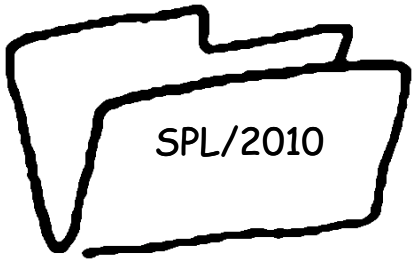
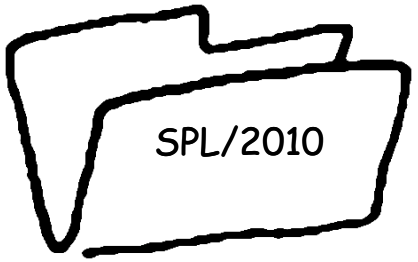
Reactor Design Pattern



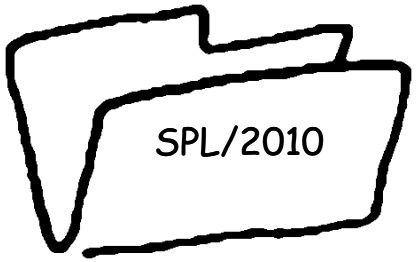
Overview

- blocking sockets - impact on server scalability.
- non-blocking IO in Java - java.niopackage
  - complications due to asynchronous nature
  - impact on message parsing algorithms
- Reactor design pattern
  - generic server
  - more scalable than our earlier solutions …



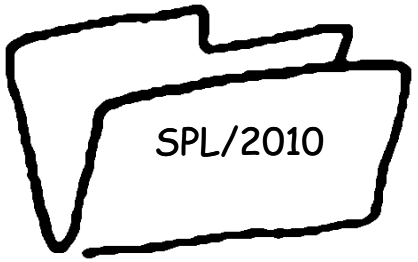
Sockets

- Socket API: interaction with RTE when a process requires communication services.
- three properties of Sockets:
  1. ServerSockets: accept incoming connections.
  2. OutputStream: send bytes through Socket
  3. InputStream: receive bytes through Socket



blocking IO operations

- control does not return to calling thread until operation terminates.
- accept() - calling thread is blocked until new connection established.
- write(byte [] buffer) - calling thread is blocked until all buffer sent to network.
- read(byte [] buffer) - calling thread is blocked until buffer.length is received.

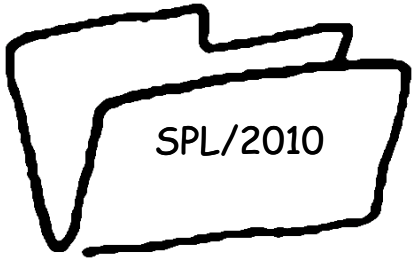


Drawbacks

server for multiple clients needs:

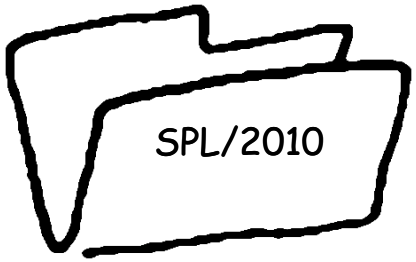
- thread for each client
- thread to accept (new) incoming connections.

- scalability problems: resources of server process (threads, memory, CPU) increase linearly by number of clients



Goals - design scalable and robust servers

- More scalable - grow while using bounded amount of resources
  -

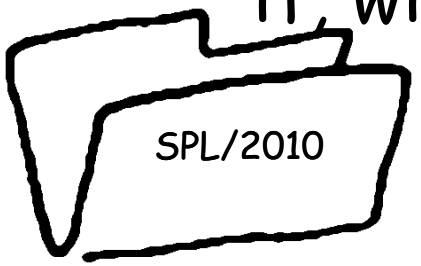


Non-blocking IO

- **Check** socket has some data available to read.
- **Non-block** read available data from socket. Return with any amount of data.

- **Check** socket can send some data.
- **Non-block** write data to socket. Return immediately.

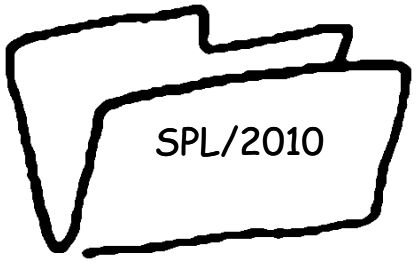
- **Check** new connection is requested. If so, accept it, without blocking!



RTE perspective

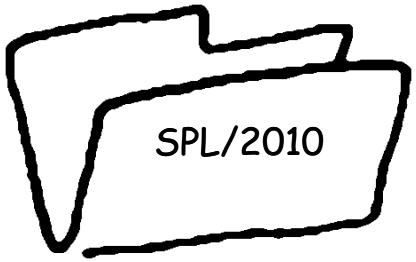
- partition solution in two logical parts:
  1. **Readiness** notification
  2. **Non-blocking** input output.

