socket can be done using the `recvfrom` function, writing not only data to a buffer, but also the source address to a socket address buffer.

Once the application decides to terminate the connection, the `shutdown` function can be called to disable further send operations, receive operations or both. It is also possible to call `close` on the socket file descriptor to terminate the connection.

An overview of all functions, their parameters and return types is shown in Appendix A.

2.2.2. Name Resolution

In many of the presented calls, socket addresses play a prominent role. They must be provided by the application, often as the result of user input. In most cases they contain numeric identifiers, such as IP addresses, which are inconvenient to remember. As identifiers, host names are much easier to use. For example, in web browsing, we most commonly do not enter the IP address of a remote web server for accessing a website, but use a URL that contains its host name. Before the browser can connect to the web server, it has to find out its IP address. This is the purpose of name resolution, a process where host names are resolved to one or multiple network addresses, also called lookup. Reverse lookups that resolve IP addresses into host names are also possible. For both of these processes, a name server is queried, most commonly DNS server.

Applications can perform name resolution through functions of the resolver library. This library is considered part of the Socket API, but not strictly integrated, as its calls can be used without creating a socket.

The most versatile and commonly used function of the resolver library is `getaddrinfo`. Its arguments and return type are also shown in Appendix A. It can take a host name or IP address to resolve, returning a list of socket addresses. It can also resolve service names into port numbers on the local host, returning available local addresses with the appropriate port number matching the service. Thus the resolver can provide useful information for both the local and the remote half of the association. By setting the "hints" parameter for the lookup, applications can specify whether they need local or remote addresses, whether they require a certain address family and much more information. The response is a list of socket addresses which match the query.

Note that a call to this function is not performed on a specific socket. This implies that a socket does not need to exist yet when `getaddrinfo` is called. Within the program, it can simply be called at any point before its information is needed.

For instance, it can be called directly before `connect` to find a suitable destination address for a socket where the local half of the association already exists. In this case, the application has already given the socket an address family at creation time, so it now needs to query for remote addresses of the same family. Another possibility would be to call the resolve function before any socket has been set up. In this way, one could get a list of addresses of multiple families, e.g. if the host has both IPv4 and IPv6 addresses available, and try to connect to all of them. For each returned address, one would first set up a socket, then try connecting it and destroy the socket if this has failed. In this way, an application could be flexible to use either IPv4 or IPv6, depending on what is available. This also means that the order of socket calls should not be regarded as fixed, but there is some possibility to vary it.

2.2.3. Socket Options

As shown in Section 2.2.1, sockets are created by providing some basic parameters such as their communication domain and type. It is possible for applications to further tweak their properties and behavior by modifying their socket options. Many extensions of the Socket API have been implemented using them.

Any option exists on a certain level, which specifies whether the option applies to the socket's general properties or to the behavior of a specific protocol used by it. In the first case, the level would be `SOL_SOCKET`, e.g. where one can instruct the socket to let others reuse its address or to allow broadcast. In the second case, the level would correspond to the protocol, e.g. `IPPROTO_TCP`, where one can specify details such as the congestion control algorithm to use.

To set an option, `getsockopt` is used with the level, option name and value of the desired option. The option value can have any type and length, as defined by the part of the OS where it is later processed. It is possible to query for a socket option using `setsockopt` providing similar parameters.

Since the Socket API is incorporated in the file API in most OSes, it is also possible to change some parameters of a socket using calls that operate on files, such as the `ioctl` or the `fnctl` interface.

2.3. Networking Implementation in Unix-Based OSes

Generally, the task of an OS is to manage hardware resources on computing devices and provide services to applications. Consequently, managing networking devices and granting applications access to them is also part of this task. Furthermore, OSes manage connections and hand received data to the correct application. As stated in Section 2.2, most of them implement the Socket API.

Most OSes distinguish between at least two protection domains of their memory, kernel space and user space. In kernel space, the operating system loads and executes tasks of the system's core functionality, e.g. the access to hardware resources. User space is the

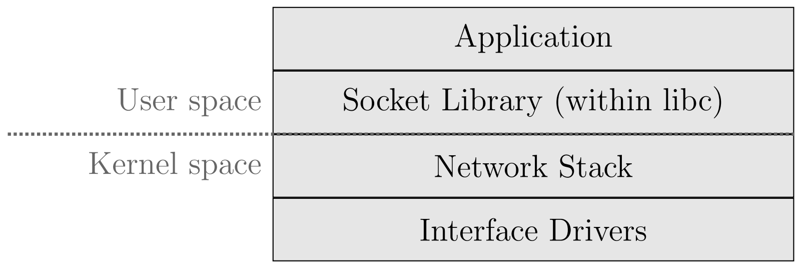


Figure 4: Scheme of the networking implementation within modern OSes

part where user mode applications are run. If such an application accesses a functions from kernel space, this is called a system call.

Figure 4 shows the components of the networking implementation of a Unix-based operating system. Some of them lie in user space, some in kernel space. Applications run in user space and make calls to the Socket API, as explained in Section 2.2. The functions that they call are implemented within the Socket library which still resides in user space and is part of libc, a standard library. The Socket API thus forms the interface between the application and the socket library, and it remains the same across many OSes, enabling them to run independently of the actually used OS. The implementation within the socket library, however, may differ across OSes or different versions of them.

From within the socket library, system calls are performed, i.e. functions in kernel space are called. These system calls differ depending on OS and CPU architecture.

Within the network stack implemented in the kernel, packets are assembled to be sent to the network or disassembled to be received by the application. This is done by creating protocol headers and adding them to messages, or reading protocol headers and processing them. Messages are handed to or received from device drivers or networking hardware such as network cards, wireless cards and UMTS sticks. All of the structure and implementation within kernel space is heavily dependent on the actual OS. The components in user space may be similar across OSes, making changes there much easier to port.

2.4. Possible Points of Modification

In the previous sections, we have discussed networking from an application's perspective in great detail. With this in mind, we are interested in interface selection and enhancing this decision with additional information provided by the application.

As discussed in Section 2.2.3, socket options provide a generic interface that allows applications to pass arbitrary information through it. This part of the API is an obvious choice for extending by defining additional options on an existing level or by adding a new level new that socket options can be set on.

Regarding interface selection, we note that local addresses are usually configured on a specific interface. This means that the choice of a specific local address implies the use of its associated interface.

As already mentioned in Section 2.2.1, applications can set up a connection without explicitly choosing a local address. In this case, the OS chooses a local socket address for it.

This choice is made within the kernel using a simple policy that depends on the available local interfaces and on the kernel's routing table. For source interface selection, this policy's decision must be overridden in some way. This can be done either by modifying the kernel, discussed in Section 2.3, or it can be done by calling `bind` for the application before the default policy can come into effect.

Similar to the local host, it is also possible that a remote destination host has multiple addresses. As explained in Section 2.2.2, most applications rely on name resolution to choose a destination address when setting up a connection. The `getaddrinfo` function can return a list of possible addresses, which are typically tried in the order that they are given. The result is that in the end, the one that is finally used is the first one from the list where connecting succeeded. The order of the addresses in the returned list are thus likely to play a role in destination address selection. By default, this order is determined somewhere within the kernel or the resolver library. However, it would be possible to reorder the list before handing it to the application to bias destination address selection.

In summary, the Socket API enables operating systems to provide a common and standardized interface to applications, which they can use for network communication. Within and beneath this interface, modifications can be made to influence the behavior of the system.

3. Related Work

This section gives an overview of related work regarding the potential benefits and challenges of using multiple interfaces. It also presents different approaches to giving the application more control over connections by expressing needs or preferences. For both of these topics, several use cases and existing solutions are presented. Some extensions of the Socket API, especially concepts for leveraging the presence of multiple interfaces or knowledge from the application for improving service towards it, are introduced and delimited from the Intents approach.

3.1. Interface Selection and Management

The presence of multiple interfaces on a host enabling it to connect to different access networks yields a lot of potential, but also poses a number of implementation questions. A problem statement for multiple interfaces [4] illustrates the potential challenges to implementers of an OS. In addition to the selection of addresses and interfaces, the issue of distinct provisioning domains is outlined there. With multiple access networks being available, the device could receive conflicting configuration information. The resolution of such conflicts is not standardized, so different OSes are likely to find different solutions. Other challenges include possible DNS resolution complications, e.g. when a name is only resolvable on a certain interface of a host.

In RFC 6419 [24], current approaches implemented on mobile and desktop OSes are presented, which include centralized connection management, per-application connection settings and altering of the network stack. Our framework addresses some of these problems as well, especially address and interface selection, while others such as routing issues are out of scope. However, we aim for a solution that is compatible with multiple operating systems.

3.2. Expressing Application Preferences

On most modern operating systems, applications do network communication through the Socket API, as presented in Section 2. There is literature available covering its implementation [20] which shows, e.g., the relationship between the Socket API and the protocol suites. Furthermore, the IPv6 extensions [8, 19] for the socket interface provide an insight of how an extension of this heavily used API could look like.

Some Socket API extensions have been proposed that enable applications to express preferences in terms of socket address selection in the context of IPv6 multihoming [16,23]. Here, an application can influence the choice of source address and override possible default policies for source address selection by setting flags within socket options. Design alternatives, e.g. regarding the scope of the preferences, and implementation considerations are discussed in detail while the necessity of incorporating name resolution is highlighted. However, these address selection policies mostly regard the scope of the address, whether it is temporary or permanent, and mobility considerations. The Intents approach aims to allow for more criteria, e.g. to take different characteristics of the links that belong to different interfaces into account. It is not limited to address selection, but also wants to tune network parameters or bundle interfaces. Furthermore, the rules proposed there are rather static with only some possibilities to alter them, while we prefer a more flexible set of rules.

Somewhat inspiring the term Intents, an application-informed approach to source interface selection is taken by Intentional Networking [9]: They enable applications to label their connections with expected characteristics, such as whether it contributes towards an interactive user application as "foreground" traffic or whether it is a transfer operating in the "background". These labels are not attached to the socket as a whole, but to a sequence of ordered messages sent on the socket, between which the application can distinguish using message annotations in its send calls. For each such message sequence with a certain combination of labels, the most appropriate interface is determined according to a static policy, and the data is sent over this one. Multiple such ordered message sequences can exist within a single socket, and they can have different labels. Furthermore, Intentional Networking allows message sequences to be delay tolerant and to have different ordering constraints. For respecting these constraints, meta-information is sent as message overhead along with the data and thus support from the peer is required. In contrast, our approach operates on a whole socket and focuses on interface selection and parameter tuning on the host that it is implemented on. Mandatory behavior specification such as ordering constraints are out of scope for Intents, and they do not require support from the communication partner, making them easier to adapt.

Another approach to communicating an application's needs through the Socket API defines sets of desired characteristics and assigns a protocol number to each of them [25]. Setting one of these protocol numbers on a socket, an application is able to place its traffic into a certain category, enabling the networking implementation to base interface selection or parameter tuning on it. However, the mapping between every possible combination of traffic characteristics and the protocol numbers has to be made available to application developers and kept up to date at all times, introducing usability issues and potential incompatibilities.

3.3. Quality of Service

The principle of expressing and satisfying the requirements of an application has also been made in the context of Quality of Service (QoS). The two predominant concepts here are Integrated Services (IntServ) and Differentiated Services (DiffServ). Both of them aim to provide certain service guarantees to applications, such as assuring a delay below n milliseconds for each packet of a delay-critical connection. IntServ [26] make use of a resource reservation protocol along the path of a connection, while DiffServ [15] provide a classification scheme for IP packets which are then handled accordingly on intermediate devices, e.g. by using different scheduling algorithms.

In the context of QoS, some approaches exist that communicate application requirements through the Socket API in a quite detailed way. They introduce new data structures that contain constraints or requested parameters of the connection and pass them to the networking implementation through extended socket calls. QSockets [2] provide a "QoS-enhanced Socket API", enabling applications to directly set parameters for queueing and scheduling on socket setup. Furthermore, they enable the annotation of messages with flags, e.g. requesting a certain delivery time. Another approach [18] stems from the Future Internet architecture context. It defines a comprehensive list of possible requirements as conjunctions of effects, operators and attributes, e.g. "delay", "less than"and "100ms". These requirements are placed within a list and passed through the Socket API at connection time.

While Intents also express applications' expected traffic characteristics or categories which can then be treated differently to provide adequate service, they are not a concept to guarantee certain service parameters. Rather they aim to improve the service to applications on a best-effort basis, optimizing the use of the available resources according to the application's needs. It would be possible to take advantage of QoS technologies from within the Intents framework if they are available. However, the framework does not rely on it, as different access networks on different interfaces already offer different QoS characteristics on a best-effort basis by default.

3.4. Using Multiple Interfaces in Parallel

Instead of having to choose between interfaces, there are protocols are able to use multiple ones in parallel. A fairly recent example is MultiPath Transmission Control Protocol (MPTCP) [7,17], a modified version of the transport protocol TCP. It attempts to use multiple available interfaces in parallel by establishing a TCP connection to the same peer through each of them, pooling the links' capacities while being transparent to the application. To be compatible with existing applications, the changes to the Socket API are kept to a minimum, but nevertheless applications can explicitly enable MPTCP through the Socket API if the system supports it. As such, MPTCP is orthogonal work to this thesis: For instance, it is one example of how multiple interfaces could be bundled to improve service for connections with a high bit rate to meet an application's needs.

