Naiad

a *timely dataflow* model

What’s it hoping to achieve?

1. high throughput

2. low latency

3. incremental computation

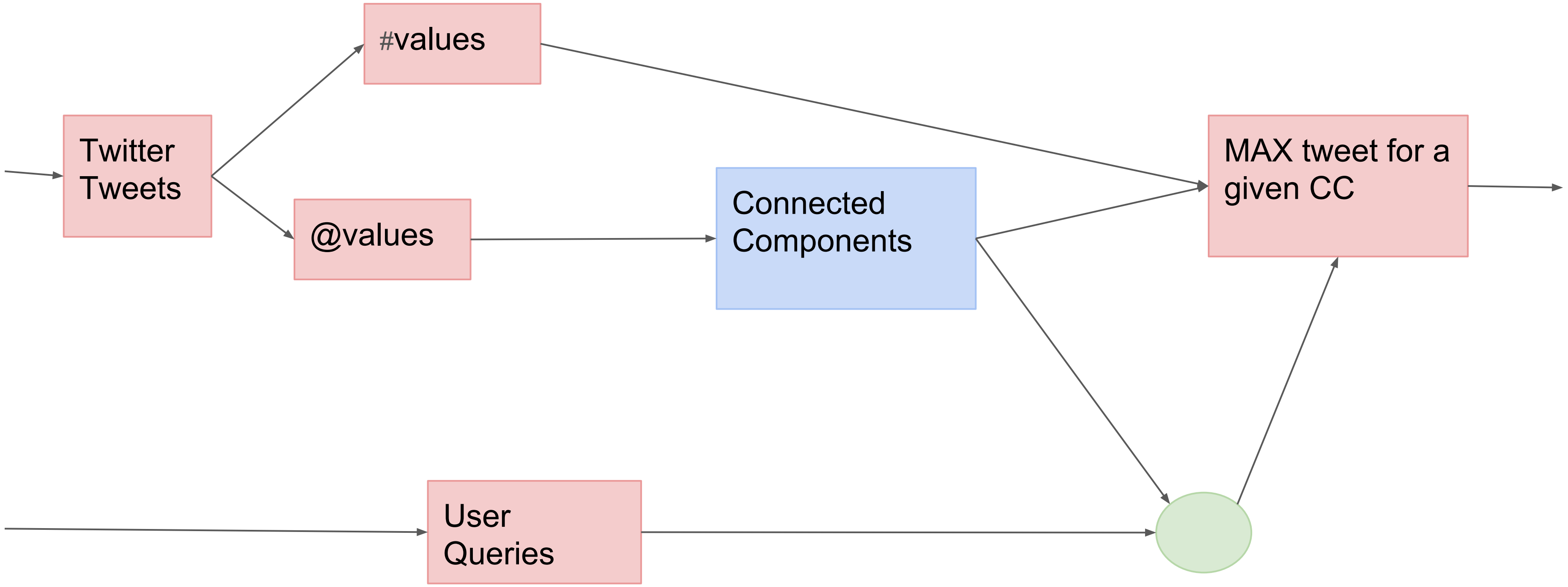
Why?

→ So much data!

Problems with other, contemporary *dataflow* systems:

1. Too specific (e.g. Map-Reduce, Hadoop)
2. **Batch-based systems**
3. **Graph-based systems**
4. **Stream processing systems**

