ECE 428/CS 425 Distributed Systems **Navid Mokhlesi, navidm2**
Spring 2020 **Amber Sahdev, asahdev2**
MP 1
Cluster: g22
URL: https://gitlab.engr.illinois.edu/asahdev2/ece428
Revision Number: 3a9aac45d0b57d90fcadd7f0c3a1f04c3756eaef

For MP1 we extensively used Go's concurrency features such as Go routines and channels.

To build the code run the following within the MP1 directory of the repository

```
> make all
```

After running the makefile, we can run the logger as follows:

```
> ./mp1_node [number of nodes] [path to file for host List] [Local Node Number]
```

We stored our list of hosts on hosts.txt. And Local Node Number is Zero indexed

To pipe the event generator output into node:

```
> python3 -u gentx.py [hertz] | ./mp1_node [number of nodes] [path to file for host list] [Local Node Number]
```

We also need to install certain python dependencies to generate graphs as follows:

```
> pip3 install matplotlib
> pip3 install numpy
```

The python script to generate graphs can be run as follows

```
> python3 graph.py
```

For analyzing our distributed logging system, we plotted out two graphs that measure the

- Time delay between when the node generated the transaction, and the min and max time when the last node commits it.
- Bandwidth as the length of bank message struct (number of bytes) downloaded by a node per second

Graphs were generated by writing to a log file on every node and then combining these log files in graph.py to process and generate plots.

System Design:

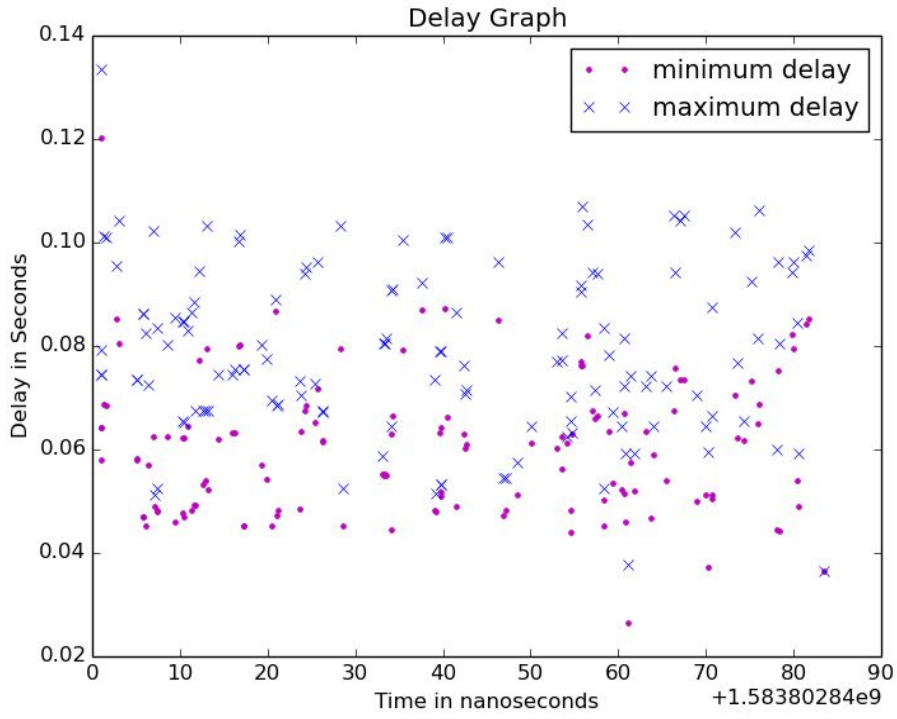
Our application is designed to support a reliable distributed Total Causal ordering of bank transactions occurring across N identical nodes where N in [1, 10]. We use a TCP R-Multicast, and well architected reception and handling code to ensure all communication occurs via FIFO ordered channels to produce fully causal communication. We ensure no redelivery of messages by including the original sender of an R-multicast, and the message number sent from that node, so that the same message is never delivered twice. We can tolerate up to but not including ½ of the nodes dying simultaneously.

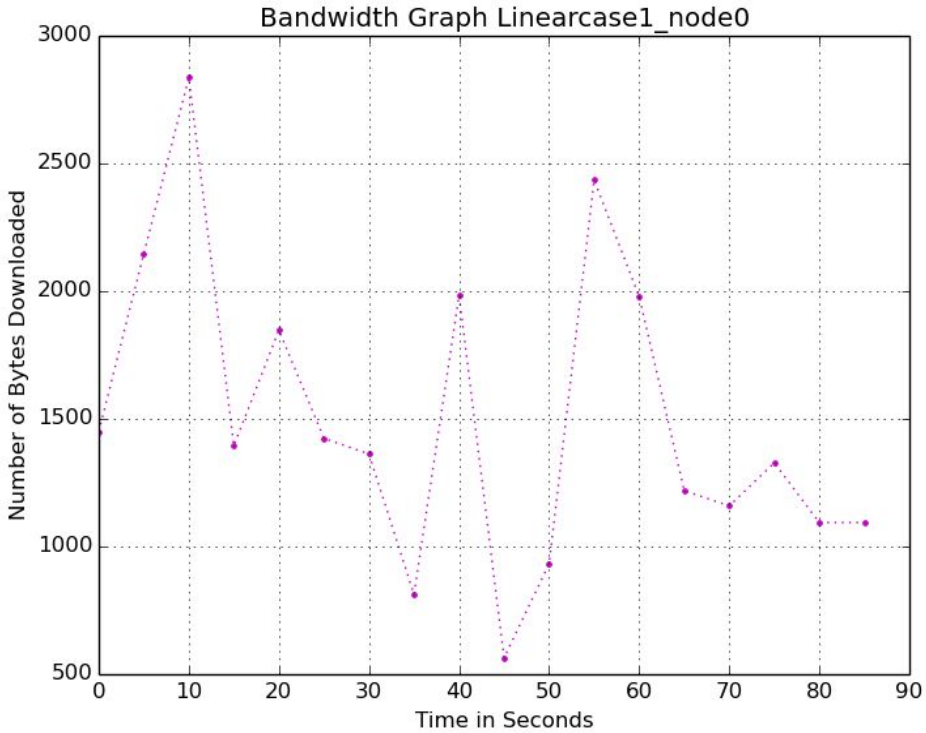
We ensure total ordering of events implemented on top of the Causal ordering through decentralized proposition and finalization of order via the ISIS algorithm presented in lecture.

Our design works by launching a thread to receive incoming connections locally that spawns an incoming thread handler for each incoming connection. These threads deliver messages to a thread safe queue for processing in Fifo order. We then attempt to connect to all the expected nodes in our system, and wait for any remaining nodes to connect to our node before we attempt to set up our outgoing connection to those nodes (each outgoing connection is handled in it's own thread). Once all connections are established with the number of expected nodes, we begin receiving local events from the event generator. For local events, we rebroadcast them to all the other nodes, asking for a proposed sequence number. When receiving a request for a proposed sequence number we provide the largest previously proposed sequence number + 1, or the max finalized sequence number + 1. When the originator of the event recollects these responses, it takes the max for its local event's proposals, and once it receives all remaining proposals for connected nodes, it communicates the final agreed sequence number for the event, and checks if any messages from the min-heap priority queue, keyed with priorities (proposed of finalized), has a finalized event at it's head, then we pop and deliver to the application. This ensures a total order. As for reliability, we also check to see if proposed events blocking the head of the queue belong to a disconnected node, and remove those events as to not halt because a node died. Additionally, we only check to see if connected nodes have submitted a proposal before sending out an agreed/final transaction number, (as determined by TCP error handling).

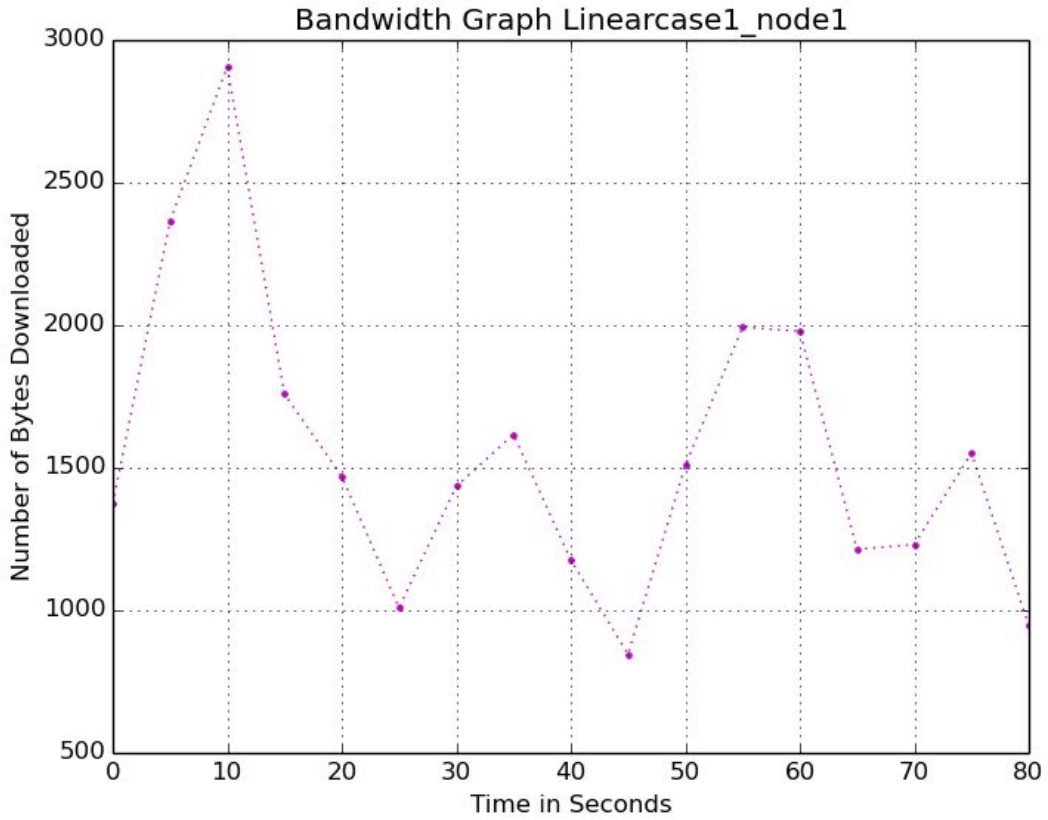
Graphs of the evaluation as described above:

