NIO API Implementation

pedro.benedicteillescas@bsc.es

1 General behavior

1.1 Classes and interfaces TODO

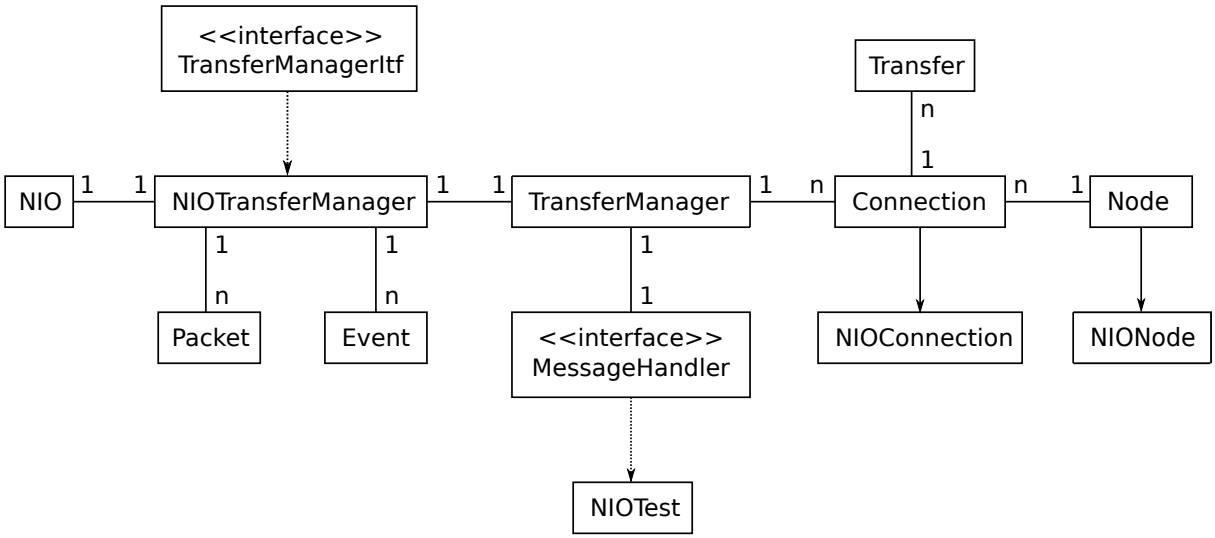

Figure 1: UML Diagram

- **NIO:** Asynchronous network level transfer using the Java NIO library.
- **Transfer:** Represents a transfer between two nodes. It can be reused for consecutive transfers between the same nodes.
- **TransferNIO:** Specific NIO implementation of Transfer.
- **TransferManager:** Manages the various active transfers. It serves as an interface with the NIO layer.
- **MessageHandler:** Interface to implement by the application programmer. Contains the callbacks used by TransferManager.

1.2 Threads

The whole NIO implementation is done using two threads. One thread only runs code from the NIO class, and it checks for incoming connections, send data and receive data. The main thread is the TransferManager thread that does the rest of the work. It runs the TransferManager, the callbacks and code of MessageHandler, and two functions of NIO: start server and start connection.

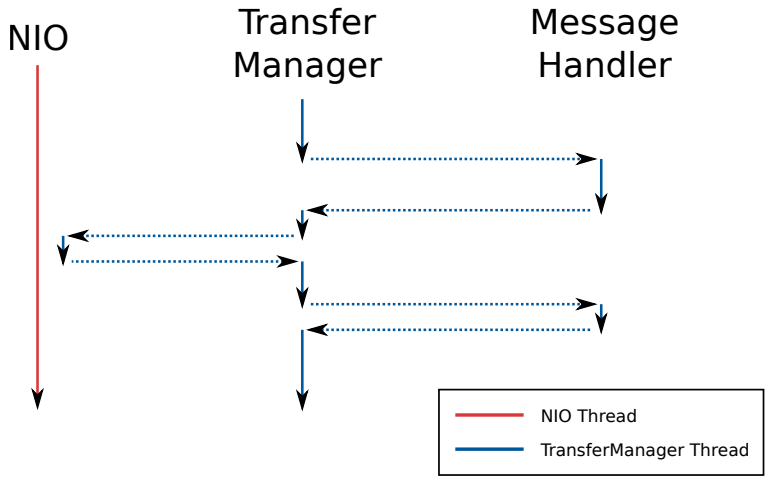

Figure 2: NIO threads

2 NIO

The main purpose of this class is to transfer data from one node to another. For this purpose the Java NIO (Non-blocking I/O) library has been used. The NIO library uses TCP as a transport layer, making NIO connection-oriented. An important aspect of NIO is that allows non-blocking transfer of data; this means that multiple connections can be open at once reading or writing without the thread waiting for each specific read or write. The NIO class in this implementation is the one that uses the Java NIO library.
The NIO class manages the servers, connections and reads/writes to the network interface. As explained in Section 1, a specific thread is running this NIO class.

