signalframe

Declarative Concurrent Programming in Streaming Systems

Srdjan Marinovic

June, 2019

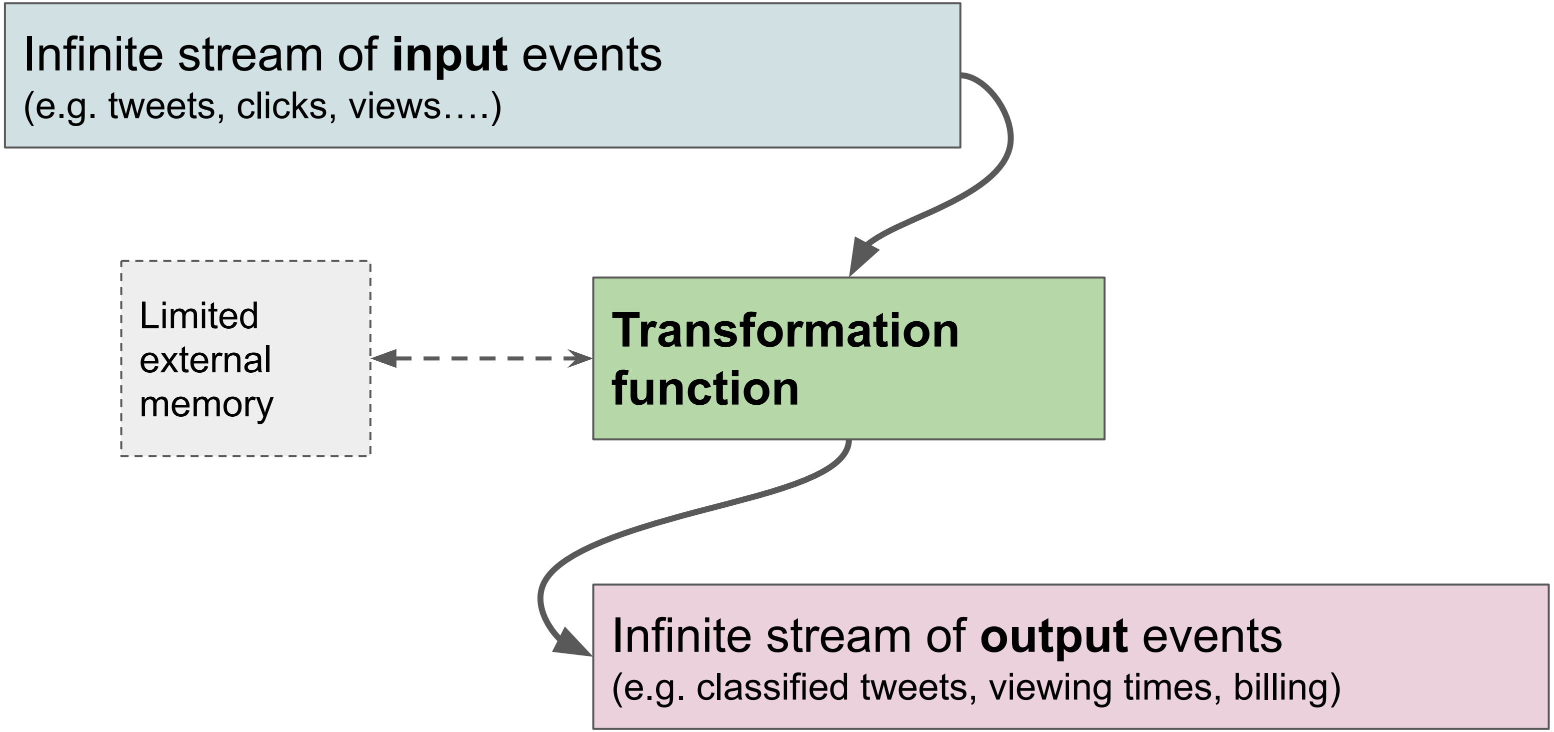
Overview

- Streaming systems overview
- Process-calculi concurrency languages
- Proposed abstractions for declarative concurrency

