Orca End-to-end Evaluations

Parham Yassini

Last Update: May 22 2020

We evaluate Orca in our datacenter testbed to show how much our method contributes to reducing end-to-end multicast transfer time and application execution time in a realistic multicast application.

I. Testbed Setup

In our setup, we have 1 host as multicast sender and 4 hosts as receivers. The testbed topology for the end-to-end evaluations is shown in figure 1. The testbed includes servers of different generations with different CPU speeds and each host is equipped with a 10Gb Network Interface Card (NIC). For each ToR switch, Orca switch logic is synthesised and implemented on a NetFPGA-SUME board which supports four 10 Gbps connections.

For the experiments that use unicast for sending data via UDP (details in Section II), the same topology is used but the NetFPGA devices at ToR switches are programmed to work as conventional layer 2 network switches.

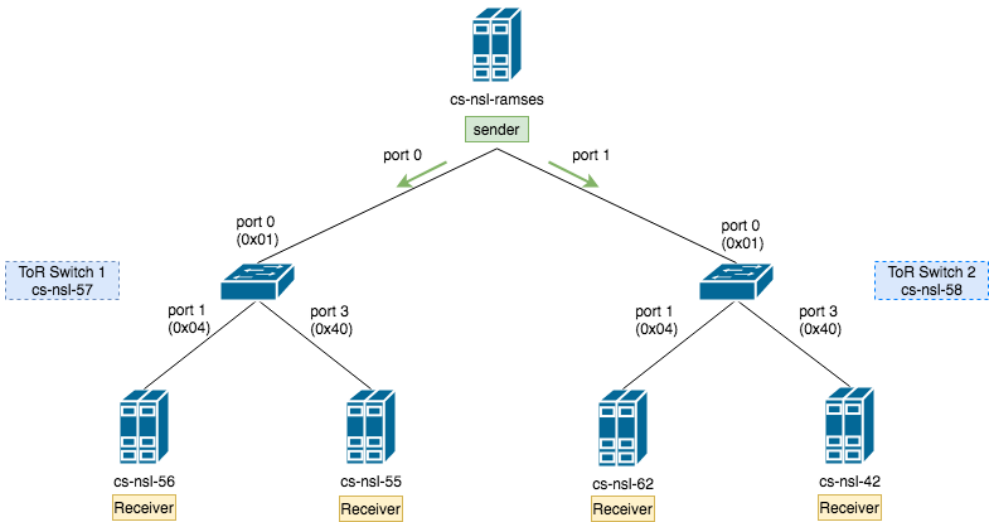

Figure 1: Testbed setup for end-to-end evaluation

In order to imitate a sender that is deployed on another rack, we use a double port NIC on the sender server (cs-nsl-ramses) and connect these ports to both of the ToR switches. With this setup, we simulate an aggregate switch that is forwarding the multicast data from another rack on the downstream path. Figure 2 illustrates an abstract view of the implemented testbed topology.

