EGE ÜNİVERSİTESİ FEN BİLİMLERİ ENSTİTÜSÜ

(YÜKSEK LİSANS TEZİ)

RESTFUL SERVİSLER İÇİN ÖLÇEKLENEBİLİR BİR DEĞİŞİKLİK İZLEME ALTYAPISININ GELİŞTİRİLMESİ

Oğuzhan SALTIK

Tez Danışmanı: Prof. Dr. Oğuz DİKENELLİ

Bilgisayar Mühendisliği Anabilim Dalı

Sunuş Tarihi: 02.09.2020

Bornova-İZMİR
2020

Oğuzhan SALTIK tarafından **YÜKSEK LİSANS TEZİ** olarak sunulan "**RESTFUL SERVİSLER İÇİN ÖLÇEKLENEBİLİR BİR DEĞİŞİKLİK İZLEME ALTYAPISININ GELİŞTİRİLMESİ**" başlıklı bu çalışma EÜ Lisansüstü Eğitim ve Öğretim Yönetmeliği ile EÜ Fen Bilimleri Enstitüsü Eğitim ve Öğretim Yönergesi'nin ilgili hükümleri uyarınca tarafımızdan değerlendirilerek savunmaya değer bulunmuş ve **02.09.2020** tarihinde yapılan tez savunma sınavında aday oybirliği/oyçokluğu ile başarılı bulunmuştur.

Jüri Üyeleri:	**İmza**
Jüri Başkanı: Prof. Dr. Oğuz DİKENELLİ	. . . .
Raportör Üye: Doç. Dr. Rıza Cenk ERDUR	. . . .
Üye: Doç. Dr. Tuğkan TUĞLULAR	. . . .

EGE ÜNİVERSİTESİ FEN BİLİMLERİ ENSTİTÜSÜ

ETİK KURALLARA UYGUNLUK BEYANI

EÜ Lisansüstü Eğitim ve Öğretim Yönetmeliğinin ilgili hükümleri uyarınca Yüksek Lisans Tezi olarak sunduğum "**RESTFUL SERVİSLER İÇİN ÖLÇEKLENEBİLİR BİR DEĞİŞİKLİK İZLEME ALTYAPISININ GELİŞTİRİLMESİ**" başlıklı bu tezin kendi çalışmam olduğunu, sunduğum tüm sonuç, doküman, bilgi ve belgeleri bizzat ve bu tez çalışması kapsamında elde ettiğimi, bu tez çalışmasıyla elde edilmeyen bütün bilgi ve yorumlara atıf yaptığımı ve bunları kaynaklar listesinde usulüne uygun olarak verdiğimi, tez çalışması ve yazımı sırasında patent ve telif haklarını ihlal edici bir davranışımın olmadığını, bu tezin herhangi bir bölümünü bu üniversite veya diğer bir üniversitede başka bir tez çalışması içinde sunmadığımı, bu tezin planlanmasından yazımına kadar bütün safhalarda bilimsel etik kurallarına uygun olarak davrandığımı ve aksinin ortaya çıkması durumunda her türlü yasal sonucu kabul edeceğimi beyan ederim.

02/09/2020

İmzası

Adı-Soyadı

Oğuzhan Saltık

ÖZET

RESTFUL SERVİSLER İÇİN ÖLÇEKLENEBİLİR BİR DEĞİŞİKLİK İZLEME ALTYAPISININ GELİŞTİRİLMESİ

SALTIK, Oğuzhan

Yüksek Lisans Tezi, Bilgisayar Mühendisliği Anabilim Dalı
Tez Danışmanı: Prof. Dr. Oğuz DİKENELLİ
Eylül 2020, 69 sayfa

Geliştirilmekte olan yazılımlardaki veri miktarının her geçen gün artmakta olduğu günümüzde, kullanıcılar verilerin değişiminden haberdar olmak istemekte, bu da verilerin etkin ve ölçeklenebilir bir şekilde iletilmesini gerekli kılmaktadır. Üretilen verileri kullanmak isteyen uygulamalar, veriye erişim açısından sorun yaşamaktadırlar. Çoğunlukla veri üreten ve tüketen uygulamalar birbirinden ayrı ve birbirine erişemez durumdadır. Kullanılan yoklama tabanlı çözümler, değişimlerin tespit edilmesinde gecikmeye sebep olarak ölçeklenebilirlik ve performans kriterlerinde beklentileri karşılayamamaktadır. Bu problemler verilerin birden fazla alıcıya ölçeklenebilir bir şekilde yönlendirilerek sorunsuz bir şekilde entegrasyonunu sağlamaya yönelik bir mekanizmaya ihtiyaç duyulduğunu göstermektedir. Artan veri miktarındaki değişimleri takip edecek sistemlerin bu yükü karşılayabilmesi için dağıtık mimariler önem kazanmaktadır. Bu çalışmada, ihtiyaç duyulan mekanizma için abone/yayımcı tarafların birbirinden tamamıyla bağımsız olması durumuna, ölçeklenebilirlik ve hataya dayanıklılık özelliklerine önem verilerek bir prototip geliştirimi yapılmıştır. Geliştirilen prototip sayesinde bir Web kaynağını dinlemek isteyen istemcinin, ilgili kaynağa sistem aracılığıyla abone olması sağlanmıştır. Abone olunan kaynaklar sistem tarafından belirli aralıklarda sorgulanarak, değişiklik içeren kaynaklara ait değişim, geliştirilen prototip aracılığıyla bir yayımcı taraf kullanılarak tüm abonelere iletilmektedir. Yapılan gerçekleştirim, PubMed isimli bir medikal veri seti üzerinde oluşturulan bir kullanım durumu ile denenmiştir.

Anahtar Sözcükler: Restful Servis İzleme Altyapısı, Ölçeklenebilirlik, Yayıncı-Abone Mimari, Yazılım Aktörleri.

ABSTRACT

DEVELOPMENT OF SCALABLE RESTFUL SERVICE MONITORING INFRASTRUCTURE

SALTIK, Oğuzhan

MSc in Computer Engineering
Supervisor: Prof. Dr. Oğuz DİKENELLİ
September 2020, 69 pages

Today, where the amount of data in the software being developed is increasing day by day, users want to be informed about the change of the data, so this necessitates the efficient and scalable transmission of data. Applications that want to use the data produced have problems in terms of access to the data. Mostly, applications that produce and consume data are separate and inaccessible. The polling based solutions do not meet expectations in scalability and performance criteria by delaying detection of changes. These problems show that a mechanism to ensure that the data is seamlessly integrated and forwarding that data to multiple recipients by scalable way is needed. Distributed architectures are gaining importance in order for the systems to monitor the changes in the increasing amount of data to meet this load. In this study, a prototype was developed for the mechanism needed, with the emphasis on the independence of subscriber/publisher parties, scalability and fault tolerance. Through our prototype, the client who wants to listen to a web resource is provided to subscribe to the relevant resource through the system. The subscribed sources are queried at certain intervals by the system and the change of the sources containing the changes was forwarded to all subscribers by using a publisher through the developed prototype. The implementation has been tested with a use case created on a medical data set called PubMed.

Keywords: Scalable Restful Service Monitoring System, Scalability, Publish-Subscribe Architecture, Software Actors.

TEŞEKKÜR

Yüksek lisans eğitim hayatımda benden desteğini esirgemeyen, bilgi ve tecrübelerinden yararlandığım, yönlendirme ve bilgilendirmeleriyle çalışmamı bilimsel temeller ışığında şekillendiren saygıdeğer danışmanım Prof. Dr. Oğuz Dikenelli'ye, değerli görüş ve önerileri ile katkıda bulunan, hiçbir zaman yardımdan kaçınmayan ve desteğini esirgemeyen Yük. Müh. Burak Yönyül'e teşekkürlerimi sunarım.

Hayatıma girdiği andan itibaren karşılaştığım her zorlukta ve her anımda beni koşulsuzca destekleyen eşime, hayatım boyunca her türlü güçlükte elimi tutan ve başarılarımı gururla taşıyan aileme çok teşekkür ederim. Çalışmamı, insanlığı daha ileriye götürmesini ve dünyanın daha iyi bir yer haline gelmesine katkı sağlamasını hayal ettiğim, bu serüvende bize katılan en küçük yol arkadaşım olan biricik kızım Bilge Saltık'a adıyorum.

İÇİNDEKİLER

Sayfa

ÖZET vii

ABSTRACT ix

TEŞEKKÜR xi

ŞEKİLLER DİZİNİ xv

ÇİZELGELER DİZİNİ xvii

ALGORİTMALAR DİZİNİ xix

SİMGELER VE KISALTMALAR DİZİNİ xxi

1 GİRİŞ 1

2 MİMARİ BİLEŞENLER 4

2.1 Yayımcı/Abone Sistem 4

2.1.1 Niçin Yayımcı/Abone Sistemler Tercih Edilmelidir 4

2.1.2 Yayımcı/Abone Sistemlerin Kullanım Detayları 5

2.1.3 Yayımcı/Abone Sistemlerin Çeşitleri. 6

2.1.4 Kafka 7

2.1.5 Geliştirilen Prototip ile Bağlantısı 7

2.2 Aktör Tabanlı Eşzamanlılık 8

2.2.1 Aktör Modeli 8

İÇİNDEKİLER (devam)

Sayfa

2.2.2 Eşzamanlı Programlama Gerçekleştirimi 10
2.2.3 Aktörler ile Programlama 12
2.2.4 Eşzamanlı Uygulama Mantığı için Aktörler 13
2.2.5 Scala Programlama Dili 13
2.2.6 Akka 15
3 BENZER ÇALIŞMALAR 16
3.1 Literatürdeki Benzer Çalışmalarla Karşılaştırılması 27
4 DEĞİŞİKLİK İZLEME ALTYAPISININ GELİŞTİRİMİ 30
4.1 Değişiklik İzleme Altyapısının Mimari Yapısı 30
4.1.1 Kontrolcü Aktör 32
4.1.2 Denetçi Aktör 33
4.1.3 Planlayıcı Aktör 34
4.1.4 Getirici Aktör 35
4.1.5 Dedektör Aktör 36
4.1.6 Yayımcı Aktör 37
4.1.7 Tüketici Aktör 38
4.1.8 Ayrıştırıcı Actor 40

İÇİNDEKİLER (devam)

Sayfa

4.2 Değişiklik İzleme Mimarisinin Gerçekleştirimi 40

4.2.1 Yayımcı Uygulama 41

4.2.2 Abone Uygulaması. 42

5 DEĞİŞİKLİK İZLEME SİSTEMİNİN DEĞERLENDİRİLMESİ 43

5.1 Kullanım Alanı 43

5.2 Yayımcı Tarafı 44

5.3 Abone Tarafı 45

6 SONUÇ VE TARTIŞMA 48

KAYNAKLAR DİZİNİ 50

ŞEKİLLER DİZİNİ

Şekil Sayfa

3.1 Corona Mimarisi 16

3.2 Gerçek Zamanlı Web Sitesi Yayımcı Abone Sistemi 17

3.3 SQRT-C Sistem Mimarisi 20

3.4 Tarihçe Tabloları Kullanarak Gereksiz İlgili Olmayan Değişiklikleri Önleme Akışı Diyagramı 25

3.5 Paralel Arama Robotu Mimarisi 26

4.1 Prototip Değişiklik İzleme Altyapı Mimarisi 31

4.2 Yayımcı Uygulama Diyagramı 31

4.3 Abone Uygulama Diyagramı 32

ÇİZELGELER DİZİNİ

Çizelge Sayfa

3.1 Literatür Karşılaştırması 29

4.1 Kafka Konu(Topic) Yapısı 37

ALGORİTMALAR DİZİNİ

Algoritma Sayfa

1 Kontrolcü Aktör Mesajlaşma Algoritması 33

2 Denetçi Aktör Mesajlaşma Algoritması 34

3 Planlayıcı Aktör Mesajlaşma Algoritması 35

4 Getirici Aktör Mesajlaşma Algoritması 36

5 Dedektör Aktör Mesajlaşma Algoritması 37

6 Yayımcı Aktör Mesajlaşma Algoritması 38

7 Tüketici Akışı Algoritması 39

8 Tüketici Aktör Mesajlaşma Algoritması 39

9 Ayrıştırıcı Aktör Mesajlaşma Algoritması 40

SİMGELER VE KISALTMALAR DİZİNİ

Kısaltmalar	**Açıklama**
REST	Representational State Transfer *(Temsili Durum Transferi)*
SOA	Service Oriented Architecture *(Servis Odaklı Mimari)*
API	Application Programming Interface *(Uygulama Geliştirme Arayüzü)*
OOP	Object Oriented Programming *(Nesneye Dayalı Programlama)*
CPU	Central Processing Unit *(Merkezi İşlem Birimi)*
I/O	Input/Output *(Girdi/Çıktı)*
STM	Software Transactional Memory *(Yazılım İşlem Belleği)*
OTP	Open Telecom Platform *(Açık Telekom Platformu)*
JVM	Java Virtual Machine *(Java Sanal Makinesi)*
TCP	Transmission Control Protocol *(İletim Denetimi Protokolü)*
IP	Internet Protokol *(İnternet Protokolü)*
URL	Uniform Resource Locator *(Düzgün Kaynak Konumlandırıcısı)*
HTML	HyperText Markup Language *(Hiper Metin İşaretleme Dili)*
XHTML	Extensible HyperText Markup Language *(Genişletilebilir Hiper Metin İşaretleme Dili)*

Kısaltmalar	**Açıklama**
DDS	Data Distribution Service *(Veri Dağıtım Servisi)*
DCPS	Data Centric Publish Subscribe *(Veri Merkezli Yayımcı Abone)*
RTPS	Real Time Publish Subscribe *(Gerçek Zamanlı Yayımcı Abone)*
QoS	Quality of Service *(Hizmet Kalitesi)*
HTTP	Hyper-Text Transfer Protocol *(Hiper-Metin Transfer Protokolü)*
DB	Database *(Veritabanı)*
SoC	Seperation of Concerns *(Kaygıların Ayrılması)*
XML	Extensible Markup Language *(Genişletilebilir İşaretleme Dili)*
UUID	Universally Unique Identifier *(Evrensel Benzersiz Tanımlayıcı)*
JSON	Javascript Object Notation *(Javascript Nesnesi Gösterimi)*
UI	User Interface *(Kullanıcı Arayüzü)*

1 GİRİŞ

Günümüz dünyasında gerçek zamanlı bilgi, uygulamalar (iş, sosyal veya diğer türler) tarafından sürekli olarak üretilmektedir. Bu bilginin, birden fazla alıcı tipine güvenilir ve hızlı bir şekilde yönlendirilmesinin kolay yollarına ihtiyaç vardır. Çoğu zaman, bilgi üreten ve bu bilgiyi kullanan uygulamalar birbirinden ayrı ve birbirine erişemez durumdadır. Bu durum zaman zaman aralarında bir bütünleşme noktası sağlamak için bilgi üreticilerinin veya tüketicilerinin yeniden geliştirilmesine yol açmaktadır. Bu nedenle, bir uygulamanın her iki tarafındaki yazılımların yeniden yazılmasını önlemek için, üretici ve tüketici yapılarının sorunsuz bir şekilde entegrasyonunu sağlamaya yönelik bir mekanizma gereklidir[1].

Tercih edilen dağıtılmış bilgi işlem paradigması olarak geniş ölçüde savunulan geleneksel istek/cevap (request/response) paradigması, bu gereklilikleri karşılamak için yeterli değildir. Bu klasik çekme temelli (pull-based) yaklaşımda, anında bilgi güncellemesi gerektiren bir istemcinin sürekli olarak bilgi sağlayıcısını (yani sunucuyu) yoklaması gerekmektedir. Bu olay ise sunucuda kaynak çekişmesine, ağın aşırı yüklenmesine ve tıkanmasına yol açmaktadır. Birçok mobil uygulamada enerji, kısıtlı bir kaynak olduğu için gereksiz bilgi taleplerinden kaçınılmalıdır.

Çekme temelli bir çözüm, yukarıdaki senaryoları karakterize eden bilgi kaynaklarının yüksek dinamikliğini desteklememektedir. Çünkü yeni kaynaklar sadece kapsamlı bir şekilde ağ araştırması yapılarak keşfedilebilir. Ağ büyük olduğunda bu çok zor olabilir ve sürekli bir ağ erişiminin her zaman bulunmayacağı mobil ve kablosuz ortamlarda ağ araştırması yapmak mümkün değildir. Bu gereklilikler kullanıcılarda, itme temelli (push-based) bir yaklaşımda bulunan ve mümkün olduğu kadar yüksek düzeyde bilgi sunan bir bilgi yayma modeline duyulan ihtiyacı arttırmaktadır. Bu yaklaşım ise yayımcı/abone (publish/subscribe) paradigması ile çok iyi bir şekilde desteklenebilmektedir (Cugola et al., 2002).

Yayımcı/abone, uygulama bileşenleri arasında yaygın olarak kullanılan bir asenkron iletişim modelidir. Mesajların gönderenleri ve alıcıları birbirlerinden ayrıştırılır ve bir aracı, yayımcı/abone sistemle etkileşime girer. Bir abone (subscriber) talebini bir abonelik biçiminde kaydeder. Mesajlar yayımcılar tarafından yayımcı/abone sistemine yayımlanır. Sistem, yayımlanan mesajları aboneliklerle eşleştirir ve bir bildirim mekanizması kullanarak ilgilenen abonelere mesajları iletir(Li et al., 2011). Yayımcı/abone modeli kullanılarak alıcıların

[1]Nishant Garg, "Apache Kafka", PACKT Publishing

ve göndericilerin birbirinden bağımsız hareket etmesi ve asenkron iletişime geçebilmesi oldukça önem arz etmektedir. Ayrıca abone olunan konu (topic) sayısı ve konulardaki değişiklerin frekansı arttıkça klasik mimariler darboğaz yaratacağından ölçeklenebilir bir mimariye gereksinim duyulmaktadır.

Publish-subscribe mimarilerinin, günümüzde popülerliği giderek artan REST tabanlı sistemlere entegre edilebilir ve ölçeklenebilir olması; veri temsili, veriye erişim hızı ve kolaylığı açısından büyük avantaj sağlamaktadır. Son zamanlarda oldukça popüler olan servis odaklı mimari ve mikroservis mimarisi gibi yeni nesil mimariler sayesinde; modüllerin bağımsız bir birim olarak mevcut sistemlere entegrasyonu oldukça kolaylaşmaktadır.

SOA (Servis Odaklı Mimari), monolitik uygulamaları iş hedeflerine yönelik daha küçük modüllere ayırmanın bir yolu olması amaçlanan mimari tasarım stilidir. Bu modüller küçük uygulama servislerinden büyük işletme servislerine kadar geniş bir aralıkta bulunabilir[2]. Mikroservisler, bir uygulamayı gevşek bağlı (loosely coupled) servisler topluluğu olarak yapılandıran Servis Odaklı Mimari (SOA) tarzının bir çeşididir. Mikroservis mimarisinde, servisler ince taneli (fine-grained) ve protokoller hafif olmalıdır[3]. Mikroservis mimarisinin en önemli avantajlarından bir tanesi her servisin diğer servislerden bağımsız bir şekilde ölçeklenebilir olmasıdır. Bu çalışmada geliştirilen prototip ile servis odaklı mimariye ya da mikroservis mimarisine sahip olan uygulamaların, ölçeklenebilir Web kaynağı izleme sistemini mevcut altyapılarına yeni bir servis olarak kolaylıkla ekleyebilmesini planlanmaktadır. Ayrıca değişikliklerin monitör edilmesi için oluşturulan prototip, bir modül haline getirilerek her yazılımcının bu işlevselliği sağlaması için tekrar çaba harcamasının önüne geçilecektir.

