Multimaster isn't magic

How - and when - to use multi-master replication and BDR

Craig Ringer - BDR and PostgreSQL developer - 2ndQuadrant Inc. (www.2ndQuadrant.com)

Who am I?

- Co-lead developer on the BDR project at 2ndQuadrant
- Active in PostgreSQL community for 10 years
- Involved in replication-related core PostgreSQL development
- Interested in usability and new-user experience in PostgreSQL
- Sandgroper and Kiwi expat

About 2ndQuadrant

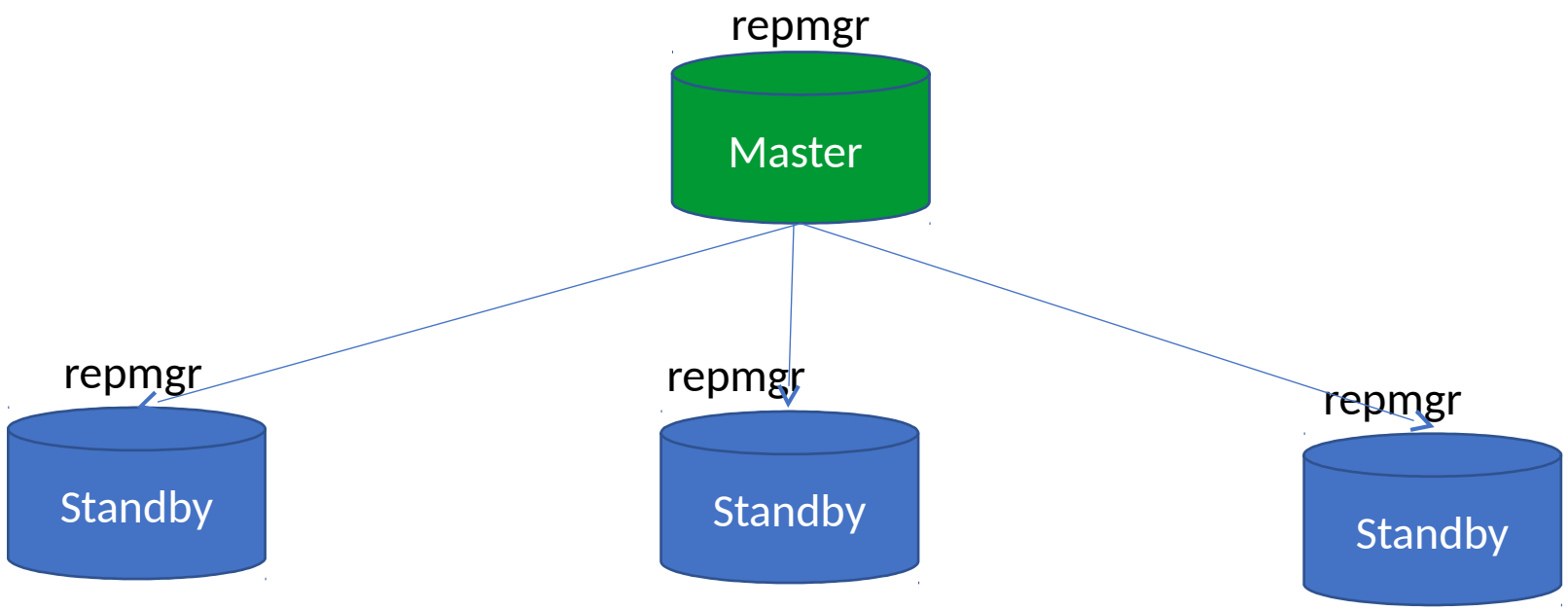
- Founded by Database Architect who saw the need to implement Enterprise features in Postgres
  - Backup and Restore
  - Point in Time Recovery
  - Streaming Replication
  - Logical Replication
- Funding through support of PostgreSQL servers
- We wrote the code, we wrote the books
- 4 members of PostgreSQL Security team
- Platinum Sponsor of PostgreSQL project
- Over 15 project contributors who do support

Agenda

- What the term “multi-master” means to different people
- The problems different multi-master solutions solve
- Compromises inherent in multi-master systems
- Loosely coupled replication based multi-master (like BDR) vs. other solutions
- Implementation factors for multi-master