Another transport protocol designed for this is SCTP [21,22], which is message-oriented like UDP, but ensures reliable delivery of messages like TCP. An association is established between two SCTP endpoints, each with its own socket address. However, the presence of multiple socket addresses means that there are multiple potential SCTP endpoints. In this case, multiple associations can be established on a single socket, which often implies the use of multiple interfaces in parallel. This is called the one-to-many style interface, as it allows multiple associations on a single socket. Other than MPTCP, this mode requires extensive support from an application, which may need to specify multiple local addresses it wants to use and manage the individual streams by itself. Like MPTCP, SCTP is orthogonal work to this thesis.

3.5. Mobile Data Offloading

Multiple access connectivity allows us to offload a portion of the traffic on one access network to another access network through which the communication peer is reachable as well. A popular example of this is cellular mobile providers moving traffic from their heavily used 3G networks to parallel WiFi networks to relieve them from some of their load. This topic has gotten a lot of attention recently in the research community as well as in the industry [6].

In the research community, offloading has two major aspects: Interface selection and delay tolerant networking. The former focuses on using the "better" interface when it is available and in cases where it is appropriate for the application. The latter goes further by modifying applications to defer their communications to a point in time where the "better" interface becomes available. It may also involve changes to the way that information is distributed and addressed.

Lee et al. [12] demonstrate via a quantitative study the performance benefit of offloading 3G mobile data to WiFi networks. They find that while relieving the 3G network of about 65% of the mobile data traffic, 55% of battery power is saved on the client devices. Even more gains are possible by leveraging delay tolerance, deferring data transfers for a certain time in situations where no WiFi network is available. Reflecting the mobility patterns of average users, they report further gains of more than 20% when the timeout for such transfers is set to 1 hour or longer.

Balasubramanian et al. [3] design a system called Wiffler to augment mobile 3G capacity. They leverage delay tolerance to overcome availability and performance problems of WiFi, aiming to reduce overall 3G usage. Furthermore, they respect the sensitivity of some applications to delay or loss by performing fast switching between the two networks. From their experiments, they report a reduction of 45% of 3G traffic when setting the delay tolerance to 60 seconds.

Delay tolerant networking does not seem to be compatible with Intents. The reason is that it requires major modifications to the application and possibly also to the network, while Intents are a purely host-based concept. However, we think that interface selection, as an aspect of offloading, is a conceivable use case for Intents. On the whole, they may be quite useful for basing decisions on the need of the application.

4. Intents

There is a large variety of applications, each of which has certain requirements regarding the preferred properties of its network connections. These requirements can be quite diverse, but are typically known to the application itself.

The operating system wants to provide an optimal service to the application and to the user. However, what "good" performance means in practice largely depends on the needs of the application. If a system does not know about these wishes, it is more difficult to fulfill them. Thus when it makes choices such as selecting a source interface, it would be advantageous to take the application's preferences into account.

The purpose of Intents is to bring the application and the system together in this regard. Intents are pieces of information about the preferences or expectations regarding an application's communication on a per-connection basis. They can be expressed to the system in order to enable it to better meet the needs of the application.

Intents should be understood as an assistance for the OS, not as constraints to be satisfied. They are accounted for on a best-effort basis and do not imply any hard requirements or QoS guarantees. For some Intents it might not even be possible to quantify their "success". Even if they express a requirement that cannot be fulfilled by the system, this should not impair connection establishing and the application should continue to function in any case.

As a purely host-based concept, Intents do not require any support by the network or by the communication peer. In particular, this means that peers are not required to implement Intents as well.

In this section, Intents are introduced as a concept for such pieces of information. After presenting their principle, some concrete examples of what could be communicated are shown. Finally, the question of how to implement Intents and integrate them into the Socket API is discussed.

4.1. Approach to Intents

Aiming to explore which information is usable as Intents, a few design choices have to be made.

Concerning the scale of the information, Intents are defined on a per-socket basis[1] rather than per application, and an application can set different Intents of each of its sockets. The reason is that multiple sockets can have very different purposes and thus different requirements. For instance, a browser can open a socket to retrieve constrasting kinds of data, such as fetch small objects on web sites or retrieve a long video stream.

Intents can describe characteristics of the traffic of a connection, but are not in itself a complete definition of these characteristics. Pieces of information that are already given, e.g. those required for choosing a transport protocol, should not be duplicated. Consequently, reliable delivery, the ordering of packets, and the preservation of packet

[1]A per-socket basis often, but not necessarily, corresponds to a per-connection and per-flow basis. A flow is here understood as all packets with the same source address, source port, destination address and destination port. However, flows can be bundled using protocols such as MPTCP, so there can be multiple flows per socket.

boundaries are unsuitable as Intents. Moreover, these are mandatory requirements that lead to the choice of a specific protocol and are thus incompatible with the optional and best-effort nature of Intents.

Instead of defining how messages of the flow are handled, the goal of Intents is to help aiding decisions such as interface selection or parameter tuning on a socket. As these decisions are related to association, they are made before a connection is established, so it is useful to set Intents before this. However, applications are allowed to set Intents on a socket at any point in time. Providing them is voluntary and it is highly unlikely that information for every possible Intent is or can be set on a given socket.

4.2. Intents of our Prototype

Before we go into the details of the Intents implemented in our prototype, let's start with some example use cases:

(i) If the antivirus software needs an update this usually implies downloading a large file. This is a bulk transfer of an object with a certain file size, which the application may know or be able to query before retrieving the data. Timing is typically noncritical, since the download most likely runs in the background instead of being part of a direct interaction between machine and user. As long as it completes eventually, its finishing time is improbable to have a significant impact on the user experience. Moreover, if the the TCP connection was interrupted during the transfer but then continued after a short time, the application would not suffer. For this connection the application can set "bulk transfer" as a general category. Additional information provided can be file size, timeliness, and resilience against connection loss. As a consequence, the system might take these cues into account by choosing an interface with high bitrate but high delay over an interface with opposite characteristics. It could also refrain from taking additional precautions against connection interruption, preventing it from causing unnecessary overhead.

(ii) If you want to watch a video online, it usually means establishing a streaming session over which video and audio data is transmitted continuously. The duration of this transfer may be known to the client in advance. Furthermore, the application may know the bitrate or be able to infer it, e.g. from the choice of codec and resolution. Connection interruption can have an effect that is visible to the user, greatly affecting user experience in a negative way. Especially in the case of live streaming, connection loss is undesirable or even disastrous. Knowing this, an application can put this connection into the general category "stream" with additional information about duration, bitrate and resilience requirements. Making an effort to provide the best possible service to the application, the system could choose an interface that supports the required bitrate, bundling multiple of them if necessary. It could also take precautions to improve resilience against loss by using multiple interfaces in parallel.

These examples show different kinds of information on different abstraction levels that are suitable as Intents:

- *Categories*: Broad classes of traffic
- *Detailed flow characteristics* such as file size, duration or bitrate
- *Expectations or tolerance* e.g. regarding delay or connection interruption

Intent	**Type**	**Possible values**
Category	Enum	Query, bulk transfer, keepalives, control traffic, stream
File size	Integer	Number of bytes transferred by the application
Duration	Integer	Time between first and last packet in seconds
Bitrate	Integer	Size divided by duration in bytes per second
Burstiness	Enum	Random bursts, regular bursts, no bursts or bulk (congestion window limited)
Timeliness	Enum	Stream (low delay, low jitter), interactive (low delay), transfer (completes eventually) or background traffic (only loose time constraint)
Resilience	Enum	Sensitive to connection loss (application fails), undesirable (loss is fairly inconvenient) or resilient (loss is tolerable)

Table 1: List of proposed Intents and characteristics

An overview of the Intents implemented in our framework is shown in Table 1. Some of these pieces of information can be expressed as integers, i.e. as numeric values, some as enumerations, i.e. as one of multiple possible values to choose from. They are based on information that may be readily available to some applications and are both easy to set and useful as cues for the system. Moreover, it is straightforward to add more Intents in the future.

Detailed considerations While technically, burstiness is also a detailed flow characteristic, it is not as straightforward to express as a numeric value because there is no consensus definition[2] [11]. This is why an intuitive enumeration of possible values was chosen for this detail.

Concrete numbers for these values are typically only known for certain types of connections. For example, knowing the duration of a flow in advance makes sense for a video stream, but not for a query. Knowing the file size in advance is possible when a discrete object is downloaded, but not for a constant flow of control traffic. If there is no concrete value, applications could choose to use an estimated number instead.

[2]For instance, burstiness as a property of the flow can be defined as variance burstiness, RTT burstiness, or train burstiness. For computing the variance burstiness at a time-scale T, the flow is divided into bins of duration T, and s is the number of bytes sent within a bin. The variance burstiness is the standard deviation of all s, but its value is largely dependent on the choice of T. On the other hand, RTT burstiness defines the burst size as the number of bytes sent in each RTT of the flow, and is computed as the product of mean burst size and average RTT. Train burstiness divides a flow into trains of packets and measures burst size, duration, rate and inter-burst time.

More abstractly, applications often can intuitively estimate whether a flow characteristic is "high"/"large"or "low"/"small". Lan and Heidemann [11] divide flows in the following ways:

- by size into elephants and mice
- by duration into tortoises or dragonflies
- by rate into cheetahs and snails
- by burstiness into porcupines and stingrays

Using such labels as Intents by themselves is not advised, as their significance is limited if the threshold between "high" and "low" values is not defined. However, they help to intuitively classify flows into categories. For instance, the combination of "high" file size and "high" rate fits well with our perception of a "bulk transfer". Since some of these labels have a strong correlation [11], there are some combinations that are more likely and some that are rather unlikely. Thus not all combinations make up a possible value for the category Intent. Contrarily, all possible categories imply assumptions about some of the characteristics of their flow. Furthermore, for each category there are some other Intents that are likely to be set along with it. An overview of this is shown in Table 2. Note that these are only suggestions and applications are free to set or not set any Intent that they deem appropriate.

Category	**Assumed Flow Properties**	**Likely accompanying Intents**
Query	Small, short	Desired timeliness, resilience to connection loss
Bulk transfer	Large, short, fast, bursty	File size, desired timeliness, resilience to connection loss
Control traffic	Large, long, slow, bursty	Duration, burstiness, desired timeliness, resilience to connection loss
Keepalives	Small, long, slow, not bursty	Duration, desired timeliness, resilience to connection loss
Stream	Large, long, fast, not bursty	Duration, bitrate, desired timeliness, resilience to connection loss

Table 2: Traffic categories and samples of how Intents can be combined

4.3. Implementation

Based on the principles and the general philosophy of Intents presented previously, there is the question of how to incorporate them into the Socket API.

For a Socket API extension, one has the general choice between adding parameters to existing functions and making new function calls, e.g. by setting socket options. As mentioned in Section 2.2.1, there is already some information passed within the socket setup call, such as type and address family. However, adding more parameters there would be antithetical to the principle that all Intents are optional, since in this case an application would be either forced to specify the Intent or provide a default value. This would complicate the interface from the point of view of the programmer. Consequently Intents were implemented as new socket options, which are an obvious point of extension to the Socket API as explained in Section 2.2.3. In this way, the completely voluntary nature of Intents is taken into account and existing socket calls are not modified for them, keeping the overhead to a minimum. Socket options can be freely set at any point within the application, which conforms to the principles of Intents. Furthermore, arbitrary option values can be passed through socket options, accounting for the different types that Intents can have. Furthermore, if an application attempts to set Intents on a host that does not support them, the function will return a "not supported"error. In this way, the application can react gracefully instead of crashing, which is in accordance with the philosophy that Intents should be handled as an optional help for the OS and not provide any constraints.

By specifying the level on which the option applies, one can influence the part of the networking implementation that processes the option, minimizing the risk of influencing other socket options. Intents are not related to any specific protocol, so they do not belong on one of the protocol levels mentioned in Section 2.2.3. The socket level refers to more generic properties of a socket, but contains mostly requirements and practical instructions for handling connections. The nature of Intents differs from this, so a new socket option level is introduced: `SOL_INTENTS`. This also prevents having to pay attention to not reuse an existing option name. Option value types are defined according to Table 1 in Section 4.2. An overview of the Intents as they can be used by applications is shown in Appendix B.

How Intents are handled within a host, e.g. how they aid policy decisions, is explained in Section 5.2 as it closely relates to the other modifications of the Socket API.

5. Design and Implementation

As shown in Section 2, there are several choices which applications can either make themselves or let the network implementation decide, such as source address selection. If applications make these decisions themselves, they typically do not have much knowledge about the system's state and thus make an effort to query it or pick a "good" or "bad" option more or less by chance. Moreover, if an application implemented a policy-making entity, others would have to duplicate it to make use of it. On the other hand, in unmodified OSes, letting the decision be made within the networking implementation leads to a simple, static policy being applied. Without Intents, presented in Section 4, there is not much knowledge about the application's needs and expectations there.

It appears that such decisions should be made by an instance that is within the OS, so that multiple applications can make use of it. However, it should receive some hints from them in order to adjust its decisions to the individual requirements of the applications. Furthermore, it can make use of comprehensive knowledge about all applications that use it. Since in different scenarios, a varying number of links with different properties may be available, it is desirable to make policies interchangeable.

This section shows the design and implementation of a framework to make and apply such policies, facilitating source and destination address selection as well as further tuning of network parameters while taking Intents from the application into account.

