Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мережних технологій факультету інформатики

“Scaling SignalR WebSocket Real-Time Applications”

Текстова частина до курсової роботи
за спеціальністю „Комп’ютерні науки”-122

Керівник курсової роботи:
док.техн.наук, доц. Глибовець А.М.

(підпис)
“_” _____ 2021 року

Виконала: студентка КН-1
Діденко В. О.

(підпис)
“_” _____ 2021 року

Київ 2021

Ministry of Education and Science of Ukraine
NATIONAL UNIVERSITY OF "KYIV-MOHYLA ACADEMY"
Network Technologies Department of the Faculty of Informatics

APPROVED
Head of the Network Technologies Department
associate professor, doctor of mathematics
_____ G.I. Malaschonok
(signature)
"____" _____ 2020 year.

INDIVIDUAL TASK
For the course work
For 1-year Master Degree student of the Faculty of Informatics

TOPIC: Scaling SignalR WebSocket Real-Time Applications

Output data:

Text part content of coursework:

Individual task

Calendar plan

Abstract

Introduction

Section 1: WebSocket SignalR Scaling Caveats

Section 2: Introducing Backplanes

Section 3: Scaling with Azure SignalR Service

Conclusion

References

Issue date "____" _____ 2021 year

Supervisor _____*(signature)*

Task received _____*(signature)*

CALENDAR PLAN

Theme: Scaling SignalR WebSocket Real-Time Applications

#	Stage Name	Deadline	Note
1	Acquiring coursework topic	01.10.2020	
2	Finding appropriate literature	10.11.2020	
3	Researching WebSocket scaling approaches	10.12.2020	
4	Researching load balancing and sticky sessions	27.12.2020	
5	Exploring Backplane approach	27.01.2021	
6	Exploring Azure SignalR Service	08.02.2021	
7	Researching Redis Backplane	07.03.2021	
8	Researching Azure SignalR Service Performance	15.03.2021	
9	Outline of the coursework	16.03.2021	
10	Coursework analysis with the Supervisor	16.03.2021	
11	Application implementation	28.03.2021	
12	Coursework main section and conclusion	10.04.2021	
13	Coursework analysis with the Supervisor	10.04.2021	
14	Coursework improvement	13.04.2021	
15	Creation of the presentation	13.04.2021	
16	Coursework analysis with the Supervisor	18.04.2021	
17	Handing in the coursework	11.05.2021	

Student: Didenko V.O. “____” _____

Supervisor: Glybovets A.M. “____” _____

TABLE OF CONTENTS

TABLE OF FIGURES

ABSTRACT

Real-time applications depend on persistent connections in order to provide users with high frequency data updates from the application server. The idea behind persistent connections is that when a connection is established it is kept open, hence optimizing the data transfer process by saving time on establishing a new connection. As the number of continuous connections grows in a high-traffic application sustaining a high number of clients, eventually the server can run out of connection resources. In this research work the aim is to scale the persistent connections in order to limit the number of open connections that a single application server has to handle; therefore, designing real-time applications that can serve many clients in an efficient manner. This study introduces WebSocket scaling techniques, focusing on the Azure SignalR Service as the solution for scaling data-intensive applications.

INTRODUCTION

There is a demand for data-intensive applications to provide real-time up-to-date information without end-users having to request a data refresh. [1] In addition, real-time applications are expected to be performance efficient and respond to user interactions with minimal delay in order to provide a smooth user experience. Therefore, applications delivering real-time data access require high frequency updates from the application server and to do so persistent connections between the server and the application are established. [2]

Persistent connections incorporate the idea of using the same TCP connection for managing multiple HTTP requests/responses, instead of opening a new connection for each request/response pair, therefore reducing latency since the time spent on the handshake process for establishing a new connection is saved. Persistent connections stay open and in a high-traffic application serving many clients, kept-alive connections can cause servers to reach the maximum number of connections that they can handle and cause servers to overload, therefore, leading to a poor user experience. [2]

The SignalR [3] (an open-source library for providing real-time web functionality) application load testing experiment that was conducted at the *Thousands of concurrent connections with Azure SignalR Service* NDC Conference hosted by Nelly Sattari and Stafford Williams [2][4] revealed that (using the Cranker [5] load testing tool from a local machine) a real-time SignalR application hosted on S1 App Service [6] could serve at maximum

768 concurrent connections with performance drawbacks. [2] Expanding from a local machine to 50 docker containers [7] and pointing them at the SignalR application hosted on S1 App Service helped to increase the limit to 16000 concurrent connections at which the application was not stable and performance was dropping [2][4].

In this work the aim is to scale the persistent connections in order to limit the number of concurrent connections that a single application server has to handle, hence expanding the system's capability to endure increased load as the number of concurrent client connections grows. Ensuring higher scalability in turn secures a smooth user experience in real-time applications that serve a significant amount of traffic. [1] This work focuses on SignalR Websocket applications: how they can be scaled out to serve many clients in a performance efficient manner.

This research work is organized into three sections. The first section covers scaling problems that occur when the application that depends on persistent connections (using WebSockets for the data transfer method) is scaled up and out. The second section introduces backplanes and how they are used to solve scaling issues of WebSocket applications. The third section investigates the Azure SignalR Service [8] solution for highly scalable WebSocket applications and, based on typical use case scenarios, evaluates the performance factors that impact the Azure SignalR Service capacity by implementing a performance evaluation tool.

Section 1: WebSocket SignalR Scaling Caveats

1.1 Scaling up and out

There are two main approaches to scaling a web-based application: scaling up and scaling out. An application is scaled up by increasing the amount of and enhancing the available resources, switching to a larger server or a more extensive virtual machine with more RAM, CPU, and specifications. However, as the number of connections grows, eventually, the application will reach the limit and have to scale out - also referred to as horizontal scaling - and that means adding more servers to handle the load. [1][2]

In horizontal scaling, it is essential to balance out the load among the available instances of the application to prevent critical overload of a specific instance. For example, when an application is scaled out and made available on multiple servers if the load is not evenly distributed, requests may mostly be routed to a particular server even if other servers are available, causing that server to overload, resulting in a poor end-user experience. *Figure 1: Server overloaded due to load not being balanced out* illustrates a possible scenario if the load is not correctly distributed among the instances of the application. [1][2][9][10][12]

To solve the issue of incoming client connections overloading a certain server, a load balancer is introduced to balance out the load among all available servers. Such an approach helps to prevent client connections from hitting a certain server when other servers are available, therefore preventing

a certain instance of critically overloading. This is illustrated in *Figure 2: Introducing a load balancer.* [1][2][9][10][12]

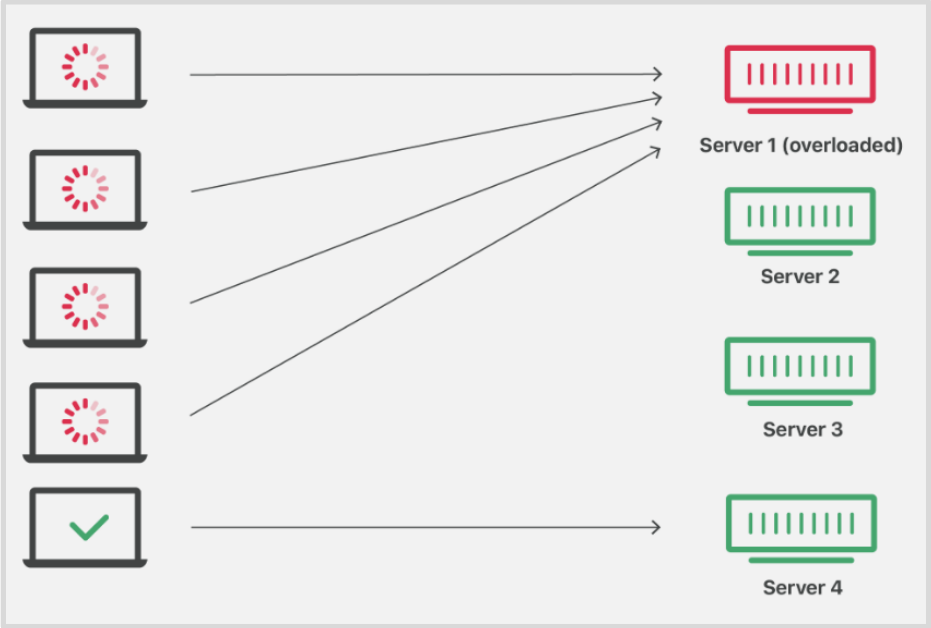

Figure 1: Server overloaded due to load not being balanced out

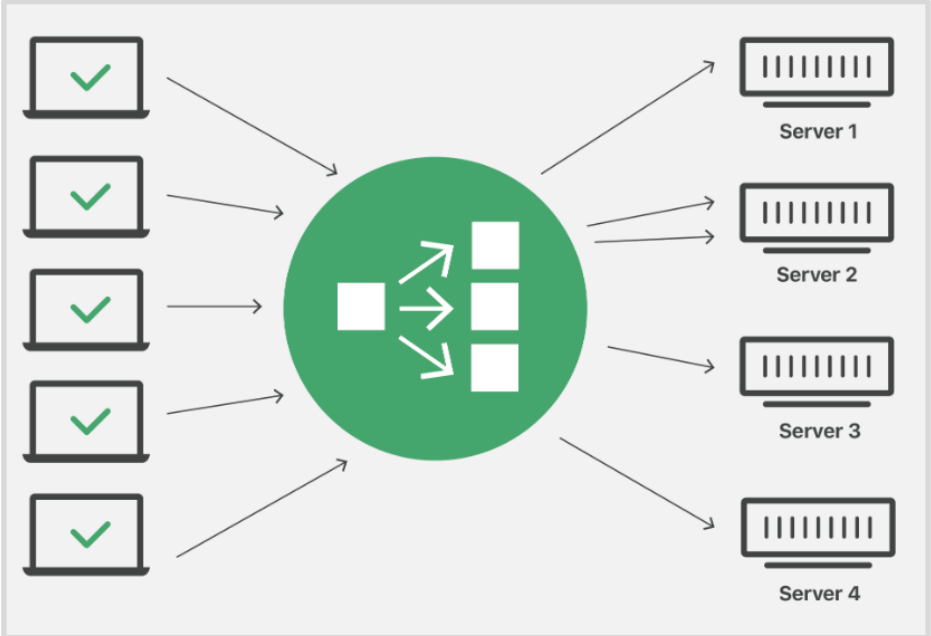

Figure 2: Introducing a load balancer

In addition, sticky sessions [13] should be implemented. In SignalR, WebSockets are chosen as the data transfer method by default. Nevertheless, if a WebSocket connection cannot get established, SignalR switches to other means of persistent communication such as Server-Sent-Events and Long Polling. Since in the latter data transfer method the connection between the client and the server needs to be reopened for every request/response pair, a previously connected client upon initiating a new connection request could, through the load balancer, get connected to a different server instead of the server that was previously processing the client's request - causing an issue in the application. [13][14][15]

