İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

DAĞITILMIŞ NESNESEL BİRİM MODELİ ÜZERİNDE GÖRÜNTÜ İŞLEME ALGORİTMALARININ PARALELLEŞTİRİLMESİ

YÜKSEK LİSANS TEZİ
Müh. Savaş KÖSE
504960601011

Tezin Enstitüye Verildiği Tarih : 31 Mayıs 1999
Tezin Savunulduğu Tarih : 22 Haziran 1999

Tez Danışmanı :	Doç.Dr. Nadia ERDOĞAN
Diğer Jüri Üyeleri	Prof.Dr. Emre HARMANCI
	Prof.Dr. Oğuz TOSUN (B.Ü.)

12.7.1999

HAZİRAN 1999

ÖNSÖZ

Bana araştırmalarımda yardımını esirgemeyen, projenin gerek araştırma gerekse uygulama kısmında beni sürekli olarak motive eden değerli hocam Doç. Dr. Nadia Erdoğan'a , projenin başından beri birilikte çalıştığımız değerli arkadaşım Cenker Şişman'a teşekkürü bir borç bilirim.

Ayrıca tezin hazırlanma ve basım aşamalarındaki yardımlarından dolayı Mehmet Dündar AKIN ve Betül AKIN'a teşekkür ederim.

Bugüne kadar hayatımın her aşamasında bana sürekli destek olan aileme, yardımlarından dolayı çok sevgili eşim Sevgi Köse'ye de şükranlarımı bildiririm.

Mayıs 1999 Savaş KÖSE

İÇİNDEKİLER

ÖNSÖZ ii
KISALTMALAR vi
TABLO LİSTESİ vii
ŞEKİL LİSTESİ viii
ÖZET ix
SUMMARY x

1. GİRİŞ 1

2. HESAPLAMA MODELLERİ (COMPUTATIONAL MODELS) 4
2.1 SISD Modeli 4
2.2 MISD Modeli 5
2.3 SIMD Modeli 6
2.4 MIMD Modeli 7
2.5 MIMD Bilgisayarların Programlanması 8
2.6 Donanım ile Paralel Hesaplamada Problemler 8

3. MIMD MODELİNİN GERÇEKLENMESİ 9
3.1 DCOM'un Seçilmesinin Nedenleri 9

4. ALTYAPI : DCOM , DAĞITIK NESNELER MODELİ 10
4.1 Gerçek Bir Sistem Nesne Modeli 11
4.1.1 Evrensel Tekil Tanımlayaıcılar (Globally Unique Identifiers) 11
4.1.2 Yazılan Kodların Yeniden Kullanımı ve Inheritance 12
4.1.3 Tekil Programlama Modeli 12
4.1.4 Yaşam Döngüsü 12
4.1.5 Güvenlik 13
4.1.6 Dağıtık Yapı 13
4.1.7 Nesneler ve Arayüzler 13
4.1.7.1 Arayüzlerin Özellikleri 14
4.1.7.2 Nesnelerin Resimleri 14
4.1.7.3 QueryInterface (Arayüz Sorgulama) 16
4.1.8 İstemci, Sunucu ve Nesne Gerçekleyicileri 17
4.1.8.1 Sunucular : In-Process ve Out-Of-Process 17
4.1.8.2 Saydamlık 18
4.1.9 COM Kütüphanesi 19
4.2 Nesneler ve Arayüzler 20
4.2.1 Arayüzler 20
4.2.2 Arayüzlerin tanımlanması ve Tanımlayıcısı 20
4.2.3 Arayüz Tanımlama Dili – IDL 21
4.2.4 Evrensel Tekil Tanımlayıcılar 21
4.2.5 IUnknown Arayüzü 22
4.3 COM Programları 23
4.3.1 Bellek Kullanımı 23

4.4 COM İstemcileri....24
4.4.1 Nesne Sınıflarının Belirlenmesi....24
4.4.2 Nesnelerin Yaratılması....25
4.4.2.1 Sınıf Fabrikası : IClassFactory Arayüzü....25
4.4.3 Bir CLSID için Sınıf Fabrikası Nesnesinin Oluşturulması....26
4.4.4 Nesnenin Başlatılması....28
4.4.5 Nesnenin Kullanılması....28
4.4.6 Nesnenin Serbest bırakılması....28
4.5 COM Sunucuları....29
4.5.1 Nesne Sınıflarının Tanımlanması ve Sisteme Kayıt Edilmesi....29
4.5.2 Sınıf Fabrikasının Gerçeklenmesi....31
4.5.3 Nesne Fabrikasının Kayıt Edilmesi....33
4.5.4 Sunucunun Kapatılması....35
4.5.5 Nesne Tekrar Kullanımı....36
4.6 Arayüz Erişimi....37
4.7 Güvenlik....39

5. SUNUCU VE İSTEMCİLERİN PROGRAMLANMASI....41
5.1 COM Sunucularının Programlanması....41
5.1.1 Arayüz Tanımlaması ve Nesne sınıfının gerçeklemesi....42
5.1.2 Sunucu nesnesinin kayıt edilmesi....46
5.2 COM İstemcilerinin Programlanması....47

6. GENEL PARALELLEŞTİRME ALGORİTMASI....52

7. ÖRNEK UYGULAMA VE SONUÇLARI - QUICKSORT İŞLEMİNİN PARALEL OLARAK GERÇEKLENMESİ....55
7.1 QuickSort Algoritması....55
7.2 QuickSort Algoritmasının Paralel İşleyişi....56
7.3 Sonuçlar....57
7.4 Karmaşıklık Hesabı....58

8. MODELİN GÖRÜNTÜ İŞLEMEYE UYGULANMASI....60
8.1 Genel Model Tasarımı....60
8.2 Sorunlar....63
8.2.1 Örtüşme Problemi....63
8.2.2 Bölme problemi....64
8.2.3 Meşgul bekleme problemi....65
8.3 Genel Karmaşıklık Hesabı....66
8.4 Hough Dönüşümü....67
8.4.1 Paralel Hough Dönüşümünün İşleyişi....68
8.4.2 İstemci ve Sunucu Nesnelerinin Programlanması....70
8.4.3 Hough Sunucu Nesnesi....70
8.4.3.1 ICHough Arayüzünün Tanımı....71
8.4.3.2 TCHoughServer Sunucusunun Tanımı....71
8.4.4 Hough İstemci Programı....74
8.4.4.1 Sunucu Nesnelerinin Dinamik Olarak Oluşturulması....74
8.4.4.2 Alt İşlemlerin Oluşturulması....75
8.4.4.3 Genel Paralelleştirme İşlemi....77
8.4.5 Karmaşıklık Hesabı....78

8.4.6 Sonuçlar....80
8.5 Parmak İzi Analizi....82
8.5.1 Algoritmanın Paralel İşleyişi....86
8.5.2 İstemci ve Sunucu Nesnelerinin Programlanması....86
8.5.3 Finger Sunucu Nesnesi....86
8.5.3.1 ICFinger Arayüzünün Tanımlanması....87
8.5.3.2 TCFingerServer Sunucu Nesnesinin Tanımı....87
8.5.4 Finger İstemci Programı....89
8.5.5 Karmaşıklık Hesabı....90
8.5.6 Sonuçlar....91

9. SONUÇLAR VE TARTIŞMA **93**

KAYNAKLAR **96**

EKLER **98**

ÖZGEÇMİŞ **105**

TABLO LİSTESİ

		Sayfa No
Tablo 4.1.	Sunucu kayıt anahtarları ve değerleri	30
Tablo 7.1.	Paralel QuickSort algoritmasının test sonuçları	57
Tablo 7.2.	QuickSort'un Karmaşıklığı.	59
Tablo 8.1.	Değişik boyuttaki resimler için karmaşıklık hesapları	80
Tablo 8.2.	Paralel hough dönüşümü test sonuçları	81
Tablo 8.3.	Paralel parmak izi analizi test sonuçları	92

KISALTMALAR

COM	: Component Object Model
DCOM	: Distributed Component Object Model
SISD	: Single Instruction stream, Single Data stream
MISD	: Multiple Instruction stream, Single Data stream
SIMD	: Single Instruction stream, Multiple Data stream
MIMD	: Multiple Instruction stream, Multiple Data stream
CORBA	: Component Object Request Broker Architecture
GUID	: Globally Unique Identifier
RPC	: Remote Procedure Call
IDL	: Interface Definition Language
CLSID	: Class Identifier
TLB	: Typle Library

ŞEKİL LİSTESİ

		Sayfa No
Şekil 2.1	: SISD Bilgisayarı	4
Şekil 2.2	: MISD Bilgisayarı	5
Şekil 2.3	: SIMD Bilgisayarı	6
Şekil 2.4	: MIMD Bilgisayarı	7
Şekil 4.1	: 1 A,B ve C arayüzlerini destekleyen bir nesnenin resmi	15
Şekil 4.2	: Nesneye bağlanmış bir istemci	15
Şekil 4.3	: Nesne gösterimi	15
Şekil 4.4	: QueryInterface işlemi	16
Şekil 4.5	: COM Çağrı Yapısı. COM çağrıları saydam bir şekilde yapar	19
Şekil 4.6	: Arayüz fonksiyon işaretçileri	20
Şekil 4.7	: Nesne yaratılması	25
Şekil 4.8	: DLL sunucusundan nesne oluşturulması ve COM çağrıları	33
Şekil 4.9	: EXE sunucusundan nesne oluşturulması ve COM çağrıları	35
Şekil 4.10	: Containment ile nesnelerin tekrar kullanımı	36
Şekil 4.11	: Aggregation ile nesnelerin tekrar kullanımı	37
Şekil 4.12	: İstemci ve Sunucu bağlantısı	38
Şekil 4.13	: İstemci sunucu bağlantısının kurulması	38
Şekil 4.14	: İstemci ve sunucu arasındaki arayüz erişimi	39
Şekil 5.1	: Otomasyon Nesne Sihirbazı	42
Şekil 5.2	: Arayüz tanımlama editörü	43
Şekil 5.3	: COM sunucularının konfigürasyon ayarları	47
Şekil 6.1	: İstemci sunucu modeli	52
Şekil 6.2	: Genel Paralelleştirme Algoritması	54
Şekil 7.1	: QuickSort algoritmasının işleyişi	55
Şekil 7.2	: QuickSort algoritmasının paralel işleyişi	56
Şekil 8.1	: NxN boyutundaki resmin dxd boyutunda alt resimlere bölünmesi	61
Şekil 8.2	: İşlenecek olan alt resim numaralarını tutan iş kuyruğu	61
Şekil 8.3	: Alt işlemlerin paralel sunuculara dağıtılması	62
Şekil 8.4	: Sunuculardan cevapların alınması ve yeni işlerin dağıtılması	62
Şekil 8.5	: Örtüşme problemi	63
Şekil 8.6	: Örtüşme problemi	64
Şekil 8.7	: Bir Çemberin gösterimi	67
Şekil 8.8	: Resmin alt resimlere bölünmesi	68
Şekil 8.9	: Alt işlemleri tutan kuyruk	69
Şekil 8.10	: Alt resimlerin hough nesnelerine dağıtılması	69
Şekil 8.11	: Hough nesnelerinden sonuçların alınması ve yeni işlerin gönderilmesi	70
Şekil 8.12	: Örtüşme problemi	70
Şekil 8.13	: Hough sunucu nesnesi ve arayüzleri	71
Şekil 8.14	: Galton karakteristikleri	83
Şekil 8.15	: Örnek bir parmak izi ve özel noktalar	84
Şekil 8.16	: Parmak izi temizleme aşamaları	85
Şekil 8.17	: Finger sunucu nesnesi ve arayüzleri	86

DAĞITILMIŞ NESNESEL BİRİM MODELİ ÜZERİNDE GÖRÜNTÜ İŞLEME ALGORİTMALARININ PARALELLEŞTİRİLMESİ

ÖZET

Görüntü işleme ve örüntü tanıma işlemleri çok fazla karışık ve zaman alan işlemlerdir. Bu işlemlerin yapılabilmesi için çok hızlı hesaplama yapabilen modellere ihtiyaç vardır. Özellikle örüntü tanıma işlemleri çok fazla hesaplama gerektirirler. Bu işlemleri yeteri derecede hızlı yapabilmek için özel donanımlar üzerinde çalışan yine özel algoritmalar tasarlanmaktadır. Bu projede , özel bir donanım kullanmadan varolan donanım ve yazılım sistemini kullanarak paralel hesaplama yapan bir sistem geliştirilmiştir. Yani MIMD hesaplama modelinin yazılım ile gerçeklemesi yapılmıştır ve bu modelin özel olarak görüntü işleme alanına uygulaması yapılmıştır.

Ağ performanslarının çok gelişmesi ve veri iletişim hızlarının artması, dağıtık nesneleri kullanarak yazılım geliştirmeyi olanaklı kılmıştır. Ancak yinede ağ ortamında tüm gereksinimleri karşılamak için çok fazla zaman harcamak gereklidir. COM (Component Object Model) Modeli , remote invocation , sürüm kontrolü, yük dağılımı, hata giderme gibi işlemlerin yapılabilmesini sağlayan bir alt yapı oluşturur. Distributed COM (DCOM), ise COM sistemini bir adım daha ileri taşır. Ağ ortamında nesnelerin birbiri ile konuşabilmesi ve dağıtık nesnelerin oluşturulabilmesi için gerekli altyapıyı sağlar. DCOM aynı ağda farklı bilgisayarlarda olan nesnelerin birbirleriyle doğrudan konuşabilmesine olanak sağlayan bir yapıdır. Aynı zamanda farklı makinelerde çalışan nesnelerin paylaşımına olanak sağlar. Bunun anlamı, bir program herhangi bir nesneyi program yada DLL olarak yaratıp, bu nesnenin metotlarını başka bir makinedeki başka bir programdan çağırabilir. Bu teknolojinin en önemli özelliklerinden biri iş yükünün değişik makinelere dağıtılarak paralel bir şekilde daha hızlı hesaplama yapabilmesidir.

PARALLELIZATION OF IMAGE PROCESSING ALGORITHMS USING DISTRIBUTED COMPONENT OBJECT MODEL

SUMMARY

Algorithms related with image processing and pattern recognition are usually complex and time consuming. High-speed computation is needed especially in pattern recognition applications. In order to achieve this, parallel algorithms are being developed on special purpose hardware. This project presents a new method for doing parallel computation without using special purpose hardware. That's software implementation of MIMD computers running on hardware, operating system and compilers.

Because of advances in high-speed networking, building software as distributed object applications has become preferable but still a considerable amount of time is spent to meet computational demands on the network. Component Object Model (COM) provides an architecture that supports some basic features as remote invocation, versioning, load balancing and fault tolerance. Distributed COM (DCOM) is the distributed extension of COM. It specifies the additional infrastructure needed to further extend the benefits of networked environments. DCOM is a structure that allows applications to talk to one another across a network. In particular, it allows for sharing of objects that reside on two separate machines. This means that one can create an object in one application or DLL, then call the methods of that object from an application that resides on a different computer. One of the most important properties of this technology is that it allows for the distribution of the load of a task across several machines. That way the processor will not have to expend excessive clock cycles on the task, and large database loaded onto memory will be separated across machines.

1. GİRİŞ

Projede dağıtılmış nenesel birim modeli kullanılarak genel olarak görüntü işleme algoritmalarının paralelleştirilmesi gerçekleştirilmiştir. Dağıtık nesne modeli olarak DCOM (Distributed Component Object Model) [1] seçilmiştir. DCOM'un seçilmesinin nedenleri 3. bölümde anlatılmıştır.

İlk olarak hesaplama modelleri [2] anlatılmıştır. Bu modellerden SIMD (Single Instruction Stream Single Data Stream) modeli dışında diğer modeller paralel işlem yapabilme yeteneğine sahiptirler. SIMD modeli sadece tek bir komut ve veri kümesine sahiptir, dolayısı ile bir anda sadece belli bir veri kümesi üzerinde bir tek işlem gerçekleştirir. MISD, SIMD , MIMD modelleri çeşitli seviyelerde paralel işlem yapabilirler. Bu hesaplama modelleri 1. bölümde anlatılmıştır. MIMD modelinin programlanması ve donanım olarak gerçeklenmesinde karşılaşılan problemler anlatılmıştır. Paralel hesaplama modellerinin donanım ile gerçeklenmesinde özel donanım, özel işletim sistemi ve sunulan servisleri kullanabilen özel derleyicilere ihtiyaç vardır.

Projenin amacı , en iyi paralel işleme modeli olan MIMD modelini yazılım ile gerçeklenmesidir. Donanım ile gerçeklemenin aksine yazılım ile gerçeklemede yukarıda bahsedilen problemler yoktur. Yazılım ile paralel hesaplama modelinde elde mevcut olan donanımlar, işletim sistemi ve derleyiciler kullanılmıştır. Ancak burada dağıtık nesneler arası iletişimi sağlayacak özel bir yapı kullanılmıştır. Böylelikle özel donanım , özel işletim sistemi ve derleyicilere gerek olmadan paralel hesaplama yapabilme olanağı sağlanmıştır. Nesneler arası iletişim için DCOM modeli seçilmiştir. Bunun nedeni , DCOM modelinin Windows sistemlerinde yüklü olarak gelmesi ve dağıtık nesne kullanımı için gerekli tüm fonksiyonları içermesidir. Yazılım ile paralel hesaplama ve DCOM'un seçilmesinin nedenleri 3. bölümde anlatılmıştır.

DCOM (dağıtık nesneler modeli) dağıtık nesnelerin tanımlanması, oluşturulması , nesnelerin başlatılıp kapatılmasına kadar takip edilmesi , nesneler arası veri iletişiminin sağlanması gibi bir dizi işlemi gerçekleştiren bir modeldir. Tüm nesneler belirli arayüzleri (fonksiyon gurupları) gerçekleştirerek tanımlanırlar. Bu nesnelere erişim ve kullanım da yine bu arayüzler sayesinde yapılır. Bir nesne birden fazla

arayüzü gerçekleyebilir dolayısı ile birden fazla fonksiyona sahip olabilir. Nesne sınıfları ve arayüzleri tekil tanımlayıcılara sahiptirler. Ağ üzerinde bu tanımlayıcılar kullanılarak nesneler bulunur ve çalıştırılır. Tekil tanımlayıcılar evrensel olup sadece bir nesne bir tek tanımlayıcıya sahiptir. Nesneler istemci programı tarafından çalıştırıldıkları yere göre üç kısma ayrılırlar. Birinci tipte sunucular (nesnelere aynı zamanda sunucu nesneleri de denir) istemci programı içinde bir DLL gibi çalışan program parçacıklarıdır, bunlara In-Process sunucu denir. İkinci tip sunucular istemci programının dışında çalışırlar, Out-of-Process. Üçüncü tip sunucular istemcinin çalıştığı makinede değil de uzak bir makinede çalışan nesnelerdir, bunlara Remote-Process denir. Sunucu nesnelerinin yazılmasında bazı kurallara uyulması gerekmektedir. Bu sunucu nesnelerini kullanan istemci programları ise normal bir uygulama gibi yazılır. Yani DCOM modelinde bir kere sunucu nesnelerini doğru olarak oluşturduktan sonra istemci programlar tarafından çok rahat bir şekilde kullanılabilirler. Bu nesnelere bağlantı , veri iletişimi gibi sorunlarla istemci uğraşmaz. Tüm bu sorunlarla DCOM kütüphanesi ilgilenir. Bu da sistemin saydam olmasından kaynaklanmaktadır. DCOM modeli 4. bölümde ayrıntılı olarak incelenmiştir.

Beşinci bölümde sunucu ve istemci programlarının yazılması anlatılmıştır. DCOM sunucu nesnesi tanımlanmadan önce arayüz tanımları yapılmalıdır. Daha sonra bu arayüzü kullanan sunucu nesne sınıfı tanımlanır ve arayüzdeki fonksiyonlar gerçeklenir. Sunucu programı bu şekilde oluşturulduktan sonra istemciler tarafından kullanılmak üzere sisteme kayıt edilir. İstemci programı da kullanacağı sunucu nesnesinin arayüz tanımlarını bilmesi gerekir. Aksi taktirde bu sunucunun fonksiyonlarına erişemez.

Dağıtık nesne modeli kullanılarak MIMD hesaplama modelinin yazılımla gerçeklenmesi için genel paralelleştirme algoritması oluşturulmuştur. Bu algoritmada sıralı olarak yapılan bir iş önce alt işlemlere ayrılır ve bu işlemler farklı bilgisayarlarda paralel olarak çalışan sunucu nesnelerine dağıtılarak daha hızlı çözüme ulaşılır. Genel paralelleştirme algoritması sorunsuz bir şekilde alt işlemlere bölünebilen her türlü probleme uygulanabilir. Ancak burada görüntü işleme alanına uygulanmıştır. Görüntü işleme algoritmaları küçük bir veri üzerinde çok fazla işlem gerektirdiklerinden dolayı bu yapıya çok uygundurlar. Altıncı bölümde genel paralelleştirme algoritması anlatılmıştır. Daha sonra bu yapıyı kullanan örnek bir uygulama verilmiştir. QuickSort (Hızlı sıralama) algoritması paralel olarak gerçeklenmiştir ancak sıralıya göre çok daha yavaş işlediği görülmüştür. Bunun nedeni ise QuickSort işleminde çok fazla veri transferi olmasıdır.

Bu model az miktarda veri üzerinde çok fazla işlem yapan algoritmalarda çok daha iyi sonuçlar vermektedir. Bunun nedeni nesneler arası veri transferinin çok fazla olmasındandır. Paralel nesneler farklı makineler üzerinde çalıştığı için ağ üzerinde veri iletişi yapılmaktadır. Ağ transfer hızı ve bant genişliği gibi bir çok etken bu transfer işlemini etkilemektedir. Dolayı ile ne kadar çok veri transfer işlemi varsa algoritma okadar yavaş çalışacaktır. Görüntü işleme algoritmaları, çok fazla işlem gerektirdiğinden ve işlenen görüntü alt görüntülere kolayca ayrılabildiğinden bu modele uygundur. Son bölümde yazılımla gerçeklenen MIMD modelini kullanarak görüntü işleme algoritmalarının paralelleştirilmesi iki uygulamayla anlatılmıştır. İlk olarak Hough dönüşümü algoritması paralel olarak gerçeklenmiştir. Hough dönüşümü işlemi verilen bir resimde çeşitli şekillerin (hat, çember...) bulunmasında kullanılır. Paralel olarak yapılan hough dönüşüm işleminin sıralı işleme göre çok daha hızlı çalıştığı görülmüştür. İkinci uygulama parmak izi analiz işlemidir. Parmak izi analizi işlemi kötü durumda olan bir parmak izi görüntüsünün bir dizi işlemlerden geçirilerek özelliklerinin belirlenmesidir. Bu uygulamada da başarılı sonuçlar elde edilmiştir.

2. HESAPLAMA MODELLERİ (COMPUTATIONAL MODELS)

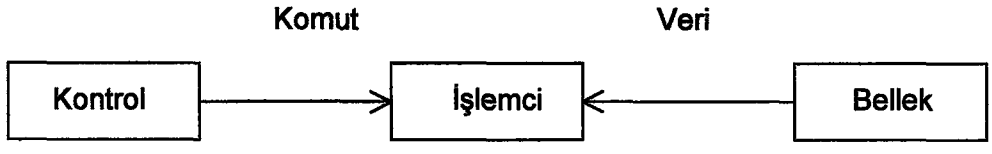
MIMD [2] modelinin yazılım ile gerçeklemesine geçmeden önce genel olarak hesaplama modelleri incelenecektir. Veriyi ve komutları işleyiş biçimine göre dört tane hesaplama modeli vardır. Bu modeller içinde SISD modeli hariç diğer modeller paralel işlem yapma özelliğine sahiptir. Bu modeller içinde en etkin biçimde paralel işlem yapabilen model ise MIMD modelidir. Tabi en zor gerçeklemeye sahip olan modelde yine bu modeldir.

Hesaplama modelleri başlıca dört gurupta toplanır:

- (SISD) Single Instruction stream, Single Data stream.
- (MISD) Multiple Instruction stream, Single Data stream.
- (SIMD) Single Instruction stream, Multiple Data stream.
- (MIMD)Multiple Instruction stream, Multiple Data stream.

2.1 SISD Modeli

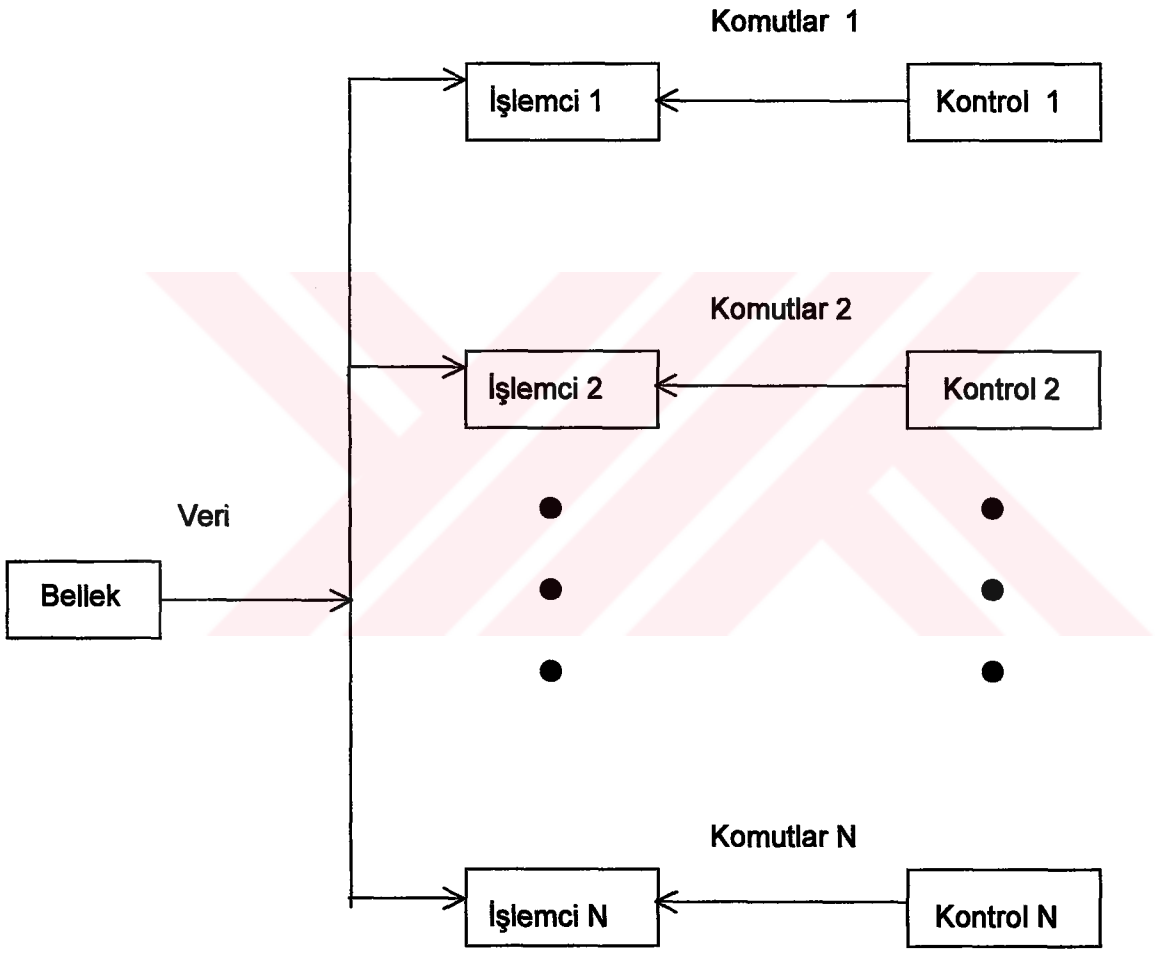
Bu bilgisayarlar bir tek veri kümesi üzerinde işleyen ve bir tek komut kümesinden komutları alan işlemciden oluşur. Şekil 2.1'de de gösterildiği gibi Kontrol birimi ve Bellek birer tanedir. Bu hesaplama modelinde hiç paralellik yoktur. Ardışık işlemlerin yapılması için uygun bir modeldir.



Şekil 2.1 SISD Bilgisayarı.

2.2 MISD Modeli

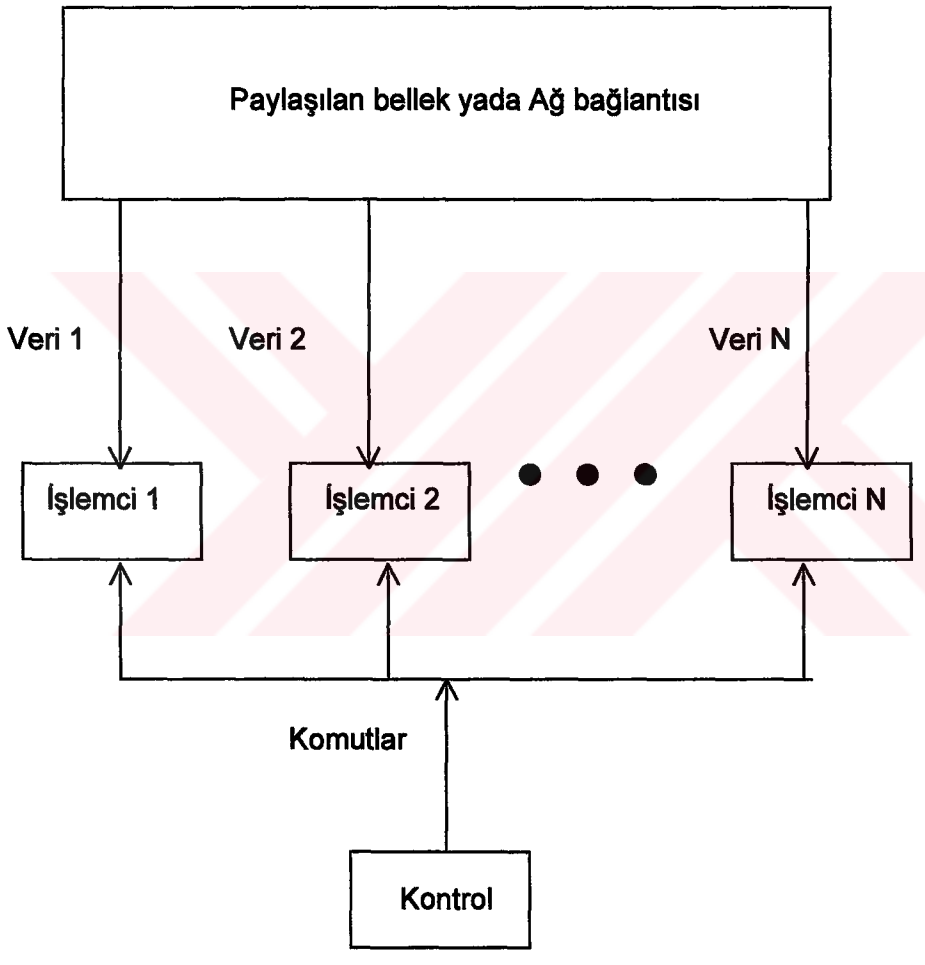
Bu bilgisayarlarda her biri kendi kontrol birimine sahip ortak bir belleği paylaşan N tane işlemci birimi vardır. Dolayısı ile bu yapıda N tane komut kümesi ve bir tane veri kümesi vardır. Her işlemci aynı veri üzerinde değişik işlemler yaptıkları için , bu bilgisayarlar paralel olarak çalışabilirler.



Şekil 2.2 MISD Bilgisayarı.

2.3 SIMD Modeli

Bu bilgisayarlar her biri kendi belleğine sahiptir. Fakat bir tek kontrol biriminden komutları alan N tane işlemciye sahiptir. Her bir işlemci değişik veriler üzerinde aynı komutları çalıştırırlar. Bu yapıda işlemcilerin birbirleriyle haberleşmesi gerekmektedir. Bunun için paylaşılan bellek ve ağ bağlantısı kullanılır.

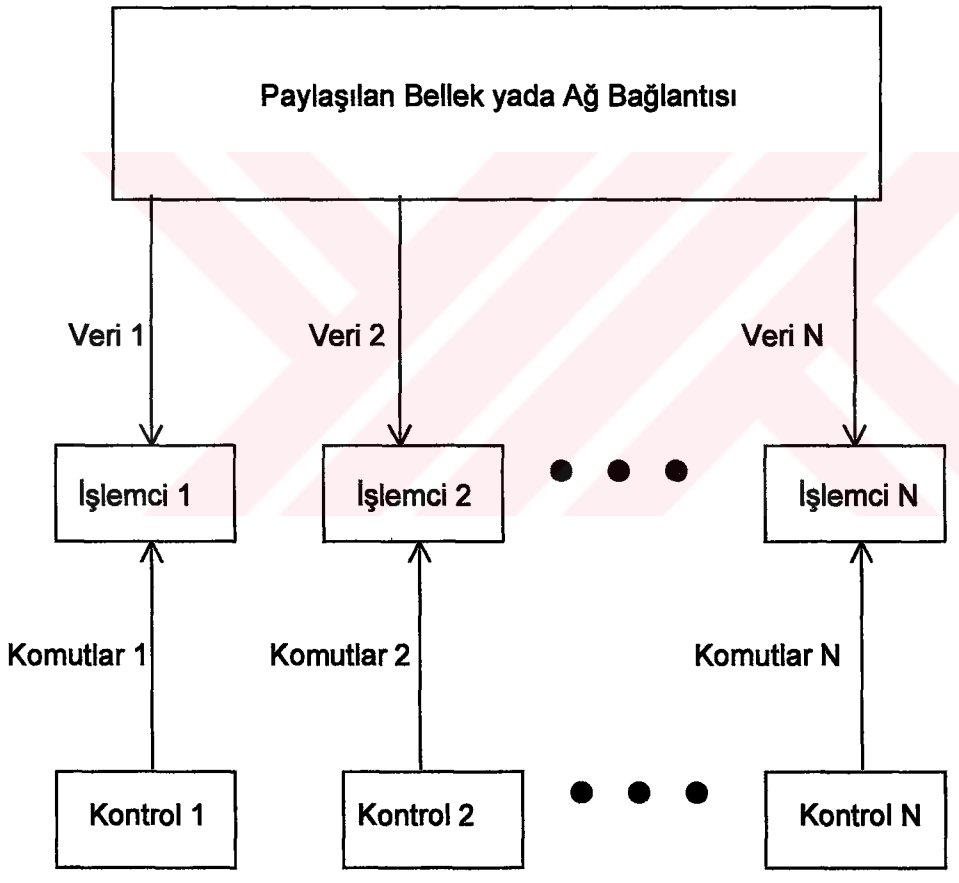


Şekil 2.3 SIMD Bilgisayarı.

2.4 MIMD Modeli

Bu bilgisayarlar en genel ve paralel işlem gücü en yüksek olan bilgisayarlardır. Bu modelde her biri kendi belleğine ve kendi komut kümesine sahip N tane işlemci vardır. Her bir işlemci değişik veriler üzerinde değişik komutları çalıştırırlar.

SIMD bilgisayarlarında olduğu gibi burada da haberleşme paylaşımlı bir bellek yada ağ bağlantısı ile sağlanır.



Şekil 2.4 MIMD Bilgisayarı.

2.5 MIMD Bilgisayarların Programlanması

Eşzamansız (Asynchronous) [2] algoritmaları incelemek , tasarlamak ve gerçeklemek çok zordur. MIMD bilgisayarlarındaki yapıyı iyi anlayabilmek için işlem (process) ve işlemci (processor) farkını iyi anlamak gerekir. Eşzamansız algoritma , tümü yada bir kısmı aynı anda uygun olan işlemcilerde çalıştırılan , işlemlerden oluşur. Başlangıçta tüm işlemciler boştur. Paralel algoritma herhangi bir işlemcide çalışmaya başlar ve bir çok yeni işlem oluşturur ve bu işlemler boşta olan diğer işlemcilere dağıtılır ve paralel çalışma sağlanır. Eğer boş işlemci yoksa işler daha sonra işlenmek üzere kuyruğa atılır.

2.6 Donanım ile Paralel Hesaplamada Problemler

Yukarıda anlatılan paralel modellerin tümünün gerçeklenmesi olanaklı değildir. Mesela MIMD modeller bir çok uygulamada son derece yararlı olabilirler. Bu modeller üzerinde verimli bir şekilde yapılacak olan işlemler de özel işlemlerdir. Çoğu uygulamalar için MISD bilgisayarların da kullanımı zor ve imkansız olabilir.

Burada birbiriyle haberleşmesi gerekli N işlemci vardır. Bunu sağlamak için tüm işlemcileri birbiriyle bağlayabiliriz. Fakat işlemci sayısını çok fazla olduğu bir sistemde bu olanaklı görünmemektedir. Aynı zamanda da maliyeti çok yüksek olacaktır.

Uygulamaları bu sistemlerle paralel olarak yapabilmek için, özel donanıma, işletim sistemine ve bu özel donanımın ve sistemin sunduğu imkanları en iyi şekilde kullanılabilmesi için derleyicilere ihtiyaç vardır. Ayrıca özel algoritmalar geliştirilmesi gereklidir.

3. MIMD MODELİNİN GERÇEKLENMESİ

Daha önce de belirtildiği gibi MIMD modeli hesaplama modellerinin içinde paralel işleme en uygun olan sistemdir. Özel bir donanım olmadan da MIMD modeli gerçeklenip çeşitli uygulama alanlarında kullanılabilir. Elde var olan donanım ve sistemler kullanılarak bu model gerçeklenebilir. Yani sadece yazılım kullanarak ta paralel hesaplama modelleri gerçekleştirilebilir. Bunun için ağ ortamında dağıtık nesneler kullanılır. Alt yapı olarak herhangi bir yapı seçilebilir, bunların içinde en önemli olanlar CORBA [3-4] ve DCOM [1,3] sayılabilir. DCOM CORBA'ya göre çok daha yeni bir teknolojidir. CORBA çok daha önceden beri kullanıldığı için bir çok sorun çözülmüştür fakat DCOM ileride de gösterileceği gibi hala tam olarak oturmamış bir sistemdir ancak bu sistemi gerçekleyebilmek için her türlü fonksiyona sahiptir.