5.1. Components

Our framework consists of three components: the Socket Intent Library, the policy manager called Multi Access Manager (MAM) and the policies, see Figure 5. Each of these components is implemented in C and compatible with Linux as well as MacOS.

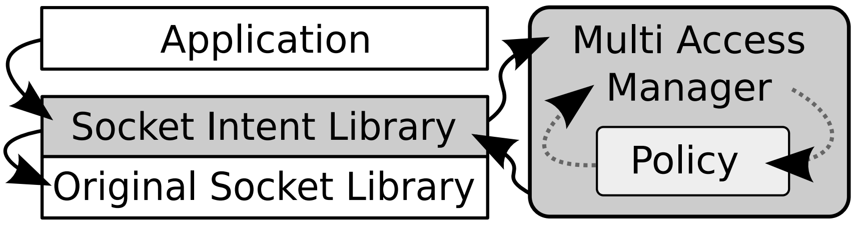


Figure 5: Components of the framework.

The Socket Intent Library provides an augmented Socket API which is largely based on the original Socket API described in Section 2. Its task is to override the simple, static policy decisions that are applied if an application does not make the decisions itself. This is done entirely in user space in the way described in Section 2.4, and the kernel is kept unmodified. All of our framework resides in user space, as this is simpler and easier to port, e.g. to machines with different OSes. The library is implemented on top of the original Socket Library, handling calls from the application first and passing information and instructions down the networking implementation through the original Socket API. For requesting policy decisions, the Socket Intent Library communicates with the MAM.

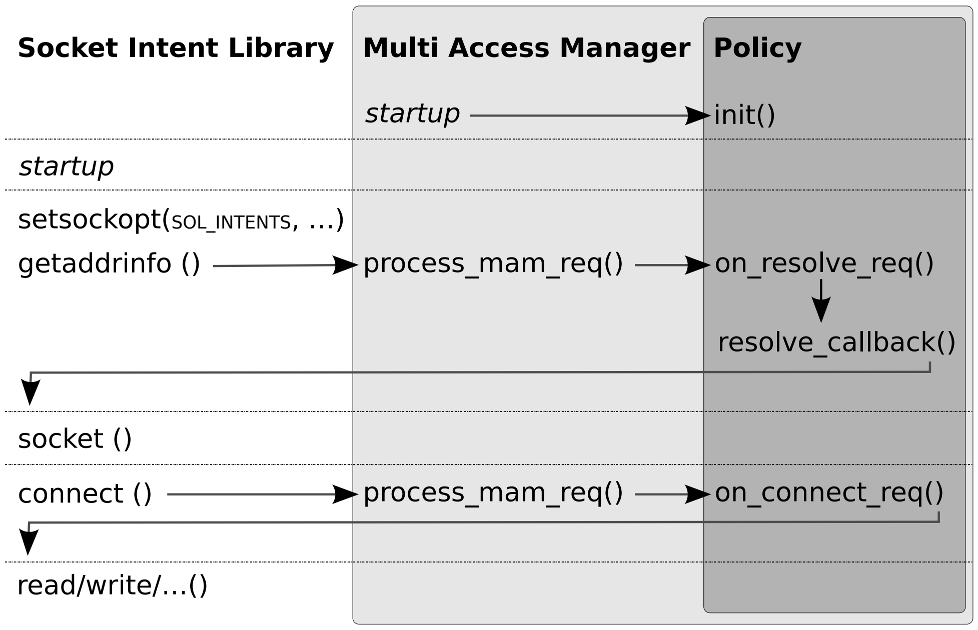


Figure 6: Interaction of the Socket Intent components.

This is a policy manager that runs on the system as a standalone daemon and can be contacted by multiple applications. The MAM loads one of the interchangeable policy modules, which are the components making the decisions while having a maximum of usable information available. It then hands the results back to the Socket Intent Library, which applies them.

Summarizing the flow of information between the components of the framework, Figure 6 shows the functions for setting up policies, performing requests and handling them.

The following sections present the individual components' design and implementation in more detail.

5.2. Socket Intent Library

The Socket Intent Library provides an enhanced Socket API to applications, enabling them to delegate decisions to the policy. Furthermore, the library supports Intents as a way of aiding policy decisions. It is implemented as a shared library that can be linked by applications.

The Socket Intent Library implements an interface that is largely based on the original Socket API described in Section 2.

Its modifications are motivated by the decisions that the framework wants to delegate to the policy manager and then apply on the connection, such as source and destination address selection. The focus lies on applications that proactively connect to another application, performing the steps shown in the diagram of Section 2.2.1. The main points of modification are those where the association is made and where socket options, e.g. Intents, are set.

5.2.1. Interception of Function Calls

Some of the Socket API calls, such as `bind`, `connect`, `getaddrinfo` and `setsockopt`, are modified in such a way that when an application calls one of these functions, they are handled by the Socket Intent Library first. A first intuition was to use the `LD_PRELOAD` environment variable to override socket calls. However, there is a problem with this approach. Some state about the socket needs to be kept within the Socket Intent Library, e.g. to store whether an application has already called `bind`, or to store Intents along with other information that is later useful for aiding policy decisions. Consequently, calls that belong to one connection have to be linked to each other. This is straightforward for calls that operate on a socket file descriptor, since it stays the same for any call on that particular socket. However, `getaddrinfo`, which is crucial for destination address selection, does not take a socket file descriptor. Worse yet, it can be called when no socket file descriptor exists yet, as explained in Section 2.2.2. Since applications can call several functions on sockets in an irregular order, it is hard to guess which other socket calls a `getaddrinfo` call belongs to.

To address this challenge, an additional parameter is passed to `getaddrinfo` that helps linking it to other socket calls for the same connection. This leads to the creation of a new function with a changed signature, which needs to be invoked directly by the application. Such a parameter could be a socket file descriptor similar to the other Socket API calls, but this is difficult if the socket does not exist at the time when `getaddrinfo` is called. To solve this problem, a new data structure, the socket context, is introduced and used instead of a socket file descriptor. This data structure is referenced by the application and used within the policy. Discarding the `LD_PRELOAD` solution, we add the socket context as a parameter to all socket calls that belong to a connection, linking them to each other and storing all important information about the connection. A complete definition of the socket context is provided in Appendix C.1.

For instance, the socket context stores which calls have already been performed on the socket, e.g. whether an application has already called `bind` explicitly. Intents and other socket options are stored there, too, as well as the socket's type and protocol. Furthermore, the socket context contains the relevant information for association, such as the local socket address if it has already been set, and the remote socket address once it has been decided. To gather all of this information, more socket calls have to be modified, not only the ones where a policy decision is requested. This means that for example, the `socket` function is modified to store its information within the socket context.

5.2.2. Policy Decisions

To request decisions from the policy, the Socket Intent Library must contact the MAM. The communication between the two components is facilitated by sockets within the local domain, the Unix domain sockets mentioned in Section 2.1. When a decision has to be made by the policy, the whole socket context is sent from the library to the MAM, along with the request type. If invoked from `getaddrinfo`, the type is "resolve request", if from `connect`, it is "connect request". As one local socket can be used for multiple such requests, a socket context also gets a context ID which is unique per application and socket in order to match responses from the policy to requests. Going further, the policy's responses, such as chosen source address, destination addresses and suggested socket options are also stored within the socket context.

Source Address Selection A host may have multiple interfaces to choose from, and source address selection decides which one of them to use for a given connection. This decision comes into effect using the `bind` function. If an application has performed source address selection by itself and calls `bind` explicitly, it might later rely on this choice. While it would still be possible to override this decision, we chose to respect it and refrain from doing source address selection via the policy in this case. If the application does not choose a source address, we want to delegate this decision to the policy. The latest point at which we can do so is directly before connecting, where we can be sure that the application does not plan to do source address selection by itself anymore. At this point, we want to override the static default policy that would be made and applied in the kernel. Consequently, we modify the `connect` function to make a request to the policy before establishing the connection. Along with the request, information such as the required address family and the Intents set by the application are communicated to the policy. If the policy then suggests a source address, we `bind` to that source address before completing the association by calling the original Socket API's `connect`.

Destination Address Selection The destination address that an application wants to connect to is given in the `connect` call. However, this address is often the result of name resolution, as explained in Section 2.2.2. In this case, `getaddrinfo` is called on a host name to obtain a list of candidate destination addresses before connecting to one of them. As mentioned in Section 2.4, it can be assumed that the list of addresses is tried in the order that it was returned to the application. This implies that by changing that order, one can bias the application's final decision. Thus, modifications for destination address selection are made within the `getaddrinfo` call, where a resolve request is sent to the MAM. We chose to provide all information necessary for name resolution to the policy, letting the policy do both name resolution and the reordering of the result. The reason is that multiple name servers may be reachable on the different interfaces, and a decision has to be made which one of them to query over which interface. Furthermore, the result from the first answering server could be used as is, or one could wait for the other name servers to respond as well. We decided to entrust these decisions to the policy, where detailed knowledge about the different interfaces is available anyway. However, if the policy fails to do name resolution and does not return a list of destination addresses, the Socket Intent Library resolves the name by itself. In this case, there is unfortunately no policy influence, but at least the application has a chance to get a valid destination address and can continue operating.

Socket Option Handling Concerning socket options, two cases have to be considered. On the one hand, socket options, particularly Intents, can be provided by the application to be communicated to the policy to aid its decisions. On the other hand, they can be communicated by the policy as an effect to apply on the socket. For the first case, if an Intent is set on a socket, the Socket Intent Library needs to handle it. This means that the `setsockopt` call needs to be intercepted. Intents are recognized by their socket option level as explained in Section 4.3 and are stored within the library. Other set socket options are stored as well, but also handed down in the network implementation to be applied right away. At this point, the set socket options are not communicated to the MAM for an immediate policy response, as they should later aid policy decisions made during other calls. Thus, a `setsockopt` call does not request a decision from a policy by itself, but the information from it is stored within the socket context and affects policy decisions later. In general, the library gathers a maximum of information and later sends all of it, including the Intents, to the policy along with resolve or connect request. In addition to suggesting a source or destination address, a policy can also suggest socket options for the connection. These options are set when the other results of querying the policy are handled, i.e. within the `getaddrinfo` or `connect` call of the Socket Intent Library.

5.3. Multi Access Manager

The MAM is a policy manager that receives requests from applications via the Socket Intent Library and hands them to the policy that is currently in effect. It then communicates the policy's decisions back to the library. The MAM does not make any decisions by itself, but hosts the policies that make them. Arbitrary policies are supported by the MAM, as long as they implement the interface presented in Section 5.4.2. Implemented as a standalone application, only one instance of the MAM can be running on a host, e.g. as a daemon.

5.3.1. Initialization

To be able to provide a maximum of information to policies, the MAM is designed to gather as much knowledge about the state of the local system as possible. This begins immediately when it starts up and can be facilitated via system calls, configuration or gathering statistics. The MAM stores its information in an internal data structure called the MAM context.

When started, it detects which interfaces are present on the system and which socket addresses are configured on them. Then it stores a list of available source prefixes in the context, each annotated by the corresponding interface name and a list of socket addresses with that prefix that are configured on the interface.

The MAM operates on a list of prefixes[3] instead of a list of interfaces or socket addresses because this is the granularity with which a policy wants to distinguish between socket addresses when making decisions.

Multiple addresses with different prefixes can be configured on a single interface. They may have different address families, so only one of them applies to a policy request on a socket with a given address family. Thus it is advantageous for policies to store addresses with different prefixes separately, as for the purpose of decision making, they are not interchangeable. However, it is also possible that multiple addresses with the same prefix are configured on a single interface, e.g. if some of them are temporary in nature. One example for this are the IPv6 Privacy Extensions [14], which avoid using the exact same

[3]A network address typically consists of two parts: Network prefix and host number. In most cases the address is accompanied by a netmask, which specifies how many bits of the network address count towards the first part, the prefix. In general, the prefix is understood to identify a subnet, i.e. a portion of a network as a whole, and the host number specifies a particular host within this subnet.

address over a large period of time by changing the host portion once in a while. As long as the address that is chosen by the policy is valid, addresses with the same prefix are equivalent with respect to policy decisions. In summary, policies may want to distinguish between addresses with different prefixes, but they can treat addresses with the same prefix as equal. Consequently, the MAM provides a list of prefixes to the policy.

Furthermore, it reads the name server configuration of the system and stores it within the context.

When the global system information has been initialized, the MAM parses its configuration file. In addition to the name of the policy module to load, it often contains supplemental information for the policy. It can specify a list of prefixes that the policy should use, which are then looked up in the list of all source prefixes and marked as "active". Furthermore, the configuration file can contain arbitrary additional information about each prefix which is stored within the source prefix list.

After all of the information has been parsed, the policy module is loaded and its initialization function is called. At this point, it is provided with all of the information that were gathered.

At the end of the initialization phase, the local Unix domain socket is set up to accept client requests from applications, and the MAM moves into operation.

5.3.2. Request Handling

Acting as a server on the local system, the MAM serves requests from applications running on the host, i.e. resolve or connect requests. It does not make any decisions by itself, but instead delegates them to the policy module that is currently loaded.

Built on the principle of nonblocking socket IO, the MAM can receive and process requests from multiple applications simultaneously, while calls from one client do not impede others. Instead of the basic, blocking Socket API, the MAM uses the nonblocking libevent2 bufferevents API.

