ETH zürich

Networked Systems
ETH Zürich — seit 2015

Distributed Machine Learning in Adversarial Networks

Semester Thesis

Author: Guanshujie Fu

Tutor: Valerio Torsiello, Lukas Röllin, Dr. Muoi Tran

Supervisor: Prof. Dr. Laurent Vanbever

October 2024 to January 2025

Abstract

Distributed training has become essential for large-scale deep learning, yet its efficiency heavily depends on network stability. This thesis presents a framework that integrates network control mechanisms with a distributed training pipeline, enabling the manipulation of network parameters in a cluster environment. Through this framework, this work systematically evaluates the impact of network conditions, such as packet loss, delay, and targeted link disruptions, on distributed training performance. We design a set of network attack schemes, including dumb attacks, one-link attacks, and multi-link attacks. Our experiments indicate that distributed training is highly vulnerable to network attacks, especially with packet loss. This work also lays as the foundation for the future research on improving the resilience and efficiency of distributed deep learning systems in adversarial network.

Contents

Chapter 1

Introduction

1.1 Motivation

Due to the growing size of modern Machine Learning (ML) models and the increasing amount of data required, training a large model today is often carried out in a distributed fashion across multiple workers in a cluster. The emergence of various distributed training frameworks like Megatron-LM[5] and DeepSpeed Ulysses[1] significantly boosts and facilitates the training of large-scale model on extensive cluster. Behind the complex mechanisms introduced in these frameworks, network communication lays the foundation for all functionalities.

Under the simple setting, communication takes place among workers in order to distribute the model parameters, collect local computation results from involved nodes, aggregate and redistribute the results so that training may proceed. In this workflow, the network infrastructure plays a critical role in ensuring efficiency and scalability.

This reliance on the network infrastructure gives rise to an intriguing problem: **How will the network influence the performance of the state-of-the-art distributed training framework?** From a security perspective, understanding how the the network constraints and interacts with these frameworks is essential for protecting the training from such effects and hence optimizing the training workflow.

1.2 Task and Goals

This project originates from the aim of providing an in-depth and insightful look into the correlation between network dynamics and distributed training frameworks by providing a quantized measurement and benchmark of the network's effectiveness over training performance.

At this stage, we put primary focus on the Distributed Data Parallelism (DDP) and Fully Sharded Data Parallelism (FSDP) modules provided in PyTorch in this project. To investigate their performance under varying network conditions, we first need a framework that allows us to manipulate the network topology. This framework should enable customization of parameter including packet loss rate, network delay and bandwidth constraints, based on targeted configuration. Together with the network manipulator, we also need to set up the workload over the cluster to simulate the practical scenario. This includes a pipeline that automates the initialization of distributed training with various popular models, such as ResNet and LLaMA, over custom datasets.

With these basic tools in place, we can then proceed to explore the training performance under combinations of different network manipulations. We propose a set of representative attack schemes that might happen in productive environment and benchmark the training performance under these

attacks, aiming at simulating a wide range of real-world network attacks that distributed training frameworks could encounter.

1.3 Overview

Section 2 provides the necessary background for distributed training including their design and communication pattern. This section establishes the theoretical and practical context for understanding how distributed training is achieved on the network and why the network is important. Section 3 presents the setup of the cluster used in this project, the design of the network manipulator and the distributed training pipeline. Section 4 will introduce different network attack schemes we designed based on the manipulator, simulating conditions such as packet loss, latency spikes, and bandwidth throttling. This section also presents an analysis of training performance under these adversarial scenarios. We also include a quantized evaluation of the network effectiveness for the training. Finally, we provide a brief summary of our findings and propose the future steps of this work.

Chapter 2

Background

In this chapter, we start by introducing the concepts of distributed training and some widely used communication operations in distributed training. We then review the design of the distributed training framework focused in this project, as well as some popular advanced frameworks brought by Hyperscalers, to give a comprehensive understanding of this field.

2.1 Distributed Training

As the name implies, distributed training mainly involves spreading training workload across multiple worker nodes in a cluster. It becomes necessary as the exploding model size and the amount of training data can no longer be fitted into a single device.

2.1.1 Parallelism Scheme

Distributed training primarily relies on the following two key paradigms:

- **Data Parallelism**: In this approach, the training dataset is partitioned across worker nodes, each of which processes a subset of the data while maintaining a full copy of the model.
- **Model Parallelism**: In this approach, the model is split across worker nodes, with each node handling computations for a specific part of the model.

These two schemes build the ground of distributed training. Emergent approaches like **pipeline parallelism**[4][5], **tensor parallelism**[5] and **sequence parallelism**[1][2][3] are somehow variations or combinations of the aforementioned schemes to further optimize resource utilization and communication overhead.

2.1.2 Collective Communication in Distributed Training

Distributed training is built on the top of collective communication operations, which enable efficient data exchange and synchronization among multiple worker nodes in a cluster. While different parallel schemes use different communication patterns, there are a few key operations:

- **All-Reduce:** It is essential for data parallelism to aggregate gradients from all nodes and distribute the aggregated results back to every node, ensuring that each node holds globally consistent model parameter in the cluster.

- **Broadcast:** It is used to distribute data or model parameters from a master node to all other worker nodes. This operation is essential at the start of training when initializing model weights across the cluster.
- **All-Gather:** Each worker node gathers data from all other nodes, creating a complete dataset or tensor copy. This is often used in model parallelism or hybrid approaches to ensure all nodes have access to the required information
- **Reduce-Scatter:** As a combination of reduce and scatter operations, it aggregates values across nodes and simultaneously distributes portions of the aggregated results.

2.1.3 Communication Backend in Distributed Training

Distributed training often operates on a heterogeneous system where multiple devices (CPU/GPU/TPU) might coexist. To make communication efficient and compatible with targeted training devices, hardware vendors provide dedicated communication backend tailored to their devices. The widely used backends include:

- **NCCL(NVIDIA Collective Communications Library)**: Designed specifically for NVIDIA GPUs with highly optimized routines for collective communication operations.
- **Gloo:** A platform-agnostic library that supports both CPU and GPU devices. Developed by Facebook, it is widely used in PyTorch for its compatibility with heterogeneous systems.

2.2 Emergent Distributed Training Frameworks

Several state-of-the-art distributed training frameworks have been developed to address the challenges of training large model. In this section, we bring a brief introduction to some of them.

2.2.1 Distributed Data Parallelism

Distributed Data Parallelism (DDP) is a widely adopted technique for parallelizing training workloads across multiple GPUs or nodes. It deploys the direct data parallelism idea where each worker maintains a full copy of the model and trains on a subset of the data. Gradients computed locally are synchronized across all workers using All-Reduce. DDP achieves high scalability and is well-suited for scenarios where the model can fit into the memory of individual devices.

2.2.2 Fully Sharded Data Parallelism

Fully Sharded Data Parallelism (FSDP) extends traditional data parallelism by partitioning not only the training data but also the model parameters, gradients, and optimizer states across devices. Before forward and backward computation is conducted, All-Gather is first performed to recover the full model parameters on each device temporarily. Instead of All-Reduce, Reduce-Scatter is used to aggregate gradients while redistributing the model in a sharded form. This sharding significantly reduces memory overhead, enabling the training of models that are too large for a single device.

2.2.3 Advanced Frameworks

Although this project does not directly involve the frameworks mentioned in this section, having an overview of their designs could provide an insight into the necessity of our work.

