Lecture 13

Introduction to Reactive Programming

SOEN 6441, Summer 2018

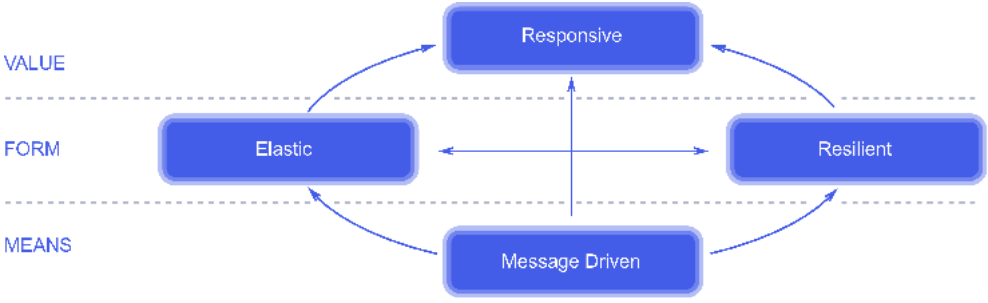
René Witte
Department of Computer Science
and Software Engineering
Concordia University

Outline

René Witte

Reactive Systems

René Witte



Origins of Reactive

René Witte
Concordia University

Transformative vs. Reactive Systems

David Harel and Amir Pnueli (1985): *"On the Development of Reactive Systems"* [HP85]

Transformative Systems: Accept input, transform input, produce output

- e.g., a calculator app

Reactive Systems: Continuously scan environment, respond to changes

- e.g., a spreadsheet app

Interactive vs. Reactive

Gérard Berry (1989): *"Real-time Programming: General Purpose or Special-Purpose Languages"*

Interactive Programs dictate the speed of interaction

Reactive Programs respond to speed dictated by environment

- continuously interact with their environment
- run at speed dictated by the environment
- work in response to external demand

The Reactive Manifesto (2013)

René Witte

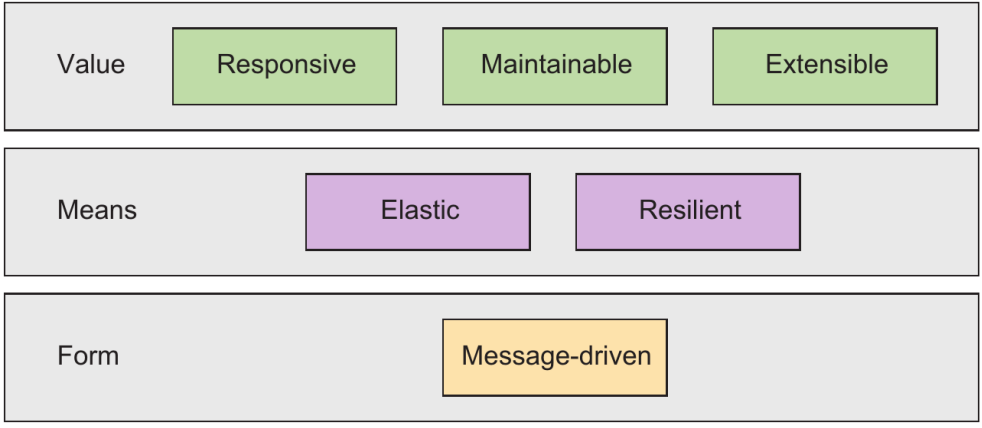
Reactive Applications

Responsive: React to users
Elastic: React to load
Resilient: React to failure
Message Driven: React to events

(The Reactive Manifesto, https://www.reactivemanifesto.org/)

The Structure of Reactive Values

René Witte



René Witte
Concordia University

Front-End Development

AngularJS, Meteor, React.js, . . .

