IoT Verileri İçin Gerçek Zamanlı ve Ölçeklenebilir Büyük Veri Mimarisi: Karşılaşılan Problemler ve Geliştirilen Çözümler

Devrim Barış Acar[1][0000-0002-2321-8104], Arif Kamil Yılmaz[2], Serkan Remzi Küçükbay[3][0000-0002-5766-8138]

[1,2,3] STM Savunma Teknolojileri Mühendislik A.Ş, Ankara, Türkiye
dbacar@stm.com.tr, akyilmaz@stm.com.tr,
serkan.kucukbay@stm.com.tr

Özet. Günümüzde IoT (Internet of Things – Nesnelerin İnterneti) cihazların kullanımındaki artış beraberinde yüksek yoğunluklu ve farklı çeşitte verilerin oluşmasına sebep olmuştur. Bu verilerin alınması, işlenmesi, saklanması ve görselleştirilmesindeki zorluklar büyük veri sistemi bileşenlerinin kullanılmasını zorunlu hale getirmiştir. Veri alımı katmanında yüksek yoğunluklu verinin alınması, alınan verilerin sistemin diğer bileşenleri tarafından yüksek performanslı olarak tüketilebilmesi için dağıtık kuyruklama sistemlerinde saklanması, saklanan verilerin yakın gerçek zamanlı olarak karmaşık olay işleme motoru tarafından işlenmesi, tespit edilen anomaliler ve ham verinin saklanması, son olarak da bu verilerden çıkarılan sonuçların yakın gerçek zamanlı olarak görselleştirilmesi gerekmektedir. Büyük veri ekosisteminde bahsi geçen tüm aşamalar için kullanılabilecek bileşenler mevcuttur. Açık kaynaklı bu bileşenler her ne kadar hazır olsa da, veri işleme hattı (data pipeline) üzerinde her aşamada farklı problemlere çözümler üretmek gerekir. Örnek olarak yüksek yoğunluklu veriyi kabul edecek ve yüksek erişilebilir bir veri alım katmanı, veri saklama bileşeninde oluşabilecek aykırı bir durum nedeniyle tüm veri işleme altyapısının bekler duruma geçip verilerin gerçek zamanlı olarak görselleştirilememesi, kuyruklama bileşeninde saklanan verinin formatından dolayı yaşanacak performans düşüşü bunlara örnek olarak gösterilebilir. Bu çalışmada, IoT verilerinin işlenmesi için büyük veri mimarisi bileşenlerinin nasıl kullanıldığı, veri işleme hattı aşamaları üzerinde son bir yılda karşılaşılan problemler ve bu problemler özelinde geliştirilen çözümler paylaşılacaktır.

Anahtar Kelimeler: Nesnelerin İnterneti, IoT, Karmaşık Olay İşleme, Büyük Veri, Mesaj Kuyruklama, NoSQL, Veri Tabanı, Internet Of Things, CEP, Complex Event Processing.

Real Time and Scalable Big Data Architecture for Internet of Things Data: Problems Encountered and Devised Solutions

Devrim Barış Acar[1][0000-0002-2321-8104], Arif Kamil Yılmaz[2], Serkan Remzi Küçükbay[3][0000-0002-5766-8138]

[1,2,3] STM Savunma Teknolojileri Mühendislik A.Ş, Ankara, Türkiye
dbacar@stm.com.tr, akyilmaz@stm.com.tr,
serkan.kucukbay@stm.com.tr

Abstract. Pervasive use of IoT (Internet of Things) devices have led us to data that is increasing in density, volume and variety. The difficulty in ingestion, processing, storage and visualisation of this data has forced industries to use big data systems. Such systems can be classified in stages of a data pipeline; namely the ingestion stage responsible for high volume data acquisition, a distributed message queue stage enabling the other stages to consume data in high performance, a realtime event processing stage for analytics and detecting anomalies and a storage engine for storing raw data and lastly a visualization stage for realtime result displaying. Open source big data systems have solutions for all stages, but they should be customized to solve problems specific to each project. As examples; making data ingestion layer highly available, decoupling the storage and realtime visualisation layer in pipeline in order to enable the user see status of devices even there is a problem in storage layer or increasing message queue performance by using a different internal serialization format can be given.

In this work, how various big data ecosystem projects are used in the aforementioned data pipeline, the problems encountered in various stages last year, and the custom solutions developed will be shared.

