NIO
Non-blocking Input/Output

Advanced Java I. Functional, Asynchronous, Reactive Java

Module 3

think.
create.
accelerate.

Luxoft | training
A DXC Technology Company

Reading and Writing Data (Input and Output)

Streams

Core concept in Java IO

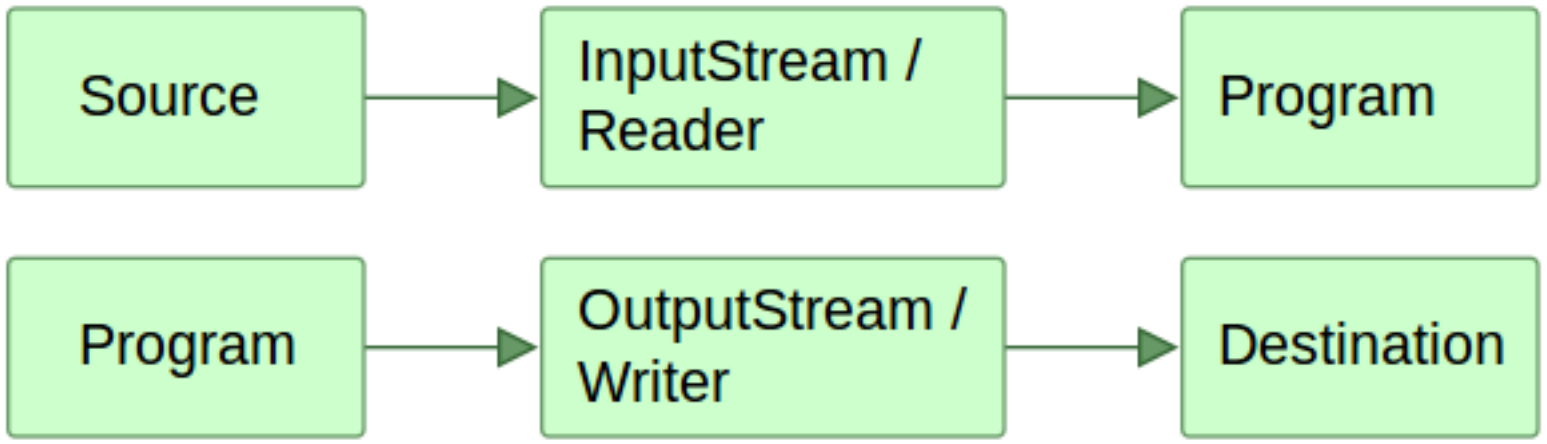
- A stream is a conceptually **endless flow of data**.
- You can **either read** from a stream **or write** to a stream.
- A stream is connected to a **data source** or a **data destination**.
- Streams in Java IO can be either **byte based** or **character based**.

Input and Output

`java.io` package

- To read data from some source, program needs an **InputStream** or a **Reader**.
- To write data to some destination, program needs an **OutputStream** or a **Writer**.

The principle of a program reading data from a source and writing it to some destination:



Sources and Destinations

- **`java.io` package**

  - Files
  - Pipes
  - Network Connections
  - In-memory Buffers (e.g. arrays)
  - System.in, System.out, System.error

`InputStream` class

Byte Based Input

The base class (superclass) of all input streams in the Java IO API.

Subclasses:

- FileInputStream
- BufferedInputStream
- PushbackInputStream
- DataInputStream
- ObjectInputStream
- ByteArrayInputStream
- PipedInputStream
- FilterInputStream

An InputStream is typically always connected to some data source, like a file, network connection, pipe etc.

InputStream example

- **java.io.InputStream**

```
InputStream inputstream =
     new FileInputStream("c:\\data\\text.txt");
int data = inputstream.read();
while(data != -1) {
 doSomethingWithData(data); //do something with data...
 data = inputstream.read();
}
inputstream.close();
```

// FileInputStream is a subclass of InputStream so it is safe to assign an instance of FileInputStream to an InputStream variable.

`read()` method

- **Returns an int which contains the byte value of the byte read.**

```
int data = inputstream.read();
char aChar = (char) data; // case the returned int to a char
```

If the read() method returns -1, the end of stream has been reached, meaning there is no more data to read in the InputStream.

`read(byte[])` method

- **Read data from the InputStream's source into a byte array.**

```
int read(byte[])

byte[] buf = new byte[100];
int read(buf)
```

- Read as many bytes into the byte array given as parameter as the array has space for.
- Returns an int telling how many bytes were actually read.

```
int read(byte[],int offset,int length)
```

- Also reads bytes into a byte array, but starts at offset bytes into the array, and reads a maximum of length bytes into the array from that position.

`OutputStream` class

Byte Based Output

The base class (superclass) of all output streams in the Java IO API.

Subclasses:

- FileOutputStream
- BufferedOutputStream

- PrintStream
- DataOutputStream
- ObjectOutputStream
- ByteArrayOutputStream
- PipedOutputStream
- FilterOutputStream

An OutputStream is typically always connected to some data destination, like a file, network connection, pipe etc.

OutputStream example

- **java.io.OutputStream**

```
OutputStream output = new BufferedOutputStream(
     new FileOutputStream("c:\\data\\text.txt"));
while(hasMoreData()) {
 int data = getMoreData();
 output.write(data);
}
output.close();
```

`write()` method

- **Writes byte/bytes to the OutputStream.**

```
write(byte) // writes a single byte to the OutputStream
```

- Takes an int which contains the byte value of the byte to write.
- Only the first byte of the int value is written.
- The rest is ignored.

```
write(byte[])
write(byte[] bytes, int offset, int length)

// writes an array or part of an array of bytes to the OutputStream
```

Non-blocking Input/Output

Main Differences Between Java NIO and IO

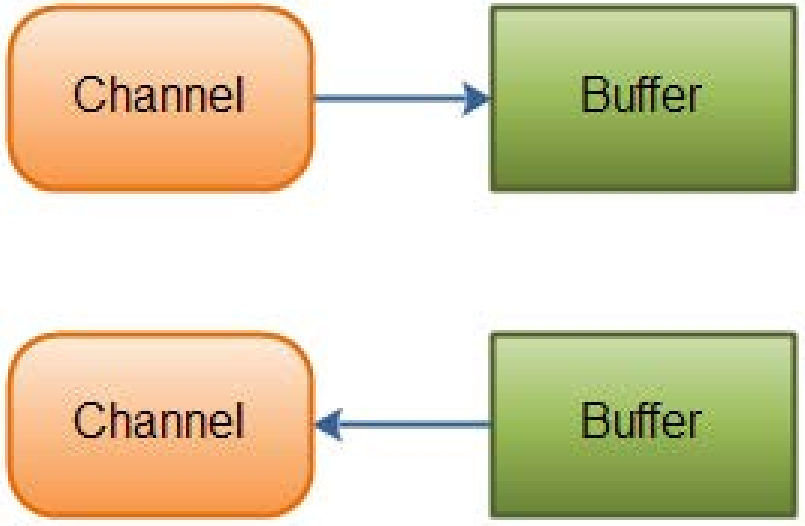
- **NIO contains classes that does much of the same as the Java IO**

IO	NIO
Stream oriented	Buffer oriented
Blocking IO	Non blocking IO
	Selectors

NIO Channel vs. Stream

Java NIO Channels are similar to streams with a few differences

- You can both read and write to a Channels. Streams are typically one-way (read or write).
- Channels can be read and written asynchronously.
- Channels always read to, or write from, a Buffer.



NIO Channel types

The most important Channel implementations in Java NIO:

- **FileChannel** reads data from and to files.
- **DatagramChannel** can read and write data over the network via UDP.
- **SocketChannel** can read and write data over the network via TCP.
- **ServerSocketChannel** allows you to listen for incoming TCP connections, like a web server does. For each incoming connection a **SocketChannel** is created.

Stream Oriented vs. Buffer Oriented

- **IO is stream oriented, where NIO is buffer oriented.**

Stream Oriented IO

- You read one or more bytes at a time, from a stream.
- What you do with the read bytes is up to you.
- They are not cached anywhere.
- You cannot move forth and back in the data in a stream.

Buffer Oriented NIO

- Data is read into a buffer from which it is later processed.
- You can move forth and back in the buffer as you need to.
- This gives you a bit more flexibility during processing.

Blocking vs. Non-blocking IO

IO’s blocking vs. NIO’s non-blocking.

IO's blocking streams

- When a thread invokes a `read()` or `write()`, that thread is **blocked** until there is some data to read, or the data is fully written.
- The thread can **do nothing else** in the meantime.

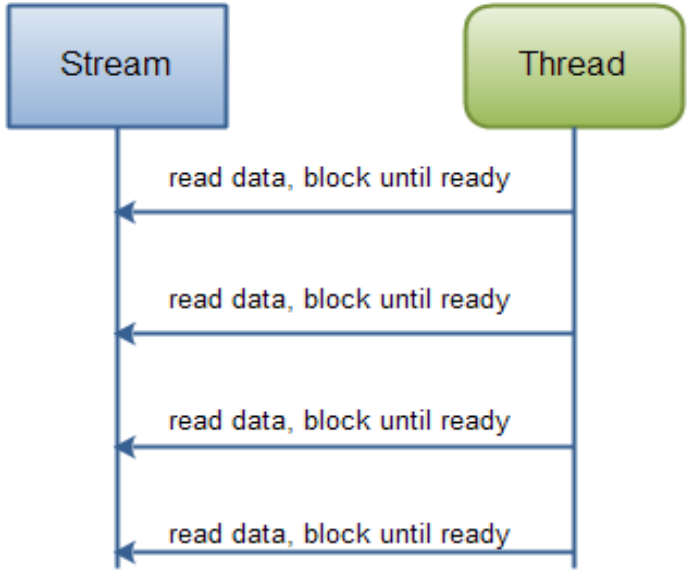
NIO's non-blocking mode

- A thread can request that some data be written/read to/from a channel, but **not wait for it to be fully written** or only **get what is currently available to read**.
- The thread can then go on and **do something else** in the mean time.