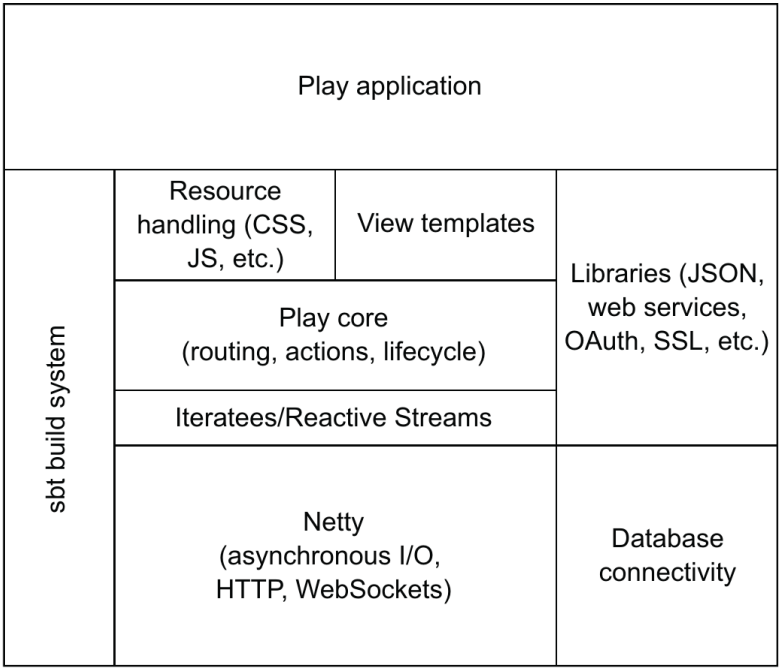
Back-End Development

Microsoft Rx, Node.js, Netty, . . .

Full-Stack Development

Play Framework (JVM)

Play Framework: High-level Architecture



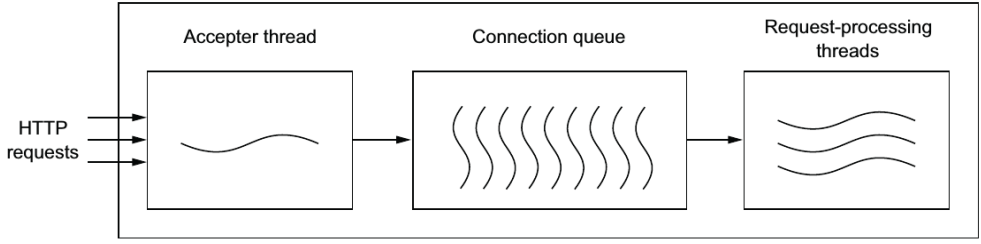
René Witte

Outline

René Witte

Concordia University

Web Application Servers
Threaded Servers (e.g., Apache Tomcat)



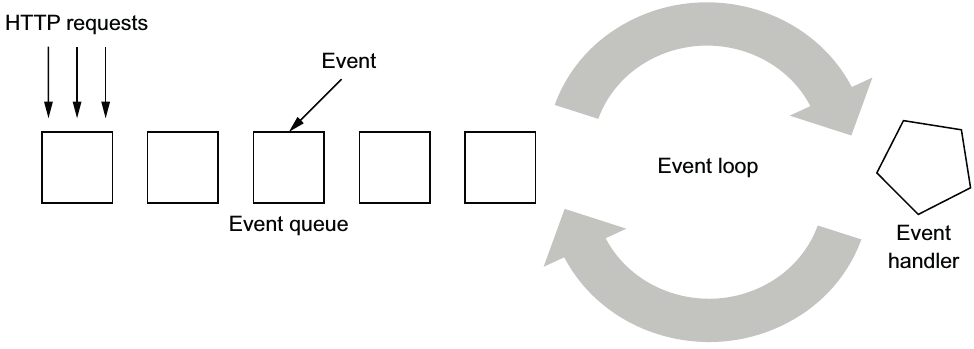
René Witte
Concordia University

René Witte
Concordia University

Train station	Threaded server
More trains come in than there are platforms; trains have to queue up and wait.	More HTTP requests reach the server than there are worker threads; users connecting to the application have to wait.
Trains hanging around at the platform for too long may be cancelled.	HTTP requests taking too long to process are cancelled; the user may see a page with *HTTP Error 408 - Request timeout.*
Too many trains queuing up in the station can cause huge delays and passengers to go home.	Too many requests queuing up can cause users to leave the site.