When it accepts a client, the MAM receives the entire socket context from it and first makes a local copy. Then it calls a function of the policy module, depending on the type of request that should be handled, i.e. resolve request or connect request. The copy of the socket context is passed to the policy as well, which modifies it. The modified version then contains the policy's decisions and is sent back to the client.

After the MAM has finished handling the request, it throws away the local copy of the socket context. Designed to be as stateless as possible, it does not keep track of the connections once they have finished.

5.4. Policies

One of the key questions within the framework is what policy to use. We note that the specific policy that is most beneficial may depend on the configuration of the host, the current location, the current network availability, etc. Therefore, we decided to make policies interchangeable and provide a generic framework in which one can develop, use and evaluate different policies.

5.4.1. Structure and Operation

A policy's logic is implemented within a dynamically loaded library, also called the module. The name of this module is specified in the configuration file for the MAM, along with additional information such as parameters for the whole policy or for individual prefixes. One policy can have multiple configuration files and thus the same logic can operate with different parameters, giving it the possibility to account for different scenarios using the same module.

The module implements at least four functions: `init`, `on_resolve_request`, `on_connect_request` and `cleanup`.

The `init` function takes the MAM context as parameter, from which it retrieves information about the current state of the system and about the additional information provided in the configuration file. It can use this information to set up its internal data structures, e.g. a list of candidate source prefixes that is already filtered to only contain those prefixes that the policy is configured to use. These internal data structures should be released within the `cleanup` function.

The decision making functions that handle resolve and connect requests have access to these internal data structures, but they can also read information from the MAM context. In addition, they receive the socket context that was sent from the Socket Intent Library as described in Section 5.2.1 to help their decisions. After the decisions have been made, this context is modified and sent back as a reply.

Generally, a function that is directly called from the MAM should return in a timely manner without blocking the entire MAM's operation in order to avoid impeding other clients' requests.

Policies are allowed to issue their own queries over the network, e.g. for name resolution, but for the above reason they are strongly discouraged to do so using blocking Socket IO through the basic Socket API shown in Section 2. Instead, policies should add their queries to the event base for nonblocking Socket IO that the MAM operates on as explained in Section 5.3.1. They can implement a callback function to be invoked when their query returns in order to complete the overall request.

This is the reason why sending back the modified context is done by the policy, not by the MAM. When a call to a decision making function of a policy returns, this does not necessarily mean that the request has already been fully handled. Situations can occur when the policy function returns, but has issued queries that have not completed yet and thus its callbacks have not been invoked yet. In fact, since the MAM supports arbitrary policies, it cannot make assumptions about the policy's internal implementation details, so it cannot know whether there are still callbacks pending. Thus it does not know when the request has been fully handled and should leave the task of responding to the policy.

5.4.2. Types of Decisions

Policies consist of a set of rules for making decisions such as source or destination address selection. They can take plenty of information into account, e.g. about the connection or about the interfaces currently available on the system. The decisions are made upon receiving a resolve request or connect request from an application through the Socket

Intent Library, as discussed in Section 5.2.1. They are communicated back to the library, where they can be applied on the socket.

Source Address Selection This decision is typically made when a connect request has been received. However, policies are advised to check whether the application has already set a source address by itself before making the effort. If this is not the case, the list of available local prefixes is scanned for the most suitable one, based on the policy logic and the information available to it. A valid socket address for this prefix is then communicated back to the Socket Intent Library as the suggested source address, to be used as part of the association there.

Destination Address Selection When a resolve request has been received, a policy can influence an application's selection of destination address. Having the query information available, name resolution is performed over one or multiple of the local interfaces. The policy decides how many interfaces to use and whether to wait for a reply on all of them, as it can abort the resolving step when it deems that it has sufficient information. Then, the received list of candidate destination addresses is ordered, moving the most desirable destination address, according to the policy, to the head of the list. This list is then communicated back through the Socket Intent Library to the application, where the addresses are typically tried in the order that they were given.

Socket option handling Having received either of the requests, a policy can choose to propose arbitrary socket options to be set on a socket. They can be required e.g. for bundling multiple interfaces by enabling MPTCP [17] or tuning parameters such as changing the congestion control algorithm. The list of proposed socket options is communicated back to the Socket Intent Library along with any other decision that the policy has made. Effects are applied on the socket directly before the connection is established.

5.4.3. Examples

A fairly simple application unaware policy would be to configure one "default" interface that one deems "best". In the list of usable prefixes in the configuration file, one could be marked as the default. Upon a connect request, source address selection could be performed, always suggesting this default address.

In Appendix C.3, the implementation of such a sample policy is shown, containing also a name resolution callback function to illustrate its use within the nonblocking socket IO scheme of the MAM.

Another policy that takes information from the application, i.e. Intents, into account, would be to use high bandwidth interfaces for application requests of the category "bulk transfer" and low latency interfaces for application requests of the category "query".

Here we make use of the Intents from the socket context to aid source address selection and choose among them based on labels such as "high bandwidth" or "low latency". Such additional information about prefixes must be provided somewhere, e.g. by specifying it in the configuration file or by having the MAM collect measurements on the interfaces, evaluate them and provide them to the policy.

6. Adding Applications and Policies

As shown in Sections 4 and 5, applications can express a wide range of preferences and useful pieces of information through the modified Socket API. These hints are then taken into account by a proactive policy in order to choose a suitable source interface, bias the destination address selection or tune parameters within the network stack.

Discussing the more practical aspects of this, this section provides guidance on how to incorporate Intents into an application, illustrated through the example of modifications to the HTTP client *wget*.

Furthermore, a variety of policies is needed to fully leverage the diversity of scenarios, taking into account the different information given in different use-cases. Consequently it is important that policies can be conceived and implemented in a simple and efficient way. This process is described in the second part of this section and demonstrated by showing and explaining the code of a sample policy.

6.1. Modifying an Application to Support Intents

Generally, any application that communicates over the network can be chosen to support Intents, as long as its source code is available. It only requires minor modifications to enable the application to benefit from policy decisions, and setting any of the Intents described in Section 4 is fairly straightforward.

One necessary step is replacing calls to the blocking version of the Socket API described in Section 2 with the modified ones described in Section 5.2. Next, developers choose Intents that suit their application, obtain meaningful values for them and set them through the modified Socket API.

6.1.1. Substituting the Socket Calls

The first change is replacing the existing Socket API calls with the ones that go to the Socket Intent Library, enabling it to enforce policy decisions for this application. This basically means substituting each function call to the Socket API with its counterpart of the modified Socket API presented in Section 5.2. Since the signatures are fairly similar, this is rather straightforward. The header file `clib/muacc_client.h`, which contains the signatures, needs to be included, as well as the utility functions in `clib/muacc_client_util.h`.

Each function of the modified Socket API takes one additional parameter, the socket context, which enables the Socket Intent Library to link function calls that belong to the same connection and to store information about it as explained in Section 5.2.1. The context has to be declared by the application and can be initialized explicitly as shown in Listing 1, or implicitly at the first time that it is used.

The same context needs to be supplied to all socket calls that belong together, as well as the corresponding `getaddrinfo`. However, it is possible to release and reinitialize a socket context explicitly, and then use it for a new connection. This can be useful in the case of multiple connections being made in succession, e.g. within a loop. Reusing an existing context for a new connection without reinitializing it is not recommended, as it may lead to anomalies. The context should be released when it is no longer needed to free its allocated resources.

```
muacc_context_t ctx;

muacc_init_context(&ctx);

[...]

muacc_release_context(&ctx);
```

Listing 1: Initializing and releasing a context, from wget/src/http.c

Furthermore, it is important that the context is available to all functions of the Socket Intent API, so the scope within which it exists needs to be chosen carefully. This is especially relevant if not all socket calls are invoked directly from the same function, but hidden within more complex helper functions specific to the application. For this, there are two possible solutions: Declaring the context as a global variable, or add it as an additional parameter to the function in question. For wget, the latter was chosen several times, causing the signatures of some wget-internal methods to change. The excerpt from the code in Listing 2 illustrates this.

```
int connect_to_host (muacc_context_t *ctx, const char *host, int port)
{
  /* Initialize muacc context */
  if (ctx == NULL)
  {
    ctx = malloc(sizeof(muacc_context_t));
    memset(ctx, 0, sizeof(muacc_context_t));

    if (muacc_init_context(ctx) != 0)
    {
      ctx = NULL;
    }
  }
  struct address_list *al = lookup_host (ctx, host, 0); //function
     contains getaddrinfo call

  [...]

    sock = connect_to_ip (ctx, ip, port, host); //function contains
       socket and connect calls
    if (sock >= 0)
      {
        /* Success. */
        address_list_set_connected (al);
        address_list_release (al);
        return sock;
      }
```

Listing 2: Modified helper function, from wget/src/connect.c

Furthermore, `NULL` can be supplied as a pointer to the context, either causing the application to allocate a context within the function or causing the Socket Intent Library to fall back to the basic Socket API without any policy decisions. This is useful in cases where the function signature requires a context, but the function is called from portions of the code where policy decisions are not needed, so one does not wish to manage a socket context there. One example is the FTP portion of the wget code which was to be modified as little as possible.

6.1.2. Setting Intents

In order to aid policy decisions by providing hints concerning expected preferences or characteristics of communication, applications may set any of the Intents described in Section 4.2. Applications will be able to specify one or multiple of them, depending on the type of the application and connection.

After determining the Intent and its value that is to be to set on a socket, applications can set it using the `muacc_setsockopt` call of the modified Socket API. They need to provide the context, the socket, the Intents option level `SOL_INTENTS`, the appropriate option name, a pointer to the value, which will be copied in the call, and its length. Definitions for the Intents are found in the header file `lib/intents.h`, which needs to be included.

Two examples, namely category and filesize, are described here in slightly more detail.

Category

As described in Section 4.2, the category Intent places a connection within a broad class of traffic. It is often obvious which category should be chosen for a certain connection, e.g. a video streaming application could use the stream category for all of its sockets. Applications that handle connections more generically could get the category from information available to them, e.g. browsers could derive the category from the content type of an object in its HTTP response. In wget, the category can be explicitly set on the command line by users and is then used for all connections in this run. Once known, the category socket option can be set on the corresponding socket, causing it to be stored in the context and sent to the MAM along with any request as long as long as the context is not released.

Filesize

The filesize Intent provides the size of an object to be fetched in bytes. In order to determine this, applications can for example issue a preliminary request asking for meta information such as the file size without retrieving the object yet. It can then be fetched in a second connection with the filesize Intent set. This approach was chosen for wget, where a partial GET request over the first N bytes of an object is performed, and setting the filesize Intent for a second request to retrieve the rest of the object. The advantage over a HEAD request is that the overhead of establishing the connection twice is avoided for small objects, as they fit within the response to the first request.

If determining the file size up front is not practical, it can also be estimated, e.g. by assuming that software updates for a certain application always have a size that roughly corresponds to the size of the existing application.

6.1.3. Building the Application

Once source code modifications have been made, the application has to be compiled and linked. The Socket Intent Library needs to be present, since the modified application uses it. As a shared library, it should be installed on the system, but it can also be placed in a fixed directory structure relative to the application. It can then be added to the linking process, e.g. within the Makefile.

6.2. Writing a Policy

Policies are plugins for the MAM which are loaded, configured and called to make decisions within the framework. They consist of one or multiple configuration files and a dynamically loaded library implemented in C, the module. The module must provide the interface shown in Listing 3, i.e. implement these methods which are called from the MAM. The first two are invoked when setting up and shutting down the module, and initialize or tear down data structures that the policy needs for operation. The latter are invoked when a request from a client application has been received, and they make the policy decisions, e.g. source or destination address selection. Appendix C.3 includes the full code of a sample policy module, parts of which are discussed in the following sections. They are marked with their line numbers to simplify finding the corresponding parts within the code in the appendix.

```
int init(mam_context_t *mctx);
int cleanup(mam_context_t *mctx);
int on_resolve_request(request_context_t *rctx, struct event_base *base)
   ;
int on_connect_request(request_context_t *rctx, struct event_base *base)
   ;
```

Listing 3: Interface between MAM and policy module, from policies/policy_sample.c

6.2.1. Writing a Configuration File

The configuration file, which is passed to the MAM as a command line argument, instructs it to load the policy module. Note that a policy can have multiple configuration files, so the same policy module can be employed in multiple scenarios, using the same logic but different parameters. The configuration file can provide additional information to the policy as a whole in a key-value manner. Keys start with a letter and may contain letters and numbers, values are strings or numbers. An example is shown in Listing 4.

Furthermore, a configuration file often includes a list of prefixes that are present on the system and that should be used. As explained in Section 5.3.1, the members of the previously auto-detected list of source prefixes are annotated when the configuration file is parsed by the MAM. It is then possible for the policy to easily filter out prefixes that are not marked as configured and enabled. Furthermore, additional information concerning the prefix can be specified in a key-value manner similar to the case of whole-policy information above. When the configuration file is parsed, the per-policy and per-prefix information is placed within dictionaries of key-value pairs within the MAM context and the prefix list. As these dictionary can be queried from the policy, all information from the configuration file is available to the policy.

The example shown in Listing 4 enables a prefix and sets it as default.

```
policy "policy_sample.la" {
    set foo "bar";
};

prefix 192.168.178.34/24 {
    enabled 1;
    set default = 1;
};
```

Listing 4: Sample MAM configuration file, from mamma-sample.conf

6.2.2. Setting Up and Shutting Down the Module

Firstly, the `init` function sets up the data structures that are used within the policy.

