Uygulama Sanallaştırma Ortamları için Kaynak Yönetimi Sistemi

Provisioning System for Application Virtualization Environments

Tolga Büyüktanır[1,2], Hakan Tüzün[3], Mehmet S. Aktaş[4]

[1]Bilgisayar Mühendisliği Bölümü, Yıldız Teknik Üniversitesi, İstanbul, Türkiye, tolga.buyuktanir@std.yildiz.edu.tr
[2]Ar-Ge Ofisi, Link Bilgisayar A.Ş., İstanbul, Türkiye, tbuyuktanir@linkbilgisayar.com.tr
[3]Ar-Ge Ofisi, Link Bilgisayar A.Ş., İstanbul, Türkiye, htuzun@linkbilgisayar.com.tr
[4]Bilgisayar Mühendisliği Bölümü, Yıldız Teknik Üniversitesi, İstanbul, Türkiye, aktas@yildiz.edu.tr

Özet-Uygulama sanallaştırma ortamlarında gerçek zamanlı olarak kaynak kullanımının izlenmesi ve önceden belirlenen kurallara uyan durumlar oluşması durumunda, sanallaştırma ortamına ayrılan kaynakların güncellenmesi, günümüzde önemli bir ihtiyaç haline gelmiştir. Bu çalışma kapsamında uygulama sanallaştırma ortamları için bir kaynak izleme ve yönetimi yazılım mimarisi tasarlanmış ve tasarlanan mimarinin prototip gerçeklemesi yapılmıştır. Elde edilen prototip yazılımın performans değerlendirilmesi yapılmış ve sonuçların başarılı olduğu gözlemlenmiştir.

Anahtar Kelimeler—uygulama sanallaştırma ortamı, docker container, kaynak ayırma, gerçek zamanlı olay analizi

***Abstract*—Monitoring application resource usage dynamically in application virtualization environments and in case of any occurence of pre-determined rules, updating resources allocated to virtualization environment have become important requirement today. In this study, a resource monitoring and management software architecture for application virtualization environments has been designed and implemented as a prototype. Performance evaluation of implemented prototype has been carried out and all the results were observed to be successful.**

Keywords—application virtualization environments, docker container, provisioning, real-time event anaylysis

I. INTRODUCTION

Mikroservis, yazılımları birbirinden bağımsız-deploy edilebilir küçük servislere ayıran ve servis tabanlı olarak geliştirilmesine olanak sağlayan, son dönemlerde yaygın olarak kullanılan bir dağıtık programlama tekniğidir. Bu yaklaşıma göre, bir servis daha küçük kapsamlı iş mantığını gerçekleyecek şekilde geliştirilmektedir. Bunun yanı sıra, mikroservis; servisler arası iletişimde, daha az network bant genişliği kullanacak şekilde, basit veri modelleri üzerinden (JSON nesneleri) iletişim kurmaktadır. Mikroservisler, farklı yeteneklere sahip olabilir, farklı programlama dilleriyle yazılmış olabilir ve farklı veritabanı teknolojileri kullanıyor olabilir.

Mikroservis mimarisi servis tabanlı mimarilerden (SOA) esinlenilerek, daha hızlı çalışan, daha az kaynak (network bant genişliği, işlemci kullanım oranı gibi) tüketen bir mimari yapı olarak ortaya çıkmıştır. Mikroservisler, bu açılardan geleneksel monolitik servislere oranla, son yıllarda yaygın kullanım alanı bulmuştur.

Mikroservisler, yaygın olarak, uygulama sanallaştırma teknolojileri ile birlikte kullanılmaktadır. Bu teknolojilerden biri olan Docker teknolojisi bu araştırma kapsamında incelenmiş ve kullanılmıştır.

Mikroservisin yaşam döngüsü içinde, uygulama sanallaştırma ortamı üzerinde çalışırken, kaynak ihtiyaçlarının anlık veya periyodik olarak değişebildiği görülmektedir. Örneğin; yaz tatillerinde hizmet veren bir otelin kullandığı bir uygulamayı düşünelim. Uygulama yaz aylarında daha sık kullanılacak ve nisbeten kış aylarında kullanımı azalacaktır. Diğer bir örnek; IMKB gün sonu verilerini uygulamaların ulaşıma açan bir servis, gün içinde farklı yoğunluklarda çalışabilecektir. IMKB kapanışı ardından, bu servisin yoğunluğu artarken, gün içi diğer zamanlarda daha az yoğunlukta çalışacaktır. Periyodik ve anlık kullanımlar için gösterilebilecek bu örneklerde, servislerin üzerinde çalıştıkları bulut bilişim ortamlarını en optimum bir şekilde kullanabilmek için, kaynakların kullanıma göre arttırılması ya da azaltılması gerekmektedir.

