NCache
Distributed Cache for .NET

Whitepaper

4 manières d'améliorer les performances ASP.NET

Lors des piques d'activité

Par

Iqbal Khan

June 2015

Sommaire

Introduction

ASP.NET est de plus en plus populaire pour développer des applications web et beaucoup de ces applications doivent supporter beaucoup de trafic avec des millions d'utilisateurs. Par conséquent, ces applications ont un très fort impact sur le business.

L'architecture des applications ASP.NET est très scalable au niveau du Tiers application, et le fait que le protocole HTTP soit stateless font que vous pouvez exécuter vos applications ASP.NET à l'intérieur d'une ferme de serveur web et l'utilisation d'un load-balancer où chaque requête HTTP est dirigé vers le serveur web le moins chargé. Ceci vous permet de facilement ajouter des serveurs web à votre ferme au fur et à mesure que le trafic des utilisateurs augmente. Et ceci fait que le Tiers applicatif ASP.NET est extrêmement scalable. Mais que veux dire scalable dans ce contexte ?

La Scalabilité est essentiellement la capacité à fournir une forte performance même lorsqu'il y a un pique d'activité fort. Donc, si le temps de réponse d'une page de votre application ASP.NET est très rapide avec 10 utilisateurs et qu'elle reste rapide avec 100 000 utilisateurs, alors votre application ASP.NET est scalable. Mais, si le temps de réponse de votre page ASP.NET diminue au fur et à mesure que vous augmentez le nombre d'utilisateurs, alors votre application n'est pas scalable.

Le Problème: les goulots d'étranglement

Malgré une architecture très scalable les applications ASP.NET d'aujourd'hui font face à de réels goulots d'étranglement. Ceux-ci se trouvent dans quatre éléments différents:

1. Base de données applicative
2. ASP.NET « Session State Storage »
3. ASP.NET « View State »
4. Exécution des pages ASP.NET via la technologie « Static Output »

Voici une explication de ces quatre éléments.

Base de données applicative

Les bases de données comme Microsoft SQL Serveur ou Oracle deviennent rapidement des goulots d'étranglement au fur et à mesure que le nombre de transactions augmente. La raison est que même si vous achetez un serveur de base de données plus puissant pour essayer de supprimer ce goulot d'étranglement vous ne pourrez jamais rajouter des serveurs de base de données supplémentaires. Par exemple, il est assez courant de voir des applications web avec entre 10 et 20 serveurs web, mais vous ne pouvez pas faire de même du côté des bases de données.

ASP.NET « Session State Storage »

De plus, les « Session State » d'ASP.NET ont besoin d'être stockés quelque part. Et par défaut, les options définies par Microsoft sont « InProc », « StateServer », et « SqlServer ». Malheureusement, toutes ces options ont un problème majeur de performance et de scalabilité.

« InProc » et « StateServer » forcent l'utilisation systématique du serveur web sur lequel la session a été créée la première fois (c'est ce que l'on nomme « sticky sessions » et par conséquent toutes les requêtes HTTP pour cette session devront toujours utiliser le même serveur Web ce qui diminue l'utilité du load balancer. Si vous utilisez « StateServer » sur un serveur dédié pour éviter les « sticky sessions », alors « StateServer » devient le talon

d'Achille et peut devenir un goulot d'étranglement. En plus MS Sql Serveur enregistre les sessions ASP.NET dans la base comme des BLOBs, ce qui a un sérieux impact sur les performances et la scalabilité.

ASP.NET « View State »

ASP.NET View State est une chaîne de caractères cachée et encodée (souvent d'une taille d'environ 100Ko) qui est envoyée vers le navigateur internet de l'utilisateur qui fait partie de la réponse HTTP. Le navigateur n'en fait rien et la retourne au serveur web dans le cas d'un Post-Back HTTP. Ceci ralentit la réponse de la page ASP.NET, rajoute un trafic inutile et encombre la carte réseau avec un surcroît de trafic, et rajoute une sur consommation de bande passante dans les deux sens. Et, nous avons tous que la bande passante à un coût non négligeable et encore plus sur le cloud.

