



THE UNIVERSITY *of* EDINBURGH  
**informatics**

SCHOOL OF LECT

# Distributed Systems (Level 11) - Lecture

E:AW8MZTAKLZ

- Friday, 16 October 2026
  - 16:10 - 18:00
- Lecture Theatre 3

Scan to check-in



Expires at 16/10/2026 18:10



# Building a Parameter Server

Introduction to the Distributed Systems coursework

---

Distributed ML · vector clocks · consistency · elasticity · fault tolerance

Björn Franke

16 October 2026

# Session plan (90 minutes)

## A. Why (25 min)

Distributed ML in one slide  
The parameter server role  
Why it is hard

## B. Concepts (30 min)

Async tasks  
Consistency models  
Vector clocks  
Hashing, replication

## C. Coursework (30 min)

Parts A–F  
Test harness  
Experiments  
Feedback

## D. Wrap-up (5 min)

How to start  
Pitfalls  
Questions

No machine-learning background is required. You build the systems part.

# Why distributed machine learning?

- Models have billions to trillions of parameters
- Training data is terabytes: one machine is too slow
- So: many machines train one shared model
- **Core problem:** keep the shared model consistent and fast to update

The parameter server is the system that holds and updates the shared model.

# What is being learned?

---

A model is a vector of numbers  $w$  (weights). Training finds the  $w$  that minimises error on data:

$$\min_w \sum_{j=1}^n \ell(y_j, \langle x_j, w \rangle) + \lambda \Omega(w)$$

- Details of the loss do not matter here
- What matters: how workers access  $w$ , and how often

# Gradient descent in one slide

- Compute a gradient  $g$ : same shape as  $w$ , says how to change each weight
- Update with learning rate  $\eta$ , repeat thousands of times

$$w \leftarrow w - \eta g$$

## Compute-heavy

Gradient computation on local data

## Communication-heavy

Reading and writing  $w$ , every iteration

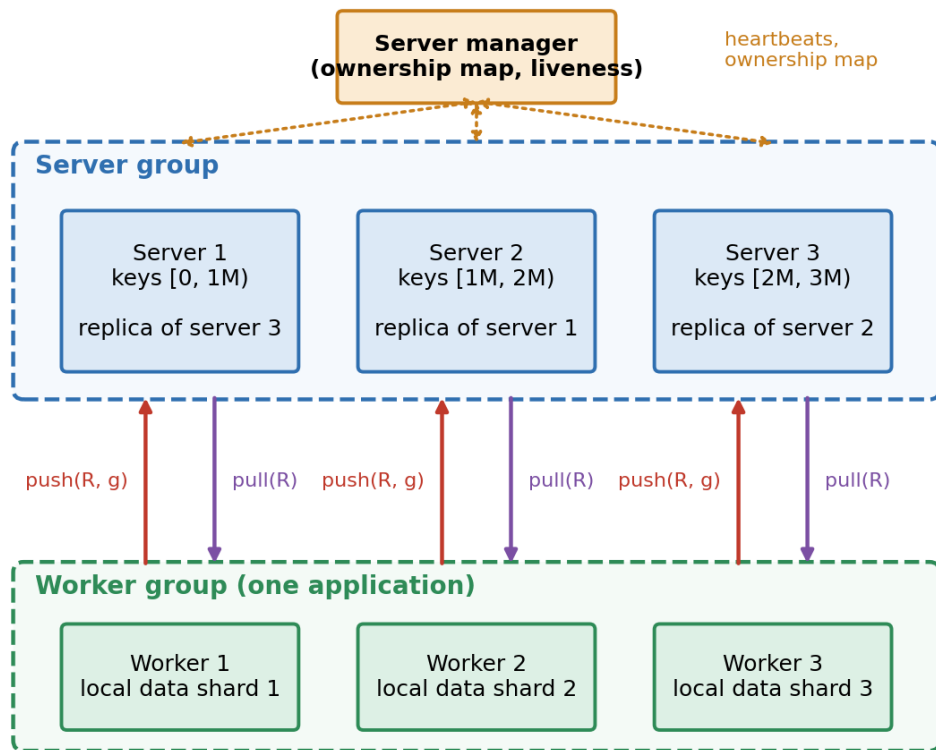
# Data parallelism: workers and shards

- Split the data over  $m$  workers; each computes a partial gradient  $g_k$
- Gradients from different shards simply add
- The weights live on servers, not on workers

$$g = \sum_{k=1}^m g_k, \quad w \leftarrow w - \eta (g + \lambda \nabla \Omega(w))$$