Bundan dolayı, bu araştırma kapsamında; uygulama sanallaştırma ortamlarında gerçek zamanlı olarak kaynak kullanımının izlenmesi ve önceden belirlenen kurallara uyan durumlar oluşması durumunda, sanallaştırma ortamına ayrılan kaynakların güncellenmesini gerçekleştirebilecek bir yazılım mimarisi üzerinde çalışıyoruz. Uygulama sanallaştırma ortamlarını dinamik olarak takip eden ve anlık olarak değişen kaynak ihtiyaçlarına göre, mikroservislerin üzerinde çalıştığı uygulama sallaştırma ortamını yöneten bir yazılım mimarisi öneriyoruz. Önerilen mimarinin prototip uygulamasının detaylarını tartışıyoruz. Bu çalışma ile amacımız; bulut bilişim kaynaklarını optimum şekilde kullanmak, kullandığın-kadar-öde metodunun mümkünlüğünü test etmek ve kullanıcıya anlık olarak kullanım verilerini göstermek şeklindedir.

Bu bildirinin organizasyon yapısı şu şekildedir: II. Bölüm'de problem tanımı yapılmaktadır. Temel kavramlar ve literatür taraması III.Bölümde paylaşılmaktadır. IV.Bölümde uygulama sanallaştırma ortamları için resource provisioning uygulama mimarisi sunulmakta, V. Bölümde bu mimariye uygun bir prototip uygulamanın gerçeklenmesi anlatılmaktadır.

Prototip uygulama üzerinde gerçekleştirilen testler VI. Bölümde anlatılmıştır. VII. Bölüm ise sonuçları ve gelecekteki çalışmaları içermektedir.

II. Problem Tanımı

Bir sanallaştırma ortamı için kaynak ayrılırken, herhangi bir sınırlama yoktur. Sanallaştırma ortamı, üstünde çalıştığı ana bilgisayarın izin verdiği ölçüde kaynakların çoğunu kullanabilir. Multi-tenant bir yapı düşünüldüğünde, birden fazla kiracı ve birden fazla uygulama aynı host üzerinde çalışabilir. Böyle durumlarda kaynakların etkili bir şekilde kullanılması gerekmektedir.

Bir uygulama için, üzerinde çalıştığı sanallaştırma ortamındaki cpu kaynağı kullanımı sınırlandırılabilir. Ancak çoğu durumda, sanallaştırma ortamında önceden belirlenmiş kullanım değerleri kullanılmaktadır. Örneğin, bir uygulama sanallaştırma örneği olan Docker Container için, Completely Fair Scheduler (CFS), kaynak kullanım değerlerini/oranlarını belirlemek için kullanılmaktadır. Yeni version Docker'larda gerçek zamanlı CFS'te kullanılabilmektedir. Cpu Scheduling and prioritization, gelişmiş kernel özellikleridir. Bu açıdan gerçek zamanlı CFS doğru bir şekilde yapılmadığı taktirde, sistem kararsız davranabilen ve hatta kullanılamaz olabilen bir sisteme dönüşecektir. Bu araştırma kapsamında yapılan çalışmalarda, cpu kaynaklarının kullanılması için default CFS ayarları kullanılmıştır [7].

Bir uygulama sanallaştırma ortamı için çok fazla memory kaynağı ayrılmaması gerektiği gibi, "out of memory exception"'a sebep olabilecek, yetersiz memory miktarı belirlemek de doğru değildir. Memory dolduğu zaman, sistem fonksiyonlarının çalıştırılması için uygulama sanallaştırma ortamı üzerinde çalışan prosesler durdurulabilir. Bu da istenmeyen ve kaçınılması gereken bir durumdur. "Out of Memory" hatası oluşumunu önleyebilmek için şu yöntemler izlenebilir: a) Uygulama için testler yapılmalı ve memory ihtiyacı belirlenmeli ve üst limit tayin edilmelidir. b) Swap alanı belirlenmelidir. Swap, memory'den daha yavaş çalışmaktadır. Ancak bu yaklaşımla "out of memory" hataları önlenebilecektir.

Bir uygulama sanallaştırma ortamı için, deneye dayalı testler yapılarak kullanılacak memory miktarı önceden belirlenmeye çalışılsa da, çoğu zaman memory kullanımı sabit olmamakta ve ani değişmektedir. Bu yüzden memory kaynaklarının ayrılması gerçek zamanlı olarak değiştirilebilmelidir. Memory kullanımı arttığında anlık olarak kaynak takviyesi yapılmalı, memory kullanımı azaldığında uygulama sanallaştırma ortamı için ayrılan memory miktarı geri alınmalıdır.

