NCache™

Whitepaper

Scaling ASP.NET SignalR Applications for Peak Performance

By

Ron Hussain

Solutions Architect

July 17, 2018

Table of Contents

Table of Figures

Introduction

ASP.NET is a very popular platform for developing real-time web applications as it provides developers with a rich set of development assistance features. One of such features is SignalR which can help you efficiently develop real-time data processing ASP.NET applications.

SignalR ("R" stands for real-time) is a well-known open source library for ASP.NET web developers that allows you to push content from the server side to all the connected clients as soon as information gets available. This helps simplify the development efforts needed for adding real-time web functionality to your applications.

Why Should You Use ASP.NET SignalR?

SignalR gives you the ability within your ASP.NET applications to have server-side code push content to connected thin clients instantaneously, as it becomes available instead of waiting for clients to make another request for new data. Clients do not need to rely on the typical HTTP polling mechanism anymore and instead, SignalR uses push mechanism for new content to be made available to the clients. This helps improve overall application performance and user experience.

SignalR has a very simple API that is very easy to use to call client-side JavaScript functions inside client browser directly from server side. It uses remote procedure calls (RPC) to invoke Client-Server communication and also provides complete .NET code around Client-Server connection management.

SignalR handles complete communication functionality between servers and clients and you do not need to implement this by yourself. In order to have a super-fast exchange of messages, a transport layer is built-in to SignalR. All you need to do is to introduce SignalR resources into your ASP.NET application and start making use of its methods.

Here is a high-level diagram for reference.

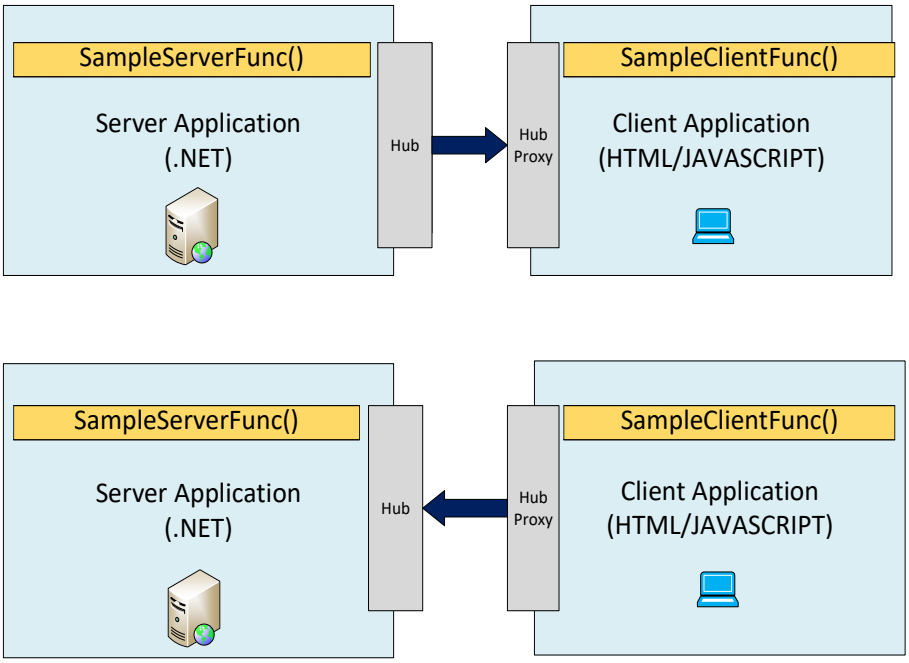

Figure 1: Invocation of Methods in SignalR (from MSDN)

SignalR handles all the connections internally and at the backend, Web Sockets are used if supported by the client (HTML5 and onward). If Web Sockets are not available then it automatically falls back to older transport mechanism where necessary. This simplifies your development efforts and you don't have to write and manage this communication code yourself anymore.

Creating Your First ASP.NET SignalR Application

Using SignalR in your application is much easier than you think. In this case, let's take an example of a Chat Hub. You can start by creating a simple ASP.NET web project with the addition of SignalR. Now, for the Chat Hub program, you can invoke generic hub classes that contain built-in methods for communication with SignalR connections.

