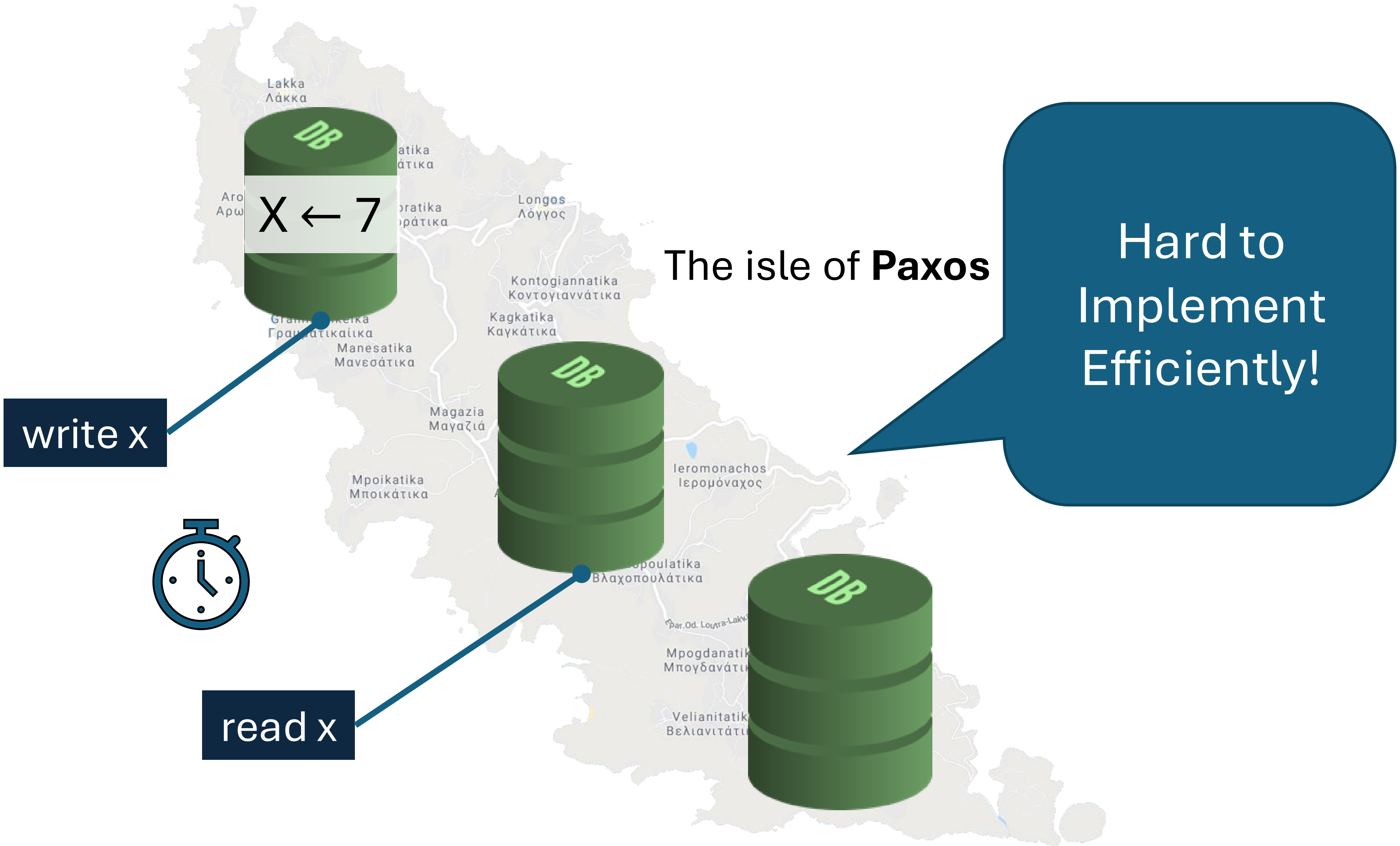
Gambit:

Sequential Consistency **without coordination** in Distributed Programming Languages

We’re building a **sequentially-consistent** programming language atop weakly-consistent replicated storage