Kaynak kullanımının örüntüsünün oluşturulması için sanallaştırma ortamlarında olan log dosyalarının toplanması ve toplanan log dosyalarının analizinin yapılması da önem arzetmektedir.

Bu araştırma kapsamında üzerinde çalıştığımız, gerçek zamanlı kaynak yönetimi uygulamasının, sistemin log dosyalarından bulunabilecek, olası aksaklıkların tespit edilebilmesi ve bu sayede bu aksaklakların giderilmesi ve uygulamanın çalışmasının iyileştirilmesi için de kullanılabileceği öngörülmektedir.

III. Temel Kavramlar ve Literatür Taraması

Mikroservis, yazılımları birbirinden bağımsız-deploy edilebilir küçük servislere ayıran ve servis tabanlı olarak geliştirilmesine olanak sağlayan, son dönemlerde yaygın olarak kullanılan bir dağıtık programlama tekniğidir. Lewis ve Fowler'a göre mikroservisler etki alanlarına göre fonksiyonel olarak yazılmalıdır. Her servis otonom ve full-stack'tir. Dolayısıyla, herhangi bir serviste yapılan değişiklik, iletişim arayüzünde değişiklik olmadığı sürece diğer servisleri etkilemez. Bu sayede loosely coupled (genellikle REST interface ile), yüksek kohezyon, esneklik, çeviklik ve ölçeklenebilirlik sağlanmaktadır [1-2].

Sam Newman ise mikroservisleri birlikte çalışan küçük ve otonom servisler olarak tanımlamaktadır. Bu servisler, işin sınırlarına göre bir fonksiyonelliğin parçası olarak geliştirilir. Geliştirme yapılırken kodun çok fazla büyüyüp içinden çıkılmaz bir hale gelmesinden kaçınılır. Bütün iletişim, network üzerinden yapılır ve servisler değiştirilse bile başka değişikliğe ihtiyaç duyulmaması için servisler birbirinden bağımsız olarak tasarlanır [3]. Mikroservisler yaygın olarak uygulama sanallaştırma teknolojileri ile birlikte bulut bilişim altyapıları üzerinde kullanılmaktadır.

Bulut teknolojileri sanallaştırma üzerine bina edilmiştir. Temel olarak üç sanallaştırma tipi vardır. Type 1 ve Type 2 sanal makina kurulumu ve önyükleme için çok fazla zamana ihtiyaç duymaktadır. Bu sanallaştırma yöntemlerinin cpu, memory ve storage kullanımı da bir hayli fazladır. Diğer sanallaştırma yöntemi light-weight sanallaştırma (uygulama sanallaştırma) tipidir. Bu sanallaştırma yöntemi diğerlerine göre daha hızlıdır ve kaynak tüketimi ihmal edilebilecek kadar küçüktür. Sanal makina oluşturma ve ön yükleme süresi oldukça azdır. Gupta ve ark. [8]'de sanallaştırma tiplerinin karşılaştırılması sonuçlarını ayrıntılı olarak paylaşmaktadır.

Uygulama sanallaştırma mikroservis mimarileri için en uygun olanıdır. The New York Times web sitesi, PayPal web sitesi, arxiv.org, Uber, ebay gibi web uygulamaları bu sanallaştırma metodunu mikroservisler ile kullanmayı tercih etmiştir. Paypal 218 milyon aktif kullanıcısına hizmet verebilmek için 700 uygulamasını Docker üzerinde çalıştırmaktadır. Bu uygulamalar için toplamda 150 milyon uygulama sanallaştırma ortamı çalışmaktadır. Uygulama sanallaştırma ile Paypal, %50 oranında geliştirici üretkenliğini arttırmıştır. Bazı uygulamalarında %10-20 oranında verimlilik artışı sağlamıştır [9].

IT bölümleri içindeki farklı disiplinlerin (Developer-Operations) güven içinde çalışması prensipleri olarak DevOps ve DevOps pratikleri yaygın olarak kullanılmaktadır. Bu pratiklerde, uygulama sanallaştırma ortamı üzerinde çalışan servislerin üzerinde değişiklik yapılması ve yeni halinin tekrar yayınlanmasının az bir sürede gerçekleşebileceği gözlemlenmektedir. Yine DevOps araçlarıyla kesintisiz monitoring sağlanarak, geliştiriciye performans ile alakalı geri dönüşler yapmak ve anomali tespiti yapmak kolaylaşmaktadır [2-5].