Ölçeklenebilirlik ve hataya dayanıklılık prototip için önemli bir kriter olduğundan dolayı iş parçacığı (thread) kullanımının etkin bir şekilde gerçekleştirilmesi ve iş parçacıklarının birbirini bloklamaması (non-blocking) gereklidir. Bu tezde tüm bu kriterleri sağlayan ve iş parçacıklarını birer yazılım aktörü olarak soyutlayan Akka araç kiti kullanılarak, sistem fonksiyonelliği yazılım aktörlerinin birbirleri arasında asenkron mesajlaşması prensibi ile birbirine bağlanacaktır. Yazılım aktörlerinin kullanılması ile sistemin yüksek seviyede ölçeklenebilir ve hataya dayanıklı özelliklere sahip bir temele sahip olması amaçlanmaktadır.

---

[2] What is SOA?, https://www.tiempodev.com/blog/microservices-vs-soa/

[3] What are microservices?, https://blog.leanix.net/en/microservices-vs-soa

Tezde geliştirilen prototip sayesinde bir Web kaynağındaki değişklikten haberdar olmak isteyen istemci (client), dinlemek istediği Web kaynağına sistem aracılığıyla abone olmaktadır. Prototip, sisteme eklenen Web kaynağındaki değişikleri belirli aralıklarda kontrol etmeye başlayacaktır. Kontrol mekanizması sayesinde tespit edilen değişiklikler prototip tarafından bir yayımcı/abone sisteminde yayımlanmaktadır. Geliştirilen prototip, yayımcı/abone sisteminde aktif bir yayımcı rolüne sahiptir. Bu sayede tespit edilen değişikliklerin abone olan istemciye ya da istemcilere yayımcı/abone sistemi aracılığıyla iletimi sağlanmaktadır. Değişikleri izleme mekanizması, yayımcı/abone yapısı olmadan da sağlanabilmektedir. Fakat yayımcı/abone yapısının sağladığı asenkron iletişim, abonelik avantajları ve alıcı ile göndericinin birbirinden bağımsız ele alınması gibi avantajlar ölçeklenebilirlik ve sistemin mimari tasarımı için olumlu katkı sağlamaktadır.

Bu tez sayesinde, herhangi bir uygulamanın ya da mikroservisin, bir RESTful Web Servis endpoint'inde bulunan bir değişikliğe abone olması ve ilgili değişikliklerin asenkron, ölçeklenebilir, hataya dayanıklı bir şekilde yayımlanması sağlanarak monitor edilmesi amaçlanmaktadır. RESTful Web Servis endpoint izleme hakkında bulunan ölçeklenebilir, hataya dayanıklı ve asenkron iletişime sahip sistemlerin eksikliğini gidererek, yayımcı/abone mimari yapısı ile istenen konudaki güncel değişikliklerin izlenmesine etkin bir çözüm sunmaktadır. Geliştirilen prototip, sağlık bilimleri konusunda yapılan uluslararası çalışmalar, yayımlanan makaleler, en son gelişmelerin bulunduğu ücretsiz bir biyomedikal veritabanı olan PubMed[4] üzerinde oluşturulan bir kullanım durumu ile denenmiştir.

[4]PubMed, https://www.ncbi.nlm.nih.gov/pubmed

2 MİMARİ BİLEŞENLER

Ölçeklenebilirliği arttırmak için tasarlanmış ve etkisi gerçek sistemlerde test edilerek kanıtlanmış birçok uygulama, kütüphane ve desen bulunmaktadır. Oluşturulacak olan prototip için önem arzeden ve ölçeklenebilirliğine etki etmesi planlanan mimari bileşenler özenle seçilerek aşağıda tanımlanmıştır.

2.1 Yayımcı/Abone Sistem

İnternet, dağıtılmış sistemlerin ölçeğini önemli ölçüde değiştirmiştir. Dağıtılmış sistemler, konum ve davranışları sistemin ömrü boyunca büyük ölçüde değişebilen, potansiyel olarak tüm dünyaya dağıtılmış binlerce varlık içermektedir. Bu gereksinimler, uygulamaların dinamik ve ayrık doğasını yansıtan daha esnek iletişim modelleri ve sistemleri için ilgili talebi göstermektedir. Noktadan noktaya(point-to-point) ve senkron(synchronous) iletişim, katı ve statik uygulamalara yol açarak dinamik büyük ölçekli uygulamaların geliştirilmesini zahmetli hale getirmektedir. Uygulama tasarımcılarının yükünü azaltmak için, bu gibi büyük ölçekli ortamlardaki farklı varlıklar arasındaki bağ, uygun bir iletişim planına dayanan özel bir ara katman altyapısı tarafından sağlanmalıdır. Yayımcı/abone etkileşim şeması gittikçe daha fazla dikkat çekerek, büyük ölçekli ortamlarda gerekli olan gevşek bağlı(loosely coupled) etkileşim biçimini sağlamaktadır(Eugster et al., 2003).

Geliştirilen prototip içerisinde yayımcı/abone sistem kullanılarak, kullanıcıların değişikliklerden haberdar olmak istediği Web kaynaklarına abone olması sağlanmaktadır. Abone olunan her kaynak için güncellemeler kullanılara yayımcı/abone sistem aracılığıyla etkin bir şekilde ulaştırılmaktadır.

2.1.1 Niçin Yayımcı/Abone Sistemler Tercih Edilmelidir

Bir gönderici ile bir alıcı arasında doğrudan ve sıkı bir bağlantı gerektiren geleneksel istemci/sunucu sistemlerine göre yayımcı/abone sistemlerinin en belirgin avantajı, bilgi gönderenlerin ve alıcıların ayrıştırılmasından kaynaklanan güçlü ölçeklenebilirlik seviyesidir. Yayımcı/abone sistemlerde; bilgi göndericisi(bir yayımcı) bilgi abonesiyle sıkı bir şekilde bağlanmamaktadır çünkü ikisi arasında yer alan ve bilgi teslimi için vekil(proxy) rolünü üstlenen bir olay aracısı(event broker) bulunmaktadır. Hem asenkron zamanlama hem de konum saydamlığı(location transparency) sağlayan bir alan üzerindeki bu gevşek bağlanma

nedeniyle katılımcılar dinamik olarak eklenebilip, kaldırılabilmektedir. Sıkı bağlı katılımcılara sahip dağıtılmış bir uygulama tasarlamak zor bir iştir. Ayrıca, katılımcıların birbirinden farklı şekilde konumlandırılmış olmaları ve değişken programlama özelliklerine sahip olmaları muhtemel olduğundan, yapım süreci de ağır ve hataya eğilimli olabilmektedir. Buna karşılık, yayımcı/abone paradigması, bir noktadan bir noktaya iletişim modelinin (örneğin, bir istemci/sunucu iletişim modeli) kısıtlamalarının üstesinden gelebilecek esnek ve dinamik bir iletişim modelini desteklemektedir. Sonuç olarak, iletişim için bir yayımcı/abone paradigmasının kullanılması, büyük ölçekli heterojen dağılmış sistemlerin kurulmasını büyük ölçüde kolaylaştırmaktadır(Kim et al., 2015).

Yayımcı/Abone modelinin en temel avantajlarından birisi, yayımcıları ve aboneleri çeşitli boyutlarda ayrıştırmasıdır(Eugster et al., 2003):

1. Zaman ayrışması; yayımcıların ve abonelerin aynı anda çevrimiçi olmaları gerekmediği anlamına gelmektedir. Bu şekilde tarafların aynı anda aktif olarak etkileşime katılmaları gerekmemektedir.

2. Alan ayrışması; abonelerin ve yayımcıların birbirlerini tanımaları gerekmemektedir. Anonimlik özelliğini belirtmektedir.

3. Senkronizasyon ayrışması; herhangi bir tarafın diğerinin yanıt verene kadar engellenmemesi anlamına gelmektedir. Yayımcı/Abone sistemlerinin asenkron yapısını ifade etmektedir.

2.1.2 Yayımcı/Abone Sistemlerin Kullanım Detayları

Yayımcı/abone etkileşim paradigması, bir yayımcı tarafından oluşturulan ve kayıtlı ilgi alanlarıyla eşleşen herhangi bir etkinliğin ardından bildirilmek üzere bir olaya veya olay deseni üzerine abonelerin ilgilerini ifade etme olanağını sağlamaktadır. Diğer bir deyişle, üreticiler bir yazılım veriyolunda (bir etkinlik yöneticisi) bilgi yayımlamaktadırlar. Tüketiciler ise bu veriyolundan almak istedikleri bilgilere abone olmaktadırlar. Bu bilgi tipik olarak olay(event) veya bildirim terimi ile ifade edilmektedir. Böyle bir servis, olay üreticileri ve aboneler arasındaki tarafsız bir arabulucuyu temsil etmektedir. Ara katman sisteminin görevi, abonelikleri kaydetmek ve olayları işlemek, bunları kaydedilmiş aboneliklerle eşleştirmek ve bir eşleşme durumunda ilgi alanlarıyla eşleşen olayların abonelerine bildirmektir(Kermarrec and Triantafillou, 2013). Bu bildirim şekli asenkron olarak gerçekleşmektedir(Eugster et al., 2003). Aboneler, olayların gerçek kaynaklarını

bilmeden, etkinlik hizmetinde bir subscribe() işlemi çağırarak olaylara olan ilgilerini bir olay aracısı sistemine(event brokering system) tanımlamaktadırlar(Kim et al., 2015). Bu abonelik bilgisi etkinlik hizmetini sağlayan sistemde saklanarak yayımcılara iletilmemektedir. Tam tersi bir işlem olan unsubscribe(), bir aboneliği sonlandırmak için kullanılmaktadır. Bir etkinlik oluşturmak için yayımcı genellikle publish() operasyonunu çağırmaktadır. Etkinlik servisi, olayları tüm ilgili abonelere yaymakla görevlidir. Bu nedenle aboneler için vekil(proxy) olarak görülebilir. Her bir abone, ilgilendiği olaylarla ilgili bilgilendirilir (başarısızlıklar abonelerin bazı etkinlikleri almasını engelleyebilmektedir)(Eugster et al., 2003).

2.1.3 Yayımcı/Abone Sistemlerin Çeşitleri

Literatürde geçen iki tür yayımcı/abone şeması bulunmaktadır: (i) Konu Temelli Şemalar (Topic Based), (ii) İçerik Temelli Şemalar (Content Based).

Konu temelli yayımcı/abone sistemlerinde [5], tüm olaylar önceden tanımlanmış konu kanalları(topic channels)(Eugster et al., 2003) aracılığıyla yayımlanır ve alınır. Bu ise konu tabanlı yayımcı/abone sistemlerini grup iletişimine benzer hale getirmektedir. Kanal tabanlı iletişim olarak da adlandırılan konu tabanlı abone/yayımcı sistemler ile oluşturulmuş birçok açık kaynaklı ve endüstriyel ürün bulunmaktadır. Ayrıca endüstride yaygın olarak kullanılmaktadır ve kavramsal olarak içerik tabanlı yayımcı/abone sistemlerden kullanım açısından daha kolaydır. Yayımcılar yayınlarını bir kanal adıyla etiketler ve o kanala abone olan tüm kullanıcılar ilgili mesajı almış olur. Basit veri modeline rağmen, konu tabanlı yayımcı/abone, trafik uyarı sistemleri, mobil cihaz bildirim çerçeveleri (örneğin, mobil cihazlara push bildirimleri göndermek için kullanılan Google Cloud Messaging gibi), anlık mesajlaşma sistemleri, olağanüstü hava durumu alarm sistemleri, Twitter ve daha birçok uygulama alanında yaygın olarak kullanılmaktadır(Gascon-Samson et al., 2015).

İçerik temelli yayımcı/abone sistemlerinde; tüm olaylar mevcut etkinlik içeriğine göre yayımlanır ve alınır. İlgi alanı olarak yayımların içeriğine filtre olarak tanımlanabilen abonelikleri desteklemektedir. Genel olarak bu tanımlar, her bir yayımla ilişkilendirilen nitelikler üzerinden belirtilen koşullar yoluyla gerçekleşmektedir(Barazzutti et al., 2014), (Altınel and Franklin, 2000), (Carzaniga et al., 2001), (Chan et al., 2002). Yayımlar bir takım öznitelik/değer çiftleri ile etiketlenir ve abonelikler öznitelikler üzerine belirtilen koşullar şeklinde ifade

[5] Amazon SNS, http://aws.amazon.com/sns/

edilir(Pietzuch and Bacon, 2002). Öznitelik değerleri abonelik koşulunu sağlarsa, yayın ilgili abonelikle eşleşir. Ayrıca, yalnızca verinin özniteliklerinde değil, doğrudan veri üzerinde hesaplanan ayrıntılı koşullar kullanılarak aboneliklerin yapılabileceği içerik tabanlı sistemler de mevcuttur(Rosenblum and Wolf, 1997). Başka bir deyişle, olaylar önceden tanımlanmış bir konuya göre değil, özelliklerine göre sınıflandırılmaktadır. Bu nedenle, olay eşleştirme, içerik tabanlı yayımcı/abone sistemlerinde önemli bir tasarım konusudur. Çok büyük ölçekli yayımcı/abone sistemlerinde oluşan darboğazın olay eşleştirme(event matching) süreci olduğu iddia edilmektedir. İçerik tabanlı yayımcı/abone sistemler son 15 yılda araştırma topluluğu tarafından çok fazla ilgi görmüştür. Çünkü eşleştirme, hesaplama açısından yoğun ve dağıtılması zor olan zorlu bir görevdir. Ayrıca genellikle altyapıya ek bir katman eklediğinden dolayı gecikme sınırları düşük uygulamalar için elverişsizdir(Gascon-Samson et al., 2015). Yayımcı/abone sistemlerinin performansı olay eşleştirme yapısının etkin bir şekilde gerçekleşmesine bağlıdır(Kim et al., 2015).

2.1.4 Kafka

Kafka, düşük gecikmeli yüksek hacimli günlük(log) verilerinin toplanması ve iletilmesi için geliştirilen dağıtık bir mesajlaşma sistemidir(Kreps et al., 2011). Ölçeklenebilir, dağıtık ve yüksek verimli bir sistem sağlarken, geleneksel günlük toplama ve mesajlaşma kombinasyonunu sunmaktadır. Endüstride popüler olarak kullanılan Kafka, çoklu mesajlaşma modeli yetenekleri(Thein, 2014), veri akışlarını gerçek zamanlı olarak verimli bir şekilde işleme yeteneği ve bunları sürekli olarak dağıtık ve çoğaltılmış şekilde saklama yeteneği ile bir dizi bağlayıcı API tarafından yüksek verimi(high throughput) koruyarak yayımcı/abone sisteminin ötesinde özellikler sunmaktadır(Zupan et al., 2017).

2.1.5 Geliştirilen Prototip ile Bağlantısı

Uygulamaların yeni veriler için her seferinde veritabanını sorgulamaya yönlendirmesi etkin bir davranış değildir. Uygulamalarda eski verilerin bulunmasına yol açan uzun yoklama aralıkları ya da depolama sisteminin iş yüküne müdahale eden sık yoklama işlemleri arasında karar vermesi gerekmektedir. Güncellemeleri tanımlayarak, ilgilenen uygulamalara bildirimler şeklinde ileten yayımcı/abone sistemler daha ölçeklenebilir bir çözüm sunmaktadır(Sharma et al., 2015). Oluşturulan prototipte izlenecek olan Web kaynaklarına ait güncellemelerin her seferinde belirli aralıklarda veritabanından sorgulanarak elde

edilmesi uygulama tasarımı açısından etkin bir çözüm değildir. Bundan dolayı güncellemeleri tüm abonelere sadece yeni bir güncelleme geldiğinde iletecek olan bir yayımcı/abone sistemi aracılığıyla gerçekleştirmek en etkin çözüm olacaktır. Bunun için geliştirilecek olan prototipte Apache Kafka sisteminin kullanılması tercih edilmiştir.

2.2 Aktör Tabanlı Eşzamanlılık

Şimdiye kadar düşündüğümüz eşzamanlılık modelleri, ortak paylaşılan durum(shared state) kavramına sahiptirler. Paylaşılan duruma aynı anda birden fazla iş parçacığı tarafından erişilebilir. Bu nedenle, kilitleme(locking) veya işlemler(transactions) kullanarak korunmalıdır. Değişebilirlik(mutability) ve durumun paylaşılması hem bu modeller için hem de karmaşıklıklar açısından normal bir durumdur. Ortak durum fikrini tamamen yasaklayan tamamen farklı bir yaklaşım mevcuttur. Durum hala değiştirilebilirdir ancak sadece aktör olarak adlandırılan tekil varlıklarla bağlantılıdır [6].

2.2.1 Aktör Modeli

Aktör modeli onlarca yıl önce o zamanlar mevcut olmayan bir ortam olan yüksek performanslı bir ağda paralel işlemeyi gerçekleştirmenin bir yolu olarak Carl Hewitt tarafından önerilmiştir. Bugün, donanım ve altyapı olanakları Hewitt'in vizyonunu yakalayıp aşmıştır. Sonuç olarak, zahmetli gerekliliklere sahip dağıtık sistemleri inşa eden kuruluşlar, geleneksel bir nesne yönelimli programlama (OOP) modeliyle tam olarak çözülemeyen zorluklarla karşılaşmaktadır. Fakat bu zorlukların çözümünde aktör modelinden faydalanabilmektedir.

Günümüzde, aktör modeli son derece etkili bir çözüm olarak tanınmakla birlikte dünyanın en zorlu uygulamalarının bazıları için üretim ortamlarında çalışabilirliği kanıtlanmıştır.[7].

Yaygın programlama pratikleri, zorlu modern sistemlerin ihtiyaçlarını tam olarak karşılamamaktadır. Aktör modeli bu eksiklikleri çeşitli prensipler ışığında ele alarak, sistemlerin zihinsel modelimize daha iyi uyacak şekilde

---

[6] Actor-based concurrency, https://berb.github.io/diploma-thesis/original/054_actors.html

[7] Why modern systems need a new programming model, https://doc.akka.io/docs/akka/current/typed/guide/actors-motivation.html

davranmasını sağlamaktadır. Aktör modelindeki soyutlama, iletişim açısından kodun düşünülmesini sağlamaktadır.

Aktörlerin kullanımı ile [8]:

- Kilitlere başvurmadan enkapsülasyon uygulanır.
- Tüm uygulamayı ileriye götürmek için sinyallere tepki veren, durumu değiştiren ve birbirlerine sinyaller gönderen kooperatif varlıklar modelini kullanılır.
- Dünya görüşümüze daha uygun bir yürütme mekanizmasına sahip olunur.

Aktör modelinin eşzamanlılık modellemesinde ve mesaj iletimi kavramları ile ilgili teorik kökleri vardır. Aktör modelinin temel fikri, aktörleri farklı şekillerde mesaj almalarını sağlayarak eşzamanlı ilkel yapılar olarak kullanmaktır. Mesaj alma şekilleri şunlardır:

1. Diğer aktörlere sınırlı sayıda mesaj göndererek
2. Sınırlı sayıda yeni aktör oluşturarak
3. Bir sonraki gelen mesaj kullanıldığında geçerli olan kendi iç davranışını değiştirerek

İletişim için, aktör modeli asenkron mesaj aktarma biçimini kullanmaktadır. Özellikle, kanallar gibi herhangi bir ara birimi kullanmaz. Bunun yerine, her aktör bir posta kutusuna sahiptir ve adresi bulunur. Bir aktör mesaj gönderdiğinde, alıcının adresini bilmesi gerekir. Ayrıca aktörlerin mesajı alacağı ve bir sonraki adımda işleyeceği şekilde birbirine mesaj göndermesine izin verilmektedir. Adreslerin ve aktörlerin ilişkilendirilmesinin kavramsal modelin bir parçası olmadığı unutulmamalıdır. Bu ilişkilendirme yapısı çeşitli gerçekleştirimlerde ele alınan bir özelliktir.