For example, the client, through the load balancer, requests Server 1 to prepare a specific order Order 1. As a result, Server 1 starts processing the specific order for the client. When the client sends a polling request, the load balancer could assign the polling request to a different server, for example to Server 2, which is not aware about Order 1. This applies to Server-Sent-Events as well since the HTTP connection could get dropped: as the connection gets restored by the EventSource, in the same way the load balancer could forward the connection request to a different instance of the scaled out application. [13][14][15]

To prevent such a scenario from occurring, sticky sessions are introduced. Sticky sessions help to ensure that the load balancer will assign the client connection to the same server that processed the client's previous request. A possible flow with sticky sessions is that the load balancer will set a cookie in the browser for tracking which server the client got connected to;

based on the cookie, the load balancer will then assign subsequent requests to the same server. [13][14][15]

1.2 Syncing clients between application instances

By design, .NET Core SignalR needs to keep track of the connected clients in each process, making it essential for each SignalR instance in the scaled-out application to be informed about which clients are connecting and disconnecting. This imposes a problem in the load balancer design which was introduced in the previous section 1.1 since there is no connection between the SignalR instances that would allow showing which clients are connecting and disconnecting. [2][8][15][19][21][22][26][27][28]

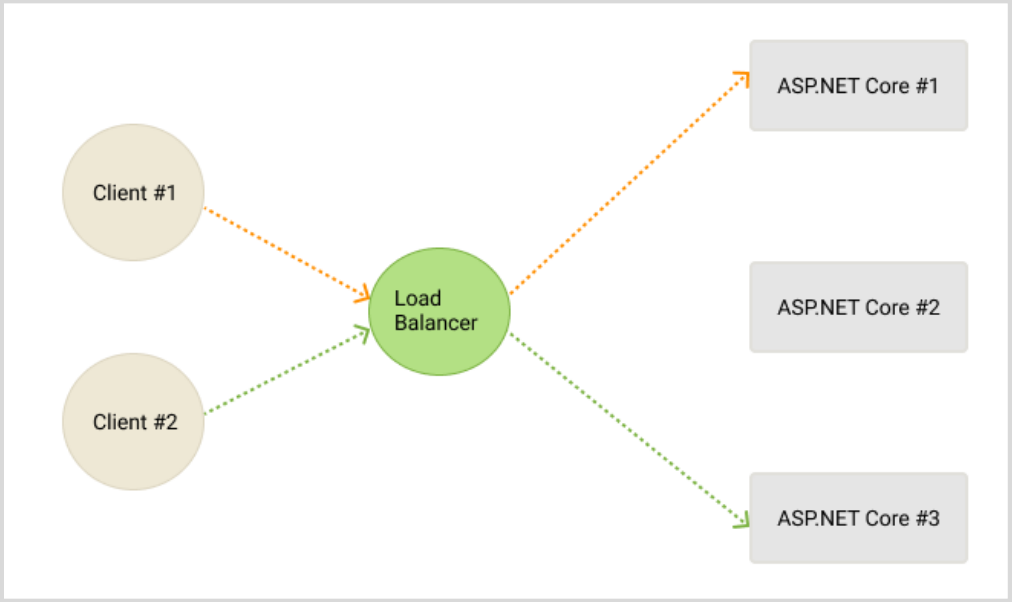

Figure 3: No connection between the application instances

Figure 3: No connection between the application instances illustrates that there are three instances of the .NET Core SignalR application behind a load balancer. It is demonstrated that when Client #1 initiates a SignalR connection, it connects to the first instance through the load balancer. It is possible that the next client who connects to the application, for example, Client #2, could be forwarded to a different instance of the application, for

example, the third one. Since no means of communication is established between the instances of the application, both instances are unaware of the other connected clients. In consequence, messages sent from the Server Hub or HubContext will be sent only to the clients that are connected to that instance of the application instead of being sent to all connected clients. [2][8][15][19][21][26][27]

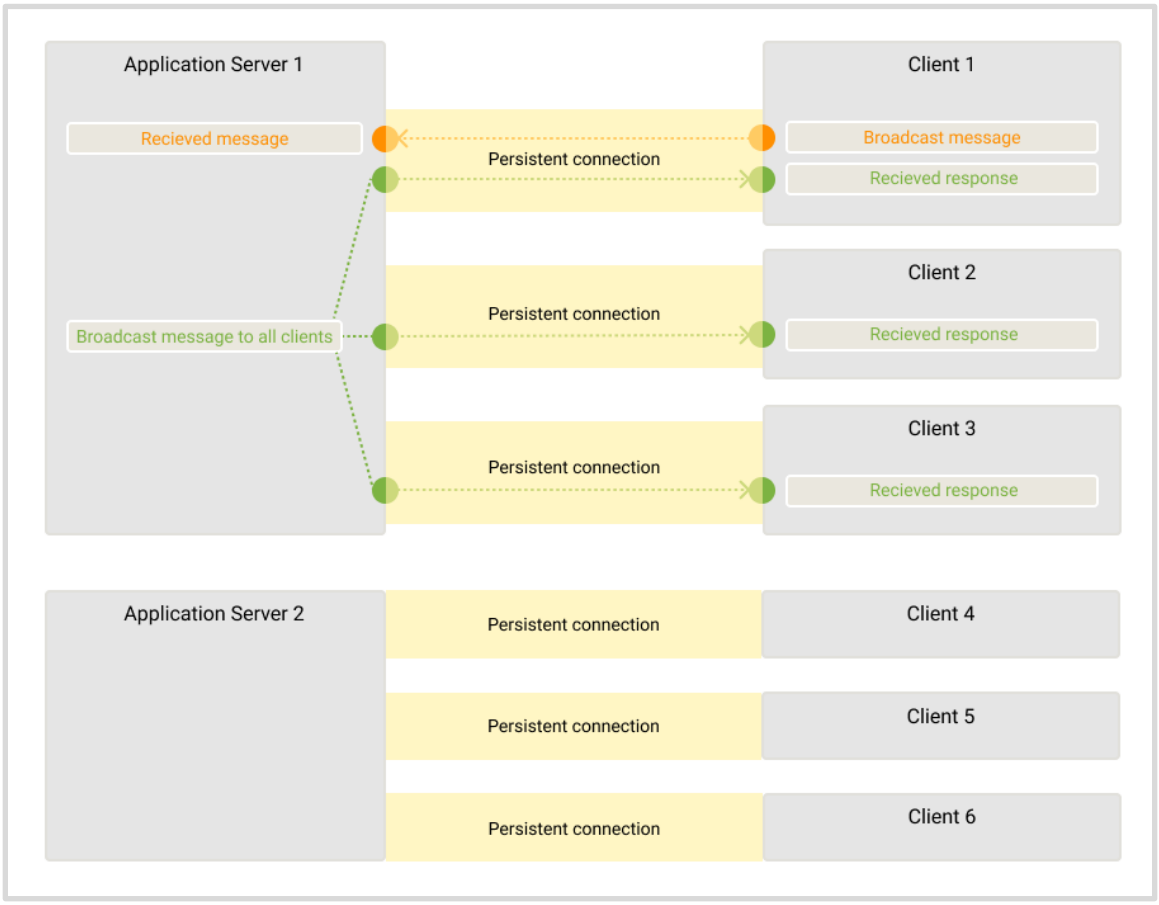

Figure 4: Broadcast issue caused by no communication between instances

For example, for collaborative clients who want to broadcast a message to all the other clients, no communication between the application instances becomes an issue. An example of such a scenario is illustrated in *Figure 4: Broadcast issue caused by no communication between instances.*

In the diagram, Client 1 sends a message to Server 1 to broadcast to the rest of the clients and Server 1 broadcasts the message to all the connected clients. However, Server 1 is not aware of Server 2 as well as the clients that are connected to Server 2, that is Client 4, Client 5, and Client 6 - as a result, only the clients that are connected to Server 1 receive the message, that is Client 1, Client 2, and Client 3. Therefore, a communication layer needs to be established between the instances of the application. The general idea is illustrated in the diagram below in *Figure 5: Establishing a communication layer for the application instances.* [2][8][15][19][21][22][26][27][28]

