@FOSDEM 2017

Not Less, Not More

Exactly Once, Large-Scale Stream Processing in Action

Paris Carbone

Committer @ Apache Flink

PhD Candidate @ KTH

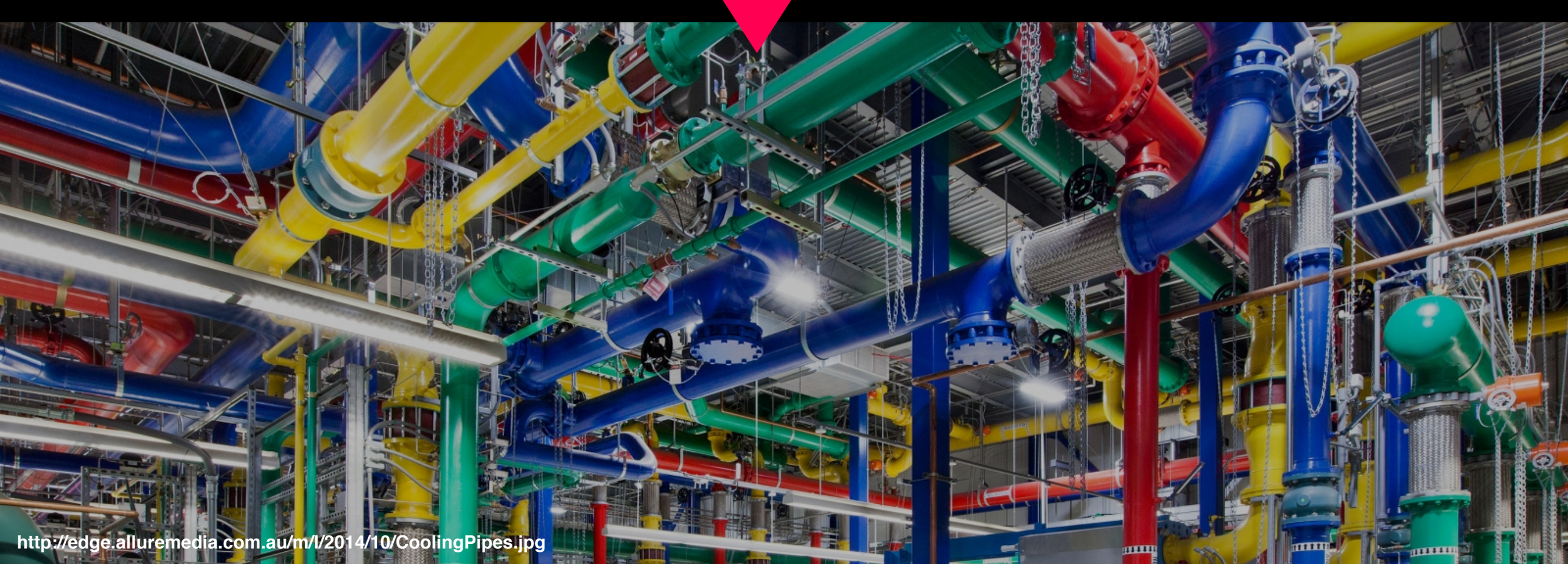
@SenorCarbone

Data Stream Processors

```
val windowCounts = text.flatMap { w => w.split("\\s") }
  .map { w => WordWithCount(w, 1) }
  .keyBy("word")
  .timeWindow(Time.seconds(5))
  .sum("count")
```

can set up any data pipeline for you

Data Stream Processor



http://edge.alluremedia.com.au/m/l/2014/10/CoolingPipes.jpg

With Data Stream Processors

With Data Stream Processors

- **sub-second latency** and **high throughput** finally **coexist**

With Data Stream Processors

- **sub-second latency** and **high throughput** finally **coexist**
- **late data*** is handled gracefully

* http://dl.acm.org/citation.cfm?id=2824076

With Data Stream Processors

- **sub-second latency** and **high throughput** finally **coexist**
- **late data*** is handled gracefully
- and btw data stream pipelines run 365/24/7 consistently without any issues in general.

* **http://dl.acm.org/citation.cfm?id=2824076**

With Data Stream Processors

- **sub-second latency** and **high throughput** finally **coexist**
- **late data*** is handled gracefully
- and btw data stream pipelines run 365/24/7 consistently without any issues in general.
- wait…**what?**

* http://dl.acm.org/citation.cfm?id=2824076

So…what about

So…what about

application updates

handling failures!

adding more/less workers

reconfiguring/upgrading the system

So…what about

application updates

handling failures!

adding more/less workers

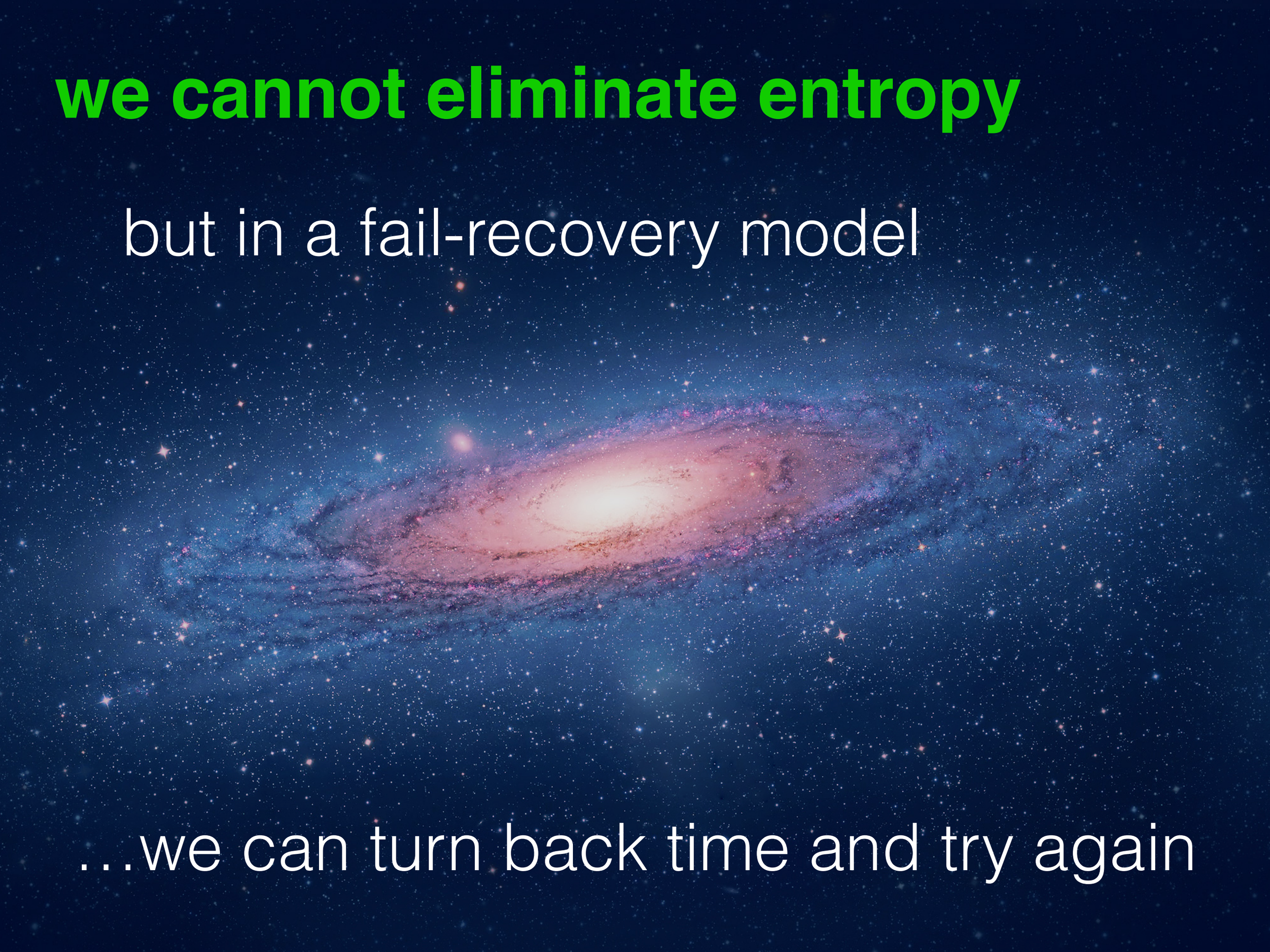
reconfiguring/upgrading the system

is it realistic to expect the thing run forever correctly?

we cannot eliminate entropy

we cannot eliminate entropy

but in a fail-recovery model



…we can turn back time and try again

Let’s talk about Guarantees

Let’s talk about Guarantees

guaranteed tuple processing

at-least once

exactly once!

processing

end-to-end

output

delivery

transactional processing

at-most once

idempotent writes

deterministic processing

high availability

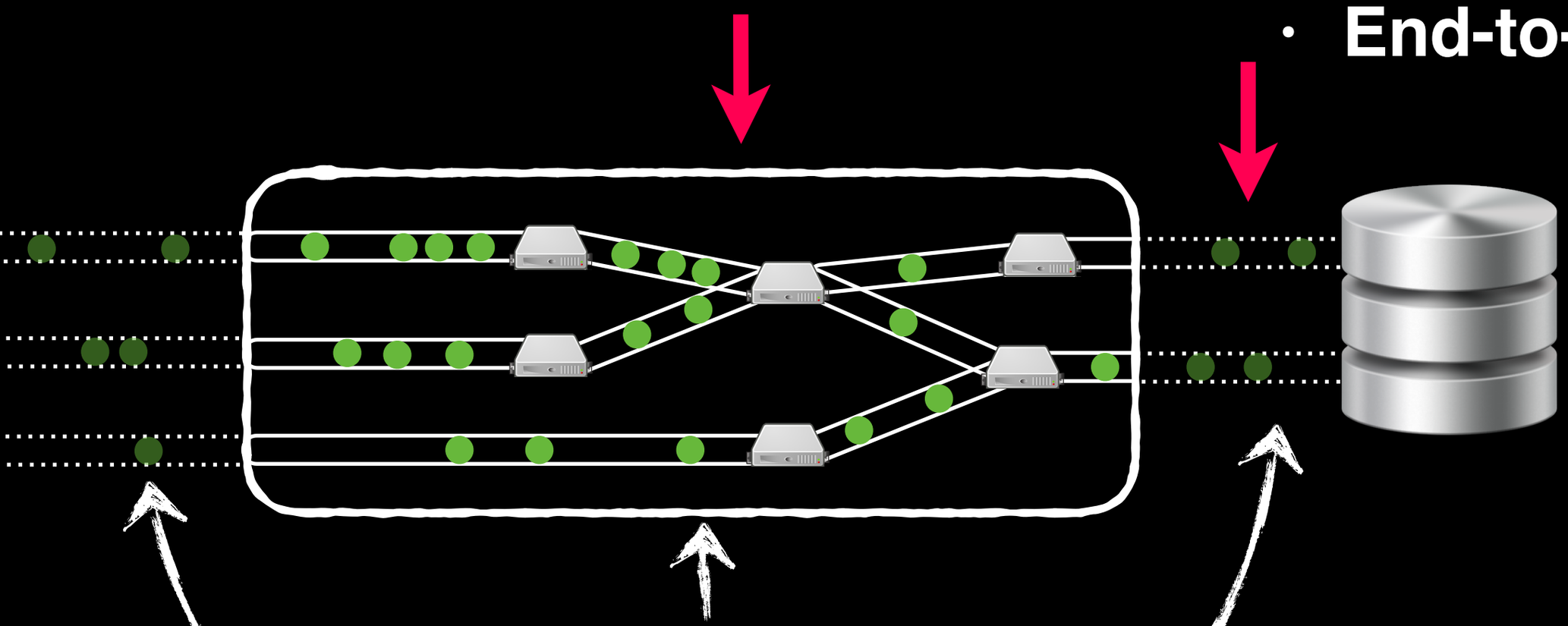
resilient state

What Guarantees

What Guarantees

1) Processing

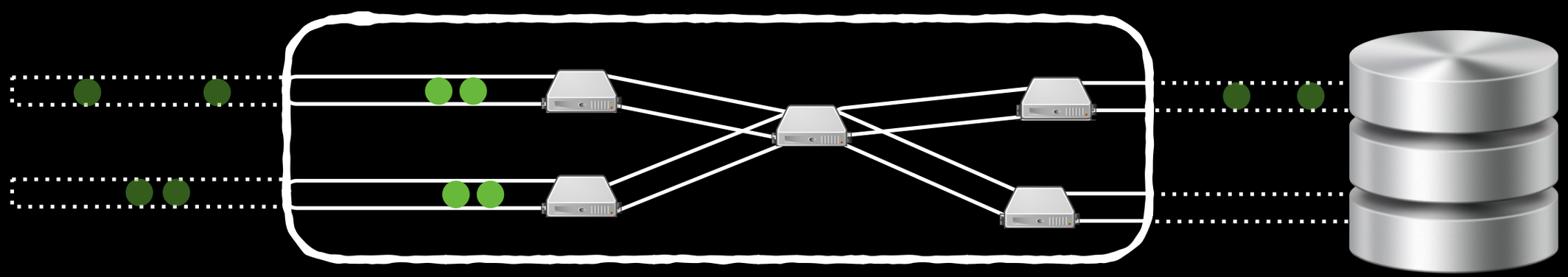
2)
- Output
- Delivery
- End-to-End



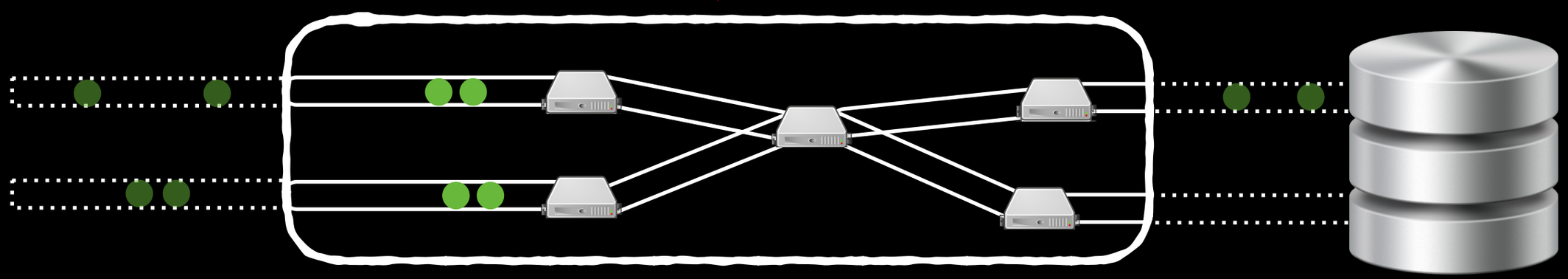
system

outside world

Processing Guarantees

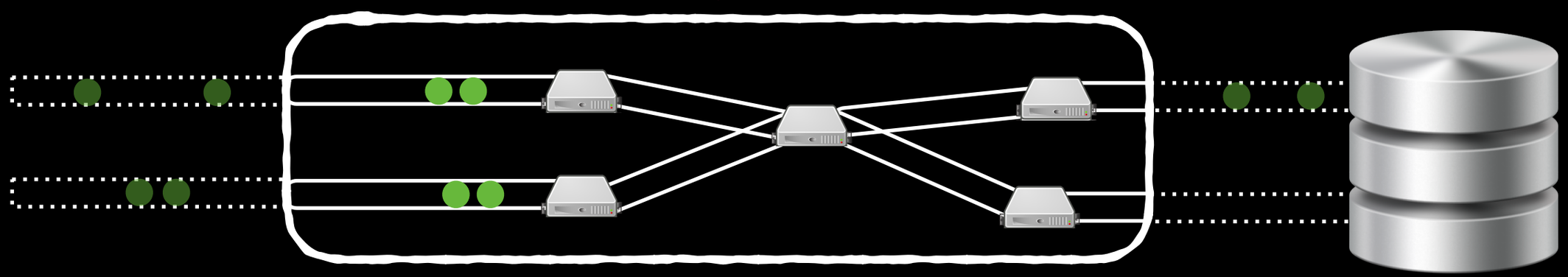


Processing Guarantees

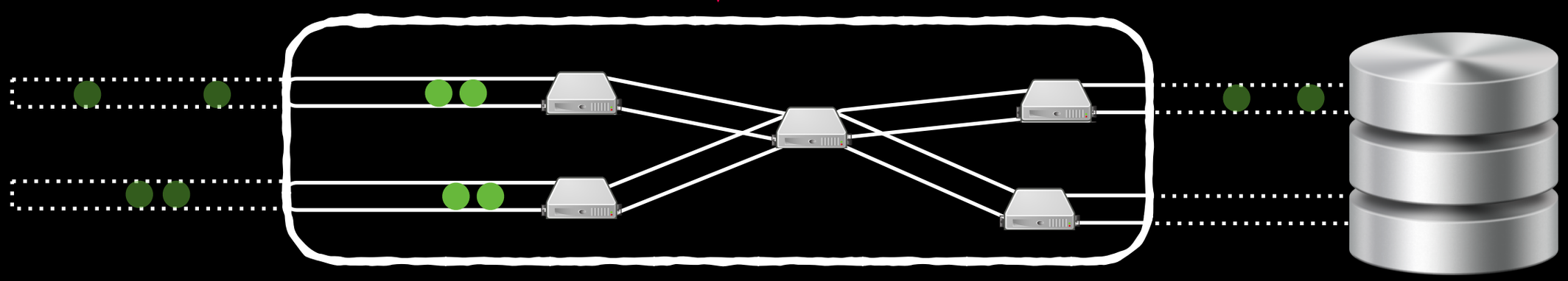


- Why should we care?
  - Processing creates side effects inside the system's internal state.
  - Less or more processing sometimes means incorrect internal state.