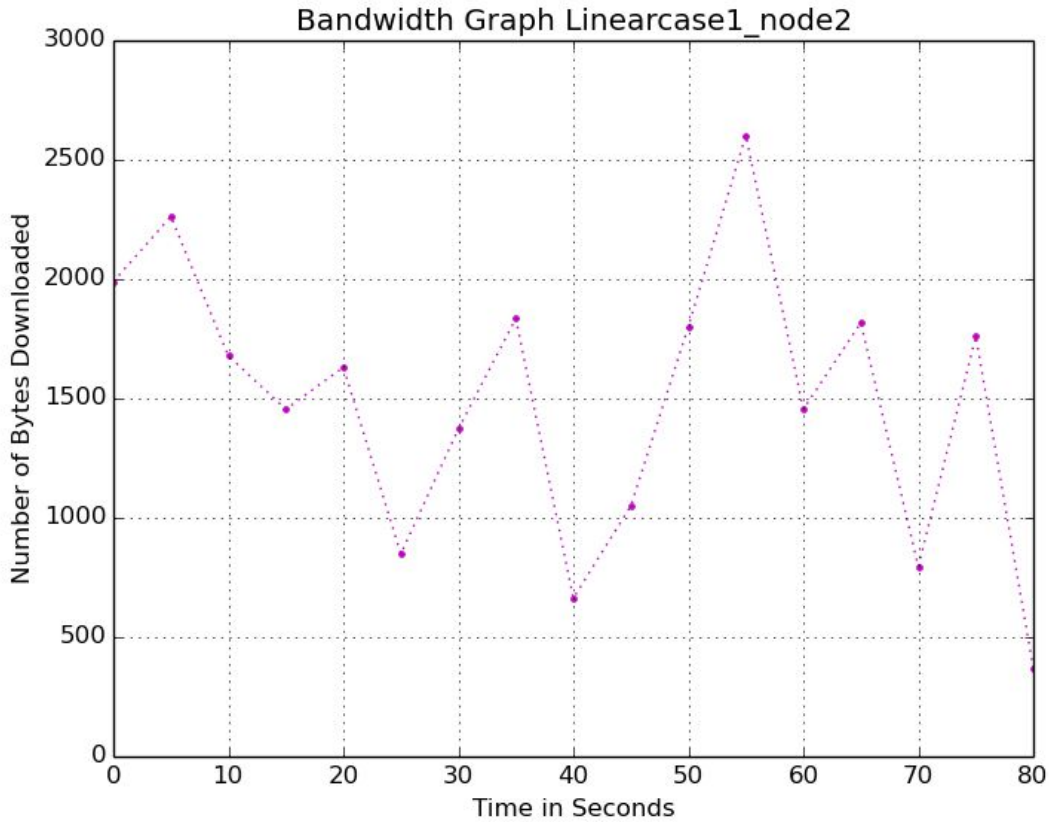
1. 3 nodes, 0.5 Hz each, running for 100 seconds





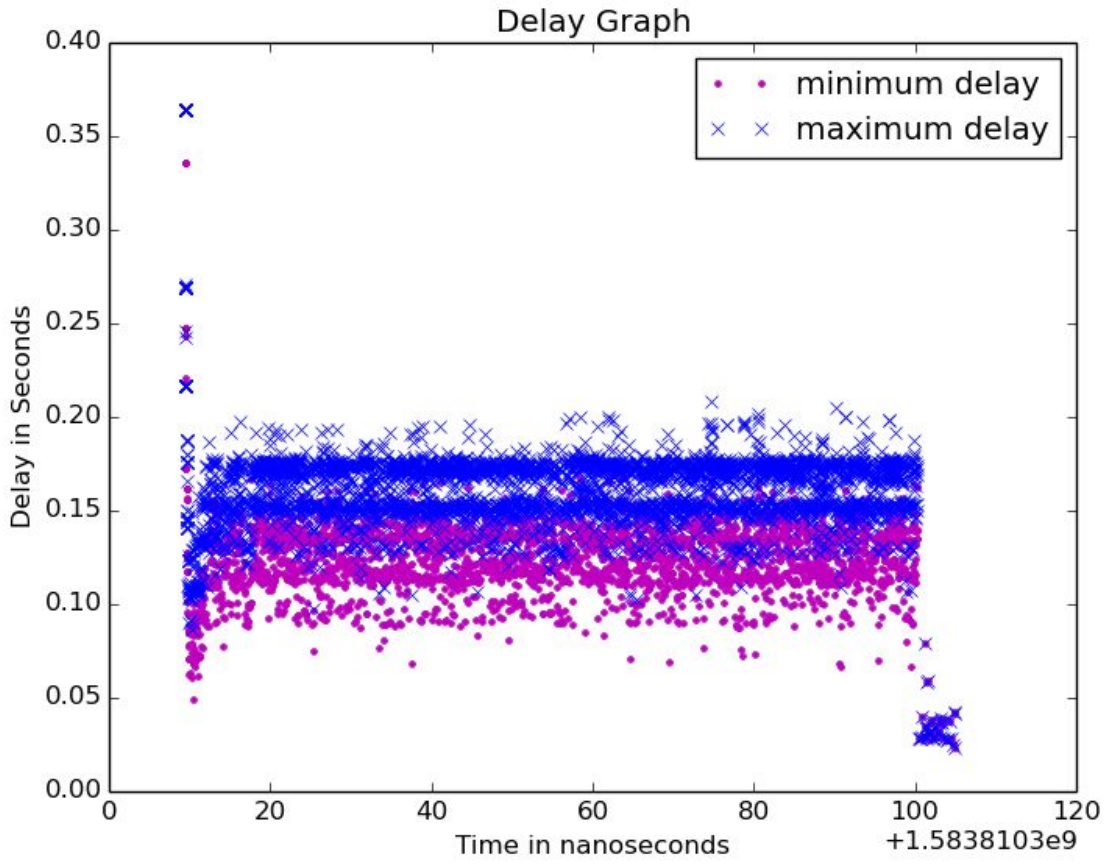
* y axis represents Number of Bytes Downloaded over 5 Seconds