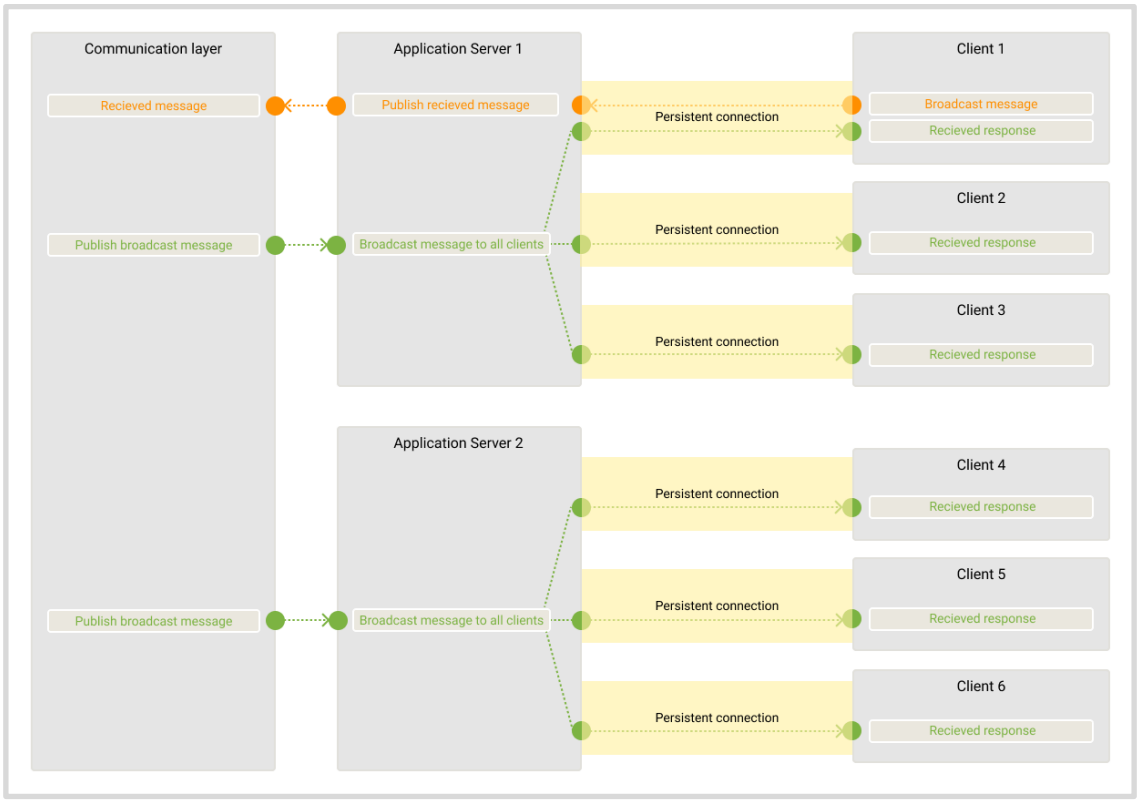

Figure 5: Establishing a communication layer for the application instances

Section 2: Introducing Backplanes

2.1 Backplanes as a communication layer

To solve the communication gap issue between the instances of a horizontally scaled SignalR application, which is described in section 1.2, the instances need to connect to a shared communication layer, which is called a Backplane. [19]

Each of the SignalR application instances connects to the Backplane by subscribing to it. If a certain instance of the application receives any new messages or client connections, it publishes the received data to the Backplane and the Backplane broadcasts the updates to all subscribed instances of the application. [18][19][20][22][26][27][28]

At the same time, whenever an update is available on a certain application instance, the instance publishes the update to the Backplane and the Backplane informs the rest of the instances about the update so that the rest of the instances could forward the update to their connected clients. In *Figure 6: Backplane as a communication layer* it is shown how the application instance Server 1 publishes the message “Hello” to the Backplane; the message is then forwarded to all the subscribers (the application instances - Server 1, Server 2, and Server 3) and each application instance sends the update to its connected clients. As a result, all clients receive the message “Hello” from Server 1 even if they are connected to another instance, for example Client #3, Client #4, and Client #5. It is important to note that without a communication layer being

established between the application instances (Server 1, Server 2, and Server 3) only the clients connected to Server 1 (Client #1 and Client #2) would have received the message “Hello”. [18][19][20][22][26][27][28]

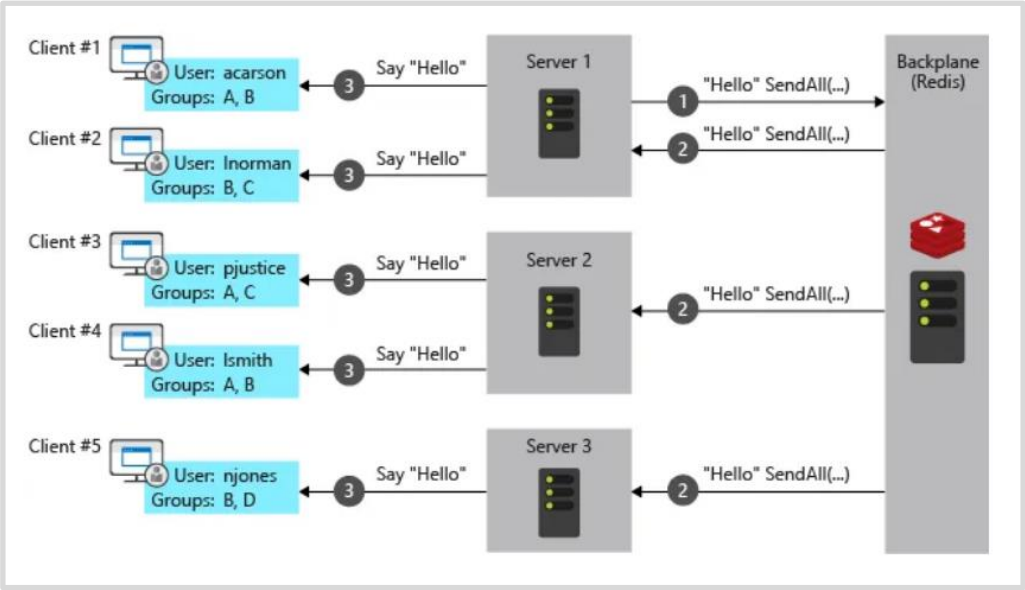

Figure 6: Backplane as a communication layer

With the help of the Backplane, a communication layer is established through which the SignalR instances can communicate with one another and share any received updates. In addition, all connected clients will be able to receive the same updates regardless of which instance of the application they get connected to, hence, solving the communication gap issue.

To create a Backplane, a service providing a subscribe and publish messaging pattern is used; for example, a Backplane can be implemented with Redis [16] - an in-memory key-value store that provides a subscribe and publish API, also referred to as Pub/Sub [16]. The Pub/Sub API implements

the Publish/Subscribe messaging paradigm and provides the SUBSCRIBE, UNSUBSCRIBE, and PUBLISH commands. [16]

```
1  SUBSCRIBE first second
2  PUBLISH first Hello
3  UNSUBSCRIBE first
```

In the code snippet above on line 1, it is shown how the SUBSCRIBE command is called to subscribe the client to the channels named first and second. If any messages are published by other clients to the first or second channels, Redis will broadcast the messages to all subscribers of the corresponding channel. On line 2 a sample command to publish a message to a channel is shown: the Hello message is published to the first channel - as a result, all subscribers of the first channel will receive Hello. To unsubscribe from a channel, the client should call the UNSUBSCRIBE command. On line 3 it is shown how to use the UNSUBSCRIBE command to unsubscribe from a certain channel, in this case, the first channel. Calling the UNSUBSCRIBE command without arguments will unsubscribe the client from all channels. [16]

In a publish and subscribe messaging service publishers and subscribers are decoupled: published messages get characterized into channels without knowing the available subscribers and if a channel has any subscribers at all; in return, subscribers can subscribe to several channels without knowing the existing publishers and if a channel has any publishers. Publishers publish messages to the corresponding channels and the subscribers on the other end receive any messages that are published to the

channels that they are subscribed to without the need to know the existing (if any) subscribers and publishers. [16][17]

Such decoupling of publishers and subscribers enables higher scalability and provides a more robust network topology. One advantage of using Redis Pub/Sub API for creating the Backplane for the communication layer is that it is lightweight, making real-time communication possible at very high throughput while keeping the latency very low. [16][17]

One way to implement a Redis Backplane for scaling an existing .NET Core SignalR application is to add the SignalR.StackExchangeRedis package to the package references of the project file (with the csproj extension). Afterward, the Redis package can be used in the project's service configuration method by calling the AddStackExchangeRedis() method and passing the Redis connection string to it. [18][27] An example of the package reference definition and configured services is provided in the code snippets below. The project's package references:

```
<ItemGroup>
    <PackageReference
        Include="Microsoft.AspNetCore.SignalR.Client"
        Version="5.0.2" />
    <PackageReference
        Include="Microsoft.AspNetCore.SignalR.StackExchangeRedis"
        Version="3.1.0" />
</ItemGroup>
```