Many policies require additional information on a per-prefix basis, e.g. the category Intent for which a given prefix is best suited or a flag that tells whether a given prefix is the default for its address family. For the purpose of storing this information and making it easily accessible to the decision making functions within the policy, a data structure can be defined. Setting up such a dedicated data structure instead of just querying the dictionary of keys and values from the configuration file has some advantages: For one thing, it allows to define reasonable defaults in case that an information has not been set for a certain prefix. Furthermore, it helps to remove all ambiguity from the data that was entered in the configuration file, such as by converting all values to lower case. All such data handling can be implemented at a central point, i.e. in the function that fills the data structures, making the overall code simpler and more readable.

The contents of this structure should correspond to the information given in the configuration file. The example used in the sample policy defines a flag of whether the given prefix is the default or not. It is possible to attach such a data structure to each element of the prefix list through a generic pointer. For this, a function can be invoked on each item of the list, which is explained later in more detail.

Furthermore, many policies make use of a list of prefixes present on the system that have been enabled in the configuration file, which is a subset of the overall prefix list. It is often desired to set up such a list of both IPv4 and IPv6 prefixes, which should be initialized as `NULL` and filled when initializing the module.

An example of an initialization function is shown in Listing 5. There, for each element of the prefix list, a function is called for setting up the per-prefix policy information. Then the list of enabled IPv4 and IPv6 prefixes is populated by calling the utility function available for this task. Note that this does not make a ”deep copy”, so the source prefix data structures from the MAM context are not copied. Instead, new linked lists of pointers to these data structures are created, only containing the desired prefixes.

```
int init(mam_context_t *mctx)
{
    printf("Policy module \"sample\" is loading.\n");

    g_slist_foreach(mctx->prefixes, &set_policy_info, NULL);

    make_v4v6_enabled_lists (mctx->prefixes, &in4_enabled, &in6_enabled)
        ;

    printf("\nPolicy module \"sample\" has been loaded.\n");
    return 0;
}
```

Listing 5: Policy initialization function, from policies/policy_sample.c

The `cleanup` function is the opposite of `init`: It serves to tear down all policy-specific data structures and restore the MAM context's source prefix list to its original state. An example is shown in Listing 6. Note that the freeing functions here do not "deep free"the data structures within the list, so the original source prefix list remains intact.

```
int cleanup(mam_context_t *mctx)
{
    g_slist_free(in4_enabled);
    g_slist_free(in6_enabled);
    g_slist_foreach(mctx->prefixes, &freepolicyinfo, NULL);

    printf("Policy sample library cleaned up.\n");
    return 0;
}
```

Listing 6: Policy cleanup function, from policies/policy_sample.c

The above examples make use of a policy-specific data structure for storing per-prefix information, as well as a number of helper functions.

If a policy uses its own data structures, it needs helper functions to set them up, tear them down and print them. One of these data structures should be attached to each element of the linked list of prefixes. The corresponding functions can be written in such a way that they only handle one element, and later be called on each element of the list.

An example for such a function is shown in Listing 7. It sets up a policy data structure and queries the dictionary to fill it with information. The last line shows how the new data structure is linked from the element in the source prefix list, which can now access it from its `policy_info` member, a generic pointer.

```
void set_policy_info(gpointer elem, gpointer data)
{
    struct src_prefix_list *spl = elem;

    struct sample_info *new = malloc(sizeof(struct sample_info));
    new->is_default = 0;

    if (spl->policy_set_dict != NULL)
    {
        gpointer value = NULL;
        if (((value = g_hash_table_lookup(spl->policy_set_dict, "default
            ")) != NULL) && value )
            new->is_default = 1;
    }
    spl->policy_info = new;
}
```

Listing 7: Policy data structure, from policies/policy_sample.c

After establishing lists of usable prefixes, it is often desired to print them along with their associated policy-specific information. For this purpose, policies are advised to define a `print_policy_info` function that takes a pointer to the opaque policy data structure from the prefix list and prints the policy-specific information. This function serves as a helper to an existing utility function, `print_pfx_addr`, which outputs a list of addresses when given a prefix list. If the prefixes in this list have additional policy information attached, the output is supplemented by `print_policy_info`.

In order to do proper memory management, policies should clean up the data that they have set up when terminating or reloading. This involves releasing the policy-specific data structure for each prefix and removing the pointer to it in order to make space for an eventual new policy that could be loaded later.

6.2.3. Destination Address Selection

The `on_resolve_request` function is invoked every time a request to resolve a host name is received from a client. From there, destination address selection can be performed, which is a little bit less straightforward to implement than source address selection.

If the policy chooses to do destination address selection[4] , it needs to resolve the name by itself, so that it can then reorder the resulting list of destination addresses. However, the MAM is designed for asynchronous communication, as explained in Section 5.3.1, so policies should not simply use the blocking `getaddrinfo` call. This would block other clients that may have sent requests to the MAM at the same time. Instead, policies should also adapt a nonblocking solution. The suggested choice is to use the same technique as the MAM, namely the event-based `libevent2` library. For this, the name resolution request is added to one or more DNS event bases along with the host name and hints for name resolution, as well as a callback to be invoked when the request completes. The event base also specifies which name server to use. An example is shown in Listing 8.

```c
int on_resolve_request(request_context_t *rctx, struct event_base *base)
{
    struct evdns_getaddrinfo_request *req;

    printf("\tResolve request: %s", (rctx->ctx->remote_hostname == NULL
        ? "" : rctx->ctx->remote_hostname));

    /* Try to resolve this request using asynchronous lookup */
    req = evdns_getaddrinfo(
            rctx->mctx->evdns_default_base,
            rctx->ctx->remote_hostname,
            NULL /* no service name given */,
            rctx->ctx->remote_addrinfo_hint,
            &resolve_request_result,
            rctx);
    printf(" - Sending request to default nameserver\n");
```

[4]Policies can avoid the complexity of destination address selection by just sending back the same request context that was received. In this case, they skip name resolution entirely and the client library will resolve the name by itself. In any case, the policy must make sure that a reply is sent back to the client at some point. Else, the client application that has issued the request to the MAM hangs while waiting for an answer indefinitely.

```
    if (req == NULL) {
        /* returned immediately - Send reply to the client */
        _muacc_send_ctx_event(rctx, muacc_act_getaddrinfo_resolve_resp);
      mam_release_request_context(rctx);
        printf("\tRequest failed.\n");
    }
    return 0;
}
```

Listing 8: Resolve request handling, from policies/policy_sample.c

The actual destination address selection is implemented in the callback function, where the list of candidate destination addresses can be sorted and sent back to the client. The example given in Listing 9 simply sends back the received destination address list unmodified.

```
void resolve_request_result(int errcode, struct evutil_addrinfo *addr,
    void *ptr)
{

    request_context_t *rctx = ptr;

    if (errcode) {
        printf("\n\tError resolving: %s -> %s\n", rctx->ctx->
            remote_hostname, evutil_gai_strerror(errcode));
    }
    else
    {
        printf("\n\tGot resolver response for %s: %s\n",
            rctx->ctx->remote_hostname,
            addr->ai_canonname ? addr->ai_canonname : "");

        assert(rctx->ctx->remote_addrinfo_res == NULL);
        rctx->ctx->remote_addrinfo_res = addr;
        print_addrinfo_response (rctx->ctx->remote_addrinfo_res);
    }

    // send reply
    _muacc_send_ctx_event(rctx, muacc_act_getaddrinfo_resolve_resp);
}
```

Listing 9: Resolve request callback, from policies/policy_sample.c

6.2.4. Source Address Selection

Every time a connect request is received from a client, the `on_connect_request` function is called. There it is possible to select the local interface by proposing an address to bind the socket to before connecting. Of course, if the application has already called `bind` itself, its choice of local source address should be respected. Thus, for source address selection, the policy should first check if a source address already exists, and, if this is not the case, find a suitable source address from its prefix lists. Then it inserts the proposed source address into the request context and sends it back to the client.

An example of such a function is shown in Listing 10. Here, the actual decision making, i.e. the selecting of the proposed source address, was moved to another function which is called from there and shown in Listing 11. The debugging output that is normally provided in this function has been omitted in Listing 10, but the complete function is shown in Appendix C.3 from line 189 on.

```c
int on_connect_request(request_context_t *rctx, struct event_base *base)
{
    if(rctx->ctx->bind_sa_req != NULL)
    {   // already bound
        printf("\tAlready bound");
    }
    else
    {
        // search address to bind to
        set_sa_if_default(rctx, sb);
    }

    // send response back
    _muacc_send_ctx_event(rctx, muacc_act_connect_resp);
    mam_release_request_context(rctx);

    return 0;
}
```

Listing 10: Connect request handling (without debug output), from policies/policy_sample.c

For making policy decisions such as selecting a source address, all available information can be taken into account, e.g. from the request context and from the lists of source prefixes with the additional policy-specific information.

The example in Listing 11 chooses a source address based on the available per-prefix information. To do this, it searches for the first prefix configured as default for the given address family. It then suggests this prefix's first address as a source address to bind to, effectively causing the corresponding interface to be used for that connection.

```c
void set_sa_if_default(request_context_t *rctx, strbuf_t sb)
{
    GSList *spl = NULL;
    struct src_prefix_list *cur = NULL;
    struct sample_info *info = NULL;

    if (rctx->ctx->domain == AF_INET)
        spl = in4_enabled;
    else if (rctx->ctx->domain == AF_INET6)
        spl = in6_enabled;

    while (spl != NULL)
    {
        cur = spl->data;
        info = (struct sample_info *)cur->policy_info;
        if (info != NULL && info->is_default)
        {
                /* This prefix is configured as default. Set source address
                   */
                set_bind_sa(rctx, cur, &sb);
                strbuf_printf(&sb, " (default)");
                break;
            }
            spl = spl->next;
        }
}
```

Listing 11: Source address selection, from policies/policy_sample.c

6.2.5. Building the Policy

A policy module is implemented in C as a dynamically loaded library, i.e. written and compiled like a regular library, but linked with the ”module” switch and, currently, a flag to avoid versioning. To simplify this, the existing modules have been built using Libtool's `libltdl` [13].

Libtool aims to avoid symbol conflicts by prefixing exported symbols with `<modulename>` _LTX_. When an application such as MAM loads a symbol from the module, this prefix is cut off to get the symbol's ”real” identifier. Since the prefixed symbol names are long and impractical to type, it is recommended to redefine them at the beginning of the module source code file. Listing 12 is an example from the sample policy module.

```c
#define init policy_sample_LTX_init
#define cleanup policy_sample_LTX_cleanup
#define on_resolve_request policy_sample_LTX_on_resolve_request
#define on_connect_request policy_sample_LTX_on_connect_request
```

Listing 12: Redefinition of symbols, from policies/policy_sample.c

For building and installing the policy, it is recommended to use the setup that is also used for building the MAM, the Socket Intent Library, their components and the existing policy modules. In order to do this, one simply has to add the name of the policy module to be built, its source code files and the name of the corresponding configuration file(s). If utility functions have been used within the policy module, the utility header and source code file must also be provided. An example is shown in Listing 13.

```
pkglib_LTLIBRARIES = policy_sample.la
mammaconfdir=$(sysconfdir)/mamma
mammaconf_DATA = mamma-sample.conf

policy_la_LDFLAGS = -module -avoid-version
policy_sample_la_SOURCES = policies/policy_sample.c policies/policy_util
    .h policies/policy_util.c
policy_sample_la_LDFLAGS = $(policy_la_LDFLAGS)
```

Listing 13: Instructions for building the policy, from Makefile.am

7. Evaluation

In this section, we assess the potential benefits of Socket Intents to applications. To do this, we use an emulated environment with a client that is connected to a server over two different access networks: One interface resembles a relatively low bandwidth DSL line with fast-path enabled (6 Mbits downstream and 768 Kbits upstream bandwidth, 10 ms RTT), the other one resembles an LTE link (12 Mbits downstream and 6 Mbits upstream bandwidth, 70 ms RTT). Over this network, we run HTTP transfers and measure their completion time to evaluate their performance. We conduct three experiments: In the first one, the "Bulk Transfer VS. Query", retrieving a website competes with a large download running in parallel and we employ a policy that distinguishes between them using the category Intent. Secondly, we download files of varying sizes over different interfaces and determine the "Filesize Threshold", the threshold over which the interface with higher bandwidth but higher delay is preferable, as parameter for a policy that uses the filesize Intent. Thirdly, we compare the filesize policy from the second experiment to an application unaware round robin policy that makes use of both interfaces in parallel. Hereafter, the setup and the experiments with their results are described in more detail.

7.1. Setup

An overview of our testbed setup is shown in Figure 7. It consist of three machines: One acts as a client, one as a server and they connect to each other through a third, intermediate machine, the shaper. The client has two interfaces and thus two links to the shaper, `i1` with relatively low delay but also only moderate bandwidth and `i2` with relatively high delay but higher bandwidth. The second machine, the traffic shaper, serves to emulate these characteristics on the interfaces `i1` and `i2`. It also connects the client to the third machine, the server, from which the client can retrieve objects over both of its access links. Constructing use cases for this, we choose HTTP since this is an ubiquitous, stateless protocol for which a lot of software exists.

