# NCCL Profiler Plugin API – A Feasibility Study

## Contents

# 1 TODO / Structure (from Markdown)

## 1.1 Table of Contents

- Abstract – motivate GPU communication profiling/tracing
- Introduction – assumed background as needed (e.g. MPI, NCCL, SLURM, Score-P)
- The Profiler API
  - How NCCL detects the profiler plugin (first draft; TODO final draft)
  - The Profiler API definition (with codeflow/swimlane diagrams showing where NCCL calls the profiler api)
  - Code examples illustrating the codeflow/behaviour (simple example, multi-node, ncclGroup)
  - What is possible with the Profiler Plugin API? Considerations and Pitfalls for (logging, running metrics, CUPTI, ...) - section title WIP
  - Performance and scalability of the Profiler Plugin API (experiments, accuracy)
- Potential integration with Score-P
- Conclusion

## 1.2 Main content chunks / concepts

- Simple code example walkthroughs
- Swim-lane diagrams (User API → init/finalize; start/stop/recordEventState)
- Benchmarking, measurements
- Conclusion

## 2 Abstract


- AI – major use case for HPC
- Expensive workloads; desire to understand and optimize application performance
- A large part of AI workloads is GPU communication (often spanning many GPUs)
- NCCL – the library that implements communication routines for NVIDIA GPUs
- Provides an interface to plug a custom profiler into NCCL and extract performance data


## 3 Introduction

TODO: mention as needed – MPI, NCCL, SLURM, Score-P.

### 3.1 NCCL vs MPI - Comparison (TODO)

cant think of any weird asymmetries, besides the involvement of gpu devices and threads vs process. maybe skip this section.

MPI

- Centered around CPU processes communicating
- Rank = CPU process
- within a communicator, a CPU process can be assigned exactly 1 unique Rank

NCCL

- Centered around GPUs communicating; CPU threads help orchestrate communication
- Rank = GPU device
- within a communicator, a CPU thread possibly handles multiple ranks (multiple devices) (TODO true if this works: check slurm 2694771 multi gpu per task ncclGroup)

### 3.2 Relevant NCCL internals

Before taking a closer look at the Profiler Plugin API, it is helpful to understand what NCCL does internally when an application calls the NCCL User API.

A typical NCCL application follows this basic structure:

```
// create nccl communicators
createNcclComm();

// allocate memory for computation and communication
prepareDeviceForWork();

// do computation and communication
callNcclCollectives();
// ...

// finalize and clean up nccl communicators
cleanupNccl();
```

During NCCL communicator creation, NCCL internally spawns a thread called `ProxyService`. This thread lazily starts another thread called `ProxyProgress`, which handles network requests for GPU communication during collective and P2P operations. See Figure 1

The guards `if (proxyState->refCount == 1)` and `if (!state->thread)` ensure that these threads are created once per shared resource (struct `ncclSharedResources`). Snippets of the relevant structs:

**/src/include/comm.h**

```
struct ncclSharedResources {
  struct ncclComm* owner; /* communicator which creates this shared res. */
  struct ncclProxyState* proxyState;
  // other fields
}
```

**/src/include/proxy.h**

```
struct ncclProxyState {
  int refCount;
  pthread_t thread;
  // other fields
}
```

By default every NCCL communicator has its own shared resource. When the application calls `ncclCommSplit()` or `ncclCommShrink()` where the original communicator was initialized with an `ncclConfig_t` with fields `splitShare` or `shrinkShare` set to `1`, the newly created communicator shares the shared resource (and the proxy threads) with the parent communicator.

```
/* proxyState is shared among parent comm and split comms.
comm->proxyState->thread is pthread_join()'d by commFree() in init.cc when
the refCount reduces down to 0.  */
```

(Quote from **/src/proxy.cc**)

Later, whenever the application calls the NCCL User API, NCCL internally decides what network operations to perform and posts them to a pool. The ProxyProgress thread reads these operations

