Common Lisp Sockets APIs
An analysis and proposal

Dave Roberts
dave-at-findinglisp-com
www.findinglisp.com

Version 0.2.5, 27-Dec-2005

1 Introduction

In recent years, network programming has increased steadily in importance. As the Internet has penetrated even the fabric of daily life, it has also penetrated the fabric of computer programming. It is rare these days that non-networking-oriented programs such as word processors and document viewers do not have at least some networked component for retrieving a remote document or at least automatically updating the program itself to patch security holes or add functionality. Clearly, being able to write network-based programs is important in the 22nd century.

In spite of this fact, Common Lisp has no standardized networking API. While Lisp has a long history of adapting to a variety of programming styles and environmental conditions, Common Lisp was standardized before the popularization of the Internet. Thus, while the Common Lisp ANSI committee saw the need to standardize file I/O and even a very general, OS-agnostic mechanism for the encoding of file pathnames, the language does not have any standard library routines for working with network connections and network objects.

To fill this hole, most Common Lisp implementations have developed a network API of their own. Unfortunately, these APIs suffer from two main problems.

First, each implementation developed its own API without the benefit of a standard and thus each is different from the others. The undesired variety found in the various APIs retards the development of higher-layer library routines for various application protocols. Indeed, what library routines do exist today have complex, implementation-specific code woven throughout in order to get them to run. Most library code works only on a couple of implementations to which the author had access during development. At best, this situation leads to numerous "compatibility layers" that try to smooth over the variations and provide a slightly higher standardized interface. These compatibility layers can lead to tremendous inefficiencies, however. For instance, it is common in such

compatibility layers to copy data between buffers in various formats, slowing down processing and creating more work for the garbage collector in the process.

Second, most of the current APIs are deficient in one way or another. Most of the APIs provide access to only a subset of standard TCP/IP functionality as exposed through other operating system API interfaces such as BSD sockets. While the functionality provided is sufficient to implement many standard protocols, many other well-known protocols rely on missing functionality and cannot be implemented without resorting to FFI interfaces. Compounding this first problem, the missing functionality differs between implementations. Thus, even for more standard protocols, it is difficult to know what will have to be done to make them work on any given implementation. As a simple example, many implementations do not have support for UDP datagram sockets. Application protocols such as DNS, TFTP, and RADIUS cannot be implemented in these CL implementations without great effort.

This paper collects information about various Common Lisp sockets APIs and describes the advantages and disadvantages we find in them. Finally, the paper proposes a new Common Lisp networking API that attempt to enable high-performance TCP/IP networking for Common Lisp programs, whether clients or servers.

2 Existing Sockets APIs

Before describing the problems and limitations of the existing APIs, it will be useful to summarize the various high-level network API features supported by multiple current Common Lisp implementations. Table 1 compares the socket APIs for the following Common Lisp implementations:

- CMUCL
- SBCL and SB-BSD-SOCKETS
- CLISP
- Allegro
- Lispworks
- OpenMCL

Note that this analysis was performed using the current documentation for each product. I have only used SBCL and CLISP for network programming. Even in those cases, my experience level is limited. In some cases, I could not determine the exact functionality solely from the documentation and did not have a full implementation with which to experiment. *Please send me corrections so I can update the information.*

[Note that some of this information is actually more than a year old; it is undoubtedly out of date as some of the implementations have released new major versions in the mean time and may have added functionality. As stated above, please send me corrections.]