Processing Guarantees

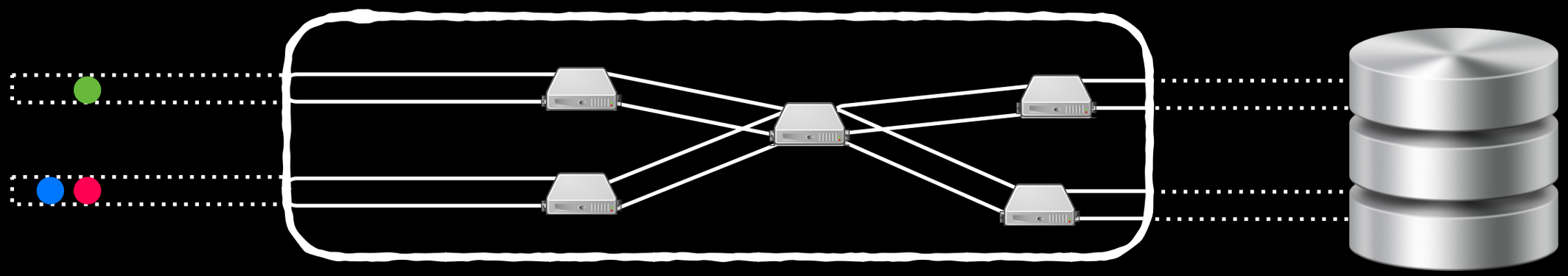


Processing Guarantees



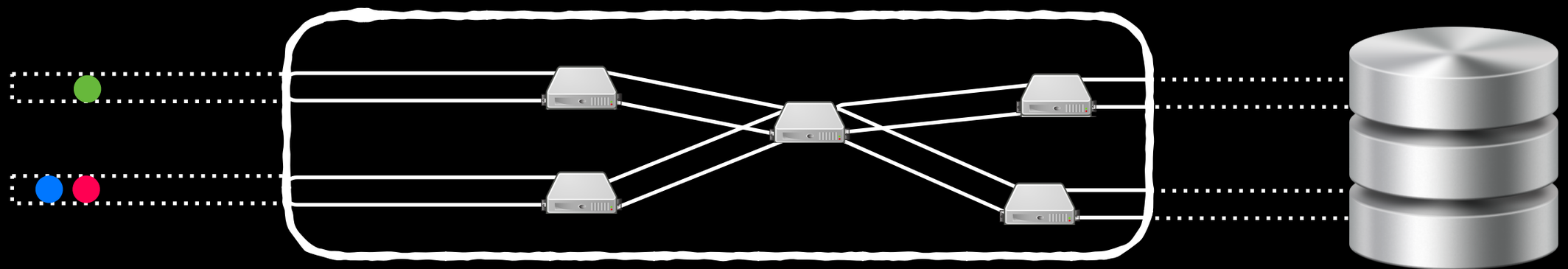
- **At-Most Once:** the system might process **less** (e.g., ignoring data if overloaded)
- **At-Least Once:** the system might process **more** (e.g., replaying input)
- **Exactly Once:** the system behaves as if input data are processed exactly once

At-Least Once Processing

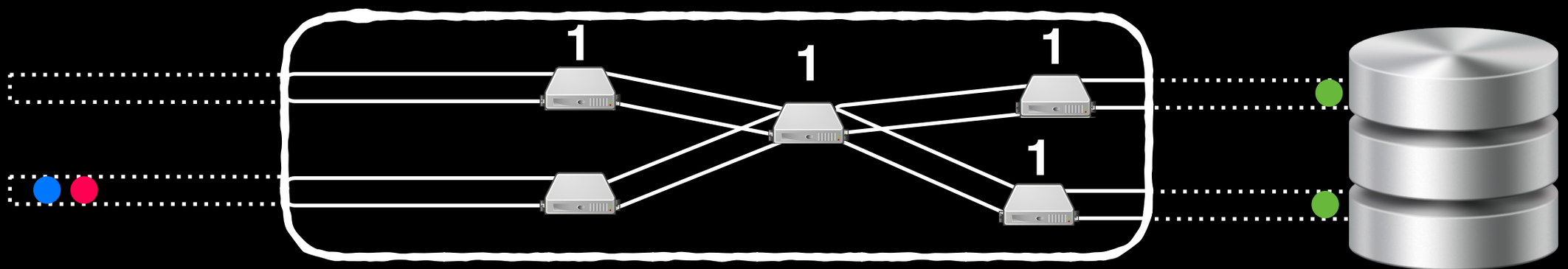


- Useful when repetition can be tolerated.
- Already offered by logs (e.g., Kafka, Kinesis)
- Manual Logging & Bookkeeping (Storm < v.2)

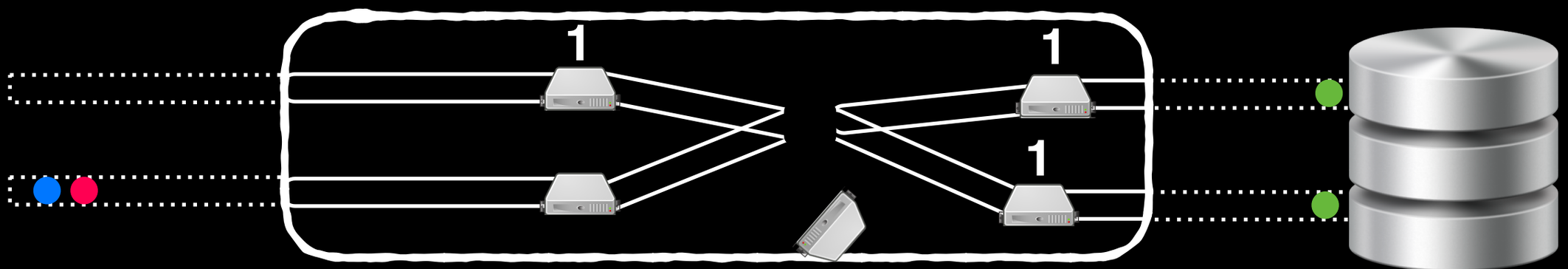
At-Least Once Processing



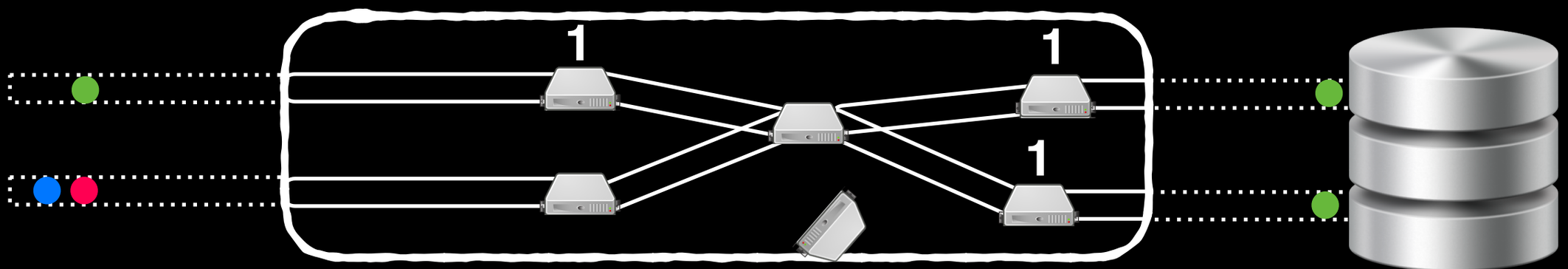
At-Least Once Processing



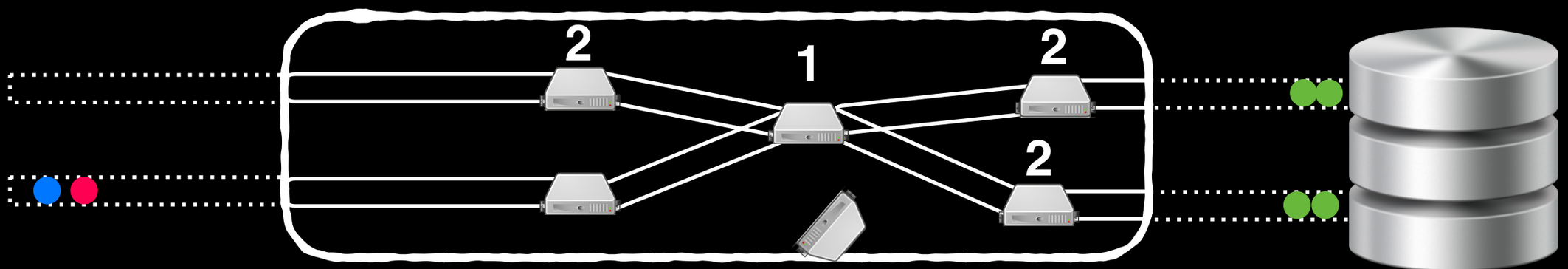
At-Least Once Processing



At-Least Once Processing



At-Least Once Processing



Exactly Once Processing

Exactly Once Processing

It is a bit trickier. We need to make sure that

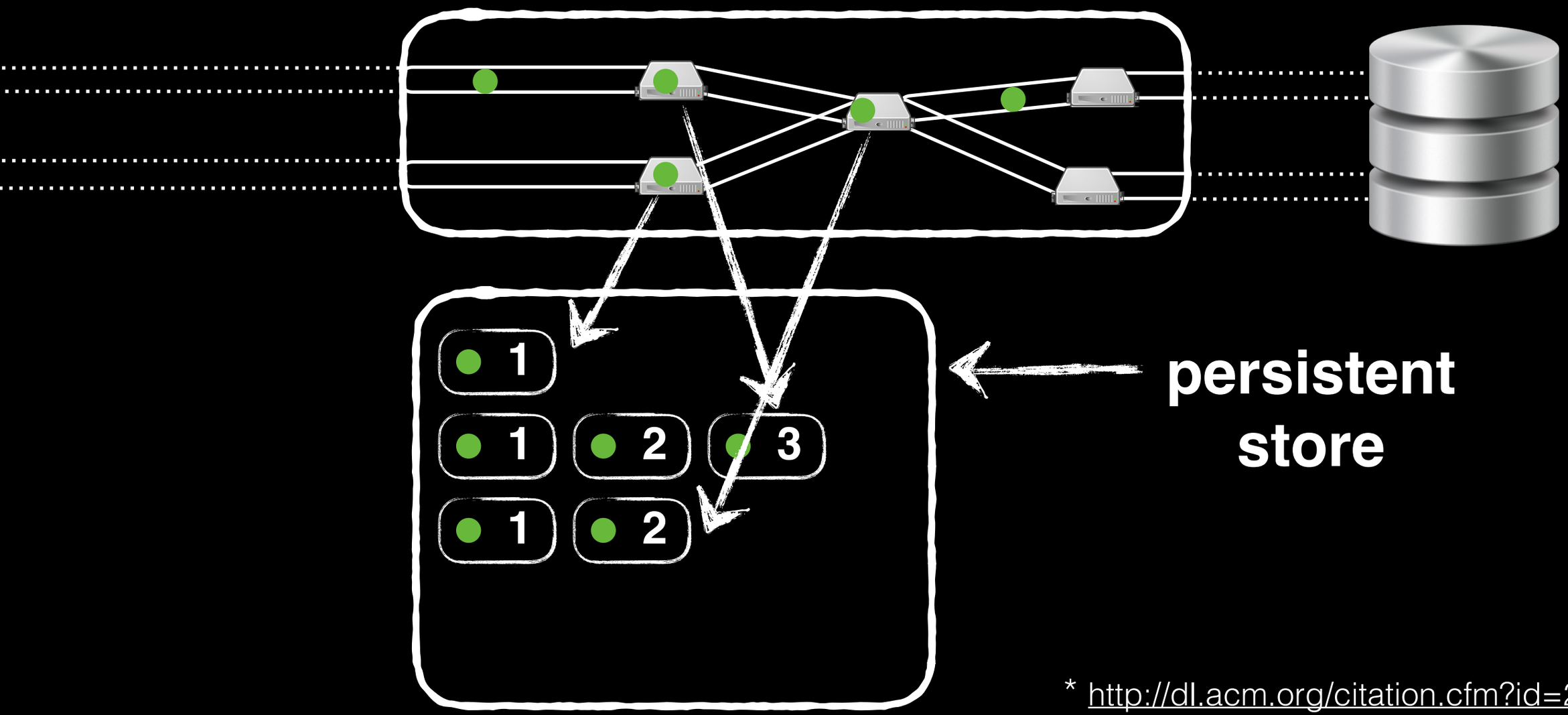
1. Data leaves sides effects **only once**

2. Failure Recovery/Re-Scaling do **not impact** the **correct execution** of the system

Exactly Once Processing

A fine-grained solution…

*Maintain a log for each operation**



* http://dl.acm.org/citation.cfm?id=2536229

Exactly Once Processing

A fine-grained solution…

*Maintain a log for each operation**

Exactly Once Processing

A fine-grained solution…

*Maintain a log for each operation**

- It allows for fine grained failure recovery and trivial reconfiguration.
- Can be optimised to batch writes

However:

- It requires a finely-tuned **performant store**
- Can cause aggressive write/append **congestion**

Remember cassettes?

Remember cassettes?

Remember cassettes?

Remember cassettes?

durable logs similarly allow you to **rollback input**

Remember cassettes?

durable logs similarly allow you to **rollback input**

Remember cassettes?

durable logs similarly allow you to **rollback input**
in parallel

Remember cassettes?

durable logs similarly allow you to **rollback input**
in parallel
from **specific offsets**

Remember cassettes?

durable logs similarly allow you to **rollback input**
in parallel
from **specific offsets**

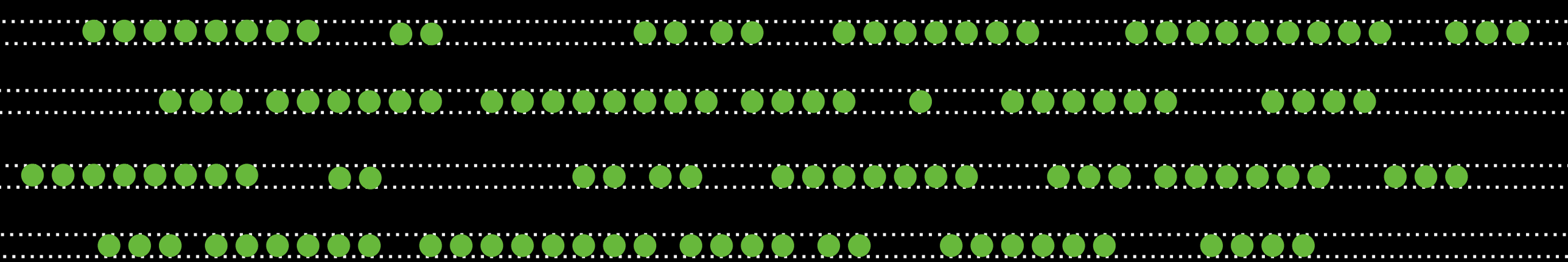
Remember cassettes?

durable logs similarly allow you to **rollback input**
in parallel
from **specific offsets**

Exactly Once Processing

Now a more coarse-grained approach…

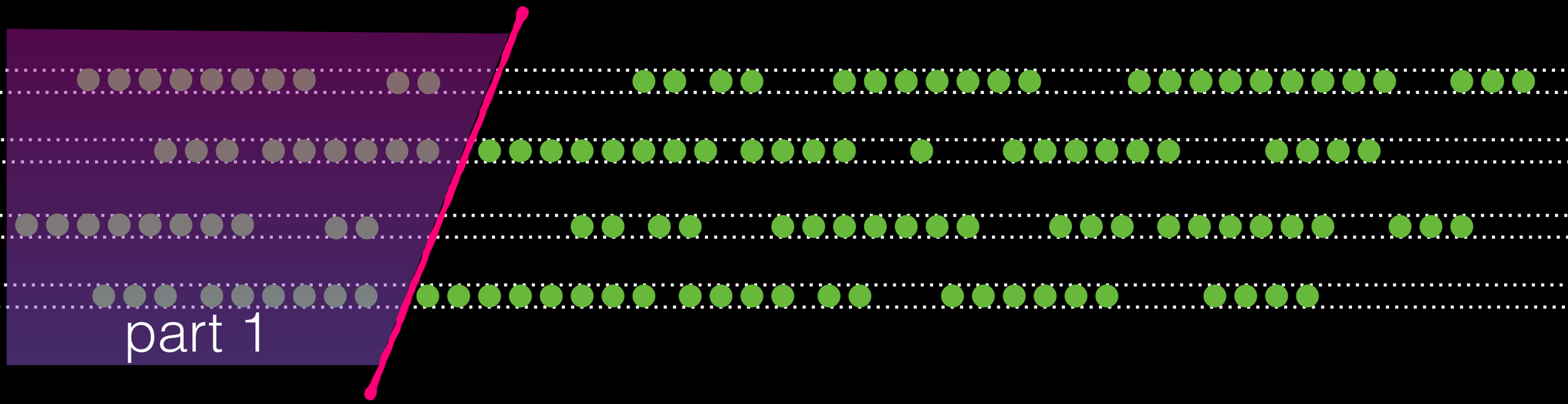
Turn continuous computation into a series of transactions



