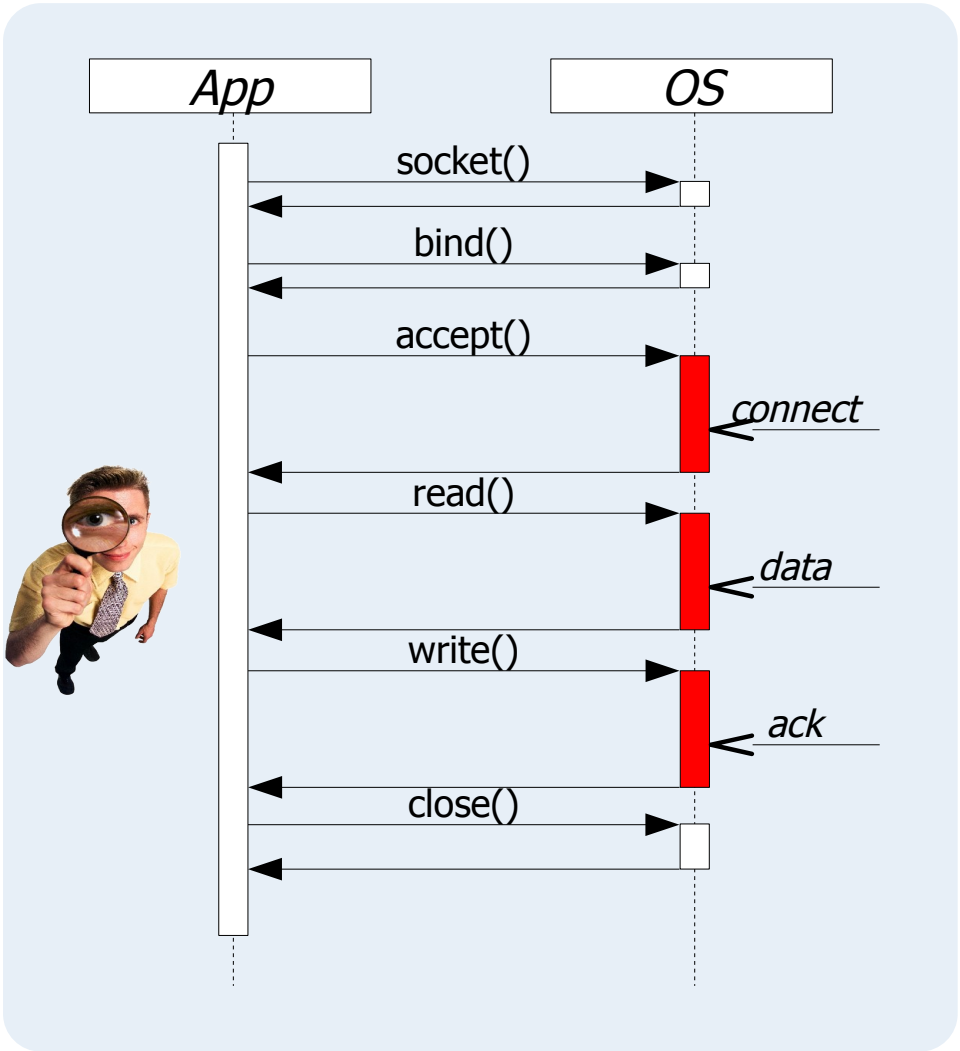
Summary

- Memory:
  - No data copying by pointing into the same buffers
  - Reuse and sharing reduces allocation
- Event-driven programs:
  - A single shallow stack
  - Minimal context switching
  - Explicit scheduling and queuing (can be purged)

Abstract Execution

```
socket()
bind()

        wait for accept...

        event!
        connect socket
alloc buffer
        wait for read....

        event!
        copy data to buffer
flip
        wait for write ...
        event!
        copy data from buffer
buffer empty
close
```

