DISTRIBUTED CACHING IN A MULTI-SERVER ENVIRONMENT

A study of Distributed Caching mechanisms and an evaluation of Distributed Caching Platforms available for the .NET Framework.

Robert Herber

robert.herber@telia.com

Abstract

English

This paper discusses the problems Distributed Caching can be used to solve and evaluates a couple of Distributed Caching Platforms targeting the .NET Framework.

Basic concepts and functionality that is general for all distributed caching platforms is covered in chapter 2. We discuss how Distributed Caching can resolve synchronization problems when using multiple local caches, how a caching tier can relieve the database and improve the scalability of the system, and also how memory consumption can be reduced by storing data distributed.

A couple of .NET-based caching platforms are evaluated and tested, these are Microsoft AppFabric Caching, ScaleOut StateServer and Alachisoft NCache. For a quick overview see the feature comparison-table in chapter 3 and for the main advantages and disadvantages of each platform see section 6.1.

The benchmark results shows the difference in read performance, between local caching and distributed caching as well as distributed caching with a coherent local cache, for each evaluated Caching Platform. Local caching frameworks and database read times are included for comparison. These benchmark results are in chapter 5.

Swedish

Denna rapport tar upp de problem distribuerad cachning kan lösa och utvärderar några distribuerade cachningsplattformar som fungerar med .NET Framework.

Grundläggande begrepp och funktionalitet som gäller för alla distribuerade cachningsplattformar tas upp i kapitel 2. Vi tar upp hur distribuerad cachning kan eliminera synkroniseringsproblem med flera lokala cachar, hur ett cachningslager kan avlasta databasen och förbättra systemets skalbarhet, och också hur minneskonsumtionen kan minskas genom att lagra data distribuerat.

Ett antal .NET-baserade cachningsplattformar har utvärderats och testats, dessa är Microsoft AppFabric Caching, ScaleOut StateServer och Alachisoft NCache. För en snabb överblick finns det en tabell med en jämförelse av tillgängliga funktioner i kapitel 3, de viktigaste fördelarna och nackdelarna med varje plattform finns beskrivna i sektion 6.1.

Mätresultaten visar skillnaden i läsprestanda, mellan lokal och distribuerad cachning samt distribuerad cachning med en koherent lokal cache, för varje cachningsplattform. Lästider från databas samt lokala cachningsramverk finns med för jämförelse. Dessa mätresultat finns i kapitel 5.

Contents

LIST OF FIGURES

1 INTRODUCTION

1.1 BACKGROUND, PURPOSE AND DELIMITATION

Devisor Innovation AB is a consulting company that specializes in helping customers with system development in Microsoft's .NET platform. Apart from system development Devisor is qualified in the areas of procurement, project leading and business intelligence.

Most systems today work against a database, everything from simple interactive websites to more complicated business systems. As these systems grow, caching is used to improve performance and lessen the load on the database server. When the system is small it works quite well to use a local cache on each client to perform this, but as multiple applications and/or servers read and write to the same database new problems arise, namely synchronization problems between these local caches.

As long as only a single application accesses the data it can be stored in its own local cache, so that when data is updated the application can update data in both the database and cache at once. But when another application is added to the system, which works towards the same database, we can no longer be sure if the locally cached data hasn't already been updated in the database.

When the number of applications increase another problem with the local caching scheme becomes apparent, the amount of memory used for caching grows. For applications caching equivalent data the total amount of memory required increases linearly. There is also a limit to how large the datasets that are cached can be determined by the amount of memory on the application server.

Depending on how heavy the traffic is on the database server, it might decrease in performance. A single server always has a limit of what it can handle in terms of requests per second, and a problem when it comes to database servers is that you can't in a simple way just add another server to distribute the load. A term for this is that a database doesn't *scale* well.

Another problem, more related to Software Engineering than scalability, might be that there is not a simple interface for the programmer to access the caching features when implementing the application. In the specific system we've worked with in this project ASP.NET built-in caching is used alongside the Caching Block of the Microsoft Enterprise Library as well as ASP.NETs Session State which complicated the interfacing with the Caching Framework because of the various ways to access the cache for the programmer.

The purpose of this project is to investigate possible solutions to these problems in the form of a Distributed Caching Platform that is to be implemented into the existing system within a year.

The project is delimited to apart from this report create a proof-of-concept implementation, but the system is complicated enough to require developers with solid knowledge and experience of the target system to make the final implementation.

1.2 PROBLEM FORMULATION

The problems to be solved in the system we're working with were these:

- Cached data is not synchronized between applications when the database is updated from another application.
- Inefficient use of memory, as each of the applications store a copy of data on their own that could theoretically be shared between multiple applications.

- It would be a bonus if the use of caching could be homogenized in the system.

The goals of the project were to:

- Analyze the caching technique used in the system today and the positive and negative aspects of the current implementation.
- Investigate how Microsoft's Distributed Caching Platform included in Windows Server AppFabric (previously known as Velocity) can help solve the problem with synchronization and optimize the use of memory.
- Investigate if there are any alternative Distributed Caching Platforms that can help solve the problem and analyze the positive and negative aspects of these products.
- Implement a proof-of-concept using one or multiple Distributed Caching Platforms.
- Discuss problems that might occur when implementing the caching component in an existing system.

1.3 TERMINOLOGY

In this report these terms will be common, and when used without further explanation these are the intended interpretations:

Cluster	The collection of cache hosts that store cache data. The cluster may be peer-to-peer-based or centrally controlled by a manager.
Host	A node in the cache cluster. It refers to the service running on the cache server which keeps cached items in its memory.
Client application	The application that reads and writes to the distributed cache through a client API. There can be multiple clients per application server.
Cache server	The physical server the cache host runs on.
Cached item	The item stored in cache including metadata like expiration options and tags as well as the raw object data.
Cached object	The raw object data that is stored in the cache.

1.4 RESEARCH METHOD

As the aim of this thesis work was to provide both a proof-of-concept as well as to identify the issues that are important to address before implementing distributed caching into the target system, the work has gone back and forth between collecting knowledge and using that knowledge in the proof-of-concept implementation.

The first thing to do was to find out which Distributed Caching Platforms that where available to solve this specific problem. This was done by reading articles that discuss distributed caching and browsing the web; and this way five possible solutions targeting the .NET-framework where discovered: Microsoft AppFabric Caching, ScaleOut StateServer, Alachisoft NCache, memcached and SharedCache.

Out of these five SharedCache and memcached were disqualified from further investigation in this project because of their open source nature. Memcached is definitely a proven Distributed Caching Platform, and while there are multiple ported .NET client libraries none of them seemed better than the other and all where open source. In a commercial system like the target system of this project it is important to know that external components will stay supported in the future, thereof the decision to exclude open source alternatives.

After this three caching solutions remained, this was a suitable number of platforms to investigate further. The main questions asked where now:

- Which features should be documented in the evaluation?
- What is the equivalence of one specific feature on one platform on another of these platforms?

These questions have been important to understand as well as to categorize the fundamental principles of distributed caching (described in detail in the section “Distributed Caching Platforms”) and to identify the unique features of each evaluated caching platform (described in detail in the section “Evaluated Distributed Caching Platforms”).

After answering these questions, the first iteration of implementation took off. The three Distributed Caching Platforms where installed, without any major issues. It was quite obvious that a generic interface towards all the caching platforms would simplify the evaluation, and it might also provide a base of how to simplify the caching interface towards the application programmer. A detailed description of how this was designed is available in the “Implementation” section.

Apart from collecting information about the features that are available for each platform, it would be interesting to measure how well the Caching Platforms perform with their shared features. These benchmarks are included in the “Results” section.

2 DISTRIBUTED CACHING

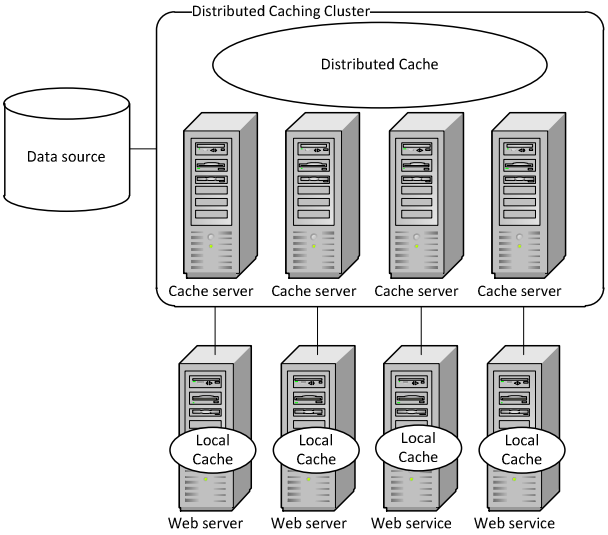

FIGURE 1: DISTRIBUTED CACHING CLUSTER

Let us begin with a metaphor to understand the intended use of distributed caching. Trains, airplanes and cars are all important means of transportation. The database is like a freight truck that can transport goods at a stable speed to a certain level, i.e. when it becomes full and some of the goods have to wait for the next transport. A local application cache is like an airplane; it has great speed and thrives when speeding the clouds alone but it's not particularly well suited for large goods. A distributed cache is like a freight train that can add new wagons as required. It can transport large goods of arbitrary amount at a stable speed.

In most cases a DBMS (Database Management System) is used for persistent storage of data. The trouble of a DBMS lies in handling scalability, we can't in a simple way set up another database server and load-balance the requests between the servers. Sooner or later the limit of what a DBMS can handle when it comes to scalability is reached. Keep in mind that a DBMS isn't necessarily slower than a Distributed Cache, in fact many DBMSs features quite potent built-in caching. Distributed caching should be considered if DBMS scalability is a problem.

On the other hand we have local caching, which keeps the data in the memory of the client application. The performance of working with a local cache is *always* better than working with a DBMS or distributed cache. The problem with local caching is as stated earlier synchronization between multiple applications and in some cases insufficient memory. Distributed caching should be considered as a solution for any of those two problems.