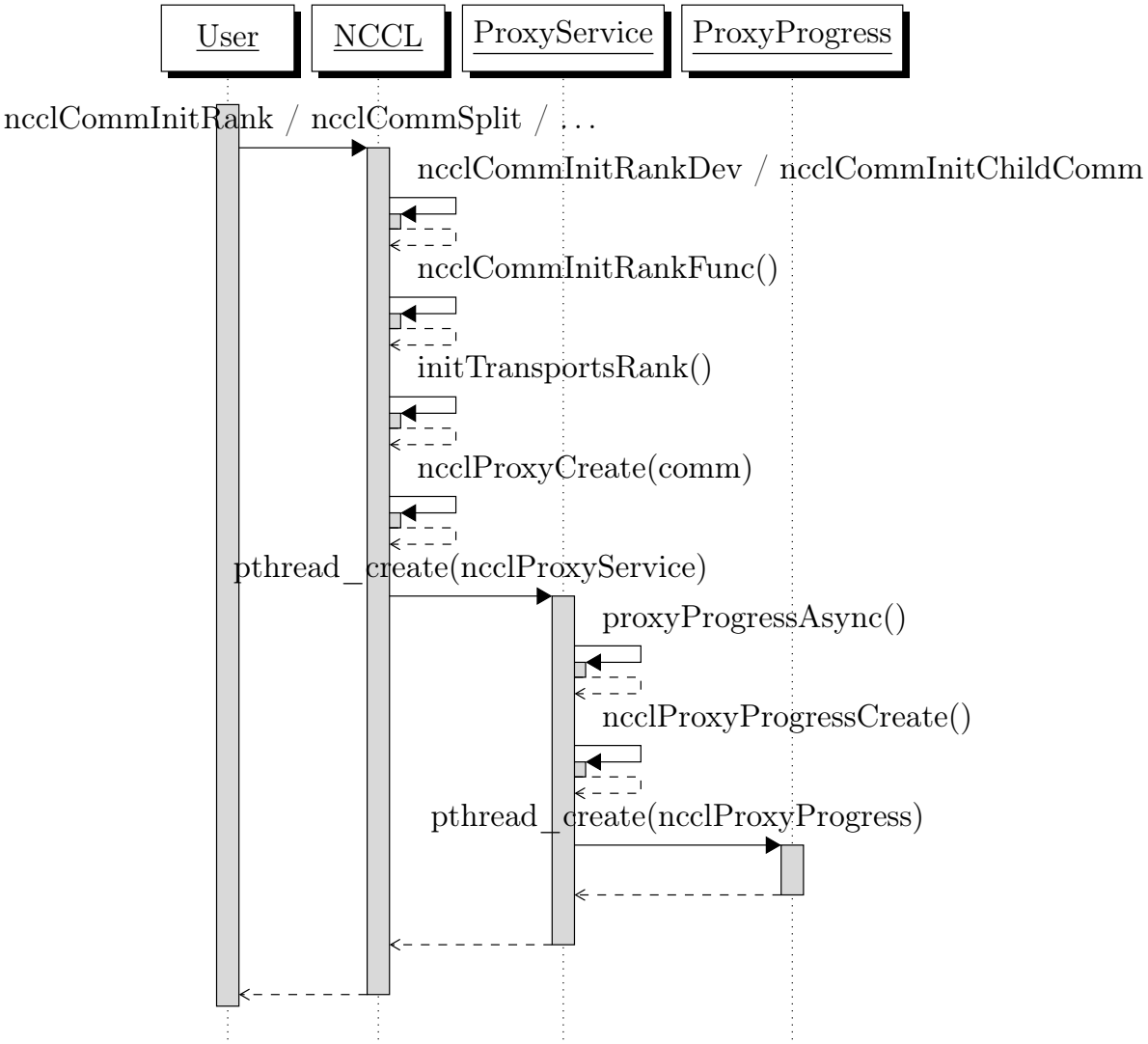



Figure 1: Thread creation flow: User API → NCCL internal init → spawn ProxyService → spawn ProxyProgress.

from the pool and progresses them. The following paths reach `ncclProxyPost()`, where ops are posted to the pool and the proxy progress thread is signalled. See Figure 2