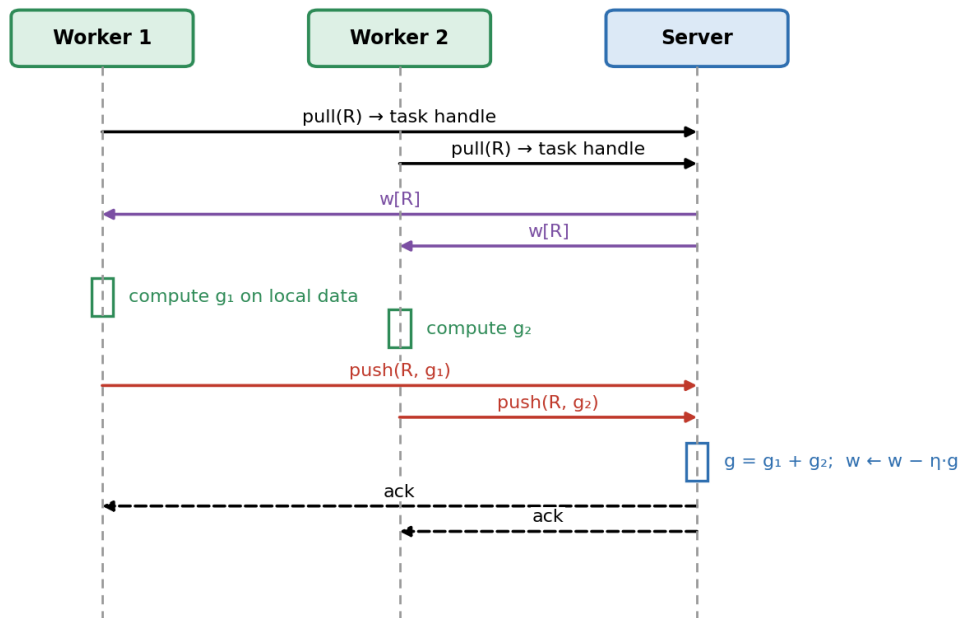
Aggregation is arithmetic: a parameter server is not just a key–value store.

# The parameter server architecture



- Servers: hold weights, partitioned by key range
- Workers: hold data shards
- Manager: ownership map and liveness
- Workers pull weights, push gradients

# One training iteration



- Pull needed weights
- Compute gradient (local)
- Push gradient
- Server sums gradients and updates  $w$
- Repeat

# Why this is a hard systems problem

A · Why

## Huge vectors

Partition across servers; communication dominates compute.

## Sparse access

Each worker needs few keys: push/pull only those.

## Tolerates staleness

Slightly old weights still converge: relax consistency.

## Commodity clusters

Failures are routine; nodes join and leave.

# Five design requirements

A · Why

| Requirement             | Meaning                        | Part |
|-------------------------|--------------------------------|------|
| Ease of use             | Sparse/range-native API        | A    |
| Efficient communication | Async, range messages, caching | A, D |
| Flexible consistency    | Sequential → eventual          | C    |
| Elasticity              | Add/remove nodes live          | E    |
| Fault tolerance         | Replication, vector clocks     | F    |

