DIANEMODB
RELIABLE GLOBAL SCALABILITY

Simplifying global-scale consistency

Rethinking database consistency

Today's Agenda

~55:00	01	Presentation
15:00	02	Q&A

DIANEMODB Vision & Mission

What

Offering better global-scale transactions

How

Creating a simple but robust, distributed, high availability transactionality solution based on the currently used standards like SQL

Why

Providing a single logical view of a unified transactional database platform for the world results in savings for everyone and lower barrier of entry into software

What we are about - theory

What could we do if we had a globally ordered, wait-free, highly available clock with perfect precision?

Wouldn't it be great if this clock would have no central consensus groups or need clock syncing?

What would it mean for our industry if we could do it simply enough for anyone to understand?

What we are about - practice

Serving query results from consistent followers could replace caches (which are external, inconsistent followers)

Retrofit ACID over a heterogeneous set of (even weakly consistent) datastores, even on the user's premises

Deal with replication and not needing observability for execution internals

Establish a single, unified platform, on which clusters could join others over arbitrary distances and still provide ANSI SQL semantics over any number of arbitrary schemas

Today's Outline

Some thoughts on time

Our algorithm

Complexity and costs

What we do now

Closing thoughts

01

02

03

04

05

It’s causality all the way down

> “
>
> We apply time mostly as a tool, or even just a metaphor, to help analyze causality. By saying ‚“a read execution R precedes a write W“ we usually wish to say ‚ ”R cannot be affected by the value written by W“ rather than ”if R terminates at 07:00 p.m. then W should not commence before 07:01 p.m.“
>
> ”

Abraham U., Ben-David S., Moran S. (1992) On the limitation of the global time assumption in distributed systems. In: Toueg S., Spirakis P.G., Kirousis L. (eds) Distributed Algorithms. WDAG 1991. Lecture Notes in Computer Science, vol 579. Springer, Berlin, Heidelberg. https://doi.org/10.1007/BFb0022434

The global time assumption

01

Strict serializability presumes speed of information to be infinite

02

Even in theory, distant computers can’t communicate faster than the speed of light

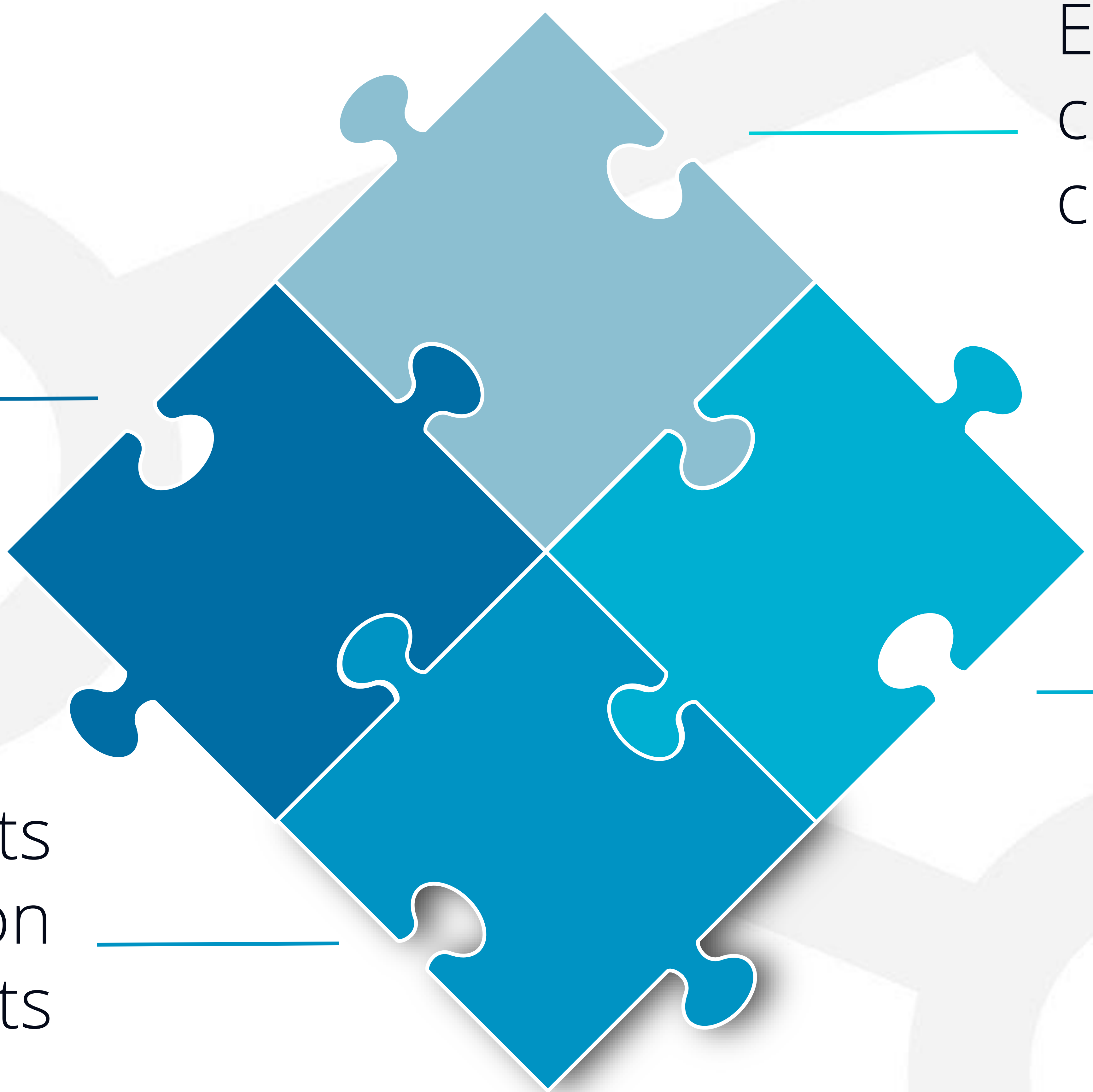
03

The shortest possible time to retrieve a piece of information half the globe away is ~67 ms
(on surface)

04

Google Spanner’s 6 ms is at least an order of magnitude too strict in some (identifiable) cases

Overview of our model I



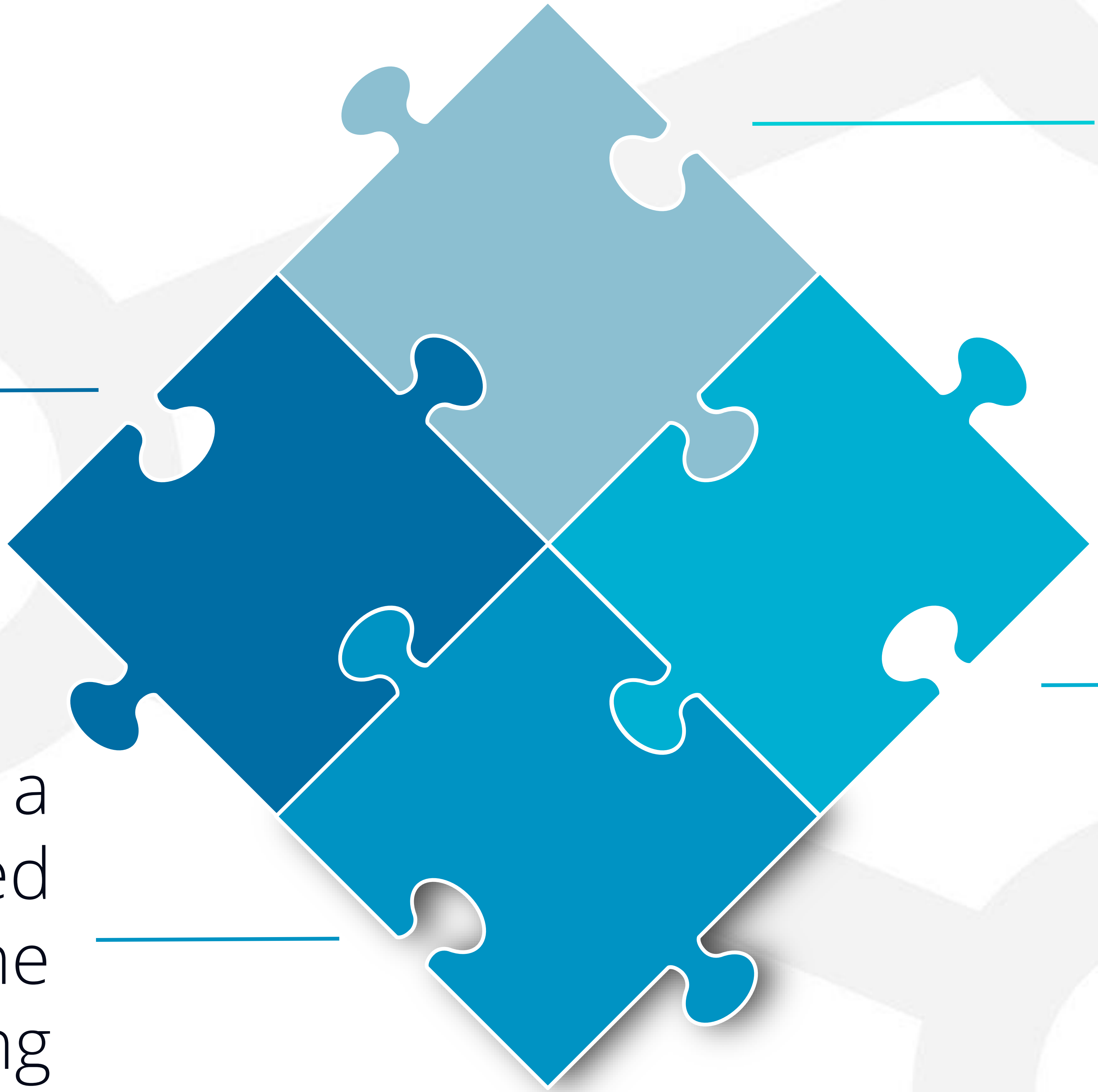
Establish global consensus over local consensus groups

Algorithm deals with both transactionality and replication / reliability

Strong consistency → SQL ANSI isolation levels, can create SQL DB

Transaction commits and clock progression must be atomic events

Overview of our model II



When reading, have a 'past timestamp', filter out any information that has a higher timestamp → no half-finished transactions

Optimistic locking, update what you read

Clock assigns commit versions locally, global translation established in a wait-free manner

Error if record has a higher committed version than the one you're updating

Data distribution

Records can be moved around different localities based on usage

Data locality is important for timeliness of information

Indices need an established range, which is mapped to nodes

Consistent read replicas can be established for both indices and records, lagging by less than half a second with data from across the globe

