Mkafka

G. Brown, M. Kuang and K. Wojcicki

April 13, 2018

1 Introduction

1.1 Context

The MINIX operating system (OS) uses the idea of message passing to support communication between user processes because MINIX uses micro kernels as opposed to other OS such as Linux which uses monolithic kernels. This means that processes communicate with each other without the use of shared memory. For example, when a user process does an *exit()* system call, it constructs a message and sends it to the memory manager which is the process that handles the *exit()* system call. Then the kernel will send a message back telling the other user process to exit. The message passing method means that user processes must make system calls to the kernel in order to communicate with other user processors.

Sometimes it may be useful for user processes to communicate with other user processes directly. For example, inter-process communication (IPC) is essential for microservice architectures. A microservices-based application is a distributed system running on multiple machines and each service instance is usually a process. Consequently, services must interact using an IPC mechanism.

1.2 Problem Statement

This project is addressing the inability for user processes in the MINIX OS to have an effective and easy method to communicate with one another without resorting to sockets or writing to a file. The goal is to give multiple user processes across potentially multiple machines the ability to communicate with one another in real time. This is relevant because if a user wants to build a microservice, the services would need to be able to coordinate between one another.

1.3 Result

The goal of allowing communication between user processes on one or multiple machines was achieved by creating a message broker called Mkafka inside of the kernel. Mkafka allows users to push or pull messages to or from its messaging queue. Processes can then continuously poll the server to wait for a new message to consume. If processes were on different machines, the networking was achieved by creating a user process called mkafka_net that can create a transmission control protocol (TCP) based connection to another mkafka_net user process. Mkafka_net will copy all messages that are produced on its machine and send them across the socket. These mkafka_net processes can be chained across multiple computers such that unlimited machines can be linked together.

1.4 Outline

The rest of the report is structured as follows. Section 2 presents background information relevant to this project. Section 3 describes in detail the results that were obtained. Section 4 analyses the results and Section 5 concludes the project.

2 Background Information

A messaging queue provides an asynchronous communications protocol (AMP) which is a system that pushes messages onto a message queue and does not require an immediate response to continue processing. The basic architecture of a message queue is straightforward, there are producers that create and send messages to the message queue and consumers that connect to the message queue and receive the messages to be processed. The messages on the message queue are removed when the consumer receives them. Apache Kafka, the message broker that our project is based on, is similar to a messaging queue but messages that get published to a topic do not get removed when they are consumed by a consumer. Instead, they get persisted which allows users to replay messages and allows multiple consumers to process logic based on the same messages. Each consumer group within a topic receives the same messages, however the messages are consumed by consumers at a user-defined probability. This means that not every consumer receives all the messages provided to a consumer group. The following figure depicts how messaging works in Kafka.

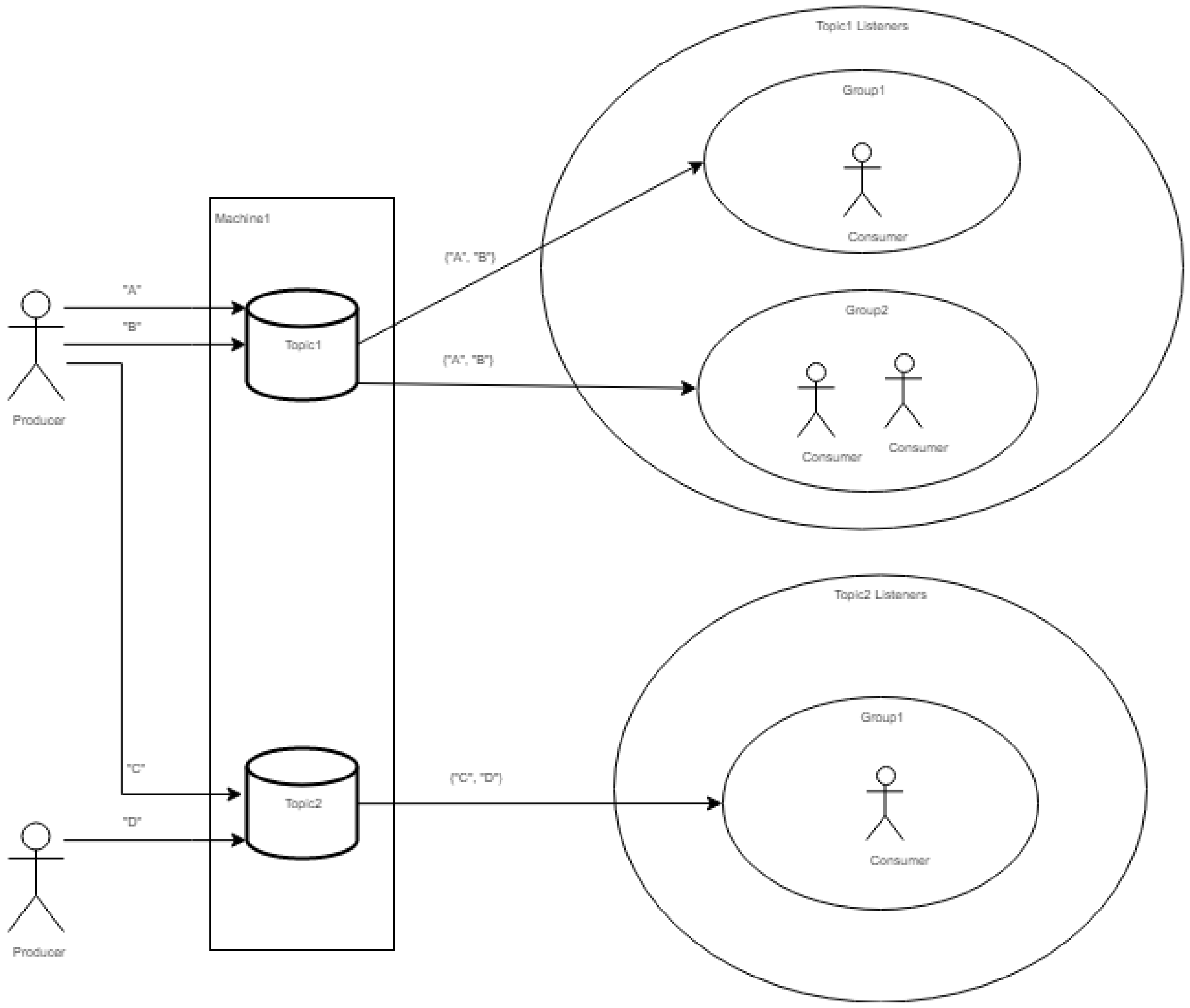


Figure 2.1. Example of a messaging queue

From figure 2.1, we can see that each consumer group of a topic will receive the same messages whenever a producer produces a message to that given topic. The consumer in Group1 of Topic1 will receive all messages from Topic1 since it is the only consumer of that group. However, the consumers in consumer Group2 of Topic1 will not. One consumer may receive both message “A” and “B” while the other consumer receives none, or one consumer may receive message “A” and the other receives “B”. The likelihood of a consumer receiving a message depends on how the user sets up Mkafka.

