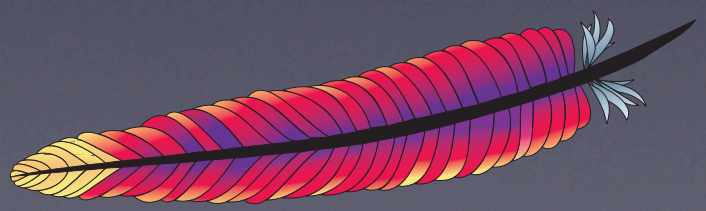
(if you can't read this, move closer!)

Apache Mina

The high-performance protocol construction toolkit.

Peter Royal <proyal@apache.org>

Originally presented at ApacheCon Europe 2007
in Amsterdam



Hi, I'm Peter

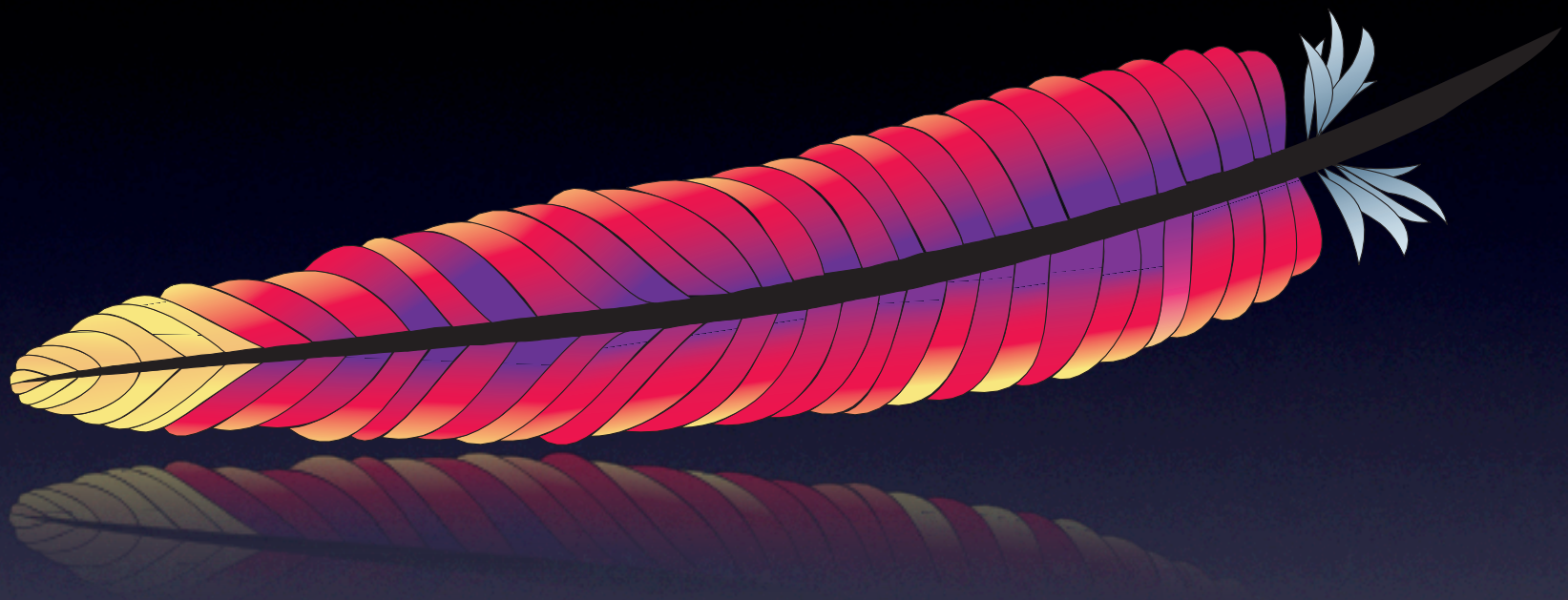
http://fotap.org/~osi

MINA hacker since
Fall 2005

San Francisco

Radar Networks

http://radarnetworks.com



Apache Member

http://apache.org

What is MINA?

MINA?

Multipurpose Infrastructure Networked Applications

http://mina.apache.org

Built on Java NIO

Non-Blocking

Asynchronous

Event-Driven

Multiple Transports

(framework is really agnostic)

TCP

UDP

(being re-written for 2.0)

In-VM

(great for testing)

RS-232

(under development)

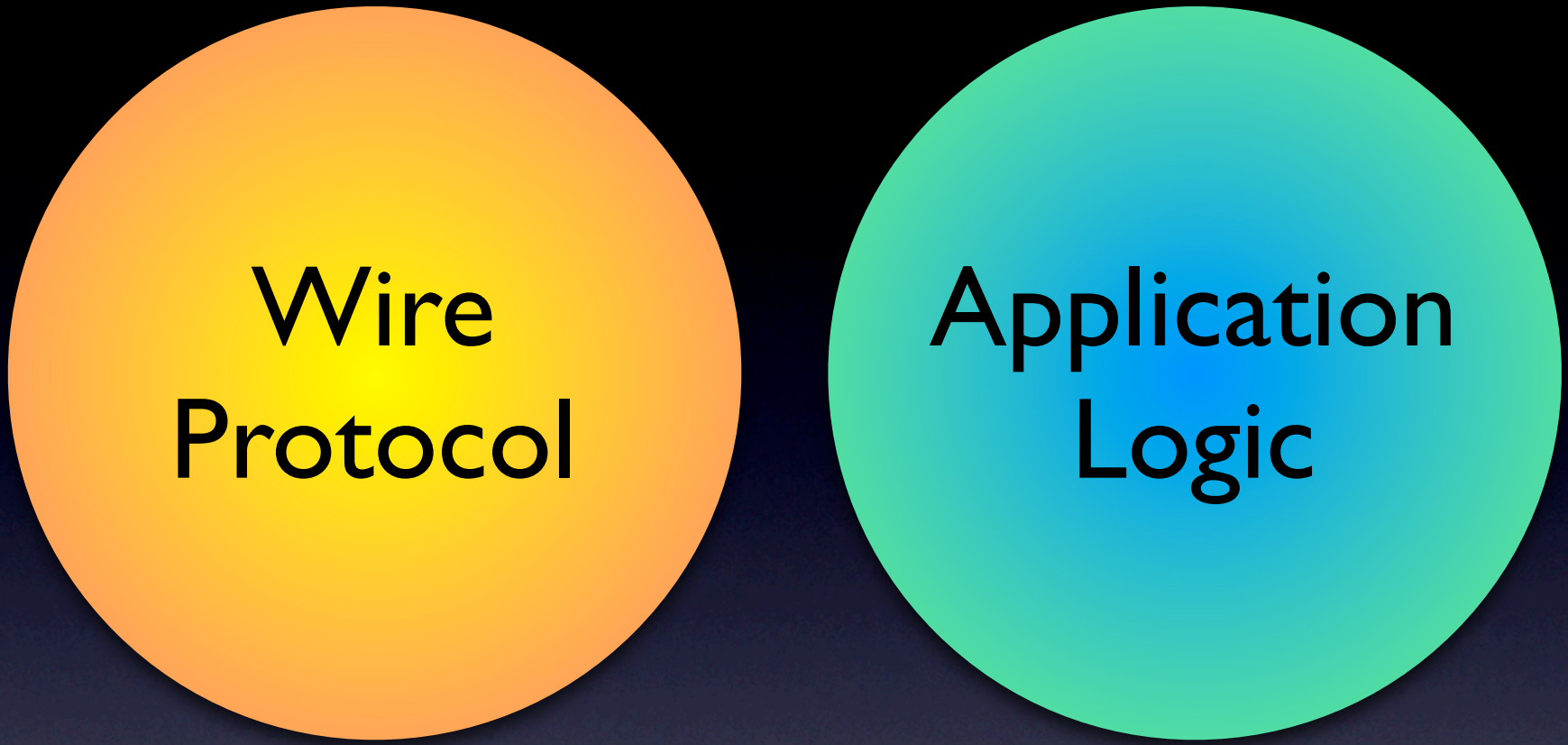
Smartly Designed

Follows Inversion of Control Pattern

(plays nicely with PicoContainer, Spring, etc)

Separation of Concerns

Rather than this...