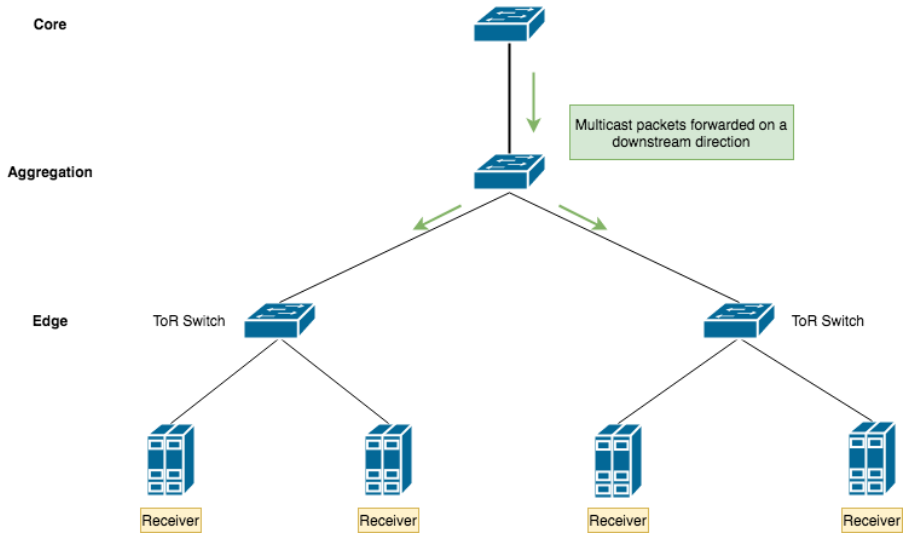

Figure 2: Abstract testbed topology for end-to-end evaluation

II. Choosing Multicast Application

Data multicast is a recurring pattern in many modern data center applications. In applications such as iterative machine learning and distributed database systems, files should be transmitted from one server to multiple receivers. In the database applications, a join query might require one table to be transmitted to multiple receivers. In each iteration of machine learning applications, receivers will receive and process the data and respond to the server with the results. In this context, many distributed data processing frameworks such as Apache Spark, have implemented multicast inside the application layer which relies on unicast in lower network layers.

Note: Proper references to be added.

In our experiments, we have implemented a simple file transmission application that performs a procedure similar to the real datacenter multicast applications. To evaluate how Orca can improve end-to-end latency in such applications, we run the same application using application layer multicast (based on UDP) and using Orca.

III. Application Implementation Details

The file transmission application consists of a server and a client application. It is written in Python 2.7 language with 218 and 170 lines of code for server and client scripts respectively. The server application is the one that transmits multicast data and aggregates the responses; while clients will receive process and acknowledge the server after processing each file.

In this section we describe the general execution model of the application and two different setups for sending the data over application-layer multicast and Orca.

A. Application Execution Model

1) Server Script: The server app reads each file in a loop and transmits buffers with size of 65507 MB over a socket in each iteration until the file is transmitted completely. The size of this buffer is determined by the maximum UDP datagram message size.

In each iteration, the server program waits for acknowledgement of all of the receivers inside another process. The next round of multicast will only start after the complete reception of the acknowledgements. This acknowledgement procedure emulates the aggregation job in distributed data processing frameworks such as Spark. In that case, this step is done by the master node after all nodes process the data and send back results.

2) Client Script: The receiver application uses receive calls on a socket in a loop for receiving data and writes the buffer data to the files.

Note: in our test application, in order to reduce the application side latency and avoid packet drops in the python application, we count the number of the bytes received instead of writing the data on the disk. Also, we simplified the process by making the clients aware of the anticipated file size in each iteration.

After complete reception of a file, the client acknowledges the server by sending a string with format ack_i where i is the iteration round number.

B. Implementation for Unicast

The server uses a dedicated UDP socket for each receiver host to send the data in this case. As an optimization option, the sending procedure can be done concurrently for groups of receivers or every single receiver. Therefore, the server is capable of sending in parallel using python multi processing feature and each process will send to a number of destination hosts in parallel.

In the receiver hosts, client application repeatedly calls *recv()* function on a UDP socket with address of the server to receive the file data.

Table I: Assigned Networks and IP addresses in Unicast Setup

Host Interface	Subnet	IP Address
cs-nsl-ramses (port 0)	192.168.1.0/24	192.168.1.1
cs-nsl-ramses (port 1)	192.168.0.0/24	192.168.0.1
cs-nsl-55 (port 0)	192.168.1.0/24	192.168.1.2
cs-nsl-56 (port 0)	192.168.1.0/24	192.168.1.3
cs-nsl-42 (port 0)	192.168.0.0/24	192.168.0.4
cs-nsl-62 (port 0)	192.168.0.0/24	192.168.0.2

As illustrated in Figure 1, we have two different destination networks that are managed by two switches. We manually assign the IP addresses for each host in the unicast setup.

C. Implementation For Orca

We designed the system such that minimal changes to the application is needed for switching between Orca and Unicast setup. Therefore, a very similar procedure is done by the application in Orca setup. The only difference is that application will not send data over dedicated UDP sockets and it sends the data over a Unix Domain Socket. This socket is our general purpose interface for communication with applications, any data written to this socket will be handled by Orca agent and will be transmitted to all of the receiver hosts.