Exactly Once Processing

Now a more coarse-grained approach…

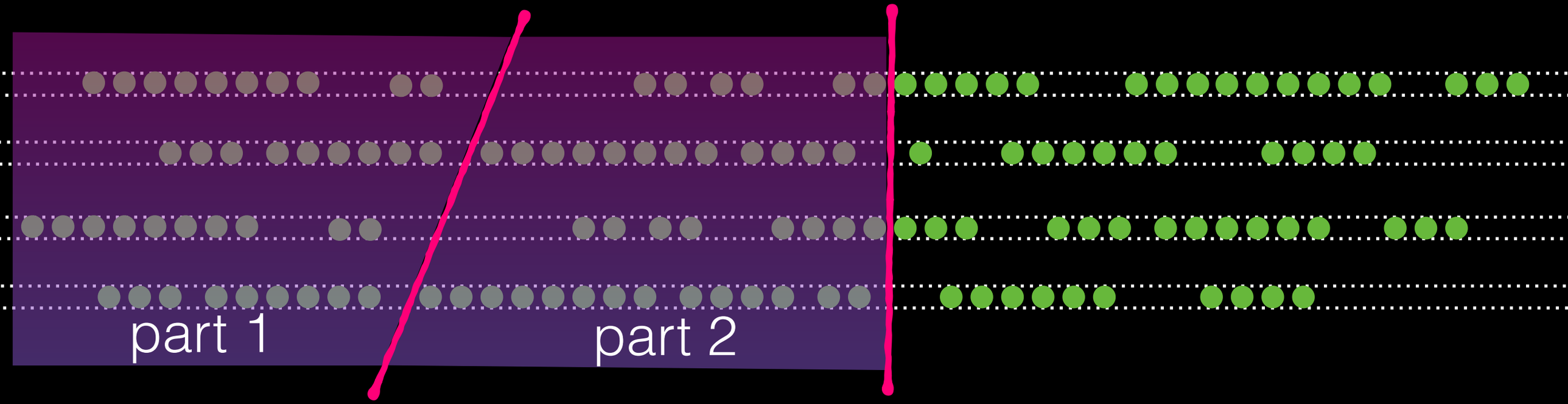
Turn continuous computation into a series of transactions



Exactly Once Processing

Now a more coarse-grained approach…

Turn continuous computation into a series of transactions



Exactly Once Processing

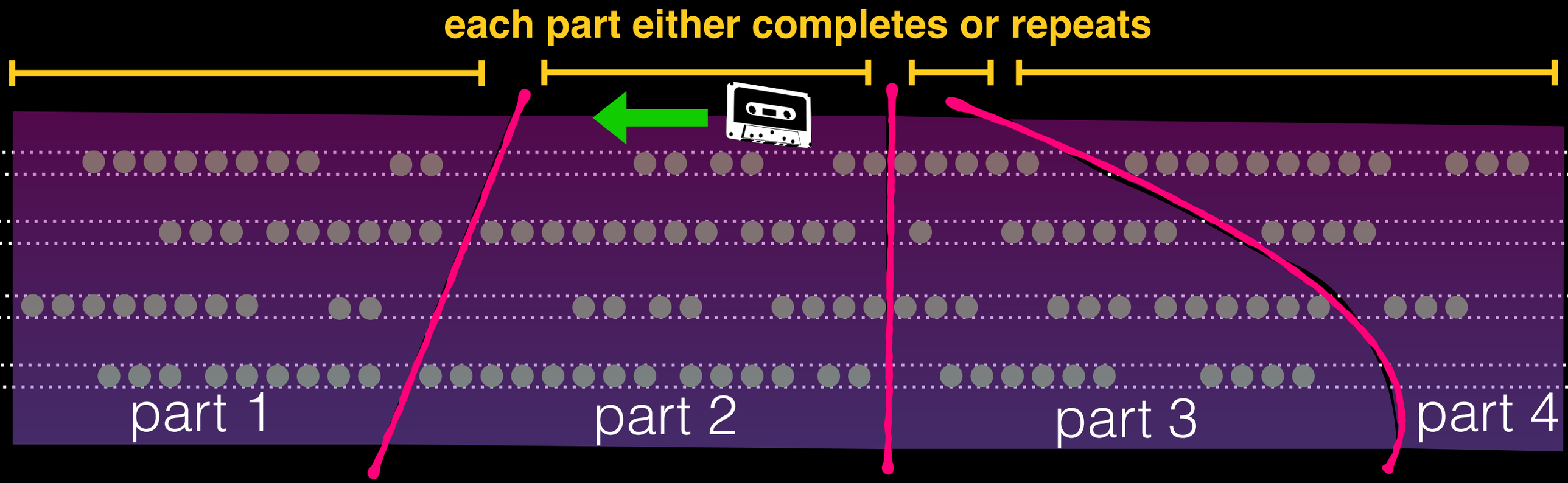
Now a more coarse-grained approach…

Turn continuous computation into a series of transactions

Exactly Once Processing

Now a more coarse-grained approach…

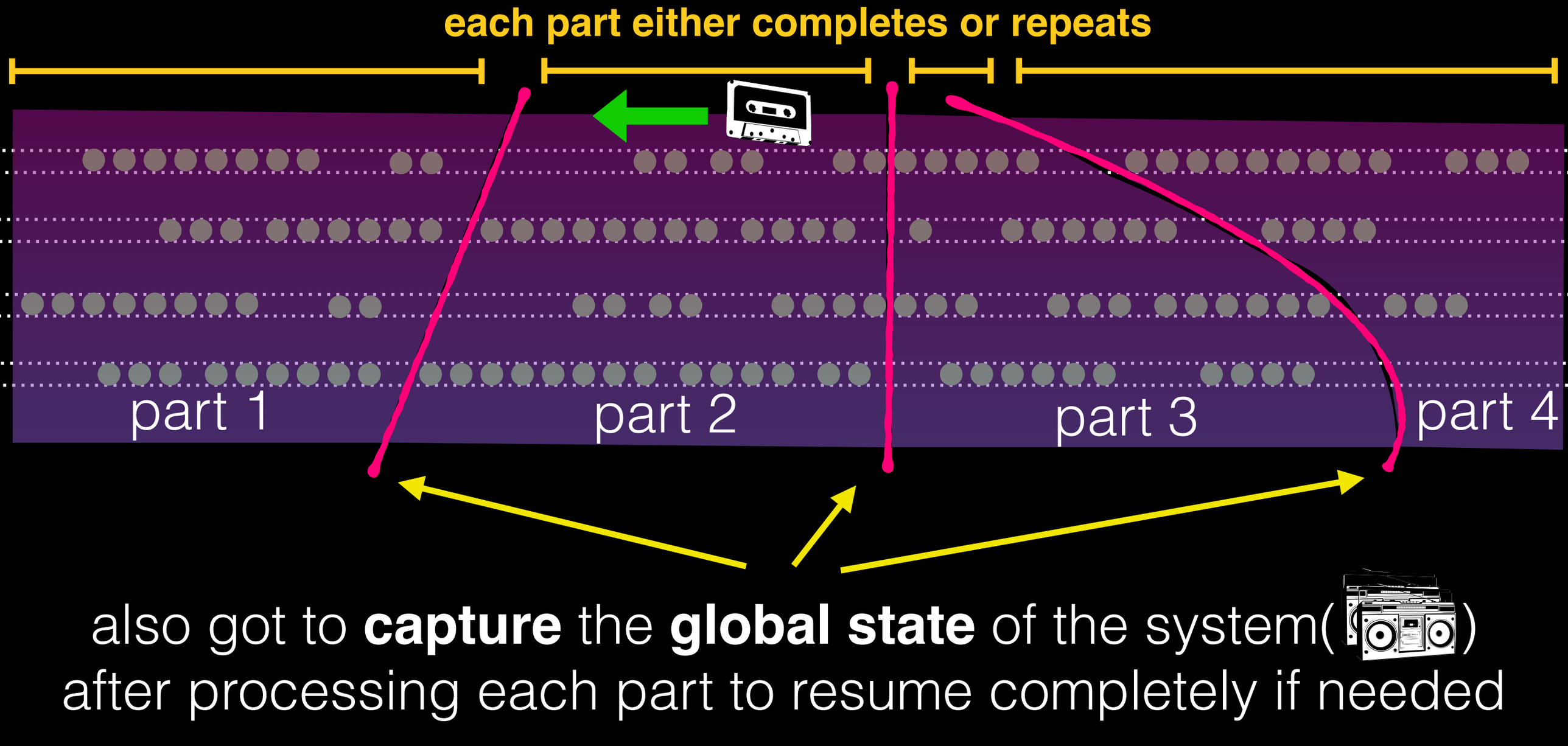
Turn continuous computation into a series of transactions



Exactly Once Processing

Now a more coarse-grained approach…

Turn continuous computation into a series of transactions



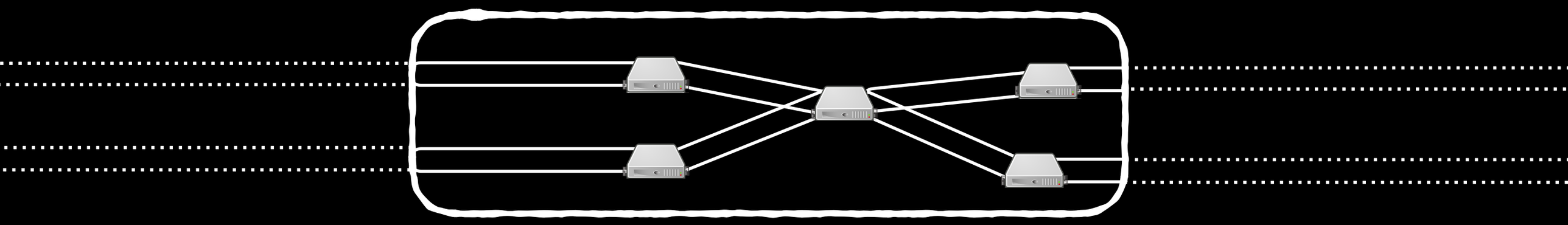
also got to **capture** the **global state** of the system( )
after processing each part to resume completely if needed

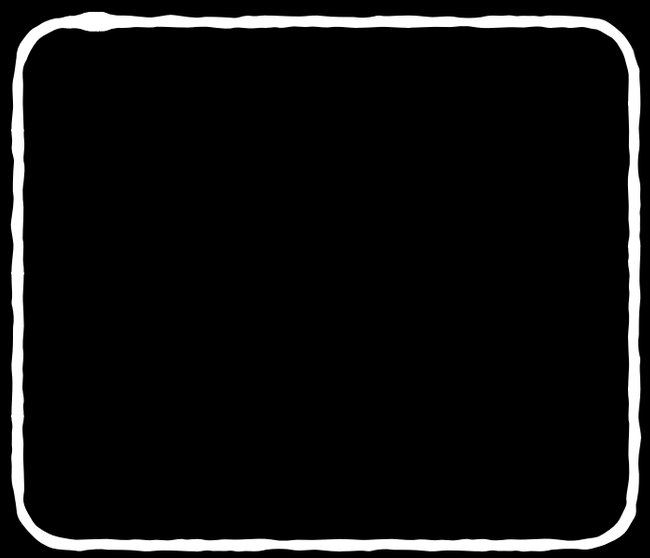
Coarse Grained Fault Tolerance - Illustrated

Type 1

Discrete Execution (micro-batch)

*Snap **after** each part has being processed*





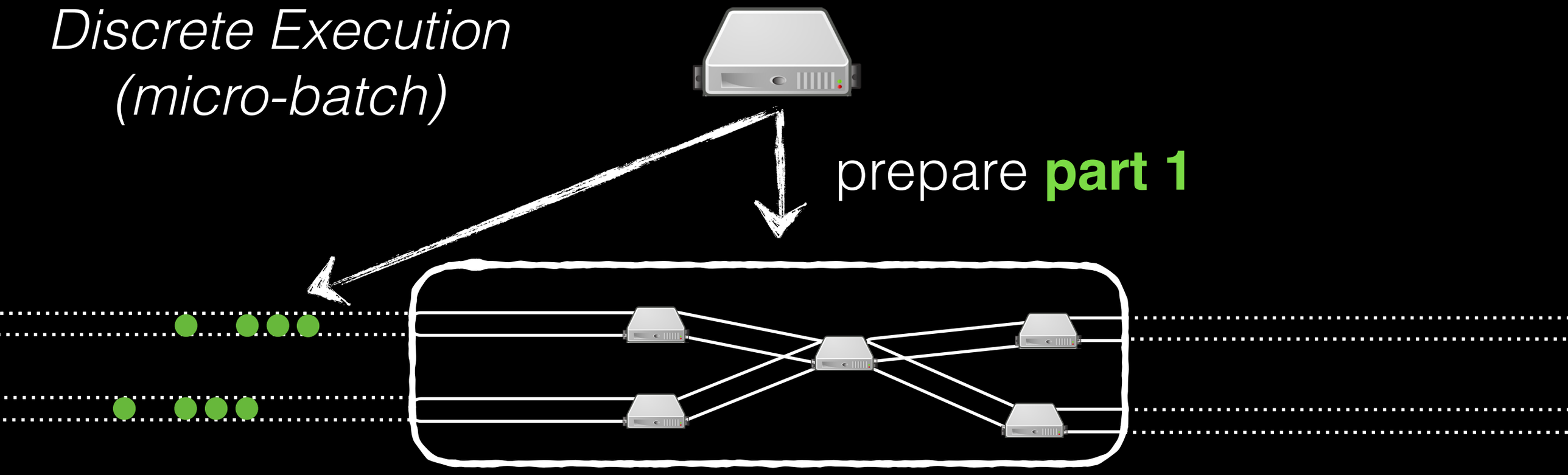
System State Store

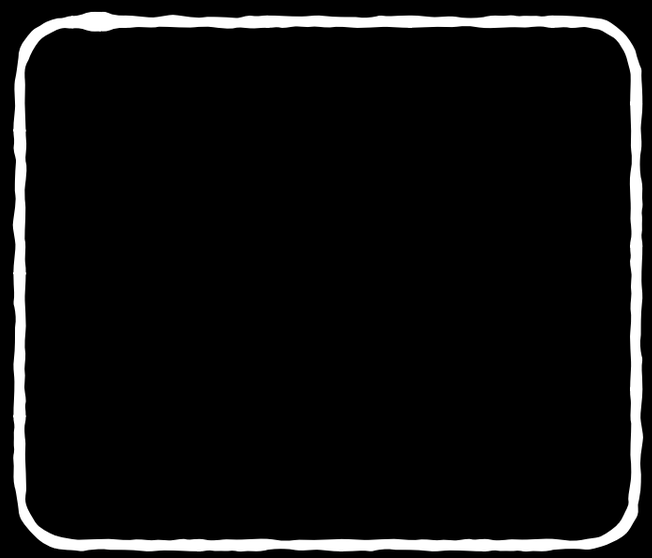
Coarse Grained Fault Tolerance - Illustrated

Type 1

Discrete Execution (micro-batch)

*Snap **after** each part has being processed*





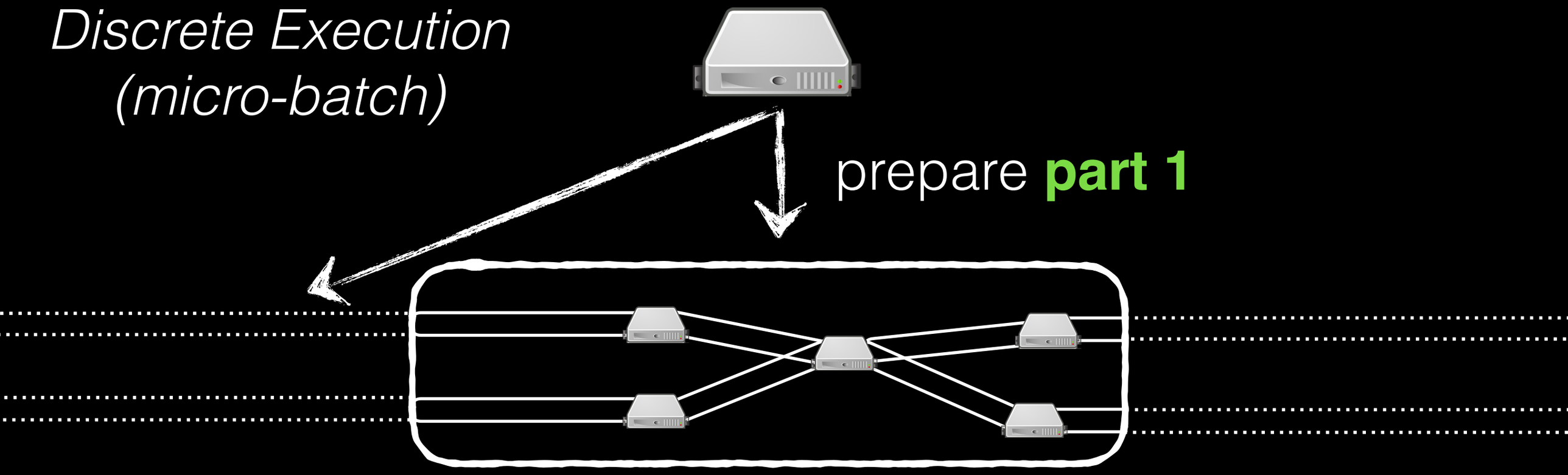
System State Store

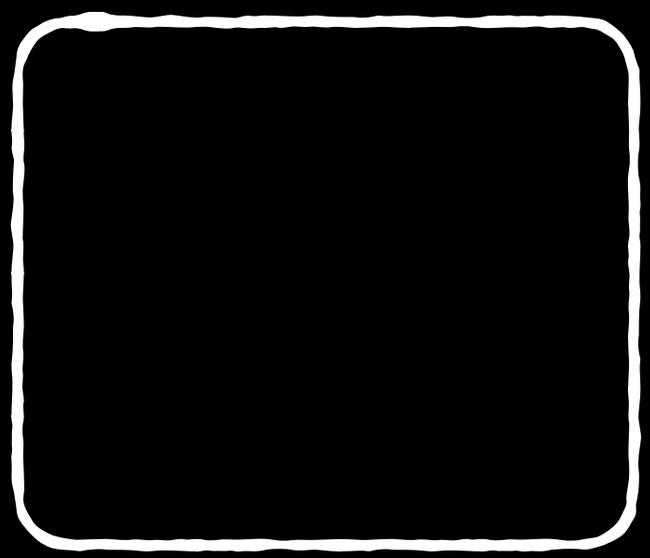
Coarse Grained Fault Tolerance - Illustrated

