NCache
Distributed Cache for .NET

Whitepaper

Scaling .NET Core Applications

to

Extreme Performance

by

Iqbal Khan

August 05, 2019

Table of Content

Table of Figures

1 Introduction

.NET Core and ASP.NET Core are gaining popularity due to their simplicity of design, being lightweight, open source and able to run on both Windows and Linux. As a result, many existing applications are also moving to .NET Core from the .NET Framework. Almost all new applications are being developed in .NET Core.

Many of these .NET Core applications are high traffic in nature, serving millions of users and transactions. As a result, these applications have a huge impact on your business and are therefore very important.

1.1 Who Needs Scalability?

The .NET Core applications that usually need scalability are server applications that must process a lot of transactions very quickly with very fast response times. Many of these applications are customer facing, meaning they're processing customer requests. If they do not perform customer requests quickly, the cost to the business is high in terms of lost revenue and losing happy customers.

Following .NET Core applications requires scalability:

1. **Web Apps (ASP.NET Core):** These are usually customer-facing applications but could also be internally facing applications for large companies.
2. **Web Services (ASP.NET Core):** These could either be directly providing Web APIs to customers or could be a part of another high transaction application containing application tier logic in these web services.
3. **Real-time Web Apps (ASP.NET Core SignalR):** These are real-time applications that must provide frequent refreshes to their users by using ASP.NET Core's SignalR framework. They must also perform fast as they're usually customer facing.
4. **Microservices (.NET Core):** This is a new application architecture for server-side applications. And just like Web Services, these Microservices are usually part of a customer facing web application or a customer-facing Web Services application. As a result, they also have high-performance requirements under heavy transaction loads.
5. **Other Server Apps (.NET Core):** There are a rich variety of other server applications that must process a large amount of transactions really fast. These could be batch processing applications handling various types of backend workflows or they could be stream processing applications ingesting a large amount of data for near real-time processing. The list goes on.

2 The Problem: Scalability Bottlenecks

Interestingly, all the applications mentioned above have very scalable application-level architectures. Each of them allows you to linearly scale as your transaction load grows by adding more servers, VMs, or container instances along with a load balancer.

But, despite a very scalable architecture at the application tier, .NET Core server applications today are facing major scalability bottlenecks. These bottlenecks are occurring in different areas like:

1. **Application Databases (Relational Databases):** This is the biggest bottleneck of all. I explain it in more detail below.

2. **ASP.NET Core Session Storage:** If sessions are stored in SQL Server, then your ASP.NET Core application will face huge bottlenecks.
3. **ASP.NET Core Repetitive Page Processing:** If the same pages are executed repeatedly and their output or response stays the same then it is a waste of resources and a performance bottleneck.
4. **ASP.NET Core SignalR Backplane Provider:** If a live web app using SignalR has to scale, then its Backplane Provider can easily become a bottleneck.
5. **Pub/Sub Messaging (Not In-Memory):** If your .NET Core application is using Pub/Sub messaging, then chances are it is not In-Memory and therefore a bottleneck.