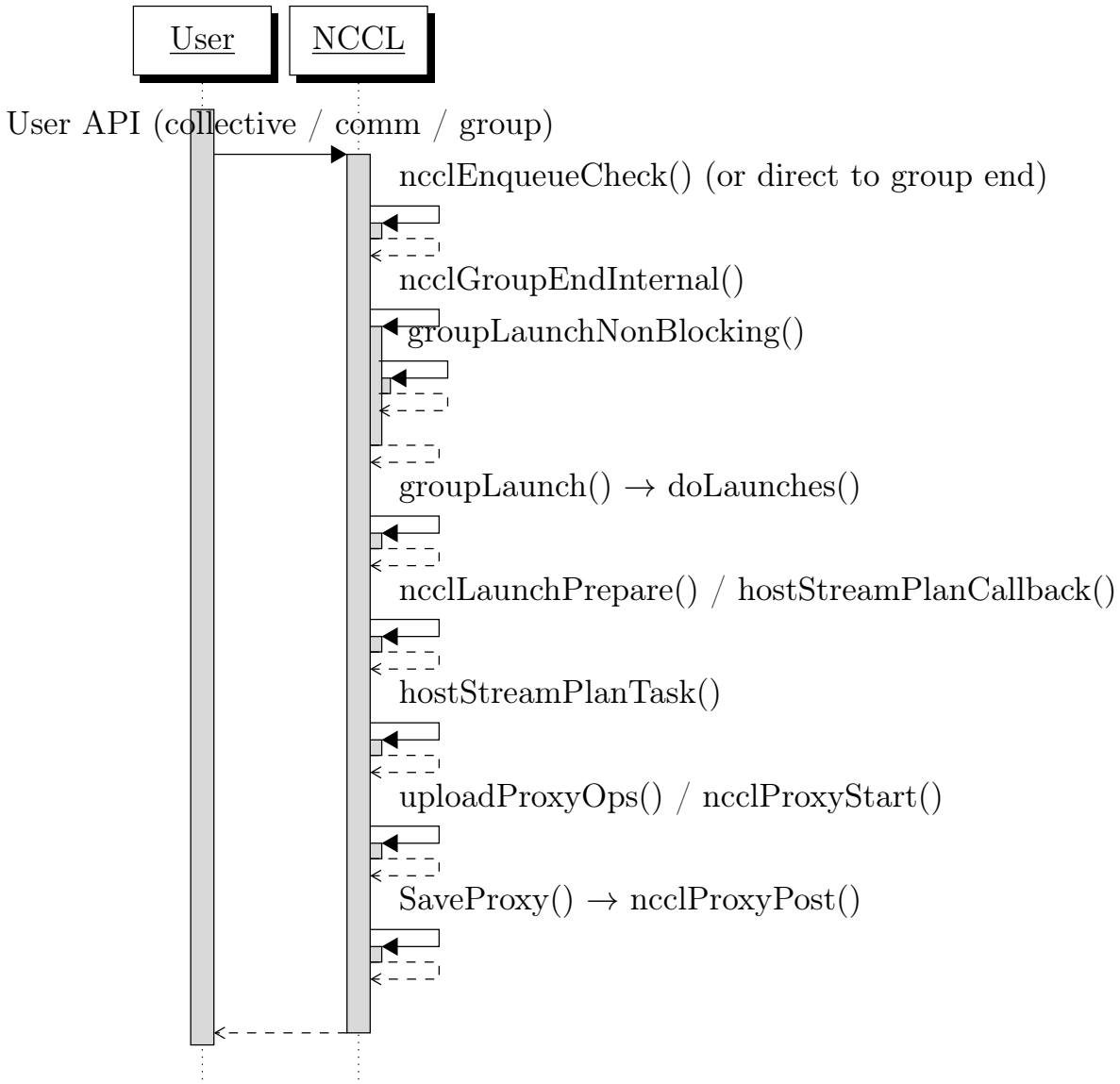



Figure 2: Flow from User API to `ncclProxyPost()`: two entry families merge at `ncclGroupEndInternal()`, then launch and proxy path to `ncclProxyPost()`.

The proxy progress thread reads from this pool when calling `ncclProxyGetPostedOps()` and progresses the ops. See Figure 3

Understanding this behaviour is useful for the Profiler Plugin API and network-related activity in the next section.

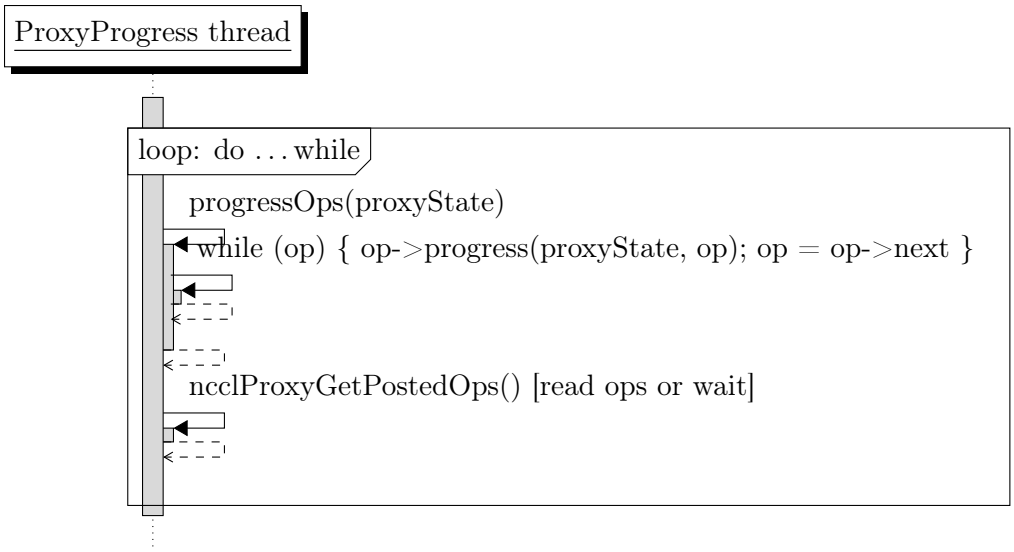



Figure 3: /**src**/**proxy.cc** `ncclProxyProgress()` progressing loop: progress ops, then get posted ops (or wait).

# 4 The Profiler API

## 4.1 How NCCL detects the profiler plugin

When an NCCL communicator is created, NCCL looks for a shared library that represents the profiler plugin, checking the environment variable `NCCL_PROFILER_PLUGIN`: `profilerName = ncclGetEnv("NCCL_PROFI`

It then calls `handle* = dlopen(name, RTLD_NOW | RTLD_LOCAL)` and `ncclProfiler_v5 = (ncclProfiler_v5_t* "ncclProfiler_v5")` to load the library immediately with local symbol visibility.

- If `NCCL_PROFILER_PLUGIN` is set: attempt to load the library with the specified name; if that fails, attempt `libnccl-profiler-<NCCL_PROFILER_PLUGIN>.so`.
- If `NCCL_PROFILER_PLUGIN` is not set: attempt `libnccl-profiler.so`.
- If no plugin was found: profiling is disabled.
- If `NCCL_PROFILER_PLUGIN` is set to `STATIC_PLUGIN`, the plugin symbols are searched in the program binary.

(Source: `https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html#nccl-profiler-plugin`)

The flow from user API call to loading the profiler plugin:

See Figure 4

The profiler plugin is loaded when creating a communicator (before proxy thread creation). The plugin loading mechanism expects the struct variable name to follow the naming convention `ncclProfiler_v{versionN` which also indicates the API version.

