Applying Reactive Programming with RxJava

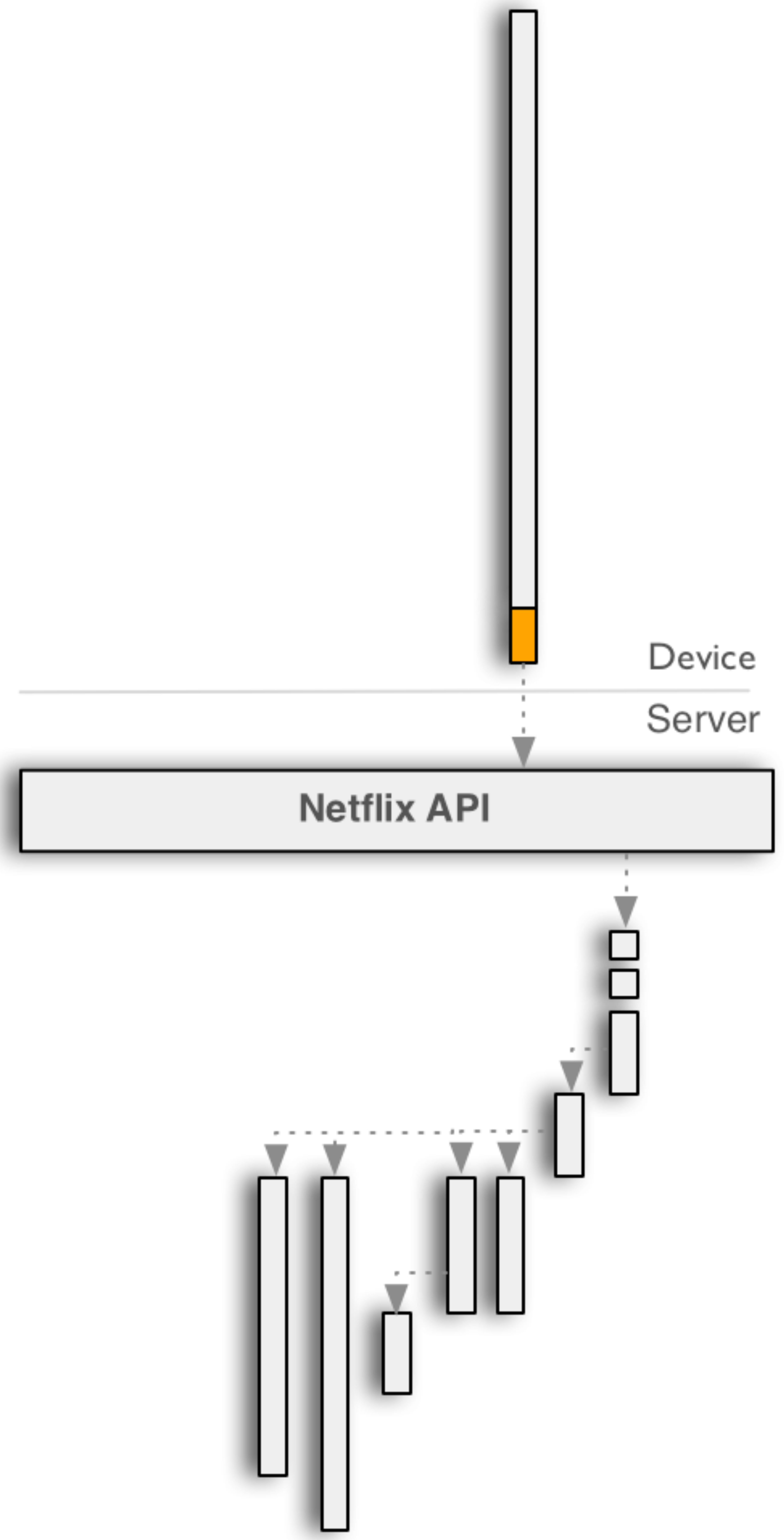
GOTO Chicago - May 2015

Ben Christensen
Edge Engineering at Netflix
@benjchristensen

NETFLIX | OSS
http://techblog.netflix.com/

NETFLIX


1919
CLOSE CITY
SCHOOL


Device
Server
Netflix API

async + callbacks

service composition

+

error handling

developer productivity

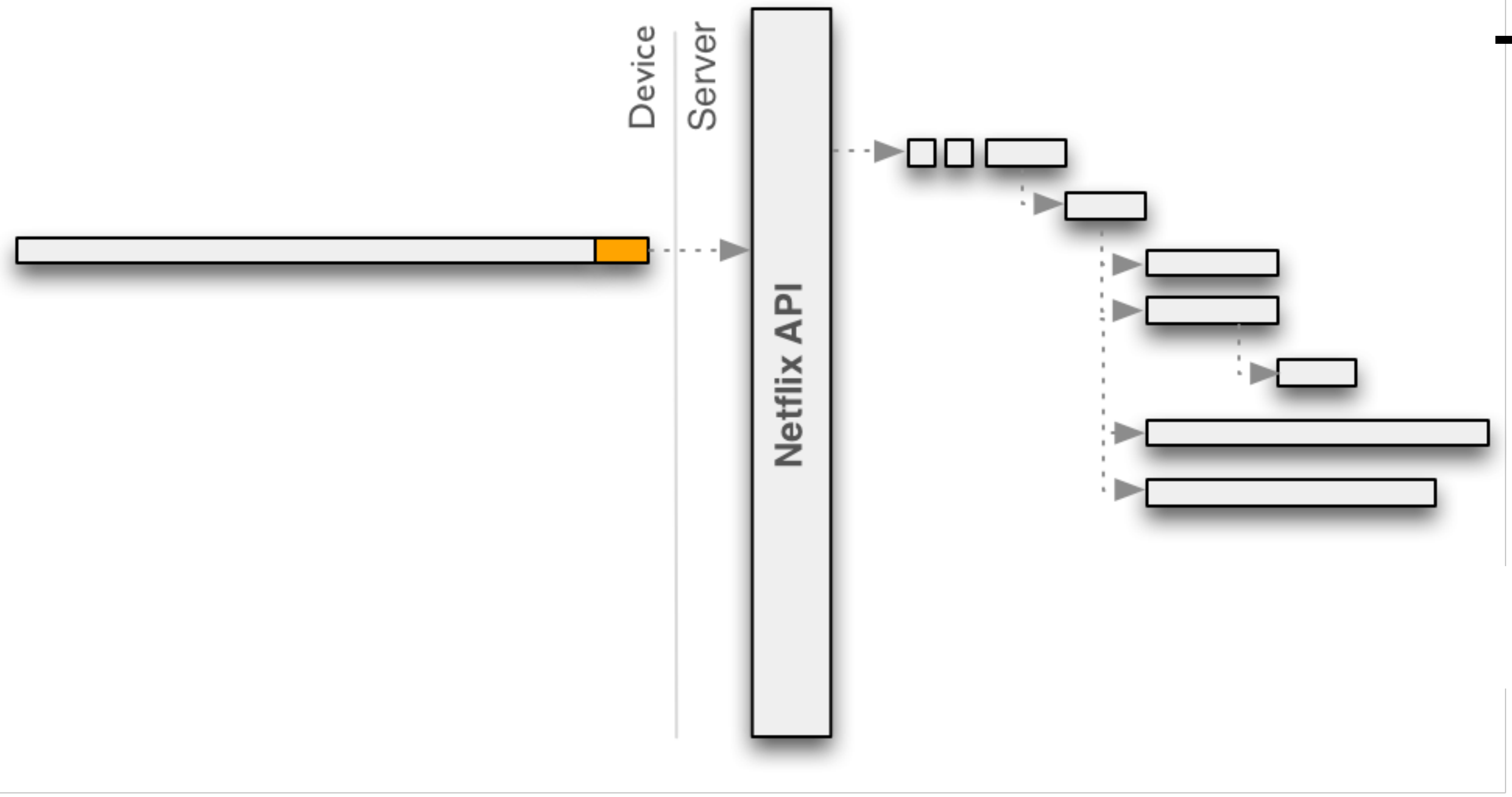
async + ~~callbacks~~

Future<T>

Future<List<T>>

Future<T>

Future<List<T>>



→ getListOfListOfMovies
  → forEachList
    → forEachMovie
      → getBookmark
      → getRating
      → getMetadata
      → getSimilars
      → etc

Future<T>

Future<List<T>>

Future<List<Future<T>>>

Future<T>

Future<List<T>>

Future<List<Future<T>>>

↓

Observable<T>

Everything is a Stream

getOrders → getProducts → getShippingStatus

```
List<Order> getOrders(int customerId)
List<Product> getProducts(Order order)
ShippingStatus getShippingStatus(Order o, Product p)
```

getOrders → getProducts → getShippingStatus

```
List<Order> getOrders(int customerId)
List<Product> getProducts(Order order)
ShippingStatus getShippingStatus(Order o, Product p)
```

Blocking APIs

getOrders → getProducts → getShippingStatus

```
Thread[main,5,main] [Sync] Shipping Status: Order: 1 Product: product-description-4114
Thread[main,5,main] [Sync] Shipping Status: Order: 1 Product: product-description-3433
Thread[main,5,main] [Sync] Shipping Status: Order: 1 Product: product-description-7521
Thread[main,5,main] [Sync] Shipping Status: Order: 1 Product: product-description-6487
Thread[main,5,main] [Sync] Shipping Status: Order: 1 Product: product-description-8354
```

IPC via Apache HTTP
Memcached
Cassandra
Tomcat and Servlets

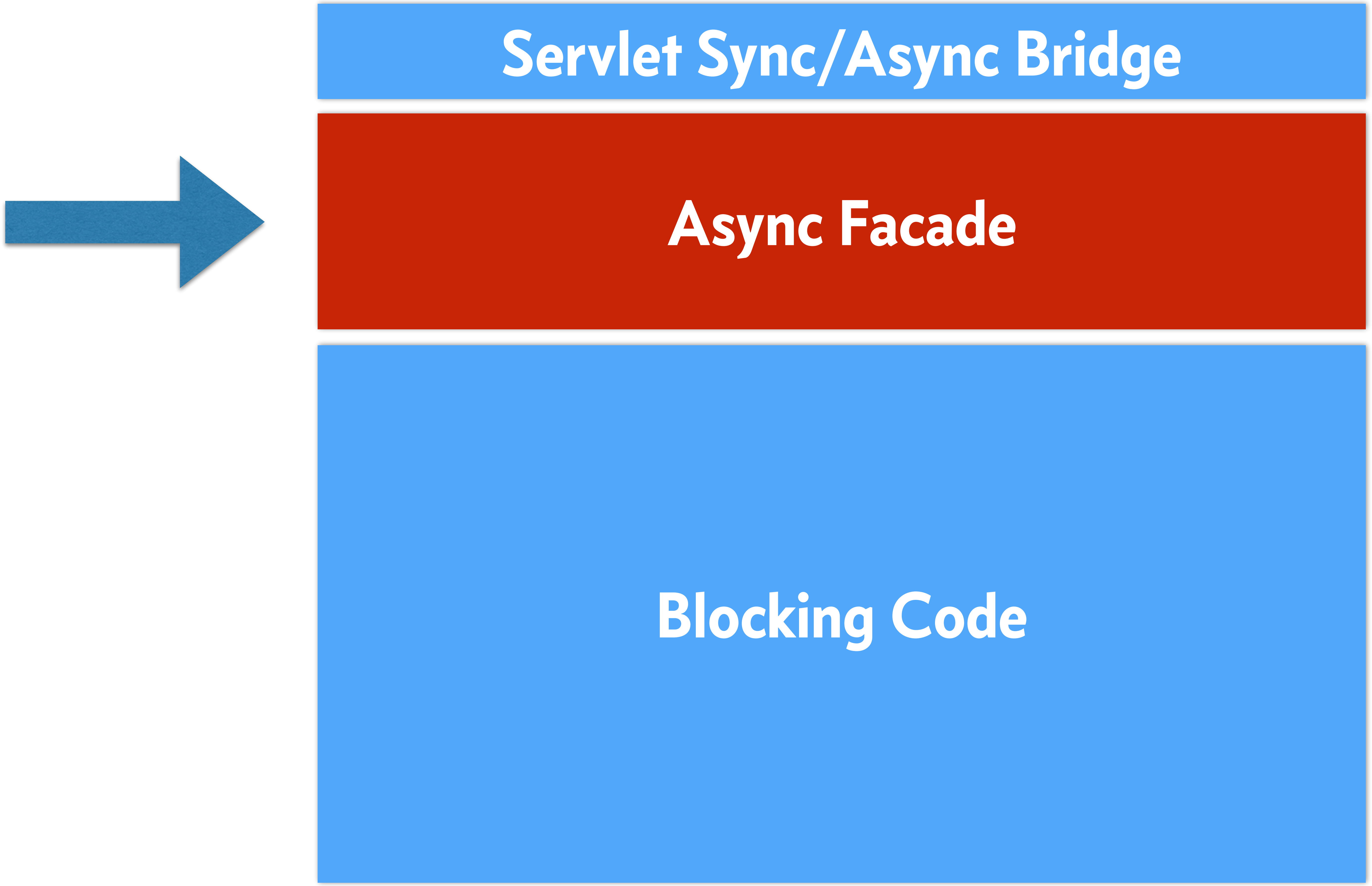
IPC via Apache HTTP
Memcached
Cassandra
Tomcat and Servlets

All Blocking

Many things could not **realistically** change

We *could* change our service layer

Async Facade


Servlet Sync/Async Bridge
Async Facade
Blocking Code

getOrders → getProducts → getShippingStatus

getOrdersAsync → getProductsAsync → getShippingStatusAsync

```
Observable<Order> getOrders(int customerId)
Observable<Product> getProducts(Order order)
Observable<ShippingStatus> getShippingStatus(Order o, Product p)
```

getOrders → getProducts → getShippingStatus

getOrdersAsync → getProductsAsync → getShippingStatusAsync

```
Observable<Order> getOrders(int customerId)
Observable<Product> getProducts(Order order)
Observable<ShippingStatus> getShippingStatus(Order o, Product p)
```

Observable APIs

```java
getOrdersAsync(1)
        .limit(1)
        .flatMap(o -> {
            return getProductsAsync(o)
                    .flatMap(p -> {
                        return getShippingStatusAsync(o, p);
                    });
        })
       .forEach(s -> System.out.println(Thread.currentThread()
                        + " [Async] Shipping Status: " + s));
```

```
getOrdersAsync(1)
        .limit(1)
        .flatMap(o -> {
            return getProductsAsync(o)
                .flatMap(p -> {
                    return getShippingStatusAsync(o, p);
                });
        })
       .forEach(s -> System.out.println(Thread.currentThread()
                           + " [Async] Shipping Status: " + s));
```

an async Observable

```java
getOrdersAsync(1)
        .limit(1)
        .flatMap(o -> {
            return getProductsAsync(o)
                .flatMap(p -> {
                    return getShippingStatusAsync(o, p);
                });
        })
       .forEach(s -> System.out.println(Thread.currentThread()
                            + " [Async] Shipping Status: " + s));
```

an async Observable stream

```
getOrdersAsync(1)
        .limit(1)
        .flatMap(o -> {
            return getProductsAsync(o)
                        .flatMap(p -> {
                            return getShippingStatusAsync(o, p);
                        });
        })
       .forEach(s -> System.out.println(Thread.currentThread()
                            + " [Async] Shipping Status: " + s));
```

another async Observable stream

```java
getOrdersAsync(1)
        .limit(1)
        .flatMap(o -> {
            return getProductsAsync(o)
                        .flatMap(p -> {
                            return getShippingStatusAsync(o, p);
                        });
        })
        .forEach(s -> System.out.println(Thread.currentThread()
                                + " [Async] Shipping Status: " + s));
```

```
Starting ...
Thread[RxCachedThreadScheduler-5,5,main] [Async] Shipping Status: Order: 1 Product: product-description-6487
Thread[RxCachedThreadScheduler-5,5,main] [Async] Shipping Status: Order: 1 Product: product-description-4114
Thread[RxCachedThreadScheduler-5,5,main] [Async] Shipping Status: Order: 1 Product: product-description-3433
Thread[RxCachedThreadScheduler-5,5,main] [Async] Shipping Status: Order: 1 Product: product-description-7521
Thread[RxCachedThreadScheduler-5,5,main] [Async] Shipping Status: Order: 1 Product: product-description-8354
Thread[RxCachedThreadScheduler-5,5,main] [Async] Shipping Status: Order: 1 Product: product-description-740
Thread[RxCachedThreadScheduler-5,5,main] [Async] Shipping Status: Order: 1 Product: product-description-2653
```

```
ShippingStatus getShippingStatus(Order o, Product p)
```

```
Observable<ShippingStatus> getShippingStatusAsync(Order o, Product p)
```

```java
Observable<ShippingStatus> getShippingStatusAsync(Order o, Product p)

  return Observable.defer(() -> {
      return Observable.just(getShippingStatus(o, p));
  }).subscribeOn(Schedulers.io());
```

```
Observable<ShippingStatus> getShippingStatusAsync(Order o, Product p)

  return Observable.defer(() -> {
      return Observable.just(getShippingStatus(o, p));
  }).subscribeOn(Schedulers.io());
```

```
Observable<ShippingStatus> getShippingStatusAsync(Order o, Product p)

  return Observable.defer(() -> {
      return Observable.just(getShippingStatus(o, p));
  }).subscribeOn(Schedulers.io());
```

```
Observable<ShippingStatus> getShippingStatusAsync(Order o, Product p)

  return Observable.defer(() -> {
      return Observable.just(getShippingStatus(o, p));
  }).subscribeOn(Schedulers.io());
```

```
Observable<ShippingStatus> getShippingStatusAsync(Order o, Product p)
```

```
return Observable.defer(() -> {
    return Observable.just(getShippingStatus(o, p));
}).subscribeOn(Schedulers.io());
```

```
Observable<ShippingStatus> getShippingStatusAsync(Order o, Product p)
```

```
return Observable.defer(() -> {
    return Observable.just(getShippingStatus(o, p));
}).subscribeOn(Schedulers.io());
```

```java
List<Order> orders = getOrders(1);
```

```java
Observable<Order> orders = getOrdersAsync(1);
```

```
List<Order> orders = getOrders(1);
```

```
Observable<Order> orders = getOrdersAsync(1);
```

```
List<Order> orders = getOrders(1);
```

```
Observable<Order> orders = getOrdersAsync(1);
```

```
Observable<Order> orders = getOrdersAsync(1);
```

```
return Observable.defer(() -> {
    return Observable.from(getOrders(customerId));
}).subscribeOn(Schedulers.io());
```

```java
Observable<Order> orders = getOrdersAsync(1);
```

```java
return Observable.defer(() -> {
    return Observable.from(getOrders(customerId));
}).subscribeOn(Schedulers.io());
```

```
Observable<Order> orders = getOrdersAsync(1);
```

```
return Observable.defer(() -> {
    return Observable.from(getOrders(customerId));
}).subscribeOn(Schedulers.io());
```

```java
Observable<Order> orders = getOrdersAsync(1);
```

```java
return Observable.defer(() -> {
    return Observable.from(getOrders(customerId));
}).subscribeOn(Schedulers.io());
```

```java
Observable<Order> orders = getOrdersAsync(1);
```

```java
return Observable.defer(() -> {
    return Observable.from(getOrders(customerId));
}).subscribeOn(Schedulers.io());
```