Concerns are Separated

Stable and Production-Ready

- v1.0 released Fall 2006
- v1.1 released April 2007
  - Same API as v1.0 but uses Java 5 Concurrency primitives
- v2.0 this year
  - API simplification based on lessons learned

Many Users

Apache Directory

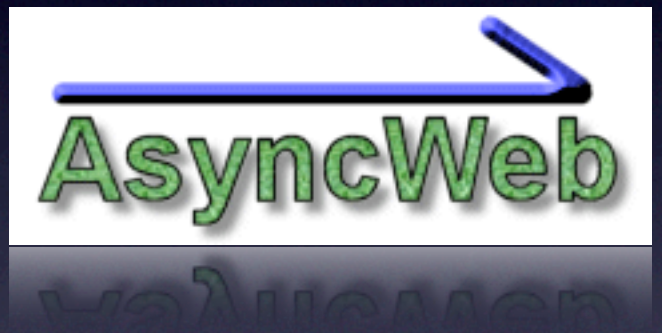
http://directory.apache.org

LDAPv3, NTP, DNS, DHCP and Kerberos

AsyncWeb

http://asyncweb.safehaus.org
(joining MINA @ Apache very soon though!)

HTTP/HTTPS

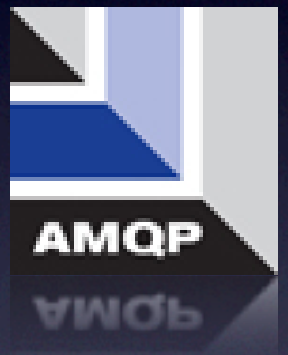


Apache (incubating) Qpid

http://cwiki.apache.org/qpid/

Advanced Messaging Queuing Protocol (AMQP)

(from Wall Street!)



QuickFIX/J

http://www.quickfixj.org/

Financial Information eXchange
(FIX)

Openfire

http://www.jivesoftware.com/products/openfire/

XMPP

red5

http://www.osflash.org/red5

RTMP
(talk to Flash player)

...and more!

(maybe you, next time!)

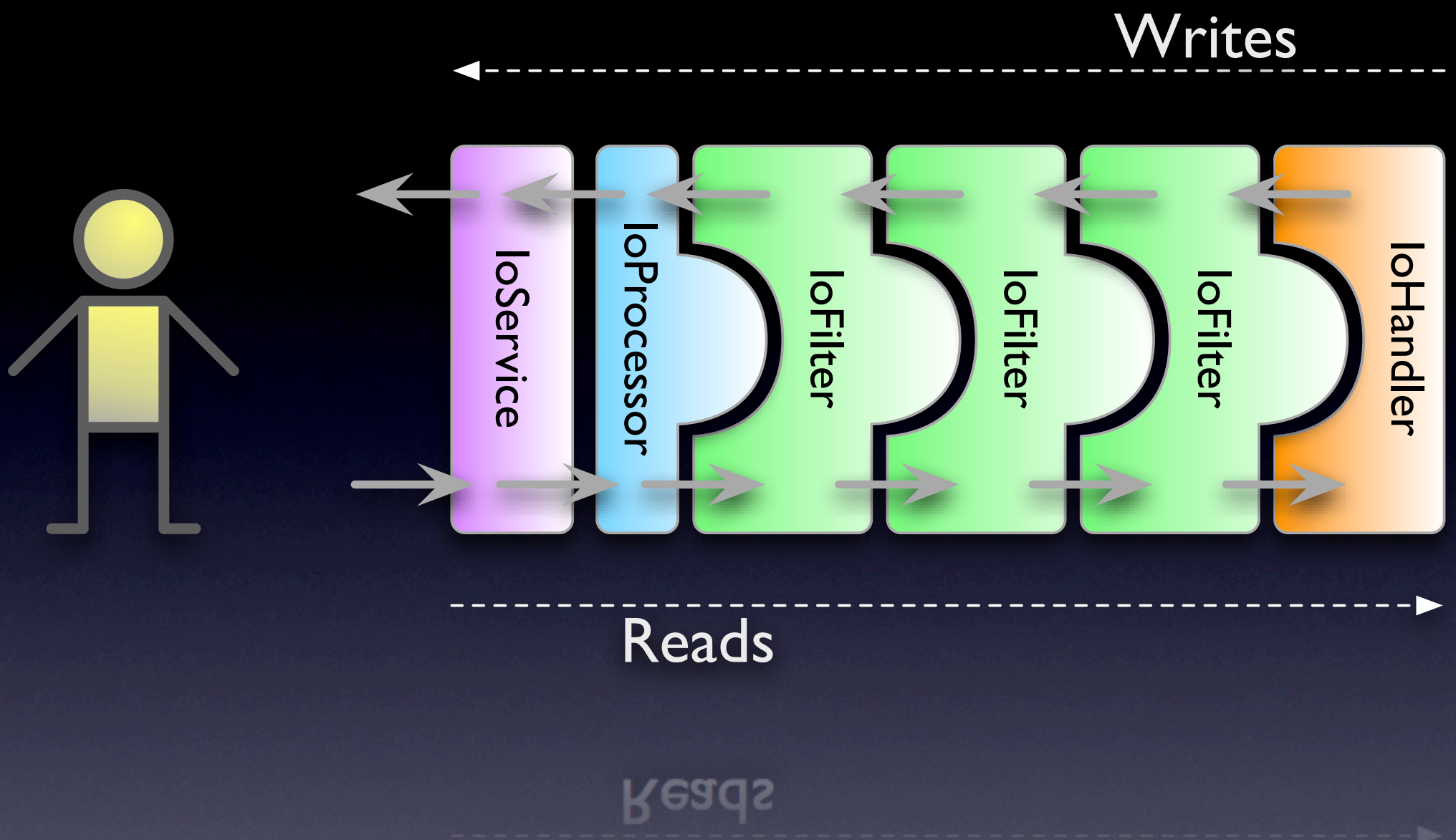
Key Concepts

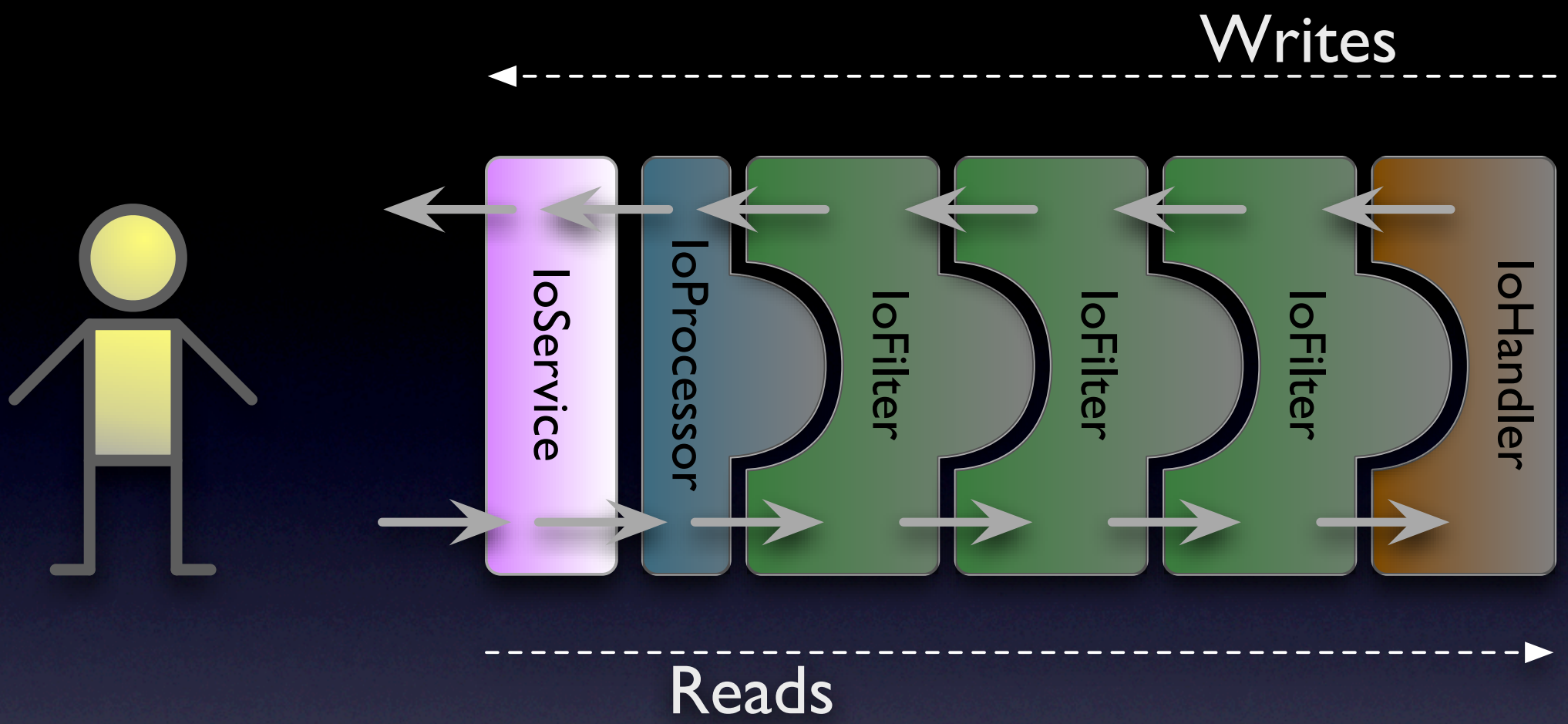
ByteBuffer

- Core NIO construct
- MINA version that wraps and provides **additional convenience methods**
  - auto-expanding, string encoding
- MINA gives control...
  - allocate from the **Heap** or **Stack**
  - optional **Pooling**
    - (in v2, will be non-pooled and heap-only, as it provides the best performance)