(In current impl.) data distribution is based on record attributes

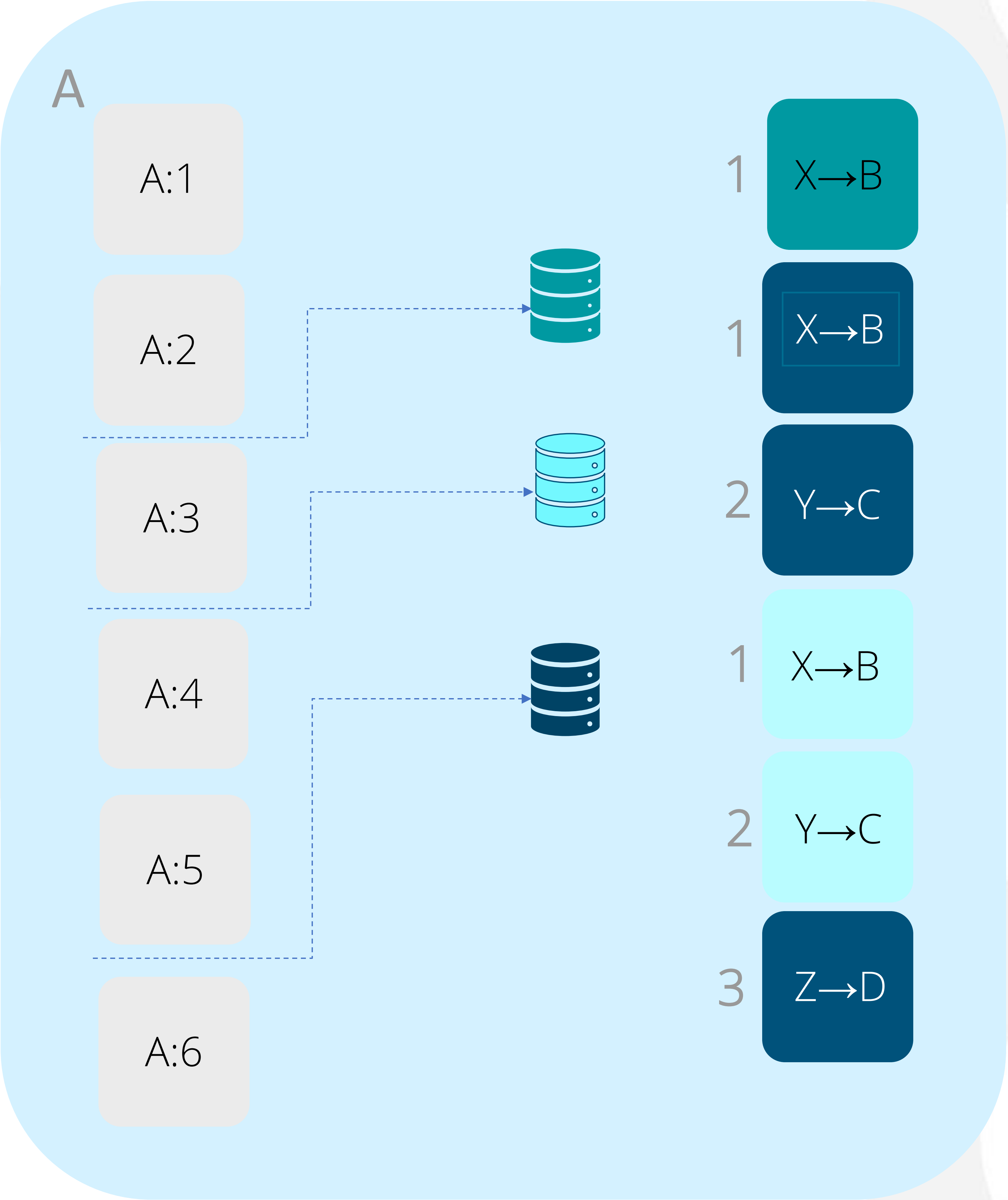
Solution-overview

Logical clock establishes a global order between any two transactions agreed on by any two observers

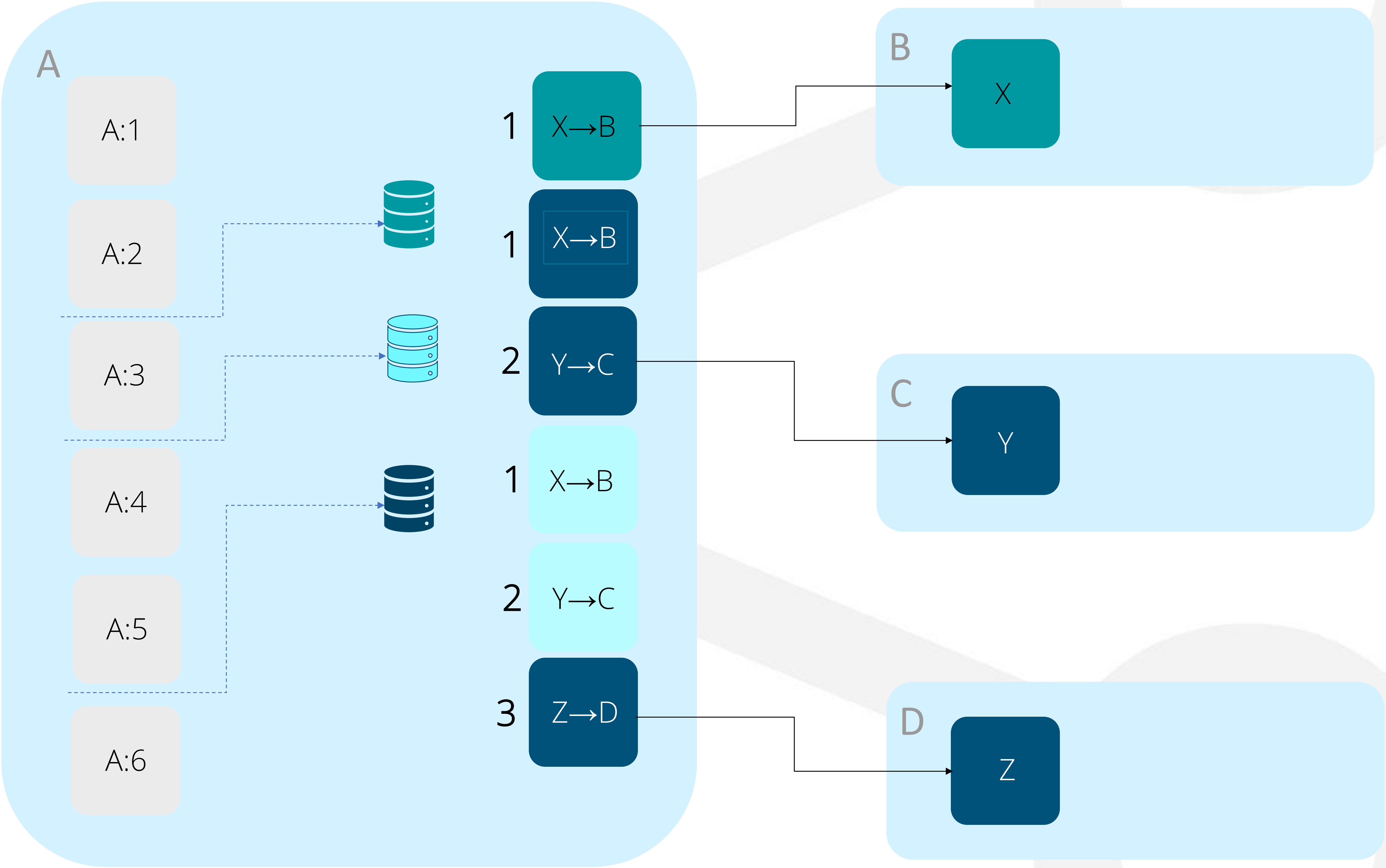
Commit algorithm ensures record-linearizability

Because time-definition is precise, protocol can be kept simpler

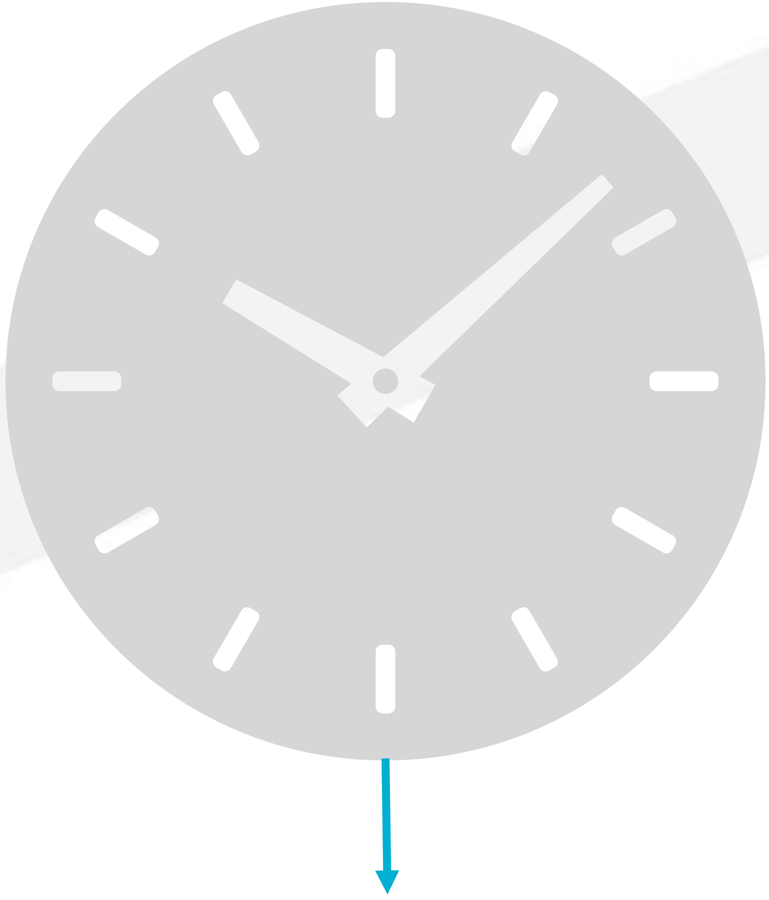
Clock event sourcing



Clock event sourcing



Clock definitions



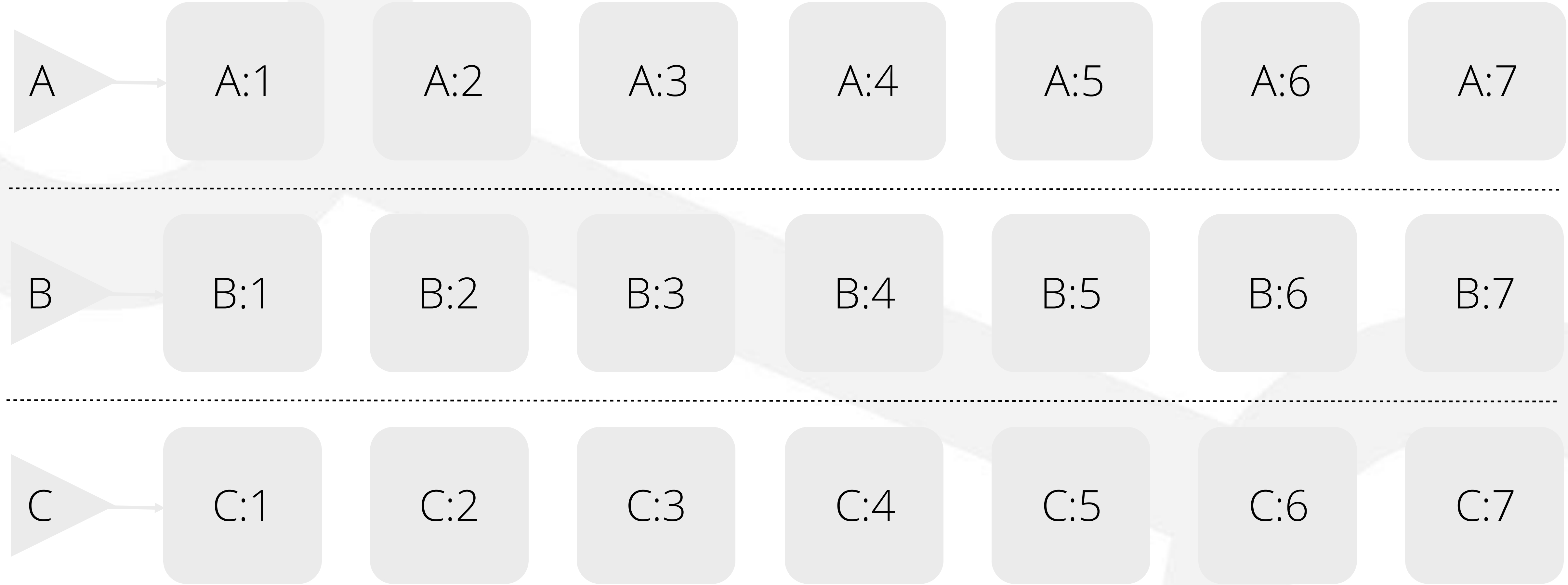
- Nodes are represented by their input events here (inputs = state)
- Small collection of ordered nodes is called a "group"
- They communicate with each other through event logs only
- Node stores all state, so both data and intermediate tx states

Independent consensus groups

Separate series of events, independent of each other.

