NAIAD: A Timely Dataflow System

**Derek G. Murray, Frank McSherry, Rebecca Isaacs,
Michael Isard, Paul Barham, Martín Abadi
(Microsoft Research Silicon Valley)**
2013

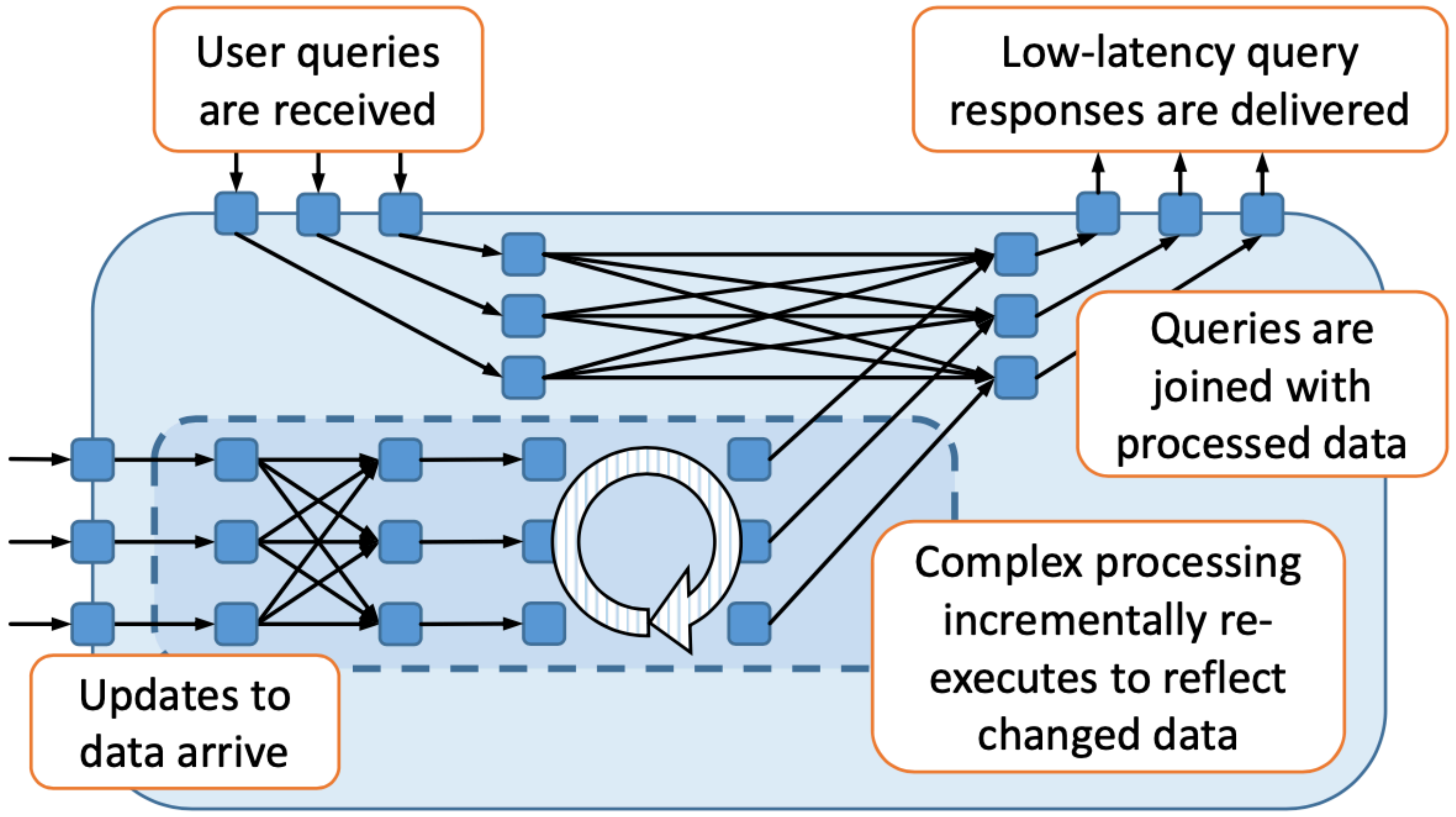
Alexander Frost for R244

Presentation Structure

- What is NAIAD?
- Brief recap: batch + stream processors
- What is timely dataflow?
- NAIAD framework
- Conclusion and discussion

What is NAIAD?