The project's service configuration:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR().AddStackExchangeRedis("RedisConnectionString");
}
```

As an alternative solution to implementing and managing the Redis Backplane, SignalR provides a fully managed scaling solution - the Azure SignalR Service [8] which is introduced in the next section of this research work, in *Section 3: Scaling with Azure SignalR Service*.

Section 3: Scaling with Azure SignalR Service

3.1 The Azure SignalR Service concept

Azure SignalR Service provides a fully managed backplane allowing to massively scale WebSocket applications. In fact, Azure SignalR Service is viewed more as a proxy than a backplane since it manages all the client connections while the application instances just need to establish a few persistent connections to the service. [8][15][19][20][21][22][23][24][25][27]

Applications that are scaled using the Azure SignalR Service have the following initial connection flow. At First, a constant connection is established between the application server and the Azure SignalR Service. Once the connection is established between the application server and the Azure SignalR Service, the system is ready to accept connections. Next, the client sends an authentication and connection request to the application server; upon receiving the client connection request, the application server requests an authentication token from the Azure SignalR Service; the Azure SignalR Service responds with the Authentication Token to the application server; the application server redirects the client with the Authentication Token; and, as a result, the client gets connected to the Azure SignalR Service. The described flow is illustrated below in *Figure 7: Azure SignalR Service flow diagram*. [8][15][19][20][21][22][23][24][25][27][28]

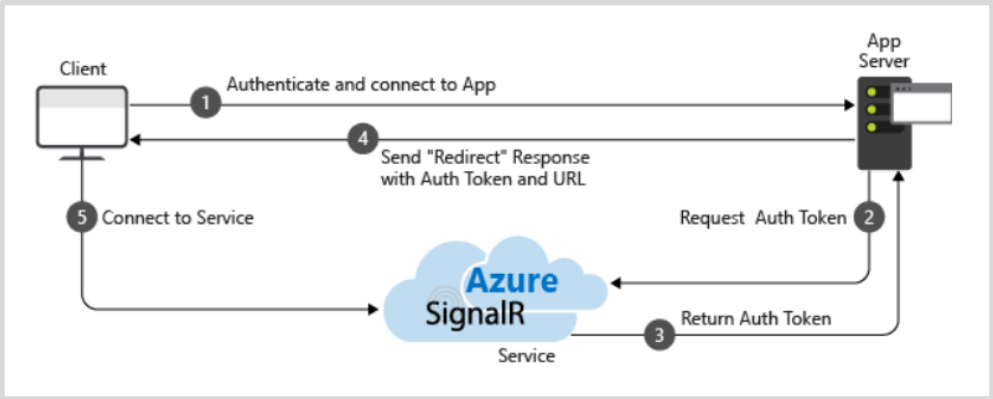

Figure 7: Azure SignalR Service flow diagram

As illustrated in the diagram below in *Figure 8: Azure SignalR Service scale-out approach*, clients are connected to the Azure SignalR Service rather than directly connecting to an instance of the .NET Core SignalR application. The SignalR Service acts as a mediator between the application instances and the connecting clients - the clients and the application's SignalR hub communicate through the service. This design enables scaling the service and adjusting it to manage different levels of traffic without having to modify the application's source code or make changes to the hosting environment. [8][15][19][20][21][22][23][24][25][27][28]

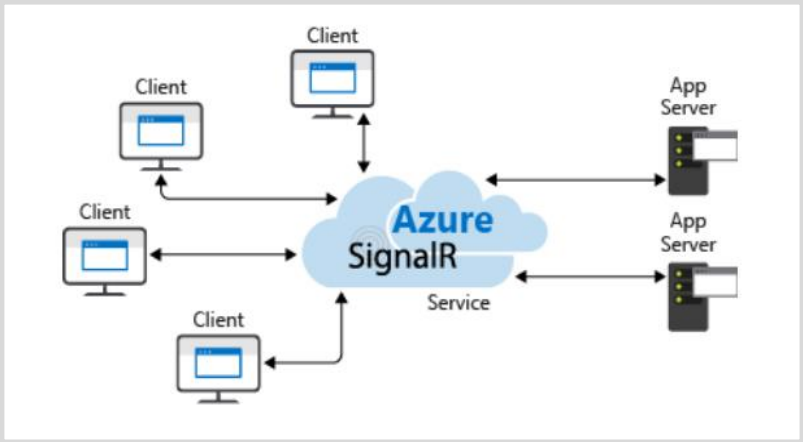

Figure 8: Azure SignalR Service scale-out approach

Due to the decoupled Azure SignalR Service design, explained in the previous paragraph, integrating the service to scale an existing SignalR application requires minimal changes to the source code. The Azure SignalR Service can be integrated into an existing .NET Core SignalR application via installing the Microsoft.Azure.SignalR NuGet package. Next, an Azure SignalR Service needs to be created on the Azure Portal to generate the connection string to the service. The generated connection string is then specified in the SignalR application. In the code snippet below on line 3, it is shown how the Azure SignalR Service is enabled by appending .AddAzureSignalR() to the application configuration services declaration. Application routes also need to be updated by changing UseSignalR to UseAzureSignalR. [8][15][19][20][21][22][23][24][25][27][28]

