EGE ÜNİVERSİTESİ FEN BİLİMLERİ ENSTİTÜSÜ

(YÜKSEK LİSANS TEZİ)

DAĞITIK SİSTEMLERDE DİNAMİK YÜK DENGELEME

Oğuz AKAY

Uluslararası Bilgisayar Anabilim Dalı

Bilim Dalı Kodu : 619.03.03

114133

Sunuş Tarihi : 27.08.2001

Tez Danışmanı: Prof. Dr. Kayhan ERCİYEŞ

Sayın **Oğuz AKAY** tarafından YÜKSEK LİSANS TEZİ olarak sunulan "**Dağıtık Sistemlerde Dinamik Yük Dengeleme**" adlı bu çalışma, "Lisansüstü Eğitim ve Öğretim Yönetmeliği" nin 12 inci madde (c) ve (d) bentleri ve Enstitü yönergesinin ilgili hükümleri dikkate alınarak tarafımızdan değerlendirilmiş olup yapılan sözlü savunma sınavında aday oy .birliği..... ile başarılı bulunmuştur. Bu nedenle Oğuz AKAY'ın sunduğu metnin yüksek lisans tezi olarak kabulüne oy .....birliği..... ile karar verilmiştir.

27 Ağustos 2001

Jüri Başkanı : Prof. Dr. Kayhan ERCİYEŞ.....

Raportör : Prof. Dr. Turhan TUNALI.....

Üye : Prof. Dr. Sinan YILMAZ.....

Bu tezin kabulü, Fen Bilimleri Enstitüsü Yönetim Kurulu'nun .3/9/..01... gün ve ...36/44.. sayılı kararı ile onaylanmıştır.

Süleyman BORUZANLI
Enstitü Sektreteri

Prof. Dr. Alaettin TAYSUN
Enstitü Müdürü

ÖZET

DAĞITIK SİSTEMLERDE DİNAMİK YÜK DENGELEME

AKAY, Oğuz
Yüksek Lisans Tezi, Uluslararası Bilgisayar Enstitüsü
Tez Yöneticisi: Prof. Dr. Kayhan ERCİYEŞ
Ağustos 2001, 166 sayfa

Bu tezde, gerçek zamanlı dağıtık sistemler için geliştirilmiş bir iletişim protokolünun uygulaması gerçekleştirilmiş ve bu protokol üzerine dinamik bir yük dengeleme modeli tasarlanmış ve bu modelin uygulaması gerçekleştirilmiştir.

Uygulaması gerçekleştirilen protokol, kümeleme tabanlı, hiyerarşik halka protokollerinden oluşmaktadır. Halka protokolleri senkron olarak çalışmaktadır. Alt katman düğüm adı verilen işlemcilerin yer aldığı kümelerden oluşurken, üst katmanda ise bu kümeleri yöneten temsilciler bulunur. Sistemde iki ya da daha fazla katman bulunabilir. Her bir kümede bağımsız halka protokolleri çalışmaktadır. Bu yapıya ayrıca bir hata toleransı mekanizması entegre edilmiştir.

Bu protokol üzerine geliştirilen dağıtık dinamik yük dengeleme modülü, sisteme iletilen işleri düğümlere adil ve saydam bir şekilde dağıtarak, sistemin genel performansını en üst düzeye çıkarmayı amaçlamaktadır. Bu işlemi gerçekleştirirken sistemin gerçek zamanlı özelliğini de göz önünde bulundurmaktadır.

Anahtar özcükler: Dağıtık, gerçek zamanlı, kümeleme, hiyerarşik, hata toleransı, dinamik, yük dengeleme

ABSTRACT

DYNAMIC LOAD BALANCING IN DISTRIBUTED SYSTEMS

AKAY, Oğuz
MSc, International Computing Institute
Supervisor: Prof. Dr. Kayhan ERCİYEŞ
August 2001, 166 pages

In this thesis, a communication protocol designed for distributed real-time systems is implemented and a dynamic load balancing model is developed and implemented over this protocol.

The implemented protocol consists of cluster based, hierarchical ring protocols. The ring protocols are synchronous. At the lowest level in the hierarchy, there are clusters that consist of computing processors, called nodes. The higher level consists of the cluster representatives that manage the clusters of the lower level. There can be two or more levels in the hierarchy. Ring protocols in every cluster are independent of each other. Also, a fault tolerance mechanism is integrated to the protocol.

The dynamic distributed load balancing module developed over the protocol aims to maximize the overal performance of the whole system by distributing the load submitted to the system judiciously and transparently among the nodes. While performing operations to achieve this goal, the module also considers the real-time constraints of the system.

Keywords: Distributed, real-time, cluster based, hierarchical, fault tolerance, dynamic, load balancing

TEŞEKKÜR

Bu çalışma süresince yardımlarını esirgemeyen tez yöneticim sayın Prof. Dr. Kayhan Erciyeş'e teşekkürlerimi sunarım.

Ayrıca,

yardımları için sayın Prof. Dr. Turhan Tunalı'ya,

tezin ilk bölümünde birlikte çalıştığım dostum Ahmet Şahan'a,

yüksek lisans ders ve tez çalışmalarım süresince bana kolaylık gösteren Dyo A.Ş. ve Yaşar Astron Proje Ofisi'ndeki sayın yöneticilerime

ve

tüm hayatım boyunca olduğu gibi bu dönemde de bana sonsuz desteğini esirgemeyen sevgili aileme teşekkürü bir borç bilirim.

İÇİNDEKİLER

Sayfa

İÇİNDEKİLER (DEVAM)

Sayfa

İÇİNDEKİLER (DEVAM)

Sayfa

İÇİNDEKİLER (DEVAM)

Sayfa

İÇİNDEKİLER (DEVAM)

Sayfa

ŞEKİLLER DİZİNİ

Şekil Sayfa

ÇİZELGELER DİZİNİ

1 GİRİŞ

Gün geçtikçe hızını arttırarak ilerleyen bilgisayar sistemlerindeki gelişmeler bu sistemleri, pahalı ve devasa boyutlardaki bilgisayarlardan küçük ama çok hızlı mikroişlemcili ve çok daha ucuza malolan iş istasyonlarına ulaştırmıştır. Bilgisayar iletişim sistemlerindeki gelişmeler sayesinde farklı yapı ve konfigürasyonlardaki bu küçük ve güçlü bilgisayarlar birbirleriyle saniyede megabitler hatta gigabitler düzeyinde veri iletişimi gerçekleştirebilmektedirler. Bu gelişmeler sayesinde birbirine yüksek hızlarla bağlı, birden çok işlemciden (bilgisayardan) oluşan dağıtık sistemler oluşturulması çok uygun ve kolay bir hale gelmiştir. Bu işlemci birimlerinin işbirliği halinde beraberce bir işi yürütebilmesi, tek bir sistemmiş gibi işlev görebilmesi için ise özel yazılımların geliştirilmesi gerekmiştir. Farklı amaçlarla oluşturulan dağıtık sistemler için kullanım alanlarına göre çeşitli tasarımlar modellenmiştir.

Genel tanımıyla dağıtık bir sistem birden fazla bağımsız işlemcinin birlikte çalışmasıyla oluşan, bütünüyle tek bir bilgisayar gibi görünen bir yapıdır. Böyle bir sistemin iyi fiyat/performans oranı sağlaması, ölçeklenebilirliği, güvenilir olması gibi avantajlarının yanında karmaşık yazılımlar gerektirmesi, iletişim yönünden potansiyel darboğazlara sebep olabilmesi gibi bazı dezavantajları da bulunmaktadır. Buna karşın son yıllarda dağıtık sistemlere olan yönelimin hızla artmakta olduğu görülmektedir. Bunda da gelişen teknolojiyle birlikte donanım maliyetlerinin düşmesi, işlemci ve iletişim hızlarının artmasının rolü büyüktür. Modern bilgisayar sistemleri artık sıklıkla çok işlemcilidir.

Dağıtık sistemlerin beklenen avantajları sağlayabilmesi tasarımının başarısına bağlıdır. Dağıtık sistemleri tasarlarken dikkat edilmesi gereken önemli noktalardan en temeli saydamlıktır. Sistemin dağıtık yapısı bütünüyle kullanıcı ve uygulama programlarından gizlenmeli, tek bir sistem görünümü kazandırılmalıdır. Dikkat edilmesi gereken diğer bir özellik ise esneklik, yani değişimlere kolay adapte olabilme özelliğidir. Bunların yanında güvenilirlik, performans ve ölçeklenebilirlik de dağıtık sistemler tasarlanırken göz önünde bulundurulmalıdır.

Dağıtık sistemlerin özel bir kategorisi olan gerçek zamanlı dağıtık sistemler, zaman bağımlı işlerde kullanılmak üzere geliştirilmişlerdir. Gerçek zamanlı bir sistem, dış dünyadan aldığı bir takım uyartılar doğrultusunda belli bir zaman kısıtı içerisinde bir işlem yürütüp yine dış dünyaya bir tepki üretir. Bu noktada sistemin göstereceği tepkinin zamanı önem kazanmaktadır. Başka bir deyişle sistemin ürettiği cevap kadar bu cevabı ne zaman verdiği de önemlidir. Gerçek zamanlı sistemlerde zamanında üretilememiş sonuç doğru sonuç olarak değerlendirilemez.

Gerçek zamanlı dağıtık bir sistem ise, birbirlerine bir iletişim ağı ile bağlı, tümü ya da bir kısmı dış dünyadan uyarılar alan veya dış dünyaya tepkiler veren, ya da dışsal bazı cihazları kontrol eden işlemcilerden oluşur. Bu tür sistemler bir bütün olarak zaman bağımlı işler yürütürler.

Gerçek zamanlı dağıtık bir sistemi tasarlarken göz önünde bulundurulması gereken önemli noktalardan biri saat senkronizasyonudur. Sistemin en önemli parametresi olan zaman kısıtını tüm sistemin bir bütün olarak ele alabilmesi için, sistemin bağımsız işlemcilerinin senkronize bir saat sistemine sahip olmaları gerekmektedir.

Diğer bir önemli nokta hata toleransıdır. Gerçek zamanlı sistemler genellikle kritik işlemlerde kullanılırlar ve bu tür işlemler sistemde meydana gelen problemlere karşı dayanıklı olmayı, çalışmanın kesintiye ve kayıba uğramadan devam ettirilmesini gerektirirler. Gerçek zamanlı dağıtık sistemler ayrıca gerçek zamanlı bir iletişim ortamına da ihtiyaç duyarlar. Gerçek zamanlı bir iletişim ortamı, mesaj iletimini belli zaman kısıtlarında gerçekleşeceğini garanti eder ki, bu gerçek zamanlı dağıtık bir sistemin zaman bağımlı işlevini gerçekleştirebilmesi için büyük önem taşır.

Bu tezde, gerçek zamanlı ve hata toleranslı dağıtık sistemler için geliştirilmiş senkron bir iletişim protokolünün uygulaması ile bu protokol üzerine geliştirilen dinamik bir dağıtık yük dengeleme modülü ve bu modülün uygulaması yer almaktadır.

Tezin ilk bölümünde, dağıtık gerçek zamanlı bir platform oluşturmak amacıyla Tunalı, Erciyeş ve Sosyert (1998) tarafından tasarlanmış, hiyerarşik olarak çalışan, ölçeklenebilir, hata toleranslı iletişim altyapısı sunan, senkron bir halka protokolünün uygulaması gerçekleştirilmiştir.

Gerçek zamanlı dağıtık bir sistem için geliştirilmiş olan bu protokolün temel olarak şu gereksinimleri karşılaması beklenmiştir:

- Performans
- Ölçeklenebilirlik
- Hata toleransı, güvenilirlik

- Saydamlık (tek bir sistem görünümü)

Senkron bir protokol tasarlanmasındaki neden, senkron sistemlerin olay yönlendirmeli olarak çalışan asenkron sistemlere göre gerçek zamanlı işler için daha uygun olması ve daha iyi performans sağlamasıdır.

Sistem, hiyerarşik olarak iki ya da daha fazla küme tabanlı katmandan oluşmaktadır. Küme içi ve kümeler arası bağımsız senkron halka protokolleri çalışmaktadır. En alt katmandaki kümeler "düğüm" olarak adlandırılan dağıtık sistemin işlemcilerinden (bilgisayarlarından) oluşmaktadır. Her bir küme temsilciler tarafından yönetilmektedir. Aynı şekilde küme temsilcilerden oluşan, bir üst katmanda yer alan kümelerin de birer temsilcisi bulunmaktadır. Tek bir kümeden oluşan en üstteki katmanın, dolayısı ile de tüm sistemin yöneticisi "lider" olarak adlandırılmaktadır.

Oluşturulan bu hiyerarşik kümeleme sistemi, gerçek zamanlı sistemler için uygun bir altyapı sağlamakta, aynı zamanda iletişim yoğunluğunu düşürerek, iletişim altyapısında meydana gelebilecek darboğazları ve gecikmeleri de azaltmaktadır. Bunun yanı sıra her bir kümenin diğerlerinden bağımsız halka protokolüne sahip olması paralel çalışan bir iletişim ortamını da sağlamakta, bu da performası arttırmaktadır. Böylece sistemin ölçeklenebilir bir yapıya kavuşması sağlanmış olmaktadır. Sistem, yeni işlemci ya da kümelerin eklenmesi halinde bu hiyerarşik protokol sayesinde iletişim yükünü belirli sınırlar içinde tutmayı başarabilmektedir.

Sisteme entegre edilen hata toleransı mekanizması, küme içi ve kümeler arası iletişim altyapısında oluşan problemleri bertaraf

etmektedir. Çökme durumlarında devreye giren bu mekanizmada, içerisinde problem oluşan küme tekrar oluşturularak sistemin çalışması devam ettirilmektedir.

Kümeler arası halka protokolü sayesinde, periyodik olarak temsilcilerin kümelerindeki düğümlerinden topladığı bilgiler üst katmana iletilmekte ve sonunda sistem yöneticisine ulaştırılmakta, aynı şekilde sistem yöneticisinin verdiği kararlar küme temsilcilerine iletilerek sonunda en uçtaki düğümlere ulaştırılmaktadır. Bu protokol sayesinde tüm sistemin ortak bir platform halinde çalışması, başka bir deyişle bütünlüğü sağlanmaktadır.

Bahsi geçen özellikleri sağlaması amaçlanarak tasarlanmış olan sistemin uygulaması gerçekleştirilerek bu amaçları ne derece başardığının da test edilmesi sağlanmıştır.

Bu modelin misyonu, gerçek zamanlı dağıtık bir sistemin üst model ve uygulamaları için altyapı sağlamasıdır. Bu altyapı, üzerinde, saat senkronizasyonu, toplam olay sıralaması (total event ordering), grup yönetimi ve dağıtık yük izlencelemesi (distributed process scheduling) gibi dağıtık sistem fonksiyonları geliştirilmesi için uygun bir platform oluşturmaktadır.

Tezin ikinci bölümünde, sözü geçen altyapı üzerinde çalışan bir dağıtık yük dengelemesi modeli tasarlanmış ve bu tasarımın uygulaması gerçekleştirilmiştir.

Dağıtık sistemlerin, iletişim altyapısının üzerine oturtulan fonksiyonlarından biri olan dağıtık yük izlencelemesi mekanizması, sistemin kaynaklarını yöneten önemli bir parçasıdır. Dağıtık bir sistemin

işlem kapasitesinin etkin bir şekilde kullanılabilmesi, avantajlarından faydalanılabilmesi sistemin kaynak yönetiminin başarısına bağlıdır. Dağıtık işlem programlayıcısı (distributed process scheduler) sistemin kaynak yönetimi modülüdür. Bu modül, sistemin yükünü sistemin genel performansını en üst seviyede tutacak şekilde işlemci birimleri arasında adil ve saydam bir şekilde dağıtır. Saydamlıktan kasıt, yük dağıtımının işlemi sisteme yükleyen kişi ya da uygulamaya farkettirmeden gerçekleştirilmesi ve sonuçlandırılması, başka bir deyişle sistemin bütünüyle tek bir işlemci birimi gibi görünümesinin sağlanmasıdır.

Gerçek zamanlı dağıtık sistemler sözkonusu olduğunda göz önünde bulundurulması gereken bir başka parametre olarak, zaman ortaya çıkmaktadır. Gerçek zamanlı dağıtık yük dağıtımı mekanizması da bu parametreyi kapsamına dahil ederek, yük dağıtımını adil, saydam ve etkin bir şekilde gerçekleştirmesinin yanında, belirlenmiş zaman kısıtlarına bağlı kalarak gerçekleştirmek zorundadır. Bu modül, sisteme yüklenen bir işlemin belirli bir zaman dilimi içerisinde başlamasını garanti etmelidir.

Tasarlanan dağıtık yük dağıtımı modelinin, gerçek zamanlı dağıtık bir sistemin yukarıda sözü geçen yükümlülükleri yerine getirmesi amaçlanmıştır. Oluşturulan kümeleme tabanlı hiyerarşik halka yapısının üzerine, bu altyapıyı kullanan bir sistem entegre edilerek sisteme dinamik yük dağıtımı fonksiyonu kazandırılmıştır. Bu modelde, küme yöneticilerinin kendi küme üyelerinden topladıkları yük bilgileri doğrultusunda sistemin kaynaklarının kullanımının etkin bir şekilde gerçekleştirilmesi için yükü fazla olan birimlerden yükü az olan birimlere işlemlerin kaydırılması, zaman kısıtı da değerlendirmeye katılarak gerçekleştirilmektedir.

Tezin son bölümünde tasarlanan modelin uygulaması gerçekleştirilerek çalışması test edilmiş ve gereksinimleri ne ölçüde karşıladığı gözlemlenmiştir.

Tez kitapçığının ikinci bölümünde, gerçek zamanlı dağıtık sistemler ve dağıtık sistemlerde yük izlencelemesi ile ilgili temel bilgiler verilmektedir. Üçüncü bölüm, uygulaması gerçekleştirilen gerçek zamanlı dağıtık sistem modelini ve iletişim protokolünün tasarımını içermektedir. Dördüncü bölümde bu protokol üzerine yerleştirilen dağıtık işlem izlenceleme modülünün tasarımını açıklamaktadır. Tezin beşinci bölümünde üç ve dördüncü bölümde bahsedilen protokol ve dağıtık sistem modülü algoritmalarının doğruluk analizleri yapılmıştır. Altıncı bölüm, tasarımı yapılan protokol ve dağıtık işlem izlencelemesi modülünün çalışır bir uygulamasını ve bu uygulama üzerinde gerçekleştirilen testleri içermektedir. Son bölümde ise, teorik bilgiler ile uygulama sonunda varılan sonuçlar tartışılmıştır.

2 TEORİK ÇALIŞMA

2.1 Dağıtık Sistemler

Genel tanımıyla dağıtık bir sistem bağımsız bilgisayarlardan oluştuğu halde kullanıcıya tek bir bilgisayar gibi görünen bir sistemdir (Tanenbaum, 1995). Bu tanımdan iki temel özellik göze çarpmaktadır. Bunlardan ilki donanımla ilgili olup, dağıtık sistemi oluşturan bilgisayarların bağımsız (otonom) olduğudur. İkincisi ise yazılımla ilgilidir ve dağıtık sistemin kullanıcıya tek bir bilgisayarmış gibi göründüğüdür.

2.1.1 Dağıtık sistemlerin üstünlük ve zayıflıkları

2.1.1.1 Merkezi sistemlere göre avantajları

Dağıtık sistemlere olan yönelimin gerçek sebebi ekonomiyle ilgilidir. Günümüzün mikroişlemci teknolojisi sayesinde, çok küçük maliyetlerle 1980'lerin anaçerçeve bilgisayarlarından çok daha yüksek işlem kapasitesine sahip işlemciler satın alınabilmektedir. Bunun anlamı, kapasite arttırımında mali yönden en etkin çözümün daha büyük bir bilgisayar almak yerine, bir çok işlemciyi tek bir sistemde bir araya getirmek olduğudur. Sonuç olarak dağıtık sistemleri oluşturmak merkezi sistemlere göre çok daha iyi fiyat/performans oranı sunmaktadır.

Fiyat/performans avantajının yanı sıra, dağıtık sistemler merkezi sistemlerin ulaşamayacağı performans değerlerine sahip olabilmektedir. Örnek olarak, günümüz teknolojisinde 10.000 işlemciden oluşan bir

sistem oluşturmak mümkündür. Her bir işlemcinin 50 MIPS'te çalıştığını farzedersek toplamda 500.000 MIPS gibi yüksek bir işlemci gücüne sahip olabiliriz. Bu gücü tek bir işlemcide oluşturmak teorik olarak ve mühendislik açısından mümkün değildir.

Dağıtık sistemleri oluşturmadaki bir başka neden bazı uygulamaların yapısal olarak dağıtık olmasıdır. Örnek vermek gerekirse, birçok yerel şubesi olan bir süpermarket zincirini düşünelim. Her bir şube kendi çevresinden mal almakta, kendi çevresine mal satmakta ve hangi malları satacağına da bulunduğu bölgenin yapısına göre kendi karar vermektedir. Bu nedenle bu mağazaların satış ve stok durumunu merkezi bir bilgisayarda tutmak yerine, her bir mağazanın kendi yerel bilgisayarında tutmak daha anlamlı olacaktır. Sorgulama ve güncellemelerin çoğu da bu yapıda yerel olacaktır. Fakat aynı zamanda şirket üst yönetimi de mağazaların satış ve stok durumlarıyla ilgili bazı raporlar görmek isteyeceklerdir. Bunu sağlamanın bir yolu tüm sistemi, uygulama ve raporlama yazılımlarına, dolayısı ile kullanıcılara tek bir bilgisayarmış gibi gösteren, ama gerçekte bağımsız çalışan yerel bilgisayarlardan oluşan bir sistem kurmaktır.

Dağıtık sistemlerin merkezi sistemlere göre bir diğer avantajı daha yüksek bir güvenilirlik sağlamasıdır. İş yükünün birden çok bilgisyara paylaştırılması halinde bir donanım sorunu sadece bir bilgisayarın çalışmasını durduracak, sistemin diğer kısımları çalışmaya devam edeceklerdir. Nükleer reaktörler ya da hava taşımacılığında olduğu gibi kiritik uygulamalara sahip sistemlerin dağıtık olarak tasarlanması yüksek güvenilirlik sağlaması bakımından tercih edilmektedir.

Son olarak, büyüme kapasitesi bakımından da dağıtık sistemler merkezi sitemlere göre daha uygundur. Bir anaçerçeve (mainframe) satın

aldığınızda bir süre sonra iş yükündeki artış nedeniyle daha fazla işlem kapasitesine ihtiyaç duymaya başlarsanız yapmanız gereken anaçerçevenizi daha güçlü bir başkasıyla değiştirmek ya da bir tane daha satın almaktır. Fakat dağıtık bir sisteme sahipseniz kapasite artışını sisteme daha fazla işlemci ekleyerek kolayca sağlayabilirsiniz.

2.1.1.2 Kişisel bilgisayarlara göre avantajları

Mikroişlemcilerin düşük fiyat avantajı düşünüldüğünde dağıtık bir sistem kurmak yerine, herkese kullanması için bir PC vererek bağımsız bir çalışma ortamı sağlanabileceği akla gelebilir. Fakat bu, ihtiyaç duyulan bazı özelliklerden vazgeçmek anlamına gelir.

İlk olarak kullanıcıların birbirleriyle veri paylaşımında bulunmaları gerekecektir. Örnek olarak bir havayolu rezervasyon merkezindeki çalışanlar birbirlerinin yaptığı rezervasyonları görmek zorundadırlar. Bu gibi durumlarda bilgisyarların birbirlerine bağlı olması gerekmektedir. Veri paylaşımının yanında donanım birimlerinin de paylaşımına ihtiyaç duyulmaktadır (yazıcı, arşivleme cihazları vb.). Dağıtık sistemler bu tür veri ve kaynak paylaşımını sağlamaktadırlar.

Bağımsız bilgisyarları birbirine bağlama ihtiyacına sebep olan bir başka neden de iletişimdir. Dağıtık sistemler sayesinde kullanıcılar birbirleriyle elektronik iletişim araçlarıyla (e-mail gibi) kolay ve hızlı bir şekilde bağlantı kurabileceklerdir.

Son olarak, dağıtık bir sistem kurmak birbirinden izole bilgisayarlara göre çok daha esnek bir yapıdadır ve bu esneklik kaynakların etkin kullanılabilmesini sağlar. Dağıtık bir sistem sayesinde

yük paylaşımı gerçekleştirilerek boş durumda bulunan bilgisayarların işlemci gücünden de yararlanılabilmektedir.

2.1.1.3 Dağıtık sistemlerin dezavantajları

Dağıtık sistemler güçlü taraflarının yanında, bazı zayıf yönlere de sahiptirler. Bunlardan ilki yazılımdır. Dağıtık sistemlerin ihtiyaç duyduğu yazılımları geliştirmek oldukça karmaşık ve zordur. Bu nedenle bu alanda geliştirilmiş çok sayıda yazılım bulunmamaktadır.

İkinci problem, dağıtık sistemlerin ihtiyaç duyduğu iletişim ağıdır. İletişim ağında oluşabilecek veri kayıplarının tespit edilmesi ve tekrar iletiminin sağlanması gerekmektedir. Ayrıca, iletişim ağının aşırı yüklü duruma gelmesi de mümkündür. Böyle bir durumda da iletişim hattı kapasitesinin arttırılması ek donanım ve fiziksel çalışma gerektirebilecektir.

Son olarak, veri ve kaynak paylaşımı sağlaması bakımından dağıtık sistemler güvenlik açıklarına sebep olabileceklerdir. Sistemi kullanan kişilerin potansiyel olarak sistemde bir yerlerde bulunan bazı gizli bilgilere ulaşma, yetkisi olmayan kaynakları kullanma ihtimali bulunmaktadır.

2.1.2 Tasarım özellikleri

Dağıtık bir sistem tasarlarken, sistemin gereksinimleri karşılayabilmesi için göz önünde bulundurulması gereken bazı temel özellikler bulunmaktadır. Bunlar; saydamlık (transparency), esneklik

(flexibility), güvenilirlik (reliability) ve performanstır (Tanenbaum 1995).

2.1.2.1 Saydamlık

Saydamlık, sistemin en önemli özelliği olarak da sayılabilir. Çünkü saydamlık, kullanıcıların dağıtık sistemi tek bir sistemmiş gibi algılamalarını sağlar.

Saydamlık iki ayrı seviyede tasarlanabilir. Bunlardan ilki ve daha kolay olanı dağıtık yapıyı kullanıcıdan saklamaktır. Örnek verecek olursak, kullanıcı UNIX sistemindeki *make* komutunu kullanarak birçok dosyayı derlerken, bu komut işletilirken arkaplanda derleme işlemi için aynı anda birden fazla işlemci kullanıldığının, birden fazla dosya sunucudan okuma/yazma işlemi yapıldığından farkında değildir, bunları bilmesine gerek de yoktur. Tek farkettiği işlemin normalden daha hızlı tamamlandığıdır. Kullanıcının perspektifinden sistem, sanki tek bir bilgisayarmış gibi görünür.

İkinci ve diğerine göre daha alt seviyedeki saydamlık ise, dağıtık yapının uygulama programlarından gizlenmesidir. Bunun gerçekleştirmesi diğerine göre çok daha zordur. Böyle bir sistemde sistem çağırıları çok işlemcili yapıyı gizleyecek biçimde yeniden tasarlanmalıdır. Dağıtıklığı programcıdan saklamak kullanıcıdan saklamaya göre çok daha zordur.

Dağıtık sistemlerde saydamlık çeşitli açılardan ele alınabilir. Bunlardan biri olan *lokasyon saydamlığı* kullanıcının donanım ve yazılım kaynaklarının (işlemci, yazıcı, dosyalar, veritabanları, vb.) yerlerini

bilmemesidir. Kullanıcı sadece ihtiyacı olan kaynağın ismini bilir ve bu ismi kullanarak kaynağa erişir. *Taşıma saydamlığı*, kaynakların isimlerinin değiştirilmeden bir lokasyondan başka bir lokasyona taşınabilme özgürlüğünü getirir. *Çoğaltma (replication) saydamlığı*, sistemin, kullanıcıya farkettirmeden kaynakların istediği kadar kopyasını tutabilmesidir. *Ortak kullanım (concurrency) saydamlığı*, kullanıcıların farkında olmadan sistem kaynaklarını diğer kullanıcılarla paylaşmasıdır. Bunu sağlamanın bir yolu bir kullanıcı kaynaklardan birini kullanırken diğer kullanıcılara belli etmeden o kaynağın o kullanıcıya kilitlenmesi, ve kullanıcının işi bittikten sonra kaynağın sırayla diğer istekli kullanıcılara atanmasıdır. Saydamlığın uygulandığı alanlardan en zoru *paralellik saydamlığı*dır. Buna örnek olarak, bir programcının bir işi aynı anda birden fazla işlemcide yaptırmak istemesini gösterebiliriz. Sistem, böyle bir paralelliği programcıya hissettirmeden gerçekleştirebiliyorsa bu özelliği karşılıyor demektir. Bu özellik de gerçekleştirildiği taktirde dağıtık sistemlerle ilgili tüm hedeflere ulaşılmış sayılabilir.

2.1.2.2 Esneklik

Dağıtık sistemler üzerindeki çalışmalar sürmekte ve her gün değişik teknikler ortaya çıkmakta, bugün için doğru sayılan bir kavram, yarın başka bir şekilde ifade edilerek doğruluğunu ve güncelliğini yitirebilmektedir. Bu nedenle, tasarlanan dağıtık sistem esnek bir yapıda olmalı, gelişmelere ve değişmelere kolayca adapte edilebilmelidir.

Dağıtık bir sistemin tasarımında iki farklı yapı ortaya çıkmaktadır. Bunlardan ilki, sistemdeki her bir makinenin geleneksel işletim sistemi çekirdeğine benzer bir çekirdeğe sahip olması ve sistem servislerinin çoğunu kendi üzerinden karşılamasıdır. Diğer yapıda ise, çekirdek çok

küçük olmakta, sadece en temel birkaç servisi sağlamaktadır. İlk model monolithic kernel, diğeri ise microkernel olarak adlandırılmaktadır. Monolithic kernel, günümüzün işletim sistemi çekirdeğine bilgisayar ağı özellikleri ile uzak servislerin entegre edilmiş halidir. Bu yaklaşımda sistem çağırılarının çoğu çekirdek tarafından gerçekleştirilir. Bu yapıda bilgisayarların çoğu kendi lokal dosya sistemine sahiptir. UNIX işletim sisteminden türetilmiş dağıtık sistem yapıları bu özelliktedir. Monolithic kernel modelinin tek avantajı performansıdır. Sistem servislerinin çekirdek tarafından karşılanması sistemin performansını arttırmaktadır.

Yeni tasarlanan dağıtık sistemler microkernel modelini uygulamaktadırlar. Bu yapı diğerine göre daha esnektir, çünkü yaptığı iş temel olarak, süreçler-arası iletişim (interprocess communicaiton), biraz bellek yönetimi, bir miktar altseviye işlem yönetimi ve altseviye girdi/çıktı işlemlerinden ibarettir. Diğer servisleri bünyesinde barındırmaz. Diğer tüm işletim sistemi servisleri kullanıcı-seviyesi servisleri olarak gerçekleştirilmektedir. Bu servislere çağrılar ilgili sunucuya mesaj gönderilerek gerçekleştirilir. Bu modelin avantajı modüler olması, her bir servisin belli bir arayüzü olması ve bulunduğu lokasyondan bağımsız olarak tüm kullanıcılara açık olmasıdır. Ayrıca yeni servislerin eklenmesi, mevcut servislerin çıkarılması, değiştirilmesi de kolaydır.

2.1.2.3 Güvenilirlik

Dağıtık sistemlerin geliştirme amaçlarından biri bu sistemleri tek işlemcili sistemlerden daha güvenilir yapmaktır. Buradaki temel fikir, sistemdeki bazı bilgisayarlarının çökmesi halinde diğer bilgisayarların yükü paylaşarak çalışmaya devam etmesidir.

Güvenilirliğin bazı alt parçaları bulunmaktadır. Bunlardan birisi *kesintisizliktir (availability)*. Kesintisizlik, sistemin belli bir zaman dilimi içinde çalışır durumda olduğu sürenin oranını ifade eder. Kesintizliği yükseltmenin bir yolu çoklamadır (redundancy). Çoklama sistemdeki kritik donanım ya da yazılım birimlerinin aynı anda birden fazla kopyasının çalışır durumda tutulması ve birinde meydana gelen bir arıza durumunda diğer kopyasının işi kesildiği noktadan devralmasıdır. Ne kadar çok kopya bulunursa kesintisizlik o kadar yükselir, ancak kopya sayısı arttıkça bu kopyaların birbirleriyle tutarlılığının (consistency) sağlanması da o kadar karmaşıklaşır.

Güvenilirliğin bir başka parçası *güvenliktir (security)*. Dosyalar ve diğer kaynaklara yetkisiz erişim engellenmelidir. Bu problem tek işlemcili sistemlerde de yer almakla birlikte, dağıtık sistemlerin çok parçalı yapısı dolayısı ile sorunun çözümü daha da zordur.

Güvenilirlik ile ilgi bir başka kavram da *hata toleransı*dır. Bir işlemcinin aniden çöküp tekrar geri gelmesi durumunda, sistemin bu çökmeyi farketmesi, çökme sonucu oluşan veri kayıplarını gidermesi gerekmektedir.

Güvenilirliği etkileyen tüm bu fonksiyonların kullanıcıya fark ettirmeden yerine getirilmesi gereği de altı çizilmesi gereken bir noktadır.

2.1.2.4 Performans

Dağıtık sistemlerin en temel gereksinimlerinden biri genelde arkaplanda gizlenen performanstır. Sistem ne kadar saydam, esnek, güvenilir olursa olsun yavaş ve hantalsa kullanışsız durumdadır. En basit

anlamda bir program dağıtık bir sistemde çalıştırıldığında tek bir bilgisayarda çalışmasından daha yüksek bir performans sağlanmalıdır.

Birçok performans ölçütü bulunmaktadır. En çok kullanılan ölçüt cevap verme süresidir (response time). Diğer ölçütler arasında birim zamanda tamamlanan iş sayısı (throughput), sistem kullanılabilirliği (utilization) ve iletişim ağı kullanım yoğunluğu sayılabilir. Bunların yanı sıra yapılan performans ölçümü sonuçları, ölçümün karakteristiğine göre de değişim göstermektedir. İşlemci-bağımlı işleri ilgilendiren bir ölçümle, yoğun dosya işlemleri yapan işleri içeren ölçümler birbirinden farklı sonuçlar üreteceklerdir.

Performansla ilgili dağıtık sistemlerin karşı karşıya olduğu, tek işlemcili sistemlerde bulunmayan bir etken de iletişimdir. Yerel bir ağda bir mesaj gönderimi ve cevabının alınması 1 milisaniye civarı sürmekte ve bu süre protokol işlemleri nedeniyle oluştuğundan önlenememektedir. Bu nedenle performansı arttırmak için iletişim amaçlı kullanılan mesaj sayısının düşürülmesi gerekmektedir. Fakat bu sayıyı azaltmak da o kadar kolay değildir. Çünkü performansı arttırmanın en iyi yolu işlemleri paralel olarak çok işlemcide çalıştırmaktır ki bu mesaj trafiğini arttıran bir fonksiyondur.

Paralellikle ilgili dikkat edilmesi gereken bir nokta da, işlemlerin parçacık (grain) boyutudur. Küçük bir işlemi çok işlemcide işletmeye çalışmak, iletişim maliyetinden dolayı tek işlemcide çalıştırmaktan daha kötü performans sağlayabilir. Buna karşılık büyük bir işlem-bağımlı işi birden çok işlemcide çalıştırmak avantajlı olabilir. Genellikle birbirleriyle sıkı ilişki içinde olan çok sayıda küçük parçadan oluşan bir işlemi paralel olarak çok işlemcide çalıştırmak mesaj trafiğini çok fazla

arttıracaktır. Bunun yanında birbirleriyle fazla iletişimde bulunmayan büyük parçalardan oluşan işler paralel çalıştırmak için çok uygundur.

Hata toleransı da mesaj trafiğini olumsuz etkiler. Güvenilirliği arttırmanın en iyi yolu birbirleriye sıkı iletişimde bulunan kopya sunucular kullanmaktır. Çok iyi bir hata tolaransı sağlamak için bir sunucuya iletilen her mesaj kopya sunucuya da iletilerek, sunucuların birbirleriyle birebir tutarlı olması sağlanmalıdır. Bu da mesaj trafiğinin ikiye katlanması demektir.

