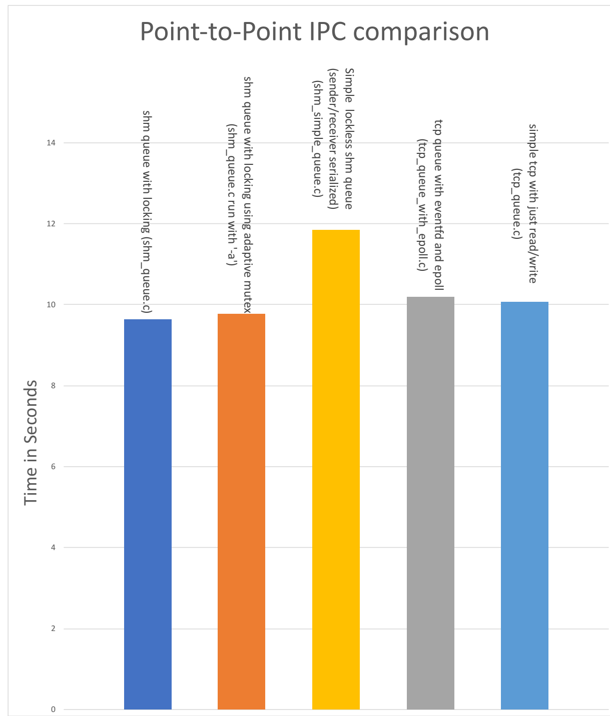
Point-to-Point IPC Comparison

```
What is this experiment
====================
In this experiment, we compared the 5 IPC mechanisms. We repeated each test 5 times
We tried to keep the paramenters comparable as much as we can
Total number of objecets to be transfered:  1,000,000,000
Size of each object:                        50 bytes
Size transfered between system calls:       100,000 bytes
Total number of batches:                    500,000


Some details about the experiment
========================
We have 1 sender and 3 receivers running on a VM running an Ubuntu VERSION="18.04.2 LTS (Bionic Beaver)", kernel version 4.18.0. The VM was assigned 4 cores on an X86 processor
with 8 cores

There are few comments regarging to "shm_queue.c" due to its complexity
- Remember that the queue is divided into packets and that a batch is a number of packets
- We tried to make sure that a total batch is of size 100,000 bytes. But in fact a batch was slightly smaller because of the packet header size. Hence  instead of 500,000 batches
there were 500501 batchs- The mutex locking/unlocking occurs at the beginning and the end of each batch. As it is known, mutex locking will cause a system call unless there is
contention
- We noticed that the most impactful parameter is the number of times the queue becomes completely empty or completely full.
- When the former occurs, the receiver waits on epoll_wait() for a pulse on the eventfd from the sender to indicate that the sender has queued some packets and it is time for the
receiver to start dequeueing.
- When the latter occurs, the sender waits on epoll_wait() for a pulse on the eventfd from the receiver to indicate to the sender that the receiver has drained some packets (i.e.
queue went below low water mark) and hence the sender can start queueing data.
- The last comment is regarding the "adaptive mutex".
- The idea of an adaptive mutex is if the locking sees that it may go into system call because of contention, it will try to spin a little bit to avoid that. This approach favors
using more CPU over making more system calls.
- We notices that using adaptive mutex slightly reduces performance. Most likely because our experiments are VERY CPU intensive and that spinnig eats up CPU
- The queue size was 256 packets, each packet is 50,000
- The low water mark is 128 packets
```




```
Buidling and  Command line used in the experments
==================================================
shm_queue.c with standard mutex
--------------
Building
rm shm_queue;  gcc -o shm_queue shm_queue.c -lrt -pthread -Wall -Wextra
Running the Sender:
./shm_queue -t -n 1000000000 -p 50000 -b 2 -o 50 -a
Running the Receiver:
./shm_queue -n 1000000000 -p 50000 -b 2 -o 50 -a

shm_queue.c with adaptive mutex
--------------
Same command line as shm_queue but added "-a" to sender and receiver

shm_simple_queue.c
-----------------------
Building
rm shm_simple_queue;  gcc -o shm_simple_queue shm_simple_queue.c shm_common.c -lrt -pthread -Wall -Wextra
Running the Sender:
./shm_simple_queue -t -o 50 -b 20000 -n 1000000000
Running the Receiver:
 ./shm_simple_queue -o 50 -b 20000



tcp_queue_with_epoll.c
----------------------
Building
rm tcp_queue_with_epoll; gcc -o tcp_queue_with_epoll tcp_queue_with_epoll.c -lrt -pthread -Wall
Running the Sender:
./tcp_queue_with_epoll -t -o 50 -b 2000 -n 1000000000 -p 65001
Running the Receiver:
./tcp_queue_with_epoll -o 50 -b 1000 -n 1000000000 -p 65001

tcp_queue.c
-------------
Building
rm tcp_queue; gcc -o tcp_queue tcp_queue.c -lrt -pthread -Wall
NOTE
-b specifies the number of objects. Hence 2000 objects each of size 50 bytes
means a batch is 100,000 bytes
Running Sender:
./shm_queue -n 1000000000 -p 50000 -b 2 -o 50 -a
Running Receiver:
./tcp_queue -o 50 -b 2000 -n 1000000000 -p 65002
```

