Doc. no: WG21/N1925=J16/05-0185
Date: 2005-12-04
Project: Programming Language C++
Reply to: Gerhard Wesp <gwesp@google.com>

Networking proposal for TR2 (rev. 1)

Contents

1 Motivation and Scope

File I/O has been a part of C++ since its beginnings. As networking I/O becomes increasingly important and in some areas even more important than file I/O, it seems natural to add this functionality to the standardized language support library.

The present proposal defines support for

- Address resolution.
- Stream communication.
- Datagram communication.

The design is based on RAII for resource handling classes like network connections and value semantics for non-resource holding classes like network addresses.

1.1 Example code

The following is an example of a streaming server implemented using the present proposal. It takes whitespace-separated words as input and writes them backwards to its output. The example here is single-threaded but can easily be extended to multiple server threads once threading becomes available in C++.

```
void reverse_server(const string& port) {

  acceptor a(port);
  clog << "Reverse server listening on port "
       << port << endl;

  while(1) {
    connection c(a);
    clog << "Connection from: "
         << c.peer().host() << endl;
    nstream ns(c);
    ns << "500 Welcome to the REVERSE server." << endl;
    string s;
    while(ns >> s) {
      if(s == "quit") {
        ns << "550 Goodbye!" << endl;
        break;
      }
      reverse(s.begin(),s.end());
      ns << s << endl;
    }
    clog << "Connection closed." << endl;
  }
}
```

1.2 Non-goals

- Out of band (OOB) data.
- Joining and leaving multicast groups.
- Support for layers other than the transport layer.

OOB data seems scarcely used, is incompatible with the classic C++ iostream library and [3] suggests a second TCP connection instead.

Joining and leaving multicast groups can be implemented by operating system specific external utilities.

The present proposal only defines support for the transport layer protocols TCP and UDP. There is valid interest to address protocols from other layers in the C++ standard, but we believe this is better done in separate proposals.

1.3 Existing C++ networking frameworks

The web site [1] lists some C++ libraries that include networking functionality, among them Socket++ and wxWindows.

TrollTech's QT library [6] includes networking functionality.

Douglas Schmidt's ADAPTIVE Communication Environment (ACE) [4, 5] is an extensive Object Oriented Programming toolkit including, among others, networking functionality.

For example, a network stream connection can be set up as follows using ACE[1]:

```
const ACE_TCHAR *server_host = "hostname";
u_short server_port = 4711;

ACE_IOStream<ACE_SOCK_Stream> server;
ACE_SOCK_Connector connector;
ACE_INET_Addr addr (server_port,
                    server_host);

if (connector.connect (server, addr) == -1)
  ACE_ERROR_RETURN ((LM_ERROR,
                     "%p\n",
                     "open"),
                    -1);

server << "1 2.3 testing" << endl;

int i;
float f;
```

[1] Example from `iostream_client.cpp` in the ACE distribution

```
ACE_IOStream_String s1;
ACE_IOStream_String s2;

server >> s1 >> i >> f >> s2;
```

The following is a non-exhaustive list of differences between ACE and the present proposal:

- ACE uses return codes instead of exceptions.
- ACE uses the same address type for stream and datagram communications.
- ACE uses a specialized IOStream string class.
- ACE provides support for higher-layer protocols such as SSL.

2 Impact on the Standard

This proposal is a pure extension. It requires one new header file and additions to one existing header file. It does not require changes to any existing standard classes or functions nor to the core language.

Due to the nature of networking, the implementation must use components outside the scope of the C++ standard.

A reference implementation is being developed for two widespread platforms. Source code for experimentation can be obtained from the author.

3 Design Desisions

3.1 Connections, senders, receivers

We define the concept of a *connection* for stream-based protocols and of a *sender* and *receiver* for datagram-based protocols.

To give implementors the maximum flexibility for implementing the proposed interface, we do not attempt to define or use the classical "socket" abstraction in C++. Implementations may actually choose to base networking on sockets, but alternatives like XTI or operating system-specific APIs are just as well possible.

3.2 Transport layer

The transport layer protocols in use today seem to be exclusively TCP and UDP. The stream-based and datagram-based communication defined by the proposal naturally map to these protocols. Still, the proposed standard text does not explicitly mention TCP or UDP. This leaves room for different underlying protocols as long as they use communication endpoints based on the "host" and "port" notion.

