Joseph S. Berríos <jberrios@mail.ucf.edu>
2005-06-24
Doc number N1838=05-0098

# A Proposal to Add Sockets to the Standard Library

## I. Table of Contents

I. Table of Contents.....1

II. Motivation and Scope.....1

III. Impact on the Standard .....3

IV. Design Decisions .....3

V. Proposed Text for the Standard.....19

VI. Acknowledgements.....19

VII. References .....19

## II. Motivation and Scope

The cornerstone of network and distributed programming is message passing. Message passing is the mechanism that is used for executing threads to communicate among each other. The most popular mechanism for message passing is the Berkeley Sockets interface, universally known as *sockets* [STE98]. The venerable socket library in UNIX is the original de facto standard that influenced all socket libraries. Bill Joy originally implemented the socket library in the Berkeley UNIX OS [JOY86]. Sockets are the de facto standard application programming interface (API) for networking, spanning a wide range of systems, such as MS Windows, Mac OS X, Linux, Palm OS, and the Java Virtual Machine (JVM). This library was implemented in classic C. A sample C program that uses this library is in Figure 1. The code can be further divided in the section that initializes the socket and the code that actually uses the socket for communication. What is obvious from this code is the following:

- The initialization of sockets (line 15 to line 32) must be very precise. This makes programming sockets an error prone activity.
- If there are errors in the use of sockets, the programmer must explicitly handle them. Error handling is contained in line 27 to line 31. This can lead to an increase of size and complexity of the code.

```
1   #include <sys/types.h>
2   #include <sys/socket.h>
3   #include <sys/un.h>
4   #include <stdio.h>
5   #include <unistd.h>
6
7   int main()
8   {
9   int sockfd;
10  int len;
11  struct sockaddr_un address;
12  int result;
13  char ch = 'A';
14
15  /* socket initialization */
16  /* Create socket for client */
17  sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
18
19  /* Name the socket as agreed with the server */
20  address.sun_family = AF_UNIX ;
21  strcpy(address.sun_path, "server_socket");
22  len = sizeof(address);
23
24  /* Now connect our socket to the server's socket */
25  result = connect(sockfd, (struct sockaddr*) &address,
                     len);
26
27  if (result == -1)
28     {
29     perror("oops:  client1");
30     exit(1);
31     }
32  /* end of socket initialization */
33
34  /* We can now read and write via:  sockfd  */
35  write(sockfd, &ch, 1);
36  read(sockfd, &ch, 1);
37  printf("char from server = %c\n", ch);
38  close(sockfd);
39  exit(0);
40  }
```

Figure 1: Socket program using the traditional C library

These problems leads to the proposal to add a socket library that supports network software development in ISO C++ 0X.

# III. Impact On the Standard

This proposal is a pure extension. It proposes the implementation of a network library, but it does not require any changes to any of the standard libraries in C++. In addition it does not require any changes in the core language and can be implemented in standard C++.

# IV. Design Decisions

To facilitate its development, the socket library is divided into three sections handling exceptions, the protocols, and the socket classes.

The exception section contains a generalized `SocketExceptions` class and specialized socket exceptions. The design for the exception section is shown in Figures 2 in UML and Figure 3 in C++ declaration. The generalized class implements two functions: `message`, which returns a string message that displays an error message, and `what`,


Figure 2: UML diagram from exception classes

```
// define the exception classes
class SocketExceptions : public exception {
protected:
  const char* message(const char *on_what);
public:
  virtual const char* what() = 0;
};

class SocketException : public SocketExceptions {
public:
  const char* what();
};

class AcceptException : public SocketExceptions {
public:
  const char* what();
};

class ConnectException : public SocketExceptions {
public:
  const char* what();
};

class HostnameException  : public SocketExceptions {
public:
  const char* what();
};

class BindException  : public SocketExceptions {
public:
  const char* what();
};

class SocknameException  : public SocketExceptions {
public:
  const char* what();
};

class ListenException : public SocketExceptions {
public:
  const char* what();
};

class IOException : public SocketExceptions {
public:
  const char* what();
};
```

Figure 3: C++ declaration for exception classes

which is called whenever an exception is raised. The `what` function is written as a virtual function, and is defined in the generalized class, but it is in the specialized class where the function is actually defined. The specialized exceptions classes are as follows:

- SocketException
- AcceptException
- ConnectException
- HostnameException
- BindException
- SocketnameException
- ListenException
- IOException

A SocketException arises when a declaration error occurs. The Other exception are specialized instances of SocketException that arise in specific circumstances that their name imply.