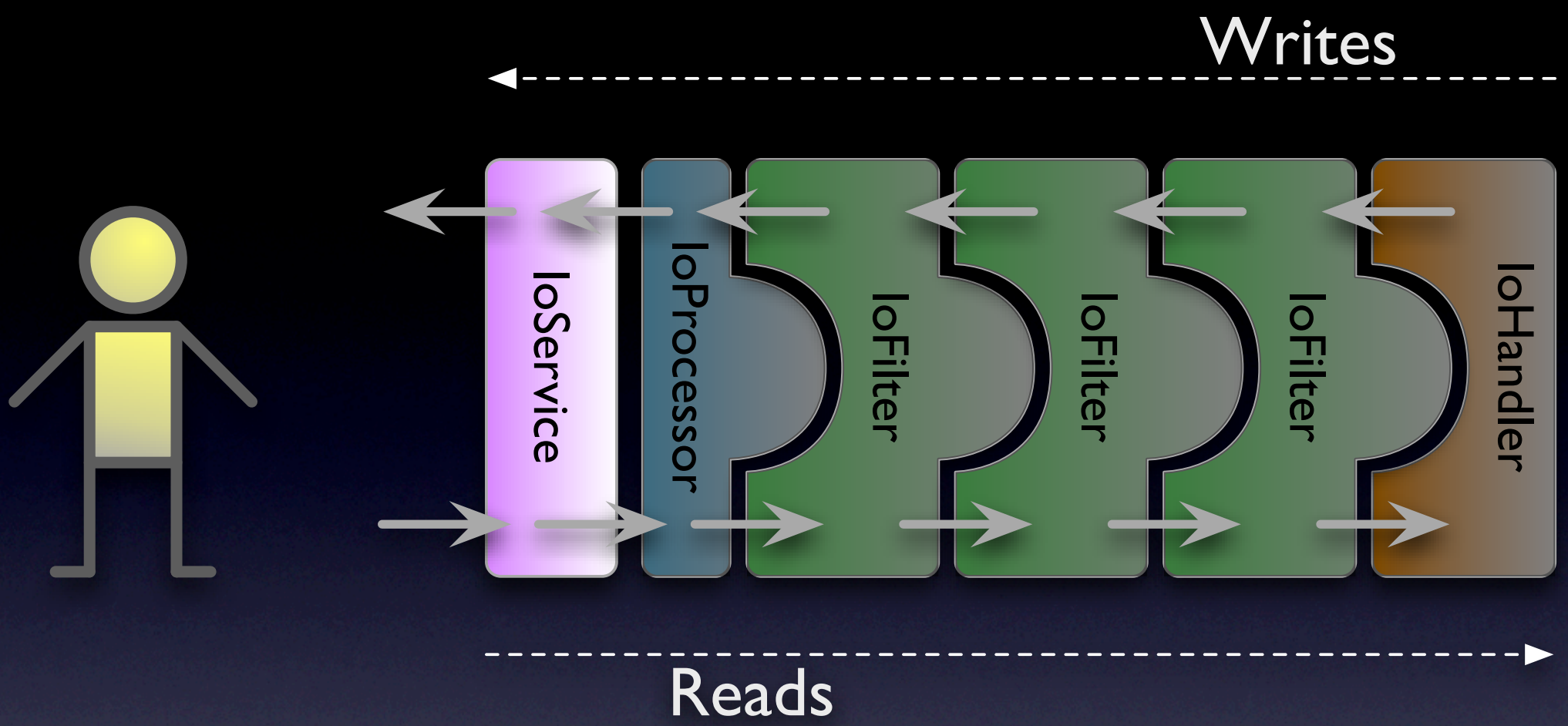
Future

- Represents a function call that completes asynchronously
- Provides blocking functions to retrieve the result
- MINA allows callbacks to be invoked upon completion, so invoking thread can “fire and forget”
  - (unlike the Java 5 Future)


