



J-Fall

11 november 2009 Spant!



The Quest for Parallelism

How to 'Upgrade' Your Application

ir. Jan-Hendrik Kuperus

Sogeti Nederland BV

Twitter: @jhkuperus

Blog: java.sogeti.nl/JavaBlog

Wave: kramor@googlewave.com



J-Fall

11 november 2009 Spant!



Errata



Today's Agenda

- Parallelism Today
- Old Patterns
- Introducing Task Modeling
- Easy Wins



J-Fall

11 november 2009 Spant!



Parallelism today...

.nl.
jug



Parallel

- Most applications are multi-processor
– Most applications are multi-processor
- Current trends in multi-processor systems
– Single core
– Dual core
– Triple core
– Quad core
– Six core 1%

78% of
available CPUs
today are multi-
core!



Parallelism today

- Number of cores influences designs

Single Core Machines	Multi Core Machines
Should not be swamped. Too much concurrency will hamper throughput.	Should be kept busy. An idle core equals lost potential!
Domain model focuses on consistency	Domain model focuses on high concurrent access
May hide subtle bugs since threads are never run simultaneously	Will demonstrate all bugs eventually



Parallelism today

- Scalability was a static requirement
 - Applications now need to be 'elastic'
 - Why pay for cores we don't use?
- Say hello to





Parallelism today

- Java 5, 6 and 7 simplify synchronization
 - Many applications are still on Java 1.4
- Java also shifts to Task modeling



Helping Features in Java

- ExecutorServices
 - Manages threads for you
- Futures
 - Great for asynchronous processing
- Atomic objects
 - Enables atomic variables for Java engineers
- Latches / Semaphores
 - Many different synchronization strategies



Thread management today

- Provided by application server
 - Servlet model
 - Enterprise Java Bean model
- Self-managed
 - Focused threads
 - Thread pools



J-Fall

11 november 2009 Spant!



Picking apart some 'old' patterns...



Common 'patterns' - Spawner

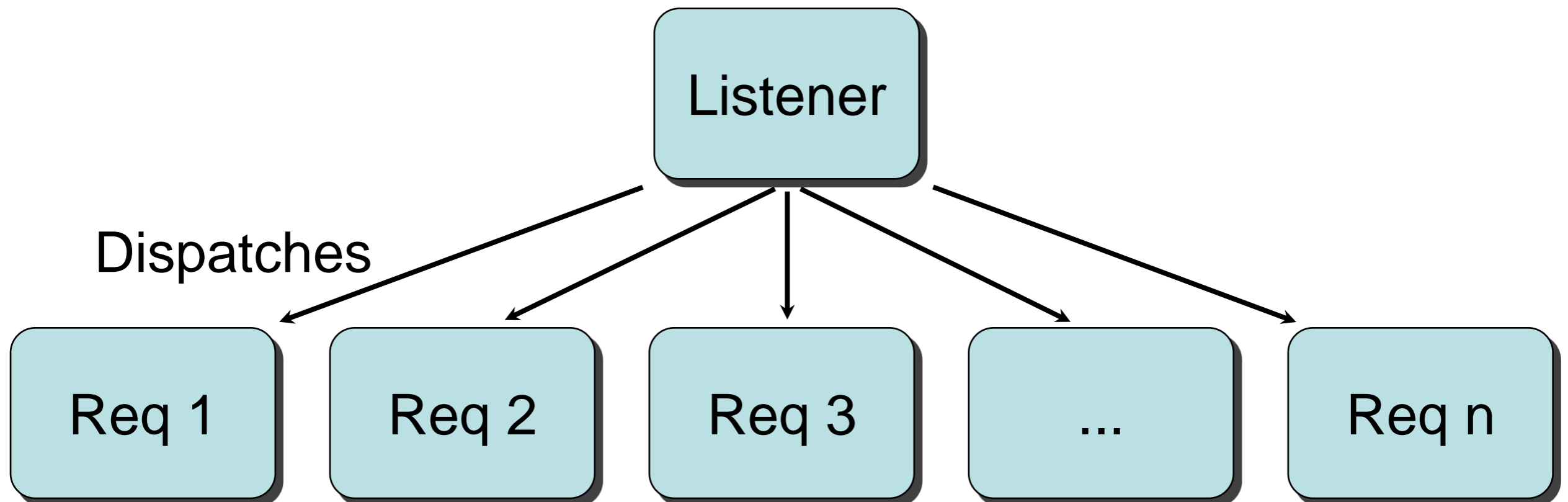
- Thread per request (Spawner)

// Request came in, create Thread to handle it

```
Runnable requestHandler = new Runnable(request);
```

```
Thread newRequest = new Thread(requestHandler);
```

