

# Solving the Overproduction Problem in Reactive Programming using Feedback Control

Master Thesis Defense

Richard van Heest

# Interactive vs. Reactive

## Interactive program

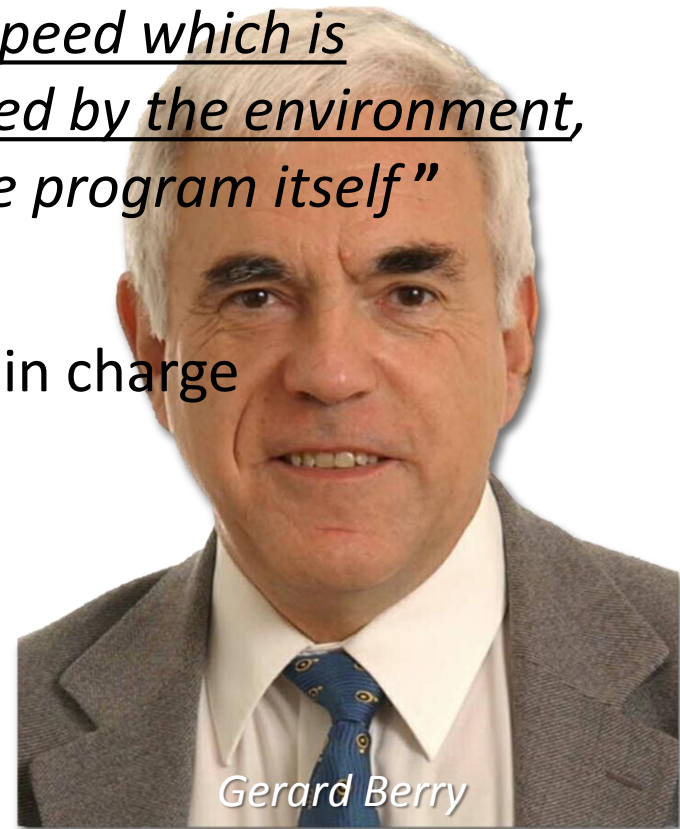
*“interacts at its own speed with users or with other programs”*

consumer in charge

## Reactive program

*“maintains a continuous interaction with its environment, but at a speed which is determined by the environment, not by the program itself”*

producer in charge



Gerard Berry

# Interactive vs. Reactive

## Interactive program

```
val it = Iterable(0, 1, 2, 3)
```

```
val iterator = it.iterator  
while (iterator.hasNext)  
    println(iterator.next())
```

```
for (i <- it)  
    println(i)
```

```
it.foreach(println)
```

## Reactive program

```
val obs = Observable(0, 1, 2, 3)
```

```
obs.subscribe(println(_))
```

# Interactive vs. Reactive

## Interactive program

```
val it = Iterable(0, 1, 2, 3)
```

```
val iterator = it.iterator  
while (iterator.hasNext)  
    println(iterator.next())
```

```
for (i <- it)  
    println(i)
```

```
it.foreach(println)
```

```
it.map(i => i + 1)  
    .filter(i => i % 2 == 0)  
    .foreach(println)
```

## Reactive program

```
val obs = Observable(0, 1, 2, 3)
```

```
obs.subscribe(println(_))
```

```
obs.map(i => i + 1)  
    .filter(i => i % 2 == 0)  
    .subscribe(println(_))
```

