İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

DAĞITILMIŞ NESNEYE DAYALI SİSTEMLER İÇİN DAĞITILMIŞ BİLEŞİK NESNE MODELİ

DOKTORA TEZİ

Y. Müh. Güray YILMAZ
504952016

Tezin Enstitüye Verildiği Tarih : 31 Ekim 2001
Tezin Savunulduğu Tarih : 26 Mart 2002

Tez Danışmanı : **Doç.Dr. Nadia ERDOĞAN**

Diğer Jüri Üyeleri

Prof.Dr. A. Emre HARMANCI (İ.T.Ü.)

Prof.Dr. Bülent ÖRENCİK (İ.T.Ü.)

Doç.Dr. Oya KALIPSIZ (Y.T.Ü.)

Doç.Dr. Can ÖZTURAN (B.Ü.)

MART 2002

ÖNSÖZ

Bu tezde dağıtılmış nesneye-yönelik sistemler için yeni bir yaklaşım olan dağıtılmış bileşik nesne modeli ve bu modeli destekleyecek bir ara katman yazılımı geliştirilmiştir.

Tez çalışması süresince bana yol gösteren, önerileri ve görüşleri ile tezin oluşmasını ve bilimsel niteliğinin yükselmesini sağlayan değerli hocam Doç.Dr. Nadia ERDOĞAN'a teşekkür ederim.

Bana bu tez çalışmasını yapabilme imkanı sağlayan, mensubu olmakla her zaman gurur duyduğum, Hava Kuvvetleri Komutanlığı'na ve İstanbul Teknik Üniversitesi'ne de teşekkürü bir borç bilirim.

Son olarak, tez çalışmam sırasında bana sabırla tahammül eden ve her zaman desteğini esirgemeyen değerli eşim Tuba YILMAZ'a sonsuz teşekkürlerimi sunarım.

MART 2002 Güray YILMAZ

İÇİNDEKİLER

KISALTMALAR

AN : Alt Nesne
AY : Adlandırma Yöneticisi
BN : Bileşik Nesne
BY : Başlatma Yöneticisi
CO : Composite Object
CORBA : Common Object Request Broker Architecture
CSCW : Computer Supported Cooperative Work
CSCWE : Computer Supported Collaborative Writing Environment
DBN : Dağıtılmış Bileşik Nesne
DBNS : Dağıtılmış Bileşik Nesne Sunucusu
DBNTO : Dağıtılmış Bileşik Nesne Tabanlı Ortam
DCE : Distributed Computing Environment
DCO : Distributed Composite Object
DCOBE : Distributed Composite Object Based Environment
DCOM : Distributed Component Object Model
DSK : DBNTO Sistem Koordinatörü
DSM : Distributed Shared Memory
DSY : Dinamik Sınıf Yükleyicisi
EY : Erişim Yöneticisi
FO : Fragmented Objects
HTML : HyperText Mark-up Language
HTTP : Hypertext Transfer Protocol
ILU : Inter Language Unification
IPC : Inter Process Communication
ISL : Interface Specification Language
JVM : Java Virtual Machine
LWNS : Light Weight Name Service
MB : Mega Byte
MHz : Mega Hertz
MPI : Message Passing Interface
MRSW : Multiple Reader-Single Writer
ms : Mili Saniye
NFS : Network File System
ns : Nano Saniye
NY : Nesne Yöneticisi
ORB : Object Request Broker
OSÜ : Otomatik Sınıf Üreteci
PRAM : Parallel Random Access Machine
PVM : Parallel Virtual Machine
RAM : Random Access Memory
RMI : Remote Method Invocation

RPC	: Remote Procedure Call
SO	: Subobject
SOS	: Somiw Operating System
SY	: Sunucu Yöneticisi
TCP	: Transfer Control Protocol
UDP	: User Datagram Protocol
URL	: Uniform Resource Locator
WWW	: World Wide Web
YY	: Yerleşke Yöneticisi

TABLO LİSTESİ

ŞEKİL LİSTESİ

DAĞITILMIŞ NESNEYE DAYALI SİSTEMLER İÇİN DAĞITILMIŞ BİLEŞİK NESNE MODELİ

ÖZET

Dağıtılmış sistemler bilgisayar ağları üzerinden sistem kaynaklarının ve bilginin paylaşımını sağlarlar. Bu sistemleri çekici yapan en önemli tasarım konusu dağıtım ile ilgili tüm ayrıntıların kullanıcılara saydam olarak gerçekleniyor olmasıdır. Fakat ne yazık ki, sistem kaynaklarını kullanıcılar arasında saydam bir şekilde paylaştıran ve yöneten genel amaçlı, geniş alana yayılmış dağıtılmış sistemler neredeyse yok denecek kadar azdır. Var olan bazı sistemler de kısıtlı ölçüde kolaylıklar sağlamaktadırlar. Dağıtılmış sistemlerde saydamlık genelde yerel alan sistemleri için sağlanmıştır ve bu sistemler için bulunan çözümler geniş alana ölçeklenememektedirler.

İnternet üzerinden veri paylaşımı gibi geniş alanlı uygulamaların geliştirilmesi için genellikle çok fazla emek sarf etmek gerekmektedir. Bu durum, üzerinde çalışılan işletim sistemi ve diğer sistem çözümlerinden gerekli haberleşme desteğinin sağlanamamış olmasından kaynaklanmaktadır.

Günümüzde iyi bilinen bir çok dağıtılmış sistem uzak-nesne modelini benimsemişlerdir. Bu modelde, nesne tek bir alana yerleştirilir ve bu nesnenin kullanıcısına bir vekil arayüzü üzerinden nesneye saydam erişim olanağı sağlanır. Bu yöntemde nesne en iyi durumda kullanıcısını haberdar etmeksizin bir başka alana göç edebilir. Bununla beraber, uzak-nesne modelinin genelde ölçeklenebilme eksikliğinden kaynaklanan bir çok dezavantajları bulunmaktadır. Kullanıcılara farklı nesne çağrı yöntemlerini gerçekleyecek bir destek sunmamaktadır.

Uzak-nesne modeli ile desteklenen bir dağıtım saydamlığı, paylaşılan bir nesnenin kullanımını ve gerçeklenmesini basitleştirmekle birlikte, bilgili tasarımcıların dağıtımın getirdiği avantajdan yararlanmalarını da önler. Bazı uygulamalar paylaşılan bir nesneye ait verinin veya kodun, çok sayıdaki alan üzerine başarım, kullanılabilirlik, koruma ve yük dengeleme amacıyla dağıtılmasına gereksinim duyulabilirler. Ancak, uzak-nesne modeli bu konuda destek sağlamaz. Bir uzak-nesne belirli bir alan üzerinde bulunur ve kendisine sadece o alanda erişilebilir. Halbuki, dağıtılmış uygulamaların ölçeklenebilirliği ve başarımı için yerellik büyük önem taşır. Paylaşılan nesnelerin farklı alanlara kopyalanması, etkin yerel erişim ve kullanılabilirlik açısından önemli yararlar sağlar.

İnternet ortamında işbirliğiyle yürütülen dağıtılmış uygulamalar genellikle iri ya da orta taneli nesneler üzerinde çalışmalarına rağmen, bu tip uygulamaların kullanıcıları çoğunlukla nesnelerin küçük bir parçasıyla ilgilenmektedirler. Eğer büyük bir nesne daha küçük alt nesnelere bölünebilirse ve kullanıcı alanına nesnenin yalnızca erişim

isteğinin hedefi olan parça gönderilirse, hem belleğin verimli kullanımı söz konusu olacak, hem de farklı alanlar arasında aktarılacak olan verinin miktarı azalacağından, çalışan uygulamanın başarımı da yükselecektir. Alt nesnelerin sağlayacağı bir diğer yarar da, kopyalanmış nesnelerin güncellenmesi sırasında ortaya çıkacaktır. Nesnenin tümünün değil de, bir parçasının güncellenmesi, doğrudan ağda aktarılacak veri miktarını azaltacaktır. Ayrıca, bölünmüş haliyle bir paylaşılan nesne, alt nesnelere paralel erişimin en yüksek düzeyde gerçekleşmesini de sağlayacaktır. Tezde bu noktalar göz önüne alınarak, uzak-nesne modeline alternatif olarak, kopyalamaya dayalı yeni bir model olan *Dağıtılmış Bileşik Nesne (DBN)* modeli önerilmektedir. Ayrıca, önerilen DBN modelinin dağıtılmış ortamda Java programlama dili üzerinden kullanımına imkan sağlayacak bir *Dağıtılmış Bileşik Nesne Tabanlı Ortamın (DBNTO)* tasarım ve gerçekleme ayrıntıları da sunulmaktadır [4-6].

DBN modelinin arkasında yatan temel fikir, çok sayıdaki alt nesnenin bileşiminden meydana gelmiş olan bir bileşik nesnenin farklı alanlar üzerinde fiziksel olarak dağıtılmasıdır. Önerilen modele göre, dağıtılmış ortamdaki uygulamalar bileşik nesneler üzerinden haberleşirler ve birbirleri ile etkileşimde bulunurlar. Bu amaçla, her bir paylaşılan nesne bir arayüz üzerinden kullanılabilen bir metotlar kümesi sunar. Nesneler pasiftir. Etkinlik, nesneleri paylaşan ve onların metotları üzerinde eşzamanlı olarak çağrı yapabilen uygulama programları tarafından sağlanır.

Bir dağıtılmış bileşik nesnenin iki görünüşü vardır: Kullanıcının bakış açısına göre, tekil paylaşılan bir nesnedir. Farklı adres alanlarında bulunan çok sayıdaki istekçi uygulamaları tarafından paylaşılmaktadır. Bu nesneye programcı-tanımlı bir arayüz üzerinden erişilir. Onun bileşenleri ve özellikle de nasıl dağıtıldıkları görünemez. Tasarımcısının bakış açısına göre ise, bir dağıtılmış bileşik nesne, birlikte çalışan bir alt nesneler kümesinden oluşmaktadır. Bu yapıdaki her bir alt nesne merkezi bir gösterilime sahiptir. Yürütme sırasında, kullanıcı istekleri doğrultusunda, bir bileşik nesnenin alt nesneleri farklı adres alanlarına kopyalanarak dağıtılmış bir nesne yapısını oluştururlar.

Dağıtılmış bileşik nesneyi oluşturan alt nesneler ve bunların yapıları yalnızca bu nesnenin tasarımcısı tarafından bilinir. Kullanıcılar ise, nesne hakkında kendilerine sunulan istekçi arayüzü dışında başka bir bilgiye sahip değildirler. Nesne üzerinde bir metot çağrısı yürüttüklerinde, çağrının gerçeklenme aşamalarından haberdar olmazlar.

Önerilen modeli gerçeklemek üzere, dağıtılmış bileşik nesneyi oluşturan alt nesnelerin farklı alanlar üzerinde rahatlıkla bağlanabilmesi amacıyla bir *bağlantı nesnesi* ve alt nesne kopyalarının senkronizasyon ve tutarlılığının sağlanabilmesi amacıyla da bir *kontrol nesnesi* tasarlanmıştır. Bağlantı nesnesi ile dinamik bağlanma sağlanırken, kontrol nesnesi ile, dağıtılmış bileşik nesnenin alt nesneleri bazında düzenlenebilen çeşitli tutarlılık ve senkronizasyon mekanizmaları kurulabilmektedir.

Dağıtılmış bileşik nesnenin farklı alanlarda bulunan alt nesne kopyaları arasında, alt nesne bazında düzenlenebilen çeşitli tutarlılık ve senkronizasyon mekanizmaları kurulabilmekte ve tüm gerçekleme ayrıntıları kullanıcıya saydam olarak kontrol nesnesi ve DBNTO ara katman yazılımı tarafından sağlanmaktadır. Bileşik nesneyi oluşturan alt nesneler, bu nesnelerin farklı alanlar üzerinde kopyalanma ayrıntıları, kopyalar arasında yürütülen tutarlılık protokolleri, eşzamanlılık kontrolünün

sağlanması gibi gerçekleme ile ilgili ayrıntılar nesnenin bir parçası olarak onun arayüzünün arkasına gizlenmiştir.

DBN modelinin gerçeklenmesi için sunulan DBNTO ara katman yazılımı dağıtılmış bileşik nesneleri desteklemek için tasarlanmış Java paketlerinden oluşmaktadır. DBNTO özellikle internet çapına yayılmış bilgisayar destekli müşterek uygulamalar geliştirecek olan kullanıcılar için Java programlama dili üzerinden erişilebilen bir arayüz sağlamak amacıyla tasarlanmıştır. DBNTO bir çok uygulama için gerekli olan nesne paylaşımı, dağıtımı, kopyalama, tutarlılığın sağlanması, uygulamanın dinamik olarak yüklenmesi gibi temel mekanizmalar sunmaktadır.

DBNTO ara katman yazılımını kullanarak, programcılar merkezi bir ortamda çalışıyor görüntüsü altında dağıtılmış uygulamalar geliştirebileceklerdir. Daha sonra, üretilen kod üzerinde herhangi bir değişiklik yapılmaksızın, uygulama dağıtılmış ortamda çalışacak şekilde düzenlenecektir. Bu esnek yapı, dinamik bağlama, senkronizasyon fonksiyonları ve tutarlılık protokolleri gibi kolaylıklar sağlayan kod parçalarının otomatik olarak alt nesnelere eklenmesi suretiyle elde edilmektedir.

Bağlantı ve kontrol nesnelerinin üretildikleri sınıf tanımlamaları alt nesne sınıflarının arayüz tanımlamalarından yararlanılarak, bir sınıf üreteci tarafından otomatik olarak üretilmektedir. Otomatik sınıf üretme işleminde tasarımcıya düşen görev, sadece bir alt nesne metodunun nesne verilerine hangi düzeyde eriştiğini belirlemek ve arayüz tanımlama dosyasını ona göre hazırlamaktır.

Önerilen DBN modelinin başarımını ölçme ve benzer diğer sistemlerle karşılaştırmak amacıyla, DBNTO ara katman yazılımı üzerinde bir takım testler uygulanmıştır. Aynı testler uzak-nesne modeline dayanan ve literatürde iyi bilinen Java RMI yapısı üzerinde de yapılmıştır. Her iki sistemden elde edilen ölçüm sonuçları karşılaştırıldığında, DBN modelinin internet üzerinde müşterek uygulama geliştirmek amacıyla uygun bir yapı sunduğu görülmüştür.

DISTRIBUTED COMPOSITE OBJECT MODEL FOR DISTRIBUTED OBJECT BASED SYSTEMS

SUMMARY

Distributed systems provide sharing of resources and information over a computer network. A key design issue that makes these systems attractive is that all aspects related to the distribution are transparent to users. Unfortunately, general-purpose wide area distributed systems that allow users to share and manage arbitrary resources in a transparent way hardly exist. Constructing wide area applications, such as sharing data across the Internet, often requires a substantial development effort. This is mainly caused by the lack of proper communication facilities as offered by the underlying operating systems and middleware solutions.

All well-known distributed systems today adopt the remote-object model. In this model, an object is located at a single location only, whereas the client is offered access transparency through a proxy interface. At best, the object is allowed to move to other locations without having to explicitly inform the client. There are a number of serious drawbacks to the remote-object model, most notably its lack of scalability. The remote-object model itself provides no mechanisms that support a developer in designing and implementing different invocation schemes, which is necessary if we are to apply scaling techniques such as caching, replication and distribution.

As an alternative to the remote-object model, we propose a new object model, called **Distributed Composite Object** (DCO) model. In our opinion, internet-wide distributed-collaborative applications generally manipulate *large or coarse-grained* objects, while clients of these types of applications are usually interested in only a small part of the object. So, if we partition a coarse-grained object into *finer-grained* **subobjects** (SO), an application's performance would be enhanced by the reduction in the amount of data accessed and the amount of data transferred unnecessarily between distributed sites.

The DCO model is based on the idea that the implementor of an object can distribute the state of the object among multiple SOs. This implies that the state of the *root object* becomes partitioned. These SOs can be placed on different sites, may be replicated, or may even be composite objects themselves.

Several SOs are grouped together and form a **Composite Object** (CO). A CO is created on a single node with its SOs. Then, this CO can be replicated on demand over several nodes together with some of its SOs. All COs which spread out on several nodes constitute a DCO.

The fundamental idea behind the design of the DCO is that it is *physically distributed* over multiple sites. Most current middleware systems, such as CORBA, and DCOM, view a distributed object as an object running on a single machine, possibly with copies on other machines. This object is presented to remote clients as a local object by means of proxies. Our view of what a distributed object is gives us flexibility with respect to partitioning, replication and distribution of the object's state.

The DCO model allows us to describe applications in terms of a single composite object whose implementation details are embedded and encapsulated in different types of SOs. Instead of viewing a distributed object as an entity running on a single machine, possibly with full copies on other machines. We view a distributed object as a conceptual object, distributed over multiple machines with its several SOs.

A DCO has two aspects: From the client's point of view, it is a single shared object. It is shared by several client objects, which are localized in different address spaces, possibly on different sites. It is accessed via a programmer-defined interface. Its components, and particular their distribution, are not visible. From the designers point of view, a distributed object encapsulates a set of cooperating SOs. Each SO of the DCO is an elementary object. In other words, each SO has a centralized representation.

In this thesis, we have also developed a middleware, **D**istributed **C**omposite **O**bject **B**ased **E**nvironment (DCOBE) that facilitates the development of internet-wide distributed applications. DCOBE is a collection of Java packages intended to support the construction of distributed composite objects. Specifically, DCOBE provides a uniform interface for Internet-wide collaborative application developers to use. It also provides basic mechanisms for facilities needed by many applications, such as communication, object composition and replication, consistency management and dynamic deployment of application.

The mechanisms we want to factorize are naming, binding and consistency management. In order to be able to dynamically bind a reference to a remote object, we need to associate a unique name with each object, thus allowing an object to be located and copied to the requesting node. In order to implement object binding without modification to the Java virtual machine, we need to manage indirection objects that allow object faults to be triggered if the reference is not yet bound. In order to manage objects consistency, we need to exchange messages between cooperating nodes to invalidate and update copies according to the consistency model.

The problem is to manage dynamic binding of references to SOs that may be fetched dynamically from remote nodes. To solve this problem, we provide a mechanism for dynamic binding of SO references. Our solution relies on intermediate objects, called *connective objects*, which are inserted between the referenced SO and the object which contains the reference. A connective object contains a reference that points to the referenced SO, if it is already there and null if not. It also contains a unique *object_id* associated with the SO, which allows it to be located and copied on the local host.

The class file of the connective object is generated from the interface of the SO's class to which it points. The connective object implements the same interface as the

SO. Each method invocation is forwarded to the actual SO if the reference is already bound, i.e. if the references in the connective object is not null. If this reference is null, then a function of the runtime system is invoked in order to check whether the SO is already cached. A copy of the SO is fetched if required and the reference in the connective object is updated.

Another problem we have to solve is to efficiently manage distributed replicas of SOs while keeping distribution transparent to the application programmer. Managing object replicas requires mechanisms for invalidating or updating objects in order to ensure consistency. These mechanisms should be hidden to the application programmer.

In order to solve this problem, we decided to utilize another type of intermediate object, called *control object*, which is inserted between the connective object and the actual SO it points to. Similar to the connective object, a control object also forwards method invocations to the referenced SO if its internal reference is not null. Otherwise, a function of the runtime system is invoked in order to fetch a consistent copy of the object and the reference in the control object is updated. In addition, an invalidation of a SO on one node simply consists of assigning the null value to the reference in its control object.

A programmer who uses the DCO model develops applications using the Java Programming Language without any language extensions, nor system support classes, in a manner similar to developing a centralized application program. First, the class code for each subobject class which constitutes a DCO is written. (To provide distinction between this type of classes and normal Java classes, we use prefix *Sub*). After this step, interface declarations are written for these classes. An access mode keyword, which indicates type of the method, reader-R or writer-W, is assigned to each method in the interface declaration.

Since an application is developed in a centralized manner, it does not deal with synchronization and consistency issues. A second step in the configuration is to associate synchronization and consistency mechanisms to each class. In other words, the class file of the control object is generated in this step.

After the creation of the control object's class file, an application can be configured for distribution by creating the class file of the connective object, which is generated from the interface definition of the subobject class.

In order to measure its performance, several tests have been made on DCOBE middleware. These tests are also applied to a well-known distributed object system, JAVA RMI. On comparing the test results of both systems, we conclude that the DCO model provides the users with shorter access times and is especially suitable for computer supported cooperative work applications on the internet.

1. GİRİŞ

Dağıtılmış sistemler bilgisayar ağları üzerinden sistem kaynaklarının ve bilginin paylaşımını sağlarlar. Dağıtım ile ilgili tüm ayrıntıların kullanıcılara saydam olarak gerçekleniyor olması bu sistemleri çekici yapan önemli bir tasarım konusudur. Fakat ne yazık ki, sistem kaynaklarını kullanıcılar arasında saydam bir şekilde paylaştıran ve yöneten genel amaçlı, *geniş alana yayılmış dağıtılmış sistemler (wide area distributed systems)* neredeyse yok denecek kadar azdır. Var olan bazı sistemler de kısıtlı ölçüde kolaylıklar sağlamaktadırlar. Dağıtılmış sistemlerde saydamlık genelde *yerel alan sistemleri (local area systems)* için sağlanmıştır. Fakat, bu sistemler için bulunan çözümler geniş alana ölçeklenememektedirler.

İnternet üzerinden veri paylaşımı gibi geniş alan uygulamalarının geliştirilmesi için genellikle çok fazla gayret sarf etmek gerekmektedir. Bu durum esas olarak, üzerinde çalışılan işletim sistemi ve diğer sistem çözümlerinden gerekli haberleşme desteğinin sağlanamamış olmasından kaynaklanmaktadır.

Geniş alan uygulamaları kullanıcılarına bilgi alış-verişi ve paylaşımı için bazı kolaylıklar sağlarlar. Bu bağlamda, dağıtılmış sistemler üzerine günümüze değin bir çok proje geliştirilmiştir. Geliştirilen bu projeler, genelde geniş alan ağı ortamında, bilinen dosya işlemleri sunmuşlardır. Uygulamaların tümü iç yapılarında, basit ve tekdüze bir kavramsal modele sahiptirler. Bu modeller genelde düşük-düzeyli haberleşme ilkellerini kullanarak gerçeklenmişlerdir. Var olan bu ilkeller; *uzaktan yordam çağırma (Remote Procedure Call-RPC), noktadan-noktaya mesaj iletimi (point-to-point message passing), akışa-yönelik haberleşme (stream-oriented communication), dağıtılmış paylaşılan bellek (distributed shared memory)* kullanımı gibi yöntemlerdir. Ancak, bu nispeten düşük düzeyli haberleşme modelleri uygulamada bazı problemlerin de kaynağı olurlar. Öncelikle, bu modeller *kopyalama (replication)* ve *verinin göç ettirilmesi (data migration)* gibi, tüm dağıtılmış sistemler için geçerli olan haberleşme problemlerine çözüm sağlayamamaktadırlar. Sonuçta, her uygulama amacına uygun, kendine ait bir çözüm aramak zorunda kalmaktadır.