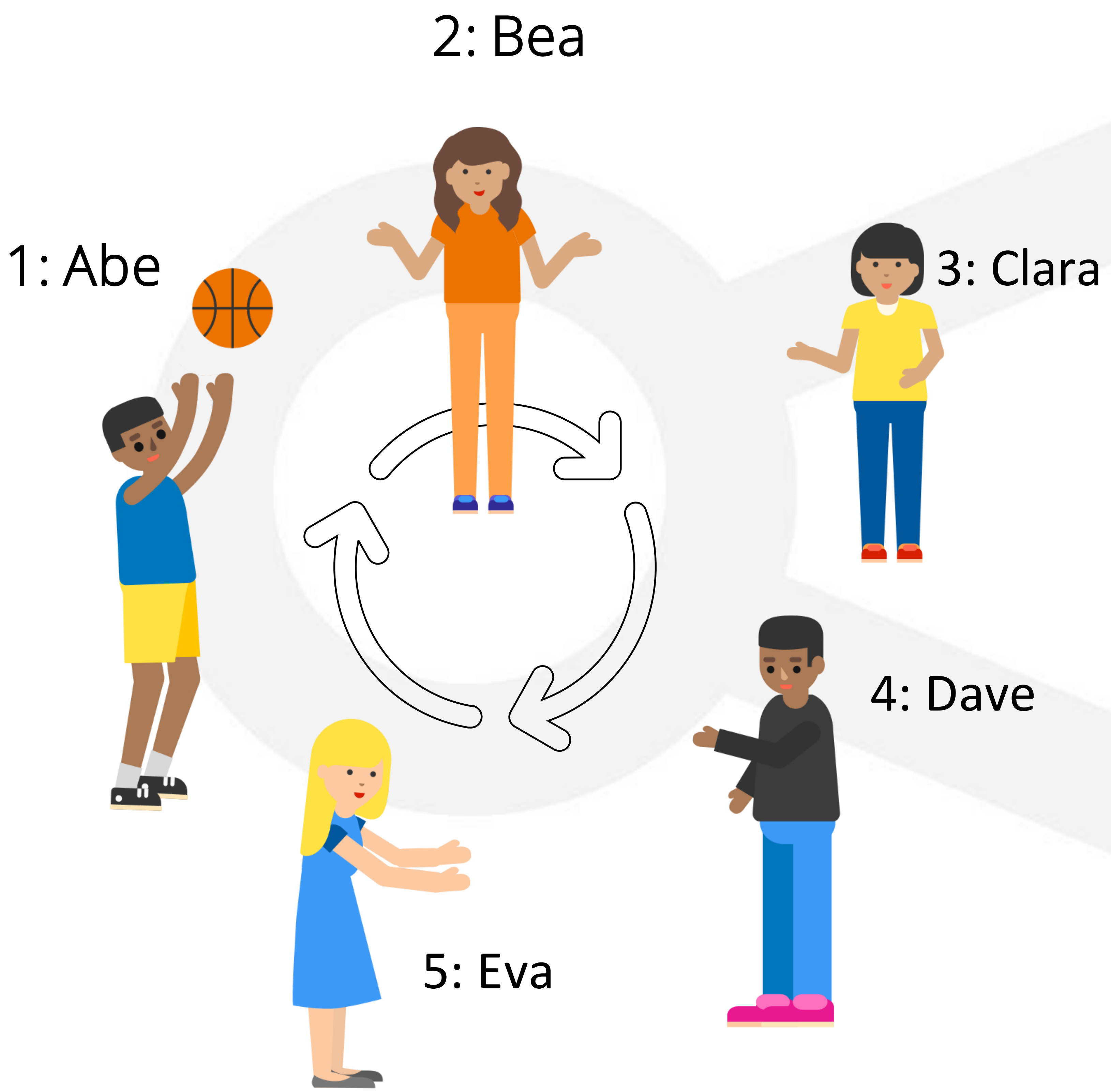
Potentially replicated, (somewhat) localised consensus-groups.

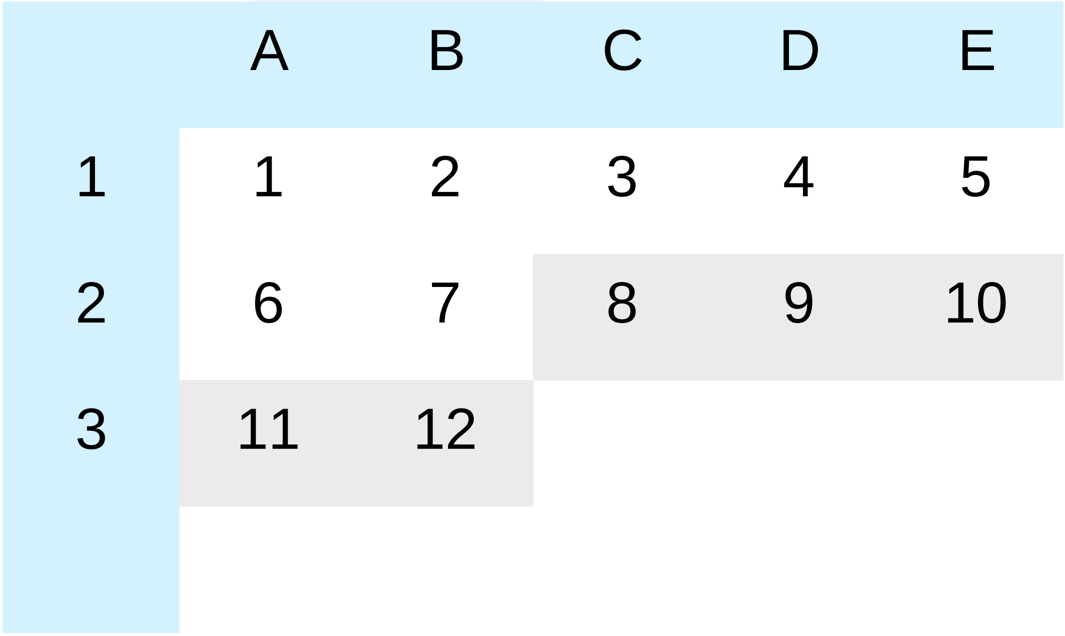
In this example, we have 3 event logs (Kafka partitions)



A	A:1	A:2	A:3	A:4	A:5	A:6	A:7
B	B:1	B:2	B:3	B:4	B:5	B:6	B:7
C	C:1	C:2	C:3	C:4	C:5	C:6	C:7

Token passing





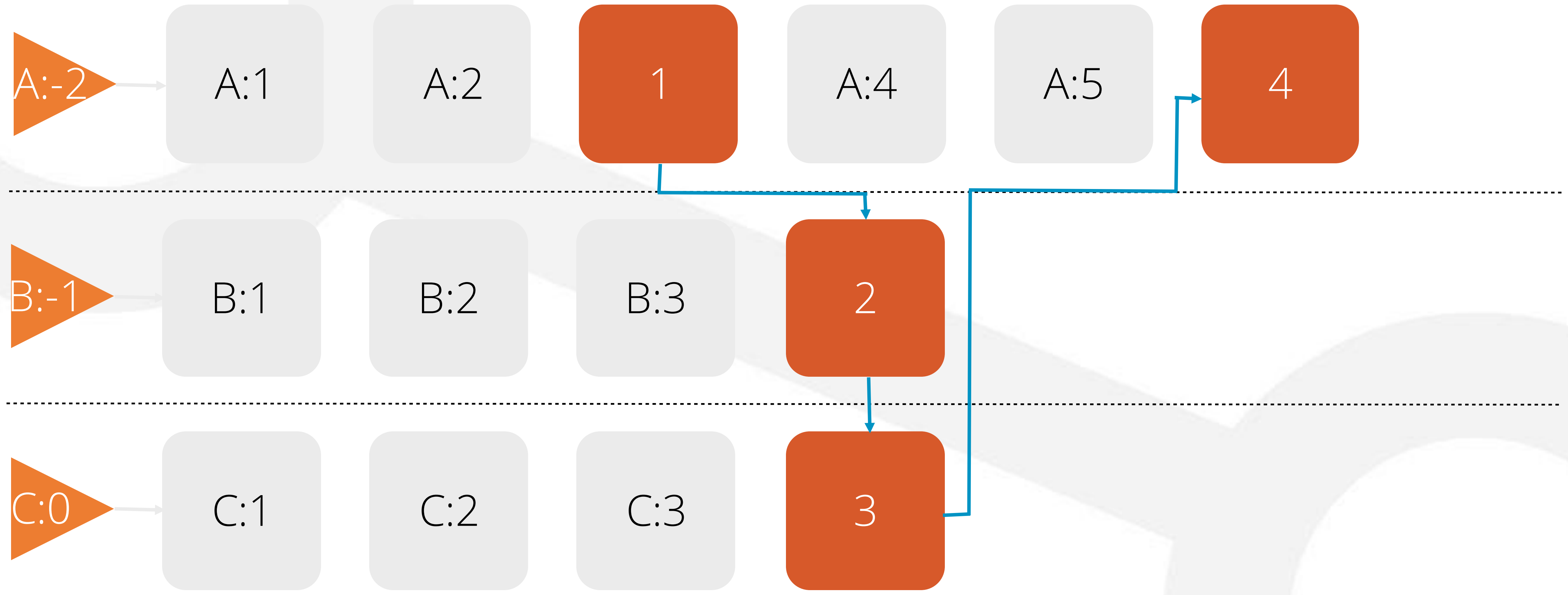
	A	B	C	D	E
1	1	2	3	4	5
2	6	7	8	9	10
3	11	12			

i + 5 * n (i: index, n: # of circles)

i + g * n (i: index, g: # in group, n: # of circles)

