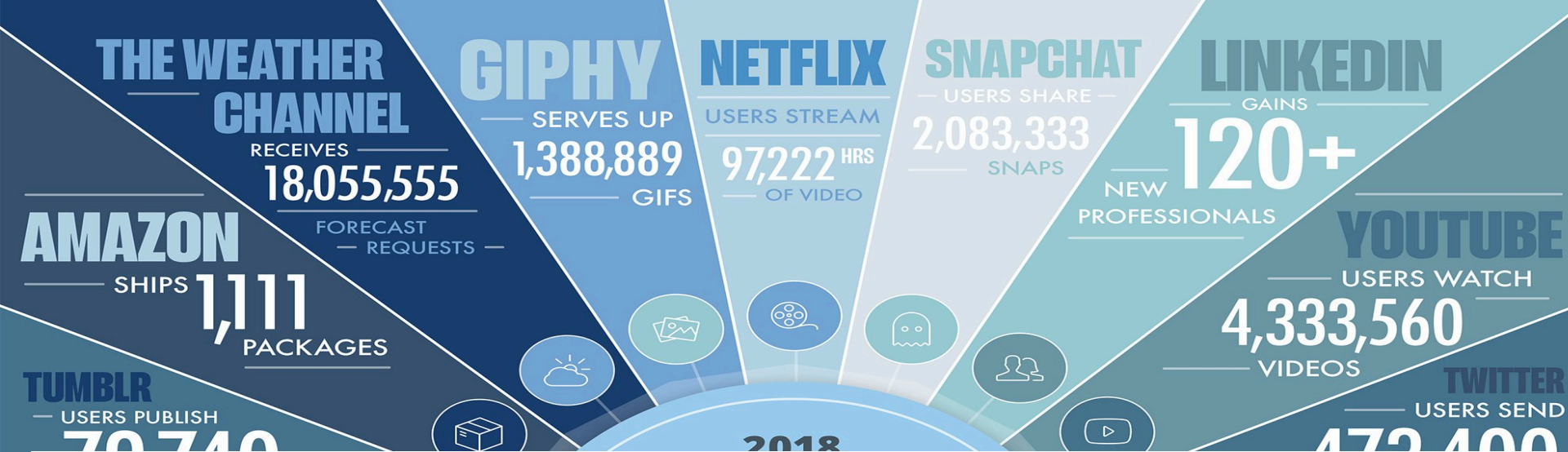


Why Languages for Distributed Systems are Inevitable*

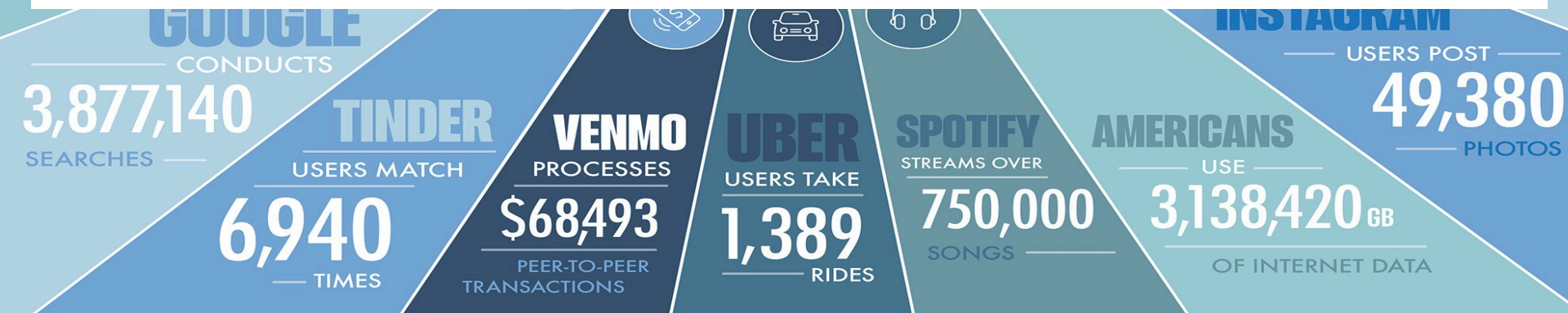
Prof. Guido Salvaneschi

***With a tip of the hat to Jonathan Aldrich**
[Jonathan Aldrich. The power of interoperability: why objects are inevitable, Onward! 2013]



2 years = 90% of data ever generated

By 2020: 1.7MB per second by each person



Real Time Processing



Fraud Detection



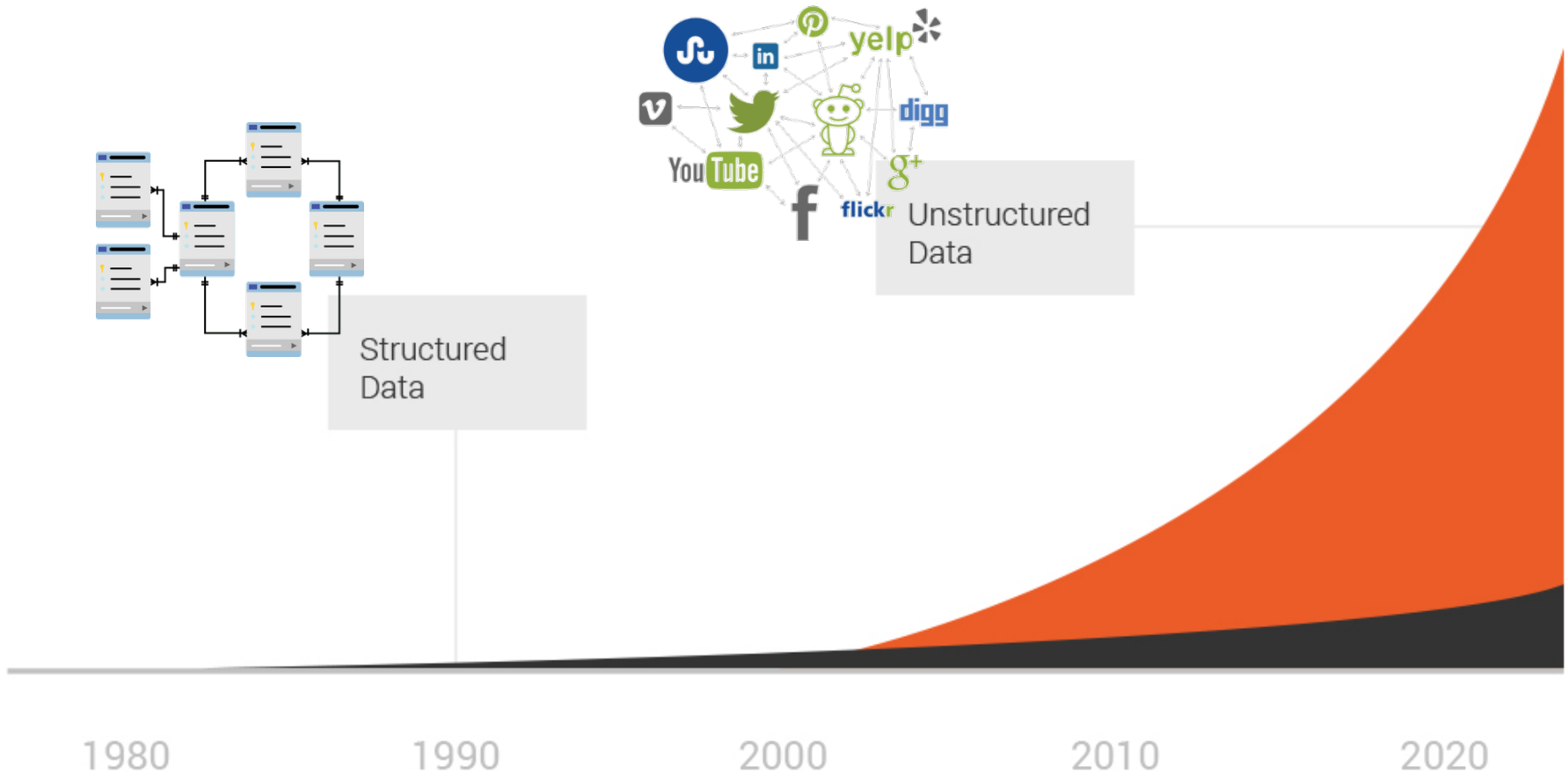
Real Time Business
Intelligence



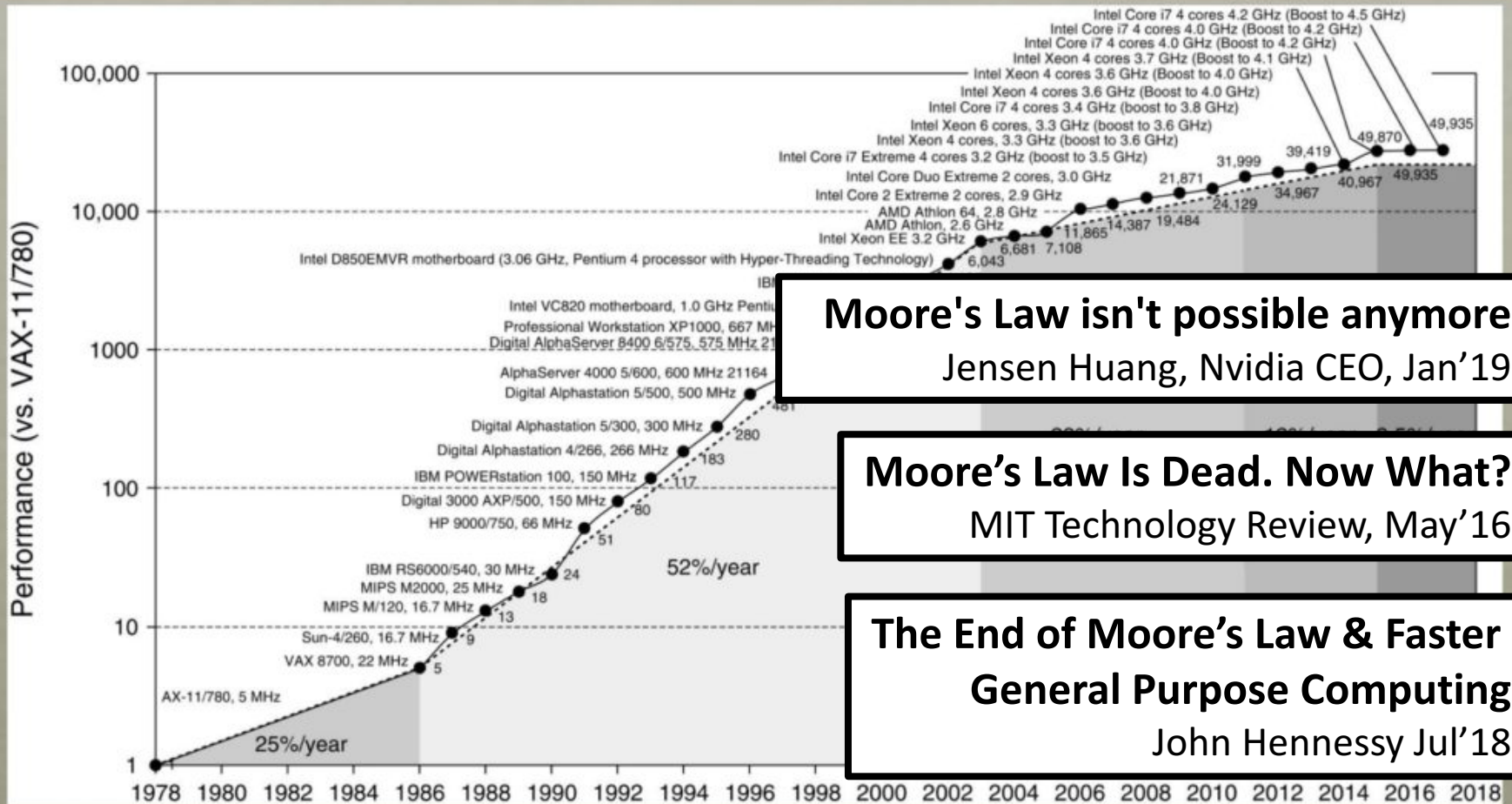
Cloud Monitoring

Structured and Unstructured Data

IDC and EMC project that data will grow to 40 ZB by 2020



UNIPROCESSOR PERFORMANCE (SINGLE CORE)

