Functional Reactive Programming

Bob Reynders

PLaNES, 4 Februari 2015

Functional Reactive Programming

- **Event** discrete values
  - `Mouse.click: Event[Click]`
  - $\approx$ `List[(Time, A)]`
- **Behavior** time-varying values
  - `Mouse.position: Behavior[Coordinate]`
  - $\approx$ `Time => A`

FRP API (subset)

```
Event[T].map[A](f: T => A): Event[A]
Beh[T].map[A](f: T => A): Beh[A]

// Filter
Event[T].keepIf(f: T => Boolean): Event[T]

// Lift
Beh[T].combine[A, B](b: Beh[A])(f: (T, A) => B): Beh[B]
Event[T].merge(e: Event[T]): Event[T]

// Sampling
Beh[T].sampledBy(e: Event[_]): Event[T]

// State
Event[T].fold[A](initial: A)(fun: (A, T) => A): Beh[A]
```

GUI Example

```
// input
val plusE: Event[Int] = plusButton.click.map(_ => 1)
val minE: Event[Int] = minButton.click.map(_ => -1)
val merged: Event[Int] = plusE.merge(minE)

// state
val counter: Beh[Int] = merged.fold(0) { (acc, e) => acc + e }

// view
def template(data: Int): GUI = ...
val main: Beh[GUI] = counter.map(template)
```