Alt yapı olarak bu sistemleri kullanılabileceği gibi özel bir altyapı oluşturulabilir. Yani dağıtık nesnelerin birbiri ile doğrudan , hatasız bir şekilde konuşabildikleri bir sistem gerçeklenebilir. Fakat bu altyapıyı hazırlamak zaten başlı başına bir sorun olduğu için bu projede DCOM alt yapı olarak seçilmiştir.

3.1 DCOM'un Seçilmesinin Nedenleri

Ağ ortamında nesnelerin haberleşmesi ve veri iletişimi için DCOM seçilmiştir. Bunun başlıca nedeni daha önce de belirtildiği üzere alt yapının baştan yazılmasının başlı başına bir sorun olmasıdır. DCOM zaten dağıtık işlem için gerekli tüm fonksiyonları sağlamaktadır. Böylece projede dağıtık nesneleri birbiri ile konuşturmak ve alt yapı ile uğraşmaktan çok projenin asıl amacı için çalışmalara ağırlık verilmiştir.

CORBA'ya karşı DCOM'un [3] tercih edilmesinin nedeni ise CORBA'nın ayrı bir uygulama olarak gelmesidir. Projede sunucu olarak WindowsNT ve istemci olarak Windows95 sistemleri kullanılmıştır. DCOM bu sistemlerde hazır olarak gelir.

4. ALTYAPI : DCOM , DAĞITIK NESNELER MODELİ (DISTRIBUTED COMPONENT OBJECT MODEL)

DCOM , COM [5] modelinin dağıtılmış halidir. Bir COM sunucusu bir çok değişik nesne sınıfının örneklerini (instance) yaratabilir. Bir COM nesnesi bir den fazla interface (arabirim) destekleyebilir. Bunların her biri nesnenin değişik davranışlarını belirlerler. COM istemcisi COM nesnelerine erişebilmek için, bu nesnenin arabirimlerine bir işaretçi alır ve sanki nesne kendi adres uzayındaymış gibi metot çağrılarını yapar. Interface (arabirim, arayüz) nesnenin işlevlerini belirten metotlar topluluğunun adıdır. COM nesnelerine sadece bu arabirimler sayesinde erişim yapılabilir. Bu arabirimlerin kullanılması ile COM önemli avantajlar sağlar:

Uygulamaların ileride geliştirilmesinde sağladığı kolaylık : Bu özellik QueryInterface (Arayüz sorgulama) servisinin kullanılmasından gelir. Bu servis sayesinde bir nesne hali hazırda bulunun arabirimlerine dokunmadan yeni arabirimler de destekleyebilirler. Yani bir sunucu nesnesine yeni bir arabirim eklendiğinde bu nesneyi kullanan istemcilerde hiçbir değişiklik yapmaya gerek kalmadan, hatta yeniden derlemeden , çalışmalarına devam ederler. Çünkü COM nesnelerin birden fazla arabirim desteklemesine olanak sağlar. Böylece sürüm sorunları da ortadan kalkar. COM nesneleri birden fazla sürümü içerebilirler.

Aynı işlem içinde nesnelerin çok hızlı haberleşmesi: Bir istemci bir COM nesnesine bağlantı kurduktan sonra , bu nesnenin metotlarına erişim iki tane indirect (dolaylı) işaretçi üzerinden yapılır. Bu yüzden aynı adres uzayında kullanılan ve çağrılan nesneler için çağrı süresindeki zaman kaybı çok fazla değildir.

Nesnelerin konumsal saydamlığı: İkili standart yapısında olduğu için COM, nesnelerin metotlarına yapılan çağrıları kendisi alarak, RPC(Remote Procedure Call) [6] şeklinde gerçek nesneye yönlendirir. Bu nesne başka bir adres uzayında ve hatta başka bir makinede olabilir. Burada önemli olan şey istemcinin bundan hiç haberi olmadığıdır. İstemci sanki nesne kendi adres uzayındaymış gibi metotları çağırır, gerisini COM kendi içinde halleder. COM ikili ve ağ standartları sayesinde bu in-process ve cross-network çağrılarını saydam bir şekilde yapar. Böylece tüm nesneler istemciye tek bir tipte saydam şekilde görünürler.

Programlama dilinden bağımsızlığı: COM ikili (binary) standarda sahip olduğu için nesneler herhangi bir programlama dilinde yazılıp derlenebilirler. Sunucu nesneleri ve bu nesneleri kullanan istemcilerin aynı dilde yazılması gerekmez. İşaretçilerle çalışabilen, işaretçilerle fonksiyonları çağırabilen herhangi bir programlama dili kullanılabilir. Örnek olarak C, C++, Pascal, Ada, Smalltalk ve hatta Basic bile kullanılabilir.

4.1 Gerçek Bir Sistem Nesne Modeli

Gerçek bir sistem modeli olabilmesi için, bir nesne yapısı, dağıtık ve gelişebilen , milyonlarca nesneyi hatasız bir şekilde destekleyebilen bir sisteme sahip olması gerekir. COM tüm bunları sağlayan bir yapıdadır. COM gerçek bir sistem nesne modelidir, çünkü:

- COM nesne sınıflarını ve arabirimlerini sorunsuz olarak tanımlamak için GUI (globally unique identifiers) evrensel tekil tanımlayıcıları kullanır.
- Önceden yazılan kodların tekrar kullanımının problemsiz bir şekilde yapılmasını sağlar.
- In-process (aynı adres uzayında), cross-process (farklı adres uzayında) ve cross-network (farklı makinelerde) modelleri için tekbir programlama modeli oluşturur.
- Nesnelerin yaşam döngüsünü reference counting (erişim sayacı) sayesinde takip eder.
- Nesne seviyesinde çok esnek bir güvenlik mekanizması sağlar.

4.1.1 Evrensel Tekil Tanımlayıcılar (Globally Unique Identifiers)

Dağıtık sistemler milyonlarca arabirim ve yazılıp bileşenleri içerebilirler. Bunların tümü tekil olarak tanımlanmalıdırlar. Nesneleri okunabilir isimlerle tanımlayan bir sistem çok riskli bir sistemdir. Çünkü birden fazla nesneye yanlışlıkla aynı isim verilebilir ve sistemde çakışmalar oluşabilir.

COM nesneleri tanımlayabilmek için GUID (Globally Unique Identifiers)'ler kullanır. Bir evrensel tekil tanımlayıcı 128 bitten oluşan bir sayıdır. Her nesnenin farklı bir GUID numarası vardır. Aynı zamanda nesnelere okunabilir isimlerde verilir fakat bunlar yerel isimlerdir. Bu şekilde COM milyonlarca nesne içeren bir sistemde istenen nesneye doğru bir şekilde bağlanabilmektedir.

4.1.2 Yazılan Kodların Yeniden Kullanımı ve Inheritance

Uygulama geliştirmede, inheritance (türeme) çok önemli avantajlar sağlar. Inheritance bir nesnenin bir başka nesneden türemesi ve onun bazı fonksiyonlarını alarak bazılarını da değiştirmesi manasına gelir. Geleneksel sistemlerdeki inheritance sistemindeki en büyük problem, nesneler arasındaki arabirimlerin yada kontratın kesin olarak belirtilmemesidir. Mesela ana ve çocuk nesneleri değiştiğinde bunlarla bağlantılı olan nesnelerin nasıl davranacağı önceden bilinemez.

COM ise kodların tekrar kullanımı için iki yeni sistem kullanmaktadır comtainment/delegation ve aggregation. Birincisi bir nesnenin başka bir nesnenin istemcisi olma durumudur. Sadece o nesnenin metotlarını dışarı sunar. Aggregation sayesinde de COM nesneleri birden fazla arabirim destekleyebilir.

4.1.3 Tekil Programlama Modeli

Geleneksel olarak kullanılan inheritance yapısında in-process, out-of-process ve cross-network nesnelerinin yazılımı için aynı programlama sistemi kullanılmalıdır.

COM arabirim tabanlı bir model olduğu için, in-process ve out-of-process programlama modelleri arasındaki farkları ortadan kaldırmaktadır. Herhangi bir istemci herhangi bir nesne ile çalışabilir, bu nesnenin aynı makinede yada farklı makinelerde olması önemli değildir.

4.1.4 Yaşam Döngüsü

Geleneksel sistemlerde bir nesneni yaşam döngüsü, yaratılmasından silinmesine kadar olan zaman, programlama dili yada programcılar tarafından kontrol edilir. Yani bu sistemlerde nesnelerin yaşam döngüsünü kontrol eden, onları yaratıp silen bir kişi yada bir program kodu vardır.

Fakat dağıtık bir sistemde artık bu kontrolü yapan kişi yada program parçacıkları yoktur. Nesne yaratılması kolaydır, istemcinin nesneye erişim hakları varsa nesne yaratılır. Zor olan nesnenin kullanılmadığını anlamak ve sistemden silmektir. Mesela nesneyi oluşturan istemci bittiğinde bile bu nesneyi kapatamayız çünkü bu istemci bitmeden önce bu nesnenin bir referansını başka bir istemciye vermiş olabilir. Bu büyük bir problemdir. Bir istemcinin nesne ile işi bittiğinde nesnenin sistemden silinmesi gerekir aksi taktirde sistem şişecek ve kilitlenecektir.

Bu sorunu çözmek için COM referans sayacı kullanır. Bu sayaç sayesinde her nesne kendi kendini yok edebilir yada çalışmasına devam eder. Bir istemci bir nesne

yarattığı zaman bu nesnenin referans sayacı 1 olur eğer başka istemcilerde bu nesneyi kullanmaya başlarsa onlar içinde bu sayaç bir artırılır. Herhangi bir anda bir nesneyi kullanan istemci sayısı bu referans sayacında bellidir. İstemcilerin bu nesneyle işleri bittiğinde referans sayacını bir azaltırlar ve böylece nesneyi artık kullanmadıkları anlaşılır. Her nesne bu sayacı kontrol eder ve eğer sıfır olmuşsa kendisini kullanan bir istemci olmadığından kendi kendini yok eder.

4.1.5 Güvenlik

Dağıtık sistemlerde nesnelere ve içindeki bilgilere erişim güvenli bir şekilde olmalıdır. COM bir çok önemli seviyede güvenlik mekanizmasına sahiptir. Birincisi standart işletim sistemi güvenlik düzeyi, yani bir istemcinin bir sunucuyu çalıştırma yetkisinin olup olmadığı. COM nesneleri yaratırken istemcinin işletim sistemince verilmiş haklarına bakar, eğer bu nesneyi çalıştırma yetkisi varsa çalıştırır aksi taktirde çalıştırmaz. COM DCE RPC üzerine kurulduğu için bunun sağladığı güvenlik düzeylerini de içerir.

4.1.6 Dağıtık Yapı

COM dağıtık nesneleri destekler. Bu sayede uygulama geliştiricileri bir uygulamayı değişik parçalara bölerek bir çok makinede çalıştırabilirler. Saydamlık sayesinde tüm ağ tek bir bilgisayarmış gibi görünür.

Dağıtık nesneleri destekleyen single-process nesne modelleri de vardır, ancak bunların hiç biri in-process, out-of-process ve cross-network nesneleri için saydam bir yapı oluşturmaz. COM ise her türlü sistem için saydam bir ortam yaratır ve istemciler kullandıkları nesnelerin nerede ve nasıl çalıştığını bilmeden kullanırlar.

4.1.7 Nesneler ve Arayüzler

Nesne bir sınıfın yaratılmış bir örneğidir. Sınıf bir amaç için bir arada bulunan veri ve bu veriyi işleyen fonksiyon topluluğunun tanımıdır. Amaç bu sınıftan yaratılan nesneleri kullanan istemcilere gerekli servisleri sağlamaktır.

Bir COM nesnesi hafızada normal bir C++ nesnesi gibi saklanır. Fakat onların tersine istemci bu nesneye doğrudan erişime sahip değildirler. İstemciler bu nesnelere daha önceden açıkça belirtilen bir kontrat sayesinde , arayüzler, erişirler.

Arayüz bir biri ile ilişkili fonksiyonlar grubunun tanımı demektir. Bu fonksiyonlara arayüz üye fonksiyonları da denir. Arayüzlerin isimleri standart olarak 'I' harfi ile

başlar. Aslında arayüzlerin gerçek tanımları GUID'lerle yapılır , burada verilen isimler sadece programlama yaparken daha kolay olsun diye verilen isimlerdir.

Arayüzler herhangi bir gerçekleme tanımı yapmazlar. Örnek olarak bir IStack arayüzü düşünülürse, bu arayüz sadece Push ve Pop diye iki fonksiyon ve bu fonksiyonların parametrelerini tanımlar, ama nesne bu arayüzü nasıl gerçekleyeceği konusunda serbesttir, ister dizi ister linkli liste ile yapar.

Bir nesne bir arayüzü gerçeklediğinde, bu arayüzün tanımladığı tüm fonksiyonları gerçekler ve COM a bu fonksiyonlara erişebilmesi için işaretçiler verir. Daha sonra COM bu işaretçileri nesneye erişmek isteyen istemcilere verir.

4.1.7.1 Arayüzlerin Özellikleri

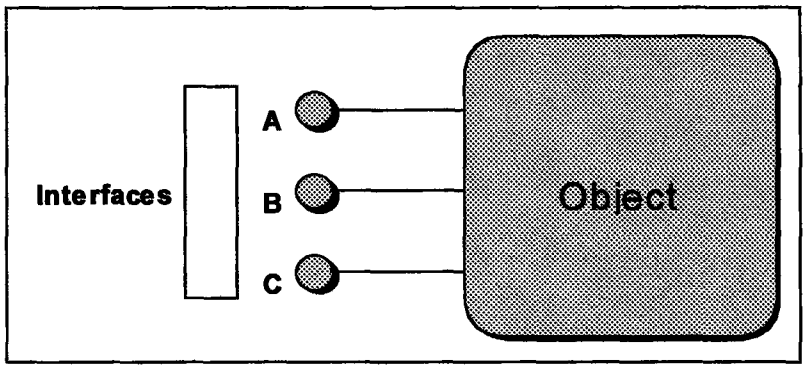
Arayüz bir sınıf değildir. Bir nesne oluşturmak için bir sınıftan bir örnek yaratılabilir. Fakat sadece bir arayüz tanımından bu yapılamaz. Bu arayüzden bir nesne gerçeklenmeli daha sonra bu nesneden bir örnek yaratılmalıdır. Değişik nesneler aynı arayüzü değişik şekillerde gerçekleyebilirler ama yapılan iş değişmez. Arayüzün fonksiyonları istemcinin istediği gibi çalışır. Arayüz bir nesne değildir. Arayüz birbiri ile ilişkili fonksiyon grupları tanımıdır. Nesne ise bu arayüzlerin gerçeklemesidir. Tüm arayüzler kendi GUID numaralarına sahiptirler böylece sistemde çakışma olmaz. Arayüzler değişmez kontratlardır. Arayüzlerde sürüm kavramı yoktur böylece sürüm problemleri de yaşanmaz. Bir arayüzün bazı fonksiyonlarının çıkarılması değiştirilmesi ile oluşturulan arayüzde tamamen farklı bir arayüzdür. Bu yüzden yeni bir arayüz eski bir arayüzle çakışmaz.

İstemciler sadece arayüzlere işaret eden işaretçilerle ilgilenirler. Bir istemci bir nesneye erişim kazanmışsa, o nesnenin arayüz fonksiyonlarına işaret eden bir işaretçi tablosuna sahiptir.

Nesneler birden fazla arayüzü aynı anda gerçekleyebilirler. Mesela bir nesne istemci ile haberleşmek için bir arayüzü gerçekleyebilir aynı zamanda edindiği bilgileri saklayabilmek içinde başka bir arayüzü gerçekleyebilir.

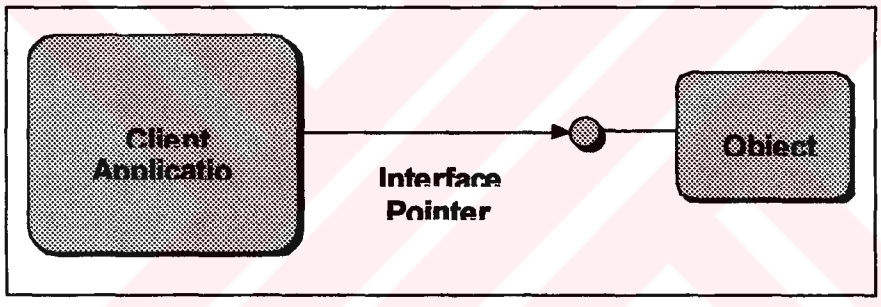
4.1.7.2 Nesnelerin Resimleri

Nesneler ve arayüzler belli bir şekille gösterilebilir. Nesne bir kutu şeklinde çizilebilir nesnenin desteklediği arayüzler bu kutuya soldan yada sağdan giren oklarla gösterilebilir. Şekil 4.1'de örnek bir nesne gösterilmiştir.



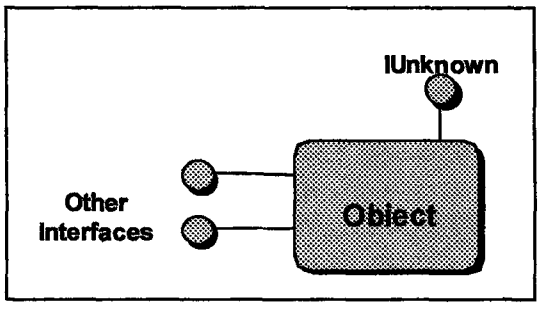
Şekil 4.1 A,B ve C arayüzlerini destekleyen bir nesnenin resmi.

Aşağıdaki şekilde ise nesneye bağlanmış bir istemci ve arayüz işaretçisi gösterilmiştir.



Şekil 4.2 Nesneye bağlanmış bir istemci

Tüm nesnelerin gerçeklemek zorunda oldukları temel bir arayüz vardır. IUnknown. Bu arayüz temel arayüz olup bir nesne oluşturmak için desteklemek şart olan tek arayüzdür. Yani nesne gerçeklenirken bu arayüzünde fonksiyonları gerçeklenmelidir. Bu arayüzüde kullanan bir nesnenin şekli aşağıda verilmiştir.



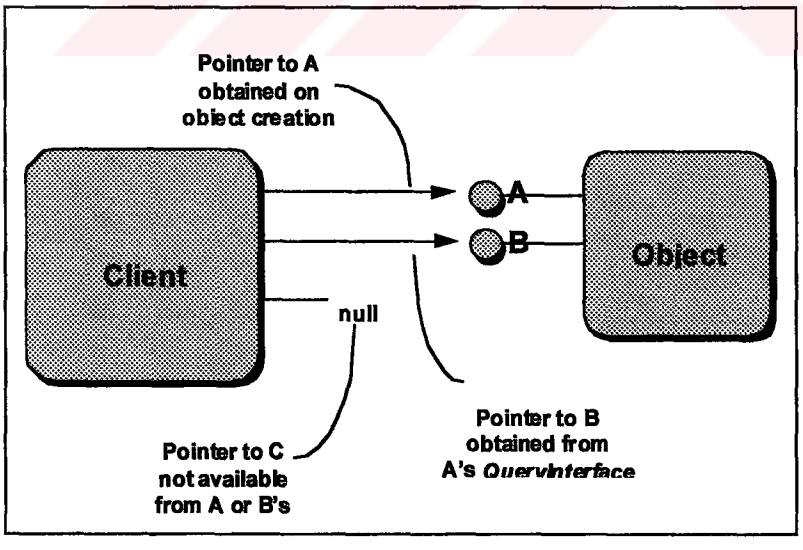
Şekil 4.3 Nesne gösterimi. IUnknown arayüzü nesnenin üzerinden bağlı olarak gösterilir.

4.1.7.3 QueryInterface (Arayüz Sorgulama)

COM da nesneler birden fazla arayüz destekleyebilirler. Yani birden fazla fonksiyon grubuna işaretçi içerebilirler. Fakat bu çok önemli sorunları beraberinde getirir. İstemci bir nesneyi oluşturduğunda kendisine sadece bu nesnedeki bir arayüze işaretçi geri döndürülür. Diğer arayüzlere nasıl erişim yapılacağını bilmek gerekir.

Cevap QueryInterface fonksiyonudur. İstemci bu fonksiyon sayesinde o nesneye istediği arayüzü destekleyip desteklemediğini sorar ve eğer nesne o ara yüzü destekliyorsa istemciye bir işaretçi döndürür.

Çalışma sistemi şöyledir: İstemci bir nesneye erişim elde ettiği anda bu nesnedeki IUnknown arayüzüne bir işaretçi elde eder. İşte QueryInterface fonksiyonu bu arayüzün bir fonksiyonudur. Fakat bu arayüz istemcinin kullanmak istediği fonksiyonları içermez nesnedeki diğer arayüzlere erişmesi gerekir. İstemci bu işaretçi üzerinden ,istediği arayüzün numarasını QueryInterface'e vererek çağırır ve eğer nesne istemciye izin verdiyse yeni arayüze işaretçi gelir ve artık istemci bu arayüzün sağladığı fonksiyonları kullanabilir. Şekil 4.4'te QueryInterface işleminin başarılı yada başarısız olma durumları gösterilmiştir.



Şekil 4.4 QueryInterface işlemi.

4.1.8 İstemci, Sunucu ve Nesne Gerçekleyicileri

COM'da Nesneler arasındaki iletişim istemci ve sunucu modelinde olur. İstemci bir sunucunun servislerini kullanan bir program parçacığıdır. Bir nesne servisler sunduğu içinde bu nesnenin gerçekleyicisine de sunucu denir.

COM'da istemci ve sunuculardan başka bir de nesne gerçekleyicileri vardır. Nesne üzerinde bir yada birden fazla arayüz destekleyen bir nesne gerçekleyen program parçacığıdır. Mesela bir istemci bağlandığı nesnelerin kendine bazı durumlarda geri çağrı yapabilmesi için bir mekanizmaya sahip olabilir. Bu durumda iki tarafta hem sunucu hem istemci olurlar. Bu durumda istemci kendisi bir nesne gerçekleyip işaretçisini diğer nesnelere verebilir.

Nesne : Bir yada birden fazla arayüzün fonksiyonlarının gerçeklenmesi.

Nesne Gerçekleyici : Bir nesneyi bir yada birden fazla arayüzle birlikte herhangi bir neden için gerçekleyen program parçası.

İstemci : Bir sunucu nesnesinin servislerini kullanan program parçası.

Sunucu : Bir nesne sınıfını belli bir şekilde oluşturan ve COM tanımlayıcıları veren bir program parçası.

Bu tanımları düşünerek, bir istemcinin COM servislerini kullanarak bir nesne yaratmak istediği düşünülsün. COM bu sınıfla ilgili sunucuyu çalıştırır, ve bu istenen nesneyi yaratmasını sağlar daha sonrada yaratılan nesnenin arayüz işaretçisi istemciye gönderilir. Bu işaretçi ile daha sonra istemci istediği arayüz için nesneyi sorgulayabilir. Eğer istemci bu nesnedeki bir değişimden haberdar olmak istiyorsa , kendiside bir nesne gerçekler ve bu nesnenin işaretçisini nesneye gönderir ve sunucu nesnesi de istemcinin bir istemcisi olur.

4.1.8.1 Sunucular : In-Process ve Out-Of-Process

COM sunucuları işlem dahili (In-Process) yada işlem harici (Out-of-Process) olabilirler. İşlem dahili sunucular, istemcilerin işlem alanlarında çalışırlar, işlem harici sunucular istemcilerin işlem alanında değil kendilerine ati bir işlem alanında çalışırlar. Yani aynı bilgisayarda başka bir işlem alanında yada uzak bir bilgisayarda başka bir işlem alanında çalışırlar.

İşlem dahili sunucu : Bu sunucular istemcinin işlem alanına yüklenir ve istemciye işlem dahili nesneler sunarlar. Windows sisteminde bu sunucular DLL olarak oluşturulurlar.

Yerel Sunucular : Bu sunucular başka bir işlem alanında fakat istemci ile aynı bilgisayarda çalışırlar. Bu sunucular ayrı birer çalıştırılabilir program parçacıklarıdır. EXE yapısında oluşturulurlar.

Uzak Sunucular : Uzak sunucular yine kendi işlem alanında fakat istemci ile farklı bir bilgisayarda yüklenerek çalışırlar. Uzak sistemler DLL yada EXE olarak oluşturulup kullanılabilirler.

4.1.8.2 Saydamlık

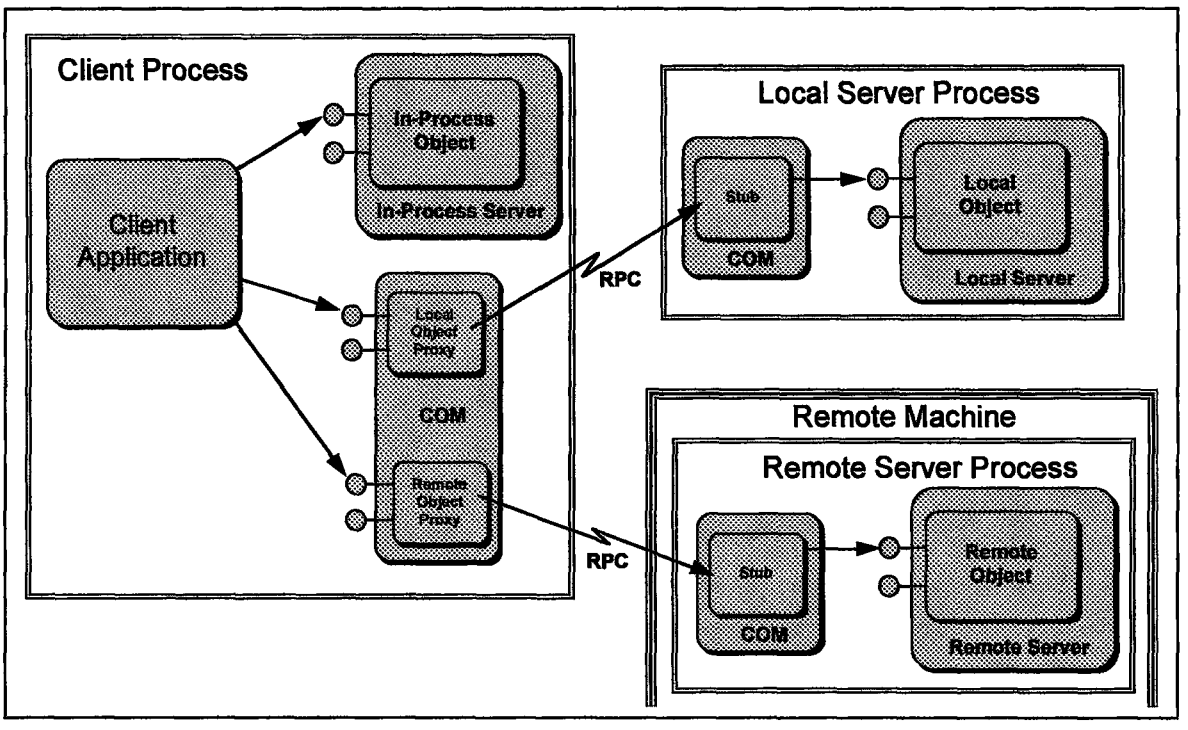
COM istemcilerin kullandıkları nesnelere çalıştıkları yerden bağımsız olarak saydam bir şekilde erişebilmesini sağlar.

İstemci gözünde tüm nesneler aynıdır ve hepsine arayüz işaretçileri sayesinde aynı şekilde erişilebilir. Tüm fonksiyon çağrıları ilk olarak bir in-process programına gider eğer erişilen nesne bir in-process nesne ise direk olarak bu nesne çağrılır. Eğer out-of-process nesne ise, çağrı bir proxy nesnesine gönderilir. Proxy nesneleri COM tarafından yaratılan diğer adres uzayında bulunan nesneye RPC çağrılarını yapan nesnelerdir.

Sunucu tarafında ise eğer sunucu in-process ise sunucunun fonksiyonlarını çağıran istemcinin kendisidir ama eğer out-of-process ise çağrıları yapan stub nesnesidir. Stub nesnesi de proxy den gelen RPC çağrılarını alarak sunucudaki gerekli fonksiyonu sanki istemci çağırıyormuş gibi çağırır. Bu yapı Şekil 4.5'te gösterilmiştir.

Bu saydamlığın bize sağladığı faydalar:

- Problemlere sunucu ve istemci arasındaki uzaklıktan bağımsız olarak ortak bir çözüm.
- Programlanmanın daha kolay yapılması. Programcılar arayüzlerle ilgilenmeyi öğrendikten sonra yeni arayüzlerle de çok rahat bir şekilde çalışabilirler.
- Sistem gerçeklemesinin merkezileştirilmesi.
- Arayüz tasarımcıları sadece tasarım üzerinde çalışırlar gerçekleme ayrıntıları ile uğraşmazlar.



Şekil 4.5 COM Çağrı Yapısı. COM çağrıları saydam bir şekilde yapar.

4.1.9 COM Kütüphanesi

Buraya kadar anlatılanlardan COM'un kendisinin de sistem seviyesinde bir gerçeklemesinin olduğunu anlaşılır. COM çekirdeğinde nesneler arası iletişiminin yapılması için gerekli spesifikasyonlar tanımlanmıştır. Bunun için bazı standartlar tanımlamaktadır.

- Arayüzlere erişim işleminin QueryInterface ile yapılması.
- Nesnelerin hayat döngüsünü takip edebilmesi için referans sayacı mekanizması.
- Hafıza kullanımı için gerekli kurallar.
- Hatasız ve zengin hata raporlama mekanizması.

COM bir spesifikasyon olmasının yanı sıra aynı zamanda COM Kütüphanesi olarak ta gerçeklenmiştir. COM kütüphanesi Windows altında DLL olarak gerçeklenmiştir.COM programları çalıştırmak için bir dizi API fonksiyonları içerir.

- Yer belirleme mekanizması.
- Saydam RPC çağrıları.
- Hafızayı kontrol etmek için standart bir mekanizma.

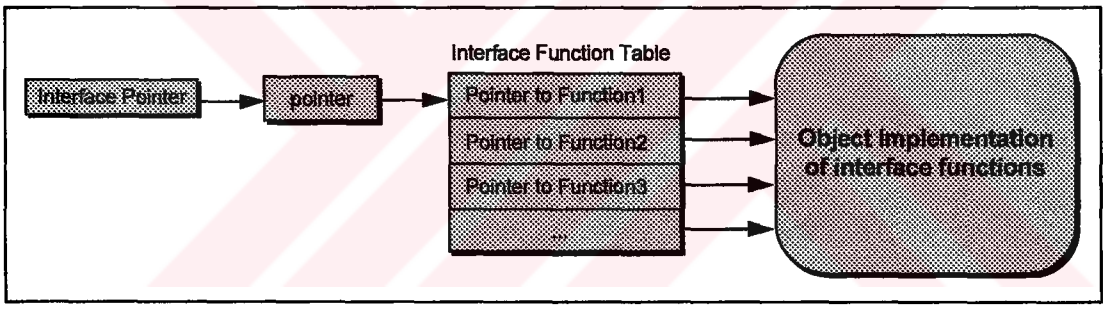
4.2 Nesneler ve Arayüzler

Daha öncede görüldüğü gibi COM kütüphanesi istemcilerin sunuculara yerden bağımsız olarak erişebilmesini sağlar. Bunun için gerekli olan proxy ve stub nesnelerini COM gerçekler ve sunar.

4.2.1 Arayüzler

Arayüz nesne ve nesnenin kullanıcısı (istemci) arasında olan bir kontrattır. Nesne belirli fonksiyonları hatasız şekilde istemciye sunma garantisi verir.

Arayüz istemcinin sunucu nesnesinde gerçeklenmiş fonksiyonlara erişimini sağlayan fonksiyonlar grubundan başka bir şey değildir. İstemci arayüze bir işaretçi saklar. Aslında bu işaretçi arayüzdeki fonksiyonların nesnedeki gerçeklemelerine işaretçiler içeren bir işaretçiler tablosuna işaretçidir. Daha iyi anlayabilmek için bu yapı aşağıdaki şekilde gösterilmiştir.



Şekil 4.6 Arayüz fonksiyon işaretçileri. İstemci, fonksiyonlara işaret eden işaretçiler tablosuna işaret eden bir işaretçiye sahiptir.

4.2.2 Arayüzlerin tanımlanması ve Tanımlayıcısı

Her arayüz 'I' harfi ile başlayan bir isme sahiptir. Bu isim sadece derleme sırasında kullanılır, yani değişik arayüzleri ayırt etmek için değil programcının daha rahat program yapabilmesi için kullanılır. Arayüzlerin tanımı ve isimleri IDL (Interface Description Language, arayüz tanımlama dili) dilinde tanımlanır. Mesela C++ dilinde arayüzler aşağıdaki gibi abstract sınıflar olarak tanımlanırlar.

```
interface IUnknown
        { virtual HRESULT  QueryInterface(IID& iid, void** ppv) =0;
          virtual ULONG            AddRef(void) =0;
          virtual ULONG            Release(void) =0; };
```

Bu tanım daha öncede değinilen IUnknown arayüzünün tanımıdır. Burada verilen derleme zamanı ismi çalışma zamanında kullanılmaz. Çalışma zamanında kullanılmak üzere başka bir tanımlayıcı belirtmek gerekir. Bu tanımlayıcılar da GUID lerdir. Bunlar IID tipinde 16 byte sayılardır. Arayüz tanımını yapan kişi bu arayüze bir GUID yaratıp onuda tanımlar. Bu şekilde tüm sistemde arayüzler tekil bir tanımlayıcıya sahip olurlar. Ve daha sonra arayüz tanımları ile birlikte bu GUID tanımları da diğerler programcılara verilir. GUID tanımları da arayüz isminin başına IID_ konarak belirlenir. Mesela IUnknown arayüzü için tanımlayıcısı IID_IUnknown olarak isimlendirilir.

4.2.3 Arayüz Tanımlama Dili – IDL

IDL , Open Software Foundation'un (OSF) , DCE spesifikasyonunda arayüzler ve özelliklerini tanımlamak için kullandığı bir yapıdır. COM IDL tanımını genişleterek dağıtık nesneleri desteklemektedir.

Bir tasarımcı bu dili kullanarak yeni bir arayüz tanımlayıp kullanabilir. Bu IDL tanımı sadece arayüzde olması gereken değişkenleri ve fonksiyonları tanımlar , gerçekleme yapılmaz. Sadece istemci ile bu nesne arasında olan kontrat belirtilir.

Bir arayüz için IDL dosyasını yazılıp IDL derleyicisinde derlendiğinde proxy ve stub nesnelerinin oluşturulması için gerekli kaynak kodu ve başlık kodları oluşur. Daha sonra bu başlık kodu (header) bu arayüzü kullanmak isteyen kullanıcılara verilir. Proxy ve Stub nesneleri DLL şeklinde oluşur ve bu nesneyi kullanmak isteyen istemcilere proxy programı verilir. Sunucuların bulunduğu yerde de stub DLL leri bulunur. IDL arayüzleri tanımlamak için kullanılan tek dil değildir, bu iş için başka diller de vardır.

4.2.4 Evrensel Tekil Tanımlayıcılar

CLSID ve IID'lerin oluşturulduğu GUID 128-bit yada 16-byte boyutunda bir sayıdır. Bu kaydın tanımlaması aşağıdaki gibi yapılır.

```
typedef struct GUID          {
        DWORD  Data1;
        WORD   Data2;
        WORD   Data3;
        BYTE   Data4[8];
        } GUID;
```

Bu şekilde tanımlayarak tanımlayıcı numaranın istenilen alanına ulaşma imkanı bulunur. Çoğunlukla programlar GUID'leri direk olarak kullanmazlar. GUID tanımlayıcılarının oluşturulması ise başlı başına bir problemdir. En çok kullanılan

yöntemlerden biri 48-bit makine tanımlayıcısın yanında UTC zamanını kullanmaktır. Bu teori sayesinde 3240 yıl boyunca saniyede 10.000.000 GUID yaratabilir.

4.2.5 IUnknown Arayüzü

IUknown arayüzü tüm nesnelerin gerçeklemek zorunda oldukları temel bir arayüzdür. Bu arayüzün 3 tane fonksiyonu vardır, bunlardan ilk ikisi AddRef, Release referans sayacını artırıp azaltmak için kullanılır. 3.'sü QueryInterface ise nesnedeki diğer arayüzlere erişmek için kullanılır. Bu arayüzün IDL tanımlaması aşağıda verilmiştir.

```
[
  object,
  uuid(00000000-0000-0000-C000-000000000046),
  pointer_default(unique)
]
interface IUnknown
{
        HRESULT         QueryInterface([in] REFIID iid, [out] void **ppv) ;
        ULONG           AddRef(void) ;
        ULONG           Release(void);
}
```

HRESULT IUnknown::QueryInterface(iid, ppv)

IID parametresinde belirtilen arayüze işaretçi döndürür. Eğer bulunduğu nesne be istenen arayüzü gerçeklememişse yani desteklemiyorsa NULL değeri döndürür. Örnek olarak

```
IOtherInterface * pother;
HRESULT  hr;
hr=psome->QueryInterface(IID_IOtherInterface, &pother);          //line 4
```

psome nesnesindeki QueryInterface fonksiyonu çağrılmıştır ve IID_IotherInterface arayüzü aranmaktadır, eğer destekleniyorsa pother işaretçisine bu arayüzün işaretçisi gelecektir ve artık bu işaretçi kullanılarak psome nesnesindeki OtherInterface fonksiyonlarına erişilebilir.

