Functional Reactive Programming with F#

"Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that non-determinism."

— *Edward A. Lee*

(The Problem with Threads, Berkeley 2006)

Riccardo Terrell

Agenda

- What & Why Functional Reactive Programming
- Reactive Manifesto in details
- Reactive Extensions
- Async Workflow
- Actor Model
- Code & Code

Something about me -Riccardo Terrell

- Originally from Italy, currently
  - Living/working in Washington DC ~10 years
- +/- 18 years in professional programming
  - C++/VB → Java → .Net C# → Scala → Haskell → C# & F# → ??
- Organizer of the DC F# User Group
- Software Architect @ Microsoft
- Currently drafting
  - "Concurrency and Distributed Systems in Functional Programming"
- Believer in polyglot programming as a mechanism in finding the right tool for the job

rickyterrell.com @trikace rterrell@microsoft.com

Goals - Plan of the talk

F# really **shines** in the area of distributed computing

Applications must be built with concurrency in mind
(possibly from the beginning)

Functional Programming helps to go Reactive

Implications are massive, change is unavoidable

Users

Users are demanding richer and more personalized experiences.

Yet, at the same time, expecting blazing fast load time.

Applications

Mobile and HTML5; Data and compute clouds; scaling on demand.

Modern application technologies are fueling the always-on, real-time user expectation.

Businesses

Businesses are being pushed to react to these changing user expectations…

...and embrace modern application requirements.

Cost of writing applications that are not Reactive

- Cost to your wallet and t
  environment of leveragi
  unnecessary hardware
- No ability to b
  your user
  traf
  loss of custome

As a matter of necessity, businesses are going Reactive!

Twitter Paper

Your Server as a Function

Marius Eriksen
Twitter Inc.
marius@twitter.com

Abstract

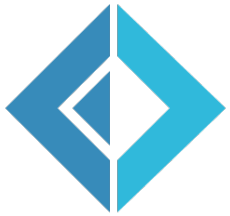
Building server software in a large-scale setting, where systems exhibit a high degree of concurrency and environmental variability, is a challenging task to even the most experienced programmer. Efficiency, safety, and robustness are paramount—goals which have traditionally conflicted with modularity, reusability, and flexibility.

We describe three abstractions which combine to present a pow-

Services Systems boundaries are represented by asynchronous functions called *services*. They provide a symmetric and uniform API: the same abstraction represents both clients and servers.

Filters Application-agnostic concerns (e.g. timeouts, retries, authentication) are encapsulated by *filters* which compose to build services from multiple independent modules.

Twitter Paper



Your Server as a Function

Servers in a large-scale setting are required to process tens of thousands, if not hundreds of thousands of requests concurrently; they need to handle partial failures, adapt to changes in network conditions, and be tolerant of operator errors. As if that weren't enough, harnessing off-the-shelf software requires interfacing with a heterogeneous set of components, each designed for a different purpose. These goals are often at odds with creating modular and reusable software [6].

We present three abstractions around which we structure our server software at Twitter. They adhere to the style of functional programming—emphasizing immutability, the composition of first-class functions, and the isolation of side effects—and combine to present a large gain in flexibility, simplicity, ease of reasoning, and robustness.

The World is changed

"In today's world, the demand f… …ner expectations such as… …companies have come f… …on.

"Modern applica… …rporating this behavior into their …NA".

- R… …ive App… …n Development (Manning)

Functional Reactive Programming

Functional Reactive Programming

Functional Programming

It's declarative, stateless, side-effects free and immutable

Functional Programming

It's declarative, stateless, side-effects free and immutable

describes what we want, not how we want it

> *"Functional programming allows developers to describe what they want to do, rather than forcing them to describe how they want to do it."*
>
> Anders Hejlsberg, C# creator

Functional Programming

It's declarative, stateless, side-effects free and immutable

describes what we want, not how we want it

system relys only on inputs, not external state

Functional Programming

It's declarative, stateless, side-effects free and immutable

describes what we want, not how we want it

system relys only on inputs, not external state

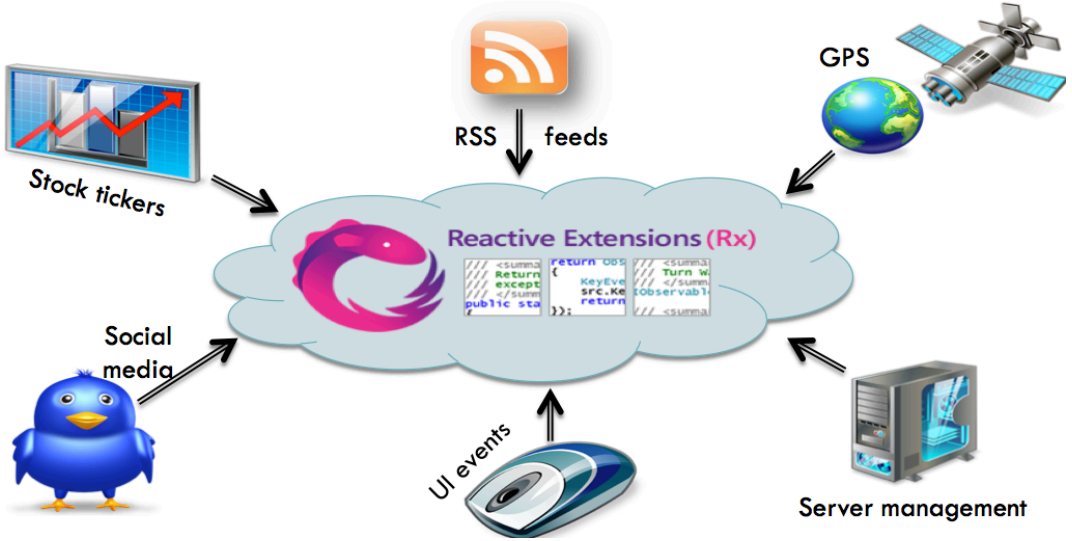
once value has been set, we can't override its value

Return new value, instead of altering existing ones

Functional Reactive Programming

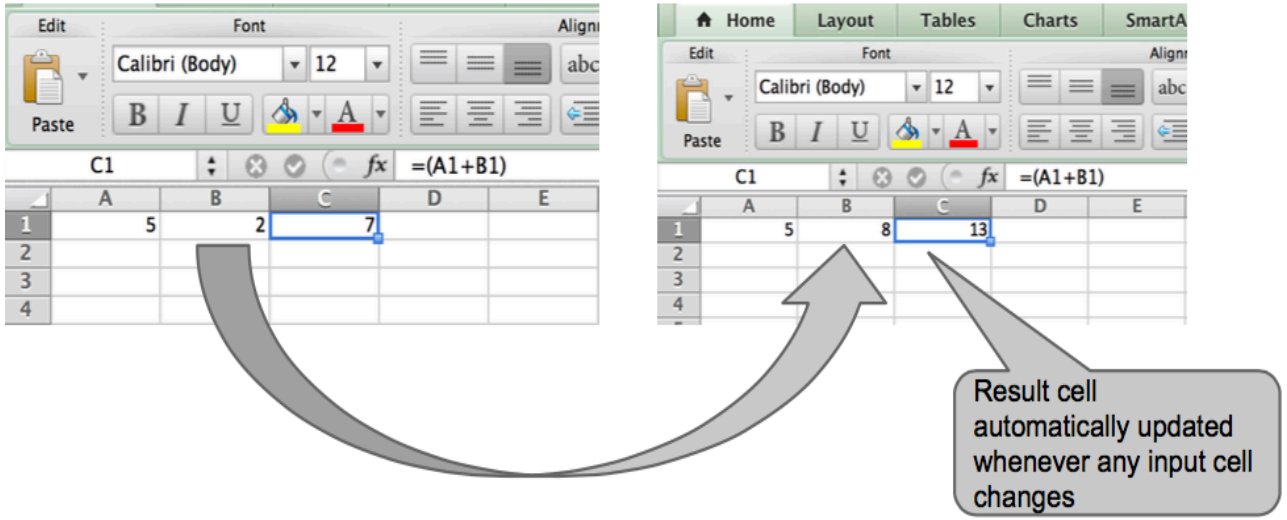
What is Reactive Programming

A programming paradigm oriented around data flows
and the propagation of change.



What is Reactive Programming

SpreadSheet == Mother of All Reactive Programming



Functional Reactive Programming

Push-Pull Functional Reactive Programming

Push-Pull Functional Reactive Programming

Conal Elliott
LambdaPix
conal@conal.net

Abstract

Functional reactive programming (FRP) has simple and powerful semantics, but has resisted efficient implementation. In particular, most past implementations have used demand-driven sampling, which accommodates FRP's continuous time semantics and fits well with the nature of functional programming. Consequently, values are wastefully recomputed even when inputs don't change. more composable than their finite counterparts, because they can be scaled arbitrarily in time or space, before being clipped to a finite time/space window.

While FRP has simple, pure, and composable semantics, its efficient implementation has not been so simple. In particular, past implementations have used demand-driven (pull) sampling of reactive behaviors, in contrast to the data-driven (push) evaluation typ-

What is Functional Reactive Programming

- ‣ Created in 1997 by Conal Elliott for the reactive animations framework Fran, in Haskell ‣ Since then other implementations have appeared: reactive-banana, NetWire, Sodium

- ‣ And then FRP-inspired ones: Rx.NET | Java | JS], Baconjs, reagi (Clojurescript) ‣ Main abstractions: *Behaviors* e *Events*

What is Functional Reactive Programming

"FRP is about handling time-varying values like they were regular values"

- Haskell Wiki

```
type Behavior a = Time -> a
```

```
type Event a = [(Time, a option)]
```

FRP vs FRP

Continuous Time

Virtual Time

Functional Reactive Programming

Definitions Async vs. Concurrent vs Parallel

Asynchronous

- Non-blocking, specifically in reference to I/O operations (not necessarily parallel, can be sequential.)

Parallel

- Multiple operations processed simultaneously.

Concurrent

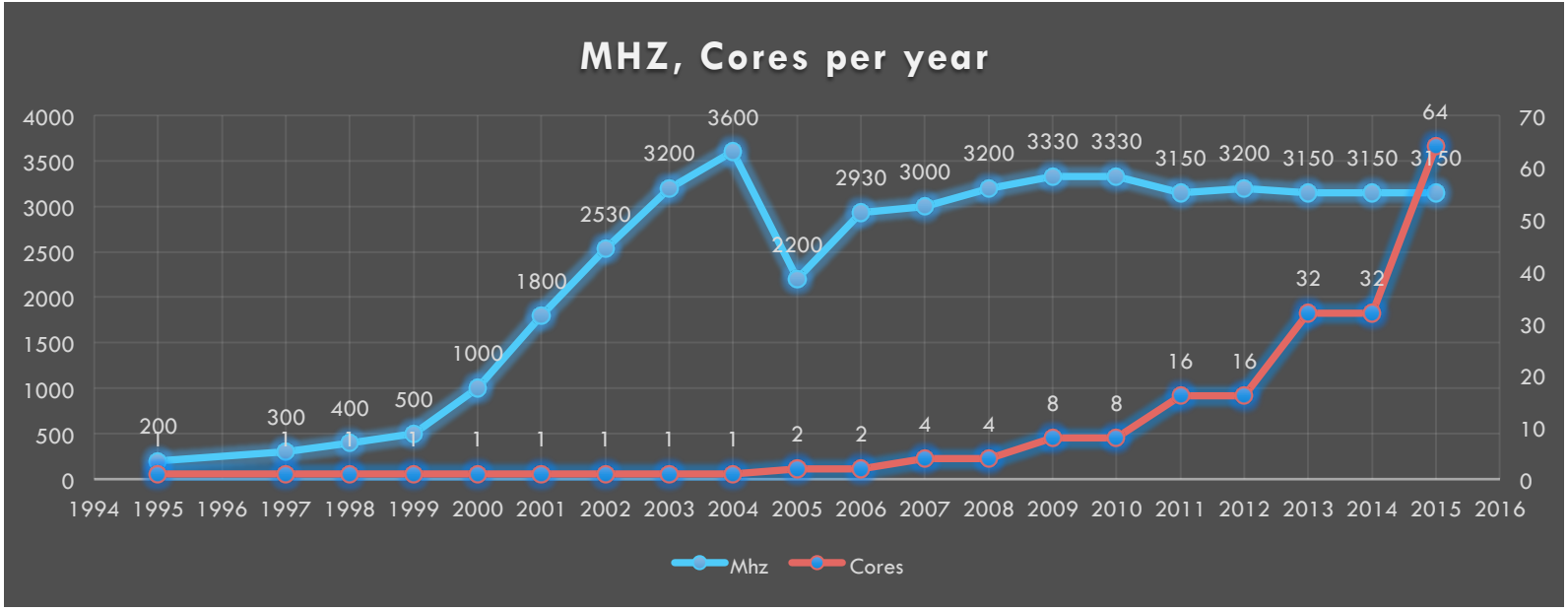
- Multiple operations happening at the same time (not necessarily in parallel).