2.1 Non-blocking I/O

The NIO library can be used for non-blocking I/O. This is done is by using a selector. A selector can be used to register interests for certain sockets. For instance one socket can register an interest in reading; when the socket receives data, the selector will call the read method for that socket. After one read, the socket will continue to check if there is more data. In order to stop checking for data to read, or any other interest, this interest has to be unregistered.
Four different interests can be registered:

- OP_ACCEPT: will notify the socket when there is an incoming connection.
- OP_CONNECT: will notify the socket when the connection is accepted.

- OP_READ: will notify the socket when there is data to read.
- OP_WRITE: will notify the socket when is ready to write.

More than one interest can be registered at once, although only one interested is registered at a time in this implementation.
It is important to note that the sockets are blocking by default, they must be set to non-blocking.

2.2 Connections

Since TCP is used as a transport layer, NIO needs to establish a connection before sending or receiving data. In order to establish a connection between two nodes, one node needs to open a server socket and register its interest as OP_ACCEPT. In this implementation, there is a list of servers for each node, making it possible to listen for incoming connections on different ports or network interfaces.

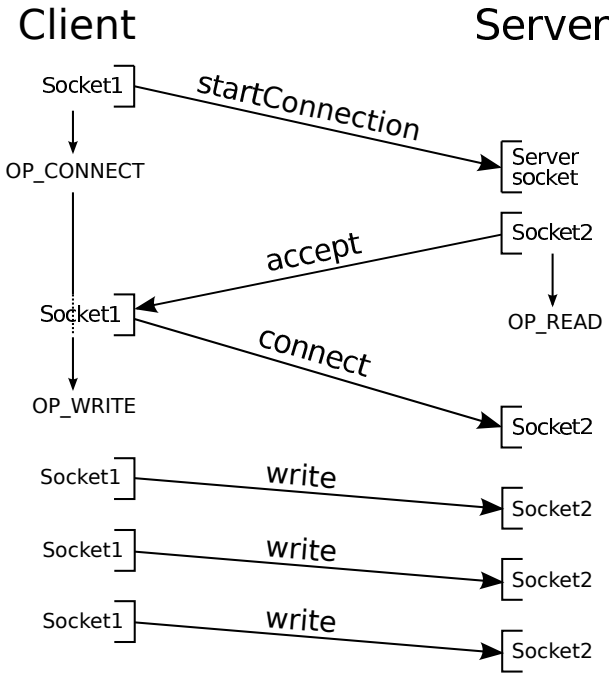

Figure 3: NIO connection process

A connection in NIO consists of a three step process similar to the TCP three way handshake. Usually, a connection is initiated by a node that wants to send data (client) to another node (server). The client opens a new socket specifically for this connection, connects to the desired server socket, and registers its interest in OP_CONNECT. On the server side, the server socket is listening for incoming connections. When it gets the connection query from the client, it creates a new socket for that connection, accepts the connection and registers the socket's interest in OP_READ. Finally, the client finishes the connection sending another message to the server, and changes the socket's interest to OP_WRITE. Now the client is ready to send messages and the server is ready to receive them.