The `ProtocolImpl` class is designed to implement the protocols to be used for communication. The design for this section is shown in Figures 4 in UML and Figure 5 in C++ declaration. The most popular current communication protocol is Internet Protocol Version 4, commonly known as IPV4 [STE98]. The `ProtocolImpl` class implements the `getDomain` function that returns the value of `domain`.

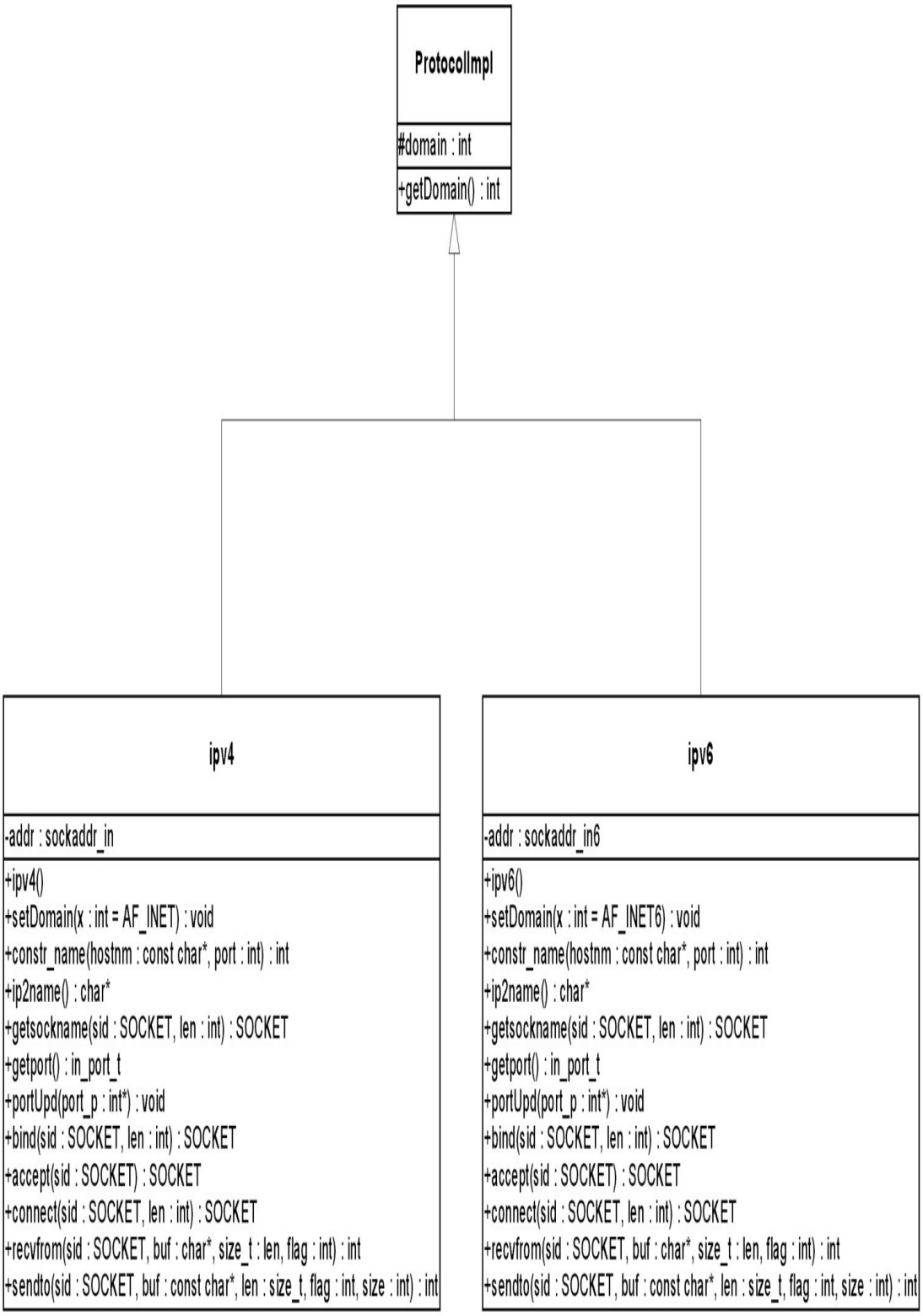


Figure 4: UML diagram for protocol classes

```
class ProtocolImpl {
public:
  int getDomain();
};

class ipv4 : public ProtocolImpl {
public:
  // Constructor
  ipv4();

  void setDomain(int x = AF_INET);

  // Build a internet socket name based on a hostname and port #
  int constr_name(const char *hostnm, int port);

  // Convert an IP address to a character string host name
  char *ip2name();

  // get socket name
  SOCKET getsockname(SOCKET sid, int len);

  // get port number
  in_port_t getPort();

  // port update
  void portUpd(int *port_p);

  // assign a UNIX or an internet name to a socket
  SOCKET bind(SOCKET sid, int len);

  // A server socket accepts a client connection request
  SOCKET accept(SOCKET sid);

  // A server socket accepts a client connection request
  SOCKET connect(SOCKET sid, int len);

  // reads a message using a datagram
  int recvfrom(SOCKET sid, char* buf, size_t len,
               int flag);

  // writes a message using a datagram
  int sendto(SOCKET sid, const char* buf, size_t len,
             int flag, int size);
};
class ipv6 : public ProtocolImpl {
public:
  // Constructor
  Ipv6();

  void setDomain(int x = AF_INET);

  // Build a internet socket name based on a hostname and port #
  int constr_name(const char *hostnm, int port);
```

```
  // Convert an IP address to a character string host name
  char *ip2name();

  // get socket name
  SOCKET getsockname(SOCKET sid, int len);

  // get port number
  in_port_t getPort();

  // port update
  void portUpd(int *port_p);

  // assign a UNIX or an internet name to a socket