Reactive

- Programming is writing applications in such a way that they respond to events as they occur in real time.

Moore's law - The Concurrency challenge

Modern computers come with multiple processors, each equipped with multiple cores, but the single processor is slower than it used to be!

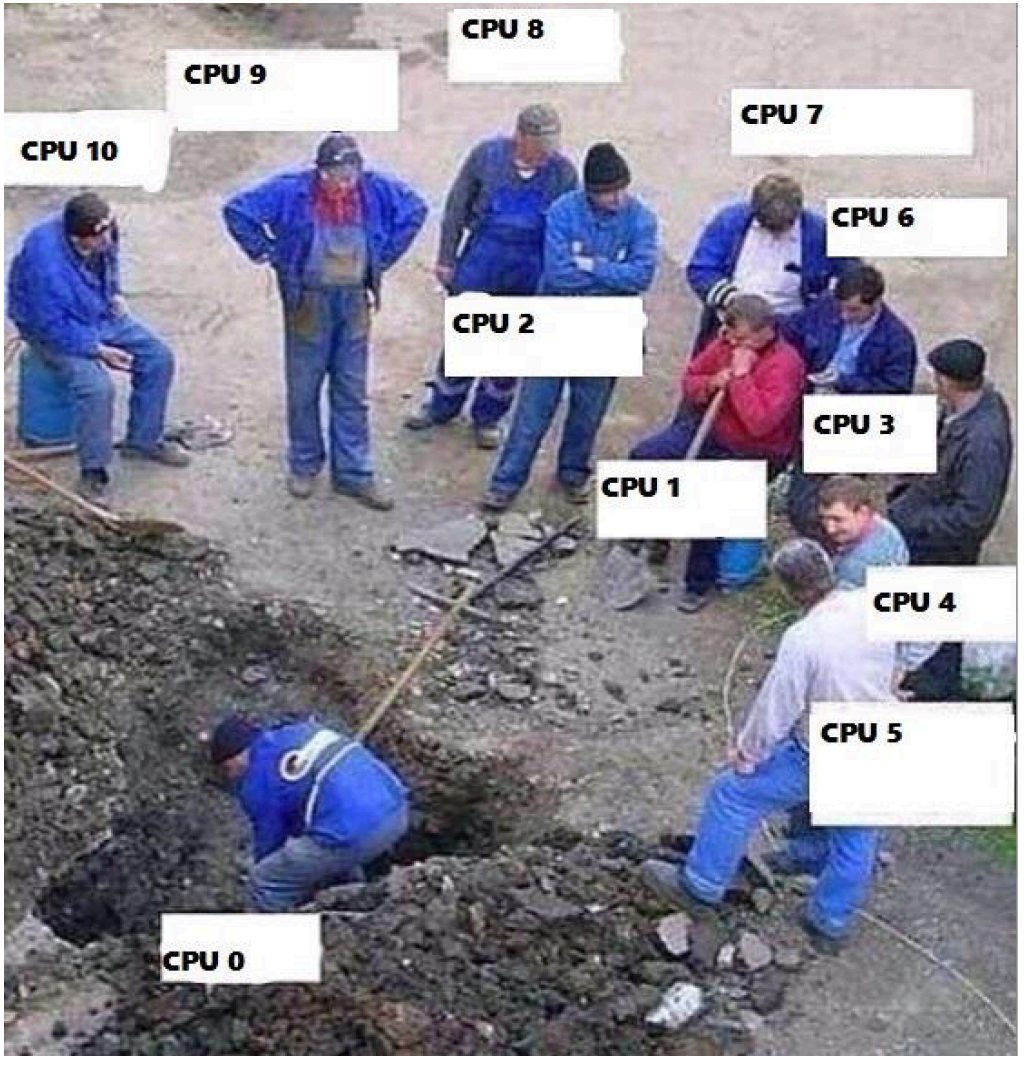


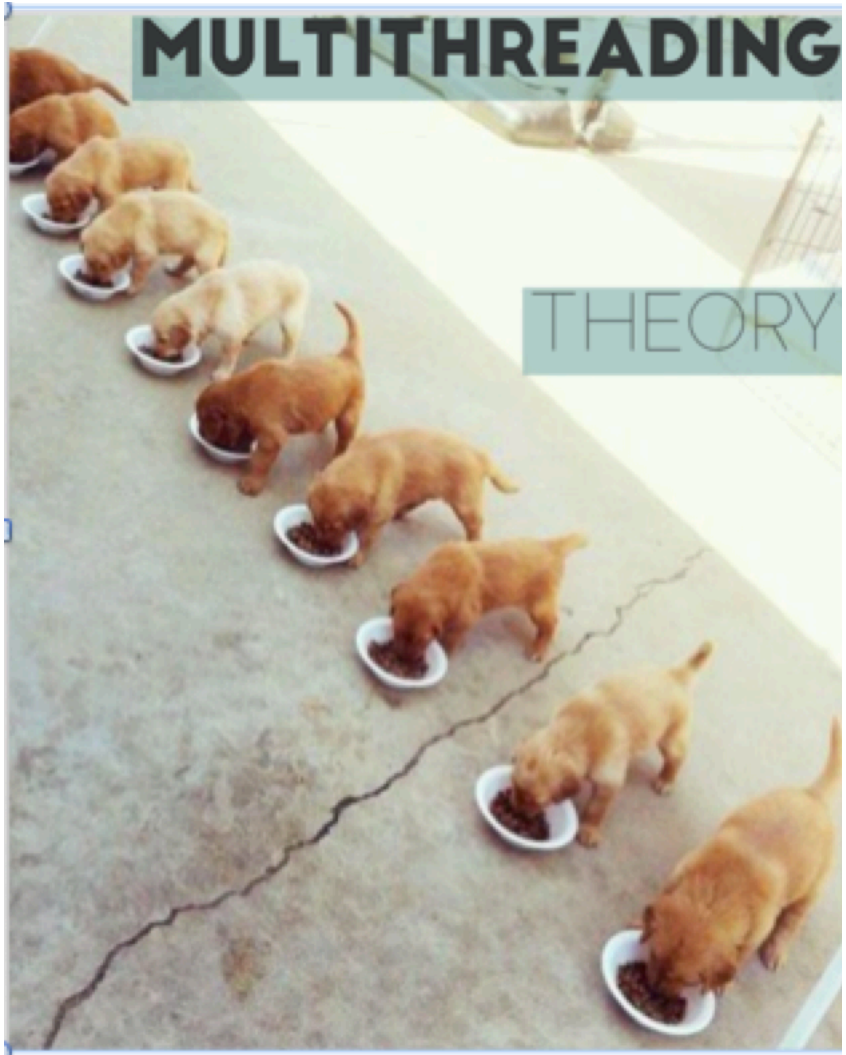
The free lunch is over

There is a problem…

the free lunch is over

- *Programs are not doubling in speed every couple of years for free anymore*
- *We need to start writing code to take advantage of many cores*


CPU 8
CPU 9
CPU 7
CPU 10
CPU 6
CPU 2
CPU 3
CPU 1
CPU 4
CPU 5
CPU 0


MULTITHREADING
THEORY


PRACTICE

The issue is Shared of Memory

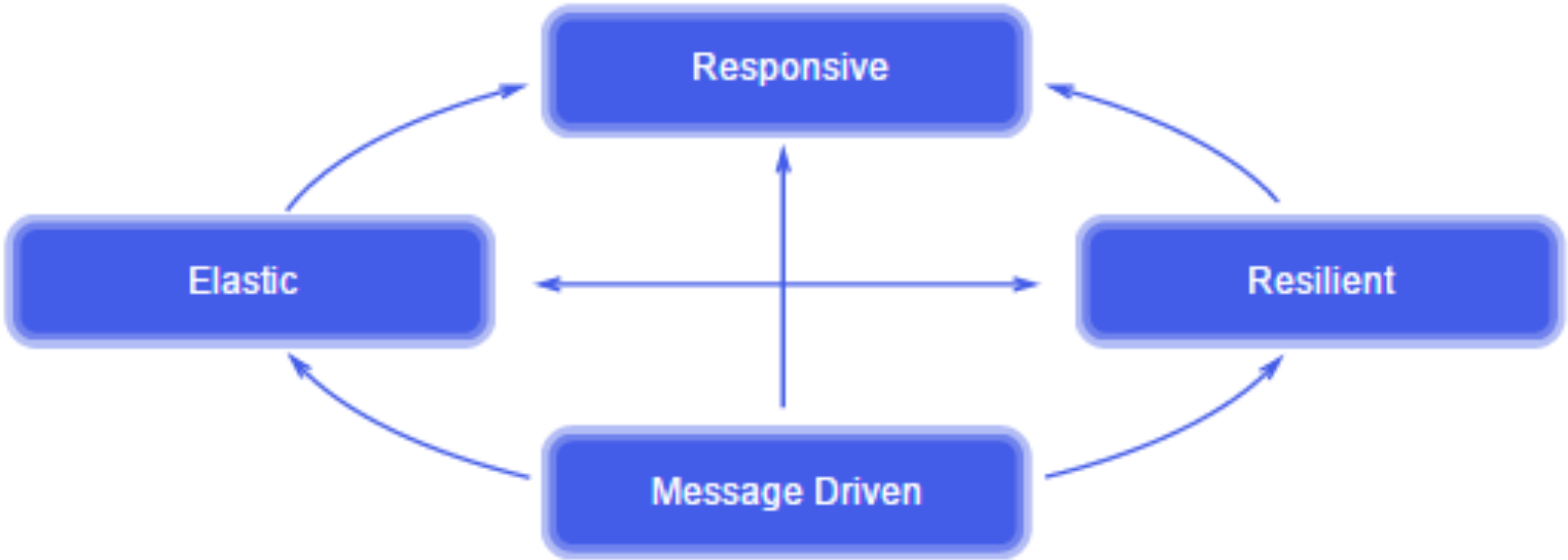
- Shared Memory Concurrency
- Data Race / Race Condition
- Works in sequential single threaded environment
- Not fun in a multi-threaded environment
- Not fun trying to parallelize
- Locking, blocking, call-back hell

Immutability

- Immutability is very important characteristic of FP but mostly is about transformation of state…
  Immutable data forces you to use a "transformational" approach