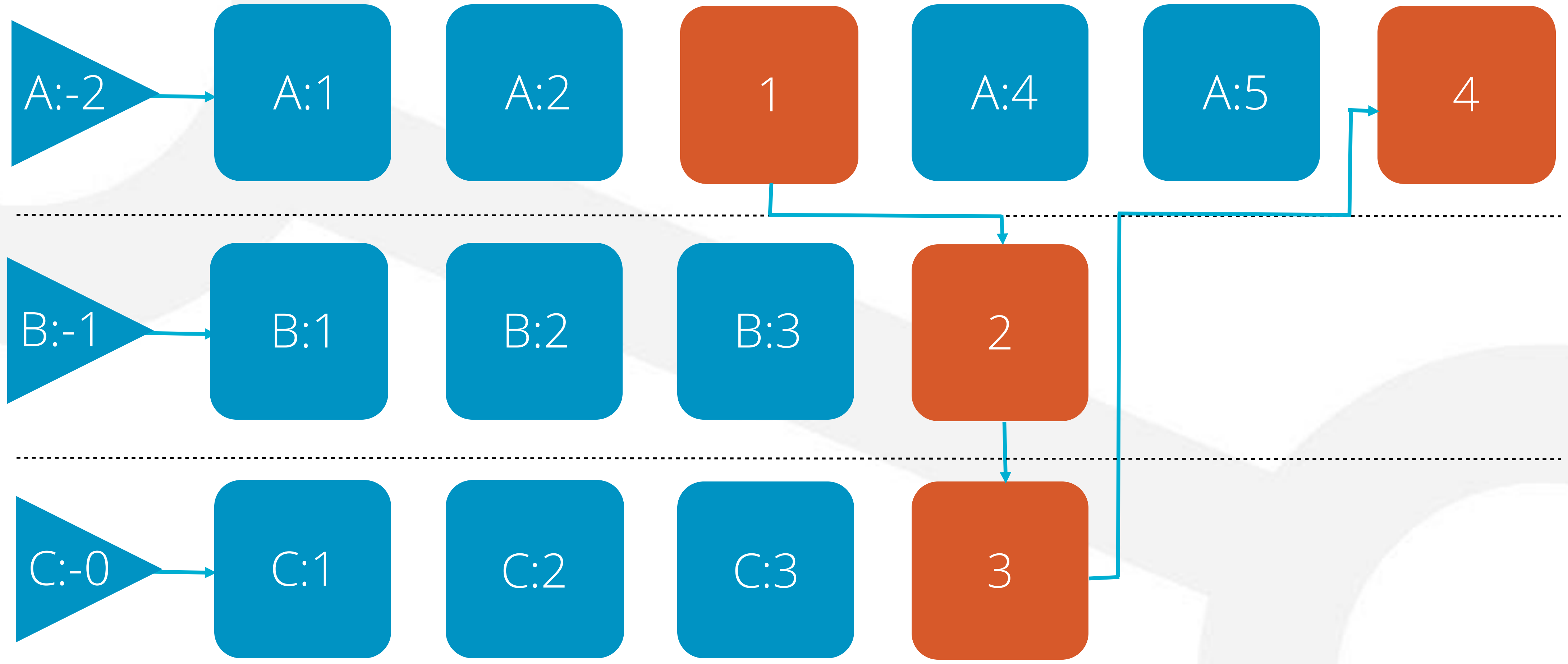
Token passing

A token-message is passed around by these in a continuous circle.
This establishes an increasing number.
Last one received is called the “closed” version, the next we’ll receive is “open”.
Sequence for first one is: 1, 4, 7 … i + g * n (i: index, g: # in group, n: # of circles).

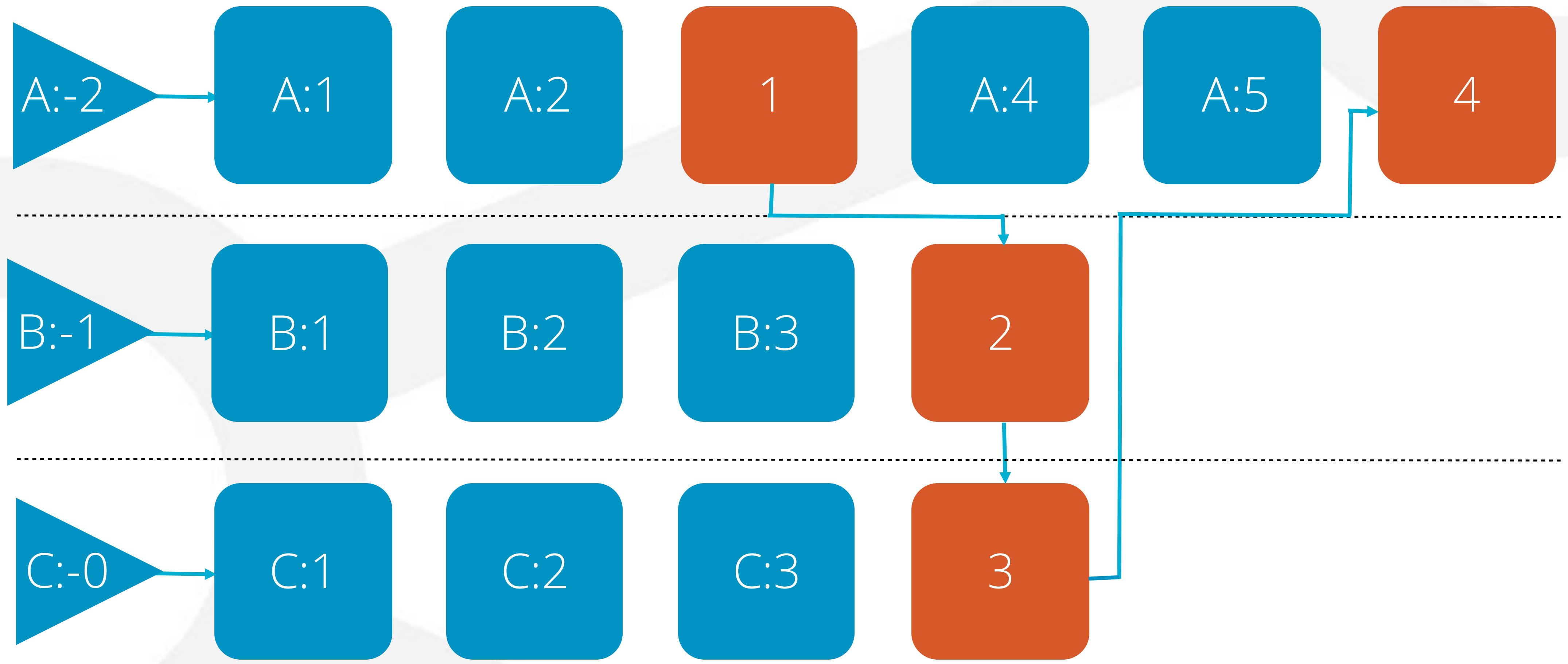


Local clock

The set of information represented by closed is final.
The sets represented by open is still append-able.
The blue squares represent readable, consistent state.
Closed on this node means everything lower is also closed on others in group.



Group clock- logical view

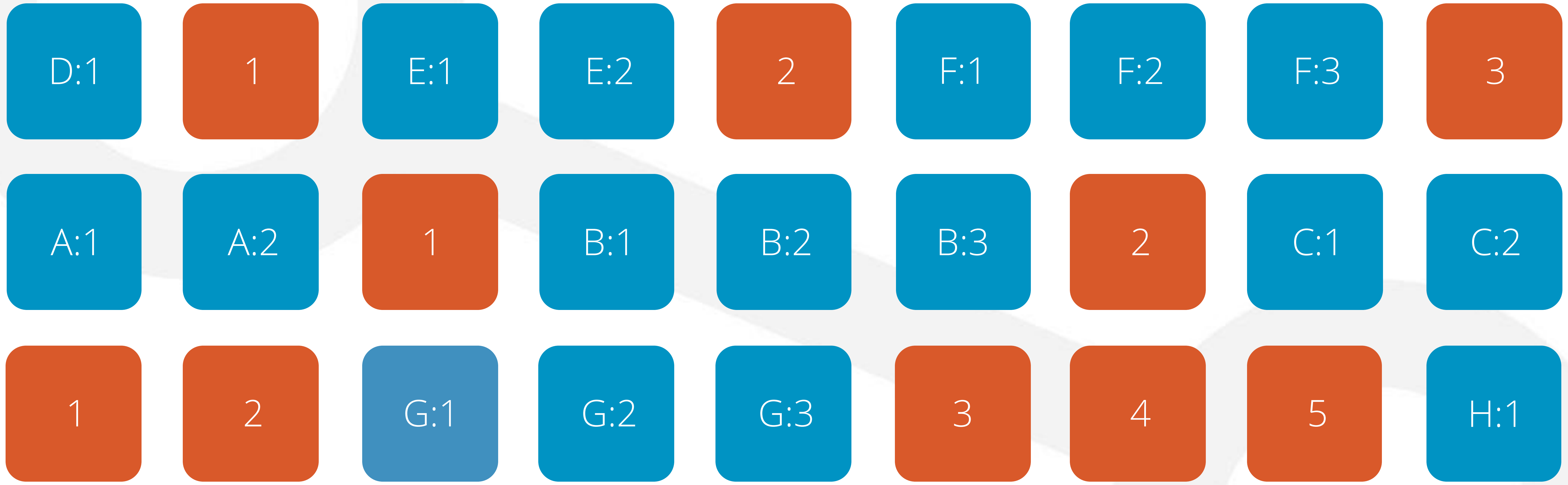


Clock group summarised

- “Open” means that it’s still gathering information
- “Closed” means that its information is final
- We have a version we can rely on, which we can assign on each node
- The data represented will be exposed to readers atomically
- Establishes a logical clock between different nodes
- How do we go from a single group to a global cluster?

Visualising multiple clocks

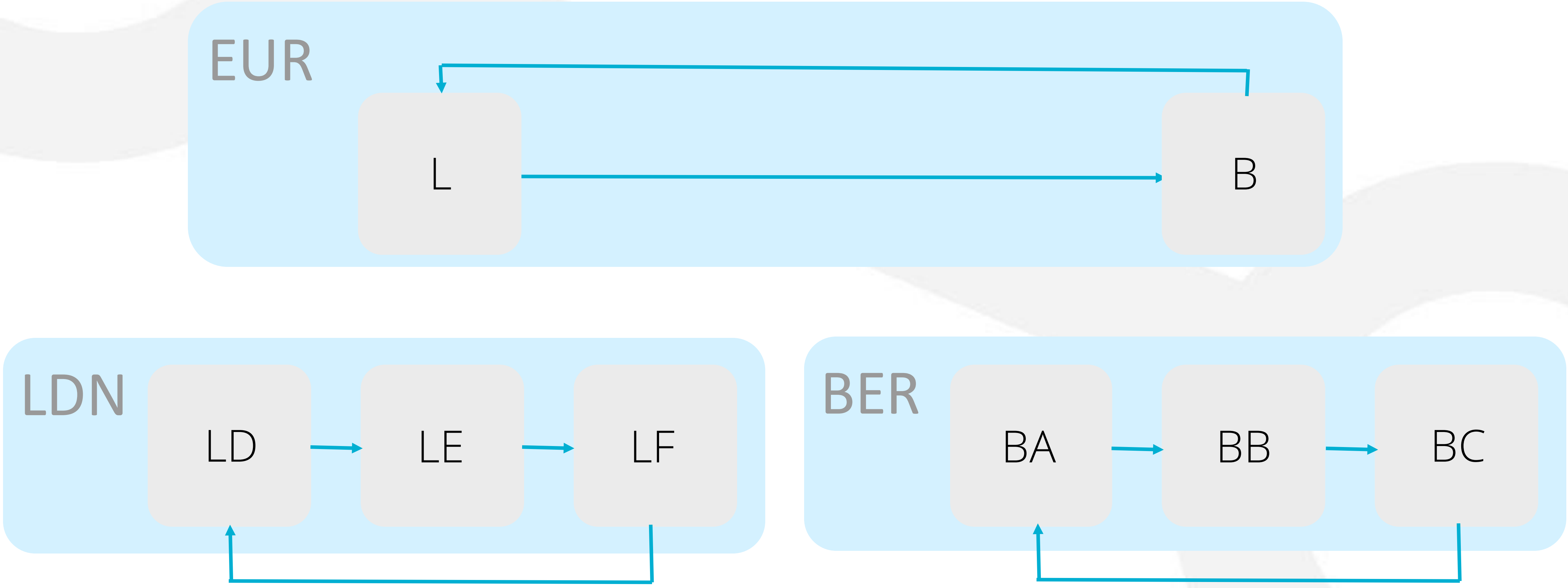
Multiple groups are being run in parallel.
Their periods will not be aligned with each other.
But their open still means “can append” and closed means “finalised”.