Uygulama sanallaştırma ortamlarının bir çok avantaj sağladığı gibi ölçeklenebilirlik açısından da kolaylık sağladığı görülmektedir. Ancak, uygulama sanallaştırma ortamlarının

sayısı arttıkça, kullanılan bilişimsel kaynak miktarı artmaktadır. Kaynakların yetersiz kaldığı durumlarda bütün uygulama sanallaştırma ortamlarının kaynaklarının güncellenmesi ya da kaynak ihtiyacı artan uygulama sanallaştırma ortamının kaynak miktarının arttırılması gerekmektedir. Bu tür karmaşık durumlar ele alınarak, kaynak planlaması, kullanılan sanallaştırma ortamından bağımsız bir şekilde, yapılmalıdır. Bu sayede uygulama performansı arttırılmakta ve ya mevcut ihtiyaçlar için optimum sonuç bulunabilmektedir [11-12].

IV. Önerilen Yazılım Mimarisi

Çalışmada önerdiğimiz dağıtık uygulamanın mimarisi Şekil 1'de gösterilmiştir. Önerdiğimiz mimariye göre birden fazla ve her biri farklı konumlarda olan bulut sanallaştırma ortamlarının kaynak yönetimi gerçekleştirilebilmektedir.

Şekil-1' de resmedilen, mavi dikdortgenler içinde görülen ve "*Containerization Platform Hosts*" olarak adlandırılan komponentlerde uygulama sanallaştırma ortamları vardır. Her ortam üzerinde bir mikroservis çalışmaktadır. Mikroservisler renkli beşgenler ile gösterilmiştir.

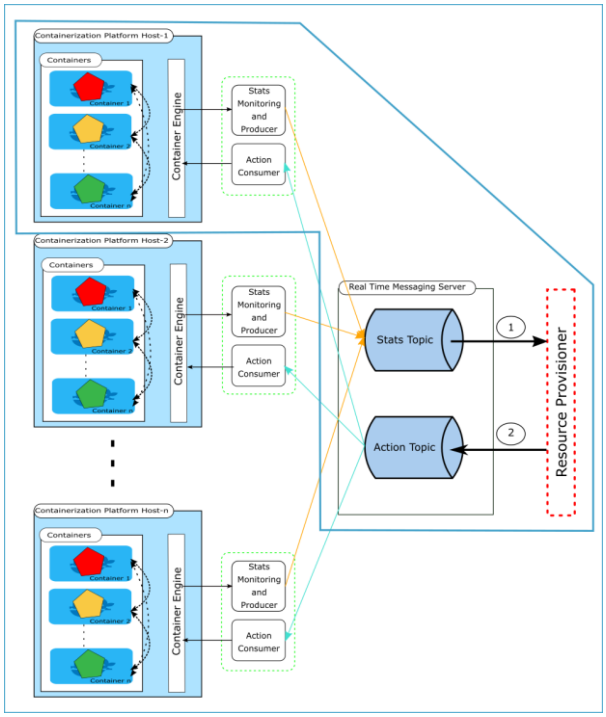
Mikroservisler sanallaştırma ortamları içerisinde çalışırken, yeşil ve kesik-çizgili dikdörtgenler içerisinde gösterilmiş olan "*Stats Monitoring and Producer"* olarak adlandırılan komponentler, sanallaştırma ortamlarından kaynak kullanım bilgilerini almakta ve konu tabanlı üye-ol/yayınla mesaj iletim kanalı (topic based pub-sub messaging system) üzerinden kaynak yönetim sistemine iletmektedir. Burada mesaj iletimi "Stats Topic" adlı konu üzerinden gerçekleşmektedir.

Şekil 1'de kırmızı kesik-çizgili dikdörtgen ile gösterilen *"Resource Provisioner"* olarak adlandırdığımız komponent, *"Stats Topic"* adlı konu üzerinden gönderilen mesajları, bu konuya üye olduğu için alabilmektedir. Mikroservisler sanallaştırma ortamı içerisinde çalışırken, "*Resource Provisioner"* komponent' i sanallaştırma ortamlarının istatistiklerini gerçek zamanlı olarak takip edebilmektedir. Sistemden gelen mesajların kaynak yönetim sistemine akış yönü, Şekil 1'de, 1 numaralı ok ile gösterilmiştir.

"Resource Provisioner", herhangi bir sanallaştırma ortamının kullandığı memory miktarı %90 üzerine çıktığı anda, ilgili sanallaştırma ortamının memory miktarını iki katına çıkaran komutu üreterek, kaynak değişikliliği ile ilgili aksiyonun gerçekleşmesi için pub-sub mesaj iletim sistemi üzerindeki bir başka konuya (*Action Topic*)'e yazmaktadır. Memory kullanımı %10'un altına düştüğü anda ise memory ihtiyacı düşen sanallaştırma ortamının memory kaynak miktarının yarıya düşürülmesi için gerekli komut üretilerek, yine "*Action Topic"*'e yazılmaktadır. Bu işlemdeki mesajlaşmanın yönü, Şekil 1'de 2 numaralı ok ile gösterilmiştir. %90 ve %10 oranları konfigürasyon dosyasında belirlenmiş, kullanıcı tarafından konfigüre edilebilir değerlerdir.