Evented Web Server

René Witte

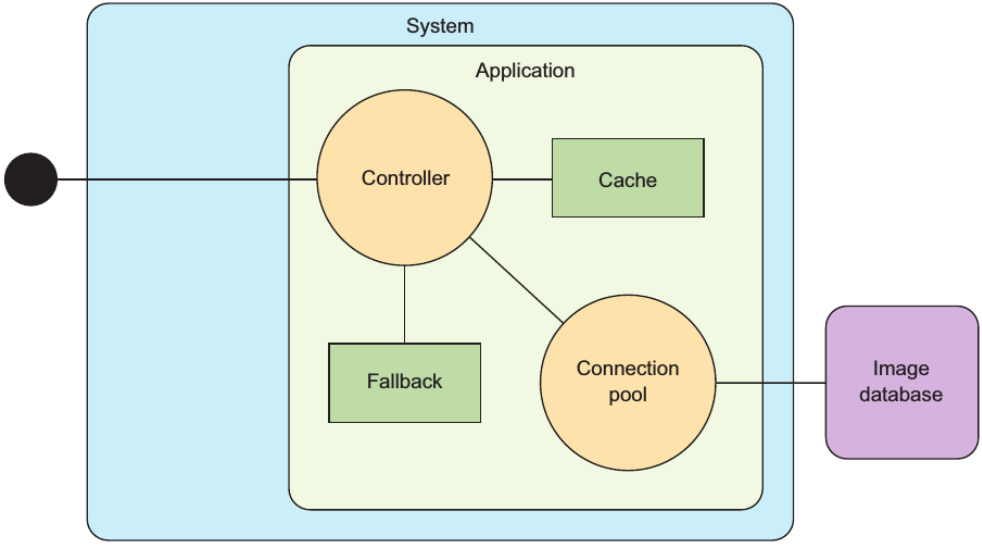


Outline

René Witte

Concordia University

Image Server: Traditional Approach



René Witte

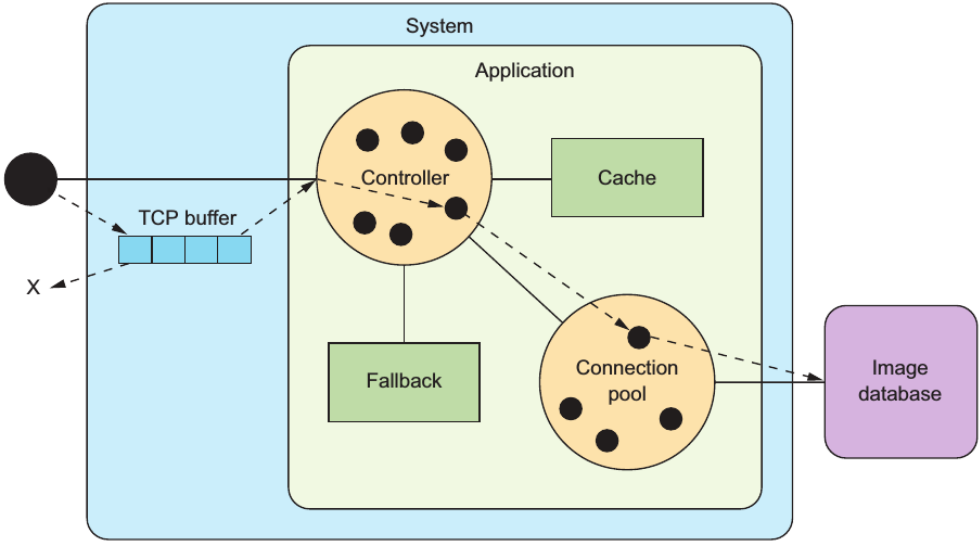
Image Server Controller

```
public interface Images {
  Image get(String Key);
  void add(String key, Image image);
}

public Images cache;
public Images database;

Image result = cache.get(key);
if (result != null) {
  return result;
} else {
  result = database.get(key);
  if (result != null) {
    cache.add(key, result);
    return result;
  } else {
    return fallback;
  }
}
```

