Sockets and Beyond: Assessing the Source Code of Network Applications

Miika Komu
Aalto University, Department of Computer Science and Engineering
miika@iki.fi

Samu Varjonen, Andrei Gurtov, Sasu Tarkoma
University of Helsinki and Helsinki Institute for Information Technology
firstname.lastname@hiit.fi

Abstract

Network applications are typically developed with frameworks that hide the details of low-level networking. The motivation is to allow developers to focus on application-specific logic rather than low-level mechanics of networking, such as name resolution, reliability, asynchronous processing and quality of service. In this article, we characterize statistically how open-source applications use the Sockets API and identify a number of requirements for network applications based on our analysis. The analysis considers five fundamental questions: naming with end-host identifiers, name resolution, multiple end-host identifiers, multiple transport protocols and security. We discuss the significance of these findings for network application frameworks and their development. As two of our key contributions, we present generic solutions for a problem with OpenSSL initialization in C-based applications and a multihoming issue with UDP in all of the analyzed four frameworks.

1 Introduction

The Sockets API is the basis for all internet applications. While the number of applications using it directly is large, some applications use it indirectly through intermediate libraries or frameworks to hide the intricacies of the low-level Sockets API. Nevertheless, the intermediaries still have to interface with the Sockets API. Thus, the Sockets API is important for all network applications either directly or indirectly but has been studied little. To fill in this gap, we have statistically analyzed the usage of Sockets API to characterize how contemporary network applications behave in Ubuntu Linux. In addition to merely characterizing the trends, we have also investigated certain programming pitfalls pertaining the Sockets API.

As a result, we report ten main findings and how they impact a number of relatively new sockets API extensions. To mention few examples, the poor adoption of a new DNS look up function slows down the migration path for the extensions dependent on it, such as the APIs for IPv6 source address selection and HIP. OpenSSL library is initialized incorrectly in many applications, causing potential security vulnerabilities. The management of the dual use of TCP/UDP transports and the dual use of the two IP address families creates redundant complexity in applications.

To escape the unnecessary complexity of the Sockets API, some applications utilize network application frameworks. However, the frameworks are themselves based on the Sockets API and, therefore, subject to the same scrutiny as applications using the Sockets API. For this reason, it is natural to extend the analysis for frameworks.

We chose four example frameworks based on the Sockets API and analyzed them manually in the light of the Sockets API findings. Since frameworks can offer high-level abstractions that do not have to mimic the Sockets API layout, we organized the analysis of the frameworks in a top-down fashion and along generalized dimensions of end-host naming, multiplicity of names and transports, name look up and security. As a highlight of the framework analysis, we discovered a persistent problem with multiplicity of names in all of the four frameworks. To be more precise, the problem was related to multihoming with UDP.

In this article, we describe how to solve some of the discovered issues in applications and frameworks using the Sockets API. We also characterize some of the inherent limitations of the Sockets API, for instance, related to complexity.

2 Background

In this section, we first introduce the parts of the Berkeley Sockets and the POSIX APIs that are required to understand the results described in this article. Then, we briefly introduce four network application frameworks built on top of the two APIs.

2.1 The Sockets API

The Sockets API is the de-facto API for network programming due to its availability for various operating systems and languages. As the API is rather low level and does not support object-oriented languages well, many networking libraries and frameworks offer additional higher-level abstractions to hide the details of the Sockets API.

Unix-based systems typically provide an abstraction of all network, storage and other devices to the applications. The abstraction is realized with *descriptors* which are also sometimes called *handles*. The descriptors are either file or socket descriptors. Both of them have different, specialized accessor functions even though socket descriptors can be operated with some of the file-oriented functions.

When a socket descriptor is created with the `socket()` function, the transport protocol has to be fixed for the socket. In practice, `SOCK_STREAM` constant fixes the transport protocol to TCP and `SOCK_DGRAM` constant to UDP. For IPv4-based communications, an application uses a constant called `AF_INET`, or its alias `PF_INET`, to create an IPv4-based socket. For IPv6, the application uses correspondingly `AF_INET6` or `PF_INET6`.

2.1.1 Name Resolution

An application can look up names from DNS by calling `gethostbyname()` or `gethostbyaddr()` functions. The former looks up the host information from the DNS by its symbolic name (forward look up) and the latter by its numeric name, i.e., IP address (reverse look up). While both of these functions support IPv6, they are obsolete and their modern replacements are the `getnameinfo()` and `getaddrinfo()` functions.

2.1.2 Delivery of Application Data

A client-side application can start sending data immediately after creation of the socket; however, the application typically calls the `connect()` function to associate the socket with a certain destination address and port. The `connect()` call also triggers the TCP handshake for sockets of `SOCK_STREAM` type. Then, the networking stack automatically associates a source address and port with the socket if the application did not choose them explicitly with the `bind()` function. Finally, a `close()` call terminates the socket gracefully and, when the type of the socket is `SOCK_STREAM`, the call also initiates the shutdown procedure for TCP.

Before a server-oriented application can receive incoming datagrams, it has to call a few functions. Minimally with UDP, the application has to define the port number and IP address to listen to by using `bind()`. Typically, TCP-based services supporting multiple simultaneous clients prepare the socket with a call to the `listen()` function for the following `accept()` call. By default, the `accept()` call blocks the application until a TCP connection arrives. The function then “peels off” a new socket descriptor from existing one that separates the particular connection with the client from others.

A constant `INADDR_ANY` is used with `bind()` to listen for incoming datagrams on all network interfaces and addresses of the local host. This wildcard address is typically employed in server-side applications.

An application can deliver and retrieve data from the transport layer in multiple alternative ways. For instance, the `write()` and `read()` functions are file-oriented functions but can also be used with socket descriptors to send and receive data. For these two file-oriented functions, the Sockets API defines its own specialized functions.

For datagram-oriented networking with UDP, the `sendto()` and the `recvfrom()` functions can be used. Complementary functions `sendmsg()` and `recvmsg()` offer more advanced interfaces for applications [19]. They operate on scatter arrays (multiple non-consecutive I/O buffers instead of just one) and also support so-called ancillary data that refers to meta-data and information related to network packet headers.

In addition to providing the rudimentary service of sending and receiving application data, the socket calls also implement access control. The `bind()` and `connect()` limit ingress (but not egress) network access to the socket by setting the allowed local and remote destination end point. Similarly, the `accept()` call effectively constrains remote access to the newly created socket by allowing communications only with the particular client. Functions `send()` and `recv()` are typically used for connection-oriented networking, but can also be used with UDP to limit remote access.

2.1.3 Customizing Networking Stack

The Sockets API provides certain default settings for applications to interact with the transport layer. The settings can be altered in multiple different ways.

With "raw" sockets, a process can basically create its own transport-layer protocol or modify the network-level headers. A privileged process creates a raw socket with constant `SOCK_RAW`.

A more constrained way to alter the default behavior of the networking stack is to set socket options with `setsockopt()`. As an example of the options, the `SO_REUSEADDR` socket option can be used to disable the default "grace period" of a locally reserved transport-layer port. By default, consecutive calls to `bind()` with the same port fail until the grace period has passed. Especially during the development of a networking service, this grace period is usually disabled for convenience because the developed service may have to be restarted quite often for testing purposes.