Şekil 1'de yeşil noktalı dikdörtgenler içerisinde gösterilen "*Action Consumer"*, "*Action Topic"* üzerinden gönderilen mesajları okuyarak, mesajın içeriğinde yer alan komutu gerçekleştirmektedir. Bu sayede kaynak ayırma/bırakma işlemi lokasyon bağımsız olarak gerçekleştirilebilmektedir.

"Resource Provisioner" aynı zamanda cpu kullanım miktarını da takip etmekte ve gerçek zamanlı olarak cpu ihtiyaçlarını güncelleyebilmektedir. Bu çalışmadaki amacımız, kaynakların anlık olarak değiştirilerek daha verimli kullanılabileceğinin gösterilmesidir. Kaynak kullanımı anlık olarak değişebileceği gibi periyodik olarak da değişebilmektedir. Periyodik değişikliklerin kestiriminin yapılması ve 'kullandığın kadar öde' yaklaşımının işletilebilmesi, sanallaştırma ortamı üzerinde çalışan uygulamaların log dosyalarının kaydedilmesiyle ve bu kayıtlardan çıkarılacak analizlerle mümkün olabilecektir.



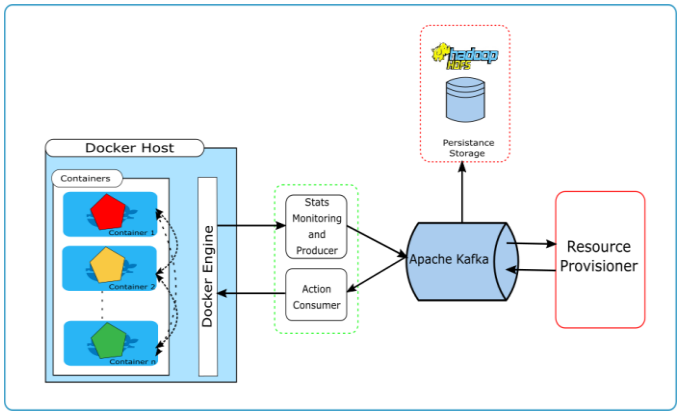
Şekil 1- Uygulama Yazılım Mimarisi

V. Prototip Uygulama

Bu bölümde, bir önceki başlıkta önerdiğimiz ve Şekil 1'de sunduğumuz mimari ile tasarlanmış prototip uygulamamızı anlatmaktayız. Prototip uygulamamızda kullandığımız teknolojiler Şekil-2' de resmedilmektedir. Prototip uygulamasında uygulama sanallaştırma ortamı olarak Docker container platformu tercih edilmiştir. Pub-Sub sistemi olarak Apache Kafka kullanılmıştır. Uygulamamız bir Docker container' ı çalıştıran sunucu üzerinde koşturulmuştur.

Sunulan uygulamada merkezi ve gerçek zamanlı çalışan bir kaynak ayırma (provisioning) sistemi geliştirilmesi hedeflenmiştir. Bu sebeple, gerçek zamanlı docker container stats verileri okunarak, Apache Kafka Server'a mesaj olarak gönderilmektedir. Eşzamanlı bir şekilde, "*Resource Provisioner"* modülü tarafından mesajlar geliş sırasına göre okunmakta ve bu mesajlara göre kaynak (memory,cpu vb.) miktarının değiştirilmesi gereken uygulama sanallaştırma ortamlarına karar verilmektedir. Kaynak miktarı artacak ve azalacak Docker container' lar için kaynak değiştirme komutu mesaj olarak tekrar Apache Kafka'ya iletilmektedir. Stats verilerinin ve güncelleme komutlarının iletildiği iki temel konu

kullanılmıştır: “Stats Topic” ve “Action Topic”. Kaynak güncellemelerinin iletilmesi için kullanılan “Action Topic” konusuna yeni bir mesaj gelir gelmez, bu mesajda yer alan komut docker engine üzerinde anında uygulanmakta ve kaynak miktarı güncellenmektedir.



