Web Aplikasyonlarında Yüksek Performans için Caching Mekanizmaları

KurumsalJava.com

Özcan Acar
Bilgisayar Mühendisi
http://www.ozcanacar.com

Giriş

Facebook gibi, aynı anda binlerce insanın kullandığı web platformlarının temel sıkıntısı kullanılan kaynakların (hardware, software) yetersiz kalmasıdır. Kullanıcı sayısı arttıkça sistem kaynaklarının da artırılması gerekmektedir. Bu genelde bilgibankası için yeni server sistemlerini takviyesiyle gerçekleşir ya da mevcut server sistemlerinin hafıza (ram) alanlarının genişletilmesiyle. Kullanıcı sayısı çoğu platformlarda her gün arttığı için, kaynak kapasitenin artırılması da kaçınılmazdır. Web girişimciliğinde alt yapı giderlerinin önemli bir rol oynadığını düşünecek olursak, her kaynak kapasite artırımı, girişimcinin sırtına ek gider bindirecektir.

Yüksek trafiğe sahip web platformlarının en büyük sıkıntısı, web server sistemleri üzerinde çalışan program ile bilgibankası arasındaki veri trafiğinin zaman içinde dar boğaza dönüşmesi ve yetersiz kalmasıdır. Bu sorunu çözmek için sisteme yeni bilgibankası serverleri eklenir ve veriler replikasyon (replication – kopya etme) mekanizmaları kullanılarak sisteme ait tüm bilgibankası serverlerinde senkron tutulur. Bu sistemin bakımını ve geliştirilmesini zorlaştırabilecek bir gelişme olarak değerlendirilebilir.

Bu yazıda caching (ön belleğe alma) mekanizmaları kullanılarak dar boğaza dönen bilgibankalarının kapasite takviyesine gerek kalmadan nasıl rahata kavuşturulabileceğini yakından inceleyeceğiz.

Caching Nedir?

Ön belleğe alma olarak tercüme edebileceğimiz caching mekanizmaları ile kullanılmak istenen veri, verinin kaynağına ulaşmak zorunda kalmadan, bilgisayar hafızasında (ram) tutulur. Bilgisayar hafızasına erişim, verinin bulunduğu kaynağa (örneğin harddisk) erişimden daha hızlı olduğu için, verinin işlenme süresi kısaltılmış olur. Ayrıca verinin kaynağına erişim veri transferi açısından bir dar boğaz olabileceği için, caching mekanizmaları ile dar boğazlılık sorunu giderilir.

Caching mekanizmaları birçok alanda kullanılmaktadır. Bunlardan bazıları şöyledir:

- Ana işletim birimi (cpu) hafıza köprüsü (memory bus) üzerinden gerekli verileri elde ettikten sonra bu verileri kendi bünyesinde barındırdığı cache alanlarında saklar. Bu işletim biriminin çalışma hızını artırır.
- Harddiskler cache mekanizmaları kullanarak, diskler üzerinde bulunan verileri tekrar edinmek zorunda kalmadan kullanıcı sisteme aktarabilirler. Bu harddisklerin performansını artırır.
- Veriler bilgibankasından okunduktan sonra caching mekanizmaları kullanılarak hafızada tutulur. Bu verileri kullanan programın, sıkça bilgibankasını kullanmak zorunda olmadığı için performansını artırır.

Cache içinde tutulan veri şüphesiz orijinal verinin bir kopyasıdır. Er ya da geç veri asıl bulunduğu kaynak üzerinde (örneğin bilgibankası) değişikliğe uğrayacaktır. Bu durumda cache içinde bulunan kopyanın statüsü nedir?

Eskiye rağbet olsaydı, bit pazarına nur yağardı demiş atalarımız ☺ Kopya hiç bir zaman orijinalın yerini alamayacağı için kıymet taşımaz. Lakin bu bilgisayar sistemlerinde işlenen veriler için geçerli değil. Bir kopya veri, orijinalı değişmediği sürece onunla aynı değerdedir. Ne zaman orijinal veri değişikliğe uğradı, o zaman kopya için ölüm çanları çalmaya başlar. Orijinal veri değişikliğe uğradıktan sonra, cache içinde bulunan kopyanında değişikliğe uğraması gerekmektedir, aksi taktirde cache içinde bulunan veriyi kullanan program verinin en son haline sahip olmadığı için yanlış işlem yapabilir hale gelecektir. Buradan söyle bir sonucu çıkartabiliriz: "**Bir cache içinde tutulan verinin belli bir zaman diliminden sonra kendisini imha ederek, verinin tekrar asıl kaynağından edinilmesini sağlaması gerekmektedir**".

Cache içine atılan bir veri için geçerlilik süresi tespit edilmelidir. Aksi taktirde veri sonsuza dek (bilgisayar restart edilene kadar) cache içinde kalacak ve verinin tekrar asıl kaynağından edinilmesini engelleyecektir. Örneğin bir müşterinin bilgilerini ihtiva eden bir nesne cache içine atılmadan önce 30 saniye geçerli olacak şekilde yapılandırılabilir. Her 30 saniyenin bitiminde müşteri bilgileri bilgibankasından tekrar edinilerek, cachelenir. Buradan şöyle bir sonucu çıkartabiliriz: "**Caching mekanizmasının sağlıklı çalışabilmesi ve kullanıcı programı yanıltmaması için verilerin geçerlilik süresine (expiration) sahip olmasına gerekmektedir**". Geçerlilik süresi nesneyi cache sistemine atan program tarafından belirlenen bir özelliktir ve bir zaman birimini (saniye, dakika, saat) ihtiva eder. Örneğin sıkça değişikliğe uğramayan veriler için 1 gün, 1 ay gibi geçerlilik süresi tayin edilebilir. Sıkça değişikliğe uğrayan veriler için bu süre 30 saniyenin altında olacaktır.

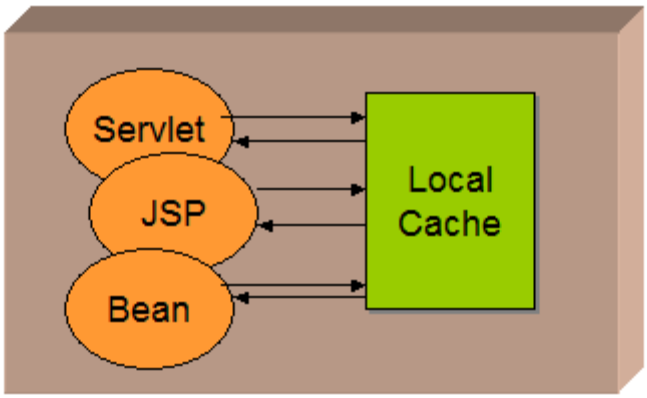
Bir veriyi caching sistemine atmamız ve geçerlilik süresini belirlememiz yeterli değildir. Veriyi tekrar edinebilmek için bu veriyi **adresleyebilmemiz** gerekmektedir. Bunu bir anahtar (key) kullanarak yapabiliriz. Bu anahtarın tüm sistem içerisinde eşsiz olması gerekmektedir. Sadece bu sayede istenilen veriye ulaşılabilir ve aynı anahtarı taşıyan iki verinin birbirlerini yok etmeleri engellenir.

Caching Türleri

Caching sistemleri **lokal** ve **global** olmak üzere iki gruba ayrılır.

Lokal Caching Sistemleri

Bu tür caching sistemlerinde caching sistemi kullanıcı program ile aynı hafıza alanında (in memory) faaliyet gösterir. Kullanıcı program ile aynı hafıza alanı ve JVM içinde bulunan caching sistemi, kullanıcı programa yakınlığından dolayı lokal caching sistemi olarak isimlendirilir.



Lokal caching sistemlerinin performansı yüksektir, çünkü caching işlemleri kullanıcı program ile aynı hafıza alanını paylaşan lokal cache üzerinde hızlı bir şekilde gerçekleştirilir.

Lokal cache sistemlerinde sadece bir tane cache kopyası (instance) mevcuttur. Her aplikasyon için bir lokal cache kopyasının oluşturulması gerekmektedir. İki değişik adres alanında faaliyet gösteren aplikasyon aynı lokal cache kopyasını kullanamazlar.

Daha sonra yakında inceleyeceğimiz EhCache caching sistemi ile lokal caching sistemleri uygulamak mümkündür.