Keywords: Internet of Things, Big Data, Message Queue, NoSQL, Database, CEP, Complex Event Processing

1 Giriş

Nesnelerin interneti (IoT), gerçek dünyadaki nesnelerin internet ağı üzerinden diğer nesne ya da sistemlerle iletişim içinde olduğu dinamik evrensel bir network yapısı olarak tanımlanabilir. Büyük veri tanımlarında 3 V olarak bahsedilen Hacım(Volume), Hız(Velocity), Çeşitlilik(Variety) kavramlarının en büyük kaynak sağlayıcısı IoT cihazlar olarak karşımıza çıkar. Akıllı olarak nitelendirdiğimiz bu cihazlar,

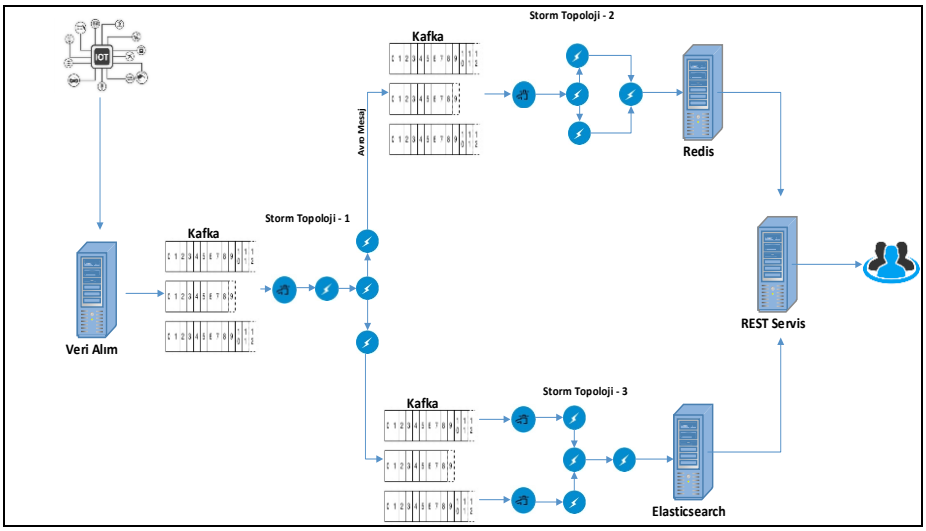
çevrelerinden topladıkları bilgileri merkezi bir sisteme aktararak bu verilerden anlamlı sonuçlar çıkarılmasına aracılık ederler. Bu cihazların sayılarındaki artış bunların ürettikleri verilerin artmasına ve bu verileri işleyebilecek sistemlerin gerekliliğini beraberinde getirmiştir. Çoğu açık kaynak kodlu olarak geliştirilen bu altyapılar bir araya gelerek büyük veri ekosistemi bileşenlerini oluşturmaktadır. Üretilen veri, büyük veri ekosistemi bileşenleri arasında bir sistemden başka bir sisteme hareket eder. Büyük veri ekosistemindeki uygun bileşenlerin bir araya getirilerek verinin işlenebilmesi için oluşturulan yapılara veri hattı adı verilir. Veri hattını oluşturan her bileşen mümkün oldukça birbirinden bağımsız, yüksek erişilebilir ve hat üzerindeki diğer bileşenin performansını artırmaya yönelik olarak tasarlanmalı ve seçilmelidir.

Bu çalışmamızda IoT cihazlarda üretilen farklı formatda, yüksek yoğunluklu ve hızlı olarak verileri işlemek için bir veri hattı tasarlanmıştır. Veri hattı üzerinde, veri alım aşamasında, yüksek erişilebilirliği, özelleşmiş veri doğrulama kurallarını işletmeyi ve yüksek hacımli veri alımını sağlamak için mevcut büyük veri ekosistemi bileşenleri yerine özgün bir çözüm üretilmiştir. Veri işleme aşamasında, garanti veri işleme prensibine sağdık kalmak koşuluyla, farklı sistemleri besleyecek çıktıların oluşması sağlanmıştır. Bir sistemi besleyecek çıktının üretilmesi aşamasında oluşabilecek aykırı durumun diğer sistemin çıktısında gecikmeye sebep olmasını engellemek için birbirinden bağımsız gerçek zamanlı veri işleme topolojileri tasarlanmıştır. Birbirinden bağımsız topolojiler arasında veri alışverişini sağlamak için kuyruklama sistemi kullanılmıştır. Kuyruklama sistemi üzerinde veriler farklı formatlarda tutulmuş, farklı veri formatlarının veri işleme hızına etkileri karşılaştırılmalı olarak incelenmiştir.

