DON'T BLOCK YOUR MOBILES AND INTERNET OF THINGS

Use non blocking I/O for scalable and resilient server applications

MAGNUS LARSSON, PÄR WENÅKER, ANDERS ASPLUND

2014-10-23 | CALLISTAENTERPRISE.SE

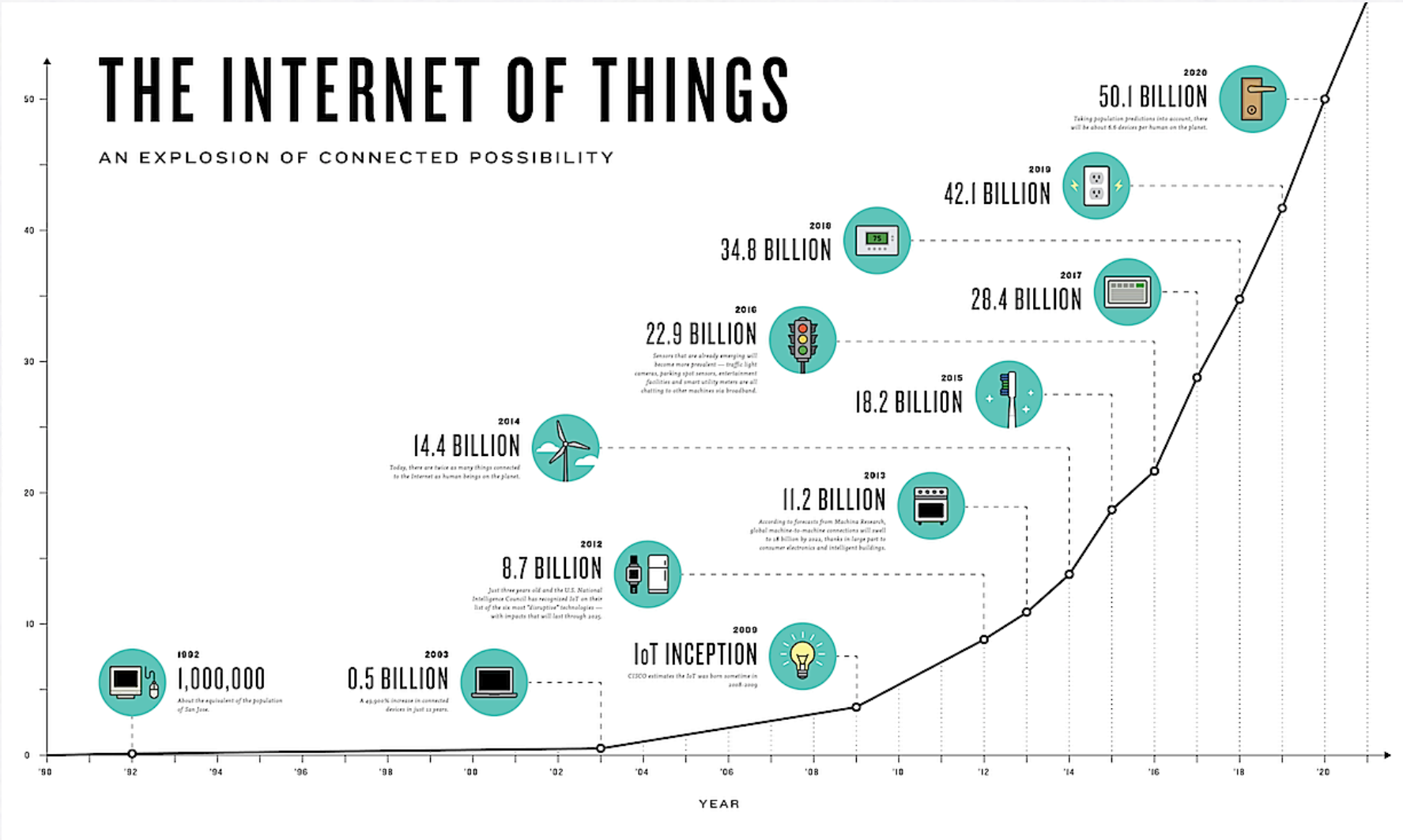
CALLISTA
– ENTERPRISE –

AGENDA

- The Big Picture
- Demonstration
- Details
- Experiences from a real life projects
- Summary & next step

The Big Picture

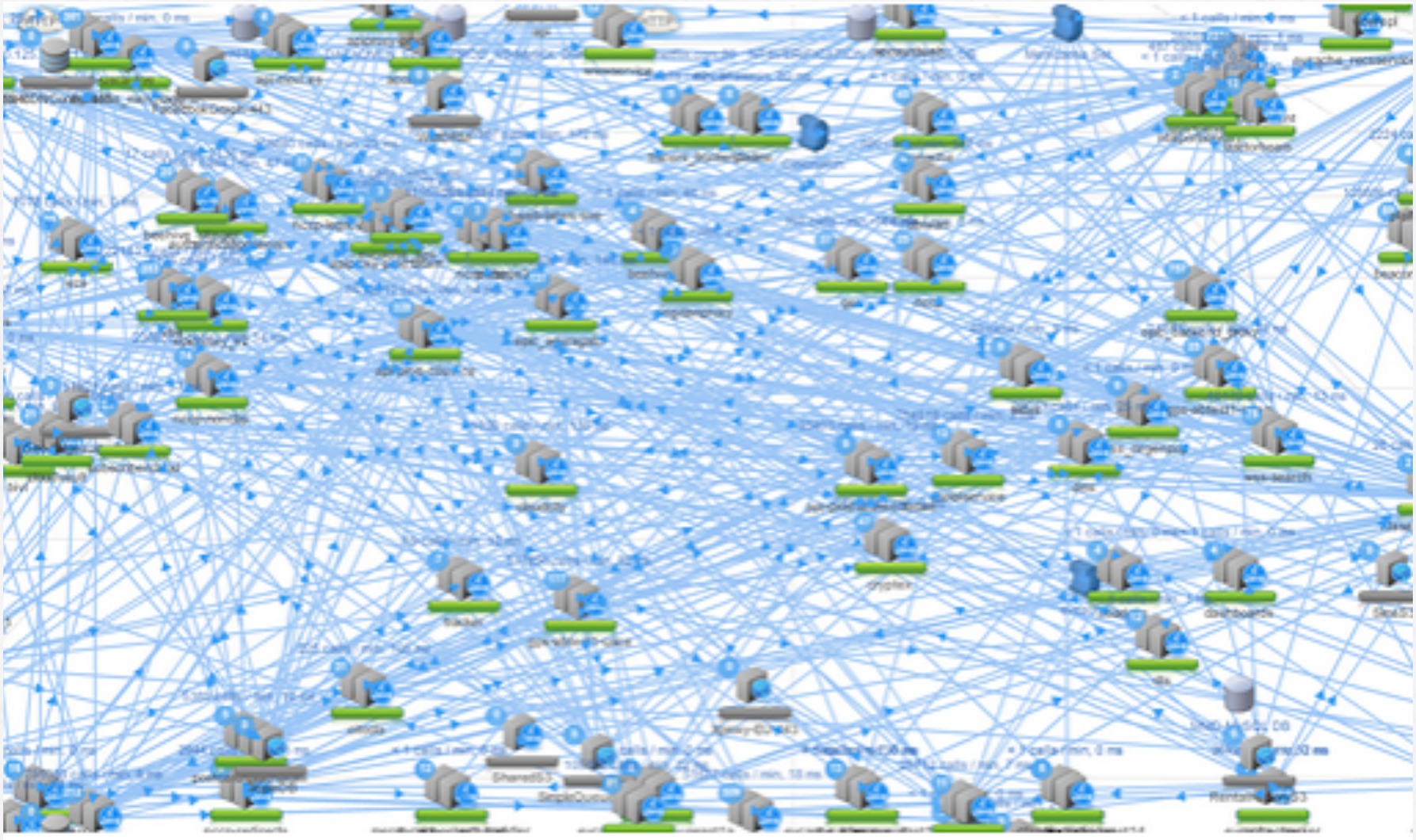
THE SCALABILITY CHALLENGE…



Source: http://www.theconnectivist.com/2014/05/infographic-the-growth-of-the-internet-of-things/

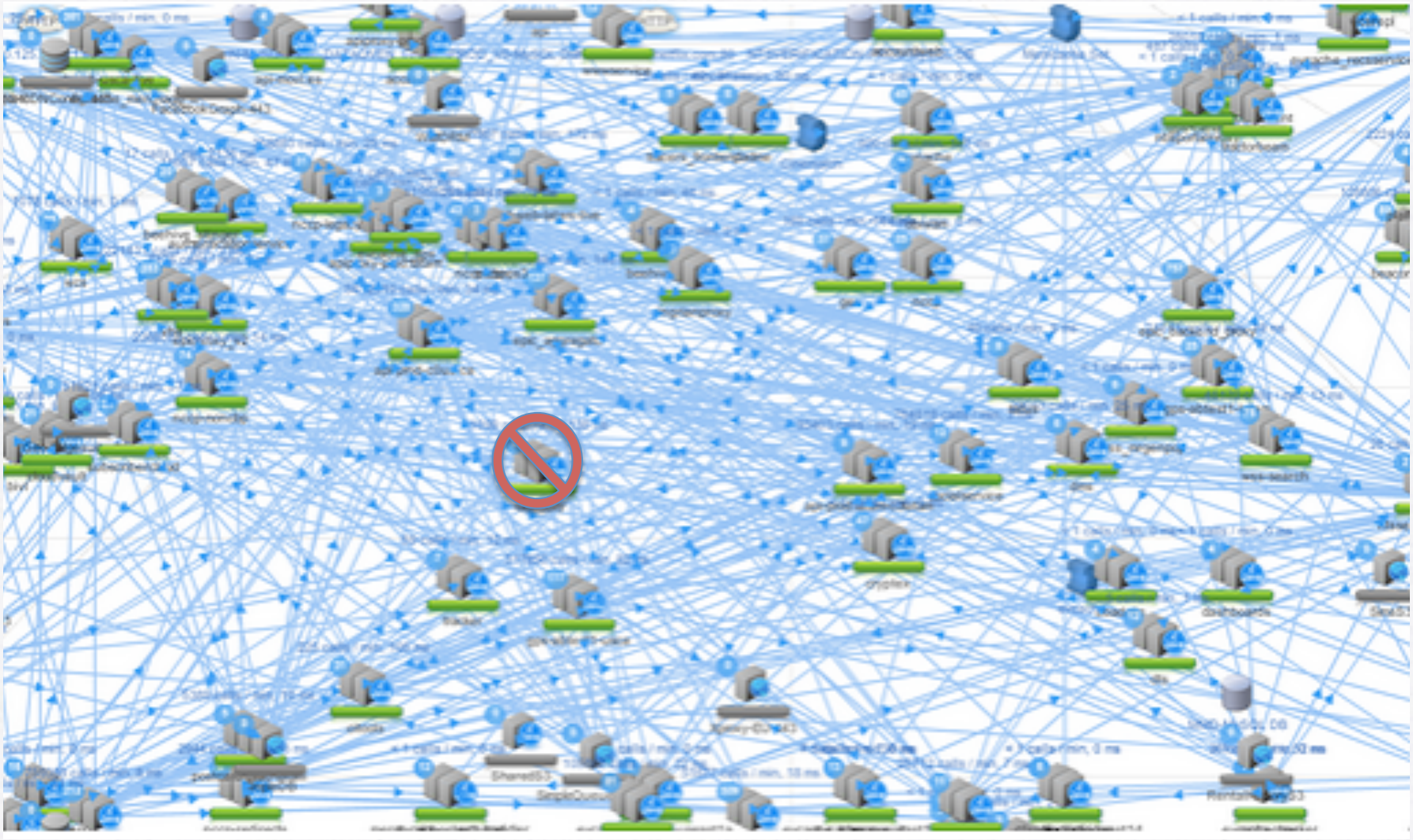
CALLISTA
– ENTERPRISE –

...SERVICES ARE CONNECTED...



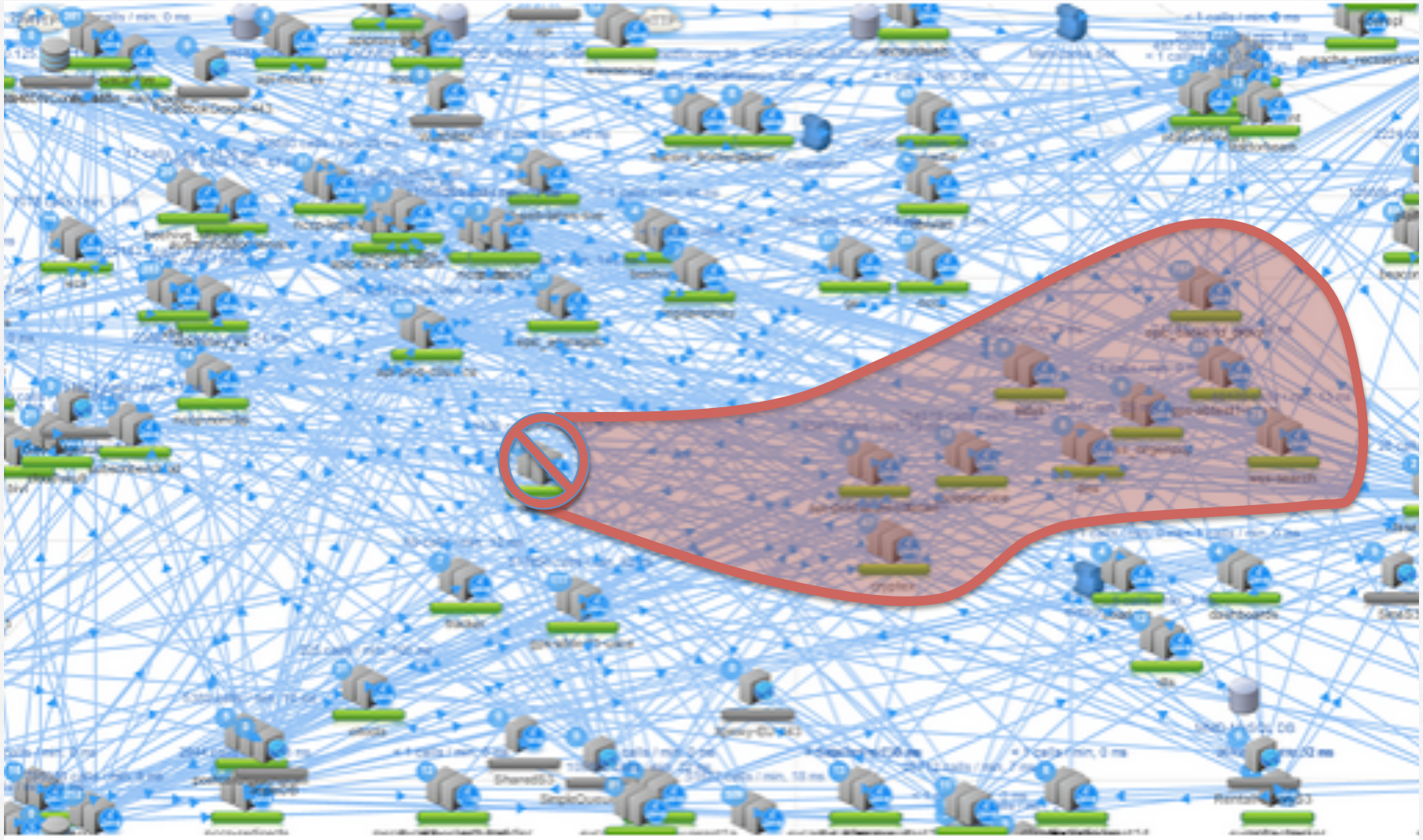
Source: http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html

...SERVICES FAILS...



