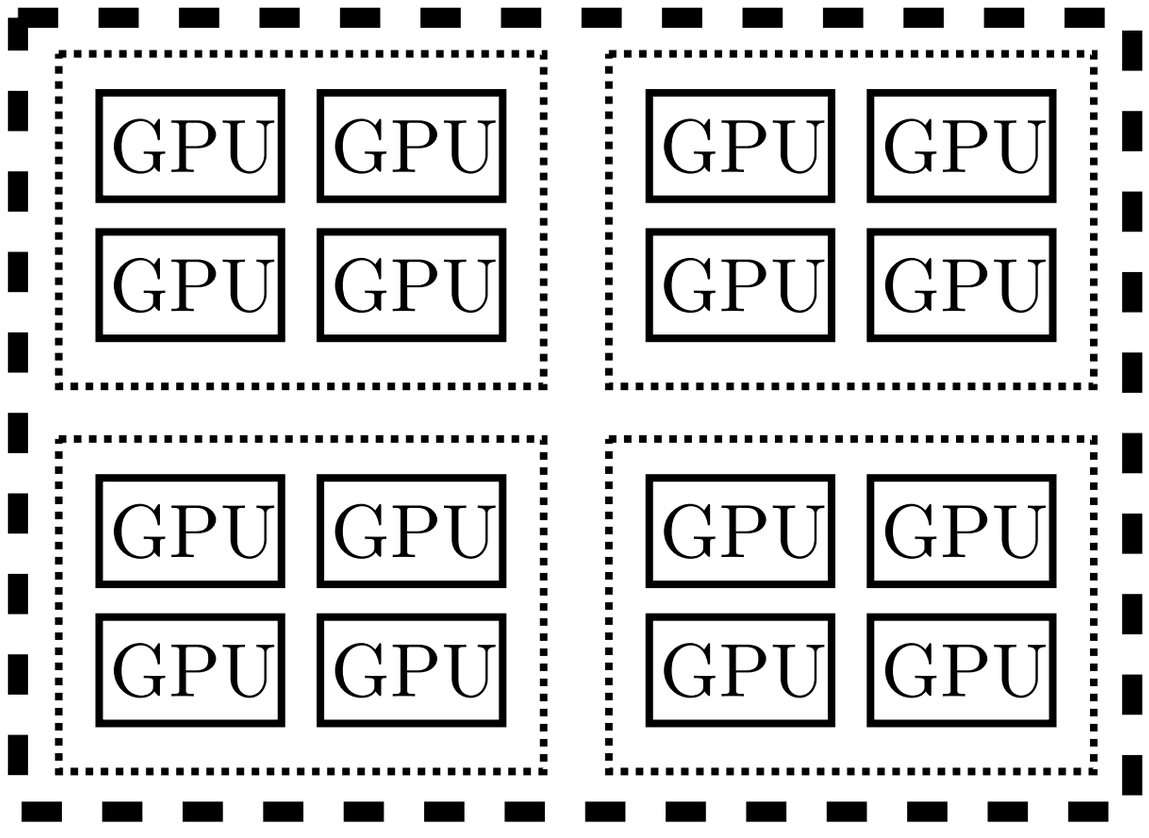
Lecture 4: Communication-Efficient Distributed Optimization

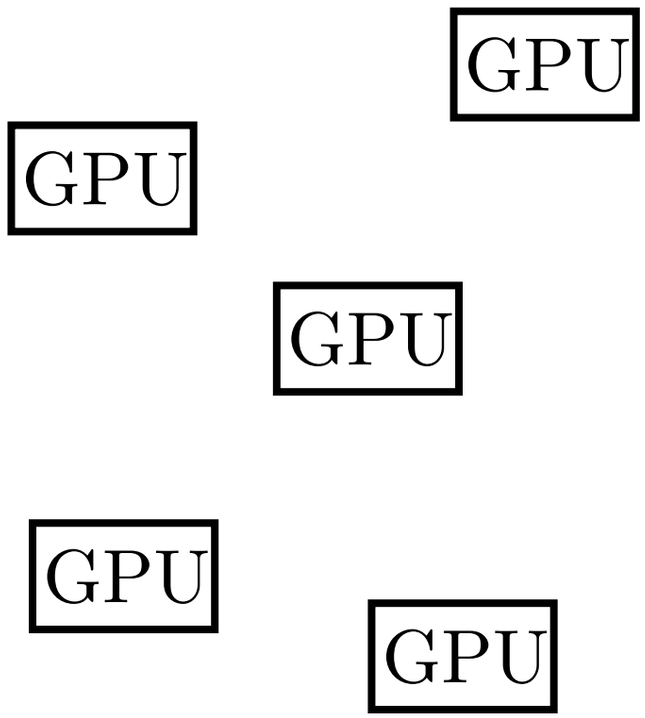
Edouard Oyallon
edouard.oyallon@cnrs.fr
CNRS, ISIR

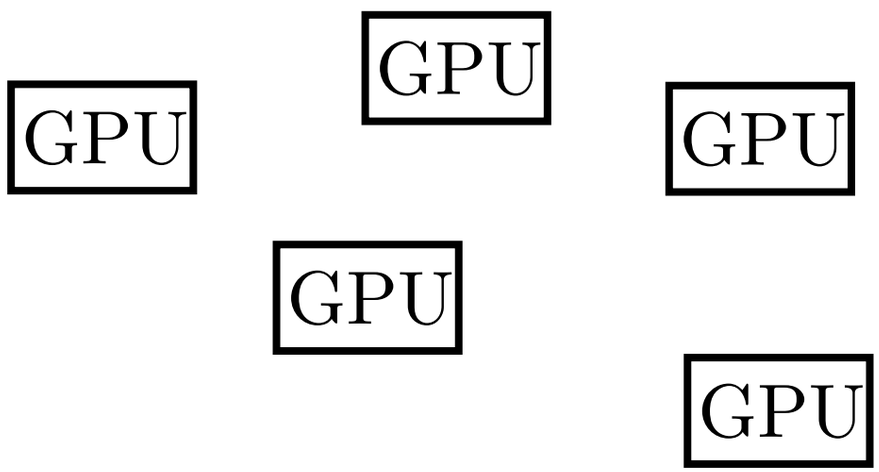
What you will learn today

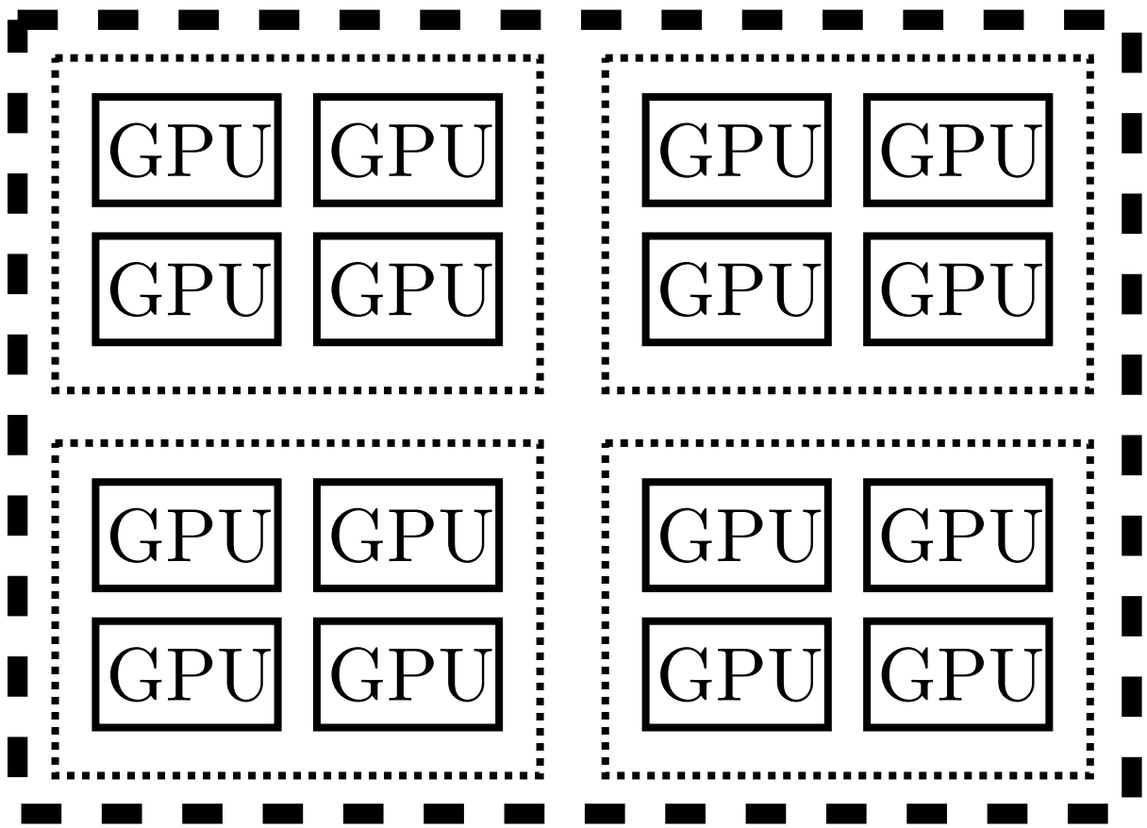
- Communication Topology of GPUs clusters
- Unbiased VS biased compressors, to compress messages
- Local SGD and Diloco, to reduce gradient synchronisation
- Decentralized learning, to scale up LLMs on the internet
- Fault tolerance for training LLMs.