René Witte

Image Server: Listener Threads + Connection Pool

René Witte
Concordia University



Analyzing Latency: Little's Law

René Witte
Concordia University

Estimating number of database connections L

$$L = \lambda \times W$$

Image Server Example

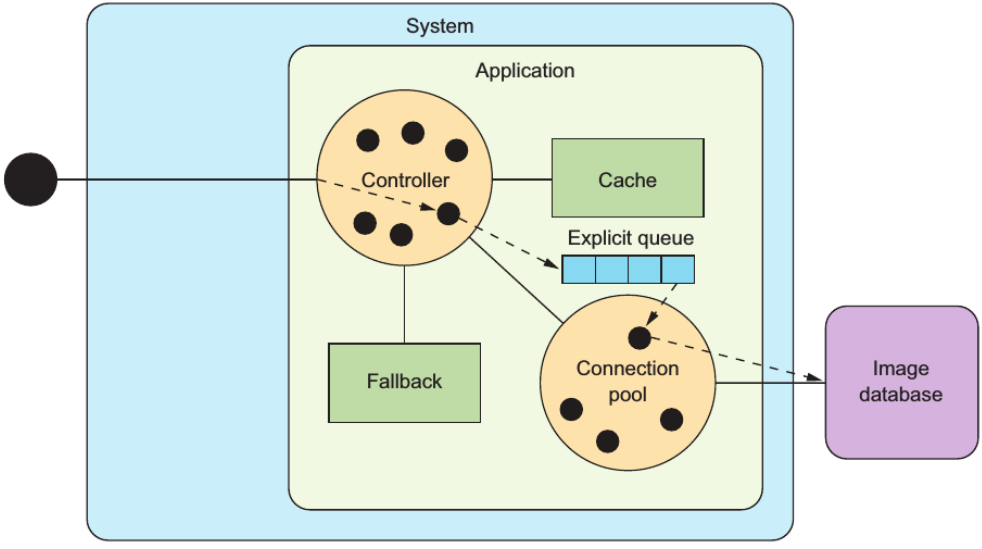
- database takes on average 30ms to respond
- system receives 500 requests per second

$$L = 500r/s \times 0.03s/r$$

Need (on average) at least 15 connections (e.g., threads)

Image Server: Limiting maximum latency with a queue

René Witte
Concordia University



Old-Style Asynchronous Programming

Nested Callbacks in JavaScript (“Callback Hell”)

```
var fetchPriceList = function() {
  $.get('/items', function(items) {
    var priceList = [];
    items.forEach(function(item, itemIndex) {
      $.get('/prices', { itemId: item.id }, function(price) {
        priceList.push({ item: item, price: price });
        if ( priceList.length == items.length ) {
          return priceList;
        }
      }).fail(function() {
        priceList.push({ item: item });
        if ( priceList.length == items.length ) {
          return priceList;
        }
      });
    }
  }).fail(function() {
    alert("Could not retrieve items");
  });
}
```

René Witte

Asynchronous Programming with Java 8

René Witte
Concordia University

Using `CompletableFuture` and Akka

```
ask(actorRef, request, timeout).thenApply(Response.class::cast)
                               .thenAccept(response -> <process>);
```

Properties

Non-blocking (asynchronous) – returns immediately

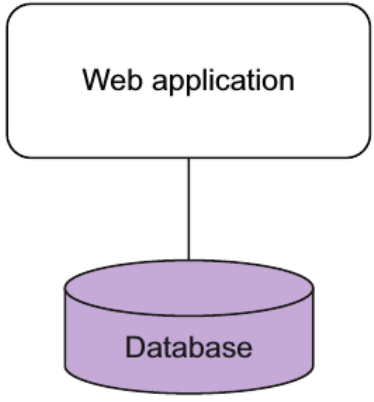
Composable Future – callback to process when response is received

Outline

René Witte

Scalability: Horizontal Application Architecture

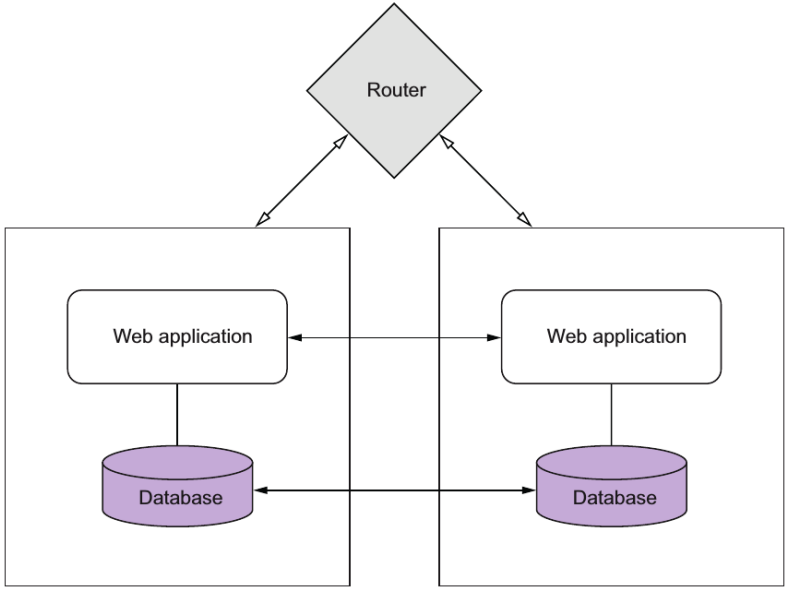
Single-server deployments (vertical scaling)