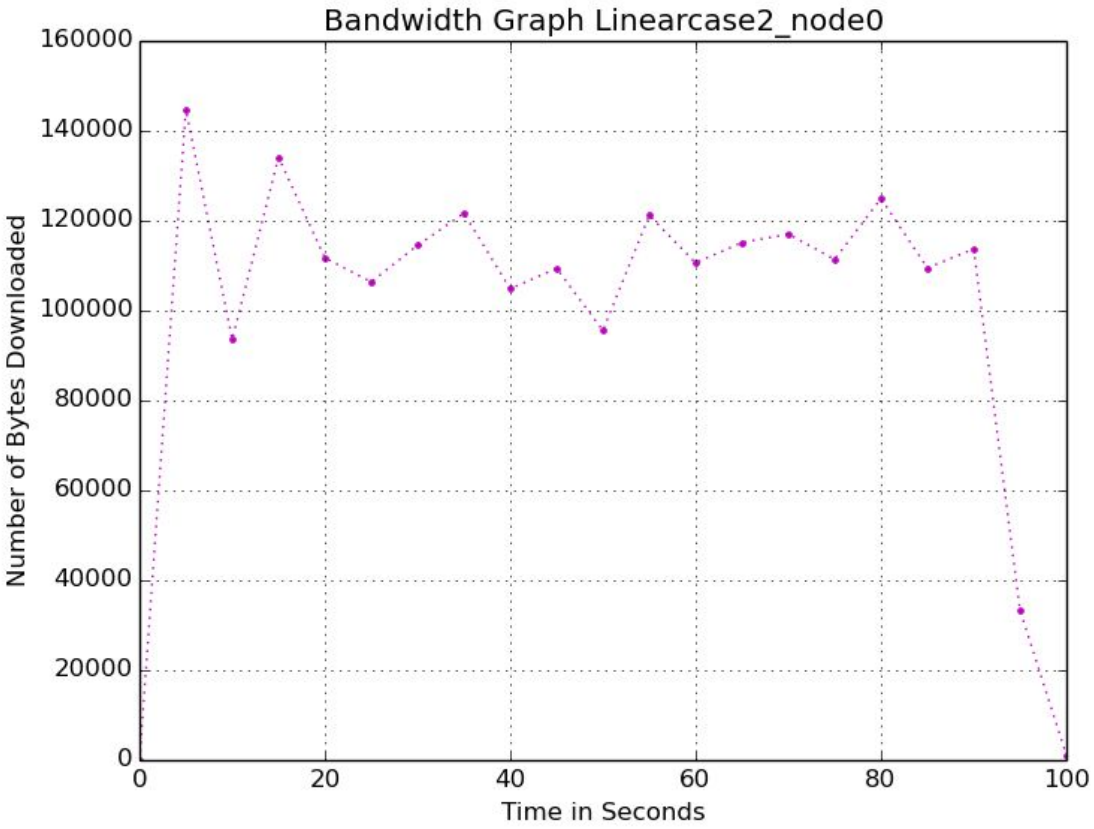




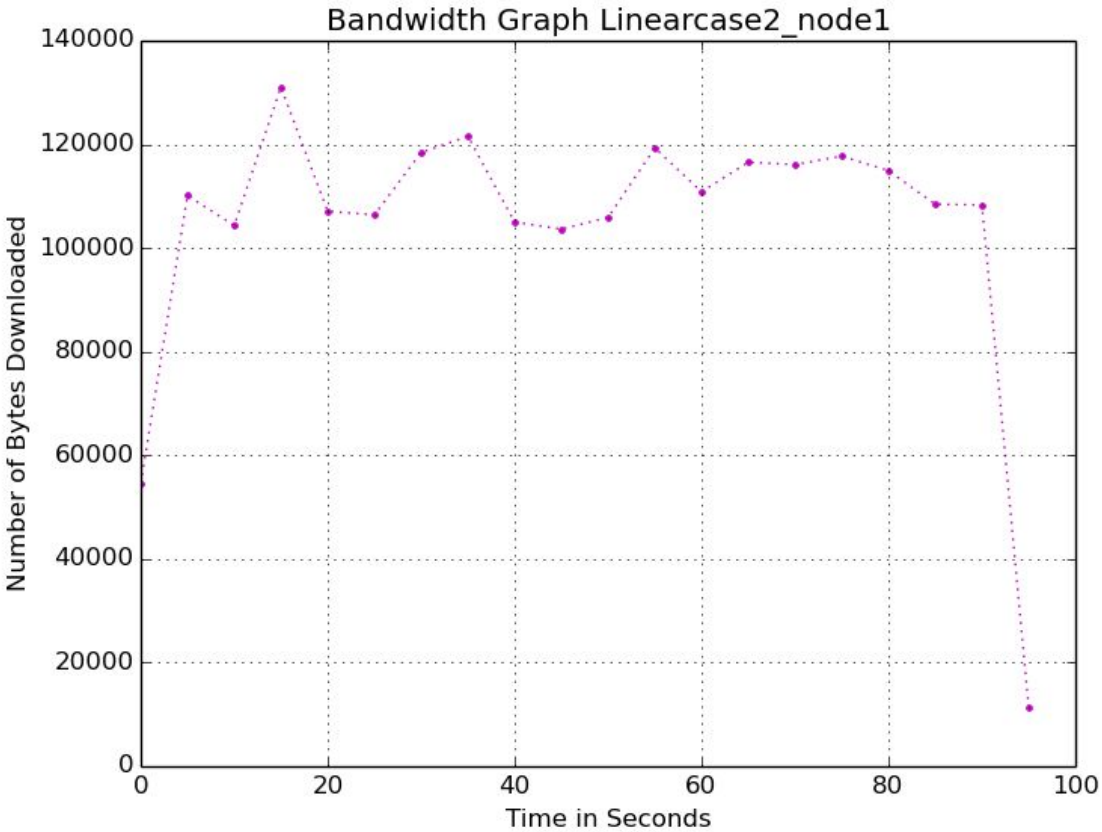
* y axis represents Number of Bytes Downloaded over 5 Seconds