Minix servers sit between userspace and kernel space acting as a bridge between the two. Most of the operating system's functionality is located here. Servers allow for restricted access to the kernel which increases reliability and security of the system since all kernel calls are filtered through a server. Other advantages of minix servers include protection against infinite loops, crashing servers and bad pointers, on other architectures these error would kill/hang the system, but on minix these would only affect the server, which could then be recovered from through the resurrection server which monitors other servers.

3. Result

The underlying structure of Mkafka is depicted in Figure 3.1.

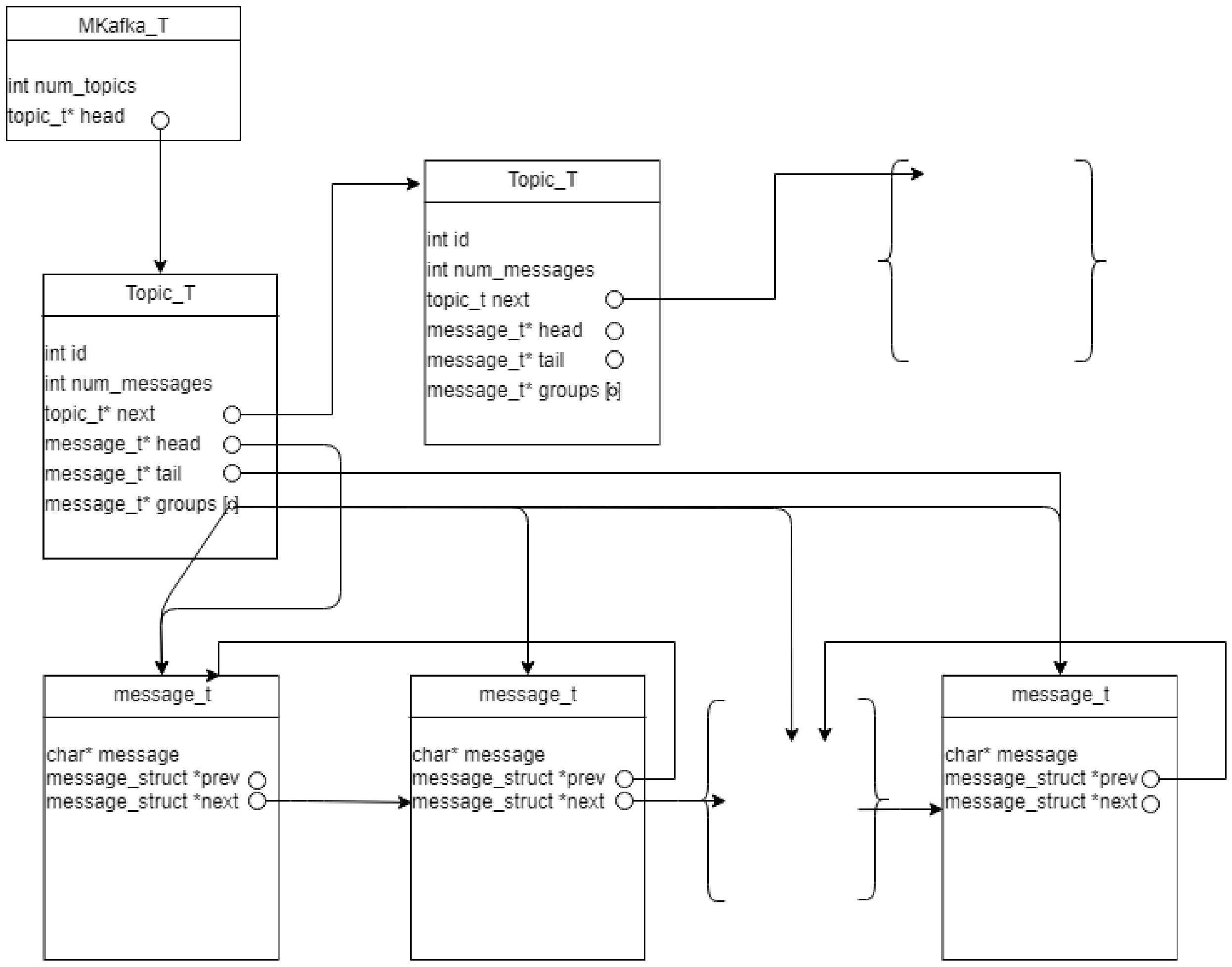


Figure 3.1 Structure of Mkafka

As can be seen, Mkafka_T stores a linked list of topics where each topic stores a linked list of message_t pointers, and the idea of consumer groups are stored in an array of message_t pointers called *groups*. When a producer produces a message, it specifies which topic it wants to produce the message to and Mkafka_T pushes the message to the front of the specified topic. This means the *head* pointer is always pointing to the newest message while the *tail* points to the oldest message in the queue. The *groups* variable allows consumers to consume messages from the topic. At the start, all message_t pointers in *groups* point to the tail of the topic. When a consumer consumes a message from that topic, that message_t pointer of the specified group increments to the next oldest message in the queue. This way, we can allow for data persistence because if a user wants to replay a message, they would subscribe to the topic into a new group so the message_t pointer would point back to the tail where the oldest message exists.

To illustrate how a user collaborates with Mkafka, the following interaction diagrams shows more in detail of how a user can pull and push from Mkafka.