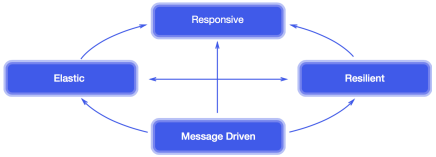
- When you're writing programs using immutable types, the only "thing a method can do is return a result, it can't modify the state of any objects
- Immutable data makes the code predictable is easier to work
- Prevent Bugs
- Concurrency is much simpler, as you don't have to worry about using locks to avoid update conflicts
  - Automatic thread-safe, *controlling mutable state in concurrency is very hard*

Reactive Manifesto



http://www.reactivemanifesto.org

Reactive Manifesto

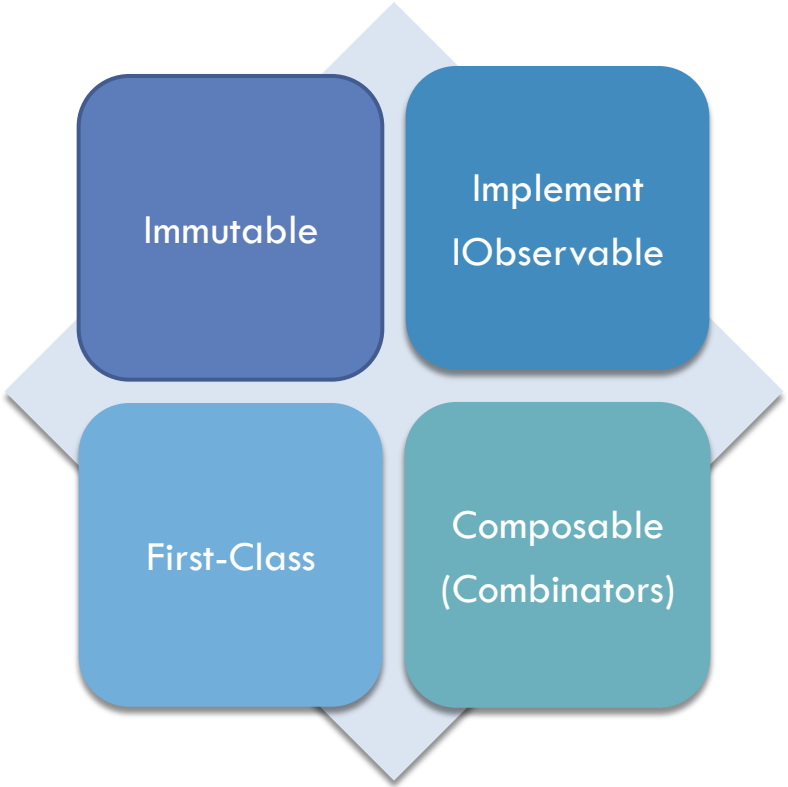


- Responsive
- Message-Driven
- Resilient
- Elastic

The **system responds in a timely manner** if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that **problems may be detected quickly** and dealt with effectively. Responsive systems focus on providing **rapid and consistent response times**, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behavior in turn simplifies error handling, builds end user confidence, and encourages further interaction.

http://www.reactivemanifesto.org

Events in F#


Immutable
Implement IObservable
First-Class
Composable (Combinators)

Event Combinators

The biggest benefit of using higher-order functions for events is that we can express the flow in a Declarative way

- **What** to do with received data using event **combinators**
- Use functions for working with *event values*

Seq Module

```
let myData = seq {1..10}

myData
    |> Seq.map(fun e -> e.ToString())
    |> Seq.filter(fun e -> e <> "")
    |> Seq.choose(fun e -> Some e)
```

Observalbe Module

```
let myEvent = Event<int>()

myEvent
    |> Event.map(fun e -> e.ToString())
    |> Event.filter(fun e -> e <> "")
    |> Event.choose(fun e -> Some e)
```

Event map & filter & add

- Filter and trigger events in specific circumstances with `Event.filter`
- Create a new event that carries a different type of value with `Event.map`

```
Event.map    : ('T -> 'R)   -> IEvent<'T> -> IEvent<'R>
Event.filter : ('T -> bool) -> IEvent<'T> -> IEvent<'T>
```

- Register event with `Event.add`

```
Event.add : ('T -> unit) -> IEvent<'Del,'T> -> unit
```

Event merge & scan

- Merging events with `Event.merge`
  - Triggered whenever first or second event occurs
  - Note that the carried values must have same type

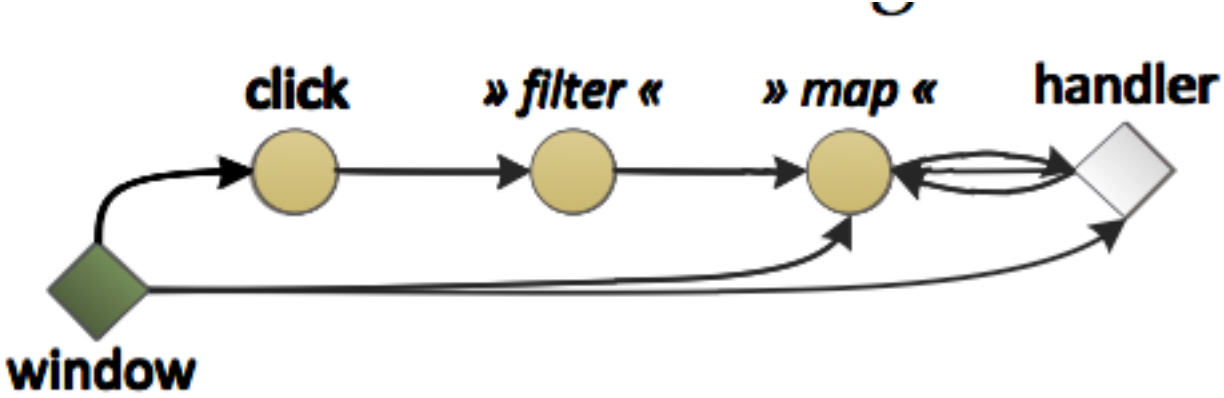
```
IEvent<'T> -> IEvent<'T> -> IEvent<'T>
```

- Creating stateful events with `Event.scan`
  - State is recalculated each time event occurs
  - Triggered with new state after recalculation

```
('St -> 'T -> 'St) -> 'St -> IEvent<'T> -> IEvent<'St>
```

Event are Observable… *almost*

- Memory leak ☹
  - IEvent does not support *removing* event handlers *(RemoveHandler on the resulting event, it leaves some handlers attached… leak!)*
  - Observable is able to remove handlers - **IDisposable**

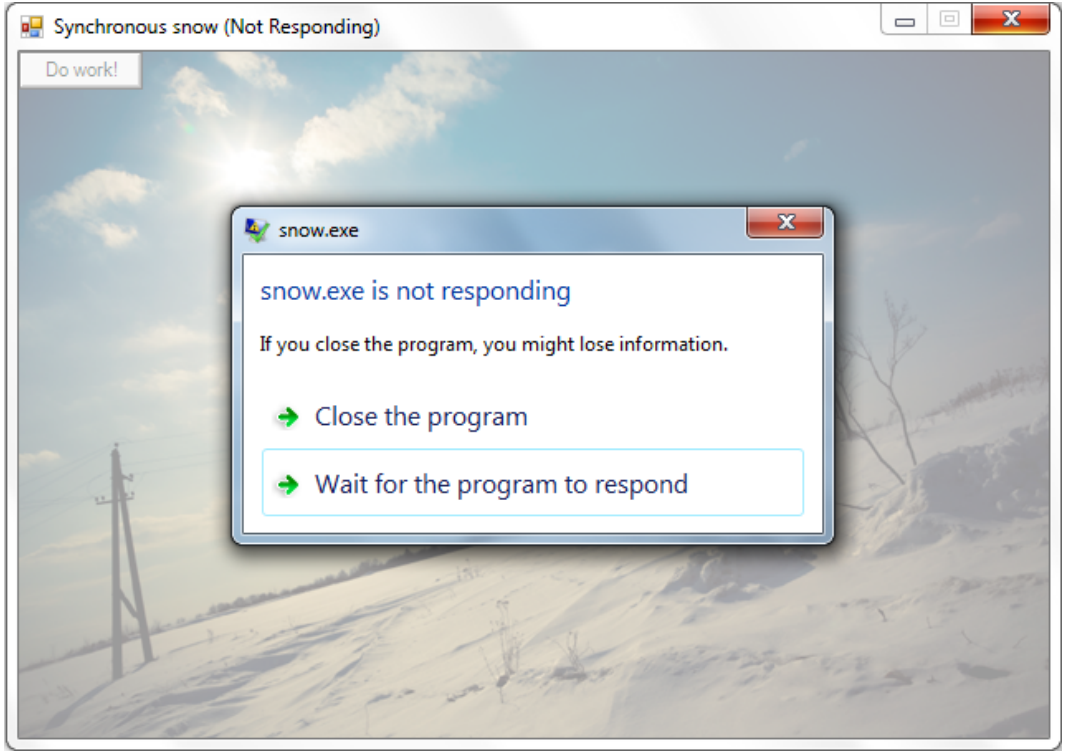


Observable in F#

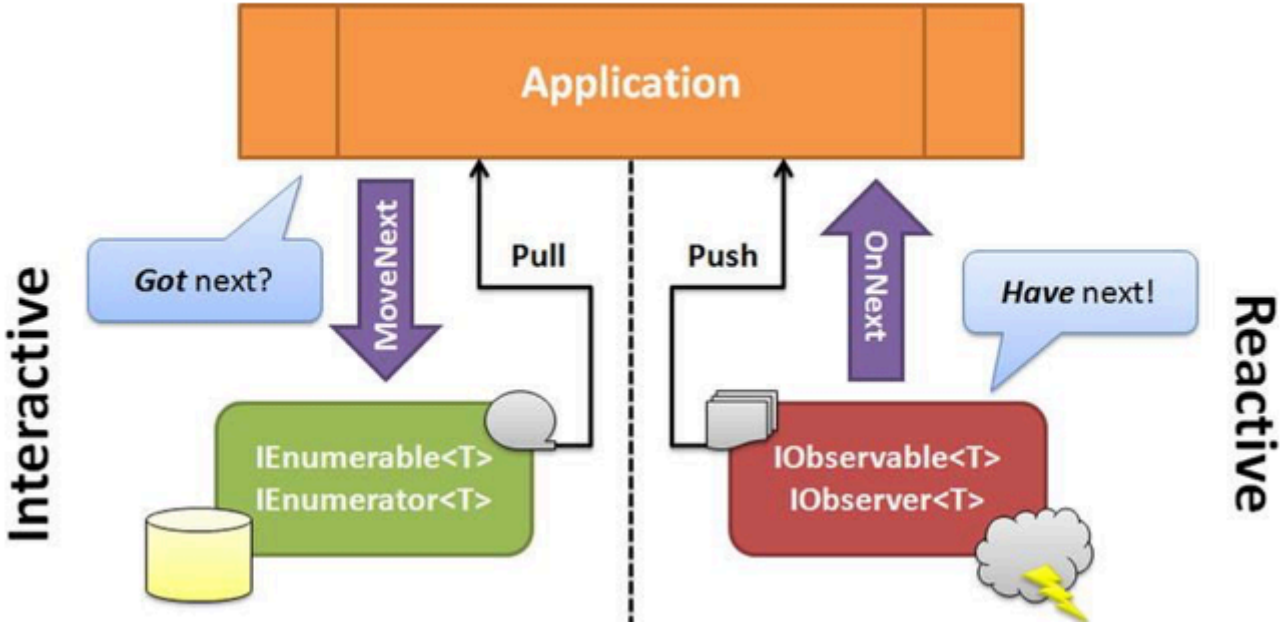
- Event<'T> interface inherits from IObservable<'T>
  - We can use the same standard F# Events functions for working with Observable

```
Observable.filter : ('T -> bool) -> IObservable<'T> -> IObservable<'T>
Observable.map    : ('T -> 'R)   -> IObservable<'T> -> IObservable<'R>
Observable.add    : ('T -> unit)   -> IObservable<'T> -> unit
Observable.merge  : IObservable<'T> -> IObservable<'T> -> IObservable<'T>


      Observable.subscribe  : IObservable<'T> -> IDisposable
```

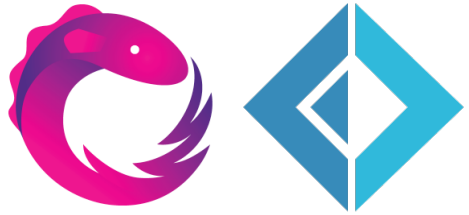
What are Reactive Extensions



Pull vs Push


Application
Got next?
MoveNext
Pull
Push
OnNext
Have next!
Interactive
Reactive
IEnumerable<T>
IEnumerator<T>
IObservable<T>
IObserver<T>

Pull vs Push

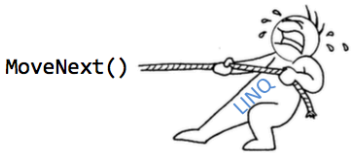


- For pull, data is processed at the leisure of the consumer.
- The consumer "pulls" the next item from the producer

- For push, data is send via demands of the source.
- The producer pushes the data to the consumer.

