CHAPTER 5

IMPLEMENTATION AND RESULT

5.1. Implementation

This project was developed using the Go programming language. In the initial stage or the first stage, an application is built in the form of Client program code to send commands to the Local Server using the gRPC protocol using the Go programming language and an application in the form of Server program code is also built to send a response to the client.

In the client application program that uses the gRPC protocol, import the library "google.golang.org/grpc" to access the gRPC protocol and import the "transaction service" function library to receive the response service that the client will get from the server when it hits the service. of transaction data.

The application developed uses the GO programming language and the gRPC protocol to receive services from a local server, only connecting to a single database. Where the type of database used is a PostgreSQL database.

```
1. import (
2. "google.golang.org/grpc"
3. )
```

Then in the main function, the program code is created to connect the client to the server. Where the connection process is only done once. Also, when the connection to the database fails, the function stops executing the program and returns an error message on the terminal side.

In the GO programming language, a client application that uses the gRPC protocol, the program code declaration is almost the same as in the Java programming language, where the executive function is stored in the main function section. Because of this, the connection process from the client to the server can be done safely and not exposed to the environment outside the application development. Therefore, some of the functions used in the client application are also created separately in several packages.

After successfully connecting to the server, then the server function is declared to the client which performs the process of calling data to the server.

```
4. c := ts.NewTransactionServiceClient(conn)
```

And if the server function declaration process takes too long, the connection will be disconnected. After successfully declaring the server function from the client, the next stage is the calling process from the server, where this process is done by receiving data from the client which is then sent to the server or hit the server, which is done repeatedly using a looping process that has been determined by the server. researcher.

After the data transmission process is successful, then the server will provide a response service using the program code, along with an error response that will be given if the service from the server process fails.

After the development of the application from the client side using the gRPC protocol is completed, the second stage is to build an application from the server side that serves to send services to clients according to requests sent by client applications. Where at this stage it is also still carried out import libraries that function to support the process of the server application in the form of libraries "google.golang.org/grpc" and "transaksiRepository". The library "google.golang.org/grpc" is used for service delivery using the gRPC protocol and "transaction repository" is used to provide access between server and client to the PostgreSQL database.

```
5.  import (
6.  "google.golang.org/grpc"
7.  "project/transactions-service-
    grpc/transaksiServer/transaksiRepository"
8.  )
```

After the data import is done on the main server function, then the port variable declaration process is carried out for the server to be run and calls the protobuf function to run on the server. Then, after the function is successfully executed on the server side, the client will then make a service call using the inquiry function, which at this stage begins with the declaration of the response for the function, and continues with the process to get the value of 'cpuUsage' and 'memoryUsage'.

```
9.  var result ts.ResponseInquiryTransaction
10. memory, _ := mem.VirtualMemory()
11. cpu, _ := cpu.Percent(time.Second, false)
12. memoryUsage := int(math.Ceil(memory.UsedPercent))
13. cpuUsage := int(math.Ceil(cpu[0]))
```

The next stage is to perform the variable values that will be sent from the client to be entered into the transaction table. Service data sent from the server to the client is calculated in units of time milliseconds (ms) using the time start variable. Service data that is successfully sent from the server to the client will be marked as a success message, which is sent to the InquiryTransaction function.

The process of sending requests from the client to the server and the process of service data from the client-server is carried out in several trials or hits, where the process is carried out using the JMeter measuring instrument with the results shown in ANNEX 1.

In the third stage, an application is built in the form of Client program code to send commands to the Local Server using the Rest API protocol using the Go programming language, but the Rest API does not build Server program code to send responses to the client, but program commands can be executed directly through the server using the " net/HTTP".

The application developed for the Rest API protocol is the same as gRPC, which uses the Go programming language and is used to receive services from a local server and continues with a connection to a single database. Where the type of database used is a PostgreSQL database.

```
68.  import (
69.  "encoding/json"
```

```
70.  "net/http"
71.  )
```

In the GO programming language, a client application that uses the Rest API protocol, the functions to be executed are stored in the main function section. Because the execution command has been built directly on the server, there is no need to connect to the server but can directly process file input at the request stage.

```
72.  fmt.Println("Input Batch Data :")
73.  fmt.Scanln(&dataAmount)
```

The next stage or the fourth stage is the process of calling data from the server, where this process is done by receiving data from the client which is then sent to the server or hit the server, which is done repeatedly using a looping process that has been determined by the researcher.

