Subject: Re: SFTP or full SSH2 support for U++?
Posted by Oblivion on Sun, 02 Jul 2017 19:14:59 GMT
View Forum Message <> Reply to Message

Hello Koldo and Tom,

Thank you both for testing the code, an dproviding me the crucial feedback!

Quote:- SFtp::Init(), Stop(), and Ssh.WhenWait() do not exist.

Yes, they are removed in favour of a better approach.
I believe initialization and deinitializaton should be made automatic, analogous to RAII.
If you request a resource, say a Scp channel, which should be allocated per request anyway, you simply spawn it and bind it to an ssh session.
It will be automatically deallocated when the object is destroyed, as is with any C++ object:

```
//
Scp scp1;
scp1.Session(session);                    // Scp channel is "scheduled" to be initialized. There is no need for manual deinitialization. It will be deinitialized at the end of the code block.
//Or
Scp scp2(session);                        // Scp channel is "scheduled" to be initialized. There is no need for manual deinitialization. It will be deinitialized at the end of the code block.
```

In the same vein, you don't actually need to explicitly disconnect from the ssh server. (Of course you can, but it is by no means necessary).
Ssh objects automatically attempt to disconnect when they are being destroyed.

WhenWait was a member of JobQueue. But there is no need for WhenWait anymore.
As you can see in the SshAsyncRequests example that I provided with the package, you can use SocketWaitEvent where it is useful.
If you want to include it in a synchronous call, you can override JobQueue::Execute(). It is virtual. :)

As for the old Scp Demo.
Getters and Putters are now in U++ style.

```
 bool Get(Stream& out, const String& path, Gate<int64, int64> progress = Null);
 bool Put(Stream& in, const String& path, long mode, Gate<int64, int64> progress = Null);
```

So the code needs a slight change:

```
Cout() << "Getting file Demo.mo\n";
FileOut fout(AppendFileName(GetDesktopFolder(), "compile_run.sh"));
const char *path = "compile_run.sh";
Scp scp(ssh);
if(!scp.Get(fout, path, [=](int64 total, int64 done) { return false; })) // <--
    Cout() << scp.GetErrorDesc();
fout.Close();
```

Regarding the warnings, I'll try to fix them. Main problem is that Stream return int64 for file size, etc. but libssh2 functions expect size_t.

Hello Tom,

Those warnings are typical and can be suppressed. Usually they are not harmful. But I'll see what I can do.

Quote:I wonder if "Unable to request SFTP subsystem" was to be expected here.

Yes, this is expected. It can happen time to time. But it should only be a rare case. If not, then there might be a problem with initialization.
I'll look into it.

There are two things to note here though:
1) Remote servers are not always reliable. Rebex is fine but response times of it's public test server are somewhat slow.

2) There is a "little" problem with ssh protocol and it's implementations: Ssh session works over a single socket.
   Each ssh subsystem, be it sftp or scp, are actually channels on a single socket. And AFAIK there is no official way to "wait" on channels.
   So, spawning more than one channel, especially when they are of different subsystems, can result in a such error.
   It is also discussed in the libssh2 documents, There is a suggested solution for this by the libssh2 devs, but it is not implemented yet:

libssh2/TODO, lines 80 to 120 reads:

New Transport API
=================

THE PROBLEM

The problem in a nutshell is that when an application opens up multiple
channels over a single session, those are all using the same socket. If the
application is then using select() to wait for traffic (like any sensible app
does) and wants to act on the data when select() tells there is something to

for example read, what does an application do?

With our current API, you have to loop over all the channels and read from
them to see if they have data. This effectively makes blocking reads
impossible. If the app has many channels in a setup like this, it even becomes
slow. (The original API had the libssh2_poll_channel_read() and libssh2_poll()
to somewhat overcome this hurdle, but they too have pretty much the same
problems plus a few others.)

Traffic in the other direction is similarly limited: the app has to try
sending to all channels, even though some of them may very well not accept any
data at that point.

A SOLUTION

I suggest we introduce two new helper functions:

libssh2_transport_read()

- Read "a bunch" of data from the given socket and returns information to the
  app about what channels that are now readable (ie they will not block when
  read from). The function can be called over and over and it will repeatedly
  return info about what channels that are readable at that moment.

libssh2_transport_write()

- Returns information about what channels that are writable, in the sense
  that they have windows set from the remote side that allows data to get
  sent. Writing to one of those channels will not block. Of course, the
  underlying socket may only accept a certain amount of data, so at the first
  short return, nothing more should be attempted to get sent until select()
  (or equivalent) has been used on the master socket again.

I haven't yet figured out a sensible API for how these functions should return
that info, but if we agree on the general principles I guess we can work that
out.

For the time being, best workaround for this problem seems to be increasing the socket "wait"
time and adopting a "max. retries" approach. :)

I am in the process of writing the api docs, and a guide for SSH protocol and package, I'll discuss
these problems there too.

Best regards,

Oblivion