Here's how you import the Microsoft.AspNet.SignalR Hub class in your sample program that calls an example *broadcastMessage* on all connected clients.

```
namespace SignalRChat
{
   public class ChatHub : Hub
   {
      public void Send(string name, string message)
      {
         // Call the broadcastMessage method to update clients.
                  Clients.All.broadcastMessage(name, message);
      }
   }
}
```

Figure 2: SignalR ChatHub class implementation

The following example defines the method in a JavaScript client end.

```
<!--Add script to update the page and send messages.-->
    <script type="text/javascript">
        $(function () {
            // Declare a proxy to reference the hub.
            var chat = $.connection.chatHub;
            // Create a function that the hub can call to broadcast messages.
          }  chat.client.broadcastMessage = function (name, message) {
                // Html encode display name and message.

            …
           });
     </script>
```

Figure 3: JavaScript implementation

ASP.NET SignalR Use Cases

Any ASP.NET application that is using a real-time web functionality is the right candidate for incorporating SignalR in its architecture. In other words, this is a scenario where the application is pushing a lot of new content from the server to the client and the user needs to consume it as data is changing.

Similarly, SignalR can be used to enhance the performance of the applications that use AJAX long polling mechanism to retrieve new data. So, as soon as there is a data update in the ASP.NET application, client-side JavaScript methods are called by SignalR automatically to request the dat. This ensures that real-time updates are being sent to the web clients instantaneously.

A few popular use cases for SignalR are mentioned below.

1. Chat Systems
2. Internet of Things (IoT)
3. Gaming Applications
4. Airline Booking Systems
5. Stock Exchange Applications
6. More…

The Problem: SignalR is Not Scalable Out of the Box

While implementing the SignalR library in your ASP.NET application can be a wise choice, its inability to linearly scale can actually defeat its whole purpose. Here's why:

As SignalR is working on a web sockets-based model, its single server deployment works fine with all messages being sent to all the web server connected clients.

However, single server can soon hit capacity based on increased application request load. At this point, you need to scale out by adding more web servers and creating a 'Web Farm'. While doing so, your client requests are distributed and hence may also get routed to different web servers. A client that is connected to one web server via web sockets will not receive messages sent from another web server. This will lead to a situation where clients will not receive SignalR messages and will remain out of sync.

Those clients are actually going to wait until that functionality is pushed by their own respective web servers which will increase latency. Also, in a multi-server-based application, it is not necessary that all web servers are invoking the incoming new data. So, there are strong chances that this new data will not be pushed to the respective clients at all and messages are completely missed.

Let's take a Real Time Stock Market application as an example, where thousands of stock values are changing every second. In such a scenario, the end clients need accurate and absolutely correct information every time any price has changed.

Stock markets deal with huge amounts of high transaction data and it is very important to have all information pushed to all the users in a timely manner. Having a Web farm can cause increased latency for some users as well as some users may miss important updates completely. This creates a poor user experience and will adversely impact your business.

To counter these issues, Microsoft provides the option of using a backplane, which can be generally defined as a central message store, to which all the web servers are connected simultaneously.

SignalR backplane allows web servers to connect and send all messages to itself first instead of sending to their own connected clients. Backplane then broadcasts these messages to all the web servers which intern, transmit these messages to their connected clients. This ensures that all messages are sent to all end clients even invoked by web server where client was not connected.