Writes
IoService
IoProcessor
IoFilter
IoFilter
IoFilter
IoHandler
Reads



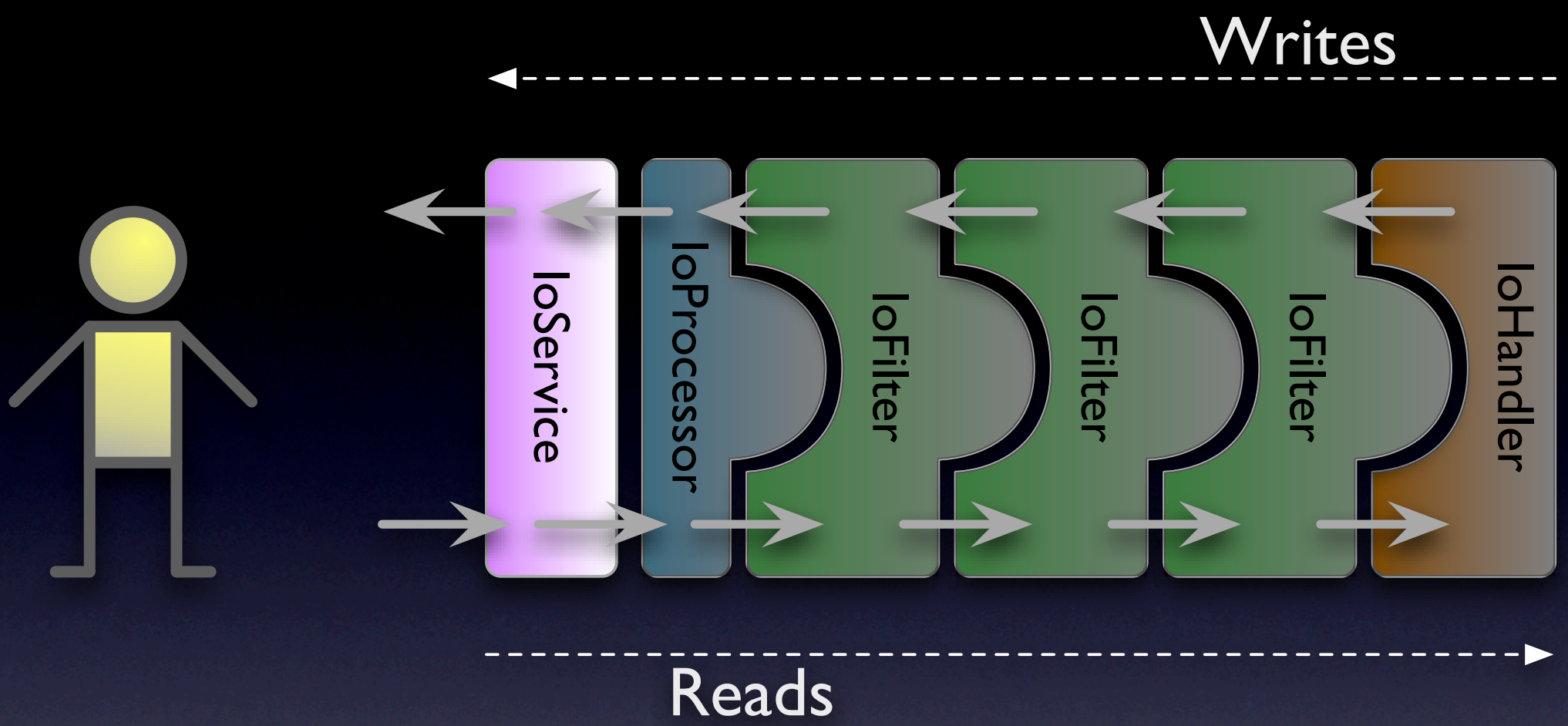
Two Versions



IoAcceptor

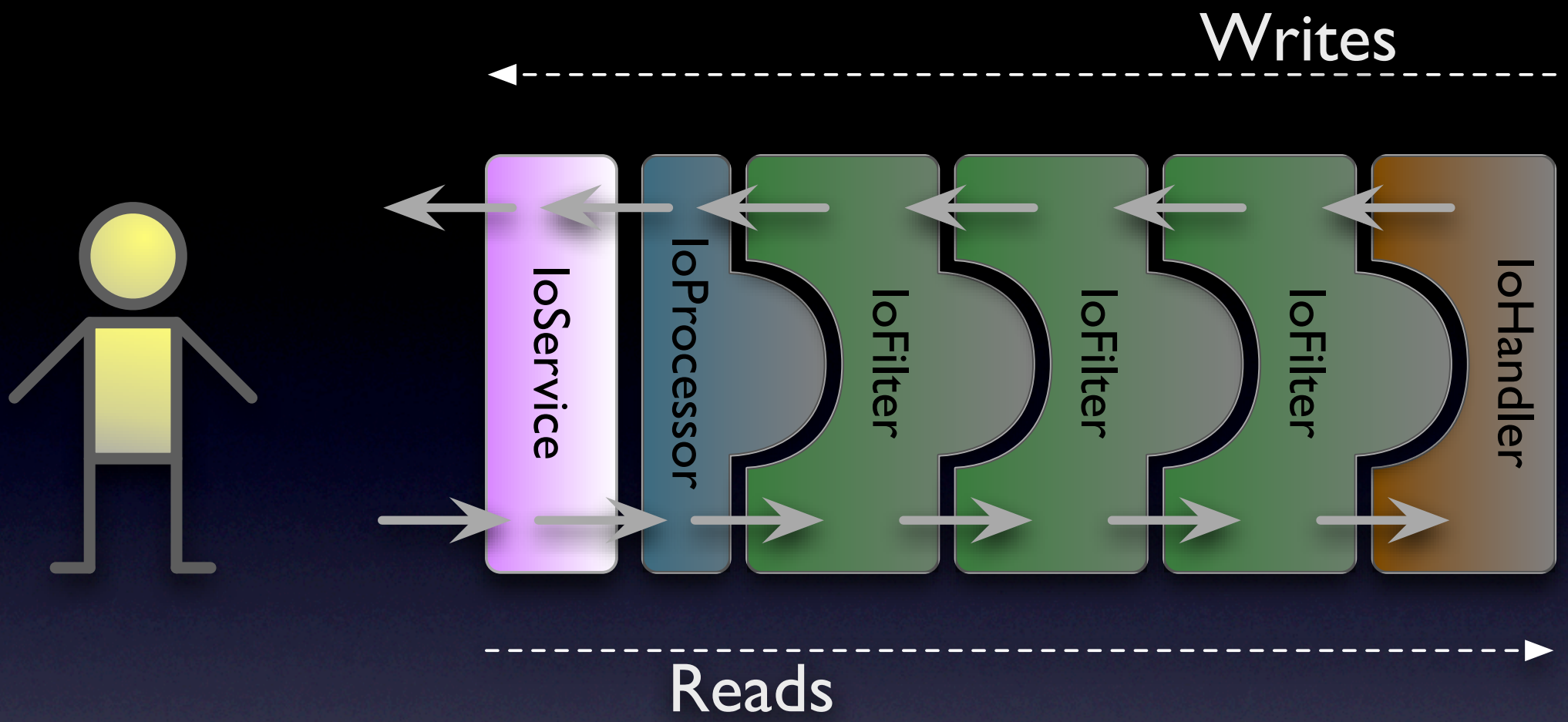
act as Server

single thread for new connections