- **Megatron-LM:** A distributed training framework tailored for large-scale transformer models introduced by NVIDIA. It employs a set of advanced parallelism techniques, including tensor parallelism and pipeline parallelism, to efficiently train models with billions of parameters.
- **DeepSpeed:** A framework brought by Microsoft known for its memory-efficient innovations by introducing sequence parallelism known as DeepSpeed Ulysses and an advanced data parallelism known as Zero Redundancy Optimizer (ZeRO), which enables the training for extremely large models.

Our exploration into the impact of network conditions on distributed training will complement aforementioned frameworks' objectives. By evaluating how different network scenarios affect training performance, we aim to provide insights that could inform improvements for these and other similar frameworks.

Chapter 3

Design

In this chapter, we will discuss specific design choices and structures of the overall system from cluster setup and network topology manipulator to the model distributed training pipeline. In the last section, we will present the attacking schemes we designed to simulate real-world network conditions and evaluate their impact on distributed training frameworks.

3.1 Cluster

We shall first start with the setup of a computational cluster which acts as the backbone for our experiments. The cluster is designed to be scalable, flexible, and capable of handling the computational demands of modern machine learning workloads.

The current cluster consists of 16 worker nodes and 1 head node. The head node serves as the control plane for the cluster, managing job submissions, monitoring system health, and coordinating communication between worker nodes. It is also responsible for hosting the workload manager and other administrative tools. Each worker node is equipped with computational hardware to support the machine learning workload.

At this stage, we are putting primary focus on the network side, and as such, the worker nodes in our cluster are virtual machines (VM) equipped with moderate resources. Each VM is configured with 4-Core CPU and 8GB RAM. This configuration provides necessary computational capability for moderate workload like ResNet-18 training, which is sufficient to inspect network-related aspects of distributed training. For future needs, the cluster can be easily expanded by adding more VMs or upgrading the hardware resources.

3.1.1 Workload Manager

The workload manager is a critical component of the cluster, responsible for orchestrating and managing the distribution of tasks across worker nodes.

For this project, we are using Slurm (Simple Linux Utility for Resource Management) as the workload manager. It is the most widespread cluster software for its adoption in high-performance computing (HPC) environments. Slurm provides critical functionalities such as job scheduling, resource allocation, and workload monitoring, ensuring optimal utilization of the cluster's computational and network resources.

The setup of Slurm is straightforward and does not require complex coding. To streamline the installation and configuration process, we provide partially automated scripts that facilitate the setup of Slurm in our cluster.

3.1.2 Network File System (NFS)

Distributed workloads running on different worker nodes should be guaranteed to execute in the same software environment to avoid potential discrepancies caused by variations in dependencies, libraries, or configurations.

Network File System (NFS) provides a centralized and shared storage solution that ensures consistency and accessibility for all worker nodes. This approach simplifies the environment setup and the deployment of distributed training workloads across worker nodes.

3.1.3 Network Topology

For research and development purpose, the network topology deployed in our cluster is designed to guarantee efficient communication while facilitate traffic control and network manipulation. We outline the following two design points for our project.

A. Virtual Network

The network used in this project is a virtual network, realized using Linux's `veth` (virtual Ethernet) interface pairs. This design enables the creation of isolated and customizable network environments, allowing for precise control over traffic patterns and communication parameters. By leveraging veth pairs, we can construct a realistic but fully virtualized network topology that mirrors the behavior of physical networks while maintaining low overhead and high configurability.

B. Network Separation

With veth, the network topology of the cluster is designed to ensure efficient data transfer between worker nodes while maintaining separation between different types of traffic. To achieve this, each worker node is attached to two distinct networks:

1) Management Network: This network is dedicated to user management, SSH reachability, and Slurm coordination traffic. It handles tasks such as job submission, cluster monitoring, and administrative operations. By isolating this traffic, we ensure that management operations do not interfere with the performance of distributed training tasks.

2) Workload Communication Network: This network is exclusively used for communication carried by computational workloads such as gradient aggregation, parameter synchronization, and data exchange between worker nodes during the distributed training. The separate network for workloads also makes it easier to operate network parameters and inspect the performance effects.

C. Centralized Architecture

The cluster should also provide functionality for network manipulation and traffic monitoring. We employ a centralized switch-based architecture, where all worker nodes and the head node are connected to a switch node that acts as the communication bridge.

The switch node runs OpenSwitch, an open-source network operating system, which provides flexibility and programmability for managing network traffic. It efficiently routes traffic between worker nodes and provides a centralized platform to conveniently manipulate the network parameters without the need to operate on all worker nodes. It allows fine-grained control over network conditions, enabling the simulation of various network scenarios (e.g., packet loss, latency, bandwidth throttling) for our experiments.

3.2 Network Manipulator

In this section, we dive into the design of network manipulator, which works as a core component of our project that enables the simulation and control of various network conditions. We introduce

various customizable parameters such as packet loss, latency spikes, and bandwidth throttling in the network manipulator. Below, we describe some key elemental tools utilized in our implementation and the overall architecture design of the system.

3.2.1 Network Traffic Controller

`Linux Traffic Control (TC)` is a powerful tool for managing and manipulating network traffic on Linux systems. It provides fine-grained control over network parameters, making it an ideal choice for implementing our network manipulator. TC operates at the kernel level and is bound to network interface, allowing us to modify network behavior without requiring changes at the application layer. In the network manipulator design, the following 2 functionalities provided by TC are utilized:

A. qdisc Network Hook

The queueing discipline (qdisc) is a fundamental mechanism in TC that determines how packets are queued and scheduled for transmission. It acts as a hook into the kernel's networking stack as depicted in Fig. 3.1, enabling us to introduce specific behaviors such as delaying packets, dropping packets, or limiting bandwidth. Our framework highly relies on the qdisc for controlled network perturbations with the outlined key features.

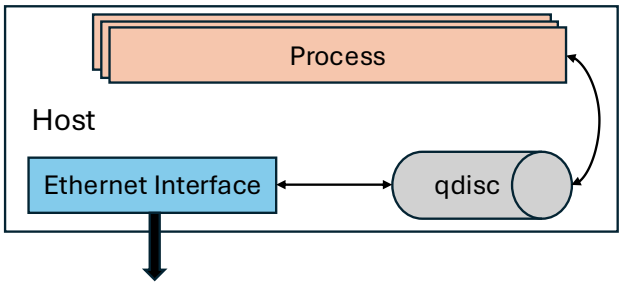


Figure 3.1: **Queueing Discipline in Host System**

1) Queueing Algorithms: A variety of queueing algorithms are available in TC, such as FIFO (First-In-First-Out), HTB (Hierarchical Token Bucket), and NETEM (Network Emulator). In this project, shown in Fig. 3.2, HTB is used to construct a hierarchical network manipulator, while NETEM is employed to control specific network parameters as explained below.

2) Network Emulator: The NETEM qdisc provides the capability to emulate real-world network conditions, such as packet loss, delay, jitter, and reordering. This feature is essential for simulating the effects of network dynamics on distributed training in this project. The following example demonstrates how to apply a 1% loss rate and 5ms delay on all packets passing through network interface `enp1s0`:

```
tc qdisc add dev enp1s0 root netem delay 5ms loss 1%
```

B. u32 Network Filter

The u32 filter is a versatile component of TC that enables packet classification based on specific criteria, such as IP addresses, ports, or protocol types. It works in conjunction with qdisc to apply network manipulations selectively to specific traffic flows. For example, traffic matching user-specified source and destination IP addresses can be directed to a specific NETEM qdisc, simulating network disturbances on the logical link between worker nodes as defined by the attacker.