ULONG IUnknown::AddRef(void)

Bu fonksiyon nesnenin referans sayacını bir artırır. Bu sayaç 31 bit uzunluğunda bir sayıdır.

ULONG IUnknown::Release(void)

Nesnenin referans sayacı bir azaltılır. Bir istemci programında nesneye n tane AddRef çağrısı yapılmışsa nn tane de Release çağrısı yapılmalıdır.

Daha önce de belirtildiği gibi nesnelerin yaşam döngüsü referans sayacı tarafından kontrol edilir. İstemciler kullandıkları nesnelerin AddRef fonksiyonunu çağırarak bu nesnenin hafızada durmasını sağlarlar. Nesneile işleri bittiğinde ise Release fonksiyonunu çağırarak hafızadan silinmesini sağlarlar.

4.3 COM Programları

İstemci yada sunucu programlar belirli görevleri yerine getirmelidirler. Burada COM programlarının yapması gerektiği işlemleri ve kullanacakları COM fonksiyonları verilecektir. Kısaca bir COM program üç temel göreve sahiptir.

- Program başlangıcında COM kütüphanesinin sürümü kontrol edilmeli ve programın çalışabilmesi için yeterli olup olmadığına bakılmalıdır.
- Program başlangıcında COM kütüphanesi başlatılmalıdır.
- Program bitiminde COM kütüphanesi kapatılmalı böylece sistem kaynaklarını geri vermesi sağlanmalıdır.

COM Kütüphanesinin sürümünün kontrolü için CoBuildVerison fonksiyonu kullanılır. Kütüphanenin başlatılması için CoInitialize fonksiyonu kapatılması içinse CoUnInitialize fonksiyonu kullanılır.

4.3.1 Bellek Kullanımı

COM bellek kullanımının belli bir görev ayırıcı birim tarafından yapılmasını istiyor. Bunun için bir arayüz tanımlanmıştır. Bu arayüz :

IMalloc: Bu arayüz bellek ayırma birakma gibi bir dizi işlem için kullanılır.

```
interface IMalloc : IUnknown {
        void *      Alloc([in] ULONG cb);
        void *      Realloc([in] void * pv, [in] ULONG cb);
        void                Free([in] void* pv);
        ULONG   GetSize([in] void * pv);
        int                 DidAlloc([in] void * pv);
        void                HeapMinimize(void);
        };
```

4.4 COM İstemcileri

COM istemcisi bir nesneyi arayüzleri üzerinden kullanan program parçacığıdır. COM istemcisi aynı zamanda bir COM sunucusu da olabilir.

Eğer COM istemcisi çalıştırılabilir bir program şeklindeyse EXE, bir önceki bölümde anlatılan işlemleri yerine getirmesi gereklidir. Bir istemci bir nesneyi kullanmak için şu adımları gerçekleştirmesi gerekir.

- Kullanılacak nesnenin sınıfını belirle.
- Bu sınıf için sınıf yaratıcısını oluştur, ve istenen nesneyi yaratarak arayüzüne işaretçi döndürmesini iste.
- Henüz yaratılan nesnenin başlatma fonksiyonunu çağırarak nesneyi başlat.
- Daha sonra nesneyi kullan. QueryInterface fonksiyonunu çağırarak diğer arayüzlere erişim yapılır.
- Nesne ile iş bittiğinde Release fonksiyonu çağır.

Şimdi tüm bu adımlar incelenecektir. Fakat şunu hatırlamak gerekir, istemci nesneyi isteyip arayüzüne işaretçiyi aldığı andan itibaren nesnenin in-process yada out-of-process yada cross-network process olduğunu bilemez. Yani tüm nesneler saydam bir şekilde görünür istemciye. Bu nesneler arasındaki iletişimi COM daha alt düzeyde kendisi istemciden saklı olarak gerçeklemiştir.

4.4.1 Nesne Sınıflarının Belirlenmesi

COM sunucu modüllerini CLSID tanımlayıcısı ile (bir GUID tanımlayıcısı) tanımlarlar. Sunucuların yerlerinin belirlenmesi ve çalıştırılması işini COM kendisi halleder. Genellikle sistem bazında CLSID registery bilgilerini tutarlar ve böylece hangi sınıf tanımlayıcısının hangi sunucuya ait olduğunu ve yerini bulabilir. Örnek olarak Windows ta COM sunucuların (in-process DLL yada yerel EXE'leri) dizin isimlerini CLSID adı altında bir registryde saklar. Bu sayede istemcilerin bu bilginin nerede nasıl saklandığını ve nasıl kullanıldığını bilmesi gerekmez.

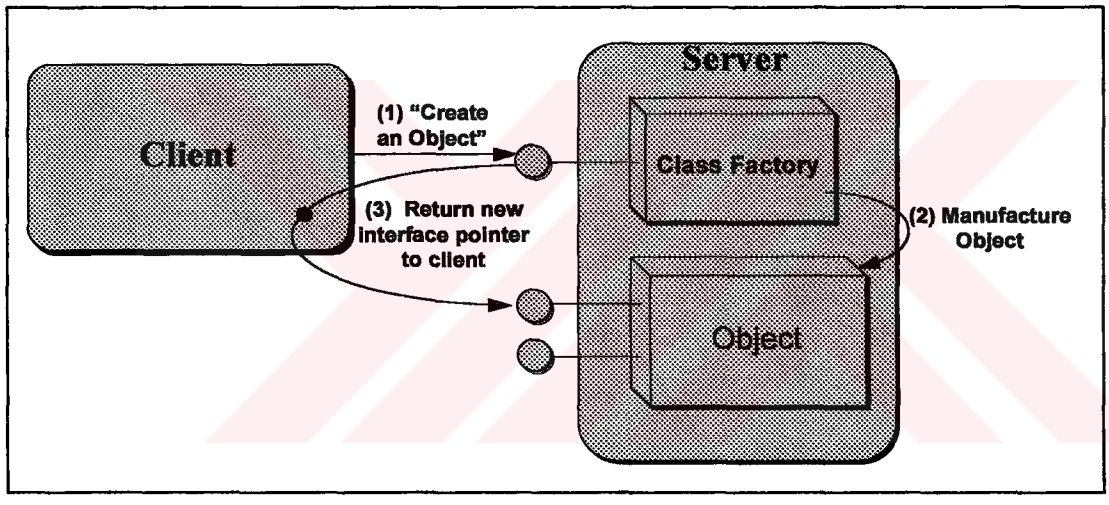
İstemci bir çok değişik yoldan COM'a istediği nesnenin sınıfını bildirebilir. Birinci yol, nesne sabit bir CLSID değerine sahiptir ve istemci derleme sırasında bunu bildiği için direk olarak COM'a bildirebilir. İkinci yol, istemci bu CLSID'yi gösteren okunabilir bir isme sahip olabilir COM bu isimden yola çıkarak nesnenin sınıfını bulur. Bir başka yol ise istemci daha önceden saklanmış CLSID 'yi belirten bir bilgiye sahip olabilir. Bir başka yol ise sistemdeki tüm sınıfların listesinin kullanıcıya gösterilmesi.

4.4.2 Nesnelerin Yaratılması

Nesnenin CLSID sini elde ettikten sonra istemci nesneyi kullanabilmesi için yaratması gerekir. Bu iki adımda yapılır:

- Bu CLSID için sınıf yaratıcısının al.
- Sınıf yaratıcından istenilen nesneyi yaratmasını ve bir ara yüzüne işaretçi döndürmesini iste.

Bu aşamalar tamamlandıktan (şekilde gösterilmiştir) sonra , istemci bu nesne ile istediği şeyi yapmakta serbesttir. Bu nesnenin üzerindeki tüm arayüzleri kullanabilir.



Şekil 4.7 Nesne yaratılması. İstemci sınıf fabrikasından bir nesne yaratmasını isteyip o nesneye işaretçi almıştır.

4.4.2.1 Sınıf Fabrikası : IClassFactory Arayüzü

Sınıf fabrikası istenen istemcilerin istediği nesneleri yaratmakla görevli olan nesnedir. Bir sınıf fabrikası nesnesi sunucu modülü olarak DLL yada EXE olarak tanımlanmıştır. Bu nesnenin arayüzünün tanımı aşağıdaki gibidir.

```
[
  object,
  uuid(00000001-0000-0000-C000-000000000046), // IID_IClassFactory
  pointer_default(unique)
]
interface IClassFactory : IUnknown
```

```
{
        HRESULT CreateInstance([in] IUnknown * pUnkOuter, [in] REFIID iid, [out] void * ppv);
        HRESULT LockServer([in]BOOL fLock);
}
```

İki tane fonksiyon içerir. Birincisi CreateInstance fonksiyonu. Bu fonksiyon başlatılmamış bir nesne yaratmak için kullanılır. IID parametresi ile verilen nesneden yaratır. Ve bu yeni yaratılan nesneye bir işaretçi döndürür.

LockServer fonksiyonu sunucunun hiçbir nesne bulundurmadığı durumda da hafızada bulunması için kullanılır. Normal olarak sunucu hiçbir nesne içermiyorsa kendini kapatır, bu fonksiyonla istersek sunucunun devamlı çalışır vaziyette kalmasını sağlanabilir.

4.4.3 Bir CLSID için Sınıf Fabrikası Nesnesinin Oluşturulması

Sınıf fabrikasının ne işe yaradığı gösterilmişti, şimdi istemcilerin bundan nasıl yararlanabileceği gösterilecektir. Aynı makine üzerindeki tüm nesneler için istemci CoGetClassObject fonksiyonu çağırır. Bu fonksiyon sınıf fabrikasını oluşturmak için gerekli tüm işlemleri yerine getirir. İstemci bu sınıf fabrikasının işaretçisini aldıktan sonra IClassFactory::CreateInstance fonksiyonunu kullanarak nesnelerini oluşturur. Daha sonrada IClassFactory::Release ile sınıf fabrikasını kapatır. Bu işlem birden fazla nesne yaratılmak isteniyorsa kullanılır. Eğer istemci sadece bir nesne yaratacaksa bunun için yukarıdaki anlatılanları tek başına yapan bir fonksiyon olan CoCreateInstance kullanılır.

```
HRESULT CoGetClassObject(clsid, grfContext, pServerInfo, iid, ppv)
```

Clsid parametresi ile verilen nesneye ait sınıf fabrikasının oluşturur ve döndürür. Eğer gerekli ise COM gerekli kütüphaneleri yükler. Buradaki iid arayüzü IID_IClassFactory arayüzüdür. Bu sınıf fabrikasının işi bittiğinde Release çağrılarak silinmesi sağlanır. Buradaki grfContext ise nesnenin nerede yaratılacağı ile ilgilidir.

```
typedef enum tagCLSCTX {
        CLSCTX_INPROC_SERVER        = 1,
        CLSCTX_INPROC_HANDLER       = 2,
        CLSCTX_LOCAL_SERVER         = 4,
        CLSCTX_REMOTE_SERVER        = 16.
        } CLSCTX;
```

Örnek olarak aşağıdaki kod verilebilir. İstemci COM'u sadece inproc sunucular kullanmaya zorluyor.

```
IClassFactory *     pCF;
IUnknown *          pUnkObj;
HRESULT             hr;

hr=CoGetClassObject(CLSID_TextRender, CLSCTX_INPROC_SERVER, NULL, IID_IClassFactory, (void *)pCF);
if (FAILED(hr))
        //Could not obtain class factory, creation fails completely.

/*
 * Create the object. If this call succeeds the pUnkObj will
 * be valid and have a reference count on it on behalf of the caller
 * which the caller must Release.
 */
hr=pCF->CreateInstance(NULL, IID_IUnknown, (void *)pUnkObj);

//Caller must call Release regardless of CreateInstance result
pCF->Release();

if (FAILED(hr))
        //Object creation failed: interface may not be supported

/*
 * Now use the object in whatever capacity the caller desires.
 * The first step will be initialization.
 */

//Release the object when finished with it.
pUnkObj->Release();
```

Görüldüğü gibi CoGetClassObject fonksiyonundan sonra IClassFactory::CreateInstance ve Release fonksiyonları çağrılıyor. COM tüm bu işlemler bir fonksiyonlar yapabilmek için yeni bir fonksiyon tanımlamıştır.

```
HRESULT CoCreateInstance(clsid, pUnkOuter, grfContext, iid, ppvObj)
```

Bu fonksiyon clsid ile istenen bir nesneden yaratarak işaretçisini döndürür. Daha öncede belirtildiği gibi bu fonksiyon yukarıdaki anlatılan fonksiyonları çağırır. Bu yüzden aşağıdaki gibi tanımlanabilir.

```
HRESULT CoCreateInstance(REFCLSID clsid, IUnknown * pUnkOuter,
        DWORD grfContext, REFIID iid, void * ppvObj)
        {
        IClassFactory * pCF;
        HRESULT     hr;

        hr=CoGetClassObject(clsid, grfContext, NULL, IID_IClassFactory, (void *)pCF);

        if (FAILED(hr))
                return hr;

        hr=pCF->CreateInstance(pUnkOuter, iid, (void *)ppv);
        pCF->Release();
```

```
/*
 * If CreateInstance fails, ppv will be set to NULL. Otherwise
 * ppv has the interface pointer and hr contains NOERROR.
 */
return hr;
}
```

HRESULT CoCreateInstanceEx(clsId, pUnkOuter, grfContext, pServerInfo, dwCount, rgMultiQI)

Bu fonksiyonda CoCreateInstance fonksiyonu ile aynı işi yapar. Fakat burada nesne başka bir makine üzerinde yaratılabilir.

4.4.4 Nesnenin Başlatılması

İstemci başarılı bir şekilde yarattıktan sonra bu nesneyi başlatması gereklidir. Bu işlem genellikle nesnenin başlatma arayüzüne bir çağrı ile yapılır. Nesne başlatılmadan önce erişilebilen fonksiyonlar sadece IUnknown fonksiyonlarıdır.

Tüm nesnelerin başlatılması gerekmeye bilir bu nesnenin tanımında belirtilir. Mesela bilgilerini bir dosyaya yazan bir nesne IPersistFile arayüzünü gerçeklemelidir. Bu arayüzün Load fonksiyonu nesnenin dosyadan bilgilerini yüklemesini sağlar, dolayısı ile bu fonksiyon nesnenin başlatma fonksiyonudur.

4.4.5 Nesnenin Kullanılması

Tüm bu işlemler bittikten sonra istemci nesneyi kullanıma hazır demektir. Fakat istemci nesneden istediği işlemleri yapmasını istemek için kullanacağı fonksiyonları içeren arayüzlerini destekleyip desteklemediğini öğrenmesi ve destekliyorsa işaretçisini alması gerekir. Buda daha öncede anlatılan QueryInterface fonksiyonu ile yapılır. Yani istemci yarattığı nesnenin istediği fonksiyonları destekleyemeyeceği gerçeğine hazır olması gerekir. Yani her QueryInterface çağrısı başarılı olacak demek değildir. Bu işlemin nasıl yapıldığı ileride Delphi üzerinde gösterilecektir.

4.4.6 Nesnenin Serbest bırakılması

İstemcinin en son işlemi nesneyi silmek olacaktır. Bu işlem nesnenin Release fonksiyonu ile olur. İstemci nesneleri yarattığında bir referans sayaca sahip olurlar, ve bu sayaç her referansta 1 artar. Bir nesneye birden fazla istemci erişimde bulunabilir bu taktirde referans sayacı 1 den yüksek olacaktır. Bu durumda Referans sayacını 0 a indiren en son Release'den sonra nesne kendini yok edecektir. Kısacası her istemci ne kadar AddRef yaptıysa o kadar Release çağırmalıdır. Nesnenin yok edilip edilmeyeceği bu referans sayacına bağlıdır. Aynı şekilde sunucular içinde bir kilit mekanizması vardır. İstemci nesne içermeyen sunucuların

kapanmaması için LockServer(True) fonksiyonu çağırır. Bu fonksiyon sunucu içindeki bir sayacı artırır daha sonra LockServer(False) ile bu sayaç azaltılır ve 0 olduğunda da sunucu kendini yok eder.

4.5 COM Sunucuları

COM sunucusu bir yada daha fazla nesne sınıfını gerçekleyen ve istemcilere sunan DLL yada EXE şeklinde yazılmış programdır. Sunucu nesne sınıflarını gerçekler ve bunlara verilen CLSID sayesinde istemcilerin bu nesneleri yaratıp kullanmalarını sağlar. Aynı zamanda COM sunucuları diğer nesnelerin de istemcisi olabilirler. Mesela bir sunucu fonksiyonlarını gerçeklemek için başka nesneleri kullanabilir, o zaman o nesnenin istemcisi olmuş olur.

Eğer sunucu programı EXE şeklindeyse daha önce anlatılan COM programlarının yapması gereken şeyleri yapmalıdır. Eğer DLL şeklindeyse (in-process sunucu) en azından COM kütüphanesinin versioynunu kontrol etmesi gerekir. Bir COM sunucusunun yapması gereken adımlar:

- Gerçeklediği her sınıf için bir CLSID numarası al ve sistemin bu numarayla sunucuyu bulmasını sağla.
- Desteklenen her sınıf için IClassFactory arayüzünü gerçekleyerek bir nesne fabrikası nesnesi oluştur.
- COM kütüphanesine bu nesne fabrikasını bildir. Gerektiğinde bulup yükleyebilmesi için.
- Sunucunun bir nesnesi olmadığı zaman kapatılması için gerekli ayarlamaları yap.

Sunucuların asıl görevi istemcilere nesneleri sunmasıdır. Şimdi bir nesnenin basit olarak yapısı gösterilecektir.

4.5.1 Nesne Sınıflarının Tanımlanması ve Sisteme Kayıt Edilmesi

COM sisteminin güçlü yanı nesne sınıflarını tanımlamak için evrensel tekil tanımlayıcıları kullanmasıdır. Bu tanımlayıcılar (GUID) yerel makinede, ve tüm platformdaki makinelerde de tekil olarak geçerlidir. Bunu sağlayan algoritma COM kütüphanesinde bulunan CoCreateGuid fonksiyonunda bulunur. Bu fonksiyonu kullanarak bir nesne gerçekleyicisi bu nesne sınıfı için COM dan bir GUID alması gerekir.

Eğer istemci CLSID ile nesneye ulaşamazsa , bir nesneyi CLSID ile tanımlamak anlamsız olacaktı. Daha öncede gösterildiği gibi istemciler çeşitli yollardan bu CLSID'leri kullanarak nesneleri oluşturuyorlardı. En basit yolu derleme sırasında CLSID'leri tanımlamak ve kullanmak. Diğer bir yolda çalışma zamanında CLSID'leri sistemden öğrenip kullanmak. Bu yüzden dinamik olarak CLSID'leri saklayan ve işleyen bir yapıya ihtiyaç vardır. Ayrıca sistem çapında CLSID numaralarından sunucu kodunu bulabilmek için bir yol gereklidir. Bu konu COM kütüphanesinin görevidir. Sunucu nesne sınıflarını oluşturup COM'a kaydettiği andan itibaren , bu kodun gerektiğinde bulunup çalıştırılması COM'a aittir.

Windows ortamında gerçeklenen COM modelinde bu bilgileri saklamak için windows system registry (sistem kayıt sicili) kullanılır. Kayıtta CLSID adında bir kök anahtarı kullanılır. Bir sunucu kendini kayıt ettirdiğinde bu kök anahtarın altında bir alt anahtar oluşturur ve kendine ait bilgileri buraya yazar.

Örnek olarak TextRender nesnesini verilebilir Bu nesnenin ilk oluşturduğu kayıt anahtarı şöyle olur.

```
CLSID
        {12345678-ABCD-1234-5678-9ABCDEF00000} = TextRender Example
```

ve sunucunun çalıştığı yere göre bir başka alt anahtar kayıt daha oluşturulur.

Tablo 4.1 Sunucu kayıt anahtarları ve değerleri

Sunucu Yeri	Alt Anahtar Kayıt	Değer
In-Process	InprocServer32	DLL'in yeri
Local	LocalServer32	EXE'nin yeri
Object Handler	**InprocHandler32**	DLL'in yeri

Böylece TextRender için kayıt aşağıdaki gibi olur.

```
CLSID
        {12345678-ABCD-1234-5678-9ABCDEF00000} = TextRender Example
                InprocServer32 = c:\objects\textrend.dll
```

COM sunucularını kullanan programlar programın kurulumu sırasında bu sunucuları sisteme kayıt ederler.

EXE programı şeklindeki sunucuları sisteme kayıt etmek için program parametresi olarak /REGSERVER girilir. Ve program çalıştıktan sonra kendi kendini kayıt eder. Eğer kayıt edilen bir sunucuyu silmek istiyorsak bu seferde /UNREGSERVER girilir.

4.5.2 Sınıf Fabrikasının Gerçeklenmesi

İstemcide CLSID numarasının olması , aslında bu sınıfa ait nesneleri yaratıp sunan bir sınıf fabrikası olması anlamına gelir. Sunucular bu fabrikayı oluşturup COM'a kayıt etmekle sorumludurlar. İlk olarak bu nesnenin tanımlanması gereklidir. Mesela TextRender nesnesinin sınıf fabrikasının tanımı aşağıdaki gibi olacaktır.

```
class CTextRenderFactory : public IClassFactory
        {
        protected:
                ULONG        m_cRef;

        public:
                CTextRenderFactory(void);
                ~CTextRenderFactory(void);

                //IUnknown members
                HRESULT QueryInterface(REFIID, pLPVOID);
                ULONG AddRef(void);
                ULONG Release(void);

                //IClassFactory members
                HRESULT CreateInstance(IUnknown *, REFIID iid, void **ppv
                HRESULT LockServer(BOOL);
        };
```

Buradaki üye fonksiyonların gerçeklemeleri aşağıdaki gibi yapılır.

```
//A global variable that counts objects being served
ULONG   g_cObj=0;
HRESULT CTextRenderFactory::CreateInstance(IUnknown * pUnkOuter, REFIID iid, void ** ppv) {
        CTextRender *      pObj;
        HRESULT          hr;
        *ppv=NULL;
        hr=E_OUTOFMEMORY;
        if (NULL!=pUnkOuter)
                return CLASS_E_NOAGGREGATION;
        //Create the object passing function to notify on destruction.
        pObj=new CTextRender(pUnkOuter, ObjectDestroyed);
        if (NULL==pObj)
                return hr;
        [Usually some other object initialization done here]
        //Obtain the first interface pointer (which does an AddRef)
        hr=pObj->QueryInterface(iid, ppv);
        //Kill the object if initial creation or FInit failed.
        if (FAILED(hr)) delete pObj;
        else _cObj++; return hr;
        }
```

CreateInstance fonksiyonu CTextRender nesnesinin bir örneğini yaratarak arayüz işaretçisini döndürür. Bu tüm sunucular için standart bir fonksiyondur.

Burada iki önemli nokta vardır : Kodda görüldüğü üzere nesne oluşturulduktan sonra bir kere QueryInterface çağrılmıştır. Böylece nesne referans sayacı bir artırılır çünkü bu nesneyi yaratan istemci bu nesneye referans etmektedir.

Diğer nokta ise sunucuda bulunan nesnelerin sayılması ise g_cObj sayacı ile yapılır. COM bu sayıyı tutmak için özel bir sistem içermez. Nesnenin silinmesinde ise bu sayaç bir azaltılır.

```
void ObjectDestroyed(void) {
        g_cObj--;
        [Initiate unloading if g_cObj is zero and there are no locks]
        return;
        }
CTextRender::CTextRender(void (* pfnDestroy)(void)) {
        m_cRef=0;
        m_pfnDestroy=pfnDestroy;
        [Other initialization]
        return;
        }
ULONG CTextRender::Release(void) {
        ULONG     cRefT;
        cRefT=--m_cRef;
        If (0L==m_cRef) {
                if (NULL!=m_pfnDestroy)
                        (*m_pfnDestroy)();
                delete this;
                }
        return cRefT;
}
```

Bir başka önemli fonksiyon ise LockServer fonksiyonudur. Bu fonksiyon sunucunun nesne olmaması durumunda kapanıp kapanmayacağını ayarlar. Tanımı aşağıdaki gibidir. Bunun içinde bir sayaç tutulur g_cLock , eğer bu sayaç 0 dan büyükse sunucu hiç kapatılmaz ama eğer 0'sa ve de sunucuda bir nesne yoksa sunucu kapatılır.

```
//Global server lock count.
ULONG     g_cLock=0;
HRESULT CTextRenderFactory::LockServer(BOOL fLock)
        {
        if (fLock)
                g_cLock++;
        else
                {
                g_cLock--;
                [Initiate unloading if there are no objects and no locks]
                }

        return NOERROR;
        }
```

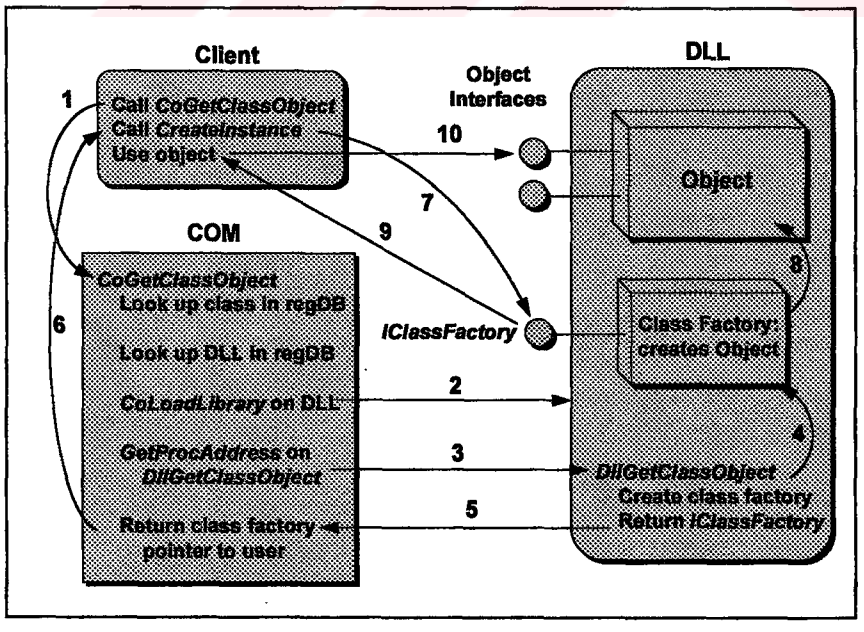
4.5.3 Nesne Fabrikasının Kayıt Edilmesi

Sunucular nesne fabrikalarını oluşturduktan sonra bunları COM'a kayıt ettirmeleri gerekir. Aksi taktirde istemciler bu nesneleri kullanamazlar. Sunucu tipine göre (DLL yada EXE) bu kayıt işlemi değişmektedir.

DLL sunucularının kayıt edilmesi: DLL şeklinde yazılan sunucuların DllGetClassObject adında bir fonksiyon tanımlayıp export etmeleri gerekir. COM bu fonksiyonu CoGetClassObject içinde kullanarak fabrikayı oluşturur. Fonksiyonun tanımı yukarıda verilmiştir.

```
HRESULT DllGetClassObject(REFCLSID clsid, REFIID iid, void **ppv) {
        CTextRenderFactory * pCF;
        HRESULT         hr=E_OUTOFMEMORY;
        if (!CLSID_TextRender!=clsid)
                return E_FAIL;
        pCF=new CTextRenderFactory();
        if (NULL==pCF)
                return E_OUTOFMEMORY;
        //This validates the requested interface and calls AddRef
        hr=pCF->QueryInterface(iid, ppv);
        if (FAILED(hr))
                delete pCF;
        else
                ppv=pCF;
        return hr;
}
```

DLL sunucularının nesnelerinin ve sınıf fabrikasının yapısı şekilde gösterilmiştir.



Şekil 4.8 DLL sunucusundan nesne oluşturulması ve COM çağrıları.

EXE sunucuların kayıt edilmesi: EXE sunucularının kayıt edilmesi DLL sunucularınkinden farklıdır. COM CoGetCLassObject içinden bir programı yüklediği zaman istenen CLSID 'ye ait sınıf fabrikasının CoRegisterClassObject ile kayıt edilmesini bekler. Bu fabrika hazır olduktan sonra buna ait işaretçiyi döndürür. Eğer sunucu bu kayıt işleminde çok geç kalırsa CoGetClassObject fonksiyonu zaman aşımına uğrayıp hata verebilir.

Bir sunucu birden fazla sınıf destekleyebilir desteklediği her sınıf için CoRegisterClassObject fonksiyonunu kullanarak bu sınıf fabrikalarını COM'a kayıt ettirir. CoRevokeClassObject fonksiyonu da bu işlemin tersini yapar, yani daha önceden kayıt edilmiş bir sınıfı siler.

HRESULT CoRegisterClassObject(clsid, pUnk, grfContext, grfFlags, pdwRegister)

PIUnk değişkeni ile belirlenen sınıf fabrikası COM'a kayıt edilir. Bir COM sunucusu başladığında ilgili sınıf fabrikalarını oluşturup hepsi için bu fonksiyonu çağırır ve COM'a bildirir. Sunucu kapanırken de CoRevokeClassObject fonksiyonu daha önce kayıt ettiği sınıfları siler. grfFlags değişkeni nesneye bağlantının nasıl yapılacağını gösterir. Alabildiği değerler:

```
typedef enum tagREGCLS
        {
        REGCLS_SINGLEUSE = 0,
        REGCLS_MULTIPLEUSE = 1,
        REGCLS_MULTI_SEPARATE = 2
        } REGCLS;
```

REGCLS_SINGLEUSE : Sunucuya sadece bir istemci bağlanabilir.

REGCLS_MULTIPLEUSE : Birden fazla istemci bağlanabilir.

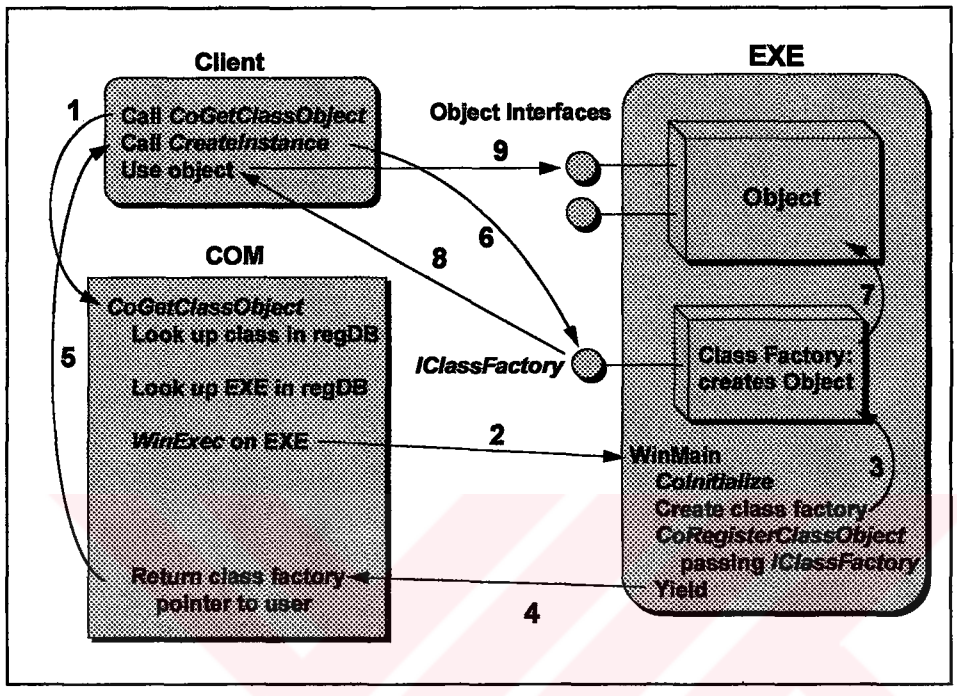
REGCLS_MULTI_SEPARATE : Her bir istemci için ayrı bir sunucu oluşturulur.

HRESULT CoRevokeClassObject(dwRegister)

Daha önce kayıt edilen bir sınıf fabrikasını siler. Sunucular gerekli kapanış işlemlerini yaptıktan sonra bu fonksiyonu çağırır. Sunucunun kapanması için aşağıdaki şartlar yerine gelmesi gerekir.

- Sunucuda kullanılan bir nesne olmaması gerekir. Nesne sayacı sıfır olması gerekir.
- Sınıf fabrikası lock sayacı sıfır olması gerekir.
- Sunucu programının kullanıcı arayüzü olmaması gerekir.

Sunucu programının , nesnelerin ve sınıf fabrikasının yapısı ve işleyişi şekilde verilmiştir. İstemcinin CoGetClassObject fonksiyonunu çağırdığı zaman yapılan işlemler gösterilmiştir.



Şekil 4.9 EXE sunucusundan nesne oluşturulması ve COM çağrıları.

DLL de olduğu gibi, sunucu yapısı aynıdır, ikisi de nesne ve sınıf fabrikasına sahiptir. Dikkat edilirse istemci tarafından nesne oluşturma işlemi aynıdır. İstemci bir kere istediği nesnenin CLDIS'sini belirledikten sonra geri kalan işlemleri CoGetClassObject fonksiyonu yapmaktadır. Aradaki tek fark COM kütüphanesinde olmaktadır sınıf fabrikasının tanımlanması ve yüklenmesinde fark vardır.

4.5.4 Sunucunun Kapatılması

Sunucunun kullanılan bir nesnesi olmadığı yoksa , lock sayacı sıfır olmuşsa ve de bir kullanıcı arabirimi yoksa sunucu kendini kapatması gerekir. Aksi taktirde hafızada gereksiz yer işgal edecektir. DLL ve EXE sunucuların kapanış işlemleri farklılıklar gösterir. Bir sunucu programı kendi kendini kapatabilir fakat DLL sunucular başkasının kendilerini kapatmasını beklerler. Bu yüzden kapanışları farklıdır.

DLL Sunucuların Kapatılması: Yukarıda da belirtildiği üzere bir DLL sunucu başkasının kendisini kapatmasını bekler. Fakat sunucunun kendisinin kapatılıp kapatılmayacağını belirten bir mekanizması vardır. Bu DllCanUnloadNow fonksiyonu ile belirlenir.

HRESULT DllCanUnloadNow(void)

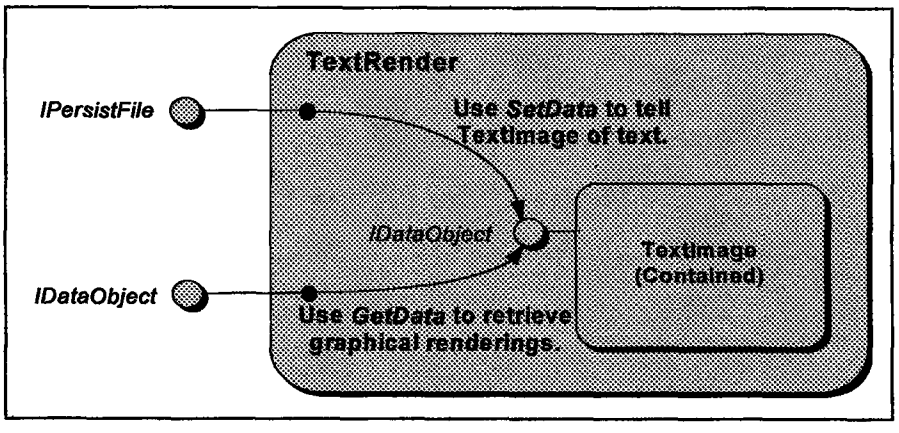
Bu fonksiyon COM tarafından sağlanmaz, bu fonksiyon DLL tipi sunucular yazılırken gerçeklenmelidir. Eğer bu fonksiyon S_OK döndürürse sunucu kapatılabilir, aksi taktirde meşgul demektir.

EXE Sunucularının Kapatılması : Kapanış şartları oluştuğunda EXE program ana modülünden çıkarak kendi kendini kapatabilir. Sunucu kapanırken CoRevokeClassObject fonksiyonu çağırarak daha önce kayıt ettiği sınıf fabrikalarını silmesi gerekir.

4.5.5 Nesne Tekrar Kullanımı

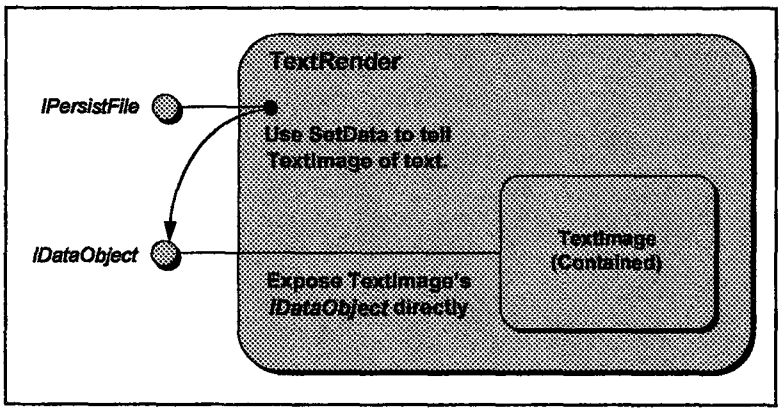
Nesne tabanlı programlamada , programda yapılmak istenilen bazı şeylerin daha önceden nesne şeklinde yazılmış olarak hazır olduğu görülür. Aynı işi yapan nesneleri tekrar yazmaktansa bu hazır yazılanları kullanmak daha iyidir.

COM daha önceden yazılan nesnelerin tekrar kullanımı için iki yöntem içerir, containment ve aggregation. Containmentta en dıştaki nesne en içteki nesnenin fonksiyonlarını kullanır. Şekil 4.10'da gösterildiği gibi dıştaki nesne içteki nesnenin bir istemcisi durumundadır. Onun fonksiyonlarını kendi fonksiyonlarını gerçeklemek için kullanır



Şekil 4.10 Containment ile nesnelerin tekrar kullanımı.

