Dağıtık Ağ Güvenliği Uygulamaları
ve
Başarım İyileştirmeleri

H. Kerem Cevahir [1] Burak Oğuz[2]

[1,2]Medra Teknoloji, Ankara

[1]e-posta: hkerem@gmail.com [2] e-posta: burak.oguz@gmail.com

Özetçe

Ağ güvenliği sektöründe dağıtıklık, yeni nesil ürünlerinin tümünde aranan bir özellik haline gelmektedir. Bilgi güvenliği sektörünün artan gereksinimleri ile birlikte tüm ağ trafiği üzerinde yapılan veri inceleme operasyonları da artmaktadır. Bu durum kurumun ölçeğiyle birlikte, bilişim kaynaklarının tüketimini önemli bir ölçüde etkilemektedir. Sonuç olarak, büyük ölçekli ağlarda yapılacak veri inceleme operasyonları, tek bir bilgisayardan çok daha fazla fiziksel kaynak gerektirmektedir. Bu da aynı zamanda hata riskini arttırmaktadır. Dolayısıyla, dağıtıklık ile birlikte performans, eş zamanlılık, ölçeklenebilirlik ve hata toleranslılık ağ güvenliği uygulamaları için temel özellikler haline gelmektedir. Bu makalede, bu özellikleri bir ağ güvenliği uygulamasına sağlayabilecek bir uygulama çatısı için gereksinimlerden bahsedilecek, bu uygulama çatısının geliştirilmesinde kullanılacak altyapı için alternatifleri ile karşılaştırmalar yapılacak, uygulama çatısı için tüm bir model önerilecek ve iş mantığı doğrultusunda yapılabilecek başarım iyileştirmeleri için yönlendirmeler sunulacaktır.

1. Giriş

Günümüzde ağ güvenliği sektörü, klasik güvenlik anlayışını terk edip, bilgi odaklı güvenlik ürünlerine yönlenmektedir. Başka bir deyişle, sadece kaynak-hedef ağ öğelerini engellemek üzerine kurgulanmış ürünler yerine, bunları içerik analizi ve sınıflandırması ile destekleyen ürün türlerine doğru yönelmektedir. Bilgi sızıntısı saptama ürünleri, web uygulamaları için güvenlik duvarları, içerik filtreleme ürünleri vs. gibi örneklerin tümü bu kapsamdadır.

Uygulama seviyesinde (ISO yedinci katman) ayrıştırma ve analiz işlemleri, taşıma seviyesinde (ISO dördüncü katman) yapılan engelleme ve bağlantı takibi yapma işlemlerine göre daha fazla kaynak tüketir. Bu durum, mevcut donanımların büyük ölçekli ağlarda yetersiz kalmasına sebep olur. Sonuç olarak bu tip sistemlerde ölçeklenebilirlik zorunlu bir özellik haline gelmeye başlamıştır.

Ancak verimli ölçeklenebilirlik, sadece bir uygulamayı birden fazla fiziksel makine üzerinde çalıştırarak sağlanabilecek bir özellik değildir. Uygulamanın sürekliliğinin direkt olarak herhangi bir fiziksel makineye veya üzerindeki bir kaynağa bağlı olmamalıdır. Rastgele bir fiziksel makinede sorun oluştuğunda uygulama çalışmaya devam edebilmeli ve bu sorunda haberdar olabilmelidir. Ayrıca, bunları sisteme yok sayılabilir bir fazladan yük getirerek sağlanmalıdır. Tüm bunlar sağlandığında tam olarak ölçeklenebilirlikten söz edilebilir [1].

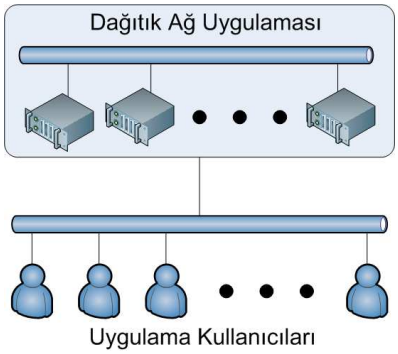
Uygulama seviyesinde analiz yapan ağ güvenliği uygulamalarında ölçeklenebilirlik ancak düzgün bir dağıtık sistem kurgusu ile sağlanabilir.