2.1.2.5 Ölçeklenebilirlik

Dağıtık sistemlerin çoğu birkaç yüz bilgisayardan oluşacak şekilde tasarlanmaktadır. Gelecekte çok daha fazla işlemci gerektirebileceklerdir. Bu durumda 200 işlemci için tasarlanan bir sistem 200.000 işlemci ile kullanılabilecek midir? Ölçeklenebilirliği sağlamanın bir yolu merkezi birimlerden, merkezi tablolardan ve merkezi algoritmalardan kaçınmaktır. Örnek olarak 50 milyon kullanıcı için tek bir e-posta sunucusu kullanmak hiç de akılcı bir davranış değildir. Tek sunucu bu iş için yeterli işlem kapasitesi sağlasa bile, bu sunucuya olan iletişim trafiğini karşılamak da ayrı bir sorun olacaktır. Aynı zamanda böyle bir sistem güvenilirlik açısından da uygun değildir. Sunucunun çökmesi tüm sistemin çökmesi anlamına gelmektedir.

Merkezi tablolar kullanmak da merkezi birimler kullanmak kadar kötüdür. Büyük bir veri topluluğunu tek bir veritabanında tutmak, disk kapasitesi bakımından problem olmasa bile bu veritabanına yönelen yoğun iletişim trafiği sorun teşkil edecektir. Aynı zamanda bu

veritabanın kullanılmaz duruma gelmesi halinde tüm veri topluluğu ulaşılamaz hale gelecektir.

Son olarak merkezi algoritmalar da ölçeklenebilirlik açısından problem oluşturmaktadır. Tüm işlemci birimlerinden bilgi toplayarak bir işlem yapan algoritmalar büyük sistemlerde mesaj trafiğinin yoğunlaşmasına neden olacaklardır. Bu nedenle dağıtık algoritmalar kullanılmalıdır. Bu tür algoritmalarda:

- Hiçbir bilgisayar tüm sistemin durumu hakkında tam bir bilgiye sahip değildir.
- Bilgisayarlar sadece yerel bilgileri kullanarak karar verirler.
- Bir bilgisayarın çökmesi algoritmanın çalışmasını engellemez.
- Sistemde global bir saatin varlığından söz edilmez.

Bunlardan sonuncusunu biraz açmak gerekirse, dağıtık sistemde bağımsız işlemcilerin fiziksel saatlerinin tam bir senkronizasyon içerisinde olması düşünülemez. Senkronizasyon sorunu daha büyük sistemlerde daha da büyük olacaktır. Gerçek zaman değerleri kullanan bir algoritma böyle bir senkronizasyonun bulunmaması nedeniyle hatalı sonuçlar üretecektir.

2.1.3 Dağıtık sistemlerde iletişim

Dağıtık sistemlerle tek işlemcili sistemler arasındaki en büyük fark süreçlerarası iletişimdir (interprocess communication, IPC). Tek işlemcili

sistemlerde IPC büyük ölçüde paylaşmlı bellek üzerinden gerçekleştirilir. Dağıtık sistemlerde ise paylaşımlı bellek yoktur. Bu nedenle IPC mekanizması bütünüyle yeniden tasarlanmak durumundadır.

Paylaşımlı bellekten yoksun oldukları için dağıtık sistemlerde tüm iletişim mesaj iletimi ile sağlanır. Sistemdeki bir süreç A, başka bir süreç B ile iletişime geçmek istediğinde, önce kendi adres sahasında bir mesaj yaratır. Daha sonra bir sistem çağrısı yapar ve işletim sistemi bu mesajı alarak bilgisayar ağı üzerinden B'nin bulunduğu bilgisayara gönderir. B'nin bulunduğu bilgisayarın işletim sistemi tarafından alınan mesaj B'ye iletilir. Kabaca bu şekilde işleyen iletişimin altında karmaşık bir teknoloji yatmaktadır. Herşeyden önce A ve B anlaşabilmek için iletilen mesajın içerisindeki her bir bitin ne anlama geldiği üzerinde ortak bir anlaşmaya varmış olmalıdır.

Bu anlaşma konusunda çeşitli yöntemler geliştirilmiştir. İşin en temelinde fiziksel olarak 0 ve 1 bit değerlerinin elektronik açıdan ne şekilde ifade edileceğinden başlanarak, mesaj sonunun belirlenmesine, mesajların doğruluğunun tespit edilmesine, sayısal bilgilerin, karakter bilgilerin ne şekilde gösterileceğine kadar üzerinde anlaşılması gereken farklı seviyelerde birçok durum vardır. Bu tür ortak bir iletişim dili sağlayan, protokol dediğimiz kural kümeleri oluşturulmuştur. Bu seviyeleri daha kolay ele almak ve standartları belirleyebilmek için katmanlı protokoller tasarlanmıştır. OSI (Open Systems Interconnection) referans modeli de bu amaçla geliştirilmiş katmanlı bir yapıdır ve tüm dünyada iletişim için ortak bir model haline gelmiştir. Oluşturulan katmanlı yapıda her bir katman bir altındaki katmanın sunduğu servisleri kullanmakta, alt katmanın detaylarını bilmemektedir. Böylece hem modüler bir yapı oluşturulmuş, hem de karmaşıklık azaltılmış, standartların belirlenmesi kolaylaşmıştır. Ancak buna karşılık katmanlar

arttıkça iletişim maliyetleri de büyümekte, bu da performansı düşürmektedir. OSI modeli en temelde, verinin fiziksel ortamda iletimini, sinyallemesini içeren fiziksel katmandan başlayarak ve en üst seviyede uygulamaların birbirleriyle iletişim standartlarını belirleyen uygulama katmanına kadar yedi ayrı katmandan oluşur.

TCP/IP protokolü günümüzde yaygın olarak kullanılan ve Internet standardı haline gelen bir protokol kümesidir (Tanenbaum, 1996). Ağ seviyesi protokolü olan IP (Internet Protocol) mesajların bir noktadan karşı noktaya iletiminden sorumludur. Bu noktada bağlantı-temelli ve bağlantısız mesaj iletim yöntemlerinden söz etmek gerekir. Bağlantısız yapıda mesajlar herhangi bir ön hazırlık olmaksızın iki nokta arasında gönderilirler. Mesaj gönderilirken karşı tarafın mesaj alımı için hazır olup olmadığı, ya da mesaj iletimi sonunda mesajın karşı tarafa başarıyla ulaşıp ulaşmadığı gibi kontroller yapılmaz. IP protokolü de bağlantısız bir protokoldür. Buna karşılık bağlantı-temelli protokol yapısında veri iletimine başlamadan öncelikle iletimde bulunacak iki nokta arasında bir bağlantı kurulur, bir takım kurallar üzerinde anlaşmaya varılır. Bağlantı kurulduktan sonra veri iletimi gerçekleştirilir ve iletim sonunda bağlantı kesilir. IP protokolü üzerine yerleştirilen TCP (Transmission Control Protocol) bağlantı temelli bir yapıdadır. Bağlantı temelli protokoller veri iletimi için ön hazırlık yapmasından, iletilen verinin doğru bir sırada iletilmesinden, veri kayıplarını önlemesinden dolayı daha güvenilir protokollerdir. Ancak bağlantı kurulması, sonlandırılması, sıralama ve hata kontrolü gibi fonksiyonlar nedeniyle bağlantısız protokollere göre performansı düşüktür. İletişim ağının güvenilirliğinin düşük olduğu geniş alan ağlarında TCP gibi bağlantı temelli protokoller kullanılarak veri kayıpları önlenirken, hızlı ve güvenilir iletişim ortamı sunan yerel ağlarda bağlantısız protokoller gereksiz kontrollerden arınarak yüksek

iletişim performans sağlarlar. IP protokolü üzerinde yer alan bağlantsız yapıdaki UDP en yaygın kullanılanlardan biridir.

Yerel ağlar üzerine kurulan dağıtık sistemlerde katmanlı protokol yapısının çok daha basitleştirilmiş şekilleri kullanılır. Bu tür istemci/sunucu modellerinde istemci bilgisayar sunucuya isteğini belirten bir mesaj gönderir, ve buna karşılık sunucu de istemciye bu isteğe ilişkin bir cevap döndürür. Mesajların iletiminde genellikle UDP gibi bağlantısız protokoller kullanılır. Katmanların bir çoğunun ortadan kaldırılması performansı büyük ölçüde arttırmaktadır.

İstemci/sunucu modellerindeki mesaj iletiminin de çeşitli özellikleri bulunmaktadır. Bunlardan biri bloklama özelliğidir. Bloklamalı sistemlerde gönderici, mesaj gönderim komutunu çağırdıktan sonra, mesaj sistem tarafından iletilinceye kadar bekletilir. Aynı şekilde alıcı da mesaj bekleme konumunda bloklanarak mesaj gelinceye kadar bekletilir. Bloklamasız çalışan sistemlerde ise bu bekleme süreçleri ortadan kaldırılmış, bu sürelerde işlemcinin gereksiz yere boş durumda beklemesi önlenmiştir.

Bir başka özellik tamponlamadır (buffering). Tamponlamasız sistemlerde, iletişim için alıcı sürecin adres sahasında tek bir bellek sahası ayrılır ve iletilen mesajlar işletim sistemi tarafından bu sahaya kopyalanır. Böyle bir sistemde aynı anda birden fazla istemcinin sunucuya gönderdiği mesajların yönetimi, iletimi problemlere sebep olmakta, bazı mesajların kaybolmasına yol açmaktadır. Bu nedenle tamponlamalı sistemler tasarlanmıştır. Bu sistemlerde alınan mesajlar işletim sistemi tarafından bir tampon bellekte tutulmakta, böylece aynı anda gelen ve süreçler tarafından henüz alınamamış mesajların kaybolması önlenmektedir.

Mesaj iletimiyle ilgili diğer bir özellik ise güvenilirliktir. Güvenilir bir iletişim, göndericinin gönderdiği mesajın alıcı tarafından alınıp alınmadığının anlaşılması, kayıp mesajların tekrar gönderilmesini gerektirir. İstemci/sunucu sistemlerde kullanılan yaygın bir yöntem, sunucunun, istemcinin gönderdiği mesajı aldıktan sonra istemciye cevap (acknowledgement) göndermesidir. Bir başka yöntem ise mesajların alındığına dair bilgi mesajlarının işletim sistemi çekirdeği tarafından iletilmesidir.

Şimdiye kadar bahsedilen IPC örnekleri yalnızca iki süreci kapsamaktadır. Bazı durumlarda ikiden fazla süreçten oluşan işlem grupları arasında iletişimde bulunulması gerekebilir. Örnek olarak hata toleranslı bir dosya sistemini oluşturan bir grup dosya sunucuyu verebiliriz. Böyle bir sistemde istemcinin dosyalar üzerinde yaptığı değişikliklerin tüm dosya sunuculara iletilmesi gerekmektedir. Bu tür işlem gruplarının oluşturulması, yönetilmesi, grup üyelerinin kendi aralarında iletişim kurabilmeleri için grup iletişimi modelleri oluşturulmuştur. Grup iletişimi bu tezin kapsamında yer almadığından bu konudan daha fazla söz edilmeyecektir.

2.2 Gerçek Zamanlı Dağıtık Sistemler

2.2.1 Gerçek zamanlı sistemler

Çoğu program için doğruluk, komutların işletilme zamanından ziyade bu komutların mantıksal işletim sırasıdır. Eğer bir reel sayının karekökünü bulan bir program 550MHz'lik PentiumIII bir bilgisayarda doğru bir şekilde çalışıyorsa, aynı program 66 MHz'lik bir 486 makinede daha yavaş çalışacak ama aynı sonucu üretecektir.

Buna karşılık, gerçek zamanlı programlar ve sistemler zamana bağımlı olarak dış dünya ile etkileşimde bulunurlar. Sistem, dış dünyadan gelen uyarılara belli bir şekilde, belirli bir zaman aralığı içinde cevap vermek zorundadır. Sistemin cevabı doğru bile olsa, eğer bu cevabı belirlenmiş zaman aralığını aştıktan sonra üretmişse, başarısız olmuş demektir. Bu tür sistemlerde cevabın doğruluğu kadar, verildiği zaman da büyük öndem taşımaktadır (Tanenbaum, 1995).

Gerçek zamanlı bir sisteme örnek olarak bir CD çaları gösterebiliriz. Bir CD çalar, diskten okuduğu sayısal bilgileri işleyerek müzik üreten bir işlemciye sahiptir. Bu işlemci bilgileri gerektiği hızda işleyemezse müziği kesintili ve ahengi bozuk bir şekilde üretecektir. Sonuçta işleyişi doğru bile olsa zamanlaması hatalı olacağından anlamlı bir sonuç üretememiş sayılacaktır.

Gerçek zamanlı uygulamaların çoğu diğer genel amaçlı dağıtık sistemlerden çok daha yapısaldır. Genel olarak, dışsal bir cihaz bilgisayara bir uyarı üretir, bunun üzerine bilgisayar belli bir zaman aralığı içerisinde bir işlem gerçekleştirir. Bu işlem tamamlandığında sistem tekrar boş duruma döner ve bir sonraki uyarıyı bekler. Genellikle uyarılar periyodiktir, bir saat belirli zaman aralıklarında uyarılar üretir. Bazı durumlarda ise uyarılar aperiyodik, yani düzensiz aralıklarla tekrar eden bir yapıdadır. Uyarılar bazı durumlarda ise sporadik (beklenmedik) olabilir, ne zaman üretileceği belli değildir. Büyük sistemlerde genellikle birden fazla ve farklı çeşitlerde uyarılar üreten olaylar bulunabilir. Böyle durumlarda sistemin tüm olaylara zamanında cevap verebilmesi çok daha zorlaşmakta ve işlem karmaşıklaşmaktadır.

Tasarımcılar bu karmaşıklığı gidermek için genellikle her bir gerçek zamanlı cihazın önüne, sadece o cihazdan uyarı alıp ilgili işlemi

gerçekleştiren adanmış bir mikroişlemci koymayı düşünürler. Bu tasarım, gerçek zamanlı karakteristiğe sahip dağıtık bir sistemi ortaya çıkarır.

Gerçek zamanlı dağıtık sistemler genellikle bir kısmına bir dışsal cihaz bağlı, birbirleriyle bir bilgisyar ağı üzerinden iletişim kuran bir işlemci grubundan oluşurlar. Dışsal cihazlara bağlı işlemciler bu cihazların içerisine gömülmüş küçük işlemciler ya da bağımsız bilgisayarlar olabilmektedir. Bunlar, gerçek zamanlı olarak cihazlardan bilgi alırlar, bilgi üretirler, ya da bu cihazları idare ederler.

Gerçek zamanlı sistemler genellikle zaman kısıtının yapısına göre ikiye ayrılırlar: yumuşak (soft) gerçek zamanlı sistemler ve katı (hard) gerçek zamanlı sistemler.

Yumuşak gerçek zamanlı bir sistemin özelliği bazı durumlarda zaman kısıtının aşılmasının makul karşılanabilmesidir. Örnek olarak bir telefon santralının aşırı yüklü olduğu bir durumda 10.000 çağırıdan birini kaçırması ya da yanlış yönlendirmesi önemsiz olarak değerlendirilebilir. Buna karşılık katı gerçek zamanlı sistemlerde zaman kısıtının aşılması asla kabul edilemez. Çünkü böyle bir durum kritik sistemlerde felakete yol açabilir.

2.2.2 Tasarım özellikleri

2.2.2.1 Saat senkronizasyonu

Bu karakteristik gerçek zamanlı sistemin en kritik parametresi olan zamanın senkronize edilmesini içerir. Dağıtık bir sistemde bulunan her bir işlemcinin kendi yerel saati vardır ve bu tüm sistemin global bir saat

üzerinde anlaşması sorununu doğurur. Dağıtık sistemlerde saat senkronizasyonu konusu bu tezin kapsamı dışında olduğundan ayrıntılarından söz edilmeyecektir.

2.2.2.2 Tetikleme mekanizması

Gerçek zamanlı bir sistemin çalışmasını tetikleyen iki temel yöntem vardır: Olay tetiklemeli ve zaman tetiklemeli.

Olay tetiklemeli sistemlerde dış dünyada oluşan bir uyarı doğrultusunda sistem belli bir işlevi gerçekleştirir. Dışsal bir cihazda meydana gelen bir olay ona bağlı bilgisayarda bir kesinti (interrupt) oluşturur ve bunun üzerine bilgisayar ilgili işlemi başlatır. Bu yapı yumuşak gerçek zamalı sistemler için uygundur ve yaygın olarak kullanılır.

Olay tetiklemeli sistemlerin problemi aşırı yük durumunda sistemin başarısız olabilme olasılığıdır. Sistemde aynı anda birçok olay oluşursa sistem tüm bu olaylara zamanında cevap veremeyebilecektir. Özellikle katı gerçek zamanlı sistemlerde bu göze alınamayacak bir durumdur. Bu nedenle bu problemi ortadan kaldırmaya yönelik zaman tetiklemeli sistemler tasarlanmıştır. Bu tür sistemlerde bir saat belli aralıklarda kesinti oluşturarak sistemi tetiklemektedir. Her bir saat atışında dışsal cihazların değerleri alınır ve bu değerlere göre belli işlevler gerçekleştirilir. Sistemde saat atışından başka bir kesinti olmaz.

Zaman tetiklemeli sistemlerde saat atış süresinin seçimi çok önemlidir. Küçük bir saat değeri sistemi gereksiz kesintilere uğratırken, büyük bir değer sistemde oluşan ciddi olayların işlenmesinde geç

kalınmasına neden olabilir. Ayrıca hangi cihazın hangi saat atışında kontrol edileceği ve saat atışı süresi dolmadan önce oluşan olayların saat atışına kadar saklanması da dikkat edilmesi gereken durumlardır.

2.2.2.3 Tahmin edilebilirlik (Predictability)

Gerçek zamanlı sistemlerin en önemli özelliklerinden biri de tahmin edilebilirliktir. İdeal olarak, sistemin tasarım aşamasında tüm zaman kısıtlarını en fazla yüklü olduğu durumda bile yerine getirebileceği kesinlik kazanmalıdır. Sistem üzerinde yapılan istatistiksel davranış analizlerinin olayların bağımsız olduğunu farzetmesi, çalışma anında olayların birbirleriyle olan bağlantılarını göz önüne alamaması tahminleri zorlaştırmaktadır.

Dağıtık sistem tasarımcılarının çoğu sistemin, tıpkı kullanıcıların paylaşımlı bir dosya sistemine rastgele zamanlarda erişmesi gibi tahmin edilemez özellikte olduğunu düşünürler. Fakat gerçek zamanlı sistemlerde bu az rastlanılan bir durumdur. Genellikle bir olay E oluştuğu zaman, işlem X'in işletileceği, ardından da işlem Y'nin işletileceği ve Z'nin de diğerleriyle paralel çalıştırılacağı bilinmektedir. Hatta, sıklıkla bu işlemlerin çalışma süreleri de bilinebilmektedir. Bu nedenle bu tür sistemlerin modelleme yöntemleri deterministik olabilmektedir.

2.2.2.4 Hata toleransı

Çoğu gerçek zamanlı sistemler taşıtlar, hastaneler, güç birimleri gibi kritik cihazların kontrolünde kullanılmaktadır. Bu nedenle bu tür sistemlerde hata toleransı kritik bir önem taşımaktadır. Hata toleransı için, kullanılan kritik cihaz ya da işlemcilerin kopyalarının tutulması

gerekmektedir. Tüm kopyaların tutarlılığının sağlanması için kullanılan bir yöntem tüm kararları bir yöneticinin vermesi ve diğer birimlere iletmesi, diğer birimlerin de herhangi bir anda yönetimi ele almaya hazır durumda olmasıdır.

Kritik sistemlerde önemli olan bir başka durum da sistemin en kötü şartlara bile dayanıklı olmasıdır. Genellikle sisteme olan aşırı yüklenme, maksimum sayıda birimin kullanım dışı kalması sırasında oluşmaktadır. Bu nedenle hata toleranslı gerçek zamanlı sistemler aynı anda oluşan en fazla sayıda hata ve en yüksek yük durumuna karşı dayanıklı olmalıdır.

Bazı sistemlerin de belli bazı durumlarda işletimi durdurması gerekmektedir. Örnek olarak belli bir şiddetin üzerinde deprem olması durumunda elektrik santralı elektriği anında kesmek durumundadır. Bu gibi durumda işletimi durdurarak tehlikeyi önleyen sistemlere hata-güvenli (fail-safe) sistemler adı verilir.

2.2.3 Gerçek zamanlı iletişim

Gerçek zamanlı dağıtık sistemlerdeki iletişim kavramının diğer dağıtık sistemlerden farkı, yüksek performansın yanında tahmin edilebilirlik ve kararlılık özelliklerine sahip olması gereğidir.

Tahmin edilebilirlik dağıtık bir sistemde süreçler arası iletişimin süresinin tahmin edilebilir olmasıdır. Ethernet gibi yerel alan ağı protokolleri yapıları gereği iletişim süresi açısından bir üst sınır sağlayamadıkları için tahmin edilebilir bir yapıda değildirler. Paket iletim zamanı için en kötü durumu belirlemek mümkün değildir.

Buna karşılık token ring protokolünde bir paketin en kötü durumda iletim süresi tahmin edilebildiğinden bu prototol gerçek zamanlı sistemler için uygun bir iletişim altyapısı sağlayabilmektedir. Token ring ayrıca paketler üzerinde öncelik sınıflandırmasına imkan tanımaktadır. Bu sayede iletimi beklenen paketlerden yüksek önem taşıyanın daha önce iletilmesi mümkün olmaktadır.

Token ring protokolüne alternatif olarak TDMA (Time Division Multiple Access) protokolü gösterilebilir. Bu protokol belli sayıda saha (slot) içeren sabit uzunlukta paketlerin iletimini gerçekleştirir. Her bir saha bir işlemciye atanmıştır. İşlemciler sıraları geldiğinde gönderecekleri bilgiyi kendi sahalarına koyarlar. Bu sayede ethernet protokolünde görülen çakışmalar (collision) ve gecikmeler önlenmekte, aynı zamanda işlemcilere belli bir bant genişliği ayrılabilmektedir (Tanenbaum, 1996).

Geniş alan ağlarında çalışan gerçek zamanlı dağıtık sistemlerde iletişim bağlantı temelli olmaktadır. Bu tür sistemlerde öncelikle iki uç arasında gerçek zamanlı bir bağlantı kurulmaktadır. Bağlantı kurulurken ağ servis sağlayıcısı ile ağ kullanıcısı arasında servis kalitesi konusunda birtakım anlaşmalar yapılır ve garanti edilen maksimum gecikme, paket iletim zamanlarındaki değişimler, minimum bant genişliği gibi parametreler belirlenir. Geniş alan dağıtık sistemlerinin bir problemi, yüksek paket kayıplarıdır. Standart protokoller bu problemi gönderilen paketler için belli bir süre cevap bekledikten sonra tekrar iletimde bulunarak çözerler. Fakat bu yöntem, gecikme süresini sınırlayamadığından gerçek zamanlı sistemler için uygun değildir. Bunu yerine, göndericinin her zaman her paketi iki defa ya da daha fazla, tercihan başka bir bağlatı üzerinden göndermesi paket kayıplarını önleyici bir yöntem olabilir. Bu yöntem bant genişliğinin gereksiz

kullanıma yol açsa da belli ölçülerde mesaj iletim süresini sınırlayabilmekte ve paket kayıplarını önleyebilmektedir.

2.2.4 Gerçek zamanlı iş planlaması

Gerçek zamanlı sistemler genellikle her biri belli bir fonksiyonu gerçekleştiren ve belli bir işletim süresi olan küçük iş parçacıkları (task) toplulukları halinde programlanırlar. Sisteme gelen bir uyarıya verilecek cevap genellikle birden çok iş parçacığının belli bir sırada işletilmesini gerektirir. Bunun yanında, hangi işin hangi işlemcide çalıştırılacağına da karar verilmesi gerekmektedir.

Gerçek zamanlı iş planlaması algoritmaları şu parametrelere göre karakterize edilirler:

1. Katı gerçek zamanlı veya yumuşak gerçek zamanlı

2. Bir işi yarıda keserek (preemptive) veya kesmeden (nonpreemptive)

3. Dinamik veya statik

4. Merkezi veya dağıtık

Katı gerçek zamanlı algoritmalar tüm zaman kısıtlarını garanti edebilmelidir. Buna karşılık yumuşak gerçek zamanlı algoritmalar zaman kısıtı konusunda daha esnektirler.

Bir işi yarıda keserek yapılan iş planlaması, önceliği yüksek olan bir iş geldiğinde, çalışmakta daha düşük öncelikli işin işletiminin geçici

olarak durdurulması ve yüksek öncelikli işin işletimi tamamlandığında kaldığı durumdan devam etmesini gerektirir. İşi kesmeden yapılan iş planlaması ise yeni bir işi işletmek için çalışan işin sonuçlanmasını bekler.

Dinamik algoritmalar, iş planlamasını sistemin çalışması sırasında gerçekleştirirler. Bir olay tespit edildiğinde, dinamik ve bölünmüşlük mantığıyla bir algoritma bu olayla ilgili işin başlatılacağına ya da o anda çalışan işin devam edeceğine karar verir. Dinamik ve bölünmemiş özellikteki bir algoritma ise yeni işi iş kuyruğuna ekler ve işletilmekte olan işin tamamlanmasını bekler. Bu işin tamamlanmasının ardından işletilmeyi bekleyen işler arasından hangi işin başlatılacağına karar verir.

Buna karşılık statik algoritmalarda ise iş planlaması sistemin çalışmasından önce belirlenir. Sistem çalışırken bir olay oluştuğunda algoritma bir statik tabloya bakarak ne yapacağına karar verir.

Son olarak, iş planlaması sistem hakkındaki tüm bilgileri toplayan ve kararlar veren merkezi bir birim tarafından yapılabilir ya da her bir işlemci kendi kararlarını kendi verebilir. Dağıtık yöntemde işlemcilere yapılacak iş atamaları ile işletilecek işin seçimi birbirinden ayrı olarak gerçekleştirilir. Merkezi yöntemde ise bu iki iş aynı anda yapılabilir.

2.2.4.1 Dinamik iş planlaması

Dinamik algoritmalara bir örnek olarak rate monotonic algoritmasını gösterebiliriz (Tanenbaum, 1995). Bu algoritma aralarında bir sıralama ve yarış koşulu (race condition) olmayan işleri bölünmüş olarak işletir. İşlere çalışma sıklıklarına göre öncelik verilir, daha sık

çalışan işlerin önceliği daha yüksektir. Çalışma sırasında algoritma işletim için daima en yüksek öncelikli işi seçer. Bu algoritmanın optimal olduğu kanıtlanmıştır.

Dinamik algoritmalara ikinci örnek earliest deadline first algoritmasıdır (Audsley, N. and Burns, A., 1992). Bir olay tespit edildiğinde, ilgili iş bekleyen işler kuyruğuna eklenir. Bu kuyruk daima, işlerin zaman kısıtlarına göre sıralı bir şekilde tutulur. En erken zamanda çalışması gereken iş kuruğun en önünde bulunur. Periyodik işlerin zaman kısıtı bir sonraki çalışma zamanıdır. Algoritma çalıştırmak için kuyruğun en başındaki (zaman kısıtına en yakın olan) işi seçer. Bu algoritma da optimal bir sonuç üretmektedir.

Bölünmüş olarak çalışan üçüncü bir algoritma, öncelikle her işin boş olarak geçireceği zamanı hesaplar. Bu zamana laxity denilmektedir. Eğer bir iş 200 msn içinde bitirilmeliyse ve çalışmak için bir 150 msn'si daha varsa bu işin laxitiy'si 50 msn'dir.Algoritma laxity'si en düşük olan, yani en az bekleme zamanı olan işi seçer (Tanenbaum, 1995).

Bu algoritmaların hiçbirisi dağıtık sistemler için optimal sonuç üretmezler. Bunun yanısıra hiçbirisi yarış koşullarını göz önünde bulundurmazlar. Sonuç olarak pratikte sistemlerin bir çoğu işler hakkında yeterli bilgiye sahip olmaları halinde statik algoritmaları kullanırlar.

2.2.4.2 Statik iş planlaması

Statik iş planlaması sistem çalışmaya başlamadan önce gerçekleştirilir. Bu algoritma girdi olarak işlerin ve bu işlerin çalışma sürelerinin bir listesini alır. Algoritmanın amacı statik olarak işlerin

işlemcilere atanması ve her bir işlemcinin işleteceği işlerin sırasının belirlenmesidir. Teoride algoritma optimum çözümü bulmak için detaylı bir araştırma yapmalıdır. Ancak araştırma süresi iş sayısının karesiyle orantılı olduğundan bu tür bir arama genellikle yapılmaz, daha çok geçmişe dönük bilgilerden yararlanılır.

Statik algoritmalar, işlerin çalışma anındaki davranışlarının tamamiyle deterministik olmasını, çalışma anından önce bilinmesini gerektirir. İletişim ve işlemci hataları meydana gelmedikçe sistem gerçek zamanlı kısıtlarına her zaman ulaşacaktır. İşlemci ve iletişim hatalarının çözümü için daha önce bahsedilen yöntemler kullanılmalıdır.

2.2.4.3 Statik ve dinamik algoritmaların karşılaştırması

Dinamik ya da statik algoritmaların seçimi çok önemli ve sisteme etkilerinin fazla olduğu bir konudur. Zaman-tetiklemeli sistemler için statik algoritmalar uygunken, dinamik algoritmalar olay-tetiklemeli sistemlere daha çok uyarlar. Bunun yanında statik algoritmalar çok dikkatli bir şekilde planlanmalı ve parametreleri dikkatli belirlenmelidir. Dinamik algoritmalar ise, kararlar çalışma anında verildiği için bu tür bir detaylı çalışma gerektirmezler.

Dinamik algoritmalar statik olanlara göre sistem kaynaklarını daha etkin kullanırlar. Statik algoritma kullanan sistemlerde, sistem normalde ihtiyacı olandan daha fazla kapasiteye gereksinim duyabilmektedir. Ancak, özellikle katı gerçek zamanlı sistemlerde zaman kısıtlarının garantilenebilmesi için ihtiyaç fazlası kaynakların kullanılmasına göz yumulmaktadır.

Diğer taraftan, yeterli işlem kapasitesiyle statik bir sistem için optimum ya da ona yakın bir iş planlaması hesaplanabilir. Örnek olarak, bir nükleer reaktör kontrol sistemi için, uzun bir süre, belki bir kaç ay en iyi izlencelemeyi hesaplamak için zaman ve kaynak kullanmaya değecektir. Dinamik sistemler ise çalışma sırasında bu tür karmaşık ve uzun hesaplamalar yapma lüksüne sahip değildir. Bu nedenle bu sistemler de güvende olabilmek için ihtiyaç fazlası kaynak kullanabilirler ve bu sistemlerin belirlenen amaçları herzaman karşılama garantileri de yoktur.

Son olarak, iş planlaması konusunda göz önünde bulundurulması gereken daha pek çok etken ve kısıt olduğu, bazı sistemlerin çalışma anında daha gelişmiş planlama yapabilmek için hibrid sistemler kullandığı da söylenebilir.

2.3 Dağıtık Sistemlerde Dinamik Yük Dengeleme

2.3.1 Yük dağıtımı gereksinimi

Dağıtık sistemlerin güçlü işlem kapasitesinin avantajlarından etkin bir şekilde yararlanabilmek için iyi bir kaynak dağıtım modeline ihtiyaç bulunmaktadır. Dağıtık işlem programlayıcısı (distributed process scheduler), sistemin genel performansını en üst seviyede tutacak şekilde sistemin yükünü adil ve saydam bir şekilde işlemciler arasında dağıtır (Singhal and Shivaratri, 1994). Geniş alan ağlarındaki yüksek iletişim gecikmelerinden dolayı dağıtık işlem dağıtımı yerel ağlarda çalışan dağıtık sistemler için daha uygundur.

Dağıtık yük dağıtımına duyulan ihtiyacı açıklamak için şöyle bir senaryo düşünülebilir: Yerel bir ağ ile birbirine bağlı durumdaki bağımsız bilgisayarlardan oluşan dağıtık bir sistemde kullanıcılar kendi bilgisayarlarından sisteme birtakım işler (task) yüklerler. Böyle bir sistemde, sisteme farklı zaman dilimlerinde yüklenen farklı servis zamanlarına sahip işlemler sözkonusu olduğunda yüksek bir olasılıkla bazı bilgisayarlar ağır iş yükü altında ezilirken, bazıları boş durumda olacaklardır. Eğer bazı bilgisayarlara yüklenen iş yoğunluğu diğerlerinden fazlaysa, ya da bazı işlemciler diğerlerinden daha yavaş işlem hızına sahiplerse bu durumun daha da sıklıkla oluşacağı açıktır. Tüm bilgisayarların eşit işlem kapasitesine sahip olduğu ve uzun dönem iş yüklerinin birbirlerine eşit yoğunlukta olduğu bir sistemde ilk bakışta yük dağıtımının gereksiz olduğu düşünülse de, yapılan bazı çalışmalarda bu tür homojen dağıtık sistemlerde bile iş geliş ve servis zamanlarındaki değişimlerin yüksek olasılıkla, belli bir anda sistemde bir iş servis beklerken en az bir işlemcinin boş durumda olduğu ortaya çıkarılmıştır (Livny and Melman, 1982).

Sonuç olarak, dağıtık sistemlerde, sistem homojen bir yapıda olsa bile işlerin, yükü ağır işlemcilerden hafif yüklü işlemcilere transferi sistemin genel performasını arttırmaktadır. Bu noktada sistem performasının nasıl belirleneceği sorusu akla gelebilir. Sistem performans ölçütlerinden biri işlemlere verilen ortalama cevap verme süresi (average response time), yani bir işin sisteme yüklenip, sonucunun geri dönmesine kadar geçen süredir. Yük dağıtımının amacı sıklıkla bu süreyi düşürmektir. Akla gelebilecek bir başka soru ise bir işlemcideki yük durumunun nasıl belirleneceğidir. Bu konudaki ölçütlerden sıklıkla kullanılanı işlemcinin iş kuyruğunun boyudur (CPU queue length).

Sistem performansını arttırmak bir yük dağıtımı modelinin ana hedefi olmakla birlikte, karşılaması gereken birkaç önemli gereksinim daha bulunmaktadır (Singhal and Shivaratri, 1994):

Ölçeklenebilirlik: Yük dağıtım modeli, daha büyük dağıtık sistemlerde de çalışabilmelidir. Bu, minimum maliyetle hızlı dağıtım kararları alabilmeyi gerektirir.

Lokasyon bağımsızlığı: Transfer edilen işlem transfer edildiği işlemcide de orjinal işlemcide çalıştırılması halinde üretilen sonuçların aynını üretmelidir.

İşi bölme (preemption) : Bir işistasyonunun sahibi yokken, kaynaklarından yaralanmak için yük transferi yapılırken, o işistasyonunun sahibi bilgisayarını kullanmak istediğinde bu kişiye düşük bir performans sunulmamalıdır. Bunun için transfer edilmiş işler durdurulmalı ve iş istasyonu sahibinin işlerine öncelik verilmelidir.

Heterojenlik: Yük dağıtım modeli, farklı mimarilerde, değişik donanımlarda çalışan sistemleri ayırt edebilmelidir.

2.3.2 Yük dağıtımı ile ilgili karakteristikler

2.3.2.1 Yük değeri ölçümü

İşlemci kuyruk uzunluğu, işlem cevap verme süresini doğrudan etkileyen bir birim olduğu için yük değeri olarak iyi bir belirteçtir. Üstelik işlemci kuyruk uzunluğunu ölçmek kolay ve sisteme çok az yük getiren bir işlemdir. Bununla birlikte eğer yük transferi belli bir

gecikmeye sebep olabiliyorsa, yük miktarı için sadece kuyruk uzunluğunu kullanmak bir işlemcinin bir yük transferi henüz tamamlanmamışken, diğer istekleri de kabul etmesine neden olabilir. Bu durumda diğer kabul ettiği yük transferleri de tamamlandığında bu işlemci fazla iş yükü altında kalabilir. Bunu önlemenin bir yolu, işlemci yük transferini kabul ettikten sonra yük değerini manuel olarak arttırmaktır. Eğer transfer başarısızlıkla sonuçlanırsa, yani belli bir sürede gerçekleşmezse, yük değeri tekrar eski değerine alınır.