[Memory] Consistency in Distributed Storage Systems

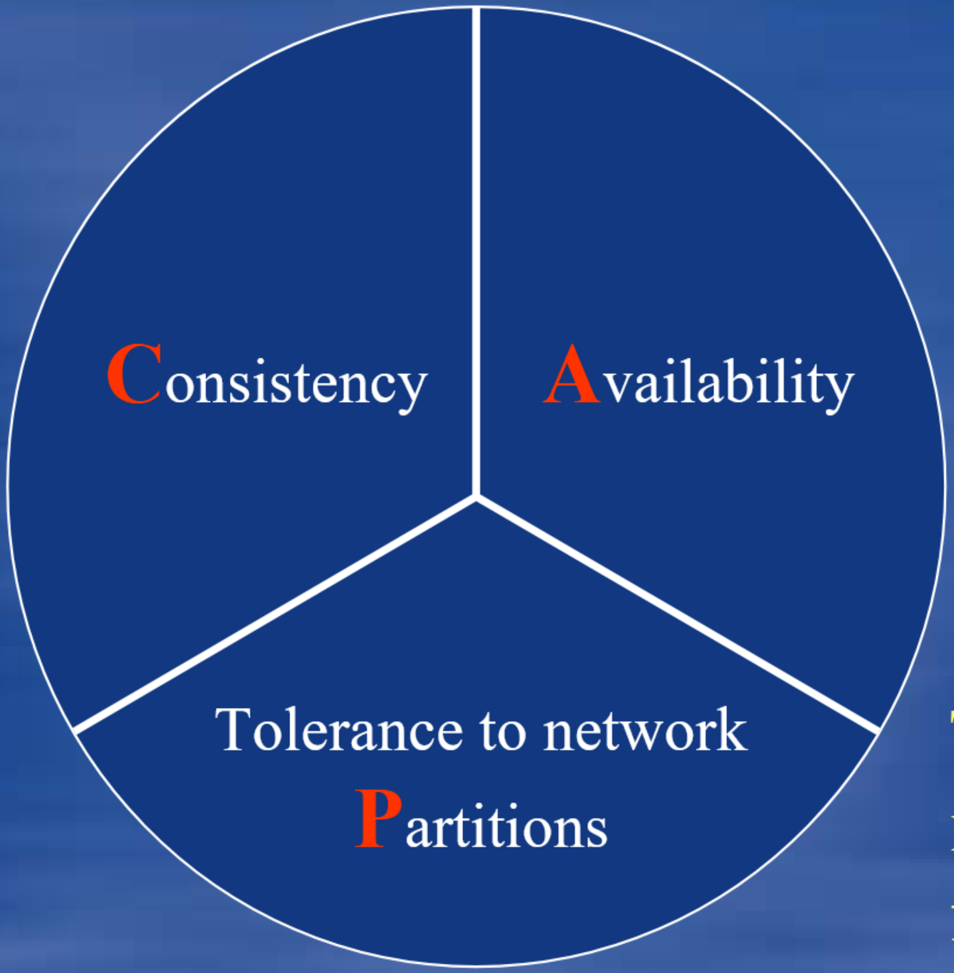
[Memory] Consistency in Distributed Storage Systems



Linearizability: Strong Consistency

The CAP Theorem

Inktomi



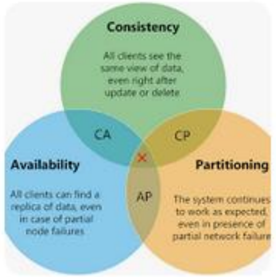
Theorem: You can have **at most two** of these properties for any shared-data system

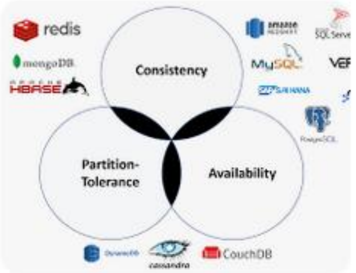
Google

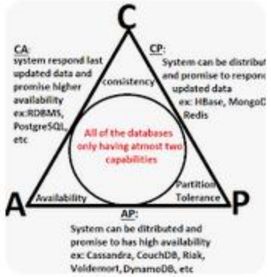
cap theorem talk

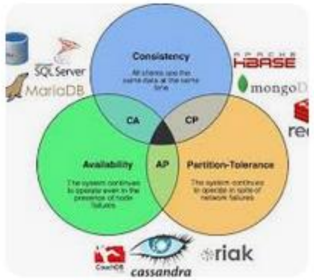
All | Images | Videos | Shopping | Maps | Books | News | More | Tools

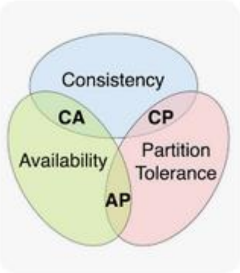
Databases | Nosql databases | Pacelc theorem | Mongodb | Distributed systems | Partition tolerance | System design | Distributed computing | Consistency availability

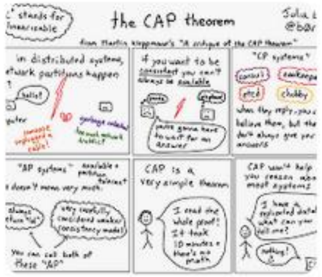

Medium
Understanding CAP the...

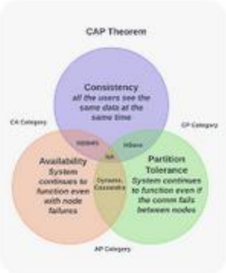

Medium
CAP Theorem — An impossible ...


GeeksforGeeks
The CAP Theorem in ...


Ashvin Choudhary - Medium
Understanding CAP Theore...


Wikipedia
CAP theorem - Wikip...

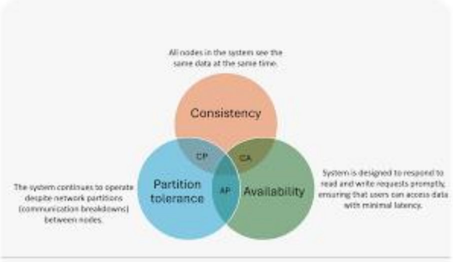

Julia Evans
A Critique of the CAP Theor...


Medium
CAP Theorem Expl...