René Witte

Scalability: Horizontal Application Architecture

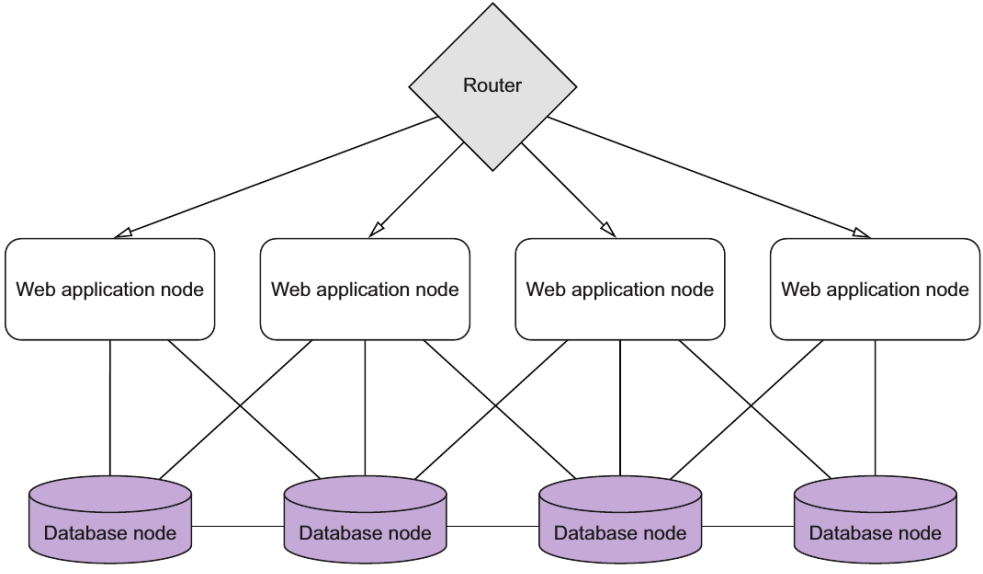
Replicated deployments



René Witte

Scalability: Horizontal Application Architecture

Horizontal deployments (share-nothing, hot redeploy, PaaS)



René Witte
Concordia University

Amdahl's Law

René Witte
Concordia University

Amdahl's Law (1967): Speedup of a program with sequential and parallel parts

Maximum increase in speed that can be achieved by adding additional threads:

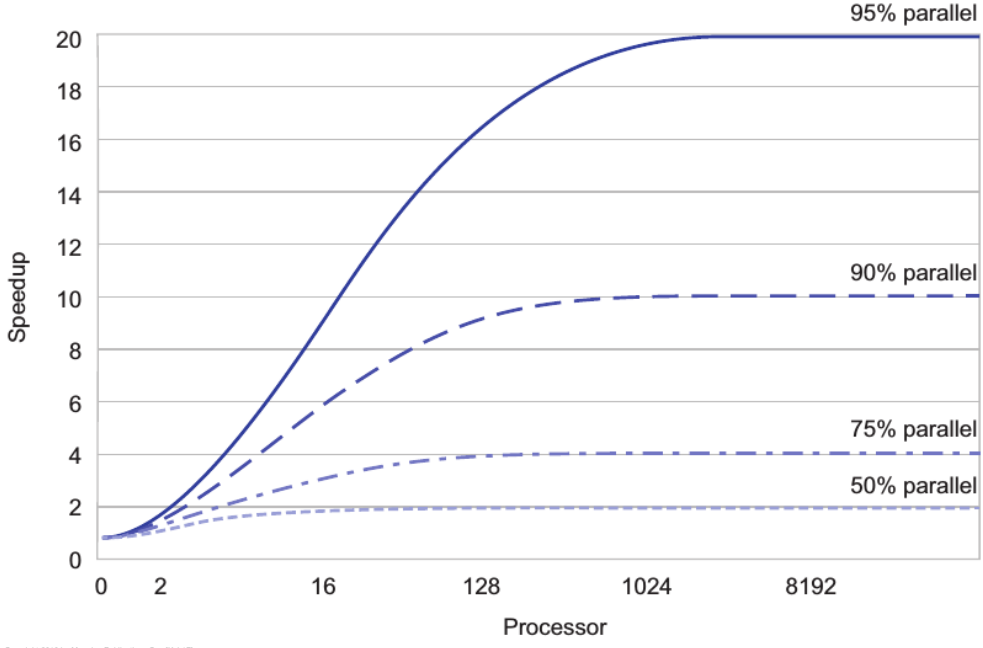
$$S(N) = \frac{T(1)}{T(N)} = \frac{1}{\alpha + \frac{1-\alpha}{N}} = \frac{N}{1 + \alpha(N-1)}$$