2 Genel Sistem Mimarisi

Aşağıdaki Şekil 1'de çalışmamız sırasında tasarlayıp geliştirdiğimiz sistemin genel mimarisi görünmektedir. IoT cihazlar tarafından üretilen veriler Veri Alım katmanı tarafından yüksek erişilebilir olarak sistem içine alınır. Veri Alım katmanındaki bileşen, birden fazla makinada Docker container içerisinde çalışan, yüksek erişebilirliği Kubernetes altyapısı ile sağlanmış, Go programlama dili kullanılarak yazılmış bir uygulama parçasıdır. Doğrulama işlemine tabi tutulan veriler Apache Kafka kuyruklama sistemi üzerinde belirli bir topic ile ilişkilendirilerek 8 farklı bölümde kuyruklanır. Apache Kafka sistemi 3 farklı makinada yüksek erişilebilir olarak hizmet vermektedir. Partition sayısı kadar paralellikte mesajları tüketen sistemlere performanslı mesaj tüketim imkanı sağlar. 3 farklı makine üzerinde yüksek erişilebilir ve hataya karşı duyarlı bir altyapı sunan Apache Storm gerçek zamanlı veri işleme altyapısı kullanılarak veriler üzerinde anomaliler ve analizler gerçekleştirilir. Apache Storm üzerinde 3 farklı topoloji tasarlanmıştır. Topolojiler arasındaki veri alışverişi için yine Kafka kuyruklama altyapısı kullanılmıştır. Giriş topolojisinin çıktıları tekrar Kafka üzerine yazılarak diğer topolojiler için girdi olarak kullanılır. Böylece birbirinden bağımsız olarak çalışan topolojilerde meydana gelen aykırı durumlar diğer topolojilerin ve bileşenlerin çalışmasını etkilemez. Her topoloji işlediği mesajlarla ilgili offset bilgilerini Kafka üzerinden takip ederek garanti mesaj işleme prensibini gerçekleştirir. Topolojiler boyunca akan mesajlarda tespit edilen anomaliler ve veriler

üzerinde yapılan analizler, anlık olarak Redis bellek tabanlı veri tabanında, toplu olarak da Elasticsearch veri tabanında saklanmıştır. Redis ve Elasticsearch veri tabanlarında anlık ve zaman serisi olarak saklanan bu veriler REST servisler kullanılarak dış dünyanın kullanımına açılmıştır.



Şekil 1. Genel Sistem Mimarisi

3 Karşılaşılan Problemler ve Geliştirilen Çözümler

3.1 Test Verisi

Sistemin yük altında nasıl davrandığının bulunması için IoT cihazlarını benzeştiren eden bir araç ("simülator") geliştirilmiştir. Veri üretim frekansı, tanımlı IoT cihazları listesi ve diğer yardımcı parametreler verilerek komut satırından çalıştırılan bu araç, 50.000 araca kadar saniyede bir veri üretimini benzeştirebilmektedir.

Bu araç ek olarak ürettiği verileri bir dosya içine de kaydedebilmektedir. Bu özellik veri alım altyapısının performans testi için kullanılmaktadır.

3.2 Veri Alım Altyapısı

Veri alım altyapısı IoT cihazlardan gelen verileri mesaj kuyruğuna aktarma görevini yerine getirmektedir. Veri alım altyapısı TCP soket bağlantısı açarak bu porta gelen verileri dinlemektedir. Mesaj kuyruğu yapısında kullanılan Apache Kafka bileşeni verileri bölümlere ayırarak saklamakta ve gelen mesajların sırasını sadece bu bölümler içinde garanti etmektedir. Gerçek zamanlı veri işleme altyapısında mesajların IoT cihazı bağlamında sıralı olması beklenildiği için verilerin mesaj kuyruğuna yazılırken IoT cihazı bazında bölümünün belirtilmesi gerekmektedir. Veri alım altyapısının ikinci

bir işlevi de bu bölüm belirleme işlevini yapmaktır. Her gelen mesaj ayrıştırılarak IoT cihaz ID'si bulunmakta ve bu ID'ye göre mesaj kuyruğu bölümü bulunarak bu bölüme mesaj yazılmaktadır. Aşağıda mevcut veri alım altyapısındaki problemler ve bunlara karşı geliştirilen çözümler listelenmiştir.

Performans

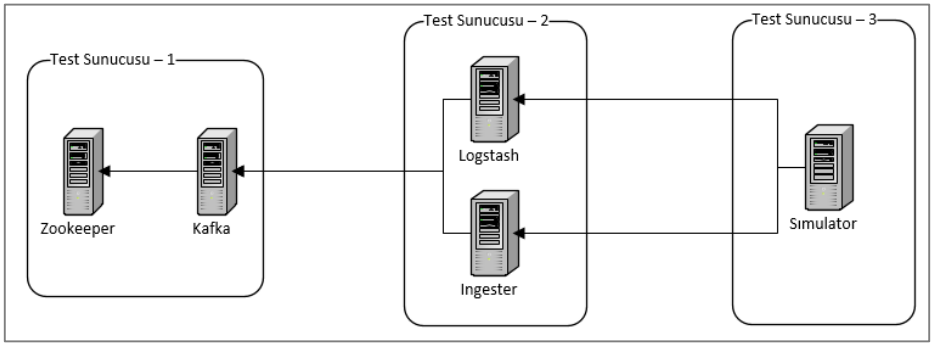
Veri alım altyapısında ilk etapta Elastic firmasının ürünü olan Logstash kullanılmıştır. Bu ürün çok farklı protokollerde ve formatlarda mesajları alarak çok farklı kaynaklara yazabilecek, Jruby dili ile yazılmış JVM (Java Virtual Machine) üzerinde çalışan bir üründür. Bir konfigürasyon dosyası üzerinden ayarları yapılarak çalıştırılmaktadır.