- Modern RTEs supply both mechanisms:
  - Is data is available for read in socket
  - Is socket ready to send some data
  - Is there new connection pending for socket
  - Non-blocking interface to read and write



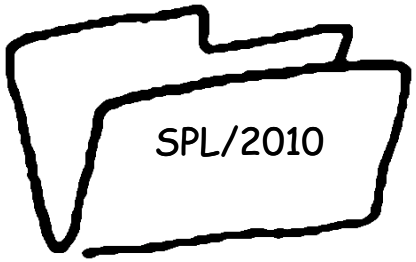
output buffer

- to understand how non-blocking operations work - understand how RTE internally manages IO
  - buffer associated with each socket
  - write to socket - RTE copies bytes to internal buffer
  - RTE proceeds to send bytes from buffer to network.



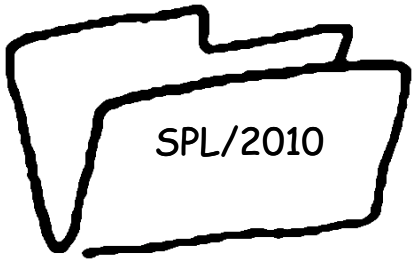
Block vs. Non-block Write

- network slower than process - output buffer fills more quickly than RTE sends bytes to network.
- if output buffer is full - RTE **blocks** process writes until output buffer has enough free space for data.
- non-blocking write - RTE copies bytes from process as possible
  - notifies process how many bytes have been copied.
  - if bytes need to be re-written, process responsible to **re-invoke** the write operation



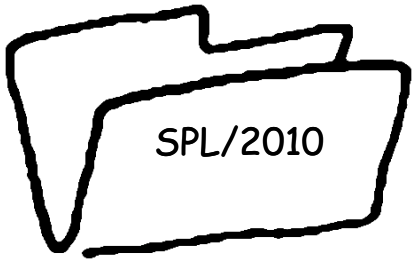
Block vs. Non-block Read

- RTE need to receive bytes over network,
- deliver these bytes to our process.
- RTE buffer bytes until process actually requests them using a read operation.



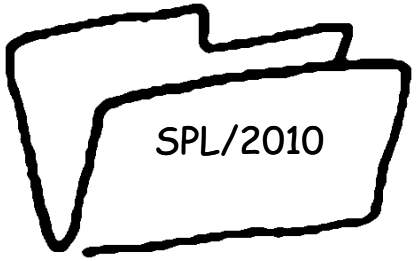
input buffer

- allocated to each socket
- RTE stores incoming bytes to input buffer
- when process read from socket, RTE copies the from socket's input buffer to process buffer.
- if process request more bytes than available, RTE **blocks** until enough bytes.
- non-block - RTE copy bytes available in socket's input buffer, and notify number of bytes copied.



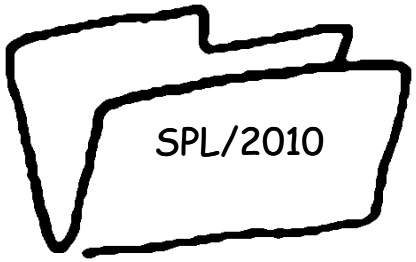
Input Buffer Overflow

- input buffer has limited space
- input overflowed - process reads data more slowly than data arrives from network.
- if input buffer full, RTE discard new data arriving from network
  - sending side will retransmit the data later?



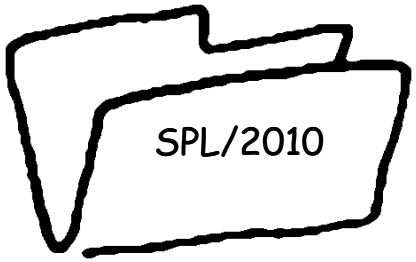
Java's NIO Package

- Java's interface to non-blocking IO and readiness notification services of RTE
- provides wrapper classes for readiness notification and non-blocking IO.



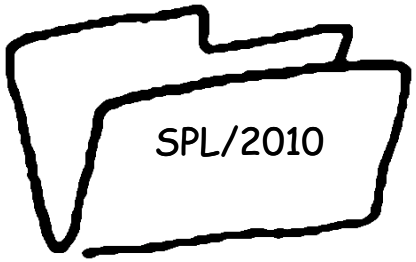
Channels

- connections to I/O entities
  - Represent data source, destination
- Examples: ServerSocketChannel SocketChannel, FileChannel
  - default - new channels are in blocking mode
  - must be set manually to non-blocking mode.



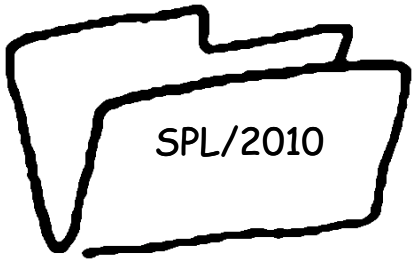
Channels

- methods for writing and reading bytes.
- ServerSocketChannel provides accept()
  - returns a SocketChannel (similar to accept() method of ServerSocket class).