Wrapped with Threads using subscribeOn

Wrapped with Threads

JDBC?

```
List<Order> orders = getOrders(1);
```

```java
try (Connection c = Database.getConnection()) {
    ResultSet rs = c.createStatement()
                    .executeQuery("select * from orders");
    ArrayList<Order> orders = new ArrayList<>();
    while (rs.next()) {
        orders.add(new Order(rs.getInt(1)));
    }
    rs.close();
    return orders;
} catch (Exception e) {
    throw new RuntimeException(e);
}
```

```
List<Order> orders = getOrders(1);
```

```
try (Connection c = Database.getConnection()) {
    ResultSet rs = c.createStatement()
                    .executeQuery("select * from orders");
    ArrayList<Order> orders = new ArrayList<>();
    while (rs.next()) {
        orders.add(new Order(rs.getInt(1)));
    }
    rs.close();
    return orders;
} catch (Exception e) {
    throw new RuntimeException(e);
}
```

```
Observable<Order> orders = getOrdersAsync(1);

return Observable.<Order> create(o -> {
    try (Connection c = Database.getConnection()) {
        ResultSet rs = c.createStatement()
                                   .executeQuery("select * from orders");
        while (rs.next() && !o.isUnsubscribed()) {
            o.onNext(new Order(rs.getInt(1)));
        }
        rs.close();
        o.onCompleted();
    } catch (Exception e) {
        o.onError(e);
    }
}).subscribeOn(Schedulers.io());
```

```
              Observable<Order> orders = getOrdersAsync(1);

return Observable.<Order> create(o -> {
    try (Connection c = Database.getConnection()) {
        ResultSet rs = c.createStatement()
                              .executeQuery("select * from orders");
        while (rs.next() && !o.isUnsubscribed()) {
            o.onNext(new Order(rs.getInt(1)));
        }
        rs.close();
        o.onCompleted();
    } catch (Exception e) {
        o.onError(e);
    }
}).subscribeOn(Schedulers.io());
```

```java
Observable<Order> orders = getOrdersAsync(1);

return Observable.<Order> create(o -> {
    try (Connection c = Database.getConnection()) {
        ResultSet rs = c.createStatement()
                              .executeQuery("select * from orders");
        while (rs.next() && !o.isUnsubscribed()) {
            o.onNext(new Order(rs.getInt(1)));
        }
        rs.close();
        o.onCompleted();
    } catch (Exception e) {
        o.onError(e);
    }
}).subscribeOn(Schedulers.io());
```

```
            Observable<Order> orders = getOrdersAsync(1);

return Observable.<Order> create(o -> {
    try (Connection c = Database.getConnection()) {
        ResultSet rs = c.createStatement()
                         .executeQuery("select * from orders");
        while (rs.next() && !o.isUnsubscribed()) {
            o.onNext(new Order(rs.getInt(1)));
        }
        rs.close();
        o.onCompleted();
    } catch (Exception e) {
        o.onError(e);
    }
}).subscribeOn(Schedulers.io());
```

```
            Observable<Order> orders = getOrdersAsync(1);

return Observable.<Order> create(o -> {
    try (Connection c = Database.getConnection()) {
        ResultSet rs = c.createStatement()
                                .executeQuery("select * from orders");
        while (rs.next() && !o.isUnsubscribed()) {
            o.onNext(new Order(rs.getInt(1)));
        }
        rs.close();
        o.onCompleted();
    } catch (Exception e) {
        o.onError(e);
    }
}).subscribeOn(Schedulers.io());
```

```
Observable<Order> orders = getOrdersAsync(1);

return Observable.<Order> create(o -> {
    try (Connection c = Database.getConnection()) {
        ResultSet rs = c.createStatement()
                         .executeQuery("select * from orders");
        while (rs.next() && !o.isUnsubscribed()) {
            o.onNext(new Order(rs.getInt(1)));
        }
        rs.close();
        o.onCompleted();
    } catch (Exception e) {
        o.onError(e);
    }
}).subscribeOn(Schedulers.io());
```

or with “reactive pull” backpressure …

```java
return Observable.create(AbstractOnSubscribe.<Order, ResultSet> create(s -> {
    ResultSet rs = s.state();
    try {
        if (rs.next()) {
            s.onNext(new Order(rs.getInt(1)));
        } else {
            s.onCompleted();
        }
    } catch (SQLException e) {
        s.onError(e);
    }
}, s -> {
    Connection c = Database.getConnection();
    try {
            Statement stmt = c.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                                               java.sql.ResultSet.CONCUR_READ_ONLY);
            stmt.setFetchSize(Integer.MIN_VALUE);
            return stmt.executeQuery("select * from orders");
    } catch (SQLException e) {
        s.onError(e);
        return null;
    }
}, rs -> rs.close() } // ignoring try/catch for slide brevity
)).subscribeOn(Schedulers.io());
```

```java
return Observable.create(AbstractOnSubscribe.<Order, ResultSet> create(s -> {
    ResultSet rs = s.state();
    try {
        if (rs.next()) {
            s.onNext(new Order(rs.getInt(1)));
        } else {
            s.onCompleted();
        }
    } catch (SQLException e) {
        s.onError(e);
    }
}, s -> {
    Connection c = Database.getConnection();
    try {
            Statement stmt = c.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                                               java.sql.ResultSet.CONCUR_READ_ONLY);
            stmt.setFetchSize(Integer.MIN_VALUE);
            return stmt.executeQuery("select * from orders");
    } catch (SQLException e) {
        s.onError(e);
        return null;
    }
}, rs -> rs.close() } // ignoring try/catch for slide brevity
)).subscribeOn(Schedulers.io());
```

```
return Observable.create(AbstractOnSubscribe.<Order, ResultSet> create(s -> {
    ResultSet rs = s.state();
    try {
        if (rs.next()) {
            s.onNext(new Order(rs.getInt(1)));
        } else {
            s.onCompleted();
        }
    } catch (SQLException e) {
        s.onError(e);
    }
}, s -> {
    Connection c = Database.getConnection();
    try {
            Statement stmt = c.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                                                     java.sql.ResultSet.CONCUR_READ_ONLY);
            stmt.setFetchSize(Integer.MIN_VALUE);
            return stmt.executeQuery("select * from orders");
    } catch (SQLException e) {
        s.onError(e);
        return null;
    }
}, rs -> rs.close() } // ignoring try/catch for slide brevity
)).subscribeOn(Schedulers.io());
```

```java
return Observable.create(AbstractOnSubscribe.<Order, ResultSet> create(s -> {
    ResultSet rs = s.state();
    try {
        if (rs.next()) {
            s.onNext(new Order(rs.getInt(1)));
        } else {
            s.onCompleted();
        }
    } catch (SQLException e) {
        s.onError(e);
    }
}, s -> {
    Connection c = Database.getConnection();
    try {
            Statement stmt = c.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                                               java.sql.ResultSet.CONCUR_READ_ONLY);
            stmt.setFetchSize(Integer.MIN_VALUE);
            return stmt.executeQuery("select * from orders");
    } catch (SQLException e) {
        s.onError(e);
        return null;
    }
}, rs -> rs.close() } // ignoring try/catch for slide brevity
)).subscribeOn(Schedulers.io());
```

```java
return Observable.create(AbstractOnSubscribe.<Order, ResultSet> create(s -> {
    ResultSet rs = s.state();
    try {
        if (rs.next()) {
            s.onNext(new Order(rs.getInt(1)));
        } else {
            s.onCompleted();
        }
    } catch (SQLException e) {
        s.onError(e);
    }
}, s -> {
    Connection c = Database.getConnection();
    try {
            Statement stmt = c.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                                               java.sql.ResultSet.CONCUR_READ_ONLY);
            stmt.setFetchSize(Integer.MIN_VALUE);
            return stmt.executeQuery("select * from orders");
    } catch (SQLException e) {
        s.onError(e);
        return null;
    }
}, rs -> rs.close() } // ignoring try/catch for slide brevity
)).subscribeOn(Schedulers.io());
```

```java
return Observable.create(AbstractOnSubscribe.<Order, ResultSet> create(s -> {
    ResultSet rs = s.state();
    try {
        if (rs.next()) {
            s.onNext(new Order(rs.getInt(1)));
        } else {
            s.onCompleted();
        }
    } catch (SQLException e) {
        s.onError(e);
    }
}, s -> {
    Connection c = Database.getConnection();
    try {
            Statement stmt = c.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                                               java.sql.ResultSet.CONCUR_READ_ONLY);
            stmt.setFetchSize(Integer.MIN_VALUE);
            return stmt.executeQuery("select * from orders");
    } catch (SQLException e) {
        s.onError(e);
        return null;
    }
}, rs -> rs.close() } // ignoring try/catch for slide brevity
)).subscribeOn(Schedulers.io());
```

```java
return Observable.create(AbstractOnSubscribe.<Order, ResultSet> create(s -> {
    ResultSet rs = s.state();
    try {
        if (rs.next()) {
            s.onNext(new Order(rs.getInt(1)));
        } else {
            s.onCompleted();
        }
    } catch (SQLException e) {
        s.onError(e);
    }
}, s -> {
    Connection c = Database.getConnection();
    try {
            Statement stmt = c.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                                               java.sql.ResultSet.CONCUR_READ_ONLY);
            stmt.setFetchSize(Integer.MIN_VALUE);
            return stmt.executeQuery("select * from orders");
    } catch (SQLException e) {
        s.onError(e);
        return null;
    }
}, rs -> rs.close() } // ignoring try/catch for slide brevity
)).subscribeOn(Schedulers.io());
```

```java
return Observable.create(AbstractOnSubscribe.<Order, ResultSet> create(s -> {
    ResultSet rs = s.state();
    try {
        if (rs.next()) {
            s.onNext(new Order(rs.getInt(1)));
        } else {
            s.onCompleted();
        }
    } catch (SQLException e) {
        s.onError(e);
    }
}, s -> {
    Connection c = Database.getConnection();
    try {
            Statement stmt = c.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                                                java.sql.ResultSet.CONCUR_READ_ONLY);
            stmt.setFetchSize(Integer.MIN_VALUE);
            return stmt.executeQuery("select * from orders");
    } catch (SQLException e) {
        s.onError(e);
        return null;
    }
}, rs -> rs.close() } // ignoring try/catch for slide brevity
)).subscribeOn(Schedulers.io());
```

```
return Observable.create(AbstractOnSubscribe.<Order, ResultSet> create(s -> {
    ResultSet rs = s.state();
    try {
        if (rs.next()) {
            s.onNext(new Order(rs.getInt(1)));
        } else {
            s.onCompleted();
        }
    } catch (SQLException e) {
        s.onError(e);
    }
}, s -> {
    Connection c = Database.getConnection();
    try {
            Statement stmt = c.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                                                java.sql.ResultSet.CONCUR_READ_ONLY);
            stmt.setFetchSize(Integer.MIN_VALUE);
            return stmt.executeQuery("select * from orders");
    } catch (SQLException e) {
        s.onError(e);
        return null;
    }
}, rs -> rs.close() } // ignoring try/catch for slide brevity
)).subscribeOn(Schedulers.io());
```

```java
return Observable.create(AbstractOnSubscribe.<Order, ResultSet> create(s -> {
    ResultSet rs = s.state();
    try {
        if (rs.next()) {
            s.onNext(new Order(rs.getInt(1)));
        } else {
            s.onCompleted();
        }
    } catch (SQLException e) {
        s.onError(e);
    }
}, s -> {
    Connection c = Database.getConnection();
    try {
            Statement stmt = c.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                                               java.sql.ResultSet.CONCUR_READ_ONLY);
            stmt.setFetchSize(Integer.MIN_VALUE);
            return stmt.executeQuery("select * from orders");
    } catch (SQLException e) {
        s.onError(e);
        return null;
    }
}, rs -> rs.close() } // ignoring try/catch for slide brevity
)).subscribeOn(Schedulers.io());
```

```java
return Observable.create(AbstractOnSubscribe.<Order, ResultSet> create(s -> {
    ResultSet rs = s.state();
    try {
        if (rs.next()) {
            s.onNext(new Order(rs.getInt(1)));
        } else {
            s.onCompleted();
        }
    } catch (SQLException e) {
        s.onError(e);
    }
}, s -> {
    Connection c = Database.getConnection();
    try {
            Statement stmt = c.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                                                  java.sql.ResultSet.CONCUR_READ_ONLY);
            stmt.setFetchSize(Integer.MIN_VALUE);
            return stmt.executeQuery("select * from orders");
    } catch (SQLException e) {
        s.onError(e);
        return null;
    }
}, rs -> rs.close() } // ignoring try/catch for slide brevity
)).subscribeOn(Schedulers.io());
```

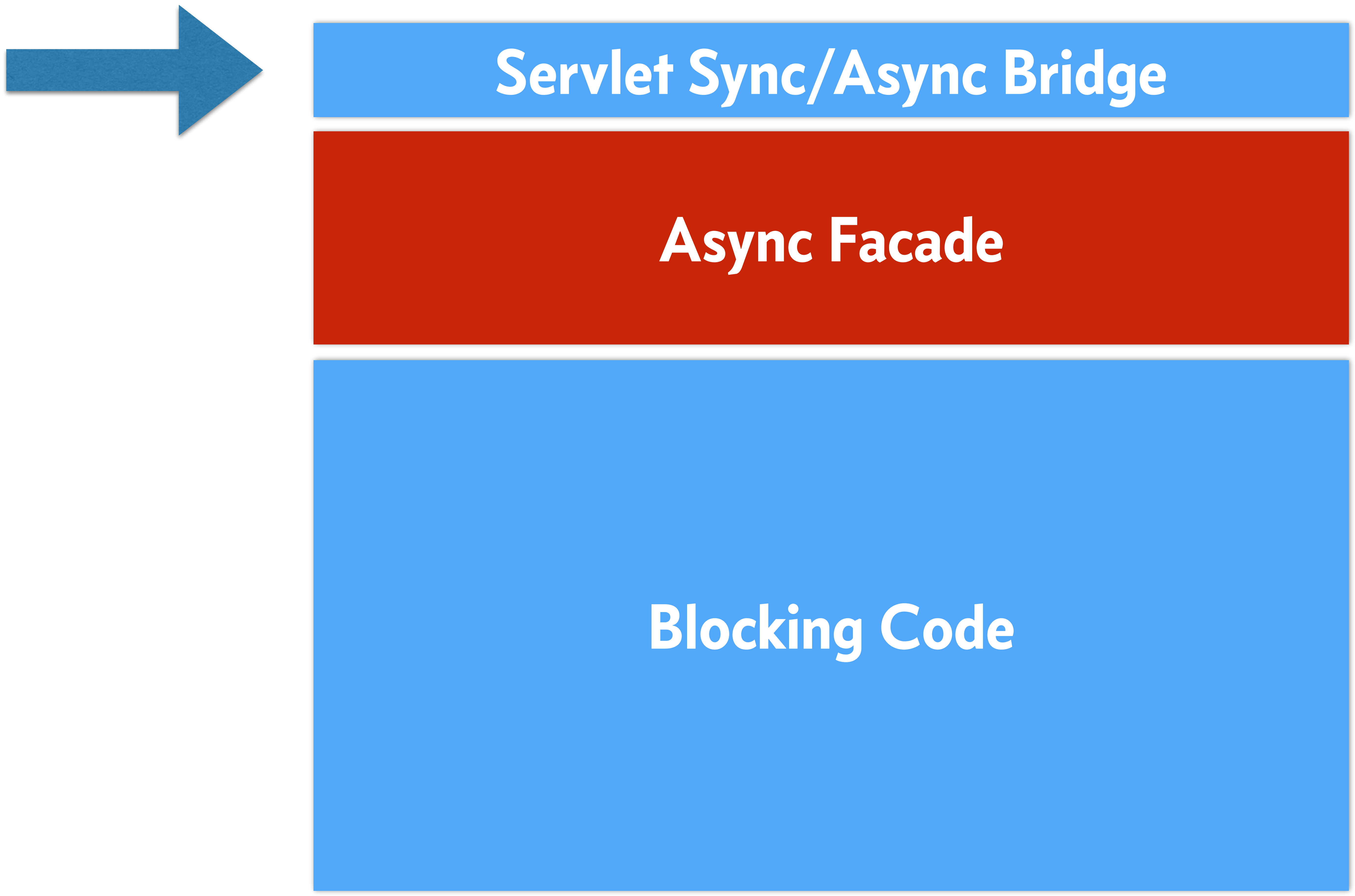
```java
return Observable.create(AbstractOnSubscribe.<Order, ResultSet> create(s -> {
    ResultSet rs = s.state();
    try {
        if (rs.next()) {
            s.onNext(new Order(rs.getInt(1)));
        } else {
            s.onCompleted();
        }
    } catch (SQLException e) {
        s.onError(e);
    }
}, s -> {
    Connection c = Database.getConnection();
    try {
            Statement stmt = c.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                                               java.sql.ResultSet.CONCUR_READ_ONLY);
            stmt.setFetchSize(Integer.MIN_VALUE);
            return stmt.executeQuery("select * from orders");
    } catch (SQLException e) {
        s.onError(e);
        return null;
    }
}, rs -> rs.close() } // ignoring try/catch for slide brevity
)).subscribeOn(Schedulers.io());
```

```java
return Observable.create(AbstractOnSubscribe.<Order, ResultSet> create(s -> {
    ResultSet rs = s.state();
    try {
        if (rs.next()) {
            s.onNext(new Order(rs.getInt(1)));
        } else {
            s.onCompleted();
        }
    } catch (SQLException e) {
        s.onError(e);
    }
}, s -> {
    Connection c = Database.getConnection();
    try {
            Statement stmt = c.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                                               java.sql.ResultSet.CONCUR_READ_ONLY);
            stmt.setFetchSize(Integer.MIN_VALUE);
            return stmt.executeQuery("select * from orders");
    } catch (SQLException e) {
        s.onError(e);
        return null;
    }
}, rs -> rs.close() } // ignoring try/catch for slide brevity
)).subscribeOn(Schedulers.io());
```

```
return Observable.create(AbstractOnSubscribe.<Order, ResultSet> create(s -> {
    ResultSet rs = s.state();
    try {
        if (rs.next()) {
            s.onNext(new Order(rs.getInt(1)));
        } else {
            s.onCompleted();
        }
    } catch (SQLException e) {
        s.onError(e);
    }
}, s -> {
    Connection c = Database.getConnection();
    try {
            Statement stmt = c.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                                               java.sql.ResultSet.CONCUR_READ_ONLY);
            stmt.setFetchSize(Integer.MIN_VALUE);
            return stmt.executeQuery("select * from orders");
    } catch (SQLException e) {
        s.onError(e);
        return null;
    }
}, rs -> rs.close() } // ignoring try/catch for slide brevity
)).subscribeOn(Schedulers.io());
```