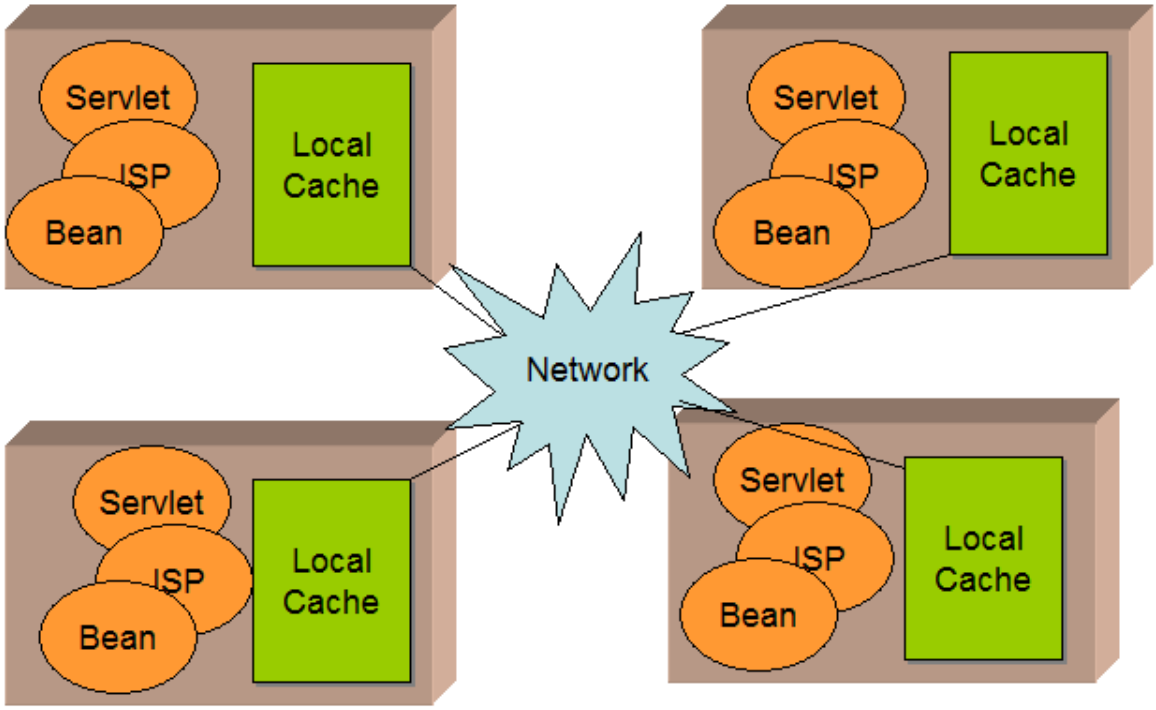
Global Caching Sistemleri

Birden fazla server üzerine dağıtılmış (distibuted) ama tek hafıza alanıymış gibi faaliyet gösteren caching sistemlerine global caching sistemi adı verilir. Global caching sistemleri kendi aralarında iki gruba ayrılır:

- Global tek hafıza caching sistemleri
- Global bölümsel (partial) hafıza caching sistemleri

Global Tek Hafıza Caching Sistemleri

Bu tür caching sistemlerinde her aplikasyon serveri kendi hafıza alanında bir lokal caching sistemine sahiptir. Birden fazla aplikasyon serveri ve lokal caching sisteminin kullanılması bir global cache oluşturulduğu anlamına gelmez. Global bir caching sisteminin oluşabilmesi için ağ içindeki lokal caching sistemlerinin birbirlerinden haberdar olmaları gerekmektedir. Kullanılan caching sistemleri ağ üzerinden birbirleriyle bağlantı kurarak, bünyelerinde meydana gelen değişiklikleri ağ içindeki diğer lokal cachelere bildirirler. Bu işlem için JGroups ya da JMS gibi komünikasyon teknolojileri kullanılır. Bu sayede bir global caching sistemi oluşur.

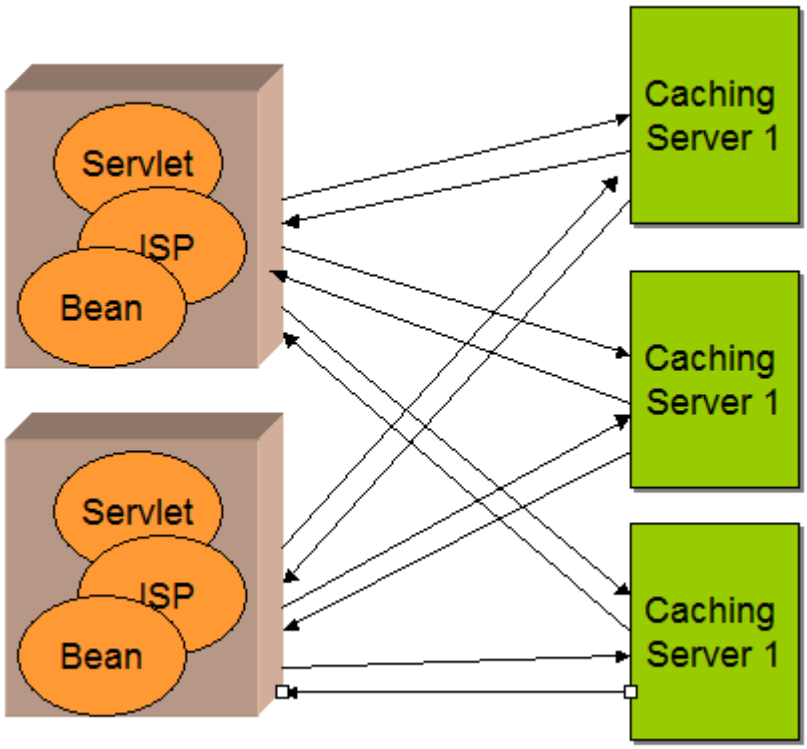


Lokal cacheler arası verinin replikasyonu (kopyalanması) ile tüm lokal cacheler aynı veriye sahip olur.

Bu tarz global bir caching sistemi oluşturabilmek için verinin lokal cache sistemleri arası replike (kopyalanması) edilmesi gerekmektedir. Daha sonra yakından inceleyeceğimiz EhCache bu tür faaliyet gösterebilen bir caching sistemidir.

Global Bölümsel (Partial) Hafıza Caching Sistemleri

Bu tür global caching sistemlerinde caching komponentleri arasında replikasyon yapılmaz.



Global caching sistemini kullanmak isteyen her aplikasyon, sistemdeki tüm caching serverleri ile bağlantı kurarak, caching işlemlerini gerçekleştirir. Kullanılan özel bir caching API yardımıyla ile veriler değişik caching serverlerine dağıtılır. Bu yüzden bu tür faaliyet gösteren caching sistemlerine global bölümsel caching sistemleri adı verilmektedir, çünkü veriler değişik caching serverlerine dağıtılarak cachelenir. Daha sonra yakından inceleyeceğimiz MemCached bu tarz caching sistemleri kurmak için kullanılabilecek bir caching sistemidir.

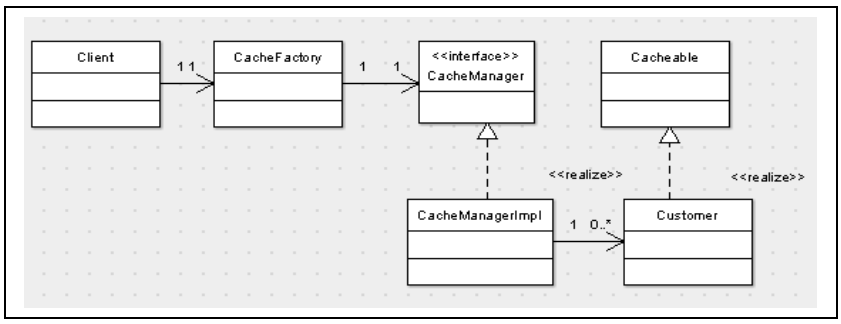
Caching Konseptleri

Bu yazıda ismi geçen caching konseptlerini yakından inceleyelim:

- Ön belleğe anılacak olan verinin kendisi (**cacheable object**) . Örneğin bir sınıftan oluşturulan bir nesne veri olarak ön belleğe alınabilir.
- Verinin ön bellekteki kalma süresi (**expire time**). Bu değer baz alınarak, verinin ön bellekte hangi zaman dilimi içim kalacağı belirlenir. Verinin ön bellek içinde sonsuza dek kalmasını engellemek için limitli bir zaman diliminin seçilmesi gerekmektedir, örneğin 30 saniye.
- Veriyi ön belleğe yerleştirmek ve tekrar edinmek için kullanılan anahtar (**key**). Bu anahtarın sistem bünyesinde eşsiz (unique) olması gerekmektedir. Aksi taktirde aynı anahtara sahip verilerin birbirlerini yok etmeleri ihtimali oluşmaktadır.