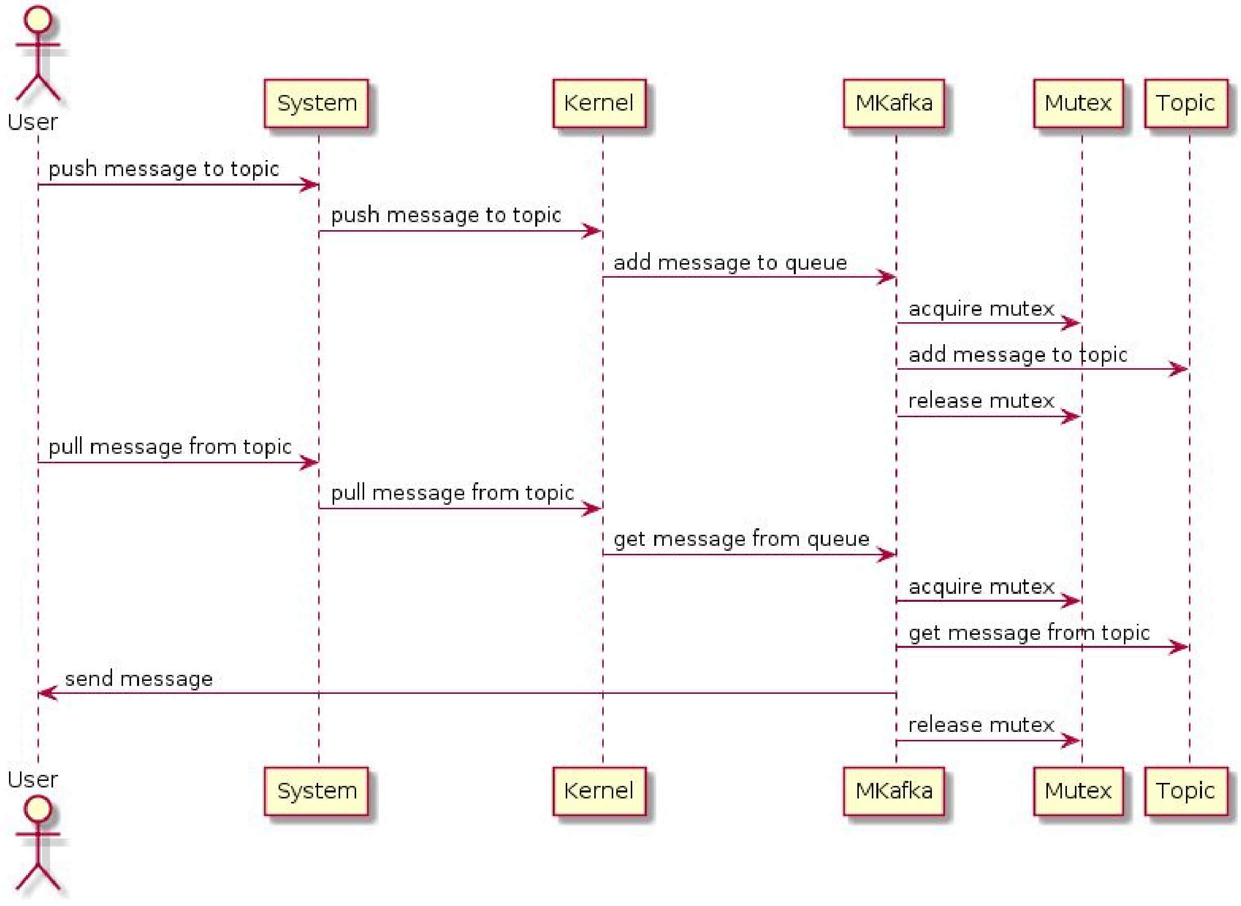


Figure 3.2 Users pushing and pulling from Mkafka

Figure 3.2 describes how a user pushes and pulls from Mkafka. When a user wants to push a message to a topic in Mkafka, it makes a system call *push* to the kernel which then tells Mkafka to add a message to its messaging queue. Mkafka first needs to acquire the mutex and once it has done so, it adds the message to the specified topic then releases the mutex. Similarly, when users want to pull from a topic, it makes a system call *pull* to the kernel which then tells Mkafka to get new a message from the queue. Mkafka first acquires for a mutex then gets the message from the topic and sends it back to the user.