D:1	1	E:1	E:2	2	F:1	F:2	F:3	3
A:1	A:2	1	B:1	B:2	B:3	2	C:1	C:2
1	2	G:1	G:2	G:3	3	4	5	H:1

Association introduction

- Group established with approximate locality
- Nodes "London 1, 2, 3" and nodes "Berlin 1, 2, 3"
- New node created, one for "London", one for "Berlin",
- These two new nodes also form a new group called "Europe"
-These also establish a clock
- We have 3 groups, "LDN", "BER" and "EU", advancing separately



Association - introduction

Parent group associates its currently open version with child's highest value

We call this node “parent node” and its group the “parent group”

We call the original group the “child group”

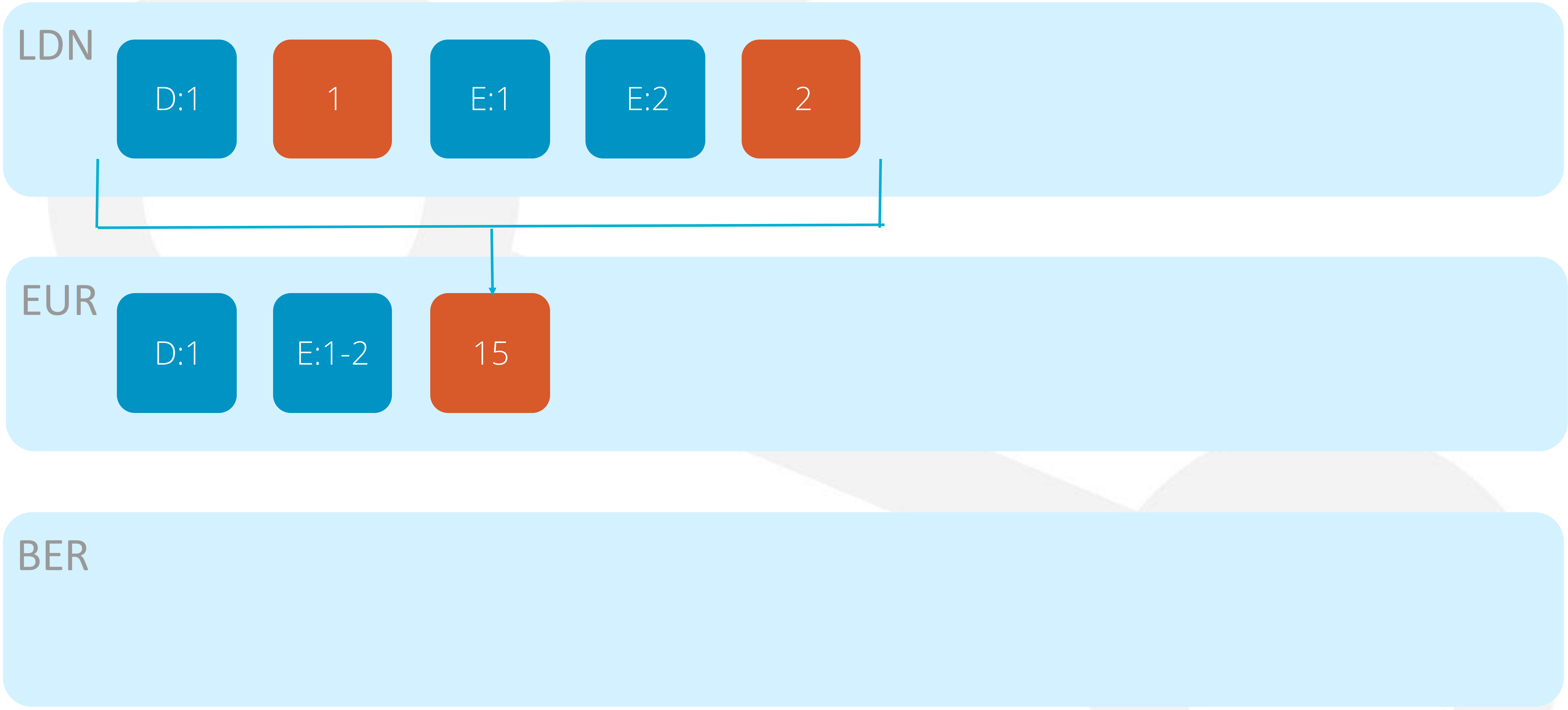
Any node can “request a translation” from the child group to the parent group

“Hey parent node, what does my 5 mean to your group?”

If value (or a higher one) has already been associated (from the child group), it returns that value

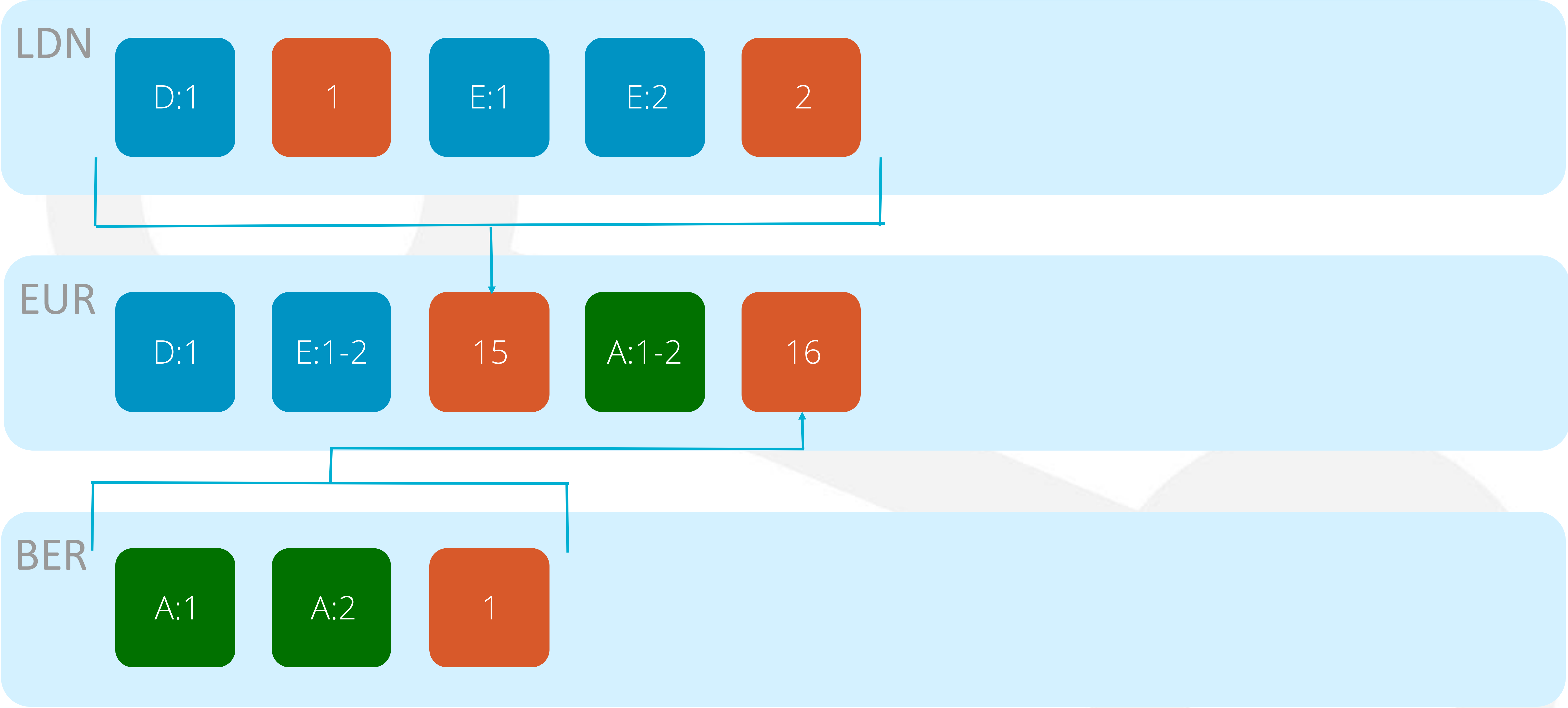
An example association

Two child-groups, {A, B, C} and {D, E, F}
First group associated 15 and 17, second group 16 and 18



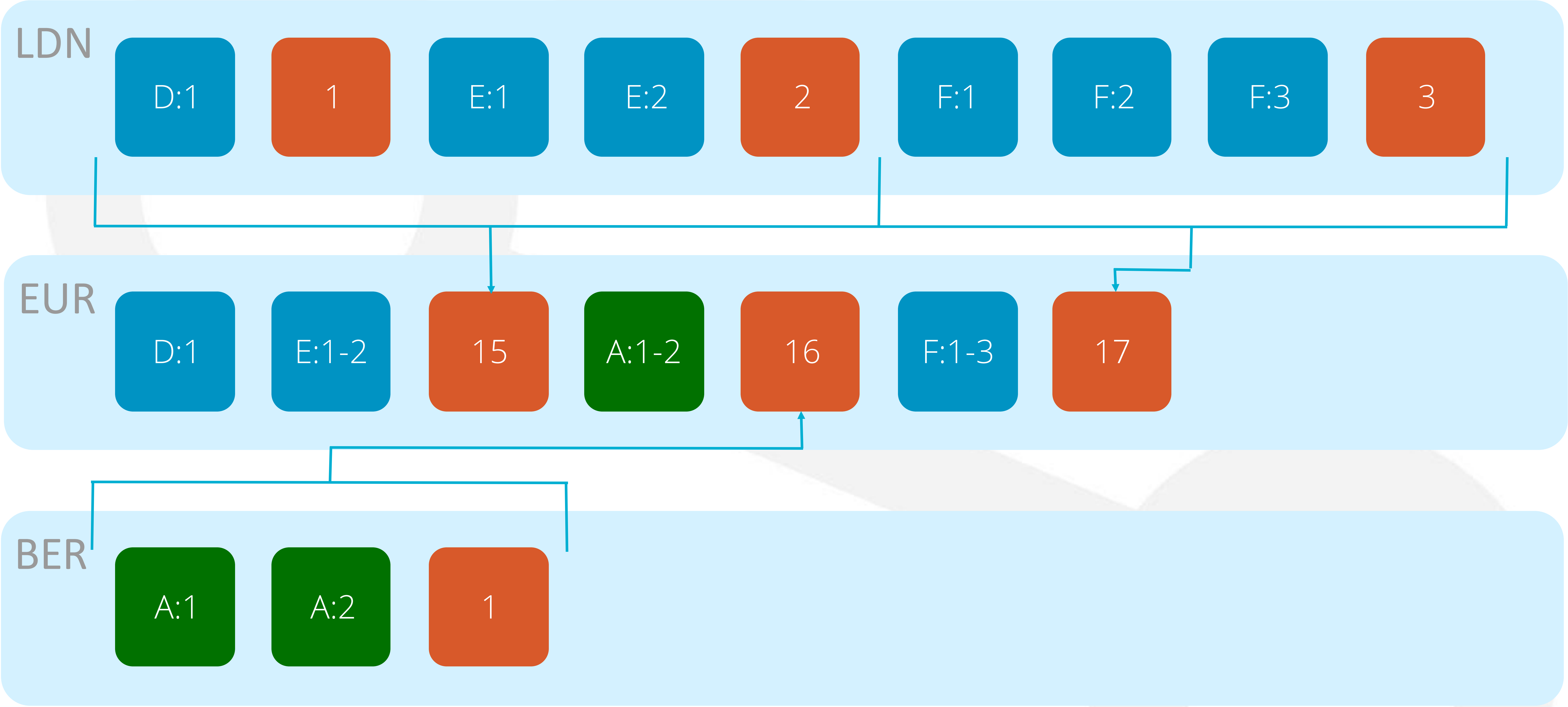
An example association

Two child-groups, {A, B, C} and {D, E, F}
First group associated 15 and 17, second group 16 and 18