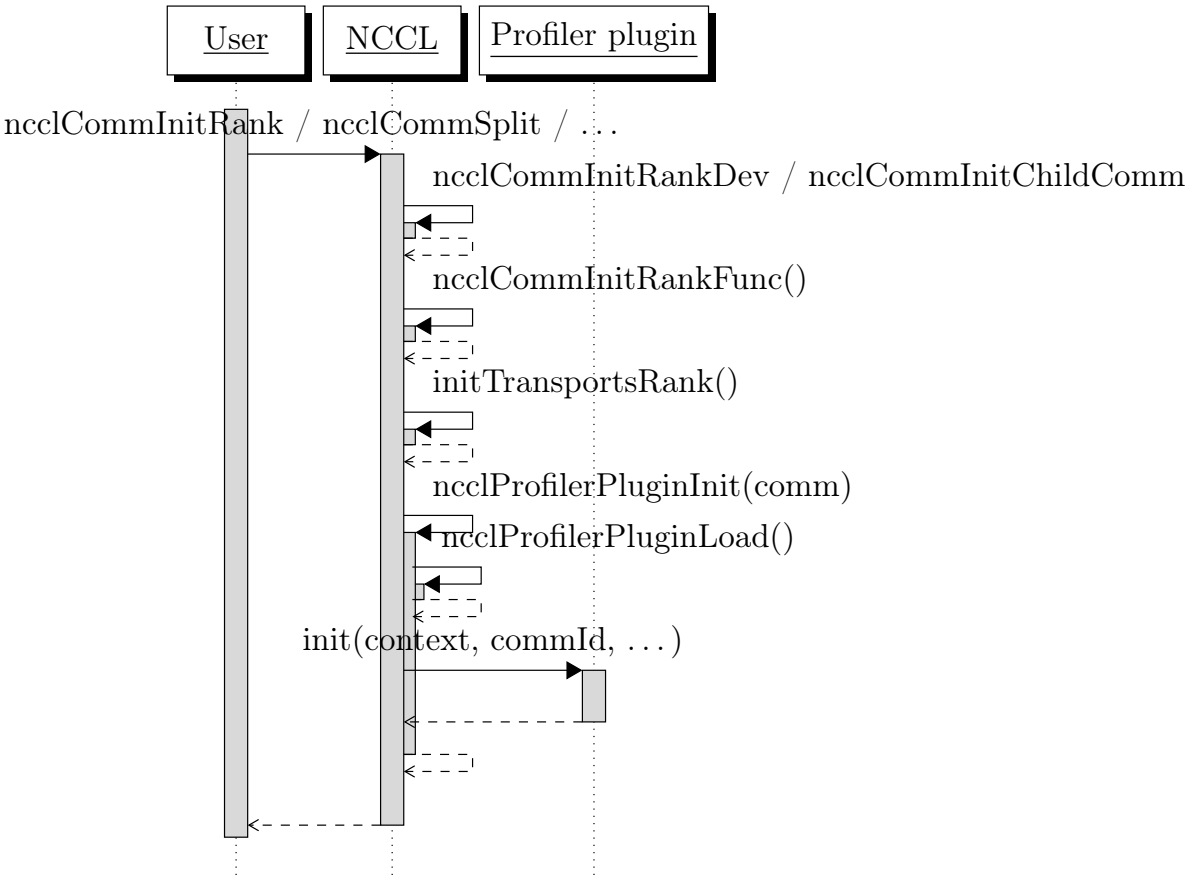



Figure 4: User API → NCCL init → load profiler plugin and call `profiler->init()`.

The profiler API has changed multiple times with newer NCCL releases; backwards compatibility with older plugins seems limited (TODO: fact check. iirc there was some verison breaking feature that didnt allow backwards compatibilty to v2? but current nccl relrease features the structs for older versions @/src/include/plugin/profiler and has a fallback mechanism to older api versions @/src/plugin/profiler.cc i think the problem is that with every new api version release, older profiler definitions under @/$src/plugin/profiler/profiler_vVersionNum.cc$ must be updated to handle/void/hide the new api version features. But looking at the commit history this doesnt seem to happen consistently. if they are not hidden or made backwards compatible, a plugin implemented against an old api version may be handed features from the new api version, which it does not how to handle (when faithfully implementing the old api)).

The exact implementation is in **/src/plugin/plugin_open.cc** and **/src/plugin/profiler.cc**.

## 4.2 The profiler API definition

The plugin must implement a profiler API specified by NCCL by exposing a struct. This struct should contain pointers to all functions required by the API. A plugin may expose multiple versioned structs (backwards compatibility).

```
ncclProfiler_v5_t ncclProfiler_v5 = {
  const char* name;
  ncclResult_t (*init)(...); // NCCL calls this right after loading
  ncclResult_t (*startEvent)(...); // at start of operations/activities
  ncclResult_t (*stopEvent)(...); // at end of these operations/activities
  ncclResult_t (*recordEventState)(...); // to record state of certain operations
  ncclResult_t (*finalize)(...); // before unloading the plugin
};
```

The full profiler API is under **/src/include/plugin/profiler/profiler_v{versionNum}.cc**. As of NCCL v2.29.1, version 6 is the latest; five functions must be implemented. Internally NCCL wraps calls in custom functions (found in **/src/include/profiler.h**).

NCCL invokes the profiler API at different levels to capture start/stop of groups, collectives, P2P, proxy, kernel and network activity. Below the API functions and where NCCL triggers them are explained.

### 4.2.1 init

`init` initializes the profiler plugin. NCCL passes follwing arguments:

```
ncclResult_t init(
  void** context, // out param - opaque profiler context
  uint64_t commId, // communicator id
  int* eActivationMask, // out param - bitmask for which events are tracked
  const char* commName, // user assigned communicator name
  int nNodes, // number of nodes in communicator
  int nranks, // number of ranks in communicator
  int rank, // rank identifier in communicator
  ncclDebugLogger_t logfn // logger function
```

```
);
```

Every time a communicator is created, `init()` is called immediately upon successful plugin load in `ncclProfilerPluginLoad()` (see diagram above). If `init` does not return `ncclSuccess`, NCCL disables the plugin.

> As soon as NCCL finds the plugin and the correct ncclProfiler symbol, it calls its init function. This allows the plugin to initialize its internal context used during profiling of NCCL events.

(Source: **/ext-profiler/README.md**)

`void** context` is an opaque handle that the plugin developer may point to any custom context object; this pointer is passed again in `startEvent` and `finalize`. This context object is separate per communicator.