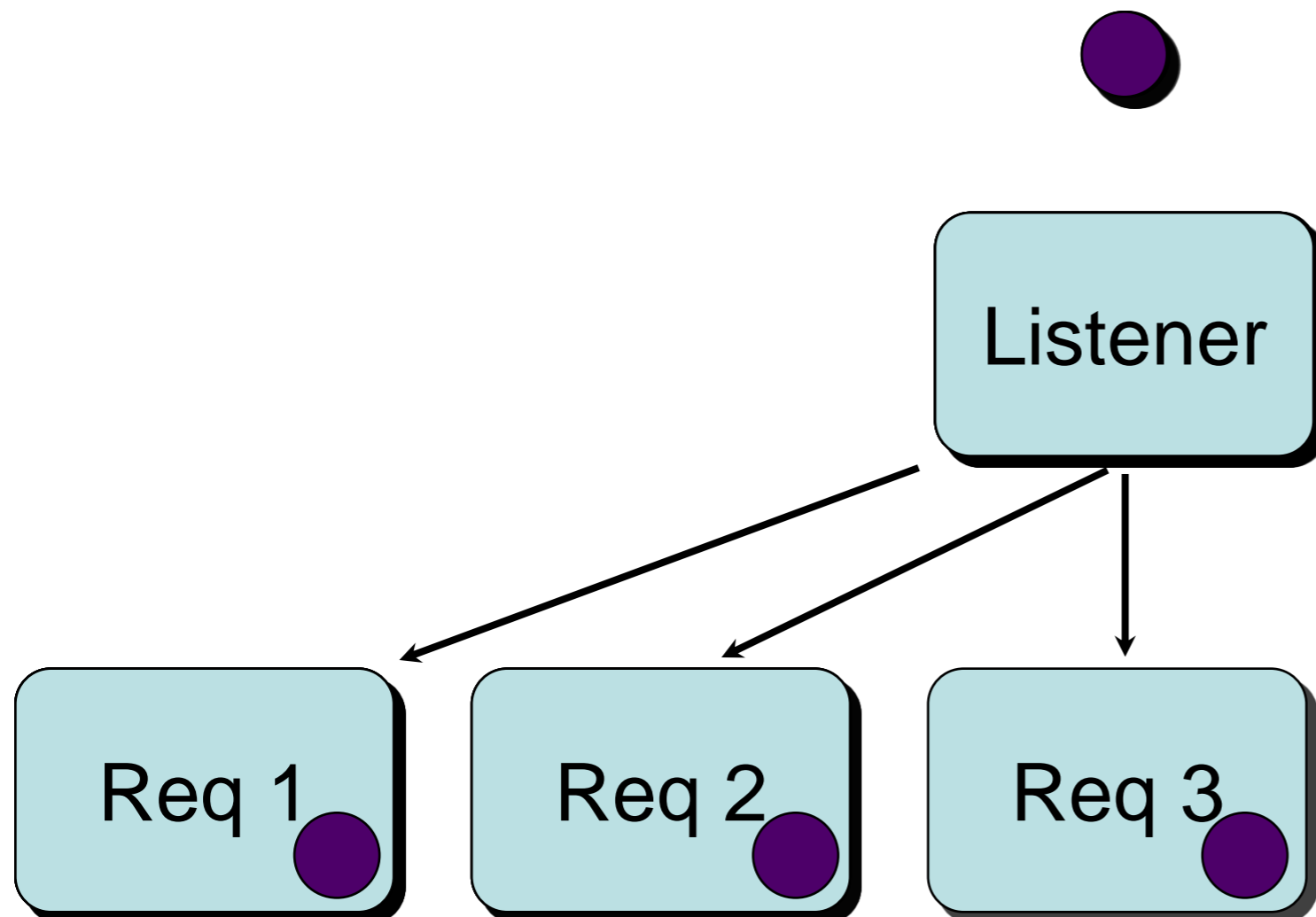
```
newRequest.start();
```





Spawner advantages

- Listener can fire and forget
 - Each task is after all a Thread





Spawner advantages

- Tasks are completely independent
 - If the data model is!
 - And nothing escapes the tasks/threads



Spawner advantages

- If a core is free, a new request can take it
 - Response time usually stable under low load



Spawner disadvantages

- Request bursts will clog op the scheduler
 - Response time degradation
- Threads are never reused
 - Threads are expensive objects!
- Scales reasonably
 - But no profit from ‘quiet’ periods



Common 'patterns' - Pipelining

- Thread per action type (Pipelining)

// Create a producer Thread

```
Producer p = new Producer();  
Thread producer = new Thread(p);
```