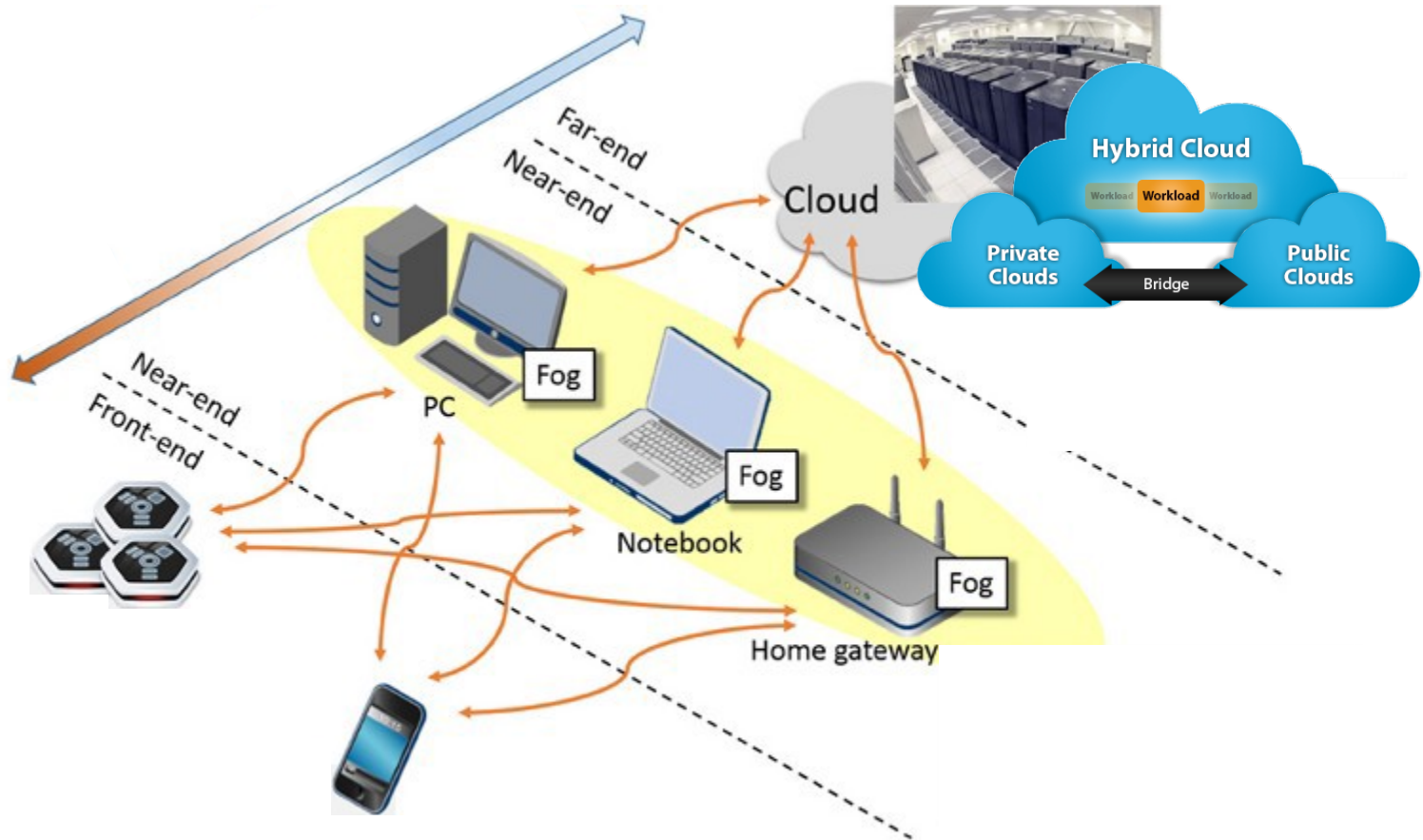


Moore's Law isn't possible anymore
Jensen Huang, Nvidia CEO, Jan'19

Moore's Law Is Dead. Now What?
MIT Technology Review, May'16

**The End of Moore's Law & Faster
General Purpose Computing**
John Hennessy Jul'18

Processing at the Edge





**Building a distributed system requires
a methodical approach to requirements.**

BY MARK CAVAGE

There Is No Getting Around It: You Are Building a Distributed System

*Distributed systems are
difficult to understand,
design, build, and operate.*

*They introduce exponentially
more variables into a design
than a single machine does,
[...]*

DISTRIBUTED SYSTEMS ARE difficult to understand, design,
build, and operate. They introduce exponentially
more variables into a design than a single machine
does, making the root cause of an application problem


Architecting for **Failure**

Why are distributed systems so hard?

Markus Eisele



Why Distributed Systems Are Hard to Develop — and How to Motivate Your Team

 Jon Edvald in Garden [Follow](#)
Mar 28 · 5 min read

Mot

- Developing, testing and tuning distributed applications is **hard**


THE NEW STACK Ebooks Podcasts Events Newsletter

Architecture Development Operations

CLOUD NATIVE / CONTAINERS / MICROSERVICES / CONTRIBUTED

Distributed Systems Are Hard

25 Aug 2017 6:00am, by Anne Currie



Distributed Systems: **Ugly**, **Hard**, and **Here to Stay**

LANGUAGES FOR DISTRIBUTED APPLICATIONS

SCALABLE

LOW LATENCY

EVENT BASED

DISTRIBUTED

HETEROGENEOUS



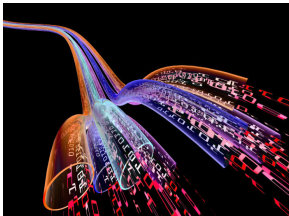
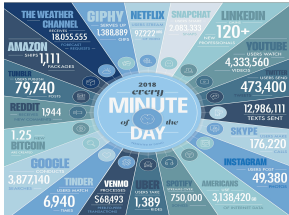
Big Data

Real Time
Requirements

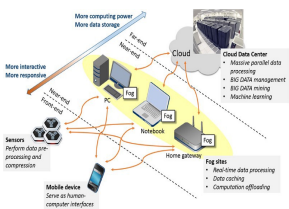
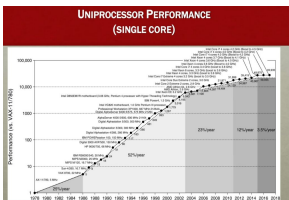
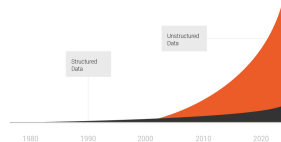
Unstructured
Data

No Moore's
Law

Edge



Structured and Unstructured Data
IDC and EMC project that data will grow
to 40 ZB by 2020



DATA-INTENSIVE
DISTRIBUTED APPLICATIONS

SCALABLE

LOW LATENCY

EVENT BASED

DISTRIBUTED

HETEROGENEOUS

Languages for Distributed Applications

Reactivity

Software Design

Privacy

Fault tolerance

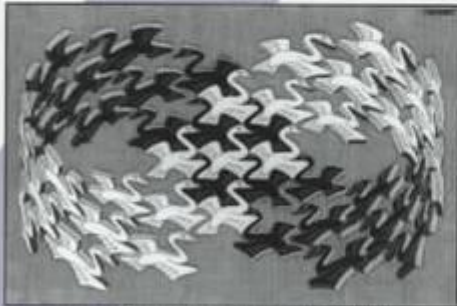
Consistency

REACTIVITY

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Graphics Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Observer Pattern

*“Define a one-to-many dependency between objects so that when one object changes state, **all its dependents are notified and updated automatically.**”*

Is current technology enough?

01:12:04
ww:dd:hh

This is all what we want to express!

```
val tick = 0 // Increase
val hour <== tick % 24
val day <== (tick/24)%7 + 1
val week <== ...
```

EVENTS

```
imperative evt tick[Unit]
var hour: Int = 0
var day: Int = 0
var week: Int = 0

tick += nextHour
def nextHour() {
  hour = (hour + 1) % 24
}

evt newDay [Unit] = tick && (() => hour == 0)
newDay += nextDay
def nextDay () {
  day = (day + 1) % 7
}

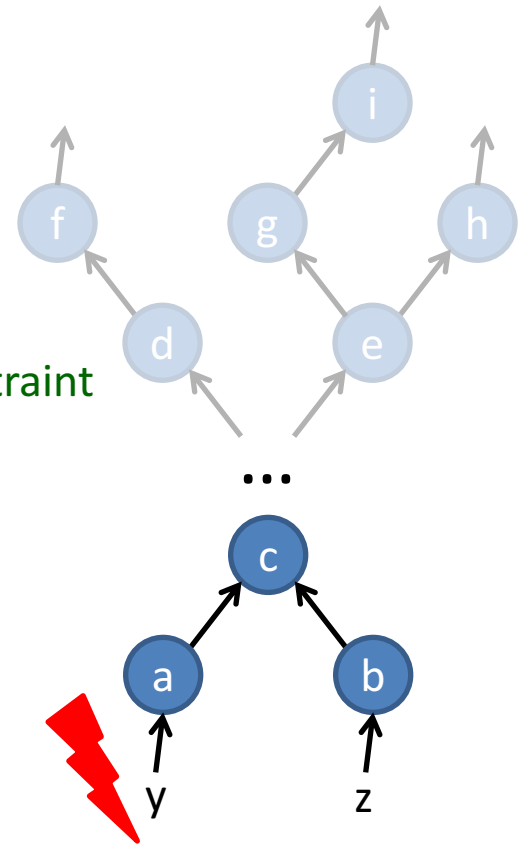
evt newWeek [Unit] = ...
newWeek += nextWeek
def nextWeek() {
  ...
}
```

REScala: Combining Signals and Events

- Signals: What about expressing functional dependencies as constraints ?

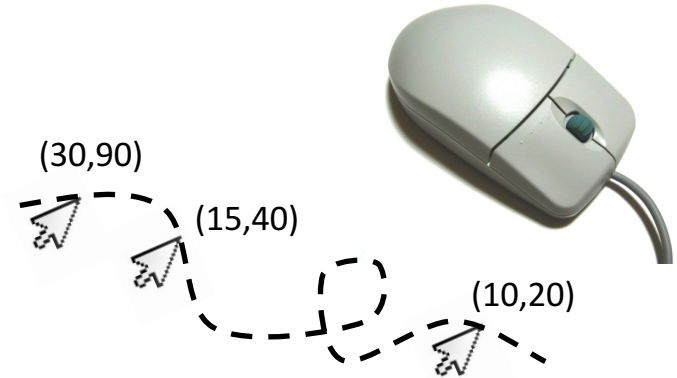
```
val a = 3
val b = 7
val c = a + b // Statement
...
println(c)
> 10
a = 4
println(c)
> 10
```

```
val a = Var(3)
val b = Var(7)
val c = Signal{ a + b } // Constraint
...
println(c)
> 10
a = 4
println(c)
> 11
```



Reactive Programming: Example

- Mixing signals and events
- Reactive code is simple!



val position: Signal[(Int,Int)] = mouse.position

val shiftedPosition: Signal[(Int,Int)] = Signal{ mouse.position + (10, 10) }

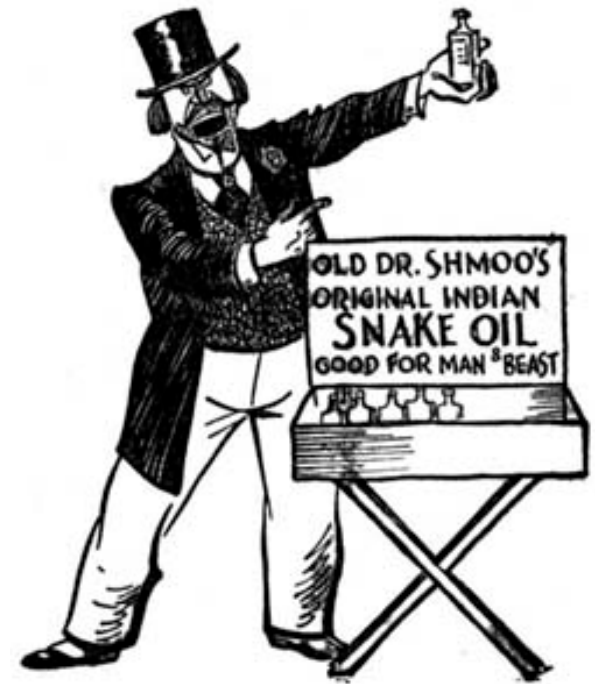
evt clicked: Event[Unit] = mouse.clicked

val lastClick: Signal[(Int,Int)] = position snapshot clicked

OO integration: Both signals and events are subject to inheritance and runtime polymorphism!

