Lab Report: Distributed Sorting Application

Team 2

Abstract

Abstract: As part of our lab assignment, we have implemented a working prototype that can sort data on multiple machines without assuming a shared storage service. The application sorts records generated by gensort record generator. First, the output stream for writing records is initialized and the key span size for each node is determined. Each node reads its own input file and key of each record is divided by the key span size which becomes the index to the record. If the key belongs to a different node, the record is sent to the node the key belongs to else the node is saved to the temporary file on the current node. The same occurs for records received from other nodes. Once all sockets are closed, the temporary file that contains all the records is used for sorting and then the temporary file is deleted.

1 Introduction

Processing and organizing large amounts of data has become very important in recent time. As the number of users and internet connections is rising, so does the amount of data that applications have to process. The rise of distributed ecosystems is introducing new challenges for data processing, but also new possibilities. Facebook, Google, Microsoft and others are coming up with more and more efficient distributed applications that are designed for data processing and sorting. That is why this problem is very interesting for computer scientists and engineers who need to implement the system that not only performs well, but also is fault tolerant, scalable and generally has all the attributes of a modern distributed application.

In this report we are presenting a distributed sorting application that processes records from the sortbenchmark.org. Our application is designed for the Indy sort benchmark category which means it sorts 100 byte records with 10 byte keys. The system is consisted of 3 parts: processing initial data, sending and receiving data between processing nodes, and sorting and merging.

In the following chapters we will describe our system architecture and explain design choice we have made, and after that we will mention how does our system handle concerns like fault tolerance and scalability. In the end some experimental results and analysis will be mentioned.

2 Background on Application

As it is mentioned in the introduction, this distributed sorting application will sort records as generated by `gensort` record generator. The keys are 10 bytes long, and each key is followed by 90 bytes of data. This application should not assume any common storage, except for reading initial files and writing to the output (if ran on a single machine).

2.1 Consistency

When sending and reading from sockets there are multiple threads ran on each node, therefore racing conditions might occur. In order to prevent this synchronization is applied to the block of code that writes received data to the file.

This is the only code in the application, that performs a mutate operation of the same entity from multiple threads. Therefore the application is thread-safe and inconsistencies cannot occur except due to the corruption of data passed through the socket, which is handled by the network transport layer and is not in scope for this project.

One important detail of the implementation: master node fills the array of worker node addresses not sequentially, but based on the node index, received from the node itself. Therefore the correct order of output files is known at start time and does not depend on the order in which worker nodes were able to connect to the master node.

2.2 Scalability

Application does not impose any restrictions on the number of worker nodes, so the limitation for scalability is the number of sockets and threads that the host hardware and software are able to maintain.

2.3 Fault tolerance

Recovering from a lost node was not implemented because it would require a significant complication of node selection mechanics, which right now is implemented as NODES[KEY / SPAN_SIZE]. However, it could be added with use of master node which would recalculate array of nodes are resend it to all the nodes. Of course nodes would have to maintain some kind of buffering mechanics to be able to rollback to the a state before losing one of workers. However, this particular project was designed in a way that nodes have their own input files, therefore losing a node would likely mean losing the file as well, so the suggested approach would be to fix the issue and restart the job.

3 System Design

3.1 General architecture

Application is organized as a pair of classes: a Master and a Worker.

Master is a class, which, when launched, listens for N (command line argument) workers to connect to it and receives IP addresses and ports from them. Upon receiving all of the data, it sends the list of those addresses to all the connected workers. At this stage the role of the master is completed and it exits.

Worker is an abstract class, that performs described interaction with the master, and after receiving the list of addresses, opens a socket and spawns a thread for every other worker. Before spawning threads, after spawning threads, and after completion of multi-threaded operations it performs abstract actions preProcess, process and postProcess on the main thread. Also, in each non-main thread it passes the corresponding socket to abstract function listenForRecords.

Sorting application is a subclass of Worker, that implements abstract methods for performing distributed sorting, with details given below.