Static Dependency Graph

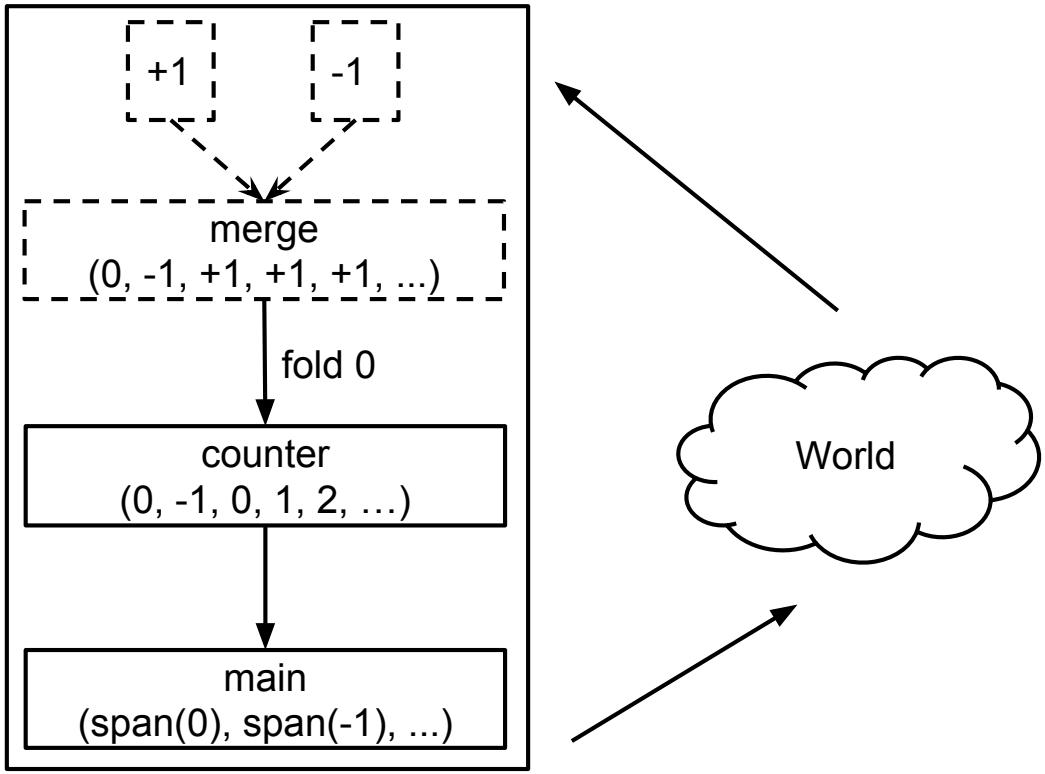


Figure 1: Static graph

- First order FRP
- Static guarantees

Dynamic Dependency Graph

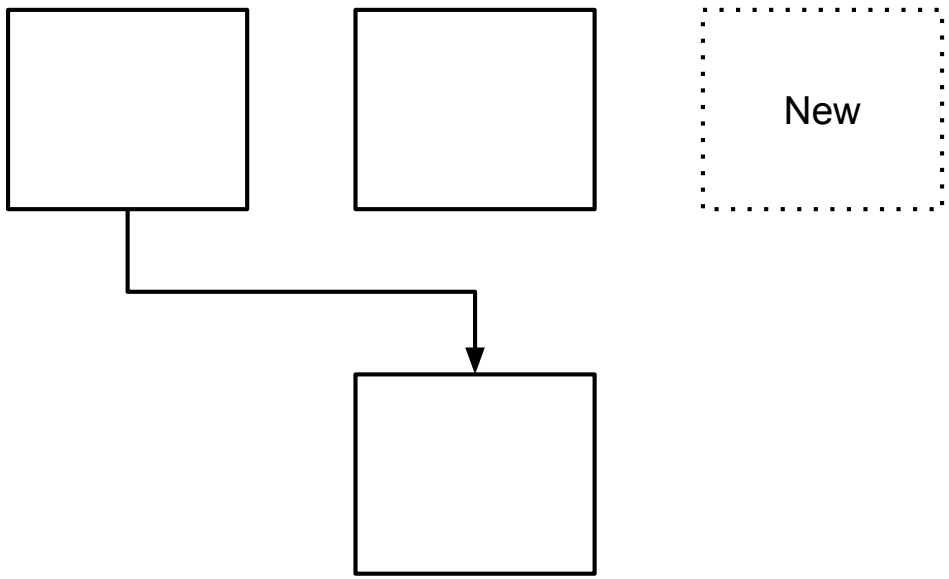
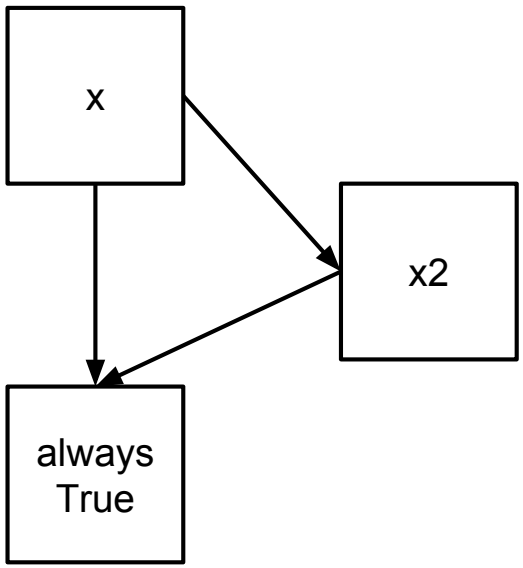


Figure 2: Switch

- Higher order FRP
- `Event[Behavior[T]].switch: Behavior[T]`

Glitch-freedom




```
val x: Beh[Int]
val x2 = x.map(_ * 2)
val alwaysTrue = x.combine(x2)(_ <= _)
```

Multi-tier Functional Reactive Programming

Multi-tier Functional Reactive Programming