```
1  public void ConfigureServices(IServiceCollection services)
2  {
3     services.AddSignalR().AddAzureSignalR();
4  }
```

The next section investigates and evaluates the performance factors that impact the Azure SignalR Service in order to properly configure the service for designing highly scalable data-intensive applications.

3.2 Azure SignalR Service performance factors

It is important to investigate the performance factors and benchmarks that have an effect on the Azure SignalR Service inbound and outbound capacity. For evaluating the Azure SignalR Service performance, a sample benchmark tool was created based on typical use-case scenarios which is described in detail in the subsequent section, Section 3.3 Tool for evaluating Azure SignalR Service performance.

To begin, for evaluating performance requirements such as the inbound/outbound capacity, it is important to define which performance factors impact the Azure SignalR Service. Next, it is essential to determine which Azure SignalR Service tier best covers the requirements for each specific use case. In this study, it is presumed that the application server is powerful enough and is not the performance bottleneck; with this in mind, the maximum inbound and outbound bandwidth is checked for every tier focusing on two used transport types: echo and broadcast. [25]

The following paragraphs describe the performance factors that have an affect on the Azure SignalR Service inbound and outbound capacity: computational resources (the selected pricing tier), number of connections, message size and send rate; data transport type; and the routing cost of the use case scenario. [25]

To start, the available computational resources such as the Central Processing Unit (CPU), memory, and network are one of the factors that limit Azure SignalR Service capacity: the more connections the service accepts

and manages, the more memory is utilized; at the same time more Central Processing Unit cycles are required to process larger message traffic where each message exceeds 2,048 bytes. [25]

The next performance factor is the data transfer type: WebSocket, Server-Sent-Event, or Long-Polling - with WebSockets being the most performance efficient, followed by Server-Sent-Events as second best, and Long Polling showing the lowest performance. SignalR uses WebSockets by default and switches to other data transfer types if WebSockets are not supported by the client. In this work, WebSockets is chosen as the data transfer method. [25]

The third performance factor is the message routing cost: acting as a message router, Azure SignalR Service routes incoming messages from clients or servers to other clients or servers; subsequently, a separate routing policy is required for every other API or use case. There are four main scenarios when it comes to types of WebSocket transport: echo, broadcast, send to group, and send to connection. Echo requires the lowest routing cost since in this scenario the client sends the message to itself and sets itself as the routing destination. The rest of the WebSocket transport use cases require more processing units, memory, and increased network bandwidth (the total size of incoming and outgoing messages in 1 second) since for these scenarios the service turns to its internal distributed data structure to find the target connections, which in turn slows down the performance. [25]

It can be concluded that the following performance factors impact Azure SignalR Service inbound and outbound capacity: computational

resources (the selected pricing tier), number of connections, message size and send rate; data transport type; and the routing cost of the use case scenario. The next stage in performance evaluation includes calculating the maximum inbound and outbound bandwidth at which a smooth user experience can still be delivered. [25]

Inbound bandwidth defines the total incoming message size per second and is calculated with the following formula: *inbound bandwidth = inbound connections * (message size / send interval)* [25]. Inbound connections represent the number of connections that are sending the message, message size is the size of a single message that is being sent on average, and the send interval is the timespan in which a single message is sent that is set to one second. Respectfully, outbound bandwidth defines the total outgoing message size per second and is calculated with the following formula: *outbound bandwidth = outbound connections * (message size / send interval)* [25], with the outbound connections representing the number of connections that are receiving the message. [25]

It is important to consider that the maximum inbound bandwidth and outbound bandwidth differs for each pricing tier (the number of selected processing units); and in order for a smooth user experience to be guaranteed, the inbound and outbound connection values should stay below the maximum values. [25]

For mixed use case scenarios where several transport types are being used, evaluating the overall capacity (inbound and outbound bandwidth) is calculated in the following steps: first, mixed use cases are divided into the

basic use cases (echo, broadcast, send to group, or send to connection); second, the maximum inbound and outbound message bandwidth is calculated for each transport type separately using the preceding formulas; afterward the total maximum inbound/outbound bandwidth is calculated by summing the bandwidth calculations of each transport type. [25]

According to performance benchmarking recommendations provided in the Azure SignalR Service documentation [25], the maximum inbound and outbound bandwidth is calculated depending on the number of processing units used. In addition, for each number of units, the committed maximum threshold for Azure SignalR Service is provided; after which, if exceeded, the user experience could suffer from connection throttling. [25]

Performing a quick evaluation by simulating connections (every client represents a single connection) to the Azure SignalR Service each sending a 2,048 bytes in size message every second and by increasing the number of connections until performance starts to give in, revealed that the echo transport type describes the maximum inbound bandwidth since it has the lowest routing cost whereas the broadcast transport type determines the maximum outbound message bandwidth. [25][29] The generated echo evaluation test results for each unit count type are illustrated in *Figure 9: Echo quick evaluation benchmarking test results* above and for broadcast the results are provided in *Figure 10: Broadcast quick evaluation benchmarking test results* below.

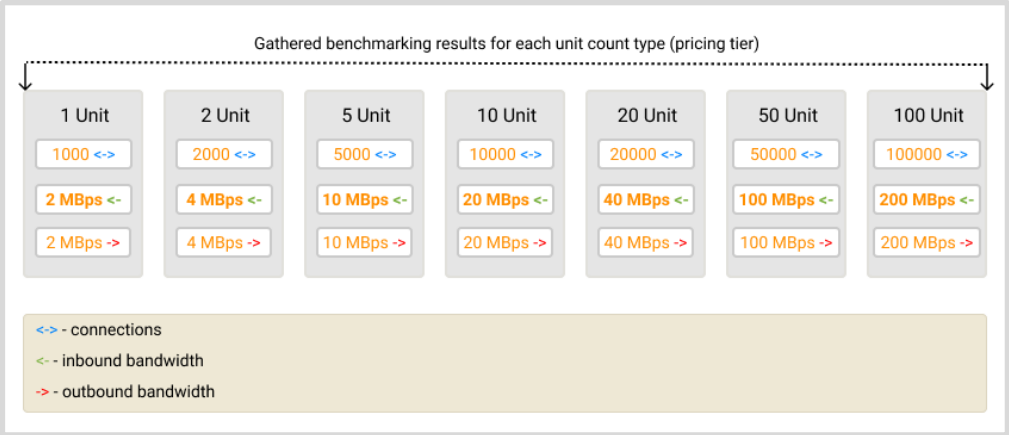

Figure 9: Echo quick evaluation benchmarking test results

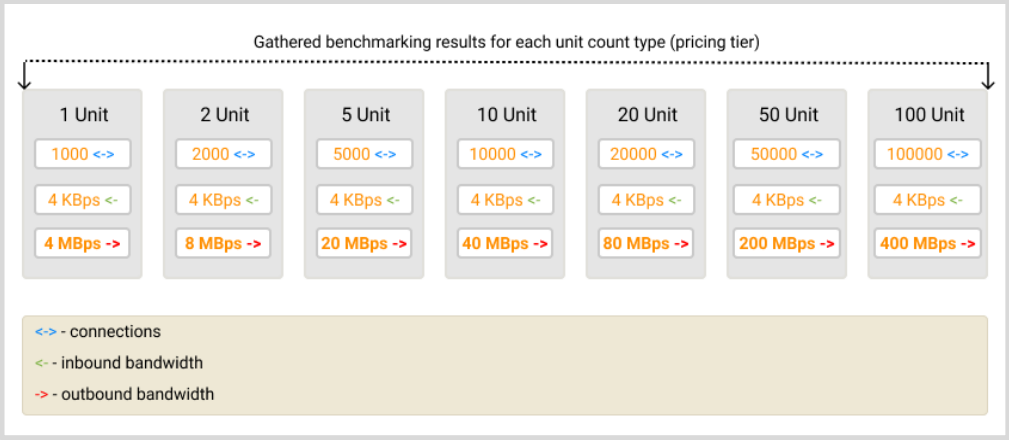

Figure 10: Broadcast quick evaluation benchmarking test results

Having determined the maximum inbound and outbound bandwidth for each azure SignalR Service unit count type, the following paragraphs provide more details on the benchmarking case study for echo and broadcast. Next, it is determined how Azure SignalR Service should be configured (for example, number of application servers and server connections) to best correspond to the performance requirements (inbound and outbound capacity) for use case scenarios when the echo or broadcast WebSocket transport type is being used; therefore, to ensure a smooth user experience in real-time applications that serve a significant amount of traffic.

For both echo and broadcast, the connection establishment stage is the same: first, the application server connects to Azure SignalR Service; when clients connect to the application server, they get redirected to the Azure SignalR Service with the access token and endpoint URL; afterward, WebSocket connections are established between the clients and the Azure SignalR Service. This process is illustrated in the diagram below, in *Figure 11: Initial connection process to Azure SignalR Service.* [25]

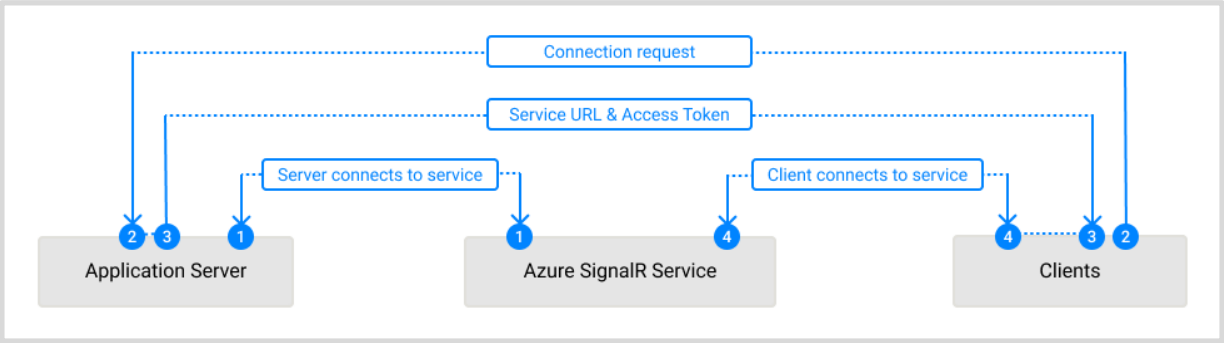

Figure 11: Initial connection process to Azure SignalR Service

The echo Websocket transport type is evaluated by creating the following use case scenario: once clients connect to Azure SignalR Service, each connected client sends a message containing a timestamp to a specific SignalR Hub with a one-second rate and, upon receiving the echoed response from the application server, calculates the latency. This process is executed for five minutes and the statistics of all message latency are provided as a result of the performance test. The communication flow for the echo transport type evaluation is illustrated in *Figure 12: The echo WebSocket transport flow*. [25]

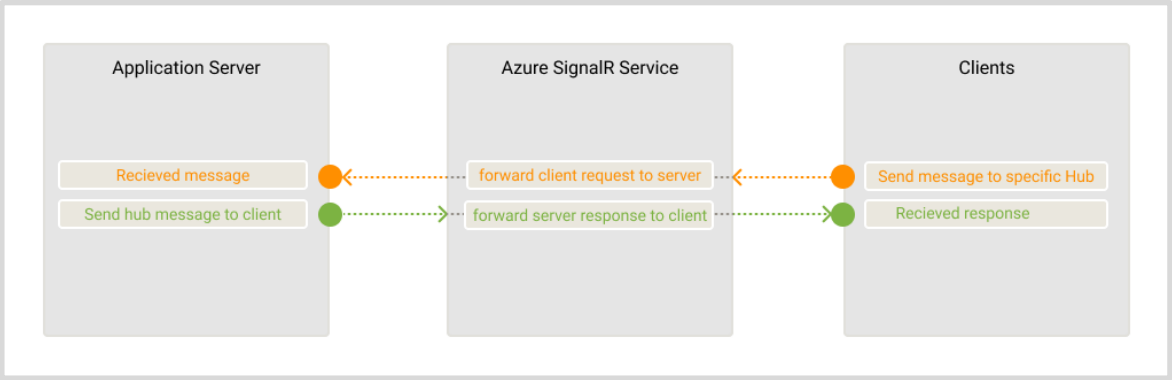

Figure 12: The echo WebSocket transport flow

Echo overall performance is impacted by the following factors: the client connection number, message size, message sending rate, selected unit count type, and CPU/memory of the application server. To ensure a smooth user experience when using echo, the recommendations (for each unit count type) regarding the required application server count are provided in *Figure 13: Recommended application server count in case of echo* below. [25]

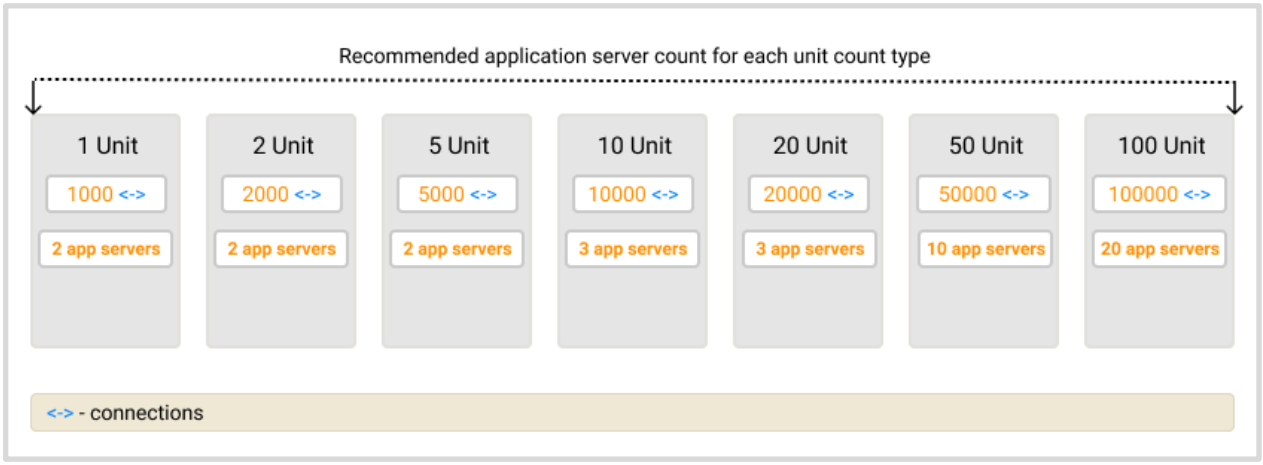

Figure 13: Recommended application server count in case of echo

In the case of broadcast, once the application server receives the message, it broadcasts to all clients. The more clients there are to broadcast, the more messages there are to send to all clients resulting in a more significant amount of traffic. In this use case, in contrast to the echo use case test scenario, a smaller number of clients are broadcasting and even though the inbound message bandwidth is small, the outbound bandwidth turns out significant: with the increase of client connections or broadcast rate, the outbound message bandwidth increases. The broadcast flow is illustrated in *Figure 14: The broadcast WebSocket transport flow*. [25]

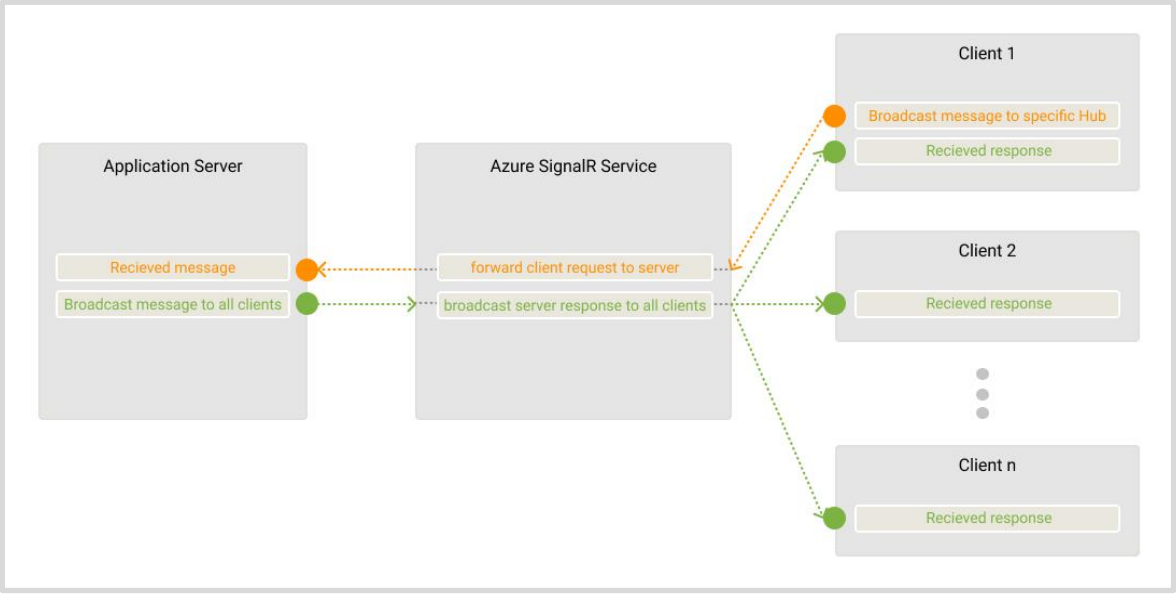

Figure 14: The broadcast WebSocket transport flow

Broadcast overall performance is impacted by the following factors: the client connection number, message size, message sending rate, and the selected unit count type. To ensure a smooth user experience when using broadcast, the recommendations (for each unit count type) regarding the required application server count are provided in *Figure 15: Recommended application server count in case of broadcast*. Fewer application servers are

required in case of broadcast compared with echo since the inbound message bandwidth is significantly smaller. [25]

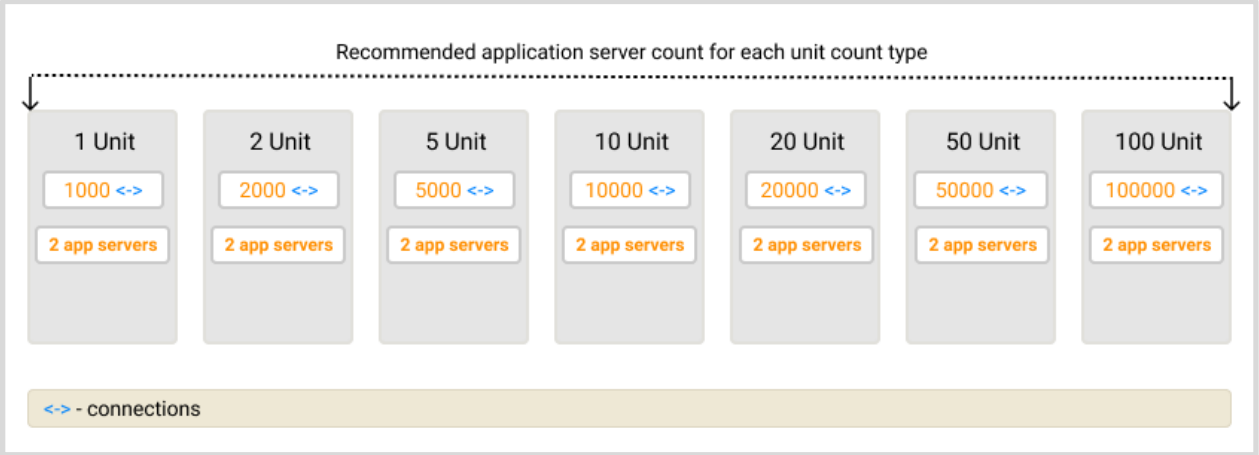

Figure 15: Recommended application server count in case of broadcast

3.3 Tool for evaluating Azure SignalR Service performance

An application was created for evaluating the Azure SignalR Service performance (throughput [1] and latency [1]) as the load increases to ensure high performance of high-traffic data-intensive real-time applications as well as the system's capability to remain performance efficient as the number of potential concurrent users increases.

In the created application, an important metric that is used for describing Azure SignalR Service performance is latency percentile. [1] When describing performance of online systems, it is essential to analyze the time between a client sending a request and receiving a response, also known as the service response time. Since in real case scenarios, when the system is serving different types of requests the response time can turn out significantly different with occasional outlier requests that are processed much longer than expected occurring due to unpredictable causes, response time is viewed as a distribution of measurable values rather than a single value. [1]

To provide relevant insight and analyze how many users got affected by the response time delay, the percentile aggregation is applied to the collected response time data. For this, the list of response time values is sorted from shortest to longest, then the halfway point, the median value, is determined as the 50th percentile: this means that, if for instance the halfway point turned out to be 100 milliseconds, it means that the system responds to half of the user requests less than 100 milliseconds, and the other half is handled above 1000 milliseconds. It is important to consider and analyze tail latencies - high

response time percentiles (for example, the 95th, 99th, and 99.9th percentiles) since such metrics have a direct impact on the user experience. [1] Therefore, during the performance evaluation, these statistics are gathered and provided as part of the performance tests evaluation results.

By design, the application runs in the parent process and workers mode: one parent process oversees several worker nodes. In this process, SignalR clients are delegated to each worker by the parent process, after which the parent process gives each worker a benchmark task to complete: each worker connects to Azure SignalR Service, sends a message and calculates the latency, providing the parent process with the calculation results upon completion. This functionality is implemented with the gRPC [32] (Remote Procedure Call) high-performance framework.

The application uses the methodology of finding the maximum throughput, number of messages that can be processed with Azure SignalR Service, based on the criteria that 99 percent of end-to-end latency of message sending does not exceed the one second threshold. The application’s performance evaluation case study, benchmarking process, and statistics result generation is described in detail in the subsequent paragraphs.

To begin, all SignalR clients connect to Azure SignalR Service at an equal speed, for example, a thousand clients connect to the service at the speed of a hundred client connections per second; in about ten seconds, once all clients are successfully connected to the service, all clients can start sending messages.

With the client connections established, each client starts sending messages at the same message sending rate set as one second. Consequently, in case of echo transport type, the message sending process involves sending messages from the client to the server and then back to the same client; for broadcast, the clients send messages to all clients respectfully.

The message sending process repeats several times: each time with a different number of clients that are sending messages, while the remaining connected clients are kept in idle state. For example, if a thousand client connections are established, at first part of the connected clients are chosen as message senders in a random manner, for instance, two hundred clients are selected to send messages every second to Azure SignalR Service for a certain amount of time, for example for one minute, while the rest of the eight hundred connected clients remain idle. In case during the specified duration, 200 times 99 percent of the message's latency stays below the one second threshold, the number of idle clients is decreased as more clients are assigned the message sending task to increase the load, for instance, the number of message senders is increased to four hundred, and so on.

As the load increases with more and more concurrent messages being sent, the performance test eventually completes when one of the following is true: no more connected clients are left in idle state and all of them are sending messages to the Azure SignalR Service, the set criteria of the 99th latency percentile being below the one second threshold cannot be met, or if the number of disconnected clients exceeds a certain threshold (if clients

get disconnected during the evaluation process and the threshold is not exceeded, the clients will try to reconnect to keep the same load in each process). Upon completion, the gathered statistics of the message latency and number of connections per second during the whole message sending process can be reviewed. An example of the overall performance test result output is provided below:

```
Type : echo
2021-05-03 11:32:22.522 +03:00 [INF] Stop collecting...
2021-05-03 11:32:22.554 +03:00 [INF] -----------
2021-05-03 11:32:22.554 +03:00 [INF]   1000 connections established in 3s
2021-05-03 11:32:22.557 +03:00 [INF] -----------
2021-05-03 11:32:22.557 +03:00 [INF]  Connections/sendingStep: 1000/500 in
5s
2021-05-03 11:32:22.557 +03:00 [INF]  Messages: requests: 1.19MB,
responses: 1.19MB
2021-05-03 11:32:22.558 +03:00 [INF]    Requests/sec: 115.60
2021-05-03 11:32:22.558 +03:00 [INF]    Responses/sec: 115.60
2021-05-03 11:32:22.558 +03:00 [INF]    Write throughput: 237.67KB
2021-05-03 11:32:22.558 +03:00 [INF]    Read throughput: 237.67KB
2021-05-03 11:32:22.558 +03:00 [INF]  Latency:
2021-05-03 11:32:22.558 +03:00 [INF]   50.00%: < 100 ms
2021-05-03 11:32:22.558 +03:00 [INF]   90.00%: < 100 ms
2021-05-03 11:32:22.558 +03:00 [INF]   95.00%: < 100 ms
2021-05-03 11:32:22.558 +03:00 [INF]   99.00%: < 100 ms
2021-05-03 11:32:22.558 +03:00 [INF]  99% time to connect (ms): 501
2021-05-03 11:32:22.559 +03:00 [INF] -----------
2021-05-03 11:32:22.559 +03:00 [INF]  Connections/sendingStep: 1000/1000 in
4s
2021-05-03 11:32:22.559 +03:00 [INF]  Messages: requests: 2.44MB,
responses: 2.44MB
2021-05-03 11:32:22.559 +03:00 [INF]    Requests/sec: 296.50
2021-05-03 11:32:22.559 +03:00 [INF]    Responses/sec: 296.50
2021-05-03 11:32:22.559 +03:00 [INF]    Write throughput: 609.60KB
2021-05-03 11:32:22.559 +03:00 [INF]    Read throughput: 609.60KB
2021-05-03 11:32:22.559 +03:00 [INF]  Latency:
```