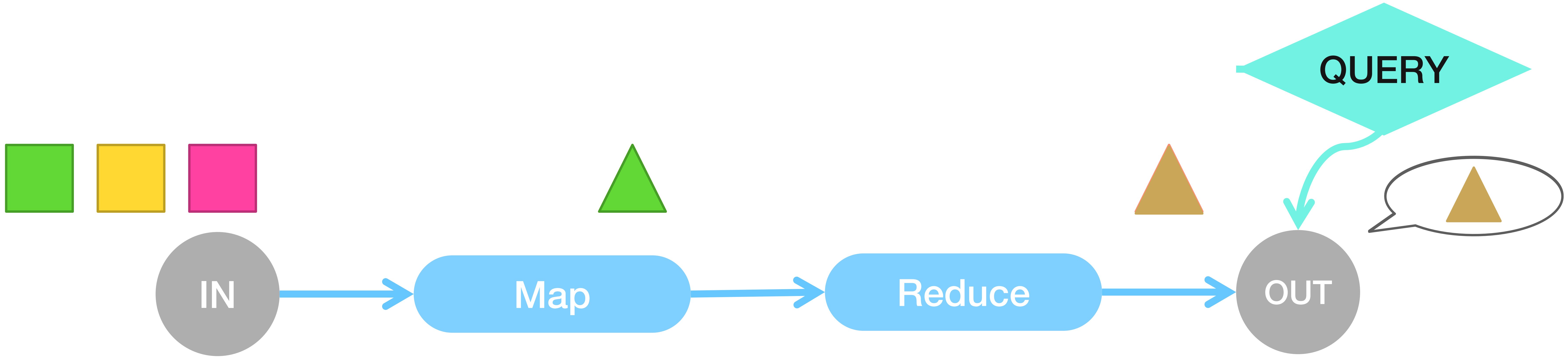
- Want the ***high throughput*** of batch processing systems?
- And the ***low latency*** of stream processors?
- Do you also need to handle ***incremental and iterative computation***?
- Naiad does ***all of this and more!!***



Brief recap

Batch processors

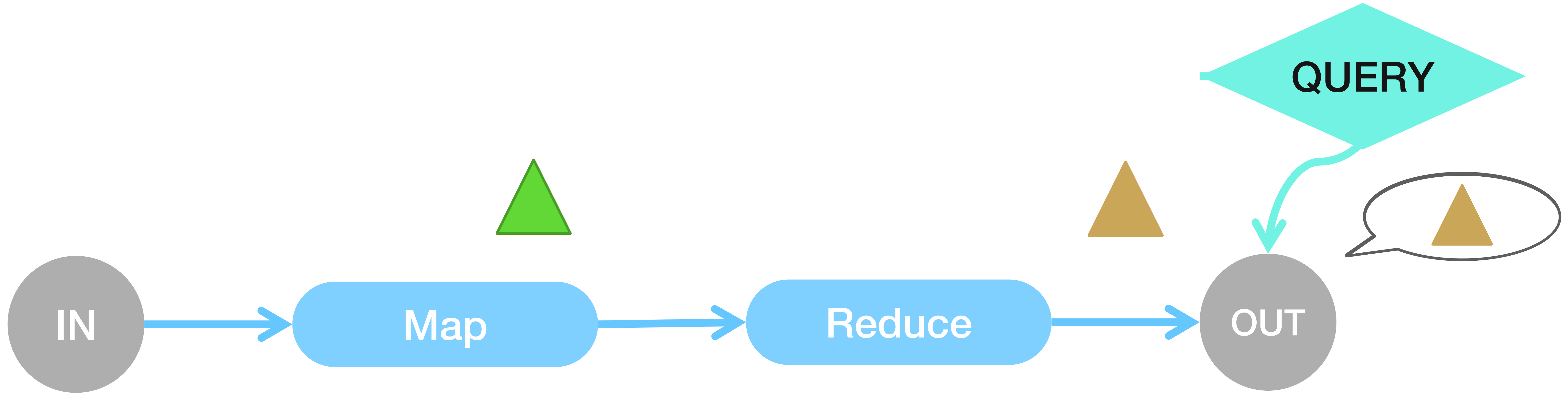
- Input data grouped, e.g. per hour, per N transactions
- Data is processed all together
- Use cases: time-insensitive, or completeness requirements



Brief recap

Stream processors

- Continuous input
- Data processed as soon as input active
- Use case: time-critical, estimations okay



Brief recap

Existing systems

- Batch processors: high latency for queries
- Stream processors like MillWheel have no support for iteration
- Trigger based systems support iteration on data streams by updating shared state (such as key-value tables, c.f. Oolong), but no consistency guarantees
- Enter Naiad's timely dataflow model

What is timely dataflow?