The plugin developer should set `int* eActivationMask` to a bitmask indicating which event types the profiler wants to track. The mapping is in **/src/include/plugin/nccl_profiler.h**; internally the mask defaults to 0 (no events). Setting it to 4095 will track all events.

TODO: what to do with `ncclDebugLogger_t logfn`?

### 4.2.2 startEvent

`startEvent` is called when NCCL begins certain operations:

```
ncclResult_t startEvent(
  void* context, // opaque profiler context object
  void** eHandle, // out param - event handle
  ncclProfilerEventDescr_v5_t* eDescr // pointer to event descriptor
);
```

As of release v2.29.1 NCCL does not care about the return value. `void** eHandle` may point to a custom event object; this pointer is passed again in `stopEvent` and `recordEventState`. `eDescr` describes the started event; details in **/src/include/plugin/profiler/**.

The field `void* parentObj` in the event descriptor is the `eHandle` of a parent event (or null).

All user API calls to collectives or P2P start a Group API event; when networking is required (multi-node), ProxyCtrl events may be emitted. Depending on `eActivationMask`, further (child) events are emitted in deeper parts of the code – an event hierarchy with several depth levels:

```
Group API event
|
+- Collective API event
|  |
|  +- Collective event
```

```
  |     |
  |     +- ProxyOp event
  |     |  |
  |     |  +- ProxyStep event
  |     |     |
  |     |     +- NetPlugin event
  |     |
  |     +- KernelCh event
  |
  +- Point-to-point API event
  |  |
  |  +- Point-to-point event
  |     |
  |     +- ProxyOp event
  |     |  |
  |     |  +- ProxyStep event
  |     |     |
  |     |     +- NetPlugin event
  |     |
  |     +- KernelCh event
  |
  +- Kernel Launch event

ProxyCtrl event
```

(Source: **/ext-profiler/README.md**)

If the profiler enables tracking for event types lower in the hierarchy, NCCL also tracks their parent event types.

The following diagrams show where NCCL emits `startEvent` and `stopEvent` See Figure 5

Implementation: **/src/init.cc**, **/src/plugin/profiler.cc**.

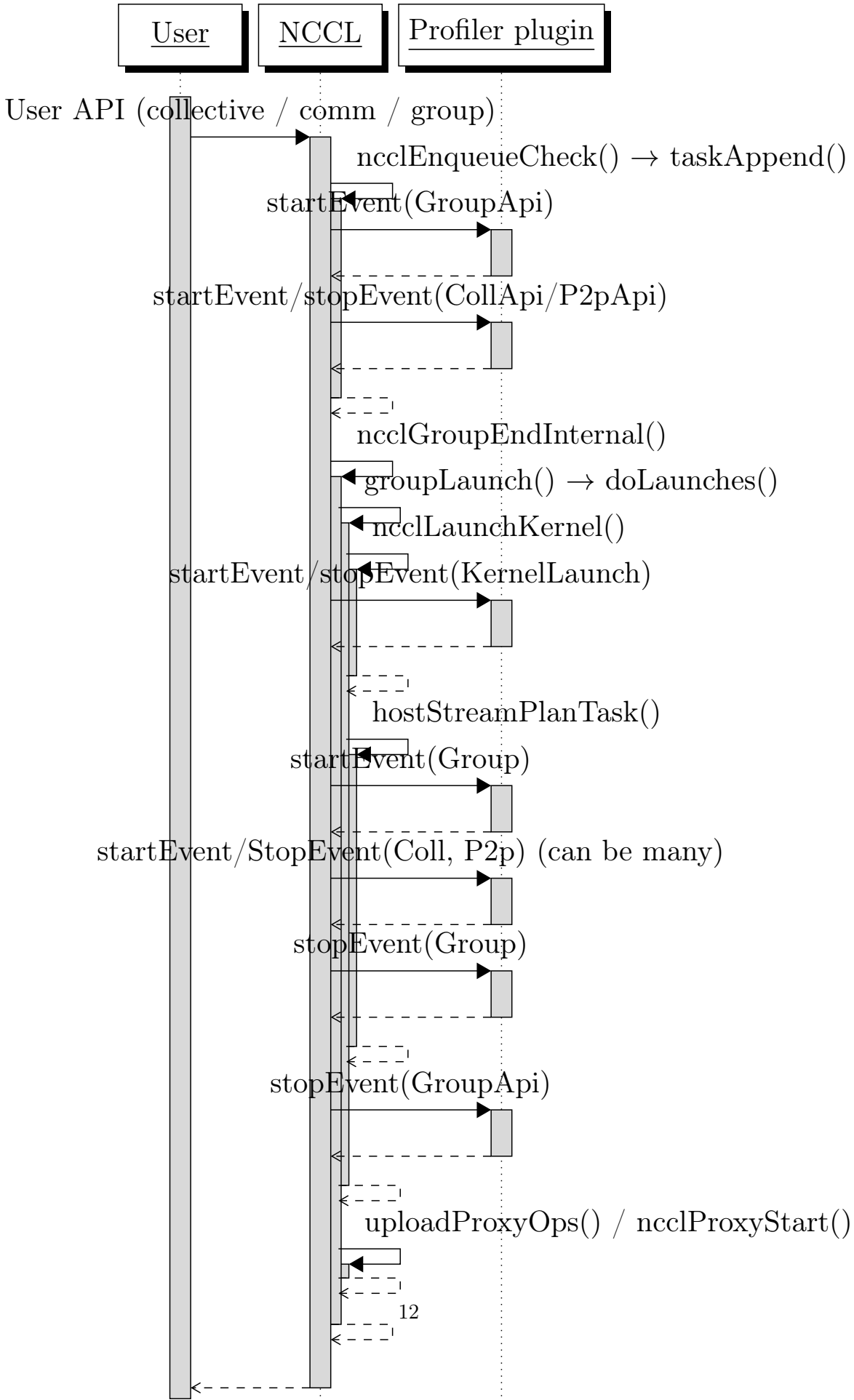
The ProxyProgress thread also emits `startEvent`/`stopEvent` while progressing ops. see Figure 6

