In this tutorial, you learn how to deploy a cluster of distributed Memcached (https://memcached.org/) servers on Google Kubernetes Engine (GKE (/kubernetes-engine/) using Kubernetes (https://kubernetes.io/), Helm (https://helm.sh), and Mcrouter (https://github.com/facebook/mcrouter). Memcached is a popular open source, multi-purpose caching system. It usually serves as a temporary store for frequently used data to speed up web applications and lighten database loads.

Memcached has two main design goals:

- **Simplicity**: Memcached functions like a large hash table (https://wikipedia.org/wiki/Hash_table) and offers a simple API to store and retrieve arbitrarily shaped objects by key.
- **Speed**: Memcached holds cache data exclusively in random-access memory (RAM), making data access extremely fast.

Memcached is a distributed system that allows its hash table's capacity to scale horizontally across a pool of servers. Each Memcached server operates in complete isolation from the other servers in the pool. Therefore, the routing and load balancing between the servers must be done at the client level. Memcached clients apply a consistent hashing (https://wikipedia.org/wiki/Consistent_hashing) scheme to appropriately select the target servers. This scheme guarantees the following conditions:

- The same server is always selected for the same key.
- Memory usage is evenly balanced between the servers.
- A minimum number of keys are relocated when the pool of servers is reduced or expanded.

The following diagram illustrates at a high level the interaction between a Memcached client and a distributed pool of Memcached servers.

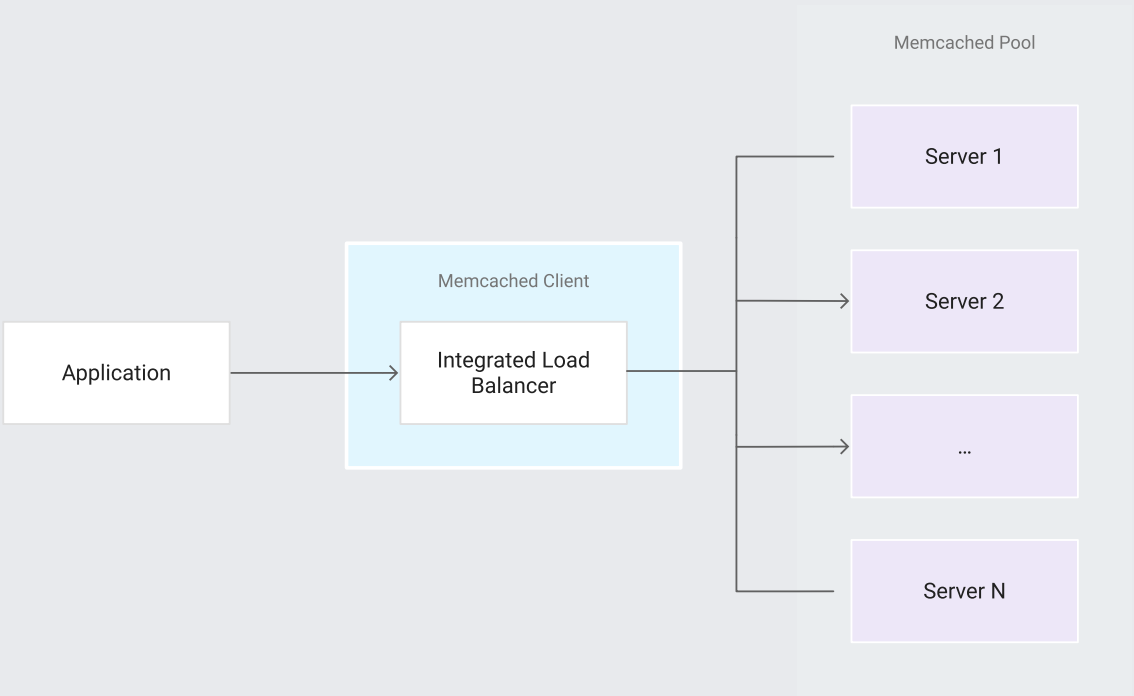

Figure 1: High-level interaction between a Memcached client and a distributed pool of Memcached servers.

