Microsoft

Programming Microsoft Azure Service Fabric

Second Edition

Professional

Haishi Bai

Microsoft

Programming Microsoft Azure Service Fabric

Second Edition

Haishi Bai

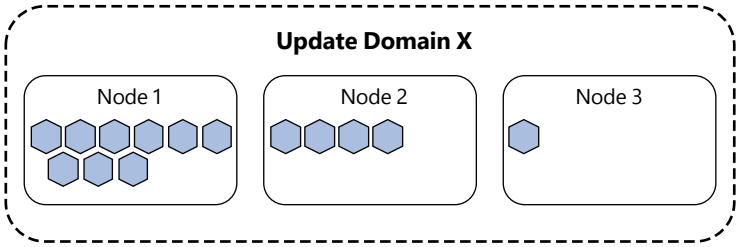

FIGURE 6-6 Service replica distribution after defragmentation.

The following cluster manifest snippet shows how to identify defragmentation metrics and define load-balancing thresholds. Note that when you identify defragmentation metrics, you must assign a Boolean value to each of the placement metrics.

```
<FabricSettings>
  <Section Name="DefragmentationMetrics">
    <Parameter Name="MyBoolean" Value="true" />
    <Parameter Name="MyValue" Value="false" />
  </Section>
  <Section Name="MetricBalancingThresholds">
    <Parameter Name="MyValue" Value="5" />
  </Section>
</FabricSettings>
```

If the load-balancing threshold is not defined for a metric, the default threshold is 1. This means defragmentation will be performed until there's at least one empty node per update domain or fault domain. If you don't want a metric to participate in the defragmentation process, set its corresponding threshold to 0.

Resource Reserves

When you design your placement strategy, leave some wiggle room to accommodate for extra demands. For example, you can allocate a node buffer to keep a percentage of nodes free for failovers. During the load-balancing process, the nodes in this buffer won't be used. However, Service Fabric can use these nodes for failovers to guarantee there will always be enough reserves for faster failover. The following cluster manifest snippet shows an example of a node buffer of 10% for a metric named `MyValue`:

```
<FabricSettings>
  <Section Name="NodeBufferPercentage">
    <Parameter Name="MyValue" Value="0.1" />
  </Section>
</FabricSettings>
```

Service Failovers

A failover may be triggered by either a software error or a hardware error. The multiple replicas of a stateless service serve as active backups for each other. When an instance fails, client requests are handled by other healthy replicas as Service Fabric tries to initiate a new service replica. For a stateful

service, *failing over* means promoting one of the active secondaries as the primary. This promotion process is usually quick and negligible to clients.

It's fairly easy to experiment with service failovers in Service Fabric Explorer. The following example walks you through a quick failover test on a local Service Fabric cluster:

1. In Visual Studio 2017, create a new Service Fabric application with a single stateless service.
2. Publish the application on the local cluster using the local publish profile, which sets the instance count to 1.
3. Open the **View** menu, choose **Other Windows**, and select **Diagnostic Event Viewer** to open the Diagnostic Event Viewer.
4. In a browser, navigate to http://localhost:19080/ to open Service Fabric Explorer.
5. In the left pane, expand the application node and observe to which node the single instance is deployed. Then expand the **Nodes** node and click the corresponding node.
6. In the right pane, click the **Actions** button and choose **Deactivate (Restart)** to simulate a node crash. Because the node is hosting the only service instance, Service Fabric needs to bring up another instance on another healthy node to bring the service back online.
7. In the left pane, observe the service instance now running on a new node.
8. When you are finished, use Service Fabric Explorer to reactivate the deactivated node.

Routing and Load-Balancing

As you've seen, a service replica may be relocated throughout its lifetime. For a client to communicate with a service hosted on Service Fabric, it must first resolve the actual node where the service replica is hosted and then make a connection. The client also needs to retry name resolutions in case the previously connected service replica has been moved. Service Fabric provides three ways for a client to resolve and connect to a service:

- Service Fabric Naming Service
- Reverse proxy
- DNS service

Service Fabric Naming Service

Service Fabric Naming Service resolves service names to specific nodes. It allows clients to address services by stable service names and dynamically routes individual requests to the appropriate nodes.

With the default configuration, a Service Fabric cluster runs Service Fabric Naming Service with three partitions to ensure the service is highly available. A service or a client can connect to any of the three primary replicas to discover the addresses of the other services. Then, as discussed in Chapter 1, the service or client establishes a direct link to the discovered node.

To use Service Fabric Naming Service, a client needs to use Service Fabric API to resolve Service Fabric addresses (such as fabric:/myApp/myService) into a specific hosting node. Chapter 2 showed how to use an SDK-supplied `ServiceProxy` class that encapsulates the naming resolution for remote `IService` interfaces. You've also seen examples of using the `IServicePartitionResolver` interface to explicitly resolve service addresses.