IV. Orca Agent and Application Interface

As mentioned earlier, the application data is being sent to the Unix socket domain. The data flow through our modules is illustrated in figure 3.

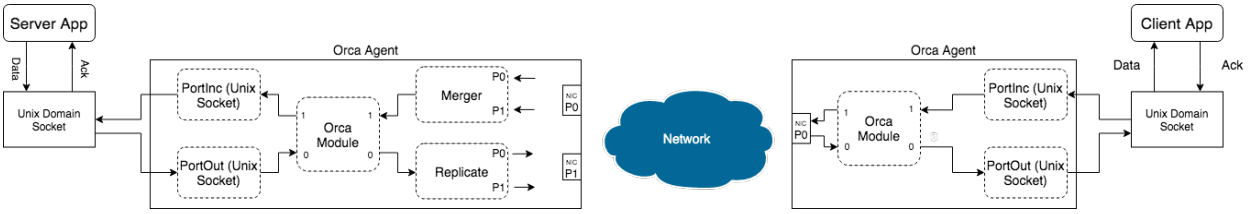

Figure 3: Orca Agent implementation, interfacing with applications

A. Unix Socket Port

This module works with two functionalities as an input or output for the agent.

1) ***Input Mechanism****:* The module receives data from the Unix socket domain and packetizes these raw data so that it can be sent over the network. The module accepts a template for the header that contains label placeholders and copies the header at the beginning of each packet data buffer. Also, it gets the maximum length of a packet as an input for packetizing the data. In our current setup this maximum length is set to 900 Bytes since we observed that larger packets caused pkt drops in the NetFPGAs. The packets are grouped together as a batch with size of maximum 32 packets.

Notes: The socket module implementation of BESS were designed to work with SOCKET_SEQPACKET which is a message based IPC mechanism. This forces the application side to send data buffers with limited size (e.g messages with maximum length of 900B) and increases the number of send calls which affects performance adversely. In our module, we use SOCKET_STREAM which provides a byte stream connection. With this design choice, the application does not need to know about the message sizes and sends the data as byte streams and in our socket we put these bytes into packets.

Maximum length of the received buffer in each socket system call is:

$$ReceiveSize_{max} = BatchSize \times PacketLength_{max} \tag{1}$$

Before this optimization, the module called the receive function per packet which made the IPC delay to be dominant all over the system. Using this optimization and based on the updated socket module in BESS, we could reduce the number of receive calls on the socket and gained better performance.

2) ***Output Mechanism****:* This socket module is also used as a PortOut module whenever data needs to be delivered to the application. When the file data arrives at the receivers or when the ACK messages arrive at the sender, this mechanism will deliver the data to the application layer. In this case, for each packet in the arrived batch, it will remove the labels from the packets and sends all of them (using one system call per batch) on the socket to be consumed by the app.

The implemented module, were designed to work in a best effort manner and did one non blocking *sendmmsg()* call for each batch. This would work fine if the application is fast enough to receive all of the packets with each call, but the python client script started to lose some of the packets in larger workloads.

In our socket module, we implemented a retry mechanism: The *sendmmsg()* function returns number of the successfully sent messages. If the number of sent messages were less than desired we retry up to 4 times for the remaining messages. The number of retries were obtained by a try and error.

B. Orca Module

This module is in charge for the attaching and removing labels necessary for Orca routing logic at implemented at the switches. It exposes a simple API for the control plane to update the multicast session states and adding labels to Orca agents. Figure 4 shows the label structure that is modified by this module upon packet arrivals.

The module handles two different packet flows:

1) Application Flow: Any packet coming on the port 0 of the module (Figure 3) is labeled with the *pilot label* before sending to NIC output. The pilot label contains the all of the forwarding instructions except the last ToR downstream which is unknown at the time to the sender host and it will be determined by the active agent in each rack. The pilot label is found based on a mapping between multicast session and the labels that are inserted by the control plane using the mentioned API. Upon arrival of the packet at the last ToR switch, the ToR switch will forward any packet with the missing *Rack label* to the active agent.