Exécution des pages ASP.NET via la technologie « Static Output »

Finalement, le Framework ASP.NET exécute une page ASP.NET suite à une requête utilisateur même si cette page ne change pas depuis la dernière requête. Ceci peut être acceptable dans un environnement avec de faibles volumes de transaction. Mais dans un environnement avec un grand nombre de transactions où les ressources sont déjà menées à rude épreuve, ces exécutions inutiles deviennent coûteuses et deviennent aussi un goulot d'étranglement.

Comme on le dit souvent, « la force d'une chaine est limitée à celle de son maillon faible ». Donc, tant que vous avez des goulots d'étranglement et des points faibles dans votre architecture d'ASP.NET, la totalité de votre application devient moins performante au fur et à mesure que le trafic augmente et peux même mettre KO votre système.

C'est assez ironique, car cela arrive très souvent quand le trafic est important, c'est-à-dire quand votre activité business est la plus importante. Par conséquent, l'impact du ralentissement du système voir son arrêt est le plus coûteux pour votre business.

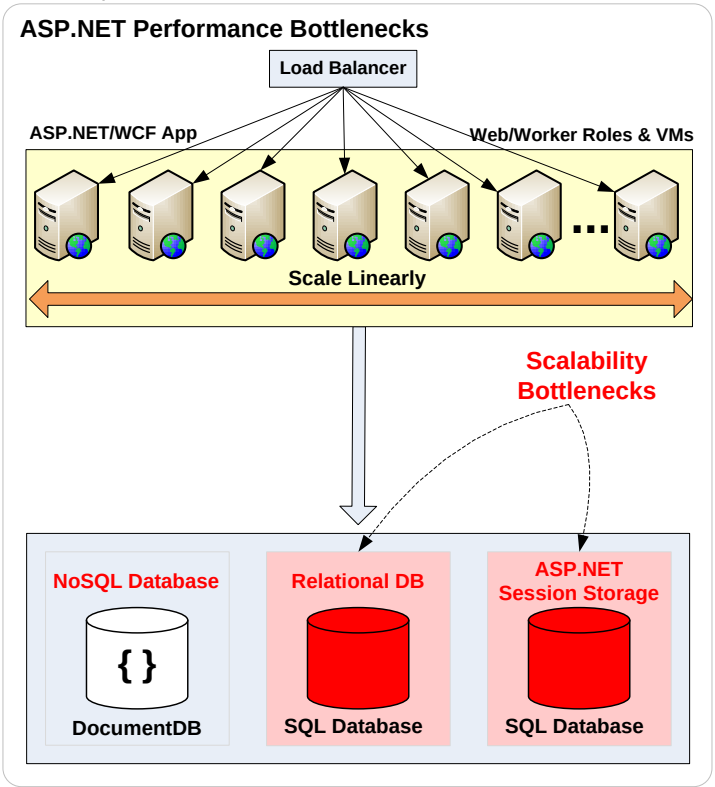

Figure 1: ASP.NET face aux goulots d'étranglement

Les bases de données NoSQL ne sont pas la solution

Le mouvement des bases de données NoSQL a démarré suite aux problèmes de scalabilité dans les bases de données relationnelles. Les bases NoSQL partitionnent les données sur plusieurs serveurs et vous permettent d'avoir une scalabilité similaire à celle du tiers applicatif.

Mais les bases NoSQL requirent que vous abandonniez votre base de données relationnelle et que vous mettiez vos données dans une base NoSQL. C'est assez facile à dire, mais en pratique il y a beaucoup de raisons pour que cela soit difficile à voir impossible dans certains cas.

Les bases NoSQL n'ont pas les mêmes outils avancés de gestion des données et de recherche que les bases de données relationnelles et nécessitent de stocker les données d'une manière totalement différente. En plus, l'écosystème interne autour des bases de données relationnelles est trop important pour pouvoir les abandonner.

