Reducing the load on transaction-intensive systems through distributed caching

Joachim Andersson
Johan Lindbom Byggnings
Abstract

Reducing the load on transaction-intensive systems through distributed caching

Joachim Andersson and Johan Lindbom

Scania is an international trucks, buses and engines manufacturer with sales and service organization in more than 100 countries all over the globe (Scania, 2011). In 2011 alone, Scania delivered over 80,000 vehicles, which is an increase by a margin of 26% from the previous year.

The company continues to deliver more trucks each year while expanding to other areas of the world, which means that the data traffic is going to increase remarkably in the transaction-intensive fleet management system (FMS). This increases the need for a scalable system; adding more sources to handle these requests in parallel. Distributed caching is one technique that can solve this issue. The technique makes applications and systems more scalable, and it can be used to reduce load on the underlying data sources.

The purpose of this thesis is to evaluate whether or not distributed caching is a suitable technical solution for Scania FMS. The aim of the study is to identify scenarios in FMS where a distributed cache solution could be of use, and to test the performance of two distributed cache products while simulating these scenarios. The results from the tests are then used to evaluate the distributed cache products and to compare distributed caching performance to a single database.

The products evaluated in this thesis are Alachisoft NCache and Microsoft AppFabric. The results from the performance tests show that that NCache outperforms AppFabric in all aspects. In conclusion, distributed caching has been demonstrated to be a viable option when scaling out the system.

Keywords: Scania, transaction-intensive systems, distributed cache
Reducing the load on transaction-intensive systems through distributed caching

Joachim Andersson
Johan Lindbom Byggnings
Acknowledgements

We would like to take the opportunity to thank our supervisor at Scania, Andreas Höglund, who has provided valuable feedback and guidance throughout the entire work process.

Furthermore we would like to thank Magnus Eriksson from Scania Infomate for his expertise and helpfulness. Magnus' knowledge in the field of distributed caching and his willingness to help has been of great use in this thesis.

We want also like to thank Jonas Gustafsson at Kambi sport solutions for a rewarding meeting about distributed caching and alternative techniques.

Finally, we would like to thank the employees in the REI department for taking the time to answer questions and participating in discussions and presentations.

Södertälje, October 2012

Joachim Andersson & Johan Byggnings
# Table of contents

1. Introduction .......................................................................................................................... 1  
   1.1 Background ..................................................................................................................... 1  
   1.2 Problem formulation ......................................................................................................... 1  
   1.3 Purpose ............................................................................................................................ 2  
   1.4 Goals of the research ....................................................................................................... 2  
   1.5 Delimitations .................................................................................................................... 2  
   1.6 Report structure ............................................................................................................... 2  

2 Theoretical framework ............................................................................................................ 4  
   2.1 Cache .............................................................................................................................. 4  
   2.2 Distributed cache ............................................................................................................ 5  
      2.2.1 Topologies .................................................................................................................. 7  
      2.2.2 Data synchronization models ...................................................................................... 8  
      2.2.3 Serialization ............................................................................................................... 9  
      2.2.4 Concurrency .............................................................................................................. 10  
      2.2.5 Coherency .................................................................................................................. 11  
      2.2.6 What to measure when testing a distributed cache .................................................. 11  
   2.3 The CAP theorem .......................................................................................................... 13  
   2.4 Databases ....................................................................................................................... 13  
      2.4.1 ACID ........................................................................................................................ 14  
   2.5 Return-on-investment ...................................................................................................... 15  

3. Methodology ......................................................................................................................... 17  
   3.1 Method overview ............................................................................................................ 17  
   3.2 Distributed caching - Area of use ................................................................................... 19  
      3.2.1 Spotify ...................................................................................................................... 19  
      3.2.2 Kambi sports solutions ............................................................................................ 19  
      3.2.3 OAS (Scania) .......................................................................................................... 20  
   3.3 Distributed caching in FMS ............................................................................................ 20  
   3.4 Selection of products to evaluate ................................................................................... 21  
   3.5 Evaluated distributed cache products ............................................................................. 22  
      3.5.1 Alachisoft NCach Enterprise edition ........................................................................ 22
A. Appendix – Feature comparison

B. Appendix – Agenda of the tests

C. Appendix – Hardware setups
List of tables

Table 1: Advantages and disadvantages - NCache ................................................................. 48
Table 2: Advantages and disadvantages - AppFabric............................................................. 49
Table 3: Direct costs .................................................................................................................. 52
Table 4: Direct returns ............................................................................................................. 53
List of figures

Figure 1: As the size of the memory grows, so does the distance from the CPU .............................................. 4
Figure 2: A distributed cache system ................................................................................................................. 6
Figure 3: Mirrored cache ................................................................................................................................. 7
Figure 4: Replicated cache ............................................................................................................................... 7
Figure 5: Partitioned cache ............................................................................................................................. 7
Figure 6: High-availability cache .................................................................................................................... 8
Figure 7: Serialization .......................................................................................................................................... 10
Figure 8: Method overview .............................................................................................................................. 18
Figure 9: NCache Manager .............................................................................................................................. 25
Figure 10: PowerShell for AppFabric ............................................................................................................. 30
Figure 11: Activities of test engineers ........................................................................................................... 32
Figure 12: Test environment for the FMP scenarios ....................................................................................... 34
Figure 13: Test environment for the product scenario .................................................................................... 35
Figure 14: Test environment for database performance .................................................................................. 36
Figure 15: TestScenarios interface .................................................................................................................. 37
Figure 16: CacheFeatures interface ................................................................................................................ 37
Figure 17: Database table for the scenarios .................................................................................................... 41
Figure 18: AppFabric. High-availability. 50% reads / 50% writes .......................................................... 43
Figure 19: NCache. High-availability. 50% reads / 50% writes ................................................................. 43
Figure 20: AppFabric. High-availability. 90% reads / 10% writes ............................................................. 44
Figure 21: NCache. High-availability. 90% reads / 10% writes ................................................................. 44
Figure 22: AppFabric. High-availability. 10% reads / 90% writes .............................................................. 45
Figure 23: High-availability. 10% reads / 90% writes .................................................................................. 45
Figure 24: Comparison of Writes/s and Reads/s - AppFabric and NCache with 1024B objects .............. 46
Figure 25: Read/Write 100 000 objects to the database ............................................................................. 46
Glossary

FMS - Fleet Management System
FMP - Fleet Management Portal
MP - Message Platform
RTC - Road traffic communicator
OAS - Object and Structure tool
ROI - Return of Investment
REI - Fleet management department
Cache server/cache node – Server that holds the cached items
1. Introduction

1.1 Background
Scania is an international trucks, buses and engines manufacturer with sales and service organization in more than 100 countries all over the globe (Scania, 2011). Scania has over 35000 employees and operates one of the largest truck fleets in Sweden. In 2011 alone, Scania delivered over 80 000 vehicles, which is an increase by a margin of 26% from the previous year.

The company continues to deliver more trucks each year while expanding to other areas of the world. Today, Scania has a large number of active vehicles on the roads transmitting data to their information systems (RTC vehicles). In 2015, the projected number of RTC vehicles is to exponentially increase, which means that the data traffic is going to increase remarkably.

As the amount of data grows, so does the requirement for systems to be able to handle the data flow. A single data source can handle only a given number of requests within a certain time interval. This increases the need for a scalable system; adding more sources to handle these requests in parallel. Scalability is often an issue in large-scale applications and systems, and therefore requires advanced technical solutions to avoid congestion or loss of information.

Distributed caching is one technique that can solve this issue. The technique makes applications and systems more scalable, and it can be used to reduce load on the underlying data storage source (e.g. a database). Caching is a well-known concept in the field of IT and is used to temporarily store data for faster access in the future. By distributing caches across multiple servers, the reads/writes throughput can increase drastically. The cache system also provides the ability to replicate copies of data between the cache servers as backups.

1.2 Problem formulation
The Message Platform (MP), a product of Scania Fleet Management Services (FMS), is the hub of the wireless communication with Scania’s vehicles. Today, MP handles very high traffic volumes and will need to handle even larger volumes in the future. Scania is interested in looking at different strategies to cope with the increased data volumes, and one idea for a possible solution is to use distributed caching.

This thesis is part of developing a basis for the use of distributed caching as a technical solution, to investigate how this technology could fit in Scania’s system and how it works in theory and practice.
1.3 Purpose
The purpose of this thesis is to evaluate whether or not distributed caching is a suitable technical solution for Scania FMS.

1.4 Goals of the research
- To answer the following research questions:
  - RQ1: How does distributed caching work in theory and in practice?
  - RQ2: Which parts of Scania's FMS could use a distributed cache?
  - RQ3: Which products meet Scania’s requirements?
  - RQ4: Is a cost-free distributed caching product comparable to a commercial distributed caching product?
- To test the selected products.
- To evaluate the cost of implementation.

1.5 Delimitations
We will only focus on distributed caching as a technical solution to reduce load in transaction-intensive systems. This is because we did not have the time to immerse ourselves in additional techniques. We have also kept the test scenarios on a simple level to make them easy to understand, but at the same time provide a general picture of how it works in practice. Due to the limited time frame of this thesis, and the complexity of product implementation, we also had to delimit the number of products to evaluate and test. In an ideal world, we would have tested them all to see which one is the better, but since this is not possible, we had to make a selection of products based on a theoretical evaluation.

1.6 Report structure
The structure of this report will be as follows:

Theoretical framework
Introduction to caching in general and distributed caching. Theory about return of investment in IT-systems are also presented in this section.

Methodology
The chapter begins with a motivation of the selected method and its description, followed by a description of the meetings/interviews.
Results and analysis
Focuses on presenting the results from the conducted tests and interpretation of the results. It also contains a section with the ROI-analysis.

Discussion and conclusion
Discussion, summarizes and critically reviews of the results, and how they relate to the research goals. It also contains a conclusion and suggestions for future work.
2 Theoretical framework

This chapter consists of previous research on caches, distributed caches, basic database knowledge and return on investment for IT investments. It starts with the most fundamental part of the caching concept, covers distributed caching and its main features, databases, the CAP theorem and ends with a section where the basics of a return-on-investment analysis are presented.

2.1 Cache

The word cache refers to pretty much any storage that can take advantage of locality of access. The principle of locality states that “programs access a relatively small portion of their address space at any instant of time” (Patterson & Hennessy, 2007). Thus, we can take advantage of this principle when working with memories of limited size. Locality can be divided into two types:

- Temporal locality: Items being used are likely to be used again in a near future.
- Spatial locality: Items near a used item are likely to be used in the future.

Since memory closer to the CPU generally is faster than memory farther away (the memory hierarchy model), we can use the principles of locality to predict what data that will be used in the future, and thus save time when accessing it. And since memory closer to the CPU tends to be more expensive, and therefore also much smaller, we must choose wisely what to put there.

Due to the size of the cache being much smaller than that of a regular back-end storage (such as a hard drive), it will eventually become full, and thus will have to evict data. Eviction refers to dropping data from the cache, thereby making space for new data. Again, since the cache is limited in size, it is important to evict data that is unlikely to be reused in a foreseeable future. There are several strategies for evicting objects from the cache (Dahlin & Korupolu, 2002). Some of the most popular eviction strategies are:
● First in first out (FIFO): The item that first entered the cache will be thrown out first when more space is needed.
● Least recently used (LRU): The item that has been unused for the longest period of time will be dropped from the cache.
● Least frequently used (LFU): Removes the item from the cache that is used the least, i.e. the item with the least number of hits.

Another way of keeping the cache populated with relevant data is to use expiration. Expiration allows the programmer to specify for how long the cached item will stay in the cache, before being removed automatically. In the simplest type of expiration, called absolute expiration, items are rejected once the specified time has expired, after being added to the cache. This way, stale data (old version of the data) can be avoided to some extent. A problem with absolute expiration is that eventually, data will be removed from the cache even if it is still being used frequently. In those cases, the data will have to be gathered from the slower data storage medium (e.g. a database), which can be costly. This is where a technique called sliding expiration comes in handy. Sliding expiration allows an item’s expiration time to be reset upon a read or update request. This way, more relevant data is present in the cache which leads to less expensive requests to e.g. a database.

2.2 Distributed cache

While caching is a widely recognized concept among IT-people, many are still unfamiliar with the concept of distributed caching. Rather than having a cache run on a single unit, a distributed cache spans over multiple servers/clients and thus allows for a much higher throughput than a cache running on its own. Also, distributed caches allow for data balancing, something which traditional caches have issues with (Paul & Fei, 2001). Furthermore, a distributed and cooperating cache has been proven to provide a better hit ratio, response time and traffic handling capacity than a cluster of caches with no cooperation (Paul & Fei, 2001).
The main benefit of distributed caching is that it can keep growing according to the users’ needs. If running out of space or transaction capacity, one can simply add more clients to the cache cluster and thereby achieve linear scalability. The reason for this is that distributed caches are much simpler than other data storage systems such as relational databases. While the relational database management (RDBM) uses complex relations between entities, a distributed cache is a simple key/value store and the data is stored in the RAM. Also, distributed caches have various storage mechanisms which can be selected by the user depending on his/her needs, called topologies, which will be discussed later in this section.

Distributed caches also provide mechanics for failover protection, in the way that it can recover lost data if it has been copied to another cache in the cluster or to another data source.

Working against a distributed cache cluster is simple because it provides a logical view of a single, large cache (figure 2). Thus, the programmer does not need to know which specific cache client the data needs to be stored in or gathered from, but rather treats the cluster as a single, large traditional cache.

Another feature of distributed caches is the ability to add more capacity. According to Gualtieri (2010, blog), an elastic cache platform is a distributed cache that performs at the same rate when volumes increase, provides scaling without downtime, and is fault-tolerant. Elastic caching platforms provide elastic scalability, i.e. it allows the user to add or remove nodes without shutting the system down (Qin et. al., 2011). Many of the top distributed caching products claim to support this feature, however, it is not certain that they function as well as they are supposed to (according to Jonas Gustafsson at Kambi sports solutions).
2.2.1 Topologies

Mirrored cache

A mirrored cache consists of two servers with one working as an active server and the other one is passive. All clients that connect to the cluster are connected only to the active server and when updates to the active server are made, the clients return into control. In the meantime the passive server is updated by a background thread. The replication is done by the active server asynchronously bulking the data to the passive server.