Mesajlar asenkron olarak gönderilir ve alıcının posta kutusuna ulaşması uzun sürebilir. Ayrıca, aktör modelleri, mesajların sıralanmasına dair herhangi bir garanti vermez. Posta kutusundaki iletilerin sıraya alınması ve çıkarılması

[8] How the Actor Model Meets the Needs of Modern, Distributed Systems, https://doc.akka.io/docs/akka/current/typed/guide/actors-intro.html

atomik işlemlerdir, bu nedenle bir yarış durumu(race condition) olamaz. Bir aktör, posta kutusuna gelen mesajları, yukarıda belirtilen tepki olasılıklarını kullanarak sırayla işler. Kendi içsel davranışını değiştirme ihtimali ise sonunda değişken durumla(mutable state) başa çıkmayı sağlar. Ancak, yeni davranış yalnızca geçerli mesaj işlendikten sonra uygulanır. Bu nedenle, her mesaj işleme operasyonu hala kavramsal bir perspektiften bakıldığında yan etkisi(side-effect) olmayan bir işlemi temsil eder. Aktör modeli, doğal olarak eşzamanlı(concurrent) sistemleri modellemek için kullanılabilir, çünkü her bir aktör diğerlerinden tamamen bağımsızdır. Paylaşılan ortak bir durum bulunmaz ve aktörler arasındaki etkileşim tamamen asenkron mesajlara dayanmaktadır.

Geliştirilen prototipte aktör tabanlı eşzamanlılığı sağlamak amacıyla aktör modeli kullanılmıştır. Bu sayede uygulamanın dikey ve yatay bir şekilde kolaylıkla ölçeklenebilmesi sağlanmıştır.

2.2.2 Eşzamanlı Programlama Gerçekleştirimi

Eşzamanlı sistemler için teorik bir modelin yanı sıra, aktörlerin altındaki fikir aynı zamanda bir eşzamanlı programlama modelinin planını temsil eder. Tamamen işlevsel adaptasyonlardan nesne yönelimli paradigmanın uzantılarına kadar çeşitli kavramsal uygulamalar göz önünde bulundurulmuştur (Agha, 1990). İlk olarak, eşzamanlılık için aktör modelini dahil eden (oldukça popüler bir programlama dili olan) Erlang (Vinoski, 2007) olmuştur. Aktör modeli son zamanlarda giderek daha popüler hale gelmiştir ve çoğu kez birinci sınıf dil konsepti olarak birçok yeni programlama diline dahil edilmiştir. Diğer birçok dilde, aktör modeli, geleneksel çoklu iş parçacıkları üzerine kurulu üçüncü taraf kütüphaneleri aracılığıyla kullanılabilmektedir.

Aktör modelini uygularken, orijinal fikir tarafından tanımlanan kurallara uymak önemlidir. Öncelikle ve en önemlisi, oyuncular hiçbir durumu paylaşmamalıdır. Bu kural, aktörlerin bir mesajın parçası olarak referansları, işaretçileri veya başka herhangi bir paylaşılan veriyi iletmesine izin vermemektedir. Aktörlerin yalnızca değişmez verileri ve adresleri gönderilmelidir. Aktörler arasında gönderilen mesajlar, en iyi çaba(best-effort) tarzına kıyasla, birkaç garanti ile daha zenginleştirilir. Çoğu gerçekleştirimler, bir aktörden diğerine gönderilen iki mesajın varış sırasını korumasını sağlar. Mesajlaşma her zaman asenkrondur ve birden fazla aktör tarafından gönderilen mesajların karışması belirsiz bir ihtimal olarak bulunmaktadır.

Mesaj işleme için çoğu gerçekleştirim, desen eşleşmesini(pattern matching) sağlar. Bu yapı, geliştiricinin farklı mesaj türlerini ayırt etmesini ve mesaj türüyle ilgili farklı kullanım kodları belirtmesini sağlamaktadır. Mesajları almak, aktörün bakış açısından engelleyici(blocking) bir işlem olsa da, yeni mesajları göndermek hiçbir zaman engelleyici değildir(non-blocking). Bazı gerçekleştirimler ayrıca seçici mesaj alma semantiği sağlamaktadır. Davranışına bağlı olarak, bir aktör posta kutusunda belirli bir mesajı bekleyebilir ve diğerlerini geçici olarak erteleyebilir.

Temel çalışma zamanı platformu, sınırlı sayıda CPU çekirdeğine ve kaynağına çok sayıda aktör tahsis etmeli ve bekleyen mesajlara sahip aktörlerin yürütülmesini planlamalıdır. Atomik davranış sadece posta kutusu işlemleri için gerekli olduğundan çoğu sistem prensip olarak kilitlenmeyen(lock-free) bir uygulama kullanmaktadır. Örneğin, Erlang sanal makinesi tek bir işlem başlatır ve mevcut çekirdek sayısına bağlı olarak bir iş parçacığı havuzu oluşturur (Larson, 2008). Dahili zamanlayıcı(scheduler), aktörlerin süreç içi kuyruklarını düzenler ve önleyici(preemptive) bir şekilde çalışır. Çalışan bir aktör, bir iletiyi işlediğinde veya belirli sayıda fonksiyon çağrısı gerçekleştirdiğinde, zamanlayıcı çalışmaya hazır olan bir sonraki aktöre geçer. I/O işlemleri için Erlang sanal makinesi, ayrık işletim sistemi iş parçacıkları üretir. Azalan limitler ve arkaplan I/O işlemleri adalet(fairness) ve canlılığı(liveness) arttırır çünkü hiçbir aktör CPU süresini daha uzun süre bağlayamaz.

Aktör modelinin pek çok gerçekleştirimi, aktör modelinin dalları olan ek özellikler sağlar, yani dağıtım desteği(distribution support) ve hata toleransı(fault tolerance). Eşzamanlılık kavramları normalde tek makineleri hedefler. Aktör modeli mesajlaşma için çok fazla garanti vermemektedir. Asenkron, sınırsız mesajlaşma, aslında ağ tabanlı iletişime benzemektedir. Aktörler, konum şeffaflığını(location transparency) kolayca sağlayabilen adresleri kullanarak tanımlanır. Aktör modeli doğası gereği paraleldir, bu yüzden dağıtık yüklemeleri desteklemek için aktör modelinin gerçekleştirimlerini genişletmek çok kolaydır. Örneğin, dağıtılmış Erlang sistemleri bir Erlang sanal makinesini çalıştıran çoklu düğümleri kullanabilir ve şeffaf bir şekilde dağıtılmış mesajlar iletebilir.

Geleneksel iş parçacığı tabanlı eşzamanlılık, hata toleransı açısından zorlu bir mücadele vermektedir. Paylaşılan durum ve kilitleme ile birlikte öngörülemeyen zamanlama, kopyalama(replication) ve anlık görüntü alma(snapshotting) için çok karmaşık stratejiler gerektirmektedir. Aktör modeli, kopyalamayı çok daha kolaylaştıran diğer özelliklerle ortaya çıkmaktadır. Aktörlerin posta kutularında sıraya giren mesajlar ve izole edilmiş durumlar, anlık görüntülere ve günlüklere(logs) çok benzemektedir. Bir aktörün mesaj yönetimi tek iş parçacıklıdır

ve üstü kapalı bir kazanç sağlamaktadır. Erlang "çökmesine izin ver(let is crash)" felsefesini benimsemektedir. İzole olma ve hiçbir şey paylaşmama özelliği, tek bir aktörün diğer aktörleri etkilemeden başarısız olmasına izin verir. Dahası, aktör modelinin kendisi, denetleme için aktörlerin hiyerarşi ağaçlarını oluşturarak hata toleransı için kullanılabilir. Bir aktör çöktüğünde, denetleyici aktör bir mesaj alır ve tepki verebilir. Bir süpervizör, aktörü yeniden başlatabilir, diğer aktörleri durdurabilir veya kendi süpervizörüne bir hata mesajı gönderebilir.

Aktörler, durum üzerinde izole edici bir etkiye sahiptir ve paylaşılan değişken durumu etkili bir şekilde önlemektedirler. Ayrıca, eşzamanlı programlama için hiçbir kilit gerekmez. Bununla birlikte, kilitlenmeler ve yarış koşulları gibi eşzamanlılık sorunları, bu uygulama modelinde, yanlış uygulamalarla yeniden ortaya çıkabilecekleri için, henüz tamamen kaldırılamamaktadır. Birbirinden bir mesaj bekleyen iki aktör döngüsel bir bağımlılığı temsil eder. Uygulamadaki yaklaşan kilitlenme, zaman aşımları(timeouts) kullanılarak önlenebilir. Aktörler tarafından gönderilen mesajların keyfi sıralanması, bazı geliştiriciler tarafından geleneksel bir yarış koşulu olarak yorumlanabilir. Fakat bu olay asenkronluğu test eden aktör modelinin karakteristik bir özelliğidir. Bundan dolayı bazı geliştiriciler, aktör modelinin temel fikirlerini görmezden gelerek ortaya çıkan yarış durumunu aslında uygunsuz uygulama tasarımının bir yansıması olarak görmektedirler.

2.2.3 Aktörler ile Programlama

Aktör modeli, kilitler veya STM ile iş parçacığı bazlı eşzamanlılıktan çok farklıdır. İzole edilebilir değişken durum ve asenkron mesajlaşma, iş parçacıklarının yapabildiği diğer programlama modellerini sağlar.

Aktörler, iş parçacıklarına göre çok hafiftirler. Minimum ek yük ile çoğaltılabilir ve yok edilebilirler. Bu nedenle, çok sayıda aktörü paralel olarak oluşturmak ve kullanmak mümkündür. Aktörler ayrıca bir mesaja cevap olarak karmaşık hesaplamalar yapabilirler. Aktörler, özyinelemeyi canlandıran mesajlaşma düzenlerine izin vererek kendilerine mesaj gönderebilirler. Ayrıca, aktörler adresleri bilinen diğer aktörlere mesajlar gönderebilirler; bu nedenle, aktör-temelli program aslında dinamik bir aktör ağıdır. Sonuç olarak, uygulama entegrasyonu için mevcut mesaj tabanlı modeller, aktörlerle de eşzamanlı programlama için kullanılabilecek kapsamlı bir desen kümesi sağlar. Yönlendirme, filtreleme, dönüşüm ve kompozisyon için popüler mesajlaşma modellerini içermektedir.

Değişken durumu izole etmek ve değişmez mesajların kullanımını zorlamak, üstü kapalı bir şekilde senkronizasyonu garanti etmektedir. Bir uygulama için, fikir birliği veya birden fazla aktör arasında ortak bir durum görüşü gerekebilir. Farklı bir uygulama işlevi sağlamak için birden fazla aktörün sıkı bir şekilde yönetilmesi gerektiğinde, doğru mesajlaşma çok zor olabilir. Bu nedenle, birçok uygulama, karmaşık mesaj akışlarına dayanan düşük seviyeli koordinasyon protokollerini uygulayan, ancak iç karmaşıklığı geliştiriciden gizleyen daha yüksek seviyeli soyutlamalar sağlamaktadır. Erlang için OTP, zengin bir soyutlama kümesi, genel protokol uygulamaları ve davranışları içeren standart bir kütüphanedir.

2.2.4 Eşzamanlı Uygulama Mantığı için Aktörler

Uygulama sunucuları aktör modelini kullandığı durumda, gelen her istek yeni bir aktörü temsil etmektedir. İstek işlemlerini paralelleştirmek için aktör, yeni aktörleri oluşturur ve mesajlar yoluyla onlara iş atar. Bu, paralel I/O sınırlı işlemlerin yanı sıra paralel hesaplamaları da mümkün kılar. Genellikle, tek bir talebin akışı, dağıt/topla(scatter/gather), yönlendirici(router), zenginleştirici(enricher) veya toplayıcı(aggregator) gibi mesajlaşma modellerini kullanarak, birden fazla aktör arasındaki az ya da çok karmaşık bir mesaj akışını temsil eder.

Aktör modelinin bir diğer özelliği de, yeni makineler ekleyerek aktör sistemini bir bütün olarak ölçeklendirme olasılığıdır. Örneğin, Erlang, sanal makinelerin dağıtık bir sistem oluşturmasını sağlar. Bu durumda, uzak aktörler izole edilmiş uygulama durumunu tutabilir, fakat sistemin diğer tüm aktörlerine mesajlaşma yoluyla erişilebilmektedir.

2.2.5 Scala Programlama Dili

Scala [9], JVM'de çalışan genel amaçlı, nesnesel ve fonksiyonel bir dildir. Java ile birlikte çalışır ancak gelişmiş ifade gücünü, ileri programlama kavramlarını ve fonksiyonel programlamanın birçok özelliğini içermektedir. Eşzamanlılık için Scala, aktör tabanlı eşzamanlılık modelini uygular ve değerlerin açık değişmezliğini(immutability) destekler. Bununla birlikte, Scala uygulamaları, Java programlama dilinin eşzamanlılık ilkelerine geri dönebilir.

Erlang düşük seviyeli birçok iş parçacığı meydana getirerek ve aktörleri

[9]Scala Programming Language, http://www.scala-lang.org/

çalıştırmak için özel bir zamanlayıcı uygularken, Scala JVM'in çok iş parçacıklı yapısından faydalanmaktadır. Ayrıca, Scala'nın aktör gerçekleştirimi dil çekirdeğinin bir parçası değil, standart kütüphanesinin bir parçasıdır. Bu, aktör kütüphanesinin kendisinin Scala'da uygulandığı anlamına gelir. Scala aktörlerinin ilk meydan okuması, JVM'deki çoklu iş parçacığı kısıtlamaları nedeniyle ortaya atılmıştır. Muhtemel iş parçacığı sayısı sınırlıdır, ortak bir zamanlama mevcut değildir ve kavramsal olarak aktörler iş parçacıklarından daha hafiftir. Sonuç olarak, Scala tek bir aktör kavramı sağlar fakat mesaj işleme için iki farklı mekanizma bulundurmaktadır(Haller and Odersky, 2006; Haller and Odersky, 2009).

İş parçacığı temelli aktörler alma(receive) ilkesi ile kullanıldığında dahili olarak özel bir iş parçacığı tarafından desteklenir. Bu açıkça ölçeklenebilirliği sınırlar ve yeni mesajları beklerken iş parçacığının askıya alınmasını ve engellenmesini gerektirir. Olay odaklı aktörler tepki(react) ilkesi ile kullanıldığında, aktörleri doğrudan iş parçacıkları ile birleştirmeyen olay odaklı çalıştırma stratejisine izin verilir. Bunun yerine, birden fazla aktör için bir iş parçacığı havuzu kullanılabilir. Bu yaklaşım, aktörü ve ona ait durum bilgisini içine almak için çağrılan fonksiyona tek bir argüman ile geçirilen bir değer kullanır. Bununla birlikte, bu mekanizmanın çeşitli kısıtlamaları vardır ve kontrol akışını gizler(Haller and Odersky, 2009). Kavramsal olarak, bu uygulanış biçimi, iş parçacığı havuzu tarafından desteklenen bir olay döngüsüne çok benzemektedir. Aktörler olay işleyicilerini temsil eder ve mesajlar olaylara benzer. Genel olarak, her aktör özel bir iş parçacığına bağlanamadığından tepki ilkesi tercih edilmelidir. Bu nedenle tepki ilkesi daha iyi ölçeklenebilirlik sonuçları vermektedir.

Mesajlaşma için Scala aktörlerinin sözdizimi Erlang stilini izler. Mesajların değişmez değerler olması gerekiyordu, ancak şu ana kadar bir zorunluluk getirilmemiştir. Genellikle, özel bir sarmalayıcı sınıfı olan vaka sınıfları(case class) kullanılır. Bunlar özellikle varışta mesaj türünü belirlemek için örüntü eşleme(pattern matching) kullanıldığında daha faydalıdır. Scala ayrıca, TCP/IP üzerinden iletişim kuran ve Java serileştirmesine dayanan uzak aktörler sağlayarak, aktörlerin dağıtımını da desteklemektedir[10].

Scala programlama dili alt yapısında sağladığı iş parçacığı yönetimi avantajları sebebiyle geliştirilen prototipte tercih edilmiştir.

[10]Concurrency in Scala, https://berb.github.io/diploma-thesis/original/054_actors.html

2.2.6 Akka

Ağları ve işlemci çekirdeklerini kapsayan ölçeklenebilir, esnek sistemler tasarlamak için birçok açık kaynaklı kütüphane bulunmaktadır. Akka, güvenilir davranış(reliable behaviour), hata toleransı ve yüksek performans sağlamak için düşük seviye kod yazmak yerine iş gereksinimlerini karşılamaya odaklanılmasına olanak tanımaktadır.

Pek çok yaygın uygulama ve kabul edilen programlama modelleri, modern bilgisayar mimarileri için sistemler tasarlamada, doğasında var olan önemli zorlukları ele almamaktadır. Başarılı olmak için, dağıtılmış sistemler, bileşenlerin yanıt vermeden çöktüğü, kablo üzerinde iz bırakmadan mesajların kaybolduğu ve ağ gecikmesinin değiştiği bir ortamda baş etmek zorundadır. Bu sorunlar, dikkatli bir şekilde yönetilen veri merkezi ortamlarında ve hatta sanallaştırılmış mimarilerde dahi düzenli olarak meydana gelir.

Bu gerçeklerle başa çıkmanıza yardımcı olmak için Akka şunları sağlamaktadır:

- Atomik veya kilit gibi düşük seviyeli eşzamanlılık yapıları kullanmadan çok iş parçacıklı davranış sağlar ve bu şekilde bellek görünürlüğü sorunları hakkında düşünmeye dahi gerek kalmaz.
- Sistemler ve bileşenleri arasında şeffaf bir şekilde uzaktan iletişim sağlar. Bu şekilde zor ağ kodlarının yazılmasına ve sürdürülmesine gerek kalmaz.
- Gerçek bir reaktif sistem oluşturulması için elastik, ölçeklenebilir özelliklere sahip yüksek kullanılabilir(high-availability) kümelenmiş bir mimari sağlar.

Akka'nın aktör modelini kullanması, eşzamanlı, paralel ve dağıtılmış doğru sistemler yazmayı kolaylaştıran bir soyutlama düzeyi sağlamaktadır.[11].

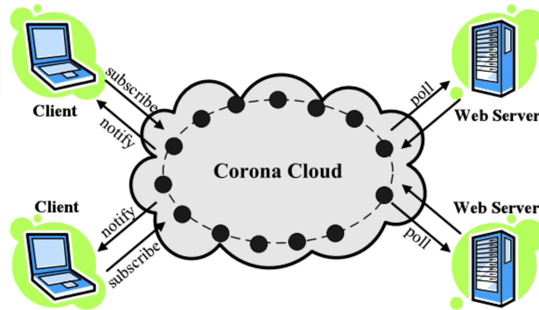
---

[11] Introduction to Akka, https://doc.akka.io/docs/akka/current/typed/guide/introduction.html

3 BENZER ÇALIŞMALAR

Bu bölümde geliştirilen prototip ile ilgili geçmişte oluşturulmuş benzer çalışmalar ve önerilen mimari yapılar ölçeklenebilirlik, değişiklik tespiti ve mimari tasarımları açılarından incelenmektedir.