To sum it up:

- A distributed cache can scale indefinitely. As the number of data requests increase new caching servers can be added to the cache cluster to maintain performance where the performance of a database server alone would decrease.
- A distributed cache provides the ability to update the cached data for all clients at once, resolving the synchronization issue.
- A distributed cache provides the ability to serve the same data to multiple clients with only one copy of the data in-memory.

2.1 LOCAL AND DISTRIBUTED CACHING

Caching is all about increasing performance, so distributed caching should only be used if it directly or indirectly improves the performance of the system. The performance of distributed caching is drastically lower in comparison to a local cache (see benchmarks in the "Results" section); and the cause for this is because of serialization (discussed in Section 4.2.1) and also increased network traffic. If data *can* be cached locally, it should *always* be cached locally.

The true power of distributed caching is unleashed when it works together with a local cache, which all of the Distributed Caching Platforms that are investigated in this report can do natively. This way the outstanding performance of a local cache can be combined with the distributed cache which can handle synchronization as well as storing the largest amount of data. All of the Distributed Caching Platforms evaluated has the option to check on every read whether the data in the local cache is the most recent also in the distributed cache so that no old data slips through.

2.1.1 COHERENCY

Coherency is the issue of making sure that the local caches return the same versions of the items as the distributed cache contains. Coherency is always handled by the caching platform, but can sometimes be configured. How to handle coherency is a trade-off between performance and actuality of the data. There are three common ways to ensure coherency:

- To ensure guaranteed coherency a check has to be made on each data access. An identification of the items state has to be sent to the distributed cache, and the response has to contain either a confirmation that the local copy of the item is the latest, or if not, the updated item.
- By using event notifications the client application could be notified when an object is updated in the distributed cache. This is not immediate because the client has to poll the cluster; none of the evaluated caching platforms supports push notifications from cluster to client.
- The third way is to remove items from the local cache a certain period of time after it was added. Compared to the notification-based check an additional issue is that items are removed from the local cache even when they haven't been updated.

2.2 WHICH DATA TO CONSIDER FOR CACHING

By dividing data into well-defined groups, it's easier to decide how to handle the caching of that particular data. Cached data is commonly divided into the following three groups[1]:

Classification	Description	Example	Where to cache	Note
Reference	Never or seldom updated, shared data.	Language localization data or product catalog data.	Use a distributed cache if periodic synchronization is required. Use a local cache if memory allows it.	
Activity	Frequently updated, non-shared data.	Session data or shopping cart data.	Local cache if memory allows it.	All evaluated Distributed Caching Platforms can act as an ASP.NET session provider.
Resource	Frequently updated,	Auction bids or inventory/stock	The data is in need of synchronization; therefore a distributed cache might be	If the updates to the database slow down the system, write-behind

	shared data.	data.	used. Use a local cache if the memory of the application server allows it. Decide whether the cached data requires instantaneous synchronization or if periodic synchronization will suffice.	(which we'll talk about right after this) might be a way to lessen the load on the database.

2.3 EXPIRATION AND EVICTION

The purpose of a cache is to increase response times when working with frequently used data. To control how cached items should behave expiration and eviction settings can be specified[2], and it's used both with local and distributed caches.

Expiration means that items should be removed from the cache after a certain amount of time, the purpose of this is to remove items that aren't accessed any longer. In some systems it can be used to handle coherency, when the data source is updated without the cache being updated, which is the case for all local caches. The common way to specify expiration behavior is with a timeout accompanied with a flag specifying if the item should expire after the *absolute* time or by a *sliding* time, i.e. resetting the timeout on each data access.

Eviction is used by the cache if there isn't enough memory to store new objects. To control this most caching APIs makes it possible to specify a priority for each cached item. The items with lowest priority are evicted first from the cache.

2.4 CONCURRENCY

Concurrency is about handling the order in which to update cache items, and it's an issue when multiple clients update the same items. There are two common approaches to handle concurrency[3]:

The *pessimistic* model[4] means that client can put a lock on an item to ensure that no other client can tamper with that particular item until the client unlocks it.

The *optimistic* model means that version tracking of items are used to make sure the client updates the item based on the latest version. If two clients try to update the same item only the first update will succeed, as the second update is no longer based on the latest version of the item.

Concurrency doesn't need to be handled, but it is *recommended* if frequent updates are done to the same items from multiple clients. All evaluated caching platforms support concurrency handling in their API.

2.5 BACK STORE

An important consideration before implementing a Distributed Caching Platform is where in the application architecture the cache access should reside. Now we'll talk about the architectural design where database access is handled through the cache. This enables certain interesting features[5], but there are of course other ways to design a streamlined data interface towards the application.

Read/write-through is a technique to handle the communication between the database and cache, where the programmer works toward the cache at all times, which in turn requests/submits data to the database when required. This requires an implementation of database connection and translation of the requested cache keys to a database query. The benefit of this approach is first and foremost that it simplifies the data interface.

It also enables the cache to work against the database asynchronously in a way that's called *write-behind*, which is a built-in feature of two out of three of the evaluated caching platforms. Write-behind makes the cache write updates to the database with a configurable interval, and this means that cache clients doesn't have to wait for the database update to complete before continuing on its next task. This approach can also result in that some updates never need to be written to database, as the records can be updated before being submitted to the database and lighten the load on the database.

Refresh-ahead is a related technique that means that the Caching Platform updates data asynchronously that is about to expire. This means that the cache won't always need to access the database when a client requests an expired item, and in turn it increases response times. It can be a good way of keeping the cache populated, which should be a goal on dedicated caching servers.

If all data is contained in the cache the use of read/write-through also simplifies the data handling for the programmer, as he doesn't have to consider whether to work against the cache or the database. It might not be as beneficent in a system where only part of the data is cached.

2.6 Dedicated Caching Servers

It is recommended to use dedicated caching servers so that each cache host can use the full memory and CPU capacity of the server, but it is not required and it really depends on how the cache cluster should be used. Keep in mind that paging should be avoided at all costs because it'd slow down both the distributed cache and any other applications running on the same server, and that would probably eliminate the advantage of using a distributed cache at all.

2.7 Cluster management

Distributed Caching Platforms can be built around a peer-to-peer principle or with some kind of server controlling the flow of the cache cluster. The way this is handled is individual for each platform, and is important for availability if cache hosts go offline for some reason. The peer-to-peer alternative is generally more fault-tolerant.

2.8 Physical Organization of Cached Data: Topologies

Distributed Caching Topologies[6] define how the cluster behaves internally and how it distributes the cached data among the cache hosts. There are a couple of caching topologies that are commonly used in distributed caching, but the topologies available for individual systems vary and can be a combination of these general topologies. The topologies available might for some scenarios be an important aspect in the decision of which Platform to use.

Remember that the client applications see the cluster as a single unit, so the client-side implementation is the same for all topologies; it is simply a question of cluster configuration.

2.8.1 Partitioned Topology

A partitioned (sometimes called *distributed*) topology means that each piece of data only is located at one single host in the cluster. This is the classical topology and excellent from a scalability point of view, because each item is stored only once there is no replication between nodes which is the case in the two other topologies. The read performance is also stable.

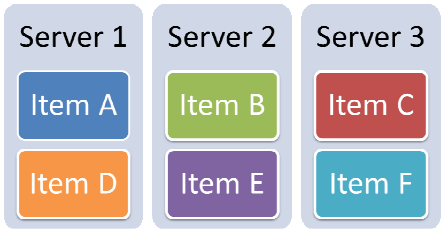

FIGURE 2: PARTITIONED CACHE

But there is a drawback and that's if one of the cache hosts for any reason fails, then its data will be lost to the cluster. In many cases this is acceptable, if the database has the primary data and the cost to fetch the data again isn't too expensive. In some cases though, for example if we don't persist the data to the database at once, we need to make sure data is not lost. Figure 2 shows how cached items are distributed among the servers, as you can see each item occurs only once across the cluster.

Advantages	Disadvantages	Replication factor
Scales indefinitely	Availability	1x

2.8.2 Replicated Topology

A replicated topology[7] means that the cache replicates each data item in the cluster to all hosts in the cluster; it's quite the opposite of a partitioned topology. This means that the read access is fast as long as one of the cache hosts is close to the cache client, and it could be important to keep read performance high in a scenario where many clients access the same data.

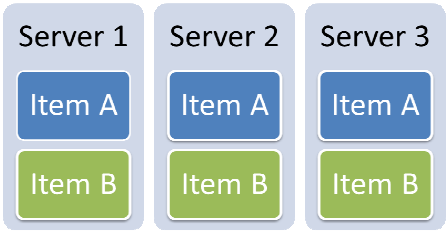

FIGURE 3: REPLICATED CACHE

The write performance however is not very good. This is because every update to the cache is replicated to all hosts in the cluster. This makes replicated caches effective for data that is not modified very often, because of its horrible scalability properties it should not be used in large clusters.

Because every host contains all the data, the availability of the cached data is very high; as long as any one host is online in the cluster the data will be available. Figure 3 shows how cached items are distributed among the servers with the Replicated Topology, as you can see the items are replicated on each host in the cluster.

Advantages	Disadvantages	Replication factor
Availability, read performance	Write performance, poor scalability	Increases linearly with the number of cache hosts

2.8.3 High Availability Topology

A high availability cache can be considered as a combination of the replicated and partitioned topologies. Every cache item is replicated to one (or more) additional hosts. All updates and reads are done towards the primary copy of the item, and replicated to the other host. In effect this means that no read speed is gained from the replica, as with the replicated cache.

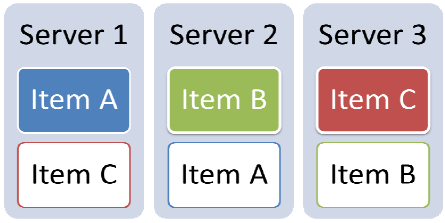

FIGURE 4: HIGH AVAILABILITY TOPOLOGY

The replication cost is a factor, as well as for a replicated topology, but because the number of replicas is fixed it doesn't affect scalability when the number of cache hosts increase.