  SOCKET bind(SOCKET sid, int len);

  // A server socket accepts a client connection request
  SOCKET accept(SOCKET sid);

  // A server socket accepts a client connection request
  SOCKET connect(SOCKET sid, int len);

  // reads a message using a datagram
  int recvfrom(SOCKET sid, char* buf, size_t len,
               int flag);

  // writes a message using a datagram
  int sendto(SOCKET sid, const char* buf, size_t len,
             int flag, int size);
};
```

Figure 5: C++ declaration for protocol classes

For the example of a specialized class is the ipv4. The functions defined in the class that supports the IPV4 protocol are:

- `ipv4()`: constructor function.
- `void setDomain(int x = AF_INET)`: Sets the domain type.
- `int constr_name(const char *hostnm, int port)`: Builds a internet socket name based on a hostname and a port number.
- `char *ip2name()`: Converts an IP address to a character string host name.
- `SOCKET getsockname(SOCKET sid, int len)`: Obtains the socket name.
- `in_port_t getPort()`: Obtains the port number.
- `void portUpd(int *port_p)`: Updates the port.
- `SOCKET bind(SOCKET sid, int len)`: Assigns a UNIX or an internet name to a socket.
- `SOCKET accept(SOCKET sid)`: Accepts a client connection request.
- `SOCKET connect(SOCKET sid, int len)`: Accepts a client connection request.
- `int recvfrom(SOCKET sid, char* buf, size_t len, int flag)`: Reads a message using a datagram
- `int sendto(SOCKET sid, const char* buf, size_t len, int flag, int size)`: Writes a message using a datagram

The `Socket` class inherents from two classes: `SocketSuper` and `SocketPlatform`. The `SocketSuper` class is used to implement the naming functions that enables binding at compile time. The functions implemented by this class are the `self` function that returns the `SocketType` and the `change` function that is designed to allow an object to change its `SocketType`. The design for the Socket class is shown in Figures 6 in UML and Figure 7, 8, 9, and 10 in C++ declaration.