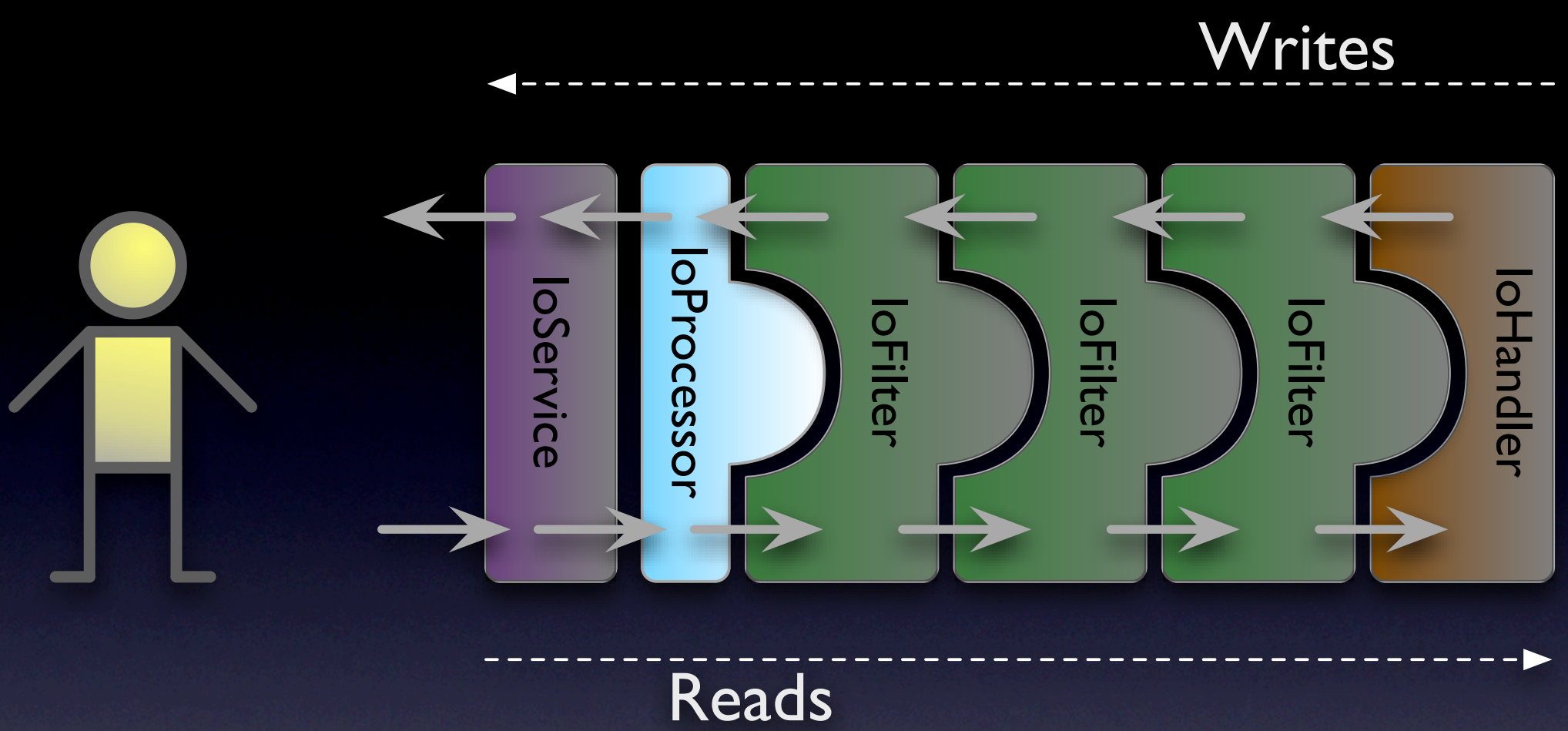


IoConnector

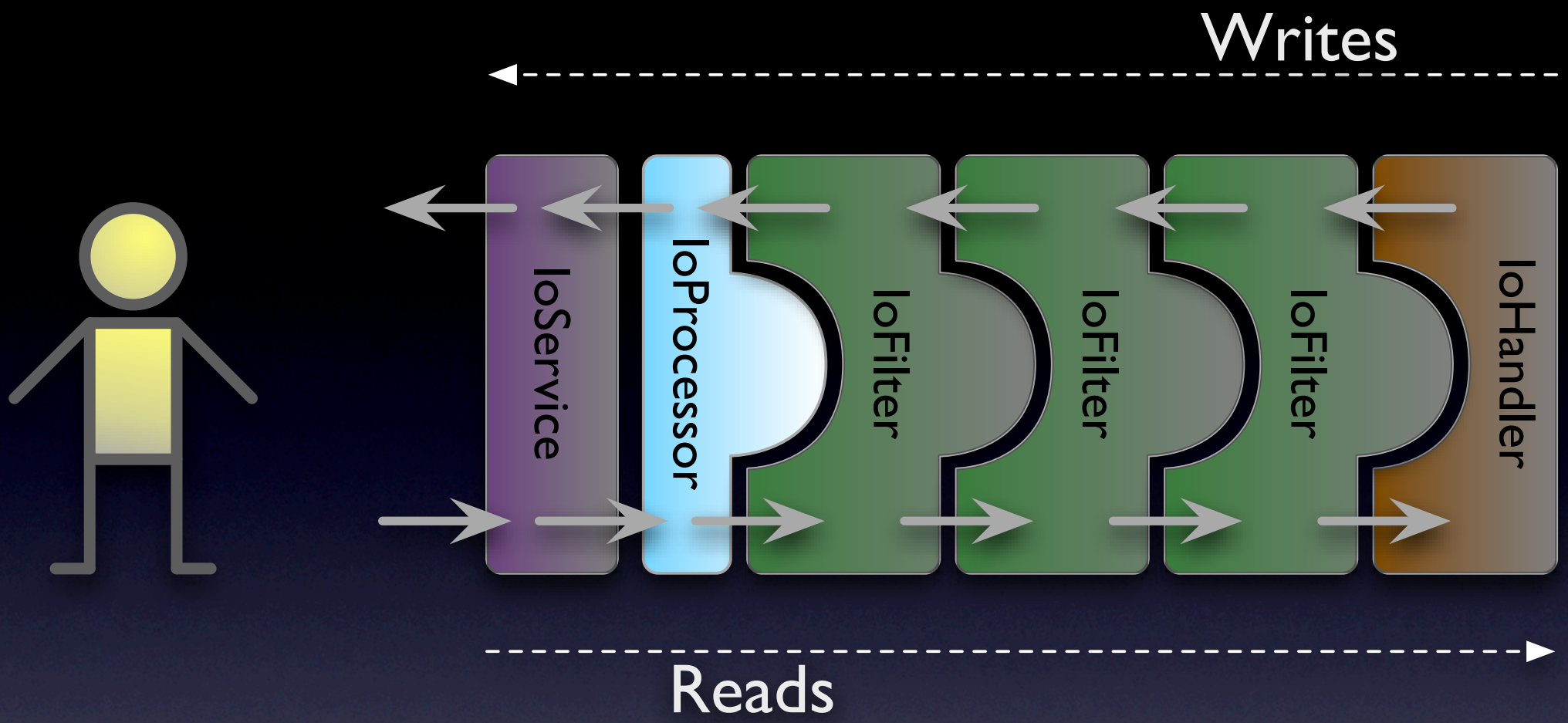
act as Client



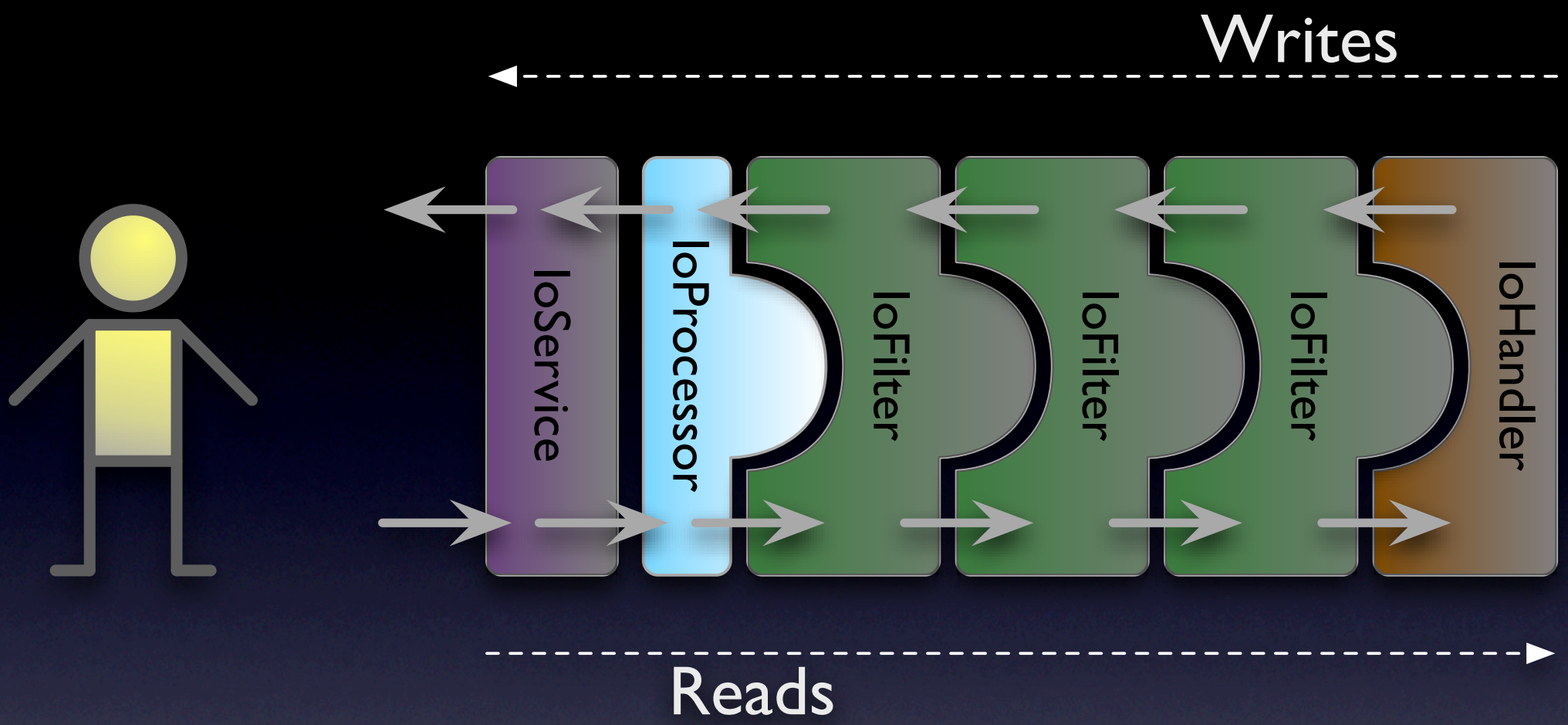
Connection instance is an IoSession



Handles reads and writes

