What is an Ambassador Container?

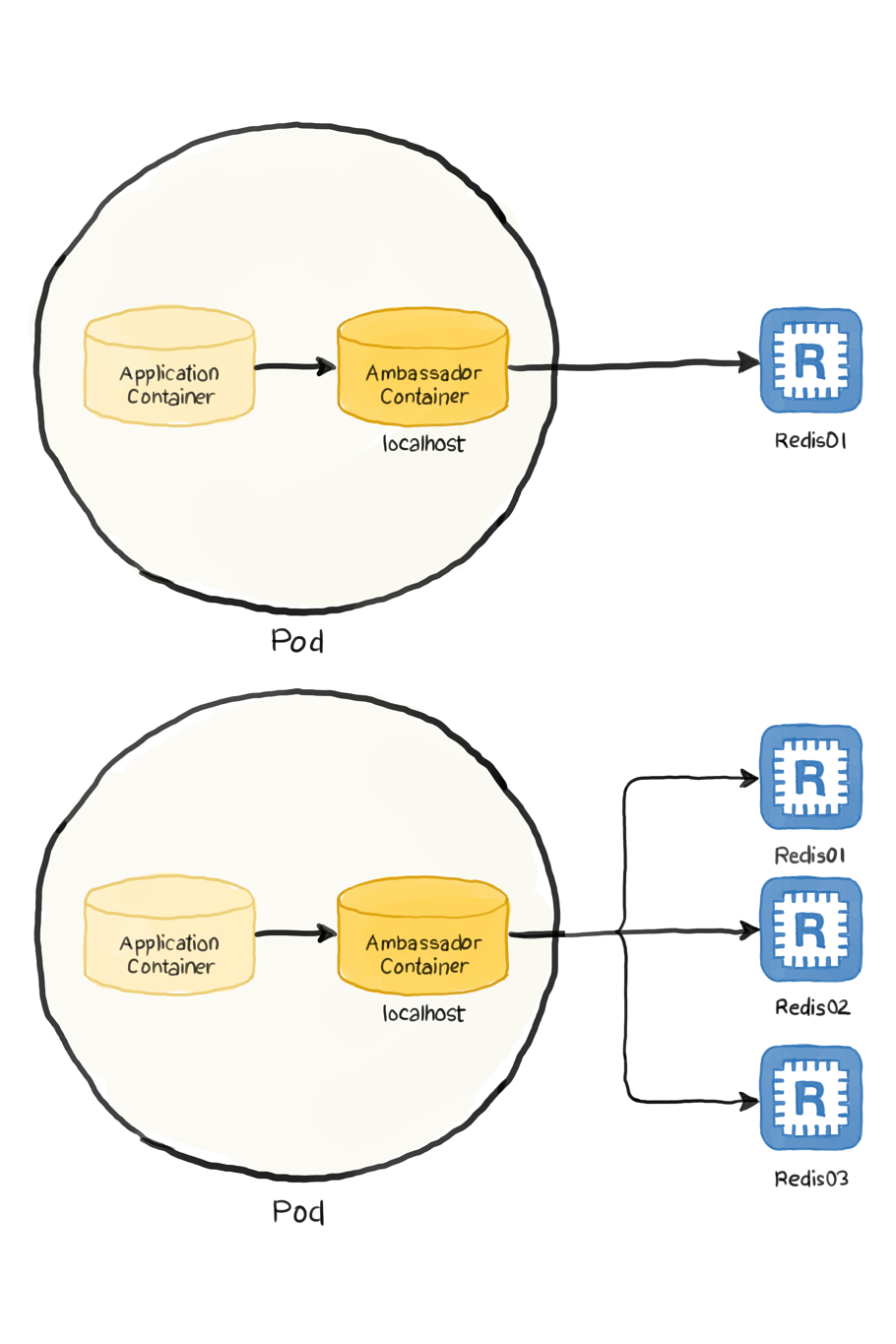
An Ambassador container is a sidecar container that is in charge of proxying connections from the application container to other services. However, while the Adapter container acts as a reverse proxy, the Ambassador container acts as a client proxy. You might be wondering, why do we need to proxy the application connection requests? Because we need to follow the separation of concerns principle. Each container should do its task and do it well. If there are other tasks that require the application’s function in order to work correctly, we may hand those tasks to the sidecar container.

For example, almost all applications need a database connection at some phase. In a multi-environment place, there would be a test database, a staging database, and a production database. When writing the Pod definition for their application’s container, developers must pay attention to which database they’ll be connecting to. A database connection string can be easily changed through an environment variable or a configMap. We could also use a sidecar pattern that proxies DB connections to the appropriate server depending on where it runs. Developers needn’t change the connection string, they could leave the DB server at localhost as usual. When deployed to a different environment, the Ambassador container detects which environment it is running on (possibly through the Reflection pattern), and connects to the correct server.

Another well-known use case for the Ambassador container is when your application needs to connect to a caching server like Memcached or Redis. Let’s have a Redis example scenario to demonstrate this pattern.

Scenario: Connecting to Multiple Redis Servers

Let’s assume that your application needs to connect to a caching server like Redis. In the development environment, Redis is installed on your laptop and listening on the localhost. In other environments, you have more than one Redis instance for high availability. You need to configure your application and select a healthy node then connect to it. One possible solution to this case is to use an Ambassador container that proxies connections to the backend Redis servers. The Ambassador container listens on the localhost as it is sharing the same Pod as the application container. In our example, we use [Twemproxy](https://github.com/twitter/twemproxy) to handle connecting to the Redis instances. The scenario can be depicted in the following illustration:



Our definition file may look as follows:

apiVersion: v1

kind: Pod

metadata:

name: ambassador-example

spec:

containers:

- name: redis-client

image: redis

- name: ambassador

image: malexer/twemproxy

env:

- name: REDIS\_SERVERS

value: redis-st-0.redis-svc.default.svc.cluster.local:6379:1,

redis-st-1.redis-svc.default.svc.cluster.local:6379:1

ports:

- containerPort: 6380

The definition defines two Pods:

* The application Pod: It is the first pod in the list that runs Redis image, but we are using it as a client to other Redis servers.
* The ambassador Pod: It is the second container that runs the [Twemproxy](https://github.com/twitter/twemproxy) as we discussed. We used the [malexer/twemproxy](https://github.com/malexer/docker-twemproxy) image, which needs the Redis instances passed as environment variables (lines 11-13) in this form address:port:weight.
* The container by default listens on port 6380 (for advanced configurations, please visit the image’s project page at [malexer/twemproxy](https://github.com/malexer/docker-twemproxy)).

Therefore, the workflow can be summarized as follows:

1. The application (simulated by the Redis client) initiates a request to localhost:6380. As far as the application is concerned, this is the Redis server.
2. The Ambassador container picks up the request and relays it to the Redis servers defined in its configuration.

The only missing thing here is the two Redis instances that the Ambassador container is configured to communicate with. The following StatefulSet definition creates them:

apiVersion: v1

kind: Service

metadata:

name: redis-svc

labels:

app: redis

spec:

ports:

- port: 6379

name: redis-port

clusterIP: None

selector:

app: redis

---

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: redis-st

spec:

serviceName: "redis-svc"

replicas: 2

selector:

matchLabels:

app: redis

template:

metadata:

labels:

app: redis

spec:

containers:

- name: redis

image: redis

ports:

- containerPort: 6379

name: redis

volumeMounts:

- name: redis-data

mountPath: /data

volumeClaimTemplates:

- metadata:

name: redis-data

spec:

accessModes: [ "ReadWriteOnce" ]

resources:

requests:

storage: 1Gi