Observable.defer
Observable.from/just
Observable.subscribeOn

Observable.create


Servlet Sync/Async Bridge
Async Facade
Blocking Code

```java
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                                             throws ServletException, IOException {
    try {
        // perform service composition asynchronously
        getShippingStatusForCustomerOrdersAsync(req.getParameter(“customerId"))
                .toBlocking() // block on servlet thread
                .forEach(status -> {
                    try {
                        // write output using blocking IO
                        resp.getWriter().write(status.toString());
                    } catch (Exception e) {
                        throw new RuntimeException("unable to write", e);
                    }
                });
    } catch (Exception e) {
        // toBlocking and forEach will convert async onError events into thrown exceptions
        resp.setStatus(500);
        // write and log error ...
    }
}
```

```java
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                                        throws ServletException, IOException {
    try {
        // perform service composition asynchronously
        getShippingStatusForCustomerOrdersAsync(req.getParameter("customerId"))
                .toBlocking() // block on servlet thread
                .forEach(status -> {
                    try {
                        // write output using blocking IO
                        resp.getWriter().write(status.toString());
                    } catch (Exception e) {
                        throw new RuntimeException("unable to write", e);
                    }
                });
    } catch (Exception e) {
        // toBlocking and forEach will convert async onError events into thrown exceptions
        resp.setStatus(500);
        // write and log error ...
    }
}
```

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                                             throws ServletException, IOException {
    try {
        // perform service composition asynchronously
        getShippingStatusForCustomerOrdersAsync(req.getParameter("customerId"))
                .toBlocking() // block on servlet thread
                .forEach(status -> {
                    try {
                        // write output using blocking IO
                        resp.getWriter().write(status.toString());
                    } catch (Exception e) {
                        throw new RuntimeException("unable to write", e);
                    }
                });
    } catch (Exception e) {
        // toBlocking and forEach will convert async onError events into thrown exceptions
        resp.setStatus(500);
        // write and log error ...
    }
}
```

```java
Observable<ShippingStatus> getShippingStatusForCustomerOrdersAsync(String customerId) {
    return getOrdersAsync(Integer.parseInt(customerId))
            .limit(1)
            .flatMap(o -> {
                return getProductsAsync(o)
                        .flatMap(p -> {
                            return getShippingStatusAsync(o, p);
                        });
            });
}
```

```java
Observable<ShippingStatus> getShippingStatusForCustomerOrdersAsync(String customerId) {
    return getOrdersAsync(Integer.parseInt(customerId))
            .limit(1)
            .flatMap(o -> {
                return getProductsAsync(o)
                        .flatMap(p -> {
                            return getShippingStatusAsync(o, p);
                        });
            });
}
```

```
Observable<ShippingStatus> getShippingStatusForCustomerOrdersAsync(String customerId) {
    return getOrdersAsync(Integer.parseInt(customerId))
            .limit(1)
            .flatMap(o -> {
                return getProductsAsync(o)
                        .flatMap(p -> {
                            return getShippingStatusAsync(o, p);
                        });
            });
}
```

```java
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                                            throws ServletException, IOException {
    try {
        // perform service composition asynchronously
        getShippingStatusForCustomerOrdersAsync(req.getParameter("customerId"))
                .toBlocking() // block on servlet thread
                .forEach(status -> {
                    try {
                        // write output using blocking IO
                        resp.getWriter().write(status.toString());
                    } catch (Exception e) {
                        throw new RuntimeException("unable to write", e);
                    }
                });
    } catch (Exception e) {
        // toBlocking and forEach will convert async onError events into thrown exceptions
        resp.setStatus(500);
        // write and log error ...
    }
}
```

```java
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                                             throws ServletException, IOException {
    try {
        // perform service composition asynchronously
        getShippingStatusForCustomerOrdersAsync(req.getParameter("customerId"))
                .toBlocking() // block on servlet thread
                .forEach(status -> {
                    try {
                        // write output using blocking IO
                        resp.getWriter().write(status.toString());
                    } catch (Exception e) {
                        throw new RuntimeException("unable to write", e);
                    }
                });
    } catch (Exception e) {
        // toBlocking and forEach will convert async onError events into thrown exceptions
        resp.setStatus(500);
        // write and log error ...
    }
}
```

```java
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                                              throws ServletException, IOException {
    try {
        // perform service composition asynchronously
        getShippingStatusForCustomerOrdersAsync(req.getParameter("customerId"))
                .toBlocking() // block on servlet thread
                .forEach(status -> {
                    try {
                        // write output using blocking IO
                        resp.getWriter().write(status.toString());
                    } catch (Exception e) {
                        throw new RuntimeException("unable to write", e);
                    }
                });
    } catch (Exception e) {
        // toBlocking and forEach will convert async onError events into thrown exceptions
        resp.setStatus(500);
        // write and log error ...
    }
}
```

Observable.toBlocking

Observable.**toBlocking**

for bridging between async and sync


So how did it go adding reactive
to an existing codebase?

First attempt took 3 tries

What the *&!@#$^!!!

Tech **Worked** …

… but …

Needed to relearn idiomatic solutions

```java
Iterable<String> is = Arrays.asList("a", "b", "c");
ArrayList<String> ts = new ArrayList<>();
for(String s : is) {
    ts.add(s + "-modified");
}
// do stuff with ts ...
```

```java
Observable<String> os = Observable.just("a", "b", "c");
os.map(s -> s + "-modified");
// do stuff with os ...
```

```java
try {
    for (String s : is) {
        ts.add(s + "-modified");
    }
} catch (Exception e) {
    // do something with exception
}
```

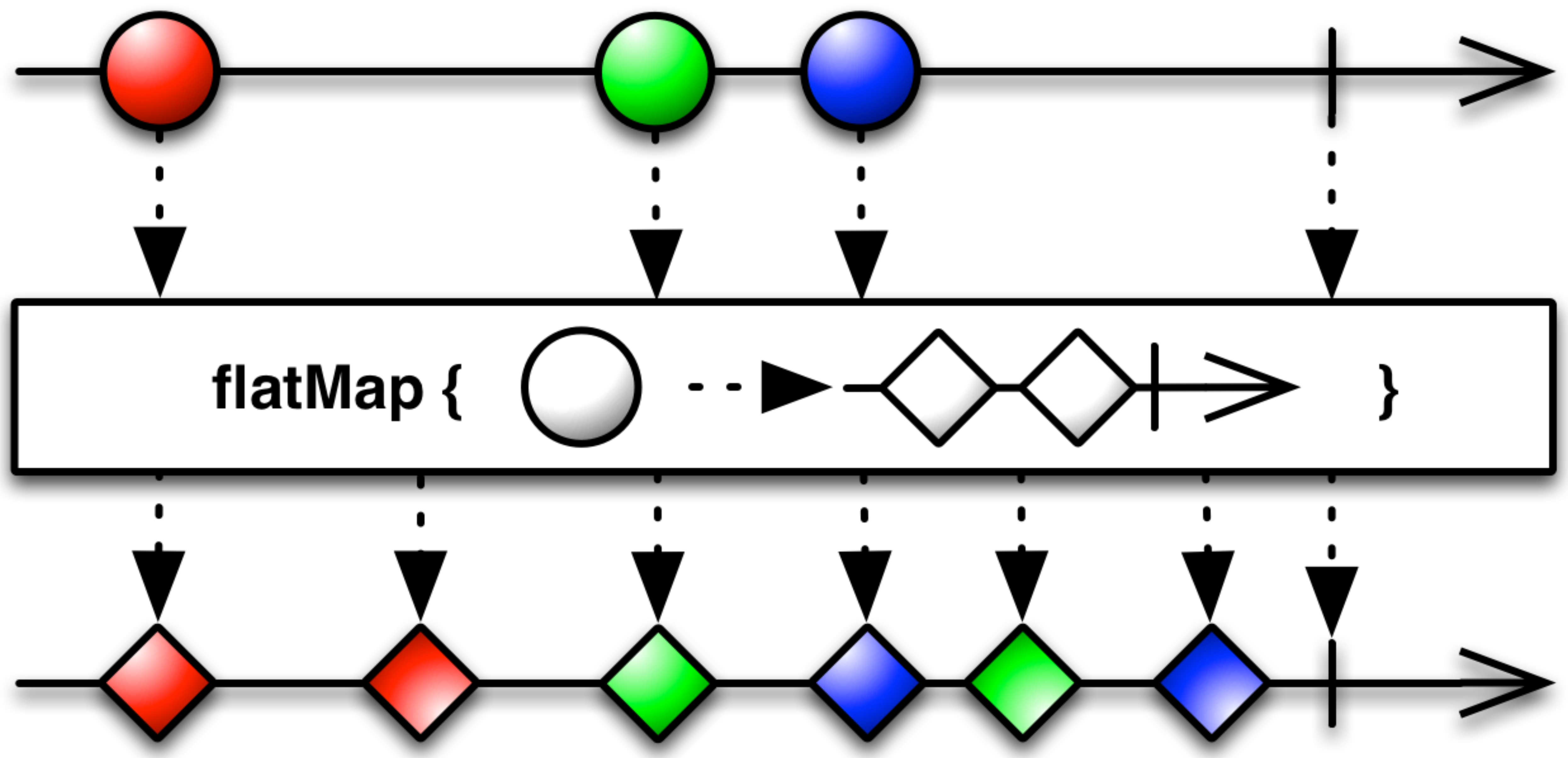
```java
os.map(s -> s + "-modified")
  .onErrorResumeNext(exception -> {
      // do something with exception
      return Observable.empty();
  });
```

Needed to **invest** in documentation

flatMap

```
public final <R> Observable<R> flatMap(Func1<? super T,? extends Observable<? extends R>> func)
```

Returns an Observable that emits items based on applying a function that you supply to each item emitted by the source Observable, where that function returns an Observable, and then merging those resulting Observables and emitting the results of this merger.



Scheduler:

`flatMap` does not operate by default on a particular `Scheduler`.

Parameters:

`func` - a function that, when applied to an item emitted by the source Observable, returns an Observable

Returns:

an Observable that emits the result of applying the transformation function to each item emitted by the source Observable and merging the results of the Observables obtained from this transformation

See Also:

ReactiveX operators documentation: FlatMap

Composition via Observable Operators

Observables and observers are only the start of ReactiveX. By themselves they'd be nothing more than a slight extension of the standard observer pattern, better suited to handling a sequence of events rather than a single callback.

The real power comes with the "reactive extensions" (hence "ReactiveX") — operators that allow you to transform, combine, manipulate, and work with the sequences of items emitted by Observables.

These Rx operators allow you to compose asynchronous sequences together in a declarative manner with all the efficiency benefits of callbacks but without the drawbacks of nesting callback handlers that are typically associated with asynchronous systems.

This documentation groups information about the various operators and examples of their usage into the following pages:

Creating Observables
`Create`, `Defer`, `Empty`/`Never`/`Throw`, `From`, `Interval`, `Just`, `Range`, `Repeat`, `Start`, and `Timer`

Transforming Observable Items
`Buffer`, `FlatMap`, `GroupBy`, `Map`, `Scan`, and `Window`

Filtering Observables
`Debounce`, `Distinct`, `ElementAt`, `Filter`, `First`, `IgnoreElements`, `Last`, `Sample`, `Skip`, `SkipLast`, `Take`, and `TakeLast`

Combining Observables
`And`/`Then`/`When`, `CombineLatest`, `Join`, `Merge`, `StartWith`, `Switch`, and `Zip`

Error Handling Operators
`Catch` and `Retry`

Utility Operators
`Delay`, `Do`, `Materialize`/`Dematerialize`, `ObserveOn`, `Serialize`, `Subscribe`, `SubscribeOn`, `TimeInterval`, `Timeout`, `Timestamp`, and `Using`

Conditional and Boolean Operators
`All`, `Amb`, `Contains`, `DefaultIfEmpty`, `SequenceEqual`, `SkipUntil`, `SkipWhile`, `TakeUntil`, and `TakeWhile`

Mathematical and Aggregate Operators
`Average`, `Concat`, `Count`, `Max`, `Min`, `Reduce`, and `Sum`

Converting Observables
`To`

Connectable Observable Operators
`Connect`, `Publish`, `RefCount`, and `Replay`

ReactiveX Introduction Docs Languages Resources Community

- `IgnoreElements` — do not emit any items from an Observable but mirror its termination notification
- `Last` — emit only the last item emitted by an Observable
- `Sample` — emit the most recent item emitted by an Observable within periodic time intervals
- `Skip` — suppress the first *n* items emitted by an Observable
- `SkipLast` — suppress the last *n* items emitted by an Observable
- `Take` — emit only the first *n* items emitted by an Observable
- `TakeLast` — emit only the last *n* items emitted by an Observable

Combining Observables

Operators that work with multiple source Observables to create a single Observable

- `And` / `Then` / `When` — combine sets of items emitted by two or more Observables by means of `Pattern` and `Plan` intermediaries
- `CombineLatest` — when an item is emitted by either of two Observables, combine the latest item emitted by each Observable via a specified function and emit items based on the results of this function
- `Join` — combine items emitted by two Observables whenever an item from one Observable is emitted during a time window defined according to an item emitted by the other Observable
- `Merge` — combine multiple Observables into one by merging their emissions
- `StartWith` — emit a specified sequence of items before beginning to emit the items from the source Observable
- `Switch` — convert an Observable that emits Observables into a single Observable that emits the items emitted by the most-recently-emitted of those Observables
- `Zip` — combine the emissions of multiple Observables together via a specified function and emit single items for each combination based on the results of this function

Error Handling Operators

Operators that help to recover from error notifications from an Observable

- `Catch` — recover from an `onError` notification by continuing the sequence without error
- `Retry` — if a source Observable sends an `onError` notification, resubscribe to it in the hopes that it will complete without error

Observable Utility Operators

A toolbox of useful Operators for working with Observables

- `Delay` — shift the emissions from an Observable forward in time by a particular amount
- `Do` — register an action to take upon a variety of Observable lifecycle events
- `Materialize` / `Dematerialize` — represent both the items emitted and the notifications sent as

Needed to allow for time and mistakes

Also … Unit Testing & Debugging

Async Unit Tests

```java
TestSubscriber<String> ts = new TestSubscriber<>();

Observable.interval(200, TimeUnit.MILLISECONDS)
        .take(5)
        .map(i -> {
            return i + " value";
        }).subscribe(ts);

ts.awaitTerminalEvent();
ts.assertNoErrors();
ts.assertReceivedOnNext(Arrays.asList("0 value", "1 value",
                                      "2 value", "3 value", "4 value"));
```

```java
TestSubscriber<String> ts = new TestSubscriber<>();

Observable.interval(200, TimeUnit.MILLISECONDS)
        .take(5)
        .map(i -> {
            return i + " value";
        }).subscribe(ts);

ts.awaitTerminalEvent();
ts.assertNoErrors();
ts.assertReceivedOnNext(Arrays.asList("0 value", "1 value",
                                      "2 value", "3 value", "4 value"));
```

```java
TestSubscriber<String> ts = new TestSubscriber<>();

Observable.interval(200, TimeUnit.MILLISECONDS)
        .take(5)
        .map(i -> {
            return i + " value";
        }).subscribe(ts);

ts.awaitTerminalEvent();
ts.assertNoErrors();
ts.assertReceivedOnNext(Arrays.asList("0 value", "1 value",
                                      "2 value", "3 value", "4 value"));
```

```java
TestSubscriber<String> ts = new TestSubscriber<>();

Observable.interval(200, TimeUnit.MILLISECONDS)
        .take(5)
        .map(i -> {
            return i + " value";
        }).subscribe(ts);

ts.awaitTerminalEvent();
ts.assertNoErrors();
ts.assertReceivedOnNext(Arrays.asList("0 value", "1 value",
                                      "2 value", "3 value", "4 value"));
```

```java
TestSubscriber<String> ts = new TestSubscriber<>();

Observable.interval(200, TimeUnit.MILLISECONDS)
        .take(5)
        .map(i -> {
            return i + " value";
        }).subscribe(ts);

ts.awaitTerminalEvent();
ts.assertNoErrors();
ts.assertReceivedOnNext(Arrays.asList("0 value", "1 value",
                                "2 value", "3 value", "4 value"));
```

```java
TestSubscriber<String> ts = new TestSubscriber<>();

Observable.interval(200, TimeUnit.MILLISECONDS)
        .take(5)
        .map(i -> {
            return i + " value";
        }).subscribe(ts);

ts.awaitTerminalEvent();
ts.assertNoErrors();
ts.assertReceivedOnNext(Arrays.asList("0 value", "1 value",
                                  "2 value", "3 value", "4 value"));
```

```java
TestScheduler test = Schedulers.test();
TestSubscriber<String> ts = new TestSubscriber<>();

Observable.interval(200, TimeUnit.MILLISECONDS, test)
        .map(i -> {
            return i + " value";
        }).subscribe(ts);

test.advanceTimeBy(200, TimeUnit.MILLISECONDS);
ts.assertReceivedOnNext(Arrays.asList("0 value"));

test.advanceTimeTo(1000, TimeUnit.MILLISECONDS);
ts.assertReceivedOnNext(Arrays.asList("0 value", "1 value”,
                                      "2 value", "3 value", "4 value"));
```

```
TestScheduler test = Schedulers.test();
TestSubscriber<String> ts = new TestSubscriber<>();

Observable.interval(200, TimeUnit.MILLISECONDS, test)
        .map(i -> {
            return i + " value";
        }).subscribe(ts);

test.advanceTimeBy(200, TimeUnit.MILLISECONDS);
ts.assertReceivedOnNext(Arrays.asList("0 value"));

test.advanceTimeTo(1000, TimeUnit.MILLISECONDS);
ts.assertReceivedOnNext(Arrays.asList("0 value", "1 value”,
                              "2 value", "3 value", "4 value"));
```

```java
TestScheduler test = Schedulers.test();
TestSubscriber<String> ts = new TestSubscriber<>();

Observable.interval(200, TimeUnit.MILLISECONDS, test)
        .map(i -> {
            return i + " value";
        }).subscribe(ts);

test.advanceTimeBy(200, TimeUnit.MILLISECONDS);
ts.assertReceivedOnNext(Arrays.asList("0 value"));

test.advanceTimeTo(1000, TimeUnit.MILLISECONDS);
ts.assertReceivedOnNext(Arrays.asList("0 value", "1 value”,
                              "2 value", "3 value", "4 value"));
```

```java
TestScheduler test = Schedulers.test();
TestSubscriber<String> ts = new TestSubscriber<>();

Observable.interval(200, TimeUnit.MILLISECONDS, test)
        .map(i -> {
            return i + " value";
        }).subscribe(ts);

test.advanceTimeBy(200, TimeUnit.MILLISECONDS);
ts.assertReceivedOnNext(Arrays.asList("0 value"));

test.advanceTimeTo(1000, TimeUnit.MILLISECONDS);
ts.assertReceivedOnNext(Arrays.asList("0 value", "1 value”,
                              "2 value", "3 value", "4 value"));
```

```java
TestScheduler test = Schedulers.test();
TestSubscriber<String> ts = new TestSubscriber<>();

Observable.interval(200, TimeUnit.MILLISECONDS, test)
        .map(i -> {
            return i + " value";
        }).subscribe(ts);

test.advanceTimeBy(200, TimeUnit.MILLISECONDS);
ts.assertReceivedOnNext(Arrays.asList("0 value"));

test.advanceTimeTo(1000, TimeUnit.MILLISECONDS);
ts.assertReceivedOnNext(Arrays.asList("0 value", "1 value”,
                                 "2 value", "3 value", "4 value"));
```

```java
TestScheduler test = Schedulers.test();
TestSubscriber<String> ts = new TestSubscriber<>();