- Verileri ön belleğe almak için put olarak isimlendirilen işlem gerçekleştirilir. put işleminde ön belleğe alınacak verinin kendisi, verinin ön bellekteki kalış süresi ve ve veriyi adresleyen anahtar (key) parametre olarak kullanılır.
- Verileri ön bellekten edinmek için get işlemi gerçekleştirilir. get işleminde veriye ulaşmak için put işleminde oluşturulan anahtar (key) kullanılır.

Basit Bir Cache Sistemi



Java'da basit bir cache sistemi uygulayarak, caching konseptlerini yakından inceleyelim

```
package com.kurumsaljava.com.sample.caching;

public interface Cacheable
{
	void setExpiration(long expire);

}
```

Cacheable isminde, caching sistemine eklenene sınıfların implemente ettiği bir interface sınıf oluşturuyoruz. Biliyoruz ki caching sistemine eklenen her nesne Cacheable tipindendir. Bu bizim ilerde cache üzerinde yapılması gereken işlemleri kolaylaştıracak bir yapıdır.

```
package com.kurumsaljava.com.sample.caching;

public interface CacheManager
{
	void set(String key, Cacheable obj, long expire);
	Cacheable get(String key);
}
```

Değişik tipte caching mekanizmaları uygulayabilmek için CacheManager isminde bir interface sınıf tanımlıyoruz. Bu interface bünyesinde set (veriyi cache sistemine ekle) ve get (veriyi cache sisteminden edin) metotları yer almaktadır. set() metodunun imzasında (method signature) üç değişik parametre yer almaktadır. Bunlar:

- key: Veriye ulaşmak için kullanılan anahtar.
- obj: Ön belleğe alınan ve Cacheable interface sınıfını implemente eden sınıf nesnesi.

- expire: Verinin ön bellekte kalma süresi.

```
package com.kurumsaljava.com.sample.caching;

import java.util.HashMap;
import java.util.Map;

public class CacheManagerImpl implements CacheManager
{

	private final Map<String, Cacheable> cache =
		new HashMap<String, Cacheable>();

	public Cacheable get(String key)
	{
		Customer customer = (Customer)cache.get(key);
		if(customer.getExpire() > System.currentTimeMillis())
		{
			return customer;
		}
		else
		{
			cache.remove(key);
			return null;
		}
	}

	public synchronized void set(String key,
			Cacheable obj, long expire)
	{
		obj.setExpiration(expire+System.currentTimeMillis());
		cache.put(key, obj);
		System.out.println(key + " eklendi.");
	}
}
```

CacheManagerImpl bünyesinde CacheManager sınıfını implemente ediyoruz. Caching sistemi olarak bir HashMap seçilmiştir. set() metodu ***synchronized*** olduğu için cache sistemine eklemeler kontrollü bir şekilde yapılmaktadır.

```
package com.kurumsaljava.com.sample.caching;

public class Customer implements Cacheable
{

	private long expire;

	private String name;

	public String getName()
	{
		return name;
	}

	public void setName(String name)
	{
```

```
            this.name = name;
      }

      public void setExpiration(long expire)
      {
            setExpire(expire);
      }

      public long getExpire()
      {
            return expire;
      }

      public void setExpire(long expire)
      {
            this.expire = expire;
      }
}
```

Bilgibankasında bulunan müşteri bilgilerini ön belleğe alabilmek için Customer isminde bir sınıf oluşturuyoruz. Bu sınıf Cacheable interface sınıfını implemente ettiği için ön bellekte tutulabilir. Customer sınıfından olan bir nesnenin ön bellekte kalma süresini tayin etmek üzere long tipinde (expire) bir sınıf değişkeni tanımlıyoruz. Bu değişken nesnenin ön bellekte kalma süresini mili saniye olarak tutacaktır. Örneğin 30 saniyelik kalma süresi düşünüldüyse, expire = 30000 değerini taşıyacaktır (1 saniye 1000 ms).

```
package com.kurumsaljava.com.sample.caching;

public class CacheFactory
{
      private static final CacheManager manager =
                  new CacheManagerImpl();

      private CacheFactory()
      {

      }

      public static CacheManager getCacheManager()
      {
            return manager;
      }
}
```

CacheFactory sınıfı ile, hangi CacheManager implementasyonunun kullanıldığı bilgisini kullanıcı (Client) sınıftan saklamış oluyoruz. Bu ilerde bize, Client sınıfını etkilemeden kullanılan CacheManager implementasyonunu değiştirme imkanı tanıyacaktır. Client sınıfının muhatabı sadece CacheFactory sınıfı olduğu için, Client ve kullanılan caching sistemi arasında esnek bir bağ oluşturulmuştur.

```
package com.kurumsaljava.com.sample.caching;

public class Client {

      public static void main(String[] args)
                        throws InterruptedException
```

```
	{
		Customer cus1 = new Customer();
		cus1.setName("Ahmet");
		CacheFactory.getCacheManager().set("cus1_ahmet",
				cus1, 20000);

		Customer cus2 = new Customer();
		cus2.setName("Mehmet");
		CacheFactory.getCacheManager().set("cus2_mehmet",
				cus2, 20000);


		cus1 = (Customer)CacheFactory.
				getCacheManager().get("cus1_ahmet");
		cus2 = (Customer)CacheFactory.
				getCacheManager().get("cus2_mehmet");

		if(cus1 != null) System.out.println(cus1.getName());
		else System.out.println("cus1 is null");

		if(cus2 != null) System.out.println(cus2.getName());
		else System.out.println("cus2 is null");

		Thread.sleep(30000);

		cus1 = (Customer)CacheFactory.
				getCacheManager().get("cus1_ahmet");
		cus2 = (Customer)CacheFactory.
				getCacheManager().get("cus2_mehmet");

		if(cus1 != null) System.out.println(cus1.getName());
		else System.out.println("cus1 is null");

		if(cus2 != null) System.out.println(cus2.getName());
		else System.out.println("cus2 is null");

	}
}
```

Cache kullanımını göstermek için Client isminde, caching sistemini kullanan bir sınıf oluşturuyoruz. cus1 ve cus2 isimlerinde iki Customer nesnesi oluşturarak, bu nesneleri 20 saniye ön bellekte kalma süreleri olmak üzere ön belleğe alıyoruz. Client akabinde cache sisteminden bu iki nesneyi edinerek, müşteri isimlerini ekranda görüntülüyor. Client 30 saniye bekledikten sonra aynı işlemi tekrarlıyor. Ekran çıktısı şu şekilde olacaktır.

```
cus1_ahmet eklendi.
cus2_mehmet eklendi.
Ahmet
Mehmet
cus1 is null
cus2 is null
```

Client 30 saniye bekledikten sonra edinilen Customer nesnelerinin null değerinde olduğunu görüyoruz, çünkü bu nesneler ön bellekte kalma süreleri (20 saniye) dolduğu için caching sisteminden silindiler.