2. 8nodes, 5 Hz each, running for 100 seconds

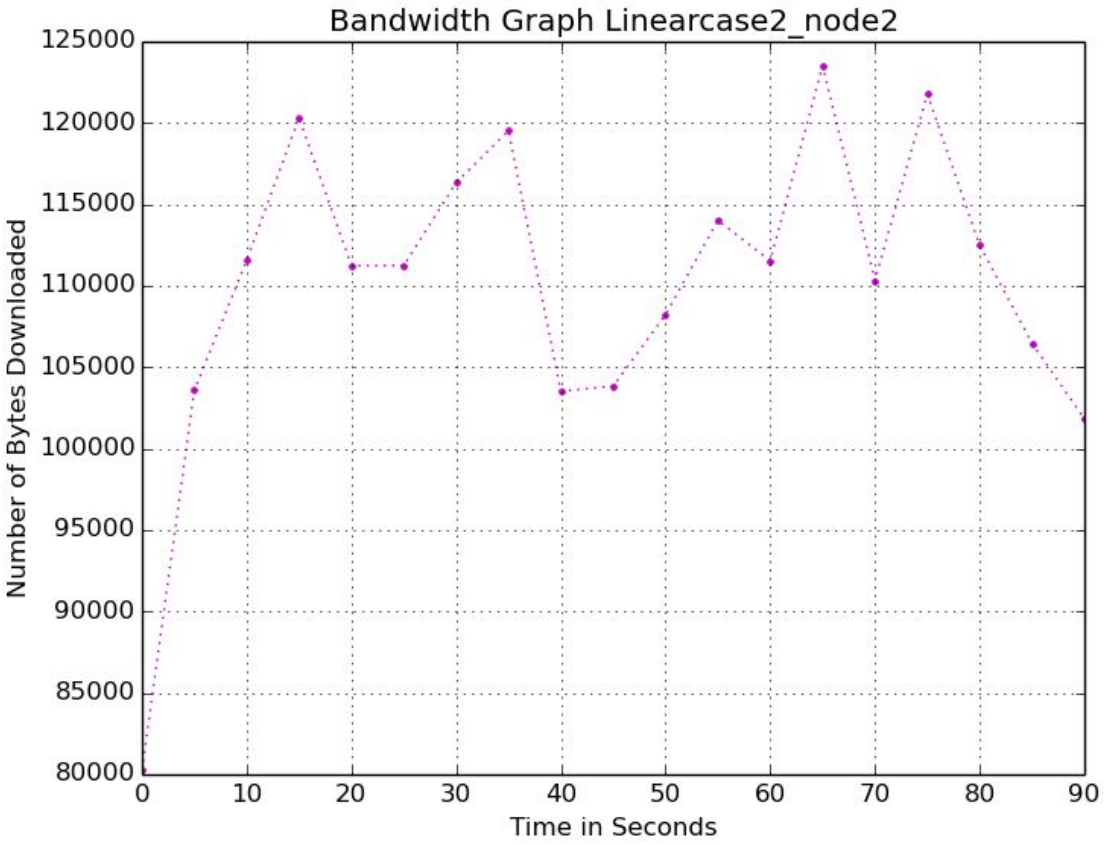


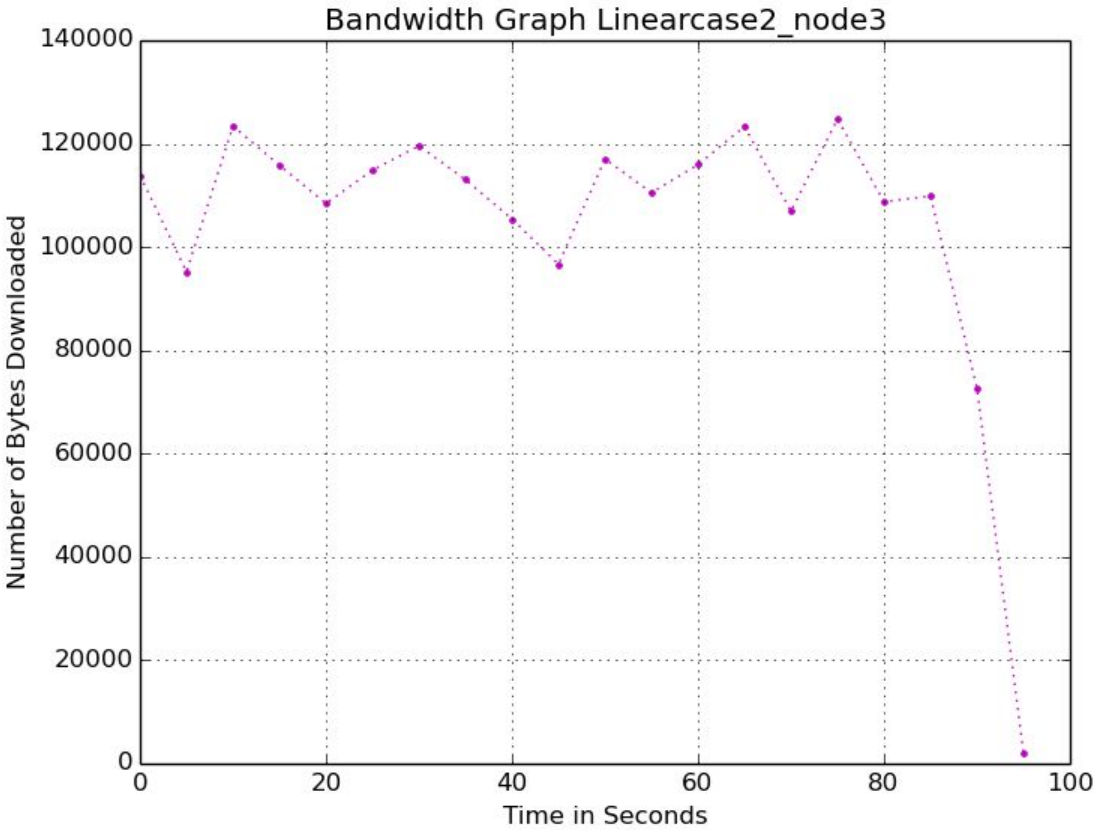


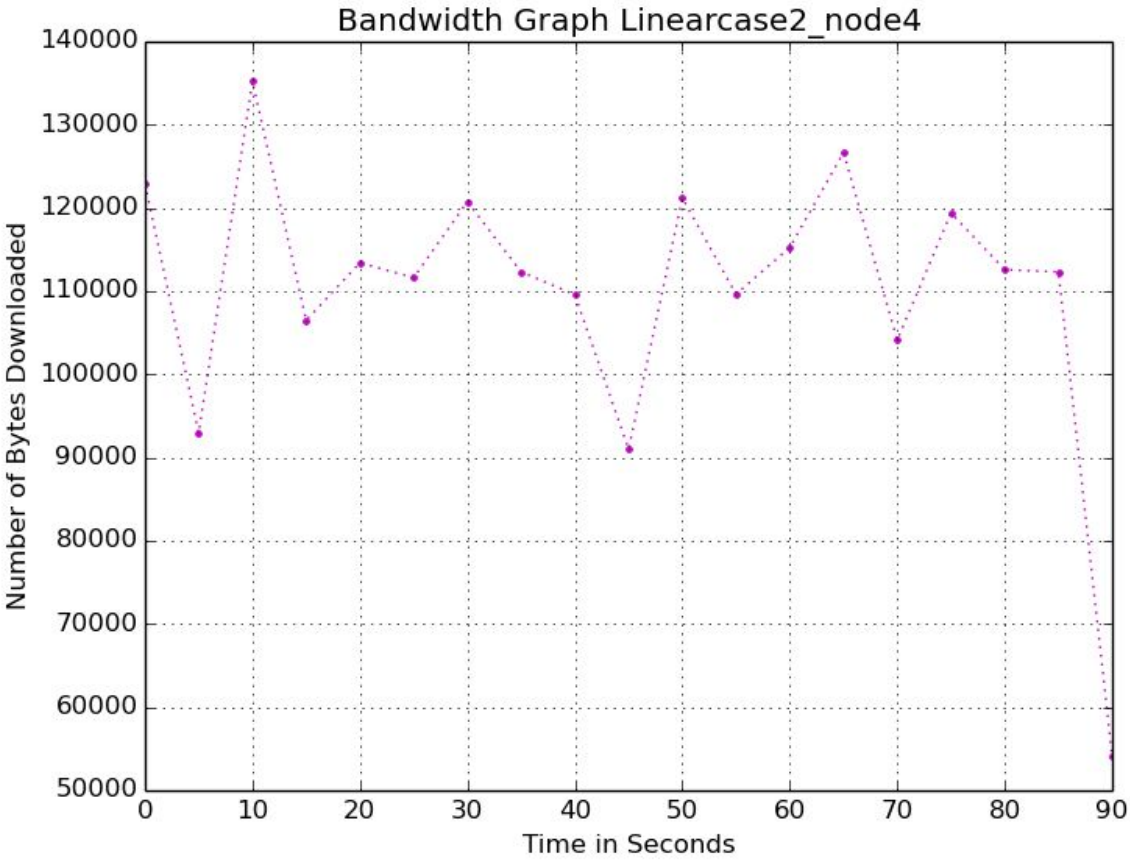
* y axis represents Number of Bytes Downloaded over 5 Seconds

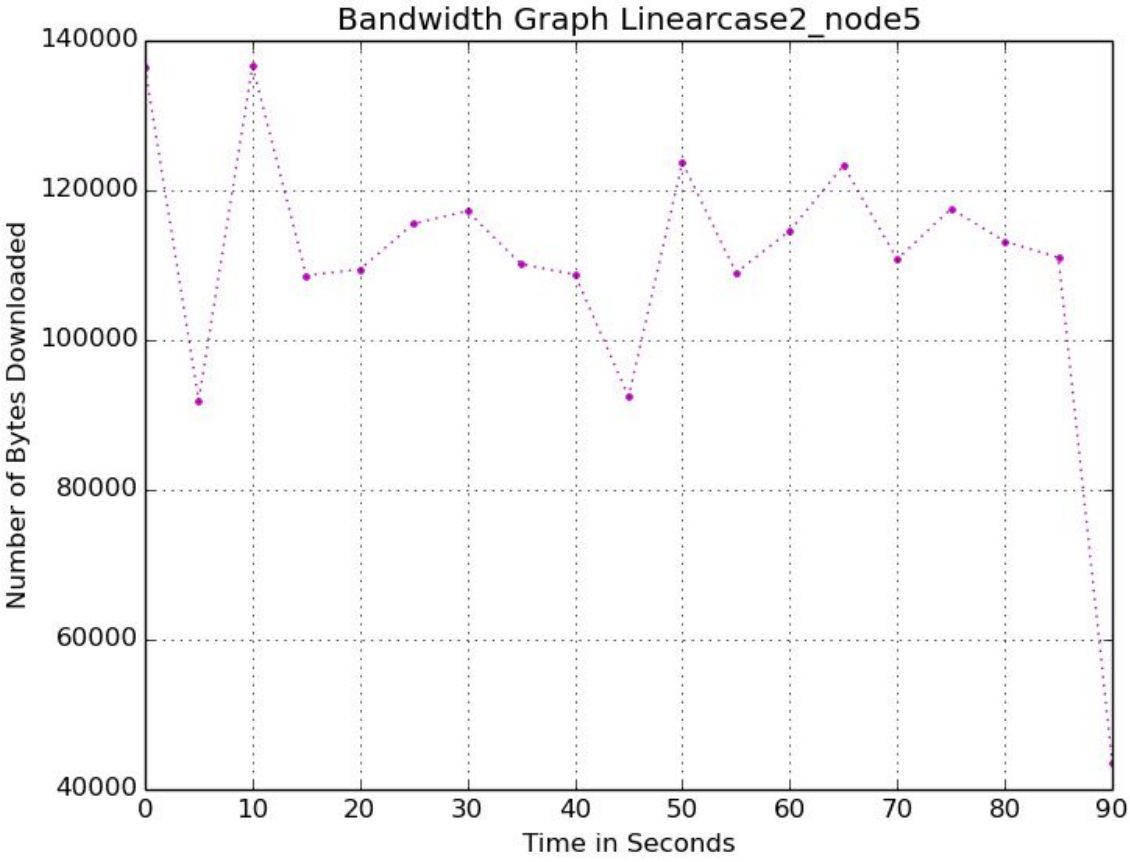


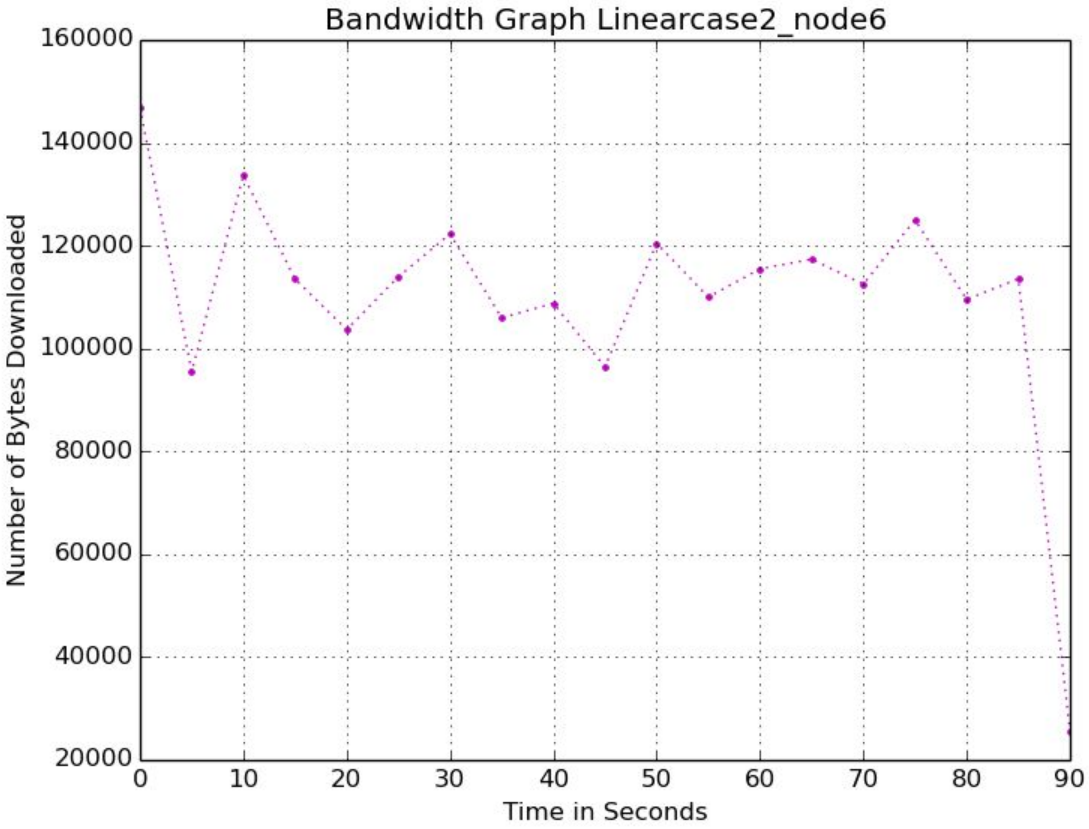
* y axis represents Number of Bytes Downloaded over 5 Seconds