Bu bağlamda, dağıtıklık uygulama seviyesinde analiz yapan ağ güvenliği uygulamalarında bir zorunluluk olarak düşünüldüğünde, bu tip uygulamalarda dağıtıklığın sağlanabilmesi için genel geçer bir yöntem önerilebilir.

2. Dağıtık Ağ Güvenliği Uygulaması

Ağ uygulamaları, çok sayıda kullanıcıya yerel ağ içinde hizmet veren uygulamalardır. Başka bir deyişle, bir ağ uygulaması eş zamanlı olarak birden fazla kullanıcıya hizmet vermektedir.

Dağıtık sistem, bir ağ üzerinden haberleşerek ortak bir hedefi gerçekleştirmeye çalışan birden fazla otonom sistemin tümüne verilen isimdir. Dolayısıyla, dağıtık sistemden söz edildiğinde kendi aralarında mesajlaşarak organize olan uygulamalar anlaşılmalıdır.



Şekil 1: Dağıtık Ağ Uygulaması

Uygulama ağ güvenliği için kullanıldığında ise başarım gereksinimi ortaya çıkar. Çünkü ağ güvenliği uygulamaları genelde tüm ağın trafiğini kontrol etmek, yönlendirmek amaçlı kullanıldığından, dış ağ ile iç ağ arasına konumlandırılır. Dolayısıyla, uygulamanın hızı direkt olarak kullanıcının hizmete ulaşımındaki gecikmeyi etkiler.

2.1. Kullanılacak Altyapı İçin Temel Gereksinimleri

Dağıtık ağ güvenliği uygulamasının tanımının getirdiği gereksinimlerin dışında, uygulamanın hayata geçirebilmesi için gereken gereksinimler vardır. Örnek olarak normal bir eş zamanlı ağ uygulamasında geliştirici, eş zamanlılığı tutarlı bir şekilde sağlayabilmek için, iş mantığını geliştirmeye odaklanamaz. Uygulamanın iş mantığı ile ilgili kısımlarını geliştirirken dahi, eş zamanlılıkta tutarlılığı sağlamak için kilit yapıları, birbirini dışlayan durumlar gibi konulara dikkat etmek zorundadır.

Dolayısıyla, bu tip uygulamalar için kullanılacak altyapı;

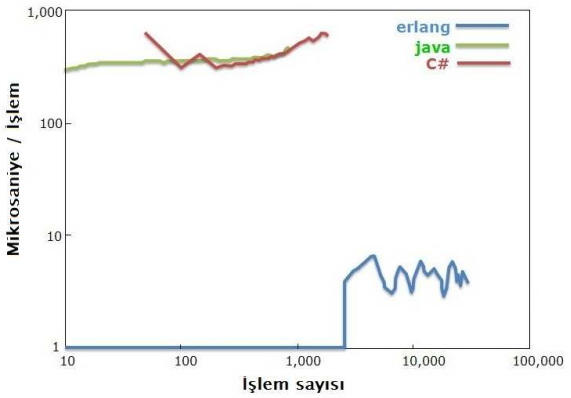
- Birden fazla iş parçacığını eş zamanlı yapabilmeli,
- Kendi benzerleriyle mesajlaşabilmeli,
- Performanslı olmalı,
- Geliştirici uygulamanın eş zamanlı tutarlılığını sağlamak için fazladan efor tüketmemelidir.

Tüm bu özelliklerin sağlanabilmesi için kullanılacak altyapının tüm bu özellikleri yerel (native) olarak sağlaması gerekmektedir.

2.2. Kullanılacak Altyapının Seçilmesi

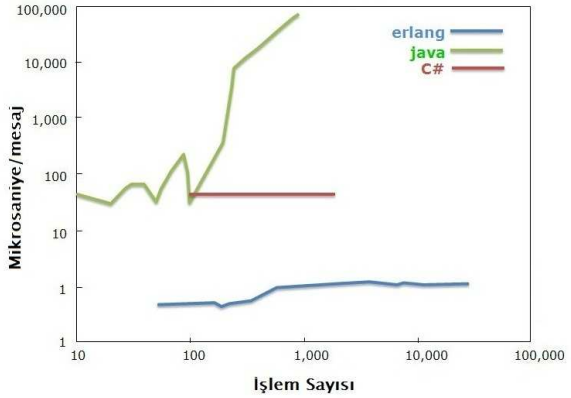
Gereksinimlere göre alternatifler incelendiğinde, Ericsson tarafından eş zamanlı, hata toleranslı, durmayan sistemler için geliştirilen Erlang dikkat çekmektedir. Dil doğal olarak gereksinimlerin tümünü desteklemektedir [2]. Ayrıca, tüm bu özelliklerin yüksek yük altında başarım ve kararlılıkları korunmaktadır [3].