Şekil 2 – Prototip Uygulama Mimarisi

Şekil 2’de mimarisi sunulan bu uygulama için üç python script yazılmıştır. Bunlardan ilki; yeşil kesik-çizgili dikdörtgen içerisinde gösterilen “*Stats Monitoring and Producer”* modülünü gerçeklemek içindir. Bu modül ile gerçek zamanlı olarak container üzerinden durum bilgileri okunur ve Apache Kafka üzerinde bulunan “*Stats Topic”*’ine yazılır. Şekil 2’de gösterilen kırmızı kesik-çizgili dikdörtgen ile gösterilen “*Resource Provisioning”* modülü için ise ikinci bir python script yazılmıştır. Bu modülde “*Stats Topic*”’ten container’ların istatistikleri sıralı olarak alınmaktadır. Memory miktarı %90 kullanımın üzerinde ya da %10 kullanımın altında ise sırasıyla memory miktarını iki katına çıkarma ya da yarıya indirme komutları oluşturulur ve mesaj olarak “*Action Topic*”’e yazılır. Bu sayede kaynak günceleme kararı merkezi olarak verilmiş olmaktadır. “*Action Topic*”’i sürekli olarak dinleyen, yeşil kesik-çizgili dikdörtgen içerisinde gösterilmiş “*Action Consumer”* sunulan uygulamanın son modülünü oluşturmaktadır ve bu modül içinde üçüncü bir script yazılmıştır. Bu script “*Action* Topic” e gelen bütün mesajları dinlemektedir. Gelen mesaj Docker container için bir kaynak ayırma komutudur. “*Action Consumer”* bu mesajı işler ve container’ların memory kaynağını, ilgili servisin metodlarını çağırarak günceller. Consumer ve Producer’lar *kafka-python* kütüphanesi kullanılarak yazılmıştır.

Apache Kafka üzerinde kaydedilen mesajlar aynı zamanda dağıtık bir dosya sistemine de (Hadoop Distributed File System (HDFS)) kaydedilmektedir.

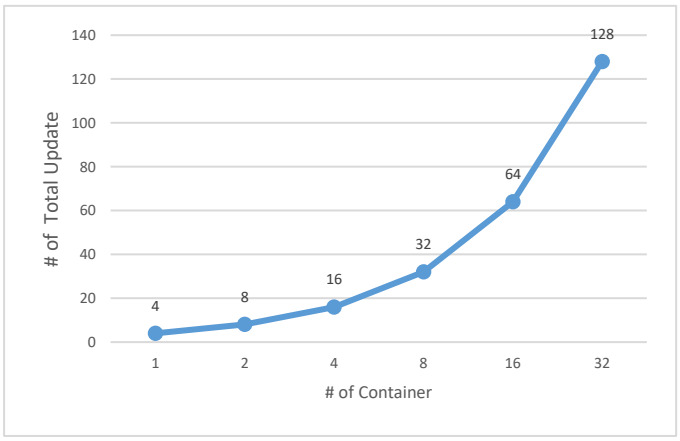
Apache Kafka’dan consumer ile mesajların Apache Kafka’ya yazıldığı sırayla okunması, ilgili konunun tek partition olarak tutulması ile garanti edilmektedir. Sunmakta olduğumuz uygulamamızda konuları tek partition’da tutmaktayız. Birden fazla consumer ihtiyacı olan uygulamalar için partition sayısının arttırılması, performans açısından önemlidir. Mesajların sıralı bir şekilde gönderilmesi de bir gereklilik olduğundan, bu konuda farklı çözümler üretilebilir.

VI. Performans Değerlendirmesi

Bu bölümde, prototip uygulama üzerinde gerçekleştirdiğimiz performans testleri ve test sonuçları paylaşılmaktadır. Testler, Ubuntu 16.04 LTS kurulu bir sanal makine’de 4 CPU ve 10 GB memory ile yapılmıştır. Testler için Docker 18.03 versiyonu kullanılmıştır.

Docker containerlar üzerinde, container’ın memory ihtiyacını sürekli olarak arttırmak amacıyla yazılmış bir python script kullanılmıştır. Jenerik olarak container oluşturmak için container’lar bash script içerisinde oluşturularak içerisinde python script çalıştırılmış ve başlangıç memory miktarı 5 megabayt olarak belirlenmiştir. Containerlar jenerik bir şekilde oluşturulup başlatıldığından herhangi bir zaman kaybı yaşanmamıştır.

