CHAPTER 5
IMPLEMENTATION AND TESTING

5.1 Implementation

This project uses Golang version 1.15.1 programming language; every 1 second, a batch of data will be sent. This process is carried out for approximately 40 minutes. The data sent is in the form of the time when the data was sent and a list of ticket codes that have been made. Each data set sent will be received by each message broker. After the data is received, the time to receive the data will be calculated and record the performance of the CPU and memory used during the process.

The first thing to do before implementing is to determine how long the program will last. The next thing is to determine how much data to send. Moreover, the last step is to determine the interval the data is sent. The code below shows the time taken during implementation, the amount of data sent during implementation and the time interval for sending the data. The following code snippet is owned Publish code.

```
68.  var dataAmount int
69.  runTime := 40 * time.Minute
70.  intervalTime := 1 * time.Second
71.  fmt.Println("Input Batch Data :")
72.  fmt.Scanln(&dataAmount)
```

The next thing that needs to be done after determining the length of the program running, the amount of data sent, and the time interval in sending the data is to generate data. The data generated from the process is a ticket code. After getting the ticket code data, the next thing to do is save time. This time, which will be stored later, will be a reference for calculating the time required during the data transmission process. Next, another thing that needs to be saved is the ticket code. When the system has received the time data and ticket code, the next thing is to enter the data into a struct to be sent to the message broker.

```
88.  kodeTiket:=fmt.Sprintf("TIKET”+timecreate+"+"+"%06d",i)
```

Above is the code used to create the ticket code on Publish code. The ticket code format was made using when the ticket was created and ended with a counter number. This process is done so that every ticket code generated will always be unique.

After the ticket code generation has been completed, the ticket code will be sent using a message broker. It will be inputted into the database simultaneously. This process can be done because the message broker process is asynchronous, which makes both processes run. This process is carried out to reduce response time so that visitors do not need to wait until the data entry process into the database is completed first.

After the data is sent through the message broker, the system will calculate the performance performed when the message sending process occurs. The calculated performance includes CPU usage, memory usage, and latency. Here is the code to calculate the performance on RabbitMQ Consumer code.

```
78.  timecreate, _ := time.Parse("2006-01-02 15:04:05.000000 MST",
data.TimeCreate)
79.  memory, _ := mem.VirtualMemory()
80.  cpu, _ := cpu.Percent(time.Second, false)
81.  memoryUsage := int(math.Ceil(memory.UsedPercent))
82.  cpuUsage := int(math.Ceil(cpu[0]))
```

The value of CPU usage and memory usage can be directly determined, but not for latency. The way to measure the speed of the latency is to compare when the data is sent and when the data is received. In order to see the difference in latency between the two message brokers, the unit of time to measure latency uses milliseconds..

After getting the three values of CPU usage, memory usage, and latency, the next thing to do is save the data into the database. The database used in this message broker is a different database from that used by the data sender. So that the message broker's database can be used as a backup in the event of a failure to insert the database on the main system, this also proves that message brokers can be used on microservices systems, where microservices have many different databases for each service.

5.2 Testing

The testing that has been done in this research is carried out on each message broker based on each amount of data sent. This test is run for 40 minutes on each message broker and category of the amount of data sent. The data from this study are the average data latency, CPU usage, and memory usage on Redis and RabbitMQ.

1. Latency

The diagram below shows the average latency (ms) performed by Redis. The latency speed is influenced by the amount of data sent at one time. In the diagram below, the average Redis latency for 100 data, 1000 data, and 10000 data has a high difference in average latency..

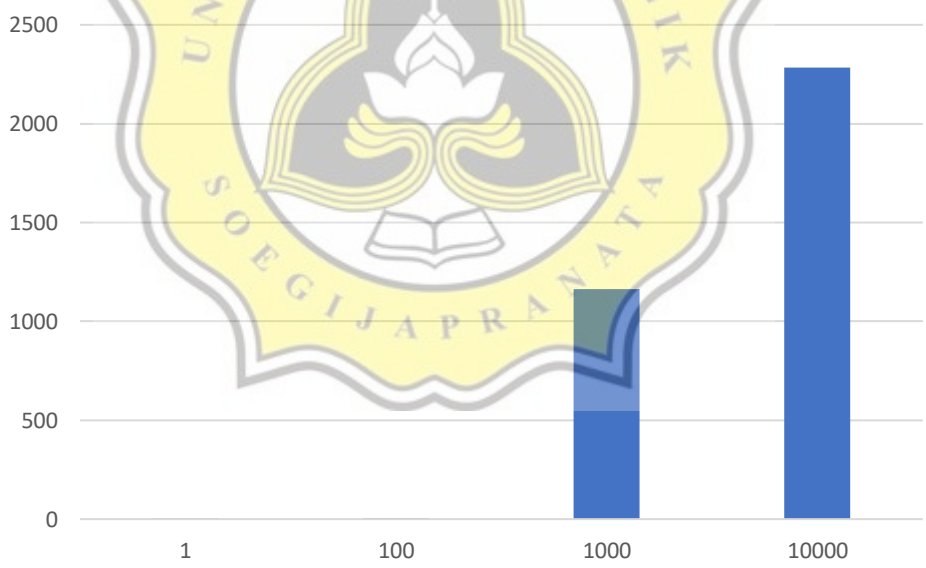

Figure 5.2 : Diagram Average Latency Redis (ms)