Replicated cache

The replicated topology consists of two nodes or more and replicates all data to all the cache nodes in the cluster. All nodes contain the same data which increases availability and thus reading speed. However, due to synchronous replication, this cache topology provides very poor writing performance as the number of nodes increase.

Partitioned cache

In the partitioned topology, the data in the cluster is distributed between all nodes (each cache node contains unique data) and the combined memory of all the nodes in the cluster can be used to cache data. It is a very scalable topology and the total throughput increases linearly when adding new nodes to the cluster. The distribution of data between the nodes in the cluster can be done synchronously or asynchronously. This topology does not have any consistency issues (since no copying is done between nodes), but if a node goes down, the data stored there is lost completely from the cache cluster.
High availability (also known as Partition- replica)
The high availability topology provides a replication of a node's data to one other node in the cluster. The replication can be done synchronously or asynchronously. As the name says, this topology is the better of two worlds; it has the replication of the cached items and still is very scalable as the portioned topology. With this technology the cluster can obtain both high availability and certain consistency when there is less synchronization.

Figure 6: High-availability cache

Near cache / Client cache
The near cache topology uses a local cache in cooperation with a distributed cache. The topology provides extremely low access times to a local cache, while ensuring fail-over protection by making sure all local data is coherent with cluster data (Pfaeffle, 2008). This technique is therefore extremely fast for read-data while being slow for write-data. It should thus only be used in read-intensive systems.

2.2.2 Data synchronization models
Since a distributed cache can lose data when failing, sensitive data needs to be transferred (or duplicated) to a more reliable data source, such as a database. Distributed caching products generally offer three technologies for synchronizing the cache cluster with a database; cache-aside read/write-through and write-behind.

Cache aside
In a cache-aside scenario, the application is responsible for making sure that the cache is synchronized with the database. The cache does not know about the backing source and thus does not interact with it. Typically, the application first checks the cache for the desired piece of data. If the data is not present in the cache, the application fetches it from the database and puts the data in the cache. Similarly, when writing, the application first adds to the cache and then also to the database, if desired. To make sure that the cache always contains the latest information, the cache is updated each time an object in the database is updated.

Read/Write-through
Is a technique where the information is written to the cache and to the back store. The application treats the cache as the main source of data and lets the cache handle the interaction with the database. Depending on the number of write/read misses, the load on the database can be heavy. (Hennessy & Patterson, p. C-10, 2007). Read/write through allows the
application to treat the distributed cache as the only source of data, thus relieving the programmer of having to handle data replication and synchronization. In this case, the cache handles these features. Read-through means that when the application asks for an item the cache will return the item if it is present in the cache. If it is not present, the cache will call the database and get the object from there. When adding an item to the cache with write-through enabled, the item is first added to the cache and then to the database. The transaction is not successful until the item is added to both the cache and the database, which ensures concurrency (which is explained in section 2.2.4).

Write-behind
In write-behind caching, the application writes only to the cache. The distributed cache then handles the communication with the backing source. This technique can reduce the load on the database significantly (Hennessy & Patterson, p. C-10, 2007). Using the write-through technique in a write-intensive system will not be any faster than writing directly to the database (because each item is written to the cache and the database). So in a write-intensive scenario, the write-behind technique is usually a much better option. Write-behind solves this issue by writing asynchronously (batch-wise) to the database with a given delay, or simply when the database is available. This comes with several benefits:

- Higher throughput since the application no longer has to wait for the database.
- Better scalability.
- Lower load on the database due to batched writes.

2.2.3 Serialization
Serialization is a technique used when sending objects over a network link or when storing objects persistently on a storage medium (such as a database, physical file or a cache). Hawkins and Obermeyer (2002) define serialization as “the process of storing the state of an object instance to a storage medium”. In other words, the serialization process converts the state of the object into stream of bytes so that it can be reconstructed when it reaches to other end. The process of reconstructing the object to an identical clone of its initial state is called deserialization.
The .NET framework provides a built-in mechanism for serializing objects which handles both circular references and object graphs automatically. To make a class as serializable in the .Net language, simply mark it with the attribute "[Serializable]".

The object can then be sent over the network with e.g. NCache’s API through a simple command on the application side, and then be deserialized on the cache server side. All of the evaluated products in this paper use serialization (some provide custom serialization to boost performance).

Note that serialization should be avoided when possible, as the serialization/deserialization process is costly from a performance perspective. But when handling complex application storing and retrieving thousands of objects, serialization should be used as it provides a simple and efficient mechanism, where other methods often are error-prone and becomes less effective as the complexity increases (Hawkins & Obermeyer, 2002).

2.2.4 Concurrency

When working with distributed systems, where one object is stored on more than one place, it is important to make sure that a change of value of a shared object is carried out through the entire system within a certain time period, and in the correct order. This technique is called concurrency. Cleveland and Smolka (1996) describes concurrency as “the fundamental aspects of systems of multiple, simultaneously active computing agents that interact with one another”.

The two most common approaches to handling concurrency in distributed cache systems are the so-called pessimistic and optimistic models (MSDN, Magnus).

- Optimistic: No locks are used in the optimistic concurrency model, instead, the cache uses version tracking to make sure no update overwrites someone else’s changes. Here, the client first sends the object that is to be updated, together with its version number to the system. The object is then only updated if that version matches the current version of the object. Every time an object is updated, its version number is changed.

- Pessimistic: In the pessimistic concurrency model, the client uses a lock to make sure no other client can interfere and change the object until it has been unlocked. A time-out is often provided to make sure the object gets unlocked in the case where the client fails.
2.2.5 Coherency

Coherency is used to make sure that all the caches in the distributed system return the same version of a certain data, i.e. the data is consistent in shared memory. In distributed caches, this quickly becomes a problem since data can be stored in a local cache on the client side, on multiple distributed cache servers, and even in a database or other back-end storage medium. In distributed caching systems, coherency is maintained by the cache platform.

Coherency can be achieved by using a variety of techniques:

- The objects can be given an expiration time in the local caches, thereby forcing the objects to be re-fetched from the cluster upon request.
- The application client can be notified when an object is updated by using event notifications, thereby providing a mechanism to maintain coherency. Some caching platforms allow for push notifications while other only support poll notifications.
- Another way is to only write new objects to the local cache, and then transfer to the cache cluster. Coherency is guaranteed by not allowing shared files to be modified. To update an object, a new version containing the updated information is created with a different version number. This ensures that all copies of an object with a certain version number contains the same data. (Kent, 1987).

2.2.6 What to measure when testing a distributed cache

The testing phase is one of the most time consuming parts in this thesis. It is therefore important to clarify what to test before initiating the testing phase. Felcey (2008) argues that the following aspects are important to keep in mind when considering what to measure, and how to test a distributed cache system:

*Latency*
What is the response time in the system?

*Throughput*
To measure the maximum throughput in the benchmark, we need to make use of multiple threads that concurrently access the cache, and possibly several test machines. By adding threads, the load on the network and CPU will increase and show peaks in the performance counters. The following aspects will be examined in this thesis:

- Writes/sec with different topologies and database synchronization.
- Reads/sec with different topologies and database synchronization
- Interleaving - Write/Read ratios.
• Different eviction-policies

Data types
.net objects/byte[]/xml etc.

Data size
1KB, 2.5KB and 10KB
What is the difference between using many small objects and using few, but large objects?

Serialization/Deserialization
The difference in performance depending of data type/size and serialization technique.

Scalability
How does the throughput change with the number of machines?
How is the cluster affected when a machine is shut down or when a node is added?
Take into consideration that the increase from one to two nodes may not provide the same scalability as an increase from two to four, etc. This is due to the scalability from one node to two nodes causes the cache to begin making backups on second node, which will lead to more cache operations.

Data reliability
A fundamental requirement in distributed caching is that one must be certain that all data is still accurate and available when a server crash occurs or virtual clients crash. The system must be able to receive requests and all transactions must succeed, and the system should simultaneously balance back to a steady state with the primary data and backup copies distributed across the cache environment.

Destructive Testing
To understand what the system is capable of, one must know when it will fail and what the cause is. Destructive tests should include:

• Overloading the cache with a larger amount of data than the memory allocated to the cache is available to hold, to see how the cache will react, compensate, etc.

• Performing a long-term test without a break, where a number of clients accesses data in a large distributed cache to capture the performance characteristics and to determine which outliers that occur.
2.3 The CAP theorem

The CAP Theorem (also known as Brewer’s theorem) states that when building a distributed system (such as a distributed cache), you can choose two of the three desired characteristics:

- **Consistency (C)** - This is achieved when operations on a distributed memory acts as if it were executed on one single node when a request is made (Gilberth & Lynch, 2002).
- **Availability (A)** - Each request will be met with a response (even though it might be stale data).
- **Tolerance to network partitions (P)** - The system should continue to function and act correctly after a network failure where one or more nodes are no longer connected to the system.

Brewer (2012) explains the CAP theorem by imagining two nodes on different sides of a partition. When one of the nodes update the state of an object in the partition, the other node will not have the same data; they will not be consistent. If the goal is to keep the nodes consistent, one of the nodes would act as if it were unavailable. This, however, would be at the cost of availability. The only way to achieve both would be if the nodes were constantly communicating. But to achieve this, partition tolerance would have to be forfeited.

The CAP theorem is however a simplified view of reality. Brewer (2012) explains that the purpose of introducing the CAP theorem was merely to “open the minds of designers to a wider range of systems and tradeoffs”. The “two of three” is not as strict as it may appear because there are techniques which provide flexibility for recovery and handling of partitions (such as the high-availability topology). Brewer (2012) argues that the goal should be to maximize the combinations of both consistency and availability according to the needs of the specific application.

2.4 Databases

A database is a collection of related data (Elmasri & Navathe, 2010), and represents a certain part of the real world. Any change to this part is mirrored in the database. A database is logically structured and has, to some extent, inherent meaning. It is also built for a reason, and with a specific purpose. This means that a random collection of data should not be called a database.

The user accesses the database through a database management system (DBMS). The DBMS is a set of software that works an intermediary between the user and the database. It allows the user (or administrator) to define, construct, manipulate, share, protect and maintain the database.
In distributed caching, databases are often used to store data persistently, as they allow for much more storage capacity than a cache. This is important because some information is too valuable to be lost (e.g. user names and passwords), or take up too much space to fit in a distributed cache cluster. Databases for persistent data storage typically use magnetic or optical memory (non-volatile memory) as they (unlike volatile memory such as a RAM) retain all data when power is turned off (Bekman & Cholet, 2003).

**Terminology in database technology**

According to McCarthy and Risch (2009, p. 83-84), these are some of the most important concepts in database technology:

- **The Relational model**
  A data model which describes reality by storing data in tables.
- **Relational database**
  A database organized as relational models, that is, with all data stored in tables.
- **Relation**
  A table of the type used in the relational model. Can also simple be called ‘table’.
- **Primary key**
  Something that uniquely identifies a particular entity, such as the social security number of a person. In a relational database this is a column or a combination of columns, which always has a unique value for each row in the table.
- **Candidate key**
  In some cases, there are several possible primary keys. These are called candidate keys and you choose one of them to be your primary key. A candidate key in a relational database is, similar to the primary key, a column which has a unique value for each row in the table.

**2.4.1 ACID**

Transactions to a database should possess a set of properties to ensure reliability. These properties are called the ACID properties:

- **Atomicity (A).** Refers to the requirement that the transaction is performed entirely or not at all.
- **Consistency (C).** Ensures that the database is taken from one (valid) state to another. A transaction should be completed without interference from other transactions.
- **Isolation (I).** Isolation requires that even though transactions occur concurrently, a given transaction should appear as if it has been executed in complete isolation from the other transactions.
• Durability (D). Means that after a committed transaction, the changes in the database must persist even after e.g. a power failure or system error.

In modern database systems, the ACID properties are supplemented with concurrency control and recovery methods.

2.5 Return-on-investment

Return on investment (ROI) is a widely recognized and accepted tool for evaluating information systems and is used for making informed decisions based on that evaluation, when return on investments are not easily calculated based on monetary values. It is one of the most popular performance measurements and evaluation metrics in business analysis (Andru & Botchkarev, 2011). ROI is a popular because it is easy to understand, easy to use, encourages cost efficiency and because it’s seemingly easy to perform. Typical metrics when measuring return-on-investment are costs of:

• IT infrastructure
  - Software
  - Hardware
  - License costs

• Labour
  - Salaries/Wages
  - Consultant services
  - System maintenance

• Training

Specifying costs is generally considered to be the easier part of the ROI-calculation (Andru & Botchkarev, 2011), as it can often be turned into real numbers based on hours of work or dollars invested in hardware. The general rule when calculating the return-on-investment is to include “all” of the costs and financial returns. Financial returns can for example be:

• Cost savings
• Cost avoidance
• Increased revenue
• Revenue protection

However, when looking into IT projects, the returns are not always simply financial (J. A. Flinn, 2010). IT projects should rather be seen as a way of generating services and products which the customers in turn pay for. Flinn (2010) suggests measuring returns in IT projects by something called functional yield (FY). Thus, the benefits of an IT investment cannot simply be measured in financial terms. Functional yield depends on four factors:
- Dollars - Original budget vs. how much that was spent.
- Time invested - Intended delivery date contra the date when the system is running.
- Functions - Desired functions contra delivered functions.
- Perceived value - What the business thinks about the system.

Lucas (1999) also brings up indirect benefits of investing in IT, benefits that sometimes are unanticipated when the investment is first made. Such indirect benefits include simplifying the business with customers and encouraging business by using good technology to create a good impression. Also, IT technology can offer flexibility for unexpected events such as peaks in data transfer, lower costs of later projects and reduce time for new product to reach the market. For example, investing in a flexible database infrastructure can save years of development for a company and thus also save extreme amounts of money (Lucas, 1999, p. 104).

But investing in an IT system does not only benefit the customers. Andru and Botchkarev (2011) argue that non-financial assets can improve internal productivity since it can increase effectiveness and decision making, organizational planning and flexibility, resource control and asset utilization.