As with replicated caches, the availability is high, as long as not both the active and passive hosts becomes unavailable for a particular item it'll still be available. Figure 4 shows an example of items distributed across the cluster according to the High Availability Topology, as you can see each item has a copy on another host in the cluster.

Advantages	Disadvantages	Replication factor
Availability; scales indefinitely	Slight performance cost of replication	2x (or more, depending on configuration)

2.9 Logical Organization of Cached Data

2.9.1 Named Caches

Most caching APIs, it doesn't matter if they're distributed or not, support Named Caches[8]. Named Caches is a way to access multiple caches side-by-side through the same API. Maybe we want different caches for different parts of an application, or for two entirely different applications. It can be a way to group data in the cache. Figure 5 shows the default cache along with two named

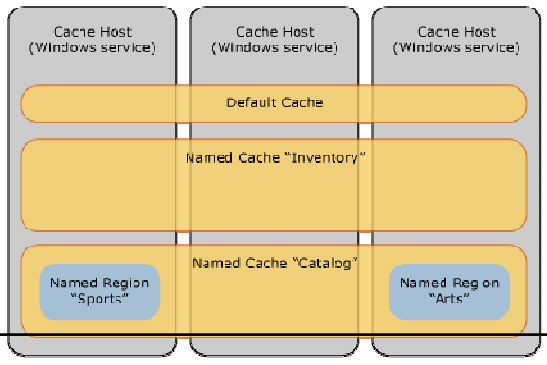

FIGURE 5: REGIONS AND NAMED CACHES

caches distributed across the cluster in Windows AppFabric Caching.

In the case of distributed caching, we might have two different applications that both want to utilize the same distributed cache, Named Caches offers a great way to separate the data of the two applications.

If Named Caches are not necessary in the implementation it is often sufficient to use the default cache of the caching API.

2.9.2 REGIONS

Regions[9] (sometimes defined as groups) are used to group data of a specific type inside a named cache. Regions are often bound to a specific cache host, and might therefore behave differently than other chunks of data stored in the cache. Some client APIs uses regions to provide fast tag-based searches or flushes because of its spatial locality. Figure 5 shows how Regions are bound to a single host in AppFabric Caching.

2.10 SERIALIZATION

Serialization is the process of converting an object in memory to a sequence of bytes that can be stored to a file or transferred over the network; deserialization is its opposite and converts the sequence back to an object. Serialization is required to use any of the distributed caches evaluated in this paper. Both the serialization and deserialization is done on the client side, before the data is sent over the network.

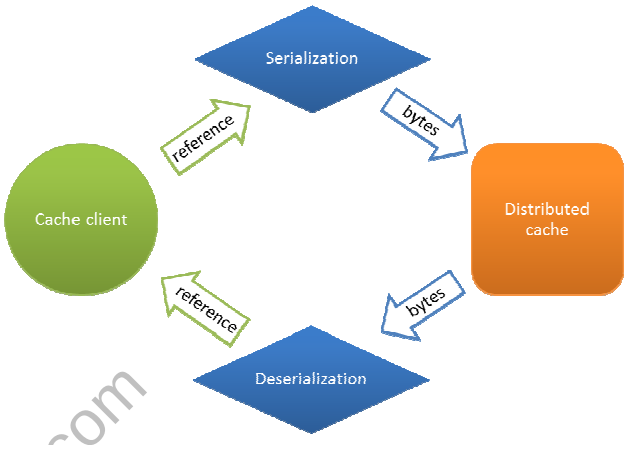

FIGURE 6: THE SERIALIZATION PROCESS

Serialization is meant to be programming language-independent, and .NET provides standard implementations to serialize data both to binary and XML data, the latter for the reason of human-readability which is not of much interest when it comes to distributed caching. When using the built-in serialization the programmer can mark a class with the *Serializable* attribute, which makes an attempt to serialize that object as well as all members of that object. To skip a field in the serialization process it can be marked with the *NonSerialized* attribute, which is interesting for fields that are easily recreated in the deserialization process.

It is important to skip serialization when possible out of a performance point of view, as everything that doesn't need to be serialized saves cost in both the serialization/deserialization process as well as when transmitting data across the network and memory cost in the distributed cache. Consider that if object A contains a reference to object B and we cache both objects, we'll have double copies of object B in the cache.

For an object without references serialization is simple, but that is not the case for objects that has references to other objects. A choice has to be made whether to serialize all underlying references, or choose to skip some of these references. Consideration has also to be made to the serialization of all the objects the underlying references themselves reference to. As you might understand, this requires an in-depth plan for how to handle each object that's to be contained in the distributed cache. Serialization is an issue for all distributed caching solutions, as the data has to be sent over the network and that can't be done without serializing the objects, unlike local caching solutions where the data is contained in memory as objects so no serialization is needed.

Apart from the standard .NET serialization the programmer can choose to construct his/her own per-object solution by implementing the *ISerializable* interface. When wanting a highly customized or company-specific serialization, perhaps including encryption, this can be an interesting option. It should be noted that .NET serialization is not perfect in terms of size or performance, so that might also be a reason for an alternative solution[10]. The programmer has to

provide an implementation of *GetObjectData* for serialization and a constructor for deserialization so the object can be restored to its original state.

Important: Serialization *must* be properly handled for the objects to be stored in a distributed cache; attempts to cache data that is not serializable will cause run-time exceptions.

2.11 Related Information

A group of students at the University of Gothenburg has made a similar evaluation[11] targeted at low-level solutions to deal with distribution of data; their research is based on the .NET-platform, just as this paper. They discuss methods like Inter-process Communication, Shared Memory, Database Caching as well as .NET Remoting. It's probably most interesting if you're interested to know how a Distributed Caching Platform works under the hood or just want some specific features.

Thomas Seidmann at Slovak University of Technology has written a paper on implementing a Distributed Shared Memory using .NET Remoting[12].

Cuong Do Cuong held a presentation[13] on June 23, 2007 about YouTube's handling of scalability, which is quite exciting considering its rapid growth. It's not specifically about Distributed Caching, but he talks about the type of problems that Distributed Caching can be used to solve. Among other things he states that YouTube has used the memcached platform for Distributed Caching.

Thorsten Shuett presents[14] his own distributed system that's using the same principles as Distributed Caching (even though the terminology is a little different). He gives a quite in-depth description of how it works. It might give you some ideas on how a distributed cache can work internally.

2.12 Related Products

In this paper we'll delve into a couple of commercial .NET-based Distributed Caching Platforms. Here follows a couple of other alternatives, among them .NET alternatives that aren't looked at extensively later in this paper as well as caching platforms that targets other programming languages like Java or C/C++.

Probably the most known distributed caching platform is Memcached. Some of its users are Youtube[11], Facebook[15], Twitter[16] and Zynga[17] which all have massive user bases that require top-notch scaling of their sites and applications. There are .NET client libraries available for Memcached, but none of them are commercial. For those interested in Memcached, the official wiki should be a perfect read[18]. Memcached has been around for a while and has client APIs targeting most development platforms and languages, thanks to its open source nature. Memcached itself is written in C.

An interesting Caching Platform targeting .NET, as well as Java and C++, is Oracle Coherence[19]. There wasn't enough time to evaluate it; otherwise it would have been one of the tested Distributed Caching Platforms in this paper. Coherence is completely peer-to-peer-based and it provides write-behind functionality. However, the pricing can be a bit steep[20].

When it comes to Java there are a couple of platforms to choose between. First of all, two of the three caching platforms that are evaluated in this paper have Java Client APIs; Alachisoft NCache and ScaleOut StateServer. Other interesting products targeting Java is Terracotta Ehcache[21], Apache Jakarta Projects Java Caching System (abbr. JCS)[22], HazelCast[23] and JCache[24].

3 EVALUATED DISTRIBUTED CACHING SYSTEMS

Below follows a summary of supported features of the different Caching Platforms, the following sections will discuss these features in detail:

	Alachisoft NCache Enterprise Edition	**Microsoft AppFabric Caching**	**ScaleOut StateServer** Ecommerce Edition
Compatible OS		Windows Vista SP2, Windows 7, Windows Server 2008 SP2/R2	Windows (unspecified), Red Hat Enterprise 4/5, Fedora 7, Solaris
Price	1245$/Client 2495$/Server	Free	~ 500$/Client ~ 1500$/Server
Partitioned topology	Yes	Yes	Yes
Replicated topology	Yes	No	No
High Availability	Yes, 1 replica only	Yes, 1 replica only	Yes
Coherency model	Notification	Per-request, notification, timeout	Per-request
ASP.NET Session	Yes	Yes	Yes
Cluster event notifications	Yes	Yes	No
Back store	Yes	No	Yes
Regions	Yes	Yes	No
Tags	Yes	Yes	No
Remote	Backup only	No	Yes, it costs 7500$/datacenter
Configuration & Management	GUI and command-line	Only PowerShell	GUI and command-line

3.1 MICROSOFT APPFABRIC CACHING

Microsoft AppFabric which just has been released as version 1.0 features a distributed caching component that was previously stand-alone and called Velocity. The AppFabric caching cluster is centralized and controlled by a cluster manager, either an SQL Server instance or a couple of lead hosts.

The cache hosts can run on the same machines as the clients, but can also without any configuration changes run as remote clients as long as any firewalls have been set up correctly. Nothing needs to be installed on cache clients.

AppFabric Caching features a PowerShell-based administration tool to configure both the cluster as a whole and each individual cache host, no graphical interface or API is available.

3.1.1 SUPPORTED CACHING TOPOLOGIES

AppFabric Caching uses a partitioned topology by default and it also includes an option for High Availability.

3.1.1.1 HIGH AVAILABILITY

If a Cache Host goes down the data on that particular host will be lost unless the AppFabric caching cluster is configured to use *High Availability*[25]. In AppFabric case this means that one replica of each item will be stored at an additional cache hosts to prevent loss of data in case the host where the primary item resides is lost. The parameter to control High Availability is called *Secondaries* and can be set in PowerShell; it can at this moment only be set to 0 and 1, but maybe we'll see more flexibility in the future?

With High Availability only the primary objects are accessed by the clients, so there is no gained performance from the replication of objects. There is however a small performance cost for the replication process as the client has to wait for the process to complete between the cache hosts.