Schematic representations of Communications









Communication issues

- Communication across distant nodes brings two key constraints: bandwidth (how much you can move) and latency (how long each exchange takes). The following numbers vary a lot and are rather order of magnitudes than a ground truth.

- **Intra-node** (GPU↔GPU, NVLink/NVSwitch): **1000GB/s** per GPU.

- **Inter-node** (same cluster, InfiniBand): typically **100GB/s** per node.

- **Across the Internet:** typically **0.1GB/s** per node, and latency also dominates performance.

- **Topology matters a lot:** the switch layout (the actual links between components) changes **congestion**, and therefore the **real** communication cost.

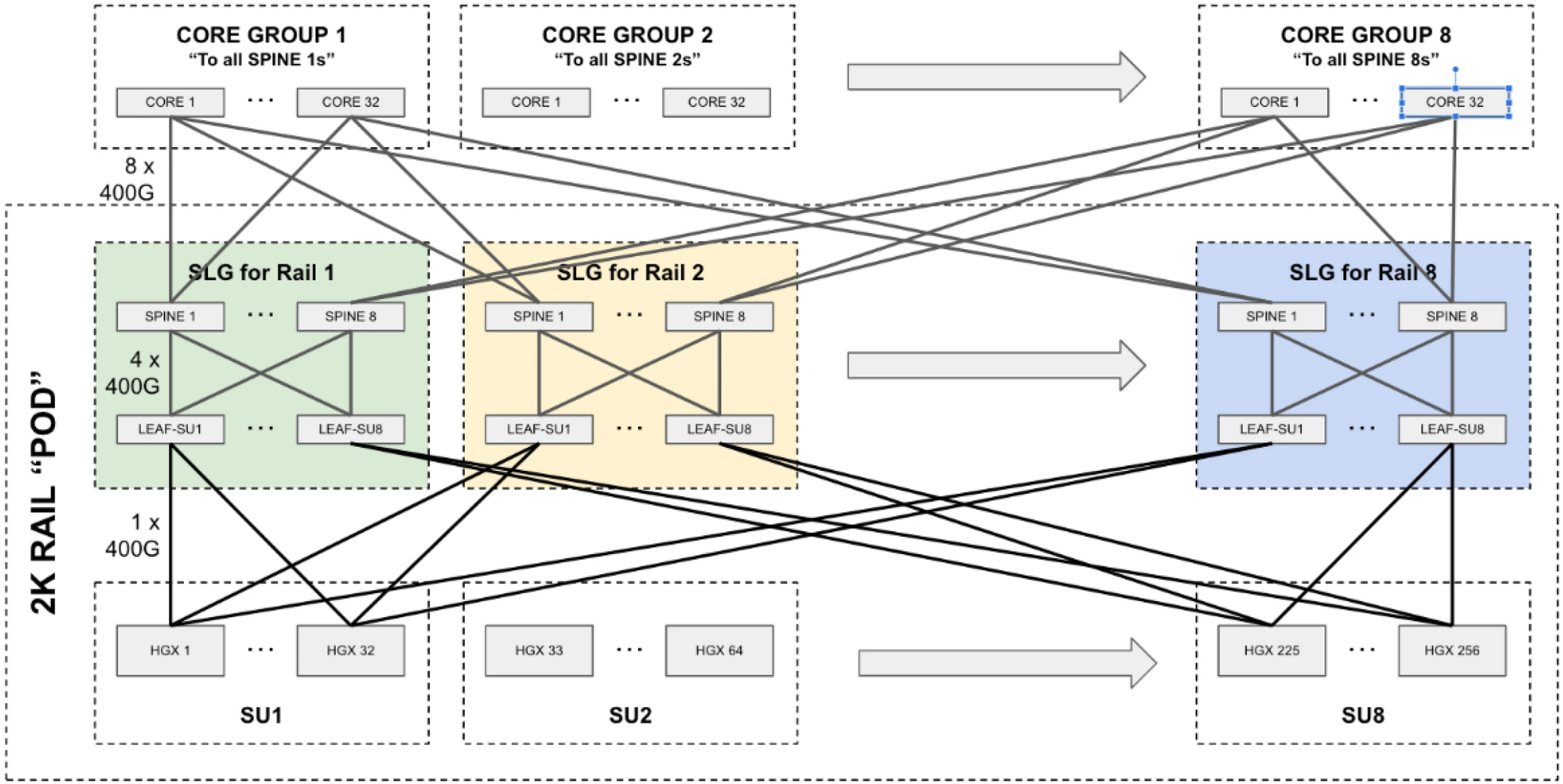
Dissection of a GPU Fabric

HGX: GPU server; SU (Scaling Unit): Group of server

2K RAIL "POD": a single 256x 8 GPUs (an independent network plan)

SLG: Spine-leaf group per rail; Spine: aggregation per rail

Core group = uplink layer, grouped by spine index



- **Performance at 2K-GPU scale:** 8 identical **leaf–spine rails** provide high bandwidth, keeping congestion and latency small.
- **Resilience + operability: Rails** and **SU** compartmentalize failures and maintenance while the core-group→spine-index scales up connectivity; multiple rails let you **add aggregate network capacity** by striping traffic across parallel fabrics.

Reducing the Communication Burden?

- Communication is both an algorithmic and a systems challenge.

- While these ideas can be combined in a single framework, four common strategies to reduce communication overhead are:

  - Reduce *how much* you communicate: e.g., compress what you communicate,

  - Reduce *when* you communicate: e.g., communicate periodically,

  - Reduce *who* communicates: e.g., communicate only to your neighbours,

  - Reduce *what* you communicate: e.g., communicate parameters, not gradients.

The initial algorithm

- Baseline: synchronous data-parallel SGD with n workers, so that each worker applied the same update.

Until convergence, on worker i:

$$\text{sample} \quad \xi_t^i \sim \Xi^i$$

$$g_t^i = \nabla f(\theta_t, \xi_t^i)$$

$$g_t = \texttt{AllReduce}(g_t^1, \ldots, g_t^n)$$

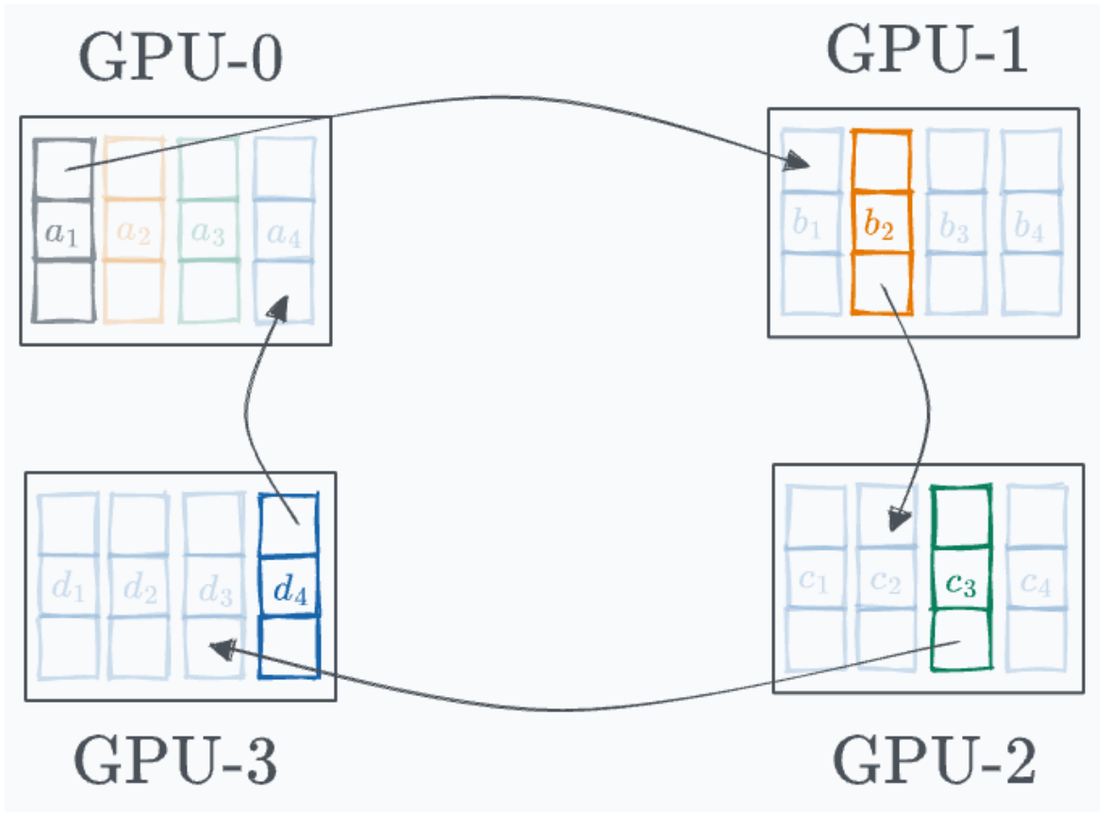
$$\theta_{t+1}, s_{t+1} = \texttt{Optimizer}(\theta_t, g_t, s_t)$$

- **All-Reduce will dominate runtime** when bandwidth is limited or latency is high.