- Learn about some characteristics of Memcached's distributed architecture.
- Deploy a Memcached service to GKE using Kubernetes and Helm.
- Deploy Mcrouter, an open source Memcached proxy, to improve the system's performance.

This tutorial uses the following billable components of Google Cloud:

- Compute Engine

To generate a cost estimate based on your projected usage, use the pricing calculator (/products/calculator). New Google Cloud users might be eligible for a free trial (/free-trial).

1. Sign in (https://accounts.google.com/Login) to your Google Account.

   If you don't already have one, sign up for a new account (https://accounts.google.com/SignUp).

2. In the Cloud Console, on the project selector page, select or create a Cloud project.

★ **Note**: If you don't plan to keep the resources that you create in this procedure, create a project instead of selecting an existing project. After you finish these steps, you can delete the project, removing all resources associated with the project.

Go to the project selector page (https://console.cloud.google.com/projectselector2/home/dashboard)

3. Make sure that billing is enabled for your Google Cloud project. Learn how to confirm billing is enabled for your project (/billing/docs/how-to/modify-project).

4. Enable the Compute Engine API.

   Enable the API (https://console.cloud.google.com/flows/enableapi?apiid=compute_component)

5. Start a Cloud Shell instance.

   OPEN Cloud Shell (https://console.cloud.google.com/?cloudshell=true)

One simple way to deploy a Memcached service to GKE is to use a Helm (https://helm.sh) chart. To proceed with the deployment, follow these steps in Cloud Shell:

1. Create a new GKE cluster of three nodes:

★ **Note:** The cluster's zone specified here is arbitrary for the purposes of this tutorial. You can select another zone for your cluster from the available zones (/compute/docs/regions-zones/regions-zones#available).

2. Download the `helm` binary archive:

3. Unzip the archive file to your local system:

4. Add the `helm` binary's directory to your `PATH` environment variable:

   This command makes the `helm` binary discoverable from any directory during the current Cloud Shell session. To make this configuration persist across multiple sessions, add the command to your Cloud Shell user's `~/.bashrc` file.

5. Create a service account with the cluster admin role for Tiller, the Helm server:

6. Initialize Tiller in your cluster and update information of available charts:

7. Install a new Memcached Helm chart (https://github.com/kubernetes/charts/tree/master/stable/memcached) release with three replicas, one for each node:

The Memcached Helm chart uses a StatefulSet controller (https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/). One benefit of using a StatefulSet controller is that the pods' names are ordered and predictable. In this case, the names are `mycache-memcached-{0..2}`. This ordering makes it easier for Memcached clients to reference the servers.

8. To see the running pods, run the following command:

The Google Cloud Console output looks like this:

The Memcached Helm chart uses a headless service (https://kubernetes.io/docs/concepts/services-networking/service/#headless-services). A headless service exposes IP addresses for all of its pods so that they can be individually discovered.

1. Verify that the deployed service is headless:

The output `None` confirms that the service has no `clusterIP` and that it is therefore headless.

The service creates a DNS record for a hostname of the form:

In this tutorial, the service name is `mycache-memcached`. Because a namespace was not explicitly defined, the default namespace is used, and therefore the entire host name is `mycache-memcached.default.svc.cluster.local`. This hostname resolves to a set of IP addresses and domains for all three pods exposed by the service. If, in the future, some pods get added to the pool, or old ones get removed, **kube-dns** (https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/) will automatically update the DNS record.

It is the client's responsibility to discover the Memcached service endpoints, as described in the next steps.

2. Retrieve the endpoints' IP addresses:

   The output is similar to the following:

   Notice that each Memcached pod has a separate IP address, respectively `10.36.0.32`, `10.36.0.33`, and `10.36.1.25`. These IP addresses might differ for your own server instances. Each pod listens to port `11211`, which is Memcached's default port.