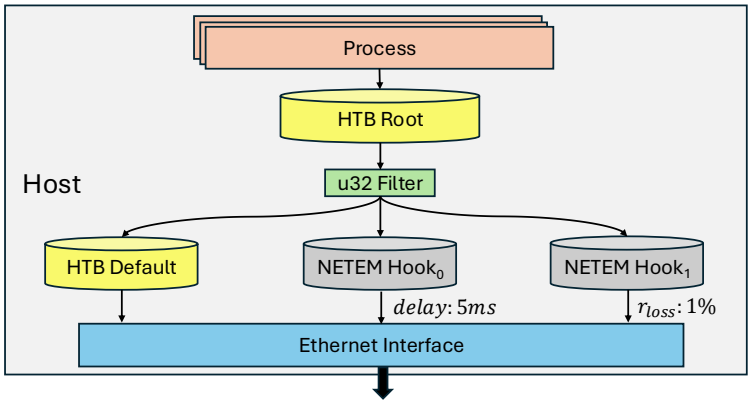


Figure 3.2: **Hierarchical Manipulation with HTB and NETEM**

3.2.2 Network Traffic Monitor

Our system is also designed to capture and analyze network performance during distributed training, both with and without the network manipulation hook. This allows for a comprehensive evaluation of how network conditions impact training efficiency.

`tcpdump` is a packet analysis tool used to monitor and capture TCP network traffic in real-time. It enables detailed inspection of packets, helping to diagnose network behavior and verify the effects of applied network manipulations. Our system utilized `tcpdump` for post-training network analysis to obtain an overview of key metrics like total network volume and packet transmission rates that are essential for evaluating network performance during distributed training.

3.2.3 Network Manipulator System Design

With network traffic controller and monitor in place, we design the network manipulator with user customized configuration and various functionalities. The overall architecture design is presented in Fig. 3.3. As discussed in Section 3.1.3, the network topology of our cluster follows the centralized architecture with a switch node running as the bridge for all worker nodes, and the network manipulator is hence deployed at the switch node.

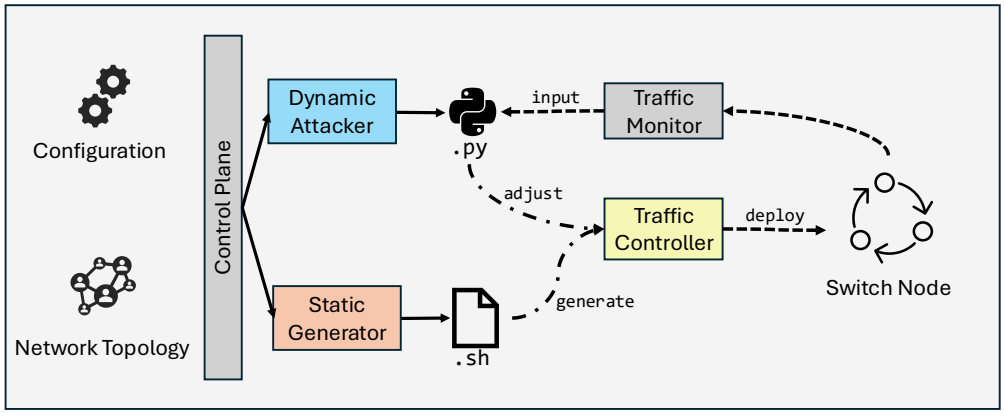


Figure 3.3: **Network Manipulator System Design**

The network manipulator requires two user inputs. The first is the configuration for network manipulation, which includes the setup of network parameters, targeted logical links, and relevant metadata. The configuration should be provided as a `yaml` file that can be easily configured to cause various network impacts and parsed by the control plane. The second is the network topology of the cluster, providing the structural information necessary for applying the specified network modifications.

In our design, we offer two distinct types of network manipulators, each tailored to different objectives and attack scenarios, as discussed below.

A. Static Manipulator

The first type is the static manipulator. The term static refers to the fact that network impacts are directly applied to dedicated logical links and remain unchanged throughout the entire training process. The implementation of the static manipulator is hence straightforward. Based on user configuration, our system provides a static generator that will produce corresponding `shell` script which can be directly executed and deployed on the switch node.

B. Dynamic Manipulator

The second type is the dynamic manipulator. Unlike the static manipulator, the dynamic manipulator responds to real-time inputs like interface throughput from the network traffic monitor, triggering network effects as needed and dynamically adjusting network parameters throughout the training process based on user configuration. This adaptability allows for more realistic simulations of fluctuating network conditions, making it particularly useful for evaluating the resilience of distributed training under varying network constraints.

These functionalities complete the network manipulator system.

3.3 Training Pipeline

In this section, we discuss the design of the training pipeline for a distributed training setup. We present an integrated framework that allows users to configure and select specific distributed training frameworks and models while also providing support for training profiling.

3.3.1 Framework and Model Selector

To accommodate diverse training requirements, our system includes a distributed training framework and model configuration that allows users to choose frameworks and models.

At this stage, we provide support for PyTorch Distributed Data Parallel (DDP) and Fully Sharded Data Parallel (FSDP). These are widely used distributed training frameworks that offer efficient scaling across the cluster while requiring minimal setup as the PyTorch backend takes care of anything necessary.

Additionally, users can specify machine learning models ranging from simple architectures to complex deep neural networks, enabling flexible experimentation across different workloads. The framework currently supports ResNet-18, a widely used deep learning model for image classification, and ToyModel, a lightweight model written intrinsically for easy deployment and verification. This selection allows users to benchmark training performance under varying network conditions while providing a scalable foundation for integrating additional models in the future.

3.3.2 Training Profiler

To provide an intrinsic understanding of how training performance is affected by the network conditions, we integrated the training profiler in the pipeline to benchmark performance. This

profiler collects detailed metrics including training time at different granularity, such as epoch-level and batch-level training time.

Meanwhile, to maintain a primary focus on network impact, we keep a trace on the key collective communication operation calling timestamps and durations by utilizing PyTorch Profiler [**?**]. For DDP, The profiler captures the timeline of `All-Reduce` operations, while for FSDP the profiler monitors `All-Gather` and `Reduce-Scatter` operations.

3.3.3 Distributed Training Pipeline System Design

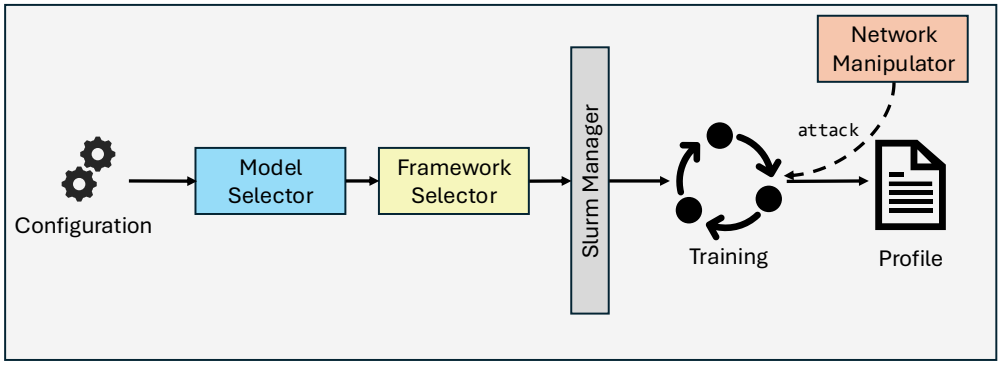


Figure 3.4: **Distributed Training Pipeline System Design**

The system design of the training pipeline is presented in Fig. 3.4. The training pipeline requires user input as a `json` file to configure setup including framework and model choice, dataset configuration, and related metadata.