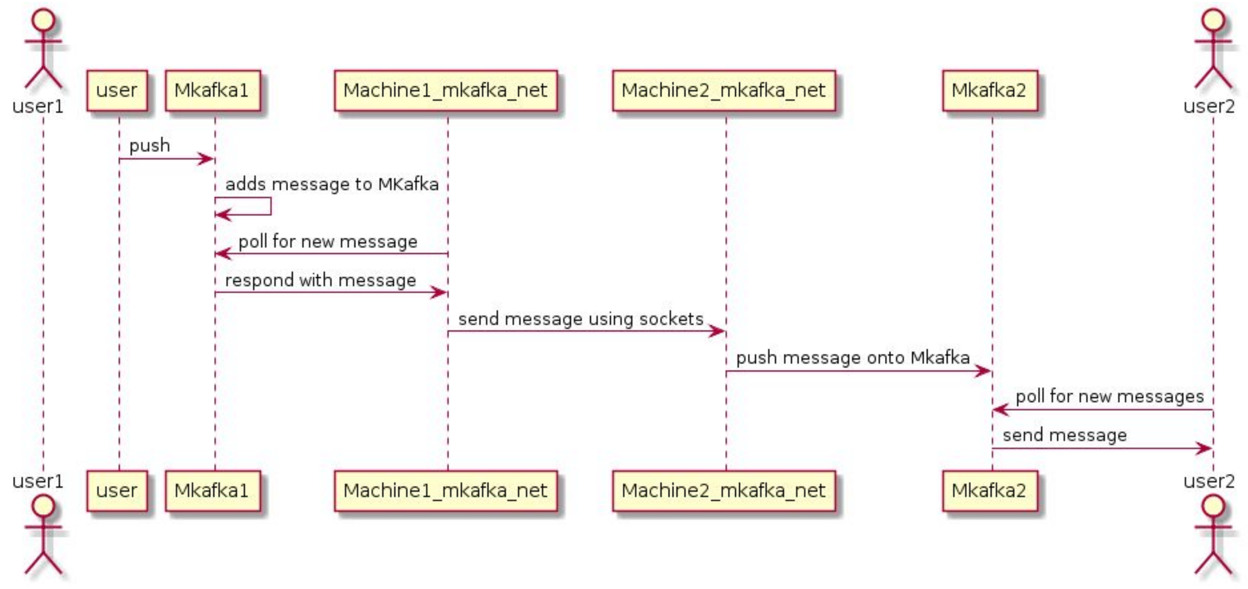


Figure 3.3. Users pushing and pulling from Mkafka from separate machines

If processes are in separate machines, they can communicate with each other via sockets. FIgure 3.3 shows how pushing and pulling from Mkafka works with networking in place. Using the same logic from figure 3.2, user1 pushes a message into Mkafka1. Mkafka1 would then add the message into its queue. The Machine1_mkafka_net is continuously polling Mkafka1 for new messages and since user1 pushed a message into Mkafka1, Mkafka1 will respond to the Machine1_mkafka_net with the new message. When Machine1_mkafka_net receives the message, it sends the message using sockets to the other Machine2_mkafka_net. Machine2_mkafka_net then pushes the message to its Mkafka. When user2 polls for a message, Mkafka2 responds by sending a message to user2.

One example of how Mkafka can be useful is the simulation use case. In the figure 3.4, it shows that a user can send workloads to Process1 which then gets added to Mkafka. Machines M1, M2 and M3 continuously poll Mkafka for workloads. Once they have received a workload, they would process the simulation and the result of the simulation is sent to the database.

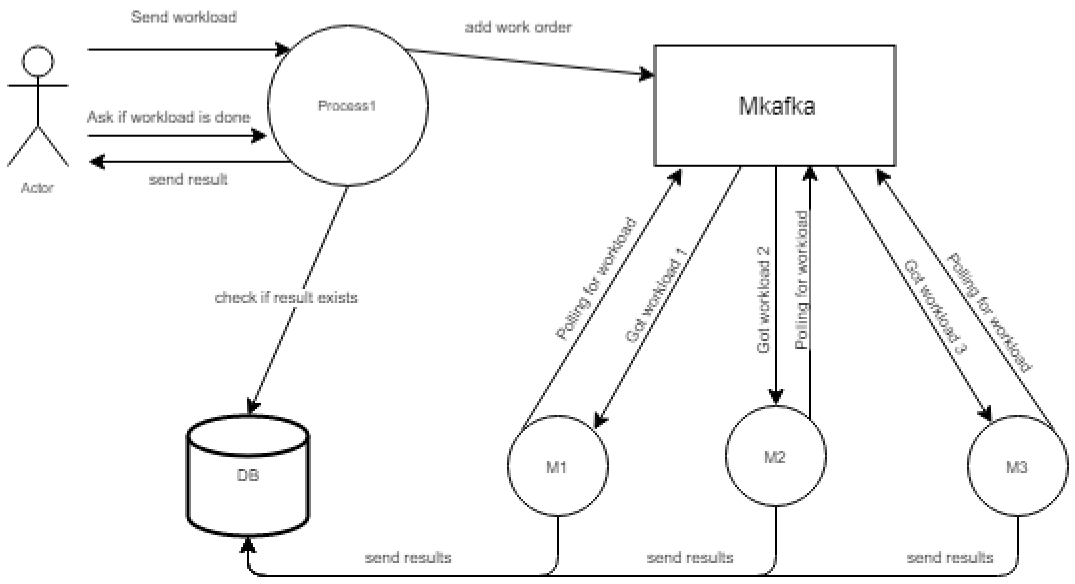


Figure 3.4 Simulation use case scenario

The user can then ask Process1 if the workload is complete and Process1 will check the database for results and send the result back to the user.

Another example of how Mkafka could be useful is collecting data from stress tests on multiple processes as depicted in figure 3.5.

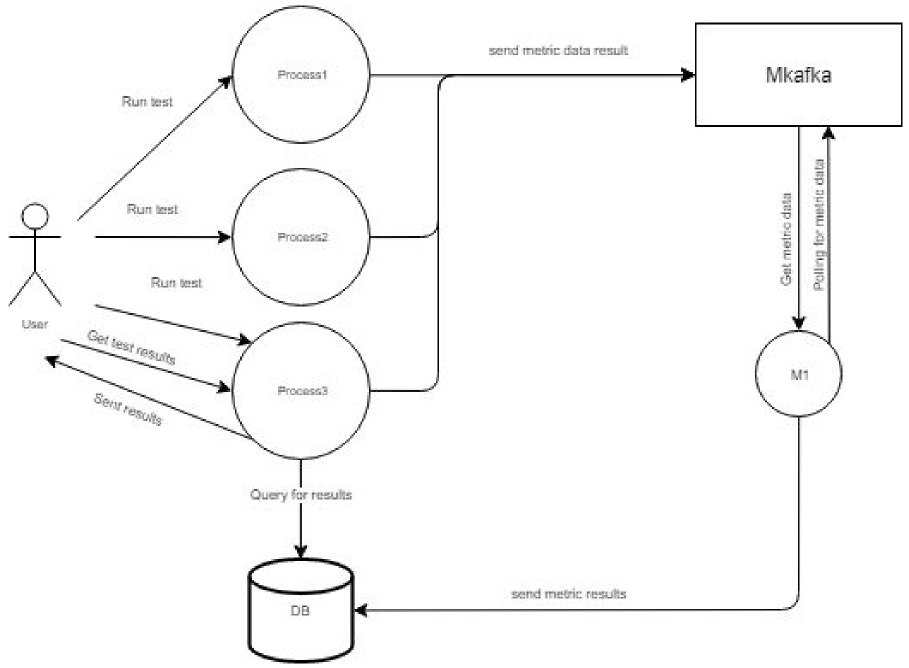


Figure 3.5. Stress test use case scenario

The user would tell each process to run the test. Each process would generate performance metrics during the test and send the data to Mkafka. The consumer M1 would continue to poll for new metric data and send the results to a database (DB). The user can then get the results after the tests are complete.