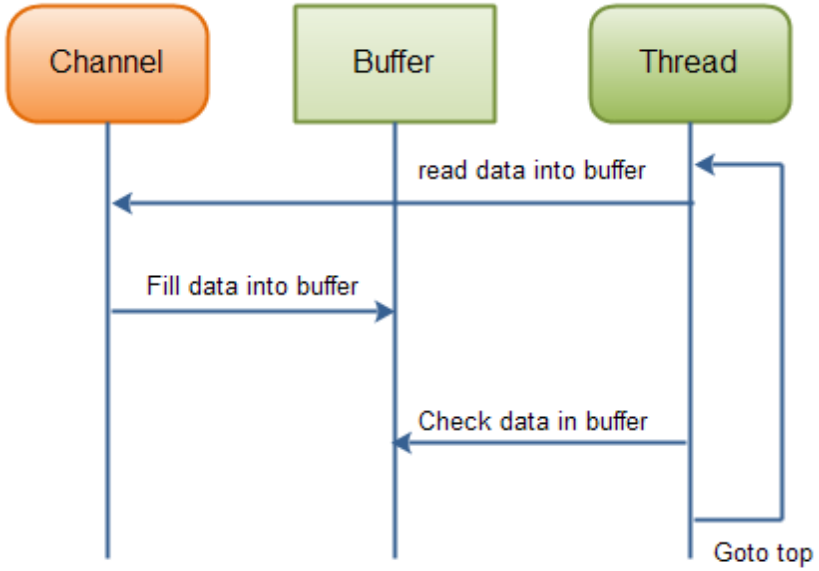
How NIO and IO Influences Application Design

- **IO toolkit may impact the following aspects of your application design:**

- The API calls to the NIO or IO classes.
- The processing of data.
- The number of thread used to process the data.



Java IO: Reading data from a blocking stream



Java NIO: Reading data from a channel until all needed data is in buffer

Java NIO core components:

- Channels
- Buffers
- Selectors

Java NIO Buffer

- **A buffer is essentially a block of memory into which you can write data, which you can then later read again.**

Using a Buffer to read and write data typically follows this little **4-step process**:

- Write data into the Buffer
- Call buffer.flip() // to switch the buffer from writing mode into reading mode
- Read data out of the Buffer
- Call buffer.clear() or buffer.compact() // to clear the whole buffer or only the data which you have already read

Basic Buffer Example

- **A simple Buffer usage example:**

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");
FileChannel inChannel = aFile.getChannel();
ByteBuffer buf = ByteBuffer.allocate(48); //create buffer with capacity of 48 bytes
int bytesRead = inChannel.read(buf); //read into buffer
while (bytesRead != -1) {
   buf.flip();  //make buffer ready for read
   while(buf.hasRemaining()){
      System.out.print((char) buf.get()); // read 1 byte at a time
   }
   buf.clear(); //make buffer ready for writing
   bytesRead = inChannel.read(buf);
}
aFile.close();
```

Buffer Capacity, Position and Limit

- **A Buffer has three properties you need to be familiar with:**

- **Capacity**

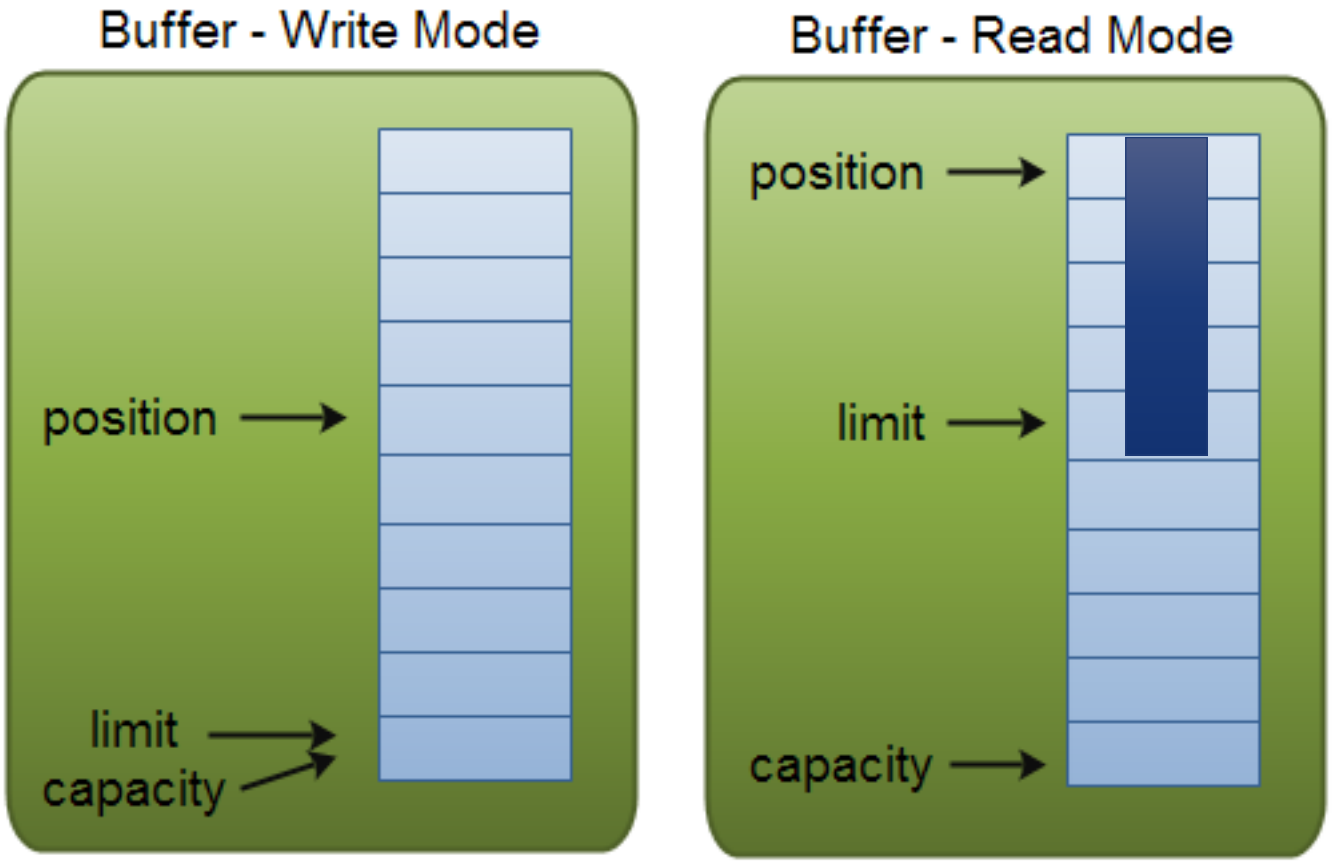
// Buffer's certain fixed size

- **Position**

// a certain position, at which you read/write data

- **Limit**

// in read mode, it is the limit of how much data you can write; in write mode, it equals the capacity



Buffer Types

- Java NIO comes with the following Buffer types:

ByteBuffer

MappedByteBuffer

CharBuffer

DoubleBuffer

FloatBuffer

IntBuffer

LongBuffer

ShortBuffer

Writing Data to a Buffer and Reading Data from a Buffer

▶ **There are two ways you can write/read data to/from a Buffer:**

- By means of Channel:

```
int bytesRead = inChannel.read(buf); // read into buffer
int bytesWritten = inChannel.write(buf); // read from buffer into
channel
```

- Via the buffer's `put()` and `get()` methods:

```
buf.put(127); // write data into a buffer
byte aByte = buf.get();  // read data from a buffer
```

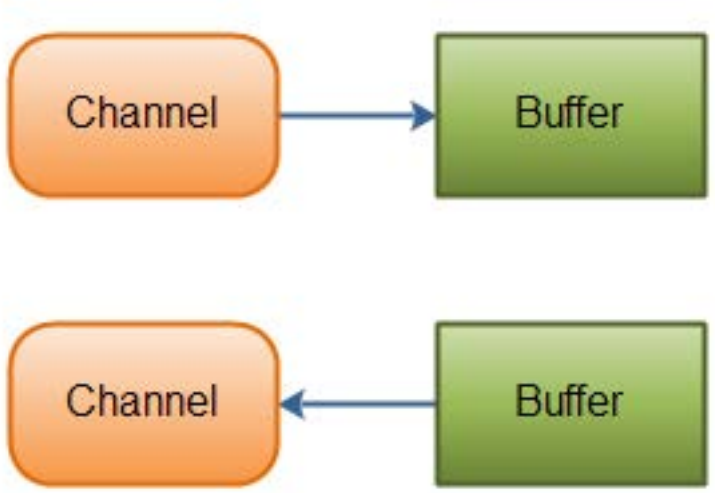
Buffer demos

- BufferDemo
- BufferFlipDemo
- BufferMarkDemo

Java NIO Channel

Channels are similar to streams with a few differences:

- You can both read and write to Channels. Streams are typically one-way (read or write).
- Channels can be read and written asynchronously.
- Channels always read to or write from a Buffer.

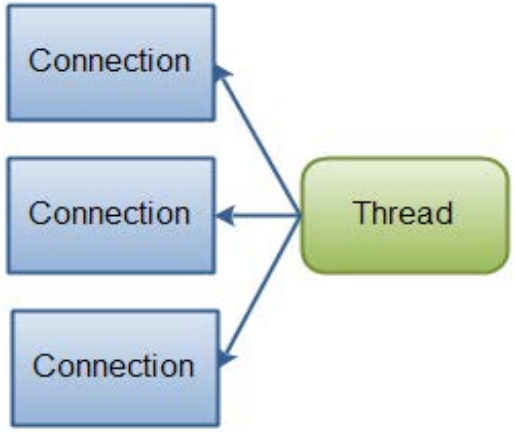


When should I use IO and when should I use NIO?

- **NIO's multiple channels managing using a single thread vs. simple data reading from a blocking stream.**

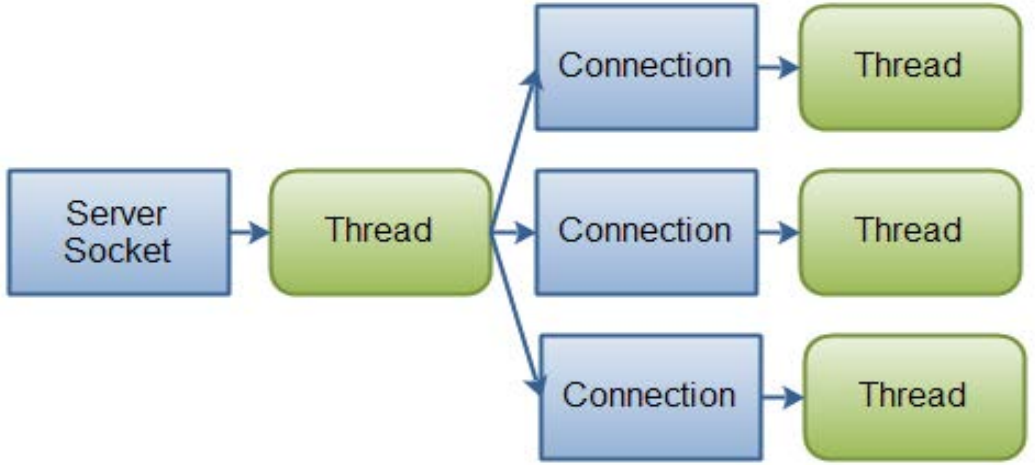
NIO

- Manage thousands of open connections simultanously, while each of them sends a little data (**chat server**).
- Keep a lot of open connections to other computers (**P2P network**).



IO

- A few connections with very high bandwidth, sending a lot of data at a time.



Java NIO Channel

▶ Channel implementations:

- The **FileChannel** reads data from and to files.
- The **DatagramChannel** can read and write data over the network via UDP.
- The **SocketChannel** can read and write data over the network via TCP.
- The **ServerSocketChannel** allows you to listen for incoming TCP connections, like a web server does.

Opening a channel

