



NCCL Watchdog Timeout: Hang Triage with Flight Recorder, RAS

Plan and build private AI compute with GPU Smith. We connect workload requirements with GPU hardware, networking, deployment and operating decisions so your infrastructure fits the work it must perform.

Published by GPU Smith · 2026-10-20 · nccl watchdog timeout · nccl hang debug · pytorch flight recorder · nccl ras · nccl timeout troubleshooting · distributed training hang debugging · fr_trace analysis · nccl debug environment variables · nccl all_reduce hang

Original article: <https://gpusmith.com/articles/en/nccl-hang-watchdog-timeout-triage>



In this article

1. Executive Summary
 2. Introduction and Background
 3. Anatomy of a Watchdog Timeout
 4. Before Failure: Instrumentation That Survives the Incident
 5. At the Stall: Live Evidence Capture
 6. After the Timeout: Flight Recorder Analysis
 7. Triage Branches
 8. Reproduction Ladder and Disposition
 9. Incident Bundle and UTC Timeline
 10. Data Analysis and Evidence
 11. Case Studies and Real-World Examples
 12. Implications and Future Directions
 13. Frequently Asked Questions (FAQs)
 14. Conclusion
-

Executive Summary

An **NCCL watchdog timeout** is a symptom report, not a diagnosis. PyTorch's own engineers describe it as a catch-all that fires on "anything that can cause a rank to wait indefinitely on a collective" (pytorch.org), and they state that "the rank that first raises the NCCL watchdog timeout is rarely the culprit" (pytorch.org). The error is raised by the PyTorch watchdog thread rather than by the NCCL library (pytorch.org), and NCCL itself performs no argument checking, so mismatched collectives simply hang. The first job of an incident commander is therefore to preserve evidence from every rank, not to change a timeout.

This report is an incident-evidence runbook pinned to the versions documented as of October 2026: **NCCL 2.32.3** (docs.nvidia.com) and **PyTorch 2.14.0** (docs.pytorch.org). Three instruments carry most of the load. The PyTorch **Flight Recorder** is a per-rank, CPU-side ring buffer shared across all process groups (pytorch.org) that writes one file per rank on timeout (docs.pytorch.org). **NCCL RAS** (reliability, availability, serviceability), available since NCCL 2.24, answers live queries on localhost port 28028 and flags communicator ranks whose collective counts disagree. Host evidence (kernel Xid lines, RDMA counters, scheduler placement) closes the loop.

The key reading rules are quantified. PyTorch's NCCL default timeout is 10 minutes (docs.pytorch.org). A RAS mismatch that does not change across repeated queries suggests a stuck communicator, and RAS declares an unreachable process permanently dead after 60 seconds. Meta's published fleet experience attributes more than 60 percent of timeouts across its four training stacks to CPU-side desynchronization (pytorch.org) and between 20 and 30 percent to network or hardware issues (pytorch.org); these are vendor-attributed figures, not independent measurements. RAS diagnostics explicitly "do not exercise NCCL data paths" (docs.nvidia.com), so a clean RAS report is not proof that the fabric is healthy.

The recommended disposition logic is evidence-gated: retry only when dumps from all ranks show a recoverable pattern; drain or quarantine a node only with hardware signals (for example an NVLink Xid or a PCIe fall-off-the-bus Xid) or repeat failures on the same hardware; escalate to application owners when Flight Recorder shows a missing rank or a type, size or order mismatch; and record "unknown" when files are missing. No universal timeout, buffer size or retry count is endorsed here; those are policy values for the operator to set and justify.

Introduction and Background

Distributed training jobs fail in a characteristic way. One rank logs a watchdog timeout, the process group aborts, the scheduler restarts the attempt, and the evidence that would have identified the cause disappears with the processes. By the time an engineer opens the logs, the first rank to complain is often a bystander. PyTorch's documentation of the failure mode is blunt: the stack trace in the error message comes from the watchdog thread, not the thread that scheduled the collective, and the message is "largely unhelpful". Without recorded state, root-causing can take hours or longer, often necessitating job reruns.

This report targets GPU-cluster site reliability engineers, distributed-training platform engineers and incident commanders who must decide, under time pressure, whether a timeout is most consistent with **collective desynchronization**, an **application or rank stall**, a **GPU or host failure**, or a **fabric problem**. It then asks what evidence to capture before aborting, retrying or quarantining a node.

GPU Smith is an independent engineering advisor on enterprise GPU infrastructure and is not a provider of the tools discussed. Nothing here describes a client engagement or a benchmark run by the publisher; every operational fact is attributed to a vendor document, a project source tree or a preprint, and every threshold is left as policy for the reader.

Define the incident boundary first

Before any command is run, write down the following, because every later artifact is joined on these keys:

- **Job attempt and scheduler identifiers**, including the restart counter. `torchrun` exposes `TORCHELASTIC_RESTART_COUNT` as the number of worker group restarts so far.
- **Process group and rank map**: global rank, local rank, host and GPU for every rank, plus the process-group names that appear in Flight Recorder.
- **Hardware identity**: GPU UUID and PCI bus ID, queried with `nvidia-smi --query-gpu`. The UUID is a stable join key between kernel Xid lines (which print PCI addresses) and rank maps.
- **Pinned versions**: framework, NCCL, CUDA and driver. Defaults and dump behavior change between releases, and the RAS documentation itself warns that the exact format and scope of the information being made available varies from release to release.
- **First observed stall time in UTC**, from the earliest of application log, RAS monitor event, scheduler event or alert.

