Improving NIO 2 (and Tomcat)

How NIO 2 was used in Apache Tomcat ™

Rémy Maucherat

Software engineer at Red Hat

Working on Red Hat JBoss Web Server

Apache Tomcat committer since 2000

ASF member

Contents

NIO 2 Primer

Overview

Brand new API and IO code introduced in Java 7

Internally non blocking API like NIO

No complex socket polling code to write

Exposes blocking and async semantics

Fancy symmetrical API

Async reads / writes

Non blocking read and write operations

Framework will call a completion handler once operation is done

Pending exception on any concurrent operation

Blocking reads / writes

Non blocking read and write operations that return a future object

Future allows blocking until the operation is complete

Pending exception on any concurrent operation

Vectored IO

Single async IO operation can read / write to / from multiple buffers

Only one OS call

Complex to use

Optimal with fixed length binary protocols

Performance

API is GC intensive ...

... And feels high level

But actually performance is great !

How does it work ? Magic ? Will see about that later ...

SSL with NIO 2

Uses SSL engine

Same SSL engine as for NIO

Not compatible with legacy JSSE, like NIO

Slightly different use patterns than NIO, new bugs to be found !

More complex than NIO

Effectively NIO 2 is two separate read / write APIs: async calls, or "blocking" calls

Twice the amount of SSL code compared to NIO

An operation may not produce network data

A network read may not produce application data

So needs recursion is some cases

Boilerplate …

Any other NIO 2 SSL implementation will duplicate tons of code

Integration needs a lot of hooks though

Should have been included

Despite difficulties, this should have been in core NIO 2

The excuse is that core NIO doesn't have it

NIO 2 SSL is harder than NIO SSL

Problems ...

State

State after an IO call is not fully known … except for Future calls

Operation could be done …

Inline ? Or not … Still done, but different sync needed

Or pending (= in progress) ? Congestion if writing or concurrent read attempt if reading

NIO 2 socket IO throws an IllegalStateException for pending

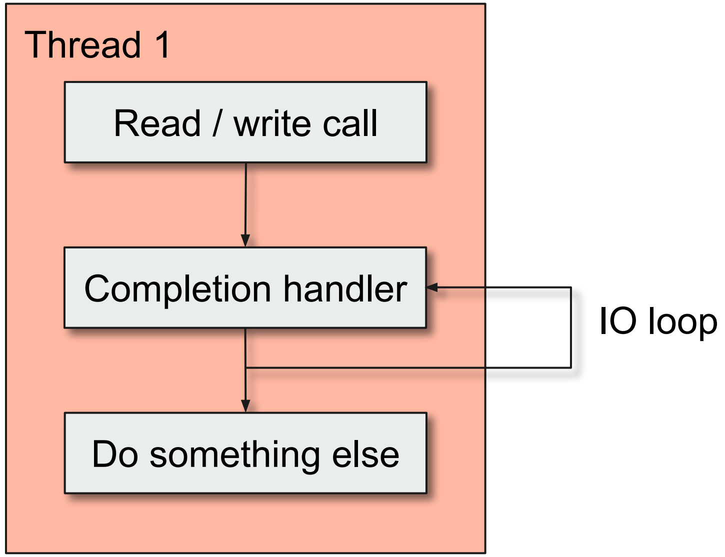
Synchronization

Made complex due to the unknown operation state

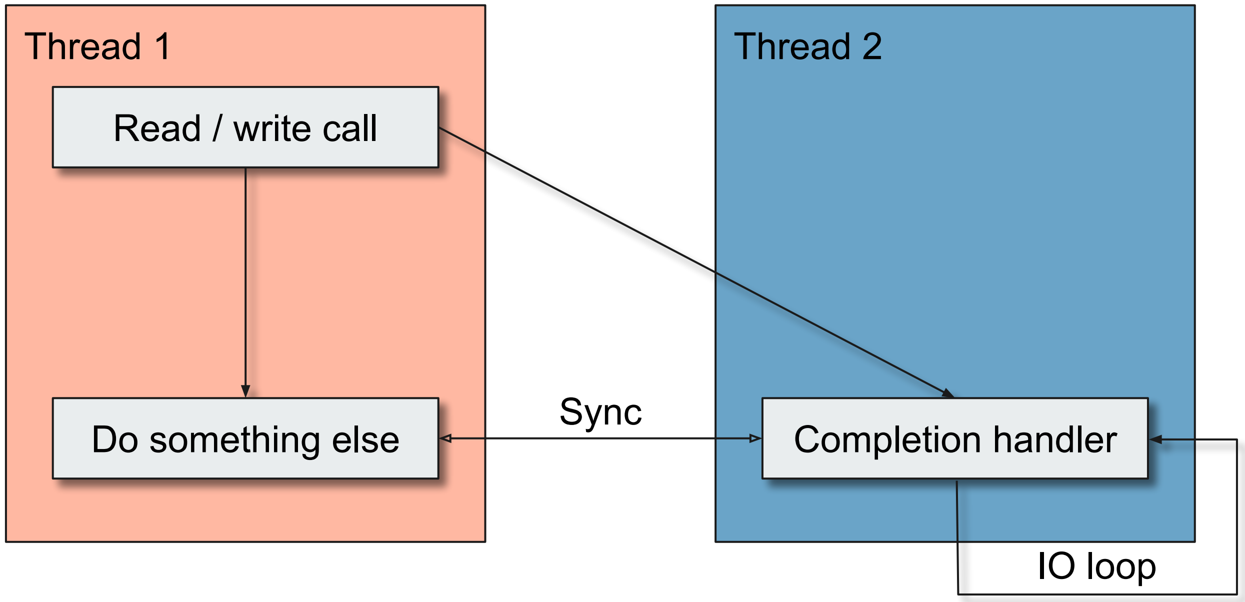
Could otherwise be optimized

Mix with IO and recursion can create deadlocks

Case 1: Inline



Case 2: Pending



Completion handler behavior

Needless completion handler calls

May force recursion from the completion handler (or loop on a get future)