// Create stage one of the pipeline

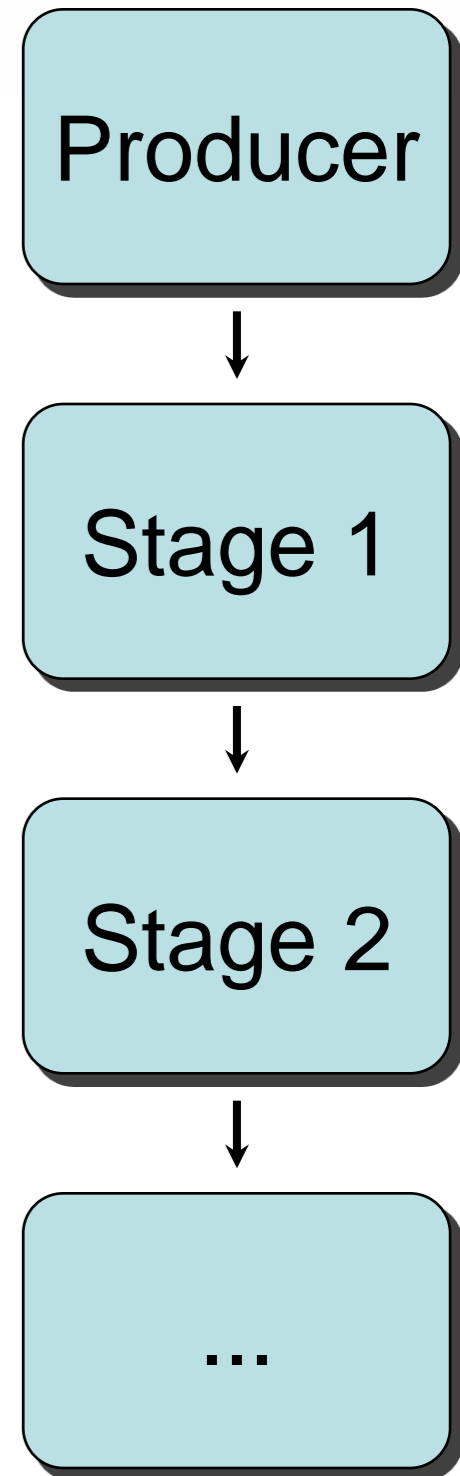
```
StageOne so = new StageOne();  
Thread stageOne = new Thread(new StageOne());
```

// Create a consumer Thread

```
Consumer c = new Consumer();  
Thread consumer = new Thread(c);
```

// Connect the stages

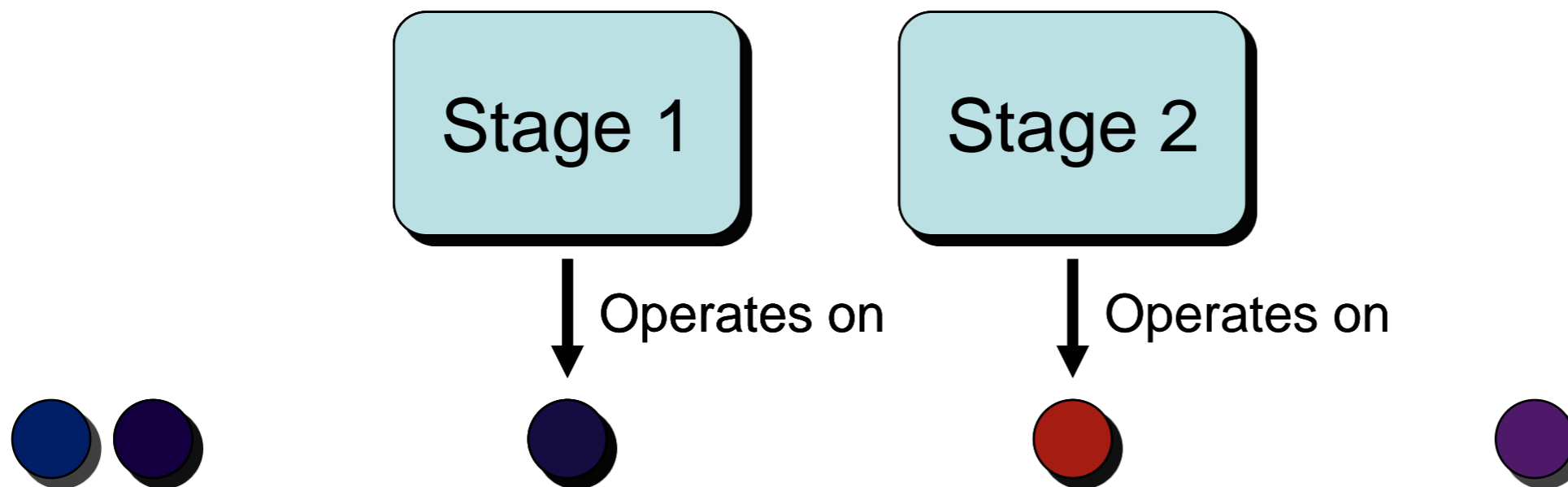
```
so.setInputQueue(p.getOutputQueue());  
c.setInputQueue(so.getOutputQueue());
```