After the data transmission process is successful, then the server will provide a response service using the program code, along with an error response that will be given if the service from the server process fails when using the Rest API protocol.

The request or server hit process for each gRPC protocol and Rest API was carried out in three trials, wherein in the first experiment 10 server hits were carried out, in the second experiment 100 server hits were carried out, and the last stage was 1000 server hits. for each of these protocols.

After the application is complete, several testing conditions are carried out on applications that use the Rest API and gRPC protocols. After the application has been built, then the experiment is carried out through several stages.

a) On Local Server in Computer A

The first experiment was carried out by making server requests from both applications using the Rest API and gRPC protocols for 10 repetitions or hitting the server using the GET and POST methods. In the first experiment, the request command was performed using Apache JMeter, so the process could be carried out without the need for direct human assistance. After testing the two applications using the Rest API and gRPC protocols, the results are shown in Table 5.1.

Table 5.1. Testing Data 10 gRPC and RestAPI Computer A

Fitur	Data Average Get and Post Method (10 Request)			
	gRPC		RestAPI	
	Get	Post	Get	Post
CPU Usage	36.2	14.4	36.97	16.1
Memori Usage	82	72.9	83	74.3
TTL	1014.9	1009.5	1018.6	1018.3

After the first experiment was completed, the second experiment was carried out by requesting a server from both applications using the Rest API and gRPC protocols for 100 repetitions or hitting the server using the GET and POST methods. In the second experiment, the request command was carried out using Apache JMeter, so that the process could be carried out without the need for direct human assistance. After testing the two applications using the Rest API and gRPC protocols, the results are shown in Table 5.2.

Table 5.2. Testing Data 100 gRPC and RestAPI Computer A

Fitur	Data Average Get and Post Method (100 Request)			
	gRPC		RestAPI	
	Get	Post	Get	Post
CPU Usage	31.95	23.99	40.2	27.92
Memori Usage	84	74.98	84	76
TTL	1015.32	1010.99	1038.52	1011.69

After the second experiment was completed, the third experiment was carried out by requesting the server from the two applications using the Rest API and gRPC protocols for a thousand repetitions or hitting the server using the GET and POST methods. In the third experiment, the request command was carried out using Apache JMeter, so that the process could be carried out without the need for direct human assistance. After experimenting with both applications using the Rest API and gRPC protocols, the results are shown in Table 5.3 as follows.

Table 5.3. Testing Data 1000 gRPC and RestAPI Computer A

Fitur	Data Average Get and Post Method (1000 Request)			
	gRPC		RestAPI	
	Get	Post	Get	Post
CPU Usage	63	54	65	58
Memori Usage	85.1	76	85.9	77
TTL	1532.17	1169.298	1608.95	1557.999

b) On Local Server on Computer B

The first experiment was carried out by making server requests from both applications using the Rest API and gRPC protocols for 10 repetitions or hitting the server using the GET and POST methods. In the first experiment, the request command was performed using Apache JMeter, so the process could be carried out without the need for direct human assistance. After experimenting with both applications that use the Rest API and gRPC protocols, the results are shown in table 5.4.

Table 5.4. Testing Data 10 gRPC and RestAPI Computer B

Fitur	Data Average Get and Post Method (10 Request)			
	gRPC		RestAPI	
	Get	Post	Get	Post
CPU Usage	66.7	57	71.6	61
Memori Usage	52	54	61.5	54
TTL	1018.8	1015.6	1020.4	1043.1

After the first experiment was completed, the second experiment was carried out by requesting a server from both applications using the Rest API and gRPC protocols for 100 repetitions or hitting the server using the GET and POST methods. In the second experiment, the request command was carried out using Apache JMeter so that the process could be carried out without the need for direct human assistance. After testing the two applications using the Rest API and gRPC protocols, the results are shown in Table 5.5.