The diagram below shows the average latency (ms) performed by RabbitMQ. Similar to Redis, the average latency speed is also affected by the amount of data sent at one time. In addition, the difference in average latency between the number of data has a high graph increase just like Redis' graph.

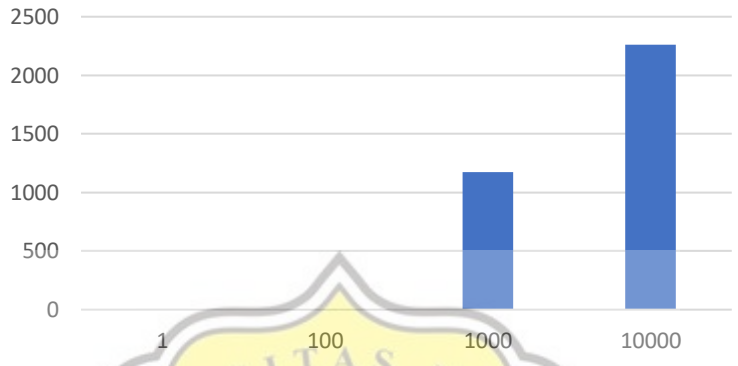

Figure 5.2 : Diagram Average Latency RabbitMQ (ms)

Below is a comparison table of the latency speed between the use of Redis and RabbitMQ message brokers. Redis and RabbitMQ perform nearly the same in terms of latency. Then the data that is sent increases, the comparison of latency speed performance is almost the same. It can be concluded that the use of Redis in terms of latency will be more suitable when the amount of data sent is small. At the same time, RabbitMQ will be more suitable when the amount of data sent is quite large.

Table 5.2: Table Average Latency (ms)

Result	Average Latency (ms)
Redis_1	2.27817015
Redis_100	4.51092833
Redis_1000	1163.00354
Redis_10000	2284.53528
Rabbit_1	2.61295083
Rabbit_100	5.72802958
Rabbit_1000	1173.74358
Rabbit_10000	2262.96858

2. CPU Usage

Below is a diagram of the average CPU usage results using Redis. From the diagram, it can be seen that the amount of data sent also affects CPU performance. The difference in CPU performance is quite significant when the data sent is above 1000 data per second..

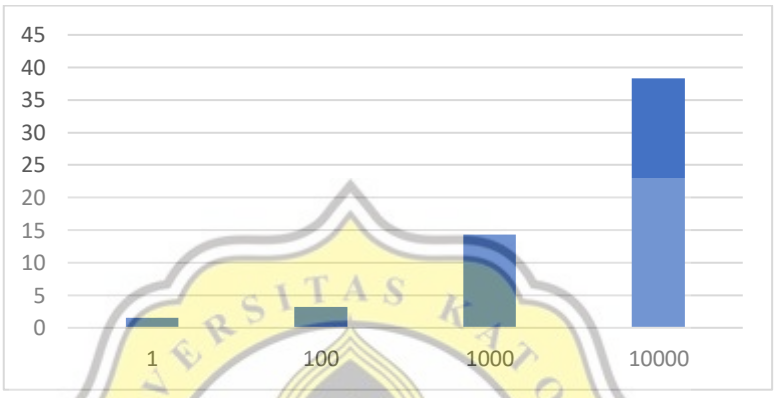

Figure 5.2 : Diagram Average Redis CPU Usage (%)

Below is a diagram of the average CPU usage using RabbitMQ. The use of CPU usage on RabbitMQ with fewer data requires higher performance than RabbitMQ. However, if the data used is more than 1000, RabbitMQ will be more efficient than Redis.

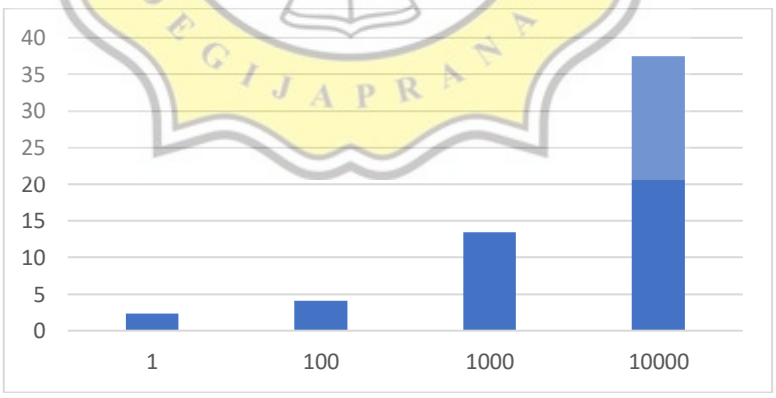

Figure 5.2 : Diagram Average RabbitMQ CPU Usage (%)

The difference in CPU usage between Redis and RabbitMQ can be seen in this table. Redis will use fewer CPU resources compared to RabbitMQ when the data transmitted is small. However, Redis will consume more resources than RabbitMQ if a large amount of power is sent.