2.2 Sockets API Extensions

Basic Socket Interface Extensions for IPv6 [5] define additional data structures and constants, including `AF_INET` and `sockaddr_in6`. The extensions also define new DNS resolver functions, `getnameinfo()` and `getaddrinfo()`, as the old ones, `gethostbyname()` and `gethostbyaddr()`, are now obsoleted. The older ones are not thread safe and offer too little control over the resolved addresses. The specification also defines IPv6-mapped IPv4 addresses to improve IPv6 interoperability.

An IPv6 application can typically face a choice of multiple source and destination IPv6 pairs to choose from. Picking a pair may not be a simple task because some of the pairs may not even result in a working connectivity. IPv6 Socket API for Source Address Selection [13] defines extensions that restrict the local or remote address to a certain type, for instance, public or temporary IPv6 addresses. The extensions include new socket options to restrict the selection local addresses when, e.g., a client application connects without specifying the source address. For remote address selection, new flags for the `getaddrinfo()` resolver are proposed. The extensions mainly affect client-side connectivity but can affect also at the server side when UDP is being used.

The Datagram Congestion Control Protocol (DCCP) is similar to TCP but does not guarantee in-order delivery. An application can use it - with minor changes - by using `SOCK_DCCP` constant when a socket is created.

Multihoming is becoming interesting because most of the modern handhelds are equipped with, e.g., 3G and WLAN interfaces. In the scope of this work, we associate "multihoming" to hosts with multiple IP addresses typically introduced by multiple network interfaces. Multihoming could be further be further characterized whether it occurs in the initial phases of the connectivity or during established communications. All of the statistics in this article refer to the former case because the latter requires typically some extra logic in the application or additional support from the lower layers.

When written correctly, UDP-based applications can support multihoming for initial connectivity and the success of this capability is investigated in detail in this article. However, supporting multihoming in TCP-based applications is more difficult to achieve and requires additional extensions. A solution at the application layer is to recreate connections when they are rendered broken. At the transport layer, Multipath TCP [4] is a TCP-specific solution to support multihoming in a way that is compatible with legacy applications with optional APIs for native applications [16].

The Stream Control Transmission Protocol (SCTP, [21]) implements an entirely new transport protocol with full multihoming capabilities. In a nutshell, SCTP offers a reliable, congestion-aware, message-oriented, in-sequence transport protocol. The minimum requirement to enable SCTP in an existing application is to change the protocol type in `socket()` call to SCTP. However, the application can only fully harness the benefits of the protocol by utilizing the `sendmsg()` and `recvmsg()` interface. Also, the protocol supports sharing of a single socket descriptor for multiple simultaneous communication partners; this requires some additional logic in the application.

Transport-independent solutions operating at the lower layers include Host Identity Protocol [11] and Site Multihoming by IPv6 Intermediation (SHIM6) [12]. In brief, HIP offers support for end-host mobility, multihoming and NAT traversal. By contrast, SHIM6 is mainly a multihoming solution. From the API perspective, SHIM6 offers backwards compatible identifiers for IPv6—in the sense that they are routable at the network layer—whereas the identifiers in HIP are non-routable. HIP has its own optional APIs for HIP-aware applications [9] but both protocols share the same optional multihoming APIs [8].

Name-based Sockets are a work-in-progress at the IETF standardization forum. While the details of the specification [23] are rather immature and the specification still lacks official consent of the IETF, the main idea is to provide extensions to the Sockets API that replace IP addresses with DNS-based names. In this way, the responsibility for the management of IP addresses is pushed down in the stack, away from the application layer.

2.3 NAT Traversal

Private address realms [18] were essentially introduced by NATs, but Virtual Private Networks (VPNs) and other tunneling solutions can also make use of private addresses. Originally, the concept of virtual address spaces was created to alleviate the depletion of the IPv4 address space, perhaps, because it appeared that most client hosts did not need publicly-reachable addresses. Consequently, NATs also offer some security as a side effect to the client side because they discard new incoming data flows by default.

To work around NATs, Teredo [7] offers NAT traversal solution based on a transparent tunnel to the applications. The protocol tries to penetrate through NAT boxes to establish a direct end-to-end tunnel but can resort to triangular routing through a proxy in the case of an unsuccessful penetration.

2.4 Transport Layer Security

Transport Layer Security (TLS) [22] is a cryptographic protocol that can be used to protect communications above the transport layer. TLS, and its predecessor Secure Socket Layer (SSL), are the most common way to protect TCP-based communications over the Internet.

In order to use SSL or TLS, a C/C++ application is usually linked to a library implementation such as OpenSSL or GNU TLS. The application then calls the APIs of the TLS/SSL-library instead of using the APIs of the Sockets API. The functions of the library are wrappers around the Sockets API, and are responsible for securing the data inside the TCP stream.

2.5 Network Frameworks

The Sockets API could be characterized as somewhat complicated and error-prone to be programmed directly. It is also “flat” by its nature because it was not designed to accommodate object-oriented languages. For these reasons, a number of libraries and frameworks have been built to hide the details of the Sockets API and to introduce object-oriented interfaces. The Adaptive Communication (ACE) [17] is one such framework.

ACE simplifies the development of networking applications because it offers abstracted APIs based on network software patterns observed in well-written software. Among other things, ACE includes network patterns related to connection establishment and service initialization in addition to facilitating concurrent software and distributed communication services. It supports asynchronous communications by inversion of control, i.e., the framework takes over the control of the program flow and it invokes registered functions of the application when needed.

Boost::Asio is another open source C++ library that offers high-level networking APIs to simplify development of networking applications. Boost::Asio aims to be portable, scalable, and efficient but, most of all, it provides a starting point for implementing further abstraction. Several Boost C++ libraries have already been included in the C++ Technical Report 1 and in C++11. In 2006 a networking proposal based on Asio was submitted to request inclusion in the upcoming Technical Report 2.

Java provides an object-oriented framework for the creation and use of sockets. *Java.net* package (called Java.net from here on) supports TCP (`Socket` class) and UDP (`Datagram` class). These classes implement communication over an IP network.

Twisted is a modular, high-level networking framework for python. Similarly to ACE, Twisted is also based on inversion of control and asynchronous messaging. Twisted has built-in support for multiple application-layer protocols, including IRC, SSH and HTTP. What distinguishes Twisted from the other frameworks is the focus on service-level functionality based adaptable functionality that can be run on top of several application-layer protocols.

3 Materials and Methods

We collected information related to the use of Sockets API usage in open-source applications. In this article, we refer to this information as *indicators*. An indicator refers to a constant, structure or function of the C language. We analyzed the source code for indicators in a static way (based on keywords) rather than dynamically.[1] The collected set of indicators was limited to networking-related keywords obtained from the keyword indexes of two books [20, 15].

We gathered the material for our analysis from all of the released Long-Term Support (LTS) releases of Ubuntu: Dapper Drake 6.06, Hardy Heron 8.04, Lucid Lynx 10.04. Table 1 summarizes the number of software packages gathered per release. In the table, "patched" row expresses how many applications were patched by Ubuntu.