Time in Seconds to reliably transfer 1000,000,000 objects each of size 50 bytes

		repetition 1	repetition 2	repetition 3	repetition 4	repetition 5	Average	stddev	Min	Max
shm queue with locking (`shm_queue.c`)		9.657623	9.569354	9.603463	9.498156	9.852691	9.6362574	0.13409281	9.498156	9.852691
	Number of times receiver wakeup needed (queue empty)	2302	2216	2230	2204	2338	2258	58.7367006	2204	2338
	Number of times sender wakeup needed (queue full)	2316	2242	2199	2214	2349	2264	65.4560922	2199	2349
shm queue with locking using **adaptive mutex** (`shm_queue.c` run with '-a')		10.133244	9.52705	9.967378	9.818893	9.421098	9.7735326	0.29748517	9.421098	10.133244
	Number of times receiver wakeup needed (queue empty)	2488	2051	2393	2128	2220	2256	181.86396	2051	2488
	Number of times sender wakeup needed (queue full)	2492	2072	2405	2177	2242	2277.6	170.189013	2072	2492
Simple lockless shm queue (sender/receiver serialized) (`shm_simple_queue.c`)		11.498142	12.051948	11.845329	11.719317	12.080871	11.8391214	0.24207262	11.498142	12.080871
tcp queue with eventfd and epoll (`tcp_queue_with_epoll.c`)		10.175977	10.276338	10.207182	10.200836	10.06165	10.1843966	0.07807091	10.06165	10.276338
simple tcp with just read/write (`tcp_queue.c`)		10.321483	10.626222	9.775946	9.802784	9.859803	10.0772476	0.37895214	9.775946	10.626222

Point to Multi-point IPC Comparison

What is this experiment
=====================
We modified the code from the single receiver code so that a sender can send objects to multiple receivers using a single shared memory queue containing a single copy of each message that needs to be received by multiple receivers

For this experiment, we ONLY modified
- shm_queue.c: maximal use of shared memory
- tcp_queue.c: maximal use of TCP socketxs

these two represent the two extremes of reliable point-to-multi-point IPC techniques within the same CPU.
IN particular tcp_queue.c is optmized for sheer sendung and receiving over TCP as follows
- The sender is doing nothing and looping to send one buffer at ta time using "send()" to each receiver
- Each receiver is doing nothing except looping and receiving one buffer at a time using recv()

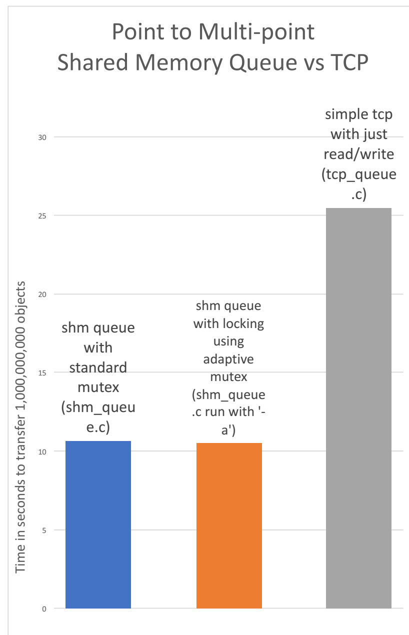
How did we run the test
=====================
- To make the test comparable with the point-to-point case, we used the same parameters EXCEPT for the number of receivers