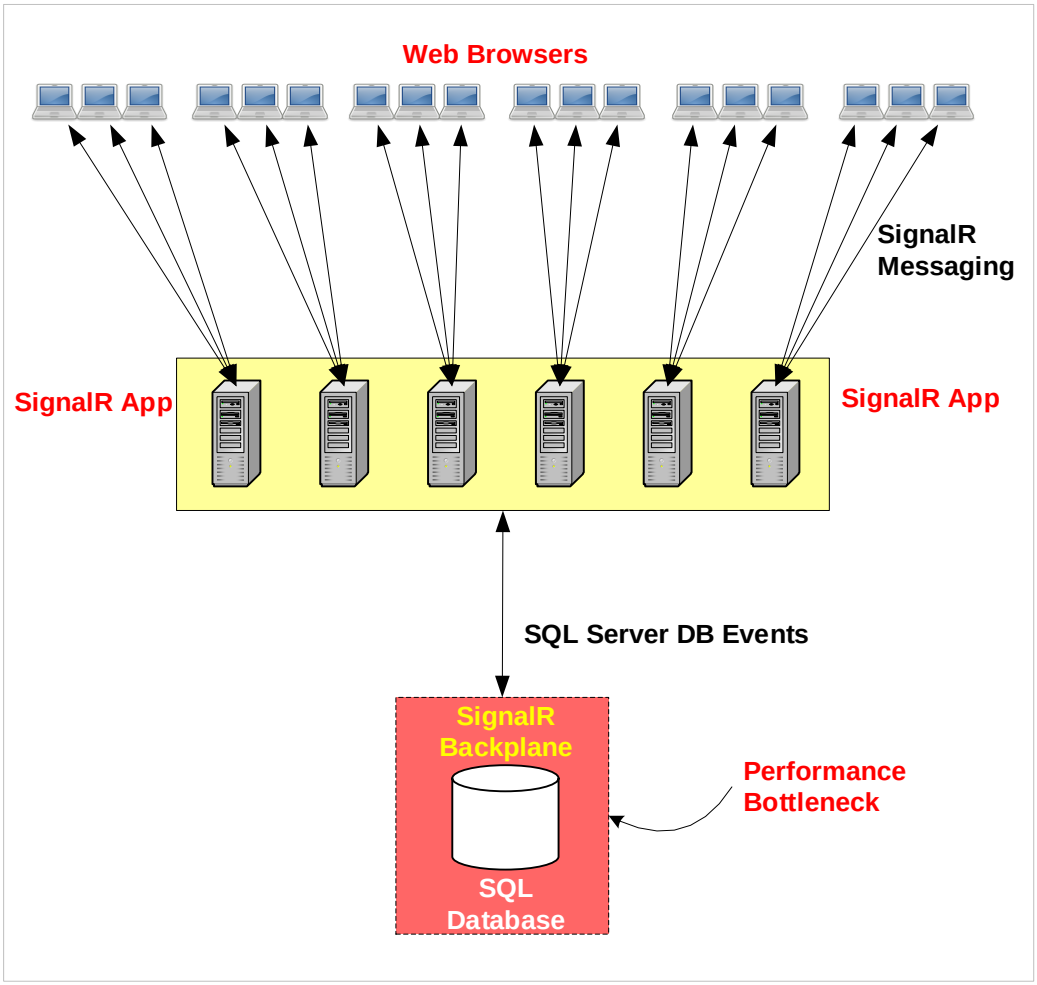

Figure 4: Using Backplane in a SignalR based application

SignalR backplane could be a disk, a database, a message bus or an in-memory distributed cache. Let me discuss most commonly used backplane options and their associated issues.

Issues with Relational Database as SignalR Backplane

SignalR is used in Real time applications dealing with huge volumes of data. Backplane needs to be able to manage extreme message load and that too in a fast and reliable manner. Relational database is commonly used as backplane which is not very suitable as backplane needs to be scalable in nature.

Let's take an example where you have an ASP.NET E-Commerce application which is deployed on 10 Web servers in a web farm. All these 10 web servers have their separate browser-based clients connected to them through SignalR and the Web servers are also connected to a Backplane that is a database in this case.

If one of the web servers generates 1 message each making a total of 10 messages, these 10 messages will get transmitted to the backplane and the database will then broadcast all these messages to all the connected 10 Web servers. This totals up to 100 messages being broadcasted by the database.

Now imagine the same setup but application is notoriously chatty and your servers produce a total of 10000 messages (1000 messages per Web server) at any given moment. The backplane will have to broadcast 100000 messages (10000 x 10) at the same time. You can see how easy it is for database to choke down when the numbers of requests increase.

Here are some issues when relational databases are used for SignalR backplane:

1. Usually, the transaction load is very high and a faster delivery is required to rule out the latency factor. Relational databases are slow and can even choke down under peak real-time data processing loads
2. Being based on a disk, a relational database can never work fast enough to achieve the desired amount of throughput under high load.
3. Most importantly, the relational databases clearly lack the linear scaling ability where you cannot add more servers to handle more load

Issues with Redis as SignalR Backplane

The second option can be to use Redis as your SignalR backplane which although resolves performance and scalability related problems that you have with relational databases but it is also not a suitable choice to be made. Here are some of the reasons around this:

- Redis is a Linux based distributed cache and is not a native .NET option. Using a Linux based system for ASP.NET SignalR does not make a lot of sense where you will have to have Linux stack along with windows and will need separate expertise to manage and monitor this setup.
- A .NET developer always longs for a 100% native .Net stack while developing such real-time web applications. It will be counter intuitive to manage a non-native .NET solution as a SignalR backplane while all other application modules are 100% native .NET.
- Another limitation in Redis is that its .NET windows ported version in Microsoft Azure platform is full of bugs as per user reviews which even refrains Redis from using their own .NET Windows ported version in Azure.