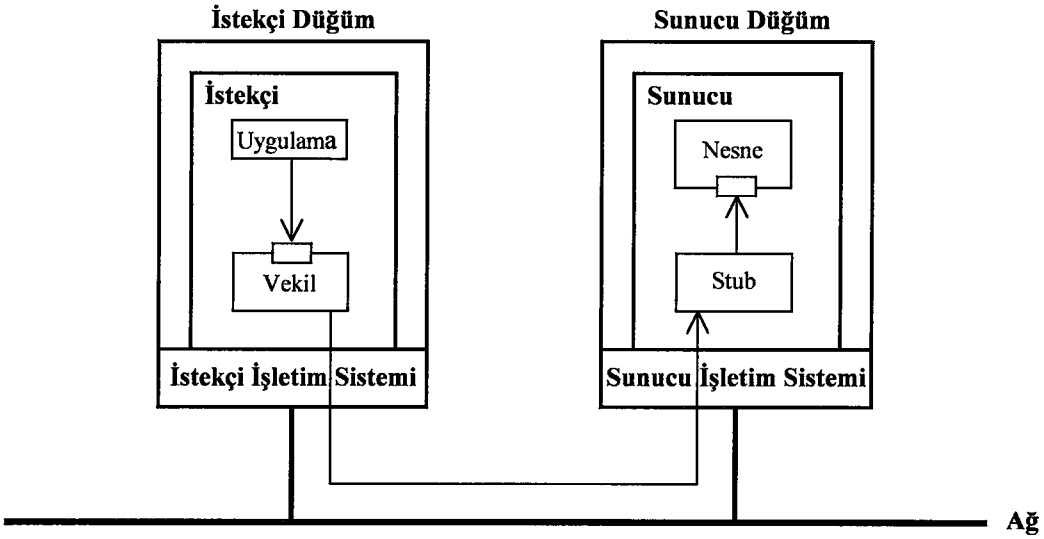
İkinci olarak, sunulan kolaylıkların çeşitliliği farklı gerçeklemeleri gerektirdiği için, uygulamaları tek bir çatı altında bir araya getirmek, ya da uygulamalar arasında bilgi alış-verişini sağlamak zorlaşmaktadır. Ayrıca, düşük düzeyli haberleşme modelleri semantikleri ve arayüzleri açısından incelendiğinde, genelde bir standardın var olmadığı görülür. Sonuç olarak, dağıtılmış bir uygulamanın başarımını etkileyen en önemli tasarım konusu olan bilgi alış-verişi ve paylaşımı problemi, uygulamanın bir parçası olarak ele alınacak, fakat her tasarımcı farklı bir çözüm geliştireceği için uygulamalar arasında bir heterojenlik problemi ortaya çıkacaktır.

Bu noktada, geniş alan dağıtılmış sistemleri için bilgi alış-verişi ve paylaşımı konusunda karşılaşılan haberleşme problemlerini çözümleyecek bir modele ihtiyaç olduğu görülmektedir. Model, basit haberleşme ilkellerine oranla daha soyut düzeyde ve aynı zamanda uygulamadan da bağımsız olmalıdır. Özellikle internet üzerinde ortaklaşa iş yürütmeyi destekleyen, farklı yapılardaki bir çok geniş alan uygulamasının geliştirilebilmesine olanak sağlayan genel ve basit bir gerçekleme ortamı sunmalıdır.

İki sebepten dolayı, dağıtılmış sistemler için nesne tabanlı çözümlerin daha elverişli olduğu görülmüştür. Öncelikle, nesne kavramı işlevselliği gerçeklemeden ayıran doğal bir yapı sunmaktadır. Bu ayırım gereklidir. Çünkü, geniş alan uygulamaları bileşenlerin farklı özelliklerdeki alt sistemler üzerine dağıtılmalarını gerektirebilir. Bu durum, aynı işlevselliğe sahip, farklı gerçeklemelerin bulunmasını zorunlu kılar. İkinci sebep ise, nesnelerin verilere yalnızca önceden tanımlanmış olan işlemler üzerinden erişilmesine izin veren doğal yapısıdır.

Günümüzde, DCOM [1], DCE [2] ve CORBA [3] gibi iyi bilinen bir çok geniş alan dağıtılmış sistemi *uzak-nesne (remote object)* modelini benimsemişlerdir. Bu modele göre, nesne Şekil 1.1'de görüldüğü gibi tek bir adres alanına yerleştirilir ve kullanıcıya bir *vekil (proxy)* arayüzü üzerinden nesneye saydam bir şekilde erişim olanağı sunulur. En iyi durumda, nesne kullanıcısını haberdar etmeksizin bir başka alana göç edebilir. Fakat, uzak-nesne modelinin genelde ölçeklenebilme eksikliğinden kaynaklanan bir çok dezavantajları bulunmaktadır. Uzak-nesne modeli kullanıcılara nesneyi kopyalama veya kendi adres alanına yükleyerek yerel çağrıda bulunma gibi farklı çağrı yöntemlerini gerçekleyecek bir destek sunmamaktadır. Halbuki, sisteme verinin farklı alanlar üzerinde fiziksel olarak dağıtımı ve nesnenin

veya dağıtılan nesne parçalarından bir kısmının kopyalaması gibi bazı olanaklar eklenmek istendiğinde bu destek oldukça gerekli olmaktadır.



Şekil 1.1. Bir uzak-nesne üzerinde yürütülen metot çağrısının gerçeklenme adımları

1.1 Tezin Amacı

Bu tezin amacı, geniş alana dağıtılmış sistemlerdeki haberleşme problemine yönelik yeni bir "dağıtılmış paylaşılan nesne modeli" geliştirmek ve bu modeli destekleyecek bir çalışma ortamını tasarlamak ve gerçeklemektir. Böylelikle, farklı alanlarda bulunan çok sayıdaki kullanıcı arasında bilgi paylaşımına ve ortaklaşa çalışmaya imkan sağlayan, *bilgisayar destekli birlikte iş yürütme (Computer Supported Cooperative Work-CSCW)* uygulamaları için paylaşılan nesneler kümesi olarak düzenlenmiş, yeni bir dağıtılmış haberleşme ortamı sunmaktır.

CSCW uygulamalarına örnek olarak, dökümanların birbirlerinden uzak kullanıcılar arasında eşzamanlı olarak paylaştırılmasına izin veren editör sistemleri ya da yine farklı alanlarda yer alan kullanıcı gruplarının üzerinde ortaklaşa çalıştıkları bir bilgisayar destekli tasarım uygulaması verilebilir.

Uzak-nesne modelindeki gibi desteklenen bir dağıtım saydamlığı, paylaşılan bir nesnenin kullanımını ve gerçeklenmesini basitleştirmekle birlikte, bilgili tasarımcıların dağıtımın getirdiği avantajdan yararlanmalarını da önler. Paylaşılan bir nesneye ait verinin veya kodun, çok sayıdaki alan üzerine *başarım (performance),*

kullanılabilirlik (availability), koruma (protection) ve yük dengeleme (load balancing) amacıyla dağıtılmasına gereksinim duyulabilir. Ancak, uzak-nesne modeli bu konuda destek sağlamaz. Bir uzak-nesne belirli bir alan üzerinde bulunur ve kendisine sadece o alanda erişilebilir. Dağıtılmış uygulamaların ölçeklenebilirliği ve başarımı için *yerellik (locality)* büyük önem taşır. Paylaşılan nesnelerin farklı alanlara *kopyalanması (replication)* etkin yerel erişim ve kullanılabilirlik açısından önemli yararlar sağlar.

Kopyalanmış nesne sistemleri, yerel veriler nedeniyle, uzak-nesnelere oranla çok daha hızlı yanıt sürelerine sahiptirler. Özellikle okuma erişimlerinin yazma erişimlerine oranla daha fazla olduğu durumlarda, nesnelerin çağrının yapıldığı alan üzerine kopyalanması başarımı önemli ölçüde arttıracaktır. Ayrıca, çağrı ve geri dönüş parametrelerinin fazla olduğu durumlarda, iletim sırasında aktarılacak veri miktarının da artacağı göz önünde bulundurulduğunda, uzak-nesne erişimine nazaran, nesnenin kopyalanarak yerel erişim imkanının sağlanmasının daha elverişli olduğu görülmektedir.

Tabii ki, nesnelerin farklı alanlar üzerinde kopyalanması sisteme bir ek yük de getirmektedir. Bir nesne kopyası üzerinde yapılan bir çağrı eğer o kopyayı diğerlerinden farklı kıldıysa, bu durumda nesne kopyaları arasında tutarlılık sağlanmalıdır. Bu amaçla literatürde geliştirilmiş bir çok yöntem bulunmaktadır. Bu yöntemler ilerleyen bölümler içerisinde ayrıntılı olarak incelenecektir.

Kopyalama ile birlikte sisteme gelen bir diğer ek yük de, nesne erişiminde kopyalar arasında senkronizasyonun sağlanmasıdır. Aynı nesnenin farklı kopyaları üzerinde eşzamanlı olarak yürütülen çağrıların var olması halinde, eğer bu çağrılardan en az biri nesne durumu üzerinde değişiklik yapan bir çağrı ise, nesnenin bütünlüğü bozulabilir. Bu yüzden, nesne bütünlüğünü bozabilecek çağrıların ardışıl olarak yürütülmesinin sağlanması gerekmektedir.

CSCW uygulamaları genellikle büyük miktarlardaki paylaşılan veri blokları ile yürütülürler. Aynı şekilde, nesne tabanlı CSCW uygulamaları da *iri ya da orta taneli nesneler (large or coarse grained objects)* üzerinde çalışmaktadırlar. Fakat, bu tip uygulamaların kullanıcıları genelde bu nesnelerin bütünüyle değil, küçük bir alt parçasıyla ilgilenmektedirler. Bundan dolayı, kopyalanmış paylaşılan nesnelere

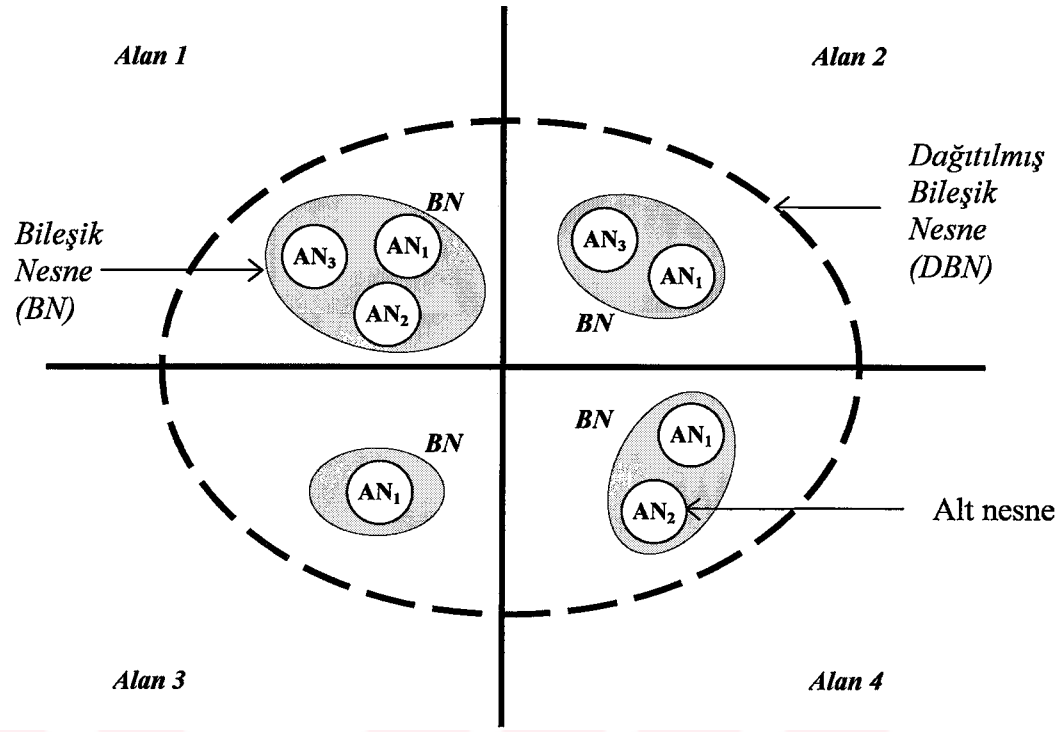
dayalı bir sistem içinde, erişim sırasında tüm nesnenin kullanıcının bulunduğu alana taşınması ve yüklenmesi yerine, sadece hedeflenen alt parçanın kopyalanması daha etkin bir çözüm yolu olarak görülmektedir. O halde, eğer büyük bir nesne daha küçük alt nesnelerin bileşimi şeklinde tasarlanarak ifade edilirse, erişilecek ve farklı alanlar arasında aktarılacak olan verinin miktarı azalacağı için, çalışan uygulamanın başarımı ve kullanılabilirliği önemli oranda yükselecektir.

Bu tez çalışmasında, uzak-nesne modeline alternatif olarak, kopyalamaya dayalı yeni bir model olan *dağıtılmış bileşik nesne (DBN)* modeli önerilmektedir. Ayrıca, önerilen DBN modelinin dağıtılmış ortamda Java programlama dili üzerinden kullanımına imkan sağlayan *Dağıtılmış Bileşik Nesne Tabanlı Ortamın (DBNTO)* tasarım ve gerçekleme ayrıntıları da sunulmaktadır [4-6].

1.2 Dağıtılmış Bileşik Nesne Modeli ve Destekleme Ortamı

Önerilen modele göre, dağıtılmış ortamdaki uygulamalar bileşik nesneler üzerinden haberleşirler ve birbirleri ile etkileşimde bulunurlar. Bu amaçla, her bir paylaşılan nesne bir arayüz üzerinden kullanılabilen bir metotlar kümesi sunar. Nesneler pasiftir. Etkinlik, nesneleri paylaşan ve onların metotları üzerinde eşzamanlı olarak çağrı yapabilen uygulama programları tarafından sağlanır. Bu modele göre, bir dağıtılmış paylaşılan nesne, Şekil 1.2'de sunulduğu gibi, öncelikle tekil bir adres alanı üzerinde bir veya daha fazla sayıda *alt nesne (AN)*'nin bir kabuk nesne altında birleştirilmesi suretiyle bir *bileşik nesne (BN)* olarak yaratılır. Bileşik nesne diğer adres alanları üzerinde, yalnızca o alanlarda "çalışan uygulamaların gerektirdiği alt nesneler" ile birlikte kopyalandıktan sonra, dağıtılmış bileşik nesne yapısı oluşur.

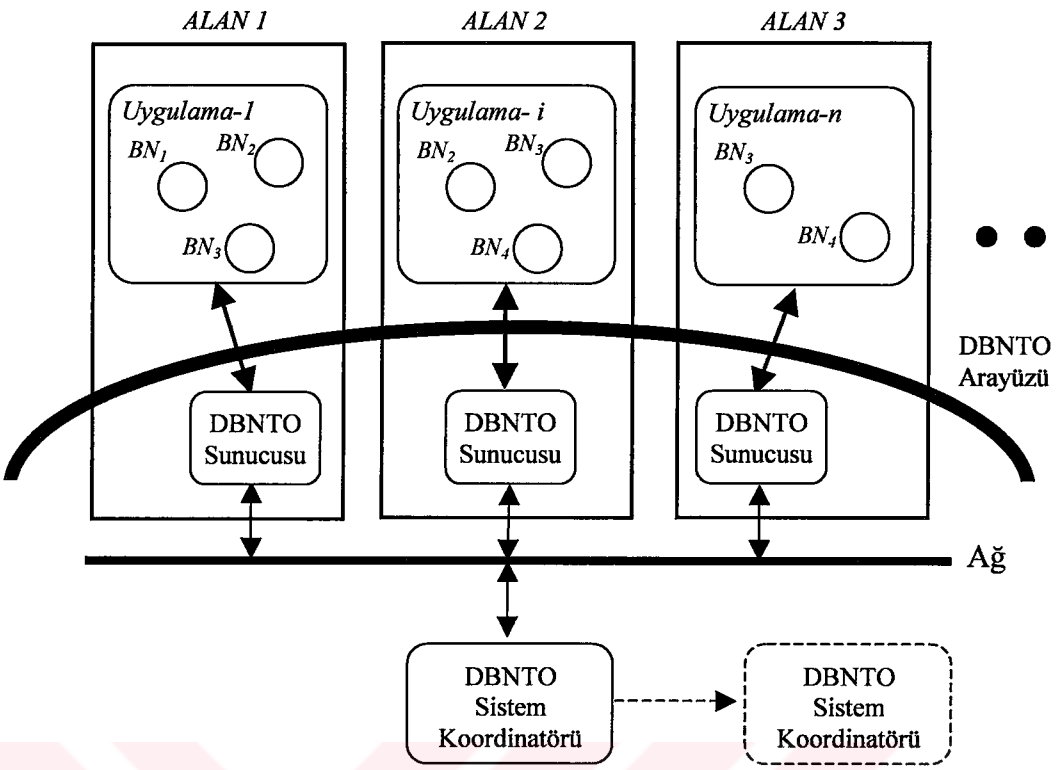
Şekil 1.2'de, *Alan 1* üzerinde üç alt nesnesi (AN_1, AN_2 ve AN_3) ile yaratılan BN bileşik nesnesi görülmektedir. *Alan 2*'de yer alan bir uygulamanın BN üzerinde yürütmüş olduğu metot çağrıları sonucunda bileşik nesne yalnızca gerek duyulan AN_1 ve AN_3 alt nesneleri ile *Alan 2* üzerinde kopyalanmıştır. Benzer şekilde, *Alan 3* üzerinde AN_1, *Alan 4* üzerinde de AN_1 ve AN_3 alt nesneleri ile birlikte kopyalanması suretiyle bir "dağıtılmış bileşik nesne" ortaya çıkmıştır.



Şekil 1.2. Dört farklı adres alanı üzerine yayılmış bir dağıtılmış bileşik nesne

Önerilen modelde, dağıtılmış bileşik nesnenin farklı alanlarda bulunan alt nesne kopyaları arasında, alt nesne bazında düzenlenebilen çeşitli tutarlılık ve senkronizasyon mekanizmaları kurulabilmekte ve tüm bu gerçekleme ayrıntıları kullanıcıya saydam olarak DBNTO ara katman yazılımı tarafından sağlanmaktadır. Bileşik nesneyi oluşturan alt nesneler, bu nesnelerin farklı alanlar üzerinde kopyalanma ayrıntıları, kopyalar arasında yürütülen tutarlılık protokolleri, eşzamanlılık kontrolünün sağlanması gibi gerçekleme ile ilgili tüm ayrıntılar nesnenin bir parçası olarak onun arayüzünün arkasına gizlenmiştir.

DBN modelinin gerçeklenmesi için sunulan DBNTO sistemi dağıtılmış bileşik nesneleri desteklemek için tasarlanmış Java paketlerinden oluşmaktadır. DBNTO özellikle internet çapına yayılmış CSCW uygulamaları geliştirecek olan kullanıcılar için Java programlama dili [7] üzerinden erişilebilen bir arayüz sağlamak amacıyla tasarlanmıştır. Yapısı Şekil 1.3'te verilen DBNTO, bir çok uygulama için gerekli olan nesne paylaşımı, dağıtımı, kopyalama, tutarlılığın sağlanması, uygulamanın dinamik olarak yüklenmesi gibi temel mekanizmalar sunmaktadır.



Şekil 1.3. DBNTO sisteminin genel görünüşü

Şekil 1.3'teki her bir alanda yer alan uygulama, *DBNTO Sunucusu* olarak adlandırılan, bir sistem birimi aracılığıyla dağıtılmış bileşik nesne ortamına giriş yapacaktır. Ayrıca, sunucuların ortama giriş/çıkış işlemlerini denetleyen ve yeni katılan bir sunucuyu diğerlerine tanıtan, ayrıca bir DBN'yi oluşturan her bir alt nesne için tekil bir nesne tanımlayıcısı sağlayan, *DBNTO Sistem Koordinatörü* olarak adlandırılan, bir birim daha bulunmaktadır. Sistemin hata-hoşgörü oranını arttırmak amacıyla, Şekil 1.3'te kesikli çizgilerle gösterilmiş olan, yedek bir sistem koordinatörü daha vardır. Herhangi bir şekilde asıl koordinatörün devre dışı kalması durumunda, yedek koordinatör üzerinden sistem problemsiz bir şekilde çalışmasına devam edebilecektir.

Programcılar DBNTO sunucusunun kendilerine sağladığı arayüzü kullanarak, uygulamalarını sanki merkezi bir ortamda çalışıyorlarmış gibi geliştirebileceklerdir. Daha sonra, kod üzerinde herhangi bir değişiklik yapılmaksızın, uygulama dağıtılmış ortamda çalışacak şekilde düzenlenecektir. Bu esnek yapı, dinamik bağlama, senkronizasyon fonksiyonları ve tutarlılık protokolleri gibi kolaylıklar sağlayan kod parçalarının otomatik olarak alt nesnelere eklenmesi suretiyle elde edilmektedir.

DBNTO sistemi kullanıcılarına şu temel fonksiyonları sunmaktadır:

- **İnternet uygulamaları için destek sağlama:** DBNTO özellikle, internet üzerine yayılmış dağıtılmış nesne-tabanlı uygulamaların geliştirilmesi için tasarlanmış olan bir *arakatman (middleware)* yazılımıdır.

- **Dağıtılmış bileşik nesne modeli:** DBNTO, DBN nesne modelini esas almaktadır. DBN'nin tasarımındaki temel fikir, bir bileşik nesnenin alt nesnelerinin kopyalanması suretiyle fiziksel olarak dağıtılmış olmasıdır. Diğer bir deyişle, nesnenin durumu, aynı anda farklı adres alanları üzerine yerleştirilmiş bir çok alt nesnenin durumlarından oluşmaktadır.

- **Kopyalama:** Dağıtılmış ortamda birlikte yürütülen uygulamalar için başarımı etkileyen en önemli faktörlerden biri de kopyalamadır. DBNTO, dağıtılmış bileşik nesnenin paylaşılan alt nesnelerinin birlikte iş yürütmekte olan farklı alanlar üzerinde kopyalanmasına izin verir. Böylece, dağıtılmış nesneler üzerinde metot çağrıları, uzak-nesne modelinden farklı olarak, birer yerel çağrı şeklinde yürütülürler ve bu şekilde iletim gecikmesi süresi de azaltılır. Ayrıca, kopyalama sırasında bileşik nesne yeni alan üzerinde tümüyle kopyalanmaz. Hangi alt nesnelerin ilgili alan üzerinde kopyalanacağını o alanda yürütülen metot çağrılarının şekli belirler. Yani, her bir alt nesne, ihtiyaç duyuldukça yeni alan üzerinde kopyalanır.

- **Tutarlılık:** Bir dağıtılmış bileşik nesnenin farklı alanlar üzerinde kopyalanmış olan alt nesnelerinin tutarlılığının sağlanması amacıyla tasarımcıya iki farklı yöntem sunulmaktadır. Tasarımcı bir alt nesne için, nesnenin özelliklerini göz önünde bulundurarak, *yazma-güncelleme (write-update)* ya da *yazma-geçersiz kılma (write-invalidate)* yöntemlerinden birini seçer. Yazma-güncelleme yöntemine göre, bir alt nesne kopyasının durumu üzerinde değişiklik yapıldığında, diğer kopyalar da aynı şekilde güncellenir. Yazma-geçersiz kılma yöntemine göre ise, tüm diğer kopyalar geçersiz kılınırlar. Daha sonra, geçersiz bir kopyaya sahip bir alandan nesne erişim isteği yapıldığında, son güncel değer ilgili alandan alınmak suretiyle, erişim gerçekleştirilir.

- **Adlandırma ve bağlama hizmeti:** DBNTO sistemi, dağıtılmış bileşik nesnelerin adlandırılması ve yerleşkelerinin bulunması amacıyla da bir yöntem sunmaktadır.

Her bir bileşik nesne kullanıcı tanımlı bir ad ile kaydedilir ve bir bileşik nesneyi kendi adres alanına yüklemek isteyen bir kullanıcı yürüteceği bir arama komutu ile bileşik nesneyi ifade eden kabuk nesneyi kendi adres alanına yükler. Daha sonra yapılan bir metot çağrısı için gerekli olan alt nesne ya da nesnelerin de o adres alanı üzerinde bir kopyası oluşturulur.