	CMUCL	SBCL	CLISP	Allegro	Lispworks	OpenMCL
Sockets Implementation	CMUCL	SB-BSD-SOCKETS	SOCKET	ACL-SOCKET	COMM	OPENMCL-SOCKETS
Servers	Yes	Yes	Yes	Yes	Yes	Yes
Stream Sockets	Yes[1]	Yes	Yes	Yes	Yes	Yes
Datagram Sockets	No	Partial[2]	No	Yes	No	Yes
Multicast	No	?	No	?	No	?
UNIX Domain Sockets	Yes	Yes	No	Yes	No	Yes
Lisp Streams	Yes**	Yes	Yes	Yes	Yes	Yes
Binary Streams	Yes	Yes	Yes	Yes	Yes	Yes
Bivalent Streams	No?	No	No	Yes	?	Yes
Buffered Streams	Yes	Yes?	Optional	?	Always[5]	?
Shutdown	No	No	Yes	Yes	No	Yes
OOB	Yes	Partial[3]	No	No	No	No
Socket Options	Yes	Yes	Yes	Yes	No	Partial[7]
Nonblocking sockets	No	No?	Yes	Accept only[4]	Yes[6]	?
Multiplexing model	Serve Event	Threads, Serve Event	Select-like	Blocking Threads	Blocking Threads	Blocking Threads
Address Representation	Integer?	Vector of octets	String	Integer	Integer	Integer
IPv6 Support	No?	No?	No?	No?	No?	No?

Table 1: Summary of existing APIs

Table 1 Notes:

1 CMUCL's sockets return a file descriptor which can then be used to construct a stream.

2 In SB-BSD-SOCKETS, each write to a stream associated with a datagram socket is converted to an individual UDP datagram. *Note that this may be out of date with very recent SBCL changes.*

3 SBCL supports OOB data in the receive direction only. There is no ability to send OOB data.

4 In Allegro, it seems as if only the accept function is non-blocking, with all other sockets operating in blocking mode.

5 There doesn't seem to be a way to use non-buffered sockets with Lispworks.

6 Lispworks has non-blocking sockets, but the semantics of the API are the same as when EOF is reached, making this seem a bit suspect. Or perhaps I'm not understanding the API.

7 OpenMCL supports some socket options, but only a partial list. Socket options must be set during the call to MAKE-SOCKET and cannot be changed thereafter.

3 Analysis of Existing APIs

Given the basic data in Table 1, let's now examine the API features in more detail. It's interesting to note that no implementation gets everything right, including the commercial implementations. There are some broad classes of feedback about the APIs.

1. Problems that affect protocol development
2. Inconsistencies that hurt portability
3. Positive features that should be retained

3.1 Problems that affect protocol development

Broadly speaking, there are some issues that make it difficult, or nearly impossible to create a fully-conforming implementation of a standard application protocol. Sometimes, it's impossible to create even a *non-conforming* implementation because required API features are simply missing. The four main problems are:

1. Lack of UDP sockets — While TCP is used for the vast majority of Internet protocols, there are several key, important protocols that are implemented using UDP. Without UDP socket support, these protocols simply cannot be implemented. Such protocols include DNS, RADIUS, and TFTP.

2. Lack of bivalent streams — Most of the current CL sockets APIs access TCP sockets through the standard Common Lisp streams interface. The streams interface allows you to set the type of stream when it is created,

but does not allow it to be changed thereafter. In many Internet protocols, this is not a problem, but several important protocols require a program to switch back and forth between textual data and binary data on the same transmission channel. Using a standard CL stream to send and receive such data can be more difficult than it needs to be. Example protocols include: HTTP where headers are in text and the body can be a binary object, SMTP where the protocol exchange is in text and the system can either negotiate cryptography in mid-session or transfer a binary object. A couple of the Common Lisp implementations support the concept of "bivalent streams" which greatly help the implementation of such protocols by providing a stream interface that can be switched between textual data in different character formats or binary data.

3. Lack of independent directional shutdown — In some TCP protocols, either endpoint can signal the other that it has completed its portion of the protocol exchange by closing its end of the connection. Note that this is not the same as closing the socket itself, but rather a TCP-level protocol device called a "half close." At the operating system layer, this is commonly referred to as "shutdown" to distinguish it from a full close of both directions of the socket. Common Lisp implementations that lack shutdown capability make it difficult to implement some protocols in a conforming manner.