```
2021-05-03 11:32:22.559 +03:00 [INF]   50.00%: < 100 ms
2021-05-03 11:32:22.559 +03:00 [INF]   90.00%: < 100 ms
2021-05-03 11:32:22.559 +03:00 [INF]   95.00%: < 100 ms
2021-05-03 11:32:22.559 +03:00 [INF]   99.00%: < 100 ms
2021-05-03 11:32:22.559 +03:00 [INF]  99% time to connect (ms): 501
```

Throughout the performance test, a statistics collector gathers the message latencies and organizes the results in specific latency slots with the maximum value being 1000 milliseconds: 0-100 milliseconds, 100-200 milliseconds, 200-300 milliseconds, 300-400 milliseconds, 400-500 milliseconds, 500-600 milliseconds, 600-700 milliseconds, 700-800 milliseconds, and 900-1000 milliseconds. Along with the latency calculations, the statistics collector also records the connection status and number of client join/leave message sender process statistics. All the performance evaluation results are collected with a one second rate. An example of such statistics is provided below (on lines 16-26 and 63-73 the latency slots can be viewed):

```
1   2021-05-03 11:32:20.903 +03:00 [INF]
2   Statistic type: echo
3   connection:connect:success : 1000
4   connection:connect:fail : 0
5   connection:connect:reconnect : 0
6   group:join:success : 0
7   group:join:fail : 0
8   group:leave:success : 0
9   group:leave:fail : 0
10  message:received : 1186
11  message:sent : 1186
12  message:sentSize : 2438416
13  message:recvSize : 2438416
14  epoch : 2
```

```
sendingStep : 1000
message:lt:100 : 1186
message:lt:200 : 0
message:lt:300 : 0
message:lt:400 : 0
message:lt:500 : 0
message:lt:600 : 0
message:lt:700 : 0
message:lt:800 : 0
message:lt:900 : 0
message:lt:1000 : 0
message:ge:1000 : 0
message:streamItemMissing : 0
connection:connect:lifespan:0.5 : 13856
connection:connect:lifespan:0.9 : 14231
connection:connect:lifespan:0.95 : 14291
connection:connect:lifespan:0.99 : 14352
connection:connect:cost:0.5 : 29
connection:connect:cost:0.9 : 67
connection:connect:cost:0.95 : 92
connection:connect:cost:0.99 : 501
connection:reconnect:cost:0.5 : 0
connection:reconnect:cost:0.9 : 0
connection:reconnect:cost:0.95 : 0
connection:reconnect:cost:0.99 : 0
connection:sla:0.5 : 100
connection:sla:0.9 : 100
connection:sla:0.95 : 100
connection:sla:0.99 : 100
connection:connect:offline:0.5 : 0
connection:connect:offline:0.9 : 0
connection:connect:offline:0.95 : 0
connection:connect:offline:0.99 : 0
2021-05-03 11:32:21.897 +03:00 [INF]
Statistic type: echo
connection:connect:success : 1000
```

```
connection:connect:fail : 0
connection:connect:reconnect : 0
group:join:success : 0
group:join:fail : 0
group:leave:success : 0
group:leave:fail : 0
message:received : 1186
message:sent : 1186
message:sentSize : 2438416
message:recvSize : 2438416
epoch : 2
sendingStep : 1000
message:lt:100 : 1186
message:lt:200 : 0
message:lt:300 : 0
message:lt:400 : 0
message:lt:500 : 0
message:lt:600 : 0
message:lt:700 : 0
message:lt:800 : 0
message:lt:900 : 0
message:lt:1000 : 0
message:ge:1000 : 0
message:streamItemMissing : 0
connection:connect:lifespan:0.5 : 14851
connection:connect:lifespan:0.9 : 15226
connection:connect:lifespan:0.95 : 15286
connection:connect:lifespan:0.99 : 15347
connection:connect:cost:0.5 : 29
connection:connect:cost:0.9 : 67
connection:connect:cost:0.95 : 92
connection:connect:cost:0.99 : 501
connection:reconnect:cost:0.5 : 0
connection:reconnect:cost:0.9 : 0
connection:reconnect:cost:0.95 : 0
connection:reconnect:cost:0.99 : 0
```

```
connection:sla:0.5 : 100
connection:sla:0.9 : 100
connection:sla:0.95 : 100
connection:sla:0.99 : 100
connection:connect:offline:0.5 : 0
connection:connect:offline:0.9 : 0
connection:connect:offline:0.95 : 0
connection:connect:offline:0.99 : 0
```

The described application for evaluating the Azure SignalR Service performance consists of three main components: the sample application server provided in the appserver folder; the gRPC configuration for simulating the client connections (the worker nodes) provided in the RpcServer folder; and the benchmark tool (parent process node) that uses the gRPC configuration for simulating the clients (the worker nodes), points the clients to connect to the application server, and starts performing the benchmark tests - provided in the master folder. The application's structure is provided in the diagram below:

```
.
// The sample application server component:
├── appserver
│   ├── Hub
│   │   └── BenchHub.cs
│   ├── Program.cs
│   ├── Startup.cs
// The benchmark tool (parent process) component:
├── master
│   ├── Controller.cs
│   ├── echo.yaml
│   ├── plugins
// Helper third-party Microsoft Azure SignalR Benchmark plugin:
```

```
│   │   ├── Plugin.Microsoft.Azure.SignalR.Benchmark.dll
│   ├── Program.cs
│   ├── protos
│   │   └── rpc.proto
│   ├── rpc
│   │   ├── RpcClient.cs
│   │   ├── RpcConfig.cs
│   │   └── RpcUtils.cs
│   └── Startup.cs
// The rpc component for simulating client connections (worker
nodes) component:
├── RpcServer
│   ├── Program.cs
│   ├── Protos
│   │   └── rpc.proto
│   ├── rpc
│   │   ├── IRpcServer.cs
│   │   └── RpcServer.cs
│   ├── Services
│   │   └── RpcServiceImpl.cs
│   └── Startup.cs
// Helper third-party components of Microsoft Azure SignalR
Benchmark plugin:
├── rpc
├── signalr
├── interface
├── common
└── utils
```

The application server uses the Azure SignalR Service, through which the concurrent simulated client connections are scaled in order to handle the load. This is demonstrated in the code snippet below (on lines 11-18 and lines 38-41):

```
1  public void ConfigureServices(IServiceCollection services)
2  {
3     if (_useLocalSignalR)
4     {
5         services.AddSignalR();
6     }
7     else
8     {
9         services.AddSignalR()
10                .AddMessagePackProtocol()
11                .AddAzureSignalR(option => {
12                    option.AccessTokenLifetime =
13          TimeSpan.FromHours(_serverConfig.AccessTokenLifetime);
14                    option.ConnectionCount =
15          _serverConfig.ConnectionNumber;
16                    option.ConnectionString =
17          _serverConfig.ConnectionString;
18                 });
19    }
20
21    services.Replace(ServiceDescriptor.Singleton(
22        typeof(ILoggerFactory), typeof(TimedLoggerFactory)));
23 }
24 public void Configure(IApplicationBuilder app)
25 {
26   app.UseRouting();
27   if (_useLocalSignalR)
28   {
29     app.UseEndpoints(endpoints =>
30         {
31           endpoints.MapHub<BenchHub>(HUB_NAME);
32         }
33     );
34   }
35   else
36
```