This makes Redis, an ambivalent choice between .NET developers from the compatibility point of view.

The Solution: Use NCache as SignalR Backplane

NCache is an In-Memory Distributed Caching System developed by Alachisoft and is the most appropriate option to be used as a backplane. NCache is a cluster of inexpensive cache servers which are pooled together for providing scalable memory and CPU capacity. It's essentially, a logical capacity of multiple servers that comes with all the capabilities to be used as a SignalR backplane.

The best part about using NCache as your SignalR backplane is that it is hundred percent native .NET and you don't need any major code changes in your ASP.NET application. It's like a plug-n-play option which not only gives you the ability to manage your backplane messages, but you can also monitor these messages using the NCache performance monitoring tools.

NCache provides SignalR extension to be used as backplane in your ASP.NET applications.

All your web servers in the web farm are registered with NCache using NCache Pub/Sub messaging platform. Your Web Servers register a specific Topic for SignalR messages and NCache then broadcasts these messages to all Web Servers. It is basically a two-way message transmitting structure where your publishers can be subscribers and similarly vice versa.

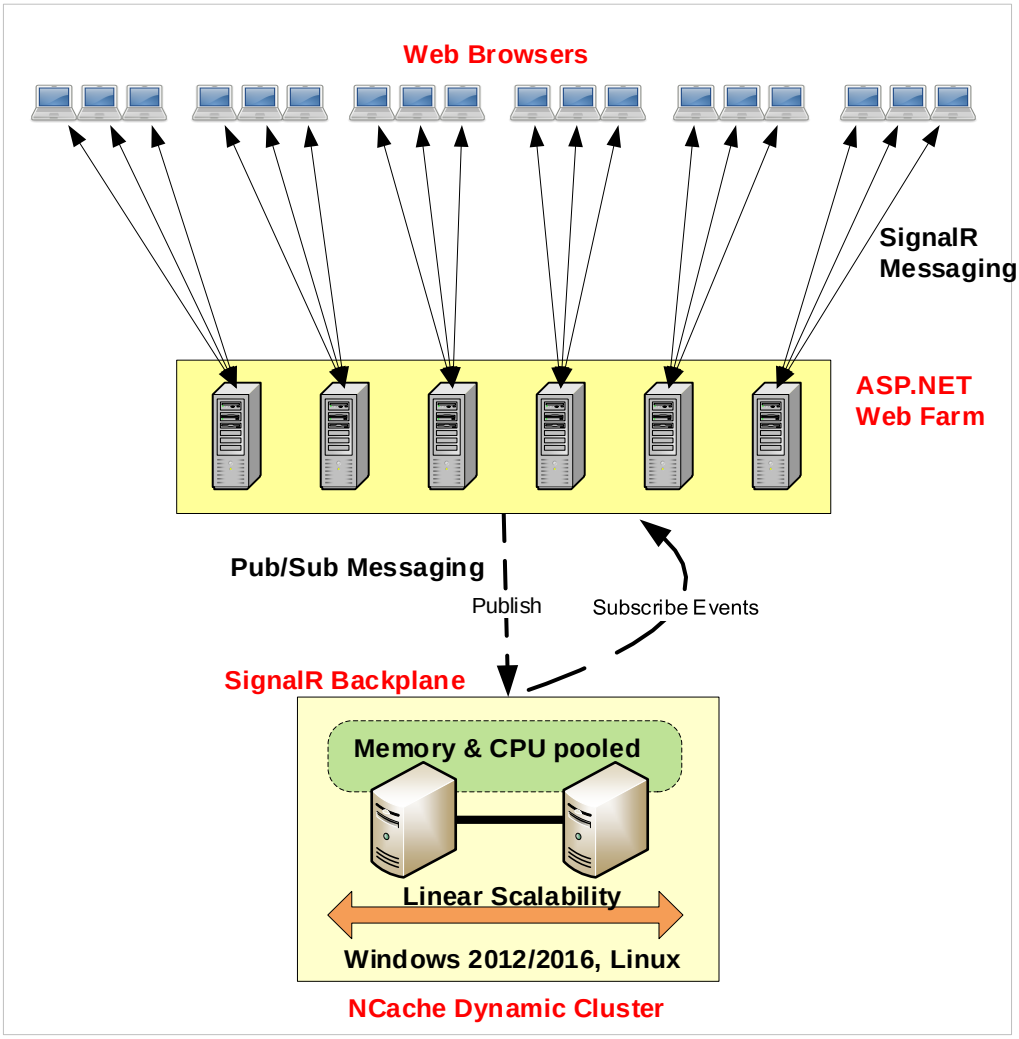

Figure 5: Using NCache as a SignalR Backplane

To plug-in NCache as a backplane in your SignalR-based application, you only need to introduce one line of code, which is mentioned in the step-by-step implementation guide.