An Example: Streaming via Twitter

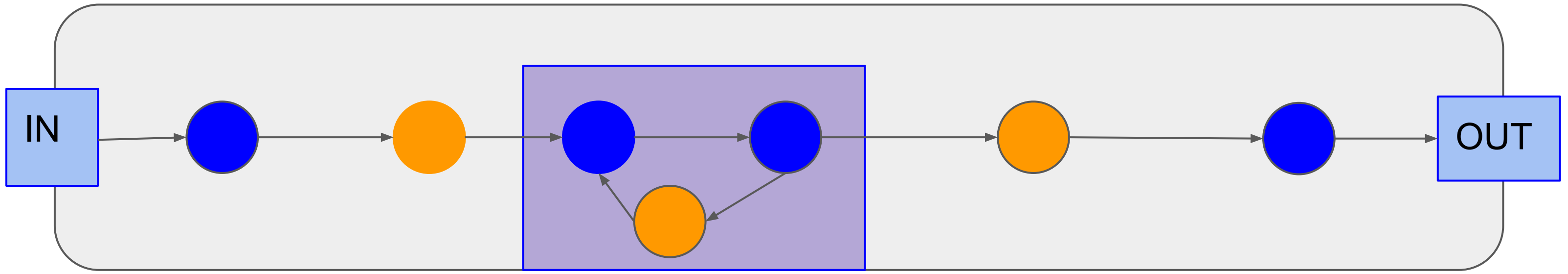


A **new** computational model: *timely dataflow*

→ structured loops

→ stateful dataflow vertices

→ notifications for vertices



Notifications for Vertices

Vertex methods:

v.OnRecv(e:Edge, m:Message, t:Timestamp)

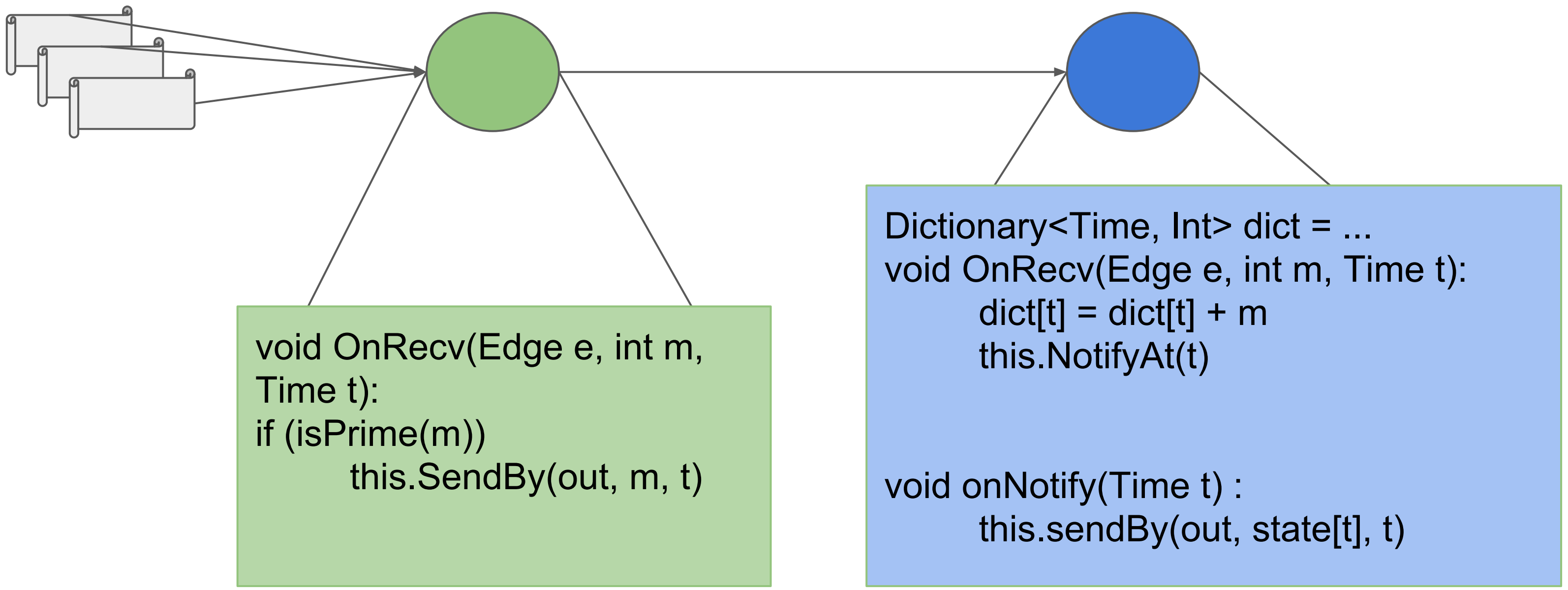
v.OnNotify(t:Timestamp)

System-provided methods:

this.SendBy(e:Edge, m:Message, t:Timestamp)

this.NotifyAt(t:Timestamp)