A user with ample knowledge of how a messaging bus works will also have an easy time transitioning to using MKafka. Initially it was planned for the networking portion of Mkafka to be in its own server. This was to follow the design of Minix where each portion of functionality is in its own server. However similar to Linux, Minix has a subset of the C library which you can use inside of the kernel. This subset does not include the socket library. It was attempted to manipulate the makefiles and build process of Minix to link in the socket.o and socket.c files however this was unsuccessful. Adding the entire C library was attempted next, while this successfully linked the socket library it caused problems later on in the compiling which we were unable to resolve. As a result it was decided to put the networking portion of Mkafka inside of a user process that could be started when needed. Later it was also agreed that adding networking to a kernel server is not the best idea either due to the fact it opens up the outside world to create a connection directly with a process inside the kernel. Opening up many potential security risks.

For Mkafka it was instead decided if the linked list was full and a new message was pushed the oldest message would be removed. This however can cause issues if for example two processes p1 (consumes slowly), p2 (consumed quickly) are consuming from Mkafka with a third process p3 depending on the combined analysis of the messages by p1, p2. It might be possible that p2 analyzes a message that is removed before p1 has time to analyze it. Now p3 has a message that is half analyzed and cannot move forward with it. This adds complexity to the users using Mkafka to account for such scenarios.

There are a few possibilities to fix the scenarios mentioned above. One is to keep messages forever, this way all messages will eventually be processed. However this comes with the obvious drawback that Mkafka, given the fact it stores all messages in memory, would eventually consume all the RAM and kill the machine. Another possibility is to not let users push messages once the topic is full and instead return an error code indicating topic full. The problem with this is if you have a consumer that is dead/slow the system could come to a halt waiting for that consume to slowly consume messages. The final possibility is to let the linked_ ist grow infinite in size and prune based on timestamps. A user could specify how long a message can last in a topic ie 1 day, 1 hour etc and then Mkafka would have to handle removing old messages. This seems like the optimal solution, it does not let the topic grow infinity in size and given the user understands their system they should have a decent understand of how much a consumer can lag behind. However now Mkafka would have to, on some time interval, iterate over all the messages and find old messages which could be expensive.

For the networking portion it was decided to do peer to peer networking instead of a server-client architecture. The reason for this was primarily for ease of coding and for proof of concept. Ideally the networking should be done using server-client. One of the reasons for this is performance as discussed in Section 4, requiring a message to go across multiple machines greatly increases the time it takes for messages to be spread out. In addition with peer to peer it becomes nontrivial how to synchronize all clients and ensure data order (in Mkafka there was no synchronization like this meaning the order of messages on multiple machines could be different). If there are two machines m1,m2 who both produce messages w1, w2 the order of these two messages should be the same on both machines otherwise desynchronization can happen. With server-client there is only one source of truth which is the server, therefore order would always be maintained. However there being only one source of truth can cause outages of service. In the peer 2 peer model if one machine goes down the rest of them can continue working. In server-client if the server goes down all the clients are stuck. This can be solved by having multiple servers but then the issue of data synchronization and load balancing come into play.

4. Evaluation and Quality Assurance

Multiple test programs were created to exercise many of the typical execution paths of the software. There were 3 consumer programs that were made that would consume from a given topic/consumer group at different rates: slow, medium and fast speed. And to match that 3 producer programs were made to produce to a given topic at the same speed rates. Using these programs together many different possibilities were tested. Such as one medium producer and 3 consumers, one consuming slowly and eventually missing some messages, one consuming at a medium speed and consuming all messages, one consuming at a fast speed and having spare cycles. Another scenario tested was fast producer with 2 consumers one medium one slow. The two consumers were put into the same consumer group resulting in the medium one processing most of the messages with the slower consumer consuming much less.

Data was collected about three different scenarios that could happen. The first scenario that was analyzed was with one producer producing 7 messages, while 1-n processes would consume these messages. The wall-time was measured.

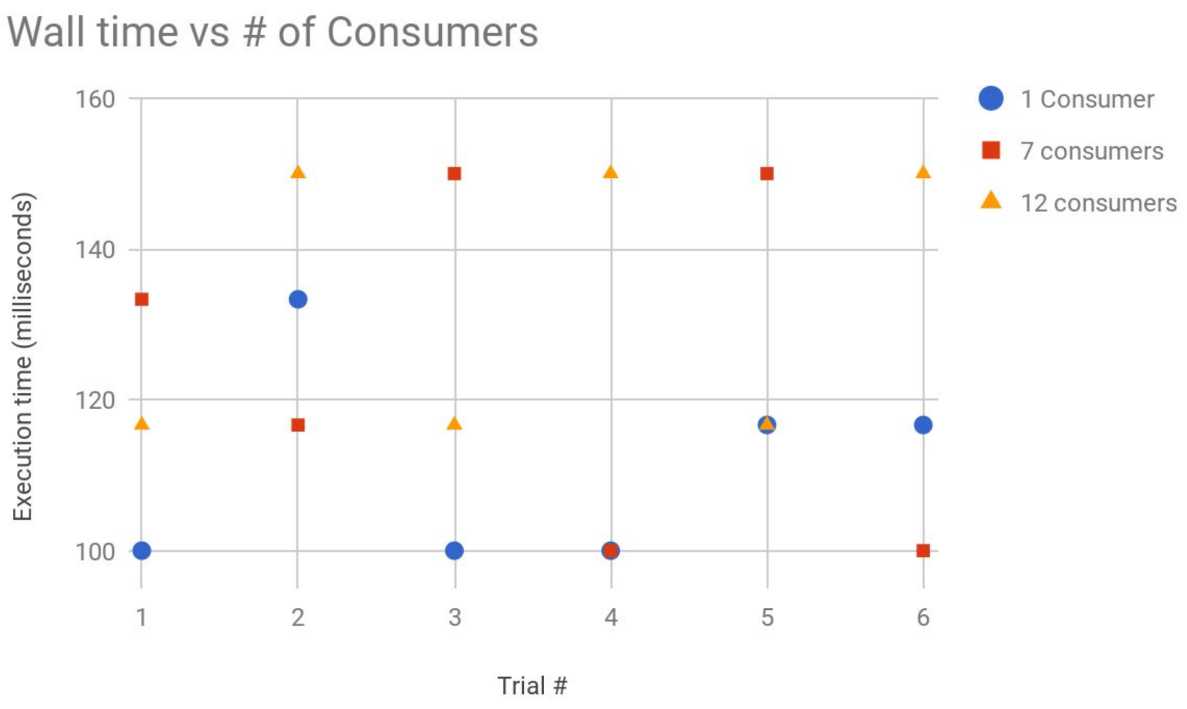
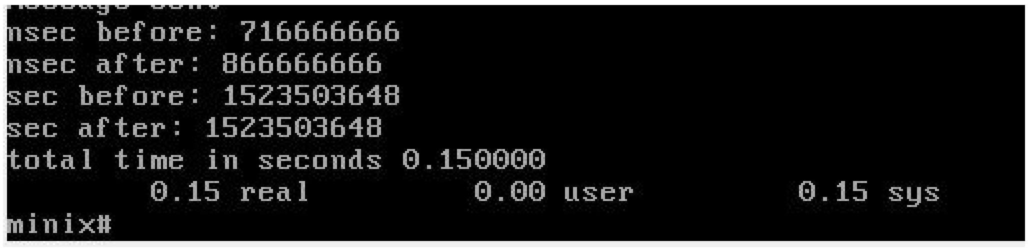