Claim: RP beats OO (Observer)

- Easier to compose
- Declarative style
- **Easier program comprehension**
- State management not explicit
- Automatic memory management



Flapjax: A Programming Language for Ajax Applications

“Obviously, the Flapjax code may not appear any ‘easier’ to a first-time reader”

[Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, Shriram Krishnamurthi, *Flapjax: A Programming Language for Ajax Applications*, OOPSLA'09]

This paper presents Flapjax, a language designed for contemporary Web applications. These applications communicate with servers and have rich, interactive interfaces. Flapjax provides two key features that simplify writing these applications. First, it provides *event streams*, a uniform abstraction for communication within a program as well as with ex-

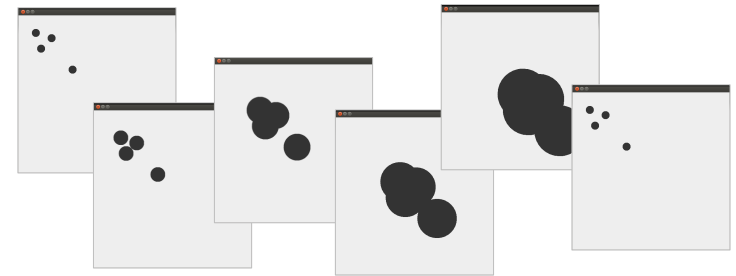
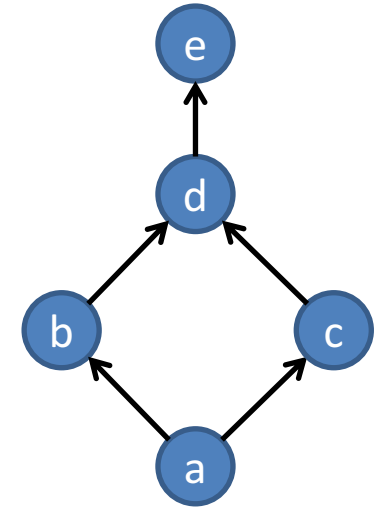
Keywords JavaScript, Web Programming, Functional Reactive Programming

1. Introduction

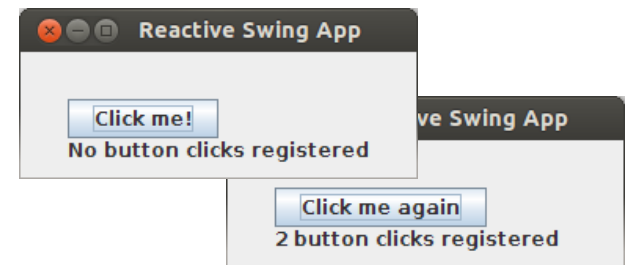
The advent of broadband has changed the structure of application software. Increasingly, desktop applications are migrating to the Web. Programs that once made brief forays

The study

- 10 applications, ~130 subjects
 - RP and OO group (**between** subj.)
 - Questions for comprehension
- What to measure?
 - **Time** to answer a question
 - Amount of **correct answers**



```
Advanced code understanding
Read the code and answer the question.
1 import commons._
2 import react._
3 import react.Signal
4 import macro.SignalMacro.{SignalM => Signal}
5 import react.events._
6 import swing.{Panel, JFrame, SimpleSwingApplication}
7 import swing.Swing
8 import java.awt.{Color, Graphics2D, Dimension}
9 import javax.swing.JPanel
10 import Animation._
11
12
13
14
15
16
17
18 object Squares_Reactive extends SimpleSwingApplication {
19
20   // -- APPLICATION LOGIC -----
21
22   object square1 {
23     val position = Signal { Point(time().s * 100, 100) }
24   }
25
26   object square2 {
27     val v = Signal { time().s * 100 }
28     val position = Signal { Point(time().s * v(), 200) }
29   }
30
31   // painting components
32   (square1.position.changed ||
33    square2.position.changed) += { _ => Swing onEDT { top.repaint() } }
34
35   // -- Graphic Stuff -----
36
37   lazy val panel: JPanel = new JPanel {
38     override def paintComponent(g: Graphics2D) {
39       super.paintComponent(g)
40       g.fillRect(
41         square1.position.getValue.x.toInt - 8,
42         square2.position.getValue.y.toInt - 8,
43         16, 16)
44     }
45   }
46 }
```



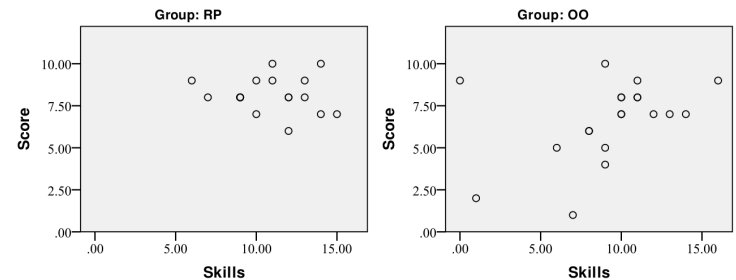
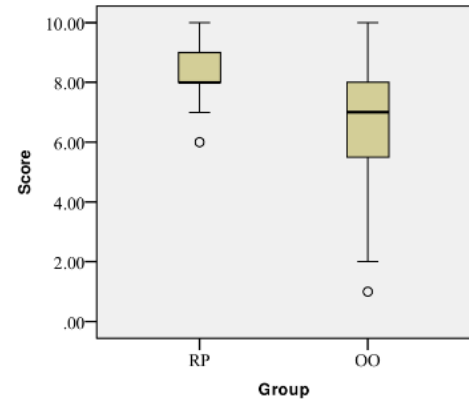
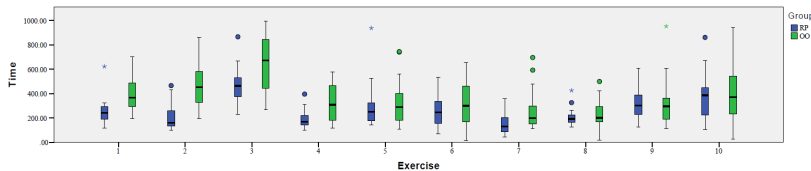
Results



REScala **increases correctness** of program comprehension



In REScala, comprehension is **no more time-consuming**



[Guido Salvaneschi, Sven Amann, Sebastian Proksch, Mira Mezini, **An Empirical Study on Program Comprehension with Reactive Programming**, FSE'14.]

[G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, M. Mezini, **On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study**, TSE'17]

Teaching Reactive Programming

Master course (9CP)

Software Engineering: Design & Construction

Design patterns

Domain specific languages

Software architecture

Reactive Programming

...

HOW TO DEBUG

REACTIVE PROGRAMS

!?



Debugging for Reactive Programming

Traditional debugging
(Imperative)



```
0x051DE590 a0 e5 1d 05 4c
0x051DE5A0 20 e6 1d 05 b1
0x051DE5B0 40 e6 1d 05 00
0x051DE5C0 bc c3 94 70 b4
```

Program Stack

~~Step over
statements~~

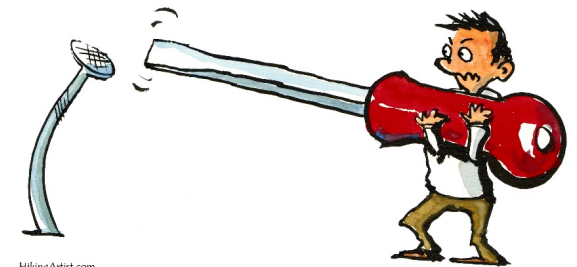
~~Inspect state~~

Reactive Programming
(Declarative)

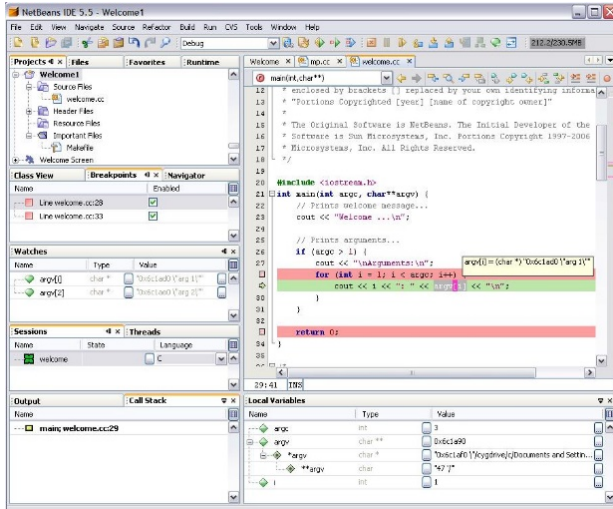
Signals

Abstract
over state

!?



A Paradigm Shift



Traditional Debugging

Stepping over statements

Breakpoint on line X

Inspect memory

Navigate object references

Per-function absolute performance

RP Debugging

Stepping over the dependency graph

Breakpoint on node X

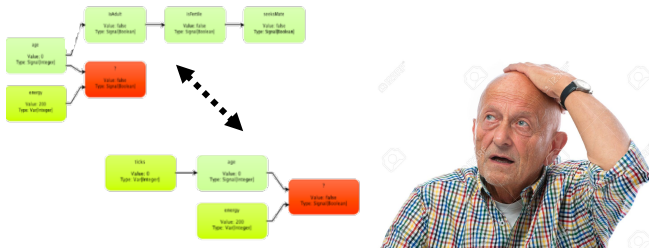
Inspect values in the dependency graph

Navigate signals in the graph

Per-node relative performance

Bug Hunting with Reactive Debugging

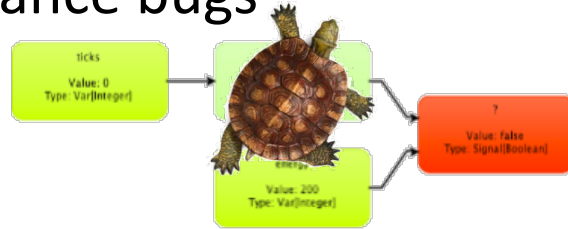
Missing dependencies



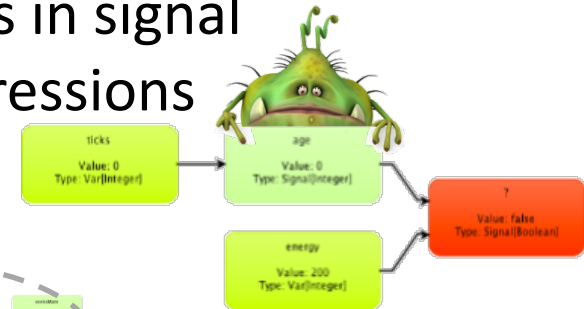
Understanding RP programs



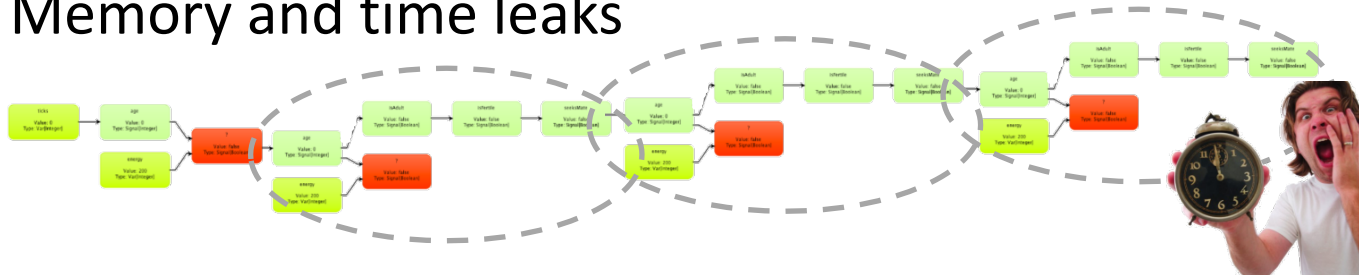
Performance bugs



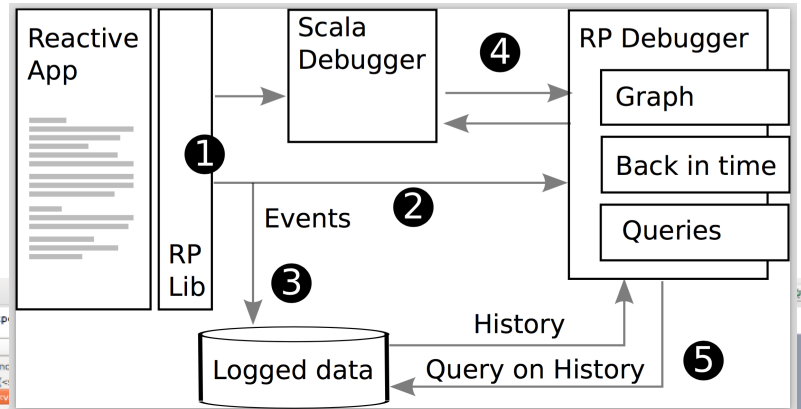
Bugs in signal expressions



Memory and time leaks



Reactive Inspector (Eclipse plugin - Scala IDE)



The screenshot shows the Eclipse IDE with the Reactive Inspector plugin. The code editor on the left displays the following code:

```

object AnimationExample extends SimpleSwingApplication {
  rescala.ReactiveEngine.Log = new REScalaLogger

  val time = App.time // Global time signal

  val slowIncreasing = Signal { time() / 100 }
  val fastIncreasing = Signal { time() / 10 }
  val pulsing = Signal { (time() % 5) * 50 }
  val quadratic = Signal { time() * time() }

  val circle = new Circle((fastIncreasing, fastIncreasing), pulsing)
  val square = new Square((Var(200), slowIncreasing), pulsing)
  val pic = new Picture(new Image("java_logo.png"), pulsing, (fastI
  val pic2 = new Picture(new Image("java_logo.png"), Var(3), (slowI

  val panel = new AnimationPanel

  panel.add(circle)
  panel.add(square)
  panel.add(pic)
  panel.add(pic2)

  def top = new MainFrame {
    title = "Animation Example"
    contents = panel
  }

  class Square(center: (Signal[Int], Signal[Int]),
    side: Signal[Int]) extends Drawable {
    val changed = center.1.changed || center.2.changed || side.chan

    def paint(g: Graphics2D) {
      g.fillRect(center.1.get - (side.get / 2),
        center.2.get - (side.get / 2),
  
```

The Reactive Tree in the center shows a dependency graph with nodes like 'tick', 'fastIncreasing', 'pulsing', 'quadratic', 'square', 'pic', and 'pic2'. A tooltip for the 'pic' node shows its properties: Name: height, Type: Signal, Value: 7, Class: de.tuda.stg.reclipse.examples.Picture, Source Line: 84.