An Example Program

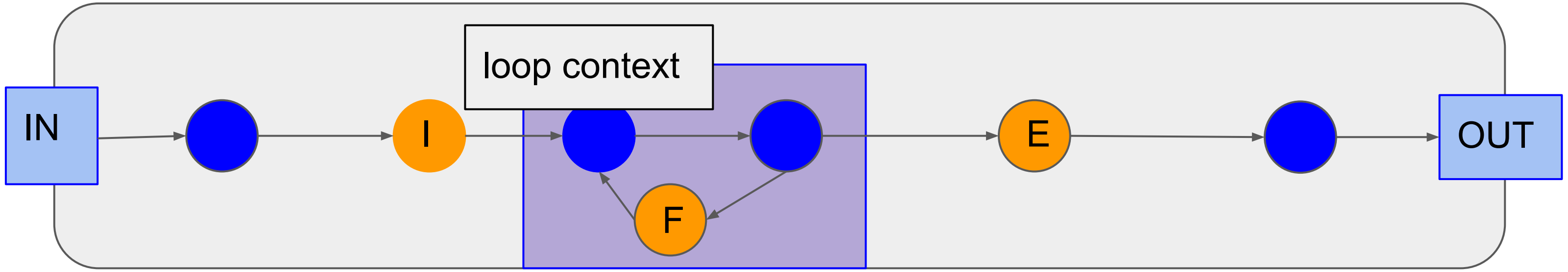


```
void OnRecv(Edge e, int m,
Time t):
if (isPrime(m))
        this.SendBy(out, m, t)
```

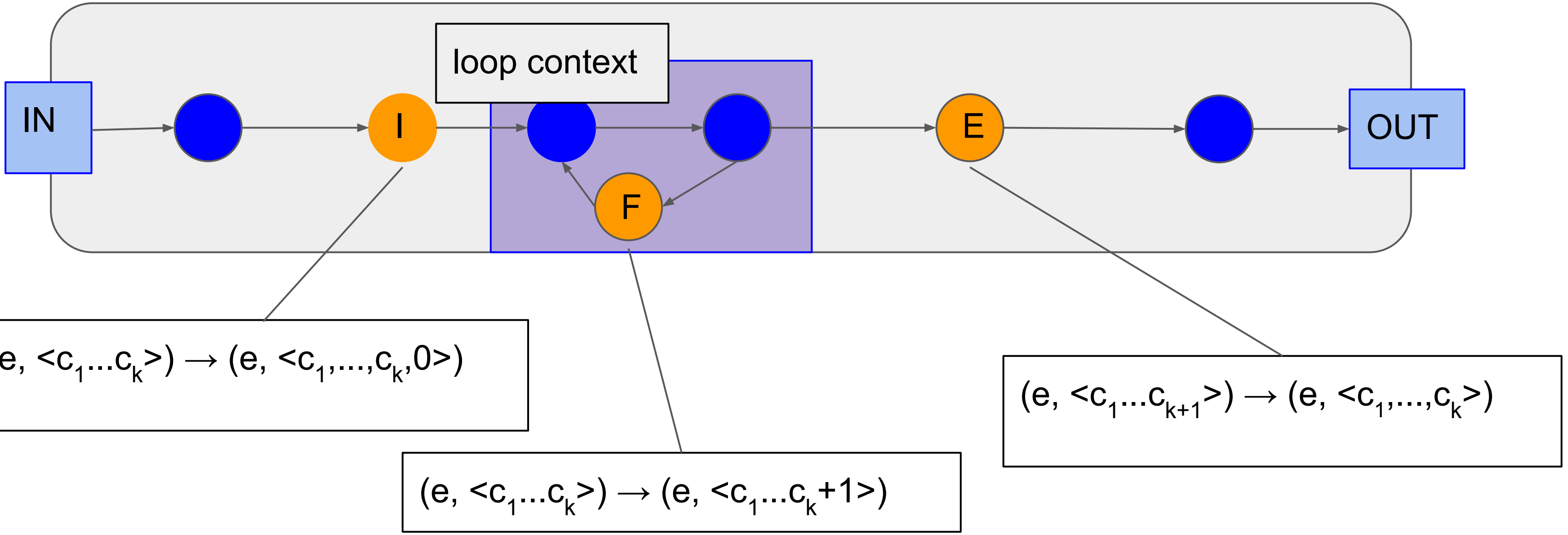
```
Dictionary<Time, Int> dict = ...
void OnRecv(Edge e, int m, Time t):
        dict[t] = dict[t] + m
        this.NotifyAt(t)


void onNotify(Time t) :
        this.sendBy(out, state[t], t)
```