Overview

- Timestamps
- Loop contexts
- Message passing and notifications
- Pointstamps (time-and-location)
- Pointstamp dependencies
- Consistency guarantees

What is timely dataflow?

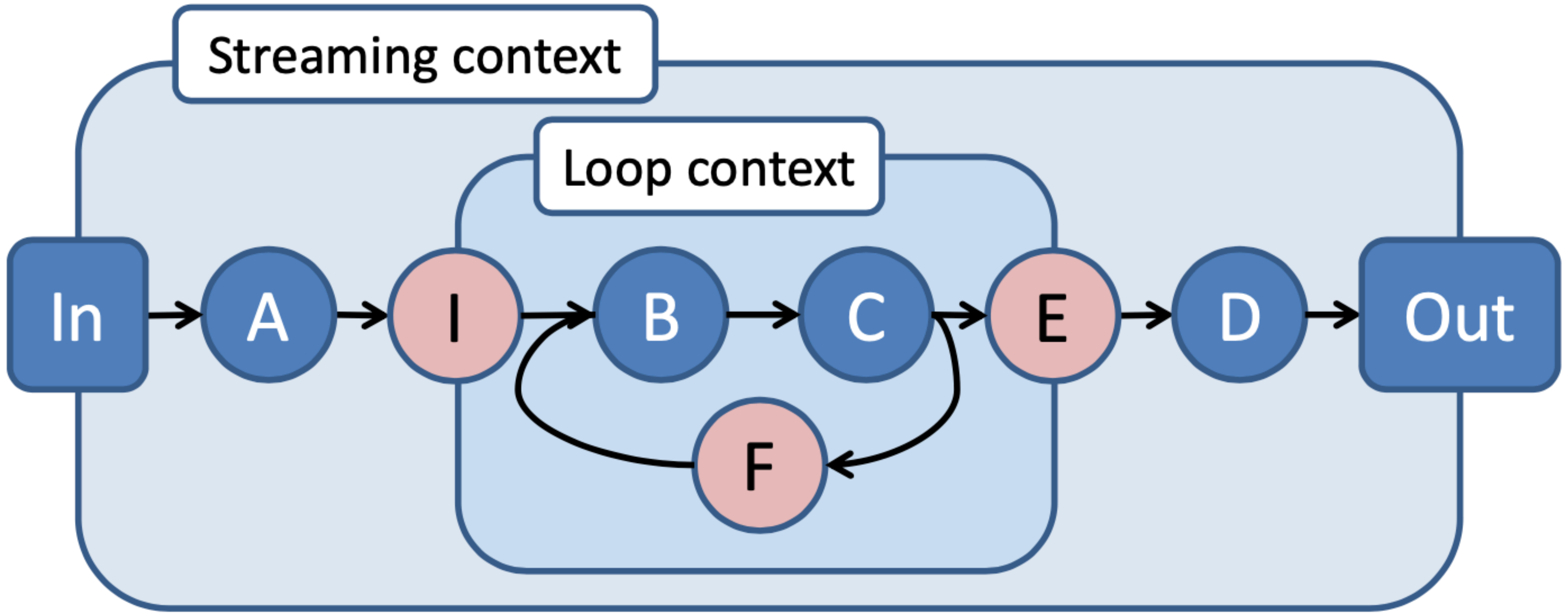
Traditional timestamps

- Timestamps affixed to input data to simulate batch processing
- Not a new idea in itself – used by processors like MillWheel to make certain guarantees about output
- Useful when real time data arrives in system out of order, (e.g. event at 1402hrs not seen until 1408hrs)
- Naiad assigns each input an integer epoch

What is timely dataflow?

Loop contexts

- Naiad supports iteration using loop contexts
- Top-level streaming context contains entire computation, and loop contexts may be nested
- Loop contexts always have:
  - Ingress node
  - Egress node
  - Feedback node



What is timely dataflow?