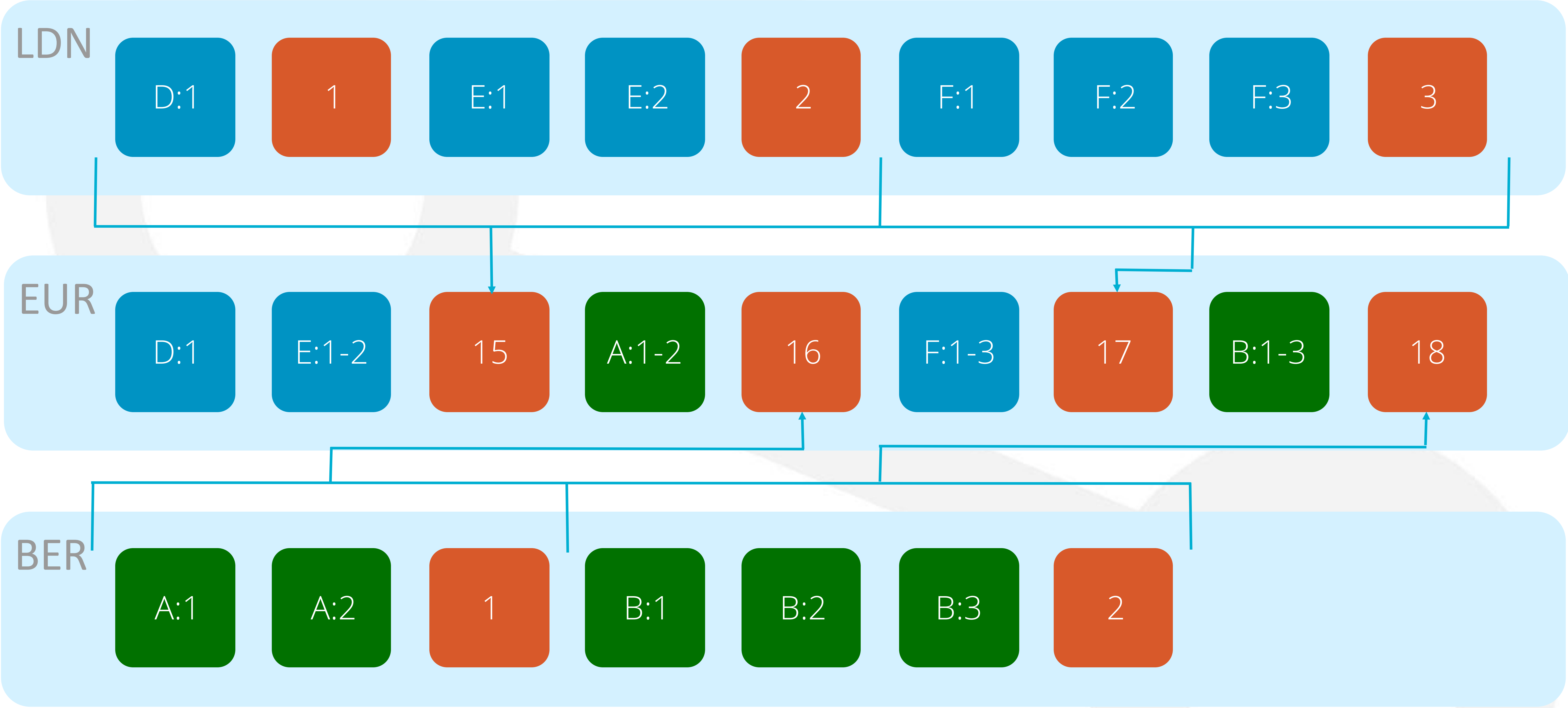
An example association

Two child-groups, {A, B, C} and {D, E, F}
First group associated 15 and 17, second group 16 and 18



An example association

Two child-groups, {A, B, C} and {D, E, F}
First group associated 15 and 17, second group 16 and 18



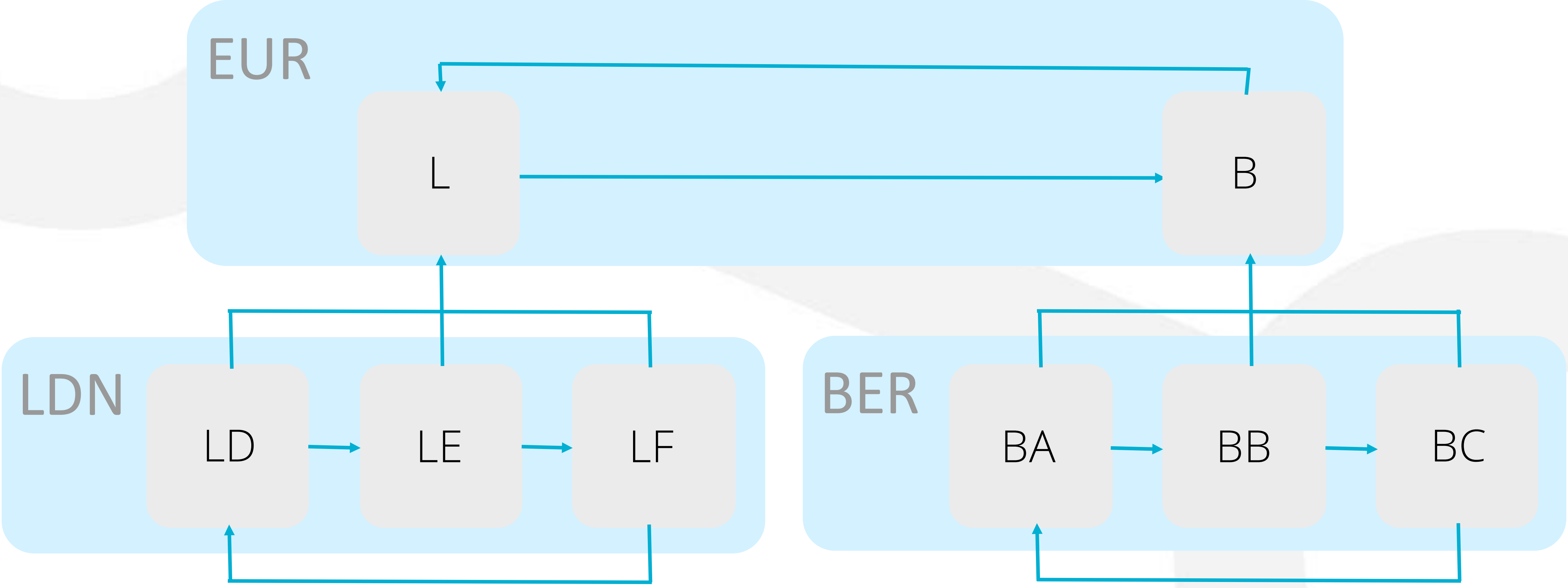
LDN
D:1 1 E:1 E:2 2 F:1 F:2 F:3 3

EUR
D:1 E:1-2 15 A:1-2 16 F:1-3 17 B:1-3 18

BER
A:1 A:2 1 B:1 B:2 B:3 2

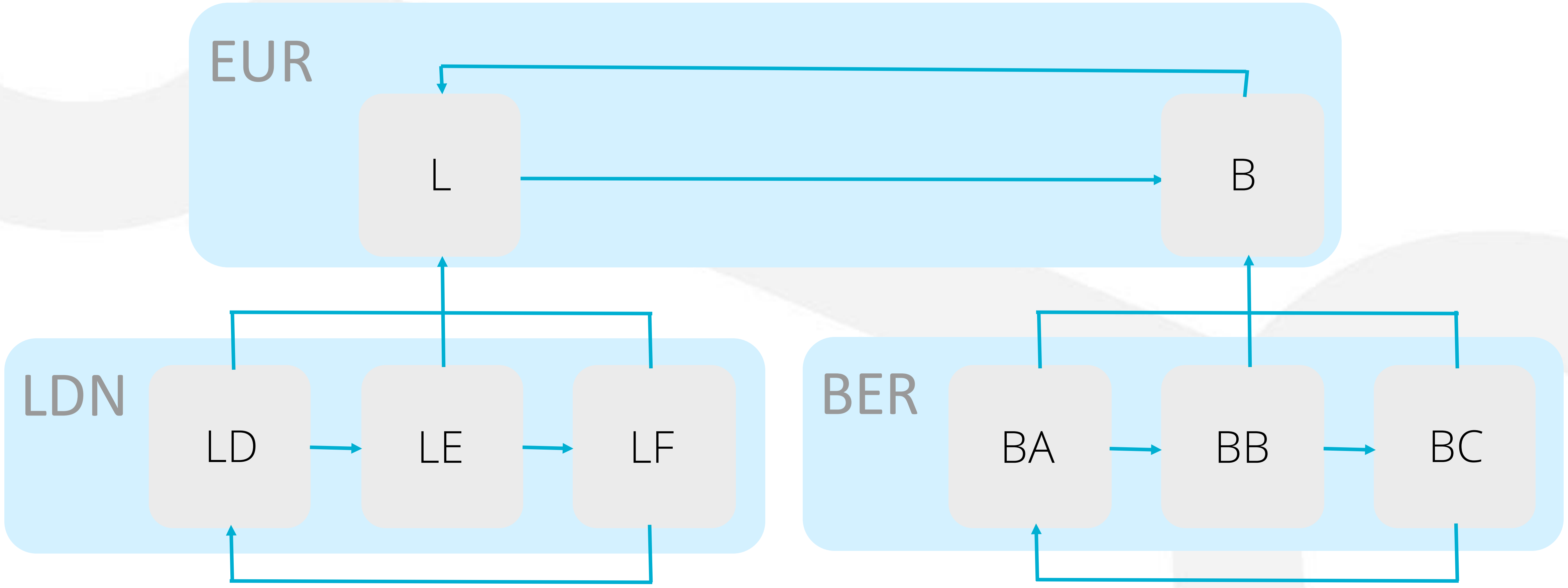
Association - continued I

In impl, only closed versions are queried, so that data is already final (simple).
Parent's open version is associated with child's (highest queried) version, so it's associated with the mutable version and they are finalised at next token receive.
Parent's version-closure is communicated to the child-group, so that it knows what version is reliable on parent's level.



Association - continued II

The parent's response to the child's query is immutable, so it can be cached or communicated.
It can be used to compare versions from other groups, since they also form
a single order of events.
Lowest common parent-group exists between any two groups.



Typical latencies - how close are we to global time?

