

# Re: Non-blocking and semi-blocking Sockets class.

---

*Source:* <http://coding.derkeiler.com/Archive/Java/comp.lang.java.programmer/2007-01/msg01924.html>

---

- *From:* "Karl Uppiano" <[karl.uppiano@xxxxxxxxxxxxx](mailto:karl.uppiano@xxxxxxxxxxxxx)>
  - *Date:* Thu, 18 Jan 2007 06:12:50 GMT
- 

"nukleus" <[nukleus@xxxxxxxxxxxxx](mailto:nukleus@xxxxxxxxxxxxx)> wrote in message  
[news:eomuht\\$3m3\\$4@xxxxxxxxxxxxx](mailto:news:eomuht$3m3$4@xxxxxxxxxxxxx)

I am working on a network related application.  
There is an issues with sockets blocking and how to handle it.  
There are several approaches:

1. Use blocking sockets and set timeout.  
In this case, if connection is lost or socket object is not created yet as we are in the middle of connect operation, the process effectively hangs and user has to wait for timeout expiration.  
Not a very desirable approach.

2. Use byte reads and buffered input stream and check if data is available. In case of text, if it is, then read it until you get a line separator character, and then return a text line to a caller of `getLine()`.

If eof condition happens, returned by `-1` on a read, that could mean the server closed a connection for whatever reason. In this case, simply return. Non blocking operation.

If there is nothing in the buffer, but you are waiting for a server standard response or data, you loop until such data arrives. This is blocking operation and has all sorts of nasty side effects. The loop uses a separate timer from the socket timeout timer. When that timer expires, the `getLine()` returns with error or throws a `Timeout` exception.

But it could be resolved using a separate thread for a blocking socket.

One of the alternatives here is to use a byte read loop with that could be interrupted, say, if user realizes there is error, and hits Cancel button. In this case, the event handler sets the event, that is tested in a read loop, and if the flag is set, the loop terminates.

This is interruptible blocking operation, which should effectively

## Re: Non-blocking and semi-blocking Sockets class.

behave like a non-blocking operation for practical purposes.

### 3. Blocking socket running as a separate thread.

This seems to be the best overall approach, as we can allow the socket thread to block, but we are not block on the main thread level and all our gui stuff operations as advertised.

But there are issues with this. Since you are running a separate thread, how do you tell it what command we issue to the server and how does the main thread knows when operation is completed successfully. Anotherwords, how do you exchange the information and various error conditions between the main object and the socket?

From windows C++ standpoint, you can go asynchronous using overlapped structures for one thing.

Question: what is the best approach with java and what are the exact steps to design the separate socket thread?

Do we use Events?

What happens to Exceptions running in a separate thread that need to be communicated to a different thread of different frame object?

Since we create a separate thread for a socket that has its own run() loop and we could be performing either send or receive network operation, how do we exchange data between socket thread and a parent class?

### 4. Fully asynchronous design.

This design, by definition, is non-blocking.

The overall architecture may be implemented with a state machine approach. In this case, there is never any blocking on a socket object. But how do you do it in Java?

It seems events are the way to go, but how do you set up your architecture to do it?

Related question:

Anyone can suggest a good state machine design in Java?

I used my own design in C++ that had a vector of state class objects and a state machine iterating code. Each state object had a paramter telling it which is the next state to proceed to in case of successful result, which state to proceed to in case of error result, and which alternative states to proceed to in case of minor error conditions, such as incorrect user command issued to the server in our situation, or incorrect value of data passed to server.

This design is flexible in that the state machine could even be substituted for another state machine, need be. For example, if you are trying to send email, and you made a mistake in your configuration that tells the server that you are connecting to

## Re: Non-blocking and semi-blocking Sockets class.

NNTP server, than the server log-in response code would be different. In this case, you could switch the state machine on the fly and perform the NNTP related operation. This is just an abstract example. You wouldn't probably want to do that in real life. :--}

Does anyone have any feedback on this all?

Thanks in advance.

For clients and smaller server applications, Solution 3 seems like the best option to me. Your concerns about communicating between threads can be solved with standard threading design patterns. Java 1.5 introduced three entire packages devoted to thread synchronization (`java.util.concurrent`, `java.util.concurrent.atomic` and `java.util.concurrent.locks`). I think you need to look into that part of the problem.

Package `java.nio` (possibly option 4) is good for server networking applications that need to scale massively, but NIO has a pretty steep learning curve.

State machines... I have written lots of them. They usually consist of a state variable that might hold a reference to an interface or abstract base class that implements the state.

```
interface state {
/**
 * State implementations execute their state,
 * and set a state variable to the "next" state.
 */
public void execute();
}
```

```
class stateMachine {
stateVariable = new state0();

while(true) {
stateVariable.execute();
}
}
```

Obviously, this pseudo code glosses over the details. I would probably make the state implementations inner classes of the state machine, so they would have access to `stateVariable` to set it. I might even make the state implementations themselves stateless, and static final, so setting the state would be a simple assignment of "constants". Maybe enums could represent the state. I haven't had to do a state machine since Java 1.5 introduced enums, but it sounds promising.

Re: Non-blocking and semi-blocking Sockets class.