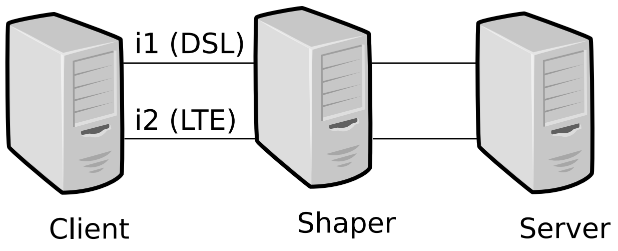


Figure 7: Evaluation setup: Emulated multihomed host

For the different scenarios, we have three main types of workload lying on the server: One huge file to emulate a bulk transfer, mirrors of real web pages to emulate web browsing, and a set of autogenerated files of specific sizes to assess the filesize policy.

Our use case scenarios are emulated in an environment where we have explicit control over all components, the RouterLab. All of the machines (Intel Xeon L5420, 2x4 Cores, 16GB RAM) are interconnected via 1Gbit/s Ethernet links. The traffic between client and server is routed via a machine of the same type on which we run a traffic shaper to simulate the characteristics of different access networks. More specifically, for shaping we use the *TC hierarchical token bucket (HTB)* scheduler and for delay the *TC Network emulator* scheduler.

7.1.1. HTTP Client and Server

We modified one of the "simplest" HTTP clients, namely GNU *wget* version 1.14.4, to enable it to set Socket Intents. More specifically, we added command line options to it, which instruct wget to set the Intents "category" and "filesize".

The category for a run of the application can be explicitly provided on the command line, and any of the values presented in Section 4.2 are possible. In practice they will most likely be "query" for small downloads or "bulk transfer" for large downloads. This information is useful for policies, since the latter are more sensitive to delay while the former are more sensitive to bandwidth availability. For the filesize Intent, wget sets the object size, which we do not necessarily know up front. Therefore, wget was modified to perform its downloads in two phases, taking advantage of the HTTP range query capabilities. More specifically, our modified wget first issues a GET request with a Range header for the first m bytes of each object[5] in order to acquire the size of the object n. It then issues another GET request with a new connection for which it sets the filesize Intent to n to download the remaining $n - m$ Bytes. Of course this two-phase download is only enabled when wget is instructed to use the filesize Intent.

Furthermore, wget was modified to precisely measure the time for resolving names, connecting to a server (i.e. the initial handshake and query) and retrieving the object in milliseconds, which are then logged to a file. Completion time of a request has a major influence on web Quality of Experience (QoE) for end users, which is why precisely measuring it is important to our evaluation.

One unfortunate limitation of wget is that it neither supports HTTP pipelining nor multiple TCP connections in parallel. It simply fetches one objects after another in sequence, thus it cannot use multiple network interfaces at the same time by default. In order to take advantage of two independent network interfaces simultaneously, one has to run multiple wget instances.

As the HTTP server, we choose the widely used Apache2 web server version 2.2.22 and run it in its default configuration, only serving static websites and other objects to clients.

7.1.2. Workloads

With regards to the workload, we use objects of varying sizes. For the bulk transfer, we use a large file of 48 MBytes to emulate the download of a system update. To emulate web browsing, we mirror two web pages: The landing page of a popular newspaper, New York Times (`www.newyorktimes.com`), and a sub-page of a popular photo sharing site, Flickr (`www.flickr.com/explore`) from Jun 10th 2013. Both pages have a size

[5] The m for the initial request should be chosen in order to address the trade-off when a TCP download is dominated by the round trip time and when it is dominated by the network bandwidth. We suggest to use values for m that fit within the initial TCP congestion window.

of 2.8 MBytes. The first one has roughly 130 embedded objects which are relatively small with a range from 48 bytes to 263 KBytes, a median of 6 KBytes, and a mean of 21 KBytes. The second one consist of only 30 embedded objects with size between 43 bytes to 572 KBytes, a median of 65 KBytes, and a mean of 94 KBytes. Both pages were retrieved using Google Chrome's save whole webpage function and copied to the server. Furthermore, for Experiment 2 we create an artificial workload of objects from 10 KBytes to 2 MBytes in steps of 10 KBytes.

7.1.3. Network Emulation

The multihomed client is equipped with two independent Ethernet interfaces on which we simulate different access networks by using different parameters for the traffic shaper. To allow a fair competition, we choose opposite properties for the two interfaces, which are referred to as `i1` and `i2`. The interface `i1` resembles a relatively low bandwidth DSL line with fast-path option enabled. Its parameters are 10ms RTT, 6Mbits downstream bandwidth and 768Kbits upstream bandwidth, as this is the type of DSL line in most parts of Germany [5]. To have an even larger difference compared to `i2`, we also use a slightly modified version of `i1` where useful. The interface `i1'` has only 2 Mbits downstream, which corresponds to a very low bandwidth DSL line as it is available in some of the more rural parts of Germany. Finally, `i2` resembles a reasonable LTE network access, i.e., 70ms RTT, 12Mbits downstream and 6Mbits upstream [10].

7.2. Experiments

Having presented the setup of our emulation in detail in the previous section, here we describe the three illustrative use cases mentioned in the introduction. We constructed and ran them as experiments in our RouterLab. As they are all based on fetching objects from the HTTP server, we measure the completion time in order to compare different policies and how good they perform.

7.2.1. Bulk Transfer vs. Query

In our first scenario we revisit a performance problem often encountered at home: browsing a web site while downloading a large file. The first one is response time critical and we impatiently wait for its completion while the latter, which is not as important to our user experience, is hawking the bandwidth. We simulate this scenario by running two parallel wget instances: (i) wget with Intent "bulk transfer" for the large file (ii) wget with Intent "query" for our two websites. The policy that we use here is that bulk transfers are sent over the higher bandwidth interface `i2` and queries over the low delay interface `i1`. When no policy is configured, the client is restricted to either of the two interfaces.

While doing this experiment we encountered some interesting problems. In cases where only one interface was used for both the bulk transfer and the web queries, the former used up all of the interface's capacity quite aggressively, which led to starvation of the website query until the large transfer finished. We think that the bulk transfer's TCP congestion window scaled up to the point where it clogged all of the bandwidth, and as the small web site queries could not compete at all and there was no other traffic within the network,

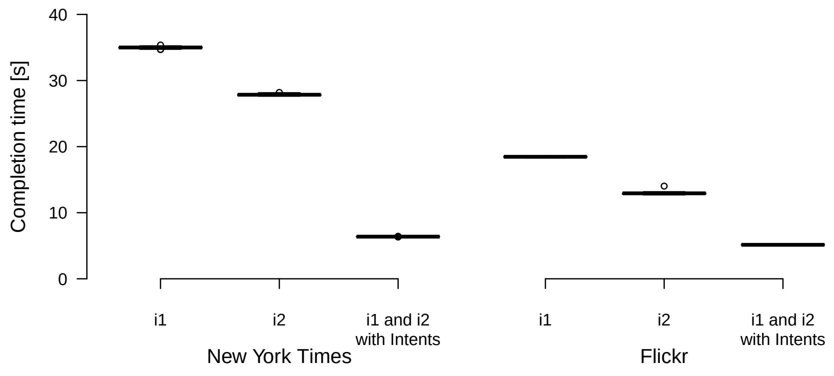


Figure 8: Scenario 1: Boxplot of completion times.

it did not suffer any loss, which would have decreased the window size. We see this as a limitation of our emulated environment, since in a more realistic setting, the bulk transfer would have eventually experienced packet loss and would then have decreased its window size, allowing other transfers to complete. To prevent the bulk transfer from clogging the bandwidth too aggressively, we disabled TCP window scaling for it, limiting the window size of the bulk transfer to a value that allows other transfers to compete. Doing that also prevented the bulk transfer from fully utilizing the Link `i2`, but we decided to stick to this limitation.

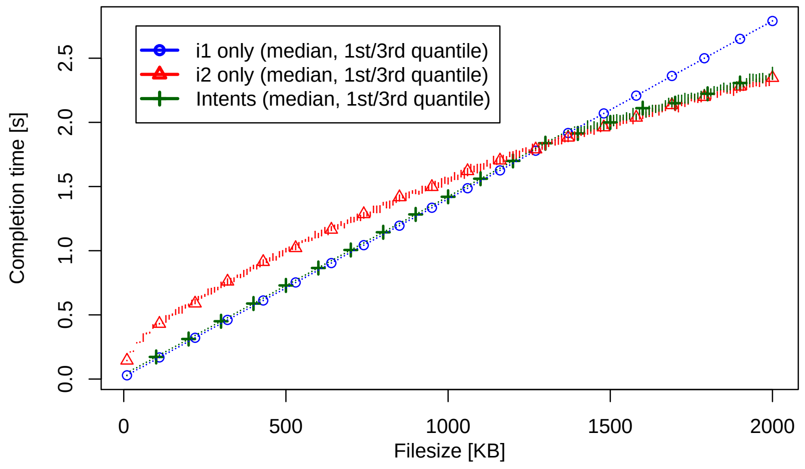
The measured completion times for the web downloads are shown in the boxplot of Figure 8. Recall that a boxplot displays the median, the spread and the skewness of all experiments in one plot. The experiment was repeated 30 times.

Overall, the download time for the Flickr page is smaller than for the New York Times one although the total size is about equal. However, the median object size of Flickr is significantly larger. Moreover, using the high bandwidth interface `i2` rather than the low bandwidth interface is beneficial. Enabling Socket Intent policies improves the page download performance by more than a factor of 60% without penalizing the bulk download. The reason is that the page download can now be scheduled on the network interface `i1` and does no longer compete with the bulk download. We note that this example is in some sense the best case as we can now fully use the second network interface.

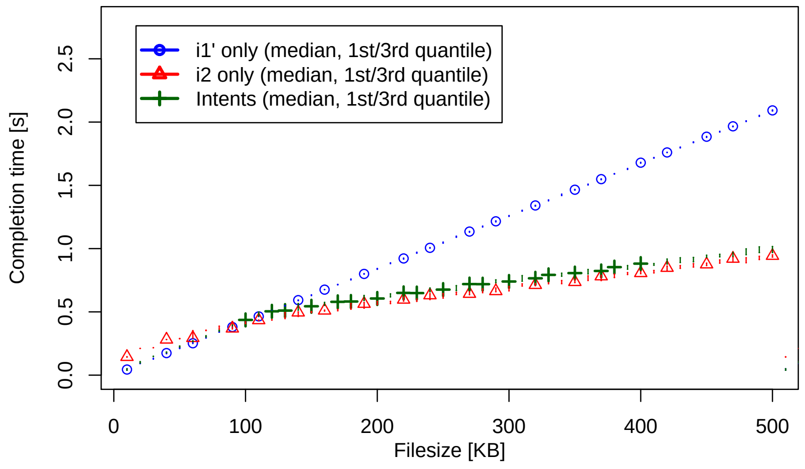
In future work we plan to evaluate how the policy can take advantage of multipath transport protocols such as MPTCP. We predict that using MPTCP will degrade the performance of the web page download while increasing the performance of the bulk download.

7.2.2. Offload by Filesize

We want to utilize the filesize Intent for choosing the "best" interface to download an object of a certain size. The purpose of the second scenario is to find the threshold above which an object should count as "large"for the purpose of our policy, which uses the low delay interface `i1` for small objects and offloads large transfers to the high bandwidth interface `i2`. This means that we determine the value t of the filesize Intent above which connections should be offloaded to the high bandwidth interface.



(a) With `i1` (6Mbit/s) and choosing `i2` for filesizes above 1400 KB



(b) With `i1'` (2Mbit/s) and choosing `i2` for filesizes above 100 KB

Figure 9: Scenario 2: Download times per object size

For this, we compare the use of an interface with low RTT and moderate bitrate against the use of an interface with higher RTT and high bitrate. For the low bandwidth interface, we choose `i1` (6 Mbits/s downstream, 10 ms RTT) in the first run of the experiment and the slight modification `i1'` (2 Mbits/s downstream, 10 ms RTT) in the second run. For the latter, the high bandwidth interface, `i2` (13 Mbits/s downstream, 70 ms RTT) is used. Small transfers should benefit from the low RTT of `i1` or `i1'` while large transfers benefit from the higher bitrate of `i2`. In order to find the threshold filesize, we choose an artificial workload consisting of objects from 10 KBytes to 2 MBytes in steps of 10 KBytes which we download using a single wget instance. Firstly, the completion times of the transfers using only a single interface were determined. The filesize at which the completion time on `i2` is smaller than on `i1` or `i1'` marks the threshold t for the filesize policy. The results of using each interface individually compared to using the resulting Intents enabled filesize policy are shown in Figure 9.

Figure 9 includes the median of the 15 experiment runs as well as the 1st and 3rd quantiles. Note that the experimental variability is small as the quantiles hardly differ from the median. For the second run, the threshold was considerably lower than for the first run, so there we only show the results for files from 10 KBytes up to 500 KBytes and therefore use a different scale on the x axis. We see that in the first run, using the threshold of 1400 KBytes ensures that the policy is always picking the "best" interface and in the second run, this threshold drops to 100 KBytes. In both cases, there is a small overhead of wget retrieving the actual filesize with the initial partial request of 15 KBytes of the two-step download. However, this overhead is minimal compared to the overall completion time.