3. For an alternative to step 2, retrieve those same records using a standard DNS query with the `nslookup` command:

   The output is similar to the following:

Notice that each server has its own domain name of the following form:

For example, the domain for the `mycache-memcached-0` pod is:

4. For another alternative to step 2, perform the same DNS inspection by using a programming language like Python:

    a. Start a Python interactive console inside your cluster:

    b. In the Python console, run these commands:

    The output is similar to the following:

5. Test the deployment by opening a `telnet` session with one of the running Memcached servers on port `11211`:

    At the `telnet` prompt, run these commands using the Memcached ASCII protocol (https://github.com/memcached/memcached/blob/master/doc/protocol.txt):

The resulting output is shown here in bold:

You are now ready to implement the basic service discovery logic shown in the following diagram.

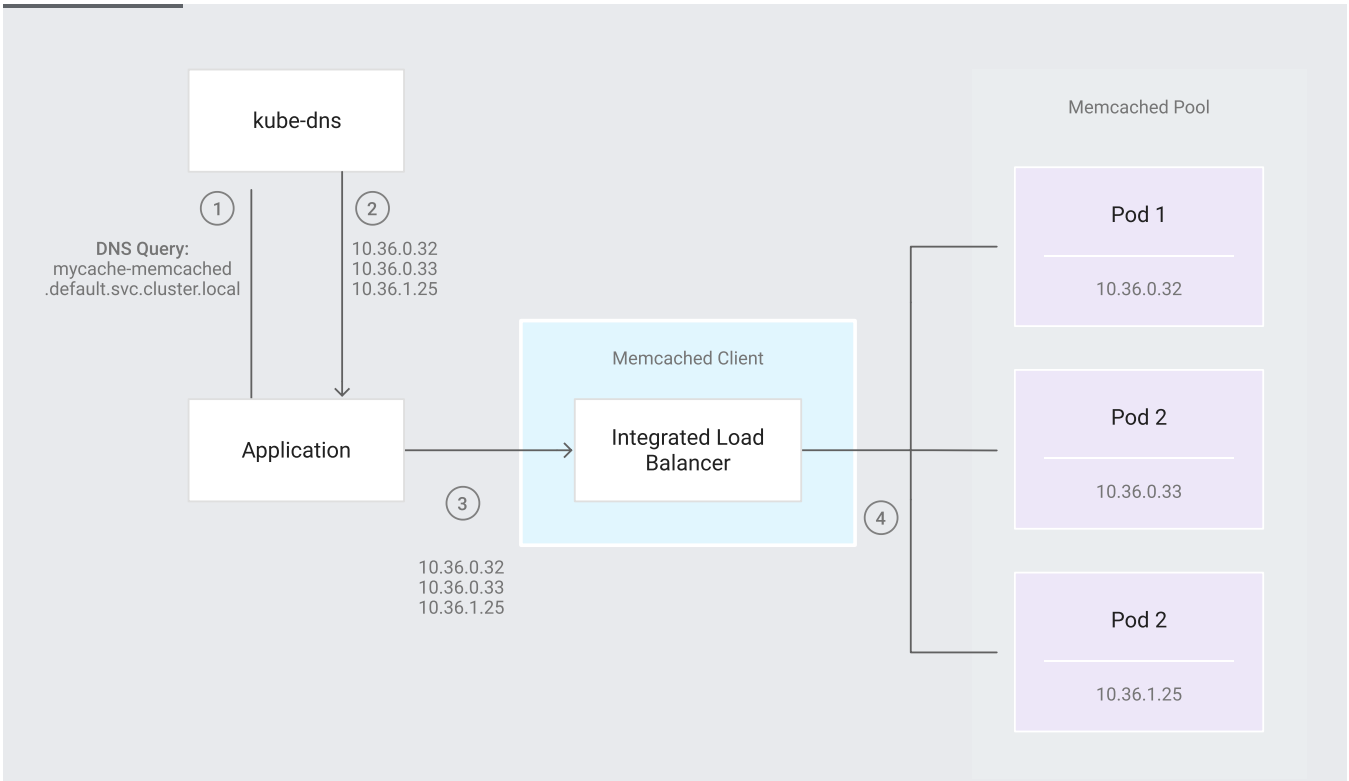

Figure 2: Service discovery logic.