- /!\: we're not necessarily in a Federated Learning setting: no notion of central server here and data distribution might or might not be identical.

All-Reduce Implementation

- **Split** the tensor into N **chunks** (N workers).
- **Phase 1: Reduce-Scatter** (N-1 steps):
  - each step, every worker sends one chunk to its "next" neighbor and receives one chunk from its "previous" neighbor, *reducing (summing) as it goes.*
  - After N-1 steps, each worker holds $1/N$ **of the final reduced result**.
- **Phase 2: All-Gather** (N-1 steps):
  - same ring traffic pattern, but now workers **just forward chunks** to replicate the full reduced tensor everywhere.



(This is ring-All-Reduce, even though NCCL will select the most efficient implementation)

Why reducing communication isn't free

- Without compression, we exchange gradient $g_t^i \in \mathbb{R}^d$ at every steps.
- If we store each entry with b bytes (FP32: $b = 4$, FP16/BF16: $b = 2$), then we need to exchange

$$S = b\,d \quad \text{bytes}$$

- If we send only a fraction $\rho \in (0, 1]$ of the bytes, so

$$S_\rho = \rho S = \rho\,b\,d \quad \text{bytes}$$

- **Trade-off**: smaller ρ faster communication, but may slow convergence (more steps): bytes saved don't automatically mean faster training if we need more steps.
- Most of the techniques will present now are composable together.

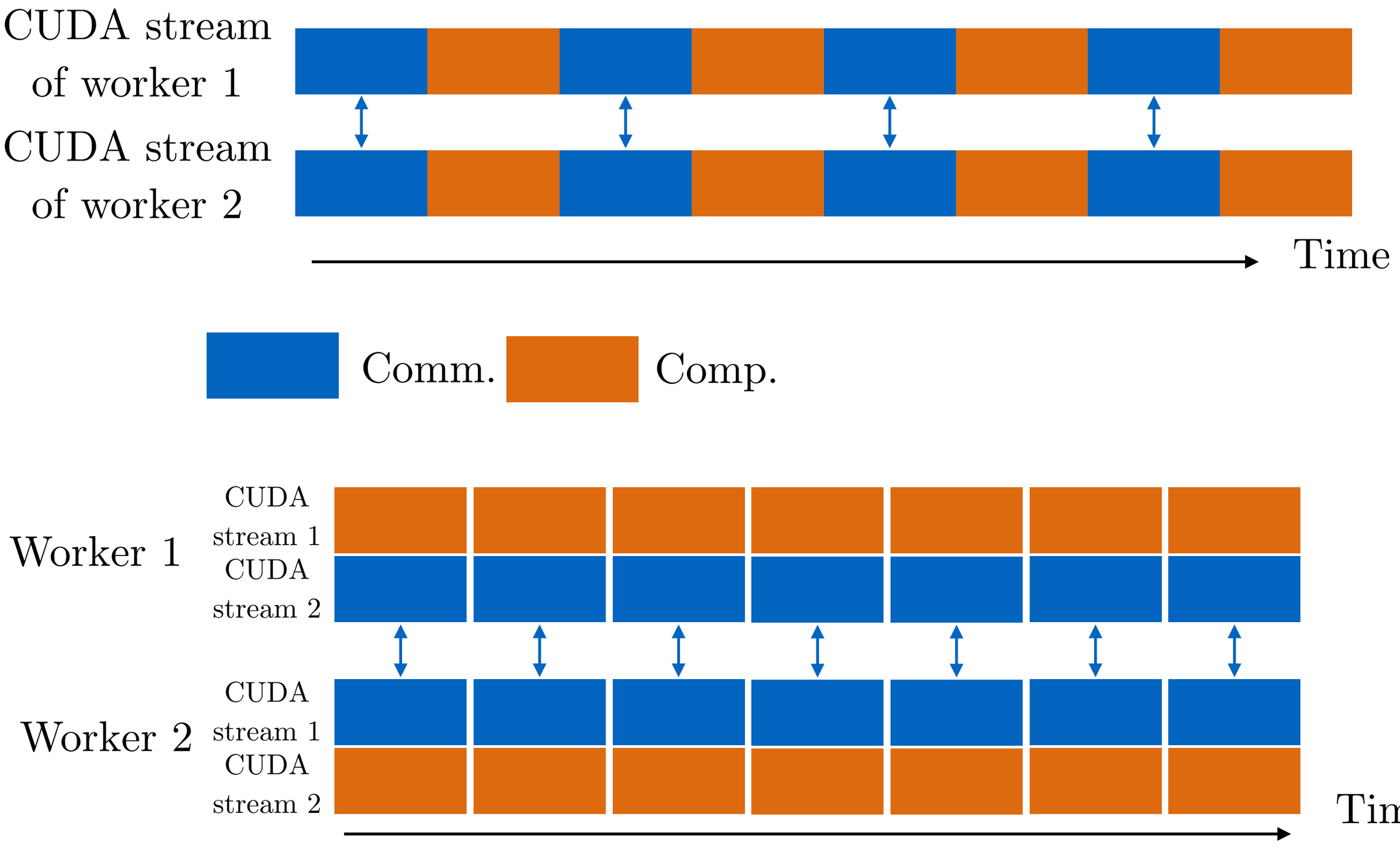
Outline: Reducing Communication Overhead

- Topics we will discuss
  - Communication–computation overlap (latency hiding, compute-bound execution)
  - Message compression (smaller payloads)
  - Local SGD (less frequent global synchronization)
  - Decentralized optimization (local neighbor mixing vs. global coordination)
  - Fault resilience (robustness to failures and performance variance)

Overlapping Communication and Computation

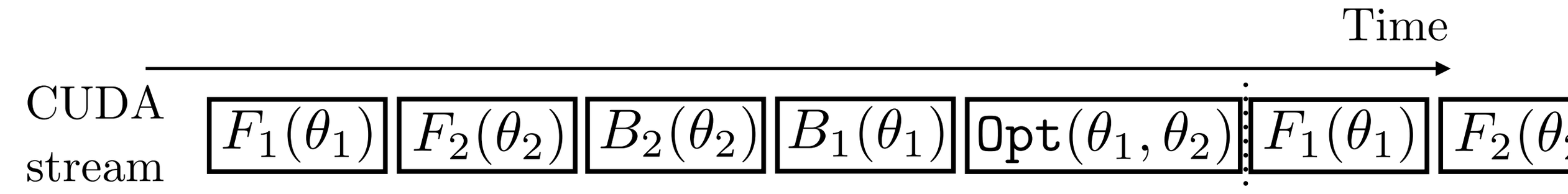
Overlapping Comm and comp synchronously

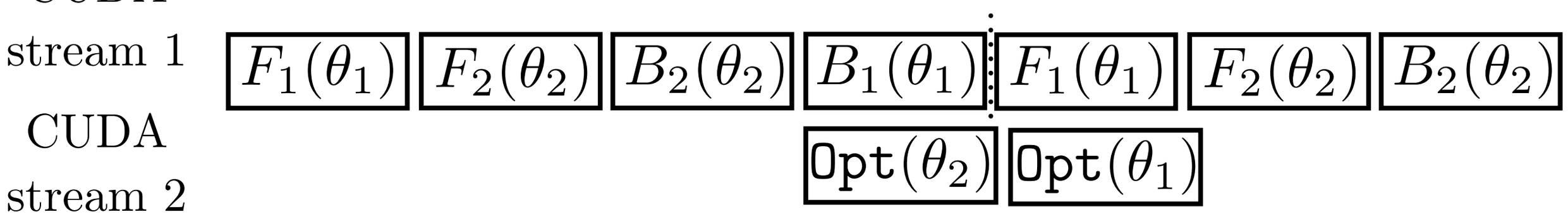
- Communication should be overlapped with computation as much as possible to keep the GPU compute bound.



No Algorithmic Changes: Implementation-Level Comms/Compute Overlap

- Let a network perform a forward pass via F_1, F_2, a backward pass B_1, B_2 and an optimizer step via `Opt`.





- **Overlap via hooks.** Using backward hooks, a standard technique is to launch communication for a layer's gradients immediately after that layer's backward computation completes, overlapping communication with remaining compute.

Delayed Parameter Update

- Maintain two parameter buffers: a compute buffer θ^{comp} and used for forward/backward on GPU, and a **communication buffer** θ^{comm} **used for sync&transfer.**

- **GPU runs SGD on** θ^{comp} while comm runs **in parallel** on θ^{comm} to **maximize overlap** and reduce stalls.

- θ^{comp} tracks θ^{comm} with a **delayed update** (captures offload latency and pipelining).

