### **Distributed Locking with Dapr**

With developing applications used by many users, there may be situations where data security is required. There may be scenarios such as the same or different services making changes to or using data at the same time. Here, Dapr offers the distributed lock feature to ensure data accuracy or to eliminate order confusion in the use of services.

Distributed locking is used to control concurrent access of resources across microservices. This prevents microservices from accessing the same resource at the same time and possible conflicts.

The distributed lock feature can be used to read sequential messages sequentially, to access stock data in e-commerce applications, and in trade applications where data changes rapidly.

#### **Concurrency Control**

Distributed lock synchronizes multiple services’ use of the same resource. This is vital to maintain data consistency and system integrity.

#### **Flexibility**

The locking mechanism can work across a variety of storage systems and platforms.

#### **Scalability**

In large-scale systems, Dapr’s distributed lock enables efficient sharing of resources even in high-demand situations.

#### **Timeout and Retry Mechanisms**

If a resource is locked, Dapr manages other services’ access to that resource through timeout and retry mechanisms. Bir uygulama bir kilidi edindiğinde, bir istisna ile karşılaşır ve kilidi serbest bırakamazsa, kilit bir süre sonra otomatik olarak serbest bırakılır. Bu, uygulama başarısızlıklarında kaynak kilitlenmelerini önler.

### **Real Life Usage Scenario**

Let’s assume we have an e-commerce platform. Let’s imagine that there are two different systems that update a product stock and take orders. In this case, the number of product stocks must be managed correctly.

Let App1(top) represent the service that updates the stock. Let App1(bottom) represent the ordering service. Dapr’s distributed locking feature manages these transactions trying to access the same data at the same time, ensuring that each transaction updates the stock count sequentially. Thus, incorrect calculation of the number of stocks is prevented.

Dapr locks App1(bottom) using the redis lock mechanism. Then App1(top) updates the relevant stock data. For App1(bottom), the lock is removed and updated stock data for the order is provided. Thus, Dapr prevents confusion in stock data.

### **Demo**

Let’s experience Dapr’s magical tool in an application. Of course, the application will be a .NET application :)

#### **Senerio**

Let’s design the scenario we gave as an example in a simple demo. We will lock the stock version until the stock updating endpoint is finished.

Let’s add a model containing ItemId and Quantity for stock.

public class StockItem  
{  
 public string ItemId { get; set; }  
 public int Quantity { get; set; }  
}

Next, we will create some configuration files for dapr. Let’s create a “components” folder in the same directory as the project. Next, let’s create the “lockstore.yaml” file in this folder.

apiVersion: dapr.io/v1alpha1  
kind: Component  
metadata:  
 name: lockstore  
spec:  
 type: lock.redis  
 version: v1  
 metadata:  
 - name: redisHost  
 value: localhost:6379  
 - name: redisPassword  
 value: password

An inventory service will be required that includes two methods that return and update inventory data. We added a dictionary to use it like a database because, this application is a simple demo.

public class StockService : IStockService  
{  
 private readonly Dictionary<string, StockItem> \_stockDatabase = new Dictionary<string, StockItem>()  
 {  
 {"product1",new StockItem{ItemId = "product1", Quantity = 23}}  
 };  
  
 public Task UpdateStockAsync(StockItem item)  
 {  
 \_stockDatabase[item.ItemId] = item;  
 return Task.CompletedTask;  
 }  
  
 public Task<StockItem?> GetStockAsync(string itemId)  
 {  
 \_stockDatabase.TryGetValue(itemId, out var item);  
 return Task.FromResult(item);  
 }  
}

The lock that updates the stock in StockController is added with the Lock method as follows. The lock is maintained until the using block is disposed.

public class StockController : ControllerBase  
{  
 private readonly IStockService \_stockService;  
 private readonly DaprClient \_daprClient;  
  
 public StockController(IStockService stockService, DaprClient daprClient)  
 {  
 \_stockService = stockService;  
 \_daprClient = daprClient;  
 }  
  
 [HttpPost("update-stock")]  
 public async Task<IActionResult> UpdateStock([FromBody] StockItem stockItem)  
 {  
 await using (var lockResponse = await \_daprClient.Lock("lockstore","stockdata","lockOwner",1))  
 {  
 if (!lockResponse.Success)  
 return BadRequest("Not locked");  
 await \_stockService.UpdateStockAsync(stockItem);   
 }  
 }  
   
 [HttpGet("get-stock/{itemId}")]  
 public async Task<IActionResult> GetStock(string itemId)  
 {  
 var stockItem = await \_stockService.GetStockAsync(itemId);  
 return Ok(stockItem);   
 }

### **Inferences**

In this article, the features of the dapr distributed lock component to maintain data consistency are mentioned. A simple demo was designed to guide with a scenario for distributed lock.

Please contact me if you would like to add to or correct any parts of this article. All the best…