Table 5.2: Table Average CPU Usage (%)

Result	Average CPU Usage (%)
Redis_1	1.52508361
Redis_100	3.22240803
Redis_1000	14.3681319
Redis_10000	38.3333333
Rabbit_1	2.35460993
Rabbit_100	4.06199461
Rabbit_1000	13.4197531
Rabbit_10000	37.5

3. Memory Usage

The results of memory usage on Redis can be seen below. The difference in Redis memory usage between under 100 data is not significant. However, when the data sent is more than 1000 data, there will be a fairly high increase in memory..

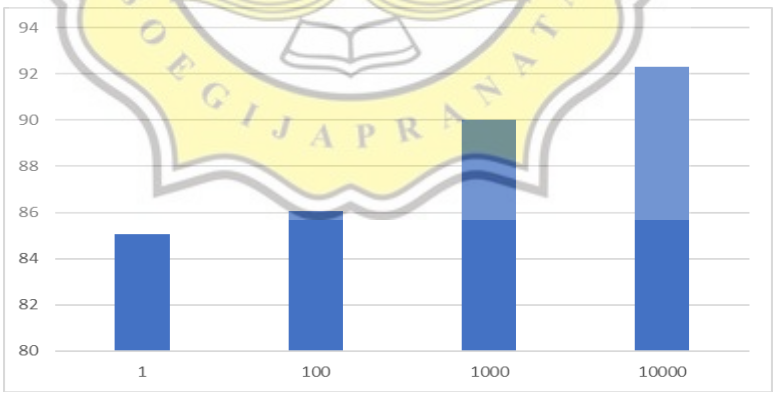

Figure 5.2 : Diagram Average Redis Memory Usage (%)

RabbitMQ memory usage can be seen in the diagram below. The increase in the RabbitMQ memory usage diagram looks quite stable. However, there is a slight spike when the data sent is 10000 data per second.

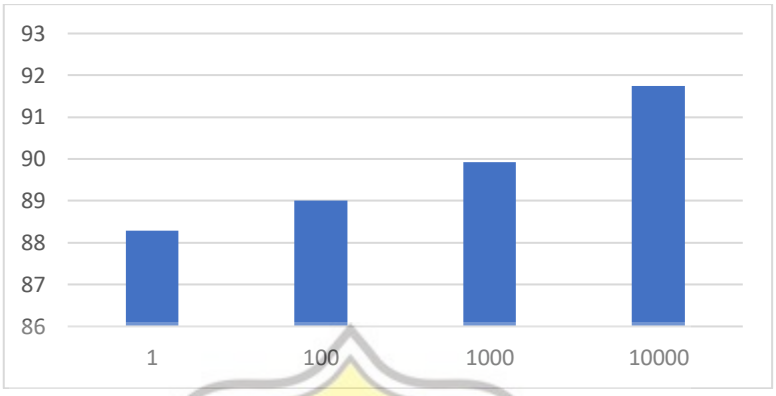

Figure 5.2 : Diagram Average RabbitMQ Memory Usage (%)

Below is the memory usage of two message brokers. RabbitMQ has more stable diagrams compared to Redis. However, Redis is more efficient than RabbitMQ when the data sent is small. In this case, we can see that Redis is more suitable when transmitting small amounts of data, and RabbitMQ is more stable at sending large amounts of data..

Table 5.2: Table Average Memory Usage (%)

Result	Average Memory Usage (%)
Redis_1	85.0568562
Redis_100	86.0685619
Redis_1000	90
Redis_10000	92.2930822
Rabbit_1	88.2907801
Rabbit_100	89
Rabbit_1000	89.9183402
Rabbit_10000	91.7412983

From the experimental results above, we get some comparison results. The results of this comparison can be seen in the table below. The table below shows which message broker is better in each test carried out based on each performance area being compared.

Table 5.2: Table Comparison Each Performance

Amount Data	CPU		Memory		Latency	
	Redis	**RabbitMQ**	**Redis**	**RabbitMQ**	**Redis**	**RabbitMQ**
1	v	-	v	-	v	-
100	v	-	v	-	v	-
1000	-	v	-	v	v	-
10000	-	v	-	v	-	v