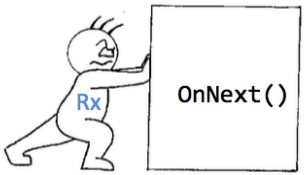
Yin & Yang

Pull collections



- In memory collections
- Database queries
- Generated sequences
- Message queues
- …

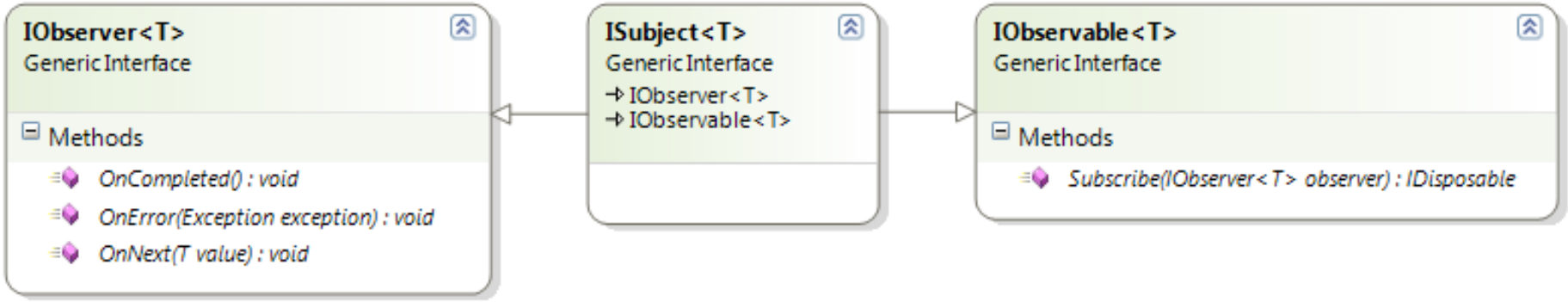
Push collections



- Event streams
- Asynchronous computations
- Asynchronous queries
- Asynchronous enumerations
- …

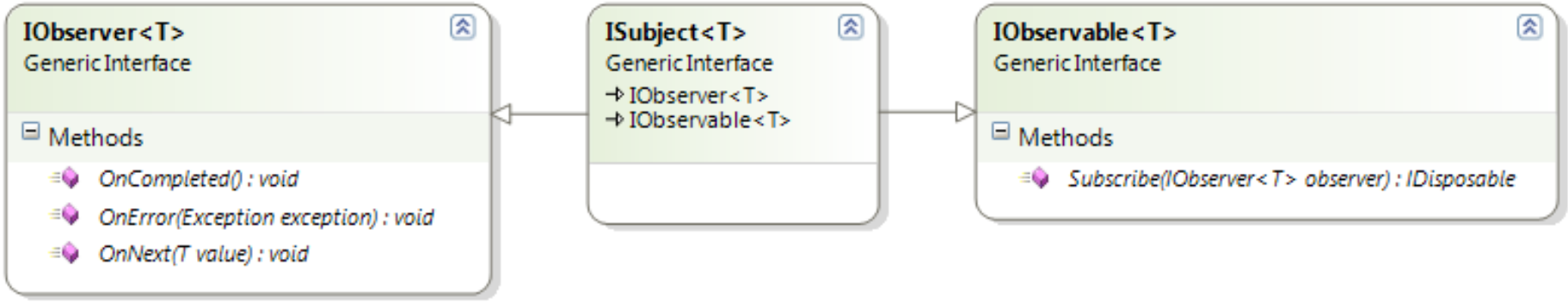
What are Reactive Extensions

Rx is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators



What are Reactive Extensions

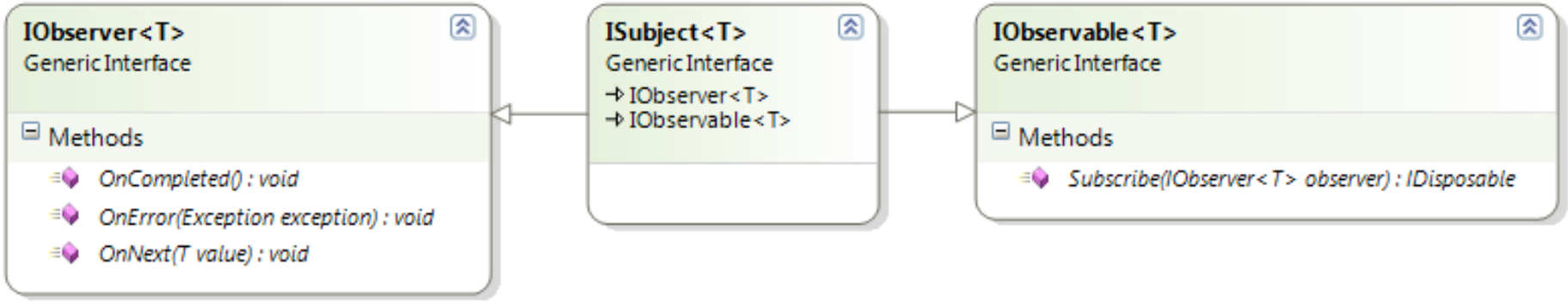
Rx is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators



publisher

What are Reactive Extensions

Rx is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators

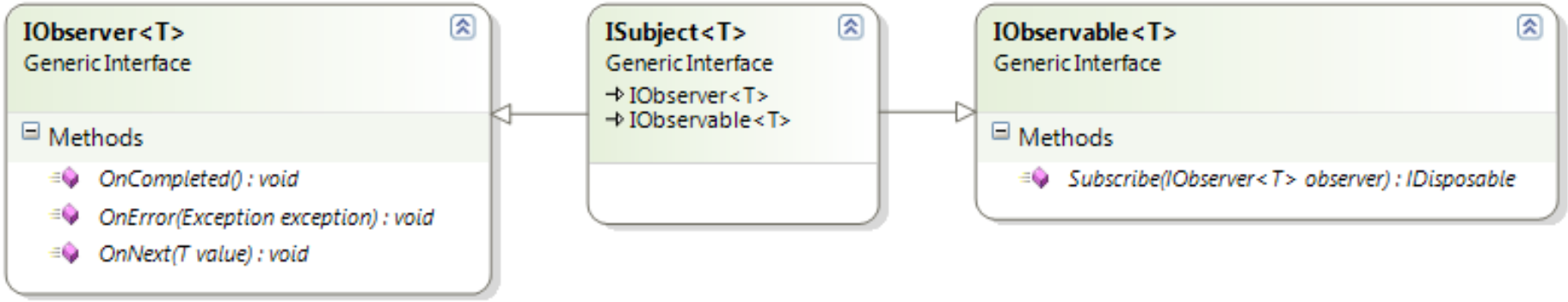


publisher

subscriber

What are Reactive Extensions

Rx is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators

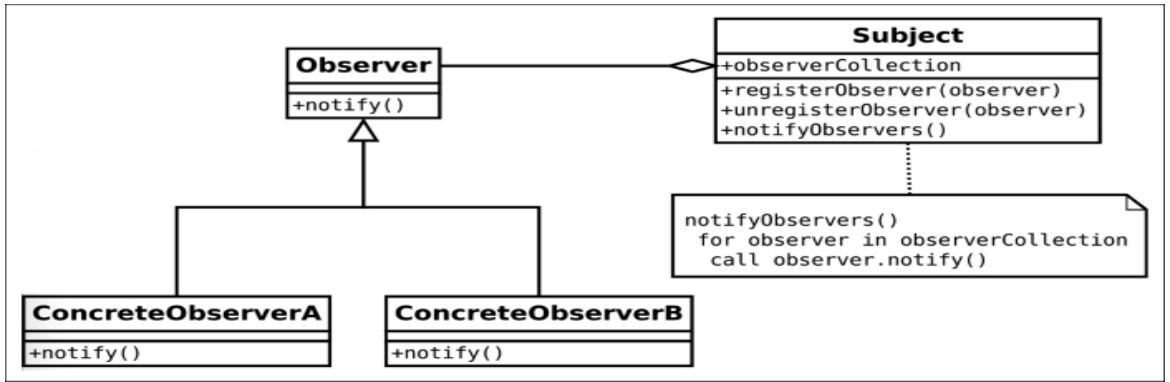


publisher *both* *subscriber*

Observer pattern- Fundamental Abstractions

The Observer pattern is the perfect fit for any of these scenarios:

- When your architecture has two entities, one depending on the other, and you want to keep them separated to change them or reuse them independently.
- When a changing object has to notify an unknown amount of related objects about its own change.
- When a changing object has to notify other objects without making assumptions about who these objects are.



IObserver & IObservable

```
type IObserver<'a> = interface
    abstract OnCompleted : unit -> unit
    abstract OnError : exn -> unit
    abstract OnNext : 'a -> unit
end

type IObservable<'a> = interface
    abstract Subsrcribe : IObserver<'a> -> IDisposable
end
```

IObserver & IObservable

```
let observable = { new IObservable<string> with
                        member x.Subscribe(observer:IObserver<string>) = {
                           new IDisposable with
                              member x.Dispose() = ()
                        }
                  }

let observer = { new IObserver<string> with
                        member x.OnNext(value) = ()
                        member x.OnCompleted() = ()
                        member x.OnError(exn) = ()
                }
```

RX the Dual of IEnumerable

Formal definition [edit]

http://en.wikipedia.org/wiki/Dual_(category_theory)

We define the elementary language of category theory as the two-sorted first order language with objects and morphisms as distinct sorts, together with the relations of an object being the source or target of a morphism and a symbol for composing two morphisms.