Type 1

Discrete Execution (micro-batch)

*Snap **after** each part has being processed*





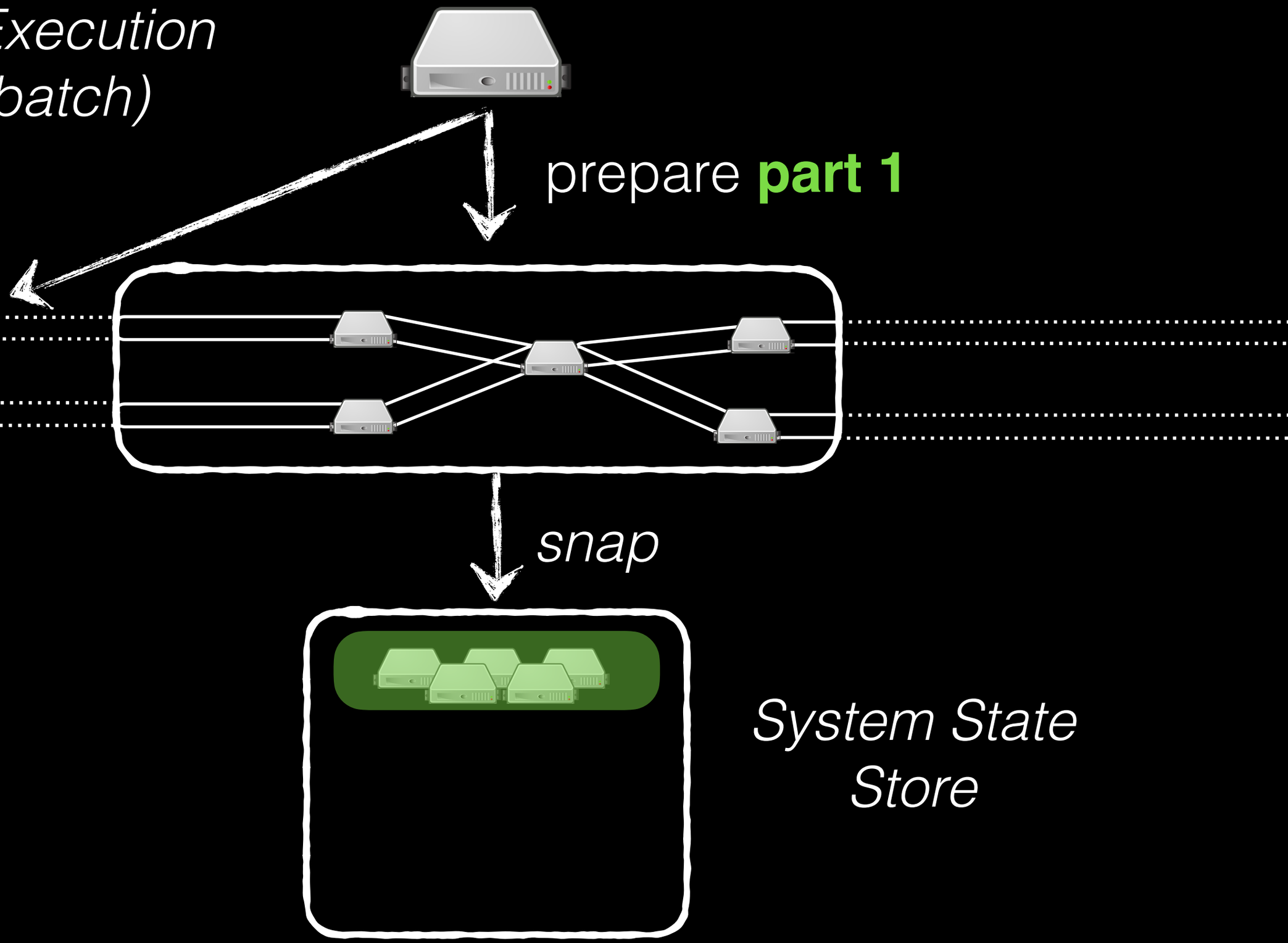
System State Store

Coarse Grained Fault Tolerance - Illustrated

Type 1

Discrete Execution (micro-batch)

*Snap **after** each part has being processed*

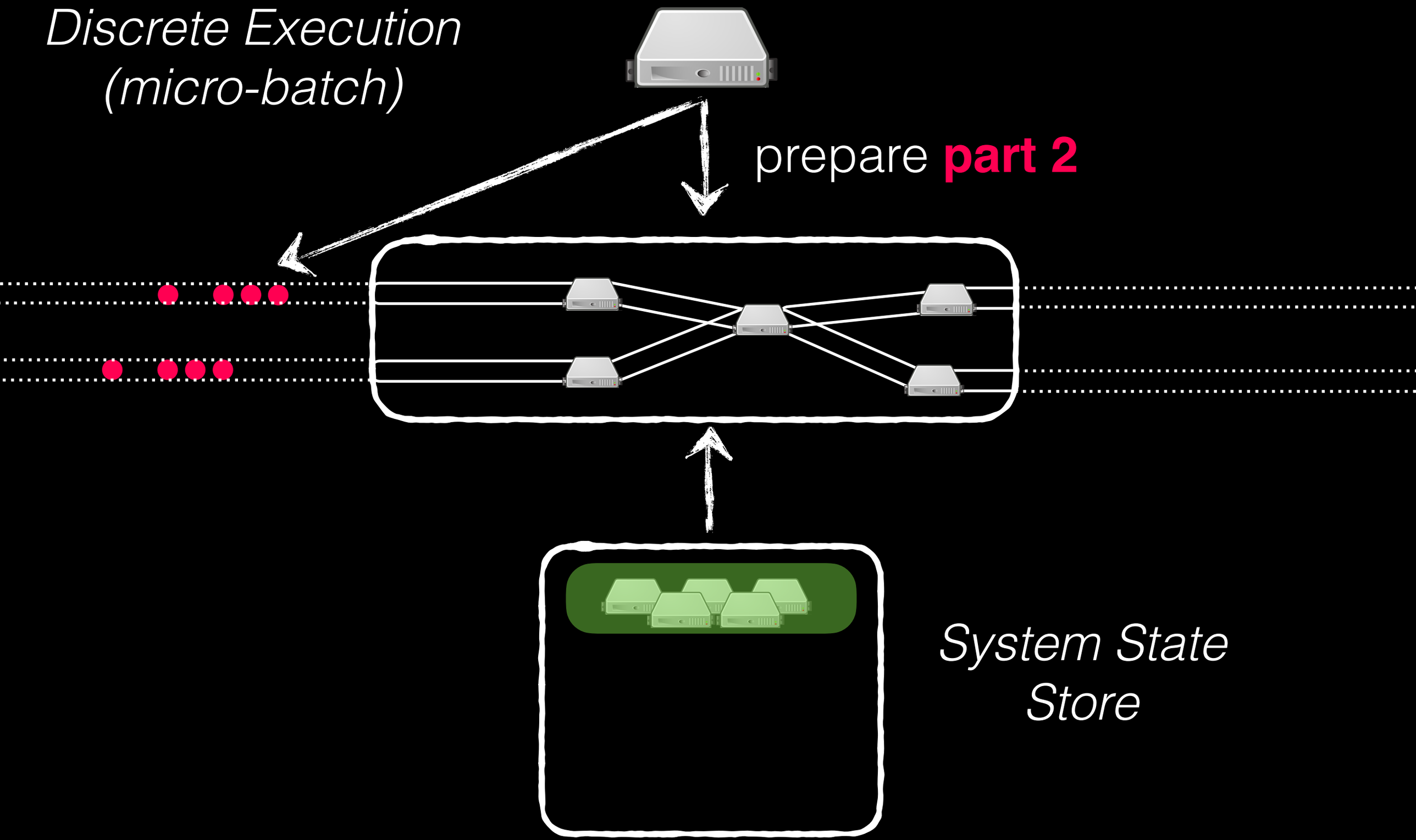


Coarse Grained Fault Tolerance - Illustrated

Type 1

Discrete Execution (micro-batch)

*Snap **after** each part has being processed*

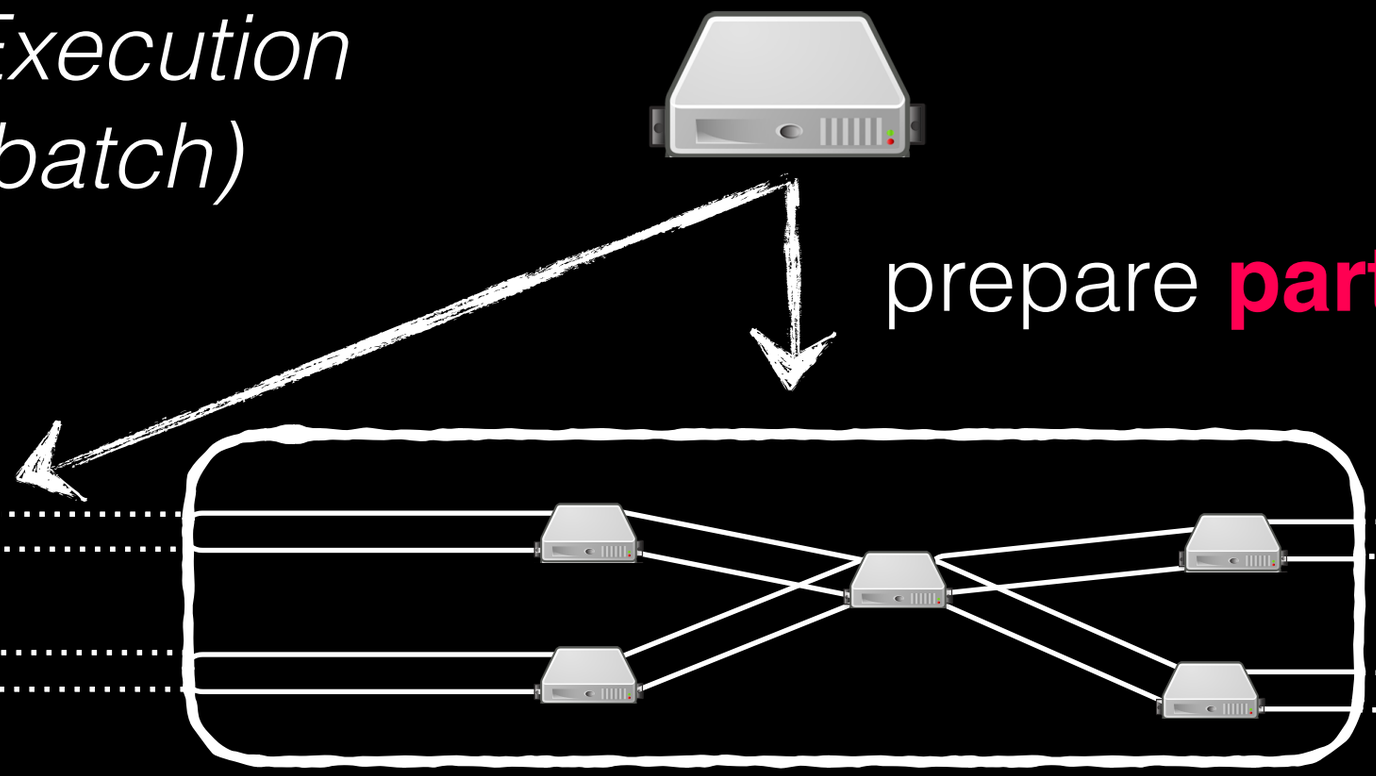


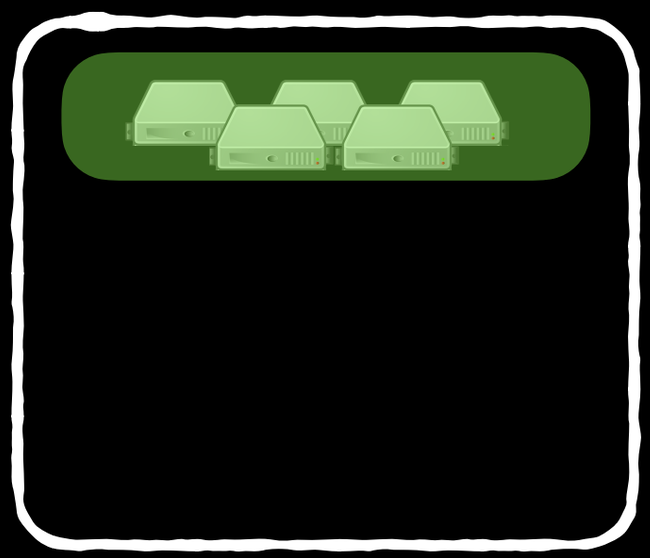
Coarse Grained Fault Tolerance - Illustrated

Type 1

Discrete Execution (micro-batch)

*Snap **after** each part has being processed*





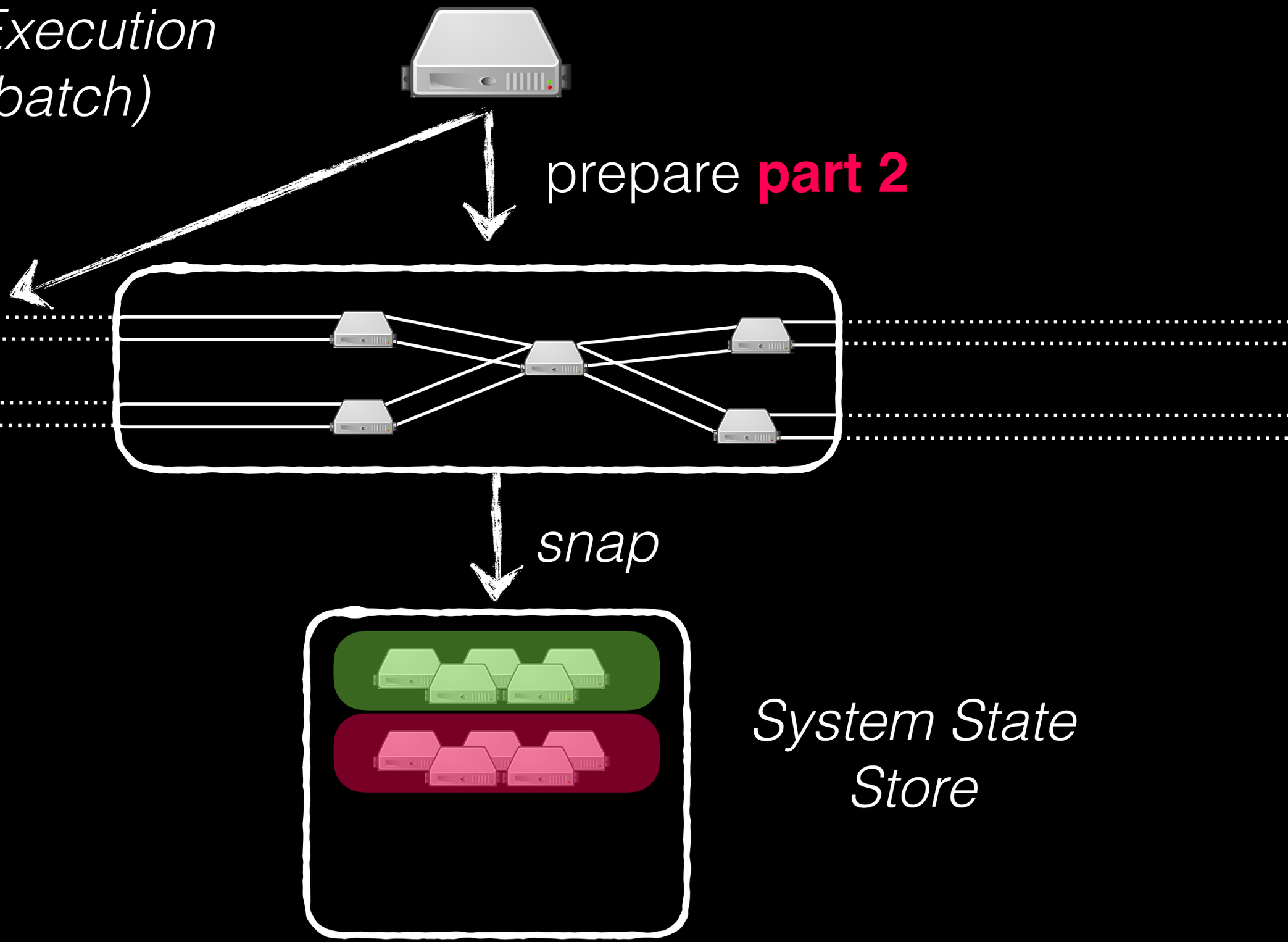
System State Store

Coarse Grained Fault Tolerance - Illustrated

Type 1

Discrete Execution (micro-batch)

*Snap **after** each part has being processed*



Exactly Once Processing

*http://shivaram.org/drafts/drizzle.pdf

Exactly Once Processing

Micro-batching:

- A fine example of discretely emulating continuous processing as a series of transactions.

