Ignite Caching for ActiveNet Servlet Design

# 1 Problem

Following section is taken from other document [*Servlet Caching Redesign*](https://activexperience.active.local/gpmo/projects/communities/ActiveNet_New/Shared%20Documents/Development/Architecture/Servlet%20Caching%20Redesign.docx)

## 1.1 Overview

From the very beginning, ActiveNet was designed with a very aggressive homegrown caching system for database data:

* **Full table caches of “administrative” tables:** Roughly, the types of data ActiveNet works with can be considered either administrative or transactional. Transactional tables including the population system (customers, companies, etc), and all associated financial and inventory data; these are not basically cached at all. Administrative tables configure the operation of the system, and are comparatively small (although a large org might have tens of thousands of activities or facilities). At startup, the class associated with an administrative table reads the entire table, creating objects for each record, in some type of cache structure (ArrayList, HashMap, etc.), so the full table is available. There are some variations of this model to reduce the amount of data cached for some tables, but this should be considered the baseline design. The code relies extensively on the fact it can enumerate through this data in memory for many purposes, such as displaying select boxes, lists of records for editing, and some in-memory searching. There are approximately 240 such classes.
* **Cache synchronization between servers:** For redundancy, each org has at least two servlet instances on different servers. In order to keep the caches on these servers in sync as administrative tables are edited, ActiveNet has a simple homegrown cache coherence system. There is a separate application called CacheControl, which is a very simple messaging application. On each insert, update or delete into an admin table, the base classes of the administrative table system send a message with a serialized Java object to all other servers for that org registered with CacheControl; upon receiving those messages, they update their copy of the cache. A simple softlock system (the System\_Locks table) is used to prevent concurrent updates of admin objects.

## 1.2 Problems with current design

* **Memory-bound architecture**: This cache design requires significant memory. Memory has been the limiting factor in how many servlets could run in the current single-tenant (ST) architecture; ActiveNet app servers are memory-bound, not CPU-bound. The multi-tenant (MT) servlet now under development should allow significantly more sites to run on a server, but it will still be necessary to assign specific sites to specific servers, and the system is still expected to have plenty of CPU headroom. Reducing the memory footprint of caching will allow us to use servers more efficiently. The ideal would be if any server can take requests from any site, so the entire set of servers can serve as a “rush pool” for peak loads
* **Duplication of cache data:** Each server running an org has the entire cache present in its memory. The more servers assigned to a site, the more memory impact
* **Startup time:** When a servlet starts up, before it can process any request, it loads its full cache. For large orgs, this can take a few minutes or more. This causes additional downtime if servers need to be restarted. It also significantly effects deployment time, because thousands of servlets are loading their caches during this time.
* **Cache coherence reliability:** The CacheControl service is a simple home-grown design, and has had reliability problems from time to time. This is one reason each ANet servlet has a scheduled daily reload of the cache.

# 2 Ignite Solution

## 2.1 Ignite in a Glance

Apache Ignitetm In-Memory Data Fabric is a high-performance, integrated and distributed in-memory platform for computing and transacting on large-scale data sets in real-time, orders of magnitude faster than possible with traditional disk-based or flash-based technologies.

There are a lot of features provided by Ignite. You can view Ignite as a collection of independent, well integrated, in-memory components geared to improve performance and scalability of your application. Some of these components include:

* Advanced Clustering
* Data Grid (JCache)
* Streaming & CEP
* Compute Grid
* Service Grid
* Ignite File System
* Distributed Data Structures
* Distributed Messaging
* Distributed Events
* Hadoop Accelerator
* Spark Shared RDDs

## 2.2 Why Ignite?

Ignite is the recommended key-value caching system by EA team, the reasons include:

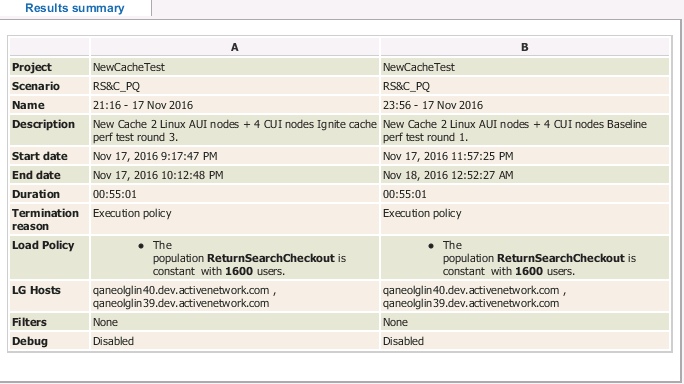
* It can either run as a stand-alone application or be embedded in our applications.
* It offers off-heap memory store mode to minimize pauses for Java GC.
* It’s fully compliant with JCache specification.
* It’s easy to be scaled out.
* It supports Partitioned, Replicated and Local store modes.

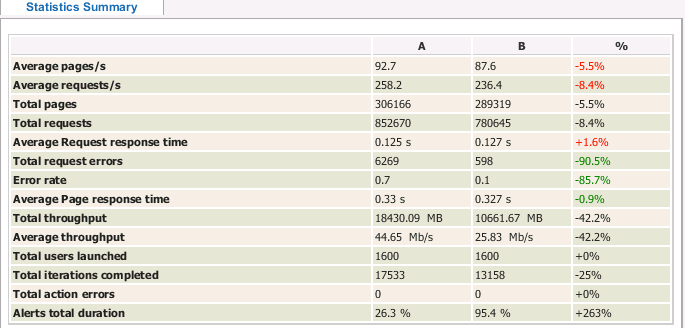
Comparing with Redis, the major advantages of Ignite include:

* Ignite is a in-memory data grid, that means we can operate java object with it directly, no needs for considering serialization/deserialization in the application side.
* Ignite provides the ability of “scan queries”, which allows for querying cache in distributed form based on some user defined predicate. Particularly for ActiveNet, The data like activities cached in Ignite don’t need to be downloaded to local memory then execute the query locally like what Redis solution does; the query will be executed on the remote cache server directly and only the search result will be returned accordingly.

## 2.3 Performance Test

A POC project has been done to test the performance of Ignite. In this project, cache for activities is moved to a remote Ignite cache server and all the get/put/iterate operations were executed remotely. The test shows a competitive result between using Ignite (column A) and not using Ignite (column B):





For the particular page /Activity\_search, which will run a iterating query in the remote cache node, shows the result as below:





The test with Ignite is even 3.9% faster than the test without Ignite.

## 2.4 Target Architecture

### 2.4.1 Description

Ignite caching will use a distributed architecture to replace the homegrown “local memory + cache control” mode.