```
SeekableByteChannel in = FileChannel.open(
        Paths.get("file.txt"), StandardOpenOption.READ);

SeekableByteChannel out = FileChannel.open(Paths.get("file.txt"),

        StandardOpenOption.WRITE, StandardOpenOption.CREATE);
```

Other StandardOpenOption:

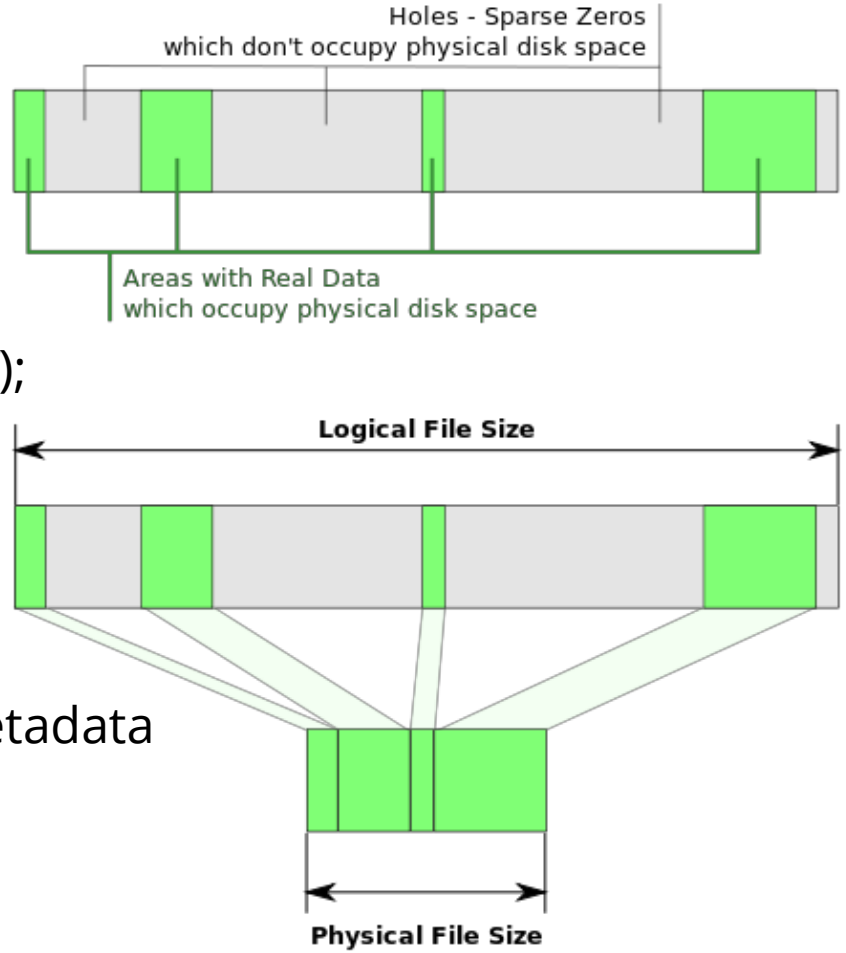
- APPEND
- TRUNCATE_EXISTING
- CREATE_NEW
- DELETE_ON_CLOSE

SYNC – sync content & metadata

DSYNC – sync content

SPARSE

Sparse file



NIO Channel demos

- ChannelCopyDemo
- ChannelTransferDemo
- ChannelServer+ChannelClient

Direct buffer

NIO supports a type of ByteBuffer usually known as a *direct* buffer. Direct buffers can essentially be used like any other ByteBuffer, but have the property that their underlying memory is **allocated outside the Java heap**.

- once allocated, their memory address is fixed for the lifetime of the buffer
- because their address is fixed, the kernel can safely access them directly and hence direct buffers can be used more efficiently in I/O operations;
- in some cases, accessing them from Java can be more efficient

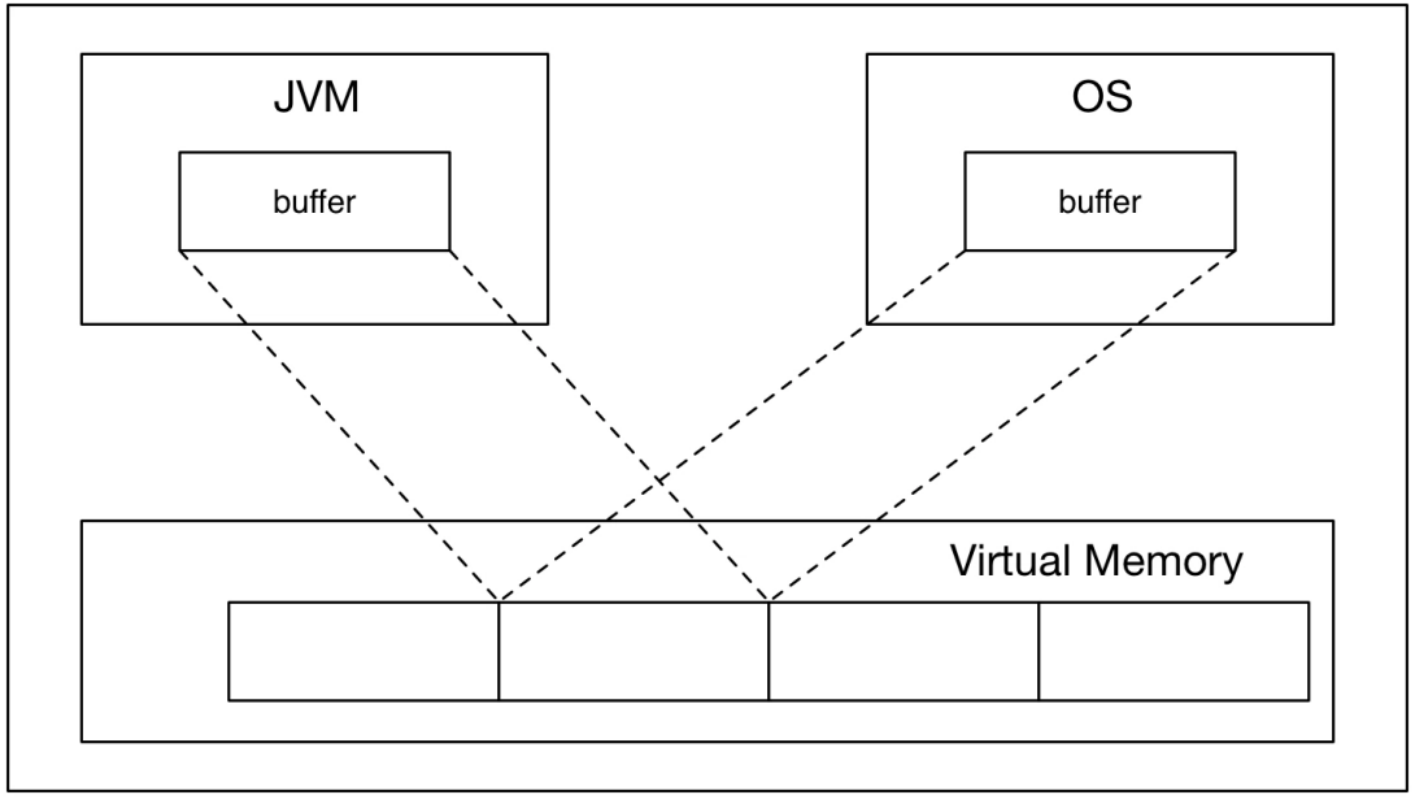
```
ByteBuffer directBuf =
        ByteBuffer.allocateDirect(noBytes);
```

In general, direct buffers are best suited to cases where:

- you're creating a relatively restricted number of buffers that will be relatively long-lived;
- performance is crucial, and you're reasonably sure that using a direct buffer will have a performance gain

Mapped buffer

MappedByteBuffer directly map with open file in Virtual Memory by using map method in FileChannel. The MappedByteBuffer object work like buffer but its data stored in a file on Virtual Memory.



Mapped file benefits

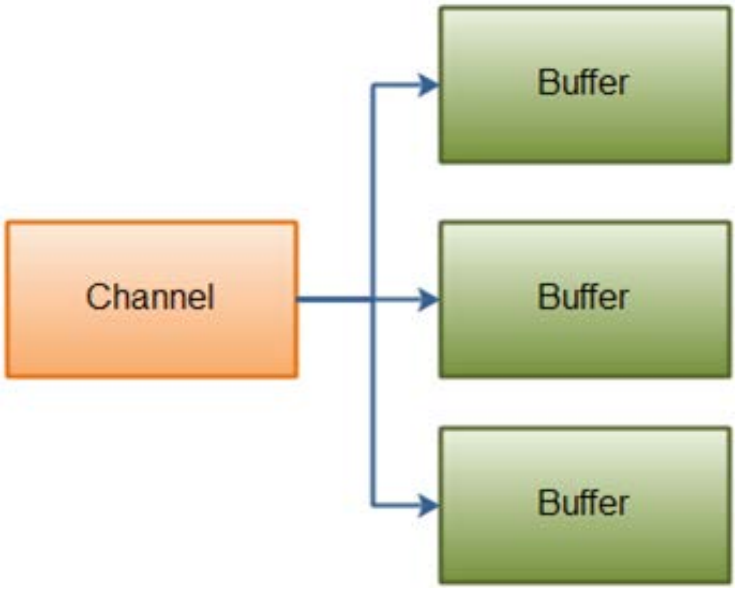
MemoryMappedFileReader
MemoryMappedFileWriter

1.The JVM directly process on Virtual Memory hence it will avoid system read() and write() call.
2.JVM doesn’t load file in its memory besides it uses Virtual Memory that bring the ability to process large data in efficient manner.
3.OS mostly take care of reading and writing from shared VM without using JVM
4.It could be used more than one process based on locking provided by OS. We will be discussing locking later.
5.This also provides ability map a region or part of file.
6.The file data always mapped with disk file data without using buffer transfer.

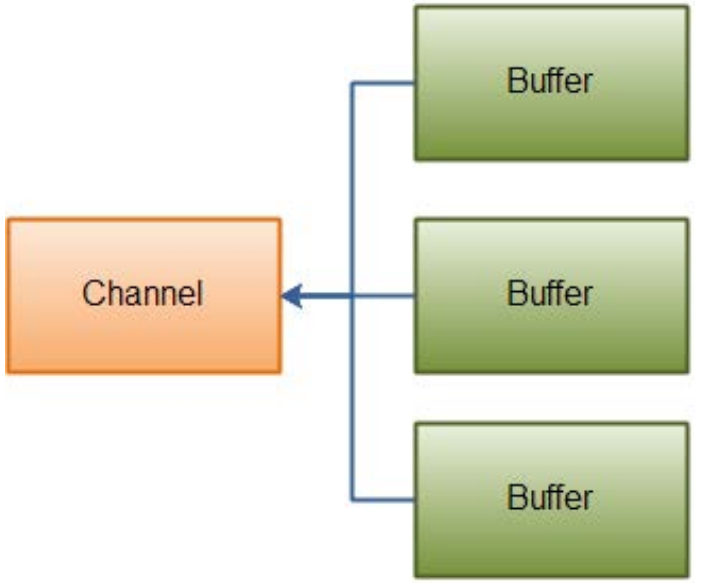
ScatterGatherChannelDemo

Scattering and gathering channels

- In Java NIO the channel provides an important capability known as scatter/gather or vectored I/O. It is a simple yet powerful technique through which the bytes can be written from a set of buffers to a stream using a single write() function and bytes can be read from a stream into a set of buffers using a single read() function.



The **'scattering read'** is used for reading the data from a single channel into multiple buffers.

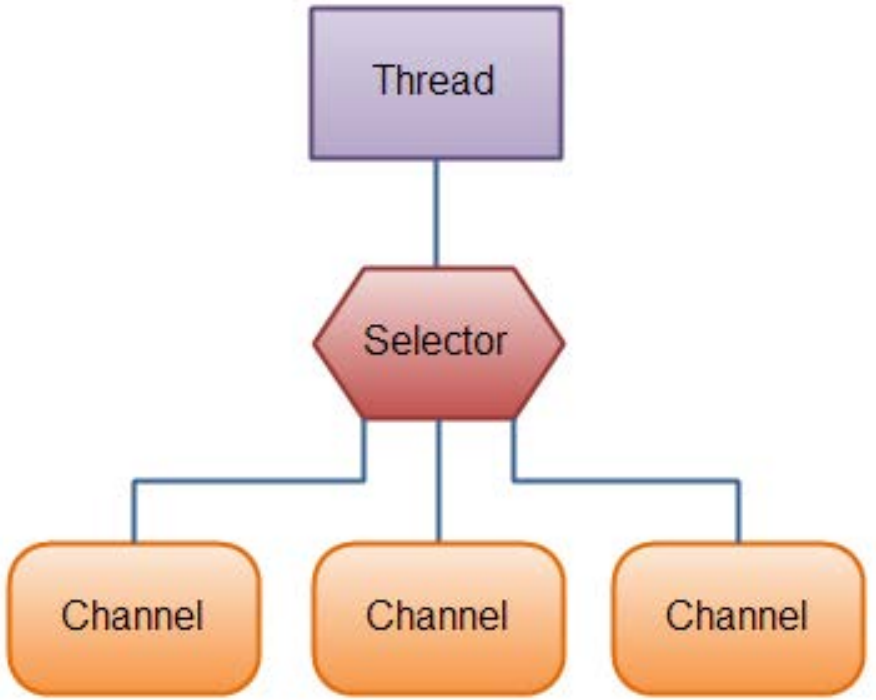


The **'gathering write'** is used for writing the data from a multiple buffers into a single channel.

Java NIO Selector

- Component that can examine one or more NIO Channel's, and determine which of them are ready for reading or writing.

Via Selector a single thread can manage multiple channels, and thus multiple network connections.



Java NIO Selector

▶ Using a Channel with a Selector.

- **Creating a Selector**:

```
Selector selector = Selector.open();
```

- **Registering Channels with the Selector:**

```
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```

// The Channel must be in non-blocking mode to be used with a Selector.

SelectionKey

- **The register() method returns a SelectionKey object with following properties:**

The interest set: `int interestSet = selectionKey.interestOps();`

- `selectionKey.OP_CONNECT`
- `selectionKey.OP_ACCEPT`
- `selectionKey.OP_READ`
- `selectionKey.OP_WRITE`

The ready set: `int readySet = selectionKey.readyOps();`

- `selectionKey.isAcceptable();`
- `selectionKey.isConnectable();`
- `selectionKey.isReadable();`
- `selectionKey.isWritable();`

Channel + Selector

- `Channel  channel  = selectionKey.channel();`
- `Selector selector = selectionKey.selector();`

An attached object (optional)

Selecting Channels via Selector

- **Once you have registered one or more channels with a Selector, you can call one of the select() methods:**

- **`int select()`** // blocks until at least one channel is ready for the events you registered for
- **`int select(long timeout)`** // does the same as select() except it blocks for a maximum of timeout milliseconds (the parameter)
- **`int selectNow()`** // doesn't block at all, returns immediately whatever channels are ready

The int returned by the select() methods tells how many channels are ready between each select() call.

Accessing Ready Channels

- **Once you have called one of the select() methods, you can access ready channels via the "selected key set", by calling selectedKeys() method:**

You can iterate selected key set to access ready channels:

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
while(keyIterator.hasNext()) {
    SelectionKey key = keyIterator.next();
    if(key.isAcceptable()) {   // a connection was accepted by a ServerSocketChannel
    } else if (key.isConnectable()) {   // a connection was established with a remote server
    } else if (key.isReadable()) {   // a channel is ready for reading
    } else if (key.isWritable()) {   // a channel is ready for writing
    }
    keyIterator.remove();
}
```

NIO Selector demos

- SelectorClient/SelectorServer

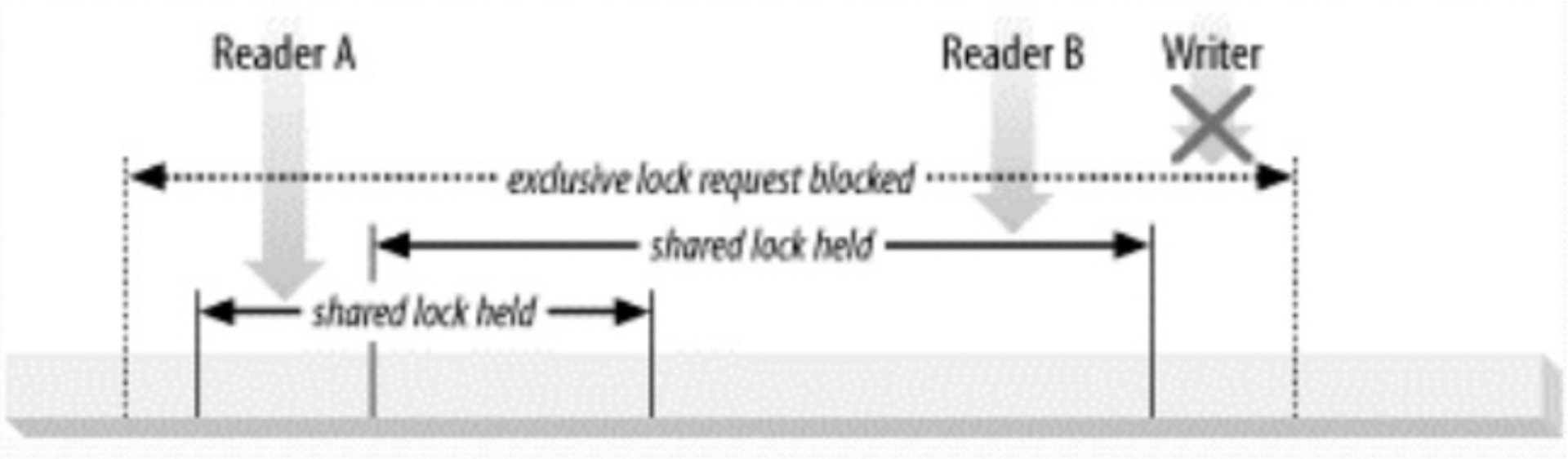
Asynchronous I/O

AsyncFileChannelDemo
AsyncFileHandlerDemo
AsyncClient/AsyncServer
AsyncToCompletableFuture

Asynchronous Channels:

- AsynchronousFileChannel
- AsynchronousServerSocketChannel
- AsynchronousSocketChannel

File lock

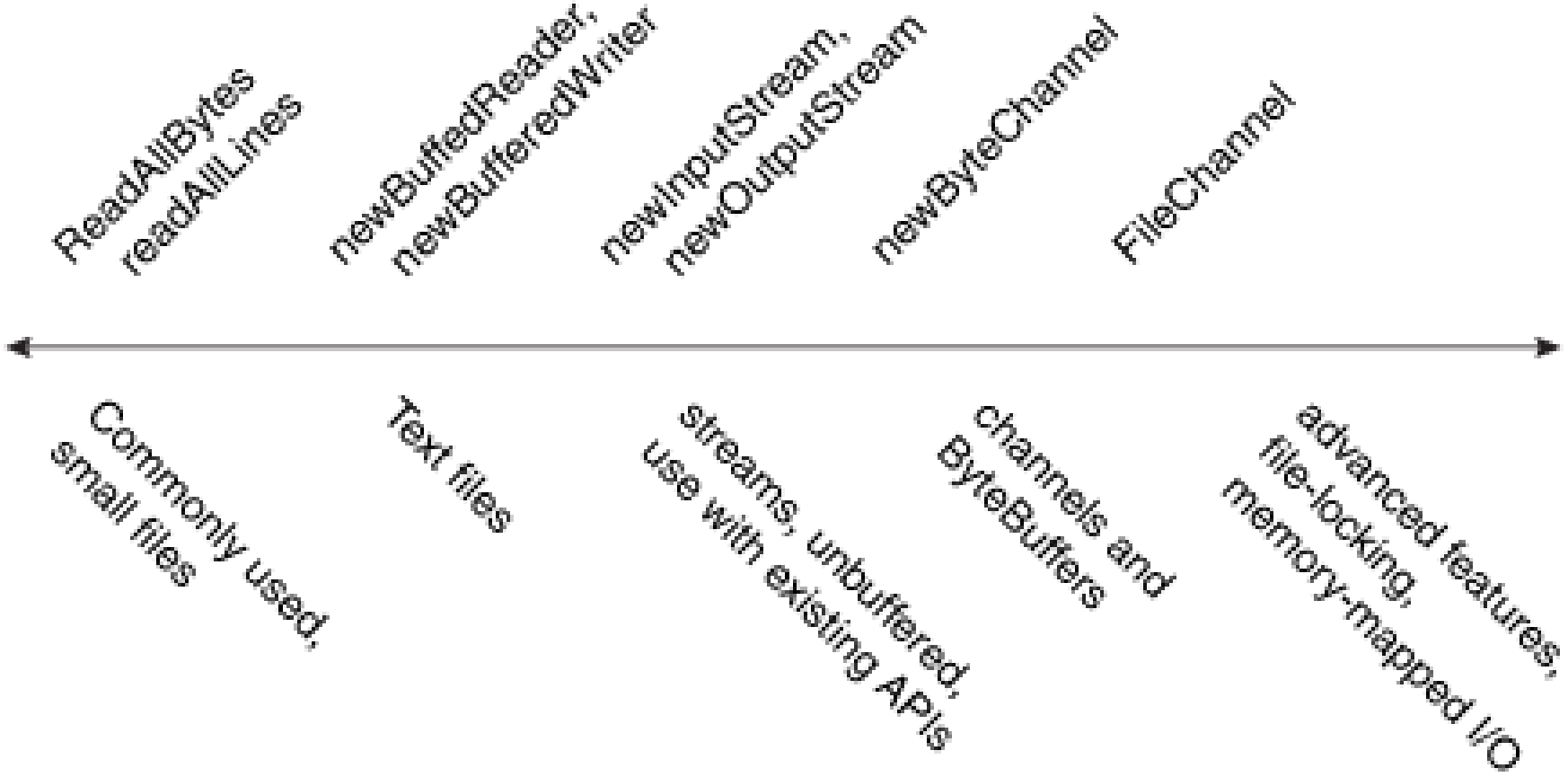


shared lock *prevents other concurrently-running programs from acquiring an overlapping exclusive lock, but does allow them to acquire overlapping shared locks*

- FileLock lock()
- FileLock lock(long position, long size, boolean shared)

- FileLock tryLock()
- FileLock tryLock(long position, long size, boolean shared)

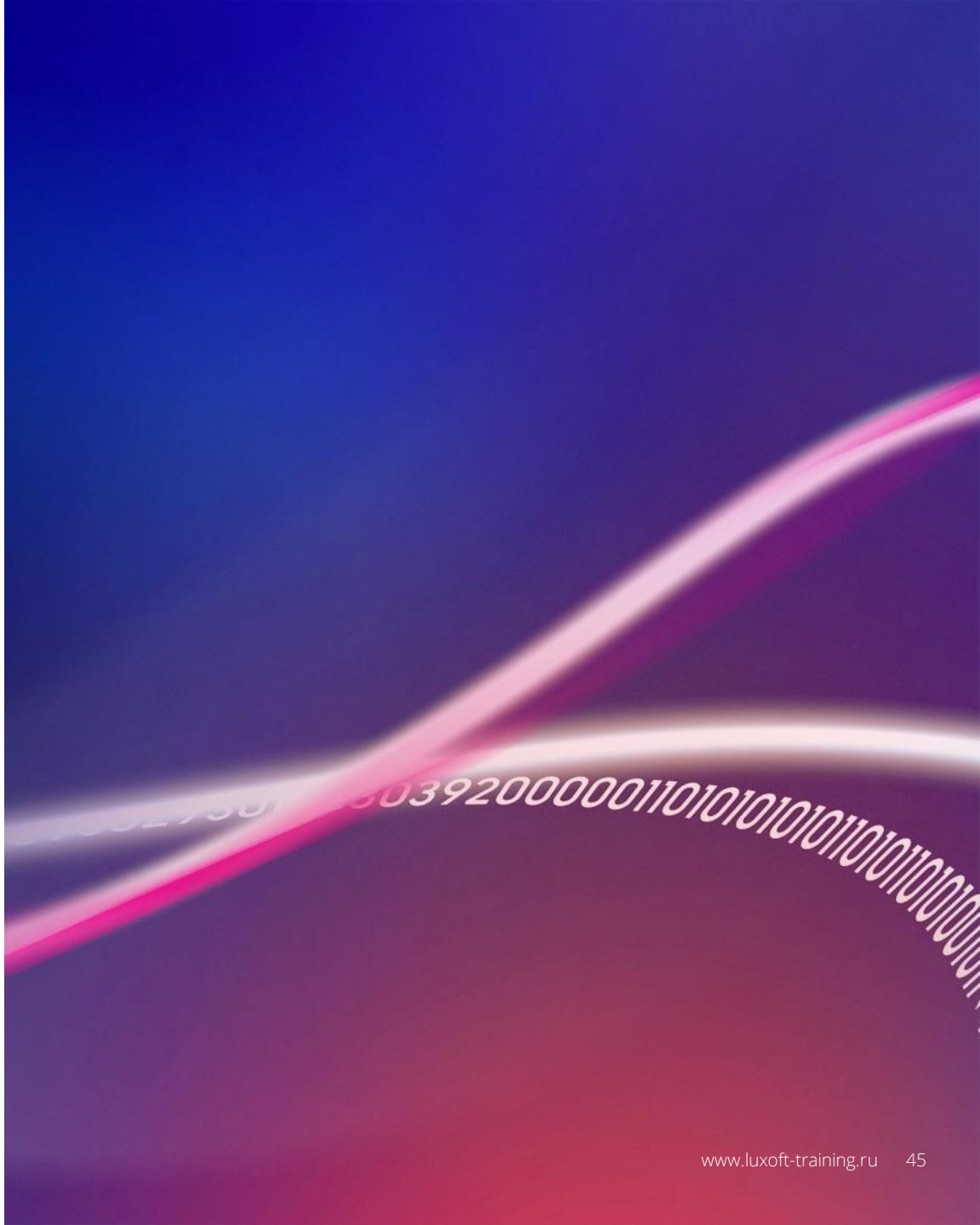
Working with files in NIO



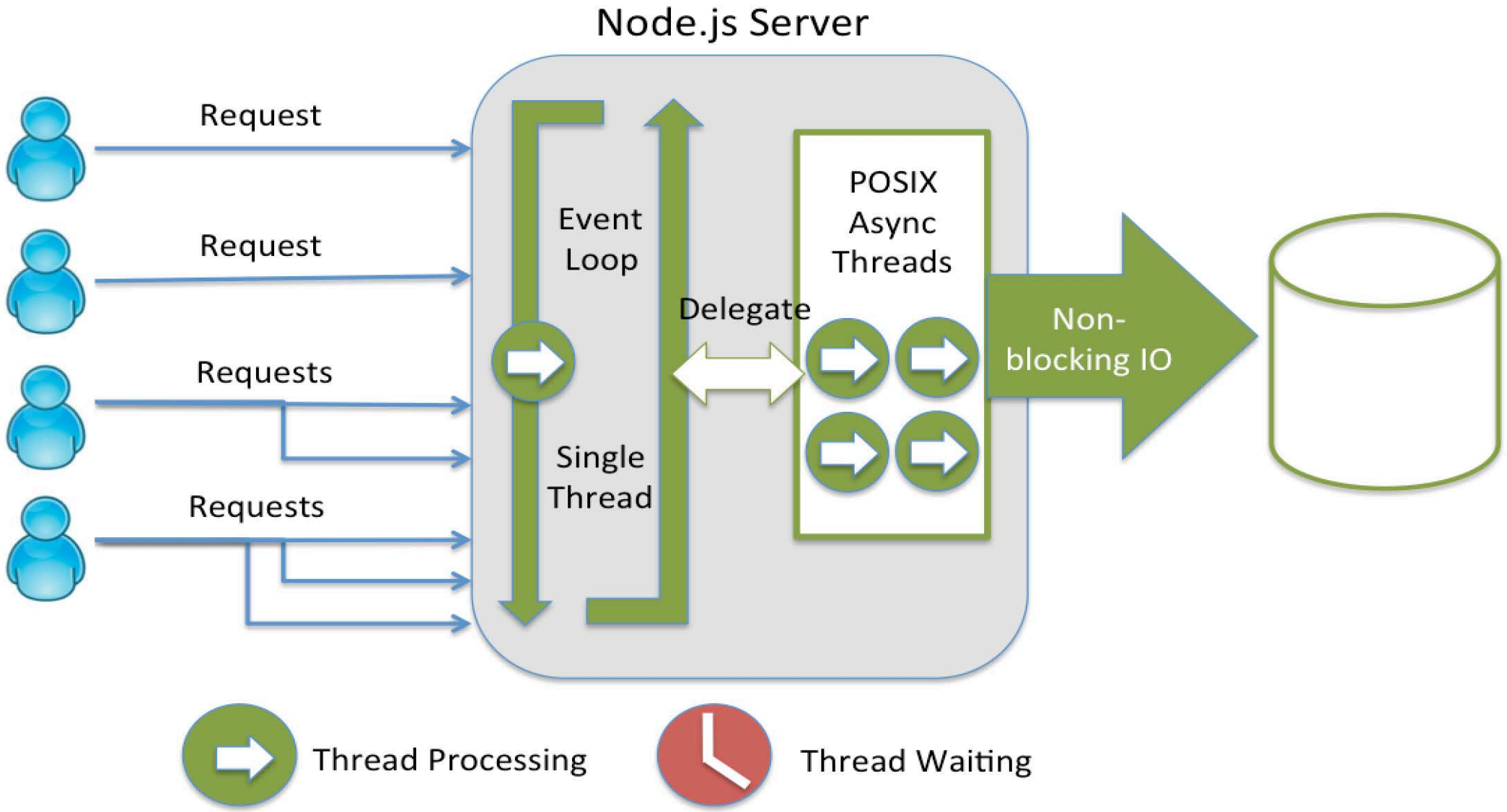
SmallFileReadWrite – simple API to read/write files

CopyBenchmark – measurement of speed of various ways to copy a file