Figure 6: UML diagram for socket classes

```
template <typename SocketType>
class SocketSuper
{
protected:
  SocketSuper();
  SocketType& Self();
  template <typename T>
  void change(T* old);
};

class SocketPlatform {
protected:
  SocketPlatform();

  // get process id
  int getpid();

  // Socket init
  bool platformInit();

  // close connection
  void platformClose(SOCKET sid);

  // close socket
  void platformCloseSocket(SOCKET sid);

  // socket error
  bool platformSocketError(SOCKET rc);

  // invalid socket
  bool platformInvalidSocket(SOCKET rc);
};
```

Figure 7: C++ declaration for SocketSuper and SocketPlatform

```
template <typename Protocol, typename SocketType>
class Socket : public SocketSuper<SocketType>, public
SocketPlatform  {
protected:
  // Build a domain  name based on a pathname
  int constr_name(sockaddr &addr, const char* const Pathnm);
  void SocketInit(int type, int prot);
  void SocketBind(const char* name = NULL, int port = 0);
  Socket(int type, int prot);
public:
  // constructor
  Socket() { init(); }
  template <typename P, typename S>
  Socket(Socket<P, S>& s);

  // destructor
  Virtual ~Socket();       // discard a socket

  void init();

  // return a socket's id #
  int SocketId();

  // assign a UNIX or an internet name to a socket
  void bind(const char* address = NULL, int port = 0);
  void bind(const std::string address, int port = 0);

  // A client initiates connection request to a server
  void connect(const char* hostnm, int port = -1);
  void connect(const std::string hostnm, int port = -1);

  // A client initiates connection request to a server
  // socket
  void connect();

  // close connection
  void close();

  // resize buffer length
  void setBuffer(int x);

  // return buffer size
  int bufferSize() { return bufLength; }

  // functions for I/O
  void write(const char* buf);
  void write(const std::string buf);

  void read(char* buf);
  void read(std::string& buf);

  // shutdown connection of a socket
  int shutdown(int mode = 2);
};  // class Socket
```

Figure 8:  C++ declaration for the Socket class

```
template <typename Protocol>
class StreamSocket : public Socket <Protocol, StreamSocket
<Protocol> > {
public:

  // constructor
  StreamSocket();
  StreamSocket(const char* address, int port);
  StreamSocket(const std::string address, int port);

  void init();
  // assign a UNIX or an internet name to a socket
  void bind(const char* address = NULL, int port = 0);

  // A server socket accepts a client connection request
  int accept(char* address, int* port_p);
  int accept()  { return accept(0, 0); }
  int accept(std::string& address, int* port_p);

  // writes a message to a connected stream socket
  void write(const char* buf);
  void write(const std::string buf);

  // reads a message from a connected stream socket
  void read(char* buf);
  void read(std::string& buf);
};  // class StreamSocket

template <typename Protocol>
class ServerStreamSocket : public StreamSocket <Protocol> {
public:
  // constructor
  ServerStreamSocket(int port);
};
```

Figure 9:  C++ declaration for the StreamSocket and ServerStreamSocket class

```
template <typename Protocol>
class DatagramSocket : public Socket <Protocol,
DatagramSocket<Protocol> > {
public:
  // constructor
  DatagramSocket();
  DatagramSocket(const char* address, int port);
  DatagramSocket(const std::string address, int port);

  void init();

  // assign a UNIX or an internet name to a socket
  void bind(const char* address = NULL, int port = 0);

  // writes a message to a connected datagram socket
  void write(const char* buf);
  void write(const std::string buf);

  // reads a message from a connected datagram socket
  void read(char* buf);
  void read(std::string& buf);
};  // class DatagramSocket
```

Figure 10: C++ declaration for the DatagramSocket class

The `SocketPlatform` is designed to implement all the platform specific functions. The functions implemented in this class are:

- `SocketPlatform()`: Constructor.
- `int getpid()`: Function that gets the process id.
- `bool platformInit()`: Function that initializes the socket.
- `void platformClose(SOCKET sid)`: Function that closes the socket.
- `void platformCloseSocket(SOCKET sid)`: Function that closes the socket and frees the resources.
- `bool platformSocketError(SOCKET rc)`: Function that returns an error message.
- `bool platformInvalidSocket(SOCKET rc)`: Function that determines wheter a socket is invalid.

The `Socket` class is implemented with the following public functions:

- `Socket()`: Constructor that calls the init() function.

- `Virtual ~Socket()`: Destructor that frees all used resources. If a socket is initialized and the user does not close it, the destructor automatically closes the socket for the programmer.

- `void init()`: Function that binds to the init function from the specialized class.