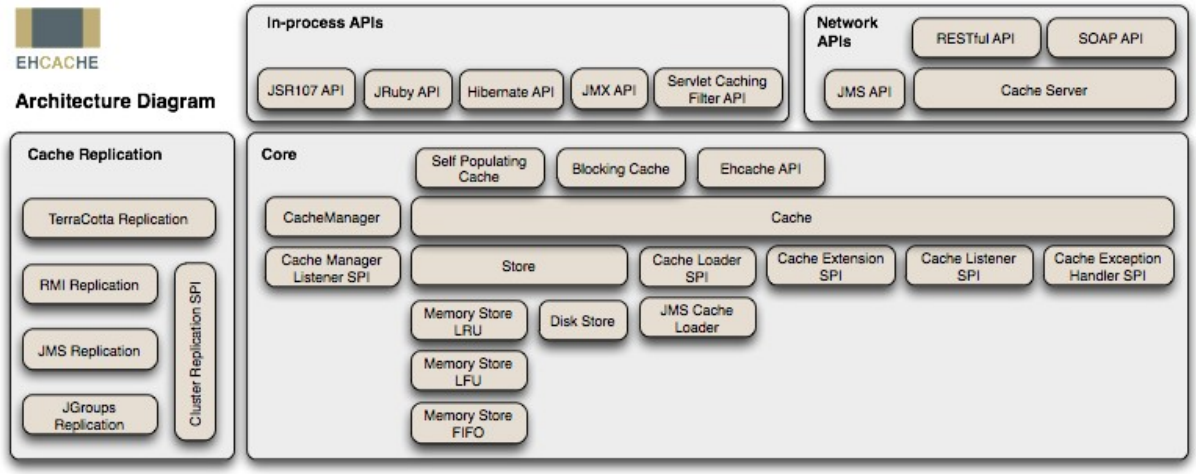
Bu yapılabilecek en basit cache sistemidir, lakin kurumsal projelerde böyle bir cache mekanizmasının kullanımını tavsiye etmiyorum. Bunun bazı nedenlerini şu şekilde sıralayabiliriz:

1. Bu şekilde global bir caching mekanizması uygulamak mümkün değildir ya da çok zordur. Her JVM (java virtual machine) içinde bir caching sistemi kopyası olacağından, birden fazla JVM'in olduğu sistemlerde caching sistemlerinin senkronize edilmesi gerekliliği doğmaktadır.
2. Kullanılan senkronizasyon mekanizmaları sistem performansını negatif etkileyebilir.

Kurumsal alt yapılarda genelde birden fazla aplikasyon serveri hizmet verdiği için, caching mekanizmasının dar boğaz haline gelmesini önlemek için birden fazla servere dağıtılması (distributed cache – global cache) gerekmektedir. Bu durum doğal olarak beraberinde senkronizasyon sorunlarını getirmektedir. Cluster olarak tabir edilen bu tür (birden fazla caching sistemi kopyasınından oluşan, beraber çalışan global caching sistemi) yapılarda her caching sistemi global bir caching sisteminin parçası olarak işlev göstermek zorundadır.

EhCache: Java İçin Caching Sistemi

EhCache[1], Java projelerinde kullanılabilecek ve open source olan bir caching sistemidir.



EhCache programını caching sistemi olarak kullanabilmek için ehcache.xml isminde bir konfigürasyon dosyasının oluşturulması gerekiyor. Bunun bir örneği aşağıda yer almaktadır.

```
<ehcache>

    <diskStore path="c:/temp"/>

    <defaultCache
        maxElementsInMemory="10000"
        eternal="false"
        timeToIdleSeconds="120"
```

[1] Bakınız: http://ehcache.sourceforge.net/

```
        timeToLiveSeconds="120"
        overflowToDisk="true"
        diskSpoolBufferSizeMB="30"
        maxElementsOnDisk="10000000"
        diskPersistent="false"
        diskExpiryThreadIntervalSeconds="120"
        memoryStoreEvictionPolicy="LRU"
        />


    <cache name="sampleCache1"
        maxElementsInMemory="10000"
        eternal="false"
        overflowToDisk="true"
        timeToIdleSeconds="300"
        timeToLiveSeconds="600"
        />

</ehcache>
```

Gerekli Jar dosyalarını (ehcache.jar) projeye ekledikten sonra, aşağıda yer aldığı şekilde caching sistemi kullanılabilir.

```
package com.kurumsaljava.com.sample.ehcache;

import java.io.Serializable;
import java.net.URL;

import net.sf.ehcache.Cache;
import net.sf.ehcache.CacheManager;
import net.sf.ehcache.Element;

public class Client
{
      public static void main(String[] args)
      {
            URL url = Client.class.getResource("/ehcache.xml");
            CacheManager manager = new CacheManager(url);
            Cache cache = manager.getCache("sampleCache1");
            Element element = new Element("key1", "value1");
            cache.put(element);

            cache = manager.getCache("sampleCache1");
            element = cache.get("key1");
            Serializable value = element.getValue();
            System.out.println(value);
      }

}
```

EhCache ile hafıza alanı haricinde harddisk alanı kullanılabilmektedir. Hafızanın caching sistemi için yetmediği durumlarda verilerin harddisk üzerine kaydırılması (swap) mümkündür.

EhCache ile global, birden fazla server üzerinde çalışan caching sistemi oluşturmak mümkündür. Değişik tarza protokol ve teknolojiler kullanılarak global caching sisteminin

parçası olan serverler ve sahip oldukları veriler senkron tutulur. Global caching sistemi oluşturmak için EhCache tarafından desteklenen protokoller:

- RMI
- JGroups
- JMS
- Terracotta
- Cache Server

EHCache, kullanıldığı programla ile aynı hafıza alanını (in memory) paylaştığı için get ve put operasyonları uzun zaman almamaktadır. Bu caching mekanizmasında yüksek performans anlamına gelmektedir. Bu durum EhCache global cache sistemi olarak kullanıldığında da geçerlidir. put işlemi aynı hafıza alanında bulunan EhCache kopyası (instance) üzerinde yapılır. Akabinde EhCache tarafından kullanılan dağıtım (distibution) protokolü tarafından bu değişiklik diğer serverlerdeki EhCache kopyalarına transfer (replication) edilir.

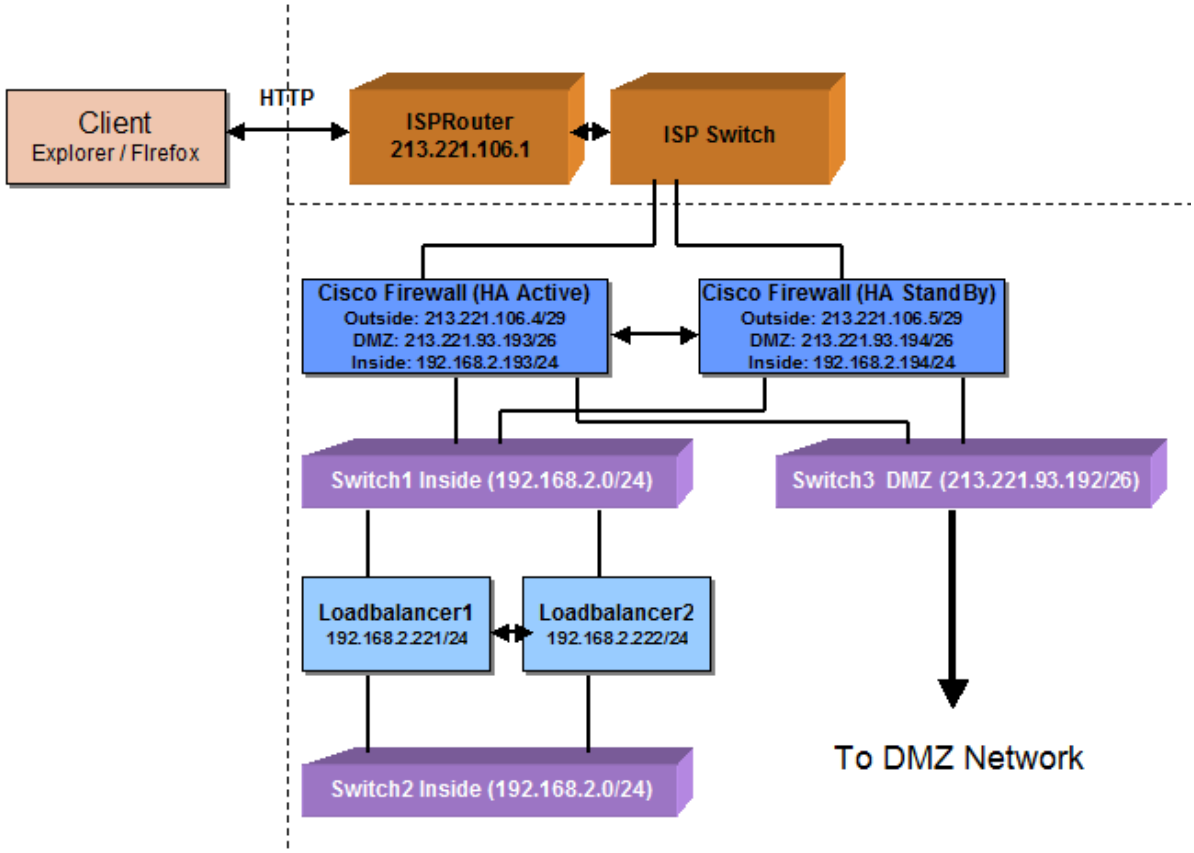
MemCached

MemCached [2] client-server tarzı çalışan bir global caching sistemidir. Open Source olan bu ürünü yazılımını yaptığım ve alt yapısını oluşturduğum BizimAlem.com [3]projesinde kullandık.