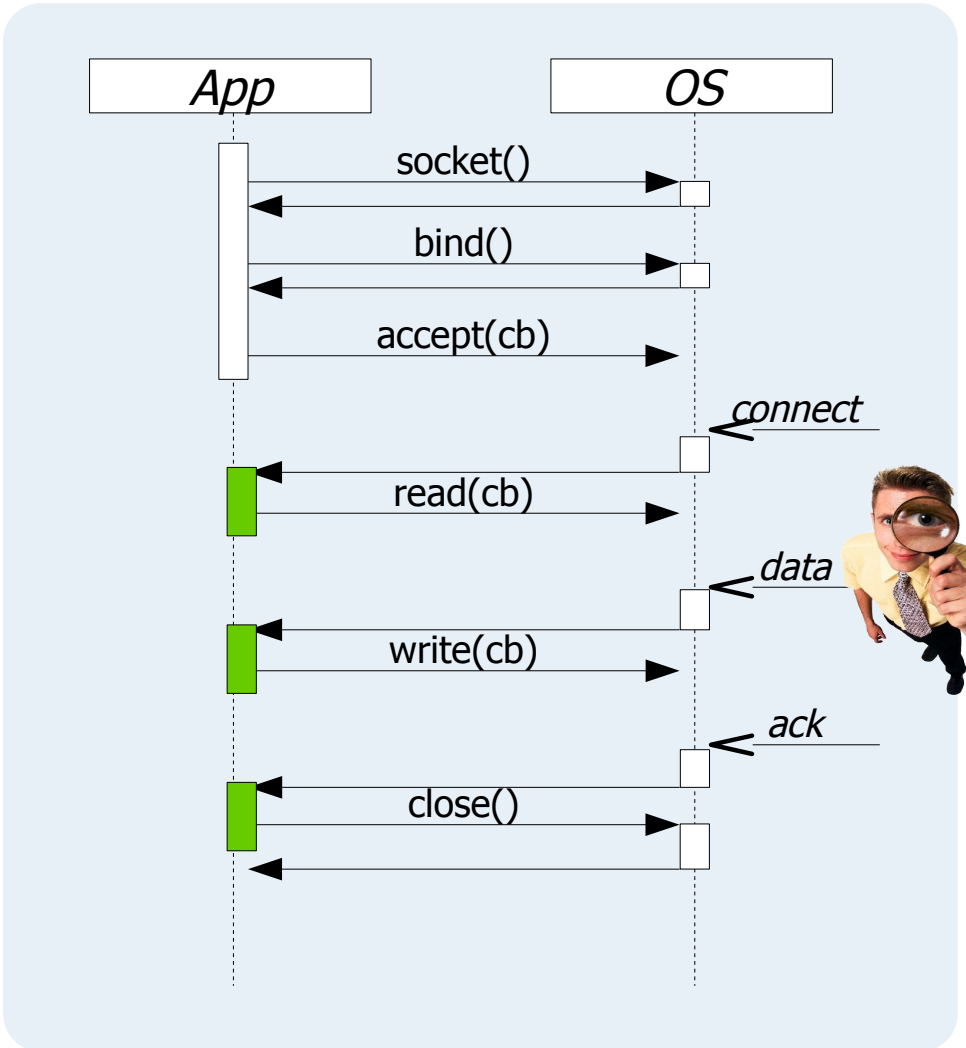
Threaded version

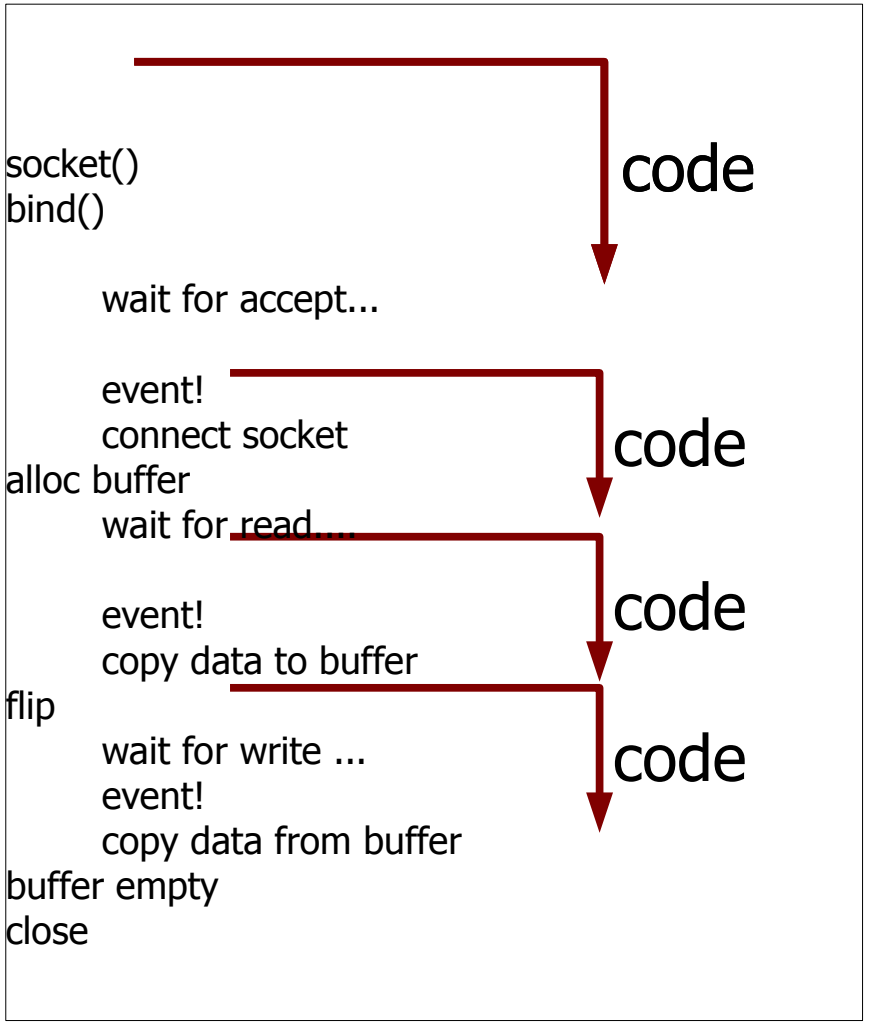



```
socket()
bind()

        wait for accept...

        event!
        connect socket
alloc buffer
        wait for read....

        event!
        copy data to buffer
flip
        wait for write ...
        event!
        copy data from buffer
buffer empty
close
```