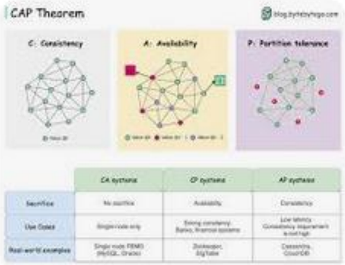

Hazelcast
What is the CAP Theo...


ScyllaD
What is C

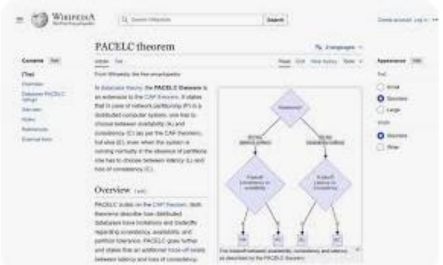

YouTube
CAP Theorem (Data Engineering with A...

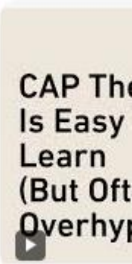

LinkedIn
CAP Theorem Principle


Hazelcast
What is the CAP Theorem...

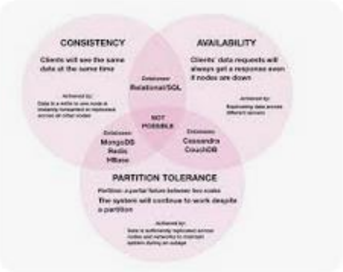

ByteByteGo
ByteByteGo | CAP Theorem: O...

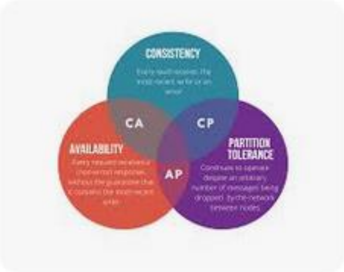

Curious Engineer - Subs...
What is CAP theorem? ...


Daily.dev
CAP Theorem Explained: Consistency ...

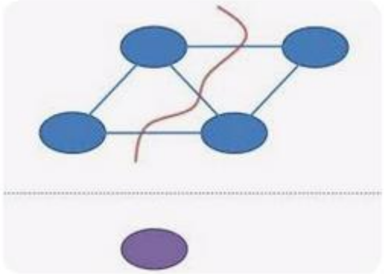

YouTube
Availability vs

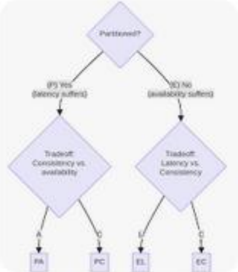

LinkedIn
CAP Theorem ...

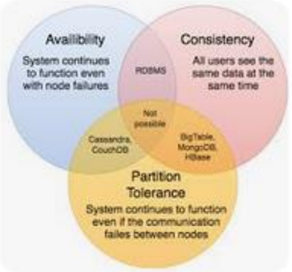

DEV Community
CAP Theorem: Why You Can't ...


Medium
Beginners -CAP Theorem | by ...