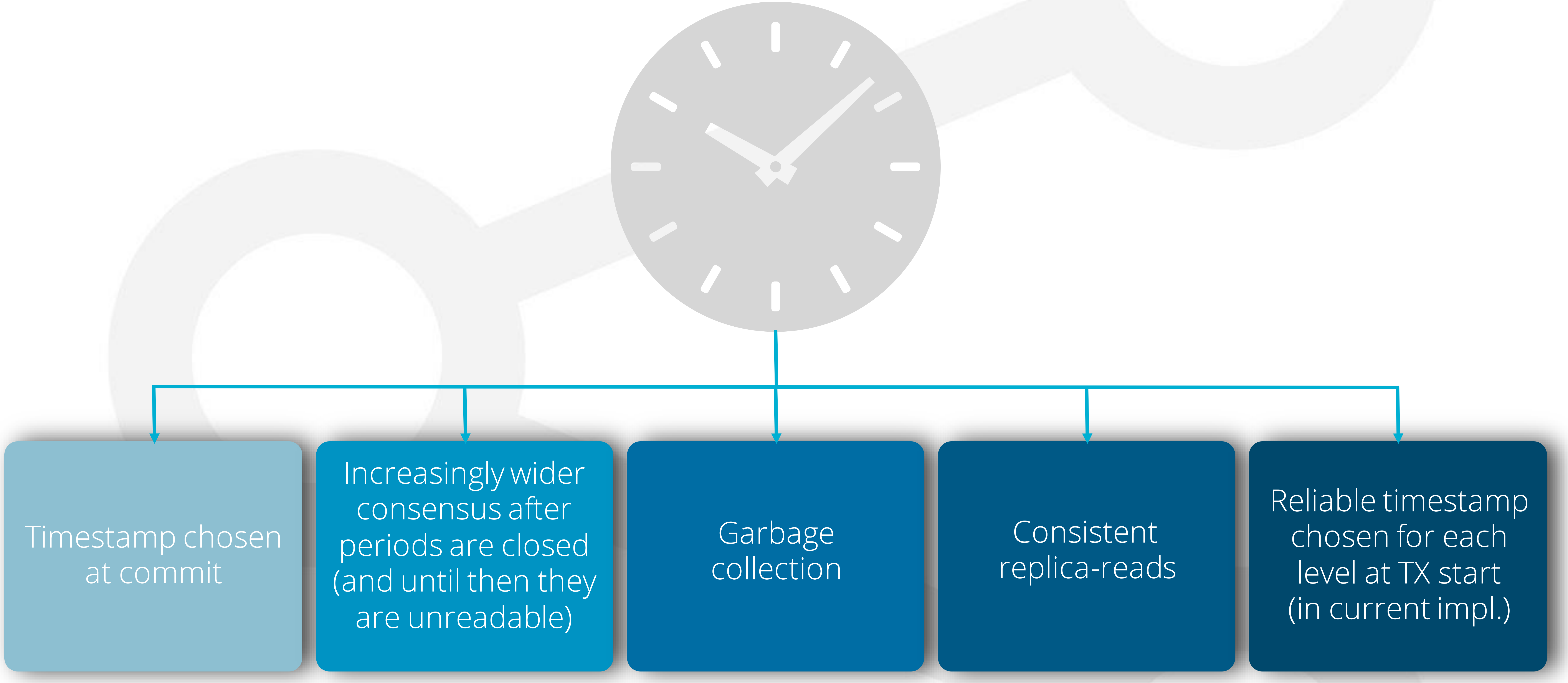
No single node needs to be distributed beyond geo-location (LDN).
Groups higher in hierarchy would be more spread out.

Typical latencies:

- < 1ms — Same data-centre
- < 5 ms — Same geo-location (London)
- < 20 ms — Same region (London-Paris-Amsterdam)
- < 50 ms — Same continent (Moscow-London-Rome)
- < 250 ms — Worldwide

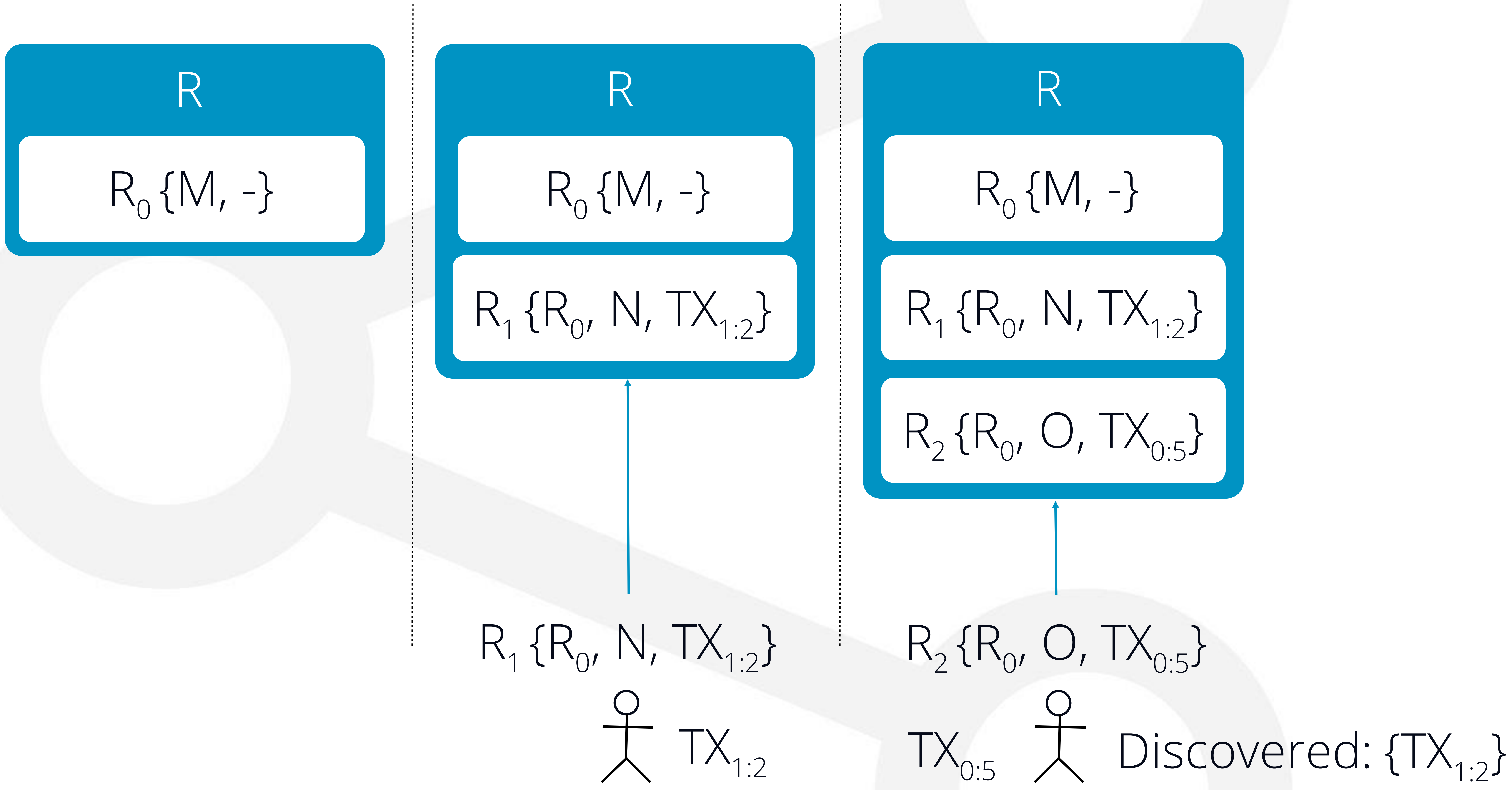
We presume that most data will be managed locally, where this setup typically progresses quicker than 6ms.

Clocks in DIANEMODB



- Timestamp chosen at commit
- Increasingly wider consensus after periods are closed (and until then they are unreadable)
- Garbage collection
- Consistent replica-reads
- Reliable timestamp chosen for each level at TX start (in current impl.)

TX commit - discovery



R

R_0 {M, -}

R

R_0 {M, -}

R_1 {R_0, N, $TX_{1:2}$}

R

R_0 {M, -}

R_1 {R_0, N, $TX_{1:2}$}

R_2 {R_0, O, $TX_{0:5}$}

R_1 {R_0, N, $TX_{1:2}$}

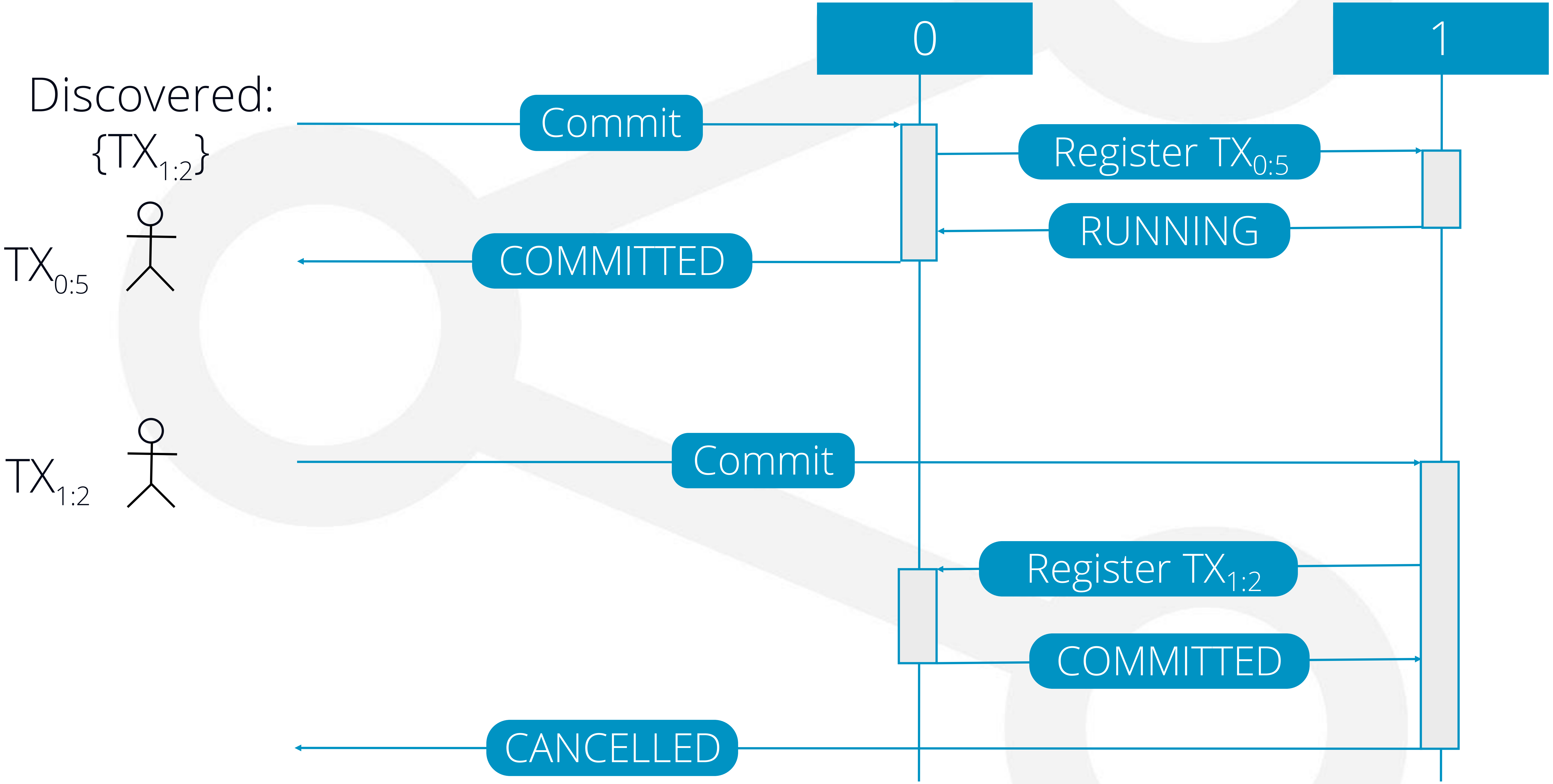
$TX_{1:2}$

R_2 {R_0, O, $TX_{0:5}$}

$TX_{0:5}$

Discovered: {$TX_{1:2}$}

TX commit - registration



TX commit - summary

If a tx updates the same record as another one, saves it in its discovered list

Independent transactions never wait for each other

If a tx starts its commit, it makes its own registered list immutable, and for each $tx in its discovered list

Click to add text

- queries the state of $tx
- tries to write itself into $tx's registered list (via the query message)

A tx can commit (and is rolled back otherwise) if

- each tx in its registered list was cancelled
- it successfully wrote itself into each tx's registered list in its discovered list

TX commit - remarks, edge cases

The transactions which start their commits don’t accept any more to register, the ones that come later are cancelled

If a transaction A registered into B successfully, B has to wait for A to either succeed or fail before progressing with its commit

It can happen that both transactions are cancelled, if they both start their commits at the same time (on different nodes)