After framework and model setup, the job queueing and scheduling are handled by the Slurm workload manager to start training process in the cluster. The network manipulator must be manually triggered by the user to orchestrate with the training pipeline to simulate real-world network attack on the distributed training. By integrating manual control over network manipulations, the system enables customized and reproducible experiments, allowing users to systematically evaluate the resilience of distributed training under dynamic network environments.

At the end of the training, the profile of the training and network will be generated automatically for the next step analysis.

3.4 Attacking Schemes

In this section, with both network manipulator and distributed training pipeline in hands, we move forward to designing various attack schemes under different network conditions to simulate real-world scenarios. We will discuss the attack schemes from a high-level perspective, outlining the purpose, characteristics, and general impact on distributed training. The detailed configuration, implementation, and evaluation of these attack scenarios will be examined in Chapter 4.

We categorized different attacking schemes into 3 types, while each type can be operated in both static and dynamic fashion as introduced in Section 3.2.3. All 3 types of attacking schemes are diagrammed in Fig. 3.5 for visualization. The black solid line represents normal logical link. The red half-dashed line indicates attack impacts on logical link.

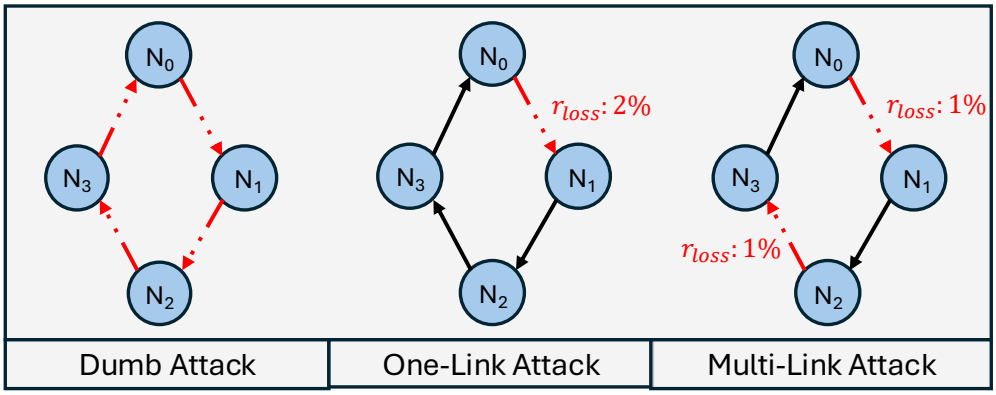


Figure 3.5: **Attack Schemes in 4-Node Ring-Style Topology**

3.4.1 Dumb Attack

As the name implies, the dumb attack scheme lacks any sophisticated design or adaptive behavior. In our context and design, this attack affects all links in the network topology uniformly, applying predefined impacts such as packet loss, latency, or bandwidth throttling across the entire communication network.

This scheme serves as a baseline for evaluating the fundamental impact of network disturbances on distributed training to compare with other attacking schemes.

3.4.2 One-Link Attack

The one-link attack scheme targets at a single, specific logical or physical link within the network topology. This can represent a scenario where a single node during the training experiences network degradation or there exists a bottleneck link between two worker nodes.

Compared with dumb attack, one-link attack could be designed to be smarter and harder to detect. It provides an insight into whether there exists a 'fragile' link in distributed setting, one that if disrupted, could throttle the whole training process and lead to catastrophic degradation.

3.4.3 Multi-Link Attack

The multi-link attack extends the concept of the one-link attack by targeting multiple links simultaneously with same effects. The targeted links can be determined randomly in runtime or predefined by user in the configuration. Compared with one-link attack that affects a single link, the multi-link attack can be viewed as spreading the one-link attack impacts across multiple links within the topology.

For instance, instead of applying a 4% packet loss on a single logical link, the multi-link attack could distribute the same total impact by applying a 1% packet loss on four different logical links. This allows for a broader yet more diffused network disturbance and could provide insights into whether a concentrated or distributed network attack lead to a greater performance degradation.

3.4.4 Attack Cost

To compare the effects of different attack schemes under the same criteria, we also introduce the concept of Attack Cost, defined as:

$$Attack\ Cost = C_{attack} = P_{network} \cdot N_{link}$$

where $P_{network}$ represents the network parameter (e.g., packet loss rate (%) or delay (ms)) applied to each affected link and N_{link} is the number of links affected by the attack. It quantifies the overall severity of network parameter applied during distributed training and provides a standardized way to compare the impact of different network attack.

For instance, a One-Link Attack with packet loss rate $P_{network} = 2\%$ results in $C_{attack} = 2 \cdot 1 = 2$ while a Multi-Link Attack on 2 links with packet loss rate $P_{network} = 1\%$ and $N_{link} = 2$ also results in $C_{attack} = 1 \cdot 2 = 2$. Even though the total network effects in the cluster remains constant, the impact on training may vary due to the distribution of network disruptions across multiple nodes.

Chapter 4

Evaluation

In this chapter, we conduct detailed attack experiments and performance benchmarks to evaluate the impacts of different network attack schemes on distributed training. We analyze how various network disruptions, ranging from dumb attacks to targeted one-link and multi-link attacks, affect the training efficiency and performance.

4.1 Experiments Setup

The experiments are conducted on the cluster utilizing 8 nodes. The training runs are implemented using Python 3.12.3 and PyTorch 2.5.1, with training performed entirely on the CPU using Gloo backend.

For evaluation, instead of using popular models like ResNet-18, we will mainly present the results using the ToyModel which consists of 2 Linear Layers of size (1000, 1000) and (1000, 500) with a ReLU activation function. This setup provides a lightweight but representative training workload and the suitability of ToyModel in our experiments will be further verified in Section 4.2.

4.2 Verification and Comparison

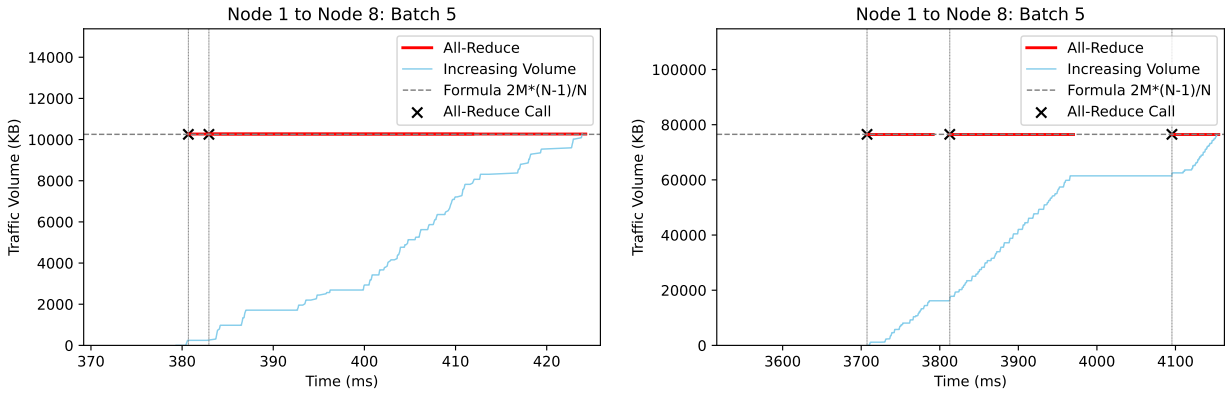
In this section, we present a brief verification and comparison analysis to justify the use of the ToyModel in our evaluation. Specifically, we compare the profiling results of the ToyModel with those of ResNet-18 to ensure that the selected model provides a meaningful representation of distributed training characteristics.