Cache Pool 1

Cache Server

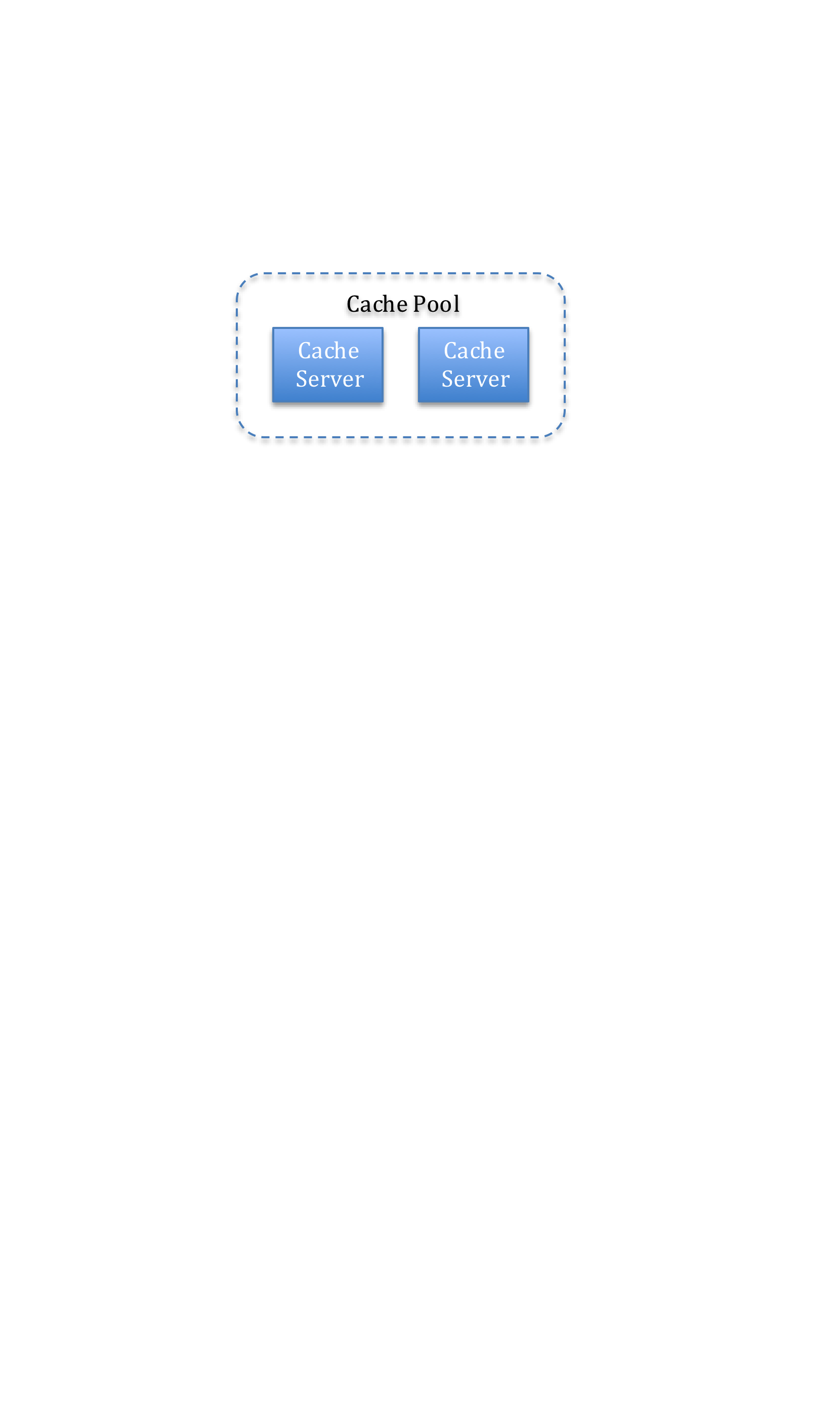
Cache Server

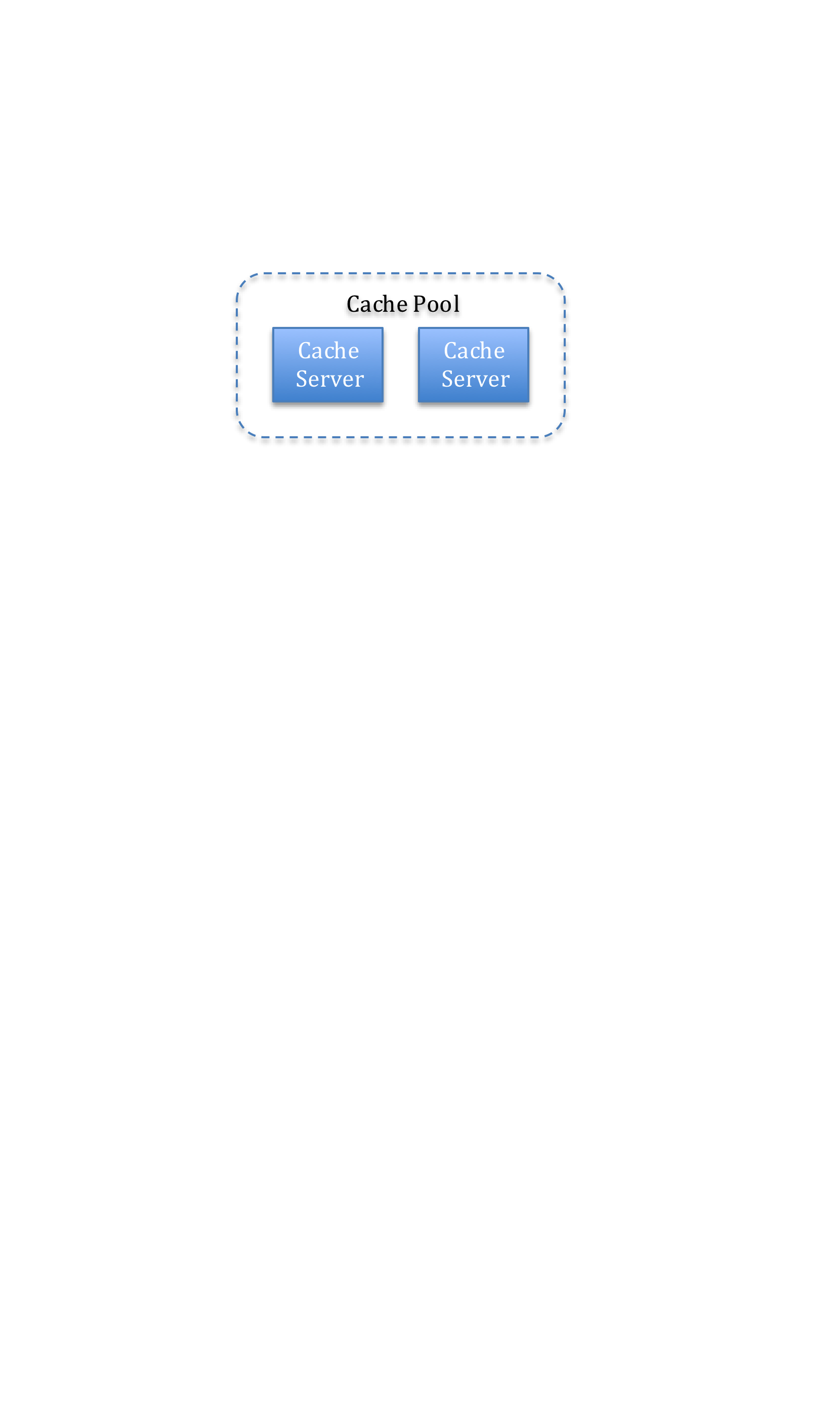
Cache Pool 2

Cache Server

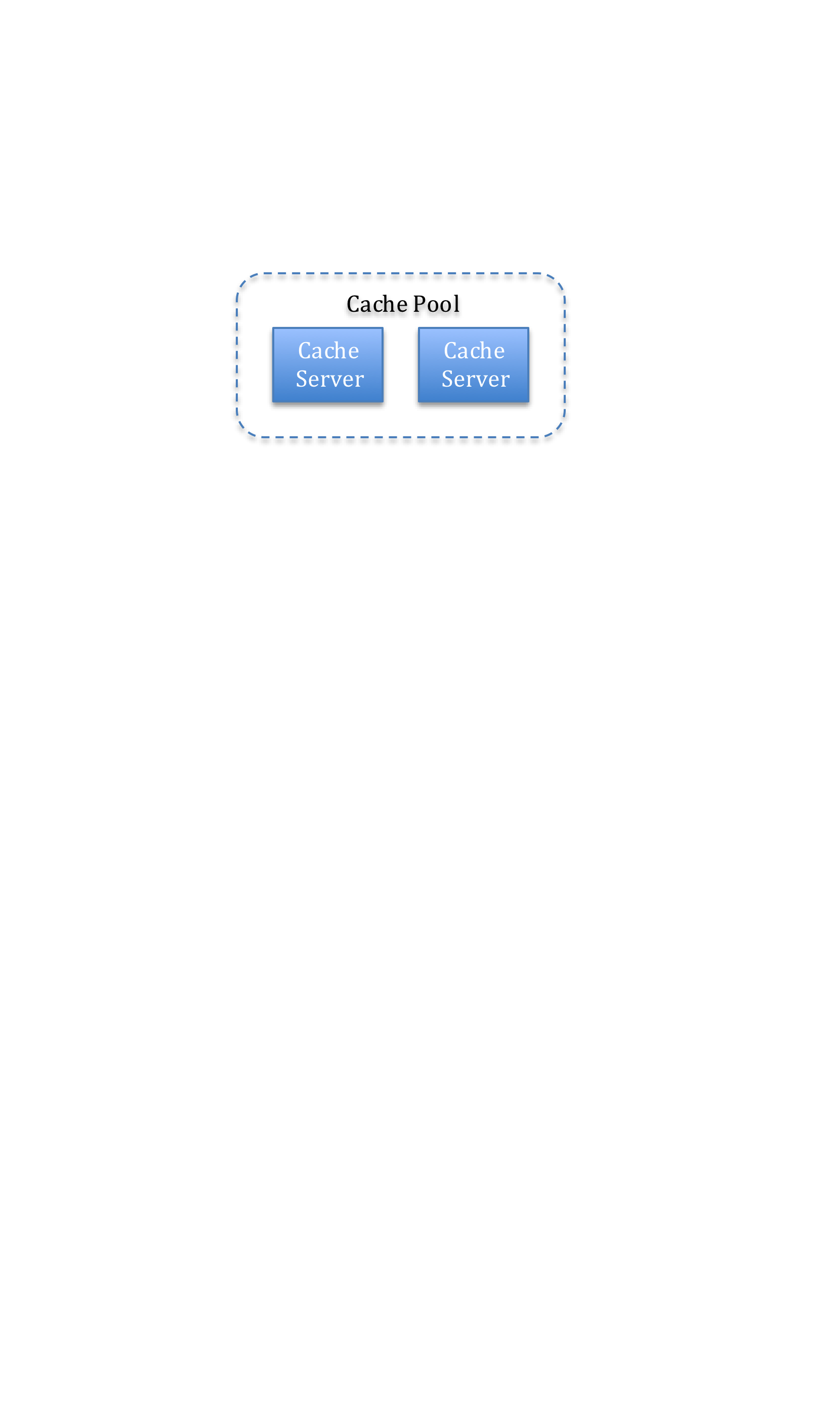
Cache Server

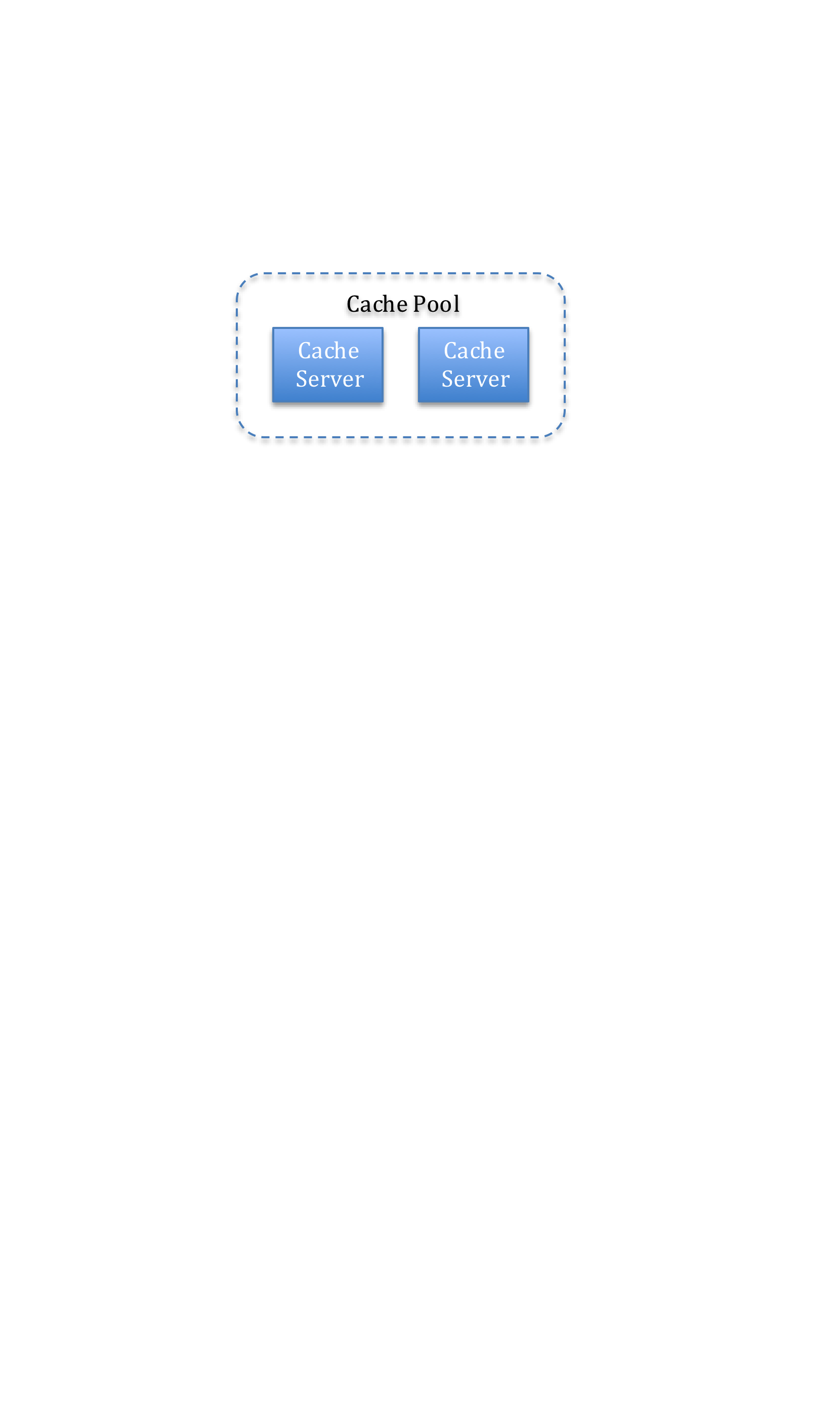
Org Site 1

Web Server

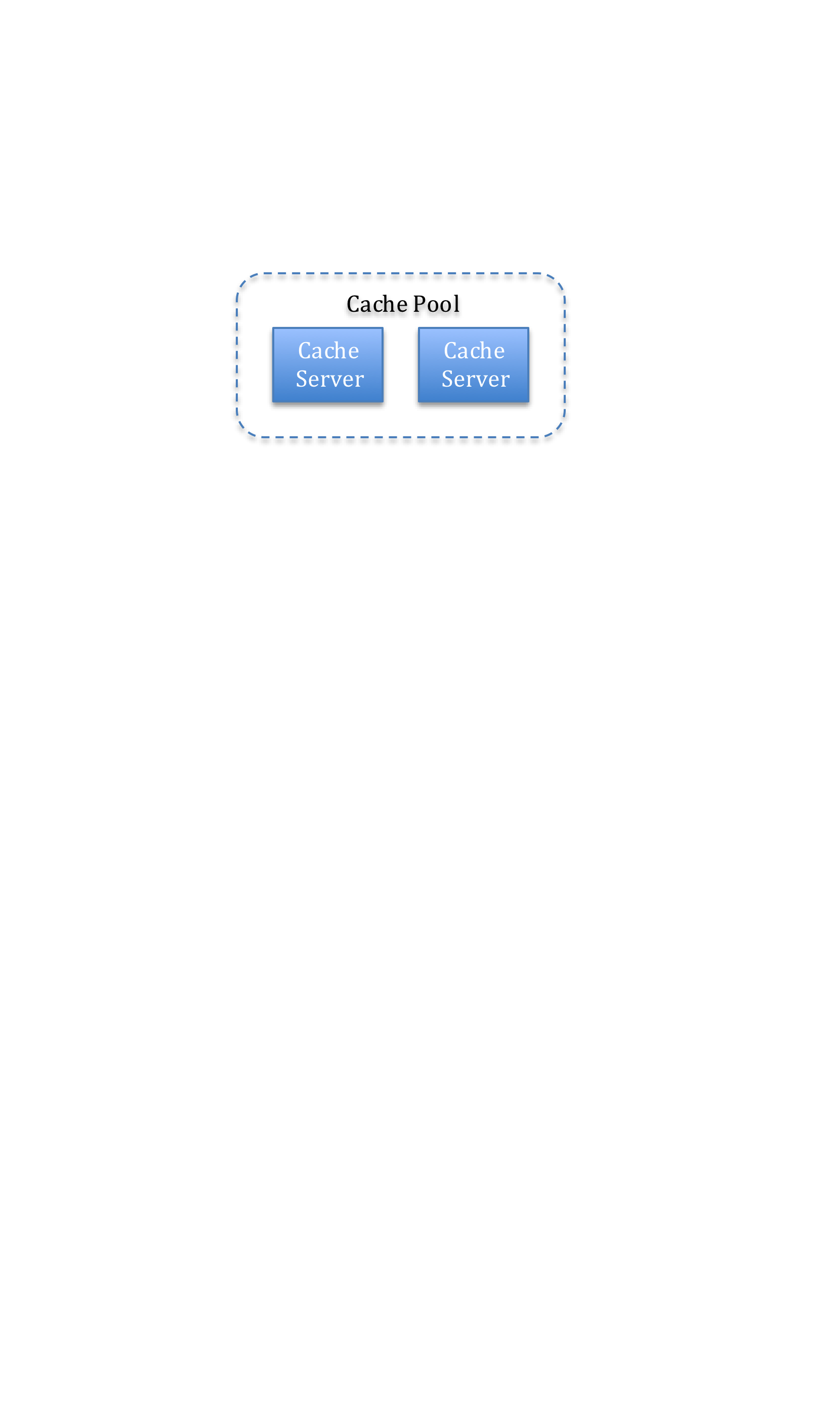
Web Server

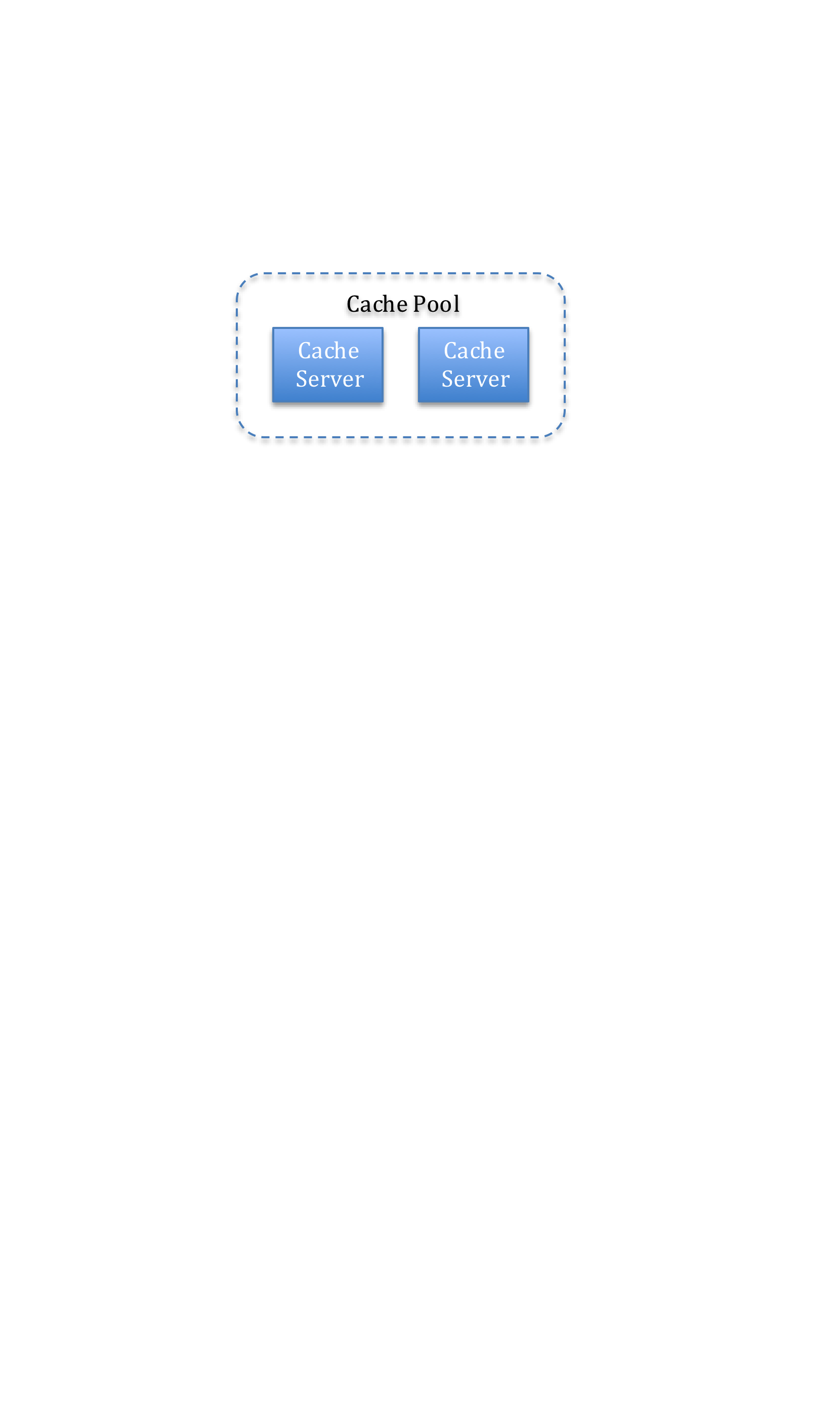
Org Site 3

Web Server

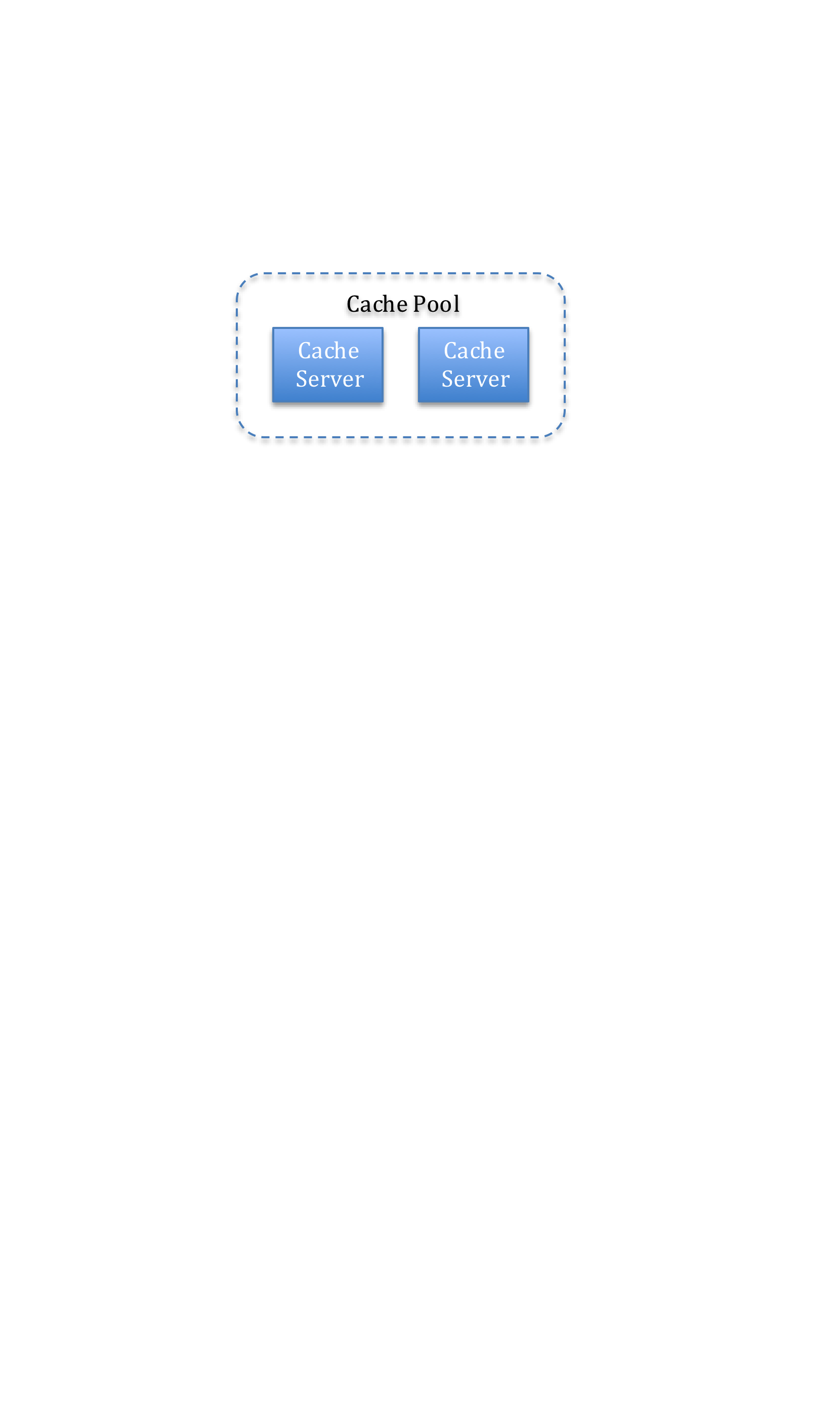
Web Server

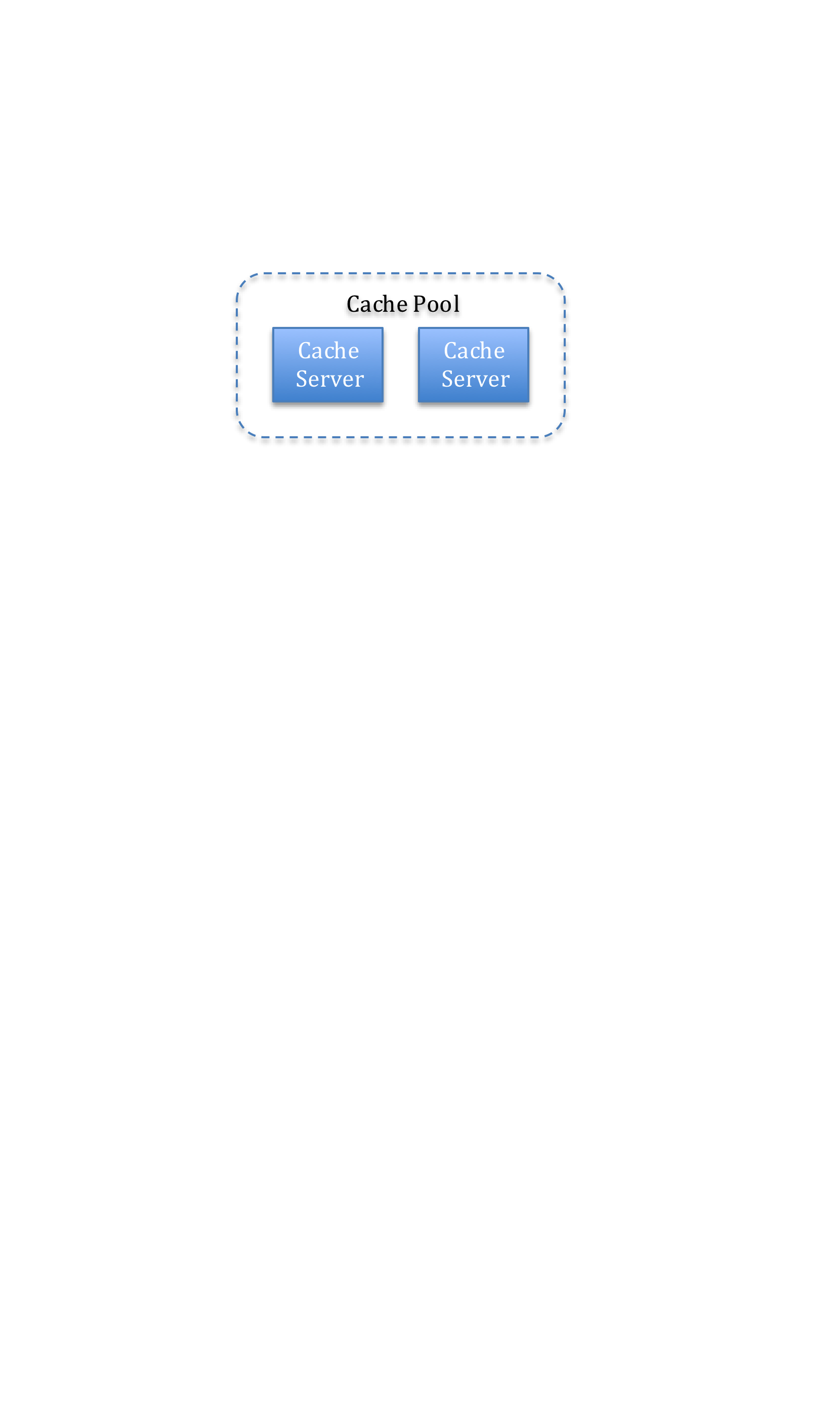
Org Site 2

Web Server

Web Server

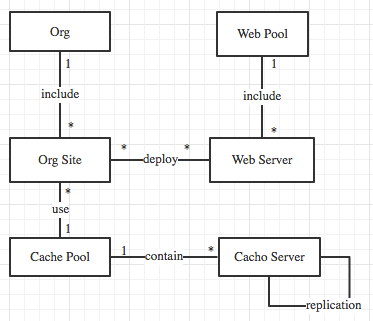
Org Site 4