- Benefit: **hides comm latency**; cost: introduces **staleness which can hurt convergence.**

$$\theta_{t+1}^{\text{comm}} = \texttt{AllReduce}\left(\theta_{1,t}^{\text{comp}}, ..., \theta_{n,t}^{\text{comp}}\right),$$
$$\theta_{i,t+1}^{\text{comp}} = \theta_{t}^{\text{comm}} - \eta \nabla f_i(\theta_{t}^{\text{comm}}).$$

Gradient Compression

Gradient Compression

$$x_t \xrightarrow{C} \text{bits} \xrightarrow{D} \hat{x}_t \quad \text{with} \quad \hat{x}_t = D\big(C(x_t)\big)$$

- A given code is encoded via a compressor and then later decoded via D. The relevant quantities are thus:
  - **Communication cost:** bits / iteration (or #coordinates sent)
  - **Distortion**: keep $\hat{x}_t \approx x_t$.
- A simple idea is to combine both a compressor (and a decompressor). Several choices are:
  - **Top-K sparsification**: keep K largest-magnitude entries (send indices + values)
  - **Rand-K sparsification**: keep K random entries
  - **Sign-SGD** (1-bit): send $\operatorname{sign}(x_t)$
  - **Quantization:** map entries to a small set of levels (b-bit; deterministic or stochastic)

Analysing compressors : biased VS unbiased

Unbiased Compressors	Biased Compressors
$\mathbb{E}[\mathcal{C}(x)] = x$	$\mathbb{E}[\|\mathcal{C}_b(x) - x\|^2] \leq (1 - \frac{1}{\omega_b})\|x\|^2$
$\mathbb{E}[\|\mathcal{C}_u(x) - x\|^2] \leq \omega_u^2 \|x\|^2$	
$\omega_{\text{Rand}\,k} = \frac{d}{k} - 1$	$\omega_{\text{Top}\,k} = \frac{d}{k}$

- For L-smooth objectives, using an unbiased compressor typically slows convergence by a factor ω.

- If n workers apply an **unbiased compressor**, then aggregating the n compressed messages reduces the compression noise variance by $\frac{1}{n}$.

- Deterministic compressors are necessarily biased (so unbiasedness usually requires randomization).

- **Example:** stochastic quantization (random rounding) can be designed to be **unbiased**: $x \in [0,1] \longrightarrow \mathcal{B}(x)$

Error Feedback to Mitigate Compression Error

- There's extensive work on **unbiased vs biased** compression, with strong theoretical guarantees, but practical performance depends on details like **network regime, model scale and optimizer.**

- A simple and effective fix is **Error Feedback (EF)**: keep a **memory** variable that stores what compression dropped, then add it back at the next step.

- **Intuition**: "don't throw information away: delay it."

- EF is principled, and, in practice, EF is often robust and enables **much higher compression ratios** with little loss in final accuracy.

$$\hat{g}_t^i = \mathcal{C}(g_t^i + e_t^i) \quad // \text{ will be sent}$$

$$e_{t+1}^i = g_t^i + e_t^i - \hat{g}_t^i \quad // \text{ is stored locally on worker } i$$

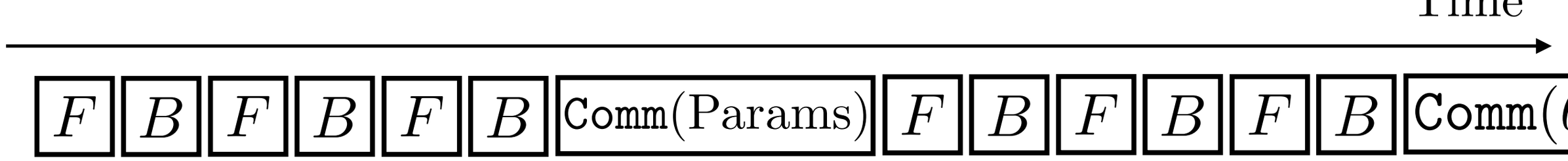
$$\theta_{t+1} = \theta_t - \frac{\eta}{n} \sum_{i=1}^{n} \hat{g}_t^i \quad // \text{ compressor is All-Reduced}$$

We'll discuss it in few slides.

Local SGD and Variants

Local SGD

- The goal of Local SGD is to do multiple local steps (K local steps between synchronizations) to **save communication** (sync less often).



- With **Local SGD,** replicas see the same data distribution, yet models **drift apart**: each client moves toward its own local optimum. (if workers see different distribution, this is also referred to as *Federated Learning*)

- Compare to **SGD** at the **same number of samples processed**; the aim is to match the **final accuracy** while communicating less.

- Local SGD is **hard to combine with sharding**: optimizer states must stay **resident on each worker**; "re-materializing" them would require extra communication: exactly what Local SGD is trying to avoid.

Local SGD algorithmic block

- Each worker keeps its **own local optimizer state** (not synchronized between workers).
- Communication happens only every K steps, so the synchronization frequency, and thus the communication volume for global sync, is reduced by roughly a factor of K.

Until convergence:

$$\theta^i_{t,0} = \theta_t$$

$$\text{for } k = 0, \ldots, K-1 :$$

$$\text{sample } \xi^i_{t,k} \sim \Xi_i$$

$$g^i_{t,k} = \nabla f(\theta^i_{t,k}, \xi^i_{t,k})$$

$$\theta^i_{t,k+1}, s^i_{tK+k+1} = \texttt{Optimizer}(\theta^i_{t,k}, g^i_{t,k}, s^i_{tK+k})$$

$$\theta_t = \texttt{AllReduce}(\theta^1_{t,K}, \ldots, \theta^n_{t,K})$$

/!\: you can't average gradients! why?

From Local SGD to DiLoCo

- Local SGD works well in vision benchmarks (e.g., **CIFAR-10**); **LLMs are more fragile** (drift hurts stability).
- **DiLoCo**: add a **global stabilizer** on top of local training:
  - **inner optimizer** (often **Adam**) for local steps
  - **outer optimizer** (often **SGD + momentum, often kept at recommended defaults - in `torchtitan` their values are fixed**) for synchronized updates, which adds global control over local Adam
- Trade-offs:
  - can improve convergence vs naïve Local SGD
  - reported diminishing returns beyond **8 workers and it requires an extra optimizer state.**
- **Best fit:** multi-island / geo-distributed training where interconnect is slow or unreliable, memory is huge, so reducing synchronization frequency matters most

DiLoCo

- DiLoCo = inner optimizer + outer optimizer.

- **Avoid sharding optimizer state** if you can — it complicates scaling and adds overhead.

Until convergence:

$$
\begin{array}{l}
\left|\begin{array}{l}
\theta^i_{t,0} = \theta_t \\
\text{for } k = 1, \ldots, K: \\
\quad \left|\begin{array}{l}
\text{sample } \xi^i_{t,k} \sim \Xi_i \\
g^i_{t,k} = \nabla f(\theta^i_{t,k}, \xi^i_{t,k}) \\
\theta^i_{t,k+1}, s^i_{tK+k+1} = \texttt{Optimizer}^{\text{in}}(\theta^i_{t,k}, g^i_{t,k}, s^i_{tK+k})
\end{array}\right. \\
g_t = \dfrac{1}{n} \displaystyle\sum_{i=1}^{n} (\theta^i_{t,K} - \theta^i_{t,0}) \\
\theta_{t+1}, s_{t+1} = \texttt{Optimizer}^{\text{out}}(\theta_t, g_t, s_t)
\end{array}\right.
\end{array}
$$

Streaming DiLoCo

- **Stream partial outer updates (by parameter fragments)**: Split the model into P stages, yet instead of syncing *all* parameters every K steps, **sync one fragment at a time** on a staggered schedule. **Peak bandwidth drops** because each sync only moves a fraction of the model while total bandwidth is unchanged.