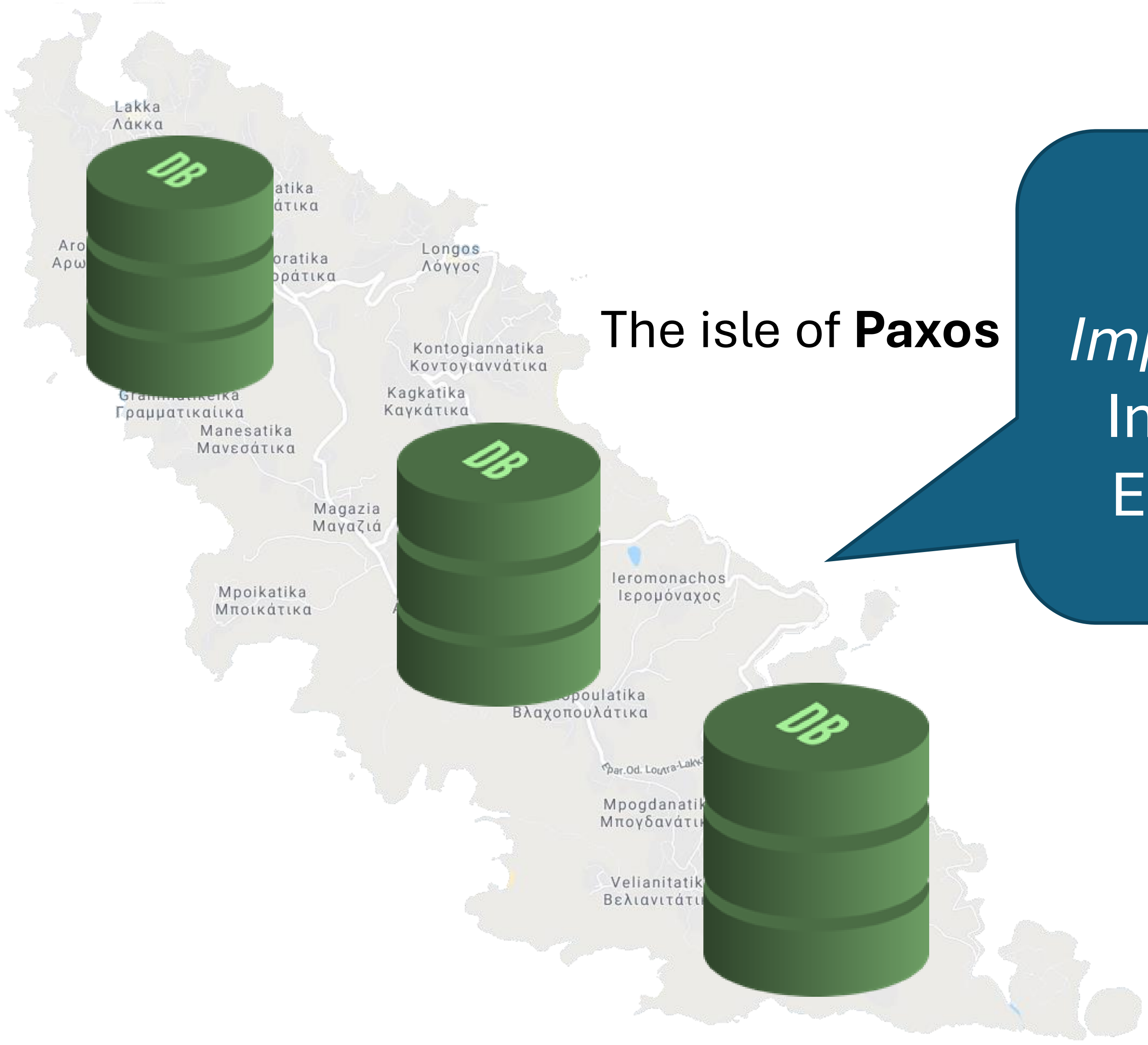

Bikas Katwal - Medium
MongoDB vs Cassandra vs...


Manh Phan
CAP Theorem of the distributed sy...


Wikipedia
PACELC design prin...


Manh Phan
CAP Theorem of the distri...


YouTube
How Lance



The isle of **Paxos**

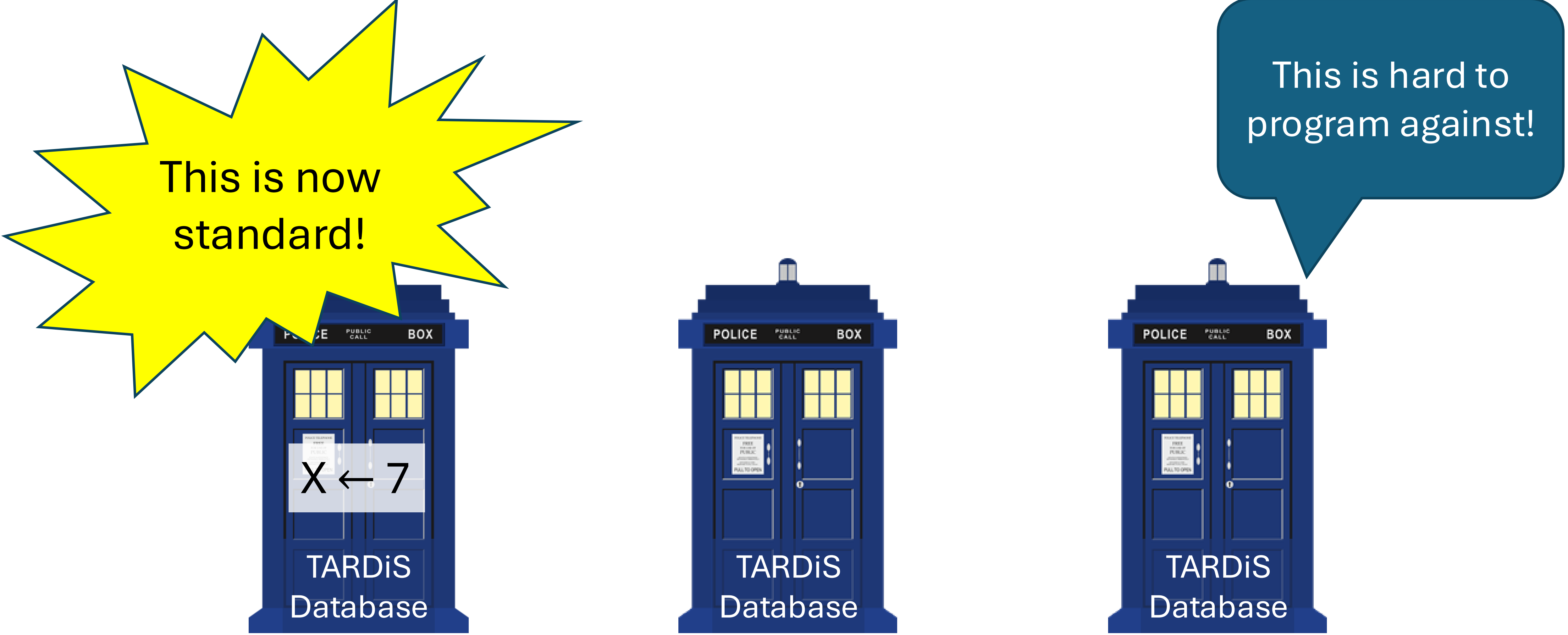
~~Hard to~~
Impossible to
Implement
Efficiently!