Belirtilen gereksinimler için dili popüler alternatifleriyle karşılaştırmak gerekirse, işlem üretim ve mesaj aktarım başarımı en önemli iki ölçüt olacaktır. Dağıtık ağ güvenliği uygulamalarında tüm sistem içerisinde en çok kullanılan özellikler bu iki özelliktir.



Şekil 2- İşlem üretim zamanları

Şekil-2'de de görüldüğü üzere, Java ve C# ortalama 400 mikro saniyede bir işlem üretirken 3000 işleme kadar Erlang 1 mikro saniyede yeni işlem üretebilmektedir. Ayrıca 2000 – 3000 işlem üretildikten sonra Java ve C#'ta yeni işlem üretilmeye çalışıldığında problemler yaşanmaya başlamasına rağmen, Erlang 200000'in üzerinde işlem var olsa dahi sorunsuzca yeni işlem üretebilmektedir [4].



Şekil 3- Mesaj aktarım zamanları

Şekil 3'te ise üretilen işlem sayısının işlemler arası mesajlaşma süresine etkisi görülmektedir. C# 80 mikro saniyede sabit bir mesaj aktarım süresine sahipken, Java 100 işlem sayısından sonra mesaj aktarımı her yeni işlem ile birlikte yavaşlamaya başlamaktadır. 2000-3000 işlemden sonra ise, her ikisi de yine sorunlar yaşamaya başlamakta ve cevapsız kalmaktadır. Erlang ise 50.000 işlem üretildiğinde dahi, 1 mikro saniyenin altında mesaj aktarım hızını korumaktadır [4].

Bunların dışında Erlang tasarımı gereği, sistemde tek başına koşabilen herhangi bir uygulama, eş zamanlı olarak kendi kopya işlemleriyle koşturulduğunda da problem yaşamamaktadır [5]. Dolayısıyla geliştiriciye ek bir yük getirmemektedir.

2.3. Uygulama gereksinimleri

Uygulamanın birden fazla oluşumu (instance) aynı anda çalışmakta olacağından dolayı, bu oluşumları düzenleyen bir yönetici oluşum olmalıdır. Ancak, bu yönetici oluşumun hata üretme olasılığı mevcuttur. Bu durumda diğer oluşumlardan biri yöneticiliği devralmalıdır ve geride kalan diğer oluşumlar ise bu oluşumu kendilerine yeni yönetici olarak atamalıdırlar.

Bu durum akla güvenlik odaklı sorunları getirmektedir. Üçüncü bir uygulamanın bu yönetim devralma işlemini kötüye kullanabilme ihtimali vardır. Ancak, Erlang düğümlerin (node) protokol seviyesinde ortak bir güvenlik anahtarı kullanarak sağladığı güvenlik buna müsade etmemektedir. Kötü niyetli üçüncü kişinin bu anahtarı bilmeden düğümler arası iletişim protokolüne sızması pratikte mümkün değildir [6].

Hata durumunda yönetici görevinin başka bir düğüme aktarılması işlemi, her düğümün her an yönetici düğümün yerine geçebilmeye hazır olmasını gerektirir. Bu durumda yönetici düğümün her durum değişikliğinde durumu bildiren bir mesajı diğer düğümlere göndermesi ile mümkün olmaktadır. Ancak bu durum mesajının büyüklüğü sistemin toplam başarımı için büyük önem taşımaktadır. Bu mesaj

yönetici düğümün çalışma zamanında oluşturduğu eşsiz verilerden oluşmalıdır ve olabildiğince kısa olmalıdır.

Yöneticide hata oluştuğunda diğer düğümlerin bu durumdan anında haberdar olması gerekmektedir. Erlang bu gereksinimi de uygulamaya bırakmamaktadır. İletişimden olunan herhangi bir düğümdeki, her hangi bir işlemde bir hata olduğunda, bu hatanın tespiti yapılabilmektedir [2]. Bu sayede, diğer düğümlerde yeni yönetici düğüm atama işlemi başlatılabilir.