code

Event-driven version





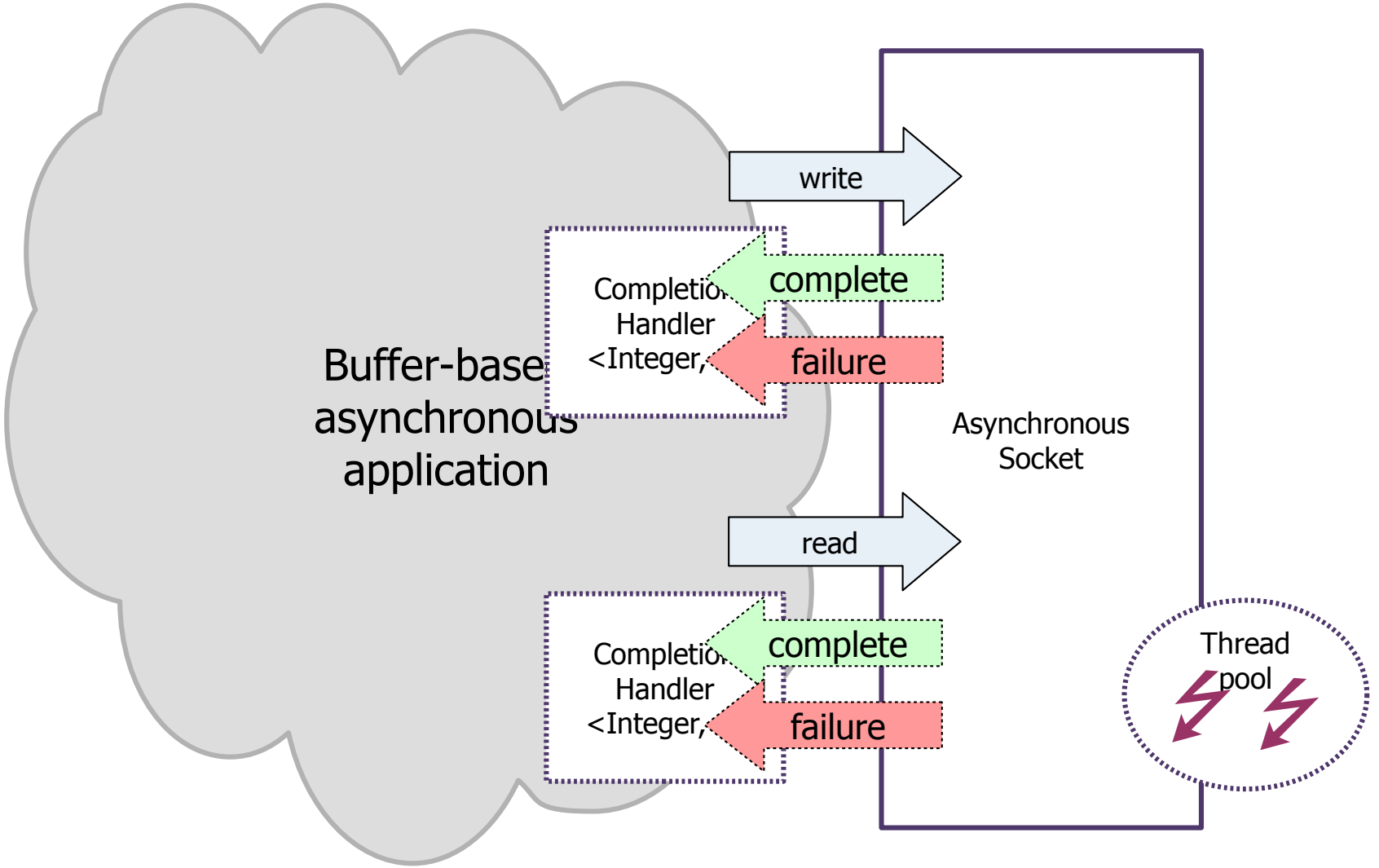
Inversion of Control (IoC)

- With threads:
  - The program controls flow
  - Calls into the framework for specific tasks
- With events:
  - The framework controls flow
  - Calls back the program for specific tasks

Case study

- Improve the chat server with:
  - Work with lines, not buffers
  - Validate login and password
- Assess impact on:
  - Ease of use
  - Reuse and composition (incl. w/ threaded code)

