NCache™

Whitepaper

How to Handle AppFabric End of Life?

By

Alachisoft

January 18, 2017

Table of Contents

Introduction

Distributed caching has been recognized as an important and integral part of high performing .NET web applications. This stands true for those end-user applications which demand quick response-time from the deployed middle tier services or web components. For these reasons, Microsoft introduced AppFabric, an in-memory distributed cache, to provide a performance and scalability boost for applications using the .NET Framework.

AppFabric caching has been mostly used to store ASP.NET session state data and to provide basic object caching capabilities. But even though caching is an integral part of any N-Tier system, Microsoft blogged that AppFabric support ends as of April 11, 2017 for all versions. This is a good incentive for .NET stacks employing AppFabric to move to a better, supported, distributed cache.

Why Find an AppFabric Replacement?

So, what does ending support actually mean? It means:

- No bug fixes
- No security updates
- No new features or feature requests
- No product specific questions and answers
- No free or paid support
- No hotfixes
- No updates to online content (knowledge based articles, etc.)

Therefore, it means you'll be using AppFabric at your own risk. Microsoft won't be liable for mishaps or bugs. Because of this we have to evaluate whether business operations can sustain downtime because of a bug with no quick resolution.

Such questions may drive you toward alternatives. If you are ready for an alternative, there is one which provides seamless migration from your current setup to a .NET distributed cache with minimal code changes, hence introducing no bugs at all. This alternative distributed cache not only supports seamless migration, but it is also faster and provides more features than AppFabric.

Choose NCache as your AppFabric Replacement

NCache is a strong choice as an alternative. Microsoft in fact recommended NCache in their announcement ending AppFabric support. NCache earned this recommendation as it is used extensively within the .NET community, and has been the market leader in .NET distributed caching for 10 years running. NCache provides exceptional performance along with a wide range of features including:

- High availability and self-healing clusters
- Linear scalability during run-time
- SQL querying on cached data
- Synchronization with data sources
- Client caching
- GUI based monitoring and management
- Distributed Locking
- .NET and Java portability

- Big Data computation through MapReduce in .NET
- Official 24/7 support

Other benefits of NCache include ease of use (it is rated by a leading analyst firm as the easiest-to-use distributed cache) strong management and monitoring capabilities through its purpose-built GUI tools and it is available as Open Source (on GitHub) with practically zero cost before and after migration. Open Source and the Enterprise version, with official support, are available on premises and in the cloud (Amazon Web Services and Azure).

How NCache is also better than Redis

While Redis is a good In-Memory cache, the Redis community notes that Redis is not Windows ready, despite availability of a Redis Windows port. Redis is not fully scalable nor does it offer high availability features. Redis' monitoring and managing tool set is CLI only. NCache provides many advantages over Redis as shown in this video link (and this slide share smart guide) including.

- Keeping cache fresh (cache invalidation and DB synchronization)
- SQL searching
- Server side code (read/write through, MapReduce)
- Client cache (near cache)
- Multi-data center support (GEO replication)
- Platform and technology (Windows and .NET native)

To meet needs of any operating environment, NCache provides three ways to migrate .NET applications from AppFabric to NCache; each method having its own set of advantages.

The three migration methods are:

1. NCache Migration Option 1: Wrapper for NCache Enterprise
2. NCache Migration Option 2: Wrapper for NCache OSS
3. NCache Migration Option 3: Direct API Calls

NCache AppFabric Wrappers offer a smoother and more seamless migration, but migrating through direct API calls provides more control and allows you to take advantage of all the features of NCache.

NCache Migration Option 1: Wrapper for NCache Enterprise

For those organizations looking for the easiest and quickest transition from AppFabric, this wrapper provides the easiest migration ever. Only three minor changes are required in the .NET application to make it NCache compliant without introducing any possible bugs at all. These changes are:

1. Add NCache in the App.config or Web.config

```
<appSettings>
  <add key="region1" value="myPartitionedCache"/>
  <add key="Expirable" value="True"/>
  <add key="TTL" value="5"/>
</appSettings>
```

2. Add references to the NCache Library Alachisoft.NCache.Data.Caching.dll and remove the AppFabric Libraries

3. Change Microsoft.AppFabric NameSpaces to Alachisoft.NCache. For example replace:

```
using Microsoft.ApplicationServer.Caching.Client;
using Microsoft.ApplicationServer.Caching.core;
```

with:

```
using Alachisoft.NCache.Data.Caching;
```

And that's it! No need for any other code changes are required within the .NET application.