Table 5.5. Testing Data 100 gRPC and RestAPI Computer B

Fitur	Data Average Get and Post Method (100 Request)			
	gRPC		RestAPI	
	Get	Post	Get	Post
CPU Usage	99.13	92.55	99,735	99.62
Memori Usage	53	55	63.75	55
TTL	1449.03	1325.37	1455.73	1422.54

After the second experiment was completed, the third experiment was carried out by requesting the server from the two applications using the Rest API and gRPC protocols for a thousand repetitions or hitting the server using the GET and POST methods. In the third experiment, the request command was carried out using Apache JMeter so that the process could be carried out without the need for direct human assistance. After experimenting with both applications using the Rest API and gRPC protocols, the results are shown in Table 5.6 as follows

Table 5.6. Testing Data 1000 gRPC and RestAPI Computer B

Fitur	Data Average Get and Post Method (1000 Request)			
	gRPC		RestAPI	
	Get	Post	Get	Post
CPU Usage	99.486	99.703	100	100
Memori Usage	58.695	56,1	65.288	56.929
TTL	5461.308	4586.195	5711.028	5296.988

5.2. RESULT

Testing carried out in this study has results that depend on the amount of data sent. This test was carried out in three tests, where the first test carried out 10 hits, the second test carried out 100 hits, and the third test carried out 1000 hits. The data obtained from the test results are in the form of CPU usage, memory usage, and latency time.

1.a. Local Server Latency Time In Computer A

Figure 5.1 below shows the average value of latency time (ms) generated by applications that request data using the gRPC and RestAPI protocol. The speed of the latency time is affected by the amount of data sent in one test. Figure 5.1 shows an image of the graph of the average latency time for three tests, with 10 data, 100 data, and 1000 data sent each.

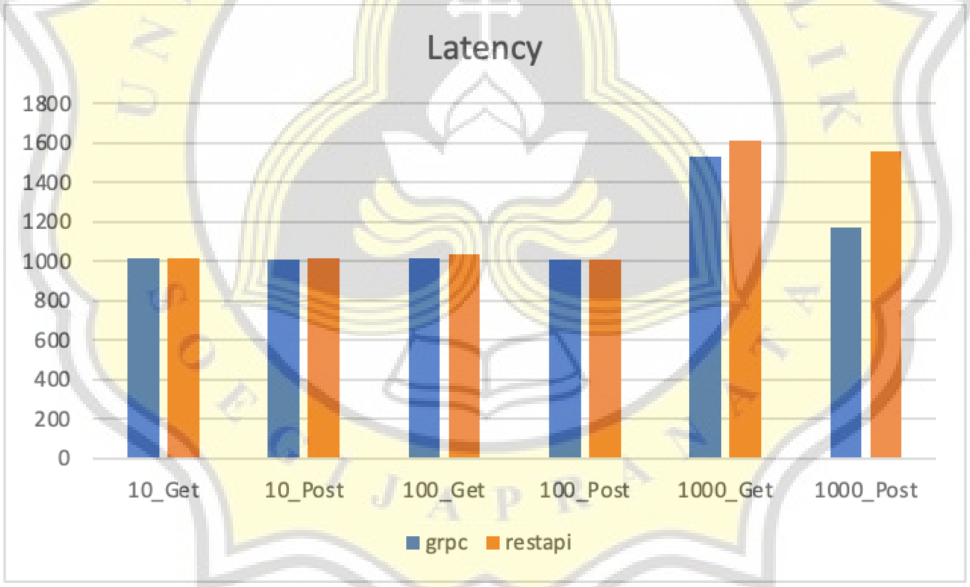

Figure 5.1.Latency Time gRPC and RestAPI Computer A

A comparison of latency time speed performance between gRPC and RestAPI protocols when requesting data is shown in table 5.7. The gRPc and RestAPI protocols show an increasingly large time difference, which depends on the amount of data sent. From table 5.7, it can also be seen that the RestAPI protocol will be more suitable if it is used to transmit small amounts of data, while the gRPC protocol will be more suitable if it is used for sending larger amounts of data.

Table 5.7. Average Latency Time Computer A

Data Sent	gRPC	RestAPI
10_Get	1014.90	1018.60
10_Post	1009.50	1018.30
100_Get	1015.32	1038.52
100_Post	1010.99	1011.69
1000_Get	1532.17	1608.95
1000_Post	1169.30	1557.99
STDEV	208.92	290.65

1.b. Local Server Latency Time Computer B

Figure 5.2 below shows the average value of latency time (ms) generated by applications that request data using the gRPC and restAPI protocol. The speed of the latency time is affected by the amount of data sent in one test. Figure 5.2 shows an image of the graph of the average latency time for three tests, with 10 data, 100 data, and 1000 data sending each.