In Fig. 4.1 we represent traffic volume trends for All-Reduce communication in Distributed Data Parallel training using ResNet-18 and ToyModel without any network impacts. The trends are shown for a single iteration, as DDP performs All-Reduce at the end of each iteration to aggregate gradients across worker nodes.

By default, Gloo communication backend performs a ring-based All-Reduce communication to optimize efficiency. This algorithm ensures that gradient synchronization is evenly distributed across worker nodes, minimizing bandwidth congestion. Following the ring-based algorithm, we can then estimate the communication volume per iteration [6]. Given model size of M and cluster size of N, we can get the theoretical transmission volume between neighboring worker nodes:

$$2 \cdot \frac{M(N-1)}{N}$$

Based on the figure and the theoretical analysis, we can conclude the following key observations and statements:



(a) ToyModel DDP Epoch 1 Iteration 5

(b) ResNet-18 DDP Epoch 1 Iteration 5

Figure 4.1: **Comparison of ToyModel and ResNet-18**

1) Similar Increasing Volume Trend: Both models follow a similar increasing All-Reduce volume trend, confirming that ToyModel still experiences relevant communication overhead despite being a simpler model.

2) Matched Theoretical Formula: The formula used for All-Reduce communication holds for both cases, indicating that ToyModel retains essential distributed training characteristics.

Though there exists significant differences in time scale and total traffic volume, these are caused by the intrinsic differences in the model structure, which primarily affect computational performance and do not fundamentally alter communication behavior. Hence, we can safely state that ToyModel remains a suitable choice for evaluating distributed training under varying network conditions in this project.

4.3 Evaluation Results

In this section, we present the evaluation results of our experiments and analysis for the impact of network attack schemes on distributed training. We compare different attack scenarios with various parameters, assess their influence on communication overhead and training efficiency.

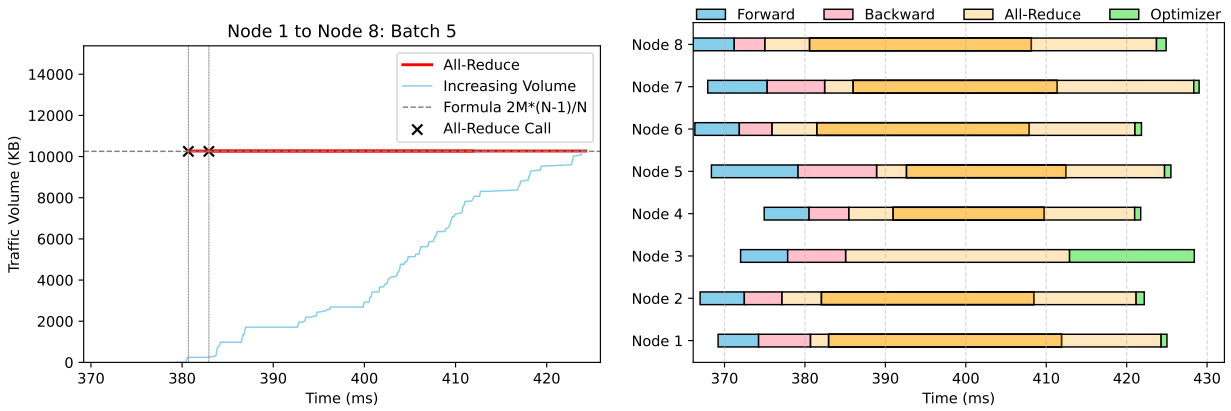
4.3.1 Vanilla Results

We begin by evaluating distributed training performance under normal network conditions, serving as a reference for comparison against various network attack scenarios. This experiment is conducted without any artificial network impairments. As shown in Fig. 4.2, the first figure presents the volume trends during one batch training. The second figure provides a detailed breakdown of computation and communication phases across worker nodes, illustrating the timing of:

1) Forward propagation: model inference computation carried locally in each worker node.

2) Backward propagation: gradient computation carried locally in each worker node.

3) All-Reduce Synchronization: the communication phase in DDP training. PyTorch employs optimizations in the backend by utilizing a bucket-based approach for communication and computation overlap. Instead of synchronizing the entire gradient tensor after the computation, PyTorch aggregates multiple small gradients into buckets and trigger All-Reduce operation when the bucket is full. This explains why there exists 2 All-Reduce operations during the gradient synchronization of ToyModel.



(a) Network Volume Trends (b) Computation-Communication Timeline

Figure 4.2: **Vanilla Experiment Results**

Metrics	**Description**
T_{train}	Total training time in second
$T_{train} \uparrow$	Total training time increasing rate
T_{epoch}	Average epoch training time in second
R_{comm}	Ratio of communication time to training time

Table 4.1: **Evaluation Metrics**

4) Optimizer updates: computation phase after All-Reduce operations are finished to update model parameters.

We also summarize the metrics we utilized to evaluate performance in Table 4.1.

4.3.2 Dumb Attack

We step into experiments using the Dumb Attack on all logical links with various parameters. The targeted metrics evaluations are presented in Table 4.2.

A. Loss Rate 1% - 2%

In this experiment, we introduce {1%, 2%} packet loss across all communication links and evaluate its impact on training performance. The results are presented in Fig. 4.3.

Attack Parameter	C_{attack}	T_{train}	$T_{train} \uparrow$	T_{epoch}	R_{comm}
No Attack	0	4.39	1.0x	0.43	0.50
Loss Rate 1%	8(%)	25.24	5.7x	2.52	0.72
Loss Rate 2%	16(%)	105.27	24.0x	10.64	0.88
Delay 5ms	40(ms)	10.69	2.4x	1.06	0.67
Delay 10ms	80(ms)	16.34	3.72x	1.63	0.79

Table 4.2: **Dumb Attack Performance Degradation**

(a) Dumb Attack 1 with Loss Rate 1%

(b) Dumb Attack 1 Batch 5 Timeline

(c) Dumb Attack 2 with Loss Rate 2%

(d) Dumb Attack 2 Batch 5 Timeline

Figure 4.3: **Dumb Attacks A Experiments Results**

As we can observe, by introducing packet loss rate from 1% to 2%, the total training time increased from 25.24s to 105.27s with 5.7x to 24.0x increasing rate.

The volume trending curve greatly varies as presented. It indicates an unstable and unpredictable behavior during the communication phase. With the increasing packet loss rate, there is a increase in the total network volume indicated in Fig. 4.3a and 4.3c, as retransmissions caused by packet loss contribute to additional communication overhead. The communication phase dominates the training process, as shown by the timeline graphs, highlighting the severe impact of network corruption on distributed training.

Dumb Attack with even small packet loss rates (e.g., 1%) lead to substantial increases in both training time and communication overhead. As the loss rate doubles from 1% to 2%, the impact on training performance becomes disproportionately severe, with communication bottlenecks accounting for the majority of the delay.