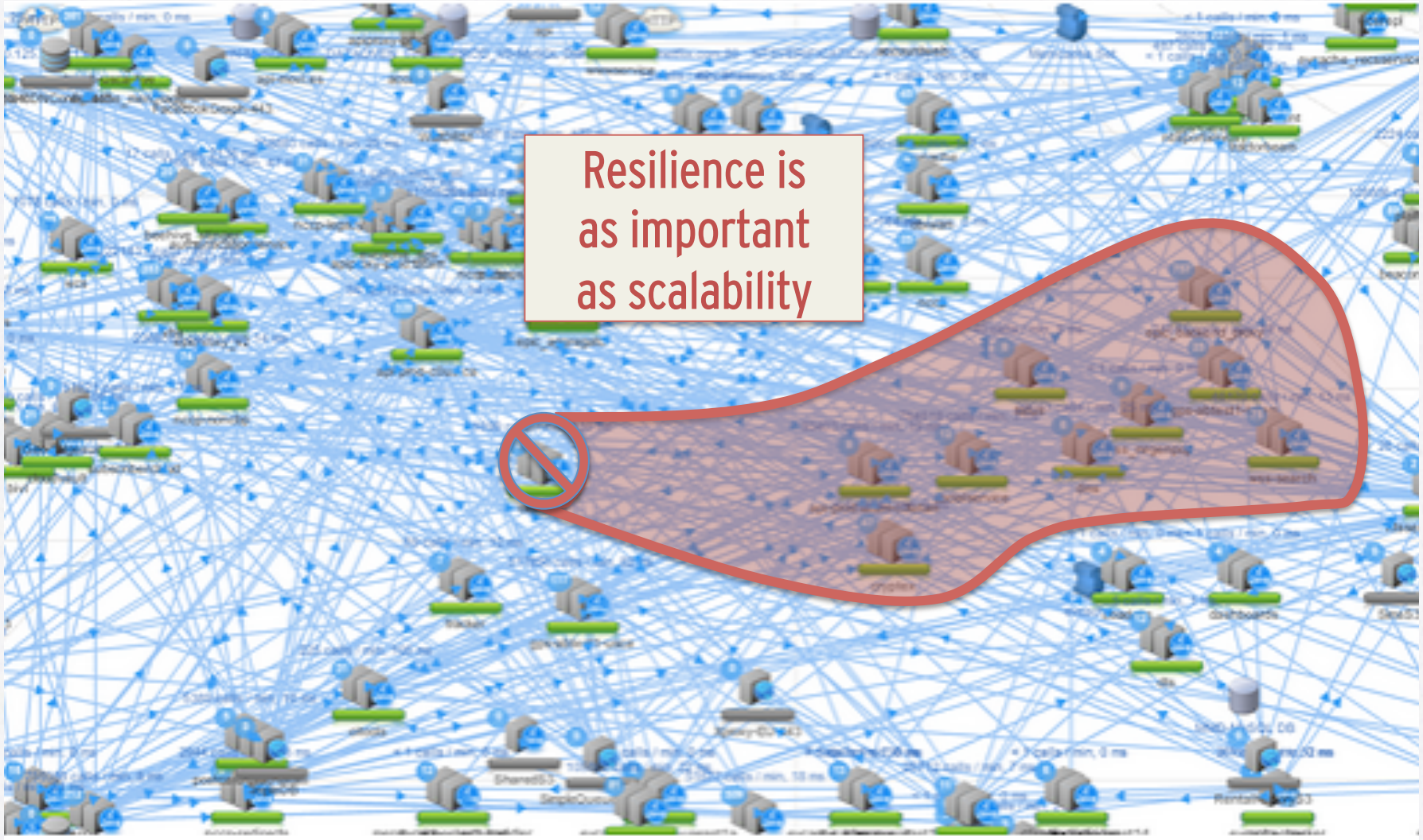
Source: http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html

WATCH OUT FOR THE DOMINO EFFECT!



Source: http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html

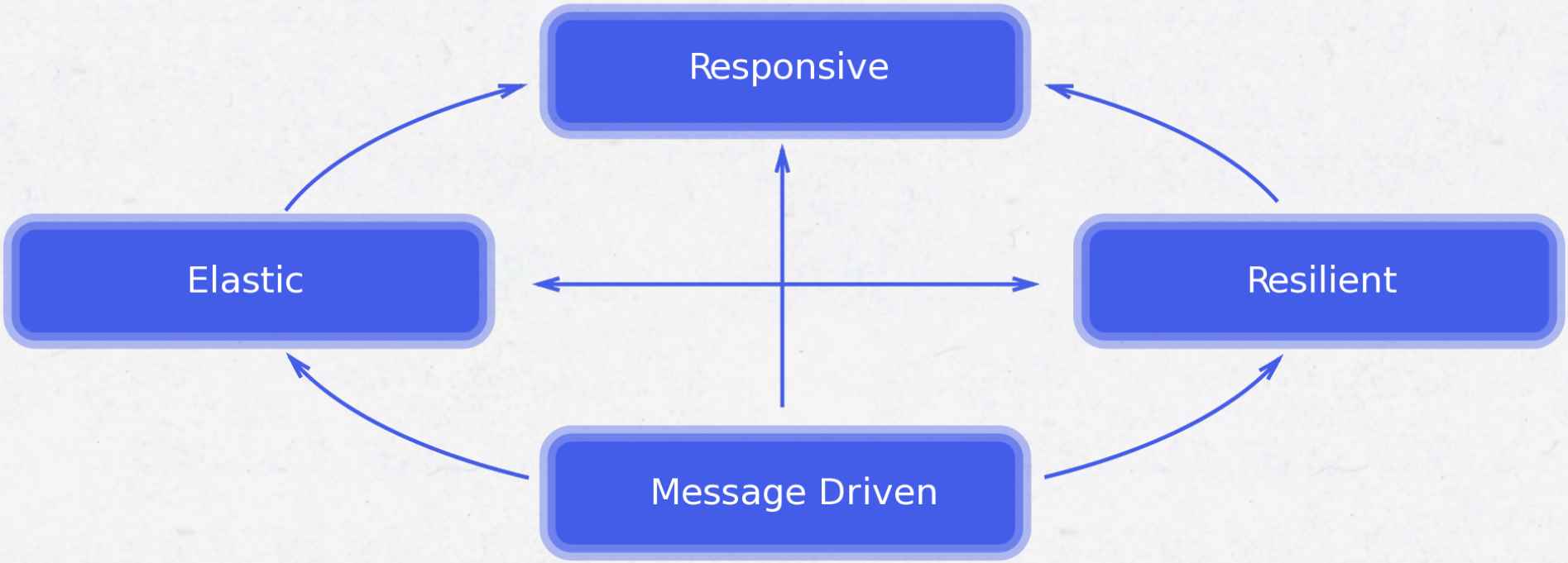
WATCH OUT FOR THE DOMINO EFFECT!



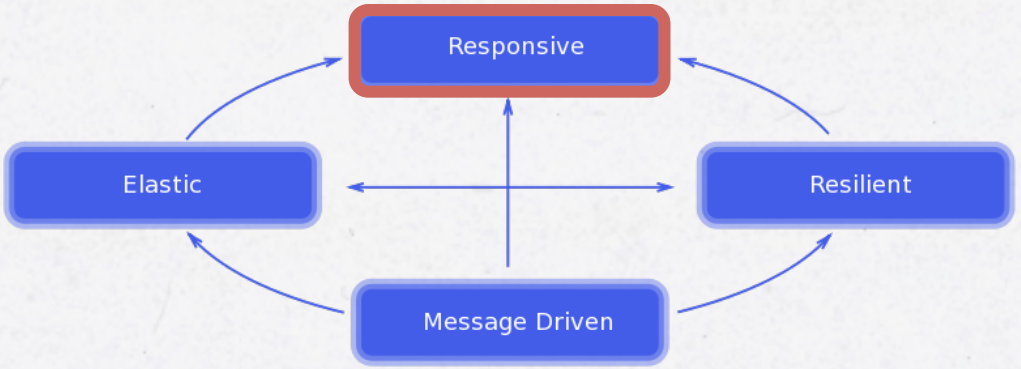
Source: http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html

THE REACTIVE MANIFESTO

- http://www.reactivemanifesto.org

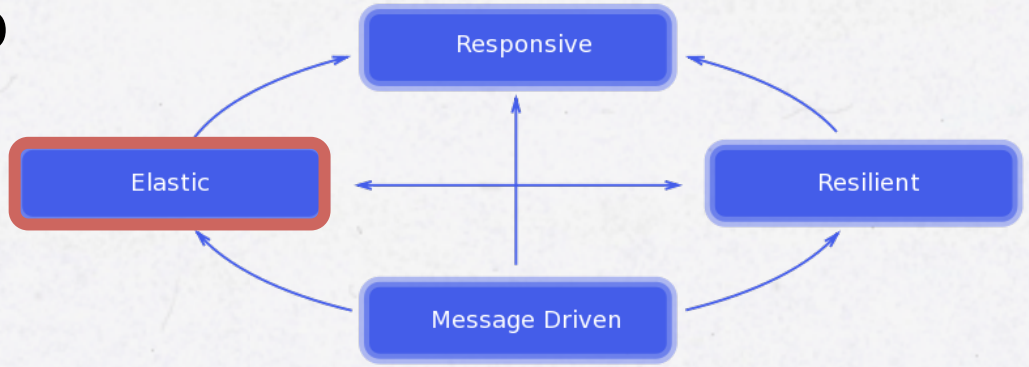


THE REACTIVE MANIFESTO



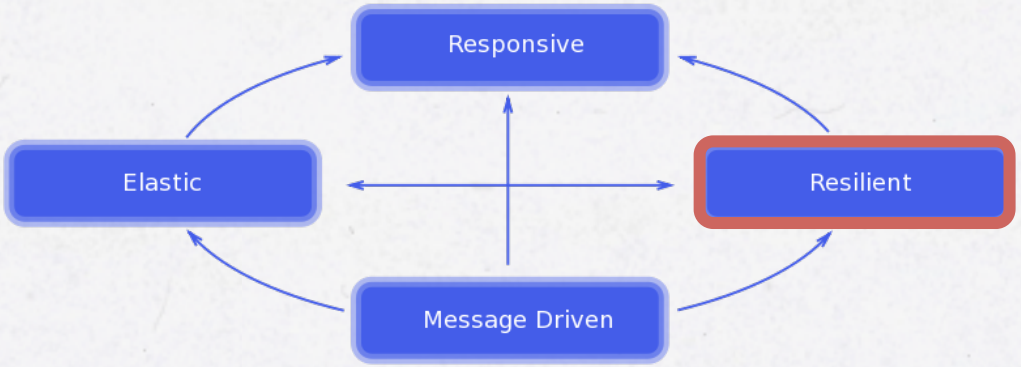
- **Responsive**
  - *"The system **responds in a timely manner** if at all possible. Responsiveness is the cornerstone of **usability and utility**, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing **rapid and consistent response times**, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behaviour in turn simplifies error handling, builds end user confidence, and encourages further interaction."*

THE REACTIVE MANIFESTO



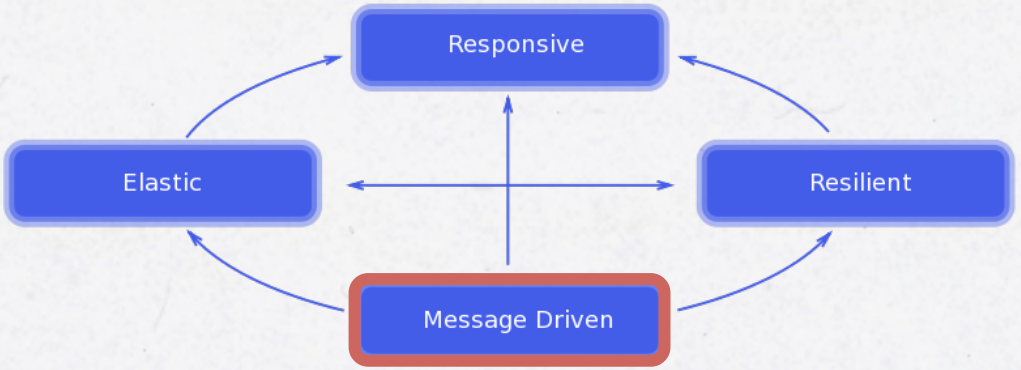
- **Elastic (scalable)**
  - *"The system **stays responsive under varying workload**. Reactive Systems can react to changes in the input rate by increasing or decreasing the **resources allocated to service these inputs**. This implies designs that have no contention points or central bottlenecks, resulting in the ability to **shard or replicate components** and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, scaling algorithms by providing relevant live performance measures. They achieve elasticity in a **cost-effective way on commodity hardware** and software platforms."*

THE REACTIVE MANIFESTO



- **Resilient**
  - *"The system **stays responsive in the face of failure**. This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by **replication, containment, isolation and delegation**. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The **client of a component is not burdened with handling its failures**."*

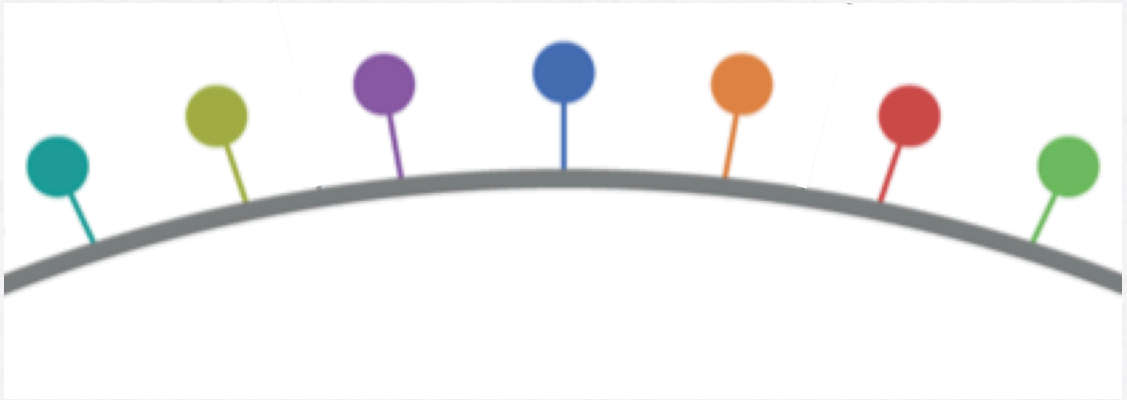
THE REACTIVE MANIFESTO



- **Message driven**
  - *" Reactive Systems rely on **asynchronous message-passing** to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing **enables load management, elasticity, and flow control** by shaping and monitoring the message queues in the system and **applying back-pressure when necessary**. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. **Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead**."*

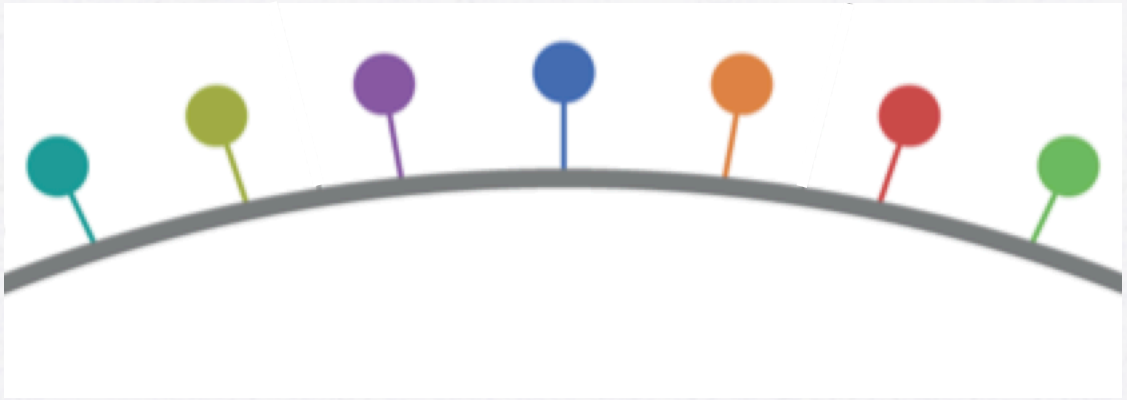
HOW DO WE FULFILL THE REACTIVE MANIFESTO???

?