Web Server

Web Server

et

There will be two new deployment elements Cache Pool and Cache Server added into the blueprint:



There will be one cache pool assigned to every org site. Because in MT mode, one org site could be deployed to multiple web servers, also one web server could include multiple org site, so the cache pool should be assigned to org site, not web server.

For a giving cache pool, there will be more than one cache server included; all the cached data is replicated to every cache server in this cache pool. There could be numbers of cache pool in the whole ActiveNet deployment; each will be used by a certain collection of org site.

### 2.4.2 REPLICATED Mode

There are three cache modes provided by Ignite, PARTITIONED, REPLICATED and LOCAL. According to the current cache usage of ActiveNet, REPLICATED mode will be adopted as:

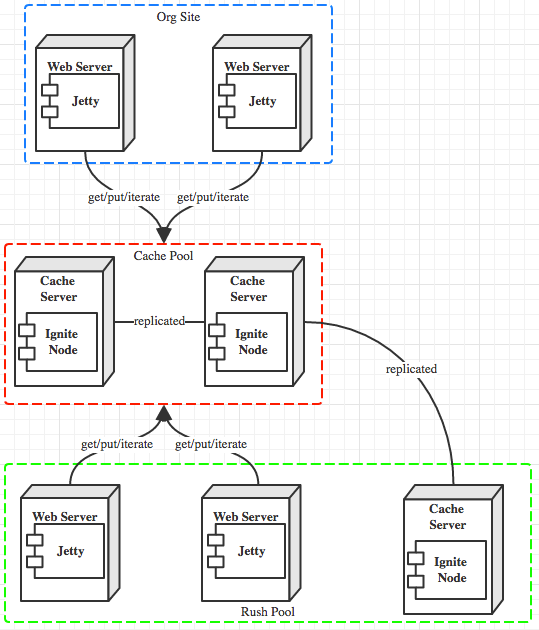
* The code in ActiveNet iterates though the cache, like populating lists, displaying lists of records, Activity search and others. In REPLICATED mode, all the cached data is hosted in one server, which will maximize the iteration performance as no communication between cache servers needed.
* The data in cache server is usually administrative data, like activities, facilities. The quantity of this kind of data is comparable small, tens of thousands in maximum. Therefore there is no scenario that the size of cache for a particular site is too large to fit in a single server.
* There are far more looking up than updating to the administrative data, so the impact on performance of propagating the data updates to all the other servers in a pool should be acceptable.

### 2.4.3 Load Balancing

Ignite has build-in capability to control load on all servers and make sure that every server in the pool is equally loaded. There are strategies like Round-Robin and Random and Weighted supported.

### 2.4.4 Rush Hour

Cache Server can be added into or removed from pool dynamically. During rush hours, web servers and cache servers could be added by a certain ratio to handle the extra pressure. For example, 2 web servers and 1 cache server could be added to handle the registration for a particular event; after the event is over, those severs could be removed without effecting the work of other servers.