# Range-based key–value API

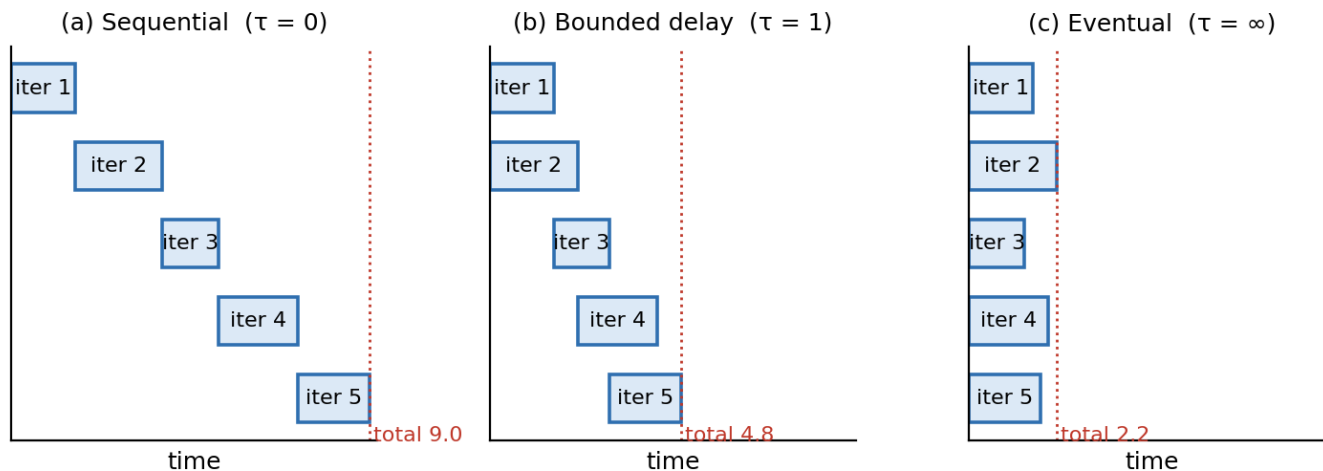
- Keys: 64-bit ids. Values: numeric vectors
- $\text{push}(R, g): w_i \leftarrow w_i - \eta g_i$  for every key  $i$  in range  $R$
- $\text{pull}(R)$ : returns the weights for  $R$
- Single key = ordinary key–value access
- Keys need not be resent if both sides already agree on them

Design for sparse key lists AND dense ranges.

# Asynchronous tasks and dependencies

- Every push/pull is a task: returns a handle immediately
- A task may declare dependencies on earlier handles
- The server starts it only when they complete
- Overlap communication with computation

```
h1 = pull(K)    →    compute    →    h2 = push(K, g, deps=[h1])
```



- Sequential ( $\tau=0$ ): wait for the previous iteration
- Bounded delay ( $0<\tau<\infty$ ): limited overlap
- Eventual ( $\tau=\infty$ ): never wait

# Bounded delay: the trade-off

iteration  $t$  may start only when iterations  $1, \dots, t - \tau - 1$  have completed

## Larger $\tau$ gains

Hides stragglers and network latency  
Higher throughput

## Larger $\tau$ costs

Workers compute on staler weights  
Slower or unstable convergence

You will measure this trade-off in the harness (experiment X3).

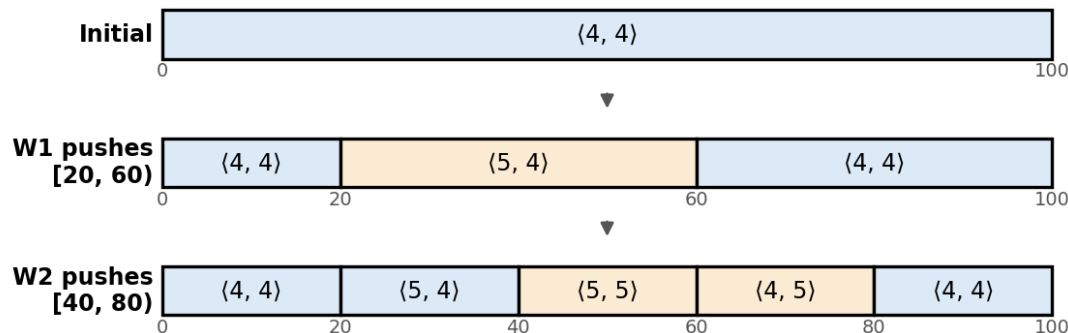
# Vector clocks: ordering updates

- A vector of counters: one entry per worker
- Clock a is no later than b if every entry is  $\leq$
- Neither  $\leq$  the other: concurrent
- Used to order updates and to know what to resend after failure