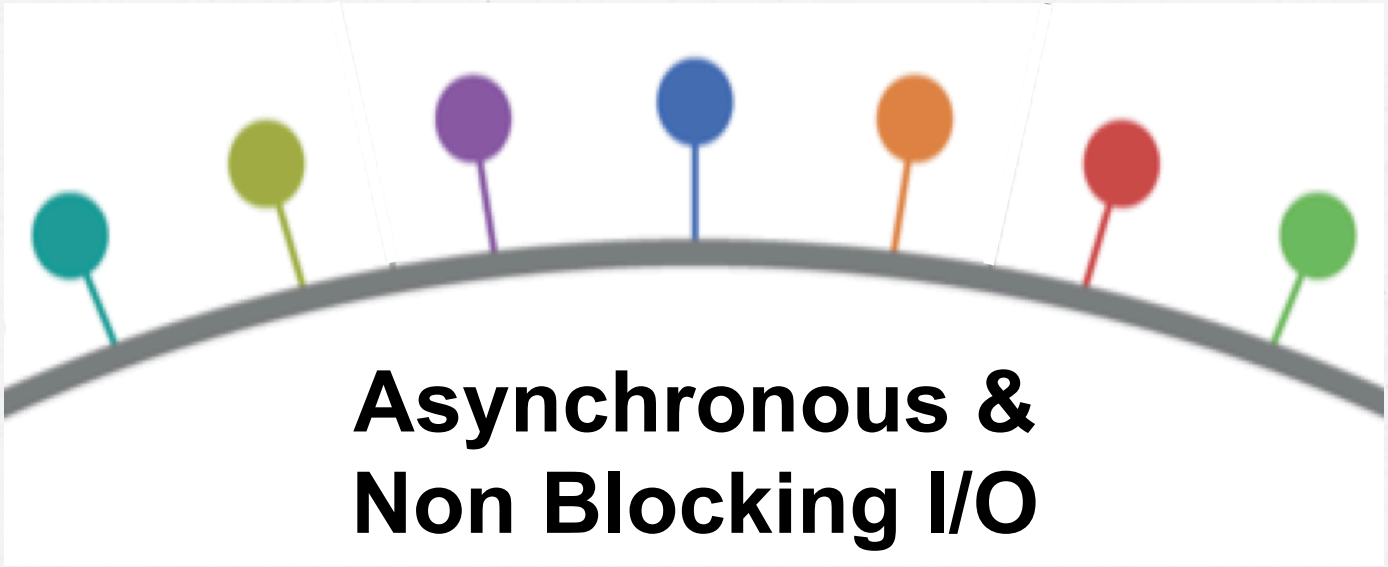


TECHNICAL CHALLENGES…

- Large scale of concurrent requests
  - Load balancing
  - Circuit breaker
- Large number of services
  - Distributed configuration
  - Service registration and discovery
- Large number of connected services
  - Configurable routing
  - Resilient service-to-service calls
- Responsive services require more efficient protocols than HTTP
  - WebSockets
  - IoT protocols (http://iot.eclipse.org/)
    - MQTT, CoAP, LWM2M…



THE FUNDAMENT OF REACTIVE SYSTEMS



ASYNCHRONOUS PROCESSING VS NON BLOCKING I/O

…at least for the scope of this presentation…

- **Asynchronous processing ≠ non blocking I/O**
- Asynchronous processing means that a thread hands over processing to another thread
- Non blocking I/O means that a thread is not waiting for external resources, such as databases or another services

➜Asynchronous processing can either use blocking I/O or non blocking I/O!

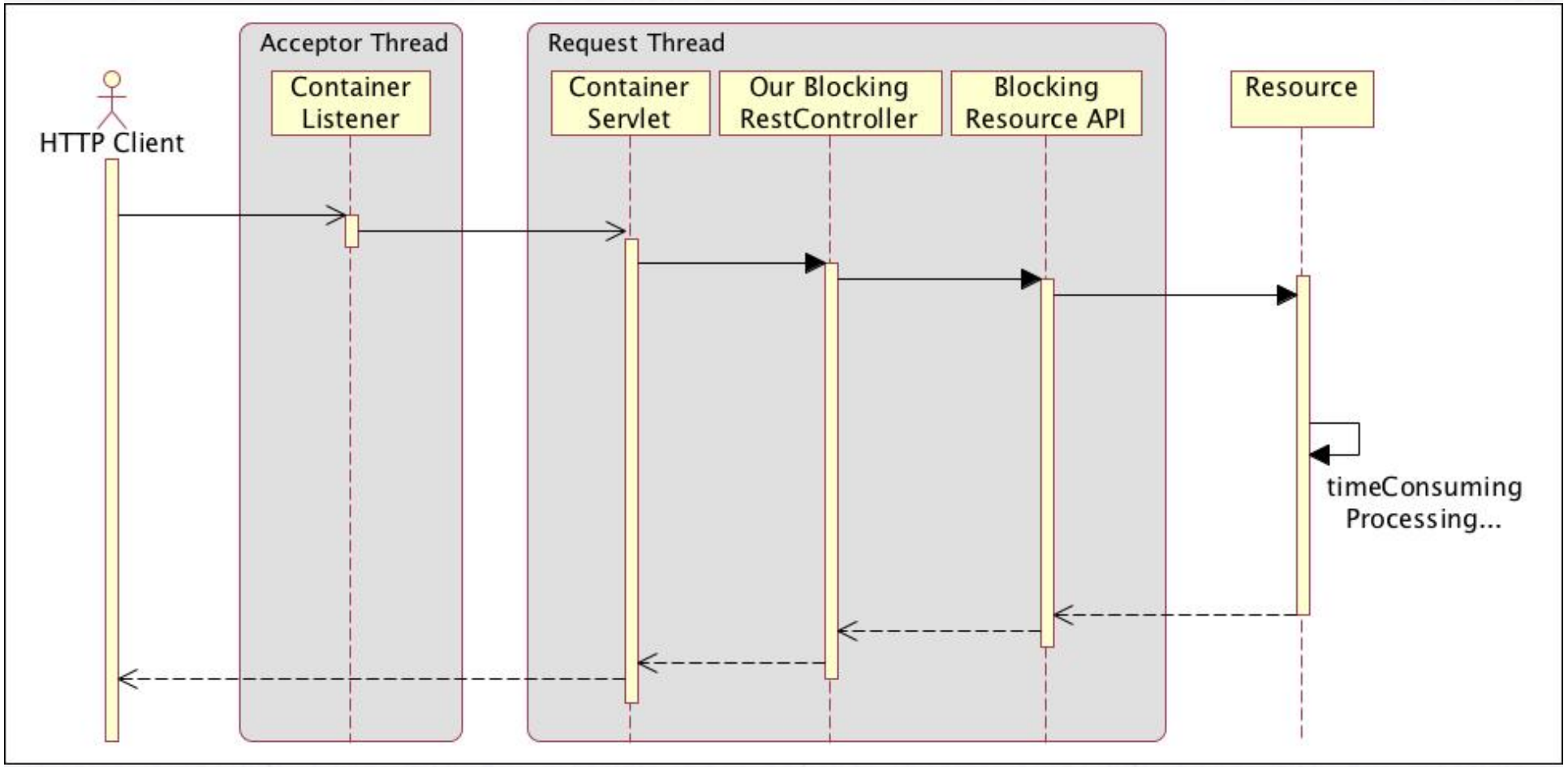
  - Stay tuned for examples…

IS NON-BLOCKING I/O NEW?

- **No!!!**
- A short history lesson..
  - Non blocking I/O has been supported in operating systems for ever

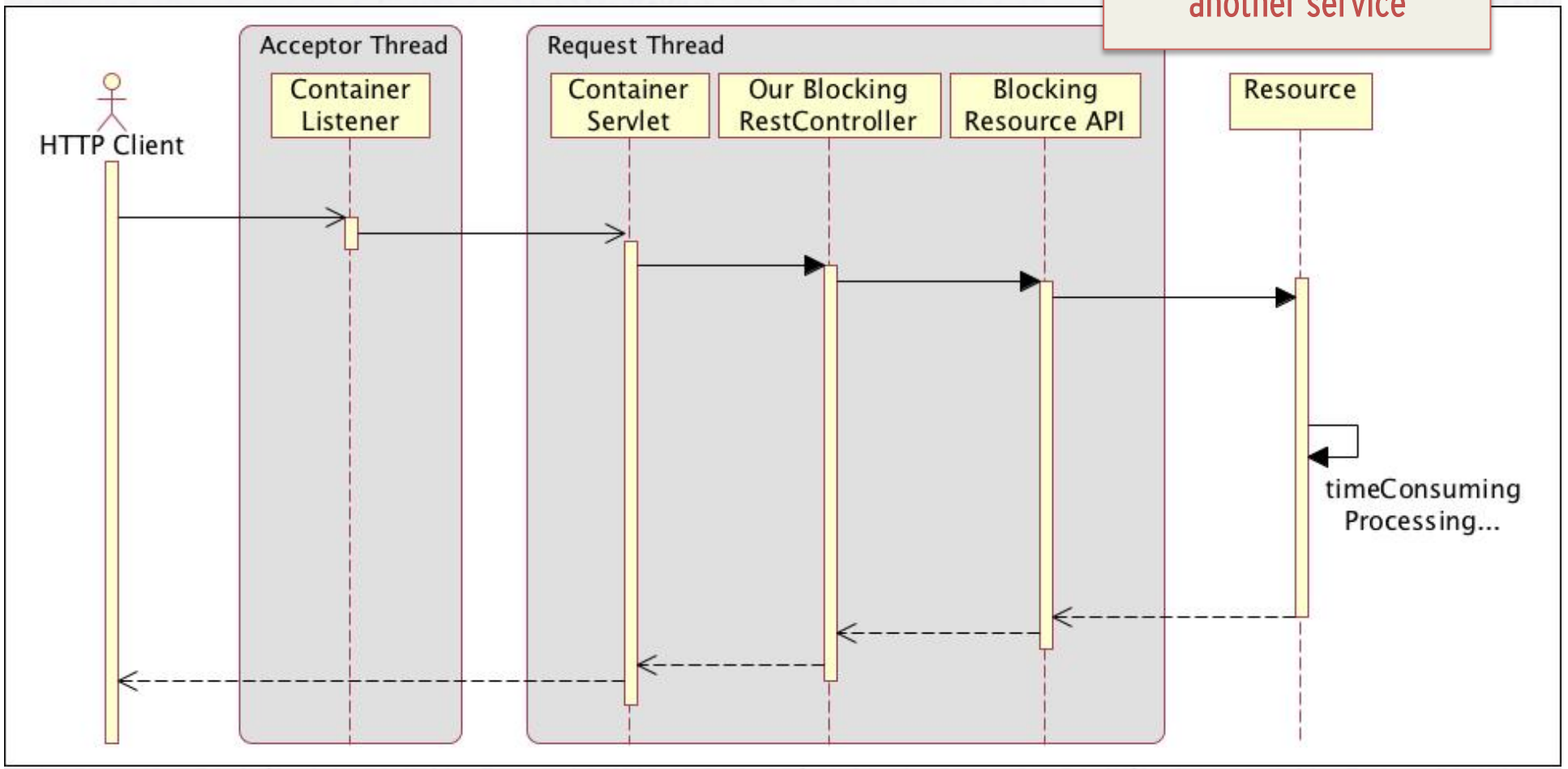
2002	Java SE v1.4	New I/O (NIO) – fundamental support for non blocking I/O in the Java platform
…	Netty, Jetty, …	Early adopting web server and frameworks improved the NIO support
2006	Akka, …	Application frameworks that use specific non blocking web servers & frameworks, e.g. Netty and Jetty
2009	Servlet 3.0 specification	**A portable specification for non blocking I/O based HTTP – services!**
2010	NING asynch-http-client	A non blocking I/O HTTP - client
2012	Spring MVC 3.2	Spring support for non blocking I/O based HTTP – services. **Based on Servlet 3.0 but much easier to use!!!**
2013	Servlet 3.1 specification	Enhanced non blocking I/O based HTTP – services
2013	Spring Framework 4.0	Includes Spring MVC as a core part of the Framework