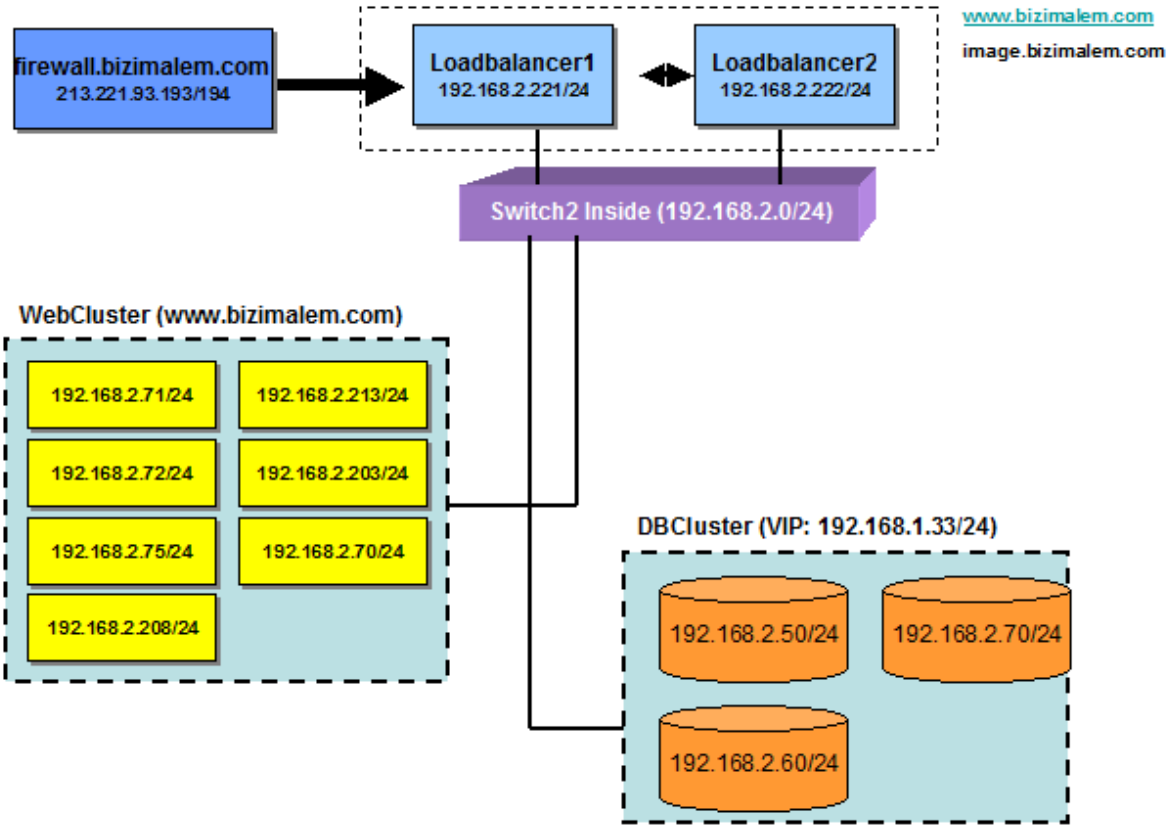
BizimAlem.com aynı anda binlerce insanın online olduğu, özellikle Avrupa'da yaşayan Türklerin kullandığı bir web community platformudur. Kırka yakın serverin çalıştığı sistemde, zaman içinde bilgibankaları, dikey (vertical) ve yatay (horizontal) genişletilmiş olmalarına rağmen yetersiz kalmaya başladılar. Bilgibankası kullanımı fazla olduğu için, aplikasyon server ile bilgibankaları arasındaki trafik dar boğaza dönüştü. Sorunu çözmek için yeni bir caching mekanizmasının uygulanması gerekiyordu. Caching sistemi olarak MemCache'i kullanmaya karar verdik.

BizimAlem.com'un teknik alt yapısı caching öncesi su şekilde idi:

[2] Bakınız: http://www.danga.com/memcached/
[3] Bakınız: http://www.bizimalem.com



Trafik ISP router ve switch komponentleri aracılığıyla BizimAlem firewall sistemlerine iletilmektedir. HA (high available) firewall Cluster iki Cisco Firewall komponentinden oluşmaktadır. Aktif olan (resimde solda) Cisco Firewall 192.168.2.193/24 ağı için olan trafiği **Switch1** aracılığıyla aynı ağın parçası olan Loadbalancer komponentlerine iletmektedir. Loadbalancer komponentleri Cisco Firewall komponentlerinde olduğu gibi HA konfigürasyonu ile çalışmaktadır. Komponentlerden birisi devre dışı kaldığında, pasif konumdaki komponent trafiği üstlenerek, sistemi ayakta tutmaktadır.



BizimAlem.com için **Webcluster** olarak isimlendirilen bir grup aplikasyon serveri mevcuttur. Her serverin üzerinde bir Resin (Tomcat benzeri bir server) aplikasyon serveri çalışmaktadır. BizimAlem için geliştirdiğimiz web tabanlı programın birer kopyası bu aplikasyon serverlerinde çalışır (deployed) durumdadır. Kullanıcılardan gelen istekler (request) Loadbalancer komponenti tarafından Webcluster içinde bulunan herhangi bir aplikasyon servere iletilir. Böylece yük birden fazla server dağıtılmış olur. Bu kullanıcının farkına varmadığı bir durumdur, çünkü Webcluster içinde bulunan serverler sadece http://www.bizimalem.com adresinden erişilebilir durumdadır. Aplikasyon serverleri connection pool mekanizması üzerinden **DBCluster** olarak isimlendirilen bilgibankası serverleri üzerinde işlem yapabilirler.

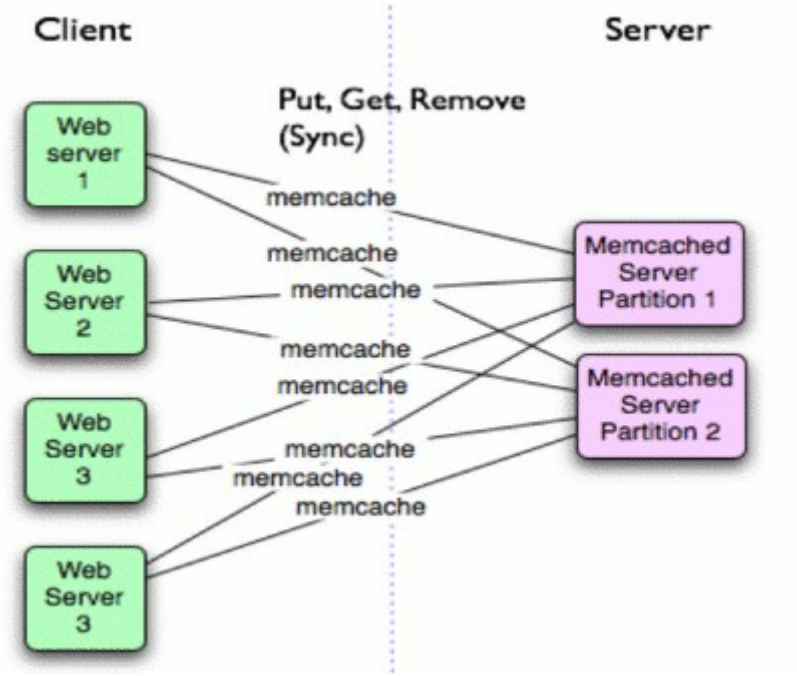
Kullanıcı sayısı arttıkça, bilgibankası üzerindeki yük artmaya başlar. Bilgibankası istekleri karşılayamaz duruma gelir. Bunun en bariz göstergesi aplikasyon serverleri tarafından kullanılan connection pooller (bilgibankası bağlantı havuzu) içinde bağlantı kalmaması (connection saturation) sonucu, yeni bilgibankası bağlantılarının yapılamamasıdır. Aplikasyon serverleri ile bilgibankası arasındaki dar boğazı ortadan kaldırmak için MemCached cache sistemi kullanmaya karar verdik.