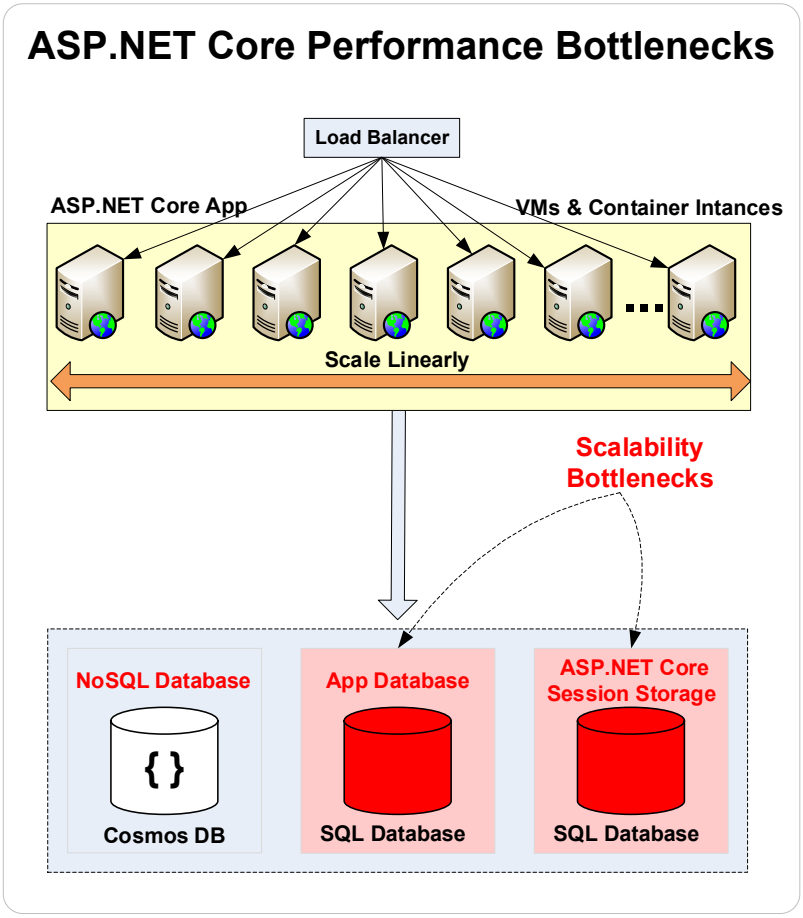

Figure 1: ASP.NET Core App Facing Scalability Bottlenecks

2.1 Relational Database Bottleneck

The biggest bottleneck for all high traffic .NET Core applications is their application database. Most applications today are still using a relational database like SQL Server or Oracle. These databases quickly become scalability bottlenecks as you increase transaction loads on these applications. This is true whether you're using SQL Server on a VM or Azure SQL Database.

This happens because a relational database cannot be logically partitioned like a NoSQL database and instead stays in one physical location; even some column level partitioning is nothing like a true NoSQL style partition. Therefore, you cannot grow the database tier transaction capacity by adding more database servers like you can with a NoSQL database.

For example, while your application tier can easily have 10, 20, 30 or more application servers as your transaction load grows, your database tier cannot grow in the same fashion at all.

Because of all this, your relational database becomes a performance bottleneck for any data you store in it (application data or other data).

2.2 Database Server In-Memory Optimizations Not Enough

SQL Server has introduced In-Memory optimizations to increase the number of transactions per second. Oracle has also provided their own version of In-Memory tables.

While In-Memory optimizations bring about performance improvements, they do not address the core issue of linear scalability. In-Memory tables are generally used for read-only data and in order to scale a read-only transaction capacity, you need to add more instances of SQL Server on higher-end machines.

In-Memory tables also have limitations on the size of the data; you cannot put large tables in memory since the entire table must be put in memory. And their replication to other SQL Server instances can only be done to other In-Memory tables and not a proper database.

In summary, these In-Memory Optimizations in SQL Server and Oracle databases are not able to fully address your .NET Core application's scalability needs.

2.3 NoSQL Database Not the Answer

One of the reasons NoSQL databases became popular is because they provide proper partitioning of data based on Hash-based and other algorithms. This resolves many of the issues of scalability for transaction capacity that relational databases like SQL Server and Oracle face.

But, there are reasons NoSQL databases are not the ideal solution for these database bottlenecks.

1. **Not an In-Memory Store:** NoSQL databases store their data on the disk just like a relational database. This means that no matter what you do, the slow performance of the disk ultimately becomes a performance bottleneck.
2. **Cannot be Used Most of the Time:** NoSQL databases require you to abandon using relational databases like SQL Server and Oracle and to replace them with a NoSQL database. This is not possible in the majority of cases for both technical and non-technical reasons. Essentially, your business depends on your relational database and cannot easily abandon it. As a result, you're unable to take full advantage of a NoSQL database.

3 The Solution: In-Memory Distributed Cache (NCache)

The solution to all the problems mentioned above is to use an In-Memory Distributed Cache like NCache in your .NET Core application deployment. NCache is an Open Source distributed cache for .NET and .NET Core that is extremely fast and linearly scalable. Think of it as an in-memory data store that is also distributed. Being in-memory makes it extremely fast and being distributed makes it linearly scalable.

NCache is linearly scalable because it builds a TCP cluster of low-cost cache servers (same config as your web app servers but with more memory) and pools the memory and CPU resources of all these servers into one logical capacity. NCache then allows you to add cache servers to this cluster

at runtime as your transaction load grows. And, since NCache is all in-memory, it is super-fast and gives you sub-millisecond response times that you cannot expect from your relational databases or even NoSQL databases.