Par conséquent, les bases NoSQL sont utiles seulement quand vous travaillez avec des données non structurées ou sur des projets complètement nouveaux avec un besoin qui le justifie. La plupart des applications ASP.NET travaillent essentiellement sur des données structurées qui conviennent à des bases de données relationnelles. Par conséquent, ces données ne peuvent pas être migrées vers des bases NoSQL facilement.

Et même ceux qui finissent par utiliser des bases NoSQL le font souvent pour une petite sous partie de leurs données. Ils utilisent les bases NoSQL en conjonction avec leurs bases de données relationnelles.

Ainsi, dans la majorité des cas vous devrez vivre avec vos bases de données relationnelles et trouver une autre solution pour résoudre vos goulots d'étranglement. Heureusement, il existe une solution viable qui a fait ses preuves et c'est ce que nous allons discuter maintenant.

The Solution: Le Cache distribué en mémoire

La solution à tous les problèmes que nous avons passés en revue plus haut est d'utiliser un Cache distribué en mémoire (« In-Memory Distributed Cache ») dans le déploiement de vos applications par exemple NCache. NCache est une solution Open Source de cache distribué pour .NET qui est extrêmement performante et qui permet une scalabilité linéaire. Pensez-y comme une base de données objet distribué en mémoire. Le fait de stocker les données en mémoire permet d'obtenir la performance maximum et le fait d'être distribué permet la scalabilité linéaire.

Le bon côté des solutions de cache en mémoire distribué comme NCache est que cela vous permet ne nécessite pas d'arrêter d'utiliser les bases de données relationnelles. Vous pouvez utiliser la solution de cache par-dessus vos bases de données relationnelles, car le cache supprime tous les goulots d'étranglement qui leur sont liés.

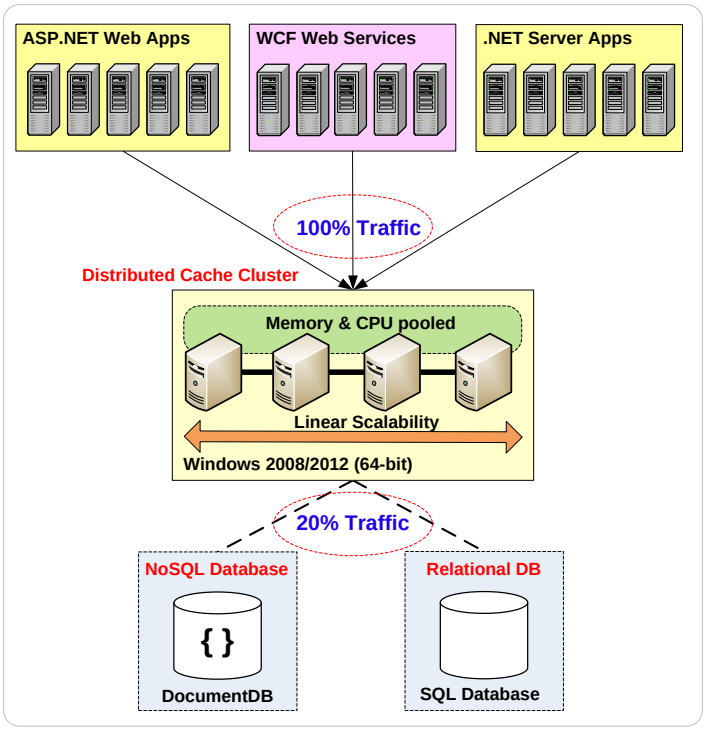

Figure 2: NCache apporte une scalabilité linéaire

Comment une solution utilisant un cache distribué en mémoire est-elle scalable qu'une solution sans? En fait, NCache crée un cluster de serveur de cache et regroupe ensemble dans un pool leur CPU, leur mémoire et toutes les autres ressources de ces serveurs. NCache vous permet aussi d'ajouter des serveurs de cache à chaud sans arrêter le cache ni l'application. Ceci vous permet de monter en puissance votre application de manière linéaire et de traiter des trafics extrêmes. Il est impossible de faire cela avec une base relationnelle.