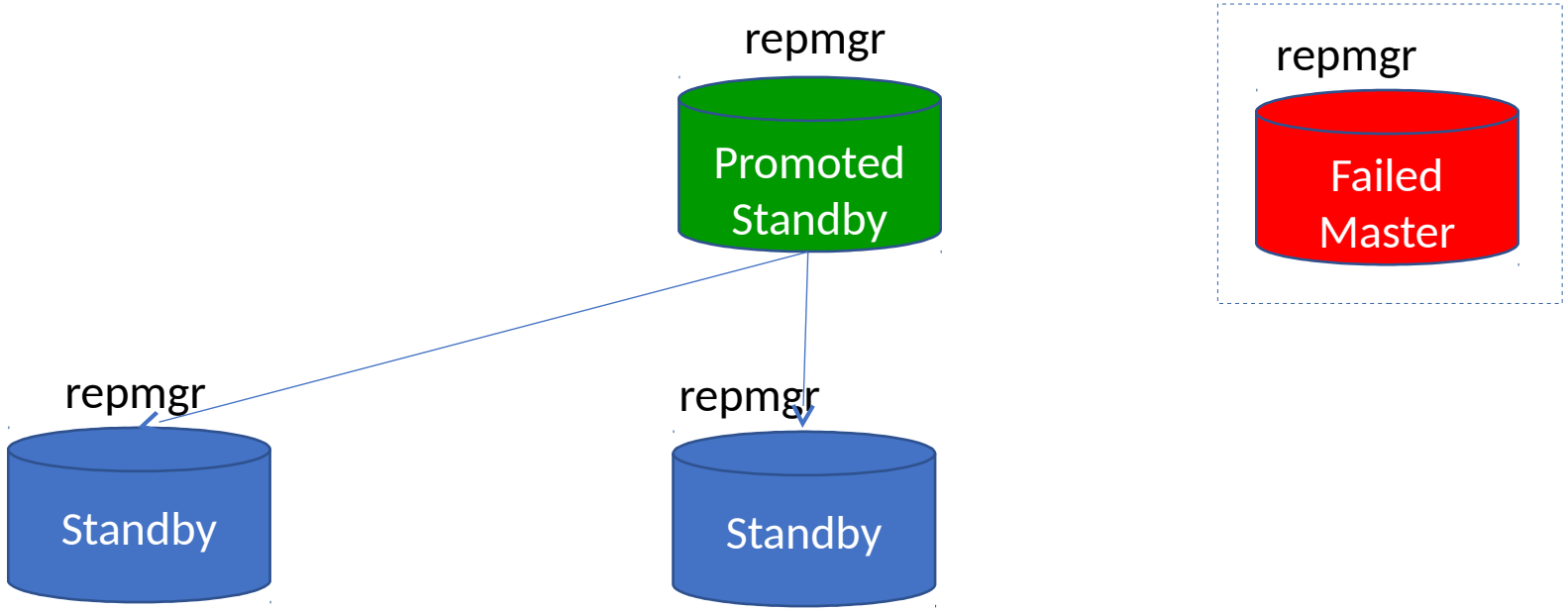
Master/Standby Solutions for HA

Read Traffic to Master and Replicas and write traffic to Master

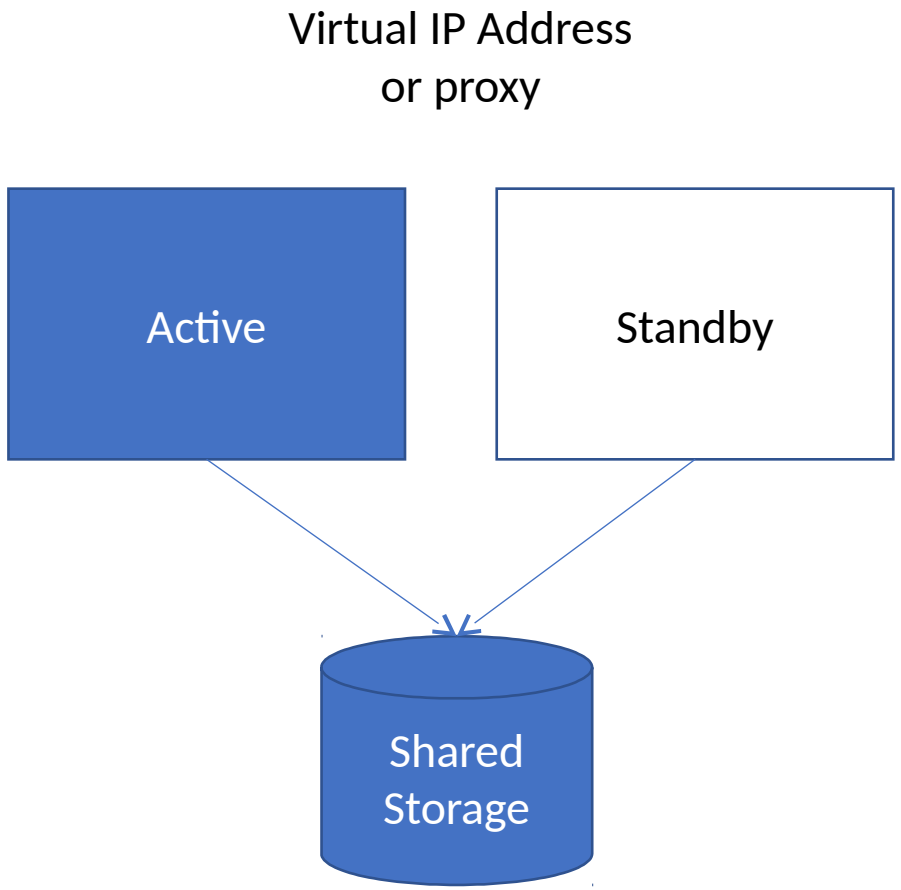


Master/Standby Solutions for HA

Promotion is used to replace a failed master

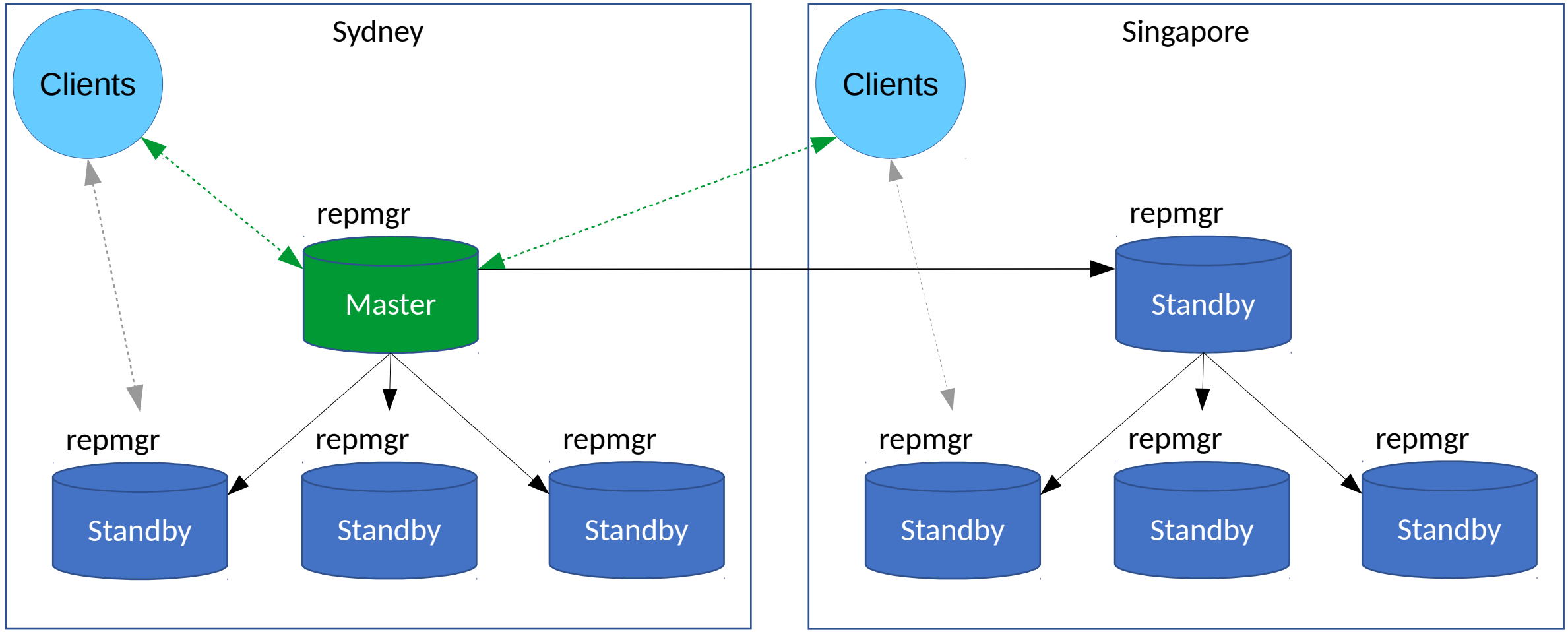


Another Availability PostgreSQL Solution



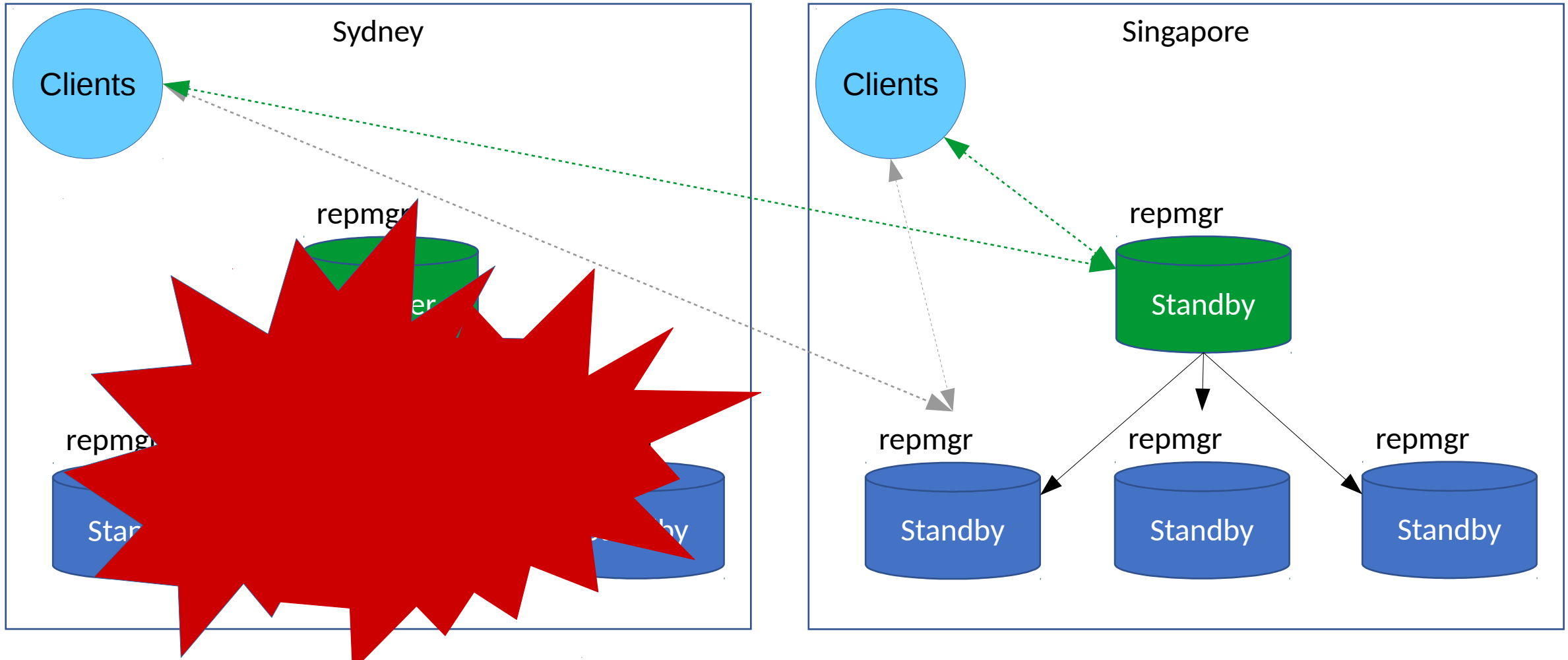
Multi-Site Master/Standby

Read Traffic to Masters and Replicas and write traffic to Master



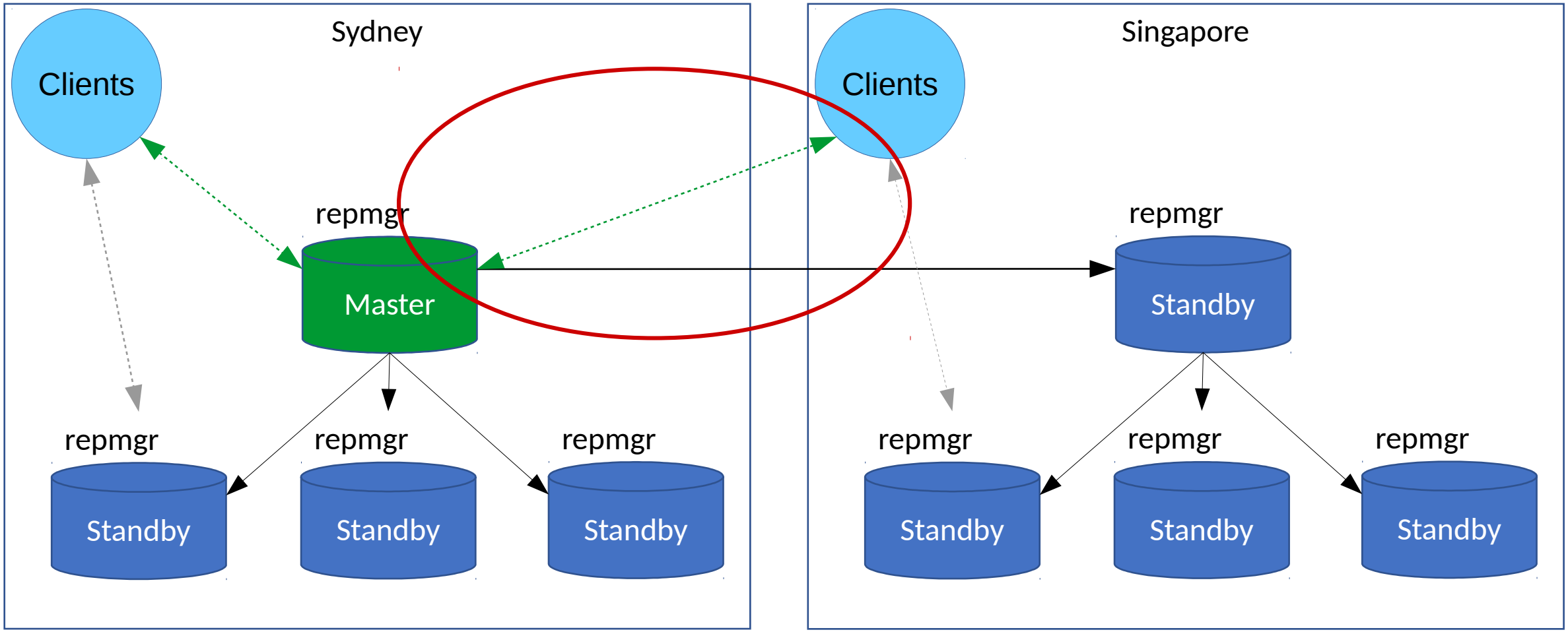
Multi-Site Master/Standby

Disaster recovery failover by promoting another site's master



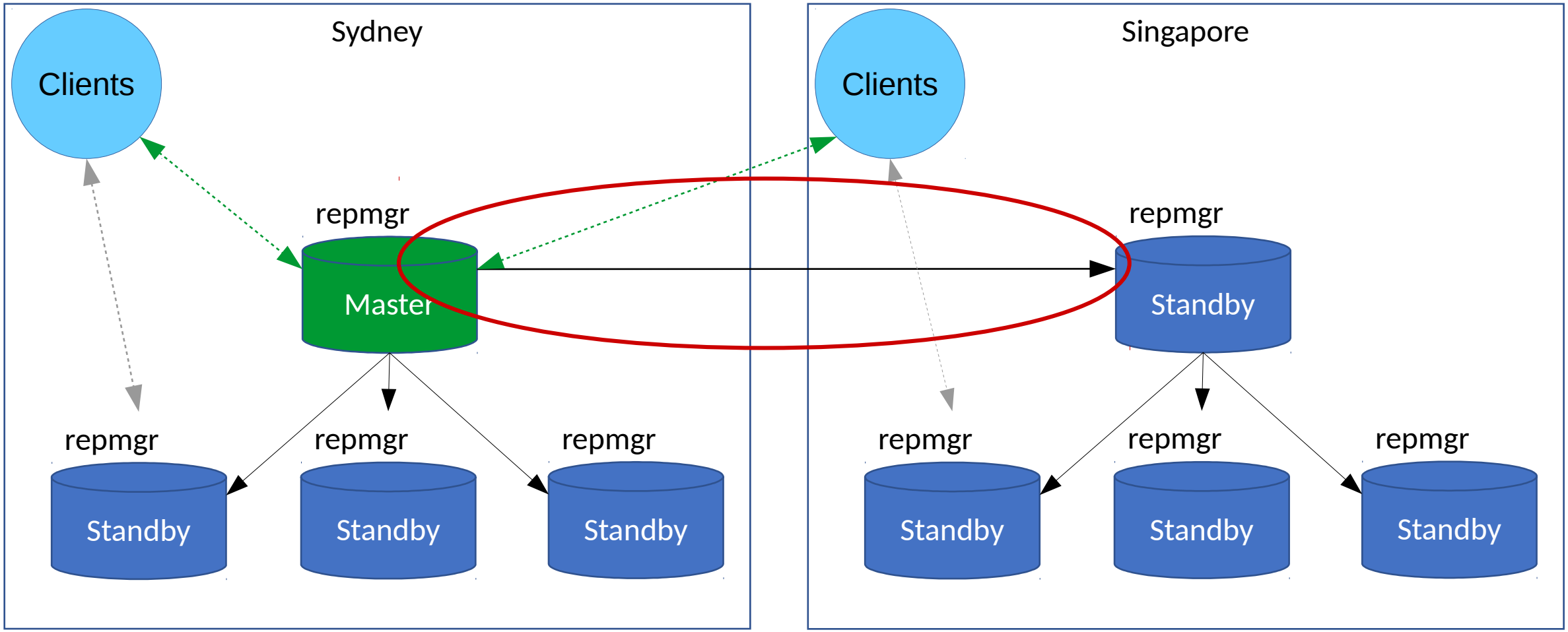
Problem: Write Query Latency

Read/write queries can have limited bandwidth and high latency



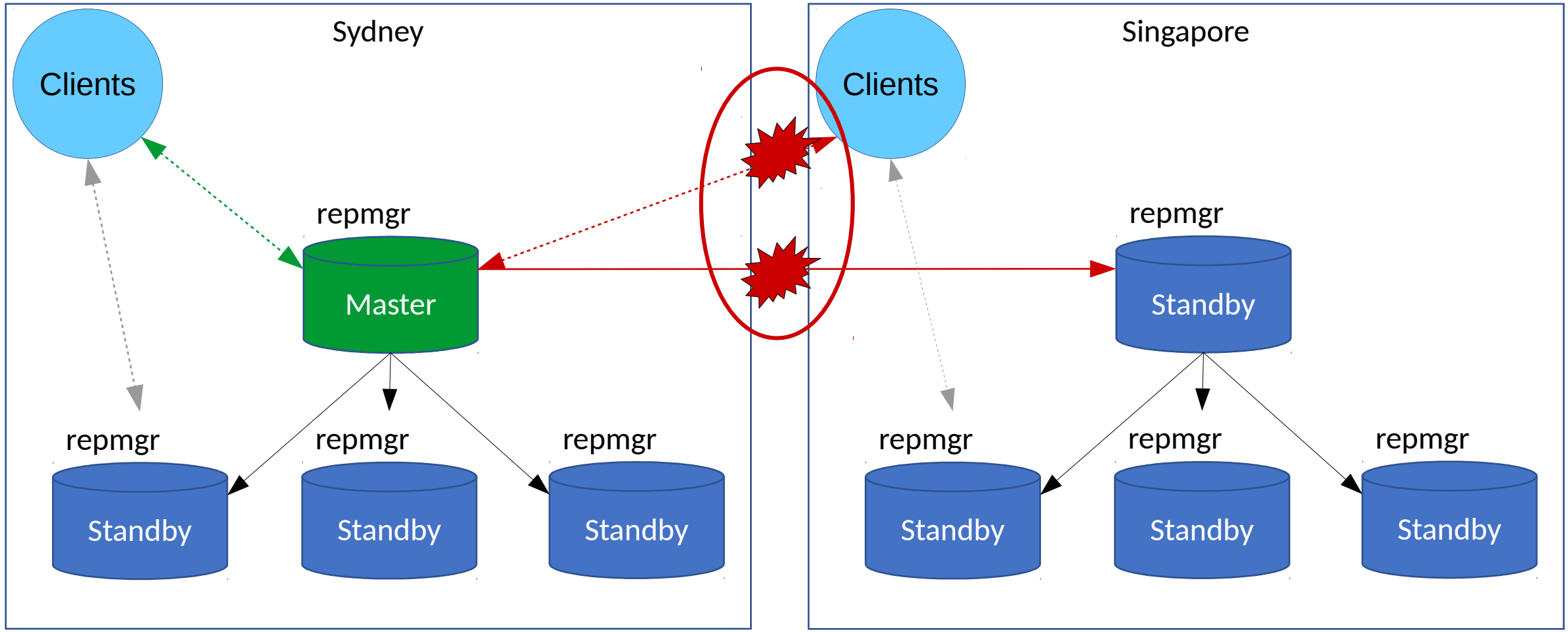
Problem: Write Visibility Delay

It can take a while for writes on the master to be visible on secondary site standbys