Genel olarak sistem performans testleri yapılması esnasında ürünün performans olarak yavaş kaldığı ve çok fazla CPU gücü tükettiği gözlemlenmiştir. Bu sebeple Google tarafından geliştirilen ve direk olarak makine kodu üreten bir dil olan Golang ile özel bir veri alım bileşeni olan "ingester" geliştirilmiştir.

Performans testleri için düzenlenen sistem Şekil 1 içinde gösterilmektedir. Test ortamında kullanılan makine özellikleri "Tablo 1. Test makine konfigürasyonları" içinde listelenmektedir.

Tablo 1. Test makine konfigürasyonları

	İşlemci	Bellek	Network
Test Sunucusu – 1	2x14 Core	384 GB	10Gb
Test Sunucusu – 2	1x4 Core	32 GB	1Gb
Test Sunucusu – 3	1x4 Core	32 GB	1Gb



Şekil 2- Veri alım altyapısı test ortamı

Örnek Veri. "Simülator" aracı ile 10.000.000 adet IoT verisi üretilmiştir. Bu veri bir text dosyasında saklanarak Linux işletim sisteminde bulunan "nc" komutu aracılığı ile test edilen veri alım bileşenlerine (Logstash ve Ingester) beslenmiştir.

Test Sonuçları. Aşağıdaki tabloda, örnek verinin veri alım sistemlerine beslenmesi ve mesaj kuyruğuna eksiksiz kayıt edilmesi arasındaki süre zarfında Test Sunucusu-2

makinasında oluşan ortalama CPU kullanım verileri listelenmiştir. CPU kullanım oranları "pidstat" ve "top" komutları kullanılarak hesaplanmıştır.

Tablo 2. Logstash ve Ingester performans testi sonuçları

	Logstash	Ingester	Oran
İşlenen Mesaj Sayısı	10.000.000	10.000.000	1
Toplam Süre	312 saniye	33 saniye	9.45
Toplam CPU Zamanı	38 dakika	1 dakika 20 saniye	28.5

Sonuç. Sonuçlardan görülebileceği üzere Ingester hem daha verimli çalışmakta hem de işlemi daha hızlı yerine getirmektedir. Fakat Logstash tarafından sağlanan esneklik, farklı kaynaklara erişim gibi özellikler bulunmadığı için bu özellikler geliştirilmek istenirse kod geliştirilmesi gerekecektir.

Yüksek Erişilebilirlik ve Yük Dengeleme

Veri alım altyapısının devamlı surette açık olarak veri almaya devam etmesi ve hatalara karşı tolere eden bir yapısının bulunması gerekmektedir. Yazılım altyapısındaki olası buglar, nadir karşılaşılan koşullar sonucu karşılaşılan hatalar, işletim sistemi ve donanım seviyesindeki hatalar, ani oluşan yükler bu altyapı tarafında düşünülmesi gereken durumlar olarak ilk etapta karşımıza çıkmaktadır. Bu tarz altyapılarda genel olarak önerilen çözüm yazılımın farklı makinalarda koşturularak hem yük dengelemesi yapabilmesi hem de bir makinada oluşan hata durumunda diğer makinalara otomatik olarak yükün paylaştırılmasıdır. Bu çözümler genel olarak en önde bir yük dengeleyici yazılım (veya donanım) kullanılarak gerçekleştirilmektedir. Yük dengeleyici yazılımlar genel olarak yükü çalışmakta olan makinalara doğru bir şekide dağıtsa da çalışmayan bir makinadaki duruma müdahale etmemektedirler. Sunulan sistem kapsamında, bu problem Kubernetes container orkestrasyon aracı çözülmeye çalışılmıştır.

Ubuntu 18.04 Linux dağıtımı baz alınarak oluşturulan bir Docker container imajı içine Ingester uygulaması kurulmuştur. Kubernetes altyapısı için kubeadm aracı kullanılarak, 3 adet sanal makine üzerine, 1 master – 3 slave olmak üzere bir kubernetes kümesi kurulmuştur. Ingester Docker imaji gene Kubernetes üzerinde koşan bir Docker registry üzerine yüklenerek, slave makinalar üzerinde dağıtılması sağlanmıştır. Daha sonra bu altyapı üzerinde Ingester imajı baz alınarak minimum 3 adet container çalıştırılmıştır (podlar). Kubernetes "service" soyutlaması sayesinde çalıştırılan bu containerların(pod) hangi makinada olduğuna bakmaksızın, istemcilere tek bir erişim noktası sağlamakta ve bunlar arasında yük dağılımı yapabilmektedir. Herhangi bir podda sıkıntı olması durumunda minimum 3 adet prensibine göre yeni bir container (pod) otomatik olarak ayağa Kubernetes tarafından kaldırılmaktadır.