Observable.interval(200, TimeUnit.MILLISECONDS, test)
        .map(i -> {
            return i + " value";
        }).subscribe(ts);

test.advanceTimeBy(200, TimeUnit.MILLISECONDS);
ts.assertReceivedOnNext(Arrays.asList("0 value"));

test.advanceTimeTo(1000, TimeUnit.MILLISECONDS);
ts.assertReceivedOnNext(Arrays.asList("0 value", "1 value”,
                                      "2 value", "3 value", "4 value"));
```

```java
TestScheduler test = Schedulers.test();
TestSubscriber<String> ts = new TestSubscriber<>();

Observable.interval(200, TimeUnit.MILLISECONDS, test)
        .map(i -> {
            return i + " value";
        }).subscribe(ts);

test.advanceTimeBy(200, TimeUnit.MILLISECONDS);
ts.assertReceivedOnNext(Arrays.asList("0 value"));

test.advanceTimeTo(1000, TimeUnit.MILLISECONDS);
ts.assertReceivedOnNext(Arrays.asList("0 value", "1 value”,
                              "2 value", "3 value", "4 value"));
```

```java
TestScheduler test = Schedulers.test();
TestSubscriber<String> ts = new TestSubscriber<>();

Observable.interval(200, TimeUnit.MILLISECONDS, test)
        .map(i -> {
            return i + " value";
        }).subscribe(ts);

test.advanceTimeBy(200, TimeUnit.MILLISECONDS);
ts.assertReceivedOnNext(Arrays.asList("0 value"));

test.advanceTimeTo(1000, TimeUnit.MILLISECONDS);
ts.assertReceivedOnNext(Arrays.asList("0 value", "1 value”,
                              "2 value", "3 value", "4 value"));
```

Async Debugging is Hard

Ugly Stacktraces

```
java.lang.RuntimeException: failure!
    at jdbc.Test.lambda$19(Test.java:63)
    at jdbc.Test$$Lambda$8/888452645.call(Unknown Source)
    at rx.internal.operators.OperatorMap$1.onNext(OperatorMap.java:55)
    at rx.internal.operators.OperatorSubscribeOn$1$1$1.onNext(OperatorSubscribeOn.java:76)
    at rx.observables.AbstractOnSubscribe$SubscriptionState.accept(AbstractOnSubscribe.java:533)
    at rx.observables.AbstractOnSubscribe$SubscriptionProducer.doNext(AbstractOnSubscribe.java:367)
    at rx.observables.AbstractOnSubscribe$SubscriptionProducer.request(AbstractOnSubscribe.java:345)
    at rx.internal.operators.OperatorSubscribeOn$1$1$1$1.request(OperatorSubscribeOn.java:88)
    at rx.Subscriber.setProducer(Subscriber.java:177)
    at rx.Subscriber.setProducer(Subscriber.java:171)
    at rx.internal.operators.OperatorSubscribeOn$1$1$1.setProducer(OperatorSubscribeOn.java:81)
    at rx.observables.AbstractOnSubscribe.call(AbstractOnSubscribe.java:191)
    at rx.observables.AbstractOnSubscribe$LambdaOnSubscribe.call(AbstractOnSubscribe.java:274)
    at rx.Observable.unsafeSubscribe(Observable.java:7495)
    at rx.internal.operators.OperatorSubscribeOn$1$1.call(OperatorSubscribeOn.java:62)
    at rx.internal.schedulers.ScheduledAction.run(ScheduledAction.java:55)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
    at java.util.concurrent.FutureTask.run(FutureTask.java:266)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$201(ScheduledThreadPoolExecutor.java:180)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:293)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
Caused by: rx.exceptions.OnErrorThrowable$OnNextValue: OnError while emitting onNext value: jdbc.Product.class
    at rx.exceptions.OnErrorThrowable.addValueAsLastCause(OnErrorThrowable.java:98)
    at rx.internal.operators.OperatorMap$1.onNext(OperatorMap.java:58)
    ... 20 more
```

Ugly Stacktraces

```
java.lang.RuntimeException: failure!
    at jdbc.Test.lambda$19(Test.java:63)
    at jdbc.Test$$Lambda$8/888452645.call(Unknown Source)
    at rx.internal.operators.OperatorMap$1.onNext(OperatorMap.java:55)
    at rx.internal.operators.OperatorSubscribeOn$1$1$1.onNext(OperatorSubscribeOn.java:76)
    at rx.observables.AbstractOnSubscribe$SubscriptionState.accept(AbstractOnSubscribe.java:533)
    at rx.observables.AbstractOnSubscribe$SubscriptionProducer.doNext(AbstractOnSubscribe.java:367)
    at rx.observables.AbstractOnSubscribe$SubscriptionProducer.request(AbstractOnSubscribe.java:345)
    at rx.internal.operators.OperatorSubscribeOn$1$1$1$1.request(OperatorSubscribeOn.java:88)
    at rx.Subscriber.setProducer(Subscriber.java:177)
    at rx.Subscriber.setProducer(Subscriber.java:171)
    at rx.internal.operators.OperatorSubscribeOn$1$1$1.setProducer(OperatorSubscribeOn.java:81)
    at rx.observables.AbstractOnSubscribe.call(AbstractOnSubscribe.java:191)
    at rx.observables.AbstractOnSubscribe$LambdaOnSubscribe.call(AbstractOnSubscribe.java:274)
    at rx.Observable.unsafeSubscribe(Observable.java:7495)
    at rx.internal.operators.OperatorSubscribeOn$1$1.call(OperatorSubscribeOn.java:62)
    at rx.internal.schedulers.ScheduledAction.run(ScheduledAction.java:55)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
    at java.util.concurrent.FutureTask.run(FutureTask.java:266)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$201(ScheduledThreadPoolExecutor.java:180)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:293)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
Caused by: rx.exceptions.OnErrorThrowable$OnNextValue: OnError while emitting onNext value: jdbc.Product.class
    at rx.exceptions.OnErrorThrowable.addValueAsLastCause(OnErrorThrowable.java:98)
    at rx.internal.operators.OperatorMap$1.onNext(OperatorMap.java:58)
    ... 20 more
```

Missing Async Call Chain

```
java.lang.RuntimeException: failure!
    at jdbc.Test.lambda$19(Test.java:63)
    at jdbc.Test$$Lambda$8/888452645.call(Unknown Source)
    at rx.internal.operators.OperatorMap$1.onNext(OperatorMap.java:55)
    at rx.internal.operators.OperatorSubscribeOn$1$1$1.onNext(OperatorSubscribeOn.java:76)
    at rx.observables.AbstractOnSubscribe$SubscriptionState.accept(AbstractOnSubscribe.java:533)
    at rx.observables.AbstractOnSubscribe$SubscriptionProducer.doNext(AbstractOnSubscribe.java:367)
    at rx.observables.AbstractOnSubscribe$SubscriptionProducer.request(AbstractOnSubscribe.java:345)
    at rx.internal.operators.OperatorSubscribeOn$1$1$1$1.request(OperatorSubscribeOn.java:88)
    at rx.Subscriber.setProducer(Subscriber.java:177)
    at rx.Subscriber.setProducer(Subscriber.java:171)
    at rx.internal.operators.OperatorSubscribeOn$1$1$1.setProducer(OperatorSubscribeOn.java:81)
    at rx.observables.AbstractOnSubscribe.call(AbstractOnSubscribe.java:191)
    at rx.observables.AbstractOnSubscribe$LambdaOnSubscribe.call(AbstractOnSubscribe.java:274)
    at rx.Observable.unsafeSubscribe(Observable.java:7495)
    at rx.internal.operators.OperatorSubscribeOn$1$1.call(OperatorSubscribeOn.java:62)
    at rx.internal.schedulers.ScheduledAction.run(ScheduledAction.java:55)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
    at java.util.concurrent.FutureTask.run(FutureTask.java:266)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$201(ScheduledThreadPoolExecutor.java:180)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:293)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
Caused by: rx.exceptions.OnErrorThrowable$OnNextValue: OnError while emitting onNext value: jdbc.Product.class
    at rx.exceptions.OnErrorThrowable.addValueAsLastCause(OnErrorThrowable.java:98)
    at rx.internal.operators.OperatorMap$1.onNext(OperatorMap.java:58)
    ... 20 more
```

Stack Pollution

```
java.lang.RuntimeException: failure!
    at jdbc.Test.lambda$19(Test.java:63)
    at jdbc.Test$$Lambda$8/888452645.call(Unknown Source)
    at rx.internal.operators.OperatorMap$1.onNext(OperatorMap.java:55)
    at rx.internal.operators.OperatorSubscribeOn$1$1$1.onNext(OperatorSubscribeOn.java:76)
    at rx.observables.AbstractOnSubscribe$SubscriptionState.accept(AbstractOnSubscribe.java:533)
    at rx.observables.AbstractOnSubscribe$SubscriptionProducer.doNext(AbstractOnSubscribe.java:367)
    at rx.observables.AbstractOnSubscribe$SubscriptionProducer.request(AbstractOnSubscribe.java:345)
    at rx.internal.operators.OperatorSubscribeOn$1$1$1$1.request(OperatorSubscribeOn.java:88)
    at rx.Subscriber.setProducer(Subscriber.java:177)
    at rx.Subscriber.setProducer(Subscriber.java:171)
    at rx.internal.operators.OperatorSubscribeOn$1$1$1.setProducer(OperatorSubscribeOn.java:81)
    at rx.observables.AbstractOnSubscribe.call(AbstractOnSubscribe.java:191)
    at rx.observables.AbstractOnSubscribe$LambdaOnSubscribe.call(AbstractOnSubscribe.java:274)
    at rx.Observable.unsafeSubscribe(Observable.java:7495)
    at rx.internal.operators.OperatorSubscribeOn$1$1.call(OperatorSubscribeOn.java:62)
    at rx.internal.schedulers.ScheduledAction.run(ScheduledAction.java:55)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
    at java.util.concurrent.FutureTask.run(FutureTask.java:266)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$201(ScheduledThreadPoolExecutor.java:180)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:293)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
Caused by: rx.exceptions.OnErrorThrowable$OnNextValue: OnError while emitting onNext value: jdbc.Product.class
    at rx.exceptions.OnErrorThrowable.addValueAsLastCause(OnErrorThrowable.java:98)
    at rx.internal.operators.OperatorMap$1.onNext(OperatorMap.java:58)
    ... 20 more
```

RxJava Tries

```
java.lang.RuntimeException: failure!
    at jdbc.Test.lambda$19(Test.java:63)
    at jdbc.Test$$Lambda$8/888452645.call(Unknown Source)
    at rx.internal.operators.OperatorMap$1.onNext(OperatorMap.java:55)
    at rx.internal.operators.OperatorSubscribeOn$1$1$1.onNext(OperatorSubscribeOn.java:76)
    at rx.observables.AbstractOnSubscribe$SubscriptionState.accept(AbstractOnSubscribe.java:533)
    at rx.observables.AbstractOnSubscribe$SubscriptionProducer.doNext(AbstractOnSubscribe.java:367)
    at rx.observables.AbstractOnSubscribe$SubscriptionProducer.request(AbstractOnSubscribe.java:345)
    at rx.internal.operators.OperatorSubscribeOn$1$1$1$1.request(OperatorSubscribeOn.java:88)
    at rx.Subscriber.setProducer(Subscriber.java:177)
    at rx.Subscriber.setProducer(Subscriber.java:171)
    at rx.internal.operators.OperatorSubscribeOn$1$1$1.setProducer(OperatorSubscribeOn.java:81)
    at rx.observables.AbstractOnSubscribe.call(AbstractOnSubscribe.java:191)
    at rx.observables.AbstractOnSubscribe$LambdaOnSubscribe.call(AbstractOnSubscribe.java:274)
    at rx.Observable.unsafeSubscribe(Observable.java:7495)
    at rx.internal.operators.OperatorSubscribeOn$1$1.call(OperatorSubscribeOn.java:62)
    at rx.internal.schedulers.ScheduledAction.run(ScheduledAction.java:55)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
    at java.util.concurrent.FutureTask.run(FutureTask.java:266)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$201(ScheduledThreadPoolExecutor.java:180)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:293)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
Caused by: rx.exceptions.OnErrorThrowable$OnNextValue: OnError while emitting onNext value: jdbc.Product.class
    at rx.exceptions.OnErrorThrowable.addValueAsLastCause(OnErrorThrowable.java:98)
    at rx.internal.operators.OperatorMap$1.onNext(OperatorMap.java:58)
    ... 20 more
```

Stepping Severely **Limited**

```java
        getOrdersAsync(1)
                .limit(1)
                .flatMap(o -> {
                    return getProductsAsync(o)
                            .flatMap(p -> {
                                return getShippingStatusAsync(o, p);
                            });
                })
                .toBlocking()
                .forEach(s -> System.out.println(Thread.currentThread() + " [Async] Shipping Status: " + s));
    }

```

```java
        getOrdersAsync(1)
                .limit(1)
                .flatMap(o -> {
                    return getProductsAsync(o)
                            .flatMap(p -> {
                                return getShippingStatusAsync(o, p);
                            });
                })
                .toBlocking()
                .forEach(s -> System.out.println(Thread.currentThread() + " [Async] Shipping Status: " + s));
```

Stepping Severely Limited

```java
        getOrdersAsync(1)
                .limit(1)
                .flatMap(o -> {
                    return getProductsAsync(o)
                            .flatMap(p -> {
                                return getShippingStatusAsync(o, p);
                            });
                })
                .toBlocking()
                .forEach(s -> System.out.println(Thread.currentThread() + " [Async] Shipping Status: " + s));
    }
```

```java
        getOrdersAsync(1)
                .limit(1)
                .flatMap(o -> {
                    return getProductsAsync(o)
                            .flatMap(p -> {
                                return getShippingStatusAsync(o, p);
                            });
                })
                .toBlocking()
                .forEach(s -> System.out.println(Thread.currentThread() + " [Async] Shipping Status: " + s));
```

Stepping Severely Limited

```java
        getOrdersAsync(1)
                .limit(1)
                .flatMap(o -> {
                    return getProductsAsync(o)
                            .flatMap(p -> {
                                return getShippingStatusAsync(o, p);
                            });
                })
                .toBlocking()
                .forEach(s -> System.out.println(Thread.currentThread() + " [Async] Shipping Status: " + s));
    }
```

```java
        getOrdersAsync(1)
                .limit(1)
                .flatMap(o -> {
                    return getProductsAsync(o)
                            .flatMap(p -> {
                                return getShippingStatusAsync(o, p);
                            });
                })
                .toBlocking()
                .forEach(s -> System.out.println(Thread.currentThread() + " [Async] Shipping Status: " + s));
```

Debugging via Logging

```
orders.flatMap(o -> {
    return getProductsAsync(o)
            .doOnNext(t -> System.out.println("product: " + t))
            .flatMap(p -> {
                return getShippingStatusAsync(o, p)
                        .map(s -> {
                            return s;
                        });
            });
})
```

Debugging via Logging