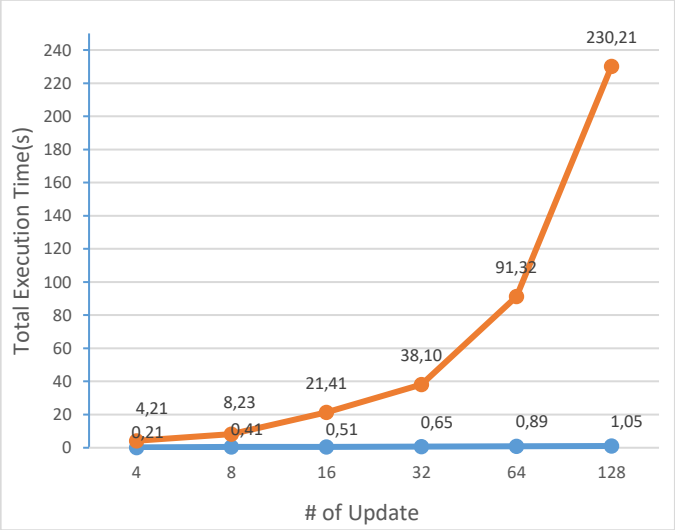
Docker container içinde çalıştırılan uygulama öncelikle 5 MB memory ile başlatılmıştır. İçerisinde çalışan uygulamanın memory kullanımı ihtiyacı arttıkça, container’ın memory ihtiyacına göre, *Resource Provisioner* container için ayrılan kaynağı 4 kez (80 mb’a kadar) arttırmıştır. Aynı işlem 2, 4, 8, 16 ve 32 farklı container başlatılarak da tekrarlanmıştır. Yapılan testler sonucunda, ortalamada herbir container’ın 4 defa kaynaklarının güncellendiği görülmüştür. Şekil 3’te başlatılan container sayısı değişirken ve ihtiyaç duyulan kaynak güncelleme sayısı gösterilmektedir.



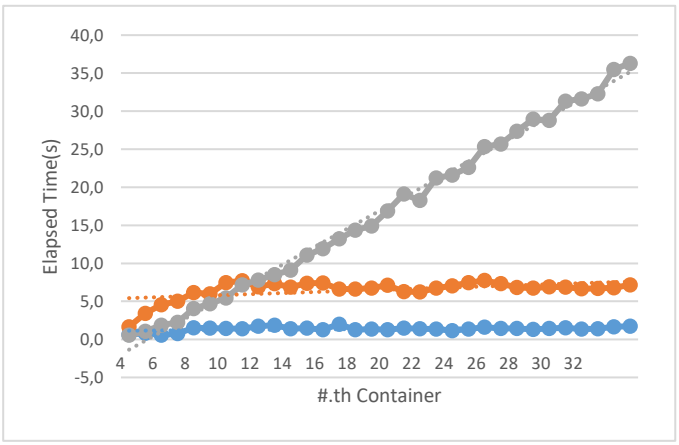
Şekil 3 - Başlatılan container sayısı ve containerlar için ihtiyaç duyulan toplam kaynak güncelleme sayısı gösterilmektedir.

Güncellemelerin tamamının gerçekleştirilmesi için geçen süreler ve ihtiyaç duyulan güncelleme sayısı arttıkça, bir güncellemenin ortalama gerçeklenme süresi Şekil 4’te sunulmuştur. Şekil-4’ de resmedildiği üzere, 4 update gerçekleştiği durumda, ortalamada her bir kaynak güncelleme 0.21 saniye sürmekte ve container tarafından ihtiyaç duyulan memory miktarının tamamen karşılanıp, memory kullanımın dengelenmesi için 4.21 saniye geçmektedir. Aynı şekillde; 32 container için toplam 128 defa kaynak güncellemesi gerekmektedir. Her bir güncelleme ortalama 1.05 saniye sürmekte ve bütün kaynak güncellemelerinin bitirilip, ihtiyaç

duyulan memory miktarının karşılanması 230.21 saniye sürmektedir.



Şekil 4 Sunulan grafikte, aynı anda başlatılan container sayısı arttıkça, ihtiyaç duyulan güncellemelerin her birinin ortalama gerçekleştirilme zamanı ve bütün güncellemelerin gerçekleştirilip kaynak ihtiyaçlarının tamamen giderilmesi için geçen zaman gösterilmektedir.



Şekil 5 Containerların kaynak ihtiyaçlarının, yeni containerların başlatılma süresine olan etkisi sunulmuştur.

Şekil 5 için 3 test yapılmış ve testler yapılırken 10 container çalışır vaziyette tutulmuştur. Her testte 32 container yeniden başlatılmış ve containerların her birinin başlama süresi incelenmiştir. İlk testte, containerlar yetersiz memory miktarıyla (5MB) başlatılmış ve "*Resource Provisioner"* çalıştırılmamıştır. Bu test, Şekil 5'te sunulan grafikte gri renkli eğri ile gösterilmiştir. İkinci testte, ilk testte olduğu gibi containerlar yetersiz memory miktarıyla (5MB) başlatılmış fakat bu kez "*Resource Provisioner"* çalıştırılmıştır. Şekil 5'te turuncu renkli eğri ile bu test sonucu sunulmuştur. Son olarak da containerlar yeterli memory miktarı (80MB) ile başlatılmıştır. Son test mavi-renkli eğri ile Şekil 5'te mevcuttur. Sonuç olarak; sistemde çalışan ve kaynaklarının güncellenmesi gereken container sayısı arttıkça, docker komutlarının çalıştırılma süresi bir miktar arttığı gözlemlenmektedir. Fakat containerların kaynaklarının yetersiz olduğu ve resource provisioning sisteminin olmadığı durumlarda, docker komutları kayıtsız kalınamayacak kadar yavaş çalışmaktadır. Dolayısıyla resource provisioning sistemi sadece kaynakların optimum şekilde kullanılması için değil aynı zamanda docker komutlarının işletilmesinin yavaşlamasını önlemek için de gereklidir.