Why NCache Backplane is Better than SQL Server?

To achieve the best user experience in your ASP.NET application, NCache acts as a very fast, reliable and scalable backplane to handle huge amounts of notifications. Below mentioned are some key advantages of using NCache instead of RDBMs as a backplane.

1. NCache is Linearly Scalable (High Throughput & Low Latency)

The most important characteristic of NCache backplane is that it delivers maximum throughput & minimum latency. It seamlessly handles large amounts of data in a fast manner by keeping all message processing in-memory, ruling out any chances of latency increase.

To deliver all the messages in time, NCache delivers maximum throughput during the transmission by distributing the load on all available servers in the cache cluster as well as it also allows you to add more servers at runtime.

Here are some performance benchmark numbers for a reference.

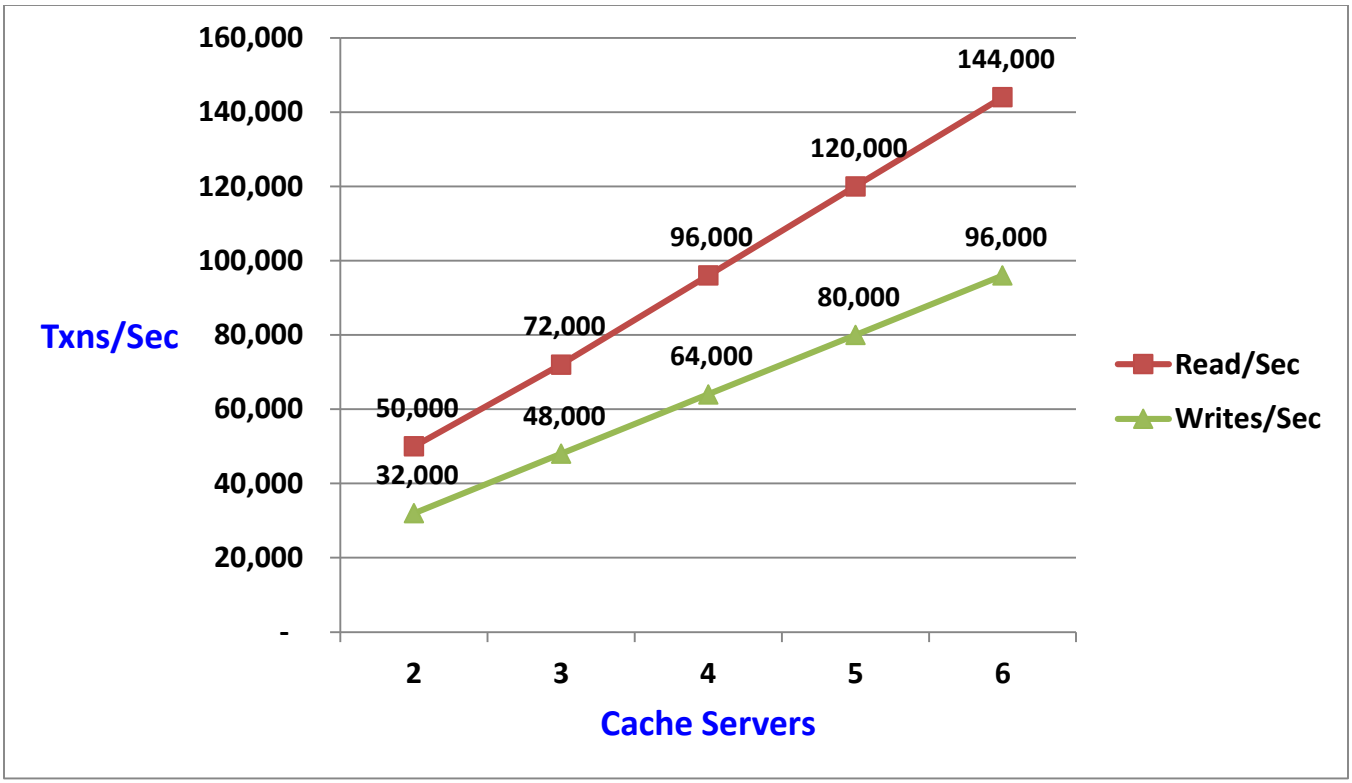

Figure 6: Performance Numbers for NCache

This gives you an overall linear scalability within your application architecture and you don't have to worry about your application performance going down and specifically under extreme loads.