Yapılan çalışmalar işlemci kuyruk uzunluğunun işlemci kullanılabilirliği (utilization) üzerinde özellikle interaktif sistemlerde az bir etkisi olduğunu göstermiştir. Bu nedenle bazı sistemler yük değeri ölçümünde işlemci kullanılabilirliğini kullanmışlardır. Fakat kullanılabilirliği ölçmek için arkaplanda çalışan işlemler kullanılmakta, bu da sisteme ek yük getirmektedir.

2.3.2.2 Yük dağıtım algoritmaları sınıflandırması

Bir yük dağıtım algoritmasının en temel fonksiyonu yükü ağır makinelerden yükü hafif makinelere iş transferi yapmaktır. Yük dağıtım algoritmaları genel olarak üç gruba ayrılabilirler: Statik, dinamik ve adaptif. Dinamik yük dağıtım algoritmaları sistem durum bilgilerini (işlemcilerdeki yük miktarı) kullanarak yükün dağıtımı ile ilgili kararlar verirler. Statik algoritmalar ise sistem durum bilgisini kullanmaksızın, sistem hakkında önceden edinilmiş ve algoritmanın içine gömülmüş birtakım kurallara göre dağıtım yaparlar. Dinamik algoritmalar sistemin yük durumunda kısa süreli değişimler yaratarak statik algoritmalara göre daha iyi performans sağlayabilirler. Bunun yanında dinamik algoritmalar sistem durum bilgisi toplama, işleme ve analiz etme gibi işlemler

gerektirmesi dolayısı ile sisteme ek yük getirirler. Adaptif yük dağıtım algoritmaları, sistem durmuna göre algoritmanın bazı parametrelerini değiştirerek kendi aktivitelerini ayarlayan, dinamik algoritmaların bir başka çeşididir. Örnek vermek gerekirse dinamik bir yük dağıtım algoritması sistemin genel yük durumu yüksek olduğunda bile, sistem durum bilgisi toplamaya devam edecektir. Bu durumda sistemde yükü hafif bir işlemci bulma olsalığı çok düşük olduğundan hem gereksiz yere işlem yapacak, hem de sistemin zaten fazla olan yükünü biraz daha arttıracaktır. Adaptif bir algoritma ise böyle bir durumda gereksiz bilgi toplama işlemini durduracaktır.

2.3.2.3 Yük dengelemesi ve yük paylaşımı

Yük dağıtımı algoritmaları dağıtım prensiplerine göre, yük dengelemesi ve yük paylaşımı algoritmaları olarak farklı şekilde de sınıflandırılabilirler. Her iki çeşit algoritma da sistemin yük durumu açısından paylaşımsız durumunu (bazı işlemciler çok yüklüyken, bazılarının boş olduğu durum) azaltmaya çalışırlar. Fakat yük dengeleme algoritmaları bunun bir adım daha önüne geçerek tüm işlemcilerdeki yük seviyesini eşitlemeye çalışırlar. Bu amaçları sonucu yük dengeleme algoritmaları yük paylaşım algoritmalarına göre daha yüksek bir yük transferi yoğunluğuna sahiptirler ve bu da sisteme daha fazla yük getirerek bazı durumlarda sistem performansına olumsuz etkide bulunurlar.

Yük transferleri iletişim gecikmeleri ve iş durumlarını toplama sırasında yaşanan gecikmeler dolayısı ile anlık olarak gerçekleşmez. Yük transferindeki gecikmeler sistemin paylaşımsız durumunun süresini de uzatır. Bunu önlemenin bir yolu kısa bir süre içerisinde boş duruma

düşebilecek işlemcilere tahmini yük transferi yapmaktır. Bu tür tahmini transferler yük transfer yoğunluğunu arttıracağından bu tür yük paylaşımı algoritmaları yük dengeleme algoritmalarına benzer bir haldedirler. Bu nedenle yük dengeleme algoritmalarına yük paylaşımı algoritmalarının tahmini yük transferi yapan özel bir şeklidir diyebiliriz.

2.3.2.4 Bölerek (preemptive) veya bölmeden (nonpreemptive)

Bölünmüş yük transferleri, işletilmeye başlanmış işlerin çalışmasının kesilerek bir başka işlemciye transfer edilmesini ifade eder. Bu transfer, kesintiye uğratılan işin büyük ve karmaşık olabilecek tüm durum bilgilerinin toplanarak işle birlikte transfer edilmesini gerektirdiğinden oldukça maliyetli ve zor bir işlemdir. Buna karşılık bölünmemiş yük transferleri yalnızca henüz işletilmeye başlamamış, dolayısı ile durum bilgisi bulunmayan işleri transfer ederler. Her iki durumda da işle ilgisi çevre bilgilerinin (çalışma klasörü, işin miras aldığı erişim hakları, vb.) transfer edilmesi gerekmektedir. Bölünmemiş yük transferleri aynı zamanda yük yerleştirmesi şeklinde de adlandırılırlar.

2.3.2.5 Dengelilik (stability)

Bir sistemin uzun dönem iş geliş oranı, sistemin iş yapabilme kapasitesinden fazlaysa, işlemci kuyrukları sürekli bir şekilde büyümeye başlar. Böyle bir sistem dengesiz (unstable) olarak ifade edilir. Sistem durum bilgisi toplamak için sürekli mesaj iletişimi yapan bir yük dağıtım algoritması düşünelim. Sisteme yüklenen iş miktarı ile bu algoritma tarafından sisteme getirilen yük miktarı birlikte sistemin servis kapasitesini aşabilir ve sistemi dengesiz bir hale sürükleyebilir.

Bunun yanısıra, bir algoritma dengeli bir yapıda olsa bile sistemi yük dağıtımının hiç yapılmadığı durumdan daha kötü bir performansa sürükleyebilir. Bu nedenle algoritmaları daha kısıtlayıcı bir kriter ile incelemek gerekir. Bu amaçla etkinlik (effectiveness) kavramını kriter olarak kullanabiliriz. Bir yük dengeleme algoritması, belli koşullar altında eğer sistemi yük dengelemesinin yapılmadığı durumdan daha iyi bir performansa getirebiliyorsa etkin olarak ifade edilir.

Algoritmik perspektiften bakacak olursak, eğer bir algoritma sürekli ve belirsiz bir şekilde yararsız işlemler gerçekleştiriyorsa bu algoritma dengesiz olarak ifade edilir. Örnek olarak, yükü fazla olan bir sistemde, bir transfer sonucu fazla yüklü duruma gelen bir işlemci, başka bir yük transferine sebep olacak ve bu işlem işlemciden işlemciye belirsiz bir süre devam edecek, iş sürekli bir işlemciden başka bir işlemciye aktarılacak ve hiç servis alamayacaktır.

2.3.3 Yük dağıtımı algoritması parçaları

Genel olarak bir yük dağıtım algoritması dört parçadan oluşur: Hangi işlemcinin yük transferi için uygun olduğunu belirleyen *transfer politikası*, hangi işin transfer edileceğini belirleyen *seçim politikası*, yük transferinin hangi işlemciye yapılacağını belirleyen *lokasyon politikası* ve sistemin yük durumu bilgisini toplayan *bilgi politikası* (Singhal and Shivaratri, 1994).

2.3.3.1 Transfer politikası

Transfer politikaları genellikle eşik (threshold) yöntemine dayanır. Eşik değeri işlemci yük değeri cinsinden bir değerdir. Yeni bir iş

geldiğinde, eğer işlemci yükü eşik değerini aşıyorsa işlemci gönderici, yani yük transferini yapacak işlemci olarak belirlenir. Eğer yük değeri eşik değerinin altına düşmüşse işlemci alıcı, yani yük transferinin yapılacağı işlemci olarak belirlenir.

Bir başka yöntem ise, yük transferini bilgi politikası tarafından işlemciler arası yük durumlarında bir dengesizlik tespit edildiği anda tetikler.

2.3.3.2 Seçim politikası

Seçim politikası, transfer politikasının bir işlemciyi gönderici olarak belirlemesinin ardından transfer edilecek işi seçer. En basit yaklaşım, işlemcinin gönderici olarak belirlenmesine yol açan, yani işlemci yükünün eşik değerini aşmasına sebep olan yeni gelen işi seçmektir. Bu tür işlerin transfer edilmesi de kolaydır, çünkü bu işler henüz işletilmeye başlanmadığı için bölünmemiş(nonpreemtive)tir.

Transfer için iş seçiminde bazı kriterler bulunmaktadır. Bunlardan bir tanesi transfer işleminin sonucunda sistemin ortalama cevap verme süresinin düşmesidir. Bir başkası, işin transfer edilmesi sonucu gerçekleşen çalışma süresinin, işlemin transfer edilmeden gerçekleşen çalışma süresinden büyük olmamasıdır. İş seçiminde, transfer işleminin sisteme getireceği yükün minimal düzeyde tutulması, transfer edilecek işin lokasyon bağımlı sistem çağırılarının minimal düzeyde olması gibi başka faktörler de göz önüne alınmalıdır.

2.3.3.3 Lokasyon politikası

Lokasyon politikası yük paylaşımı için uygun işlemciler (alıcı ve gönderici) bulmakla yükümlüdür. Kullanılan yöntemlerden biri sorgulamadır (polling). Bu yöntemde bir işlemci bir başka işlemciyi yük transferi için uygun olup olmadığı konusunda sorgular. Sorgulama rastgele bir işlemci seçilerek, ya da bilgi politikasınca toplanan bilgiler doğrultusunda belirlenen bir işlemciye yapılabilir. Bir başka yöntem ise yayındır (boradcast).

2.3.3.4 Bilgi politikası

Bilgi politikası, sistemin işlemcilerinden ne zaman, nereden ve hangi bilginin toplanması gerektiğinin belirlenmesinden sorumludur. Bunu için üç yöntem bulunmaktadır:

İsteğe-bağlı (demand driven): Bu yöntemde işlemci, gönderici ya da alıcı duruma geçtiği anda (transfer ve seçim politikaları sonucunda), yani yük transferi için uygun hale geçtiğinde diğer işlemcilerden bilgi toplar. Bu politika gönderici-başlatımlı, alıcı-başlatımlı, ya da simetrik-başlatımlı olabilmektedir. Gönderici-başlatımlı yöntemde gönderici yük transferi için uygun bir alıcı arar, alıcı-başlatımlı yöntemde alıcı yük transferi yapabileceği uygun bir gönderici arar, simterik-başlatımlı yöntem diğer iki yöntemin kombinasyonudur.

Periyodik: Bu yöntemde işlemciler periyodik olarak birbirlerinden bilgi toplarlar. Toplanan bilgi doğrultusunda transfer politikası bir işlemcide yük transferine karar verir. Bu yöntem, kendini sistem durumuna göre adapte edemez. Sistemin yük seviyesinin yüksek olduğu

durumlarda işlemcilerin hemen hepsi yüklü olduğundan yük transferi yarar getirmez. Bu durumda bilgi toplamak da gereksizdir. Ancak periyodik yöntemde, bu durumda bile bilgi toplama işlemi sürdürüldüğü için sisteme gereksiz yere ek yük getirilmektedir.

Durum-değişimli(state-change-driven): Bu yöntemde, işlemciler durumları belli derecede değişime uğradığında bilgilerini diğer işlemcilere yayarlar. Bu yöntemin isteğe bağlı yöntemden farkı işlemcilerin bilgi toplamak yerine durumlarını diğer işlemcilere yaymasıdır. İşlemciler durumlarını ya merkezi bir işlemciye iletirler ya da tüm diğer işlemcilere yayarlar.

2.3.4 Yük dağıtımı algoritmaları

2.3.4.1 Gönderici-başlatımlı (sender-initiated) algoritmalar

Bu tip algoritmalarda yükü eşik değerini aşan işlemci (gönderici) yükü hafif bir işlemci (alıcı) arar (Chang and Livny, 1986).

Transfer Politikası: Bu algoritmalar eşik tabanlı işlemci kuyruk uzunluğu kullanırlar. Transfer politikası yeni bir iş geldiğinde başlar. İşlemci kendini eğer yükü eşik değerinin üzerindeyse gönderici, altındaysa alıcı olarak belirler.

Seçim Politikası: Bu tür algoritmalar sadece yeni gelen işleri transfer için seçerler.

Lokasyon Politikası: Lokasyon politikasına göre üç çeşit gönderici-başlatımlı algoritma bulunmaktadır:

Rastgele: Bu yöntemde diğer işlemcilerin durum bilgileri kullanılmaz. Yük transferi rastgele seçilen bir işlemciye yapılır. Bu noktada oluşabilecek bir problem, rastgele seçilen işlemcinin de yüklü durumda bulunmasıdır. Böyle bir durumda bu işlemci de yükü başka bir işlemciye göndermeye çalışacaktır ve bu da gereksiz yük transferlerine sebep olacaktır. Bunu önlemenin bir yolu bir iş için transfer edilebilme sayısını sınırlamaktır. Bu algoritma yük dağıtımı yapmayan sistemlere göre oldukça iyi sonuçlar vermektedir.

Eşik (treshold): Rastgele yöntemde meydana gelen gereksiz yük transferi problemi, rastgele seçilen bir işlemcinin alıcı olup olmadığının sorgulanmasıyla önlenmektedir. Eğer seçilen işlemci alıcıysa yük transferi gerçekleştirilmekte, değilse başka bir işlemci seçilerek sorgulanmaktadır. Sorgulanan işlemci sayısı belli bir eşik değeriyle sınırlandırılmakta, eşik değeri aşılıncaya kadar uygun bir işlemci bulunamazsa iş orijinal işlemcide çalıştırılmaktadır. Gereksiz transferleri önleyerek eşik yöntemi rastgele yönteme göre oldukça iyi sonuçlar vermektedir.

En-kısa: Bu algoritmanın diğerlerinden farkı, en iyi alıcıyı belirlemeye çalışmasıdır. Gönderici, belli sayıda işlemciyi sorgulayarak kuyruk uzunluklarını elde eder ve kuyruk uzunluğu eşik değerinin altında (alıcı durumda) olan işlemcilerden en kısa kuyruk uzunluğuna sahip olan işlemciye yük transferini gerçekleştirir. Yapılan testlerde bu algoritmanın eşik algoritmasına göre çok az bir performans artışı sağladığı görülmüştür. Bu da ayrıntılı durum bilgisi elde etmenin gereksiz olabileceğini göstermektedir.

Bilgi Politikası: Eşik ve en-kısa lokasyon politikalarında bilgi sorgulama işlemi, transfer politikası bir işlemciyi gönderici olarak

belirlediğinde gerçekleştirilmektedir. Yani bilgi politikası olarak isteğe-bağlı yöntem kullanılır.

Bu tür algoritmalar sistemi, sistem yükünün fazla olduğu durumlarda dengesiz bir hale sürükleyebilirler. Sistem yüklü bir durumdayken işlemcilerin çoğu gönderici durumunda bulunduğu için bu işlemcilerin alıcı bulmak için yaptıkları sorgulamalar başarısız olacaktır. Yük arttıkça sorgulamalar da artacak ve bir noktada sistemin mevcut işlem kapasitesi büyük ölçüde sorgulama ve bu sorgulamalara cevap vermek için kullanılacaktır. Sisteme gelen yük miktarı ile yük dağıtım işleminden kaynaklanan yük miktarı sistemin kapasitesini aştığında dengesizlik oluşacaktır. Bu nedenle bu tür algoritmalar yüksek yük durumunda etkin değildirler ve sistemi dengesizliğe sürükleyebilirler.

2.3.4.2 Alıcı-başlatımlı (receiver-initiated) algoritmalar

Bu tür algoritmalarda yük dağıtım işlemi yüklü bir işlemciden yük isteyen alıcı (yük durumu hafif olan işlemci) tarafından başlatılır (Chang and Livny, 1986).

Transfer Politikası: Bu algoritmalar eşik tabanlı işlemci kuyruk uzunluğu kullanırlar. Transfer politikası, çalışmakta olan bir iş tamamlandığında tetiklenir. İşlemci eğer yükü eşik değerinin altındaysa alıcı durumuna geçer. Yükü eşik değerinin üzerinde olan işlemci ise gönderici durumundadır.

Seçim Politikası: Bu algoritma bölüm 2.3.3.2'de bahsedilen yaklaşımlardan herhangi birine göre seçim politikasını belirler.

Lokasyon Politikası: Bu algoritmada, rastgele seçilen bir işlemcinin yükünün transfer işleminin sonunda eşik değerinin altına düşüp düşmeyeceği sorgulanır. Eğer düşmeyecekse sorgulanan işlemci yük transferi yapar. Diğer durumda ise rastgele bir başka işlemci seçilerek aynı sorgulama yapılır. Bu işlem belli sayıda tekrarlanır. Eğer bu sorgulamalardan sonra da yük transferi yapılamamışsa, alıcı başka bir iş tamamlanıncaya kadar bekler ve tekrar sorgulamaya başlar. Algoritmanın başka bir şekline göre ise alıcı belli bir süre bekler ve bu süre sonunda hala alıcı durumundaysa sorgulama işlem yeniden başlatılır. Eğer ikinci yöntem kullanılmazsa, alıcının belli bir iş tamamlanıncaya kadar bekleme süresindeki boş işlem kapasitesi kaybedilmiş olacaktır.

Bilgi Politikası: İsteğe-bağlı bilgi politikası kullanılır, çünkü herhangi bir işlemci alıcı konumuna geçtiğinde bilgi toplanır.

Bu tür algoritmalar sistemi dengesizliğe sürüklemezler. Çünkü sistemin yüklü olduğu durumlarda sistemde çok sayıda gönderici bulunmaktadır ve alıcıların yaptığı sorgulamalar yüksek bir başarı oranına sahip olacaktır. Bu da sorgulama mekanizmasının etkin bir şekilde kullanılmasını sağlar. Sistemin az yüklü olduğu durumlarda ise fazla sayıda olan alıcılar az sayıda bulunan göndericileri bulmaya çalışırken çok sayıda sorgulama yapmak durumunda kalacaklardır. Ancak sistemin işlem kapasitesinin büyük bölümü zaten boş olduğundan başarısız sorgulamalar sistemi dengesiz bir yapıya sürüklemeyecktir.

Bunun yanında, alıcı-başlatımlı algoritmalar, büyük ölçüde işletimine başlatılmış işlerin transfer edilmesine sebep olur. Çünkü bu algoritmalarda alıcı göndericiden yük transferi istediği anda göndericide henüz başlatılmamış iş yoksa, çalışmakta olan bir işin transferi

gerçekleştirilecektir. Bu nedenle bu algoritmalar bölünmüş yapıda (preemptive) algoritmalardır ve yük transferleri zor ve karmaşıktır.

2.3.4.3 Simetrik algoritmalar

Bu tür algoritmalarda hem gönderici hem de alıcı konumundaki işlemciler transfer sürecini başlatabilirler. Bu nedenle bu algoritma önceki iki algoritmanın da avantajlarından yararlanır. Sistemin yük durumunun düşük olduğu durumlarda gönderici-başlatımlı taraf yükü hafif işlemci bulmakta başarılı olurken, yüksek sistem yükünde alıcı-başlatımlı taraf gönderici bulmakta başarılı olmaktadır. Fakat aynı zamanda bu yöntem, bu iki algoritmanın dezavantajlarına da sahiptir. Algoritma, gönderici-başlatımlı algoritmalarda olduğu gibi yüksek yük durumunda sistemi dengesizliğe sürükleyebilirken, alıcı-başlatımlı algoritmalardaki işin yarıda kesilip transfer edilmesinden kaynaklanan transfer işlemi zorluğuna da sahiptir (Singhal and Shivaratri, 1994).

Bu tür algoritmalara örnek olarak, above-average algoritmasını (Krueger and Finkel, 1984) verebiliriz. Bu algoritma, her bir işlemcideki yük seviyesini belli bir kabul edilebilir aralıkta tutmaya çalışır.

Transfer Politikası: Transfer politikası eşik yöntemini kullanır. Bunun için iki adet adapte edilebilir eşik değeri bulunmaktadır. Bu eşikler bir işlemcinin sistemin yük durumu hakkında yaptığı tahmine göre belirlenir. Örnek olarak, eğer işlemci tüm sistemin yükünü 2 olarak tahmin ettiyse küçük eşik 1 büyük eşik ise 3 olacaktır. Yükü küçük eşikten düşük olan işlemci alıcı, büyük eşikten büyük olan işlemci ise gönderici olarak belirlenir. Yük durumu iki eşik arasında olan işlemciler alıcı ya da gönderici olarak belirlenmezler.

Seçim Politikası: Bu algoritma bölüm 2.3.3.2'de bahsedilen yaklaşımlardan herhangi birine göre seçim politikasını belirler.

Lokasyon Politikası: Lokasyon politikası iki parçadan oluşur:

Gönderici-başlatımlı parça:

- Gönderici, *TooHigh* mesajı yayınlar, *TooHigh* zaman-aşımı (timeout) alarmını başlatır ve zaman aşımı oluşuncaya kadar *Accept* mesajı bekler.

- *TooHigh* mesajı alan alıcı *TooLow* zaman-aşımını iptal eder, *TooHigh* mesajını gönderen işlemciye *Accept* mesajı gönderir, yük değerini arttırır ve *AwaitingTask* zaman-aşımı alarmını başlatır. Eğer transfer işlemi gerçekleşmeden *AwaitingTask* zaman-aşımı gerçekleşirse yük değerini tekrar eski haline alır.

- *Accept* mesajı alan işlemci eğer hala gönderici durumunda ise transfer işlemi için en uygun işi seçer ve mesajı gönderen işlemciye transfer eder.

- *TooHigh* zaman-aşımı dolduğunda gönderici eğer hiç *Accept* mesajı almadıysa, sistemin yük değeri tahmininin çok düşük olduğunu anlar ve bu durumu düzeltmek, yani diğer işlemcilerin yük tahminlerini yükseltmeleri için *ChangeAverage* mesajı yayınlar.

Alıcı-başlatımlı parça:

- Alıcı *TooLow* mesajı yayınlar, *TooLow* zaman-aşımı alarmını başlatır ve *TooHigh* mesajı bekler.

- Eğer alıcı *TooHigh* mesajı alırsa, gönderici-başlatımlı parçada bahsedilen uzlaşma sürecini gerçekleştirir.

- Eğer *TooHigh* mesajı almadan *TooLow* zaman-aşımı oluşursa, alıcı diğer işlemcilerdeki ortalama yük tahmini değerini düşürmek için *ChangeAverage* mesajı yayınlar.

Bilgi Politikası: İsteğe-bağlı bilgi toplama yöntemini kullanır. Burada altı çizilecek bir nokta, ortalama sistem yük değerinin her bir işlemcide belirlenmesiyle, küçük bir miktar performans kaybı yaratılıp, mesaj karmaşıklığının azaltılmasının sağlanmasıdır. Bir başka nokta ise, kabul edilebilir aralık sayesinde sistemin yük dağıtım işleminin iletişim ağının yoğunluğuna göre adapte olabilmesidir. İletişim ağının yüklü/yüksüz durumuna göre (mesaj iletim sürelerine bakılarak belirlenir) kabul edilebilir aralık büyütülüp/küçültülerek algoritmanın kendini adapte etmesi sağlanır.

2.3.4.4 Adaptif algoritmalar

Bu tür algoritmalara örnek olarak dengeli simetrik bir algoritma örnek verilebilir (Singhal and Shivaratri, 1994). Diğer algoritmalardaki, yük paylaşımından kaynaklanan dengesizliğin sebebi göndericinin yaptığı başarısız sorgulamalardır. Bu algoritma, sorgulamalardan elde edilen bilgilerden yararlanarak (diğer algoritmalar bu bilgileri kullanmazlar) işlemcileri alıcı, gönderici ve normal olarak sınıflandırır. Bu üç çeşit sınıflandırma bilgileri her bir işlemcideki üç ayrı listede tutulur (gönderici listesi, alıcı listesi ve normal işlemci listesi). Bu listeler, üzerinde güncelleme ve arama işlemleri çok az işlemci gücü gerektirecek şekilde özel olarak tasarlanmışlardır. Başlangıçta tüm

işlemciler alıcı olarak kabul edilirler ve alıcı listesine yerleştirilirler, diğer listeler boştur.

Transfer Politikası: Transfer politikası, işlemci kuyruk uzunluğunu kullanan eşik yöntemidir. Transfer politikası yeni bir iş geldiğinde ya da işletilen bir iş tamamlandığında aktive olur. Transfer politikası iki ayrı eşik değeri kullanır. Eğer bir işlemcinin yük değeri küçük eşikten düşükse alıcı, büyük eşikten yüksekse gönderici, iki değer arasındaysa normal olarak belirlenir.

Seçim Politikası: Bu algoritmanın gönderici-başlatımlı parçası sadece yeni gelen işleri seçer. Alıcı-başlatımlı parçası ise bölüm 2.3.3.2'de bahsedilen yaklaşımlardan herhangi birine göre seçim politikasını belirler.

Lokasyon Politikası:

Gönderici-başlatımlı parça:

Gönderici başlatımlı parça, bir işlemci gönderici durumuna geçince tetiklenir. Gönderici, alıcı listesinin en başındaki işlemciyi sorgular. Sorgulanan işlemci, sorgulayan işlemciyi bulunduğu listeden çıkararak gönderici listesinin başına ekler ve o anki yük durumuna göre kendi durumunu göndericiye bildirir. Gönderici, eğer sorguladığı işlemci alıcı ise yeni gelen işi bu işlemciye transfer eder. Eğer sorgulanan işlemci alıcı değilse, gönderici bu işlemciyi alıcı listesinden çıkarır ve bildirdiği yük durumuna göre diğer iki listeden birinin başına ekler. Ardından alıcı listesinin en başındaki işlemciyi sorgular. Sorgulama süreci, gönderici bir alıcı buluncaya, belli bir sorgulama sayısı aşılıncaya ya da alıcı listesi

boşalıncaya kadar devam eder. Eğer sorgulama süreci başarısızlıkla sonuçlanırsa iş göndericide başlatılır.

Alıcı-başlatımlı parça:

Alıcı-başlatımlı parçanın amacı yükünü alabileceği uygun bir gönderici bulmaktır. İşlemciler şu sırada sorgulanır: Gönderici listesi baştan sona doğru (en güncel bilgiyi kulanarak), normal işlemci listesi sondan başa doğru (en eski bilgiyi kullanarak), alıcı listesi sondan başa doğru.

Alıcı-başlatımlı parça bir işlemci alıcı konuma geçtiğinde tetiklenir. Alıcı, yukarıda belirtilen şekilde seçilen işlemciyi sorgular. Sorgulanan işlemci eğer göndericiyse üzerindeki bir işi alıcıya transfer eder ve transfer sonrası yük durumunu alıcıya bildirir. Eğer sorgulanan işlemci gönderici değilse, sorgulayan işlemciyi bulunduğu listeden çıkarır ve alıcı listesinin başına ekler ve alıcıya o anki yük durumunu bildirir. Alıcı sorgulanan işlemciden cevap alınca, bu işlemciyi bulunduğu listeden çıkarır ve uygun listenin başına ekler.

Sorgulama işlemi bir gönderici bulunduğunda, alıcının durumu değiştiğinde (gönderici ya da normal komuna geçtiğinde) veya belli sayıda sorgulama yapıldığında sona erer.

Bilgi Politikası: Bilgi politikası, bir işlemci gönderici ya da alıcı konuma geçtiği anda başladığı için isteğe-bağlı bilgi toplama yöntemidir.

Sistem yükünün fazla olduğu durumlarda, gönderici-başlatımlı kısım başarısız sorgulamalarda bulunacaklardır. Bu sorgulamalar sonucunda, sorgulanan işlemciler alıcı listesinden çıkarılacaklardır ve en

sonunda alıcı listesi boşalacaktır. Alıcı-başlatımlı kısım başarısız sorgulamalarda bulunmadıkça alıcı listesi boş kalacaktır. Alıcı listesi boş kaldıkça gönderici-başlatımlı kısımın çalışması durmuş olacaktır, çünkü gönderici yalnızca alıcı listesindeki işlemcileri sorgulamaktadır. Sonuç olarak yüksek sistem yükünde göndericilerin gereksiz sorgulamalar yaparak sistemi dengesizliğe sürüklemesi önlenmiş olacaktır. Bu durumlarda sadece alıcı-başlatımlı parça çalışır durumda olacaktır.

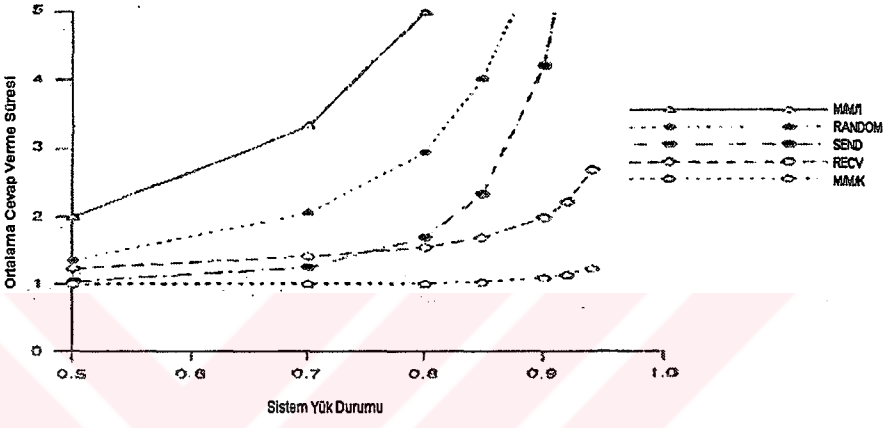
Sistem yükünün az olduğu durumlarda alıcı-başlatımlı kısım başarısız sorgulamalarda bulunsa da, bu sorgulamalar sistem yükü düşük olduğundan performans kaybına neden olmayacak, bunun yanısıra alıcı listelerinin güncel tutulmasını sağlayacaktır. Alıcı listelerinin güncel olması da gönderici-başlatımlı sorgulamaların başarı oranını yükseltecektir.

Sonuç olarak yüksek yük durumunda alıcı-başlatımlı, düşük yük durumunda gönderici-başlatımlı, ortalama yük durumlarında da simetrik politikalar kullanılarak sistemin geniş bir yük seviyesi aralığında performansı optimum düzeyde tutulacak ve dengeli yapısı korunacaktır.

2.3.5 Performans karşılaştırması

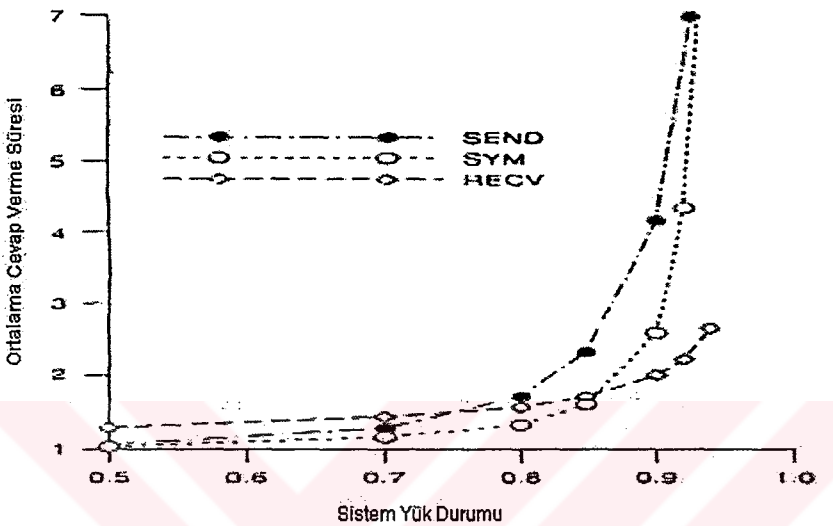
Şekil 2.1'de de görüldüğü gibi rastgele lokasyon politikası kullanan gönderici-başlatımlı algoritmalar (şekilde RANDOM olarak belirtilmiştir), hiç yük dağıtımı yapmayan bir sistemle (şekilde M/M/1 olarak belirtilmiştir) karşılaştırıldığında uyguladığı bu basit yöntemle bile oldukça iyi performans artışı sağlamaktadır (Livny and Melman 1984). Yine aynı şekilde eşik yöntemi uygulayan gönderici-başlatımlı (SEND) ve alıcı-başlatımlı (RECV) algoritmaların da dikkate değer bir

performans artışı sergiledikleri görülmektedir. Şekilde, M/M/K ile belirtilen, yük dağıtımından kaynaklanan hiç bir performans kaybı olmayan ideal bir algoritmayı ifade etmekte ve yük dağıtımı için olabilecek maksimum performans sınırını göstermektedir.



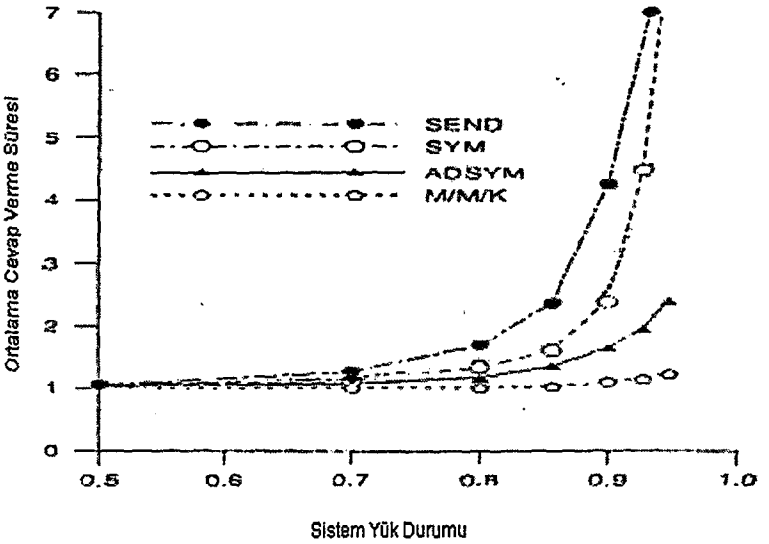
Şekil 2.1 Sistem yüküne göre ortalama cevap verme süreleri

Şekil 2.2'de simetrik algoritmanın (SYM) diğer iki yöntemle karşılaştırması görülmektedir. Elde edilen sonuçlar bu algoritmanın diğer iki algoritmanın avantajlarını kullanarak daha iyi performans sağladığını ortaya çıkarmaktadır. Bu algoritma, her tür yük seviyesinde gönderici-başlatımlı algoritmadan, düşük yük seviyesinde de alıcı-başlatımlı algoritmadan daha iyi sonuç vermektedir. Ancak yüksek yük durumunda bu algoritma gönderici-başlatımlı kısmın başarısız sorgulamalarından kaynaklanan performans kayıplarıyla karşı karşıya kalmakta ve sistemi dengesiz bir yapıya sürüklemektedir.



Şekil 2.2 Sistem yüküne göre ortalama cevap verme süreleri (SYM karşılaştırması)

Dengeli simetrik algoritmanın ideal bir yük dağıtım algoritmasının performansına yakın bir sonuç verdiği şekil 2.3'te (şekilde ADSYM olarak belirtilmiştir) görülmektedir. Bu algoritmanın performansı düşük yük seviyesinde gönderici-başlatımlı algoritmaya benzer bir seviye almakta, yüksek yük seviyelerinde de kendini yük durumuna göre adapte etme özelliği olmayan diğer algoritmalardan belirgin derecede daha iyi sonuç vermektedir. Bu performansın kaynağı sorgulamalardan elde edilen bilginin kullanılmasıdır. Adaptif algoritma ayrıca sistemi dengesizliğe sürüklememektedir.



Şekil 2.3 Sistem yüküne göre ortalama cevap verme süreleri (ADSYM karşılaştırması)

3 GERÇEK ZAMANLI DAĞITIK SİSTEM MODELİ

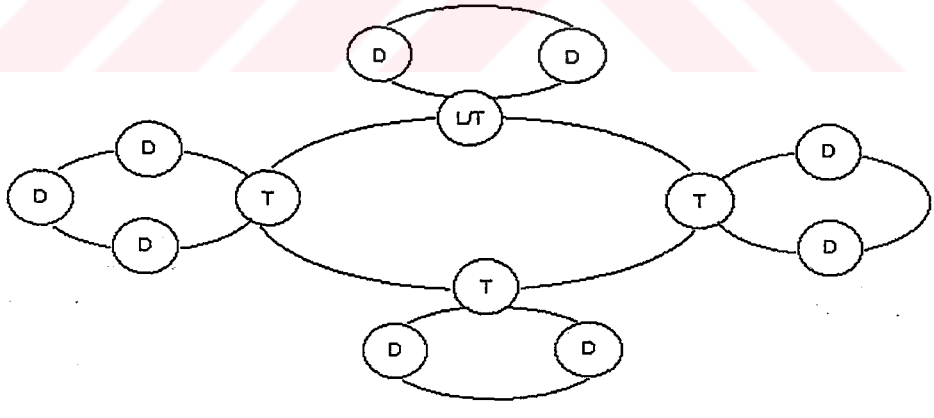
Bu bölümde, uygulaması gerçekleştirilen, Tunalı, Erciyeş ve Soysert (1998) tarafından gerçek zamanlı dağıtık sistemler için geliştirilen hata toleranslı hiyerarşik halka protokolünün tasarım özelliklerinden bahsedilecektir.