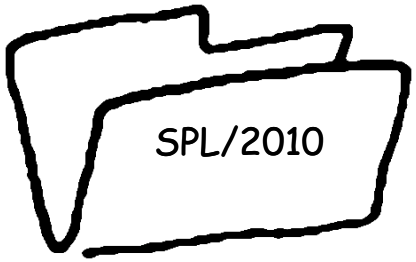
Selectors

- implements **readiness notification**.
- channels may be registered to a selector for specific readiness **events**
  - read /write /accept
  - selector can be polled to get list of ready channels



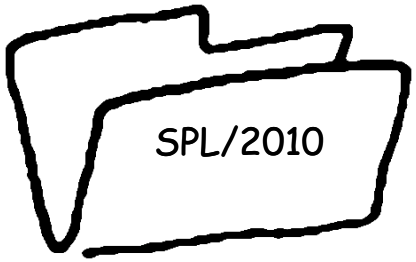
Selectors

- channel **ready** for read guarantees that a read operation will return some bytes.
- channel **ready** for write guarantees that a write operation will write some bytes
- channel **ready** for accept guarantees that accept() will result in a new connection.



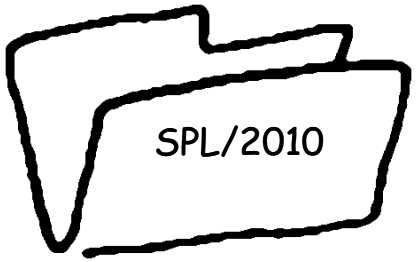
Selectors

- Selector class abstracts service given by os under the sys. call **select** (**epoll**).
- receives a collection of sockets (and **blocks**) until one ready for reading or writing.
- When call returns, caller is informed with ready sockets.



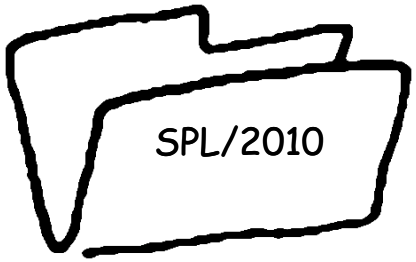
Buffers

- wrapper classes used by NIO to represent data interchanged through a Channel.
- usually backed by array types.
- Example: SocketChannels use ByteBuffer class for sending and receiving bytes.

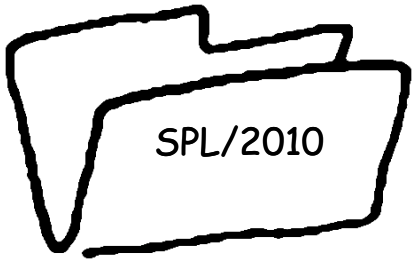


Charsets

- together with associated *decoders* and *encoders* - translate between bytes and Unicode characters;

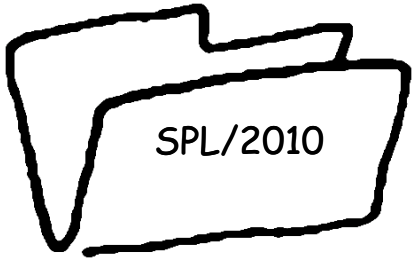


Reactor Design Pattern



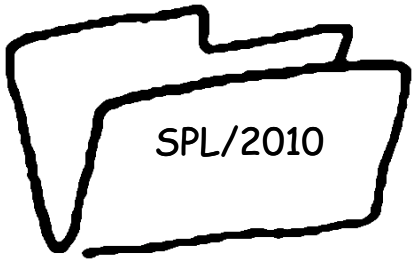
Reactor features

- solve scalability problems we encountered before.
- employs non-blocking IO.
- maintains a set of sockets, using a selector, reactor polls for readiness.
- For each socket, reactor attaches state.



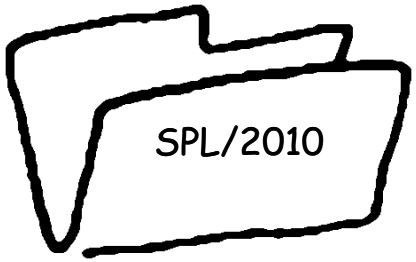
Reactor IO

- if bytes ready to read from socket, reactor read bytes and transfer to protocol (previous lecture)
- if socket is ready for writing, reactor checks if write request - if so, reactor sends data
- reactor is accepting new connections.