3.1.2 AVAILABILITY

In AppFabric it is the cluster manager's responsibility to keep the cache cluster up and running[26], as well as handling cache hosts that join or leave the cluster. The cluster management can be configured as following:

Storage Type	Storage Location	Possible Cluster Managers
XML File	Shared Network Folder	Lead Hosts
SQL Server Database	SQL Server	Lead Hosts or SQL Server
Custom Provider	Custom Store	Custom Store

As you can see the possible Cluster Managers are limited by the Storage Type chosen, to be able to choose SQL Server as Cluster Manager the configuration needs to be stored there as well.

No more than one cache host is required for the cluster to work, but depending on the configuration either the SQL Server or the *majority* of the lead hosts need to be running for the cluster to stay available. When it comes to lead hosts this means that if only two lead hosts are present in the cluster, the cluster requires both lead hosts to remain available.

If an SQL Server database is the primary source of the data that is stored in the cache, it might be a good idea to use SQL Server as the cluster manager to avoid the extra complexity of lead hosts. This would make the cache available for as long as the persistent data is available.

3.1.3 LOCAL CACHING

AppFabric supports all three coherency models:

- A local timeout is always present. All objects in the local cache uses the same timeout.
- Notification-based coherency can also be used. The polling interval is set to 300 seconds by default.
- In addition, the API method *GetIfNewer*[27] can be used to check whether the local cache item is the same as in the distributed cache. The method returns null if the items are identical or the object if it has been updated.

3.1.4 CONFIGURATION AND ADMINISTRATION

Configuration and Administration is done through the Windows PowerShell[28]. AppFabric comes packed with PowerShell commands that are available to control the Caching component. There is documentation available for those commands at MSDN, for both configuration[29] and administration[30].

There is no official GUI available for AppFabric Caching, an attempt[31] has been made by using "cmdlets" which works directly against the PowerShell console, but it doesn't seem to work very well. Microsoft has no plans on releasing an API to make it possible for developers to create their own administration interface[32].

3.1.5 SECURITY

Security[33] is enabled by default in AppFabric Caching. When security is enabled Cache clients need to have their associated Windows Account added to the clusters list of trusted accounts, which is done with the PowerShell administration tool.

To enable or disable security the security mode is set to either 'Transport' or 'None'. There is also an option to set protection level, where the options are 'None', 'Sign' and 'EncryptAndSign'. Note that the cache clients need to have at least as secure settings as the cache cluster for the connection to work, more restrictive security settings are also allowed.

3.1.6 REGIONS

All data cached with AppFabric Caching is contained in regions. These regions are either explicitly created through the API or otherwise automatically generated by the cluster.

All data of a single region is always contained at one single cache host. This can result in scalability issues if a large fraction of the cached data is stored in one region, or one region is under particularly heavy load. Note that the cluster maintains more than one region when data is stored without a specified region.

Some API features, like tag-based search, flushing of items, and bulk get operations are available only for data in a specified region.

3.1.7 CLIENT API

The Client API[34] of AppFabric Caching is contained in the namespace *Microsoft.ApplicationServer.Caching*, which becomes available by referencing the libraries Microsoft.ApplicationServer.Caching.Client.dll and Microsoft.ApplicationServer.Caching.Core.dll from the %windir%/System32/AppFabric folder.

The base of the client API is the *DataCache* class which is instanced by the factory class *DataCacheFactory*. The DataCache object is then used to perform actions on the cache data.

The mostly used methods of the API are the simple *Add*, *Put*, *Get* and *Remove* methods which are recurring in all caching frameworks, with occasional differences in chosen words. The programmer has the possibility to use tags in addition to a cache key, and in that way be able to quickly access various related objects in the cache.

Interesting is that even if regions are not explicitly used by the programmer, they are needed for many operations of the cache. Get all objects in the cache is done using the *GetObjectsInRegion* method for each region, and clearing data in the cache has to be done through the *ClearRegion* method.

There are callbacks available on certain events in the cluster. In the *DataCacheFactory* the polling time of these events can be specified, it defaults to 300 seconds. Notifications are available on the cache level (all region and item events), region level and item level.

3.1.8 FIREWALL CONFIGURATION

AppFabric Caching uses the following ports[35] by default:

Port Number	Port Name	Description
22233 (TCP)	Cache port	Communication between host and client
22234 (TCP)	Cluster port	Communication to ensure availability between hosts
22235 (TCP)	Arbitration port	Makes sure the cache host Is unavailable on failure
22236 (TCP)	Replication port	Used to transfer data between hosts when high availability is enabled

These ports has to be open on each cache host for the cluster to work, the client does not require any firewall configuration. The ports can be different for each host in a cluster as the information is stored centrally in the cluster. The port settings can be reconfigured in the PowerShell management console.

3.1.9 ARTICLES

Grid Dynamics - Scaling .NET Web Applications with Microsoft's Project Code-named "Velocity"[36]

MSDN Magazine (June 2009) – Build Better Data-Driven Apps with Distributed Caching[37]

MSDN Magazine (June 2010) – Real-World Usage and Integration[38]

3.1.10 LINKS

A couple of useful links are the official MSDN documentation[39], download page[40], MSDN forum[41] and the projects Development Team Blog[42].

3.2 ALACHISOFT NCACHE

Alachisoft NCache is a feature-rich Distributed Cache that targets both .NET Framework and Java clients on both Linux and Windows and it has been around since 2005[43]. NCache also has Memcached Edition[44] that extends the functionality of Memcached, which has been mentioned previously. NCache is decentralized, so there is no single point of failure.

The latest version (3.8) has added support for Entity Framework, LINQ (not fully supported) as well as streaming and a whole bunch of other interesting features[45]. Not all features are discussed in this paper.

3.2.1 SUPPORTED CACHING TOPOLOGIES

NCache features most caching topologies of all the evaluated Caching Platforms, with a couple of variants of each standard topology[46]:

3.2.1.1 PARTITIONED TOPOLOGY

The partitioned topology of NCache distributes data across all cache hosts in the cluster. The read and write performance remains constant as the cache size increases, and the total throughput as well as the memory available in the cluster can easily be improved by adding new cache hosts to the cluster. Note that the replication process can be done asynchronously in NCache which should ensure that clients don't have to wait for the replication.

3.2.1.2 REPLICATED TOPOLOGY

NCache provides a replicated topology, which replicates all data to all cache hosts, to boost the read performance of the cluster. However, the write performance is slow as the data has to be replicated to every node, so it's not very effective for large clusters or write-intensive applications.

3.2.1.3 PARTITION-REPLICA TOPOLOGY

The partition-replica topology is a High Availability topology where one replica of each item is stored on another host. This replication can be done asynchronously or synchronously.

3.2.1.4 MIRRORED TOPOLOGY

The mirrored topology is the only topology featured in the free version of NCache. It's a type of high availability topology which supports only 2 nodes. One of the hosts is active and the other passive and all data is replicated from the active to the passive host, in a high availability cache all the servers are contain active and replicated items. The replication is done in the asynchronously, so the cache client doesn't have to wait for the process to complete, which is another difference from the ordinary high availability topology.

3.2.2 AVAILABILITY

The NCache cluster has a decentralized peer-to-peer structure, i.e. no central server is needed for the cluster to keep working. This means that as long as there is a node available in the cluster it is available to *all* clients. On each client machine an NCache installation has to be made, so client applications communicates (through the API) with their local cache installation which in turns communicates with the cluster.

3.2.3 LOCAL CACHING

The local caching in NCache features multiple configuration options that are unique compared to AppFabric Caching and StateServer. The local cache can be contained either out of process or in the running process; this is easily set in the configuration tool without any code changes.

The performance of an out of process-instance is not as good as when using in-process cache, but there can be times when this option is worth considering. Let's say we have multiple web services or applications on a single server

working towards the same distributed cache, in that case the amount of memory used for local caching on that server could be minimized with an out-of-process cache.

The default coherency model used by NCache is notification-based. The default polling interval is 15 seconds. There is also a *GetIfNewer* method available, but it seems to work only for version control (if the item is available in the local cache that's what it returns).

3.2.4 Configuration and Administration

NCache features a GUI-based administration tool called NCache Manager. In NCache Manager all options of both the cluster and the clients, including client caches, can be set. It is simple to use, but has to stop the cluster before changing some of the configuration options and then start it once again, so those configuration options should be decided before deploying NCache in a production environment.

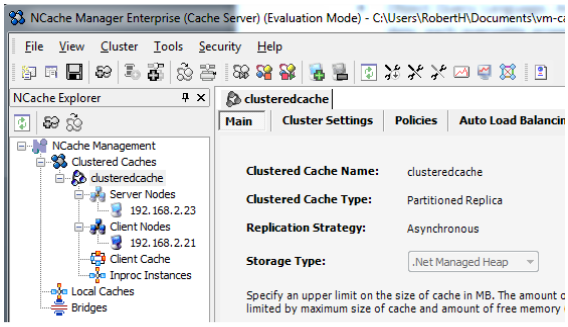

FIGURE 7: NCACHE MANAGER

3.2.5 Client API

The NCache API can be hard to understand, mainly because that there are three different Namespaces that seems to handle caching functionality:

- Alachisoft.NCache.Web.Caching
- Alachisoft.NCache.Caching
- Alachisoft.NCache.Data.Caching

In fact, only the Web.Caching namespace is meant to use for client development, the others are according to Alachisoft Support to be regarded as internal to NCache.

A big disadvantage of NCaches API is that it doesn't have any XML Comments, which makes developing require many lookups in the documentation that comes with NCache. It should be noted that the documentation is very shallow and doesn't feature any search functionality, so contact with Alachisofts Support might be required more often than desired.