On top of providing linear scalability, a distributed cache like NCache replicates data intelligently so your performance is not compromised while achieving data reliability in case any cache server goes down.

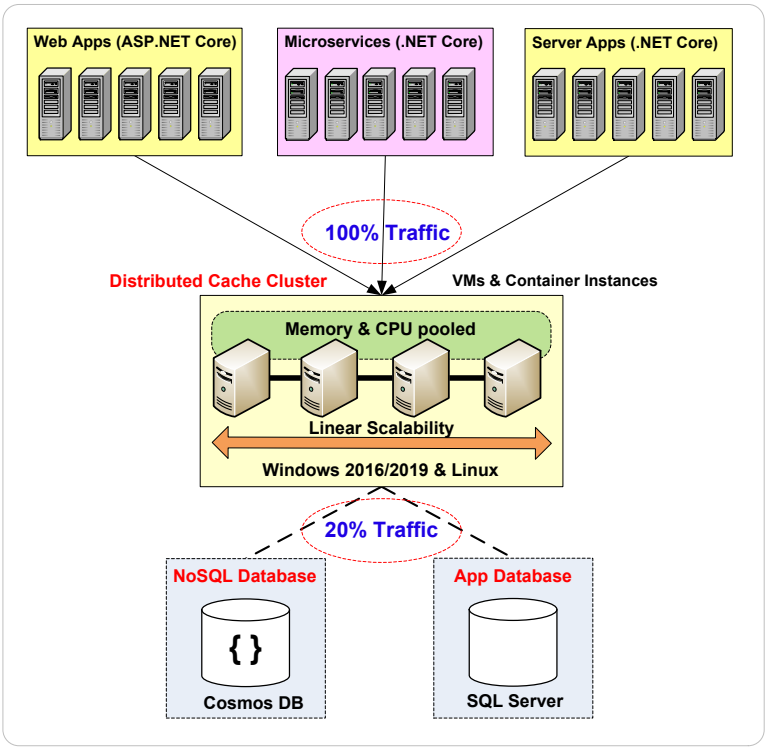

Figure 2: NCache Deployed in the Enterprise for .NET Core

NCache lets you scale your .NET Core applications through the following:

- Application Data Caching
- ASP.NET Core Session Storage
- ASP.NET Core Response Cache Middleware
- ASP.NET Core SignalR Backplane
- Pub/Sub Messaging & CQ Events (In-Memory)
- Continuous Query Events (In-Memory)

3.1 Application Data Caching

The most important bottleneck faced by .NET Core applications is the "Application Database". The nice thing about NCache is that unlike NoSQL databases, NCache does not ask you to stop using your existing relational database. You can keep using SQL Server, Azure SQL Database, Oracle, etc. as your database and still achieve linear scalability by using NCache on top of your relational database. This is because NCache removes all of the relational database scalability bottlenecks because unlike your database, NCache is actually linearly scalable.

Application Data Caching enables you to remove your database bottlenecks. NCache allows you to cache application data and reduce those expensive database trips. You can expect to divert 80-

90% of database traffic to NCache. This reduces pressure on your database and allows it to perform faster and handle larger transaction loads without slowing down.

Application data caching means you cache whatever application data you get from your relational database. This is usually in the form of domain objects (also called entities). Here is an example of how to use a distributed cache like NCache for application data caching.

```
Customer Load(string custId)
{
    ICache cache = CacheManager.GetCache("myCache");

    string key = "Customer:CustomerID:" + custId;
    Customer cust = cache.Get<Customer>(key);

    if (cust == null) {
       // Item not in cache so load from db
       LoadCustomerFromDb(cust);
       // Add item to cache for future reference
       cache.Add(key, cust);
    }
    return cust;
}
```

Figure 3: Using In-Memory Distributed Cache for App Data Caching

3.2 ASP.NET Core Session Storage

Another possible bottleneck is if you store your ASP.NET Core Sessions in SQL Server or stand-alone MemoryCache. Both options have great limitations about performance and scalability. SQL Server storage is not good for ASP.NET Core sessions and quickly becomes a bottleneck just like for Application Data.