```
// One language
ServerEvent[T].map[A](f: T => A): ServerEvent[A]
ClientEvent[T].map[A](f: Rep[T] => Rep[A]): ClientEvent[A]
...
```

- Embed Javascript as a DSL in Scala
  - Rep[String] = (Client) Javascript String
  - String = (Server) Scala String

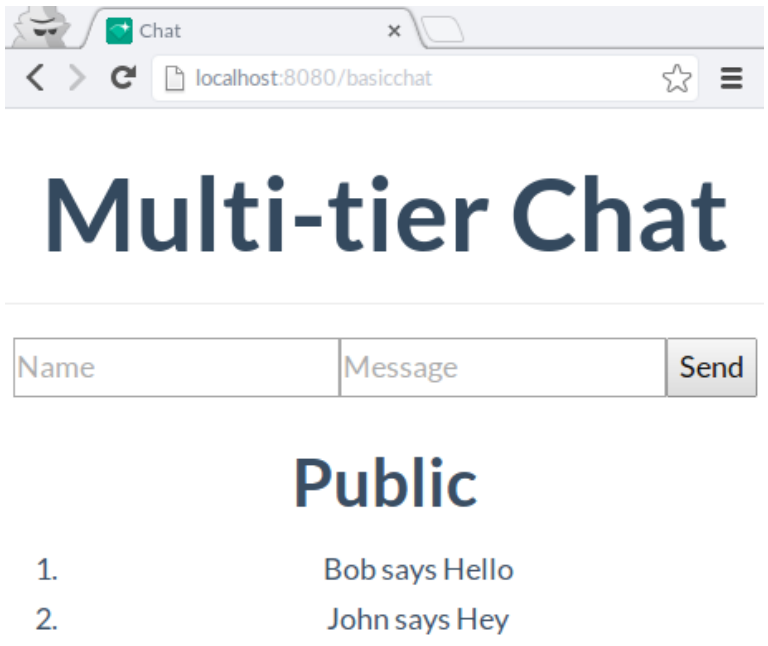
Multi-tier API Extension

```
// Replication between tiers
ClientEvent[T].toServerAnon: ServerEvent[T]
ServerEvent[T].toAllClients: ClientEvent[T]

// Server -> Client
ServerBeh[T].toAllClients: ClientBeh[T]
```