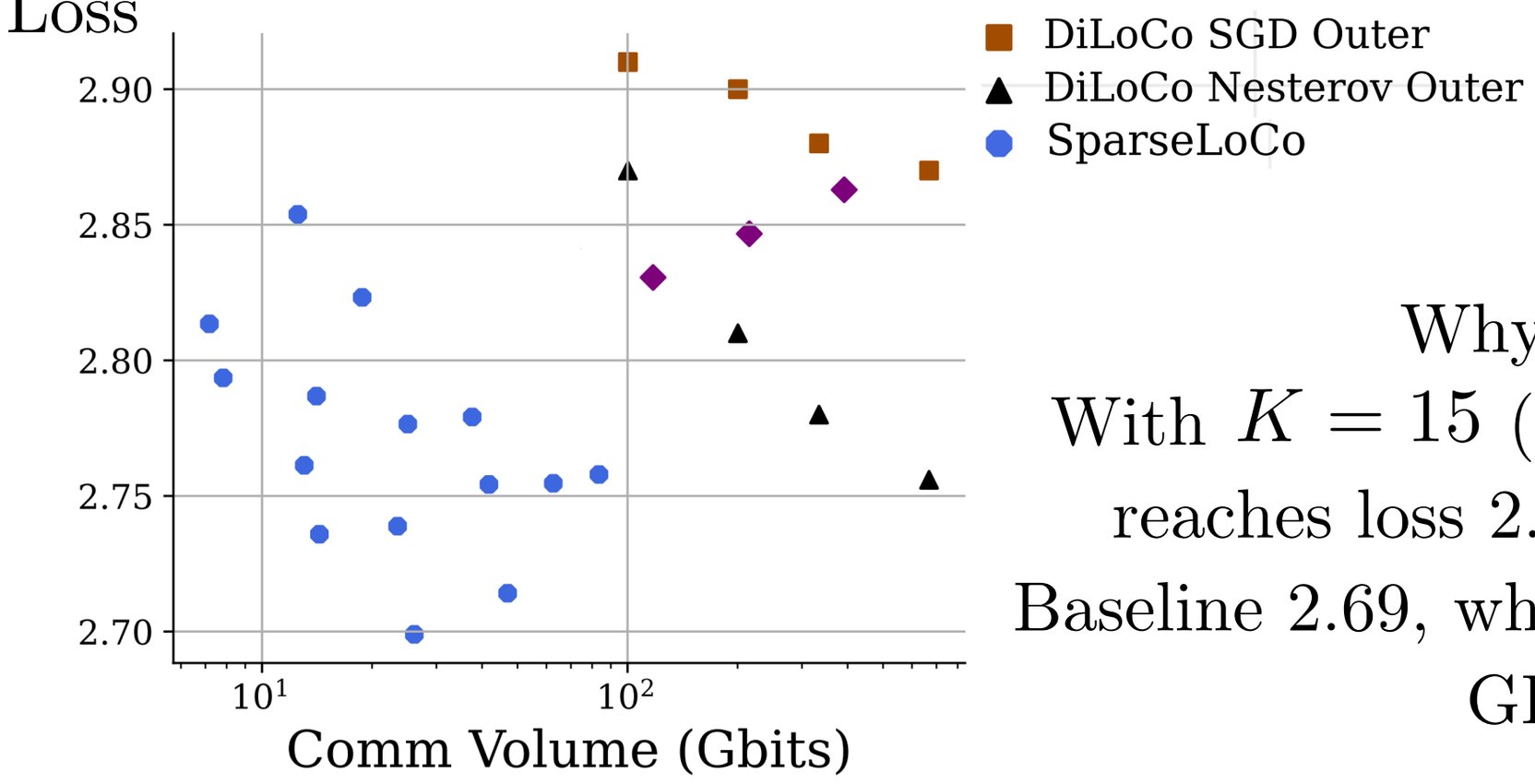
- **Overlap communication with computation**: Start the next inner steps immediately, then **apply the received outer update after a delay** τ (that represent communication time), mixing it into the current local fragment with factor α, so that if an update occurs at time (t, k) then, on worker i and for fragment p, do

$$\theta^i_{p,k,t+1} \;=\; (1-\alpha)\,\theta^i_{p,k,t} \;+\; \alpha\,\tilde{\theta}^{\tau}_{p}, \qquad \alpha \in [0,1].$$

  The non streaming case corresponds to $(\tau, \alpha) = (0, 0)$.
- At any moment, **different fragments are at different synchronization ages**, so the model is **partially stale**, which can hurt convergence

SparseLoco

- In bandwidth-limited settings (cross-datacenter / internet), *even infrequent sync methods* still often **communicate dense pseudo-gradients**, becoming the bottleneck.
- **SparseLoCo** = DiLoCo-style local training + aggressively compressed pseudo-gradients:
  - Top-k sparsification (as low as 1–3% density) + 2-bit quantization
  - Replace outer momentum with error feedback (OuterEF) so the EF accumulator acts like a local approximation of outer momentum, letting you drop *global* outer momentum.



Why it matters?

With $K = 15$ (163 syncs), SparseLoCo reaches loss 2.70 vs DiLoCo 2.76 vs Baseline 2.69, while sending 17 MB vs 0.5 GB vs 1GB.

When All-Reduce Becomes the Problem

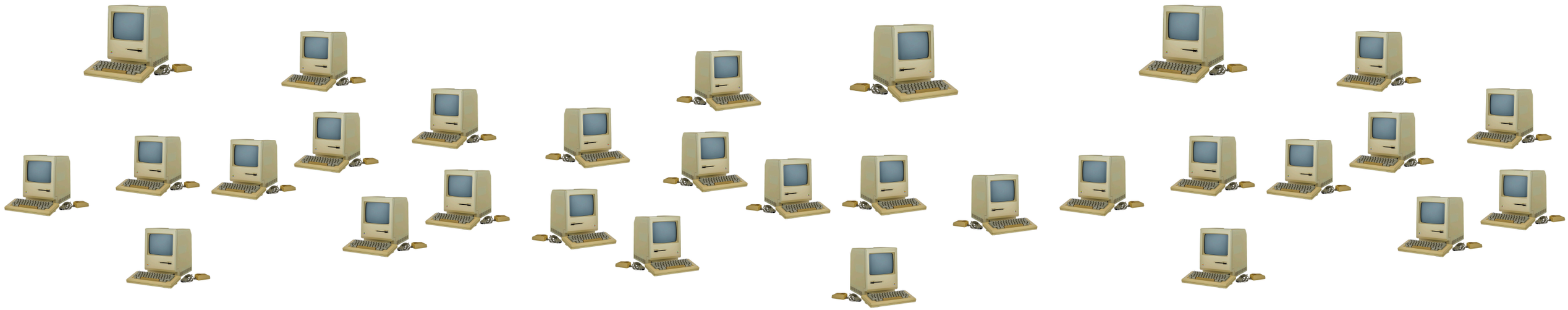
- **Local-SGD suggests a key lesson:** *gradient* communication is often a poor use of bandwidth—high frequency, high volume, and tightly coupled to the critical path.

- **A more communication-efficient viewpoint** is to exchange **model states** (parameters / deltas / low-rate model snapshots) rather than raw gradients, enabling larger, less frequent messages that are easier to overlap and tolerate delay.

- Scaling pain point: even with optimized collectives, All-Reduce overhead grows with the number of n workers (e.g. $\log n$ latency terms + bandwidth costs). Can we **amortize coordination further** by reducing the need for global synchronization?

Decentralized Communications

From Centralized to Decentralized Learning



- **Many devices can collaborate to train a model,** from smartphones and gaming PCs to GPU clusters.

- **Centralized training assumes a global coordinator** (e.g., a parameter server or global All-Reduce) that orchestrates communication and synchronization.

- **Decentralized training removes the single global bottleneck:** workers **compute and communicate locally** (e.g., with neighbors), which can improve scalability and robustness.

- You can view centralized training as a special case of decentralized training—the difference is the communication pattern and available bandwidth (global all-to-all vs. local links).

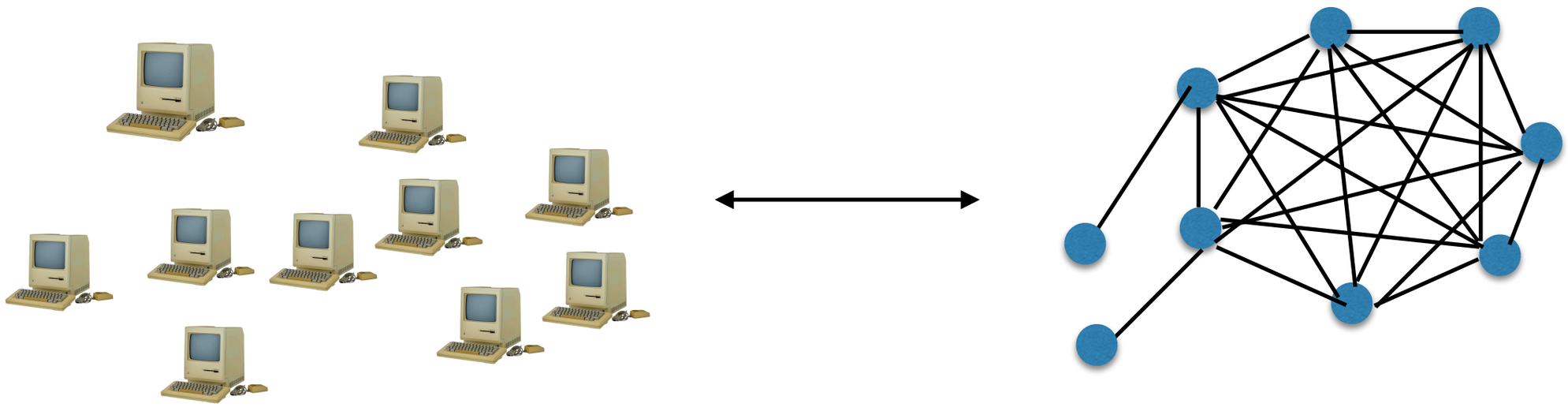
Decentralized Learning

The objective function explicitly decouples variables across workers:

$$\inf_{\theta \in \mathbb{R}^d} \sum_{i=1}^{n} f_i(\theta) \iff \inf_{\theta_i \in \mathbb{R}^d, \theta_i = \theta_1} \sum_{i=1}^{n} f_i(\theta_i)$$

with e.g., f_i the loss of the i-th worker.

And the connectivity between workers is explicitly modelled:



Communication graph

- **Assumption 1**: Local data, ∇f_i and local memory buffer available only on a given node.

