Netcomm library

Overview

This library is a C++ layer built above Windows Sockets (TCP sockets particularly), but Windows Sockets is not the only transport layer that can be utilized for data transmission. Other implementations (based say on UDP) can be provided.

Moreover above the simple transport layer, which provides abstracted streaming of data, a messenger service is implemented which provides generalized messaging functionality.

Finally several specialized messages are implemented and a service for sending/receiving these messages.

Terminology

* Address – common address string shared between connecting parties
* Binding – can be server binding or client binding. Correspondingly it is a server listening on an address or a client connecting to this address.
* Stream – Transmission of low level data as basic as bytes. Streams do not have any predefined length, but streams are reliable as TCP connections which means UDP-based stream implementation should implement some internal ordering of packets.
* Message – messages are similar to streams and are built on top of streams except that they do have predefined length and type. Messages of a similar type are not obligated to have similar size, each message can have its own size, but the size and type are known before a message is transmitted.
* Endpoint – some high level abstraction of a peer. Not really a part of a shared library, rather a particular implementation used in the *svc* component.
* Delegate – used in source code as an alias for event sinks. Basically an interface implementation assigned to a corresponding object in order to listen for events from this objects.

Source Code

Source code is a Visual Studio 2008 solution which contains several projects:

* *interfaces* – a collection of interfaces that need to be implemented in order to extend the library, there are also some delegates in *net* and *msg* projects. Probably they should be moved here, but they are tightly coupled with objects they service for, thus are defined in corresponding projects.
* *net* – implementation of the *net* component
* *msg* – implementation of the *msg* components
* *svc* – sample service utilizing *net* and *msg* components
* *client* – client utilizing the service
* *server* – server utilizing the service
* *util* – collection of utility classes, most of them should be substituted for more efficient implementations like *boost::shared\_ptr*, or *boost::scoped\_array*. Also *ThreadMutex* and *ScopedLock* are good candidates for substitution by *ACE*, *boost* or even *MFC* classes.
* *netcomm* – unit tests for the library

Components

There are three main components:

* *net* – implementation of raw data streaming
* *msg* – messaging service
* *svc* – an application-specific service utilizing functionality of the *net* and *msg* components

Each is built a top of a previous one.

net

Main interfaces:

* *IBinding* – Two implementation must be provided: for server and for the client binding. Server implementation in its *bind()* implementation must listen for incoming connections and actually provide some acceptor-connector pattern implementation. Client implementation must connect to the address provided. Both implementations must create *IStream* instance and notify of its creation to an *IBindingDelegate* provided. After a binding is established and a stream is created there's no difference between server and client side. All library's functionality disregards server vs client difference.
* *IBindingDelegate* – this interface must be implemented by a class listening for new connections. Once a connection is established (after server and client *IBinding* implementations successfully *bind()* to the same address), a method of this interface are called to notify the listening party of the new connection. This is the best place to put a newly created stream under *StreamListener*'s control (see further).
* *IStream –* implementation is responsible for sending and receiving bytes over an established connection. The receiving part is non-blocking so if no data are available implementation must not wait for incoming data, rather should return zero. Note, this differs from semantics of Windows Socket's *recv* function.

Main classes:

* *TcpServer* – server-side implementation of the *IBinding* interface.
* *TcpClient* – client-side implementation of the *IBinding* interface.
* *TcpStream* – implements *IStream* data sendning and receiving methods over a TCP connection.
* *BindingFactory* – factory class to create bindings. Since only TCP connection is utilized this factory is not really required for this implementation, but can be required if a library is extended, say, the way that bindings are established above bindings of other types. Or in case dynamic binding type is determined at runtime (e.g. a user chooses between a TCP binding or some other custom binding).
* *IStreamListenerDelegate* - *StreamListener* event handler
* *StreamListener* – very important class. Despite of the long description provided below its usage is pretty straitforward. If there's a stream and a class interested in receiving events about this stream, the class must implement *IStreamListenerDelegate* and register itself with a corresponding stream within *StreamListener*. Once registered, use *StreamListener::writeStream()* method instead of direct *Istream::write()*. But devil is in details, thus detail follow.

StreamListener

*StreamListener* is not intended to be sub-classed. Rather it is a class that listens for registered streams and reacts to incoming data by firing events to registered delegates. It is a singleton, which means there's only one instance of this class for a module (exe or dll).

It is a handy class to avoid going into a manual data listening loop both on server and client side. *StreamListener* utilizes several threads to listen for incoming data. Number of threads is limited to number of processors (cores) on a system. Threads access streams in round-robin fashion by checking streams one by one and if data are available for some stream these data are pulled from that stream and an event is fired to a corresponding delegate.

