Multiplexing

Updated Jun 19, 2011 by michael....@gmail.com

This package implements a binary protocol for multiplexing read and write streams. The protocol intends to remain very simple, so that it can be implemented in other languages or implementations of Smalltalk if desired.

Multiplexing

Multiplexing is the act of sharing a read/write between multiple green threads; fairly splitting up data based on their process priority. Only certain kinds of streams can be used in this manner; eg: unix domain sockets, tcp sockets. The streams must have the ability to be bi-directional.

Every time the underlying stream is shared, a new "channel" is created, which acts as a substream. It is expected that each channel will run in its own green thread; but it is not necessary to do so. If you are running a multiplexer connected to itself with-in the same image, then you will need to run each multiplexer in its own thread. The first thing the multiplexer does is negotiate the protocol version, which will block on itself.

```
| sockets server client |
sockets := OS.SocketAccessor phonyPair.
[[server := Multiplexer on: sockets first]
        ensure: [sockets first close]] fork.
[[client := Multiplexer on: sockets last]
        ensure: [sockets last close]] fork.
```

Once the multiplexer is established, one side can create a new output channel. This will inform the other side that the output channel exists and it can then fetch its corresponding input channel. Here you can see where separate threads or images is useful, because you cannot fetch the input channel on the 'client' until the server has initiated the action by getting an output channel.

```
| output input |
output := server getOutputChannel.
input := client getInputChannel.
```

An output channel is a non-positionable binary write stream. An input channel is a non-positionable binary read stream. At this point you can communicate any kind of data across the stream you like. You can close the channels substream like normal and it will only close that channel, not the underlying stream. You must close the underlying stream when you are done.

You do not have to close the underlying stream to stop multiplexing on it. Instead, you can close all substreams on it, then tell the multiplexer to #release and it will stop expecting input. This may not always work as expected, as the remote side may still be using the stream with a multiplexer. some sort of negotiation should be applied before performing this technique.

Multiplexer Protocol v0

Future protocols **must** honor older protocols. The first thing that transmits over the stream is a single byte version number. If this implementation is newer than the remote implementation, this implementation should not send commands that the remote implementation will not understand. If the remote implementation is newer than this implementation, it should not send this implementation commands it will not understand.

Once negotiation is complete, the stream has four kinds of packets it will transmit. All tokens are sent as four byte little endian unsigned integers.

- ChannelOpen:

  <0>

  <channel id>

      A new channel has been created by the remote side.

- ChannelClose:

  <1>

  <channel id>

      A channel has been closed by the remote side.

A channel has been closed by the remote side.

- ChannelMore:

  <2>

  <channel id>

  The remote side is requesting more data for a channel.

- ChannelData: <channel id> <packet size> <byte data x packet size>

  The remote side has sent a packet of data for a channel.

Channel id's start at 10, leaving some room for seven other commands without having to modify the ChannelData protocol. However, given the similarity of this protocol to the SSH2 multiplexing protocol, it seems unlikely many more forms of packets will be needed.

The sending side will never send more than 2,000,000 bytes of data to the remote side at a time. Once that quota has been reached or the receiving side has consumed all the data that was sent to it, the remote side will send a ChannelMore packet, indicating it wants more data to be sent. This avoids one particular channel from saturating the underlying stream if it is not being read by the remote side. At most, each reading channel will buffer up to 2,000,000 bytes of unread data.

When the sending side is sending small chunks of data and the receiving side is reading them quickly, you may end up with a protocol chattiness that is undesirable, eg: [-> Send 10 bytes. <- Read 10 bytes. <- Request more. -> Read request more] repeat. This is an undesirable use of the protocol and if this is happening (and creating noticeable latency over a network), buffer up your data before sending it, so that it can be sent in bulk.

```
        output := output buffering: 2000000
```