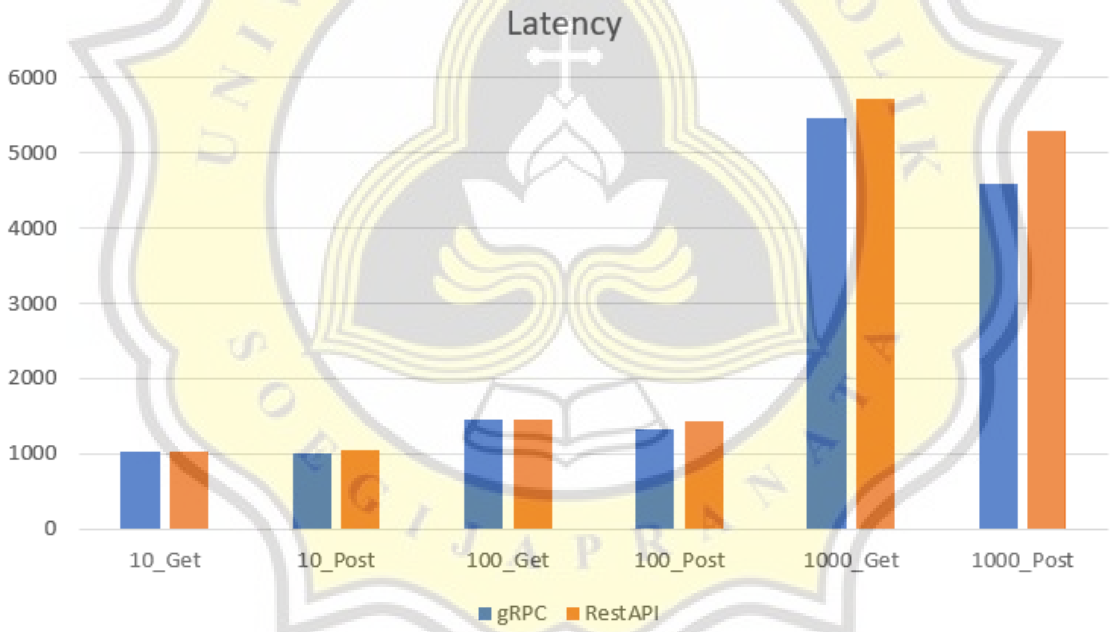

Figure 5.2 Latency Time gRPC and RestAPI Computer B

The comparison of latency time speed performance between gRPC and RestAPI protocols when requesting data is shown in table 5.8. The gRPc and RestAPI protocols show a larger time difference, which depends on the amount of data sent. From table 5.8, it can also be seen that the RestAPI protocol will be more suitable if it is used for sending small amounts of data, while the gRPC protocol will be more suitable if it is used for sending larger amounts of data.

Table 5.8. Average Latency Time Computer B

Data Sent	gRPC	RestAPI
10_Get	1018.8	1020.4
10_Post	1015.6	1043.1
100_Get	1449.03	1455.73
100_Post	1325.37	1422.54
1000_Get	5461.308	5711.028
1000_Post	4586.195	5296.988
STDEV	1999.99	2215.70

2.a. CPU Usage Computer A

Figure 5.3 below shows the average value of CPU Usage (bytes) generated by applications that request data using the gRPC and RestAPI protocol. CPU Usage value is affected by the amount of data sent in one test. Figure 5.3 shows a graph of the average CPU Usage value for three tests, with 10 data, 100 data, and 1000 data sent each.