with

- N number of available processors (threads)
- α fraction of the program that is serialized
- $T(N)$ time needed when executing with N threads

See https://en.wikipedia.org/wiki/Amdahl%27s_law

Amdahl's Law: Possible Reduction in Runtime



René Witte
Concordia University

Universal Scalability Law

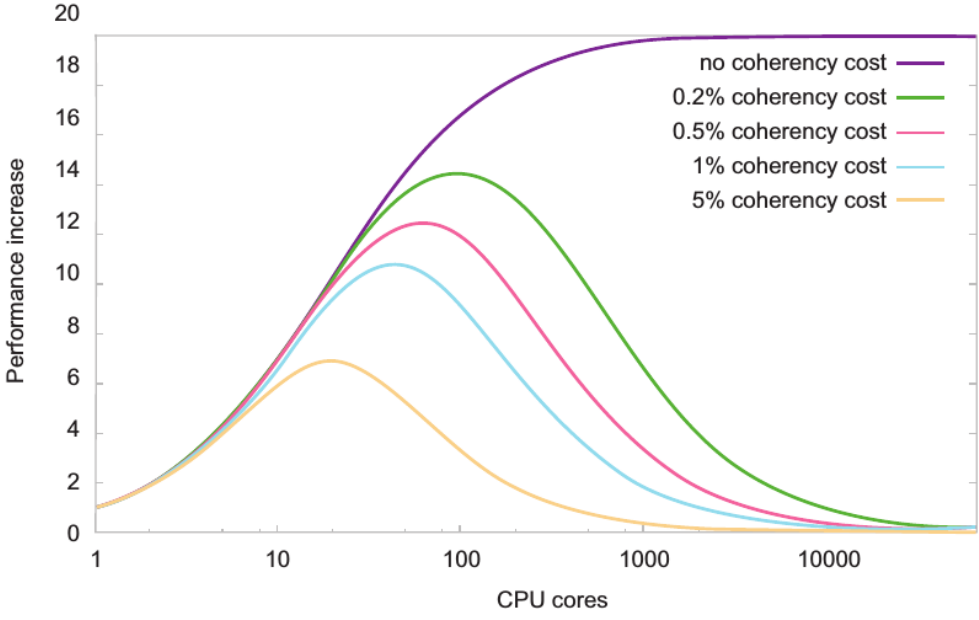
Universal Law of Computational Scalability

Maximum increase in speed that can be achieved by adding additional threads, with an additional factor to account for coordination between threads:

$$S(n) = \frac{N}{1 + \alpha(N-1) + \beta N(N-1)}$$

See https://en.wikipedia.org/wiki/Neil_J._Gunther#Universal_Law_of_Computational_Scalability

Universal Scalability Law: Cost of Coordination



René Witte
Concordia University

Outline

René Witte

Reacting to failure

René Witte
Concordia University

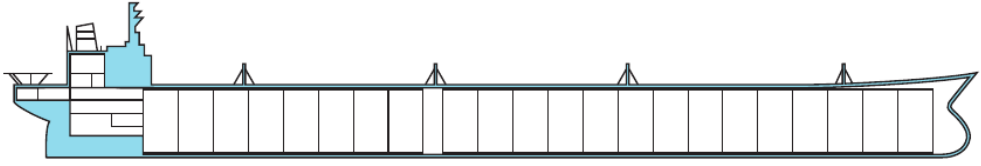
Something in your System *will* fail

- Software will fail
- Hardware will fail
- Humans will fail
- Timeout is failure