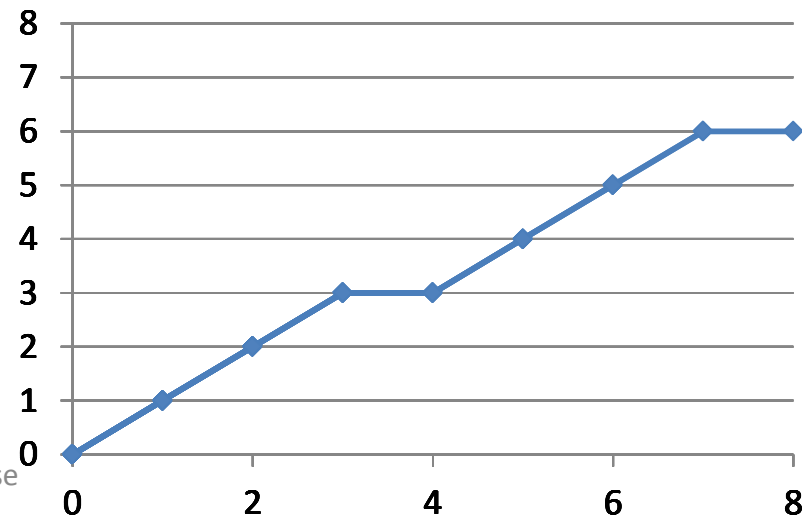
# Fast producer, slow consumer

producer

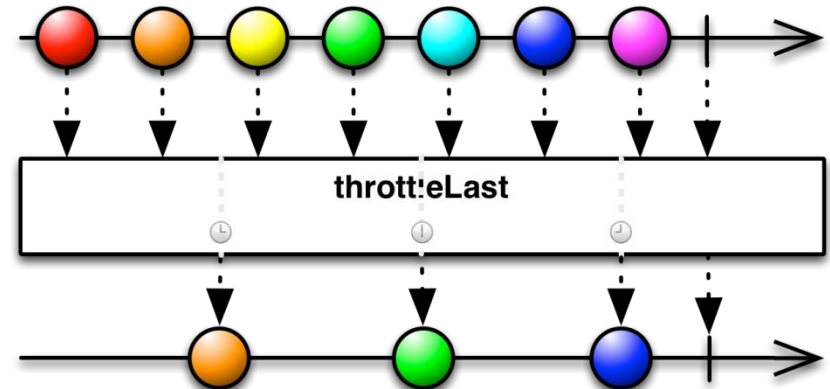
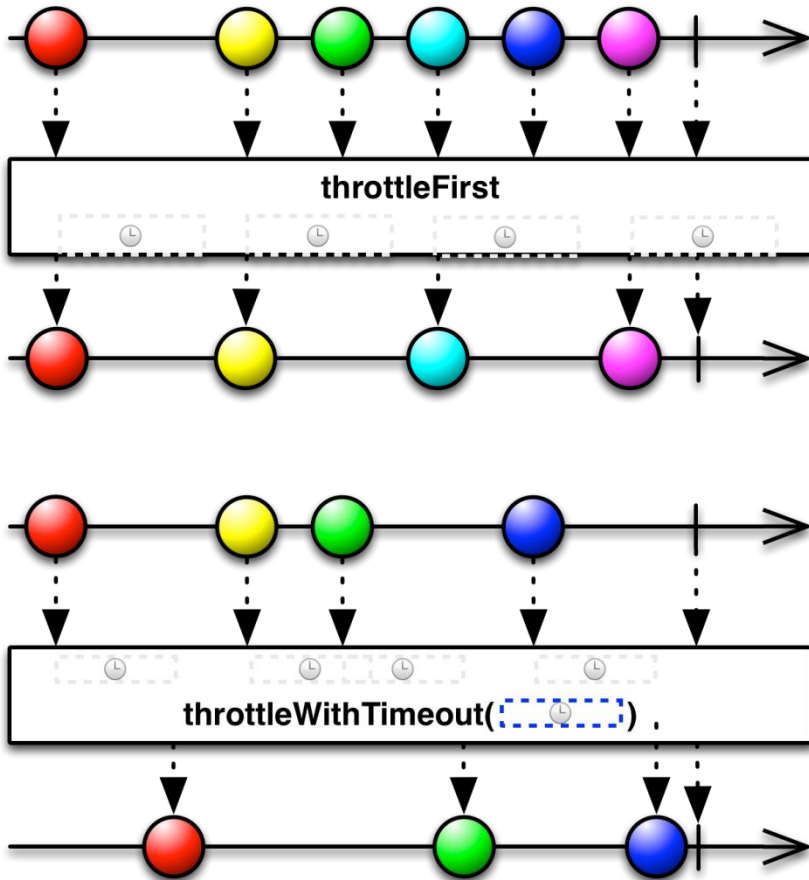