Les caches distribués en mémoire comme NCache peuvent avoir une scalabilité linéaire en vous permettant d'ajouter des serveurs de cache à chaud. Mais, quel type de performance devriez-vous attendre d'une solution comme NCache.

Ci-dessous les performances type obtenues via NCache. Vous pouvez voir l'ensemble des résultats avec des explications sur comment elles ont été obtenues ici : [NCache performance benchmark]

Taille du cluster	Lectures / Seconde	Ecritures / Seconde
2 serveurs	50,000	32,000
3-node cluster	74,000	48,000
4-node cluster	98,000	64,000
5-node cluster	122,000	80,000
6-node cluster	146,000	96,000

Figure 3: Performance avec NCache

Comme on peut le voir, un cache distribué en mémoire comme NCache permet des performances en dessous de la milliseconde pour la lecture et l'écriture et vous permet ainsi d'avoir une application scalable de manière linéaire simplement en ajoutant des serveurs de cache.

Voyons maintenant comment un cache distribué en mémoire comme NCache peut résoudre différents goulots d'étranglement mentionnés ci-dessus.

Cache des données applicatives

L'utilisation du cache pour les données vous permet de supprimer les goulots d'étranglement sur la base de données. Un cache distribué en mémoire comme NCache vous permet de cacher les données de l'application et de réduire ces allers et retours coûteux sur la base de données. Vous pouvez espérer supprimer entre 70 et 90% du trafic sur la base de données grâce au cache distribué en mémoire. Ceci réduit la pression sur vos bases de données et leur permet d'être plus performantes et de traiter plus de transactions sans ralentir.

```
Customer Load(string customerId)
{
   // Key format: Customer:PK:1000
   string key = "Customers:CustomerID:" + customerId;

   Customer cust = (Customer) _cache[key];

   if (cust == null)
   {  // Item not in cache so load from db
       LoadCustomerFromDb(cust);

       // Add item to cache for future reference
       _cache.Insert(key, cust);
   }
   return cust;
}
```

Figure 4: Utilisation du cache distribué pour cacher les données de l'application

L'utilisation du cache avec les données applicatives vous permet d'utiliser le cache pour stocker n'importe quelle donnée de votre base de données relationnelle dans le cache. C'est souvent sous la forme de d'objet business (Entité). Dans cet exemple nous montrons comment on utilise NCache pour mettre un objet client dans le cache.

Cache des ASP.NET « Session State »

Les caches distribués en mémoire comme NCache est aussi un excellent moyen de stocker centralement vos ASP.NET « Session State ». C'est beaucoup plus rapide et plus scalable que les trois options mentionnées ci-dessus (InProc, StateServer, et Sql Serveur). NCache est plus rapide, car tout est stocké directement en mémoire (aucun accès disque) et permet d'accéder aux objets sous forme de clé / valeur, la valeur étant l'objet ASP.NET Session State. Ceci permet une scalabilité pour la gestion des sessions, car le cache est distribué.

NCache réplique aussi de manière intelligente les sessions grâce à ses différentes topologies. De cette manière si un serveur de cache s'arrête il n'y a pas de perte de données. Cette réplication est nécessaire, car Ncache ne sauvegarde pas les données sur disque, donc la duplication permet de supprimer la volatilité de la mémoire.

NCache accélère aussi la sérialisation des objets « Session State » ASP.NET qui est nécessaire avant leur stockage en dehors du Process. NCache effectue cette opération en utilisant sa fonctionnalité « Dynamic Compact Serialization » qui prend 10 fois moins de temps que la sérialisation normale d'ASP.NET. Vous pouvez utiliser cette fonctionnalité sans modification de votre code.