Structured Loops & Stateful Vertices



Timestamps: (e ∈ ℕ, $\langle c_1 \ldots c_k \rangle$ in N^k)

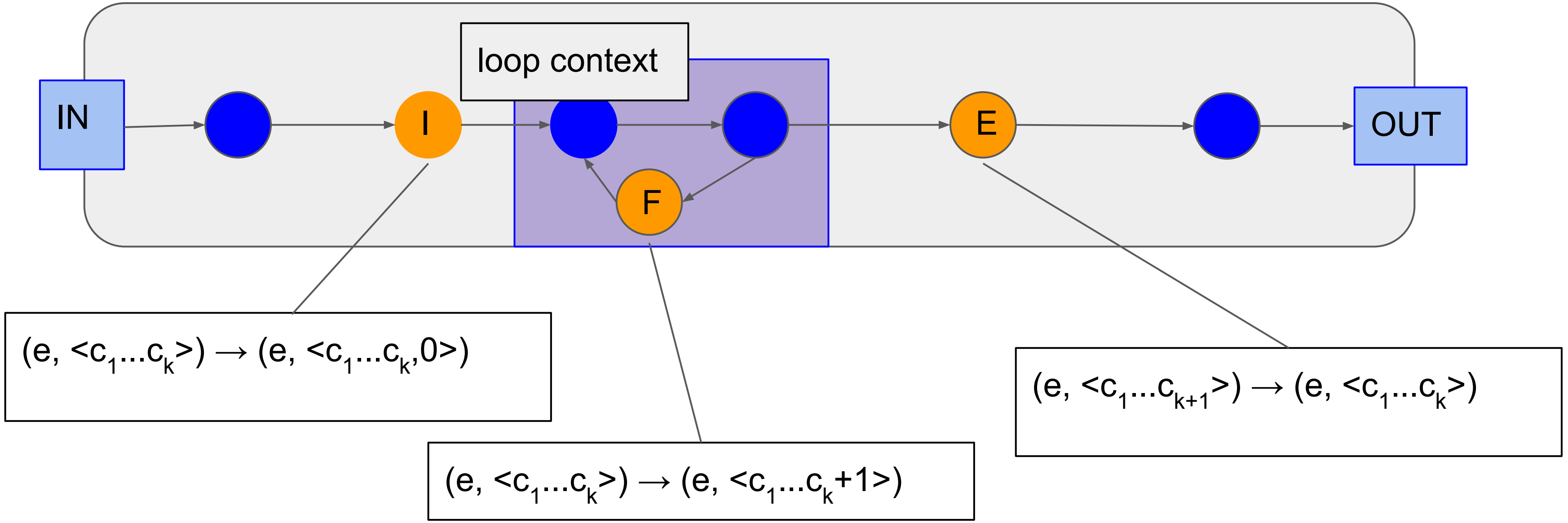


(e, $\langle c_1 \ldots c_k \rangle$) → (e, $\langle c_1, \ldots, c_k, 0 \rangle$)

(e, $\langle c_1 \ldots c_k \rangle$) → (e, $\langle c_1 \ldots c_k + 1 \rangle$)

(e, $\langle c_1 \ldots c_{k+1} \rangle$) → (e, $\langle c_1, \ldots, c_k \rangle$)

Timestamps: ($e \in \mathbb{N}$, $\langle c_1 \ldots c_k \rangle$ in N^k)


IN
I
loop context
F
E
OUT
$(e, \langle c_1 \ldots c_k \rangle) \rightarrow (e, \langle c_1 \ldots c_k, 0 \rangle)$
$(e, \langle c_1 \ldots c_k \rangle) \rightarrow (e, \langle c_1 \ldots c_k + 1 \rangle)$
$(e, \langle c_1 \ldots c_{k+1} \rangle) \rightarrow (e, \langle c_1 \ldots c_k \rangle)$