consumer

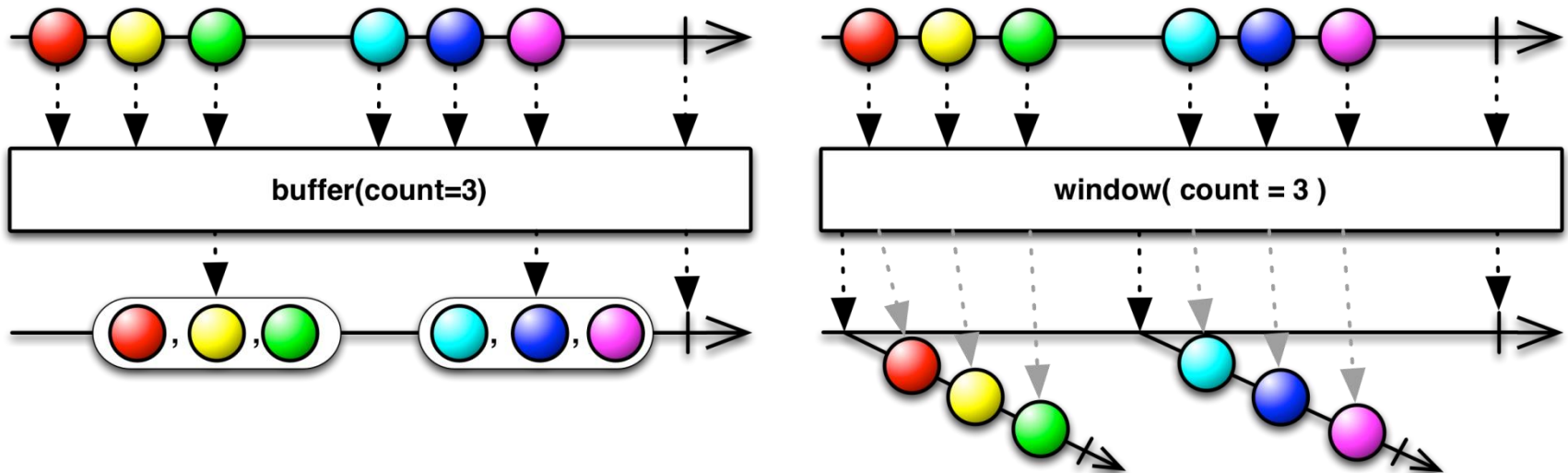
time = 8



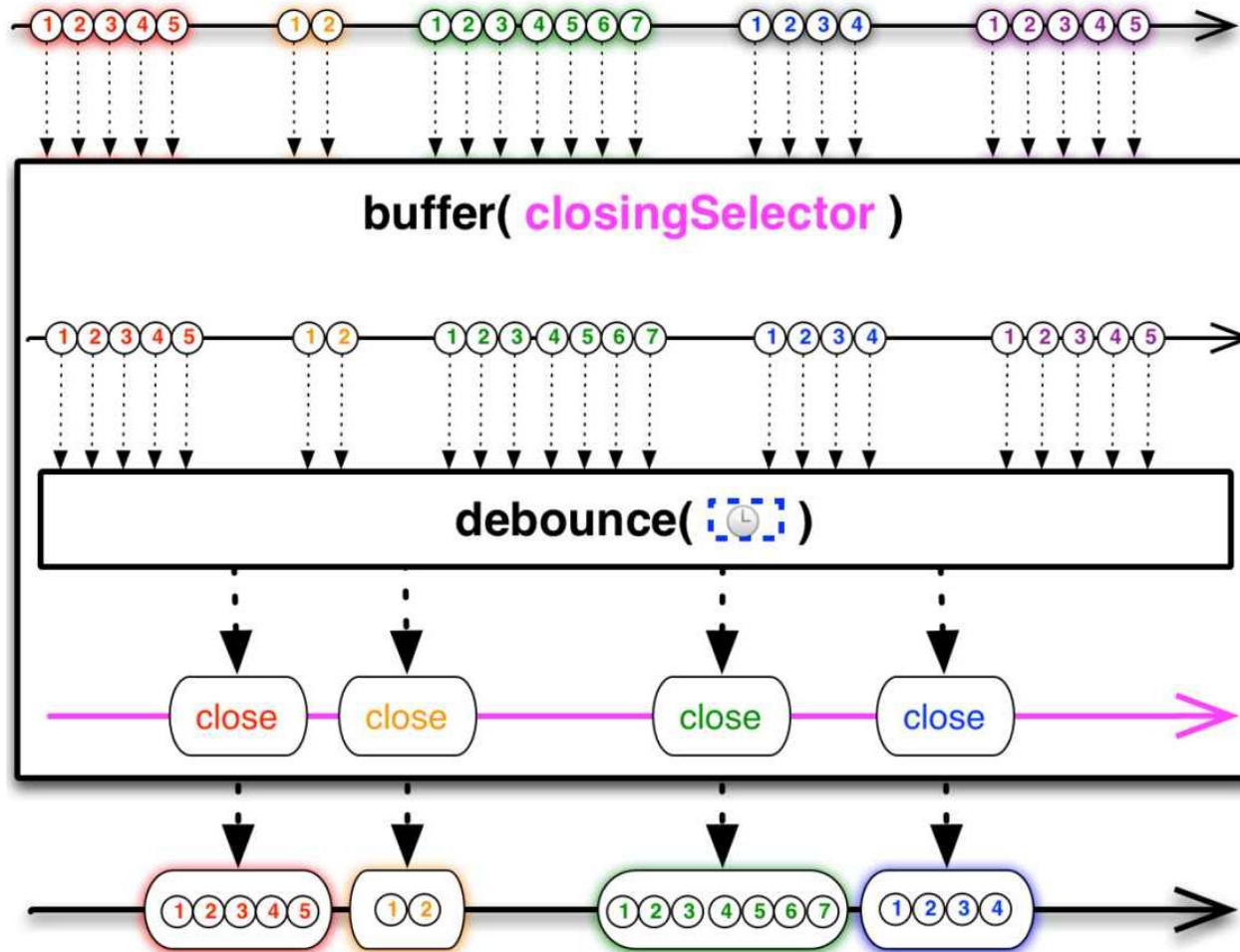
# Lossy operators



# Loss-less operators



# Combined lossy & lossless operators





# The source matters!

## Cold source

- Can be interacted with
- Multiple subscribers:  
stream duplicates
- Examples:
  - Database query result
  - Network response
  - In-memory list of data

## Hot source

- Can't be interacted with
- Multiple subscribers:  
stream continues  
(broadcast)
- Examples:
  - Mouse events
  - Stock tickers
  - Clocks

# The source matters!

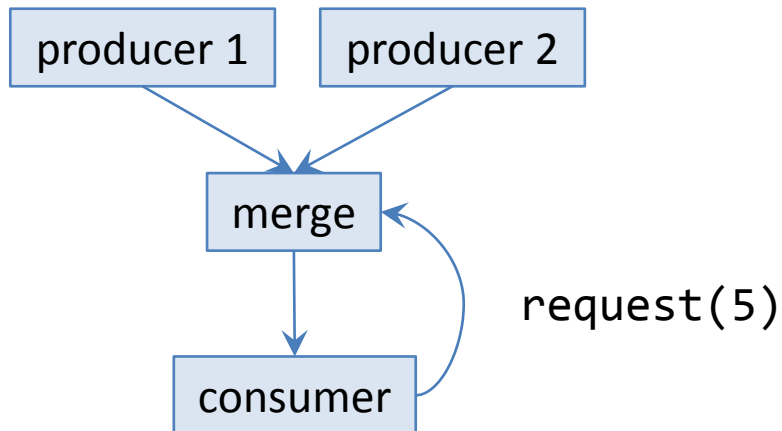
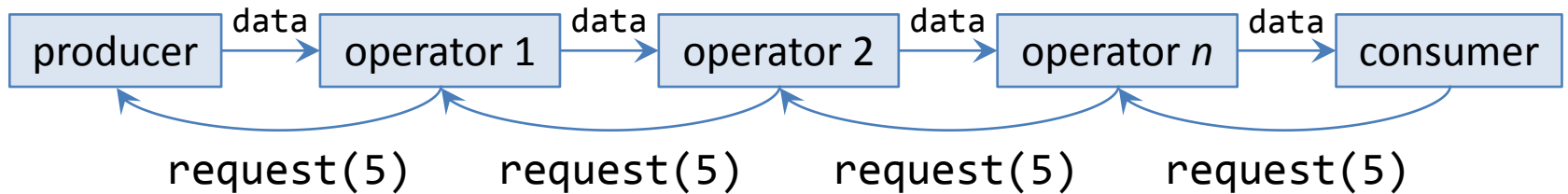
## Cold async source