The console at the bottom shows the following log output:

```

evaluationYielded(pulsing,"200")
nodeCreated(<nodeName>)
nodeEvaluated(<signalName>)
nodeValueSet(<varName>)
dependencyCreated(<nodeName>,<nodeName>)
evaluationYielded(<nodeName>,<value>)
evaluationException(<nodeName>?)
  
```

Blue arrows from the diagram above point to the following features in the screenshot:

- 1. Node Search: Points to the search bar in the Reactive Tree.
- 2. Node Breakpoints: Points to the 'Enable Watchpoint' button in the Reactive Tree.
- 3. Tree Inspection: Points to the Reactive Tree structure.
- 4. Node Queries: Points to the 'Submit Query' button in the console.
- 5. Back-in-Time Debugging: Points to the 'Back in time' button in the RP Debugger.
- 6. Reactive Breakpoints: Points to the 'Reactive Breakpoints' panel in the console.

NODE
SEARCH

NODE
BREAKPOINTS

TREE
INSPECTION

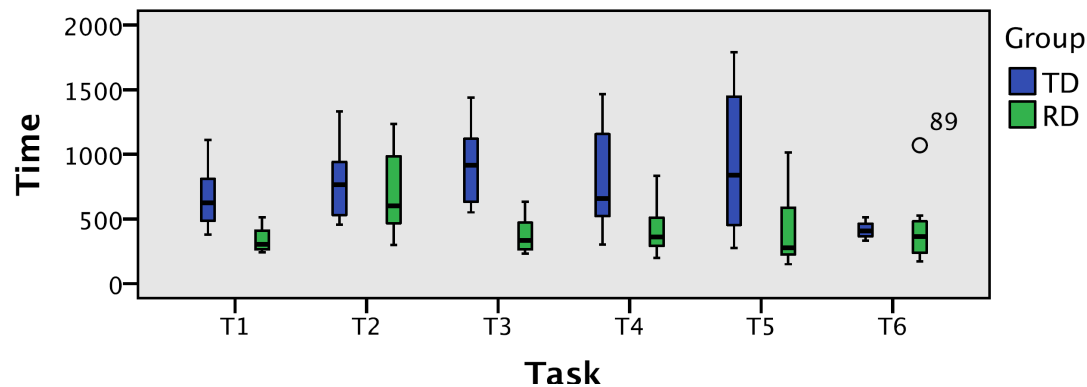
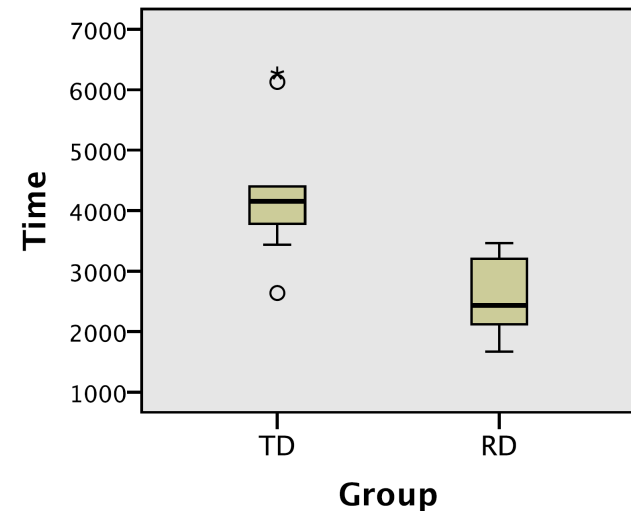
NODE
QUERIES

BACK-IN-TIME
DEBUGGING

REACTIVE
BREAKPOINTS

Evaluation

- 18 subjects, 2 groups
- 6 applications,
 - 2D simulation, fisheye animation, reactive network, arcade Pong, RSS Feed reader, shapes animation



Automated Refactoring to Reactive Programming

Mirko Köhler
Reactive Programming Technology
Technische Universität Darmstadt
Darmstadt, Germany
koehler@cs.tu-darmstadt.de

Guido Salvaneschi
Reactive Programming Technology
Technische Universität Darmstadt
Darmstadt, Germany
salvaneschi@cs.tu-darmstadt.de

Abstract—

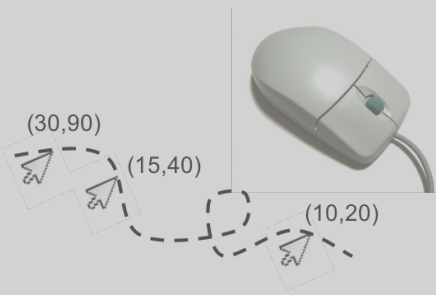
Reactive programming languages and libraries, such as ReactiveX, have been shown to significantly improve software design and have seen important industrial adoption over the last years. Asynchronous applications – which are notoriously error-prone to implement and to maintain – greatly benefit from reactive programming because they can be defined in a declarative style, which improves code clarity and extensibility. In this paper, we tackle the problem of refactoring existing software that has been designed with traditional abstractions for asynchronous programming. We propose 2RX, a refactoring approach to automatically convert asynchronous code to reactive programming. Our evaluation on top-starred GitHub projects shows that 2RX is effective with common asynchronous constructs and it can provide a refactoring for 91.7% of their occurrences.

Keywords-refactoring; asynchronous programming; reactive programming; Java;

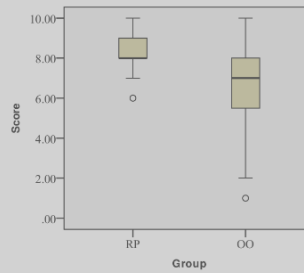
over low level abstractions like threads, but come with their own limitations. For example, `AsyncTask` does not easily support composition, like sequencing multiple asynchronous computations.

Recently, Reactive Programming (RP) has emerged as a programming paradigm specifically addressing software that combines events [3]. Crucially, RP allows to easily express computations on event streams that can be chained and combined using high-order functional operators. This way, each operator can be scheduled independently, providing a convenient model for asynchronous programming. As a result, RP provides means to describe asynchronous programs in a declarative way. Previous research showed that this declarative software design of reactive programs can be used to

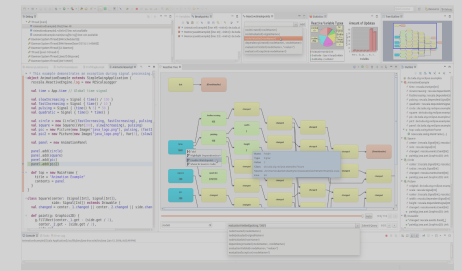
[ASE'19]



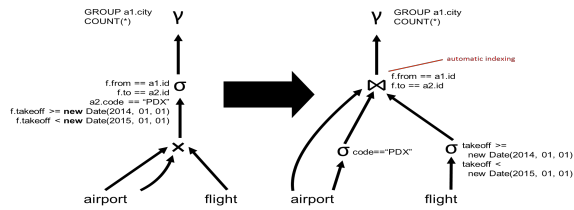
Language **abstractions** for OO reactive programming



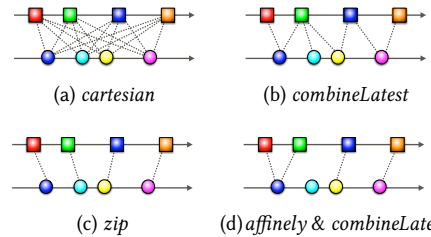
Controlled **experiments**



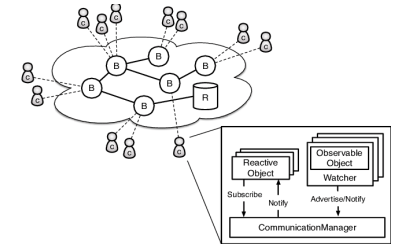
Tools supporting the development process



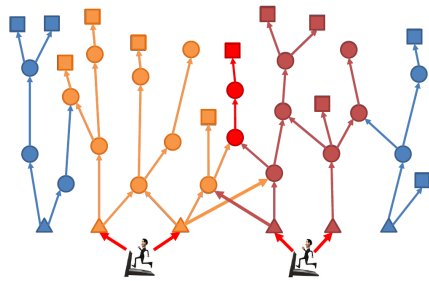
Incremental changes



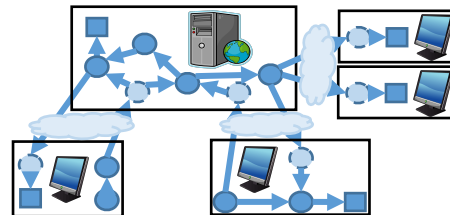
Semantics of Event Correlation



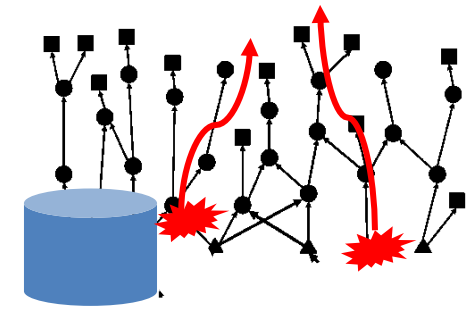
Configurable Consistency



Concurrency



Distribution



Fault Tolerance

Distributed REScala: An Update Algorithm for Distributed Reactive Programming

Joscha Drechsler,
Guido Salvaneschi
Technische Universität Darmstadt,
Germany
<lastname >@cs.tu-darmstadt.de

Ragnar Mogk
Technische Universität Darmstadt,
Germany
ragnar.mogk@stud.tu-darmstadt.de

Mira Mezini
Technische Universität Darmstadt,
Germany; Lancaster University, UK
mezini@cs.tu-darmstadt.de

Abstract

Reactive programming improves the design of reactive applications by relocating the logic for managing dependencies between dependent values from the application logic to the application runtime. This is particularly useful for distributed applications.

continuously process incoming network packets fall into this category. Historically, reactivity has been achieved via callbacks and inversion of control [14], commonly implemented using the observer pattern to facilitate modular composition. While successful in decoupling and thus making composition easier, these approaches are not well suited for distributed applications.

[OOPSLA'14]

- Synchronous Semantics
- Decentralized
- Dynamic Edges



Thread-Safe Reactive Programming

JOSCHA DRECHSLER, Technische Universität Darmstadt, Germany
RAGNAR MOGK, Technische Universität Darmstadt, Germany
GUIDO SALVANESCHI, Technische Universität Darmstadt, Germany
MIRA MEZINI, Technische Universität Darmstadt, Germany

The execution of an application written in a reactive language involves transfer of data and control flow between imperative and reactive abstractions at well-defined points. In a multi-threaded environment, multiple such interactions may execute concurrently, potentially causing data races and event ordering ambiguities. Existing RP languages either disable multi-threading or handle it at the cost of reducing expressiveness or weakening consistency. This paper proposes a model for thread-safe reactive programming (RP) that ensures abort-free strict serializability under concurrency while sacrificing neither expressiveness nor consistency. We also propose an architecture for integrating a corresponding scheduler into the RP language runtime, such

[OOPSLA'18]

- Synchronous Semantics
- Fine-Grained Parallelism
- Dynamic Edges

A Fault-tolerant Programming Model for Distributed Interactive Applications

RAGNAR MOGK, Technische Universität Darmstadt

JOSCHA DRECHSLER, Technische Universität Darmstadt

GUIDO SALVANESCHI, Technische Universität Darmstadt

MIRA MEZINI, Technische Universität Darmstadt

Ubiquitous connectivity of web, mobile, and IoT computing platforms has fostered a variety of distributed applications with decentralized state. These applications execute across multiple devices with varying reliability and connectivity. Unfortunately, there is no declarative fault-tolerant programming model for distributed interactive applications with an inherently decentralized system model.