VII. Sonuçlar ve Gelecekteki Çalışmalar

Bu araştırma kapsamında, uygulama sanallaştırma ortamlarında gerçek zamanlı olarak kaynak kullanımının izlenmesi ve önceden belirlenen durumlar oluştuğunda, sanallaştırma ortamına ayrılan kaynakların güncellenmesini sağlayacak bir uygulama için dağıtık yazılım mimarisi geliştirilmiştir. Geliştirilen yazılım mimarisinin bir prototip uygulaması gerçeklenmiştir. Prototip uygulamanın nasıl çalıştığının ortaya konulması amacıyla performans testleri gerçekleştirilmiştir.

Gerçekleştirilen testler sonucunda, uygulama sanallaştırma ortamının, kaynak ihtiyaçları yeterli olacak şekilde başlatıldığında, bu ortamlarda kaynak kullanım değişikliği gerçekleştirme için kullanılan komutların gecikmeler olmadan işletilebildiğini göstermektedir. Bu yüzden, uygulama sanallaştırma ortamları kullanılırken, en ideal yaklaşımın, kaynak ihtiyaçları önceden kestirilmiş bir şekilde, bu ortamları başlatmak olduğu saptanmıştır. Böylece hem kaynakları optimum şekilde kullanmak hem de anlık kaynak ihtiyaçlarına yanıt vermek için kaynak yönetim sistemlerini yüksek performasla kullanmak mümkün olabilecektir.

Gelecekteki çalışmalarımız arasında, uygulama sanallaştırma ortamlarından elde edilecek log dosyalarından yola çıkarak, bu ortamların başlatılması esnasında ilk ayrılması gereken kaynakların tahmin edilmesi yönünde, veriden öğrenme gerektiren araştırma ve geliştirme çalışmalarını yürütmeyi planlıyoruz.

Teşekkür

Bu araştırma süresince, çalışma ortamı, sunucu imkanları, bilgisayar kullanımı olanakları sağlayan Link Bilgisayar' a teşekkürlerimizi sunuyoruz.

References

[1] J. Lewis and M. Fowler, "Microservices," https://martinfowler.com/articles/microservices.html, 2014, [Online; accessed 18-January-2017].

[2] Aderaldo, C. M., Mendonça, N. C., Pahl, C., & Jamshidi, P. (2017, May). Benchmark requirements for microservices architecture research. In Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (pp. 8-13). IEEE Press.

[3] S. Newman, Building Microservices. O'Reilly Media, 2015.

[4] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara,

F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. arXiv preprint arXiv:1606.04036, 2016.

[5] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. Linux Journal, 2014(239):2, 2014.

[6] Dragoni, N., Lanese, I., Larsen, S. T., Mazzara, M., Mustafin, R., & Safina, L. (2017, June). Microservices: How to make your application scale. In International Andrei Ershov Memorial Conference on Perspectives of System Informatics (pp. 95-104). Springer, Cham.

[7] "Docker Resource Constrainsts," https://docs.docker.com/config/containers/resource_constraints/, 2018, [Online; accessed 17-April-2018].

[8] Gupta, Vipin, Karamjeet Kaur, and Sukhveer Kaur. "Performance comparison between light weight virtualization using docker and heavy weight virtualization." International Journal of Advanced Technology in Engineering and Science, Volume 05 (2017): 509-514.

[9] "THE JOURNEY TO 150,000 CONTAINERS AT PAYPAL," https://blog.docker.com/2017/12/containers-at-paypal/, 2018, [Online; accessed 04-May-2018].

[10] "Python client for Apache Kafka," https://github.com/dpkp/kafka-python, 2018, [Online; accessed 07-May-2018].

[11] NARDELLI, Matteo. Elastic Allocation of Docker Containers in Cloud Environments. In: ZEUS. 2017. p. 59-66.

[12] HOENISCH, Philipp, et al. Four-fold auto-scaling on a contemporary deployment platform using docker containers. In: International Conference on Service-Oriented Computing. Springer, Berlin, Heidelberg, 2015. p. 316-323.