NCache is a great place to store your ASP.NET Core Sessions because it is much faster and more scalable than other storage options. NCache is faster because it is in-memory and provides a key-value interface with the value being an "object" which an ASP.NET Core Session is. And, it is scalable because it is a distributed cache.

And, NCache also intelligently replicates ASP.NET Core sessions through its rich caching topologies so even if a cache server goes down, there is no session data loss. This replication is needed because NCache provides an in-memory store and memory is violate storage.

NCache also speeds up your serialization of the ASP.NET Core Session that is required before it can be stored out-of-process. NCache does this by using its Dynamic Compact Serialization feature that is 10 times faster than regular .NET and .NET Core serialization. You can use this feature without making any code changes.

You can use NCache as your ASP.NET Core Session store in two ways.

1. **IDistributedCache for ASP.NET Core Session:** NCache has implemented the `IDistributedCache` interface that allows you to automatically plug-in NCache as your ASP.NET Core Session store provider. But this has fewer features than the other option.
2. **NCache Provider for ASP.NET Core Session:** NCache has also implemented its own more feature-rich ASP.NET Core Session store provider that you can use. It has more features in terms of extra locking, timeouts, etc.

Below is an example of how you can configure your ASP.NET Core application to use NCache Session Provider:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
       // Specify NCache as the session provider
       services.AddNCacheSession(Configuration.GetSection("NCacheSettings"));
       ...
    }

    public void Configure(IApplicationBuilder app, ...)
    {
       // select NCache session provider for ASP.NET Core
       app.UseNCacheSession();
       ...
    }
}
```

Figure 4: Plug-in NCache as ASP.NET Core Sessions

3.3 ASP.NET Core Response Cache Middleware

ASP.NET Core applications, that otherwise have quite a dynamic content, face situations where for some of their pages the content or response does not change across multiple requests. But these pages still have to be executed each time the request comes. And, this puts an unnecessary burden on the web server resources and also on all the tiers of this application. As a result, this also adds to performance bottlenecks and limits the scalability of the application.

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
       // Turn on ASP.NET Core Response Cache with IDistributedCache
       services.AddResponseCaching();

       // Select NCache as IDistributedCache provider
       services.AddNCacheDistributedCache(
                                 Configuration.GetSection("NCacheSettings"));
       ...
    }
}
```

Figure 5: Plug-in NCache as ASP.NET Core Response Cache Middleware

To address the overhead of repetitive page execution where the page response does not change, ASP.NET Core has provided a page response caching mechanism called ASP.NET Response Cache Middleware. And, NCache has implemented `IDistributedCache` interface in ASP.NET Core due to which you can seamlessly plug-in NCache as your ASP.NET Core Response Cache Middleware.

So, you can use NCache to cache ASP.NET Core page responses for a certain period of time so next time the same page is called with the same parameters, this cached response can be retuned instead of executing the entire page again. Below is the code example on how to configure NCache as your ASP.NET Core Response Cache Middleware.

3.4 ASP.NET Core SignalR Backplane

If your ASP.NET Core application is a real-time web application then it is most likely using ASP.NET Core SignalR for providing this real-time behavior. Real-time web applications provide high-frequency updates from the server to the client. The examples of such applications include gaming, auction, voting, social networks, etc.

If your ASP.NET Core application is running in a load-balanced multi-server environment then it has to use an ASP.NET Core SignalR Backplane provider in order to share events across multiple web servers. And, this Backplane has to be scalable. Otherwise, your ASP.NET Core SignalR application starts to face performance bottlenecks.

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
       // Specify NCache as the ASP.NET Core SignalR Backplane
       services.AddSignalR().AddNCache(ncacheOptions =>
       { ncacheOptions.CacheName = "myPartitionedCache"; });
       ...
    }