Let σ be any statement in this language. We form the dual σ^{op} as follows:

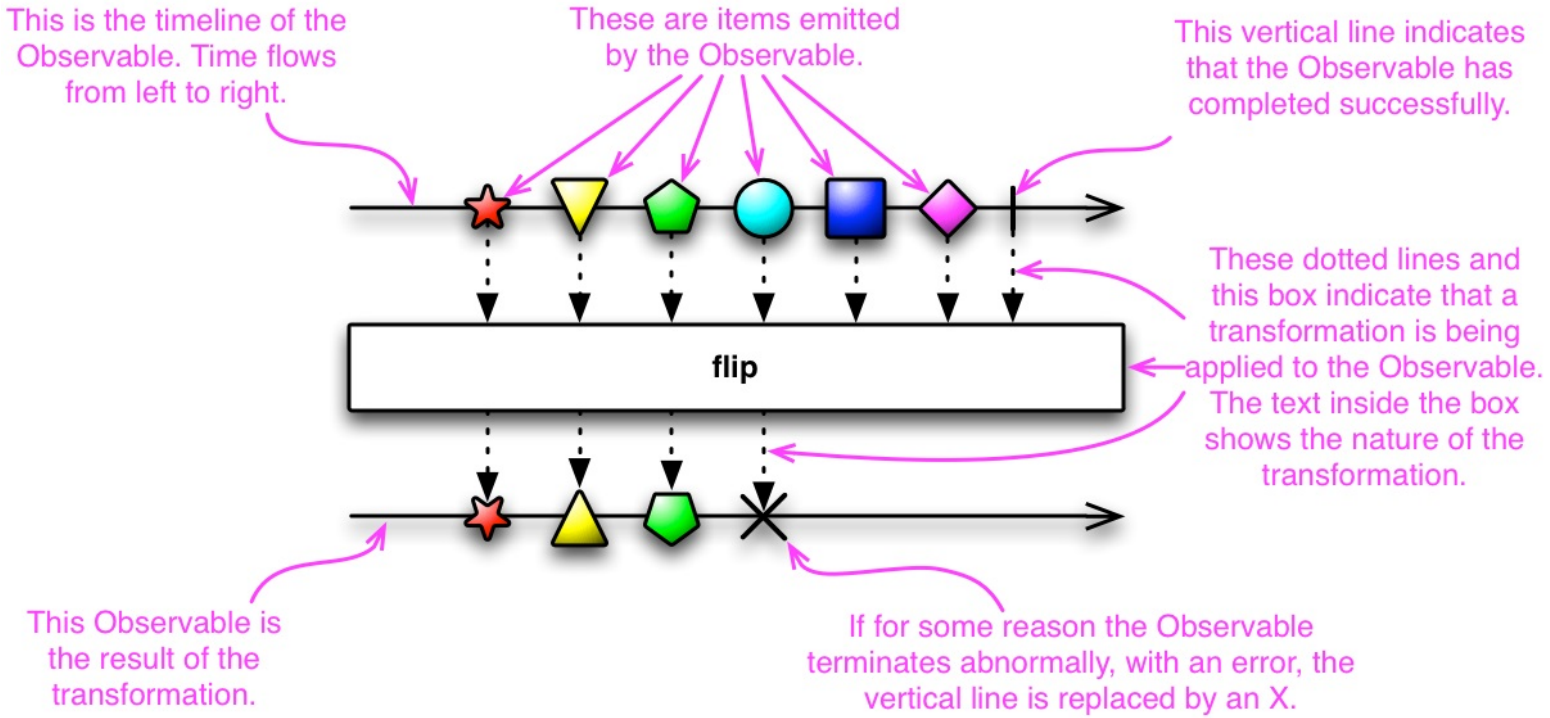
1. Interchange each occurrence of "source" in σ with "target".
2. Interchange the order of composing morphisms. That is, replace each occurrence of $g \circ f$ with $f \circ g$

Informally, these conditions state that the dual of a statement is formed by reversing arrows and compositions.

Duality is the observation that σ is true for some category *C* if and only if σ^{op} is true for C^{op}.

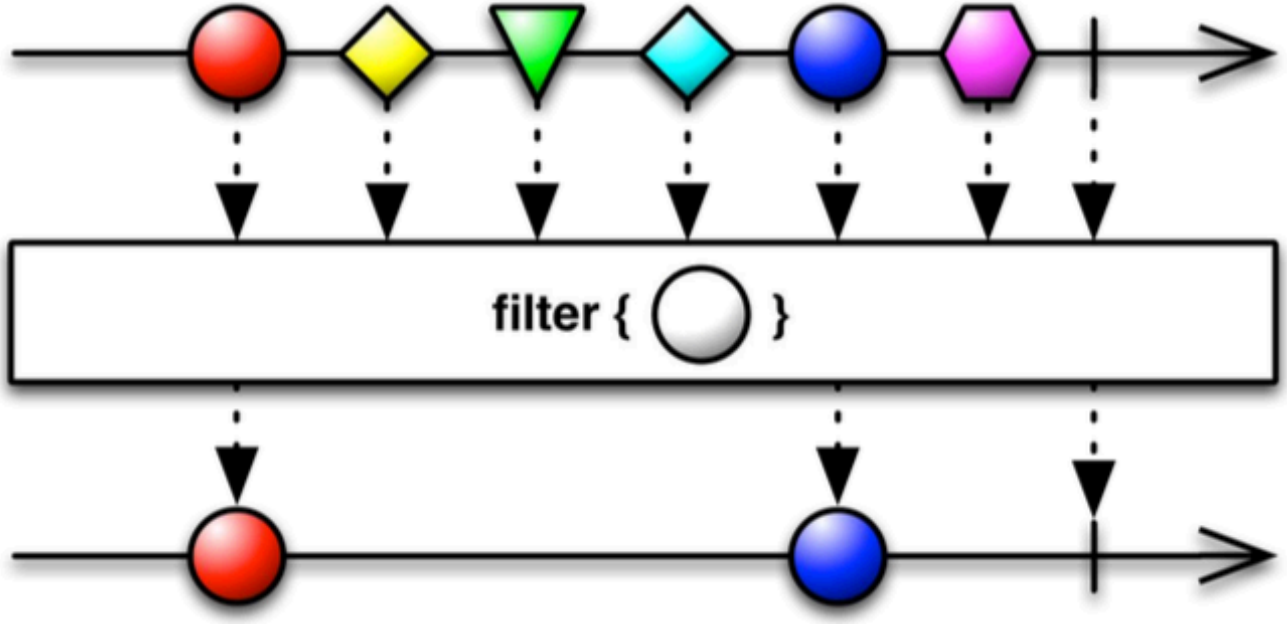
Reversing arrows
The input becomes output and <->

Marble diagram



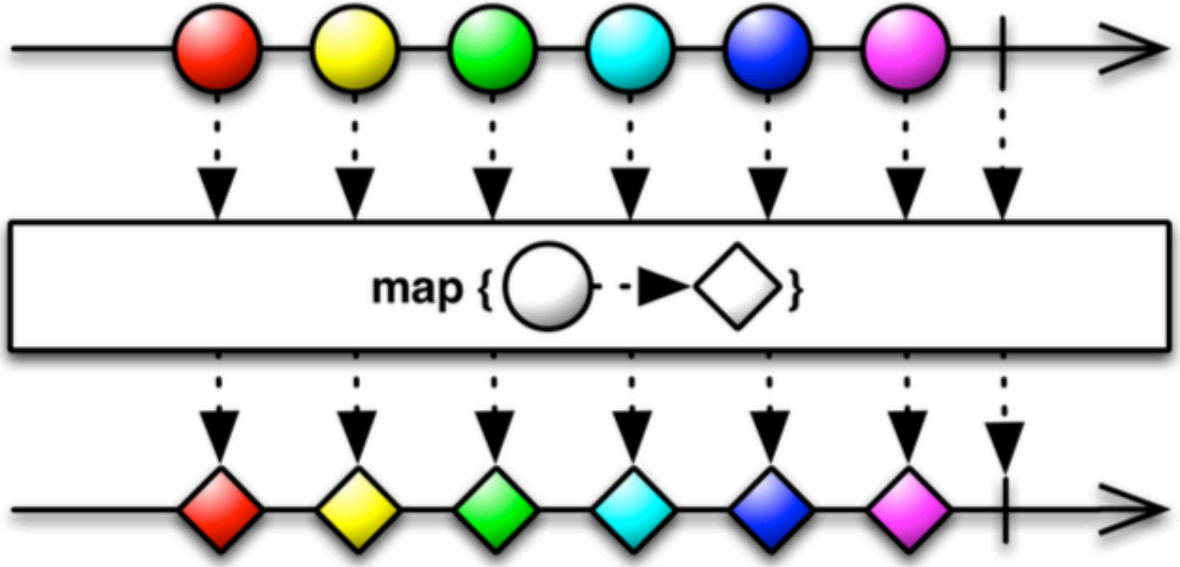
Filter

Filter items emitted by a Collection



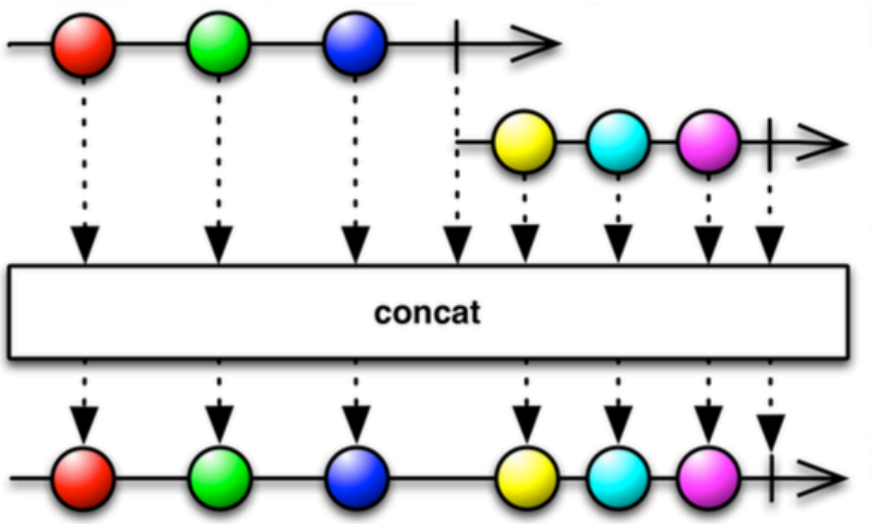
Map

Transform the items emitted by a Collection by applying a function to each of them