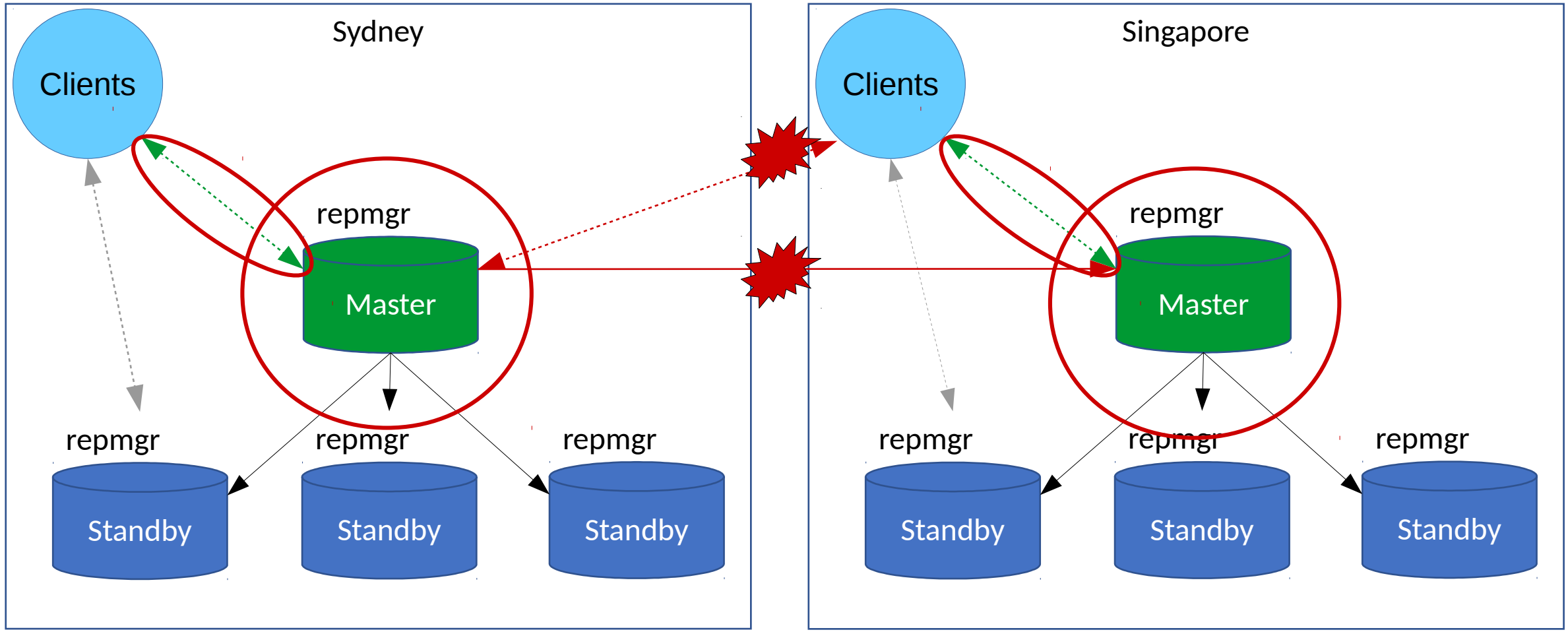
Problem: Site Write Outage

Clients at secondary sites lose all write access if WAN is disrupted



Problem: Split Brain / Divergence

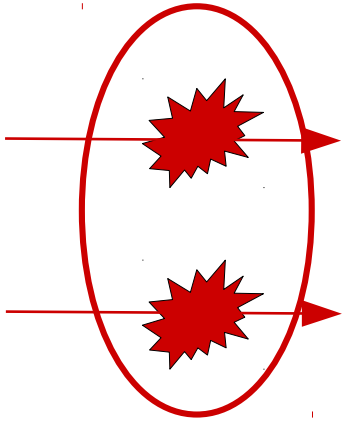
Data can diverge if promotion without STONITH and fencing leads to two active masters



Problem: Split Brain / Divergence

Data can diverge if promotion without STONITH and fencing leads to two active masters

If the WAN is down....



- How do you tell the other clients about the new master?
- How do you tell the old master to reject writes because it’s been replaced?

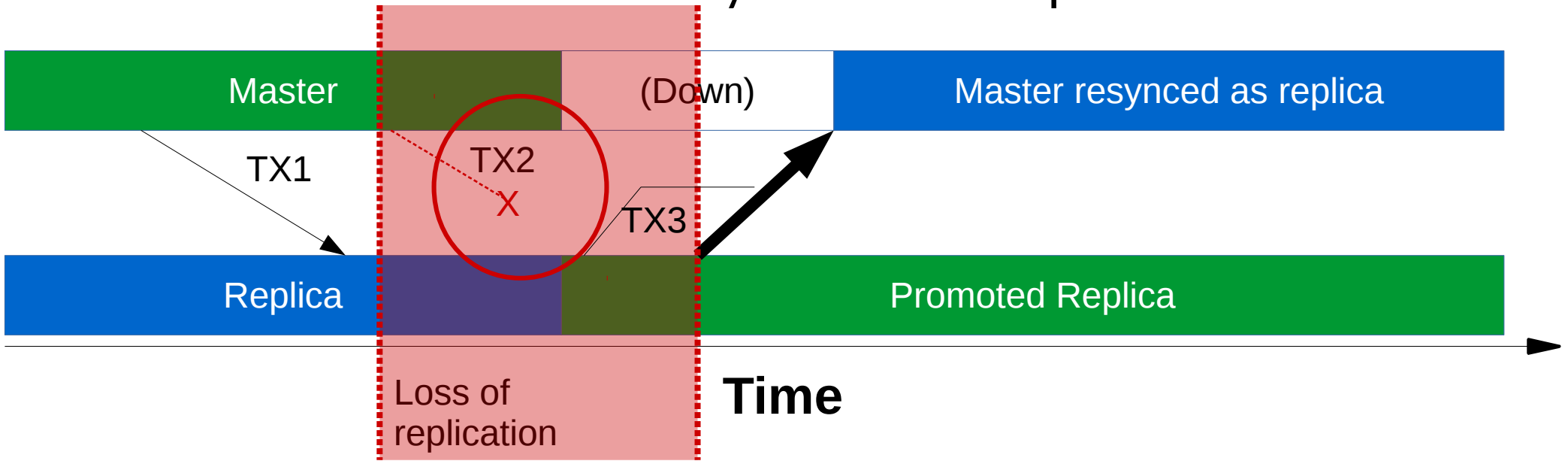
Problem: Split Brain / Divergence

- Workarounds include:
  - Tolerating a SPoF in the form of a proxy or virtual IP that all clients must connect through
  - DNS-based or routing-protocol-based connection redirection (unreliable for fencing, no STONITH)
  - Using a sideband to enforce STONITH/fencing