At a high level, the service discovery logic consists of the following steps:

1. The application queries `kube-dns` for the DNS record of `mycache-memcached.default.svc.cluster.local`.
2. The application retrieves the IP addresses associated with that record.
3. The application instantiates a new Memcached client and provides it with the retrieved IP addresses.
4. The Memcached client's integrated load balancer connects to the Memcached servers at the given IP addresses.

You now implement this service discovery logic by using Python:

1. Deploy a new Python-enabled pod in your cluster and start a shell session inside the pod:

2. Install the `pymemcache` (https://pymemcache.readthedocs.io) library:

3. Start a Python interactive console by running the `python` command.
4. In the Python console, run these commands:

   The output is as follows:

   The `b` prefix signifies a bytes literal (https://docs.python.org/3/reference/lexical_analysis.html#string-and-bytes-literals), which is the format in which Memcached stores data.
5. Exit the Python console:
6. To exit the pod's shell session, press `Control+D`.

As your caching needs grow, and the pool scales up to dozens, hundreds, or thousands of Memcached servers, you might run into some limitations. In particular, the large number of open connections from Memcached clients might place a heavy load on the servers, as the following diagram shows.

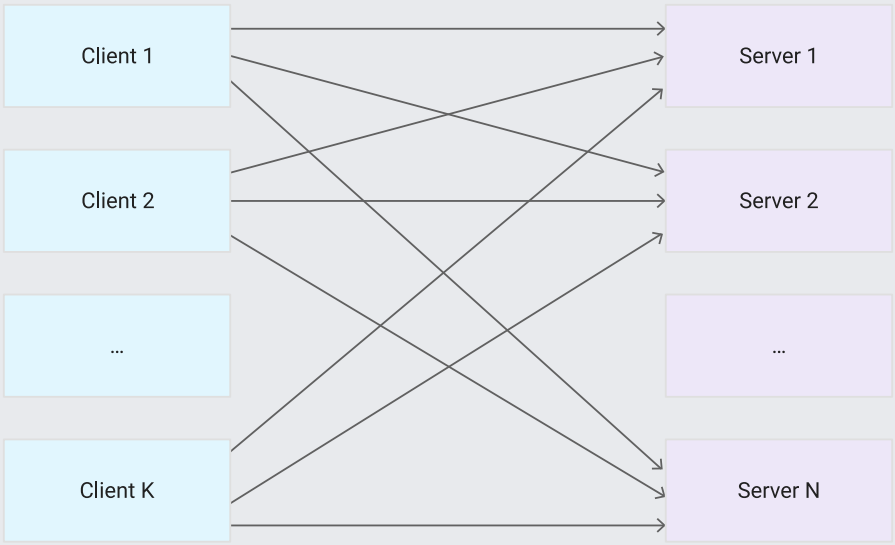

Figure 3: High number of open connections when all Memcached clients access all Memcached servers directly.

To reduce the number of open connections, you must introduce a proxy to enable connection pooling, as in the following diagram.

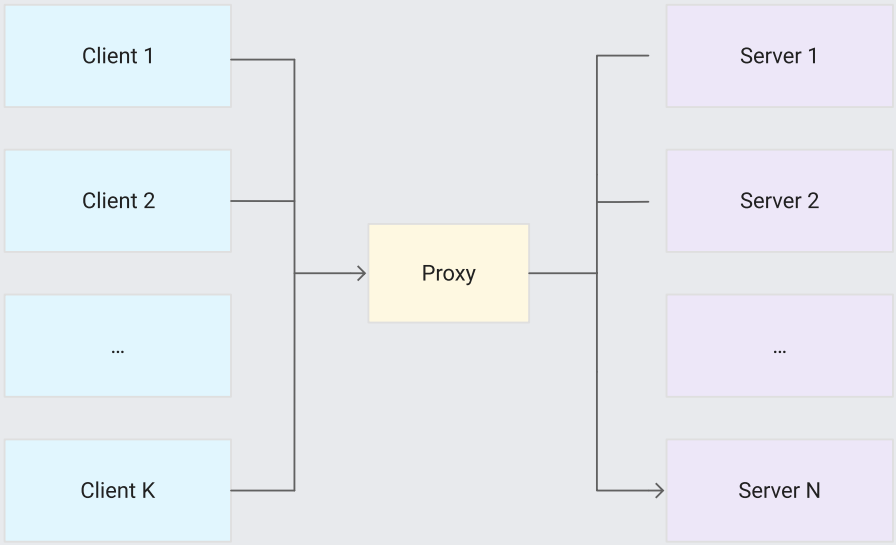

Figure 4: Using a proxy to reduce the number of open connections.