- However:

  - It enforces a somewhat inconvenient ***think-like-a-batch*** logic for a continuous processing programming model.

  - Causes unnecessarily **high periodic scheduling latency** (can be traded over higher reconfiguration latency by pre-scheduling multiple micro-batches*)

*http://shivaram.org/drafts/drizzle.pdf

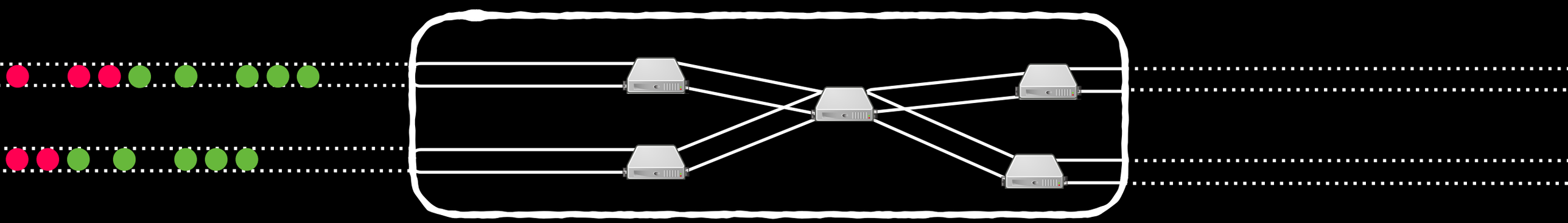
Coarse Grained Fault Tolerance - Illustrated

Type 2

Long-Running

Synchronous

*Snap **while** each part is being processed*



System State Store

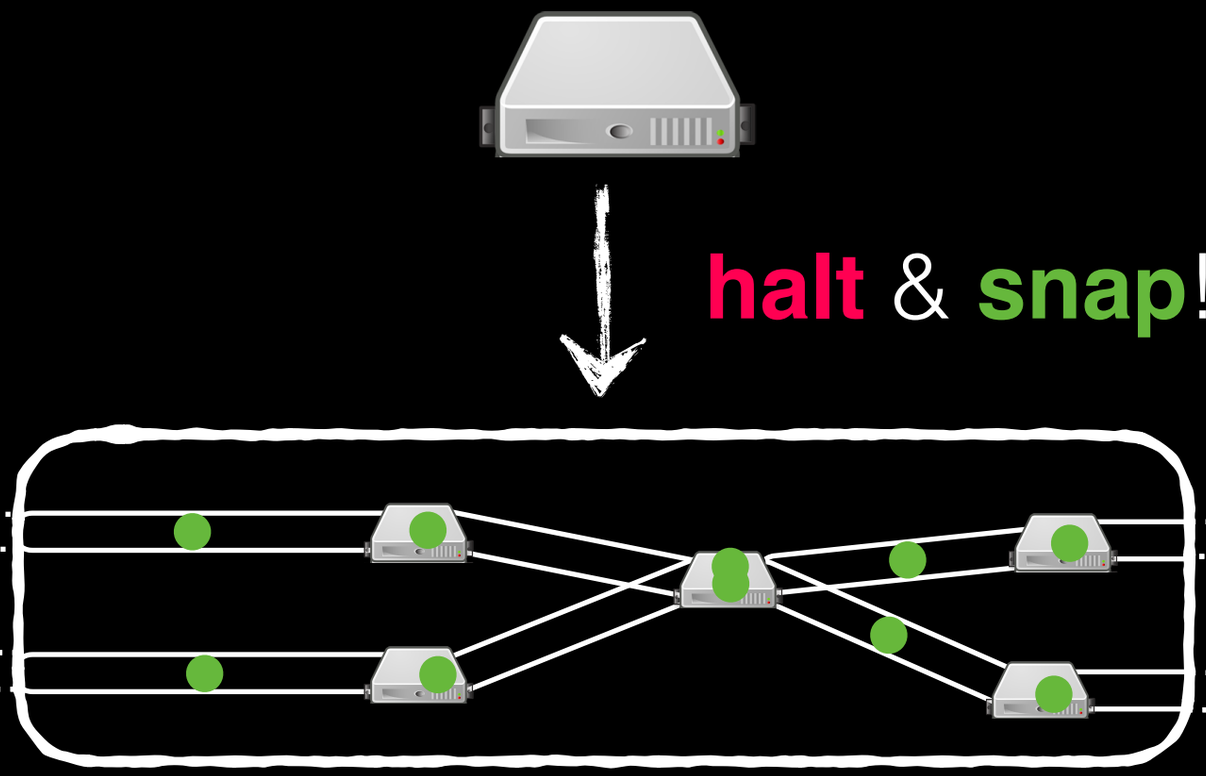
Coarse Grained Fault Tolerance - Illustrated

Type 2

Long-Running

Synchronous

*Snap **while** each part is being processed*



System State Store

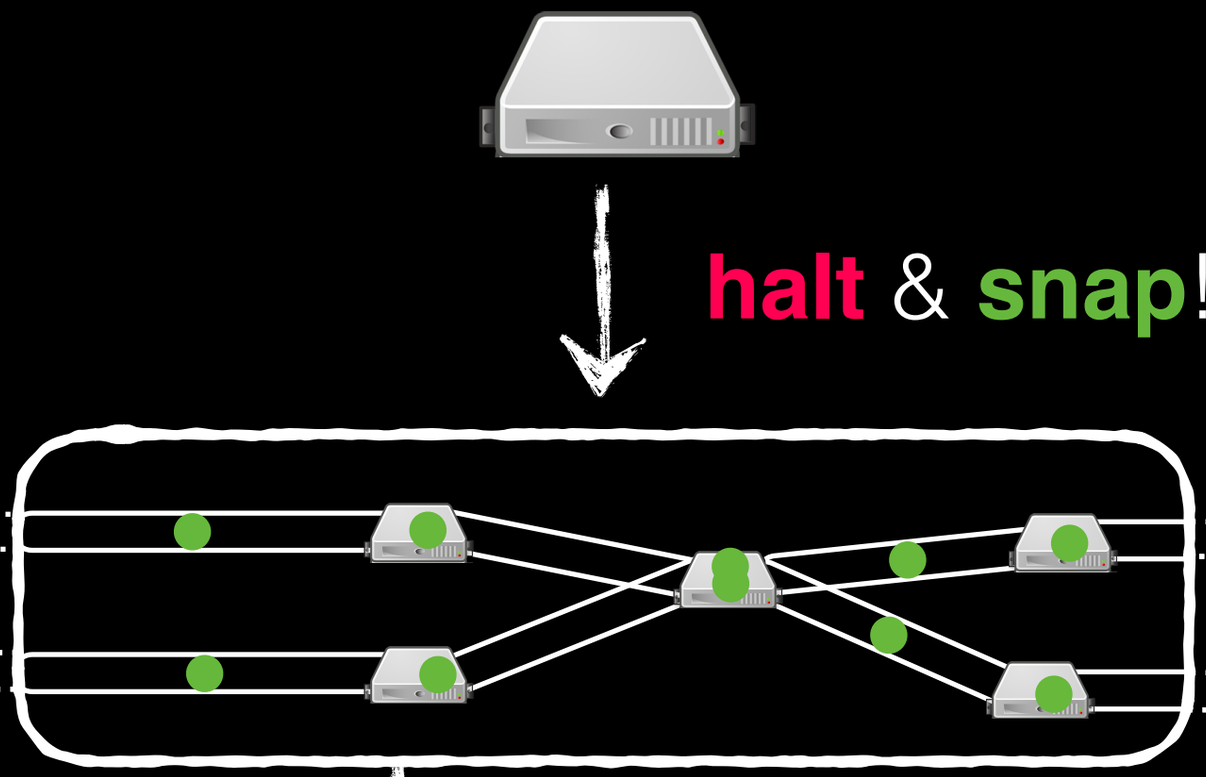
Coarse Grained Fault Tolerance - Illustrated

Type 2

Long-Running

Synchronous

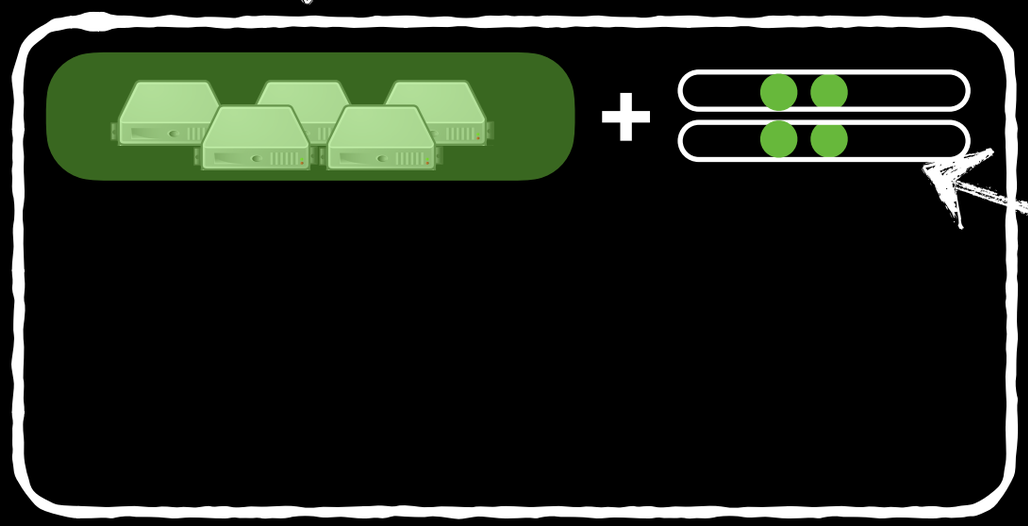
*Snap **while** each part is being processed*



halt & **snap**!

snap

System State Store



in-transit events to replay

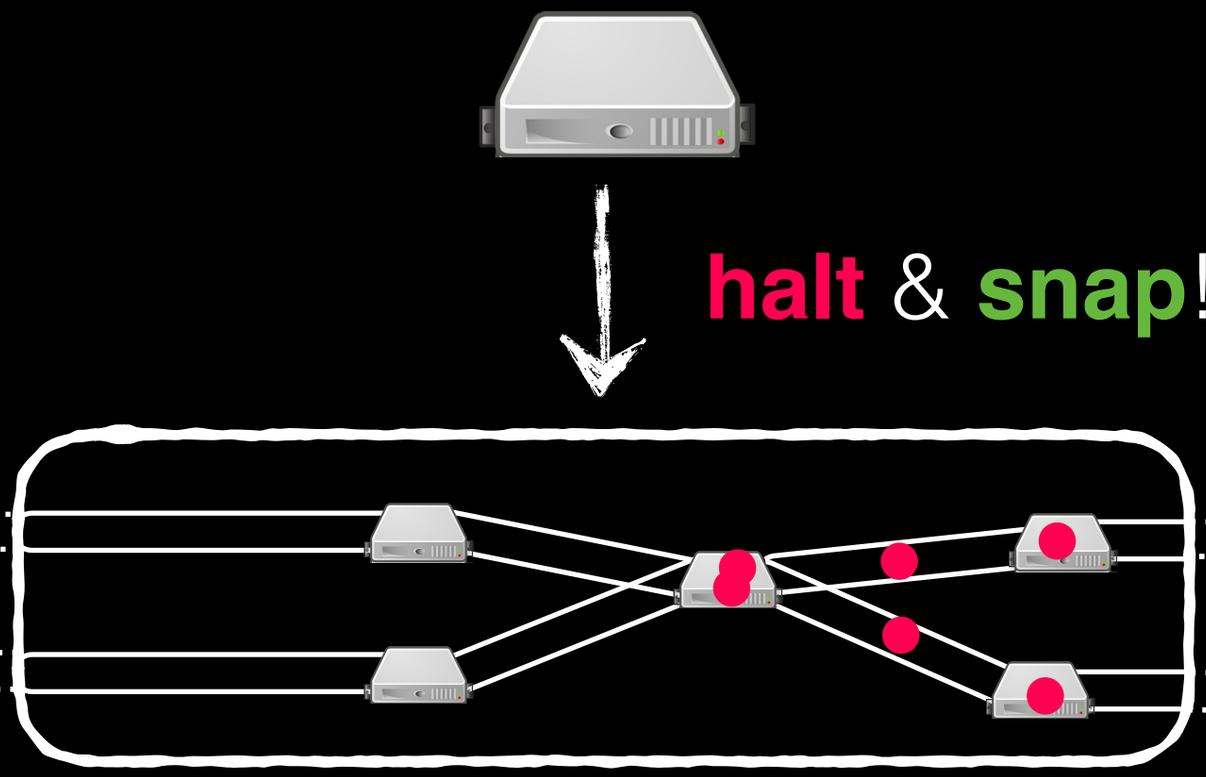
Coarse Grained Fault Tolerance - Illustrated

Type 2

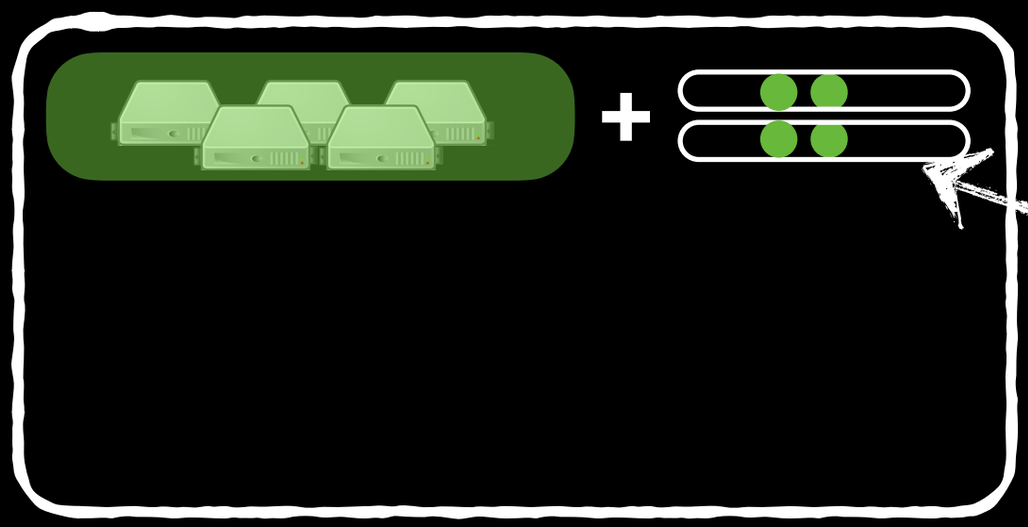
Long-Running

Synchronous

*Snap **while** each part is being processed*



System State Store



we want to **capture**
distributed **state without**

- **enforcing** it in the **API** or
- **disrupting** the **execution**

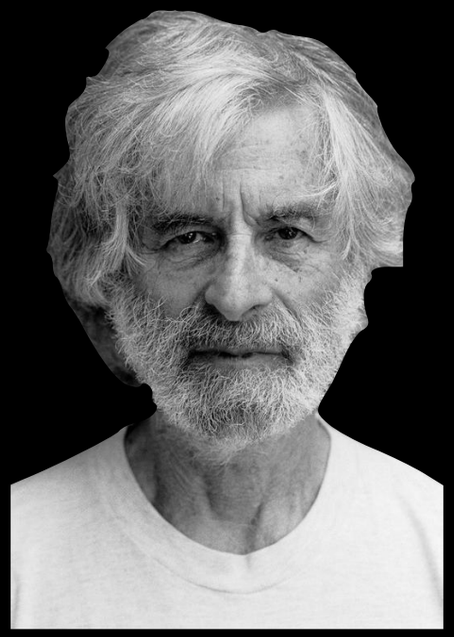
we want to **capture**
distributed **state without**

- **enforcing** it in the **API** or
- **disrupting** the **execution**

also, do we really need those in-transit events?