Buffer-based application



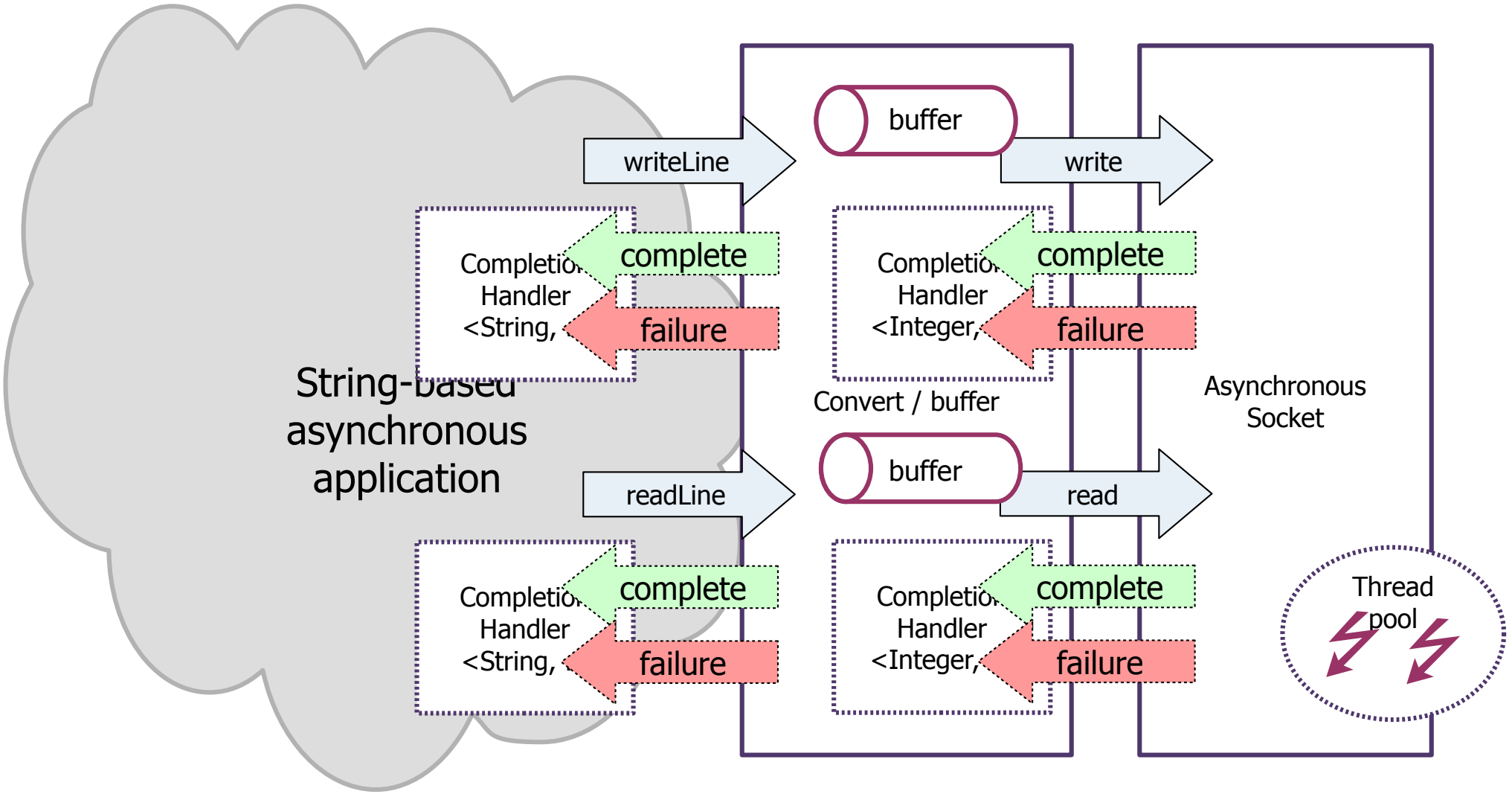
Asynchronous line buffer

```
public class AsynchronousLineBuffer {

    private AsynchronousSocketChannel sock;

    private CompletionHandler<String, Object> rHandler;
    private Object rValue;

    public <A> void readLine(final A value, CompletionHandler<String, A> handler) {
        ...
    }

    private CompletionHandler<Void, Object> wHandler;
    private Object wValue;

    public <A> void writeLine(String line, final A value,
                              CompletionHandler<Void, A> handler) {
        ...
    }
}
```