```
37      {
38
39          app.UseAzureSignalR(routes =>
40          {
41              routes.MapHub<BenchHub>(HUB_NAME);
42          });
43      }
    }
```

The application server's class diagram is provided in *Figure 16: Application server class diagram* below.

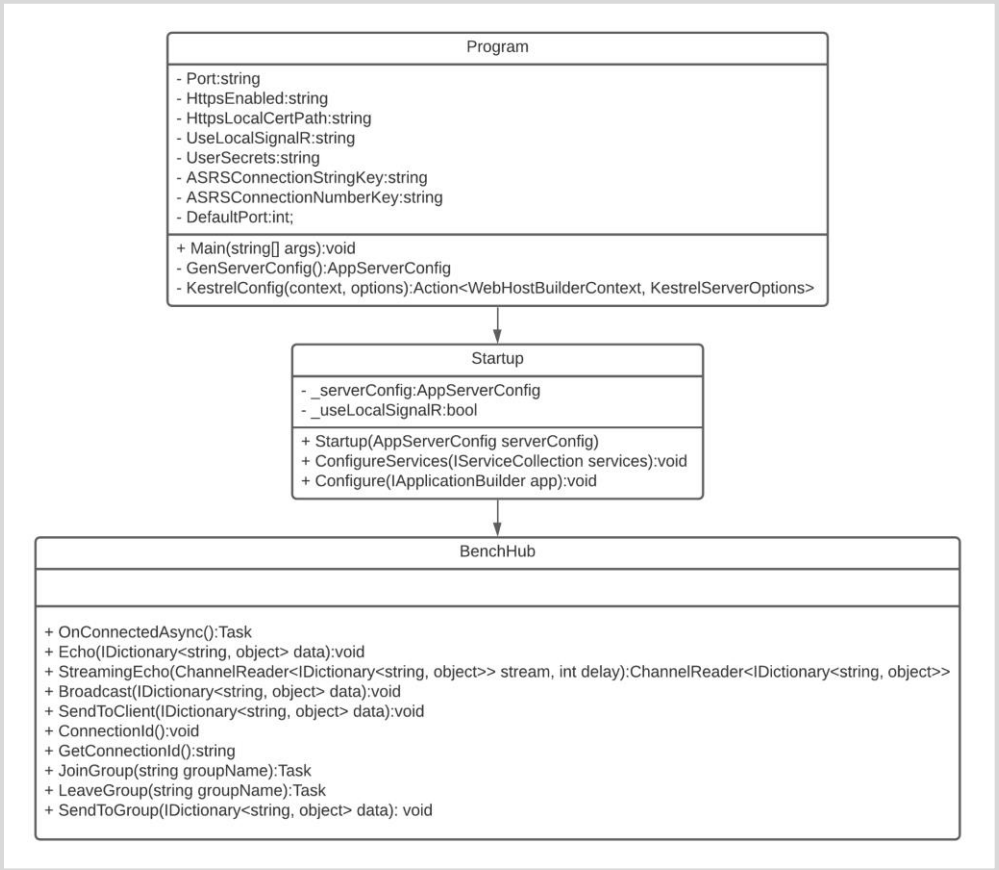

Figure 16: Application server class diagram

The parent process (master node), which represents the benchmark tool component, class diagram is presented in *Figure 17: The benchmark tool component class diagram*.

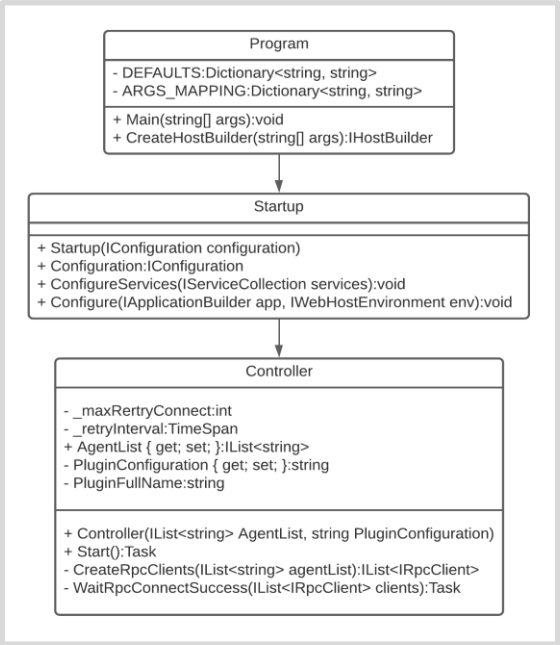

Figure 17: The benchmark tool component class

Finally, the RpcServer class diagram is provided in *Figure 18: The RpcServer class diagram*.

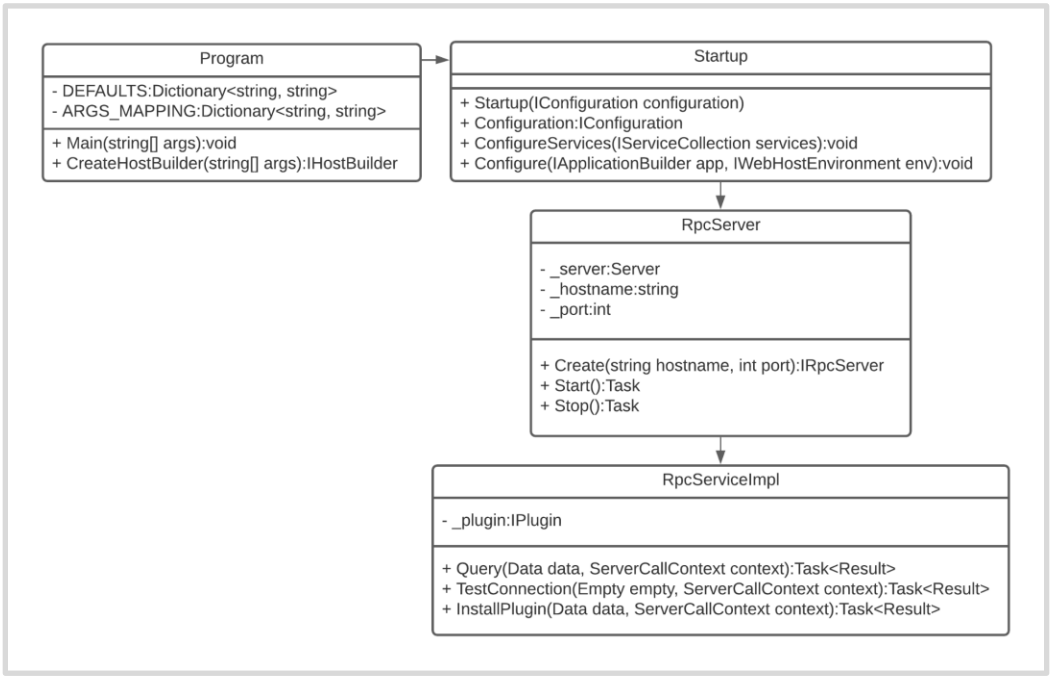

Figure 18: The RpcServer class diagram

CONCLUSION

This work was inspired by the desire to increase real-time applications' capability to remain performance efficient as the number of concurrent client connections grows; therefore, ensuring a smooth user experience in data-intensive applications that serve a significant amount of traffic. In this work persistent connections were scaled out in order to limit the number of concurrent connections that a single application server has to handle, and as a result the system's ability to cope with increased load was improved. Focusing on SignalR Websocket applications, it was determined how they can be scaled out to serve a significant number of clients in a performance efficient manner.

This study introduced two main WebSocket scaling solutions: the Redis Backplane and the Azure SignalR Service, focusing on the Azure SignalR Service as the solution for scaling data-intensive applications. The following performance factors that impact the Azure SignalR Service were determined: computational resources, number of connections, message size and send rate; data transport type; and the routing cost of the use case scenario focusing on echo and broadcast. Next, the maximum inbound and outbound bandwidth at which a smooth user experience can still be delivered was defined. As a result of this work, an Azure SignalR Service performance evaluation tool was created, using latency percentiles as the performance evaluation metric that revealed outstanding scalability of the system, capable of scaling out to serve thousands of connections and at the same time delivering a smooth user experience.

REFERENCES

[1]. Martin Kleppmann Designing Data-Intensive Applications. O'Reilly Media, Inc., March 2017. ISBN: 9781449373320

[2]. Nelly Sattari & Stafford Williams Thousands of concurrent connections with Azure SignalR Service. NDC Conferences, October 2019. https://www.youtube.com/watch?v=s3cq4sQldcM

[3]. https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction

[4]. https://staffordwilliams.com/blog/2019/06/10/load-testing-aspnet-core-signalr/

[5]. https://github.com/dotnet/aspnetcore/tree/main/src/SignalR/perf/benchmarkapps/Crankier

[6]. https://azure.microsoft.com/en-us/pricing/details/app-service/windows/

[7]. https://www.docker.com/

[8]. https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-overview

[9]. https://www.cloudflare.com/learning/performance/what-is-load-balancing/

[10]. https://medium.com/@itIsMadhavan/what-is-load-balancer-and-how-it-works-f7796a230034

[11]. https://avinetworks.com/what-is-load-balancing/

[12]. https://www.nginx.com/resources/glossary/load-balancing/

[13]. https://docs.aws.amazon.com/elasticloadbalancing/latest/application/sticky-sessions.html

[14]. https://docs.aws.amazon.com/elasticloadbalancing/latest/classic/elb-sticky-sessions.html

[15]. https://www.nellysattari.com/real-time-applicationsazure-signalr-service/
[16]. https://redis.io/topics/pubsub
[17]. https://cloud.google.com/pubsub/docs/overview
[18]. https://blexin.com/en/blog-en/redis-as-backplane-to-scale-your-blazor-applications/
[19]. https://docs.microsoft.com/en-us/aspnet/core/signalr/scale
[20]. https://docs.microsoft.com/en-us/azure/architecture/example-scenario/signalr/
[21]. https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-concept-scale-aspnet-core
[22]. https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-howto-scale-signalr
[23]. https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-quickstart-dotnet-core
[24]. https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-concept-internals
[25]. https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-concept-performance
[26]. https://codeopinion.com/practical-asp-net-core-signalr/
[27]. https://codeopinion.com/practical-asp-net-core-signalr-scaling/
[28]. https://www.linkedin.com/learning/scaling-out-using-azure-signalr-service
[29]. https://github.com/dcomartin/Practical.AspNetCore.SignalR
[30]. https://github.com/Azure/azure-signalr/blob/dev/specs/ServiceProtocol.md

[31]. https://www.sas.co.uk/blog/what-is-network-latency-calculator-to-calculate-throughput

[32]. https://docs.microsoft.com/en-us/aspnet/core/grpc/