$\{t_1 = (x_1, \mathbf{c_1})\} \ \square \ \{t_2 = (x_2, \mathbf{c_2})\} \Leftrightarrow x_1 \ \square \ x_2 \ \& \ \mathbf{c_1} \ \square \ \mathbf{c_2}$

A Single-Threaded scheduler

Pointstamp : ($t \in$ Timestamp, $l \in$ Edge $\cup$ Vertex)

- *could-result-in*: $(t_1,l_1) \le (t_2,l_2) \Leftrightarrow \Phi[l_1,l_2](t_1) \le t_2$

1. maintains a set of *active pointstamps*
2. maintains an *occurrence count*
3. maintains a *precursor count*

A Single-Threaded scheduler: in action

1. A pointstamp P becomes active
    a. initialize precursor count to number of existing active pointstamps that *could-result-in* P
    b. increment precursor count of any pointstamp P *could-result-in*

2. A pointstamp P leaves the active set (occurrence count = 0)
    a. decrement precursor count of any pointstamp P *could-result-in*

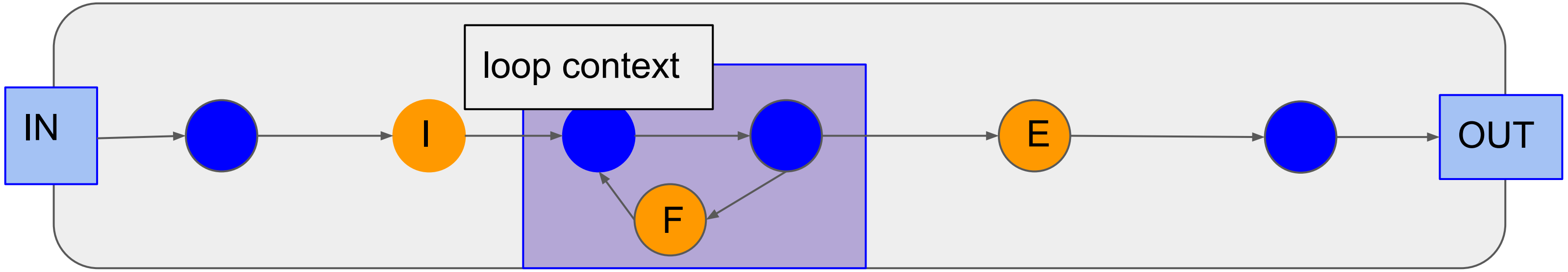
3. A pointstamp P reaches the *frontier* of active pointstamps (precursor count = 0)
    a. scheduler can deliver any notification originating from P