- We have 1 sender and 3 receivers running on a VM running an Ubuntu VERSION="18.04.2 LTS (Bionic Beaver)", kernel version 4.18.0. The VM was assigned 4 cores on an X86 processor with 8 cores

- For the "tcp_queue.c", we have

```
  Total number of objects to be transfered:  1,000,000,000
  Size of each object:                       50 bytes
  Size transfered between system calls:      100,000 bytes
  Total number of batches:                   500,000
```

- For "shm_queue.c"
  We still have the same number of objects. But because of the packet header, the number of packets sent/received is about 0.1% more. Hence instead of 500,000 batches there were 500,501 batchs



Command Line options for shm_queue.c

Sender with standard mutex

```
./shm_queue -t -n 1000000000 -p 50000 -b 2 -o 50 -r 3 -u
```

Sender with adaptive mutex

```
./shm_queue -t -n 1000000000 -p 50000 -b 2 -o 50 -r 3 -u -a
```

Receivers

```
./shm_queue -n 1000000000 -p 50000 -b 2 -o 50 -r3 -i0 -u
./shm_queue -n 1000000000 -p 50000 -b 2 -o 50 -r3 -i1 -u
./shm_queue -n 1000000000 -p 50000 -b 2 -o 50 -r3 -i2 -u
```

Command line options for "tcp_queue.c"

Sender

```
./tcp_queue -t -o 50 -b 2000 -n 1000000000 -r3 -p 65002
```

Receiver(s)

```
./tcp_queue -o 50 -b 2000 -n 1000000000 -r3 -p 65002
```

Important comment about timing for "tcp_queue.c"
==
Except for the last receiver, the sender, and hence all receivers except the last, have wait until the last receiver joins. Hence all receivers except the last one will measure significantly more time that the last one

Hence the accurate time is outputed by the last receiver

Some comments related to shm_queue.c
==================================
- For a single receiver, we used to use eventfd for singalling where the eventfd() file descriptor is sent by the sender to the receiver using ancillary SCM_RIGHTS message
- In theory, multiple processes can be waiting using epoll_wait() on that eventfd file descriptor and they should be awakened by a single write() from the sender to that file descriptor
- But I noticed that When I use eventfd with multiple receivers, I can see   that a pulse gets lost and we fall into a deadlock after few hundred thousands to few million objects.
- Hence I wll NOT use eventfd for signalling
- This may very well be a bug in my code or the fact that a single write to an eventfd cannot be used reliably to wakeup multiple waiters on that same eventfd using epoll_wait()
- Hence I am disallowing the use of eventfd for signaling from sender to recievers when there are more than one receiver
- Instead, I am using the AF_UNIX socket that was used to send the SCM_RIGHTS message for signaling by individually sending a single byte to each receiver by the sender or from a receiver to the sender
- I even disallowed using eventfd with multiple receivers for a receiver to pulse itself when the user uses the "-d" option because I saw some problems
- Bottom line, it seems sharing the same eventfd file descriptors among multiple processes (and possibly multiple threads within the same process) is either unreliable OR needs some different code
- It is also possible to use a separate eventfd file descriptor per receiver instead of the AF_UNIX socket. We know that point-to-point eventfd signalling is reliable from the point-to-point IPC experments.
- Anyway this is not the issue that we are addressing in this code so we will not look at it as this point in time