`op->progress()` progresses transport specific ops. this is implemented as a function pointer type (defined in **/src/include/proxy.h**). Confusingly the variable is called 'op', although its type is `ncclProxyArgs` and NOT `ncclProxyOp`.

```
typedef ncclResult_t (*proxyProgressFunc_t)(struct ncclProxyState*, struct ncclProxyArgs
    *);

struct ncclProxyArgs {
  proxyProgressFunc_t progress;
  struct ncclProxyArgs* next;
  /* other fields */
}
```

Which specific function this calls depends on the Op. This also decides on which profiler events (ProxyStep Event, KernelCh Event or Network plugin specific events) are being started or stopped. The transport-specific progress functions are in **/src/transport/net.cc**, **coll_net.cc**, **p2p.cc**, **shm.cc**.


User
NCCL
Profiler plugin
User API (collective / comm / group)
ncclEnqueueCheck() → taskAppend()
startEvent(GroupApi)
startEvent/stopEvent(CollApi/P2pApi)
ncclGroupEndInternal()
groupLaunch() → doLaunches()
ncclLaunchKernel()
startEvent/stopEvent(KernelLaunch)
hostStreamPlanTask()
startEvent(Group)
startEvent/StopEvent(Coll, P2p) (can be many)
stopEvent(Group)
stopEvent(GroupApi)
uploadProxyOps() / ncclProxyStart()

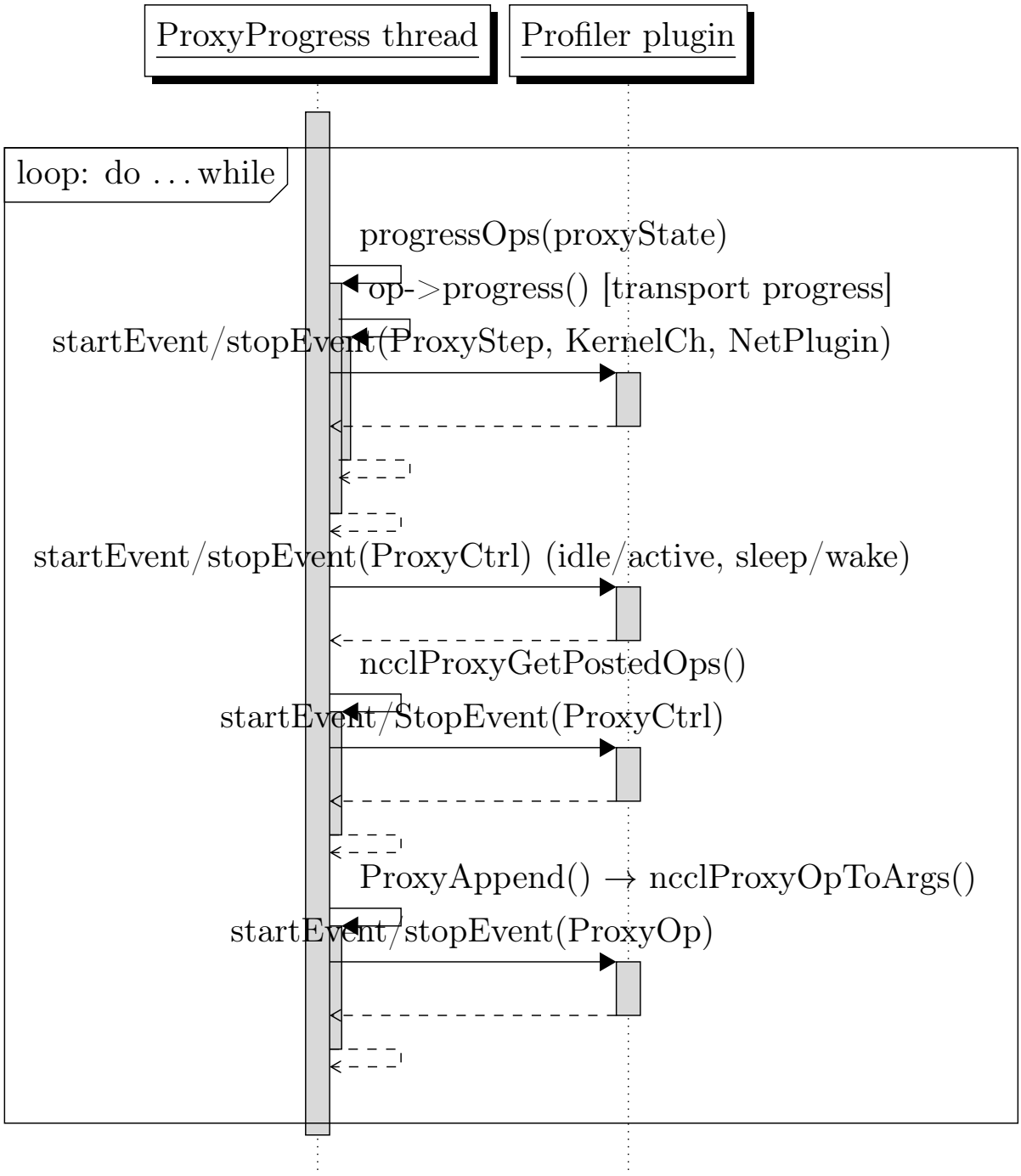



Figure 6: `ncclProxyProgress()` event emission: progress ops (emit ProxyStep/KernelCh/NetPlugin), state transitions (ProxyCtrl), get posted ops, ProxyAppend (ProxyOp).