- **Dinamik yayılma:** Bir DBN erişim isteğinde bulunulan düğümlere, kendisine sahiplik eden düğümden (gerekli olan alt nesneleri ile birlikte) yürütme sırasında dinamik olarak kopyalanır. Böylece, kullanıcılar uygulamayı çalıştırmaya başlamadan önce, paylaşılan nesnelere ilişkin herhangi bir bilgiyi kendi alanlarına yüklemek zorunda kalmazlar.

- **Dinamik uyarlama:** Büyük ölçekli geniş alana yayılmış olan bir uygulama dinamik olarak yeni bileşenlerin eklenmesi ve var olanların çıkarılabilmesine olanak tanımalıdır. DBNTO, mevcut durumda nesneye bağlantı sağlamış olan istekçileri etkilemeksizin, DBN'ye yeni bir alt nesne eklenmesine, var olanların çıkarılmasına ya da değiştirilmesine imkan tanır.

- **Saydamlık:** DBNTO sisteminde, uygulamanın hem geliştirilme, hem de yürütülme aşamalarında dağıtımın getirdiği problemlerin etkisinin en aza indirilmesi amaçlanmıştır. Dağıtılmış ortamdaki çok sayıdaki kullanıcı tarafından ortaklaşa yürütülen bir uygulama merkezi bir yapıda çalışılıyormuş gibi geliştirilebilir. Dağıtım, senkronizasyon ve tutarlılık ile ilgili konular uygulama kodundan bağımsız olarak çözümlenir. Bu aynı zamanda, mevcut merkezi uygulamaların dağıtılmış ortamda çalışabilmelerini de mümkün kılar.

1.3 Uygulama Alanları

Dağıtılmış bileşik nesne yapısı özellikle internet üzerinden veri paylaşımına izin veren bir çok dağıtılmış sistem uygulamalarında etkinlikle kullanılabilecek bir nesne yapısıdır. Bölüm 1.2'de de ele alınan CSCW uygulamaları bunlardan biridir. Ayrıca, hali hazırdaki mevcut WWW teknolojisinin tekrar yapılandırılarak daha etkin bir şekilde kullanıma sunulabilmesi için de dağıtılmış bileşik nesne modelinden yararlanılabilir. Bu yöntem daha çok sıklıkla güncellenen web sayfalarında kullanılabilir. Sayfa içinde yer alan düz yazı, resim, ses ve video görüntüleri gibi

bilgilerin her birinin ayrı birer alt nesne olarak düşünülüp, bunların bir kabuk nesne içinde bir bileşik nesne yapısında gerçeklenmesi ile sağlanabilir.

1.3.1 CSCW Uygulamalarında Dağıtılmış Bileşik Nesnelerin Kullanımı

Ortaklaşa yürütülmekte olan dağıtılmış uygulamaları destekleyecek bir alt yapı sistemi geliştirme, son yirmi yılda işletim sistemleri ve programlama dilleri üzerine yapılan araştırmaların önemli bir ayağını oluşturmuştur. Özellikle son yıllarda internetin hızlı gelişimi ve çok çeşitli uygulama alanlarında kendini hissettirmesi, yapılan araştırmaların artarak sürmesini sağlamıştır. Nesneye-yönelim kavramının dağıtılmış ortama iyi bir uyum sağlaması dolayısıyla, dağıtılmış paylaşılan uygulama uzayı bir nesneler kümesi olarak düzenlenmeye başlanmıştır.

Dağıtılmış bileşik nesne modeli, tipik CSCW uygulamalarından biri olan dökümanların birbirlerinden uzak kullanıcı grupları arasında eşzamanlı olarak paylaştırılmasına izin veren editör sistemlerinde etkinlikle kullanılabilir.

Çok sayıdaki kullanıcı grubu arasında paylaşılan bir döküman bir bütün olarak görüldüğünde, istek durumunda dökümanın tamamı bir alandan diğerine aktarılacak ve o alan üzerinde yüklenecektir. Ayrıca, her hangi bir anda dökümanın farklı bölümleri üzerinde değişiklik yapmak isteyen kullanıcılardan birinin eşzamanlı erişim denetimi nedeniyle diğerini beklemesi gerekecektir. Bunun yerine, dökümanın bölümlerden, her bir bölümün de alt bölümlerden ve hatta alt bölümlerin de paragraflardan oluştuğu şeklinde parçalı bir yapı düşünülürse, hem istek durumunda aktarılacak veri miktarı azalacak, hem de eşzamanlı erişimde paralellik sağlanmış olacaktır.

Yukarıda ele alınan örnekte her bir bölüm, alt bölüm ya da paragraf bir alt nesne olarak düzenlendiğinde, dökümanın tamamı da bir dağıtılmış bileşik nesne olarak yapılandırılabilir. Bu şekilde hem modüler bir yapı elde edilmiş olur, hem de doküman nesne yapısının sağladığı doğal koruma yapısını kazanmış olur.

Aynı örnek, yine farklı alanlara dağılmış laboratuvarlarda çalışan bir çalışma grubunun üzerinde ortaklaşa çalıştıkları bir bilgisayar destekli mühendislik tasarımı uygulamasına da genişletilebilir.

1.3.2 Mevcut WWW Teknolojisinde Dağıtılmış Bileşik Nesnelerin Kullanımı

Geniş alan ağlarının gelişimi ile birlikte dünyanın internete bakış açısı değişmiş ve internet son on yıl içerisinde günlük yaşamın vazgeçilmez bir parçası haline gelmiştir. İnternetin günlük yaşamın içinde bu denli yayılmasını sağlayan en önemli faktör kuşkusuz WWW teknolojisidir. Fakat, internet ve WWW'in çok hızlı gelişimi ve her alanda kendini göstermesi, beraberinde bazı problemleri de getirmiştir.

1.3.2.1 WWW'de Ortaya Çıkan Problemler

Internet erişiminde WWW kullanımının çok hızlı yükselişi ile birlikte, onun bazı sınırmalarıyla da karşı karşıya kalınmıştır. Problemler özellikle internet üzerinden sayfalara erişme ve sayfa aktarımı sırasında oluşmaktadır. Örneğin, WWW sunucuları hizmet verebilme kapasitesinden daha fazla istek geldiğinde çoğu kez ulaşılamaz duruma düşmektedirler. Aynı şekilde, band genişliği sınırlamaları ve güvenilir olmayan bağlantılar, özellikle resim, ses ve video dosyalarının, yükleme zamanlarının çok uzun olmasına neden olmaktadır. Bu problemlere çözüm olarak, *ön bellekleme (caching)* ve kopyalama gibi geleneksel ölçekleme teknikleri uygulanmaktadır [8]. WWW'de ön bellekleme konusuna çok fazla ilgi gösterilmiştir. Fakat, son zamanlarda ön bellekleme yönteminin de tek başına yeterli olmadığı görülmüş, özellikle güncellemelerin istekçilere otomatik olarak iletildiği kopyalama tekniklerine ihtiyaç olduğu sonucuna ulaşılmıştır [9].

WWW'de kullanılmaya başlanan ön bellekleme ve kopyalama yöntemleri beraberinde tutarlılık problemlerini de getirmiştir. Bir sayfa *ön belleğe (cache)* alındığı veya kopyalandığı zaman, bir kopyası üzerinde yapılan bir değişiklik bu kopyayı tüm diğerlerinden farklı kılmaktadır. WWW'de kullanılan ön bellekleme yönteminde, tutarlılık yönetimi amacıyla benimsenmiş basit bir protokol vardır [10]. Bu protokole göre, her ne zaman bir sayfa ön bellekten bulunup getirildiği zaman, ön bellek öncelikle bu sayfanın sunucuda en son ne zaman güncellenmiş olduğunu kontrol eder. Eğer sayfa istekçi tarafında ön belleğe alındıktan sonra güncellenmiş ise, bu durumda yeni sayfa sunucudan tekrar getirilerek bilgi tazelenir. Aksi durumda, ön bellekte bulunan bilgi kullanılmaya devam edilir.

Daha zayıf bir başka tutarlılık yönetimi şekli de; ön belleğe alınmış olan sayfalar için belli bir eskime süresi verilmesidir. Yalnızca bu süre dolduğunda, ön belleğe alınmış olan sayfa sunucu üzerinden tazelenir. Tabi ki, bu durumda da istekçinin artık eskimiş bir sayfayı bir süre kullanması söz konusudur.

1.3.2.2 WWW İçin Önerilen Çözüm

Ağların ölçeklenebilirliğini arttırmak için, tutarlılık problemlerine çözüm sağlayacak alternatif ön bellekleme ve kopyalama yöntemlerinin geliştirilmesi gerekmektedir. Bir alternatif olarak, WWW için nesne tabanlı bir alt yapının kullanılmasının bu problemleri çözeceği düşünülebilir. Buna göre, web sayfalarında bulunan dökümanlar (düz yazılar, resimler, şekiller, ses ve video görüntü dosyaları, vs.) dağıtılmış bileşik nesne modelindeki alt nesnelere karşı düşen ayrı birer nesne olarak tasarlanacak ve tüm bu nesneler üst seviyede bir taşıyıcı nesne içine konarak kullanıcılara bir bütün olarak sunulacaktır.

Örneğin, WWW'in hali hazırdaki ana uygulamalarından biri olan bir kişiye, projeye, konsorsiyuma ya da bir organizasyona ait web sayfaları incelendiğinde; her birinin bir giriş noktasının olduğu ve birbirlerine *üst-bağlatılarla (hypertext)* bağlanmış çok sayıda sayfadan oluştuğu görülür. Tipik olarak bu yapı DBN modeli ile modellenebilir. Web sayfası, bileşik yapıda bir ağ nesnesi ve tekil olan her bir sayfa da bileşik nesneyi oluşturan bir alt nesne olarak düşünülebilir.

Bu şekilde, WWW nesnesinde yer alan her bir alt nesne için farklı bir kopyalama ve dağıtım politikası da belirlenebileceği gibi, tutarlılık ile ilgili tüm işlemler de nesne düzeyinde halledilecektir. Tüm WWW nesnelerini kapsayacak genel bir tek çözüm bulmak yerine, alt nesneler bazında düzenlenebilen dağıtım ve kopyalama stratejileri oluşturulabilecektir. Bu ayırım çok gereklidir; çünkü, kişisel bir web sayfasına bakış açışıyla, büyük bir konsorsiyumun veya organizasyonun web sayfasına bakış açısı bir olmamalıdır. Bu sayfalardan birine günde binlerce, hatta onbinlerce erişim yapılırken, diğerine belki de günlerce hiç erişim olmayacaktır. Bu durum, farklı dağıtım ve kopyalama politikalarının kullanılmasının gerekliliğini ortaya koymaktadır.

1.4 Tez Planı

Tez kitabının ikinci bölümünde, dağıtılmış nesneye dayalı sistemlerindeki önemli tasarım konuları ve dağıtılmış nesne modelleri üzerine yapılmış çalışmalar; geleneksel süreçler arası haberleşme yöntemlerini kullanan modeller, vekil-tabanlı nesne modelleri ve bölünmüş nesne modelleri başlıkları altında incelenmiş ve her bir modelin değerlendirmesi yapılmıştır.

Üçüncü bölümde, java ortamının kod hareketliliği, çok biçimlilik ve dinamik bağlama, nesne serileştirme, artık toplama, sistem seviyesi kaynaklara erişim, uzaktan yordam çağırma ve dinamik sınıf yükleme konularında tasarım ve gerçekleme aşamalarında sağladığı kolaylıklar anlatılmıştır.

Dördüncü bölümde, tezde önerilen dağıtılmış bileşik nesne modeli; dağıtılmış bileşik nesne, bileşik nesnenin iç yapısı, bir bileşik nesnenin hazırlanması ve bileşik nesne üzerinde bir metot çağrısının yürütülme aşamaları başlıkları altında açıklanmıştır.

Beşinci bölümde, çok sayıdaki alan üzerinde kopyalanmış bulunan alt nesnelerin yönetimi konusu; nesne erişim kategorileri, tutarlılık modelleri ve eşzamanlılık kontrolü ve tutarlılığı sağlama başlıkları altında açıklanmıştır.

Altıncı bölümde, dağıtılmış bileşik nesne yapısına alt sistem desteğini sunan ve kullanıcıyı dağıtılmış sistemin getirdiği zorluk ve ayrıntılardan gizleyen DBNTO sisteminin yapısı açıklanmıştır. Sırasıyla, DBNTO tasarım konuları, dağıtılmış bileşik nesne ortamının genel yapısı ve bu ortamda temel dağıtılmış bileşik nesne işlemlerinin nasıl gerçeklendiği konuları incelenmiştir.

Yedinci bölümde, dağıtılmış bileşik nesne yapısının en önemli elemanları olan bağlantı ve kontrol nesnesi sınıflarının otomatik olarak üretildiği sınıf üretecinin genel yapısı anlatılmıştır.

Sekizinci bölümde, önerilen sistem üzerinde geliştirilen "internet üzerinden müşterek kitap yazma ortamının oluşturulması" uygulaması tanıtılmıştır.

Dokuzuncu bölümde, önerilen DBNTO sistemi üzerinde gerçekleştirilen testler ve bu testlerden elde edilen sonuçlar verilmiştir. Aynı testler Java RMI [22] yapısı üzerinde

de uygulanmıştır. Son olarak, elde edilen test sonuçları karşılaştırılarak, değerlendirmesi yapılmıştır.

Onuncu ve son bölüm olan sonuç ve öneriler bölümünde, geliştirilen dağıtılmış bileşik nesne modeli ve bu modele destek veren arakatman yazılımı olan DBNTO sisteminin sağladığı katkılar anlatılmış ve öneri olarak, gelecekte sistemin daha da geliştirilmesine yönelik yapılabilecek çalışmaların neler olabileceği üzerinde durulmuştur.

2. DAĞITILMIŞ NESNEYE DAYALI SİSTEMLER VE BU KONUDA YAPILMIŞ ÇALIŞMALARIN İNCELENMESİ

Dağıtılmış sistemlerin geliştirilmesine çok elverişli olan nesne modeli özellikle son yirmi yılda büyük önem kazanmıştır. Bu bölümde, dağıtılmış nesneye dayalı sistemlerin tasarım ve gerçekleme aşamalarındaki bazı önemli noktalar kısaca incelendikten sonra, bu tür sistemler üzerine yapılmış çalışmalar ele alınacaktır.

2.1 Dağıtılmış Nesneye Dayalı Sistemlerindeki Önemli Tasarım Konuları

Bir dağıtılmış sistem için en önemli tasarım konusu, sistem üzerinde geliştirilecek olan uygulamalar için kullanımı kolay, esnek yapılı ve çok amaçlı bir ara katman (middleware)'ın nasıl kurulacağıdır. Bu konuda, nesneler doğal yapılarında var olan özellikleri ve dağıtılmış sistemlere uyarlanabilmede göstermiş oldukları başarı ile, önemli bir tasarım alt yapısı sunmaktadırlar.

2.1.1 Nesne Terminolojisi

Nesne ile ilgili kavramlar için düzgün bir terminoloji tanımlamak amacıyla bir çok çalışma yapılmıştır [11]. Fakat, hali hazırdaki durumun hala oldukça karmaşık olduğu söylenebilir. Bu nedenle, kavram karmaşasını ortadan kaldırmak amacıyla bu bölümde tez içerisinde geçecek olan ana terimler kısaca tanımlanacaktır.

Bir *nesne,* içinde veriler ve bu verileri işleyecek olan kod parçalarının bir arada bulunduğu, mantıksal bir varlıktır. Her nesne bir *duruma (state)* ve bir *davranışa (behavior)* sahiptir. Bir nesnenin durumu, nesne içinde tanımlanmış olan veriler ve bu verilerin o anki değerlerinden oluşur. Veriler *örnek değişkenler (instance variables)* olarak da adlandırılırlar. Bir nesnenin davranışı, onun örnek değişkenleri üzerinde yürütmüş olduğu işlemler sonucunda oluşan etkidir.

Nesne durumu üzerinde yürütülen işlemler bir *metotlar* kümesi olarak tanımlanırlar. Nesne içinde tanımlı olan örnek değişkenlere erişmenin ve nesne durumu üzerinde değişiklik yapmanın tek yolu, önceden tanımlanmış olan metotlara çağrılar göndermektir.

Nesneler, metot ve verilerin kullanıcılar tarafından görülebilir olmasını sağlayan bir veya daha fazla sayıda *arayüze (interface)* sahiptirler. Her bir arayüz kendi içinde bir metotlar kümesi içerir. Kullanıcı yalnızca kendisine sunulan arayüz içinde bulunan metotlara çağrılar gönderebilir. Böylece, farklı kullanıcıların farklı nesne metotlarını kullanabilmeleri de sağlanmış olur.

Bir Nesne, *sınıf (class)* adı verilen ve nesne içinde tanımlı örnek değişkenleri ve metot yapılarının tanımlamalarını içeren bir *şablon (template)*'dan türetilir. Bundan dolayı, nesneye türetildiği sınıfın bir *örneği (instance)* olarak da bakılabilir.

Nesneler kullanıldıkları yerlere göre dört ana başlık altında incelenebilirler:

- **Denetim:** Nesneler bir programın normal akışı sırasında bilgi işleme denetim elemanı olarak görev yapabilirler. Program içinde çalışacak olan sistem bir nesne ile gösterilir ve denetim bu nesnenin sunduğu fonksiyonlar ile sağlanır.

- **Uygulama:** Bir sistemde belirli amaçların gerçekleştirilmesi için nesneler kullanılabilir. Çeşitli varlıkların bilgisayar ortamında gösterimi nesnelerle yapılabilir. Bu da daha kolay bir soyutlama ve kapsama sağlar.

- **Veri yönetimi:** Üzerinde işlem yapılacak veriler nesne haline getirilir ve bu nesneler üzerinde yapılan işlemler bir dizi metotların içinde tanımlanır. Nesneye-yönelik veri tabanı yönetim sistemleri buna bir örnektir.

- **Arabirim:** Kullanıcı ile uygulama arasındaki arabirim nesnelerle sağlanabilir. Bu da her iki düzey arasında bağımsızlık yarattığından herhangi birinde yapılacak bir değişiklik veya gelişim diğerini etkileyecek büyük sorunlar çıkarmaz.

2.1.1.1 Nesnelerin Özellikleri

Nesneler doğal yapılarının beraberinde getirdiği bazı özelliklere sahiptirler. Bu özellikler kısaca aşağıdaki gibi özetlenebilir:

Çok biçimlilik (polymophism): Bir ismin birbirleriyle bağlantılı, fakat aslında değişik olan birden fazla amaç için kullanılmasıdır. Çok biçimliliğin amacı, tek bir ismi genel bir işlevler kümesini tanımlamada kullanmaktır. Örneğin bir nesne aynı isim altında birden fazla metot yapısına sahip olabilir. Bu durumda, yürütülecek olan metodu çağrı içinde yer alan parametrelerin tipleri belirler.

Kalıtım (inheritance): Bir nesnenin bir başka nesneye ait özellikleri kullanabilmesi demektir. Bu özellik sınıflandırma kavramını sağladığı için önemlidir. Gerçek hayatta da bir çok sınıflandırma örnekleri görmek mümkündür. Kırmızı otomobil, *otomobil* sınıfından belirli bir nesnedir. Otomobil sınıfı *kara taşıtları* sınıfına ait, o da daha büyük bir sınıf olan *taşıt* sınıfına aittir. Bu örnek sınflandırmada taşıt en üst sınıftır. Kara taşıtları sınıfı, taşıt sınıfının bir alt sınıfı, otomobil sınıfı da kara taşıtlarının bir alt sınıfıdır. Kara taşıtları sınıfı kalıtım yoluyla taşıt sınıfının tüm özelliklerini alır. Ayrıca taşıt sınıfı içinde yer almayan, kara taşıtlarına ait özellikler kara taşıtları sınıfı içinde tanımlanır. Eğer bu sınıflandırma olmasaydı, her nesnenin tüm özelliklerini sınıf tanımlaması içinde açıkça belirtme zorunluluğu ortaya çıkacaktı. Fakat, kalıtım özelliği kullanılarak bir sınıfın sadece kendisini o sınıf içinde farklı kılan özelliklerini belirtmek yeterli olmaktadır.

Kapsama (encapsulation): Veriler ve bu veriler üzerinde işlem yapan program kodunun bir nesne içinde birbirine bağlanmasına *kapsama* denir. Nesne içinde tanımlı verilere tek erişim şekli metotlara yapılan çağrılar ile olduğundan dolayı, kapsama özelliği veriler üzerinde doğal bir koruma sağlar. Nesnenin sınıfını tasarlayan kişi tarafından kasıtlı olarak izin verilmediği sürece, bir kullanıcının doğrudan bir veriye erişip, üzerinde işlem yapma olanağı yoktur.

2.1.2 Dağıtılmış Sistemlerin Yapılandırılmasında Nesne Modelinin Sağlayacağı Yararlar

Elde edilen deneyimler, dağıtılmış sistemler ve üzerlerinde çalışacak olan uygulamaların yapılandırılmasında nesne modelinin oldukça elverişli olduğunu ortaya koymuştur. Bunun nedenleri aşağıdaki şekilde sıralanabilir:

- Nesneler modülerlik ve kuşatma yoluyla veriler üzerinde koruma sağlarlar. Ayrıca, arayüz ile gerçekleme arasında açık bir ayrılmaya da izin verirler. Aynı arayüze sahip, farklı gerçeklemeler oluşturulabilir.

- Nesneler arası haberleşme, metot çağrıları şeklinde, yapısı iyi anlaşılmış ve etkin gerçekleme teknikleri bilinen düzenli bir üst-seviye mekanizma ile halledilmektedir. Örneğin, Java'nın sunduğu *uzaktan metot çağırma (Remote Method Invocation-RMI)* mekanizması, farklı adres alanları üzerinde bulunan nesnelerin metotlarına yerelmiş gibi çağrı yapılabilmesini sağlar.

- Nesneler bilgi paylaşımı için uygun bir yöntem sunarlar. Paylaşılan dosyalara göre avantajları, daha üst seviyedeki semantikleri ve programlama hatalarına karşı daha iyi bir koruma sağlamalarıdır.

- Nesne yapısı temel alınarak kurulmuş olan sistemler, modülerliğinin sağladığı avantaj sayesinde, yalnızca gerekli yeni kısımların eklenmesi suretiyle kolaylıkla geliştirilebilirler. Ayrıca, tekil olan nesnelerin ya da alt sistemlerin, *kalıtım (inheritance)* mekanizması ile kolaylıkla yeniden kullanımları sağlanarak, yeni işletimlere ya da çevresel şartlara uyarlaması sağlanabilir.

- Eğer nesne modeli bir programlama dili ile destekleniyor ise, arayüzlerin statik kontrolü güvenliği sağlar. Çünkü, bir çok potansiyel çalışma-zamanı hataları derleme zamanında ortaya çıkmaktadır.

2.1.3 Farklı Nesne Yapıları

Bu bölümde kısaca, dağıtılmış nesneye dayalı sistemler için destek mekanizmaları hazırlayan tasarımcının karşılaştığı ana problemlere değinilecektir. Nesne alt sisteminin tasarımcısı aşağıda özetlendiği şekilde bir çok önemli kararlar vermek zorundadır:

- *Aktif ya da pasif nesneler:* Aktif nesneler kendi başlarına, bağımsız işlemsel varlıklardır. Diğer bir deyişle, nesne ve süreç yapılarının her ikisine de sahip olan varlıklardır. Pasif nesneler ise, yalnızca bir (dağıtılmış, görüntü) bellek organizasyonunu tanımlarlar; onlara erişen aktif varlıklar ayrı olarak tanımlanırlar. Bu konu taneciklik ile ilişkilidir. Aktif nesneler tipik olarak büyük,

yükü ağır sunucular için daha uygun iken, pasif nesneler ince-taneli veriler için daha uygundur.