A client such as a browser cannot carry out this service-discovery process. This is because such a client is unaware of Service Fabric APIs. Instead, client requests are distributed to service replicas through an external load balancer. Typically, this load balancer has no knowledge of Service Fabric; it simply distributes traffic to all virtual machines that have been registered in its machine pool. This requires you to deploy an instance of your web-facing service onto each Service Fabric cluster node. (Chapter 9 discusses how to manage this on Azure using Azure Management Portal.) Furthermore, because load balancers don't know about service partitions, they can't effectively route for stateful services. To address these problems, Service Fabric offers reverse proxy.

Reverse Proxy

When enabled, reverse proxy runs on all cluster nodes to hide all name resolutions, retries, and connection-management details from HTTP clients. This allows any service on the cluster to address other HTTP services on the cluster through a localhost address. To enable reverse proxy on a cluster, simply select the Enable Reverse Proxy check box when you create the Service Fabric cluster. (Refer to Figure 6-4.)

Figure 6-7 shows how reverse proxy works. In this example, a service (fabric:/app1/service2) has three named partitions (A, B, and C), deployed to node 2, node 3, and node 4, respectively. A service on node 1 is trying to connect to partition A of the service, and a reverse proxy instance is running on node 1 and listening to port 19008. The caller simply constructs a URL in the following format:

```
http(s)://localhost:19008/<target application name>/<target service name>/<routes in the service
(in this case, API/values)>?PartitionKey=<partition key>&PartitionKind=<partition type>&Tar
getReplicaSelector=<target replica selector>&ListenerName=<listener name>&Timeout=<timeout
in seconds>
```

Here are some more details on some of these fields:

- **`PartitionKind`** This can be either `Int64Range` or `Named`.
- **`TargetReplicaSelector`** This specifies how the target replica for a stateful service is selected. For stateful services, this parameter can have a value of `PrimaryReplica` (the default), `RandomSecondaryReplica`, or `RandomReplica`. For stateless services, this parameter has no effect, and reverse proxy picks a random service instance to handle the request.
- **`ListenerName`** This specifies the endpoint to which the client request should be forwarded. When a service has multiple listeners, this parameter can be used to identify which listener endpoint to use. This parameter is ignored if there's only one listener.
- **`Timeout`** This defines the maximum time allowed for the downstream service to handle the HTTP request generated by reverse proxy. This parameter is optional.

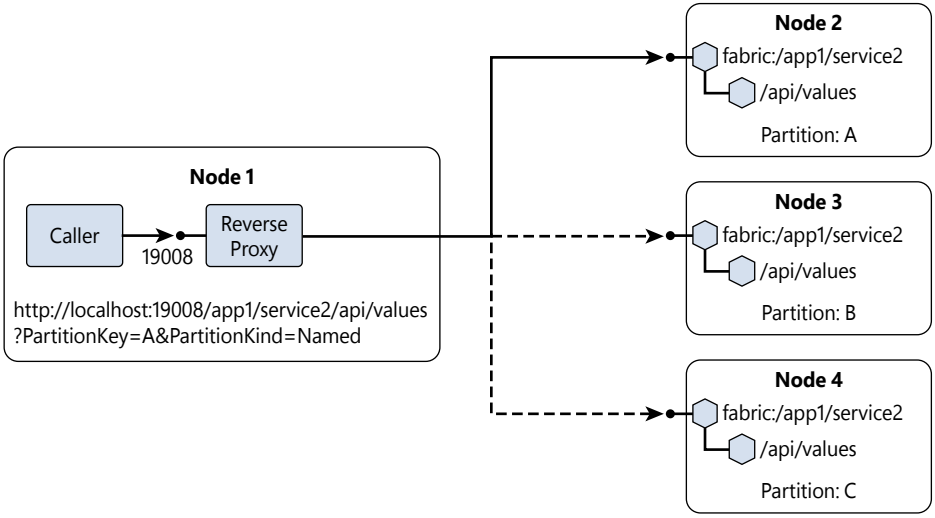

FIGURE 6-7 Service Fabric reverse proxy.

404 Errors

When reverse proxy sees a 404 error, it assumes the service replica has been moved to another node and tries to resolve the address again. If, however, the client requested a nonexistent resource, your service must help the reverse proxy stop trying to resolve the address again by returning an `X-ServiceFabric: ResourceNotFound` header. This header informs the reverse proxy that the requested resource is indeed not found on the server (but the server is still working).

Note You can configure the reverse proxy port with a public load balancer (such as an Azure load balancer) to allow your services to be accessed from a remote client. If you do, all HTTP/HTTPS services are exposed through the public load balancer and reverse proxy.

DNS Service

For services that are unaware of Service Fabric runtime, such as containerized services in Docker containers, Service Fabric provides a DNS service that provides the DNS protocol on top of the Service Fabric Naming Service. Figure 6-8 shows how DNS service works:

1. A service registers its listening address and DNS name with the DNS service.
2. A client does a DNS lookup to resolve service2.app1.
3. The DNS service uses the registered service name (fabric:/app1/service2) and uses Service Fabric Naming Service to look up the service's listening address.