NCache Migration Option 2: Wrapper for NCache OSS

Using NCache Open Source allows you to migrate from AppFabric without cost since both products are free. NCache Open Source has a more limited set of features as compared to the Enterprise edition, but without any degradation in the core cache performance.

For ease of migration, a separate version of the AppFabric wrapper is provided for NCache OSS, which is compatible with the open source API. Given fewer features in NCache Open Source, the NCache Open Source wrapper works around the missing features which include:

- Cache item versioning
- Groups
- Cache based events

Cache Item Versioning

The NCache Open Source wrapper for AppFabric overcomes the challenge of maintaining item versioning by encapsulating every object within another class. This is done by introducing a new class named MetaDataCapsule in the modified wrapper.

```
namespace Alachisoft.NCache.Data.Caching.Util
{
    [Serializable]
    class MetaDataCapsule
    {
        private object value;

        public ItemVersion CacheItemVersion { get; set; }
        public string Group { get; set; }
        public object Value { get { return this.value; } }

        private MetaDataCapsule(object value, ItemVersion version, string group)
        {
            this.CacheItemVersion = version;
            this.Group = group;
            this.value = value;
        }

        public static MetaDataCapsule Encapsulate(object value, string region)
        {
            return new MetaDataCapsule(value, 1, region);

        }
 } }
```

This helps to maintain the item version of an object and carry other useful information if required. The class is marked as Serializable as it needs to travel over the network.

To ensure item-version and object consistency throughout all the multiple instances of different or similar applications, distributed locking is used. Distributed locking is a feature of NCache which allows multiple applications to access and modify the same resource without compromising data consistency and avoiding any possible race conditions.

```
LockHandle lockhandle = null;
MetaDataCapsule oldMetadata;
bool lockAcquired = GetLock(_key, ref lockhandle, out oldMetadata);

if(!lockAcquired)
{
    _NCache.Unlock(key);
    throw new DataCacheException("Unable to acqurie lock to update Item Version");
}

//Return in case of version provided
if (oldVersion != null && oldMetadata.CacheItemVersion == oldVersion._itemVersion)
{
    _NCache.Unlock(key);
    return null;
}

if (lockAcquired)
    (_item.Value as MetaDataCapsule).CacheItemVersion = ++metadata.CacheItemVersion;

_NCache.Insert(_key, _item, lockhandle, true);
```

In summary, on every Put operation:

1. The item to-be-updated is fetched and locked (with expiration)
2. The returned value is cast as MetaDataCapsule and the current item version is extracted.
3. ItemVersion is incremented by 1
4. Data is inserted back in the cache.
5. If everything works smoothly, release the lock with INSERT overload (key, item, lockhandle, releaselock)
6. For other exceptional scenarios, unlock the item followed by throwing an exception.

Locking in the above scenario will ensure no other application can update or remove the item if another application is trying to modify the same key.

In the above code, there is a method named GetLock, the implementation is as follows.


```
/// <summary>
/// Tries to acqurie lock and return LockHandle and MetaDataCapsule object.
/// If LockHandle is provided uses that instead.
/// <para>Default Lock Timeout (default 5 seconds), Retry Count (default 3)
/// and retry interval (default 100 ms) is set when Cache Handler is instantiated.
/// </para>
/// </summary>
/// <param name="key">Formatted Key i.e compile from formatter</param>
/// <param name="lockHandle">Either provide a lock or keep it null to acquire a new lock
/// </param>
/// <param name="metadata">The returned object for internal use</param>
/// <returns>true if lock was acquired</returns>
internal bool GetLock(string key, ref LockHandle lockHandle,
                        out MetaDataCapsule metadata)
{
    //Item is locked
    int count = _retries;
    while (count != 0)
    {
        //If lock was provided attempt to acquire the lock from the given handle
        if (lockHandle == null)
        {
            lockHandle = new LockHandle();
        }

        object obj = _NCache.Get(key, _defaultLockTime, ref lockHandle, true);

        //obj is null of the lock was not acquired
        if (obj == null)
        {
            count--;
            Thread.Sleep(_retryTimeOut);
        }
        else
        {
            metadata = obj as MetaDataCapsule;
            return true;
        }
    }

    lockHandle = null;
    metadata = null;
    return false;

}
```

The GetLock method will try to acquire the lock thrice. If failed it will return "null" letting the calling method handle this exception.