3.2.6 Special Features

NCache has some special features that are unique compared to the other two evaluated Caching Platforms:

- Compact Serialization (serialization will be explained later). Instead of using .NET built-in serialization feature, NCache can handle serialization by letting the cache cluster preload the object metadata of the objects to be cached. This method minimizes the data overhead for each object sent to the cache and is configured through the NCache Manager. It is only available in the Enterprise Edition of NCache. However, be aware that this feature does not support .NET 4.0 libraries as of NCache 3.8.
- Object Query Language. An SQL-like querying language that can be used to perform actions on the cached data, each property that should be used in a query must be indexed (which is configured in the NCache Manager). Only available in the Enterprise Edition.
- Support for multiple network cards. NCache can make use of multiple network cards to increase scalability; a cache host can for example use one network card towards the rest of the cluster and another network card towards the cache clients.

3.2.7 FIREWALL CONFIGURATION

Port Number	Port Name	Description
7800 (TCP) -	Cache port	Port range of named caches
8250 (TCP)	Service port	Communication to the cache service on each host
9800 (TCP)	Server port	Host-client communication

3.2.8 ARTICLES

MSDN Magazine (October 2007), Scott Mitchell, Manage databases, easier FTP, and clustered caching[47]

MSDN Magazine (June 2010), Iqbal Khan, Address Scalability Bottlenecks with Distributed Caching[48]

Managing Data Relationships in Distributed Cache, Iqbal Khan[49]

3.2.9 LINKS

A couple of links you might be interested if you want to check out NCache is the download[50], pricing[51], support[52] and forum[53].

3.3 SCALEOUT STATESERVER

ScaleOut StateServer has been around since 2005[54]. Some of its users are AT&T, Banverket, Logitech, Burger King, E.ON US, Morgan Stanley & Co and T-Mobile[55].

3.3.1 SUPPORTED CACHING TOPOLOGIES

With ScaleOut StateServer each host in the cluster takes responsibility for a user-defined amount of data which makes it possible to distribute the data load in a way that fits the hardware of each host. For each host the parameter *store_weight* can be specified, it defaults to 500 and is used by StateServer to compute the fraction of data that should be stored at each host. This load-balancing automatically scales as new hosts are added to the cluster.

The topology used by ScaleOut StateServer is a high availability topology. With StateServer the manager has the choice to use 1 or 2 backup replicas across the cache hosts. As with AppFabric Caching, the clients only read from the active copy so no read speed is gained. The choice of using an additional replica can give a higher availability of data than AppFabric Caching's High Availability, but at the cost of additional replication across the cluster.

3.3.2 AVAILABILITY

As with Alachisoft NCache, ScaleOut StateServer is a distributed peer-to-peer platform, and because of this there is no single point of failure. This means that as long as one server is available the cache cluster will be usable.

For high availability of data at least two servers need to be present in the cluster. This also means that if critical data that mustn't be lost is stored in the cache, a couple of extra servers are to recommend, maybe in combination with more than one replica of each item.

3.3.3 LOCAL CACHING

StateServer always uses per-request coherency checks towards the distributed cache. This is great for items that require constant synchronization.

A separate caching API for local-only caching should probably be considered for objects that doesn't require constant synchronization because of the cost of the checks.

3.3.4 CONFIGURATION AND ADMINISTRATION

ScaleOut StateServer comes with both a GUI-based configuration tool called StateServer Console (yes, it still is GUI-based) as well as the StateServer Object Browser, which can be used to monitor the data contained in the cache. The

StateServer Console is easy to use and very simple in its nature, mostly because of its few configuration options. Essentially what you can do is:

- Start/Stop the cache host services.
- Monitor the state of each host.
- Clear the cache.
- Configure network settings such as IP and port-mappings.

There are also settings that can be specified in configuration files called soss_params.txt for Cache Hosts and soss_client_params.txt for Cache Clients. In this file more settings can be changed:

- Configure the maximal memory the Cache Service can use on hosts
- Configure the maximal memory the Local Cache can use in each application.
- Maximum number of replicas to use for increased availability.

3.3.5 FIREWALL CONFIGURATION

ScaleOut StateServer requires all host servers to be on the same subnet, as multicast addressing is used to auto-detect other servers as well as for failure recovery.

The default port mappings of ScaleOut StateServer are[56]:

Port Number	Port Name	Description
717 (TCP and UDP)	Management port	Used to exchange multicast packets between hosts (UDP) as well as the local management console. Required to be the same for all cache hosts.
718 (UDP)	Server port	Access from local client to cache host.
719 (TCP and UDP)	Interconnect port	Cross-host communication

Remote clients[57], applications located on servers that aren't part of the cluster, connect to a gateway address and port which may differ from the local cache host IP and server port. If using a different server and port is necessary it has to be set up on the cache host as well.

When connection has been established with the cluster it is recommended to run the "Populate" command on the remote client at regular intervals, which gets the addresses of all the cache hosts available in the cluster for future connections or if the primary cache host is offline.

3.3.6 ARTICLES

EggHeadCafe, Distributed Data Grids - Share Objects between Windows Service and ASP.NET, Robbe Morris[58]

3.3.7 LINKS

For more information about you might want to check out the official help file[59], download it[60] and check out the pricing[61].

4 IMPLEMENTATION

4.1 PROOF OF CONCEPT

Here follows a description of how the of the proof-of-concept is designed and organized:

4.1.1 CACHE PLATFORM MANAGERS

The implementation uses interfaces to streamline the communication to and from the caching component. The most essential interface is the ICachePlatformManager, which provides a unified way to communicate with multiple caching platforms. Most basic functionality is very much alike in the different caching systems, even the distributed and local caching has very much in common when it comes to the basic methods of the client API.

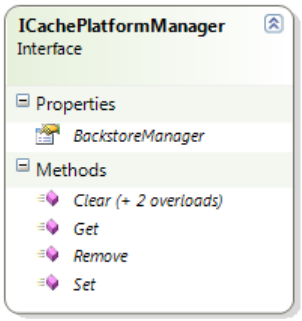

FIGURE 8: THE ICACHEPLATFORMMANAGER INTERFACE WHICH PROVIDES UNIFIED ACCESS TO THE CLIENT API ON DIFFERENT CACHING PLATFORMS.

This interface has been very useful when benchmarking because of the simple way to switch between caching frameworks. But this approach is definitely usable in production code as well, as it provides a simple interface towards the rest of the system. It makes it easy to work against caches and makes it easy to add new caching platforms if needed.

4.1.2 CACHE MANAGER FACTORY

A design pattern that has been used in the caching component is a factory class that handles the instances that implement ICachePlatformManager, making it easy to reuse the same instance across the whole application. It also enables simple usage of multiple caching instances working side-by-side by supporting named cache managers. This can be useful in two cases:

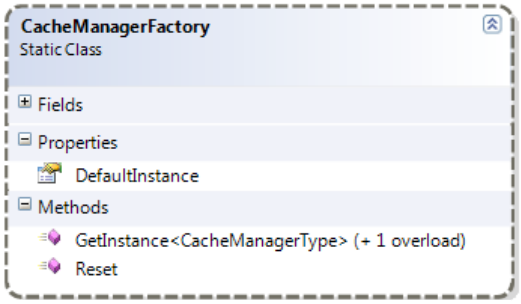

FIGURE 9: THE CACHEMANAGERFACTORY WHICH PROVIDES STREAMLINED ACCESS TO CACHEMANAGERS THROUGHOUT THE APPLICATION.

- It makes it possible to use a local cache platform together with the distributed caching platform for data that doesn't need to be stored in a distributed cache. As mentioned earlier (and shown in the benchmarks later on), *local caching should be used whenever possible* because of its unmatchable speed.
- Multiple cache managers can be used to communicate with the *same* cache, which is interesting in some cases when separate configurations are preferred on the cache managers but we still want to work against the same caching platform. Say that one cache manager works directly against the distributed cache, which could be preferred for synchronization-critical data, and another cache manager should work against the same distributed cache *with a local cache* enabled to increase read performance.

4.1.3 CACHE ITEM OPTIONS

There are a couple of options that can be specified on an item level in the cache that's fairly consistent across caching platforms; these are expiration and eviction settings. In the proof-of-concept implementation a CacheItemOptions has been implemented to specify platform-independent options for this, and it also includes a list of strings which will be saved with the object in the cache as related tags. The options available are:

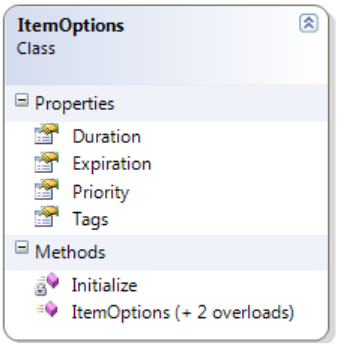

FIGURE 10: THE ITEMOPTIONS INCLUDES DATA RELATED TO THE CACHED ITEM.

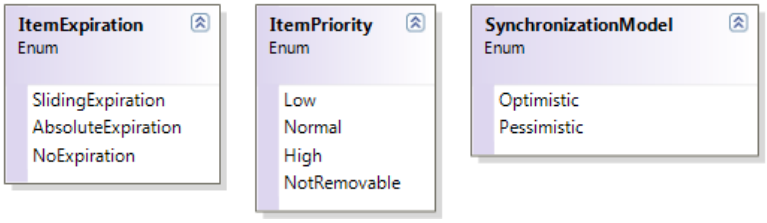

FIGURE 11: THE EXPIRATION AND EVICTION OPTIONS THAT CAN BE SPECIFIED FOR EACH CACHED ITEM.

- Expiration is the type of expiration to use; this can be set to Sliding, Absolute or NoExpiration.
- Timeout specifies the time before the item is expired, this is a TimeSpan and functionality depends on the expiration.
- Priority decides which cache items should be evicted when the memory available for the cache becomes to low. Can be set to NotRemovable, High, Medium and Low.
- This is then converted inside the individual CachePlatformManagers to the platform-specific item options before inserted into the cache.

The item options specify everything except the cache key and the cached object itself.

4.1.4 CACHE ATTRIBUTE

By using a Cache attribute the usage of caching can be simplified for the programmer utilizing the caching component. When setting or getting data from the data source, the existence of the cache attribute can be checked and handled accordingly, i.e. deciding if the data should work directly toward the data source (if no cache attribute is present) or through the cache. In this attribute it is possible to specify settings for the cached item, like expiration and eviction. This makes the attribute the only thing necessary to specify for the programmer.