3.3 Addresses and address lists

We define a value-semantics abstraction of network addresses to enable storage in the standard containers. This ability seems crucial for a wide variety of applications.

Since more than one address may be associated with a given host/port pair, communication class constructors actually accept lists of addresses where appropriate. The implementation is then free to chose any appropriate pair of addresses in order to establish communication.

We chose to use different address types for stream and datagram addresses for maximum type safety. Indeed, threre are protocols where the port number vs. port name mapping is different for on TCP or UDP, cf. [2], Section 6.5.

IPv4 and IPv6 addresses are handled transparently by the implementation, i.e. application code is independent of which IP protocol version is used.

No assumption is made about the internal representation of addresses. The `host()` and `port()` observers should return the host and port part in some standard notation for the underlying address type, e.g. dotted-decimal for IPv4 addresses and decimal for port numbers.

3.4 Waiting, timeout and non-blocking I/O

We define non-blocking I/O for datagram communications, where sending is non-blocking by its nature and a timeout can be given for reception.

For single-threaded applications handling multiple stream connections, the `iowait()` multiplexing function is defined.

3.5 Support only for character streams

All networking protocols appear to be character- or octet-based, so there seems no need to templatize the communication classes on the character type.

3.6 Error reporting

Many network errors are transient by nature and applications may wish to try an operation again if an error occurs. Examples include DNS address resolving where DNS is temporarily unavailable. The library will throw a `transient_error` exception in these cases.

If the problem is likely to be a permanent error, the library will throw a `permanent_error` exception. Examples include DNS being unable to resolve a hostname or a nonexistant port name.

3.7 Numeric ports

We decided not to define address resolver functions with numeric port values as arguments. Numeric port values can be given as strings. Adding scalar arguments would lead to combinatorial explosion of an already high number of overloads. The performance impact is expected to be minimal or even beneficial

since most applications get their port numbers in form of strings anyway, e.g. on the command line or in configuration files.

3.8 Use of `double` for time values

We intend to enable fractional time values for operations that might time out. Many operating systems or libraries define their own structures to represent fractional time values. However, a simple `double` variable easily serves the purpose and it therefore seems unnecessary to introduce a new type. Even `float` precision will be enough in most cases, but we don't think using `double` incurs significant overhead over using `float`.

4 Proposed Text for the Standard

4.1 Header <network> synopsis

```
namespace tr2 {
  // Network addressing.
  typedef (implementation defined) stream_address;
  typedef (implementation defined) datagram_address;

  // Container of addresses.
  typedef vector<stream_address> stream_address_list;
  typedef vector<datagram_address> datagram_address_list;

  // Resolve host/port names to addresses.
  const stream_address_list resolve_stream(const string& host,
                                           const string& port);
  const datagram_address_list resolve_datagram(const string& host,
                                               const string& port);
  const stream_address_list resolve_stream(const string& port);
  const datagram_address_list resolve_datagram(const string& port);

  // Network connections.
  class acceptor;
  class connection;

  // Datagram sender and receiver.
  class datagram_sender;
  class datagram_receiver;

  // IOStream interface.
  typedef (implementation defined) instreambuf;
  typedef (implementation defined) onstreambuf;
  class nstreambuf: public instreambuf , public onstreambuf;
```

```
class instream: public istream;
class onstream: public ostream;
class nstream: public iostream;

// I/O multiplexing.
template < class InputIteratorA, class InputIteratorB,
           class     ContainerA, class     ContainerB >
bool iowait(const InputIteratorA& abeg,
            const InputIteratorA& aend,
            const InputIteratorB& bbeg,
            const InputIteratorB& bend,
                  ContainerA& aready,
                  ContainerB& iready,
                  ContainerB& oready,
            const double& to = -1);
```

4.2 The `address` concept

An *address* is an endpoint of network communications. Two addresses are necessary in any type of network communications: A *local* address on the host on which the program is executing and a *remote* address on any host to which a network route exists. In *stream-oriented* communications, we speak of the two *endpoints* of the network connection. In *connection-less* or *datagram oriented* communication, we speak of the source and destination address of a datagram.

An address consists of a *host* part identifying a host or a network interface of a host and a *port* part identifying a specific network port.

Both the host and the port part can be defined by a name or in numeric form. The resolve family of functions use systems like DNS or port databases to translate host and port names into addresses and vice versa.

A *wildcard* address is a special type of address to be used for local endpoints only. It signals the system that it is free to choose the local host part, port part, or both.