Aggregation da ise en dıştaki nesne sadece içteki nesnenin fonksiyonlarını aynen dışarı yansıtır. Şekilde de görüldüğü gibi tekrar kullanılan nesnenin arayüzleri dışa verilmiştir.



Şekil 4.11 Aggregation ile nesnelerin tekrar kullanımı.

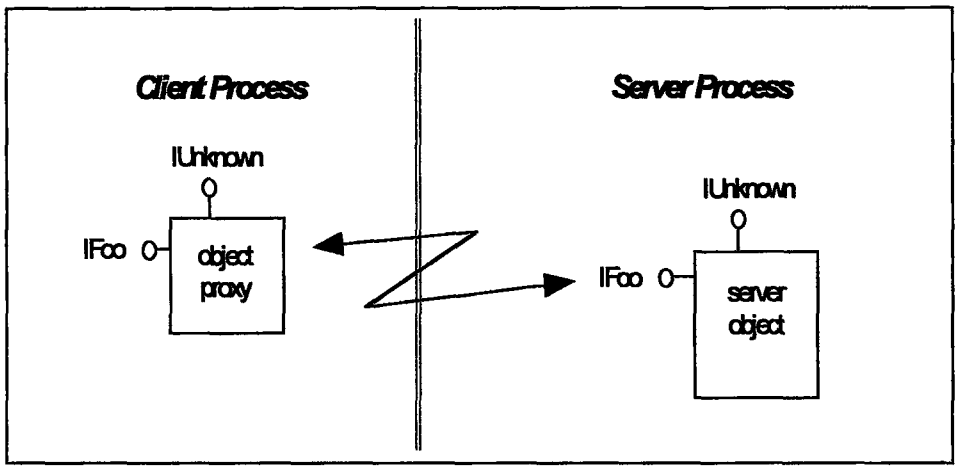
4.6 Arayüz Erişimi

COM istemcileri bağlı oldukları nesnelerle arayüz işaretçileri ile vasıtasıyla haberleşirler. Nesnelerdeki işlemlerden yararlanmak için bu arayüzlerdeki fonksiyonlar kullanılır.

Arayüz erişim mekanizması, fonksiyon çağrılarının başka bir programdaki yada başka bir bilgisayardaki nesnelere arayüz işaretçisini döndürmesini sağlar. Bu işlem istemci ve sunucu için saydam bir şekilde gerçekleşir. Yani her ikisi de bu işlemin nasıl yapıldığını bilmez. Şimdi bu arayüz erişimini incelenecektir.

İlk olarak istemci ve sunucu nesnesi arasındaki ilk erişimin kurulduğunu kabul edilsin. Örnek olarak Şekil 4.12 aşağıdaki durum kabul edilsin:

Bir sunucu programında IFoo adında arayüzü destekleyen bir nesnemiz olduğunu düşünelim. İstemci ve sunucu arasındaki ilk bağlantının bir şekilde yapıldığını kabul ediyoruz. İstemci tarafında bir proxy nesnesi vardır, bu nesne sunucudaki gerçek nesnenin desteklediği tüm arayüzleri destekler. Fakat buradaki fonksiyonlarin gerçeklenmesi farklıdır. Kendilerine gelen fonksiyon çağrılarını diğer taraftaki sunucudaki gerçek nesneye yöneltir. Yani sadece yöneltme yapar. Buna marshal (paketleme) işlemi denir. Karşı tarafa (sunucuya) gelen bilgiler unmarshal (paketleri acma) işlemi yapılır.

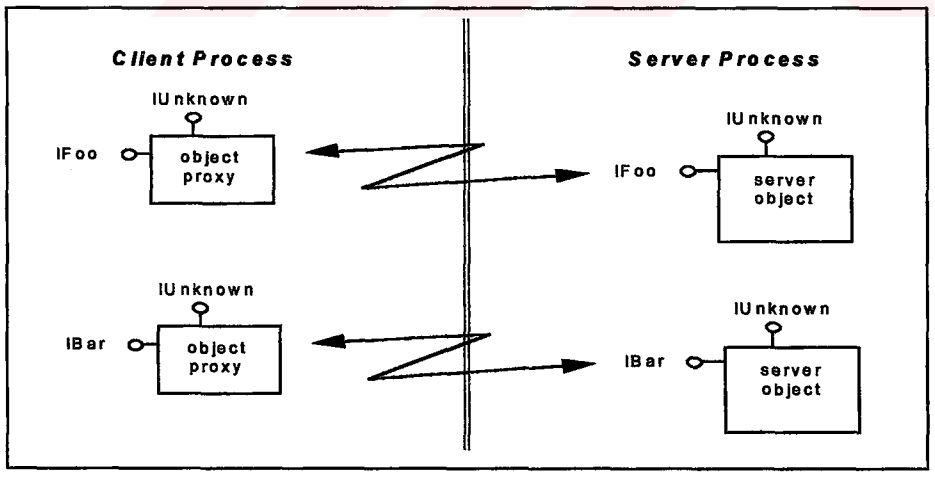


Şekil 4.12 İstemci ve Sunucu bağlantısı.

Basit olması için IFoo arayüzünün aşağıdaki gibi tanımlandığını varsayılsın.

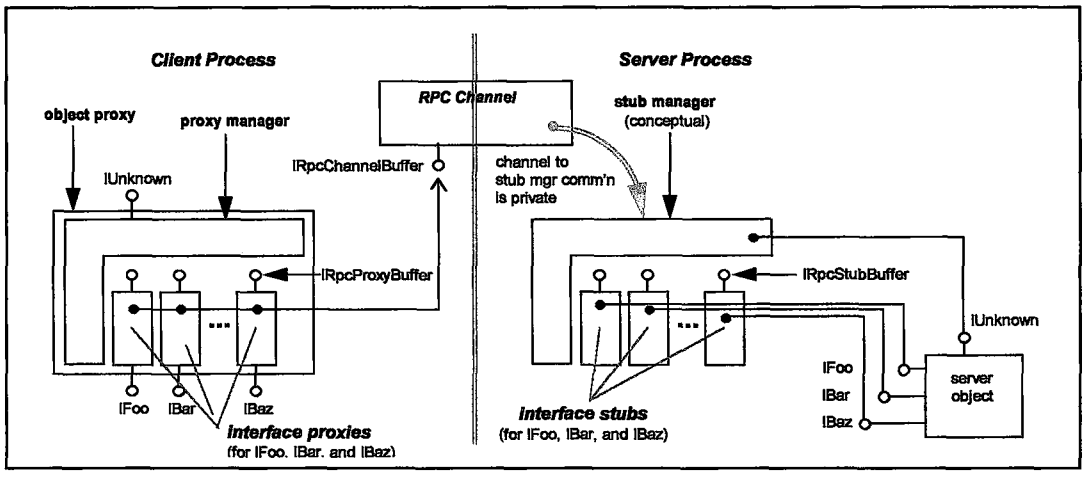
```
interface IFoo : IUnknown {
        IBar *    ReturnABar();
        };
```

Eğer istemci pFoo->ReturnABar() fonksiyonu çağırırsa, proxy nesnesi bu çağrıyı alıp, sunucuda bulunan nesnenin IFoo::ReturnABar() fonksiyonuna yönlendirir. Daha sonra sunucu nesnesi de işlemini bitirdikten sonra IBar* tipinde cevabı geri döndürmek zorundadır. Bunun içinde ikinci bir bağlantı kurulmalıdır.



Şekil 4.13 İstemci sunucu bağlantısının kurulması.

Sunucu tarafında oluşturulan IBar* cevabı paketlenir, Daha sonra bir şekilde bu paket istemciye gönderilir ve paket açılarak yeni proxy nesnesi yaratılır. İstemci ve Sunucu arasındaki arayüz erişimi basit olarak aşağıdaki Şekil 4.14'te gösterilmiştir.



Şekil 4.14 İstemci ve sunucu arasındaki arayüz erişimi.

İstemci tarafında bir proxy nesnesi vardır. Bu nesne sunucu tarafında olan (çalıştığı yer belli olmayan) gerçek nesnenin bir kopyasıdır. Bu proxy içinde birde proxy düzenleyici vardır. Birde proxy arayüzleri vardır. Sunucu tarafında ise stub nesnesi bulunmaktadır. Bunun içindede stub arayüzleri vardır.

İstemci , nesnenin bir arayüzündeki bir fonksiyonu çağırmak için proxy nesnesinin içindeki fonksiyonları çağırır, gerekli proxy fonksiyonu bu çağrıyı alır ve parametreleri paketleme yaptıktan sonra RPC üzerinden karşıdaki stub nesnesine gönderir. Stub nesnesindeki ilgili fonksiyon bu çağrıyı alır paketleri açar ve sunucuda bulunan gerçek nesnenin fonksiyonunu çağırır, sonra oradan cevabı aldıktan sonra yine aynı yol üzerinden cevabı paketleyerek gönderir ve ilgili proxy fonksiyonu cevabı alıp istemciye döndürür.

4.7 Güvenlik

COM'un desteklediği iki tane güvenlik mekanizması vardır. Bunlar activation güvenliği (başlangıç güvenliği) ve çağrı (call) güvenliği. Bunlardan birincisi yeni nesnelerin nasıl yaratılacağı, yada var olan nesnelere erişimi kontrol eder. Yada Sınıf Tablosuna (Class Table) ve Çalışan Nesne Tablosu (Running Object Table) na erişimleri kontrol eder. Çağrı güvenliği bağlantı kurulan nesnelere yapılan tüm çağrıları kontrol eder.

COM'daki bu güvenlik mekanizması platform bağımlıdır. Burada Windows sistemindeki güvenlik incelenecektir.

Başlatma Güvenliği (Activation Security): Nesneler istemcilere iki değişik şekilde sunulurlar. Birincisi statik olarak, bir nesne kullanımdan önce sisteme tanıtılır ve istendiği zaman SCM (Service Control Manager) tarafından yüklenirler. İkinci yol ise dinamik olarak nesnelerin tanıtılması dır. Bu iş için daha önce anlatılan CoRegisterClassObject kullanılır. Başlatma güvenliği SCM tarafından yapılır. Bir çağrı geldiği zaman SCM bu çağrıyı yapan istemcinin bu sunucuyu çağırıp çağıramayacağını kontrol eder. Bunun için de sunucu kayıtlarında saklanan güvenlik bilgilerini kullanılır. İleriki bölümlerde windows ortamında bunun nasıl yapıldığı incelenecektir.

Çağrı Güvenliği : COM iki değişik mekanizma kullanır, bunlardan birincisi DCE-RPC ye benzemektedir. Programların kendi güvenlik kontrolünü yapmaları için COM API çağrıları içerir. İkincisi ise COM tarafından otomatik olarak yapılır. Eğer program gerekli bilgiyi belirtirse , COM bu güvenliği sağlamak için gerekli kontrol işlemlerini yapacaktır. Bu otomatik mekanizma işlem bazında yapılır, nesne ve fonksiyon bazında yapılmamaktadır. Bir program daha fazla güvenlik istiyorsa bunları kendisi yapmalıdır fakat hem COM tarafından otomatik güvenlik kontrolü hem de kendi kontrolünü yapılabilir.

5. SUNUCU VE İSTEMCİLERİN PROGRAMLANMASI

Programlar Windows altında Delphi (Pascal) programlama dilinde yazılmıştır. Sunucu bilgisayar olarak WindowsNT kullanılmıştır. Delphi programının seçilmesinin nedenleri:

- Delphi'de otomasyon nesneleri hazır olarak vardır. Otomasyon nesneleri COM nesnelerinin yazılmasında kullanılan , daha önce anlatılan çoğu işlemi hazır olarak içlerinde barındıran nesnelerdir.
- Delphi görsel bir dil olduğu için programlama ile uğraşmak yerine daha çok probleme ağırlık verilebilir. Delphi'nin sunduğu olanaklardan yararlanarak kullanıcı arabirimi gibi zahmetli işlerle uğraşılmaz.
- COM altyapısının ayrıntılarının bilinmesine gerek yoktur. Çünkü otomasyon nesneleri COM kütüphanesini kullanarak bir çok işlemi otomatikleştirmişlerdir.

Şimdi örnek bir programı inceleyerek adım adım COM sunucularının ve istemcilerinin nasıl yazıldığını gösterilecektir. Burada verilen örnek program (QuickSort algoritması) ileride ayrıca bölüm olarak ele alınacaktır.

5.1 COM Sunucularının Programlanması

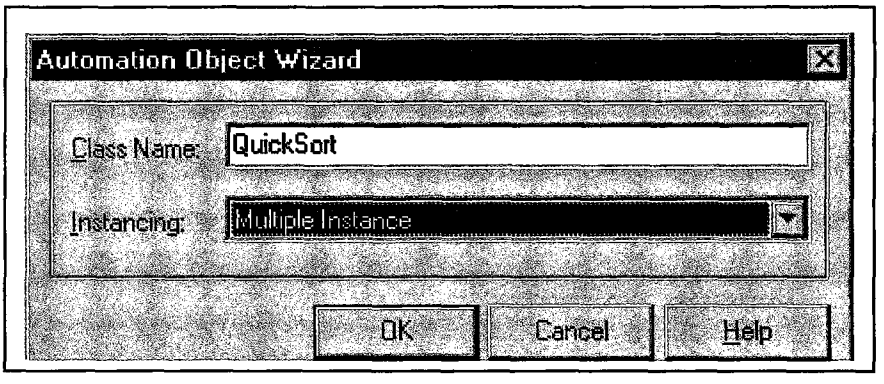
Şimdi bir COM sunucusu yazmak için gerekli işlemler gösterilecektir. Programlama dili olarak Delphi Client/Server sürümü kullanılmıştır.

- Normal bir uygulama açılır.
- Bu uygulamaya bir otomasyon nesnesi eklenir.
- Bu nesnenin tip kütüphanesi (arayüz tanımlaması) oluşturulur ve derlenir.
- Daha sonra oluşturulan bu arayüzle nesne gerçeklemesi yapılır.
- Hatasız olarak derlendikten sonra sisteme sunucu kayıt edilir.

Bu adımların yapılmasından sonra artık sunucu sistemde istemciler tarafından kullanıma hazırdır. Şimdi bu adımlar sırası ile incelenecektir.

İlk olarak Delphi editöründe File->NewApplication' menüsünden yeni bir uygulama açılır. Yeni uygulama açıldığında uygulamada sadece bir tane Form vardır.

Daha sonra bu uygulama COM sunucusu olacağından File->New'dan ActiveX menüsünden Automation Object (otomasyon nesnesi) seçilir. Bu işlemden sonra nesnenin bilgileri sorulacaktır. Aşağıdaki ekran gelecektir:



Şekil 5.1 Otomasyon Nesne Sihirbazı.

Burada sınıf ismi yerine COM nesnesinin sınıf ismi yazılmıştır. Instancing (örnekleme) seçeneği "Multiple Instance" olarak seçilmiştir daha öncede gördüğümüz gibi bu sınıftan örneklenen nesnelerin nasıl oluşturulacağını belirler. Buradaki örnekte QuickSort sınıfından birden fazla örnek yaratılabilir. Yani yaratılan sunucuya birden fazla istemci bağlanabilir. Eğer "Single Instance" olursa her istemcinin örneklediği nesne ayrı bir sunucu tarafından kontrol edilir.

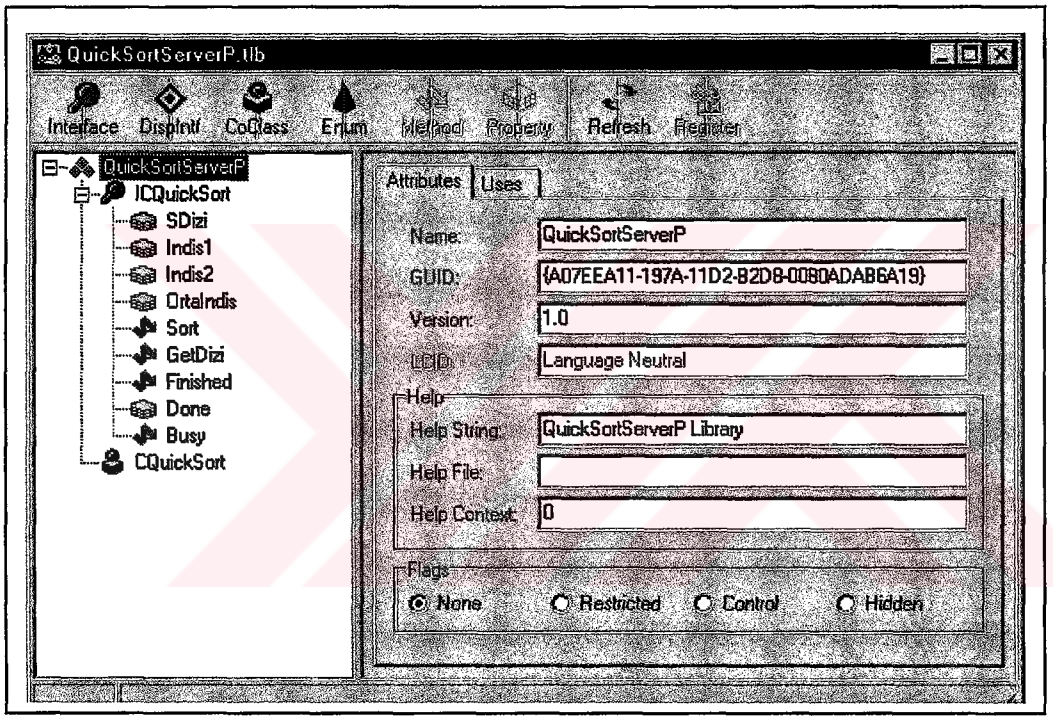
Otomasyon nesnesi uygulamaya eklendikten sonra projeye iki yeni modül eklenir. Tanımlanan sınıfın kaynak modülü ve tip kütüphanesi modülü (Type Library) eklenir.

5.1.1 Arayüz Tanımlaması ve Nesne sınıfının gerçeklemesi

COM sunucu nesnesi yazmak için ilk olarak bu nesnenin destekleyeceği arayüzler tanımlanacaktır. Arayüz tanımlaması bildiğimiz gibi IDL dilinde yazılıp daha sonra derlenerek kaynak kod üretilmektedir. Delphi'de IDL dilini bilmeden de bunu yapabiliriz. Deplhi tip kütüphanesini tanımlamak için bir tip kütüphane tanımlama editörü sunmaktadır. Bu program kullanılarak istenilen arayüzler tanımlanır ve derlenerek gerekli kaynak kod üretilir. Tip kütüphanesi tanımlama editörü Şekil 5.2'de gösterilmiştir.

Görüldüğü gibi burada tüm arayüzler tanımlanır. Örneğimizde ICQuickSort arayüzü tanımlanmıştır. Bu arayüzün içinde Sdizi, Indis1, Indis2, Ortalndis, Done adında beş tane değişken vardır. Bu değişkenler sunucunun işlemleri yaparken kullandığı değişkenlerdir. Ayrıca Sort, GetDizi, Finished, Busy adında fonksiyonlara sahiptir.

Burada tüm tanımlamalar yapıldıktan sonra Refresh tuşuna basarak kaynak kodun tekrar oluşturulması sağlanır. Yani oluşturduğumuz arayüzün kaynak kod olarak tanımlaması ilgili modülde tanımlanmıştır. Bu kaynak kod aşağıda verilmiştir.



Şekil 5.2 Arayüz tanımlama editörü.

```
unit QuickSortServerP_TLB;
{ This file contains pascal declarations imported from a type library.
  This file will be written during each import or refresh of the type
  library editor.  Changes to this file will be discarded during the
  refresh process. }
{ QuickSortServerP Library }
{ Version 1.0 }
interface
uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;
const
  LIBID_QuickSortServerP: TGUID = '{A07EEA11-197A-11D2-B2D8-0080ADAB6A19}';
const
{ Component class GUIDs }
  Class_CQuickSort: TGUID = '{A07EEA13-197A-11D2-B2D8-0080ADAB6A19}';
type
{ Forward declarations: Interfaces }
  ICQuickSort = interface;
```

```
  ICQuickSortDisp = dispinterface;
{ Forward declarations: CoClasses }
  CQuickSort = ICQuickSort;
{ Dispatch interface for QuickSort Object }
  ICQuickSort = interface(IDispatch)
    ['{A07EEA12-197A-11D2-B2D8-0080ADAB6A19}']
    function Get_SDizi: OleVariant; safecall;
    procedure Set_SDizi(Value: OleVariant); safecall;
    function Get_Indis1: Integer; safecall;
    procedure Set_Indis1(Value: Integer); safecall;
    function Get_Indis2: Integer; safecall;
    procedure Set_Indis2(Value: Integer); safecall;
    function Get_OrtaIndis: Integer; safecall;
    procedure Set_OrtaIndis(Value: Integer); safecall;
    procedure Sort(Dizi: OleVariant; I1, I2: Integer); safecall;
    function GetDizi(var Dizi: OleVariant; var I1, K, I2: Integer): Integer; safecall;
    function Finished: Integer; safecall;
    function Get_Done: Integer; safecall;
    procedure Set_Done(Value: Integer); safecall;
    function Busy: Integer; safecall;
    property SDizi: OleVariant read Get_SDizi write Set_SDizi;
    property Indis1: Integer read Get_Indis1 write Set_Indis1;
    property Indis2: Integer read Get_Indis2 write Set_Indis2;
    property OrtaIndis: Integer read Get_OrtaIndis write Set_OrtaIndis;
    property Done: Integer read Get_Done write Set_Done;
  end;
{ DispInterface declaration for Dual Interface ICQuickSort }
ICQuickSortDisp = dispinterface
    ['{A07EEA12-197A-11D2-B2D8-0080ADAB6A19}']
    property SDizi: OleVariant dispid 1;
    property Indis1: Integer dispid 2;
    property Indis2: Integer dispid 3;
    property OrtaIndis: Integer dispid 4;
    procedure Sort(Dizi: OleVariant; I1, I2: Integer); dispid 5;
    function GetDizi(var Dizi: OleVariant; var I1, K, I2: Integer): Integer; dispid 6;
    function Finished: Integer; dispid 7;
    property Done: Integer dispid 8;
    function Busy: Integer; dispid 9;
  end;
{ QuickSortObject }
  CoCQuickSort = class
    class function Create: ICQuickSort;
    class function CreateRemote(const MachineName: string): ICQuickSort;
  end;
implementation
uses ComObj;
class function CoCQuickSort.Create: ICQuickSort;
begin
  Result := CreateComObject(Class_CQuickSort) as ICQuickSort;
end;
class function CoCQuickSort.CreateRemote(const MachineName: string): ICQuickSort;
begin
  Result := CreateRemoteComObject(MachineName, Class_CQuickSort) as ICQuickSort;
end;
end.
```

Kaynak kodda sınıfın GUID numaraları tanımlanmıştır.

```
Class_CQuickSort: TGUID = '{A07EEA13-197A-11D2-B2D8-0080ADAB6A19}';
```

Bu satırda QuickSort sınıfının sınıf tanımlama numarası tanımlanmıştır. Bu numara ICQuickSort arayüzünün GUID numarasıdır.

```
['{A07EEA12-197A-11D2-B2D8-0080ADAB6A19}']
```

Görüldüğü gibi bu tip tanımlama modülü sadece arayüz tanımlamalarını yapmaktadır. Bu nesne sınıfının asıl gerçeklemesi sınıf proje modülünde yapılır. Şimdi bu modül incelenecektir.

```
unit QuickSortServer;
interface
uses
  ComObj, QuickSortServerP_TLB,QuickSiralaThread,sysutils;
type
  TCQuickSort = class(TAutoObject, ICQuickSort)
  private
    SDizi: OleVariant;
    Indis1: Integer;
    Indis2: Integer;
    OrtaIndis: Integer;
    Done :Integer;
ThSirala:SiralaThread;
    procedure ThreadFinished(Sender:TObject);
  protected
    function Finished: Integer; safecall;
    function Get_Indis1: Integer; safecall;
    function Get_Indis2: Integer; safecall;
    function Get_OrtaIndis: Integer; safecall;
    function Get_SDizi: OleVariant; safecall;
    function GetDizi(var Dizi: OleVariant; var I1, K, I2: Integer): Integer;
  safecall;
    procedure Set_Indis1(Value: Integer); safecall;
    procedure Set_Indis2(Value: Integer); safecall;
    procedure Set_OrtaIndis(Value: Integer); safecall;
    procedure Set_SDizi(Value: OleVariant); safecall;
    procedure Sort(Dizi: OleVariant; I1, I2: Integer); safecall;
    function Get_Done: Integer; safecall;
    procedure Set_Done(Value: Integer); safecall;
    function Busy: Integer; safecall;
  end;
implementation
uses ComServ, QuickSortForm;
procedure TCQuickSort.ThreadFinished(Sender:TObject);
begin
  Done:=1; {Islem tamamlandi. Ama henuz get islemi yapilmadi. }
end;
{ 0: Server bos.
  1: Server islem bitirmistir. GetDizi cagrilmali.
  2: Server busy
  }
function TCQuickSort.Finished: Integer;
begin
 Finished:=Done;
end;
function TCQuickSort.GetDizi(var Dizi: OleVariant; var I1, K,
  I2: Integer): Integer;
begin
  if (Done=0) then
```

```
   GetDizi:=-1 {Hata islem bitmedi.}
  else
   begin
    GetDizi:=1;
    Dizi:=SDizi;
    I1:=Indis1;
    K:=OrtaIndis;
    I2:=Indis2;
    done:=0; {artik server bos. Yeni bir request alabilir.}
   end;
end;
procedure TCQuickSort.Sort(Dizi: OleVariant; I1, I2: Integer);
begin
  SDizi:=Dizi;
  Indis1:=I1;
  Indis2:=I2;
  Done:=2; //Server busy.
  ThSirala:=SiralaThread.Create(SDizi,I1,I2,OrtaIndis,QuickSortServerForm.List);
  ThSirala.FreeOnTerminate:=true;
  ThSirala.OnTerminate:=ThreadFinished;
end;
procedure TCQuickSort.Set_Done(Value: Integer);
begin
 done:=Value;
end;
initialization
 TAutoObjectFactory.Create(ComServer, TCQuickSort, Class_CQuickSort, ciMultiInstance);
end.
```

Burada TCQuickSort nesnesi tanımlanmıştır. Bu nesne TautoObject ve ICQuicksort nesnelerinden türemiştir.

```
TCQuickSort = class(TAutoObject, ICQuickSort)
```

Artık burada sunucu nesnesinin yapacağı işler programlanmıştır. Mesela Sort (procedure TCQuickSort.Sort(Dizi: OleVariant; I1, I2: Integer)) fonksiyonu kendine verilen bir diziyi sıralar. COM nesneleri arasında en basit olarak parametreler OleVariant yada Integer olarak aktarılır. OleVariant özel bir değişken tipidir , bir olevariant değişkeni byte, integer, float ya da dizi gibi çok karışık değişken tiplerini içerebilir.

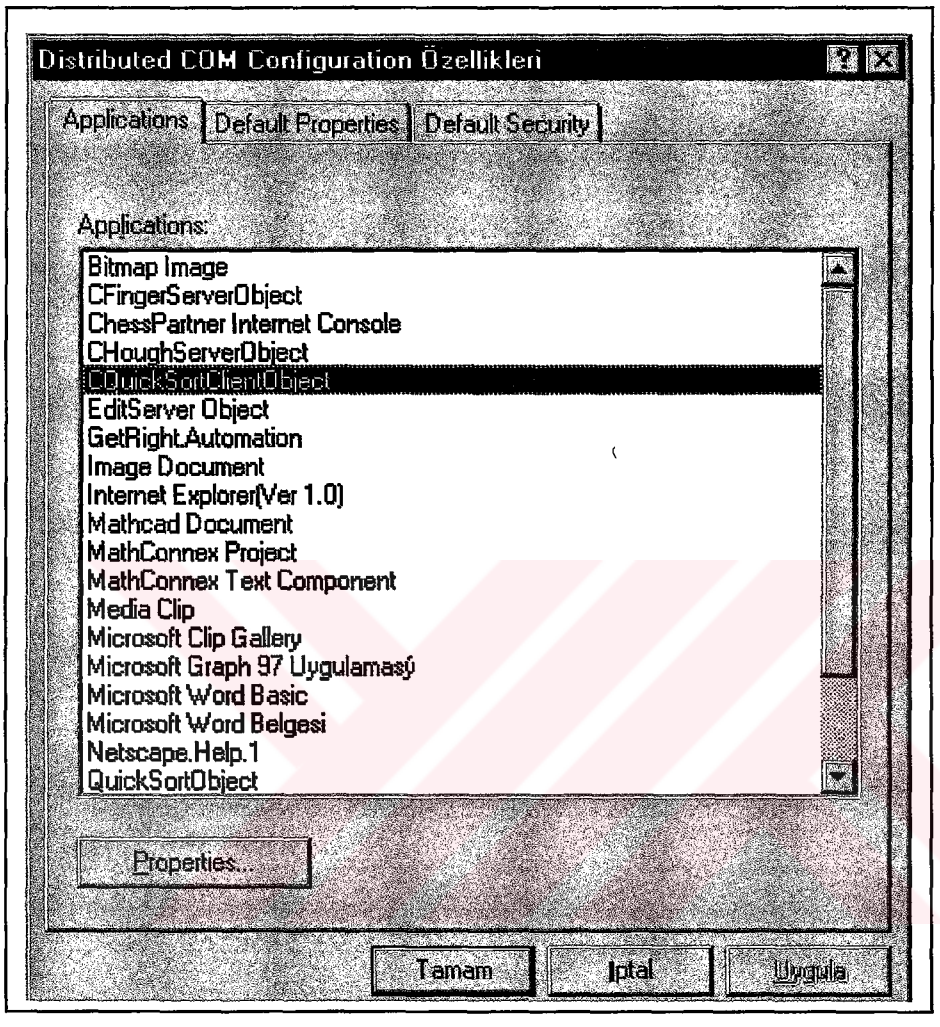
5.1.2 Sunucu nesnesinin kayıt edilmesi

Sunucu nesnesi tanımlama ve gerçekleme işlemleri bittikten sonra derleme işlemi yapılır. Hatasız olarak derlendikten sonra artık sunucu siteme kayıt edilmeye hazır demektir. Bunun için komut satırında parametre olarak /regserver girilir.

```
C:\Quickserver.exe /regserver
```

Bu işlem başarı ile bittikten sonra nesne sisteme kayıt edilmiştir. Sisteme kayıt edildikten sonra bu nesne için erişim denetimlerinin ayarlanması gerekir. Bunun için

DCOMCNFG.EXE programı kullanılır. Bu program Windows NT altında DCOM ile birlikte gelen konfigürasyon programıdır. Aşağıda örnek bir ekran gösterilmiştir.



Şekil 5.3 COM sunucularının konfigürasyon ayarları.

Bu listede sisteme kayıtlı olan tüm sunucu nesneleri görülmektedir. Herhangi bir nesnenin özelliklerine girilerek bağlantı ayarları yapılabilir. Örnek olarak uzaktan erişim DCOM desteği olup olmaması, güvenlik kontrolünün çağrı yada bağlantı

yaratmaktır. COM istemcileri sunucuda tanımlanan tip kütüphanesi modülünü kullanmak zorundadırlar. Sunucuyu yazan taraf bu sunucu nesnelerini kullanacak kullanıcılara bu modülü vermek zorundadırlar. Aksi taktirde istemciler bu nesnelere bağlamazlar.

Şimdi örnek programda istemci programı incelenecektir. Programın kodu aşağıda verilmiştir.

```
unit QuickSortClientForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,QuickSortServerP_TLB,
  StdCtrls,Stack;
 const
{ Component class GUIDs }
  IID_ICQuickSort: TGUID = '{A07EEA12-197A-11D2-B2D8-0080ADAB6A19}';
type
  TForm1 = class(TForm)
    DiziOlustur: TButton;
    List: TListBox;
    List2: TListBox;
    Edit1: TEdit;
    Sirala: TButton;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label1: TLabel;
    lstHatalar: TListBox;
    lstServers: TListBox;
    Label2: TLabel;
    Label3: TLabel;
    edtTopZaman: TEdit;
    Label7: TLabel;
    edtNetZaman: TEdit;
    Label8: TLabel;
    edtIslemZaman: TEdit;
    procedure FormCreate(Sender: TObject);
    procedure DiziOlusturClick(Sender: TObject);
    procedure SiralaClick(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
    Pkunk   : IUnknown;
    SiralaObj: array [1..20] of CQuickSort;
    ObjN    :Integer;
    Dizi    :OleVariant;
    Dizi2    :OleVariant;
    N       :Integer;
    Indis1,Indis2,OrtaIndis:Integer;
    Stak    :ClassStack;
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
uses ComObj, HoughClientGrafikForm;
{$R *.DFM}
```

```
procedure TForm1.FormCreate(Sender: TObject);
var
 f:TextFile;
 str:String;
begin
 AssignFile(f,'Servers.txt');
 Reset(f);
 ObjN:=1;
 while (not eof(f)) do
  begin
   ReadLn(f,str);
   try
    Pkunk:=CreateRemoteComObject(str,Class_CQuickSort);
   except
    //ShowMessage('Sirala Class yaratilamadi....');
    str:=str+' server error.';
    lstHatalar.Items.Add(str);
    Continue;
   end;
   Pkunk.QueryInterface(IID_ICQuickSort,SiralaObj[ObjN]);
   if (SiralaObj[ObjN]=nil) then
    begin
     //ShowMessage('Interface Desteklenmiyor...');
     Pkunk._Release;
     str:=str+' instance not supported.';
     lstHatalar.Items.Add(str);
     Continue;
    end;
   ObjN:=ObjN+1;
   lstServers.Items.Add(str);
  end;
  CloseFile(f);
  ObjN:=ObjN-1;
 Stak:=ClassStack.Create(2000);
end;
procedure TForm1.DiziOlusturClick(Sender: TObject);
var
 i:integer;
begin
 N:=StrToInt(Edit1.Text);
 Dizi := VarArrayCreate([1,N+5],varInteger);
 Dizi2:= VarArrayCreate([1,N+5],varInteger);
 Randomize;
 List.Items.Clear;
 for i:=1 to N do
  begin
   Dizi[i]:=Random(1000);
   List.Items.Add(Dizi[i]);
  end;
end;
function ZamanHesapla(T1,T2:TDateTime):Integer;
 var
  Hour1, Min1, Sec1, MSec1: Word;
  Hour2, Min2, Sec2, MSec2: Word;
  S1,S2:Integer;
 begin
  DecodeTime(T1, Hour1, Min1, Sec1, MSec1);
  DecodeTime(T2, Hour2, Min2, Sec2, MSec2);
  S1:=MSec1+Sec1*1000+Min1*60*1000+Hour1*60*60*1000;
  S2:=MSec2+Sec2*1000+Min2*60*1000+Hour2*60*60*1000;
  ZamanHesapla:=S2-S1;
 end;
```

```
//Iki zaman arasindaki farki sanise farki ile dondurur.
procedure DecodeZaman(S:Integer; var dakka,saniye,sanise:Integer);
 begin
  dakka:=S div 60000;
  S:=S mod 60000;
  saniye:=S div 1000;
  S:=S mod 1000;
  sanise:=S;
 end;
procedure TForm1.SiralaClick(Sender: TObject);
var
 i,j:Integer;
 done:Boolean;
 c:integer;
 B,S,NB,NS        :TDateTime;
 IslemZaman,NetZaman,TopZaman:Integer;
 Dakka,Saniye,Sanise,Sure:Integer;
begin
 IslemZaman:=0; //Server da yapilan islem suresi optime.
 NetZaman  :=0; //Network transfer zamani
 TopZaman  :=0; //Clienttaki toplam sure.
 B:=Now;
 done:=false;
 Stak.Push(1,N);
 for i:=1 to ObjN do
  SiralaObj[i].Set_Done(0); {server durumunu sifirla.}
 while (not done) do
  begin
   done:=True;
   for i:=1 to ObjN do
    begin
     c:=SiralaObj[i].Finished;
     if c=1 then {eger islem bitmisse.}
      begin
       NB:=Now;
       SiralaObj[i].GetDizi(Dizi2,Indis1,OrtaIndis,Indis2);
       NS:=Now;
       NetZaman:=NetZaman+ZamanHesapla(NB,NS);
       for j:=Indis1 to Indis2 do
         Dizi[j]:=Dizi2[j];
       if (Indis1 < OrtaIndis-1) then
         Stak.Push(Indis1,OrtaIndis-1);
       if (Indis2 > OrtaIndis+1) then
          Stak.Push(OrtaIndis+1,Indis2);
       if (not Stak.Empty) then
         done:=false;
      end
     else if (c=0) then {eger server bossa}
      begin
       if (Stak.Pop(Indis1,Indis2)) then {eger stak dolu ise}
         begin
          NB:=Now;
          SiralaObj[i].Sort(Dizi,Indis1,Indis2);
          NS:=Now;
          NetZaman:=NetZaman+ZamanHesapla(NB,NS);
          done:=false;
         end;
      end
     else
      done:=false;
    end;
 end;
```

```
  S:=Now;
  TopZaman:=ZamanHesapla(B,S);
  IslemZaman:=TopZaman-NetZaman;
  DecodeZaman(TopZaman,Dakka,Saniye,Sanise); //toplam islem suresi
  edtTopZaman.Text:=IntToStr(dakka)+':'+IntToStr(saniye)+':'+IntToStr(sanise);
  DecodeZaman(NetZaman,Dakka,Saniye,Sanise);   //Network transfer zamani
  edtNetZaman.Text:=IntToStr(dakka)+':'+IntToStr(saniye)+':'+IntToStr(sanise);
  DecodeZaman(IslemZaman,Dakka,Saniye,Sanise); //Serverlarin islem zamanlari
  edtIslemZaman.Text:=IntToStr(dakka)+':'+IntToStr(saniye)+':'+IntToStr(sanise);
  GrafikForm.TopZaman:=TopZaman;
  GrafikForm.NetZaman:=NetZaman;
  GrafikForm.IslemZaman:=IslemZaman;
  GrafikForm.Show;
  List2.Items.Clear;
  for i:=1 to N do
    List2.Items.Add(Dizi[i]);
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  SiralaObj[1].Get_OrtaIndis;
end;
end.
```

İstemci programı tip kütüphanesini kullanır. İlk olarak program açıldığında FormCreate fonksiyonunda sunucu nesneler oluşturulur.

```
Pkunk:=CreateRemoteComObject(str,Class_CQuickSort);
```

CreateRemoteComObject uzaktaki bir makinede istenilen nesneyi yaratır. Burada str makinenin ağdaki TCP/IP numarasıdır. Görüldüğü gibi bu fonksiyon istenilen makine üzerinde CQuickSort tipinde bir nesne yaratmaktadır ve bu yaratılan nesnenin IUnknown arayüz işaretçisi geri gelir. Pkunk değişkeni IUnknown tipinde bir değişkendir. Fakat henüz bu nesnenin Sort gibi fonksiyonlarına erişilemez. Çünkü bu fonksiyonlar bu nesnenin ICQuickSort arayüzünde bulunmaktadır. O zaman bu arayüze işaretçi alınmalıdır.