Pipelining advantages

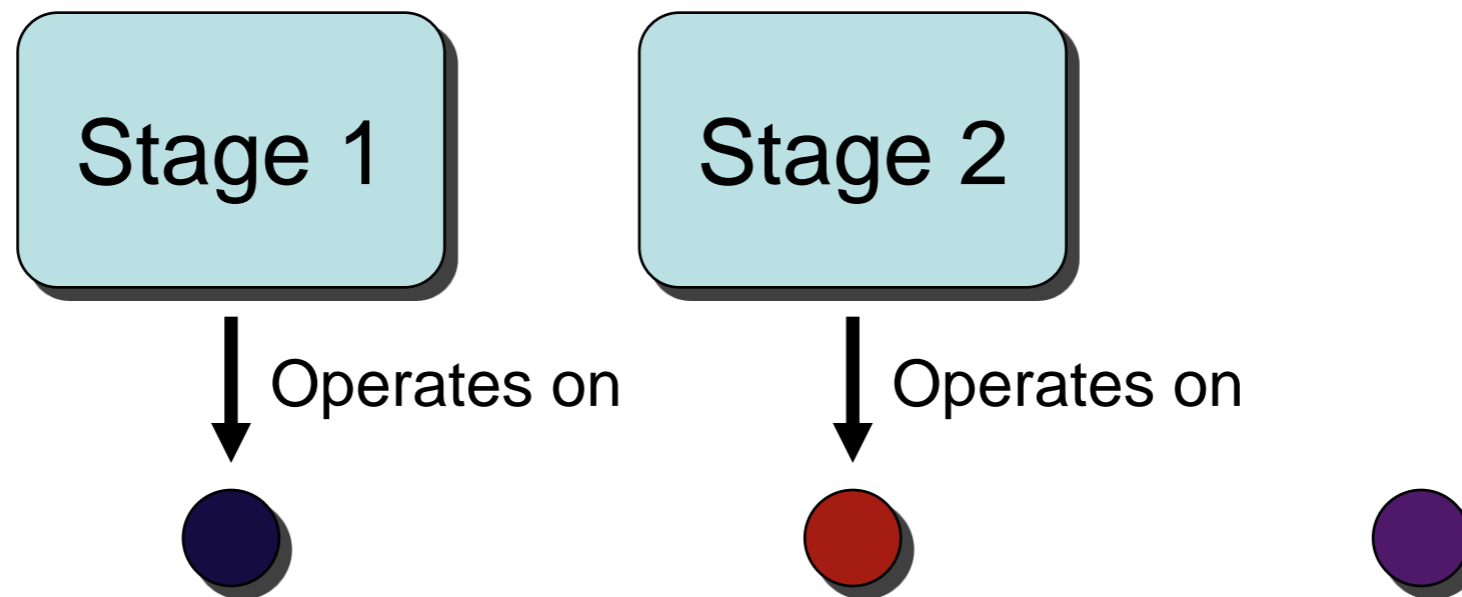
- Clear task boundaries
 - Task data is not shared in stages





Pipelining advantages

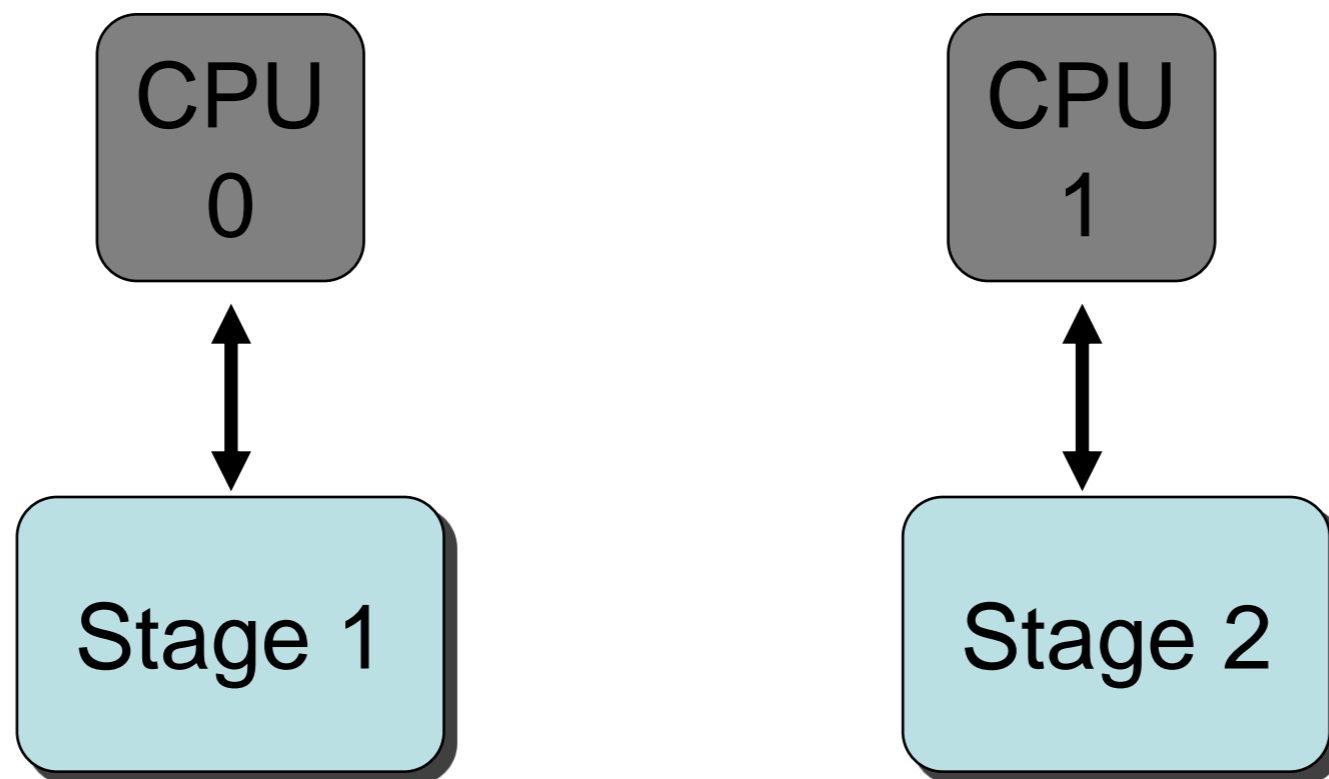
- Linear dependency between tasks
 - Stage 2 can assume stage 1 completed