    "Sarah called and said to shut down Turkey1, she's promoting Turkey2 now"
  - Scripted repair processes to resync after a split

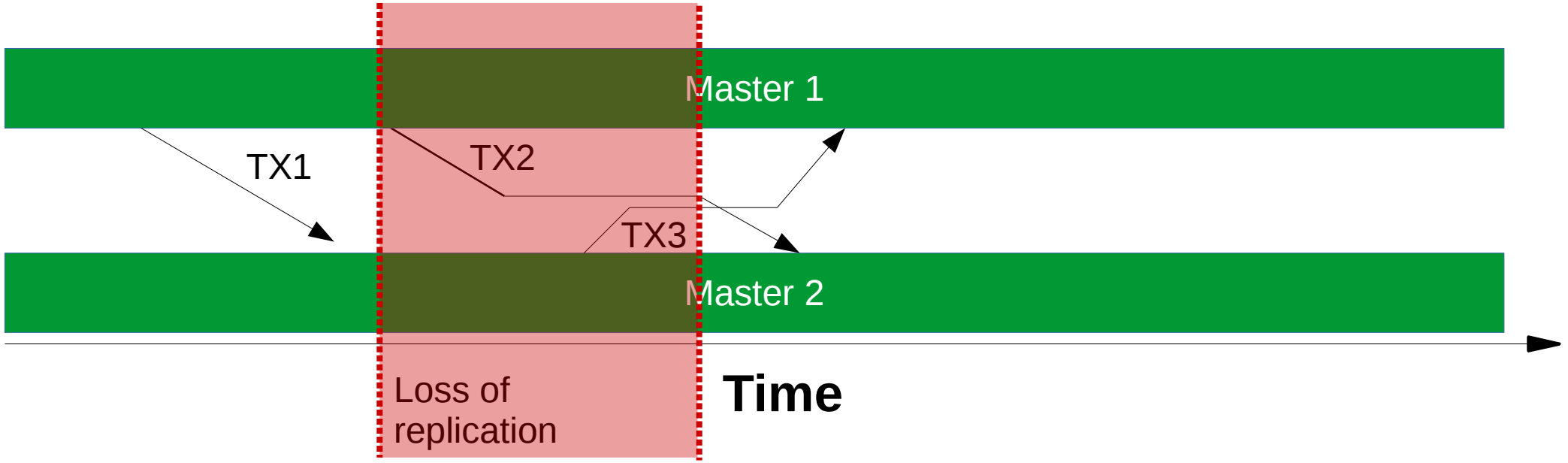
Problem: Split Brain / Divergence

- Master and replica have usually already diverged before a failover decision is made
- Some resynchronisation is needed to prevent data loss from old master: TX2 would be lost with asynchronous replication



Multi-master solves everything!

- Master and replica divergence resolves once network returns
- No manual repair or resynchronisation required!



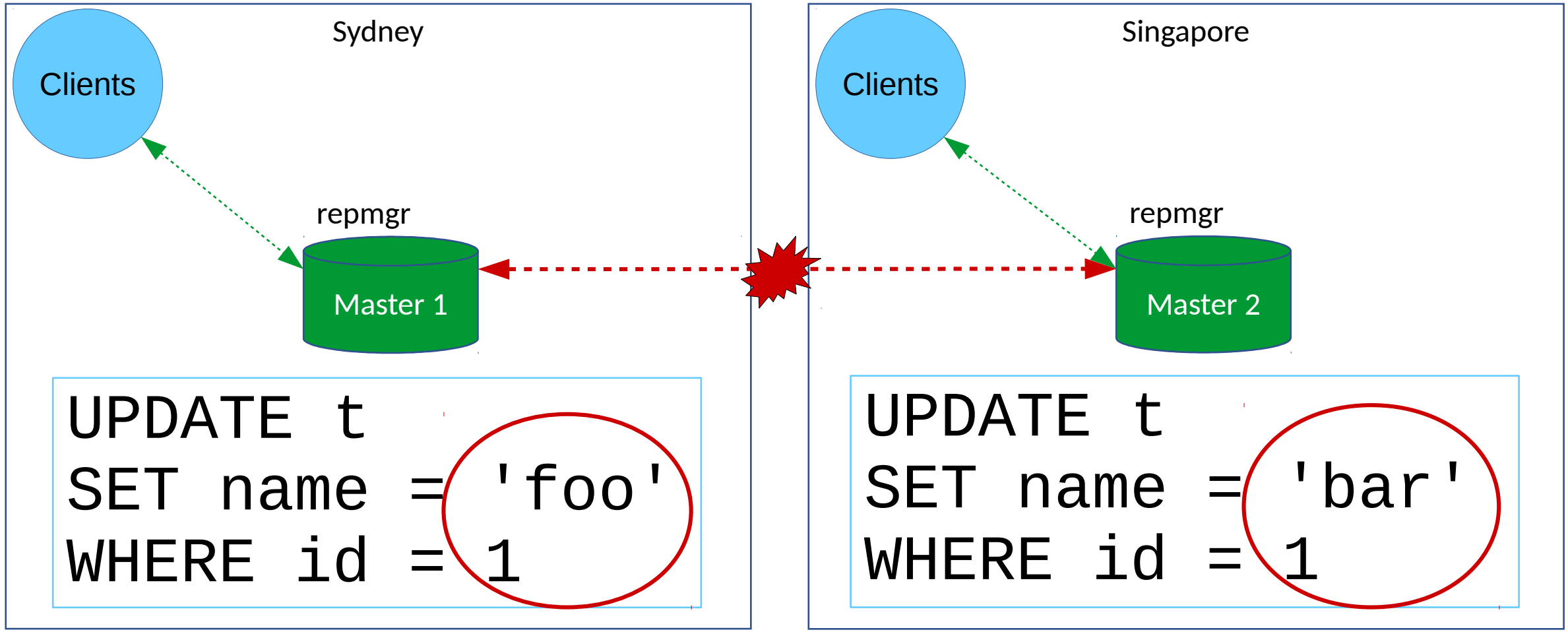
Multi-master solves everything!

- Low latency for write queries on satellite sites
- No more write replication delays for satellite site writes
- Satellite sites remain writable while disconnected
- Data divergence resolves when connectivity is regained
- You don’t need to manage failover, STONITH and fencing anymore

Image credit: Rainbow Unicorn Kitty by spazoidbunny on DeviantArt

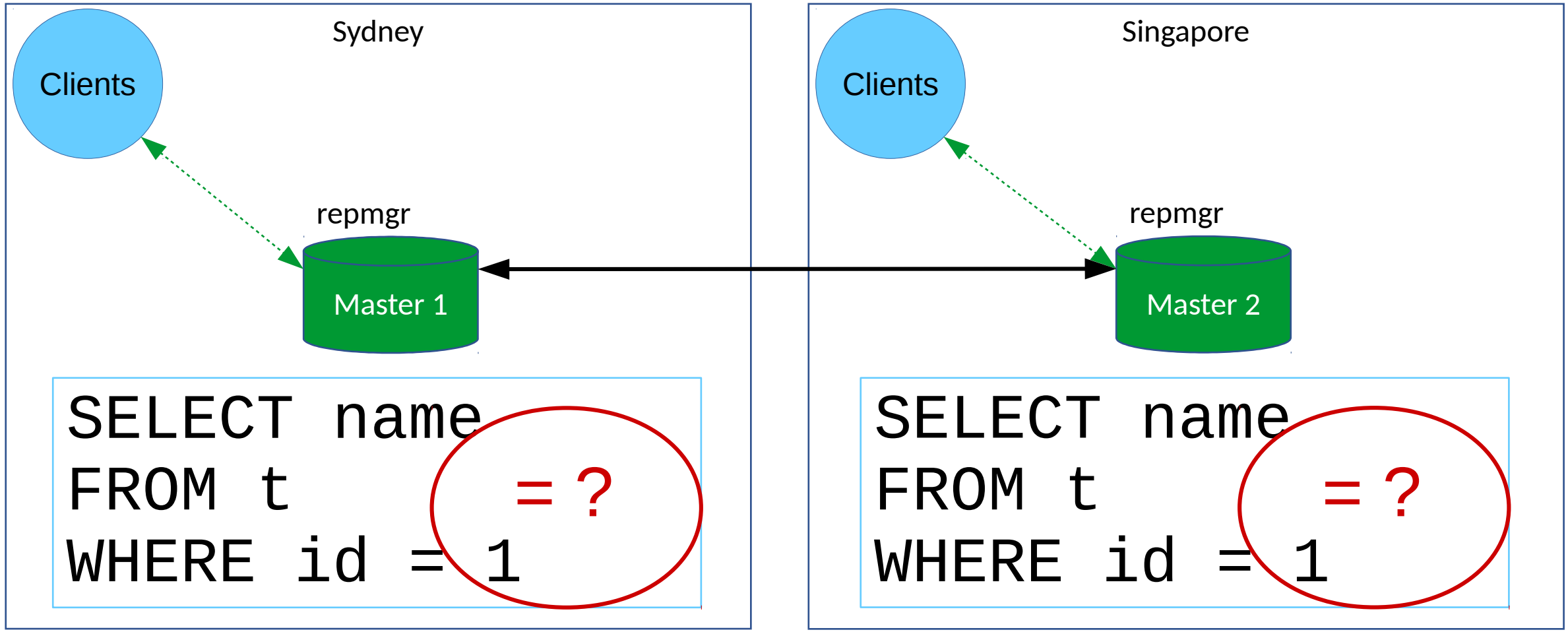
Not so fast!

Allowing modifiations while disconnected has consequences



Not so fast!

Allowing modifiations while disconnected has consequences



Multi-master solves everything?