MemCached daha öncede belirttiğim gibi client-server tarzı çalışan bir caching sistemidir. Fiziksel bir server üzerinden MemCached kurulduktan sonra, aşağıdaki şekilde çalıştırılır.

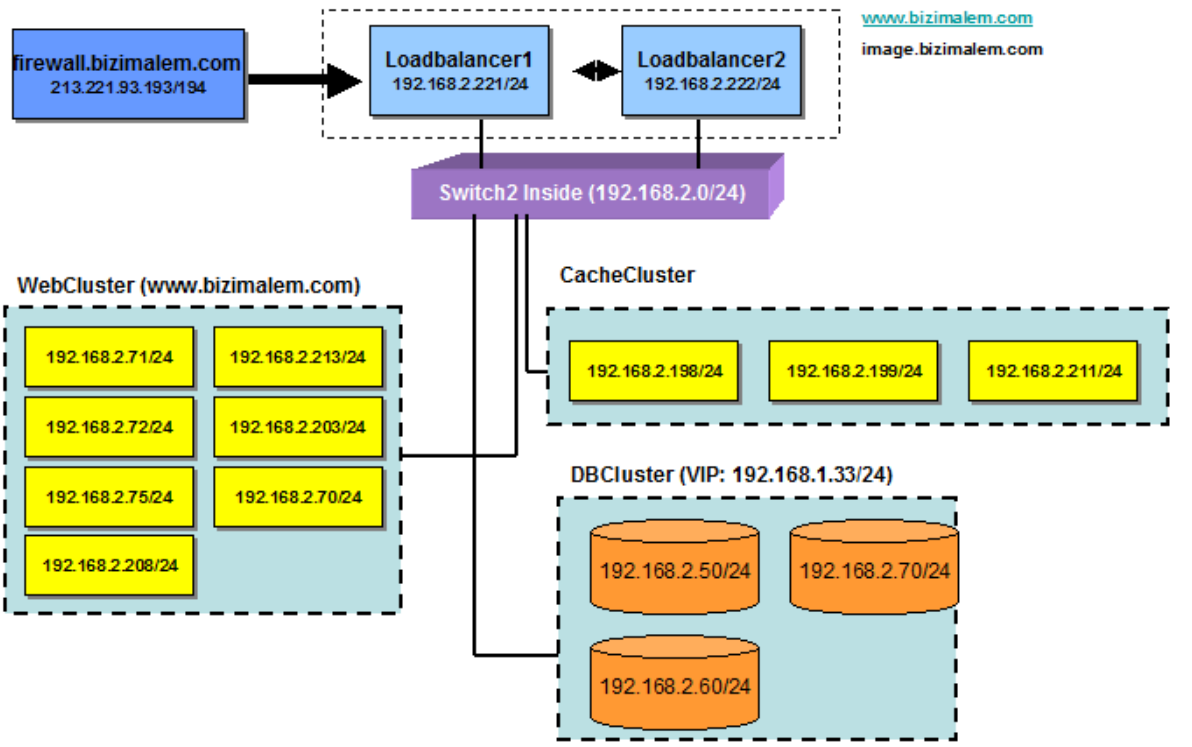
```
./memcached -d -m 2048 -l 192.168.1.10 -p 11211
```

Yukarda yer alan örnekte MemCached 192.168.1.10 IP numaralı server üzerinde 2 GB hafıza alanını kullanacak ve 11211 numaralı porttan erişilebilir olacak şekilde çalıştırılmıştır.

Birden fazla MemCached ağ içinde aktif olabilir. MemCached serverler arasında veriler replikasyon (kopyalama işlemi) usulüyle senkron tutulmaz. MemCached serverlerine verileri transfer eden Client API (daha sonra detaylı olarak göreceğiz) cachelenmesi gereken verinin anahtarı yardımıyla eşsiz (unique) bir hash değeri oluşturarak, listeden bir MemCached serverini seçer ve veriyi direk bu MemCached serverine transfer eder. Veriyi edinmek için tekrar aynı anahtar kullanıldığında Client API yine aynı hash değerini oluşturarak, verinin hangi MemCached server üzerinde olduğunu belirler ve veriyi temin eder.



Bir önceki resimde görüldüğü gibi aplikasyon serverleri (Web Server ismini taşıyan komponentler) ağdaki her MemCached serverine bağlantı kurarak, caching sistemini kullanırlar.



Bir önceki resimde görüldüğü gibi **CacheCluster** isminde, üç MemCached serverin bulunduğu bir yapı oluşturduk. Switch2 üzerinden WebCluster içinde bulunan aplikasyon serverleri CacheCluster içinde bulunan MemCached serverlere erişebilirler. Caching sistemini kullanabilmek için Java dilinde implemente edilmiş bir MemCached Client API implementasyonuna ihtiyacımız var.

MemCached Client API

MemCached serveri ile bağlantı kurup, işlem yapabilmek için değişik dillerde Client API'ler[4] oluşturulmuştur. Java dilinde Greg Whalin[5] tarafından implemente edilen MemCached Client API'yi kullanıyoruz.

```
package smart.core.cache;

import java.util.Date;
import java.util.Map;
import com.danga.MemCached.MemCachedClient;
import com.danga.MemCached.SockIOPool;


public class MemcachedImpl extends BaseCache
{

      protected static MemCachedClient client =
            new MemCachedClient(Thread.currentThread().
```

[4] Bakınız: http://www.danga.com/memcached/apis.bml
[5] Bakınız: http://whalin.com/

```
                        getContextClassLoader());

    static
    {
      // get mcache servers
      String[] serverlist =
          getProperty( "memcached.servers" ).split( "," );

      // get server weights
      Integer[] weights      =
          new Integer[serverlist.length];

      String[] serverWeights =
          getProperty("memcached.servers.weights" ).split( "," );

      for ( int i = 0; i < serverWeights.length; i++ )
      {
          weights[i] = new Integer( serverWeights[i] );
      }


      // idle time set to 4 hours
      long maxIdle = 1000 * 60 * 60 * 6;

      // socket read timeout
      int readTO = 1000 * 3;

      // initialize the pool for memcache servers
      // see javadocs on this method
      // for tuning the connection pool
      SockIOPool pool = SockIOPool.getInstance();
      pool.setServers( serverlist );
      pool.setWeights( weights );
      pool.setInitConn( Integer.parseInt(getProperty("initcon")) );
      pool.setMinConn(  Integer.parseInt(getProperty("mincon")) );
      pool.setMaxConn(  Integer.parseInt(getProperty("maxcon")) );
      pool.setMaintSleep(  Integer.parseInt(getProperty("maintsleep")) );
      pool.setNagle( Boolean.getBoolean(getProperty("nagle")) );
      pool.setSocketTO( readTO );
      //pool.setSocketConnectTO( 0 );
      pool.setMaxIdle( maxIdle );
      pool.setFailover(Boolean.getBoolean(getProperty("failover")));
      pool.setFailback(Boolean.getBoolean(getProperty("failback")));
      pool.initialize();
      getClient().setCompressEnable(
                Boolean.getBoolean(getProperty("compression")) );
  }



    public static MemCachedClient getClient()
    {
          return client;
    }



    public void set(String key, Object obj)
    {
          getClient().set(key, obj);
```

```
	}


	public void set(String key, Object obj, Date date)
	{
		getClient().set(key, obj, date);
	}


	public Object get(String key)
	{
		return getClient().get(key);
	}


	public Map getStats()
	{
		return getClient().stats();
	}

	public void delete(String key)
	{
		delete(key,null);
	}

	public void delete(String key, Date expiry)
	{
		getClient().delete(key,expiry);
	}



	public Map getStats(String[] server)
	{
		return getClient().stats(server);
	}

}
```

Yukarda yer alan MemcachedImpl sınıfı ile MemCached serverlerini kullanmaya başlayabiliriz. Sistem için gerekli ayarları (örneğin hangi MemCached serverlerin kullanıldığı) memcached.properties dosyasında tutuyoruz. Bu dosyanın içeriği aşağıda yer almaktadır.

```
#PROD

# memcached.server=server1:port,server2:port....
memcached.servers=192.168.2.198:11211,192.168.2.199:11211,192.168.2.211:11211
memcached.servers.weights=1,2,1
memcached.activated=true


initcon                 = 100
mincon                  = 100
maxcon                  = 512
maintsleep              = 1000
nagle                   = false
alivecheck              = true
```

```
failover                = false
failback                = false
compression             = false

domain                  = prod

impl                    = smart.core.cache.MemcachedImpl
```

memcached.servers anahtarı, kullanılan MemCached serverlerin IP adresleri ve port numaralarını ihtiva etmektedir. Kullanılan MemCached serverler virgül ile birbirlerinden ayrılmıştır. Buna göre 192.168.2.198 - 192.168.2.199 - 192.168.2.211 IP numaralı serverler MemCached serverleridir ve 11211 numaralı port üzerinden bu servise ulaşılabilir.