TRADITIONAL BLOCKING I/O



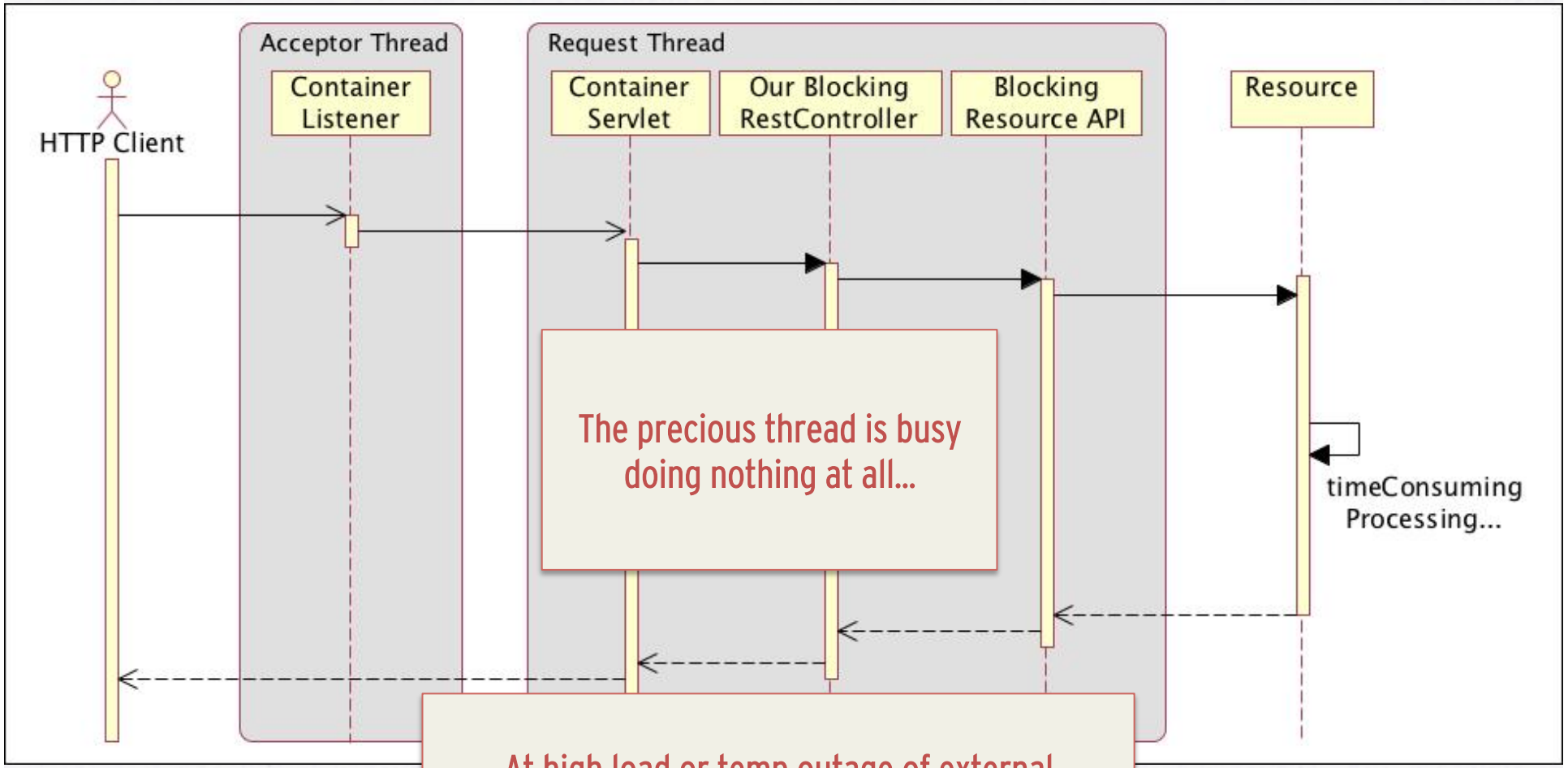
TRADITIONAL BLOCKING I/O

Slow external resource, e.g. a database or another service

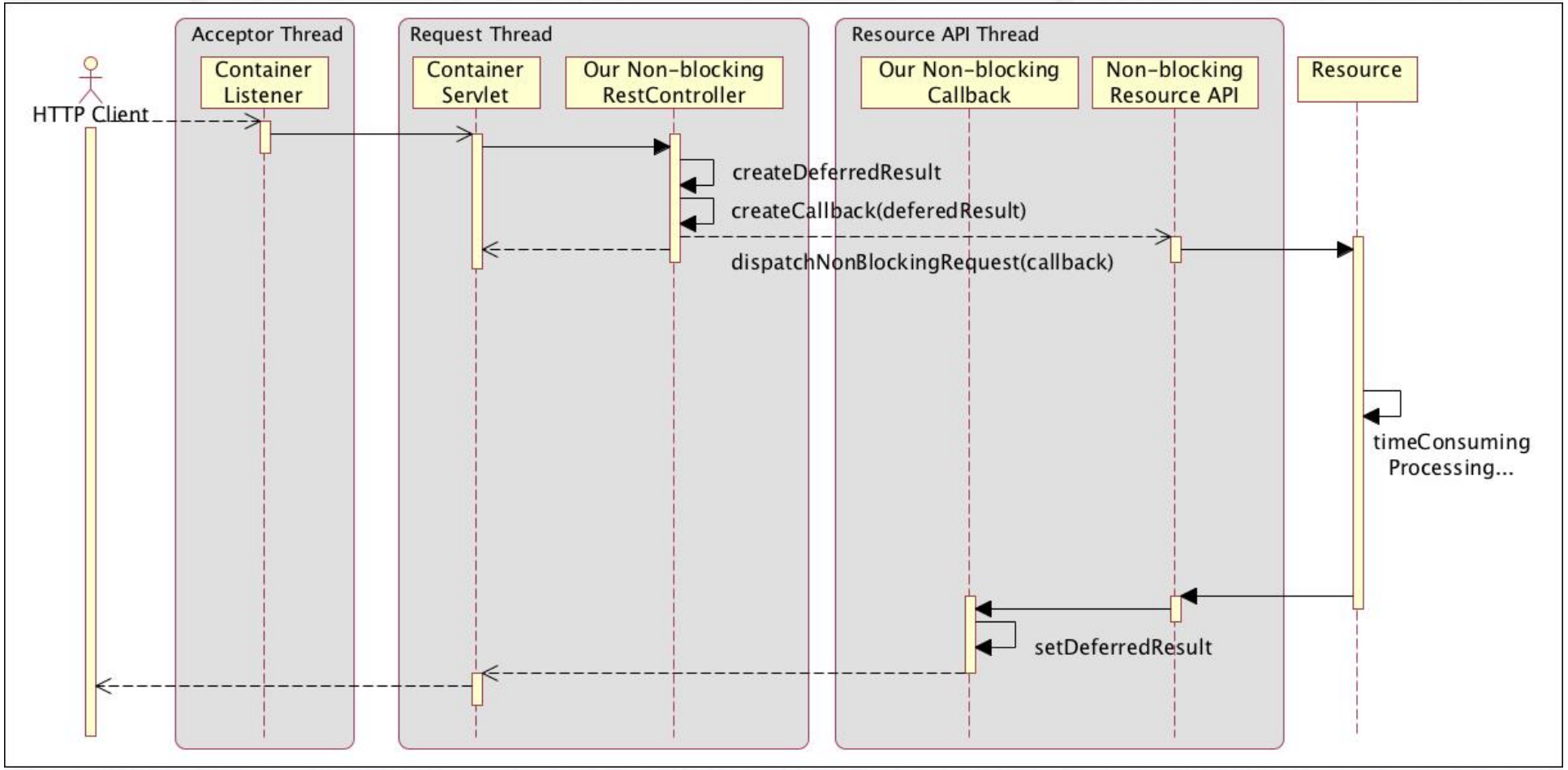


CALLISTA
– ENTERPRISE –

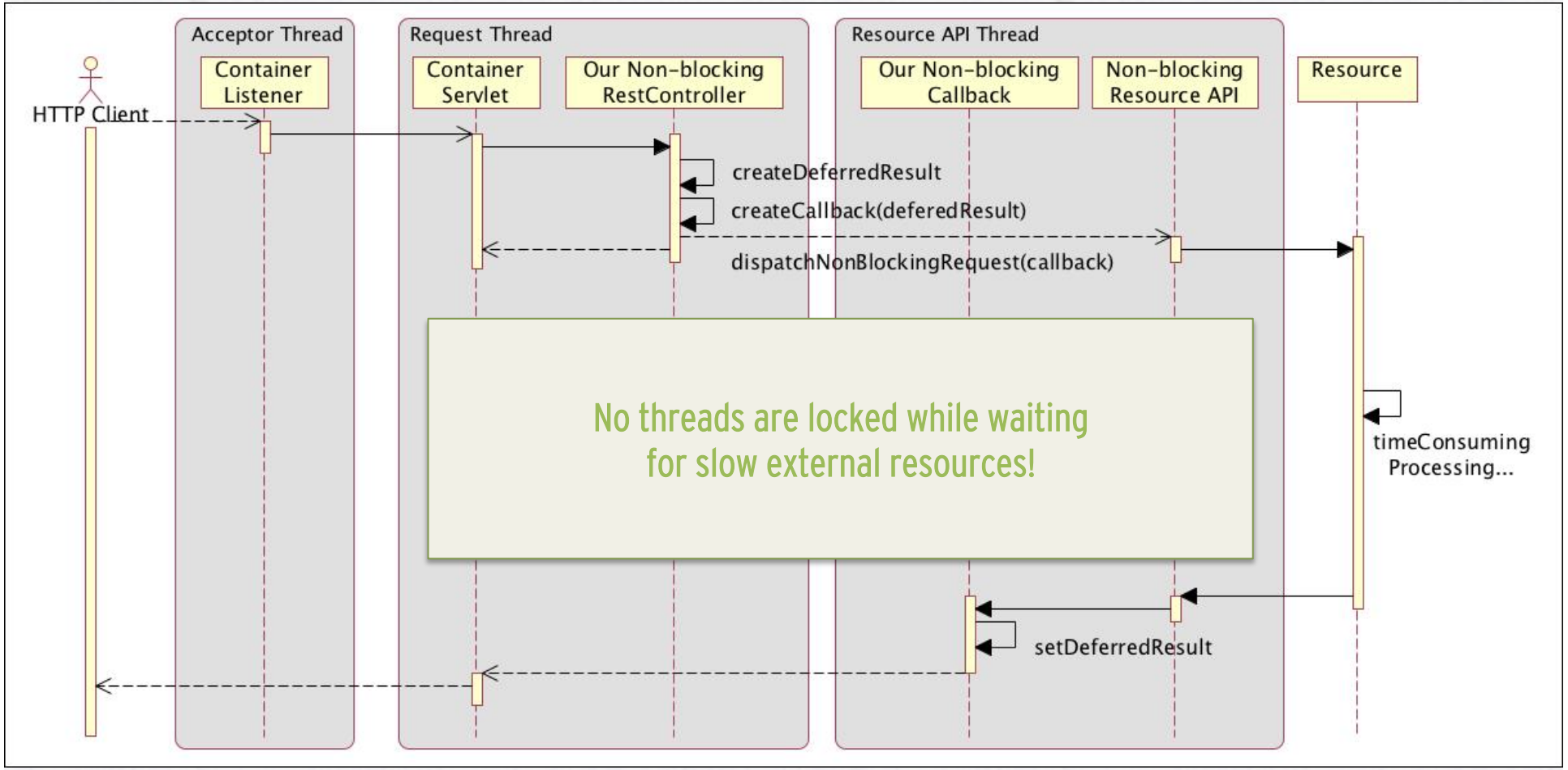
TRADITIONAL BLOCKING I/O



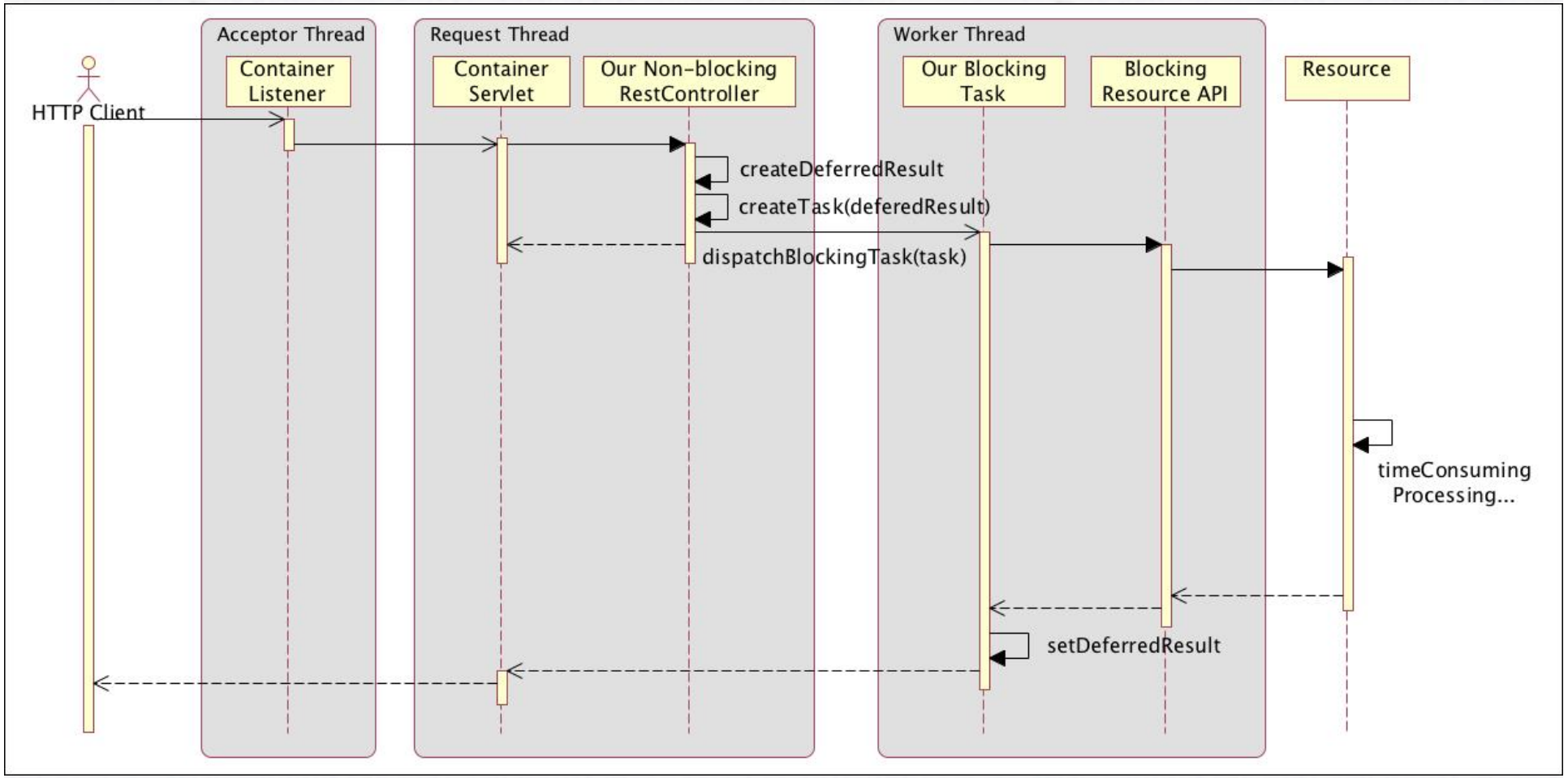
NON-BLOCKING I/O



NON-BLOCKING I/O

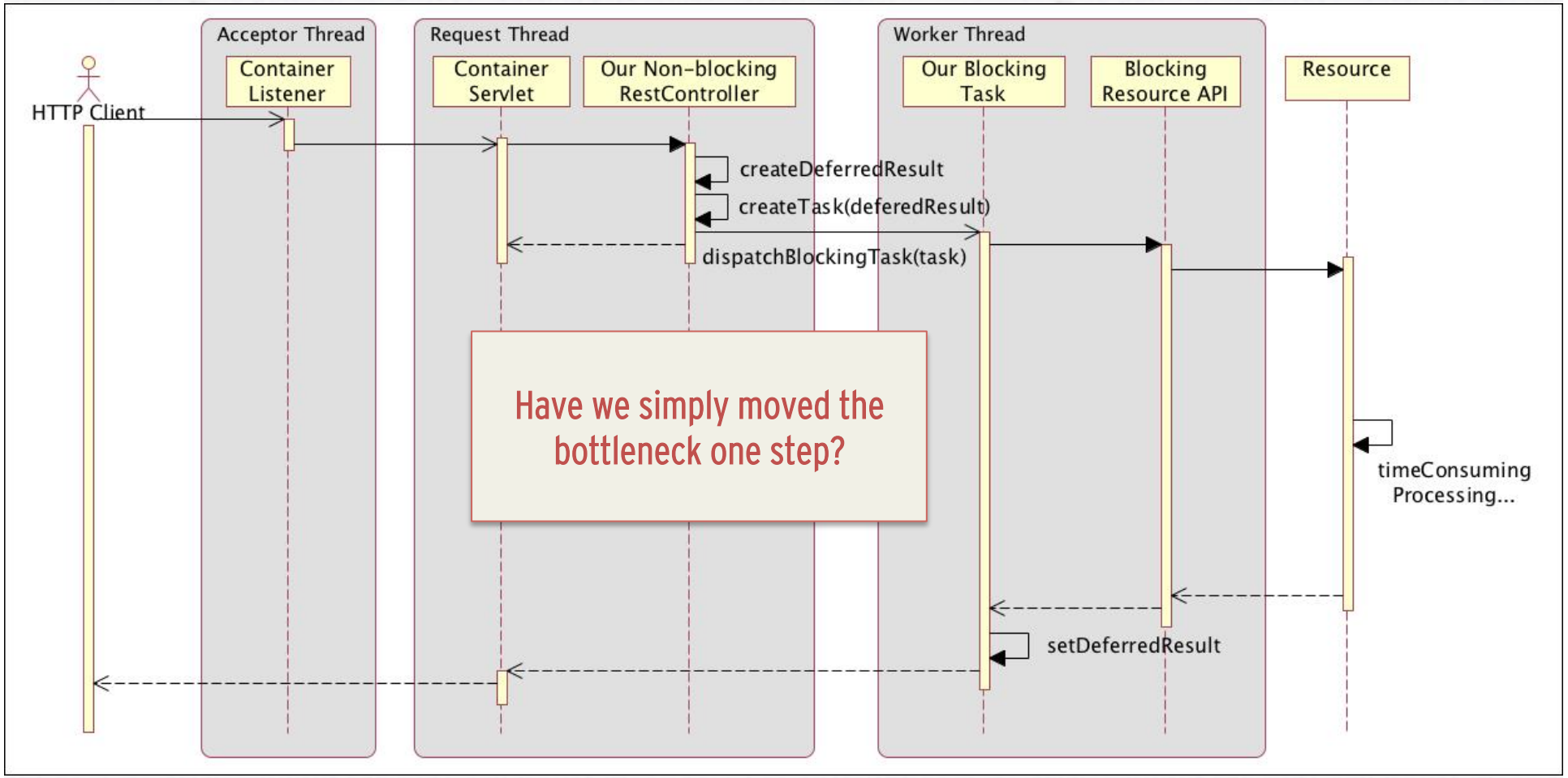


WHAT ABOUT BLOCKING RESOURCE API'S??? (E.G. JDBC)