Yeni yönetici atama işlemi başarıyla tamamlandıktan sonra, sistem işlevsel olarak önceki durumundan herhangi bir farklılık göstermemelidir.

Oluşturulacak olan bu uygulama çatısı yeni bir uygulamaya kolayca uygulanabilmelidir. Geliştiricinin, tüm özellikleriyle bu uygulama çatısını kullanabilmesi için fazladan yapacağı iş yok sayılabilir olmalıdır. Dolayısıyla Erlang sunucu uygulamalarının çoğunda kullanılan `gen_server` davranışıyla uyumlu olmalıdır.

2.4. Uygulama Çatısı Fonksiyonları

Öncelikle tüm uygulama çatısında kullanılacak ve düğümler arasında paylaştırılacak mevcut durumu sarmalayan `durum` isimli kayıt tanımının oluşturulması gerekmektedir.

```
-record(durum, { isim,
                 yoneticiDugum,
                 yonetilenDugumListesi=[],
                 modulDurum,
                 modul}).
```

Bu kayıttaki `modulDurum` uygulamanın türüne ve kapsamına göre değişebilmektedir. Bazı uygulamalar için sadece bir tamsayı iken, bazıları için milyonlarca sekiz ikillik bir boyutta olabilir.

```
init( . . . )    ->
        process_flag(trap_exit, true),
        . . .
        . . .
```

Uygulama hata oluşumlarında kendini kapatmamalı, aksine hata durumu işleyebilmelidir. Hata toleranslı sistem oluşturmanın temelini bu özellik oluşturmaktadır [7].

```
handle_info({'EXIT', Dugum, Sebep}, Durum)
when Dugum==Durum#durum.yoneticiDugum ->
        { YeniYoneticiDugum,
        YeniYonetilenDugumListesi,
        YeniDurum } =
        yeni_yonetici_dugum_ata
                (Durum, Dugum),
        . . .
        . . .
        diger_dugumlere_bildir(YeniDurum)
        ,
        {noreply, YeniDurum};
```

Yönetici düğümde bir hata olduğunda, bu durumu algılayan düğümler yeni yönetici düğüm atamaya çalışırlar, bu sırada rastgele ya da uygulama alanı gereksinimlerine uygun bir yöntemle yeni düğüm seçilebilir. Ancak bu seçme işlemi yine diğer düğümler ile haberleşilerek yapılmalıdır. OTP'nin `"global"` kütüphanesi, iletişim halinde olan Erlang düğümlerin kendi aralarında herhangi bir işlemi "yarış durumu" oluşturmadan gerçekleştirmesi için gerekli altyapıyı sağlar [8]. Bu kütüphanedeki `global:trans` fonksiyonu kullanarak herhangi bir işlem tüm düğümler arasında herhangi bir yarış durumu oluşturmadan atomik olarak gerçekleştirilebilir. Bu durumda örnek bir `yeni_yonetici_dugum_ata( Durum, Dugum)` fonksiyonu;

```
yeni_yonetici_dugum_ata(Durum, Dugum) ->
        global:trans({uzak_dugum,
                yonetici_dugum_atama}
                fun() ->
                        . . .
                        . . .
                end),
        . . .
        . . .
```

şeklinde olmalıdır.

```
handle_info({'EXIT', Dugum, Sebep}, Durum)
->
        YeniListe = lists:delete(Dugum,
        Durum#durum.yonetilenDugumListesi
        ),
        YeniDurum = . . .
        diger_dugumlere_bildir(YeniDurum)
        ,
        {noreply, YeniDurum}.
```

Kümedeki yönetici düğümün dışında bir düğümde hata oluştuğunda, bu düğüm `durum` veri yapısından çıkarılmalıdır ve yeni durum diğer düğümlere bildirilmelidir.

Ayrıca hatasız uygulama akışında düğümlerin eklenebileceği ve çıkarılabileceği altyapılar oluşturulmalıdır. Her düğüm kümeye katılırken ve ayrılırken bu durumu yönetici düğüme `{katil, Dugum}` ya da `{ayril, Dugum}` benzeri mesajlar ile bildirmelidir.

```
handle_cast({katil, Dugum}, Durum) ->
        YeniDurum =
        duruma_yeni_dugum_ekle
        (Durum, Dugum),
        . . .
        . . .
        diger_dugumlere_bildir(YeniDurum)
        ,
        {noreply, YeniDurum};
```

Yönetici düğüme ilgili mesaj geldiğinde, mevcut `yonetilenDugumListesi`'ni güncellemektedir ve yeni oluşan yeni `durum` veri yapısını diğer düğümlere bildirmektedir. Böylece kümeye eklenen yeni düğümden kümedeki bütün düğümler haberdar olacaktır.