It is important to point out that there is no standardized way to perform ROI calculations (Andru and Botchkarev, 2011), and one should probably not pay too much attention to the intangible costs, as they are usually not included in the ROI analysis. However, it is important to remember that there are intangible aspects when investing in IT systems that in some cases can lead to a decline in staff morale, productivity or even a declining company image in the form of upset customers as a result from changes in the system.
3. Methodology

The following chapter describes the methodology that was used in this thesis. The chapter begins with a motivation of the selected method and its description, followed by a description of the meetings/interviews that were held with people from Spotify, Kambi and OAS. Furthermore, the chapter contains a section were the conducted tests are explained.

3.1 Method overview

The first step to understanding the task given to us by Scania was to study the area of distributed caching. This was first done through literature studies. This knowledge was then used as a basis when formulating the interview questionnaire for the meetings with external companies with experience of working with distributed caching in practice.

The main purpose of the meetings was to understand how distributed caching can be used in practice i.e. how other companies or developers have used it to solve various technical problems, but also to gather information about costs and workload as basis for the return-on-investment analysis. The interviews also gave us knowledge about the complexity of using distributed caches in some cases. Meetings and/or interviews were conducted with representatives from Spotify, Kambi and OAS (described below/in section 3.2).

Additional meetings were then held with representatives from the FMS who provided information about the system that we looked into. We also held a couple of internal presentations to get feedback from other employees with different knowledge and skills. The idea behind this approach was to make sure that we understood the system and problem correctly and that we were on the right track with our work.

After learning about distributed caching and Scania’s system, we proceeded with selecting which products were appropriate according to the situation (section 3.3 provides more information about this task).

The chosen products (Alachisoft NCache and Microsoft AppFabric) were then tested individually to cover the requirements devised earlier in the process, following a rather extensive implementation phase (section 3.7 describes how the two products were implemented). The test results were then recorded. Following the testing phase, the results were analyzed.

To understand how a distributed caching solution affects a developer’s everyday routines, a meeting was held with a distributed systems expert (Magnus Eriksson). Finally, a return-on-investment analysis was developed where pros and cons regarding both financial and non-
financial aspects were taken into account, including information from other developers as a basis to draw conclusions from.

Figure 8: Method overview
3.2 Distributed caching - Area of use

The following section describes how distributed caching is used in practice. The level at which these caching solutions are described varies. This is due to some companies being secretive about proprietary solutions.

3.2.1 Spotify

Spotify is a Swedish music streaming service with more than 20 million subscribers in various countries across the globe (Music industry, 2008).

Spotify basically uses a three-tier solution for distributing the music, providing a median playback latency of only 265 ms (Kreitz & Niemelä, 2010). The first layer is a peer-to-peer network which consists of the Spotify users’ local caches. The size of the local caches is 10% of free disk space by default. The peer-to-peer layer accounts for approximately 34% of the users’ data usage. The second layer (called the production storage) is a distributed cache located in Stockholm, London and other areas for short distance, containing approximately 90 TB of data. The servers have fast hard drives and lots of ram which provides quick data access. The distributed cache handles the highest amount of traffic which is approximately 55% of all the data being sent back and forth. The third layer, called the master storage, is a series of large databases of approximately 290 TB of data. It works as a distributed hash table and contains every song available on Spotify. Only 9% of the requests are gathered from the master storage. (Kreitz & Niemelä, 2010).

3.2.2 Kambi sports solutions

Kambi is a business-to-business company which offers sports betting solutions for their clients. They employ approximately 240 people in their offices located in London, Malta, Mantila and Stockholm. Kambi’s clientele include Unibet, expect.com and Nordicbet (Kambi, 2012).

Kambi uses two different techniques for solving their main issues:

● Providing the customers with live services on the website (e.g. odds for live betting, pre-match betting etc.).
● Information for back-office users.

Since Kambi works with frequently updated data which is being accessed by multiple users in real time (first issue), they need extremely low access times. This is solved by using local caching on the front-end servers where each object has a given time-to-live before expiring, which allows for quick access to frequently used items. Objects that are not present in the cache are gathered from backend storage (in this case, a large database) and put in the cache.
The back office data is handled by a distributed caching solution (Terracotta). Kambi uses a write-behind data synchronization technique between the cache servers and the database for being able to handle large amounts of writes.

### 3.2.3 OAS (Scania)

OAS (Object and Structure tool) is a system within Scania that manages the company’s product data.

In the OAS system, the primary goal of the distributed cache is to improve the response time for accessing data and to make the system more scalable. This is because they need to handle all the reads of all parts of Scania’s vehicles, engines etc. This goal is achieved by using a cache-aside topology in coordination with near caching. This way, OAS is able to handle very high amounts of reads/sec, since the hit rate of the system is >90% in the clients’ local caches. Coherence is achieved by eviction of modified objects and loading the unmodified objects to the cache. The cache cluster consists of three cache nodes that are homogeneously typed and thus contains different types of data with no replication (partitioned topology).

### 3.3 Distributed caching in FMS

As already mentioned, the current FMS is heading towards a situation where it is no longer able to handle the growing number of data packets. Previous tests (performed by the FMS team) indicate that the database is going to be the next bottleneck, and must be able to handle at least \( X \) (classified) messages per second (given that the current status report interval from the vehicles is maintained). However, in the future it is likely that the reporting interval will increase significantly, meaning that the required throughput will become much greater.

The FMS platform is currently undergoing some major changes, which means that the future design of the FMP is yet not fully decided. However, with the help of our supervisor, we identified three possible future scenarios where a distributed cache could be of use:

**Current Message Platform (MP)**

The message platform must be able to handle large amounts of writes, and thus a cache solution may not appear to be the most natural solution in a write-intensive system, as it is traditionally used primarily for reading objects over and over again. However, today’s distributed caching products offer techniques which allows for write-intensive situations by keeping the data in the cache and then, if necessary, batch data to database when traffic is low (write-behind). Also, since the cache servers can be distributed, they can handle multiple simultaneous writes to the cache cluster which allows for an additional boost in performance.
So adding a distributed cache to the current FMS should off load the database and allow for more requests per second.

*Next Gen FMS*
Common data is the name of a possible future scenario where several internal/external stakeholders share the same basic data. The idea is to store ‘basic data’ in a database from which all stakeholders can read. The type of data stored in this database is still to be fully decided, but it could include vehicle status which a number of stakeholders (classified) would be interested in. In this case, the system would be a lot more read-intensive than earlier, so a distributed cache containing redundancy should be able to speed up the process of retrieving data significantly.

*Geofencing*
Geofencing is the name of applications that use some kind of location system for tracking signals and individuals when they exit or enter predefined areas of interest. An example of this is parents who track their child’s cell phone in real time with a variety of services, such as “Chaperone” from Verzione (LaMarca & De Lara, 2008, p. 88). In Scania, geofencing is used to track vehicles. In some situations it is interesting to know if a driver drives his/her vehicle outside the area of interest (e.g. a country or a city etc.). Scania has a geofencing system today, but it is integrated and makes the calculations within the vehicles. This scenario is based on an ‘offboard’ geofencing system, which will be an independent system in FMS. In this case a distributed cache can be a good solution in managing the huge flow of information of GPS coordinates from all vehicles. If the vehicles send information in real-time at frequent intervals we get much data that may not be of particular interest to save in persistent storage. With the write-behind feature, the cache can send data to the database in a more relevant time interval. To solve the notifications when a driver is outside a predefined area, cache event notification can be used (which is included in some of the evaluated products). If cached data contains GPS coordinates that are outside the predefined coordinate, the cache will send a notification to the application.

### 3.4 Selection of products to evaluate

There are several distributed caching products on the market. In order to find the best suited distributed caching product for REI we had to delimit the selection of products.

The selection was first made from a general perspective where aspects such as cost, product documentation, programming language support and safety aspects were taken into account. The top seven products were then identified based on these criteria and given a number
between 1 and 5 (where 1 is the least important and 5 is the most important). Thereafter, each product’s score was calculated and summarized.

In the next step of the process, we identified the most important and relevant features through meetings held with our supervisor as well as other Scania employees with knowledge within the field of distributed caching. Each feature was then given a number between 1 and 5, depending on how well the product fulfills the feature (1 is the lowest score and 5 is the highest). We then scanned the market for products to identify the most recognized products on the market, where after we browsed each product’s specification to check whether or not the product offers the identified features. One of the most important features of the product is that it supports the Microsoft .Net programming language, since it is the programming language that the developers in REIS use in their daily work. Also, write-behind database synchronization was deemed important because the system is likely to be very write-intensive. The full list of features can be found in appendix A.

Due to limited time we were forced to make a selection of two distributed caching products for evaluation. The product that provided the most of the desired features (and thus also the highest score in the checklist) was NCache, so the decision to include it in the testing was easy. The second choice came to be between Scaleout Stateserver, Oracle Coherence and Microsoft AppFabric. The product that stood out the most to us was AppFabric due to it being free of charge, .net-based and relatively feature loaded. The other two products scored low on some important criteria. Oracle Coherence did not provide full .Net support and is expensive, and Scaleout Stateserver was lacking some important features (although having full support for .Net). Also, Scania was interested in comparing a cost-free product with a commercial product. Therefore we finally decided to add AppFabric to the testing phase.

3.5 Evaluated distributed cache products
The following section provides detailed information about the evaluated products.

3.5.1 Alachisoft NCache Enterprise edition
Alachisoft is a company with its headquarters in San Ramon, California, USA. They describe themselves as the “leading provider of high performance solutions for .NET development community”. Their product, NCache, is a purely distributed cache product for N-tier and grid computing .NET applications that need to boost their performance. The version of NCache used in this report is ‘NCache version 4.1 SP1 for .NET’. (Alachisoft, 2012, Company Overview)
Features
The most important features of NCache are listed and explained in appendix A.

System Requirements
In the NCache Installation Guide, Alachisoft provides system requirements for NCache in general. At Alachisoft Download Center, one can find what kind of prerequisite software a specific version of NCache requires.

Supported operating systems:
According to Alachisoft, NCache is not bound to a specific operating system. If the version of .NET that is needed for the specific version of NCache that you are going to use is available on the operating system you have, NCache should work. However, using Windows 2003/2008 Server (64-bit) for Cache Servers is recommended.

- Supported Architectures:
  - 32-bit (x86)
  - 64-bit (x64)

- Hardware requirements:
  - Minimum Hard Disk Space: Not specified. But it says that there is no need for extra ordinary disk space since NCache makes no heavy use of disk space.
  - NCache is a highly multi-threaded application and can make use of all the extra CPUs and cores that a server has. The most common setup is a machine with dual-CPU where each CPU is dual-core.
  - NCache can make use of two network interface cards to increase the throughput. This is not a requirement but it is recommended.
  - The amount of primary memory that a server needs depends on how much data you want to store on the cache nodes. The processes of NCache itself take approximately about 40-50MB memory on each cache node. There is also a 15% overhead of whatever you cache.

- Prerequisite software:
  .NET Framework 4.0/3.5/2.0. To run some applications other software may need to be installed. For example ASP.NET needs to be installed to be able to run NCache enabled web-applications. You also need Microsoft Visual C++ 2005 Service Pack 1 Redistributable Package.

Topologies
NCache has a rich set of topologies to meet the most common needs of a clustered cache implementation. The product provides topologies that suit small cache environments (mirrored cache) to larger clusters consisting of hundreds of nodes. The following topologies are supported in NCache:
- **Mirrored Cache (2-server active/passive)**
- **Replicated Cache**
- **Partitioned Cache**
- **Partitioned-Replica Cache (also known as High-availability)**
- **Client Cache (a Local Cache connected to a Clustered Cache)**
  - In NCache, this topology uses a local cache on the client node that is connected to the cluster of cache nodes. The local cache allows for caching of frequently used data closer to the application which boosts the performance even more than when just using a cache cluster. However, if there are many updates or additions to the cache, this topology will be slower than when using only a cluster. This is due to the extra time it takes to update both the local cache and replicate that update to the cluster. The client topology can be combined with any of the other topologies or be used as a stand-alone cache.

To select a topology in NCache, simply start a new project in the GUI and make the choice in the drop list. If the replicated or high-availability topology is chosen, the replication can be done either synchronously or asynchronously.

(Alachisoft, 2012, *NCache: Caching Topologies*)

**Database synchronization**

NCache provides all the database synchronization techniques described in section 2.2.2:

- Read/Write-through
- Write-behind
- Cache-aside

Read-through and write-through is added by implementing the IReadThruProvider and IWriteThruProvider interfaces, and then registering the code with the cache cluster by browsing for the assemble file in the back store tab in NCache GUI. After the assembly is registered it is copied to all the cache servers and called upon when NCache needs to access the back-store.

(Alachisoft, 2012, *NCache: Distributed Caching Features.*)

**Cache structure**

NCache is based on peer to peer cluster architecture. This means that every server is connected to every other server in the cluster, and there is no single point of failure. This is to maintain 100% uptime if a server goes down. The first node connected to the cluster becomes the cluster coordinator. If it goes down the role is passed on to the next senior-most server in the cluster. If a new node is to be added into the cluster, NCache needs to know at least one other node in the cluster, but not all of them. Once it connects to one of the nodes in the cluster it asks for
the identity of the cluster coordinator, and asks for permission to add the new node to the membership list. The coordinator adds the node and informs all other nodes in the cluster that a new member has been added. The coordinator also informs the new node about all the other members, so that the node can establish TCP connection with them. This differentiates NCache different from other cache products that typically have the configuration file stored at a specific location, which can make the cluster go down if that location becomes unavailable. NCache has the configuration information replicated on every node which makes the cluster safer against single point failure.
(Alachisoft, 2012, NCache: Self Healing Dynamic Clustering)

**Configuration/administration**

NCache gives the user two options for configuring and administrating the cache cluster. It can be done with command-lines or with the GUI-tool called NCache Manager. NCache Manager provides all options needed to configure the cache cluster, and is simple to use. The tool is a central management system, which means that the user can add and remove nodes at runtime and see statistics for all nodes and administrate the clients from a single point. To change some of the configuration options, the cache cluster needs to be stopped. So it is a good idea to decide what options should be used in advance before deploying NCache in a production environment.
(Alachisoft NCache 4.1 - Online Documentation, 2012, Configuring NCache)

![Figure 9: NCache Manager](image-url)
API
The namespace Alachisoft.NCache.Web.Caching contains the API for connecting the server and cache in NCache. This library must be referenced in the .Net application in order to get started with developing to the cache cluster.