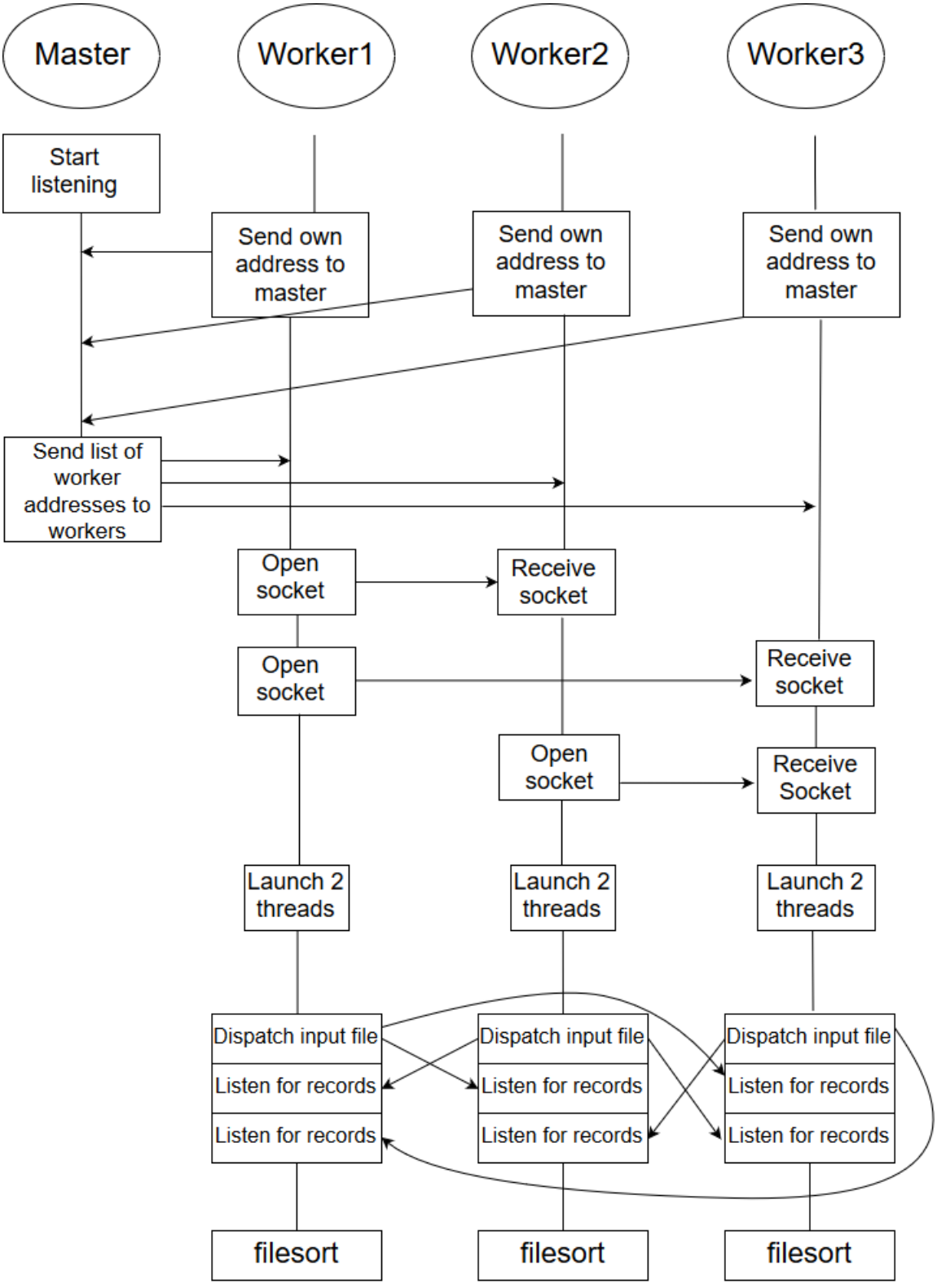

Figure 1: General system design example for three sorting nodes.

3.2 Opening sockets

Node with index X opens a socket for every node with index greater and X and accepts sockets from nodes with indices lower than X. This way all the sockets are opened in a predictable order without any locks without spawning the threads earlier than necessary.

3.3 Streams

All the sockets are opened with custom DuplexSocket wrapper, which provides both input stream and output stream immediately. These two streams are created in reversed order depending on whether the process creates or accepts the connection to avoid blocking.

3.4 Data processing

Each node has its own input file generated by running the `gensort` record generator.

3.4.1 Pre-processing stage

In the pre-processing stage, the output stream for writing records to a temporary file is initialized, and the key span size for each node is determined. Since the keys have 10 bytes, there is 2^{80} possible values. Therefore by computing $2^{80}/N$ where N is the number of processing nodes, we obtain the range of each node.