Linearizability: Strong Consistency

The speed of light is slow

We have outgrown Paxos

Replicas will (temporarily) diverge



Weak Consistency

Consider: an online game service

Correct under **delayed replication**

surprisingly **latency insensitive**

RECORD WINS AND LOSSES

PERFORM ONLINE MATCHMAKING

RUNS TOURNAMENTS

Gambit: **Provably consistent** programs atop **weakly-consistent** replication

In three easy steps!

Step 1:
change our assumptions about distributed interfaces

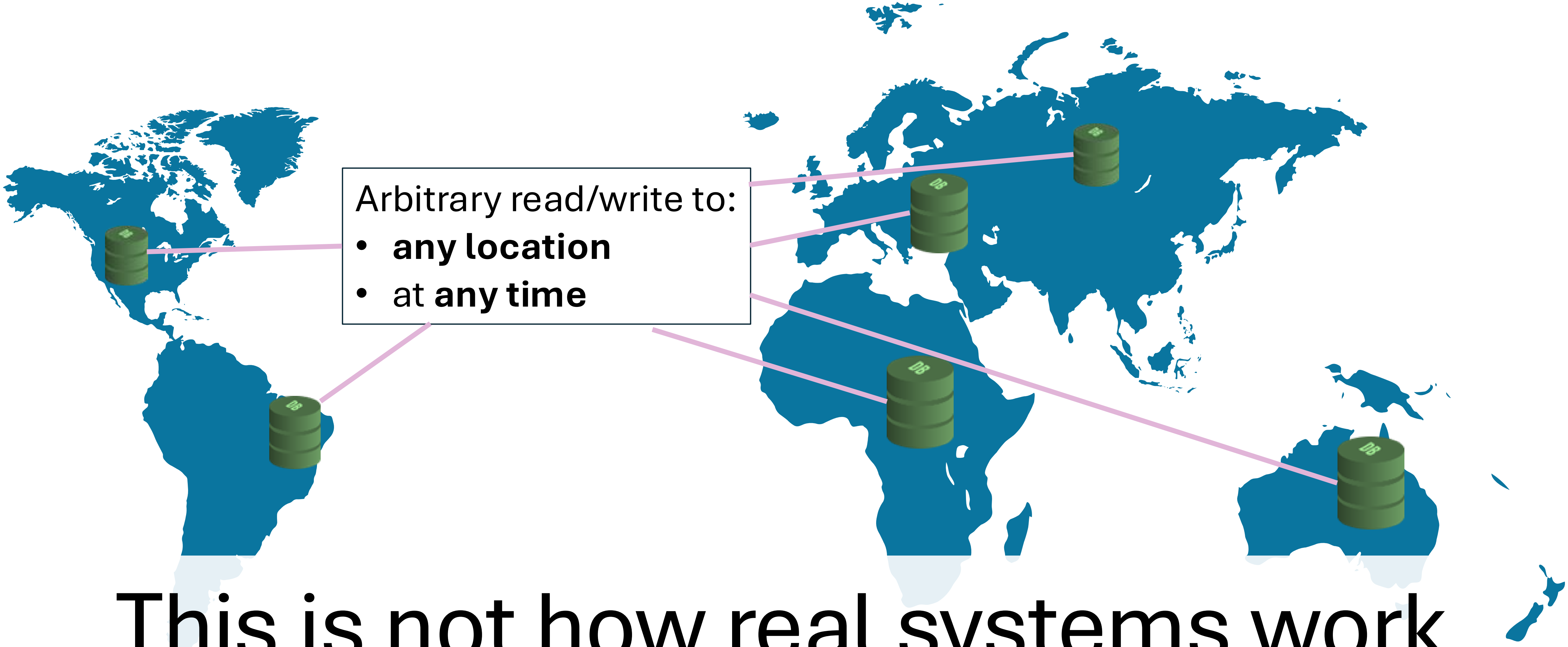
Linearizability is *needlessly strong*

Processor architectures

Language memory models

At most *sequential consistency* (weaker than linearizability)

We have the *wrong object semantics*



This is not how real systems work

Recording wins

concurrency on **`match`** calls!

```
transaction
match(player w, player l){
  wins[w]++;
  losses[l]++;
}
```