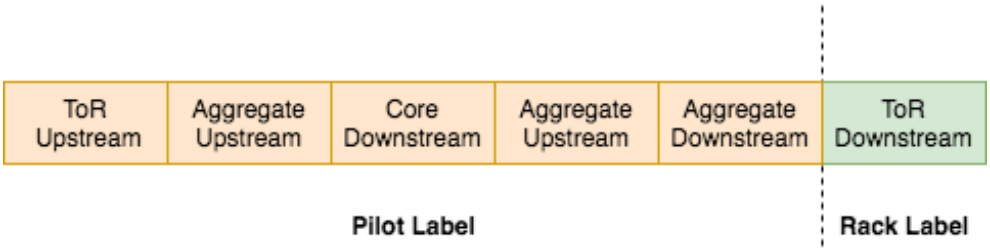

Figure 4: Packet label structure in Orca module

2) External Flow: Packets coming from the NIC will pass through the port 1 of the Orca module. It will check whether the packet contains a rack label and if it is already labeled it will pass the flow to the application pipeline (port 0 in Figure 3). The packets that do not contain a rack label, are sent from the ToR switch for labeling and therefore the Orca agent module will label them based on the most recent state of the labels mapping for the multicast session. After attaching the label, the packet will be forwarded to the NIC output on port 1. In cases that the active agent is one of the multicast receivers, Orca module will forward a copy of the packet to the application pipeline before sending it out on the line.

C. Replicate and Merger Modules

These are the pre-built modules we use for the sender side agent only. The replicate will forward the received packets on both output ports of the NIC for the two ToR switches.

The merger is used in the reverse path of the pipeline for receiving the ACK packets. The module will merge and forward the received packets from both NIC input queues.

V. Evaluation Results

A. Workloads

Size of the files to be transmitted and number of iterations were inspired by LDA (Latent Dirichlet allocation) machine learning application implemented in Spark. The multicast data in this case is the training vocabulary. The algorithm performs 33 rounds and in each round one file is sent to all of the computation nodes. The base workload consists of total 88MB data and we double the workload in different experiments to evaluate how well Orca and Unicast perform in case of workload scaling. For the largest workload, the total file size is 1.4GB with the largest transmitted file having a size of 141MB.

B. Application Execution Time

The application execution time shows the total time taken for the completion of the 33 rounds of file transfer by the application. For each workload, we ran the application using two methods 10 times and report the average execution time of the application of the sender application. Figure 5 shows the execution time for Orca and Unicast with various workloads. As illustrated, for the largest workload (1.4GB total file size) Orca achieves 4.16x improvement.

C. File Send Time

As mentioned earlier, each iteration is completed after all of the receiver hosts receive and process all of the file chunks. The file send time measures duration of each iteration from sending the first chunk of a file until receiving the last client acknowledgement at the server. Figure 6 shows the CDF of send time latencies for various workloads.

D. Client Wait Time

After complete reception of each file, the client host will send an ACK message and waits for the next file. Wait time is measured as the duration from when the client acknowledges the $file_i$ reception and the time that the $file_{i+1}$ is received. In the real world iterative applications, this time denotes the idle waiting