Anatomy of a Watchdog Timeout

What the watchdog actually does

PyTorch's ProcessGroupNCCL places CUDA events around each NCCL call and has a CPU-side watchdog poll whether the collective finished within a user-defined timeout (default is 10 minutes). The `init_process_group` reference documents the same default, 10 minutes for NCCL and 30 minutes for other backends (docs.pytorch.org). On expiry, the documentation says collectives "will be aborted asynchronously and the process will crash" (docs.pytorch.org).

`TORCH_NCCL_ASYNC_ERROR_HANDLING` has a documented default of 3 in PyTorch 2.14, which tears down the process without aborting the communicator (docs.pytorch.org). A monitoring thread aborts the process when the watchdog thread itself gets stuck. The 2.14 page states no numeric default for `TORCH_NCCL_HEARTBEAT_TIMEOUT_SEC`, so read the value in force from the pinned build.

Why the reporting rank is not the culprit

A collective times out in one of two ways according to the PyTorch blog: the kernel runs longer than the timeout or hangs, or "ranks desynchronize, causing the collective metadata or state to differ across ranks" (pytorch.org). Four root-cause families follow from that, and each has a different evidence signature.

- **CPU-side desynchronization.** The blog reports these as the dominant root cause inside Meta, at more than 60 percent across the four training frameworks it covers (pytorch.org). Examples include data loading, checkpointing or compilation stalls on one rank, and ranks running different collectives or orders.
- **GPU compute-kernel hang.** GPU execution is sequential within a stream, so any hang on the GPU while executing compute kernels will prevent the GPU from reaching the scheduled collectives (pytorch.org).
- **Misconfigured collective arguments.** PyTorch is not able to verify whether the passed-in arguments are valid, so dtype, shape or split mismatches surface as hangs.
- **Network or hardware faults.** The blog attributes between 20 and 30 percent of observed timeouts to these (pytorch.org), and notes that a faulty GPU will induce failures across multiple, unrelated jobs, which is why placement history matters.

Because most timeouts are desynchronization, "increasing the timeout value will not fix the issue" (pytorch.org). Raising the value before dumps exist delays the evidence and keeps healthy GPUs idle.

Version and feature matrix

Table 1 below consolidates the controls that matter for this runbook, with the version in which each appeared or the documentation page that defines it. Re-verify entries against the pinned builds: the PyTorch environment-variable reference is dated June 2025 on its own page (docs.pytorch.org), and the Flight Recorder tutorial lives under an "unstable" path.

Capability	Version or source	Control or command	Caveat to verify on the pinned build
NCCL RAS subsystem	Since NCCL 2.24 (docs.nvidia.com)	<code>ncclras</code> ; <code>NCCL_RAS_ENABLE</code> (default 1); <code>NCCL_RAS_ADDR</code> (default localhost:28028)	Output format varies between releases
RAS JSON output	NCCL 2.28.7 (docs.nvidia.com)	<code>-f json</code>	JSON is a raw dump; interpretation is left to the consumer
RAS monitor mode	NCCL 2.29 (docs.nvidia.com)	<code>ncclras -m</code>	Streams peer-change events until interrupted
RAS on-demand diagnostics	Documented in the 2.32.3 guide	<code>ncclras -D</code>	Asynchronous; may be delayed or incomplete for short-lived communicators
PyTorch watchdog timeout	<code>init_process_group</code> default 10 minutes for NCCL	<code>timeout=</code> argument; <code>torch.distributed.set_timeout</code>	Policy value; do not hard-code a universal number
Flight Recorder buffer	PyTorch 2.5 or later (docs.pytorch.org)	<code>TORCH_NCCL_TRACE_BUFFER_SIZE</code> greater than 0 enables collection (docs.pytorch.org); 0 disables	The tutorial recommends a size, the reference page states no default
Dump on timeout	PyTorch env reference	<code>TORCH_NCCL_DUMP_ON_TIMEOUT</code>	Requires a positive buffer size
Dump path	Tutorial: <code>TORCH_FR_DUMP_TEMP_FILE</code> (docs.pytorch.org); env reference: <code>TORCH_NCCL_DEBUG_INFO_TEMP_FILE</code>	File prefix, one file per rank	Variable names differ between pages; confirm which one the build reads
C++ stacks	Env reference default False (docs.pytorch.org)	<code>TORCH_NCCL_TRACE_CPP_STACK</code>	Symbolization mode is set by <code>TORCH_SYMBOLIZE_MODE</code>

The matrix shows two traps. RAS grew new capabilities across releases 2.24, 2.28.7, 2.29 and 2.31, so a runbook written against 2.24 will not mention JSON, monitoring or diagnostics. And the dump-path variable has two documented names, so a team that sets only one may write dumps to an unexpected place or nowhere; the rehearsal described below catches this.