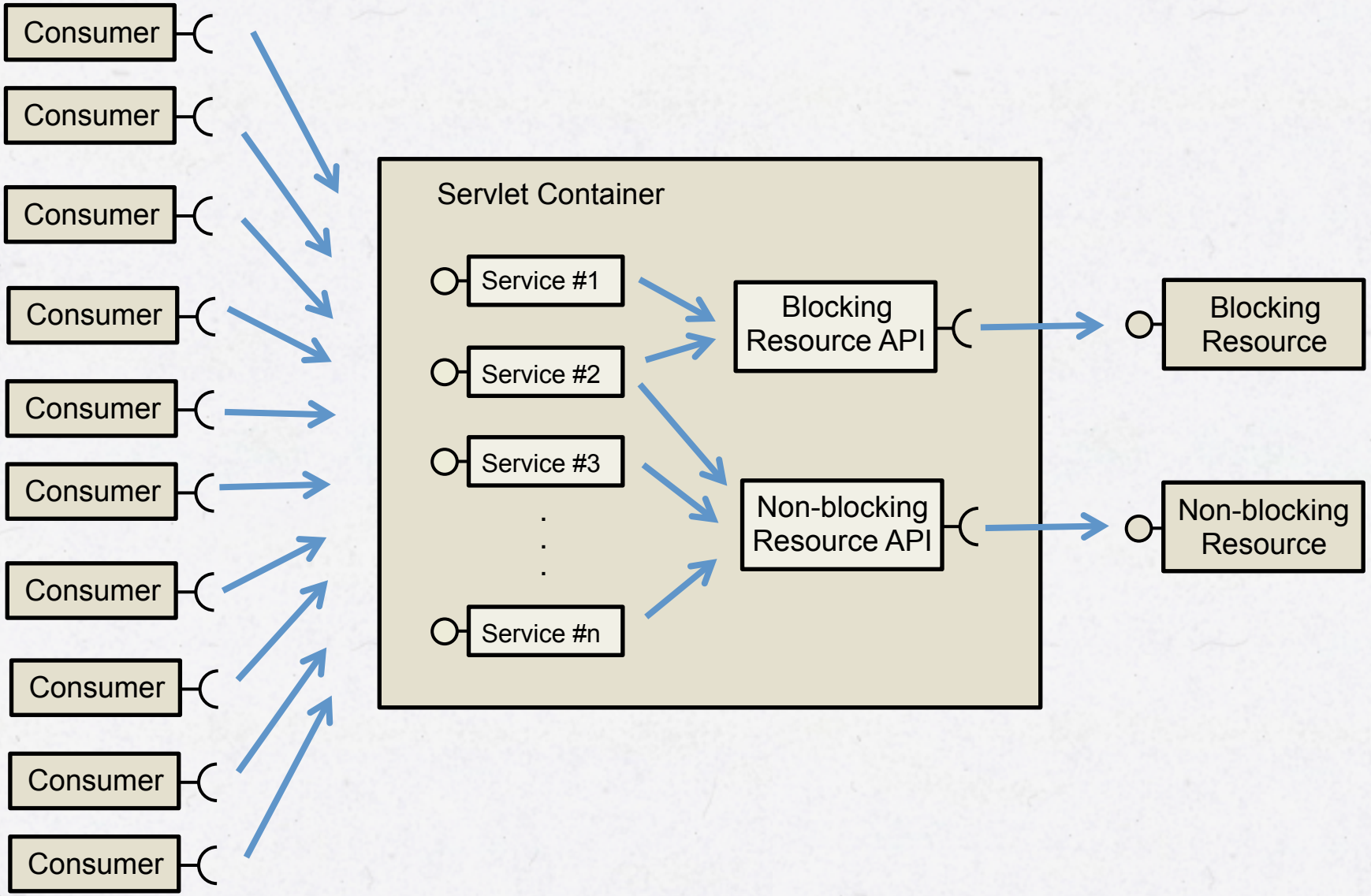
CALLISTA
— ENTERPRISE —

BLOCKING RESOURCE API'S, E.G. JDBC

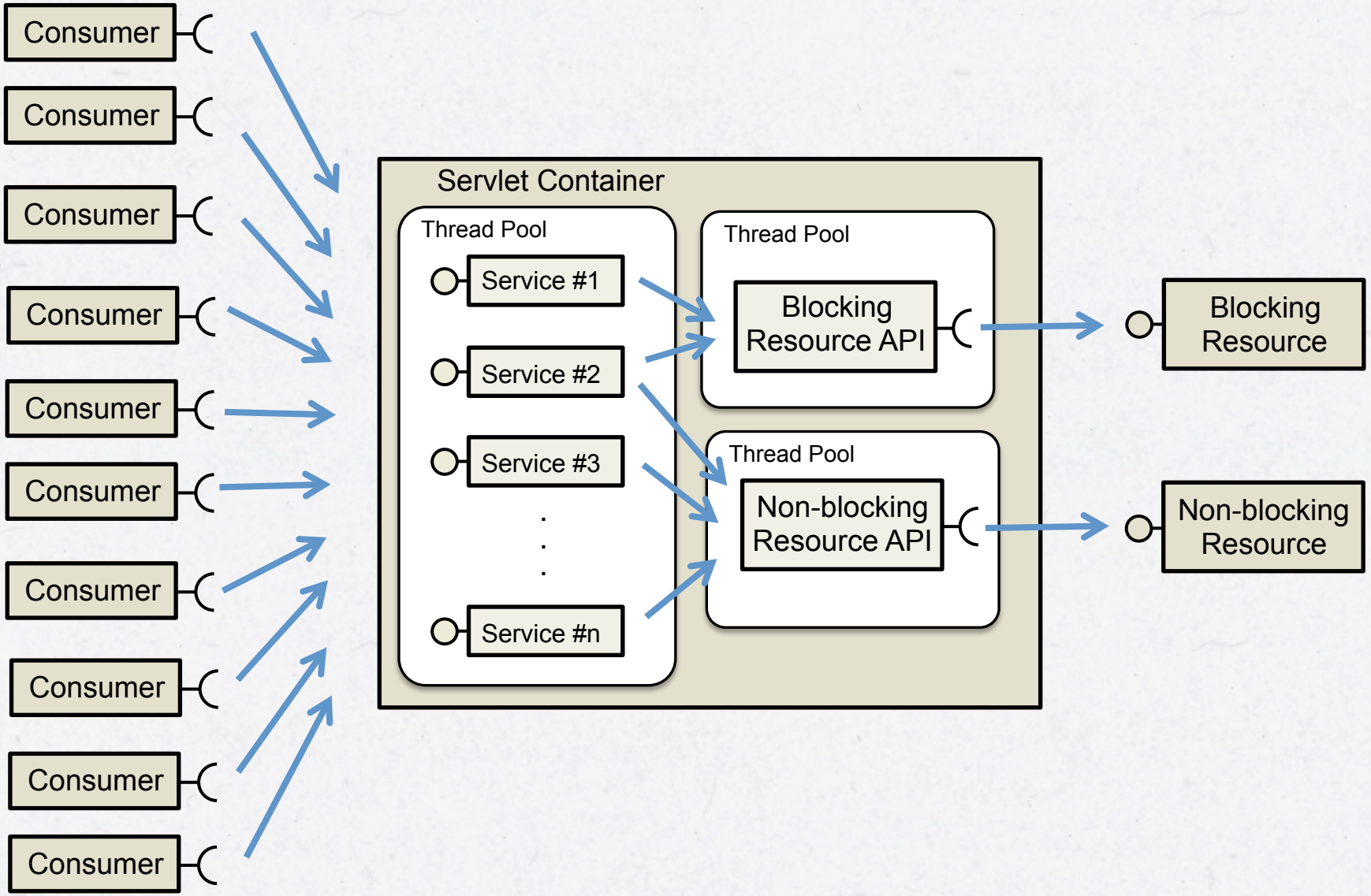


CALLISTA
— ENTERPRISE —

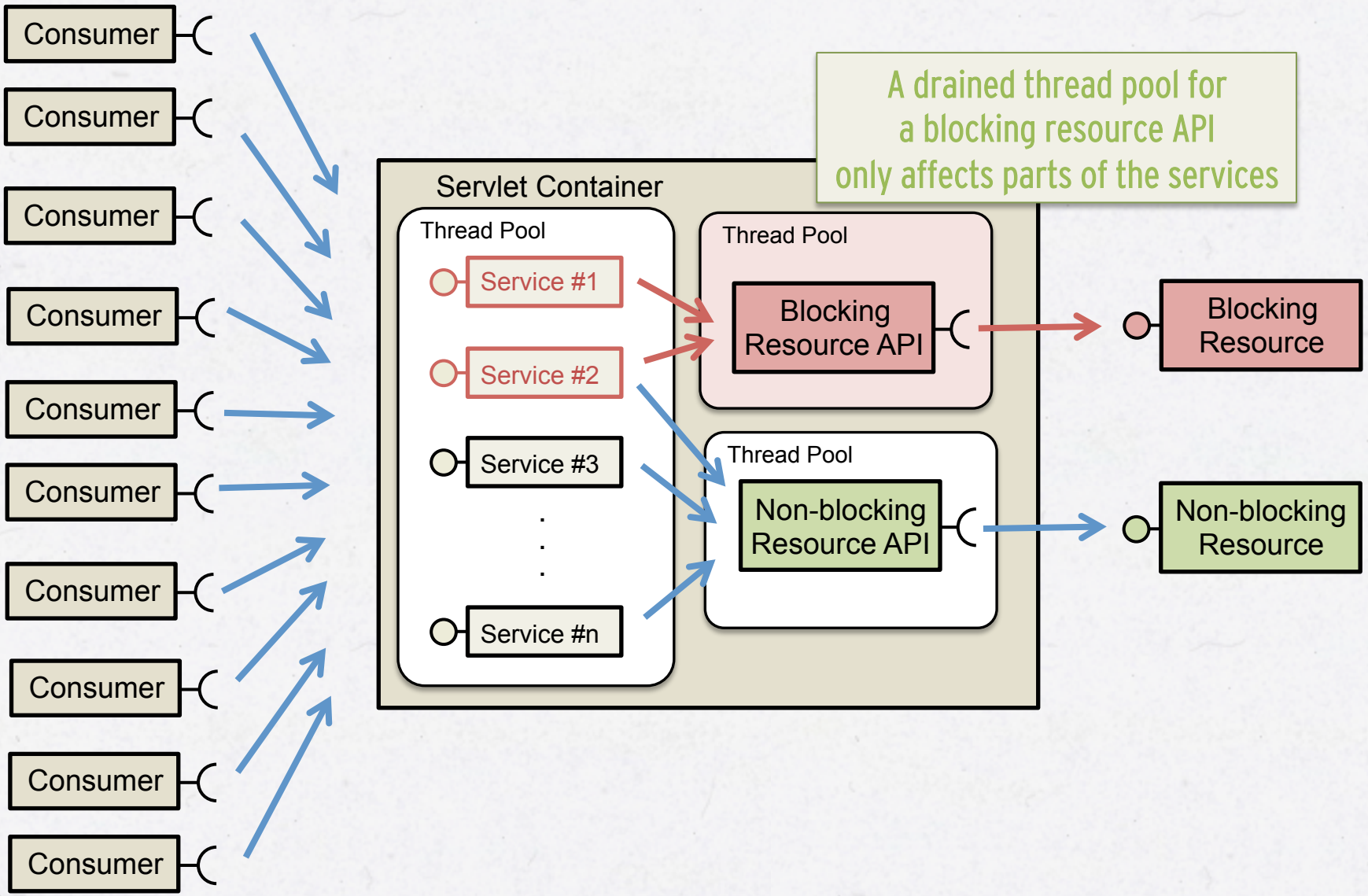
IN A MIXED ENVIRONMENT PROBLEMS CAN BE MITIGATED!



IN A MIXED ENVIRONMENT PROBLEMS CAN BE MITIGATED!



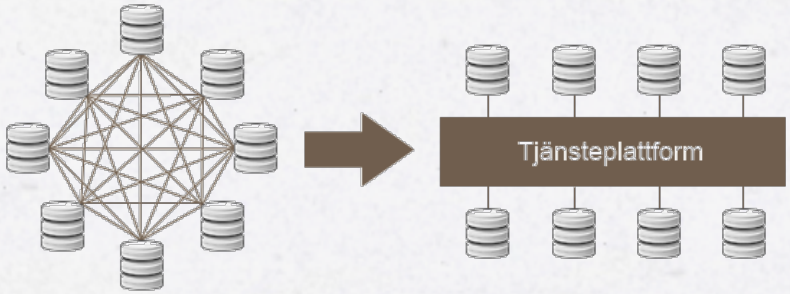
IN A MIXED ENVIRONMENT PROBLEMS CAN BE MITIGATED



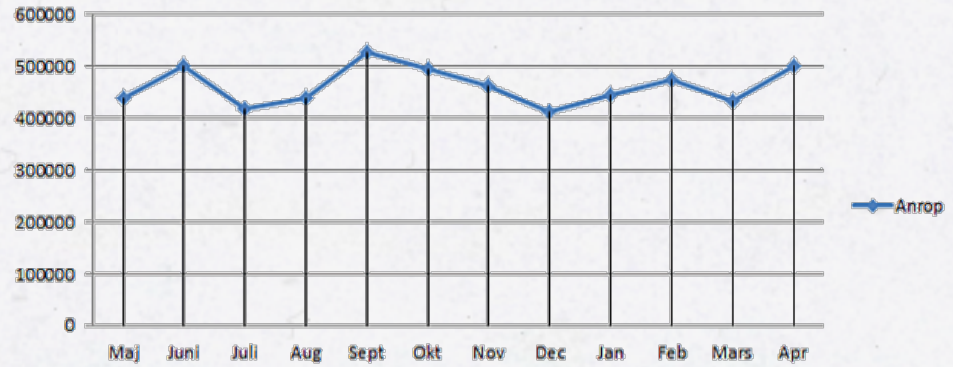
DEMONSTRATION

AN EXAMPLE OF POTENTIAL PROBLEMS WITH BLOCKING I/O

National Healthcare Service Platform

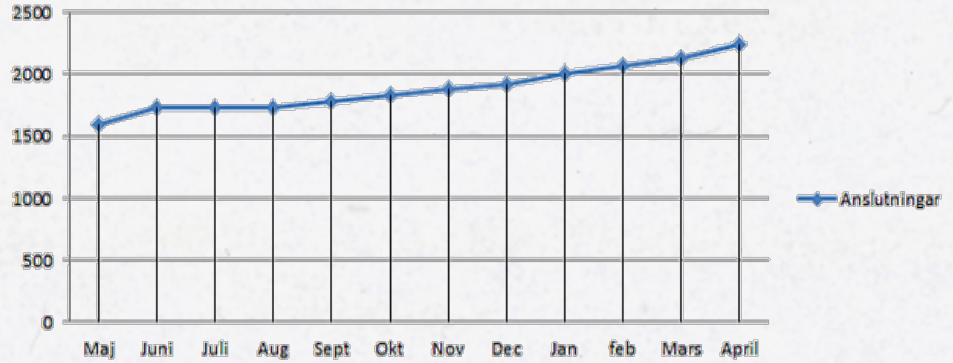


- National reference architecture
- Standardized protocols
- Standardized message formats
- Service catalog for routing
- In operation since 2010
  - > 2000 connected care units
  - > 500 000 messages/day (8h)

