Fault Tolerant Collective Communication

Karan Grover, Deepti Rajagopal

May 12, 2022

1 Introduction

Distributed Data Parallel[2] training makes use of collective communication algorithms to exchange gradients. A class of DDP algorithms make use of synchronous collective communication algorithms[5] to do gradient updates. These algorithms ensure that the gradients communicated are not stale, and hence the trained model is not only accurate, but converges faster. However, as we have observed in our experiments, these algorithms are not fault tolerant. In this project, we want to explore different techniques to make these algorithms fault tolerant. We discuss our techniques and how we wish to analyse them in the context of Accuracy, Time-To-Recovery and Memory-requirements. We primarily focus on the Ring AllReduce algorithm as it is the bandwidth optimal algorithm for AllReduce.

2 Background

2.1 Ring AllReduce

In Ring AllReduce, each process adds its local chunk to a received chunk and sends it to the next process, i.e., every chunk travels all around the ring and accumulates a chunk in each process. After visiting all processes once, it becomes a portion of the final result array, and the last-visited process holds the chunk. Finally, all processes can obtain the complete array by sharing the distributed partial results among them. This is achieved by doing the circulating step again without reduction operations, i.e., merely overwriting the received chunk to the corresponding local chunk in each process. The AllReduce operation completes when all processes obtain all portions of the final array. Figure 1 shows how this algorithm works.

2.2 Pytorch

Pytorch[6] is an open-source Machine Learning Framework based on the Torch Library. It is a widely-adopted scientific computing package used in deep learning research and applications. It supports a wide variety of machine learning algorithms.

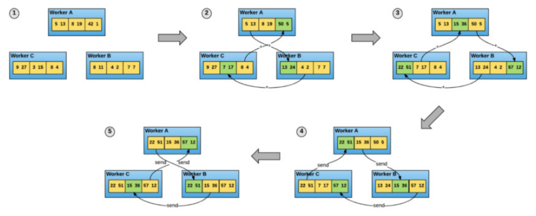


Figure 1: **Ring AllReduce Algorithm**

2.3 Data Parallelism

We used Pytorch[6] specifically for its support for Distributed Data Parallel[2] training. It provides a *DataParallel* interface for single-process multi-thread data parallel training using multiple GPUs on the same machine, and *DistributedDataParallel* for multi-process data parallel training across GPUs and machines. In our project, we use *DistributedDataParallel*. Data parallelism enables distributed training by communicating gradients before the optimizer step to make sure that parameters of all model replicas are updated using exactly the same set of gradients, and hence model replicas can stay consistent across iterations. This communication is done using a collective communication algorithm.

2.4 GLOO

Gloo[1] is a collective communications library written in C++. It comes with a number of collective algorithms useful for machine learning applications. These include a *barrier*, *broadcast*, and *allreduce*. Transport of data between participating machines is abstracted so that IP can be used at all times, or InifiniBand (or RoCE) when available. Gloo is built to run on Linux and has no hard dependencies other than libstdc++.

3 Design

Before diving into the design of our approach, we list a few of the key goals that we had for Fault Tolerant Ring Reduce:

- **Usability**: We wanted to modify RingReduce in such a way that we can package the Fault Tolerant RingReduce algorithm into the existing Pytorch library, so that users can simply call the `ft_send` and `ft_recv` commands instead of the standard `send` and `recv` commands. In order for this to happen, we only make changes in the Pytorch and the Gloo library.

- **Correctness**: In order to implement correctness and fault tolerance, we would need to do some form of checkpointing/logging, in order to track the state of the system just before any of the nodes crash. One of the main things we wanted to evaluate were if adding correctness would impact the performance of Fault Tolerant Ring Reduce.

- **Performance**: Adding fault tolerance to RingReduce was bound to increase the latency of both the `send` and `recv` commands, but we wanted to explore and evaluate how much the performance degrades by introducing fault tolerance, and if the tradeoffs of fault tolerance are worth it. It wouldn't be worth it if the performance of Fault Tolerant Ring Reduce was significantly poor, that users would find it worthwhile to just re-run RingReduce if it fails instead.

With the above goals in mind, we delve into the details of the design we implemented in the following subsections.