Before Failure: Instrumentation That Survives the Incident

Flight Recorder configuration

The tutorial states that setting `N` to a positive number enables collection for the trace buffer, and that with dump-on-timeout enabled "there will be one file per rank output in the job's running directory" (docs.pytorch.org). The env reference adds that the dump variable must be set together with a positive trace-buffer size. Recorded entries carry collective type and state, input and output sizes and data types, and call stacks (pytorch.org). The states distinguish not scheduled, scheduled, started and completed (pytorch.org), and each collective carries a sequence number that is monotonically increasing within each process group the rank participates in. That sequence number is what makes cross-rank alignment possible.

Choices for a production profile, to be validated against the pinned build:

- **Buffer size.** Do not copy a universal number. Size the ring to span several training steps for the busiest process group, then confirm in a drill that entries around a deliberate failure survive.
- **Dump location.** Use a path that survives process teardown. The tutorial notes files are written to local disks or mounted shared NFS folders; node-local scratch avoids shared-filesystem stalls but requires a collection step.
- **C++ stacks and timing.** Both are optional. C++ stack collection is off by default, and timing records start events for every collective (docs.pytorch.org). Measure overhead on a representative job first.
- **Retention and access.** Dumps contain call stacks and tensor shapes. Store them in an access-controlled location with a stated retention period, and scrub tokens, model identifiers and tenant names before sharing a bundle.

Meta's published experience is that the cost is acceptable, reporting "a near 100% Flight Recorder full dump rate" (pytorch.org); the page gives no numeric overhead figure, so none is claimed here. A dedicated monitor thread polls a store and triggers the dump on every rank.

NCCL RAS and logging configuration

RAS is on by default and runs one thread per process (docs.nvidia.com); its traffic is plain TCP on the bootstrap interface. NVIDIA describes it as lightweight, one that should not interfere with the main NCCL job. RAS listens on localhost, port 28028 by default (docs.nvidia.com); when several jobs share a node, each needs a different address value; and binding to an external interface makes it reachable from other nodes but that has security implications. Since NCCL 2.31 the same socket also accepts CONTROL commands that adjust NCCL behavior at run time (docs.nvidia.com), and the page does not discuss authentication, so exposure should be a deliberate, reviewed decision.

For logs, NVIDIA documents a per-host, per-process recipe, `NCCL_DEBUG=INFO NCCL_DEBUG_SUBSYS=ALL NCCL_DEBUG_FILE=/tmp/ncc1_%h_%p.log` (docs.nvidia.com). Timestamps are not universal: by default, only WARN messages include timestamps, so a UTC timeline needs the timestamp options set or file times used as bounds. Full ALL logging is a reproduction tool, not a standing production setting.

Rehearse the dump

Run a drill job that deliberately skips a collective on one rank, and verify that every rank writes a file, that the analyzer reads all of them, and that the collection script finds them.

At the Stall: Live Evidence Capture

When a job is stuck but has not yet timed out, work in this order: read-only queries, state snapshots, and only then anything that changes the job (abort, restart, CONTROL commands, driver resets).

Query RAS

Query RAS from any node where the job is running; the client binary ships with NCCL (developer.nvidia.com). The `-v` flag gives a more verbose output in case of problems, and responses for large or unhealthy jobs may take several seconds to generate. The interpretation rules from the RAS guide are:

- **Mismatch warning.** It signals "an inconsistency in the information provided by different communicator ranks" (docs.nvidia.com). A collective-count mismatch is evidence of lagging ranks, but it can be transient. If the output does not change across subsequent RAS queries it may indicate a stuck communicator (docs.nvidia.com).
- **Missing or incomplete data.** The tool reports lines such as Missing communicator data from 1 job process. After the first slow query, RAS reconfigures its network to route around the unresponsive process, so later queries are faster (docs.nvidia.com).
- **Dead process.** An unreachable process is declared dead permanently after 60 seconds (docs.nvidia.com). In JSON output, an "unresponsive" flag means RAS was unable to reach the rank.

Repeat the query at intervals and save each response with its UTC capture time. A single snapshot cannot distinguish a slow rank from a stopped one; change between snapshots can.

On-demand diagnostics and their limit

RAS diagnostics can be triggered with the client's `-D` option. They compare configuration across ranks and include a kernel-log scan, which first tries `/dev/kmsg` and falls back to `dmesg` for Xid lines. Initialization-triggered diagnostics are off by default (docs.nvidia.com). The limitation must be stated in the incident record: RAS diagnostics do not exercise NCCL data paths and do not provide a comprehensive cluster-health assessment (docs.nvidia.com). A clean report therefore rules out configuration drift and logged Xids at that moment; it does not rule out congestion, a degraded link or a switch fault.

Snapshot the host

The remaining live evidence is host-side. Capture it before any restart, and record the UTC time of each capture.