- *Paylaşılan (shared) ya da tek-kullanıcılı (private) nesneler:* Paylaşılan nesneler düzgün bir haberleşme ortamı sağlarlar. Ancak, paylaşılma genellikle koruma ve senkronizasyon mekanizmalarını gerektirir. Paylaşılan ve tek-kullanıcılı nesneler sistemde genellikle aynı anda bulunurlar.

- *Kendi başına bağımsız (self-confined), bileşik (composite) ya da parçalı (fragmented) nesneler:* Kavramsal olarak, bir nesne tekil bir varlık olarak görülmektedir. Ancak, nesnenin iç yapısı diğer nesneleri içermesini gerektirebilir. İç bileşim mekanizmasının tanımlaması ve iç yapıdaki bileşimin görülebilirliğinin derecesi önemli tasarım konularındandır.

- *Tekil ya da kopyalanmış nesneler:* Bu konu nesnelere özel bir konu değildir. Kullanılabilirliği ve başarımı arttırma, hata hoşgörüsünün yükseltilmesi gibi sebeplerden dolayı bir nesnenin değişik alanlar üzerinde kopyası çıkartılabilir. Ancak, bir nesne modeli kopyalamayı düzenli bir arayüzün arkasına gizleyebildiği ölçüde kendini kullanışlı bir model olarak sunabilir.

- *Nesne tanecikliği - iri, orta, küçük (Object granularity – coarse, medium, small):* Nesne tanecikliği doğrudan kullanılabilirlik ve başarım ile ilişkili olduğu için en önemli tasarım konularından biridir. Tecrübeler programcıların küçük nesneleri (bir kaç yüz sekizli uzunluktaki) kullanma eğiliminde olduklarını göstermiştir. Diğer yandan, temel sistem mekanizmaları (eşleştirme, haberleşme, paylaşma) için büyük nesneler daha hızlı ve verimlidir.

- *Geçici (temporary) ya da sürekli (persistent) nesneler:* Sürekli nesnelerin avantajları programlama yapılarının (değişkenler, kayıtlar vb.) ve dosyaların birleştirilmesinde düzenli bir görüş sağlamasıdır. Diğer yandan, dağıtılmış sürekli nesneler çoğunlukla artık nesne toplamadan kaynaklanan bazı güç problemlerin ortaya çıkmasına sebep olmaktadır.

- *Sabit (fixed) ya da hareketli (mobile) nesneler:* Nesne hareketliliği yük dengeleme, başarımı yükseltme ya da yönetimsel ve gizlilik ile ilgili bazı özel

kısıtlamalar için kullanışlıdır. Fakat, nesne hareketliliğine izin verilen sistemlerde nesnenin yerleşkesini bulma süreci oldukça karışık bir yapıya sahiptir.

- *Statik ya da dinamik bağlanma (binding):* Nesne yapısının önemli özelliklerden biri tekil bir arayüzün çoklu gerçeklemelerinin sunulmasındaki esneklik ve alt tip uyumudur. Ancak, bu esneklik uygun olan metodu çağrı anında seçme gibi dinamik bağlanma mekanizmalarının varlığını gerektirir.

Dağıtılmış nesneye dayalı sistemlerin tasarım alanı yukarıda söz edilen konulardan çok daha geniştir. Son yılardaki araştırma projelerinde bir çok varyasyonlar geliştirilmiş olmakla birlikte, üzerinde hala bir çok araştırmaların devam ettiği bir çalışma alanıdır.

2.1.4 Dağıtılmış Sistemlerdeki Nesne Yönetim Problemleri

Bu bölümde, bir dağıtılmış nesne destek sisteminin çözmesi gereken ana problemler kısaca özetlenecektir:

- *Adlandırma (Naming):* Nesne adlandırma birkaç seviyede gerçeklenen temel bir fonksiyondur. Tipik olarak, simgesel (kullanıcı seviyesi) adlar, sonuçta kendiliklerinden fiziksel adreslere çözülecek olan alt seviyedeki iç adlara karşılık düşürülür. İç adlar nesneler arası referanslarda kullanılan adlardır ve koruma ve gizlilik konularında da sisteme katkı sağlarlar. Yerel adlar verilen bağlama özel iken, evrensel adlar tüm çalışma uzayında ve zamanda tekildirler. Evrensel adlar uygun bir yapıda ölçeklenemezler ve kullanılmakta olan mevcut adlandırma sistemlerine birleştirilmeleri de oldukça zordur. Önerilen çözümler [12]'de olduğu gibi yerel olarak yorumlanan, fakat uzak nesnelere etkin bir şekilde referansta bulunabilme imkanı tanıyan, bir bağlantılar zincirini içerir.

- *Yerleşke Bulma (Location Lookup):* Nesnenin yerleşkesini bulma problemi, nesne adından yola çıkarak bulunduğu yerin belirlenmesi işlemidir. Eğer ad "saf" ise, yani içinde yerleşke ile ilgili hiç bir bilgi içermiyor ise, geniş bir sistem içerisinde nesnenin yerleşkesinin etkin bir şekilde bulunması oldukça güç bir işlemdir. Yayın yöntemi ile bulma, geniş ölçekli bir sistemde, maliyetinin yüksekliğinden dolayı belki de en son olarak düşünülecek olan bir yöntemdir. *İleri doğru işaretçiler (forwarding pointers)* ya da ön bellekleme gibi yardımcı teknikler,

nesnelerin hareketlilik derecelerinin incelenmesi veya gözlemlenmesi sonucunda elde edilecek bilgilere dayanılarak seçilmesi durumunda faydalı olabilirler.

- *Erişim:* Bir nesneye erişme ve o nesne üzerinde bir metot çağrısı yürütme şu işlemler zincirini içerir: nesneyi bul, eğer derleme-zamanında yapılmamışsa tip uyumluluğunu kontrol et yapılacak çağrının bağlamını belirle (yerel/uzak, koruma için seçilen yönteme göre, yük-dengeleme, vb.), eğer henüz bağlanmamışsa nesneyi (kod ve veri) bu bağlama bağla, eğer dinamik metot bağlama uygulanıyorsa çağrı yapılacak olan metodu belirle ve son olarak çağrıyı yürüt. Eğer nesnenin bir kopyası çağrının yapıldığı alan üzerinde fiziksel olarak bulunmuyor ise, son adımdan bir nesne hatası ile geri dönülebilir. Hata, üzerinde çalışılan işletim sisteminin uygun bir mekanizması tarafından halledilir (dağıtılmış görüntü bellek gibi).

- *Paylaşma ve Koruma (sharing and protection):* Nesne paylaşımı, mesaj iletimine göre daha üst seviyede bir soyutlama sağladığından dolayı haberleşme için doğal bir model olarak görülebilir. Ardışıl (yani eşzamanlı olmayan) paylaşım sürekli nesneler ile sağlanabilir. Eşzamanlı paylaşım eşzamanlılık kontrolü için bazı mekanizmalar içermelidir. Paylaşım birçok faklı şekilde gerçeklenebilir: Tüm çağrıların kendisine yönlendirildiği nesnenin tekil kopyasını kullanarak; nesneyi çağrının yapıldığı alana göç ettirerek; her bir çağrı için, çağrı yapılan bağlamda nesnenin bir kopyasını oluşturarak (bu durumda farklı nesne kopyaları arasında tutarlılığı sağlayacak bir mekanizma gerekecektir); "parçalı nesneler" formunda (her bir parça kendi alanındaki nesne üzerinde bir arayüz gibi davranacaktır).

- *Süreklilik (Persistence:*. Bu başlık altında genellikle birbirinde faklı iki konu ima edilir. Bunlardan ilki, nesnenin yaşam süresinin, onu kullanmakta olan süreçlerinkinden bağımsız olmasıdır. Diğeri ise, uzun süreli saklama ortamında (disk gibi) nesnenin bir kopyasının var olmasıdır. Sürekliliğin olağan tanımlaması, önceden tanımlanmış bir düğüm üzerinden erişilebilirliğidir. *Dayanıklılık (durability)* sürekliliğin sağlanmasında daima en basit yoldur. Bir sürekli nesne destek sistemi geçici ve kalıcı saklama ortamı üzerindeki referans gösterilimleri problemini çözebilmelidir. Süreklilik ile ilgili bir diğer önemli konu *artık toplama (garbage collection)*'dır. Artık toplamada kullanılan

geleneksel yöntemler iyi bir şekilde ölçeklenememektedirler. Bu amaçla, esneklik ve güvenlik arasında bir denge sağlanmalıdır.

2.2 Dağıtılmış Nesne Modelleri Üzerine Yapılmış Çalışmalar

Nesne tabanlı dağıtılmış sistemlerin tasarım ve gerçeklemesi üzerine yapılmış çok sayıda akademik ve ticari çalışmalar bulunmaktadır. Var olan modellerin ve sistemlerin önemli özelliklerini tanımlamak amacıyla aşağıdaki gibi bir sınıflandırma yapmak konunun anlaşılması açısından faydalı olacaktır.

- *Geleneksel süreçler arası haberleşme (interprocess communication-IPC) yöntemlerini kullanan modeller:* Bu modeller dağıtılmış sistemlerin önemli bir sınıfını oluşturmaktadırlar. Modele göre, dağıtılmış ortamdaki süreçler mesaj alış-verişi yoluyla ya da paylaşılan bir veri üzerindeki işlemler aracılığıyla haberleşirler.

- *Vekil-tabanlı (proxy-based) nesne modelleri:* Son yıllarda geliştirilen ve iyi bilinen bir çok sistem bu modeli benimsemiştir. Bu modelde, nesne tek bir makine üzerine yerleştirilir. Bu nesnenin herhangi bir metoduna çağrıda bulunma, istekçi tarafına bir vekil atamak suretiyle, ağ üzerinden saydam olarak yapılır.

- *Bölünmüş (partitioned) nesne modelleri:* Bu modelde, nesnenin bir çok makine üzerinde fiziksel olarak dağıtılmış olduğu kabul edilir. Fakat, kullanıcılar dağıtımın farkında değildirler. Nesne kullanıcılara bir bütün olarak görünür. Dağıtılmış olan parçaların başarım ve kullanılabilirliği arttırma amacıyla değişik alanlar üzerinde kopyalanması da mümkündür.

Konunun ilerleyen bölümlerinde yukarıda sözü edilen üç farklı model sınıfına giren ve literatürde kabul görmüş sistemleri inceleyerek, her bir modelin değerlendirmesi yapılacaktır.

2.2.1 Geleneksel IPC Modelleri Üzerine Yapılmış Çalışmalar

Geleneksel IPC modelleri üzerine yapılmış olan çalışmalar daha çok nesneye-yönelim konusunun dağıtılmış sistemler için henüz düşünülmediği seksenli yıllarda

geliştirilen sistemlerde görülmektedir. Bu sistemlerden en çok bilinenleri aşağıda sırasıyla incelenmiştir.

2.2.1.1 PVM

Paralel Sanal Makine (Parallel Virtual Machine-PVM) [13] basit, fakat etkin bir şekilde büyük, paralel sistemlerin geliştirilebileceği birleşik bir iskelet yapı oluşturmak için ortaya atılmış bir projedir. Bu projenin genel olarak amacı; bir ağ içinde toplanmış heterojen yapıdaki makinelerin genel amaçlı bir hesaplama sistemi olarak görülmesini sağlamaktır.

PVM sistemi, hali hazırda var olan programlama dillerinin içine eklenebilir bir dizi kullanıcı arayüzü ilkelleri sunar. Süreçlere çağrıda bulunma, mesaj iletimi ve mesajı yanıtlama, *yayın (broadcasting)*, bariyerler üzerinden senkronizasyon, *karşılıklı dışlama (mutual exclusion)* ve paylaşılan bellek için ilkeller mevcuttur. Süreçler senkron ve asenkron olarak başlatılabilirler. Diğer süreçlerin başlatılma ve sonlanma durumlarına veya verilerin değerlerinin aldıkları duruma göre ayarlanabilirler.

2.2.1.2 MPI

Mesaj iletimi, paralel bilgisayarlarda, özellikle de dağıtılmış bellekli ölçeklenebilir paralel bilgisayarlardan ve iş istasyonlarında oluşan ağlarda geniş bir kullanım alanına sahip bir programlama şeklidir. *Mesaj İletim Arayüzü (Message Passing Interface-MPI)* [14] standardı da, mesaj iletim yeteneklerinin geniş bir bölümü için kullanıcı arayüzü ve işlevsellik sağlamak amacıyla geliştirilmiştir. MPI'nın ana hedefi; bir çok standartlar gibi, farklı makineler arasında taşınabilirlik derecesinin arttırılmasıdır. Taşınabilirlik derecesi için umulan, mesaj iletim kaynak kodunun MPI kütüphanesi var olduğu sürece bir çok farklı makine üzerinde çalıştırılabilir olmasıdır. Tabi ki, sistemlerin temel özelliklerinden dolayı en iyi başarımı yakalayabilmek için sistemde bazı ayarlamalar yapmak gerekebilir. Mesaj iletimi çoğunlukla dağıtılmış bellekli paralel bilgisayarlar ortamı için düşünülmüş olmasına karşın, aynı kod paylaşılan bellekli bir paralel bilgisayar üzerinde de iyi bir şekilde çalışabilir. Yine, iş istasyonlarından oluşan bir ağda ve gerekirse tek bir iş istasyonu üzerinde çalışan süreçler kümesinde de kullanılabilir.

2.2.1.3 DCE

İstekçi/sunucu hesaplama yapısının kullanılmaya başlanmasıyla birlikte, bu konuda problemler de ortaya çıkmaya başlamıştır. Bu problemlerden en büyüğü; çok sayıdaki istekçi/sunucu uygulamalarının geliştirilebileceği ve genel bir dağıtılmış hizmetler kümesi sunan genel bir alt yapının eksikliğidir. Bu şekildeki bir istekçi/sunucu yapısı yaratmak için organizasyon tümüyle dağıtılmış uygulamalar ve bu uygulamalar için geliştirilen genel bir tesis gerektiren bir yapı etrafında kurulmuş olmalıdır. Ayrıca, bir çok ortam değişik ürün satıcılarının teknolojilerini içerdiğinden, bu alt yapı tüm sistemler üzerinde iyi bir şekilde çalışabilmelidir.

Yukarıda sözü edilen genel alt yapının *oluşturulması Dağıtılmış Hesaplama Ortamı (Distributed Computing Environment-DCE)*'nin asıl amacıdır [2]. DCE dağıtılmış uygulamaları desteklemek için gereken anahtar hizmetleri sağlamaktadır. Bu hizmetler:

- İstekçiler ve sunucular arasında *uzaktan yordam çağırma (Remote Procedure Call-RPC)*'ya destek verme.
- İstekçilerin sunucuları bulmalarını sağlayacak bir *katalog (directory)* hizmeti sunma.
- İstekçiler ve sunucular arasında güvenli erişimi sağlamak amacıyla güvenilir hizmetler sunma.

DCE, teknolojinin değişen şartlarına ayak uydurmak konusunda iddialı olduğu için, son yıllarda giderek önem kazanan nesneye-yönelim teknolojisinin kullanımına büyük önem vermiştir. Bu amaçla, DCE dağıtılmış nesne hesaplamaları için büyük bir destek tabanı sağlar.

2.2.1.4 NFS

Ağ Dosya Sistemi (Network File System-NFS) [15], yerel alan ağları üzerinden uzak dosyalara erişimi sağlayan bir yazılım sistemidir. En önemli tasarım amacı; farklı makineler, işletim sistemleri ve ağ mimarilerinden oluşan heterojen bir ortamda dosya hizmeti verebilmektir.

NFS birbirine bağlı iş istasyonlarını, bağımsız dosya sistemli bir bağımsız makineler kümesi olarak görür. Amacı, bu dosya sistemleri arasında saydam bir şekilde ve belli bir derecede paylaşımı sağlamaktır. Paylaşım istekçi/sunucu ilişkisine dayanır. Bir makine hem istekçi, hem de sunucu olabilir. Paylaşım sadece seçilmiş sunucu makineleri arasında değil, herhangi bir makine çifti arasında sağlanabilir. Bir uzak dosya sisteminin NFS paylaşımı yalnızca istekçi makineyi etkiler. Bu yüzden global bir paylaşılan dosya sistemi mantığına ihtiyaç yoktur.

2.2.1.5 Geleneksel IPC Modellerinin Değerlendirmesi

Bir çok dağıtılmış sistem mesajlaşma yoluyla ya da paylaşılan veri üzerindeki işlemle aracılığıyla süreçlerin haberleşmesi modelini destekler. Mesaj iletim modelleri düşük seviyede bir soyutlama sağlıyor olmalarına rağmen, iyi ölçeklenebilen sistemler olduklarından dolayı popülerliklerini hala devam ettirmektedirler. PVM ve MPI gibi haberleşme kütüphanelerine ve DCE gibi RPC temeline dayanan dağıtılmış hesaplama uygulamaları, doğrudan TCP ve UDP gerçeklemelerine dayanmaktadırlar. Ancak, mesaj iletimi yoluyla dağıtılmış uygulamalar geliştirmek zordur. En önemli sınırlama verinin yerleşim, kopyalama, tutarlılık ve sürekliliğinin yönetimi gibi önemli gerçekleme konularının uygulamaya bırakılıyor olmasıdır. Yani, her uygulama bu mekanizmaları kendileri tasarlamalı ve gerçeklemelidir. Her bir uygulamanın kendi alt yapısını kurduğu bir ortamda yeni uygulamaların geliştirilip, diğerleri ile entegrasyonunun sağlanması da son derece güç olacak, hatta belki de mümkün olmayacaktır.

Mesaj iletimine dayanan sistemlerin ardından, paylaşılan veri üzerinde işlem yürütme esasına dayanan yeni bir prosesler arası haberleşme modeli ortaya çıkmıştır. Bu modelin tipik bir örneği NFS *ve dağıtılmış paylaşılan bellek (Distributed Shared Memory-DSM)* gerçeklemeleridir. Paylaşılan veri modeli, sekizlilerin okunup yazılması için küçük bir ilkeller kümesi sunmaktadır. Buradaki ana problem, verinin tutarlılığını korurken, aynı zamanda başarım ve ölçeklenebilirliği de sağlayabilmektir. Örneğin, paylaşılan dosyaların başarımı düşürmeden özel kilit yöntemleri kullanılarak korunması ihtiyacı olabilir. Aynı şekilde, DSM sistemleri gevşek bellek tutarlılığı gerektirebilir. Sonuç, anlaşılması çok güç bir haberleşme modelidir. Ayrıca, sayfa-tabanlı DSM sistemleri bilgi gizleme konusunda bir desteğe

sahip değillerdir. Yine, ciddi başarım problemleri vardır ve asla çok geniş alana ölçeklenemezler.

2.2.2 Vekil Tabanlı Nesne Modelleri Üzerine Yapılmış Çalışmalar

Son yıllarda geliştirilen nesne-tabanlı dağıtılmış sistemler içerisinde vekil-tabanlı nesne modelleri önemli bir yer tutmaktadır. Daha önce de belirtildiği gibi bu modelde, nesne tek bir düğüm üzerine yerleştirilir. Nesnenin herhangi bir metoduna çağrıda bulunma, çağrının gerçekleştirileceği istekçi tarafına bir vekil atamak suretiyle, ağ üzerinden saydam olarak yapılır. Bu modeli benimsemiş yaygın olarak bilinen sistemler aşağıda sırasıyla incelenmiştir.

2.2.2.1 Spring

Spring, deneysel olarak geliştirilmiş, son derece modüler, dağıtılmış, nesneye-yönelik bir işletim sistemidir [16]. Adlandırma, sayfalama, ağ giriş/çıkışı ve dosya sistemleri gibi hizmetler kullanıcı seviyesi sunucular olarak gerçeklenmiştir. Bu sunucular yönettikleri kaynaklar için nesneye-yönelik arayüzler sağlarlar ve istekçiler bu nesnelere ait metotlara çağrıda bulunarak istem sunucularıyla iletişimde bulunurlar.

Spring dağıtılmış bir sistem olduğundan dolayı, bir düğüm üzerinde bulunan tüm hizmetler ve nesneler, dağıtılmış sistemdeki tüm diğer düğümler için de kullanılabilir hizmetler ve nesnelerdir.

Spring, nesneleri diğer nesneye-yönelik sistemlerden biraz farklı görmektedir. Bir çok dağıtılmış sistem nesnelerin sunucu makinelerine yerleştirildiği ve istekçi makinelerinin de sunucu makinelerindeki bu nesnelere işaret eden nesne referanslarına sahip olduğu modeli kullanır. Spring ise, istekçilerin doğrudan nesne üzerinde işlem yaptığı bir modeli benimsemiştir. Spring, sunucu tabanlı olmayan ve durumu istekçilerle sunucular arasında dağıtılmış bulunan nesneleri de destekler. Bu şekilde, istekçinin sadece bir referansa sahip olmak yerine, gerçek nesneye sahip olması çok daha uygun olmaktadır.

2.2.2.2 Emerald

Emerald nesne tabanlı bir dağıtılmış programlama dilidir ve ana hedefi de, dağıtılmış programların geliştirilebilmesi için kolay kullanımlı bir ortam sunmaktır [17]. Emerald'da süreç ve veri kavramları nesnelerin içine alınmışlardır ve nesneler dağıtımın temel üniteleridir.

Emerald ile birlikte, bir yerel alan ağında dağıtılmış programlar geliştirebilmek için kolaylıklar sağlayan bir *yürütme zamanı (runtime)* sistemi de tasarlanmıştır. Emerald'in getirdiği yenilikler üç madde içerisinde özetlenebilir. Bunlar:

- Küçük ve büyük programlamalar için kullanılabilecek tek bir nesne modeli sunması.
- *Soyut (abstract)* tipler için destek vermesi.
- Nesne yerleşkesi ve *nesne hareketliliği (object mobility)* konusunda bir fikir vermesi.

Emerald'da tüm bilgiler nesneler içinde muhafaza edilir. Tüm nesneler aynı nesne tanımlama mekanizması kullanılarak kodlanır. Emerald'in nesne modeli tamsayılar, karakterler gibi küçük nesneler için olduğu gibi kataloglar ve derleyiciler gibi büyük nesneler için de uygun bir modeldir.

Emerald'da programcılar nesne yerleşimin kavramından yararlanmak veya bunu göz ardı etmek konusunda serbesttirler. Dağıtılmış bir sistemde nesnelere yerleşkeden bağımsız olarak çağrı yapılabilmelidir. Bu kolaylık ağ hizmetlerini kolayca erişilebilir kılar. Emerald'da bunu benimsemiştir ve bu yüzden, yapılan bir çağrının hedefinin bulunması tamamen sistemin sorumluluğundadır. Uygulamalar istedikleri taktirde nesnelerin yerleşkelerini kontrol edebilirken, bir çok uygulama bu fikri tercih etmez. Çünkü,yerel ve uzak çağrıların semantiği aynıdır. Tabii ki, başarım ve kullanılabilirlik açısından bakıldığında, yerleşkenin kullanıcı tarafından görülebilir olmasının faydaları vardır.

Emerald'in önemli bir özelliği de sınırlandırılmamış nesne hareketliliğinin sağlanmış olmasıdır. Hareketlilik, dağıtılmış sistemin başarımını arttırmak amacıyla kullanılabilecek bir yöntemdir. Emerald'da bir nesne bir başka alana taşındığında,

nesne tanımlayıcısına nesnenin yeni yerine işaret eden bir *ileri doğru işaretçi (forwarding pointer)* bırakılır. Böylece, eski yerleşkeye yapılan çağrılar yeni yerleşkeye yönlendirilir.

2.2.2.3 Shadows

Shadows'un tasarım ve prototip gerçeklemesi, yerel olarak dağıtılmış, çok işlemcili sistemler üzerinde çalışan, nesneye-yönelik ve hata-hoşgörülü bir programlama sistemi olan Arjuna'dan elde edilen deneyim çalışmaların sonucu olarak ortaya çıkmıştır [18].