3.1 Sistemin Yapısı

Daha önce de belirtildiği gibi, gerçek zamanlı dağıtık bir sistem, sınırları belirli zaman aralıklarında mesaj iletimini garanti eden hata toleranslı bir iletişim ağı, sistem kaynaklarını etkin ve zaman bağımlı olarak yöneten gerçek zamanlı dağıtık bir işletim sistemi ve bu platformu gerçek zamanlı dağıtık uygulamaların kullanımına sunabilecek bir arabirime ihtiyaç duymaktadır. Geliştirilen sistem modelinin amacı da, hiyerarşik olarak çalışan, ölçeklenebilir, hata toleranslı iletişim altyapısına sahip senkron bir halka protokolü ile, dağıtık gerçek zamanlı sistemler için bir platform oluşturmaktır.

Dağıtık sistem modeli, "düğüm" (node) olarak adlandırılan işlem (process) ya da işlemci birimlerinin yer aldığı kümelerden oluşmaktadır. Aynı kümede yer alan birimlerin birbirlerine bir yerel alan ağıyla bağlı ve fiziksel olarak yakın konumlanmış oldukları düşünülmüştür. Her bir işlemci kümesinin bir adet "temsilci" (representative) olarak adlandırılan küme temsilcisi bulunmaktadır. Bu küme temsilcileri üyesi bulundukları bir üst kümenin temsilcisi ile iletişim kurmaktadır. En üstte yer alan kümenin temsilcisi ise "lider" (leader) olarak adlandırılan tüm sistemin yöneticisidir. Sistem yöneticisi, en üst kümede çalışan halka protokolü ile küme temsilcilerinden bilgi toplar. Aynı şekilde bir alt seviyedeki

kümelerde de her bir küme temsilcisi kendi küme üyelerinden bilgi toplarlar. En alt seviyedeki kümelerin temsilcileri de kendi kümlerindeki düğümlerden aynı şekilde çalışan halka protokolü ile bilgi toplarlar. Bu sayede, en alttaki işlemci birimlerinden toplanan bilgiler en üstteki sistem yöneticisine kadar ulaştırılmaktadır. Aynı şekilde sistem yöneticisinin verdiği kararlar da küme temsilcileri üzerinden en alttaki düğümlere kadar ulaştırılmaktadır. Her bir seviyede ve her bir kümede aynı halka protokolü çalışmaktadır. Şekil 3.1'de iki seviyeli bir sistemdeki halka protokollerinin yapısı görülmektedir. Bu yapıda, üst seviye liderin idare ettiği küme temsilcilerinden, alt seviye ise küme temsilcilerinin yönettiği ve düğümlerin yer aldığı kümelerden oluşmaktadır. Sistemin tasarım ve işleyişi bundan böyle bu iki seviyeli sistem modeli göz önüne alınarak açıklanacaktır. Ortaya çıkarılan bu iki seviyeli sistem modeli kullanılarak üç ya da daha fazla seviyeden oluşan sistemler kolaylıkla tasarlanabilecektir. Bu da sistemin ölçeklenebilir ve esnek bir özellikte tasarlanmış olduğunu göstermektedir.



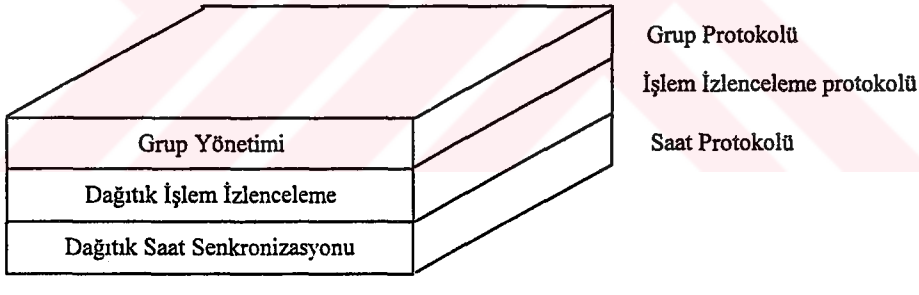
Şekil 3.1. İki seviyeli bir sistemin genel görünümü (D:Düğüm, T:Temsilci, L:Lider)

Bu modelde, sistem yapısının çeşitli katmanlardan oluştuğu düşünülmüştür. Modelin en alt katmanında dağıtık saat senkronizasyonu yer almaktadır. Sistem senkron bir yapıda tasarlandığından, sistemi oluşturan tüm birimlerin ortak bir genel saat değerine sahip olmaları gerekmektedir. Tasarlanan küme tabanlı modelin bu katmandaki uyarlamasında, dağıtık sistemin işlemci birimleri, sistemin düğümleri olarak kabul edilmişlerdir. Bu yapıda, lider sistemin düğümlerinden saat değerlerini istemektedir. Temsilciler tarafından her bir kümeye ait toplanan saat değerleri lidere ulaştığında, lider yeni bir saat değeri belirlemekte ve bunu küme temsilcileri aracılığı ile düğümlere bildirmektedir. Bu işlem, her bir periyodda tekrarlanmakta ve böylece sistemin saat senkronizasyonu sağlanmaktadır. Uygulaması gerçekleştirilen sistem üzerinde bu saat senkronizasyonu mekanizmasının bir simulasyonu da gerçekleştirilmiştir.

Katmanlı modelde, saat senkronizasyonunun üzerinde dağıtık işlem programlayıcısı (distributed scheduler) bulunmaktadır. Dağıtık sistem programlayıcısı, dağıtık sistemin kaynaklarının en etkin olarak kullanılabilmesi için gerçek zamanlı, adil ve saydam bir şekilde yönetiminden sorumludur. Bu katmanda da, tıpkı saat senkronizasyonunda olduğu gibi küme tabanlı modelin bir uyarlaması yer almaktadır. Bu uyarlamada da sistemin düğümleri işlemci birimlerinden oluşmaktadır ve saat senkronizasyonunda yer alan kümeler aynen yer almaktadır. Fakat bu katmandaki lider ve temsilciler diğer katmanla aynı olmak zorunda değildir. Bunun anlamı, modelin her bir katmanındaki küme yapıları sabit olmakla birlikte, temsilciler ve lider her bir katmanda birbirinden bağımsız olarak belirlenebilmektedir. Tezin bir sonraki aşamasında, dağıtık işlem programlayıcısının tasarım ve uygulaması gerçekleştirilmiştir.

Modelin bir üst katmanında grup yönetimi yer almaktadır. Diğer katmanlarda olduğu gibi, aynı küme tabanlı yapı bu katmana da uyarlanmıştır. Kümeler yine diğer katmalarla aynı topolojiye sahip olmakla birlikte, bu katmanda düğümler belirli bir işlemcide çalışan ve belli bir gruba ait olan işlemlerden oluşmaktadır. Küme tabanlı yaklaşım bu katmana da her bir grup için, grup lideri, küme temsilcileri ve kümelerde yer alan işlem birimlerinin belirlenmesiyle uygulanmıştır.

Sözü geçen bu yapıda, küme tabanlı model tekrar tekrar dağıtık sistemin her bir katmanı için uygulanmıştır. Sistemin saat senkronizasyonu, dağıtık işlem izlencelemesi ve grup yönetimi katmanları modelin bir boyutu, iki seviyeli küme tabanlı hiyerarşik yapı ise diğer bir boyutudur. Bu iki boyut birbirinden tamamiyle bağımsız bir yapıdadır. Şekil 3.2'de bu sistem modeli görülmektedir.



Şekil 3.2. İki boyutlu sistem modeli

3.2 İletişim Protokolü

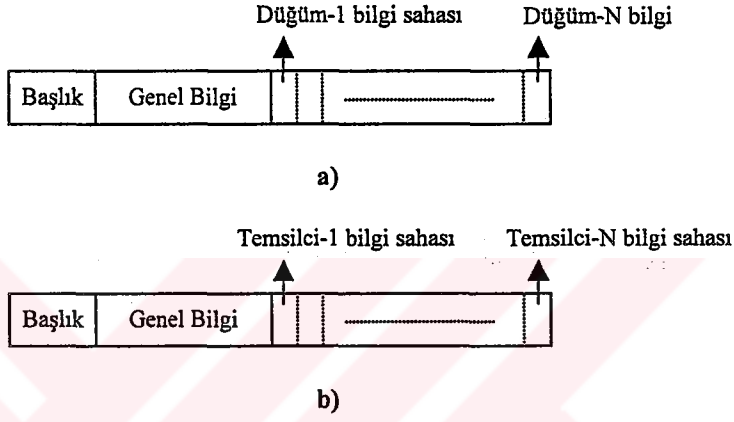
Bu bölümde, küme tabanlı sistem modelinin iletişim altyapısının işleyişi ve bu yapıya entegre edilen hata toleransı mekanizması detaylı olarak ele alınmaktadır.

İki seviyeli olarak tasarlanan bir hiyerarşik sistemde küme içi ve kümeler arası birbirinden bağımsız halka protokolleri işlemektedir. Her bir küme içinde düğümler arası bir mesaj çerçevesi dolaştırılmakta ve bu çerçeve dolaşımını küme temsilcisi idare etmektedir. İç halkada (inner ring) dolaşan bu mesaja "iççerçeve" (inframe) adı verilmektedir. Çerçeve yapısı iki ana bölümden oluşmaktadır. Bunlardan ilki, sistem yöneticisi tarafından düğümlere gönderilen, liderin verdiği kararları içeren (genel saat değeri gibi) genel bilgi bölümüdür. Bu bölüm düğümler tarafından yalnızca okunabilir özelliktedir. İkinci bölüm ise, düğümler tarafından doldurulan, liderin düğümlerden istediği lokal bilgileri (işlemcinin lokal saat değeri ya da yük durumu bilgisi gibi) içerir. İkinci bölümde her bir düğüme ayrılmış birer saha bulunur. Çerçeve yapısı şekil 3.3 (a)'da görülmektedir.

Lider olarak adlandırılan sistem yöneticisi iki seviyeli sistemin üst seviyesinde çalışan dış halkadaki mesaj dolaşımını idare eder. Dolaşan mesaj çerçevesine "dışçerçeve" (outframe) adı verilmektedir. İççerçeve gibi bu çerçeve de iki bölümden oluşur. Birinci bölüm, lider tarafından gönderilen ve küme temsilcileri tarafından okunabilir durumda olan genel bilgiyi (genel saat değeri gibi) içerirken, ikinci bölüm küme temsilcilerinin kendi küme üyelerinden sistem yöneticisine iletilmek üzere topladıkları (yerel saat değerleri ya da yük durumu bilgileri gibi) bilgileri taşır. Bu ikinci bölümde her bir küme temsilcisine ayrılmış birer saha bulunur. Çerçeve yapısı şekil 3.3 (b)'de görülmektedir.

Bir kümenin üyesi olan her bir düğüm, küme içindeki konumunu belirlemesini sağlayan, komşularının (kendinden bir sonra gelen düğüm ve kendinden bir önceki düğüm) ve küme temsilcisinin adresini bilmektedir. Ayrıca, küme temsilcisinden sonra gelen, yani kümenin ilk üyesi olan düğüm, küme temsilcisinden önce gelen, yani kümenin son

üyesi olan düğümün adresini bilmektedir. Bu bilgi sayesinde, küme temsilcisinin çökmesi durumunda, kümenin diğer düğümleri halka yapısını tekrar oluşturabileceklerdir. Bunun yanında kümelerin tüm elemanları liderin adresini de bilmektedirler.



Şekil 3.3. Çerçeve yapıları a) İç halka b) Dış halka

İç halkalarda olduğu gibi, dış halkada da sistemde yer alan her bir küme temsilcisi komşularının (kendinden bir önceki ve bir sonraki temsilcinin) ve liderin adresini bilmektedir. Dış halkanın ilk üyesi (liderden bir sonra gelen temsilci) ayrıca, dış halkanın son üyesinin (bir sonraki komşusu lider olan temsilci) adresini de bilmekte ve bu sayede liderin çökmesi halinde küme temsilcileri dış halkayı tekrar oluşturabilmektedirler.

Yukarıda bahsedilen işleyişten de anlaşılacağı gibi protokol yapısı üç ayrı modülden oluşmaktadır: "Düğüm", "Temsilci" ve "Lider". İkinci aşamasında bu protokolü kullanarak dağıtık işlem izlencelemesi modelini gerçekleştiren bu tez, düğüm ve lider modüllerinin uygulamasını

kapsamakta, temsilci modülünün uygulaması ise yine bu altyapıyı kullanarak grup yönetimi modelini gerçekleştirilen bir başka tezin kapsamında yer almaktadır. Ortak olarak uygulaması gerçekleştirilen bu protokol her iki tezin de altyapısını oluşturmaktadır.

3.2.1 Düğüm modülü tasarımı

3.2.1.1 Halkaya katılma

Düğüm modülü işleyişe başlarken üye olarak katılacağı kümenin temsilcisinin adresini bilmektedir. Düğüm ilk olarak bu temsilciye kümeye katılmak istediğini bildiren bir "KATILIM_İSTEĞİ" (JOIN_REQUEST) mesajı gönderir ve temsilciden kendisine gönderilecek olan, kümedeki konumuna ilişkin bilgilerin yer aldığı "KOMŞU_BİLGİSİ" (NEIGHBOUR_INFO) mesajını bekler. Bu mesajla düğüm, kendinden bir önce ve bir sonra gelen komşularının adresleriyle birlikte, lider adresini öğrenmiş olur ve kendi konfigürasyon bilgilerini günceller. Ardından da halkanın düzgün bir şekilde oluştuğunun garantilenmesi için küme temsilcisi tarafından dolaştırılan "İÇ_HALKA_KONTROL" (INNER_RING_CHECK) çerçevesini bekler. Bu mesajı alıp bir sonraki komşusuna ilettikten sonra düğüm kümeye katılmış olur ve olağan işleyişine geçerek "İÇÇERÇEVE" (INFRAME) beklemeye başlar ve "İÇÇERÇEVE" zamanlayıcısını başlatır.

3.2.1.2 Olağan işleyişi

Halkanın bir üyesi olan düğüm, temsilci tarafından küme içerisinde dolaşımı sağlanan "İÇÇERÇEVE" çerçevesini aldığında,

"İÇÇERÇEVE" zamanlayıcısını durdurur, mesajın "genel bilgi" kısmını okuyarak kendi içsel bilgilerini günceller. Aynı mesajın ikinci bölümünde kendisi için ayrılmış sahaya da kendi yerel bilgisini yerleştirir ve çerçeveyi kendinden bir sonra gelen düğüme gönderir. Ardından "İÇÇERÇEVE" zamanlayıcısını tekrar başlatır. Düğümün olağan işleyişini sürdürebilmesi için önceden belirlenmiş bir zaman dilimi içinde "İÇÇERÇEVE" mesajını alabilmelidir.

Olağan işleyişi sırasında herhangi bir anda küme temsilcisinden konum bilgilerini içeren "KOMŞU_BİLGİSİ" mesajı alan düğüm mesajdaki bilgiler doğrultusunda kendi bilgilerini günceller ve olağan işleyişini durdurarak halka kontrol çerçevesi olan "İÇ_HALKA_KONTROL" çerçevesinin kendine ulaşmasını beklemeye başlar.

Yine olağan işleyişi sırasında herhangi bir anda küme temsilcisinin küme üyeleriyle ilgili bilgileri toplamak amacıyla çıkarmış olduğu "HALKA_BİLGİ_TOPLA" (GATHER_RING_INFO) çerçevesini alan düğüm mesajın yazılabilir bölümünde kendisi için ayrılmış sahaya kendi adres bilgilerini kopyalar ve bu çerçeveyi bir sonraki düğüme gönderir. Bu çerçeveyi almasıyla düğüm, olağan işleyişini durdurarak halka kontrol çerçevesi olan "İÇ_HALKA_KONTROL" mesajının kendisine ulaşmasını beklemeye başlar.

Herhangi bir anda halka kontrol çerçevesi olan "İÇ_HALKA_KONTROL" mesajını alan düğüm, tekrar olağan duruma geçerek "İÇÇERÇEVE" beklemeye başlar.

3.2.1.3 Olağanüstü durumda işleyişi

Düğüm, olağan işleyişi sırasında belirli bir zaman içerisinde "İÇÇERÇEVE" alamazsa halka içerisinde bir problem olduğuna karar verir ve olağanüstü duruma geçer. Halkadaki problemin sebebini, yani çökmüş olan düğümü bulmak için, kendi komşularının çalışır durumda olduklarını anlamak amacıyla onlara birer "YAŞIYOR_MUSUN" (ARE_YOU_ALIVE) mesajı gönderir ve komşularından belirli bir zaman içerisinde "YAŞIYORUM" (I_AM_ALIVE) mesajının gelmesini bekler. Bu amaçla gönderdiği mesajlar için birer "YAŞIYOR_MUSUN" zamanlayıcısı başlatır.

Bu konumdayken bir komşusundan "YAŞIYORUM" mesajı alan düğüm ilgili komşusunun çalışır durumda olduğuna karar verir ve ilgili "YAŞIYOR_MUSUN" zamanlayıcısını durdurur.

Düğüm, herhangi bir anda bir komşundan çalışır durumda olup olmadığını sorgulayan "YAŞIYOR_MUSUN" mesajı alırsa, mesajı gönderen düğüme kendisinin çalışır durumda olduğunu bildirmek için "YAŞIYORUM" mesajı gönderir.

Bir komşusuna gönderdiği "YAŞIYOR_MUSUN" mesajına belirli bir sürede cevap alamayan düğüm ilgili komşusunun çökmüş olduğuna karar verir ve küme temsilcisine o komşsunun öldüğünü "ÖLÜ_KOMŞU" (DEAD_NEIGHBOUR) mesajı göndererek rapor eder. Bundan sonra çöken temsilciyi aradan çıkararak halkayı tekrar işler duruma getirme işlevi temsilcinin sorumluluğundadır.

3.2.1.4 Temsilcinin çökmesi durumunda işleyişi

Olağanüstü konumdayken halkanın ilk üyesi, yani kendinden bir önceki üye temsilci olan düğüm, eğer temsilciye göndermiş olduğu "YAŞIYOR_MUSUN" mesajına belirli sürede cevap alamazsa temsilcinin çökmüş olduğuna karar verir. Bundan sonra düğümün yapacağı işlem, halkayı tekrar çalışır duruma getirmek ve yeni temsilci modülünü başlatmaktır. Bunun için, adresini bildiği halkanın son üyesine yeni temsilci adresini bildirerek bozulan halkayı onarır ve ardından yeni temsilciyi başlatır. Bundan sonra halkanın tekrar işler duruma getirilmesinden yeni temsilci sorumludur.

3.2.1.5 Sonlu durum makinesi

Çizelge 3.1'de, düğüm modülünün sonlu durum makinesi gösterilmektedir. Şekil 3.4'te ise bu modüle ilişkin sonlu durum diyagramı görülmektedir. Bu şekilde, gelen olaylara karşılık modülün bulunduğu durumdan hangi duruma geçtiği görülmektedir.

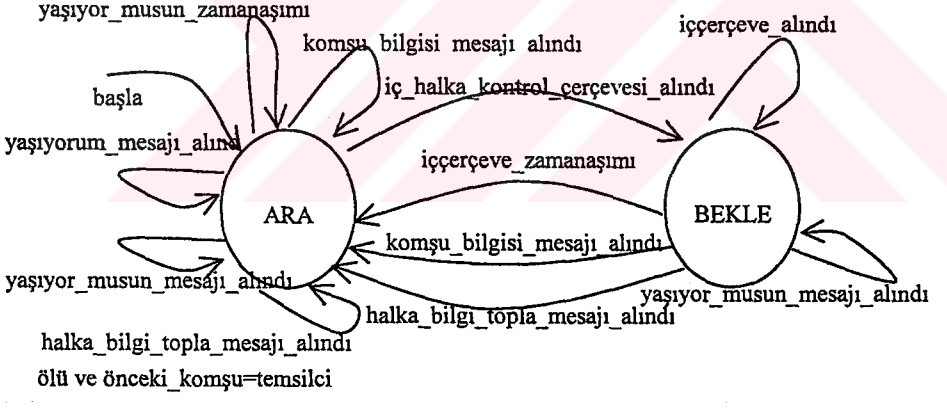
Çizelge 3.1. Düğüm modülü sonlu durum makinesi

Giden Olaylar:	**Giden Olaylar:**
(1) iç_halka_kontrol_çerçevesi_alındı	katılım_isteği_mesajı_gönder
(2) iççerçeve_alındı	yaşıyor_musun_mesajı_gönder
(3) iççerçeve_zamanaşımı	ölü_komşu_mesajı_gönder
(4) yaşıyor_musun_mesajı_alındı	yaşıyorum_mesajı_gönder
(5) yaşıyor_musun_zamanaşımı	iççerçeve_gönder
(6) yaşıyorum_mesajı_alındı	halka_bilgi_topla_çerçevesi_gönder
(7) komşu_bilgisi_mesajı alındı	iç_halka_kontrol_çerçevesi_gönder
(8) halka_bilgi_topla_çerçevesi_alındı	komşu_bilgisi_mesajı_gönder
	Durumlar:
	Ara
	Bekle

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Ara	1	0	0	4	5	6	7	9
Bekle	0	2	3	4	0	0	8	10

0:
1: durum=bekle
bilgi_sahasını_işaretle
iç_halka_kontrol_çerçevesi_gönder
iççerçeve_zamanlayıcısı_başlat
2: iççerçeve_zamanlayıcısı_durdur
genel_bilgi_al
yerel_bilgi_koy
iççerçeve_gönder
iççerçeve_zamanlayıcısı_başlat
3: durum=ara
yaşıyor_musun_mesajları_gönder
yaşıyor_musun_zamanlayıcıları_başlat
4: yaşıyorum_mesajı_gönder
5: Po: komşu_bilgisi_mesajı_gönder
yeni_temsilci_başlat
Po değil: ölü_komşu_mesajı_gönder
6: yaşıyor_musun_zamanlayıcısı_durdur
7: komşu_bilgisi_güncelle
8: durum=ara
halka_bilgisi_güncelle
9: adres_sahasını_işaretle
halka_bilgi_topla_çerçevesi_gönder
10: durum=ara
adres_sahasını_işaretle
halka_bilgi_topla_çerçevesi_gönder

Po: önceki_komşu



Şekil 3.4. Düğüm modülü sonlu durum diyagramı

Ek 1'de düğüm modülü algoritması gösterilmiştir.

3.2.2 Lider modülü tasarımı

3.2.2.1 Başlangıç işleyişi

Lider modülü ilk başlatıldığında kendi komşularının, yani dış halkanın ilk ve son üyelerinin adreslerini bilmektedir. Eğer bu adres bilgileri boş ise lider sistemde henüz hiç bir alt küme bulunmadığına karar verir ve bir temsilcinin halkaya dahil olma talebine cevap verebilmek için bekleme konumuna geçer.

Eğer komşu adres bilgileri dolu ise lider var olan dış halkada yer alan temsilcilerin adres bilgilerini toplamak amacıyla "HALKA_BİLGİ_TOPLA" (GATHER_RING_INFO) çerçevesini çıkararak halkanın ilk üyesine gönderir. Bu çerçeve halka içerisinde dolaşıp tekrar lidere ulaştığında, lider çerçeve içerisindeki, temsilcilerin adres bilgilerinin yer aldığı bölümü okuyarak kendi halka konfigürasyonu bilgilerini günceller.

Halka bilgilerini edindikten sonra lider, halkanın düzgün bir işleyişe sahip olduğundan emin olmak için halka kontrol çerçevesi olan "DIŞ_HALKA_KONTROL" (OUTER_RING_CHECK) çerçevesini çıkararak kümenin ilk üyesine gönderir. Bu çerçevenin de halka içerisinde dolaşarak lidere ulaşması sonucunda, lider olağan işleyişine başlar.

3.2.2.2 Olağan işleyişi

Lider, olağan işleyişi sırasında herhangi bir anda, bir temsilciden halkaya katılma isteğini bildiren "KATILIM_İSTEĞİ"

(JOIN_REQUEST) mesajı alırsa, isteği yapan temsilciyi halkanın sonuna ekler. Bunun için lider, halkanın son üyesine, bir sonraki komşusunun değiştiğini bildiren ve içerisinde ilgili üyenin yeni komşu adresinin yer aldığı "KOMŞU_BİLGİSİ" (NEIGHBOUR_INFO) mesajını gönderir. Bunun yanı sıra lider, halkanın ilk üyesi olan temsilciye , halkanın son üyesinin değiştiğini bildiren ve içerisinde bu yeni üyenin adresinin yer aldığı "KOMŞU_BİLGİSİ" mesajını gönderir. Lider ayrıca kendi halka bilgilerini güncelledikten sonra halkanın düzgün bir şekilde işler durumda olduğundan emin olmak için halka kontrol çerçevesi olan "DIŞ_HALKA_KONTROL" çerçevesini çıkarır. Bu çerçevenin halka üzerinde dolaşıp lidere ulaşmasının ardından lider tekrar olağan işleyişine geçer.

Lider olağan işleyişe geçtikten sonra belirli bir periyodda "DIŞÇERÇEVE" (OUTFRAME) çıkarır. "DIŞ_HALKA_KONTROL" çerçevesinin ulaşmasının ardından lider, ilk "DIŞÇERÇEVE" çerçevesini çıkarır, "DIŞÇERÇEVE" zamanlayıcısını başlatır ve bu çerçevenin belirli bir sürede geri dönmesini bekler. Eğer "DIŞÇERÇEVE" henüz lidere ulaşmadan çerçeve çıkarma periyodu dolarsa, lider yeni bir çerçeve çıkarmaz ve önceki çerçevenin gelmesini bekler. Eğer belirlenen sürede çerçeve hala lidere ulaşmamışsa, lider olağanüstü konuma geçer.

Lider, "DIŞÇERÇEVE" çerçevesini aldığında, çerçevenin yazılabilir bölümündeki temsilciler tarafından doldurulan bilgileri alarak kendi iç bilgilerini günceller. Ardından, eğer bir sonraki çerçeveyi çıkarma süresi dolmuşsa, temsilciler aracılığı ile sistemin düğümlerine iletilecek olan genel bilgiyi yeni "DIŞÇERÇEVE" çerçevesine koyar ve bu yeni çerçeveyi hemen çıkarır. Eğer çerçeve çıkarma süresi dolmamışsa, lider çerçeve çıkarma periyodunun dolmasını bekler ve bu periyod dolduğunda yeni "DIŞÇERÇEVE" çerçevesini çıkarır.

3.2.2.3 Olağanüstü durumda işleyişi

Lider çıkarmış olduğu "DIŞÇERÇEVE" çerçevesini belirli bir sürede alamazsa, dış halkada bir problem olduğuna karar verir ve olağanüstü duruma geçerek "DIŞÇERÇEVE" çıkarma periyodunu durdurur. Problemin sebebini, yani çökmüş olan temsilciyi tespit edebilmek için, öncelikle kendi komşularının çalışır durumda olduklarından emin olmak için "YAŞIYOR_MUSUN" (ARE_YOU_ALIVE) mesajları gönderir ve gönderdiği mesajlar için "YAŞIYOR_MUSUN" zamanlayıcılarını başlatır.

Lider komşularından belirli zaman içersinde "YAŞIYORUM" (I_AM_ALIVE) mesajı alamazsa ilgili komşsunun çökmüş olduğuna karar verir ve bu temsilciyi halkadan çıkararak halkayı onarır. Bunun için ölen temsilcinin diğer komşusuna yeni komşusunun kendisi olduğunu bildiren "KOMŞU_BİLGİSİ" mesajı gönderir. Ayrıca, eğer ölen temsilci halkanın son üyesiyse, halkanın ilk üyesine son üyenin değiştiğini bildiren bir "KOMŞU_BİLGİSİ" mesajı gönderir. Bu mesajların ardından lider halkanın düzgün bir şekilde işlerliğinden emin olmak için halka kontrol çerçevesi olan "DIŞ_HALKA_KONTROL" çerçevesini çıkarır. Bu çerçevenin halka içerisinde dolaşarak lidere ulaşmasının ardından, lider tekrar olağan işleyişine geçer.

Lider, gönderdiği "YAŞIYOR_MUSUN" mesajına karşılık belirli zaman içerisinde ilgili komşusundan çalışır durumda olduğunu bildiren "YAŞIYORUM" mesajı alırsa buna ilişkin başlattığı "YAŞIYOR_MUSUN" zamanlayıcısını durdurur ve bu komşusunun çalışır durumda olduğuna karar verir. Lider eğer gönderdiği her iki

"YAŞIYOR_MUSUN" mesajına da cevap almışsa "DIŞ_HALKA_KONTROL" çerçevesi çıkarır.

Lider, herhangi bir anda bir komşsunun gönderdiği "YAŞIYOR_MUSUN" mesajına karşılık "YAŞIYORUM" mesajı göndererek kendisinin çalışır durumda olduğunu bildirir.

Olağan işleyişi sırasında herhangi bir anda halkadaki bir temsilciden komşusunun çökmüş olduğunu bildiren "ÖLÜ_KOMŞU" (DEAD_NEIGHBOUR) mesajı alırsa, lider olağanüstü konuma geçer ve "DIŞÇERÇEVE" çıkarma periyodunu durdurur. Ölmüş olduğu rapor edilen temsilcinin komşularına yeni komşularını bildiren birer "KOMŞU_BİLGİSİ" mesajı gönderir ve kendi halka bilgilerini günceller. Ardından halkanın düzgü bir şekilde işlediğinden emin olmak için "DIŞ_HALKA_KONTROL" çerçevesini çıkarır ve bu çerçevenin dolaşımının ardından tekrar olağan işleyişine geçer.

3.2.2.4 Sonlu durum makinesi

Çizelge 3.2'de, lider modülünün sonlu durum makinesi gösterilmektedir. Şekil 3.5'te ise lider modülüne ilişkin sonlu durum diyagramı görülmektedir. Bu şekilde, gelen olaylara karşılık liderin bulunduğu durumdan hangi duruma geçtiği görülmektedir.

Çizelge 3.2. Lider modülü sonlu durum makinesi

Gelen Olaylar:

(1) dış_halka_kontrol_çerçevesi_alındı
(2) dış_çerçeve_alındı
(3) yaşıyor_musun_mesajı_alındı
(4) yaşıyorum_mesajı_alındı
(5) dış_halka_bilgi_topla_çerçevesi_alındı
(6) katılım_isteği_mesajı_alındı
(7) ölü_komşu_mesajı_alındı
(8) dışçerçeve_zamanaşımı
(9) yaşıyor_musun_zamanaşımı
(10) çerçeve_çıkarma_zamanaşımı

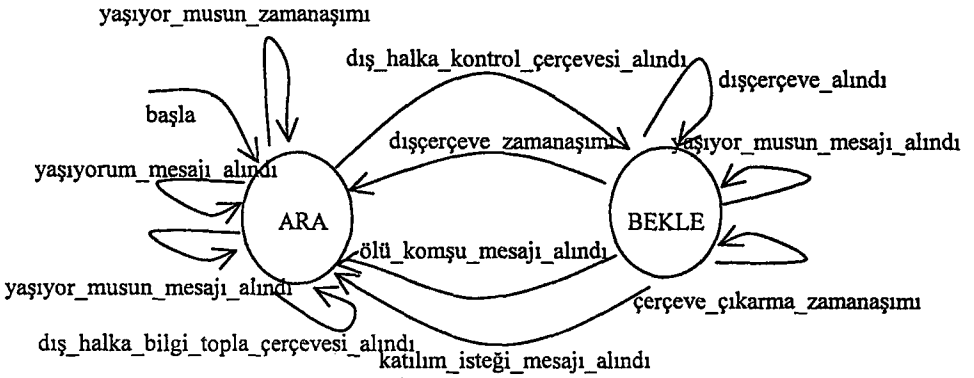
Giden Olaylar:
komşu_bilgisi_mesajı_gönder
yaşıyor_musun_mesajı_gönder
yaşıyorum_mesajı_gönder
dış_çerçeve_çıkar
dış_halka_bilgi_topla_çerçevesi_çıkar

Durumlar:
Ara
Bekle

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
Ara	1	0	3	4	5	0	0	0	9	0
Bekle	0	2	3	0	0	6	7	8	0	10

0:
1: dışçerçeve_zamanlayıcısı_durdur
durum=bekle
dışçerçeve_çıkar
dış_çerçeve_zamanlayıcısı_başlat
çerçeve_çıkarma_zamanlayıcısı_başlat
2: dışçerçeve_zamanlayıcısı_durdur
Po: dış_çerçeve_çıkar
dışçerçeve_zamanlayıcısı_başlat
çerçeve_çıkarma_zamanlayıcısı_başlat
çerçeve_durumu=boş
Po değil:
çerçeve_durumu=çerçeve_alındı
3: yaşıyorum_mesajı_gönder
4: yaşıyor_musun_zamanlayıcısı_durdur
5: dışçerçeve_zamanlayıcısı_durdur
halka_bilgisi_güncelle
komşu_bilgisi_mesajları_gönder
dış_halka_kontrol_çerçevesi_çıkar
dışçerçeve_zamanlayıcısı_başlat
6: dış_çerçeve_zamanlayıcısı_durdur
çerçeve_çıkarma_zamanlayıcısı_durdur
durum=ara
halka_bilgisi_güncelle
komşu_bilgisi_mesajları_gönder
dış_halka_kontrol_çerçevesi_çıkar
dışçerçeve_zamanlayıcısı_başlat
7: dış_çerçeve_zamanlayıcısı_durdur
çerçeve_çıkarma_zamanlayıcısı_durdur
durum=ara
halka_bilgisi_güncelle
komşu_bilgisi_mesajları_gönder
dış_halka_kontrol_çerçevesi_çıkar
dışçerçeve_zamanlayıcısı_başlat
8: dışçerçeve_zamanlayıcısı_durdur
çerçeve_çıkarma_zamanlayıcısı_durdur
durum=ara
yaşıyor_musun_mesajları_gönder
yaşıyor_musun_zamanlayıcısı_başlat
9: halka_bilgisi_güncelle
komşu_bilgisi_mesajları_gönder
dış_halka_kontrol_çerçevesi_çıkar
dışçerçeve_zamanlayıcısı_başlat
10: P1: dışçerçeve_çıkar
dışçerçeve_zamanlayıcısı_başlat
çerçeve_çıkarma_zamanlayıcısı_başlat
çerçeve_durumu=boş
P1 değil:
çerçeve_durumu=çerçeve_çıkar

Po: çerçeve_durumu=çerçeve_çıkar
P1: çerçeve_durumu=çerçeve_alındı



Şekil 3.5. Lider modülü sonlu durum diyagramı

Ek 2'de lider modülü algoritması gösterilmiştir.

4 DAĞITIK YÜK DENGELEME MODÜLÜ TASARIMI

Bu bölümde, tezin ilk aşamasında uygulaması gerçekleştirilen gerçek zamanlı dağıtık hiyerarşik halka tabanlı iletişim altyapısının üzerine geliştirilen bir dağıtık işlem izlencelemesi (distributed process scheduling) modülü açıklanmaktadır.

4.1 Sistemin Yapısı

Bölüm 2.3'te de ayrıntılarıyla anlatıldığı gibi bir dağıtık işlem programlayıcısının görevi sistemin yükünü işlemci birimleri arasında etkin, adil ve saydam bir şekilde paylaştırmaktır. Bu amaçla, uygulaması gerçekleştirilen hiyerarşik halka tabanlı dağıtık sistem modelinin üzerine tasarlanan dağıtık işlem izlenceleme modülü sistemin yapısına uygun bir şekilde yük paylaşımını gerçekleştirmektedir.

Model, şekil 3.1'deki gibi iki seviyeli bir halka yapısına uyarlandığında, düğümler dağıtık sistemin işlemci birimlerine karşılık gelmekte, her bir kümeyi birer temsilci yönetmekte ve dış halkayı, dolayısı ile tüm sistemi yöneten birim de lider olmaktadır. Temsilciler, sorumlu oldukları iç halkalarda periyodik olarak çıkardıkları çerçevelerle düğümlerden yük durumlarını toplamaktadırlar. Lider, dış halkada periyodik olarak çıkardığı çerçeveye temsilcilerin kendi kümelerine ilişkin yerleştirdikleri yük değerlerini toplayarak tüm düğümlerin yük durumlarını elde etmektedir.