Ramasubramanian ve arkadaşları, "Corona: A High Performance Publish-Subscribe System for the World Wide Web."(Ramasubramanian et al., 2006) isimli çalışmalarında geliştirdikleri yayımcı/abone prensiplerine dayanan sistem ile Web üzerindeki güncellemeleri tespit ederek bir anlık mesajlaşma uygulaması aracılığıyla kullanıcılara bildirmektedir. Tasarladıkları sistem ile kullanıcılar Web sayfalarına olan ilgilerini mevcut anlık mesajlaşma servisleri aracılığıyla belirtmektedirler. Corona, abone olunan Web sayfalarını izler, güncellemeleri verimli bir şekilde algılar ve kullanıcılara hızlı bir şekilde dağıtır. Yoklama için kaynakların tahsisi, içerik sunucularındaki yük sınırlarını aşmadan en iyi güncelleme performansını sağlayan dağıtılmış bir optimizasyon motoru tarafından yönlendirilir.



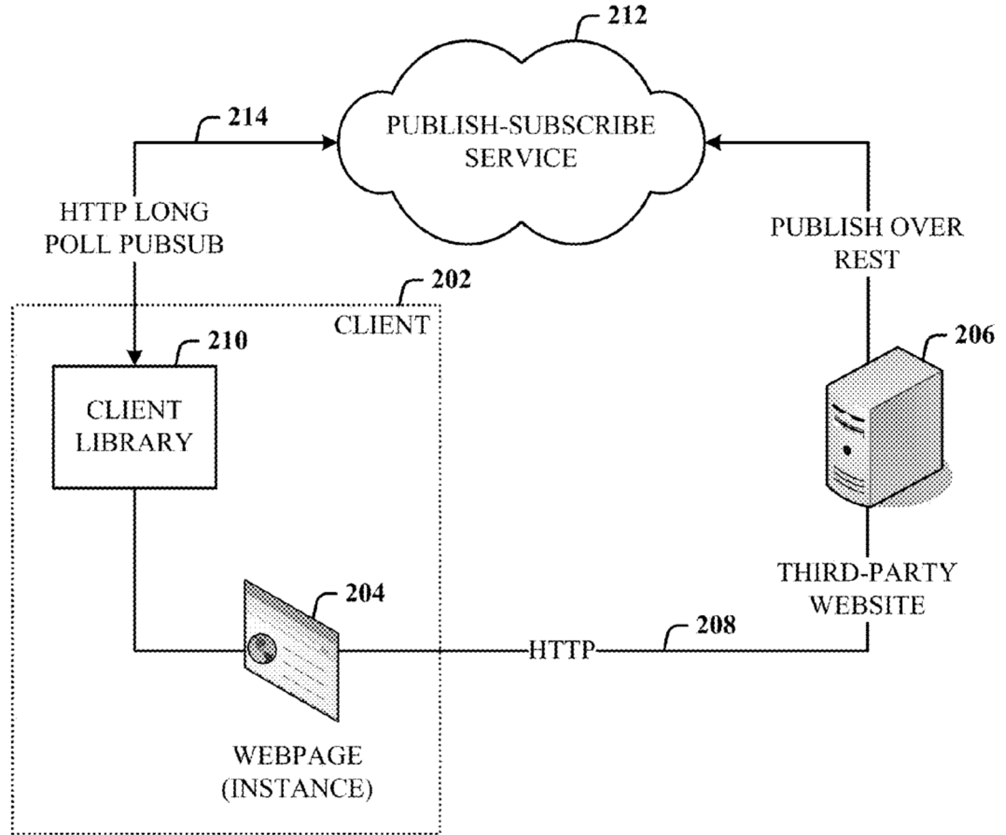
Şekil 3.1: Corona Mimarisi

Özelleştirilmiş bir fark motoru, kanaldaki temel içeriği keşfetmek için HTML veya XML içeriğini ayrıştırır ve zaman damgaları, sayaçlar ve reklamlar gibi sık sık değişen öğeleri filtreleyerek temel içeriği izole ettikten sonra bir güncelleme algılarsa bir delta kodu oluşturur. Tespit edilen değişikliklerin konu ile ilgili olup olmadığını belirler, değişen ilgili bölümleri çıkarır ve delta kodlu değişiklikleri anlık mesajlaşma(IM) sistemi yoluyla abone olan istemcilere dağıtır.

Corona'nın ön uç istemci arabirimi mevcut anlık mesajlaşma hizmetleri aracılığıyla sağlanır. Kullanıcılar kayıtlı bir Corona anlık mesajlaşma tanıtıcısına anlık ileti göndererek içerik için abone olur ve eşzamansız olarak güncelleme bildirimleri alır. Kullanıcılar, bir kanala abone olmak "`subscribe url`" ve bir abonelikten çıkmak için "`unsubscribe url`" biçiminde istek mesajları gönderir.

Corona tarafından yeni bir güncelleme tespit edildiğinde, kanaldan tüm abonelere anlık mesajlaşma sistemi içinde delta kodu ile bir anlık mesaj gönderilir. Birçok anlık mesajlaşma sisteminde, bir kerede yalnızca bir kullanıcının oturum açabilme sınırlaması bulunmaktadır, bu da tüm Corona düğümlerinin aynı anda oturum açmasını engelleyici bir faktör olarak bulunmaktadır.

Ravikant Cherukuri, "Realtime websites with publication and subscription"(Cherukuri, 2018) isimli çalışmasında bir web sayfasının gerçek zamanlı nesnelerini güncellemek için uzun yoklamaya dayalı yayımcı/abone modeli kullanan bir mimari oluşturmuştur. Her gerçek zamanlı nesne, pubsub hizmetindeki bir pubsub varlığıdır(entity). Web sayfasının her oluşturulması, bir sayfa nesnesine abonelik oluşturmaktadır. Yayımcı/abone hizmetindeki varlıklar, içeriklerin web sayfası nesnesine gerçek zamanlı olarak iletimini sağlamaktadır. Mimari hafif, gerçek zamanlı ve anonim bir yayımcı/abone modeli sağlamaktadır. Yayımcı/abone arka uç aracılığıyla web üzerinde ölçeklenebilmektedir ve mevcut Web sitesi koduna javascript aracılığıyla entegre edilebilmektedir.



Şekil 3.2: Gerçek Zamanlı Web Sitesi Yayımcı Abone Sistemi

Gerçek zamanlı bir web sayfasında, bir bölüm, başka bir bölümden

bağımsız olarak en son bilgilerle güncellenebilmektedir. Böylece kullanıcı arabirimine, olaylar gerçekleştikçe güncellemeleri bağımsız olarak uygulama olanağı sağlamaktadır. Gerçek zamanlı güncellemeler gerektiren Web sayfalarında çoğunlukla geleneksel olarak milyonlarca kullanıcıya ölçeklenmesi zor olan zamanlayıcı tabanlı bir yoklama tekniği kullanılmaktadır. Yayımcı/abone hizmeti, bildirim göndermek için tarayıcıdan gerçek zamanlı uzun yoklama bağlantısını işleyerek Web sayfasının anlık olarak oluşturulmuş örneklerini izler ve aktarımı gerçekleştirir.

Boldi ve arkadaşları, "Ubicrawler: A scalable fully distributed web crawler"(Boldi et al., 2004) isimli çalışmalarında geliştirdikleri ölçeklenebilir ve tamamen dağıtık bir şekilde çalışan UbiCrawler isimli arama robotu(crawler) aracından bahsetmektedirler. UbiCrawler, davranışlarını bağımsız olarak ve her biri Web'deki payını tarayacak şekilde koordine eden yazılım ajanlarından oluşmaktadır. Bir yazılım ajanı, görevini her biri tek bir ana makinenin ziyaretine ayrılmış birkaç iş parçacığı çalıştırarak gerçekleştirmektedir. Yazar çalışmasında iş parçacıklarının aynı anda farklı ana bilgisayarları ziyaret etmesini sağlamaktadır. Böylece her ana bilgisayar çok fazla istek tarafından aşırı yüklenmemiş olur. Belirtilen ana makinede yerel olmayan dış bağlantılar, doğru yazılım ajanına gönderilir. Yazılım ajanı ise bu bağlantıları ziyaret edilecek sayfa sırasına koymaktadır.

Ana bilgisayarların yazılım ajanlarına atanması, her yazılım ajanında bulunan yığın depolama kaynaklarını ve bant genişliğini dikkate alarak gerçekleşmektedir. Bu işlem şu anda, ana bilgisayarları dağıtmak için atama fonksiyonu tarafından kullanılan bir ağırlık işlevi gören kapasite adı verilen tek bir gösterge aracılığıyla yapılmaktadır. Belirli koşullar altında, her bir yazılım ajanı, kapasitesi ile orantılı olarak ana makinenin bir kısmına sahip olmaktadır. Ana makine başına URL sayısı aşırı bir şekilde değişse bile, URL'lerin yazılım ajanları arasındaki dağılımının büyük taramalar sırasında eşitlenme eğiliminde olmaktadır. Bunun ampirik istatistiksel nedenlerinin yanı sıra, bir ana bilgisayardan taranan maksimum sayfa sayısını ve ziyaretin maksimum derinliğini sınırlamak için politikaların kullanılması gibi başka yaklaşımlar da bulunmaktadır. Bu tür politikalara Web tuzaklarından(kötü amaçlı) kaçınmak için gereklidir.

Son olarak, UbiCrawler'ın temel bir bileşeni, çökmüş ajanları tespit etmek için zaman aşımları kullanan güvenilir bir arıza dedektörüdür (Chandra and Toueg, 1996). Güvenilirlik, çökmüş bir ajanın sonunda her aktif ajan tarafından şüphe duyulacağı anlamına gelmektedir. Hata dedektörü, UbiCrawler'ın tek senkron bileşenidir ve diğer tüm bileşenler tamamen eşzamansız bir şekilde etkileşime

girmektedir.

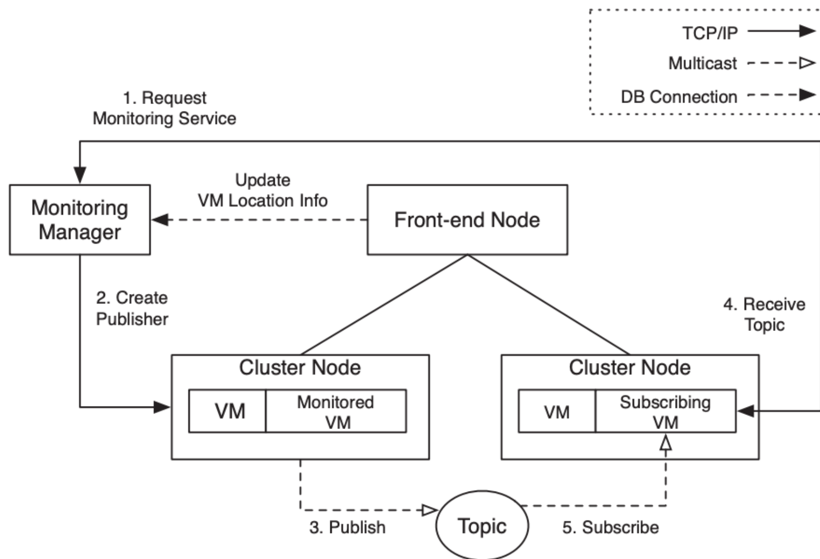
Her arama robotu iş parçacığı, bir sayfayı indirdikten sonra saklamadan önce o sayfayı ayrıştırması gerekmektedir. Hem taramanın devam etmesi için gerekli olan bağlantıları ayıklamak hem de diğer ilgili bilgileri elde etmek için ayrıştırma yapılmalıdır. UbiCrawler'ın mevcut sürümü, en yaygın hatalar üzerinde çalışabilen oldukça optimize edilmiş bir HTML/XHTML ayrıştırıcısı kullanmaktadır. Standart bir bilgisayarda, performans yaklaşık 600 sayfa/saniye'dir.

Yazarların araştırmasına göre, dağıtılmış arama robotlarının hata toleransını tahmin etmek için yaygın olarak kabul edilen bir ölçüm yoktur, çünkü hataların sebep olduğu sorunlar şimdiye kadar ciddi olarak dikkate alınmamıştır. Hata durumundaki paralel arama robotlarının sağlamlığını test etmek için bir dizi önlem tanımlamak gerçekten ilginç ve açık bir sorun olduğu için yazarlar UbiCrawler yazılım ajanlarının hatalara tepkisine genel bir bakış sunmaktadır. UbiCrawler yazılım ajanları ya beklenen (örneğin bakım için) ya da beklenmedik bir şekilde (örneğin bir ağ sorunu nedeniyle) ölebilir veya erişilemez hale gelebilir. Herhangi bir zamanda, her bir ajanın hangi ajanların canlı ve ulaşılabilir olduğu konusunda kendi görüşü bulunmaktadır ve bu görüşler hiçbir zaman çakışmamaktadır. Bir yazılım ajanı aniden öldüğünde, hata dedektörü kötü bir şey olduğunu keşfeder (örneğin zaman aşımlarını kullanarak). Atama fonksiyonunun özellikleri sayesinde, canlı yazılım ajanları kümesindeki ajanların farklı görünümlere sahip olması, tarama işlemini bozmaz. Bu tasarım seçiminin bir başka sonucu da, tarama sırasında ajanların dinamik olarak eklenebilmesidir ve bir süre sonra sorumlu oldukları tüm sayfalar, yeni ajanın doğumundan önce onları getiren ajanların mağazalarından kaldırılacaktır. Başka bir deyişle, tasarımla UbiCrawler'ın kendini dengelemesini sağlamak yazarlar için sadece hata toleransı sağlamakla kalmaz, aynı zamanda dinamik yapılandırma değişikliklerine daha fazla uyum sağlamasına da neden olmaktadır.

Eugster ve arkadaşları, "The many faces of publish/subscribe"(Eugster et al., 2003) isimli çalışmalarında yayımcı/abone paradigmasının, RESTful API'leri kullanan mevcut izleme ve yayma mekanizmalarındaki sınırlamaları aşabileceğini düşünmektedir. Ayrıca, bilginin yaygınlaştırılmasındaki güvenilirlik(yani performans ve zamanındalık) temel bir ihtiyaç olduğundan dolayı veriye dayalı yayımcı/abone sistemler için OMG Veri Dağıtım Hizmeti (DDS) (Group et al., 2004) tarafından desteklenen özel bir yayımcı/abone biçimi, sanal makinelerin kaynak izleme verilerini hem sanal olarak hem de gerçek zamanlı olarak yaymak amacıyla umut verici bir teknoloji durumundadır.

Kyoungho An ve arkadaşları, "A publish/subscribe middleware for dependable and real-time resource monitoring in the cloud" isimli çalışmalarında, bulut üzerinde çalışan gerçek zamanlı uygulamalar (SQRT-C) için ölçeklenebilir ve QoS özellikli sanal kaynak izleme sistemi adı verilen DDS tabanlı çözümlerini açıklamaktadır. SQRT-C, sanal makineler hakkında bilgi almak için bir hipervizörle etkileşime giren libvirt kitaplığından yararlanan OpenNebula, Eucalyptus ve OpenStack gibi IaaS için bulut yazılımı ile uyumludur.

OMG Veri Dağıtım Hizmeti (DDS), üç katman içeren katmanlı bir mimariden oluşmaktadır. Bu katmanlardan ikisi, DDS'yi bir bulut platformunda kaynak kullanım bilgilerinin ölçeklendirilebilir ve zamanında dağıtılmasında kullanım için umut verici bir tasarım seçeneği haline getirmiştir. Diğer katman ise veri merkezli, konu tabanlı ve gerçek zamanlı yayımcı/abone için standart bir API sağlayan Veri Merkezli Yayımcı/Abone (DCPS) sistemidir (Corsaro, 10). Verimli, ölçeklenebilir, öngörülebilir ve kaynağa duyarlı veri dağıtım özellikleri sağlamaktadır. DCPS katmanı, Gerçek Zamanlı Yayımcı/Abone (RTPS) adı verilen bir DDS ile birlikte çalışabilirlik protokolü (Schmidt and van't Hag, 2008) sağlayan başka bir katman üzerinde çalışmaktadır. DDS'nin diğer yayımcı/abone sistemler ile karşılaştırıldığında en önemli özelliklerinden birisi, DCPS katmanında sunulan QoS için sahip olduğu zengin desteğidir. DDS, ağ bant genişliği ve bellek gibi kaynakların kullanımını ve konuların kalıcılık, güvenilirlik, zamanındalık ve işlevsel olmayan diğer özelliklerini kontrol etme yeteneği sağlamaktadır (Corsaro et al., 2006).



Şekil 3.3: SQRT-C Sistem Mimarisi

SQRT-C mimarisinin yapı taşları Yayımcı, Abone, İzleme Yöneticisi ve farklı konumlarda bulunan istemcilerden oluşmaktadır. Her küme düğümü, sanal makine örneklerinin kaynak bilgilerini bir aboneye yayan bir yayımcıya sahiptir. Abone, otomatik ölçeklendirme yapan ve/veya bulutta barındırılan uygulamalar için hataya dayanıklılık sağlayan bir istemci makineye (genellikle sanal bir makine) yüklenmektedir. İzlenen sanal makinelerde hesaplama yükünü izole etmek için, yayımcı bir sanal makinede değil, fiziksel bir Küme Düğümünde barındırılır. Ön uç(frontend) düğümünde veya tek bir fiziksel düğümde (bir veritabanı bağlantısı uzaktan kurulursa) bulunan İzleme Yöneticisi, Yayımcılar ve Aboneler arasındaki DDS bağlantılarını yönetmek, istemcilerden istek almak ve Küme Düğümlerine komut göndermek için bir orkestratör görevi görmektedir. İzleme yöneticisi; sunucu ve istemci rollerinde bulunabilmektedir. İzleme yöneticisi sunucusu rolü, sanal makineler arasındaki yayımcı/abone iletişim durumunu, yayımcı/abone konu yönetimini ve OpenNebula ile veritabanı bağlantısını içeren izleme hizmetlerini denetler. İzleme yöneticisi, konuları yönetmek için üç işlem gerçekleştirmektedir (oluşturma, sonlandırma ve gösterme).

Cho ve Ntoulas, "Effective change detection using sampling"(Cho and Ntoulas, 2002) isimli çalışmalarında kaynak verilerin bağımsız olarak güncellendiğinde ve sınırlı kaynaklara sahip olunduğunda, mümkün olduğunca çok sayıda değişikliğe sahip veri öğesinin nasıl tespit edip indirebileceğine dair yaklaşımlarda bulunmaktadır. Bu senaryoda, değişmeyen öğeler tekrar tekrar indirildiğinde, kaynakların önemli bir bölümü boşa harcanabileceği için, indirmeye ve kontrol etmeye karar verilen öğenin kesinlikle önemli olduğu belirtilmektedir.

Birçok uygulama genellikle uzak veri kaynaklarının yerel kopyalarını oluşturmaktadır. Örnek olarak bir veri ambarı yerel analiz için uzaktan satış ve işlem kayıtlarını kopyalayabilir. Benzer şekilde, bir Web arama motoru Web'in bir alt kümesini kopyalar ve kullanıcıların Web sayfalarına erişmesine yardımcı olmak için dizine ekler. Birçok durumda, uzak kaynaklar yerel kopyalardan bağımsız olarak güncellenmektedir. Bu nedenle değişiklikleri tespit etmek ve kopyalara dahil etmek için kaynaklarda düzenli aralıklarla veriler yoklanmalı ve indirilmelidir.