We want **increment**, not **read/write**

`wins`: player ↦ int

`losses`: player ↦ int

New assumptions

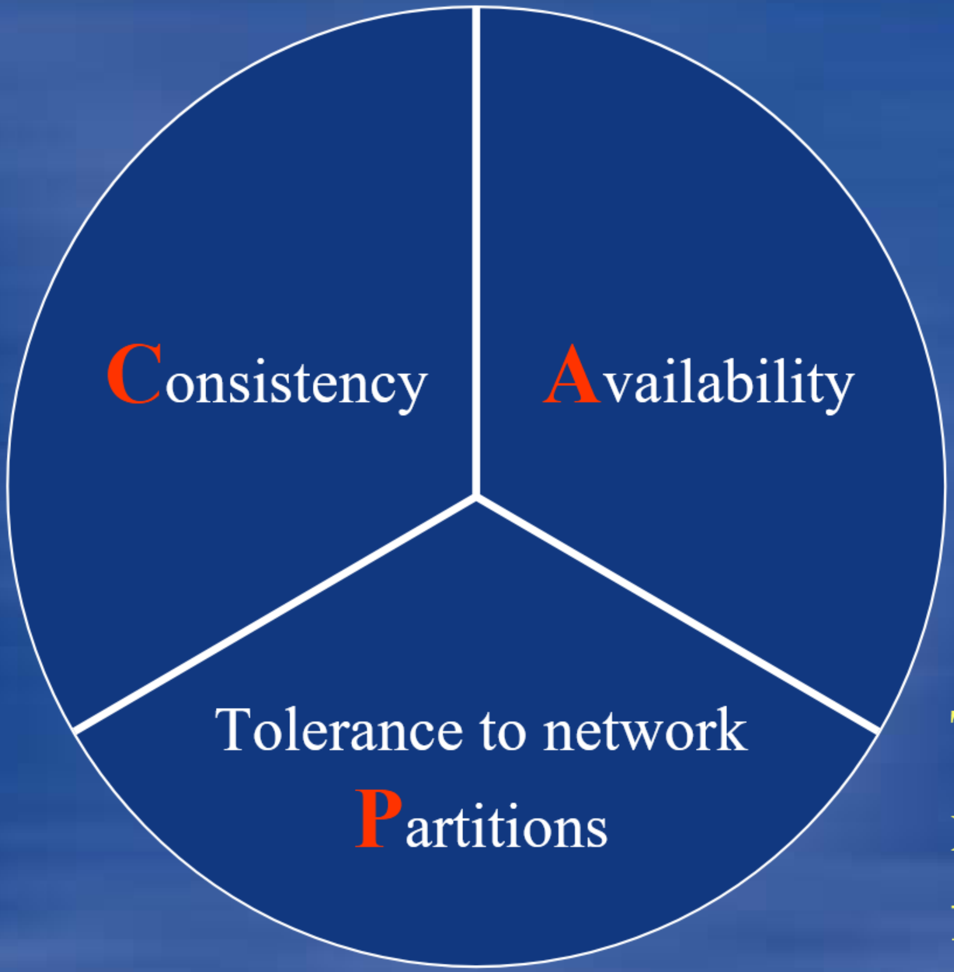
1

Provide **sequential consistency**

2

Program in terms of higher-level **replicated datatypes** with **restricted interfaces**

The CAP Theorem



Theorem: You can have **at most two** of these properties for any shared-data system

PODC Keynote, July 19, 2000

Step 2:
Reliable observations for building **sequentially-consistent** applications with **weak replication**

Reliable Observations

- Form **guarantees** about distributed state
- More **restricted mutations** allow more **general observations**