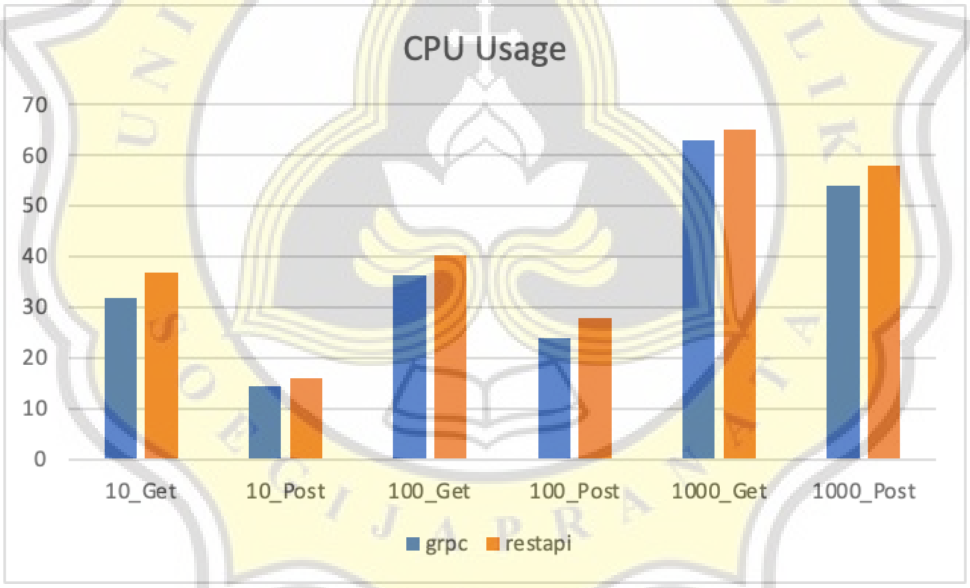

Figure 5.3 CPU Usage gRPC and RestAPI Computer A

The comparison of CPU Usage value performance between gRPC and RestAPI protocols when requesting data is shown in table 5.9. The gRPc and RestAPI protocols show a relatively large CPU Usage value, which depends on the amount of data sent. From table 5.9, it can also be seen that the RestAPI protocol will be more suitable if it is used to transmit small amounts of data, while the gRPC protocol will be more suitable if it is used for sending larger amounts of data.

Table 5.9. Average CPU Usage Computer A

Data Sent	gRPC	RestAPI
10_Get	31.95	36,97
10_Post	14.4	16.1
100_Get	36.2	40,2
100_Post	23.99	27.92
1000_Get	63	65
1000_Post	54	58
STDEV	18.28	18.29

2.b. CPU Usage Computer B

Figure 5.4 below shows the average value of CPU Usage (bytes) generated by applications that request data using the gRPC and RestAPI protocol. CPU Usage value is affected by the amount of data sent in one test. Figure 5.4 shows a graph of the average CPU Usage value for three tests, with 10 data, 100 data, and 1000 data sent each.

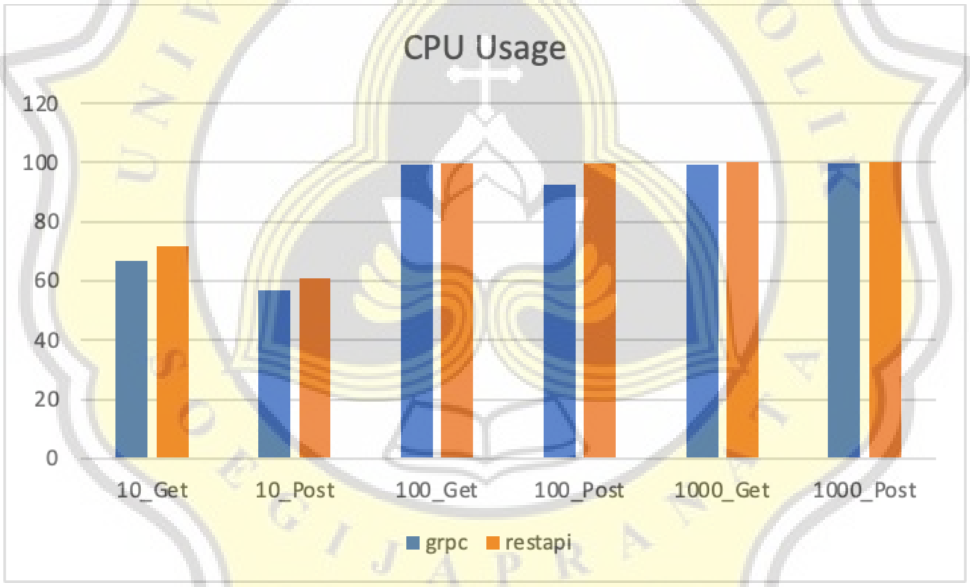

Figure 5.4 CPU Usage gRPC and RestAPI Computer B

The comparison of CPU Usage value performance between gRPC and RestAPI protocols when requesting data is shown in table 5.10. The gRPc and RestAPI protocols show relatively large CPU Usage values, which depend on the amount of data sent. From table 5.10, it can also be seen that the RestAPI protocol will be more suitable if it is used to transmit small amounts of data, while the gRPC protocol will be more suitable if it is used for sending larger amounts of data.