- **Python stacks per rank.** The PyTorch blog points to `py-spy` for CPU main-thread stacks. The `py-spy` documentation says that for a hung program "you just need a single call stack to figure out where your python program is hung" (raw.githubusercontent.com). Attaching needs `ptrace` permission, which the README describes lifting via "the `ptrace_scope` sysctl variable" (raw.githubusercontent.com); a security team should approve that in advance. Where attach is not allowed, Python's `faulthandler` can pre-register a signal-triggered dump (docs.python.org) or be enabled at startup with `-X faulthandler` (docs.python.org).
- **GPU state.** Record UUID, PCI bus ID, utilization and NVLink status through `nvidia-smi`. Xid events live in the kernel log: "Xid error messages are logged in the kernel log buffer" (docs.nvidia.com).
- **RDMA and Ethernet counters.** The Linux counter files sit under `/sys/class/infiniband/` (enterprise-support.nvidia.com). Take two reads a known interval apart, because hardware counters have an aging lifespan (enterprise-support.nvidia.com). For RoCE, `ethtool -S` queries "NIC- and driver-specific

statistics" (man7.org); for InfiniBand, `perfqquery` obtains port counters (man7.org).

- **Scheduler and placement.** Record job ID, node list, restart count and recent node state changes.

The rank evidence matrix

Table 2 below defines the minimum record per rank. It is deliberately a record of what is known and what is not: a blank cell is "unknown", never "healthy".

Field	Where it comes from	Why it matters	If missing
Global rank, local rank, host	Launcher environment, scheduler node list	Maps dump files and log lines to hardware	Mark the rank unmapped; do not infer from file order
GPU UUID and PCI bus ID	<code>nvidia-smi --query-gpu</code>	Joins rank to Xid lines and fleet history	Unknown identity; block quarantine decisions
Process group name and size	Flight Recorder entries	Identifies which communicator stalled	Record as unknown; treat alignment as incomplete
Last completed and last enqueued collective, sequence number	Flight Recorder dump	Locates the first divergence between ranks	Rank counts as missing evidence
Heartbeat or RAS state	<code>ncclras</code> snapshots with UTC time	Shows responsive, unresponsive or dead peers	No RAS answer is not proof of death
First error time	Application log, NCCL warning, scheduler event	Orders events on the UTC timeline	Use bounds from file times
Artifact paths and hashes	Bundle manifest	Makes the analysis reproducible	Treat the claim as unverified

The matrix works because it forces every later claim to name its source. If three of sixty-four ranks have no dump and no RAS answer, the honest status for those ranks is unknown, and any conclusion drawn from the remaining sixty-one carries that caveat in the incident record.

After the Timeout: Flight Recorder Analysis

Collect one file per rank, then verify completeness

The analyzer expects a directory with one trace file per rank named with a prefix and the rank number, and a prefix option exists to strip a common filename prefix "such that rank can be extracted"

(raw.githubusercontent.com). Its default behavior is strict: "FR trace require all ranks to have dumps for analysis" (raw.githubusercontent.com), with a flag to permit incomplete rank sets. The strictness is useful: a missing file is itself a finding (the process died, hung before the dump, or its node was not collected), and partial files must not yield a confident conclusion.

The analyzer is invoked on the dump directory of pickle files. The analyzer prints mismatches starting from the earliest, capped by a configurable count (raw.githubusercontent.com), which matches the incident question: where did the ranks first stop agreeing?

Reporter versus suspected origin

Distinguish three roles in the write-up.

- **Reporter.** The rank whose watchdog fired first. Its identity is a log artifact only (pytorch.org).
- **Lagging or missing rank.** A rank that never scheduled the collective others scheduled (state "not scheduled", a.k.a. missing), or whose last completed sequence number is lower than its peers.
- **Suspected origin.** The earliest divergence plus corroboration from a second source, such as a Python stack from the lagging rank, an Xid on its GPU, or a counter anomaly on its NIC.

CPU stuckness or a GPU hang shows as a missing rank or a state mismatch; CPU divergence shows as a missing rank or a mismatch in collective type, call stack or occasionally data type; misconfigured arguments show as a dtype or size mismatch (pytorch.org). A network fault may show no mismatch at all: every rank has the same collectives in flight and none completes.

TORCH_NCCL_DESYNC_DEBUG is described as helping to find "the culprit rank of collective desync" (docs.pytorch.org), and TORCH_DISTRIBUTED_DEBUG=DETAIL adds a wrapper that "performs consistency checks before dispatching the collective to an underlying process group" (docs.pytorch.org). Both are reproduction aids, not production defaults.

Triage Branches

The branches below use evidence from the two previous sections. They are ordered by how cheaply they can be confirmed or excluded.

Collective order, shape or argument mismatch

Signature: Flight Recorder shows different collective types, sizes or data types at the same sequence number, or a rank with extra or missing collectives. PyTorch cannot verify argument validity, so these are application defects, often triggered by data imbalance or divergent control flow on one rank. `monitored_barrier` can report which ranks missed a barrier, but it is "only supported with the GLOO backend" (docs.pytorch.org), so it needs a separate group.

CPU-side or dataloader stall

Signature: one rank is missing a collective that its peers scheduled, the GPU on that rank is idle, and a Python stack points at data loading, checkpoint I/O, compilation or a lock. This is the largest family in the blog's experience. Storage and scheduler dependencies belong here: a slow shared filesystem looks, from NCCL's side, like a rank that never arrived.