Kümeye dağıtılmış sistemden bir iş mantığına ait bir fonksiyon çağırılmak istendiğinde ise, kümedeki bir sunucu seçilip istek ona yönlendirilmeli ve cevap ondan beklenmelidir. Kümeden fonksiyona cevap verecek sunucunun seçimi için çeşitli politikalar belirlenebilir. Bu politika tamamen rastgele olabileceği gibi, belirli düğümleri belirli işler için ayırarak da çalışabilir, ya da bazı işler için

bazı düğümleri ağırlıklı olarak çağırıp aşırı yüklenme durumunda diğer düğümleri çağırabilir. Bu seçilim işlemi tamamen uygulamanın iş mantığına göre şekillenmelidir. Seçilim için kullanılacak düğümler kümesi zaten hali hazırda yönetici düğüm dâhil kümedeki tüm düğümlerde bulunmaktadır. Başka bir deyişle, sistemden fonksiyon çağıran istemci kütüphanesi, yönetici sunucudan isteği yönlendireceği bir düğümü sorup, isteği o düğüme yönlendirecektir.

```
is_mantigi1() ->
       Dugum = yoneticiden_dugum_iste(),
       gen_server:call(
              {global, Dugum},
              is_mantigi1 ).

yoneticiden_dugum_iste() ->
       Cevap = gen_server:call(
              {global, yonetici_dugum},
              is_fonksiyonu_icin_dugum )
              . . .
              . . .
              SecilenDugum.

handle_call(is_fonksiyonu_icin_dugum,
              _Kimden, Durum) ->
       Dugum = dugum_sec(
              Durum#durum
              .yonetilenDugumListesi),
       {reply, Dugum, Durum};


handle_call(is_mantigi1,
              Kimden, Durum) ->
       . . .
       . . .
       {reply, Cevap, YeniDurum};
```

`yoneticiden_dugum_iste` fonksiyonu yönetici düğüme istek yönlendirmek için bir düğüme ihtiyaç olduğunu simgeleyen bir mesaj göndermektedir. Yönetici düğüm ise bu isteği cevaplayarak istemciye isteğini yönlendirebileceği düğümü göndermektedir. Ancak, bu modelde dikkat edilecek husus sadece yeni bir düğüme ihtiyaç olduğu yönetici düğüme bildirilmektedir. Yönetici düğüm yapılacak işten bağımsız olarak düğüm seçilimi yapmaktadır. Önceden de bahsedildiği gibi koşturulacak işleme, hatta koşturulacak işlemin argümanlarına göre yapılacak düğüm seçilim işlemleri için, koşturulacak işlemin ve argümanlarının yönetici düğüme mesaj içinde bildirilmesi gerekmektedir. Bu sorun, yönetici düğüme `is_fonksiyonu_icin_dugum` gibi bir atom yapısı yerine `{is_fonksiyonu_icin_dugum, is_tanimi, ArgumanListesi}` gibi bir kapsüllenmiş yapı gönderilerek çözülebilir. Bununla beraber ilgili `handle_call` cümleciği de aynı şekilde düzenlenmelidir.

Uygulama çatısının şu durumdaki yapısı kullanımını oldukça zorlaştırmaktadır. Çünkü tanımlı iç fonksiyonlar ve iş mantığı fonksiyonları aynı modül içindedir. Dolayısıyla, kullanıcı uygulama çatısını kullanabilmek için tüm uygulama çatısı kodunu kendi modülünün içine aktarması gerekmektedir. Bu ise uygulama gereksinimlerinde istenmeyen bir durum olarak belirtilmiştir.

Uygulama çatısı kendine gelen çağrılardan, kendisinin kullanmayacağı çağrıları ilgili bir dış modüle aktaracak şekilde çalışmalıdır. Bu sayede kullanıcı tüm kodu kendi kodu ile birleştirmek zorunda kalmayacak, kendi kodunu uygulama çatısı kodundan kolayca ayırabilecektir.

Bu özelliğin sağlanabilmesi için uygulama çatısından türetilmiş her işlem kendine gelecek çağrıların aktarılacağı modülü bilmek zorundadır. Uygulama durumunu tutmak için tasarlanan `durum` veri yapısındaki `modul` alanı da hali hazırda bu iş için ayrılmıştır.