Sistemdeki herhangi bir işlemciye, yani düğüme bir iş isteği geldiğinde, öncelikle düğüm bu isteği karşılayabilmek için yeterli kaynağı olup olmadığını kontrol eder. Eğer kaynakları yeterliyse düğüm

işlemi kendi üzerinde başlatır. Düğüm yeterli kaynağa sahip değilse, kendi küme temsilcisine bir yük transfer isteğinde bulunur. Bu isteği alan temsilci kendi kümesinden topladığı bilgiler doğrultusunda işlemi çalıştıracak uygun bir düğüm arar. Uygun düğümü bulduğu taktirde bu düğümün adresini isteği yapan düğüme geçirir ve bundan sonra ilgili düğümler arasında yük transferi gerçekleştirme süreci başlatılır. Eğer temsilci kendi kümesinde uygun bir düğüm bulamazsa isteği lidere aktarır. Bu isteği alan lider toplamış olduğu yük bilgileri doğrultusunda sistemde yükü paylaştırabileceği uygun bir düğüm arar. Uygun düğümü bulduğu taktirde de bu düğümün adresini isteği yapan düğüme göndererek yük transfer sürecinin başlatılmasını sağlar.

Sistemin gerçek zamanlı özelliğini koruyabilmek için ise, yük transfer işlemine belli bir zaman sınırı getirilmiştir. Eğer düğümün yük transfer isteğine belli bir zaman içerisinde cevap verilemezse düğüm isteği kendi üzerinde başlatır.

Yük değeri olarak düğümde çalışan işlem sayısı, yani işlemci kuyruk uzunluğu kullanılmaktadır. Sistemin yük dengesini sağlayabilmek için temsilciler ve lider düğümleri üç ayrı sınıfa ayırırlar: yük seviyesi düşük olan düğümler, yük seviyesi orta olan düğümler ve yüksek yük seviyeli düğümler. Yük dağıtım işlemi öncelikle temsilci tarafından küme içerisinde düşük ve daha sonra orta seviyeli düğümlere doğru yapılır. Eğer ilgili kümenin tüm düğümleri yüksek yük durumundaysa, lider devreye girerek başka bir kümede yük dengelemesi için öncelikle düşük seviyeli, sonra da orta seviyeli düğümleri seçer. Böylece küme içi ve kümeler arası yük paylaşımı gerçekleştirilerek, sistem yükü belli bir dengeye getirilmektedir.

Transfer Politikası: Transfer politikası olarak işlemci kuyruk uzunluğunu baz alan bir eşik yöntemi kullanılır. Yeni bir işlemin başlatılmak istendiği düğümün yükü yüksek yük eşik seviyesini aşıyorsa düğüm transfer isteğini yapar.

Seçim Politikası: Sistemde sadece yeni gelen işler transfer edilmek üzere seçilir.

Lokasyon Politikası: Yük transferi için uygun alıcı bulma işlemi iki aşamadan oluşmaktadır. İlk aşamada temsilci edindiği bilgiler doğrultusunda kendi kümesi içerisinde uygun bir alıcı bulmaya çalışır. Eğer bu aşama başarısız olursa, ikinci olarak lider sahip olduğu sistemin bütününe ilişkin bilgiler doğrultusunda yük transferi için uygun bir alıcı arar.

Bilgi Politikası: Periyodik bilgi toplama yöntemi kullanılmaktadır. Lider belirli periyodlarda çıkardığı çerçevelerle temsilcilerden, temsilciler de kendi iç halkalarında çıkardıkları çerçevelerle düğümlerden yük bilgilerini toplarlar.

Tasarlanan bu yapı, hiyerarşik halka tabanlı protokole uygun olduğu gibi sistemin kaynaklarını etkin bir şekilde yönetebilmektedir. Bilgi toplama mekanizmasının periyodik olması, sistemin senkron çerçeve dolaştırma yapısına kolaylıkla entegre edilebilmesini sağlamıştır. Geliştirilen yapı, sistemin genel yük durumuna göre kendini adapte edebilir bir yapıda olmamasına rağmen, bilgi toplama işleminin sisteme getireceği yük sistemin yük seviyesindeki artma ya da azalmalara karşın hep aynı düzeyde kalacaktır. Ayrıca, sistemin küme tabanlı yapısı ve lokasyon politikasının merkezi bir yapıda temsilci ve lider tarafından belirlenmesi de yük dağıtımı mekanizmasının iletişim hattı kullanımını

belli sınırlarda tutabilmektedir. Bu özelliğiyle dağıtık işlem izlencelemesi modelinin sisteme getirdiği yük tahmin edilebilir bir düzeyde olmakta ve model sistemi dengesiz bir yapıya sürüklememektedir.

Bunun yanısıra seçim politikası yalnızca yeni gelen işlemleri transfer edilmek üzere seçtiğinden (nonpreemptive) transfer işlemi sisteme fazla yük getirmemektedir. Ayrıca, sistemde gölge işlem (shadow process) kavramı kullanılmakta, bu yapıda her bir düğümde sistemdeki işlemlerin birer kopyası bulunmakta ve bu sayede yük transferi yapılırken işlemin sadece adının gönderilmesi yeterli olmaktadır.

4.2 Modülün İşleyişi

4.2.1 Düğüm modülü işleyişi

Düğüm, periyodik olarak aldığı "İÇÇERÇEVE" çerçevesinde kendine ayrılmış olan sahaya o anki yük değerini kopyalar ve çerçeveyi bir sonraki düğüme gönderir.

Düğüme, uygulama tarafından yeni bir işlem başlatmak için "SÜREÇ_BAŞLAT" (START_PROCESS) mesajı geldiginde (bu komutla birlikte düğüme çalıştırılacak olan işlemin adı geçirilir), düğüm öncelikle kendi yük değerini kontrol eder, eğer yük değeri YÜKSEK (HIGH) yük seviyesinden düşükse işlemi kendi üzerinde başlatır ve yük değerini arttırır. Eğer düğümün yük miktarı yüksek yük seviyesinden büyükse düğüm temsilciye yük transferi yapmak istediğini bildiren bir "YÜK_TRANSFER_İSTEĞİ) (LOAD_TRANSFER_REQUEST) mesajı gönderir ve "YÜK_TRANSFER" zamanlayıcısını başlatır. Düğüm bu

mesaja karşılık yük transferi yapabileceğini bildiren "YÜK_TRANSFER_ALICISI" (LOAD_TRANSFER_DESTINATION) mesajı alırsa mesajda adresi yer alan diğer bir düğüme yük transferi yapmak istediğini bildiren "YÜK_TRANSFER_İSTEĞİ" mesajı gönderir ve "YÜK_TRANSFER" zamanlayıcısını başlatır. Eger bu mesaja karşılık alıcı düğümden yük transferi işleminin tamamlandığını bildiren "YÜK_TRANSFER_BİTTİ" (LOAD_TRANSFER_ COMPLETE) mesajı dönerse transfer başarıyla gerçekleştirilmiş demektir. Düğüme eğer temsilci, lider ya da yük transfer isteğinde bulunduğu düğümden yük transferi gerçekleştiremeyeceğini bildiren "YÜK_TRANSFER_RED" (LOAD_TRANSFER_REJECT) mesajı dönerse, ya da "YÜK_TRANSFER" zamanlayıcısı doluncaya kadar olumlu ya da olumsuz hiç bir cevap dönmezse düğüm işlemi kendi üzerinde başlatır ve yük değerini günceller.

Düğüme herhangi bir anda bir başka düğüm yük transfer isteğinde bulunmak için "YÜK_TRANSFER_İSTEĞİ" mesajı gönderdiğinde (mesajla birlikte transfer edilecek işlemin adı da gelir) alıcı düğüm öncelikle kendi yük değerini kotrol ederek işlemi başlatmaya uygunsa, yani yük değeri yüksek yük seviyesinin altındaysa, işlemi kendi üzerinde başlatarak yük değerini arttırır ve transfer isteğini yapan düğüme yük transferinin başarıyla gerçekleştiğini bildiren "YÜK_TRANSFER_BİTTİ" mesajı gönderir. Eğer alıcı düğümün yük değeri yüksek seviyedeyse isteği yapan düğüme yük transferini gerçekleştiremeyeceğini bildiren "YÜK_TRANSFER_RED" mesajı gönderir.

Düğümün üzerinde çalışan işlemlerden biri sonuçlandığında, öncelikle yük değerini düşürür. Ardından işlemin statüsünü kontrol eder. Eğer bu işlem yerel olarak çalışan bir işlemse sonucu çıktılar. Eğer

sonlanan işlem başka bir düğümden transfer edilmiş olan bir işlemse düğüm transfer işlemini yapan gönderici düğüme işlemin sonuçlandığını bildiren "SÜREÇ_SONLANDI" (TERMINATE_PROCESS) mesajı gönderir ve bu mesajın içerisine işlemin sonucunu da yerleştirir.

Düğüm bir başka düğümden, transfer edilmiş olan bir işlemin sonuçlandığını bildiren "SÜREÇ_SONLANDI" mesajı alırsa, işlemin uzak düğümde sonuçlandığını anlar ve mesajın içindeki işlem sonucunu sanki işlem kendi üzerinde sonlanmış gibi çıktılar.

Ek 3'te düğüm modülü algoritması gösterilmiştir.

4.2.2 Temsilci modülü işleyişi

Temsilci, iç halkada çıkarmış olduğu "İÇÇERÇEVE" çerçevesinden topladığı düğümlerin yük değerlerini okuyarak, yük değerlerine göre düğümleri üç ayrı tabloda sınıflandırır: "düşük_tablo" (low_table), "orta_tablo" (medium_table) ve "yüksek_tablo" (high_table).

Temsilci, dış halkadan aldığı, liderin periyodik olarak çıkardığı "DIŞÇERÇEVE" çerçevesinde kendine ayrılan sahaya iç halkadan topladığı, düğümlerin yük değerlerini kopyalar ve çerçeveyi bir sonraki temsilciye gönderir.

Temsilci, herhangi bir anda kendi kümesine ait bir düğümün yük transfer isteğini bildiren "YÜK_TRANSFER_İSTEĞİ" (LOAD_TRANSFER_REQUEST) mesajını alınca, öncelikle düşük_ tabloyu kontrol eder. Eğer bu tablo boş değilse, tablonun en başındaki

düğüm adresini seçer. Eğer bu tablo boş ise orta_tablonun en başındaki düğüm adresini seçer.

Temsilci, bir düğüm seçebildiyse transfer isteğini yapan düğüme seçtiği düğümün adresini "YÜK_TRANSFER_ALICISI" (LOAD_TRANSFER_DESTINATION) mesajının içerisine yerleştirerek gönderir. Seçilen düğümün yük değerini güncelleyerek, yük seviyesine göre tekrar uygun tabloya yerleştirir.

Eğer düşük_tablo ve orta_tablonun her ikisi de boş ise temsilci kendi kümesi içerisinde yük transferi yapılabilecek uygun bir düğüm bulamayarak isteği lidere yönlendirir ve "YÜK_TRANSFER_İSTEĞİ" mesajını lidere gönderir.

Ek 4'te temsilci modülü algoritması gösterilmiştir.

4.2.3 Lider modülü işleyişi

Lider, periyodik olarak çıkardığı "DIŞÇERÇEVE" çerçevesinden sistemdeki tüm düğümlerin yük değerlerini okuyarak yük durumlarına göre düğümleri üç ayrı tabloda sınıflandırır: "düşük_tablo" (low_table), "orta_tablo" (medium_table) ve "yüksek_tablo" (high_table).

Lider, herhangi bir anda bir temsilciden, bir düğümün yük transfer isteğinde bulunduğunu bildiren "YÜK_TRANSFER_İSTEĞİ" (LOAD_TRANSFER_REQUEST) mesaji alınca, öncelikle düşük_tabloyu kontrol eder. Eğer bu tablo boş değilse, tablonun en başındaki düğüm adresini seçer. Eğer bu tablo boş ise orta_tablonun en başındaki düğüm adresini seçer.

Lider, yük transferi için uygun bir düğüm seçebildiyse transfer isteğini yapan düğüme seçtiği düğümün adresini "YÜK_TRANSFER_ALICISI" (LOAD_TRANSFER_DESTINATION) mesajının içerisine yerleştirerek gönderir. Seçilen düğümün yük değerini arttırarak, tekrar yük seviyesine göre uygun tabloya yerleştirir.

Eğer düşük_tablo ve orta_tablonun her ikisi de boş ise lider yük transferi yapılabilecek uygun bir düğüm bulamayarak isteği yapan düğüme "YÜK_TRANSFER_RED" (LOAD_TRANSFER_REJECT) mesajı gönderir.

Ek 5'te lider modülü algoritması gösterilmiştir.

4.3 Mesaj ve Zaman Karmaşıklıkları

İki seviyeli bir sistemde, küme içi yük dengeleme işlemi senaryosunda düğümden temsilciye doğru yapılan isteğe temsilcinin cevap vermesi sonucunda düğüm, alıcı düğüme yük transfer isteği yapmakta ve alıcı düğüm tarafından da işlemin gerçekleştiğini bildiren bir cevap gönderilmektedir. Bir düğümden diğer bir düğüme veya temsilciye mesaj gönderim süresi t olarak kabul edilirse küme içi yük transferi işlemi $4t$ kadar bir sürede gerçekleşmekte, böylece zaman karmaşıklığı $O(1)$ olmaktadır. Aynı şekilde bu işlem toplam 4 mesajda tamamlandığından mesaj karmaşıklığı da $O(1)$ olarak gerçekleşmektedir.

Kümeler arası yük transferinde ise, temsilci tarafından lidere yapılan bir mesaj gönderimi olmakta ve transfer işlemi bu durumda toplam 5 adet ardışık mesaj gönderimi sonunda ve böylece $5t$ süresinde

tamamlanmaktadır. Bu durumda da zaman ve mesaj karmaşıklıkları *O(1)* olmaktadır.

Sonuç olarak, yük dengeleme mesaj ve zaman karmaşıklıkları sistemdeki düğüm ya da küme sayısına göre değişim göstermemektedir.

5 ALGORİTMALARIN DOĞRULUK ANALİZLERİ

Bu bölümde, bölüm 3 ve 4'te tasarımları gerçekleştirilen düğüm, lider ve dağıtık yük izlencelemesi algoritmalarının doğruluk analizleri gerçekleştirilecektir. Bu işlem için öncelikle bir algoritmanın I/O automaton modeli (Lynch, 1997) tanımlanacak, daha sonra algoritmayla ilgili birtakım teoremler ortaya atılarak ispatları gerçekleştirilecektir.

I/O automaton modeli, dağıtık sistemi oluşturan ve birbirleriyle ilişkili olan parçaları modellemede kullanılan bir yöntemdir. I/O automaton, geçişleri isimlendirilmiş aksiyonlarla ilişkilendirilmiş bir çeşit durum geçiş makinesidir. Bu aksiyonlar girdi (input), çıktı (output) ve içsel (internal) olarak üç şekilde sınıflandırılırlar. Girdi ve çıktı aksiyonları automaton'un çevresiyle iletişimde bulunmasını ifade ederken, içsel aksiyonlar dışa kapalıdır. Bir I/O automaton'un girdi, çıktı ve içsel aksiyonlarını tanımlayan yapıya o I/O automaton'un imzası (signature) adı verilir. Bir I/O automaton A tanımlanırken, imzası (girdi, çıktı ve içsel aksiyonları) ve durumları (states) ve geçişleri (transtions) belirtilmektedir.

I/O automaton modelinde, zaman kısıtlarının belirtilebilmesi için bir eklenti yapılarak zamanlanmış I/O Automaton (Timed I/O automaton) modeli ortaya çıkmıştır. Bu modelde girdi, çıktı ve içsel aksiyonların yanında bir de zaman-geçiş (time-passage) aksiyonu tanımlanır. Bir zaman-geçiş aksiyonu $v(t)$, $t \varepsilon R^+$, t değeri kadarlık bir zaman geçişini ifade eder. Bir zamanlanmış I/O automaton modelinin girdi, çıktı, içsel ve zaman-geçiş aksiyonlarından oluşan zamanlanmış imzası (timed signature) tanımlanır.

Bir I/O automaton ispatlanırken kullanılan yöntemlerden biri değişmez iddialar (invariant assertions) ortaya atmaktır. Bir değişmez iddia, bir I/O automaton'un her ulaşılabilir durumu için doğru olan bir ifadedir. Doğruluk analizleri yapılırken belirlenen teoremler, bir takım değişmez iddiaların açıklanmasıyla ispatlanmaktadır.

Bir çalışma (execution), I/O automaton'un durum değişimlerine göre gerçekleştirdiği aksiyonlar ve bu aksiyonlar sonunda geldiği durumları gösteren sonlu ya da sonsuz dizilerdir. Bir iddia I/O automaton modeline göre ispatlanırken, o iddianın çalışması (execution) da belirtilmiştir.

5.1 Düğüm Algoritması Doğruluk Analizi

5.1.1 I/O automaton modeli

M ε {INFRAME, ARE_YOU_ALIVE, I_AM_ ALIVE,
JOIN_REQUEST, NEIGHBOUR_INFO, RING_CHECK,
GATHER_RING, DEAD_NEIGHBOUR}

T ε {T_INFRAME, T_ARE_YOU_ALIVE_PRED,
T_ARE_YOU_ALIVE_SUCC}

msg contains type ε M, data, source_addr, dest_addr
ring_info contains pred_addr, succ_addr, leader_addr, rep_addr,
reppred_addr, repsucc_addr

Timed Signature:
Input:
$init_i$(rep_addr), rep_addr is addres value
$receive_{i,j}$(m), m is msg

OutPut:

$send_{i,j}(m)$, m is msg

Internal:
$process\_msg_i(m)$, m is msg
$process\_tmo_i(tm)$, tm ε T

Time-passage:
v(t), t ε R^+

States:
role ε {Node, Representative}, initially Node
state ε {wait,search}, initially search
send_buf, a FIFO queue of elements of msg, initially empty
recv_buf, a FIFO queue of elements of msg, initially empty
time_buf, a FIFO queue of elements of T, initially empty
r_info is ring_info, initially null
timeval, a vector indexed as {inframe,are_you_alive_succ,are_you_alive_pred}, initially set to some predefined timeout values
last, a vector indexed as {inframe,are_you_alive_succ,are_you_alive_pred}, initially all of them are ∞

Transitions:
$init_i(rep\_addr)$
Effect:
m is msg
r_info.rep_addr=rep_addr
m.type=JOIN_REQUEST
m.dest_addr=r_info.rep_addr
add m to send_buf
state=search

$send_{i,j}(m)$
Precondition:
m is first on send_buf
Effect:
remove m from send_buf

```
receive_{i,j}(m)
   Effect:
        add m to recv_buf

v(t)
   Effect:
        case
            last(inframe)<>∞ And now+t>=last(inframe):
               add T_INFRAME to time_buf
               last(inframe)= ∞
            last(are_you_alive_succ)<>∞ And
            now+t>=last(are_you_alive_succ):
               add T_ARE_YOU_ALIVE_SUCC to time_buf
               last(are_you_alive_succ)= ∞
            last(are_you_alive_pred)<>∞ And
now+t>=last(are_you_alive_pred):
               add T_ARE_YOU_ALIVE_PRED to time_buf
               last(are_you_alive_pred)= ∞
        now=now+t

process_msg_i(m)
   m is first on recv_buf
   Effect:
        remove m from recv_buf
        case
            m.type=INFRAME:
               if state=wait then
                        last(inframe)= ∞
                        m.dest_addr=r_info.succ_addr
                        add m to send_buf
                        last(inframe)=now+timeval(inframe)
            m.type=NEIGHBOUR_INFO:
               if state=wait then
                        last(inframe)= ∞
                        state=search
                        r_info=m.data
            m.type=RING_CHECK:
               state=wait
```

```
            last(inframe)= ∞
            m.dest_addr=r_info.succ_addr
            add m to send_buf
            last(inframe)= now+timeval(inframe)
        m.type=GATHER_RING_INFO:
            if state=wait then
                    last(inframe)= ∞
                    state=search
            append i to m.data
            m.dest_addr=r_info.succ_addr
            add m to send_buf
        m.type=ARE_YOU_ALIVE:
            m.type=I_AM_ALIVE
            m.dest_addr=m.source_addr
            add m to send_buf
        m.type=I_AM_ALIVE:
            if m.source_addr=r_info.pred_addr then
                    last(are_you_alive_pred)= ∞
            else
                    last(are_you_alive_succ)= ∞

process_tmo_i(tm)
    Precondition:
        tm is first on time_buf
    Effect:
        remove tm from time_buf
        m is msg
        case
            tm=T_INFRAME:
                state=search
                m.type=ARE_YOU_ALIVE
                m.dest_addr=r_info.pred_addr
                add m to send_buf
                last(are_you_alive_pred)=
                        now+timeval(are_you_alive_pred)
                if r_info.pred_addr<>r_info.succ_addr then
                        m.dest_addr=r_info.succ_addr
                        add m to send_buf
                        last(are_you_alive_succ)=
```

```
                        now+timeval(are_you_alive_succ)
tm=T_ARE_YOU_ALIVE_PRED
   if r_info.pred_addr=r_info.rep_addr then
         r_info.pred_addr=r_info.reppred_addr
         m.type=NEIGHBOUR_INFO
         m.dest_addr=r_info.pred_addr
         add m to send_buf
         role=representative
   Else
         m.type=DEAD_NEIGHBOUR
         m.dest_addr=r_info.rep_addr
         add m to send_buf
tm=ARE_YOU_ALIVE_SUCC
   if r_info.succ_addr<>r_info.rep_addr then
         m.type=DEAD_NEIGHBOUR
         m.dest_addr=r_info.rep_addr
         add m to send_buf
```

5.1.2 Doğruluk analizi

Düğüm modülünün işleyişi üç teorem ile açıklanmıştır. Teoremlerden ilki düğümün dahil olduğu halka içindeki olağan işleyişini, ikincisi düğümün kendi halkası içindeki bir başka düğümün çökmesi durumunda üstlendiği görevi, üçüncü teorem ise temsilcinin çökmesi durumunda düğümün yeni bir temsilci belirlemesini açıklamaktadır. Teoremler ispatlanırken işleyişle ilgili iddialar (assertion) belirlenmiş, bu iddialar önceki bölümde ortaya çıkarılan I/O Automaton modeline göre açıklanmış ve işleyişleri (trace) gösterilmiştir.

Teorem 1. *Düğüm, herhangi bir iç halkanın üyesi olarak işlev görür.*

Bu teoremin ispatı için aşağıdaki iddialar belirlenmiştir:

İddia 1.1. Düğüm başlatıldığında halkaya katılır ve iççerçeve bekleme konumuna geçer.

İspat: Düğüm, *init* aksiyonunda parametre olarak aldığı temsilci adresine *JOIN_REQUEST* mesajı gönderir. Bu durumda *state=search* durumundadır. Temsilciden gelen *NEIGHBOUR_INFO* mesajını alarak halka bilgilerini *r_info*'ya kopyalar. Böylece komşu düğümlerin adreslerini edinmiş olur. Düğüm, *RING_CHECK* çerçevesini aldıktan sonra *state=wait* konumuna geçer, çerçeveyi adresi *r_info.succ_addr* olan bir sonraki düğüme gönderir ve *inframe* zamanlayıcısını başlatır. Böylece düğüm, halkaya giriş sürecini tamamlayarak iççerçeve almaya hazır konuma gelmiş olur ve *timeval(inframe)* süresi kadar bir süre iççerçeve bekler. Bu durumun çalışması aşağıdaki gibidir:

init,send_buf=[JOIN_REQUEST];state=search,send(JOIN_REQUEST), send_buf=[],receive(NEIGHBOUR_INFO),recv_buf=[NEIGHBOUR_ INFO],process_msg,recv_buf=[];r_info=m.data,receive(RING_CHECK) ,recv_buf[RING_CHECK],process_msg,recv_buf=[];state=wait; last(inframe)=now+timeval(inframe);send_buf=[RING_CHECK],send (RING_CHECK),send_buf=[]

İddia 1.2. Düğüm, *timeval(inframe)* süresi içerisinde aldığı iççerçeveyi komşu düğüme iletir.

İspat: *state=wait* konumunda olan düğüm iççerçeveyi aldığında *process_msg* aksiyonu çalışır. *inframe* zamanlayıcısını durdurur, iççerçeveyi adresi *r_info.succ_addr* olan bir sonraki düğüme gönderir ve *inframe* zamanlayıcısını tekrar başlatarak bir *timeval(inframe)* süresi kadar daha iççerçeve bekler. Süreç bu şekilde tekrar ederek devam eder. Bu durumun çalışması aşağıdaki gibidir:

state=wait,receive(INFRAME),recv_buf=[INFRAME],process_msg,recv_buf=[];last(inframe)=now+timeval(inframe);send_buf=[INFRAME], send(INFRAME),...

İddia 1.3. Düğüm, temsilciden halka bilgisi aldığında kendi bilgilerini günceller.

İspat: Düğüm, *NEIGHBOUR_INFO* mesajı aldığında *r_info*'yu gelen mesajdaki bilgilere göre günceller ve *state=search* durumuna geçer. *inframe* zamanlayıcısını durdurur ve halkanın tekrar işler hale gelmesini bekler. Bu durumun çalışması aşağıdaki gibidir:

state=wait,receive(NEIGHBOUR_INFO),recv_buf=[NEIGHBOUR_INFO],process_msg, recv_buf=[]; state=search;last(inframe)=∞, r_info=m.data

İddia 1.4. Düğüm, halka bilgisi toplama çerçevesi aldığında, kendi bilgisini mesaja ekleyerek komşusuna iletir.

İspat: Düğüm, *state=wait* konumundayken *GATHER_RING_INFO* çerçevesi aldığında *state=search* durumuna geçer ve *inframe* zamanlayıcısını durdurur. Kendi adresini çerçeveye ekleyerek bir sonraki düğüme iletir ve halkanın tekrar işler hale gelmesini bekler. Bu durumun çalışması aşağıdaki gibidir:

state=wait,receive(GATHER_RING_INFO),recv_buf=[GATHER_RING_INFO],process_msg,recv_buf=[];state=search;last(inframe)=∞;send_buf=[GATHER_RING_INFO],send(GATHER_RING_INFO), send_buf=[]

İddia 1.5. Düğüm, halka kontrol çerçevesi aldığında halkanın işler hale geldiğine karar verir ve iççerçeve beklemeye başlar.

İspat: Düğüm, *state=search* konumundayken *RING_CHECK* çerçevesi aldığında *state=wait* konumuna geçer. Çerçeveyi bir sonraki düğüme gönderir ve *timeval(inframe)* süresince iççerçevenin gelmesini bekler. Bu durumun çalışması aşağıdaki gibidir:

state=search,receive(RING_CHECK),recv_buf[RING_CHECK],process_msg,recv_buf=[];
state=wait;last(inframe)=now+timeval(inframe);send_buf=[RING_CHECK], send(RING_CHECK),send_buf=[]

Teorem 2. *Düğüm, üyesi bulunduğu halka içerisinde meydana gelen bir çökme durumunda, çöken komşusunu tespit eder.*

Bu teoremin ispatı için aşağıdaki iddialar belirlenmiştir:

İddia 2.1. Düğüm, *timeval(inframe)* süresi içerisinde iççerçeve alamazsa halkada bir sorun olduğuna karar verir ve problemi tespit etmeye yönelik olarak komşularına birer mesaj göndererek cevap vermelerini bekler.

İspat: *v(t)* aksiyonunda *now+t>=last(inframe)* koşulu sağlandığı için *time_buf*'a *T_INFRAME* değeri eklenir. Bu işlem *process_tmo* aksiyonunun aktive olmasını sağlar. Bu durumda düğüm *state=search* konumuna geçer ve komşularına *ARE_YOU_ALIVE* mesajları gönderir ve bu mesajlara ilişkin zamanlayıcıları başlatır. Bu durumun çalışması aşağıdaki gibidir:

time_buf=[TM_INFRAME],process_tmo,time_buf=[];state=search;
last(are_you_alive_pred)=now+last(are_you_alive_pred);last(are_you_alive_succ)=now+timeval(are_you_alive_succ);send_buf=[ARE_YOU_ALIVE,ARE_YOU_ALIVE],send(ARE_YOU_ALIVE),send_buf=[ARE_YOU_ALIVE],send(ARE_YOU_ALIVE), send_buf=[]

İddia 2.2. Düğüm, komşusunun gönderdiği, yaşayıp yaşamadığını sorgulayan mesaja cevap verir.

İspat: Düğüm, eğer *ARE_YOU_ALIVE* mesajı alırsa, gönderen düğüme yaşadığını bildiren *I_AM_ALIVE* mesajı gönderir. Bu durumun çalışması aşağıdaki gibidir:

receive(ARE_YOU_ALIVE),recv_buf=[ARE_YOU_ALIVE],process_msg, recv_buf=[];send_buf=[I_AM_ALIVE],send(I_AM_ALIVE),send_buf= []

İddia 2.3. Düğüm, gönderdiği *ARE_YOU_ALIVE* mesajına cevap alırsa, ilgili komuşunun çalışir durumda olduğuna karar verir.

İspat: Düğüm, eğer komşusundan *I_AM_ALIVE* mesajı alırsa, ilgili mesaja ilişkin zamanlayıcısını durdurur. Bu durumun çalışması aşağıdaki gibidir:

Bir önceki komşusundan cevap almışsa:

receive(I_AM_ALIVE),recv_buf=[I_AM_ALIVE],process_msg,recv_buf =[]; last(are_you_alive_pred)= ∞

Bir sonraki komşusundan cevap almışsa:

receive(I_AM_ALIVE),recv_buf=[I_AM_ALIVE],process_msg,recv_buf =[]; last(are_you_alive_succ)= ∞

İddia 2.4. Düğüm, kendisinden sonra gelen komşusuna gönderdiği *ARE_YOU_ALIVE* mesajına belirli bir sürede cevap alamazsa temsilciye komşusunun yaşamadığını rapor eder.

İspat: Eğer düğümün kendisinden bir sonraki komşusundan *timeval(are_you_alive_succ)* süresi içinde cevap almazsa *v(t)* aksiyonunda *now+t>=last(are_you_alive_succ)* koşulu sağlandığı için *time_buf*'a *T_ARE_YOU_ALIVE_SUCC* değeri eklenir. Bu işlem *process_tmo* aksiyonunun aktive olmasını sağlar. Eğer bir düğümün sonraki komşusu temsilci değilse düğüm temsilciye *DEAD_NEIGHBOUR* mesajı gönderir. Bu durumun çalışması aşağıdaki gibidir:

time_buf=[ARE_YOU_ALIVE_SUCC],process_tmo,time_buf=[];send_buf= [DEAD_NEIGHBOUR],send(DEAD_NEIGHBOUR),send_buf=[]

İddia 2.5. Düğüm, kendisinden önce gelen komşusuna gönderdiği *ARE_YOU_ALIVE* mesajına belirli bir süre içerisinde cevap alamazsa temsilciye komşusunun yaşamadığını rapor eder.

İspat: Eğer düğüm bir önceki komşusundan *timeval(are_you_alive_pred)* süresi içinde cevap alamazsa *v(t)* aksiyonunda *now+t>=last(are_you_alive_pred)* koşulu sağlandığı için *time_buf*'a *T_ARE_YOU_ALIVE_PRED* değeri eklenir. Bu işlem *process_tmo* aksiyonunun aktive olmasını sağlar. Eğer kendinden bir önceki komşusu temsilci değilse düğüm temsilciye *DEAD_NEIGHBOUR* mesajı gönderir. Bu durumun çalışması aşağıdaki gibidir:

time_buf=[ARE_YOU_ALIVE_PRED],process_tmo,time_buf=[];send_buf= [DEAD_NEIGHBOUR],send(DEAD_NEIGHBOUR),send_buf=[]

Teorem 3. *Temsilcinin çökmesi durumunda o halkanın üyesi olan düğümler tarafından yeni bir temsilci seçilir.*

Bu teoremin ispatı için şu iddia belirlenmiştir:

İddia 3.1: Temsilcinin çökmesi durumunda halkanın ilk üyesi konumundaki düğüm yeni temsilciyi başlatır.