Bu işlemi de daha önceden de bilindiği üzere IUnknown arayüzünde bulunan QueryInterface değişkeni yapar.

```
Pkunk.QueryInterface(IID_ICQuickSort,SiralaObj[ObjN]);
```

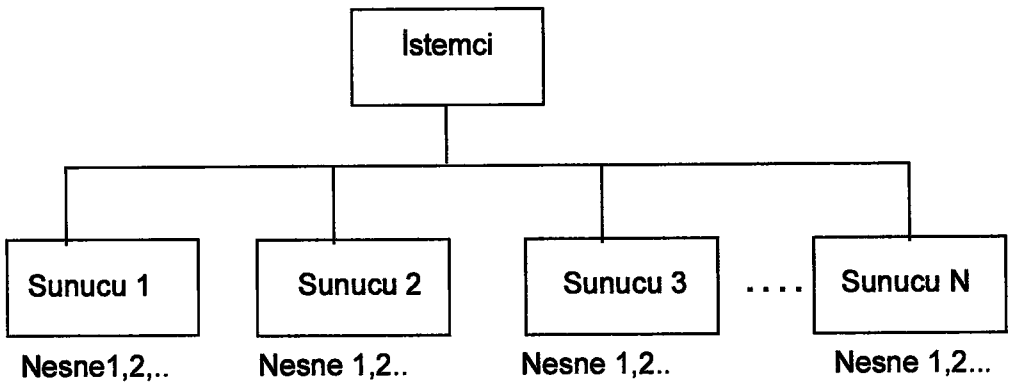
Pkunk arayüz işaretçisi üzerinde bu fonksiyonu çağrılır ve içine parametre olarak istenilen arayüzün GUID numarası verilir Eğer yaratılan nesne bu arayüzü destekliyorsa geriye SiralaObj[ObjN] değişkeninde arayüz işaretçisi gelecektir. Aksi taktirde Nil değeri gelir ve arayüzün desteklenmediğini anlaşılır. Artık bu aşamadan sonra SiralaObj[ObjN].Sort(...) fonksiyonu çağırıp nesneye istenilen işlem yaptırılabilir.

6. GENEL PARALELLEŞTİRME ALGORİTMASI

QuickSort algoritmasının dağıtık nesneler ile nasıl gerçeklendiğine geçmeden önce, genel olarak bir problemin nasıl paralel işleme uygun hale getirildiği incelenecektir. Burada anlatılan algoritma dağıtık nesneler üzerinde paralel olarak geliştirilecek her türlü sistemler için kullanılabilir.

Algoritma kuyruk (queue) yapısına dayanmaktadır. Öncelikle çözülmesi gereken problem bir birinden bağımsız alt problemlere ayrılır. Birbirinden bağımsız olmasının nedeni , işlemcilerin birbirleriyle iletişimi için özel bir donanım olmamasındandır. Eğer bu alt işlemler birbirine bağlı ise paralel çalışan nesneler arası iletişim sıklığı yüzünden yavaşlamaya sebep olacaktır. Problem gerektiği gibi bölündükten sonra , yapılması gereken alt işlemler işlenmek üzere bir kuyruğa atılır. Bazı algoritmalarda kuyruk kullanmak gerekmeyebilir , ancak dağıtık nesne modelinde hata giderme işleminin yapılabilmesi için kuyruk yapısı seçilmiştir.

Tüm işlemler kuyruğa atıldıktan sonra , dinamik olarak dağıtık nesneler oluşturulur ve kuyruktan yapılması gereken işler nesnelere dağıtılır. Böylece dağıtılan işler paralel olarak işlenmeye başlanır. İşini bitiren nesne sonucu iletir ve eğer yapılması gereken başka iş varsa kuyruktan çekilerek bu nesneye verilir. Ve tüm işler bitene kadar döngü içinde devam eder. Gelen cevapları da uygun şekilde saklayarak , gerekli birleştirme işlemleri de yapılarak problemin çözümüne ulaşılır. Paralel nesnelerin çalıştırıldığı istemci sunucu modeli Şekil 6.1'de gösterilmiştir.



Şekil 6.1 İstemci sunucu modeli.

Bilgisayarlardan en az bir tanesi istemci durumunda diğerleri sunucu durumundadır. İstemci bilgisayarda yukarıda anlatılan algoritma çalışır, sunucu bilgisayarlarda ise COM nesneleri (yani asil işlemleri yapan program) vardır.

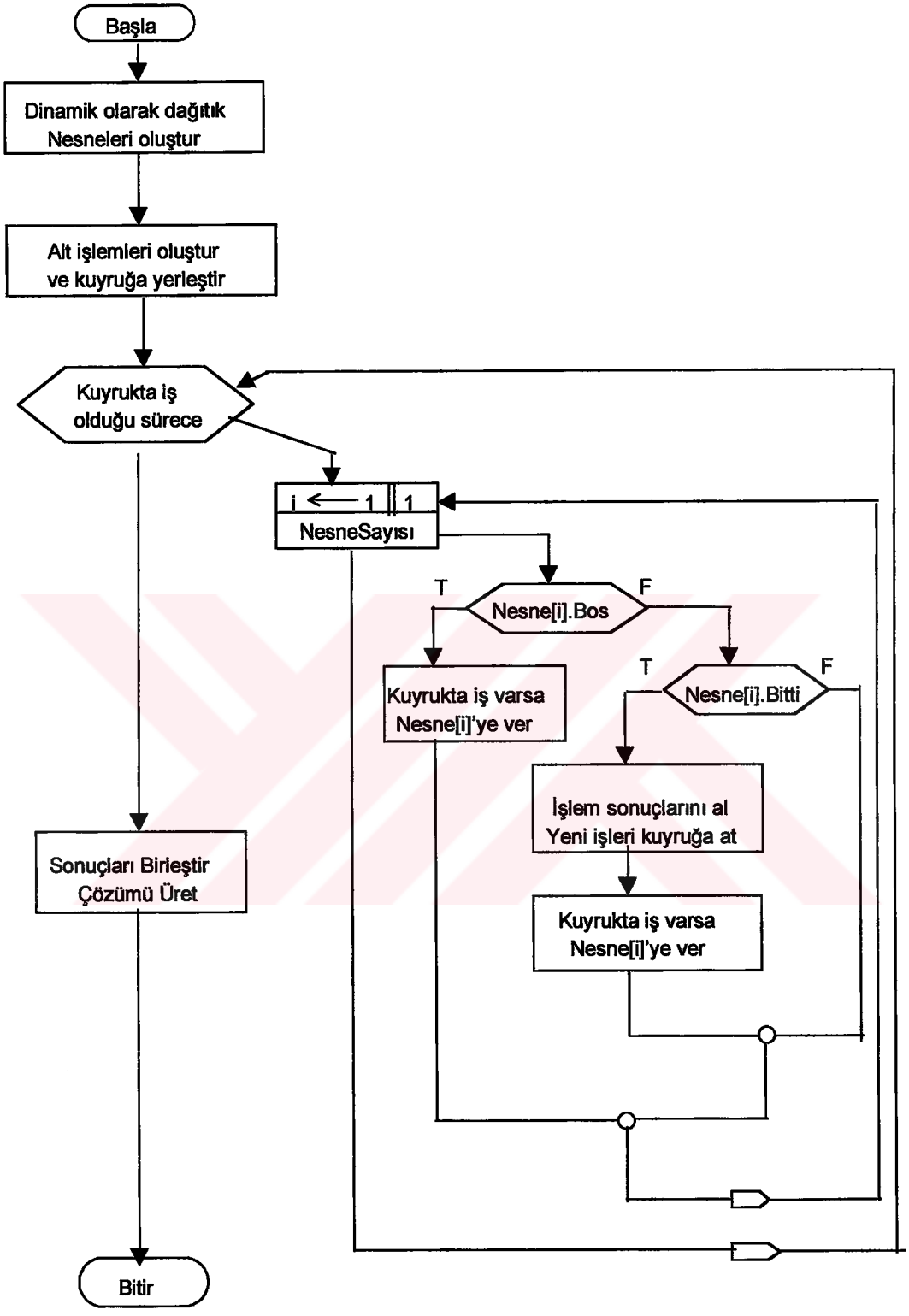
Genel Paralelleştirme algoritmasının akış diyagramı Şekil 6.2'de verilmiştir. İstemci programların işlemleri dağıtıp , paralel olarak sonuç üretebilmesi için yukarıda anlatılan algoritma kullanılmıştır. Şimdi bu yapı kullanılarak QuickSort sıralama algoritmasının nasıl paralel olarak yapıldığı incelenecektir.

İstemciler tarafından çalıştırılan algoritma:

```
Sunucu nesnelerini oluştur, diziye yerleştir server[] (#N sunucu).
Alt işlemleri oluştur ve iş kuyruğuna yerleştir.
While (kuyruk dolu) do
   For i = 1 to #N do  (tüm sunucular için)
     If (Server[i] çalışmıyor) then
       Ona verilen işi kuyruğa geri koy
     Else If   (Server[i] boşta) then
        If   (kuyrukta bekleyen iş var) then
              Server[i]'ye ver,
        Else If  (Server[i] işi bitmisse) then
              Sonuçları al Server[i]'den,
              Eğer yeni iş oluşmussa kuyruğa at
              If   (kuyrukta bekleyen iş varsa) then
                Server[i]'ye ver,
              End If
        End If
     End If
   End For
End While
Sonuçları birleştir ve problemin çözümünü oluştur.
```

Sunucular tarafından çalıştırılan algoritma:

```
d*d boyutundaki alt resmi al.
Bu resimden , tum r degerleri icin accumulator matrisini A'yı hesapla
Bu matristeki yerel maksimumları bul, cemberlerin koordinatları.
```



Şekil 6.2 Genel Paralelleştirme Algoritması.

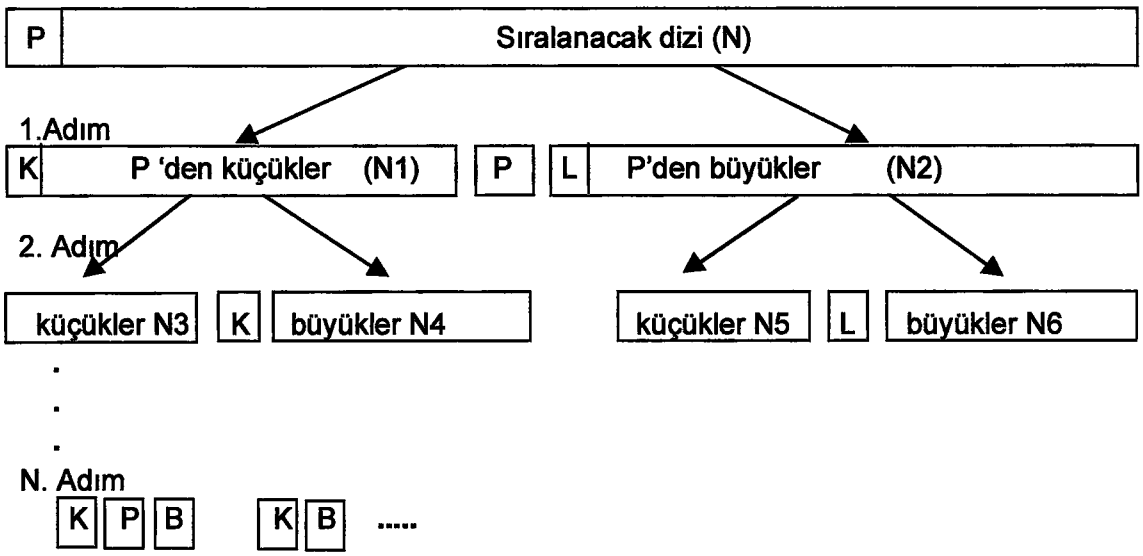
7. ÖRNEK UYGULAMA VE SONUÇLARI - QUICKSORT İŞLEMİNİN PARALEL OLARAK GERÇEKLENMESİ

7.1 QuickSort Algoritması

QuickSort sıralama algoritması doğası gereği paralel olarak çalışmaya uygun bir algoritmadır. Algoritma kısaca şu şekilde çalışır:

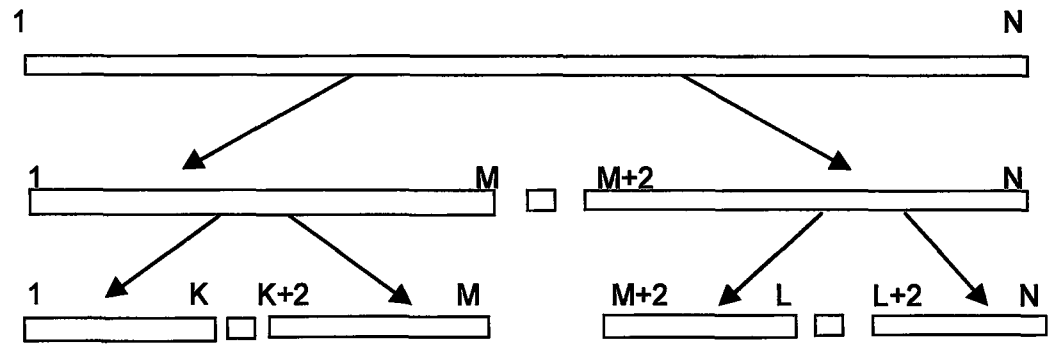
N boyutunda sıralanacak bir dizi vardır. Bu dizi küçükler başa ve büyükler sona gelecek şekilde tekrar düzenlenir. Böylece dizi ikiye bölünmüş olunur. Sonra bu iki dizi üzerinde de aynı işlem dizi eleman sayısı 2'ye inene kadar tekrarlanır. Tüm diziler bittikten sonra N boyutundaki dizimiz sıralanmış olur.

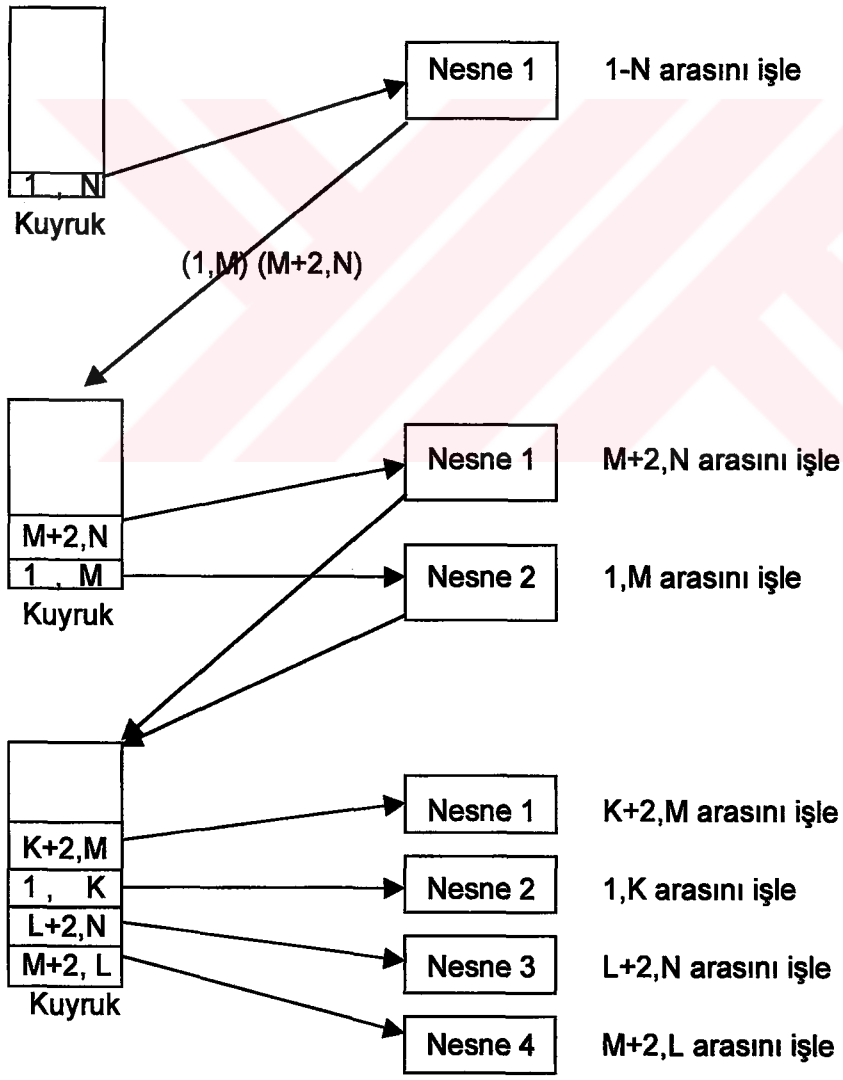
Görüldüğü gibi ilk başta 1 dizi vardır, ilk işlemden sonra dizi sayısı 2 ye çıkar. İşte bu andan itibaren paralel çalışma işlemi başlayabilir, yani bir dizi bir COM nesnesine diğer dizi de diğer bir nesneye verilerek işlemlerin paralel yapılması sağlanır. 3. adımda bu iki dizi de ikiye bölüneceği için sıralanacak 4 dizi olur ve bunlar da paralel olarak işlenebilir.



Şekil 7.1 QuickSort algoritmasının işleyişi.

7.2 QuickSort Algoritmasının Paralel İşleyişi





Şekil 7.2 QuickSort algoritmasının paralel işleyişi.

Görüldüğü gibi ilk başta paralel nesnelerden sadece biri çalışmakta ve 1-N arasını sıralar sonra iki dizi oluşur ve bunlar iki nesne tarafından paralel olarak işlenir sonra dört dizi oluşur ve bunlar da yine dört nesne tarafından paralel olarak işlenir. Böylece eleman sayısı 2 ye inene kadar paralel olarak işlenir.

7.3 Sonuçlar

Aşağıdaki tabloda QuickSort algoritmasının dağıtık nesnelerle paralel olarak çalıştırılmasıyla elde sonuçlar verilmiştir.

Tablo 7.1 Paralel QuickSort algoritmasının test sonuçları.

Sunucu	Nesne Sayısı	Dizi Boyutu	Sıralama Zamani (sn)
A	1	200	13
A	2	200	10
B	1	200	5
A B	2 2	200	5
A B	3 3	200	5
C (İstemci de)	1	200	1
C	2	200	1

Sunucular: Test için kullanılan sunucu bilgisayarların özellikleri aşağıda verilmiştir.

- A : 16 MB , Pentium 100. Win 95.
- B : 40 MB ,Pentium 100. Win 95.
- C : 32 MB , Pentium II 266 WinNT 4.0

QuickSort'un paralel olarak çalışma testi 3 bilgisayarda yapılmıştır. İlk olarak sıralama işlemi tekbir bilgisayarda paralel olmadan çalıştırılmıştır bu işlem 1 saniyede bitirilmiştir. Fakat A ve B bilgisayarlarında yaratılan sunucu nesneleri ile paralel bir şekilde yapıldığı zaman 5 saniyede yapılmıştır. Görüldüğü gibi bu sonuç istendiği gibi değildir paralel işlemin daha hızlı olması gerekirken daha yavaş bir sonuç alınmıştır. Bu da QuickSort algoritmasının bu modele uygun olmadığındandır.

Çok fazla ağ trafiği olduğu için işlemden çok veri transferi olmaktadır, bu gibi algoritmalar bu modelle paralel olarak gerçeklenemezler. Bu modele uygun olan algoritmalar çok fazla bilgi içermeyen fakat çok fazla işlem gerektiren algoritmalardır. Görüntü işleme algoritmaları bu tür işlemlerdir. Bir görüntü üzerinde çok fazla işlem

yapılmaktadır. Bu modelin görüntü işleme algoritmalarına uygulanışı ve sonuçları ilerdeki bölümlerde verilmiştir.

7.4 Karmaşıklık Hesabı

Normal QuickSort karmaşıklığı: Bilindiği gibi quicksort'un karmaşıklığı aşağıdaki gibi hesaplanır. Her adımda işlenecek dizi boyutu yarıya düştüğü için yapılan karşılaştırma sayısı NlogN olarak hesaplanır.

$$2^0\frac{N}{2^0} + 2^1\frac{N}{2^1} + 2^2\frac{N}{2^2} \cdots\cdots + 2^m\frac{N}{2^m} = (m+1)N$$

$$\frac{N}{2^m} = 2 \Rightarrow m = \log_2 \frac{N}{2}. \qquad \text{Karmaşıklık : } \mathbf{N \log N} \tag{7.1}$$

Paralel QuickSort karmaşıklığı: Quicksort işlemi paralel olarak yapıldığında her adımdaki oluşan diziler farklı makinelerde paralel olarak yapıldığı için aşağıdaki gibi hesaplanır.

Not : Buradaki paralel karmaşıklık hesabı optimum değerdir. Yani N/2 tane paralel olarak çalışan QuickSort nesnesi olduğu varsayılmıştır.

$$\frac{N}{2^0} + \frac{N}{2^1} + \frac{N}{2^2} \cdots\cdots + \frac{N}{2^m} = \frac{2^0N + 2^1N + 2^2N + \cdots\cdots + 2^mN}{2^m} = \frac{N(2^{m+1} - 1)}{2^m}$$

$$m = \log\frac{N}{2} = \log N - 1$$

$$\frac{N(2^{\log N} - 1)}{2^{\log N - 1}} = \frac{N(N-1)}{N/2} = 2N - 2 \qquad \text{Karmaşıklık : } \mathbf{2N - 2} \tag{7.2}$$

Aşağıdaki tabloda değişik N değerleri için normal ve paralel quicksort algoritmasının karmaşıklığı verilmiştir. Görüldüğü gibi paralel karmaşıklık çok daha düşük olduğu halde pratikte bu kazancı elde edemiyoruz. Nesneler arasındaki bilgi transferi sırasında geçen zaman çok fazla olduğu için bu sonuçlar elde edilemiyor.

Tablo 7.2 QuickSort'un Karmaşıklığı.

N	NlogN	2N-2
2	2	2
4	8	6
8	24	14
32	160	62
256	2048	510
1024	10240	2046

8. MODELİN GÖRÜNTÜ İŞLEMEYE UYGULANMASI

Burada geliştirilen paralel işlem modelinin verimli bir şekilde çalışabilmesi için işlenecek verinin az olması fakat işlem süresinin uzun olması gereklidir. Dolayısıyla görüntü işleme bu modele en uygun alandır. Görüntü işlemede özellikle örüntü tanıma işlemleri çok fazla hesaplama gerektiren ve doğası gereği paralel işleme uygundur.

8.1 Genel Model Tasarımı

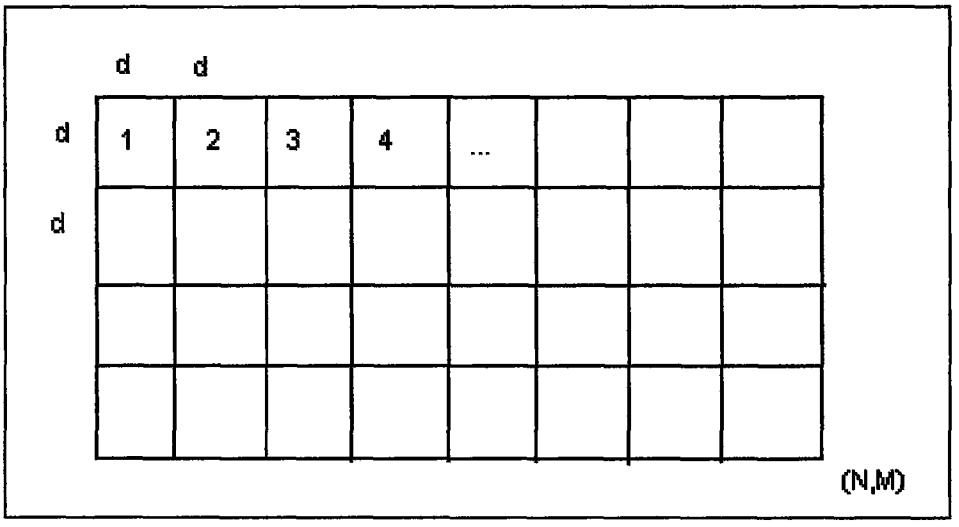
Şimdi yukarıda anlatılan genel paralelleştirme algoritmasının görüntü işleme algoritmalarına nasıl uygulandığı incelenecektir.

Bilindiği gibi genel paralelleştirme algoritmasında en önemli kısım, birbirinden mümkün olduğunca bağımsız alt işlemlerin oluşturulması ve bu işlemlerin sunuculara dağıtılmasıdır. Daha sonra sunucularda bu işlemlerin paralel olarak çalıştırılıp elde edilen sonuçların toplanması genel olarak tüm uygulamalarda aynıdır.

Şimdi bir görüntü işleme algoritması için genel model tasarımı gösterilecektir. Bu model tüm görüntü işleme problemlerinde bağımsız alt işlemleri oluşturmada kullanılır.

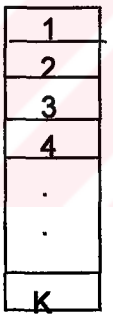
Görüntü işleme problemlerinde çoğunlukla komşuluklar önemlidir yani bir pixel'in etrafındaki diğer pixel değerlerine bakılarak hesaplamalar yapılır. Ancak birbirinden çok uzak bulunan pixellerin birbirlerini etkileme durumu yoktur. Dolayısıyla bir resmi alt resimlere ayırarak tüm resim için yapılan hesaplamalar sadece bu alt resimler için yapılıp sonuçlar birleştirilebilir.

Örnek olarak NxM boyutundaki (boyu N, eni M pixel büyüklüğünde) bir resim aşağıdaki şekilde gösterildiği gibi bölünür. d alt resimlerin boyutudur. Yani artık n tane dxd boyutunda alt resimler oluşmuştur.



Şekil 8.1 NxN boyutundaki resmin dxd boyutunda alt resimlere bölünmesi.

(N x M) boyutundaki resmi şekildeki gibi (d x d) boyutlu küçük parçalara ayırdıktan sonra oluşan alt işlemler kuyruğa atılır. Kuyrukta işlenecek olan resim parçasının numarası vardır. Buradaki 1 bölümlenen resimdeki ilk alt resmi göstermektedir, 2 ikinci resmi...



İş Kuyruğu

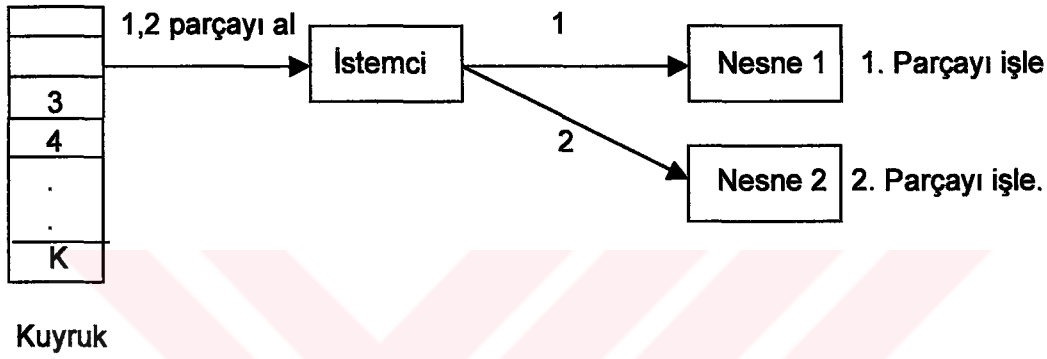
Şekil 8.2 İşlenecek olan alt resim numaralarını tutan iş kuyruğu.

Daha sonra istemci nesnesi dinamik olarak sunucu nesnelerini oluşturur. Bu sunucu nesnelerinin dinamik olarak oluşturulmasının nedeni, sunucu nesnelerinin sayısı resmin bölümleme büyüklüğünü (yani d'yi) büyük ölçüde etkilemesinden dolayıdır. Bir sonraki bölümde gösterileceği gibi paralel nesnelerin sayısı karmaşıklık hesabında önemli rol oynamaktadır.

İşlenecek resmi bu şekilde bölümledikten sonra kuyruktaki işler teker teker çekilerek boş olan nesnelere verilir. Böylece tüm nesneler aynı anda resmin değişik parçaları üzerinde ilgili algoritmayı çalıştırır. Her nesne kendine verilen parçada bulduğu

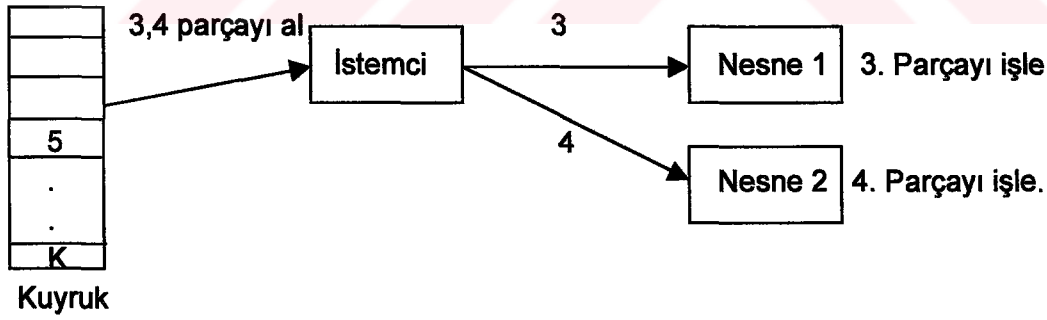
sonuçları istemciye döndürür. İstemci kuyruktaki tüm işlemleri bitirdikten sonra aldığı sonuçları birleştirir ve toplam sonucu bulunur.

Örnek olarak paralel olarak çalışan iki tane sunucu nesnesi olduğunu düşünülsün. Şekil 8.3 ve Şekil 8.4'te görüldüğü gibi istemci program kuyruktan ilk iki parçayı çeker ve nesnelere gönderir. Daha sonra nesnelerin işlemlerini bitirmesini bekler. Herhangi biri işlemi bitirdikten sonra sonuçları alır ve kuyruktan yeni bir parça alarak (eğer kalmışsa) işlemesi için tekrar ona gönderir. Böylece tüm parçalar bitene kadar işlem devam eder.



Şekil 8.3 Alt işlemlerin paralel sunuculara dağıtılması.

Nesneler işlemlerini bitirdikten sonra , istemci sonuçları alır ve yeni işlemler verir.



Şekil 8.4 Sunuculardan cevapların alınması ve yeni işlerin dağıtılması.

İki tane paralel çalışan sunucu nesnesi varsa , anlaşılacağı gibi her seferinde resmin iki parçası paralel olarak işlenmektedir. Buda yaklaşık iki kat hız kazancı demektir.

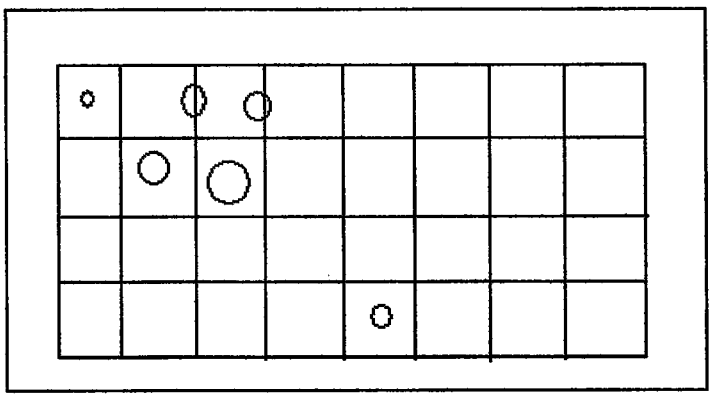
8.2 Sorunlar

Genel paralelleştirme işleminin görüntü uygulamaya uygulanması bir çok problemi de beraberinde getirir. Şimdi bu problemlerin neler olduğuna ve nasıl çözüldüğü incelenecektir.

8.2.1 Örtüşme Problemi

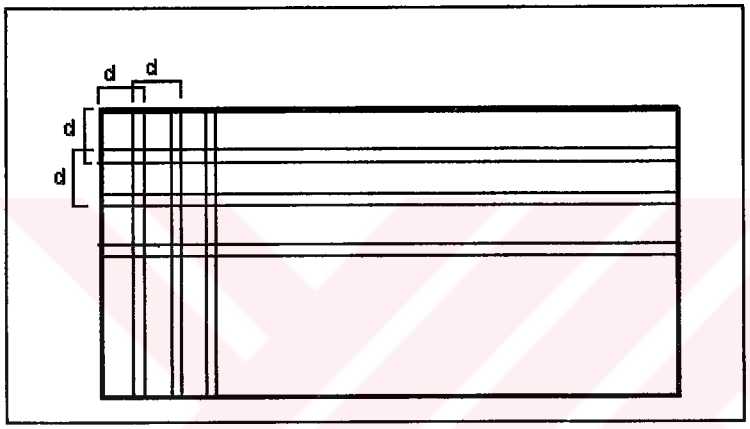
Resim alt resimlere bölündüğünde bölümler arasında (tam sınırda) kalan pixeller kesiklik gösterirler. Yani bu pixellerin komşu pixelleri kaybolur. Dolayısıyla sağlıklı sonuçlar alınamaz bazı uygulamalarda ise tamamen hatalı sonuçlar alınır.

Örnek olarak ileride ayrıntılı olarak incelenecek hough [7] dönüşümü ele alınırsa örtüşme problemi çok iyi bir şekilde anlaşılır. Hough dönüşümü bir resimdeki şekilleri bulmak için kullanılır, burada bir resimdeki çemberlerin arandığı düşünülsün. Çemberler resimde herhangi bir yerde ve herhangi bir büyüklükte olabilirler. Dolayısıyla resim genel modelde anlatıldığı gibi bölünürse bir örtüşme problemi ile karşı karşıya kalınır. Aşağıdaki şekle bakılırsa bölümler arasında bir örtüşme yapılmamıştır yani birinin bittiği yerden diğeri başlamıştır. Ama görüldüğü gibi aranan çemberlerin bazılarının yarısı bir alt resimde diğer yarısı ise diğer bir alt resimde bulunmaktadır ve bu alt resimler farklı sunucu nesneleri tarafından paralel olarak işlendiği için söz konusu çemberler hiç saptanamayacaktır.



Şekil 8.5 Örtüşme problemi. Aranan çemberler tam sınırda bulunmaktadır ve bu şekilde bulunamazlar.

Bu sorunu çözmek için bölümler belli bir miktar örtüştürülür. Yani alt resimler bir birinin içine geçerler. Bu şekilde tam sınırda kalan pixellerin komşuları da alt resme alınmış olur. Bu işlem Şekil 8.6'da gösterilmiştir. Görüldüğü gibi dxd büyüklüğündeki alt resimler birbiri ile örtüşmektedirler ve bu örtüşen kısımlar değişik nesnelerde iki kere işlenmektedirler. Hatta tam köşeye gelen kısımlar 4 tane alt resme ait olduğu için 4 defa işlenmektedirler. Bu da algoritmanın yavaşlamasına neden olmaktadır. Bu örtüşme problemi karmaşıklık hesabında kendini göstermektedir. Buradaki örtüşme büyüklüğünün ne olacağı uygulamadan uygulamaya göre değişir. Mesela ileride gösterileceği gibi hough dönüşümü işleminde örtüşme miktarı aranan çemberin yarı çapı kadar olmalıdır.



Şekil 8.6 Örtüşme problemi, örtüşen kısımlar iki kere işlenirler.

8.2.2 Bölme problemi

Resmi alt resimlere ayırma işleminde ikinci önemli problem ise bölümleme büyüklüğü (yani d'nin) ne olacağı konusudur. Eğer resim çok küçük parçalara bölünürse çok fazla yapılacak iş oluşur ve nesneler arası bilgi transfer yoğunluğu nedeniyle yavaşlama olur. Çok fazla alt resim oluşması örtüşmelerinde çok fazla olmasına dolayısıyla aynı alanların iki kere işlenmesi nedeniyle yavaşlama olur.

Diğer yandan eğer d çok fazla büyük olursa alt işlem sayısı az olur ve paralellik yok olur ve yine yavaşlamaya sebep olur.

Bir resmin optimum bölümleme büyüklüğü uygulamaya göre değişir. İleride görüleceği gibi uygulamalarda bu optimum büyüklüğün bulunmasını ve karmaşıklık hesabına etkisi incelenecektir.

Aslında pratik sonuçlardan da görüldüğü gibi, analitik olarak bulunan bu optimum değerin gerçekte optimum değer olmadığı anlaşılır.