Shadows mimarisi, istekçi tarafından istekte bulunulan işlemlerin uzak düğümde sunucu süreçler tarafından nesnelere erişim ile sağlandığı istekçi/sunucu modeline dayanır. İstekçi ve sunucu arasındaki haberleşme RPC formundadır.

Shadows'un ana amacı; nesnelere yerleşkeden-bağımsız erişim ve sistem içinde nesnelerin hareketliliğinin sağlanmasıdır. Sistemin tasarımcıları bu amaçların süreklilik, paylaşılabilirlik ve eşzamanlılık kontrolü gibi bir çok önemli nesne özelliklerinin desteklenmesine temel teşkil edecek yeterlikte olduğunu düşünmüşlerdir.

Herhangi bir düğüm üzerinde bulunan bir nesne sunucusu, istekçilere o düğüm üzerinde bulunan bir nesneler kümesine erişim imkanı sağlar. Sunucular bir ön-işlemci tarafından otomatik olarak üretilirler. Bu amaçla, ön-işlemci istekçi sınıf tanımlayıcısını okur ve ilgili bir sunucu sınıfını üretir. Sunucular aktif nesnelerdir. Her bir sunucu nesnesi sistem çapında tekil olan bir sunucu kimliğine sahiptir. Sunuculara erişim için bu kimliklerin istekçiler tarafından biliniyor olması gerekmektedir. Ayrıca, bir sunucu tarafından yönetilen her bir nesne, o sunucu tarafından bilinen bir ada sahiptir. Sunucu kimliği ve nesne adının birleşimi nesnenin sistem kimliği olarak bilinir ve nesnenin sistem içinde tekil olarak tanımlanabilmesi için yeterli bir bilgidir.

Shadows'un önemli bir özelliği yerleşkeye saydam işlem çağrısı yapabilmesidir. Bu mekanizmayı kullanarak işlem çağrıları yerel olarak ya da eğer nesne uzak bir nesne ise, o nesneyi yöneten sunucuya RPC yoluyla istek yapılmak suretiyle yürütülür ve bu işlem istekçiye saydamdır.

2.2.2.4 DCOM

Dağıtılmış Bileşen Nesne Modeli (Distributed Component Object Model-DCOM), nesneye-yönelik RPC'ler için bir uygulama seviyesi protokolüdür [1] ve bundan dolayı *nesne RPC (Object RPC-ORPC)* olarak adlandırılır. Protokol, DCE'nin RPC tanımlamaları üzerine kurulan bir dizi eklemelerden ibarettir. ORPC;

- Nesne üzerinde çağrıların nasıl yapılacağını,
- Nesne referanslarını nasıl gösterildiğini, birbirleri ile nasıl haberleştiğini ve devamının nasıl sağlandığını tanımlar.

DCOM protokolünün tasarım amacı; herhangi bir dağıtılmış uygulamaya, haberleşme protokolünün gerektirdiği standart özelliklerin yapısında var olan desteğin sağlanmasıdır. Diğer bir deyişle, DCOM dağıtılmış uygulamalar arasındaki göreve-özel haberleşme yollarının kurulmasını kolaylaştıran bir iskelet yapı olarak davranır.

2.2.2.5 CORBA

Genel Nesne İstek Aracı Mimarisi (Common Object Request Broker Architecture-CORBA) [3], günümüzün çok hızla gelişmekte olan donanım ve yazılım ürünleri arasındaki birlikte çalışabilme ihtiyacı için *Nesne Yönetim Grubu (Object Management Group-OMG)* tarafından üretilmiş olan bir standarttır. CORBA bir uygulamanın diğeriyle onun nerede bulunduğu ve kim tarafından tasarlandığı gibi bilgilere ihtiyaç duymaksızın, haberleşmesine izin verir.

CORBA ile, kullanıcılar bilgiye saydam olarak erişme imkanı kazanırlar. Burada saydamlıktan kasıt; nesnenin hangi yazılım ya da donanım platformunda bulunduğu ve yerleşkesinin ne olduğu gibi bilgilerin kullanıcıyı ilgilendirmiyor olmasıdır.

CORBA'ya bu özelliğini kazandıran, *Nesne İstek Aracısı (Object Request Broker-ORB)*'dır. ORB nesneler arasında istekçi/sunucu ilişkilerini kuran bir arayüzdür. ORB'u kullanarak, bir sunucu nesnesi üzerinde bulunan bir nesneye saydam olarak çağrı yapılabilir. Öyle ki, burada sözü edilen sunucu nesnesi aynı makine üzerinde olmak zorunda değildir, ağın herhangi bir bölümünde de olabilir. Yapılan çağrıyı alan ORB, isteği gerçekleyecek olan nesneyi bulmakla sorumludur. ORB

parametreleri nesneye aktarır, metoda çağrıda bulunur ve sonuçları geri döndürür. Bu arada, istekçi nesnenin yerini, onun programlama dilini, işletim sistemini ya da nesnenin arayüzünün bir parçası olmayan diğer sistem bilgilerini bilmek zorunda değildir. ORB bu görevi yerine getirirken, heterojen dağıtılmış ortamlardaki farklı makineler üzerindeki uygulamalar arasındaki birlikte çalışabilmeyi sağlar ve problemsiz bir şekilde çoklu nesne sistemlerini birbirine bağlar.

ORB, isteklerin hedef nesne gerçeklemeleri ile saydam olarak haberleşebilmelerini sağlayacak bir mekanizma sunar. ORB istekçiyi nesneye çağrıda bulunma işlemlerinin ayrıntılarından kurtararak, dağıtılmış programlamayı basitleştirir. Aynı zamanda, ORB istekçilerin yapmış oldukları çağrıların yerel yordam çağrıları gibi görünmelerini sağlar. İstekçi bir nesneye çağrıda bulunduğunda, ORB nesnenin bulunmasından, eğer gerekiyorsa nesnenin saydam bir şekilde aktif kılınmasından, isteği nesneye ulaştırmaktan ve yanıtı geri döndürmekten sorumludur.

2.2.2.6 ILU

Diller Arası Birleşme (Inter Language Unification-ILU) sistemi, bir *çoklu-dil (multi-language)* nesne arayüzü sistemidir [19]. ILU ile sağlanan nesne arayüzleri farklı diller, farklı adres alanları ve işletim sistemleri arasındaki gerçekleme ayrımlarını gizler. ILU iyi tanımlanmış ve herhangi bir dile ait olmayan arayüzler sayesinde, çoklu-dile ait nesneye-yönelik sınıf kütüphaneleri geliştirmek için kullanılabilir. ILU aynı zamanda, dağıtılmış sistem gerçeklemeleri için de kullanılabilir.

ILU program parçalarının modül olarak adlandırılan üniteleri arasındaki arayüzler ile ilgilidir. ILU modül için nesneye-yönelik bir arayüz yazılmasını mümkün kılan bir yöntem sağlar. ILU modüller arasında bir çok farklı bağlanma ilişkilerine izin verir. Modüller, tümü aynı dilde yazılmış bir programın parçaları olabileceği gibi, bir bellekteki çalışma zamanı desteğini paylaşan farklı dillerde yazılmış parçalar veya farklı makineler üzerindeki farklı programların parçaları olabilirler.

ILU'da kullanılan yaklaşım standart RPC yapısıdır. Bir arayüz herhangi bir dile ait olmayan *Arayüz Tanımlama Dili (Interface Specification Language-ISL)* ile tanımlanır. ILU nesneye-yönelik olduğundan, modüller birer nesnedir. Modülden dış dünyaya sunulan bir fonksiyon da o nesnenin metodudur.

2.2.2.7 Legion

Legion dağıtılmış ortamda çalışan kullanıcılara tek bir sanal makine görüntüsü sağlamak amacıyla geliştirilmiş bir projedir [20]. Bu sanal makine güvenli paylaşılan nesneler ve paylaşılan ad uzayları, uygulamaya göre düzenlenebilir hata-toleransı, hızlandırılmış yanıt süresi ve daha büyük *çıktı (throughput)* oranı sağlar.

Legion, yerel alan ağı, geniş alan ağı ve ulusal bilgi alt yapısı ile birbirine bağlanmış iş istasyonları, vektör süper bilgisayarları ve paralel süper bilgisayarlardan oluşmaktadır. Bu şekilde birbirine bağlı makinelerin hesaplama gücünün çok büyük olacağı açıktır.

Legion nesneye-yönelik bir sistemdir. Nesneye-yönelimin prensipleri Legion'un geliştirilmesinde esas olarak alınmıştır. Tüm kullanıcıların ihtiyaçlarına cevap verebilecek bir sistem değil, genişleyebilir bir sistem tasarlanmıştır. Örneğin, güvenlik konusu düşünüldüğünde, güvenlik ve başarım arasında bir denge kurulmuştur (onaylama, şifreleme vb. maliyetinden dolayı). Sabit bir seviyede güvenlik sağlama yerine, kullanıcıların kendi yöntemlerini gerçeklemeleri veya var olanları kullanmaları konusunda bir denge sağlanmıştır.

2.2.2.8 Globus

Globus, coğrafik olarak dağıtılmış hesaplama kaynaklarının bütünleştirilmesi için gereken temel alt yapıyı geliştirmek için ortaya atılmış bir projedir [21]. Bu tip hesaplamalar çok sayıdaki farklı bölgeye ve değişik yeteneklere sahip ağlara yerleştirilmiş onlarca ve hatta yüzlerce kaynağın bağlantısını gerektirebilir. Mevcut sistemler sadece yeni kaynakların tanımlanması ve sisteme entegrasyonunu sağlarken, Globus ulusal ölçekli ağlarda kullanılabilir olan kaynakların dinamik olarak tanımlanması ve bir araya getirilmesini destekleyen bir programlama ortamı ortaya koymuştur.

Hesaplama ve bilgi kaynaklarının dinamik olarak bir araya getirilmelerini desteklemek için, kaynak yerleşimi ile protokol ve kaynak yönetimi konuları üzerinde, güvenlik alanında ise, veri erişimi ile *asıllama (authentication)* ve *yetkilendirme (authorization)* konuları üzerinde durulmuştur.

Yukarıda bahsedilen temel teknikler başarımı yüksek, dağıtılmış ve ölçeklenebilir hesaplama ortamlarının geliştirilebileceği bir prototip yazılımda birleştirilmiştir.

2.2.2.9 Vekil Tabanlı Nesne Modellerinin Değerlendirmesi

Bölüm 2.2.1'de bahsedilen IPC-tabanlı modellere alternatif olarak, nesne tabanlı yaklaşımlar daha çok benimsenmiştir. Nesneler yalnızca büyük ölçekli uygulamalar oluşturmayı kolaylaştırmakla kalmaz, aynı zamanda dağıtılmış uygulamalar için iyi bir mimari modelini de beraberinde getirirler. Nesneler özellikle *ince-taneli (fine-grained)* hizmet sağlayıcıları olarak görülebilirler. Vekil-tabanlı sistemlere *Uzaktan Metot Çağırma (Remote Method Invocation-RMI)* [22], RPC'deki teknik kullanılarak gerçeklenebilir. Ancak, bu yaklaşım vekil-tabanlı modellerin dünya çapına ölçeklenebilmelerinin de ana engellerinden biridir. Bu yüzden, nesne kopyalama ve asenkron metot çağırma gibi ek mekanizmalara ihtiyaç vardır.

En basit şekliyle RMI, geleneksel RPC yapısı gibi gerçeklenebilir. Örneğin, DCOM'da bu yöntem kullanılmıştır. DCOM'da istekçiler için nesnelerden çok, uzak sunucular tarafından gerçeklenen arayüzler sunulur.

Nesneleri açıkça destekleyen yaklaşımlar daha ilginçtir. Örneğin Legion sisteminde, nesneler farklı adres alanlarına yerleştirilmişlerdir ve metot çağrısı doğrudan mesaj iletimi ile gerçeklenir. Bu durum dağıtım saydamlığını bozmasına rağmen, Legion yaklaşımı geniş alana ölçeklenebilmeyi sağlayan ender yaklaşımlardan biridir. Bu model nesnelerin nasıl gerçekleneceği konusunda bir kural koymaz. Sadece, onların nasıl etkileşimde bulunacağını belirtir. Globus projesi esnek gerçeklemeleri desteklemek amacıyla global işaretçiler geliştirmiştir. Global işaretçi uzaktaki bir nesneye bir referanstır. İşaretçi, istekçi tarafından seçilen bir çok nesne ile haberleşme protokollerini tanımlar. Dağıtım saydamlığı henüz sağlanamamış olmasına rağmen, global işaretçiler Legion yaklaşımından daha üst derecede esneklik sağlar.

Dağıtım saydamlığı en açık şekliyle ORB'larda görülmektedir. ORB, nesneler ile onların istekçileri arasında bir ara bulucudur. Nesne bir sunucunun adres alanı içinde yer alır ve istekçiler metotlara kendi adres alanlarında bulunan bir vekil nesnesi aracılığıyla çağrıda bulunurlar. Temelde ORB'lar, ILU'daki gibi sadece dilden

bağımsız ve yerleşkeye saydam metot çağrılarını destekler. CORBA'ya ait olan ORB'lar adlandırma, süreklilik ve *hareket (transaction)* gibi ilave dağıtım hizmetleri de sunarlar. Fakat ne yazık ki, CORBA henüz nesnelerin saydam olarak kopyalanması ve kopyalar arasındaki tutarlılığın sağlanması için gerekli hizmetler tanımlamamıştır. Oysa, CORBA'daki nesneler kendi kopyalama politikalarını oluşturabilirler. Ayrıca, hali hazırdaki hizmetlerin ne derecede ölçeklenebilir olarak gerçekleneceği de çok açık değildir.

ORB dağıtım hizmetlerinden sorumlu olduğunda, ek olarak çekirdek nesne modelinin bağımsız mekanizmalarına gereksinim duyulur. Bu şekil bir mekanizma Spring sisteminde kullanılan *alt-sözleşme (subcontract)* [23] mekanizmasıdır. Alt-sözleşme bir çağrı protokolünü gerçekler; istekçi tarafındaki bir metot çağrısının etkisini nesne tarafındaki metot çağrısı ya da çağrıları şeklinde tanımlar. Örneğin kopyalanma durumunda, bir istekçi tarafından yapılan metot çağrısı, her bir kopyanın metodunun çağrılması şeklinde gerçeklenir. Çağrının kopyalanması alt-sözleşme yapısının içinde gerçeklenir ve istekçiden gizlidir. Ancak, genel bir mekanizma olarak, alt-sözleşme yapısı oldukça sınırlıdır. Örneğin, çok sayıdaki istekçi tarafından paylaşılan nesneler grubunun tutarlılığın alt-sözleşmeler ile sağlanması zordur. Ayrıca, Spring geniş alana yayılmış bir sistem olarak da tasarlanmamıştır.

2.2.3 Bölünmüş Nesne Modelleri Üzerine Yapılmış Çalışmalar

Dağıtılmış ortamda çok sayıdaki istekçinin paylaşımlı olarak kullandığı, özellikle iri-taneli nesnelerin taşınabilirlik, başarım ve kullanılabilirliğini arttırmak amacıyla, daha küçük parçalara bölünmesi, fakat kullanıcılarına yine tek bir nesne görüntüsü altında sunulması fikri, özellikle son yıllar içerisinde ortaya atılmış değişik bir yaklaşım olarak göze çarpmaktadır.

Bölünmüş nesne modeli henüz yeni bir konu olduğu için, kısıtlı sayıda mevcut ve halen geliştirilmekte olan birkaç sistem bu modeli benimsemişlerdir. Aşağıda bu sistemler incelenmiştir.

2.2.3.1 Parçalı Nesneler

Parçalı Nesneler (Fragmented Objects-FO) modeli nesne kavramını dağıtılmış ortama değişik bir yapıda taşımıştır [24]. Genel olarak bakıldığında, bir FO tekil ve dağıtımı istekçilerden gizlenmiş paylaşılan bir nesnedir.

FO tasarımcıları, paylaşılan bir nesnenin veri ve/veya kodunun başarım, kullanılabilirlik, koruma ve yük dengeleme amacıyla parçalanarak, farklı alanlar üzerine dağıtılmasına ihtiyaç duyulabileceğini belirtmişlerdir. Bu parçalar bir bütün olarak, çok sayıdaki istekçi nesneleri tarafından paylaşılan basit mantıksal bir varlık oluştururlar.

Bir FO'nun dıştan (istekçilerin bakış açısına göre) ve içten (tasarımcısının bakış açısına göre) olmak üzere iki görünüşü vardır. Dışarıdan bakıldığında, FO tekil, kullanıcı tanımlı bir arayüz üzerinden erişilen, paylaşılan bir nesne olarak görülür. Nesneyi oluşturan parçalar ve özellikle de parçaların dağıtımı görülemez. İçten bakıldığında ise, FO birbirleriyle işbirliği yapan çok sayıdaki parçanın bir araya gelmesinden oluşmaktadır. Her bir parça merkezi yapıdaki gibi normal bir nesnedir. Parçalar haberleşme nesneleri olarak bilinen daha alt seviyedeki FO'lar kullanılarak birbirleriyle haberleşirler. FO tasarımcısı, nesnenin dağıtımı, en uygun haberleşme protokolünün seçimi, FO'nun semantiğine en uygun hatadan kurtulma yönteminin seçimi gibi, nesnenin karakteristik özellikleri üzerinde tam bir denetime sahiptir.

İstekçi/sunucu ve paylaşılan nesne modelleri gibi, FO modeli de istekçileri için dağıtım saydamlığını destekler. Geleneksel modelden farklı olarak, FO'lar dağıtım saydamlığını tasarımcılarının üzerine yüklemezler. Bu amaçla, bir dağıtım mekanizması da tasarlanmıştır.

2.2.3.2 Globe

Globe projesinde [25] yer alan dağıtılmış paylaşılan nesneler, parçalı nesnelerin bir diğer gerçeklemesini oluşturmaktadır. Büyük ölçekli ve geniş alana yayılmış uygulamaların gelecekteki gelişimini destekleyebilmek için yeni ve iddialı bir sistem geliştirilmiştir. Globe tasarımcıları, ortaya koydukları modelin dünya çapına yayılmış milyonlarca kullanıcının her birinin yüzlerce hatta binlerce nesnesine destek verecek

bir alt yapıya sahip olduğunu iddia etmektedirler. Globe var olan ağ işletim sistemlerinin tümü üzerinde problemsiz bir şekilde çalışabilecek bir yapıdır.

Globe'un sunduğu nesne modeli aynı anda çok sayıdaki süreç tarafından paylaşılan dünya çapına ölçeklenebilir bir yapıya izin vermektedir. Nesneler herhangi bir programlama diline bağımlı değildirler. Nesneler kullanıcıları için her biri bir metotlar kümesi içeren bir ya da birden fazla arayüz sunarlar. Nesneler pasiftir, etkinlik süreçler tarafından sağlanır. Bir süreç tarafından nesne durumunda yapılan bir değişiklik, diğerleri tarafından görülebilir.

Globe nesnesi, aynı FO modelinde olduğu gibi, fiziksel olarak dağıtılmıştır. Diğer bir deyişle, nesnenin durumu aynı anda çok sayıdaki alan üzerine dağıtılmış ve hatta kopyalanmış olabilir. Ancak, süreçler bu dağıtımın farkında değildir. Ayrıca, haberleşme protokolleri, kopyalanma stratejileri ve nesne durumunun dağıtımı ve göç ettirilmesi gibi gerçekleme ile ilgili tüm detaylar nesnenin bir parçasıdır ve onun sunduğu arayüzlerin arkasına gizlenmiştir.

Bir sürecin herhangi bir nesnenin bir metoduna çağrıda bulunabilmesi için, sürecin öncelikle o nesnenin *bağlantı noktalarından (contact points)* biri ile ilişki kurarak, nesneyi kendi adres alanına bağlaması gerekir. Bağlanma, nesnenin ilgili arayüzünün istekçinin adres alanına yerleştirilmesi ile sona erer.

Globe modelinde, nesne adları nesnelerin yerleşkeleri ile ilgili bir bilgi içermezler. Adres çözümleme işlemi iki aşamalıdır. Bir adlandırma, bir de yerleşke bulma sunucusu vardır. Bir nesneye erişmek için, nesnenin adı öncelikle adlandırma sunucusuna aktarılır. Adlandırma sunucusu geriye yerleşkeden bağımsız ve global olarak tekil bir *nesne bağı (object handle)* döndürür. Bu nesne bağı bir nesne referansı olarak süreçler arasında aktarılabilir. Bu nesne bağı yerleşke bulma sunucusuna aktarıldığında, sunucu geriye o nesnenin dağıtılmış kopyalarına ait bir dizi bağlantı adresi döndürür. Daha sonra, döndürülen bağlantı adreslerinden en uygun olanı üzerinden nesne çağrının yapıldığı adres alanına bağlanır. Bu aşamada iyi çalışan bir algoritmanın en uygun bağlantı adresini seçmesi, sistemin başarımını doğrudan etkileyecek olan çok önemli bir ayrıntıdır [26].

Yukarıda açıklandığı şekilde, addan nesneye kurulan iki aşamalı köprü sayesinde, nesnenin yerleşkesi ile ilgili tüm detaylar ad içerisinden çıkartılmış olur. Bu sayede

nesne adı üzerinde hiçbir değişiklik yapılmaksızın, serbestçe bir başka alan üzerinde kopyalanabilir veya göç ettirilebilir. Bu durum, sadece yerleşke bulma sunucusunda o nesneye işaret eden nesne bağına ait bilgilerin güncellenmesini gerektirir. Hatta bu yapı ile aynı nesneye birden fazla ad ile de erişim sağlanabilir.

2.2.3.3 DUPLEX

DUPLEX kullanıcıların internet üzerinden bağlantı kurdukları bir *müşterek yazışma ortamı (collaborative editing environment)*'dır [27]. Amacı; bir dokümanın birbirinden bağımsız parçalara bölünüp, her birinin diğerlerinden ayrık olarak ele alınmasını sağlayan bir model sunmaktır. Böyle bir uygulamada Adlandırma büyük önem kazanmaktadır. Çünkü nesneler (dokümanı oluşturan parçalar) büyük ve heterojen yapılı bir ağ üzerinde dağıtılmış ve kopyalanmıştır. Bu amaçla, DUPLEX için, *Düşük Seviyeli Adlandırma Hizmeti (Light Weight Name Service-LWNS)* olarak adlandırılan özel bir adlandırma hizmeti de tasarlanmıştır [28].

Nesneler büyük ve heterojen yapılı bir ağ üzerinde dağıtılıp, kopyalandığından dolayı, nesne yapısı oldukça karmaşıktır. Kopyaların listesinin tutulması, tutarlılığın sağlanması için kullanılan protokol gibi konular önem kazanmaktadır.

2.2.3.4 AspectIX

AspectIX, dağıtılmış nesneler için iyi bir hizmet kalitesi sağlayan ve farklı amaçlar için uyarlanabilen bir mimaridir [29]. AspectIX'de nesneler istekçilerin farklı ihtiyaçlarına ve sistemde oluşan farklı koşullara göre adapte edilebilirler. Hata hoş-görüsü, güvenlik ve ölçeklenebilirlik gibi davranışlar sunan kalite gerçeklemeleri ile nesnenin işlevsel gerçeklemesi arttırılabilir.

AspectIX CORBA uyumlu olacak şekilde tasarlanmıştır. Fakat, AspectIX CORBA'dan daha esnek bir açık ORB mimarisidir. CORBA uyumlu olmakla birlikte, ORB ve uygulamalar arasında genişletilmiş arayüzler kullanır. Böylece, sıradan CORBA nesneleri AspectIX ORB'u üzerine taşınabilmekte ve bir AspectIX gerçeklemesi CORBA uygulamalarına sunuculuk yapabilmektedir.