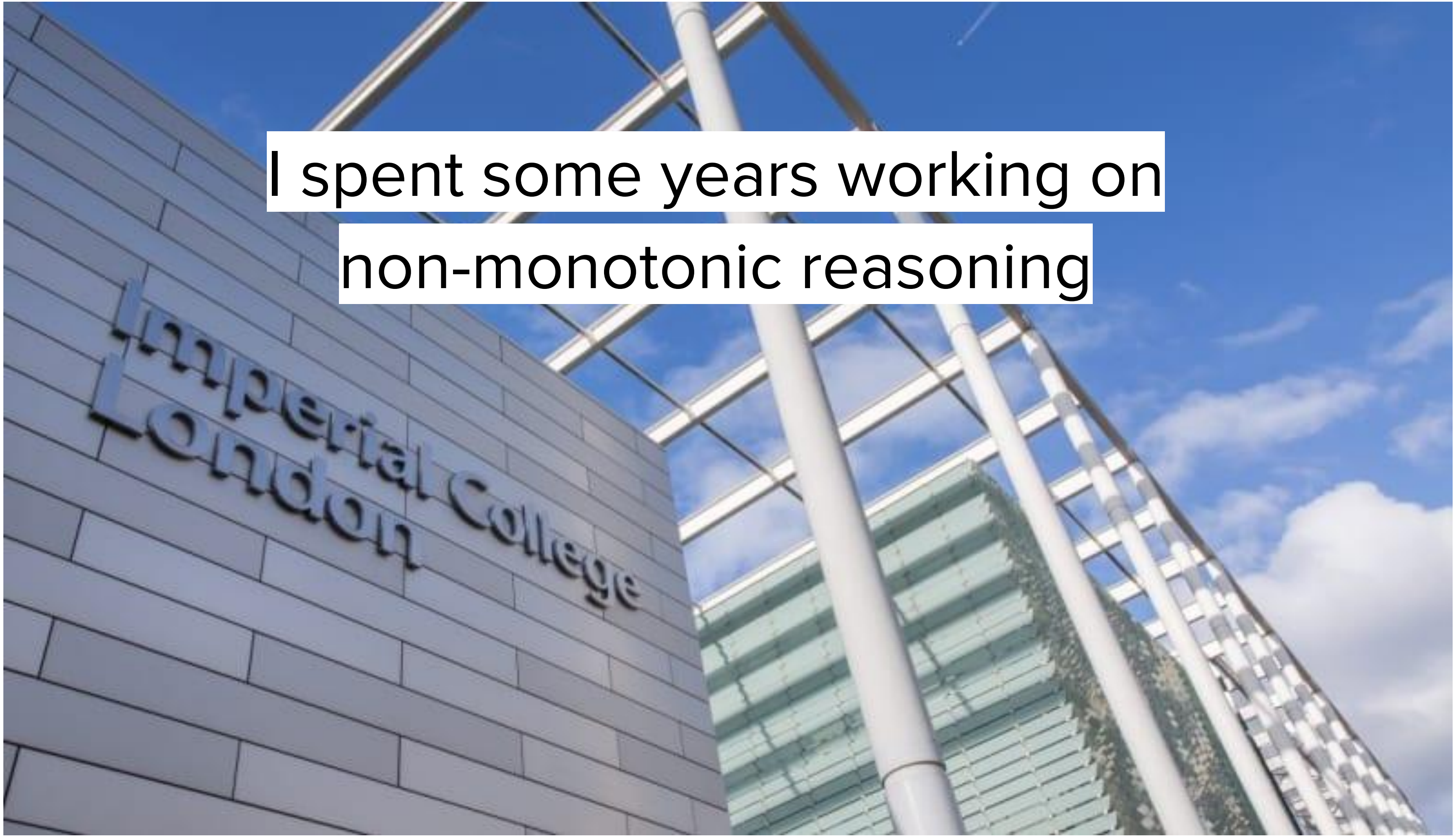
1

Streaming systems overview

Streaming system



I spent some years working on non-monotonic reasoning



I moved to ETH Zurich and started investigating streaming systems

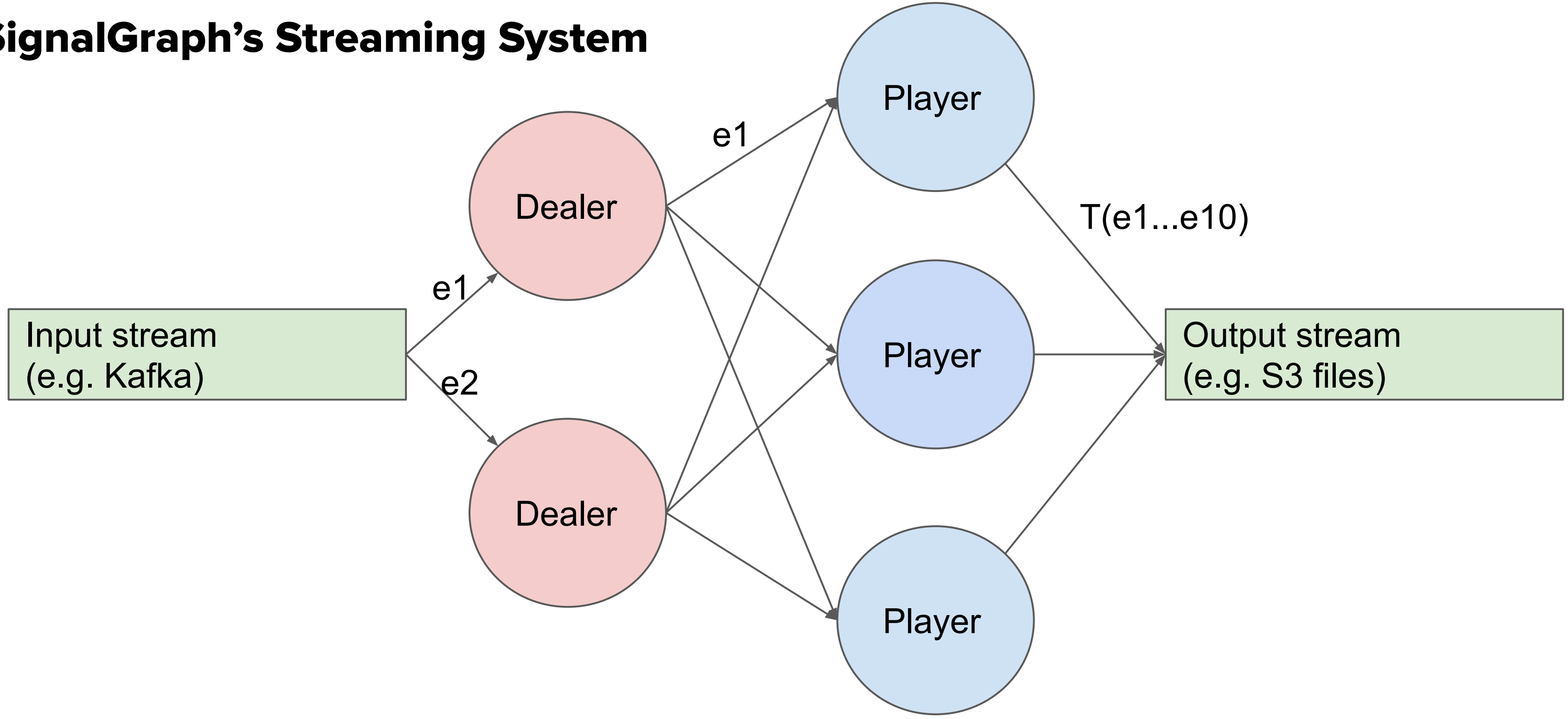
Streaming system problem:

How to reason over incomplete and infinite data?

then I started building a streaming system

- SignalGraph is a streaming graph platform with 10+ billion events/day.
- **Vertices** are wifi and BLE signals.
- **Events** denote **temporal edges** (observations of proximal signals).
- Events are clustered into graph embeddings for similarity analysis.