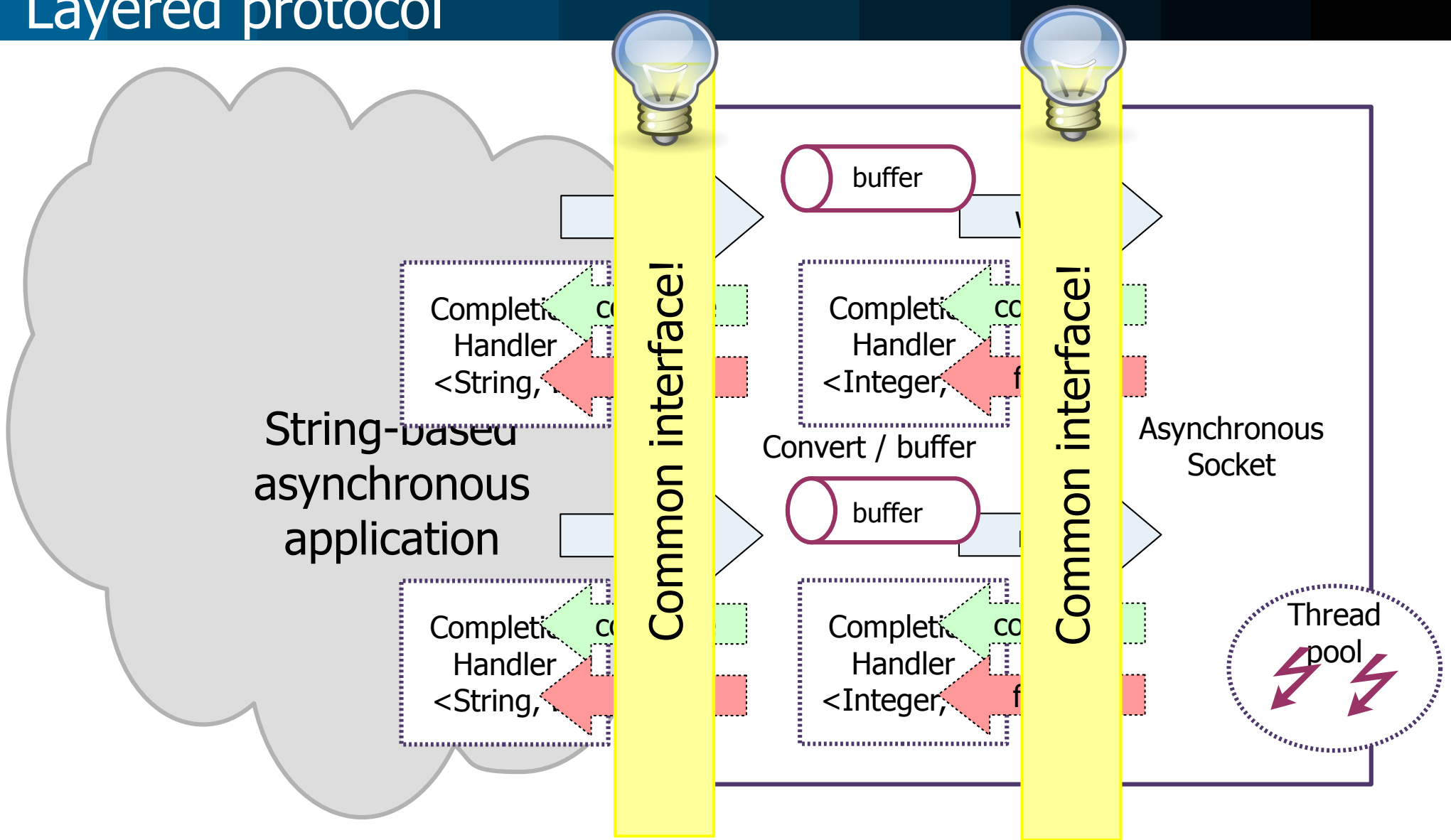
String-buffer layer

- On reading, gathers data and de-serializes
- On writing, serializes and flushes data
- Can be generalized to any object, by changing the serialization code (e.g., protobuf)