- Can be interacted with
- Dependent on a notion of time
- Examples:
  - Database query result
  - Network response

## Cold sync source

- Can be interacted with
- Not dependent on a notion of time
- Examples:
  - In-memory list of data
  - *Observable*(0, 1, 2, 3)

# Backpressure/Reactive Streams



```
producer1.merge(producer2)  
    .subscribe(println(_))
```

# Interactive vs. Reactive

## Interactive program

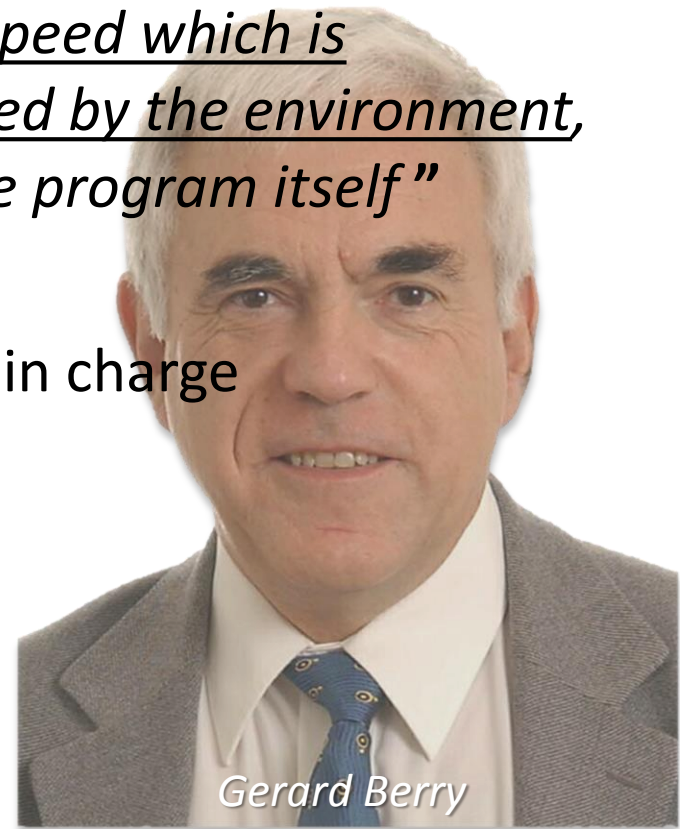
*“interacts at its own speed with users or with other programs”*

consumer in charge

## Reactive program

*“maintains a continuous interaction with its environment, but at a speed which is determined by the environment, not by the program itself”*

producer in charge



Gerard Berry

# Let's backtrack

## Cold source

- Can be interacted with
- Maybe dependent on time
- Backpressure
  - Not reactive
  - Affects operators