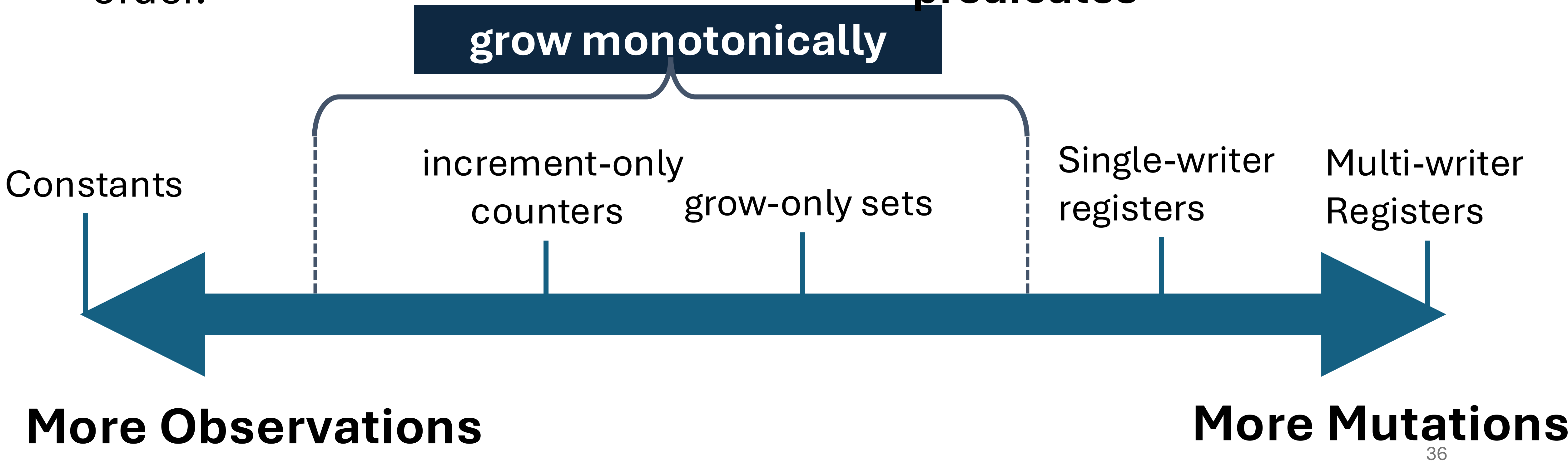
```
wins[c] ≥ 15
```

No concurrent mutations can violate

Reliable Observations

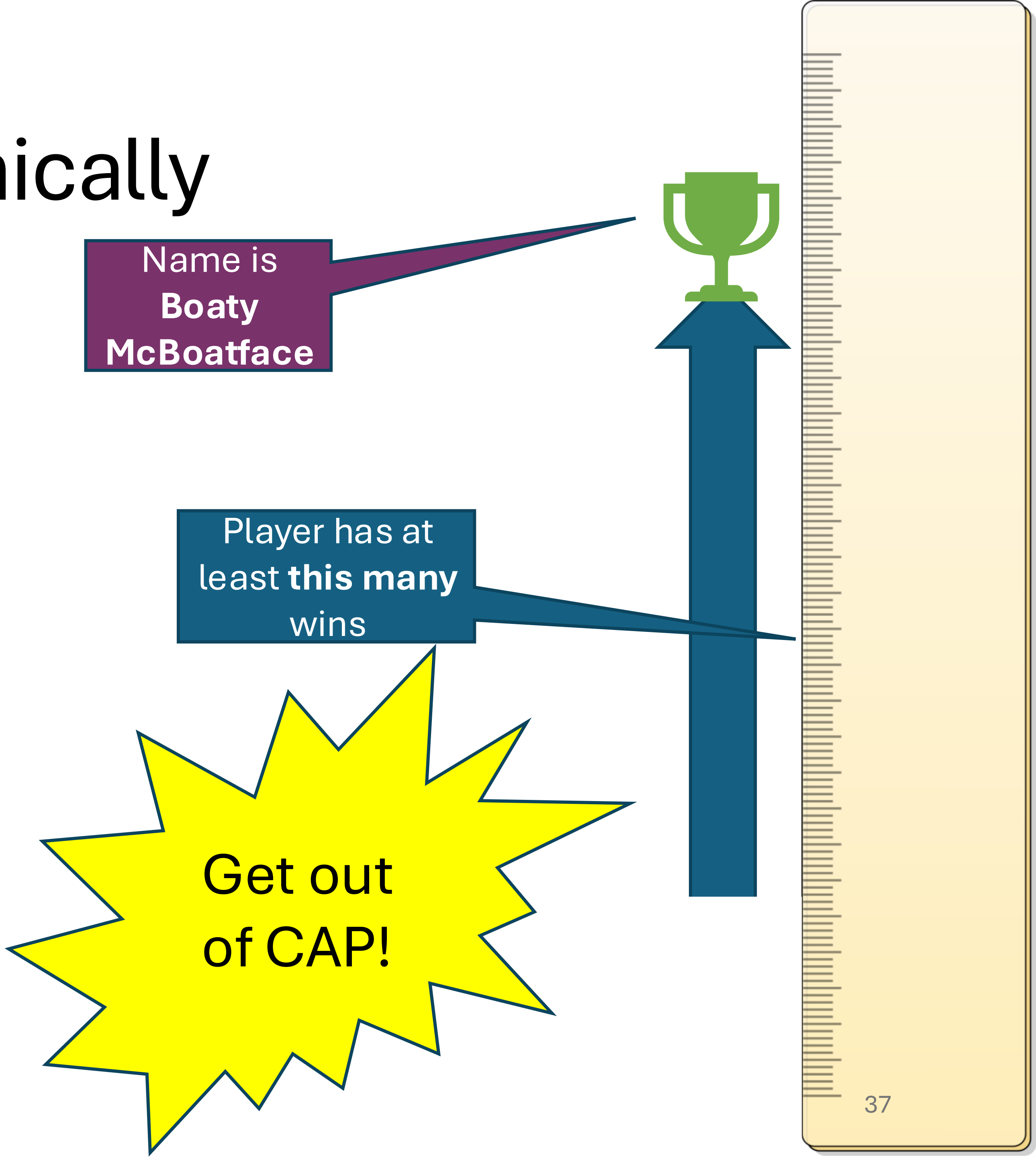
Monotonic object: mutations are inflationary with respect to some order.

Threshold observation: comparisons with constants are **stable predicates**



Programming monotonically

- If all shared objects **only grow**…
- And we only observe **thresholds**…
- Or **stable characteristics…**
- Our program can be **sequentially consistent** under **weak replication**



Can we build **something useful** with monotonicity?

Yes! many **common application behaviors** are monotonic!

`wins: player ↦ int`

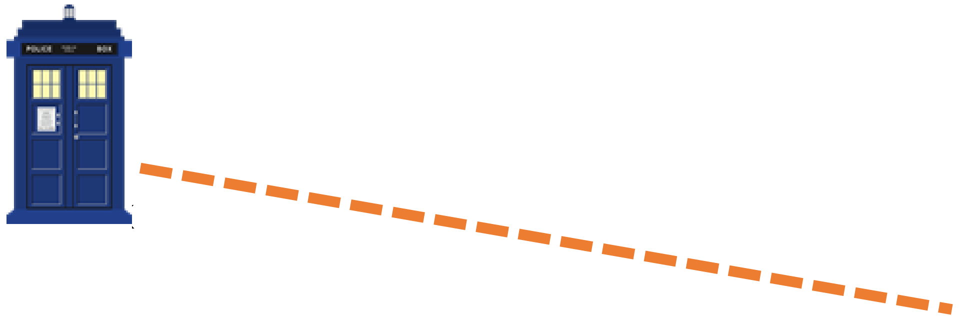
`losses: player ↦ int`

Playathon!

```
transaction playathon_check(p) {
  int played = wins[p] + losses[p];
  if (played > target)  return “thon-win!”;

}
```