Oluşturulan bu altyapı ile veri alım altyapısı hem yük dengeleme hem de yüksek erişilebilirlik özelliklerine kavuşmuştur.

Alternatif olarak dış bir proxy (haproxy, lvms, nginx) kullanılarak da yük dengelemesi yapılabilmektedir. Fakat bu altyapılar duran bir servisi tekrar ayağa kaldırma gibi özellikleri desteklememektedir.

3.3 Veri işleme katmanındaki karşılaşılan problemler

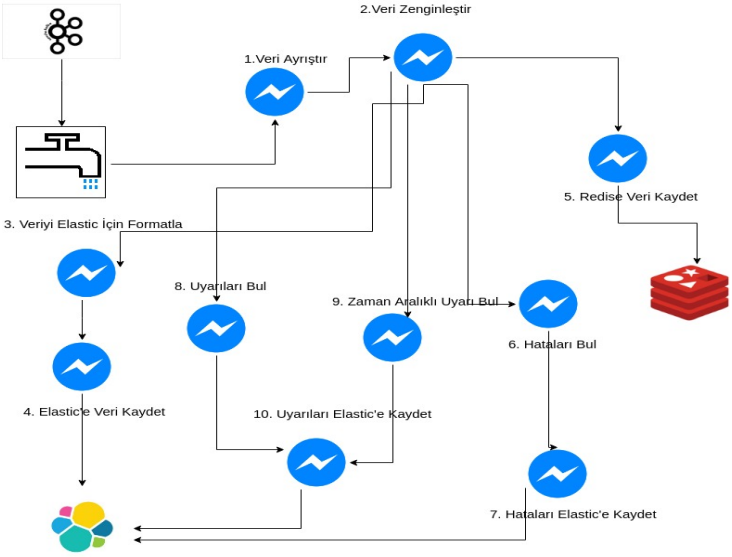
Dağıtık yakın gerçek zamanlı akan veri işleme (distributed real time stream processing), kesintisiz bir şekilde yüksek ölçekli veriler üzerinde hesaplamalara izin veren, uçtan uca işleme sürelerini düşüren ve yüksek çıktı üretme kabiliyeti sayesinde yoğun veri işleme alanında kabul görmüş ve gelişmeye devam eden büyük veri yaklaşımlarından en önemlisidir. [8] Bu yaklaşımı sektörde en çok tercih sebebi yapan ana sebep ise veri işleme sistemlerine gelen büyük veriler geldiği andan çok kısa süre içerisinde değerini kaybediyor olmasıdır[9]. Bu yüzden veriyi oluştuğu andan çok uzun süreler geçmeden işlemek için bahsedilen yaklaşımı kullanmak kaçınılmazdır. Günümüzde bu yeteneklerden faydalanarak sunulan birçok kullanım vardır. [10,11]. Fakat bu yeteneklerden tam anlamıyla yararlanabilmek için, geliştirilen sistemin güçlü bir mimariye sahip olması gerekmektedir. Güçlü bir mimari ise ilk günden tasarlanıp ortaya çıkartılması oldukça güçtür. Güçlü mimari karşılaşılan hatalara ve dar boğazlara çözümler ürettikçe kendiliğinden oluşacaktır. Akan veri işleme çözümlerinde mimari açısından iyileştirme süreçlerine başlamanın tek bir önemli adımı vardır. Bu adım, mevcut ortaya konan sistemin uçtan uca her bir bileşenin veriyi alıp görevini yerine getiriyor olması gerekmektedir. Bu bölümde geliştirilen sistemin [12] canlı ortama (production environment) alındıktan sonra karşılaşılan performans sıkıntıları ve bu sıkıntılara getirilen çözümler anlatılacaktır.

3.3.1 Bağımlılığı Yüksek (Tightly Coupled) Mimari

Yakın gerçek zamanlı veri işleme için Apache Storm [5] sistemi kullanılmıştır. Bu sistem, geliştirmek istenen hesaplama işini daha küçük iş parçacıklarına bölünmesine olanak sağlar. Her bir iş parçacığını DAG [13] (yönlü çevrimsiz çizge - directed acyclic graph) üzerindeki bir düğüm gibi işletmektedir. Oluşan bu DAG'a Storm terminolojisinde topoloji denilmektedir. Bu topoloji içerisinde yer alan her bir düğüm sıfır veya daha çok çıktı üretebilir ve bu çıktılar sıfır veya daha çok düğüm tarafından girdi olarak alınabilir.