Thus with the help of Distributed Locking, the AppFabric wrapper for NCache Open Source maintains the cache item version inserted by the client applications, including exception cases.

Limitation of NCache's Open Source AppFabric Wrapper

Due to the lack of some enterprise features in the open source edition, the following functions are marked "obsolete" and marked with "throw compilation error" to ensure a code change and avoid any surprises.

1. Any API with tags
2. Clear Region (not supported because of Groups)
3. GetObjectsInRegion (not supported because of Groups)
4. Cache/Region Level Callbacks

Therefore, if your application heavily depends on these features, selecting NCache Enterprise version might be the best option to allow a smoother, seamless and bug free migration.

NCache Migration Option 3: Direct API Calls

For developers who want to take full advantage of many NCache Enterprise features, you may prefer to directly call the NCache API instead of using the wrapper. You can also use the 'unwrap method' from the AppFabric wrapper to directly work with NCache as well.

Either way, it is recommended to conduct your migration in two phases. In phase 1, continue to use your AppFabric features, but replace your cache with NCache. In Phase 2, implement additional, new NCache features that were not available in AppFabric.

Phase I: Use AppFabric features only

In this phase your focus is just to replace AppFabric. Therefore, understanding how these two distributed caches behave is important (see more details below and in the reference section at the end). As mentioned, AppFabric stores data in regions on specific machines. On the other hand, to replicate your region data in NCache you can create a cache for every AppFabric region, or you can create NCache groups to hold AppFabric region data.

1) Regions to Groups

NCache Groups/Sub-Groups provides various ways of grouping cached items. As regions reside inside a named cache, NCache uses groups to reproduce this behavior. The only difference is that in AppFabric a key needs to be unique only within a region and not within the entire named cache. Whereas in NCache, the key needs to be unique across the whole cache and it doesn't matter which group it belongs to. This hindrance can be quickly conquered if region names are pre-pended with every key.

For example:

In AppFabric, a basic Add operation looks like the following:

```
// AppFabric API
regionCache.Add(stringKey, stringValue);
```

And in NCache the basic command is no different, except a region name could be specified as a group in the command:


```
// NCache API
cacheInstance.Add(regionName + ":" + stringKey, stringValue, regionName);


// Where regionName is as Group
```

Here, every key is pre-pended with the region name to achieve uniqueness among the cache keys. A group is assigned to the key for faster retrieval or to facilitate data retrieval using SQL queries when performing search operations.

2) Regions as Separate Caches

The other way of grouping cached items is to drop the region-to-group correlation completely and use a separate cache for every region. To obtain the same result, where every region is assigned a separate cache, a map of region-names corresponding to caches can be stored.

```
// NCache API
cacheInstance = map.get(regionName);
cacheInstance.Add(stringKey, stringValue);
```

In this example, there is no need to pre-pend the region key to the cache key since caches in NCache are mutually exclusive even if they reside in the same cluster.

If your application must deal with many regions (e.g. more than 15), then it is better to use regions as groups since 15 caches per cluster is a recommended maximum number of caches (even though a cluster can logically host an infinite number of caches).

Phase II: Add advanced NCache features

In the next phase, you can take advantage of the features that NCache provides which AppFabric and Redis lack. Features such as:

1) Keeping cache fresh (cache invalidation and DB synchronization)

Database synchronization is a very important feature for any good distributed cache. Since most data being cached is coming from a relational database, there are always situations where other applications or users might change the data and cause the cached data to become stale.

2) SQL searching

Applications often want to fetch a subset of this data and if they can search the distributed cache with a SQL-like query language and specify object attributes as part of the criteria, it makes the distributed cache much more useful for them.

3) Server Side code

Many people use distributed cache as "cache on the side" where they fetch data directly from the database and put it in the cache. Another approach is "cache through" where your application just asks the cache for the data. And, if the data isn't there, the distributed cache gets it from your data source. The same thing goes for write-through.

Along with this, NCache also supports MapReduce which enables Big Data Analytics in In-Memory data, producing results in near real time.