Mcrouter (https://github.com/facebook/mcrouter) (pronounced "mick router"), a powerful open source Memcached proxy, enables connection pooling. Integrating Mcrouter is seamless, because it uses the standard Memcached ASCII protocol. To a Memcached client, Mcrouter behaves like a normal Memcached server. To a Memcached server, Mcrouter behaves like a normal Memcached client.

To deploy Mcrouter, run the following commands in Cloud Shell.

1. Delete the previously installed `mycache` Helm chart release:

2. Deploy new Memcached pods and Mcrouter pods by installing a new Mcrouter Helm chart (https://github.com/kubernetes/charts/tree/master/stable/mcrouter) release:

The proxy pods are now ready to accept requests from client applications.

3. Test this setup by connecting to one of the proxy pods. Use the `telnet` command on port `5000`, which is Mcrouter's default port.

   In the `telnet` prompt, run these commands:

   The commands set and echo the value of your key.

You have now deployed a proxy that enables connection pooling.

To increase resilience, it is common practice to use a cluster with multiple nodes. This tutorial uses a cluster with three nodes. However, using multiple nodes also brings the risk of increased latency caused by heavier network traffic between nodes.

You can reduce this risk by connecting client application pods only to a Memcached proxy pod that is on the same node. The following diagram illustrates this configuration.