Kullanılan bazı diğer parametreler:

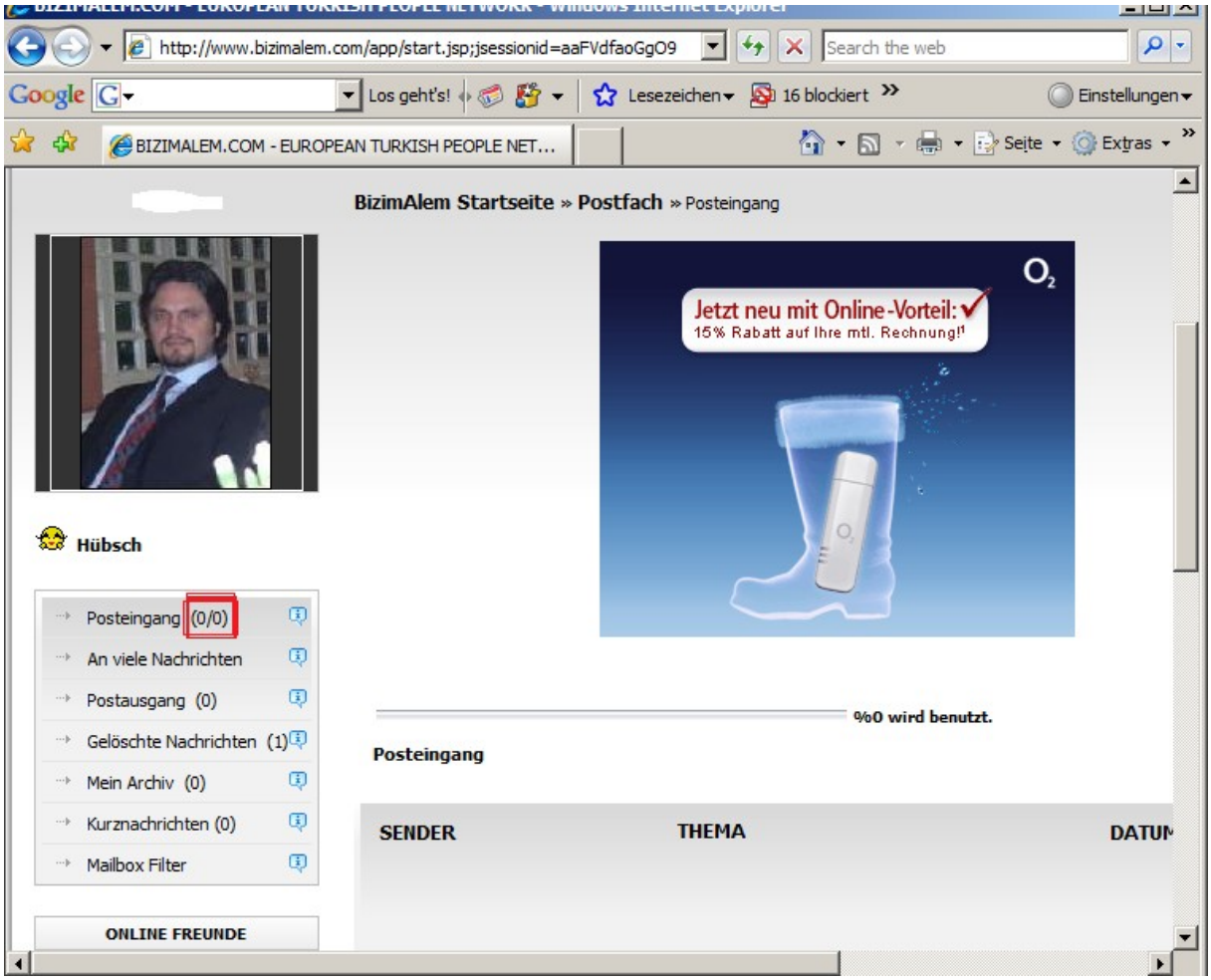
- **initcon = 100** – Client API her MemCached server için başlangıçta 100 adet TCP bağlantısı kurar. Böylece bir bağlantı (connection) havuzu (pool) oluşturularak, caching işlemleri hızlandırılmış olur.
- **mincon = 100** – Bağlantı havuzunda en az 100 bağlantı olacak şekilde ayarlamalar yapılır.
- **maxcon = 512** – Bağlantı havuzunda en fazla 100 bağlantı olacak şekilde ayarlamalar yapılır. Client API otomatik olarak bağlantı havuzunun hacmini ayarlar.

Java için kullandığımız Client API implementasyonunda MemCached serverlere olan bağlantı devamlı kontrol edilir. Eğer MemCached serverlerinden birisi görevini yerine getirmiyorsa, otomatik olarak mevcut server listesinden alınır. Bu şekilde çalışmayan komponentlerin devre dışı kalması sağlanır.

Bir sonraki kod bölümünde MemCached caching serverlerinin BizimAlem kodu bünyesinde nasıl kullanıldığı yer almaktadır.

```
public String getIncomingMailCounter() throws Exception
{
      logger.debug("getIncomingMailCounter()");

      String key = KeyAttributes.getKey(
                  KeyAttributes.KEY_MAILBOX_COUNTER , getUsername());
      String vo =null;

      try
      {

            vo = (String)CacheManager.instance().get(key);

            if(vo == null)
            {
                  vo = getIncomingMailCounter();
                        CacheManager.instance().set(key, vo,
                                    Expire.EXPIRE_IN_MINUTE_5);
            }
      }
      catch(Exception e)
```

```
        {
                reportError(e);
        }
        return vo;
        }
```



getIncomingMailCounter() metodu ile bir önceki resimde görüldüğü gibi posta kutusunda olan emaillerin adedi hesaplanmaktadır. Metot içinde vo isminde bir String tanımlanmaktadır. vo ismindeki değişken posta kutusundaki mektup adedini ihtiva eder. Bu veriyi bilgibankasından edinmeden önce, ClientManager sınıfı yardımıyla caching sisteminde arıyoruz. Bu işlem için bir anahtar oluşturmamız gerekiyor. ***KeyAttributes.getKey()*** metodu yardımıyla kullanıcıya has anahtar oluşturulmaktadır. Login yapmış bir üyeye ait bilgileri caching sisteminde tutabilmek için, anahtarın içinde üyenin ismi yer almaktadır. Üyenin isminin **test1** seklinde olacağını düşünürsek, oluşturulan anahtar **mailcounter:test1** şeklinde olacaktır. Bu şekilde değişik isimdeki üyeler için üyeye özel verileri caching sisteminde tutmak mümkündür.

MemCached EhCache Karşılaştırması

Doğal olarak hangi caching sisteminin daha performanslı olduğu sorusu akla gelmektedir. Bu sorunun cevabını vermeden önce iki sistemin özeliklerini tekrar gözden geçirelim.