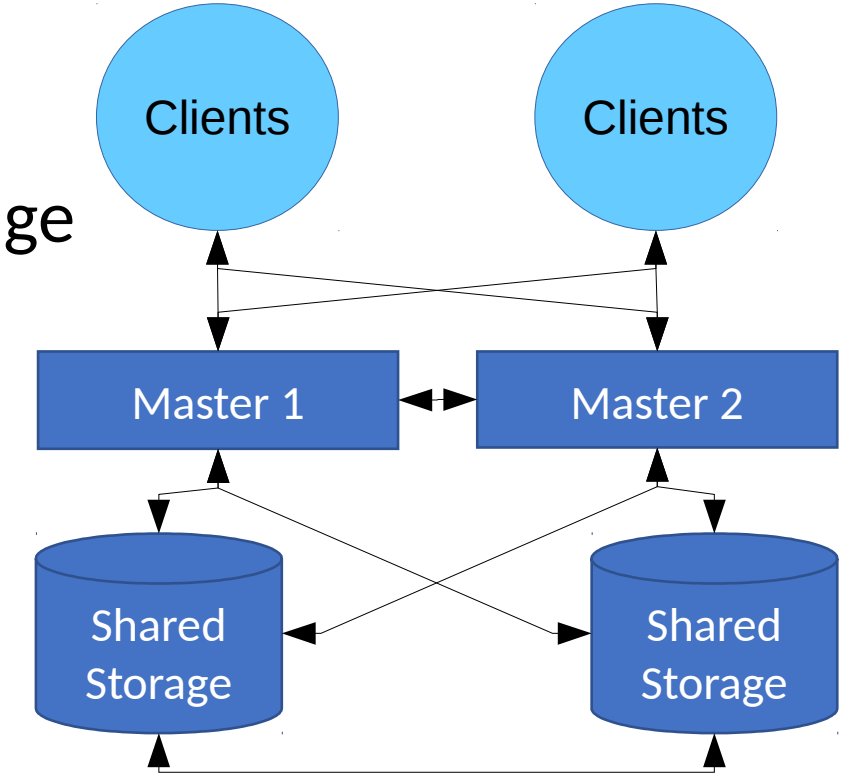
- What happens if you update rows on multiple nodes at the same time?
- How do you generate globally unique synthetic keys?
- Does it matter that writes can be visible on some nodes before others?
- How do you maintain foreign keys when one node can be adding a child row while another removes the parent row?
- Can apps trust that the ACID semantics their authors relied on will still exist?

Multi-master is just another tool

- Multi-master solves some problems but requires additional design considerations
- There are many *kinds* of multi-master, with different trade-offs
- Don’t deploy MM because of buzzword hype
- Deploy it if it solves *your problems* or enhances your user experiences
- Requirements analysis is crucial
- Understand the tools

Kinds of multi-master

- Tightly coupled *vs* loosely coupled
- Shared storage *vs* independent (replicated) storage
- Synchronous *vs* asynchronous
- Always-consistent vs eventually-consistent
- Conflict prevention vs conflict resolution
- Numerous hybrids and variants in each category
- Every model has different trade-offs

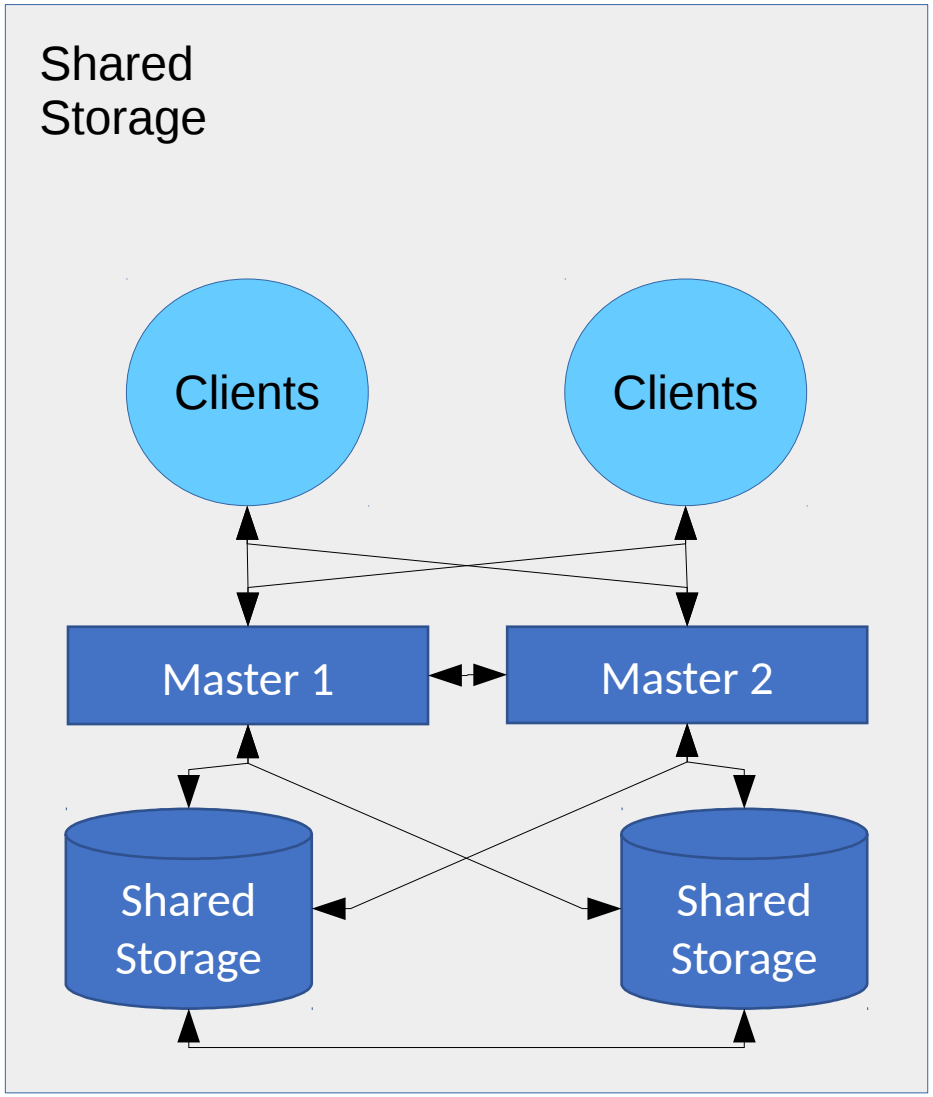


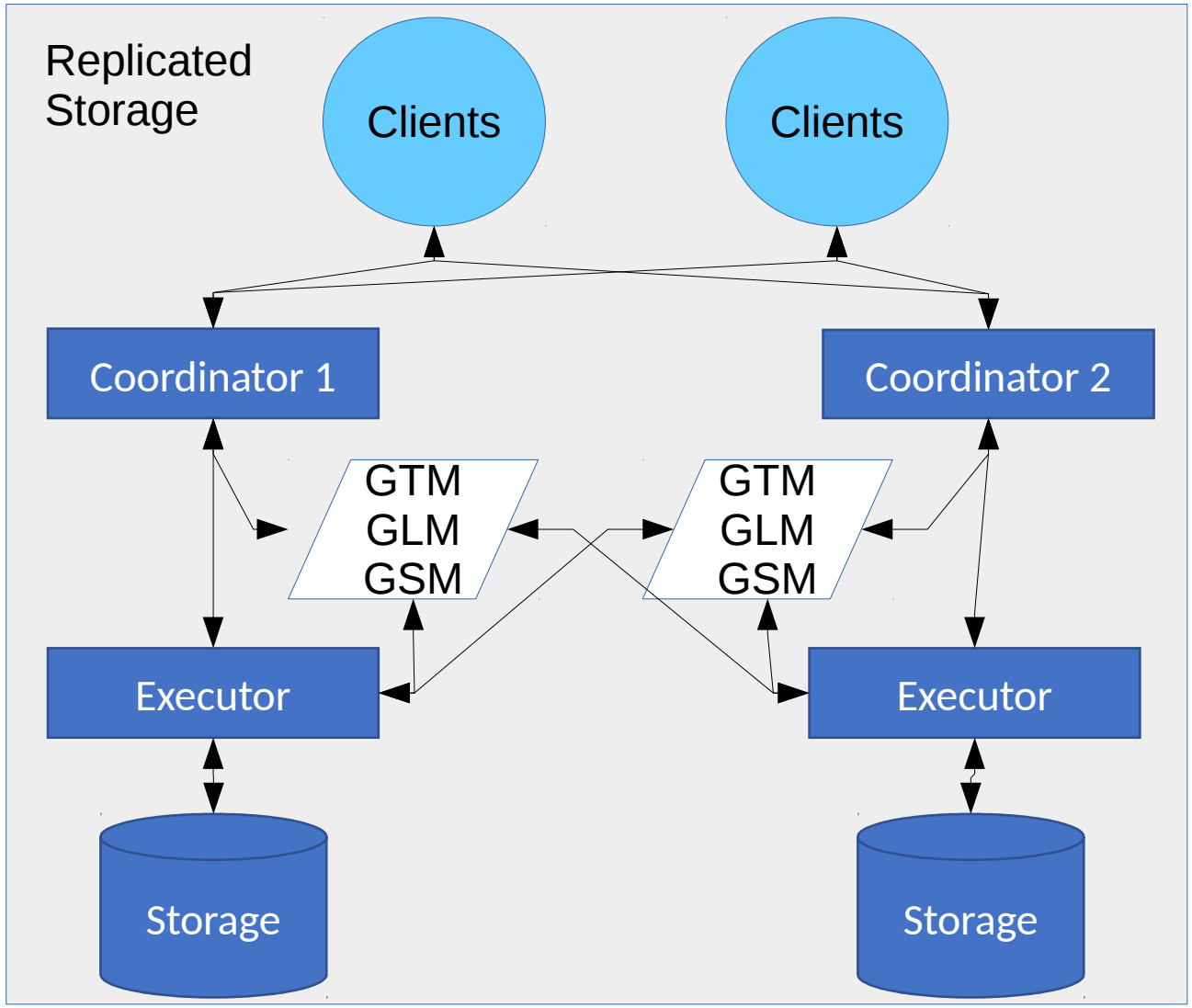
This is not the only way

Tightly vs loosely coupled

	Tightly coupled	**Loosely coupled**
Storage	Shared or sharded	Independent, replicated
Consistency and isolation	Mostly preseve ACID model	Delayed/eventual
Geographic distance (latency) and network outage tolerance	**Limited to none**	**Very good**
Data conflicts and collisions	Eagerly prevented	Optimistic, delayed resolution
Commits	Synchronously on all nodes	Usually asynchronous
Application compatibility	Transparent	May require changes