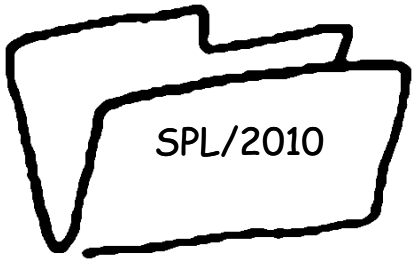
Reactor Design

- reactor composed of:
  - 3 classes
  - main thread
  - thread pool.
- abstract protocol and message format behind interfaces



Main Reactor thread

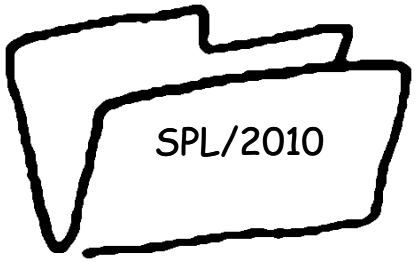
- **main** reactor thread performs the following:
  1. Create new thread pool (**executor**).
  2. Create new ServerSocketChannel, bind to port.
  3. Create new Selector.
  4. Register ServerSocketChannel in Selector, asking for accept readiness.
  5. While(true) - wait for selector notifications



Main Reactor thread

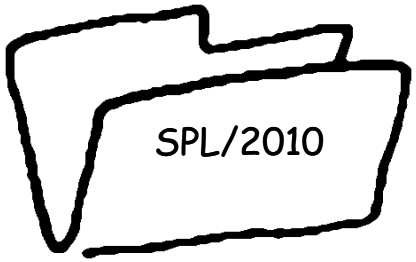
For each notification event check:

- **Accept notification** - server socket is ready to accept new connection - call accept
  - new socket created - register socket in Selector.
- **Write notification** - socket ready for writing, if protocol ask to write - write bytes to socket.
- **Read notification** - socket ready for reading, read bytes and pass them to protocol handler.



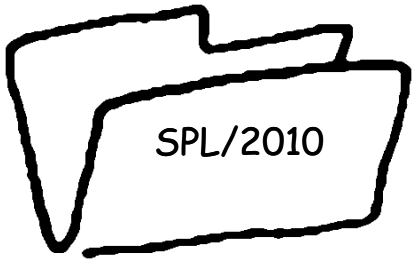
pool thread

- actual work done by protocol will be achieved with the use of thread pool;
  - protocol processing is assigned as task for pool.



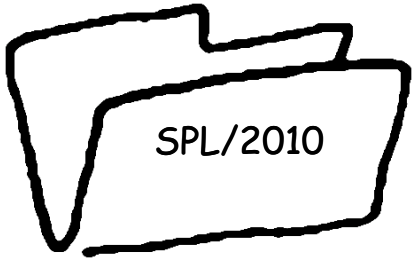
Reactor classes: ReactorData

- encapsulates internal state of reactor need outside of reactor. Contains references to:
  - Selector
  - executor service (pool)
  - protocol factory (produces MessagingProtocol objects - attached to new client).



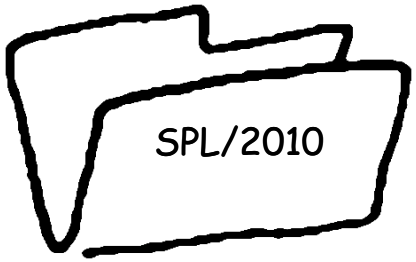
Reactor classes: ConnectionAcceptor

- accepting incoming connections, creating connection Handler



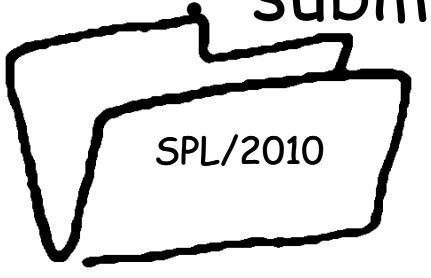
Reactor classes: ConnectionHandler

- reading/writing bytes from/to sockets.
- bytes read are passed to associated protocol object, which, in turn, updates ConnectionHandler with outgoing bytes.
- performs initializations upon creation
  - setting new socket to non-blocking mode
  - creating new protocol object
  - registering new socket in selector.



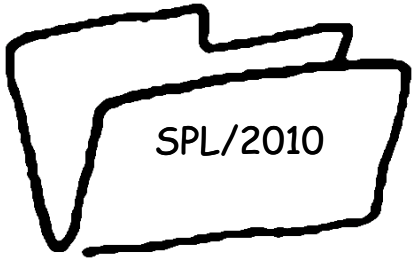
difference between reactor and one-thread-per-connection

- **ConnectionHandler:**
  - passive object (instead active object)
  - methods are executed by main thread of Reactor in reaction to events relayed by selector
  - methods don't block –execute very fast - copying bytes from one buffer to another.
- parsing/processing messages is delegated to active objects **ProtocolTask**
  - submitted to thread pool executor



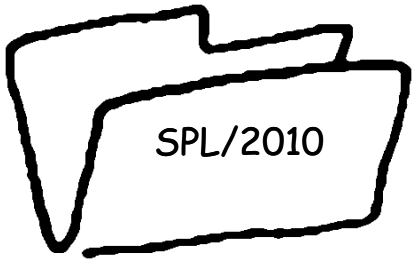
Cont'd

- must maintain direct mapping between socket channels and their associated handlers.
- Selector class allows to attach arbitrary object to channel, which can later be retrieved,
- we associate **ConnectionAcceptor** with the **ServerSocketChannel**
- we associate **ConnectionHandler** with socket created when accepting new connection.