Ek olarak, iş mantığı modülü, `gen_server` davranışında ve uygulama yapısından bağımsız bir modül olacağı için bu modülün durumunun saklanması da gerekmektedir. `gen_server` davranışın saklayacağı durum uygulama çatısının durumu olacaktır. Dolayısıyla uygulama çatısının durumunun içinde dış modülün durumu da saklanmalıdır. Yine `durum` veri yapısındaki, `modulDurum` alanı bu iş için ayrılmıştır.

```
dis_modulden_cagir(Durum, Fonksiyon,
       Argumanlar) ->

       Sonuc = apply(Durum#durum.modul,
              Fonksiyon, Argumanlar),
       . . .
       . . .
       YeniDisModulDurumu = . . .,
       . . .
       . . .
       YeniDurum = Durum#durum{
       modulDurum = YeniDisModulDurumu},
       diger_dugumlere_bildir(YeniDurum),
       YeniSonuc =
              yeni_sonuc_olustur(
              Sonuc, YeniDurum),
       YeniSonuc.

handle_call(Istek, Kimden, Durum) ->
       dis_modulden_cagir(
              Durum, handle_call,
              [Istek, Kimden,
              Durum#durum.modulDurum]).

handle_cast(Istek, Durum) ->
       dis_modulden_cagir(
              Durum, handle_cast,
              [Istek,
              Durum#durum.modulDurum]).

handle_info(Mesaj, Kimden, Durum) ->
       dis_modulden_cagir(
              Durum, handle_info,
              [Mesaj,
              Durum#durum.modulDurum]).
```

Her `handle` cümleciğinin son fonksiyon tanımlaması olarak `dis_modulden_cagir` fonksiyonu çağırılmaktadır. Bu fonksiyon `erlang:apply` fonksiyonunu [9] kullanarak tanımlanmış dış modüldeki, çalışma zamanında belirlenecek isteği koşturmaktadır. Ayrıca, koşturma sonucunda dış modül

durumu olarak geri dönen veri yapısını uygulama çatısı durumuna çevirmektedir. Çevrilen durumu da mevcut uygulama çatısı durumu ile karşılaştırarak, akış doğrultusunda yeni durum olarak atamaktadır ve diğer düğümlere göndermektedir. Dolayısıyla, uygulama çatısının tutarlılığını garanti altına almaktadır.

Dış modülün uygulama çatısını kullanmaya başlaması için, uygulama çatısının dışarıya bir fonksiyon sunması gerekmektedir. Bu başlangıç fonksiyonu dış modülün ismini parametre olarak almalı ve uygulama çatısı modelinden yeni üretilecek `gen_server` oluşumundaki `durum` veri yapısının `modul` alanına atanmalıdır.

Bunların dışında örneklerde sadece isim olarak tanımlanmış fonksiyonlar, `gen_server` davranışının gerektirdiği fonksiyonlar ve uygulama ihtiyacına göre gerek diğer fonksiyonlar da, öne koyulan akış mantığı çerçevesinde tanımlanmalıdır.

2.5. Uygulama Çatısının Kullanımı

Uygulama gereksinimleri doğrultusunda, kullanıcının uygulama çatısını kullanması için gereken fazladan iş, yok sayılabilir olmalıdır. Oluşturulan uygulama çatısını kullanmak için de, kullanıcının klasik `gen_server` sunucu tasarımı üzerine ekleyeceği tek fazladan iş başlangıç fonksiyonundaki değişiklik olacaktır.

Modülün başlangıç fonksiyonunda, `gen_server:start`'a çağrı yapmak yerine uygulama çatısındaki başlatma fonksiyonuna çağrı yapılacaktır.

3. Başarım İyileştirmeleri

Yapılacak başarım iyileştirmeleri genellikle, uygulama çatısı modelinde sadece ismi belirtilen fonksiyonların tanımlarında gerçekleşmektedir. İş mantığının akışına göre şekillenecek bu fonksiyonlar, önceki bölümlerde gösterilen modeller doğrultusunda ya da tamamen iş mantığına özgü bir şekilde özelleştirilebilir.