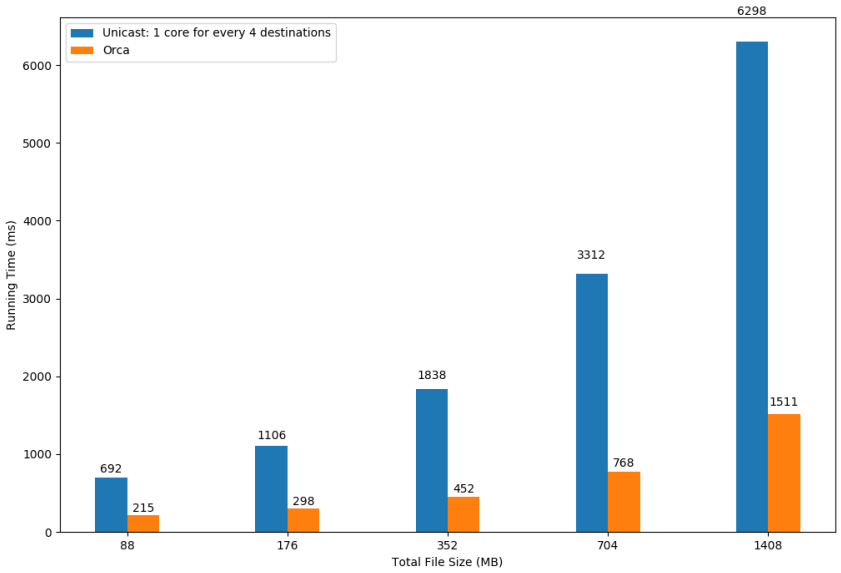

Figure 5: Total application execution time, varying the workload size

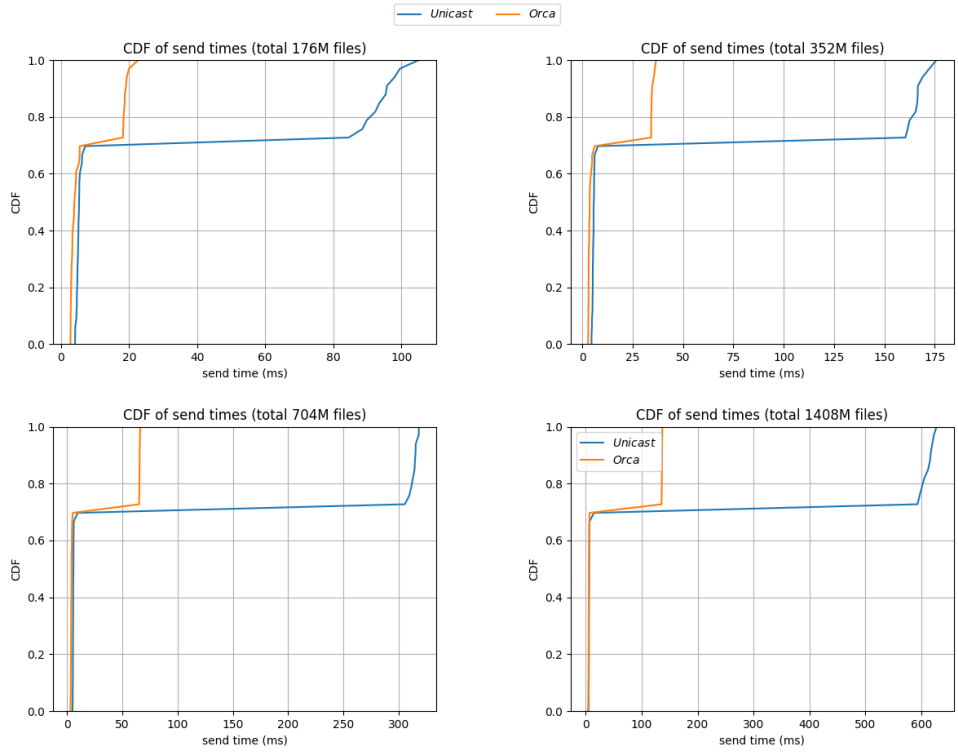

Figure 6: CDF of send times at the sender application with various workloads

time of the applications on the receiver hosts. Figure 7 shows the wait times for all of the iterations for one of the receiver hosts (cs-nsl-56) in the testbed.

Note: This metric describes the application level wait time on the receiver machines aside from the absolute transfer time of the files from the server's view.

Note: Discussion about the improvement in latency percentiles?