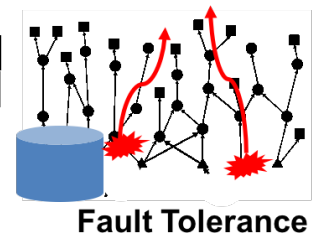
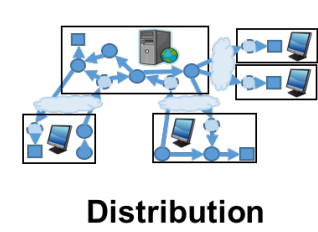
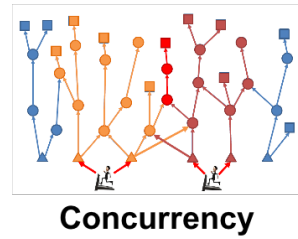
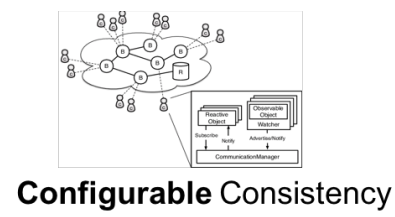
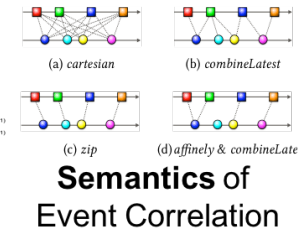
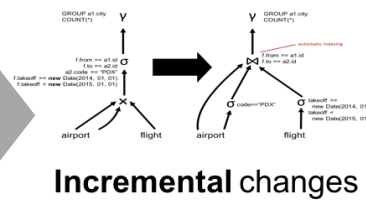
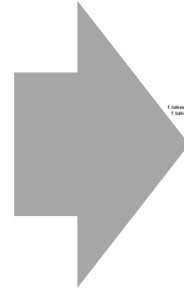
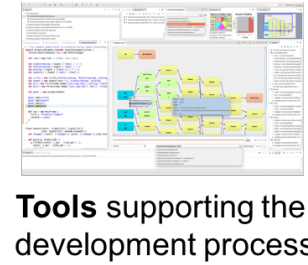
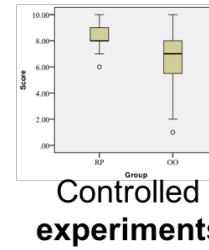
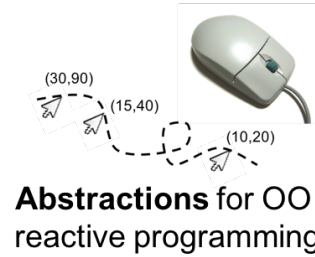
We present a novel approach to automating fault tolerance using high-level programming abstractions tailored to the needs of distributed interactive applications. Specifically, we propose a calculus that enables formal reasoning about applications' dataflow within and across individual devices. Our calculus reinterprets the functional reactive programming model to seamlessly integrate its automated state change propagation with automated crash recovery of device-local dataflow and disconnection-tolerant distribution with guaranteed automated eventual consistency semantics based on conflict-free replicated datatypes. As a result, programmers are relieved of handling intricate details of distributing change propagation and coping with distribution failures in the presence of connectivity. We also provide proofs of our claims, an implementation of our model, and a simulation using a common interactive application.

[OOPSLA'19]

- Full formalization
- CRDTs between graphs
- Recovery after disconnection

REScala is a Scala library for functional reactive programming on the JVM and the Web. It provides a rich API for event stream transformations and signal composition with managed consistent up-to-date state and minimal syntactic overhead. It supports concurrent and distributed programs.

Functional	Consistent	Concurrent
Abstractions for Events and Signals to handle interactions and state, and seamless conversions between them.	No temporary inconsistencies, no data races. Programmers define logical constraints which are automatically enforced by the runtime.	Concurrent applications are fully supported. Reactive abstractions can be safely accessed from any thread and they are updated concurrently.



www.rescala-lang.com

SOFTWARE DESIGN



Apache Flink® – Stateful Computations over Data Streams

What is Apache Flink?

Use Cases

Powered By

FAQ

Downloads

Tutorials

Documentation

Getting Help

Flink Blog

Community & Project Info

Roadmap

How to Contribute

Flink on GitHub

中文版

@ApacheFlink

Plan Visualizer

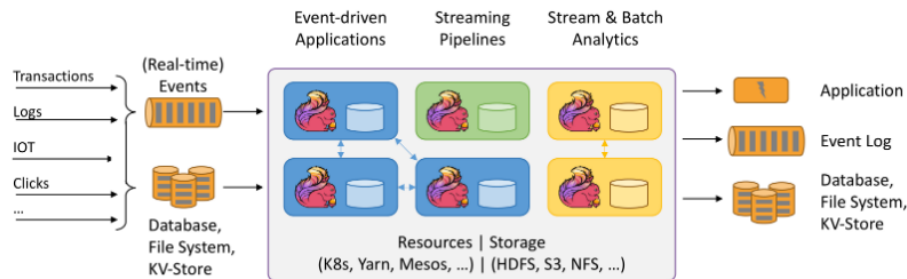
Apache Software Foundation

License

Security

Donate

Thanks



All streaming use cases

- Event-driven Applications
- Stream & Batch Analytics
- Data Pipelines & ETL

[Learn more](#)

Operational Focus

- Flexible deployment
- High-availability setup
- Savepoints

[Learn more](#)

✓ Guaranteed correctness

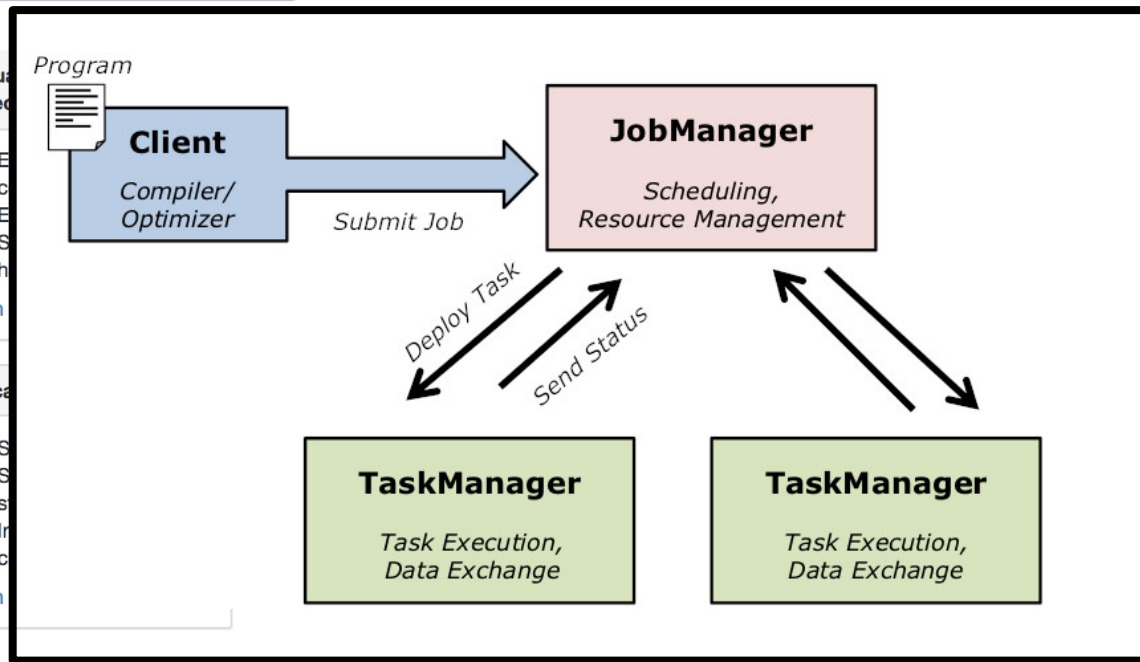
- Event-driven Applications
- Stream & Batch Analytics
- Stateful Stream Processing
- High Availability

[Learn more](#)

Scalable

- Stateful Stream Processing
- Stream & Batch Analytics
- Event-driven Applications
- High Availability

[Learn more](#)

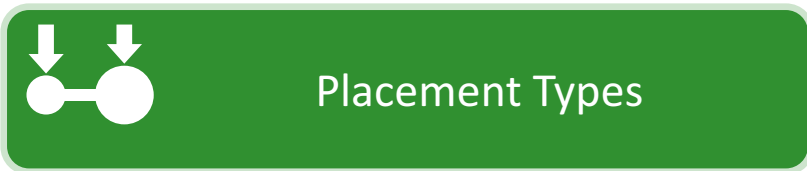
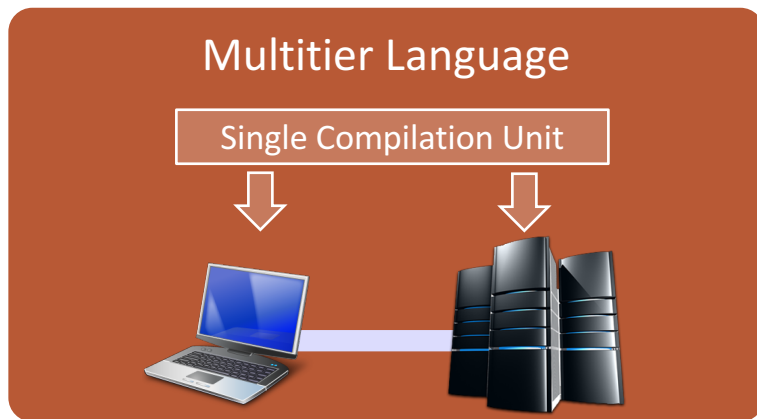


Powered by Flink



ScalaLoci Programming Framework

www.scala-loci.github.io



The screenshot shows the ScalaLoci website with the following content:

- ScalaLoci**: Research and development of language abstractions for distributed applications in Scala.
- Coherent**: Implement a cohesive distributed application in a single multitier language.
- Comprehensive**: Freely express any distributed architecture.
- Safe**: Enjoy static type-safety across components.

Two numbered steps are shown:

- 1 Specify Architecture**: Define the architectural relation of the components of the distributed system.

```
trait Server extends Peer {  
  type Tie = Multiple[Client]  
}  
trait Client extends Peer {  
  type Tie = Single[Server]  
}
```
- 2 Specify Placement**: Control where data is located and computations are executed.

```
val items = placed[Server] {  
  getCurrentItems()  
}  
val ui = placed[Client] {  
  showUI  
}
```

[P.Weisenbureger, M.Koehler, G.Salvaneschi, **Distributed System Development with ScalaLoci**, OOPSLA'18]

[P.Weisenbureger, G.Salvaneschi, **Multitier Modules**, ECOOP'19]

Placement Types

```
trait Registry extends Peer  
trait Node extends Peer
```

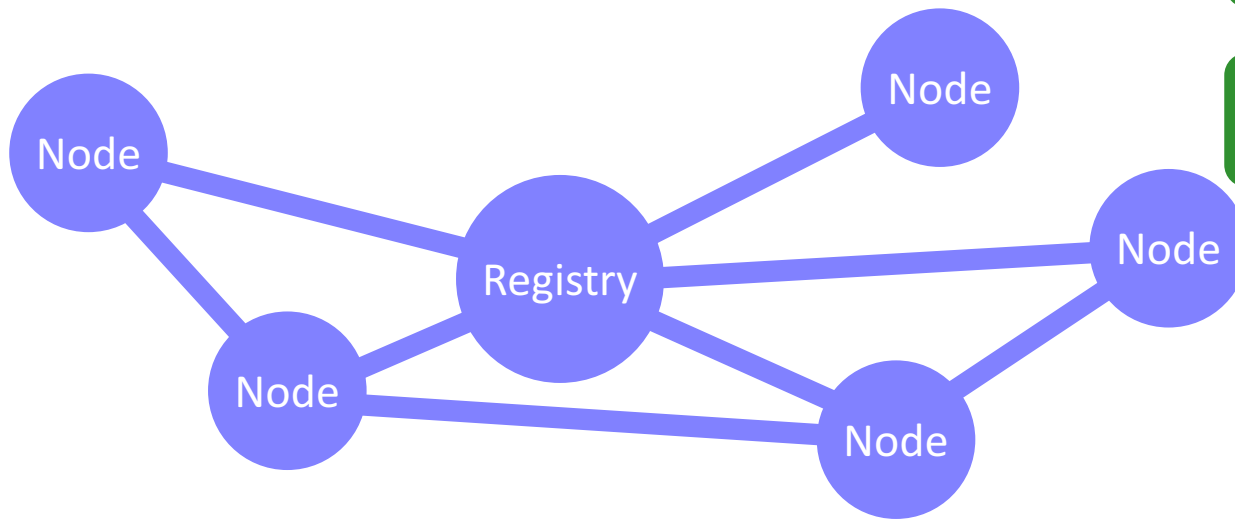
Peers

```
val message: Event[String] on Registry  
= placed { getMessageStream() }
```

Placement Types

Architecture

```
trait Registry extends Peer { type Tie = Multiple[Node] }  
trait Node extends Peer { type Tie = Single[Registry] with Multiple[Node] }
```