## Hot source

- Can't be interacted with
- Dependent on time
- Lossy and lossless operators

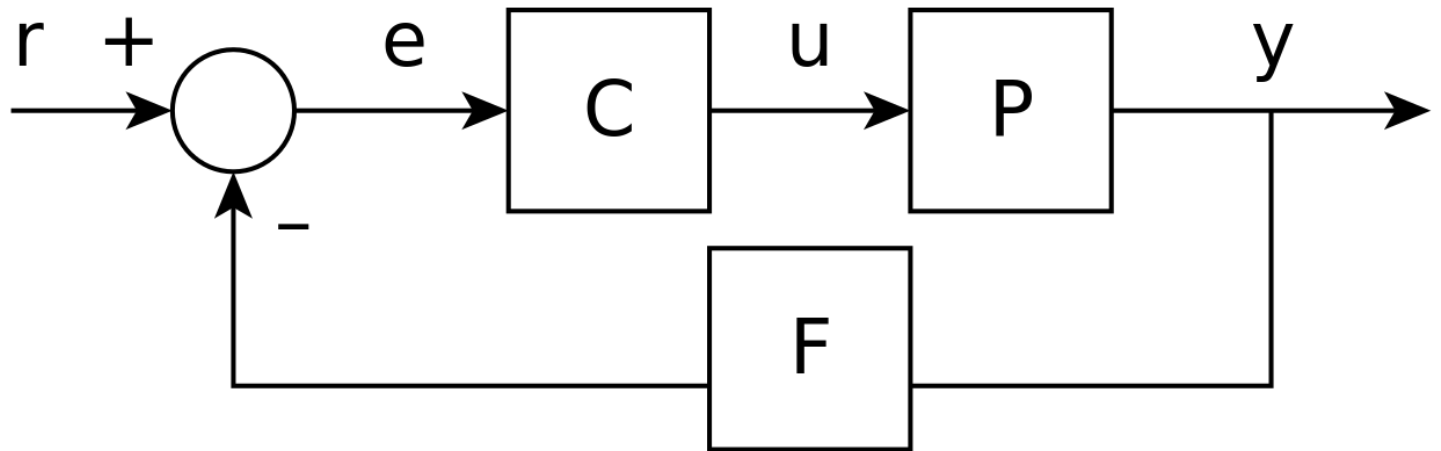
## Our solution

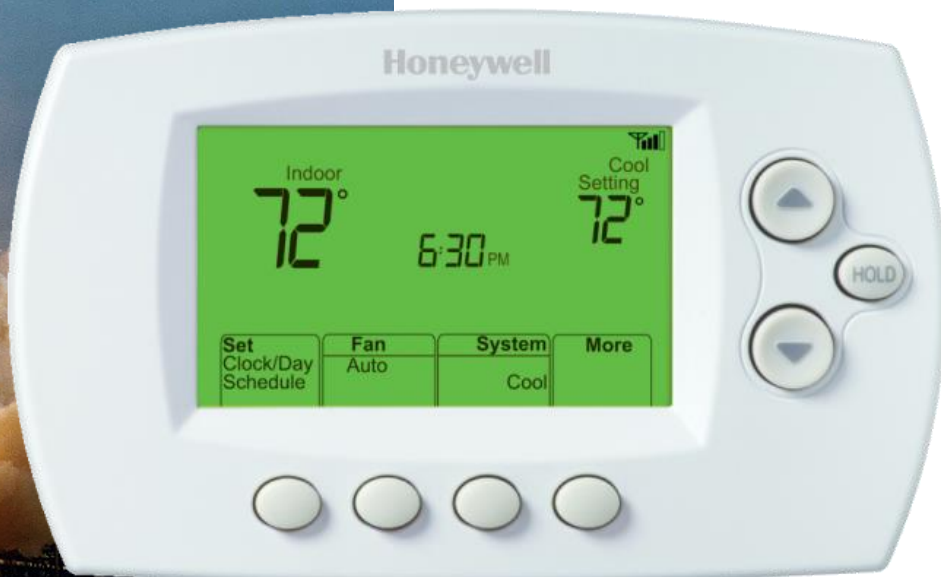
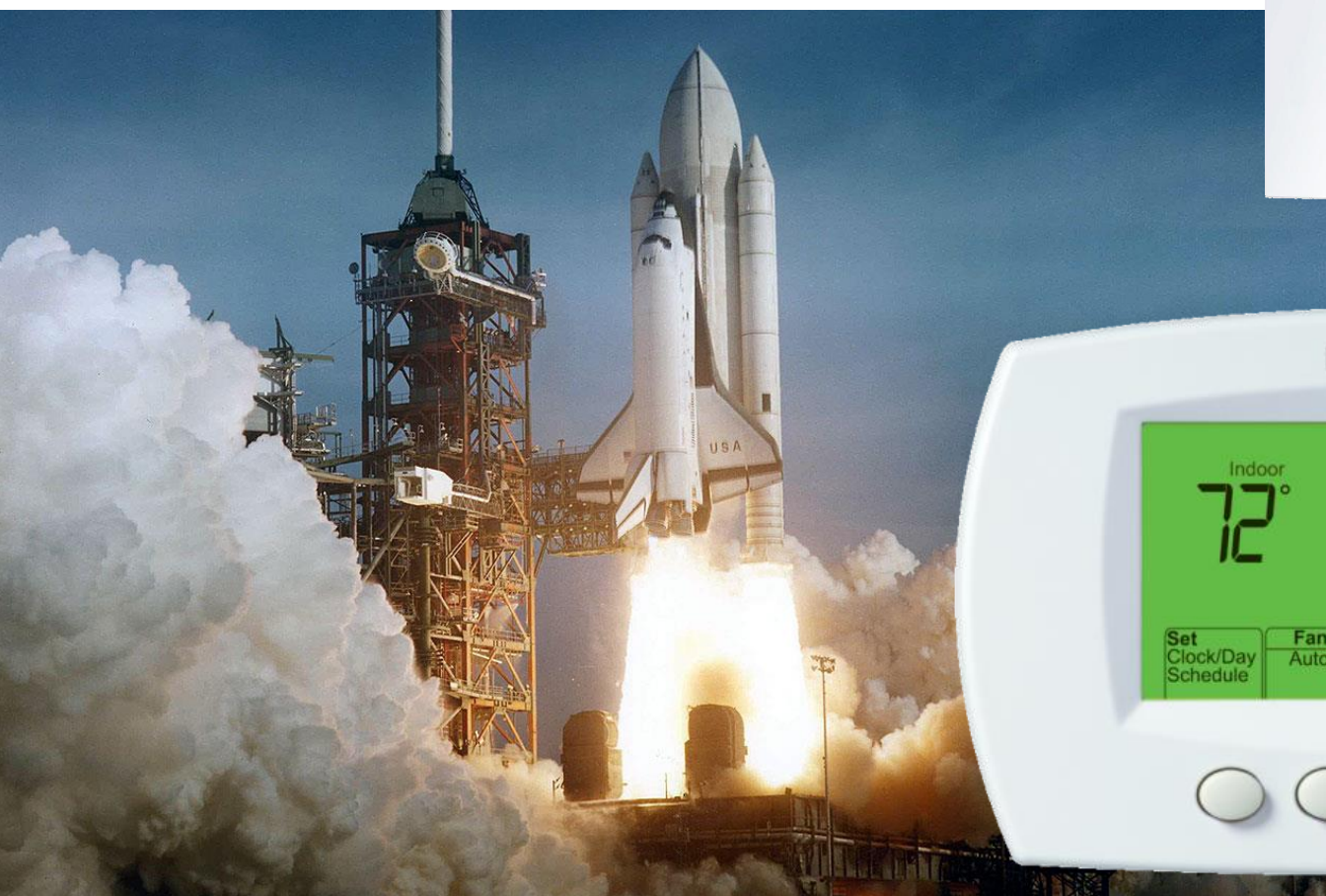
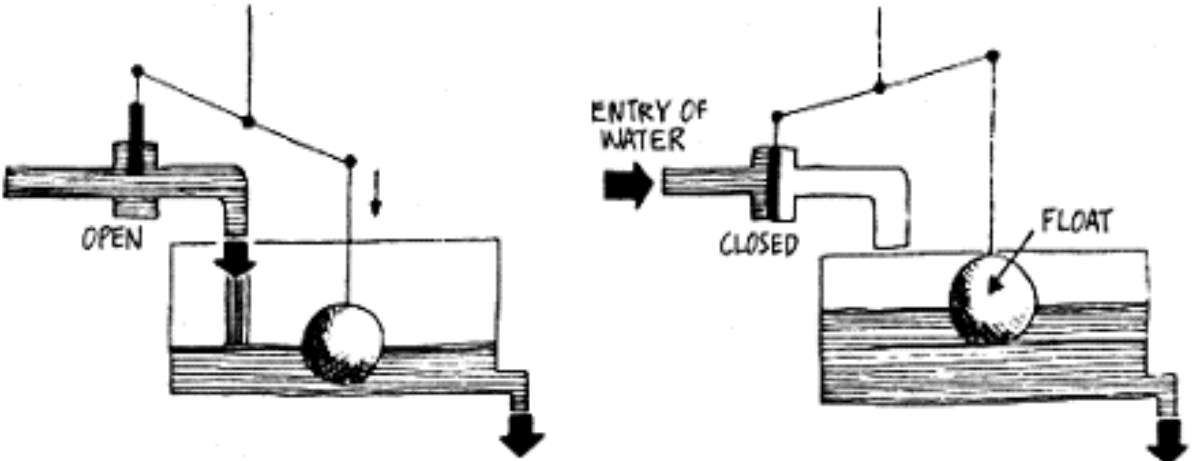
- Move overproduction control to source
- Keep the operators *reactive*
- *Automatically calculate how much data to produce*