However, we still need to do something with the caching attribute. Let's say that there is already a data model in place including an ORM (Object Relational Mapper). All we would need to add on top would be a check to see whether to use the cache or go directly to the database for the data (a write would of course result in a write to the database in both cases). This check could look like the following for a get operation with C# syntax:

```
object Get<type>(SomeFilter filter)
 {
     CacheAttribute attribute = GetCacheAttributeFromType(typeof(type));
     if (attribute != null)
         return GetCached(filter);
     else
         return GetFromDB(filter);
 }
```

4.1.5 TAGS

Some features are not common for all cache platforms. One of these features is tagging of cache items, which is only available in NCache Enterprise Edition and AppFabric Caching. However, tags are pretty useful when it comes to flushing specific areas of the cache.

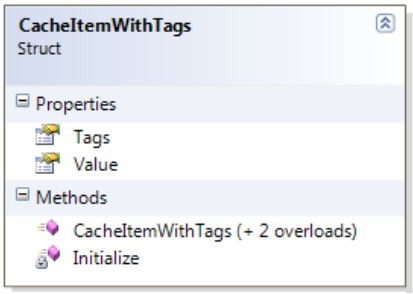

FIGURE 12: THE CACHEITEMWITHTAGS STRUCT PROVIDES TAG SUPPORT TO THE CACHING PLATFORMS WHICH DOESN'T SUPPORT TAGS NATIVELY.

Say that we have some user specific data collection that we cache, and we make a change to that data. Here there are two choices, either simply invalidating any data that could be affected or making a complicated cache update. If we choose the invalidation method we want to invalidate as little data as possible, and to do that we need to be able to specify which data to

invalidate in a very specific manner. Tagging could greatly optimize this invalidation process.

To resolve the issue that the other caching platforms don't support tagging natively a struct named *CacheItemWithTags* has been added to the cache item for these platforms. The CacheItemWithTags consist of a collection of tags and the cache item itself. This is handled inside the cache manager, so the programmer would never have to think about anything else than specifying the tags. This may not be as efficient as using the native tagging features but it provides tagging support and keeps the use of tags streamlined throughout the caching component.

4.1.6 Backing store

Using a backing store for read/write-through is supported by NCache and StateServer, and an interface has been included in the proof-of-concept to streamline this functionality. A class that implements the *ICachePlatformBackstoreManager* can be specified for each CacheManager, and by doing so the appropriate functions will be called in the implemented class when events occur in the cache.

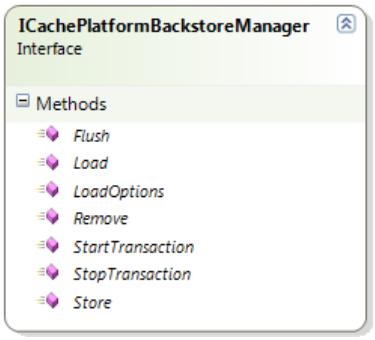

FIGURE 13: THE INTERFACE FOR BACKSTORE MANAGERS

Note that only NCache and StateServer support this and the other caching platforms will throw a *NotImplementedException* if an ICachePlatformBackstoreManager is assigned to the cache manager. However, all caching platforms could theoretically use this interface for data source communication, but without the possibility of letting the caching framework use write-behind or refresh-ahead it's probably simpler to handle it separately.

4.1.7 Inversion of Control

Unity Framework has been used to switch between caching platforms when testing without the need to recompile. It provides a simple way to inject a constructor through the applications configuration file.

4.2 Configuration issues

4.2.1 Multiple network cards

When configuring NCache on a server with multiple NICs (Network Interface Card), it is required to specify the IP address that shall be used by the caching service. The interesting thing is that it also applies to virtual machines hosted on a machine with two NICs where the virtual machine only has access to *one* NIC. This has to be set for both client and cluster-wide communications.

4.2.2 Client-cachehost security

In AppFabric Caching the user account that owns the process of the cache client has to be added to the clusters list of allowed clients. The alternative is to disable this security option.

4.2.3 Max buffer size

The maximum buffer size of AppFabric Caching is too low for big data collections sent over the network, I resolved this by setting TransportProperties.MaxBufferSize to 1 000 000 000 bytes.

5 RESULTS

5.1 BENCHMARKS

The benchmark has been conducted both with lightweight objects independent of the underlying platform, and with heavyweight objects in the system with complex references to other objects in the system. The lightweight objects give a hint about the maximum throughput of each Caching Platform. The benchmarks have been done with 1 000, 10 000 and 100 000 lightweight objects respectively. The lightweight objects are of about 200B in size each, so 1 000 objects is about 200KB, 10 000 objects is about 2MB and 100 000 is about 20MB of data.

The benchmarks of complex objects are interesting in measuring how well the platforms handle complexity and serialization, and gives a hint of the worst-case throughput of the platform. It is interesting to note the massive difference between the platforms even as all platforms use the built-in .NET-serialization in this benchmark. The benchmarks have been done with 100 and 1 000 heavyweight objects respectively. Because of the serialization all related properties also are stored in the cache, and therefore the memory on clients and hosts can't handle more than 1 000 objects. Production code could and *should* optimize the usage of heavyweight objects, no optimization has been done in these benchmarks (see the *Serialization* section in the *Distributed Caching* chapter).

The items that are benchmarked are collections of data, so 'Object Count' in the charts actually refer to the number of objects in that *one* collection object. This is because it's the normal caching pattern in the target system as of today.

5.1.1 BENCHMARK SETUP

	Cache Client	Cache and Database Server (virtual)
Processor	Intel Core 2 Duo T7400 @ 2.16GHz	Intel Core i7 X3430 @ 2.40GHz (one core reserved for virtual OS)
Memory	2GB DDR2 @ 666MHz (5-5-5-15-21)	2GB EDO @ 200MHz (3-3-3-?-?)
Hard Drive	ST910021AS 7200rpm	(Virtual)
Operating System	Microsoft Windows 7 Professional	Microsoft Windows Server 2008 R2 32-bit
DBMS	SQL Server 2008 R2	SQL Server Developer Edition 2008

The benchmarks where run on a 1 Gbps network.

Only one cache host was used in the benchmarks, so there is no specific topology used. This removes the factor of replication as well as possible penalties for searching for the right host on the network, this means that a best-case setup has been used. In the target system a High Availability topology is not of particular interest, so it has not been a high priority to benchmark results of different topologies. If a comparison of different topologies had been interesting an additional benchmark for reading from the cluster would be a good way to compare the replication performance of each Caching Platform.

5.1.2 POTENTIAL DIVERGENCES

The memory on the client machine was too low during some of the benchmarks which meant that paging occurred, something that ruined the test results and forced a rerun of the benchmarks. This has been monitored during the course of the benchmarks, but some paging activity might still have slipped through.

Note: Benchmarking the actual memory used by the Caching Platforms, on both client and host, could be another interesting benchmark. However, there hasn't been enough time to do it for this paper. That benchmark would require measuring of how much memory is used *before* any items are added, when items *have* been added, and how efficiently the memory is reclaimed *after* the items are flushed.

5.1.3 READING DIRECTLY FROM THE CLUSTER

These first benchmarks show the time to read objects directly from the distributed cache without using a local cache. The performance of SQL Server to get the same objects is included for reference, because reading directly from the cluster is the comparable behavior to reading from the database when it comes to distributed caching.

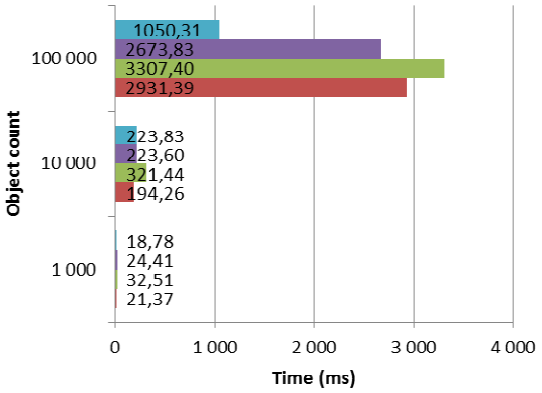

FIGURE 14: READ LIGHTWEIGHT DATA FROM DISTRIBUTED CACHE WITH LOCAL CACHE DISABLED

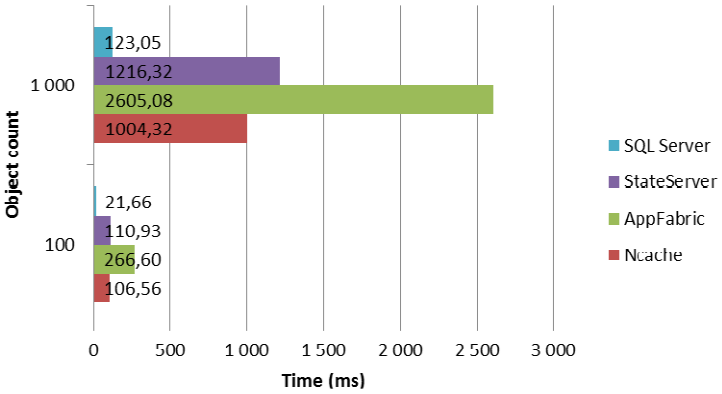

FIGURE 15: READ HEAVYWEIGHT DATA FROM DISTRIBUTED CACHE WITH LOCAL CACHE DISABLED

This first benchmark with lightweight objects shows that the database is fast with larger sets of data. The time it takes to read the data from the cache increases linearly with the amount of objects returned, while the time to read from the database scales better than linearly. This is probably because the database is optimized for bulk operations. It is important to remember that there is no additional load on the database, so the scalability is not a factor here.

This second benchmark is done with heavyweight objects, which increases the serialization cost for objects fetched from the cache. The same pattern as with the lightweight objects can be seen; the time read increases linearly for the caches and better than linearly for the database.