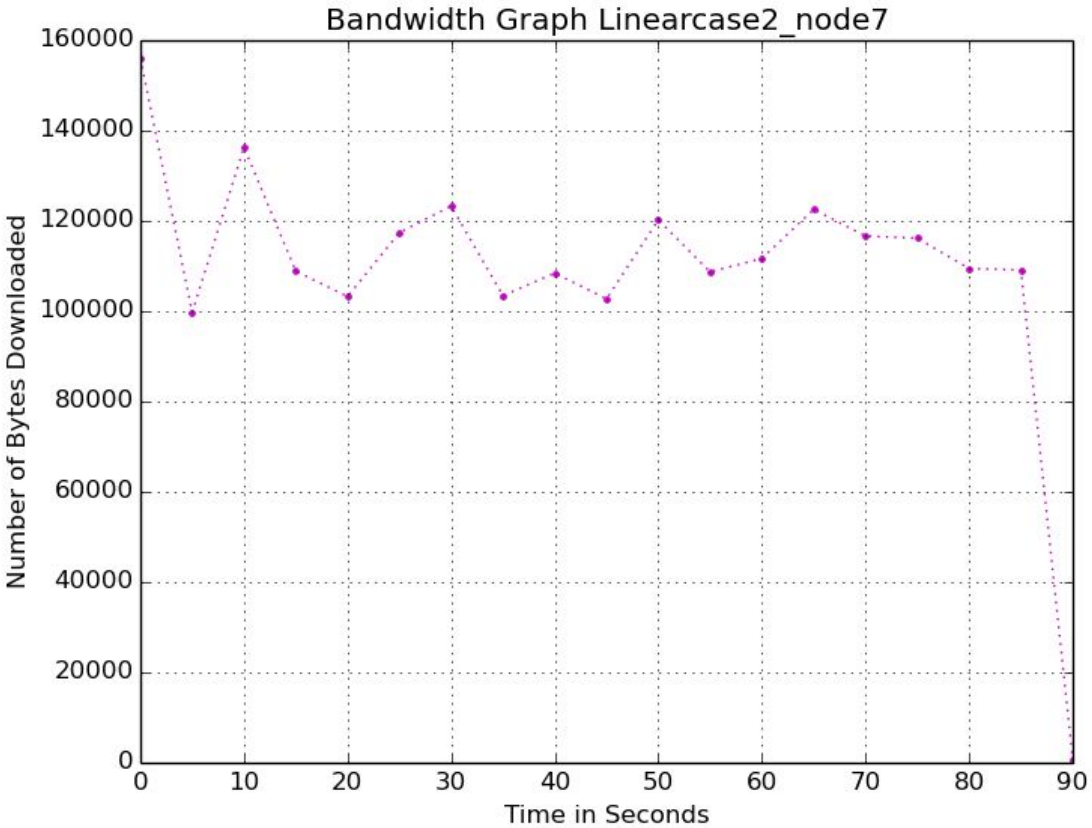





Bandwidth Graph Linearcase2_node4
Number of Bytes Downloaded
Time in Seconds
140000
130000
120000
110000
100000
90000
80000
70000
60000
50000
0
10
20
30
40
50
60
70
80
90

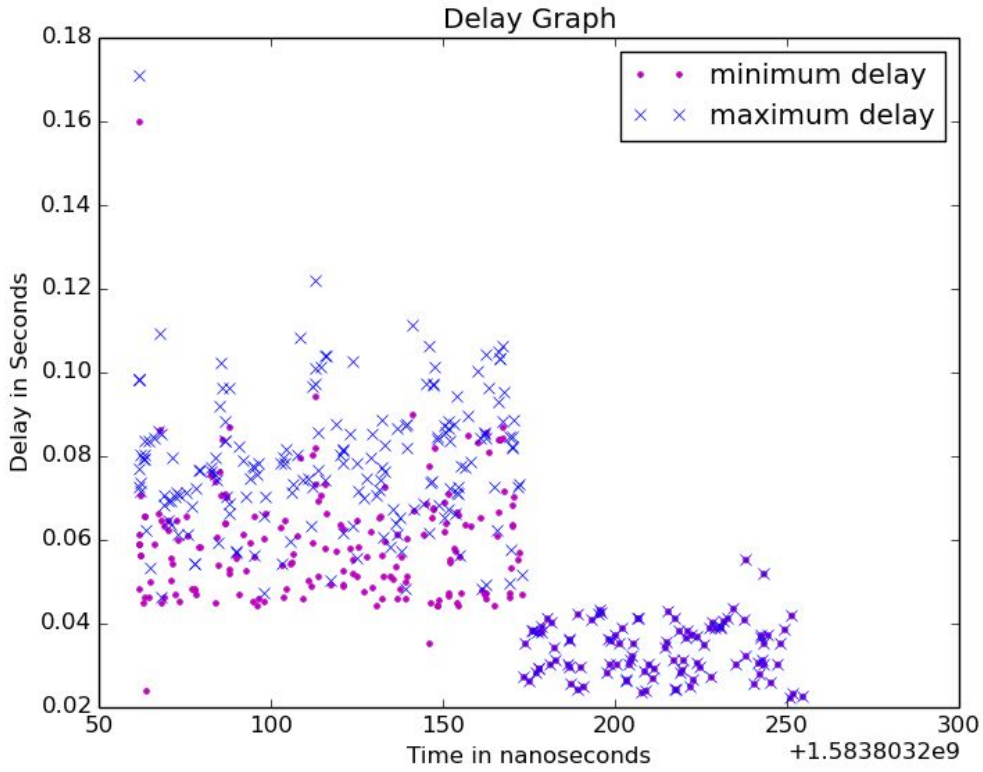

Bandwidth Graph Linearcase2_node5
Number of Bytes Downloaded
Time in Seconds
140000
120000
100000
80000
60000
40000
0
10
20
30
40
50
60
70
80
90