[*Note:* For the rest of this section, `address` refers to `stream_address` and `datagram_address` so that actually two address types are defined.]

```
class address {
  // Construct
  // Implementation-defined default address.
  address();

  // observers
  // Get host and port for this address.
  const string host() const;
  const string port() const;
```

```
    // Get host and port name for this address.
    const string host_name() const;
    const string port_name() const;

    // Get FQDN for host.
    const string host_fqdn() const;
  };
```

4.2.1 `address` constructors

```
address();
```

Effects: Constructs an implementation-defined default address.
[*Note:* The resulting address object is only required to support the `host()` and `port()` member functions.]

4.2.2 `address` observers

```
const string host() const;
```

Returns: The host part of the address in numeric form.

```
const string port() const;
```

Returns: The port part of the address in numeric form.

```
const string host_name() const;
```

Returns: The host part of the address as an unqualified host name.
Throws: `transient_error` or `permanent_error` if a host name cannot be obtained.

```
const string port_name() const;
```

Returns: The port part of the address as a service name.
Throws: `transient_error` or `permanent_error` if a service name cannot be obtained.

```
const string host_fqdn() const;
```

Returns: The host name as Fully Qualified Domain Name (FQDN).
Throws: `transient_error` or `permanent_error` if the FQDN cannot be obtained.

4.3 Resolve functions

```
const stream_address_list resolve_stream(const string& port);
```

Returns: A list of wildcard stream addresses suitable for use as the `la` argument to an `acceptor` or `connection` constructor.

Throws: `permanent_error` if the port name cannot be resolved.

```
const stream_address_list resolve_stream(const string& host,
                                         const string& port);
```

Returns: A list of stream addresses suitable for use as the `ra` argument to a `connection` constructor.

Throws: `permanent_error` or `transient_error` if either the host or the port name cannot be resolved.

```
const datagram_address_list resolve_datagram(const string& port);
```

Returns: A list of wildcard datagram addresses suitable for use as the `la` argument to an `datagram_sender` or `datagram_receiver` constructor.

Throws: `permanent_error` if the port name cannot be resolved.

```
const datagram_address_list resolve_datagram(const string& host,
                                             const string& port);
```

Returns: A list of datagram addresses each of which is suitable for use as the `ra` argument to the `send()` method of class `datagram_sender`.

Throws: `permanent_error` or `transient_error` if either the host or the port name cannot be resolved.

4.4 Class `acceptor`

```
class acceptor {
  typedef stream_address address_type;
  typedef stream_address_list address_list_type;

  acceptor(const string& ls,
           unsigned backlog = (implementation defined));
  acceptor(const address_list_type& la,
           unsigned backlog = (implementation defined));

  // Observers
  const address_type& local();
};
```

An `acceptor` manages a queue of inbound stream connection requests to a local address given in its constructor. Connections are not established until a `connection` object is constructed with the `acceptor` as its constructor argument. Inbound connection requests which are not yet established are called *pending*.

4.4.1 `acceptor` constructors

```
acceptor(const address_list_type& la,
         unsigned backlog = (implementation defined));
```

Effects: The implementation chooses an appropriate address from `la`, allows a maximum of `backlog` pending inbound connection requests at the chosen address and makes them available for `connection` constructors with this `acceptor` as its argument. If at least `backlog` inbound connections are pending, further connection requests by clients will be refused.

Throws: `permanent_error` if none of the addresses in `la` can be used as a local connection endpoint.

Precondition: `la.size()` is at least 1.

Postcondition: `local()` returns the chosen local address.

```
acceptor(const string& ls,
         unsigned backlog = (implementation defined));
```

Effects: Equivalent to `acceptor(resolve_stream(ls), backlog)`

4.4.2 `acceptor` observers

```
const address_type& local();
```

Effects: Returns the local address chosen by the constructor.

4.5 Class `connection`

```
class connection
{
public:
  typedef stream_address address_type;
  typedef stream_address_list address_list_type;
  // Construct
  connection(const address_list_type& ra,
             const address_list_type& la=address_list_type());
  connection(const string& host, const string& port);

  connection(acceptor&);

  // Destroy
  ~connection();

  // modifiers
  void no_delay(bool=true);
  // observers
  address_type const& peer() const;
  address_type const& local() const;
};
```

A `connection` represents the program's view of a bidirectional stream-oriented network connection. It has two endpoint addresses, a *local* one (on the computer where the program is executed) and a *remote* one on this or any other computer.