```
orders.flatMap(o -> {
    return getProductsAsync(o)
        .doOnNext(t -> System.out.println("product: " + t))
        .flatMap(p -> {
            return getShippingStatusAsync(o, p)
                .map(s -> {
                    return s;
                });
        });
})
```

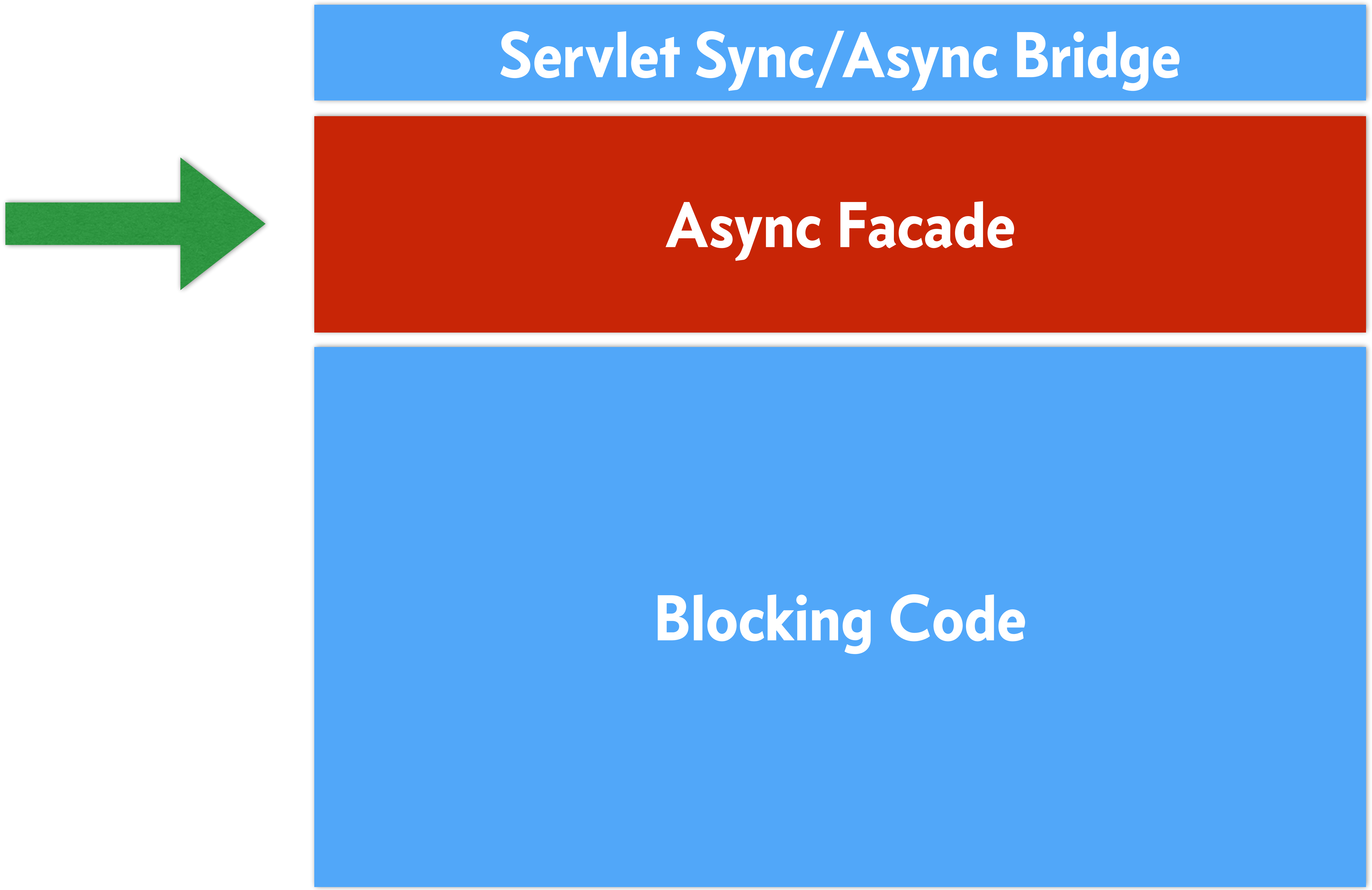
:-(

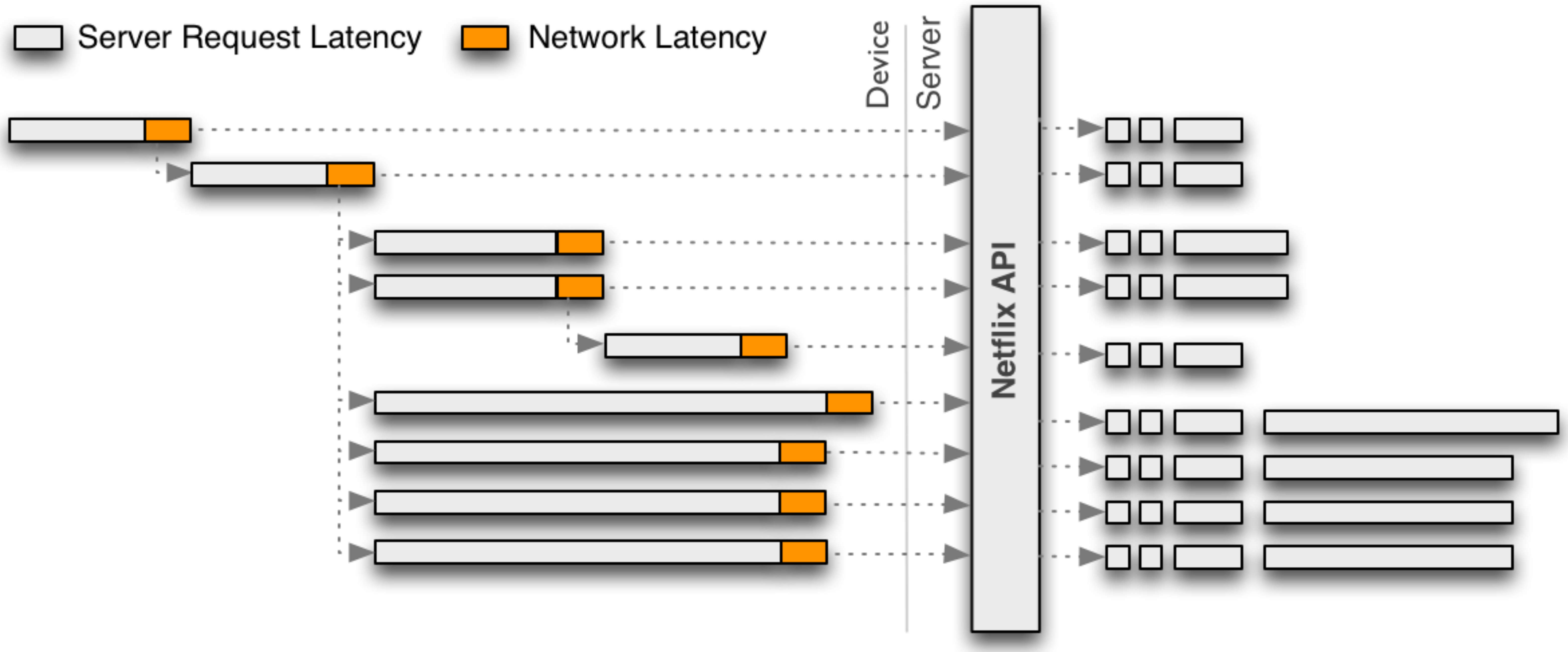
Async Debugging is Hard

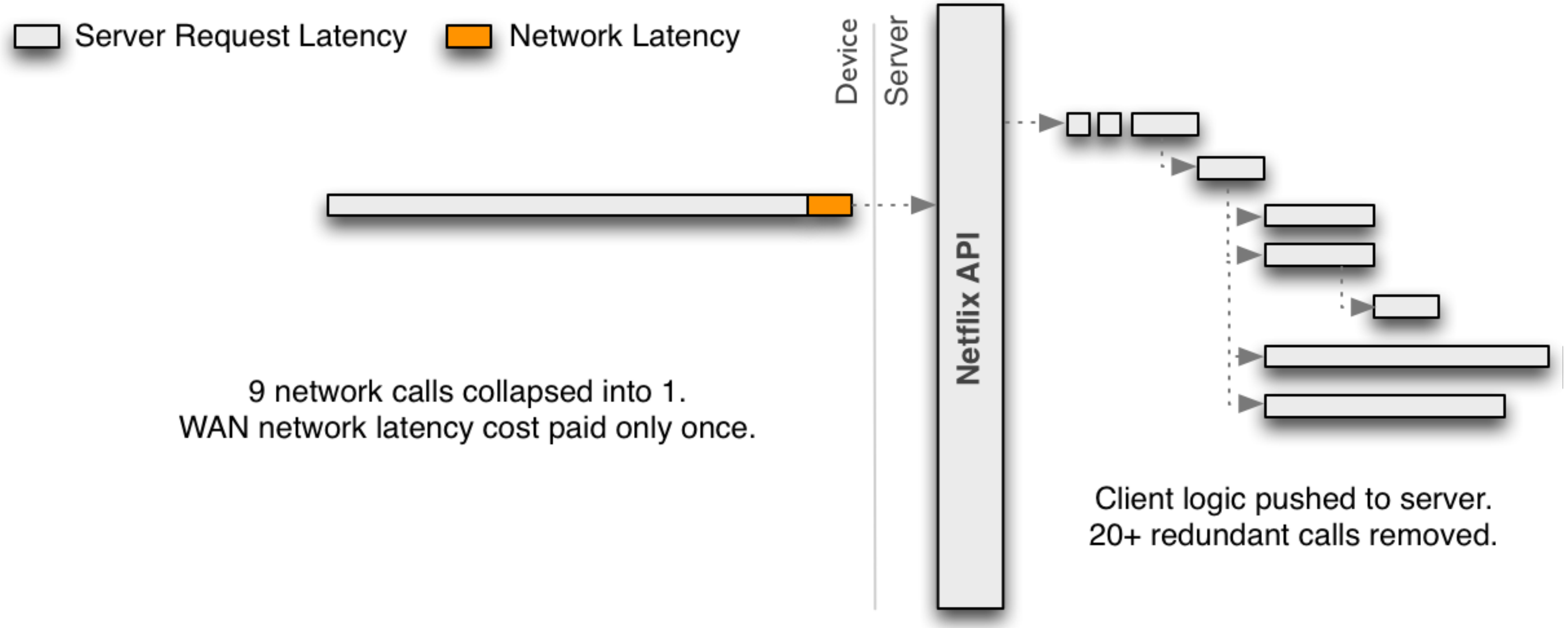
… a work in progress …


How about once all done?

Observable APIs Worked Well




Server Request Latency
Network Latency
Device
Server
Netflix API

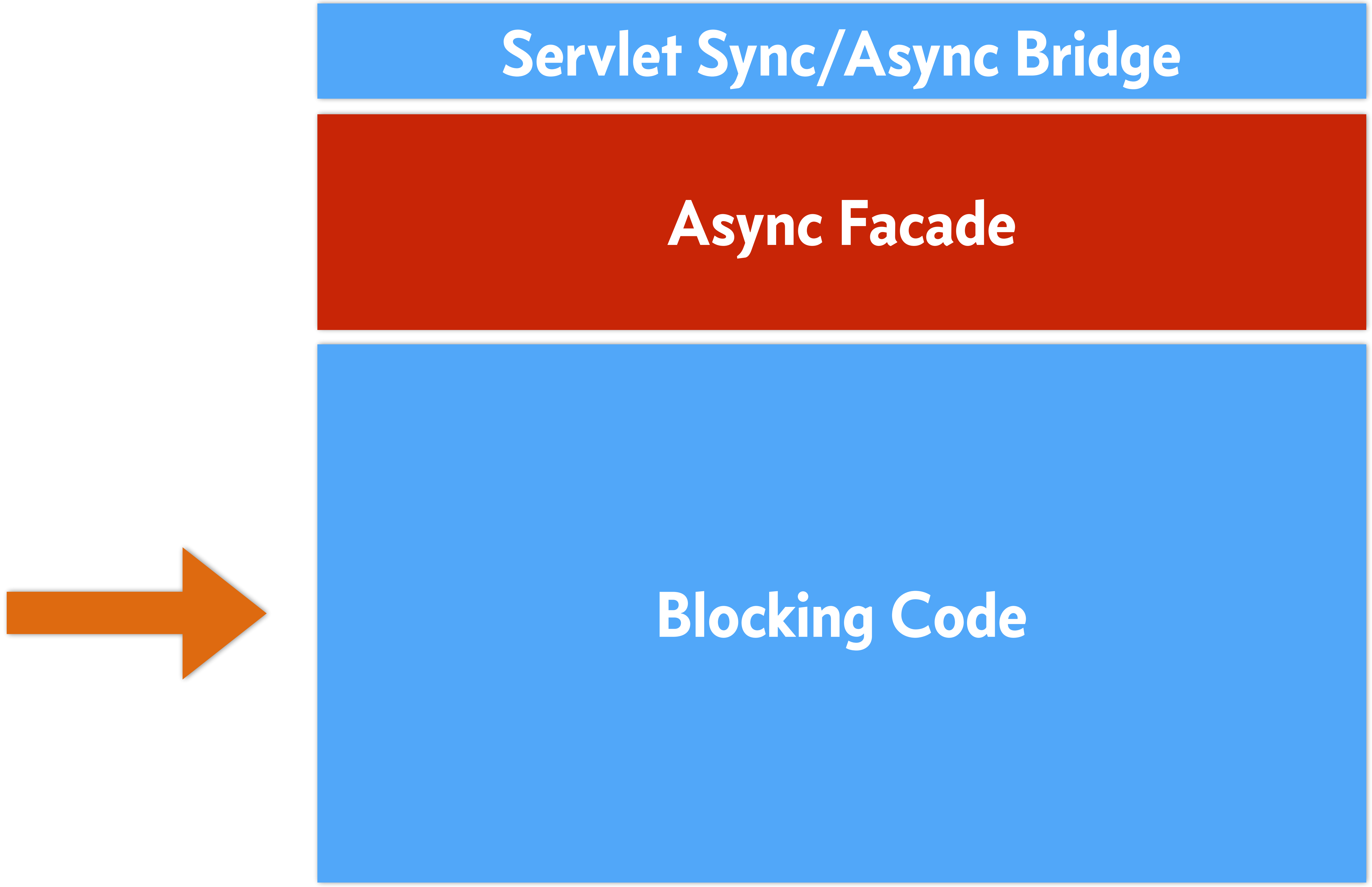

Server Request Latency
Network Latency
Device
Server
Netflix API
9 network calls collapsed into 1.
WAN network latency cost paid only once.
Client logic pushed to server.
20+ redundant calls removed.

REPLACED THE BLOCKING API …

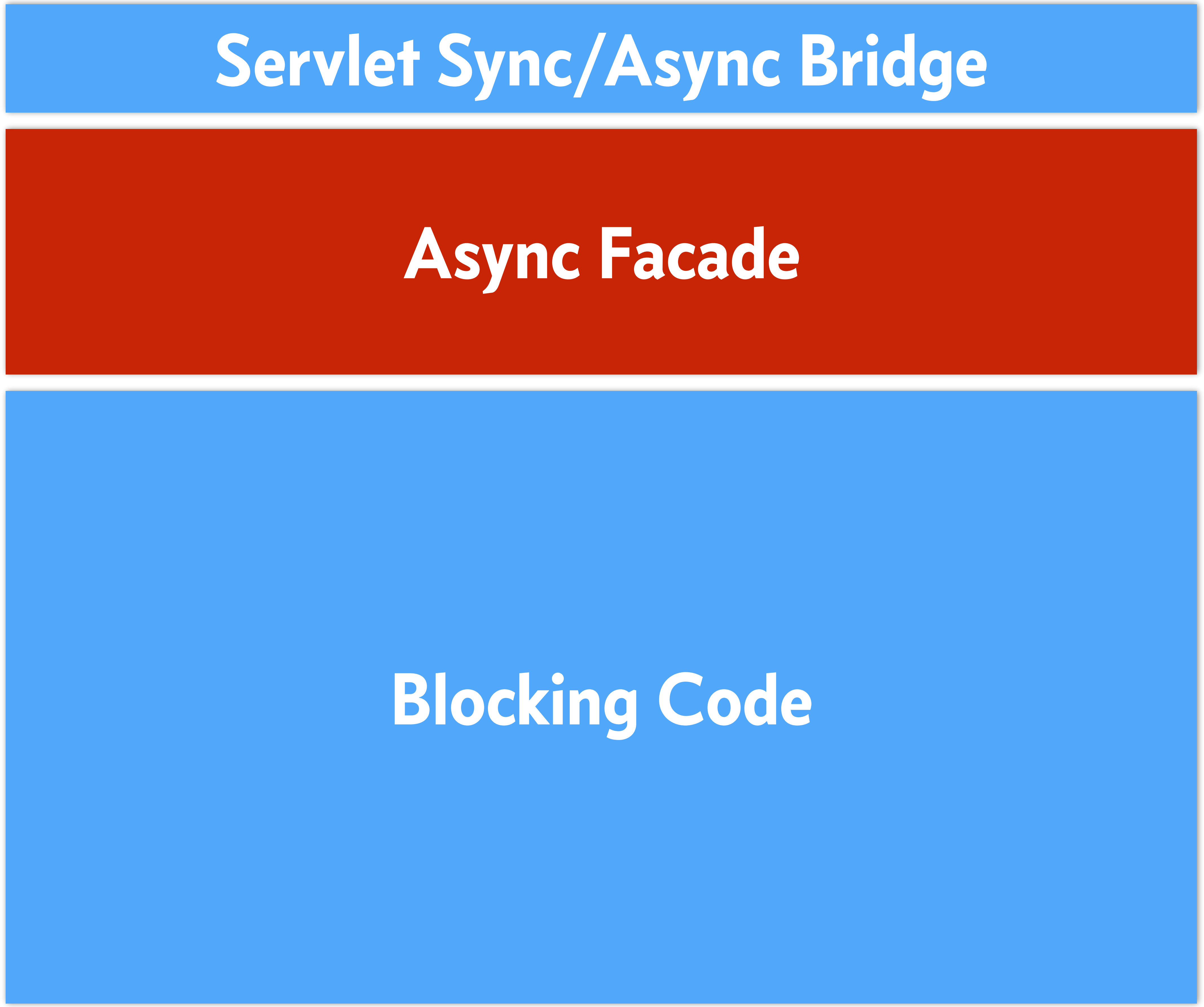
```
class VideoService {
   def VideoList getPersonalizedListOfMovies(userId);
   def VideoBookmark getBookmark(userId, videoId);
   def VideoRating getRating(userId, videoId);
   def VideoMetadata getMetadata(videoId);
}
```

… WITH AN OBSERVABLE API:

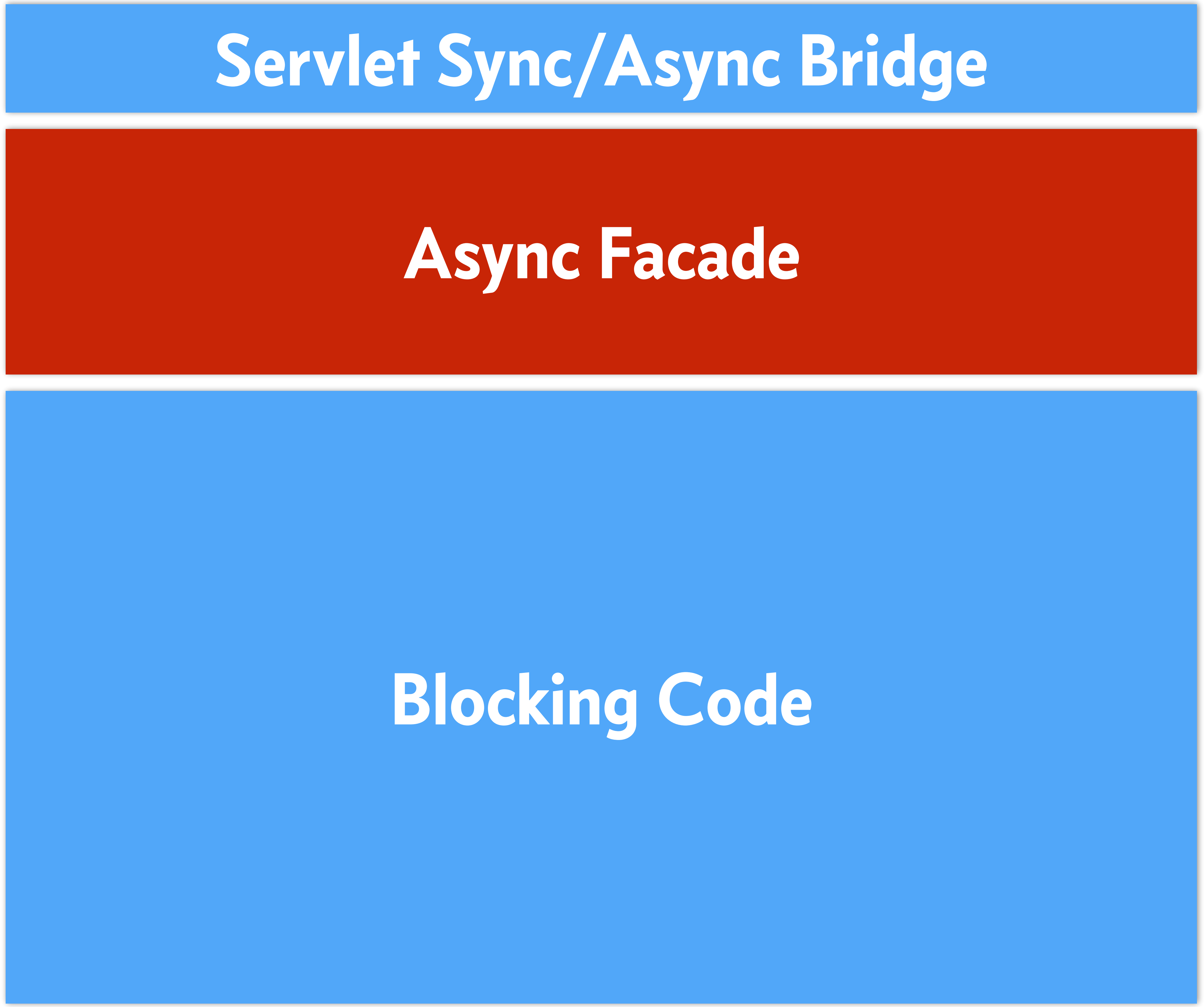
```
class VideoService {
   def Observable<VideoList> getPersonalizedListOfMovies(userId);
   def Observable<VideoBookmark> getBookmark(userId, videoId);
   def Observable<VideoRating> getRating(userId, videoId);
   def Observable<VideoMetadata> getMetadata(videoId);
}
```



Challenges Mixing Approaches



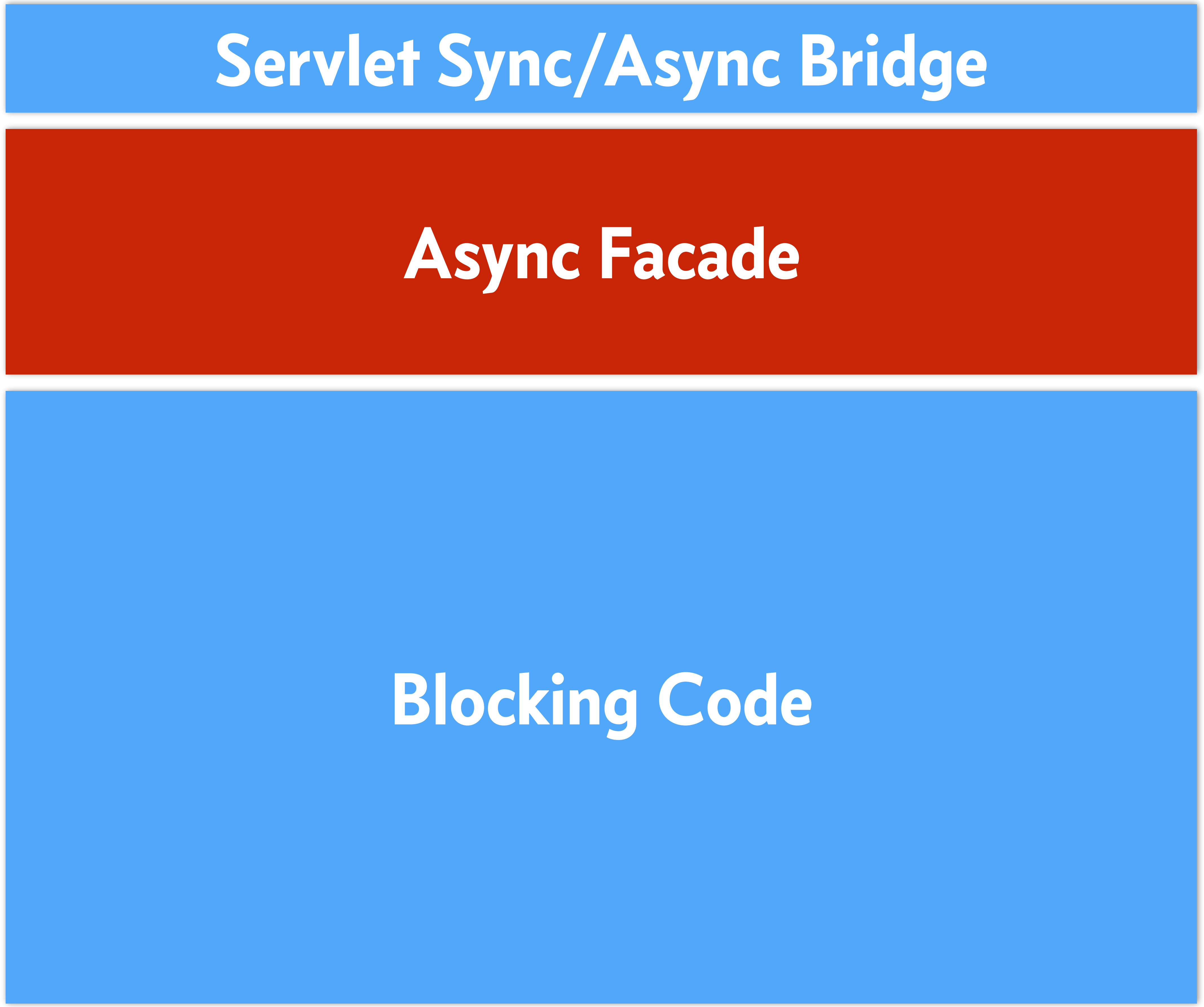
too many threads



hard to tune

&

shed load



easy to block
async code

concurrent →