The NCache Client API consists of many features to make high performance and scalable applications. Some of the methods available in the API are; tags, item versions, event notification, cache dependency and more. The most basic methods are Add(), Insert(), Get() and Delete(). These methods are the most common in all cache environments and similar between different frameworks.

Examples of the basic methods in the client API:
Add
DataCache.Add(String, Objects) - Adds an object to the cache.
Insert
DataCache.Insert(String, Objects) - Adds or updates an object in the cache.
Get
DataCache.Get(String) - Gets an object from the cache using the specified key.
Delete
DataCache.Delete(String) - Deletes an object from the cache if the object already exist in the cache.
(Alachisoft NCache 4.1 - Online Documentation, 2012, Client Side API Programming)

Security
The NCache product has built in authorization and authentication security. The product lets you specify users (can only make runtime API calls to a specific cache) and administrators (can manage the cluster). NCache uses Microsoft Active Directory Service for authentication of users. The user should exist under a given domain in LDAP (Lightweight Directory Access Protocol). In late October 2012, Alachisoft released version 4.1 SP2 which provides data encryption feature to make sure that data traveling between client and server or between cluster nodes secure. This will prevent the user data leakage even if data packets are sniffed from the network. However, there is no information about what type of encryption they use.
3.5.2 Microsoft AppFabric Caching v1.1
The multinational corporation Microsoft offers a cache product called Microsoft AppFabric 1.1 for Windows Server. The product has two major features; hosting and caching. In this thesis we are only interested in the AppFabric caching feature which adds a distributed cache to Windows Server that makes it easier to scale out .NET applications.

Features
The most important features of AppFabric are listed and explained in appendix A.

System Requirements
The Microsoft AppFabric 1.1 for Windows Server download homepage specifies that the following system requirements are needed for using AppFabric 1.1 (note that not all prerequisite software are needed if you only going to use the caching feature in AppFabric):

Supported operating systems:
AppFabric can be installed on the following operating systems:

- Windows 7
- Windows Server 2008 R2
- Windows Server 2008 Service Pack 2
- Windows Vista Service Pack 2

- Supported Architectures:
- 32-bit (x86)
- 64-bit (x64)

- Hardware requirements:
- Minimum Hard Disk Space: 2GB
- Computer with an Intel Pentium-compatible CPU that is 1 GHz or faster for single processors; 900 MHz or faster for dual processors; or 700 MHz or faster for quad processors.

- Prerequisite software:
Install the following pre-requisite software. If this software is not already installed, install in the order presented below:

- All features of AppFabric require a .NET Framework version to function. The specific version required is dependent on which features you wish to use:
  - Hosting services require Microsoft .NET Framework 4
  - Hosting administration requires Microsoft .NET Framework 4
  - Caching service requires Microsoft .NET Framework 4 and optionally requires Microsoft .NET Framework 3.5 Service Pack 1
- Cache client requires either Microsoft .NET Framework 4 or Microsoft .NET Framework 3.5 Service Pack 1
- Cache administration requires Microsoft .NET Framework 4
  - Internet Information Services (IIS) 7
  - Internet Information Services (IIS) 7 Hotfix #980423
  - IIS Web Deployment tool
  - Windows PowerShell 2.0 (final version) (this is not required for Windows 7 and Windows Server 2008 R2 users)

**Topologies**

*Partitioned*

In AppFabric the default topology is partitioned. By typing Get-Cacheconfig in PowerShell you will get the output of the cache and it will display some settings in a list. One setting is “CacheType”, which will always be set to partitioned (even when high-availability is used). To be able to use the other topology the user must change the setting “secondaries” to 1, which will enable high-availability.

The partitioned topology in AppFabric works as described in the theory section 2.2.1.

*High-Availability*

As mentioned, selecting high-availability is done by changing the parameter of “secondaries” to 1 in the cache configuration in PowerShell. To disable the high-availability feature again, simply set the parameter to 0. By default the option is set to 0, and high-availability is disabled when a cache is created. To be able to run this topology on the cache cluster, all nodes need to be running on a supported operation system. In the time of writing, the only supported operation systems for the high-availability feature are the Enterprise Edition (or higher) of Windows Server 2008 or Windows Server 2008 R2.

(Microsoft, 2012, *High Availability (AppFabric 1.1 Caching)*)

*Local Cache*

Local cache is also an available feature in AppFabric. When enabled, the cache clients stores a reference to the data object locally on the Application client. This means that the speed of retrieving objects will increase greatly if they are available in the application memory. If the local cache does not contain the specific object, the object will be retrieved from the cache cluster and then saves it locally and uses that same object. How long a cache item is stored in the local cache depends of several factors. These factors are the maximum number of objects in the cache and the invalidation policy. For local cache there are two types of invalidation; notification-based invalidation and time-out-based invalidation.

(Microsoft, 2012, *Cache Clients and Local Cache (AppFabric 1.1 Caching)*)
**Cache structure**
AppFabric needs a cluster manager that can be accessed by the lead hosts. It can be stored in an SQL database or in a shared folder as an XML file. The cluster manager’s responsibility is to keep the cache cluster running and available. The cluster only needs one cache host as lead host for the cluster to work, but if you have two or more lead hosts all those hosts must be up and running for the cluster to remain available. The change of lead hosts can be modified in the XML file.

**Database synchronization**

*Read-Through*
The read-through and write-behind features were added in the release of AppFabric version 1.1. If an item does not exist in the cache, a read-through provider can be called when the cache detects a missing item. The provider will then perform the data load, usually from the backend store.

*Write-Behind*
With the write-behind technique the cache can batch down the data from the cache to a backend store. In AppFabric this technique is enabled by setting `WriteBehindEnable` to true in the PowerShell when registering the read/write provider. When registering the provider one must also set the interval that the cache uses to write data to the back store. In AppFabric, the minimum duration that can be set is 60 seconds. (Prabhakar, P. 2011)

*Cache-aside*
When using cache-aside the application needs to handle the reload of data to the cache from the data source. This means that if the data is not present in the cache the AppFabric cluster will not reload the data into the cache from the back store.

(Microsoft, 2012, *Read-Through and Write-Behind (AppFabric 1.1 Caching)*)
(Microsoft, 2012, *Programming Model (AppFabric 1.1 Caching)*)

**Configuration and administration**
The configuration and administration of an AppFabric cache cluster is done in PowerShell, which is a command-line shell. At MSDN (*Caching Powershell Cmdlets (AppFabric 1.1)*) there is a section that provides all the PowerShell commands for AppFabric 1.1 Caching.
The cache client API in AppFabric can be found in the Microsoft.ApplicationServer.Caching namespace. The namespace provides the API that allows developers to develop applications that uses the AppFabric caching libraries from the assemblies’ microsoft.applicationserver.caching.client and microsoft.applicationserver.caching.core. (Microsoft, 2012, Microsoft.ApplicationServer.Caching Namespace)

There are many features in the client API such as tag methods, notifications methods etc. However, the most common methods are the basic cache operations, such as Add(), Put(), Get() and Remove(). The methods for these operations are very simple and they are similar across all caching frameworks. (Microsoft, 2012, Cache Client API Overview)

Examples of the basic methods in the client API:

Add
DataCache.Add (String, Object) - Adds an object to the cache.

Put
DataCache.Put (String, Object) - Adds or updates an object in the cache.

Get
DataCache.Get (String) - Gets an object from the cache using the specified key.

Remove
DataCache.Remove (String) - Indicates whether an object from the cache is removed.

(Microsoft, 2011, DataCache Methods)
Security
By default AppFabric 1.1 provides both encrypted and signed communication between cache clients and the cache cluster. The user also has to have a Windows account in the list of allowed accounts before the user can access the cache cluster.

There are two modes of protection. Either it is set to None, which means that the data sent between the cache cluster and cache clients are not encrypted or signed. Otherwise, protection is set to Transport (default), which means that the data sent is encrypted and signed. In the None state the cluster is highly exposed to network attacks that log or modify the data. It also gives the ability of any cache to communicate with the cluster, even though that specific cache client has not explicitly been granted access. However, in the Transport state, only users with Windows accounts that are in the list of allowed accounts are permitted to access the cache cluster.

In AppFabric there are three protection levels for the data sent between the cache cluster and cache clients. When the security mode is set to None, the only protection level that can be set is None. When the security mode is set to Transport the user can choose between two additional protection levels; Sign or EncryptAndSign. The Sign level protects the data on the network from being manipulated and the EncryptAndSign level encrypts the data before it is signed. (Microsoft, 2012, Security Model (AppFabric 1.1 Caching))

3.6 Design of performance tests
The main purpose of the testing was to simulate potential future situations in which a distributed cache solution could be of use. The tests should provide sufficient data for the FMS group to make a decision whether or not distributed caching is a viable option for these scenarios.

The activities described by Ammann and Offutt (2008) were followed in the testing phase (figure 11). The first task was to design the test requirements. In cooperation with our supervisor, we identified three scenarios in Scania’s current and future system to evaluate distributed caching as a solution for reducing load on the database, or for increasing efficiency when reading and writing data. This information and the theory from section 2.2.6 were then used as a basis for the requirements design, which in turn was converted into code for execution.
3.6.1 The FMP scenarios test

The tests were first conducted with NCache in a local environment. Since AppFabric requires that all cache nodes are running on an operation system of Windows Server 2008 Enterprise (or higher) we were unfortunately not able to install the software on the local machines due to internal policies. Instead, the AppFabric tests were run on virtual machines, and thus did not have the exact same hardware as the local machines. However, the machines that were most similar to the local machines were used, in order to get as fair results as possible under the circumstances. The test was run on a 100 Mbit network. Multiple threads were used to maximize the performance of the application. The appropriate number of threads was determined by adding additional threads, one by one, until the CPU or network was utilized to its maximum, or until the performance of the software started to decline. To avoid threads reading or writing the same object, a simple mutual exclusion lock was used to lock critical sections.

The database connected to the cluster had a single table with two columns. The first column contained the vehicle identification number (VIN), which is unique for all vehicles and thus it was used as our primary key. The second column held the objects in a byte array format. The object sizes in the tests were determined through examining the current and projected message types. Today’s current status message is approximately 1KB in size and includes both positioning and vehicle data. The future messages are unlikely to be much larger and therefore the largest object in our tests was set to 1024B. The smallest object was set to 256B which is a bit larger than the vehicle data of the current status message, minus the positioning data. The test protocol can be found in appendix B. Windows performance monitor (perfmon) was used to monitor and record all statistics from the tests. The test scenarios are described in the following section.
3.6.1.1 Scenarios

Scenario 1 - Current Message Platform (MP):
The purpose of the first scenario is to simulate FMS' current MP. In this case, the RTC sends a current status message to the MP platform, which is then written to the distributed cache cluster. After the message is stored in the cache, it is then read from another source and sent on to the FMP. Thus, each message is written to the cache only once, and read only once from the consumer, creating a read/write ratio of 1:1.
The scenario requires that the cache cluster is able to handle a very high amount of writes, which means that there is no time for constant synchronized replication of data between the cache servers. This leaves us with two topologies; partitioned and high-availability. Since we want to keep the risk of losing data due to a failure to a minimum, we decided to test the high-availability topology and monitor how far it can be pushed. The topology also allows for bigger amounts of data than a replicated topology, which is good when handling high amounts of writes. A database can only handle a relatively low amount of single writes/second, so in order to handle the high amount of incoming writes, asynchronous bulked writes (from the cache cluster to the database) will be used (write-behind).

Scenario 2 - Next Gen FMS:
This scenario aims to simulate a situation where the RTC sends messages with a specific interval and the message is read by a number of consumers. The consumers represent stakeholders within Scania's organization who want to take part of the data. We have previously identified eight internal stakeholders, but it is believed that there will be additional stakeholders in the future. Thus, we chose a read/write ratio of 10:1. Since the heavy part of this scenario will be the reading, less pressure will be put on the database, which is the reason for choosing write-behind (this was changed from write-through due to bad performance), since it is both secure and performs well as a synchronization model. The data will be kept on every cache server with asynchronous replication (changed from synchronous) to ensure that it is kept "clean" and always available to the consumers. The test should evaluate whether or not the cache cluster is able to handle a high enough throughput for this topology to be useful in the current scenario. Due to the high amount of reads, the test was first supposed to include client caches on the application servers. However, after a few test runs, we discovered that this was not the best option due to the writing hindering the throughput too much.

Scenario 3 - Geofencing:
The third scenario is designed to simulate a geofencing scenario where the vehicles' c200 modules send approximately 10 messages during a given time, and the message is consumed (read) by a stakeholder approximately once during that time. Due to the high number of writes, the cluster will use a high-availability topology, and updates asynchronously between the
nodes. This way, the cluster can handle large amounts of data and still provide a copy of the data on separate nodes. Just like in scenario 1, we will here use write-behind database synchronization to allow for large amounts of writes.

The following test environment was used for all three scenarios:

![Test Environment diagram](image)

The hardware setup for the FMP scenarios test with NCache can be found in appendix C.

### 3.6.2 The product and database performance test

The purpose of the second test was to get an overview of how AppFabric, NCache and a database perform against each other independently. Thus the test of the distributed cache cluster operated without any data synchronization to a database. In the database test we used a table with a key column and an object column. The key was still the VIN number and the object was a 1024B byte array. In The agenda of this test can be found in appendix B. The test was run on a 1Gbit network.
3.6.2.1 Products performance with partitioned topology

To test the products performance the cache cluster was configured with a partitioned cache structure. The partitioned cache structure is also an interesting topology when you want an indication of how much a cache node and cluster can handle in terms of writes.

The test was done with only writes to the cache or only reads to the cache and there was no database connected. The object we tested with was a heavyweight 1024B byte array. The cluster consisted of one to two nodes and two application client. During the test perfmon was used to log all statistics from the cache cluster.

The following test environment was used for product performance scenario:

![Test Environment](image)

The hardware setup for the products performance test with partitioned topology can be found in appendix C.

3.6.2.1 Database performance

The second scenario in the second test was to see how the database performs when writing and reading heavyweight objects (1024B). This is important because all test scenarios require persistent storage of the objects. Therefore, writing to the database will eventually determine how high the system’s throughput can get. If the vehicles continuously send more data to the cluster than the cluster can send to the database, the cluster will (sooner or later) have to evict data that has not yet been stored persistently.
In this test, one application client was used, running multiple threads that updated the database with the 100 000 objects. After the client had updated the database the objects were read, again with multiple threads. No bulk-operations were used to write to the database. During the test, perfmon was used to log all statistics from the database server.