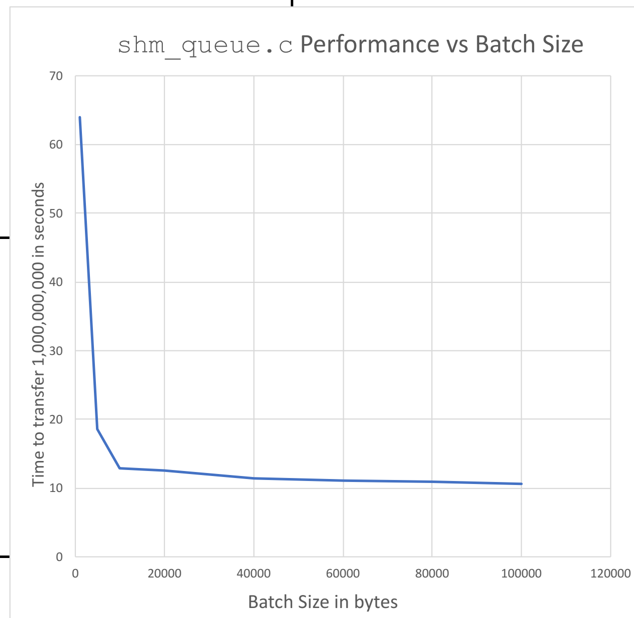
	Time in Seconds to reliably transfer 1000,000,000 objects each of size 50 bytes to all receivers								
	repetition 1	repetition 2	repetition 3	repetition 4	repetition 5	Average	stddev	Min	Max
shm queue with **standard** mutex (shm_queue.c)	10.616932	10.698094	10.408063	10.867354	10.479368	10.6139622	0.18152442	10.408063	10.867354
shm queue with locking using **adaptive** mutex (shm_queue.c run with '-a')	10.065215	10.770651	10.369507	10.683068	10.657076	10.5091034	0.2902976	10.065215	10.770651
simple tcp with just read/write (tcp_queue.c)	25.083797	26.619808	25.079775	25.577541	24.917912	25.4557666	0.69621118	24.917912	26.619808

Effect of batch size on Point to Multi-point shm_queue.c:

```
What is this experiment
=====================
The objective of this experiment is to see the effect of batch size on the
performance of point-to-multipoint shared memory queue

We did the following
- The queue size is fix at 256 packets
- The batch size in packets is fixed at 2 packets
- We varied the packet size from 500 bytes to 50,000 bytes, thereby varying the
  the batch size from 1,000 to 100,000 bytes
- At the receivers side, the batch size kept constant at 2 packets. Because the
  packet size is the same at the sender and receiver, result is having the same
  batch size in bytes at the sender and receiver

We fixed the number of objects to transfer to 1,000,000,000 where each object
is 50 bytes
```




```
`Command Line options for shm_queue.c
===================================
Sender with standard mutex
./shm_queue -t -n 1000000000 -p 50000 -b 2 -o 50 -r 3 -u


Sender with adaptive mutex
./shm_queue -t -n 1000000000 -p 50000 -b 2 -o 50 -r 3 -u -a

Receivers
./shm_queue -n 1000000000 -p 50000 -b 2 -o 50 -r3 -i0 -u
./shm_queue -n 1000000000 -p 50000 -b 2 -o 50 -r3 -i1 -u
./shm_queue -n 1000000000 -p 50000 -b 2 -o 50 -r3 -i2 -u
```

```
Quick Interpretation
====================
Remember that at every batch both sender and receiver(s) stop to update the queue
AND signal the receiver(s) and sender, respectively

Hence the smaller the batch, the more frequent the queue is updated and hence
more system calls and mutex loc/unlock as well as higher probability of
contention among the sender and receiver(s)

So it is expected to see performance degradation as the batch size goes
down. In addition,  larger than a certain size the improvement becomes less
significant while  below a certain size performance degradation becomes more
siginificant
```

Queue Size varies by packet size (Fixed 256 packets)

Batch Size		Time in Seconds to reliably transfer 1000,000,000 objects each of size 50 bytes to all receivers									
Batch Size in bytes	Packet Size	repetition 1	repetition 2	repetition 3	repetition 4	repetition 5		Average	stddev	Min	Max
100,000	50,000	10.616932	10.698094	10.408063	10.867354	10.479368		10.6139622	0.18152442	10.408063	10.867354
80,000	40,000	11.561155	11.43604	10.731869	10.828865	10.426198		10.9968254	0.48358184	10.426198	11.561155
60,000	30,000	11.030784	10.831105	10.907779	11.111847	11.314303		11.0391636	0.18815225	10.831105	11.314303
40,000	20,000	11.458606	11.403133	11.94676	11.096051	11.280716		11.4370532	0.31707253	11.096051	11.94676
20,000	10,000	12.340672	12.434161	12.843311	12.506188	12.594059		12.5436782	0.19165665	12.340672	12.843311
10,000	5,000	12.831177	12.465633	12.619161	13.13822	12.760117		12.8371688	0.2191895	12.619161	13.13822
5,000	2,500	18.962382	18.508257	17.826095	18.083704	19.321398		18.5483948	0.70865457	17.826095	19.321398
1,000	500	65.713018	62.12864	65.184959	62.189797	64.636338		63.9705504	1.69690684	62.12864	65.713018