Tightly coupled models





Advantages to tightly coupled solutions

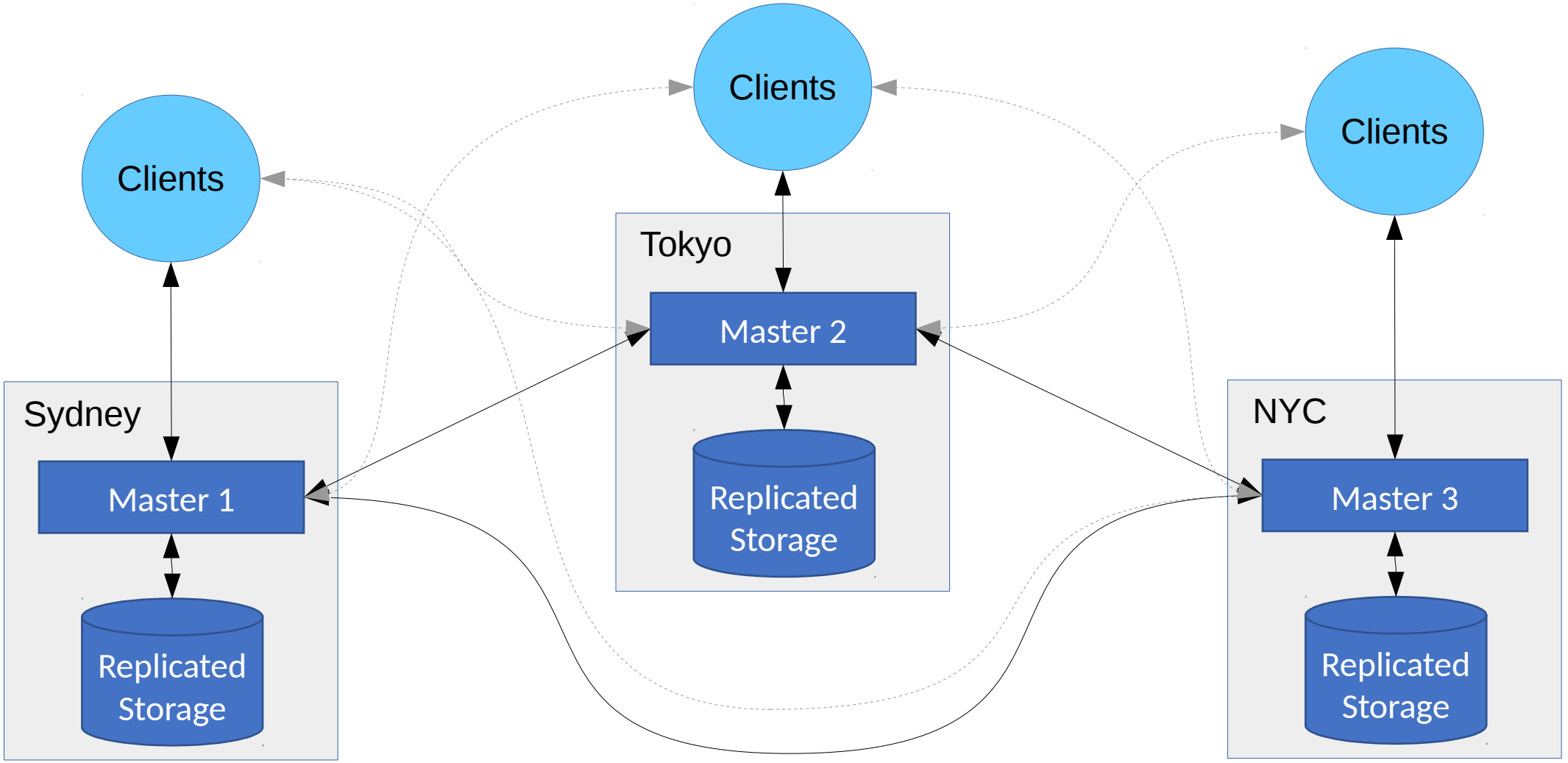
- Scalability
  - Can scale reads and writes (via sharding/distribution)
- Compatibility, consistency
  - App sees the same data on every node
  - Mostly the same semantics as a standalone DB
  - No need to change the code, queries

Disadvantages to tightly coupled solution

- Scalability
  - Read scaling limited by inter-node chatter, shared storage
  - Overheads vs single-node
- Availability
  - **Generally limited to data-center's availability**
  - Disk system can be a single point of failure
- Limited fault tolerance and geographic distribution
  - Lots of node chatter and synchronous operations = poor latency tolerance
  - Normal consistency guarantees impossible when WAN down/slow
  - Must compromise guarantees or compromise availability

Image credit: Data Center fire, TheWestsideStory.net

Loosely Coupled PostgreSQL Multi-master Replication



Advantages to loosely coupled solutions

- Availability
  - Available so long as connectivity to one node exists
  - Highly disaster-resistant due to distributed replicated storage
- Scalability
  - Linear read scaling
  - Low overheads
  - Keep data close to the user for lower latency/bandwidth

Disadvantages to tightly coupled solution

- Compatibility, consistency
  - Semantics can differ from a standalone DB
  - App can see different data depending on which node it looks at
  - Breaks some ACID assumptions
  - Multi-master **conflicts** are possible
  - Applications may require changes
- Scalability
  - Write scaling / sharding limited by latency

Can’t we have the best of both?

Compatible

Always consistent

Transparent

Conflict-free

Scalable

Synchronous

Geographically Distibuted

Highly Available

Partition Tolerant

Can’t we have the best of both?

It takes 57ms for light to travel from Sydney to London

Practical networks deliver more like 230ms.

That’s only 4.3 round trips per second

.... and 0 when the network is down

Image and time estimate: Wolfram Alpha. Practical stats: WonderNetwork.

Can’t we have the best of both?

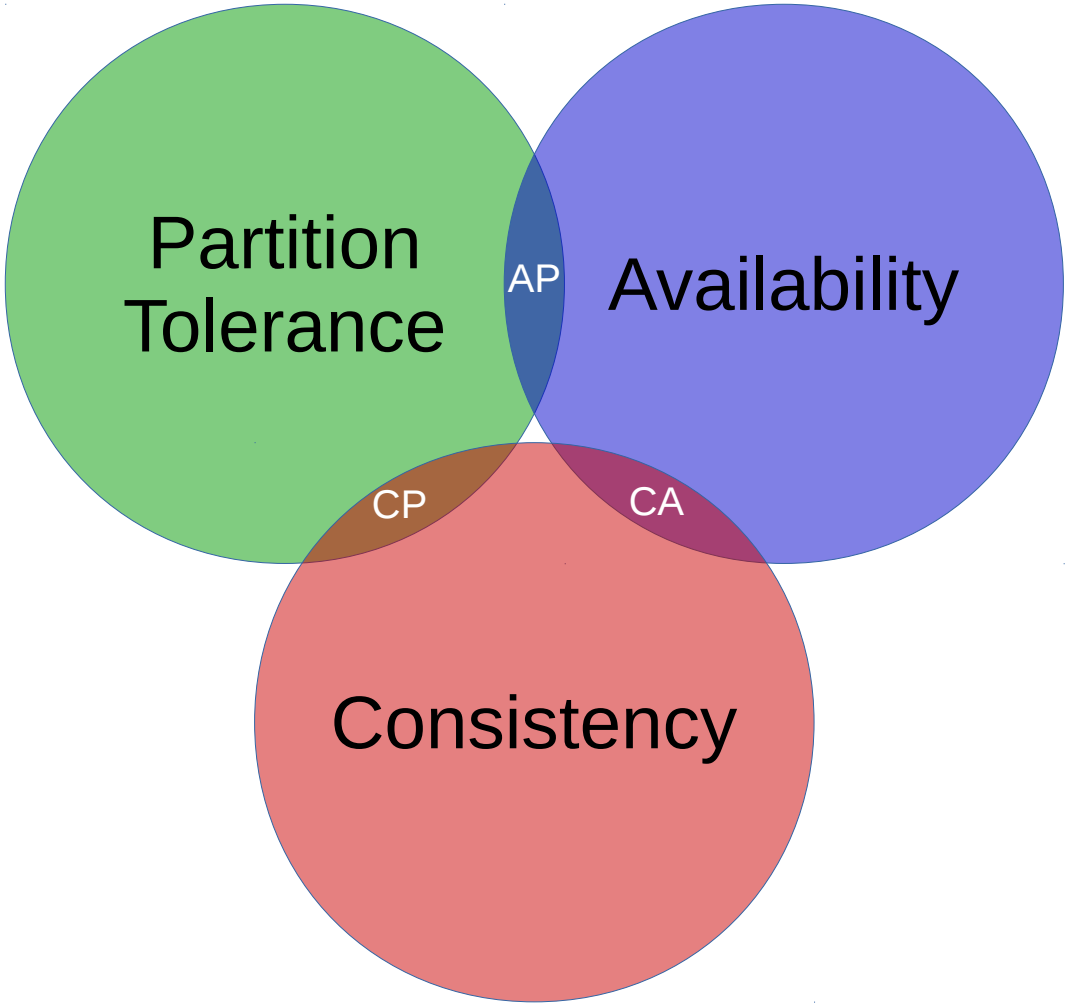
…. and 0 when the network is down

Can’t we have the best of both?

So you’ll need one of these.