Vous pouvez utiliser le plug-in NCache Session State Provider (SSP) dans votre application ASP.NET en modifiant quelques lignes de configuration dans votre fichier web.config comme décrit ci-après.


```
<system.web>
   ...
   <assemblies>
      <add assembly="Alachisoft.NCache.SessionStoreProvider, Version=4.3.0.0,
                  Culture=neutral, PublicKeyToken=CFF5926ED6A53769" />
   </assemblies>

   <sessionState    cookieless="false" regenerateExpiredSessionId="true"
                    mode="Custom" customProvider="NCacheSessionProvider"
             timeout="20">
      <providers>
         <add name="NCacheSessionProvider"
             type="Alachisoft.NCache.Web.SessionState.NSessionStoreProvider"
             useInProc="false" cacheName="myDistributedCache"
             enableLogs="false“ writeExceptionsToEventLog="false” />
      </providers>
   </sessionState>
   ...
</system.web>
```

Figure 5: Plug-in NCache ASP.NET Session State Provider (SSP) dans Web.Config

Cache des ASP.NET « View State »

Nous avons déjà décrit comment les chaînes de caractères des « View State » d'ASP.NET sont encodés et envoyé par le serveur web vers le navigateur internet de l'utilisateur qui renvoie en retour au serveur web dans le cas d'un HTTP Post Back. Mais, avec l'aide d'un cache distribué en mémoire comme NCache, vous pouvez mettre dans le cache ces « ASP.NET View State » sur le serveur de cache et renvoyer sur le serveur la clé unique qui lui correspond à la place de la longue chaîne de caractères.

NCache a implémenté un plug-in appelé ASP.NET View State caching module via un adaptateur custom de page ASP.NET et d'un « ASP.NET PageStatePersister ». De cette manière, NCache intercepte la requête HTTP ainsi que sa réponse. Au moment de l'envoi de la réponse HTTP, NCache extrait la portion "valeur" de la chaîne de caractère encodée et la met dans le cache tout en la remplaçant par la clé unique qui l'identifie dans le cache.

Puis quand, la requête HTTP suivante arrive, Ncache l'intercepte de nouveau et remplace la clé unique par la valeur correspondante mise dans le cache précédemment. De cette manière, la page ASP.NET ne voit aucune différence et utilise la chaîne de caractère encodée tout à fait normalement.

L'exemple suivant montre la portion « ASP.NET View State » encodée sans faire appel à Ncache, et en utilisant NCache.

ASP.NET View State sans l'utilisation du Cache

```
<input id="__VIEWSTATE"
       type="hidden"
       name="__VIEWSTATE"
       value="/wEPDwUJNzg0MDMxMDA1D2QWAmYPZBYCZg9kFgQCAQ9kFgICBQ9kFgJmD2QWAgIBD
              xYCHhNQcm2aW91c0NvbnRyb2xNb2RlCymIAU1pY3Jvc29mdC5TaGFyZVBvaW50Lld
              lYkNvbnRyb2xzLlNQQ29udHJbE1vZDA1XzRlMjJfODM3Y19kOWQ1ZTc2YmY1M2IPD
              xYCHhNQcm2aW91c0NvbnRyb2xNb2RlCymIAU1pY3Jvc29mdC5TaGFyZVBvaW50Lld
              lYkNvbnRyb2xzLlNQQ29udHJbE1vZDA1XzRlMjJfODM3Y19kOWQ1ZTc2YmY1M2IPD
              ... ==" />
```

ASP.NET View State avec l'utilisation du cache

```
<input id="__VIEWSTATE"
       type="hidden"
       name="__VIEWSTATE"
       value="vs:cf8c8d3927ad4c1a84da7f891bb89185" />
```

Figure 6: ASP.NET View State Encodé sans et avec l'utilisation du Cache

ASP.NET « Output Cache » pour les « Static Page Output »