Pipelining advantages

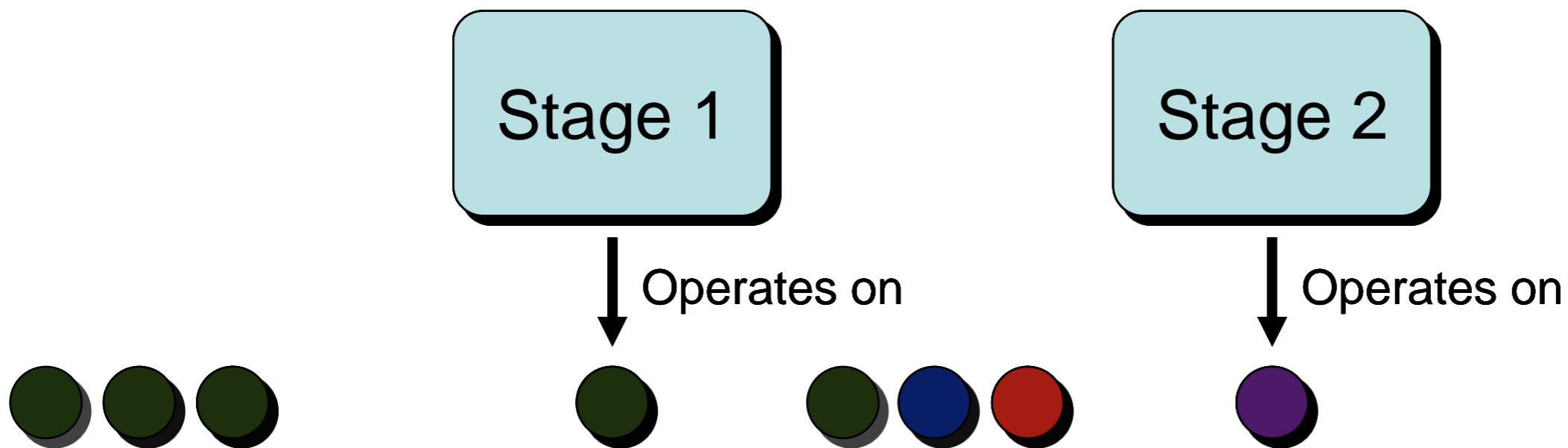
- Utilizes fixed amount of cores
 - Works well in single core environment
 - Blocking stages won't block the CPU

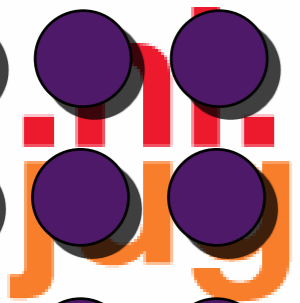




Problems with pipelining

- Tasks must take equal amounts to complete
 - Slow tasks will eventually clog the system