"The global-state-detection algorithm is to be superimposed on the underlying computation:

it must run concurrently with, but not alter, this underlying computation"



Leslie Lamport

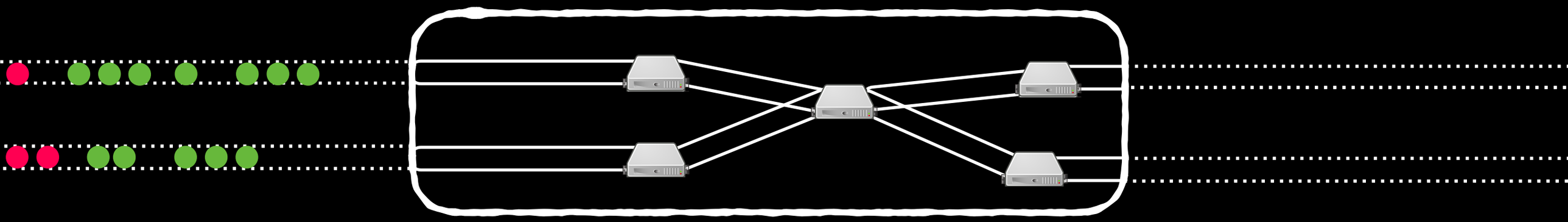
Coarse Grained Fault Tolerance - Illustrated

Type 3

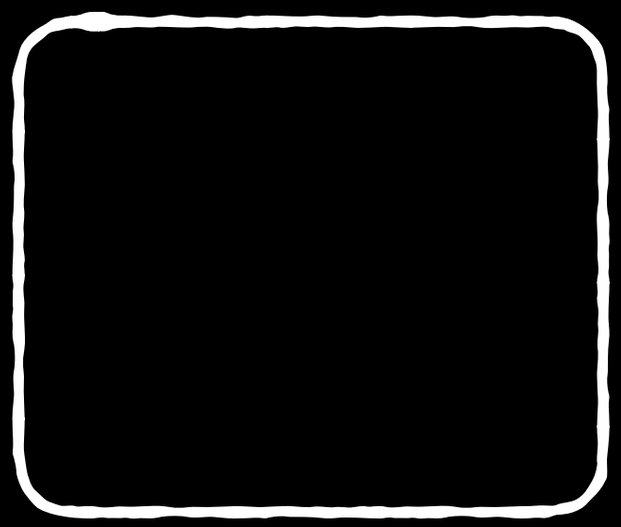
Long Running

*Pipelined**

*Snap **just in time!***



System State Store



* **https://arxiv.org/abs/1506.08603**

Coarse Grained Fault Tolerance - Illustrated

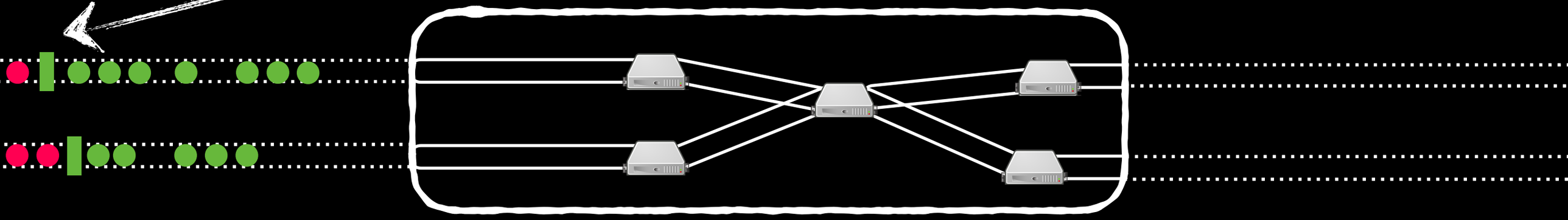
Type 3

Long Running

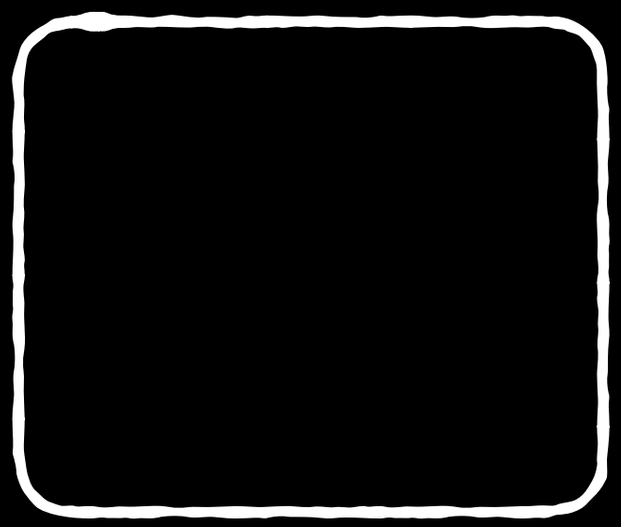
*Pipelined**

*Snap **just in time!***

insert markers



System State Store



* **https://arxiv.org/abs/1506.08603**

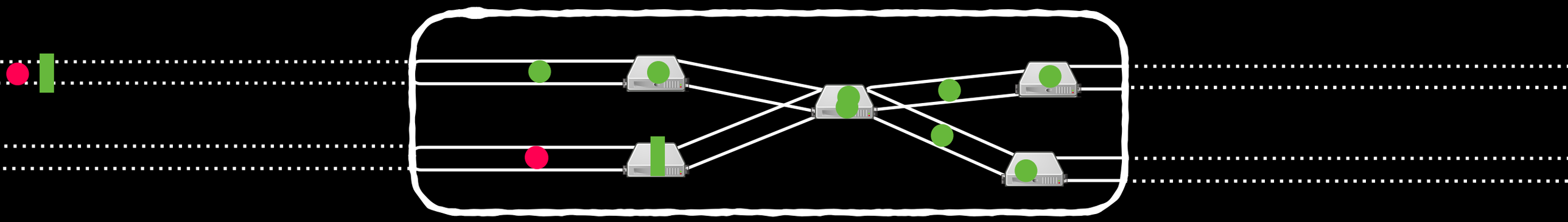
Coarse Grained Fault Tolerance - Illustrated

Type 3

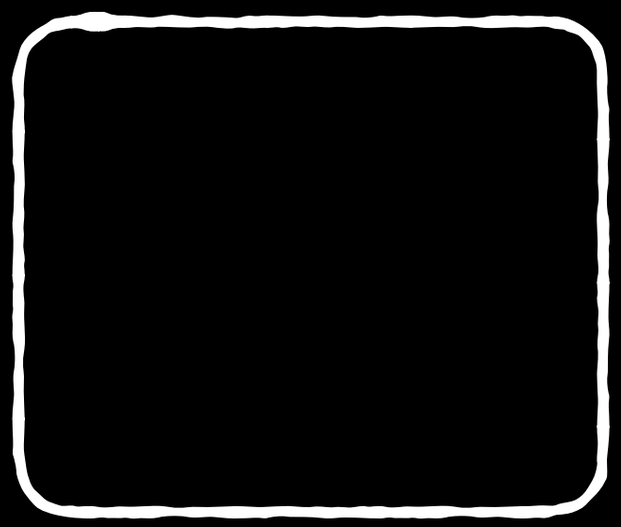
Long Running

*Pipelined**

*Snap **just in time!***



System State Store



* https://arxiv.org/abs/1506.08603

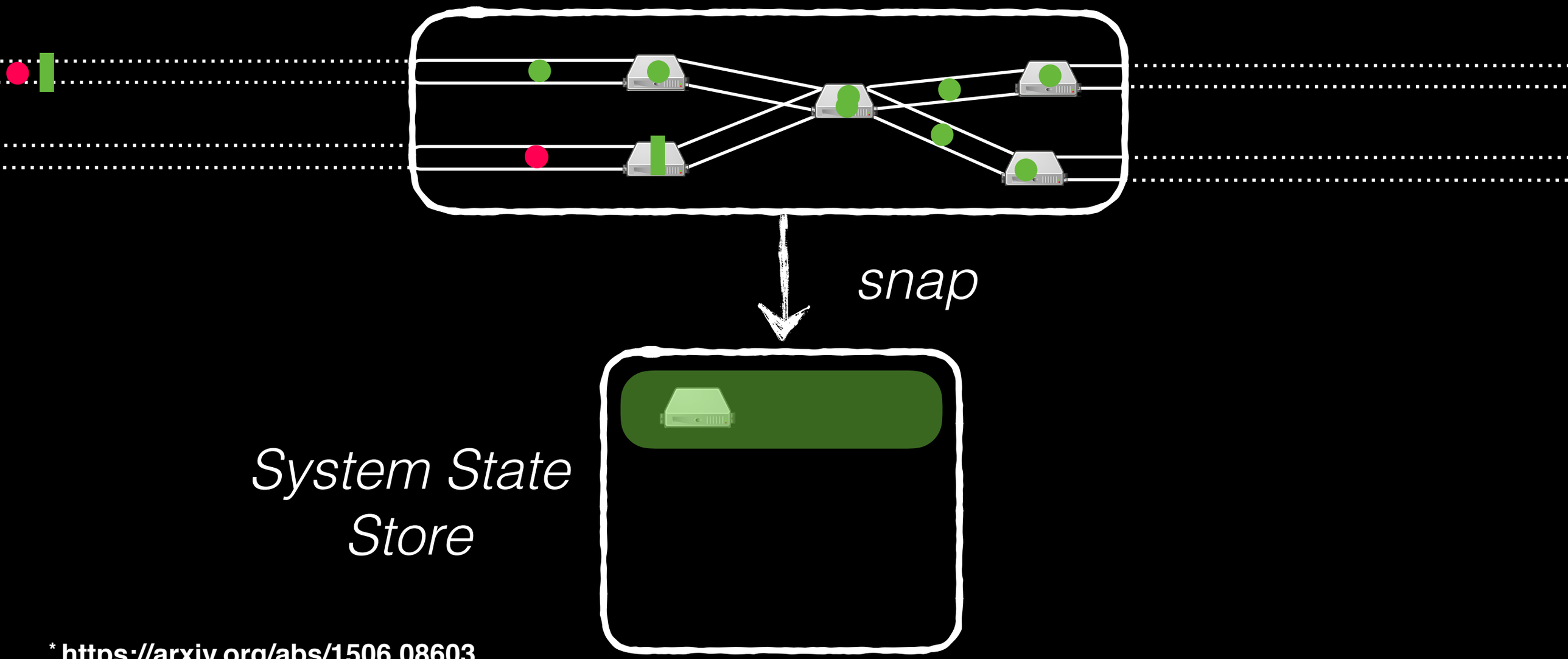
Coarse Grained Fault Tolerance - Illustrated

Type 3

Long Running

*Pipelined**

*Snap **just in time!***



* https://arxiv.org/abs/1506.08603

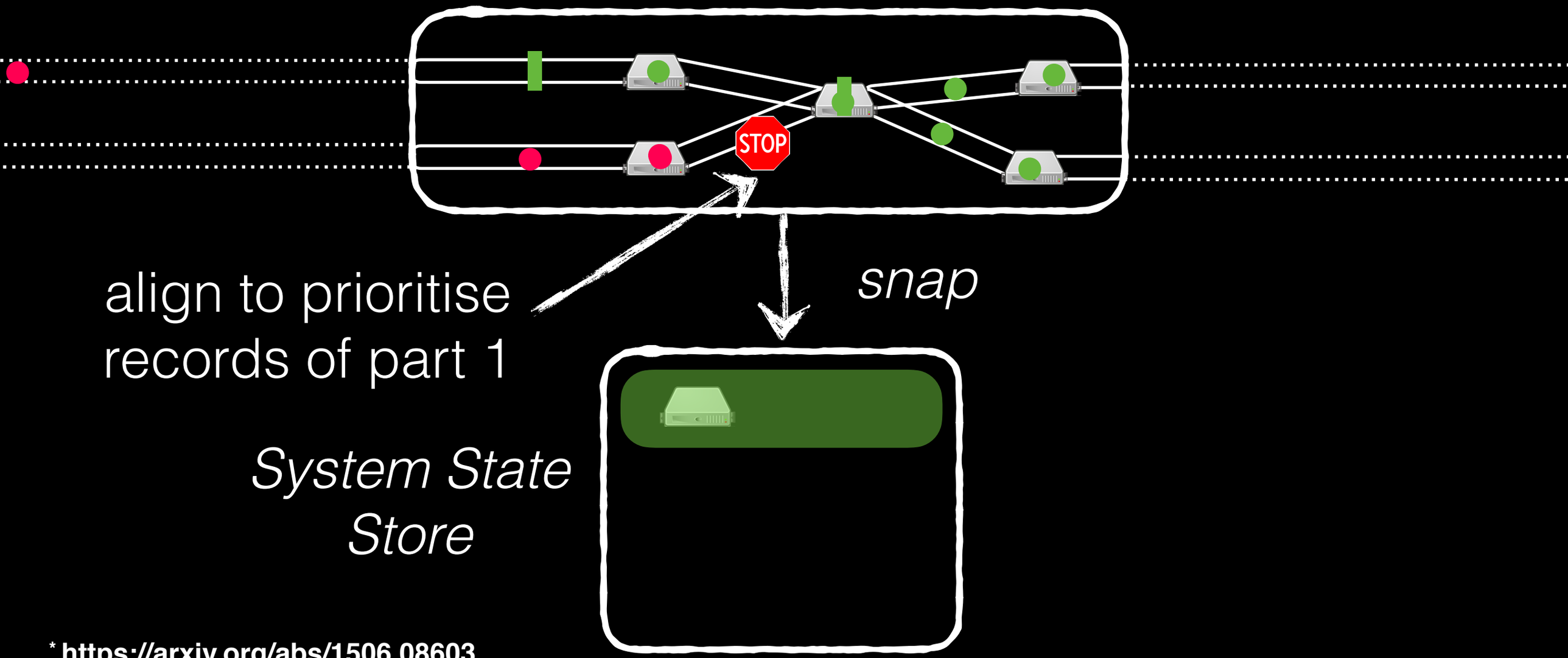
Coarse Grained Fault Tolerance - Illustrated

Type 3

Long Running

*Pipelined**

*Snap **just in time!***



* https://arxiv.org/abs/1506.08603

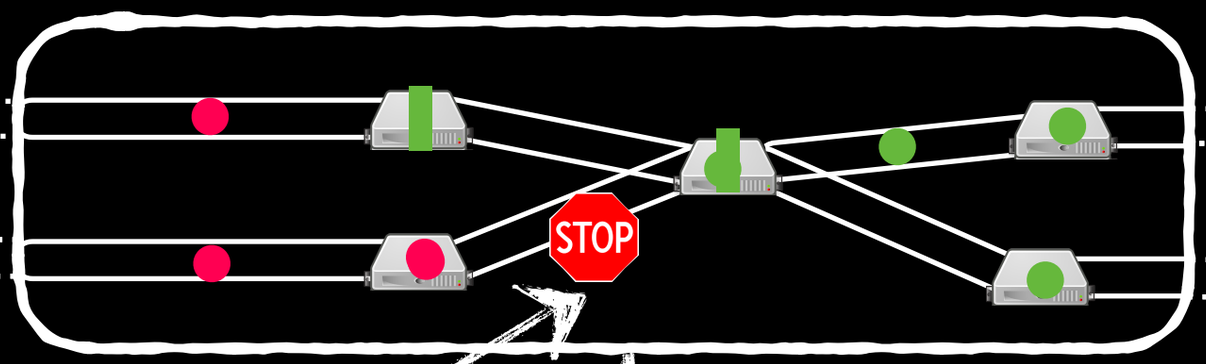
Coarse Grained Fault Tolerance - Illustrated

Type 3

Long Running

*Pipelined**

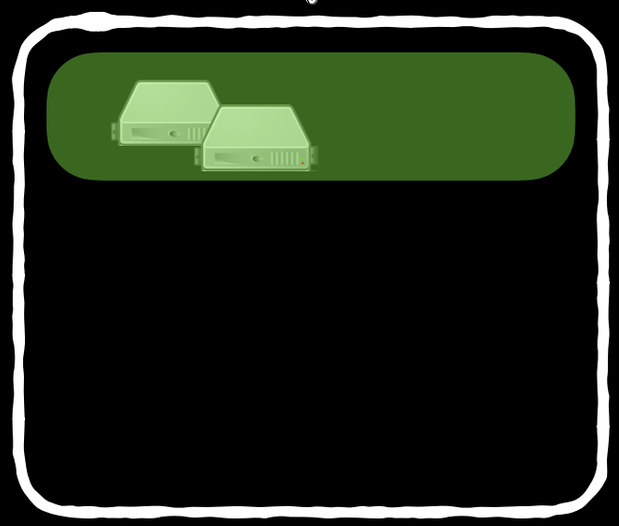
*Snap **just in time!***



align to prioritise
records of part 1

snap

System State Store



* **https://arxiv.org/abs/1506.08603**

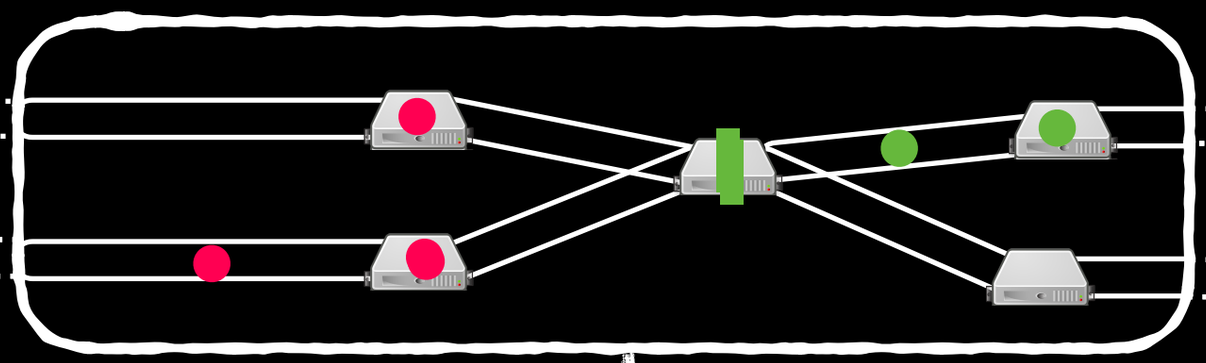
Coarse Grained Fault Tolerance - Illustrated