Naiad timestamps

- Each loop context modifies the timestamp of each message:

$$\text{Timestamp} : (\underbrace{e \in \mathbb{N}}_{\text{epoch}}, \underbrace{\langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k}_{\text{loop counters}})$$

- Feedback nodes increment loop counter c_k
- Ingress nodes increment *dimension* of loop counter, (egress decrement)

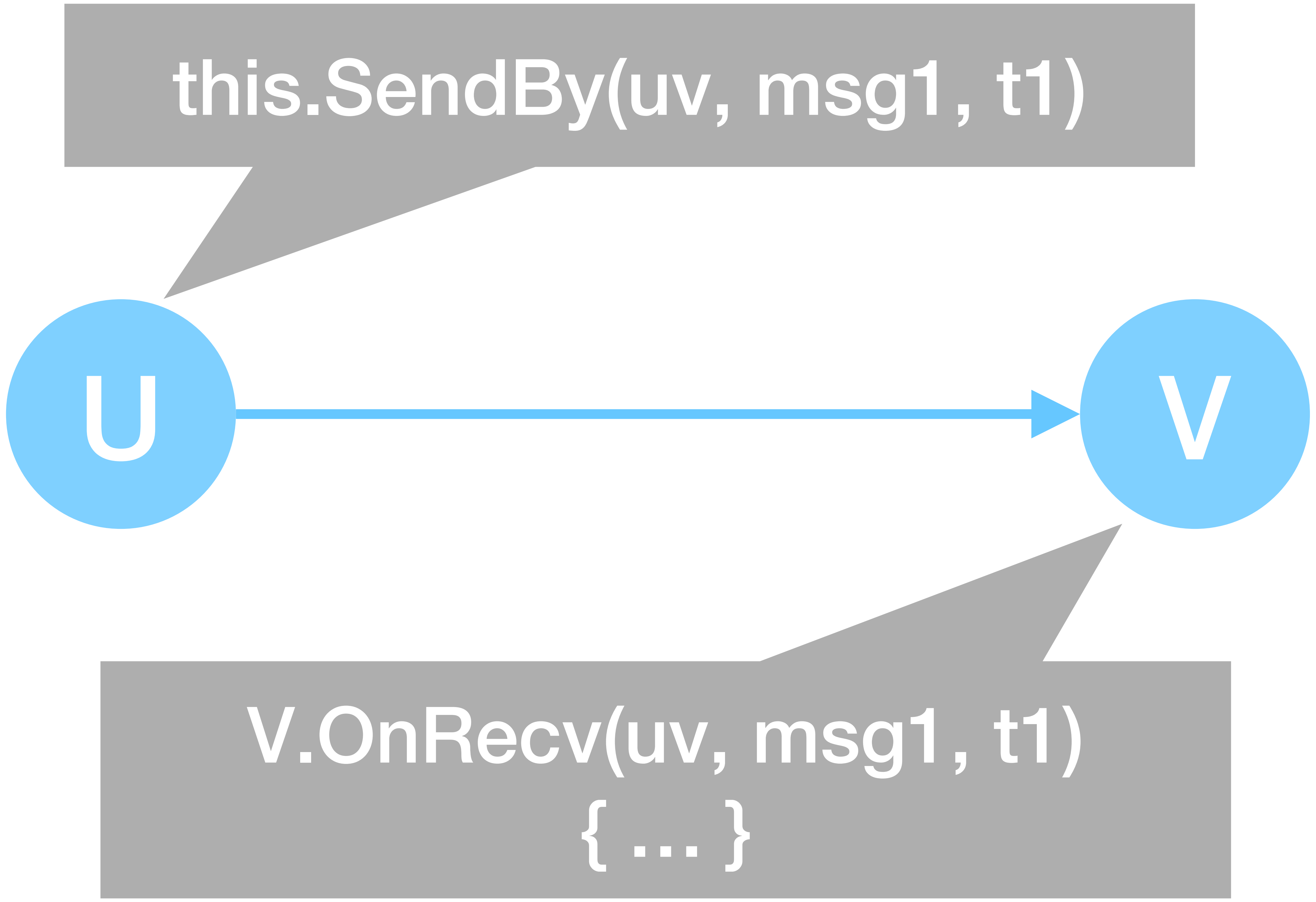
What is timely dataflow?

Message passing

- Naiad sends messages between vertices along edges:
  - `OnRecv(Edge e, Message msg, Timestamp t)`
  - `SendBy(Edge e, Message msg, Timestamp t)`
- Notifications provide a way to be certain no more OnRecv calls with some particular timestamp are incoming:
  - `OnNotify(Timestamp t)`
  - `NotifyAt(Timestamp t)`

What is timely dataflow?