4.5.1 `connection` constructors

```
connection(const address_list_type& ra,
           const address_list_type& la=address_list_type());
```

Effects: Attempts to establish a connection to a remote address in `ra` from a local address in `la`. If `la` is empty, a suitable local address is chosen by the implementation.

If the request becomes pending at the remote address, waits until the connection becomes established.

Throws: `transient_error` or `permanent_error` if the connection cannot be established.

Precondition: `ra.size()` is ≥ 1.

Postcondition: A stream-oriented network connection is established between `la` and `ra`. `local()` and `peer()` return the respective endpoint addresses.

[*Note:* The implementation shall try all suitable combinations from `la` and `ra` in the attempt to establish a connection. It shall fail only if none of the attempts succeeds, in this case throwing an exception related to the last attempt made.]

```
connection(const string& host, const string& port);
```

Effects: Equivalent to `connection(resolve_stream(host,port))`.

```
connection(acceptor& a);
```

Effects: If a connection request is pending at `a`, establish a connection with the first request in `a`'s queue and return. Otherwise, wait until a request is pending, establish the connection with the requesting peer and return.

Postcondition: `local() == a.local()`. A stream-oriented connection is established between `local()` and `peer()`.

4.5.2 `connection` destructor

```
~connection();
```

Effects: Closes the connection and destroys the `connection` object.

4.5.3 `connection` modifiers

```
void no_delay(bool=true);
```

Effects: When set to `true`, try to reduce the delay between data sent and data being received.

[*Note:* This may be useful for interactive applications over a connection with a high round trip time. Typically, this is implemented by disabling the so-called *Nagle algorithm* of TCP, cf. [3].]

4.5.4 `connection` observers

```
const address_type& peer() const;
```

Returns: The remote connection endpoint.

```
const address_type& local() const;
```

Returns: The local connection endpoint.

[*Note:* The address returned by *local()* on this computer may differ from the address returned by *peer()* on the remote computer.]

4.6 Class `datagram_receiver`

```
class datagram_receiver {
public:
  typedef datagram_address address_type;
  typedef datagram_address_list address_list_type;

  typedef (implementation defined) size_type;

  datagram_receiver(const string& ls);
  datagram_receiver(const address_list_type& la);

  // Receive.  begin must indicate beginning of large enough area.
  // Returns timeout() on timeout.
  template<class for_it>
  size_type receive(const for_it& begin,
                    const double& timeout = -1,
                    size_type n = default_size());

  // Constants.
  static size_type timeout();
  static size_type default_size();

  // observers
  // Source of last received packet.
  const address_type& source() const;
  // Local address.
  const address_type& local() const;
};
```

4.6.1 `datagram_receiver` constructors

```
datagram_receiver(const address_list_type& la);
```

Effects: Attempts to establish an endpoint for receiving datagrams at one of the local addresses in `la`.

Throws: `permanent_error` if the endpoint cannot be established at any of the given addresses.

Precondition: `la.size()` is ≥ 1.

Postcondition: `local()` returns one of the elements of `la`. Datagrams can be received at `local()`.

```
datagram_receiver(const string& ls);
```

Effects: Equivalent to `datagram_receiver(resolve_datagram(ls));`

4.6.2 `datagram_receiver` typedefs

```
typedef (implementation defined) size_type;
```

Represents: An unsigned integral type wide enough to hold all possible datagram sizes and `timeout()`.

4.6.3 `datagram_receiver` static members

```
static size_type timeout();
```

Returns: A value used to indicate receive timeout that cannot be the size of a datagram.

```
static size_type default_size();
```

Returns: An upper bound on the size of datagrams that can be received on the implementation.

4.6.4 `datagram_receiver` receive function template

```
template<class for_it>
size_type receive(for_it begin,
                  const double& timeout = -1,
                  size_type n = default_size());
```

Effects: Waits for a packet to be received. If no packet arrives within `timeout` seconds, returns `timeout()`. Otherwise, writes the contents of the received packet to the memory location pointed to by `begin`. If the received datagram contains more than `n` characters, only `n` are written.

Returns: The number of characters written, i.e. the maximum of the `n` and the size of the received packet, or `timeout()`.

Requires: `for_it` is a Forward Iterator with value type `char`.

4.6.5 `datagram_receiver` observers

```
const address_type& source() const;
```

Returns: The source address of the last received datagram.

```
const address_type& local() const;
```

Returns: The local address on which datagrams are received.