2.3 Buffers

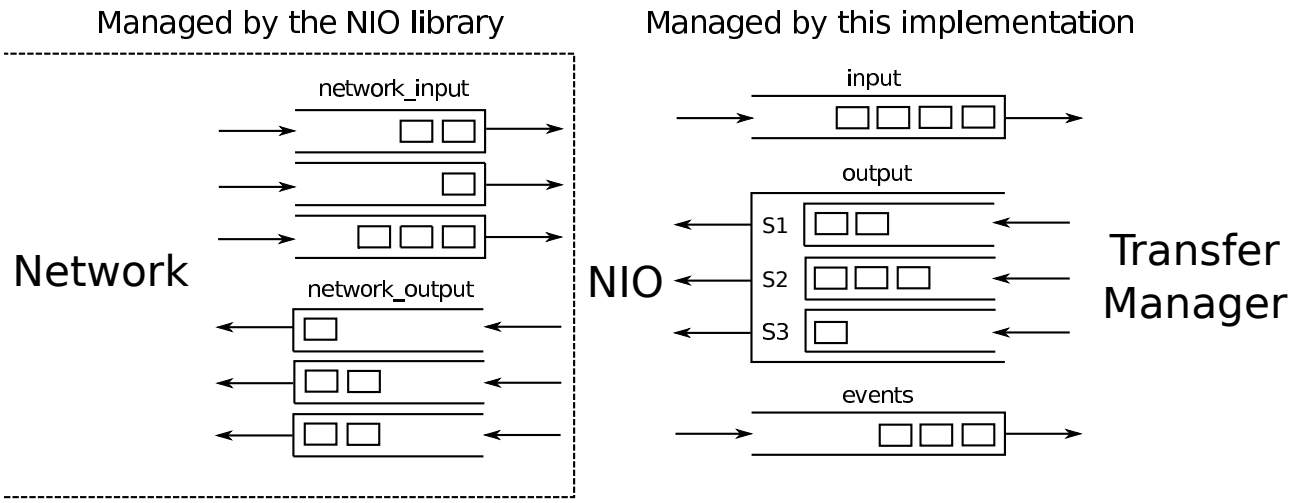

Figure 4: NIO buffers

2.4 Read

When a read interest is registered on a socket, it will call the read function whenever the network buffer receives some data. Then, the read function will put the buffer in the input queue along with the socket information from where it received the buffer, and when TransferManager gets to that element on the queue it will process it.

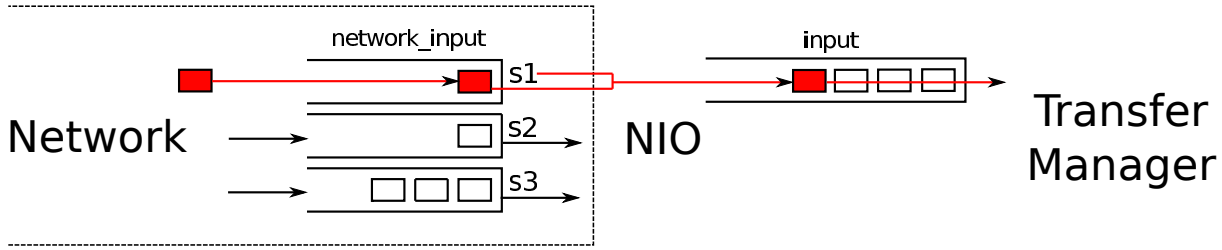

Figure 5: NIO read

An interesting behavior of the NIO socket read function is the following: if a socket from this node is registered with a read interest on a connection, and that connection is closed on the other end, it will always read and return -1 as the number of bytes read. This is used in order to terminate the connection, as explained in Section TODO.

2.5 Write

When a write interest is registered on a socket, it will call the write function always except when the network queue is full. The write function should get the first packet from the output queue for that socket and write it to the network queue.

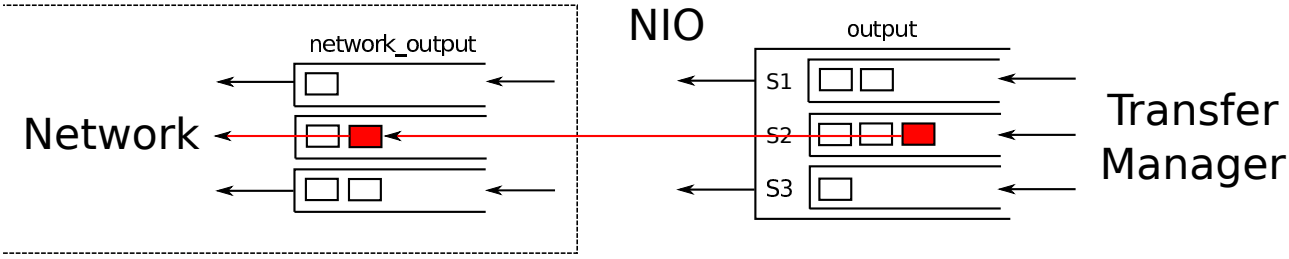

Figure 6: NIO write

If the output queue is empty, nothing will be written and an event will be sent to the TransferManager, as explained in the next section.