Message passing



What is timely dataflow?

Constraints

- Mustn’t send messages back in time!
- If v.OnRecv(e, msg, t1) is called, can only invoke calls of SendBy and NotifyAt using timestamp t2, with t2 >= t1
- Additionally, system must ensure that if V.OnNotify(t2) is called, this means there can be no further invocations of V.OnRecv(e, msg, t1) for t1 <= t2
- These constraints allow both low latency processing (using SendBy and OnRecv) *as well as* consistency in output where required (using notifications)

What is timely dataflow?

Data ordering

- Input data affixed with integer epoch (‘batch’)
- Loop contexts provide a way to compare data order (timestamps)
- Messages sent/received provide low latency
- Notification delivery provides consistency
- Problem: when can notifications actually be delivered (which timestamp)?

What is timely dataflow?

Pointstamps

- Suppose we have some notification to deliver to a particular vertex. We need to work out when this delivery ‘event’ can occur.
- Possible future timestamps constrained by current set of unprocessed events (messages + notifications), i.e. we cannot deliver a notification if it has any earlier dependencies.
- Each event has a timestamp *and a location*, either an edge or vertex. Naiad expresses these as pointstamps

$$\text{Pointstamp} : (t \in \text{Timestamp}, \overbrace{l \in \text{Edge} \cup \text{Vertex}}^{\text{location}})$$

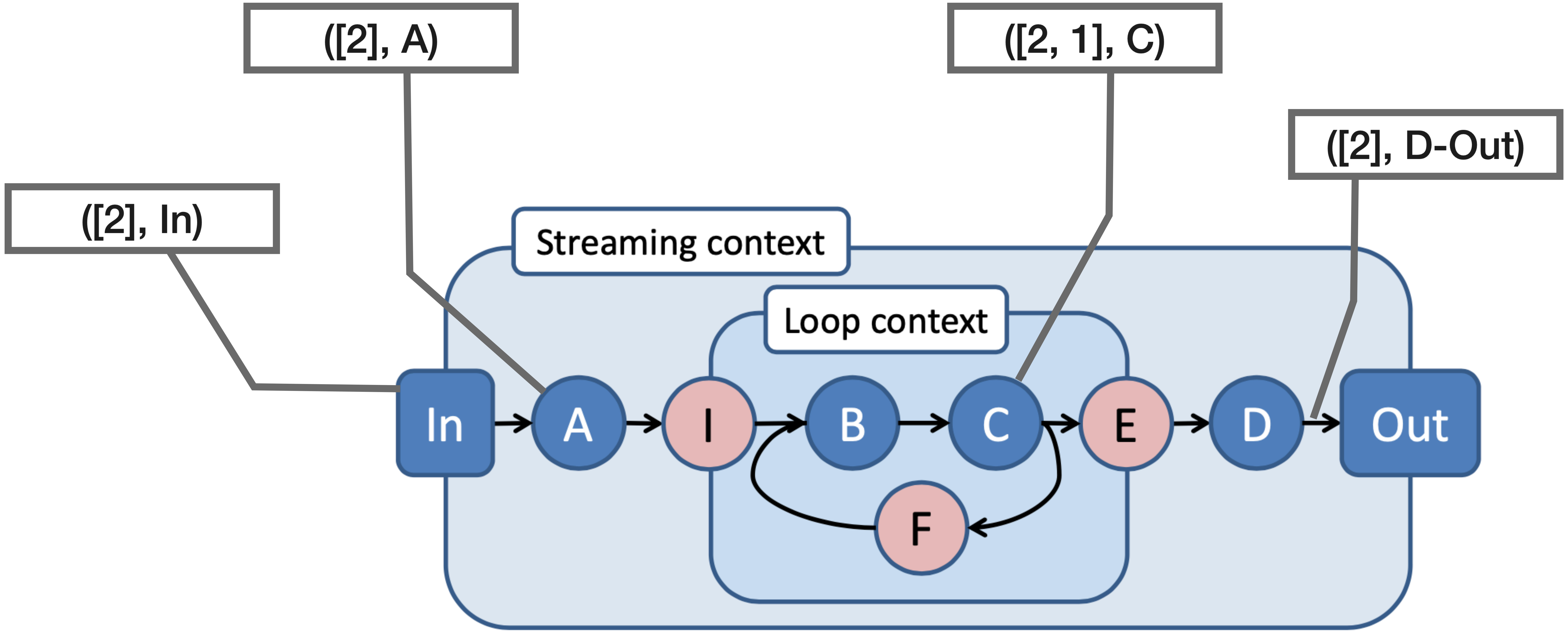
What is timely dataflow?