4.7 Class `datagram_sender`

```
class datagram_sender {
public:
  typedef datagram_address address_type;
  typedef datagram_address_list address_list_type;

  datagram_sender();
  datagram_sender(const string& ls);
  datagram_sender(const address_list_type& la);

  // Send data.
  template<class for_it>
  void send(const address_type& ra,
            for_it begin, for_it end);

  // Local address.  Undefined if default-constructed.
  const address_type& local() const;
};
```

4.7.1 `datagram_sender` constructors

```
datagram_sender();
```

Effects: Constructs a `datagram_sender` object. Subsequent calls to `send()` will use a local address determined by the system as a local address.

```
datagram_sender(const string& ls);
```

Effects: Equivalent to `datagram_sender(resolve_datagram(ls))`.

```
datagram_sender(const address_list_type& la);
```

Effects: Constructs a `datagram_sender` object.
Postcondition: `local()` returns one of the elements of `la`.
Throws: `permanent_error` if none of the addresses in `la` is useable as a local address for sending datagrams.

4.7.2 `datagram_sender` observers

```
const address_type& local() const;
```

Returns: The local address used for sending datagrams.
Throws: `permanent_error` if the `datagram_sender` object was default constructed.

4.7.3 `datagram_sender` modifiers

```
template<class for_it>
void send(const address_type& ra,
          for_it begin, for_it end);
```

Effects: Sends the data in `[begin,end)` to `ra`. Uses one of the local addresses given in the constructor or an implementation-defined address if default constructed.
Requires: `for_it` is a Forward Iterator with value type `char`.
Throws: `permanent_error` if

- `ra` is unsuitable for sending the datagram in conjunction with the local address or
- the range `[begin,end)` is too large to fit in a datagram.

[*Note:* Datagram communication is best-effort. No indication is available if the datagram was successfully delivered to its destination.]

4.8 Stream buffer classes

Three classes, `instreambuf`, `onstreambuf` and `nstreambuf`, are defined to associate input and output character sequences with a stream-based network connection. No seek operations are supported.
The classes may be implemented as template specializations.
`streambuf` is a public virtual base class for `instreambuf` and `ostreambuf`.
The class `nstreambuf` is derived from `instreambuf` and `onstreambuf`.

4.9 Class `instreambuf`

```
class instreambuf {
public:
  // Construct
  instreambuf(connection& c,
              int size = (implementation defined),
              int size_pb = (implementation defined));
protected:
  virtual int_type underflow();
private:  // Expositon only
```

```
  connection& c_;
  vector<char> buffer;
};
```

The class `instreambuf` associates the input sequence with a network connection.

4.9.1 `instreambuf` constructor

```
instreambuf(connection& c,
            int size = (implementation defined),
            int size_pb = (implementation defined));
```

Effects: Constructs an object of class instreambuf, initializing the base class with `streambuf()` and `c_` with `c`. Then creates a buffer of size `size + size_pb`.

Postcondition: Characters in the input sequence will be read from the network connection `c`.

Requires: `size` ≥ 1 and `size_pb` ≥ 1.

Throws: `out_of_range` if an argument is out of range.

[*Note:* `size_pb` is the size of the putback area.

During the lifetime of a `connection` object `c`, at most one `instreambuf` object may be constructed on `c`.]

4.9.2 Overridden virtual functions

```
virtual int_type underflow();
```

Effects: If the input sequence has a read position available, returns `traits::to_int_type(*gptr())`. Otherwise, returns `traits::eof()`.

4.10 Class `onstreambuf`

```
class onstreambuf {
public:
  // Construct
  onstreambuf(connection& c, int size = (implementation defined));
  // Destroy
  virtual ~ostreambuf();
protected:
  virtual int_type overflow(int_type c);
  virtual int sync();
private:  // Expositon only
  connection& c_;
  vector<char> buffer;
};
```

The class `onstreambuf` associates the output sequence with a network connection.

4.10.1 `onstreambuf` constructor

```
onstreambuf(connection& c, int size = (implementation defined));
```

Effects: Constructs an object of class onstreambuf, initializing the base class with `streambuf()` and `c_` with `c`. Then creates a buffer of size `size`.

Postcondition: Characters in the output sequence will be written to the network connection `c`.

Requires: `size` ≥ 1.

Throws: `out_of_range` if an argument is out of range.