    public void Configure(IApplicationBuilder app, ...)
    {
       // Use SignalR in ASP.NET Core
       app.UseSignalR(config => { config.MapHub<MessageHub>("/messages"); });
       ...
    }
}
```

Figure 6: Plug-in NCache as ASP.NET Core SignalR Backplane Provider

NCache has implemented an ASP.NET Core SignalR Backplane provider. NCache's ASP.NET Core SignalR Backplane provider uses Pub/Sub Messaging features of NCache that are super-fast due to being totally in-memory. This allows your ASP.NET Core SignalR application to speed up SignalR event propagation among all the web servers and as a result to the clients.

And, this makes your real-time web application more responsive in delivering those frequent updates to the clients. And, you can keep growing the number of clients and also add more web servers without fearing any performance bottlenecks.

3.5 Pub/Sub Messaging (In-Memory)

If your .NET Core application needs to use Pub/Sub Messaging or Events then it is most likely using a Pub/Sub messaging platform that is not fully In-Memory and instead stores all messages on the disk. As a result, this can easily become a performance bottleneck if your application is really high transaction.

NCache also provides a Pub/Sub Messaging that is super-fast because it is fully In-Memory. And, it replicates all messages to another NCache server to ensure that is not data loss in case of any one server going down.

Therefore, if your .NET Core application uses NCache as its Pub/Sub Messaging platform, it will experience super-fast performance and linear scalability because NCache itself is linearly scalable.

Below is an example of how you can use Pub/Sub Messaging provided by NCache in your .NET Core application.

```
private void PublishMessage (string topicName)
{
   ITopic topic = _cache.MessagingService.GetTopic(topicName);
   Order order = Order.GenerateOrder<Order>();

   // Publish message containing “order” with expiry
   Message message = new Message(order, new TimeSpan(0, 0, 15));
   topic.Publish(message, DeliveryOption.All, true);
}

private ITopicSubscription SubscribeMessage (string topicName)
{
   ITopic topic = _cache.MessagingService.GetTopic(topicName);

   // Subscribes to the topic. Message delivered to MessageReceivedCallback
   return topic.CreateSubscription(MessageReceivedCallback);
}

static void MessageReceivedCallback(object sender, MessageEventArgs args) { ... }
```

Figure 7: Using Pub/Sub Messaging in .NET Core Apps

4 Application Data Caching

The biggest scalability bottleneck that your .NET Core application must remove is from the application database. In this area, your applications can achieve high performance and linear scalability through application data caching. The reason for this is simple. Most .NET Core applications deal with a lot of data back and forth from the database.

4.1 Keep Cache Fresh

When it comes to application data caching, the biggest fear that people have is that the cache becomes stale, meaning it contains an older version of the data that has already been changed in the database by another user or another application.

4.1.1 Reference vs Transactional Data

This fear of a cache becoming stale is so strong that the majority of people only cache read-only or static data (reference data). But, this read-only data is only 20% of the total data in the form of lookup tables and other reference data. The bulk of the data in the database is transactional including customers, accounts, activities, etc. And, if you don’t cache this transactional data, then you do not fully benefit from caching.

So, the real benefit of caching comes if you can cache all types of data without the fear of the caching becoming stale. NCache provides a host of features to address this concern.

4.1.2 Sync Cache with Database

The most effective way to keep your cache fresh is to always keep it synchronized with your database. NCache lets you do this to a variety of databases as follows:

1. **Sync Cache with SQL Server:** using SqlDependency and DB event notifications
2. **Sync Cache with Oracle:** using OracleDependency and DB event notifications
3. **Sync Cache with Cosmos DB:** using Cosmos DB Change Feed Processing
4. **Sync Cache with Any Databases (polling-based):** using NCache provided polling based database synchronization.

When you sync your cache with SQL Server, you ask NCache to register itself as a client of SQL Server and then issue a SqlDependency call along with a SQL query-based dataset. Then, when SQL Server sees any changes in this dataset, it notifies NCache about it.

```
private static void CreateSqlDependency (Product product)
{
   string connectionString = "Data Source=localhost;Database=northwind;...";

   // SQL stmt on which the SQL Dependency is created in SQL Server
   string sqlStmt = "SELECT ProductID, ProductName, QuantityPerUnit, UnitPrice " +
                            "FROM dbo.PRODUCTS WHERE ProductID = " + product.Id;