İspat: Halkanın ilk üyesi olan düğümün bir önceki komşusu temsilcidir. Eğer temsilciden *timeval(are_you_alive_pred)* süresi içerisinde bir cevap alamazsa *v(t)* aksiyonunda *now+t>=last(are_you_alive_pred)* koşulu sağlandığı için *time_buf*'a *T_ARE_YOU_ALIVE_PRED* değeri eklenir. Bu işlem *process_tmo* aksiyonunun aktive olmasını sağlar. Eğer bir önceki komşusu temsilci ise bu durumda düğüm, ölen temsilcinin bir önceki komşusunu, yani halkanın son düğümünü (adresi *r_info.reppred_addr* olan düğüm) kendinden önce gelen düğüm yapar, bu düğüme bir sonraki komşusunun kendisi olduğunu belirten *NEIGHBOUR_INFO* mesajı gönderir. Böylece bozulan halkayı tekrar oluşturur ve *role=representative* yaparak yeni temsilcinin başlamasını sağlar. Bu durumun çalışması aşağıdaki gibidir:

time_buf=[ARE_YOU_ALIVE_PRED],process_tmo,time_buf=[];r_info.pred_addr=r_info.reppred_addr;send_buf=[NEIGHBOUR_INFO];role=representative, send(NEIGHBOUR_INFO),send_buf=[]

5.2 Lider Algoritması Doğruluk Analizi

5.2.1 I/O automaton modeli

M ε {OUTFRAME, ARE_YOU_ALIVE, I_AM_ ALIVE, JOIN_REQUEST, NEIGHBOUR_INFO, RING_CHECK, GATHER_RING, DEAD_NEIGHBOUR}

T ε {T_OUTFRAME, T_ARE_YOU_ALIVE_PRED, T_ARE_YOU_ALIVE_SUCC, GENERATE_FRAME}

msg contains type ε M, data, dest_addr
ring_info contains pred_addr, succ_addr, leader_addr, leadpred_addr, leadsucc_addr
reppred(addr) is a function that finds the predecessor of a representative (specified by addr)
repsucc(addr) is a function that finds the successor of a representative (specified by addr)

Timed Signature:
Input:
$init_i$(pred,succ), pred and succ are address values
$receive_{i,j}$(m), m is msg

OutPut:
$send_{i,j}$(m), m is msg

Internal:
$process\_msg_i$(m), m is msg
$process\_tmo_i$(tm), tm ε T

Time-passage:
v(t), t ε R^+

States:
state ε {wait,search}, initially search
send_buf, a FIFO queue of elements of msg, initially empty
recv_buf, a FIFO queue of elements of msg, initially empty
time_buf, a FIFO queue of elements of T, initially empty
timeval, a vector indexed as {outframe,are_you_alive_succ, are_you_alive_pred, generate_frame}, initially set to some predefined timeout values
last, a vector indexed as {outframe,are_you_alive_succ,are_you_alive_pred,generate_frame}, initially all of them are ∞
pred_addr, address value initially null
succ_addr, address value initially null
frame_state ε {null,outframe_received,generate_frame}, initially null
rep_list, a list of addresses, initially empty

Transitions:

```
init_i(pred,succ)
   Effect:
        if pred<>null then
             m is msg
             state=search
             pred_addr=pred
             succ_addr=succ
             m.type=GATHER_RING_INFO
             m.dest_addr=succ_addr
             add m to send_buf
             last(outframe)=now+timeval(outframe)
        else
             state=wait

send_i,j(m)
   Precondition:
        m is first on send_buf
   Effect:
        remove m from send_buf

receive_i,j(m)
   Effect:
        add m to recv_buf

v(t)
   Effect:
        case
             last(outframe)<>∞ And now+t>=last(outframe):
                add T_OUTFRAME to time_buf
                last(outframe)= ∞
             last(are_you_alive_succ)<>∞ And
             now+t>=last(are_you_alive_succ):
                add T_ARE_YOU_ALIVE_SUCC to time_buf
                last(are_you_alive_succ)= ∞
             last(are_you_alive_pred)<>∞ And
now+t>=last(are_you_alive_pred):
```

```
            add T_ARE_YOU_ALIVE_PRED to time_buf
            last(are_you_alive)= ∞
        last(generate_frame)<>∞ And now+t>=last(generate_frame):
            add T_GENERATE_FRAME to time_buf
            last(generate_frame)= ∞
    now=now+t

process_msg_i(m)
  Precondition:
    m is first on recv_buf
  Effect:
    remove m from recv_buf
    case
      m.type=OUTFRAME:
        if state=wait then
            last(outframe)= ∞
            if frame_state=generate_frame then
                m.dest_addr=succ_addr
                add m to send_buf
                last(outframe)=now+timeval(outframe)
                last(generate_frame)=
                        now+timeval(generate_frame)
                frame_state=null
            else
                frame_state=outframe_received
      m.type=RING_CHECK:
        if state=search then
            last(outframe)= ∞
            state=wait
            frame_state=null
            m.type=OUTFRAME
            m.dest_addr=succ_addr
            add m to send_buf
            last(outframe)=now+timeval(outframe)
            last(generate_frame)=
                    now+timeval(generate_frame)
      m.type=GATHER_RING_INFO:
        last(outframe)= ∞
        set rep_list to m.data
```

```
        m.type=RING_CHECK
        m.dest_addr=succ_addr
        add m to send_buf
        last(outframe)=now+timeval(outframe)
    m.type=ARE_YOU_ALIVE:
        m.type=I_AM_ALIVE
        m.dest_addr=m.source_addr
        add m to send_buf
    m.type=I_AM_ALIVE:
        if m.source_addr=pred_addr then
                last(are_you_alive_pred)= ∞
        else
                last(are_you_alive_succ)= ∞
    m.type=JOIN_REQUEST:
        if state=wait then
                last(outframe)= ∞
                last(generate_frame)= ∞
                state=search
                if rep_list is not empty then
                        m.type=NEIGHBOUR_INFO
                        m.dest_addr=pred_addr
                        add m to send_buf
                        if succ_addr<>pred_addr then
                                m.type=NEIGHBOUR_INFO
                                m.dest_addr=succ_addr
                                add m to send_buf
        m.type=NEIGHBOUR_INFO
        m.dest_addr=m.source_addr
        add m to send_buf
        append m.source_addr to rep_list
        pred_addr=m.source_addr
        if succ_addr=null then
                succ_addr=m.source_addr

        m.type=RING_CHECK
        m.dest_addr=succ_addr
        add m to send_buf
        last(outframe)=now+timeval(outframe)
    m.type=DEAD_NEIGHBOUR:
```

```
            if state=wait then
                    state=search
                    last(outframe)= ∞
                    last(generate_frame)= ∞
                    if m.data=succ_addr then
                            succ_addr=repsucc(m.data)
                            m.type=NEIGHBOUR_INFO
                            m.dest_addr=succ_addr
                            add m to send_buf
                            m.type=NEIGHBOUR_INFO
                            m.dest_addr=pred_addr
                            add m to send_buf
                    else
                            m.type=NEIGHBOUR_INFO
                            m.dest_addr=reppred(m.data)
                            add msg to send_buf
                    if m.data=pred_addr then
                            pred_addr=reppred(m.data)
                            m.type=NEIGHBOUR_INFO
                            m.dest_addr=pred_addr
                            add m to send_buf
                            m.type=NEIGHBOUR_INFO
                            m.dest_addr=succ_addr
                            add m to send_buf
                    else
                            m.type=NEIGHBOUR_MSG
                            m.dest_addr=repsucc(m.data)
                            add m to send_buf
                    remove m.data from rep_list
                    m.type=RING_CHECK
                    m.dest_addr=succ_addr
                    add m to send_buf
                    last(outframe)=now+timeval(outframe)

process_tmo_i(tm)
   Precondition:
        tm is first on time_buf
   Effect:
        remove tm from time_buf
```

```
m is msg
case
    tm=T_OUTFRAME:
        state=search
        last(generate_outframe)= ∞
        m.type=ARE_YOU_ALIVE
        m.dest_addr=pred_addr
        add m to send_buf
        last(are_you_alive_pred)=
                now+timeval(are_you_alive_pred)
        if pred_addr<>succ_addr then
                m.dest_addr=succ_addr
                add m to send_buf
                last(are_you_alive_succ)=
                        now+timeval(are_you_alive_succ)
    tm=T_ARE_YOU_ALIVE_PRED:
        remove pred_addr from rep_list
        if pred_addr=succ_addr then
                pred_addr=null
                succ_addr=null
                state=wait
        else
                pred_addr=reppred(pred_addr)
                m.type=NEIGHBOUR_INFO
                m.dest_addr=pred_addr
                add m to send_buf
                m.type=NEIGHBOUR_INFO
                m.dest_addr=succ_addr
                add m to send_buf
                m.type=RING_CHECK
                m.dest_addr=succ_addr
                add m to send_buf
                last(outframe)=now+timeval(outframe)
    tm=T_ARE_YOU_ALIVE_SUCC:
        remove succ_addr from rep_list
        if pred_addr=succ_addr then
                pred_addr=null
                succ_addr=null
                state=wait
```

```
        else
                succ_addr=repsucc(succ_addr)
                m.type=NEIGHBOUR_INFO
                m.dest_addr=succ_addr
                add m to send_buf
                m.type=NEIGHBOUR_INFO
                m.dest_addr=pred_addr
                add m to send_buf
                m.type=RING_CHECK
                m.dest_addr=succ_addr
                add m to send_buf
                last(outframe)=now+timeval(outframe)
tm=T_GENERATE_FRAME:
    if state=WAIT Then
                if frame_state=outframe_received then
                        m.type=OUTFRAME
                        m.dest_addr=succ_addr
                        add m to send_buf
                        last(outframe)=now+timeval(outframe)
                        last(generate_frame)=
                                now+timeval(generate_frame)
                        frame_state=null
                else
                        frame_state=generate_frame
```

5.2.2 Doğruluk analizi

Lider modülünün işleyişi iki teorem ile açıklanmıştır. Teoremlerden ilki liderin dış halkayı yönettiği olağan durumları, ikincisi ise dış halkada oluşan bir temsilci çökmesi durumunda liderin sorunu tespit edip gidermesini açıklamaktadır. Teoremler ispatlanırken yine işleyişle ilgili iddialar belirlenmiş ve bu iddialar açıklanarak işleyişleri gösterilmiştir.

Teorem 1. *Lider, temsilcilerden oluşan dış halkayı yönetir.*

Bu teoremin ispatı için aşağıdaki iddialar belirlenmiştir:

İddia 1.1. Lider ilk başlatıldığında halka bilgisini toplar ve halkanın problemsiz çalıştığını kontrol ettikten sonra ilk dışçerçeveyi çıkarır.

Bu durumun ispatı için aşağıdaki iddiaları belirtmek gerekmektedir:

İddia 1.1.1. Lider halka toplama bilgisini çıkarır ve bu bilginin geri dönmesini bekler

İspat: Lider başladığında, init aksiyonu aktive olur. Eğer önceden kurulmuş olan bir halka varsa (*pred* ve *succ* değerleri boş değilse) *state=search* yaparak öncelikle üye bilgilerini toplamak amacıyla *GATHER_RING_INFO* çerçevesini çıkararak bu çerçevenin dönmesini bekler.

init,pred_addr=pred;succ_addr=succ;state=search;send_buf= [GATHER_RING_INFO];last(outframe)=now+timeval(outframe),send (GATHER_RING_INFO),send_buf=[]

Eğer lider başlatıldığında dış halkanın hiçbir elemanı yoksa (*pred* ve *succ* boş ise) lider *state=wait* konumuna geçerek beklemeye başlar.

init,state=wait

İddia 1.1.2. Lider halka toplama bilgisini almasının ardından halka kontrol çerçevesini çıkarır.

İspat: *GATHER_RING_INFO* mesajı geri döndükten sonra lider, üye bilgilerini *rep_list*'e kopyalar ve ardından *RING_CHECK* çerçevesini çıkararak bu çerçevenin dönmesini bekler.

receive(GATHER_RING_INFO),recv_buf=[GATHER_RING_INFO], process_msg,recv_buf=[];rep_list=m.data;send_buf=[RING_CHECK]; last(outframe)=now+timeval(outframe),send(RING_CHECK),send_buf= []

İddia 1.1.3. Lider, halka kontrol çerçevesini geri aldıktan sonra halkanın sağlıklı bir şekilde kurulduğundan emin olur ve normal işlevini başlatır.

İspat: Eğer *timeval(outframe)* süresi içerisinde *RING_CHECK* mesajı geri gelirse lider *state=wait* konumuna getirir ve ilk dışçerçeveyi çıkararak *timeval(outframe)* süresi içerisinde geri dönmesini bekler. Lider ayrıca bir sonraki çerçevenin çıkarılacağı periyodu *timeval(generate_frame)* olarak belirler.

state=search,receive(RING_CHECK),recv_buf=[RING_CHECK], process_msg,recv_buf=[];state=wait;frame_state=null;send_buf= [OUTFRAME];last(outframe)=now+timeval(outframe);last(generate_ frame)=now+timeval(generate_frame),send(outframe), send_buf=[]

İddia 1.2. Lider halkaya katılmak isteyen bir temsilciyi halkanın sonuna ekler.

İspat: Lider *state=wait* konumundayken *JOIN_REQUEST* mesajı aldığında *outframe* ve *generate_frame* zamanlayıcılarını durdurarak çerçeve iletimini durdurur ve *state=search* durumuna geçer. Eğer *rep_list* boş değil ise (halkada en az bir üye varsa) halkanın son üyesine bir sonraki komşusunun değiştiğini bildiren *NEIGHBOUR_INFO* mesajı

gönderir. Eğer halkada birden fazla üye varsa, halkanın ilk üyesine de halkanın son üyesinin değiştiğini bildiren *NEIGHBOUR_INFO* mesajı gönderir. Lider ayrıca yeni üyeye halka bilgisini *NEIGHBOUR_INFO* mesajı ile bildirir. Kendi bir önceki komşu adresini yeni gelen üye olarak değiştirir ve halkanın doğru bir şekilde kurulduğunu kontrol etmek amacıyla *RING_CHECK* çerçevesi çıkarır.

state=wait,receive(JOIN_REQUEST),recv_buf=[JOIN_REQUEST], process_msg,recv_buf[];state=search;last(outframe)=∞;last(generate_ frame)=∞;send_buf=[NEIGHBOUR_INFO,NEIGHBOUR_INFO, NEIGHBOUR_INFO];rep_list=rep_list|m.dest_addr;pred_addr= m.dest_addr;send_buf=[NEIGHBOUR_INFO,NEIGHBOUR_INFO, NEIGHBOUR_INFO,RING_CHECK];last(outframe)=now+timeval (outframe),send(NEIGHBOUR_INFO),send_buf=[NEIGHBOUR_INFO, NEIGHBOUR_INFO,RING_CHECK],send(NEIGHBOUR_INFO),send_ buf=[NEIGHBOUR_INFO,RING_CHECK],send(NEIGHBOUR_INFO), send_buf=[RING_CHECK],send(RING_CHECK),send_buf=[]

RING_CHECK çerçevesinin *timeval(outframe)* süresi içerisinde dönmesinin ardından *state=wait* konumuna geçerek dışçerçeveyi çıkarır ve halkanın normal işlevine dönmesini sağlar. Bu işleyiş İddia 1.1.3'te açıklanmıştı.

İddia 1.3. Lider, sabit bir periyodda dışçerçeve çıkarır.

Bu durumun ispatı için aşağıdaki iddiaları belirtmek gerekmektedir:

İddia 1.3.1. Lider, göndermiş olduğu dışçerçeveyi geri aldıktan sonra yeni bir çerçeve çıkarmaya hazır hale gelir.

İspat: Lider *state= wait* konumunda iken, göndermiş olduğu dışçerçeveyi *timeval(outframe)* süresi içerisinde geri aldığında, eğer bir sonraki çerçeveyi çıkarma periyodu gelmişse (*timeval(generate_frame)* süresi aşılmışsa, *frame_state = generate_frame* konumundadır) yeni dışçerveyi hemen çıkarır.

state=wait;frame_state=generate_frame,receive(OUTFRAME),recv_buf =[OUTFRAME],process_msg,[recv_buf=[];frame_state=null;send_buf =[OUTFRAME];last(outframe)=now+timeval(outframe);last(generate_ frame)=now+timeval(generate_frame), send(OUTFRAME),send_buf=[]

Eğer dışçerçeve geri alındığında bir sonraki çerçeveyi çıkarma periyodu henüz gelmemişse (*timeval(generate_frame)* süresi aşılmamışsa,*frame_state=null* konumundadır) *outframe* zamanlayıcısı durdurularak *frame_state=outframe_received* durumuna getirilerek çerçeve çıkarma periyodunun gelmesi beklenir.

state=wait,frame_state=null,receive(OUTFRAME),recv_buf= [OUTFRAME],process_msg,recv_buf=[];last(outframe)=∞;frame_state =outframe_received

İddia 1.3.2. Lider, göndermiş olduğu dışçerçeveyi geri almadan yeni bir dışçerçeve çıkarmaz.

İspat: Lider, *T_GENERATE_FRAME* zamanaşımı oluştuğunda eğer *frame_state=outframe_received* konumunda değilse *frame_state=generate_frame* konumuna geçer ve yeni bir dışçerçeve çıkarmaz.

time_buf=[GENERATE_FRAME];frame_state=null,process_tmo,time_ buf=[]; frame_state=generate_frame

İddia 1.3.3. Lider, *timeval(generate_frame)* periyodunda yeni bir outframe çıkarır.

İspat: Lider, *T_GENERATE_FRAME* zamanaşımı oluştuğunda eğer *frame_state=outframe_received* ise (bir önceki çerçeve geri gelmişse) yeni bir dışçerçeve çıkarır ve *frame_state=null* yapar. *generate_frame* zamanlayıcısını tekrar başlatarak bir sonraki dışçerçeve çıkarma periyodunu belirler.

time_buf=[GENERATE_FRAME];frame_state=outframe_received, process_tmo,time_buf=[];send_buf=[OUTFRAME];last(outframe)=now +timeval(outframe);last(generate_frame)=now+timeval(generate_ frame);frame_state=null

Teorem 2: *Lider, dış halkadaki temsilcilerden birinin çökmesi durumunda halkayı tekrar oluşturur.*

Bu teoremin ispatı için aşağıdaki iddialar belirlenmiştir:

İddia 2.1. Lider, *timeval(outframe)* süresi içerisinde dışçerçeveyi alamazsa halkada bir problem olduğunu anlar ve çökmüş olan üyeyi tespit etmek için komşularının yaşayıp yaşamadıklarını anlamak için birer mesaj gönderir.

İspat: *v(t)* aksiyonunda *now+t>=last(outframe)* koşulu sağlandığı için *time_buf*'a *T_OUTFRAME* eklenir. Bu işlem aksiyonunun aktive olmasını sağlar. Bu durumda *state=search* konumuna geçerilir ve lider komşularına *ARE_YOU_ALIVE* mesajları gönderir ve bu mesajlara ilişkin zamanlayıcıları başlatır.

time_buf=[TM_OUTFRAME],process_tmo,time_buf=[];state=search; last(are_you_alive_pred)=now+timeval(are_you_alive_pred);last(are_you_alive_succ)=now+timeval(are_you_alive_succ);send_buf=[ARE_YOU_ALIVE,ARE_YOU_ALIVE],send(ARE_YOU_ALIVE),send_buf=[ARE_YOU_ALIVE],send(ARE_YOU_ALIVE),send_buf=[]

İddia 2.2. Lider, komşusundan yaşayıp yaşamadığını sorgulayan mesaj alırsa bu mesaja cevap verir.

İspat: Lider, eğer *ARE_YOU_ALIVE* mesajı alırsa, gönderen kişiye yaşadığını bildiren *I_AM_ALIVE* mesajı gönderir.

receive(ARE_YOU_ALIVE),recv_buf=[ARE_YOU_ALIVE],process_msg, recv_buf=[];send_buf=[I_AM_ALIVE],send(I_AM_ALIVE),send_buf=[]

İddia 2.3. Lider, gönderdiği *ARE_YOU_ALIVE* mesajına cevap alırsa, ilgili komşusunun çalışmakta olduğuna karar verir.

İspat: Lider, eğer göndermiş olduğu *ARE_YOU_ALIVE* mesajına cevap alırsa, ilgili mesaja ilişkin zamanlayıcısını durdurur.

Bir önceki komşusundan cevap almışsa:

receive(I_AM_ALIVE),recv_buf=[I_AM_ALIVE],process_msg,recv_buf =[]; last(are_you_alive_pred)=∞

Bir sonraki komşusundan cevap almışsa:

receive(I_AM_ALIVE),recv_buf=[I_AM_ALIVE],process_msg,recv_buf =[]; last(are_you_alive_succ)=∞

İddia 2.4. Lider, bir sonraki komşusuna gönderdiği *ARE_YOU_ALIVE* mesajına belirli bir sürede cevap alamazsa bu komşsunun öldüğüne karar verir ve halkayı tekrar oluşturur.

İspat: Eğer bir sonraki komşusu çökmüş ise *timeval(are_you_alive_succ)* süresi içerisinde ilgili temsilciden cevap alamaycağı için *v(t)* aksiyonunda *now+t>=last(are_you_alive_succ)* koşulu sağlandığı için *time_buf*'a *T_ARE_YOU_ALIVE_SUCC* eklenir. Bu işlem *process_tmo* aksiyonunun aktive olmasını sağlar. Lider bu temsilciyi *rep_list*'ten çıkarır. Eğer *rep_list* boş ise (halkada başka bir üye yok, *succ_addr=pred_addr*) *succ_addr=pred_addr=null* yapar ve *state=wait* konumuna geçerek bekleme durumuna geçer.

time_buf=[are_you_alive_succ];succ_addr=pred_addr,process_tmo,
time_buf=[];rep_list=[];pred_addr=null;succ_addr=null;state=wait

Eğer *rep_list* boş değilse (halkada en az bir üye varsa) halkayı tekrar kurabilmek için öncelikle bir sonraki komşusunu ölen üyenin bir sonraki komşusuna eşitler ve bu temsilciye bir önceki komşusunun değiştiğinibildiren *NEIGHBOUR_INFO* mesajı gönderir. Ardından halkanın doğru bir şekilde kurulduğundan emin olmak için *RING_CHECK* çerçevesini çıkarır.

time_buf=[are_you_alive_succ];succ_addr<>pred_addr,process_tmo,
time_buf=[]; rep_list=rep_list-succ_addr;succ_addr=repsucc
(succ_addr);send_buf=[NEIGHBOUR_INFO,NEIGHBOUR_INFO,
RING_ CHECK];last(outframe)=now+timeval(outframe),send
(NEIGHBOUR_INFO),send_buf=[NEIGHBOUR_INFO,RING_CHECK]
*,send(NEIGHBOUR_INFO),send_buf=[RING_CHECK],send(RING_*
CHECK),send_buf=[]

İddia 2.5. Lider, bir önceki komşusuna gönderdiği *ARE_YOU_ALIVE* mesajına belirli bir sürede cevap alamazsa bu komşusunun öldüğüne karar ve halkayı tekrar oluşturur.

İspat: Eğer bir önceki komşsu çökmüş ise timval(are_you_alive_pred) süresi içerisinde ilgili temsilciden cevap alamayacağı için *v(t)* aaksiyonunda *now+t>=last(are_you_alive_pred)* koşulu sağlandığı için *time_buf*'a *T_ARE_YOU_ALIVE_PRED* eklenir. Bu işlem *process_tmo*'nun aktive olmasını sağlar. Lider bir önceki komşusunu *rep_list*'ten çıkarır. Eğer *rep_list* boş ise (halkada başka bir üye yok, *succ_addr=pred_addr*) *succ_addr=pred_addr=null* yapar ve *state=wait* konumuna geçirerek bekleme durumuna geçer.

time_buf=[are_you_alive_pred];succ_addr=pred_addr,process_tmo,
time_buf=[];rep_list=[];pred_addr=null;succ_addr=null;state=wait

Eğer *rep_list* boş değilse (halkada en az bir üye varsa) halkayı tekrar kurabilmek için öncelikle bir önceki komşusunu ölen üyenin bir önceki komşusuna eşitler ve bu temsilciye bir sonraki komşusunun değiştiğini bildiren *NEIGHBOUR_INFO* mesajı gönderir. Ardından halkanın doğru bir şekilde kurulduğundan emin olmak için *RING_CHECK* çerçevesi çıkarır.

time_buf=[are_you_alive_pred];succ_addr<>pred_addr,process_tmo,
*time_buf=[]; rep_list=rep_list-pred_addr;pred_addr=reppred (pred_*
*addr);send_buf=[NEIGHBOUR_INFO,NEIGHBOUR_INFO,RING_*
*CHECK];last(outframe)=now+timeval(outframe),send(NEIGHBOUR_*
INFO),send_buf=[NEIGHBOUR_INFO,RING_CHECK],send
(NEIGHBOUR_INFO),send_buf=[RING_CHECK],send
(RING_CHECK),send_buf=[]

İddia 2.6. Lider, herhangi bir temsilciden halkadaki bir üyenin öldüğüne dair mesaj alırsa ilgili temsilciyi halkadan çıkarır.

İspat: Lider *state=wait* konumunda iken *DEAD_NEIGHBOUR* mesajı aldığında, öncelikle *state=search* yapar ve *outframe* ile *generate_frame* zamanlayıcılarını durdurur. Böylece halkanın normal işlevini durdurmuş olur. Eğer ölü olarak bildirilen temsilci halkanın ilk üyesiyse **İddia 2.4**'te açıklanan işlemlere benzer şekilde bu üyeyi halkadan çıkarır ve ardından *RING_CHECK* çerçevesi çıkarır.

state=wait,receive(DEAD_NEIGHBOUR),recv_buf=[DEAD_ NEIGHBOUR],process_msg, recv_buf=[];rep_list=rep_list-succ_addr; succ _addr = repsucc (succ_addr); send_buf=NEIGHBOUR_INFO, NEIGHBOUR_INFO,RING_CHECK];last(outframe)=now+timeval (outframe),send(NEIGHBOUR_INFO),send_buf=[NEIGHBOUR_INFO, RING_CHECK],send(NEIGHBOUR_INFO),send_buf=[RING_CHECK] , send(RING_CHECK),send_buf=[]

Eğer ölü olarak bildirilen temsilci halkanın son üyesiyse İddia 2.5'te açıklanan işlemlere benzer şekilde bu temsilciyi halkadan çıkarır ve ardından *RING_CHECK* çerçevesi çıkarır.

state=wait,receive(DEAD_NEIGHBOUR),recv_buf=[DEAD_ NEIGHBOUR],process_msg, recv_buf=[];rep_list=rep_list-pred_addr; pred_addr=reppred(pred_addr);send_buf=[NEIGHBOUR_INFO, NEIGHBOUR_INFO,RING_CHECK];last(outframe)=now+timeval (outframe),send(NEIGHBOUR_INFO),send_buf=[NEIGHBOUR_INFO, RING_CHECK],send(NEIGHBOUR_INFO),send_buf=[RING_CHECK] , send(RING_CHECK),send_buf=[]

Eğer ölü olarak bildirilen üye liderin komşusu değilse, öncelikle *rep_list*'ten silinir. Ölen üyenin komşularına değişen halka bilgileri

NEIGHBOUR_INFO mesajı ile gönderilerek halkanın tekrar kurulması sağlanır ve ardından *RING_CHECK* çerçevesi çıkarılır.

state=wait,receive(DEAD_NEIGHBOUR),recv_buf=[DEAD_ NEIGHBOUR],process_msg, recv_buf=[];rep_list=rep_list-m.data; send_buf=[NEIGHBOUR_INFO,NEIGHBOUR_INFO,RING_CHECK]; last(outframe)=now+timeval(outframe),send(NEIGHBOUR_INFO), send_buf=[NEIGHBOUR_INFO,RING_CHECK],send(NEIGHBOUR_ INFO),send_buf= [RING_CHECK],send(RING_CHECK),send_buf=[]

5.3 Dağıtık Yük Dengeleme Modülleri Doğruluk Analizleri

5.3.1 Düğüm algoritması doğruluk analizi

5.3.1.1 I/O automaton modeli

M ε
{INFRAME,LOAD_TRANSFER_REQUEST,LOAD_TRANSFER_DE STINATION,
LOAD_TRANSFER_REJECT,LOAD_TRANSFER_COMPLETE, TERMINATE_PROCESS}

msg contains type ε M, source_addr, dest_addr, pno,rem_pno,pname, pload, presult,host_addr
process_table contains pname,pno,ptype,pload,host_addr,rem_pno

function find_empty_slot(proc_table) finds an ampty slot in proc_table
function run_process(pno) runs the process specified by pno

Timed Signature:
Input:
receive$_{i,j}$(m), m is msg
process_start(pname,pload), pname is string, pload is integer

process_terminate(presult), presult is string

OutPut:

$send_{i,j}$(m), m is msg

process_end(presult), presult is string

Internal:

$process\_msg_i$(m), m is msg

$process\_tmo_i$(pno), pno is integer

Time-passage:

v(t), $t \varepsilon R^+$

States:

rep_addr, address value, initially set to the representative address

send_buf, a FIFO queue of elements of msg, initially empty

recv_buf, a FIFO queue of elements of msg, initially empty

time_buf, a FIFO queue of elements of integer, initially empty

res_buf, a FIFO queue of elements of string

proc_table, a vector of process_table indexed by 1 to n, initially all null

last, a vector of integer indexed by 1 to n, initially all of them are ∞

Transitions:

$send_{i,j}$(m)

Precondition:

m is first on send_buf

Effect:

remove m from send_buf

$receive_{i,j}$(m)

Effect:

add m to recv_buf

v(t)

Effect:

for i=1 to n

if last(i)<>∞ And now+t>=last(i):

```
                add i to time_buf
                last(i)= ∞
        now=now+t

process_msg_i(m)
    Precondition:
        m is first on recv_buf
    Effect:
        remove m from recv_buf
        case
            m=INFRAME:
                m.data=m.data || load
                add m to send_buf
            m=LOAD_TRANSFER_REQUEST:
                if load<HIGH then
                        pno=find_empty_slot(proc_table)
                        proc_table(pno).pname=m.pname
                        proc_table(pno).rem_pno=m.pno
                        proc_table(pno).pload=m.pload
                        proc_table(pno).ptype=TRANSFERRED
                        proc_table(pno).host_addr=m.source_addr
                        load=load+m.pload
                        run_process(pno)
                        m.type=LOAD_TRANSFER_COMPLETE
                        m.rem_pno=pno
                        m.dest_addr=m.source_addr
                        add m to send_buf
                Else
                        m.type=LOAD_TRANSFER_REJECT
                        m.dest_addr=m.source_addr
                        add m to send_buf
            m=LOAD_TRANSFER_DESTINATION:
                        last(m.pno)=∞
                        proc_table(m.pno).host_addr=m.host_addr
                        m.type=LOAD_TRANSFER_REQUEST
                        m.pno=m.pno
                        m.pload=proc_table(m.pno).pload
                        m.pname= proc_table(m.pno).pname
                        m.dest_addr=m.host_addr
```

```
                        add m to send_buf
                        last(m.pno)=now+TIMEVAL
            m=LOAD_TRANSFER_REJECT:
                        last(m.pno)=∞
                        proc_table(m.pno).ptype=LOCAL
                        load=load+proc_table(m.pno).pload

                        run_process(m.pno)
            m=LOAD_TRANSFER_COMPLETE:
                        last(m.pno)=∞
                        proc_table(m.pno).rem_pno=m.rem_pno
            m=TERMINATE_PROCESS:
                        add m.presult to res_buf
                        proc_table(m.pno)=null

process_tmo_i(pno)
    Precondition:
        pno is first on time_buf
    Effect:
        remove pno from time_buf

        proc_table(pno).ptype=LOCAL
        load=load+proc_table(pno).pload
        run_process(pno)

process_start(pname,pload)
    Effect:
        pno=find_emty_slot(proc_table)
        proc_table(pno).pname=pname
        proc_table(pno).pload=pload

        if load<HIGH then
            proc_table(pno).ptype=LOCAL
            load=load+pload
            run_process(pno)
        Else
            proc_table(pno).ptype=REMOTE
            m.type=LOAD_TRANSFER_REQUEST
            m.pno=pno
```

```
            m.pname=pname
            m.pload=pload
            m.dest_addr=rep_addr
            add m to send_buf
            last(pno)=now+TIMEVAL

process_terminate(pno,presult)
    Effect:
        if proc_table(pno).ptype=LOCAL then
            add presult to res_buf
        Else
            m.type=TERMINATE_PROCESS
            m.pno=proc_table(pno).rem_pno
            m.presult=presult
            m.dest_addr=m.host_addr
            add m to send_buf

        load=load-proc_table(pno).pload
        proc_table(pno)=null

process_end(presult)
    Precondition:
        presult is first on res_buf
    Effect:
        remove presult from res_buf
```

5.3.1.2 Doğruluk analizi

Teorem 1. *Düğüm, üst birimden iletilen, çalıştırılmak istenen bir işlemi ya kendi üzerinde çalıştırır ya da uygun olan başka bir düğüme transfer eder ve çalışma sonucunu her iki durumda da üst birime geri döndürür.*

Bu teoremin ispatı için aşağıdaki iddialar belirlenmiştir:

İddia 1.1. Düğüm, yük durumunu belirli zamanlarda aldığı iççerçeveye koyarak temsilciye rapor eder.

İspat: Düğüm, iççerçeveyi aldığında kendi yük durumunu çerçevenin bilgi kısmına ekler ve çerçeveyi bırakır.

receive(INFRAME),recv_buf=[INFRAME],process_msg(INFRAME), recv_buf=[];send_buf=[INFRAME],send(INFRAME),send_buf=[]

İddia 1.2. Düğüm, üst birimden yapılan işlem başlatma isteklerine cevap verir.

Bu durumun ispatı için aşağıdaki iddiaları belirtmek gerekmektedir:

İddia 1.2.1. Düğüm, eğer yük durumu yüksek eşik değerinin altındaysa işlemi kendi üzerinde başlatır.

İspat: İşlem başlatılmak istendiğinde düğüm, öncelikle *load* değerini kontrol eder. *load<HIGH* koşulu sağlanıyorsa işlemi kendi üzerinde başlatır ve *load* değerini arttırır.

load<HIGH,process_start(pname,pload),load=load+pload

İddia 1.2.2. Eğer yük durumu yüksek eşik değerinin üzerinde ise düğüm temsilciye yük transfer isteğinde bulunur.

İspat: İşlem başlatılmak istendiğinde eğer *load<HIGH* koşulu sağlanmıyorsa düğüm temsilciye *LOAD_TRANSFER_REQUEST* mesajı gönderir ve ilgili zamanlayıcıyı başlatır.

load>=HIGH,process_start(pname,pload),last(pno)=now+TIMEVAL; send_buf=[LOAD_TRANSFER_REQUEST],send(LOAD_TRANSFER_ REQUEST),send_buf=[]

İddia 1.2.3. Eğer yük transer isteğinde bulunan düğüm, yük transferi yapabileceği bir düğüm adresi geri alırsa ilgili düğüme transfer isteğinde bulunur.

İspat: Eğer düğüm, *LOAD_TRANSFER_DESTINATION* mesajı alırsa, mesaj ile kendisine bildirilen düğüme *LOAD_TRANSFER_REQUEST* mesajı göndererek yük transfer isteğinde bulunur ve ilgili zamanlayıcıyı başlatır.

receive(LOAD_TRANSFER_DESTINATION),recv_buf=[LOAD_ TRANSFER_DESTINATION],process_msg(LOAD_TRANSFER_ DESTINATION],recv_buf=[];last(m.pno)=now+TIMEVAL;send_buf= [LOAD_TRANSFER_REQUEST],send(LOAD_TRANSFER_REQUEST), send_buf=[]

İddia 1.2.4. Eğer düğümün yaptığı transfer isteği reddedilirse düğüm işlemi kendi üzerinde başlatır.

İspat: Eğer düğüm, *LOAD_TRANSFER_REJECT* mesajı alırsa, yük transferi yapabileceği bir başka düğüm bulunamadığından işlemi kendi üzerinde başlatır ve *load* değerini arttırır

receive(LOAD_TRANSFER_REJECT),recv_buf=[LOAD_TRANSFER_ REJECT],process_msg(LOAD_TRANSFER_REJECT],recv_buf=[];last (m.pno)=∞;load=load+ proc_table(m.pno).pload

İddia 1.2.5. Eğer transfer isteği zaman aşımına uğrarsa düğüm işlemi kendi üzerinde başlatır.

İspat: Yaptığı yük transfer isteğine *TIMEVAL* değeri kadar bir süre içinde cevap alamazsa zamanaşımı mekanizması işleyen düğüm işlemi kendi üzerinde başlatır ve yük değerini günceller.

time_buf=[pno],process_tmo(pno),time_buf=[];load=load+proc_table (pno).pload

İddia 1.2.6. Düğüm, üzerinde çalıştırdığı işlem sonlanınca sonucu üst birime geri döndürür.

İspat: İşlem sonlandığında oluşan *process_terminate* aksiyonunda *ptype=LOCAL* koşulu sağlanarak kendi üzerinde çalıştırmış olduğu işleme ilişkin sonuç bilgisini üst birime *process_end* aksiyonu ile aktarır ve *load* değerini düşürür.

proc_table(pno).ptype=LOCAL,process_terminate(pno,presult),load= load-proc_table(pno).pload;proc_table(pno)=null;res_buf=[presult], process_end(presult),res_buf=[]

İddia 1.2.7. Düğüm, başka bir düğüme transfer etmiş olduğu işleme ilişkin sonucu, transfer ettiği düğümden aldığında sonucu üst birime geri döndürür.

İspat: Eğer transfer etmiş olduğu bir işleme ilişkin *TERMINATE_PROCESS* mesajı alırsa düğüm, mesaj ile gelen sonuç bilgisini üst birime geri döndürür.

receive(TERMINATE_PROCESS),recv_buf=[TERMINATE_PROCESS], process_msg(TERMINATE_PROCESS),recv_buf=[];proc_table(m.pno) =null;res_buf=[m.presult],process_end(m.presult),res_buf=[]

Teorem 2. *Düğüm, aldığı yük transfer isteklerine o anki yük değerine göre olumlu ya da olumsuz cevap verir, cevabı olumlu ise transfer edilen işlemi çalıştırır ve sonucunu isteği yapan düğüme geri döndürür.*

İddia 2.1. Düğüm, eğer yük durumu yüksek eşik değerinin altındaysa transfer edilmek istenen işlemi başlatır ve isteği yapan düğüme bildirimde bulunur.

İspat: *LOAD_TRANSFER_REQUEST* mesajı aldığında *load<HIGH* koşulu sağlandığı taktirde işlemi kendi üzerinde başlatır, *load* değerini arttırır ve isteği yapan düğüme *LOAD_TRANSFER_COMPLETE* mesajı gönderir.

load<HIGH,receive(LOAD_TRANSFER_REQUEST),recv_buf= [LOAD_TRANSFER_REQUEST],process_msg(LOAD_TRANSFER_ REQUEST),recv_buf=[];load=load+m.pload;sendbuf=[LOAD_ TRANSFER_DESTINATION],send(LOAD_TRANSFER_DESTINATON); send_buf=[]

İddia 2.2. Eğer o anki yük durumu yüksek eşik değerinin üzerindeyse isteği yapan düğüme isteği reddettiğini bildirir.

İspat: *LOAD_TRANSFER_REQUEST* mesajı aldığında *load<HIGH* değeri sağlanmadığı taktirde isteği yapan düğüme *LOAD_TRANSFER_REJECT* mesajı gönderir.

load>=HIGH,receive(LOAD_TRANSFER_REQUEST),recv_buf= [LOAD_TRANSFER_REQUEST],process_msg(LOAD_TRANSFER_ REQUEST),recv_buf=[];sendbuf=[LOAD_TRANSFER_REJECT],send (LOAD_TRANSFER_REJECT);send_buf=[]

İddia 2.3. Düğüm, üzerinde çalışan ve transfer edilmiş olan işlem sonlanınca sonucu transferi yapan düğüme geri döndürür.