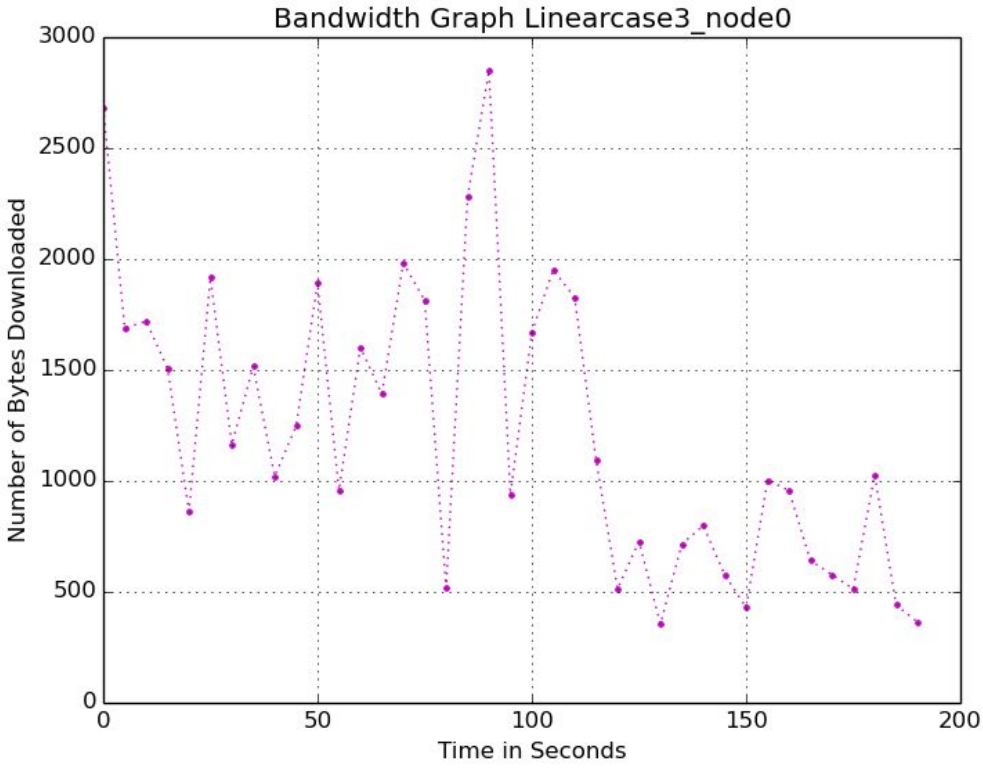


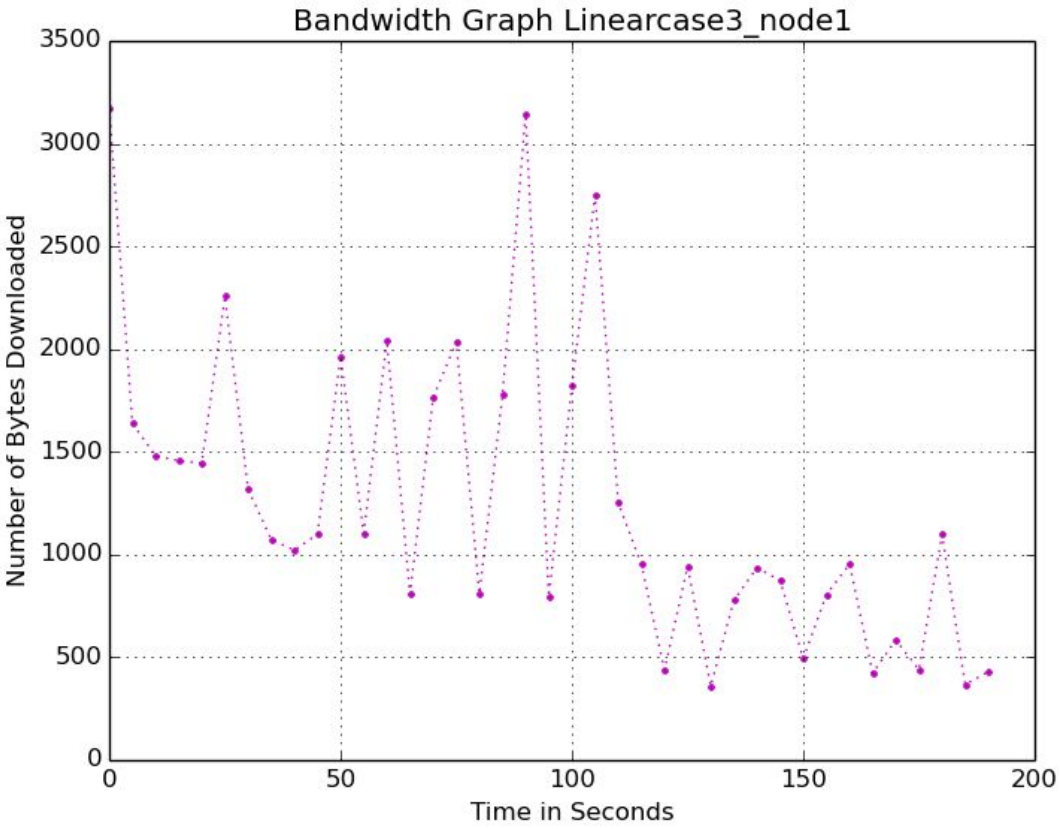


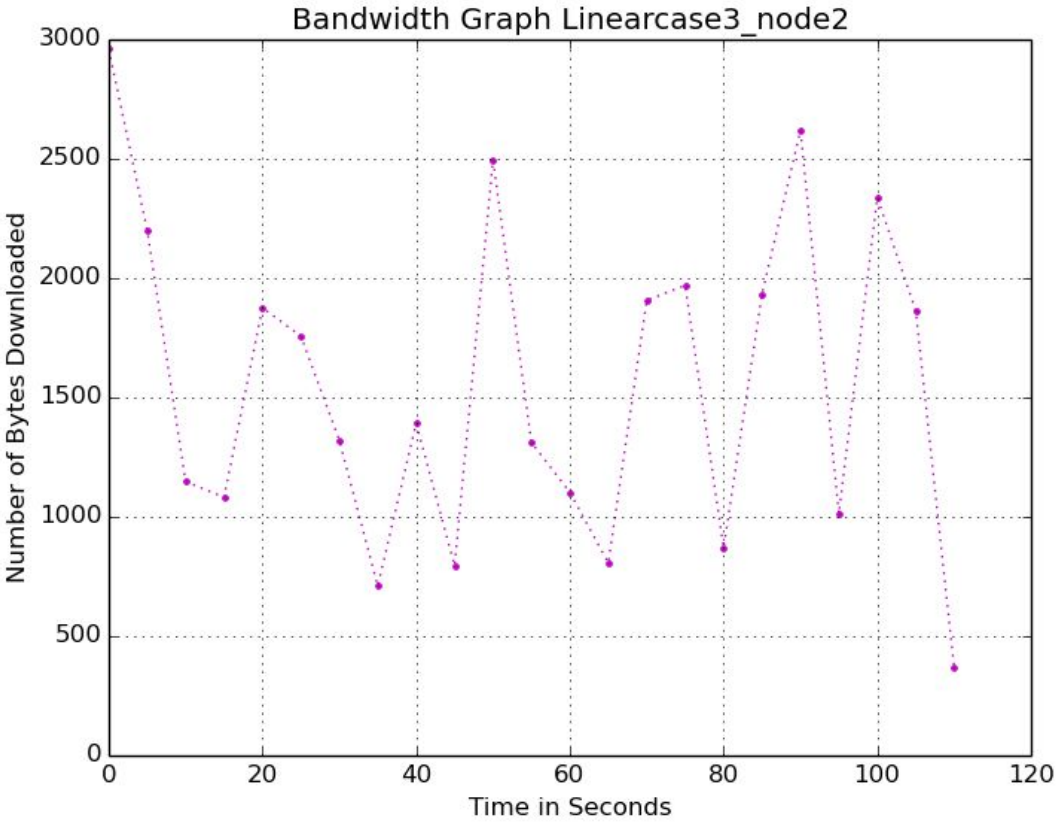
* y axis represents Number of Bytes Downloaded over 5 Seconds

2. **3 nodes, 0.5 Hz each, runing for 100 seconds, then one node fails, and the rest continue to run for 100 seconds**