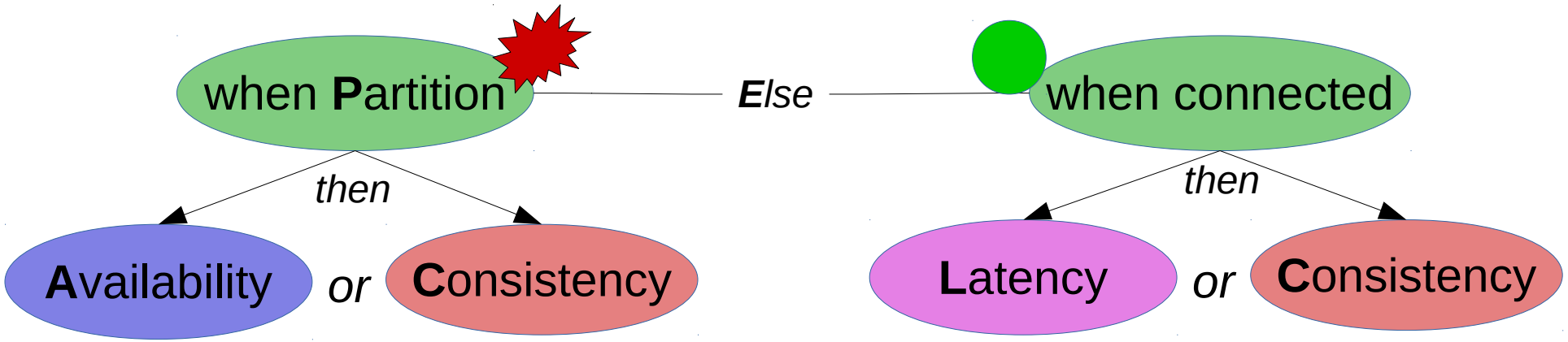
CAP and PACELC

- CAP
  - In the presence of a network **p**artition, one has to choose between **c**onsistency and **a**vailability
- “Consistent, available, partition-tolerant: pick two”
- But: “CP”, “AP” oversimplified
- CAP’s terms have different meanings
- Martin Kleppman: Please stop calling databases CP or AP [blog]



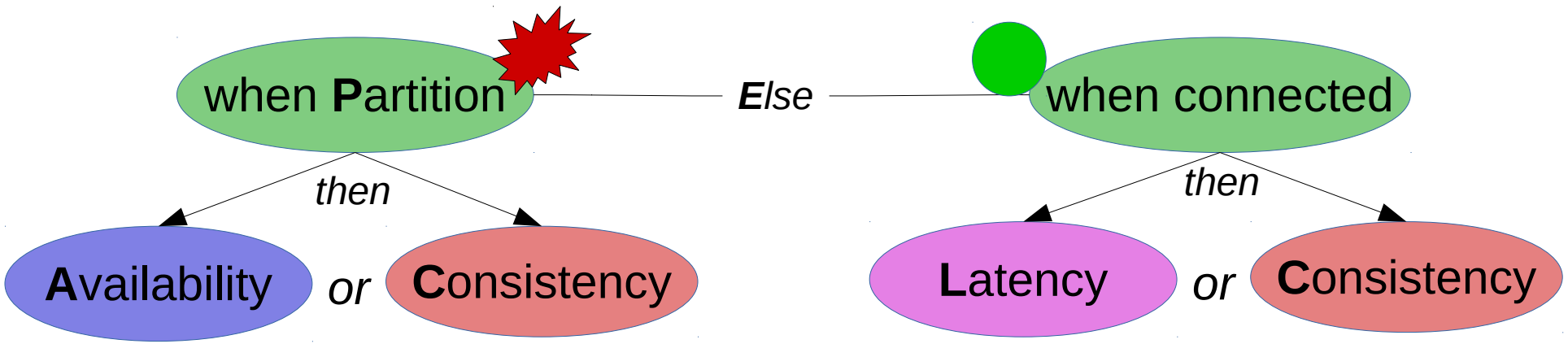
CAP and PACELC

- Better formulation: PACELC
  - In case of network partitioning (**P**) in a distributed computer system, one has to choose between availability (**A**) and consistency (**C**) (as per the CAP theorem), but else (**E**), even when the system is running normally in the absence of partitions, one has to choose between latency (**L**) and consistency (**C**).



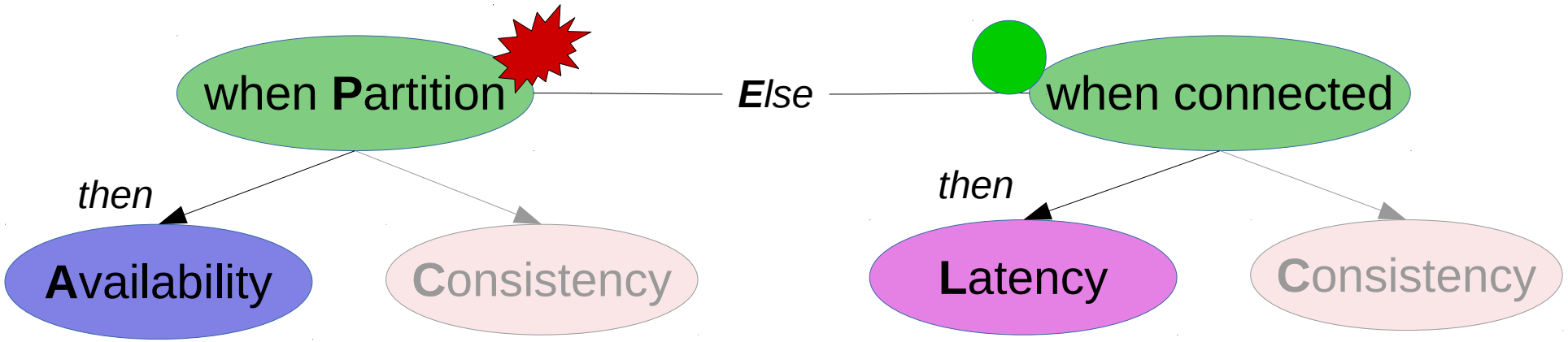
CAP and PACELC

- If it's up, it's either fast or consistent
- If it's down, it's either available or consistent



BDR is a loosely coupled system

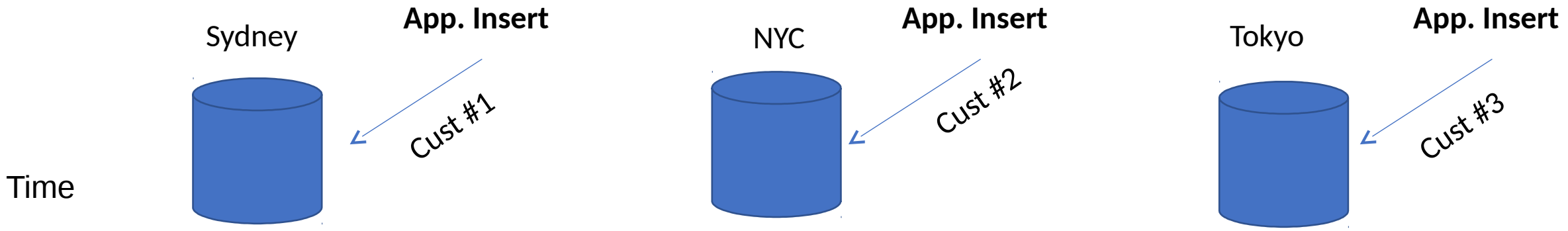
- When **P**artitioned: maintain **A**vailability and sacrifice **C**onsistency; **E**lse when connected, tolerate **L**atency and sacrifice **C**onsistency
- Frequently oversimplified to “AP system”
- Prefers availability and latency tolerance over inter-node consistency



Designing for loosely coupled

- Group of users updating the same group of replicated tables
- Primary Keys need a global generation mechanism
- The ability to replicate something at each node
- Need the ability to handle **conflicts**

Primary Key Generation



Customer ID	Customer Name
21	Cust #1

Customer ID	Customer Name
81	Cust #2

Customer ID	Customer Name
108	Cust #3

Customer ID	Customer Name
21	Cust #1
81	Cust #2
108	Cust #3

Customer ID	Customer Name
21	Cust #1
81	Cust #2
108	Cust #3

Customer ID	Customer Name
21	Cust #1
81	Cust #2
108	Cust #3

Distributed Primary Keys

- Use an external system at time of Key Generation
- Use a natural or external key (i.e. US Social Security #)
- Have a node-coordinated algorithm via node communication
- Use a node-independent algorithm (e.g. step/offset)

Step/offset ID generation

- Pros
  - Can use native PostgreSQL Types
  - A single node can generate IDs without connecting to other nodes
- Cons
  - Values not sequential between nodes
  - Fewer values for the total sequence

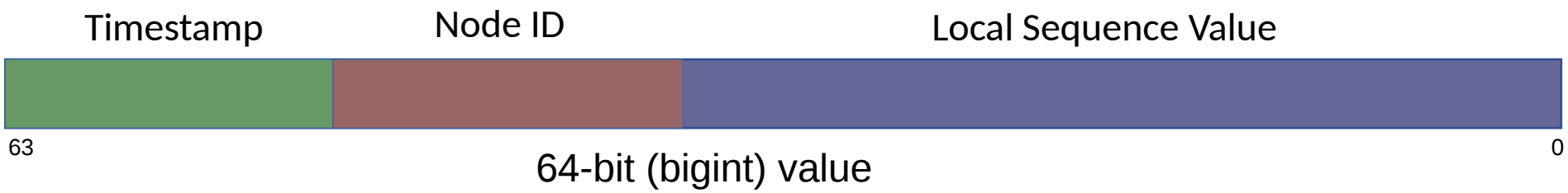
```
-- Reserve a key-space
ALTER SEQUENCE 'my_sequence' INCREMENT 100;

-- then on each node, where $1 is a node-
id:
SELECT setval('my_sequence', $1);
```

Node 1	Node 2
101	102
201	202
301	302
…	…

Time-based ID generation

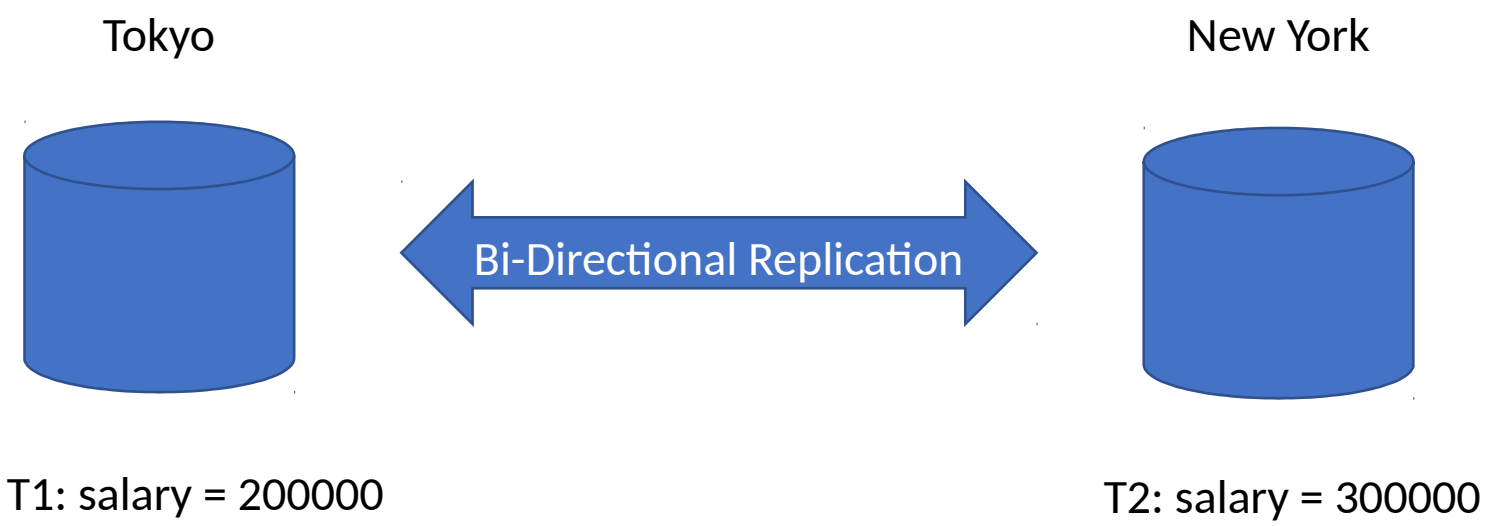
- Same as step/offset, but:
  - No manual setup on each node
  - Values mostly sequential
  - Limit to the number of IDs generated on a single node in a given time period

```
ALTER TABLE my_table DEFAULT bdr.global_seq_nextval('my_seq');
```



Conflicts

Update / Update : Conflict



In most systems: by default … last update wins. And it most cases that is OK.