2. **NCache is Extremely Reliable**

The second characteristic of NCache backplane is its extreme reliability. To ensure a guaranteed delivery of messages, especially in the case of mission-critical applications, NCache can be used as a backplane so you don't have to worry about any of your messages being lost due to servers going down or being brought down for maintenance. It keeps backup of each Cache server used in the cluster for SignalR backplane which is made automatically available in case a server goes down.

The extreme reliability of NCache is attributed to its characteristic that broadcasts all the messages to each and every web server that is connected to the backplane. The backup of your data on other servers ensures that while your mission-critical applications are transferring heavy payloads, there are no chances of data loss.

3. **NCache is Highly Available**

Another important characteristic of NCache backplane is its high availability. With NCache installed as your SignalR backplane, your system becomes highly available as your messages are now being transferred through an always active cache cluster.

NCache architecture ensures the high availability feature through its 'Always Active Cache Cluster' which can be based upon one of our various caching topologies. For example, the 'Partition-replica' topology provides you an active partition on all servers and each server has a backup of it. So, if a server goes down, the clients detect that automatically and failover their connection to the other surviving nodes.

The cache cluster of NCache is capable of working even if only 1 server is up and working and that's what you need at the minimum for hundred percent uptime scenario. This feature pays off in cases where your servers are either hit by a natural calamity or you intentionally bring them down for maintenance purposes. In brief, the pragmatic caching topologies of NCache help you to achieve a 100% uptime in your SignalR-based applications.