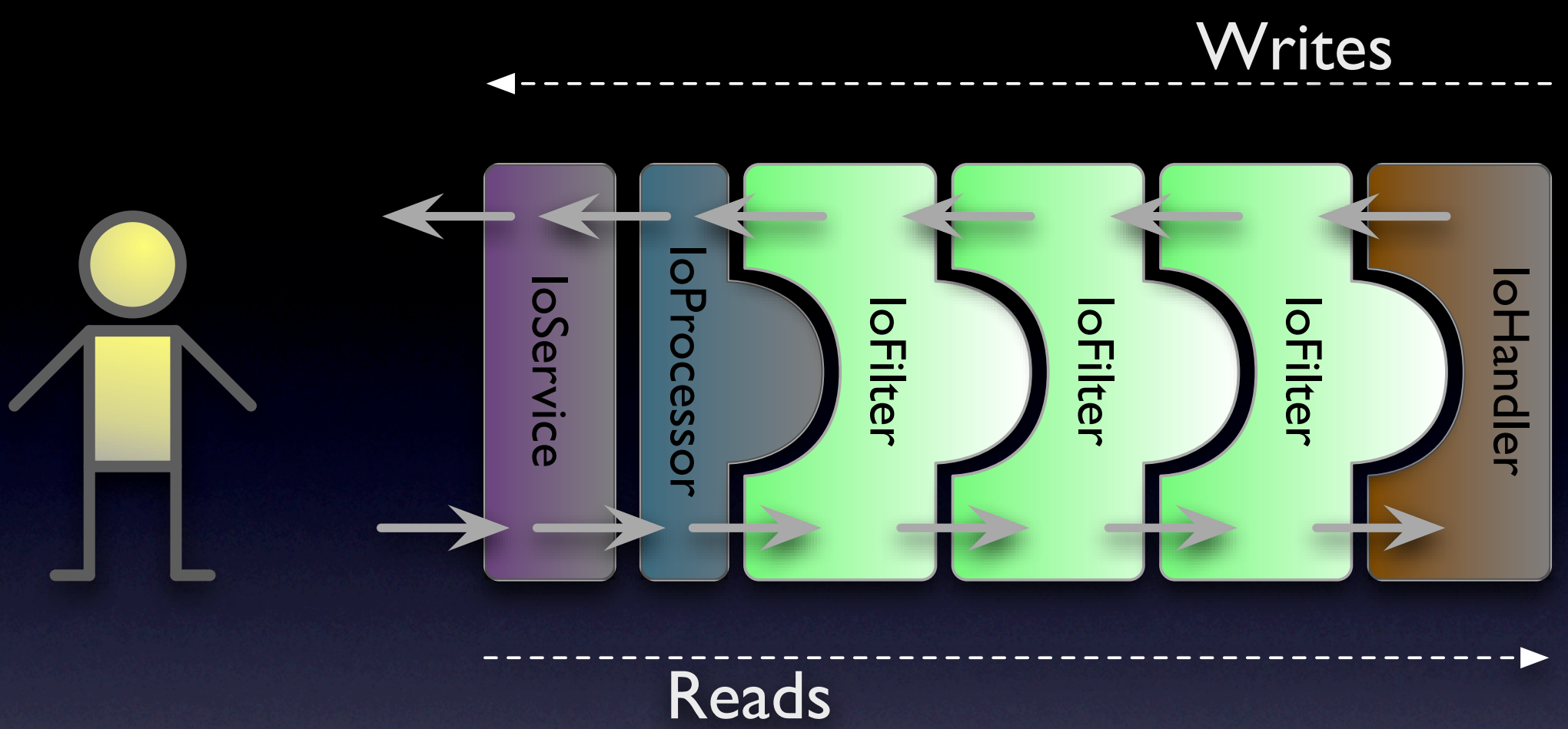


Instance count scales with CPU/Load

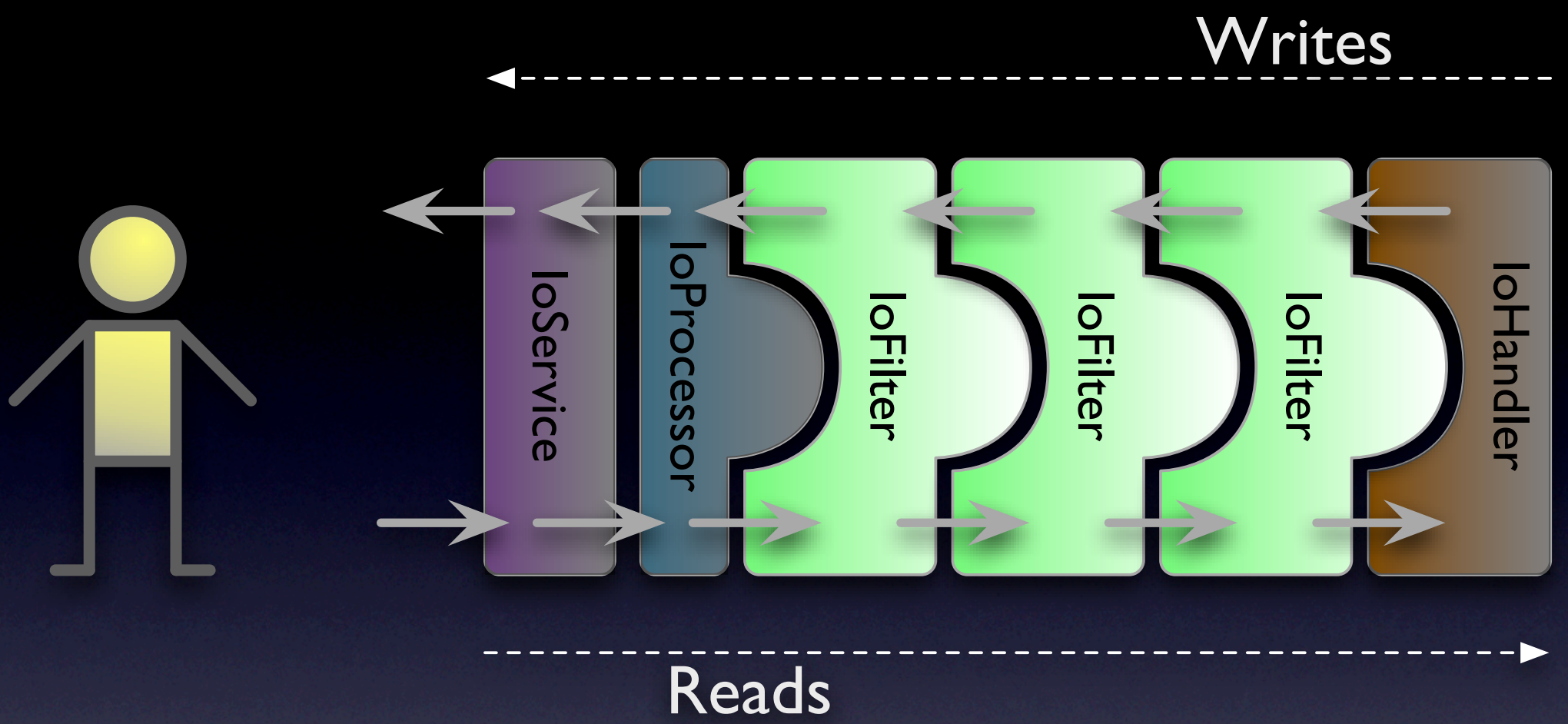


Session fixed to an Instance

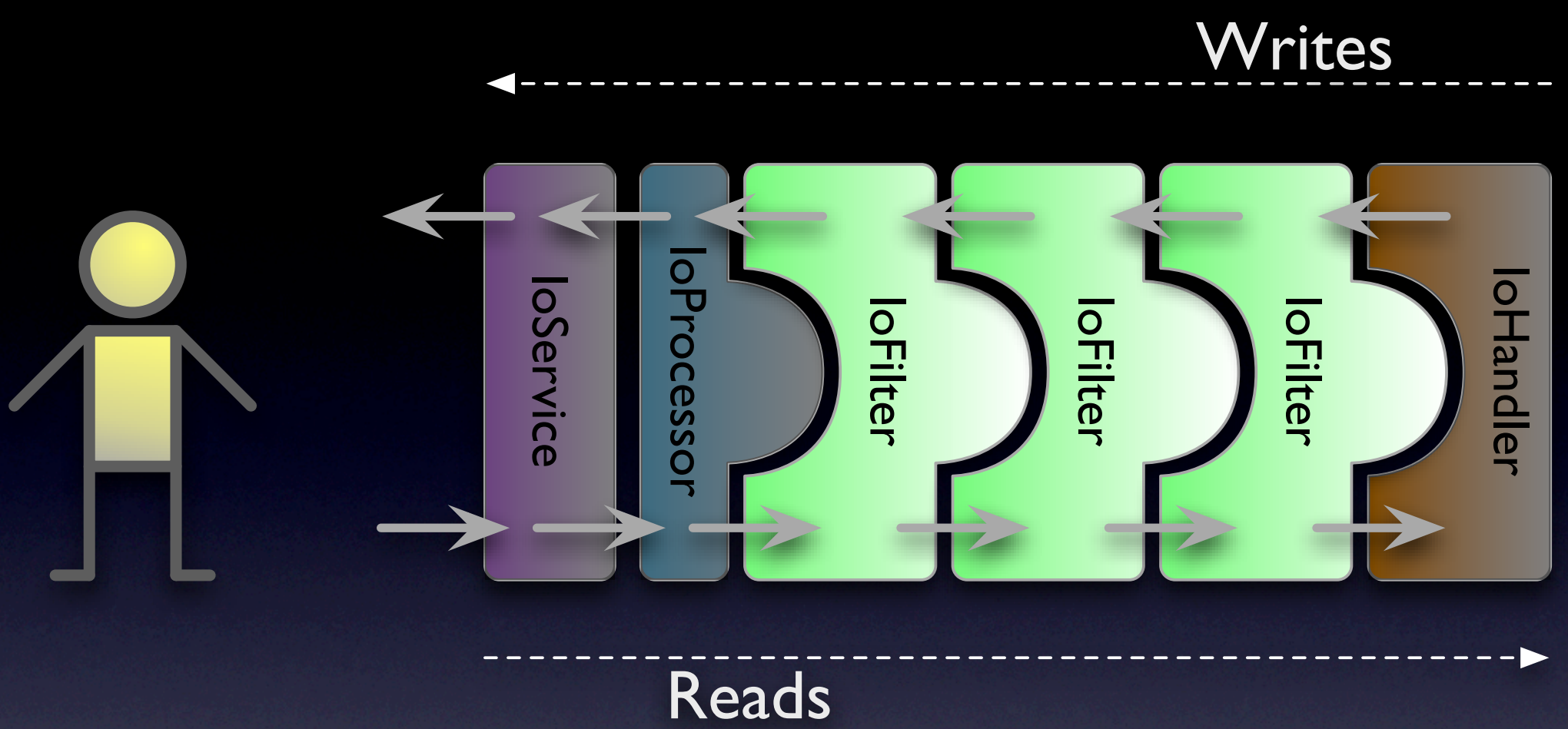
(under review for v2)