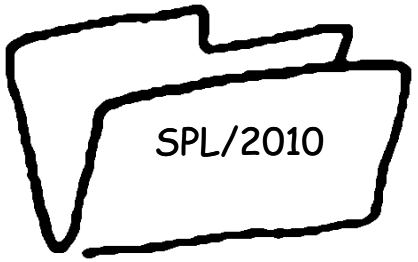
Synchronization?

- assume client initiates connection by sending message first
  - newly created socket is registered in the selector for read notifications only.
- after some bytes has been received and the protocol wants to reply
  - socket registration must change to read/write.
- After all bytes were written, socket is returned to read only notifications.



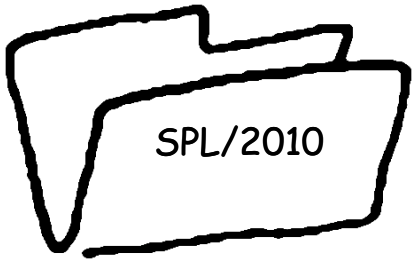
Reactor Overview

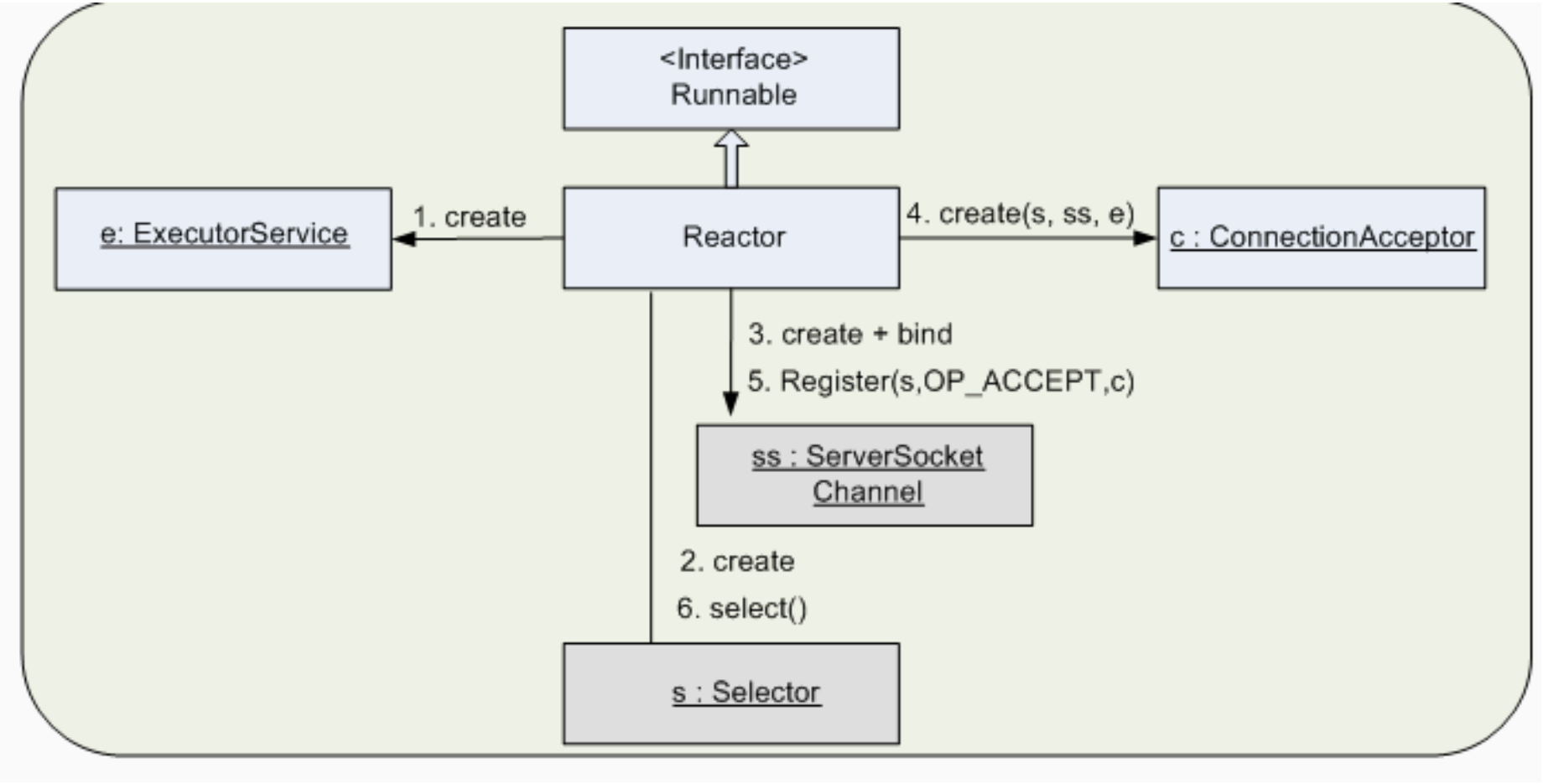
- Reactor diagram depicts the messages sent by reactor object once it starts running.
- reactor is run by a thread we call *ReactorThread*
- Selector response to new connection or read/write readiness of its bound socket as an *event*.
- 3 diagrams with the sequence of messages after an event took place.