#### 4.2.3 stopEvent

`stopEvent` tells the plugin that the event has stopped. `stopEvent` for collectives simply indicates to the profiler that the collective has been enqueued and not that the collective has been completed.

```
ncclResult_t stopEvent(void* eHandle); // handle to event object
```

NCCL ignores the return value. `stopEvent` is called in the same functions that call `startEvent`, except for the GroupApi event (see diagram).

#### 4.2.4 recordEventState

Some event types can be updated by NCCL through `recordEventState` (state and attributes). Supported states: **/src/include/plugin/profiler/profiler_v{versionNum}.h**.

```
ncclResult_t recordEventState(
  void* eHandle,
  ncclProfilerEventState_v5_t eState,
  ncclProfilerEventStateArgs_v5_t* eStateArgs
);
```

Called at the same sites as `startEvent`.

#### 4.2.5 finalize

After a user API call to free resources associated with the communicator, `finalize()` is called in `ncclProfilerPluginFinalize()`; afterwards the plugin is unloaded via `dlclose(handle)` in `ncclProfilerPluginUnload()`.

```
ncclResult_t finalize(void* context);
```

See Figure 7

See implementation at **/src/init.cc**, **/src/plugin/profiler.cc**, **/src/plugin/plugin_open.cc**.

#### 4.2.6 name

The profiler plugin struct also has a `name` field.

> The name field should point to a character string with the name of the profiler plugin. It will be used for all logging, especially when `NCCL_DEBUG=INFO` is set.

(Source: **/ext-profiler/README.md**)

TODO: Copy-engine-based events?

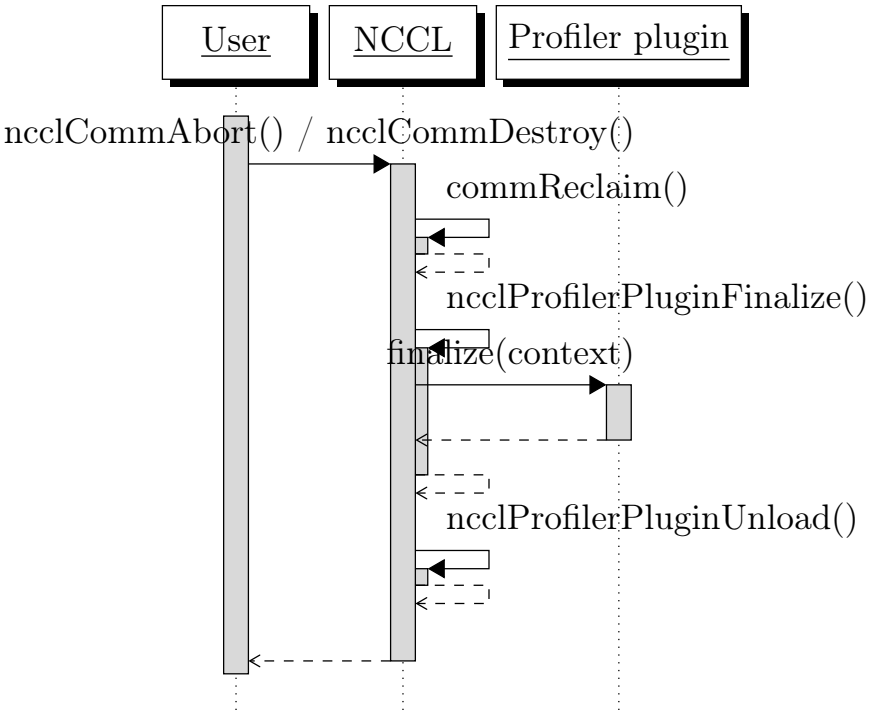



Figure 7: User API → `commReclaim()` → `finalize()` → plugin unload.

### 4.3 Code Examples

The following examples illustrate the profiling behavior under different environment settings and user application.

- single process multiple devices
- mpi multiple processes with multiple device per process (multiple threads per process)
- mpi multiple processes with multiple device per process (single thread per process, utilizing ncclGroup)
- behavior when utilizing ncclGroup for collective operations

TODO add exampless and pictures (profiler that just does logging + application examples)

### 4.4 What is possible with the Profiler Plugin API? Considerations and Pitfalls for (logging, running metrics, CUPTI, ...) when writing the plugin - section title WIP

TODO Logging

- Logging function from `init` (TODO)

- Code snippet: custom logging infrastructure, timestamping

TODO Tracking & running metrics

> Due to the asynchronous nature of NCCL operations, events associated with collectives and point-to-point are not easy to delimit precisely. `stopEvent` for collectives simply indicates to the profiler that the collective has been enqueued. Without proxy and/or kernel activity the plugin cannot determine when a collective ends. With proxy/kernel events enabled, the plugin can estimate when it ends.

(slightly rephrased from **/ext-profiler/README.md**)

void* parentObj

if the plugin developer wants utilize this field, they should ensure that potential address reuse does not create ambiguity to what the parentObj was originally pointing to. Custom memory management is advised. This field is useful when trying to understand which user API call triggered which events of lower level operations or activity such as network activity.