# Feedback Control

*Laplace transformation*

$$Y(s) = \left( \frac{P(s)C(s)}{1 + F(s)P(s)C(s)} \right) R(s)$$

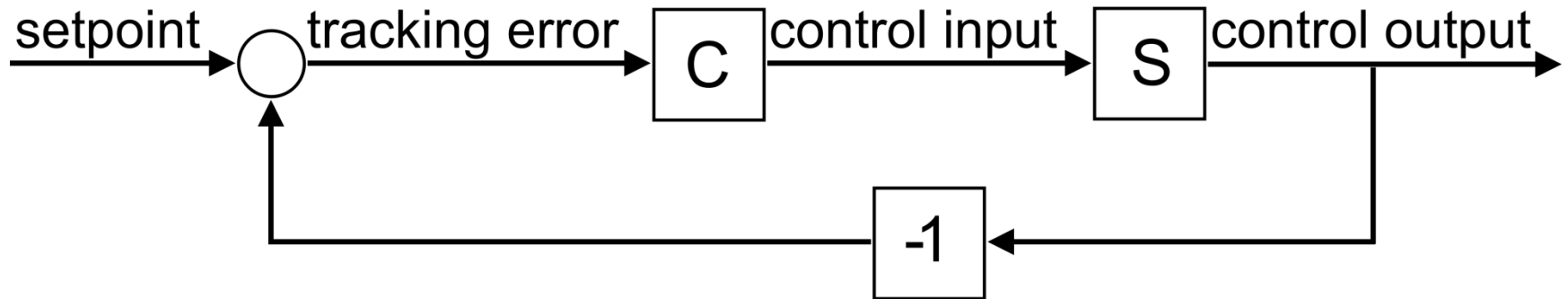






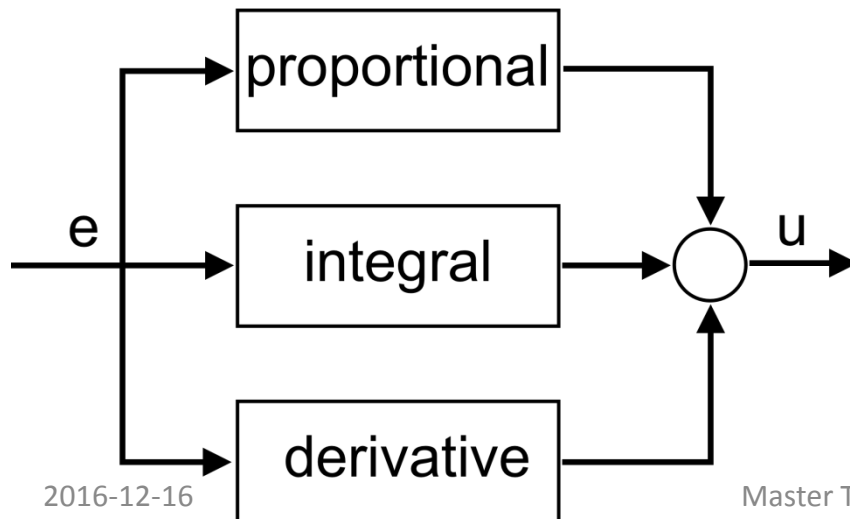


# Feedback system



*PID controller*

$$u(t) = k_p \cdot e(t) + k_i \int_0^t e(\tau) d\tau + k_d \cdot \frac{de(t)}{dt}$$



# feedback4s

Components are transformations over *Observables*:

```
class Component[I, O](transform: Observable[I] => Observable[O]) {  
  def run(is: Observable[I]): Observable[O] = transform(is)  
  
  def >>>[X](other: Component[O, X]): Component[I, X] =  
    Component(other.run _ compose this.run)  
  
  def map[X](f: O => X): Component[I, X] = this >>> create(f)  
  
  // many more operators  
}
```

```
object Component {  
  def create[I, O](f: I => O): Component[I, O] =  
    new Component(_ map f)  
}
```