We were surprised by the fact that the small change in bitrate between `i1` and `i1'` had such a high influence on the threshold for the filesize policy. In particular, the threshold of 1400 KBytes seems quite large to us, while the threshold of 100 KBytes in the second run appears more reasonable. The high threshold shows that the low latency of `i1` gives it an advantage over `i2` for objects up to this size. We note that suitable values for policy parameters, such as the threshold in this example, largely depend on the scenario. They can be heavily influenced by individual interface parameters, the current client location within the network, and the congestion.

7.2.3. Filesize Intent vs. Round-Robin

While the first examples show the advantages of Intents in principle, the comparisons can be considered somewhat unfair, since cases where only one interface is used are contrasted with cases where multiple interfaces are used in parallel. For a more equitable competition, we now compare the use of Intents to an application unaware policy which makes use of both interfaces as well.

More specifically, we examine the performance of the previously used filesize policy next to a policy that uses both interfaces in a round-robin fashion, i.e. alternates between them for the connections. From Experiment 2 we have obtained the filesize thresholds of 1400 KBytes for `i1` and 100 KBytes for `i1'`. Since all objects of the two web sites are smaller than 1400 KBytes, using the filesize policy with this threshold would be equal to an "`i1` only" policy, defeating the purpose of Intents. As only the threshold of 100 KBytes from the second run of Experiment 2 lies within the object size range of our workload, we use the lower bandwidth interface `i1'` with 2 Mbits.

For being able to use multiple interfaces in parallel, we need parallel requests to the server. We choose to download both of the web pages at the same time simulating two users that are using an access point with multiple network interfaces, running two wget instances simultaneously and measuring the completion times of both.

The New York Times site benefits from the application unaware round-robin policy with an improvement in performance by 20% and 18% vs. the interface `i1'` only and the interface `i2` only case while the other does not perform worse (results not shown). The web download times for the round-robin and filesize policies are shown in the boxplot of Figure 10. We point out that downloading the New York Times site benefits significantly from using the filesize Intent. Its download time is improved by 35% or, put differently, using round-robin is 1.5 times slower. For the Flickr site the completion time advantages

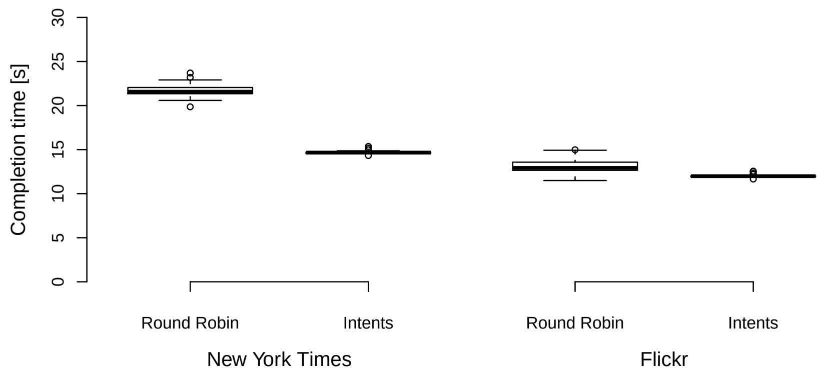


Figure 10: Scenario 3: Boxplot of completion times.

are smaller. This is mainly due to the large number of small objects on the New York Times site, which benefit more from using the low latency interface `il'`.

7.3. Conclusion

All of the shown scenarios only employ simple, intuitive policies, which nevertheless visibly improve the system's service to the application by decreasing completion times. However, the policies can have parameters that greatly depend on network characteristics, such as the threshold of the filesize policy. Thus it is important for policies to be not only be aware of the application's needs, but also of their environment. Moreover, the third experiment highlights that even a simple application unaware policy can improve performance for multi-access devices, while application aware policies can yield even better performance.

8. Conclusion and Future Work

One of the limiting factors for leveraging multiple interfaces to improve service to applications is good information about the applications' needs. The trend with internet capable devices has gone from adding a second network interface to adding the third or even fourth, and several solutions for choosing which network interface to use for which communication exist. However, none of them has become standard practice, and many of them are not aware of the applications' individual characteristics and requirements. We propose to resolve this with *Socket Intents*, a way to express information about the expected characteristics and requirements of a connection, to aid decisions such as interface selection.

In this thesis we describe the principle and propose a set of Intents with information that we think is useful to provide. If necessary to support more use cases, it is simple to define additional Intents without having to alter the principle. We propose a realization of Socket Intents via a modified socket API, exchangeable policies which can be environment-specific, and a policy manager called MAM. These components make up a framework where applications can set Intents for their connections, which can influence policy decisions such as the selection of source and destination address and the tuning of network parameters. The results of these decisions are then applied on the socket of the connection, aiming to improve the system's service to the application. This approach allows for a large variety of use cases with different access networks, applications and policies. The thesis gives practical advice of how applications can be modified to interact with the framework and how policies are written.

In a first feasibility study to assess the potential benefit of applying Intent aware policies, three scenarios are emulated within a testbed. Our results show that making application informed decisions about the interface to use can indeed improve performance for the application, compared to employing an application unaware policy or no policy at all. However, this evaluation has limited applicability to real-world scenarios. A multihomed test machine which runs the Socket Intents prototype exists, but has not become part of this thesis anymore. It would be possible to conduct a larger study of the use of Intents in real-world scenarios using genuine access networks, as opposed to emulated ones.

For further expansion of the framework, we see three tiers: More use cases (possibly involving new topologies), more application scenarios, and more policies.

Within the following two use cases, we think that Intents can be employed with only minor additional effort: Interface selection for mobile data offloading on smartphones and access bundling at home. The latter refers to the enhancement of slow access networks by using cellular mobile communications, which can for example be useful in rural areas. In the this case, the framework is used on a new topology, as the end device may not be directly connected to multiple networks, but only to one small network within the home. Within this network, several routers or other access devices may be present. As multiple prefixes can be configured on a single interface, one can ask the question of how to handle multiple received advertisements from multiple access devices: Is it feasible to configure one prefix for each available access device and distinguish between them on the end host?

Concerning application scenarios, so far our client side is limited to a very simple client, namely wget. We are planning to add Socket Intent support to more applications, such as more complex HTTP clients. Enabling a browser or its plugins to use Intents would make it possible to distinguish between different types of media that are sent over HTTP. For instance, we could handle video streams separately from small queries or bulk downloads, optimizing policy decisions according to each connection's needs. As websites often include multiple requests to the same destination, many clients support HTTP pipelining, i.e. sending multiple requests over a single connection. As this has the potential to greatly improve performance, it is desirable to add support for this to the framework. This may involve modifications to the Socket Intent Library, the MAM, and the policies. Another possible expansion of the Socket Intent Library is to request reestablishing a connection in case of major changes of policy decisions, e.g. due to major changes of network conditions. This could also require support from the application.

In addition to exploring more use cases and application scenarios, in future work we plan to expand our exploration of possible policies. On the one hand, this means having more information available to base a decision on. Applications can set more Intents, but we are also planning to expand the policy framework with advanced network statistics. For this, interface counters can be read, and statistics could be gathered, e.g. RSSI or carrier data rate. This allows for policies that are not only application informed, but that possess cross-layer information. Moreover, we plan to experiment with more possible effects of policy decisions that can be applied on a socket, especially regarding the combination of multiple interfaces on a single host. For this, we intend to include newer protocols with path management capabilities, such as MPTCP and SCTP. Furthermore, we see potential for optimization of network parameters such as choosing the congestion control algorithm for TCP.

Furthermore, the Intents framework would benefit from better integration with the OS. Interaction with the components that manage network interfaces, such as Network Manager on many Linux distributions, would enable the MAM to automatically react to changes of the host's network environment. For instance, the MAM could load a different policy that is more fit for the new scenario, and it could automatically adjust the list of prefixes to use. Information for autoconfiguration that is received on one or multiple interfaces can be incorporated within MAM, which could also try to resolve conflicts among the autoconfiguration from different provisioning domains. Besides MAM integration, it is also conceivable to integrate the Socket Intent Library deeper in libc, where the original Socket API is implemented.

Moving forward we claim that the use of Socket Intents and our framework to assess application aware policies can yield valuable insights regarding the use of multiple interfaces.

We are confident that one can go some way towards universally taking advantage of one's multiple network interfaces and improving service to many different applications by including the proposed Socket Intent API and the MAM or a similar solution in popular OSes.

A. Socket API Functions

Name	**Parameters**	**Value**	**Return Type**
`socket`	family	e.g. `AF_UNIX` (Unix domain socket), `AF_INET` (IPv4), `AF_INET6` (IPv6)	File descriptor of the created socket
	type	e.g. `SOCK_STREAM`, `SOCK_DGRAM`	
	protocol	`0` or protocol number	
`bind`	socket	File descriptor of the socket	`0` if successful
	localaddr	Local socket address	other value if not
	addrlen	Length of socket address	
`connect`	socket	File descriptor of the socket	`0` if successful
	remoteaddr	Remote socket address	other value if not
	addrlen	Length of socket address	
`listen`	socket	File descriptor of the socket	`0` if successful
	queuesize	Maximum number of waiting connection requests	other value if not
`accept`	socket	File descriptor of the socket	File descriptor of the new connected socket
	remoteaddr	Filled with the address of the peer socket	
	addrlen	Length of socket address	
`shutdown`	socket	File descriptor of the socket	`0` if successful
	how	`SHUT_RD` (disable read), `SHUT_WR` (disable write) or `SHUT_RDWR` (disable both)	`-1` if error
`getaddrinfo`	node	Host name or IP address to resolve	`0` if successful
	service	Service name to resolve	error code otherwise
	hints	constraints for the lookup	
	res	buffer where the response will be stored	

Table 4: Socket API calls for setting up, using and tearing down sockets, and resolver library call

Name	Parameters	Value	Return Type
`send`	socket buffer length flags	File descriptor of the socket Data to send Length of the data to send Additional settings for this message	Number of bytes that were sent
`recv`	socket buffer length flags	File descriptor of the socket Buffer where data should be read into Length of the buffer Additional settings for this message	Number of bytes that were read
`sendto`	socket buffer length flags remoteaddr addrlen	File descriptor of the socket Data to send Length of the data to send Additional settings for this message Address of the remote socket Length of socket address	Number of bytes that were sent
`recvfrom`	socket buffer length flags remoteaddr addrlen	File descriptor of the socket Buffer where data should be read into Length of the buffer Additional settings for this message Filled with the address of the remote socket Length of socket address	Number of bytes that were read

Table 5: Socket API calls for sending and receiving

Name	Parameters	Value	Return Type
`setsockopt`	socket level optname optval optlen	File descriptor of the socket Level on which the option applies Identifier of the option Buffer where the value is stored Length of the buffer with the value	`0` if successful `-1` if error
`getsockopt`	socket level optname optval optlen	File descriptor of the socket Level on which the option applies Identifier of the option Buffer where the value is stored Length of the actual returned value	`0` if successful `-1` if error

Table 6: Socket API calls for setting and querying options

B. Definitions of Intents

```
#define SOL_INTENTS 300

#define INTENT_CATEGORY 1   /* Traffic category */
#define INTENT_FILESIZE 2   /* Number of bytes transferred */
#define INTENT_DURATION 3   /* Time of last - first data [seconds] */
#define INTENT_BITRATE 4    /* Size / duration [bytes per second] */
#define INTENT_BURSTINESS 5 /* Burstiness category */
#define INTENT_TIMELINESS 6 /* Timeliness category */
#define INTENT_RESILIENCE 7 /* Resilience category */

/* One of five brief categories into which the traffic may fit */
typedef enum intent_category
{
  INTENT_QUERY,            /* Small, short */
  INTENT_BULKTRANSFER,     /* Large, short, fast, bursty */
  INTENT_CONTROLTRAFFIC,   /* Large, long, slow, bursty */
  INTENT_KEEPALIVES,       /* Small, long, slow, not bursty */
  INTENT_STREAM            /* Large, long, fast, not bursty */
} intent_category_t;

/** Qualitative description of the application traffic's bursts. */
typedef enum intent_burstiness
{
  INTENT_RANDOMBURSTS,
  INTENT_REGULARBURSTS,
  INTENT_NOBURSTS,
  INTENT_BULK              /* Congestion window limited */
} intent_burstiness_t;

/** Desired characteristics regarding delay and jitter. */
typedef enum intent_timeliness
{
  INTENT_STREAMING,         /* Low delay, low jitter */
  INTENT_INTERACTIVE,       /* Low delay, possible jitter */
  INTENT_TRANSFER,          /* Should complete eventually */
  INTENT_BACKGROUNDTRAFFIC /* Only loose time constraint */
} intent_timeliness_t;

/** Impact on application if connection fails and is reestablished */
typedef enum intent_resilience
{
  INTENT_SENSITIVE,         /* Connection loss makes application fail */
  INTENT_TOLERANT,          /* Connection loss tolerable, but
      inconvenient */
  INTENT_RESILIENT          /* Connection loss acceptable */
} intent_resilience_t;
```

Listing 14: Definition of Intents. From lib/intents.h

C. Implementation Details of the Framework

C.1. Socket Context

The socket context has a "public" part which is used by the application and contains parameters for managing the context, shown in Listing 15. The actual information about the socket is stored within the "internal" context, shown in Listing 16, and only this portion is serialized.

```
/* Context of a socket on the client side */
typedef struct muacc_context
{
    int      usage;               /* reference counter */
    uint8_t locks;                /* lock to avoid multiple concurrent
       requests */
    int      mamsock;             /* socket to talk to MAM */
    struct _muacc_ctx *ctx;       /* internal struct with relevant socket
       context data */
} muacc_context_t;
```