The following test environment was used for the database performance scenario:

![Test environment for database performance](image)

**Figure 14: Test environment for database performance**

The hardware setup for the database performance test can be found in appendix C.
3.7 Implementation

The following section describes how the distributed caching products were implemented prior to the testing.

3.7.1 Distributed Cache Interface

The TestScenarios interface (figure 15) uses a simple interface which combines the two distributed caching products. The interface provides all communication with the cache cluster and is controlled from the application servers. The basic functions for communicating with the cache cluster are very similar between NCache and AppFabric, which made the implementation process a lot easier.

3.7.1.1 Scenario testing

Switching between NCache and AppFabric is done simply by selecting the desired platform in the radio button list on the user interface (UI). Once the platform is chosen, it is simply a matter of clicking on the desired functions/methods, object size and read/write ratio to start the testing. The TestScenarios interface implements the following functions/methods:

- **DB and Cache**: Uses the write-behind technique (which is used in all three scenarios) to add and get the selected object to/from the cache and the database.
- **Only DB**: Adds and gets the object to/from the database only.
- **Only Cache**: Adds and gets the object to/from the cache cluster only.
- **Clear DB**: Deletes all objects from the database.
- **Clear Cache**: Deletes all objects from the cache cluster.

3.7.1.2 Distributed Cache General Functions

The CacheFeatures interface (figure 16) implements the following functions/methods:

- **Insert**: This method/function adds an object to the cache and database using either write-through or write-behind, depending on which one is selected. The object consists of the string and integers typed into the textboxes.
- **Update**: The button ‘Update’ updates the object containing the string entered into the VIN textbox with the message entered into the Message textbox.
● **Load**: Loads the object containing the string entered into the VIN textbox using the read-through option.
● **Clear**: Deletes all entries to the cache cluster and the database.
● **Expiration**: Sets the object’s expiration time. I.e., the time before it is automatically removed from the cache cluster.

### 3.7.2 NCache implementation

In NCache, the desired cache cluster settings are selected when creating a new project in the NCache Manager. The first thing to choose is a name for the cluster. After that, the topology, size of the cache nodes, eviction policy etc. are specified. We selected topology size and eviction policy depending of which of our tests we were about to carry through. The agenda (which can be found in appendix B) provides information of the settings used in each test. After the desired options are set, it’s time to add cache nodes to the cluster. This can be done by browsing the network or entering a node’s IP address. When this configuration is done the cache cluster should work as a key/value store.

To implement the write-behind/read-through features of NCache, simply go to the options menu and select ‘back store’. In this section of NCache Manager the user can add a read-through provider and a write-through/write-behind provider. The providers are added by browsing for the assemble file that is built in e.g. Visual Studio 2010. The assemble file should be located in a subfolder named ‘deploy’ in the installation folder of NCache. After adding the assemble file the only thing left is to add the connection string. The connection string should include the source of the database with IP-address, id and password. An example of a connection string:

"user id=admin; password=admin; server=192.168.xxx.xxx,1433; Trusted_Connection=yes; Integrated Security = false; database=ExjobbDB; connection timeout=15"

To use write-behind, simply check the box next to the write-through provider implementation.

### 3.7.3 AppFabric implementation

AppFabric is a bit different from NCache in terms of creating a cache cluster. In AppFabric, the first step is to open the AppFabricServer Configuration Wizard. In the wizard, click all the way down to the tab called ‘Caching Service’. In this section the system-level configuration for the caching service can be set. Start by checking the box ‘Set Caching Service configuration’. After that, choose how the configuration file should be saved. It can be saved in a XML-file on a location all nodes can access or in a SQL server, depending on if your cache hosts operate on a domain or a workgroup. In our tests, we chose store the configuration file on the database server in a database called AppFabricConf. After deciding what store to use, check the radio
button named ‘New cluster’ and choose cluster size in the drop-down list. This procedure must be done on all nodes that are to be included in the cluster. The only difference in adding the remaining cache nodes is that in the final step you have to check the radio button ‘Join cluster’ instead of ‘New cluster’.

Move to the next tab and set the configuration of the cache node. Select node ports and make some Windows firewall exceptions choices. The only thing that is left to do after this is to press ‘Finish’ and close the Wizard.

The next step is to open the Caching Administration PowerShell that comes with installing AppFabric, to operate the cache cluster. Operating the cache cluster is easy. AppFabric offers easy commands such as:

- **New-Cache myCache** - creates a new cache named myCache
- **Start-CacheCluster** - Starts the cache cluster (as the command suggests).
- **Get-Cachehost** - Shows all hosts connected to the cluster.
- **Get-CacheStatistics** - Shows statistics for the specified cache, such as: size, number of objects, number of cache misses and requests.
- **Get-CacheConfig** - Shows configuration details for the specified cache, such as: Cache name, topology and eviction type.

It is also possible to make changes to the cluster through an XML-based configuration file which can be imported and exported using the PowerShell. By default the cluster use the partitioned topology. To change this to high-availability, which we used in our test of the FMP scenarios, write the command ‘Set-CacheConfig myCache -Secondaries 1’ or use the XML-file configure file and set the ‘secondaries’ attribute to ‘1’. Remember that the cluster must be stopped in order to make use of the Set-CacheConfig command.

To be able to use write-behind in AppFabric you need to register your provider which is the assemble file containing the server code. This is done by typing the following command line:

```powershell
Set-CacheConfig TestCache -ReadThroughEnabled true -WriteBehindEnabled true -WriteBehindInterval 60 -ProviderType "ClassLibrary2.Provider, ClassLibrary2, Version=1.0.0.0, Culture=neutral, PublicKeyToken=078c62513e9186d3" -ProviderSettings @{"DbConnStr"="<your connection string>"};
```

If there is no read-through provider implemented in the provider code, set ReadThroughEnabled to false. The WriteBehindInterval is the time interval to write cached items to the backstore. In ProviderType pass the string that you got when registered the assemble file into GAC. In the last parameter -ProviderSettings pass the connection string to
the database. In our tests we used the exact same string as in NCache; “DBConnStr”=”user id=admin; password=admin; server=192.168.xxx.xxx,1433; Trusted_Connection=yes; Integrated Security = false;database=Exjobb; connection timeout=15”.

3.7.3 Server-side implementation
All code regarding functions operating from the cache-cluster is stored on each cache node in a dll-file. This includes e.g. read/write-through code, code for write-behind handler and pre-loading the cache cluster with files at startup. The dll-file needs to be signed with a strong file name and registered in the global assembly cache (GAC) on each cache node. The developer must specify information for the cache application such as SQL-strings for communicating with the data source, and implement the cache applications’ methods for communicating with other entities (e.g. a database). The following server-side functions were implemented for the tests:

- **Read-through**: Uses the classes from the AppFabric and NCache API’s which implements the DataCacheStoreProvider (AppFabric) and IReadThruProvider (NCache). The class that implements these interfaces receives requests to read the object from the database, should the object not be present in the cache cluster.

- **Write-through**: Write-through is implemented using the classes implementing the DataCacheStoreProvider (AppFabric) and IWriteThruProvider (NCache). The class that implements these interfaces receives requests to store the object in the database when an object has been added to the cache cluster.

- **Write-behind**: The write-behind technique does not need any additional server side implementation, when write-through is implemented. Instead, it is enabled in the powershell for AppFabric or in the NCacheManager for NCache.

Remember to put locks in the critical sections of the server code to prevent mutual exclusion if you have multiple threads from your client implementation accessing the server code.

3.7.4 Database structure
The structure used in the database was quite simple. We used a single table with only two columns, as in a key/value store. The first column was VIN (representing the Vehicle Identification Number and is unique for each vehicle) which also was our primary key. The second column was the object, which in our case was a byte array. The name of the column was MessagePayload (to draw parallels to what the object contains in a real world scenario).
3.8 Method critique

The FMP scenarios tests run on a 100 Mbit network, which is not optimal when testing distributed caching products. Generally when testing distributed systems, the network should not be a bottleneck. In our tests the network becomes overloaded when the largest objects (1024B) were transmitted. This meant that we could only send out the amount of items needed to max out two cache nodes in the CPU. The third node was impossible to max out due to the network's limited capacity. It is also possible that the network load during the testing had some influence on the test results. The cluster and database tests were run on a 1Gbit switch, so in this test the network capacity was not an issue. Another aspect is that the hardware of the cache nodes had an influence on the result. This is due to the case that a more powerful cache node can handle more objects without the CPU being the bottleneck. The difference in our hardware setups for the AppFabric and NCache tests may therefore also have influenced the results of the products. However, we tried to minimize this source of error by trying to use as similar setups as possible. The test results could have been even fairer if all tests had been run on the virtual environment. Unfortunately, this was not an option since we were running out of time when the virtual environment was acquired for testing.
4. Results and Analysis

In this chapter, we begin by presenting the results of our tests. The results are followed by a section where we analyze and interpret the results. Next follows a section including the Return on Investment calculation and a discussion of what a developer’s work day with a distributed caching system can be like.

4.1 Performance tests

The following section contains the results from the performance tests. First, we display the results of the FMS scenario tests with NCache and Appfabric on separate charts. The comparison of the products without an underlying backing source is then displayed together in one chart. Lastly, the database test results are presented on an individual chart.
4.1.1 Test 1 - The FMP scenarios

Scenario 1 - Current MP

**AppFabric**

![AppFabric Graph](image)

<table>
<thead>
<tr>
<th></th>
<th>1 node</th>
<th>2 nodes</th>
<th>3 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>256B</td>
<td>8872</td>
<td>11753</td>
<td>12544</td>
</tr>
<tr>
<td>512B</td>
<td>6853</td>
<td>9677</td>
<td>12138</td>
</tr>
<tr>
<td>1024B</td>
<td>6449</td>
<td>9131</td>
<td>11573</td>
</tr>
</tbody>
</table>

*Figure 18: AppFabric. High-availability. 50% reads / 50% writes*

**NCache**

![NCache Graph](image)

<table>
<thead>
<tr>
<th></th>
<th>1 node</th>
<th>2 nodes</th>
<th>3 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>256B</td>
<td>15233</td>
<td>16209</td>
<td>18127</td>
</tr>
<tr>
<td>512B</td>
<td>15163</td>
<td>16524</td>
<td>15756</td>
</tr>
<tr>
<td>1024B</td>
<td>6905</td>
<td>10352</td>
<td>14630</td>
</tr>
</tbody>
</table>

*Figure 19: NCache. High-availability. 50% reads / 50% writes*
Scenario 2 - Next Gen FMS:

**AppFabric**

![Graph showing AppFabric's performance with different node configurations and data sizes.]

<table>
<thead>
<tr>
<th>Data Size</th>
<th>1 node</th>
<th>2 nodes</th>
<th>3 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>256B</td>
<td>8649</td>
<td>13104</td>
<td>17920</td>
</tr>
<tr>
<td>512B</td>
<td>7547</td>
<td>12060</td>
<td>15268</td>
</tr>
<tr>
<td>1024B</td>
<td>7116</td>
<td>9903</td>
<td>13439</td>
</tr>
</tbody>
</table>

Figure 20: AppFabric. High-availability. 90% reads / 10% writes

**NCache**

![Graph showing NCache's performance with different node configurations and data sizes.]

<table>
<thead>
<tr>
<th>Data Size</th>
<th>1 node</th>
<th>2 nodes</th>
<th>3 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>256B</td>
<td>20425</td>
<td>24515</td>
<td>32791</td>
</tr>
<tr>
<td>512B</td>
<td>14283</td>
<td>18619</td>
<td>25444</td>
</tr>
<tr>
<td>1024B</td>
<td>8906</td>
<td>14413</td>
<td>16386</td>
</tr>
</tbody>
</table>

Figure 21: NCache. High-availability. 90% reads / 10% writes
Scenario 3 - Geofencing:

**AppFabric**

![Graph showing AppFabric's performance with different node configurations and request sizes. The x-axis represents different request sizes (256B, 512B, 1024B) and the y-axis represents request/s. The graph compares performance across 1, 2, and 3 nodes, with 10% reads and 90% writes.]

**Figures:**
- Figure 22: AppFabric. High-availability. 10% reads / 90% writes.
- Figure 23: High-availability. 10% reads / 90% writes.

**NCache**

![Graph showing NCache's performance with different node configurations and request sizes. The x-axis represents different request sizes (256B, 512B, 1024B) and the y-axis represents request/s. The graph compares performance across 1, 2, and 3 nodes, with 10% reads and 90% writes.]

*Request/s (1 node, 2 nodes, 3 nodes)*
- **AppFabric**
  - 256B: 6707, 6936, 9147
  - 512B: 6024, 7310, 9559
  - 1024B: 5133, 7240, 9240

*Request/s (1 node, 2 nodes, 3 nodes)*
- **NCache**
  - 256B: 11974, 12295, 16718
  - 512B: 12065, 12562, 12709
  - 1024B: 8706, 10409, 11615

*Note:* The data points represent the average request rates in requests per second (Requests/s) for different node configurations and request sizes.
4.1.2 Test 2 - The products and database performance

**Products performance with partitioned topology**

![Comparison of Writes/s and Reads/s - AppFabric and NCache with 1024B objects](image)

Figure 24: Comparison of Writes/s and Reads/s - AppFabric and NCache with 1024B objects

**Database performance**

![Read/Write 100 000 objects to the database](image)

Figure 25: Read/Write 100 000 objects to the database
4.2 Interpretation of the results

4.2.1 AppFabric vs. NCache

Scenario 1 - Current MP
Figures 18 and 19 show that NCache provides a significantly higher throughput compared to AppFabric, especially when it comes to smaller objects. AppFabric displays a clear increase of throughput when another node is added to the cluster, although the gap from one to two nodes is larger than that of two to three. The scalability for NCache seems to be better with larger objects, but we can still see an increase of throughput of the smaller objects as well.

Scenario 2 - Next generation FMS
Again in figures 20 and 21, NCache seems to provide much higher throughput than AppFabric does. The difference is significantly larger when smaller objects are being transferred across the network. NCache shows a higher throughput, while AppFabric appears to have better scalability. However, as the object size grows, the difference between the two products shrinks. When handling small objects of 256 bytes, NCache is approximately 2.5 times more efficient on one node, and then shrinks to be approximately twice as fast on three nodes. But when the objects are 1024 bytes large, the gap is much smaller. Both products perform significantly better than in the previous scenario, which is natural since a read operation is generally much faster than a write operation.