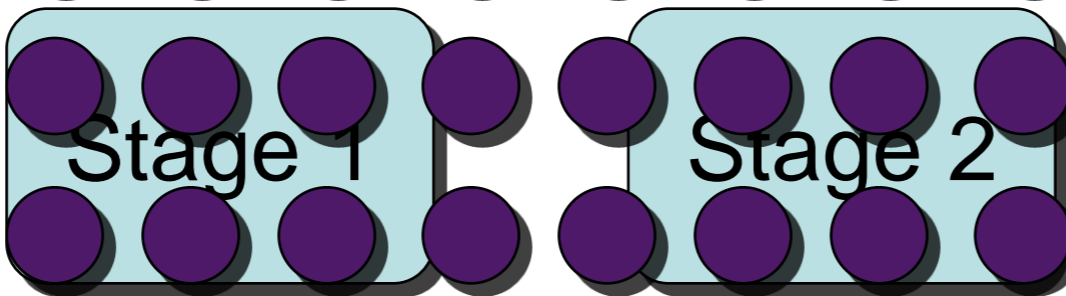




Problems with pipelining

- Clocks on sub-blocks

– Each stage is a single pair of flip-flops

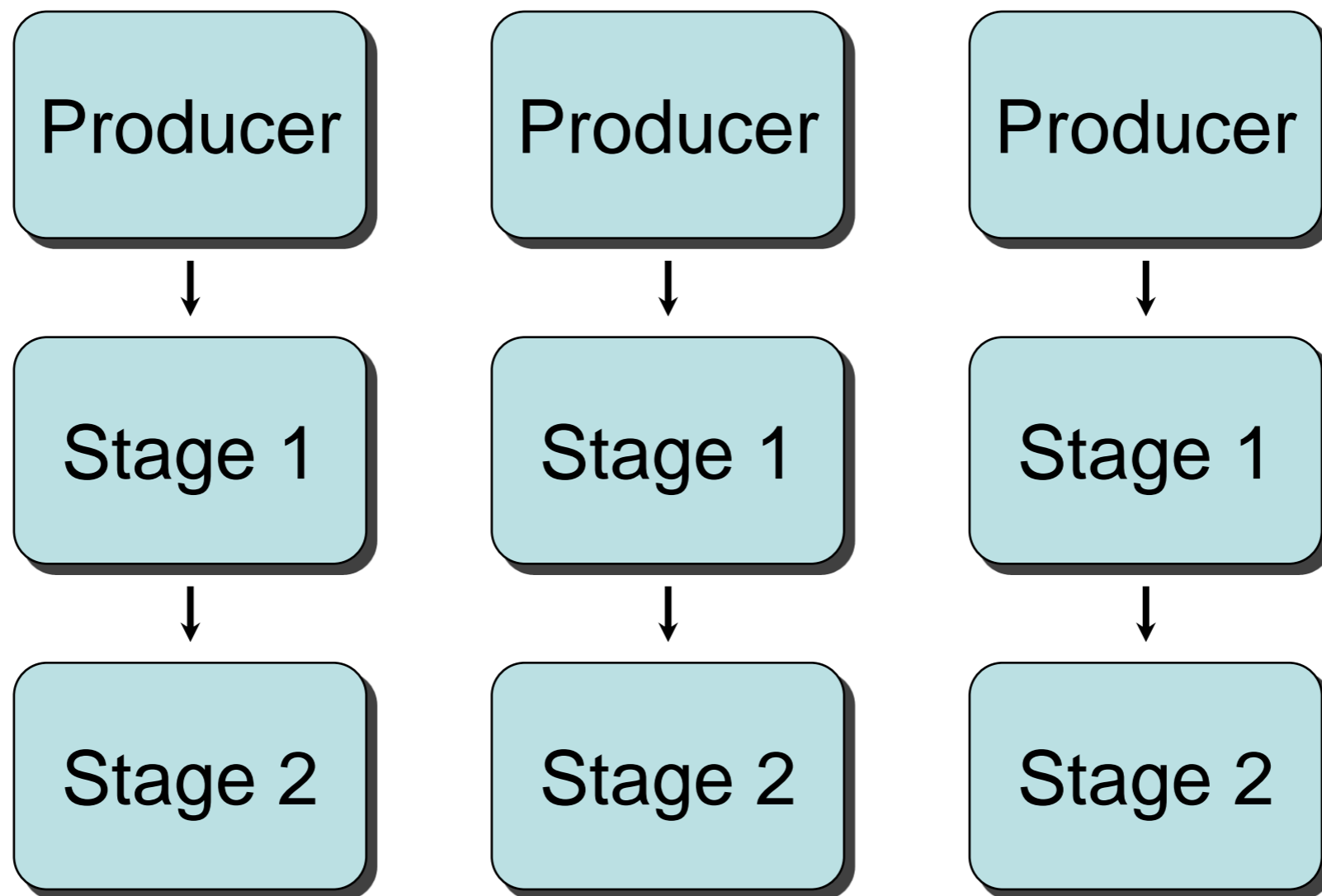


↓ Clocks on sub-blocks



Problems with pipelining

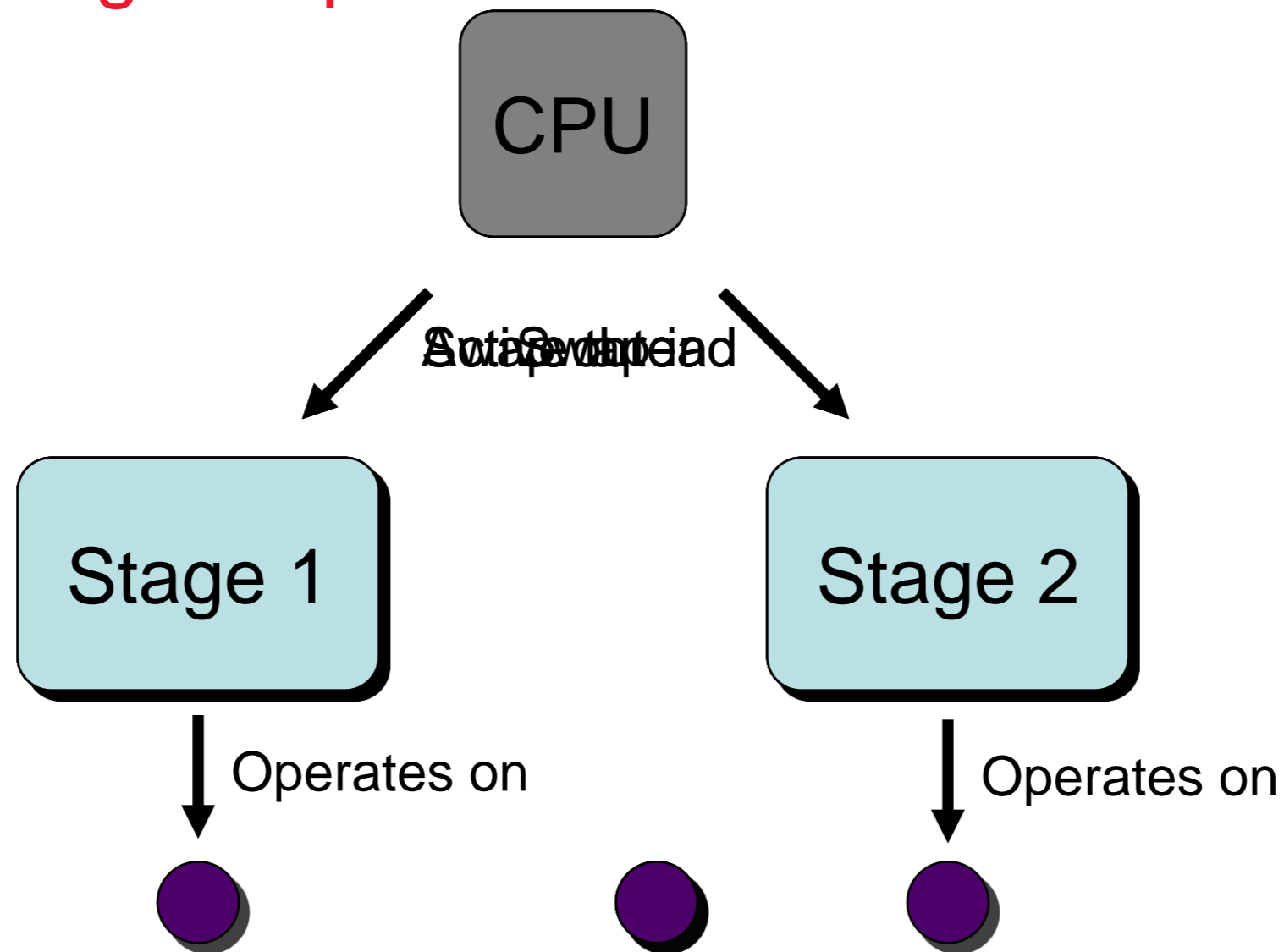
- Scaling requires duplication of pipeline
 - Also a lot of cores!





Problems with pipelining

- One task requires many context switches
 - More stages equals more switches





The 'old' patterns

- Both models rely on Thread modeling
 - What should this Thread do?
- Force us to manage Threads manually
 - Including scaling to meet a load
- Generally unfit for cloud frameworks
 - Clouds want to manage threads
 - Even stricter than App Servers



J-Fall

11 november 2009 Spant!



Introducing task modeling...



Introducing Task Modeling

- So what's the difference with Thread Modeling?
 - Basically: leave the Threads to Java
 - Think of efficient task decomposition



Introducing Task Modeling

- Example: web mashup request

// Fetch remote information

```
ResultSet data1, data2, data3;
```

```
data1 = includeWebData("http://foo.bar/service");
```

```
data2 = includeWebData("http://example.com/weather");
```

// Fetch local database information

```
data3 = service.getDataSet();
```

// Display the view

```
renderView(data1, data2, data3);
```



Web mashup request

Request thread



- Render view
- Fetch database query
- Fetch <http://example.com/weather>
- Fetch <http://foo.bar/service>



Web mashup with tasks

```
Future<ResultSet> data1, data2, data3;
```

```
// Dispatch subtasks
```

```
data1 = dispatchInclude("http://foo.bar/service");
```

```
data2 = dispatchInclude("http://example.com/weather");
```

```
data3 = dispatchQuery();
```