4. Lack of IP multicast — IP multicast usage is increasing and is critically important in some commercial vertical environments (financial services, for example). Since IP multicast often relies on UDP to provide the application-layer protocol, any implementation that doesn't support UDP cannot easily support IP multicast. It was not clear whether implementations that support UDP might also support IP multicast and it was rarely mentioned explicitly in the documentation of various implementations.

5. Lack of out-of-band data — TCP has a facility called urgent mode that allows a sender to signal a receiver that it has sent "urgent data" of some sort. This facility allows the TCP stack to send data to the receiver in spite of various TCP flow-control protocols that would otherwise prevent it (the receiver is advertising a receive window of zero, for instance). In the BSD sockets API, the urgent facility is named out-of-band data. This is a misnomer, however; all urgent data is sent in-band on the same TCP connection as regular data. The TCP urgent facility is used by protocols such as Telnet, rlogin, and FTP to send things like interrupt signals that have to get to the other side in a timely manner. Without access to this facility through the CL sockets API, it is impossible to write a fully compliant implementation of one of these protocols (in practice, you can still do a lot without this, however).

6. Lack of IPv6 support — While IPv4 is currently the dominant Internet protocol, portions of the user base are slowly starting to adopt IPv6 and

support for this protocol is only increasing. Most (all?) of the CL APIs that I examined did not provide support for IPv6.

3.2 Inconsistencies that hurt portability

In many cases, the biggest problem with all the various CL sockets APIs is simply that they are so different from each other that it is difficult to write a single program with networking functionality that runs on each without including large amounts of implementation-specific code. The following inconsistencies cause problems.

1. Inconsistent multiplexing model — In client programs, it is common to only have open a single network socket at one time. Servers, however, will routinely have tens, hundreds, or even thousands of sockets open simultaneously. Data on the sockets arrives asynchronously and each socket can be in a totally different state at any given time. The server program must multiplex processing across all those sockets and ensure that each receives service in a timely fashion. Two techniques are commonly used in various operating systems and language APIs: select-based notification (including the use of /dev/poll and other more efficient mechanisms) and blocking threads. In select-based notification, the server has one thread (or a very small number greater than one) responsible for interfacing with a group of sockets. On UNIX, the thread calls the `select(2)` system call (or uses /dev/poll or other similar mechanisms) and the operating system returns with information about which sockets have data that needs to be read, which are ready for writing, etc. Other operating systems use a similar mechanism, even if it is renamed something other than `select`. In thread-based multiplexing, a socket is assigned to a single thread which makes blocking calls on the socket to read and write data. The operating system takes care of unblocking threads at appropriate times which data is available or has been written. While both very useful, these multiplexing methods require different styles of program architecture and are not often used in the same program. Writing code to convert from one method to another is difficult and results in inefficient code for one case or the other.

2. Inconsistent socket option support — In some cases, programs need to be able to control various TCP socket options with great specificity. Terminal programs, for instance, will want to disable the Nagle algorithm to enhance interactive response. Unfortunately, many CL implementations are inconsistent in which socket options are allowed and when those options are able to be set. Some implementations allow a program to specify options only when a socket is created, while others provide a BSD sockets-like call that allows the program to (re-)set options at any time. Still others have no ability to set socket options at all.

3. Inconsistent address representation — Different implementations represent addresses in various formats. Functionally, this is not a huge problem,

but requires various conversion routines to be created if values are going to be displayed as text.

4. Incomplete database functions — The BSD sockets API has a rich set of lookup functions for retrieving host IP addresses, aliases, host names, and socket numbers. In some cases, the various CL APIs specify only a subset of the functionality available in a standard BSD socket interface. One limitation that is particularly problematic is returning just a single host address from the equivalent of BSD's `gethostbyname` function. When building reliable applications, it is necessary to be able to access hosts on alternate IP addresses if the first fails for some reason.

3.3 Positive features that should be retained

While the various CL sockets APIs are riddled with inconsistencies and differing limitations, some good solutions to common problems have also become apparent. These should be retained where possible.