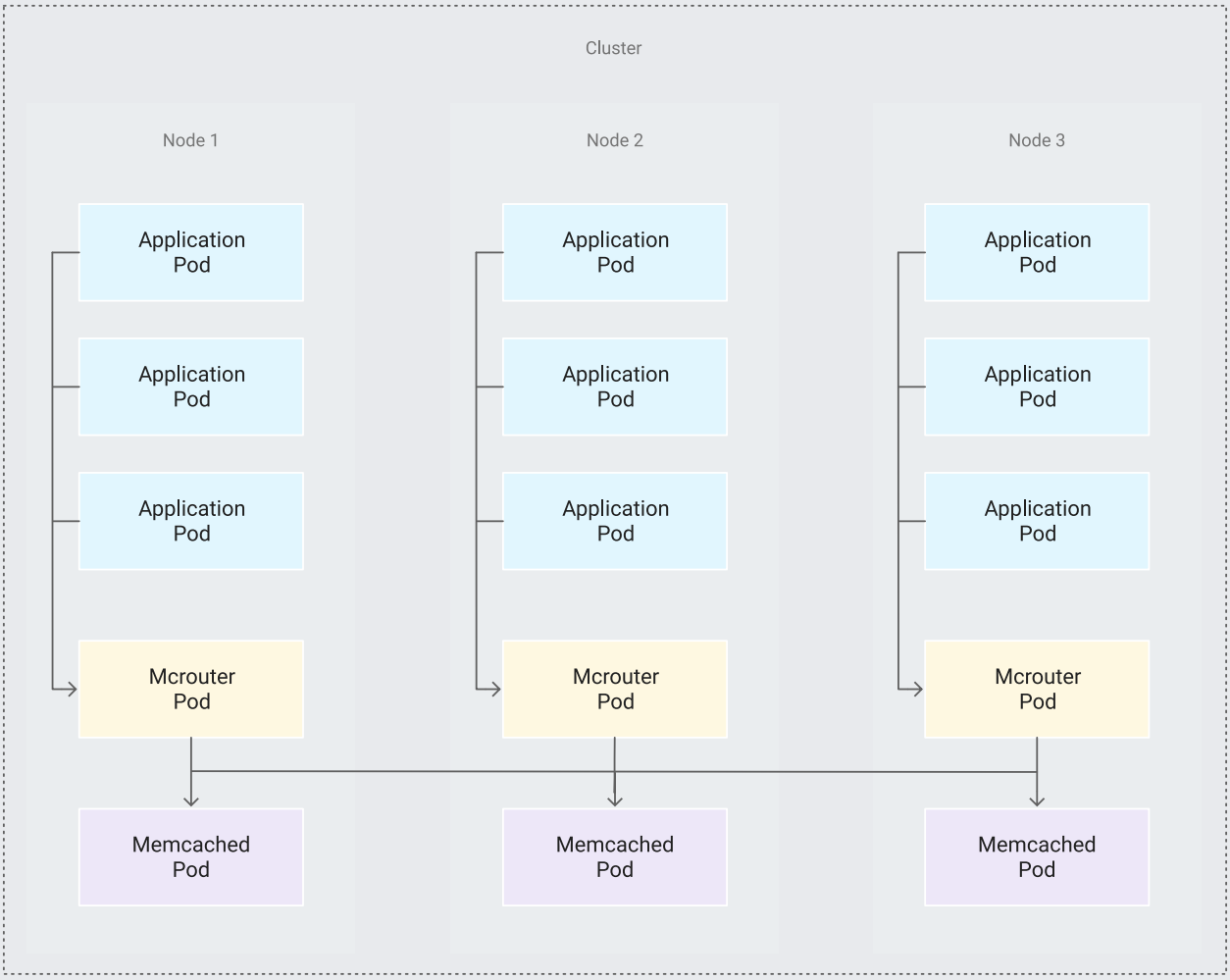

Figure 5: Topology for the interactions between application pods, Mcrouter pods, and Memcached pods across a cluster of three nodes.

Perform this configuration as follows:

1. Ensure that each node contains one running proxy pod. A common approach is to deploy the proxy pods with a DaemonSet controller (https://kubernetes.io/docs/concepts/workloads/controllers/daemonset). As nodes are added to the cluster, new proxy pods are automatically added to them. As nodes are removed from the cluster, those pods are garbage-collected. In this tutorial, the Mcrouter Helm chart that you deployed earlier uses a DaemonSet controller (https://github.com/kubernetes/charts/blob/master/stable/mcrouter/templates/daemonset.yaml) by default. So, this step is already complete.
2. Set a `hostPort` value in the proxy container's Kubernetes parameters to make the node listen to that port and redirect traffic to the proxy. In this tutorial, the Mcrouter Helm chart uses this parameter by default for port `5000`. So this step is also already complete.

3. Expose the node name as an environment variable inside the application pods by using the `spec.env` entry and selecting the `spec.nodeName` `fieldRef` value. See more about this method in the Kubernetes documentation (https://kubernetes.io/docs/tasks/inject-data-application/environment-variable-expose-pod-information/) .

    a. Deploy some sample application pods:

4. Verify that the node name is exposed, by looking inside one of the sample application pods:

    This command outputs the node's name in the following form:

The sample application pods are now ready to connect to the Mcrouter pod that runs on their respective mutual nodes at port `5000`, which is Mcrouter's default port.

1. Initiate a connection for one of the pods by opening a `telnet` session:

2. In the `telnet` prompt, run these commands:

   Resulting output:

Finally, as an illustration, the following Python code is a sample program that performs this connection by retrieving the `NODE_NAME` variable from the environment and using the `pymemcache` library:

To avoid incurring charges to your Google Cloud Platform account for the resources used in this tutorial:

1. Run the following command to delete the GKE cluster:

2. Optionally, delete the Helm binary:

- Explore the many other features (https://github.com/facebook/mcrouter/wiki#features) that Mcrouter offers beyond simple connection pooling, such as failover replicas, reliable delete streams, cold cache warmup, multi-cluster broadcast.
- Explore the source files of the Memcached chart (https://github.com/kubernetes/charts/tree/master/stable/memcached) and Mcrouter chart (https://github.com/kubernetes/charts/tree/master/stable/mcrouter) for more details on the respective Kubernetes configurations.
- Read about effective techniques (/appengine/articles/scaling/memcache) for using Memcached on App Engine. Some of them apply to other platforms, such as GKE.
- Try out other Google Cloud features for yourself. Have a look at our tutorials (/docs/tutorials).