Chat Application

```
// prepare input
val nameB: ClientBeh[String] = nameInput.text
val msgB: ClientBeh[String] = msgInput.text
val clickE: ClientEvent[MouseData] = sendBtn.click

// model entries (across tiers!)
val entryB = nameB.combine(msgB)(_ + ": " + _)
val submitE = entryB.sampledBy(clickE).toServerAnon

// model server application state
val chatB: ServerBeh[List[String]] =
    submitE.fold(List.empty[String]) {
      (acc, entry) => entry :: acc
    }

// view (template on the client)
def template(view: Rep[List[String]]): Rep[GUI] = ...
val main: ClientBeh[GUI] = chatB.toAllClients.map(template)
```

Chat Application



Multi-tier API Extension

```
// Client-aware replication
// - Track sources
ClientEvent[T].toServer: ServerEvent[(Client, T)]
ServerEvent[Client => Option[T]].toClient: ClientEvent[T]

ServerBeh[Client => T].toClient: ClientBeh[T]
```

Multi-tier API Extension

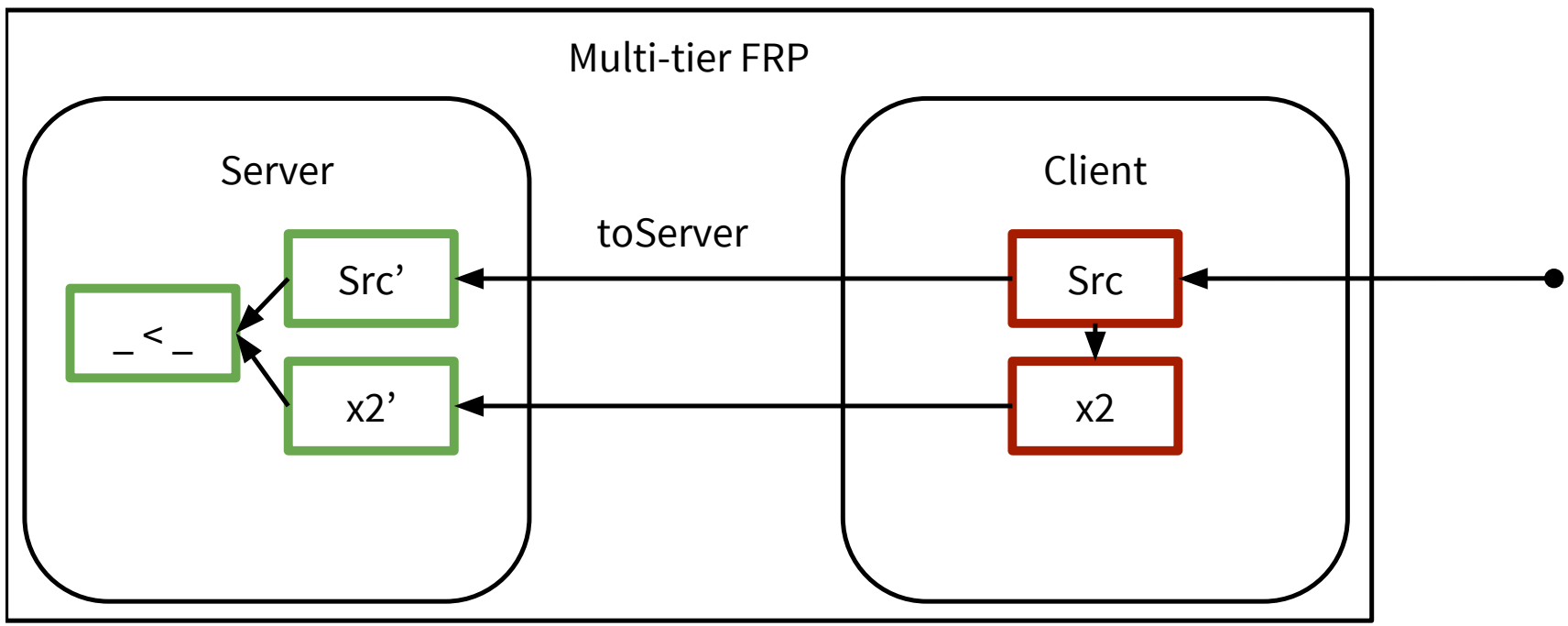
```
// Generalise to categories of programs
Event[C1, T].anonTo[C2]: Event[C2, T]

// - Track sources
Event[C1, Id[C2] => Option[T]].to[C2]: Event[C2, (Id[C1], T)]
```