Scenario 3 - Geofencing
Both NCache and AppFabric display a lower throughput than the previous scenarios, which is explained by the increased amount of writes to the cache clusters. And just like in the other tests, NCache performs better, especially on smaller objects. Again, AppFabric displays an almost linear scalability, while NCache appears to have a less steep slope (especially when handling objects of 512 bytes. Also, NCache shows a significant (unexplained) increase in throughput after cache server three is added. The test shows that the distributed cache products can perform well even when exposed to very high amounts of writes.

Database synchronization:
All scenarios used write-behind as database synchronization, and in all the tests of these scenarios we had an approximate write speed of 1400 writes per second to the database with NCache. This is due to the bulk write-operation NCache use to write down data with write-behind. However, the write-behind functionality in AppFabric was much slower in the given tests. AppFabric had an approximate write speed of 800 writes per second to the database.

In summary, NCache beats AppFabric in all tests, both in the performance of requests/s to the cache cluster and how fast it writes down data to the database with the write-behind feature.
However, AppFabric shows that it can handle a great amount of requests compared to a database.

**List of advantages and disadvantages of the two evaluated products**
During our tests and evaluation of the products we have discovered a number of good and bad aspects with the products in question. The following list of pros and cons highlights them:

**NCache**

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Many features (See appendix X for a feature comparison).</td>
<td>● Expensive compared to AppFabric</td>
</tr>
<tr>
<td>● Includes all of the most popular topologies.</td>
<td>● NCache forum is very inactive. Contact should be made by phone or mail if you need a fast answer to a question.</td>
</tr>
<tr>
<td>● Very good performance compared to AppFabric.</td>
<td>● NCache Manager is sometimes very slow. Adding/deleting nodes sometimes takes a great deal of time.</td>
</tr>
<tr>
<td>● Good support through mail and phone.</td>
<td>● The documentation is not searchable, and it can be quite difficult to find what you are looking for, and sometimes scattered across different web pages.</td>
</tr>
<tr>
<td>● User-friendly with a GUI.</td>
<td>● A bit tricky to change the server code in the assemble file and to debug it. Needs to close the NCache services manually and restart the manager.</td>
</tr>
<tr>
<td>● Worked almost flawless out of the box if you only use it as a simple key/value store without back-end storage.</td>
<td>● If the IP address of the computer that has NCache installed changes, you have to reinstall NCache to make the program work again. We did not find any other solution on Alachisoft homepage.</td>
</tr>
<tr>
<td>● Faster synchronization with database.</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Advantages and disadvantages - NCache
### Advantages and Disadvantages - AppFabric

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>● It is Free of charge.</td>
<td>● Does not provide as many features as NCache (See appendix A for a feature comparison).</td>
</tr>
<tr>
<td>● Got smart features such as tagging and regions.</td>
<td>● No GUI requires PowerShell to configure and manage the cache cluster.</td>
</tr>
<tr>
<td>● Have all the coherency concepts available.</td>
<td>● No support for write-through.</td>
</tr>
<tr>
<td></td>
<td>● Performance is not as good as NCache.</td>
</tr>
<tr>
<td></td>
<td>● A bit tricky to get the cache cluster to work flawlessly. Sometimes the nodes did not add themselves to the cluster or had trouble finding the configuration file.</td>
</tr>
<tr>
<td></td>
<td>● The high-availability topology requires the Microsoft Server 2008 R2 (or later) operating system.</td>
</tr>
<tr>
<td></td>
<td>● The gets in AppFabric do not always show properly in perfmon or in AppFabric’s own statistics.</td>
</tr>
</tbody>
</table>

**Table 2: Advantages and disadvantages - AppFabric**

### 4.2.2 Distributed cache vs. database

As we can see from the results, distributed caching is superior to using a database when comparing throughput. There is a big difference between a database and AppFabric/NCache both in writes and reads. If we look at test 2, that use the partitioned topology, we can see how much the distributed cache products can handle in form of writes at their most optimal scenario for writes. As figure 24 shows, NCache has almost two times higher throughput than AppFabric when it comes to writes/s and about 1.5 times higher when dealing with reads/s. They both have much higher throughput than the database (figures 24 and 25). With two cache
nodes, AppFabric had almost 5 times more requests/s in reads and 28 times more in writes compared to the database. NCache had almost 8 times higher throughput in reads and 51 times higher in writes.

The performance of the distributed cache products when using the high-availability topology is also much better than the database could perform. Since we did not use any bulk-operations when testing the database, it can perform better than in the test. But the test was done to simulate the performance in the current MP system in Scania, which writes one object at a time. The numbers received from NCache when using write-behind (from cache cluster to database) was approximately 1400 writes/s, which is more than two times faster than when writing directly to the database (800 writes/s). AppFabric on the other hand did not perform nearly as well as NCache when using write-behind. AppFabric's numbers for write-behind was approximately 800 writes/s, which is almost identical to the number we got from the test where we only tested the database. We suspect that this is because NCache batches data to the database whilst AppFabric does not. Since NCache provides more efficient transactions to the database, it can “catch up” quicker with the cache cluster during less transaction-intensive time periods (for example during the night), as compared to AppFabric.

4.2.3 A developers workday

Through the interviews held with companies and especially Magnus Eriksson at Scania, we got a good picture of how experienced developers think it is to work with distributed cache. We also have gained personal experience of what it is like to work with distributed cache through the tests we conducted for this research. The conclusion drawn from this is that distributed caching is quite simple if it is used as only a key/value store. The complexity increases with the rate of components added to the environment. If for example a back store is added to the system, you need to create assembly files, which often can turn out to be quite problematic. From our own experience we can say that there are many small things that you need to fiddle with to get the products to work properly.

Through the interviews, we also learned how difficult and time consuming it can be to make calculations within the cache itself, which is recommended if you do much calculations on data and do not want to retrieve the data each time and then make calculations. An example of this is a scenario where GPS data needs to be aggregated in the cache to display e.g. the route travelled during the last three hours.

The learning curve for managing a cache product is not very steep. If you have some basic knowledge in programming you should get the cache to work as a key/value store quite easy. Adding a back store to the cache requires a bit more from the developer. As written above
there are many things that can mess up when adding a back store. For instance you need to know a bit about signing assembles and also basic computer knowledge to get the network connection between the nodes to work flawless. If the cache should be used for more complex tasks that require aggregation of data inside the cache itself, it requires a developer with good experience in cache development or distributed systems to avoid any conflicts that can occur, for example, time coherence, validation of items etc. The need for training could be solved by using Scania’s internal knowledge about distributed cache system. E.g. Magnus Eriksson at Infomate, who was involved in the development of OAS, could hold lectures about distributed cache.

The tools needed depend on what kind of product and programming language that is intended to use. In this thesis we used the .net framework and Visual Studio 2010 as IDE (integrated development environment). Both of the evaluated products were developed in this IDE. If you are going to use a back store you will also some form of database tools, e.g. SQL server 2008 or SQL Express or similar. The product themselves also needs to be installed and can be fetched from respective product’s homepage. NCache can be run on Windows XP, Windows Vista, Windows 7 and Windows Server 2003/2008 (64-bit recommended). The system requirements can be found in section 3.4.1. AppFabric 1.1 can be installed on the following operation systems Windows 7, Windows Server 2008 R2 and Windows Vista. Windows Server 2008 is needed if you want to use the topology high-availability. Other prerequisites and system requirements for AppFabric 1.1 can be found in the section 3.4.2.

4.3 Return on investment analysis

It is hard to estimate specific costs for these kinds of IT-investments. Since the numbers of cache servers depend on how the scenarios are implemented (e.g. status report interval etc.), we can only make some rough estimates. It is also very hard to put numbers on the returns that the system would provide. Because of this we decided to make a lightweight return-on-investment analysis, leaving room for further investigations if necessary. The analysis is calculated over a time period of 3 years.

4.3.1 Costs

Based on the needs of Scania’s system, we have estimated a requirement of at least two cache servers as they would be able to handle the projected load without any issues and provide data redundancy. This is based on the results from the FMP test, showing that both of the evaluated distributed caching products can handle at least 6936 transactions per second (Figure 22), which is significantly higher than the amount of messages/sec which is the estimated required throughput in 2015 (FMP volymprognos). A high-availability topology would work well as it can
easily handle the required throughput while providing failover protection, and thus should solve FMS’ scalability issues for years to come.

**Software / Licenses**
Since AppFabric is included in the windows operating system, the licensing costs differ significantly between the two products.

**Hardware**
Scania has 14 test environments for FMS, one of which is similar to the production environment. Scania must pay licensing costs to NCache for each of these environments. We also need to calculate how much the hardware will cost. Every environment needs a set of minimum two cache servers and two application servers. As cache clients, Scania will use already existing hardware, but as cache servers they need to acquire two servers to every environment. The two cache servers need to have at least dual core CPU and have a gigabit network interface card. They should also have at least 4 GB of RAM.

**Direct costs**

<table>
<thead>
<tr>
<th>Cost component</th>
<th>Description</th>
<th>Amount sek / month</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software / Licenses</td>
<td>• Developer license: FREE</td>
<td>0 Kr (AppFabric)</td>
</tr>
<tr>
<td></td>
<td>• Remote client: $1245 / CPU</td>
<td>19 600 Kr (NCache)</td>
</tr>
<tr>
<td></td>
<td>• Cache server: $2495 / CPU</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Maintenance and support:</td>
<td>+ 4900Kr support</td>
</tr>
<tr>
<td>Hardware</td>
<td>• 2x Cache servers</td>
<td>33600 Kr</td>
</tr>
<tr>
<td></td>
<td>• 2x Application servers (use existing)</td>
<td></td>
</tr>
<tr>
<td>Training</td>
<td>Personnel training: 2 employees. Course</td>
<td>1200 Kr</td>
</tr>
</tbody>
</table>

Table 3: Direct costs

**Indirect/Intangible costs**
Some of the intangible costs that could arise if Scania choose to implement a distributed cache solution:
- Cost of failure due to inappropriate system or faulty implementation.
- Incompatibility with other systems leading to unexpected costs of software amendments, tailoring, and maintenance.
- Cost of errors due to lack of experience in using new system (IT dept, users).
- Loss of investment in prior systems
(Andru & Botchkarev, 2011)
4.3.2 Financial returns

Direct returns

<table>
<thead>
<tr>
<th>Return component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost savings</td>
<td>Investments in database scale up (hardware and software). Reduced need for scale out activities or segmentation in FMP (e.g. through multiple sets of database’s at different geographic locations.)</td>
</tr>
<tr>
<td>Cost avoidance</td>
<td>Compared to other solutions that Scania has evaluated.</td>
</tr>
<tr>
<td>Increased revenue</td>
<td>Increase of sales as a result of a better system.</td>
</tr>
</tbody>
</table>

Table 4: Direct returns

Indirect/intangible returns

Some of the intangible returns that could arise if Scania chooses to implement a distributed cache solution:

- Enhanced system performance.
- Increased productivity and time savings.
- Increased intellectual capital.
- Higher customer satisfaction.
- Adds the possibility to build other services and solutions.

(Andru & Botchkarev, 2011)
5. Discussion and conclusion

This chapter contains a discussion, a summary and a critically review of the results, and how they relate to the research goals. It also contains a conclusion and suggestions for future work.

5.1 Purpose and research questions

The purpose of this master thesis was “to evaluate whether or not distributed caching is a suitable technique for Scania FMS?” and the goals of the research was:

- To answer the following research questions:
  - RQ1: How does distributed caching work in theory and in practice?
  - RQ2: Which parts of Scania’s FMS could use a distributed cache?
  - RQ3: Which products meet Scania’s requirements?
  - RQ4: Is a cost-free distributed caching product comparable to a commercial distributed caching product?

- To test the selected products.
- To evaluate the cost of implementation.

The following subsection will discuss the results for each research goal and put them into the right context.

5.1.1 RQ1: Evaluate/investigate how distributed caching works in theory and practice

The test results show that a distributed cache increases (somewhat) linearly when adding more nodes to the cluster. In our tests, we obtained a write speed of 31000 objects per second with NCache, which proves to be 51 times higher throughput than the tested database. NCache had also almost eight times higher throughput than the database in reads per second. AppFabric had a write speed of 17000 objects per second and had approximately 28 times higher throughput than the database in writes per second and five times higher in reads per second.

The newest feature and concept in distributed caching is elasticity, which stands for the ability to remove and add nodes ‘on the fly’ to the cache cluster (Qin et. al., 2011). In theory this is a really nice feature that could come in handy when scaling out, since a developer could add more force to the cluster without even having to shut it down and reload data. The evaluated products claim to provide this feature. However, our testing shows that in practice, this feature does not fully work. To evaluate the products’ elasticity, we both added and removed a node to the cluster while the application was running.
In the tests we tried to add and remove nodes in NCache manager, while the applications were running. Adding a node worked two times out of five without losing any data under heavy load, and five times out of five while in an idle cluster. In the successful attempts, the nodes started to distribute part of its data between to the newly added node (once it had been registered in the cluster) resulting in a fully operating cluster. This test was made with high-availability as topology.

We also tried to disconnect a node by unplugging the network cable; this caused the application to crash and throws an error, but the cluster continued running as expected. However, we did not track the exact objects lost during the process, but we did notice a lack of added objects in the final result.

5.1.2 RQ2: Which parts of Scania’s FMS could use a distributed cache?

In this thesis, we have evaluated three scenarios in which distributed caching can be an interesting technique to use. The three scenarios are:

- The current MP
- Future FMS
- Geo fencing

We argue that all scenarios/systems could make use of distributed caching, and based on the tests conducted, we argue that both products would significantly increase the throughput. It could also make the system much more scalable than a database could. However, there are also other factors to take into account when considering using a cache solution in a system. Before making a final decision, one should ask himself/herself the following question(s):

- Can we afford to lose data without affecting the system or its stakeholders?
  - Even though we use a high-availability topology that replicates data between the cache nodes and make them more reliable to cache-node failure, data located in the write-behind queue will be lost if a node goes down.
- Is it worth the implementation time/costs?
  - How hard is it to implement in the current system? Magnus Eriksson argued that it can be difficult to implement a distributed cache into a running system.
- Is NCache worth the extra cost over AppFabric, or does Appfabric provide sufficient features and performance for the future needs?