5.1.4 READING FROM LOCAL CACHE WITHOUT PER-REQUEST COHERENCY

This next benchmarks shows read performance when using local caching without using per-request coherency. Here the local-only APIs are included for reference as well because they're comparable with these settings on the distributed caches. Note that StateServer is excluded from this benchmark because per-request coherency cannot be disabled for that particular platform.

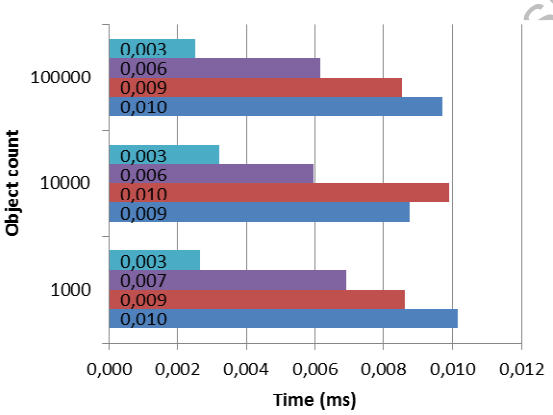

FIGURE 16: READ LIGHTWEIGHT OBJECTS WITH LOCAL CACHING ENABLED WITHOUT PER-REQUEST COHERENCY

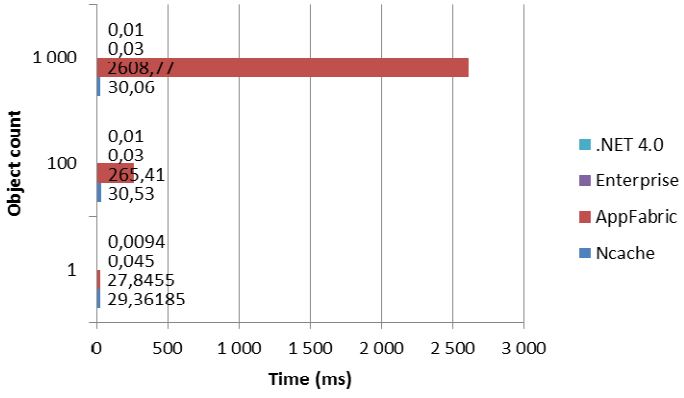

FIGURE 17: READ HEAVYWEIGHT OBJECTS WITH LOCAL CACHING ENABLED WITHOUT PER-REQUEST COHERENCY

As we can see in Figure 16, the local-only caches in Enterprise Library and .NET 4.0 performs better than the distributed caches when it comes to local caching. The functionality is the same for all the caches in this benchmark, and the measured time is only the the time to read from the local cache.

As we can see in Figure 17, some caching APIs has a harder time to handle heavyweight objects. This is still done locally, so the time taken should be constant. But as we can see, the number of objects has much effect on some of the caching APIs. Only the local-only caches EnterpriseLibrary and .NET 4.0 is handles this great. NCache handles the increase of object linearly, but it takes relatively long time.

For AppFabric the performance is almost identical to the performance with the local cache disabled, which suggests it doesn't use the local cache at all. Probably there is some limit of how complexity limit of the objects stored in the local cache. The only parameter that can be set in configuration is the object count, a limit that is not exceeded in this case.

5.1.5 READING FROM LOCAL CACHE WITH PER-REQUEST COHERENCY

The next couple of benchmarks show read performance when a local cache is enabled and using per-request coherency. NCache doesn't feature per-request coherency and is not included in these benchmarks.

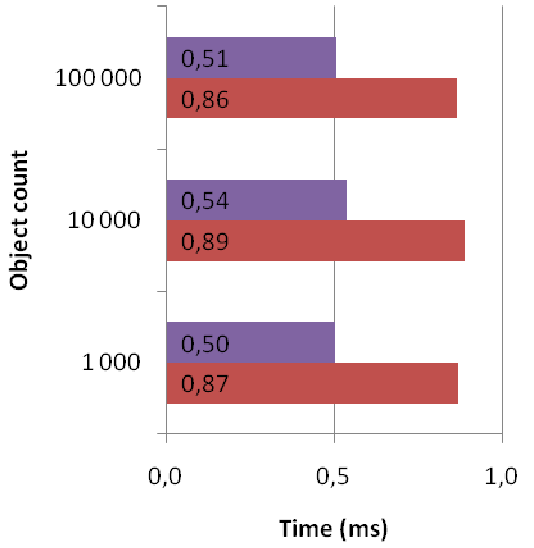

FIGURE 18: READ LIGHTWEIGHT OBJECTS WITH LOCAL CACHE AND PER-REQUEST COHERENCY

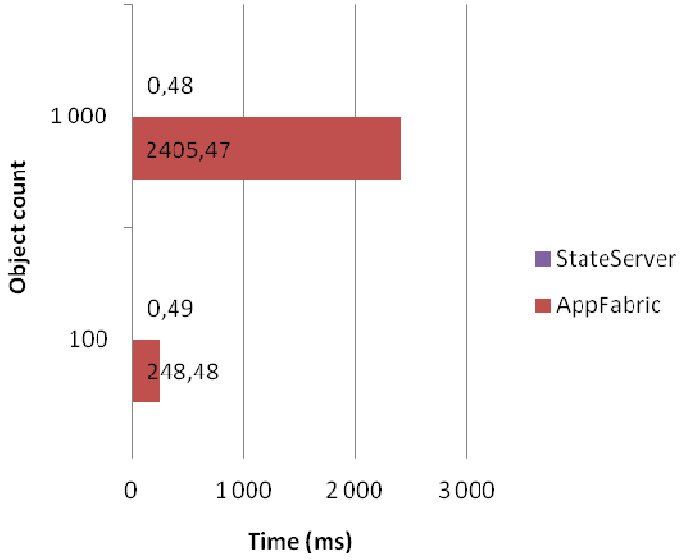

FIGURE 19: READ HEAVYWEIGHT OBJECTS WITH LOCAL CACHE AND PER-REQUEST COHERENCY

In Figure 18 we can see that the performance of using per-request coherency checks is much faster (increasingly faster with increasingly complicated objects) than working toward the distributed cache, but not as fast as working directly toward the local cache (about 50-80 times slower). As we work with a single data collection of different sizes only a version number has to be sent to the cluster, so the time required for the checks is constant. With smaller collections the advantage towards using the distributed cache is smaller, but still significant.

In figure 19 we can see the same benchmark done with heavyweight objects. AppFabric has the same issue as when using local caching without per-request coherency. StateServer performs as expected at a constant time as the size of the object increases, as with the lightweight objects.

6 CONCLUSION

The main conclusions of the project are these:

- Distributed caching is definitely a plausible solution to the problem of cache synchronization and there are a couple of synchronization options available depending on the Caching Platform.
- Distributed caching could also be used to optimize the amount of memory used. But for performance a local cache should be used when possible.
- The implementation shows a homogenized and simple way to use a distributed cache.

The implementation of the proof-of-concept was very interesting. It opened up new thoughts about how todays caching system could be improved; even things that aren't directly related to Distributed Caching but more to caching in general. A couple of these features have been documented in the *Implementation* section, and listed as recommendations later in this section.

The implementation and testing was an important part of understanding and documenting the dynamics of Distributed Caching. Reading has been important, but without testing and seeing the results of different configurations it would've been hard to sort out the key areas that really matter when implementing a Distributed Caching solution, as well as the main performance factors. The problems that where handled along the way have resulted in new sections in this report, serialization is one example which complicated the testing towards the target system but in turn proved how important it is to take into consideration.

6.1 EVALUATION

Here follows the conclusions regarding the evaluated Caching Platforms:

6.1.1 MICROSOFT APPFABRIC CACHING

AppFabric Caching is a new platform and will only get better with time, and even now it seems mature enough to consider using in a production environment. The one significant issue is that of not handling heavyweight objects well with local caching enabled.

Advantages	Disadvantages
• It has all coherency methods available, which makes it flexible. • It's free. • It features tag-based searching, which could be used to optimize flushing.	• The performance is not as good as the other two, especially when it comes to heavyweight serialization. • It doesn't support read/write-through, and therefore not write-behind which could improve write performance. • It requires PowerShell for configuration and management, which might be a bit inconvenient (no API or GUI is available).

6.1.2 ALACHISOFT NCACHE

NCache performs well and has many features. However, NCache Manager is a weakness and the documentation is outright bad.

Advantages	Disadvantages
• NCache is the most feature-packed Caching Platform of the three. • The performance is good compared to the other Distributed Caching Platforms. • Support through phone and video conferencing has been good.	• NCache Manager can corrupt the configuration files and is sometimes very slow. • Some of NCache's features is incompatible with .NET 4.0 (Compact Serialization) and even .NET 3.5 (Read/Write-through). • The API is completely missing XML comments. • The documentation is shallow and not searchable. • The NCache Forum is very inactive (it took 6 days to get an answer, and that was after e-mail contact had been made with the support) so contacting the Alachisoft support team seems to be the only way to get help. • Groups (NCaches equivalent of regions) and tags cannot be used at the same time.

6.1.3 SCALEOUT STATESERVER

ScaleOut StateServer seems like the most mature of the three Distributed Caching Platforms. One weakness is that there is no time-based synchronization, but StateServer compensates with the best per-request synchronization performance.

Advantages	Disadvantages
• It performs well in all the benchmarks. • It features back store functionality and could be used to improve write performance towards the cache. • It doesn't require much configuration. • The e-mail communication with ScaleOut support has been good. • Can be replicated across multiple datacenters. • Efficient per-request coherency.	• The configuration options are few, only per-request coherency is available. • The help file could be better organized. • Tags and regions are not available. • Cluster event notifications are not available.

6.2 RECOMMENDATIONS

Here follows some recommendations of how to improve today's system, all of these points are further explained in the Implementation section:

- The architecture of the proof-of-concept is a possible way to implement multiple caching platforms side-by-side in a way that's easily extensible. The use of a factory class makes usage of the caching component simple and streamlined throughout all applications that make use of it.
- *Tag-based flush* could improve today's system by providing selective flushing. It would make it possible to clear only the data that has changed, instead of flushing all objects of the same type. Tags could be used both with distributed and local-only caching APIs.