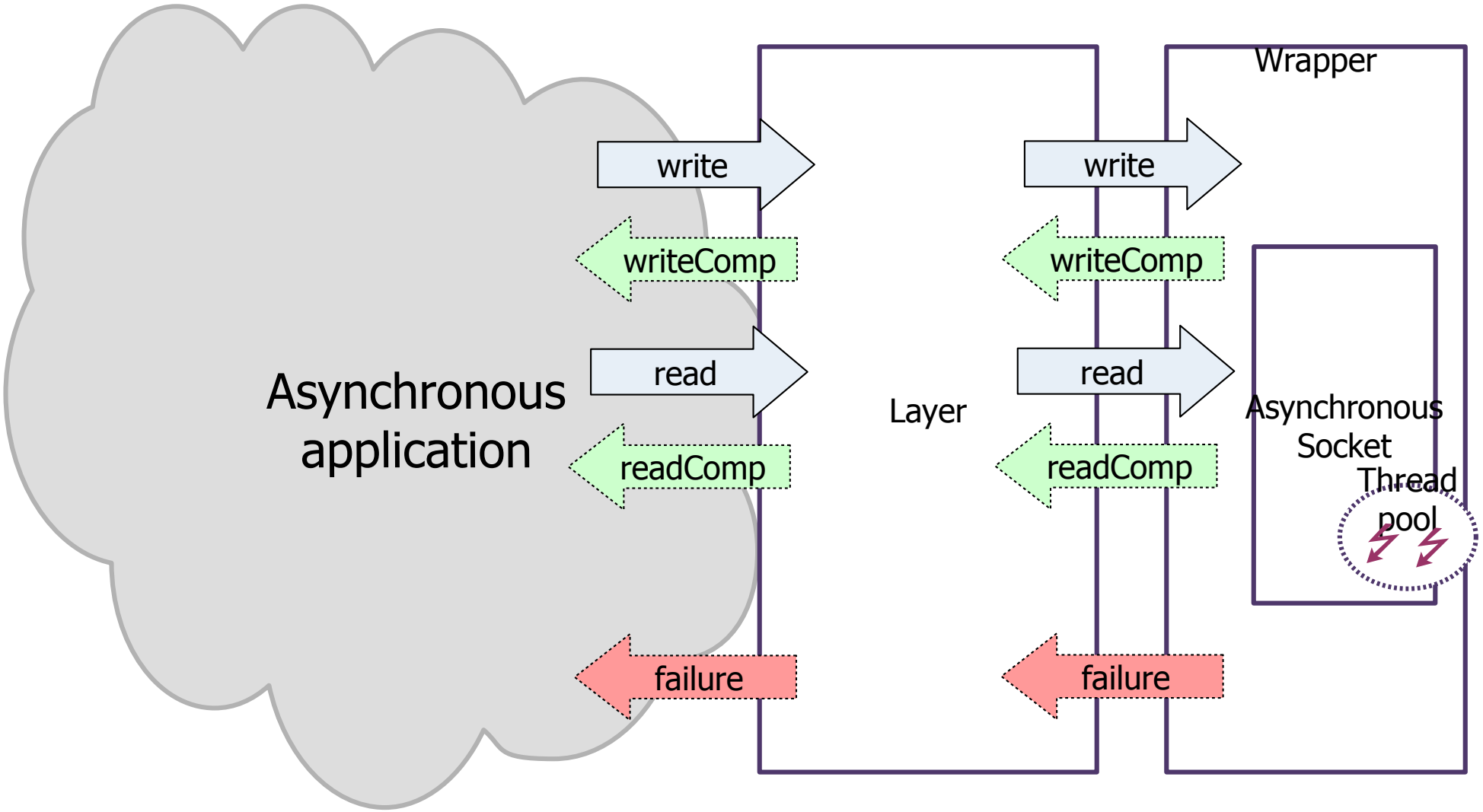
String-based application



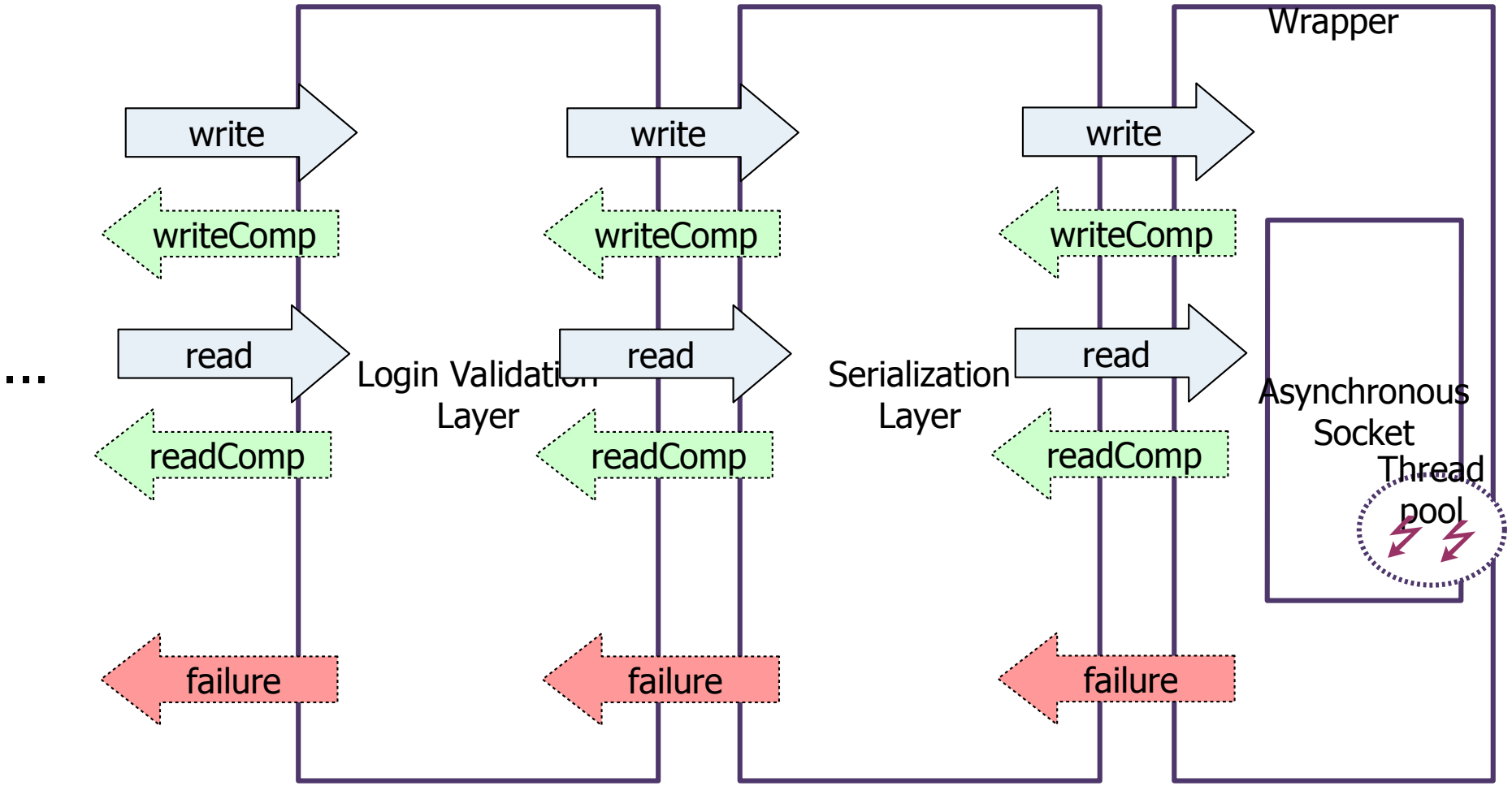
Layered protocol



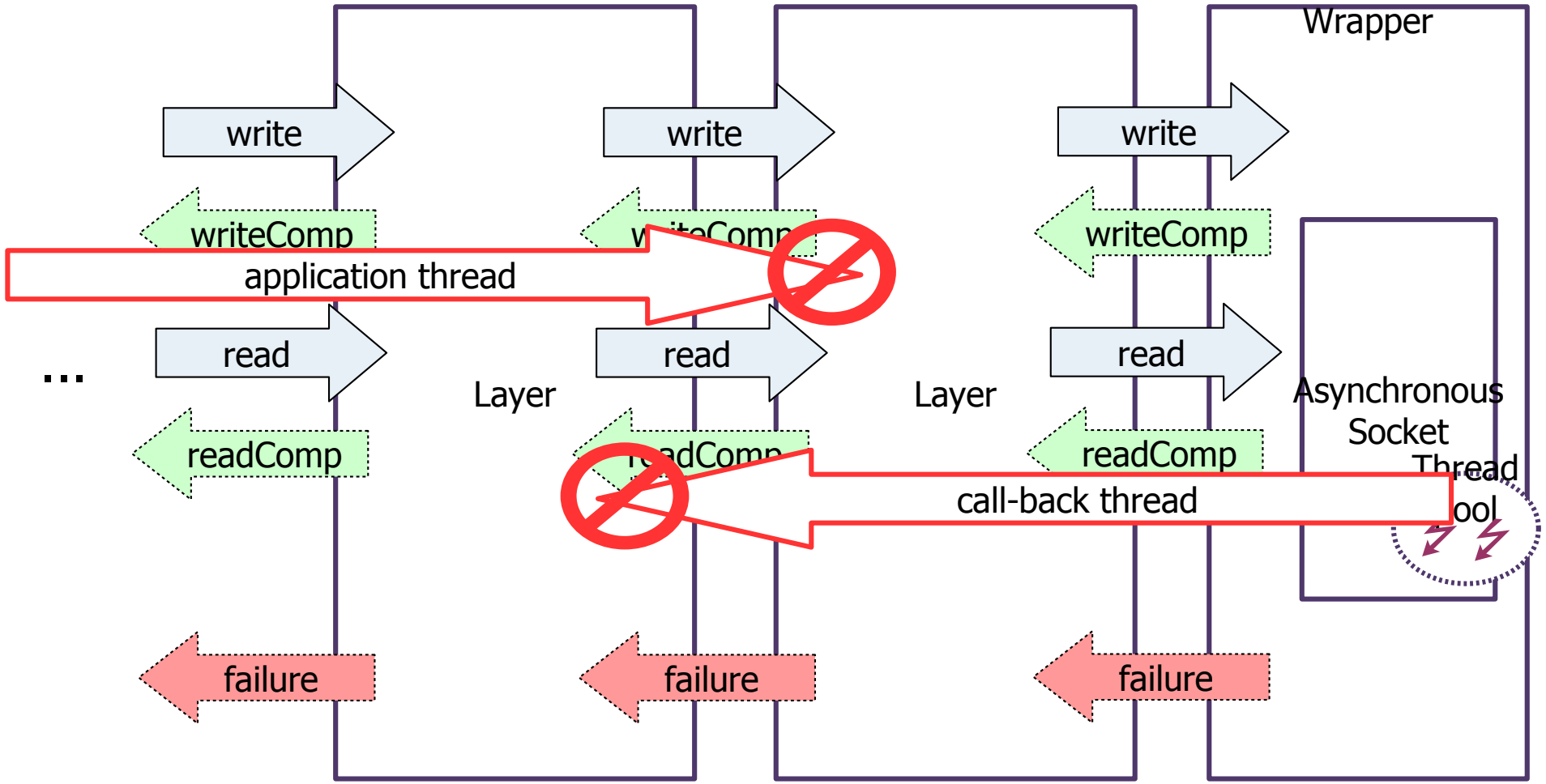
Layered protocol



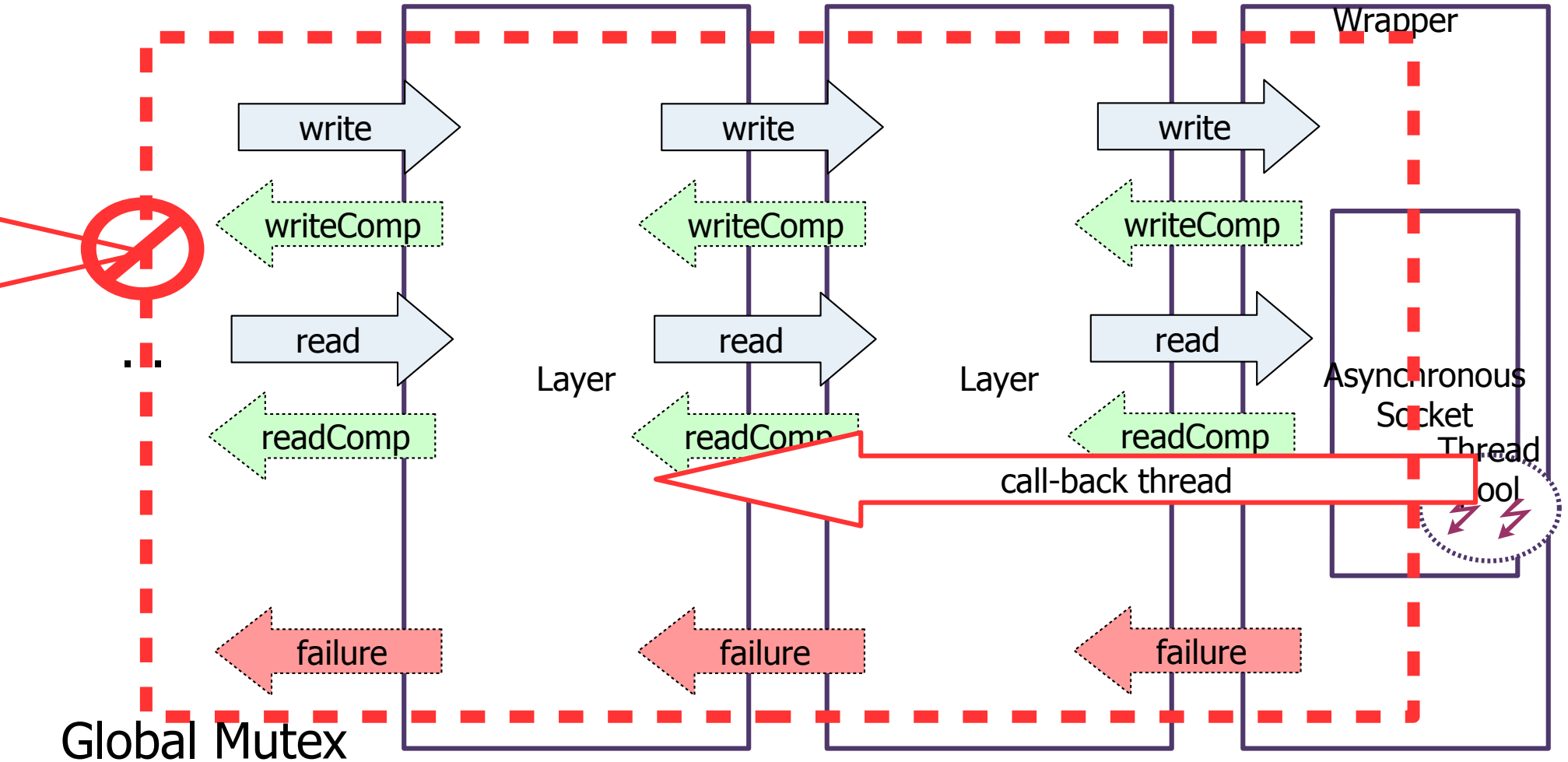
Layered protocol



Potential deadlock



Potential deadlock



Layered protocols

- Emphasis on composing event-driven code
- Pipeline that processes a data flow
  - Layers with a common interface
- Threading:
  - Lock by pipeline
  - Don't call back into the application
- Example:

Asynchronous line buffer

```
public class AsynchronousLineBuffer {
    private AsynchronousSocketChannel sock;

    private CompletionHandler<String, Object> rHandler;
    private Object rValue;

    public <A> void readLine(final A value, CompletionHandler<String, A> handler) {

        public void complete(...) {
            if (rHandler != null) rHandler.complete(..., rValue);
        }
    }
    private CompletionHandler<String, Object> wHandler;
    private Object wValue;
    public <A> void writeLine(String line, CompletionHandler<Void, A> handler) {
        ...
    }
}
```

Repeated code!

Monadic asynchronous

- Encapsulate call-back in a standard reusable class: `CompletableFuture`