[*Note:* During the lifetime of a `connection` object `c`, at most one `onstreambuf` object may be constructed on `c`.]

4.10.2 `onstreambuf` destructor

```
virtual ~onstreambuf();
```

Effects: Calls `sync()`.

[*Note:* As the destructor cannot return a value nor throw an exception, programs wishing to check if all data was written to the controlled sequence should call `sync()` before destroying the onstreambuf object.]

4.10.3 Overridden virtual functions

```
int_type overflow(int_type c);
```

Effects: Appends the character designated by `c` to the output sequence, if possible, in one of two ways:

- If `traits::eq_int_type(c,traits::eof())` returns false and if either the output sequence has a write position available or the function makes a write position available (as described below), the function calls `sputc(c)`. Signals success by returning `c`.
- If `traits::eq_int_type(c,traits::eof())` returns true, there is no character to append. Signals success by returning a value other than `traits::eof()`.

Notes: The function can make a write position available by adjusting the put pointer by calling `pbumb()`.

Returns: `traits::eof()` to indicate failure.

```
virtual int sync();
```

Effects: Synchronizes the controlled sequences with the buffer. That is, if `pbase()` is non-null the characters between `pbase()` and `pptr()` are written to the controlled sequence. The pointers are then reset as appropriate.

Returns: Zero on success and -1 if writing to the controlled sequence fails.

4.11 Class `nstreambuf`

```
class nstreambuf : public instreambuf, public onstreambuf {
public:
  // Construct
  nstreambuf(connection& c, int size = (implementation defined));
             int size_in = (implementation defined)
             int size_pb = (implementation defined)
             int size_out = (implementation defined));
};
```

The class `nstreambuf` associates the input and the output sequence with a network connection.

4.11.1 `nstreambuf` constructor

```
nstreambuf(connection& c,
           int size_in = (implementation defined)
           int size_pb = (implementation defined)
           int size_out = (implementation defined));
```

Effects: Constructs an object of class onstreambuf, initializing the `instreambuf` base class with `instreambuf(c,size_in,size_pb)` and the `onstreambuf` base class with `onstreambuf(c,size_out)`.

4.12 Stream classes

The class `onstream` is defined as an `ostream` specialization to extract data from the network stream from the local to the remote computer. The class `instream` is defined as an `istream` specialization to extract data from the network stream from the local to the remote computer. The class `nstream` is defined as a `iostream` specialization to extract data from and insert data into the network stream from the local to the remote computer.

4.13 Class `instream`

```
class instream : public istream {
public:
  // Construct
  instream(connection&);
private:
  instreambuf sb;  // exposition only
};
```

The class `instream` supports reading from an established network connection. It uses an `instreambuf` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

- `sb`, the instreambuf object.

4.13.1 `instream` constructor

```
instream(connection& c);
```

Effects: Constructs an object of class instream initializing `sb` with `c`, and initializing the base class to use `sb` as its buffer.

Postcondition: `rdbuf()` returns `&sb`.

4.14 Class `onstream`

```
class onstream : public ostream {
public:
  // Construct
  onstream(connection&);
private:
  onstreambuf sb;  // exposition only
};
```

The class onstream supports writing to an established network connection. It uses an onstreambuf object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

- `sb`, the onstreambuf object.

[*Note:* Applications should carefully check an `onstream`'s state, since write errors on network connections are much more common than for other types of `ostream`s.]

4.14.1 `onstream` constructor

```
onstream(connection& c);
```

Effects: Constructs an object of class onstream initializing `sb` with `c`, and initializing the base class to use `sb` as its buffer.

Postcondition: `rdbuf()` returns `&sb`.

4.15 Class `nstream`

```
class nstream : public iostream {
public:
  // Construct
  nstream(connection&);
private:
  nstreambuf sb;  // exposition only
};
```

The class `nstream` supports reading from and writing to an established network connection. It uses an `nstreambuf` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

- `sb`, the `nstreambuf` object.

4.15.1 `nstream` constructor

```
nstream(connection& c);
```

Effects: Constructs an object of class `nstream` initializing `sb` with `c`, and initializing the base class to use `sb` as its buffer.

Postcondition: `rdbuf()` returns `&sb`.