ASP.NET propose le « ASP.NET Output Cache Framework » pour résoudre le problème du traitement excessif lors de la création du rendu des pages alors qu'elles n'ont pas changée. Ce Framework vous permet de mettre en cache le résultat d'une page entière page ou des parties de la page de telle manière que le prochain appel à cette page, elle ne soit pas exécutée, mais que son image dans le cache soit utilisée à la place. Afficher une page qui est déjà dans le « Output Cache » est bien plus rapide que de l'exécuter de nouveau.

```
<caching>
  <outputCache defaultProvider ="NOutputCacheProvider">
    <providers>
      <add name="NOutputCacheProvider"
       type="Alachisoft.NCache.OutputCacheProvider.NOutputCacheProvider,
             Alachisoft.NCache.OutputCacheProvider, Version=x.x.x.x,
             Culture=neutral, PublicKeyToken=1448e8d1123e9096"

             cacheName="myDistributedCache" exceptionsEnabled="false"
             writeExceptionsToEventLog="false" enableLogs="true" />"
    </providers>
  </outputCache>
</caching>
```

Figure 7: Configurer le « ASP.NET Output Cache Provider » pour NCache dans Web.Config

NCache a implémenté un ASP.NET « Output Cache Provider » pour .NET 4.0 et versions ultérieures. Ceci vous permet de connecter NCache facilement sans effort de programmation. Avec NCache, ce « provider » est pour accéder au cache distribué en mémoire et permet d'accéder à de multiples serveurs de manière transparente. Donc si votre application ASP.NET est installée sur une ferme de serveurs web utilisant un load balancer, le « page output » mis dans le cache par le serveur 1 est immédiatement disponible depuis tous les autres serveurs de la Ferme Web. Vous pouvez voir ci-dessus comment configurer le plug in NCache Output Cache Provider pour ASP.NET.

L'architecture des caches distribués

Les applications ASP.NET avec un trafic important ne peuvent pas se permettre de « tomber » spécialement pendant une forte activité. Pour ces types d'applications, il y a trois buts importants qu'une solution bonne de cache distribué en mémoire tel que NCache doit offrir:

1. Haute disponibilité
2. Scalabilité Linéaire
3. Réplication des données et fiabilité

Passons en revue ces éléments.

Haute disponibilité

Un des objectifs architecturaux le plus importants de NCache est d'apporter une haute disponibilité et une élasticité du système de Cache. Ceci est fait grâce aux capacités de l'architecture suivantes :

1. **Cluster en peer-2-peer auto géré:** NCache construit un cluster de serveurs de cache sur TCP/IP. Ce cluster a une architecture peer-2-peer ce qui veut dire qu'il n'y a pas de serveur maître ni esclave. A la place, chaque nœud est égal aux les autres. Ceci permet à NCache de gérer des situations où n'importe quel nœud peut s'arrêter et le cluster s'ajustera automatiquement et continuera de fonctionner sans aucune interruption pour l'application.
2. **Configuration dynamique:** Ceci signifie que vous n'avez pas besoin de figer vos fichiers de configurations. Ceci est réalisable, car NCache propage la plupart des informations de configuration vers les clients pendant l'exécution (runtime). Donc, quand vous ajoutez un serveur de cache en cours d'exécution, les informations sur les nœuds du cluster sont automatiquement distribuées et mises à jour sur chaque nœud du cluster et sur chaque client. Il y a beaucoup d'autres éléments de la configuration qui sont distribués dynamiquement.
3. **Support pour les pertes de connexion:** Cette fonctionnalité qui consiste à détecter qu'une connexion à un nœud n'est plus disponible, permet au reste du cluster de cache et aux clients du cache de s'adapter automatiquement à cette nouvelle configuration et d'éviter toute interruption.

Réplication des données avec Scalabilité Linéaire

Comme les caches en mémoire distribué comme NCache utilisent la mémoire pour stocker les informations, ces solutions doivent offrir une réplication des données afin d'assurer leur fiabilité. Mais dans un même temps, elles ne doivent pas compromettre la scalabilité linéaire, car c'est la raison principale d'utiliser un cache en mémoire distribué comme NCache.

Voici quelques topologies proposées par NCache qui permettent de réaliser ces deux objectifs.