Geliştirilen topolojilerde her bir düğüm aslında bir işlem sırası gözetilerek tasarlanmaktadır. Düğümlere bölünmüş olan bir işin, eksiksiz olarak tamamlanıp tamamlanmadığı da çoğu zaman bir topolojideki belirli sayıda düğümün veya bütün düğümlerin işlerini eksiksiz ve sorunsuz olarak gerçekleştirip gerçekleştirmediğine bakılarak karar verilir. Karar verme altyapısı özel olarak işaretlenmiş olan düğümlerden bir onay bilgisi bekler. Eğer ilgili onay bilgisi gelmezse, işlenen verinin topolojinin ilk giriş noktasından itibaren tekrar işletilmesini sağlar. Apache Storm'un bu özelliği garanti mesaj işleme imkanı tanımaktadır.

Yapmış olduğumuz bir çalışmada [12], tek topolojiden oluşan bir mimari izlenmişti. Garantili mesaj işleme alt yapısı kullanan bu topoloji, mesajları Kafka [4] dağıtık kuyruklama teknoloji üzerinden okuyup, yapması gereken işlem ve hesaplamaları yaptıktan sonra çıktıları Redis[14] ve Elasticsearch[2] içerisinde saklıyordu. Topoloji içerisinde üretilen sonuçlar ise web uygulaması aracılığı ile son kullanıcı ile paylaşılıyordu. Şekil 2'de bahsi geçen topolojinin yapısı verilmiştir.

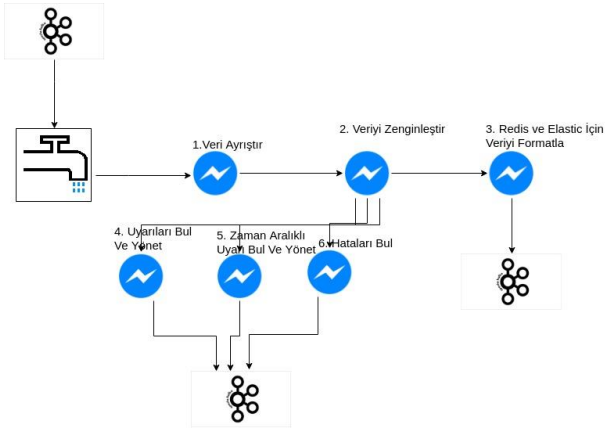


Şekil 3. [12] de sunulan topoloji yapısı

Topoloji yapısından sebep olduğu problem ve sunulan çözüm gerçek örnekler ile aşağıda anlatılmıştır.

Bahsi geçen mimaride birden çok farklı iş (low cohesive) tek bir topoloji içerisinde yapılmaktaydı. Şekil 2'de verilen {1,2,5}, {1,2,6,7}, {1,2,3,4}, {1,2,8,9,10} düğümler (Apache Storm terminolojisinde Bolt olarak isimlendirilmektedir.) aslında tamamen ayrı amaçları olan işlemler yapmaktaydı. Örneğin, bu yapının yarattığı en temel sorunlardan biri, Elasticsearch üzerinde herhangi bir sebepten dolayı bir sıkıntı olduğunda (ağ bağlantı hatası vb.) topolojide zamanla sıkışmalar oluyor ve zamanla kendini durduruyordu. Bu durumun gerçekleşme sebebi ise garanti mesaj işleme kuralları gereği her düğümün onay göndermesi gerekirken, Elasticsearch'e veri atmakla görevli olan düğümün bu işlemi yerine getirmemesiydi. Bunun sonucunda aslında Elasticsearch ile hiçbir bağlantısı olmayan, Redis'e veri kaydetmekle görevli olan düğümlerinde çalışmasını yukarıda bahsedilen sebepten dolayı durduruyordu. Bu sorunlar zinciri en son olarak son kullanıcıyı etkilenmesine sebep oluyordu. Redis üzerinden beslenen web uygulaması alanları da kullanılmaz hale geliyordu.

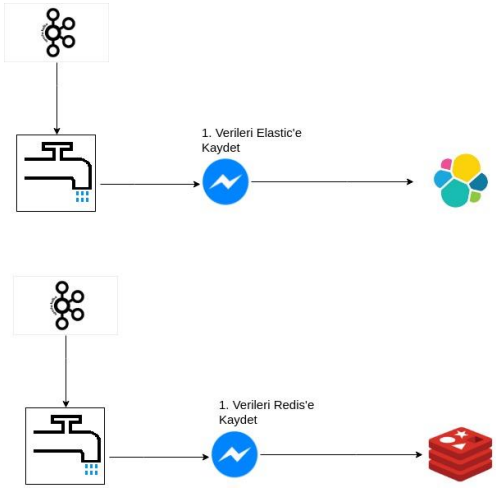
Yukarıda verilen sorunu çözmek için, mimari içerisinde ana iş kalemleri belirlendi. Ve bu ayrı işler için, ayrı topolojiler tasarlandı. Tek bir topolojiden üç ayrı topoloji geliştirildi. Bu topolojilerden biri verileri Apache Kafka üzerinden okuyup, veri üzerinde ön hazırlık (pre-processing) aşamalarını gerçekleştirecek şekilde geliştirildi. Bu topolojinin çıktıları ise tekrar Apache Kafka üzerinde depolandı. Şekil 3'de bahsi geçen topolojinin yapısı verilmiştir.