Pointstamps

- The pointstamp (t1, l1) could-result-in (t2, l2) if there is a path between l1 and l2, through the various ingress, egress and feedback nodes, such that t1 <= t2
- We call a pointstamp *active* if it corresponds to at least one unprocessed event
- A pointstamp p has a certain number of *precursor* pointstamps, i.e. the other pointstamps which could-result-in p
- Once p has no more precursors, then the scheduler may deliver any notification with pointstamp p, as can be sure no later event is on the way

What is timely dataflow?

Pointstamps



Could p1 = ([2, 4], C) result in p2 = ([2, 4], B) ?

What about the other way around?

What is timely dataflow?

Occurrence count

- How many events are associated with a pointstamp p?
- Scheduler keeps a list of Occurrence Counts, updated according to:

Operation	Update
$v.\textsc{SendBy}(e,m,t)$	$\mathrm{OC}[(t,e)] \leftarrow \mathrm{OC}[(t,e)] + 1$
$v.\textsc{OnRecv}(e,m,t)$	$\mathrm{OC}[(t,e)] \leftarrow \mathrm{OC}[(t,e)] - 1$
$v.\textsc{NotifyAt}(t)$	$\mathrm{OC}[(t,v)] \leftarrow \mathrm{OC}[(t,v)] + 1$
$v.\textsc{OnNotify}(t)$	$\mathrm{OC}[(t,v)] \leftarrow \mathrm{OC}[(t,v)] - 1$

- Scheduler keeps handling events for p until OC drops to 0

What is timely dataflow?

Initialisation

- Need to define how to initialise the computation.
- We start off by assigning a pointstamp for epoch 1 at each input vertex, (with OC=1)
- When the epoch is finished at some vertex, this is signalled by assigning a new pointstamp for epoch 2 as before. The old pointstamp is then removed.

What is timely dataflow?

Putting it all together

- When an input vertex marks the end of an epoch, any pointstamps that had the input pointstamp as their precursor have their precursor count decremented

- Any pointstamp with a precursor count of zero then becomes part of the ‘frontier’ set of pointstamps

- The scheduler can deliver notifications for frontier pointstamps freely, until their occurrence count drops to zero

- When OC[p] = 0, p is no longer active: any pointstamps with p as precursor have their precursor count decremented.

What is timely dataflow?

Summary

- Events tracked through dataflow using pointstamps
- Message sent/received along edges for low latency
- Notifications delivered at vertices to indicate when to perform computation
- Occurrence and precursor counts indicate dependencies between pointstamps, and calculated by considering time taken to traverse minimum paths between locations
- Removing input pointstamps for epochs allows computation to drain out the system

NAIAD Framework

(And more!!)

- Graph processing
- Differential dataflow – graphs can propagate changes quickly, e.g. for connected components, without recomputing the entire algorithm
- Distributed workflow using worker-specific schedulers
- Local occurrence and precursor counts are kept by each worker, and updates to these are broadcasted as necessary (i.e. incremented/decremented according to rules already discussed, with optimisations)
- Checkpoint and Restore methods for each vertex provides fault tolerance

Conclusion and discussion

My thoughts

- Difficult to see precisely where Naiad fits: by authors' own admission, possible to build it by combining more than one other system
- Development on .NET API ceased shortly after release
- There is ongoing development in Rust however

References

- Murray, D., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM.
- Mitchell, C., Power, R., and Li, J. 2012. Oolong: Asynchronous distributed applications made easy. *Proceedings of the Asia-Pacific Workshop on Systems, APSYS'12*.
- Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In *Very Large Data Bases* (pp. 734–746).
- If interested, see also Derek Murray presenting Naiad at SOSP https://www.youtube.com/watch?v=yyhMI9r0A9E&t=273s
- And Frank McSherry's short explanation of timely dataflow https://www.youtube.com/watch?v=yOnPmVf4YWo
- And also McSherry's demonstration of differential dataflow https://channel9.msdn.com/posts/Frank-McSherry-Introduction-to-Naiad-and-Differential-Dataflow
- The original naiad implementation is found at https://github.com/MicrosoftResearch/Naiad , as mentioned is no longer maintained, but McSherry was still working more recently on an implementation in Rust https://github.com/frankmcsherry/timely-dataflow