CUDA or GPU hang

Signature: the rank scheduled the collective but compute kernels ahead of it never finish, often with a corroborating Xid. NVIDIA's [Xid catalog](#) describes graphics-engine exceptions (Xid 13) as general user application faults, memory-management-unit faults (Xid 31) as MMU-reported, and uncontained memory errors (Xid 95) as requiring a GPU reset before applications can restart. Several codes list both application and hardware causes, so one Xid is a lead, not a verdict.

Process exit

Signature: a rank vanished. NCCL's API distinguishes a remote error, possibly due to a network error or a remote process exiting prematurely. After an asynchronous error the guide says all other ranks must call `ncclCommAbort` on their own communicators, and a blocking communicator can possibly get stuck inside NCCL calls due to network errors. Process-exit evidence includes the launcher's error file, which `torchrun's @record` writes to the path in `TORCHELASTIC_ERROR_FILE`.

Interface or bootstrap problem

Signature: failure at or soon after initialization, a wrong interface chosen, or ranks unable to reach each other. NVIDIA recommends INFO logging with the INIT, BOOTSTRAP and NET subsystems "if your application hangs during NCCL initialization" (docs.nvidia.com), and notes that in containers NCCL needs `/sys` to discover PCI topology.

RDMA or fabric degradation

Signature: no mismatch in Flight Recorder (all ranks have the same collective in flight), RAS shows all peers responsive, and fabric counters move. Useful counters include `symbol_error` for "minor link errors detected on one or more physical lanes" (enterprise-support.nvidia.com), `link_downed` for links downed after failed recovery (enterprise-support.nvidia.com), `port_xmit_wait` for ticks with data to send but nothing sent (enterprise-support.nvidia.com), and `out_of_sequence` packets (enterprise-support.nvidia.com). RoCE congestion indicators include CNP counters (enterprise-support.nvidia.com) and a receive-buffer-over-85-percent event counter in the kernel's `mlx5` documentation (docs.kernel.org). Switch-side bit-error-rate checks are available in NVIDIA UFM fabric health reports (networking-docs.nvidia.com).

Table 3 below gathers the branches into a symptom-to-evidence decision table. It separates what each branch would predict from what would contradict it.

Branch	Flight Recorder signature	RAS or live signature	Host, GPU or fabric corroboration	Candidate disposition
Collective desync (order, shape, dtype)	Mismatch in type, size or dtype at one sequence number	Op-count mismatch that persists	None required; stacks show divergent code paths	Application escalation
CPU or dataloader stall	Rank missing collectives that peers scheduled	Lagging rank, still responsive	Idle GPU, Python stack in I/O or compile	Application or storage escalation; retry after fix
GPU hang	Rank scheduled but compute never finished	Rank may remain responsive	Xid lines, repeat history on same UUID	Drain and quarantine on hardware evidence
Process death	Dump missing or truncated	Peer marked dead after 60 seconds	Exit code, OOM or kernel log	Retry if cause is transient; otherwise application escalation
Fabric degradation	No mismatch; all collectives started, none completed	Peers responsive; diagnostics clean but data path untested	Counter deltas, switch logs, NCCL INFO warnings	Network escalation; drain suspect links or nodes

Branch	Flight Recorder signature	RAS or live signature	Host, GPU or fabric corroboration	Candidate disposition
Unknown	Incomplete files or ambiguous	RAS unavailable	Evidence gaps	Retry once with enhanced capture; do not quarantine on this alone

Rows are hypotheses, not verdicts: two rows can be true at once (a degraded link can coexist with an application bug), and a case that matches no row cleanly belongs in the last one, with the missing evidence listed.

Reproduction Ladder and Disposition

Controlled reproduction ladder

Change one variable per step, on the same placement as the failure, and only after the evidence bundle is sealed.

1. **Smallest communicator and message.** Run NVIDIA's `nccl-tests`, which "check both the performance and the correctness of NCCL operations" ([raw.githubusercontent.com](https://raw.githubusercontent.com/NVIDIA/nccl-tests/master/README.md)), for example `all_reduce_perf -b 8 -e 128M -f 2 -g 8` from the README ([raw.githubusercontent.com](https://raw.githubusercontent.com/NVIDIA/nccl-tests/master/README.md)), on the failing hosts only.
2. **Same placement, scale up.** Move to the full node set of the failed attempt. Compare bus bandwidth, which "reflects how optimally the hardware is used" ([raw.githubusercontent.com](https://raw.githubusercontent.com/NVIDIA/nccl-tests/master/README.md)).
3. **Isolate transports.** NCCL documents switches for disabling peer-to-peer (`NCCL_P2P_DISABLE`) and InfiniBand Verbs. The GPU guide recommends comparing runs with peer-to-peer disabled or restricted. For GPUDirect RDMA, compare host-memory and GPU-memory results. These switches are diagnostic only.
4. **Hardware diagnostics.** DCGM's `dcgmi diag` has levels from 1 to 4, with level 3 labelled Long (System HW Diagnostics) and level 4 Extended (Longer-running System HW Diagnostics). The DCGM health checks are separate and based on passive, background monitoring.
5. **Application scale.** Re-run the real job only after lower rungs pass or have identified the faulty component.