4) Client cache (near cache)

Client cache is simply a local InProc/OutProc cache on the client machine but one that stays connected and synchronized with the distributed cache cluster. This way, application performance really benefits from this "closeness" without compromising data integrity.

5) Multi-Data Center support (GEO replication)

WAN replication is an important feature for many applications are deployed in multiple data centers either for disaster recovery purpose or for load balancing of regional traffic.

The idea behind WAN replication is that it must not slow down the cache in each geographical location due to the high latency of WAN. NCache provides Bridge Topology to handle all of this.

Conclusion and Additional Resources

Support for Microsoft's AppFabric ends in April 2017. This white paper presents three possible ways to migrate .NET applications from AppFabric to NCache, and advantages of making this change. NCache is a stable, full featured, distributed .NET cache available as Open Source in Azure and Amazon Web Services, as well as on the NCache website. Full support is also available.

Using the free NCache AppFabric wrapper for either NCache Open Source or NCache Enterprise, offers the easiest, no-coding, bug free migration method. Migrating by directly calling the NCache API provides more data control and enables all NCache features. All migration methods are described in this white paper.

Please check these additional resources to meet your evaluation and product needs.

- NCache page about migration to AppFabric
- NCache edition comparison
- Download NCache
- NCache OSS free AppFabric wrapper
- NCache Enterprise free AppFabric wrapper
- Technical NCache /AppFabric Migration Guide
- NCache OSS in Azure Cloud
- NCache OSS in AWS Cloud
- NCache vs. AppFabric
- NCache vs Redis
- NCache details

Reference

Accessing Regions in AppFabric

Regions in AppFabric are logical containers of key-value pairs stored in an instance of a cache, which is bound to one server machine. Because of this, data inside regions cannot be distributed. This architectural structure claims to enable faster fetches. But it significantly reduces overall performance compared to a distributed cache.

NCache is a distributed cache at its core. All the data inside NCache is distributed, and a backup-copy is maintained in another physical machine if the Partition Replica (PR) topology is used. PR provides world class, super-fast performance, and fail-over replicas for zero data loss.

Thus, regions in AppFabric are configured as separate caches in NCache without compromising any of the data fetching capability. The AppFabric wrapper takes over this complexity and automatically stores all the region data in the provided cache name. The only requirement is that caches in NCache should pre-exist. You create caches via NCache Manager, a GUI based cluster management tool. Once the caches are created, data residing in AppFabric regions are distributed to the specified NCache cluster(s). Despite the data being truly distributed, there is no compromise in your ability to execute advanced searches, including SQL querying (SQL querying is not available with AppFabric at all).

Creating & Configuring Caches

The other noticeable difference between NCache and AppFabric is the way both products are configured. AppFabric is configured by the application at runtime (even the cache creation) and in contrast NCache is configured before the cache is started. For example, cache topologies cannot be changed at runtime whereas other configurations for example, cache default expirations etc., can be changed later and those are called as hot-apply-able. Even though NCache can be managed using the .NET application (including cache creation) but it is preferable to manage from the NCache Manager - A dedicated single tool for managing the cache clusters. A cache can also be created, configured and maintained from the CLI as well.

While NCache AppFabric-wrapper provides the easiest way to migrate .NET applications to NCache, which is also with the least code changes possible, some developers still prefer to take advantages of all the features that NCache must offer. That is where directly calling the NCache API helps. There is also the option to use both methods by calling the Unwrap method from the AppFabric wrapper.

Caching Topologies

The AppFabric cluster is not completely dynamic. Dependency on "lead hosts majority rule" means cluster can go down very easily if even one lead host goes down. These lead host nodes also resemble "master" and "slave" architecture and is therefore not fully peer to peer as well.

NCache whereas is highly dynamic and lets you add or remove cache servers at runtime without any interruption to the cache or your application. Data is automatically rebalanced (called state transfer) at runtime without any interruption or performance degradation. NCache clients keep on communicating with cache servers, independent of state of server. Clusters ensure execution of client operations even when data balancing is in process.

This means that even for NCache, there is no "master" or "slave" nodes in the cluster. There is a "primary coordinator" node that is the senior most node. And, if it goes down, the next senior most node automatically becomes the primary coordinator. All of this happens without any interruption to the client operations.