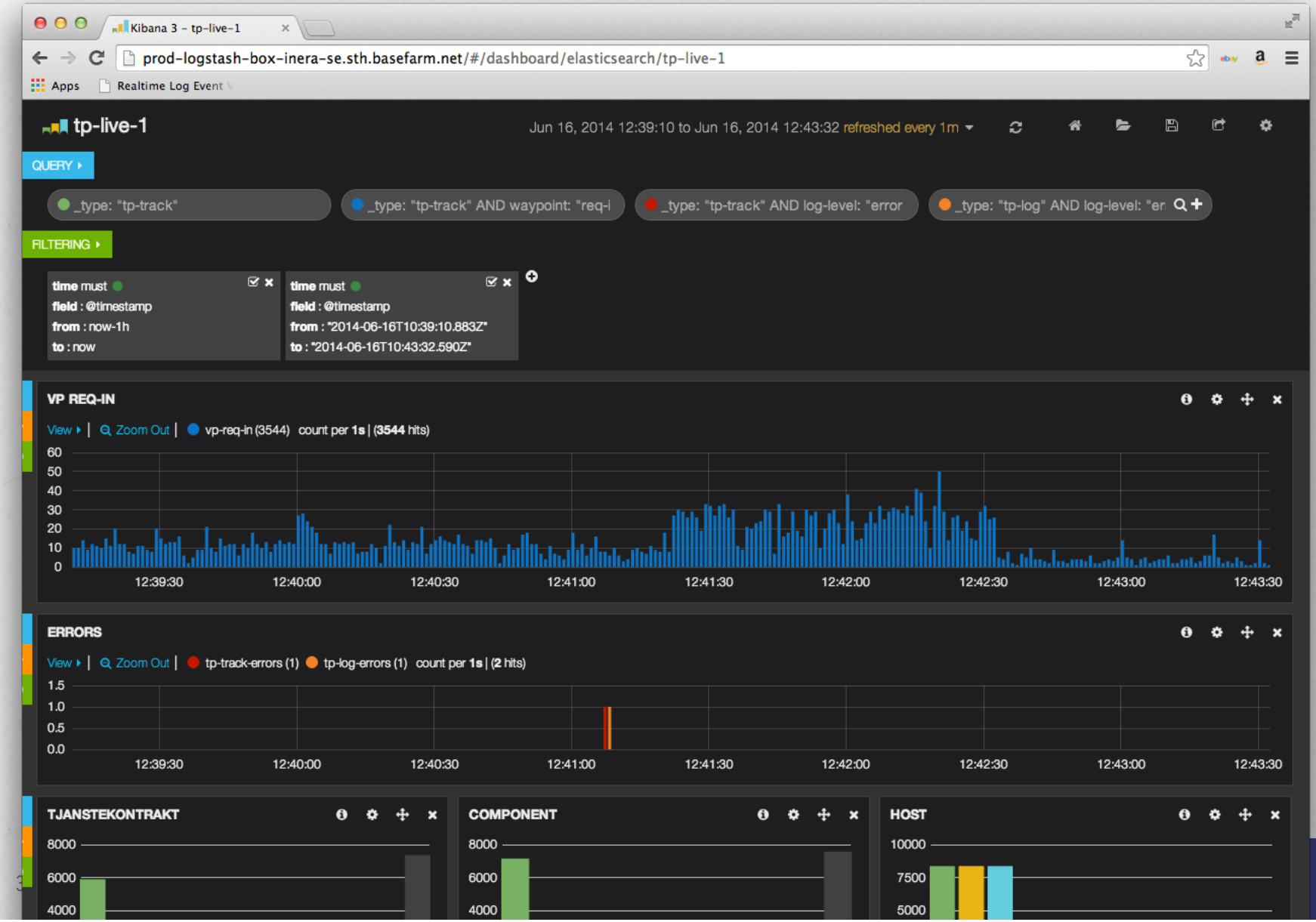


Totalt antal verksamheter anslutna till domäner

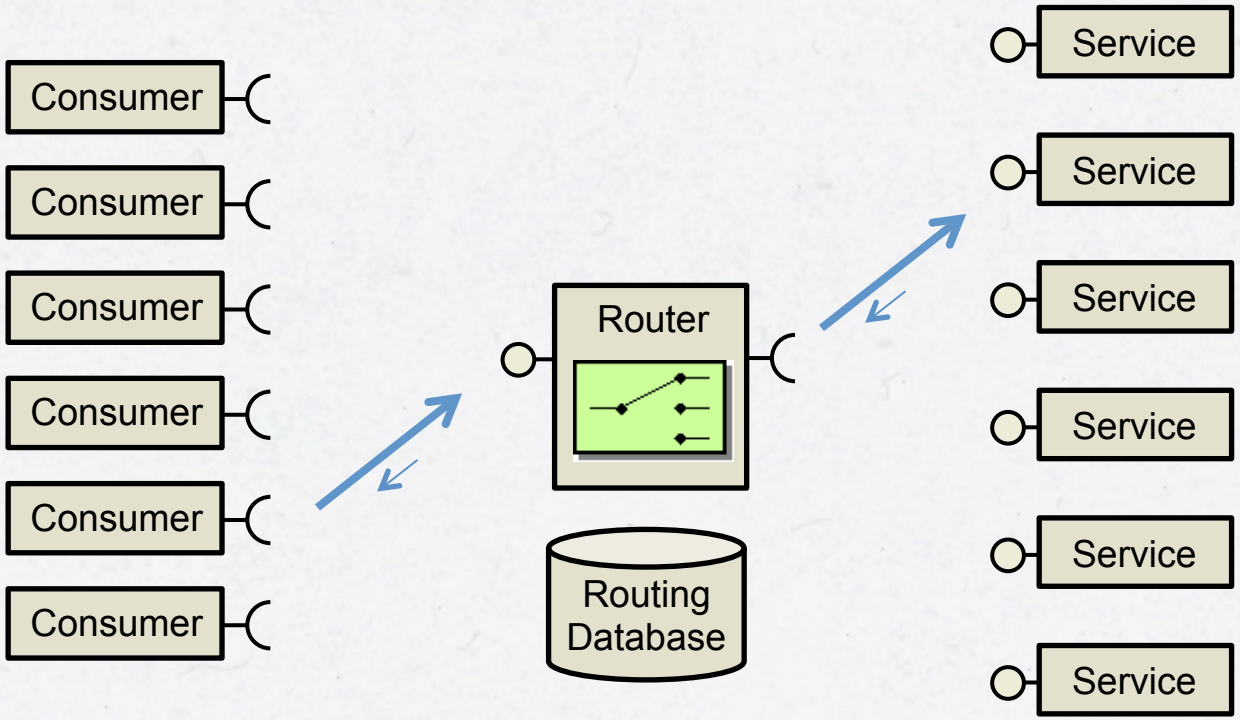
I diagrammet nedan kan du följa utvecklingen av hur många verksamheter som anslutit till Tjänsteplattformen.



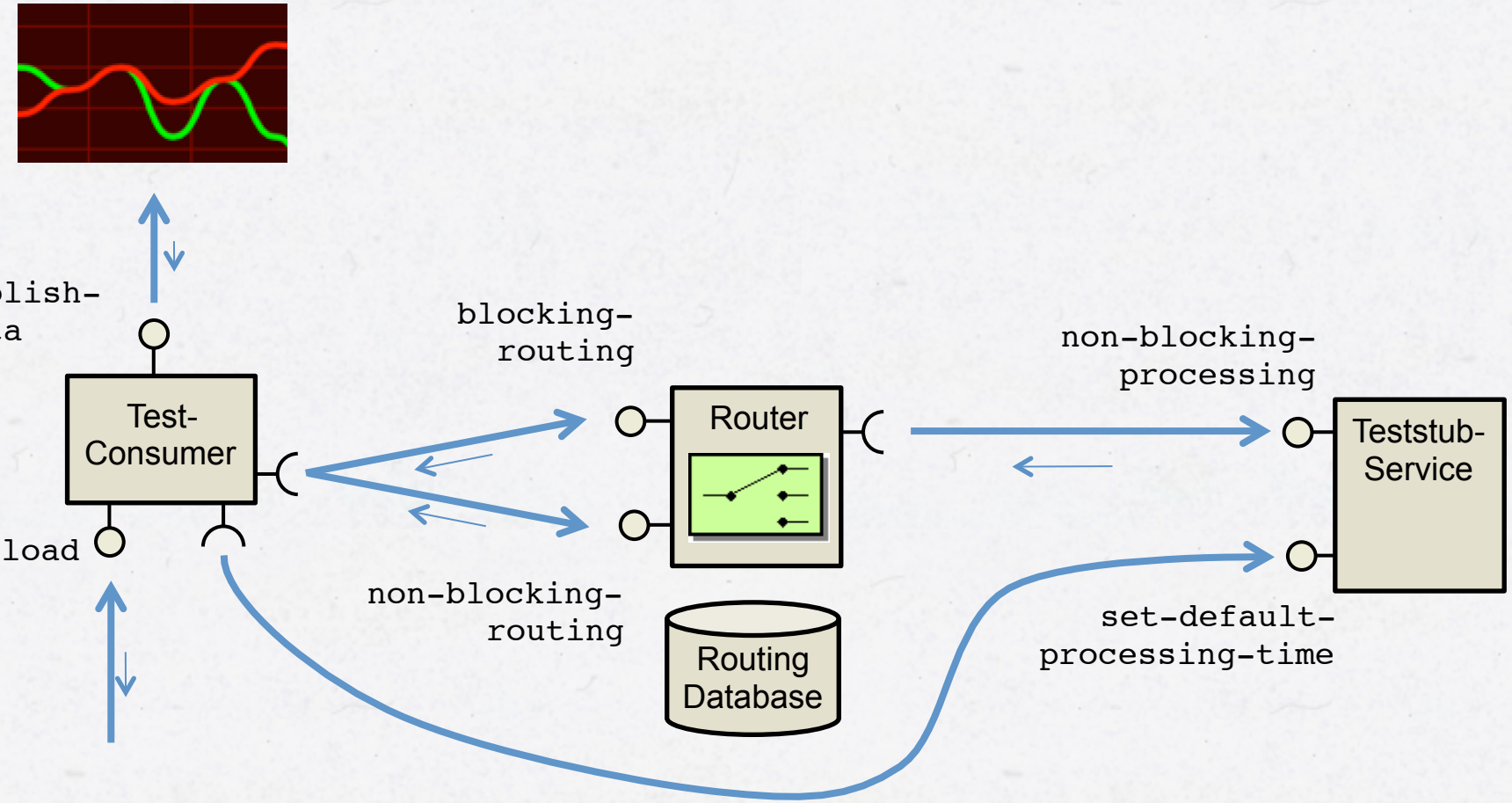
VIEW FROM THE RUNNING SYSTEM IN PRODUCTION



HIGH LEVEL ARCHITECTURE...

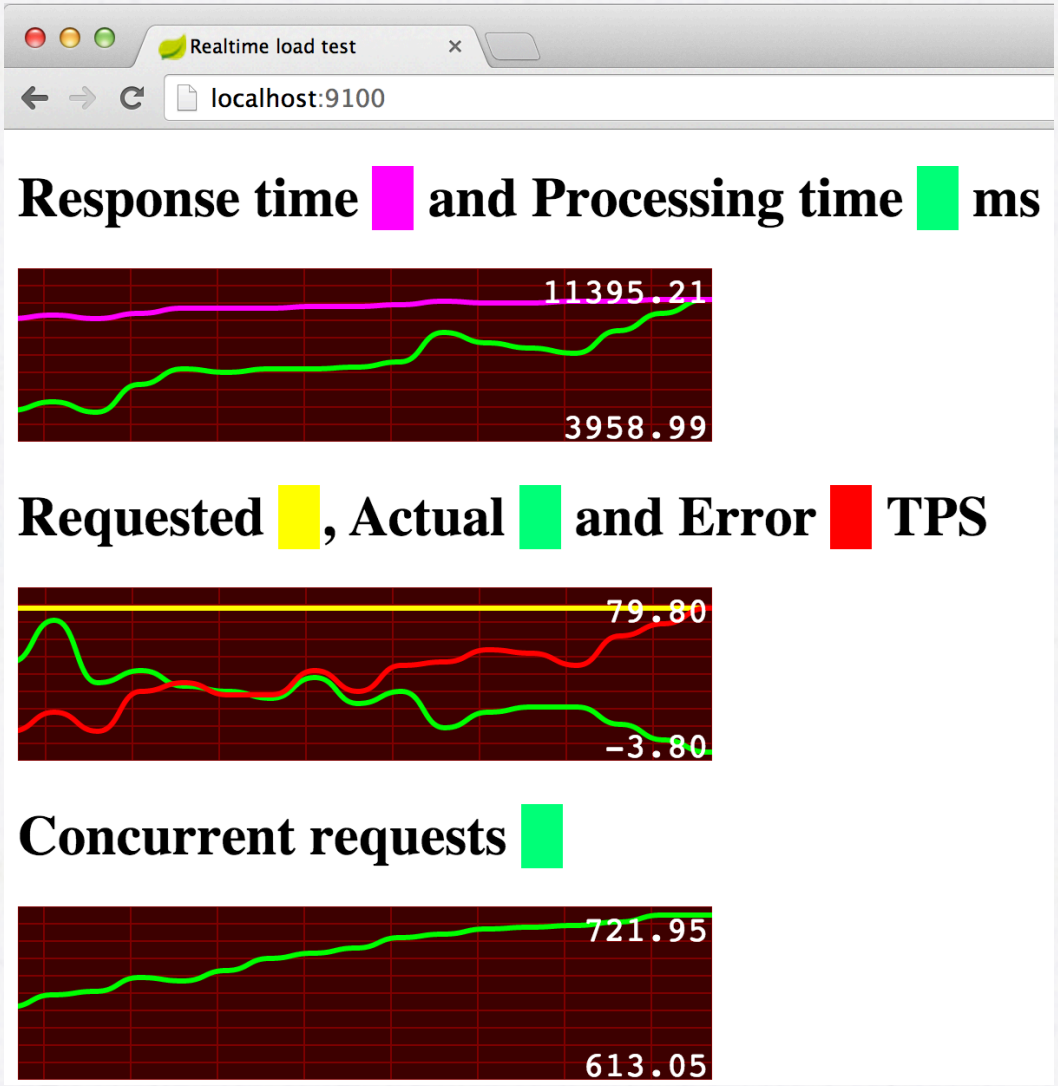


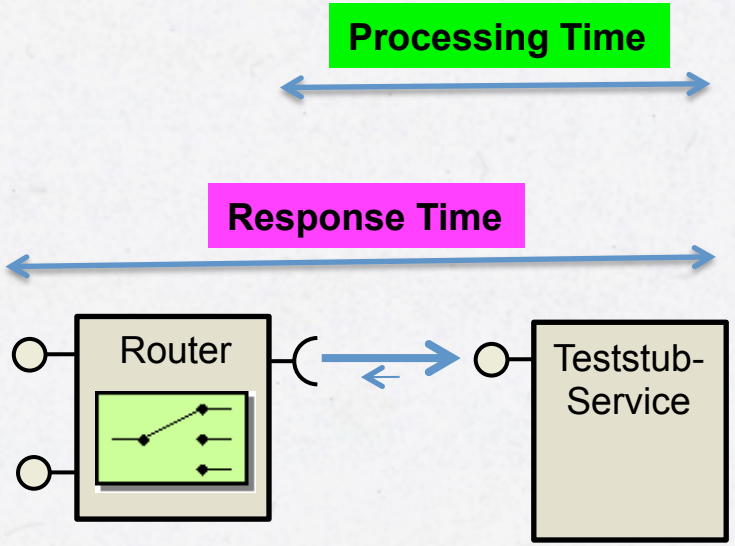
SIMULATION OF THE ENVIRONMENT




```
$ curl "http://localhost:9100/load?
  port=9080&
  uri=router-blocking&
  minMs=3000&
  maxMs=6000&
  tps=50"
```

SAMPLE OUTPUT FROM A LOAD TEST





CALLISTA
– ENTERPRISE –

DEMO

- Normal load is
  - 20 – 50 reqs/s
  - Service Provider response times: 3-6 s
  - Default request timeout: 10 s

- Start with 20 reqs/s
- Step up to 50 reqs/s
- If ok
  - Add a increase of load, 65 reqs/s
  - Add a minor problem, increase response times by 1s
  - What happens?
  - Why?

- Switch to non blocking I/O and **go unleashed**!!!

THE DETAILS…

SERVLET 3.X

- Servlet 3.0, JSR 315, dec 2009, Java EE 6
  - Async Request Handling
  - Threads allocated to connection when needed
  - AsyncContext
  - Requires servlets and filters to declare that they are async enabled
  - **A portable API for non blocking I/O based HTTP services!**
    - » **Web apps can be moved between e.g. Tomcat and Jetty**
- Servlet 3.1, JSR 340, apr 2013, Java EE 7
  - Async I/O
  - Allow writing and reading large responses and requests without blocking

EXAMPLE SERVLET 3.0

```
@WebServlet(name="...", urlPatterns={"/..."}, asyncSupported=true)
public class MyServlet extends HttpServlet {

    ExecutorService executorService = ...;

    public void doGet(HttpServletRequest request, HttpServletResponse
response) {
        AsyncContext aCtx = request.startAsync(request, response);

        executorService.submit(() -> {
            doSomeLongRunningWork(aCtx);
            aCtx.dispatch("/WEB-INF/jsp/page.jsp");
        });
    }
}
```

Enable async

Start an async context

Dispatch work on a separat thread

Dispatch back to servlet container and complete async context

INTRO TO SPRING MVC

- Simplifies development of HTTP services
- Annotation driven
- Inversion of Control
- Takes care of all the mess details with good default values
  - Convention over Configuration
  - Any default behavior can be overridden, e.g. using an annotation
  - E.g. marshal/unmarshal to json or xml, handling of http headers…
- Plays nicely with Spring Boot

SPRING MVC - CODE EXAMPLES

1. Blocking vs non blocking services
2. Asynch call to a non blocking resource
   a. Callback vs anonymous inner class vs Java 8 Lambda expressions
3. Asynch call to a blocking resource
4. Patterns
   a. Router, Aggregator and Routing Slip
5. Routing slip - example of "the callback hell"

- All examples comes from the git-repo in the blog:
  - http://callistaenterprise.se/blogg/teknik/2014/04/22/c10k-developing-non-blocking-rest-services-with-spring-mvc/
  - Look in the master branch

BLOCKING I/O

```
@RestController
public class MyController {

  @RequestMapping("/block")
  public R block(...) {
    ...
    return new R(...);
  }
}
```

1. SPRING MVC -BLOCKING I/O VS. NON BLOCKING I/O

BLOCKING I/O

```
@RestController
public class MyController {

  @RequestMapping("/block")
  public R block(...) {
    ...
    return new R(...);
  }
}
```

NON BLOCKING I/O

```
@RestController
public class ProcessingController {

  @RequestMapping("/non-block")
  public DeferredResult<R> nonBlock(...) {

    DeferredResult<R> dr =
      new DeferredResult<>();
    dispatch(new MyTask(dr, ...));
    return dr;
  }
}
```

```
public class MyTask extends MyCallback {

  private DeferredResult<R> deferredResult;
  public MyTask(DeferredResult<R> dr, ...) {
    this.df = df;
  }
  public void done() {
    df.setResult(new R(...));
  }
}
```