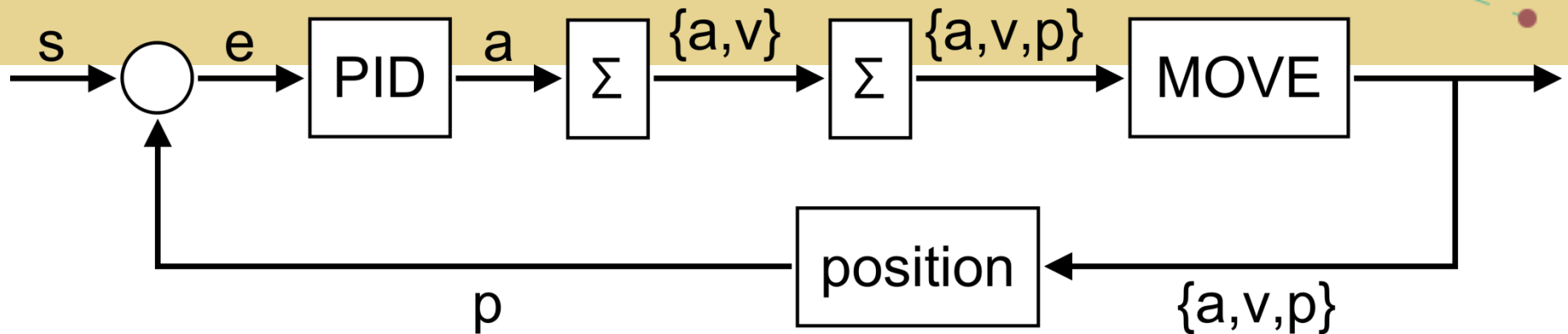
# feedback4s

```
<dependency>  
  <groupId>com.github.rvanheest</groupId>  
  <artifactId>feedback4s</artifactId>  
  <version>1.0</version>  
</dependency>
```

<https://github.com/rvanheest/feedback4s>

<http://doi.org/10.5281/zenodo.169095>

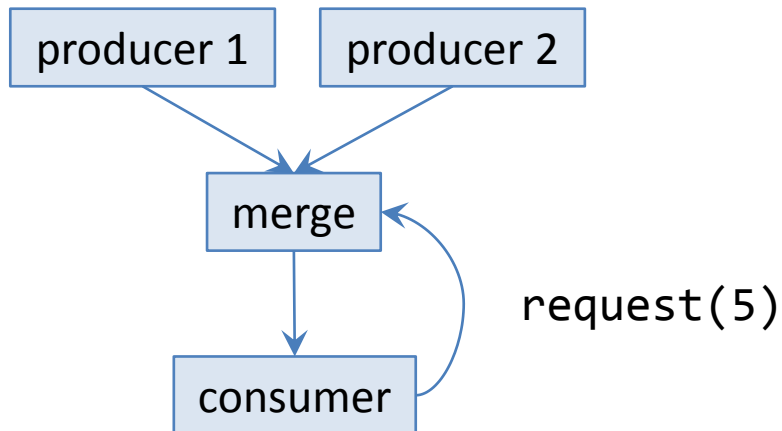
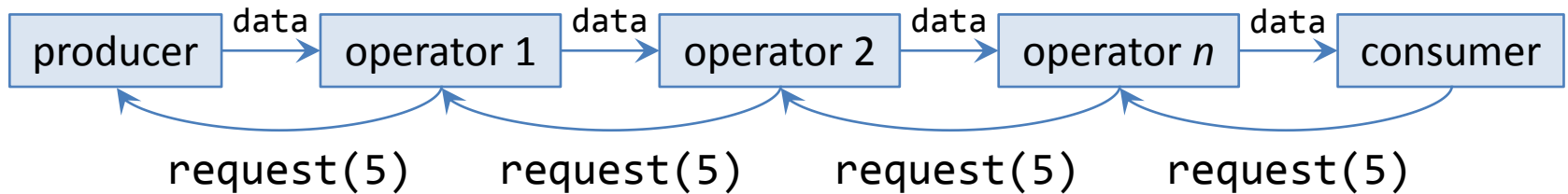
# Demo



# Feedback applied to overproduction

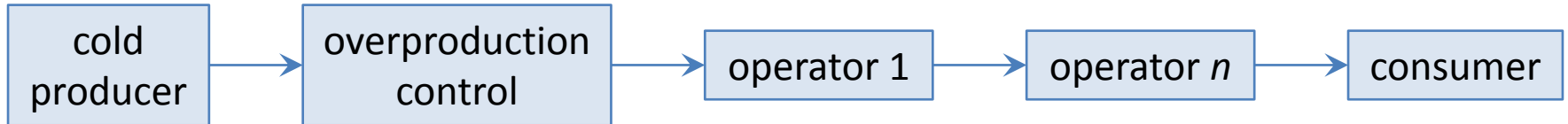
- Overproduction: fast producer, slow consumer
- Hot source: can't be interacted with
  - Lossy & lossless operators
- Cold source: can be interacted with
  - Backpressure