A distributed caching product should, based on the test results, be able to provide enough throughputs to handle the future data traffic in the MP system. One very important aspect is the time the cache takes to transfer objects to the database. This could become an issue,
depending on the load. To decrease the number of transactions to the database, one could even choose not to write every single object to the back store. For example, we could decide that we only need one out of five objects to be stored in the database, and keep the remaining objects in the cache to be consumed by the stakeholders. However, should all data be stored in database, we must ask ourselves the following question: *Does the high-availability of the cache cluster provide enough failover protection?* This is an issue because both NCache and AppFabric provide only one replica of the data. It is a very tough decision to make and is up to the architects of REI to decide whether or not it is sufficient.

The next generation of FMS scenario could also be a system that could make good use of a distributed cache solution. The cache cluster increases the systems performance a lot if there are many reads. Aggregation of data within the cache cluster would make it even more interesting. The question here is more how to implement the cache into the system. As we have written about before, it is quite complex to develop the cache to make operations within itself.

In our tests we have also seen that a scenario with many writes, as in the geofencing scenario, is suitable for a distributed cache solution. In a real world scenario of geofencing, there is probably no need to save every message of data persistent in a back store. An alternate solution could be to use a cache-aside as data synchronization, where the cache holds on to all the data and the client writes, for example, every tenth item to the back store. This solution requires less back store capacity and could therefore be cheaper. The cached data could then be evicted when needed, or expired after a certain period of time when running out of storage capacity in the cache cluster.

### 5.1.3 RQ3: Which product meet Scania’s requirements?

The first thing we needed to do was to make a selection of which products to test. This selection was made based on an evaluation of a few products. Based on the evaluation of the two evaluated products, we found that NCache was the faster distributed cache product both when it comes to reading and writing. It is also easier to use with the graphical interface that NCache Manager provides than AppFabric’s command-line solution with PowerShell. We also thought that NCache was easier to work with and did not have as many problems with the cache statistics, which AppFabric have (it does not always show the correct number of reads in the statistics). However AppFabric does get the work done. It may not be as fast as NCach and it does not have as many features, but it is faster than a database and both products are quite similar when it comes to API, programming and how they work in practice. One major difference is the cache structure between the products. AppFabric must have a configuration file saved persistent in a xml file or in a SQL database. All cache nodes must then point to this
configuration file. NCache uses peer-to-peer as cache structure and when you configure a cluster on a node it shares the information to the other nodes and in this way, each node in the cluster gets the same configuration. This has the advantage that if one node goes down, the other still has the configuration information. In AppFabric if the configuration file located in for example a SQL database goes down, the cluster cannot access the configuration information and will not be able to run. A list of positive and negative aspects of both products can be found in section 4.2.1.

5.1.3 RQ4: Is a cost-free distributed caching product comparable to a commercial distributed caching product?

No, the ‘free’ product is not as good as the commercial product that we have tested. NCache, which is the commercial product, outperformed the free AppFabric product in all tests. NCache is also the more updated software and has a user friendly, graphical interface. NCache also provides a lot more features than AppFabric does. However, AppFabric did perform well (much better than a database) in the tests and should still be considered a viable candidate in the future evaluation process of a distributed cache solution.

5.1.4 Testing the selected products.

As discussed in the result section 4.2.1, we see that NCache performs best of the two evaluated products in all scenarios. In the MP scenario NCache provides a significantly higher throughput compared to AppFabric, both with small and large objects. AppFabric displays a clear increase of throughput (for all object sizes) when another node is added to the cluster. The scalability for NCache is more evident for large objects than for small, but we still see an increased throughput of small objects as well.

In the ‘next generation FMS’ scenario, we again see that NCache seems to provide much higher throughput than AppFabric does. The difference is significantly larger when smaller objects are being transferred across the network. NCache shows a higher throughput, while AppFabric appears to have better scalability. The difference between the products shrinks when the size of the objects grows. NCache is approximately 2.5 times more efficient on one node than AppFabric with 256 bytes objects. On three nodes NCache performs approximately twice as fast. However, when the size of the objects is 1024 bytes, the gap between the products is much smaller. Since reads are generally much faster than write operations the products perform significantly better in this scenario than in the other two scenarios.

In the last scenario (geofencing), both products provide a lower throughput than in the other two scenarios. This is due to the high amount of writes to the cache cluster. AppFabric displays
an almost linear scalability, while NCache is more unpredictable, but overall can handle more requests per second than AppFabric can. The test shows that both products perform well in write heavy scenarios and that distributed cache is a possible solution to these kinds of scenarios.

The final step in the evaluation of the products in the tests was to see how they performed with the write-behind feature. We used write-behind as database synchronization in all tests and scenarios, and we found that NCache had an approximate write speed of 1400 writes per second to the database, no matter of the object size or test scenario. AppFabric was much slower in the given tests. AppFabric had an approximate write speed of 800 writes per second to the database. Another difference was that NCache started writing to the database directly after receiving the first object, while AppFabric waited approximately one minute before the process started. This means that NCache can handle a higher throughput before the cache cluster is full, since it can transfer data for persistent storage faster than AppFabric.

5.1.5 Evaluation of the cost of implementation

The return of investment analysis from section 4.3 gives us a basis to draw conclusions about what an implementation of these products might cost. It is however difficult to calculate an exact figure and to put a number on the returns that the implementation would provide.

The analysis shows that implementing NCache would cost approximately 24 500kr more per month than AppFabric in licensing costs. This should be compared to other viable solutions for handling the upcoming increase of transactions. It is also important to take the indirect benefits of implementing a distributed cache into considerations.

There would no longer be a need for implementing alternative solutions. However, one should also consider other (non-financial) returns such as higher customer satisfaction and the possibility to build other services and solutions based on the distributed cache system. NCache would probably provide more indirect returns as it is richer on features. So it is up to the people of REI to decide whether or not NCache is worth investing in, or if AppFabric is adequate for the future scenario.
5.2 Conclusion

Distributed cache proves to be a viable option when scaling out the system. The tests results show that a distributed cache significantly out performs a single database.

If the real world scenario requires fast writes to a persistent store NCache is the product to use as it is proven to be almost twice as fast as AppFabric. AppFabric write-behind feature was not that much faster than writing to the database only, which means that AppFabric might not be the best solution if everything written to the cache cluster must be stored persistently. If the real world scenario does not need everything that we cache to be stored persistent, both products should be good enough to use. Both products are capable of handling large amounts of requests/s, much higher than a single database.

It is also important to remember that data can be lost in a distributed cache solution if we use the write-behind feature. If a node goes down the data that is to be sent to the database will be lost and the data in the write-behind queue from that node will also be lost.

The return-on-investment analysis showed that an implementation of NCache would cost significantly more than AppFabric. Since we are in no position to make decisions based on this information, it is up to the architects/employees of FMS to decide whether or not the extra features provided by NCache are worth the additional cost over AppFabric, or if any of the products would be worth investing in.

The ultimate conclusion is that NCache outperformed AppFabric in all tests and NCache is the software that most often gets updated and has the most user friendly interface. However, AppFabric did perform okay in the tests and should still be considered as a good candidate in the future evaluation process of a distributed cache solution if the scenarios are not too dependent on fast database synchronization.

5.3 Recommendations for future work

Evaluation of distributed caching as concept and different distributed caching products can be a time consuming and frustrating process. There are many things in the chain of development that can (and will) go wrong, some of which are described in section 4.2.1.

It would be interesting to conduct some advanced testing on the cache to see how it reacts. A scenario of this could be aggregation of ‘route data’ that has to be calculated within the cache itself. Another test that could be done in a more advanced configuration is the scenario of geofencing. This can be done by letting the cache send event notifications to the application
when a cached item’s geo-values are outside the pre-specified geo-boundaries. Also, long-term testing should be performed to ensure the reliability and stability of the distributed cache system.

More in depth tests should be done to identify certain risks with using distributed caches. For example; security between the nodes and client and also tests of what data will get lost when nodes die or certain errors in the cluster arises. Further destructive testing should be done with the HA topology, to ensure that the cluster does not crash under any circumstances.

It could be interesting to investigate how well a cache cluster with better hardware performs. It would also be interesting to track a specific object from application through the cache and down to the database.

Finally, it is always interesting to see how more products perform under testing. A product that is widely used in many businesses and even here at Scania is Oracle's caching product. It would be interesting to see how it performs against the tested products.
References:

**Scientific papers**


Literature


Internet sources


# A. Appendix – Feature comparison

## Checklista för urvalskriterier

<table>
<thead>
<tr>
<th>Generellit</th>
<th>Vikt (1-5 - där 5 är viktigast)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kommersiell användning av produkten</td>
<td>4</td>
</tr>
<tr>
<td>Bra support</td>
<td>5</td>
</tr>
<tr>
<td>Framtida uppdatering av bibliotek och underhåll av cache</td>
<td>3</td>
</tr>
<tr>
<td>Erkända företag som använder produkten?</td>
<td>4</td>
</tr>
<tr>
<td>Kostnad</td>
<td>5</td>
</tr>
<tr>
<td>Licenser</td>
<td>3</td>
</tr>
<tr>
<td>Servrar - Antal maskiner, hårddvara, mjukvara,</td>
<td>4</td>
</tr>
<tr>
<td>Utvecklingskostnad - Kompetens, tid</td>
<td>4</td>
</tr>
<tr>
<td>Prestanda – skalbarhet/benchmarking/...</td>
<td>3</td>
</tr>
<tr>
<td>Väldokumenterad</td>
<td>4</td>
</tr>
<tr>
<td>Elastiskt</td>
<td>5</td>
</tr>
<tr>
<td>API</td>
<td>2</td>
</tr>
<tr>
<td>.NET-typer</td>
<td>4</td>
</tr>
<tr>
<td>Java-typer</td>
<td>3</td>
</tr>
<tr>
<td>Säkerhetsaspekter</td>
<td>3</td>
</tr>
<tr>
<td>Enkelhet</td>
<td>4</td>
</tr>
</tbody>
</table>

## Features

### Databassynkronisering

<table>
<thead>
<tr>
<th>Vikt (1-5 - där 5 är viktigast)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write-behind</td>
</tr>
<tr>
<td>Cache-aside</td>
</tr>
<tr>
<td>Read/Write through</td>
</tr>
<tr>
<td>Refresh-ahead</td>
</tr>
</tbody>
</table>

### Topologier

<table>
<thead>
<tr>
<th>Vikt (1-5 - där 5 är viktigast)</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Availability</td>
</tr>
<tr>
<td>Replicated topology</td>
</tr>
<tr>
<td>Partitioned</td>
</tr>
<tr>
<td>Near cache</td>
</tr>
</tbody>
</table>

### Expiration

<table>
<thead>
<tr>
<th>Vikt (1-5 - där 5 är viktigast)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Least Recently Used</td>
</tr>
<tr>
<td>Least Frequently Used</td>
</tr>
<tr>
<td>Prio-based</td>
</tr>
<tr>
<td>First in first out</td>
</tr>
</tbody>
</table>

### Data structures/types

<table>
<thead>
<tr>
<th>Vikt (1-5 - där 5 är viktigast)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML</td>
</tr>
<tr>
<td>ORL</td>
</tr>
<tr>
<td>Bytearray[]</td>
</tr>
</tbody>
</table>

### Named cache

<table>
<thead>
<tr>
<th>Vikt (1-5 - där 5 är viktigast)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Region store</td>
</tr>
<tr>
<td>Tagging</td>
</tr>
</tbody>
</table>

### Cache Query

<table>
<thead>
<tr>
<th>Vikt (1-5 - där 5 är viktigast)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomiska operationer</td>
</tr>
<tr>
<td>Multi Gets/Puts</td>
</tr>
<tr>
<td>Multi-threading</td>
</tr>
<tr>
<td>Bulk loading</td>
</tr>
<tr>
<td>Data dependencies</td>
</tr>
</tbody>
</table>

### Data storage

<table>
<thead>
<tr>
<th>Vikt (1-5 - där 5 är viktigast)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key/Value</td>
</tr>
<tr>
<td>Load balancing</td>
</tr>
<tr>
<td>SQL-Synkronisering</td>
</tr>
<tr>
<td>Data Compression</td>
</tr>
<tr>
<td>Customized serialization</td>
</tr>
</tbody>
</table>

### Concurrency

<table>
<thead>
<tr>
<th>Vikt (1-5 - där 5 är viktigast)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimistic</td>
</tr>
<tr>
<td>Pessimistic</td>
</tr>
</tbody>
</table>

### Coherency

<table>
<thead>
<tr>
<th>Vikt (1-5 - där 5 är viktigast)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failover handling</td>
</tr>
</tbody>
</table>

## Data dependencies

<table>
<thead>
<tr>
<th>Vikt (1-5 - där 5 är viktigast)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data storage</td>
</tr>
<tr>
<td>Appfabric</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>2,5</td>
</tr>
<tr>
<td>?</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Summa: 26,5 35 35 22,5 31 32 33

<table>
<thead>
<tr>
<th>Appfabric</th>
<th>Oracle Coherence</th>
<th>Ncache</th>
<th>Memcached</th>
<th>EHCache Enterprise</th>
<th>Gigaspaces</th>
<th>Scaleout</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x (proxy based)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Summa: 21 22 28 4 23 21 17
Beskrivningar

**Generellt**

Kommersiell användning av produkten
En bra och lättanterlig support av produkten. Tex, kan en svensk kundsupport vara att föredra i vissa fall.

Framtida uppdatering av bibliotek och underhåll av cache
Cache-produkten ska vara framtidssäker, dvs produkten bör underhållas upphovsskaparna med uppdateringar av bibliotek när detta krävs. Det bör även finnas ett förbättringsarbete av produkten med optimering och implementering av framtida features.

Erkända företag som använder produkten?
Att en cacheprodukt används av erkända företag kan vara en god fingervisning om att cacheprodukten i sig håller den standard som krävs för den ska vara ett bra alternativ för ett företags it-system.