Type 3

Long Running

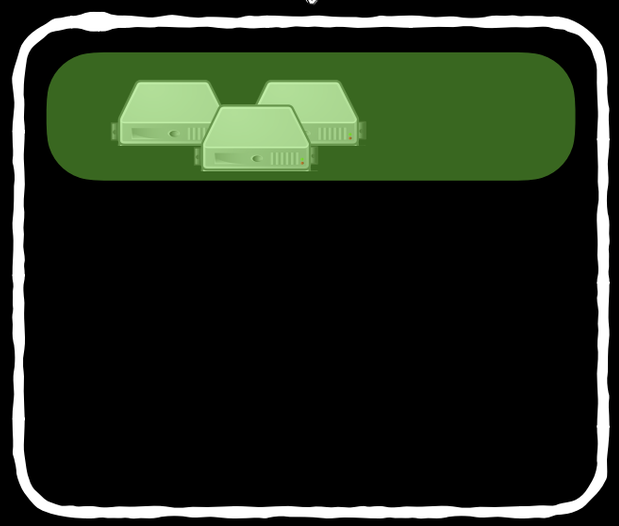
*Pipelined**

*Snap **just in time!***



snap

System State Store



* **https://arxiv.org/abs/1506.08603**

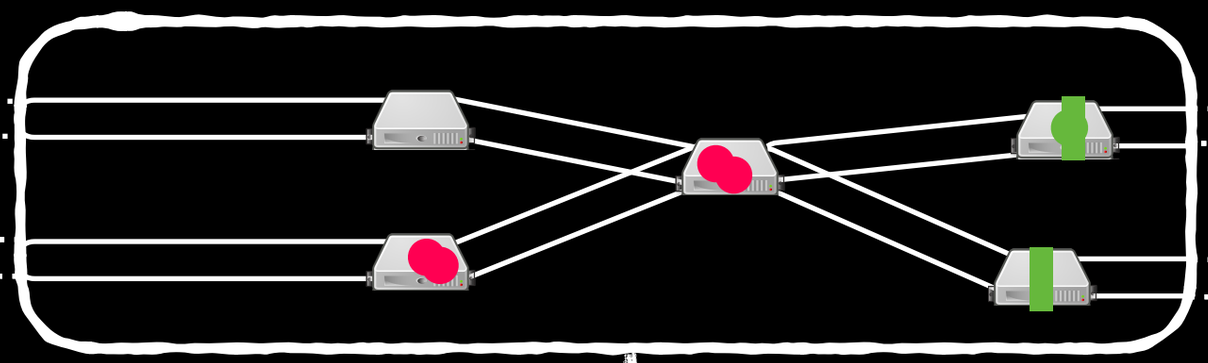
Coarse Grained Fault Tolerance - Illustrated

Type 3

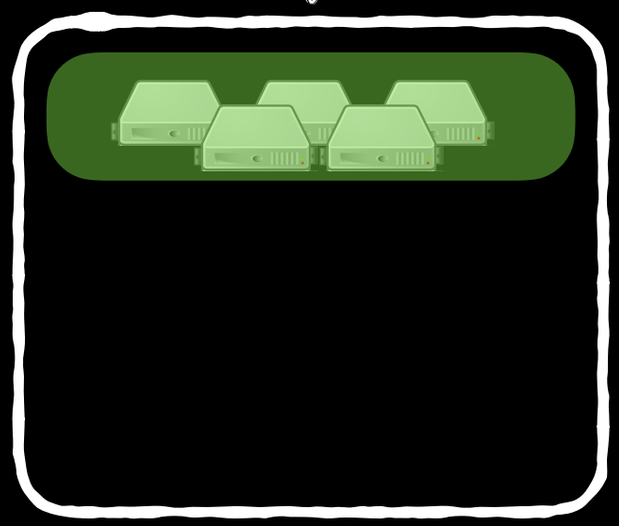
Long Running

*Pipelined**

*Snap **just in time!***



snap



System State Store

* **https://arxiv.org/abs/1506.08603**

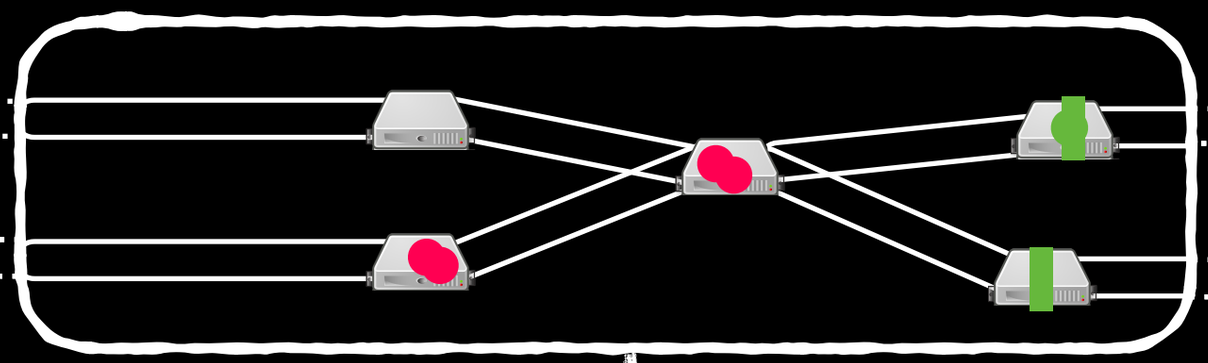
Coarse Grained Fault Tolerance - Illustrated

Type 3

Long Running

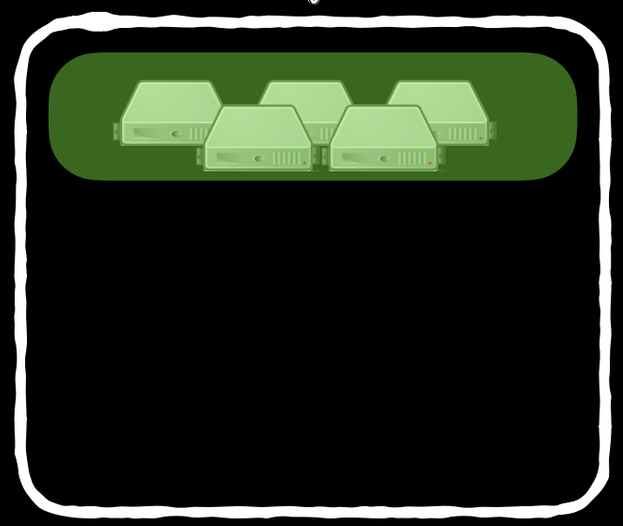
*Pipelined**

*Snap **just in time!***



snap

got a full snapshot!
with no records
in-transit



System State Store

* **https://arxiv.org/abs/1506.08603**

Facts about Flink's Snapshotting

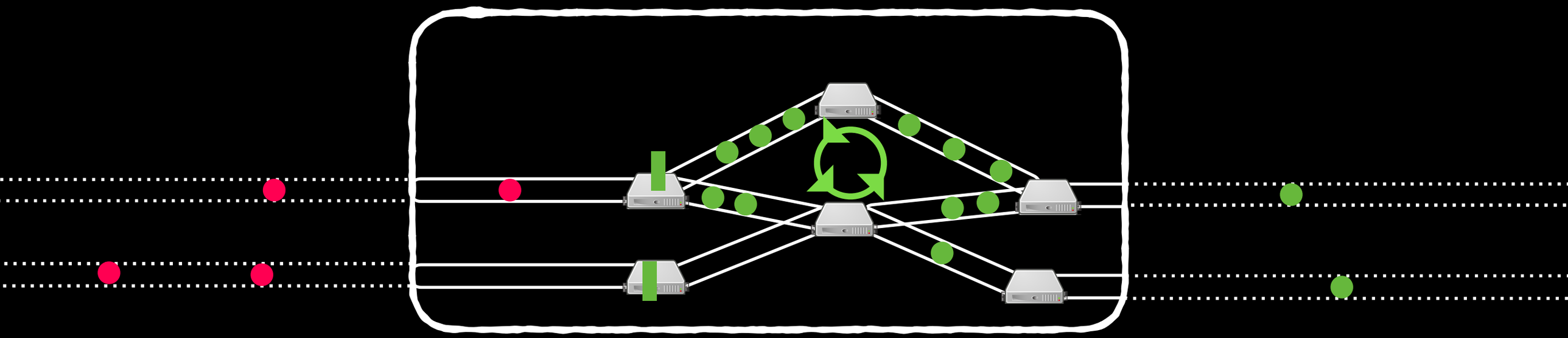
*http://lamport.azurewebsites.net/pubs/chandy.pdf

Facts about Flink's Snapshotting

- It **pipelines naturally** with the data-flow (respecting back-pressure etc.)
- We can get **at-least-once processing** guarantees by simply **dropping aligning** (try it)
- **Tailors** Chandy-Lamport's original approach* to dataflow graphs (with minimal snapshot state & messages)
- It can also work for cycles (with a minor modification)

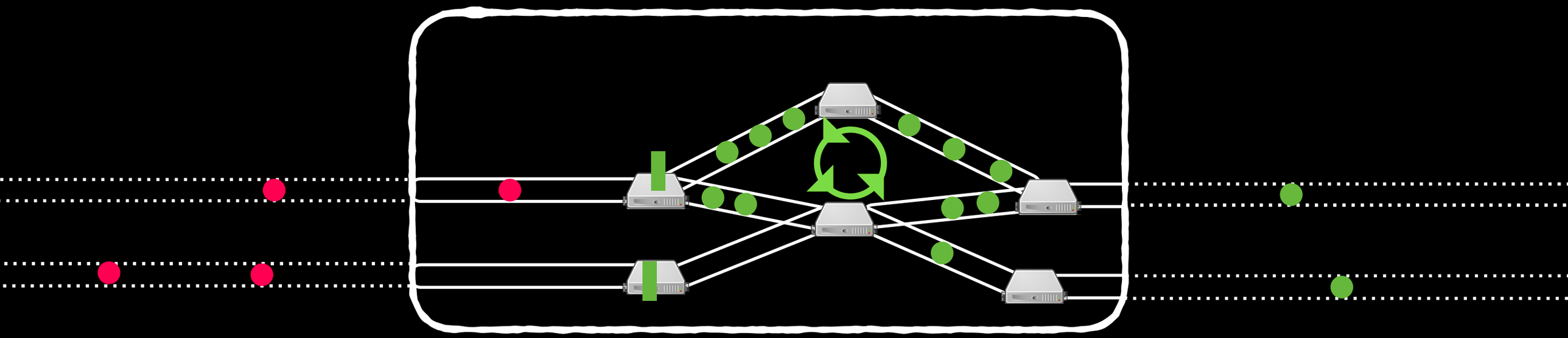
*http://lamport.azurewebsites.net/pubs/chandy.pdf

Supporting Cycles



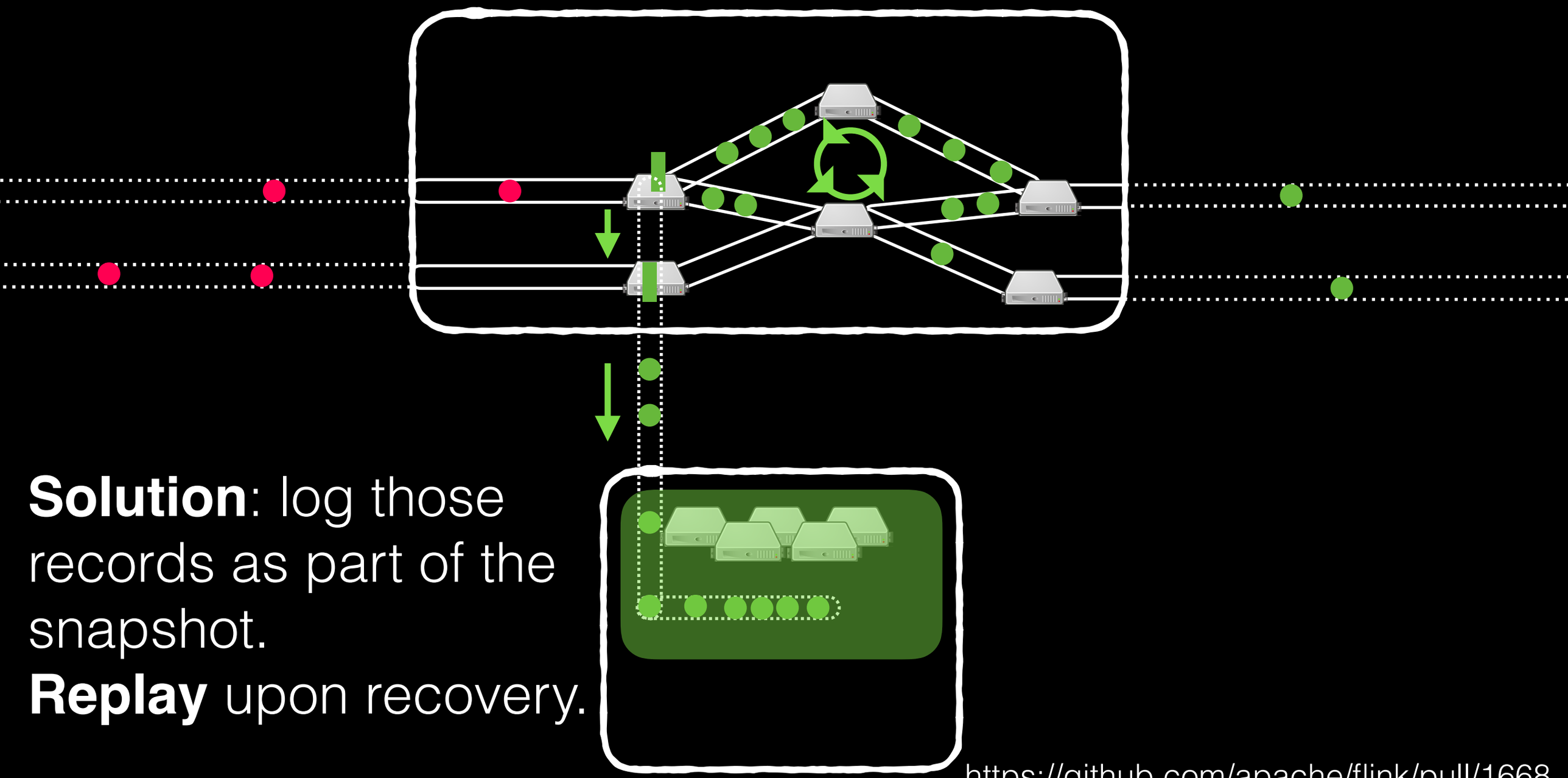
Supporting Cycles

Problem: we cannot wait indefinitely for records in cycles



Supporting Cycles

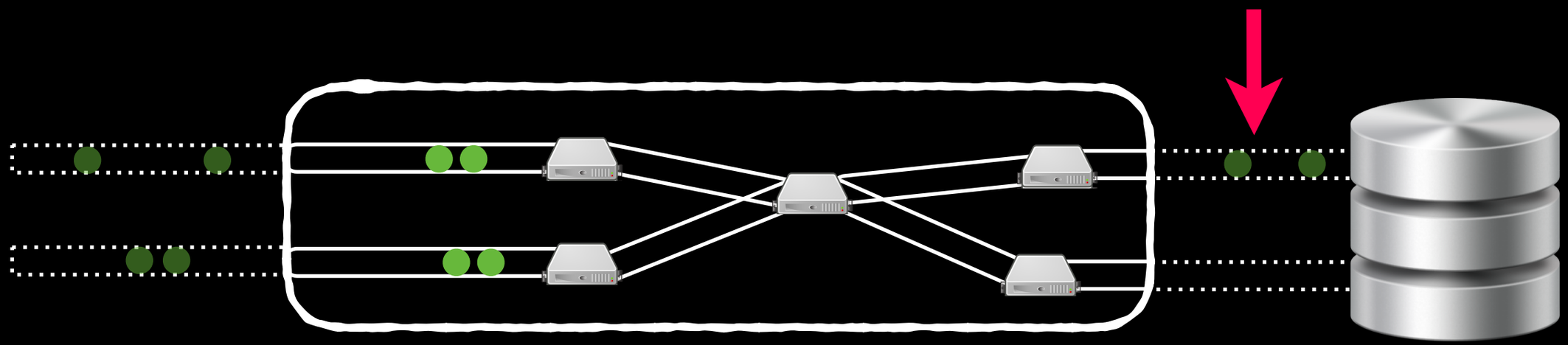
Problem: we cannot wait indefinitely for records in cycles