$$C_a \leq C_b \iff \forall k : C_a[k] \leq C_b[k]$$

$\langle 3, 2 \rangle$  vs  $\langle 2, 3 \rangle$ : neither is no later than the other, so they are concurrent.

# Per-range vector clocks



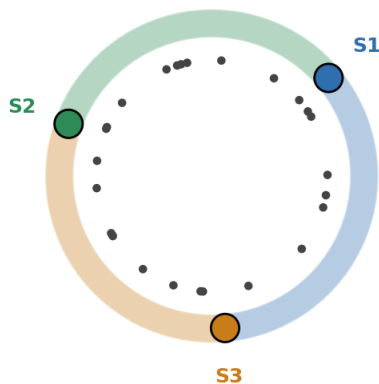
*Clock entries (W1, W2) per range. Ranges split only at push boundaries, so clock count tracks the number of distinct ranges, not keys.*

- Per-key clocks cost  $O(m \cdot p)$ : too expensive
- Operations act on ranges, so one clock can cover a range
- Split a range only when a push covers part of it
- Your task C3: show clock count stays small

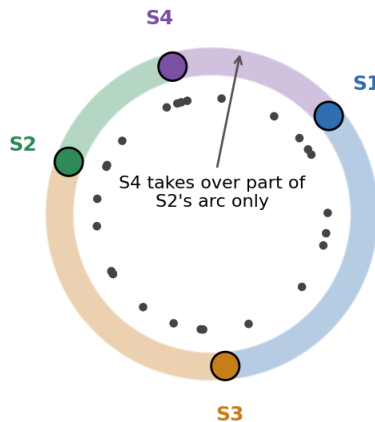
$$\mathcal{O}(m \cdot p) \rightarrow \mathcal{O}(m \cdot r), \quad r \ll p$$

# Consistent hashing and elasticity

Before: 3 servers



After S4 joins

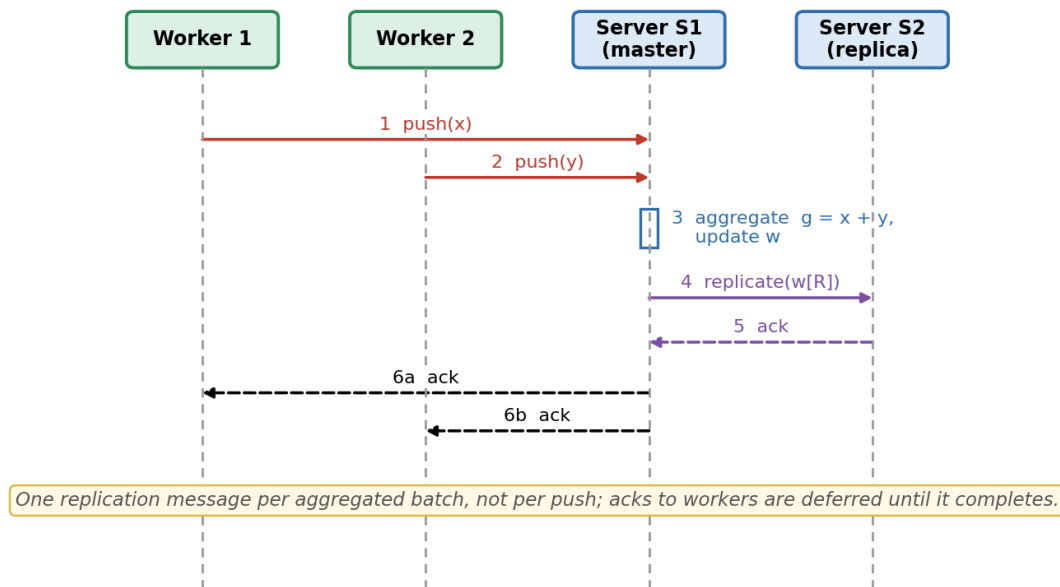


Each server owns the arc ending at its position (grey dots = keys). Only keys on the affected arc move.