```
getOrdersAsync(1)
        .flatMap(o -> {
            return getProductsAsync(o)
                        .flatMap(p -> {
                            return getShippingStatusAsync(o, p);
                        });
        });
```

sequential →

```
getOrdersAsync(1)
.flatMap(o -> {
    return getProductsAsync(o)
                .map(p -> {
                    return getShippingStatus(o, p);
                });
});
```

concurrent →

```
getOrdersAsync(1)
        .flatMap(o -> {
            return getProductsAsync(o)
                        .flatMap(p -> {
                            return getShippingStatusAsync(o, p);
                        });
        });
```

sequential →

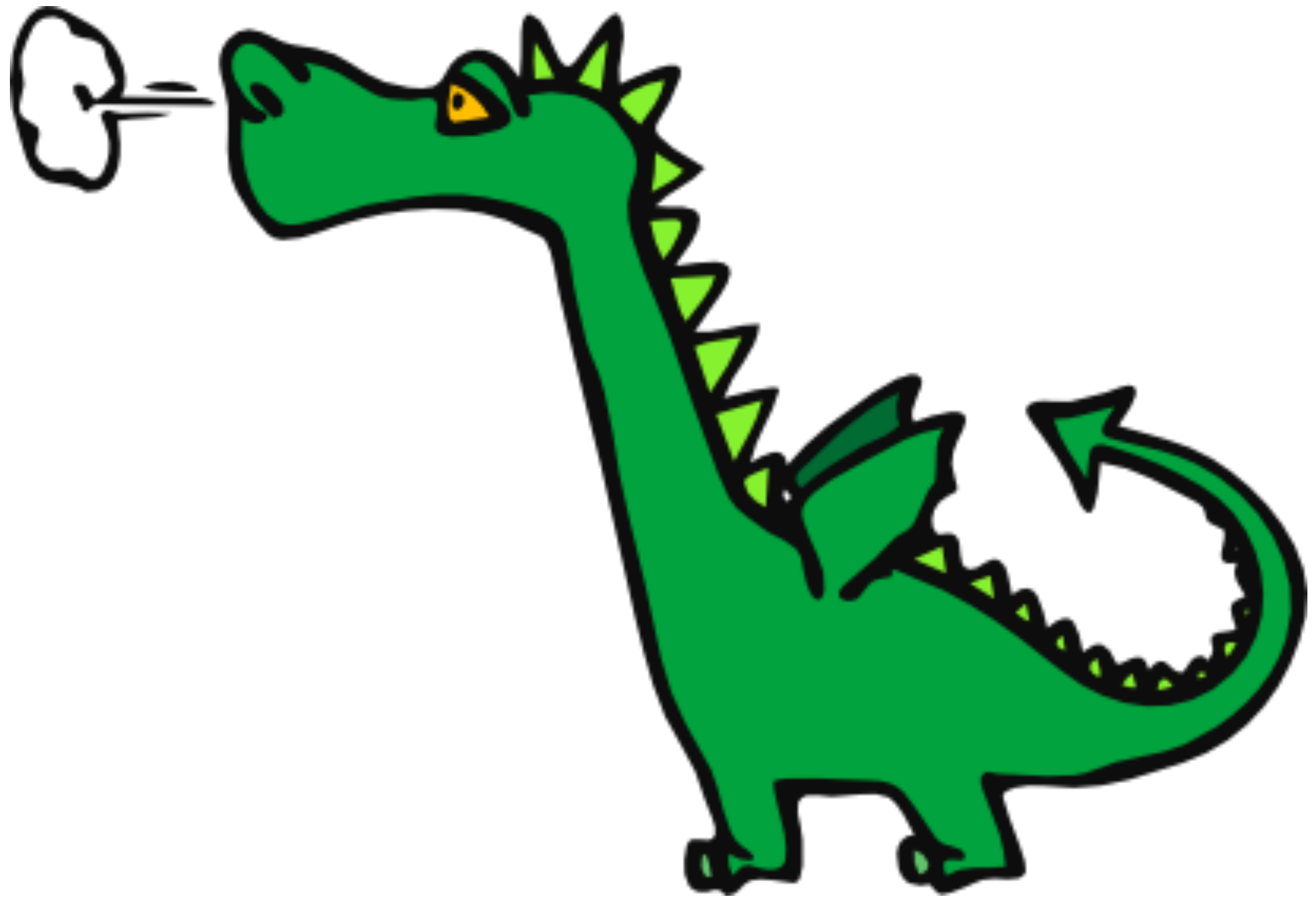
```
getOrdersAsync(1)
.flatMap(o -> {
    return getProductsAsync(o)
            .map(p -> {
                return getShippingStatus(o, p);
            });
});
```

Async Facade only has Observable APIs

```
getOrdersAsync(1)
        .flatMap(o -> {
            return getProductsAsync(o)
                    .flatMap(p -> {
                        return getShippingStatusAsync(o, p);
                    });
        });
```