1. SPRING MVC -BLOCKING I/O VS. NON BLOCKING I/O

BLOCKING I/O

```
@RestController
public class MyController {

  @RequestMapping("/block")
  public R block(...) {
    ...
    return new R(...);
  }
}
```

DeferredResult **IS THE KEY SPRING MVC ABSTRACTION!**

NON BLOCKING I/O

```
@RestController
public class ProcessingController {

  @RequestMapping("/non-block")
  public DeferredResult<R> nonBlock(...) {

    DeferredResult<R> dr =
      new DeferredResult<>();
    dispatch(new MyTask(dr, ...));
    return dr;
  }
}
```

CALLBACK MODEL

```
public class MyTask implements MyCallback {

  private DeferredResult<R> deferredResult;
  public MyTask(DeferredResult<R> dr, ...) {
    this.df = df;
  }
  public void done() {
    df.setResult(new R(...));
  }
}
```

2.A ASYNCH CALL TO A NON BLOCKING RESOURCE

- Non Blocking I/O HTTP calls with Ning async-http-client

```
private static final AsyncHttpClient asyncHttpClient = new AsyncHttpClient();

@RequestMapping("/router-non-blocking-callback")
public DeferredResult<String> nonBlockingRouter(...) {

  DeferredResult<String> dr = new DeferredResult<String>();
  asyncHttpClient.prepareGet(getUrl(...)).execute(new MyCallback(dr));
  return dr;
}
```

```
public class MyCallback extends AsyncCompletionHandler<Response> {

  private DeferredResult<String> dr;
  public MyCallback(DeferredResult<String> dr) {
    this.dr = dr;
  }

  public Response onCompleted(Response response) {
    dr.setResult(response.getResponseBody());
  }

  public void onThrowable(Throwable t){...}
}
```

2.B ANONYMOUS INNER CLASS

```
@RequestMapping("/router-non-blocking-anonymous")
public DeferredResult<String> nonBlockingRouter(...) {

  final DeferredResult<String> dr = new DeferredResult<>();

  asyncHttpClient.prepareGet(getUrl(..)).execute(

    new AsyncCompletionHandler<Response>() {

      public Response onCompleted(Response response) {
        dr(response.getResponseBody());
      }

      public void onThrowable(Throwable t){...}
    });

  return dr;
}
```

2.C JAVA 8 AND LAMBDAS

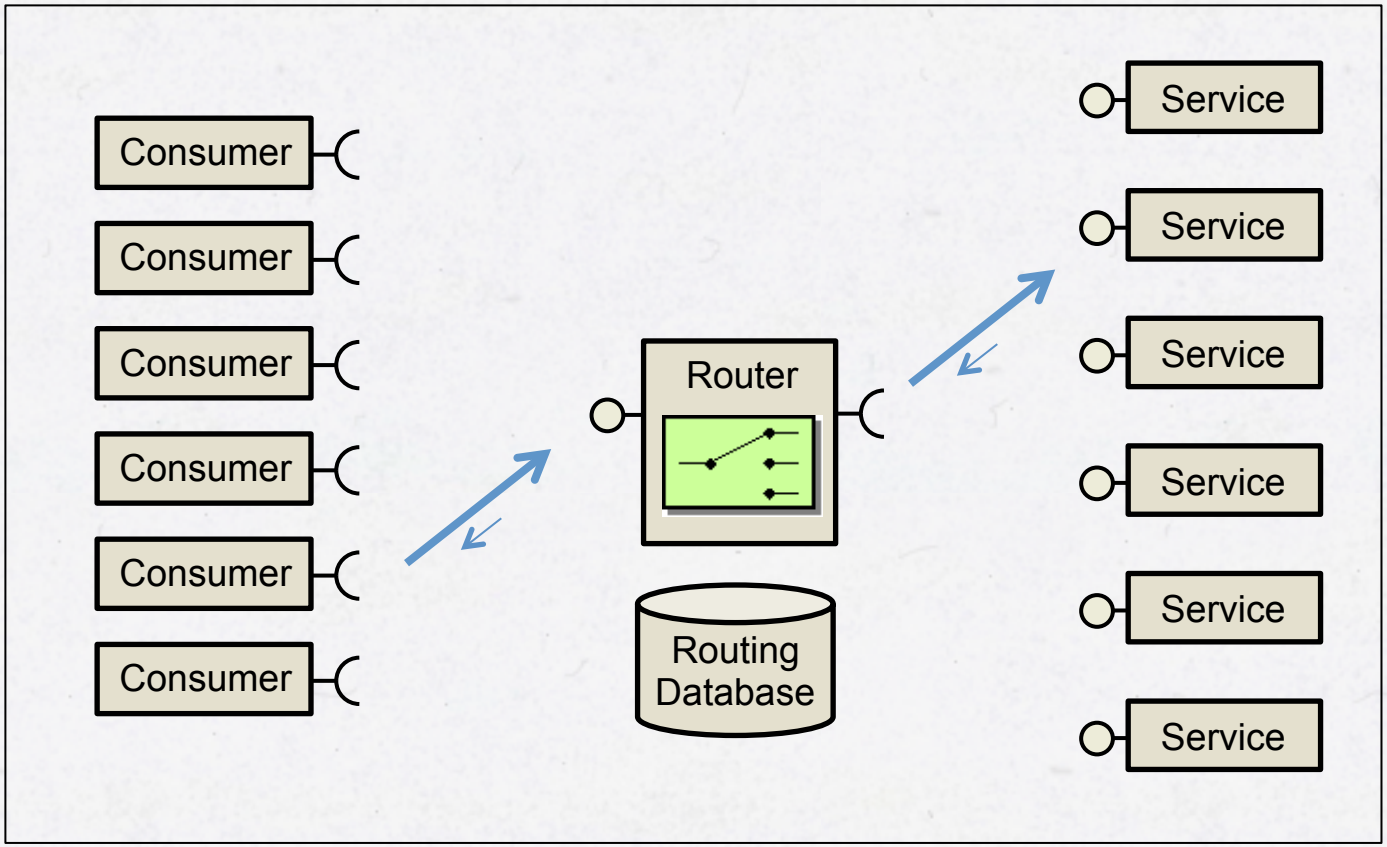
```
@RequestMapping("/router-non-blocking-lambda")
public DeferredResult<String> nonBlockingRouter(...) {

  final DeferredResult<String> dr = new DeferredResult<>();

  asyncHttpClient.execute(getUrl(...),
    (response) ->  {
      dr.setResult(response.getResponseBody());
    },
    (throwable) -> {...}
  );

  return dr;
}
```

3. ASYNCH CALL TO A BLOCKING RESOURCE

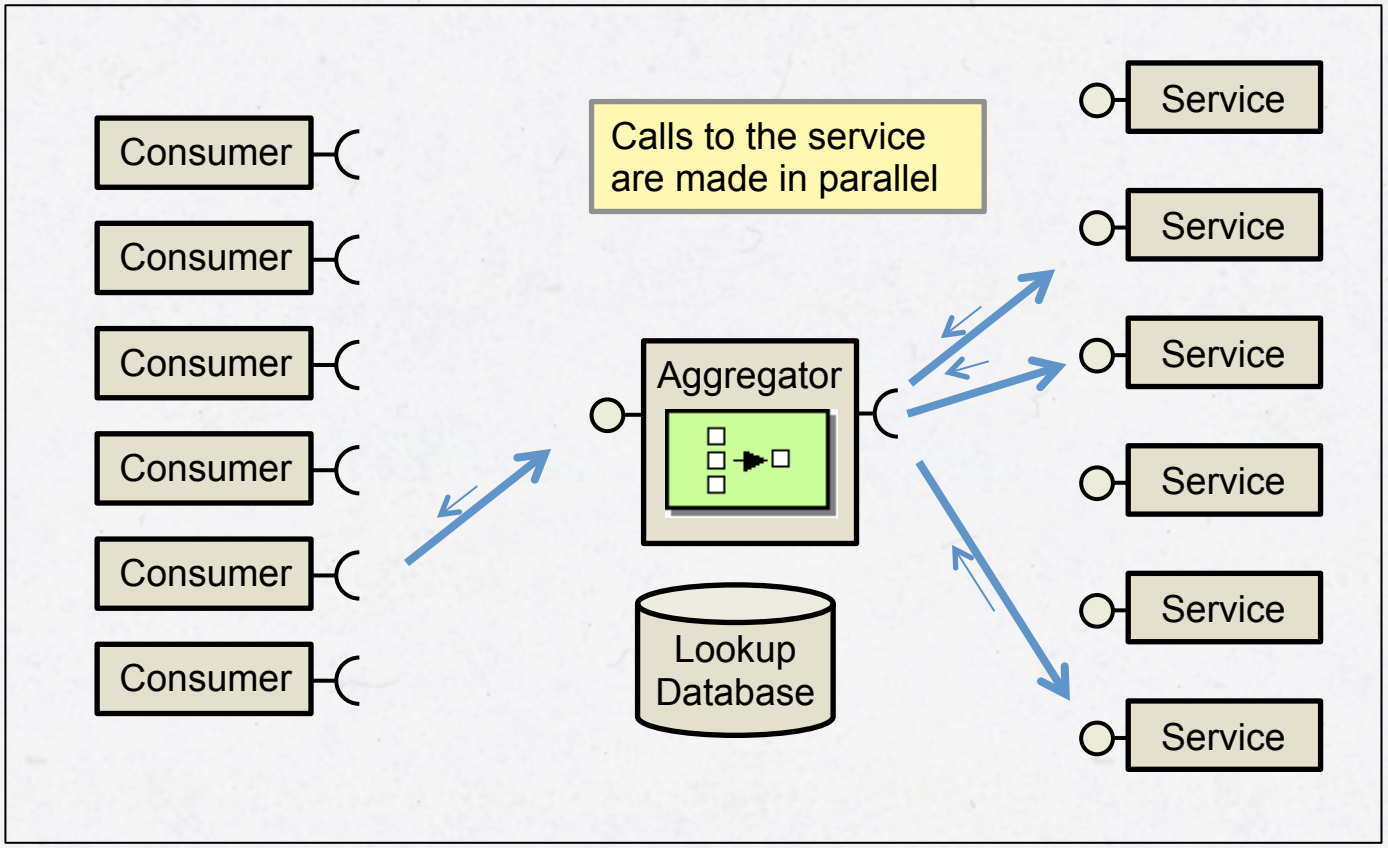
```
@Autowired
@Qualifier("dbThreadPoolExecutor")
private TaskExecutor dbThreadPoolExecutor;

@RequestMapping("/aggregate-non-blocking-callback")
public DeferredResult<String> nonBlockingAggregator(...) {

  DeferredResult<String> dr= new DeferredResult<String>();
  dbThreadPoolExecutor.execute(new DbLookupRunnable(dr, ...));
  return dr;
}
```

```
public class DbLookupRunnable implements Runnable {

  private DeferredResult<String> dr;
  public DbLookupRunnable(DeferredResult<String> dr, ...) {
    this.deferredResult = dr;
  }
  public void run() {
    // Perform blocking database operation
    ...
    dr.setResult(responseFromDatabaseOperation)
  }
}
```

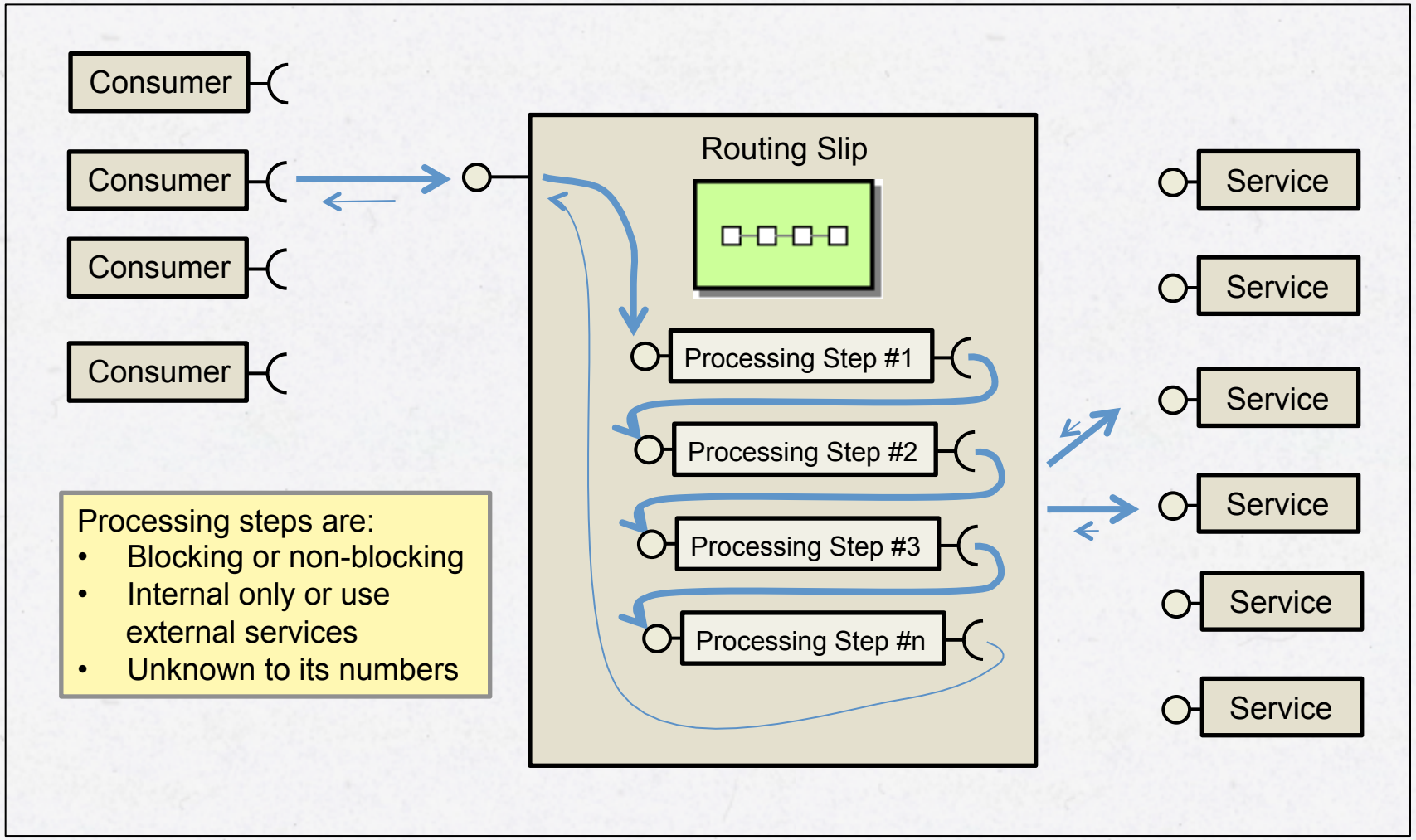
4.A PATTERNS - ROUTER



4.B PATTERNS - AGGREGATOR



4.C PATTERNS - ROUTING SLIP



5. ROUTING SLIP - EXAMPLE OF "THE CALLBACK HELL"

- Perform 5 sequential Non Blocking I/O calls…

```
@RequestMapping("/routing-slip-non-blocking-lambda")
public DeferredResult<String> nonBlockingRoutingSlip() throws IOException {

  final DeferredResult<String> dr = new DeferredResult<>();

  // Send request #1
  ListenableFuture<Response> execute = asyncHttpClient.execute(getUrl(1),
    (Response r1) -> {
      processResult(r1.getResponseBody()); // Process response #1

      // HOW TO SEND REQUEST #2 ???
```

5. ROUTING SLIP - EXAMPLE OF "THE CALLBACK HELL"

- Perform 5 sequential Non Blocking I/O calls…

```
@RequestMapping("/routing-slip-non-blocking-lambda")
public DeferredResult<String> nonBlockingRoutingSlip() throws IOException {

  final DeferredResult<String> dr = new DeferredResult<>();

  // Send request #1
  ListenableFuture<Response> execute = asyncHttpClient.execute(getUrl(1),
    (Response r1) -> {
      processResult(r1.getResponseBody()); // Process response #1
      asyncHttpClient.execute(getUrl(2),   // Send request #2
        (Response r2) -> {
          processResult(r2.getResponseBody()); // Process response #2
          asyncHttpClient.execute(getUrl(3),   // Send request #3
            (Response r3) -> {
              processResult(r3.getResponseBody()); // Process response #3
              asyncHttpClient.execute(getUrl(4),   // Send request #4
                (Response r4) -> {
                  processResult(r4.getResponseBody()); // Process response #4
                  asyncHttpClient.execute(getUrl(5),   // Send request #5
                    (Response r5) -> {
                      processResult(r5.getResponseBody()); // Process response #5
                      // Get the total result and set it on the deferred result
                      dr.setResult(getTotalResult());
                      ...
```

5. ROUTING SLIP - EXAMPLE OF "THE CALLBACK HELL"

- Perform 5 sequential Non Blocking I/O calls…

```
@RequestMapping("/routing-slip-non-blocking-lambda")
public DeferredResult<String> nonBlockingRoutingSlip() throws IOException {

  final DeferredResult<String> dr = new DeferredResult<>();

  // Send request #1
  ListenableFuture<Response> execute = asyncHttpClient.execute(getUrl(1),
    (Response r1) -> {
      processResult(r1.get                                    ss response #1
      asyncHttpClient.exec                                    request #2
        (Response r2) -> {
          processResult(r2                                    rocess response #2
          asyncHttpClient.                                    end request #3
            (Response r3)
              processResult(r3.getResponseBody()); // Process response #3
              asyncHttpClient.execute(getUrl(4),   // Send request #4
                (Response r4) -> {
                  processResult(r4.getResponseBody()); // Process response #4
                  asyncHttpClient.execute(getUrl(5),   // Send request #5
                    (Response r5) -> {
                      processResult(r5.getResponseBody()); // Process response #5
                      // Get the total result and set it on the deferred result
                      dr.setResult(getTotalResult());
                      ...
```

This is not OK!