***Error handling with StreamListener***

There are constraints on stream usage imposed by possible errors and appropriate handling of these errors in regards to *StreamListener*. Since errors can encounter while receiving data, *StreamListener* is also responsible for notifying delegates about stream's death. If a stream is disconnected an event is fired to the same delegate listening for incoming data events. Since stream's death can be detected not only during data receiving but also while sending data, *StreamListener* provides a special method for writing to a stream *writeStream()* which allows *StreamListener* to be aware of stream's death and be able to notify the same delegate. Thus any streams registered within *StreamListener* must send data by *StreamListener::writeStream()* method.

***Registration of delegates***

Delegates for stream events must implement *IStreamListenerDelegate* interface methods. A delegate interested in event for a particular stream calls *StreamListener::addDelegate()* method which 1) registers a stream within *StreamListener*,2) registers a delegate listening for events for the specified stream.

Streams and delegates can be registered within *StreamListener:*

* before *StreamListener::run()* is called
* after it has stopped running
* while it is running

but cannot be registered between *cancelRun()* is called and before *run()* has completed execution. Basically since all delegates are unregistered and streams are destroyed when *StreamListener* stops, there's no point in adding new delegates at this moment.

***StreamListener delegates lifetime***

*IStreamListenerDelegate* implementations must be alive all the time they are a subject to event notification. All events to delegates are fired while *StreamListener* is running thus delegates must be alive until *StreamListener::run()* stops execution. *StreamListener::writeStream()* can also detect stream's death but all delegates are unregistered after *StreamListener* is stopped, thus there's no one to notify about stream's death. So if *StreamListener* is stopped, do not send data through it. But it is OK to send data after *StreamListener* has a delegate registered for the corresponding stream.

*StreamListener* starts listening for incoming data when its *run()* method is called. This method is blocking, it does not exit until *StreamListener* stops all threads listening for incoming data. In order to stop listening *cancelRun()* method should be called. Obviously from some other thread, say from one of the delegate event handlers or from a UI thread if *run()* is not executed on the UI thread (otherwise it would be a bad idea since a UI thread would be blocked).

*cancelRun()* notifies *StreamListener* that it should be stopped, but it might take time to stop. In order to wait for *StreamListener* to actually stop *joinRun()* method should be called. It blocks until *StreamListener* is stopped. Grounding for such separation is that a decision that *StreamListener* should stop can be made withing of *IStreamListenerDelegate* methods, thus *cancelRun()* could be called from within delegate's method which in turn is executed from the *StreamListener*. Thus if *StreamListener* waits for a delegate's method to finish, while the method is waiting for the *StreamListener* to stop, it causes a deadlock straight away.

Binding lifetime

Streams are not owned by bindings, streams are owned and their lifetime is managed by *StreamListener.* Bindings are also not owned by streams. This design decision is questionable, probably streams should own their bindings, but currently they don't. If a binding is destroyed a stream dies. Thus a client and a server must keep corresponding bindings alive until a stream dies.

msg

Main interfaces:

* IMessage – implementation of a message. Every message has a type. Every implementation must be able to serialize its data into a stream and to deserialize its data from a stream.
* *IBindingDelegate* – handles stream creation event. Despite it is named the same way as an interface in the *net* component, and is in fact to be implemented to handle the same stream creation event, it differs in its semantics because implementation is not required to put created stream under *StreamListener*'s control, rather it is a good place to register *ImessengerDelegate*-s for a newly created streams.
* *IMessengerDelegate* – similar to *IStreamListenerDelegate* in regards to semantics of events received, but it instead of row data it receives notifications of high-level events.
* *IMessageFactory* – implementation of this interface must be provided in order to create messages by their types.

Main classes:

* *Messenger* – supplied with a *IMessageFactory* implementation and a *IBindingDelegate* implementation it handles messages transmitted over the streams and notifies registered delegates.

*Messenger* is similar to *StreamListener*, but unlike the latter it works with messages as opposed to raw stream data. Messages are high-level abstractions of application specific data sent over network.

When sending messages over streams *Messenger* uses a message header which is 8 bytes long. First 4 bytes are reserved for message type, second 4 bytes – are for message payload length. Thus message payload length cannot exceed 4Gb.

svc

Actually this is a sample implementation of interfaces described above and utilization of *StreamListener* and *Messenger* classes.

Main interfaces:

* *IserviceDelegate* – client and server implement this interface to get notifications about newly connected endpoints or disconnected endpoints and also directory listing responses.

Main classes:

* *SvcMsgFactory* – message factory for service-specific messages (*IMessageFactory* implementation). Creates messages supported by the service.
* *MessageIdentity* – a message that is sent when a connection is established. It contains endpoint id. The message is sent to notify the other connection party about endpoint identity.
* *MessageRequestDir* – a message that is sent to an endpoint to request a directory contents
* *MessageResponseDir* – a response message that contains a listing of files in a directory
* *Service* – this class should be made a singleton in a real world scenario, but for simplicity is left as is. It provides even higher level of abstraction by providing specialized methods and notifications for asynchronous directory listing requests. This class utilizes the full stack including bindings, *StreamListener* and *Messenger* and can be used as an example of application service implementations.