```
getOrdersAsync(1)
.flatMap(o -> {
    return getProductsAsync(o)
            .map(p -> {
                return ~~getShippingStatus~~(o, p);
            });
});
```

So what about going "**fully** reactive"?



“cause it’s better”

“cause it’s better”

:-/

“it’s worse”

“it doesn’t matter”

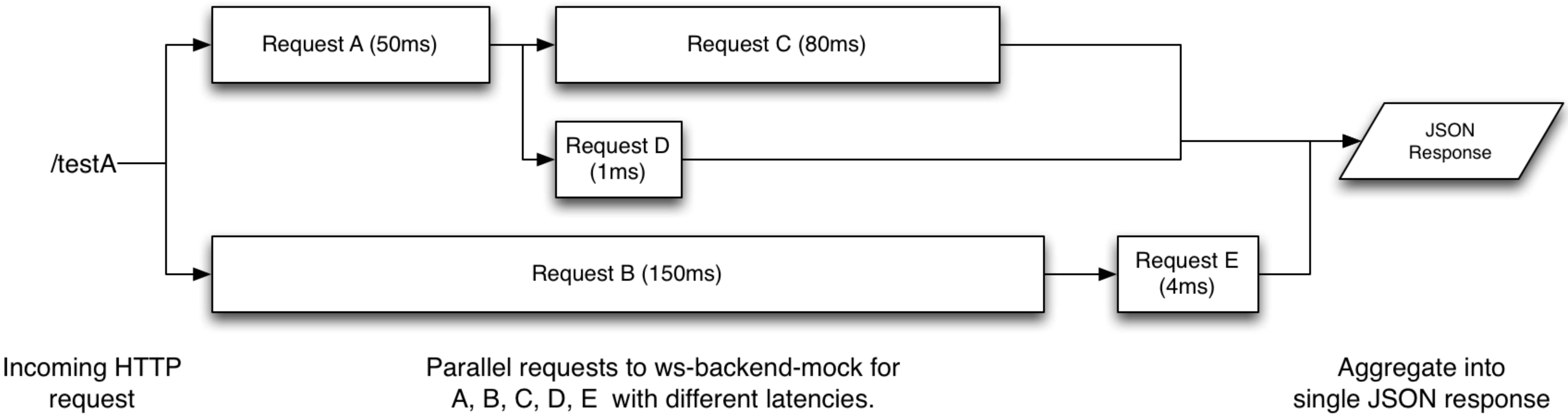
green threads, native threads, fibers, actors, event loops, theoretically equivalent, OS kernels, etc, etc, etc …

C10k and beyond

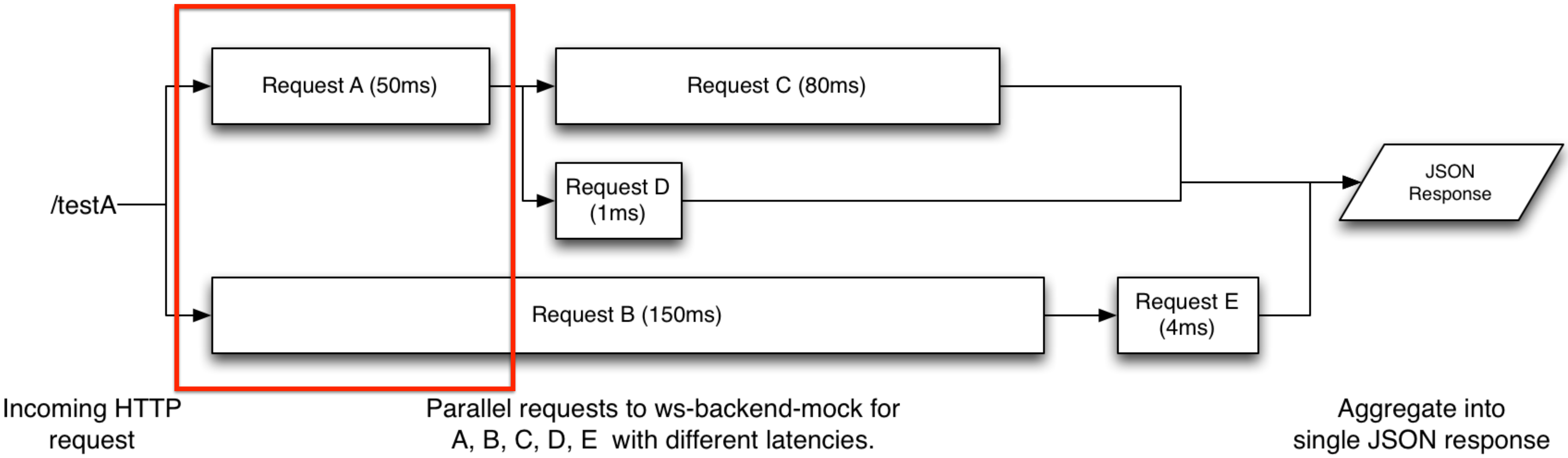
What to **bet future** on?

What to **bet future** on?

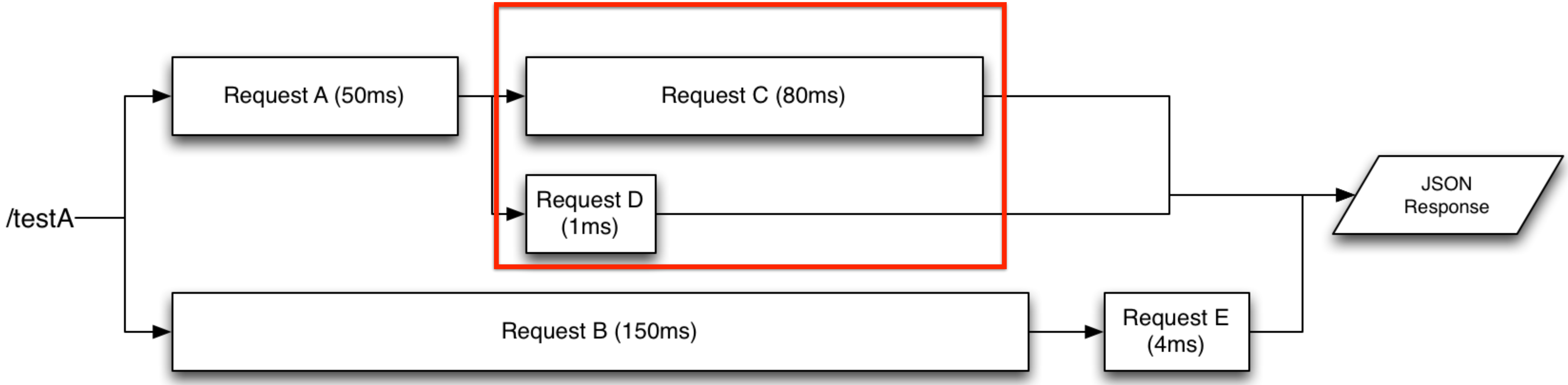
WSPerfLab: https://github.com/Netflix-Skunkworks/WSPerfLab



WSPerfLab: https://github.com/Netflix-Skunkworks/WSPerfLab



WSPerfLab: https://github.com/Netflix-Skunkworks/WSPerfLab

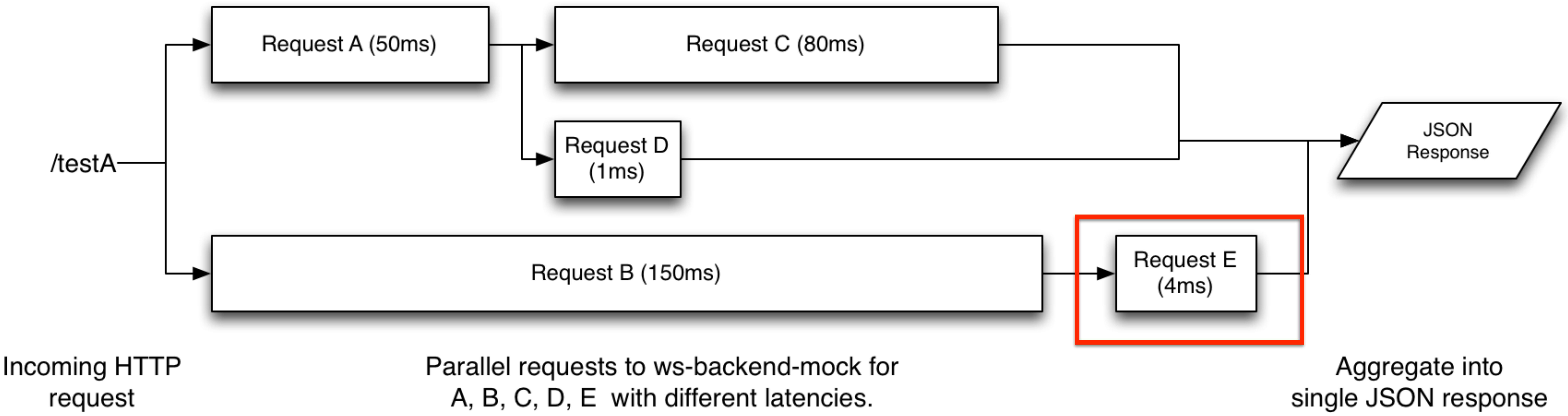


Incoming HTTP request

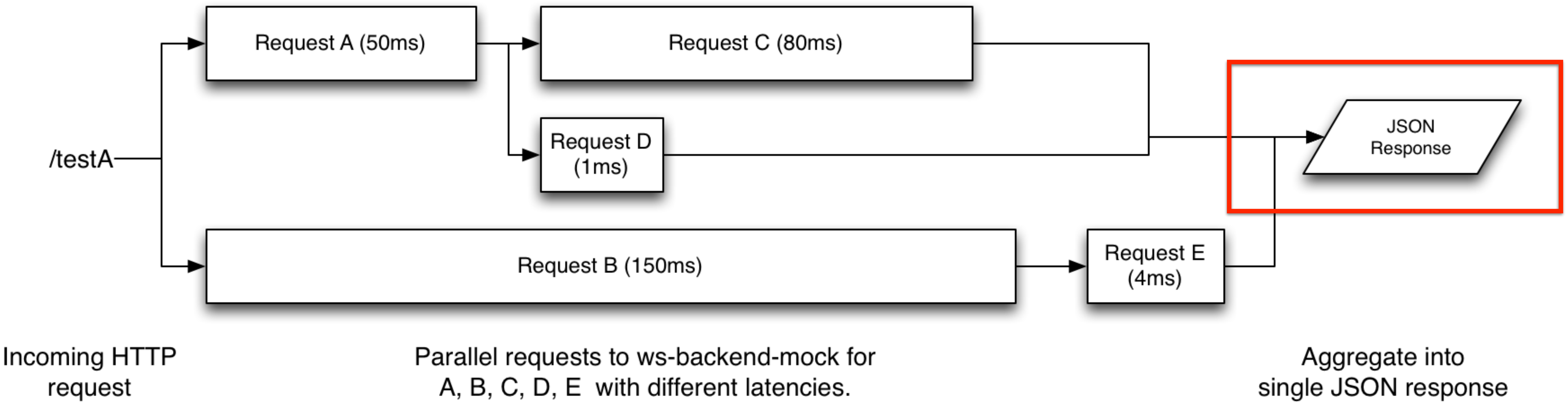
Parallel requests to ws-backend-mock for A, B, C, D, E with different latencies.

Aggregate into single JSON response

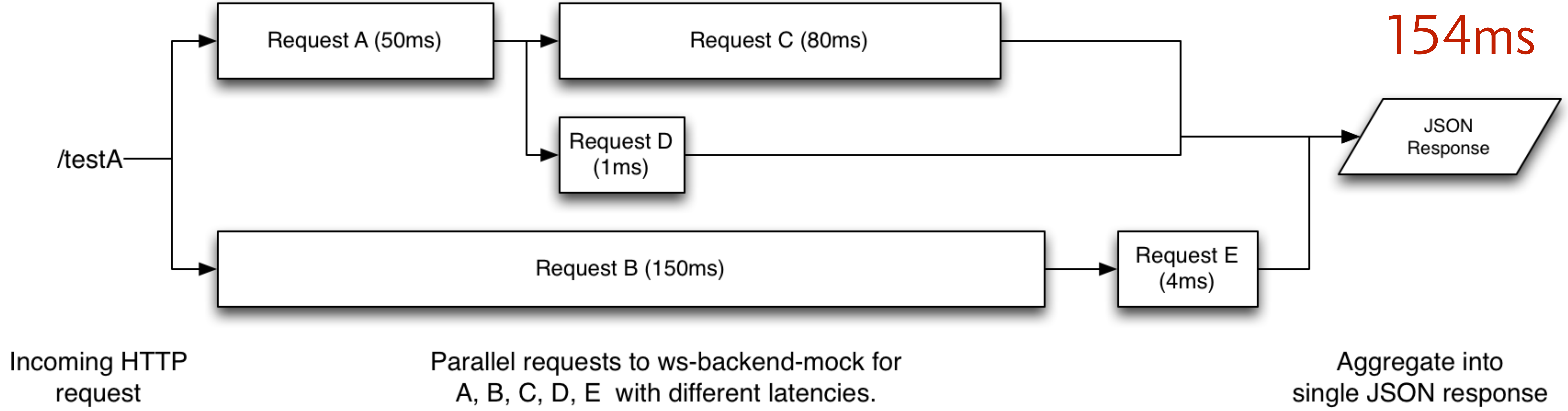
WSPerfLab: https://github.com/Netflix-Skunkworks/WSPerfLab

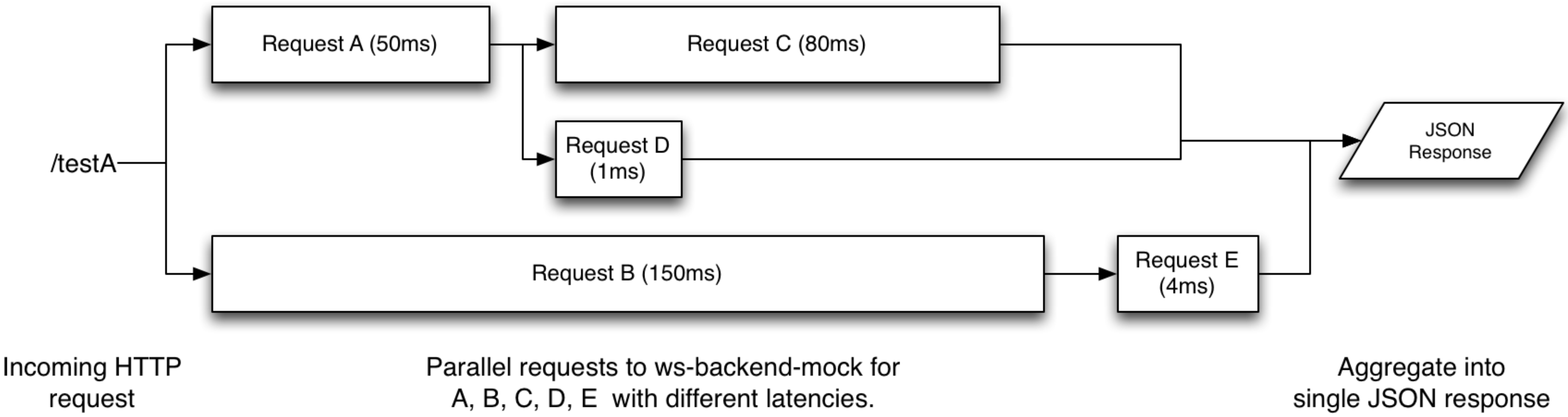


WSPerfLab: https://github.com/Netflix-Skunkworks/WSPerfLab



WSPerfLab: https://github.com/Netflix-Skunkworks/WSPerfLab

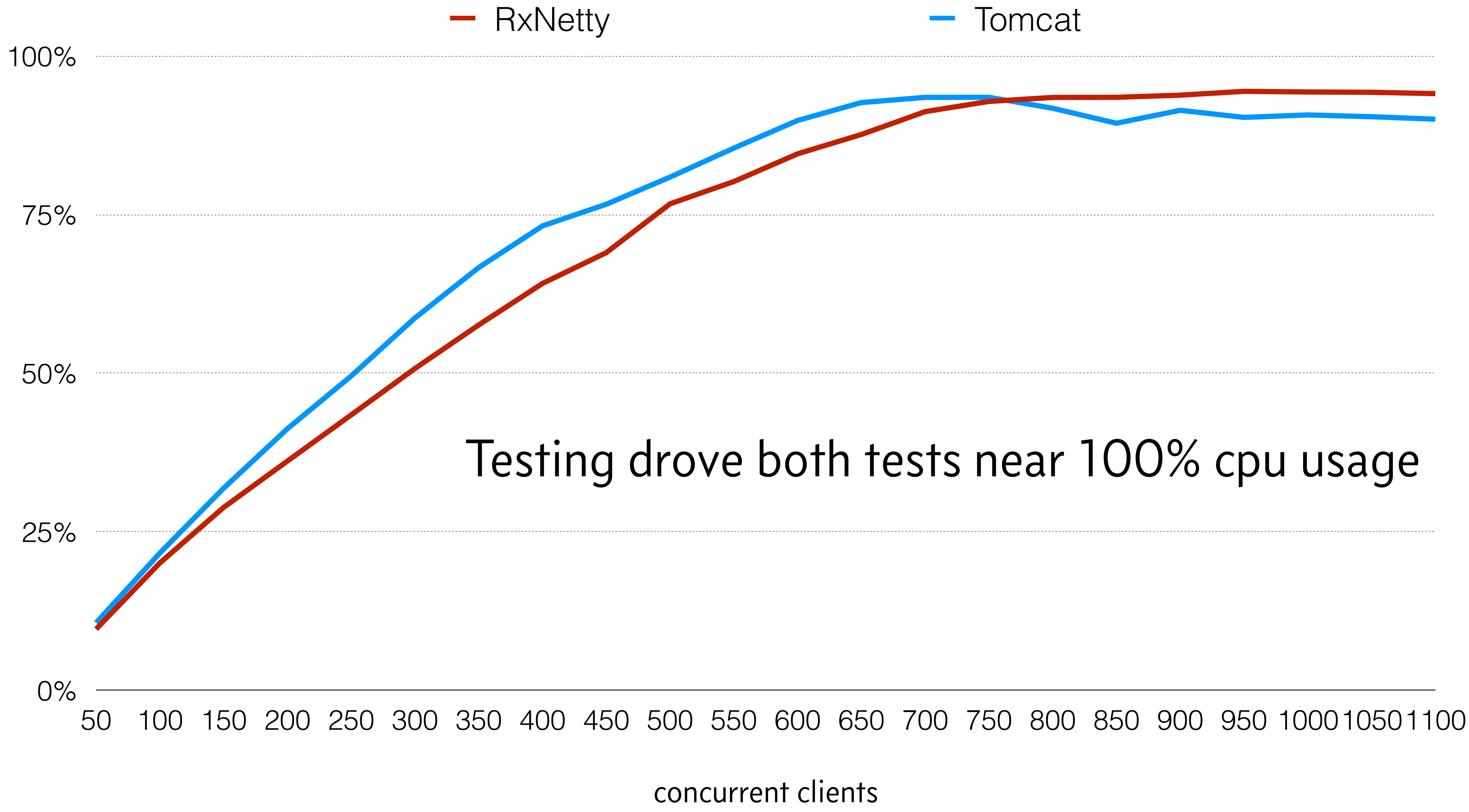




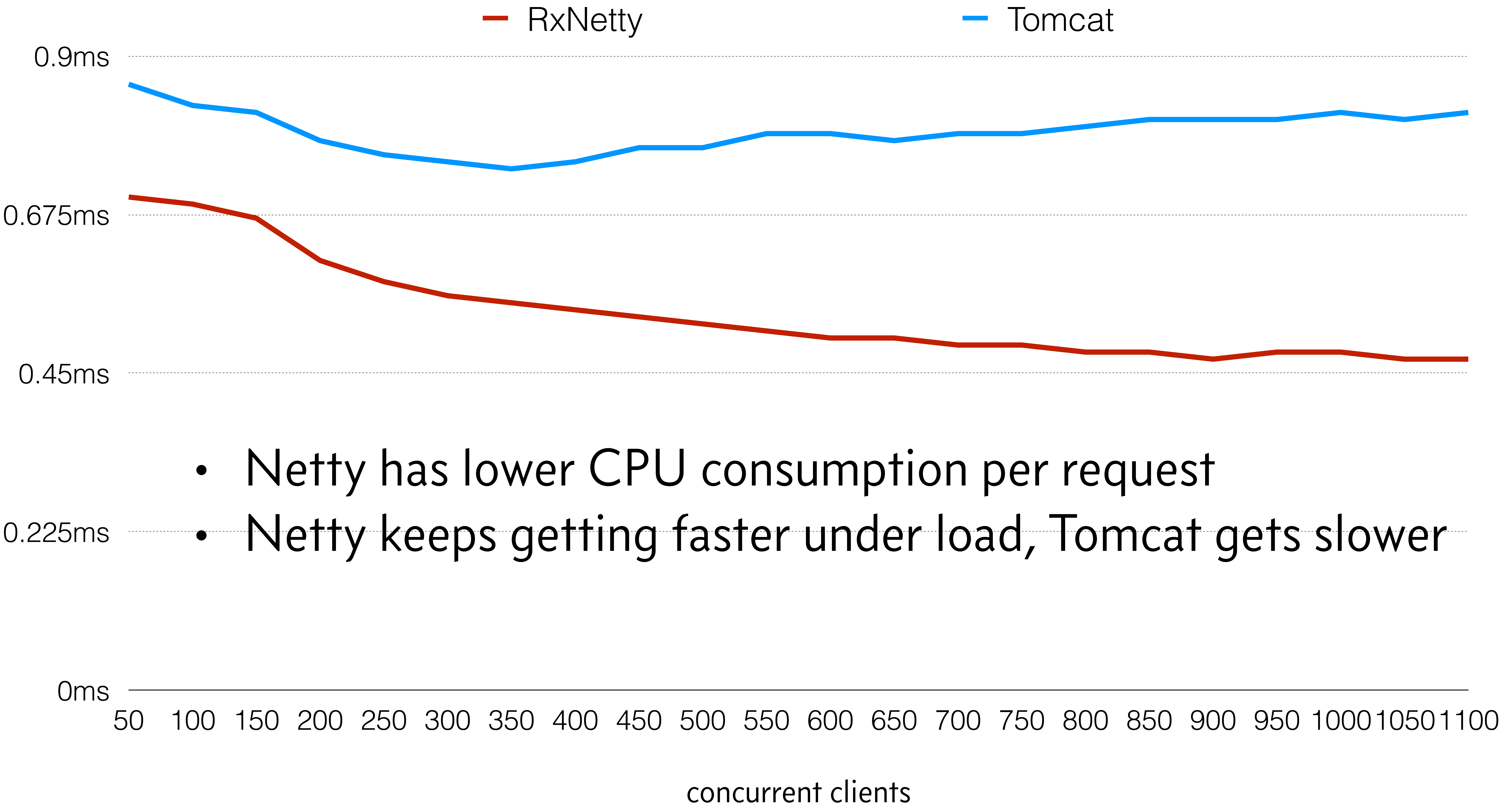
Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz: 4 cores, 1 thread per core
OpenJDK 8 with frame pointer patch

RxNetty 0.4.8 with Netty 4.0.25.Final vs Tomcat 7.0.45

CPU Usage in Percent

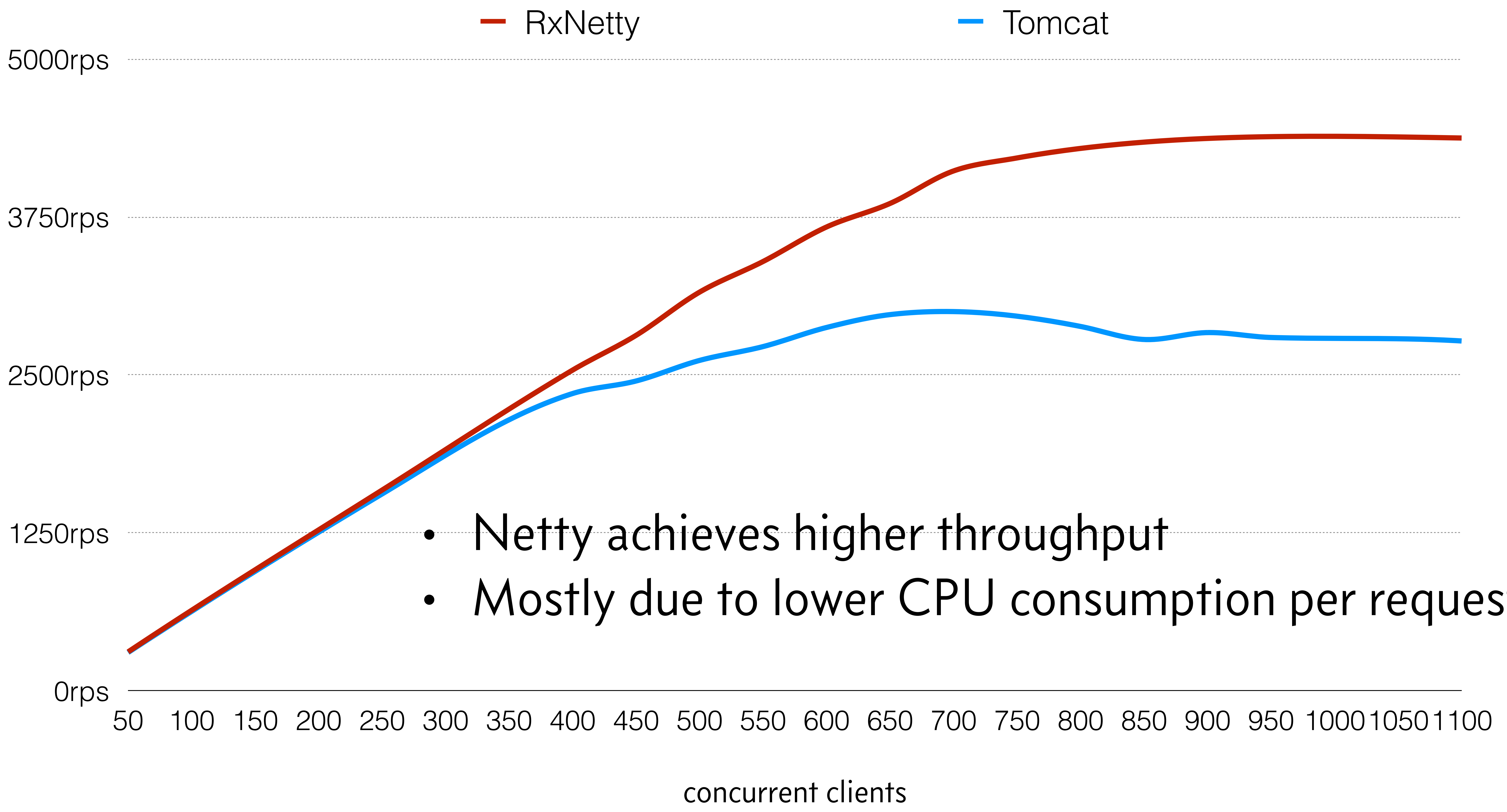


CPU Consumption per Request



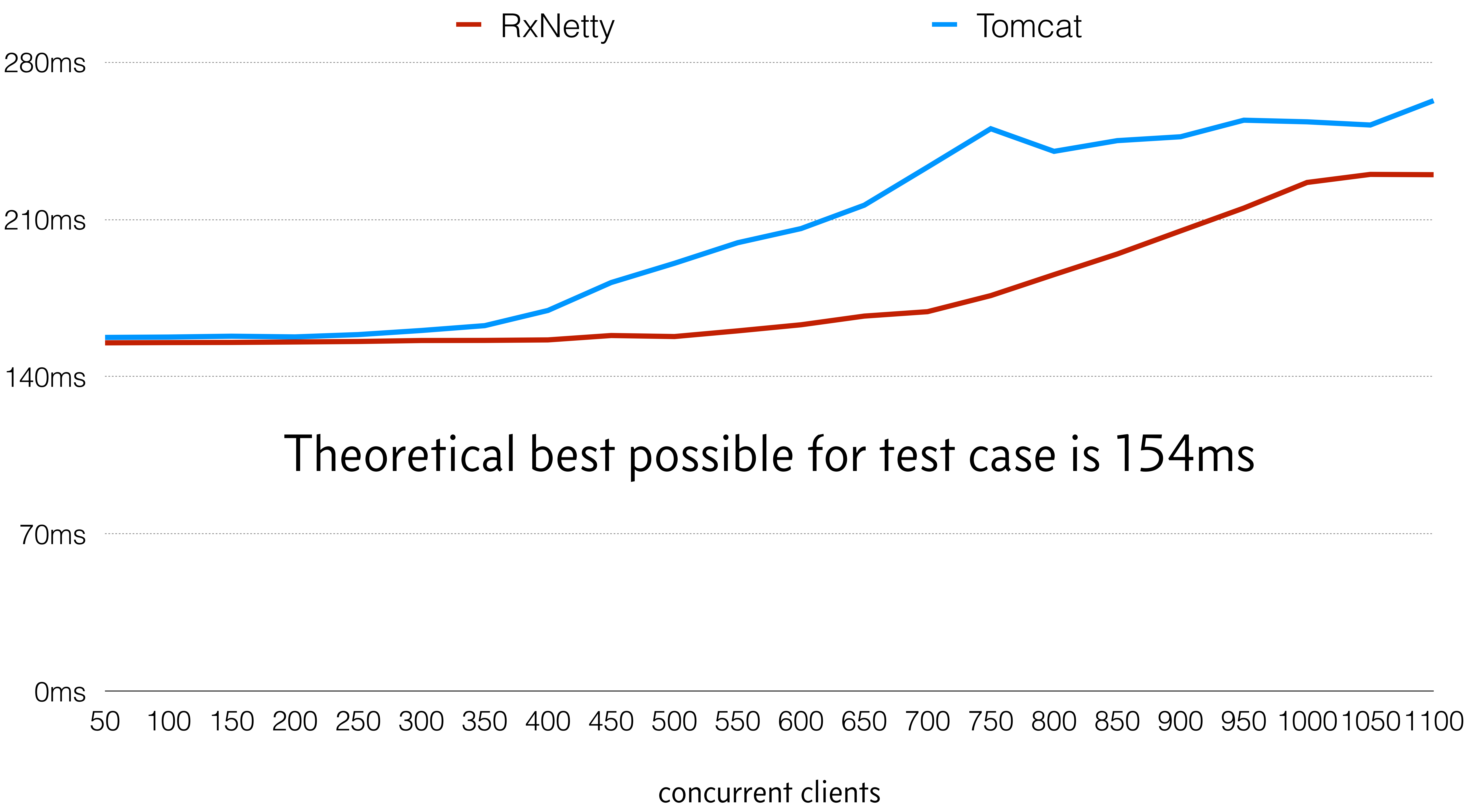
- Netty has lower CPU consumption per request
- Netty keeps getting faster under load, Tomcat gets slower

Throughput

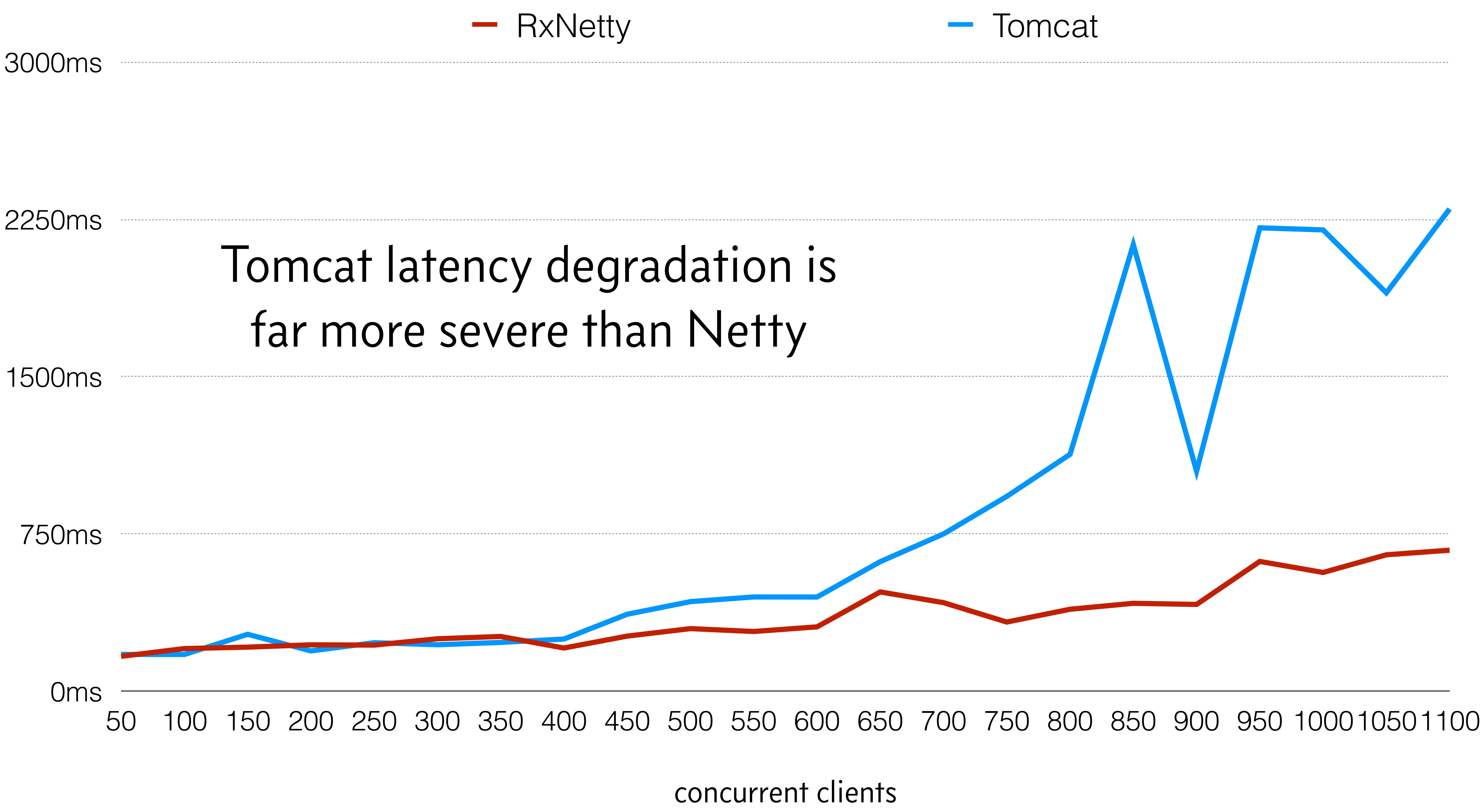


- Netty achieves higher throughput
- Mostly due to lower CPU consumption per request

Average Latency

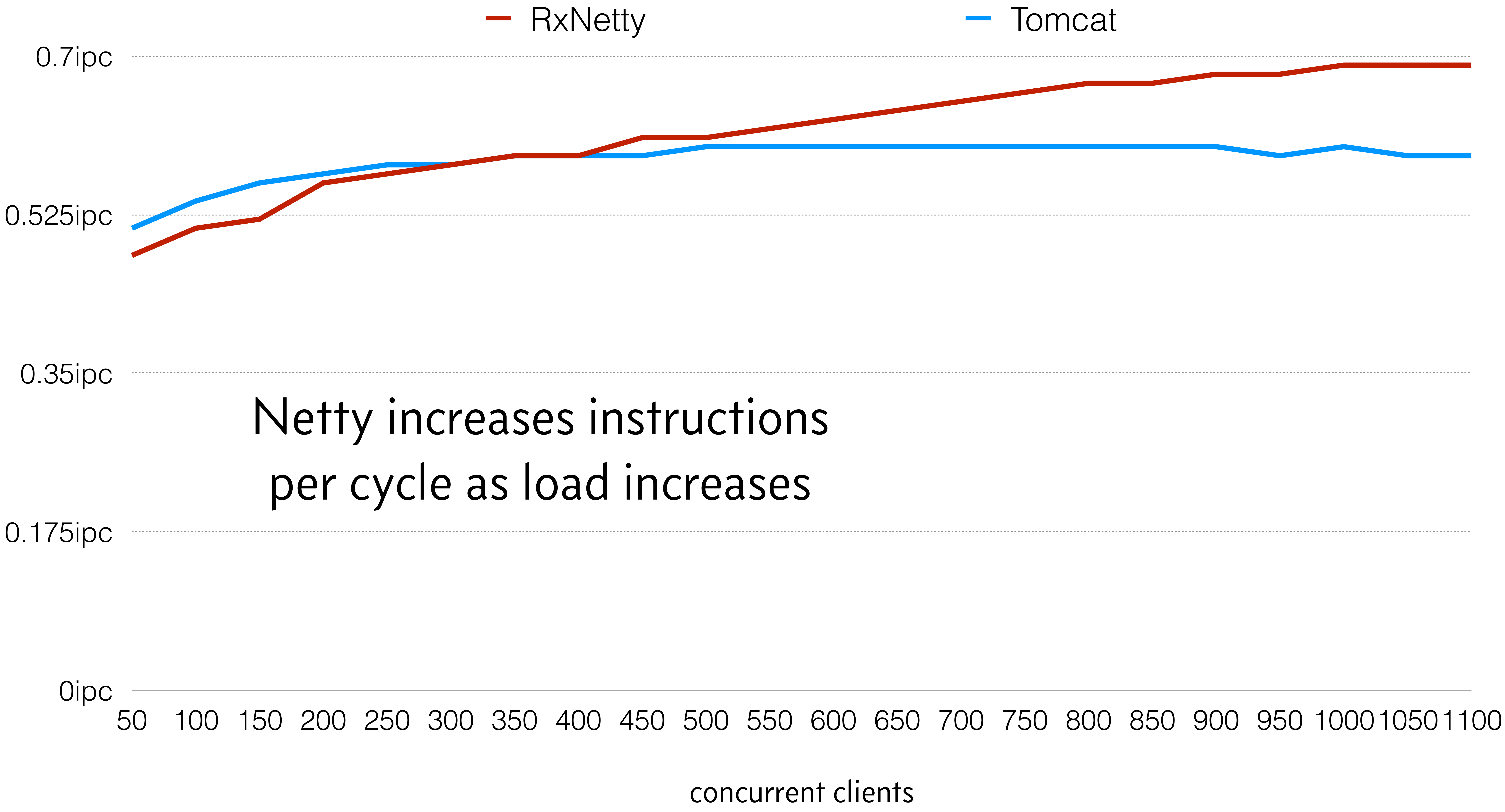


Max Latency



RxNetty
Tomcat
3000ms
2250ms
1500ms
750ms
0ms
Tomcat latency degradation is far more severe than Netty
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750
800
850
900
950
1000
1050
1100
concurrent clients

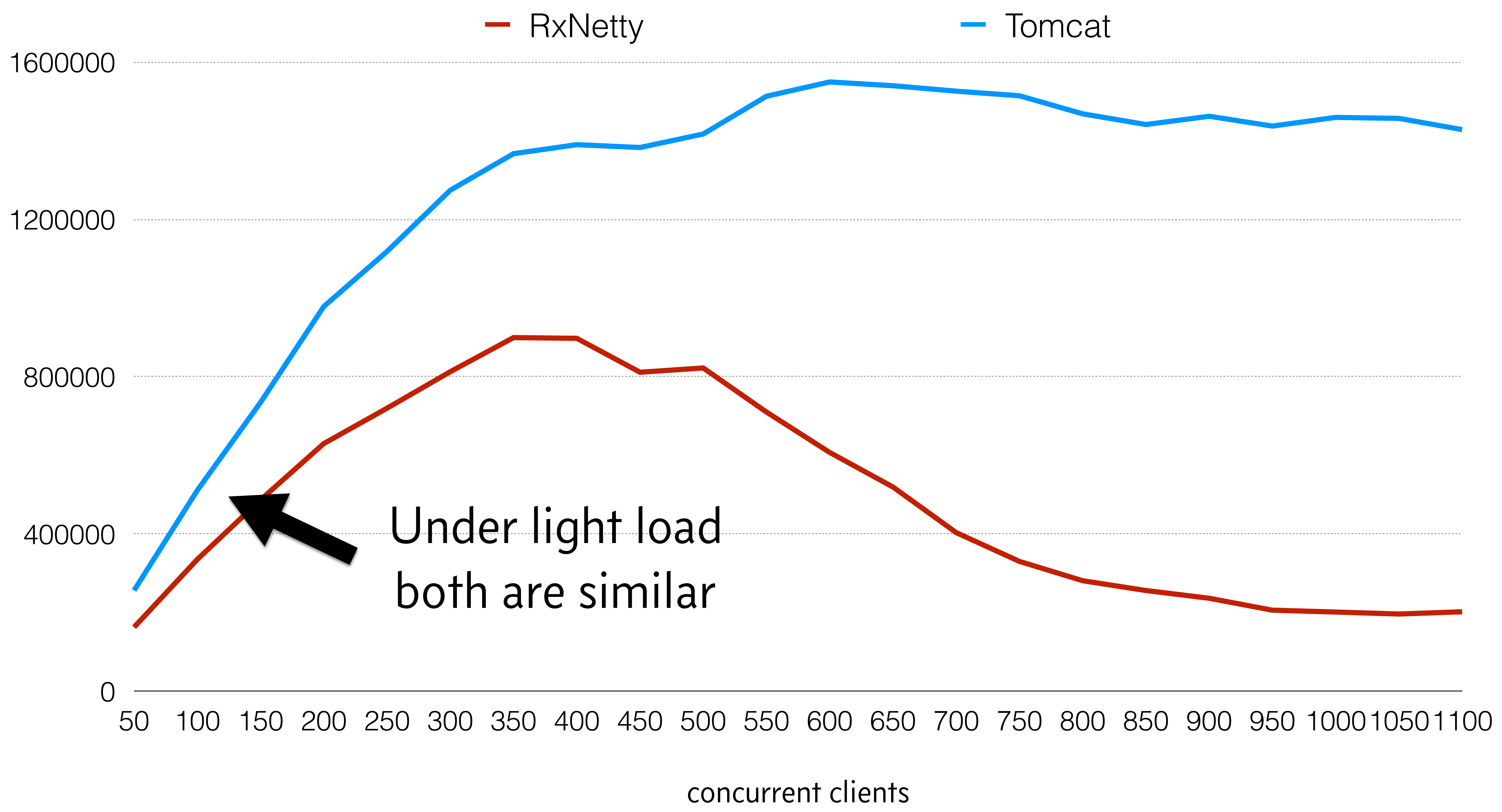
CPU Instructions per Cycle



Netty increases instructions per cycle as load increases


Thread Migrations
RxNetty
Tomcat
1600000
1200000
800000
400000
0
significant divergence
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750
800
850
900
950
1000
1050
1100
concurrent clients

Thread Migrations




Thread Migrations
RxNetty
Tomcat
1600000
1200000
800000
400000
0
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750
800
850
900
950
1000
1050
1100
Increased load causes less migrations, improving IPC and CPU usage per request with event loop architecture