A Single-Threaded scheduler: in action

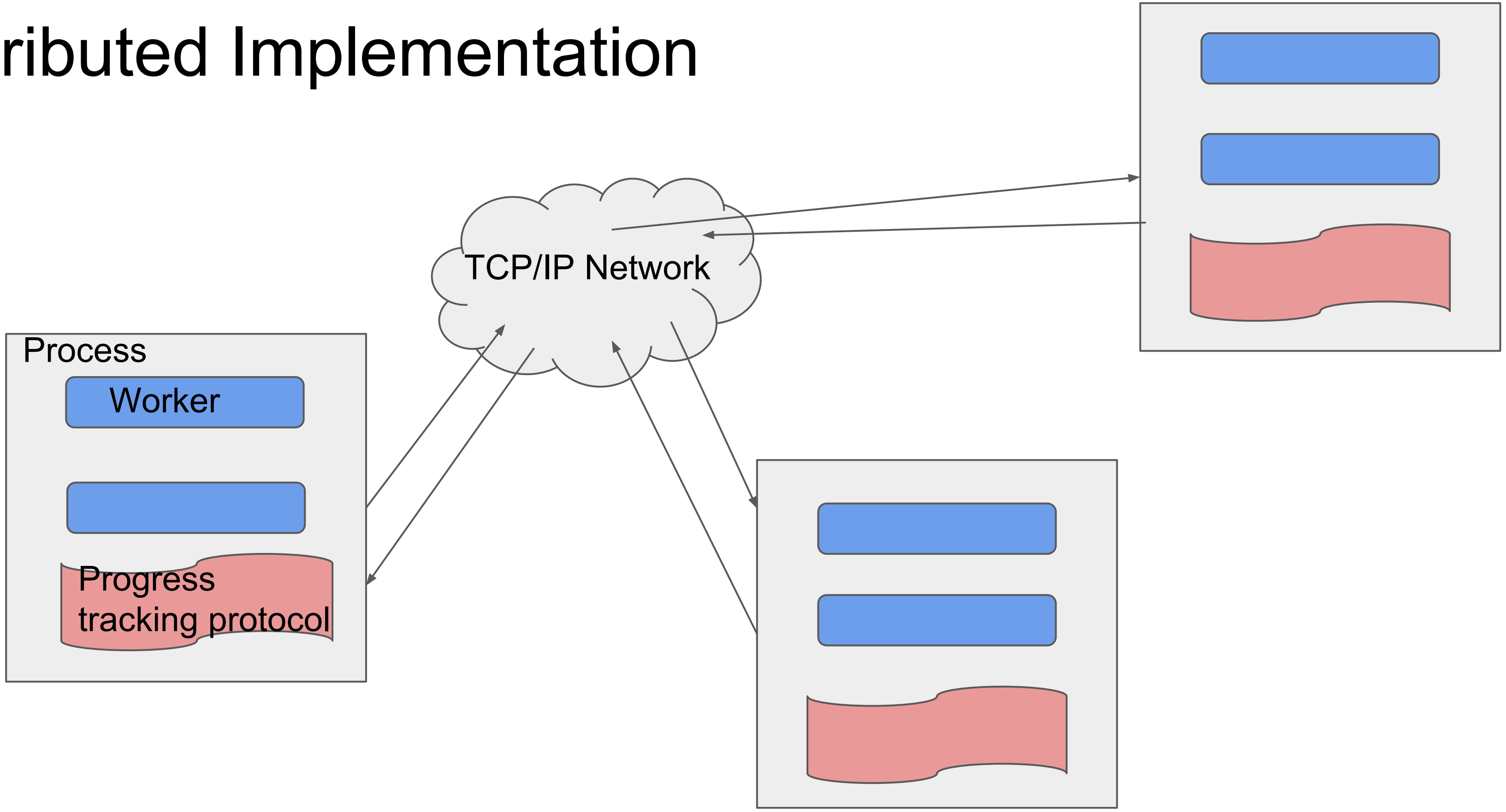
1. A pointstamp P becomes active
   a. initialize precursor count to number of existing active pointstamps that *could-result-in* P
   b. increment precursor count of any pointstamp P *could-result-in*

2. A pointstamp P leaves the active set (occurrence count = 0)
   a. decrement precursor count of any pointstamp P *could-result-in*

3. A pointstamp P reaches the *frontier* of active pointstamps (precursor count = 0)
   a. scheduler can deliver any notification originating from P