Mixing Consistency Across Transactions



wins: player ↦ int

losses: player ↦ int

Playathon!

```
transaction playathon_check(p) {
  int played = wins[p] + losses[p];
  if (played > target)  return “thon-win!”;
  else abort;
}
```

How do we expose this reasoning **programmatically?**

Step 3: expose *monotonic observations* via a programming language

Our goals:

- Require only **weak replication**
- Support **imperative** / object-oriented programming
- Share **user-provided** datatypes
- Guarantee **sequential consistency**

Big idea: refine datatype interfaces via *shared restrictions*

```
interface Counter{
    void inc();
    void dec();
    int  get();
    void set(int i);
}
```

```
interface Map<K,V>{
    void add(K k, V v);
    void clear(E e);
    Maybe<V> lookup (K k);
}
```

```
Map<Player,Counter> wins;
Map<Player,Counter> losses;
```

```
interface Counter{
    void inc();
    void dec();
    int  get();
    void set(int i);
}
```

```
interface Map<K,V>{
    void add(K k, V v);
    void remove(K k);
    Maybe<V> lookup (K k);
}
```

Restricts get to **positive, monotonic** uses

```
restriction Up for Counter{
    allows inc;
    allows mon +get
}
```

```
restriction CheckOnly for Map{
    allows mon +lookup;
}
```

```
restriction Down for Counter{
    allows mon -get
}
```

```
restriction RemoveOnly for Map{
    allows remove;
    allows mon -lookup;
}
```

```
restriction Write for Counter{
    allows set;
}
```

```
restriction Up for Counter{
    allows inc;
    allows mon +get
}
```

```
restriction CheckOnly for Map{
    allows mon +lookup;
}
```

```
shared[Up] Counter c;
...
c.inc();
if (c.get() > 13){ . . . }

else {…}
```

Statically Guaranteed monotonic!

```
restriction Up for Counter{
    allows inc;
    allows mon +get
}
```

```
restriction CheckOnly for Map{
    allows mon +lookup;
}
```

```
Map<Player, shared[Up] Counter> wins, losses;
```

```
void match(Player w, Player l){
  wins[w]++;
  losses[l]++;
}
```

```
string playathon_check(Player p) {
  int played = wins[p] + losses[p];
  if (played > target) return “winner!”;
  else abort;
}
```

Abort always allowed

must return **string**

Big idea: track **provisional observations** via an **information-flow** type system

```
restriction Up for Counter{
    allows inc;
    allows mon +get
}
```

```
restriction CheckOnly for Map{
    allows mon +lookup;
}
```

```
Map<Player, shared[Up] Counter> wins, losses;
```

Inferred **provisional label:**

Provisional observation: wins/losses *may be inconsistent*

```
provisional string playathon_check(Player p) {
  int played = wins[p] + losses[p];
  if (played > target) return "winner!";
  else return "no";
}
```

Error: no **visible actions** on **provisional data**

```
. . .
provisional string cr = playathon_check(…);
print(cr);
```

```
restriction Up for Counter{
    allows inc;
    allows mon +get
}
```

```
restriction CheckOnly for Map{
    allows mon +lookup;
}
```

```
Map<Player, shared[Up] Counter> wins, losses;
```

```
provisional string playathon_check(Player p) {
  int played = wins[p] + losses[p];
  if (played > target) return “winner!”;
  else return “no”;
}
```

block until provisional status resolves

```
. . .
provisional string cr = playathon_check(…);
await cr;
print(cr);
```

```
interface Map<K,V>{
    void add(K k, V v);
    void clear(E e);
    Maybe<V> lookup (K k);
}
```

```
restriction CheckOnly for Map{
    allows mon +lookup;
}
```

```
await transaction new_player(Player p, shared[?] Map m) {
  m.add(p, new Counter());
}
```

```
restriction Up for Counter{
    allows inc;
    allows mon +get
}
```

```
restriction Down for Counter{
    allows dec;
    allows mon -get
}
```

```
shared[Read] Sum<shared[Up] Counter, shared[Down] Counter> c;
```

```
await transaction swap_restriction(shared[?] Sum<shared[?] T, shared[?] T> c) {
  if (staged.is_left()) staged.right = staged.left;
  else staged.left = staged.right;
}
```

By **restricting** objects to **monotonic** interfaces,
and tracking provisional actions via **information-flow**,
Gambit provides **strong consistency** atop **weak replication.**

A **system**, *not just* a language

Erlang/Java implementations

Custom replication protocols

Convergent, transactional semantics

Initial, buggy implementation

Gambit

- **Step 1**: program against **objects**, not **read-write registers**
- **Step 2**: define **stable observations** in terms of monotonicity
- **Step 3: build** a **new programming language** for monotonicity