Bu problemi çözmek için algoritmanın karmaşıklık fonksiyonu bulunur ve bu fonksiyonu minimum yapan d değeri bölümleme değeri olarak alınır.

8.2.3 Meşgul bekleme problemi

Bir diğer problem ise istemcinin sunucu nesneler üzerinde meşgul bekleme yapmasıdır. İstemci tüm nesnelere kuyruktan aldığı bir işi verir ve döngü içinde sunucu nesnelerin işlerini bitirip bitirmediğini sorar. Yani istemci devamlı olarak sunuculara işiniz bitti mi diye soru sorar ve işi biten sunucudan sonuçları alır. Bu şekildeki bir yapıda (yani meşgul bekleme) ağ performansı büyük ölçüde düşmektedir. İstemci devamlı olarak sunucu nesneleri ile haberleştiği için ağ üzerinde aşırı ve gereksiz bir bant genişliğini kullanmaktadır. Bu da diğer ağ işlemlerinin yavaşlamasına dolayısıyla ağ performansının düşmesine neden olmaktadır.

Aslında bu problemin asıl nedeni burada kullanılan sunucu istemci haberleşmesinin senkron olmasından kaynaklanmaktadır. Yani DCOM nesneleri senkron olarak haberleştirilmektedir, eğer asenkron bir haberleşme olsaydı böyle bir problem olmayacaktı. Asenkron haberleşmede istemci sunucuya işi verir ve bir daha hiç kontrol etmez sunucu işi bittiği anda istemciyi bundan haberdar eder. Ancak bu sistem DCOM nesnelerinin , algoritmaların paralelleştirme işleminde kullanımına uygun değildir. Amaç paralel olarak hızlı işlemler yapmak olduğuna göre sunucuların göçmesi durumunu da göz önünde bulundurmak gerekir. Yani işlemler sırasında herhangi bir sunucu göçerse ve ona ait iş yapılmazsa sonuç elde edilemez. Bunun için tüm sunucuların devamlı olarak kontrol edilmesi ve eğer göçen bir sunucu varsa ona verilen işin acilen başka bir sunucuya transferi yapılmalıdır bu şekilde en kısa sürede sonuca ulaşılabilir.

Tüm bu sebeplerden dolayı sistem senkron olarak tasarlanmıştır. Yukarıda anlatıldığı gibi bu sistemde de ağ performansını büyük ölçüde düşüren meşgul bekleme problemi vardır. Bu problemi çözmek içinde timer'lar kullanılmıştır. Yani sunucu nesneleri devamlı olarak kontrol edilmemektedir. İlk olarak işler sunuculara dağıtılıp bir timer vasıtasıyla istemci belli bir süre beklemektedir ve timer aktif olduğunda sunucular kontrol edilmektedir ve işi biten sunuculardan sonuçlar alınmaktadır. Bu timer'in bekleme süresi ilk başta 0 saniyedir. Ve sunucuların işlem zamanlarına göre değişen bir süredir. Sunucular işlem sonuçlarını istemciye iletirken işlemi ne kadar zamanda yaptıklarını da iletir. İstemci tüm bu zamanların ortalamasını alarak sunucuların ortalama işlem zamanını hesaplayıp timer zamanını

ona göre ayarlamaktadır. Bu şekilde ağ üzerinde bant genişliği gereksiz yere kullanılmamaktadır.

8.3 Genel Karmaşıklık Hesabı

d : alt resim büyüklüğü.
m : örtüşme büyüklüğü.
N,M : resmin büyüklüğü.
n : paralel çalışan sunucu nesne sayısı.

Bir görüntü işleme algoritmasının karmaşıklığının K(N,M) olduğunu düşünülürse. (Burada basit olarak tek geçişli bir işlem düşünülsün yani tüm pixellerin sadece 1 kere işlendiği varsayılsın o zaman ardışıl karmaşıklık NxM olur.)

Ardışıl işlem karmaşıklığı : $K = N \times M$ (8.1)

Paralel işlem karmaşıklığı: Paralel karmaşıklıkta iki durum söz konusudur. Optimum ve gerçek karmaşıklık hesapları.

Optimum karmaşıklık : $K = d \times d$ (8.2)

(Optimum karmaşıklığı hesaplarken alt resim sayısı kadar sunucu nesnesi olduğu varsayılmıştır. Yani tüm alt işlemler aynı anda paralel olarak işlenmektedir.)

Gerçek karmaşıklık: Gerçekte hiçbir zaman alt işlem sayısı kadar sunucu nesnesi olmaz. Paralel çalışan n tane sunucu nesnesi olduğu düşünülürse genel karmaşıklık aşağıdaki gibi olur.

$$K = \frac{1}{n} \frac{N \times M}{(d-m)(d-m)} d \times d \qquad (8.3)$$

Burada m örtüşme aralığıdır. Görüldüğü gibi örtüşme aralığı önemli bir rol oynamaktadır. Fonksiyondaki 2. terim (NM/(d-m)(d-m)) alt işlemlerin sayısını verir. 1/n terimi paralel sunucu nesne sayısından dolayı gelen hızlanmayı belirtir, dxd ise alt resimlerin karmaşıklığıdır.

Bu karmaşıklık genel olarak hesaplanmıştır , işte bu fonksiyondan yararlanılarak d ve m nin optimum değerleri bulunabilir ve uygulamada kullanılır. Daha sonraki uygulama bölümlerinde bu hesaplamalar gösterilecektir.

8.4 Hough Dönüşümü

Hough [7-8] transformu görüntü işlemede bir resimde belirli şekillerin (doğru, çember...) aranmasında kullanılır. Hough transformun genelleştirilmiş [8] bir hali herhangi bir şekildeki nesnenin de aranmasında kullanılır. Kısaca kullanım alanları: tıp alanında tümörlerin otomatik olarak bulunmasında, askeri fotoğraflarda düşmana ait araçların ve yerlerin tespitinde, hatta bir aracın sürücü olmadan gidebilmesi için de hough dönüşümü kullanılır.

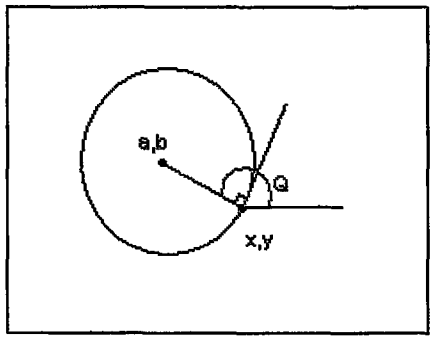
Projede hough transformu bir resimdeki çemberleri bulmak için kullanılmıştır. Aşağıda çemberleri bulmak için hough transformunun kullanımı açıklanmıştır.

Çember denklemi (8.4)'te gösterilmiştir.

(a,b) : çemberin merkezi
r : yarıçap.

$$(x-a)^2 + (y-b)^2 = r^2 \qquad (8.4)$$

Bu denklemde aranan değerler a , b , r dir. Yani r yarıçapındaki bir çemberin merkezi aranmaktadır. Çembere ait noktaların x , y koordinat değerleri zaten resimden bellidir. Ancak bu noktaların oluşturduğu çemberin yarı çapı ve merkez koordinatı (a,b) belli değildir.



Şekil 8.7 Bir Çemberin gösterimi.

Şekil 8.7'deki çemberin merkez koordinatları,

$$a = x + r \cos Q$$
$$b = y + r \sin Q \qquad (8.5)$$

olarak hesaplanır. Bu denklemlerde (x,y) degeri bellidir , noktanın resimdeki koordinatlarıdır, Q açısı da resimde o noktadaki teğet açısıdır ve komşuluk değerlerine bakılarak hesaplanır. Dolayısıyla üç bilinmeyen vardır. Bu denklemlerde tüm (x,y) noktaları için sabit bir r değeri için Q açısı hesaplanır ve (a,b) değerleri bulunur ve bir matriste bu deler artırılır.

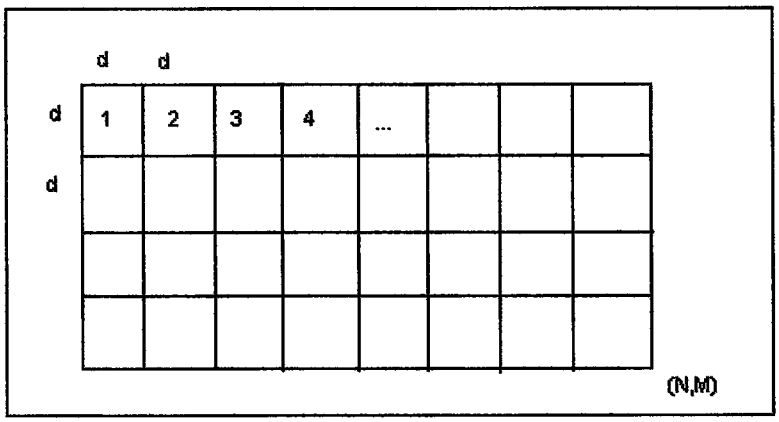
$$A(a, b) = A(a, b) + 1 \qquad (8.6)$$

Belli bir r değeri için tüm noktalar işlendikten sonra A toplama matrisindeki yerel maksimum noktalar bulunur ve bu nokta koordinatları olan (a,b) değeri r yarıçaplı bir çemberin merkezini belirtir. Aynı işlemler r değeri artırılarak tekrarlanır ve istenilen yarıçaplı tüm çemberler bulunur.

Görüldüğü üzere resimdeki her noktadaki komşu noktalara da bakılarak açı hesaplaması yapılır. Sadece komşu noktalar birbiriyle etkileşirler uzak noktadaki pixeller birbirini etkilemezler. Dolayısıyla genel paralel algoritmasını kullanarak dağıtık nesnelerle bu işlem paralel bir şekilde yapılabilir. Şimdi hough dönüşümünün paralel olarak nasıl yapıldığı incelenecektir.

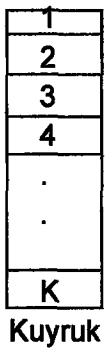
8.4.1 Paralel Hough Dönüşümünün İşleyişi

Şimdi genel paralelleştirme algoritmasını kullanarak bir resimdeki çemberleri bulmak için hough transformu işleminin dağıtık nesnelerle nasıl gerçeklendiği incelenecektir. Amaç 1 ile verilen bir r arasındaki yarı çaplı tüm çemberleri bulmak.



Şekil 8.8 Resmin alt resimlere bölünmesi.

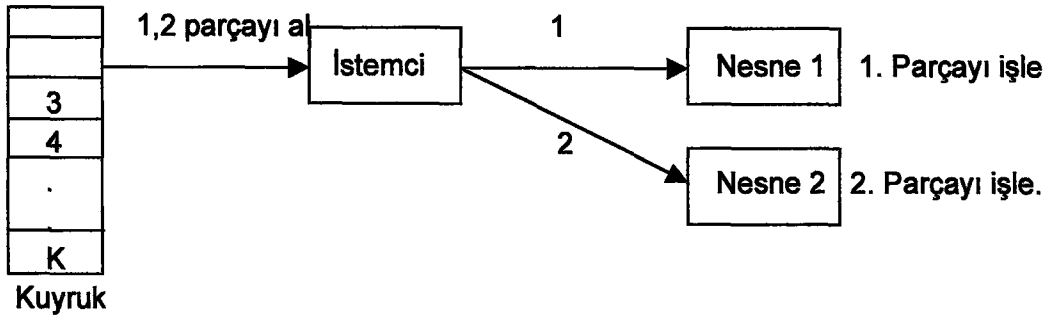
İlk iş olarak (N x M) boyutundaki resim Şekil 8.8'deki gibi (d x d) boyutlu küçük parçalara ayrılır. Oluşan alanlar kuyruğa atılır. Kuyrukta işlenecek olan resim parçasının numarası vardır.



Şekil 8.9 Alt işlemleri tutan kuyruk.

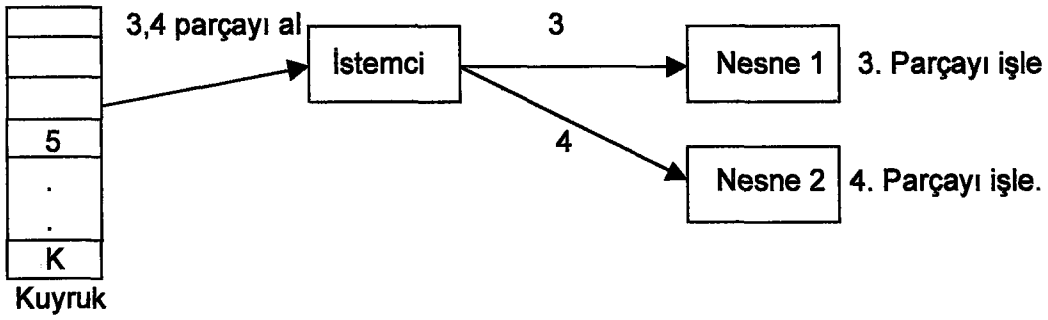
İşlenecek resmi bu şekilde bölümledikten sonra kuyruktaki işler teker teker çekilerek boş olan hough nesnelerine verilir. Böylece tüm nesneler aynı anda resmin değişik parçaları üzerinde hough transformunu yapmış olur. Her nesne kendine verilen parçada bulduğu çemberlerin koordinatlarını ve yarıçapını istemciye döndürür. İstemci kuyruktaki tüm işlemleri bitirdikten sonra aldığı koordinatları gerçek koordinatlara çevirerek birleştirir ve tüm resimdeki çemberler bulunur.

Örnek olarak paralel olarak çalışan iki tane hough nesnesi olduğu düşünülsün. Şekil 8.10'da görüldüğü gibi istemci program kuyruktan ilk iki parçayı çeker ve nesnelere gönderir. Daha sonra nesnelerin işlemlerini bitirmesini bekler. Herhangi biri işlemi bitirdikten sonra sonuçları alır ve kuyruktan yeni bir parça alarak (eğer kalmışsa) işlemesi için tekrar ona gönderir. Böylece tüm parçalar bitene kadar işlem devam eder.



Şekil 8.10 Alt resimlerin hough nesnelerine dağıtılması.

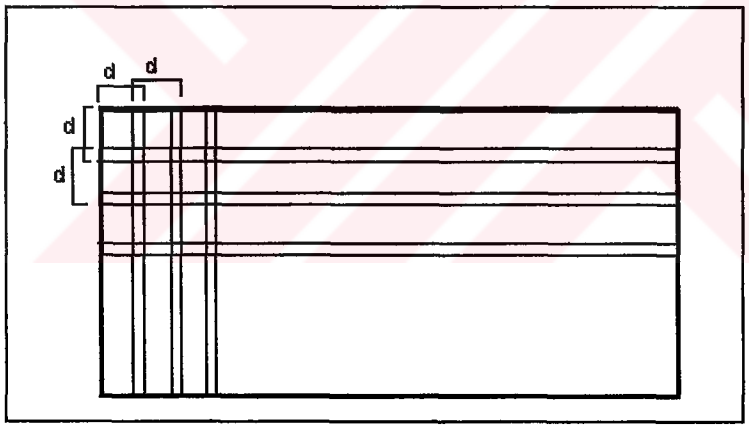
Nesneler işlemlerini bitirdikten sonra , istemci sonuçları alır ve yeni işlemler verir.



Şekil 8.11 Hough nesnelerinden sonuçların alınması ve yeni işlerin gönderilmesi.

İki tane paralel çalışan hough nesnesi varsa , anlaşılacağı gibi her seferinde resmin iki parçası paralel olarak işlenmektedir. Buda yaklaşık iki kat hız kazancı demektir.

Daha öncede anlatıldığı gibi alt resimler bir biriyle örtüşecek şekilde oluşturulmaktadır. Örtüşme problemi Şekil 12'de gösterilmiştir. Arada kalan örtüşen kısımlar ayrı ayrı sunuculardaiki kere işlenirler. Bu da gereksiz yere fazladan veri transferi demektir. Karmaşıklık hesabında bunun etkisi incelenecektir.



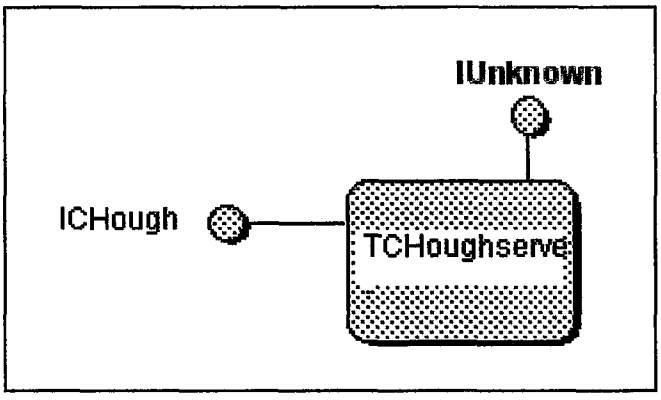
Şekil 8.12 Örtüşme problemi.

8.4.2 İstemci ve Sunucu Nesnelerinin Programlanması

Bu bölümde hough nesnelerinin nasıl programlandığı ve sorunların nasıl çözüldüğü incelenmiştir. Öncelikle paralel çalışan hough sunucularının yapısı incelenecektir.

8.4.3 Hough Sunucu Nesnesi

Hough sunucu nesnesi bir DCOM nesnesi olup ICHough adında bir arayüze sahiptir. İstemci nesnesi sunucu ile iletişimini bu arayüz üzerinden yapmaktadır. Bu nesnenin tanımı ve arayüz'ün içerdiği fonksiyonlar aşağıda verilmiştir.



Şekil 8.13 Hough sunucu nesnesi ve arayüzleri.

8.4.3.1 ICHough Arayüzünün Tanımı

Aşağıda ICHough arayüzünün Delphide yapılan TLB tanımı bulunmaktadır. Görüldüğü gibi bu arayüz bir dizi fonksiyonu içermektedir. Bu arayüzü gerçekleyen sunucu nesneleri de hough dönüşümünü yapmaktadırlar. Başlıca fonksiyonlar:

- FindCircles : verilen bir alt resimdeki çemberleri bulur,
- Finished : sunucunun işinin bitip bitmediğini belirtir.
- GetResult : sunucudan sonuçları almak için kullanılır.
- Initialize : sunucu nesnelerinin başlatılması için kullanılır.

Bu fonksiyonları biraz sonra daha ayrıntılı bir şekilde incelenecektir.

```
ICHough = interface(IDispatch)
  ['{BAB9B1C9-6982-11D2-9B7E-00609419D8B0}']
  procedure FindCircles(Resim: OleVariant; Dx, Dy, R: Integer); safecall;
  function Finished: Integer; safecall;
  function GetResult(var S: OleVariant; var N, OpTime: Integer): Integer; safecall;
  function Get_Durum: Integer; safecall;
  procedure Set_Durum(Value: Integer); safecall;
  procedure Initialize; safecall;
  property Durum: Integer read Get_Durum write Set_Durum;
end;
```

8.4.3.2 TCHoughServer Sunucusunun Tanımı

Hough sunucu nesnesi aşağıdaki gibi tanımlanmıştır. Artık bu aşamada arayüz fonksiyonlarının gerçeklenmesi gerekmektedir. Daha öncede anlatıldığı gibi arayüz tanımı sadece fonksiyonları ve parametrelerini listeliyordu fakat fonksiyonlarla ilgili bir gerçekleme yapılmıyordu. Artık sunucu nesnesini tanımlarken bu fonksiyonlarında gerçeklemesi yapılır. Şimdi bu fonksiyonları incelenecektir.

```
TCHoughServer = class(TAutoObject, ICHough)
 private
   SResim  :OleVariant; //Aranacak resim.
   SDX     :Integer;   // Dx boyutu.
   SDY     :Integer;   // Dy boyutu.
   SR      :Integer;   // Aranacak maximum circle yaricapi.
   Durum   :Integer;   //Serverin durumunu dondurur.
   Sonuc   :OleVariant; //Sonuclarin tutuldugu dizi.
   SN      :Integer;   //Sonuctaki circle sayisi.
   B,S     :TDateTime; //Islem baslangici ve bitisi.
   Sure    :Integer; //Toplam islem zamani.
  ThHough  :HoughThread;
  procedure ThreadFinished(Sender:TObject);
 protected
  procedure FindCircles(Resim: OleVariant; Dx, Dy, R: Integer); safecall;
  function Finished: Integer; safecall;
  function GetResult(var S: OleVariant; var N, OpTime: Integer): Integer;
   safecall;
  function Get_Durum: Integer; safecall;
  procedure Set_Durum(Value: Integer); safecall;
  procedure Initialize; safecall;
  constructor Create;
  destructor Destroy;
  function _AddRef:Integer;stdcall;  //override
  function _Release:Integer;stdcall; //override
 end;
```

FindCircles() : Bu fonksiyon parametre olarak verilen Resim alt resminde (DX x DY boyutunda) R büyüklüğündeki tüm çemberleri bulur. İlk olarak HoughThread nesnesi yaratılıyor ve gerekli işlemleri yapması sağlanıyor. Sunucu nesnesi bu işlemi yaptıktan sonra hemen istemciye geri dönmektedir. Yani henüz verilen iş bitmeden istemciye geri dönülmektedir. Zaten işlemlerin paralel çalışması bu sayede olmaktadır. İstemci diğer sunucularla uğraşırken bu sunucu kendine verilen işi yapmaktadır.

Daha sonra istemci bu sunucuyu devamlı olarak kontrol ederek işinin bitip bitmediğine bakar. Bu fonksiyon kodu aşağıda verilmiştir.

```
procedure TCHoughServer.FindCircles(Resim: OleVariant; Dx, Dy, R: Integer);
begin
 B:=Now;
 SResim:=Resim;
 Durum:=2;      //Server busy.
 ServerForm.Status.Lines.Add('Yeni bir iş geldi : Sunucu meşgul...');
 ThHough:=HoughThread.Create(SResim,DX,DY,R,Sonuc,SN);
 ThHough.FreeOnTerminate:=true;
 ThHough.OnTerminate:=ThreadFinished;
end;
```

Finished() : Bu fonksiyon sunucunun işini bitirip bitirmediğini kontrol için kullanılır. Sadece sunucunun Durum değişkenini döndürür. Bu değişken sunucunun meşgul yada boş olduğunu belirtir. Durum değişkenin alabileceği değerler:

0: Sunucu boş.
1: Sunucunun işlemi bitmiştir. GetResult çağrılmalı.
2: Sunucu şu anda meşgul

```
function TCHoughServer.Finished: Integer;
begin
 Finished:=Durum;
end;
```

GetResult() : Sunucudan cevapları almak için kullanılır. Bu fonksiyon Finished fonksiyonundan gelen cevaba göre çağrılır. Eğer Finished fonksiyonundan 1 gelmişse sunucunun işi bitmiştir ve artık sonuçlar alınabilir.

S değişkeni içinde bulunan N tane çemberin koordinatları gelir. OleVariant değişkenleri herhangi bir tipte değişken tutabilir. Bu değişken bir dizi olabilir burada da integer sayı dizisi tutmaktadır. Bu dizideki her ikili sayı bir çemberin koordinatını vermektedir.

Bu fonksiyonla ayrıca sunucunun işlem zamanı da döndürülmektedir. Bu zaman salise cinsinden olup timer ayarlarında kullanılmaktadır. Daha önce anlatıldığı gibi bu timer meşgul bekleme sorununu ortadan kaldırmak için kullanılmıştır. Bu zaman ayrıca işlem sonunda sunucu istatistik bilgilerini çıkarmak için de kullanılmaktadır.

```
function TCHoughServer.GetResult(var S: OleVariant; var N, OpTime: Integer): Integer;
begin
   ServerForm.Status.Refresh;
   S    :=Sonuc;      //Elde edilen sonuclari geri gonder
   ServerForm.Status.Refresh;
   N    :=SN;         //Circle sayisi.
   ServerForm.Status.Refresh;
   OpTime:=Sure; //Toplam islem suresi.
   Durum :=0;
end;
```

Initialize() : Bu fonksiyon sunucu nesneleri kullanılmadan önce bir kere çalıştırılmalıdır. Sunucu nesnesinin başlangıç değerlerini ayarlamasını ve gerekli hafızayı ayırmasını sağlar.

```
procedure TCHoughServer.Initialize;
begin
  Durum:=0; //Server bosta.
  Sonuc:=VarArrayCreate([0,MAX_CIRCLES*3], varInteger);
  ServerForm.Status.Lines.Add('Yeni bir sunucu nesnesi başlatıldı... iş bekleniyor...');
end;
```

Sunucularda çalıştırılan hough dönüşüm işlemi HoughServerThread nesnesi içinde kullanılan HoughClass nesnesi tarafından yapılır. Sunucu nesnesini oluşturmak için yazılan modüller şunlardır:

HoughServer : Bu modülde TCHoughServer nesnesinin tanımı ve gerçeklemesi vardır. Yukarıda anlatılan fonksiyonlar bu modülde tanımlanmıştır.

HoughServerForm : Sunucu nesnesinin ekran modülüdür. Bu modülde sunucu nesnesi çalışınca ekrana gelecek olan görüntüler ve işlemler tanımlanmıştır.

HoughServerP_TLB : ICHough arayüzünün IDL tanımlaması yer almaktadır. Bu modül sadece gerekli arayüzü tanımlar. Hem sunucu hem de istemci programları bu modüle ihtiyaç duyarlar. Bu dosya Delphi içinde bir menü vasıtasıyla otomatik olarak oluşturulur. El ile değiştirilirse Delphi tekrar eski tanımları geri yükler.

HoughServerThread : Bu modül sunucu nesnesinde çalışan thread nesnesinin tanımlandığı ve gerçeklendiği modüldür. Burada verilen bir resimdeki çemberler bulunur.

HoughClass : Hough dönüşüm işlemini yapan nesnedir. Bu nesne HoughServerThread tarafından kullanılır.

8.4.4 Hough İstemci Programı

Hough istemci programı bir DCOM nesnesi olmayıp sunucu nesnesini kullanan bir istemci programdır. Genel paralelleştirme algoritmasının gerçeklendiği bir programdır. İstemci gerekli alt işlemleri oluşturup , sunucu nesnelerine dağıtmakla görevlidir. Şimdi bu istemcinin ne gibi işleri nasıl yaptığı incelenecektir.

8.4.4.1 Sunucu Nesnelerinin Dinamik Olarak Oluşturulması

İstemci ilk olarak sunucu nesnelerini oluşturur. Bunun nedeni resmi alt işlemlere ayırırken kullanılacak olan bölümleme büyüklüğünün hesaplanmasıdır. Bu bölümleme büyüklüğü (d) paralel çalışan sunucu nesnesi sayısı ile orantılı olduğu için sunucu sayısının bilinmesi gerekmektedir. Sunucu nesnelerini oluşturan fonksiyon aşağıda verilmiştir.

Paralel sunucu nesnelerinin oluşturulacağı bilgisayar isimleri servers.txt adında bir dosyadan alınır. Buradaki her bilgisayar için

```
Pkunk:=CreateRemoteComObject(str,Class_CHoughServer);
```

Fonksiyonu çalıştırılır. Buradan gelen Pkunk işaretçisine bakılarak bu bilgisayarda Hough sunucu nesnesi olup olmadığı belirlenir. Eğer yoksa hata verilir varsa bu işaretçi üzerinde (IUnknown arayüzüdür bu işaretçi)

```
Pkunk.QueryInterface(IID_ICHough,ServerObj[ObjN]);
```

Fonksiyonu çalıştırılarak bu sunucu nesnesinin kullanmak istenilen ICHough arayüzünü destekleyip desteklemediği sorulur eğer desteklemiyorsa hata verilir , destekliyorsa bu arayüze işaretçi ServerObj[] içine arttırılır.

Bu işlemler tamamlanınca ServerObj[] dizisinde ObjN tane sunucu nesnesi oluşturulmuştur. Artık bu nesneler kullanılarak paralel işlemler yapılabilir.

```
procedure TClientForm.btnSearcServersClick(Sender: TObject);
var
 f:TextFile;
 str:String;
begin
 AssignFile(f,'Servers.txt');
 Reset(f);
 ObjN:=1;
 while (not eof(f)) do
  begin
   ReadLn(f,str);
   StatusForm.Status.Lines.Add('Testing :'+str);
   StatusForm.Status.Lines.Add('  Looking for CHoughServer object...');
   try
    Pkunk:=CreateRemoteComObject(str,Class_CHoughServer);
   except
    StatusForm.Status.Lines.Add(' Server error CHoughServer not found...!');
    Continue;
   end;
   StatusForm.Status.Lines.Add(' ServerObject found. Testing interface...');
   Pkunk.QueryInterface(IID_ICHough,ServerObj[ObjN]);
   if (ServerObj[ObjN]=nil) then
    begin
     //ShowMessage('Interface Desteklenmiyor...');
     Pkunk._Release;
     StatusForm.Status.Lines.Add(' Interface ICHough not supported...');
     Continue;
    end;
   ServerObj[ObjN].Initialize;
   ServerName[ObjN]:=str;
   ObjN:=ObjN+1;
   StatusForm.Status.Lines.Add(' Interface supported...server initiated.');
  end;
  CloseFile(f);
  ObjN:=ObjN-1;
  StatusForm.Status.Lines.Add(IntToStr(ObjN)+' Servers active.');
end;
```

8.4.4.2 Alt İşlemlerin Oluşturulması

Sunucu nesneleri oluşturulduktan sonra resim alt resimlere ayrılır. Bu işlemi ResimParçala fonksiyonu yapar. dx,dy boyutundaki resmi parçalar. Burada r değeri aranan çemberin yarıçapıdır ve aynı zamanda örtüşme büyüklüğü olarak kullanılır.

Resim parçalara ayrıldıktan sonra ParResim[] adındaki bir diziye oluşan alt resimlerin asıl resimdeki koordinatları girilir. Daha sonra bu oluşan alt resim numaraları (ParResim dizisindeki indis numaraları) bir yığına atılır. Ve buradan artık işler paralel sunuculara dağıtılır.

```
procedure TClientForm.ResimParcala(dx,dy,r:Integer);
 var
  si,sj,dr,i,j,mi,mj,art:Integer;
  mibitti,mjbitti :Bool;
 begin
  dr :=Carpim*r;
  art:=(Carpim-1)*r;
  PN:=0;
  mi:=0; // asagi dogru.
  mibitti:=false;
  while (mi<Dy)and (not mibitti)  do
   begin
    mj:=0; // saga dogru.
    mjbitti:=false;
    while (mj<Dx) and (not mjbitti) do
     begin
      inc(PN);
      ParResim[PN].Basladi:=false;
      ParResim[PN].Bitti  :=false;
      ParResim[PN].mi     :=mi;
      ParResim[PN].mj     :=mj;
      if ((mi+dr)<dy) then
        ParResim[PN].di:=dr
      else
        begin
          ParResim[PN].di:=Dy-mi;
          mibitti :=true;
        end;
      if (dy-(mi+dr) < r) then //efer
        mibitti:=true;
      if ((mj+dr)<dx) then
        ParResim[PN].dj:=dr
      else
       begin
         ParResim[PN].dj:=Dx-mj;
         mjbitti:=true;
       end;
      if (dx-(mj+dr) < r) then //efer
        mjbitti:=true;
      mj:=mj+art;
     end;
    mi:=mi+art;
   end;
 Image.Canvas.Pen.Color  :=clGray;
 for i:=1 to PN do
  begin
    Image.Canvas.MoveTo(ParResim[i].mj,ParResim[i].mi);
    Image.Canvas.LineTo(ParResim[i].mj,ParResim[i].mi+ParResim[i].di);
    Image.Canvas.LineTo(ParResim[i].mj+ParResim[i].dj,ParResim[i].mi+ParResim[i].di);
    Image.Canvas.LineTo(ParResim[i].mj+ParResim[i].dj,ParResim[i].mi);
    Image.Canvas.LineTo(ParResim[i].mj,ParResim[i].mi);
  end;
end;
```

8.4.4.3 Genel Paralelleştirme İşlemi

İstemci sunucu nesnelerini oluşturup alt işlemleri de oluşturduktan sonra bu işlemleri paralel olarak yapılmak üzere sunucu nesnelerine dağıtır. Bu işlemi yapan fonksiyonun kaynak kodu aşağıda verilmiştir.

```
procedure TClientForm.btnHoughClick(Sender: TObject);
begin
 for i:=1 to ObjN do
   begin
     ServerSure[i]:=0;
     ServerSayi[i]:=0;
   end;
 YariCap:=StrToInt(edtYaricap.Text);
 Carpim:=StrToInt(edtCarpim.Text);
 ResimParcala(Dx,Dy,YariCap); //Resmi segmentlere ayir.
  SegmentResim:=VarArrayCreate([0,Carpim*YariCap+1,0,Carpim*Yaricap+1],varByte); //Segmente edilmis resmi
saklayan dizi.
 StackInit;
 for i:=1 to PN do
   Push(PN-i+1);
 IslemZaman:=0; //Server da yapilan islem suresi optime.
 NetZaman  :=0; //Network transfer zamani
 TopZaman  :=0; //Clienttaki toplam sure.
 StatusForm.Status.Lines.Add('Image size : '+IntToStr(Dx)+' x '+IntToStr(Dy));
 StatusForm.Status.Lines.Add('Segment count : '+IntToStr(PN));
 StatusForm.Status.Lines.Add('Operation started:');
 B:=Now;       //Islem basi. Top zaman hesabi.
 done:=False;
 CN:=0;
 pi:=1;
 //Timer kontrolu alacak ve islemleri yapacaktir.
 //Islem bitmedigi surece
 while (not done) do
  begin
    done:=True;
    //Tum serverlari tara. Eger isi biten varsa sonuclari al ve yeni segment ver.
    for i:=1 to ObjN do
     begin
       try
         if (ServerObj[i]=nil) then
           continue;
       except
         continue;
       end;
       try
         Durum:=ServerObj[i].Finished;
       except
         StatusForm.Status.Lines.Add('Server    disconnected    :'+ServerName[i]+'    '+IntToStr(ServerSegment[i])+'.
segment tekrar kuyruga atildi.');
         Push(ServerSegment[i]);
         ServerObj[i]:=nil;
         continue;
       end;
       if (durum=1) then //Eger serverin isi bitmisse.
         begin
           ParResim[i].Bitti:=true;
           NB:=Now;
           ServerObj[i].GetResult(Sonuc,SN,Sure); //Bu serverin isi bitmistir.
           ServerSure[i]:=ServerSure[i]+Sure;    //Serverin toplam islem zamani.
```

```
      NS:=Now;
      NetZaman:=NetZaman+ZamanHesapla(NB,NS);
      IslemZaman:=IslemZaman+Sure;
      k:=1;
      while (k<SN) do
       begin
         inc(CN);
         Circles[CN].A:=Sonuc[k]+ParResim[ServerSegment[i]].mi;  //i asagi
         Circles[CN].B:=Sonuc[k+1]+ParResim[ServerSegment[i]].mj;//j saga
         Circles[CN].r:=Sonuc[k+2]; //r
         k:=k+3;
       end;
      //Eger kalan bir segment varsa bu servera islemesi icin yeni bir segment ver.
      if (not StackEmpty) then
       begin
         pi:=Pop;
         SegmentOlustur(pi);
         NB:=Now;
         ServerObj[i].FindCircles(SegmentResim,ParResim[pi].Dj,ParResim[pi].Di,YariCap);
         NS:=Now;
         NetZaman:=NetZaman+ZamanHesapla(NB,NS);
         StatusForm.Status.Lines.Add(IntToStr(pi)+'. segment given to : '+ServerName[i]);
         ServerSayi[i]:=ServerSayi[i]+1;     //islenen segment sayisi.
         ServerSegment[i]:=pi;
         ParResim[i].Basladi:=true; //Segment islemi basladi.
         pi:=pi+1;
         done:=False;
       end;
     end
    else if (Durum=0) then //Eger serverin bossa.
     begin
       if (not StackEmpty) then //Ve islenecek segment kaldiysa.
         begin
           pi:=Pop;
           SegmentOlustur(pi);
           NB:=Now;
           ServerObj[i].FindCircles(SegmentResim,ParResim[pi].Dj,ParResim[pi].Di,YariCap);
           NS:=Now;
           NetZaman:=NetZaman+ZamanHesapla(NB,NS);
           ServerSayi[i]:=ServerSayi[i]+1;     //islenen segment sayisi.
           StatusForm.Status.Lines.Add(IntToStr(pi)+'. segment given to : '+ServerName[i]);
           ServerSegment[i]:=pi;
           ParResim[pi].Basladi:=true; //Segment islemi basladi.
           pi:=pi+1;
           done:=False;
         end;
       end
     else
      if (Durum=2) then //Eger serverin busy ise.
        done:=false;
   end;
 end;
```

While döngüsünden çıkıldığında Circles[CN] dizisinde CN tane çember vardır.

8.4.5 Karmaşıklık Hesabı

Şimdi normal ve paralel Hough transform işlemelerinin karmaşıklık hesapları ve ne kadar kazanç elde edildiği incelenecektir.

r : aranan maksimum çember yarıçapı.
d : bölünen parçaların boyutu.
m : parçaların örtüşme aralığı.
N,M : resmin boyutları.
k : inceltme işleminde geçiş sayısı.
n : paralel çalışan hough nesne sayısı.

Normal Hough Dönüşümü karmaşıklığı:

$$(N M) + k(N M) + r(N M) = (k + r + 1) N M \Rightarrow r N M \qquad (8.7)$$

Kenar bulma ve inceltme işlemlerinin hough dönüşümü yanında önemsiz olduğu düşünülürse formüldeki (k + 1) çarpımı ihmal edilebilir. NM çarpanı resimdeki noktaların sayısını verir, bu noktaları 1-r arasındaki tüm çemberleri bulmak için r kere işlendiği için normal karmaşıklık rNM olur. Yani her nokta r kere işleme tabi tutulur.