Şekil 4. Geliştirilen 1 numaralı Topoloji

Diğer iki topolojisi ise, Apache Kafka üzerinden işlenmiş (bir numaralı topolojinin çıktıları) olan verileri alıp, Redis ve Elasticsearch üzerinde ilgili yerlere kayıt işlemini gerçekleştiren topolojilerdir. Şekil 5'de bahsi geçen topolojilerin yapısı verilmiştir.

Yapılan bu mimarisel değişiklik ile bağımlılıklar görevlerine göre gruplanmıştır. Böylece tek bir hata noktasının bütün sistemi etkilemesi engellenmiştir. Kazanılan bu avantajın yanında, artık topolojiler ayrı ayrı konfigüre edilebilecek yapıya ulaşmıştır ve üstlenmiş oldukları yüklere göre ayrı ayrı kaynak ayrımı yapılabilecek dinamiklik kazanılmıştır.



Şekil 5. İşlenmiş Verileri Redis'e Aktaran Topoloji

3.3.2 Kafka Bölümlerinde (Partition) Dengesiz Mesaj Dağılımı

Genel sistem mimarisinde bahsedildiği gibi, Kafka saklamış olduğu mesajları kendi içerisinde bölümlere (partition) ayırarak saklar. Bu bölümlerde saklanan her bir mesaj değişmez (immutable) bir şekilde sıralı olarak saklanır. Bu bölümlendirme sayesinde Kafka üzerinden mesajlar hiç bir kaynak için yarışma durumu (race condition) problemleriyle uğraşmadan paralel olarak tüketilebilir. Kafka kendisine gönderilen mesajların hangi bölüme ait olduğunu kendisine verilen özel olarak geliştirilmiş bölümlendirme algoritması kullanarak veya ön tanımlı bölümlendirme algoritmasını kullanarak tespit eder. Bölüm 3.3.1'de yeni geliştirilen topolojiler arasında haberleşmenin Kafka üzerinden gerçekleştirildiğinden bahsedilmişti. Burada topolojilerin üretmiş olduğu ara değerler Kafka'ya gönderiliyor, diğer topolojiler ise Kafka'dan o mesajları okuyordu. Yapılan çalışmada ilk olarak Kafka'nın ön tanımlı bölümlendirme algoritması kullanılmıştı. Canlı ortama alınan topolojilerin performans açısından güzel sonuçlar üretmesine rağmen zamanla belirli Kafka bölümlerini tüketen iş parçalarında sıkışmalar olduğu gözlemlendi.

Bahsi geçen sıkışmanın sebep olduğu problem ve sunulan çözüm gerçek örnekler ile aşağıda anlatılmıştır.

Canlı ortama almış olduğumuz topolojilerde yukarıda bahsedildiği gibi zamanla sıkışmalar olduğu gözlemlendi. Bu durumun asıl sebebini anlamak için, ilk olarak her bir iş parçacığının performans metrikleri (CPU, hafıza tüketimi, garbage collector süreleri vb.) izlendi. İş parçacıklarının kendilerinin sebep olduğu bir hataya rastlanmadı. Ardından direkt olarak her bir bölümdeki toplam mesaj sayısı ve tüketilen mesaj sayıları izlenmeye başlandı. Burada dikkat çeken ayrıntı tüketilen mesaj sayıları her bir bölüm için yaklaşık eşit iken, toplam mesaj sayılarında kısa sürelerde büyük farklar oluşmaya başlıyordu. Problemin ana sebebini bulmak için, Kafka'nın ön tanımlı bölümlendirme algoritması incelendi. Bu bölümlendirme, mesaj ile verilen anahtar bilgisi üzerinden ilerleyen ve anahtarın murmur2 [15] anahtarlama (hashing) algoritması ile bir atama yapan algoritmadır. Bahsi geçen anahtarlama algoritmasının, Kafkaya anahtar olarak gönderilen değerleri 8 bölüm için hangi oranlarda dağıttığı test edildi. Ardından Java'nın String sınıfı için ön tanımlı olarak verdiği anahtarlama kodu aynı anahtar değerleriyle hangi oranda dağıttığı test edildi. Bu bilgiler tablo 3'de sunulmuştur. Buradan çıkarttığımız sonuç, sistem içerisinde anahtar olarak kullandığımız bilginin, murmur2 anahtalarma algoritması ile 8 bölüm için düzgün bir dağıtım yapmadığı olmuştur. Bu yüzden Kafka'nın ön tanımlı bölümlendiricisini kullanmak yerine, özel olarak verilmiş Java'nın String sınıfı için ön tanımlı olarak sunulan anahtarlama kodu kullanılmıştır. Bu değişikliğin canlı ortama atılmasından sonra, bahsi geçen problemle bir daha karşılaşılmamıştır.