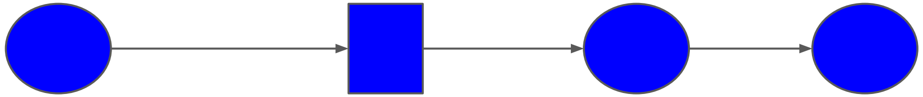


Distributed Implementation

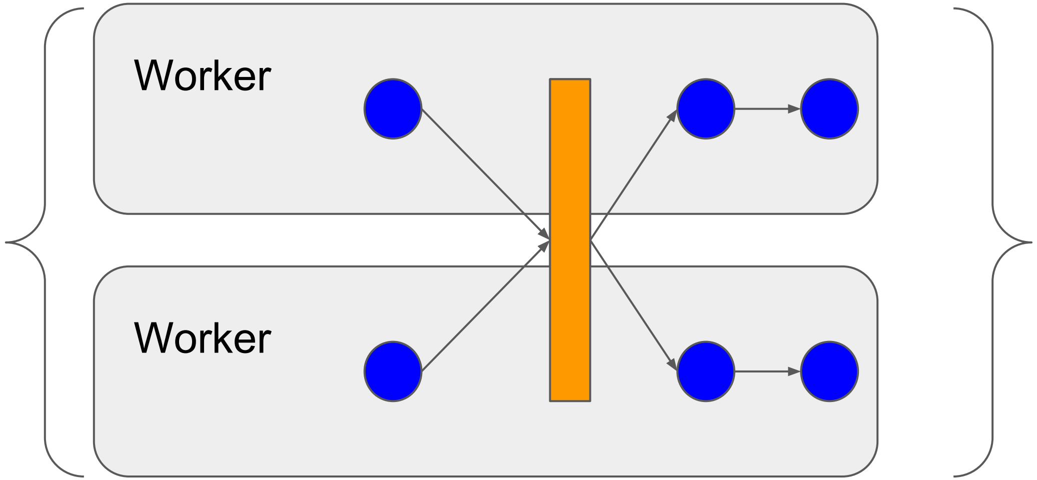


Data parallelism: how do we achieve it?

Logical Graph:



Physical Graph:



Distributed Progress Tracking

For each active pointstamp, a worker maintains its version of the global state:

- a *local occurrence count*
- a *local precursor count*
- a *local frontier*

Distributed Progress Tracking

For each active pointstamp, a worker maintains its version of the global state:

- a *local occurrence count*
- a *local precursor count*
- a *local frontier*

Optimisations:

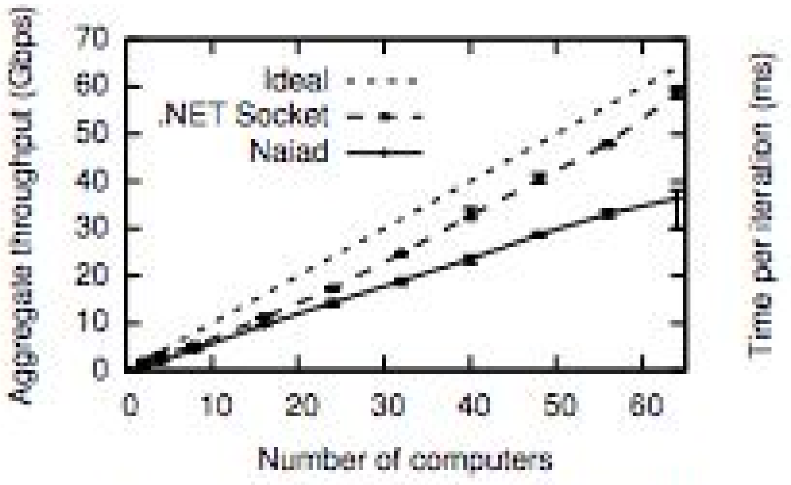
1. projected pointstamps
2. use a local buffer
3. use UDP packets for updates before sending via TCP
4. threads can be woken either by a broadcast or unicast notifcation

Results: Throughput

Benchmark: construct a cyclic dataflow network which repeatedly performs an all-to-all data exchange

1. linear scaling

2. not ideal



(a) All-to-all exchange throughput (§5.1)

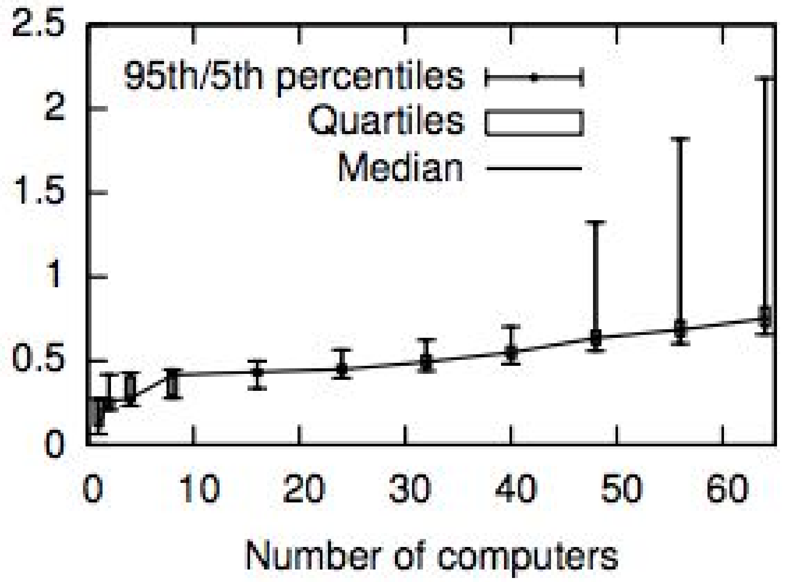
Results: Latency

Benchmark: construct a simple cyclic graph in which vertices request/receive completeness notifications