### 2.4.5 Messaging

For now CacheControl in ActiveNet is not only playing the role of keeping the caches on servers in sync, but also working as message queue so servers could publish events to others in the same org site. Ignite also provides a cluster-wide messaging mechanism via which servers could subscript and publish topics they are interested. Using Ignite to replace the home made messaging queue allows removing CacheControl from ActiveNet architecture permanently.

### 2.4.6 Clustering support

Ignite is not only a distributed caching system, it also supports computing and transacting on large-scale data sets in real-time. It supports features like clustering, data grid, streaming, distributed computing; it supports the integration with other frameworks like Hadoop and Spark. Adopting Ignite gives ActiveNet a solid foundation for possible requirement like cluster computing in the future.

# 3 The Process of Migrating to Ignite

Adopting Ignite is supposed to last for a relative long while, which may span several sprints, involve effort from teams like development, SCM and QA. The major milestones should include:

* Set up the infrastructure of Ignite cluster;
* Move cache data to Ignite;
* Move messaging to Ignite;
* Move iterating look-ups to Ignite.

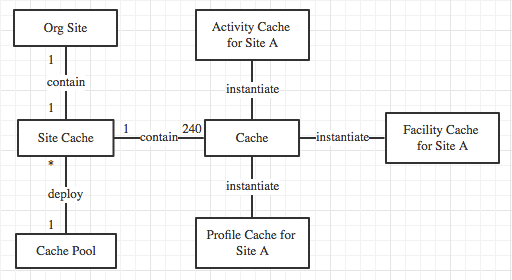
The whole process will be separated to phases and steps; the outcome for each step could be deployed to product environment for testing, they are also allowed to be disabled or reverted in case of significant error discovered.

The process allows different cache modes co-exist in the same deployment temporarily. For example, the activities data is saved in remote Ignite cache server while other data belongs to the same Org Site is still in local memory. This approach will allow the migration to Ignite moving forward in a gradual manner.

## 3.1 Cache Design

To achieve the above goal, the following design is made.

### 3.1.1 Cache Definition



#### Cache

Cache refers to a data collection of a particular type for a particular Org Site, like activities for Site A, facilities for Site B.

Caches are named according to the data type and the org site they belong to, like “LSTGChicagoparkdistrict \_Activity”, “LSTGgepark \_Facility”.

### Site Cache

Site Cache refers to the collection of caches belong to one particular org site; there are about 240 caches for one site cache.

From the perspective of deployment, all the caches belong to one site cache will be deployed on one cache pool and replicated to all the cache servers belong to this pool. During an iterating look up, all the caches for one org site should be available in the cache server as caches may refer to each other.

Because cache servers in one cache pool will replicate each other, there could be more than one copy for one site cache in the whole ActiveNet deployment.

### 3.1.2 Cache Mode

There will be three cache modes supported by the caching system: NATIVE, LOCAL and REMOTE.

Cache Mode is an attribute of cache, which means even in the same site cache, caches are allowed to have its own mode.

#### NATIVE

NATIVE mode refers to the current homegrown cache mode of ActiveNet, saying that all the data is saved in local memory; CacheControl is used to sync the changes for a cache among servers.

#### LOCAL

LOCAL mode refers to the mode that the data for a cache is taken care by Ignite, but the data is saved locally rather than on the remote node. In this mode, Ignite will be responsible for syncing changes for one particular cache among servers; messaging using Ignite could also be achieved in the mode. If a cache is in LOCAL mode, there is no need for CacheControl for this cache at all.

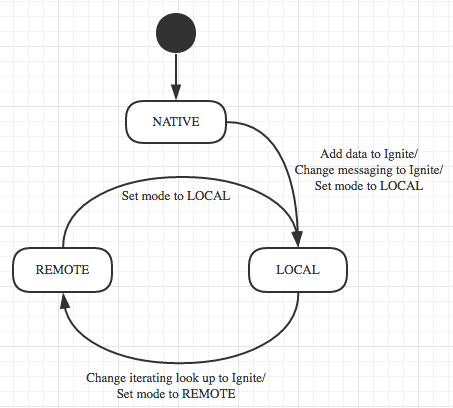
#### REMOTE

REMOTE mode refers to the mode that the data for a cache is saved in a remote cache pool; all the cache operation get/put/iterate will be happened in a certain cache server belonged to this pool. In this mode, memory usage will be maximized but extra time and CPU will be spent on data serialization/transfer/deserialization.

## 3.2 Migration Process Design

Generally speaking, the process of adopting Ignite is the process of transforming every cache of every site cache from NATIVE mode to LOCAL mode then to REMOTE mode; the process will involve both code change and deployment change.

### 3.2.1 Mode Transform Definition

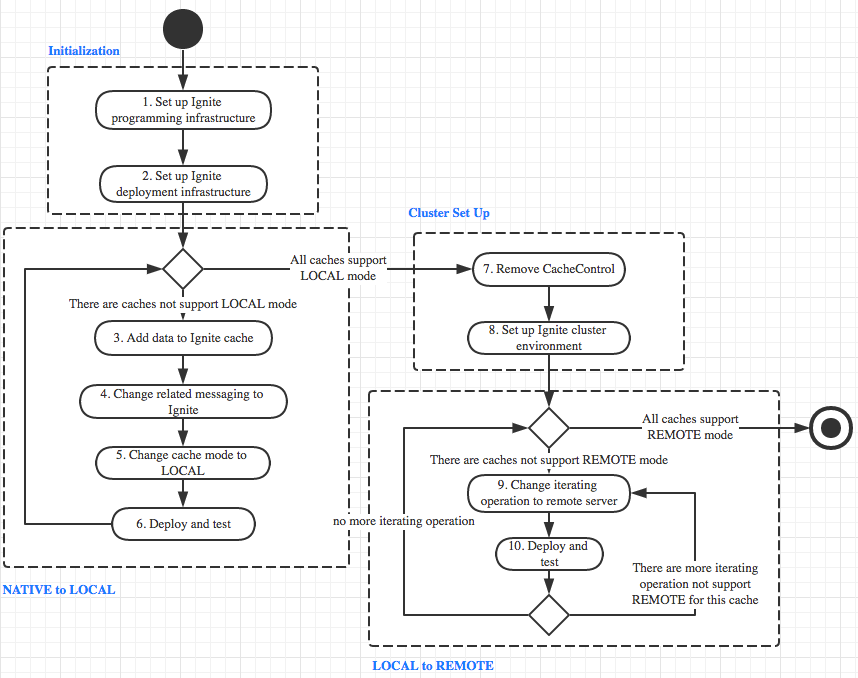


By the end of this process, there will be no need for CacheControl so normally NATIVE should not exist anymore, but LOCAL and REMOTE may co-exist. As there are no multiple web servers for trainer sites, trainer sites will stay with LOCAL mode.