- **Assumption 2**: Only adjacent nodes can communicate.

Decentralised Learning Benefits

- **No global bottleneck:** replaces global All-Reduce / coordinator with **local neighbor exchanges**, improving scalability as the number of workers grows.

- **Better overlap + robustness:** communication is **local and parallel**, so delays/stragglers or a single failing link have a smaller blast radius.

- **Topology-aware efficiency:** you can match the algorithm to the **physical network** (who can talk to whom), reducing long-distance traffic.

- **Natural "consensus" objective:** the goal becomes agreeing on a common model using only local links—and that agreement is precisely captured by the **graph Laplacian**.

The Graph Laplacian

Let some edges $\mathcal{E} \subset \{1, ..., n\} \times \{1, ..., n\}$ with

and fix e_i : i-th canonical vector of $\mathbb{R}^n$ $\quad (i,j) \in \mathcal{E} \Leftrightarrow (j,i) \in \mathcal{E}$

- Define the Laplacian as:

$$\Lambda = \sum_{(i,j)\in\mathcal{E}} (e_i - e_j)(e_i - e_j)^T \qquad \text{so that} \quad \Lambda \succcurlyeq 0$$

Remind that:

$$\text{(i)} \;\; u^T \Lambda u = \sum_{(i,j)\in\mathcal{E}} (u_i - u_j)^2 \qquad \text{(ii)} \;\; \Lambda \mathbf{1} = 0$$

$$\text{(iii)} \quad \operatorname{Ker} \Lambda = \operatorname{Span}(\mathbf{1}) \;\; \text{iff} \;\; \mathcal{E} \text{ is a connected graph} \qquad \text{(iv)} \; \operatorname{Tr} \Lambda = 2|\mathcal{E}|$$

- Define the spectral gap (for connected graphs), i.e., conditioning number of Λ as:

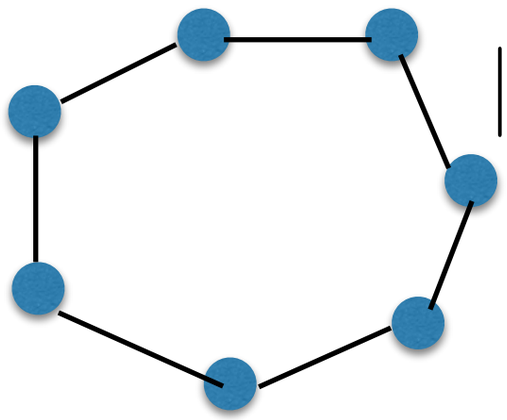
$$\gamma \triangleq \|\Lambda\| \|\Lambda^+\| \geq 1$$

- Spectral gaps are measures of *good* connectivity.

Spectral gap of well-known graphs

Line/circle

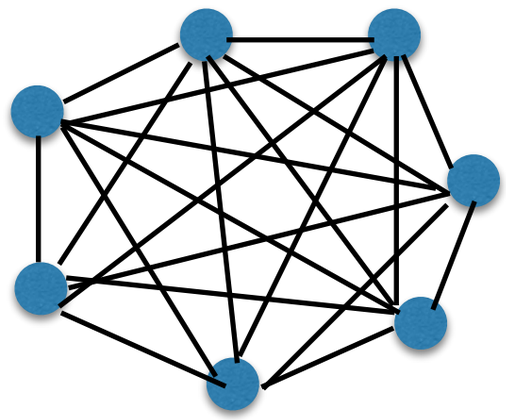
$$\|\Lambda^+\| = n^2$$



$$\|\Lambda\| = 1$$

$$|\mathcal{E}| = n$$

Complete $\|\Lambda^+\| = 1$



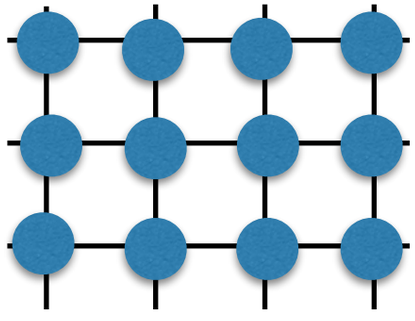
$$\|\Lambda\| = 1$$

$$|\mathcal{E}| = n^2$$

Which graphs have a *good* connectivity?

d-connectivity

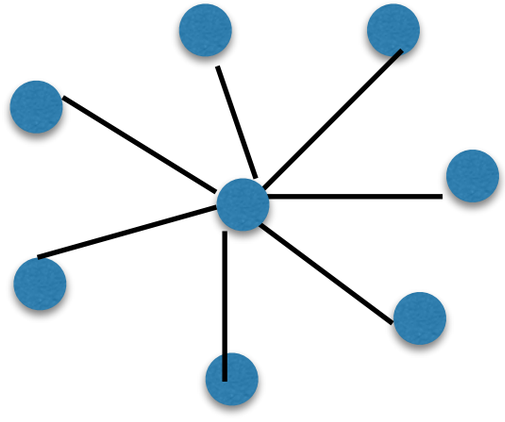
$$\|\Lambda^+\| = n^{2/d}$$



$$\|\Lambda\| = 1$$

$$|\mathcal{E}| = dn$$

Star-graph



$$\|\Lambda^+\| = 1$$

$$\|\Lambda\| = n$$

$$|\mathcal{E}| = n$$

A simple use case: Gossip Algorithms

- Define the degree as $d_i = \#\{j, (i,j) \in \mathcal{E}\}$
- And consider the following averaging procedure:

$$x_{t+1}^i = \frac{1}{d_i} \sum_{(i,j)\in\mathcal{E}} x_t^j = x_t^i - \frac{1}{d_i} \sum_{(i,j)\in\mathcal{E}} (x_t^i - x_t^j) = x_t^i - \frac{1}{d_i} \sum_{(i,j)\in\mathcal{E}} (e_i - e_j)^T x_t$$

- This can equivalently written as $x_{t+1} = (\mathbf{I} - \frac{1}{2} D^{-1}.\Lambda) x_t$
- Since we have $(e_i - e_j)(e_i - e_j)^T \prec 2(e_i e_i^T + e_j e_j^T) \Rightarrow \Lambda \prec 2D$

We know that this procedure admits a minimiser $x^* \in \arg\inf_x \|\Lambda x\|^2$

If the graph is connected, then $x^* = \frac{1}{n} \sum_{i=1}^n x_i$ and ϵ -precision is reached after

$$\mathcal{O}(\log \frac{1}{\epsilon} \|\Lambda\|.\|\Lambda^+\|) \text{ steps.}$$

Decentralised SGD

$$y_t = x_t - D^{-1}\Lambda x_t$$
$$x_{t+1} = y_t - \eta \nabla F(y_t)$$

where $\quad [\nabla F(y)]_i = \nabla f_i(y_i)$

- Decentralized SGD can be performed in two steps:
  - **Step 1 (Consensus):** each node replaces its model by a **weighted average of neighbors** using a Laplacian-based update.
  - **Step 2 (Local learning)**: each node takes an SGD step on its local objective using the mixed model
- **Interpretation:** mixing reduces disagreement across nodes; the gradient step improves the objective, together they trade off **consensus vs. progress**.

Fault Tolerance

Faults Are the Norm in Distributed Training

- **Failures are frequent:** GPU/node crashes, network hiccups, NCCL timeouts, stragglers, and storage/I/O stalls can cause **job crashes** or **multi-minute pauses**.

- **Baseline recovery = checkpoint & restart:** periodically save state, then resume from the latest checkpoint after a failure.

- **To resume faithfully, checkpoint more than weights:** model parameters + optimizer state + LR scheduler + random seeds + dataloader/sampler state (otherwise training diverges or repeats data).

- **Why it matters even more in peer-to-peer settings:** connectivity is inherently less reliable (devices can drop at any time).

- **But it's not just P2P:** even in well-managed clusters, transient network/storage issues make failures and slowdowns **common enough** that resilience is a first-class concern.

The node failure on clusters

Ref.: The Llama 3 Herd of Models, Llama Team, 2024

Component	Category	Interruption Count	% of Interruptions
Faulty GPU	GPU	148	30.1%
GPU HBM3 Memory	GPU	72	17.2%
Software Bug	Dependency	54	12.9%
Network Switch/Cable	Network	35	8.4%
Host Maintenance	Unplanned Maintenance	32	7.6%
GPU SRAM Memory	GPU	19	4.5%
GPU System Processor	GPU	17	4.1%
NIC	Host	7	1.7%
NCCL Watchdog Timeouts	Unknown	7	1.7%
Silent Data Corruption	GPU	6	1.4%
GPU Thermal Interface + Sensor	GPU	6	1.4%
SSD	Host	3	0.7%
Power Supply	Host	3	0.7%
Server Chassis	Host	2	0.5%
IO Expansion Board	Host	2	0.5%
Dependency	Dependency	2	0.5%
CPU	Host	2	0.5%
System Memory	Host	2	0.5%

Table 1: **Root-cause categorization of unexpected interruptions** during a 54-day period of *Llama 3 405B* pre-training. About 78% of unexpected interruptions were attributed to confirmed or suspected hardware issues.