3.1 Crash Discovery

It is common knowledge that collective communication algorithms are not fault tolerant. This means that when one node goes down, all nodes go down and the process has to be restarted from scratch. We made the exact same observation in our experiments, however the reality is more nuanced. We observed that when a node crashes, only the nodes communicating to this node crash. This is because an exception is thrown in the Gloo library when the connection between two nodes is closed. However, this exception is discovered in the application layer when an explicit `send` or `recv` is called. Hence, a crashed node is discovered when an attempt is made to communicate with it. Once this exception is discovered, the neighbouring nodes die with a *Runtime Exception*. The process crash on these nodes is discovered when their neighbouring nodes make an attempt to communicate with them. This is how a crash cascades through the system.

To make this whole process fault tolerant, we made sure that none of the nodes, except the one that has crashed, die because of the Runtime Error. Given that the crash cascades through the system, catching the error would mean that only the neighbouring nodes now detect the crash. Since they dont die, no other node in the system is able to discover the crash. To get around this problem, we induce fake crashes in the system. We do this by closing connection to our neighbouring node whenever we detect a crash. Closing the connection is a way of notifying the neighbouring nodes that a crash has been detected in the system. Once everyone is notified that a crash has been detected, they can all go into the Crash Recovery Protocol.

3.2 Peer Discovery

The first step in the Crash Recovery Protocol is Peer Discovery. This is also the first step in any distributed data parallel implementation of a machine learning algorithm in Pytorch. This is done using a function called `init_process_group`. This function blocks until all processes have joined. It initializes the default process group and also initializes the distributed package. To be able to discover peers, `init_process_group`, needs to know the total number of nodes that will join, its current rank in the group and the way this information will be communicated to it. By default, all nodes communicate with node0 using TCP and after the data is exchanged, everyone stores this information in their memory.

In the context of fault tolerance, once a node crashes and comes back up, everyone needs to exchange this information again. We observed that using the default TCP method of initializing results in some nodes using stale data. To overcome this problem, we configure the nodes to communicate with each other using a shared file. This file is shared using NFS and each node writes information about how to contact them on the shared file by acquiring file locks. This makes sure that there is no stale data in memory and everyone has one source of truth about how to contact other nodes.

After a node crashes, all nodes detect that a crash has happened using our Crash Discovery Protocol described in Section 3.2. Once they are notified, they need to wait for stale on-file data to be cleared before they can re-initialize the process group. This is done by the node assigned Rank 0. Hence, as soon as Rank 0 is notified that a crash has happened, as a part of its crash recovery protocol, it will delete the old shared file, it updates the metadata file which has location of the new shared file. Once this is done, Rank 0 signals to all nodes that they can proceed with their Crash Recovery Protocol. On receiving the signal, everyone clears stale objects in their memory and tries to re-initialize the process group.

3.3 Logging

It is important to note that, in collective communication algorithms, the number of sends and receives and their order is strictly defined. In order to be able to implement correctness in the Fault Tolerant version of the Ring Reduce algorithm, it is necessary to track the exact starting point of the crash. We use logging in order to find out the point at which any node crashes, if it does. Logging is a well-known technique that is widely implemented in distributed systems to enable fault tolerance. It involves recording the operations that produced the current state, so that they can be repeated, if necessary. In our specific usecase, logging would mean that each node logs what it sends and receives, that is:

- Type of operation: This could be either a send or receive
- Parameters to the function: This includes a list of all the parameters that are required for a failed node to execute when it comes back up during recovery, and includes the tensor split and indices corresponding to it.

We now go over the details of how we finalized an approach for logging the above information and provide correctness for Fault Tolerant Ring Reduce.

3.3.1 Attempt 1

In our initial implementation, we had used separate log files in order to log a node's `send`, and its successor's `receive` operations. This soon resulted in a major bug and hence a correctness problem, wherein one or multiple nodes would hang. This is explained using a scenario, as shown in Figure 2 and Figure 3. Suppose that node0 crashes at *iteration* 0 after the `send` function was successfully executed, but before it could write this to the log file *0.log*. Now that the `send` function completed, and node1 received this buffer, it then proceeds to write to the *1.log* file which goes through successfully. This now means that node1 proceeds to *iteration* 1. It is important to note that while node1 has proceeded to *iteration* 1, node0 would still resume from *iteration* 0. Now, when node0 comes back up, as a part of the recovery protocol, it reads the log file it created, *0.log*, and sees that there's no entry on it, which means it wasn't able to log the send operation. Because of this, node0 retries the send operation, and node1 at *iteration* 1 ends up receiving node0's tensor for *iteration* 0. This is a major correctness concern.