4.16 I/O multiplexing

```
template < class InputIteratorA, class InputIteratorB,
           class     ContainerA, class     ContainerB >
bool iowait(const InputIteratorA& abeg,
            const InputIteratorA& aend,
            const InputIteratorB& bbeg,
            const InputIteratorB& bend,
                  ContainerA& aready,
                  ContainerB& iready,
                  ContainerB& oready,
            const double& to = -1);
```

Requires:

- `InputIteratorA` is an input iterator type whose value type dereferences to `acceptor`.
- `InputIteratorB` is an input iterator type whose value type dereferences to `streambuf`.
- `ContainerA` is a container type supporting `push_back()` and whose value type is `InputIteratorA`.
- `ContainerB` is a container type supporting `push_back()` and whose value type is `InputIteratorB`.

Effects: Blocks the program until either of the postconditions below can be satisfied or the timeout period of `to` seconds expires, whichever occurs earlier. Uses `push_back()` to append iterators referring to acceptors or stream buffers to `aready`, `iready` and `oready`.

Postconditions:

- For all iterators `i` appended to `aready`, the constructor call `connection(**i)` shall not block.
- For all iterators `i` appended to `iready`, the call `(**i).sgetc()` shall not block.
- For all iterators `i` appended to `oready`, the call `(**i).sputc(c)` shall not block.

Returns: `true` if and only if at least one output container was written to.

Note: A negative value for `to` causes the function never to time out. A value of zero may be used for polling.

4.17 Additions to header <stdexcept>

```
namespace tr2 {
  class network_error;
    class transient_error;
    class permanent_error;
}
```

4.18 Class `network_error`

```
  class network_error : public runtime_error {
  public:
    explicit network_error(const string& what_arg);
  };
```

The class `network_error` defines the type of objects thrown as exceptions to report errors occuring on networking operations.

```
  network_error(const string& what_arg);
```

Effects: Constructs an object of class `network_error`.
Postcondition: `strcmp(what(), what_arg.c_str()) == 0`.

4.19 Class `transient_error`

```
  class transient_error : public network_error {
  public:
    explicit transient_error(const string& what_arg);
  };
```

The class `transient_error` defines the type of objects thrown as exceptions to report transient network errors.

[*Note:* Transient network errors are errors that are likely to go away if the operation is retried at a later stage.]

```
  transient_error(const string& what_arg);
```

Effects: Constructs an object of class `transient_error`.
Postcondition: `strcmp(what(), what_arg.c_str()) == 0`.

4.20 Class `permanent_error`

```
class permanent_error : public network_error {
public:
  explicit permanent_error(const string& what_arg);
};
```

The class `permanent_error` defines the type of objects thrown as exceptions to report permanent network errors.

[*Note:* Permanent network errors are errors that are likely to persist even if the operation is retried at a later stage.]

```
permanent_error(const string& what_arg);
```

Effects: Constructs an object of class `permanent_error`.
Postcondition: `strcmp(what(), what_arg.c_str()) == 0`.

5 Unresolved Issues

1. Discuss a "performance preferences" setting for TCP connections, similar to Java's setPerformancePreferences(). This would generalize `no_delay()`.
2. Discuss `iowait()` semantics. Should we include `datagram_receiver`s as well?
3. Discuss istream extractors/ostream inserters for address types? If yes, which address format to choose? Should we have ostream manipulators for address formatting?
4. Local addresses should be reusable by default, check [3] for the exact conditions.
5. Standardese for `stream_address` and `datagram_address` is not yet quite standard.
6. This section should be removed.

6 Revision History

Differences from first draft:

- Added `iowait()`.
- Some naming changes for `address`es.

7 Acknowledgements

I'd like to thank Matt Austern and Benjamin Koznik for motivation, discussions and constructive input. Isabel Drost kindly proofread the proposal with meticulous accuracy.

References

[1] Available C++ libraries FAQ. `http://www.trumphurst.com/cpplibs/cpplibs.phtml`.

[2] R. Gilligan et al. RFC 2553—Basic socket interface extensions for IPv6. `http://www.faqs.org/rfcs/rfc2553.html`, 1999.

[3] W. Richard Stevens et al. *UNIX Network Programming*, volume 1. Addison-Wesley Professional, third edition, 2003.

[4] Douglas C. Schmidt. The ADAPTIVE Communication Environment (ACE(TM)). `http://www.cs.wustl.edu/~schmidt/ACE.html`.

[5] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming*, volume 2 of *Systematic Reuse with ACE and Frameworks* . Addison-Wesley Professional, 1st edition, 2002.

[6] TrollTech homepage. `http://www.trolltech.com/`.