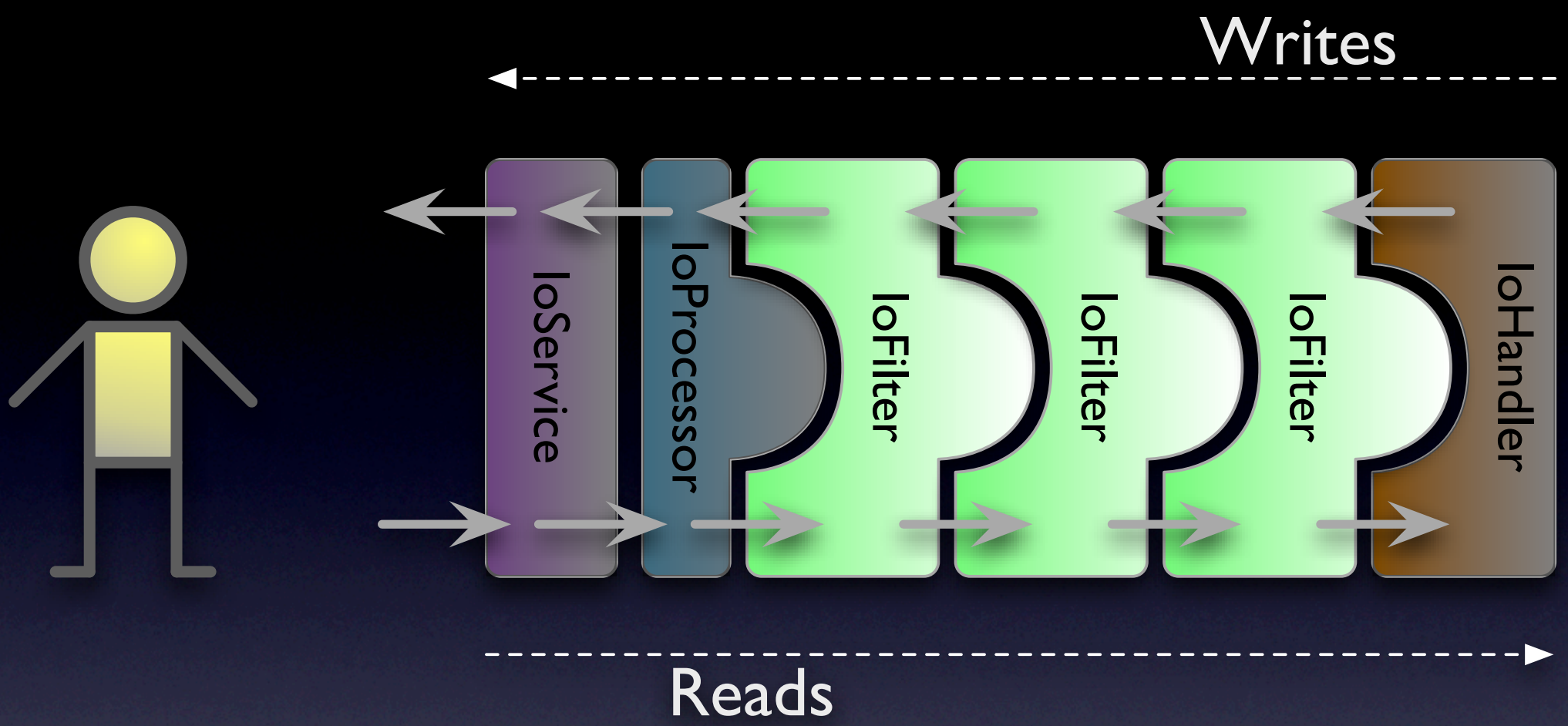
Chain of IoFilter's



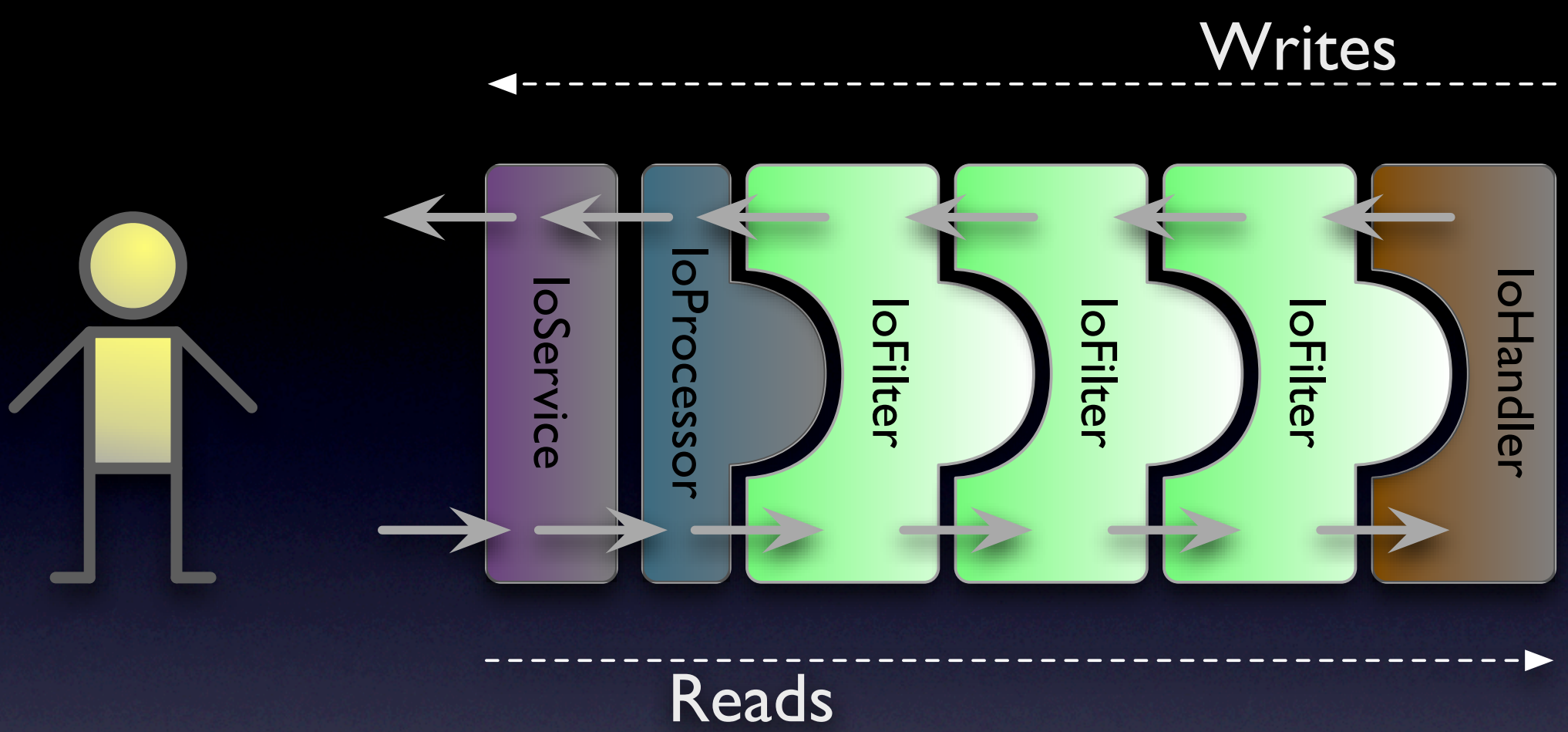
Per Connection


Writes
IoService
IoProcessor
IoFilter
IoFilter
IoFilter
IoHandler
Reads

Reusable

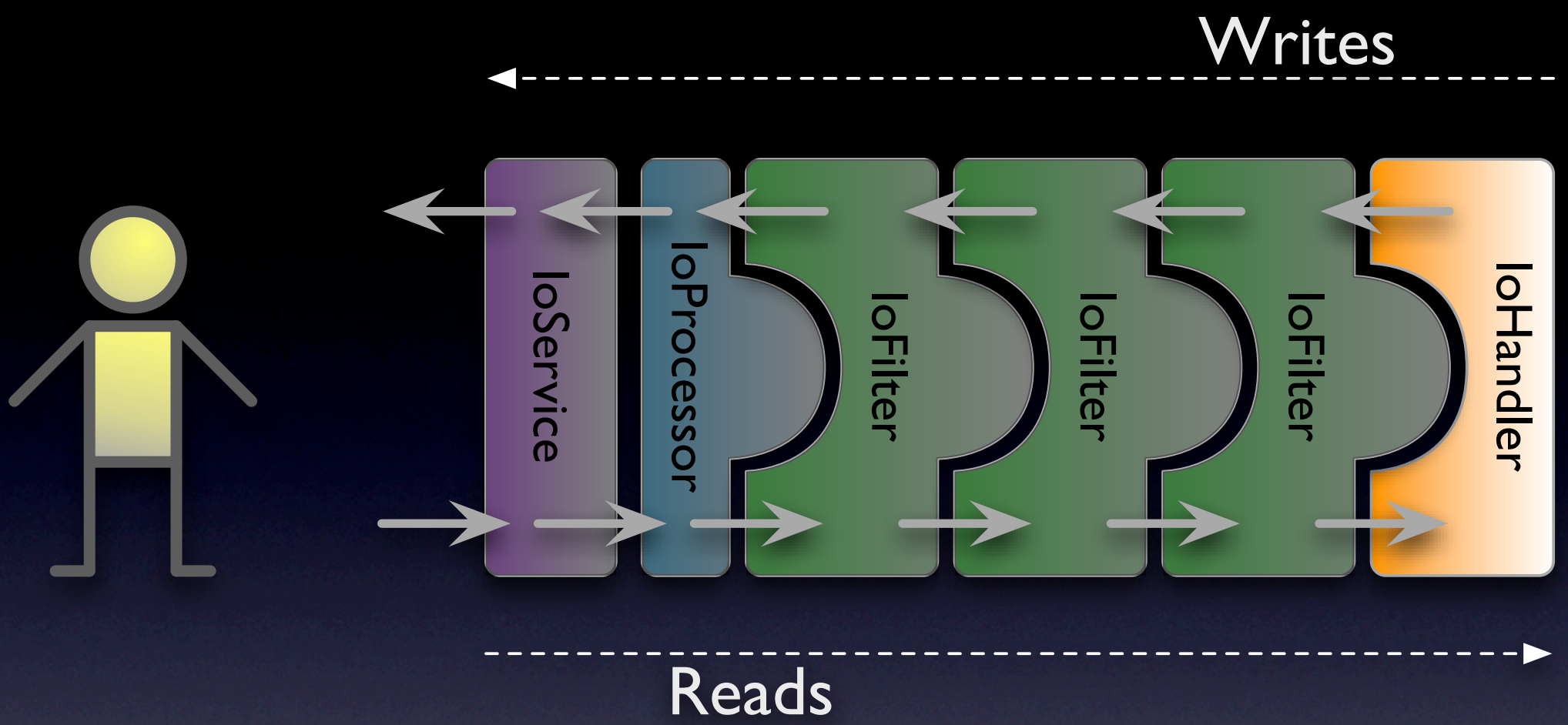


Hot Deployable



Filter all events

Read / Write / Idle / etc



Application Logic Lives Here

Large Library of IoFilter's

Protocol Conversion

- Framework to plug in your own codecs
- Existing codecs
  - Text-based
  - Java Serialization

Blacklist

Logging

(great for debugging!)

SSL / TLS

Compression

Read Throttling

Thread Models

(a necessary evil)

“single threaded”

One IoProcessor Thread

Scalability sucks

Add more IoProcessor Threads

(at least one per CPU core)

Lowest latency
Scales nicely

(connection latency to be addressed in v2)

“multi threaded”

use ExecutorFilter

IoProcessor threads only do reads & writes

(the intent)

Filters execute on a different thread.

(filter location is key!)

Work for a session is serialized

(queued per session)

Work is delegated to an Executor

(generally a java.util.concurrent.ThreadPoolExecutor)

Size thread pool to “active” session count

(too much in queue == OOM!
Use the Read Throttle filter)

“recommended application pattern”

- use **ExecutorFilter**
  - unless you need really low latency
- use **ProtocolCodecFilter**
  - convert the wire protocol into a Java representation
- put application logic into an **IoHandler**
- store state in the **IoSession**
- minimum of **Java 5**
  - java.util.concurrent rocks!

Demo Time!

Everybody loves a Haiku

```
public class HaikuValidator {
    private static final int[] SYLLABLE_COUNTS = { 5, 7, 5 };

    public void validate( Haiku haiku ) throws InvalidHaikuException {
        String[] phrases = haiku.getPhrases();

        for ( int i = 0; i < phrases.length; i++ ) {
            String phrase = phrases[i];
            int count = PhraseUtilities.countSyllablesInPhrase( phrase );

            if ( count != SYLLABLE_COUNTS[i] ) {
                throw new InvalidHaikuException( i + 1, phrase, count, SYLLABLE_COUNTS[i] );
            }
        }
    }
}
```

This is our Haiku validator

Simple Protocol

- Connect
- Send 3 lines of text
- Receive **HAIKU!** or **NOT A HAIKU:**
  - (plus a little reason why not)

ProtocolCodecFilter + TextLineCodecFactory

Bytes to Java String's. For free!

```
public class ToHaikuIoFilter extends IoFilterAdapter {

    @SuppressWarnings( { "unchecked" } )
    @Override
    public void messageReceived( NextFilter nextFilter, IoSession session, Object message )
        throws Exception
    {
        List<String> phrases = (List<String>) session.getAttribute( "phrases" );

        if ( null == phrases ) {
            phrases = new ArrayList<String>();
            session.setAttribute( "phrases", phrases );
        }

        phrases.add( (String) message );

        if ( phrases.size() == 3 ) {
            session.removeAttribute( "phrases" );

            super.messageReceived( nextFilter,
                                   session,
                                   new Haiku( phrases.toArray( new String[3] ) ) );
        }
    }
}
```

ToHaikuIoFilter

Three String's to a Haiku

```
public void testThreeStringsMakesAHaiku() throws Exception {
    Mock list = mock( List.class );
    list.expects( once() ).method( "add" ).with( eq( "two" ) ).will( returnValue( true ) );
    list.expects( once() ).method( "add" ).with( eq( "three" ) ).will( returnValue( true ) );
    list.expects( once() ).method( "toArray" ).with( isA( String[].class ) )