Figure 4.1: Graph showing how long it takes 1,7 or 12 consumers to consume 14 messages.



```
nsec before: 716666666
nsec after: 866666666
sec before: 1523503648
sec after: 1523503648
total time in seconds 0.150000
        0.15 real          0.00 user          0.15 sys
minix#
```

Figure 4.2: showing example run time of 12 producers consuming 14 messages

In figure 4.1 the wall time of how long it took 1,7 or 12 consumes to consume 14 messages that were produced by another producer. One thing to note is minix's/the laptop used to test this clock does not seem to have a very accurate resolution time as seen in figure 4.2. These results show that sending and receiving messages using Mkafka is very quick mostly due to the speed of Minix's IPC. It also shows that adding more consumers while it does decrease performance it is not too drastic.

Next data was captured for round trip time for one process to send a message that will be received by a process another another machine who will then send a message back. One note to be made is that the mkafka server user process was slightly modified during these trials by reducing the timeouts, sleeps to improve performance.

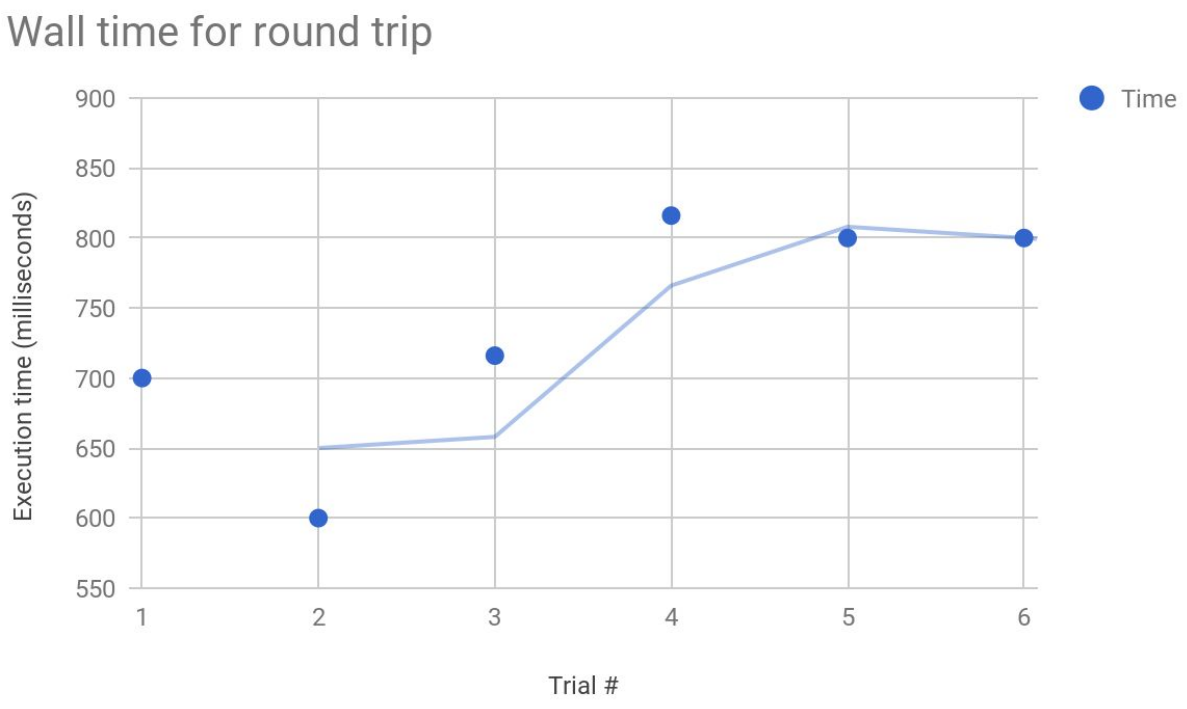
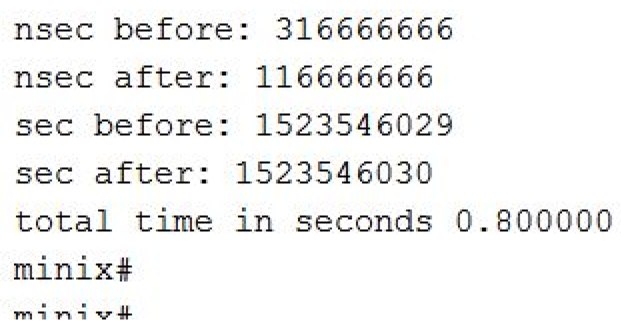


Figure 4.3: Graph showing the time to took for a message to be sent from one machine to another and back again



```
nsec before: 316666666
nsec after: 116666666
sec before: 1523546029
sec after: 1523546030
total time in seconds 0.800000
minix#
minix#
```

Figure 4.4: Showing example run time of second scenario

In figure 4.3 one can see the moving average seems to go close to 800ms, while this is not instantaneous and is much worse than local consuming it is most likely due to the polling that is done in the network Mkafka user process. Since as discussed before, the networking cannot be done in its own server and instead must be done as a user process, this requires the process to poll kafka while also polling the socket for any new packets. The timeout for the socket is ~400ms meaning potentially up to half of the time is just the network Mkafka user process waiting idly for a message. A potential solution for this is for the network Mkafka user process to fork/use threads, and have one part reading from Mkafka and one part reading from the socket. This would reduce some of the latency however as networking will always add some latency.