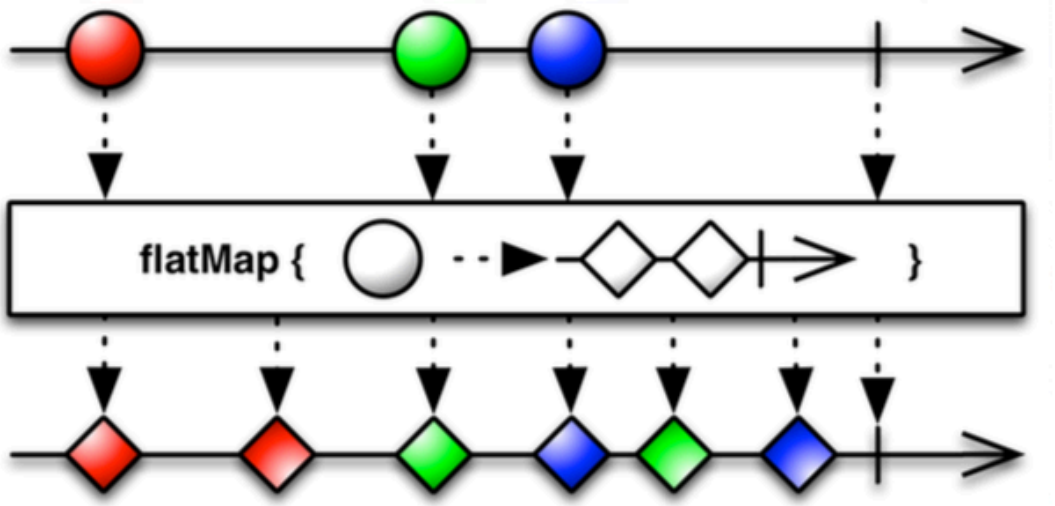
ConactAll

Concatenate two or more Collections sequentially

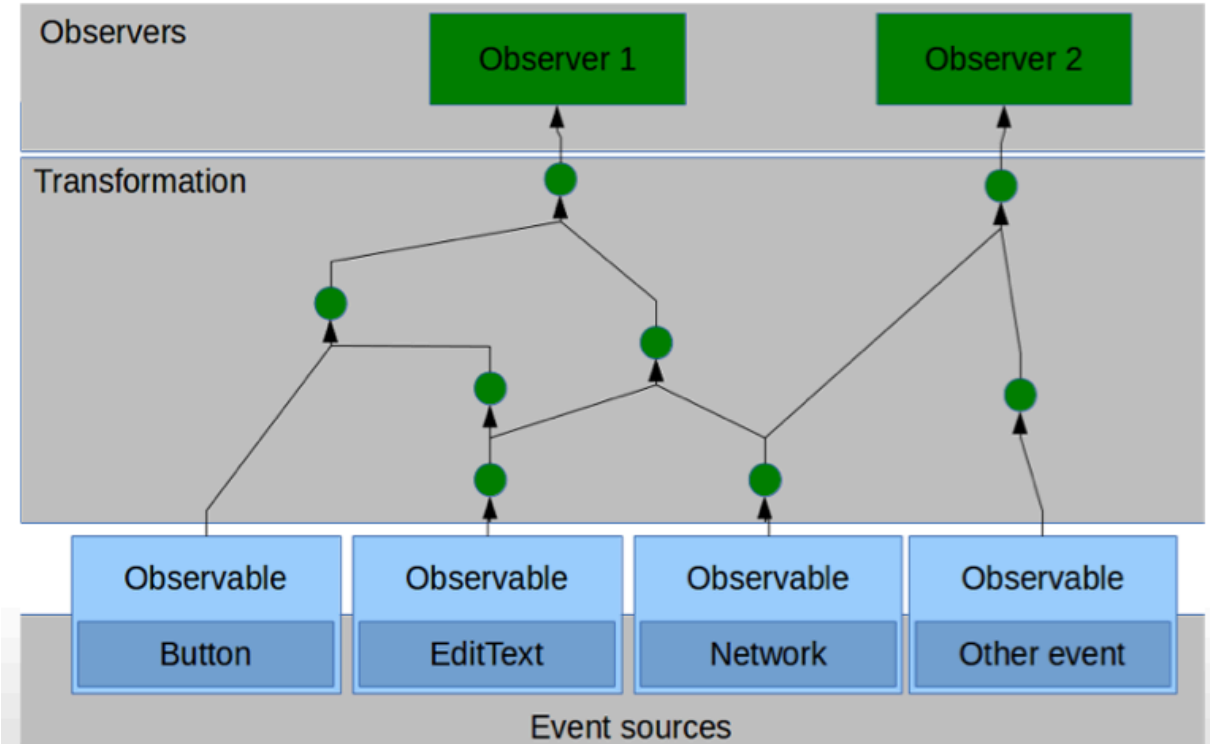


flatMap

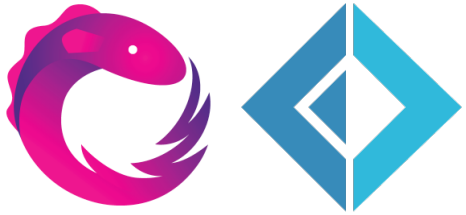
Transform the items emitted by a Collection into Collections, then flatten this into a single Collection



Functional Reactive



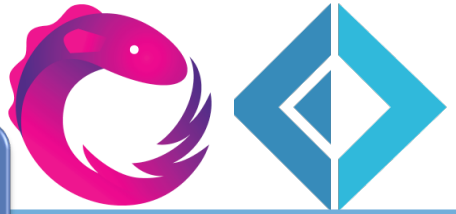
Subject<'a>.. as a bus



```
// Since RX is all about sequences of events/messages
// it does fit very well together with any sort of message bus or event broker.
type RxBusCommand() =
    let subject = new Subject<RxCommand>()

    member this.AsObservable() =
       subject.AsObservable()

    member this.Send(item:RxCommand) =
        subject.OnNext(item)


let busud = RxBusCommand()

busud.AsObservable().Do(fun t ->
                        match t with
                        | BuyTickets(name, quantity) -> printfn "I am buying %d tickets for %s" quantity name
                        | OrderDrink(drink) -> printfn "I am getting some %s to drink" drink).Subscribe() |> ignore

// The nice thing about this is that you get automatic Linq support since it is built into RX.
// So you can add message handlers that filters or transform messages.
busud.Send(BuyTickets("Opera", 2))
busud.Send(OrderDrink("Coke"))
```

```
type RxCommand =
    | BuyTickets of string * int
    | OrderDrink of string
```

Subjects



BehaviorSubject

- emits the most recent item it has observed and all subsequent observed items to each subscribed Observer

AsyncSubject

- buffers all items it observes and replays them to any Observer that subscribes.

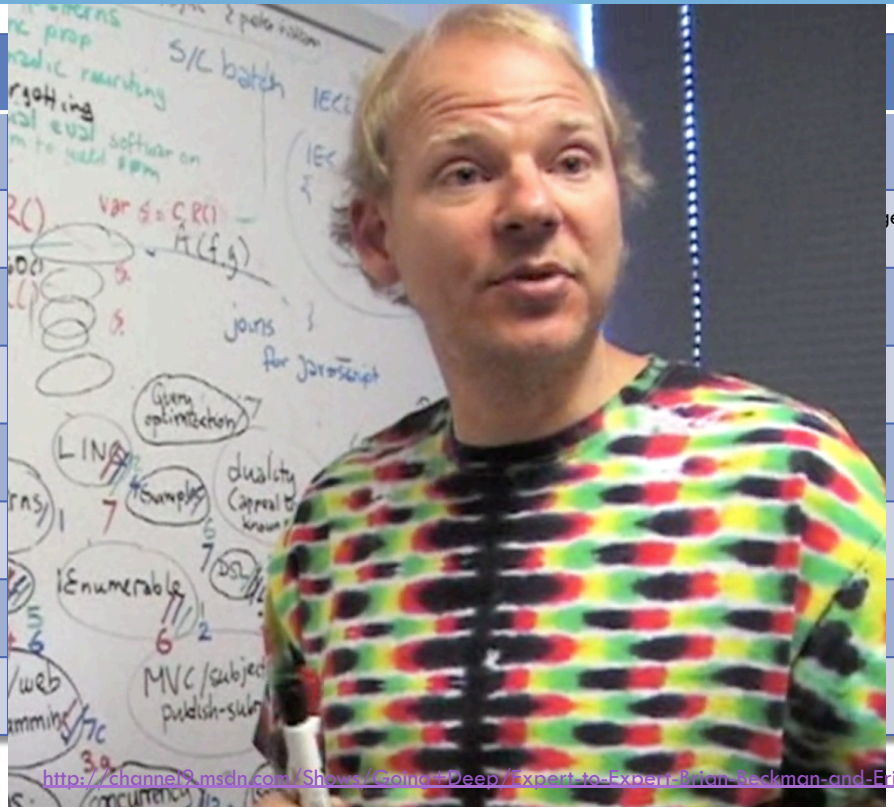
ReplaySubject

- publishes only the last item observed to each Observer that has subscribed, when the source Observable completes

F# & .Net RX Api

F# Observable
add
choose
filter
merge
pairwise
partition
scan
subscribe

.NET RX Api	
All	Amb
Combine	Count
Finally	First
Last	Latest
Multicast	Range
Switch	Select
Take	TakeLast
Window	When



	Catch	Chunhify
ged	ElemntAt	ElementAtOrDefault
	If	IgnoreElement
	MinBy	MostRecent
	Never	Sum
	SkipUntill	Then
	While	When

http://channel9.msdn.com/Shows/Going+Deep/Expert-to-Expert-Brian-Beckman-and-Erik-Meijer-Inside-the-NET-Reactive-Framework-Rx

Asynchronous Workflows

- *Software is often I/O-bound, it provides notable performance benefits*
  - Connecting to the Database
  - Leveraging web services
  - Working with data on disk
- *Network and disk speeds increasingly slower*
- *Not Easy to predict when the operation will complete* (no-deterministic)

Many Kinds of Async Boundries

- *Between different application*
- *Between network nodes*
- *Between CPUS*
- *Between Threads*
- *Between Actors*

Synchronous Programming

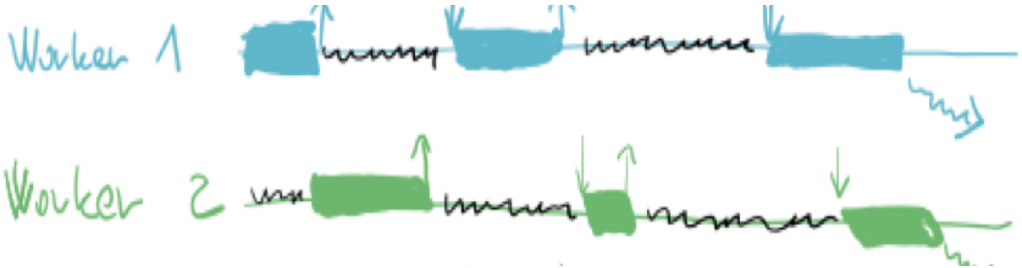
- Synchronous I/O or user-interface code

```
let wc = new WebClient()
let data = wc.DownloadData(url)
outputStream.Write(data, 0, data.Length)
```

- Blocks thread while waiting
  - Does not scale
  - Blocking user interface – when run on GUI thread
  - Simple to write – loops, exception handling etc

Classic Asynchronous programming

- We're used to writing code linearly



- Asynchronous programming requires decoupling Begin from End
- Very difficult to:
  - Combine multiple asynchronous operations
  - Deal with exceptions, cancellation and transaction

Classic Asynchronous Programming

```
let wc = new WebClient()
wc.DownloadDataCompleted.Add(fun e ->
  outputStream.BeginWrite( e.Result, 0, e.Result.Length,
      (fun ar -> outputStream.EndRead(ar)), null))
wc.DownloadDataAsync(url)
```

- Writing the code that performs asynchronous operations is difficult to implement using the current techniques
- Two different programming models
  - **BeginFoo** & **EndFoo** methods
  - **FooCompleted** & **FooAsync** using events
- **Operation completes in different scope**

Anatomy of Async Workflows

```
let getLength url =
  let wc = new WebClient()
  let  data = wc.DownloadString(url)
  data.Length
```

- Easy transition from synchronous
  - Wrap in asynchronous workflow with the ***async*** keyword, use **let!** for async calls and add **return**
  - No need of explicit callback
  - Easy to debug
- Supports loops, recursion, exceptions, cancellation, resource management
- **Operation complete in the same scope**

Anatomy of Async Workflows

```
let getLength url = async {
  let wc = new WebClient()
  let  data = wc.DownloadString(url)
  data.Length }
```

- Easy transition from synchronous
  - Wrap in asynchronous workflow with the ***async*** keyword, use **let!** for async calls and add **return**
  - No need of explicit callback
  - Easy to debug
- Supports loops, recursion, exceptions, cancellation, resource management
- **Operation complete in the same scope**

Anatomy of Async Workflows

```
let getLength url = async {
  let wc = new WebClient()
  let  data = wc.DownloadString(url)
  return data.Length }
```

- Easy transition from synchronous
  - Wrap in asynchronous workflow with the ***async*** keyword, use **let!** for async calls and add **return**
  - No need of explicit callback
  - Easy to debug
- Supports loops, recursion, exceptions, cancellation, resource management
- **Operation complete in the same scope**

Anatomy of Async Workflows

```
let getLength url = async {
  let wc = new WebClient()
  let! data = wc.AsyncDownloadString(url)
  return data.Length }
```

- Easy transition from synchronous
  - Wrap in asynchronous workflow with the ***async*** keyword, use **let!** for async calls and add **return**
  - No need of explicit callback
  - Easy to debug
- Supports loops, recursion, exceptions, cancellation, resource management
- **Operation complete in the same scope**

Asynchronous Workflows

```
let openFileAsynchronous : Async<unit>
  async { use  fs = new  FileStream(@"C:\Program Files\..., …)
          let  data = Array.create (int fs.Length) 0uy
          let!  bytesRead = fs.AsyncRead(data, 0, data.Length)
          do  printfn "Read Bytes: %i, First bytes were:
          %i %i %i ..." bytesRead data.[1] data.[2] data.[3] }
```

- Async defines a block of code we would like to run asynchronously
- We use let! instead of let
  - let! binds asynchronously, the computation in the async block waits until the let! completes
  - While it is waiting it does not block
  - No program or OS thread is blocked

Async – Exceptions & Parallel

Creates an asynchronous computation that executes all the given asynchronous computations queueing each as work items and using a fork/join pattern.