**Kostnad**
Kostnaden för en cacheprodukt är beroende av flera faktorer.
- Licenskostnad per server/klient.
- Licenskostnad för eventuella uppdateringar/uppg radningar eller extra features.
- Licenskostnad för övrig mjukvara (win7, SQL Server 08 etc.)
- Kostnaden för de fysiska maskinerna (CPU, Minne etc.)
- Utvecklingskostnader i form av kompetens inom utvecklgruppen samt den tid det tar att utveckla systemet.

**Prestanda – skalbarhet/benchmarking/...**
Finns det gjorda underökningar som jämför prestanda mellan olika produkter?
Prestandatester av företagen själva.
Vad tycker företagens kunder?
Egna tester.

**Väldokumenterad**
Hur mycket finns dokumenterat om produkten. Det kan
bland annat innefatta dessa frågor. Finns alla features lättöverskådligt för en användare och är de bra förklarade? Finns det mycket dokumentation om hur man implementerar produkten och dess olika egenskaper i olika situationer? Samt hur mycket dokumentation finns det om eventuella problem och lösningar på dessa?

Elastisk
Klarar cache-produkten att starta upp nya noder utan att hela systemet behöver startas om / stängas ned?

API

Säkerhetsaspekter (specificera)
Säkerhet är en viktig aspekt i alla applikationer som hanterar data. Det är viktigt att endast applikationer och användare som är autentiserade kan använda datat.

Features
Databassynkronisering
Cache-aside: Cachen och databasen hanteras som två olika källor. Applikationen måste själv hantera var data ska sparas och läsas.
Write/read-through: Data skrivs till cachen och till databasen. Applikationen behandlar cachen som den enda datakällan och cachen hanterar synkronisering till databasen. Kan vara långsamt då applikationen behöver vänta på ack från cachen, som i sin tur måste få ack från databasen.
Write-behind: Modifierad data skickas asynkront ner till databasen med ett givet intervall. Detta avlastar databasen markant och ökar därmed prestandan.

Topologier
Replicated: Alla cache-noder innehåller samma data. Ökar availability och därmed läshastighet. Tappar dock i consistency då alla noder måste synkroniseras.
Partitioned: Varje cache-nod innehåller unik data. Detta ger
inget consistency-problem men om en nod går ner så försvinner datan helt.
High availability: Noderna innehåller unik data samt en kopia från en annan nod. Med denna teknik erhålls både hög tillgänglighet samt viss consistency då det blir mindre synkronisering.

Expiration

Eviction (LRU, FIFO, LIFO)
Teknik som bestämmer hur objekt raderas från cachen. Kan vara genom t.ex. LRU (least recently used) eller FIFO (first in first out)

Region store

Data structures
Beskriver vilka datastrukturer/typer cacheprodukten kan hantera.

Data grouping
Möjliggör gruppering av data i olika tabeller på olika cache-servrar. Man kan t.ex. ha ett lastbilsåkeri på en cache-server och ett annat åkeri på en annan.

.NET-typer
Om klienterna till stor del är .NET-baserade så kan det vara vettigt att ha en cache-lösning som stödjer .NET (native). Detta ökar prestandan vid t.ex. serialization.
Java-typer
Om en del av klienterna är java-baserade så kan det vara bra om cache-produkten stödjer java-typer för att minska antalet omvandlingar, och därmed öka prestandan.

Atomiska operationer
Tillåter klienter att läsa och skriva i en enda (atomsisk) operation. På detta sätt kan man undvika dirty reads/writes (dvs. reads/writes på objekt som inte blivit commitade).

Multi gets/puts

Bulk-operationer
En funktion som tillåter cachen att ladda stora data-chunks samtidigt. Detta används för att kunna populera cachen igen efter t.ex. en crash från en backup-cache eller ett ”senast fungerande stadie”.

Multi-threading support

Stöd för relationer/unioner
Gör det möjligt att skapa en tabell som hanterar relationer mellan objekt i cachen. Denna funktion bör endast användas om det är absolut nödvändigt då det ökar access-tider pga. extra hop samt låsningar.

Load balancing
Har cachen en inbyggd lastbalansering som sköter balanseringen av objektet i klustret. Dvs. så att objekten
fördeles jämt över noderna

Coherency
Coherency handlar om att alla cacher inom det distribuerade system innehåller samma version av ett visst data. Detta kan uppnås med hjälp av en rad olika tekniker:

Data Dependecies
En möjlighet att skapa beroenden mellan objekt i cachen. När ett objekt uppdateras eller tas bort så raderas även objektets relaterade cache-objekt för att bibehålla konsistent data.

Concurrency
Concurrency handlar om hur man uppdaterar objekt i cachen och blir snabbt ett problem när man jobbar med distribuerade system. De två vanligaste sätten att hantera detta är med den så kallade pessimistiska eller optimistiska modellen.

Optimistisk:
Pessimistisk:

SQL-Synkronisering
Vissa cache-produkter erbjuder möjlighet att synkronisera en del av cachen med en SQL-databas. Detta kan underlätta när cachen går ner eller när ett system inte kan läsa från cachen.

Enkelhet
Hur enkel är produkten att använda. Fungerar den "out of the box". Eller är det väldigt mycket pill innan det går att använda den?

Custom Serialization
Har cacheprodukten en egen inbyggd serializations-mekanism?
## Agenda of the tests

<table>
<thead>
<tr>
<th>Test</th>
<th>Description</th>
<th>Expected Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB1</td>
<td>Test query performance with large datasets</td>
<td>High availability</td>
</tr>
<tr>
<td>DB2</td>
<td>Test query performance with medium datasets</td>
<td>High availability</td>
</tr>
<tr>
<td>DB3</td>
<td>Test query performance with small datasets</td>
<td>High availability</td>
</tr>
<tr>
<td>DB4</td>
<td>Test query performance with mixed datasets</td>
<td>High availability</td>
</tr>
</tbody>
</table>

**Appendix**

<table>
<thead>
<tr>
<th>Partitioned ReadWrite</th>
<th>Test1</th>
<th>Test2</th>
<th>Test3</th>
<th>Test4</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Logistics**

- Test results will be reviewed under coming date.
- Test results are available in the database under coming date.
- Test results are available in the database under coming date.
- Test results are available in the database under coming date.

**Cache Size (Med/Small) Sensational Edition Database Sync**

- Test results are available in the database under coming date.
- Test results are available in the database under coming date.
- Test results are available in the database under coming date.
- Test results are available in the database under coming date.

**Topology**

- Test results are available in the database under coming date.
- Test results are available in the database under coming date.
- Test results are available in the database under coming date.
- Test results are available in the database under coming date.

**Network**

- Test results are available in the database under coming date.
- Test results are available in the database under coming date.
- Test results are available in the database under coming date.
- Test results are available in the database under coming date.
# C. Appendix – Hardware setups

The hardware setup for the FMP scenarios test with NCache:

<table>
<thead>
<tr>
<th>App Clients</th>
<th>2x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor:</td>
<td>Intel Xeon W3530 2 2,80GHz 2,80GHz</td>
</tr>
<tr>
<td>Memory:</td>
<td>6GB RAM</td>
</tr>
<tr>
<td>Hard Drive:</td>
<td>ST3320413AS</td>
</tr>
<tr>
<td>Network Interface:</td>
<td>Broadcom NetXtreme 57xx Gigabit Controller</td>
</tr>
<tr>
<td>Operating System:</td>
<td>Win 7 Enterprise SP1 64bit</td>
</tr>
<tr>
<td>Software:</td>
<td>Alachisoft NCache Enterprise Edition x64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache Servers</th>
<th>1x</th>
<th>2x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor:</td>
<td>Intel Core 2 Duo CPU E8400 @ 3.00GHz 2,99GHz</td>
<td>Intel Core 2 Duo CPU E8400 @ 3.00GHz 2,99GHz</td>
</tr>
<tr>
<td>Memory:</td>
<td>4GB RAM 3,21 usable</td>
<td>4GB RAM</td>
</tr>
<tr>
<td>Hard Drive:</td>
<td>WDC WD1600AAJS-75M0A0</td>
<td>wdc wd2500aakx-753ca1</td>
</tr>
<tr>
<td>Network Interface:</td>
<td>Intel(R) 82567LM-3 Gigabit Network Connection</td>
<td>Intel(R) 82567LM-3 Gigabit Network Connection</td>
</tr>
<tr>
<td>Operating System:</td>
<td>Win XP Pro SP3 32bit</td>
<td>Win 7 Enterprise 64bit SP1</td>
</tr>
<tr>
<td>Software:</td>
<td>Alachisoft NCache Enterprise Edition x86</td>
<td>Alachisoft NCache Enterprise Edition x64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Database</th>
<th>1x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor:</td>
<td>Intel Xeon W3530 2 2,80GHz 2,80GHz</td>
</tr>
<tr>
<td>Memory:</td>
<td>6GB RAM</td>
</tr>
<tr>
<td>Hard Drive:</td>
<td>ST3320413AS</td>
</tr>
<tr>
<td>Network Interface:</td>
<td>Broadcom NetXtreme 57xx Gigabit Controller</td>
</tr>
<tr>
<td>Operating System:</td>
<td>Win 7 Enterprise SP1 64bit</td>
</tr>
<tr>
<td>Software:</td>
<td>Windows SQL Server 2008 R2</td>
</tr>
</tbody>
</table>
The hardware setup for the FMP scenarios test with AppFabric:

<table>
<thead>
<tr>
<th></th>
<th>2x</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>App Clients</strong></td>
<td></td>
</tr>
<tr>
<td>Processor:</td>
<td>Intel Xeon W3530 2 2.80GHz 2.80GHz</td>
</tr>
<tr>
<td>Memory:</td>
<td>6GB RAM</td>
</tr>
<tr>
<td>Hard Drive:</td>
<td>ST3320413AS</td>
</tr>
<tr>
<td>Network Interface:</td>
<td>Broadcom NetXtreme 57xx Gigabit Controller</td>
</tr>
<tr>
<td>Operating System:</td>
<td>Win 7 Enterprise SP1 64bit</td>
</tr>
<tr>
<td><strong>Cache Servers</strong></td>
<td></td>
</tr>
<tr>
<td>Processor:</td>
<td>Intel Xeon CPU X5650 @ 2.67 GHz 2.66GHz</td>
</tr>
<tr>
<td>Memory:</td>
<td>4GB RAM</td>
</tr>
<tr>
<td>Hard Drive:</td>
<td>WDC WD1600AAJS-75M0A0</td>
</tr>
<tr>
<td>Network Interface:</td>
<td>Broadcom NetXtreme 57xx Gigabit Controller</td>
</tr>
<tr>
<td>Operating System:</td>
<td>Windows Server R2 Enterprise 64bit with SP1</td>
</tr>
<tr>
<td>Software:</td>
<td>Microsoft AppFabric 1.1 for Windows Server x64</td>
</tr>
<tr>
<td><strong>Database</strong></td>
<td></td>
</tr>
<tr>
<td>Processor:</td>
<td>Intel Xeon W3530 2 2.80GHz 2.80GHz</td>
</tr>
<tr>
<td>Memory:</td>
<td>6GB RAM</td>
</tr>
<tr>
<td>Hard Drive:</td>
<td>ST3320413AS</td>
</tr>
<tr>
<td>Network Interface:</td>
<td>Broadcom NetXtreme 57xx Gigabit Controller</td>
</tr>
<tr>
<td>Operating System:</td>
<td>Win 7 Enterprise SP1 64bit</td>
</tr>
<tr>
<td>Software:</td>
<td>Windows SQL Server 2008 R2</td>
</tr>
</tbody>
</table>

The hardware setup for the products performance test with partitioned topology:

<table>
<thead>
<tr>
<th></th>
<th>2x</th>
<th>2x</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>App Clients</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor:</td>
<td>Intel Xeon W3530 2 2.80GHz 2.80GHz</td>
<td>Intel Core 2 Duo CPU E8400 @ 3.00GHz 2.99GHz</td>
</tr>
<tr>
<td>Memory:</td>
<td>6GB RAM</td>
<td>4GB RAM</td>
</tr>
<tr>
<td>Hard Drive:</td>
<td>ST3320413AS</td>
<td>wdc wd2500aakx-753ca1</td>
</tr>
<tr>
<td>Network Interface:</td>
<td>Broadcom NetXtreme 57xx Gigabit Controller</td>
<td>Intel(R) 82567LM-3 Gigabit Network Connection</td>
</tr>
<tr>
<td>Operating System:</td>
<td>Win 7 Enterprise SP1 64bit</td>
<td></td>
</tr>
<tr>
<td>Software:</td>
<td>Alachisoft NCache Enterprise Edition x64</td>
<td></td>
</tr>
<tr>
<td>Operating System:</td>
<td>Win 7 Enterprise 64bit SP1</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------</td>
<td></td>
</tr>
<tr>
<td>Software:</td>
<td>Alachisoft NCache Enterprise Edition x64</td>
<td></td>
</tr>
</tbody>
</table>

The hardware setup for the database performance test:

<table>
<thead>
<tr>
<th><strong>App Clients</strong></th>
<th><strong>2x</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor:</td>
<td>Intel Xeon W3530 2 2,80GHz 2,80GHz</td>
</tr>
<tr>
<td>Memory:</td>
<td>6GB RAM</td>
</tr>
<tr>
<td>Hard Drive:</td>
<td>ST3320413AS</td>
</tr>
<tr>
<td>Network Interface:</td>
<td>Broadcom NetXtreme 57xx Gigabit Controller</td>
</tr>
<tr>
<td>Operating System:</td>
<td>Win 7 Enterprise SP1 64bit</td>
</tr>
<tr>
<td>Software:</td>
<td>Alachisoft NCache Enterprise Edition x64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Database</strong></th>
<th><strong>1x</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor:</td>
<td>Intel Xeon W3530 2 2,80GHz 2,80GHz</td>
</tr>
<tr>
<td>Memory:</td>
<td>6GB RAM</td>
</tr>
<tr>
<td>Hard Drive:</td>
<td>ST3320413AS</td>
</tr>
<tr>
<td>Network Interface:</td>
<td>Broadcom NetXtreme 57xx Gigabit Controller</td>
</tr>
<tr>
<td>Operating System:</td>
<td>Win 7 Enterprise SP1 64bit</td>
</tr>
<tr>
<td>Software:</td>
<td>Windows SQL Server 2008 R2</td>
</tr>
</tbody>
</table>