We used sections "main", "multiverse", "universe" and "security" from Ubuntu. The material was gathered on Monday 7th of March 2011 and was constrained to software written using the C language. Since our study was confined to networking applications, we selected only software in the categories of "net", "news", "comm", "mail", and "web" (in Lucid, the last category was renamed "httpd").

	Dapper	Hardy	Lucid
Total	1,355	1,472	1,147
Patched	1,222	1,360	979
C	**721**	**756**	**710**
C++	57	77	88
Python	126	148	98
Ruby	19	27	13
Java	9	10	8
Other	423	454	232

Table 1: Number of packages per release version.

We did not limit or favor the set of applications, e.g., based on any popularity metrics. We believed that an application was of at least of some interest if the application was being maintained by someone in Ubuntu. To be more useful for the community, we analyzed all network applications and did not discriminate some "unpopular" minorities. This way, we did not have to choose between different definitions of popularity—perhaps Ubuntu popularity contest would have served as a decent metric for popularity. We did perform an outlier analysis in which we compared the whole set of applications to the most popular applications (100 or more installations). We discovered that the statistical "footprint" of the popular applications is different from the whole. However, the details are omitted because this contradicted with our goals.

In our study, we concentrated on the POSIX networking APIs and Berkeley Sockets API because they form the de-facto, low-level API for all networking applications. However, we extended the API analysis to OpenSSL to study the use of security as well. All of these three APIs have bindings for high-level languages, such as Java and Python, and can be indirectly used from network application frameworks and libraries. As the API bindings used in other languages differs from those used in C language, we excluded other languages from this study.

From the data gathered,[2] we calculated sums and means of the occurrences of each indicator. Then we also calculated a separate "reference" number. This latter was formed by introducing a binary value to denote whether a software package used a particular indicator (1) or not (0), independent of the number of occurrences. The reference number for a specific indicator was collected from all software packages, and these reference numbers were then summed and divided by the number of packages to obtain a *reference ratio*. In other words, the reference ratio describes the extent of an API indicator

[1] Authors believe that a more dynamic or structural analysis would not have revealed any important information on the issues investigated

[2] `http://www.cs.helsinki.fi/u/sklvarjo/LS12/`

with one normalized score.

We admit that the reference number is a very coarse grained metric; it indicates capability rather than 100% guarantee that the application will use a specific indicator for all its runs. However, its binary (or "flattened") nature has one particular benefit that cancels out an unwanted side effect of the static code analysis, but this is perhaps easiest to describe by example. Let us consider an application where memory allocations and deallocations can be implemented in various ways. The application can call `malloc()` a hundred times but then calls `free()` only once. Merely looking at the volumes of calls would give a wrong impression about memory leaks because the application could have a wrapper function for `free()` that is called a hundred times. In contrast, a reference number of 1 for `malloc()` and 0 for `free()` indicates that the application has definitely one or more memory leak. Correspondingly, the reference ratio describes this for the entire population of the applications.

In our results, we show also reference ratios of combined indicators that were calculated by taking an union or intersection of indicators, depending on the use case. With combined indicators, we used tightly coupled indicators that make sense in the context of each other.

4 Results and Analysis

In this section, we show the most relevant statistical results. We focus on the findings where there is room for improvement or that are relevant to the presented Sockets API extensions. Then, we highlight the most significant patterns or key improvements for the networking applications. Finally, we derive a set of more generic requirements from the key improvements and see how they are met in four different network application frameworks.

4.1 Core Sockets API

In this section, we characterize how applications use the "core" Sockets API. Similarly as in the background, the topics are organized into sections on IPv6, DNS, transport protocols and customization of the networking stack. In the last section, we describe a multihoming issue related to UDP.

In the results, the reference ratios of indicators are usually shown inside brackets. All numeric values are from Ubuntu Lucid unless otherwise mentioned. Figure 1 illustrates some of the most frequent function indicators by their reference ratio and the following sections analyze the most interesting cases in more detail.

4.1.1 IPv6

According to the usage of AF and PF constants, 39.3% were IPv4-only applications, 0.3% IPv6-only, 26.9% hybrid and 33.5% did not reference either of the constants. To recap, while the absolute use of IPv6 was not high, the relative proportion of hybrid applications supporting both protocols was quite high.

4.1.2 Name Resolution

The obsolete DNS name-look-up functions were referenced more than their modern replacements. The obsolete forward look-up function `gethostbyname()` was referenced roughly twice as often as its modern replacement `getaddrinfo()`. Two possible explanations for this are that either that the developers have, for some reason, preferred the obsolete functions, or have neglected to modernize their software.

4.1.3 Packet Transport

Connection and datagram-oriented APIs were roughly as popular. Based on the usage of `SOCK_STREAM` and `SOCK_DGRAM` constants, we accounted for 25.1% TCP-only and 11.0% UDP-only applications. Hybrid applications supporting both protocols accounted for 26.3%, leaving 37.6% of the applications that used neither of the constants. By combining the hybrids with TCP-only applications, the proportion of applications supporting TCP is 51.4% and, correspondingly, 37.3% for UDP. It should not be forgotten that typically all network applications implicitly access DNS over UDP by default.

4.1.4 Customizing Networking Stack

While the Sockets API provides transport-layer abstractions with certain system-level defaults, many applications preferred to customize the networking stack or to

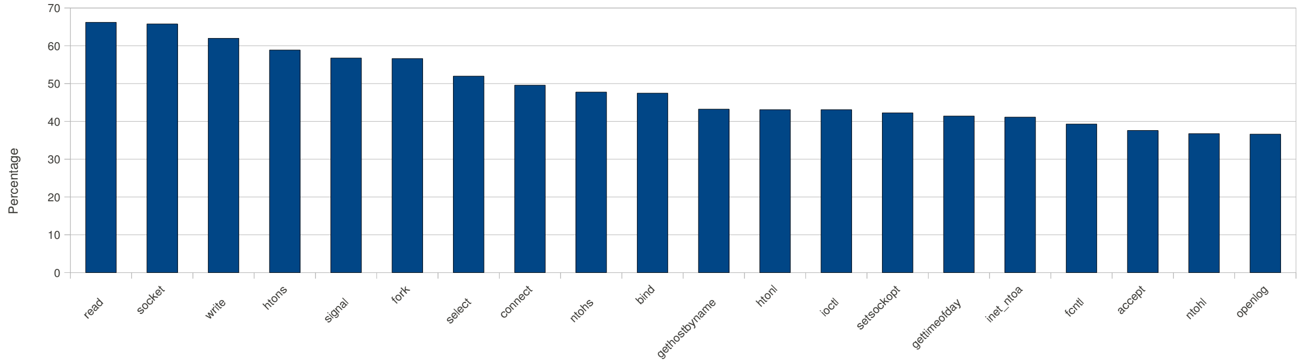

Figure 1: The most frequent functions in Ubuntu Lucid

override some of the parameters. The combined reference ratio of SOCK_RAW, setsockopt(), pcap_pkthdr and ipq_create_handle() indicators was 51.4%. In other words, the default abstraction or settings of the Sockets API are not sufficient for the majority of the applications.