Reactive Programming: Building Resilient Systems

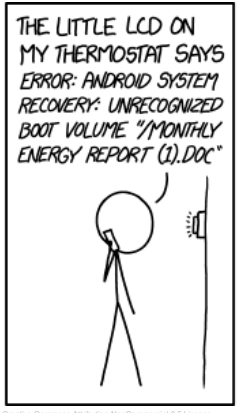
- Fault-tolerant systems (“let it crash”)
- Recover from errors (“self-healing systems”)

Compartmentalization and bulkheading



René Witte

Error Messages



René Witte

Error Handling

Traditional approaches

1. Do nothing (“Cowboy Coding”)
2. Terminate the program
3. Print an error message
4. Use special return codes for errors
5. Set an error flag and let clients check them
6. Throw an exception

The Problem with Exceptions

```
class MyTaxCalculator {
  ...

  try {
    getProvincialTaxRate( provinceName );
  } catch (UnknownProvinceException e) {
    // deal with the error
  } catch (WebServiceTimeoutException e) {
    // ???
  }
}
```

René Witte

Supervisors and Actors

René Witte
Concordia University

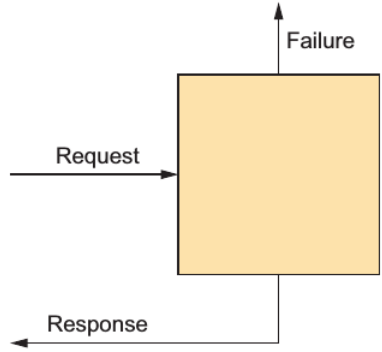
Separation of Concerns

Program split into Supervisors and Actors

- Supervisors manage Actors
- Application work is done by Actors
- Checking progress, error handling, recovery is done by Supervisors

Actor-based Programming

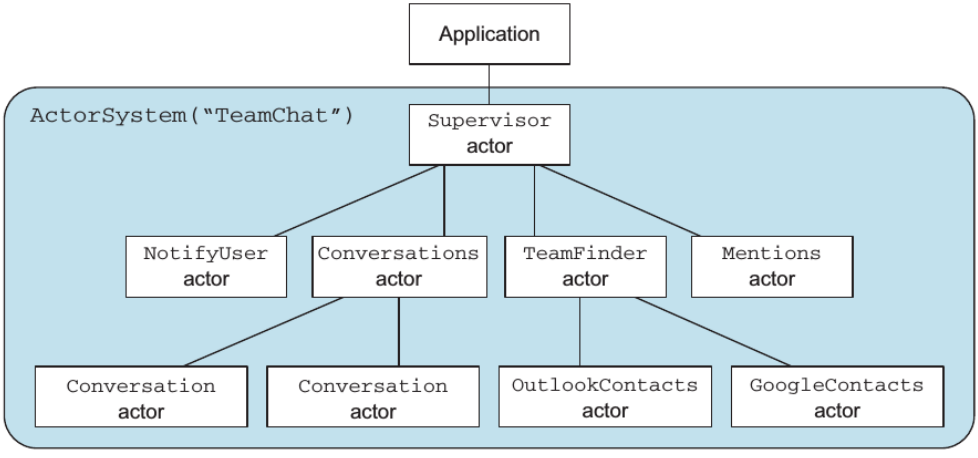
- Comes from the Erlang programming language
- Java, Scala: Akka library (http://akka.io)



Actor-based Programming

Hierarchy of Supervisor Actors and Worker Actors

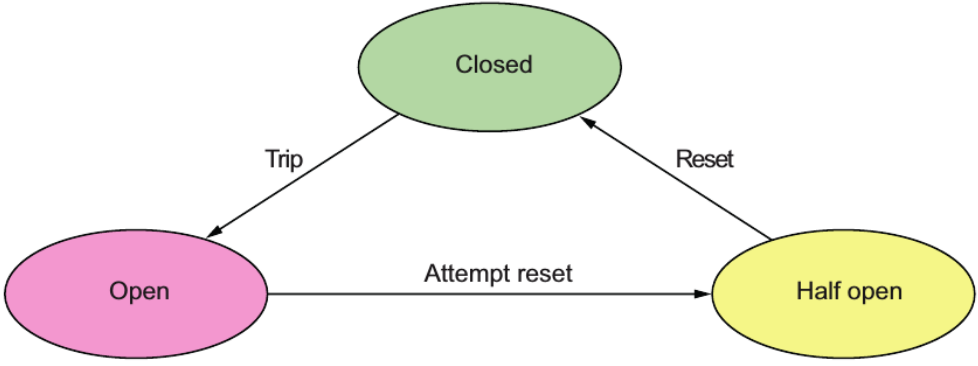
René Witte
Concordia University



Implementation based on Akka (https://akka.io) [RBW17]