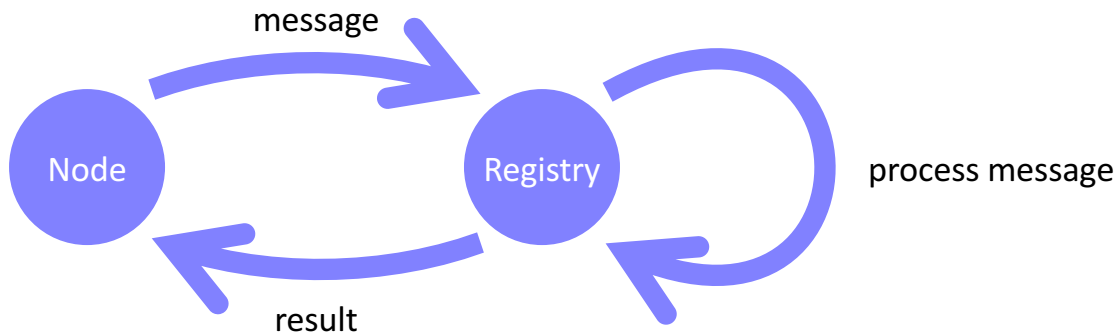


Architecture Specification
through Peer Types

Architecture-Based Remote
Access

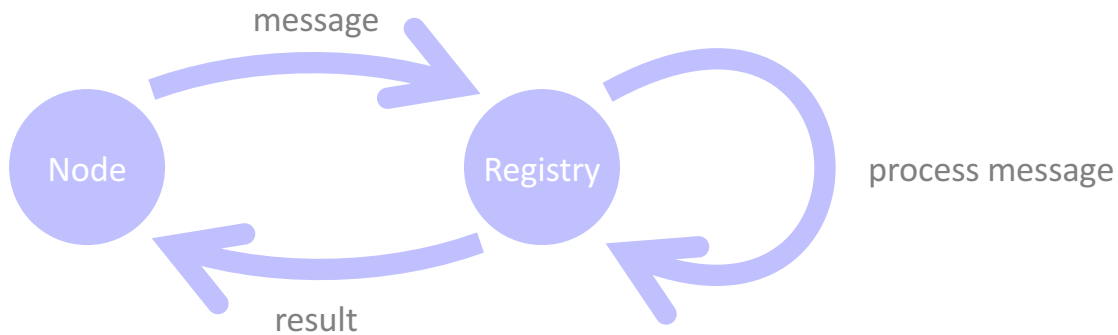
Data Flow

```
val message = Event[String]()  
val result = message map processMessage  
val ui = new UI(result)
```



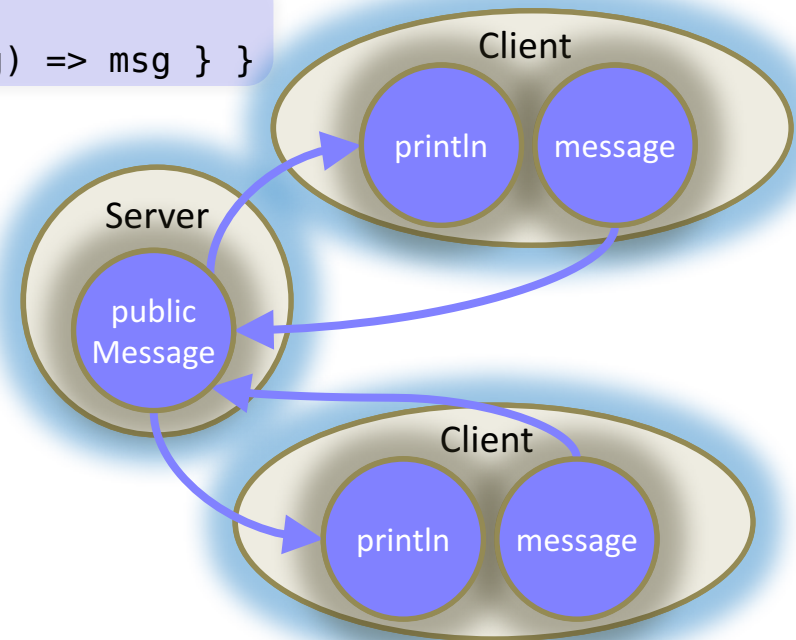
Distributed Data Flow

```
val message: Event[String] on Node = placed[Node] { Event[String]() }  
val result = placed[Registry] { message.asLocal map processMessage }  
val ui = placed[Node] { new UI(result.asLocal) }
```

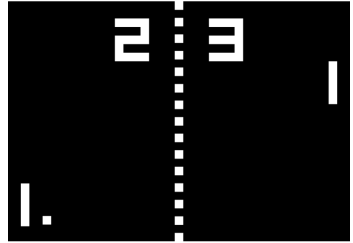


Complete Distributed Chat

```
@multitier object Chat {  
  trait Server extends Peer { type Tie = Multiple[Client] }  
  trait Client extends Peer { type Tie = Single[Server] }  
  
  val message = placed[Client] { Evt[String] }  
  
  val publicMessage = placed[Server] {  
    message.asLocalFromAllSeq map { case (_, msg) => msg } }  
  
  placed[Client].main {  
    publicMessage.asLocal observe println  
    for (line <- io.Source.stdin.getLines)  
      message fire line } }
```



Porting to Distribution



Local

```
val ballSize = 20
val maxX = 800
val maxY = 400
val leftPos = 30
val rightPos = 770
val initPosition = Point(400, 200)
val initSpeed = Point(10, 0)

val clientMouse = placedClient {
  Signal { UI.mousePosition(x, y) }

  val isPlaying = placedServerLocal {
    Signal { remoteClient.connected.size > 2 }
  }

  val ball: Signal[Point] = Server = placed {
    tick.fold(initPosition) { (ball, _) =>
      if (isPlaying.get) ball + speed.get else pos }
  }

  val areas = placedServerLocal {
    val racketY = Seq(
      Signal { UI.mousePosition(x, y) },
      Signal { ball(y) })
    val leftRacket = Racket(leftRacketPos, racketY(0))
    val rightRacket = Racket(rightRacketPos, racketY(1))
    val rackets = List(leftRacket, rightRacket)
    Signal { rackets map { _.area() } }
  }

  val leftWall = ball.changed && { _ .x < 0 }
  val rightWall = ball.changed && { _ .x > maxX }

  val yBounce = {
    val ballInRacket = Signal { areas() exists { _ contains ball() } }
    val collisionRacket = ballInRacket.changedTo true
    leftWall || rightWall || collisionRacket
  }
  val yBounce = ball.changed &&
    { ball => ball.y < 0 || ball.y > maxY }

  val speed = {
    val x = yBounce.toggle (initSpeed.x, -initSpeed.x)
    val y = yBounce.toggle (initSpeed.y, -initSpeed.y)
    Signal { Point(x(), y()) }
  }

  val score = {
    val leftPoints = rightWall.iterate(0) { _ + 1 }
    val rightPoints = leftWall.iterate(0) { _ + 1 }
    Signal { leftPoints() + " : " + rightPoints() }
  }

  val ui = new UI(areas, ball, score)
}
```

ScalaLoc

```
trait Server extends ServerPeer[Client]
trait Client extends ClientPeer[Server]

val ballSize = 20
val maxX = 800
val maxY = 400
val leftPos = 30
val rightPos = 770
val initPosition = Point(400, 200)
val initSpeed = Point(10, 0)

val clientMouse = placedClient {
  Signal { UI.mousePosition(x, y) }

  val isPlaying = placedServerLocal {
    Signal { remoteClient.connected.size > 2 }
  }

  val ball: Signal[Point] = Server = placed {
    tick.fold(initPosition) { (ball, _) =>
      if (isPlaying.get) ball + speed.get else pos }
  }

  val areas = placedServerLocal {
    val racketY = Seq(
      remoteClient.connected match {
        case left :: right :: _ => Seq(Some(left), Some(right))
        case _ => Seq(None, None) } })
    client = remoteClientFromClientLocal() getOrElse
      initPosition.y
  }
  val leftRacket = Racket(leftPos, Signal { racketY()(0) })
  val rightRacket = Racket(rightPos, Signal { racketY()(1) })
  val rackets = List(leftRacket, rightRacket)
  Signal { rackets map { _.area() } }
}

val leftWall = placedServerLocal { ball.changed && { _ .x < 0 } }
val rightWall = placedServerLocal { ball.changed && { _ .x > maxX } }

val yBounce = placedServerLocal {
  val ballInRacket = Signal { areas() exists { _ contains ball() } }
  val collisionRacket = ballInRacket.changedTo true
  leftWall || rightWall || collisionRacket
}
val yBounce = placedServerLocal { ball.changed &&
  { ball => ball.y < 0 || ball.y > maxY } }

val speed = placedServerLocal {
  val x = yBounce.toggle (initSpeed.x, -initSpeed.x)
  val y = yBounce.toggle (initSpeed.y, -initSpeed.y)
  Signal { Point(x(), y()) }
}

val score = placedServerLocal {
  val leftPoints = rightWall.iterate(0) { _ + 1 }
  val rightPoints = leftWall.iterate(0) { _ + 1 }
  Signal { leftPoints() + " : " + rightPoints() }
}

val ui = placedClient {
  new UI(areas, ball, ball, score, placed)
}
```

Akka

```
val ballSize = 20
val maxX = 800
val maxY = 400
val leftPos = 30
val rightPos = 770
val initPosition = Point(400, 200)
val initSpeed = Point(10, 0)

class Server extends Actor {
  def receive = addPlayer orElse mouseChanged
  val clients = Var[Seq.empty[ActorRef]]
  val mousePositions = Var[Map.empty[ActorRef, Int]]
  def mouseChanged: Receive = { case MouseChanged(y) =>
    mousePositions transform { _, sender => y } }
  val isPlaying = Signal { clients.size == 2 }

  val ball: Signal[Point] =
    tick.fold(initPosition) { (ball, _) =>
      if (isPlaying.get) ball + speed.get else ball }

  def addPlayer: Receive = { case AddPlayer =>
    clients transform { _, sender => } }

  val players = Signal {
    clients match {
      case left :: right :: _ => Seq(Some(left), Some(right))
      case _ => Seq(None, None) } }

  val areas = {
    val racketY = Signal {
      flatMap { mousePositions.get } getOrElse initPosition.y }
    val leftRacket = new Racket(leftRacketPos, Signal { racketY()(0) })
    val rightRacket = new Racket(rightRacketPos, Signal { racketY()(1) })
    val rackets = List(leftRacket, rightRacket)
    Signal { rackets map { _.area() } }
  }

  val leftWall = ball.changed && { _ .x < 0 }
  val rightWall = ball.changed && { _ .x > maxX }

  val yBounce = {
    val ballInRacket = Signal { areas() exists { _ contains ball() } }
    val collisionRacket = ballInRacket.changedTo true
    leftWall || rightWall || collisionRacket
  }
  val yBounce = ball.changed &&
    { ball => ball.y < 0 || ball.y > maxY }

  val speed = {
    val x = yBounce.toggle (Signal { initSpeed.x }, Signal { -initSpeed.x })
    val y = yBounce.toggle (Signal { initSpeed.y }, Signal { -initSpeed.y })
    Signal { Point(x(), y()) }
  }

  val score = {
    val leftPlayerPoints = rightWall.iterate(0) { _ + 1 }
    val rightPlayerPoints = leftWall.iterate(0) { _ + 1 }
    Signal { leftPlayerPoints() + " : " + rightPlayerPoints() }
  }

  areas observe { areas => clients.now foreach { _ ! updateAreas(areas) } }
  ball observe { ball => clients.now foreach { _ ! updateBall(ball) } }
  score observe { score => clients.now foreach { _ ! updateScore(score) } }

  clients observe { clients =>
    client ! UpdateReq(areas.now)
    client ! UpdateBall(ball.now)
    client ! UpdateScore(score.now) }

  abstract class Client(server: ActorSelection) extends Actor {
    val areas = Var[List.empty[Area]]
    val ball = Var[Point(0, 0)]
    val score = Var[0 : 0]

    mousePosition observe { pos =>
      server ! MouseChanged(pos.y) }

    val ui = new UI(areas, ball, score)

    def receive = {
      case UpdateAreas(areas) => this.areas.set(areas)
      case UpdateBall(ball) => this.ball.set(ball)
      case UpdateScore(score) => this.score.set(score)
    }
  }
}
```