Listing 15: Socket context, as presented to applications, from clib/muacc_client.h

```
/* Internal muacc context struct
   All data will be serialized and sent to MAM */
struct _muacc_ctx {
  muacc_ctxid_t       ctxid;             /* identifier for the context */
  unsigned int        calls_performed;  /* flags of performed calls */
  int                 domain;            /* socket domain, e.g. AF_INET */
  int                 type;              /* socket type, e.g. SOCK_STREAM
      */
  int                 protocol;          /* particular protocol */
  struct sockaddr     *bind_sa_req;      /* local address by application
      */
  socklen_t            bind_sa_req_len; /* length of bind_sa_req*/
  struct sockaddr     *bind_sa_suggested;      /* local address suggested
       by MAM */
  socklen_t            bind_sa_suggested_len; /* length of bind_sa_res*/
  char                *remote_hostname;        /* hostname to resolve */
  struct addrinfo     *remote_addrinfo_hint;   /* hints for resolving */
  struct addrinfo     *remote_addrinfo_res;    /* candidate remote
      addresses (sorted by MAM preference) */
  struct sockaddr     *remote_sa;              /* remote address choosen
      in the end */
  socklen_t            remote_sa_len;          /* length of remote_sa_res
       */
  struct socketopt    *sockopts_current;       /* socket options
      currently set */
  struct socketopt    *sockopts_suggested;     /* socket options
      suggested by MAM */
};
```

Listing 16: Socket context: Library internal data, from lib/muacc.h

C.2. MAM Context

The MAM context shown in Listing 17 stores information about the current state of the system and is passed to policies.

```c
/* Context of the MAM */
typedef struct mam_context {
  int                        usage;          /* Reference counter */
  GSList                     *prefixes;      /* Local prefixes on this system */
  lt_dlhandle                policy;         /* Handle of policy module */
  struct event_base          *ev_base;       /* Libevent Event Base */
  struct evdns_base          *evdns_default_base; /* Fallback DNS base */
  GHashTable                 *policy_set_dict; /* From config file */
} mam_context_t;
```

Listing 17: MAM context, from mam/mam.h

C.3. Sample Policy

This policy is intended to be used as a template for implementing new policies. Parts of it are discussed in Section 6.2 in further detail.

```c
#define init policy_sample_LTX_init
#define cleanup policy_sample_LTX_cleanup
#define on_resolve_request policy_sample_LTX_on_resolve_request
#define on_connect_request policy_sample_LTX_on_connect_request

/** Dummy to illustrate how policies work
 *  Policy_info: Whether interface has been specified as default in the config file
 *               (e.g. set default = 1 in the prefix statement)
 *  Behavior:
 *  Getaddrinfo - Resolve names using the default dns_base from the MAM context
 *  Connect     - Choose the default interface if available
 */

#include "policy.h"
#include "policy_util.h"

/** Policy-specific per-prefix data structure that contains additional information */
struct sample_info {
  int is_default;
};

/** List of enabled addresses for each address family */
GSList *in4_enabled = NULL;
GSList *in6_enabled = NULL;

/** Helper to set the policy information for each prefix
 *  Here, check if this prefix has been configured as default
 */
```

```c
void set_policy_info(gpointer elem, gpointer data)
{
  struct src_prefix_list *spl = elem;

  struct sample_info *new = malloc(sizeof(struct sample_info));
  new->is_default = 0;

  if (spl->policy_set_dict != NULL)
  {
    gpointer value = NULL;
    if (((value = g_hash_table_lookup(spl->policy_set_dict, "default"))
       != NULL) && value )
      new->is_default = 1;
  }
  spl->policy_info = new;
}

/** Helper to print additional information given to the policy
 */
void print_policy_info(void *policy_info)
{
  struct sample_info *info = policy_info;
  if (info->is_default)
    printf(" (default)");
}

void freepolicyinfo(gpointer elem, gpointer data)
{
  struct src_prefix_list *spl = elem;

  if (spl->policy_info != NULL)
    free(spl->policy_info);

  spl->policy_info = NULL;
}

/** Helper to set the source address to the default interface,
 *  if any exists for the requested address family
 */
void set_sa_if_default(request_context_t *rctx, strbuf_t sb)
{
  GSList *spl = NULL;
  struct src_prefix_list *cur = NULL;
  struct sample_info *info = NULL;

  if (rctx->ctx->domain == AF_INET)
    spl = in4_enabled;
  else if (rctx->ctx->domain == AF_INET6)
    spl = in6_enabled;

  while (spl != NULL)
  {
    cur = spl->data;
    info = (struct sample_info *)cur->policy_info;
    if (info != NULL && info->is_default)
    {
      /* This prefix is configured as default. Set source address */
      set_bind_sa(rctx, cur, &sb);
      strbuf_printf(&sb, " (default)");
      break;
    }
    spl = spl->next;
  }
}

/** Initializer function (mandatory)
 *  Is called once the policy is loaded and every time it is reloaded
 *  Typically sets the policy_info and initializes the lists of
    candidate addresses
 */
int init(mam_context_t *mctx)
{
  printf("Policy module \"sample\" is loading.\n");

  g_slist_foreach(mctx->prefixes, &set_policy_info, NULL);

  make_v4v6_enabled_lists (mctx->prefixes, &in4_enabled, &in6_enabled);

  printf("\nPolicy module \"sample\" has been loaded.\n");
  return 0;
}

/** Cleanup function (mandatory)
 *  Is called once the policy is torn down, e.g. if MAM is terminates
 *  Tear down lists of candidate addresses (no deep free) and policy
    infos
 */
int cleanup(mam_context_t *mctx)
{
  g_slist_free(in4_enabled);
  g_slist_free(in6_enabled);
  g_slist_foreach(mctx->prefixes, &freepolicyinfo, NULL);

  printf("Policy sample library cleaned up.\n");
  return 0;
}

/** Asynchronous callback function for resolve request
 *  Invoked once a response to the resolver query has been received
 *  Sends back a reply to the client with the received answer
 */
void resolve_request_result(int errcode, struct evutil_addrinfo *addr,
    void *ptr)
{

  request_context_t *rctx = ptr;

  if (errcode) {
      printf("\n\tError resolving: %s -> %s\n", rctx->ctx->
          remote_hostname, evutil_gai_strerror(errcode));
  }
  else
  {
    printf("\n\tGot resolver response for %s: %s\n",
      rctx->ctx->remote_hostname,
      addr->ai_canonname ? addr->ai_canonname : "");

    assert(rctx->ctx->remote_addrinfo_res == NULL);
    rctx->ctx->remote_addrinfo_res = addr;
    print_addrinfo_response (rctx->ctx->remote_addrinfo_res);
  }

  // send reply
  _muacc_send_ctx_event(rctx, muacc_act_getaddrinfo_resolve_resp);

  // hack - free addr first the evutil way
    if(addr != NULL) evutil_freeaddrinfo(addr);
  rctx->ctx->remote_addrinfo_res = NULL;
  // then let mam clean up the remainings
    mam_release_request_context(rctx);

}

/** Resolve request function (mandatory)
 *  Is called upon each getaddrinfo request from a client
 *  Must send a reply back using _muacc_sent_ctx_event or register a
     callback that does so
 */
int on_resolve_request(request_context_t *rctx, struct event_base *base)
{
    struct evdns_getaddrinfo_request *req;

  printf("\tResolve request: %s", (rctx->ctx->remote_hostname == NULL ?
      "" : rctx->ctx->remote_hostname));

  /* Try to resolve this request using asynchronous lookup */
    req = evdns_getaddrinfo(
        rctx->mctx->evdns_default_base,
      rctx->ctx->remote_hostname,
      NULL /* no service name given */,
            rctx->ctx->remote_addrinfo_hint,
      &resolve_request_result,
      rctx);
  printf(" - Sending request to default nameserver\n");
    if (req == NULL) {
    /* returned immediately - Send reply to the client */
    _muacc_send_ctx_event(rctx, muacc_act_getaddrinfo_resolve_resp);
    mam_release_request_context(rctx);
    printf("\tRequest failed.\n");
  }
  return 0;
}
```

```c

/** Connect request function (mandatory)
 *  Is called upon each connect request from a client
 *  Must send a reply back using _muacc_sent_ctx_event or register a
     callback that does so
 */
int on_connect_request(request_context_t *rctx, struct event_base *base)
{
  strbuf_t sb;
  strbuf_init(&sb);
  strbuf_printf(&sb, "\tConnect request: dest=");
  _muacc_print_sockaddr(&sb, rctx->ctx->remote_sa, rctx->ctx->
      remote_sa_len);

  if(rctx->ctx->bind_sa_req != NULL)
  { // already bound
    strbuf_printf(&sb, "\tAlready bound to src=");
    _muacc_print_sockaddr(&sb, rctx->ctx->bind_sa_req, rctx->ctx->
        bind_sa_req_len);
  }
  else
  {
    // search address to bind to
    set_sa_if_default(rctx, sb);
  }

  // send response back
  _muacc_send_ctx_event(rctx, muacc_act_connect_resp);
  mam_release_request_context(rctx);
    printf("%s\n\n", strbuf_export(&sb));
    strbuf_release(&sb);

  return 0;
}
```

Listing 18: Sample policy module, from policies/policy_sample.c

References

[1] Ieee std. 1003.1-2001 standard for information technology – portable operating system interface (posix). Open Group Technical Standard: Base Specifications, Issue 6, 2001.

[2] H. Abbasi, C. Poellabauer, K. Schwan, G. Losik, and Richard. A quality-of-service enhanced socket api in gnu/linux. In *In the 4th Real-Time Linux Workshop*, 2002.

[3] Aruna Balasubramanian, Ratul Mahajan, and Arun Venkataramani. Augmenting mobile 3g using wifi. In *ACM MobiSys*, 2010.

[4] M. Blanchet and P. Seite. Multiple Interfaces and Provisioning Domains Problem Statement. RFC 6418 (Informational), Nov 2011.

[5] Bundesministerium für Wirtschaft und Technologie. Breitbandatlas. Website `http://www.zukunft-breitband.de/DE/breitbandatlas` accessed on 2013-06-12, 2013.

[6] Cisco Systems, Inc. Architecture for mobile data offload over wi-fi access networks (whitepaper). `http://www.cisco.com/en/US/solutions/collateral/ns341/ns524/ns673/white_paper_c11-701018.html`, 2012.

[7] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural Guidelines for Multipath TCP Development. RFC 6182 (Informational), Mar 2011.

[8] R. Gilligan, S. Thomson, J. Bound, J. McCann, and W. Stevens. Basic Socket Interface Extensions for IPv6. RFC 3493 (Informational), Feb 2003.

[9] B. D. Higgins, A. Reda, T. Alperovich, J. Flinn, T. J. Giuli, B. Noble, and D. Watson. Intentional networking: opportunistic exploitation of mobile network diversity. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking*, MobiCom '10, pages 73–84. ACM, 2010.

[10] Junxian Huang, Feng Qian, Alexandre Gerber, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *ACM MobiSys*, 2012.

[11] K. Lan and J. Heidemann. A measurement study of correlations of internet flow characteristics. *Computer Networks*, 50(1):46–62, 2006.

[12] Kyunghan Lee, Joohyun Lee, Yung Yi, Injong Rhee, and Song Chong. Mobile data offloading: how much can wifi deliver? In *ACM CONEXT*, 2010.

[13] GNU Libtool. Libtool dynamic linking (ltdl). `http://www.gnu.org/software/libtool/manual/libtool.html`, 2013.

[14] T. Narten, R. Draves, and S. Krishnan. Privacy Extensions for Stateless Address Autoconfiguration in IPv6. RFC 4941 (Draft Standard), Sep 2007.

[15] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474 (Proposed Standard), Dec 1998. Updated by RFCs 3168, 3260.

[16] E. Nordmark, S. Chakrabarti, and J. Laganier. IPv6 Socket API for Source Address Selection. RFC 5014 (Informational), Sep 2007.

[17] M. Scharf and A. Ford. Multipath TCP (MPTCP) Application Interface Considerations. RFC 6897 (Informational), Mar 2013.

[18] A. A. Siddiqui and P. Müller. A requirement-based socket api for a transition to future internet architectures. In *IMIS*, pages 340–345, 2012.

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

[20] W. Richard Stevens and Gary R. Wright. *TCP/IP illustrated (vol. 2): the implementation.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[21] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), Sep 2007. Updated by RFCs 6096, 6335.

[22] R. Stewart, M. Tuexen, K. Poon, P. Lei, and V. Yasevich. Sockets API Extensions for the Stream Control Transmission Protocol (SCTP). RFC 6458 (Informational), Dec 2011.

[23] D. Thaler, R. Draves, A. Matsumoto, and T. Chown. Default Address Selection for Internet Protocol Version 6 (IPv6). RFC 6724 (Proposed Standard), Sep 2012.

[24] M. Wasserman and P. Seite. Current Practices for Multiple-Interface Hosts. RFC 6419 (Informational), Nov 2011.

[25] M. Welzl, S. Jorer, and S. Gjessing. Towards a protocol-independent internet transport api. In *Communications Workshops (ICC), 2011 IEEE International Conference on*, pages 1 –6, june 2011.

[26] J. Wroclawski. The Use of RSVP with IETF Integrated Services. RFC 2210 (Proposed Standard), Sep 1997.