B. Delay 5ms - 10ms

In this experiment, we introduce {5ms, 10ms} delay across all communication links and evaluate its impact on training performance. The results are presented in Fig. 4.4.

By adding delay to all packets, the total training time increases as expected, with delays of 5ms and 10ms leading to 2.4x and 3.72x increases respectively.

The performance degradation is not as significant as when packet loss is introduced. This

(a) Dumb Attack 1 with Delay 5ms

(b) Dumb Attack 1 Batch 5 Timeline

(c) Dumb Attack 2 with Delay 10ms

(d) Dumb Attack 2 Batch 5 Timeline

Figure 4.4: **Dumb Attacks B Experiments Results**

is because the delay only slow down the communication but does not cause retransmissions or additional traffic. As illustrated in the volume trending graphs, the overall communication remains intact with a clear stair-like volume trending curve. This pattern reflects the sequential aggregation of gradients. The stair-like trend also indicates that the communication process is stable and uninterrupted, even with the introduction of network delay. This leads to a predictable and linear increase in training time. Furthermore, as shown in the batch training timeline, the communication and computation phases exhibit a stable distribution across worker nodes, indicating that the added delay does not cause significant imbalances or bottlenecks in the training process.

4.3.3 One-Link Attack

Building on the observations from the previous section, we now evaluate the impact of targeted network attacks applied to a single communication link. Specifically, we focus on the link between Node 1 (N_1) and Node 8 (N_8), exploring both static and dynamic configurations.

A. Static Loss Rate on Link (N_1, N_8)

In this scenario, a static packet loss rate from 0.001% to 8% is applied exclusively to the link between N_1 and N_8. We select three of them to present in Fig. 4.5 and the overall increasing trends in Fig. 4.9.

As we can observe, DDP training is highly sensitive to packet loss, with even a small increase

(a) Loss Rate 0.1% (b) Loss Rate 2% (c) Loss Rate 8%

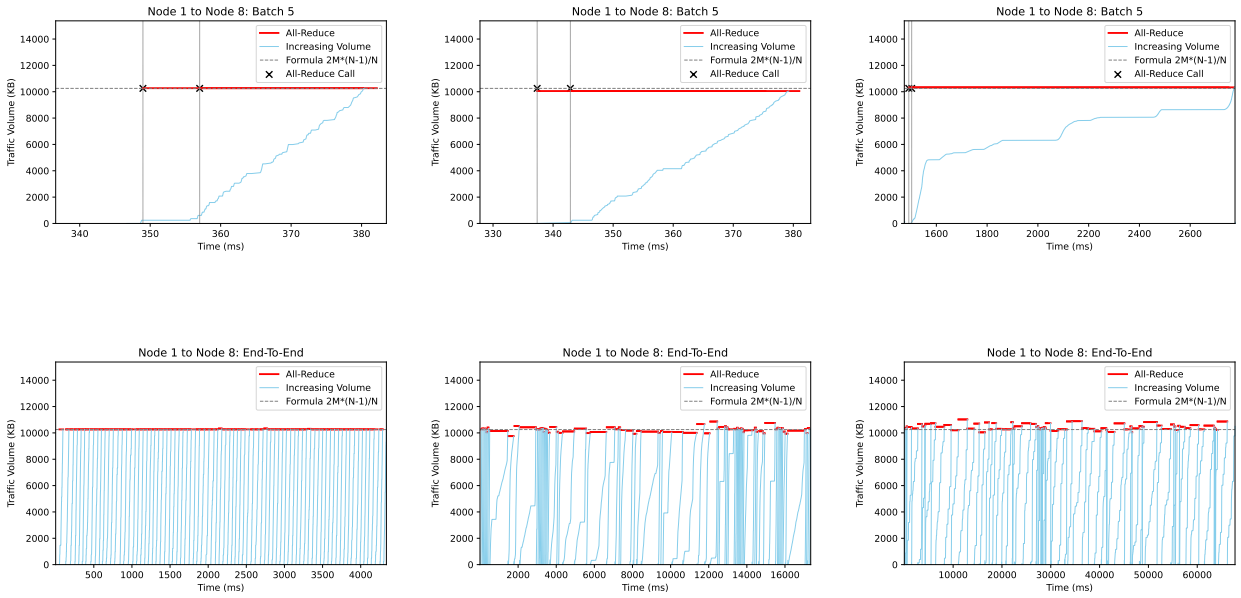
Figure 4.5: **Static One-Link Loss Rate Attack Experiments Results**

in loss rate on a single link leading to significant degradation in training efficiency. From 0.001% to 8% loss rate, the total training time exhibits an exponential increase, highlighting the compounding effects of packet loss on distributed training. Additionally, the packet loss rate introduces significant workload imbalance across worker nodes, as shown in the timelines. The affected link causes delays in gradient synchronization, forcing certain nodes to wait longer for updates, thereby disrupting the uniform distribution of workload. As presented in the middle row of Fig. 4.5, the effects appear 'uniform' across each epoch, indicating that the static loss rate consistently impacts the communication phase in a predictable manner.

B. Dynamic Loss Rate on Link (N_1, N_8)

In this scenario, a packet loss rate from 0.001% to 8% is applied exclusively to the link between N_1 and N_8. The packet loss is designed to take effect only during the communication phase of the training process. We select three of them to present in Fig. 4.6 and the overall increasing trends in Fig. 4.9.

Compared with the static attack, we omit the timelines information in this section because the dynamic attack introduces variations in each iteration and epoch, making the timeline less uniform and presents no similarity in all iterations. This behavior is illustrated in the End-to-End overview in Fig. 4.6, where the dynamic attack exhibits non-uniform effects across the training process.



(a) Loss Rate 0.1% (b) Loss Rate 2% (c) Loss Rate 8%

Figure 4.6: **Dynamic One-Link Loss Rate Attack Experiments Results**

This non-uniformity is expected, as the dynamic attack is tied to the interface traffic throughput in our design. The packet loss rate attack fluctuates the network throughput due to the frequent retransmission, and results in a non-uniform attacking pattern.

C. Static Delay on Link (N_1, N_8)

In this scenario, a static delay from 1ms to 10ms is applied exclusively to the link between N_1 and N_8. We select three of them to present in Fig. 4.7 and the overall increasing trends in Fig. 4.9.

As discussed earlier, the impact of delay is less significant compared to packet loss. Both volume trends and iteration timeline present similarities among increasing delay effects. The total training time shows a linear increase as the delay increases, demonstrating a proportional relationship between the magnitude of delay and its impact on training performance.

Compared with packet loss rate, delay effects are more gentle. While packet loss introduces irregular disruptions, retransmissions, and severe communication overhead, delay only slows down synchronization.