4. The client uses the resolved listening address to invoke the service.

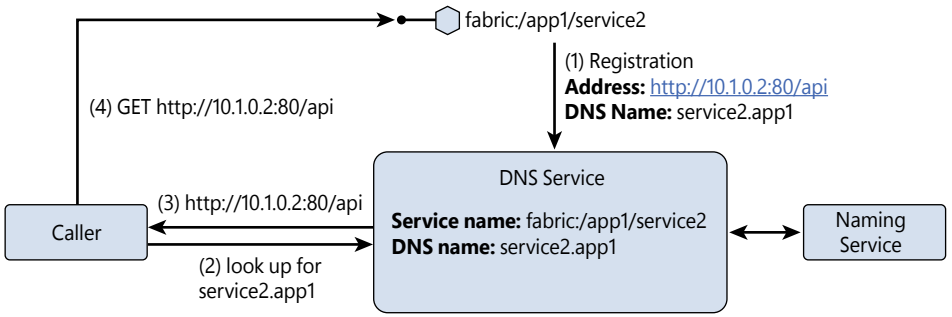

FIGURE 6-8 Service Fabric DNS service.

You can enable DNS service on a new Service Fabric cluster when you create it. To do so, open the Cluster Configuration blade in Azure Management Portal and select the **Include DNS Service** check box. (See Figure 6-9.)

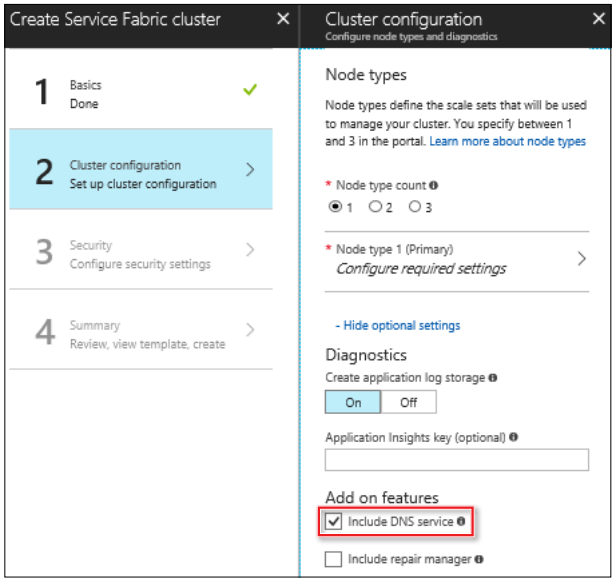

FIGURE 6-9 Enabling DNS service when creating a new cluster.

To define a DNS name for a service, add a `ServiceDnsName` attribute to the service in your application's manifest file, as shown in the following code snippet:

```
<Service Name="Stateless1" ServiceDnsName="service1.application1">
  <StatelessService ServiceTypeName="Stateless1Type" InstanceCount="1">
    <SingletonPartition />
  </StatelessService>
</Service>
```

Advanced Rolling Upgrades

As noted in Chapter 5, Service Fabric performs a rolling upgrade when upgrading an application. During this process, Service Fabric walks through update domains, making sure each update domain has been upgraded and verified before moving to the next one. Chapter 5 covered the basics of rolling upgrades. This section covers a couple of more advanced topics in application upgrades.

Configuration and Data Changes

A Service Fabric service is made up of code packages, configuration packages, and data packages. Each package is versioned separately. During rolling upgrades, only packages with newer version numbers are updated. This means the service code must be restarted only when there are actual code changes. Because changing configuration packages or data packages doesn't cause the service to restart, the service replica runs continuously during configuration and data changes. To pick up new configurations and data, your service code must handle a couple of system events to load the updated contents.

Let's walk through an example to see how to load new configurations dynamically:

1. Create a new Service Fabric application named ConfigurationUpdate with one stateless service named Stateless1.

2. Modify the Settings.xml file in the PackageRoot\Config folder of the service project to define an `IncrementStep` setting:

```
<Settings ...>
  <Section Name="MyConfigSection">
    <Parameter Name="IncrementStep" Value="1" />
  </Section>
</Settings>
```

3. Define a local variable in the `Stateless1` class:

```
int incrementStep = 1;
```

4. At the beginning of the `RunAsync` method, add the following event handler to pick up configuration changes automatically:

```
this.Context.CodePackageActivationContext.ConfigurationPackageModifiedEvent+=
    CodePackageActivationContext_ConfigurationPackageModifiedEvent;
```

   The event handler code looks like this:

```
private void CodePackageActivationContext_ConfigurationPackageModifiedEvent(object
sender, PackageModifiedEventArgs<ConfigurationPackage> e)
{
    var configSection = this.Context.CodePackageActivationContext
        .GetConfigurationPackageObject("Config");
    var text = configSection.Settings.Sections["MyConfigSection"]
        .Parameters["IncrementStep"].Value;
    incrementStep = int.Parse(text);
}
```