RMI

```
val ballSize = 20
val maxX = 800
val maxY = 400
val leftPos = 30
val rightPos = 770
val initPosition = Point(400, 200)
val initSpeed = Point(10, 0)

remote trait Server {
  def addPlayer(client: Client): Unit
  def mouseChanged(client: Client, y: Int): Unit
}

class ServerImpl extends Server {
  val clients = Var[Seq.empty[Client]]
  val mousePositions = Var[Map.empty[Client, Int]]
  def mouseChanged(client: Client, y: Int) = synchronized {
    mousePositions() = mousePositions.get + (client -> y) }
  val isPlaying = Signal { clients.size == 2 }

  val ball: Signal[Point] =
    tick.fold(initPosition) { (ball, _) =>
      if (isPlaying.get) ball + speed.get else ball }

  def addPlayer(client: Client) = synchronized {
    clients transform { _, client => } }

  val players = Signal {
    clients match {
      case left :: right :: _ => Seq(Some(left), Some(right))
      case _ => Seq(None, None) } }

  val areas = {
    val racketY = Signal {
      flatMap { mousePositions.get } getOrElse initPosition.y }
    val leftRacket = new Racket(leftRacketPos, Signal { racketY()(0) })
    val rightRacket = new Racket(rightRacketPos, Signal { racketY()(1) })
    val rackets = List(leftRacket, rightRacket)
    Signal { rackets map { _.area() } }
  }

  val leftWall = ball.changed && { _ .x < 0 }
  val rightWall = ball.changed && { _ .x > maxX }

  val yBounce = {
    val ballInRacket = Signal { areas() exists { _ contains ball() } }
    val collisionRacket = ballInRacket.changedTo true
    leftWall || rightWall || collisionRacket
  }
  val yBounce = ball.changed &&
    { ball => ball.y < 0 || ball.y > maxY }

  val speed = {
    val x = yBounce.toggle (initSpeed.x, -initSpeed.x)
    val y = yBounce.toggle (initSpeed.y, -initSpeed.y)
    Signal { Point(x(), y()) }
  }

  val score = {
    val leftPoints = rightWall.iterate(0) { _ + 1 }
    val rightPoints = leftWall.iterate(0) { _ + 1 }
    Signal { leftPoints() + " : " + rightPoints() }
  }

  areas observe { updateAreas(areas) }
  ball observe { updateBall(ball) }
  score observe { updateScore(score) }

  clients observe { clients =>
    updateReq(areas)
    updateBall(ball)
    updateScore(score) }

  def updateAreas(areas: Seq[Area], list: List[Area]) =
    clients foreach { _ updateAreas(areas) }
  def updateBall(ball: Point) =
    clients foreach { _ updateBall(ball) }
  def updateScore(areas: Seq[Client], score: String) =
    clients foreach { _ updateScore(score) }
}

remote trait Client {
  def updateAreas(areas: List[Area]): Unit
  def updateBall(ball: Point): Unit
  def updateScore(score: String): Unit
}

class ClientImpl(server: Server) extends Client {
  val self = makeStubClient(this)

  val areas = Var[List.empty[Area]]
  val ball = Var[Point(0, 0)]
  val score = Var[0 : 0]

  UI.mousePosition observe { pos =>
    server mouseChanged (self, pos.y) }

  val ui = new UI(areas, ball, score)

  def updateAreas(areas: List[Area]) = synchronized { this.areas() = areas }
  def updateBall(ball: Point) = synchronized { this.ball() = ball }
  def updateScore(score: String) = synchronized { this.score() = score }

  server ! AddPlayer
}
```

multi-user support
distribution



Flink

```
class TaskManagerGateway {
  def receiveFromJobManager(instanceId: InstanceID, cause: Exception, mgr: ActorRef): Receive = {
    case Disconnect(instanceIdToDisconnect, cause) => {
      log.debug("Received disconnect message for wrong instance id " + instanceIdToDisconnect)
    }
    case StopCluster(applicationStatus, message) => {
      log.info("Stopping TaskManager with final application status " + shutdown())
    }
    case RequestTaskManagerLog(requestType) => {
      handleRequestTaskManagerLog(requestType, currentJobManager.get) match {
        case Left(message) => sendToJobManager(message)
        case Right(actorStatus.Failure(new IOException("ActorStatus not available, cannot upload taskManager logs."))) => {
          log.debug("Cannot find task to stop for execution $executionID")
          sendToJobManager(Status.Failure(t))
        }
        case _ => {
          log.debug("Cannot find task to cancel for execution $executionID")
          sendToJobManager(Acknowledge.get())
        }
      }
    }
    case TriggerCheckpoint(jobId, taskExecutionId, checkpointId, timestamp, checkpointOptions) => {
      log.debug("TaskManager received a checkpoint request " + s"for unknown task $taskExecutionId.")
    }
    case NotifyCheckpointComplete(jobId, taskExecutionId, checkpointId, timestamp) => {
      log.debug("TaskManager received a checkpoint confirmation " + s"for unknown task $taskExecutionId.")
    }
  }
}
```

```
class TaskManager extends Actor {
  def receiveFromTrace => sendStackTrace() foreach { message =>
    case _ => createMessage(message)
  }
  case Disconnect(instanceIdToDisconnect, cause) => {
    if (instanceIdToDisconnect.equals(instanceId)) {
      handleJobManagerDisconnect(jobManager requested disconnect: " + cause.getMessage())
      triggerTaskManagerRegistration()
    }
    log.debug("Received disconnect message for wrong instance id " + instanceIdToDisconnect)
  }
  case StopCluster(applicationStatus, message) => {
    log.info("Stopping TaskManager with final application status " + shutdown())
  }
  case RequestTaskManagerLog(requestType) => {
    diobService {
      case Left(message) => sendToJobManager(message)
      case Right(actorStatus.Failure(new IOException("ActorStatus not available, cannot upload taskManager logs."))) => {
        log.debug("Cannot find task to stop for execution $executionID")
        sendToJobManager(Status.Failure(t))
      }
      case _ => {
        log.debug("Cannot find task to cancel for execution $executionID")
        sendToJobManager(Acknowledge.get())
      }
    }
  }
  case TriggerCheckpoint(jobId, taskExecutionId, checkpointId, timestamp, checkpointOptions) => {
    log.debug("TaskManager received a checkpoint request " + s"for unknown task $taskExecutionId.")
  }
  case NotifyCheckpointComplete(jobId, taskExecutionId, checkpointId, timestamp) => {
    log.debug("TaskManager received a checkpoint confirmation " + s"for unknown task $taskExecutionId.")
  }
}
```

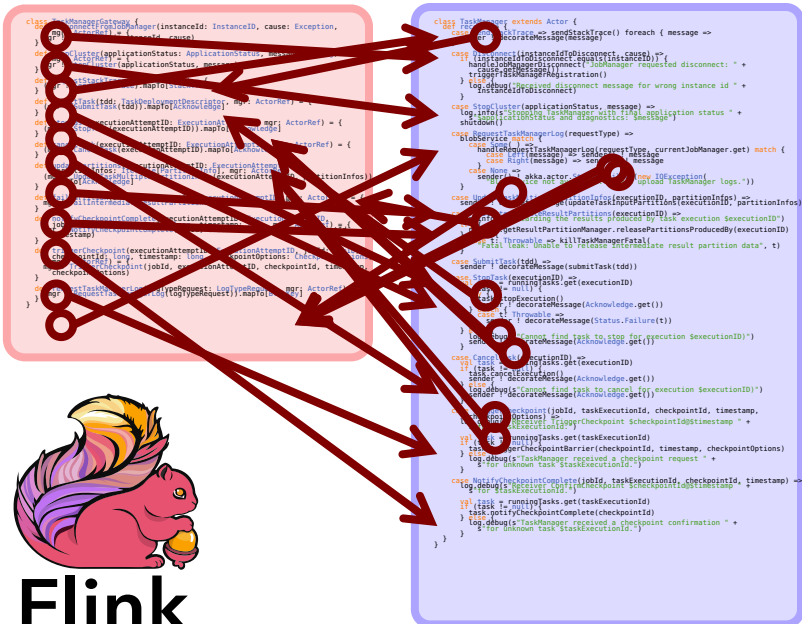
```
multitier (trait TaskManagerGatewayClusterTask {
  trait JobManager extends Peer { type Id <:= MultiId[TaskManager] }
  trait TaskManager extends Peer { type Id <:= Single[JobManager] }
  def disconnectFromJobManager(instanceId: InstanceID, cause: Exception, mgr: ActorRef): Receive = {
    case Disconnect(instanceIdToDisconnect, cause) => {
      handleJobManagerDisconnect(jobManager requested disconnect: " + cause.getMessage())
      triggerTaskManagerRegistration()
    }
    log.debug("Received disconnect message for wrong instance id " + instanceId)
  }
  def stopCluster(applicationStatus: ApplicationStatus, message: String, mgr: ActorRef): Receive = {
    log.info("Stopping TaskManager with final application status " + applicationStatus and diagnostics: $message)
    shutdown()
  }
  def requestTaskManagerLog(mgr: ActorRef, requestType: TaskManagerLogRequestType) => {
    diobService {
      case Left(message) => placed[JobManagerPeer] { mgr }
      case Right(actorStatus.Failure(new IOException("ActorStatus not available, cannot upload taskManager logs."))) => {
        log.debug("Cannot find task to stop for execution $executionID")
        placed[JobManagerPeer] { mgr }
      }
      case _ => {
        log.debug("Cannot find task to cancel for execution $executionID")
        placed[JobManagerPeer] { mgr }
      }
    }
  }
  def triggerCheckpoint(jobId: JobID, taskExecutionId: ExecutionAttemptID, checkpointId: Long, timestamp: Long, checkpointOptions: CheckpointOptions) => {
    log.debug("TaskManager received a checkpoint request " + s"for unknown task $taskExecutionId.")
  }
  def notifyCheckpointComplete(jobId: JobID, taskExecutionId: ExecutionAttemptID, checkpointId: Long, timestamp: Long) => {
    log.debug("TaskManager received a checkpoint confirmation " + s"for unknown task $taskExecutionId.")
  }
}
```

```
multitier (trait TaskManagerGatewayPartitionCheckOn {
  trait JobManager extends Peer { type Id <:= MultiId[TaskManager] }
  trait TaskManager extends Peer { type Id <:= Single[JobManager] }
  def updatePartitionInfo(executionAttemptID: ExecutionAttemptID, partitionInfos: java.lang.Iterable[PartitionInfo], mgr: ActorRef): Receive = {
    case UpdatePartitionInfo(executionAttemptID, partitionInfos) => {
      log.debug("TaskManager received a partition update request " + s"for unknown task $taskExecutionId.")
    }
  }
  def failPartition(executionAttemptID: ExecutionAttemptID, mgr: ActorRef): Receive = {
    case FailPartition(executionAttemptID) => {
      log.info("Discarding the results produced by task execution $executionID")
      network.getResultPartitionManager.releasePartitionsProducedBy(executionID)
    }
  }
  def notifyCheckpointComplete(executionAttemptID: ExecutionAttemptID, jobId: JobID, checkpointId: Long, timestamp: Long, checkpointOptions: CheckpointOptions) => {
    log.debug("TaskManager received a checkpoint confirmation " + s"for unknown task $taskExecutionId.")
  }
  def triggerCheckpoint(executionAttemptID: ExecutionAttemptID, jobId: JobID, checkpointId: Long, timestamp: Long, checkpointOptions: CheckpointOptions) => {
    log.debug("TaskManager received a checkpoint request " + s"for unknown task $taskExecutionId.")
  }
  def requestTaskManagerLog(logTypeRequest: LogTypeRequest, mgr: ActorRef): Receive = {
    case RequestTaskManagerLog(logTypeRequest) => {
      handleRequestTaskManagerLog(logTypeRequest, currentJobManager.get) match {
        case Left(message) => placed[JobManagerPeer] { mgr }
        case Right(actorStatus.Failure(new IOException("ActorStatus not available, cannot upload taskManager logs."))) => {
          log.debug("Cannot find task to stop for execution $executionID")
          placed[JobManagerPeer] { mgr }
        }
        case _ => {
          log.debug("Cannot find task to cancel for execution $executionID")
          placed[JobManagerPeer] { mgr }
        }
      }
    }
  }
}
```

Eliminated 23 non-exhaustive pattern matches and 8 type casts



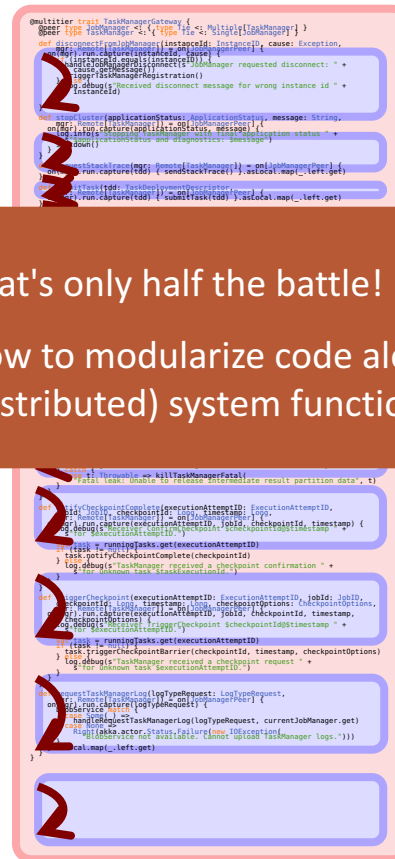
Crosscutting functionality separated among compilation units



Flink



Developers are **not** forced to modularize along network boundaries



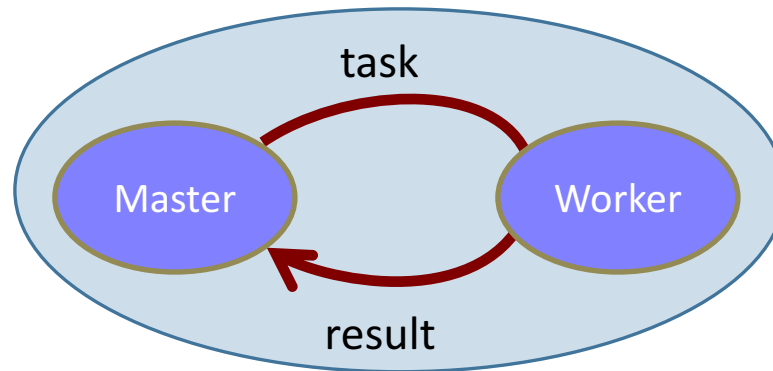
That's only half the battle!