# Backpressure/Reactive Streams



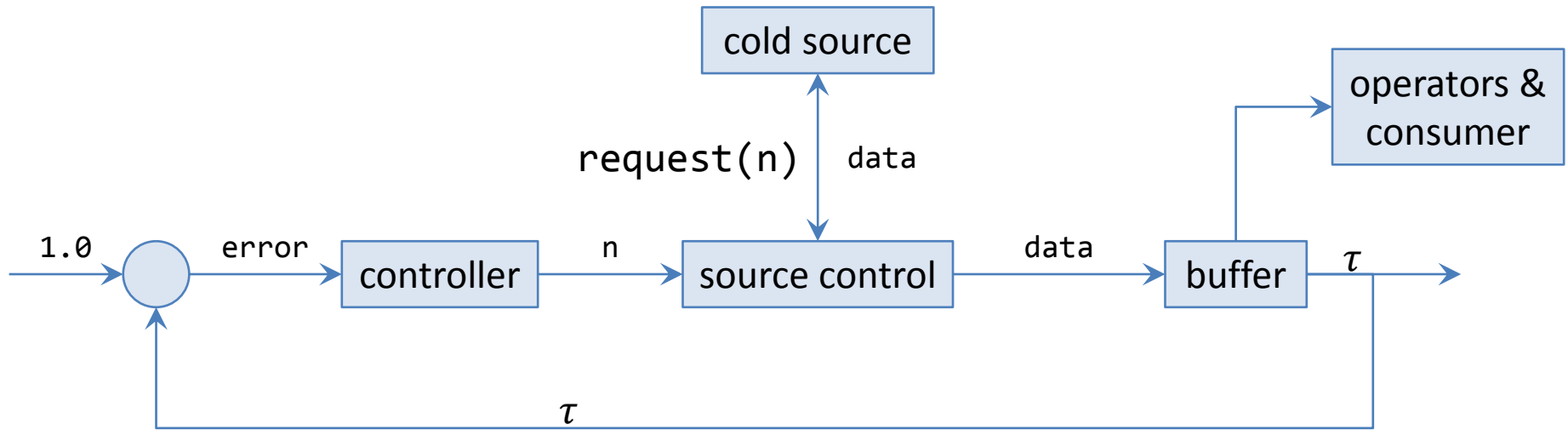
```
producer1.merge(producer2)  
    .subscribe(println(_))
```

# *'Backpressure'* in the source



- Overproduction control where it is handled best:
  - **In the source**
- Everything after that is considered to be hot!

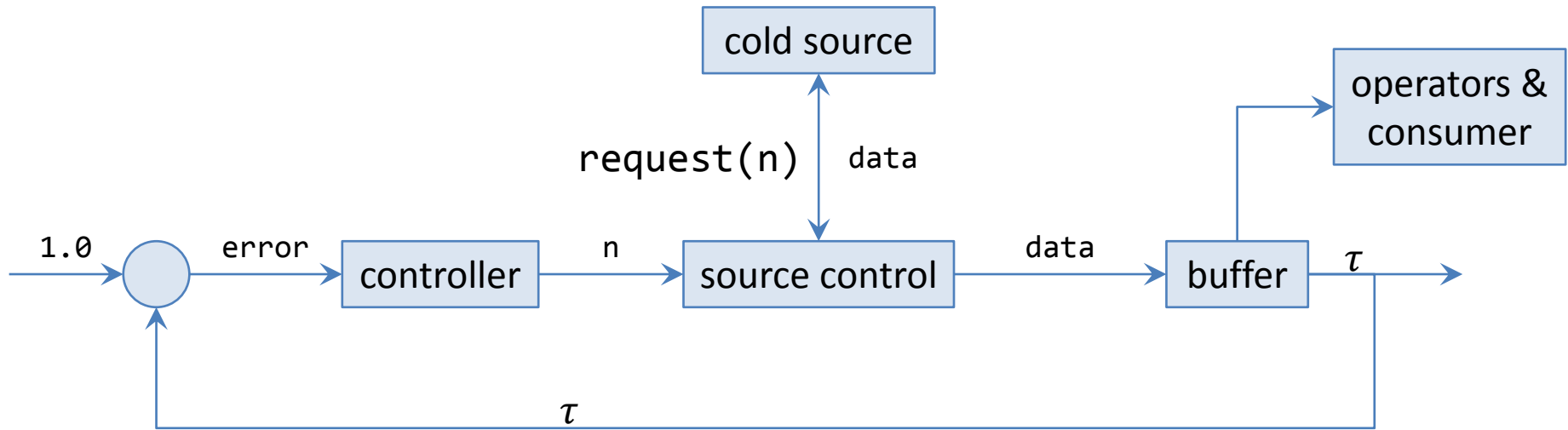
# Overproduction feedback system



- Metrics
  - #elements in buffer
  - throughput



# Overproduction feedback system



- Throughput

- $\tau_t = \frac{q_{t-1} + n_t - q_t}{q_{t-1} + n_t} = 1 - \frac{q_t}{q_{t-1} + n_t}$  with  $q_{t-1}, q_t, n_t$  integers  $\geq 0$

- $0.0 \leq \tau_t \leq 1.0$

# Overproduction in Rx

```
trait Requestable[T] {  
  protected final val subject = PublishSubject[T]()  
  
  final def results: Observable[T] = subject  
  
  def request(n: Int): Unit  
}  
  
object Requestable {  
  implicit class ReqObs[T](val requestable: Requestable[T]) {  
    def observe(timeout: Duration = 1 milli): Observable[T] = ...  
  }  
}
```

# Overproduction in Rx

```
Requestable.from(0 until 133701)
  .observe
  .filter(_ % 2 == 0)
  .map(2 *)
  .drop(10)
  .take(20)
  .subscribe(i => println(s"> $i"))
```

# Overproduction in Rx

- > 40
- > 44
- > 48
- > 52
- > 56
- > 60
- > 64
- > 68
- > 72
- ...

# Compared to backpressure

- Wrap cold source in reactive programming model
- Overproduction safety
- Source + feedback system = hot
- Operators can be implemented purely reactively

# Main achievements

- Analysis of sources and existing solutions for overproduction
- feedback4s
- Reactive solution to overproduction for cold sources

# Solving the Overproduction Problem in Reactive Programming using Feedback Control

Master Thesis Defense

Richard van Heest