1. Representing addresses as vectors of octet values — Many of the CL APIs represent an IPv4 address as an unsigned integer. While this is efficient for the internal code, it is less than optimal for a human reader. SB-BSD-SOCKETS represents its IP addresses as a vector of octet values. Thus, the IP address "1.2.3.4" can be represented in code, or more importantly in the REPL, as `#(1 2 3 4)`. This is far more helpful for a programmer trying to debug a problem.

   Note, however, that this format may be punitive for IPv6 which uses 128-bit addresses and typically represents those addresses in hexadecimal format, not the typical default of the Lisp printer, possibly with compression of long addresses. The only solutions may be to use strings where standard IPv6 address formats and compression shortcuts can be employed or to use a printing function on a CLOS object.

4 Proposal Goals

This proposal has the following goals:

1. Provide a standard API that can be implemented across a number of Common Lisp systems running on a variety of well-known (and perhaps not-so-well-known) operating systems.

2. Provide a complete API that exposes the full set of IP-related functionality as present in today's C-level sockets APIs. In short, the API should not preclude the development of any Internet protocol that may otherwise be written in C.

3. Provide an efficient API. There are two types of efficiency that we are worried about. The first, and most obvious, is how efficiently the API

maps to standard Lisp data structures, data types, and function calling protocols. The second is how efficiently the API interacts with underlying OS mechanisms. For instance, the original Java networking API relied on threads to handle blocking issues. This was sufficient for basic client-side implementations that only used a few sockets, but it did not scale well for servers that might be managing tens of thousands of open sockets. The Java API had to be extended later to support a select-like, event-driven model (NIO). We'd like to learn from this mistake and avoid repeating it. Specifically, we'll rely on lessons documented at sites such as `http://www.kegel.com/c10k.html` [1] to help guide us. As an explicit subgoal, it should be possible to write server-side applications in Common Lisp that scale to support tens of thousands of simultaneously-connected clients. There may be other limitations in the system (memory, for example), but the Common Lisp networking API should not be the bottleneck.

4. Provide an extensible API that can evolve over time. While the API should remain fairly static for interoperability reasons, new networking protocols are implemented all the time and may require extensions to support them. The API should provide a framework that is able to incorporate extensions without breaking existing code.

5. Provide a "Lispy" interface that takes advantage of Lisp's unique strengths with respect to dynamic typing, garbage collection, and other features not present in languages such as C, upon which many networking APIs are based. Thus, while we'll look to APIs such as the BSD sockets API for inspiration, we want to reflect the best strengths of these APIs into a Lisp environment rather than slavishly copying them and in the process forcing the Lisp programmer to deal with underlying C limitations.

5 Proposal Strategies

In order to achieve the various goals described in the section above, we'll employ a couple of strategies:

1. First, we'll use the fairly standard BSD sockets API as a general guide. Even on operating systems that are not UNIX-based, the BSD API has served as a model for the implementation of user-space networking APIs. For instance, while Microsoft Windows has developed its own networking API, it still supports the basic BSD API and even its proprietary API retains the same basic concepts.

2. The API should be structured into multiple layers. The lowest layer should expose the full range of functionality and all the atomic operations provided in the BSD sockets API. Higher layers will combine the basic atomic functions into standard protocol sequences to make it easier to interface to the API when common usage patterns are all that is required. For

instance, the high-level API will provide a single function that allows an application program to create a socket, perform a host name lookup, and connect the socket to the resulting address.

3. The API should support mutable operations on data buffers. In the same way that `nconc` exists to provide a mutable version of `append`, there needs to be a capability to recycle buffers, if desired, in the networking API. Simply, transferring a gigabyte of data through the API should *not necessarily* create a gigabyte of garbage for the garbage collector. I say "not necessarily" because in the same way that `append` exits, it may be more convenient for simple programs to let the API and GC manage the creation and destruction of buffers when efficiency is less of a concern.

6 Related Issues

Note that a networking API, particularly a high-level networking API designed to interface with a high-level language such as Lisp, does not exist in isolation. In particular, such things as the implementation's multi-threading model and support for international character sets have an impact on the API.