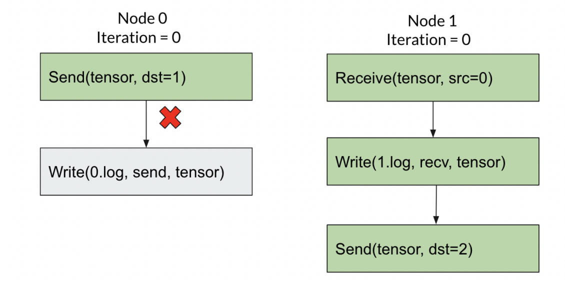


Figure 2: **Attempt 1: Failure Scenario**

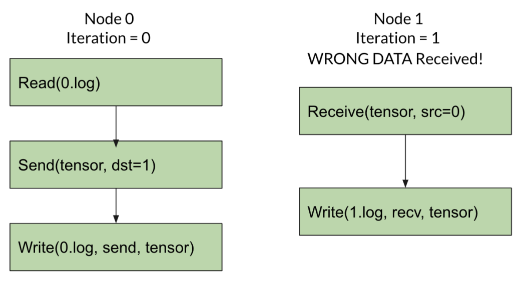


Figure 3: **Attempt 1: Node Recovery**

3.4 Attempt 2

The correctness problem in the previous approach arises due to the fact that we used separate log files to track the state of a pairwise send-receive communication. As is observed in ring reduce, communication always happens pairwise, and hence, the pairwise state must be stored in such a way that both the nodes of the pair are aware about whether a `send` / `receive` has succeeded or not. In order to achieve this, we create a pairwise log file for every pair that communicates, which means, each node writes to two files - one, the file onto which both node *i* and its successor *(i + 1) % n* communicate, and two, the file onto which the node *i* and its predecessor *(i + 1) % n* communicate. This is essential so that, every time a `send` or `receive` between a pair has succeeded, the log file corresponding to that pair will have a value, which states that at least one of the nodes considered it as a success. Figures 4 and 5 demonstrate how this approach can overcome the correctness concern of the previous approach. As can be seen, in the same scenario as depicted in Figures 2 and 3, in *iteration* 0, node 0 crashes after successfully sending a buffer, but without having written to the log file *01.log*. This results in node 1 receiving this buffer from node 0, writing to the log file *01.log*, and then proceeding on to *iteration* 1. Now, when node 0 comes back up and runs the recovery protocol, it reads the log file *01.log*, and sees that there has been a `send` / `receive` or both, and hence skips the send in *iteration* 0, and moves on to *iteration* 1. This, therefore, fixes the correctness problem described above, and RingReduce succeeds eventually.

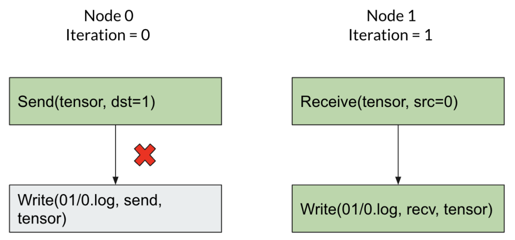


Figure 4: **Attempt 2: Failure Scenario**

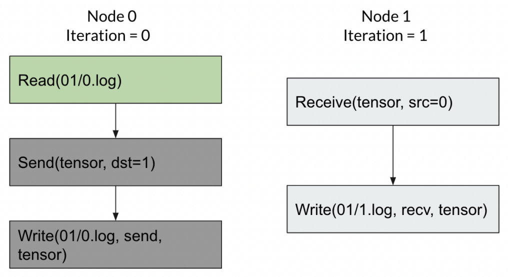


Figure 5: **Attempt 2: Recovery**

4 Challenges and Learnings

During the implementation of our, we faced a number of challenges. Some of them are described here.

Gloo Codebase: We spent considerable amount of time understanding the Gloo codebase and how it interfaces with Python. We couldn't get gdb to run with the C code, especially one that interfaces with Python so our debugging strategy was mostly adding print statements. Through exploring the codebase, we learned a lot about of Python C Bindings work and how to use C++ code with Python.

TCP vs Shared File: Initially we spent a lot of time debugging what would turn out to be stale data in the C++ objects of node when try to connect to a node that had just come back after crashing. Realizing that the cause was stale data required digging deep into the C++ code and with the help of the Professor, we were able to solve the problem using a shared file.