AspectIX, aynı nesnenin farklı parçalarının farklı adres alanları üzerinde bulunduğu ve kullanıcıların dağıtılmış nesneye ait kendi yerel parçalarına kendi adres alanları

içinde sahip olduğu, bölünmüş nesne modelini benimsemiştir. AspectIX'te bir dağıtılmış nesneyi oluşturan yerel parçalar kullanıcılardan saydam olarak diğer yerel parçalarla haberleşebilmektedirler.

Dağıtılmış nesnenin bir bölümünü oluşturan bir yerel parça, gelen çağrıları uzaktaki bir diğer parçaya yönlendiren basit bir nesne olabileceği gibi, dağıtılmış nesnenin durumunun bir bölümüne sahip ve işlevsellik sunan bir nesne de olabilir.

AspectIX'in tasarımcılarına göre, dağıtılmış sistem kullanıcıları için sundukları nesne yapısı, özellikle geleneksel *hareketli etmenler (mobile agents)* uygulamaları için uygun bir yapı oluşturmaktadır. AspectIX nesneleri tabanlı etmenler, geleneksel etmenlere oranla daha esnek bir yapı sunmaktadır. Özellikle, göç etme maliyetinin çok yüksek olduğu çok büyük etmenlerin gerçeklenmesinde AspectIX nesne yapısının oldukça kullanışlı olduğu iddia edilmektedir.

2.2.3.5 Bölünmüş Nesne Modellerinin Değerlendirmesi

Genel dağıtım hizmetleri geliştirmeye bir alternatif olarak sunulan bölünmüş nesne modelinde, nesnenin bölünme şekli, kopyalanması, göç ettirilmesi gibi nesnenin dağıtımı ile ilgili tüm ayrıntılar onun gerçeklemesinin bir parçasıdır ve nesnenin sunduğu arayüzün arkasına gizlenmiştir. Bölünmüş nesneler ilk olarak SOS işletim sisteminde [30] FO modeli ile ortaya atılmıştır. Globe projesindeki dağıtılmış paylaşılan nesneler bölünmüş nesnelerin bir diğer gerçeklemesini oluşturur. DUPLEX'te ise, müşterek bir grup çalışmasının sağlanması amacıyla parçalanan bir doküman dağıtılmış bir nesne olarak ele alınır. Fakat, DUPLEX'te geniş alana bir dağıtım sağlanmış olmasına rağmen, yöntemin ilgilendiği nesneler sayıca azdır ve yalnızca bir dokümanın paylaşımı söz konusudur. Bu amaçla, yalnızca uygulamaya özel bir yapı geliştirilmiştir. Ayrıca, ilgilenilen nesne sayısı az olduğu için adlandırma yapısı basittir.

Bölünmüş nesne modelleri arasında FO ve Globe'un dağıtılmış paylaşılan nesneleri iddialı iki modeldir. Bu iki model arasındaki en önemli farklılık, FO modelinin geniş alana yayılmış ağlar için tasarlanmamış olmasıdır. FO için temel olarak kullanılan haberleşme nesneleri yalnızca yerel alan ağları için tasarlanmış ve gerçeklenmiştir. Ayrıca, FO modelinde nesneye özel kopyalama stratejileri için de bir yöntem

sunulmamıştır. Bu model farklı tutarlılık politikalarının gerçeklenmesi yönünde bir yaklaşım sunmamakta ve platform heterojenliği konusunu da ele almamaktadır. Fakat, bölünmüş nesne modelini ilk olarak ortaya atan bir yaklaşım olduğu için, diğer yaklaşımlara ilham kaynağı olmuştur.

Globe, programlama dilinden bağımsız olan nesnelerinin alt parçalara ayrılması, alt parçaların kopyalanması, her biri için farklı kopyalama stratejilerinin belirlenebilmesi, nesne durumunun göç ettirilebilmesi gibi her türlü dağıtım politikalarının nesne bazında düzenlenebildiği yapısı ile mevcut nesneye-yönelik dağıtılmış sistem perspektifine radikal bir değişim getirmiştir. Alt tabakada bulunan ve dünya çapına dağılmış milyonlarca kullanıcının binlerce nesnesine hizmet verebilecek kapasitede olduğunu iddia ettikleri adlandırma ve yerleşke bulma hizmetiyle de diğer yaklaşımlardan ayrılmaktadır.

AspectIX, CORBA uyumlu ve bir dağıtılmış nesnenin farklı adres alanlarına yüklenmiş yerel parçalardan oluştuğu yapıyı destekleyen bir ORB sunar. AspectIX'te, nesneler istekçilerin ihtiyaçları doğrultusunda ve değişen sistem koşullarına göre uyarlanabilir olma özelliğine sahiptir. Globe nesne modeli içinde bulunan yerel nesnenin farklı dağıtım, kopyalama, tutarlılık politikaları sunan gerçeklemelere izin veren yapısı itibariyla iki model birbirine oldukça benzemekle beraber, AspectIX'in CORBA uyumlu olması ve hatta CORBA uygulamalarına sunuculuk yapabilmesi gibi özellikleri de vardır.

2.3 Dağıtılmış Bileşik Nesne Modeli ile Bölünmüş Nesne Modelinin Karşılaştırılması

Dağıtılmış bileşik nesne modeli ile bölünmüş nesne modeli karşılaştırıldığında, bir çok açıdan benzerlikler içermekle birlikte, bazı yönlerden birbirlerinden ayrılmaktadırlar. İki model arasındaki benzerlik ve farklılıklar şu şekilde sıralanabilir.

- *Geniş alan uygulamaları için destek sağlama:* Her iki model de geniş alana yayılmış ve müşterek yürütülen uygulamaları destekleyecek bir alt yapı sunmaktadır. Yalnızca, [24] ile sunulan FO projesi bir yerel alan ağında çalışacak şekilde tasarlanmıştır.

- *Kopyalama:* Her iki model de belli ölçülerde nesne kopyalama yöntemini desteklemektedirler. Tüm nesne erişimleri yerel kopyalar üzerinden yapılır. Ancak, DBN modelinde tüm yerel kopyalar birbirleri ile aynı iken, örneğin Globe nesne modelinde, bir yerel kopya alt nesneye ait bir durum bilgisi içermeyip, tüm çağrıları bir uzak-kopyaya ileten bir vekil gibi davranabilmektedir.

- *Adlandırma ve yerleşke bulma hizmetleri:* Bölünmüş nesne modeli sınıfına giren FO ve DUPLEX'in sunduğu adlandırma ve yerleşke bulma hizmetinin yapısı, DBN modeli tarafından sunulan yapı ile benzerlikler göstermektedir. Üçünde de yönetilen nesne sayınının çok fazla olmadığı (binler mertebesinde) kabul edilir. Diğer yandan, Globe projesinin sunduğu yapı, dünya çapına dağılmış milyonlarca kullanıcının her birinin on binlerce nesnesini destekleyecek şekilde tasarlanmıştır.

- *Nesne erişim şekli:* DBN modelinde, yeni yaratılan bir bileşik nesne kullanıcı tanımlı bir ad ile *kaydedildikten (registration)* sonra, bir başka alanda çalışan uzak kullanıcıya ait bir uygulama tarafından bir *arama (lookup)* komutu yürütüldüğünde, o alana alt nesneleri olmaksızın yalnızca kabuk nesne kopyalanır. Daha sonra yürütülecek metot çağrıları ile gerekli alt nesneler de ilgili alan üzerinde kopyalanırlar. Yani, alt nesnelerin varlığı kabuk nesne içine gizlenmiştir. Bölünmüş nesne modeli sınıfına giren projelerde ise, erişim doğrudan alt nesneleredir ve kullanıcı erişeceği alt nesneye ait bir arayüz tablosu aracılığıyla metot çağrılarını gerçekleştirir. Bundan dolayı, DBN modeli daha üst seviyede bir soyutlama sağlamaktadır. Öyle ki, bu yapıda kullanıcı dağıtılmış ve parçalı bir nesneye erişiyor olduğunu dahi bilmiyor olabilir.

- *Dinamik yayılma:* DBN modelinde bir bileşik nesne tek bir alan üzerinde tüm alt nesneleri ile birlikte yaratılır ve zaman içerisinde diğer alanlardan gelen istekler ile dinamik olarak yayılarak dağıtılmış bileşik nesne görüntüsünü kazanır. Bölünmüş nesne modelinde ise, bir nesne farklı alanlar üzerinde yaratılabilir.

- *Dinamik uyarlama*: Büyük ölçekli geniş alana yayılmış olan bir uygulama dinamik olarak yeni bileşenlerin eklenmesi ve var olanların çıkarılabilmesine olanak tanımalıdır. DBNTO, mevcut durumda nesneye bağlantı sağlamış olan istekçileri etkilemeksizin, DBN modeli bir bileşik nesneye yürütme zamanında

yeni bir alt nesne eklenmesine, var olanların çıkarılmasına ya da değiştirilmesine imkan tanır. Fakat, bölünmüş nesne modeli sınıfına giren çalışmalar üzerinde yapılan incelemelerde böyle bir özelliğe rastlanılmamıştır.

- *Saydamlık:* DBN modelinde, dağıtılmış ortamda bulunan çok sayıdaki kullanıcı tarafından ortaklaşa yürütülen bir uygulama merkezi bir yapıda çalışılıyormuş gibi geliştirilebilmektedir. Alt nesnelerin dağıtımı, senkronizasyon ve tutarlılık ile ilgili konular uygulama kodundan bağımsız olarak çözümlenmektedir. Aynı yapı, Globe ve FO projelerinde de görülmektedir. Bölünmüş bir nesnenin parçaları için ayrı ayrı düzenlenebilen dağıtım, senkronizasyon ve tutarlılık yöntemlerini gerçekleyecek alt seviye nesneler bir kütüphaneden elde edilebilmektedir.

3. JAVA ORTAMININ SAĞLADIĞI KOLAYLIKLAR

Bu bölümde DBNTO sisteminin geliştirilmesinde neden Java ortamının seçildiği ve Java programlama dilinin sistemin geliştirilmesi aşamalarında sağladığı kolaylıklar anlatılacaktır.

Bilindiği gibi Java, C++ benzeri, platformdan bağımsız, hemen hemen tüm makine tipleri üzerinde çalışabilen, nesneye-yönelik bir programlama dilidir. Java'nın oldukça popüler ve iyi bilinen bir programlama dili olmasından dolayı, burada sadece DBNTO sisteminin geliştirilmesi sırasında yararlanılan belirgin özelliklerinden söz edilecektir.

3.1 Kod Hareketliliği

Java'nın anahtar özelliklerinden biri *kod hareketliliği (code mobility)*'dir. Java sınıfların uzaktaki düğümlerden dinamik olarak yüklenebilmesine imkan tanır. Kodun bu şekildeki hareketliliği, hata yayılmasını önlemek için kod taşınabilirliği ve güvenlik ile ilgili bazı yaptırımların uygulamalarını gerektirir. Kod taşınabilirliği *sekizli-kod (byte code)*'un yorumlanması suretiyle sağlanır. *javac* derleyicisi makine kodu üretmez, bunun yerine tüm donanımlardan bağımsız genel bir kod olan sekizli-kodu üretir. Üretilen sekizli-kod yürütme sırasında Java'nın o anda kullanılmakta olan yürütme-zamanı sistemine uygun olarak yorumlanır. Güvenlik, Java dilinin sunduğu güvenlik mekanizması tarafından uygulanır. Java programın adres alanına doğrudan erişime izin vermez. Nesneler *işaretçiler (pointers)* üzerinden kullanılamaz. Bunun yerine, dil seviyesi referanslar kullanılır. Bir program bir Java referansını, ya bir nesne yaratma işlemi ya da bir nesne çağrı parametresinin dönüş değeri ile elde edebilir.

Java'nın kod hareketliliği *ağ (web)*'da geniş bir kullanım alanı bulmaktadır. Bir çok *ağ tarayıcısı (web browser)* bir *Java Sanal Makinesi (Java Virtual Machine-JVM)* içerir ve HTML sayfaları da Applet olarak adlandırılan bazı Java programlarına

referanslar içerebilirler. HTML sayfası bir web tarayıcısı aracılığıyla yüklendiğinde, Java programı ağ tarayıcısı içine gömülü bulunan Java makinesi tarafından çalıştırılır.

3.2 Çok Biçimlilik ve Dinamik Bağlama

Java'nın kullanılan bir diğer önemli özelliği *çok biçimlilik (polymorphism)*'tir. Çok biçimlilik, arayüzlerin (veya tiplerin) ve sınıfların birbirlerinden ayrı olarak tanımlanabilme yeteneğinin olmasıdır. Arayüz bir sınıfın metotlarına ait bir imzadır. Arayüzde yer alan her bir imza içinde; metodun adı, parametre tanımlamaları ve metodun döndürdüğü değerin tip tanımlaması yer alır. Arayüz herhangi bir gerçeklemeye bağlı değildir ve bu yüzden bir çok farklı sınıflar ile gerçeklenebilir.

Java, kod hareketliliği için büyük önem arz eden *dinamik bağlanma (dynamic binding)* özelliğine de sahiptir. Dinamik bağlanma, bir metot çağrısı için yürütülecek olan kodun çalışma zamanında belirlenebilmesi yeteneğidir. Java sınıfların dinamik olarak yüklenebilmelerine izin verdiği için, arayüz tipindeki bir değişkene dinamik olarak yüklenmiş olan bir sınıfa ait bir nesneye işaret eden bir referans atanabilir. Java, (bu değişkenin) kodunun bağlanması işlemini çağrı zamanına kadar geciktirir. Böylece, dinamik olarak yüklenen sınıfların çalıştırılabilmelerine imkan sağlamış olur.

3.3 Nesne Serileştirme

Java nesnelerin farklı çalışma sistemleri arasında aktarılabilmelerine izin veren nesne *serileştirme (serialization)* özelliğine sahiptir. Bu özellik bir nesneler grafının serileştirilerek bir sekizli katarına dönüştürülmek suretiyle ağ üzerinden bir diğer JVM'ye mesaj olarak gönderilebilmesini ya da disk üzerinde bir dosyanın içine kaydedilebilmesini sağlar. Mesajın alıcısı ya da dosyayı okuyan, serileştirilmiş olan sekizli katarından *ters-serileştirme (deserialization)* işlemi ile tekrar aynı nesneler grafını oluşturur. Bu işlem sonunda graf içindeki Java referansları değişmekle birlikte, grafın yapısı korunur.

Serileştirilebilir (serializable) arayüzünü gerçekleyen bir sınıfın tüm örnekleri serileştirilebilir nesnelerdir. Bu amaçla, serileştirme ve ters serileştirme için *writeObject* ve *readObject* şeklinde iki metot tanımlanmıştır. Serileştirme işleminin kontrol altında tutabilmek için, bu metotları önemsemeyerek, nesnenin hangi alanlarının aktarılacağı ve nesnenin tekrar nasıl oluşturulacağı tanımlanabilir. Bu, örneğin bir nesne serileştirilirken nesne içinden referansta bulunulan diğer nesnelerin serleştirilmemelerini sağlamak amacıyla kullanılabilir.

Serileştirilebilir bir nesne içinden referansta bulunulan tüm diğer nesneler de serileştirilirler (bu nesneler serileştirilebilir arayüzünü gerçeklemelidir).

3.4 Artık Toplama

Java yürütme-zamanı ortamında yalnızca erişilmekte olan nesnelerin tutulmasını, hiç bir referansı olmayan nesnelerin bellekten atılmasını sağlayan *artık toplama (garbage collection)* mekanizması vardır. Sanal makine üzerinde çalışan proseslerden en az biri tarafından erişilen bir nesne Java çalışma-zamanı ortamında muhafaza edilir. Bir nesne artık erişilmez duruma geldiğinde, o nesne için tahsis edilen tüm kaynaklar (özellikle bellek) yürütme-zamanı sistemi tarafından geri alınır ve diğer nesneler tarafından tekrar kullanılabilir hale getirilir. Artık toplama işlemi kullanıcı programlarına saydam, yalnızca nesne referanslarına bağlı olarak yürütülür.

3.5 Sistem Seviyesi Kaynaklara Erişim

Java, dosyalar ve ağ bağlantıları gibi işletim sistemi seviyesinde yönetilen kaynaklara erişim imkanı sağlar. Özellikle, Java bir programın dosyaları açıp, okuyup, yazmalarına ve bir serileştirmenin sonucunun dosyada saklanmasına ve böylece *sürekli* Java nesnelerinin disk üzerinde yönetilmelerine imkan tanır. Bir web sunucusu üzerinde sorgulama yapabilmek için, uzak bir makinenin *iskele (port)*'sinin hem soket arayüzü, hem de HTTP protokolü üzerinden bağlamak için de sınıflar tanımlanmıştır.

3.6 Uzaktan Yordam Çağırma

Java, kullanıcılarına *uzaktan yordam çağırma (RMI)* olarak adlandırılan ve uzaktaki bir Java sanal makinesi üzerinde nesne çağrısı yapabilme imkanı tanıyan bir hizmet sunmaktadır [22]. Bir sınıf tanımlanırken bu sınıfa ait nesnelere uzaktan da çağrı yapılabileceğini belirtmek amacıyla *uzak sınıf (remote class)* olarak tanımlanır. Bir adlandırma sunucusu bir uzak nesne referansının bir alan üzerinden diğer alanlardaki uygulamalara ihraç edilebilmesini sağlar. Daha sonra, herhangi bir makine üzerindeki bir Java programı bu referansı alarak, uzaktaki nesne üzerinde bir metot çağrısı yürütebilir. Aynı şekilde, bu çağrının geri dönüş parametresi olarak istekçiye diğer uzak nesnelere ait referanslar da döndürülebilir.

RMI'nın önemli bir özelliği, bir uzak sınıfa ait arayüzün, hem uzak nesnelere, hem de serileştirilebilir nesnelere ait Java referansı içerebilmesidir. Bir uzak nesne referansının iletilmesi, o referansa ait nesneye ait metotların herhangi bir diğer makineden çağrılabileceğini gösterir; bu durumda nesne referans ile aktarılmıştır demektir. Bir serileştirilebilir nesne referansının aktarılması RMI'nın nesnenin bir kopyasını elde etmesine izin verir; bu da nesnenin değer ile aktarıldığı anlamına gelir.

RMI'nın sunduğu bir başka yenilik ise, belki de dünyanın birbirinden oldukça uzak iki farklı noktasındaki makineler üzerinde çalışan Java uygulamalarının normal metot çağrılarını kullanarak birbirleriyle haberleşebilmelerinin sağlanmasıdır.

RMI'nın bir diğer önemli özelliği de, haberleşme için üst seviyede, programatik bir arayüz sunmasıdır. Metot tabanlı arayüzler ile, uzaktaki bir nesne üzerinde metot çağrıları yerel çağrılar gibi yürütülebilmektedir. RMI, soket yapısına göre bir çok açıdan daha kullanışlı ve doğaldır. Ancak, ağ bağlantısının her iki ucunda da Java'nın çalışıyor olması gerekmektedir.

3.7 Dinamik Sınıf Yükleme

Java platformunun kayda değer yeteneklerinden biri de; Java yazılımlarının Java sanal makinesine farklı bir fiziksel sistem üzerinde çalışan herhangi bir URL'den dinamik olarak yüklenebilmesidir. Sonuç olarak, uzaktaki bir sistem üzerinde bir program, örneğin bir *applet*, yerel diske yüklenilmeden çalıştırılabilmektedir.

Java RMI, bu yeteneğin sağladığı avantajı kullanarak, yerel disk üzerinde bulunmayan sınıfların uzaktaki bir alandan dinamik olarak yüklenebilmesini ve çalıştırılabilmesini sağlayacak, *dinamik sınıf yükleme (dynamic class downloading)* mekanizmasını geliştirmiştir. RMI uygulama programı arayüzü ve herhangi bir JVM'yi kullanarak, sadece web tarayıcılarıyla değil, aynı zamanda bir java programı içinden de herhangi bir uzaktaki bir makinede bulunan bir sınıfının yükleme işlemi gerçekleştirilebilir.

Java programlama dilinde bulunan *sınıf-yükleyici (classloader)*'lerinin kullanımından *kod-tabanı (codebase)* fikri ortaya çıkmıştır. Kod-tabanı, bir JVM'ye sınıf yükleme işleminin nereden yapılacağını gösteren alan ya da kaynak olarak tanımlanabilir. Bir makinede tanımlanan bir *sınıf-yolu (classpath)*'da bir nevi "yerel kod-tabanı" olarak adlandırılabilir. Çünkü, sınıf-yolu yerel sınıfların disk üzerinde nerelerden yüklenebileceğini gösteren bir listedir. Yerel disk-tabanlı bir kaynaktan sınıf yükleneceği zaman, sınıf-yolu değişkenine başvurulur. Bir Java programı bir sınıf-yükleyici kullandığında, o sınıf yükleyicisi sınıfları yüklemesine izin verilen kod-tabanını bilme ihtiyacını duyar. Genellikle, sınıf yükleyicileri Java platformu için derlenmiş olan sınıflara hizmet veren bir HTTP sunucusu ile birlikte kullanılırlar.

4. DAĞITILMIŞ BİLEŞİK NESNE MODELİ

4.1 Giriş

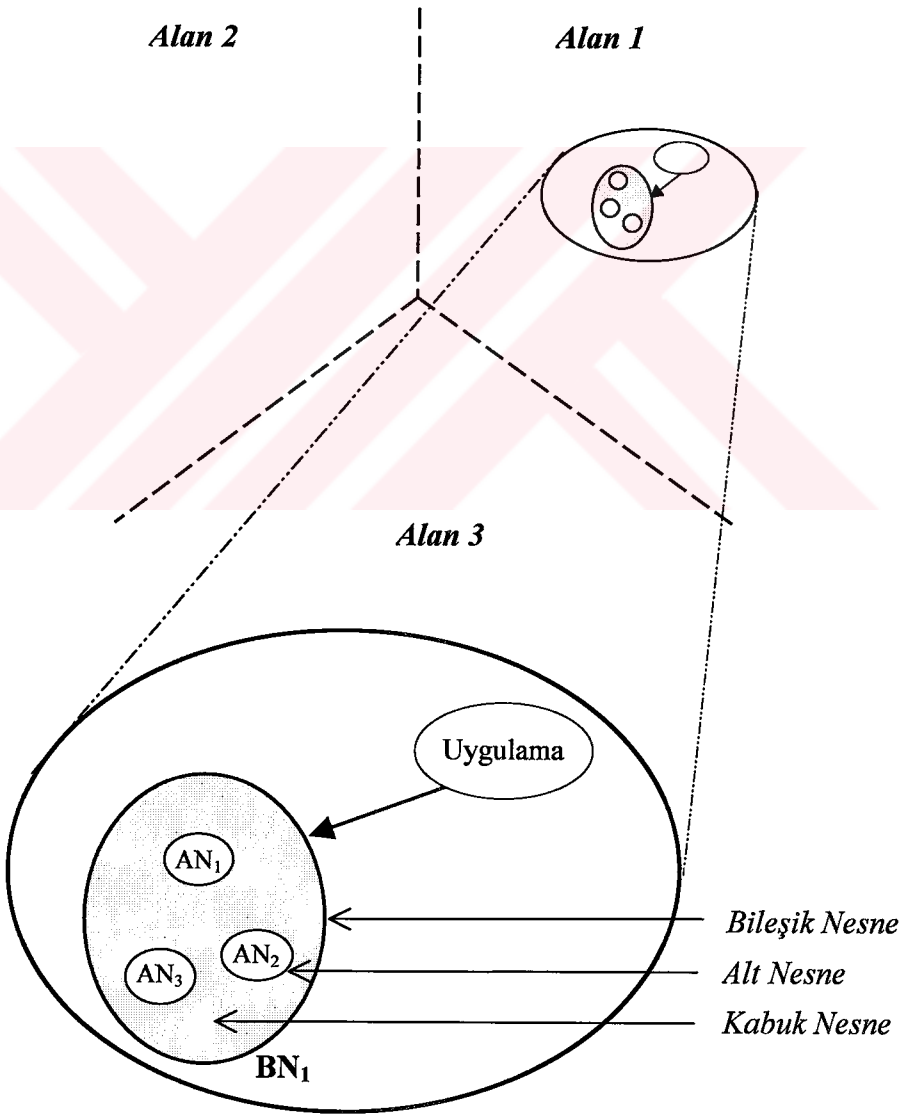
Bu tezde önerilen dağıtılmış bileşik nesne (DBN) modeli, dağıtılmış ortamda birbirleriyle müşterek çalışan uygulamalar tarafından paylaşımlı olarak kullanılan nesneler için yeni bir yaklaşım sunmaktadır.