        .will( returnValue( new String[]{ "one", "two", "three" } ) );
    list.expects( exactly( 2 ) ).method( "size" )
        .will( onConsecutiveCalls( returnValue( 2 ), returnValue( 3 ) ) );

    Mock session = mock( IoSession.class );
    session.expects( exactly( 3 ) ).method( "getAttribute" ).with( eq( "phrases" ) )
        .will( onConsecutiveCalls( returnValue( null ), returnValue( list.proxy() ),
                                   returnValue( list.proxy() ), returnValue( list.proxy() ) ) );
    session.expects( exactly( 1 ) ).method( "setAttribute" )
        .with( eq( "phrases" ), eq( Collections.emptyList() ) );
    session.expects( exactly( 1 ) ).method( "removeAttribute" ).with( eq( "phrases" ) );

    IoSession sessionProxy = (IoSession) session.proxy();

    Mock nextFilter = mock( IoFilter.NextFilter.class );
    nextFilter.expects( once() ).method( "messageReceived" )
        .with( eq( sessionProxy ), eq( new Haiku( "one", "two", "three" ) ) );

    IoFilter.NextFilter nextFilterProxy = (IoFilter.NextFilter) nextFilter.proxy();

    filter.messageReceived( nextFilterProxy, sessionProxy, "one" );
    filter.messageReceived( nextFilterProxy, sessionProxy, "two" );
    filter.messageReceived( nextFilterProxy, sessionProxy, "three" );
}
```

Filter is very testable

(mock objects rock!)

```
public class HaikuValidatorIoHandler extends IoHandlerAdapter {

    private final HaikuValidator validator = new HaikuValidator();

    @Override
    public void messageReceived( IoSession session, Object message ) throws Exception {
        Haiku haiku = (Haiku) message;

        try {
            validator.validate( haiku );
            session.write( "HAIKU!" );
        } catch( InvalidHaikuException e ) {
            session.write( "NOT A HAIKU: " + e.getMessage() );
        }
    }
}
```

IoHandler is very simple

Validate Haiku, send result

```
public void testValidHaiku() throws Exception {
    Mock session = mock( IoSession.class );
    session.expects( once() ).method( "write" ).with( eq( "HAIKU!" ) );
    IoSession sessionProxy = (IoSession) session.proxy();

    handler.messageReceived( sessionProxy, new Haiku( "Oh, I drank too much.",
                                                      "Why, oh why did I sign up",
                                                      "For an eight thirty?" ) );
}

public void testInvalidHaiku() throws Exception {
    Mock session = mock( IoSession.class );
    session.expects( once() ).method( "write" )
      .with( eq( "NOT A HAIKU: phrase 1, 'foo' had 1 syllables, not 5" ) );
    IoSession sessionProxy = (IoSession) session.proxy();

    handler.messageReceived( sessionProxy, new Haiku( "foo", "a haiku", "poo" ) );
}
```

Also very testable

```
public class HaikuValidationServer {
    public static void main( String... args ) throws Exception {
        ExecutorService executor = Executors.newCachedThreadPool();
        SocketAcceptor acceptor =
          new SocketAcceptor( Runtime.getRuntime().availableProcessors(), executor );

        SocketAcceptorConfig config = new SocketAcceptorConfig();

        config.getFilterChain().addLast( "executor", new ExecutorFilter( executor ) );
        config.getFilterChain().addLast( "to-string",
          new ProtocolCodecFilter( new TextLineCodecFactory( Charset.forName( "US-ASCII" ) ) ) );
        config.getFilterChain().addLast( "to-haiki", new ToHaikuIoFilter() );

        acceptor.bind( new InetSocketAddress( 42458 ), new HaikuValidatorIoHandler(), config );
    }
}
```

Very easy to hook it all up

Questions?

Thank You!