Değişiklik algılama ve indirme genellikle kaynakların ve/veya istemcilerin gerçekleştirdiği ana görevlerle etkileşimi önlemek için çoğunlukla yoğun olmayan saatlerde toplu olarak düzenli aralıklarla gerçekleştirilmektedir. Ancak verilerin boyutu büyüdükçe değişiklikleri tespit etmek ve kopyalara dahil etmek giderek zorlaşmaktadır. Sınırlı ağ ve hesaplama kaynakları nedeniyle, veri kaynaklarındaki her veri öğesi sınırlı zaman aralığı içinde kontrol edilemeyebilmektedir. Bu nedenle kaynaklardaki bazı değişiklikler kaçırılmaktadır.

Yazarların ana fikri örneklemeyi(sampling) kullanmaktır. Yani, önce her veri kaynağından az sayıda veri öğesi örnek olarak indirilir ve örnekleri hangi kaynaklardan daha fazla veri öğesi indirildiğine dair karar vermek için kullanılır. Fikir basit olsa da, yazarların analiz ve deneylerine göre örnekleme temelli politikaların büyük bir potansiyele sahip olduğu ve önemli ölçüde iyileşmeye yol açtığı anlaşılmaktadır.

Değişiklik algılama ve indirme sorunu çeşitli bağlamlarda ortaya çıksa da, yazarların çalışmaları ağırlıklı olarak Web verilerini yönetmeye yönelik olmuştur. WebArchive projesinde [12], kullanıcıların Web sayfalarının örneğin 10 yıl önce Web'e erişebilmeleri için, zaman içinde birden çok Web sürümü saklanmaktadır. Ancak sınırlı ağ kaynaklar nedeniyle, değişiklikleri kontrol etmek için her sayfa sürekli olarak indirilememektedir. Bu nedenle hangi sayfaların indirileceğini ve kontrol edileceğinin dikkatli bir şekilde seçilmesi gerekmektedir. Benzer bir hizmet şu anda WayBack Machine tarafından sağlanmaktadır [13]. Web arama motorları da aynı sorunu ele almak zorundadır. Çünkü dizinlerini güncel tutmak için Web sayfalarını periyodik olarak tekrar ziyaret etmeleri gerekmektedir. Bu görev genellikle Web arama robotu adı verilen bir program tarafından gerçekleştirilmektedir.

(Coffman Jr et al., 1998) ve (Coffman Jr et al., 1998) çalışmalarında yazarlar, bir tarayıcının sayfa değiştirme sıklıklarını tahmin ederek nasıl daha fazla değişiklik algılayabileceğini araştırmışlardır. Diğer bir deyişle, tarayıcı, bir sayfanın geçmiş değişiklik geçmişine göre ne sıklıkta değiştiğini sürekli olarak tahmin etmekte ve bu tahmini gelecekte sayfayı ne sıklıkta yeniden ziyaret edeceğine karar vermek için kullanmaktadır. Farklı olarak yazarlar bu çalışmada örnekleme kullanarak nasıl daha fazla değişiklik tespit edebileceğini incelemektedir. Gerçekleştirdikleri deneyler ile örnekleme temelli politikaları, birçok durumda, sıklık temelli politikalara göre önemli ölçüde iyileşmeye yol açmaktadır. Gerçek Web verileri üzerinde yapılan bir denemede, örnekleme temelli politika, belirli durumlarda sıklık temelli politikanın iki katı değişiklik tespit etmiştir.

Yazarlar (Olston and Widom, 2002) itme (push) modelini araştırmaya başlamıştır. Ancak çekme modelinin World Wide Web dahil bazı mevcut uygulamalar için daha uygun olabileceği görüşüne inanmaktadırlar.

---

[12]WebArchive Project. http://webarchive.cs.ucla. edu

[13]Internet Archive. http://www.archive.org

Her indirme döngüsünde yalnızca bir veri öğesi alt kümesi indirebildiğimizde, hangi veri öğesinin indirileceğini dikkatli bir şekilde belirlememiz gerekmektedir. Bu karar için aşağıdakiler de dahil olmak üzere çok sayıda yol vardır:

1. Round-robin: Veri öğeleri her bir indirme döngüsünde round-robin tarzında indirilmektedir. Örneğin, ilk haftadaki ilk 1 milyon sayfayı, ikinci haftadaki ikinci 1 milyon sayfayı vb. indirilir. Yerel olarak 10 milyon sayfa sağlandığı için, her sayfa tam olarak her 10 haftada bir güncellenir.

2. Change-frequency-based: Bir veri öğesinin geçmiş değişiklik geçmişine dayanarak, öğenin ne sıklıkta değiştiğini tahmin eder ve öğenin ne sıklıkta yeniden ziyaret edileceğine karar verilmektedir. Örneğin, öğe bir yıl boyunca ayda bir kez indirildiyse ve 4 değişiklik tespit edildiğinde, öğenin 4 ayda bir değiştiği tahmin edilebilir ve öğe buna göre yeniden ziyaret edilebilir.

3. Sampling-based: Önce her bir veri kaynağından (örn. Bir Web sitesi) az sayıda veri öğesi örneklenmektedir ve bu kaynaktaki kaç öğenin değiştiği tahmin edilmektedir. Daha sonra, tahminlere göre indirme kaynakları, her veri kaynağına göre tahsis edilmektedir. Örneğin,10.000 Web sitesinin her birinden 10 sayfa örnek olarak indirilebilir (toplam 100.000 sayfa örneği) ve örneklerde kaç sayfa değiştiği sayılabilmektedir. (Şimdilik, sayfanın değişip değişmediğini görmek için bir sayfanın gerçekten indirilmesi gerektiği varsayılmaktadır.) Ardından sayılara dayanarak, kalan 900.000 indirme kaynağı her bir Web sitesine uygun şekilde tahsis edilmektedir.

Freivald ve Noble, "Unique-change detection of dynamic web pages using history tables of signatures"(Freivald and Noble, 2000) isimli çalışmalarında sıklıkla değişime uğrayan Web sayfalarındaki değişimi tespit eden ve kullanıcının mail adresine bildirim olarak gönderen bir araç tasarlamışlardır. Web sayfalarında bulunan bilgiler kolayca güncellenebilmekte veya değiştirilebilmektedir. Ancak, kullanıcılar değişikliklerin farkında olmayabilir. Kullanıcılar Web sayfalarındaki bilgileri sık sık ziyaret edip yeniden okumadığı sürece, bilginin değiştiğini fark etmeyebilirler.

Web'deki belgeler Web sayfaları olarak bilinir ve bu Web sayfaları sık sık değişmektedir. Kullanıcılar genellikle belirli Web sayfalarında ne zaman değişiklik yapıldığını bilmek isterler. Yazarlar bu çalışmada kullanıcıların Web sayfalarını kaydetmelerine izin veren bir değişiklik algılama aracından bahsetmektedirler. Kayıtlı her Web sayfası periyodik olarak indirilmekte ve bir değişiklik olup olmadığını belirlemek için kayıtlı sayfanın saklı bir sağlama toplamı(checksum)

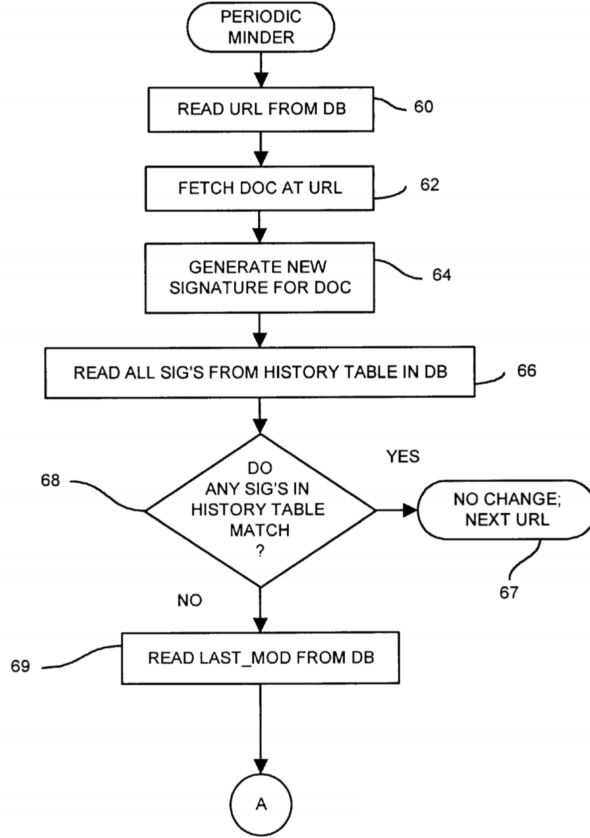
veya imzası ile karşılaştırılmaktadır. Bir değişiklik algılandığında, kullanıcı e-posta ile bilgilendirilmektedir. Üst uygulamanın değişiklik algılama aracı, kullanıcının değişiklik algılaması için bir Web sayfası belgesinin bölümlerini seçmesine olanak tanımaktadır. Bu sayede herhangi bir Web sayfasındaki diğer bölümler yoksayılıp sadece istenilen bölümlerdeki değişikliklerden haberdar olunması sağlanmıştır.

Kullanıcı, değişiklik algılama aracı için bir kullanıcı arabirimi aracılığıyla sayfayı kaydetmektedir. Kullanıcı e-posta adresini ve Web sayfasının URL'sini girer. Değişiklik algılama aracı bu sayfanın bir kopyasını alır ve bir imza oluşturur. Değişiklik algılama aracı, bir değişiklik olup olmadığını görmek için bu Web sayfasını düzenli olarak indirmektedir. Yeniden indirilen sayfa için yeni bir imza oluşturulur ve yeni imza veritabanında saklanan eski imza ile karşılaştırılır. Uyumsuzluk durumu bir değişikliğin algılandığını göstermektedir. Değişiklik algılama aracı, kayıtlı her sayfayı birkaç saatte veya günde yeniden indirmektedir. Web sayfası güncellendiğinde, değişikliğin tespit edildiğine dair ona özel farklı bir imza oluşturulmaktadır. Yeni imza veritabanında saklanır ve kullanıcı e-posta ile bilgilendirilir.

Bazı değişiklik algılama yazılımları, yalnızca bir Web sunucusundan alınan HTTP yanıtındaki 'Son-Değiştirilme' üstbilgisine(Last-Modified header) dayanır. Örneğin, Microsoft Internet Explorer 4.0, "Sık Kullanılanlar" menüsü altında Web sayfalarındaki değişiklikleri algılayan "Abonelikler" adlı bir özelliğe sahiptir. Bu özellik, bir Web sayfasının ne zaman değiştiğini belirlemek için 'Son-Değiştirilme' üstbilgisine dayanır. Ne yazık ki, birçok Web sayfası 'Son-Değiştirilme' üstbilgisini döndürmez ve Internet Explorer 'Son-Değiştirilme' üstbilgisi içermeyen bir Web sayfasını her denetlediğinde yanlış değişiklik bildirimleri oluşturur.

Tüm belgeler 'Son-Değiştirilme' üst bilgisini içermez. Bu nedenle 'Son-Değiştirilme' üstbilgisi, dinamik içerikteki değişiklikleri yansıtmayabilir. Bazı Web sunucuları 'Son-Değiştirilme' üstbilgisini yalnızca statik içerik değiştiğinde güncelleştirir. Böylece dinamik içerik değiştiğinde, değişiklik bildirimleri üretilmez. Dinamik içerik kullanıcının değişikliklerini kontrol etmek istediği şey olduğunda bu istenmeyen bir durum olabilir. Örneğin, kullanıcı belirli bir ürün veya şirket adının görünümü için haber gruplarında aramak istediğinde, aramanın sonucu dinamik bir içerik olmaktadır. Kullanıcı basit bir değişiklik algılama aracı kullandığında, Web sunucusu 'Son-Değiştirilme' üstbilgisini döndürmezse, arama sonucu her kontrol edildiğinde kullanıcı her seferinde bilgilendirilir. Web sunucusu yalnızca statik içeriğe dayalı olarak bir 'Son-Değiştirilme' üstbilgisi döndürürse, arama sonuçlarındaki dinamik içerik değiştiğinde kullanıcı bilgilendirilmemiş olmaktadır.

’Son-Değiştirilme’ üstbilgisi, HTML üstbilgisi değiştiğinde de görülebilir, ancak görünür belge değişmemiştir. Bu gibi bir durumda yanlış değişikliklerin bildirilmesine neden olabilir. Değişiklik algılama aracı, içeriği yalnızca ’Son-Değiştirilme’ üstbilgisine güvenmek yerine, değişiklikler için analiz edecek kadar akıllı olsa bile, sunucu bazı hatalar nedeniyle Web sayfasının yalnızca bir bölümünü döndürdüğü durumlarda da yanlış değişiklikler bildirebilmektedir.



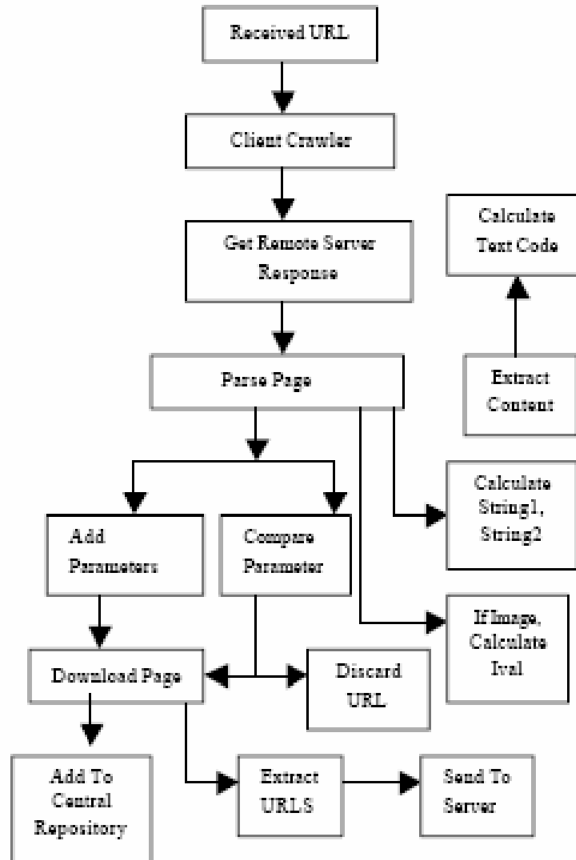
Şekil 3.4: Tarihçe Tabloları Kullanarak Gereksiz İlgili Olmayan Değişiklikleri Önleme Akışı Diyagramı

Yadav ve arkadaşları, "Parallel crawler architecture and web page change detection"(Yadav et al., 2008) isimli çalışmalarında, istemci sunucu mimarisi çizgileri üzerine inşa edilmiş bir mimariyi ortaya koymaktadır. Birden fazla makine kullanarak Web’i paralel taramak için yeni bir yaklaşımı ortaya çıkarmakta ve tarama ile ilgili önemsiz birçok konuyu da bütünleştirmektedir. Web’i taramanın en önemli kullanım amaçları; endekslemek ve Web sayfalarını güncel tutmaktır. Daha sonra bu veriler arama motoru tarafından son kullanıcı sorgularını sunmak için kullanılmaktadır. Web’in önemli bir kısmı dinamiktir ve bu nedenle değiştirilen Web sayfalarının sürekli güncellenmesi ihtiyacı ortaya çıkmaktadır. Yazarlar bu çalışmada sayfa yenileme işlemi için üç adımlı bir algoritma kullanmıştır ve bu

sayede bir Web sayfasının yapısının, metin içeriğinin veya görüntüsünün değiştirilip değiştirilmediğini kontrol etmişlerdir.

Bu çalışmada, paralel bir arama robotu oluşturmak için yeni bir tasarım mimarisi önerilmektedir. Tasarımı yaparken karşılaşılan ana zorluk, arama robotunun performansını en üst düzeye çıkarmak ve paralelleşme nedeniyle ortaya çıkan bellek, bant genişliği vb. giderlerini minimumda tutmayı sağlamaktır. İşlemlerin en verimli şekilde çalışmasını sağlamak için her birinin düzgün çalışması gerekmektedir. Bundan dolayı yazarlar çalışmalarında tarama işlemlerinin karşılaştığı bazı genel sorunları ele almışlardır.

Paralel tarayıcıdaki tüm işlemler birbirinden bağımsızsa, çekirdek URL'lerinden başlamaktadırlar. URL'leri onlardan ayrıştırırlar ve sıralarına eklerler. İki farklı sayfa, aynı sayfayı gösterebileceğinden dolayı farklı işlemlerin aynı sayfaları indirmesi oldukça olasıdır (Cho and Garcia-Molina, 2002). Bu durum, bellek ve bant genişliği kaybına yol açar. Veritabanı aynı Web sayfasının birden fazla kopyasını bulundurmamalıdır. Bu kopyalar, farklı zaman dilimlerinde indirilmiş olabileceğinden dolayı farklı olabilir. Bu gibi durumlarda arama motoruna ait Web havuzunun kalitesi düşmektedir.



Şekil 3.5: Paralel Arama Robotu Mimarisi

Web'in dinamik doğasıyla mücadele etmek için, mevcut bir sayfanın değiştirilip değiştirilmemesine bağlı olarak ne zaman yeniden indirileceğine karar veren uygun bir yaklaşım önerilmesi gerekmektedir (Liu et al., 2000; Rocco et al., 2003; Buttler et al., 2004; Francisco-Revilla et al., 2001; Khan et al., 2002; Zhang et al., 2004; Yadav et al., 2007b; Yadav et al., 2007a).

3.1 Literatürdeki Benzer Çalışmalarla Karşılaştırılması

Değişiklik izleme prototipine ait literatür taramaları ışığında hazırlanmış değişim iletim kanalı, mimari, kullanım alanı ve değişim tespiti yapıları açısından ele alınan karşılaştırmalara Tablo:3.1 'de yer verilmiştir.

Ramasubramanian ve arkadaşları, "Corona: A High Performance Publish-Subscribe System for the World Wide Web." isimli çalışmalarında; kullanıcıların değişimleri alması amacıyla anlık mesajlaşma uygulamaları tercih edilmiştir. Benzer şekilde Freivald ve Noble, "Unique-change detection of dynamic web pages using history tables of signatures" isimli çalışmalarında, tarihçe tabloları kullanılarak Web sayfalarındaki değişimin tespit edilmesi ve tespit edilen değişikliklerin talep eden kullanıcılara e-posta yoluyla bildirilmesi sağlanmaktadır. Fakat günümüz dünyasında artık yazılımların da Web üzerindeki değişiklerden haberdar olmak istemesi ile uygulamalara entegrasyon aşamasında bu tezde geliştirilen prototipin önemi ortaya çıkmaktadır.

Ravikant Cherukuri, "Realtime websites with publication and subscription" isimli çalışmasında geliştirmiş olduğu araç ile, web sayfası nesnelerine ek özellikler ekleyerek javascript kodu aracılığıyla güncellemeleri iletmeyi amaçlamıştır. Fakat bu dönüşüm için ek bir maliyet gerekmektedir. Bu sebepten dolayı böyle bir yaklaşım bu tezde tercih edilmemiştir. Ayrıca uzun yoklama metodunu kullanması da yüksek bant genişliği kullanmasına ve sunucuya ek yük getirmesine neden olmaktadır. Bu tezde güncellemelerin ilgili kullanıcılara iletilmesinde tercih edilen yaklaşım ise yayımcı/abone mimarisi kullanımı ile gereksiz yoklamalardan kaçınarak sunucuya düşen yük miktarını en aza indirmektir.