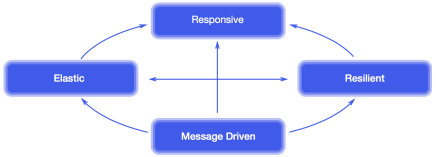
```
let rec fib x = if x <= 2 then 1 else fib(x-1) + fib(x-2)

let fibs =
    Async.Parallel [ for i in 0..40 -> async { return fib(i) } ]
    |> Async.Catch
    |> Async.RunSynchronously
    |> function
        | Choice1Of2 result -> printfn "Successfully %A" result
        | Choice2Of2 exn -> printfn "Exception occurred: %s" exn.Message
```

Async – Limitations

Executing code in parallel there are numerous factors to take into account

- No control of the number of processor cores run simultaneously
- It doesn't consider the existing CPU workload
- There is no throttling of executing threads to ensure an optimal usage
- For CPU-level parallelism, use the .NET Task Parallel Library

Reactive Manifesto



- Responsive
- Message-Driven
- Resilient
- Elastic

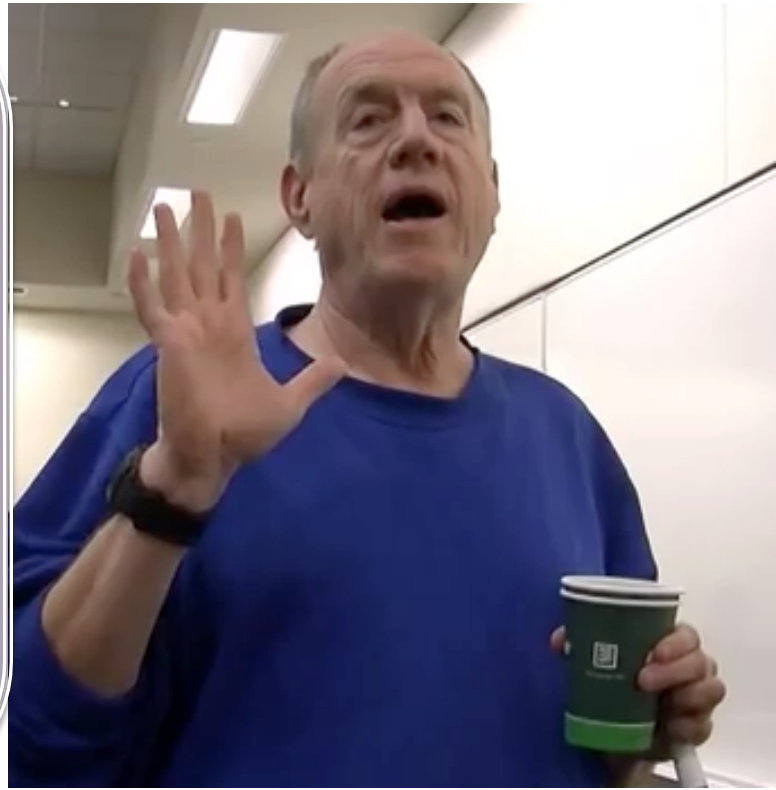
Reactive Systems rely on **asynchronous message-passing** to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing **enables load management, elasticity, and flow control** by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host.

http://www.reactivemanifesto.org

- Processing
- Storage – State
- Communication only by messages
- Share Nothing
- Message are passed by value
- Lightweight object.
- Running on it's own thread.
- No shared state.
- Messages are kept in mailbox and processed in order.
- Massive scalable and lightening fast because of the small call stack.

Carl Hewitt's Actor Model

- **Determine** what to do with the next incoming message
- **Create** more actors
- **Send** messages to another actor



Multi-Threads vs Multi Agents

1 MB per thread (4 Mb in 64 bit) vs 2.7 million Actors per Gigabyte

VS

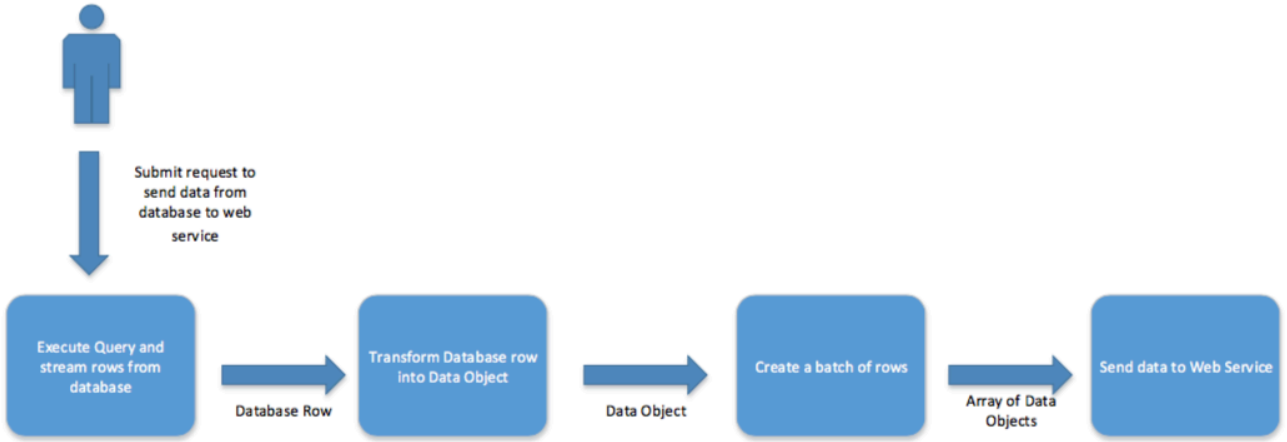
The Solution is Immutability and Isolation

IMMUTABILITY +
ISOLATION =

BEST CONCURRENT MODEL PROGRAMMING

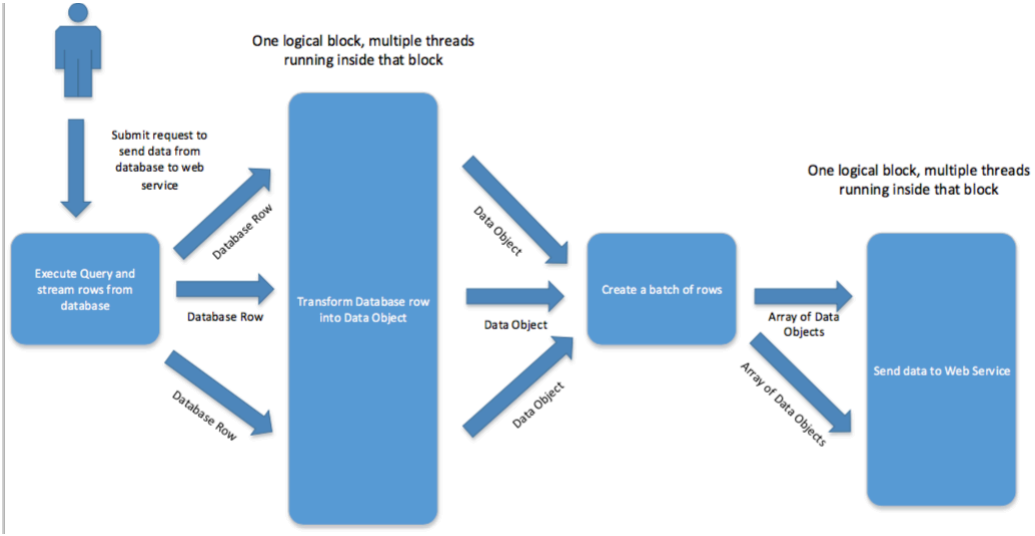
Agent solution

Each block has its own thread
While a message is being transformed another is being fetched from the database.
This is akin to Henry Ford's production line



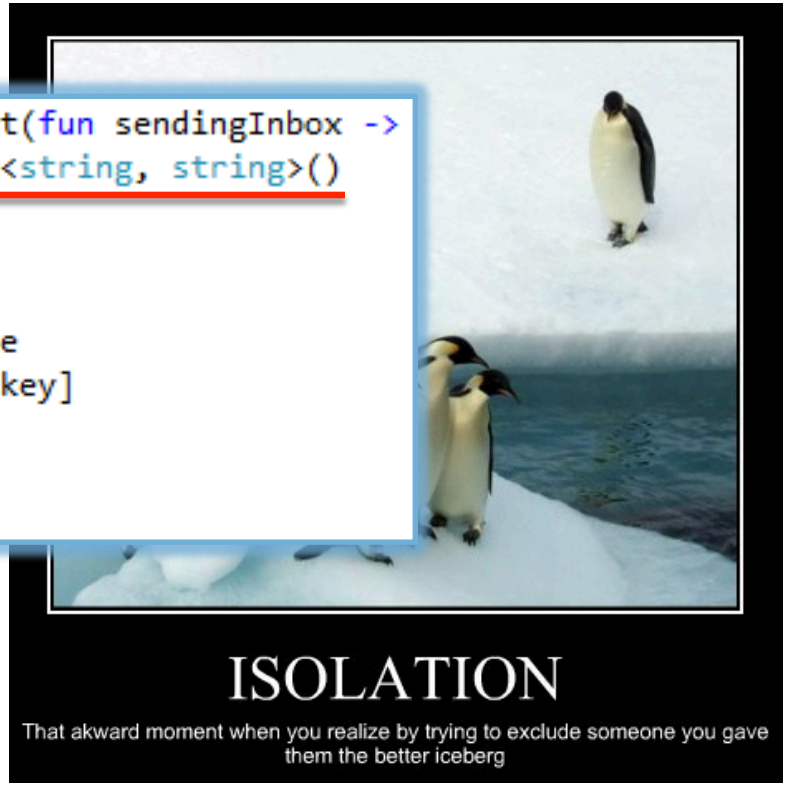
Agent solution

Same structure, just told it now to use many threads per block



Immutability OR Isolation

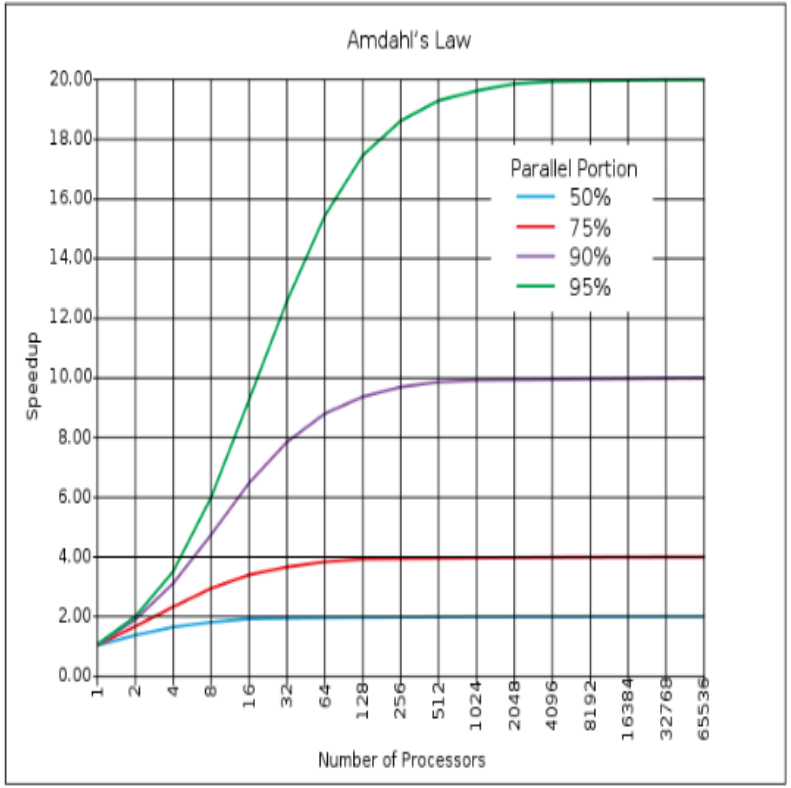
```
let (lockfree:Agent<Msg<string,string>>) = Agent.Start(fun sendingInbox ->
    let cache = System.Collections.Generic.Dictionary<string, string>()
    let rec loop () = async {
        let! message = sendingInbox.Receive()
        match message with
            | Push (key,value) -> cache.[key] <- value
            | Pull (key,fetch) -> fetch.Reply cache.[key]
        return! loop ()
        }
    loop ())
```



The new reality : Amdahl's law

The speed-up of a program using multiple processors in parallel computing is limited by the sequential fraction of the program.