Disposition

- **Retry.** Appropriate when all ranks produced dumps, the signature is a documented transient (for example a single process death with a clear external cause), and no hardware signal is present. `torchrun` restarts all workers when any worker fails "up to" `max_restarts` (docs.pytorch.org), so a retry budget should be set by policy rather than inherited from a default.
- **Drain or quarantine.** Requires hardware evidence tied to a UUID or host: an NVLink Xid, where the catalog notes that another connected GPU "may report an Xid 74" (docs.nvidia.com), so the reporting side may not be the failed side; a PCIe fall-off-the-bus Xid; or repeated failures of unrelated jobs on the same GPU. In Slurm, a drained node accepts no new jobs (slurm.schedmd.com), whereas marking a node down stops all running jobs (slurm.schedmd.com); `sacct` reports `NODE_FAIL` when allocated

nodes fail (slurm.schedmd.com). In Kubernetes, cordon marks a node unschedulable (kubernetes.io), and drain is the documented way to "safely evict all of your pods from a node before you perform maintenance" (kubernetes.io); drain honors budgets, "the PodDisruptionBudgets you have specified" (kubernetes.io), and the documentation advises issuing drain to a single node at a time (kubernetes.io). Record the evidence bundle ID in the drain reason.

- **Application escalation.** Flight Recorder mismatch or missing-rank evidence with Python stacks pointing into user code.
- **Network escalation.** No mismatch, responsive RAS, and counter or switch evidence. Remember that RAS diagnostics do not test the data path.
- **Unknown.** Incomplete evidence. Say so, preserve what exists, and improve capture before the next attempt.

Slurm's HealthCheckProgram can "verify the node is fully operational and DRAIN the node or send email" (slurm.schedmd.com); such automation should read, not overwrite, the incident bundle.

Incident Bundle and UTC Timeline

Seal one bundle per job attempt before any restart or drain. It should hold a manifest (job attempt, versions, rank map, GPU UUIDs, file hashes); the Flight Recorder dumps and analyzer output; timestamped RAS snapshots; NCCL and application logs per host; kernel-log extracts, `nvidia-smi` and DCGM output; before and after counter reads; and scheduler records of placement, state changes and exit codes. Scrub tokens, model data and tenant identifiers before the bundle leaves the owning team. Kernel-log reads and process attach typically need elevated privileges, so agree access with platform and security owners in advance rather than during the incident.

The timeline joins every source on one UTC axis. A template:

```
time_utc | source | host/rank | event | artifact | clock_offset_known
source in: application, flight_recorder, nccl, ras, scheduler, kernel, gpu, switch
```

Record clock offsets between hosts, mark gaps as unknown, and keep the reporter, lagging rank and suspected origin as separate rows so the postmortem can show how the first alert differed from the cause.

Data Analysis and Evidence

Timing arithmetic that frames the first minutes

The quantities that order an incident are small enough to compare directly. These are documented defaults; the sums are analyst arithmetic, not vendor claims.

- **Watchdog timeout.** 10 minutes for NCCL in PyTorch 2.14.
- **RAS dead-peer declaration.** 60 seconds of failed contact before a process is declared dead, with a 5-second default timeout for the RAS client's own queries, before any `NCCL_RAS_TIMEOUT_FACTOR` scaling (docs.nvidia.com).

- **InfiniBand transport timeout.** NCCL_IB_TIMEOUT defaults to 20 since NCCL 2.23 (docs.nvidia.com). The documented formula is 4.096 microseconds times 2 to the power of the setting, which gives about 4.3 seconds per attempt. With the default retry count of 7 and the rule that total retry time is the product of the retry count and the timeout, the total is on the order of 30 seconds.

Transport failures can therefore surface well before the 10-minute watchdog, while a desync raises no transport error and only the watchdog or an external check notices it. Raising the watchdog value is not neutral: a 10-minute value leaves 600 seconds for capturing live RAS and stack evidence, whereas a 60-minute value leaves 3,600 seconds of idle GPUs with no extra diagnostic value if the cause is a missing rank.

Published figures on cause mix and detection

The published figures are summarized here in prose. In the PyTorch blog, CPU-side issues are described as the dominant root cause (more than 60 percent across the training frameworks the blog covers) and network or hardware issues account for 20 to 30 percent of observed timeouts (pytorch.org). These percentages come from one operator's fleet and are priors, not expected rates elsewhere.

Research on automated localisation reports the following, all from arXiv preprints and therefore not independently confirmed here:

- **CCL-D** reports that it diagnoses "slow/hang anomalies with high precision within 6 minutes" (arxiv.org), deployed on a 4,000-GPU cluster over one year (arxiv.org), with overhead under 0.45 percent (arxiv.org).
- **C4/C4D** reports a response time of mere tens of seconds for error detection and node identification (arxiv.org), detecting slowness by "comparing completion time of messages in different workers" (arxiv.org), and a system-efficiency improvement from 30 to 45 percent (arxiv.org).
- **ByteRobust** flags stalls on "zero RDMA traffic within 10 minutes or low TensorCore utilization" (arxiv.org) and isolates machines that share a parallel group with the outliers (arxiv.org); it reports 97 percent effective training time ratio for a three-month, 9,600-GPU job (arxiv.org).
- **SCOUT** uses an out-of-band CPU observer that "remains responsive when training hangs" (arxiv.org), and **ARGUS** targets always-on collection under a combined overhead of less than 2 percent (arxiv.org).

