Dynamic caching in the cloud with ReplCache

Bela Ban, JBoss

February 2009

Overview

In a previous article I wrote about an implementation of memcached on JGroups.

ReplCache is a large virtual hashmap spanning multiple nodes, similar to memcached. If we have 5 processes (nodes) with 1GB of memory allocated to each of their hashmap, then we have a virtual hashmap of 5GB. By starting another nodes, the size would increase to 6GB, and if a node leaves, the size shrinks accordingly.

When we distribute our elements (key and values) to the virtual hashmap, then – based on consistent hashing – an element gets assigned to a single node, and so we can use the full 5GB. Reads and writes are always sent to the same node using the consistent hash of the key. If the node storing element K crashes, the next read returns null, and the user has to retrieve K from the database and re-insert it into the virtual hashmap, this time picking a different node.

Distribution therefore always requires some stable storage (e.g. a DB) from which we can retrieve our elements should a node hosting an element crash. All writes are of course also written to the DB in this case.

Distribution is similar to RAID 0.

If we don't want to use a database, e.g. because it is a single point of failure, and access to it is slow, we can just keep our data in memory. However, to prevent data loss due to a node crash, we have to replicate all elements.

Replication is the opposite of distribution: we have a copy of a given element K on every node. Should a node crash, clients can simply pick a different node and K is still present. Updates to K have to be sent to all nodes. Note that to prevent a catastrophic failure in which all nodes in the cloud crash, we could still write updates to the DB, but this could be done in the background.

While replication increases availability and prevents data loss, we can now effectively only use 1GB out of the 5GB of the above example: if a node has 1GB of data, and every node replicates its data to every other node in the 5 node cluster, we'd use up the 5GB available to us.

Replication is similar to RAID 1.

So we can either choose to not replicate anything with potential data loss but maximal use of the 5GB, or replicate everything to everyone, which minimizes the risk of data loss but limits us to 1GB out of the 5GB.

RAID 5 is a solution which doesn't replicate data everywhere (it only replicates it K times where K < number of disks) and increases the use of the available size. However, K is fixed and for RAID 5 to work, we need at least 3 disk.

Enter ReplCache, which allows a developer to define how many times an element should be available in a cluster. *This is defined per data item with a replication count (K)*:

- K == -1: the element is stored on all cluster nodes (full replication)
- K == 1: the element stored on a single node only, determined through consistent hashing (distribution)
- K > 1: the element is stored K times in the cluster

If an element is important, and loss is catastrophic or recreation costly, then -1 should be chosen. If an element can easily be fetched from the database again, then 1 might be picked. A value greater than 1 decreases the risk of data loss and saves memory, e.g. if K == 3 (N == 10), then 3 cluster nodes have to crash at the same time to lose K. If they don't crash simultaneously, the cluster will rebalance its data so that K == 3 (unless N < K).

The advantage of defining K per data element is that an application can define data reliability and thus use more of the virtual memory allocated[1].

API

The API is trivial and consists of put(), get() and remove().

```
public void put(K key, V val, short repl_count, long timeout)
```

Stores a key and value in the cloud.

key	The key to insert into the cloud
val	The value to insert into the cloud. Note that key and val have to be serializable because they are sent over the network
repl_count	The number of replicas of key and val that should be in the cloud: -1: store on all nodes in the cloud 1: store on a single node in the cloud >1: store on multiple nodes
timeout (ms)	Number of milliseconds after which a key/value will be evicted if not accessed: 0: cache forever -1: don't cache at all > 0: cache for timeout ms

```
public V get(K key)
```

Returns a value associated with a given key. If the key has no associated value, null will be returned.

```
public void remove(K key)
```

1Compare this to for example RAID 1 where everything is replicated, even directories like /tmp, which are probably not needed in a crash case.

Removes a key and value from the cloud.

Maintaining replicas

When new nodes are added to the cluster, or removed from the cluster, ReplCache has to make sure that elements with K == 1 are still stored on the correct node, based on the consistent hash and the new cluster topology.

Also, with elements that are stored multiple times in the cluster (K > 1), ReplCache has to make sure that elements are moved or copied to other nodes. For example, if id=322649 has K == 3 and the cluster has nodes A and B, 'id' will be stored on A and B. As soon as C is added, we also have to copy 'id' to C in order to have 3 copies available. When D is added, we have to figure out whether to store 'id' on D, too. If that's the case, 'id' can be removed on either A, B or C.

To handle the latter case (K > 1), we compute K hash codes for the changed cluster topology and pick K nodes (NEW-NODES). Then we compute K hash codes for the old cluster topology and pick K nodes (OLD-NODES). If NEW-NODES is the same as OLD-NODES, we don't need to rebalance a given element. If not, we multicast a PUT, which every node receives. Every node then checks whether it is in the NEW-NODES set for the given key and applies the PUT if so, or discards it if not.

Conclusion

ReplCache is a clustered cache, spanning multiple nodes.

Elements can be distributed and optionally replicated to optimimally store them in the cloud, giving applications control over space versus reliability tradeoffs by defining replication counts for individual elements.

We're looking into defining regions of keys, where sets of related keys are co-located on the same nodes. This would require the addition of a parameter 'region', which is then used as input to the consistent hash function (rather than using the key directly). An example that would benefit from this is HTTP sessions: all attributes of a given session would be co-located.

Links

[1] ReplCache: http://www.jgroups.org/javagroupsnew/docs/replcache.html

[2] JGroups: http://www.jgroups.org

[3] Memcached article: http://www.jgroups.org/javagroupsnew/docs/memcached.html