It is worth mentioning that we conducted a brute-force search to find frequently occurring socket options sets. As a result, we did not find any recurring sets but merely individual socket options that were popular.

4.1.5 Multihoming and UDP

In this section, we discuss a practical issue related to UDP-based multihoming, but one which could be fixed in most applications by the correct use of SO_BINDTODEVICE (2.3%) socket option. The issue affects UDP-based applications accepting incoming connections from multiple interfaces or addresses.

On Linux, we have reason to believe that many UDP-based applications may not handle multihoming properly for initial connections. The multihoming problem for UDP manifests itself only when a client-side application uses a server address that does not match with the default route at the server. The root of the problem lies in egress datagram processing at the server side.

The UDP problem occurs when the client sends a "request" message to the server and the server does not send a "response" using the exact same address pair that was used for the request. Instead, the sloppy server implementation responds to the client without specifying the source address, and the networking stack invariably chooses always the wrong source address - meaning that the client drops the response as it appears to be arriving from a previously unknown IP address.

A straightforward fix is to modify the server-side processing of the software to respect the original IP address, and thus to prevent the network stack from routing the packet incorrectly. In other words, when the server-side application receives a request, it should remember the local address of the received datagram and use it explicitly for sending the response.

Explicit source addressing can be realized by using the modern sendmsg() interface. However, a poorly documented alternative to be used especially with the sendto() function is the socket option called SO_BINDTODEVICE. The socket option is necessary because bind() can only be used to specify the local address for the ingress direction (and not the egress).

We discovered the UDP problem by accident with iperf, nc and nc6 software. We have offered fixes to maintainers of these three pieces of software. Nevertheless, the impact of the problem may be larger as a third of the software in our statistics supports UDP explicitly. To be more precise, the lack of SO_BINDTODEVICE usage affects 45.7% (as an upper bound) of the UDP-capable software, which accounts for a total of 121 applications. This figure was calculated by finding the intersection of all applications not using sendmsg() and SO_BINDTODEVICE, albeit still using sendto() and SOCK_DGRAM. We then divided this by the number of applications using SOCK_DGRAM.

4.2 Sockets API Extensions

In this section, we show and analyze statistics on SSL and the adoption of a number of Sockets API extensions.

4.2.1 Security: SSL/TLS Extensions

Roughly 10.9% of the software in the data set used OpenSSL and 2.1% GNU TLS. In this section, we limit the analysis on OpenSSL because it is more popular. Unless separately mentioned, we will, for convenience, use the term SSL to refer both TLS and SSL protocols. We only present reference ratios relative to the applications using OpenSSL because this is more meaningful from the viewpoint of the analysis. In other words, the percentages account only the 77 OpenSSL-capable applications and not the whole set of applications.

The applications using OpenSSL consisted of both client and server software. The majority of the applications using OpenSSL (54%) consisted of email, news and messaging software. The minority included network security and diagnostic, proxy, gateway, http and ftp server, web browsing, printing and database software.

The reference ratios of SSL options remained roughly the same throughout the various Ubuntu releases. The use of SSL options in Ubuntu Lucid is illustrated in Figure 2.

The use of `SSL_get_verify_result()` function (37.7%) indicates that a substantial proportion of SSL-capable software has interest in obtaining the results of the certificate verification. The `SSL_get_peer_certificate()` function (64.9%) is used to obtain the certificate sent by the peer.

The use of the `SSL_CTX_use_privatekey_file()` function (62.3%) implies that a majority of the software is capable of using private keys stored in files. A third (27.3%) of the applications use the `SSL_get_current_cipher()` function to request information about the cipher used for the current session.

The `SSL_accept()` function (41.6%) is the SSL equivalent for `accept()`. The reference ratio of `SSL_connect()` function (76.6%), an SSL equivalent for `connect()`, is higher than for `ssl_accept()` (41.6%). This implies that the data set includes more client-based applications than server-based. Furthermore, we observed that `SSL_shutdown()` (63.6%) is referenced in only about half of the software that also references `SSL_connect()`, indicating that clients leave dangling connections with servers (possibly due to sloppy coding practices).

We noticed that only 71.4% of the SSL-capable software initialized the OpenSSL library correctly. The correct procedure for a typical SSL application is that it should initialize the library with `SSL_library_init()` function (71.4%) and provide readable error strings with `SSL_load_error_strings()` function (89.6%) before any SSL action takes place. However, 10.4% of the SSL-capable software fails to provide adequate error handling.

Only 58.4% of the SSL-capable applications seed the Pseudo Random Number Generator (PRNG) with `RAND_load_file()` (24.7%), `RAND_add()` (6.5%) or `RAND_seed()` (37.7%). This is surprising because incorrect seeding of the PRNG is considered a common security pitfall.

Roughly half of the SSL-capable software set the context options for SSL with `SSL_CTX_set_options` (53.3%); this modifies the default behavior of the SSL implementation. The option `SSL_OP_ALL` (37.7%) enables all bug fixes.

`SSL_OP_NO_SSLV2` option (31.2%) turns off SSLv2 and respectively `SSL_OP_NO_SSLV3` (13.0%) turns off the support for SSLv3. The two options were usually combined so that the application would just use TLSv1.

`SSL_OP_SINGLE_DH_USE` (7.8%) forces the implementation to re-compute the private part of the Diffie-Hellman key exchange for each new connection. With the exception of low-performance CPUs, it is usually recommended that this option to be turned on since it improves security.

The option `SSL_OP_DONT_INSERT_EMPTY_FRAGMENTS` (6.5%) disables protection against an attack on the block-chaining ciphers. The countermeasure is disabled because some of the SSLv3 and TLSv1 implementations are unable to handle it properly.

37.7% of the SSL-capable software prefers to use only TLSv1 (`TLSv1_client_method()`) and 20.1% of the SSL-capable software prefers to fall back from TLSv1 to SSLv3 when the server does not support TLSv1. However, the use of `SSL_OP_NO_TLSV1` option indicates that 7% of the software is able to turn off TLSv1 support completely. `SSL_OP_CIPHER_SERVER_PREFERENCE` is used to indicate that the server's preference in the choosing of the cipher takes precedence. `SSL_OP_NO_SESSION_RESUMPTION_RENEGOTIATION` indicates the