- `int SocketId()`: Function that returns a socket id.

- `void bind(const string address, int port = 0)`: Function that binds an address to a socket.

- `void connect(const std::string hostnm, int port = -1)`: Function that actively attempt to establish a connection.

- `void close()`: Function that releases a socket connection.

- `void setBuffer(int x)`: Function that sets the size of the buffer.

- `int bufferSize()`: Function that returns the size of the buffer.

- `void write(const string buf)`: Function that binds to the write function from a specialized class. This function sends a message through a socket.

- `void read(string& buf)`: Function that binds to the read function from a specialized class. This function reads a message from a socket.

- `int shutdown(int mode = 2)`: Function that shuts down the connection of a socket.

From the socket library two specialized classes were implemented. These are the `StreamSocket`, with a specialized class called `ServerStreamSocket`, and `DatagramSocket`. The `StreamSocket` class uses the Transfer Control Protocol (TCP, also known as stream-based sockets). TCP provides reliability and guarantees that the data will get to its destiny. The `StreamSocket` class provides the following functions:

- `StreamSocket()`: Constructor.

- `void init()`: Function that initializes a socket.

- `void bind(const char* address = NULL, int port = 0)`: Function that assigns a UNIX or an internet name to a socket.

- `int accept(std::string& address, int* port_p)`: Block caller until a connection request arrives.
- `void write(const string buf)`: A function that writes a message to a connected stream socket.
- `void read(string& buf)`: A function that reads a message to a connected stream socket.

A *`ServerStreamSocket`* is a specialization of the *`StreamSocket`* class. This class only contains a constructor. This constructor performs the same function as initializing a `StreamSocket` object with the server parameters.

The `DatagramSocket` class implements the User Datagram Protocol (UDP). This protocol does not guarantee that the packets will be delivered nor that they will arrive in the order sent by the originator. The main advantage is that this protocol does not consume as many resources as TCP, and it is much faster than TCP. The functions provided by this class are:

- `DatagramSocket()`: Constructor.
- `void init()`: Function that initializes a socket.
- `void bind(const char* address = NULL, int port = 0)`: Function that assigns a UNIX or an Internet name to a socket.
- `void write(const string buf)`: Function that writes a message to a connected datagram socket.
- `void read(string& buf)`: Function that reads a message to a connected datagram socket.

A sample C++ program that uses this library is shown in Figure 11.

```
1   #include <cstdio>
2   #include <iostream>
3   #include <string>
4   #include "Socket.h"
5
6   using namespace std;
7
8   int main(int argc, char* argv[])
9   {
10  const string MSG1 = "Hello MSG1";
11  string buf;
12
13  int port = -1;
14  if (argc < 2)
15     {
16     cerr << "usage: " << argv[0]
17          << " <sockname|port> [<host>]\n";
18     return 1;
19     }
20
21  // check if port no of a socket name is specified
22  sscanf(argv[1], "%d", &port);
23
24  // 'host' may be a socket name of a host name
25  char *host = (port == -1) ? argv[1] : argv[2];
26
27  try {
28      // create a client socket and connect it
29      net::StreamSocket<net::ipv4> socket(host, port);
30
31      // send MSG1 to a server socket
32      socket.write(MSG1);
33
34      // read MSG2 from server
35      socket.read(buf);
36      cout << "Client:  received msg using read: " << buf
37           << endl;
38
39      // shut down socket explicitly (this command is
optional, the destructor can close the socket
40      socket.shutdown();
41      }
42  catch(net::SocketExceptions& e)
43      {
44      cout << e.what() << endl;
45      }
```

```
46  }
```

Figure 11:  Socket program using the C++ socket library

# V. Proposed Text for the Standard

# VI. Acknowledgements

# VII. References

[STE98] Stevens, W. R.:  UNIX Network Programming:  Networking APIs: Sockets and XTI.  Prentice Hall, Englewood Cliffs, NJ, 1998.

[JOY86] Joy, W. N., Fabry, R. S., Leffler, S. J., McKusick, M. K., and Karels, M. J.:  “Berkeley Software Architecture Manuel, 4.3BSD Edition.”  UNIX Programmer’s Supplementary Documents, Volume 1, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version, USENIX Association, Berkeley, California, pp. 6:1-6:43, 1986.