Dağıtılmış ince-taneli nesnelerin yönetimi üzerine yapılmış olan çalışmalar [24, 31, 32] göstermiştir ki, dağıtılmış ortamlarda paylaşımlı olarak kullanılan nesneler büyüdükçe, uygulamanın başarımı ve kullanılabilirliği düşmektedir. Bu durumda, nesnenin başarımı ve kullanılabilirliği arttırma ve hata-hoşgörü oranını yükseltme amacıyla nesnelerin çoklu alanlar üzerinde kopyalanması bir çözüm olarak görülmektedir. Fakat, kopyalanma durumunda da büyük hacimli nesnenin yönetilmesi problemi ortaya çıkmaktadır.

Dağıtılmış ortamda müşterek yürütülen grup çalışmalarının özelliği, genelde paylaşılan nesnelerin büyük nesneler olmasıdır. Bu nedenle, kullanıcılara yine aynı şekilde büyük bir nesneyi paylaşıyor görüntüsü sunarak, daha alt seviyede paylaşımın sağlandığı bir nesne modeli geliştirilmiştir. Önerilen yeni modelde, tasarım aşamasında büyük bir nesnenin çok sayıdaki alt nesnelerin birleşiminden oluşan bir bileşik nesne olarak yaratılması ve istekte bulunulan uzak alanlar üzerinde nesnenin sadece yapılan metot çağrısının gerektirdiği ilgili alt nesneleriyle kopyalanması öngörülmüştür. Bu arada, ortaya çıkan parçalı nesne yapısının istekçilere gösterilmeden, onların yine tekil bir nesne ile işlem yapıyorlarmış görüntüsü altında erişimlerini gerçeklemelerini sağlayacak bir saydam model amaçlanmıştır.

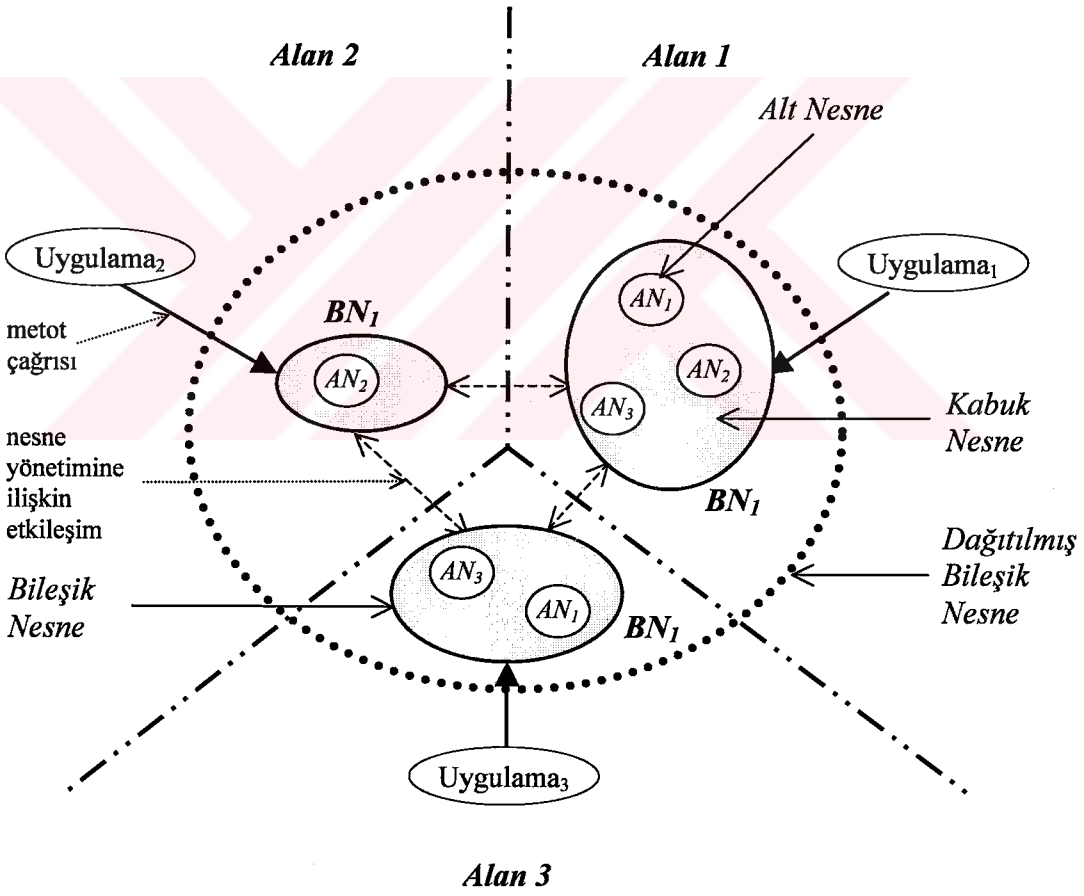
4.2 Dağıtılmış Bileşik Nesne

Önerilen DBN modelinin altında yatan ana fikir, dağıtılmış ortamda paylaşımlı olarak kullanılacak olan nesnenin tasarımcısının, nesne durumunu ve bu durum üzerinde işlemler yürüten metotları çok sayıdaki *alt nesneye (AN)* parçalamasıdır. Her bir alt nesne aynı ya da farklı sınıflardan üretilebilen merkezi yapıdaki gibi bir nesnedir. Tasarımcı bir sonraki aşamada tekil bir nesne görüntüsü sağlamak için, bu alt nesneleri Şekil 4.1'de görüldüğü gibi, üst seviyede bir *kabuk nesnesi* içinde bir bütün olarak toplamak suretiyle, *bileşik nesne (BN)* formunda düzenler. Son olarak, oluşturulan bileşik nesne tekil bir düğümde bir dağıtılmış paylaşılan nesne uygulaması içinde yaratılır.



Şekil 4.1. Tekil bir adres alanı üzerinde üç alt nesnesi ile birlikte yaratılmış bir bileşik nesne

Dağıtılmış ortamda bir başka alanda yaratılmış olan bir BN üzerinde metot çağrısı yürütmek isteyen bir uygulama, öncelikle o DBN için erişim isteğinde bulunur ve bu amaçla bir *arama (lookup)* işlemi yürütür. Bu işlem sonunda ilgili bileşik nesneye ait yalnızca kabuk nesnesi uygulamanın adres alanına yüklenir. Daha sonra, yapılacak olan metot çağrısına göre ilgili alt nesne (veya nesneler) de çağrının yapıldığı adres alanına kopyalanır. Böylece, dağıtılmış paylaşılan nesnenin sadece kullanılacak olan ilgili alt bölüm veya bölümlerinin istekte bulunulan alan üzerine getirilmesi sağlanır. Bu şekilde, öncelikle tekil bir alan üzerinde yaratılmış olan bir bileşik nesne zaman içerisinde erişim istekleri oluştukça, bir çok farklı alan üzerinde ve farklı alt nesneleri ile birlikte kopyalanmış olur. Sonuçta, Şekil 4.2'de görülen dağıtılmış nesne yapısı ortaya çıkar. Bu dağılmış nesne ile ortaya konan modele, *dağıtılmış bileşik nesne modeli* adı verilir.



Şekil 4.2. Üç adres alanı üzerine yayılmış bir dağıtılmış bileşik nesne

Eğer Şekil 4.2'de ortaya çıkmış olan DBN incelenecek olursa, BN_1 bileşik nesnesi öncelikle *Alan 2*'de üç alt nesne ile birlikte (AN_1, AN_2 *ve* AN_3) yaratılmıştır. BN_1 bileşik nesnesi daha sonra, *Alan 1*'de yalnızca AN_1 alt nesnesi ve *Alan 3*'te AN_1 ve *S*

AN_3 alt nesneleri ile birlikte kopyalanmıştır. Daha önce de belirtildiği gibi, BN_1 bileşik nesnesi farklı alanlar üzerinde, yalnızca çağrılan metodu destekleyen alt nesneler ile birlikte kopyalanır.

Şekil 4.2'deki BN_1 bileşik nesnesini oluşturan alt nesneleri ve bunların yapılarını yalnızca BN_1'in tasarımcısı bilmektedir ve tüm diğer uygulamalar, nesne hakkında kendilerine sunulan uygulama arayüzü dışında bir başka bilgiye sahip değildirler. Bu uygulamalar aynı zamanda hangi alt nesnenin kendi alanlarına yüklendiği, yükleme işleminin nasıl yapıldığı, dinamik bağlama, senkronizasyon ve tutarlılık gibi daha bir çok konunun nasıl çözümlendiğini bilmemektedirler. Tüm bu ayrıntılar istekçilere saydam olarak, Bölüm 6'da açıklandığı şekliyle, alt tabakadaki DBNTO ara katman yazılımı tarafından yürütülmektedir.

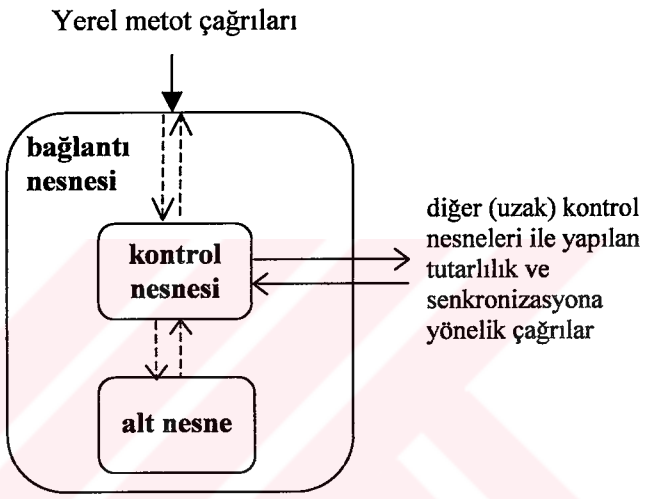
Bir DBN'nin iki farklı görünüş şekli vardır: DBN'yi kullanan uzak uygulamaların bakış açısına göre; o diğer nesnelerden farkı olmayan tekil paylaşılan bir nesnedir. Farklı adres alanlarında bulunan çok sayıdaki uygulama tarafından paylaşılmaktadır. Bu nesneye programcı tanımlı bir arayüz üzerinden erişilir. Bileşik nesneyi oluşturan alt nesneler ve özellikle de sisteme nasıl dağıldıkları görünemez. Bileşik nesnenin tasarımcısının bakış açısına göre ise; bir DBN birbirleriyle birlikte çalışan bir alt nesneler kümesinden oluşmaktadır. Bu yapıdaki her bir alt nesne temel bir nesnedir. Diğer bir deyişle, her bir alt nesne merkezi bir gösterime sahiptir. DBN'nin bir parçasını gerçekleyen her bir alt nesne başlangıçta tekil bir adres alanına yerleştirilir. Daha sonra, çalışma içerisinde farklı adres alanlarında kopyalanan alt nesneler, DBN'nin tutarlı bir görüntüsünü oluşturmak için diğer adres alanındaki alt nesnelerle haberleşirler.

4.3 Bir Bileşik Nesnenin İç Yapısı

Amaçlanan DBN yapısının, bir önceki bölümde ele alındığı şekliyle, sadece çok sayıdaki alt nesnenin bir kabuk nesnesi içerisine yerleştirilmesi suretiyle elde edilemeyeceği açıktır. Bir alt nesnenin gerektiğinde bir diğer adres alanı üzerinde kopyalanarak o alana daha önce taşınmış olan kabuk nesneye bağlanması gerekir. Ayrıca, farklı alanlar üzerinde kopyalara sahip olan bir alt nesneye, nesnenin bütünlüğünü bozmaksızın çağrıda bulunulması ve bir alt nesne kopyasının

güncellenmesi durumunda diğer kopyaların da o kopya ile tutarlılığının sağlanması gerekir. İşte bu sebeplerden dolayı nesne yapısına bazı eklentilerin yapılması zorunlu olmaktadır. Çözüm olarak, bileşik nesnenin yaratıldığı kabuk nesnesi ile her bir alt nesne arasına, *"bağlantı"* ve *"kontrol"* nesneleri olarak adlandırılan iki ara nesne eklenmiştir.

Bir bileşik nesneyi oluşturan her bir alt nesnenin önüne eklenen bağlantı ve kontrol nesneleri ile birlikte, Şekil 4.2'de yer alan her bir AN_i nesnesi aslında Şekil 4.3'teki gibi bir üçlü bir nesne grubu şeklinde gerçeklenmektedir.



Şekil 4.3. Alt nesne erişim düzeni

Bağlantı ve kontrol nesneleri bir bileşik nesnenin dağıtılmış bileşik nesne olarak kullanılabilmesini sağlayan ara nesnelerdir. Dinamik bağlama, senkronizasyon ve tutarlılığı sağlama gibi karmaşık ve gerçeklemesi zor olan dağıtılmış sistemle ilgili ayrıntıları kullanıcıdan gizleyen bu nesneler, ilgili alt nesnelerin arayüz tanımlamaları kullanılarak otomatik olarak üretilirler. Bu amaçla, ayrıntıları Bölüm 7'de açıklanan bir *Otomatik Sınıf Üreteci (OSÜ)* geliştirilmiştir.

4.3.1 Bağlantı Nesnesi

Herhangi bir DBN üzerinde bir metot çağrısı yürütüldüğünde, çağrının gerektirdiği alt nesnenin, eğer daha önceden yüklenmemiş ise, o adres alanına yüklenmesi gerekmektedir. Eğer çağrı DBN'nin yaratıldığı alandan yapılmış ise, zaten tüm alt nesneler bu alana yereldirler ve herhangi bir uzaktan yüklenme işlemi gerekmez. Ancak, DBN için bir arama işlemi yürüterek kabuk nesneyi kendi adres alanına

bağlamış olan bir uzak uygulamanın bir metot çağrısı yürütmesi durumunda yüklenme gereği ortaya çıkar.

Bir alt nesnenin yürütme anında dinamik olarak istekte bulunulan adres alanı üzerine yüklenebilmesini sağlamak amacıyla *bağlantı nesnesi* olarak adlandırılan ve kabuk nesne ile kontrol nesnesi arasında yer alan bir ara nesne kullanılır. Eğer kontrol nesnesi o adres alanı üzerinde bulunuyor ise, bağlantı nesnesi içerisinde bulunan nesne referansı kontrol nesnesine işaret eder. Eğer referans değeri *null* ise, bu alt nesnenin henüz o adres alanına yüklenmediğini gösterir. Bu durumda, Bölüm 6'da da açıklandığı üzere, bağlantı nesnesi DBNTO sistem çağrılarını kullanarak yükleme işlemini gerçekleştirir. Tüm alt nesne erişimleri bağlantı nesnesi üzerinden sağlandığı için, bağlantı nesnesi alt nesnenin sunduğu arayüzün aynısını sunmaktadır. Yüklenme işlemi tamamlandıktan sonra, metot çağrısı bir yerel çağrı şeklinde yürütülür.

4.3.2 Kontrol Nesnesi

Bir alt nesnenin farklı alanlar üzerindeki kopyalarına birden fazla uygulamadan eşzamanlı olarak metot çağrıları gelebilir. Eğer gelen bu çağrılar birbirleri ile *çelişmeyen (non-conflicting)* çağrılar ise (örneğin, hepsi alt nesnenin durumu üzerinde okuma işlemi gerçekleştirecek olan çağrılar ise), problem yoktur. Ancak, çağrılardan en az biri alt nesnenin durumu üzerinde güncelleme işlemi yürütecek bir çağrı ise, bu durumda *çelişme (conflict)* söz konusudur ve alt nesnenin bütünlüğü bozulabilir. Bu yüzden, farklı alanlardaki kopyalara yönelen eşzamanlı metot çağrıları birbirleri ile çelişiyor iseler, ardışıl olarak yürütülmelidirler. Ayrıca, yapılan bir metot çağrısı o alanda bulunan alt nesne kopyasını diğerlerinden farklı kıldıysa, bu durumda kullanılan tutarlılık protokolüne göre, diğer kopyalarında güncellenmesi veya geçersiz kılınmaları gerekmektedir. İşte bu iki sebepten dolayı alt nesnenin hemen önüne sözü edilen senkronizasyon ve tutarlılık işlemlerini gerçekleyecek ve yapılan tüm erişimleri denetleyecek bir *kontrol nesnesi* yerleştirilir.

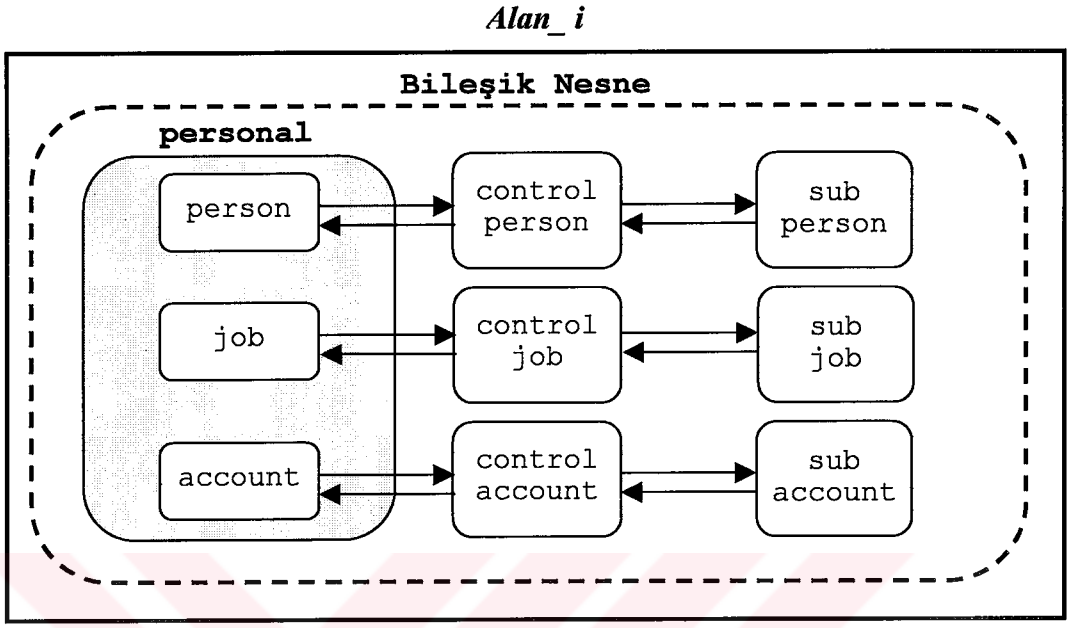
4.4 Bir Bileşik Nesnenin Hazırlanması

Dağıtılmış bileşik nesne yapısının, tekil bir adres alanında çalışmakta olan uygulama tarafından yaratılan bileşik nesnenin diğer alanlar üzerinde, yalnızca yürütülen metot çağrılarının gerektirdiği alt nesneler ile birlikte kopyalanması suretiyle oluştuğu daha önceki bölümlerde anlatılmıştı. Bu bölümde ise, bileşik nesnenin yaratılma aşamasında izlenecek işlem adımları anlatılacaktır.

Dağıtılmış Bileşik Nesne Tabanlı Ortam (DBNTO) ara katman yazılımı tasarımcıya merkezi bir yapıda tek bir uygulama tarafından kullanılacakmış görüntüsü altında, dağıtım ile ilgili ayrıntılarla ilgilenmesine gerek kalmadan, dağıtılmış bileşik nesneyi geliştirmesine olanak sağlayacak bir ortam hazırlar. Bu nedenle, tasarımcının sadece bileşik nesneyi oluşturan farklı tiplerdeki alt nesnelerin üretileceği sınıf tanımlamalarını ve alt nesnelerin bileşik nesne içerisinde bir araya getirilmesini sağlayan kabuk nesnenin sınıf tanımlamasını hazırlaması yeterlidir. Daha sonra, dağıtım ile ilgili bölümleri içeren bağlantı ve kontrol nesnelerinin sınıfları, alt nesne sınıflarına ait arayüz tanımlamaları kullanılarak sistem tarafından otomatik olarak üretilecektir.

Yukarıda sözü edilen işlem sürecini bir örnek üzerinde izlemek konunun kavranması açısından daha faydalı olacaktır. Bu amaçla, bir kişiye ait kişisel bilgiler, çalıştığı işi ile ilgili bilgiler ve banka hesap bilgilerinin, o kişiyi ifade eden bir bileşik nesne yapısı içinde bir bütün olarak tutulmak istendiği kabul edilsin. (Otomatik olarak üretilecek olan bağlantı ve kontrol nesnelerine ait sınıf yapılarındaki tüm tanımlamalar İngilizce olduğu için, bütünlüğü bozmamak amacıyla, örnek içinde verilecek olan kullanıcı tanımlı tüm sınıf, nesne ve değişken adları da İngilizce olacaktır.) Bu durumda, bileşik nesnenin `sub_person`, `sub_job` ve `sub_account` şeklinde üç alt nesnesi olacaktır. Üç alt nesneye erişimi kontrol eden üç kontrol nesnesi; `control_person`, `control_job` ve `control_account` ve alt nesneleri dinamik olarak bağlamak amacıyla kullanılacak `person`, `job` ve `account` adlı üç bağlantı nesnesi olacaktır. Son olarak, bu üç alt nesneyi bağlantı nesneleri üzerinden bir bileşik nesne yapısı içerisinde birleştirmek için kullanılan `personal` adlı kabuk nesnesi yer alacaktır. Bu nesneler arasındaki erişim düzeni ve `personal` bileşik nesnesinin son durumu

Şekil 4.4'te gösterildiği gibidir. `personal` nesnesi içerisinden doğrudan erişilen nesneler bağlantı nesneleridir. Bağlantı nesnelerinden kontrol nesnelerine; ve bu nesnelerden de alt nesnelere erişim sağlanır.



Şekil 4.4. Bileşik nesne içindeki erişim düzeni

DBN modeli içinde yer alan bağlantı ve kontrol nesneleri otomatik olarak üretileceği için, alt nesne sınıfları ve arayüz tanımlamalarını içeren dosyaların belli bir kurala göre adlandırılmaları gerekir. Bu nedenle, alt nesnelere ait sınıf dosyalarının `Sub_` öneki ile başlamaları ve arayüz tanımlama dosyalarının da `_itf` soneki ile sonlanmaları kabul edilmiştir. Bu açıklamanın ardından, yukarıda sözü edilen üç alt nesne için tasarımcı üç alt sınıf tanımlaması olan `Sub_Person.java`, `sub_Job.java`, `Sub_Account.java`'yı ve yine üç sınıf için arayüz tanımlamaları `Person_itf.java`, `Job_itf.java`, `Account_itf.java` ve son olarak `personal` nesnesinin üretileceği `Personel.java` sınıf tanımlamasını hazırlar.

Otomatik sınıf üreteci tarafından arayüz tanımlamaları kullanılarak üretilecek olan sınıflar; bağlantı nesneleri sınıfları, `Person.java`, `Job.java`, `Account.java` ve kontrol nesneleri sınıfları, `Control_Person.java`, `Control_Job.java`, `Control_Account.java` olur.