$$E[\text{keys moved}] \approx \frac{K}{S+1}$$

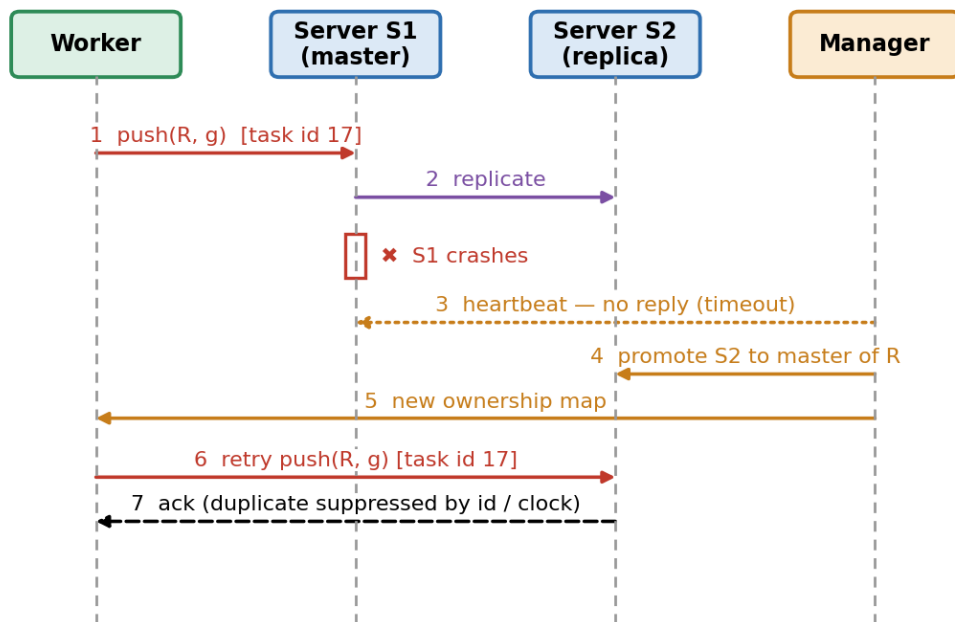
- Servers and keys on a ring
- Server owns the arc ending at it
- Join/leave moves a small share of keys
- $\text{key mod } S$  would move almost all

Virtual nodes even out arc lengths and spread load after a failure.



- Range replicated on k neighbours
- Ack worker only after replication
- Aggregate first: one replication message per batch
- Trade-off: bandwidth vs ack latency

# Manager, heartbeats and recovery



- Heartbeat timeout detects failure
- Promote replica; publish new map
- Retried push keeps its task id
- New master suppresses duplicates
- Trade-off: detection speed vs false positives

# Coursework overview

C · Coursework

## Part A · wk 1–2

Single-node server

## Part B · wk 1–2

Test harness

## Part C · wk 3

Distribution, clocks,  
consistency

## Part D · wk 4

Communication efficiency

## Part E · wk 4

Elasticity

## Part F · wk 5

Fault tolerance

**Formative: oral feedback as part of a conversation. Not marked.**

## Rules

Any language and RPC library  
No existing parameter-server library  
No off-the-shelf distributed KV store as core  
Everything runs locally; scripts to start, kill, add nodes  
README: runs in 10 minutes  
**No ChatGPT, please!**

## Deliverables

Code repository + harness + configs  
Informal(!) Report ( $\leq 8$  pages)  
10-minute recorded demo: add and kill a server under load  
Groups of 3-5 students, 30–40 hours

# Part A: single-node parameter server

---

- A1 Data model, push/pull/wait, update rule; sparse and range access
- A2 Ease of use: a worker loop in ~10 lines, good errors, quick-start
- A3 Task dependencies
- A4 Unit tests: concurrent pushes, ranges vs lists, dependencies

Checkpoint (end of week 2): demo A1–A4 driven by the harness.