SignalGraph's Streaming System



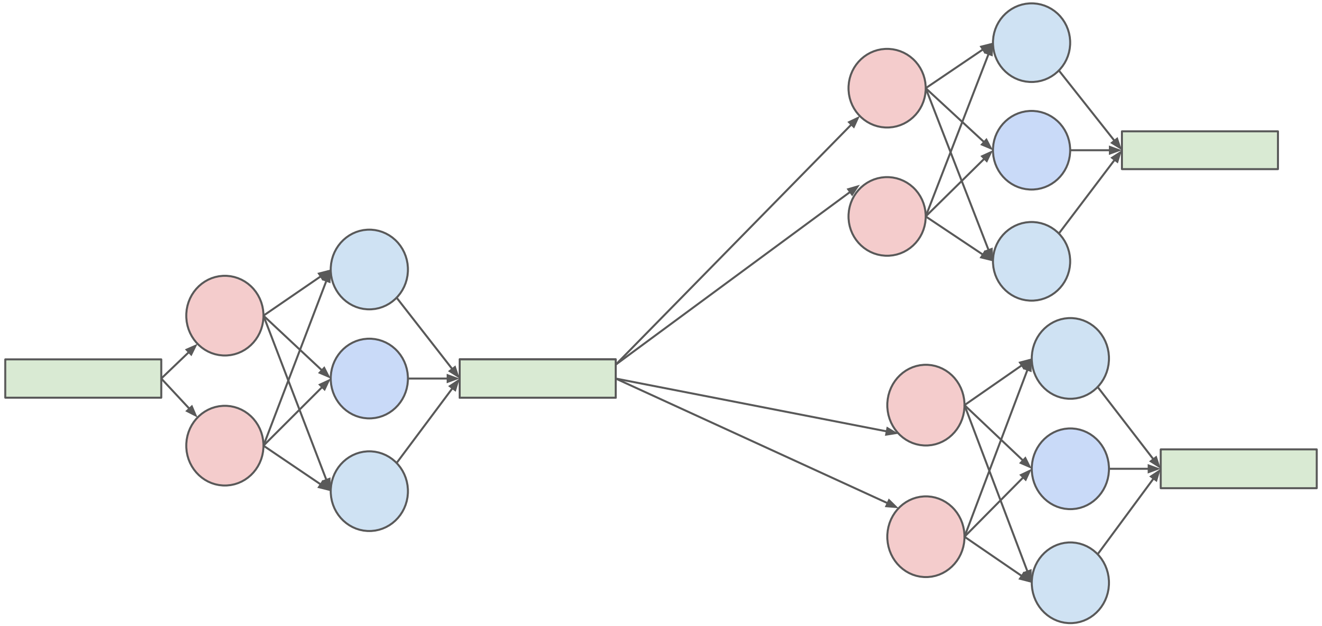
Players implement event transformations.
Dealers distribute events to players.

Development Requirements


```
Transform(in stream): out stream {
     Map(e:in, hash(e)) -> out
}

Transport(in stream, client gRPC){
     Map(e:in, resolve(e)) -> client
}

KafkaSrc -> Transform -> Transport
```

```
Transform(in stream): out stream {
     Map(e:in, count(e))
      -> Reduce(emit(24h,hll_agg))
         -> out
}

Transport(in stream, client s3){
     Map(e:in, new_file(e)) -> client
}

Dealer -> Transform -> Transport
```

SignalGraph’s Streaming Composition



Operating Environment

- Each process (dealer/player) must be highly concurrent
  - handle streams of 10bil events/day
- Processes fail regularly with a known distribution
  - OOM, Network partitions
- Running on Kubernetes
  - CPU and Network starvation at times

Operating Requirements

- Concurrency controls
- Prioritized execution
- (Lossy) backpressure control
- Cancellations and restarts of transformations

(Revised) Streaming systems problem:

How to **structure a system** that reasons over incomplete and infinite data?

2

Process-calculi concurrent programming

Modern process-calculi based languages

Processes, channels, messages

- **Go language**
  - Popular and growing (Kubernetes, Cockroach)
- **Rust language**
  - Growing fast (Servo, Firecracker)
  - Library support for CSP primitives

Are such primitives good abstractions for building highly-concurrent streaming systems?

Go’s concurrency

Producer

```
go func(input chan string) {
    for e := range input {
        i := hash(e)
        chs[i] <- e
    }
}(input)
```

Consumers

```
chs := make([]chan string, 10, 10)
for i:=1; i<10; i++{
    chs[i] = make(chan string,10)
    go func(i int) {
        for str := range chs[i]{
            count(str)
        }
    }(i)
}
```

Possible traces for the previous example

- `count("e1"), count("e2"), count("e3"), count…`
- `deadlock (count blocks internally)`

Channels are of fixed-size and blocking.

Go's concurrency... gets tricky

Producer

```
go func(input chan string) {
    for e := range input {

        go func(e string){
            i := hash(e)
            chs[i] <- e
        }(e)

    }
}(input)
```

Consumers

```
chs := make([]chan string, 10, 10)
for i:=1; i<10; i++{
    chs[i] = make(chan string,10)
    go func(i int) {
        for str := range chs[i]{
            count(str)
        }
    }(i)
}
```

Possible traces for the previous example

- `count("e1"), count("e2"), count("e3"), count…`
- `low-throughput (CPU starvation)`

Goroutines are cheap, but we cannot spawn infinitely many.