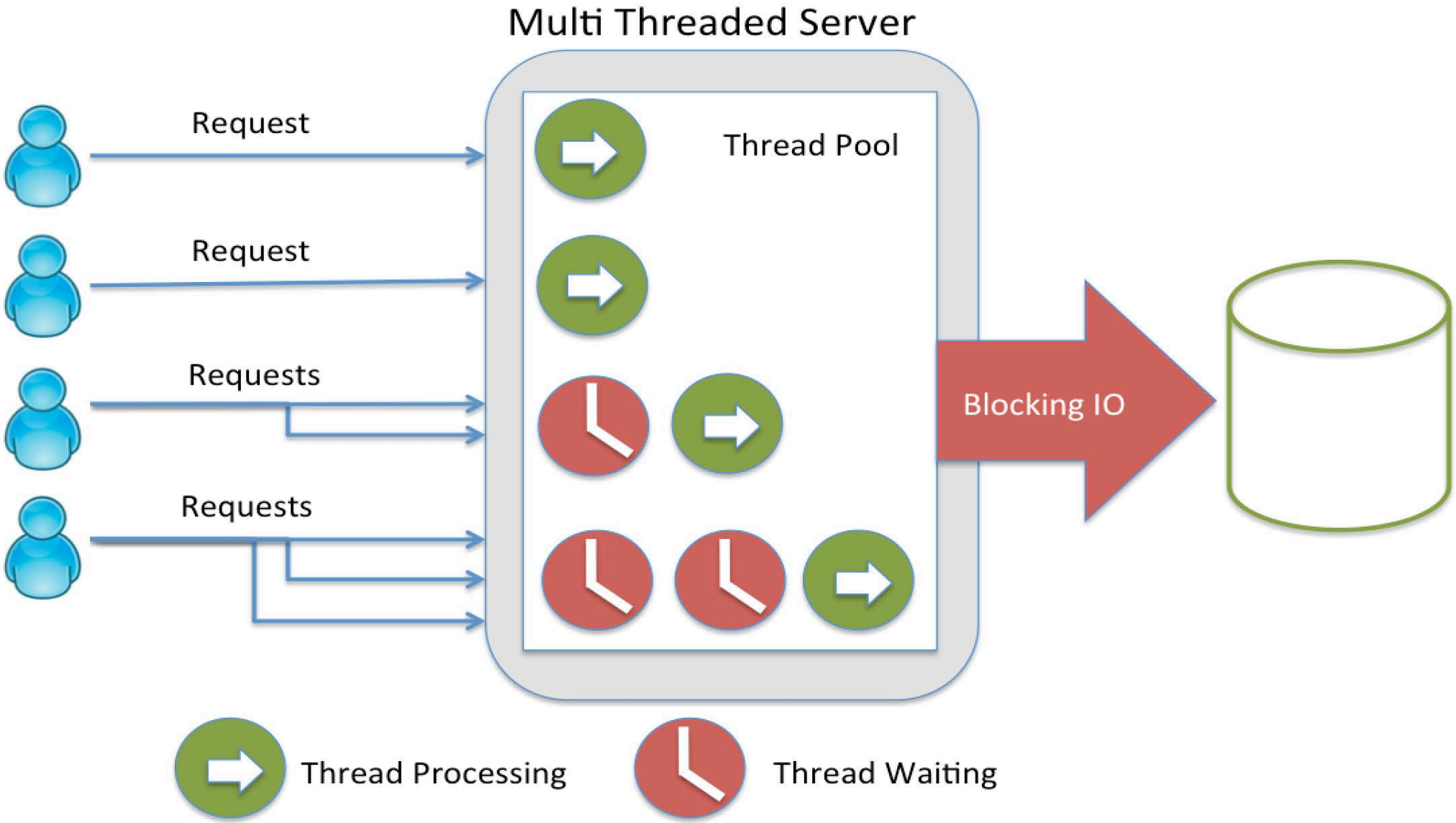
Production-ready NIO server



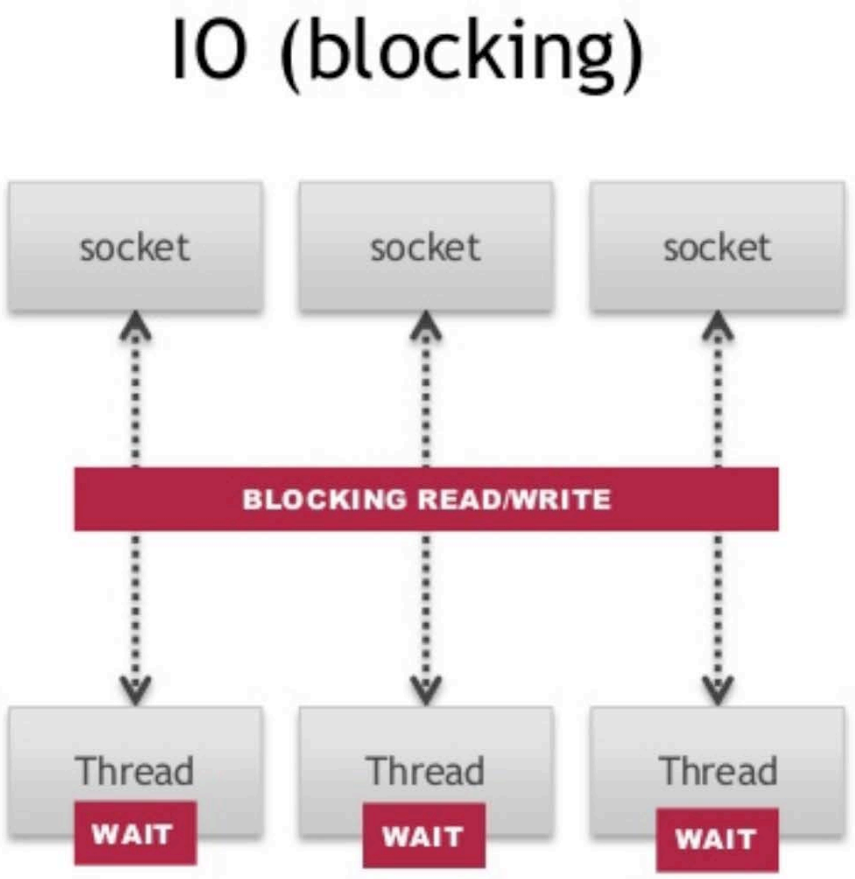
Node.js – pioneer in non-blocking I/O

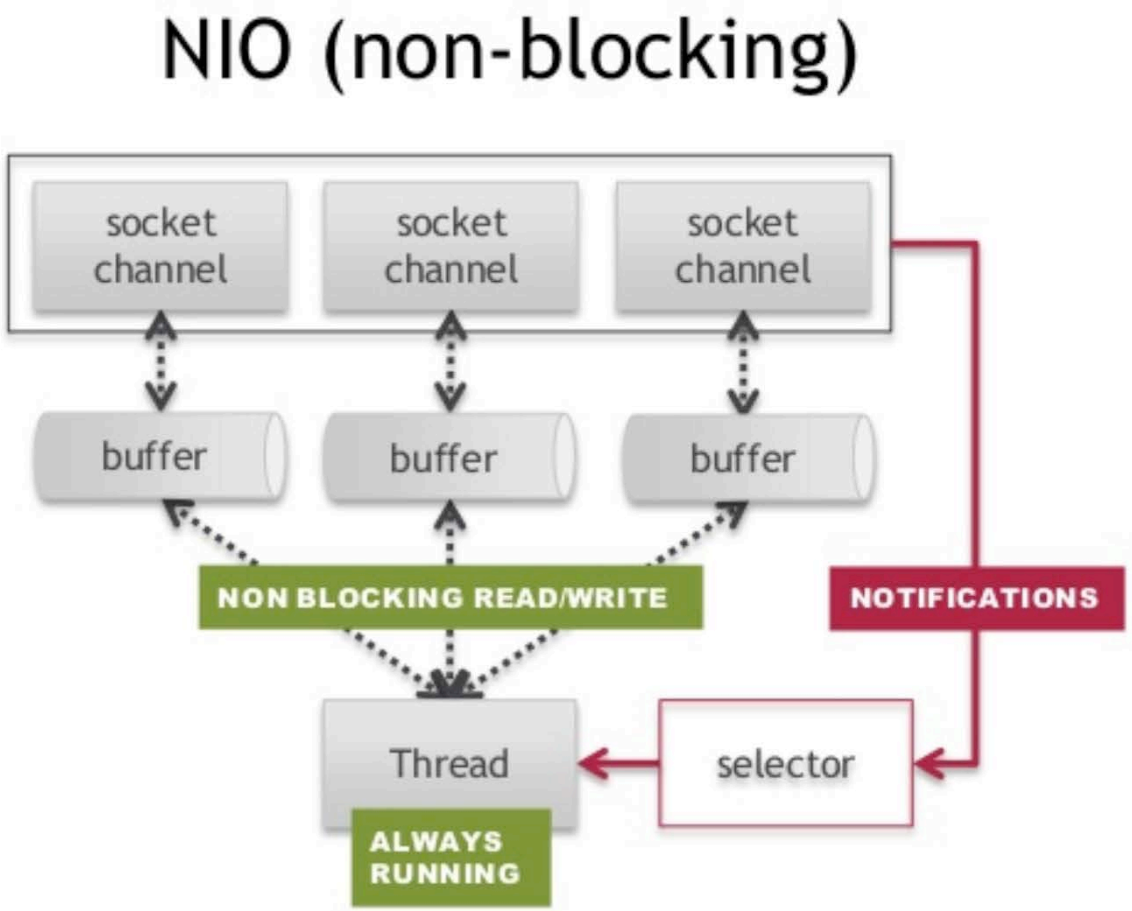


Blocking Java multi-threaded server (for example, Tomcat)

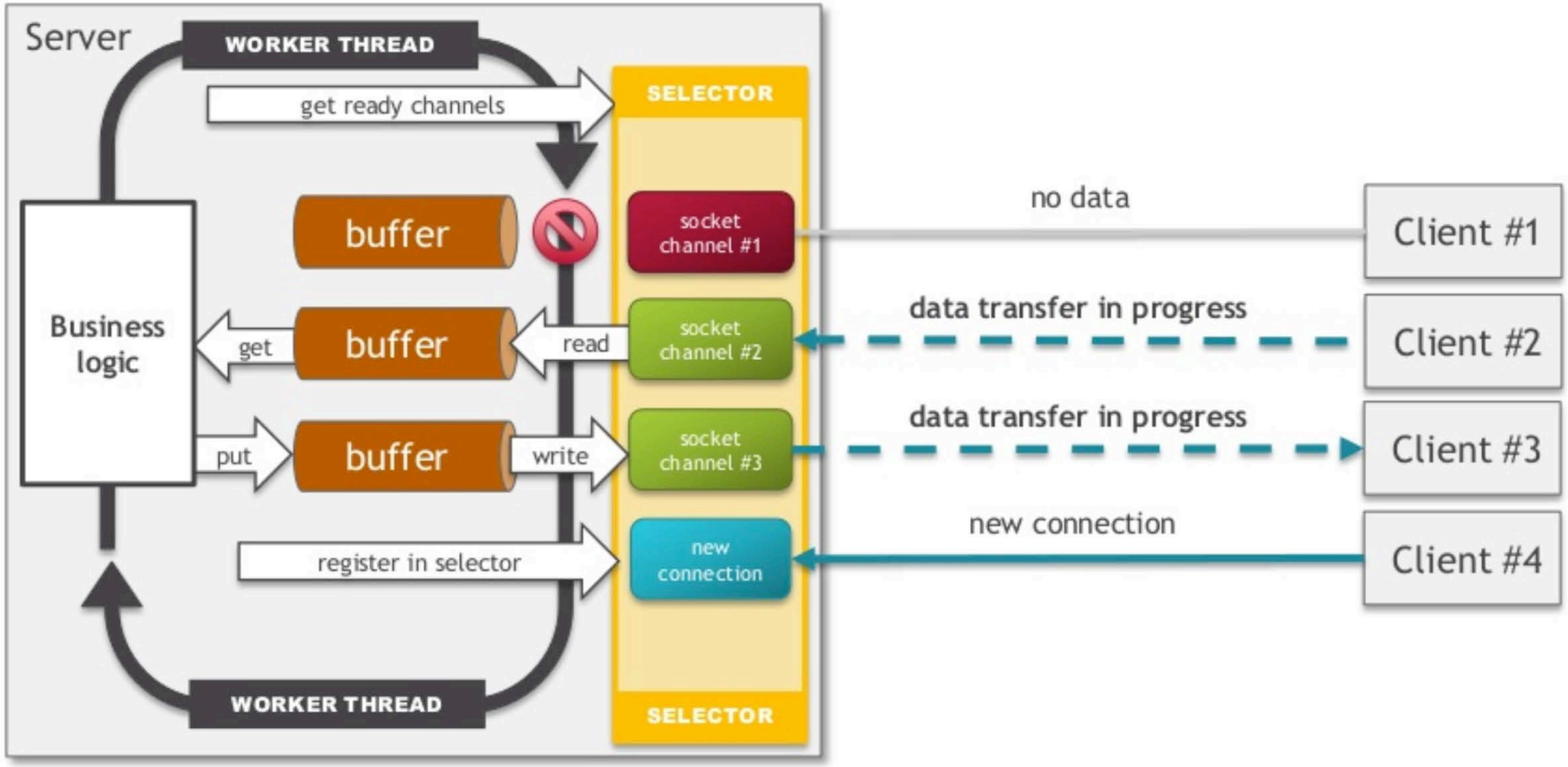


Blocking vs. Non-blocking server

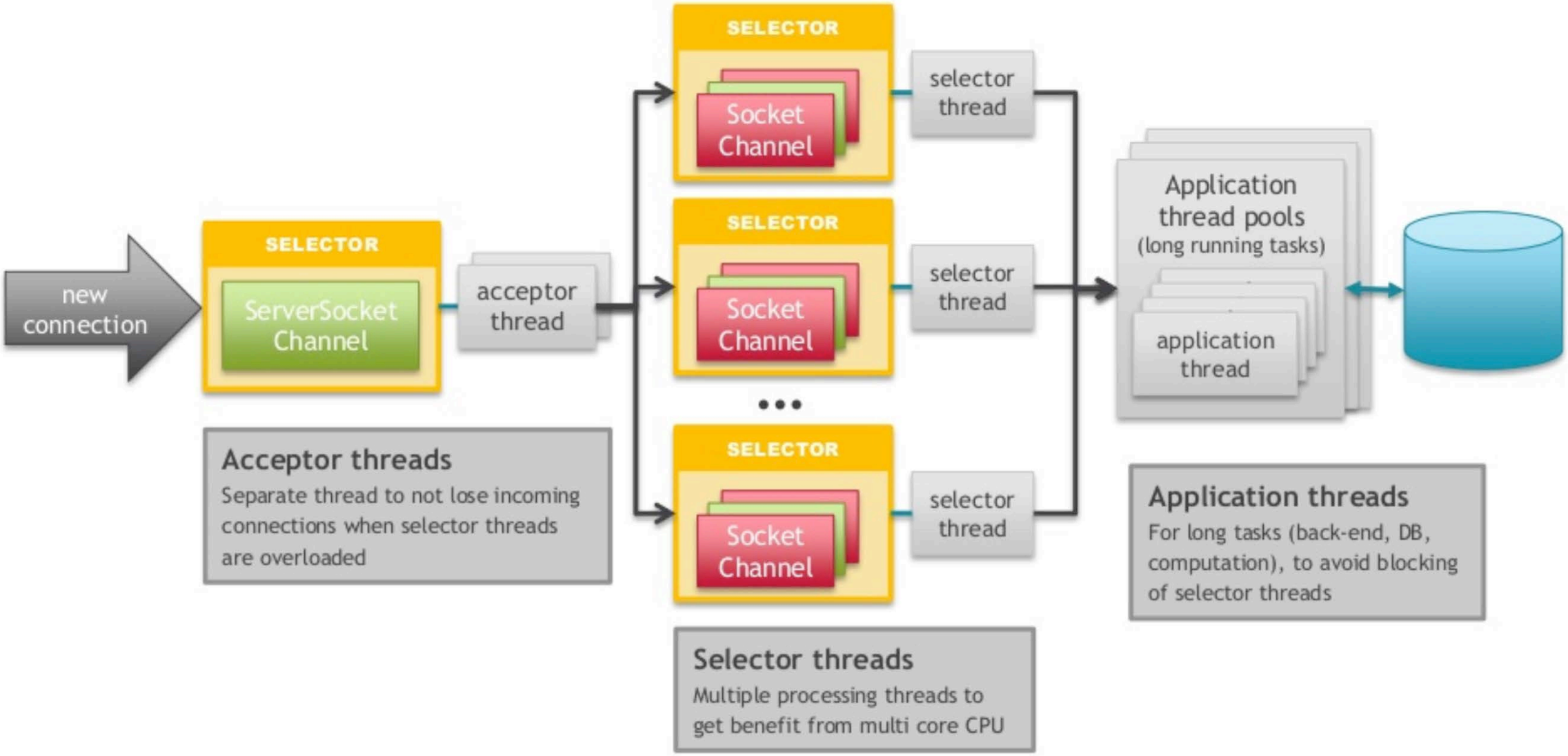




NIO Server architecture



Production-ready NIO server architecture



Netty server

- Netty is a **high performance, open source** NIO server.
- Netty **runs embedded** in your Java applications.
- Netty uses a **single-threaded concurrency model**, and is designed around **non-blocking IO**. This results in a significantly **different programming model** than when implementing Servlet applications.

Companies using Netty:

Open source projects using Netty:

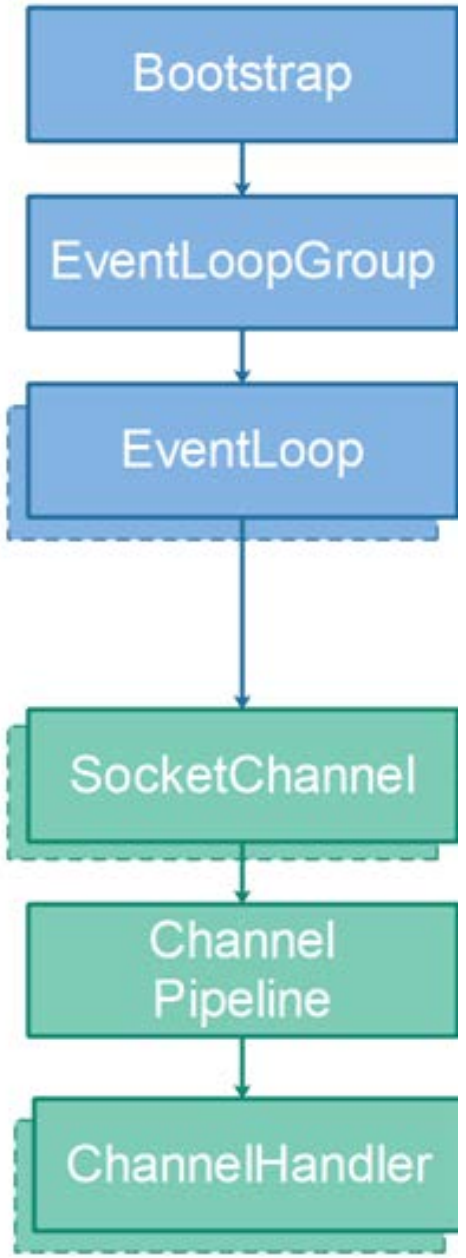
Netty ChannelFuture

- Netty's channel operations are asynchronous; i.e. they return immediately without a guarantee that it completed. The most logical question at this point is, "**How to I ensure that something runs after this task completes and not before?**" To make this possible, all asynchronous methods in Netty return a **ChannelFuture** instance. This class has methods to pipe other tasks upon completion to ensure tasks execute one after the other, but in a non-blocking manner.