6.1 Threading Models

Networking is inherently asynchronous and event-driven. Because of this, there is an interplay between the networking API and the threading models available in the system. There are two methods for dealing with the asynchronous, event-driven nature of network programming:

1. Using non-blocking operations and polling to allow single-threaded applications to avoid blocking while waiting for data to arrive. This is commonly known as the *select or polling model*. Prior to multi-threading becoming popular as an operating system feature, this was the most common model and the original BSD sockets API reflects this. In operating systems that support only a single thread per process, the select mechanism must be integrated with all other event-delivery mechanisms (for GUI events, for instance). This is why UNIX's select handles general file descriptors, allowing it to work with UNIX domain sockets and files of various sorts, in addition to networking sockets.

2. Using multiple threads to wait on blocking operations. Commonly, a single thread is dedicated to each connection or session and blocks on a socket waiting for a particular event. This is known as the *thread model*. This model has become more popular in recent years because multi-threading is more available in popular operating systems and it is sometimes perceived as being easier to write linear code for each thread rather than using event-based dispatching. (Actually, you're trading one set of complications for another, but that's another issue.)

It's important to realize that both models have their place and have been used to write successful, bug-free software. Both models can have problems with scalability for large numbers of network connections, depending on how the underlying operating system is implemented. The case of servers handling large numbers of connections is becoming increasingly common in today's always-networked computing environment.

To handle large numbers of connections, the select model requires that the `select(2)` system call or /dev/poll mechanism be highly efficient. There are known problems with the standard UNIX `select(2)` API [1], which have led to alternative mechanisms such as /dev/poll, epoll, etc.

The threading model is also not immune to scalability problems, but puts pressure instead on the operating system's scheduler. If each connection is managed by a single thread and the server is handling 10,000 clients, the operating system will have to efficiently manage and schedule 10,000 threads. When data arrives on a socket, the appropriate thread must be quickly unblocked and placed on the run queue. Some operating systems consume too many resources per thread and are thus limited in the number of threads that can be created. What may seem like a large number of threads and processes may not seem like a large number of clients. Before Linux's NPTL support was added to the kernel, Linux was very limited in the number of threads that could be created. These limits ended up affecting other systems that ran on top of Linux, such as Java, that used a thread-based networking model. Java's event-driven, select model called NIO was designed to allow Java programs to manage many more sockets with fewer numbers of threads.

In this paper, we attempt to learn from the Java NIO experience and assume that both models have advantages and will be utilized by programmers at various times. Thus, the API provides both blocking and non-blocking sockets, along with a polling mechanism that allows single-threaded programs to retrieve events.

6.2 International Character Set Support

It is becoming increasingly common for operating systems and languages to include support for international character sets beyond ASCII or ISO 8859-1 (Latin 1). Indeed, operating systems such as Windows NT were designed with Unicode support from the start. Newer programming languages such as Java use Unicode as the base character type. Indeed, some Common Lisp implementations such as CLISP and SBCL have integrated Unicode support.

International character set support interacts with the network API at the point where strings are read/written to the network socket (often through the stream interface in Common Lisp). At that point, abstract characters need to be converted to or from a specific byte encoding format.

It's important to realize that this is actually more of a general issue for streams than it is a networking-specific issue. The stream interface needs to

[1] For example, see [1] for more information. It's a good starting point for all sorts of information regarding building efficient networking servers.

be enhanced to deal with the issues of character transfer coding formats and communication channels rather than simply assuming that characters are byte-sized and can be written directly in their raw binary format. If character sets are handled correctly at the the streams level, then the networking interface simply has to present a byte-oriented transmission channel to the streams interface and the right things will happen automatically. Java actually did a reasonable job of this fairly early on in its development and the benefits have been substantial.

In this paper, we assume that the streams interface handles things correctly and that the networking layer can be blissfully unaware of any character set issues.

7 Proposed API

To be written...

8 References

References

[1] *The C10K Problem*, `http://www.kegel.com/c10k.html`.