Reactor Overview

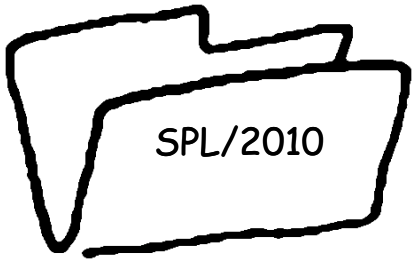
- event handling is done by two threads:
  1. **ReactorThread** pulls the bytes from socket and places them in a buffer.
  2. **Thread-pool** thread:
     - processes bytes using a tokenizer and protocol.
     - writes the protocol response back to the **connection handler** outgoing buffer




<Interface>
Runnable
e: ExecutorService
1. create
Reactor
4. create(s, ss, e)
c : ConnectionAcceptor
3. create + bind
5. Register(s,OP_ACCEPT,c)
ss : ServerSocket
Channel
2. create
6. select()
s : Selector

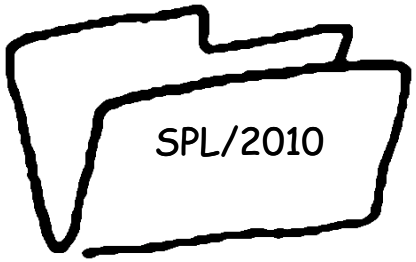
The Reactor Class

- Reactor is an active object.
- the heart of the architecture - connects other components and triggers operations.
- key components of the Reactor are:
  - selector
  - thread pool executor



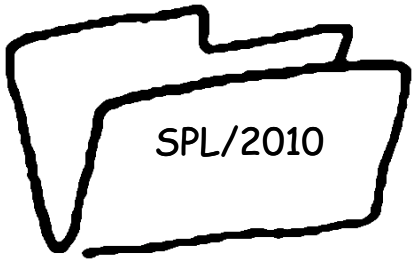
The Reactor Class

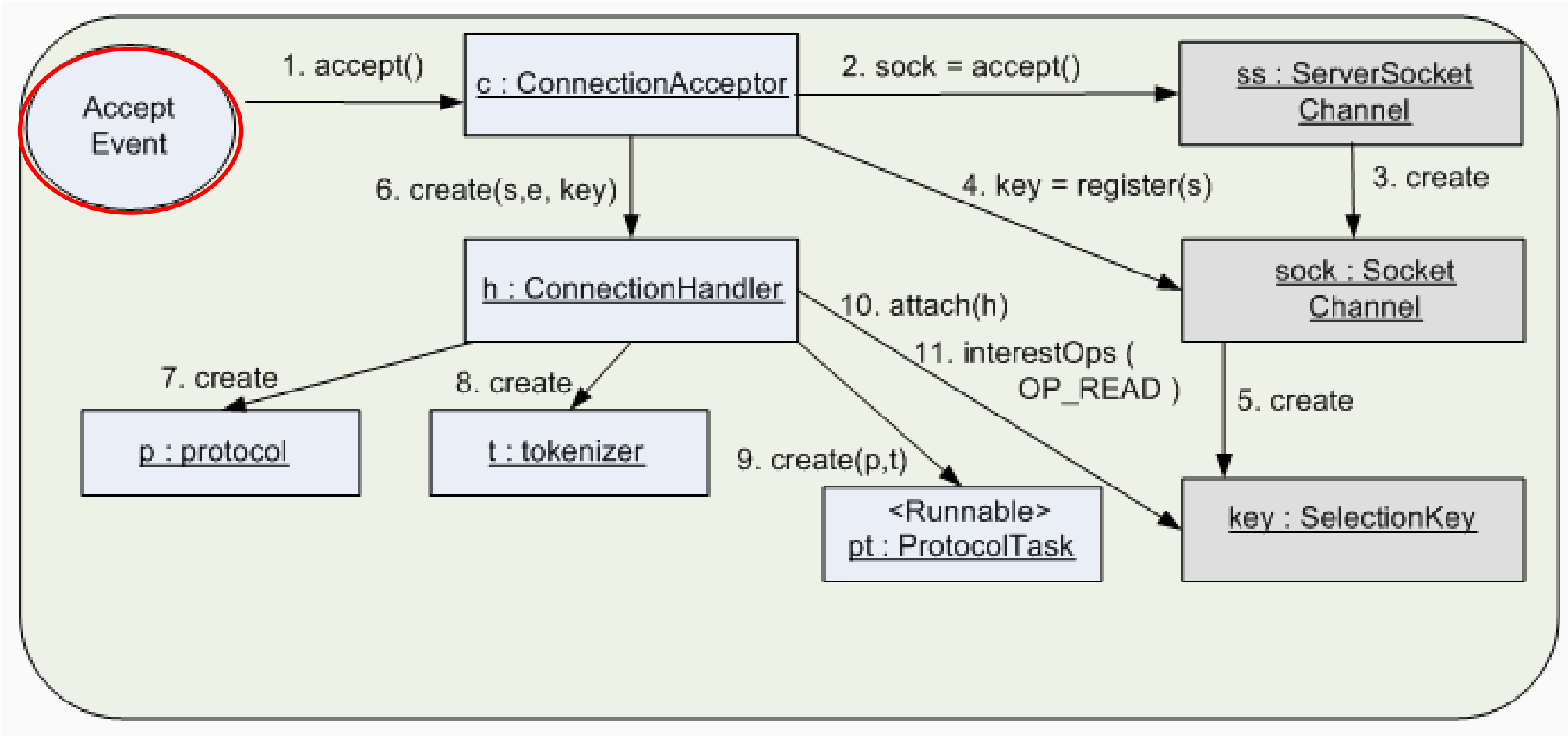
- Reactor thread listens to events from selector.
- Initially, only a ServerSocketChannel is connected to the selector.
  - Reactor can only react to accept events.