- *Attribute-based caching* could simplify and homogenize today's system for the programmer. In effect this would mean that only a caching attribute would need to be specified for each class to cache it. Cache item settings can be specified there as well, providing tailored settings caching functionality per class. Attribute-based caching could be used both with distributed and local-only caching APIs.

- *Write-behind* could improve write performance and lessen the load on the database. Here the cache would be used like a buffer for write operations, so the clients wouldn't need to wait for the database write operation to complete. This is a distributed caching-only recommendation.

7 REFERENCES

[1] MSDN Library, Windows AppFabric Caching, Data Classification (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/library/ee790832.aspx

[2] MSDN Library, Windows AppFabric Caching, Expiration and Eviction (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/library/ee790981.aspx

[3] MSDN Library, Windows AppFabric Caching, Concurrency Methods (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/library/ee790915.aspx

[4] Wikipedia, Optimistic Concurrency Control (last accessed 2010-08-10)
http://en.wikipedia.org/wiki/Optimistic_concurrency_control

[5] Oracle Coherence Developer's Guide, 9 Read-Through, Write-Through, Write-Behind, and Refresh-Ahead Caching (last accessed 2010-08-10)
http://download.oracle.com/docs/cd/E14526_01/coh.350/e14510/readthrough.htm

[6] Alachisoft, NCache Caching Topologies (last accessed 2010-08-10)
http://www.alachisoft.com/ncache/caching_topology.html

[7] Oracle Coherence Developer's Guide, 10 Introduction to Caches, Replicated Cache: (last accessed 2010-08-10)
http://download.oracle.com/docs/cd/E15357_01/coh.360/e15723/cache_intro.htm#BABJCBIJ

[8] MSDN Library, Windows AppFabric Caching, Named Caches (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/library/ee790985.aspx#sectionSection0

[9] MSDN Library, Windows AppFabric Caching, Regions (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/library/ee790985.aspx#sectionSection1

[10] Performance of protobuf-net (last accessed 2010-08-10)
http://www.servicestack.net/benchmarks/NorthwindDatabaseRowsSerialization.1000000-times.2010-02-06.html

Results of Northwind database rows serialization benchmarks run at 06/02/2010 (last accessed 2010-08-10)
http://code.google.com/p/protobuf-net/wiki/Performance

[11] Stefan Matsson and Elias Ung, Managing memory scalability in web gardens (ISSN: 1651-4769) (last accessed 2010-08-10)
http://gupea.ub.gu.se/bitstream/2077/20753/1/gupea_2077_20753_1.pdf

[12] Thomas Seidmann, Replicated Distributed Shared Memory For The .NET Framework (last accessed 2010-08-10)
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.116.9577&rep=rep1&type=pdf

[13] Cuong Do Cuong, YouTube Scalability (video) (last accessed 2010-08-10)
http://video.google.com/videoplay?docid=-6304964351441328559#

[14] Thorsten Shuett, Scalable Wikipedia with Erlang (last accessed 2010-08-10)
http://www.youtube.com/watch?v=pW339qR7DvU&feature=channel
(slides: http://onscale.de/Schuett_Google_Scalability.pdf)

[15] Paul Saab, Scaling memcached at Facebook (last accessed 2010-08-10)
http://www.facebook.com/note.php?note_id=39391378919&ref=mf

[16] Jack Dorsey and Biz Stone, It's Not Rocket Science, But It's Our Work (last accessed 2010-08-10)
http://blog.twitter.com/2008/05/its-not-rocket-science-but-its-our-work.html

[17] Leena Rao, TechCrunch, NorthScale's Memcached Data Management Technology Attracts Zynga And Others (last accessed 2010-08-10)
http://techcrunch.com/2010/03/16/northscales-data-management-technology-attracts-zynga-and-others/

[18] Memcached Wiki (last accessed 2010-08-10)
http://code.google.com/p/memcached/wiki/NewStart

[19] Oracle Coherence Overview (last accessed 2010-08-10)
http://www.oracle.com/technetwork/middleware/coherence/overview/index.html

[20] Oracle Coherence Pricing (last accessed 2010-08-10)
https://shop.oracle.com/pls/ostore/f?p=ostore:product:3807750855123333::NO:RP,3:P3_LPI:4509071028721805719892

[21] Terracotta Ehcache Distributed Cache (last accessed 2010-08-10)
http://ehcache.org/

[22] Apache Jakarta Project, Java Caching System (last accessed 2010-08-10)
http://jakarta.apache.org/jcs/

[23] Hazelcast Software (last accessed 2010-08-10)
http://www.hazelcast.com/

[24] JBoss Cache Overview (last accessed 2010-08-10)
http://jboss.org/jbosscache/

[25] MSDN Library, Windows AppFabric Caching, High Availability (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/library/ee790974.aspx

[26] MSDN Library, Windows AppFabric Caching, Lead Hosts and Cluster Management (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/library/ee790895.aspx

[27] MSDN Library, Windows AppFabric Caching, Concurrency Methods (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/library/ee790915.aspx

[28] MSDN Library, Windows AppFabric Caching, Using Windows PowerShell to Manage Windows Server AppFabric Caching Features (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/library/ee790886.aspx

[29] MSDN Library, Windows AppFabric Caching, Cache Configuration with Windows PowerShell (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/library/ff718170.aspx

[30] MSDN Library, Windows AppFabric Caching, Cache Administration with Windows PowerShell (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/library/ff718177.aspx

[31] CodePlex, AppFabric Caching Admin Tool (last accessed 2010-08-10)
http://mdcadmintool.codeplex.com/

[32] MSDN Forums, AppFabric Caching, AdminAPI (last accessed 2010-08-10)
http://social.msdn.microsoft.com/Forums/en/velocity/thread/f53e3e50-2b27-4a23-8198-1eb0f0e0e843

[33] MSDN Library, Windows AppFabric Caching, Security Model (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/library/ff718179.aspx

[34] MSDN Library, Windows AppFabric Caching, Cache Client API Overview (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/library/ee790858.aspx

[35] MSDN Library, Windows AppFabric Caching, TCP/IP Communications (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/library/ee790914.aspx

[36] Grid Dynamics - Scaling .NET Web Applications with Microsoft's Project Code-named "Velocity" (last accessed 2010-08-10)
http://download.microsoft.com/download/1/B/2/1B21D4A1-C84C-4CB8-923A-740BD927CDEB/Velocity%20Benchmark%20White%20Paper.docx

[37] MSDN Magazine (June 2009) – Build Better Data-Driven Apps with Distributed Caching (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/magazine/dd861287.aspx

[38] MSDN Magazine (June 2010) – Real-World Usage and Integration (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/magazine/ff714581.aspx

[39] AppFabric Caching Documentation (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/library/ff383731.aspx

[40] AppFabric Download (last accessed 2010-08-10)
http://www.microsoft.com/downloads/details.aspx?FamilyID=467e5aa5-c25b-4c80-a6d2-9f8fb0f337d2

[41] AppFabric Caching Forum (last accessed 2010-08-10)
http://social.msdn.microsoft.com/Forums/en-US/velocity/threads

[42] AppFabric Caching Development Team Blog (last accessed 2010-08-10)
http://blogs.msdn.com/b/velocity/

[43] Alachisoft Press Room (last accessed 2010-08-10)
http://www.alachisoft.com/newsletters/news.html

[44] Alachisoft NCache for Memcached (last accessed 2010-08-10)
http://www.alachisoft.com/ncache-memcached/index.html

[45] Alachisoft, What's new in NCache 3.8? (last accessed 2010-08-10)
http://www.alachisoft.com/ncache/whats_new.html

[46] Alachisoft, NCache Caching Topologies (last accessed 2010-08-10)
http://www.alachisoft.com/ncache/caching_topology.html

[47] MSDN Magazine (October 2007), Scott Mitchell, Manage databases, easier FTP, and clustered caching (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/magazine/cc163343.aspx#S3

[48] MSDN Magazine (June 2010), Iqbal Khan, Address Scalability Bottlenecks with Distributed Caching (last accessed 2010-08-10)
http://msdn.microsoft.com/en-us/magazine/ff714590.aspx

[49] Managing Data Relationships in Distributed Cache, Iqbal Khan (last accessed 2010-08-10)
http://www.alachisoft.com/downloads/CacheDependency.pdf

[50] NCache Download (last accessed 2010-08-10)
http://www.alachisoft.com/download.html

[51] NCache Pricing (last accessed 2010-08-10)
http://www.alachisoft.com/buy.html

[52] NCache Support (last accessed 2010-08-10)
http://www.alachisoft.com/support.html

[53] NCache Forum (last accessed 2010-08-10)
http://www.alachisoft.com/forum/index.php?showforum=22

[54] ScaleOut Software Release History (last accessed 2010-08-10)
http://www.scaleoutsoftware.com/pages/support/release-history.php

[55] ScaleOut Software Our Customers (last accessed 2010-08-10)
http://www.scaleoutsoftware.com/pages/solutions/our-customers.php

[56] ScaleOut StateServer Help File, Configuration (last accessed 2010-08-10)
http://www.scaleoutsoftware.com/support/stateServer/soss_help/installation/configuration.htm

[57] ScaleOut StateServer Help File, Configuring the Remote Client Option (last accessed 2010-08-10)
http://www.scaleoutsoftware.com/support/stateServer/soss_help/installation/configuring_the_remote_client_option.htm

[58] EggHeadCafe, Distributed Data Grids - Share Objects between Windows Service and ASP.NET, Robbe Morris (last accessed 2010-08-10)
http://www.scaleoutsoftware.com/pages/evaluate-purchase/free-trial-downloads.php

[59] ScaleOut StateServer Help File (last accessed 2010-08-10)
http://www.scaleoutsoftware.com/support/stateServer/soss_help/soss_help.htm

[60] ScaleOut StateServer Dowload (last accessed 2010-08-10)
http://www.scaleoutsoftware.com/pages/evaluate-purchase/free-trial-downloads.php

[61] ScaleOut StateServer Pricing (last accessed 2010-08-10)
http://www.scaleoutsoftware.com/pages/evaluate-purchase/pricing.php