The final scenario under which data was extracted for was round trip time between three machines that were linked together as shown below.

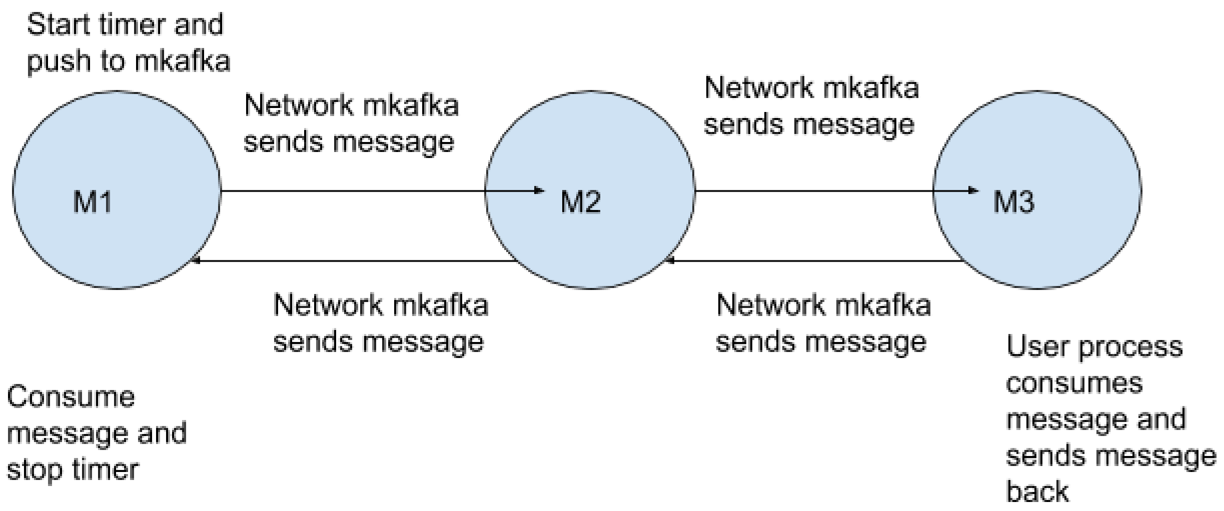

Start timer and push to mkafka
Network mkafka sends message
Network mkafka sends message
M1
M2
M3
Network mkafka sends message
Network mkafka sends message
Consume message and stop timer
User process consumes message and sends message back

Figure 4.5: showing the final scenario and when the timer will be started and stopped

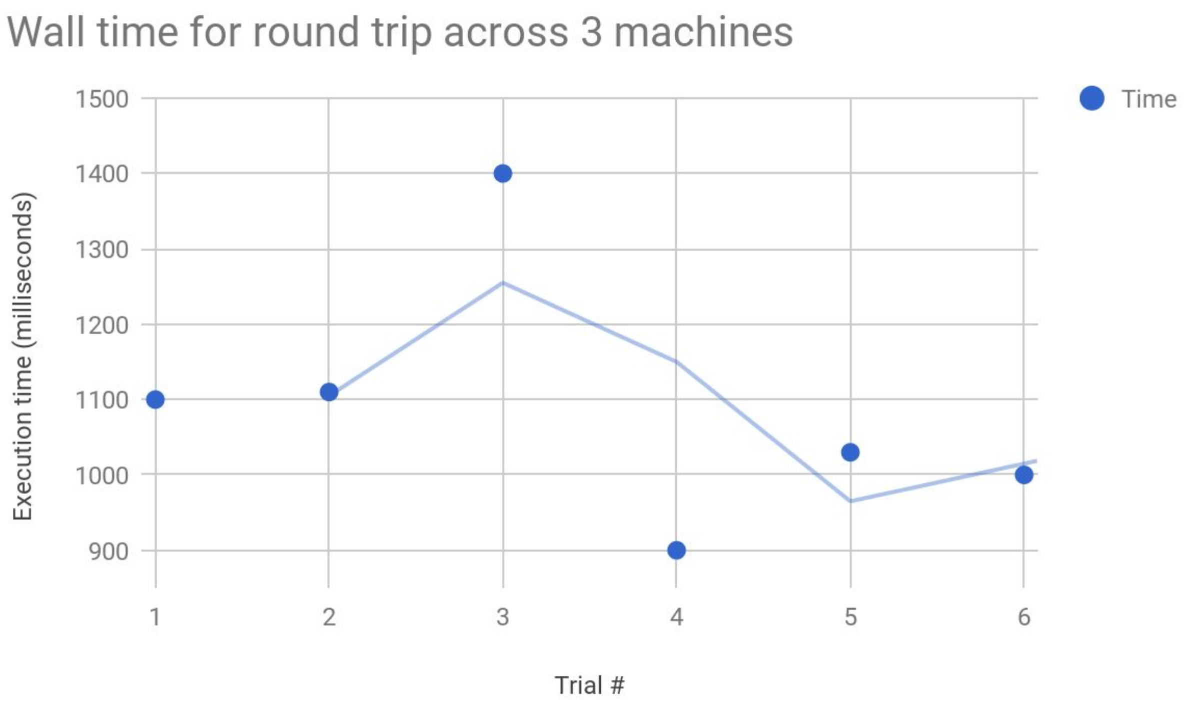
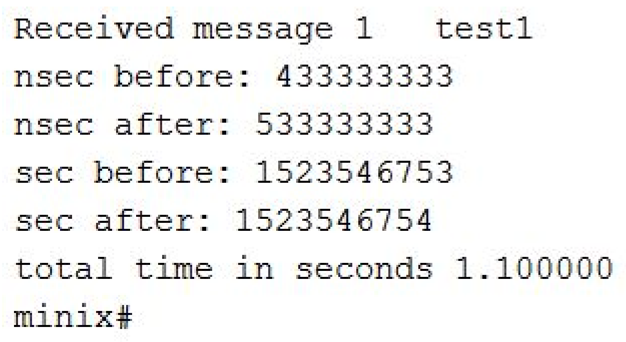


Figure 4.6: Graph showing the time to took for a message to be sent from m1 to m3 and back again



```
Received message 1   test1
nsec before: 433333333
nsec after: 533333333
sec before: 1523546753
sec after: 1523546754
total time in seconds 1.100000
minix#
```

Figure 4.7: Showing example run time of the third scenario

In figure 4.6 the average seems to around 1000-1100ms. The relatively large fluctuations can be explained as stated before by the polling used inside the networking portion of Mkafka. These results indicate that our choice of design (using peer 2 peer and connection model) was not the best choice as the more machines you add the latency just keeps on increasing rapidly.