Ancak bu fonksiyonlar arasından, kümedeki düğümlere güncellenen yeni durumu anons ederek düğümlerin eş zamanlılığını sağlayan `diger_dugumlere_bildir` fonksiyonu genel diğer tüm fonksiyonlardan daha sık çağrılmaktadır. Bu fonksiyonun tanımlanmasında sıkça düşülecek bir hata `durum` veri yapısının diğer düğümlerin tamamına olduğu gibi bir mesaj içinde dağıtılmasıdır. Bu durum uygulama ilk çalıştırıldığında henüz `durum` veri yapısı küçükken bir sorun yaratmazken, `durum` veri yapısı büyüdüğünde maliyet bir işlem olmaya başlayacaktır.

Bu noktada, bir çözüm olarak diğer tüm düğümlere `durum` veri yapısının tamamının yerine, `durum` veri yapısında ilgili değişikliği sağlayan bir fonksiyon tanımı gönderilmektedir. Uzaktaki düğüm, `durum` veri yapısını kendisindeki eski kopyası üzerinde gönderilen fonksiyon tanımını uygulamak suretiyle yeni durumu elde edebilmektedir. Ancak bu modelde gönderilecek maliyetli bir durum güncelleme fonksiyon tanımı, maliyetini düğüm sayısı kadar katlar. Dolayısıyla, fonksiyonların maliyeti düşük tutulmalıdır. Gerektiğinde basit, sadece veri yapısı üzerinde alan güncelleme işlemi yapan fonksiyon tanımları kullanılmalıdır.

Tüm küme içinde durum güncelleme konusunda, karşılaşılan başka bir sorun ise durum güncelleme sırasında oluşabilecek "yarış" durumlarıdır. Sistemin bunların önüne geçebilmesi için `diger_dugumlere_bildir` fonksiyonunda `global` kütüphanesi kullanarak gerekli kontroller yapılmalıdır. Ancak, kontrollerin ve güncellemelerin sayısı çok arttığında sistem bekleme durumunda çok zaman geçirilebilir. Bunu engellemek için, bu tip sistemlerde düğümlerin özelleştirilmesi, çağrıların yönlendirileceği düğümlerin seçilişinin bu özelleştirmeye göre yapılması ve özelleşen düğümlerin özelleşmiş `durum` veri yapıları barındırması gerekmektedir. Yine tüm bunlar uygulamanın iş mantığına göre şekillenecektir.

4. Sonuç

Dağıtık ağ güvenliği uygulamaları geliştirmek için Erlang uygun bir altyapı sunmaktadır. Sunulan bu altyapıyı kullanan ve genel dağıtık ağ uygulaması gereksinimlerini sağlayan bir uygulama çatısı modeli önermek mümkündür. Bunları üretilecek uygulamanın iş mantığı gereksinimlere göre tekrar şekillendirerek ve eğer gerekiyorsa önerilen başarım iyileştirmelerini uygulayarak performanslı, dağıtık, eş zamanlı, hata toleranslı, kolayca ölçeklenebilen bir ağ uygulaması için tüm ve kullanımı kolay bir altyapı elde edilebilmektedir.

5. Teşekkür

Makale yazarlarına ve Erlang/OTP çalışmalarında yer alan tüm araştırmacılara teşekkür eder.

6. Kaynakça

[1] Mark D. Hill, 'What is scalability?', *ACM SIGARCH Computer Architecture News*, Aralık 1990, Sayı 18 Baskı 4, Sayfa: 18-21

[2] Armstrong J., Dacker B., Lindgren T., Millroth H., Erlang White Paper Version 2.0

[3] Dacker B., Concurrent Functional Programming for Telecommunications: A Case Study of Technology Introduction, Kasım 2000.

[4] Armstrong J., http://www.sics.se/~joe/talks/ll2_2002.pdf, Sayfa: 4-5

[5] Armstrong J., Virding R., Wikström C., Williams M., (Prentice Hall, 1996, ISBN 0-13-508301-X) "Concurrent Programming in Erlang"

[6] Erlang Reference Manual, Bölüm "Distributed Erlang", http://www.erlang.org/doc/reference_manual/distributed.html

[7] Armstrong J., Programming Erlang: Software for a Concurrent World, Pragmatic Bookshelf, 2007, Bölüm "Errors in Concurrent Programs"

[8] Erlang API Reference, Modül "global", http://erlang.org/doc/man/global.html

[9] Erlang API Reference, Modül "erlang", http://erlang.org/doc/man/erlang.html