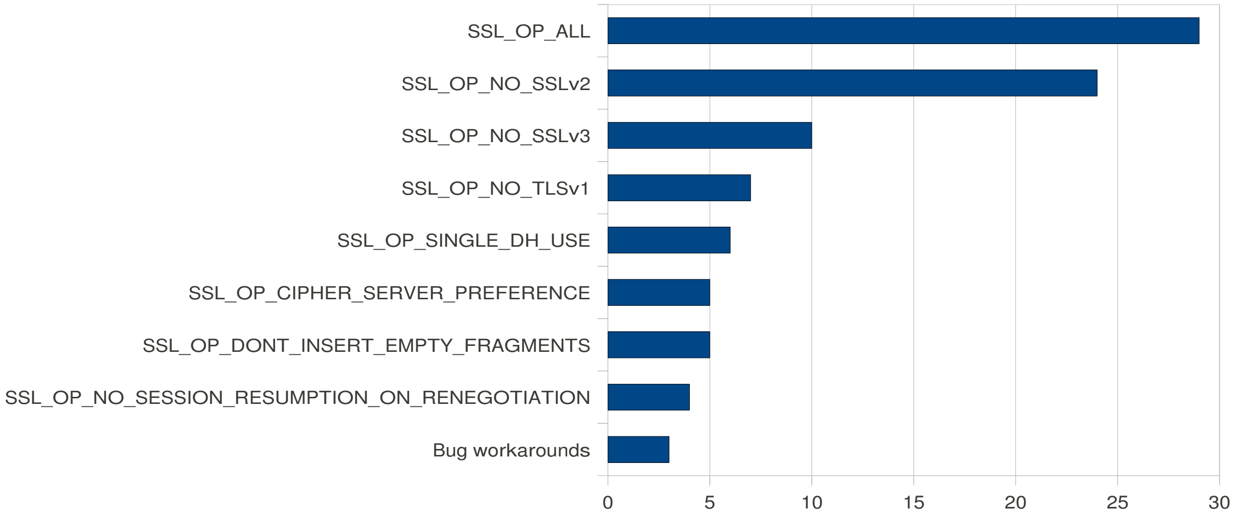

Figure 2: The number of occurrences of the most common SSL options

need for increased security as session resumption is disallowed and a full handshake is always required. The remaining options are workarounds for various bugs.

As a summary of the SSL results, it appears that SSL-capable applications are interested of the details of the security configuration. However, some applications initialize OpenSSL incorrectly and also trade security for backwards compatibility.

4.2.2 IPv6-Related Extensions

During the long transition to IPv6, we believe that the simultaneous co-existence of IPv4 and IPv6 still represents problems for application developers. For example, IPv6 connectivity is still not guaranteed to work everywhere. At the client side, this first appears as a problem with DNS look-ups if they are operating on top of IPv6. Therefore, some applications may try to look up simultaneously over IPv4 and IPv6 [25]. After this, the application may even try to call `connect()` simultaneously over IPv4 and IPv6. While these approaches can decrease the initial latency, they also generate some additional traffic to the Internet and certainly complicate networking logic in the application.

At the server side, the applications also have to maintain two sockets: one for IPv4 and another for IPv6. We believe this unnecessarily complicates the network processing logic of applications and can be abstracted away by utilizing network-application frameworks.

An immediate solution to the concerns regarding address duplication is proposed in RFC4291 [6], which describes IPv6-mapped IPv4 addresses. The idea is to embed IPv4 addresses in IPv6 address structures and thus to provide a unified data structure format for storing addresses in the application.

Mapped addresses can be employed either manually or by the use of `AI_V4MAPPED` flag for the `getaddrinfo()` resolver. However, the application first has to explicitly enable the `IPV6_V6ONLY` socket option (0.1%) before the networking stack will allow the IPv6-based socket to be used for IPv4 networking. By default, IPv4 connectivity with IPv6 sockets is disallowed in Linux because they introduce security risks [10]. As a bad omen, of the total six applications referencing the `AI_V4MAPPED` flag, only one of them set the socket option as safe guard.

The constants introduced by the IPv6 Socket API for Source Address Selection [13] are available in Ubuntu Lucid even though the support is incomplete. The flags to extend the `getaddrinfo()` resolver and the proposed auxiliary functions remain unavailable and only source address selection through socket options is available. Nevertheless, we calculated the proportion of IPv6-capable client-side applications that explicitly choose a source address. As an upper bound, 66.9% percent applications choose source addresses explicitly based the dual use of `connect()` and `bind()`. This means that a majority of IPv6 applications might be potentially interested of the extensions for IPv6 Socket API for Source Address Selection.

4.2.3 Other Protocol Extensions

The use of SCTP was very minimal in our set of applications and only three applications used SCTP. *Netperf* is a software used for benchmarking the network performance of various protocols. *Openser* is a flexible SIP proxy server. Linux Kernel SCTP tools *(lksctp-tools)* can be used for testing SCTP functionality in the userspace.

As with SCTP, DCCP was also very unpopular. It was referenced only from a single software package, despite it being easier to embed in an application by merely using the `SOCK_DCCP` constant in the socket creation.

As described earlier, multipath TCP, HIP and SHIM6 have optional native APIs. The protocols can be used transparently by legacy applications. This might boost their deployment when compared with the mandatory changes in applications for SCTP and DCCP.

The APIs for HIP-aware applications [9] may also face a similar slow adoption path because the APIs require a new domain type for sockets in the Linux kernel. While `getaddrinfo()` function can conveniently look up "wildcard" domain types, the success of this new DNS resolver (23.5%) is still challenged by the deprecated `gethostbyname()` (43.3%). SHIM6 does not face the same problem as it works without any changes to the resolver and connections can be transparently "upgraded" to SHIM6 during the communications.

The shared multihoming API for HIP- and SHIM6-aware applications [8] may have a smoother migration path. The API relies heavily on socket options and little on ancillary options. This strikes a good balance because `setsockopt()` is familiar to application developers (42.8%) and `sendmsg()` / `recvmsg()` with its ancillary option is not embraced by many (7%). The same applies to the API for Multipath TCP [16] that consists solely of socket options.

4.2.4 A Summary of the Sockets API Findings and Their Implications

Table 2 highlights ten of the most important findings in the Sockets APIs. Next, we go through each of them and argue their implications to the development of network applications.

Core Sockets API		
1	IPv4-IPv6 hybrids	26.9%
2	TCP-UDP hybrids	26.3%
3	Obsolete DNS resolver	43.3%
4	UDP-based apps with multihoming issue	45.7%
5	Customize networking stack	51.4%
OpenSSL-based applications		
6	Fails to initialize correctly	28.6%
7	Modifies default behavior	53.3%
8	OpenSSL-capable applications in total	10.9%
Estimations on IPv6-related extensions		
9	Potential misuse with mapped addresses	83.3%
10	Explicit IPv6 Source address selection	66.9%

Table 2: Highlighted indicator sets and their reference ratios

Finding 1. The number of hybrid applications supporting both IPv4 and IPv6 was fairly large. While this is a good sign for the deployment of IPv6, the dual addressing scheme doubles the complexity of address management in applications. At the client side, the application has to choose whether to handle DNS resolution over IPv4 or IPv6, and then create the actual connection with either family. As IPv6 does not even work everywhere yet, the client may initiate communications in parallel with IPv4 and IPv6 to minimize latency. Respectively, server-side applications have to listen for incoming data flows on both families.

Finding 2. Hybrid applications using both TCP and UDP occur as frequently as TCP-only applications. Application developers seem to write many application protocols to be run with both transports. While it is possible to write almost identical code for the two transports, the Sockets API favors different functions for the two. This unnecessarily complicates the application code.