Table 5.10. Average CPU Usage Computer B

Data Sent	gRPC	RestAPI
10_Get	66.7	71.6
10_Post	57	61
100_Get	99.13	99,735
100_Post	92.55	99.62
1000_Get	99.486	100
1000_Post	99.703	100
STDEV	18.96	17.64

3.a. Memory Usage Computer A

Figure 5.5 below shows the average value of Memory Usage (bytes) generated by applications that request data using the gRPC and RestAPI protocol. Figure 5.5 shows a graph of the average value of Memory Usage for three tests, with 10 data, 100 data, and 1000 data sent each.

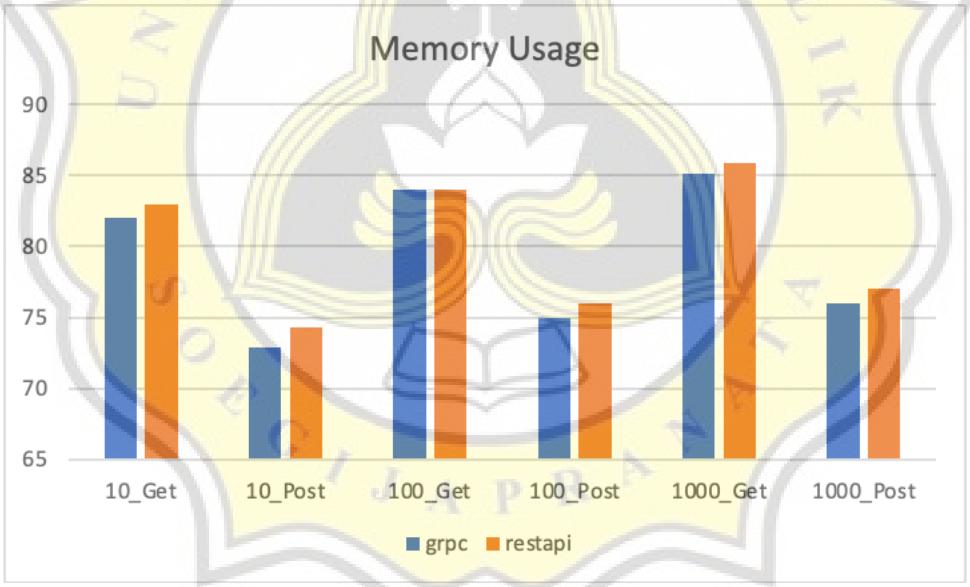

Figure 5.5 Memory Usage gRPC and RestAPI Computer A

The comparison of Memory Usage value performance between gRPC and RestAPI protocols when requesting data is shown in table 5.11. The gRPc and RestAPI protocols show Memory Usage values that are relatively the same value, which depends on the amount of data sent. From table 5.11, it can also be seen that the RestAPI and gRPC protocols are equally suitable for sending any amount of data with a relatively small and negligible difference in the Memory Usage value.

Table 5.11. Average Memory Usage Computer A

Data Sent	gRPC	RestAPI
10_Get	82	83
10_Post	72.9	74.3
100_Get	84	84
100_Post	74.98	76
1000_Get	85.1	85.9
1000_Post	76	77
STDEV	5.17	4.8

3.b. Memory Usage Computer B

Figure 5.6 below shows the average value of Memory Usage (bytes) generated by applications that request data using the gRPC protocol. In Figure 5.6, a graph of the average value of Memory Usage is shown for three tests, with 10 data, 100 data, and 1000 data sent each.