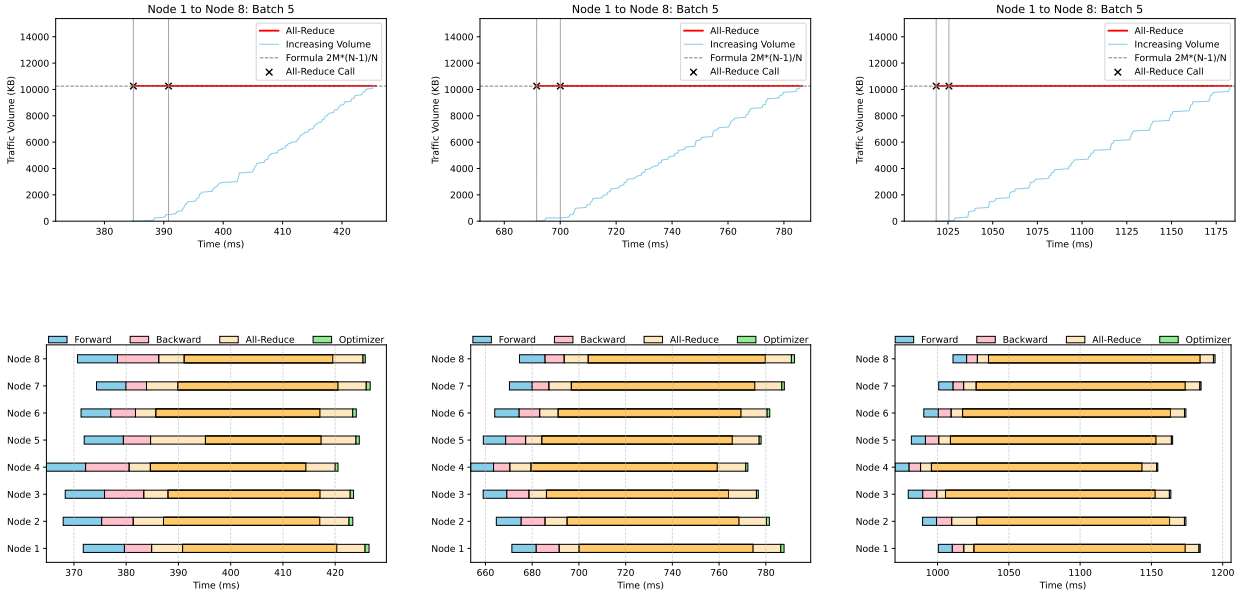
D. Dynamic Delay on Link (N_1, N_8)

In this scenario, a delay from 1ms to 10ms is applied exclusively to the link between N_1 and N_8. The network delay is designed to take effect only during the communication phase, ensuring that the impact is isolated to gradient synchronization and parameter updates in distributed training. The results are presented in Fig. 4.8.

Unlike the static delay, which maintains a consistent slowdown, the dynamic delay attack adapts based on network conditions, resulting in non-uniform synchronization delays across training steps. For this reason we also omit the iteration timeline for this section.

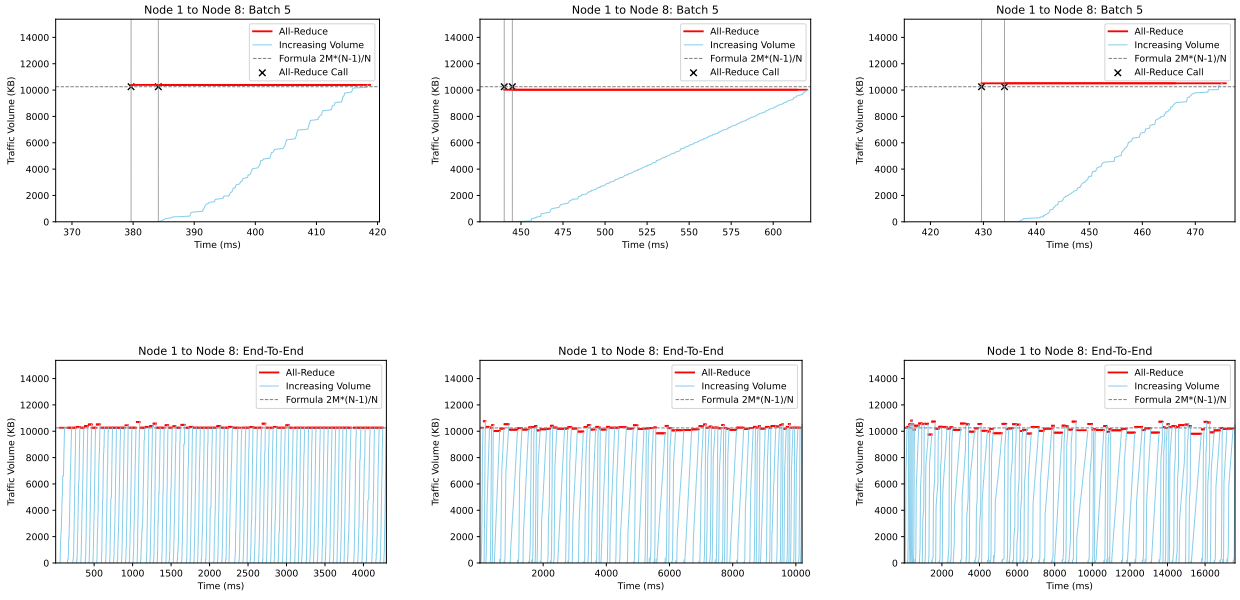
We compare both static and dynamic attacks under the same criteria in Fig. 4.9, measured by Attack Cost (loss rate in % delay in ms). The results highlight the differences in their impact on distributed training performance. We should conclude the following points:

1) Static vs. Dynamic: Static attacks consistently lead to greater performance degradation,



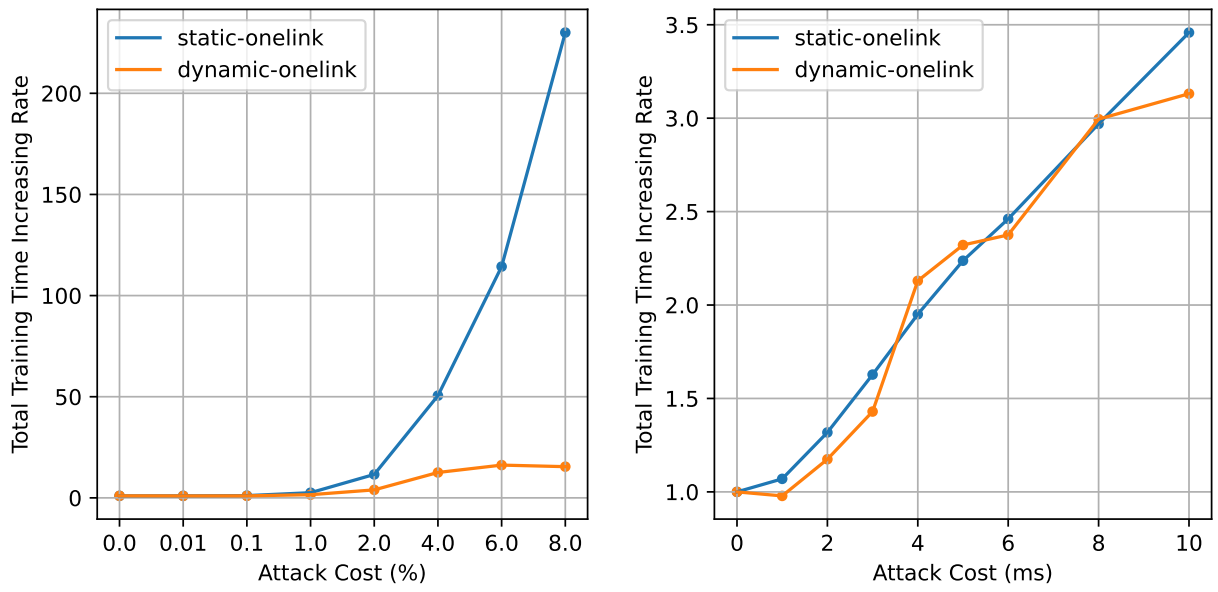
(a) Delay 1ms (b) Delay 5ms (c) Delay 10ms

Figure 4.7: **Static One-Link Delay Attack Experiments Results**



(a) Delay 1ms (b) Delay 5ms (c) Delay 10ms

Figure 4.8: **Dynamic One-Link Delay Attack Experiments Results**



(a) Loss Rate Attacks

(b) Delay Attacks

Figure 4.9: **One-Link Attack Experiments Results**

especially for packet loss. Dynamic attacks, while less severe, introduce variability in training performance.