Finding 3. The obsolete DNS resolver was referenced twice as frequently as the new one. This has negative implications on the adoption of new Sockets API extensions that are dependent on the new resolver. As concrete examples, native APIs for HIP and source address selection for IPv6 may experience a slow adoption path.

Finding 4. We discovered a UDP multihoming problem at the server side based on our experiments with three software included in the data set. As an upper bound, we estimated that the same problem affects 45.7% of the UDP-based applications.

Finding 5. Roughly half of the networking software is not satisfied with the default configuration of networking stack and alters it with socket options, raw sockets or other low-level hooking. However, we did not discover any patterns (besides few popular, individually recurring socket options) to propose as new compound socket option profiles for applications.

Findings 6, 7 and 8. Roughly every tenth application was using OpenSSL but surprisingly many failed to initialize it appropriately, thus creating potential security vulnerabilities. Half of the OpenSSL-capable applications were modifying the default configuration in some way. Many of these tweaks improved backwards compatibility at the expense of security. This opens a question why backwards compatibility is not well built into OpenSSL and why so many "knobs" are even offered to the developer.[3]

Finding 9. IPv6-mapped IPv4 addresses should not be leaked to the wire for security reasons. As a solution, the socket option `IPV6_V6ONLY` would prevent this leakage. However, only one out of total six applications using mapped addresses were actually using the socket option. Despite the number of total applications using mapped address in general was statistically small, this is an alarming sign because the number can grow when the number of IPv6 applications increases.

Finding 10. IPv6 source address selection lets an application to choose the type of an IPv6 source address instead of explicitly choosing one particular address. The extensions are not adopted yet, but we estimated the need for them in our set of applications. Our coarse-grained estimate is that two out of three IPv6 applications might utilize the extensions.

We have now characterized current trends with C-based applications using Sockets API directly and highlighted ten important findings. Of these, we believe findings 3, 4, 6 and 9 can be directly used to improved the existing applications in our data set. We believe that most of the remaining ones are difficult to improve without introducing changes to the Sockets API (findings 1, 2, 5) or without breaking interoperability (finding 7). Also, many of the applications appear not to need security at all (finding 8) and the adoption of extensions (finding 10) may just take some time.

As some of the findings are difficult to adapt to the applications using Sockets API directly, perhaps indirect approaches as offered by network application frameworks may offer easier migration path. For example, the first two findings are related to management of complexity in the Sockets API and frameworks can be used to hide such complexity from the applications.

4.3 Network Application Frameworks

In this section, we investigate four network application frameworks based the Sockets and POSIX API. In a way, these frameworks are just other "applications" using the Sockets API and, thus, similarly susceptible to the same analysis as the applications in the previous sections. However, the benefits of improving a single framework transcend to numerous applications as frameworks are utilized by several applications. The Sockets API may be difficult to change, but can be easier to change the details how a framework implements the complex management of the Sockets API behind its high-level APIs.

4.3.1 Generic Requirements for Modern Frameworks

Instead of applying the highlighted findings described in Section 4.2.4 directly, some modifications were made due to the different nature of network application frameworks.

Firstly, we reorganize the analysis "top down" and split the topics into end-host naming, look up, multiplicity of names and transport protocols and security. We also believe that the reorganization may be useful for extending the analysis in the future.

Secondly, we arrange the highlighted findings according to their topic. A high-level framework does not have to follow the IP address oriented layout of the Sockets API and, thus, we investigate the use of symbolic host names as well. The reconfiguration of the stack (finding 5) was popular but we could not suggest any significant improvements on it, so it is omitted. Finally, we split initiating of parallel connectivity with IPv4 and IPv6 as their own requirements for both transport connections and DNS look ups.

[3] Some of the implementations of SSL/TLS are considered "broken"; they do not implement at all or fix incorrectly some of the bugs and/or functionalities in SSL/TLS.

Consequently, the following list reflects the Sockets API findings as modified requirements for network application frameworks:

R1: End-host naming

R1.1 Does the API of the framework support symbolic host names in its APIs, i.e., does the framework hide the details of hostname-to-address resolution from the application? If this is true, the framework conforms to a similar API as proposed by Name Based Sockets as described in section 2.2. A benefit of this approach is that implementing requirements R1.2, R2.2, R3.1 and 3.3 becomes substantially easier.

R1.2 Are the details of IPv6 abstracted away from the application? In general, this requirement facilitates adoption of IPv6. It could also be used for supporting Teredo based NAT traversal transparently in the framework.

R1.3 IPv6-mapped addresses should not be present on the wire for security reasons. Thus, the framework should manually convert mapped addressed to regular IPv4 addresses before passing to any Sockets API calls. Alternatively, the frameworks can use the `AI_V4MAPPED` option as a safe guard to prevent such leakage.

R2: Look up of end-host names

R2.1 Does the framework implement DNS look ups with `getaddrinfo()`? This is important for IPv6 source address selection and native HIP API extensions because they are dependent on this particular function.

R2.2 Does the framework support parallel DNS look ups over IPv4 and IPv6 to optimize latency?

R3: Multiplicity of end-host names

R3.1 IPv6 source address selection is not widely adopted yet but is the framework modular enough to support it especially at the client side? As a concrete example, the framework should support inclusion of new parameters to its counterpart of `connect()` call to support application preferences for source address types.

R3.2 Does the server-side multihoming for UDP work properly? As described earlier, the framework should use `SO_BINDTODEVICE` option or `sendmsg()/recvmsg()` interfaces in a proper way.

R3.3 Does the framework support parallel `connect()` over IPv4 and IPv6 to minimize the latency for connection set-up?

R4: Multiplicity of transport protocols

R4.1 Are TCP and UDP easily interchangeable? “Easy” here means that the developer merely changes one class or parameter but the APIs are the same for TCP and UDP. It should be noted that this has also implications on the adoption of SCTP and DCCP.

R5: Security

R5.1 Does the framework support SSL/TLS?

R5.2 Does the SSL/TLS interface provide reasonable defaults and abstraction so that the developer does not have to configure the details of the security?

R5.3 Does the framework initialize the SSL/TLS implementation automatically?

4.3.2 ACE

ACE version 6.0.0 denotes one end of a transport-layer session with `ACE_INET_Addr` class that can be initiated both based on a symbolic host name and a numeric IP address. Thus, the support for IPv6 is transparent if the developer relies solely on host names and uses `AF_UNSPEC` to instantiate the class. ACE also supports storing of IPv4 addresses in the IPv6-mapped format internally but translates them to the normal IPv4 format before returning them to the requesting application or using on the wire.

In ACE, IP addresses can be specified using strings. This provides a more unified format to name hosts.

ACE supports `getaddrinfo()` function and resorts to `getnameinfo()` only when the OS (e.g. Windows) does not support `getaddrinfo()`.

With UDP, ACE supports both connected (class `ACE_SOCK_CODgram`) and disconnected communications (class `ACE_SOCK_Dgram`). We verified the UDP multihoming problem with test software included in the ACE software bundle. More specifically, we managed to repeat the problem with connected sockets which means that the ACE library shares the same bug as iperf, nc and nc6 software as described earlier. Disconnected UDP communications did not suffer from this problem because ACE does not fix the remote communication end-point for such communications with `connect()`. It should be also noted that a separate class, `ACE_Multihomed_INET_Addr`, supports multiaddressing natively.