The common pattern, an out-of-band observer plus cross-rank comparison, is the shape of this runbook, with RAS and Flight Recorder as the in-tree equivalents. The figures are not comparable across papers because workloads, scales and fault models differ.

Evidence completeness scorecard

Score each incident before assigning a cause. The score reports what is known, and absent evidence counts as unknown.

- **Rank coverage.** Dumps present divided by ranks expected. Anything below 100 percent caps confidence, because the analyzer by default requires every rank.
- **RAS coverage.** Number of timestamped RAS snapshots, and whether any peers were unresponsive or dead.
- **Host coverage.** Kernel log, GPU state and counter deltas collected per suspect host.

- **Timeline and version coverage.** Whether application, Flight Recorder, NCCL, scheduler, kernel, GPU and switch events share one UTC axis, and whether framework, NCCL, CUDA and driver versions are recorded.

Case Studies and Real-World Examples

Rehearsal of a mismatch drill (Hypothetical Example)

A platform team runs an eight-rank rehearsal in which rank 3 skips one all-reduce. Log order suggests rank 0, since it reported first. The dumps show every rank at the same sequence number except rank 3, which is one collective behind, and a saved Python stack from rank 3 points into a data-loading branch. The record reads: reporter rank 0, lagging rank 3, origin an application code path; disposition application escalation; no quarantine, since no Xid or counter evidence exists. The drill is hypothetical and not a measured result.

A fabric-suspect contrast (Hypothetical Example)

In a second rehearsal, all 64 dumps exist and show identical sequence numbers with the same collective started on every rank. RAS reports all peers responsive and its diagnostics are clean, but two NICs on one host show rising `port_xmit_wait` and `link_downed` deltas between snapshots. The record reads: no desync, data path untested by RAS, counter evidence on one host; disposition network escalation pending link tests. The example is illustrative only.

Implications and Future Directions

The evidence model above has three consequences for platform teams. First, **observability must be on before it is needed**: Flight Recorder and RAS are cheap relative to a lost attempt, but only if they are enabled, tested and retained. Second, **evidence must be joined across layers**. The NCCL RAS guide itself notes that the exact output format varies from release to release, which argues for storing the raw JSON where available and building the timeline from timestamps rather than screen output. Third, **disposition decisions should be policy-driven**. Restart budgets, suspect-node wait times and counter-delta thresholds belong to the operator's reliability policy; this report supplies the evidence to feed them, not the numbers.

Tooling is moving in the direction of live, streaming state. RAS gained a monitoring mode for real-time updates in NCCL 2.29 and a `CONTROL` namespace in 2.31, and NCCL's fault-tolerance API now includes `communicator shrink`, which creates a new communicator by removing failed ranks from the existing communicator, together with `revoke`, after which a communicator is quiesced and safe for destroy, split, and shrink. Abort remains a collective obligation: all active ranks are required to call this function. Where frameworks adopt these primitives, a failed rank may become a recoverable event rather than a job restart, and evidence workflows will need to cover shrink events.

For buyers and operators evaluating infrastructure, the same logic applies to acceptance: ask which evidence sources exist on the delivered stack, who owns the fabric counters, and whether node-level capture is possible within the site's security policy. Open questions remain: public data on dump success rates outside the originating operator is sparse, and the overhead of optional capture features has not been published as a general benchmark.

Frequently Asked Questions (FAQs)

What does "NCCL watchdog caught collective operation timeout" mean? It means the PyTorch watchdog thread observed a collective that had not completed within the configured timeout. The error is raised by the PyTorch watchdog, not the NCCL library (pytorch.org), and the rank that raises it first is rarely the culprit.

Should the timeout be increased? Not before evidence is preserved. If a legitimately slow phase such as a long checkpoint is the cause, the evidence will show all ranks at the same step, and the timeout can then be changed as a documented policy decision.

How can an NCCL hang be debugged? Query RAS for op-count mismatches and unresponsive peers, capture Python stacks and kernel logs, then analyse Flight Recorder dumps from every rank. For initialization hangs, use INFO logging with INIT, BOOTSTRAP and NET.

What are the relevant NCCL debug environment variables? NCCL_DEBUG for level, NCCL_DEBUG_SUBSYS (default INIT, BOOTSTRAP and ENV (docs.nvidia.com)) for subsystems, and NCCL_DEBUG_FILE with %h and %p patterns. PyTorch's own guidance for hangs in collective calls is to use the COLL subsystem.

What is NCCL RAS and what can it not do? RAS is a built-in health query subsystem available since NCCL 2.24. It does not test the data path.

How does an all_reduce hang differ from a slow all_reduce? A hang shows the same sequence number stuck across repeated snapshots; a slow collective completes with high duration, which requires timing to be enabled.