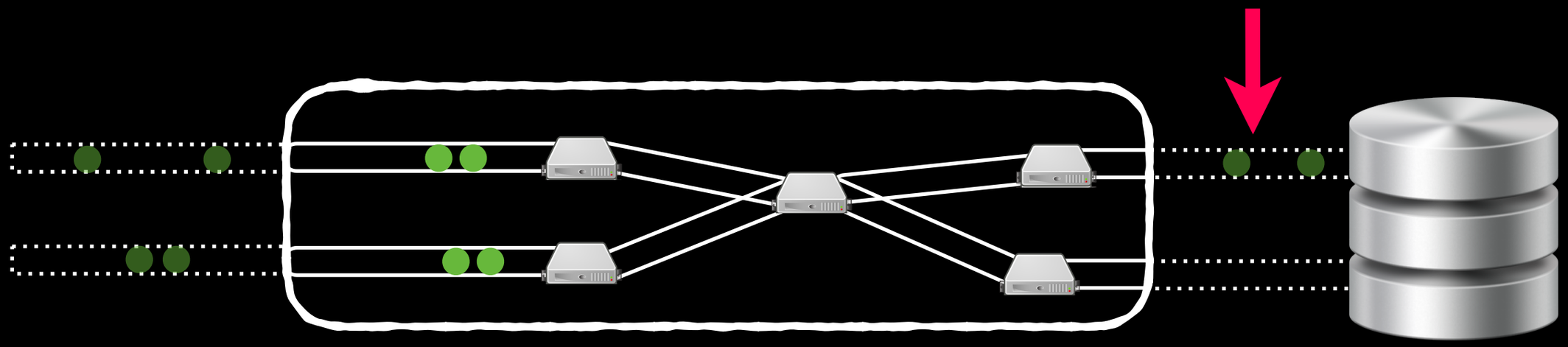
Solution: log those records as part of the snapshot.
Replay upon recovery.

https://github.com/apache/flink/pull/1668

Output Guarantees

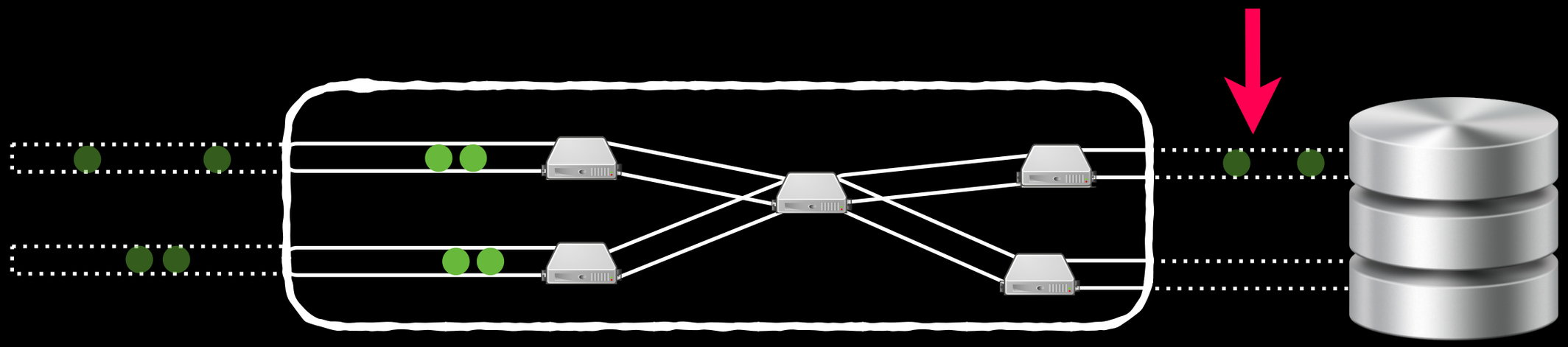


Output Guarantees



Is this a thing?

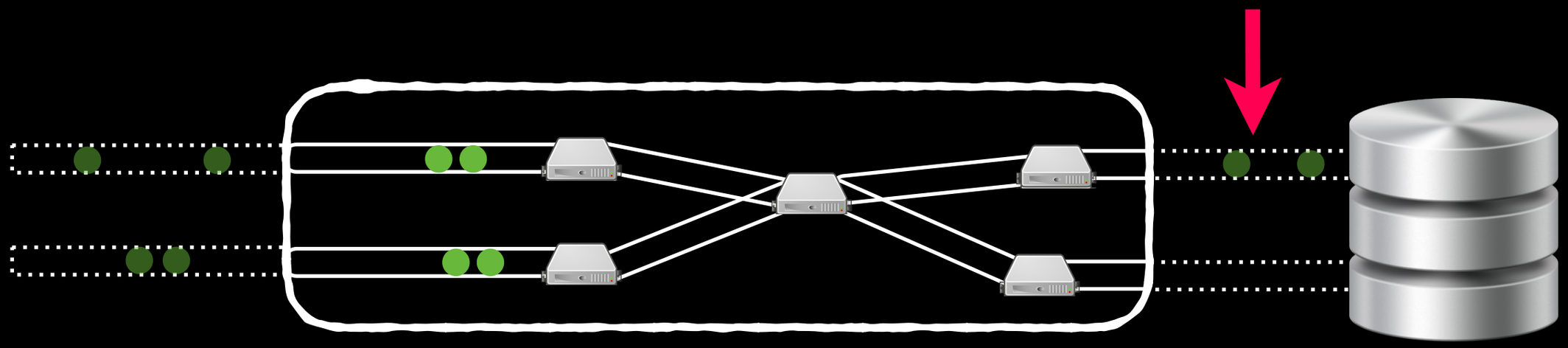
Output Guarantees



Is this a thing?

1. Can’t, it’s distributed

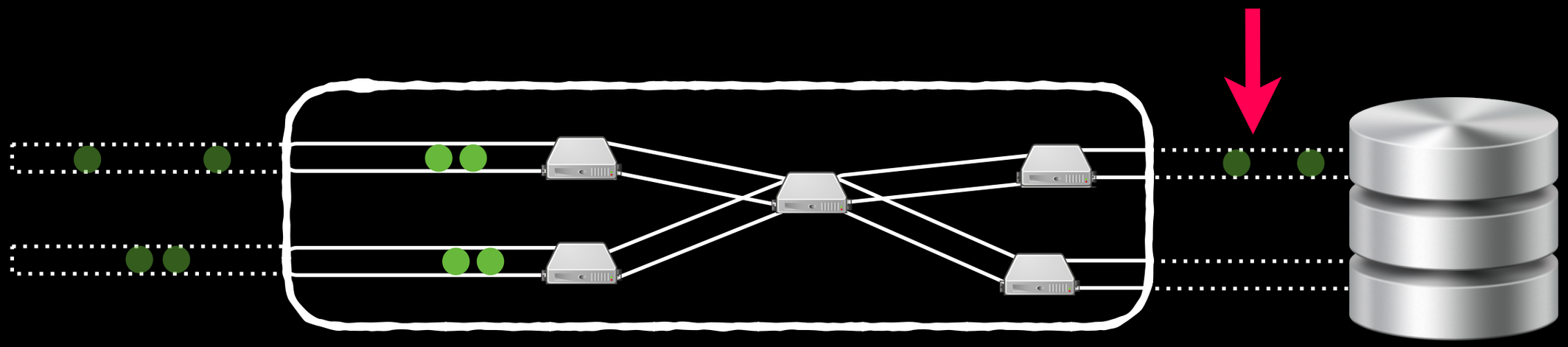
Output Guarantees



Is this a thing?

1. Can’t, it’s distributed
2. Yep easy

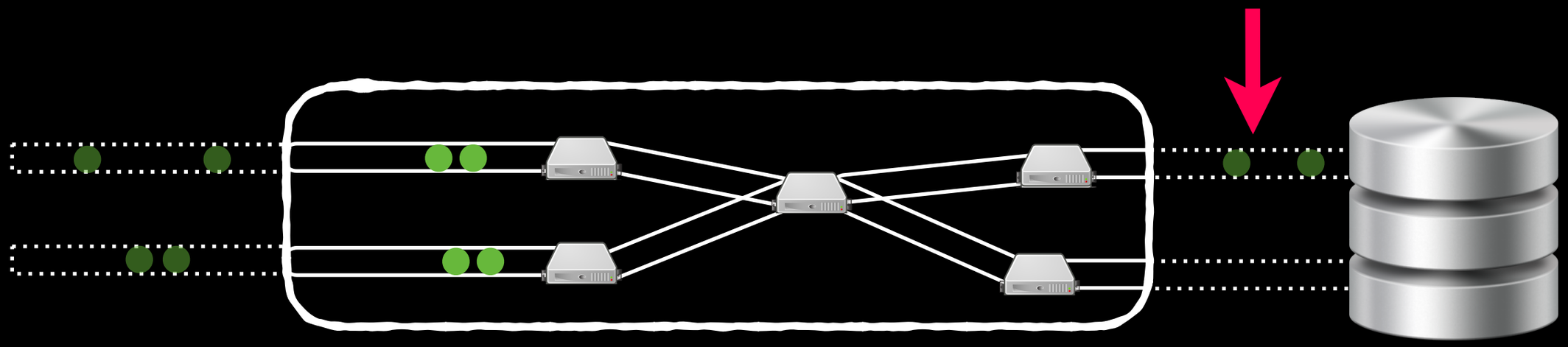
Output Guarantees



Is this a thing?

1. Can’t, it’s distributed
2. Yep easy
3. It depends ;)

Output Guarantees

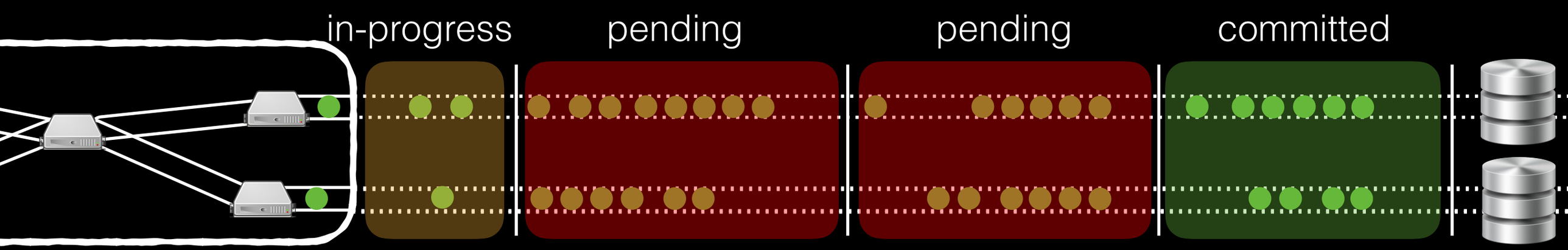


Is this a thing?

1. Can’t, it’s distributed
2. Yep easy
3. It depends ;)

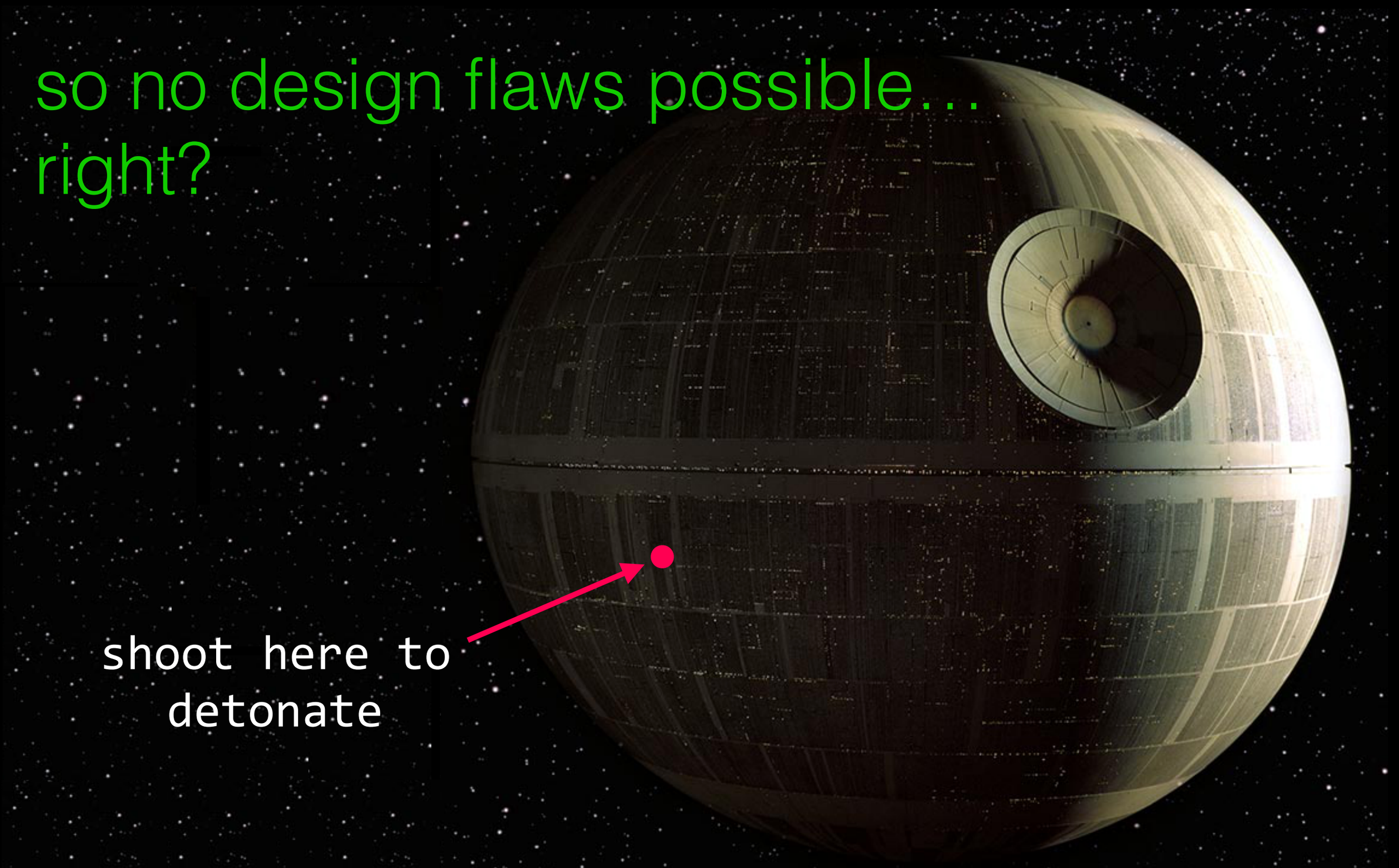
Exactly Once Output

- **Idempontency** ~ repeated operations give the same output result. (e.g., Flink's Cassandra sink*)
- **Rolling Files** ~ Pipeline output is bucketed and committed when a checkpoint is complete otherwise we roll it back. (see Flink's HDFS RollingSink**)



*https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/connectors/cassandra.html
**https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/connectors/filesystem_sink.html

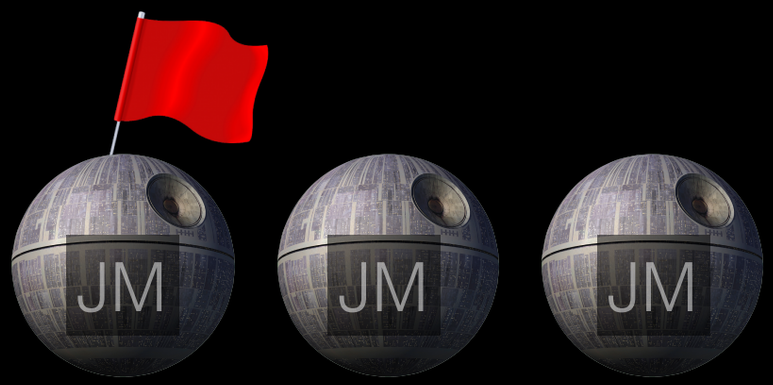
so no design flaws possible…
right?

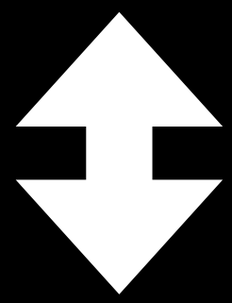


Sir, about that Job Manager…

> Abort Mission! They have HA

High Availability





Zookeeper

zab

State

- Current Leader (elected)
- Pending Pipeline Metadata
- State Snapshot Metadata

Perks of using Flink today (v1.2)

- Key-space partitioning and **key group** allocation
- **Job Rescaling** - from snapshots ;)
- **Async** state **snapshots** in **Rocksdb**
- **Managed State Structures** - ListState (append only), ValueState, ReducingState
- **Externalised Checkpoints** for custom cherry picking to rollback.
- **Adhoc** checkpoints (savepoints)

Coming up next

Autoscaling

Incremental Snapshots

Durable Iterative Processing

Acknowledgements

- Stephan Ewen, Ufuk Celebi, Aljoscha Krettek (and more folks at dataArtisans)
- Gyula Fóra (King.com)

and all contributors who have put code, effort and thought to build this unique state management system.

Not Less, Not More

Exactly Once, Large-Scale Stream Processing in Action

@SenorCarbone