İspat: Başka bir düğümden transfer edilmiş olan işleme ilişkin *process_terminate* aksiyonu oluştuğu taktirde sonlanan işleme ait *ptype<>LOCAL* koşulu sağlandığında işlemin sonucunu transferi yapan düğüme *PROCESS_TERMINATE* mesajı ile gönderir ve *load* değerini düşürür.

proc_table(pno)<>LOCAL,process_terminate(pno,pload),send_buf= [PROCESS_TERMINATE],send(PROCESS_TERMINATE),send_buf=[]

5.3.2 Temsilci modülü doğruluk analizi

5.3.2.1 I/O automaton modeli

M ε {INFRAME,OUTFRAME,LOAD_TRANSFER_REQUEST}

msg contains type ε M, source_addr, dest_addr, pno,rem_pno,pname, pload, presult,host_addr

fill_tables(load_data) fills the low_table, medium_table and high_table according to load values of nodes in load_data

Timed Signature:
Input:
receive$_{i,j}$(m), m is msg

Output:
send$_{i,j}$(m), m is msg

Internal:
process_msg$_i$(m), m is msg

States:

leader_addr, address value, initially set to the leader address
send_buf, a FIFO queue of elements of msg, initially empty
recv_buf, a FIFO queue of elements of msg, initially empty
low_table, a FIFO queue of address elements initially empty
medium_table, a FIFO queue of address elements initially empty
high_table, a FIFO queue of address elements initially empty

Transitions:

$send_{i,j}(m)$
Precondition:
m is first on send_buf
Effect:
remove m from send_buf

$receive_{i,j}(m)$
Effect:
add m to recv_buf

$process\_msg_i(m)$
m is first on recv_buf
Effect:
remove m from recv_buf
case
m=INFRAME:
fill_tables(m.data)
m=OUTFRAME:
m.data=m.data || low_table || medium_table || high_table
add m to send_buf
m=LOAD_TRANSFER_REQUEST:
if low_table is not empty then
m.type=LOAD_TRANSFER_DESTINATION
m.host_addr=first element of low_table
m.dest_addr=m.source_addr
add m to send_buf
else if medium_table is not empty then
m.type=LOAD_TRANSFER_DESTINATION
m.host_addr=first element of medium_table
m.dest_addr=m.source_addr

```
                add m to send_buf
        else
                m.type=LOAD_TRANSFER_REQUEST
                m.dest_addr=leader_addr
                add m to send_buf
```

5.3.2.2 Doğruluk analizi

Teorem 1. *Temsilci, kendi halkası içerisinde topladığı yük değerlerine göre, gelen yük transfer isteklerine cevap verir ve topladığı yük değerlerini lidere bildirir.*

Bu teoremin ispatı için aşağıdaki iddialar belirlenmiştir:

İddia 1.1. Temsilci, sorumlu olduğu iç halkadaki üyelerin yük durumunu, periyodik olarak çıkardığı iççerve ile alarak tablolarını günceller.

İspat: İç halkada dolaşan iççerçeveyi aldıığında, çerçeve içerisindeki düğümlere ilişkin yük değerlerine göre *low_table*, *medium_table* ve *high_table*'ı oluşturur

receive(INFRAME),recv_buf=[INFRAME],process_msg(INFRAME), recv_buf=[]; fill_tables(m.data)

İddia 1.2 Temsilci, sorumlu olduğu iç halkadaki üyelerin yük durumunu belirli zamanlarda gelen dışçerçeveye koyarak lidere bildirir.

İspat: Dış halkada dolaşan dışçerçeveyi aldığında *low_table*,*medium_table* ve *high_table*'daki bilgileri çerçeveye kopyalayar ve çerçeveyi bırakır.

receive(OUTFRAME),recv_buf=[OUTFRAME],process_msg (OUTFRAME),recv_buf=[];send_buf=[OUTFRAME],send (OUTFRAME),send_buf=[]

İddia 1.3. Yük transfer isteği aldığında kendi halkası üzerinde uygun bir düğüm bulursa düğüm adresini isteği yapan düğüme geri döndürür.

İspat: *LOAD_TRANSFER_REQUEST* mesajı aldığında *low_table*'daki ilk düğümün adresini, eğer *low_table* boş ise, *medium_table*'daki ilk düğümün adresini isteği yapan düğüme *LOAD_TRANSFER_DESTINATION* mesajı ile bildirir.

low_table or medium_table not empty,receive(LOAD_TRANSFER_ REQUEST), recv_buf=[LOAD_TRANSFER_REQUEST],process_msg (LOAD_TRANSFER_REQUEST),recv_buf=[];send_buf=[LOAD_ TRANSFER_DESTINATION],send(LOAD_TRANSFER_DESTINATION) ,send_buf=[]

İddia 1.4. Yük transfer isteği aldığında kendi halkası üzerinde uygun bir düğüm bulamazsa isteği lidere aktarır.

İspat: *LOAD_TRANSFER_REQUEST* mesajı aldığında eğer *low_table* ve *medium_table* boş ise kendi halkası içerisinde uygun bir düğüm bulamayarak gelen *LOAD_TRANSFER_REQUEST* mesajını lidere gönderir.

low_table and medium_table empty,receive(LOAD_TRANSFER_ REQUEST), recv_buf=[LOAD_TRANSFER_REQUEST],process_msg (LOAD_TRANSFER_REQUEST),recv_buf=[];send_buf=[LOAD_ TRANSFER_REQUEST],send(LOAD_TRANSFER_REQUEST),send_buf =[]

5.3.3 Lider modülü doğruluk analizi

5.3.3.1 I/O automaton modeli

M ε {OUTFRAME,LOAD_TRANSFER_REQUEST}

msg contains type ε M, source_addr, dest_addr, pno,rem_pno,pname, pload, presult,host_addr

fill_tables(load_data) fills the low_table, medium_table and high_table according to load values of nodes in load_data

Timed Signature:

Input:
$receive_{i,j}(m)$, m is msg

Output:
$send_{i,j}(m)$, m is msg

Internal:
$process\_msg_i(m)$, m is msg

States:
send_buf, a FIFO queue of elements of msg, initially empty
recv_buf, a FIFO queue of elements of msg, initially empty
low_table, a FIFO queue of address elements initially empty
medium_table, a FIFO queue of address elements initially empty
high_table, a FIFO queue of address elements initially empty

Transitions:
$send_{i,j}(m)$
Precondition:
m is first on send_buf
Effect:
remove m from send_buf

```
receive_{i,j}(m)
    Effect:
        add m to recv_buf

process_msg_i(m)
    Precondition:
        m is first on recv_buf
    Effect:
        remove m from recv_buf
    case
        m=OUTFRAME:
            fill_tables(m.data)
        m=LOAD_TRANSFER_REQUEST:
            if low_table is not empty then
                    m.type=LOAD_TRANSFER_DESTINATION
                    m.host_addr=first element of low_table
                    m.dest_addr=m.source_addr
                    add m to send_buf
            else if medium_table is not empty then
                    m.type=LOAD_TRANSFER_DESTINATION
                    m.host_addr=first element of medium_table
                    m.dest_addr=m.source_addr
                    add m to send_buf
            else
                    m.type=LOAD_TRANSFER_REJECT
                    m.dest_addr=m.source_addr
                    add m to send_buf
```

5.3.3.2 Doğruluk analizi

Teorem 1. *Lider, sistemden topladığı yük değerlerine göre, gelen yük transfer isteklerine cevap verir.*

Bu teoremin ispatı için aşağıdaki iddialar belirlenmiştir:

İddia 1.1. Lider, sistemdeki tüm düğümlerin yük durumlarını, periyodik olarak çıkardığı dışçerçeveden alarak tablolarını günceller.

İspat: Lider, halkada dolaşan dışçerçeveyi aldığında, çerçenin temsilciler tarafından doldurulan bilgi bölümünde yer alan, sistemdeki tüm düğümlere ilişkin yük değerlerine göre *low_table*, *medium_table* ve *high_table*'ı oluşturur

receive(OUTFRAME),recv_buf=[OUTFRAME],process_msg (OUTFRAME),recv_buf=[]; fill_tables(m.data)

İddia 1.2. Lider, bir yük transfer isteği mesajı aldığında kendi uygun bir node bulursa node adresini isteği yapan node'a geri döndürür.

İspat: LOAD_TRANSFER_REQUEST mesajı aldığında low_table'daki ilk node'un adresini, eğer low_table boş ise, medium_table'daki ilk node'un adresini isteği yapan node'a LOAD_TRANSFER_DESTINATION mesajı ile gönderir.

low_table not empty or medium_table not empty,receive(LOAD_ TRANSFER_REQUEST),recv_buf=[LOAD_TRANSFER_REQUEST], process_msg(LOAD_TRANSFER_REQUEST),recv_buf=[];send_buf= [LOAD_TRANSFER_DESTINATION],send(LOAD_TRANSFER_ DESTINATION),send_buf=[]

İddia 1.3. Yük transfer isteği aldığında uygun bir node bulamazsa isteği yapan node'a transfer isteğinin reddedildiğini bildirir.

İspat: LOAD_TRANSFER_REQUEST mesajı aldığında eğer low_table ve medium_table boş ise sistemde uygun bir node bulamayarak isteği yapan node'a LOAD_TRANSFER_REJECT mesajı döndürür.

low_table is empty and medium_table is empty,receive(LOAD_TRANSFER_REQUEST),

recv_buf=[LOAD_TRANSFER_REQUEST],process_msg(LOAD_TRANSFER_REQUEST),recv_buf=[];send_buf=[LOAD_TRANSFER_REJECT],send(LOAD_TRANSFER_REJECT), send_buf=[]

6 UYGULAMA

Bu bölümde, bölüm 3 ve 4'te açıklanan, hiyerarşik, halka tabanlı gerçek zamanlı dağıtık sistem altyapısının ve bu altyapı üzerine geliştirilen dağıtık işlem izlencelemesi modelinin gerçekleştirilen uygulamasından söz edilecektir.

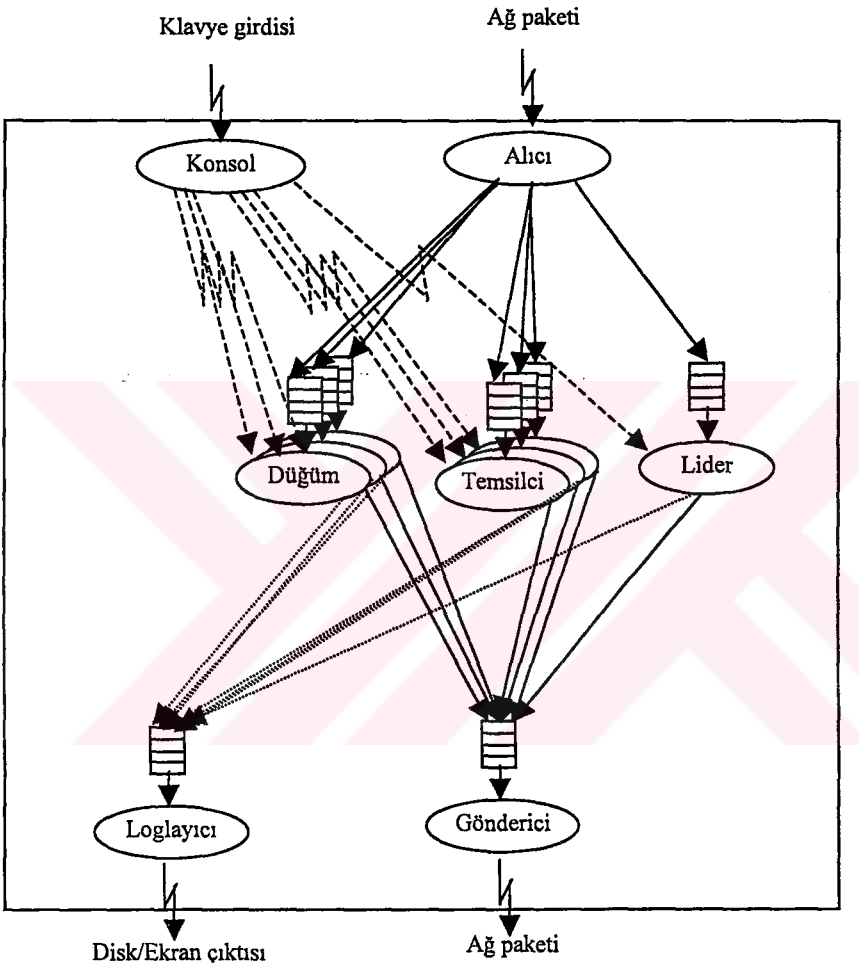
6.1 Dağıtık Sistem Modelinin Uygulaması

6.1.1 Sistemin yapısı ve işleyişi

Tezin uygulama aşamasında ilk olarak, hiyerarşik halka tabanlı protokol yapısının, UNIX tabanlı makineler üzerinde çalışan bir uygulaması geliştirilmiştir. Bunun için, sistemi oluşturan üç temel yapı olan Düğüm, Temsilci ve Lider modülleri birbirinden bağımsız olarak paralel çalışan işlem birimleri olarak tasarlanmışlardır. Bu tezde düğüm ve lider modüllerinin uygulaması gerçekleştirilmiştir. Temsilci modülünün uygulaması ise aynı altyapıyı kullanarak grup yönetimi modelini gerçekleştiren bir başka tezin kapsamında yer almaktadır. Modüllerin ayrı ayrı gerçekleştirilmesinden sonra bir araya getirilerek, sistemin bir bütün olarak işlerliği sağlanmıştır.

Geliştirilen sistemin genel yapısı şu şekildedir: Sistemdeki her bir iş istasyonunda bir çok parçacıklı (multithreaded) süreç (process) çalışmaktadır. Bu süreç, paralel olarak çalışan iş parçacıklarından (thread) oluşmaktadır. Sistem temelde düğüm, temsilci, lider, alıcı, gönderici, konsol ve loglayıcı iş parçacıklarından oluşmaktadır. Bu iş parçacıkları arasındaki iletişim, paylaşımlı bellek üzerinde geliştirilmiş

ilk giren ilk çıkar mantığıyla çalışan kuyruk (FIFO queue) yapılarıyla sağlanmaktadır. Bir sürecin yapısı şekil 6.1'de görülmektedir.



Şekil 6.1. Bir sistem sürecinin yapısı

Her bir iş parçacığının görevi ve işleyişi şu şekildedir:

Düğüm: Sistemdeki düğüm modülü işlevini gerçekleştirir. Girdi olarak kendi mesaj kuyruğundaki mesajları alır ve ilgili işlevi gerçekleştirir. Ayrıca zamanaşımlarını da işlemektedir. Başka modüllere göndereceği mesajları ise gönderici iş parçacığının kuyruğuna yerleştirir. Bu birimin işleyişi bölüm 3'te ayrıntılı olarak açıklanmıştır. Simulasyon işlemleri için sistem bir makinede birden fazla düğüm parçacığı çalışabilecek şekilde tasarlanmıştır.

Temsilci: Sistemdeki temsilci modülü işlevini gerçekleştirir. Girdi olarak kendi mesaj kuyruğundaki mesajları alır ve ilgili işlevi gerçekleştirir. Ayrıca zamanaşımlarını da işlemektedir. Başka modüllere göndereceği mesajları ise gönderici iş parçacığının kuyruğuna yerleştirir. Simulasyon işlemleri için sistem bir makinede birden fazla temsilci parçacığı çalışabilecek şekilde tasarlanmıştır.

Lider: Sistemdeki lider modülü işlevini gerçekleştirir. Girdi olarak kendi mesaj kuyruğundaki mesajlar ve oluşan zamanaşımlarına göre ilgili işlevi gerçekleştirir. Başka modüllere göndereceği mesajları ise gönderici iş parçacığının kuyruğuna yerleştirir. Bu birimin işleyişi bölüm 3'te ayrıntılı olarak açıklanmıştır.

Alıcı: Ağ üzerinden gelen mesajları karşılayan ve o makine üzerinde çalışan ilgili iş parçacığına (bir düğüme, temsilciye ya da lidere) iletimini gerçekleştiren ağ arayüzü birimdir. Girdi olarak ağ üzerinden bir paket alır ve aldığı paketin başlık bilgilerini okuyarak gönderildiği iş parçacığını tespit ederek mesajı ilgili iş parçacığının kuyruğuna bırakır.

Gönderici: Düğüm, temsilci ve lider modülleri arası mesaj iletiminden sorumludur. Girdi olarak kendi kuyruğundan gönderici modülün bırakmış olduğu mesajı alır, mesajın başlık bilgisini okuyarak

eğer o makinedeki bir başka modüle gönderilmek isteniyorsa ilgili iş parçacığının kuyruğuna bırakır. Eğer gönderilmek istenen modül başka bir makinede ise mesajı ağ paketine yerleştirerek ilgili makineye gönderir.

Konsol: Geliştirilen sistemin yönetim arabirimidir. Operatör tarafından klavyeden girilen komutları yorumlayarak ilgili işlevi gerçekleştirir. Sisteme yeni düğüm, temsilci ya da lider ekleme, çökme senaryoları simulasyonu için düğüm, temsilci ya da lider silme gibi operasyonları gerçekleştirir.

Loglayıcı: Sistemin çalışmasının izlenmesine yönelik bilgileri çıktılar. Düğüm, temsilci ve lider modüllerinin belirli adım ve aşamalarda gönderdiği bilgileri kuyruğundan okuyarak ekrana ya da diske çıktılar.

Sistemdeki bir sürecin genel işleyişi şu şekildedir: Süreç bir bilgisayarda ilk başlatıldığında alıcı, gönderici, konsol ve loglayıcı iş parçacıkları otomatik başlatılır ve bekleme konumuna geçerler. Konsol parçacığı, kullanıcıdan (klavyeden) aldığı komutlar doğrultusunda ilgili modül parçacıklarını (düğüm, temsilci ya da lider) başlatır. Bu modül parçacıkları ilk başlatılma süreçlerini yerine getirdikten sonra bir olay bekleme konumuna geçerler. Kendi kuyruklarında mesaj bulunduğunda ya da başlattıkları bir zamanlayıcı zamanaşımına uğradığında bir olay meydana gelir ve modül olayla ilgili işlemi gerçekleştirir. Her modül, kuyruğunda mesaj olup olmadığını kontrol eden birer iş parçacığı daha çalıştırmaktadır. Modül bir başka modüle yapmak istediği mesaj gönderimini, mesajı göndericinin kuyruğuna koyarak gerçekleştirir. Bundan sonra mesajın ilgili birime iletiminden gönderici sorumludur. Sürece ağ üzerinden iletilen mesajların alınması ve ilgili modülün kuyruğuna konması işlemini de alıcı gerçekleştirmektedir.

UNIX ortamında geliştirilen bu sistemde iş parçacakılarının programlaması "POSIX Threads" uygulama geliştirme arayüzü kullanılarak gerçekleştirilmiştir.

Sistemin zamanlayıcı mekanizması süreç bazında çalışan UNIX "SIGALRM" sistem kesmesi kullanılarak gerçekleştirilmiştir. Sistemdeki iş parçacıklarının birden fazla ve bağımsız zamanlayıcılar kullanabilmesi için bir genel zamanlayıcı tablosu oluşturulmuştur. Zamanlayıcı kullanmak isteyen bir iş parçacığına kullanmak istediği her bir zamanlayıcı için bu tabloda bir saha ayrılmaktadır. Bu sahaya iş parçacağı tarafından bildirilen, ilgili zamanlayıcının milisaniye bazındaki çalışma süresi ile zamanaşımı oluştuğunda tetikleyeceği fonksiyonun adresi yerleştirilmektedir. Sahada ayrıca o zamanlayıcının o anki değeri de bulunmaktadır. Zamanlayıcı çalışmıyorken bu değer –1 olmakta, iş parçacağı tarafından yaratılan zamanlayıcı başlatıldığında zamanlayıcının değeri, bildirilmiş olan çalışma süresine eşitlenmektedir. Her bir milisaniyede, bir sistem kesmesi tarafından zamanlayıcı tablosundaki değerler birer azaltılmakta, sıfır değerine ulaşan zamanlayıcılar için ise ilgili zamanaşımı fonksiyonları çağırılmaktadır. Bu mekanizma sayesinde süreç içerisindeki iş parçacıkları için milisaniye bazında çalışan bir zamanlayıcı yapısı sağlanmıştır.

6.1.2 İletişim yapısı

Sistemde, süreç içi, yani bir süreçte çalışan iş parçacıkları arasındaki iletişim sürecin bellek sahasında açılan ve tüm iş parçacıklarının ulaşabildiği kuyruk yapılarıyla sağlanmaktadır. Alıcı ve konsol dışındaki iş parçacıkları için birer kuyruk sahası bulunmaktadır. Alıcı ve konsol, süreç içerisindeki bir başka iş parçacığından herhangi bir

girdi almadıklarından bu iş parçacıkları için kuyruğa gerek yoktur. Ortak bellek sahasında yer alan bu kuyruklara erişimi senkronize etmek, çakışmaları önlemek için iş parçacıkları bazında çalışan semafor yapıları kullanılmıştır. İş parçacığına gönderilmek istenen mesajlar, ilgili kuyruğun sonuna eklenmekte, iş parçacığı ise kendi kuyruğunun en başındaki mesajı okumakta ve bu mesaj silinmektedir.

Sistemdeki, farklı bilgisayarlarda çalışan iş parçacıkları arasındaki iletişim ise her bir süreçte çalışan alıcı ve gönderici iş parçacıkları aracılığı ile iletişim ağı üzerinden gerçekleştirilmektedir. Herhangi bir iş parçacığı mesaj göndermek istediğinde, oluşturduğu mesajı göndericinin kuyruğuna bırakmakta, gönderici kendi kuyruğundan okuduğu mesajların gönderileceği adresleri inceleyerek, bir ağ paketi oluşturmakta ilgili bilgisayara göndermektedir. Diğer bilgisayarda, alıcı tarafından alınan bu mesajın adres bilgilerinden gönderildiği iş parçacacığı belirlenerek ilgili kuyruğa bırakılmaktadır. Böylece farklı süreçlerde, dolayısı ile farklı bilgisayarlarda yer alan iş parçacıkları arasındaki iletişim sağlanmaktadır.

İletişim protokolü olarak, IP (Internet Protocol) ağ protokolü üzerinde çalışan UDP (User Datagram Protocol) kullanılmaktadır. Önceki bölümlerde de bahsedildiği gibi bağlantısız bir iletişim protokolü olan UDP, her bir paketi birbirinden bağımsız olarak bir uçtan diğer uca herhangi bir bağlantı kurulumu gerçekleştirmeden iletmektedir. Bu protokol, özellikle yerel ağlarda, bağlantı kurulumu, sonlandırılması ve bazı gereksiz kontrol işlemlerinden arınmış olduğundan yüksek performans sağlamaktadır. UNIX ortamında geliştirilen sistemde "Berkeley Sockets" uygulama geliştirme arayüzü kullanılarak ağ iletişimi fonksiyonları gerçekleştirilmiştir.

Sistemde, sabit uzunlukta bir mesaj yapısı kullanılmaktadır. Bir mesaj şu sahalardan oluşmaktadır:

Gönderici adresi : Mesajı gönderen modülün (düğüm, temsilci ya da lider) adresidir.

Alıcı adresi : Mesajın önderildiği modülün adresidir.

Modül tipi : Alıcının düğüm, temsilci ya da lider olduğunu belirler.

Mesaj tipi : Mesajın data kısmında, ne tür bir bilgi olduğunu belirler.

Data : Gönderilen bilginin bulunduğu sahadır.

Sistemdeki adresleme mekanizması da şu şekildedir: Sistemdeki her modülün kendini belirten tekil bir adresi bulunmaktadır. Bu adres iki kısımdan oluşur: Birinci kısım, modülün bulunduğu bilgisayarı tanımlayan bilgisayar adresi, ikinci kısım ise o bilgisayardaki hangi iş parçacığı olduğunu tanımlayan modül numarasıdır. Yapılan simulasyonlarda bilgisayar adresi olarak IP adresi kullanılmıştır. Simulasyon amaçlı olarak bir bilgisayarda birden fazla düğüm ve temsilci modülü çalıştırılabilmektedir. Modül numarası da, bu birden çok sayıdaki düğümleri ve temsilcileri birbirinden ayırt etmek amacıyla kullanılmaktadır.

Mesaj gönderiminde bulunan bir modül,yeni bir mesaj oluşturarak, mesajın gönderici adresi, alıcı adresi ve modül tipi sahalarını doldurur. Göndereceği mesajın tipini belirtir ve data kısmına da asıl mesajı

yerleştirir ve gönderici iş parçacığının kuyruğuna bırakır. Aynı şekilde, aldığı bir mesajın mesaj tipine göre hangi tip bilgiyi taşıdığını anladıktan sonra data kısmını okuyarak ilgili işlemi gerçekleştirir.

Gönderici iş parçacacığı, kendi kuyruğundan bir mesaj aldığında, mesajın alıcı adres sahasını okur, eğer alıcı bilgisayar adresi ile kendi bilgisayar adresi aynı ise, mesajda yer alan modül tipine göre bu mesajın hangi modüle ait olduğunu belirler ve alıcı adres sahasındaki modül numarasına göre ilgili iş parçacığının kuyruğuna bırakır. Eğer alıcı bilgisayar adresi kendi adresinden farklı ise mesajı bir UDP paketine koyarak ilgili bilgisayara gönderir.

Alıcı iş parçacığı, ağ üzerinden bir UDP paketi aldığında, paketin içinden mesajı alır, mesajdaki modül tipinden mesajın hangi modüle ait gönderildiğini belirler ve alıcı adres sahasındaki modül numarasına göre ilgili modülün kuyruğuna yerleştirir.

Ek.1'de uygulaması gerçekleştirilen bu sistemin UNIX C dilinde yazılmış kodu yer almaktadır. (Temsilci modülü bir başka tez kapsamında yer aldığından bu modülün kodu bulunmamaktadır).

6.2 Dağıtık Yük Dengeleme Modülü Uygulaması

6.2.1 Sistemin yapısı ve işleyişi

Uygulaması gerçekleştirilen dağıtık sistem modeli altyapısı üzerine, bu altyapıyı kullanan bir dağıtık işlem izlencelemesi modülü geliştirilerek uygulaması gerçekleştirilmiştir. Gerçekleştirilen yapıda, düğüm, temsilci ve lider modüllerinin üzerlerine birer üst seviye modülü

yazılmıştır. Bu modüller, kodlama tekniği açısından modülerlik sağlaması için ayrı birer modül olarak yazılmış olmakla birlikte, ilgili alt modülün birer parçası olarak çalışmaktadırlar. Bunun anlamı, bölüm 6.1'deki iş parçacıkları aynen bu yapıda da yer almakta, üst modüller için ayrı iş parçacıkları kullanılmamaktadır. Bunun yanında düğüm, temsilci ve lider modüllerine ek işlevler getirilerek dağıtık işlem izlencelemesi ile ilgili görevlerini yerine getirmeleri sağlanmaktadır. Ayrıca konsol iş parçacığına da dağıtık işlem izlencelemesiyle ilgili, kullanıcının istediği düğümde işlem başlatabilmesini sağlayan komut eklentileri yapılmıştır.

Yapılan eklentilere bakacak olursak; sistemde konsol üzerinden düğümlere işlem başlatma komutları iletildiğinde dağıtık yük izlencelemesiyle ilgili işlevler başlatılmakta, işlemin sistem üzerindeki bir düğüme atanmasıyla sonuçlanmaktadır. Ayrıca, bir düğümde çalışan bir işlem sonlandığında yine belirli aktiviteler yerine getirilmektedir. Bunun yanında, sistemde dolaşan iç çerçevelere düğümlerin yük bilgilerini koymaları ve temsilcilerin bu bilgileri alarak belirli biçimlerde kendi üzerlerinde organize etmeleri sağlanmıştır. Dış halkada dolaşan dış çerçeveye temsilcilerin iç halkadan topladıkları bu bilgileri yerleştirmeleri ve bu bilgileri liderin okuyarak kendi bilgilerini güncellemesi işlevleri de gerçekleştirilmiştir.

Sistemde çalıştırılan işlemlerle ilgili gölge işlem yapısı oluşturulmuş, tüm düğümlerde bu işlemlerin birer kopyaları tutulmuştur. Gerçekleştirilen simulasyonlarda düğümler tarafından işlemler ayrı birer süreç olarak olarak değil, birer iş parçacığı olarak başlatılmaktadırlar.

6.2.2 İletişim yapısı

Altyapının iletişim mekanizmaları aynen kullanılarak, aynı süreç içerisindeki iş parçacıkları arası mesajlaşmalar kuyruk yapıları ile, süreçler arası iletişim ise UDP protokolü üzerinden gerçekleştirilmiştir. Alıcı ve gönderici iş parçacıkları dağıtık işlem izlencelemesiyle ile ilgili mesajların da alınıp gönderilmesini gerçekleştirmektedirler. Modül adreslemesi olarak yine aynı yapı (bilgisayar adresi+modül numarası) kullanılmıştır.

Mesaj yapısı olarak alt yapının genel mesaj formatı aynen korunarak, dağıtık işlem izlencelemesine özel mesajlar, genel mesajın data kısmına yerleştirilerek "uygulama mesajı" olarak adlandırılan ayrı bir mesaj tipi ile iletilmişlerdir. Düğüm, temsilci ve lider modülleri bir uygulama mesajı aldığında, bu mesajı yorumlayarak ilgili işlevin gerçekleştirilmesini sağlayan üst birim fonksiyonunu aktive ederler. Genel mesajın data kısmına yerleştirilen bir uygulama mesajının yapısı şu şekildedir:

Mesaj tipi : Mesajın hangi tür bir bilgiyi taşıdığını tanımlar.

Data : Mesajın içerdiği bilgilerin yer aldığı bölümdür.

Alt yapı tarafından kullanılan mesaj yapısının değiştirilmeden, üst birime özel ayrı bir uygulama mesajı yapısı kullanılması sisteme modüler bir yapı kazandırmakta, alt yapı ile bu yapının üzerine geliştirilen üst birimlerin birbirinden ayrılmasını ve bağımsız olarak geliştirilebilmelerini sağlamaktadır. Bu sayede temeldeki dağıtık sistem

modeli hiç değiştirilmeden, üzerine farklı amaçlı uygulamalar geliştirilebilmektedirler.

6.3 Simulasyon ve Ölçümler

Uygulaması gerçekleştirilen sistemin testleri, Ege Üniversitesi kampüsünde bulunan toplam 16 adet SUN iş istasyonu kulanılarak gerçekleştirilmiştir. Dağıtık yük izlencelemesi simulasyonu baz alınarak sistemin mesaj uzunluğu 3,7KB, dışçerçeve çıkarma periyodu da 200ms olarak ayarlanmıştır.

6.3.1 İki katmalı halka yapısı ölçüm değerleri

İki katmanlı yapıda, tek makine üzerinde birden fazla düğüm ve temsilci iş parçacıkları çalıştırılarak çeşitli konfigürasyonlarda simulasyonlar gerçekleştirilmiştir. Ölçülen iççerçeve dolaşım süreleri çizelge 6.1'de görülmektedir. Bu verilerde iççerçeve dolaşım süresinin kümedeki düğüm sayısının artmasıyla belirli oranda artış gösterdiği ortaya çıkmaktadır.

Çizelge 6.1. İki katmanlı sistemde iççerçeve dolaşım süreleri (ms)

		Bir kümedeki düğüm sayısı			
		4	**8**	**16**	**24**
Küme sayısı	**2**	3	5	11	16
	4	4	11	21	34
	8	5	19	43	68
	16	7	21		

Çizelge 6.2'de ise aynı konfigürasyonlardaki dışçerçeve dolaşım süreleri görülmektedir. Bu değerler de küme sayısındaki artışın dışçerçeve dolaşım süresine yansımasını göstermektedir. Verilerden de anlaşılacağı gibi küme sayısındaki artışla birlikte dışçerçeve dolaşım süresi lineer sayılabilecek bir oranda artmaktadır.

Çizelge 6.2. İki katmanlı sistemde dışçerçeve dolaşım süreleri (ms)

		Bir kümedeki düğüm sayısı			
		4	8	16	24
Küme sayısı	2	2	2	2	2
	4	4	4	4	4
	8	12	14	14	15
	16	29	34		

Çizelge 6.3'te ise sistemdeki bir temsilcinin çökmesi durumunda düğüm ve lider tarafından iç ve dış halkanın onarılması işlemleri sırasında geçen süreler görülmektedir. Çizelgede (I) ile gösterilen sütunlar temsilci çökmesinin iç halkanın son düğümü tarafından tespit edilmesinden halkanın onarılıp yeni temsilcinin başlatılmasına kadar geçen süreyi, (II) ile gösterilen sütunlar, liderin ölen temsilciyi halkadan çıkarıp, yeni dışçerçeve çıkarmaya hazır hale gelinceye kadar geçen süreyi, (III) ile gösterilen sütunlar ise yeni başlayan temsilcinin lider tarafından halkaya eklenmesi ve halkanın tekrar dışçerçeve dolaşacak hale getirilmesine kadar geçen süreyi göstermektedir. İlk iki olay birbiriyle paralel olarak aynı anda, üçüncü olay ise ikinci olayın tamamlanmasından sonra gerçekleşmektedir. Bu değerlerden temsilci çökmesi durumunda dış halkanın en fazla 40ms, iç halkanın da en fazla 80ms kadar bir sürede onarılarak normal işlevine döndüğü görülmektedir.

Çizelge 6.3. İki katmanlı sistemde hata düzeltme süreleri (ms)

		Bir kümedeki düğüm sayısı											
		4			8			16			24		
		I	II	III	I	II	III	I	II	III	I	II	III
Küme sayısı	2	4	38	20	5	37	19	13	37	19	18	37	19
	4	4	42	25	11	40	30	8	40	29	12	38	27
	8	10	42	26	8	43	38	17	38	40	23	39	42
	16	11	38	30	12	35	30						

6.3.2 İki katmanlı halka yapısında dağıtık yük dengelemesi ölçüm değerleri

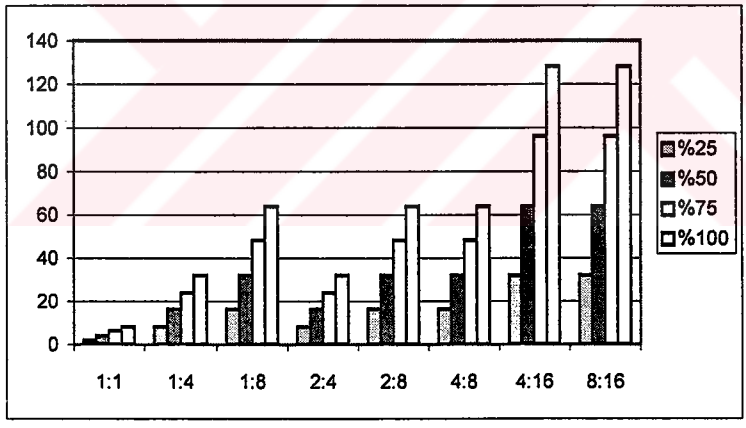
İki katmandan oluşan sistem üzerinde gerçekleştirilen denemelerde Çizelge 6.4.'de yer alan sonuçlar elde edilmiştir. Tablodaki değerler sistemin belirli yük seviyelerindeki ortalama cevap verme sürelerini (saniye bazında) göstermektedir. Sisteme, düşük (L), orta (M) ve yüksek (H) yük seviye aralıklarına karşılık gelecek şekilde yük bindirmesi yapılmış ve çeşitli konfigürasyonlarda ölçümler gerçekleştirilmiştir. Tablodaki kolon başlıkları bu konfigürasyonları ifade etmektedir ([sistemdeki küme sayısı]:[sistemdeki düğüm sayısı] biçiminde). İlk konfigürasyon, bir baz teşkil etmesi bakımından tek bir iş istasyonunda gerçekleşen ortalama değerleri göstermektedir.

Sistemde çalıştırılan işlem olarak, rastgele sıralanmış 5000 adet tamsayıdan oluşan bir diziyi "bubble sort" yöntemi ile küçükten büyüğe doğru sıralayan bir işlem kullanılmıştır. Yük seviyeleri bu işlemin paralel olarak belirli sayıda kopyasının çalıştırılmasıyla oluşturulmuştur.

Çalıştırılan işlem sayıları (sistemin yük seviyeleri), ölçüm yapılan konfigürasyondaki düğüm sayısı ile doğru orantılı olarak değişim göstermektedir. Şekil.6.2'de ölçüm yapılan konfigürasyonlarda çalıştırılan yük miktarları görülmektedir.