Paralel Hough Dönüşümü karmaşıklığı:

Paralel hough dönüşümü karmaşıklığı hesabında iki durum söz konusudur. İlk olarak alt işlem yani alt resim sayısı kadar paralel çalışan sunucu nesnesi olduğu varsayılsın.

Optimum karmaşıklık : Bölümleme aralığı ,aranan çemberin yarıçapının p katı olarak alınırsa yani $d = p r$,

$$(d * d) + k(d * d) + r(d * d) = (k + r + 1) d^2 \Rightarrow r^2 p^2 r^2 \qquad (8.8)$$

Burada bölünen parça sayısı kadar paralel çalışan hough dönüşümü nesnesi olduğu varsayılmıştır. Görüldüğü gibi teorik olarak paralel karmaşıklık NM resmin boyutuna bağlı değildir. Sadece alt resimlerin boyutuna bağlıdır ve bu boyut da istenildiği gibi ayarlanabilir.

Gerçek karmaşıklık : Ancak pratikteki gerçek karmaşıklık fonksiyonu bu şekilde değildir. Çünkü burada sonsuz sayıda sunucu nesnesi olduğunu varsayıldı. Eğer n tane paralel çalışan nesne varsa gerçek karmaşıklık aşağıdaki gibi olur.

$$r \frac{1}{n} \frac{NM}{(d-m)^2} d^2 = r \frac{1}{n} \frac{NM}{(pr-m)^2} p^2 r^2 \Rightarrow r \frac{1}{n} \frac{NM}{(p-1)^2} p^2 \qquad (8.9)$$

$$n \leq \frac{NM}{(pr-m)^2} \Rightarrow p = \frac{1}{r} \sqrt{\frac{NM}{n}} + 1 \quad , \quad m \geq r \Rightarrow m = r$$

Bu fonksiyonda r çarpanı her noktanın r kere işlendiğini gösterir. 1/n çarpanı n tane sunucu nesnesi tarafından elde edilen paralelliği gösterir. NM/(d-m)2 çarpanı alt işlem sayısını verir, d*d çarpanı da alt resimdeki nokta sayısını verir.

Burada örtüşme aralığı m>=r alınmalı, karmaşıklık fonksiyonun mimimum olması için örtüşme aralığı minimum bir değer alması gerekir, bu değerde görüldüğü gibi m=r dır.

Bu karmaşıklık fonksiyonundan bölümleme boyutu bulunabilir. Sunucu nesne sayısı hiçbir zaman alt resim sayısından daha fazla olamaz, dolayısıyla n<=NM/(pr-m)2 bulunur ve buradan p hesaplanabilir. Ve p nin bu sınırda aldığı maksimum değer bölümlemenin optimum yapılacağı değerdir.

Görüldüğü gibi karmaşıklık paralel çalışan hough nesnelerinin sayısı ile ters orantılıdır. Eğer nesne sayısı 1 ise normal karmaşıklıktan daha fazla bir sonuç elde edilmektedir. Bu da bölünen parçaların örtüşmesinden gelen fazla hesaplamalardan kaynaklanmaktadır. Eğer nesne sayısı 2 ve daha büyükse karmaşıklık azalmaktadır.

Değişik boyutlardaki resmiler için karmaşıklık hesapları Tablo 8.1 de verilmiştir.

Tablo 8.1 Değişik boyuttaki resimler için karmaşıklık hesapları.

Boyut (N M)	Normal	Optimum	Gerçek
100 x 100	220.000	52.800	79.200
200 x 200	880.000	52.800	316.800
600 x 600	7.920.000	52.800	2.851.200

r =20 , m=20 , n = 4 , p=6, k=1

Tablodan da görüldüğü gibi optimum karmaşıklık değeri sabit kalmaktadır. Bölünen parçaların boyu sabit olduğundan bu değer de sabit kalmaktadır. Paralel çalışan nense sayısı 4 için incelenirse yaklaşık 3 kat bir kazanç olduğu görülmektedir. Gerçek karmaşıklık değeri hiçbir zaman optimum değerden daha küçük olamaz.

8.4.6 Sonuçlar

Paralel hough dönüşümü 600*600 boyutunda bir resim üzerinde test edildi. Aranan maksimum çember yarıçapı r=25 olarak alındı. Yani resimdeki yarı çapları 1-25 arasında olan tüm çemberler aranmıştır.

Değişik sunucu nesne sayısı ve değişik bölümleme büyüklükleri için test yapılmış ve toplam işlem süresi ve ağ transfer süreleri ölçülmüştür.

Bu resmin sıralı olarak işlem (yani hiç paralellik yokken) süresi 107 saniye olarak ölçülmüştür. Test sonuçları aşağıdaki Tablo 8.2'de verilmiştir.

İlk kolonda sunucu nesne sayısı verilmiştir, 2. kolonda (p) bölümleme büyüklüğü verilmiştir. Tablonun grafik olarak görünümü Ek A'da verilmiştir. Test programından örnek ekran görüntüleri EK C'de verilmiştir.

Tablo 8.2 Paralel hough dönüşümü test sonuçları.

Sunucu Nesnesi	p	Toplam Süre(sn)	İşlem Süresi (sn)	Ağ Transfer Süresi (sn)
2	4	75	64	11
	6	59	56	3
	8	49	48	1
	12	54	53	1
4	4	35	29	6
	6	27	26	1
	8	28	27	1
	12	31	30	1
8	4	19	11	8
	6	15	13	2
	8	16	15	1
	12	37	36	1
12	4	14	8	6
	6	**13**	**12**	**1**
	8	15	14	1
	12	44	43	1

Tablodan da görüldüğü gibi n=12 nesne ve d=6* ile en iyi sonuç 13 saniye olarak saptanmıştır. n=12 için d nin teorik optimum değeri d=7*r olarak hesaplanır. Fakat pratikte bu optimum değer geçerli değildir. Bunun sebebi nesnelerin çalıştığı bilgisayarların hepsinin aynı hızda olmamasından dolayıdır. Toplam işlem zamanı sistemde çalışan en yavaş bilgisayarın hızına bağlıdır, bu bilgisayar işini bitirmeden sonuçlar elde edilemez.

Buradan çıkarılan diğer bir sonuçta , p'nin çok küçük değerleri için ağ transfer süresinin çok büyümesidir. Daha öncede anlatıldığı gibi bu örtüşme probleminden dolayı olmaktadır. Eğer p çok küçükse çok fazla örtüşme ve dolayısıyla çok fazla bilgi transferi vardır.

Aslında p değeri çok büyüyünce de işlem zamanı yavaşlamaktadır. Bunun nedeni ise sistemdeki bilgisayarların aynı hızda çalışmamasından dolayıdır. Yani eğer p büyürse işlenecek alt resimlerin boyu da büyür ve dolayısıyla bilgisayarlar arasındaki hız farkı da artar. En yavaş bilgisayarın işinin bitmesi beklendiği için de işlem yavaşlar.

Sonuç olarak 12 tane bilgisayarlı bir sistemde her bilgisayarda 1 sunucu nesnesi çalıştırılarak gerçekleştirilen paralel hough işlemi süresi 107 saniyeden 13 saniyeye kadar düşürülmüştür.

Test için kullanılan sunucu ve istemci bilgileri: Sunucu olarak Pentium 166 işlemci 32 MB bellek içeren Windows NT sistemi kullanılmıştır. İstemci olarak ise Pentium 166 işlemcili 32 MB bellek içeren Windows NT workstation kullanılmıştır. Ağ bağlantısı olarak 100 Mbit Ethernet seçilmiştir.

8.5 Parmak İzi Analizi

Görüntü işlemede yoğun işlem gerektiren bir diğer işlemde parmak izlerinin [10] karşılaştırma için temizlenmesi ve segmentasyon işlemidir. Parmak izi daha çok suçlu tespitinde kullanılır , olay yerinden alınan parmak izleri genellikle çok bozuk haldedirler, bu izler öncelikle bazı ön işlemlerden geçirilerek karşılaştırma yapmaya uygun hale getirilir. Bu işlemler de genellikle çok zaman alan işlemlerdir. Öncelikle parmak izinin tarihçesi [10] incelenecektir:

Parmak izleri ilk olarak bir tahta oymacısı olan Thomas Bewick tarafından 1700`lü yıllarda kullanılmıştır. Thomas kendi yaptığı eserlere kendi parmak izini çizerdi. Böylelikle parmak izlerinin kişileri ayırt etme özelliği yavaş yavaş anlaşılmaya başlandı. Fakat parmak izlerini standart olarak çeşitli tiplere ayıran ve sınıflandıran kişi John Evangelise Parkinie`dir. Herkesin kendine özgü bir parmak izi olduğunu ıspatlayan kişi ise Francis Galton`dur. Galton daha sonra kendi adıyla anılan parmak izlerine özgü 4 karakteristik çıkarmıştır. Eduart Henry parmak izlerini sınıflandırmak için bazı karakteristikler belirlemiştir. Parmak izlerini sınıflandırmak için kullanılan Henry karakteristikleri parmak izlerini birbirinden ayıran temel özellikler [11-12] degildir. Parmak izlerini asıl birbirinden ayırt eden karakteristikler Galton karakteristikleridir. Galton ve Henry karakteristikleri aşağıda verilmiştir.

Henry Karakteristikleri : Henry sadece parmak izlerini sınıflandırmak için kullanılır. Bu sistemde parmak izleri temel olarak 5 gruba ayrılır. Sola yatık, sağa yatık, spiral, basık kemer, yüksek kemer.

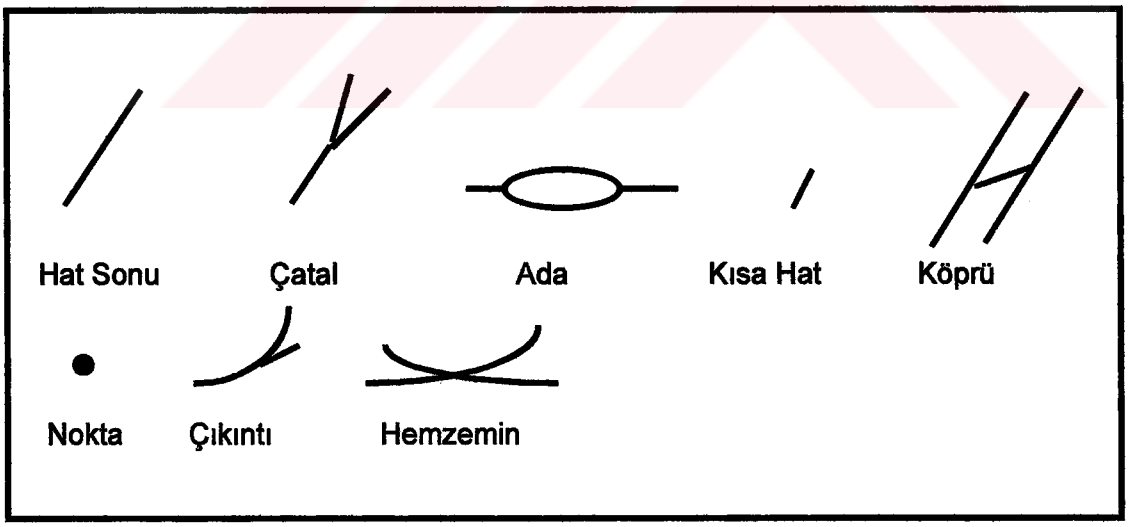
Galton karakteristikleri : Galton karakteristikleri temel olarak 4 tanedir bunlar. Hat sonu, çatal, ada ve kapalı hat.

Uzun yıllar süren araştırmalardan sonra Galton karakteristikleri geliştirilerek standart hale getirilmiştir. Artık günümüzde kullanılan karakteristikler nokta, hat sonu, ada, çatal, kısa hat, hemzemin, köprü, çıkıntı olarak belirlenmiştir.

Parmak izinin kişiyi tanımlamada kullanılmasını sağlayan başlıca özellikleri şunlardır:

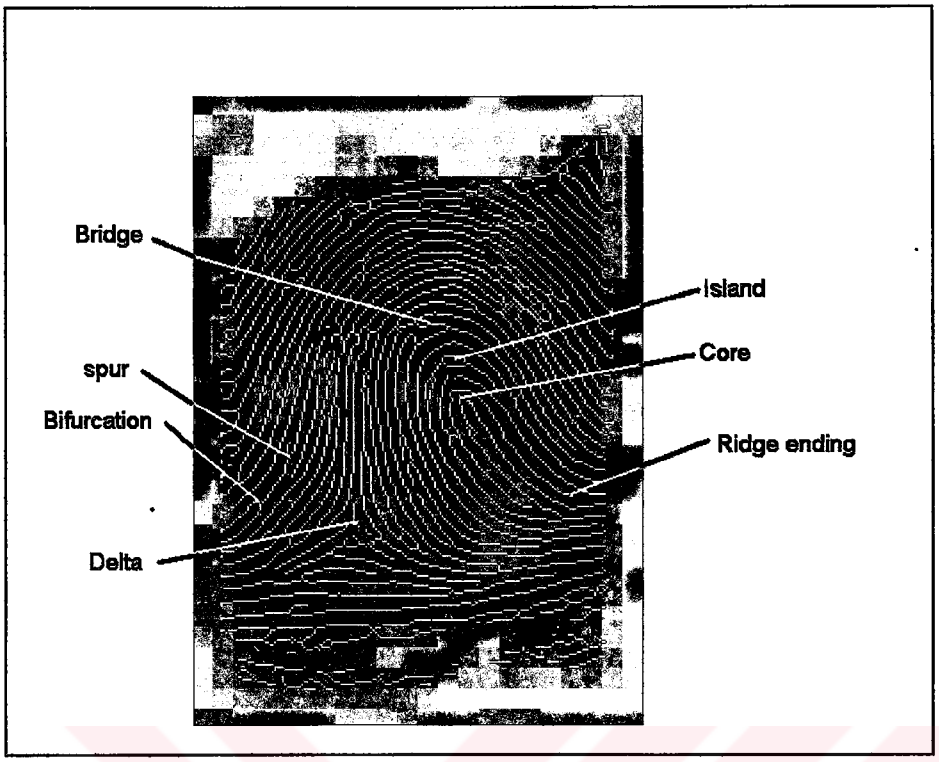
1. Değişmezlik özelliği : Parmak izindeki özellikler deride bir tahribat olmadığı sürece kişinin hayatı boyunca değişmez.
2. Tek olma özelliği : Parmak izindeki özelliklerin sayısı yerleşim kombinasyonu sınırsız olduğu için hiçbir parmak izi bir diğerinin aynısı olamaz.
3. Sınıflandırma : Bütün parmak izleri farklı olmasına rağmen , bazı benzer özellikler parmak izlerinin sınıflandırılmasına olanak sağlar.

Şekil 8.14'te Galton karakteristikleri görülmektedir. Burada görülen özelliklerden sadece hat sonu ve çatallar önemlidir. Yani iki parmak izindeki hat sonları ve çatal noktaları bulunup karşılaştırılarak tanımlama yapılabilir.



Şekil 8.14 Galton karakteristikleri.

Aşağıda örnek bir parmak izi ve üzerinde özel noktalar (galton karakteristikleri) gösterilmiştir.



Şekil 8.15 Örnek bir parmak izi ve özel noktalar.

Parmak izinin temizlenmesi ve bu özel noktaların bulunması için yapılan işlemler şunlardır:

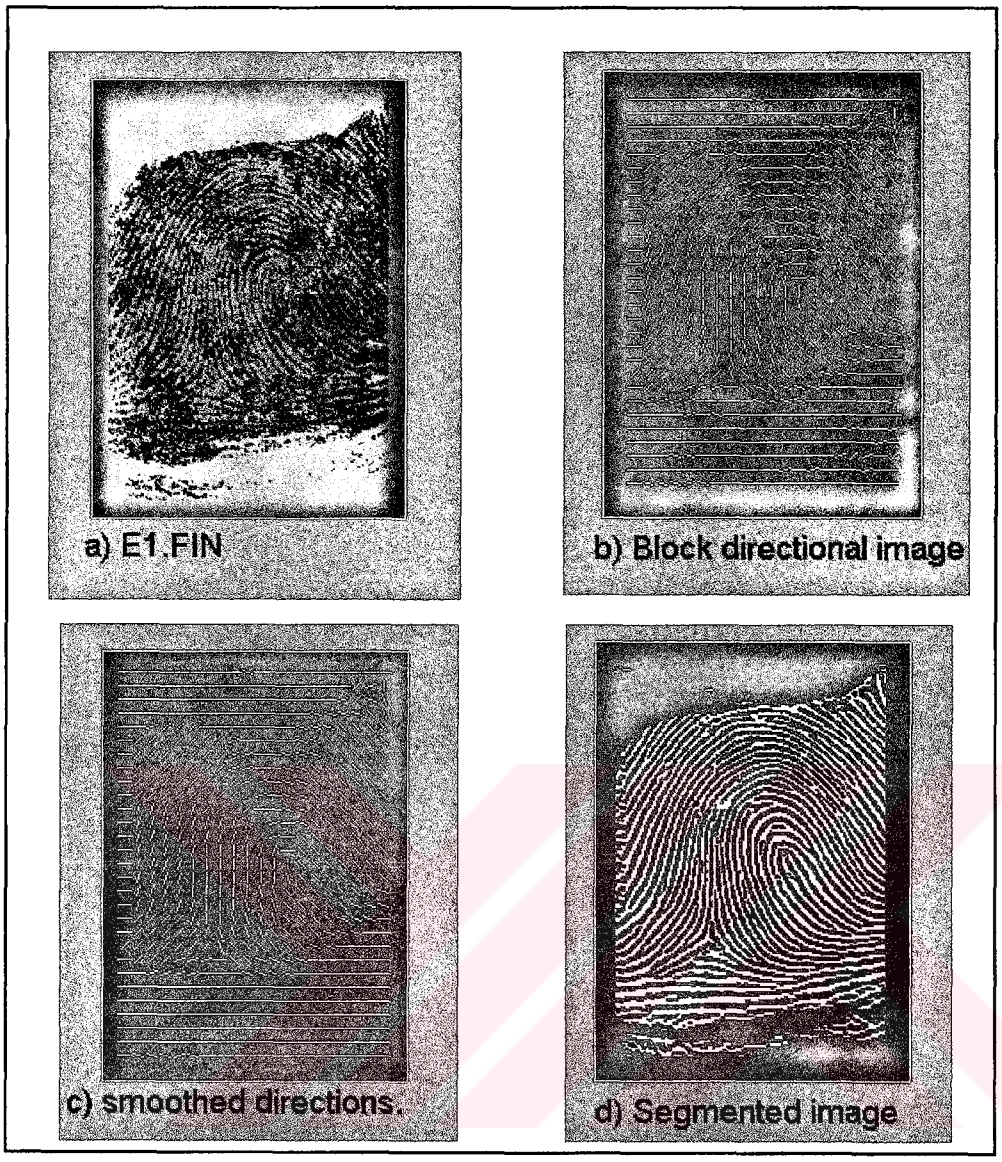
İlk olarak parmak izinin yönsel resmi (block directional image) [13-14] çıkartılır. Şekil 8.16'da bu gösterilmiştir. E1.FIN adındaki parmak izi görüldüğü gibi çok kötü bir durumdadır. Bu yönsel resim parmak izindeki hatların yönlerini gösteren bir matristir.

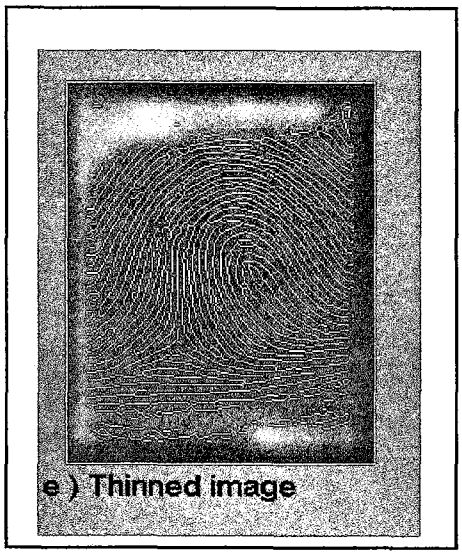
Daha sonra bu yön resmi komşu yönlere bakılarak düzgün hale getirilir. E1'in yönsel resmine bakarsak bazı noktalarda düzgün olmadığı sapmalar olduğu görülür. İşte bunlar komşu yönlerle aynı hale getirilir. (smoothed directional image [14])

Daha sonra bu yönsel resim kullanılarak her noktada uygulanması gereken filtre oluşturulur ve parmak izine uygulanır ve resmin segmentasyonu [14] yapılır. Parmak izi ikili [15] hale dönüştürülmüştür. (Segmented image)

En son olarak inceltme işlemi (thinning) [7,16] yapılır ve hatların tam olarak belli olduğu resim elde edilir. (thinned image)

Artık bu inceltilmiş görüntüden özel noktalar [17] (hat sonları ve çatallar) bulunur.





Şekil 8.16 Parmak izi temizleme aşamaları.

Projede yapılan parmak izindeki bu özel noktaların bulunma işlemidir. Daha önce de anlatıldığı gibi parmak izinin temizlenmesi çok zaman alan bir işlemdir. Dolayısıyla parmak izi görüntüsünü parçalayıp paralel bir şekilde çalışma yapılırsa daha hızlı bir sonuç alınır.

8.6 Algoritmanın Paralel İşleyişi

Parmak izi analizini paralel olarak gerçekleştirmek için hough dönüşümünde anlatılan genel yöntem aynen kullanılmıştır. İlk olarak resim bir biri üzerine örtüşen alt resimlere ayrılır ve bu alt resimler de paralel çalışan nesnelere dağıtılıp işlenir ve en sonunda sonuçlar birleştirilir.

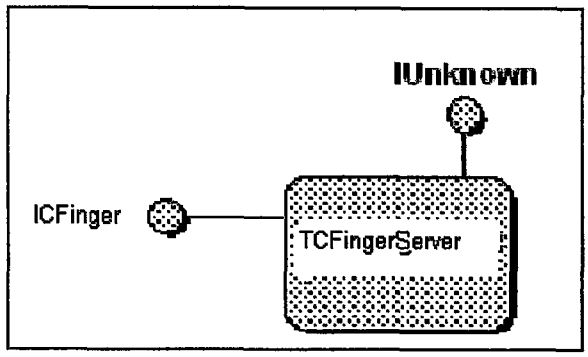
Fakat burada örtüşme büyüklüğü (m değeri) sabit bir değerdir. Hough dönüşümünden farklı olarak buradaki komşu noktaların birbiriyle fazla bir ilgisi yoktur. Yine de yönsel matrisin hesaplanması için uygulanacak olan filtre boyutu kadar bir örtüşme olmalıdır.

8.7 İstemci ve Sunucu Nesnelerinin Programlanması

Bu bölümde parmak izi analizini paralel olarak gerçeklemek için yazılan sunucu ve istemci nesnelerini incelenmiştir.

8.7.1 Finger Sunucu Nesnesi

Finger sunucu nesnesi (TCFingerServer) bir DCOM nesnesi olup aşağıda gösterilen yapıya sahiptir. Bu nesne ICFinger arayüzüne sahiptir ve kendisine verilen parmak izi resmindeki özel noktaları bulur. Sunucu ile istemci arasındaki işlemler bu arayüz sayesinde gerçekleşir.



Şekil 8.17 Finger sunucu nesnesi ve arayüzleri.

8.7.1.1 ICFinger Arayüzünün Tanımlanması

ICFinger arayüzü ile tanımlanan fonksiyonlar ve kullanım tanımları:

- FindMinutiae : Verilen bir parmak izi görüntüsündeki özel noktaları bulur.
- Finished : Sunucunun durumunu döndürür.
- GetResult : Sunucudan elde edilen sonuçların alınması işlemi.
- InitServer : Sunucunun başlatılma işlemi.

```
ICFingerServer = interface(IDispatch)
  ['{A0DBB579-A496-11D2-9BBC-00609419D8B0}']
  procedure FindMinutiae(Resim: OleVariant; Dx, Dy: Integer); safecall;
  function Finished: Integer; safecall;
  function GetResult(var SDCatal, SDEnd: OleVariant; var DN1, DN2, OpTime: Integer): Integer; safecall;
  function Get_Durum: Integer; safecall;
  procedure Set_Durum(Value: Integer); safecall;
  procedure InitServer; safecall;
  property Durum: Integer read Get_Durum write Set_Durum;
 end;
```

8.7.1.2 TCFingerServer Sunucu Nesnesinin Tanımı

TCFingerServerSunucu nesnesinin tanımı aşağıda verilmiştir. Sunucu ICHough arayüzüne sahiptir ve bu arayüze ait olan fonksiyonlar burada gerçeklenmiştir.

```
TCFingerServer = class(TAutoObject, ICFingerServer)
 private
   Durum     : Integer; //Serverin durumu.
   SResim    : OleVariant; // Islenecek resim.
   B,S       : TDateTime; //Islem baslangici ve bitisi.
   Sure      : Integer; //Toplam islem zamani.
   DCatal    : OleVariant; //Sonuclarin tutuldugu dizi.
   N1        : Integer;   //Sonuctaki circle sayisi.
   DEnd      : OleVariant; //Sonuclarin tutuldugu dizi.
   N2        : Integer;   //Sonuctaki circle sayisi.
   ThFinger  :FingerThread;
   d         :TextFile;
  procedure ThreadFinished(Sender:TObject);
 protected
  function Finished: Integer; safecall;
  function Get_Durum: Integer; safecall;
  function GetResult(var SDCatal, SDEnd: OleVariant; var DN1, DN2,
    OpTime: Integer): Integer; safecall;
  procedure FindMinutiae(Resim: OleVariant; Dx, Dy: Integer); safecall;
  procedure InitServer; safecall;
  procedure Set_Durum(Value: Integer); safecall;
  function _AddRef:Integer;stdcall;  //override
  function _Release:Integer;stdcall; //override
end;
```

Şimdi gerçeklenen arayüz fonksiyonları incelenecektir:

FindMinutiae() : OleVariant olarak verilen bir resmi (DX x DY boyutunda) temizleyerek özellikleri (minutiae) bulur. Bu fonksiyon asenkron olarak çalışmaktadır, yani istemci tarafından çağrıldıktan sonra bir thread çalıştırıp geri döner hesaplamaları yapar. İstemci iş bitiminde sunucudan sonuçları GetResult fonksiyonu ile alır.

```
procedure TCFingerServer.FindMinutiae(Resim: OleVariant; Dx, Dy: Integer);
begin
  B:=Now;
  SResim:=Resim;
  Durum:=2; // Server busy.
  ServerForm.Status.Lines.Add('Yeni bir is geldi : Sunucu calisiyor...');
  ServerForm.Status.Refresh;
  ServerForm.Refresh;
  ThFinger:=FingerThread.Create(SResim,Dx,Dy,DCatal,DEnd,N1,N2);
  ThFinger.FreeOnTerminate:=true;
  ThFinger.OnTerminate:=ThreadFinished;
end;
```

GetResult() : Sunucudan sonuçları almak için kullanılır. SDCatal, SDEnd değişkenleri içinde çatal noktaları ve hat sonu noktaları gelir. DN1, DN2 ise sırası ile bu dizilerin eleman sayısıdır. OpTime değişkeninde sunucunun işlem zamanı gelir bu zaman timer ayarlarında kullanılır.

```
function TCFingerServer.GetResult(var SDCatal, SDEnd: OleVariant; var DN1, DN2, OpTime: Integer): Integer;
begin
  ServerForm.Status.Refresh;
  SDCatal    :=DCatal; //Elde edilen sonuclari geri gonder
  ServerForm.Status.Refresh;
  SDend    :=DEnd;   //Circle sayisi.
  DN1:=N1;
  DN2:=N2;
  ServerForm.Status.Refresh;
  OpTime:=Sure; //Toplam islem suresi.
  Durum :=0;
end;
```

Finished() : Bu fonksiyon sunucunun işini bitirip bitirmediğini kontrol için kullanılır. Sadece sunucunun Durum değişkenini döndürür. Bu değişken sunucunun meşgul yada boş olduğunu belirtir. Durum değişkenin alabileceği değerler:

0: Sunucu boş.
1: Sunucunun işlemi bitmiştir. GetResult çağrılmalı.
2: Sunucu şu anda meşgul

```
function TCFingerServer.Finished: Integer;
begin
 Finished:=Durum;
end;
```

InitServer() : Bu fonksiyon sunucu nesneleri kullanılmadan önce bir kere çalıştırılmalıdır. Sunucu nesnesinin başlangıç değerlerini ayarlamasını ve gerekli hafızayı ayırmasını sağlar.

```
procedure TCFingerServer.InitServer;
begin
  Durum:=0; //Server bosta.
  DCatal:=VarArrayCreate([0,MAX_MINUTIAE*3], varInteger);
  DEnd  :=VarArrayCreate([0,MAX_MINUTIAE*3], varInteger);
  N1:=0; N2:=0;
  ServerForm.Status.Lines.Add('Yeni bir Sunucu örnegi yaratildi... yeni bir is bekleniyor...');
  ServerForm.Status.Refresh;
end;
```

Sunucularda çalıştırılan parmak izi analiz işlemi FingerServerThread nesnesi içinde kullanılan FingerClass nesnesi tarafından yapılır.

Sunucu nesnesini oluşturmak için yazılan modüller şunlardır:

FingerServer : Bu modülde TCFingerServer nesnesinin tanımı ve gerçeklemesi vardır. Yukarıda anlatılan fonksiyonlar bu modülde tanımlanmıştır.

FingerServerForm : Sunucu nesnesinin ekran modülüdür. Bu modülde sunucu nesnesi çalışınca ekrana gelecek olan görüntüler ve işlemler tanımlanmıştır.

FingerServerP_TLB : ICFinger arayüzünün IDL tanımlaması yer almaktadır. Bu modül sadece gerekli arayüzü tanımlar. Hem sunucu hem de istemci programları bu modüle ihtiyaç duyarlar. Bu dosya Delphi içinde bir menü vasıtasıyla otomatik olarak oluşturulur. El ile değiştirilirse Delphi tekrar eski tanımları geri yükler.

FingerServerThread : Bu modül sunucu nesnesinde çalışan thread nesnesinin tanımlandığı ve gerçeklendiği modüldür. Burada verilen bir resimdeki çemberler bulunur.

FingerClass : Parmak izi analiz işlemini yapan nesnedir. Bu nesne FingerServerThread tarafından kullanılır.

8.7.2 Finger İstemci Programı

Finger istemci programı bir DCOM nesnesi olmayıp sunucu nesnesini kullanan bir istemci programdır. Genel paralelleştirme algoritmasının gerçeklendiği bir programdır. İstemci gerekli alt işlemleri oluşturup , sunucu nesnelerine dağıtmakla görevlidir.

İstemci programdaki sunucu nesnelerin dinamik olarak oluşturulması, alt işlemlerin oluşturulması hough dönüşümü programı ile tamamen aynıdır.

8.7.3 Karmaşıklık Hesabı

Bu bölümde normal ve paralel parmak izi analizi işlemlerinin karmaşıklık hesapları ve ne kadar kazanç elde edildiği incelenmiştir.

fn : yön matrisinin hesabı için kullanılan filtrenin boyu.
fb : yön matrisini düzenlemek kullanılan filtrenin boyu.
fk : segmentasyon için kullanılan filtre boyu.
b : yön resmindeki blok büyüklüğü.
d : bölünen parçaların boyutu.
m : parçaların örtüşme aralığı.
N,M : resmin boyutları.
k : inceltme işleminde geçiş sayısı.
n : paralel çalışan finger nesne sayısı.

Normal İşlem karmaşıklığı:

$$fn\ N M + fb \frac{N M}{b b} + fk N M + k N M \qquad (8.10)$$

Bu fonksiyon sıralı olarak işlem karmaşıklığıdır. Fonksiyondaki 1. terim fn NM çarpanı, yönsel matrisi bulmak için yapılan işlemleri belirtir, 2. terim fb NM/bb çarpanı bulunan yön matrisinin düzeltilmesi için yapılan işlemleri, 3. terim fk NM çarpanı yön matrisi kullanılarak görüntünün segmentasyon işlemini belirtir son çarpan ise elde edilen ikili görüntünün inceltme işlemidir.

Paralel işlem karmaşıklığı : Paralel işlem karmaşıklığında iki durum söz konusudur. İlk olarak alt işlem yani alt resim sayısı kadar paralel çalışan sunucu nesnesi olduğunu düşünülsün.

Optimum karmaşıklık : Optimum karmaşıklık normal karmaşıklık ile aynıdır sadece NM yerine alt resmin boyutu olan d gelecektir.

fn (d*d) + fb (d*d) / (b*b) + fk (d*d) + k (d*d) (8.11)

Burada bölünen parça sayısı kadar paralel çalışan finger nesnesi olduğu varsayılmıştır. Görüldüğü gibi teorik olarak paralel karmaşıklık NM resmin boyutuna bağlı değildir. Sadece alt resimlerin boyutuna bağlıdır ve bu boyut istenildiği gibi seçilebilir.

Gerçek karmaşıklık : Ancak pratikteki gerçek karmaşıklık fonksiyonu bu şekilde değildir. Çünkü burada sonsuz sayıda sunucu nesnesi olduğunu varsayılmıştır. Eğer n tane paralel çalışan nesne varsa gerçek karmaşıklık aşağıdaki gibi olur.

$$\frac{1}{n}\frac{NM}{(d-m)^2}\left[fn\,d^2 + fb\,\frac{d^2}{b^2} + fk\,d^2 + k\,d^2\right] \qquad (8.12)$$

$$n \leqslant \frac{NM}{(d-m)^2} \Longrightarrow d = \sqrt{\frac{NM}{n}} + m \qquad m = \frac{fn}{2}$$

8.7.4 Sonuçlar

Paralel parmak izi işlemi 512*512 boyutunda bir resim üzerinde test edildi. Değişik sunucu nesne sayısı ve değişik bölümleme büyüklükleri için test yapılmış ve toplam işlem süresi ve ağ transfer süreleri ölçülmüştür.

Bu resmin sıralı olarak işlem (yani hiç paralellik yokken) süresi 62.47 saniye olarak ölçülmüştür. Test sonuçları Tablo 8.2'de verilmiştir. Tablonun grafiksel gösterimi EK B'de verilmiştir. Test programın örnek ekran çıktıları EK D'de verilmiştir.

İlk kolonda sunucu nesne sayısı verilmiştir, 2. kolonda (d) bölümleme büyüklüğü verilmiştir.

Tablodan da görüldüğü üzere normal olarak 62 saniyede yapılan işlem 12 paralel sunucu nesnesi kullanılarak 5.8 sn'ye indirilmiştir. Yaklaşık olarak 12 kat hız kazancı sağlanmıştır. Parmak izi analiz işleminde ağ transfer süresinin pek değişmediği görülür. Bunun nedeni bu algoritmada örtüşme probleminin olmamasından kaynaklanmaktadır. Hough dönüşüm işleminde her alt resim diğer resimlerle örtüşmekteydi bu yüzden bölümleme aralığı değiştikçe ağ transfer süresi de büyük

ölçüde değişiyordu. Fakat burada örtüşme büyüklüğü küçük ve sabit olduğundan bu süreye fazla etki etmemektedir.

Ancak bölümleme aralığı çok küçük alınırsa örtüşme probleminin etkisi görülebilir. Tabloda d'nin 50 olduğu satırlar incelenirse gerçek işlem zamanının ağ transfer süresi ile hemen hemen aynı olduğu görülür. Yani ağ transfer süresi birden bire artmıştır bunun nedeni çok fazla örtüşme olmasındandır.

Tablo 8.3 Paralel parmak izi analizi test sonuçları

Sunucu Nesne	d	Toplam süre(sn)	İşlem zamanı (sn)	Ağ Transfer Süresi(sn)
2	50	35.00	32.00	3.000
	100	32.70	31.50	1.2 00
	150	31.90	31.20	0.651
	200	32.12	31.13	0.622
	250	31.90	31.18	0.700
4	50	17.57	14.44	3.127
	100	15.00	14.80	1.510
	150	15.89	15.25	0.641
	200	17.93	17.37	0.562
	250	15.81	15.22	0.791
8	50	9.23	5.86	3.156
	100	8.40	7.25	1.150
	150	9.38	8.74	0.641
	200	9.63	8.53	0.531
	250	14.63	13.82	0.802
12	50	6.63	3.50	3.126
	100	**5.85**	**4.70**	**1.900**
	150	6.85	6.15	0.700
	200	-	-	-
	250	-	-	-

9. SONUÇLAR VE TARTIŞMA

Yapılan çalışmada, hesaplama modelleri içinde en fazla paralellik içeren MIMD modeli yazılımla gerçekleştirilmiştir. MIMD modelinin diğer modellere göre avantajı aynı anda birden fazla veri kümesi üzerinde farklı farklı komutları işletmesidir. Fakat donanım olarak tasarlanması ve gerçeklemesi en zor olan modeldir. Özel donanım, işletim sistemi ve derleyici gerektirir. Bu problemlerden kurtulmak ve maliyeti çok düşük bir paralel sistem oluşturmak için, MIMD modeli yazılımla gerçekleştirilmiştir. Daha sonra bu model üzerinde algoritmaların paralel çalıştırılabilmesi için genel bir sistem tasarlanmıştır. Tasarlanan bu sistem görüntü işleme algoritmalarına başarı ile uygulanmıştır.

Geliştirilen paralel sistemde tek bir istemci ve birden fazla sunucu yaklaşımı kullanılmıştır. Bu sistemde bir istemci diğer sunucuları kontrol ederek işlerin yapılmasını sağlar. Buradaki sunucu programları farklı makinelerde çalışan DCOM nesneleridir. Her bir nesne, istemci programı tarafından kendine verilen işi en kısa zamanda bitirir ve sonuçları yine istemci programına döndürür.