5. ROUTING SLIP - EXAMPLE OF "THE CALLBACK HELL"

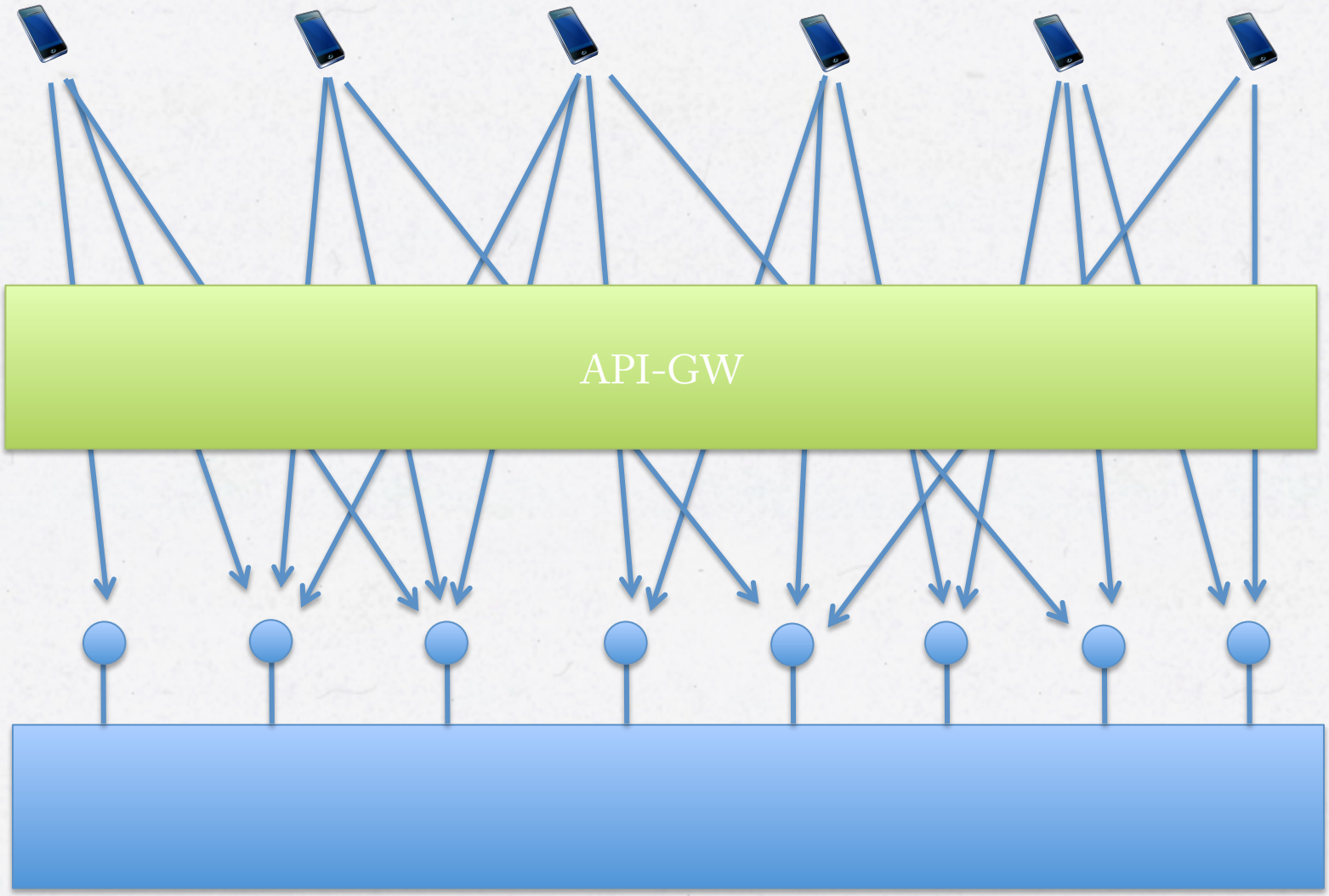
- Wrap up with a long list of nested returns…

```
                        dr.setResult(getTotalResult());
                        return r5;
                      });
                    return r4;
                  });
                return r3;
              });
            return r2;
          });
        return r1;
      });
  return deferredResult;
}
```

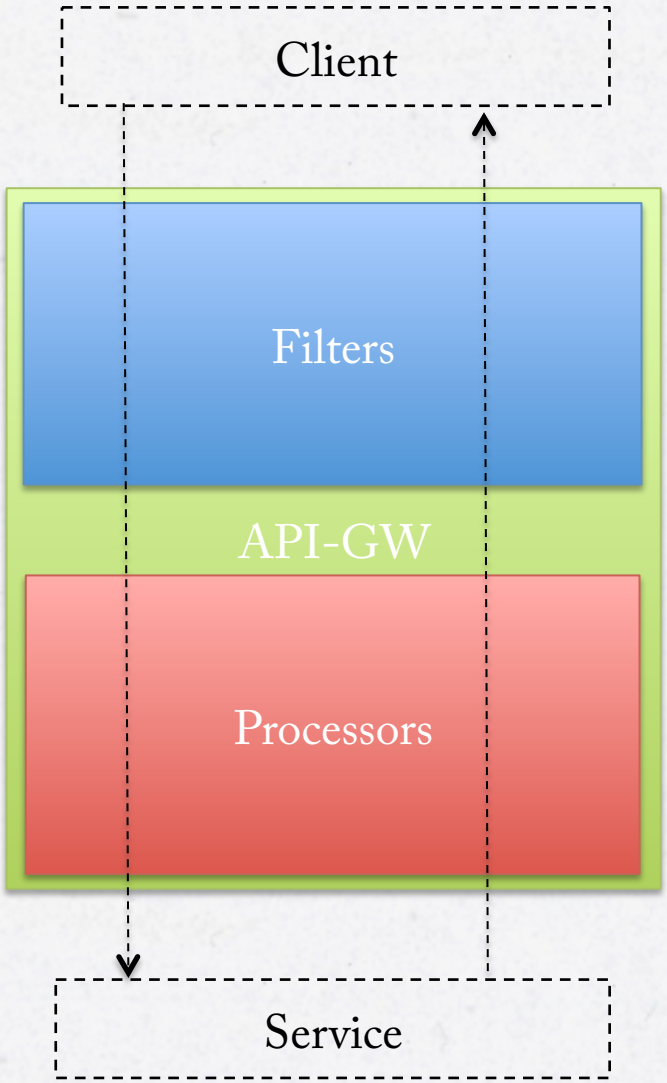
- This is a very simple example of composite non blocking I/O services, the "callback hell" can get much much worse!!!

Experiences from a real life projects

EXPERIENCES FROM REAL LIFE PROJECTS - API-GW



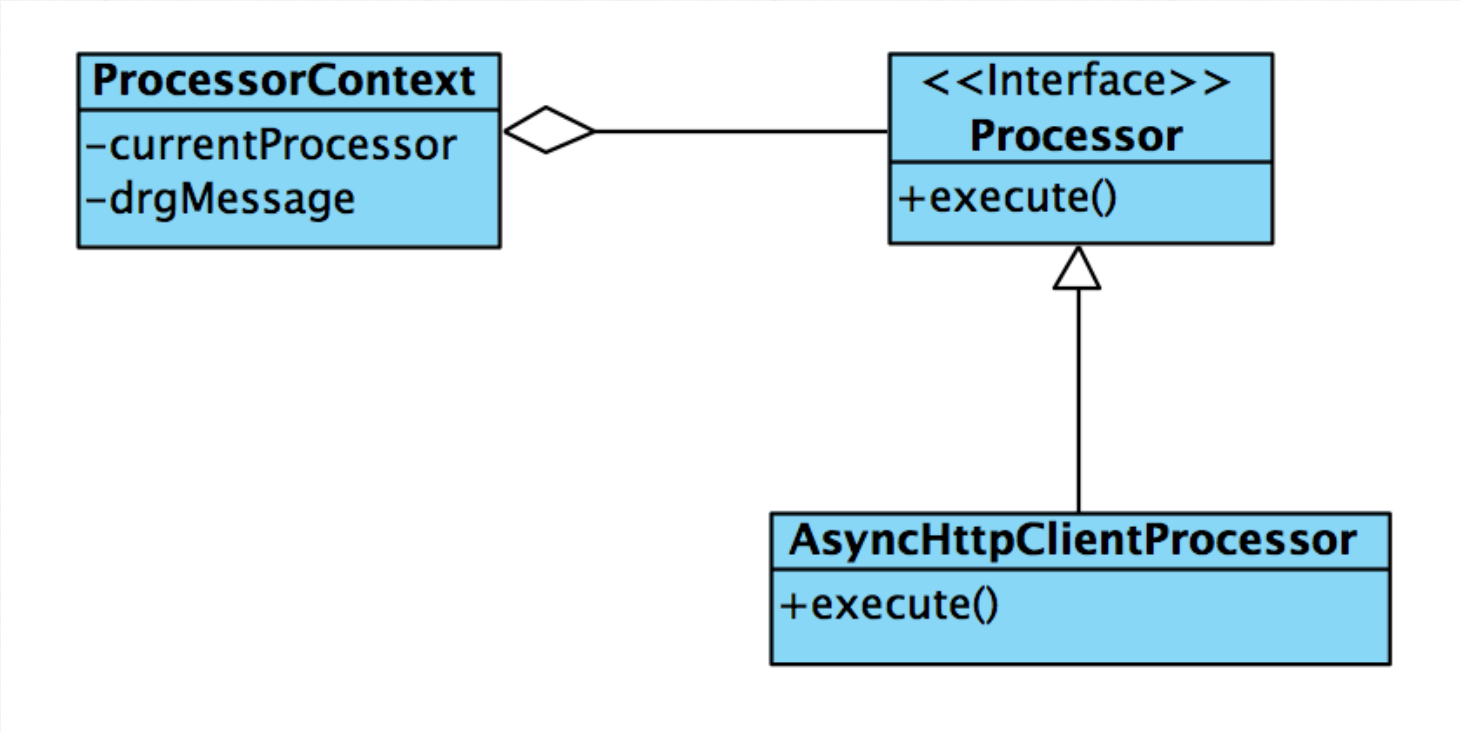
EXPERIENCES FROM REAL LIFE PROJECTS - API-GW



- API-GW
  - A **sequence** of processors.
  - Every processor may be async and potentially non blocking

EXPERIENCES FROM REAL LIFE PROJECTS - API-GW

- API-GW
  - State engine managing the processing steps

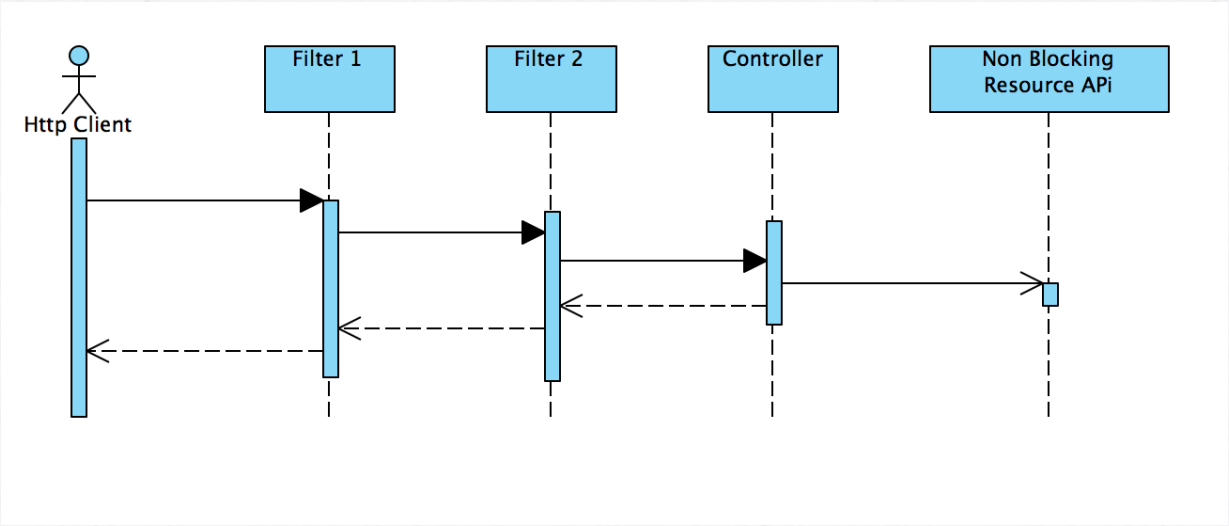


EXPERIENCES FROM REAL LIFE PROJECTS - FINDINGS

- Logging
  - Logback - MDC (Mapped Diagnostic Contexts)
    - » The MDC manages contextual information on a per thread basis
  - Log request over multiple threads?
    - » Child threads inherit a copy of the MDC context
    - » Manually move MDC context between threads

EXPERIENCES FROM REAL LIFE PROJECTS - FINDINGS

- Servlet Filters
  - Filter and Servlet in same thread (Servlet spec)
    - » Non-blocking request from filter?
  - Outbound filter
    - » Executes as soon as new thread is dispatched from servlet



Summary and next step

SUMMARY

- We have seen…
  - Requirements of improved scalability and resilience driven by increased number of connected devices (mobile devices and IoT)
  - The Reactive Manifesto to the rescue!
  - Non blocking I/O as a foundation
  - Dramatic differences can be demonstrated between blocking and non blocking solutions
  - Servlet 3.0 is the key to portable solutions
  - Spring MVC provides a really simple programming model
  - …however, the callback hell is waiting for you…
- Next time we will explain what we can do to eliminate the callback hell using reactive frameworks…

PREVIEW - *"THE WAY OUT OF CALLBACK HELL..."*

```
final DeferredResult<String> dr = new DeferredResult<>();

ListenableFuture<Response> execute = asyncHttpClient.execute(getUrl(1),
  (Response r1) -> {
    processResult(r1.getResponseBody());
    asyncHttpClient.execute(getUrl(2),
      (Response r2) -> {
        processResult(r2.getResponseBody());
        asyncHttpClient.execute(getUrl(3),
          ...
                  (Response r5) -> {
                    processResult(r5.getResponseBody());
                    dr.setResult(getTotalResult());
```

```
final DeferredResult<String> deferredResult = new DeferredResult<>();

Subscription subscription = Observable.<List<String>>just(new ArrayList<>())
  .flatMap(result -> doAsyncCall(result, 1, this::processResult))
  .flatMap(result -> doAsyncCall(result, 2, this::processResult))
  .flatMap(result -> doAsyncCall(result, 3, this::processResult))
  .flatMap(result -> doAsyncCall(result, 4, this::processResult))
  .flatMap(result -> doAsyncCall(result, 5, this::processResult))
  .subscribe(v -> deferredResult.setResult(getTotalResult(v)));
```

Q&A