Full-Event-Based Broker/Channel API

Specifications

Establish a connection

A `Broker` is an object geared to communicate to others and create a communication stream called a `Channel`. Both objects are associated with an event interface that permits the user to design a client or a server application. When the user wants to design a server application that accepts incoming connection requests from clients, it is needed to summon a binding continuous event on a given port with this `Broker`'s method:

```
void Broker::bind(Broker.AcceptListener listener, int port);
```

Before calling this function, it is necessary to create a class that implements `Broker.AcceptListeners` and its unique event `void accepted(Channel c);`. This event will be invoked everytime a new client comes to connect to your binding broker on the same port. The event provides a `Channel` that permits the user to communicate with the client.

It is possible to stop the process of continuously binding on the given port by calling:

```
void Broker::unbind(int port);
```

This method will stop the continuous binding event from accepting next clients.

NOTA BENE: since this method can be called while the binding event is accepting a connecting client, it is going to only tell the binding event to invalidate the in-building channel and not start again anymore to accept other incoming clients. So, it is possible to receive an `accepted` event with a provided channel even after the `unbind` call but this channel would be either `null` or disconnected/dangling (unusable). It is a good practice to test the provided channel in the event `accepted`.

WARNING: if the method is called on a port on which no binding event was accepting, it raises an `IllegalStateException`.

To connect to an other task, the user has to summon a connecting event by calling:

```
void Broker::connect(Broker.ConnectListener listener, String name, int port);
```

Before calling this function, it is necessary to create a class that implements `Broker.ConnectListener` and its both events `void connected(Channel c);` (invoked when the connection is successful by providing the connected channel) and `void refused();` (invoked when the connection attempt has failed).

Send bytes

Once the connection has been established, two channels are provided: one in the `accepted` event from the `Broker.AcceptListener` listener and one in the `connected` event from the `Broker.ConnectListener` listener. In both case, you must create a class that implements `Channel.RWListener` and its events:

- `void received(byte[] msg);`: invoked when you have receive some bytes. `msg` containes those received bytes from the remote channel. You have a total ownership on this array.
- `void sent(int number);` invoked when your sending event has succeed to send a certain amount of bytes. `number` is this amount.
- `void closed();` invoked when your channel or the remote channel has requested to disconnect via `void Channel::disconnect()`.

WARNING: when a channel is disconnected, it is no longer usable and can be deleted. You must establish a new connection with the remote broker to get another usable channel.

With the provided channel, you can call several useful methods:

- `void Channel::setListener(Channel.RWListener listener);`

You must provide a listener that implements `Channel.RWListener` to be able to send and receive data from the remote channel. This method allows you to change the listener at any moment.

WARNING: when a listener is changed while data are written, some bytes of the data can be lost (in case `data.length > SendEvent.MAXFRAME`, which is equal to 1500). To prevent any writing event blocking the event pump, every data is sent by 1500-bytes long frames, which leads big data to be sent by smaller arrays of maximum size 1500. Hopefully, smaller sent data would not be lost in case the listener is changed because the sending operation in the sending event is atomic.

- `void Channel::send(byte[] bytes)`

This method tells the channel's sending event to send those `bytes` to the remote channel as soon as possible. It is not blocking and it releases total ownership of the array as soon as it is called. When the user attempts to send nothing, a `null` array or an empty array, the method detects it and does nothing else aftermath. However, when the user attempts to send bytes while the channel is being disconnected, it raises an `IllegalStateException`.

Test this API

`TestEchoMain.java` is the main test program of this API. You can launch or debug this application to see how `TestEchoServer` exchanges various types of byte arrays with `TestEchoClient`.

Design

Unique worker: event pump

There is an unique thread, launched by the singleton `EventPump`, that stores posted events (`Runnable`s) in a queue and executes them as soon as possible in FIFO.

Auto-calling events

Two types of event, `SendEvent` and `AcceptEvent`, post themselves once they finish the processus while they have not been told to stop (respectively by calling `Channel::disconnect` or `Broker::unbind`).