3.4.2 Processing stage

In the processing stage, each node reads its own input file. The key of each record is divided by the key span size and the resulting integer is the index of the node that the record belongs to. If the key of the record belongs to a different node than the node that processed it, the record is sent to the socket of the node that the record belongs to. If the key is in range of the current node, the record is sent through the output stream and saved to the temporary file on the current node.

3.5 Listening for records

The records received from the other nodes are also sent to the output stream and saved to the same output file. As mentioned above, the code writing to the file is globally synchronized within all the threads of the node.

3.5.1 Post-processing stage

After each node closes all sockets, the temporary file with all records that belong to the current node is passed to the sorting algorithm by executing filesort-performing `executeSort` function. After the sorting function returns, the temporary file is deleted.

3.6 Sorting and output

Function `executeSort` implements external sort. It picks up the file name of the file received from socket as an input.

Sorting has three phases:

- Reading parts of the file into memory
- Sorting data in memory

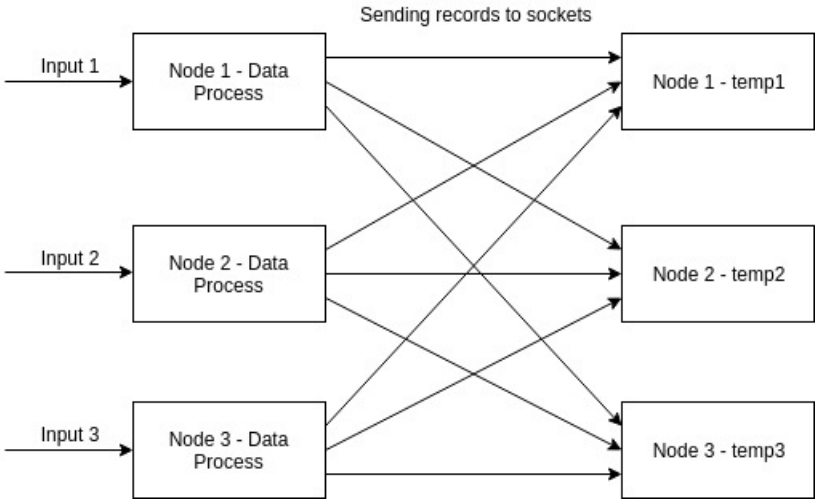

Figure 2: Data processing stage

- Writing sorted data to temporary file
- Merge of temporary files

To minimize the memory consumption implementation is storing key and (input) file offset for this record's data and not the data itself. After reader meets the memory threshold it passes the in-memory data to the sorting algorithm. Sorted data are stored in the temporary file and process is repeated till end of file is reached.

Merging of the temporary files is achieved by opening special Binary File Buffer for each file. Buffer keeps last read record in the memory. This way the program is iterating over each buffer and searches for key with lowest value. The key is "popped" out of the buffer and written to the output file. The process is repeated after till all buffers reach end of file.

4 Experimental Results

During the time of writing this paper DAS4 cluster had issues which made fail very often. Longest failure was four days. This was caused by the incapability of DAS4 to hold so much students at the same time.

For Surfsara cluster we were only able to get account for Hadoop/Spark cluster without availability to create custom distributed jobs e.g. without using any of the mentioned frameworks.

After many benchmarks on our personal machines we decided to postpone experiment till cluster is fixed. Local testing caused problems with scalability the more nodes we added the slower our application was. This included slow computation due to the overuse of single disk drive at the same time. This way we were able to sort using just one node much faster even if we used four nodes. Also we needed four times the size of input file space on drive. This might be easy to solve the problem by using HDD instead of NVME SSD or classic SSD but this may have caused even bigger overhead by IO operations.

5 Service Metrics of the Experiments

As mentioned previously, we were unable to produce beneficial experimental results.

6 Conclusion

We were able to successfully produce satisfying and sorted output and kept the architecture of a distributed system. Although, we were unable to measure meaningful execution time on the cluster, we managed to sort files locally in dispersed manner by simulating nodes.

7 Appendix A: Time sheets

Link to the repo: https://github.com/cabrenn/distributed_sorting

A. the total-time = 60h x 5 resources
B.the think-time = 8h x 5 resources
C. the dev-time = 20h x 5 resources
D. the xp-time = 6h x 5 resources
E. the analysis-time = 6h x 5 resources
F. the write-time = 8h x 5 resources
G. the wasted-time = 12h x 5 resources