TX commit - closing thoughts

Isolation levels established by choosing the right version

Each consensus group progresses independently, essentially a mesh of nodes

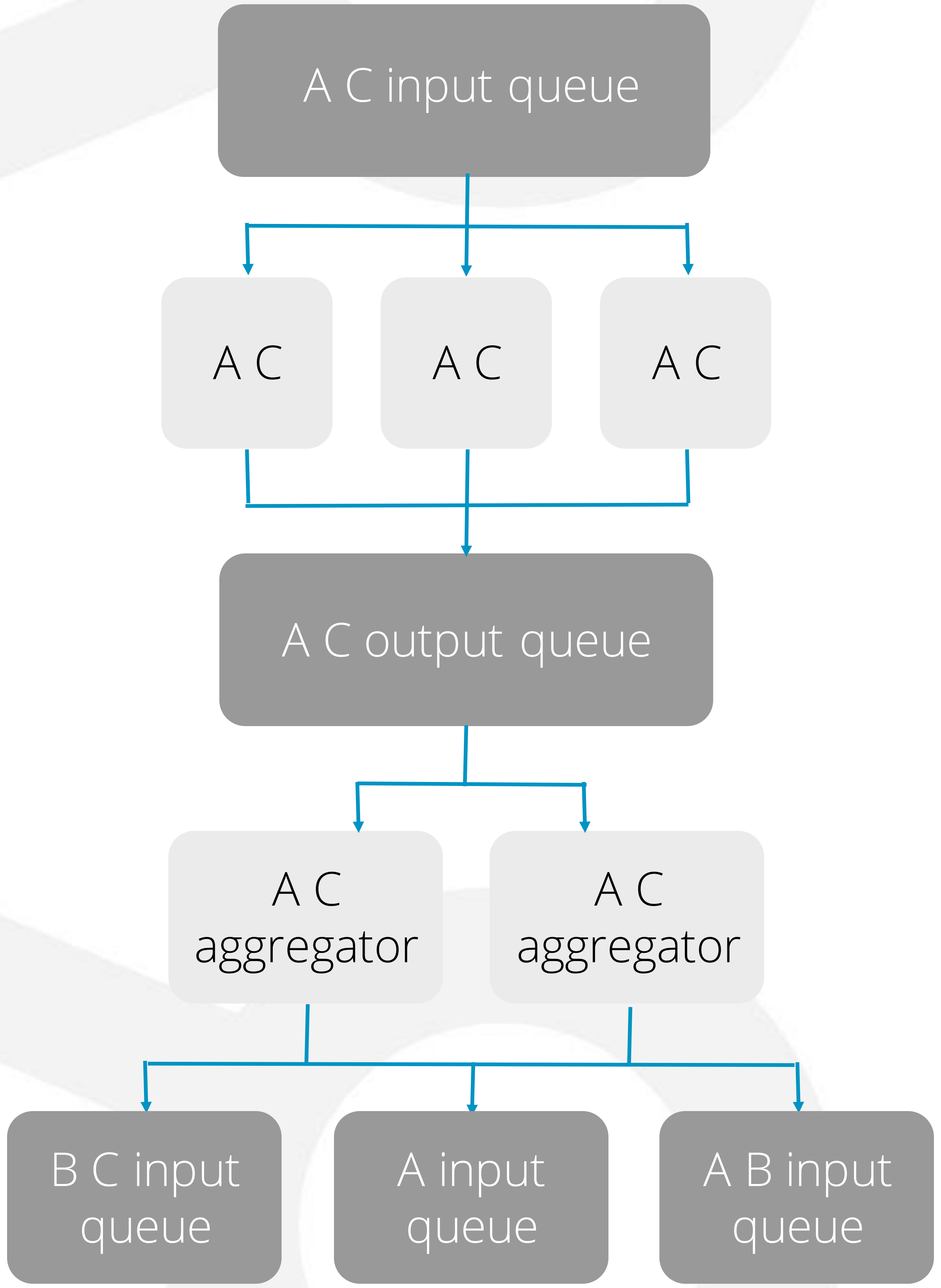
Single-thread nodes only, all internal state is observable from inputs, not just data

Consequences - quick, simple, stateless

- Time definition is strict (no tolerance needed), but still very close to temporal ordering
- Single datacenter “temporal fit” will be less than a few ms (with a group of sensible size)
- Same inputs = Same outputs
- Replication is streaming inputs
- Parallel execution

Aggregators, deployment, failures

- Tolerating failures via redundancies
- New instances can be added to each level
- No elections, no timeouts
- Gain reliability for hardware resources
- Nodes publish idempotent messages
- With multiple redundancies, can tolerate even byzantine-failures



Communication, redundancy

Can have multiple processors consuming from the same consensus group

Can find faulty instances, which disagree with the majority before sending out the request (Byzantine fault tolerance)

Follower reads do not need to be stored in event log

No consensus beyond event-logs

Writes always go to master record, from which changes are communicated to followers

Replicated, reliable, highly available consistency is expensive because it's complex

Consensus is complex

Responsive consensus gets increasingly harder the further away you go on a network.

Original RAFT paper proposes election timeouts between 150ms and 300ms (congestion eats this up quickly)

Efficiency dropoff after this range is sudden

Complexity in practice

People use SQL or no transactions

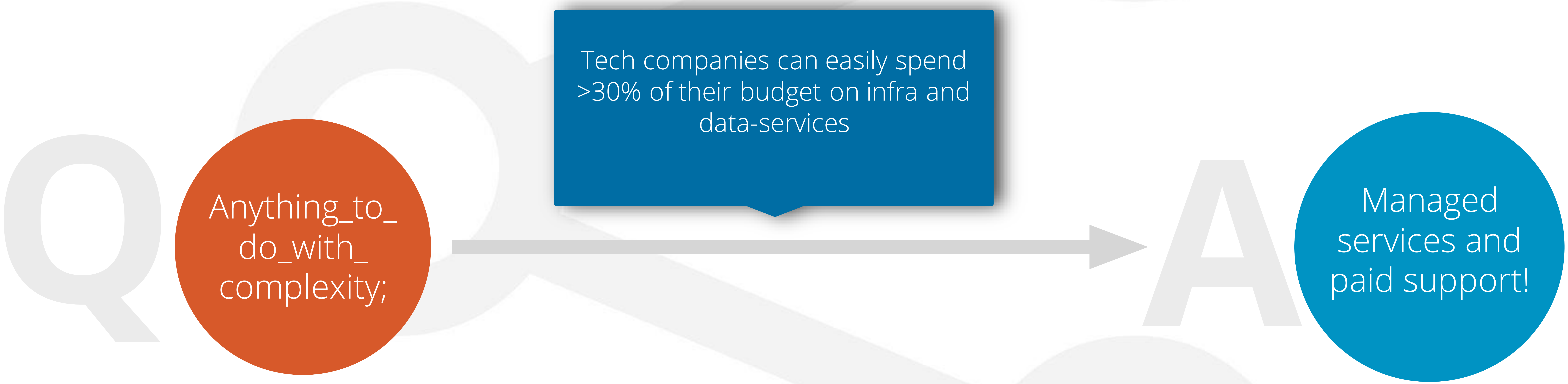
Industry hardly ever uses anything beyond “READ COMMITTED”

People don’t want to care about their DB internals

Ideally, a system which _tells them_ what happened instead of reconstructing from logs

“Causal consistency with no stale reads” vs “REPEATABLE READ”

What do we do now?



Anything_to_do_with_complexity;

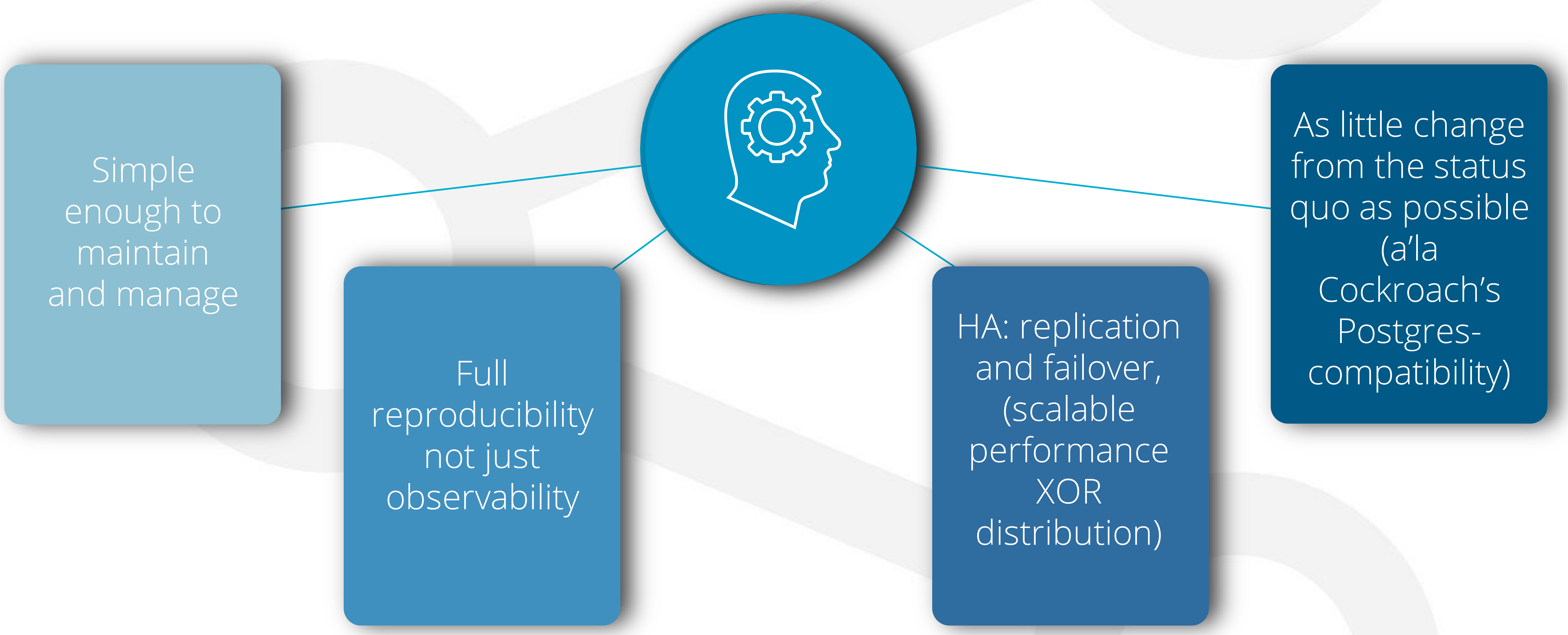
Tech companies can easily spend >30% of their budget on infra and data-services

Managed services and paid support!

What do we do now?

Others managing your data is not always an option,
it just becomes someone else’s problem, no easy fix for complex problems
Network (esp. inter-datacenter) and disk IO is extra
Typically single datacenter to minimise complexity
Complexity still needs to be “internalized” to a large extent
If you run your own, you can't replicate your PROD instance very well

So, what do we think – people would care about?



Simple enough to maintain and manage

Full reproducibility not just observability

HA: replication and failover, (scalable performance XOR distribution)

As little change from the status quo as possible (a'la Cockroach's Postgres-compatibility)

Summary – simpler, cheaper to develop for

- Much easier to reason about
- SQL-compatible isolation, currently SNAPSHOT
- Difference is deciding which region record and index lives in
- Developing tests is as easy as listing inputs

What we solve summary

Strong Consistency

High Availability

Consistent (Geo-)
Replication

Byzantine Fault
tolerance
(like CPU errors)

DIANEMODB – Products

Looking for strategic investor

Currently

Existing Software and implementation consulting

Shortly

JDBC compatible database

Plans

SQL impl, retrofitting transactionality on multiple schemas and datastores

Eventually

A service, which can join independent clusters into a single tx unit, a single global SQL DB

Thanks
for watching

DIANEMODB
RELIABLE GLOBAL SCALABILITY

András Gerlits [CTO]
andras.gerlits@dianemodb.com | www.dianemodb.com