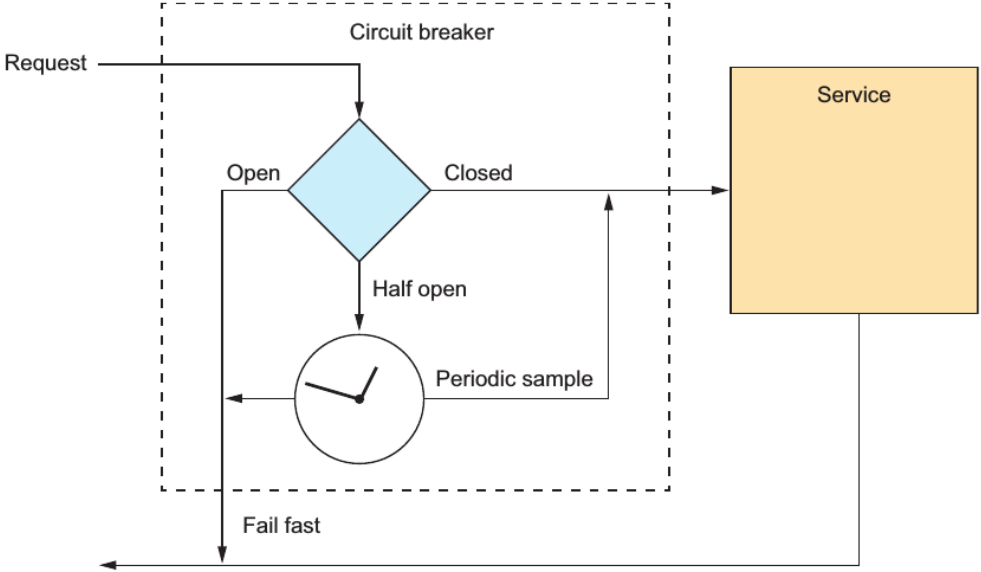
Circuit Breaker: States

René Witte



Reactive Design Pattern: Circuit Breaker

Application to Web Services



Case study: “Walmart Boosts Conversions By 20% With Lightbend Reactive Platform"

https://www.lightbend.com/case-studies/walmart-boosts-conversions-by-20-with-lightbend-reactive-platform

René Witte

Summary

Reactive Programming

- Meaning and origins of reactive applications and reactive technologies, including the Play Framework
- How threads are executed by a CPU and how an asynchronous, event-driven programming style embraced by evented servers makes better use of resources
- Different deployment models, including stateless, horizontal architectures that scale well under load
- The importance of failure handling and different methods that reactive applications employ to become resilient

Outline

René Witte
Concordia University

Reading Material

Required

- [Ber16, Chapter 1] (Reactive Web Applications)

Supplemental

- [Kuh17, Chapters 1, 2] (Reactive Manifesto)

Further Reading

- [HP85] (Reactive Systems)

References

[Ber16] Manuel Bernhardt.
Reactive Web Applications.
Manning Publications, 2016.
https://www.manning.com/books/reactive-web-applications.

[HP85] D. Harel and A. Pnueli.
Logics and models of concurrent systems.
In Krzysztof R. Apt, editor, *On the Development of Reactive Systems*, pages 477–498. Springer-Verlag, New York, NY, USA, 1985.
http://www.wisdom.weizmann.ac.il/~harel/SCANNED.PAPERS/ReactiveSystems.pdf.

[Kuh17] Roland Kuhn.
Reactive Design Patterns.
Manning Publications, 2017.
https://www.manning.com/books/reactive-design-patterns.

[RBW17] Raymond Roestenburg, Rob Bakker, and Rob Williams.
Akka in Action.
Manning Publications, 2017.
https://www.manning.com/books/akka-in-action.