Cho ve Ntoulas, "Effective change detection using sampling" isimli çalışmalarında yazarlar tespit edilecek olan değişikliklerin daha etkin bir şekilde tespit edilmesi üzerine yoğunlaşmışlardır. Yadav ve arkadaşları, "Parallel crawler architecture and web page change detection" isimli çalışmalarında, paralel bir arama robotu oluşturmak için yeni bir tasarım mimarisi önerilmektedir. Ayrıca yazarlar sayfa yenileme işlemi için üç adımlı bir algoritma kullanmıştır ve

bu sayede bir Web sayfasının yapısının, metin içeriğinin veya görüntüsünün değiştirilip değiştirilmediğini kontrol etmişlerdir. Benzer şekilde Boldi ve arkadaşları, "Ubicrawler: A scalable fully distributed web crawler" isimli çalışmalarında Ubicrawler isimli bir arama robotu geliştirilmiştir. Yapılan çalışma hataya dayanıklılık ve ölçeklenebilirlik açısından güzel sonuçlar verse de Web üzerindeki değişikliklerin tespiti ve istemcilere bildirimi açısından beklentileri karşılayamamaktadır. Bu tezde yapılan çalışmada ise değişikliklerin tespiti ve abonelere bildirimi en önemli odak noktalarından birisidir.

Kyoungho An ve arkadaşları, "A publish/subscribe middleware for dependable and real-time resource monitoring in the cloud" isimli çalışmalarında tasarlanan mimari ile gerçek zamanlı, ölçeklenebilir ve yayımcı/abone mimarisine dayanan bir izleme altyapısı oluşturmuşlardır. Fakat oluşturdukları sistemin asıl amacı sunuculara ait kaynakların izlenmesini sağlamaktadır. Bu nedenle kapsam açısından bu tezin amacına uygun değildir.

	Değişim İletim Kanalı	**Mimari**	**Kullanım Alanı**	**Değişim Tespiti**
Corona: A High Performance Publish-Subscribe System for the World Wide Web	Anlık Mesajlaşma Uygulamaları	Yüksek Performans, Yayımcı/Abone	Web Abonelikleri	Çıkartım, Filtreleme, Delta Kodu
Realtime Websites With Publication And Subscription	Javascript	Uzun Yoklama	Gerçek Zamanlı Web	Manuel
UbiCrawler: a scalable fully distributed Web crawler	-	Ölçeklenebilir, Dağıtık	Crawler	HTML Çıkartımı
A Publish/Subscribe Middleware for Dependable and Real-time Resource Monitoring in the Cloud	Yayımcı/Abone Konuları	Gerçek zamanlı, Ölçeklenebilir	Gerçek Zamanlı Kaynak İzleme	İzleme Yöneticisi
Parallel Crawler Architecture and Web Page Change Detection	-	Paralel	Crawler	3 Adımlı Algoritma

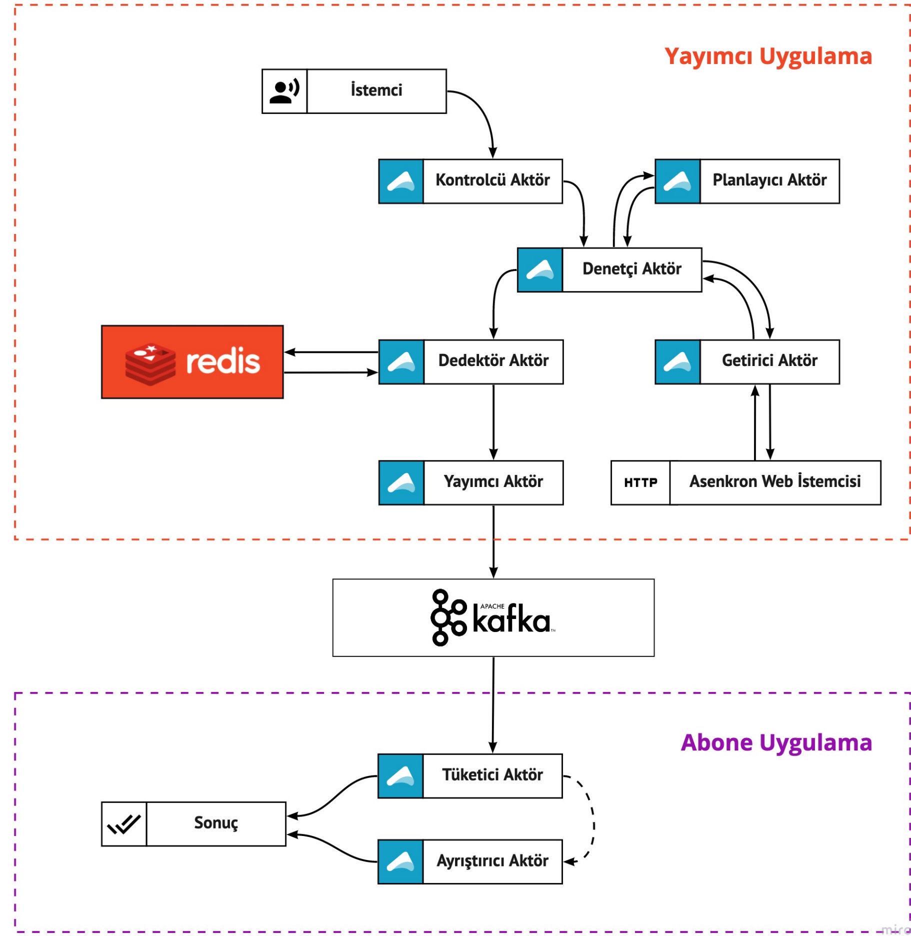
Çizelge 3.1: Literatür Karşılaştırması

4 DEĞİŞİKLİK İZLEME ALTYAPISININ GELİŞTİRİMİ

Tezin bu bölümünde geliştirilmiş olan prototip değişiklik izleme mimari yapısı ve bu mimari yapı içerisinde bulunan birimler anlatılacaktır. Tezin amacı olan ölçeklenebilirlik özelliğini sağlayabilmek için endrüstride kullanılan ve ölçeklenebilir sistemler oluşturma konusunda kendisini yüksek miktarda kullanım sağlanarak ispatlamış, açık kaynak koduna sahip Akka, Apache Kafka kullanılmıştır. Ayrıca durum saklamak ve sorgulamak için bellek kullanımlı, açık kaynak kodlu, anahtar-değer deposu olan Redis tercih edilmiştir. Elde edilen sonuçların eğer istenirse bir abone tarafından istenilen alt birimlerine erişim sağlayarak çözümlenerek kullanılabilmesi sağlanmıştır. Bundan sonraki bölümlerde sistemin mimari yapısı ve aktör hiyerarşisi ve alt bileşenlerin kullanımı anlatılacaktır.

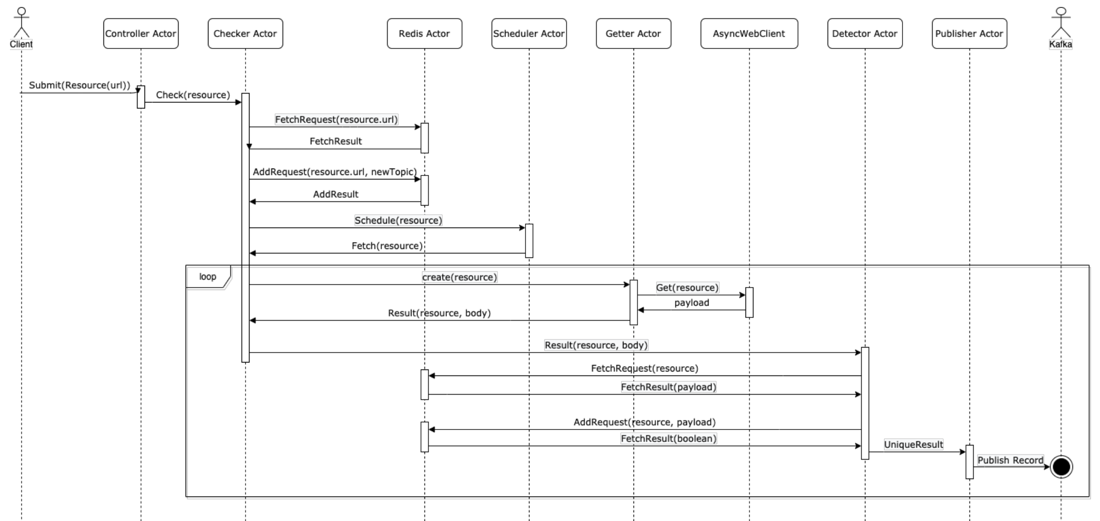
4.1 Değişiklik İzleme Altyapısının Mimari Yapısı

Geliştirilmiş olan mimari, yoğunlukla aktör hiyerarşisine dayanarak yayımcı/abone sistemi ile entegre olmaktadır. İstemci tarafında bulunan kullanıcı ya da uygulama, izlemek istedikleri endpointlere ait detayları aktörlere ileterek sistem üzerine izleme işlemini başlatmış olmaktadır. Sistemde bulunan her bir aktör kendine özel bir işlevi yerine getirmek üzere tasarlanmıştır. Bu şekilde tasarlanan mimari yapısı sayesinde kaygıların ayrılması(separation of concerns ~ SoC) tasarım prensibine daha uygun bir yapı elde edilmiş olunmaktadır. Ayrıca yazılım bileşenleri arasındaki bağlılık (coupling) ve bileşenler içerisindeki birliktelik (cohesion) gibi iki kavram da SoC için önem arz etmektedir. Düşük bağlılık (low-coupling) ve yüksek sorumluluk (high-cohesion) daha kontrollü bir sistem inşaası için tercih edilmektedir. Aynı zamanda yazılımların esnek ve genişletilebilir olmasını da sağlamaktadır.



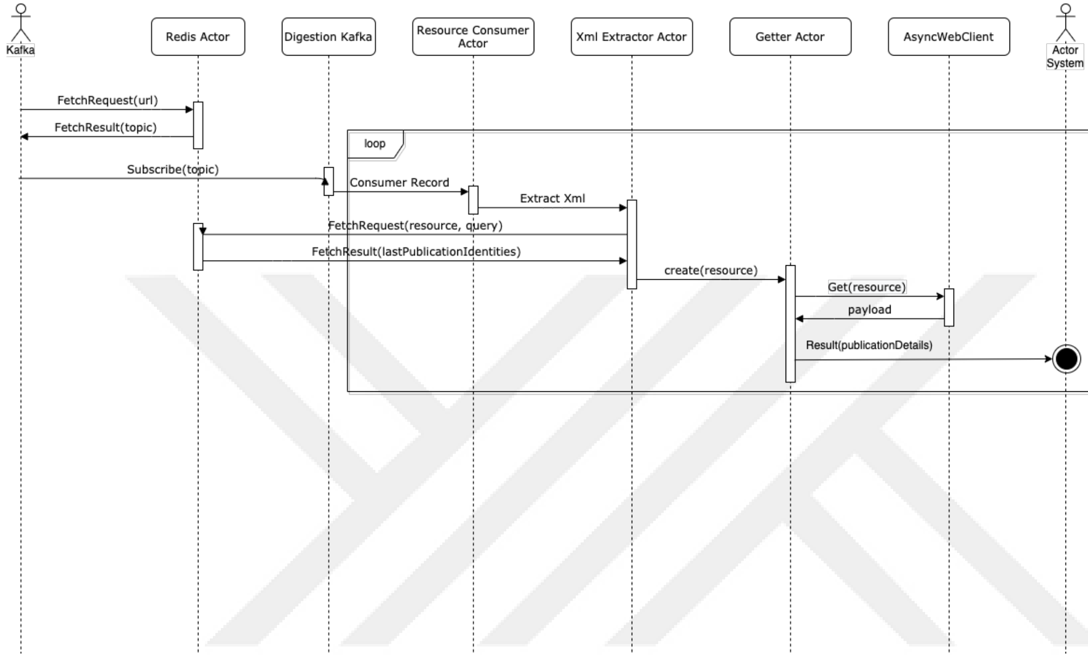
Şekil 4.1: Prototip Değişiklik İzleme Altyapı Mimarisi

Mimarinin gösterimi Şekil 4.1'de yapılmıştır.



Şekil 4.2: Yayımcı Uygulama Diyagramı

Mimarinin yayımcı uygulama tarafına ait nesne etkileşimlerini ve nesneler arasında değiştirilen mesajların sırasının diyagram ile gösterimi Şekil 4.2'de yapılmıştır.



Şekil 4.3: Abone Uygulama Diyagramı

Mimarinin abone uygulama tarafına ait nesne etkileşimlerini ve nesneler arasında değiştirilen mesajların sırasının diyagram ile gösterimi Şekil 4.3'de yapılmıştır.

4.1.1 Kontrolcü Aktör

Kontrolcü(Controller) aktör, sistemde hiyerarşik düzene tasarımsal olarak en üstte yer almaktadır. İstemciden (client) aldığı mesajları denetçi(checker) aktöre yollayarak ilgili mesajların iletiminde görev almaktadır. Aynı zamanda denetçi aktör'ün yaşam döngüsünü takip ederek, göndermiş olduğu mesaj isteğine karşılık bir cevap alamaması durumunda sahip olduğu tüm çocuk aktörlere dur emri iletir. Eğer tüm çocuk aktörlerin durduğundan emin olursa kendisini durdurmaktadır. Kontrolcü aktör ile ilgili işlevselliğe 1 numaralı algoritmada yer verilmektedir.

Algoritma 1 Kontrolcü Aktör Mesajlaşma Algoritması

```
def receive: Receive = LoggingReceive {
  case Messages.Submit(resource) =>
    log.info("Checking resource: {}", resource.toString)
    val checker = context.actorSelection("/user/app/checker")
    checker ! Messages.Check(resource)

  case Terminated(actor) =>
    log.error("Actor:{} terminated.", actor.path.toString)
    if (context.children.isEmpty) {
      log.debug("Controller context has no children. Controller stopping itself.")
      context.stop(self)
    }

  case ReceiveTimeout =>
    log.error("Got ReceiveTimeout, all of the child actors will be stopped.")
    context.children foreach context.stop

  case err: JobFailed =>
    log.error(s"Job failed. Details: $err")

  case Messages.UniqueResult(job, payload) =>
    if (payload.isEmpty) {
      log.error("No results found for job:'{}'", job)
    } else {
      log.info("Results for job:'{}':{}", job, payload)
    }
}
```

4.1.2 Denetçi Aktör

Denetçi(Checker) aktöre gelen mesaj ile aktörün ilgili kaynağı kontrol etmesi söylenmektedir. Gelen mesajın içerdiği veride bulunan kaynak detayları kullanılarak bellek tabanlı Redis anahtar-değer veritabanı sorgulanarak daha önce aynı kaynağın izlenme işleminin başlatılma durumu öğrenilmiş olunmaktadır. Aynı kaynak veritabanında bulunuyorsa tekrardan izlenmesi için zamanlama planlaması yapılmamaktadır. Eğer kaynak veritabanında bulunamadıysa Planlayıcı(Scheduler) aktör'e zamanlamasını başlatmasını söyleyen bir mesaj gönderilir.

Planlayıcı aktör tarafından zamanlanan kaynak izleme taleplerine ait mesajlar Denetçi aktör'e ulaştığında, bir alt aktör olarak Getirici(Getter) aktör oluşturulur. Getirici aktörden gelen anlık kaynak verisi, değişimin tespit edilmesi için Dedektör(Detector) aktör'e iletilir. Denetçi aktör ile ilgili işlevselliğe 2 numaralı algoritmada yer verilmektedir.

Algoritma 2 Denetçi Aktör Mesajlaşma Algoritması

```
override def receive: Receive = LoggingReceive {
  case msg@Messages.Check(resource) =>
    log.info(s"Got check message:$msg")
    redisClient.get(resource.url) match {
      case Some(topic) =>
        log.info(s"Got topic from redis:$topic")
        sender() ! Messages.JobFailed( job = Job(self, resource), reason =
s"Resource scheduling already started for topic: $topic" )

      case None =>
        val newTopic = UUID.randomUUID().toString
        log.info(s"New topic:$newTopic generated for url:${resource.url}")
        redisClient.set(resource.url, newTopic)

        val scheduler = context.actorSelection("/user/app/scheduler")
        scheduler ! Messages.Schedule(resource)
    }

  case Messages.Fetch(resource) =>
    val getter = context.actorOf(Props(new Getter(resource)), "getter")
    context.watch(getter)

  case result: Messages.Result =>
    log.info("New result received and sending to the newly created detector
actor.")
    context.stop(context.unwatch(sender))

    val detector = context.actorOf(Props[Detector], s"detector-${Random.nextInt()}")
    context.watch(detector)
    detector ! result

  case uniqueResult: Messages.UniqueResult =>
    log.info(s"Got unique result from ${uniqueResult.resource}")
    context.parent ! uniqueResult

  case Terminated(actor) =>
    log.warning("Actor:{} terminated.", actor.path.toString)

  case ReceiveTimeout =>
    log.error("Got receive timeout! All of the child actors will be stopped.")
    context.children foreach context.stop
}
```

4.1.3 Planlayıcı Aktör

Planlayıcı(Scheduler) aktör, gelen mesaj ile ilgili kaynak için zamanlayıcının başlatılmasından sorumludur. Akka araç kitinin altyapısında bulunan planlayıcı yapısı uzun süreli zamanlama işlemleri için tasarlanmamıştır. Gelecekteki olayların tetiklenmesi için planlayıcının Akka Scheduler[14] kullanılarak maximum 8 ay boyunca çökmeyeceği düşünülebilmektedir. Orjinal sitesinde de yer verildiği gibi

[14]Classic Scheduler, https://doc.akka.io/docs/akka/current/scheduler.html

eğer uzun süreli zamanlama işlevine ihtiyaç duyuluyorsa akka-quartz-scheduler[15] yapısının kullanılması tavsiye edilmektedir.

Planlayıcı aktör ile Akka Quartz Scheduler kullanılarak, mesajlar ile gelen veri içerisindeki kaynak için planlanan bir zaman(Örnek: 20 saniye) aralığında 'Fetch' mesajı gönderimi yapılmaktadır. Planlayıcı aktör ile ilgili işlevselliğe 3 numaralı algoritmada yer verilmektedir.

Algoritma 3 Planlayıcı Aktör Mesajlaşma Algoritması

```
val quartzScheduler = QuartzSchedulerExtension(context.system)

override def receive: Receive = LoggingReceive {
  case m: Messages.Schedule =>
    try {
      val jobName = m.resource.url
      val schedule = quartzScheduler.createJobSchedule(
        name = jobName,
        receiver = sender(),
        msg = Messages.Fetch(m.resource),
        cronExpression = "*/20 * * ? * *" // Will fire every 20 seconds
      )
      log.info("Job: {} scheduled at: {}.", jobName, schedule)
      log.info("Running schedule jobs: {}", quartzScheduler.runningJobs)
    } catch {
      case exception: IllegalArgumentException => exception.printStackTrace()
    }
}
```

4.1.4 Getirici Aktör

Getirici(Getter) aktör ile ilgili kaynağa ait veri bir istemci yardımıyla çekilmektedir. Bunun için com.ning.http.client.AsyncHttpClient kullanılarak asenkron ve bloklamadan ilgili kaynak Web ortamından alınır.