Reliability & Challenges at 16K-GPU LLM Training

- **Harder than CPU clusters**: 16K-GPU training has *more failure modes* than much larger CPU fleets; synchronous training means one GPU fault can restart the whole job.

- Failures are frequent at scale (on a 54-day snapshot):

  - **466** total job interruptions (**47 planned**, **419 unexpected**)

  - **78%** of unexpected interruptions were **confirmed hardware**

  - GPU issues dominate: 58.7% of unexpected issues

  - Only **3** cases needed significant **manual intervention**; the rest handled by automation

- Improving recovery mattered: reduced startup + checkpoint time

- **Stragglers hurt everyone:** one slow node can **slow thousands of GPUs.**

On the necessity to Checkpoint wisely

- **LLM-specific pain point:** optimizer state are typically **massive** (often larger than the model weights), making checkpoints heavy and slow.

- **Sharding reality:** checkpoints are often **sharded across ranks**; resuming may require either

  - restarting with the **same sharding**, or
  - using tooling to **reconstruct full weights** (or convert between formats).

- **Reshardable checkpoints:** store state so it can be **repartitioned** when you reload (not tied to a fixed parallelisation grid), this is the point of `torch.distributed.checkpoint()`.

- **Why it matters:** the ability to resume with different **DP** / **TP** / **PP** degrees enables **faster recovery** and **elastic scaling** when the cluster returns with a different GPU count or topology.

How is Checkpointing implemented?

- A **background thread** writes checkpoints while **training continues** on the next steps.

- Checkpoint data is **staged from GPU → CPU** before being persisted to storage.

- To maximize overlap, `torch.distributed.checkpoint()` offloads state-dict creation + GPU→CPU copy to a background thread.

- Each checkpoint is sharded into multiple files, typically at least one file per rank.

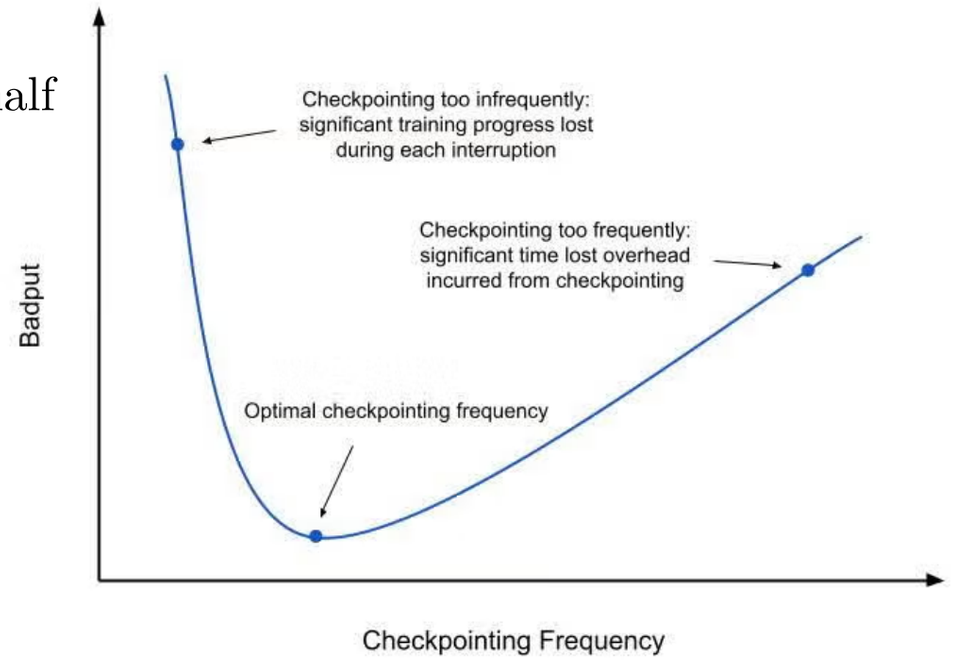
- The stager operates **in-place**: the model allocates storage first, and checkpointing **reuses that storage** via `Sateful` **objects** (objects that can save and restore a given training moment).

Checkpointing Tradeoffs

If uniform, you lose half

MTBI: mean time between interruptions

$$\texttt{ckpt\_badput\_time} = \underbrace{\texttt{ckpt\_loading\_time}}_{\text{loading}} + \underbrace{\frac{\texttt{MTBI}}{\texttt{ckpt\_interval}}\,\texttt{ckpt\_saving\_overhead}}_{\text{saving overhead}} + \underbrace{\frac{\texttt{ckpt\_interval}}{2}}_{\text{computation loss}}$$



- **Loading**: time to load checkpoint from storage when recovering from interruption

- **Saving Overhead**: overhead on training from saving checkpoints

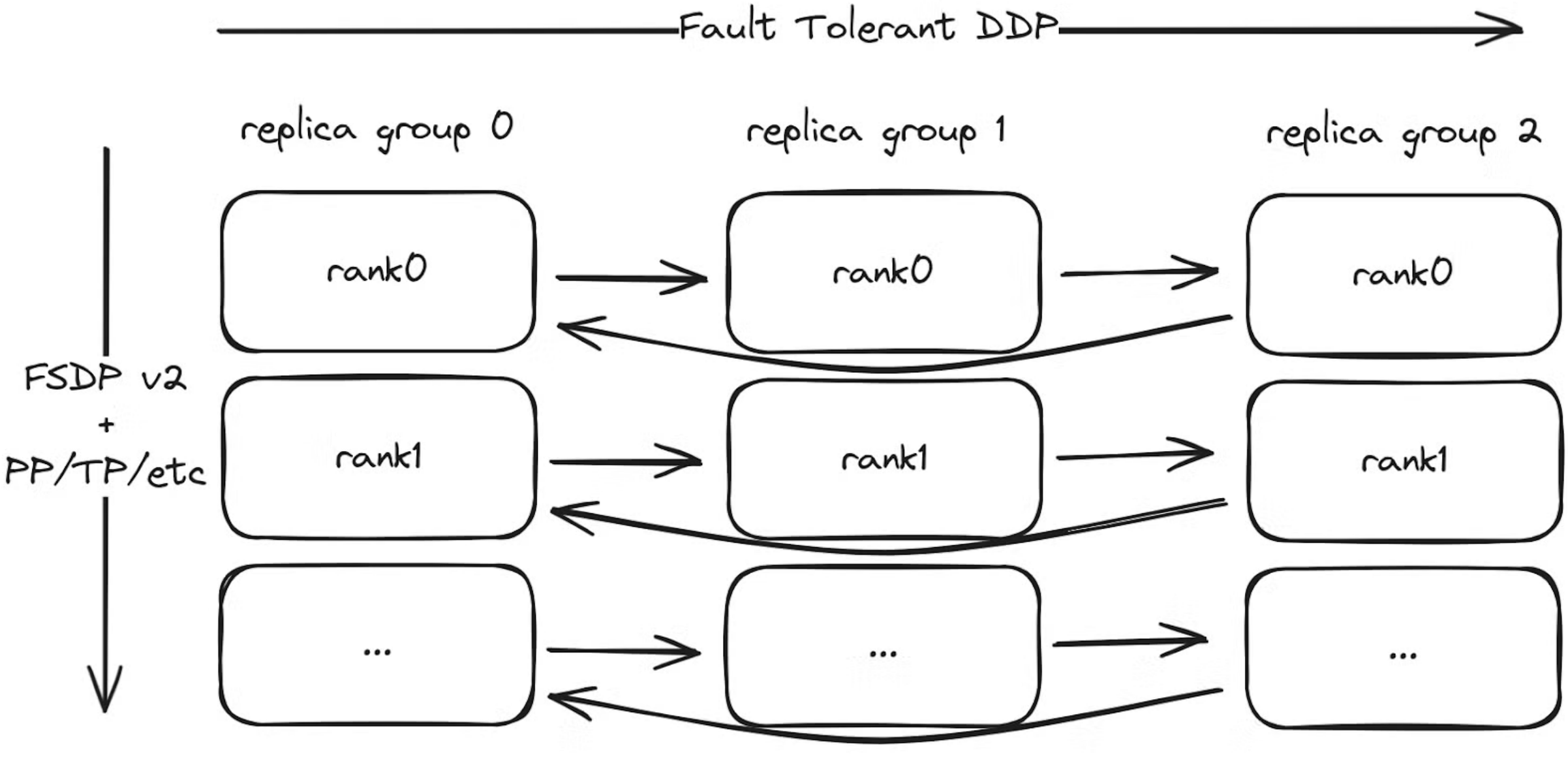
- **Computation Loss**: computation time lost when resuming from the most recent checkpoint

- **Size overhead**: frequent checkpointing requires a significant amount of storage which can be expensive.

Almost free fault tolerance checkpointing

- Assume at least one replica stays healthy: it acts as the *source of truth* for the latest model state.

- **Fault-tolerant DDP recovery:** the recovering worker **peer-to-peer pulls the weights** (optionally optimizer state) from that healthy replica to rejoin quickly.



Dealing with Stragglers

- **Synchronous DDP all-reduce**: each step is gated by the slowest rank (straggler effect).