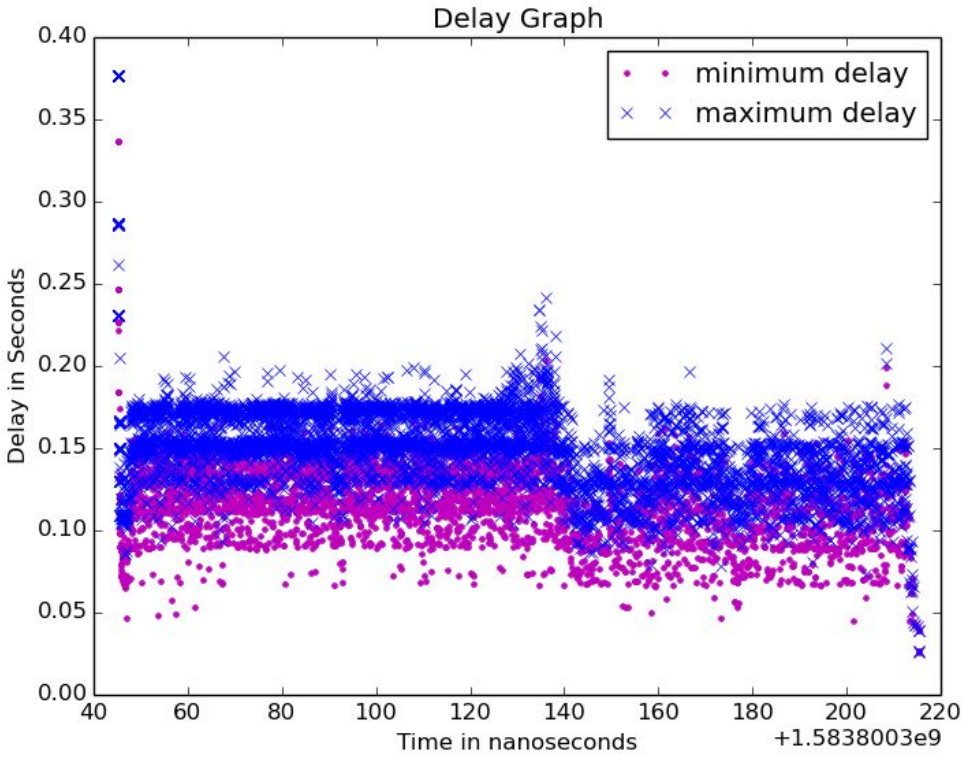


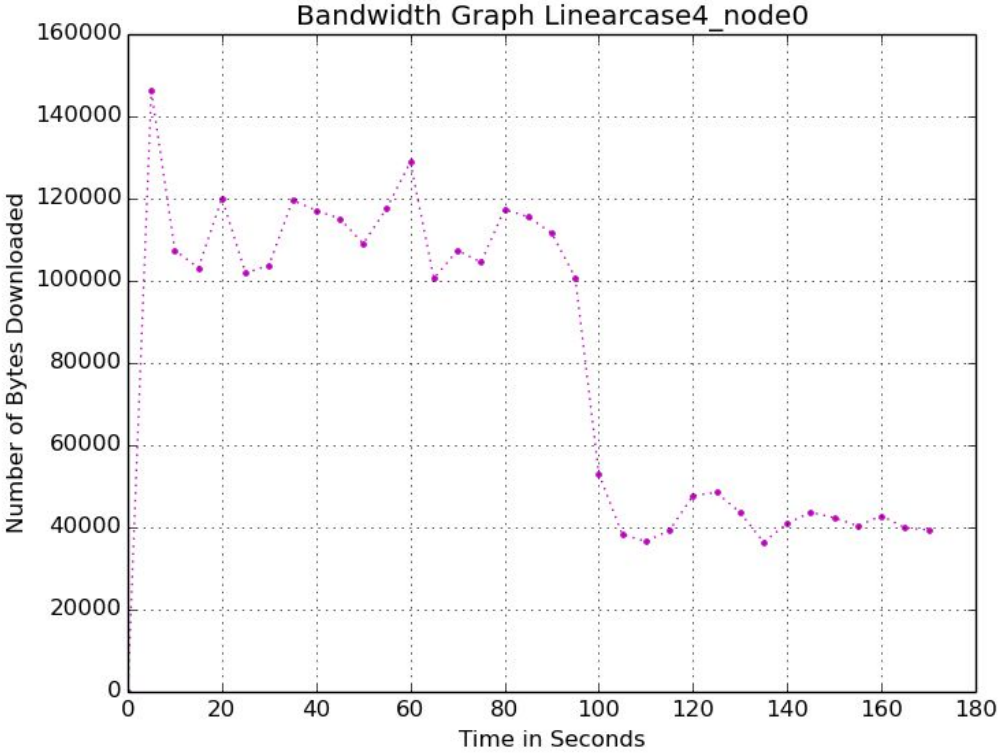


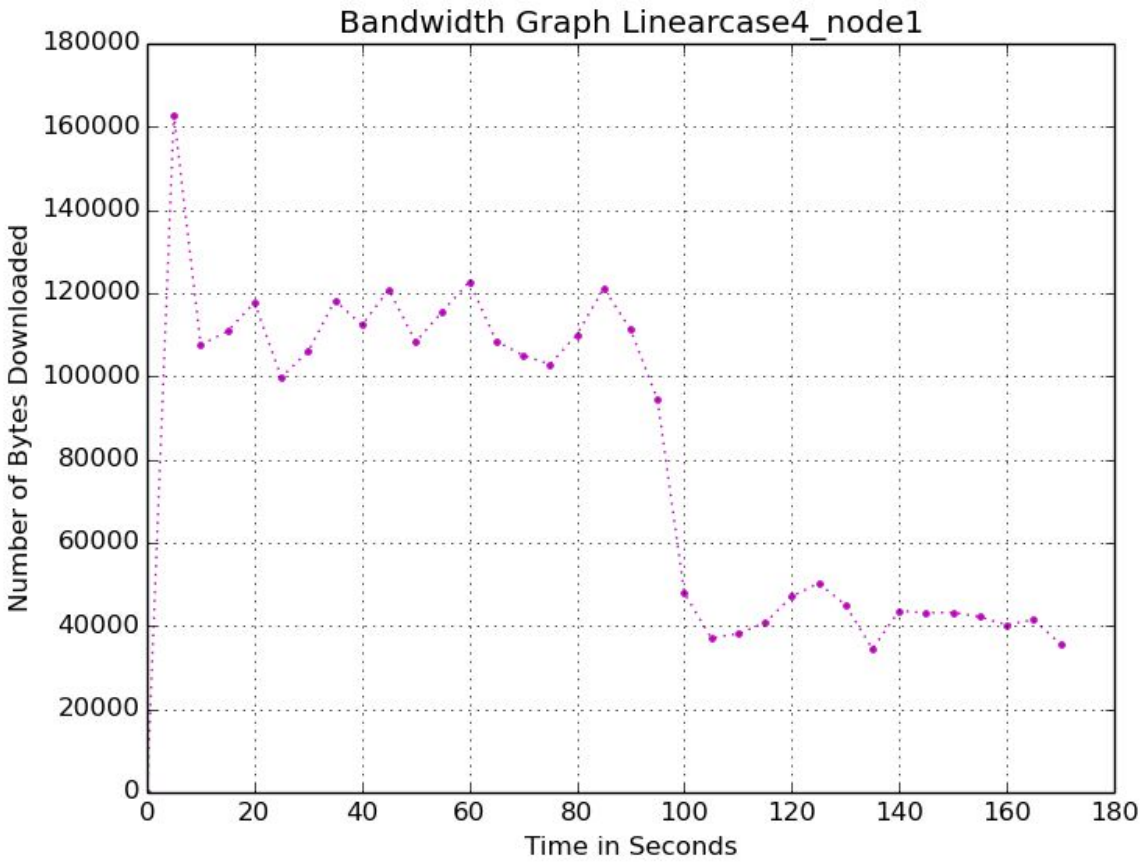


* y axis represents Number of Bytes Downloaded over 5 Seconds

3. **8 nodes, 5 Hz each, running for 100 seconds, then 3 nodes fail simultaneously, and the rest continue to run for 100 seconds.**