Glitch-freedom

```
val src: ClientEvent[Int]
val x2 = src.map(_ * 2).toServer
val alwaysTrue = src.hold(0).combine(x2.hold(0))(_ < _)
```

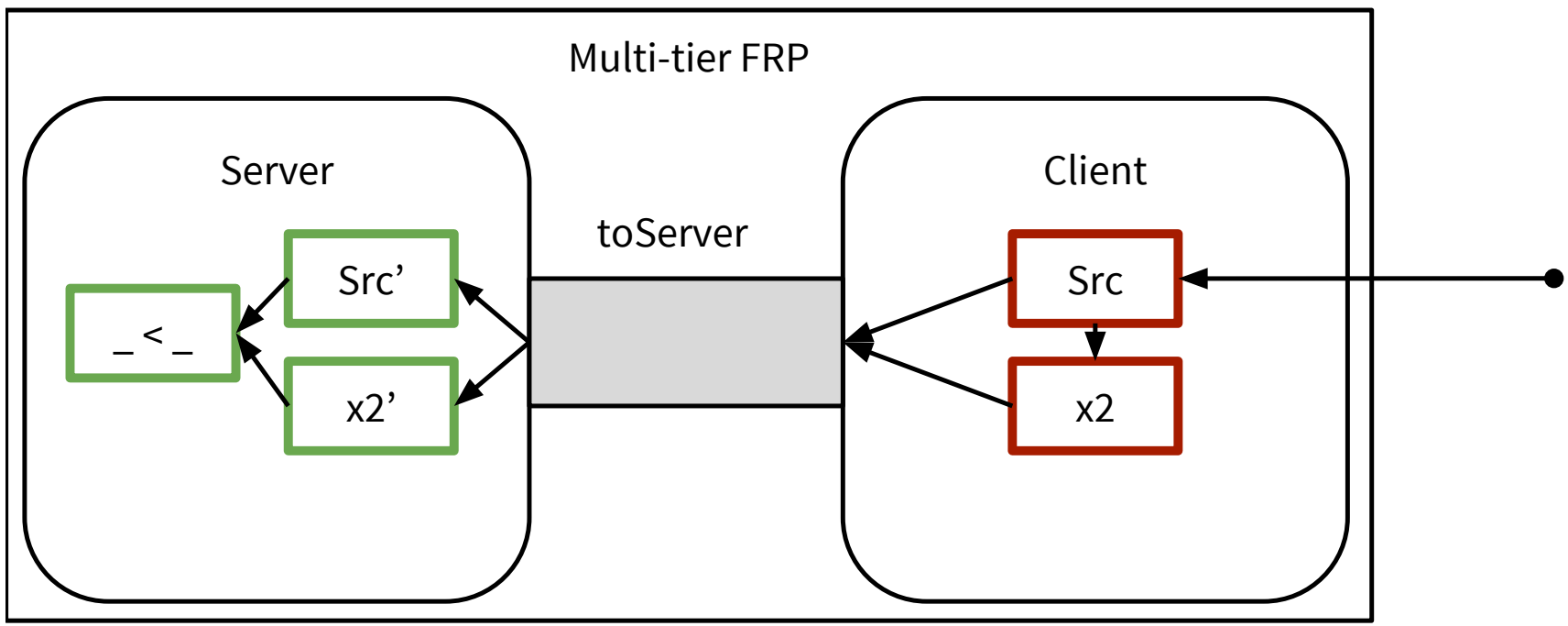
Glitch-freedom: naive

- 1 propagation on the client $\rightarrow$ 2 on the server



Glitch-freedom: glitch-free tiers

- 1 propagation on the client $\rightarrow$ 1 on the server



Glitch-freedom: total

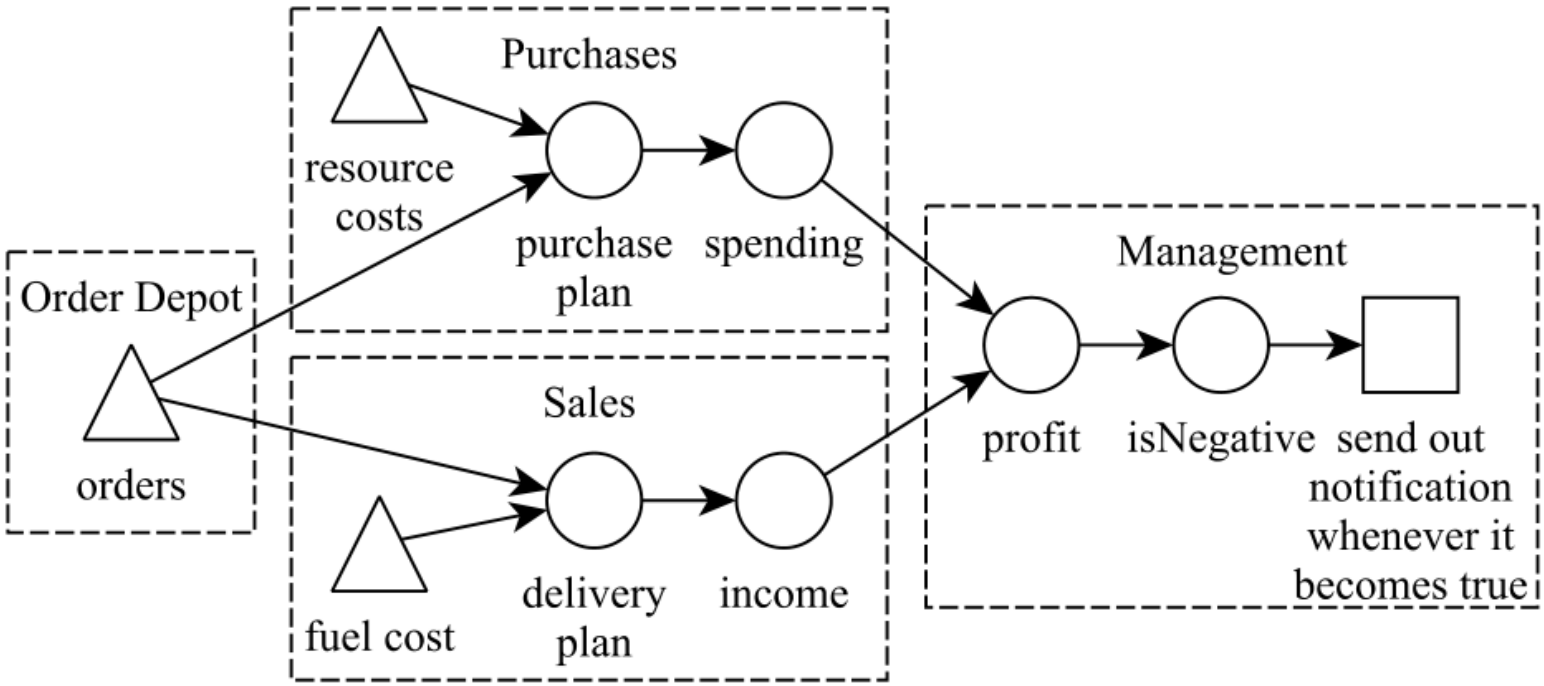


Figure 3: Distributed REScala: An Update Algorithm for Distributed Reactive Programming

Network Overhead

```
val serverB: Behavior[List[Behavior[Int]]]
// changes of Int

val serverB: Behavior[List[Int]]
// changes of List[Int]

val incServerB: IncBehavior[List[Int], Delta]
// changes of Delta
```