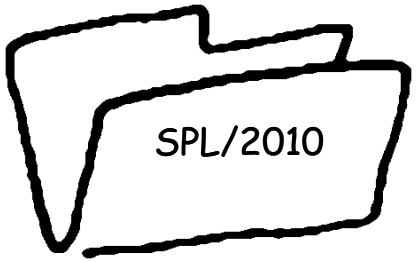


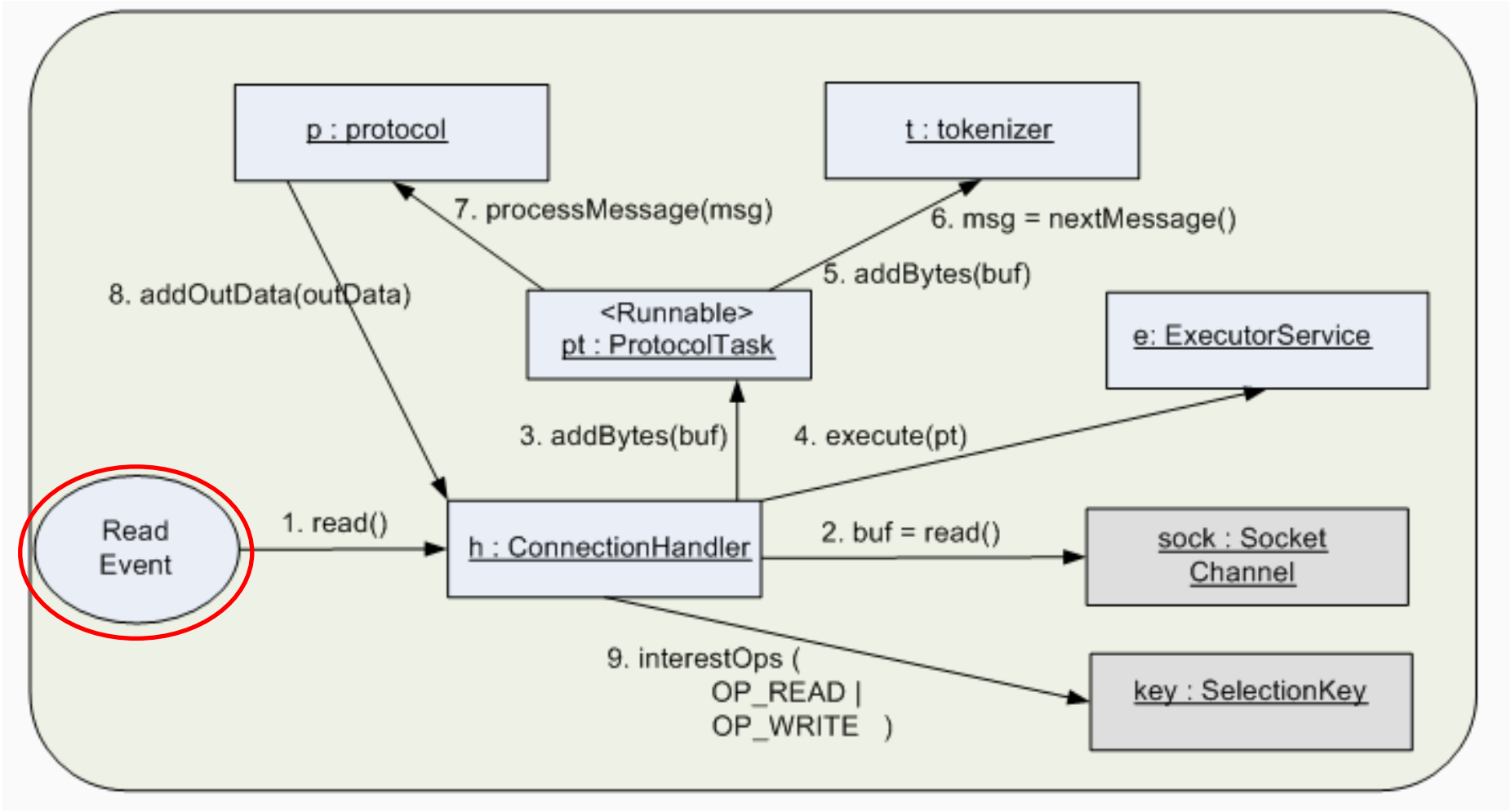
The Reactor Class

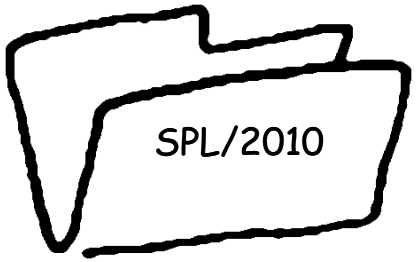
- run() method dispatches events from selector,
- reacts by invoking methods of either Accept or ConnectionHandler passive objects.
- attaching new client socket channels to selector
  - detaching when client disconnects.
  - performed by the Acceptor object.

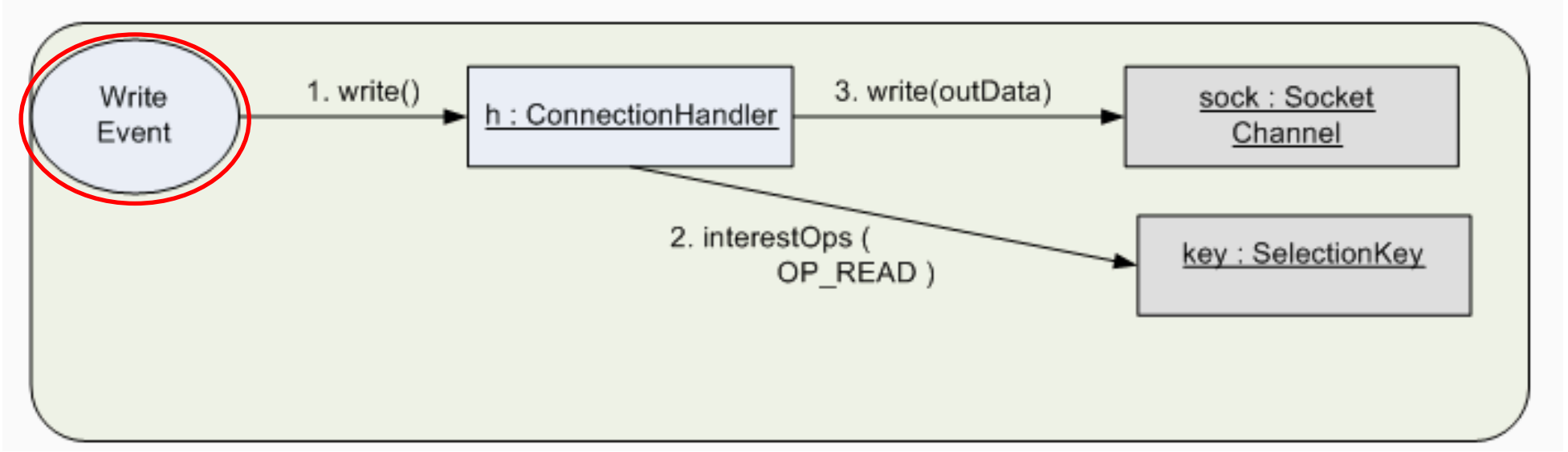



Accept Event
1. accept()
c : ConnectionAcceptor
2. sock = accept()
ss : ServerSocket Channel
3. create
4. key = register(s)
6. create(s,e, key)
h : ConnectionHandler
sock : Socket Channel
10. attach(h)
11. interestOps (OP_READ)
5. create
7. create
8. create
p : protocol
t : tokenizer
9. create(p,t)
<Runnable> pt : ProtocolTask
key : SelectionKey


SPL/2010


p : protocol
t : tokenizer
7. processMessage(msg)
6. msg = nextMessage()
8. addOutData(outData)
5. addBytes(buf)
<Runnable>
pt : ProtocolTask
e: ExecutorService
3. addBytes(buf)
4. execute(pt)
Read
Event
1. read()
h : ConnectionHandler
2. buf = read()
sock : Socket
Channel
9. interestOps (
OP_READ |
OP_WRITE)
key : SelectionKey


SPL/2010


Write
Event
1. write()
h : ConnectionHandler
3. write(outData)
sock : Socket
Channel
2. interestOps (
OP_READ)
key : SelectionKey