- median time: 753 us

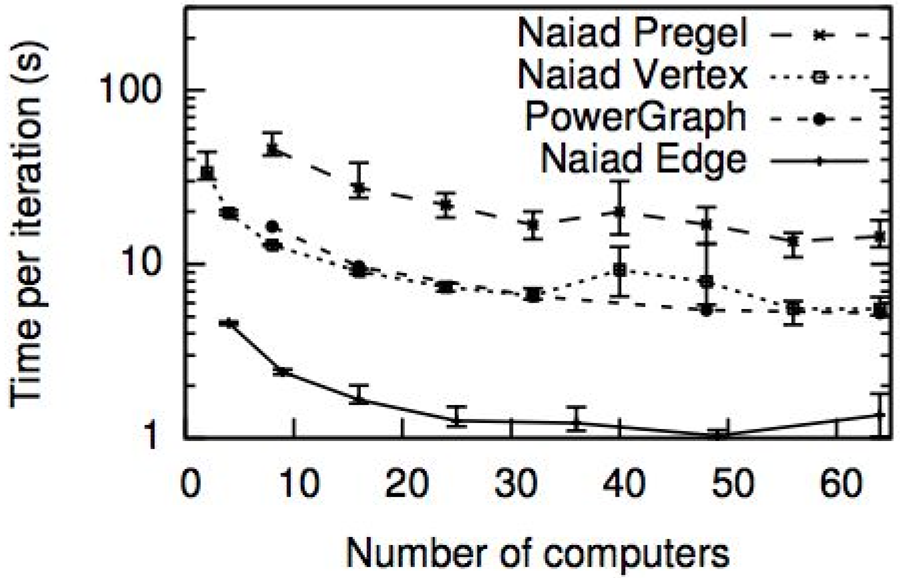
Caveat: Micro-stragglers

1. Networking: TCP over Ethernet
2. Data structure contention
3. Garbage Collection



(b) Global barrier latency (§5.2)

Results: PageRank using Twitter

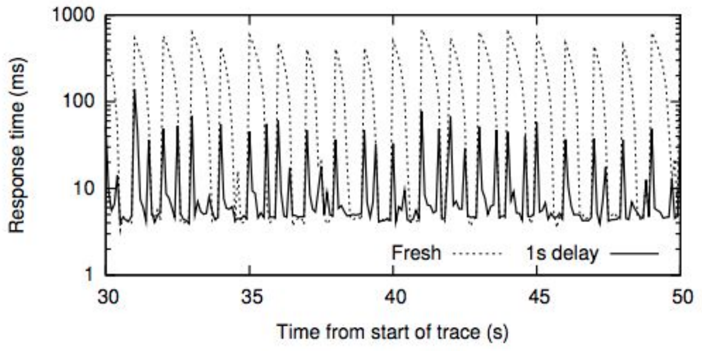


Results: Incremental computation

Benchmark: in a continually arriving stream of tweets, extract hashtags and mentions of other users to determine the most popular hashtag for a given user.

Setup:

1. two inputs for the stream of tweets and requests
   a. fed into an incremental computation
2. introduce 32,000 tweets per second
3. add a new query every 100 ms



Strengths

1. Generality

2. Simplicity

3. Incremental computation for iterations

4. Fine-grained control over partitioning

Weaknesses (on my opinion)

1. Do not test latency and throughput together

2. Though, using Naiad can achieve some substantial improvements, this depends on implementation

3. Use lines of code to measure simplicity

4. Stragglers

Limitations

1. Naiad is specifically designed for problems in which the working set fits in the total RAM of the cluster

2. Fault tolerance

Takeaway & Impact

timely-dataflow computational model is powerful because of:

1. Incremental and iterative computation

2. A general, lightweight, framework for data-parallel applications that focusses on a wide domain (e.g. not just loops) while offering low-latency and high throughput