How to modularize code along (distributed) system functionalities?

Multitier Modules

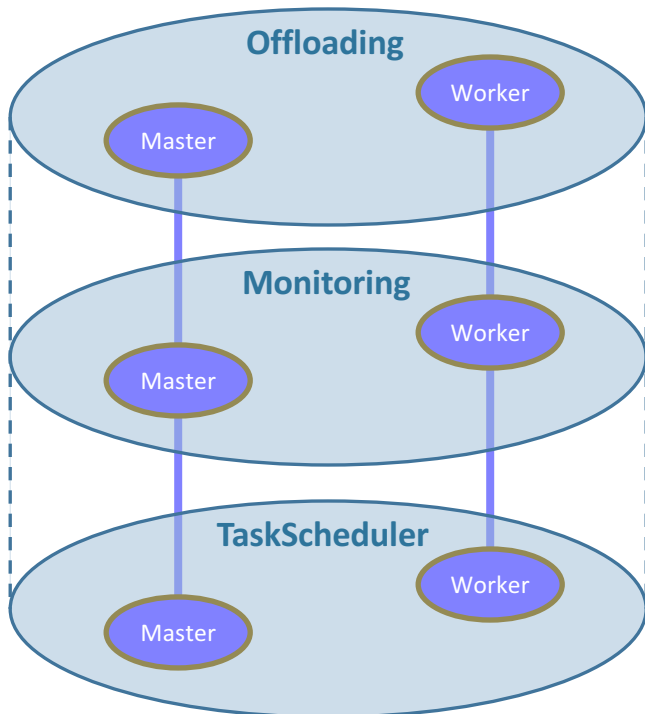
Distributed functionality = Module

Composing modules = Composing subsystems



Handle large
code bases

Stacking Multitier Modules



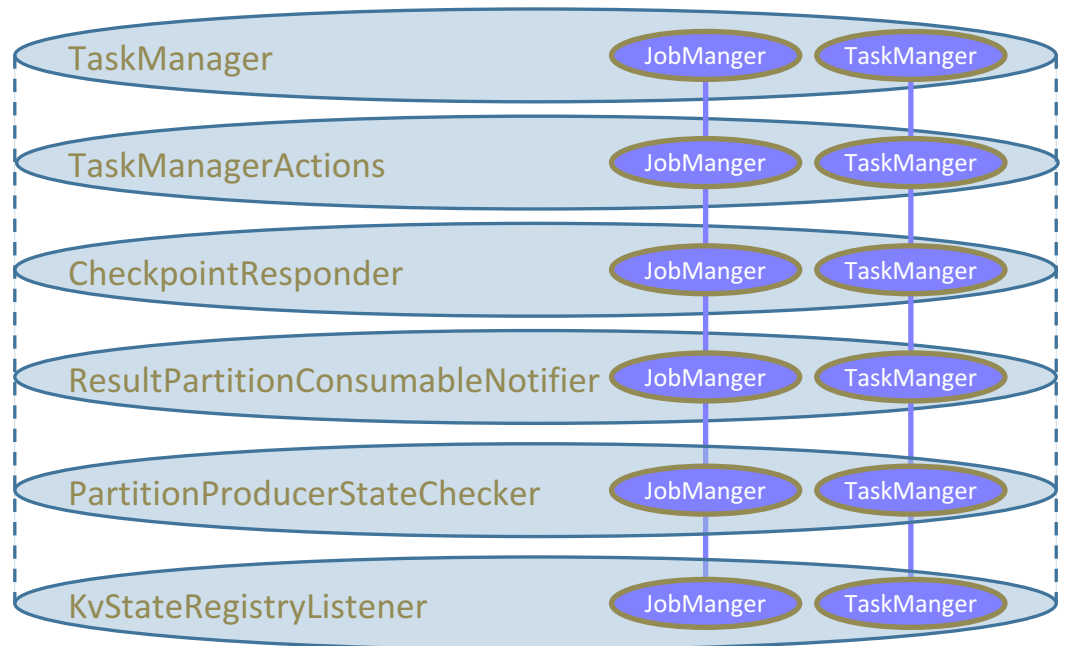
```
@multitier trait Offloading[T] {  
  @peer type Master <: { type Tie <: Multiple[Worker] }  
  @peer type Worker <: { type Tie <: Single[Master] }  
  def run(task: Task[T]): Future[T] on Master =  
    placed { (remote(selectWorker()) call execute(task)).asLocal }  
  private def execute(task: Task[T]): T on Worker =  
    placed { task.process() }  
}
```

```
@multitier trait Monitoring {  
  @peer type Master <: { type Tie <: Multiple[Worker] }  
  @peer type Worker <: { type Tie <: Single[Master] }  
  def monitoredTimedOut(monitored: Remote[Worker]): Unit on Master  
}
```

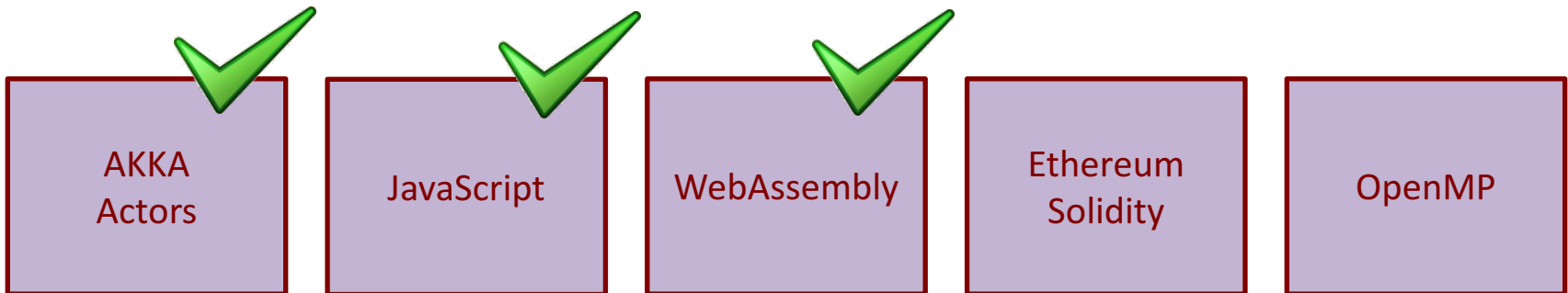
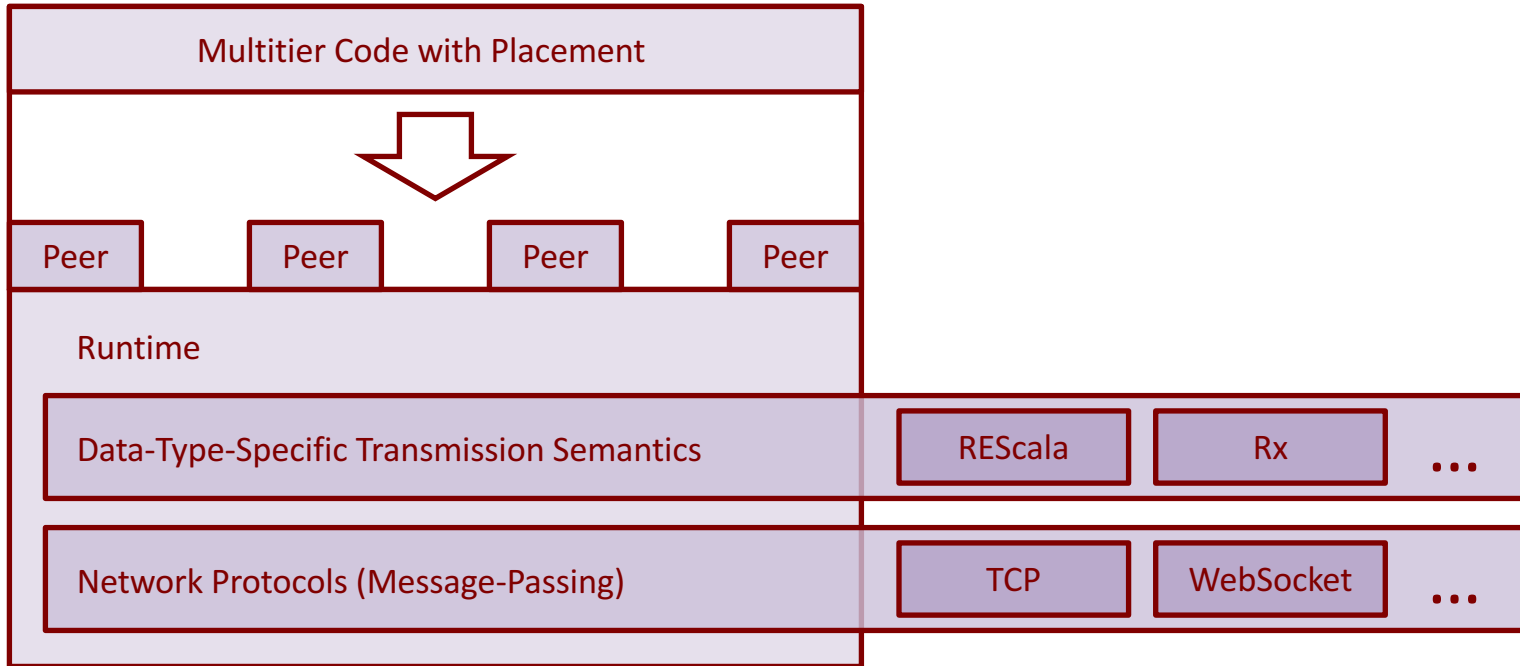
```
@multitier trait TaskScheduler[T] extends  
  Offloading[T] with  
  Monitoring
```

Flink Case Study

```
@multitier object TaskDistributionSystem extends  
  TaskManager with  
  TaskManagerActions with  
  CheckpointResponder with  
  ResultPartitionConsumableNotifier with  
  PartitionProducerStateChecker with  
  KvStateRegistryListener
```



ScalaLoc: Backends





Distributed System Development with SCALALOCI

PASCAL WEISENBURGER, Technische Universität Darmstadt, Germany
MIRKO KÖHLER, Technische Universität Darmstadt, Germany
GUIDO SALVANESCHI, Technische Universität Darmstadt, Germany

Distributed applications are traditionally developed as separate modules, often in different languages, which react to events, like user input, and in turn produce new events for the other modules. Separating concerns requires time-consuming integration. Manual implementation of communication forces developers to deal with low-level details. The combination of the two results in obscure distributed data flows among multiple modules, hindering reasoning about the system as a whole.

The SCALALOCI distributed programming language addresses these issues with a coherent model based on placement types that enables reasoning about distributed data flows, supporting multiple software components via dedicated language features and abstracting over low-level communication details and code. As we show, SCALALOCI simplifies developing distributed systems, reduces error-prone communication, and favors early detection of bugs.

CCS Concepts: • **Software and its engineering** → **Distributed programming languages**; • **Theory of computation** → *Distributed computing models*;

Fault Tolerance

Dynamic Topologies

Design Metrics

Microbenchmarks

Multiple Backends

Formalization

ScalaLoci
Research and development of language abstractions for distributed applications in Scala

- Coherent**: Implement a cohesive distributed application in a single multiter language
- Comprehensive**: Freely express any distributed architecture
- Safe**: Enjoy static type-safety across components

- Specify Architecture**: Define the architectural relation of the components of the distributed system


```
trait Server extends Peer {
  type Tie = Multiple[Client]
}
trait Client extends Peer {
  type Tie = Single[Server]
}
```
- Specify Placement**: Control where data is located and computations are executed


```
val items = placed[Server] {
  getCurrentItems()
}
val ui = placed[Client] {
  new UI
}
```

Programming, Reactive
distributed System Development with
(October 2018), 30 pages. <https://doi.org/10.1145/3211111>

Thank you

QUESTIONS?



@guidosalva

www.guidosalvaneschi.com