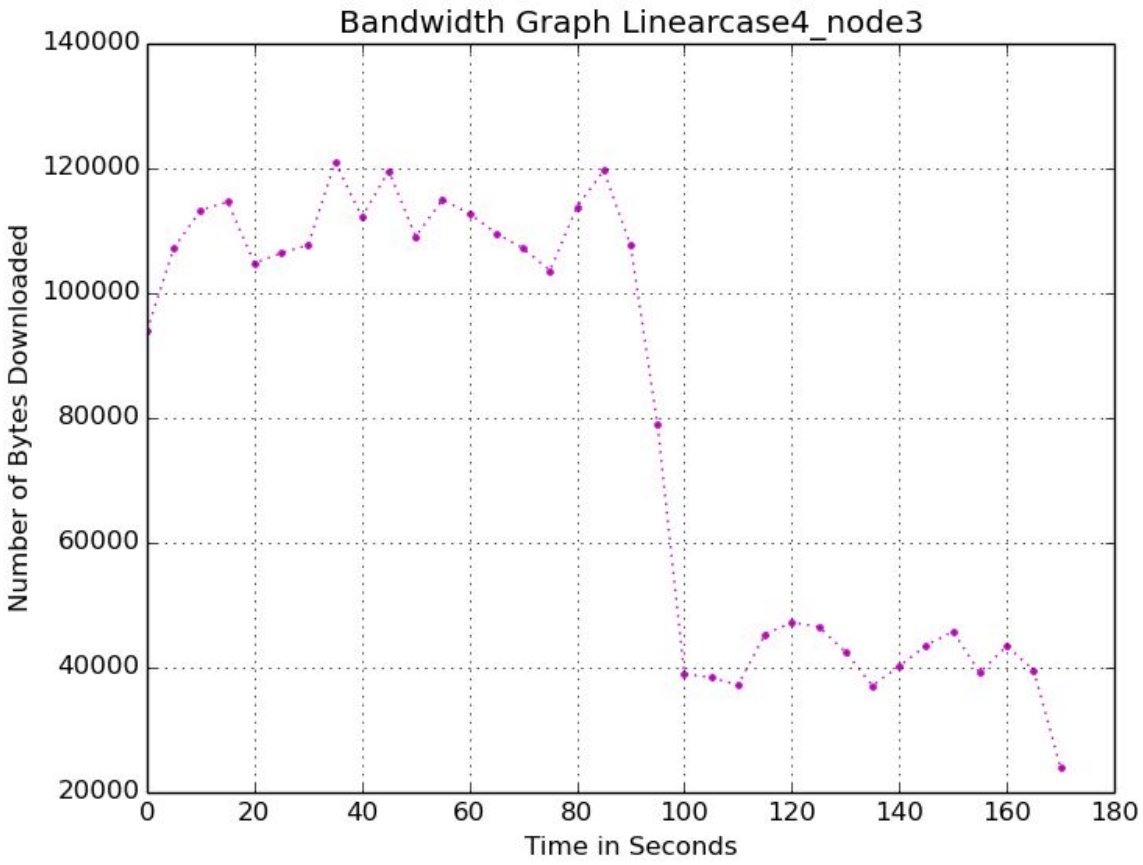


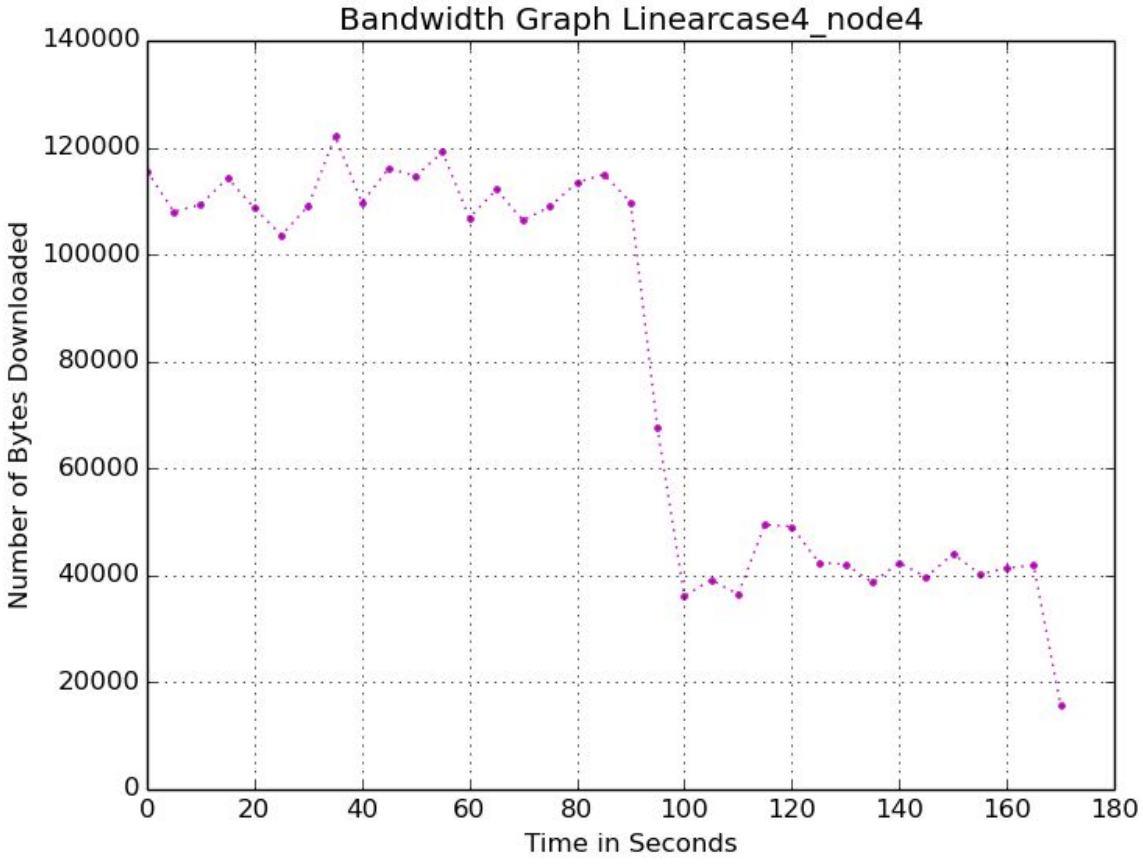
* y

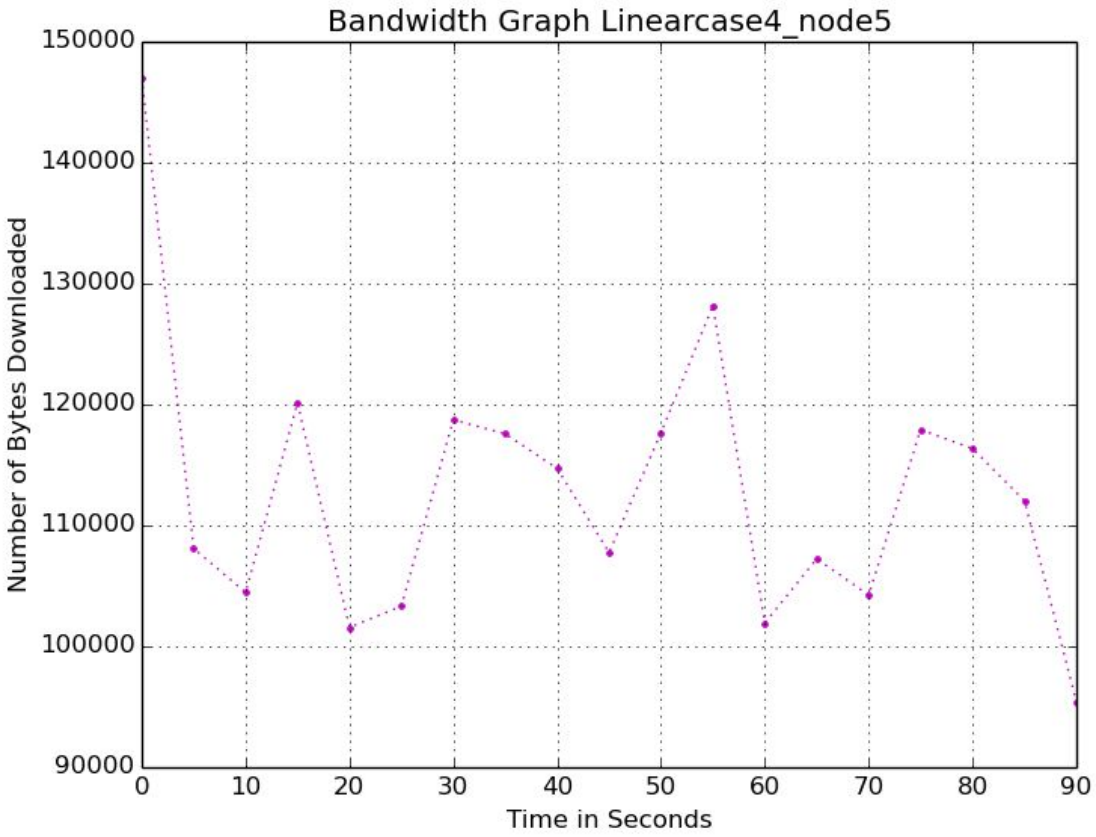
axis represents Number of Bytes Downloaded over 5 Seconds

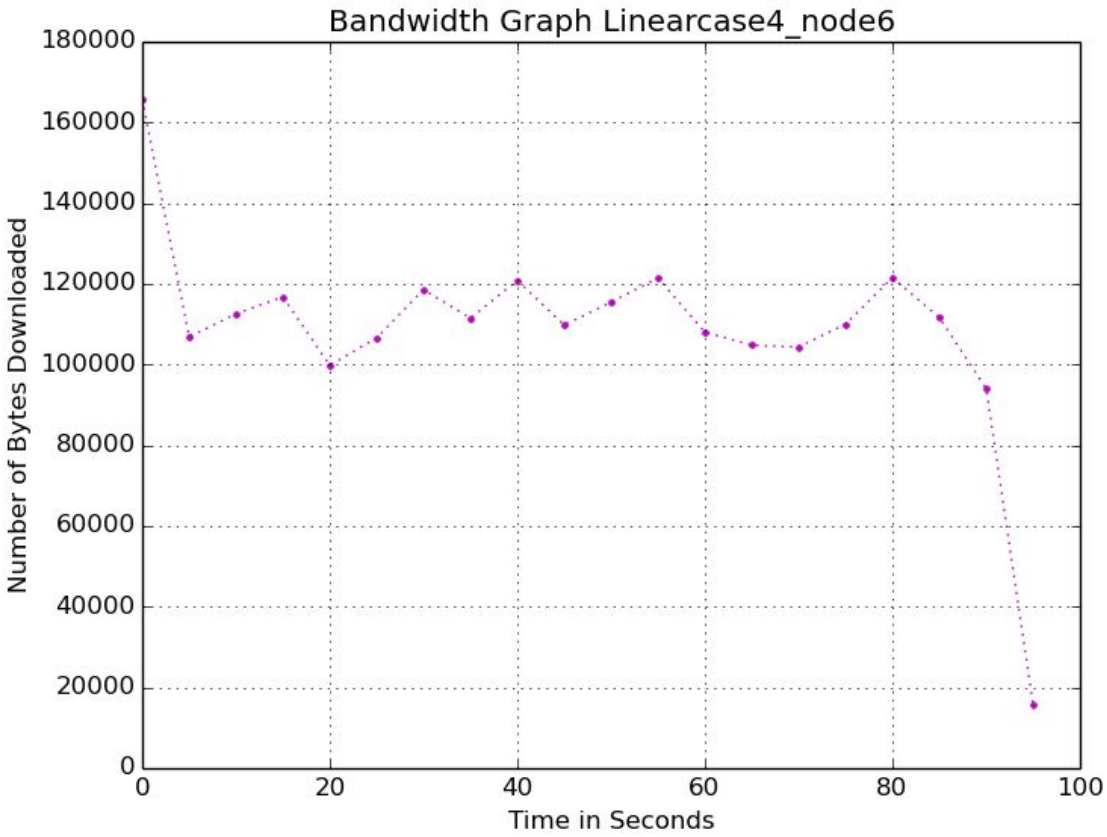
* y

axis represents Number of Bytes Downloaded over 5 Seconds

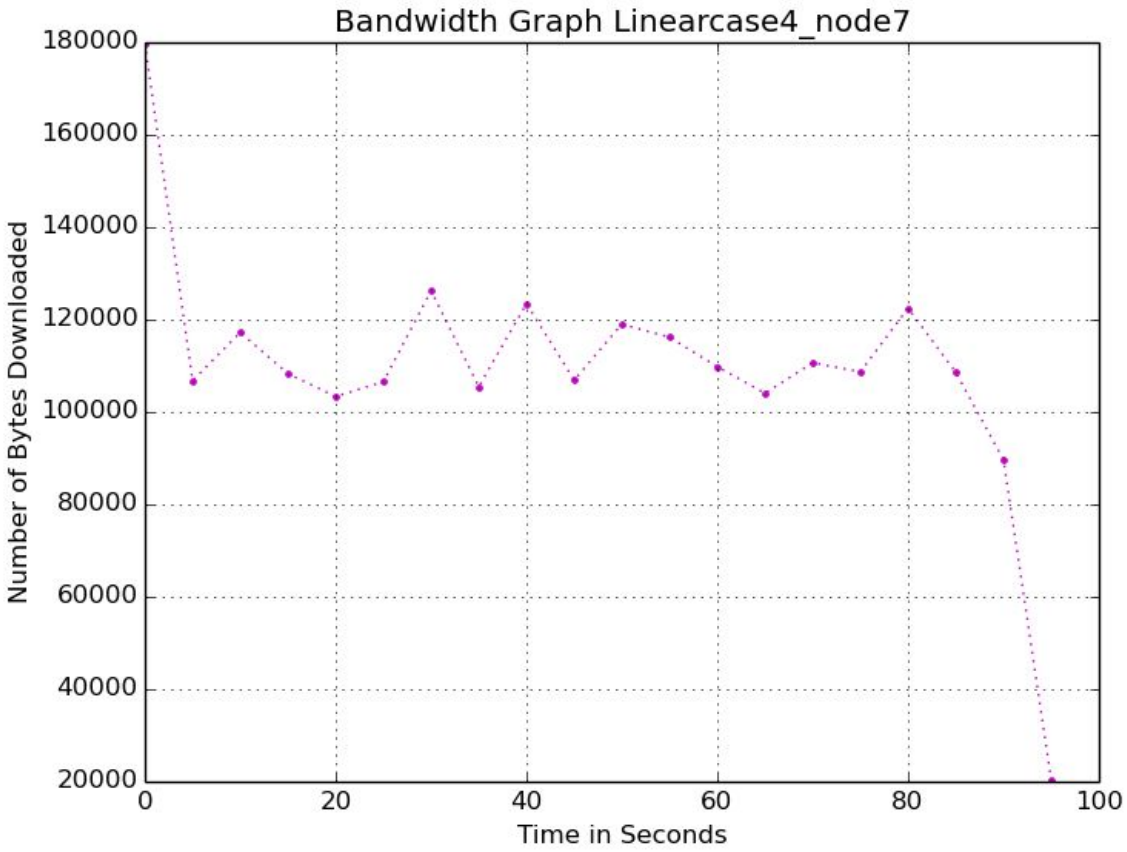








* y axis represents Number of Bytes Downloaded over 5 Seconds



* y axis represents Number of Bytes Downloaded over 5 Seconds