Cancelling go routines

```
import "context"

ctx, cancel := context.WithTimeout(ctx, 100*time.Millisecond)

go func(ctx context.Context)
    for {
        select{
        case <-ctx.Done():
        case str := <- ch:
            f_a(str)
    }}
}(ctx)
```

Cancelling go routines... gets tricky

```
go func(ctx context.Context)
    for {
        select{
        case <-ctx.Done():
        default:
        }

        select{
        case <-ctx.Done():
        case str := <- ch:
            f_a(str)
        }
    }
}(ctx)
```

No language-level support for

- Channel transformations

- Non-blocking cancellations
- Restarts after panics
- Bounded concurrency
- Prioritizing executions

*Note: This is not a criticism of process-calculi languages.

Process-calculi primitives are not the right abstractions for concurrent transformations.

3

Declarative concurrent programming

Generic Declarative Programming

Say **what** the computation is,

not how to manage it.

(WIP) Declarative Concurrent Programming Abstractions

Abstractions	Layer
Stream MapReduce, Futures	Declarative Layer
Streams, Queues	Data Structure Layer
Job Runners, **GoSafe**	Management Layer

GoSafe (High-level)

```
func GoSafeWithAutoRestart(fn func(), onFail func(), ctx context.Context) {
  SafeGo(fn).
    OnFail(onFail).
    WithAutoRestart(structs.DefaultExponentialBackoff()).
    WithContext(ctx).
    Go()
}
```

GoSafe (High-level)

```go
func (g *goSafe) Go() {
	rootCatcher := exceptions.Catch(
		rootHandler(g.failure(), g.logger, g.apm, g.name()))

	go func() {
		if g.stop() {
			return
		}
		g.fn()
	}()
}
```

Job Runner (High-level)

```
func (b *JobRunner) run(job Job, ctx context.Context) {
  ...
    running := atomic.LoadInt32(&b.running)
    nWorkers := atomic.LoadInt32(&b.nWorkers)
    canRun = running < nWorkers
    if !canRun {
      b.queue.Push(&jobWithContext{job: job,ctx: ctx,})
      return
    }
    if atomic.CompareAndSwapInt32(&b.running, running, running+1) {
      SafeGo(func() {
        defer b.checkAndDrain()
        job.Execute(ctx)
      }).Go()
     ...
```

Stream (High-level)

```
func NewStream(c context.Context) *Stream {
   ...
   stream := &Stream{
     nonBlockQ: NewQueue(cc),
     ctx:       ctx,
     cancel:    cancel,
     ccancel:   ccancel,
     done:      int32(0),
     timeout:   DefaultStreamTimeout,
   }
   stream.cond = NewConditionVar(cc, r, stream)
   return stream
 }
```

Stream (High-level)

```
func (s *Stream) Produce(element interface{}) (ret bool) {
   select {
   case <-s.ctx.Done():
     return false
   default:
   }
   select {
   case <-s.ctx.Done():
     return false
   default:
     s.nonBlockQ.Send(element)
     s.cond.Signal()
     return true
   }
 }
```

Stream (High-level)

```
func (s *Stream) Consume() (interface{}, bool) {
  for {
    element, ok := s.nonBlockQ.Recv()
    if !ok {
      select {
      case <-s.ctx.Done(): return nil, false
      default:
      }
      s.wait(func() bool {
        !s.Done()
      })
      continue
    }
    return element, true
  }}
```

Streaming MapReduce (High-level)

```
stream := patterns.NewStream(ctx)
defer stream.Cancel()

patterns.InBackground(ctx, func(ctx context.Context) {
    for i, _ := range clusters[:len(clusters)-1] {
      for k, _ := range clusters[i+1:] {
        j := i + 1 + k
        if !stream.Produce([]int{i, j}) {
          break
        }
      }
    }
    stream.End()
 })
```

Streaming MapReduce (High-level)

```
patterns.MapReduce(stream,
    func(ctx context.Context, i interface{}) interface{} {
      hc1 := clusters[i[0]]; hc2 := clusters[i[1]]
      if hc1.SimilarityWithFrequencyScore(hc2) >= setSimilarity {
        return i
      }
      return nil
    },
    func(results interface{}, result interface{}) interface{} {
      graph.AddEdge(result[0], result[1])
      return nil
    }
)
```

signalframe

Summary:

1. Streaming systems are difficult to build.
2. Highly concurrent processes are typically needed.
3. Standard CSP/Pi-calculus primitives are too low-level.
4. In our case, declarative primitives yield more stable systems.

Extending the primitives has paid off,
but we are still making mistakes and learning.

signalframe

This is a massive team effort:

Oliver Bose

Brian Wilke

Kevin Hummel

signalframe

Thank you.