TODO add picture

seqNumber

When profiling is enabled, NCCL counts the number of calls for each type of collective function per communicator.

/src/include/comm.h

```
struct ncclComm {
  uint64_t seqNumber[NCCL_NUM_FUNCTIONS];
  /* other fields */
}
```

/src/plugin/profiler.cc

```
ncclResult_t ncclProfilerStartTaskEvents(struct ncclKernelPlan* plan) {
  /* other code */
  __atomic_fetch_add(&plan->comm->seqNumber[ct->func], 1, __ATOMIC_RELAXED);
  /* other code */
}
```

This value is present in the eDescr for collective events and can be used to identify which collectives operations belong together across processes.

TODO add picture of collectives seqNum

PXN

Unless Setting the environment variable `NCCL_PXN_DISABLE`=0 (default 1), due to PXN (PCIe x NVLink) some proxy ops may be progressed in a proxy thread from another process, different to

the one that originally generated the operation Then `parentObj` in `eDescr` is not safe to dereference; the `eDescr` for `ProxyOp` events includes the originator's PID, which the profiler can match against the local PID. The `eDescr` for `ProxyStep` does not provide this field. However a workaround is possible:

The passed `context` object in `startEvent()` is also unsafe to dereference due to PXN. the profiler plugin developer may internally track initialized contexts and whether the passed `context` belongs to the local process. This is also indicative of PXN.

TODO: fact check if context is actually unsafe to deref? from code it looks like it is (context is read from pool - which iirc is posted to by potentially the originating process, not the progressing proxy thread process?) looking at inspector plugin implementation, it casts the passed context in startEvent(), but doesnt dereference it. so it doesnt crash. looking at example plugin implementation, it dereferences it in startEvent(), which is why it crashes.

TODO verify this behavior one more time: slurm 2695727

TODO

- Code snippet: example CRUD of custom context object
- Code snippet: example CRUD of custom event object

TODO Kernel tracing with CUPTI

- CUPTI extension ID mechanism briefly explained
- Code snippet: where to CUPTI init/cleanup and usage

TODO Changing profiling behaviour at runtime (TODO: check example_profiler, inspector)

### 4.5 Performance and scalability of the Profiler Plugin API

Following Experiments were run to assess the performance and scalability of profiler plugins.

TODO

- synthetic performance measuring applications
- real application

TODO check out potential use of

```
@profiler (line 120 - 182)
#define ENABLE_TIMER 0
#include "timer.h"

#if ENABLE_TIMER
```

```
// These counters are used to measure profiler overheads for different part of the code
// These counters are only useful/meaningful in controlled test environments where there
// is only one thread updating each set of counters, i.e., every communicator has its
// own proxy thread and the network uses only one thread to make progress (this is true
// for net_ib plugin but might not be true for net_socket plugin).
...
```

singlenode

- coll singlenode (1mil iters, 100 warmup iters):
  - sbatch run 1: 7.64µs overhead. averaged 26.40 µs per iter w/o. 34.04 µs w/ profiler
- p2p singlenode (1mil iters 100 warmup iters):
  - sbatch run 1: 7.26µs overhead. averaged 26.66 µs per iter w/o. 33.92 µs w/ profiler
- commops singlenode (10000 iters, 100 warmup iters):
  - sbatch run 1: check slurm 2697340 ...

multinode

- coll multinode (1mil iters, 100 warmup iters):
  - sbatch run 1: check slurm 2697154 ...
- p2p multinode (1mil iters 100 warmup iters):
  - sbatch run 1: not measurable. averaged 21.17 µs per iter w/o. 20.82 µs w/ profiler
- commops multinode (10000 iters, 100 warmup iters):
  - sbatch run 1: check slurm 2697341 ...

TODO run experiments multiple times?

TODO accuracy and reliability/consistency of the the timings of profiler api calls from nccl

Using the profiler plugin when scaled to many gpus across multiple nodes is effortless and did not require any changes for the ran examples and experiments.

# 5 Potential Integration with Score-P

TODO Substrate Plugin Route

TODO Metric Plugin Route?

# 6 Conclusion - What i have shown. Why would you use it? pros & cons

- Customizable
- May require maintenance / active development since NCCL is actively developed
- Low overhead: NVIDIA advertises their `inspector` implementation as efficient enough for "always-on" in production

## 6.1 NCCL_DEBUG

`https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html#nccl-debug`

NCCL already comes with debug logging at various levels of granularity:

- INFO – debug information
- TRACE – replayable trace information on every call
- Further options (v2.2.12 `NCCL_DEBUG_FILE`, v2.3.4 `NCCL_DEBUG_SUBSYS`, v2.26 timestamp format/levels)
- other profiling and tracing tools exists that are maintained by NVIDIA: nsight systems, nsight compute

## 6.2 Known limitations

> Kernel event instrumentation uses counters exposed by the kernel to the host and the proxy progress thread. Thus the proxy progress thread infrastructure is shared between network and profiler. If the proxy is serving network requests, reading kernel profiling data can be delayed, causing loss of accuracy. Similarly, under heavy CPU load and delayed scheduling of the proxy progress thread, accuracy can be lost.
>
> From profiler version 4, NCCL uses a per-channel ring buffer of 64 elements. Each counter is complemented by two timestamps (ptimers) supplied by the NCCL kernel (start and stop of the operation in the kernel). NCCL propagates these timestamps to the profiler plugin so it can convert them to the CPU time domain.

Source: `https://github.com/NVIDIA/nccl/tree/master/ext-profiler/README.md`

## 6.3 Summarize What i have shown TODO

TODO

# 7 TODO

- custom profiler code cleaning