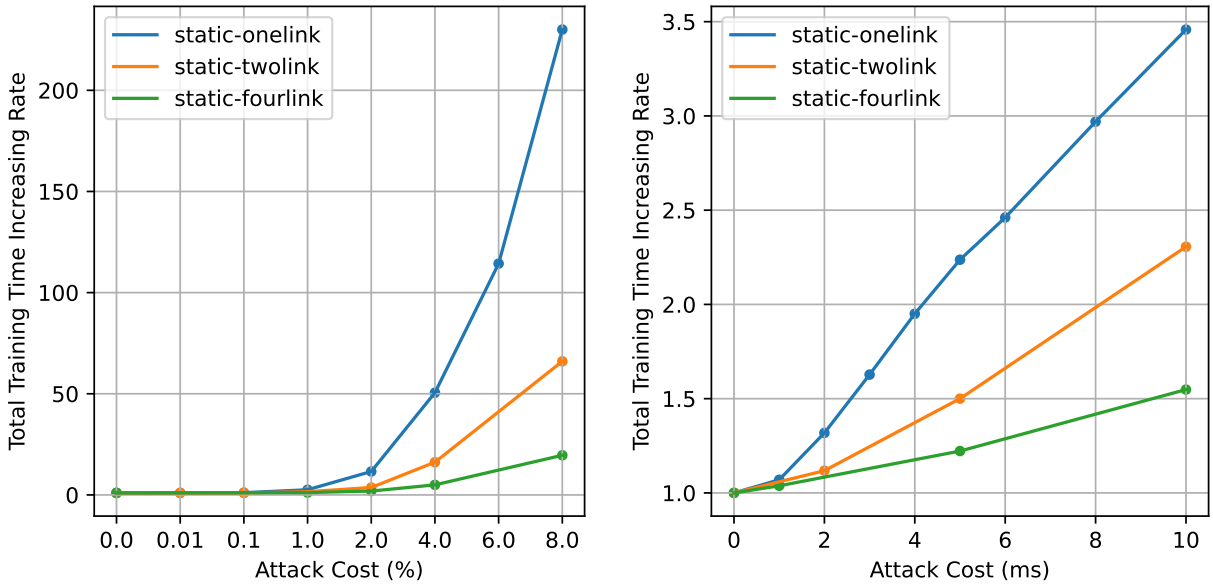
2) Packet Loss vs. Delay: Packet loss has a much greater impact, presenting an exponent increase with larger loss rate under static attack while presenting much less effects under dynamic attack. Delay, on the other hand, presents a linear increase in training time on both static and dynamic attack, making its impact more predictable and manageable.

4.3.4 Multi-Link Attack

In this section, we proceed with experiments using Multi-Link Attack. For this scenario, we extend the experiments in One-Link Attack by spreading the attack into multiple links within the cluster topology. Specifically, we evaluate 2 scenarios: Static Loss Rate on Multi-Link and Static Delay on Multi-Link, where attacks are spread into 2 links $\{(N_1, N_8), (N_2, N_1)\}$ and 4 links $\{(N_1, N_8), (N_2, N_1), (N_3, N_2), (N_4, N_3)\}$. The results are presented in Fig. 4.10.

The figures reveal the impact of Multi-Link Attacks on distributed training performance, focusing on loss rate and delay attacks under static conditions. For loss rate attacks indicaed in Fig. 4.10a, the training time increasing rate grows exponentially as the Attack Cost (loss rate) increases. The results show that static one-link attacks cause the most severe degradation, with total training time increasing drastically as loss rates rise. In contrast, two-link and four-link attacks, which distribute the network impact across multiple links, result in significantly lower degradation at the same Attack Cost. This suggests that distributing packet loss across more links mitigates its impact by reducing the load on any single communication path, demonstrating that the localization of packet loss plays a critical role in performance degradation.

For delay attacks presented in Fig. 4.10b, the training time increasing rate shows a more linear trend as the Attack Cost (delay) rises, with consistent impacts across one-link, two-link, and four-link scenarios. Similar to loss rate attacks, static one-link delay attacks have the most



(a) Loss Rate Attacks

(b) Delay Attacks

Figure 4.10: **Multi-Link Attack Experiments Results**

severe impact on training time, while distributing delay across multiple links (two-link and four-link attacks) reduces the overall effect. However, the differences between one-link, two-link, and four-link delay attacks are less pronounced compared to loss rate attacks, highlighting that delays—though impactful—cause less disruption than packet loss. These findings emphasize the disproportionate sensitivity of distributed training to localized packet loss compared to delays, as well as the benefit of spreading network impairments across multiple links to reduce their overall impact.

Chapter 5

Outlook

Evaluating and benchmarking distributed training is a complex task. In the realistic scenario, especially in a cloud-based environment where nodes from multiple regions may orchestrate and the cluster is deployed in a more complicated network topology, distributed training is subject to dynamic and unpredictable network conditions, making it crucial to design evaluation frameworks that can capture these disturbances effectively.

At this stage, this project provides limited insight into the network impacts on distributed training as the cluster is operating on a simple topology with small size, which may not fully represent the complexities of real-world environments. Furthermore, due to the limited hardware resources, this project does not explore advanced frameworks or evaluate complex models, which restricts the scope of the analysis. We also did not explore the full picture of potential attacking schemes as we primarily focused on basic network impairments, while many other network factors could impact performance.

Future work should try to address the aforementioned limitations, and hence provides a more comprehensive analysis of the interaction between network conditions, advanced frameworks, and complex models, offering deeper insights into the role of network in distributed training.

Chapter 6

Summary

This thesis designs a framework that would operate network parameters in the network topology of operating cluster and setup distributed training pipeline for targeted models. By integrating network control mechanisms with training workflows, the framework allows for a systematic evaluation of network impact on distributed training performance. Along with the framework, this thesis further explores a set of network attacking schemes that partially simulate the real-world network conditions in distributed training environments.

By systematically evaluating these attacking schemes, this work highlights the vulnerability of distributed training to network conditions and establishes a foundation for future studies on network-aware optimizations and protections in large-scale machine learning systems.

Bibliography

[1] Jacobs, S. A., Tanaka, M., Zhang, C., Zhang, M., Song, S. L., Rajbhandari, S., and He, Y. DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models, Oct. 2023.

[2] Li, S., Liu, H., Bian, Z., Fang, J., Huang, H., Liu, Y., Wang, B., and You, Y. Colossal-AI: A Unified Deep Learning System For Large-Scale Parallel Training, Oct. 2023.

[3] Liu, H., Zaharia, M., and Abbeel, P. Ring Attention with Blockwise Transformers for Near-Infinite Context, Nov. 2023.

[4] Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V. A., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM, Aug. 2021.

[5] Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, Mar. 2020.

[6] Strati, F., Elvinger, P., Kerimoglu, T., and Klimovic, A. ML Training with Cloud GPU Shortages: Is Cross-Region the Answer? In *Proceedings of the 4th Workshop on Machine Learning and Systems* (Athens Greece, Apr. 2024), ACM, pp. 107–116.