## Part B: the test harness

---

- Emulates workers: pull → simulated compute → push
- Sparse Zipf-skewed access; dense range mode
- Stragglers and heterogeneous compute times
- Scenario script: kill, add, delay, drop
- Metrics: throughput, p99 latency, bytes, convergence, recovery

Seeded and config-driven: same config + seed gives the same run.

# A correctness oracle without ML

Hidden target  $w^*$ . Each worker pushes  $g = w - w^*$ . With  $m$  workers touching every key, sequential consistency:

$$\|w(t) - w^*\| = |1 - \eta m|^t \|w(0) - w^*\|, \quad \text{converges iff } 0 < \eta m < 2$$

## Use 1

Measured error must match the formula: strong test of aggregation

## Use 2

Wrong factor reveals duplicated or lost pushes

# Part C: distribution, clocks, consistency

---

C · Coursework

- C1 Partition the key space; client routes and splits requests
- C2 Server manager: ownership map, cached by clients
- C3 Per-range vector clocks with splitting
- C4 Sequential, eventual and bounded-delay consistency
- C5 Evaluate throughput vs convergence, with stragglers

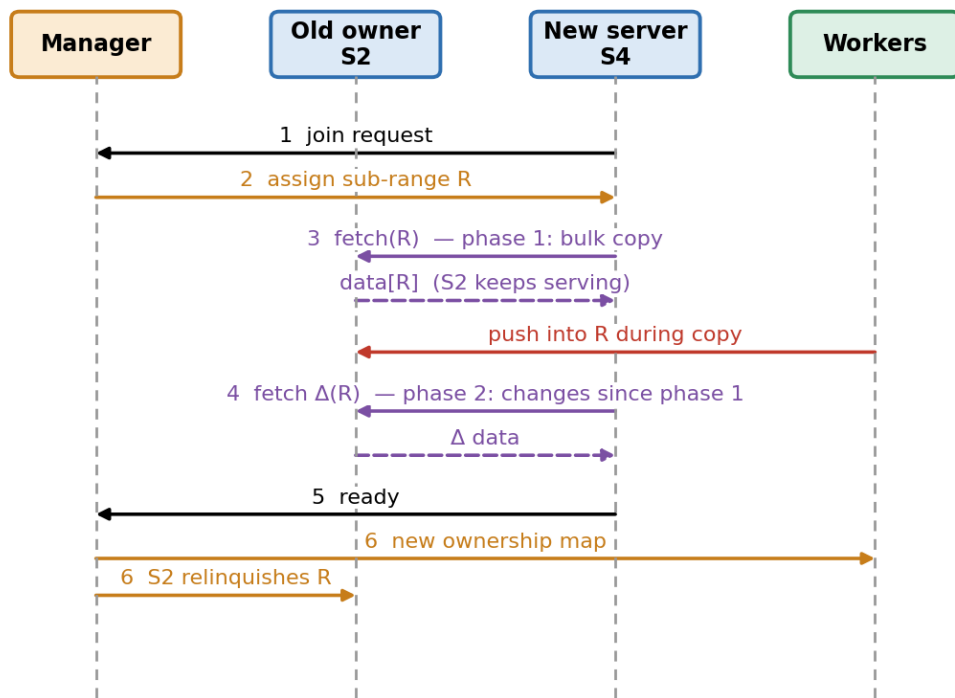
## Part D: efficient communication

---

- Choose at least three:
- Key caching (send a signature, not the key list)
- Value compression, batching, pipelining
- Aggregate before replicating
- Filters: thresholds, reduced precision