Cascading Failures: We spent a lot of time understanding the mechanism of cascading failures and how to handle them in our code. After a lot of thinking and trying different ideas, we came up with the approach of inducing fake crashing in the system.

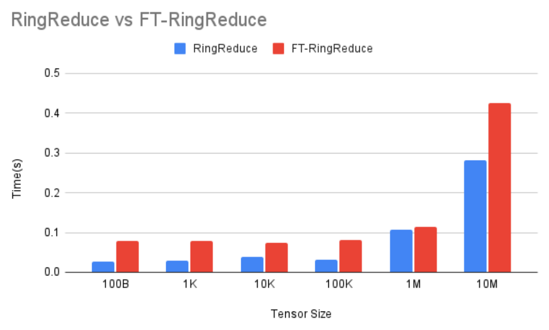


Figure 6: **Time to Converge for Vanilla Ring Reduce and Fault Tolerant Ring Reduce**

5 Evaluation

Our evaluation setup consists of 4 Linux servers on Cloudlab. Each node has 4 CPUs, 16GB RAM and 16GB hard disk. However, all our experiments were performed on a single node with four different processes. This doesn't affect the correctness of our approach. We compared the time taken for Ring Reduce to converge for the algorithm we implemented in Assignment 2 (Vanilla Ring Reduce) and the one we implemented for this project(Fault Tolerant Ring Reduce). The results are shown in Figure 6. We observe that the Fault Tolerant Ring Reduce takes more time to converge than Vanilla Ring Reduce. This is because we log the tensor in every `send` and `recv` request - as described in Section 3.3. We observe that our overhead is a little less than 2x. If the Vanilla Ring Reduce crashes just before converging, we would have to start it again from scratch. However, in the Fault Tolerant Ring Reduce all but one nodes would only pause until the crashed node comes back up and resume from where they paused. Hence, in case of a crash, vanilla Ring Reduce would take twice the time to converge, but Fault Tolerant Ring Reduce would complete earlier.

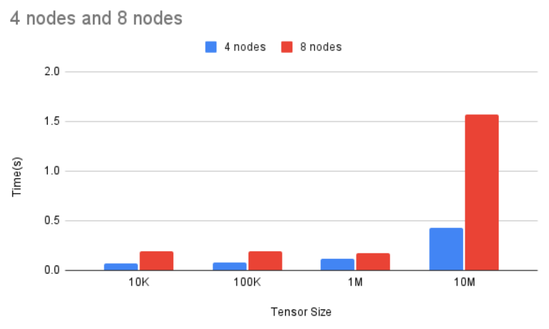


Figure 7: **Time to Converge for Fault Tolerant Ring Reduce with different nodes.**

We also evaluated different ways of saving tensors. Pytorch provides an efficient way of serializing and saving tensors on disk. We also have an option to convert a tensor to a numpy array and save that on the disk as a string. We document our observations in Figure 8. We observe that for smaller tensors, converting to numpy and saving as a string is faster. But as the tensor size increases, `torch.save` approach is faster. We assume this is because `torch.save` writes some extra metadata to speed up the loading process which is a bottleneck for smaller tensors. For bigger tensor, this extra cost of writing metadata is not too much.

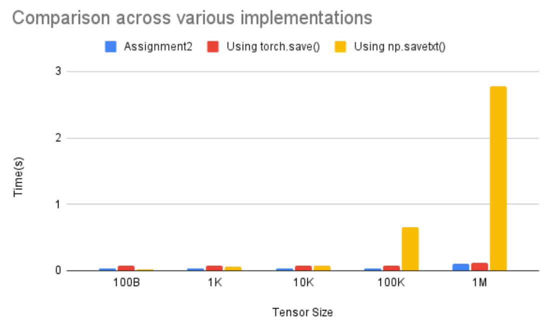


Figure 8: **Time to Converge for Vanilla Ring Reduce, Fault Tolerant Ring Reduce with `torch.save` and Fault Tolerant Ring Reduce with `np.savetxt`**

We evaluated the scalability of our approach by recording time for convergence for 4 and 8 nodes. We have presented the results in Figure 7. We notice a considerable increase in time to convergence when going from 4 to 8 nodes. This is because with double number of nodes, we approximately double the number of `sends` and `recvs`, hence doubling the number of writes to disk.