İstemci programının elindeki işi sunuculara dağıtıp paralel olarak işlemesi için genel paralelleştirme algoritması tasarlanmıştır. Bu algoritma bir problemin alt işlemlere ayrılması ve bu alt işlemlerin kuyruğa atılması işlemlerini tanımlar. Buradaki en önemli nokta problemin birbirinden bağımsız alt problemlere ayrılması işlemidir. Bu işlemin yapılması için paralel çalışan sunucu nesnelerinin sayısının bilinmesi gereklidir. İstemci elindeki algoritmayı sunuculara dağıtıp paralel olarak gerçekleştirmek için genel paralelleştirme algoritmasını kullanır.

İstemci ve sunucu arasında asenkron (eşzamansız) iletişim şekli kullanılmıştır. Yani istemci sunuculara işleri dağıtıp bir while döngüsü içinde meşgul bekleme yapmaktadır. Döngüde devamlı olarak sunuculara erişip işlerinin bitip bitmediğini kontrol eder, eğer işi bitmişse cevapları alır. Asenkron iletişimin kullanılmasının nedeni : Paralel işlemenin amacı daha hızlı bir sonuç elde etmek olduğuna göre ve bunun için de dağıtık ortamdaki nesneler kullanıldığına göre bu nesnelerin bazılarının kilitlenmeleri ve işlerini yerine getirememeleri durumu göz önüne alınmalıdır. Bu gibi bir hatada normal olarak sonuca varılamaz. Çünkü sonuç için tüm sunucularının işlerini bitirmesi gerekmektedir. İşte bu problemin çözümü için

asenkron iletişim kullanılmıştır ve eğer bir sunucunun çalışmadığı anlaşılırsa ona verilen iş hemen başka bir sunucuya yönlendirilir.

Geliştirilen sistem görüntü işleme alanında iki uygulamaya uygulanmıştır. Birinci uygulama hough dönüşümü işlemidir. Hough dönüşümü bir resimdeki istenilen şekillerin yerlerini belirler. Test için 600X600 boyutunda bir resim kullanılmıştır. Normal hough dönüşüm işlemi yaklaşık olarak 107 saniye zaman almaktadır. Geliştirilen paralel hough dönüşümü programı ile, 12 tane sunucu nesnesi kullanılarak bu süre 13 saniye kadar düşürülmüştür. Görüldüğü gibi bu işlemde yaklaşık 8 kat bir hız artışı sağlanmıştır. 12 sunu ile 12 kat hız artışı alınamayışının nedeni örtüşme problemi ve ağ üzerindeki veri transferi işlemidir. Sunucu nesnelerinin farklı bilgisayarlarda çalıştırılması da bu hızı etkiler.

İkinci uygulama olarak parmak izi analizi işlemi seçilmiştir. Bu algoritmanın hough dönüşümünden farkı burada örtüşme problemin hemen hemen hiç görülmemesidir. Yani bir parmak izi görüntüsü örtüşme olmaksızın alt resimlere ayrılıp paralel olarak işlenebilir. Bu uygulama testinden de başarılı sonuçlar alınmıştır. Test için 512X512 boyutunda bir parmak izi görüntüsü kullanılmıştır. Amaç bu görüntüdeki özel noktaların bulunması idi. Normal olarak bu işlem 62 saniyede bitirilmiştir. Paralel olarak yine 12 nesne ile yapılan testte bu zaman 5.8 saniyeye düşürülmüştür. Buradaki haz kazancı yaklaşık olarak paralel çalışan nesne sayısı ile aynıdır. Bu uygulamanın hough dönüşümüne göre daha iyi sonuç vermesinin nedeni örtüşme probleminin olmamasıdır.

Bu çalışmada geliştirilen paralel sistemde istemci ve sunucular arasında asenkron iletişim kullanılmıştır. Daha önce anlatılan problemi (sunucuların çalışmaması ve görevlerini yerine getirmemesi durumunda) çözerek , senkron (eşzamanlı) bir iletişim kullanılarak nasıl bir sonuç elde edilebileceği incelenebilir. Bu durumda sunucuların işi bittiğinde sonuçları istemciye kendileri iletirler.

Asenkron iletişim modelinde istemcinin devamlı olarak sunuculara işlerini bitirip bitirmediğini sorması işlemi gereksiz yere ağ trafiği oluşturmaktadır. Bunun için bir timer (zamanlayıcı) kullanarak belli zaman aralıklarında sunucular kontrol edilmiştir. Ancak bu iki durumda ağ üzerindeki kazancın ve kaybın ne olduğu testler sırasında ölçülememiştir. Hassas ölçüm sistemleri kullanılarak bu testler yapılabilir. Ağ veri transfer hızı değiştirilerek (10Mbps - 100 Mbps) paralel algoritmanın süresini nasıl etkilediği incelenebilir.

Burada çıkarılan karmaşıklık hesaplarında ağ üzerindeki veri hızı, bant genişliği, bilgisayarların hız farkları göz önüne alınmamıştır. Çeşitli testler yapılarak bu değişkenlerin karmaşıklık üzerinde ne gibi bir etki yaptıkları incelenebilir.

Geliştirilen paralel sistem üzerinde bazı değişiklikler yapılarak farklı uygulama alanında kullanımı araştırılabilir.

KAYNAKLAR

[1] **Brown N. and Kindel C.**,1998 , "Distributed Component Object Model Protocol – DCOM/1.0," Microsoft Corp. ; http://www.microsoft.com/com/

[2] **SELIM G.AKL.** , 1989 , "The Design and Analysis of Parallel Algorithms", Prentice-Hall International Editions .

[3] **P.E. Chung et al.**, 1998 , "DCOM and CORBA Side by Side, Step by Step, and Layer by Layer," C++ Report, Vol.10, No.1, Jan. s. 18-29, 40; http://www.research.microsoft.com/~ymwang/papers/C++R97CR.htm.

[4] **Inprice Corporation**, VisiBroker for installation and Administration; http://www.INPRICE.com/techpubs/borlandvisibroker/vbcpp33/admin/

[5] **Redmond, Wash.**, 1995, "The Component Object Model Specification," Microsoft Corp. , ; http://www.microsoft.com/comdocs.htm.

[6] **The Open Group**, 1997 , Cambridge, Mass, "DCE 1.1: Remote Procedure Call Specification,",; http://www.rdg.opengroup.org/public/pubs/catalog/c706.htm

[7] **Doç. Dr. Gökmen , Muhittin**, 1997, "Computer Vision" , Alternatif Yayın

[8] **Adrian Low**, 1991, " Introductory Computer Vision and Image Processing ", McGraw-Hill

[9] **Ballard, D.H.**, 1981, "Generalizing The Hough Transform To Detect Arbitrary Shapes", *Pattern Recognition*, Vol. 13, No.2 , s.111-122

[10] **B. Moayer and K.S.Fu** FingerPrint , "Fingerprint Classification"

[11] **Andrew K. Hrechak and James A. McHugh**, 1990 "Automated Fingerprint Recognition using structural matching", Pattern Recognition, Vol.23, No.8, s. 893-904

[12] **Thomas F. Krile and John F. Walkup**, Enhancement of fingerprints using digital and optical techniques, Texas Tech University Lubbock, Texas.

[13] **B. M. Mehtre and B. Chatterjee**, 1989, "Segmentation of Fingerprint images – A composite Method", Pattern Recognition, Vol. 22, No.4, s. 381-385

[14] **D.C Douglas Hung**, 1993, "Enhancement and Feature Purification of Fingerprint Images", Pattern recognition, Vol.26, No.11 , s. 1661-1671

[15] **Louis Coetzee and Elizabeth C.Botha**, 1993, "Fingerprint Recognition in Low Quality Images", Pattern recognition, Vol. 26. No. 10, s. 1441-1460

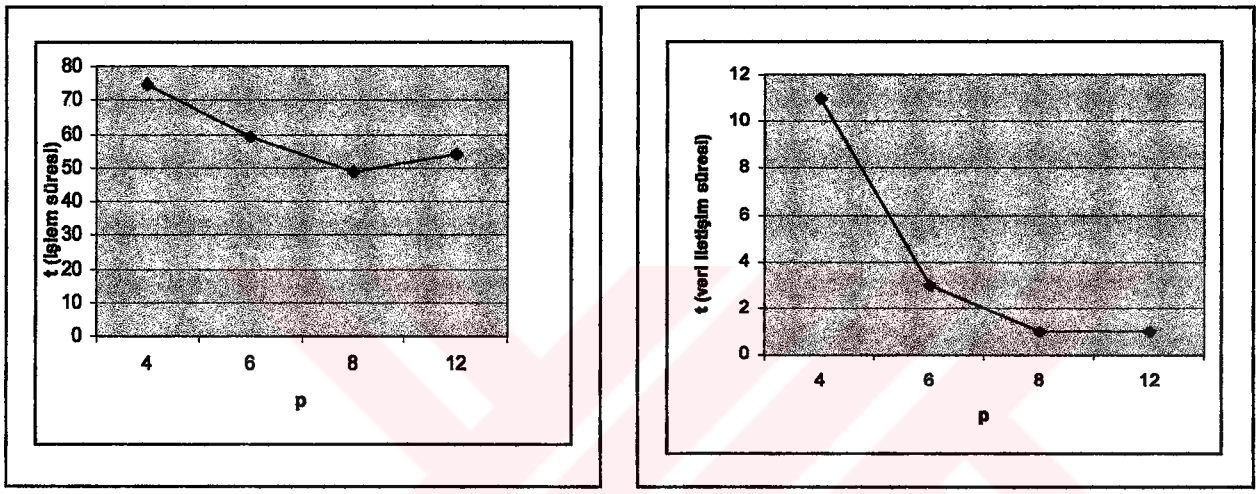
[16] **Louisa Lam, Seong-Whan Lee**, 1992, "Thinning Methodologies – A Comprehensive Survey" , IEEE Transactions on Pattern Analysis and Machine Intelligence , Vol.14 , No. 9

[17] **Quinghan XIAO and Hazem Raafat**, 1991, "Fingerprint Image Postprocessing: A Combined Statistical and Structural Approach", Pattern Recognition , Vol.24, No.10, s. 985-992

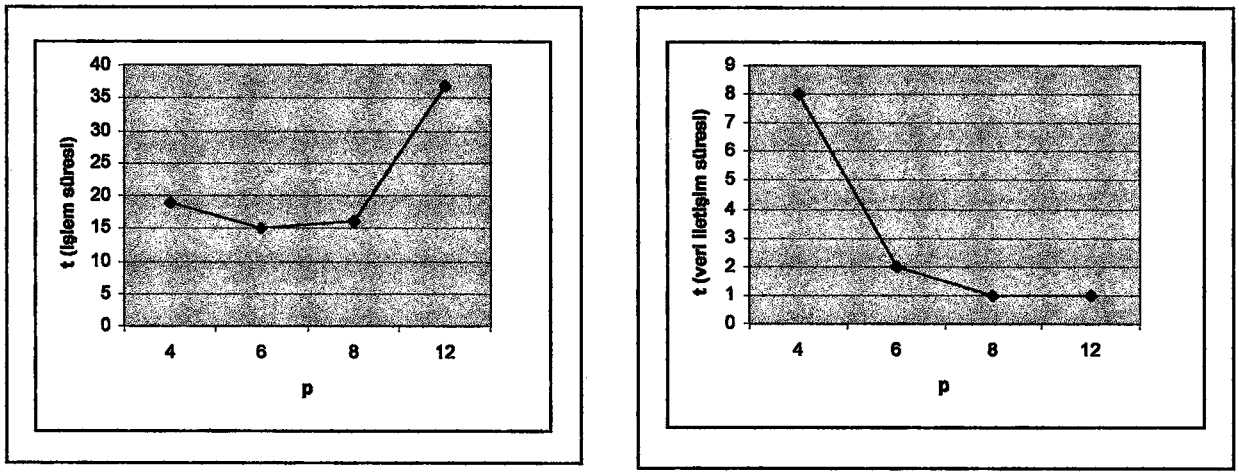
EKLER

EK A

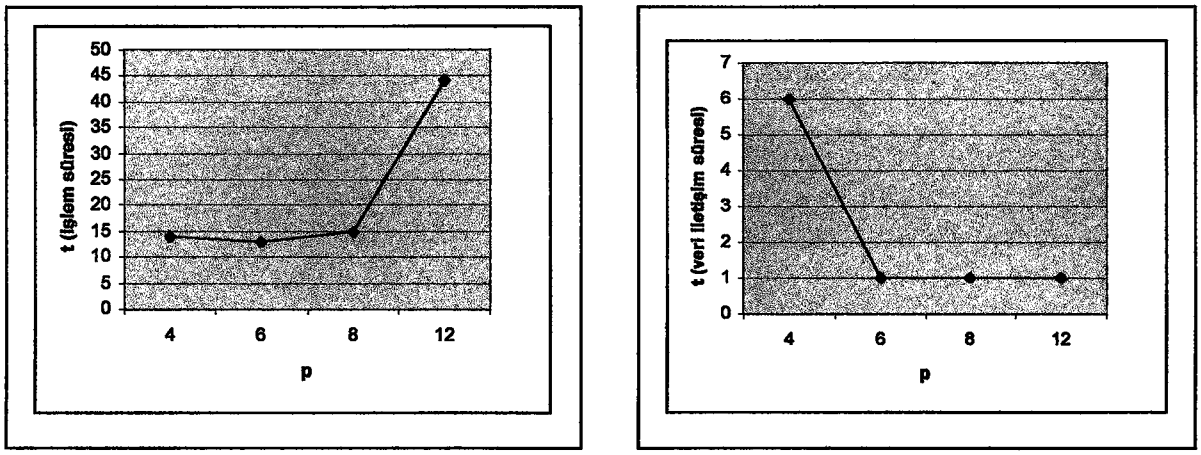
Paralel hough dönüşümü işleminin test sonuçları.



Şekil A.1 n = 2, r=30, p=15 için Toplam işlem ve Veri iletişim süreleri.



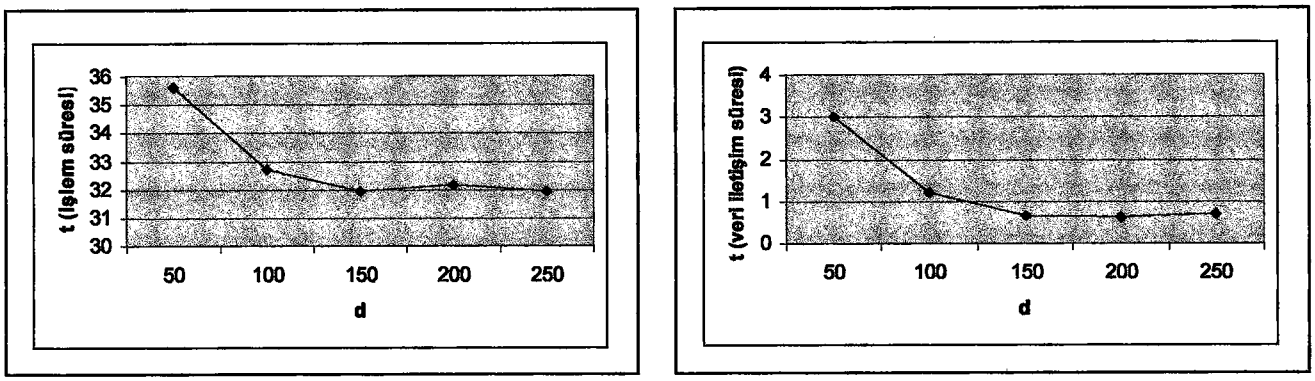
Şekil A.2 n = 8, r=30, p=8 için Toplam işlem ve Veri iletişim süreleri.



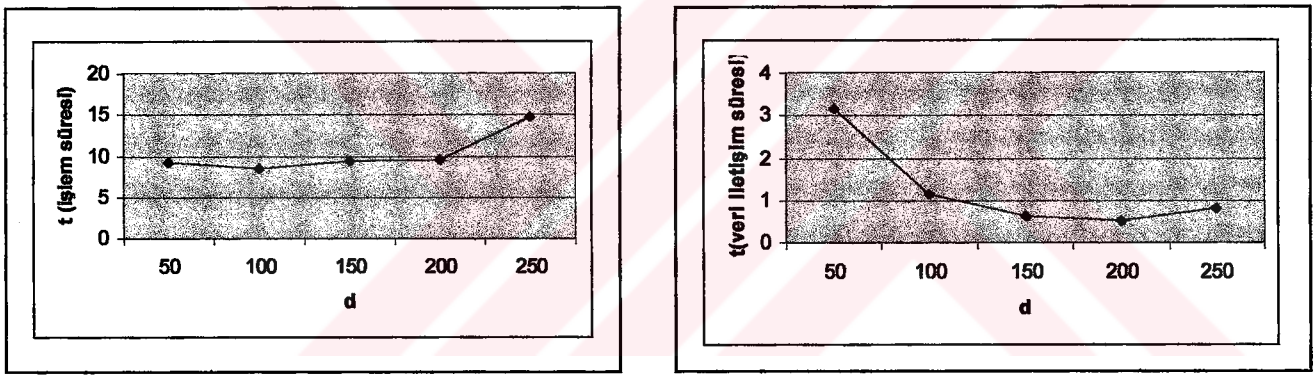
Şekil A.3 n =12, r=30, p=7 için Toplam işlem ve Veri iletişim süreleri.

EK B

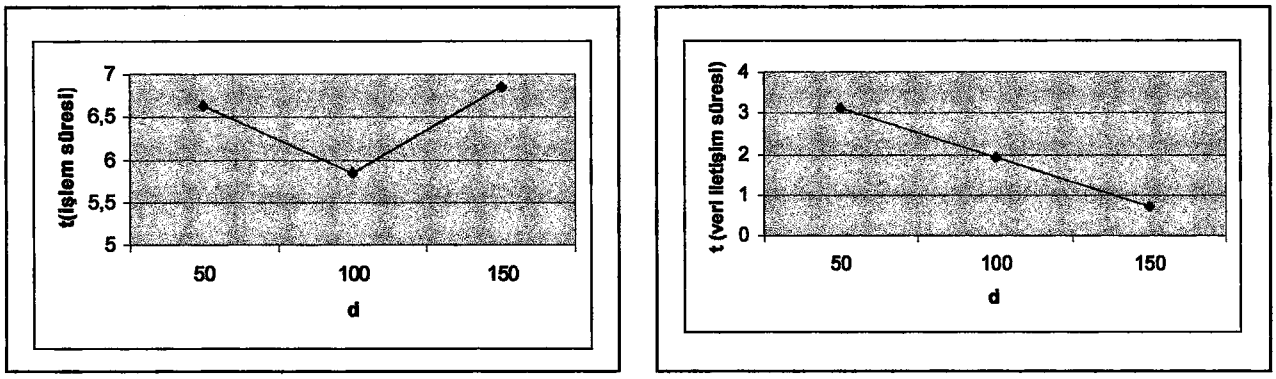
Paralel parmak izi analizi işlemini test sonuçları.



Şekil B.1 n = 2 için Toplam işlem ve Veri iletişim süreleri.



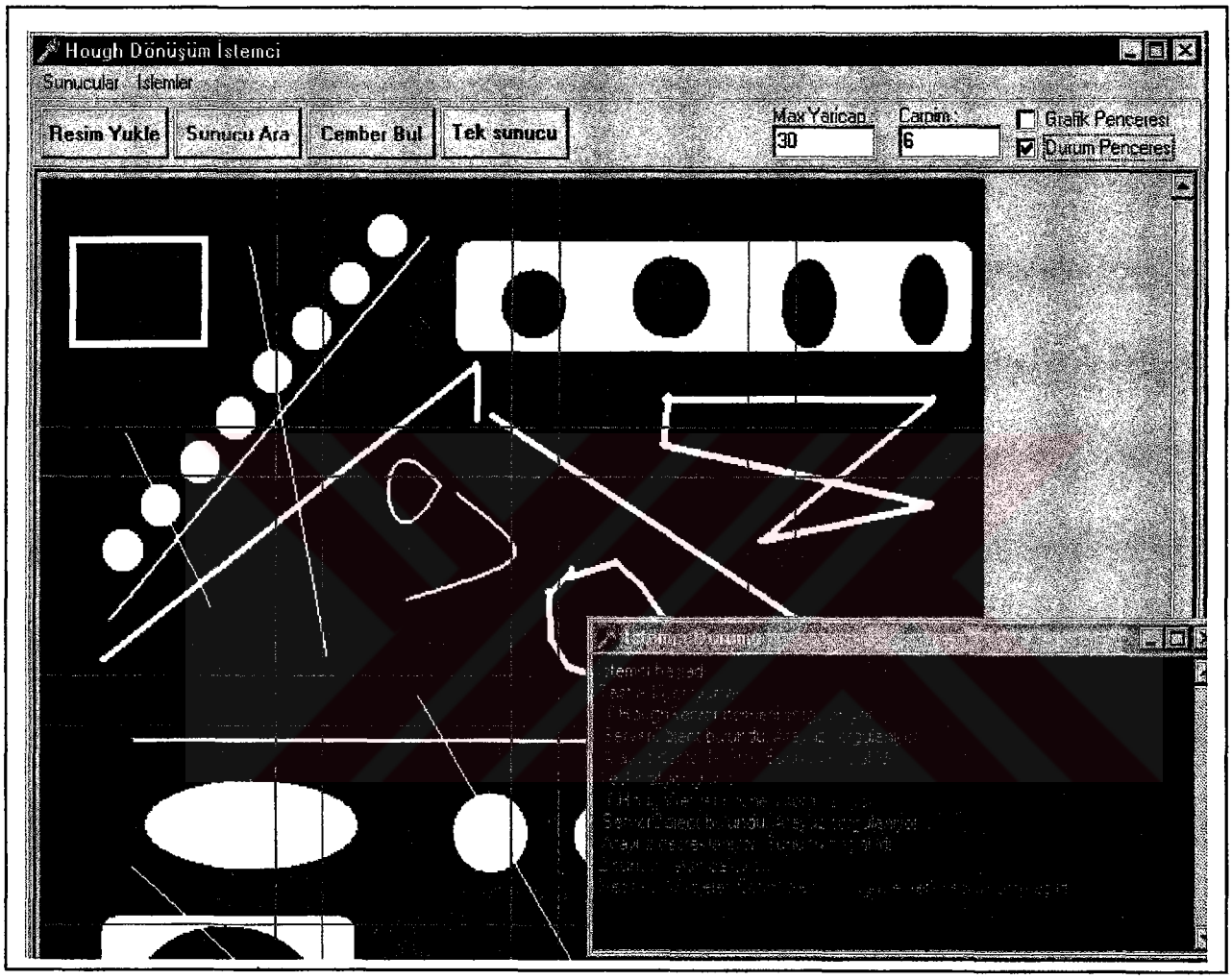
Şekil B.2 n = 8 için Toplam işlem ve Veri iletişim süreleri.



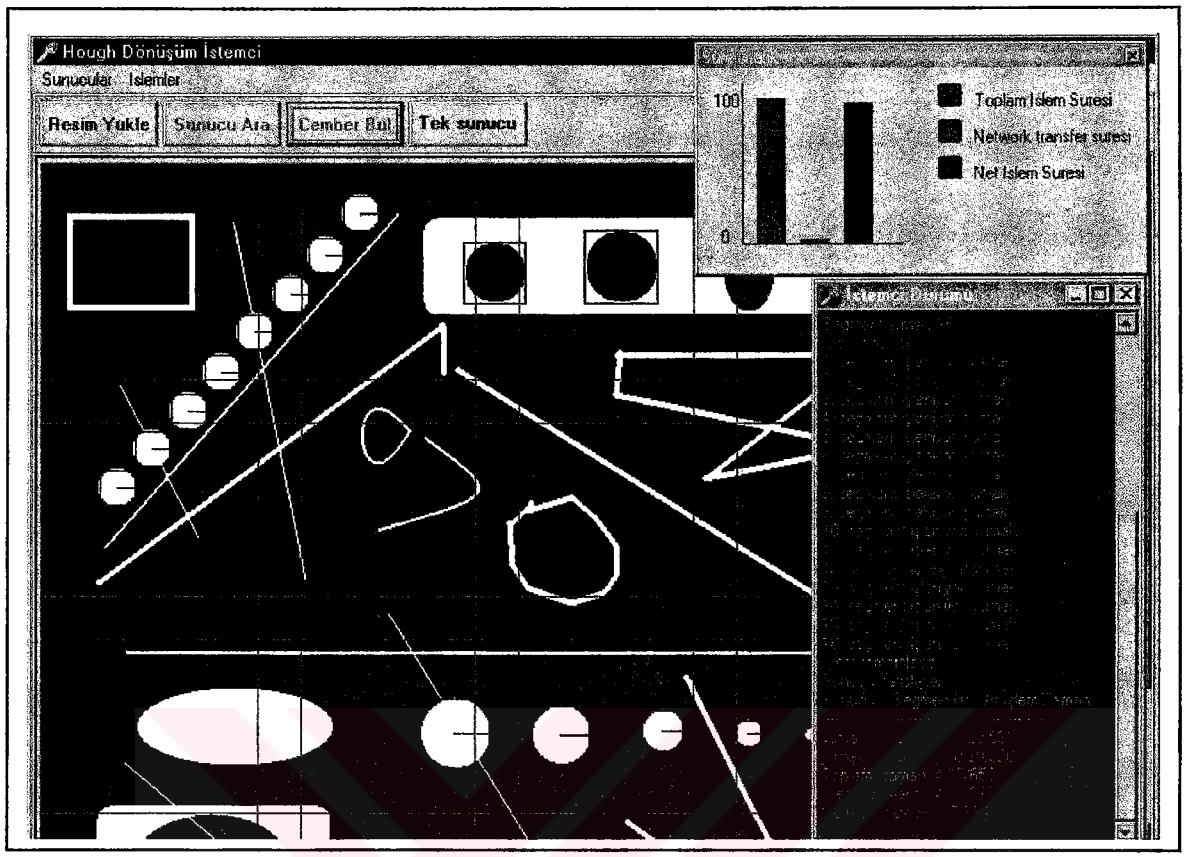
Şekil B.3 n=12 için Toplam işlem ve Veri iletişim süreleri.

EK C

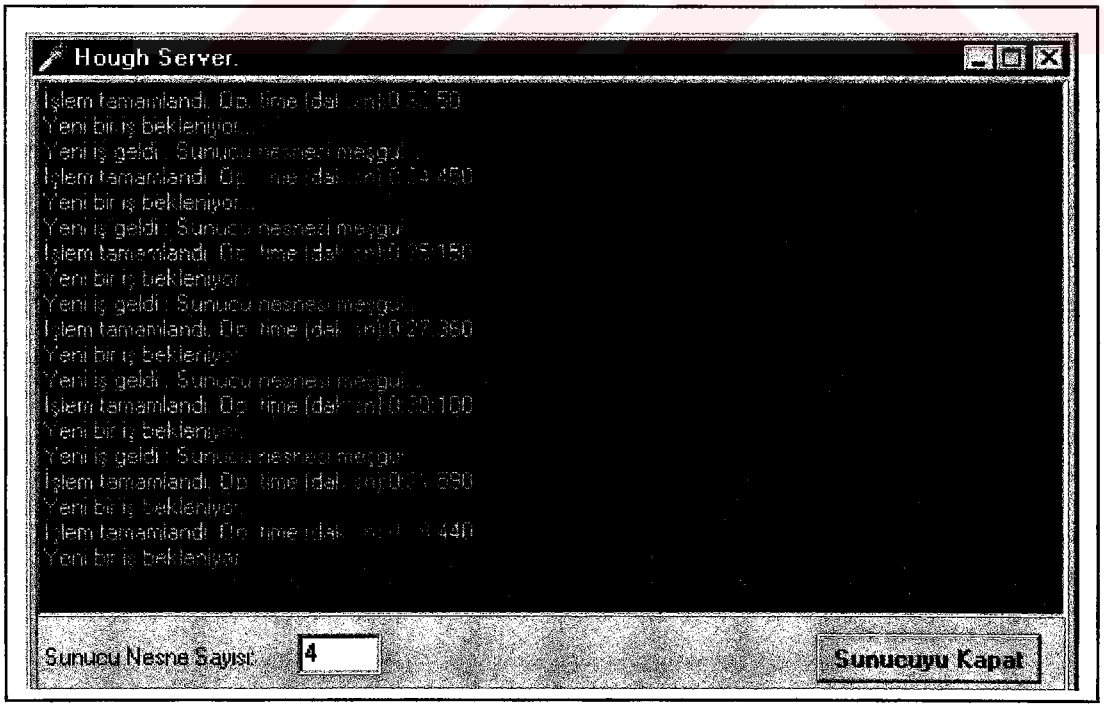
Paralel Hough dönüşümü programının örnek ekran çıktıları.



Şekil C.1 Hough dönüşümü istemci programı ve test edilen resim. Resimdeki çemberler aranmaktadır.



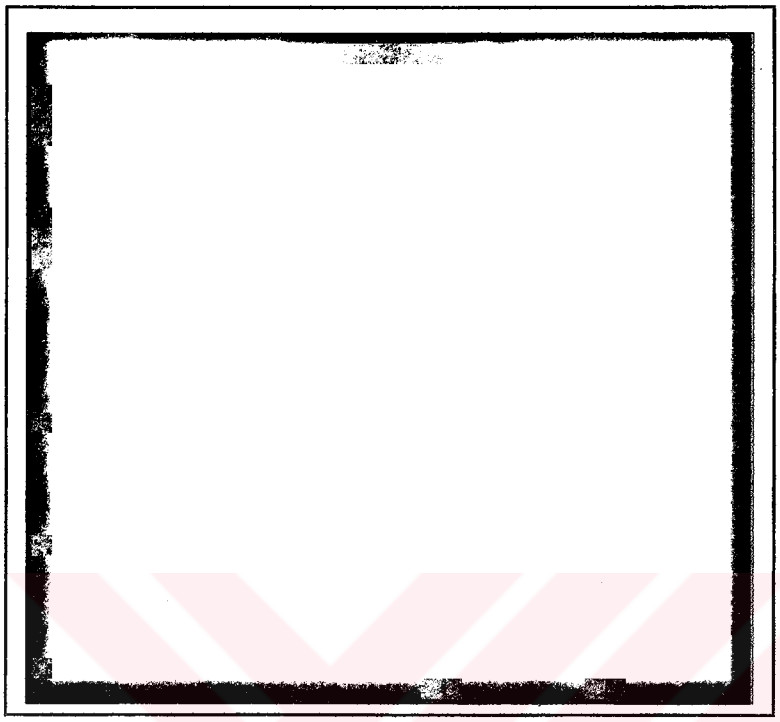
Şekil C.2 Paralel Hough dönüşümü sonucunda bulunan çemberler. Bulunan çemberler kare içinde gösterilmiştir. Toplam işlem süresi ve veri transfer süreleri de grafikte gösterilmiştir.



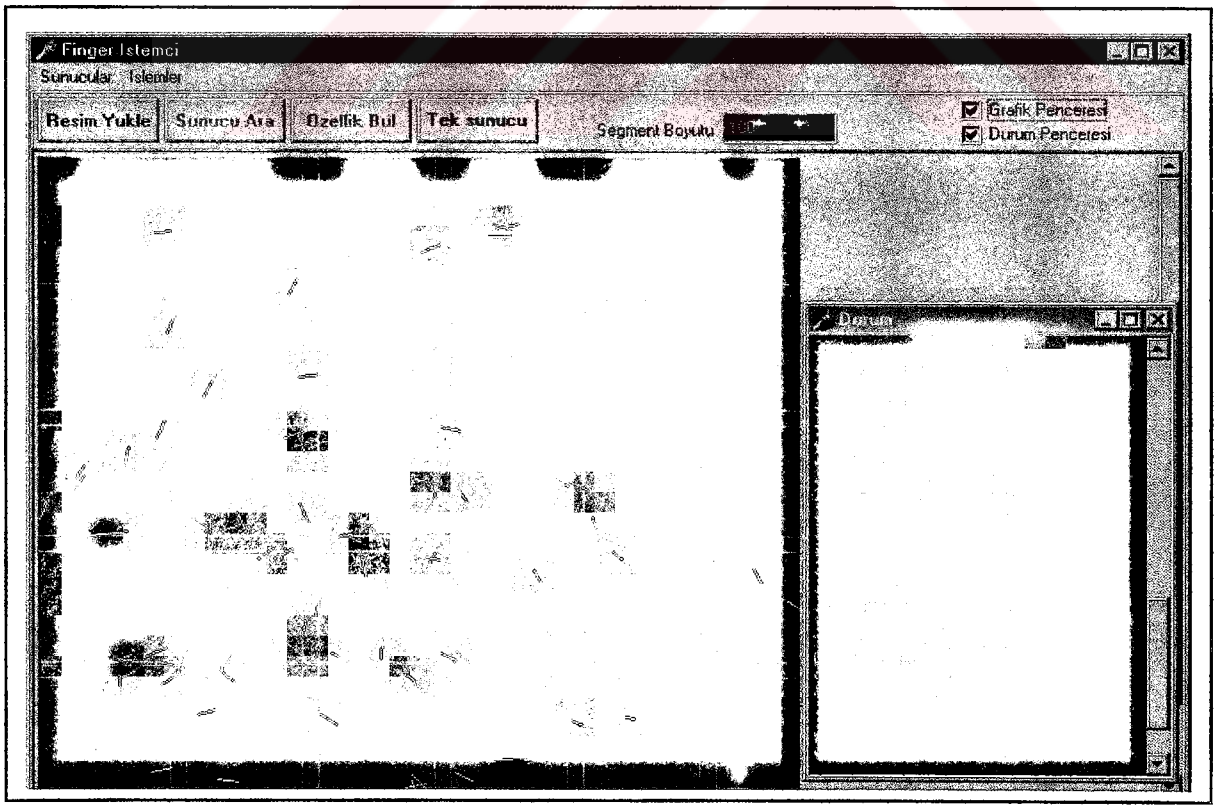
Şekil C.3 Paralel Hough sunucu programı örnek ekranı.

EK D

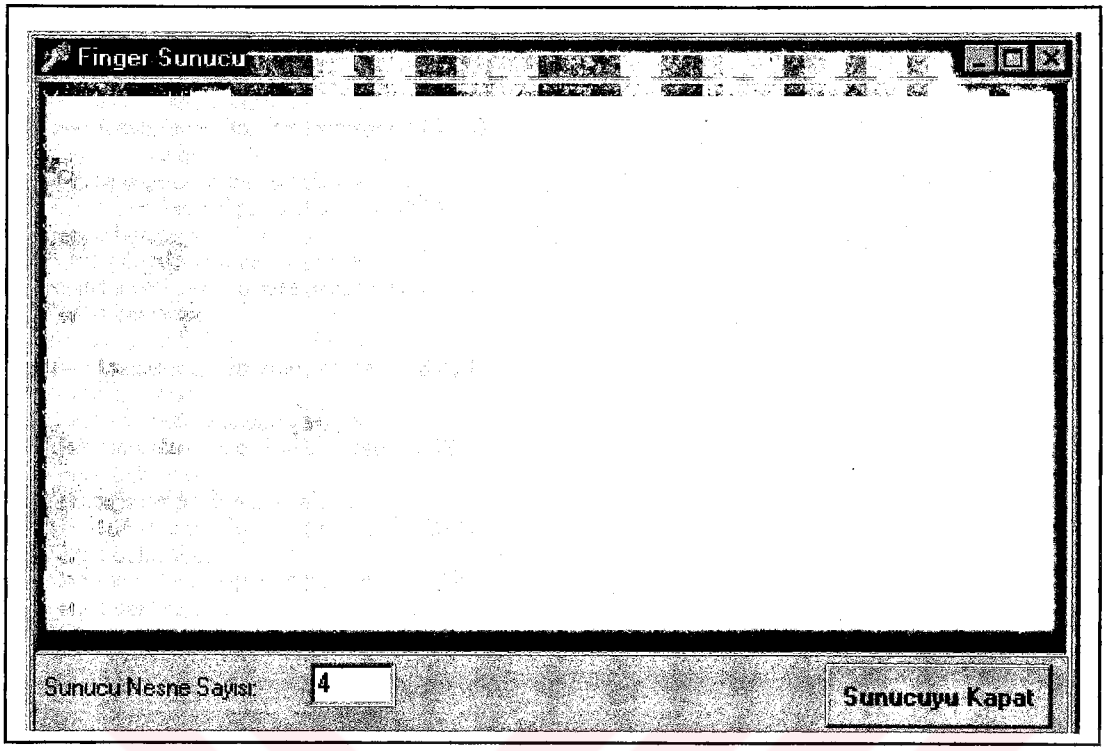
Paralel Parmak izi analiz programının örnek ekran çıktıları.



Şekil D.1 Test edilen parmak izi görüntüsü.



Şekil D.2 Parmak izi analizi istemci programı ve test sonucunda bulunan özellikler.



Şekil D.4 Paralel Parmak izi analizi sunucu programı örnek ekranı.

ÖZGEÇMİŞ

Savaş KÖSE 29 Mayıs 1974'te İstanbul'da doğdu. 1991 yılında Suadiye Lisesinden mezun oldu. 1991 yılında Yıldız Teknik Üniversitesi'nin Bilgisayar Mühendisliği bölümünde okumaya hak kazandı ve 1996 yılının haziran ayında bu bölümden mezun oldu. Ara proje olarak TUPOL 1.0 (Türkçe Programlama Lisanı) adında paralel derleyici tasarımı ve lisans bitirme projesi olarak OPIT 1.0 (Otomatik Parmak İzi Tanımlama Sistemi) programını gerçekleştirdi. 1996'da METACOM firmasında Unix uzmanı olarak çalışmaya başladı. Şu andaki araştırma alanı dağıtık nesne ortamında paralel sistem tasarımı.