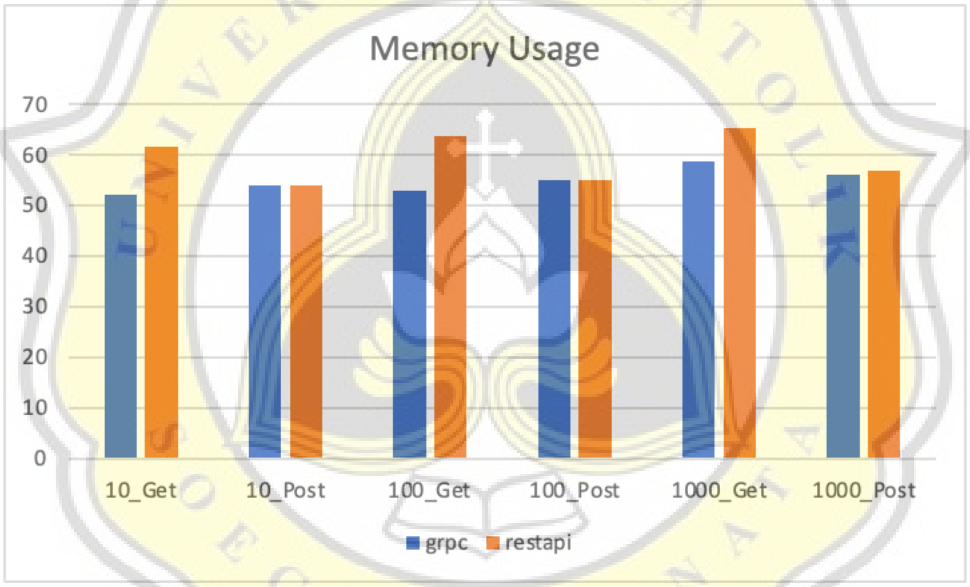

Figure 5.6 Memory Usage gRPC and RestAPI Computer B

The comparison of Memory Usage value performance between gRPC and RestAPI protocols when requesting data is shown in table 5.12. The gRPc and RestAPI protocols show Memory Usage values that are relatively the same value, which depends on the amount of data sent. From table 5.12, it can also be seen that the RestAPI and gRPC protocols are equally suitable to be used to transmit any amount of data with a relatively small and negligible difference in the Memory Usage value.

Table 5.12. Average Memory Usage Computer B

Data Sent	gRPC	RestAPI
10_Get	52	61.5
10_Post	54	54
100_Get	53	63.75
100_Post	55	55
1000_Get	58.695	65.288
1000_Post	56,1	56.929
STDEV	2.39	4.74

From the results of the tests carried out previously, several overall performance comparison results were obtained which are shown in table 7. From the table, it is shown which protocol has a better performance value when compared to other protocol values.

Table 5.13. Performance Comparison Latency

Data Sent	Latency Time							
	gRPC				RestAPI			
	Computer A		Computer B		Computer A		Computer B	
	Get	Post	Get	Post	Get	Post	Get	Post
10	V	v	v	v	-	-	-	-
100	V	v	v	V	-	-	-	-
1000	V	v	v	v	-	-	-	-

Table 5.14. Performance Comparison Memory Usage

Data Sent	Memory Usage							
	gRPC				RestAPI			
	Computer A		Computer B		Computer A		Computer B	
	Get	Post	Get	Post	Get	Post	Get	Post
10	-	v	v	-	-	-	-	-
100	-	v	v	-	-	-	-	-
1000	V	v	v	v	-	-	-	-

Table 5.15. Performance Comparison CPU Usage

Data Sent	CPU Usage							
	gRPC				RestAPI			
	Computer A		Computer B		Computer A		Computer B	
	Get	Post	Get	Post	Get	Post	Get	Post
10	v	v	v	v	-	-	-	-
100	v	v	v	v	-	-	-	-
1000	v	v	v	v	-	-	-	-

According to Johnston [11] service activities are activities that involve contact and interaction that generally occur in "real time", so that in service applications the communication process that occurs is synchronous communication. Because the communication process runs synchronously, the best performance of the service transaction process can be achieved optimally if the latency time is low. results Based on the research conducted, gRPC has a lower latency value than RestAPI in service transaction activities. So that gRPC is the right choice for service transaction activities when compared to restAPI, because gRPC uses the HTTP2 protocol, while restAPI is built using HTTP1.1.