Finally, we show the state of the system with four processes where multiple processes are crashing multiple times. We show how other nodes wait for crashed node to come back up, how much time the crashed node takes to apply the log and how they start working again in synchrony. We show this in figure 9 and 10. As we can see, Node 1 goes down twice and while it's down all other nodes are in recovery mode waiting for it to come back up. Once it's back up, all nodes together can start sending and receiving tensors in the normal operation. In Node 1, we also see a short purple line after the red line, and for Node 2 and Node 3 as well. This is the time the node spends in applying the log. Once the log is applied, it can start sending and receiving tensors just like it does in normal operation.

6 Related Work

Fault tolerance for Collective Communication has been attempted in the past. Tree-Based Fault-Tolerance[4] provides fault tolerance ignoring all responses that come in after a specified timeout, instead of killing all the processes in the system. Although this seems like a feasible approach, it must be noted that when used for Machine Learning models, this can cause correctness issues and may impact the convergence and/or accuracy of these models.

Parameter Server[7] is a well-known approach, where fault tolerance is provided by the replication of parameter servers. Additionally, Parameter Server replicas are managed by controller machines that form a Paxos cluster, which is used for consensus. All the changes across these replicas are communicated through the Two Phase Commit protocol.

The authors of Hoplite[3] describe it is an efficient and fault-tolerant collective communication layer for task-based distributed systems. For task-based distributed systems, traditional collective communication libraries are typically an ill fit, because of the pre-requisite that the communication schedule be known before runtime and additionally, they do not provide fault tolerance. They state that when a task fails, the data transfer schedule adapts quickly to allow other tasks to keep making progress.

7 Future work

Some of the interesting things we'd like to explore in the future are as follows:

- **Discard failing nodes from the network**: If a node fails, discard the node, and proceed with RingReduce after performing a new `init_process_group` with just the remaining nodes. Doing this just adds an additional overhead of re-initializing the process group among the nodes that are up and running, and doesn't impact correctness. We'd expect this to perform better than Fault Tolerant Ring Reduce because the healthy nodes are not having to wait for the nodes that are temporarily down to come back up. This also would address node failures that are not transient - even if some nodes are permanently down, since we would re-initialize the process group from scratch with as many nodes as there are available. In such an approach, the major challenge is to find the new ranks of all the nodes. One way this can be addressed, is by using a lock service like Chubby or Zookeeper, that would initializing an n-sized bit vector with the value 0 at each position, where n is the number of nodes that are running at that point. Nodes would first announce their liveness by acquiring a lock on the file and setting the bit at their current position to 1. Once everyone knows, after a timeout, which nodes are up, they will start acquiring a lock again and picking their own rank. Since we know the number of nodes that are up, we would also know the *world_size*.

- **Checkpointing**: Although we had initially come up with a checkpointing algorithm for Fault Tolerant Ring Reduce, we realized that it may be faster and less memory-intensive to perform logging rather than checkpointing, because checkpointing requires that we save the whole tensor at specific points in time (after each `send` / `recv` if we follow the algorithm similar to logging). Some of the things that we'd like to implement in this regard, is that, instead of saving the checkpoint after each iteration of `send` / `recv`, we could perform a hybrid checkpointing + logging algorithm, where we'd checkpoint after say, every 10 iterations, and perform logging for the rest of the iterations. This would mean that when we replay the log during node recovery, we can discard all the logs that are older than 10 iterations, hence saving us some time or compute by not having to redo all the operations from scratch

References

[1] Facebook. Gloo. https://github.com/facebookincubator/gloo.

[2] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *CoRR*, abs/2006.15704, 2020.

[3] H. U. of Jerusalem. Hoplite. https://bit.ly/3FEUZqd.

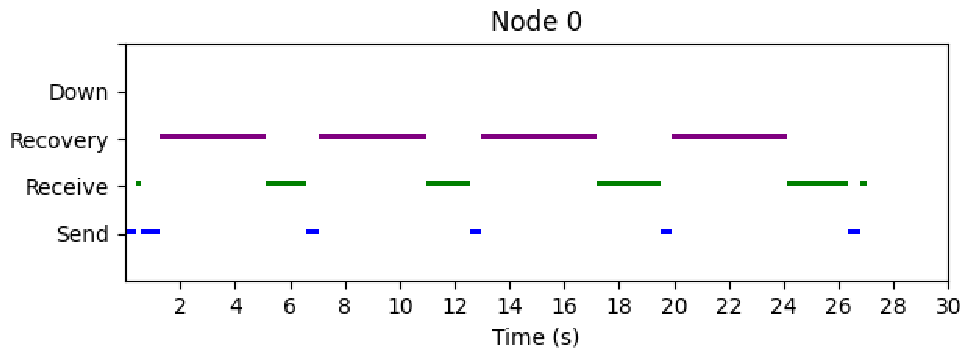
[4] H. U. of Jerusalem. Tree based fault tolerant collective operations for mpi. https://bit.ly/3FEUZqd.