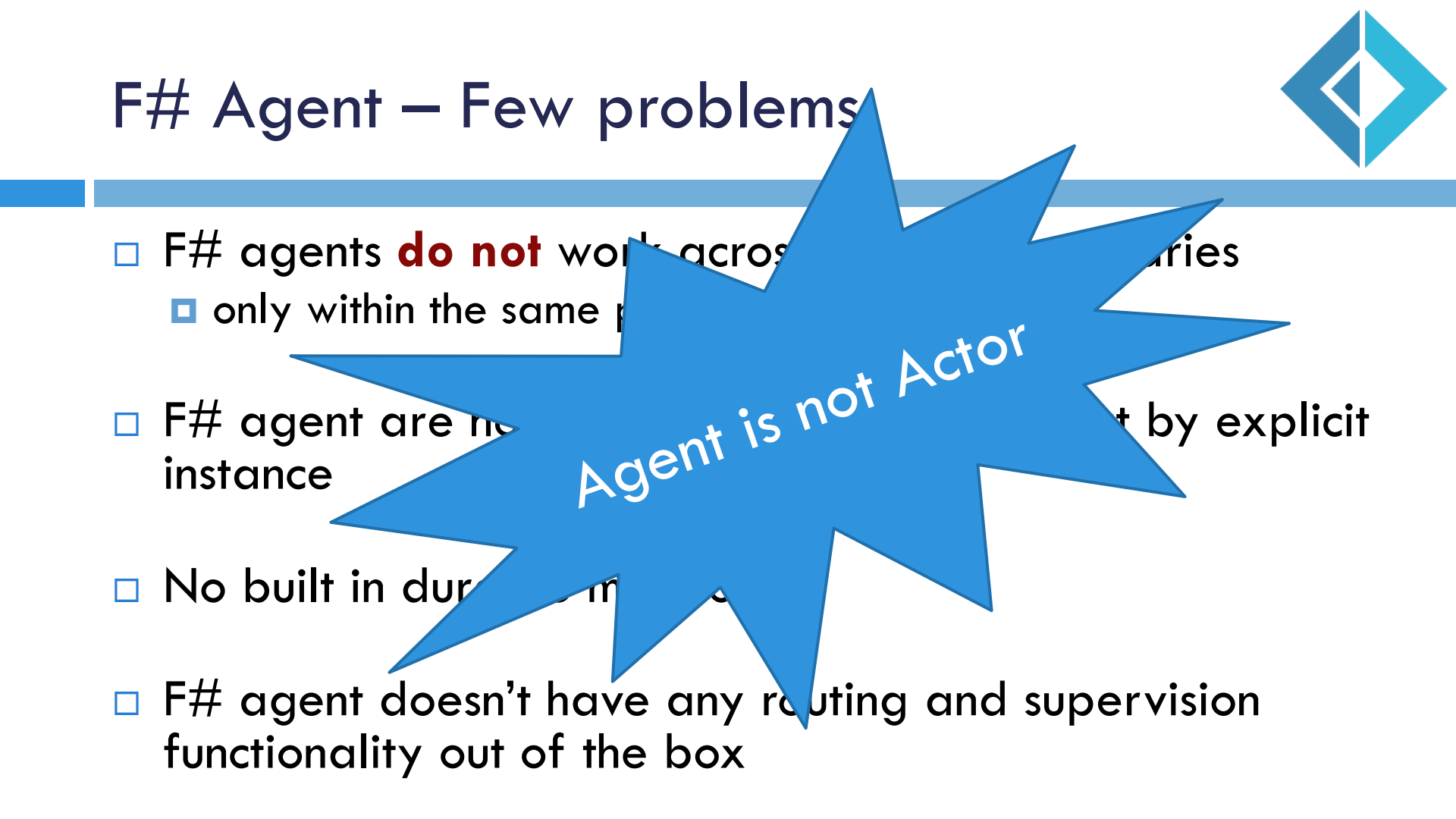
For example if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20 times, no matter how many processors are used



F# MailboxProcessor – aka Agent

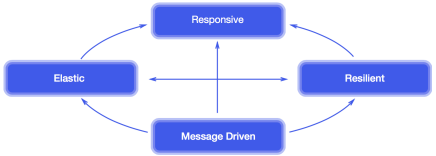
```
let agent = MailboxProcessor<Message>.Start(fun inbox ->
        let rec loop n = async {
            let! msg = inbox.Receive()
            match msg with
            | Add(i) -> return! loop (n + i)
            | Get(r) -> r.Reply(n)
                        return! loop n }
        loop 0)
```

F# Agent – Few problems



- F# agents **do not** wor[illegible] acros[illegible]aries
  - only within the same p[illegible]
- F# agent are n[illegible]t by explicit instance
- No built in dur[illegible]
- F# agent doesn't have any routing and supervision functionality out of the box

Reactive Manifesto



- Responsive
- Message-Driven
- Resilient
- Elastic

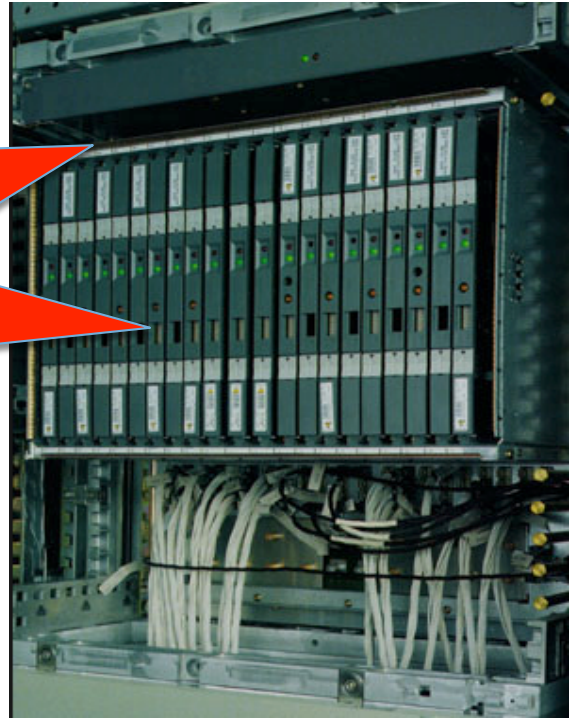
The system **stays responsive in the face of failure**. This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. **Recovery of each component is delegated to another (external) component** and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.

http://www.reactivemanifesto.org

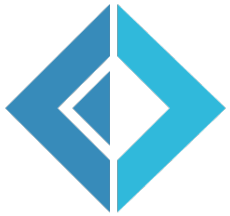
Ericson AXD 301 Switch - 1996

akka.net

99.9999999 percent uptime



Error Handling



- Tasks that are run using Async.RunSynchronously report failures back to the controlling thread as an exception.
- Use Async.Catch or try/catch to handle.
- Async.StartWithContinuations has an exception continuation.
- Supervisor pattern

Async.Catch

```
let asyncTask = async { raise <| new System.Exception(”My
Error!") }

asyncTask
|> Async.Catch
|> Async.RunSynchronously
|> function
   | Choice1Of2 result      ->
                printfn "Async operation completed: %A" result
   | Choice2Of2 (ex : exn) ->
                printfn "Exception thrown: %s" ex.Message
```

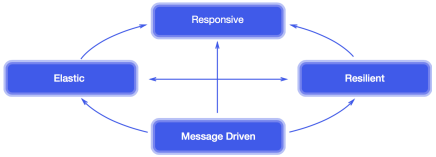
Async - StartWithContinuations

```
let runAgent = MailboxProcessor<Job>.Start(fun inbox ->
        let rec loop n =
            async {
                let! job = inbox.Receive()
                let str = sprintf "Starting job #%d" job.id
                match jobs.ParentId(job.id) with
                | Some id -> printAgent.Post <| sprintf "%s with parentId #%d" str id
                | None -> printAgent.Post str
                // Add the new job information to the list of running jobs.
                jobs.[job.id] <- { jobs.[job.id] with state = JobState.Running }

                Async.StartWithContinuations(job.comp,
                            (fun result -> completeAgent.Post(job.id, result)),
                            (fun _ -> ()),
                            (fun cancelException ->
                                    printAgent.Post <| sprintf "Canceled job #%d" job.id),
                            job.token)

                do! loop (n + 1)
                }
        loop (0))
```

Agent Supervisors

```
let errorAgent =
       Agent<int * System.Exception>.Start(fun inbox ->
         async { while true do
                   let! (agentId, err) = inbox.Receive()
                   printfn "an error '%s' occurred in agent %d" err.Message agentId })

let agents10000 =
       [ for agentId in 0 .. 10000 ->
            let agent =
                new Agent<string>(fun inbox ->
                   async { while true do
                               let! msg = inbox.Receive()
                               if msg.Contains("agent 99") then
                                   failwith "fail!" })
           agent.Error.Add(fun error -> errorAgent.Post (agentId,error))
            agent.Start()
            (agentId, agent) ]
```

Reactive Manifesto



- Responsive
- Message-Driven
- Resilient
- Elastic

The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by **increasing or decreasing the resources allocated** to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, **scaling algorithms** by providing relevant live performance measures. They achieve elasticity in a cost-effective way on commodity hardware and software platforms.

http://www.reactivemanifesto.org

Scaling agents on demand

```
let urlList = [ ("Microsoft.com", "http://www.microsoft.com/");
                ("MSDN", "http://msdn.microsoft.com/");
                ("Google", "http://www.google.com") ]

let processingAgent() = Agent<string * string>.Start(fun inbox ->
                        async { while true do
                                let! name,url = inbox.Receive()
                                let uri = new System.Uri(url)
                                let webClient = new WebClient()
                                let! html = webClient.AsyncDownloadString(uri)
                                printfn "Read %d characters for %s" html.Length name })

let scalingAgent : Agent<(string * string) list> = Agent.Start(fun inbox ->
                                   async { while true do
                                           let! msg = inbox.Receive()
                                           msg
                                           |> List.iter (fun x ->
                                                         let newAgent = processingAgent()
                                                         newAgent.Post x )})
```

Goals - Plan of the talk

Applications must be built with concurrency in mind
(possibly from the beginning)

Programmers must embrace concurrent programming

F# really **shines** in the area of distributed computing

Functional Programming helps to go Reactive


DEMO

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

-- Edsger Dijkstra

How to reach me

github.com/rikace/Presentations/FRP

meetup.com/DC-fsharp

@DCFsharp @TRikace

rterrell@microsoft.com


That's all Folks!