  - Created by the callee
  - Can be returned to the caller
  - Allows cancellation and multiple call-backs
  - Allows synchronous waiting (future)
- How to use:
  - Non-blocking method returns some Value
  - Blocking method returns some CompletableFuture<Value>

Monadic asynchronous

- Provide composition of call-back instances
  - Chain non-blocking code: `thenApply()`
  - Chain blocking code: `thenCompose()`
- Long lived blocking code:
  - Use Async version of methods for background thread

Translation to CompletableFuture

```
try {
      C c = codeBefore(...);

      R r = operation(...);

      codeAfter(c, r);

} catch(Exception e) {
      handleException(e);
}
```

```
C c = codeBefore(...);

asyncOperation(...)

      .thenAccept( (r) → codeAfter(c, r) )


      .exceptionally( (e) → handleException(e) )
```

Cheat Sheet

- Obtaining a future from scratch:

	Input			Output		
Operator	*none*	*now*	*code*	*none*	*value*	*exception*
`new`	x				x	x
`completedFuture`		x			x	
`failedFuture`		x				x
`runAsync`			x	x		x
`supplyAsync`			x		x	x

Cheat Sheet

- Composition with non-blocking code:

	Input			Output		
Operator	*none*	*value*	*exception*	*none*	*same*	*new*
`thenRun`	x			x		
`thenAccept`		x		x		
`thenApply`		x				x
`exceptionally`			x			x
`handle`		x	x			x
`whenComplete`		x	x		x	

(*Async variants run handler in background thread)

Cheat Sheet

- Composition with blocking code:

	Parallel composition			Input		Output	
Operator	*no*	*both*	*either*	*none*	*value*	*none*	*value*
`thenCompose`	x				x		x
`thenCombine`		x			x		x
`runAfterBoth`		x		x		x	
`runAfterEither`			x	x		x	
`applyToEither`			x		x		x
`allOf`		x		x		x	
`anyOf`			x	x			x

(*Async variants run handler in background thread)

Monadic line buffer

```
public class LineBuffer {

    private SocketChannel sock;

    public  String readLine() {

        ...
        sock.read(...);
        ...
            readLine();

        return line;
    }

    public void writeLine(String line) {
        ...
    }
}
```

Recursive and blocking

Monadic asynchronous line buffer

```
public class FutureLineBuffer {

    private FutureSocketChannel sock;

    public CompletableFuture<String> readLine() {


        ...
        return sock.read(...)
            .thenCompose( (r) → { ...; readLine(); } )
        ...

        return CompletableFuture.completed(line);
    }

    public CompletableFuture<Void> writeLine(String line) {
        ...
    }
}
```