concurrent clients

Why?

- **More efficient framework code**
  - object allocation rates and GC costs
- **Event loop architecture**
  - reduced thread migrations
  - cache warmth → memory locality → instructions per cycle → lower cpu consumption per request
- **Thread pool architecture**
  - lock contention
  - thread migrations

Why?

- **More efficient framework code**
  - object allocation rates and GC costs
- **Event loop architecture**
  - reduced thread migrations
  - cache warmth → memory locality → instructions per cycle → lower cpu consumption per request
- **Thread pool architecture**
  - lock contention
  - thread migrations

Why?

- **More efficient framework code**
  - object allocation rates and GC costs
- **Event loop architecture**
  - reduced thread migrations
  - cache warmth → memory locality → instructions per cycle → lower cpu consumption per request
- **Thread pool architecture**
  - lock contention
  - thread migrations

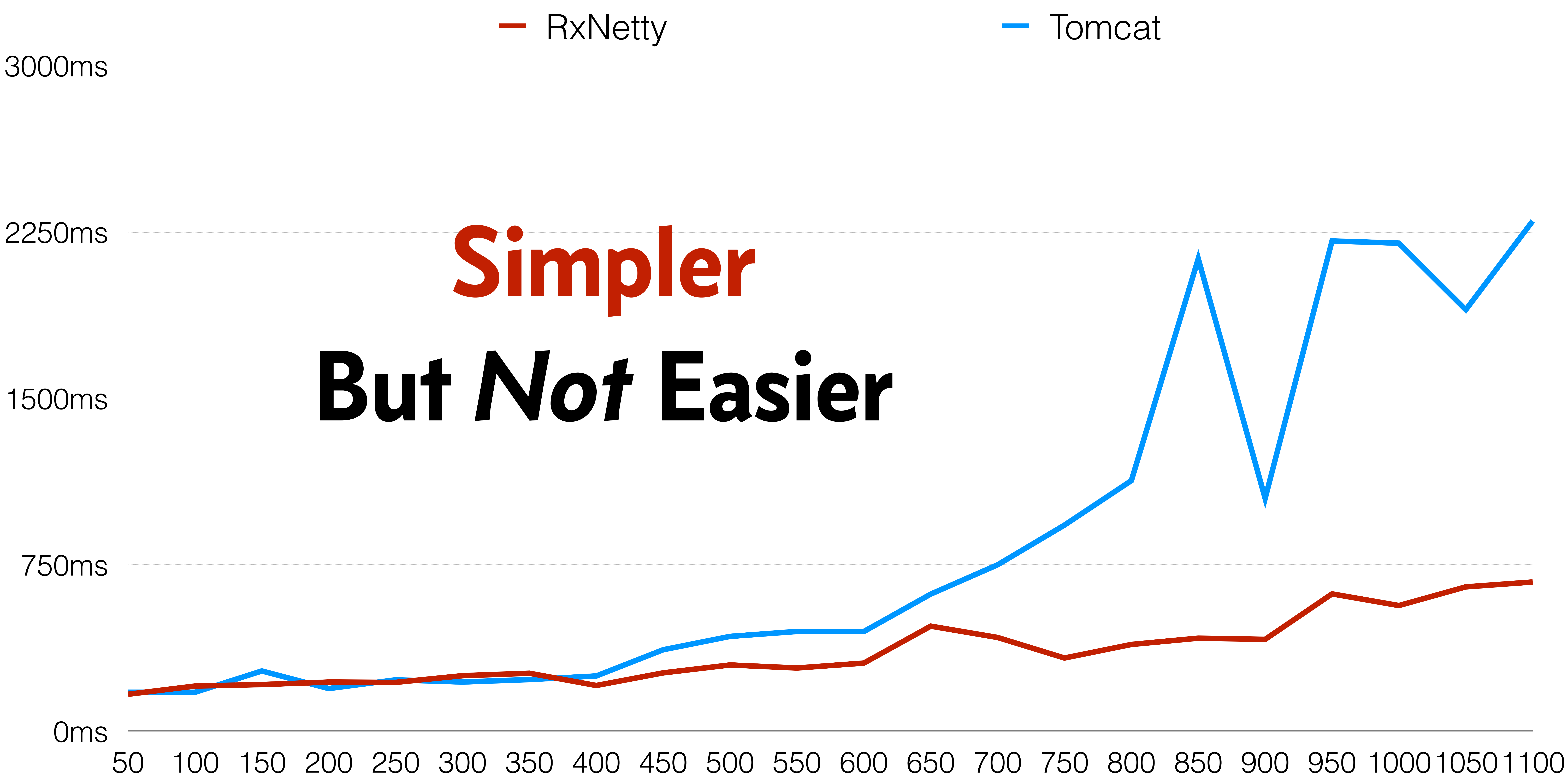
Why?

- **More efficient framework code**
  - object allocation rates and GC costs
- **Event loop architecture**
  - reduced thread migrations
  - cache warmth → memory locality → instructions per cycle → lower cpu consumption per request
- **Thread pool architecture**
  - lock contention
  - thread migrations
- Many more details …

With current JVM and Linux,
event loops have efficiency benefits.

Simpler

But *Not* Easier


RxNetty
Tomcat
3000ms
2250ms
1500ms
750ms
0ms
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750
800
850
900
950
1000
1050
1100
Simpler
But *Not* Easier

A **fully async** architecture does have benefits.


Reactive programming
can be incrementally applied.

Mixed Codebases

Concurrency via
Async Service Composition

Leverage RxJava Schedulers and Threads

Leverage RxJava Schedulers and Threads

(or use Hystrix)

Concurrency is still **non-trivial**

Concurrency is still non-trivial

Rx doesn’t trivialize it

NETFLIX

jobs.netflix.com

Reactive Programming in the Netflix API with RxJava http://techblog.netflix.com/2013/02/rxjava-netflix-api.html

Optimizing the Netflix API http://techblog.netflix.com/2013/01/optimizing-netflix-api.html

Reactive Extensions (Rx) http://www.reactivex.io

Reactive Streams https://github.com/reactive-streams/reactive-streams

RxJava

https://github.com/ReactiveX/RxJava
@RxJava

RxJS

http://reactive-extensions.github.io/RxJS/
@ReactiveX

Ben Christensen
@benjchristensen

Dragon Image: http://www.clipartpanda.com/clipart_images/you-can-use-this-clip-art-on-2194929
Greenfield Image: https://kvaes.wordpress.com/2013/06/05/lingo-explained-greenfield-vs-brownfield/
NY City Image: https://thesandincalifornia.files.wordpress.com/2012/05/newyorkcity-madness.jpg
Futuristic City: http://i.imgur.com/rPTkr9D.jpg