```
ByteBuf buf = ctx.alloc().buffer(4);
ChannelFuture f = ctx.writeAndFlush(buf);
f.addListener(new ChannelFutureListener() {
   @Override
   public void operationComplete(ChannelFuture future) {
      assert f == future;
      ctx.close();
   }
});
```

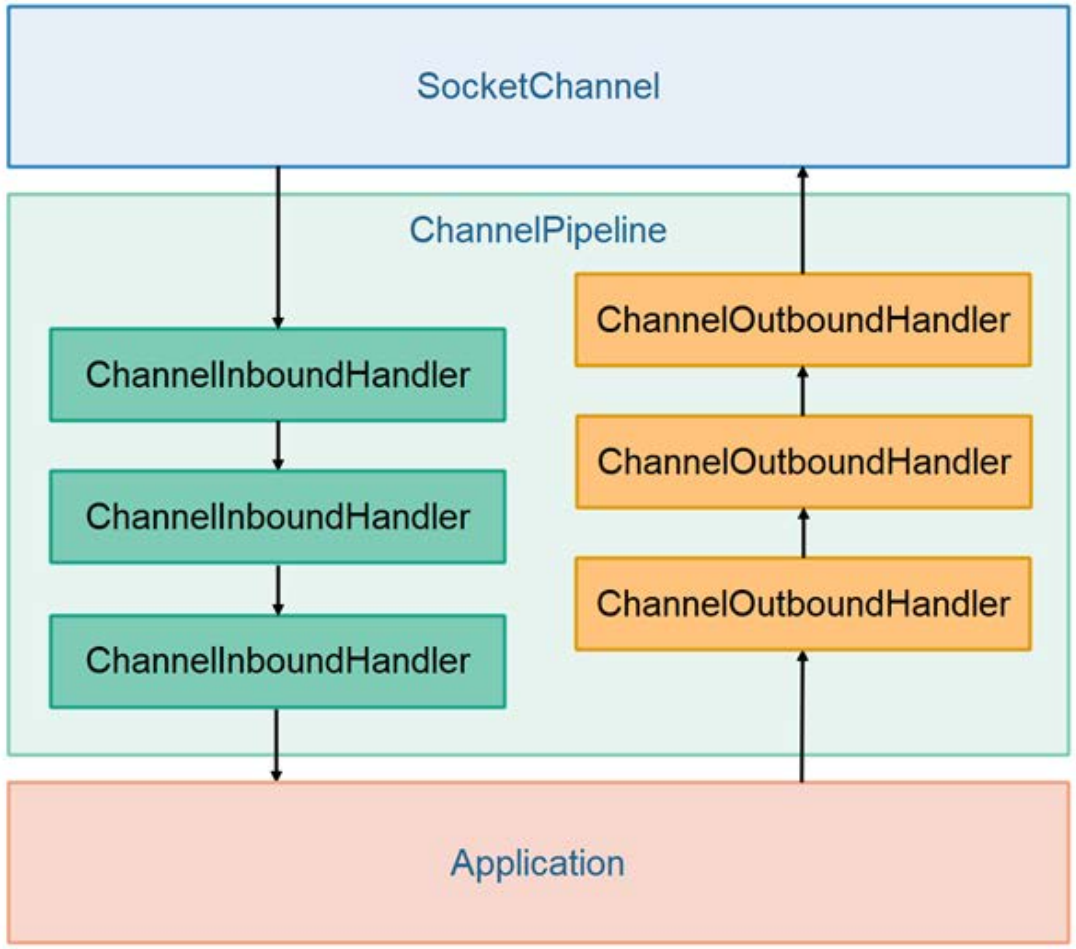
Netty running

- The **Bootstrap** classes in Netty take care of bootstrapping Netty. The bootstrapping process includes **starting threads, opening sockets etc.**
- A Netty **EventLoopGroup** is a group of **EventLoop**'s . Multiple EventLoop's can be grouped together. This way the EventLoop shares some resources like threads etc.
- A Netty **EventLoop** is a loop that keeps looking for new events, e.g. incoming data from network sockets (from SocketChannel) instances). When an event occurs, the event is passed on to the appropriate event handler, for instance a **ChannelHandler**.
- Each Netty **SocketChannel** has a **ChannelPipeline**. The ChannelPipeline contains a **list of ChannelHandler** instances.



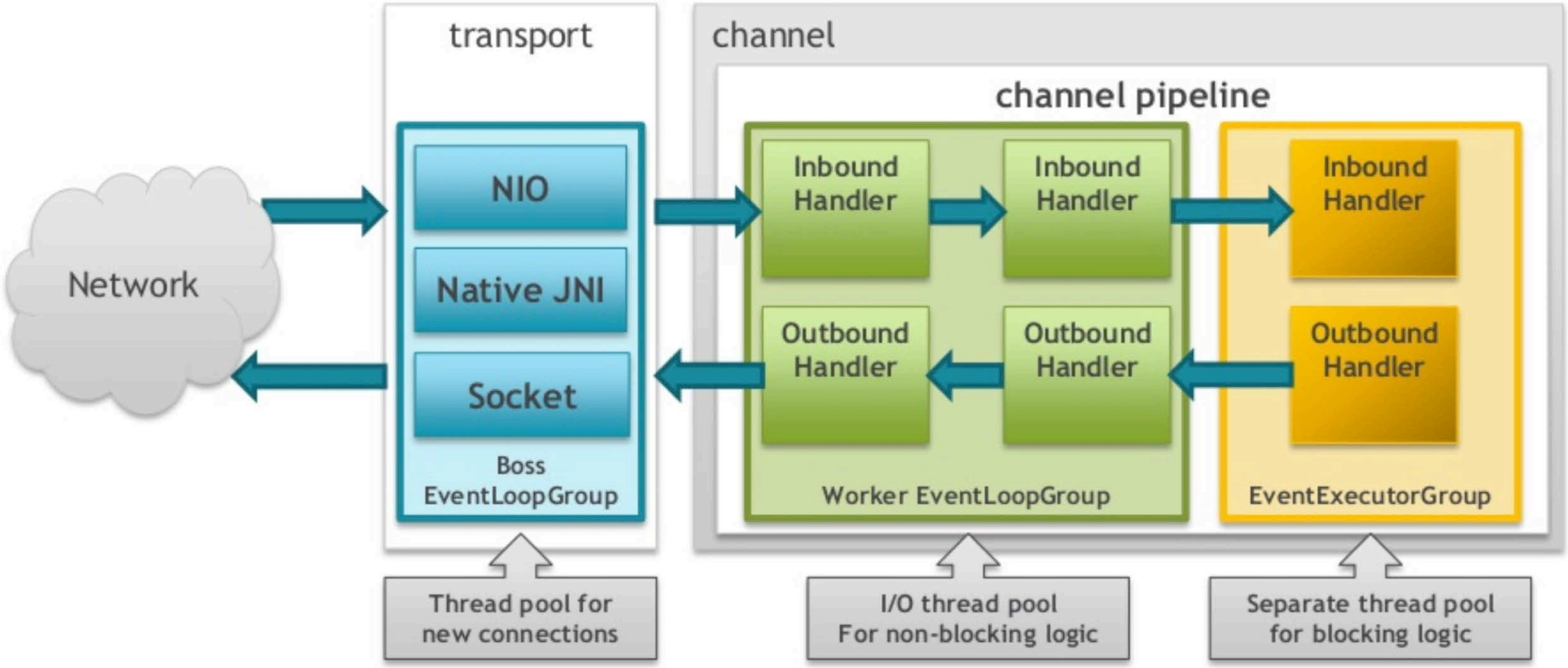
Netty ChannelPipeline

- Netty uses **Pipelines** to handle data that passes through servers created using it



There are two main types of handlers in Netty, which are **Channel Inbound Handlers**, and **Channel Outbound Handlers.** The Inbound handlers are used to handle **inbound traffic,** i.e. from Socket Channel to Application. Similarly, the Outbound Handlers handle **outbound traffic,** i.e. from Application to Socket Channel.

Netty server architecture



Netty server demos

- NettyServer
- NettyTimeServerHandler
- NettyTimeClient
- NettyTimeClientHandler

Java NIO Path & Java NIO File

java.io.File class

- The File class in the Java IO API gives you access to the underlying file system. You can:

Check if a file or directory exists:

```
File file = new File("c:\\data\\input-file.txt");

boolean fileExists = file.exists();
```

Create a directory if it does not exist:

```
boolean dirCreated = file.mkdir();
```

Read the length of a file:

```
long length = file.length();
```

java.io.File class

- **The File class in the Java IO API gives you access to the underlying file system. You can:**

Rename or move a file:

```
boolean success = file.renameTo(new File("c:\\data\\new-file.txt"));
```

Delete a file:

```
boolean success = file.delete();
```

Check if path is file or directory:

```
boolean isDirectory = file.isDirectory();
```

Read list of files in a directory:

```
String[] fileNames = file.list();
File[] files = file.listFiles();
```

java.io.File class

- **Prior to the Java SE 7 release, the java.io.File class was the mechanism used for file I/O, but it had several drawbacks:**

Many methods **didn't throw exceptions** when they failed, so it was impossible to obtain a useful error message.

The **rename method didn't work** consistently across platforms.

There was **no real support for symbolic links**.

More support for metadata was desired, such as file permissions, file owner, and other security attributes.

Accessing file metadata was **inefficient**.

Many of the File **methods didn't scale**.

java.nio.file.*

- **New API to work with file system.**

Starts from Java SE 7 release.

Covers all java.io.File functionality.

Self-consistent API, more advanced features (work with links, metadata support, changes monitoring).

java.nio.file.Path is its core entity.

java.nio.file.Path class

- **A Java Path instance represents a *path* in the file system.**

  - A path can point to either **a file or a directory.**
  - A path can be **absolute or relative.**

  Creating a Path Instance

```
Path path = Paths.get("c:\\data\\myfile.txt"); // creating an absolute path
Path projects = Paths.get("d:\\data", "projects"); // creating a relative path

Path currentDir = Paths.get("."); // creating a relative path to current directory
System.out.println(currentDir.toAbsolutePath());

Path parentDir = Paths.get(".."); // creating a relative path to parent directory
```

Path.normalize()

- **The normalize() method of the Path interface can normalize a path.**

Normalizing means that it removes all the . and .. codes in the middle of the path string, and resolves the referred path.

```
String originalPath = "d:\\data\\projects\\a-
project\\..\\another-project";
Path path1 = Paths.get(originalPath);
System.out.println("path1 = " + path1);
Path path2 = path1.normalize();
System.out.println("path2 = " + path2);

// path1 = d:\data\projects\a-project\..\another-project
// path2 = d:\data\projects\another-project
```

Interoperability With Legacy Code

- **Perhaps you have legacy code that uses java.io.File and would like to take advantage of the java.nio.file.Path functionality with minimal impact to your code:**

The **toPath()** method converts java.io.File instance to a java.nio.file.Path:

```
Path input = file.toPath();
```

File metadata

- **BasicFileAttributeView**: This is a view of basic attributes that must be supported by all file system implementations. The attribute view name is basic.
- **DosFileAttributeView**: This view provides the standard four supported attributes on file systems that support the DOS attributes. The attribute view name is dos.
- **PosixFileAttributeView**: This view extends the basic attribute view with attributes supported on file systems that support the POSIX (Portable Operating System Interface for Unix) family of standards, such as Unix. The attribute view name is posix.
- **FileOwnerAttributeView**: This view is supported by any file system implementation that supports the concept of a file owner. The attribute view name is owner.
- **AclFileAttributeView**: This view supports reading or updating a file's ACL. The NFSv4 ACL model is supported. The attribute view name is acl.
- **UserDefinedFileAttributeView**: This view enables support of metadata that is user defined.

FileVisitor Interface

- preVisitDirectory – Invoked before a directory's entries are visited.
- postVisitDirectory – Invoked after all the entries in a directory are visited. If any errors are encountered, the specific exception is passed to the method.
- visitFile – Invoked on the file being visited. The file's BasicFileAttributes is passed to the method, or you can use the file attributes package to read a specific set of attributes. For example, you can choose to read the file's DosFileAttributeView to determine if the file has the "hidden" bit set.
- visitFileFailed – Invoked when the file cannot be accessed. The specific exception is passed to the method. You can choose whether to throw the exception, print it to the console or a log file, and so on.

Other demos

- Temporary – creating temporary files & folders
- WatchDemo – possibility to watch at directory changes

Thank You!

think.
create.
accelerate.