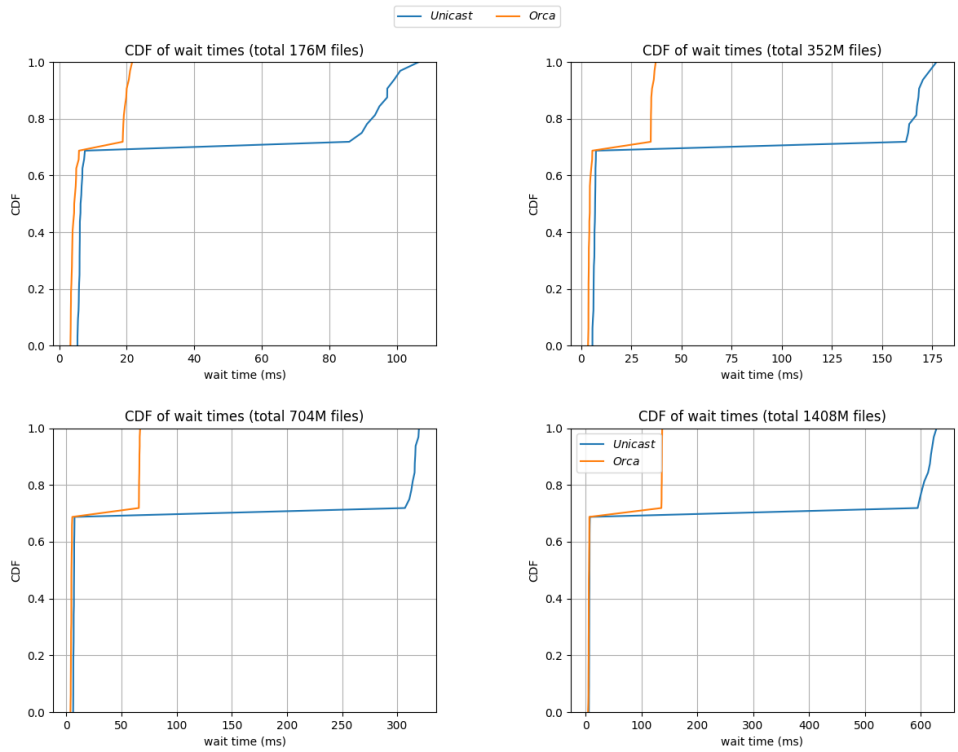

Figure 7: CDF of Wait times of the client application with various workloads

E. Optimizations for application-layer multicast

As an optimization for the application-layer multicast, the server can send each file to all of the receiver hosts concurrently using different processes on multiple available cores. We have implemented this procedure for unicast application to evaluate how Orca performs in case the sender application is taking advantage of multiple cores. The results for this experiment are shown in figure 8. Even in the case that application uses one core for each multicast destination, Orca improves the total application running time by 85%. However, it is not plausible in a real-world scenario to assign one core (at the

sender host) for each of the receivers where tens of receiver nodes are used for distributed computation of a task.

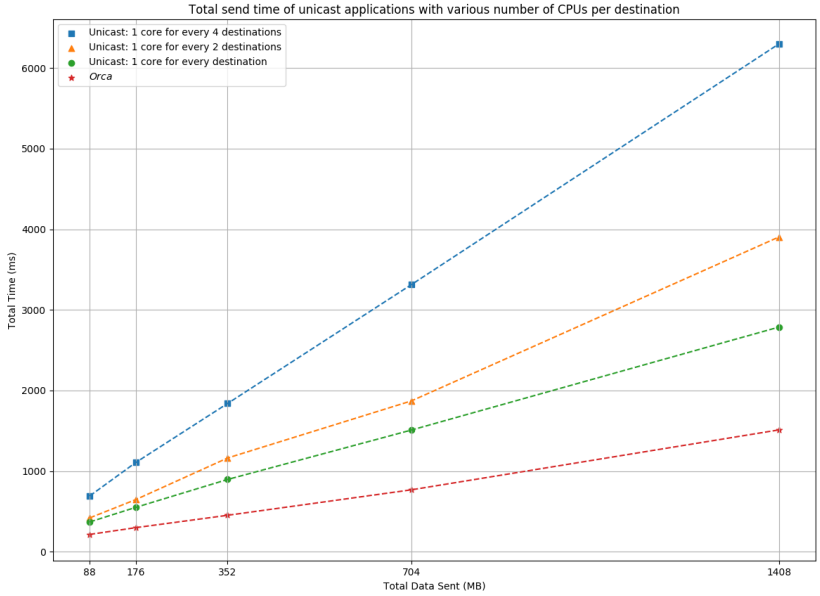

Figure 8: Total application execution time, varying the number of cores per destination for Unicast