For a given site cache, it is allowed that caches in this site cache are in different mode, but in some cases, a particular set of caches need to stay in same mode. For example, if cache for activities is moved to REMOTE mode, other caches like cache for CenterBO needs to be moved to REMOTE mode as well, because the search for activities needs to refer to center cache.

Another specific case is that in a given time, a cache could be in LOCAL mode and REMOTE mode in the same time. This situation may happen during the process of moving a cache from LOCAL mode to REMOTE mode. Taking Activity as instance, there are more than one hundred of iterating operations of it, so it will take a pretty long time to moving all the iterating operation to remote server. To persist the principle of migration in a gradual manner, the cache for activity could be saved in both local and remote, which allows some of iterating operations are run remotely and others are run locally, giving us the opportunities of testing the solution in the product environment.

### 3.2.2 Migration Process Definition



There will be 4 phases in the migration process: Initialization, NATIVE to LOCAL, Cluster set up and LOCAL to REMOTE.

#### Initialization

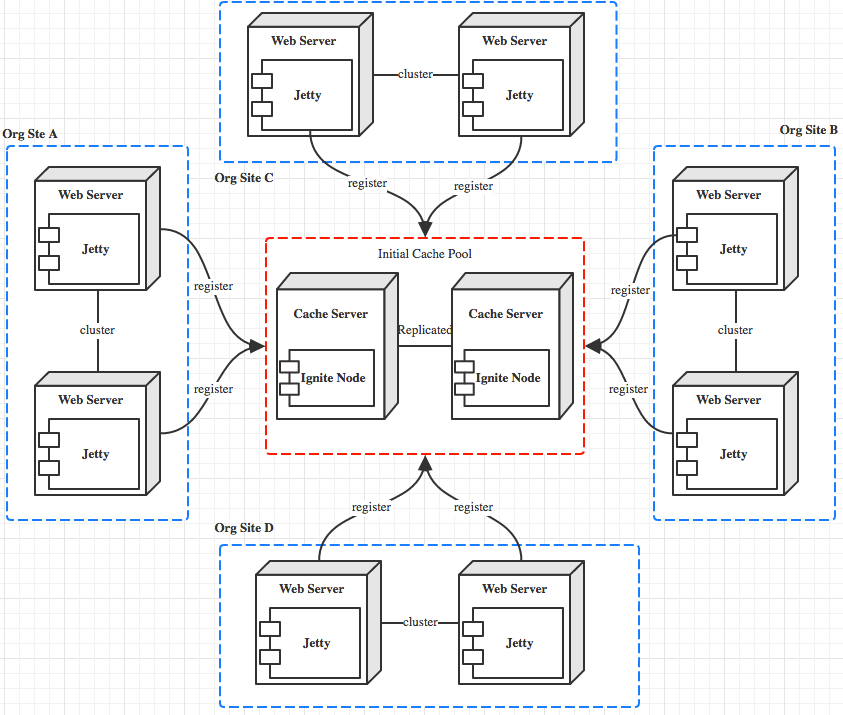
This phase includes step 1 and 2.

##### Step 1: Set up Ignite programming infrastructure

In this step, a programming infrastructure for Ignite should be set up. The infrastructure should make it easy to declare data collection as Ignite cache, maybe via java annotation.

##### Step 2: Set up Ignite deployment infrastructure

In this step, an initial cache pool will be set up, which allows every cache register itself on it. This cache pool will help to set up the cache cluster for each cache deployed on different web server. In this step, both SCM and development team will be involved.



#### NATIVE to LOCAL

This phase includes step 3, 4, 5, and 6.

##### Step 3: Add data to Ignite cache

In this step, developer will use the programming infrastructure created in step 1 to move cached data to Ignite. Because Ignite also saves the data in local memory, there should be no big change for the usage of cache, including get/put/iterate operations.

Once the cache has been moved to Ignite, cache notification needs to be changed to exclude this cache from syncing via CacheControl, as Ignite will take care of it.

##### Step 4: Change related messaging to Ignite

If there is any messaging function related to this cache, this function need to be migrated to Ignite messaging.

###### Step 5: Change cache mode to LOCAL

Change the cache mode to LOCAL, which means this cache will not need to communicate with CacheControl anymore.

##### Step 6: Deploy and test

Theoretically after every cache is moved to Ignite, it is eligible to deploy to product environment and test; but in practice, caches could be deployed and tested in batch.

#### Cluster Set Up

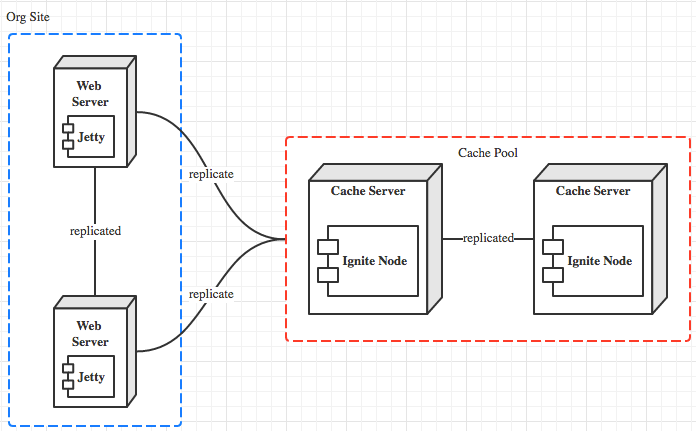
This phase includes step 7 and 8

##### Step 7: Remove CacheControl

As all the caches are in Ignite now, CacheControl is no longer needed and could be removed from the ActiveNet architecture safely, but this process could be postponed until we are confident about that. This step will involve code and deployment changes.

##### Step 8: Set up Ignite cluster environment

In step 2, an initial cache pool has been set up already, but it is used for cluster node register and discovery, there will be no cache deployed on it. In this step, cache pools will be set up not only for registration, but also for cache deployment.



#### LOCAL to REMOTE

This phase includes step 9 and 10

##### Step 9: Change iterating operation to remote server

This step is supposed to take a longer time. When moving cache to remote node, there may be problems about object serialization/deserialization. Also, when moving iterating operation to remote node, the query needs to be refactored as Ignite query.

There is a routine that can be followed when refactoring an iterating query.

* Include the iteration code to an inner class by implementing the query method;
* Fix any compile errors caused by lack of final modifier;
* Check if any other caches be referred during the search; if any, move this cache to Ignite in the first place;
* Check if any DB operation involved; if any, refactoring the code to exclude the DB operation;
* Set information like paging, ordering;
* Run the query using Ignite cache manager;
* Process the search result.

##### Step 10: Deploy and test

After all iterating operations are moved to remote cache server, caches don’t need to be deployed locally; instead all the cache will be saved in remote cache servers.