Çizelge 6.4. İki katmanlı sistemde ortalama cevap verme süreleri (sn)

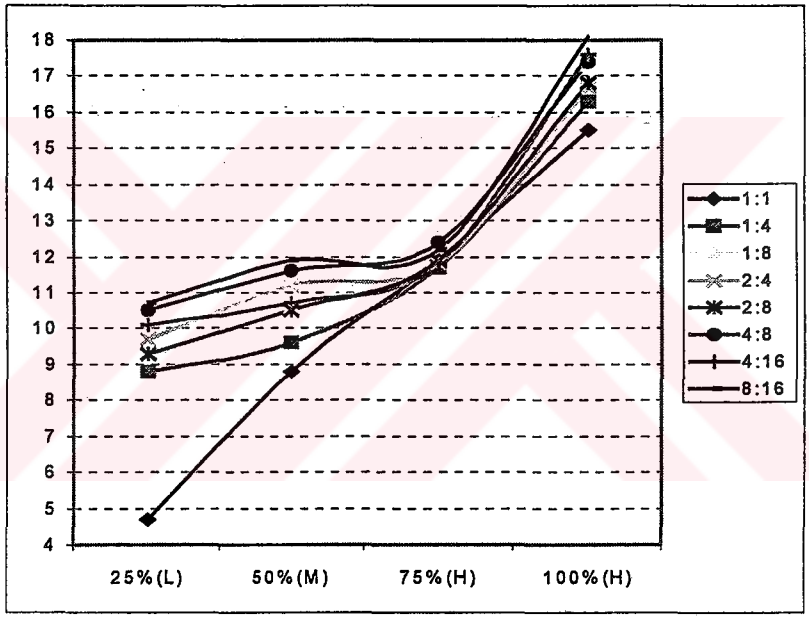
		Küme sayısı:Düğüm sayısı							
		1:1	1:4	1:8	2:4	2:8	4:8	4:16	8:16
Sistem yükü	25%(L)	4,7	8,8	9,2	9,7	9,3	10,5	10,1	10,7
	50%(M)	8,8	9,6	10,6	11,2	10,5	11,6	10,7	11,9
	75%(H)	11,9	11,7	12,5	11,8	11,9	12,4	11,8	12,2
	100%(H)	15,5	16,3	17,3	16,7	16,8	17,4	17,6	18,1



Şekil 6.2. İki katmanlı sistemde çalıştırılan süreç sayıları

Şekil 6.3'te, sistemin çeşitli konfigürasyonlarda yük seviyesine göre cevap verme süreleri karşılaştırılmaktadır. Bu grafikte, 1:1 konfigürasyonu baz olarak alındığında (bir bakıma ideal yük dağılımlı bir

sistem gibi de düşünülebilir) düğüm sayısıdaki artışın cevap verme süresinde küçük de olsa belirli bir artışa sebep olduğu görülmektedir. Düşük seviyelerde 1:1'e göre yüksek olan cevap verme süreleri, bu seviyelerde sistemin yük dağıtımı yapmamasından kaynaklanmakta, yüksek değerlere doğru ise sistemin yük dağıtımı yapmaya başlaması sonucu cevap verme süreleri bir dengeye gelmektedir. Yüksek yük seviyesi aşıldığında ise sistemin cevap verme süresinin belirgin olarak yükseldiği görülmektedir.



Şekil 6.3. İki katmanlı sistemdeki ortalama cevap verme süreleri

Orta düzey yük seviyesinde cevap verme süreleri birbirine yakın değerlerde seyretmekte, yüksek yük seviyesine doğru daha da yaklaşmaktadır. Yük artışlarının, düşük ve orta düzeylerde cevap verme süresine yaklaşık olarak aynı oranlarda yansıdığı görülürken, yüksek yük

seviyesi aşıldığında sistemin cevap verme süresinin yük artışına göre daha yüksek oranda arttığı görülmektedir. Bu da sistemin doyum noktasına ulaşmış olmasından dolayı yük transfer taleplerini geri çevirmesinden, yani sistemin daha fazla yük dağıtımı yapamamasından kaynaklanmaktadır.

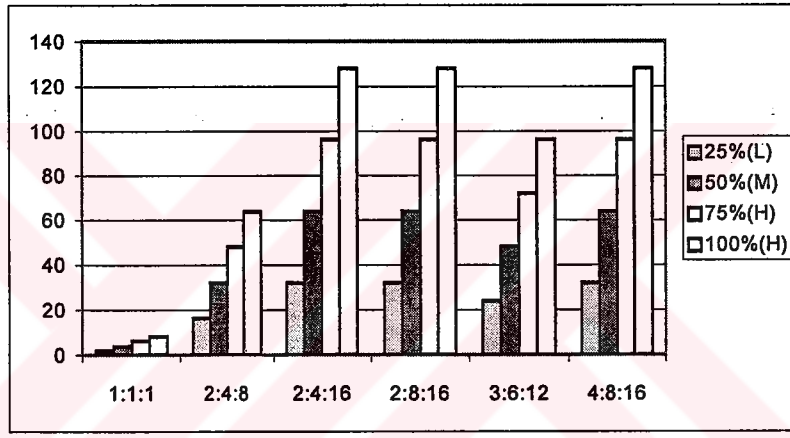
6.3.3 Üç katmanlı halka yapısında dağıtık yük dengelemesi ölçüm değerleri

İki katmanlı halka yapısında gerçekleştirilmiş olan dağıtık yük dengelemesi modelinin üç katmanlı topolojiye uyarlanmış ve bu sisteme ilişkin yapılan denemelerde çizelge 6.5.'te yer alan sonuçlar elde edilmiştir. Tablodaki değerler iki katmalı yapıdakine benzer şekilde sistemin belirli yük seviyelerindeki ortalama cevap verme sürelerini (saniye bazında) göstermektedir. Sisteme, düşük (L), orta (M) ve yüksek (H) yük seviye aralıklarına karşılık gelecek şekilde yük bindirmesi yapılmış ve çeşitli konfigürasyonlarda ölçümler gerçekleştirilmiştir. Tablodaki kolon başlıkları [üst katman küme sayısı]:[alt katman küme sayısı]:[düğüm sayısı] biçiminde bu konfigürasyonları ifade etmektedir. İlk konfigürasyon ([1:1:1]), bir baz teşkil etmesi bakımından tek bir bağımsız makinede gerçekleşen ortalama değerleri göstermektedir.

Yük olarak, iki katmanlı sistemde kullanılan işlem kullanılmıştır. Şekil.6.4'te ölçüm yapılan konfigürasyonlarda çalıştırılan işlem sayıları görülmektedir.

Çizelge 6.5. Üç katmanlı sistemde ortalama cevap verme süreleri (sn)

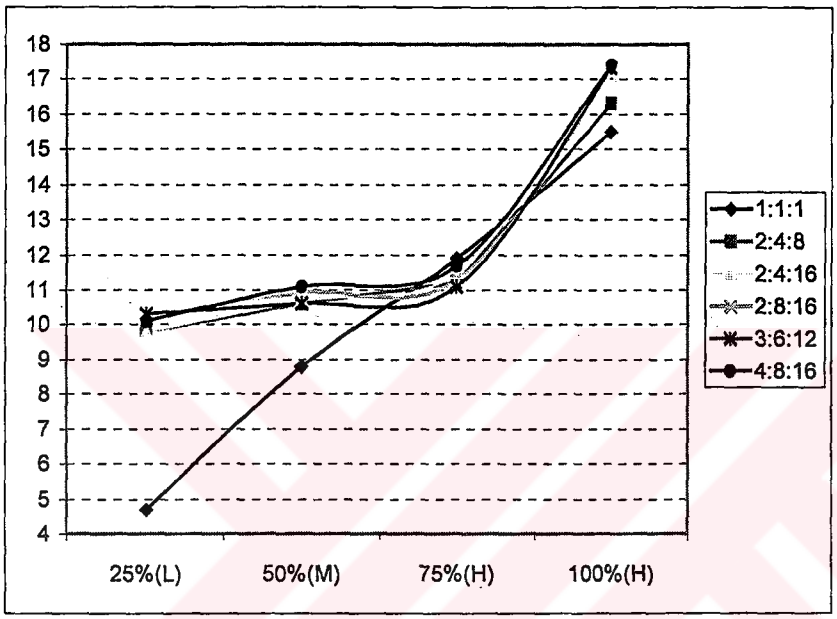
		Üst katman küme sayısı:Alt katman küme sayısı:Düğüm sayısı					
		1:1:1	2:4:8	2:4:16	2:8:16	3:6:12	4:8:16
Sistem yükü	25%(L)	4,7	9,8	9,8	10,3	10,3	10,1
	50%(M)	8,8	10,6	10,7	10,9	10,6	11,1
	75%(H)	11,9	11,2	11,5	11,3	11,1	11,7
	100%(H)	15,5	16,3	17,4	17,1	17,3	17,4



Şekil 6.4. Üç katmanlı sistemde çalıştırılan süreç sayıları

Şekil 6.5'te, sistemin çeşitli konfigürasyonlarda yük seviyesine göre cevap verme süreleri karşılaştırılmaktadır. Bu grafikte de, iki katmanlı halka yapısına benzer sonuçlar elde edildiği görülmektedir. Düşük seviyelerde 1:1:1'e göre yüksek olan cevap verme süreleri, bu seviyelerde sistemin yük dağıtımı yapmamasından kaynaklanmakta, yüksek değerlere doğru ise sistemin yük dağıtımı yapmaya başlaması sonucu cevap verme süreleri bir dengeye gelmektedir. Yüksek yük

seviyesi aşıldığında ise sistemin cevap verme süresinin belirgin olarak yükselmektedir.



Şekil 6.5. Üç katmanlı sistemdeki ortalama cevap verme süreleri

7 SONUÇ

Bu tezde, ilk olarak gerçek zamanlı, dağıtık uygulamalar için bir altyapı oluşturmak amacıyla geliştirilmiş hiyerarşik kümeleme tabanlı bir iletişim protokolünün uygulaması gerçekleştirilmiştir. Böyle bir sistemin öncelikle, sağlaması gereken temel özellikler araştırılmıştır. Böyle bir sistemin performans, ölçeklenebilirlik, güvenilirlik ve saydamlık gibi dağıtık sistemlerin başlıca hedeflerini karşılaması gerektiği, bunun yanında gerçek zamanlı uygulamalar için zaman bağımlı, tahmin edilebilir ve hata toleranslı çalışabilme özelliklerini de yerine getirmesi gerektiği ortaya çıkmıştır.

Uygulaması gerçekleştirilen yapıda, düğüm adı verilen işlemci birimlerinin oluşturduğu her bir küme, küme temsilcileri tarafından yönetilmekte, küme içi iletişim bir iç halka protokolü ile sağlanmaktadır. Küme temsilcilerinin oluşturduğu ve lider olarak adlandırılan bir birimin yönetiminde bulunan dış halkada ise ayrı bir başka halka protokolü çalışmaktadır. Çalışan halka protokolleri senkron bir yapıdadır. Her bir kümede ve kümeler arası çalışan halka protokolleri birbirinden bağımsız bir yapıdadadır.

Bu senkron halka protokolüne bir hata kontrol mekanizması da yerleştirilerek, işlemci çökmelerinin en kısa sürede tespit edilerek sorunun giderilmesi sağlanmıştır. Küme içinde meydana gelen bir çökmede sadece ilgili küme üyeleri ve temsilcisinin sorunun giderilmesinde rol oynamasıyla, hatanın sistemin diğer kümelerinden izole edilmesi ve normal işlevlerini sürdürmeleri sağlanmıştır.

Bir kümenin normal işlevi, kümeye yeni düğüm katılımları, çöken düğümün kümeden çıkarılması gibi iç operasyonlar ve bu operasyonlar için küme içinde gerçekleştirilen iletişim trafiği tamamen diğer kümelerden bağımsız olarak gerçekleştirilmektedir. Bu da iletişim ve işletim performansını arttırmakta ve ölçeklenebilirlik özelliğinin yanında sistemin hata toleranslı çalışabilmesini de sağlamaktadır. Yeni düğüm ya da küme eklemelerinde ve çökme durumlarında sistemin diğer birimleri normal işlevlerini sürdürebilmektedir.

Kümeleme yapısında sisteme yeni düğümlerin eklenmesinden, mesaj karmaşıklığı yönünden sadece ilgili küme etkilenmekte, halka protokolü sayesinde iletişim yoğunluğu artışı çok düşük seviyelerde tutulabilmektedir. Benzer olarak, sisteme yeni kümelerin eklenmesi de sadece dış halkada küçük iletişim maliyeti getirmektedir. Bu da sistemin ölçeklenebilir bir yapıda olmasını ve iletişim maliyetlerinin de tahmin edilebilir düzeylerde olmasını sağlamaktadır.

Lider tarafından belirli periyodlarda çerçeve çıkarılmasıyla gerçekleştirilen senkron iletişim protokolü, gerçek zamanlı uygulamalar için uygun bir yapı sağlamaktadır. Bu yapı üzerine, gerçek zamanlı dağıtık uygulamaların ihtiyaç duyduğu, tüm sistemin ortak saat değerine sahip olmasını sağlayan saat senkronizasyonu mekanizması geliştirilmesi sağlanabilmektedir. Ayrıca, gerçek zamanlı sistemlerin ihtiyaç duyduğu, yüksek kullanılabilirlik özelliği de, sistem üzerinde çalışan gerçek zamanlı uygulamaların, farklı kümelerde kopyalarının çalıştırılmasıyla sağlanabilecektir. Bu kopya uygulamaların tutarlılığının sağlanması için de yine bu sistem üzerinde geliştirilen ve başka bir tezin kapsamında bulunan grup yönetimi mekanizması kullanılabilecektir.

Geliştirilen platform, saat senkronizasyonu, grup yönetimi ve dağıtık yük izlencelemesi gibi gerçek zamanlı dağıtık üst modüllerin birbirinden bağımsız olarak geliştirilebilmesini ve çalıştırılabilmesini mümkün kılmakta ve üst uygulamalara saydam bir yapı sağlamaktadır.

Ege Üniversitesi kampüsündeki UNIX iş istasyonları üzerinde uygulaması gerçekleştirilen bu protokolün amaçlanan hedefleri büyük ölçüde karşıladığı gözlemlenmiştir.

Tezin ikinci bölümünde, uygulaması gerçekleştirilen dağıtık gerçek zamanlı platformun üzerine bir dağıtık yük dengelemesi modülü tasarlanmış ve uygulaması gerçekleştirilmiştir. Bu amaçla, öncelikle dağıtık sistemlerin ihtiyaç duydğu yük dağıtımı gereksinimi ve temel özellikleri araştırılmış ve belirlenmiştir. Daha sonra çeşitli dağıtık yük dengeleme algoritmaları incelenmiş ve sonunda geliştirilen platforma uygun bir model tasarlanmıştır.

Dağıtık bir yük dengelemesi modeli, sistemin kaynaklarını en etkin şekilde yönetmeli, sistemin yükünü bu kaynaklara adil ve saydam bir şekilde paylaştırabilmeli ve sistemin performansını en üst düzeyde tutabilmelidir. Bu işlevleri gerçekleştirirken ölçeklenebilir bir yapıda olmalı, sisteme yeni kaynakların eklenmesi durmunda da hedeflerini aynı kalitede gerçekleştirebilmelidir. Ayrıca lokasyon bağımsızlığı olarak da adlandırılan, yükün başka bir işlemcide çalıştırılsa bile, sanki orijinal işlemcide çalışıyormuş gibi işlev görecek ve sonuçlarını da bu şekilde üst birime iletebilecek bir saydam yapı sağlaması şarttır. Tüm bu görevleri yerine getirirken dağıtık işlem programlayıcısı, özellikle sistem yükü yüksek seviyelerdeyken sistemin performansına gereksiz ek yük getirerek sistemin dengesini olumsuz etkilememelidir.

Bu amaçlarla tasarlanan yapıya, gerçek zamanlı özellik sağlayabilmek için zaman kısıtı da eklenmiş ve ortaya bir model çıkarılmıştır. Bu modelde, sisteme iletilen bir süreç, belirli bir zaman kısıtı içerisinde yük durumu uygun bir düğüme atanmaktadır.

Sistemde belirli periyodlarda dolaşan çerçeveler ile temsilciler kendi kümelerinden, liderler de temsilcilerde düğümlerin yük bilgilerini toplayarak bu bilgiler doğrultusunda düğümlerden gelen yük transfer isteklerine cevap vermektedirler. Yük transfer istekleri öncelikle temsilciler tarafından kendi kümeleri içerisinde karşılanmaya çalışılmaktadır. Kümenin yük seviyesi isteği karşılayacak düzeyin üzerinde ise, istek temsilci tarafından lidere iletilmekte ve lider sistemde başka bir kümede yük transferi için uygun bir düğüm aramaktadır. Bu yapı sayesinde yine ölçeklenebilir bir yapı sağlanmakta, küme içi transferler sistemin diğer birimlerinden izole edilebilmektedir. Yük transferi için gerçekleştirilen mesaj trafiği de oldukça düşük seviyelerde tutulmakta ve mesaj maliyeti tahmin edilebilir düzeylerde olmaktadır. Ayrıca, bilgi toplama mekanizması için sistemin senkron çerçeve dolaşım protokolü kullanılarak sisteme ek iletişim yoğunluğu getirilmemektedir.

Yüksek yük seviyelerinde düğümlerden gelen yoğun yük transfer isteklerinin lidere ulaştırılması ve lider tarafından reddedilmesi yüksek bir olasılık olmakta, fakat bu gibi durumlarda da lider ve temsilcilerin bilgileri sınıflama yöntemi sayesinde arama işlemleri minimal bir işlemci yoğunluğu getirmekte ve iletişim trafiği de yine düşük seviyelerde tutulmaktadır. Bunun anlamı, yük eviyesinin yüksek olduğu durumlarda bile yük dengelemesi mekanizması sisteme kabul edilebilir düzeyde bir yük (overhead) getirmekte ve hızlı bir sonuç üreterek zaman kısıtının da karşılanabilmesine imkan sağlamaktadır.

Sistemde kullanılan gölge süreç mantığı sayesinde, düğümlerde süreçlerin birer kopyasının bulundurulmasıyla, yük transferleri yapılırken süreçlerin bir başka düğüme transfer edilmesi ihtiyacı ortadan kalkmış ve iletişim maliyetlerini büyük oranlarda düşürmüştür. Ayrıca yük transferlerinde sadece yeni gelen işlerin transfer kapsamına alınması da sistemi süreçlerin durum bilgilerinin transfer edilmesi işinden kurtarmıştır.

Yapılan testlerde de elde edilen sonuçlar, dağıtık yük dengelemesi mekanizmasının sistemin performansını optimum düzeyde tuttuğu, sistemi dengesiz bir yapıya sürüklemediği ve ölçeklenebilir bir yapıda olduğunu ortaya çıkarmıştır.

KAYNAKLAR DİZİNİ

Audsley, N. and Burns, A., 1992, Real-Time System Scheduling, Predicatably Dependable Computer Systems, Volume 2

Chang, H. Y. and Livny, M., 1986, Distributed Scheduling Under Deadline Constraints: A Comparison of Sender-Initiated and Receiver-Initiated Approaches, Proceedings 7th IEEE Real-time Systems Symposium

Chetali B., 1995, A Formal Proof of A Protocol For Communications Over Faulty Channels Using The Larch Prover, Rapport De Recherce, Inria Lorraine, 22p.

Christian, F., 1993, Understanding Fault-Tolerant Distributed Systems, Technical Report, Department of Computer Science and Engineering, University of California, San Diego.

Cormen T., Leiserson C., Rivest R., 1990, Introduction to Algorithms "Analysis of Algorithms", MIT Electrical Engineering and Computer Science, MIT Press, 1028p.

Coulouris G., Dollimore J., T.K., 2000, Distributed Systems: Concepts and Design, Addison Wesley Pub Co., 672p.

Denker G., Meseguer J., Talcot C., 2000, Formal Specification and Analysis of Active Networks and Communication Protocols: The Maude Experience, Computer Science Laboratory, SRI Internaional, 15p.

KAYNAKLAR DİZİNİ (DEVAM)

Filliâtre, J., 2000, Formal Proof of A Program: Find, Computer Science Laboratory, SRI International, 12p.

Jalote, P., 1994, Fault Tolerance in Distributed Systems, Prentice Hall, 432p.

P. Krueger and R. A. Finkel., 1984, An Adaptive Load Balancing Algorithm for a Multicomputer. Technical Report 539, Dept. of Computer Sciences, University of Wisconsin--Madison.

Livny M. and Melman M., 1982, Load Balancing in Homogeneous Distributed Systems. Proceedings of the ACM Computer Networking Performance Symposium, pp. 47-55

Lynch N., 1997, Distributed Algorithms, Morgan Kaufmann, San Francisco, 872p.

Mullender, S., 1995, Distribued Systems, Addison Wesley Pub Co., Third Edition. 595p.

Singhal M., Shivaratri N., 1994, Advanced Concepts In Operating Systems, McGraw Hill, New York, 448p.

Stevens, R., 1990, Unix Network Programming, Prentice Hall, New Jersey, 772p.

Tanenbaum, A., 1995, Distributed Operating Systems, Prentice Hall, New Jersey, 614p.

KAYNAKLAR DİZİNİ (DEVAM)

Tanenbaum, A., 1997, Computer Networks, Prentice Hall, New Jersey, 813p.

Toker, A., 1994, Interprocess Communication and Real-Time Programming In The Unix Environment, Final Year Project, Ege University, Izmir, 145p.

Tunalı, T., Erciyeş, K., Soysert, Z., 1998, A Ring Protocol For A Cluster Based Distributed System, BAS'98, The Third Symposium on Computer Networks , Izmir, 10p.

EKLER

Ek 1 Düğüm Modülü Algoritması

```
Düğüm(temsilci_adresi)
{
yerel_değişkenleri_ayarla
durum=ARA
temsilciye KATILIM_İSTEĞİ mesajı gönder
KOMŞU_BİLGİSİ mesajı bekle
Döngü
   olay_bekle
      olay=İÇ_HALKA_KONTROL_ÇERÇEVESİ_ALINDI
         bilgi_sahasını_işaretle
         komşuya İÇ_HALKA_KONTROL_ÇERÇEVESİ gönder
         durum=BEKLE
         İÇÇERÇEVE_ZAMANLAYICISI_başlat

      olay=İÇÇERÇEVE_ALINDI
         Eğer durum=BEKLE ise
            İÇÇERÇEVE_ZAMANLAYICISI_durdur
            Eğer dış_halka_problemi değilse Then
                  genel_bilgiyi_oku
                  yerel_bilgiyi_koy
            komşuya İÇÇERÇEVE gönder
            İÇÇERÇEVE_ZAMANLAYICISI_başlat

      olay=İÇÇERÇEVE_ZAMANAŞIMI
         durum=ARA
         bir önceki düğüme YAŞIYOR_MUSUN mesajı gönder
         YAŞIYOR_MUSUN_1_ZAMANLAYICISI_başlat
         bir sonraki düğüme YAŞIYOR_MUSUN mesajı gönder
         YAŞIYOR_MUSUN_2_ZAMANLAYICISI_başlat

      olay=YAŞIYOR_MUSUN_1_ZAMANAŞIMI
         Eğer önceki komşu temsilci ise
            lidere ÖLÜ_KOMŞU mesajı gönder
            temsilcinin bir sonraki komşusuna KOMŞU_BİLGİSİ
            mesajı gönder
            yeni_temsilci_başlat
         Değilse
```

```
            temsilciye ÖLÜ_KOMŞU mesajı gönder

        olay=YAŞIYOR_MUSUN_2_ZAMANAŞIMI
            Eğer önceki komşu temsilci değil ise
                temsilciye ÖLÜ_KOMŞU mesajı gönder

        olay=HALKA_BİLGİ_TOPLA_ÇERÇEVESİ_ALINDI
            Eğer durum=BEKLE ise
                İÇÇERÇEVE_ZAMANLAYICISI_durdur
                durum=ARA
            halka_bilgisi_güncelle
            adres_sahasını_işaretle
            sonraki komşuya HALKA_BİLGİ_TOPLA_ÇERÇEVESİ
            gönder

        olay=KOMŞU_BİLGİSİ_MESAJI_ALINDI
            Eğer durum=BEKLE ise
                İÇÇERÇEVE_ZAMANLAYICISI_durdur
                durum=ARA
            komşu_bilgisi_güncelle

        olay=YAŞIYOR_MUSUN_MESAJI_ALINDI
            göndericiye YAŞIYORUM mesajı gönder

        olay=YAŞIYORUM_MESAJI_ALINDI
            Eğer gönderici önceki komşu ise
                YAŞIYOR_MUSUN_1_ZAMANLAYICISI_durdur
            Değilse
                YAŞIYOR_MUSUN_2_ZAMANLAYICISI_durdur
DöngüSonu
}
```

Ek 2 Lider Modülü Algoritması

```
Lider(önceki_komşu_adresi,sonraki_komşu_adresi)
{
yerel_değişkenleri_ayarla
Eğer sonraki_komşu_adresi=boş ise
   durum=ARA
   sonraki komşuya DIŞ_HALKA_BİLGİ_TOPLA mesajı gönder
Değilse
   durum=BEKLE

Döngü
   olay_bekle
      olay=DIŞÇERÇEVE_ALINDI
         Eğer durum=BEKLE ise
            DIŞÇERÇEVE_ZAMANLAYICISI_durdur
            Eğer çerçeve_durumu=ÇERÇEVE_ÇIKAR ise
                  çerçeve_durumu=boş
                  sonraki komşuya DIŞÇERÇEVE gönder
                  DIŞÇERÇEVE_ZAMANLAYICISI_başlat
                  ÇERÇEVE_ÇIKARMA_ZAMANLAYICISI_
                  başlat
            Değilse
                  çerçeve_durumu=ÇERÇEVE_ALINDI

      olay=DIŞ_HALKA_KONTROL_ÇERÇEVESİ_ALINDI
         Eğer durum=ARA ise
            DIŞÇERÇEVE_ZAMANLAYICISI_durdur
            Durum=BEKLE
            çerçeve_durumu=boş
            sonraki komşuya DIŞÇERÇEVE gönder
            DIŞÇERÇEVE_ZAMANLAYICISI_başlat
            ÇERÇEVE_ÇIKARMA_ZAMANLAYICISI_başlat

      olay=DIŞ_HALKA_BİLGİ_TOPLA_ÇERÇEVESİ_ALINDI
         DIŞÇERÇEVE_ZAMANLAYICISI_durdur
         Halka_bilgisi_güncelle
         sonraki komşuya DIŞ_HALKA_KONTROL_ÇERÇEVESİ
         gönder
```

```
        DIŞÇERÇEVE_ZAMANLAYICISI_başlat

olay=YAŞIYOR_MUSUN_MESAJI_ALINDI
    Göndericiye YAŞIYORUM mesajı gönder

olay=YAŞIYORUM_MESAJI_ALINDI
    Eğer gönderici=önceki komşu ise
        YAŞIYORMUSUN_1_ZAMANLAYICISI durdur
    Değilse
        YAŞIYORMUSUN_2_ZAMANLAYICISI durdur

    Eğer her iki zamanlayıcı da durdurulmuş ise Then
      sonraki komşuya DIŞ_HALKA_KONTROL_ÇERÇEVESİ
      gönder

olay=KATILIM_İSTEĞİ_MESAJI_ALINDI
    Eğer durum=BEKLE ise
        DIŞÇERÇEVE_ZAMANLAYICISI_durdur
        ÇERÇEVE_ÇIKARMA_ZAMANLAYICISI_durdur
        durum=ARA
        Eğer önceki komşu<>boş ise
                önceki komşuya KOMŞU_BİLGİSİ mesajı gönder
                Eğer önceki komşu<>sonraki komşu ise
                        Sonraki komşuya KOMŞU_BİLGİSİ
                        mesajı gönder
        yeni eklenen temsilciye KOMŞU_BİLGİSİ mesajı gönder
        halka_bilgisi_güncelle
      sonraki komşuya DIŞ_HALKA_KONTROL_ÇERÇEVESİ
      gönder
      DIŞÇERÇEVE_ZAMANLAYICISI_başlat

olay=ÖLÜ_KOMŞU_MESAJI_ALINDI
    Eğer durum=BEKLE ise
        Durum=ARA
        DIŞÇERÇEVE_ZAMANLAYICISI_durdur
        ÇERÇEVE_ÇIKARMA_ZAMANLAYICISI_durdur
        Eğer ölü_temsilci=sonraki komşu ise
                ölü temsilcinin sonraki komşusuna
                KOMŞU_BİLGİSİ mesajı gönder
```

```
                önceki komşuya KOMŞU_BİLGİSİ mesajı
                gönder
        Değilse
                Ölü temsilcinin önceki komşusuna
                KOMŞU_BİLGİSİ mesajı gönder

        Eğer ölü_temsilci=önceki komşsu ise
                ölü temsilcinin önceki komşusuna
                KOMŞU_BİLGİSİ mesajı gönder
                sonraki komşuya KOMŞU_BİLGİSİ mesajı
                gönder
        Değilse
                Ölü temsilcinin sonraki komşusuna
                KOMŞU_BİLGİSİ mesajı gönder

      halka_bilgisi_güncelle
      sonraki komşuya DIŞ_HALKA_KONTROL_ÇERÇEVESİ
      gönder
      DIŞÇERÇEVE_ZAMANLAYICISI_başlat

olay=DIŞÇERÇEVE_ZAMANAŞIMI
    durum=ARA
    ÇERÇEVE_ÇIKARMA_ZAMANLAYICISI_durdur
    sonraki komşuya YAŞIYOR_MUSUN mesajı gönder
    YAŞIYOR_MUSUN_1_ZAMANLAYICISI_başlat
    önceki komşuya YAŞIYOR_MUSUN mesajı gönder
    YAŞIYOR_MUSUN_2_ZAMANLAYICISI_başlat

olay=YAŞIYOR_MUSUN_1_ZAMANAŞIMI
    halka_bilgisi_güncelle
    Eğer önceki komşu=boş ise
       Durum=BEKLE
    Değilse
       Ölü temsilcinin önceki komşusuna KOMŞU_BİLGİSİ
       mesajı gönder
       sonraki komşuya KOMŞU_BİLGİSİ mesajı gönder
    sonraki komşuya DIŞ_HALKA_KONTROL_ÇERÇEVESİ
    gönder
    DIŞÇERÇEVE_ZAMANLAYICISI_başlat
```

```
        olay=YAŞIYOR_MUSUN_2_ZAMANAŞIMI
            halka_bilgisi_güncelle
            Eğer sonraki komşu=boş ise
                Durum=BEKLE
            Değilse
                Ölü temsilcinin sonraki komşusuna KOMŞU_BİLGİSİ
                mesajı gönder
                önceki komşuya KOMŞU_BİLGİSİ mesajı gönder
            sonraki komşuya DIŞ_HALKA_KONTROL_ÇERÇEVESİ
            gönder
            DIŞÇERÇEVE_ZAMANLAYICISI_başlat

        olay=ÇERÇEVE_ÇIKARMA_ZAMANAŞIMI
            Eğer durum=BEKLE ise
                Eğer çerçeve_durumu=ÇERÇEVE_ALINDI ise
                    sonraki komşuya DIŞÇERÇEVE gönder
                    DIŞÇERÇEVE_ZAMANLAYICISI_başlat
                    ÇERÇEVE_ÇIKARMA_ZAMANLAYICISI_başlat
                    çerçeve_durumu=boş
                Değilse
                        çerçeve_durumu=ÇERÇEVE_ÇIKAR
DöngüSonu
}
```

Ek 3 Düğüm Dağıtık Yük Dengeleme Modülü Algoritması

```
yerel_değişkenleri_ayarla
Döngü
{
olay_bekle
   olay=İÇÇERÇEVE_ALINDI
        yük_değerini_iççerçeveye_koy
        İÇÇERÇEVE_gönder

   olay=SÜREÇ_BAŞLAT_KOMUTU_ALINDI
        süreç_tablosunda_bir_giriş_aç
        Eğer yük<YÜKSEK ise
             süreç_başlat
             yük_değerini_arttır
        Değilse
             temsilciye YÜK_TRANSFER_İSTEĞİ mesajı gönder
             YÜK_TRANSFER_ZAMANLAYICISI_başlat

   olay=YÜK_TRANSFER_ALICISI_MESAJI_ALINDI
        YÜK_TRANSFER_ZAMANLAYICISI_durdur
        alıcıya YÜK_TRANSFER_İSTEĞİ mesajı gönder
        YÜK_TRANSFER_ZAMANLAYICISI_başlat

   olay=YÜK_TRANSFER_RED_MESAJI_ALINDI
        YÜK_TRANSFER_ZAMANLAYICISI_durdur
        süreç_başlat
        yük_değeri_arttır

   olay=YÜK_TRANSFER_İSTEĞİ_MESAJI_ALINDI
        Eğer yük<YÜKSEK ise
        süreç_tablosunda_bir_giriş_aç
        Eğer yük<YÜKSEK ise
             süreç_başlat
             yük_değerini_arttır
             göndericiye YÜK_TRANSFER_BİTTİ mesajı gönder
        Değilse
             göndericiye YÜK_TRANSFER_RED mesajı gönder
```

```
olay=YÜK_TRANSFER_BİTTİ_MESAJI_ALINDI
    YÜK_TRANSFER_ZAMANLAYICISI_durdur

olay=YÜK_TRANSFER_ZAMANAŞIMI
        süreç_başlat
        yük_değerini_arttır

olay=SÜREÇ_SONLANDI_MESAJI_ALINDI
    Eğer süreç_tipi=YEREL ise
        Süreç_sonucu_yaz
        yük_değeri_azalt
    Değilse
        Eğer  süreç_tipi=UZAK ise
            süreç_sonucu_yaz
        Değilse
            yük_değeri_azalt
            göndericiye SÜREÇ_SONLANDI mesajı gönder
}
DöngüSonu
```

Ek 4 Temsilci Dağıtık Yük Dengeleme Modülü Algoritması

```
yerel_değişkenleri_ayarla
Döngü
{
olay_bekle
   olay=İÇÇERÇEVE_ALINDI
        düşük_tablo, orta_tablo ve yüksek_tablo güncelle

   olay=DIŞÇERÇEVE_ALINDI
        yük_değerlerini_dışçerçeveye_koy
        DIŞÇERÇEVE_gönder

   olay=YÜK_TRANSFER_İSTEĞİ_MESAJI_ALINDI
        Eğer düşük_tablo boş değil ise
             düşük tablonun ilk elemanını seç
        Değilse
             Eğer orta_tablo boş değil ise
                  orta tablonun ilk elemanını seç

        Eğer bir düğüm seçildi ise
             seçilen düğümün yük değerini güncelle
             seçilen düğümü uygun tabloya yerleştir
             göndericiye YÜK_TRANSFER_ALICISI mesajı gönder
        Değilse
             lidere YÜK_TRANSFER_İSTEĞİ mesajı gönder
}
DöngüSonu
```

Ek 5 Lider Dağıtık Yük Dengeleme Modülü Algoritması

```
yerel_değişkenleri_ayarla
Döngü
{
olay_bekle
   olay=DIŞÇERÇEVE_ALINDI
        düşük_tablo, orta_tablo ve yüksek_tablo güncelle

   olay=YÜK_TRANSFER_İSTEĞİ_MESAJI_ALINDI
        Eğer düşük_tablo boş değil ise
             düşük tablonun ilk elemanını seç
        Değilse
             Eğer orta_tablo boş değil ise
                orta tablonun ilk elemanını seç

        Eğer bir düğüm seçildi ise
             seçilen düğümün yük değerini güncelle
             seçilen düğümü uygun tabloya yerleştir
             göndericiye YÜK_TRANSFER_ALICISI mesajı gönder
        Değilse
             göndericiye YÜK_TRANSFER_RED mesajı gönder
}
DöngüSonu
```

ÖZGEÇMİŞ

Kişisel Bilgiler:

Adı Soyadı : Oğuz Akay
Doğum Tarihi : 07.04.1975
Doğum Yeri : Ödemiş - İZMİR
Uyruğu : T.C.
Yabancı Dilii : İngilizce
Adres : 574 Sok. No:43 A Blok 35040 Bornova/İZMİR
Telefon : (232) 371 37 28

Eğitim Durumu:

1997-2001 Yüksek Lisans, Ege Üniversitesi Uluslararası Bilgisayar Enstitüsü
Tez – Dağıtık Sistemlerde Dinamik Yük Dengeleme
Danışman: Prof. Dr. Kayhan Erciyeş

1993-1997 Lisans, Ege Üniversitesi Bilgisayar Mühendisliği Bölümü
Tez - Implementing an Agent Based Software Reuse Application over The Internet with Java
Danışman: Yrd.Doç. Dr. Oğuz Dikenelli

1992-1993 Ege Üniversitesi Yabancı Diller Bölümü
İngilizce Hazırlık Sınıfı

1989 -1992 Ödemiş Lisesi

İş Deneyimi:

11/1998-	Bilgi Teknolojileri Uzmanı Yaşar Astron Proje Ofisi - İzmir
07/1996- 11/1998	Sistem Analisti Dyo ve Sadolin A.Ş. - İzmir
07/1996-08/1996	Stajyer Yaşar Bilgi İşlem ve Ticaret A.Ş. – İzmir