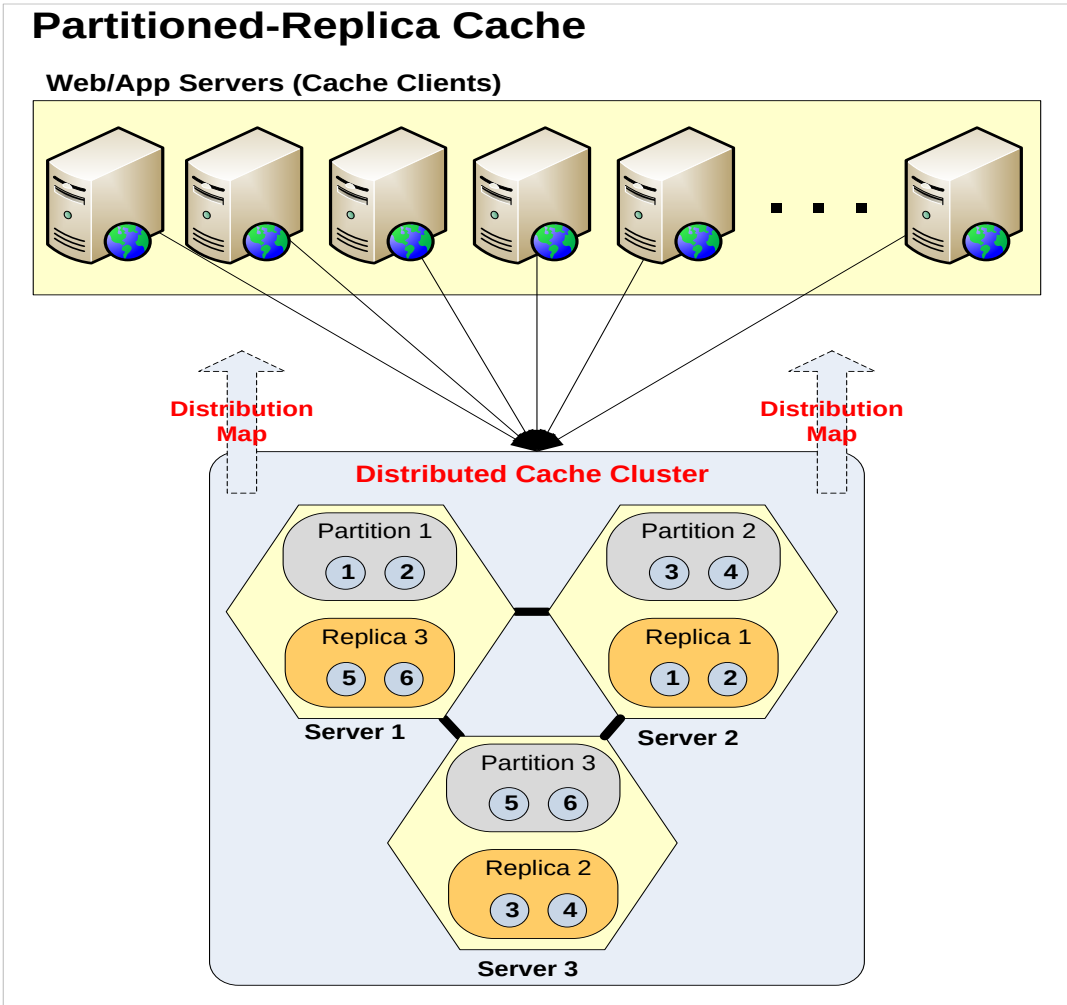

Figure 7: Partition-Replica Caching Topology

Why NCache is better than Redis?

NCache is also better than Redis from features stand point. To rule out all the shortcomings faced by developers in Redis distributed cache system, NCache has these three superlative features.

- **Native .NET:** NCache is 100% native .NET and therefore it's a perfect fit for ASP.NET applications using SignalR. On the other hand, Redis comes from a Linux background and is not a native .NET cache.
- **Faster than Redis:** NCache is actually faster than Redis. NCache Client Cache feature gives NCache a significant performance boost.
- **More Features:** NCache offers a number of very important distributed cache features that Redis does not. See more details in this **Redis vs NCache**

Implementing NCache SignalR Backplane

You can deploy NCache as your SignalR Backplane with just one-line change and you can follow the below step by step tutorial to equip your applications for this.

1. Install NuGet Package

Start off by installing the Alachisoft.NCache.SignalR NuGet package to your application by executing the following command in the Package Manager Console:

```
Install-Package Alachisoft.NCache.SignalR
```

2. Include Namespace

In your *Startup.cs* file, include these two namespaces as mentioned below:

- `Alachisoft.NCache.SignalR`
- `Microsoft.AspNet.SignalR`

3. Modify Web.config

In your Web.config file, you will have to define the cacheName and `eventKey` in the `<appSettings>` tag:

```
<configuration>
    <appSettings>
        <add key="cache" value="myPartitionedCache"/>
        <add key="eventKey" value="Chat"/>
    </appSettings>
</configuration>
```

Figure 8: Web.config changes

4. Register NCache as SignalR Backplane for your Application

Register an instance of the `UseNCache()` method in `Startup.cs` of your application in either of the following overloads:

Overload 1:

```
public static IDependencyResolver
       UseNCache(this IDependencyResolver resolver, string cacheName, string eventKey);

public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        string cache, eventKey;
        cache = ConfigurationManager.AppSettings["cache"];
        eventKey = ConfigurationManager.AppSettings["eventKey"];
        //using NCache SignalR
        GlobalHost.DependencyResolver.UseNCache(cache, eventKey);
        app.MapSignalR();
    }
}
```

Figure 9: Registering NCache SignalR Backplane (Overload 1)

Overload 2:

```
public static IDependencyResolver
UseNCache(this IDependencyResolver resolver, NCacheScaleoutConfiguration configuration);

public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        string cache, eventKey;
        cache = ConfigurationManager.AppSettings["cache"];
        eventKey = ConfigurationManager.AppSettings["eventKey"];
        NCacheScaleoutConfiguration configuration =
                    new NCacheScaleoutConfiguration(cache, eventKey);
       //using NCache SignalR
       GlobalHost.DependencyResolver.UseNCache(configuration);
        app.MapSignalR();
    }
```

Figure 10: Registering NCache SignalR Backplane (Overload 2)

Conclusion

To conclude it all, NCache is essentially a .NET based in-memory distributed cache which can be used in all real-time ASP.NET web applications as a backplane for your SignalR to increase the performance of your Web Application.

The features which set it apart from other available options include its super-fast performance, 100% uptime, Data reliability, guaranteed delivery of messages and the ability to maintain the maximum throughput with minimum latency.