2.6 Events

In order to notify the TransferManager thread of important events that happen on the NIO thread, an event queue is used. There can be 3 different events:

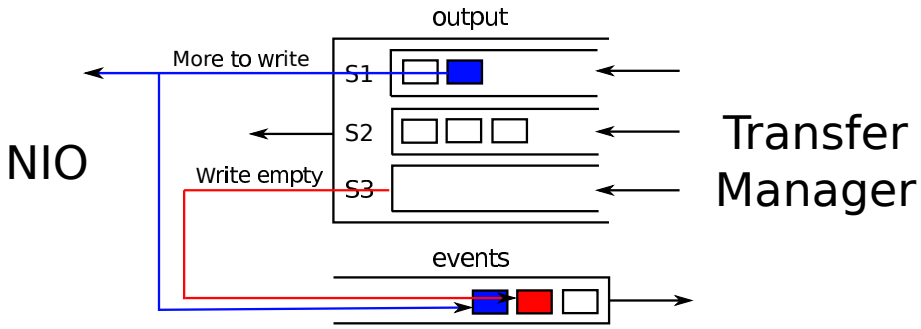

Figure 7: NIO events

- Error: there has been an error while connecting, reading, writing or finishing a connection. TransferManager will notify the MessageHandler of the error.
- More to write: in order to fill the output buffer with more packets, the TransferManager has to be notified that a number of packets have been consumed. The NIO thread will issue a More_to_write event when he has written to the network buffer PACKETS_NOTIFY number of packets. This variable can be modified in the NIO configuration file.
- Write empty: if a NIO socket is ready to write and finds its output queue empty, it will issue a Write_empty event. This can be caused because there is not more data to send, or because the More_to_write event has not been treated yet.

2.7 Socket changes

When a socket needs to change an interest (for instance as a part of the connection process), it must register its new interest on the selector. However -socket changes need to change selector -changes from selector must be made from the same thread -solution: queue that is handled by NIO thread (since is the same that uses the selector)

2.8 Packets

Whether an object or a file is being sent, the transmission starts with a 16-byte header, followed by the data.

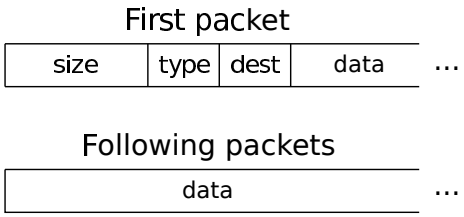

Figure 8: Packet

- Size: an 8 byte long that represents the size of the data.
- Type: a 4 byte int that represents the type of the transfer; either a Command or Data.
- Destination: a 4 byte int that represents the destination of the transfer; either an Object or a File.
- Data: the file or serialized object.

3 TransferManager

4 Connection

5 Transfer

A Transfer represents a single transfer send or received through an existing Connection between two nodes. Each transfer has a direction, a type and a destination.

- Direction: the direction of the transfer in that node: Send or Receive.
- Type: the type of the contents of the transfer: Command or Data.
  - Command: an order from one node to another. The first message from one node to another in a new Connection has to be a Command transfer.
  - Data: data in either an Object or File format.
- Destination: the format destination of the transfer: Object or File.
  - Object: the object to send must implement the Serializable interface.
  - File: in order to send/receive a file, the name of the file must be sent through a command, since a data message will only send the file contents.
  - Note (TODO): usually when one node sends a file, the other expects to receive a file; and one one node sends an object, the other expects to receive an object. However, when a serialized object has been written to a file, this node can send a file, and the

destination node can receive an object, so that instead of writing to disk, then reading and then deserializing, direct deserialization can be applied.

5.1 Streams

Depending on the kind of data to be sent (file or object), the process and streams opened will be different.

5.1.1 File

If the data to be received is a file, a File Stream is opened at the beginning of the connection. Each time new data arrives, it is written into the stream, which directly writes to disk. At the end of the connection, the stream is closed without any additional operations.
The same process applies for the sending end: a File Stream is opened, then the data is directly read from disk, and finally the stream is closed.

5.1.2 Object

When an object is transfered, the first step is to serialize the object. Afterwards, the whole serialized object is located in memory in a byte array, so a Byte Array Stream is opened. Now the byte array can be read and sent over the network. When the transmission finishes, the byte stream is closed. If the node is receiving the object, a Byte Array Stream is opened, and all the data is written in it. After the transmission has finished, the byte array is deserialized into an object.

6 File specified parameters

The following parameters must be specified in the nio.cfg file.

- **Max packets per transfer**
  Maximum packets stored in the buffer per transfer.
- **Buffer size**
  Size in packets of the buffer between TransferManager and NIO.
- **Packets notify**
  Number of packets transfered by the network after which NIO will ask TransferManager to enqueue more. Must be smaller than *Max packets per transfer*.
- **Network buffer size**
  Size in bytes of the network buffer (must be at least *Buffer size*Max packets per transfer*).
- **Server port**
  Port where the default server will listen to incoming connections.