[5] U. of Texas. Collective communication. https://bit.ly/3NaMJk3.

[6] Pytorch. Pytorch. https://pytorch.org/.

[7] M. Research. Project adam. https://bit.ly/3L4ORIV.

8 Appendix



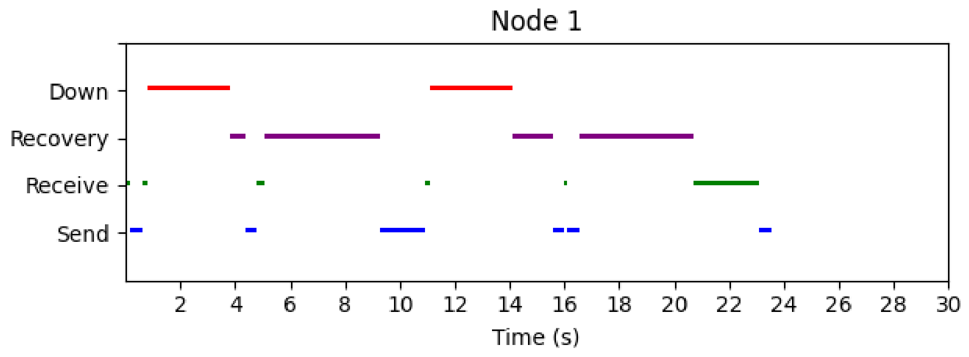
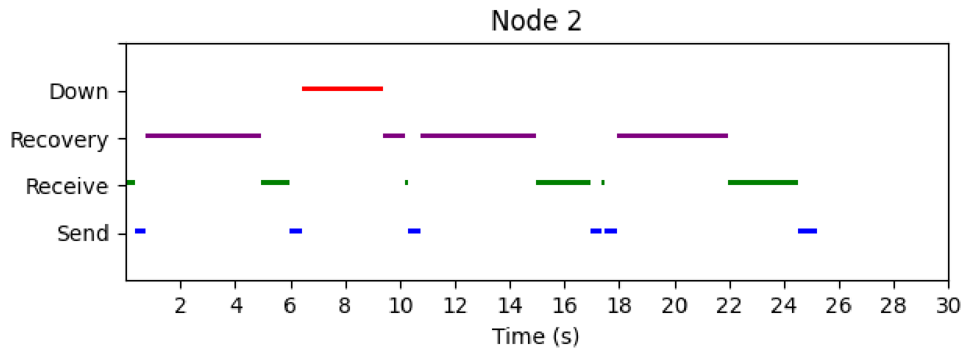


Figure 9: **State of the system during failures and recoveries - Node 0, Node 1**



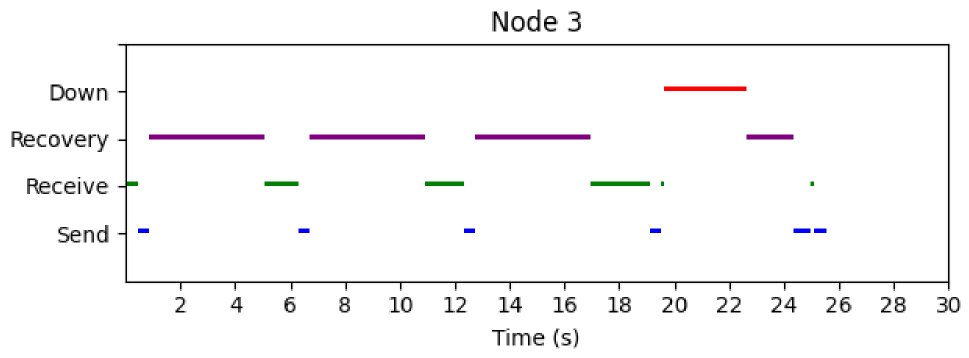


Figure 10: **State of the system during failures and recoveries - Node 2, Node 3**