A client can connect to a server using TCP with class `ACE_SOCK_Connector` in ACE. The instantiation of the class supports flags which could be used for extending ACE to support IPv6 source address selection in a backwards compatible manner. While the instantiation of connected UDP communications does not have a similar flag, it still includes few integer variables used as binary arguments that could be overloaded with the required functionality. Alternatively, new instantiation functions with different method signature could be defined using C++. As such, ACE seems modular enough to adopt IPv6 source address selection with minor changes.

For basic classes, ACE does not support accepting of communications simultaneously with both IPv4 and IPv6 at the server side. Class `ACE_Multihomed_INET_Addr` has to be used to support such behaviour more seamlessly but it can be used both at the client and server side.

Changing of the transport protocol in ACE is straightforward. Abstract class `ACE_Sock_IO` defines the basic interfaces for sending and transmitting data. The class is implemented by two classes: an application instantiates `ACE_Sock_Stream` class to use TCP or `ACE_SOCK_Dgram` to use UDP. While both TCP and UDP-specific classes supply some additional transport-specific methods, switching from one transport to another occurs merely by renaming the type of the class at the instantiation, assuming the application does not need the transport-specific methods.

ACE supports SSL albeit it is not as interchangeable as TCP with UDP. ACE has wrappers around `accept()` and `connect()` calls in its Acceptor-Connector pattern. This hides the intricacies of SSL but all of the low-level details are still configurable when needed. SSL is initialized automatically and correctly.

4.3.3 Boost::Asio

Boost::Asio version 1.47.0 provides a class for denoting one end of a transport-layer session called `endpoint` that can be initiated through resolving a host name or a numeric IP. By default, the resolver returns a set of endpoints that may contain both IPv4 and IPv6 addresses.[4] These endpoints can be given directly to the `connect()` wrapper in the library that connects sequentially to the addresses found in the endpoint set until it succeeds. Thus, the support for IPv6 is transparent if the developer has chosen to rely on host names. Boost::Asio can store IPv4 addresses in the IPv6-mapped form. By default, the mapped format is used only when the developer explicitly sets the family of the address to be queried to IPv6 and the query results contain no IPv6 addresses. The mapped format is only used internally and converted to IPv4 before use on the wire.

Boost::Asio uses POSIX `getaddrinfo()` when the underlying OS supports it. On systems such as Windows (older than XP) and Cygwin, Boost::Asio emulates `getaddrinfo()` function by calling `gethostbyaddr()` and `gethostbyname()` functions. The resolver in Boost::Asio includes flags that could be used for implementing source address selection (and socket options are supported as well).

Boost::Asio does not support parallel IPv4 and IPv6 queries, nor does it provide support for simultaneous connection set up using both IPv4 and IPv6.

We verified the UDP multihoming problem with example software provided with the Boost::Asio. We managed to repeat the UDP multihoming problem with connected sockets which means that the Boost::Asio library shares the same bug as iperf, nc and nc6 as described earlier.

Boost::Asio defines basic interfaces for sending and receiving data. An application instantiates `ip::tcp::socket` to use TCP or `ip::udp::socket` to use UDP. While both classes provide extra transport-specific methods, switching from one transport to another occurs merely by renaming the type of the class at the in-

[4] IPv6 addresses are queried only when IPv6 loopback is present

stantiation assuming the application does not need the transport-specific methods.

Boost::Asio supports SSL and TLS. The initialization is wrapped into the SSL context creation. In Boost::Asio, the library initialization is actually done twice as `OpenSSL_add_ssl_algorithms()` is a synonym of `SSL_library_init()` and both are called sequentially. PRNG is not automatically initialized with `RAND_load_file()`, `RAND_add()` or `RAND_seed()`, although Boost::Asio implements class `random_device` which can be easily used in combination with `RAND_seed()` to seed the PRNG.

4.3.4 Java.net

Java.net in OpenJDK Build b147 supports both automated connections and manually created ones. Within a single method that inputs a host name, its API hides resolving a host name to an IP address from DNS, creation of the socket and connecting the socket. Alternatively, the application can manage all of the intermediate steps by itself.

The API has a data structure to contain multiple addresses from DNS resolution. The default is to try a connection only with a single address upon request, albeit this is configurable. The internal presentation of a single address, `InetAddress`, can hold an IPv4 or IPv6 address and, therefore, the address family is transparent when the developer resorts solely on the host names. The API supports `v4_mappedaddress` format as an internal presentation format but it is always converted to the normal IPv4 address format before sending data to the network.

Before using IPv6, Java.net checks the existence of the constant `AF_INET6` and that a socket can be associated with a local IPv6 address. If java.net discovers support for IPv6 in the local host, it uses the `getaddrinfo()` but otherwise `gethostbyname()` function for name resolution. DNS queries simultaneously over IPv4 and IPv6 are not supported out-of-the-box. However, the SIP ParallelResolver package in SIP communicator[5] could be used to implement such functionality.

We verified the UDP multihoming problem with example software provided with the java.net. We managed to repeat the UDP multihoming problem with connected sockets. This means that the java.net library shares the same bug as iperf, nc and nc6 as described earlier.

Java.net naming convention favors TCP because a "socket" always refers to a TCP-based socket. If the developer needs a UDP socket, he or she has to instantiate a `DatagaramSocket` class. Swapping between the two protocols is not trivial because TCP-based communication uses streams, where as UDP-based communication uses `DatagramPacket` objects for I/O.

IPv6 source address selection is implementable in java.net. TCP and UDP-based sockets could include a new type of constructor or method, and java has socket options as well. The method for DNS look ups, `InetAddress.getByName()`, is not extensive enough and would need an overloaded method name for the purpose.

Java.net supports both SSL and TLS. Their details are hidden by abstraction, although it is possible to configure them explicitly. All initialization procedures are automatic.

4.3.5 Twisted

With Twisted version 10.2, python-based applications can directly use host names to create TCP-based connections. However, the same does not apply to UDP; the application has to manually resolve the host name into an IP address before use.

With the exception of resolving of AAAA records from the DNS, IPv6 support is essentially missing from Twisted. Thus, mapped addresses and parallel connections over IPv4 and IPv6 remain unsupported due to lack of proper IPv6 support. Some methods and classes include "4" suffix to hard code certain functions only to IPv4 which can hinder IPv6 interoperability.

Introducing IPv6 source address selection to Twisted would be relatively straightforward, assuming IPv6 support is eventually implemented. For example, Twisted methods wrappers for `connect()` function input host names. Therefore, the methods could be adapted to include a new optional argument to specify source address preferences.

The twisted framework uses `gethostbyname()` but has also its own implementation of DNS, both for the client

[5] net.java.sip.communicator.util.dns.ParallelResolver

and server side. As IPv6 support is missing, the framework cannot support parallel look ups.

The UDP multihoming issue is also present in Twisted. We observed this by experimenting with a couple of client and server UDP applications in the Twisted source package.