Conflicts

- Many kinds:
  - Parent/child foreign key conflicts
  - Insert/Insert, Update/Update, Insert/Delete, …
- BDR’s strategy:
  - Conflict resolution, not conflict-prevention
  - Row-level, not transaction-level
  - Optional transaction level conflict prevention mode coming soon

Things to consider around conflicts

- What type of conflicts can your application create?
- Consider your deployment pattern and user access patterns
  - Reduce conflicts by maintaining user/data locality
  - Separate data into shared and unshared subsets
- How do you test for conflicts?
  - BDR has tools like apply_delay to help
- Do you need to replicate to the entire database?
- Identify operations that you need to do with a global lock and commit

Make failure modes normal!

- ChaosMonkey is the right idea
- Prevent failure cascades
- Actively simulate failures on production
- Design *and run* the app so that high latency, outages, conflicts, etc are part of normal application

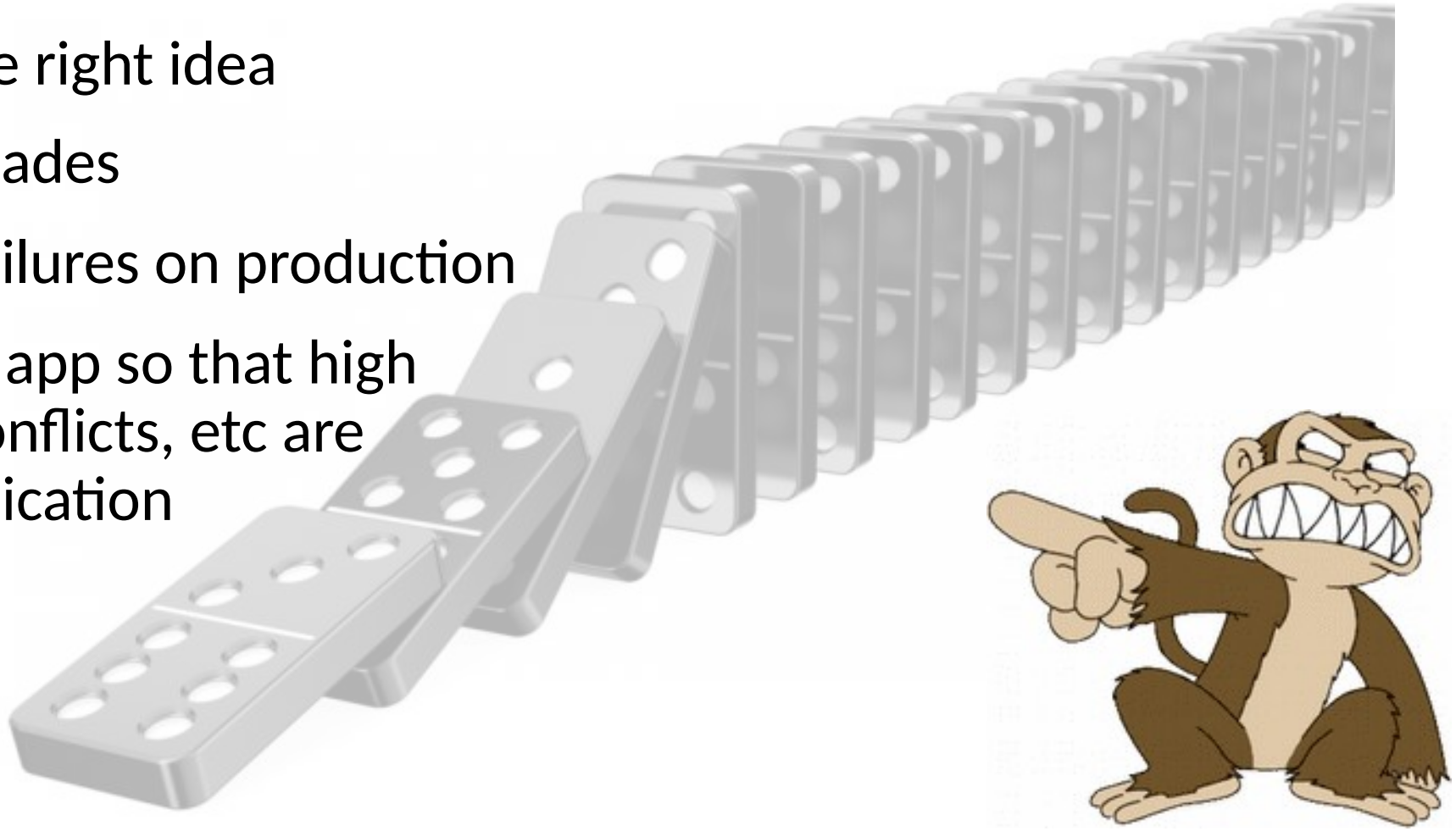
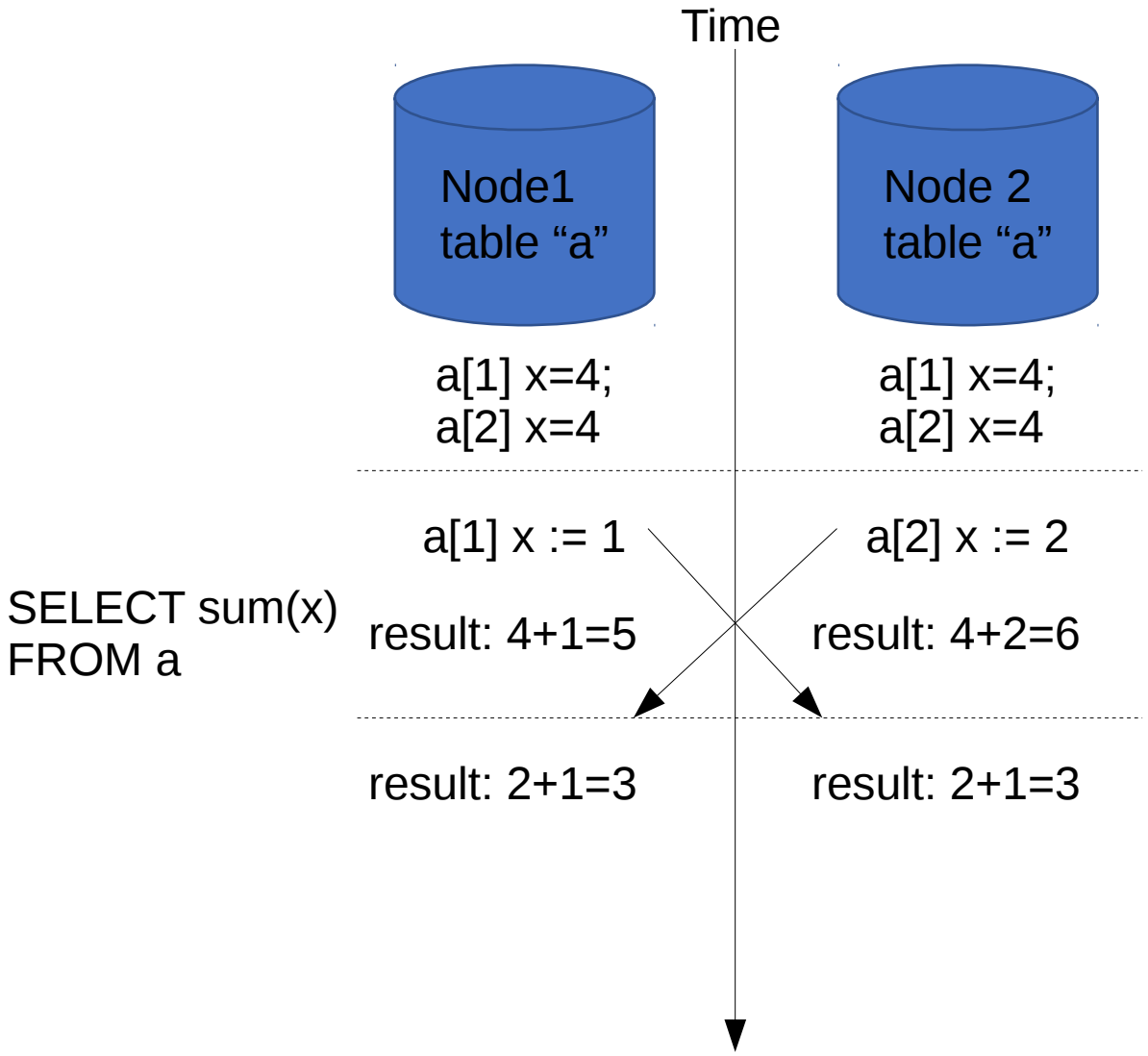


Image credits: Dominoes: Economy Watch; ChaosMonkey: CodingHorror

Visibility anomalies

- Data becomes visible on different nodes at different times
- Care is needed to prevent lost updates, phantom reads and other anomalies
- Only an issue with some access patterns
- There are also visibilty anomalies in Pg read standbys



Lock state not replicated

- Some common patterns don’t apply, like gapless sequences:

```
CREATE FUNCTION gapless_nextval() LANGUAGE sql AS $$
UPDATE my_counter_tbl SET counter = counter + 1
RETURNING counter; $$;
```

because we don’t replicate row-locks between nodes.

Schema changes

- Some schema changes need agreement of all nodes

```
ALTER TABLE t
  ALTER COLUMN c
    SET NOT NULL
```

- BDR handles this with a global lock and queue flush
- Schema changes need more planning than in stock PostgreSQL
- … but similar to low-lock schema changes in big installs

Considerable benefits

- Partition tolerant
- Disaster resistant
- Highly available
- Fast local read/write data access
- Data close to the user

You need to adapt to the differences to get the benefits.

Conclusions

- Gets the data close to the user for better user experiences
- Enable additional 9’s of availability
- Allows for selective replication for portions of the database
- 2ndQuadrant has been developing BDR for many years and contributing its foundations to Postgres

Next Steps

- 2ndQuadrant the PostgreSQL support company
- 2ndQuadrant Postgres-BDR is available now
- Come talk with me about it
- … or drop me a note at craig.ringer@2ndquadrant.com and mention this talk