1. **Cache partitionné:** NCache partitionne le cache en fonction du nombre de serveurs de cache et assigne une partition à chaque serveur. Il ajuste le nombre de partitions quand vous ajoutez ou supprimez un serveur dans le cluster de manière dynamique. Partitionner est la principale manière d'assurer une scalabilité linéaire, car quand vous ajoutez de nouveaux serveurs, vous augmentez la capacité de stockage et la capacité de traitement (CPU).
2. **Cache Partitionné-Répliqué:** En plus de partitionner, NCache apporte aussi la réplication sur chaque partition. Ces réplicas résident sur des serveurs de cache différents afin de garantir une redondance physique des données en mémoire. Ceci garantit que si un serveur s'arrête les données de ce serveur

existent sur un serveur différent. Dans ce cas les données répliquées deviennent disponible immédiatement en remplacement du réplica indisponible. En ne répliquant qu'une seule fois les partitions sur un autre serveur, NCache garantit une fiabilité des données sans compromettre la scalabilité linéaire.

3. **Cache Client (Near Cache):** Une autre fonctionnalité très importante de NCache est le Cache Client. C'est un cache local qui se trouve le serveur applicatif et qui peut même être configuré pour fonctionner en mode InProc (le cache fonctionne directement dans le processus de l'application). C'est essentiellement un cache par-dessus le cluster de cache afin d'apporter des gains de performance extrêmes tout en apportant encore plus de scalabilité à NCache en diminuant le trafic vers le cluster.

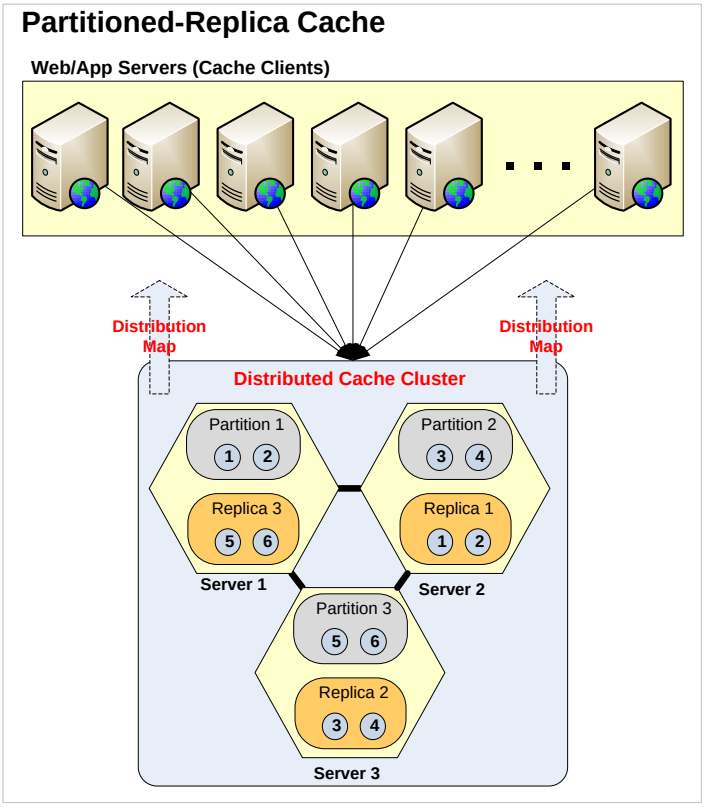

Figure 8: Topologie de cache Partitionnée-Répliquée pour NCache

Comme vous pouvez le voir, la topologie de cache Partitionnée-Répliquée met une partition et une réplication sur chaque serveur. Et s'assure que chaque réplica se trouve sur un serveur de cache différent afin de garantir la fiabilité.

Conclusion

Nous avons essayé de passer en revue les problèmes classiques de performance et les différents goulots d'étranglement présent sur une architecture ASP.NET. Et nous avons montré comment les résoudre en utilisant un cache en mémoire distribué comme NCache. NCache est une solution Open Source de cache distribué pour les applications .NET et Java. Vous pouvez trouver plus d'information à propos de NCache via les liens suivants.

NCache Details	Edition Comparison	Download NCache
NCache vs AppFabric	NCache vs Redis	NCache vs Memcached