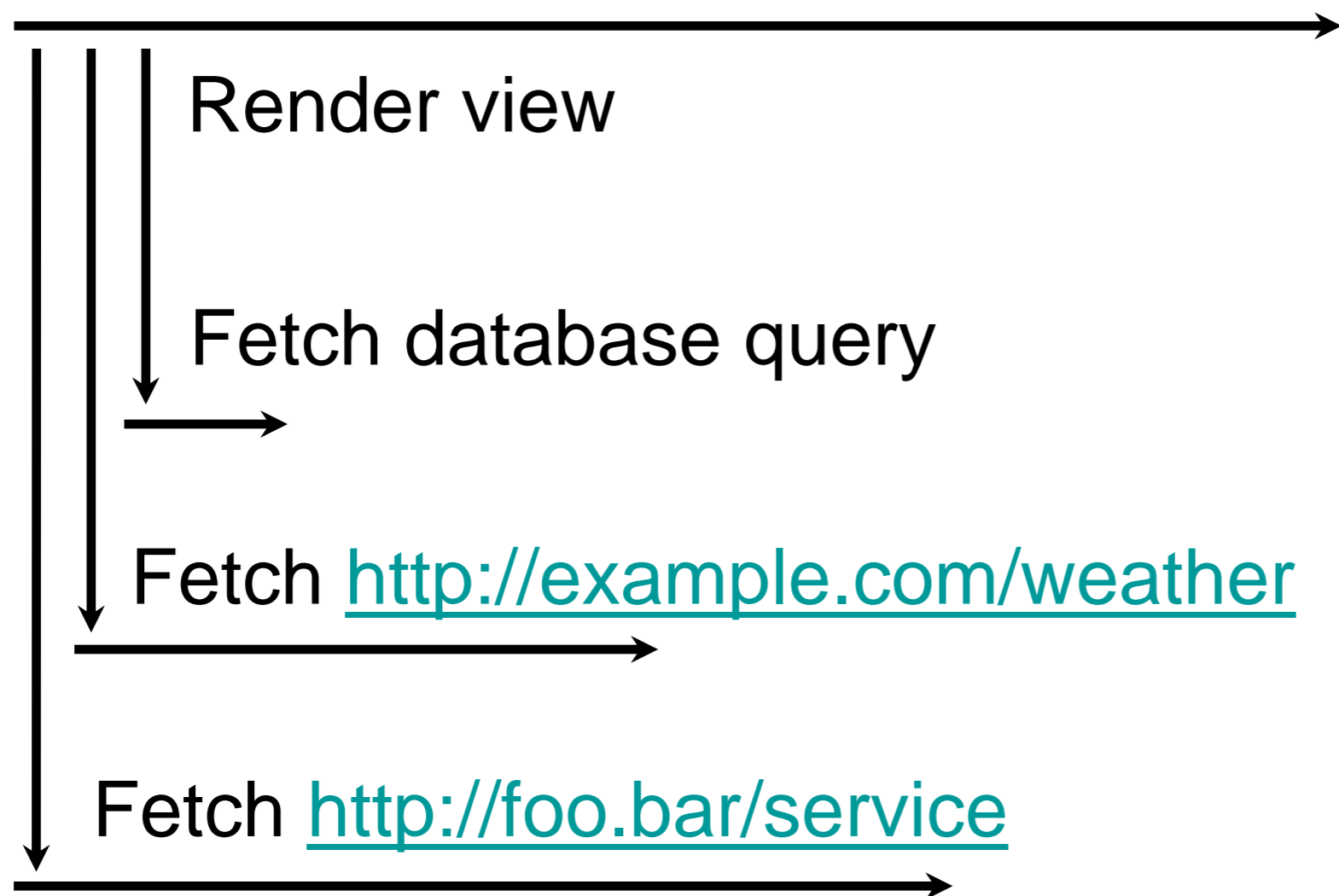
```
// Start rendering the view
```

```
renderView(data1, data2, data3);
```



Web mashup with tasks

Request thread





Getting there

- You need an `ExecutorService`
 - `CachedThreadPool`
 - `FixedThreadPool`
 - `ScheduledThreadPool`
 - `(SingleThreadExecutor)`
 - `(SingleThreadScheduledExecutor)`



Getting there

- You need to think Runnables, Callables
- Runnables are 'legacy'
 - They are essentially void-methods
 - You might already have some of these!
- Callables are the future
 - Callable is Runnable with result
 - `Future<T> submit(Callable<T> task)`



Almost there!

- You need a synchronization point!
- Either use `Future.get()`
 - Not too early though!
- `java.util.concurrent` provides nice defaults
 - `CountDownLatch`
 - `CyclicBarrier`





Advantage of task modeling

- Scales dynamically
- Predictable behavior during load and idle time
- Prepared for cloud computing
 - Many frameworks provide executor services



Potential problems

- Execution order of tasks is unknown
- Choosing invalid task boundaries
 - Shared data is murderous for this approach



J-Fall

11 november 2009 Spant!



Go forth and try it!



J-Fall

11 november 2009 Spant!



Easy wins for your applications...



Easy wins

- Logging
 - Log4J provides the AsyncAppender
 - Relieves your task from I/O

```
<appender name="ASYNC"  
  class="org.apache.log4j.AsyncAppender">  
  <appender-ref ref="FILE"/>  
</appender>
```

```
<appender name="FILE" ...>  
  ...  
</appender>
```



Easy wins

- Document generation
 - Kick off generation in `ExecutorService`
 - Poll frequently for task completion
 - Serve document or error message



Easy wins

- Large dataset updates
 - Basically identical to document generation
 - Fire off query
 - Poll for completion



Easy wins

- But I am using a Java EE appserver...
 - Just use an `ExecutorService`
 - But be aware of the transaction consequences
- After all, the EE spec is outdated on this subject
 - Improvements have been promised...
 - Some servers provide schedulers



J-Fall

11 november 2009 Spant!



Questions?