Conclusion

A watchdog timeout is where triage begins. PyTorch's authors say the first reporting rank is rarely the culprit, NCCL's argument checking is absent (pytorch.org), and the documented causes range from application desynchronization to GPU and network faults. A defensible runbook therefore captures rank-local Flight Recorder dumps, timestamped RAS snapshots, host and GPU logs, fabric counters and scheduler placement on one UTC timeline, scores completeness honestly, and only then chooses among retry, quarantine, escalation or unknown. Raising the timeout is a policy change to be made after the evidence is sealed, not a first response. Tooling is improving quickly: RAS now offers JSON output, a monitoring mode and on-demand diagnostics, and Flight Recorder removes much of the guesswork about which rank diverged. The remaining discipline is organizational: enable the instruments before the incident, keep the dumps, join the evidence on one clock, and report unknowns as unknowns. Teams that do so can answer the incident commander's question, retry, quarantine, escalate or unknown, with a stated basis rather than a guess. The versions, variable names and defaults cited here are as documented in October 2026 for NCCL 2.32.3 (docs.nvidia.com) and PyTorch 2.14.0 (docs.pytorch.org), and each should be confirmed against the pinned build before it enters a production runbook.

Source links

Links cited in this article, in order of first appearance. Full addresses are provided for readers using extracted text.

1. <https://pytorch.org/blog/flight-recorder-a-new-lens-for-understanding-nccl-watchdog-timeouts>
2. <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html>
3. https://docs.pytorch.org/docs/2.14/torch_nccl_environment_variables.html
4. https://docs.pytorch.org/tutorials/unstable/flight_recorder_tutorial.html
5. <https://docs.pytorch.org/docs/2.14/distributed.html>
6. <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/troubleshooting/ras.html>
7. <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/troubleshooting/logging.html>
8. <https://developer.nvidia.com/blog/networking-reliability-and-observability-at-scale-with-nccl-2-24>
9. <https://raw.githubusercontent.com/benfred/py-spy/master/README.md>
10. <https://docs.python.org/3/library/faulthandler.html>
11. <https://docs.nvidia.com/deploy/xid-errors/working-with-xid-errors.html>
12. <https://enterprise-support.nvidia.com/s/article/understanding-mlx5-linux-counters-and-status-parameters>
13. <https://man7.org/linux/man-pages/man8/ethtool.8.html>
14. <https://man7.org/linux/man-pages/man8/perfquery.8.html>
15. https://raw.githubusercontent.com/pytorch/pytorch/main/torch/distributed/flight_recorder/components/config_manager.py
16. <https://docs.nvidia.com/deploy/xid-errors/analyzing-xid-catalog.html>
17. https://docs.kernel.org/networking/device_drivers/ethernet/mellanox/mlx5/counters.html
18. <https://networking-docs.nvidia.com/ufmsdnappswum/4183/fabric-health-tab>
19. <https://raw.githubusercontent.com/NVIDIA/nccl-tests/master/README.md>
20. <https://raw.githubusercontent.com/NVIDIA/nccl-tests/master/doc/PERFORMANCE.md>
21. <https://docs.pytorch.org/docs/2.14/elastic/run.html>
22. <https://slurm.schedmd.com/scontrol.html>
23. <https://slurm.schedmd.com/sacct.html>
24. https://kubernetes.io/docs/reference/kubectl/generated/kubectl_cordon
25. <https://kubernetes.io/docs/tasks/administer-cluster/safely-drain-node>
26. <https://slurm.schedmd.com/slurm.conf.html>
27. <https://arxiv.org/html/2605.04478v1>
28. <https://arxiv.org/abs/2605.04478>
29. <https://arxiv.org/html/2406.04594v2>
30. <https://arxiv.org/abs/2406.04594>
31. <https://arxiv.org/html/2509.16293>
32. <https://arxiv.org/abs/2509.16293>
33. <https://arxiv.org/abs/2608.11034>
34. <https://arxiv.org/abs/2606.20374>

THE PUBLISHER

About GPU Smith

GPU Smith provides independent engineering and research for private AI infrastructure. We help technical buyers and operators reason about GPU workload sizing, hardware selection, procurement, deployment and inference operations, from the compute system to the networking, power and cooling requirements around it.

Start with the workload

Useful infrastructure decisions begin with the models, data, throughput, latency and operating constraints a team actually has. GPU Smith helps connect those requirements with system design and deployment choices. The goal is a privately controlled AI environment whose capacity and operational demands are understood before a purchasing decision.

Hardware and supplier research

Our public reference library covers GPUs, complete systems and networking components. The server-vendor directory and manufacturing research help teams investigate suppliers, compare options and follow sources behind technical claims. We distinguish manufacturer specifications from measured benchmarks and advertised capabilities from tested configurations. Reference pages are research resources, not a live inventory listing or a binding equipment quote.

Deployment and operations

GPU Smith's engineering scope includes infrastructure integration, networking, power, cooling and the practical operation of inference workloads. Our published articles explain the tradeoffs behind deployment, procurement and ongoing operations so teams can ask better questions and document decisions.

Work with GPU Smith

Explore the [hardware library](#), [GPU references](#), [networking references](#) and [vendor research](#). [Contact GPU Smith](#) with your workloads and deployment constraints to discuss a project and confirm current scope, pricing and availability.

Inclusion of a manufacturer or product in our research does not imply a partnership, certification or endorsement.

Website: <https://gpusmith.com>

This article is educational. Verify consequential medical, legal, financial, regulatory and technical decisions with appropriate professionals and the cited primary sources. Company services are described separately from the article's research findings.