Getirici aktör, diğer aktörlerden farklı olarak bir mesaj almamaktadır. Oluşturulma sırasında verilen argüman, elde edilmek istenen kaynağa ait detaylar içermektedir. Kaynağa ait detayları kullanarak ilklendirilen aktör, oluştuktan hemen sonra AsyncWebClient yardımıyla veriyi çekip işlem tamamlandıktan sonra ilgili veriyi kendisine mesaj olarak yollamaktadır. Daha sonra ise Getirici aktör aldığı bu kaynak verisini ebeveyn aktörüne ileterek kendisini durdurmaktadır. Getirici aktör ile ilgili işlevselliğe 4 numaralı algoritmada yer verilmektedir.

[15] Akka Quartz Scheduler, https://github.com/enragedginger/akka-quartz-scheduler

Algoritma 4 Getirici Aktör Mesajlaşma Algoritması

```
package sample.sharding.actors

import java.util.concurrent.Executor
import akka.actor.{Actor, ActorLogging, Status}
import akka.event.LoggingReceive
import akka.pattern.pipe
import sample.sharding.client.{AsyncWebClient, WebClient}
import sample.sharding.messages.Messages
import sample.sharding.models.Resource

import scala.concurrent.ExecutionContext

class Getter(resource:  Resource) extends Actor with ActorLogging {
  implicit val executor:  Executor with ExecutionContext =
context.dispatcher.asInstanceOf[Executor with ExecutionContext]

  private def webClient:  WebClient = AsyncWebClient

  webClient get resource.url pipeTo self

  def receive:  Receive = LoggingReceive {
    case body:  String =>
      log.debug("Body fetched successfully.")
      log.debug("Result sending to the parent actor.")
      context.parent !  Messages.Result(resource, body)
      context.stop(self)

    case _:  Status.Failure => context.stop(self)
  }
}
```

4.1.5 Dedektör Aktör

Dedektör(Detector) aktör bir kaynakta zaman içerisinde değişimin tespiti için kullanılmaktadır. Sahip olduğu bellekte tutulan bir anahtar-değer veritabanı olan Redis bağlantısı sayesinde sistemdeki mevcut kaynaklar içerisinde, yeni gelen mesaj içerisindeki veriyi sorgulayarak değişimi belirler. Eğer değişim tespit edilirse yeni veriyi Redis veritabanına kaydeder. Daha sonra ise bu kaynağı, abone olan istemcilere iletmek amacıyla Apache Kafka sistemine yollaması için Yayımcı(Publisher) aktör'e iletir. Dedektör aktör ile ilgili işlevselliğe 5 numaralı algoritmada yer verilmektedir.

Algoritma 5 Dedektör Aktör Mesajlaşma Algoritması

```
override def receive: Receive = LoggingReceive { case result: Messages.Result =>
  redisClient.get(result.resource.asJson.noSpaces) match {
    case Some(payload) =>
      if (payload.equals(result.payload)) {
        log.info("##" * 50)
        log.info("Result is the same as before.")
        log.info("##" * 50)
      } else {
        handleUniqueResult(result)
      }

    case None =>
      handleUniqueResult(result)
  }
}

private def handleUniqueResult(result: Messages.Result): Unit = {
  log.info("##" * 50)
  log.info("Found unique result!")
  log.info("##" * 50)

  redisClient.set(result.resource.asJson.noSpaces, result.payload)

  val uniqueResult = UniqueResult.fromResult(result)
  val publisher = context.actorSelection("/user/app/publisher")
  publisher ! uniqueResult
}
```

4.1.6 Yayımcı Aktör

Yayımcı(Publisher) aktör, dedektör aktör tarafından tespit edilen benzersiz sonuçları alarak ilgili abonelere dağıtılması amacıyla Apache Kafka sistemine iletmekle görevlidir.

Apache Kafka sistemine iletilen kayıtların içeriği aşağıdaki gibidir.

Topic	Key	Value
resource.url	resource.query	resource.payload

Çizelge 4.1: Kafka Konu(Topic) Yapısı

Yayımcı aktör ile ilgili işlevselliğe 6 numaralı algoritmada yer verilmektedir.

Algoritma 6 Yayımcı Aktör Mesajlaşma Algoritması

```
val producer = new KafkaProducer[String, String](Publisher.producerProperties)

override def receive:  Receive = LoggingReceive {
  case UniqueResult(resource:  Resource, payload:  String) =>
    val url:  String = resource.url
    val key:  String = resource.mayBeQuery.getOrElse("root")
    val value:  String = payload

    val topic:  String = redisClient.get(url) match {
      case Some(topic) => topic
      case None =>
        val newTopic = UUID.randomUUID().toString
        log.info(s"New topic:$newTopic generated for url:$url")
        redisClient.set(url, newTopic)
        newTopic
    }

    log.info("New record publishing...\nTopic:{}, Key:{}, Value:{}", topic, key,
value)
    val record = new ProducerRecord[String, String](topic, key, value)
    try {
      producer.send(record).get
    } catch {
      case e:  Throwable =>
        log.error(e, "An error occurred while sending record to Apache Kafka.")
    }
}
```

4.1.7 Tüketici Aktör

Tüketici(Consumer) aktör, değişiklik izleme prototipine abone olup değişikleri talep eden istemciyi ifade etmektedir. Apache Kafka sisteminde bulunan bir konu dizisine (tasarlanan prototipte konu dizisi olarak kaynağa ait url) abone olunduktan sonra, değişiklik olarak sistemin ilettiği kayıtlar elde edilmektedir. Eğer ilgili kaynak tipi ayrıştırılabilecek tipler arasındaysa ve sorgu (query) belirtildiyse, detaylı sonuçlar çıkartmak amacıyla Ayrıştırıcı(Extractor) Aktör'e yollanabilmektedir.

İlgili konu dizisine abonelik işlemine 7 numaralı algoritmada yer verilmektedir.

Algoritma 7 Tüketici Akışı Algoritması

```
private val redisClient = new RedisClient("localhost", 6379)
redisClient.get(url) match {
  case Some(topic) =>
    logger.info(s"Kafka consumer starting for topic:  $topic")
    val runningStream = Consumer.plainSource(
      settings = consumerSettings.withClientId(s"client-${UUID.randomUUID()}"),
      subscription = Subscriptions.topics(topic)
    )
    .to(consumerSink)
    .run()

    sys.addShutdownHook {
      logger.info("System shutting down...")
      Await.ready(runningStream.shutdown, 10.seconds)
      Await.ready({ materializer.shutdown system.terminate }, 10.seconds)
    }

  case None =>
    throw new IllegalStateException("Topic id not found!")
}
```

Abonelik işlemi tamamlandıktan sonra akıştan gelen her bir kayıt Tüketici aktöre ulaşacaktır. Tüketici aktör ise aldığı mesajlara karşılık olarak bir kabul mesajı ile cevaplamaktadır. Bunun amacı ise akışı bilgilendirerek elindeki işi tamamladığını ve daha fazla eleman işleyebileceğini bildirmektir.

Tüketici aktör ile ilgili işlevselliğe 8 numaralı algoritmada yer verilmektedir.

Algoritma 8 Tüketici Aktör Mesajlaşma Algoritması

```
override def receive:  Receive = {
  case StreamInitialized =>
    log.info("Stream initialized!")
    sender() !  Ack // ack to allow the stream to proceed sending more elements

  case record:  ConsumerRecord[String, String] =>
    log.info(s"Consumed a new stream message for url:${resource.url}")
    log.info(s"Message:  $record")
    if (resource.mayBeQuery.isDefined) {
      val extractor = context.actorOf(Props[Extractor])
      extractor !  ExtractHTMLResult(resource, record.value())
    }
    sender() !  Ack // ack to allow the stream to proceed sending more elements

  case StreamCompleted => log.info("Stream completed!")

  case StreamFailure(ex) => log.error(ex, "Stream failed!")

  case default =>
    log.warning(s"Unknown message received.")
    log.info(s"Message:  $default")
}
```

4.1.8 Ayrıştırıcı Actor

Geliştirilen prototipin abone tarafında kullanılması amacıyla tasarlanmıştır. Elde edilen XML verisi üzerinde istenilen yol(path) belirtilerek içerdiği veriyi scala-xml modülü ile elde etmektedir.

Ayrıştırıcı aktör ile ilgili işlevselliğe 9 numaralı algoritmada yer verilmektedir.

Algoritma 9 Ayrıştırıcı Aktör Mesajlaşma Algoritması

```
case result: ExtractXMLResult =>
val xml: Elem = XML.loadString(result.payload)
val query = result.resource.mayBeQuery.get
val idElems: NodeSeq = query.foldLeft(xml)(op = (e, s) => e \ s)
val currentIds: Set[Int] = idElems.map(_.text.toInt).toSet

val key = result.resource.asJson.noSpaces
(redisActor ? FetchElementRequest(key)).onComplete {
  case Success(value) => value match {
    case FetchElementResult(res) => res match {
      case Some(idsJsonString) =>
        decode[Set[Int]](idsJsonString) match {
          case Right(idsSet) =>
            val differences: Set[Int] = currentIds.diff(idsSet)
            if (differences.nonEmpty) {
              differences.foreach(getPublicationAbstract)
            } else {
              log.info("New publication not found!")
            }

          case Left(error) => log.error(error.getMessage)
        }

      case None =>
        for { id <- currentIds } yield {
          val url = s"https://www.ncbi.nlm.nih.gov/pubmed/$id?report=abstract&format=text"
          context.actorOf(Props(new Getter(Resource(url, None))))
        }
    }
  }

  case Failure(exception) => throw exception
}

case result: Messages.Result => log.info(s"New publication abstract extracted.
Result:${result.toString}")
```

4.2 Değişiklik İzleme Mimarisinin Gerçekleştirimi

Bölüm 4'te de anlatıldığı üzere değişiklik izleme mimarisi özellikle ölçeklenebilirlikle ilgilenmektedir. Prototip değişiklik izleme mimarisi de bu

özellikleri sağlayabilecek yapıda olmalıdır. Mimari yapı, yayımcı ve abone olarak iki ana bileşenden oluşmaktadır. Her bileşen ayrı olarak dağıtık çalışmaya uygun bir altyapı göz önüne alınarak tasarlanmıştır. Gerçekleştirilen değişiklik izleme sistemine ait mimari Şekil:4.1 'de görülebilmektedir. Böylece kullanım yoğunluğunun artması durumunda kolaylıkla ölçeklenebilecek bir yapıya sahip olmaktadır.

İki tarafın da birbirine olan bağlantısı ölçeklenebilir bir şekilde olması gerektiğinden dolayı ek bir mimari bileşen olarak yayımcı/abone sistemi kullanılmıştır. Yayımcı/abone sisteminin ölçeklenebilirliğe katkısından 2.1 bölümünde bahsedilmiştir. Yayımcı tarafındaki uygulamalar, Apache Kafka sistemine, tespit edilen değişiklikleri iletmektedir. Abone tarafındaki uygulamalar ise Apache Kafka sisteminden talep ettikleri değişiklikleri alabilmektedirler.

4.2.1 Yayımcı Uygulama

İstemci şeklinde davranan bir uygulama aracılığıyla sisteme 'Submit.Resource' isteği gelir ve bu istek sistemdeki aktör hiyerarşisinde en üst seviyede bulunan 'Kontrolcü' aktöre iletilir. Kontrolcü aktör aldığı mesajı kontrol etmesi için 'Denetçi' aktöre gönderir. Denetçi tarafından alınan mesaj ilk olarak bellekte tutulan bir veritabanı olan Redis'e iletilir ve sistemde daha önce ilgili kaynak için izleme talebi gelmiş mi diye sorgulanmaktadır. Eğer sorgulama sonucunda sistemde bir kayıt bulunursa, hatayı açıklayan bir mesaj yardımıyla Kontrolcü aktöre iletilip hata bir üst seviyedeki aktörde işlenir. Eğer sistemde bir kayıt bulunmadıysa, ilgili uç birim için konu dizisi oluşturularak Redis veritabanında saklanmaktadır. Konu dizisi oluşturumu açısından Apache Kafka'nın karakter sınırlamaları bulunmaktadır. Bundan dolayı uç birime ait adres, konu dizisi olarak eklenememektedir. Bunun yerine 128 bitlik değere sahip değişmez evrensel benzersiz tanımlayıcı olarak(immutable universally unique identifier) adlandırılan UUID oluşturulmaktadır.

İzlenmesi gereken uç birimin belirli aralıklarda yoklanması amacıyla sistem üzerinde bir zamanlayıcı oluşturulmaktadır. Akka araç kitinin altyapısında bulunan zamanlayıcı yapısı 4.1.3 bölümünde de bahsedildiği gibi 8 ay ve sonrası değerler için uygun değildir. Oluşturulan prototipte hata alma ya da tutarsız çalışmaların önüne geçebilmek için Akka Quartz Scheduler yapısı kullanılmıştır. Bu yapı ile sistem konfigurasyonu ile belirlenen zaman aralığı için bir zamanlayıcı başlatılır ve zamanlanın her gerçekleştiğinde Denetçi aktöre ilgili kaynağı içeren bir 'Fetch' mesajı iletilir. Denetçi aktör ise bu mesajı aldığında, çocuk aktör olarak Getirici

aktör oluşturarak görevini yerine getirmektedir. Değişikliklerin internet üzerindeki bir kaynak üzerinden elde edilmesi için Getirici aktör içerisinde AsyncHttpClient isimli bir istemci kullanılmaktadır. Bu sayede veri elde edildikten sonra bir üst aktöre gönderilir. Daha sonra ise Getirici aktör, sistem kaynaklarını etkin kullanmak amacıyla kendisini durdurmaktadır ve sistemin genel akışına devam edilmektedir.

Denetçi aktör ilgili uç birime ait veriyi içeren mesajı aldıktan sonra çocuk aktör olarak bir Dedektör aktör oluşturup veriyi göndermektedir. Takip edilen uç birime ait güncel veriyi alan Dedektör aktörün görevi ise; daha önce sistem tarafından tespit edilen değişimin haricinde bir değişimin olup olmadığını belirlemektir. Bunu belirleyebilmek için Redis veritabanında ilgili uç birime ait bir kayıt sorgulaması gerçekleştirir. Daha önce bir kayıt yoksa ilk defa zamanlama yapıldığını düşünerek benzersiz bir değişiklik bulduğunu kabul etmektedir. Eğer daha önce bir kayıt varsa ilgili kaydın mevcut veri ile eşleşmesini kontrol etmektedir. Eşleşiyorsa bir değişim olmadığını düşünür, eşleşmiyorsa benzersiz bir değişiklik bulduğunu kabul etmektedir. Bulunan her benzersiz değişiklik için Redis veritabanındaki ilgili uç birime özel kayıt yeni değişiklik ile güncellenir ve 'Yayımcı' aktöre iletilmektedir.

Yayımcı aktör tarafından alınan benzersiz veriye ait konu dizisi, Redis veritabanından çekilir ve Apache Kafka sistemine abonelere dağıtması amacıyla ilgili kayıt gönderilir.

4.2.2 Abone Uygulaması

Değişiklik izleme prototipi ile internet üzerinde bir uç birime ait kaynakta bulunan değişiklikleri izlemek için bir abone uygulaması çalıştırılmalıdır. Bu uygulama sayesinde uç birim ile ilişkili bir konu dizisine abone olunabilmekte ve ilgili değişiklikler alınabilmektedir. Bunun için 'DigestionKafka' isimli abone uygulaması, değişikliklerin izleneceği kaynağa ait bilgiler verilerek çalıştırılmalıdır. İlgili bilgiler ile aboneye ait bir akış oluşturularak talep edilen konu dizisine ait değişimler alınmaktadır. Bir değişim ulaştığında, 'Tüketici' aktöre iletilmekte ve değişim kaydı ile ilgili işlemler aktör aracılığıyla işlenmektedir. Aktörün akıştan daha fazla kayıt alabilmesi için elindeki işi tamamladığına dair bir bilgilendirme mesajını(Acknowledgement), kendisine mesaj gönderen aktöre iletmelidir. Bu sayede akışın iki tarafında da bir şişme meydana gelmeden verilerin işlenebilmesi sağlanmaktadır.

5 DEĞİŞİKLİK İZLEME SİSTEMİNİN DEĞERLENDİRİLMESİ

Bu bölümde, oluşturulan değişiklik izleme prototipinin bir kullanım koşulu üzerinden çalıştırılıp, elde edilen değişikliklerin kullanıcılara ölçeklenebilir bir şekilde iletilmesai amaçlanmıştır. Geliştirimiş olan prototip değişiklik izleme altyapısı, PubMed üzerinde denenmek üzere Macbook Pro (2015) 2.7 GHz Intel Core i5 İşlemci 8 GB 1867 MHz DDR3 Ram sistemine sahip bir bilgisayar üzerinde başarılı bir şekilde çalıştırılmıştır.

5.1 Kullanım Alanı

Kullanım alanı olarak sağlık bilimleri konusunda yapılan uluslararası çalışmalar, yayımlanan makaleler, en son gelişmelerin bulunduğu ücretsiz bir biyomedikal veritabanı olan PubMed[16] seçilmiştir. PubMed'in yazılım geliştiricilere sağladığı kapsamlı API sayesinde arayüz aracılığıyla kullanıcıların gerçekleştirdiği çoğu işlevselliğin yazılımsal olarak da kullanılması sağlanmaktadır.

Kullanıcılar, PubMed üzerinde belirledikleri arama kriterleri eşliğinde medikal yayın taraması yapmaktadırlar. Fakat günümüz dünyasında yapılan çalışmalar eşliğinde aynı konu üzerinde her geçen gün onlarca çalışma yayımlanmaktadır. Kullanıcıların güncel çalışmaları takip etmeleri zaman alan ve yorucu bir etkinliğe dönüşmektedir. Bu amaçla kullanıcıların PubMed üzerinde belirledikleri arama kriterlerine göre oluşturulan benzersiz bir uç birimin takip edilmesi ve bulunan sonuçlardaki değişikliklerin tespit edilmesi geliştirilen prototip ile ölçeklenebilir bir şekilde sağlanmaktadır. Değişikliklere sahip olan yayınların detaylarına, yayınların kimlikleri(id) kullanılarak PubMed sisteminde bulunan ve yayın detaylarını getiren bir diğer servis aracılığıyla erişilmiştir. Bu sayede kullanıcılar, yeni eklenen yayınlara ait başlık, özet vb. gibi çeşitli özel alanların takibini sağlamaktadır.

Geliştirilen prototip ile PubMed API üzerinden dönen XML cevabı abone uygulama tarafında ayrıştırılmaktadır. Prototip birden çok ayrıştırıcı eklenebilecek şekilde esnek bir mimaride tasarlandığından dolayı ileride HTML, Text, JSON, YAML vb. gibi oldukça yaygın Web kaynaklarının kullanımını da kolaylıkla desteklemiş olacaktır.