- **Pipeline parallelism:** a slow stage creates **pipeline bubbles** and **backpressure** across stages.

- **Diagnosis is tricky at scale:** mixed fabrics + large process groups blur whether the bottleneck is **compute, network, or a flaky device**.

- Mitigation: detect and localize stragglers, then isolate&replace them or shrink the group to keep throughput stable.

- **Forward-looking: elastic membership** (variable world size via re-rendezvous) to continue training despite churn. (quite beyond the scope of those lectures)

TorchFT

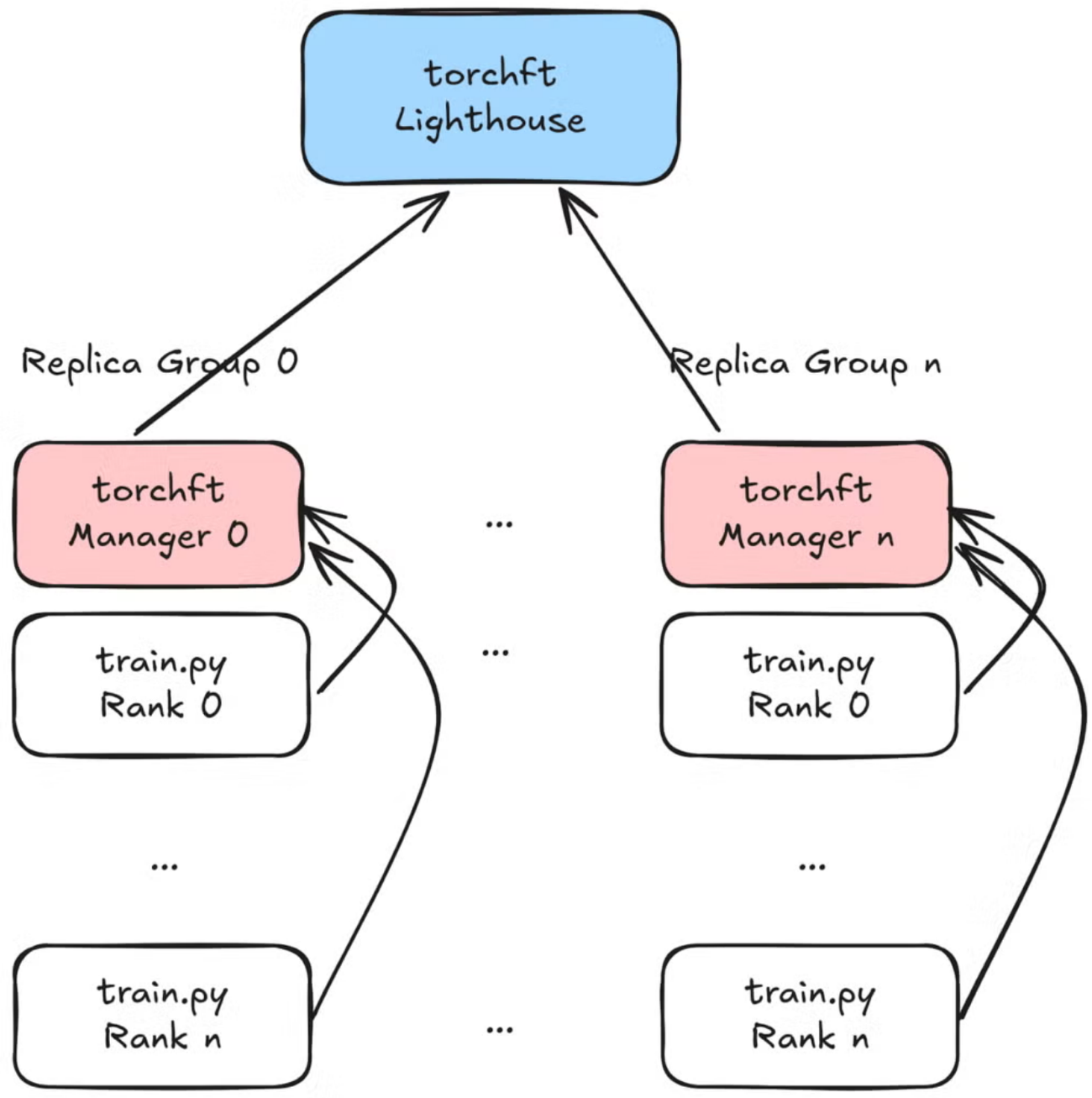
TorchFt

- TorchFT is compatible with TorchTitan and implements features like
  - **Health & coordination:** per-step **heartbeating** to determine which workers are healthy/unhealthy.
  - **Fault-tolerant collectives: ProcessGroup** implementations that surface errors clearly and can be **reinitialized** cleanly.
  - **Live recovery transport: checkpoint/state transfer** from a healthy peer for fast recovery (and some scale-up flows).
  - **Peer discovery & topology:** mechanisms like **Lighthouse** to find and connect to healthy replicas.

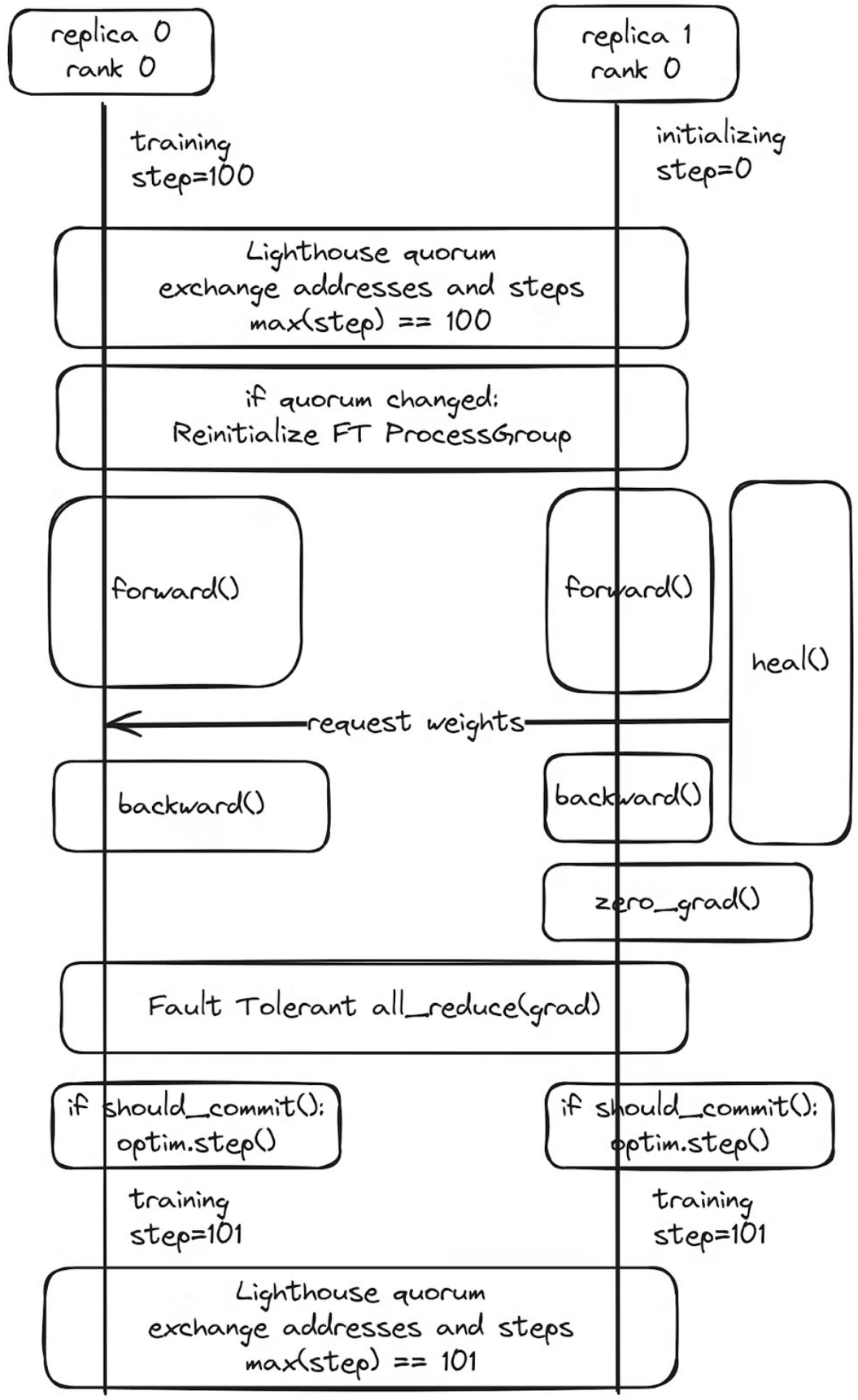
Lighthouse or Parameter Server

- **Lighthouse mode:** a coordinator tracks the **set of healthy workers** and helps peers discover each other for **P2P recovery/transfers**.

- **Parameter-server mode**: TorchFT provides a fault-tolerant parameter server built on reconfigurable ProcessGroups (which stores parameters), and it doesn't require Lighthouse.



FT without Checkpointing



Conclusion

- Now, let's try out our collaborative lab using TorchTitan and TorchFT