   CacheDependency sqlDependency = new SqlCacheDependency(connectionString, sqlStmt);
   CacheItem cacheItem = new CacheItem(product) { Dependency = sqlDependency };

   string key = "Product:ProductId:" + product.Id; ;
   cache.Add(key, cacheItem);
}
```

Figure 8: Using SqlDependency to Sync Cache with SQL Server

Then, NCache removes this item from the cache so the next time the application needs it, it will have to fetch the latest copy from the database. If you're using Read-through handler (see below) then NCache can also auto-reload the latest copy from the database for you. Below is an example of how you can use SqlDependency to sync your cache with SQL Server.

4.2 Read-through & Write-through Cache

Read-through Cache is a cache that is able to read data from your database by calling a Read-through Handler that you've developed and provided to the cache. Similarly, a Write-through Cache is able to write data changes to your database by calling a Write-through Handler that you've developed and provided to the cache. Write-behind Cache is the same as Write-through except that the database updates are done asynchronously.

Read-through, Write-through, and Write-behind provide a lot of benefits to your .NET Core applications including:

1. **Simplify App Code:** Move persistence code out of your application to the caching tier.
2. **Auto-reload items from DB:** Use Read-through at expiration or DB-sync time.
3. **Faster Writes:** Use asynchronous database writes with write-behind.

Below is an example of how you can use Read-through with NCache.


```
// Read through handler for SQL Server
public class SqlReadThruProvider : Runtime.DatasourceProviders.IReadThruProvider
{
   public void Init(IDictionary parameters, string cacheId) {}
   public void Dispose() {}

   // Get object from the database/data-source based on the key
   public ProviderCacheItem LoadFromSource(string key) {}

   // Bulk-Get objects from the database/data-source based on the keys
   public IDictionary<string, ProviderCacheItem> LoadFromSource(ICollection<string> keys)
   {}
}
```

Figure 9: Using Read-through Handler with NCache

4.3 Search the Cache

Once you're comfortable caching all data, you can start putting a lot of data in a distributed cache. Here you can start facing another peculiar problem of how to quickly and easily find your data. Since most distributed caches are key-value stores, it becomes very difficult to keep track of all your data just through keys.

This is where NCache provides you with a variety of ways to quickly find data from your cache. Examples include:

1. **Group Data in Cache (Group/Subgroup, Tags, Named Tags):** NCache gives you multiple ways to group your data logically and later fetch the entire group in one call. This really simplifies your data management.
2. **Search Data with Queries (SQL / LINQ):** In addition to finding data based on group API calls, NCache also gives you the ability to search the cache for data based on object attributes, groups, Tags, and Named Tags.
3. **Parallel Searches:** Since NCache is distributed in nature, when your application issues a search query or a search API call, that query is run in parallel on all the cache servers. Then results from all of the servers are returned to the client machine (meaning the application server) where they're merged before returning the final results to your application. This really speeds up your cache searches.

Below is an example of how you can use LINQ based queries with NCache.

```
   // Search the cache based on object attributes by using LINQ
   IQueryable<Product> products = new NCacheQuery<Product>(_cache);