TCP and UDP are quite interchangeable in Twisted when the application uses the `Endpoint` class because it provides abstracted read and write operations. However, two discrepancies exists. First, `Creator` class is tainted by TCP-specific naming conventions in its method `connectTCP()`. Second, applications cannot read or write UDP datagrams directly using host names but first have to resolve them into IP addresses.

Twisted supports TLS and SSL in separate classes. TLS/SSL can be plugged into an application with relative ease due to modularity and high-level abstraction of the framework. The details of SSL/TLS are configurable and Twisted provides defaults for applications that do not need special configurations. With the exception of seeding the PRNG, the rest of the details of TLS/SSL initialization are handled automatically.

4.3.6 A Summary of the Framework Results

We summarize how the requirements were met by each of the four frameworks in Table 3. Some of the requirements were unmet in all of the frameworks. For example, all frameworks failed to support UDP-based multihoming (R3.2) and parallel IPv4/IPv6 connection initialization for clients (R3.3). Also, SSL/TLS initialization (R5.3) was not implemented correctly in all frameworks. In total, 56 % of our requirements were completely met in all of the frameworks.

5 Related and Future Work

At least three other software-based approaches to analyze applications exist in the literature. Camara et al. [3] developed software and models to verify certain errors in applications using the Sockets API. Ammons et al. [1] have investigated machine learning to reverse engineer protocol specifications from source code based on the Sockets API. Palix et al. [14] have automatized finding of faults in the Linux kernel and conducted a longitudinal study.

Req.	ACE	Boost::Asio	Java.net	Twisted
R1.1	✓		✓	(✓)
R1.2	✓	✓	✓	
R1.3	✓	✓	✓	N/A
R2.1	✓	✓	✓	
R2.2				
R3.1	✓	✓	✓	✓
R3.2				
R3.3				
R4.1	✓	✓		(✓)
R5.1	✓	✓	✓	✓
R5.2	✓	✓	✓	✓
R5.3	✓	(✓)	✓	(✓)

Table 3: Summary of how the frameworks meet the requirements

We did not focus on the development of automatized software tools but rather on the discovery of a number of novel improvements to applications and frameworks using the Sockets API. While our findings could be further automatized with the tools utilized by Camara, Ammons and Palix et al., we believe such an investigation would be in the scope of another article.

Similarly to our endeavors with multihoming, Multiple Interfaces working group in the IETF tackles the same problem but in broader sense [2, 24]. Our work supplements their work, as we explained a very specific multihoming problem with UDP, the extent of the problem in Ubuntu Linux and the technical details how the problem can be addressed by developers.

6 Conclusions

In this article, we showed empirical results based on a statistical analysis of open-source network software. Our aim was to understand how the Sockets APIs and its extensions are used by network applications and frameworks. We highlighted ten problems with security, IPv6 and configuration. In addition to describing the generic technical solution, we also reported the extent of the problems. As the most important finding, we discovered that 28.6% of the C-based network applications in Ubuntu are vulnerable to attacks because they fail to initialize OpenSSL properly.

We applied the findings with C-based applications to four example frameworks based on the Sockets API. Contrary to the C-based applications, we analyzed the frameworks in a top-down fashion along generalized dimensions of end-host naming, multiplicity of names and transports, name look up and security. Consequently, we proposed 12 networking requirements that were completely met by a little over half of the frameworks in total. For example, all four frameworks consistently failed to support UDP-based multihoming and parallel IPv4/IPv6 connection initialization for the clients. Also the TLS/SSL initialization issue was present in some of the frameworks. With the suggested technical solutions for Linux, we argue that hand-held devices with multi-access capabilities have improved support for UDP, the end-user experience can be improved by reducing latency in IPv6 environments and security is improved for SSL/TLS in general.

7 Acknowledgments

We would like to thank Tao Wan for his initial work with the topic. We appreciate the discussion with Dmitriy Kuptsov, Antti Louko, Teemu Koponen, Antti Ylä-Jääski, Jukka Nurminen, Andrey Lukyanenko, Boris Nechaev, Zhonghong Ou, Cui Yong, Vern Paxon, Stefan Götz and Suvi Koskinen around the topic. The authors also express their gratitude to anonymous reviewers for their comments. This work was supported by grant numbers 139144 and 135230 from the Academy of Finland.

References

[1] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 4–16, New York, NY, USA, 2002. ACM.

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

[3] P. de la Cámara, M. M. Gallardo, P. Merino, and D. Sanán. Model checking software with well-defined apis: the socket case. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, FMICS '05, pages 17–26, New York, NY, USA, 2005. ACM.

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

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

[6] R. Hinden and S. Deering. IP Version 6 Addressing Architecture. RFC 4291 (Draft Standard), February 2006. Updated by RFCs 5952, 6052.

[7] C. Huitema. RFC 4380: Teredo: Tunneling IPv6 over UDP through Network Address Translations (NATs), February 2006.

[8] M. Komu, M. Bagnulo, K. Slavov, and S. Sugimoto. Sockets Application Program Interface (API) for Multihoming Shim. RFC 6316 (Informational), July 2011.

[9] M. Komu and T. Henderson. Basic Socket Interface Extensions for the Host Identity Protocol (HIP). RFC 6317 (Experimental), July 2011.

[10] Craig Metz and Jun ichiro itojun Hagino. IPv4-Mapped Addresses on the Wire Considered Harmful, October 2003. Work in progress, expired in Oct, 2003.

[11] Robert Moskowitz, Pekka Nikander, Petri Jokela, and Thomas R. Henderson. RFC 5201: Host Identity Protocol, April 2008.

[12] E. Nordmark and M. Bagnulo. Shim6: Level 3 Multihoming Shim Protocol for IPv6. RFC 5533 (Proposed Standard), June 2009.

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

[14] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. Faults in linux: ten years later. In Rajiv Gupta and Todd C. Mowry, editors, *ASPLOS*, pages 305–318. ACM, 2011.

[15] Eric Rescorla. *SSL and TLS, Designing and Building Secure Systems*. Addison-Wesley, 2006. Tenth printing.

[16] Michael Scharf and Alan Ford. MPTCP Application Interface Considerations, November 2011. Work in progress, expires in June, 2012.

[17] Douglas C. Schmidt. The adaptive communication environment: An object-oriented network programming toolkit for developing communication software. pages 214–225, 1993.

[18] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022 (Informational), January 2001.

[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, Bill Fenner, and Andrew M. Rudoff. *Unix Network Programming, Volume 1, The Sockets Networking API*. Addison-Wesley, 2004. Fourth printing.

[21] R. Stewart. RFC 4960: Stream Control Transmission Protocol, September 2007.

[22] T.Dierks and E. Rescorla. RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2, August 2008.

[23] Javier Ubillos, Mingwei Xu, Zhongxing Ming, and Christian Vogt. Name Based Sockets, September 2010. Work in progress, expires in March 2011.

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

[25] D. Wing and A. Yourtchenko. Happy Eyeballs: Success with Dual-Stack Hosts. RFC 6555 (Proposed Standard), April 2012.