MemCached özellikleri:

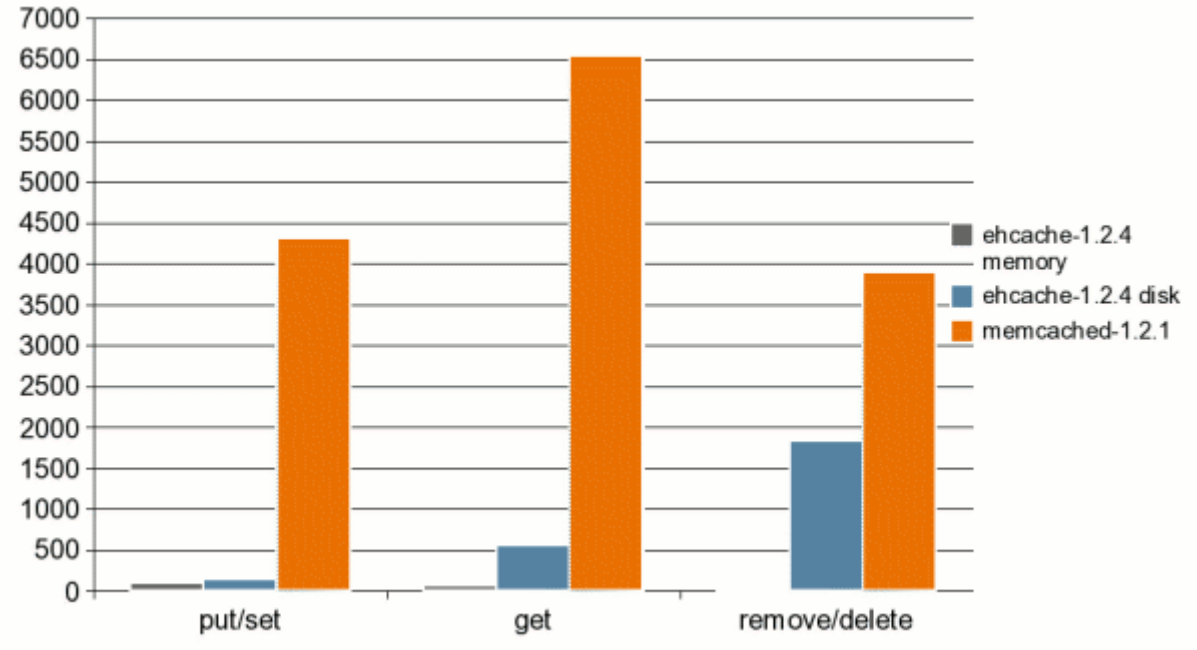
- Client-Server tabanlı ve sadece hafıza alanında (in memory caching) caching yapabiliyor.
- Caching sistemini kullanan aplikasyon ile aynı hafıza alanını (im memory) paylaşacak yapıda değil. Sadece server olarak hizmet verebilir.
- Değişik dillerdeki Client API ile MemCached serverine bağlanmak ve işlem yapmak mümkündür.
- C dilinde implemente edilmiş. Linux üzerinde kurulum için libevent [6] kütüphanesini gerektirmektedir.

EhCache özellikleri:

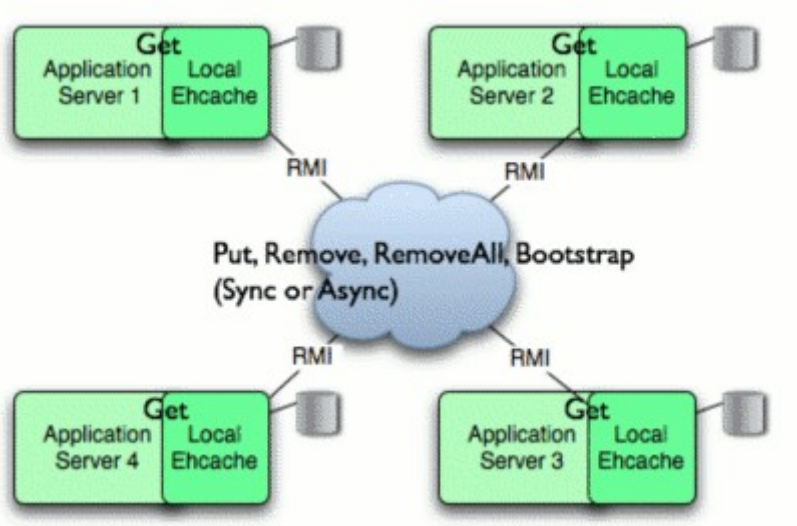
- Java dilinde implemente edilmiş. Kullanıcı aplikasyon ile aynı hafıza alanını (in memory) paylaşabilir. Bu durum, biraz sonra inceleyeceğimiz performans istatistiklerinin de gösterdiği gibi EhCache caching sisteminin performans açısından daha iyi olduğunun göstergesidir. Bu durum ama EhCache'in MemCached'den daha güvenilir olduğu anlamına gelmemektedir.
- EhCache hafıza alanı (memory caching) yanı sıra disk alanlarını (harddisk) caching için kullanabilir. MemCached bu özelliğe sahip değildir.
- JGroups, JMS gibi komünikasyon teknolojileri kullanılarak global caching sistemi oluşturmak mümkündür. Birden fazla server üzerinde oluşturulan global caching sisteminde, cachelenen veriler replikasyon teknikleri kullanılarak caching serverlerinin verinin ayni kopyasına sahip olmaları sağlanır.

Grek Luck[7] tarafından yapılan kıyaslamada[8] EhCaching ile oluşturulan caching sisteminin MemCache'e kıyasla daha performanslı olduğu görülmektedir. Bunun en büyük nedenlerinden birisi EhCache caching sisteminin kullanıcı aplikasyon ile ayni hafıza alanını paylaşabilme (in memory) özelliğine sahip olmasıdır. Bu sayede get ve set operasyonları ağ üzerinde bir server ile bağlantı kurmak zorunda kalınmadan çok hızlı bir şekilde gerçekleştirilebilmektedir.

[6] Bakınız: http://www.monkey.org/~provos/libevent/
[7] Bakınız: http://www.gregluck.com
[8] Bakınız: http://gregluck.com/blog/archives/2007/05/comparing_memca.html



Bir önceki resimde görüldüğü gibi EhCache sabit disk alanı kullanılmış olsa bile verilerin get ile alınması ve put ile aktarılması işlemlerinde MemCached'den öndedir.



Görüldüğü gibi hafıza ve sabit disk alanı kullanabilen EhCache daha performanslı bir görüntü çizmektedir. Peki bu yeterli mi? Yaptığım tecrübeler sadece yüksek performansın yeterli olmadığını gösterdi. Kullanılan caching sisteminin aynı zamanda güvenilir (reliable) alt yapı komponenti olması gerekmektedir. Bununla neyi kastettiğimi yakından inceleyelim.

EhCache ile global bir caching sistemi oluşturmak için, birden fazla server üzerinde çalışan EhCache sistemlerini senkron halde tutmak gerekmektedir Bunun bir örneği bir önceki resimde yer almaktadır. Application Server 1 bünyesinde faaliyet gösteren EhCache (local) bünyesinde bulunan bir veri üzerinde değişiklik yapıldığında ya da yeni bir verir caching sistemine eklendiğinde, ağ içinde bulunan diğer EhCache sistemlerinin bu değişikliklerden

haberdar olmaları gerekmektedir. Bunun için EhCache JGroups ya da JMS gibi komünikasyon teknolojilerinden faydalanır. Bu beraberinde, global caching sistemini yönetmek için alt yapı çalışmalarını beraberinde getirir. Örneğin replikasyon için Terracotta gibi bir ürün kullanıldığında, Terracotta için özel bir serverin faaliyet göstermesi gerekmektedir. Replikasyon için gerekli alt yapı çalışmaz hale geldiğinde bir EhCache serverinde meydana gelen değişiklik diğer EhCache serverlerine aktarılamaz ve global caching sistemi güvenilir bir şekilde hizmet veremez hale gelir. Bu durumda caching sisteminin ne kadar yüksek performansla çalıştığı önemini yitirmektedir. EhCache ile global caching sistemleri oluşturmak bu açıdan bakıldığında sorunlar yaratabilir.

MemCached ile oluşturulan global caching sistemlerinde durum farklıdır. MemCached serverleri arasında replikasyon gerekmemektedir. Her MemCached server kullanılan caching anahtarına bağımlı olarak bazı verileri bünyesinde barındırır. Client API üzerinden veriler mevcut MemCached serverlerine dağıtılır. MemCached serverlerinden birisi devre dışı kaldığında Client API durumun farkına vararak, devre dışı kalan MemCached serveri kullanımdan alır. Belirli aralıklarla devre dışı kalan MemCached serverin durumu kontrol edilir. Eğer MemCached server tekrar aktif hale geldiyse, global caching sistemi bünyesinde tekrar kullanılır hale getirilir. Bunlarında hepsi caching sistemini kullanan program için transparandir. Global caching sistemi bünyesinde sadece bir tane MemCached serveri kalmış olsa bile, caching sistemi bir global caching sistemi gibi çalışmaya devam eder.

Eğer bana hangi caching sisteminin kullanılmasını tavsiye ettiğim sorulacak olursa, cevabim MemCached olacaktır. BizimAlem bünyesinde yıllardan beri güvenli caching sistem olarak MemCached serverlerini kullanıyoruz ve değiştirmeyi de düşünmüyoruz ☺