Tablo 3. Bölümlendirme Dağılımı (6868 Farklı Anahtar Kullanılmıştır)

Bölüm Numarası	Murmur Anahtarlama	Java String Anahtarlama
0	885	860

1	899	861
2	887	861
3	864	860
4	862	858
5	855	856
6	803	855
7	813	857

Kafka tabanlı yapmış olduğumuz diğer bir değişiklik ise, topolojilerin birbirleriyle haberleşirken kullanabileceğimiz nesne sıralama (object serialization) algoritmaları performans açısından test edildi. Almış olduğumuz sonuçlar doğrultusunda Apache Avro[16] teknolojisi kullanılmış ve Kafka'nın ön tanımlı sunduğu yöntemlere göre elde edilmiş yüksek performans Tablo 4'de verilmiştir.

Tablo 4.Nesne Sıralama Türlerine Göre Performans Değerleri (Kayıt/Saniye)

Kafka Ön Tanımlı Byte Sıralama	7073
Kafka Ön Tanımlı Json Sıralama	2847
Avro Sıralama	70353

4 Sonuçlar ve Bulgular

Büyük veri ekosistemindeki Storm, Kafka, Elasticsearch, Redis gibi açık kaynaklı projeleri çalışmamızda kullanıyor olmamız, garanti mesaj işleme, mesajların belirli bir zaman penceresinde ele alınması, dağıtık kuyruklama yapısında mesajların saklanarak paralel olarak tüketilmesi ve verilerin bellek üzerinden performanslı olarak sorgulanması gibi fonskiyonlara hazır olarak ulaşmamızı sağlamıştır. Her ne kadar bu fonksiyonlar hazır olarak kullanılsa da gerçek ortamda (production environment), gerçek verilerle, yoğun yük altında ve uzun süre çalıştırıldığında bu altyapıların bazılarının peformans ihtiyaçlarına cevap vermediği, bazılarının mevcut kaynak kodlarında bulunan sıkıntılardan dolayı aykırı durumlarla karşılaşıldığı gözlemlenmiştir. Veri alım aşamasında hazır olarak kullanılabilecek açık kaynaklı altyapıların sahip oldukları, farklı protokollerden veri alma, farklı veri formatlarını kabul etme ve farklı hedef sistemleri besleme gibi özelliklerinden dolayı performans ve yüksek erişilebilirlik konularında bazı durumlarda bekleneni veremediği belirlenmiştir. Storm tarafından sunulan garanti mesaj işleme özelliği, yanlış tasarlanacak topoloji yapısıyla birlikte sistemi çıktı üretemeyen bir duruma düşürebileceği gözlemlenmiştir. Gerçek zamanlı veri işleyen sistemlerde mesajlaşma için kullanılan Kafka vb. kuyruklama sistemlerinde mesaj saklama formatlarının performansa büyük etkileri olduğu tespit edilmiştir.

Referanslar

1. Yongheng Wang, Kening Cao: A Proactive Complex Event Processing Method for Large-Scale Transportation Internet of Things. International Journal of Distributed Sensor Networks, Volume 2014.
2. Elasticsearch Homepage, https://www.elastic.co/products/logstash, son erişim 2018/05/10.
3. Golang, https://golang.org/, son erişim 2018/05/28.
4. Apache Kafka Homepage, http://kafka.apache.org/, son erişim 2018/05/11.
5. Apache Storm Homepage, http://storm.apache.org/, son erişim 2018/05/11.
6. Apache Lucene Homepage, http://lucene.apache.org/, son erişim 2018/05/11.
7. Docker Homepage, https://www.docker.com, son erişim 2018/05/11.
8. Carbone, P., Fóra, G., Ewen, S., Haridi, S., & Tzoumas, K. (2015). Lightweight asynchronous snapshots for distributed dataflows. arXiv preprint arXiv:1506.08603.
9. Zaharia, M., Das, T., Li, H., Shenker, S., & Stoica, I. (2012). Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. HotCloud, 12, 10-10.
10. Oger, M., Olmez, I., Inci, E., Kücükbay, S., & Emekci, F. (2015). Privacy Preserving Secure Online Advertising. Procedia-Social and Behavioral Sciences, 195, 1840-1845.
11. https://blog.keen.io/architecture-of-giants-data-stacks-at-facebook-netflix-airbnb-and-pinterest-9b7cd881af54 son erişim 2018 Mayıs 29
12. IoT Cihaz Verileri İçin Gerçek Zamanlı ve Ölçeklenebilir Büyük Veri Mimarisi
13. Wang, L. (2013). Directed acyclic graph. In Encyclopedia of Systems Biology (pp. 574-574). Springer New York
14. Redis Homepage, https://redis.io/, son erişim 2018/05/11
15. Wikipedia contributors. "MurmurHash." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 29 May. 2018. Web. 1 Jun. 2018.
16. Apache Avro Homepage, https://avro.apache.org/ son erişim 2018/05/11.