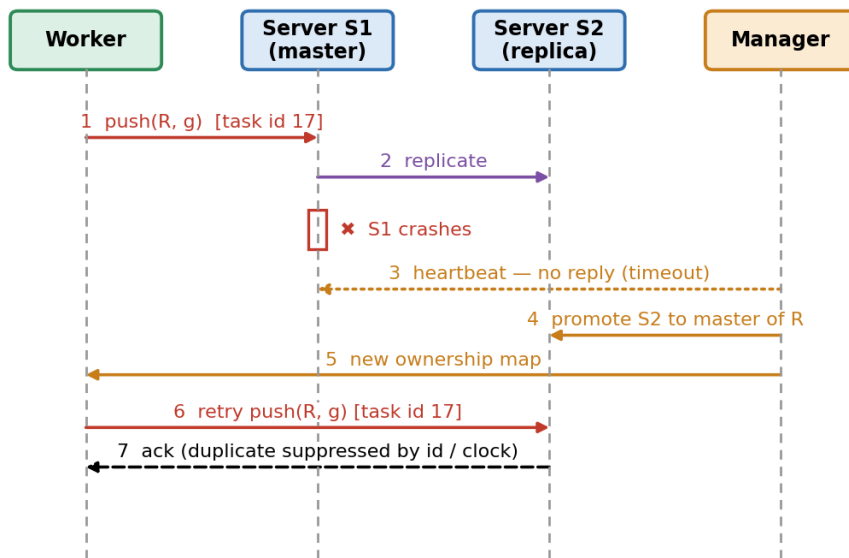
Report saving =  $1 - B_{\text{optimised}} / B_{\text{baseline}}$ , plus costs.

# Part E: elasticity



- E1 Add a server under load: two-phase transfer
- E2 Graceful removal
- E3 Workers join and leave
- E4 Double the servers mid-run; plot the transition

# Part F: fault tolerance



- F1 Heartbeat detection
- F2 Replication, configurable frequency
- F3 Promote, retry, no double application
- F4 Crashed workers must not block forever
- F5 Failure matrix

# Experiments you must report

|           |                                    |
|-----------|------------------------------------|
| <b>X1</b> | Is aggregation correct?            |
| <b>X2</b> | Does it scale with servers?        |
| <b>X3</b> | What does relaxed consistency buy? |
| <b>X4</b> | What do optimisations save?        |
| <b>X5</b> | Is elasticity smooth?              |
| <b>X6</b> | Exactly-once under failure?        |
| <b>X7</b> | Harsh failure cases?               |

# How you will get feedback

- **Feedback conversations**, shortly after **submission (December 4, 2026)**
- Nine dimensions, four levels each; no overall score
- Correctness · API ease of use · DS reasoning
- Vector clocks & consistency · communication
- Elasticity · fault tolerance
- Harness & evaluation · code quality & report

Rigorous Parts A–C plus thoughtful D–F beats a superficial attempt at everything.

# How to start, and common pitfalls

## Start here

### Work in groups of 3-5 students

Read the OSDI 2014 paper (§1–5)

Sketch the API before coding

Build the harness with Part A

Commit working states often

## Watch out for

Retries that apply a push twice

Stale ownership maps

Worker failure blocking consistency

Unmeasured optimisations

- Li et al., Scaling Distributed ML with the Parameter Server, OSDI 2014
- Lamport, Time, Clocks and the Ordering of Events, 1978
- Karger et al., Consistent Hashing and Random Trees, 1997
- Ho et al., Stale Synchronous Parallel PS, NeurIPS 2013

## Questions?



THE UNIVERSITY *of* EDINBURGH  
**informatics**

SCHOOL OF LECT

# Distributed Systems (Level 11) - Lecture

E:AW8MZTAKLZ

- Friday, 16 October 2026
  - 16:10 - 18:00
- Lecture Theatre 3

Scan to check-in



Expires at 16/10/2026 18:10