Like all software despite the many strengths of Mkafka there are also some weaknesses. To test out the ease of use and ideology of Mkafka a usability study was done. During the usability study the participants were asked to create 2 small programs in C. One program which would take input from the user and push to Mkafka. The other program would consume from Mkafka process the message (simply print out the message then sleep) and repeat. The participants were assisted with doing basic C things (such as getting user input, include statements etc) and as well show the header file to our system calls. There were 3 participants, all of these individuals have different computer science background but are all currently working in a software company. The rating scale is from 1 to 5 where 1 is poor and 5 is excellent.

The first participant was a UI Designer who has relatively limited experience with C and Unix like operating systems. Due to their experience they found the programs very hard to make and required a lot of assistance. This resulted in them giving a 1 out of 5 for the ease of use factor. The participant however understood the need for a messaging bus in distributed applications and gave it a 3 out of 5 for ideology. They compared Mkafka to event driven UI frameworks which are quite useful in their work.

The next participant was a coop front-end developer. Due to school and previous work-terms they has experience working with Unix like operating systems and C. Unlike the previous participant they were successfully able to create the two programs with minor problems. However they thought documentation inside of the header file was quite poor and could use some work to explain what the different input parameters were used for. This resulted in them giving a 3 out of 5 for ease of use. At previous work terms they worked with other messaging queues and agrees there is a need for such programs, resulting in a 4 out of 5 for ideology as well. One problem they mentioned is our lack of persistence, if the machine were to restart all the messages would be lost and there would be no way of recreating them. They mentioned that at previous work places there was a big emphasis on high availability with data persistence being one of those factors.

The final participant was an experience backend software developer with many years of experience with Unix like operating systems and C. They were able to create the two programs no problems. They were concerned at the state of documentation and debugging functions. They gave Mkafka a 3.5 out of 5 for ease of use. They believe that most distributed systems will benefit from having some sort of messaging queue for asynchronous communication, however they believed we lacked some functionality such as partitioning our topics and offset management, resulting in a 3 out of 5 for ideology.

Tester	**Ease of Use**	**Ideology**
A UI Designer	1	3
B Coop front-end developer	3	4
C Back-end developer	3.5	3

From the usability study as a group we understood from a user's perspective Mkafka was still very rough and required a lot of work. Primarily in the form of documentation and perhapes mini example programs that one could build off of. They believed the functionality was good but again would need to be better documented so the users could understand the power of it. With respect to persistence while it would be useful to add, would add a lot of complexity to the program to ensure all messages are written to disk and then can be read from disk on startup. Offset management, this is the process of instead of notifying Mkafka that you processed the message instead of Mkafka assuming when you received the message you also processed it, would also a lot of complexity but would be very useful in cases where just pulling the message does not guaranteed that it was processed. Other functionality that could be added based off the participants comments is addressed in the conclusion.

5. Conclusion

5.1. Summary

Mkafka is able to send messages between multiple users using a kafka-like messaging queue. It is also able to send messages between multiple computers. Messages are ordered under topics allowing for multiple queue to exist at once. Each message can be consumed multiple times through the use of consumer groups. When a message has been read from each group, or the queue is filled up it is removed. Multiple machines can be linked together connecting topics.

5.2. Relevance

Throughout the course we learned about adding system calls to minix, using mutexes to use shared resources and interprocess communication. Mkafka utilizes a mutex to share the mkafka structure between multiple process. We added a system call to the PM server which pull/pushes messages to the mkafka queue within the kernel.

5.3 Future Work

The main disadvantage of the current iteration of Mkafka is that it is not contained within its own server. Currently Mkafka is within the PM server, which is not the ideal place for it. Ideally we would place Mkafka within its own server. While this is only a functional/performance issue if there are bugs present in Mkafka, it is bad practice to have it within the PM server.

The way consumer groups, and the deletions of messages within a topic are not currently handled in a way that is suitable for most applications. Deleting a message once the queue fills up, may lead to situations where consumers miss out on processing a message. There are many ways to solve this issue, such as allowing for unlimited messages or not allowing for new messages to be added if the queue is full. An optimal solution to this problem would to be supporting all of these methods for removal and allowing for topics to utilize any of these methods.

The mutex that is used currently does not allow for topics to be access in parallel, creating a topic specific mutex would allow for multiple topics to be accessed at the same time, increasing the performance of mkafka when under heavy load. Related to mutexes there is the idea of partitioning topics into subtopics. The advantage of being able to partition topics into subtopics is that it would all for certain services to always consume a specific subset of the messages. It could also allow for topics to be partitioned across multiple machines if they were to grow become too large. An additional mutex related feature that would be a valuable is a blocking pull, this would work the same way as a normal pull request when there are messages in the queue, but when the queue is empty it would wait until a message was added then return the newly added message. This feature is valuable since it allows for processes that constantly poll for new messages to no longer have to, with this they could just use the blocking pull.

A stretch goal that we had was to implement some sort of security. It would be a beneficial to restrict access to topics based on user/group/pid/ip. When the user creates a topic, they could specify how and who is able to produce or consume from a topic. Without any security anyone could access any topic on the machine which makes it unsuitable for many applications.

On reboot of the machine running mkafka data that is currently stored in the messaging bus is lost. In an ideal scenario these messages could be recovered such that after the computer is rebooted. Maintenance of a server running mkafka would be much more complicated if message queues had to be cleared before they could be turned off without losing data.

Contributions of Team Members

Brown, Gordon did the server portion of the project as well as the testing for the server portion
Kuang, Michael assisted with server and networking portion of the project as well as responsible for the final editing of the report
Wojcicki, Krystian did the networking of portion of the project as well as the testing for the networking portion