   var result = from product in products
                where product.Id > 10
                select product;
   if (result != null)
   {
       foreach (Product p in result1)
       {
          // Process each "product" fetched from the database
          Console.WriteLine("ProductID : " + p.Id);
       }
   }
```

Figure 10: Using LINQ Queries with NCache

5 NCache Architecture for Extreme Scalability

High traffic .NET Core applications cannot afford to go down, especially during peak hours. For these types of applications, there are three really important architectural goals that a good In-Memory Distributed Cache like NCache fulfills.

1. Client Cache (InProc Speed)
2. Linear scalability with fast replication
3. High availability thru Dynamic Clustering

Let me explain each one below.

5.1 Client Cache (InProc Speed)

NCache provides a Client Cache that is a local cache very close to your application. It can either be InProc (meaning it resides inside your application process) or local OutProc. Either way, it provides very fast access to a subset of the cached data that your application on this app server needs at this time. Client Cache at the same time stays synchronized with the caching tier so any data that is changed by other users or applications in the caching tier is immediately propagated to the Client Cache. The client allows you to have InProc speed while still being a part of a very scalable caching tier.

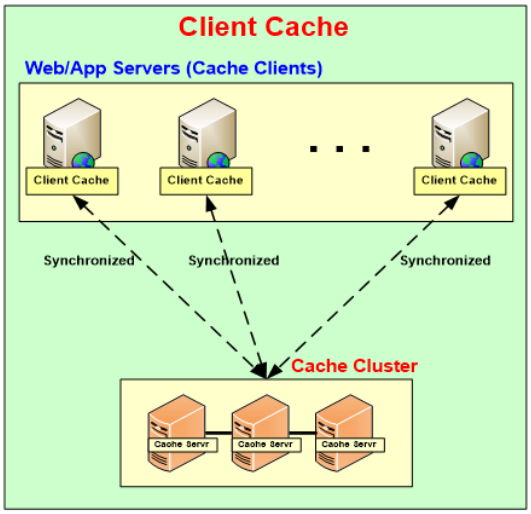

Figure 11: Client Cache Architecture in NCache for InProc Speed

5.2 Linear Scalability with Fast Replication

One of the most important architectural goals of NCache is to achieve linear scalability with data reliability through its caching topologies. Here are some NCache Caching Topologies that help achieve both of these goals.

1. **Partitioned Cache:** NCache partitions the cache based on the number of cache servers and assigns one partition to each cache server. It also adjusts the number of partitions when you add or remove cache servers at runtime. Partitioning is the primary way of ensuring linear scalability because as you add more servers, this caching topology increases the overall storage size and also CPU processing power.

2. **Partitioned-Replica Cache:** In addition to partitioning, NCache also provides replicas for each partition. These replicas reside on different cache servers than the partition itself to ensure that if a cache server goes down along with its partition, then the replica immediately becomes available. This way, data reliability is provided. By replicating each partition only once on another cache server, NCache achieves data reliability without compromising linear scalability.

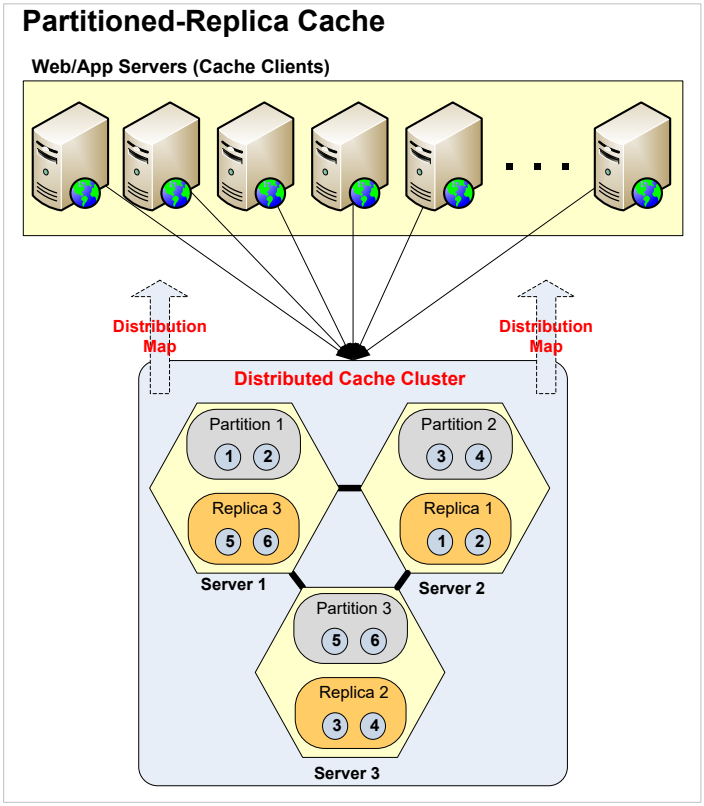

Figure 12: Partition-Replica Caching Topology of NCache

5.3 High Availability for 100% Uptime

One of the most important architectural goals of NCache is to achieve high availability and cache elasticity. It does this through the following architectural capabilities:

1. **Self-healing peer-to-peer cache cluster:** NCache builds a cluster of cache servers over TCP/IP. This cluster has a peer-to-peer architecture that means there are no master/slave nodes and no majority-rule clustering. Instead, each node is an equal peer. This enables NCache to handle situations where any node could go down and the cluster automatically adjusts itself and continues running, and there is no interruption for your application.
2. **Dynamic configuration:** This means you don't have to hard-code things in configuration files. NCache propagates configuration information to all cache clients (meaning your applications) at runtime.
3. **Connection failover support:** If a cache server goes down, the entire cache cluster and all the cache clients are able to continue working without any interruption. The cache clients continue working by interacting with other cache servers in the cluster.