---

[16]PubMed, https://www.ncbi.nlm.nih.gov/pubmed

5.2 Yayımcı Tarafı

PubMed medikal veritabanı üzerinde belirlenen bir terime karşılık aratılan sonuçlarda yeni bir yayın olduğunu tespit edebilmek için oluşturulan prototip içerisindeki yayımcı uygulamasına değişikliklerin izlenmesi talep edilen uçbirim(endpoint) iletilmelidir. Tasarlanan aktör hiyerarşisinde en üst seviyede bulunan 'Controller' isimli aktöre kaynağın gönderilmesini temsilen oluşturulan `Submit(Resource)` yapısı kullanılmaktadır. Gönderilecek PubMed uçbirimi ise db, term, retmax ve api_key parametrelerini içermektedir. Uçbirim içerisinde yer alan db alanı ncbi.nlm.nih.gov adresindeki kullanılabilir veritabanları arasından bir veritabanının seçilmesi için kullanılmaktadır. Bizim oluşturduğumuz kullanım koşulu için pubmed olarak belirtilmiştir. Uçbirimde bulunan term parametresi medikal yayınlarda çeşitli kriterlere göre aranacak yayınlardaki arama terimi için kullanılmaktadır. Uçbirimden dönen sonuçlar varsayılan olarak 20 adet sonuç döndürmektedir. Fakat değişiklikleri algılayabilmemiz için ilgili arama terimi ile eşleşen tüm sonuçları elde edip daha önceki verilerle karşılaştırmamız gerekmektedir. PubMed API içerinde retmax parametresi kullanılarak dönebilecek olan sonuç sayısını en yüksek değere atanmıştır. Son olarak uçbirimin içermesi gereken api_key alanı ile PubMed API tarafından kullanıma açılan servislerini kullanma yetki anahtarını belirtmemiz gerekmektedir.

```
controller ! Submit(
  Resource(
    "https://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.fcgi
    ?db=pubmed
    &term=asthma[mesh]+AND+leukotrienes[mesh]+AND+2009[pdat]
    &retmax=999999999
    &api_key=my-private-token"
  )
)
```

Yukarıda belirtilen yayımcı tarafının alacağı kayıt mesajı ile sistemde ilgili uçbirime ait değişiklik tespiti için çalışmalar başlatılmış olacak ve tespit edilen değişikliklerin abonelere dağıtılması amacıyla Apache Kafka sistemine gönderilecektir.

PubMed API'nin sunduğu servis kullanım limiti normal kullanıcılar için saniyede 3 istek ile sınırlıdır. Bu rakam kayıtlı kullanıcılar için ise saniyede 10 istek olarak belirlenmiştir. Yapılan testler sonucunda oluşturulan prototip PubMed API'nin sağladığı saniyede 10 adet istek limitine istisnai bir durum ile ya da bir hata ile karşılaşılmadan başarılı bir şekilde limite ulaşmıştır. Oluşturulan prototipe

daha fazla değişiklik tespit edilmesi istenen kaynak eklendiğinde ise PubMed API tarafından servis kullanım limitinin aşıldığına dair bir geri bildirim ile 429 HTTP durum koduna sahip 'Too Many Requests' bilgilendirme cevabı iletilerek istenilen servis çalıştırımına ait cevap alınamamıştır.

5.3 Abone Tarafı

PubMed medikal veritabanı üzerinde belirlenen bir terime karşılık aratılan sonuçlarda yeni bir yayın olduğunu tespit edebilmek için yayımcı uygulamasına değişiklik tespiti için arama yapılacak uçbirime ait bir mesaj yollandıktan sonra yayımcı tarafının tespit ettiği değişikliklere abone olmamız gerekmektedir. Bunun için ilk olarak Apache Kafka sisteminde uçbirim ile eşleşen konu dizisine abone olunmaktadır. Abone olunduktan sonra eğer bir değişiklik iletimi olursa elde edilecek olan veri aşağıdaki yapıya benzer bir içeriğe sahip olacaktır.

```
<eSearchResult>
  <Count>56</Count>
  <RetMax>20</RetMax>
  <RetStart>0</RetStart>
  <IdList>
    <Id>20113659</Id>
    <Id>20074456</Id>
    <Id>20046412</Id>
    <Id>20021457</Id>
    <Id>20008883</Id>
    <Id>20008181</Id>
    <Id>19912318</Id>
    <Id>19897276</Id>
    <Id>19895589</Id>
    <Id>19894390</Id>
    <Id>19852204</Id>
    <Id>19839969</Id>
    <Id>19811112</Id>
    <Id>19757309</Id>
    <Id>19749079</Id>
    <Id>19739647</Id>
    <Id>19706339</Id>
    <Id>19665766</Id>
    <Id>19648384</Id>
    <Id>19647860</Id>
  </IdList>
  <TranslationSet>
    <Translation>
      <From>asthma[mesh]</From>
      <To>"asthma"[MeSH Terms]</To>
      </Translation>
    <Translation>
      <From>leukotrienes[mesh]</From>
      <To>"leukotrienes"[MeSH Terms]</To>
    </Translation>
  </TranslationSet>
  <TranslationStack>
    <TermSet>
      <Term>"asthma"[MeSH Terms]</Term>
      <Field>MeSH Terms</Field>
      <Count>125438</Count>
      <Explode>Y</Explode>
    </TermSet
    <TermSet>
      <Term>"leukotrienes"[MeSH Terms]</Term>
      <Field>MeSH Terms</Field>
      <Count>14175</Count>
      <Explode>Y</Explode>
    </TermSet>
    <OP>AND</OP>
    <TermSet>
      <Term>2009[pdat]</Term>
      <Field>pdat</Field>
      <Count>877016</Count>
      <Explode>N</Explode>
    </TermSet>
    <OP>AND</OP>
  </TranslationStack>
  <QueryTranslation> "asthma"[MeSH Terms] AND "leukotrienes"[MeSH Terms] AND
2009[pdat] </QueryTranslation>
</eSearchResult>
```

PubMed API'den gelen XML verisi üzerindeki yayın kimliği listesindeki(IdList) her bir kimliğin değerinin değişiklik izleme prototipi tarafından biliniyor olması gerekmektedir. Bir XML verisi içerisinden ilgili veriyi içeren yol(path) verilerek kullanılabilir bir veri elde etmek amacıyla XmlExtractor adında bir aktör oluşturulmuştur. Bu sayede elde edilen yayın kimlikleri eski verilerle karşılaştırılarak yeni eklenen medikal makalelere ait kimlikler tespit edilmektedir.

Elde edilen yani medikal makale kimliklerini kullanarak makale detaylarına ulaşılması gerekmektedir. Bunun için PubMed'in sağladığı diğer bir servis yeni eklenen her bir kimlik için çağrılmalı ve makale detayları elde edilmelidir. Bu amaçla tüm yeni kimlikler için bir Getter aktör PubMed kaynağını(www.ncbi.nlm.nih.gov/pubmed/1234567?report=abstract&format=text) çekmesi için oluşturulmakta ve makale detayları Web üzerinden AsyncWebClient aracılığıyla elde edilmektedir.

Abone tarafındaki uygulamanın her abone için ayrı ayrı çalıştırılacağı göz önünde bulundurulduğunda, ölçekleme ile ilgili sorun teşkil edecek bir durum bulunmamaktadır.

6 SONUÇ VE TARTIŞMA

Web üzerinde üretilen verideki değişimlerin birden fazla alıcıya hızlı ve ölçeklenebilir bir şekilde ulaştırılması gerekmektedir. Veriyi üreten ve tüketen taraflar genellikle birbirinden ayrı ve iletişim olanakları kısıtlıdır. Geniş ölçüde kullanıma sahip istek/cevap paradigması ve bu yöntem kullanılarak oluşturulan çekme temelli çözümler ne yazık ki bu gereklileri karşılayamamaktadır. İtme temelli yaklaşımların sahip olduğu dinamiklik ise yayımcı/abone paradigması ile çok iyi bir şekilde desteklenebilmektedir. Bu tezde ölçeklenebilir bir şekilde RESTful Web Servislerinin monitor edilmesini ve tespit edilen değişikliklerin bir yayımcı/abone sistem ile talep eden istemcilere iletilmesi sağlanmıştır.

Web kaynaklarındaki değişikliklerin tespit edilip ölçeklenebilir bir şekilde izlenmesi amacıyla oluşturulmuş olan prototip, takip edilmesi istenilen kaynağa ait benzersiz adresi alarak belirlenen zaman aralıklarına kaynaktaki değişiklikleri algılamak amacıyla kaynağın anlık durumlarını monitör etmektedir. Kullanıcı izlemek istediği kaynağa ait adresi prototipe ileterek değişiklik izleme mekanizmasının başlatılmasını sağlamaktadır. Prototip belirlenen zaman aralıklarında değişikliklerin takibi için adresi kontrol edip, tespit edilen değişiklikleri talep eden tüm abonelere ölçeklenebilir bir şekilde iletmektedir. Prototip yayımcı ve abone olmak üzere iki ana bileşenden oluşmaktadır. Tespit edilen değişiklikler yayımcılar tarafından Kafka sistemine iletilir. Değişiklikleri elde etmek isteyen kullanıcılar, değişikliklerin bulunduğu adrese abone olarak ilgili değişikliklerin iletileceği aboneler arasına katılabilmektedir.

Hem mevcut fiziksel sunucu üzerinde ölçeklemeye (scale up) hem de sunucular kümesi üzerinde (scale out) ölçeklemeye olanak tanıyan ve teknoloji devleri tarafından işlevselliği kanıtlanmış olan Akka araç kiti kullanılarak ölçeklenebilirlik sağlanmıştır. Bilgisayar bilimlerinde oldukça önemli bir yere sahip olan ve ölçeklemeye katkı sağlayan kaygıların ayrılması ilkesi eşliğinde yayımcı/abone sistemi kullanılarak; geliştirlen prototip ile yayımcı ve abone tarafları birbirlerinden bağımsız iki uygulama olacak şekilde tasarlanmıştır. Ölçeklenebilir, dağıtık ve yüksek verimli bir sistem sağlayan konu tabanlı Apache Kafka yayımcı/abone sistemi olarak kullanılarak yayımcı ve abone uygulamalarının entegrasyonu gerçekleştirilmiştir.

Yapılan ölçeklenebilirlik testleri sonucunda yayımcı tarafında PubMed API'sinin sunduğu saniyede on adet olan üst limit istek seviyesine başarılı bir şekilde ulaşılmıştır. Bu değerin üzerinde bulunan istekler için fazla sayıda istek

atıldığına dair 429 durum kodlu bir hata mesajı PubMed API tarafından HTTP cevabı ile alınmıştır. Bu sayede kullanılan API'nin sağladığı istek sayısının üst limit değerleri denenerek sistemin örnek kullanım koşulu altında beklenen kriterleri karşıladığı tespit edilmiştir.

İleride prototipin kapsamını arttırmak amacıyla normal kullanıcıların da prototipi kullanarak PubMed üzerinden aramalarını takip edebilmesi için bir kullanıcı arayüzü (UI) oluşturulması, hedef kitleyi büyüterek daha fazla kullanıcıya hitap edilmesine olanak tanıyacaktır. Tespit edilen değişikliklerin yayımcı/abone sistemine yollanmadan önce kalıcı bir depolama alanında depolanması ile bu veriler ileriki çalışmalarda analiz amacıyla kullanılabilecektir. Web kaynaklarındaki çeşitliliği göz önünde bulundurduğumuzda kaynakların genellikle HTML, JSON ve XML tiplerinde olduğu görülmektedir. Abone tarafındaki uygulamanın tüm bu tipler ve erişebilir bir adrese sahip her hangi bir kaynak için değişim takibi yapabilmesi prototipin kullanım alanını arttıracaktır. Tespit edilen değişikliklerin kullanıcılara e-posta, sms veya anlık bildirim ile iletilmesi için ek bir bildirim modülü oluşturulması kullanıcılar açısından daha esnek bir takip imkanı sağlayacaktır. Ayrıca değişim tespiti algoritmalarından faydalanılması, sistem kaynaklarının ve ağın daha etkin bir şekilde kullanılmasını sağlamış olacaktır.

KAYNAKLAR DİZİNİ

Agha, G.: 1990, Concurrent object-oriented programming, *Communications of the ACM* **33(9)**, 125

Altınel, M. and Franklin, M. J.: 2000, Efficient filtering of xml documents for selective dissemination of information, in *Proc. of the 26th Int'l Conference on Very Large Data Bases (VLDB), Cairo, Egypt*

Barazzutti, R., Heinze, T., Martin, A., Onica, E., Felber, P., Fetzer, C., Jerzak, Z., Pasin, M., and Rivière, E.: 2014, Elastic scaling of a high-throughput content-based publish/subscribe engine, in *2014 IEEE 34th International Conference on Distributed Computing Systems*, IEEE, 567–576 pp

Boldi, P., Codenotti, B., Santini, M., and Vigna, S.: 2004, Ubicrawler: A scalable fully distributed web crawler, *Software: Practice and Experience* **34(8)**, 711

Buttler, D., Rocco, D., and Liu, L.: 2004, Efficient web change monitoring with page digest, in *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, 476–477 pp

Carzaniga, A., Rosenblum, D. S., and Wolf, A. L.: 2001, Design and evaluation of a wide-area event notification service, *ACM Transactions on Computer Systems (TOCS)* **19(3)**, 332

Chan, C.-Y., Felber, P., Garofalakis, M., and Rastogi, R.: 2002, Efficient filtering of xml documents with xpath expressions, *The VLDB Journal—The International Journal on Very Large Data Bases* **11(4)**, 354

Chandra, T. D. and Toueg, S.: 1996, Unreliable failure detectors for reliable distributed systems, *Journal of the ACM (JACM)* **43(2)**, 225

Cherukuri, R.: 2018, *Realtime websites with publication and subscription*, US Patent 10,002,202

Cho, J. and Garcia-Molina, H.: 2002, Parallel crawlers, in *Proceedings of the 11th international conference on World Wide Web*, 124–135 pp

Cho, J. and Ntoulas, A.: 2002, Effective change detection using sampling, in *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, Elsevier, 514–525 pp

Coffman Jr, E. G., Liu, Z., and Weber, R. R.: 1998, Optimal robot scheduling for web search engines, *Journal of scheduling* **1(1)**, 15

Corsaro, A.: 10, reasons for choosing opensplice dds," 2009, *URL http://www.slideshare.net/Angelo.Corsaro/10-reasons-for-choosing-opensplice-dds*

KAYNAKLAR DİZİNİ (devam)

Corsaro, A., Querzoni, L., Scipioni, S., Piergiovanni, S. T., Virgillito, A., et al.: 2006, Quality of service in publish/subscribe middleware, *Global Data Management* **19(20)**, 1

Cugola, G., Jacobsen, H., et al.: 2002, Using publish/subscribe middleware for mobile systems, *ACM SIGMOBILE Mobile Computing and Communications Review* **6(4)**, 25

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M.: 2003, The many faces of publish/subscribe, *ACM computing surveys (CSUR)* **35(2)**, 114

Francisco-Revilla, L., Shipman III, F. M., Furuta, R., Karadkar, U., and Arora, A.: 2001, Perception of content, structure, and presentation changes in web-based hypertext, in *Proceedings of the 12th ACM conference on Hypertext and Hypermedia*, 205–214 pp

Freivald, M. P. and Noble, A. C.: 2000, *Unique-change detection of dynamic web pages using history tables of signatures*, US Patent 6,012,087

Gascon-Samson, J., Garcia, F.-P., Kemme, B., and Kienzle, J.: 2015, Dynamoth: A scalable pub/sub middleware for latency-constrained applications in the cloud, in *2015 IEEE 35th International Conference on Distributed Computing Systems*, IEEE, 486–496 pp

Group, O. et al.: 2004, Data distribution service for real-time systems specification, *Object Management Group, Tech. Rep. Version 1.0*

Haller, P. and Odersky, M.: 2006, Event-based programming without inversion of control, in *Joint Modular Languages Conference*, Springer, 4–22 pp

Haller, P. and Odersky, M.: 2009, Scala actors: Unifying thread-based and event-based programming, *Theoretical Computer Science* **410(2-3)**, 202

Kermarrec, A.-M. and Triantafillou, P.: 2013, Xl peer-to-peer pub/sub systems, *ACM Computing Surveys (CSUR)* **46(2)**, 16

Khan, L., Wang, L., and Rao, Y.: 2002, Change detection of xml documents using signatures, in *Proceedings of Workshop on Real World RDF and Semantic Web Applications*

Kim, H., Kang, S., and Oh, S.: 2015, Ontology-based quantitative similarity metric for event matching in publish/subscribe system, *Neurocomputing* **152**, 77

Kreps, J., Narkhede, N., Rao, J., et al.: 2011, Kafka: A distributed messaging system for log processing, in *Proceedings of the NetDB*, 1–7 pp

KAYNAKLAR DİZİNİ (devam)

Larson, J.: 2008, Erlang for concurrent programming., *ACM Queue* **6(5)**, 18

Li, M., Ye, F., Kim, M., Chen, H., and Lei, H.: 2011, A scalable and elastic publish/subscribe service, in *2011 IEEE International Parallel & Distributed Processing Symposium*, IEEE, 1254–1265 pp

Liu, L., Pu, C., and Tang, W.: 2000, Webcq-detecting and delivering information changes on the web, in *Proceedings of the ninth international conference on Information and knowledge management*, 512–519 pp

Olston, C. and Widom, J.: 2002, Best-effort cache synchronization with source cooperation, in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, 73–84 pp

Pietzuch, P. R. and Bacon, J. M.: 2002, Hermes: A distributed event-based middleware architecture, in *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*, IEEE, 611–618 pp

Ramasubramanian, V., Peterson, R., and Sirer, E. G.: 2006, Corona: A high performance publish-subscribe system for the world wide web., in *NSDI*, Vol. 6, 2–2 pp

Rocco, D., Buttler, D., and Liu, L.: 2003, Page digest for large-scale web services, in *EEE International Conference on E-Commerce, 2003. CEC 2003.*, IEEE, 381–390 pp

Rosenblum, D. S. and Wolf, A. L.: 1997, A design framework for internet-scale event observation and notification, in *Software Engineering—ESEC/FSE'97*, Springer, 344–360 pp

Schmidt, D. C. and van't Hag, H.: 2008, Addressing the challenges of mission-critical information management in next-generation net-centric pub/sub systems with opensplice dds, in *2008 IEEE International Symposium on Parallel and Distributed Processing*, IEEE, 1–8 pp

Sharma, Y., Ajoux, P., Ang, P., Callies, D., Choudhary, A., Demailly, L., Fersch, T., Guz, L. A., Kotulski, A., Kulkarni, S., et al.: 2015, Wormhole: Reliable pub-sub to support geo-replicated internet services, in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 351–366 pp

Thein, K. M. M.: 2014, Apache kafka: Next generation distributed messaging system, *International Journal of Scientific Engineering and Technology Research* **3(47)**, 9478

KAYNAKLAR DİZİNİ (devam)

Vinoski, S.: 2007, Concurrency with erlang, *IEEE Internet Computing* **11(5)**, 90

Yadav, D., Sharma, A., and Gupta, J.: 2007a, Architecture for parallel crawler and algorithm for web page change detection, in *IEEE Proceeding of 10th International Conference on IT, Dec*, 17–20 pp

Yadav, D., Sharma, A., and Gupta, J.: 2007b, Change detection in web pages, in *10th International Conference on Information Technology (ICIT 2007)*, IEEE, 265–270 pp

Yadav, D., Sharma, A., and Gupta, J.: 2008, Parallel crawler architecture and web page change detection, *WSEAS Transactions on Computers* **7(7)**, 929

Zhang, S., Dyreson, C., and Schema-Less, R. T. S.: 2004, *Semantics-Based Change Detection for XML Detection, Washington State University, Pullman, Washington, USA WISE 2004, LNCS 3306*

Zupan, N., Zhang, K., and Jacobsen, H.-A.: 2017, Hyperpubsub: a decentralized, permissioned, publish/subscribe service using blockchains, in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Posters and Demos*, ACM, 15–16 pp