Mixes IO with logic

Makes things even more complex

No vectoring for "blocking" calls

No Future<Long> write(ByteBuffer ... bb)

No Future<Long> read(ByteBuffer ... bb)

Misc

Aggressive buffering makes network error detection harder

Can also cause phantom timeouts on slow connections

NIO 2 likes owning thread pools ...

... And abuses them

Threads and buffers should be considered cheap

Extending NIO 2

More flexibility

Integration with SocketWrapper managed buffers allows interoperability

Multiple blocking modes:

- Non blocking: if an operation is pending, will return "not done"
- Blocking: the call will return only once the completion handler has been called
- Semi blocking: will block until the pending operation is complete

Operation states

Operation state returned when calling the IO operation:

- Pending: operation is not done, will not complete inline
- Not done: an operation was already pending and non blocking was used
- Inline: operation is done and completed inline
- Error: typically an IOException or a timeout
- Done: operation is done but did not complete inline

Allows differentiating scenarios and optimizing completion handler calls or synchronization

No pending exceptions

NIO 2 socket IO only has pure non blocking, no concurrency or queuing

NIO 2 socket IO produces pending exceptions on any concurrent operations

Replaced by the operation state when doing a non blocking operation

Easier more efficient code flow

Completion handler call control

Completion handler will first call a CompletionCheck object

Gets current operation state (pending or inline)

Can pass the operation state to the CompletionHandler

Continue operation or done

Utility completion checks for common cases

API

Two do it all read / write methods that extend NIO 2 vectored API

- Add BlockingMode parameter
- Add CompletionCheck parameter

Return operation state, also passed to CompletionCheck call

A flag which indicates if the API is available

See Tomcat javadoc for full details

Missed items

No Future IO calls with vectoring

Threading strategy is hard to control

Buffering strategy remains a black box

Cannot pass the CompletionCheck directly to the CompletionHandler

NIO 2 in Tomcat

API available on SocketWrapper

On internal upgrade handlers with the SocketWrapper instance

On privileged Servlet API upgrade handlers with the SocketWrapper instance

Checks NIO 2 API availability, then switches to different IO code path

Status

Used by the HTTP/2 and websockets protocol handlers

Simpler IO code overall

Some successful attempts to use async IO vectoring:

- HTTP/2 non blocking input and frame parsing
- HTTP/2 frame and websockets message output
- HTTP/2 "sendfile" using mapped files

Example

HTTP/2 frame parser

https://github.com/apache/tomcat/blob/trunk/java/org/apache/coyote/http2/Http2AsyncParser.java

NIO 2 in your webapps

Hands on

Upgrade from HTTP/1.1 using Servlet 3.1 API

Extend InternalHttpUpgradeHandler rather than HttpUpgradeHandler

Container calls setSocketWrapper with the SocketWrapper

Read and write byte buffers using the SocketWrapper

```java
public class ExampleUpgradeHandler implements InternalHttpUpgradeHandler {
    private WebConnection connection;
    private SocketWrapperBase<?> socketWrapper;
    @Override
    public void setSocketWrapper(SocketWrapperBase<?> socketWrapper) {
        this.socketWrapper = socketWrapper;
    }
    @Override
    public void setSslSupport(SSLSupport sslSupport) {
        // TLS information available here
    }
    @Override
    public void init(WebConnection connection) {
        this.connection = connection;
        socketWrapper.write(BlockingMode.BLOCK, socketWrapper.getWriteTimeout(),
                TimeUnit.MILLISECONDS, null, SocketWrapperBase.COMPLETE_WRITE, null,
                ByteBuffer.wrap("hello".getBytes(Charset.defaultCharset())));
    }
    @Override
    public SocketState upgradeDispatch(SocketEvent status) {
        // May not be useful for NIO 2, but needed for APR and NIO
        return SocketState.OPEN;
    }
    @Override
    public void pause() {
        // NO-OP
    }
    @Override
    public void destroy() {
        try { connection.close(); } catch (Exception e) {}
    }
}
```

Async vectored IO

For IO intensive tasks

Excellent for any fixed length binary protocol

Innovative integrations with low level protocols

Allows replicating sendfile like performance and efficiency

Unforeseen consequences

Bypasses Servlet and container buffering

The caller now owns all buffers

No container buffering of any kind

Direct buffers use is possible and often beneficial

Only for protocol upgrades

Example

Websockets message output

https://github.com/apache/tomcat/blob/trunk/java/org/apache/tomcat/websocket/server/WsRemoteEndpointImplServer.java

Conclusion

A worthwhile addition

New async IO capabilities in Tomcat

Good performance

Future high level Reactive APIs might be easier to implement