Bu noktadan itibaren bağlantı ve kontrol nesnesi sınıflarının incelenmesi aşamasında sadece sub_person alt nesnesi üzerinde çalışılacaktır. sub_person alt nesnesinin türetildiği Sub_Person.java sınıf tanımlaması Şekil 4.5'te verildiği gibi olsun. Bu durumda, Sub_Person.java sınıfına ait arayüz tanımlaması, Person_itf.java, Şekil 4.6'da verildiği gibi olur.

```
public class Sub_Person implements
                java.io.Serializable {
   String  lastname;
   String  firstname;
   public Sub_Person() {
      lastname = new String("");
      firstname = new String("");
   }
   public void put_lastname(String name) {
      lastname = name;
   }
   public String get_lastname() {
      return lastname;
   }
   public void put_firstname(String name) {
      firstname = name;
   }
   public String get_firstname() {
      return firstname;
   }
}
```

Şekil 4.5. Bir alt nesne sınıfı olan Sub_Person.java sınıf kodu

```
public interface Person {
   public void W_put_lastname(String name);
   public String R_get_lastname();
   public void W_put_firstname(String name);
   public String R_get_firstname();
}
```

Şekil 4.6. Sub_Person.java sınıfı için Person_itf.java arayüz tanımlaması

Şekil 4.6 incelendiğinde her bir metot adının başında W_ veya R_ öneklerinin yer aldığı görülecektir. Bunun nedeni, alt nesnenin farklı alanlarda bulunan kopyaları üzerinde yürütülecek olan metot çağrıları arasında gerekli eşzamanlılık kontrollerinin yapılması gerekliliğidir. Kontrol nesnesi tarafından, eğer güncelleme erişimi

yapılacaksa yazma, okuma işlemi yapılacaksa okuma izninin alınması gerekir. Alt nesnenin tasarımcısı erişim türünü belirlemek üzere, her bir metot için, o metot adının başına güncelleme erişimi için W ve okuma erişimi için R öneklerini ekler. Böylece, kontrol nesnesi sınıfı üretilirken W ve R simgeleri dikkate alınarak, sınıf yapısının içine erişim türü için gerekli olan denetim kodu eklenir.

DBN tasarımcısı, `Sub_Job.java` ve `Sub_Account.java` sınıflarını da hazırladıktan sonra, bu sınıflar için arayüz tanımlamalarını içeren `Job_itf.java` ve `Account_itf.java` dosyalarını Şekil 4.6'dakine benzer şekilde tanımlar. Böylece, bu üç alt nesne için sınıf ve arayüz tanımlamaları hazırlanmış olur.

Bir sonraki aşama, bağlantı ve kontrol nesnelerinin üretileceği sınıf dosyalarının üretilmesi olacaktır. Bu amaçla, daha önce de belirtildiği gibi, yapısı Bölüm 7'de açıklanan OSÜ kullanılır. OSÜ, parametre olarak verilen sınıflara ait arayüz tanımlamaları `Person_itf.java, Job_itf.java, Account_itf.java` ayrıştırmak suretiyle analiz eder ve ardından bağlantı nesnesi sınıfları `Person.java, Job.java` ve `Account.java` ile, kontrol nesnesi sınıflarını `Control_Person.java,Control_Job.java,Control_Account.java` sınıflarını üretilir.

Sınıf yapısı Şekil 4.5'te `Sub_Person.java` ile ve arayüz tanımlaması da Şekil 4.6'da `Person_itf.java` ile verilen örnek için üretilen bağlantı nesne sınıfı `Person.java`'nın yapısı Şekil 4.7'de, içeriği ile ilgili ayrıntıların bir sonraki bölümde inceleneceği `Control_Person.java` sınıfının genel yapısı da Şekil 4.8'de sunulmuştur.

```
01  public class Person implements java.io.Serializable {

02    int              object_id;
03    Control_Person   control_object;

04    public Person() {
05       try {
06          control_object = new Control_Person();
07          object_id = control_object.get_id();
08       }
09       catch (Exception ex) {}
10    }

11    private void writeObject(ObjectOutputStream out)
                                        throws IOException {
12       Integer id = new Integer(object_id);
13       out.writeObject(id);
14       out.flush();
15    }

16    private void readObject(ObjectInputStream in)
             throws IOException, ClassNotFoundException {
17       Integer res;
18       res = (Integer)in.readObject();
19       control_object = null;
20       object_id = res.intValue();
21    }

22    public void put_lastname(String name) {
23       if (control_object == null) {
24          try {
25             control_object = (Control_Person)
               DCOBE_server.get_control_object(object_id);
26          } catch (Exception ex) {}
27       }
28       control_object.put_lastname(name);
29    }

30    public String get_lastname() {
31       if (control_object == null) {
32          try {
33             control_object = (Control_Person)
               DCOBE_server.get_control_object(object_id);
34          } catch (Exception ex) {}
35       }
36       return control_object.get_lastname();
37    }
//    put_firstname() ve get_firstname() metotlarının yapısı
      put_lastname() ve get_lastname() ile aynı olacağından
      burada gösterilmemiştir.
38 }
```

Şekil 4.7. OSÜ tarafından üretilen `Person.java` bağlantı nesnesi sınıf kodu

```
01  public class Control_Person {
02    int           object_id;
03    Sub_Person    sub_obj;
04    boolean       master = true;
05    int           server_id;

06    public Control_Person() {
07       sub_obj = new Sub_Person();
08       try {
09          server_id=DCOBE_server.get_server_id();
10          object_id = DCOBE_server.register_new_object
                        (this,sub_obj);
11       } catch (Exception ex) {}
12    }

13    public Object get_object() {...}
14    public void invalidate_object(){...}
15    public void update_object(Object sub_obj) {...}
16    public Object Copy_my_self(){...}
17    public void incoming_object(Object sub_obj) {...}
18    public void give_object(int ref_dest,int kind){...}
19    public void ask_object(int kind) {...}
20    public boolean ask_to_read(int ref_dest){...}
21    public void end_of_use(int ref_dest,int kind) {...}

22    public void put_lastname(String name) {
23       ask_object(W);
24       sub_obj.put_lastname(name);
25       try { if (master==true) end_of_use(server_id,W);
26             else DCOBE_server.release_object(object_id);
27       } catch (Exception ex) {ex.printStackTrace()}
28    }

29    public String get_lastname() {
30       ask_object(R);
31       String obj= sub_obj.get_lastname();
32       try { if (master==true) end_of_use(server_id,R);
33             else DCOBE_server.release_object(object_id);
34       } catch (Exception ex) {ex.printStackTrace()}
35       return obj;
36    }

//     put_firstname() ve get_firstname() metotlarının yapısı
      put_lastname() ve get_lastname() ile aynı olacağından
      burada gösterilmemiştir.
37  }
```

Şekil 4.8. OSÜ tarafından üretilen `Control_Person.java` kontrol nesnesi sınıf kodu

Buraya kadar izlenen aşamaların ardından, son olarak bileşik nesnenin türetileceği kabuk nesnenin sınıf tanımlamasının (`Personal.java`) hazırlanması gerekmektedir. Bu amaçla yazılabilecek örnek bir sınıf yapısı Şekil 4.9'daki gibi olabilir.

```
01  public class Personal implements
                 java.io.Serializable {

02    Person     person;
03    Job        job;
04    Account    account;
05    String     lastname;
06    String     firstname;
   ..................    ......................

07    public Personal() {
08       person = new Person();
09       job = new Job();
10       account = new Account();
11       lastname = new String("");
12       firstname = new String("");
      ..................    ......................
13    }

14    public void write_lastname(String name) {
15       person.put_lastname(name);
16    }

17    public String read_lastname() {
18       lastname = person.get_lastname();
19       return lastname;
20    }

21    public void write_firstname(String name) {
22       person.put_firstname(name);
23    }

24    public String read_firstname() {
25       firstname = person.get_firstname();
26       return firstname;
27    }

//    job ve account nesnelerini içeren
      ilgili diğer metot tanımlamaları
      ..........................................................
      ..........................................................
28 }
```

Şekil 4.9. Yerel olarak bir bileşik nesne yaratan kod örneği, `Personal.java`

4.5 Bileşik Nesne Üzerinde Bir Metot Çağrısının Yürütülmesi

Şekil 4.9'dan da görülebileceği gibi, bir bileşik nesne içinden doğrudan erişilen nesneler yalnızca bağlantı nesneleridir (Şekil 4.9; 18, 22, 25 numaralı satırlar). Bağlantı nesnesi içinden kontrol nesnesine (Şekil 4.7; 28 ve 36) ve kontrol nesnesi içinden de dağıtılmış bileşik nesnenin semantiğinin bir parçasını gerçekleyen alt nesneye erişilir (Şekil 4.8; 24 ve 31).

Adres alanına yerel olan bileşik nesneden gelen metot çağrısı öncelikle bağlantı nesnesi üzerinde bir çağrıya dönüşür (Şekil 4.9; 15). Eğer bağlantı nesnesi içinde bulunan nesne referansı *null* ise (Şekil 4.7; 23), bu ilgili alt nesneye ait kontrol nesnesinin henüz o adres alanına yüklenmediğini gösterir. Bu durumda, ara katman yazılımının gerekli sistem çağrısını yürüten bağlantı nesnesi (Şekil 4.7; 25) kontrol nesnesini kendi adres alanına yükler ve ardından metot çağrısını kontrol nesnesine aktarır (Şekil 4.7; 28).

Kontrol nesnesi gelen metot çağrısını alt nesneye iletmeden önce, erişim izni almak zorundadır. Çünkü, aynı alt nesnenin farklı alanlardaki kopyaları üzerinde eşzamanlı metot çağrılarının yürütülmesi mümkündür. Eğer bu eşzamanlı çağrılar salt oku düzeninde ise, eşzamanlı yürütülmeleri sistemin başarımını olumlu etkiler. Ancak, bu çağrıların alt nesnelerin bütünlüğünü bozabilecek çağrılar olmaları durumunda (okuma/yazma ya da yazma/yazma çağrıları gibi), birbirleriyle çelişen çağrılar adını alırlar ve ardışıl olarak yürütülmeleri gerekir. Bundan dolayı, nesne durumu tutarlılığının sağlanması amacıyla, bir alt nesne için *sahiplik (ownership)* tanımı yapılmıştır. Bu tanıma göre, alt nesnenin durumu üzerinde en son güncelleme işlemi yürütmüş olan alandaki kontrol nesnesi o alt nesnenin sahibidir ve sahiplik zaman içerisinde bir alandaki kontrol nesnesinden bir diğer alandakine aktarılır.

Kontrol nesnesi alt nesne üzerinde metot çağrısını yürütmeden önce, erişim türünü okuma/güncelleme şeklinde belirterek, erişim için izin isteğinde bulunur (Şekil 4.8; 23 veya 30). Eğer kontrol nesnesinin `master` değişkeninin değeri "`true`" ise, bu durum sahipliğin yerel kontrol nesnesinde olduğunu gösterir ve o an bir yazma çağrısı yürütülmüyor ve bekleyen istekler kuyruğu da boş ise, erişim isteğine izin verilir. Aksi takdirde, erişim isteği bekleyenler kuyruğuna eklenir. Eğer `master` değişkeni "`false`" ise, bu durumda ara katman yazılımının gerekli sistem çağrısını

yürüten kontrol nesnesi, isteği ağ üzerinden sahipliği elinde bulunduran kontrol nesnesine aktarır. Erişim izninin gelmesinin ardından, kontrol nesnesi alt nesne üzerinde metot çağrısını yürütür (Şekil 4.8; 24 veya 31). Yapılan çağrı sonucunda alt nesnenin durumu üzerinde güncelleme yapılmış ise, sahiplik yerel kontrol nesnesine geçecektir. Aksi durumda, sahiplik değişmeyecektir. Erişimin tamamlanması ile birlikte, erişimin sona erdiği bilgisi sahipliği elinde bulunduran kontrol nesnesine bildirilir (Şekil 4.8; 25 veya 33). Kontrol nesnesi bu arada istekte bulunmuş, fakat yürütülen metot çağrısı ile çeliştiği için beklemeye alınmış olan istekler varsa, o isteklerden mümkün olanlara izin verir.

5. KOPYALANMIŞ ALT NESNELERİN YÖNETİMİ

DBN modelinin ana hedeflerden biri de, nesne dağıtımını uygulama programcısına saydam kılarken, aynı zamanda alt nesnelerin dağıtılmış kopyalarının etkin bir şekilde yönetilmesini de sağlamaktır.

Alt nesne kopyalarının yönetiminde çözümlenmesi gereken iki ana konu vardır. Bunlar; aynı anda farklı alanlardaki kopyalara yapılan erişimlerin alt nesnenin bütünlüğünü bozmadan gerçeklenmesini sağlayacak şekilde eşzamanlılık kontrolü ile, kopyalar arasında tutarlılığın sağlanabilmesi için alt nesneler üzerinde geçersiz kılma veya güncelleme gibi işlemlere izin veren tutarlılık mekanizmalarının tasarlanmasıdır. Bu mekanizmalar, aynı zamanda, uygulama programcısından da gizli olarak çalıştırılmalıdır. Uygulama programcısı her bileşik nesneyi bir yerel nesne gibi görmeli ve sadece Java referanslarını kullanarak işlemlerini yürütmelidir.

Sözü edilen bu mekanizmalar tüm alt nesne erişimlerini denetleyen kontrol nesnesi içine yerleştirilmişlerdir. Böylece, eşzamanlılık kontrolü ve tutarlılık sağlama mekanizmaları bir bütün olarak nesne modelinin bir parçasını oluşturmaktadırlar. Bu yapının avantajı, sadece kontrol nesnesi sınıfını üreten OSÜ üzerinde yapılacak bazı değişiklikler ve eklemelerle, DBNTO sistemini oluşturan diğer modüller üzerinde hiçbir değişiklik yapmaksızın, bu mekanizmalara eklentiler yapılabilmesi ya da tamamıyla farklı yeni mekanizmaların geliştirilebilmesidir.

Tezde uygulanan modeli açıklamadan önce, literatürdeki mevcut nesne erişim kategorileri ve tutarlılık modellerini kısaca incelemek faydalı olacaktır.

5.1 Nesne Erişim Kategorileri

Dağıtılmış ortamda paylaşımlı olarak erişilen kopyalı nesnelerin tutarlı bir görüntü sunmaları gerekmektedir. Bu amaçla geliştirilen çeşitli tutarlılık modelleri mevcuttur. Uygun bir nesne tutarlılık modelinin seçilmesi, programlama ve nesne modellerinin karmaşıklıklarıyla nesne erişimlerinin en aza indirilmesi arasında kurulacak bir

denge ile belirlenir. Yapılan çeşitli çalışmalar sonucunda bir çok farklı tutarlılık modeli geliştirilmiştir. Ancak, geliştirilen bu tutarlılık modellerinin tam bir şekilde sınıflandırmasını yapacak tek bir hiyerarşik model yoktur. [33] tarafından çeşitli modellerin kolay bir şekilde sınıflandırılmasını sağlayacak bir yöntem verilmiştir.

Tutarlılık modelleri bir takım özellikleri olan erişimler üzerine düzenleme kısıtlamaları yüklemektedir. Genelde, model ne kadar çok özelliği göz önünde bulundurursa o kadar zayıf olur. Modelin ele alması gereken bazı özellikler şunlardır;

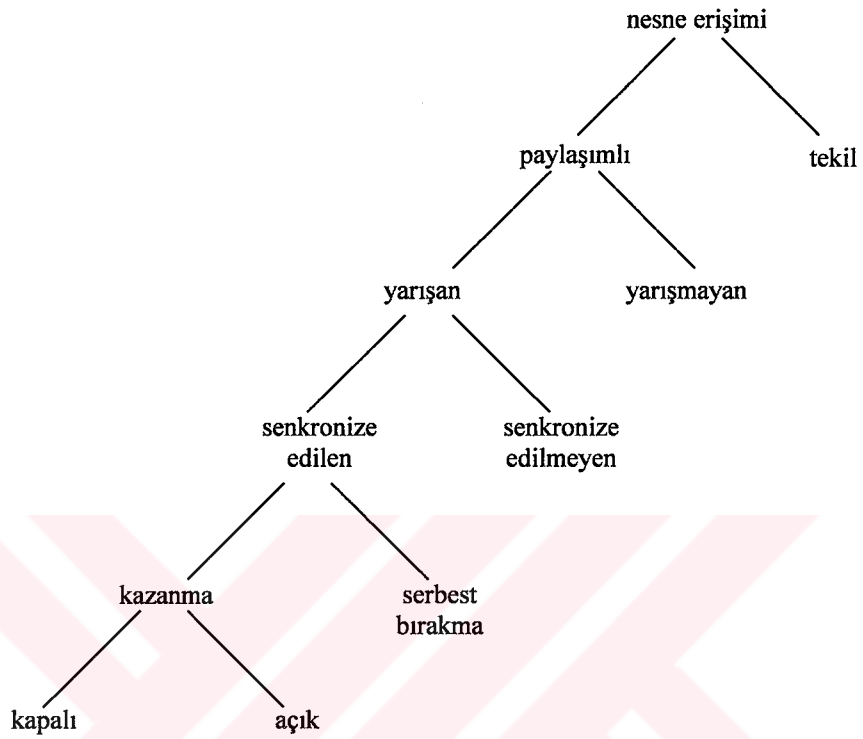
- erişimin yapılacağı yerleşke,
- erişim şekli (oku, yaz, her ikisi de),
- erişim sırasında aktarılan değer,
- erişimin nedenselliği,
- erişimin kategorisi.

Nedensellik (causality) özelliği, a_1 ve a_2 gibi iki erişim ele alındığında, a_1'in a_2'den önce veya a_2'nin a_1'den önce gerçeklenmesi durumlarını inceleyen ve sonuçlarını ortaya koyan bir ilişkidir [34].

Kullanışlı bir sınıflandırma Şekil 5.1'deki gibi yapılabilir. Bu sınıflandırma [35]'te yer almış bir sınıflandırmadır. Şekil 5.1'deki sınıflandırmaya göre, erişim paylaşılan bir nesneye ya da tekil bir nesneye olabilir. Tekil erişimlerin çözümlenmesi basittir ve burada ele alınmayacaktır. Paylaşılan erişimlerde *yarışan (competing)* ve *yarışmayan (non-competing)* olarak iki gruba ayrılır. Eğer bir nesneye aynı anda birden fazla erişim yapılıyorsa ve bunlardan en az biri yazma erişimi ise ve bu erişimler bir sıraya sokulmamışlar ise, bu durumda bu erişimlerin birbirleriyle yarışan erişimler olduğu söylenebilir. Örneğin, kritik bölge içindeki paylaşılan değişkenlere erişim, yarışmayan bir erişimdir. Çünkü, *karşılıklı dışlama (mutual exclusion)* erişimler arasındaki sıralamayı gerçeklemektedir.

Yarışan bir erişim *senkronize edilen (synchronizing)* ve *senkronize edilmeyen (non-synchronizing)* şeklinde ikiye ayrılır. Senkronize edilen erişimlerde, erişim yapan süreçler (veya nesneler) arasında bir sıralama uygulanır. Örneğin, daha önce başlatılmış olan erişimlerin tümü sonlanmadan, yürütülmekte olanlar ile yarışan yeni bir erişimin başlamasına izin verilmez. Ancak, yarışan tüm erişimler senkronize

edilmiş erişimler olmak zorunda değildir. Örneğin, bazı uygulamalar yarışan erişimlerin senkronize edilmeden yürütülmesine izin verirler. Bu uygulamalar, bir okuma erişiminin en son yazılan değerin geri döndürülmesinin gerekli olmadığı uygulamalardır.



Şekil 5.1. Nesne erişim kategorileri

Senkronize edilen erişimler de *kazanma (acquire)* ve *serbest bırakma (release)* şeklinde iki gruba ayrılırlar. Kazanma daima okuma senkronizasyonu ile birlikte kullanılırken, serbest bırakma yazma senkronizasyonu erişiminde kullanılır. Son olarak kazanma erişimi de *kapalı (exclusive)* ve *açık (non-exclusive)* olabilir. Kapalı kazanım çoklu erişimlere izin verirken, açık kazanım daha önceki kazanımların tümü serbest bırakılana kadar çoklu erişimleri geciktirir.

5.2 Tutarlılık Modelleri

Bu bölümde dağıtılmış nesne erişimlerinde kullanılan tutarlılık modelleri genel yapıları itibarıyla incelenecektir. Mevcut modeller için kurallı tanımlamalar verilmeyecektir. Kurallı tanımlamalar için ayrıntılı bilgi [35, 36]'da yer almaktadır.

Tutarlılık modelleri *kuvvetli (strong) ve zayıf (weak)* olmak üzere iki kategori altında incelenebilir.

5.2.1 Kuvvetli Tutarlılık Modelleri

Kuvvetli tutarlılık modelinde, nesne kopyalarından hiçbiri diğerlerinden farklı değildir. Bir nesne kopyası üzerinde yapılan bir değişiklik tüm diğer kopyalara da yansıtılır. Bu yüzden, tüm yazma erişimleri *ardışıl (sequential)*'dır.

Kuvvetli tutarlılık modeli sınıfına giren tutarlılık tipleri aşağıdaki şekilde sınıflandırılabilir [33].

Atomik tutarlılık (atomic consistency): Tutarlılık modelleri içinde en katı olanıdır. Bu yönteme göre, işlemlerin etkisi *işlem aralığı (operation interval)* olarak adlandırılan belli zamanlarda uygulanır. İşlem aralığı, zamanın birbiri üzerine binmeyen ardışık zaman dilimlerine ayrılması şeklinde düşünülebilir. Aynı zaman aralığı içerisinde aynı nesneye okuma ve yazma erişimleri olabilir. Bu amaçla kullanılabilecek bir çözüm; okuma işlemleri için bir okumaya-başla ve yazma işlemleri için yazmayı-bitir zamanlarında okuma ve yazmanın etkilerini göstermesini sağlayacak zaman aralıkları belirlemektir. Bu statik atomik tutarlılık olarak adlandırılır. Dinamik atomik tutarlılıkta ise, eğer sonuçta oluşan işlem sırası bir seri yürütüme eşit oluyor ise, işlemlerin etkileri işlem aralığının herhangi bir noktasında etkinleştirilebilir.

Ardışıl tutarlılık (sequential consistency): Ardışıl tutarlılık ilk kez Lamport tarafından 1979'da tanımlanmıştır [37]. Lamport'a göre; eğer işlemler belli bir sıraya göre yürütülmüş iseler, herhangi bir işlemler kümesinin sonucu diğerleriyle aynıdır. Ardışıl olarak tutarlı olan bir sistemde, işlemler tüm süreçler tarafından aynı sıra ile yürütülürler. Üst seviyede ardışıl bir sıralama vardır ve tüm yürütümler bu sıraya uymalıdır.

Ardışıl tutarlılık çoğunlukla *yazma-geçersiz kılma (write-invalidate)* protokolü kullanılarak uygulanır (bu protokolün özellikleri ilerleyen bölümlerde verilecektir). Ardışıl tutarlılık, bir dağıtılmış sistem içinde gerçeklenebilecek, tek işlemci bellek modeline en yakın olan modeldir. Ardışıl tutarlılık, bir nesne durumunda yapılacak bir değişikliğin tüm diğer alanlar üzerinde bulunan kopyalarda, sanki aynı makine üzerinde yürütülmüş bir çağrı gibi, hemen uygulanmasını gerektirir. Bir okuma

hatası oluştuğunda nesne o an mevcut sahibinden *yalnızca-okunabilir (read-only)* olarak işaretlenmek suretiyle alınır. Geçersiz kılma protokolü uygulandığından, bir nesne durumu üzerinde değişiklik yapıldığında, diğer alanlarda bulunan nesneler geçersiz kılınır.

Ardışıl tutarlılık protokolü *yazma-güncelleme (write-update)* protokolü ile birlikte de uygulanabilmektedir. Bu durumda, güncelleme mesajı kopyaya sahip tüm alanlara iletilir. Tüm alanlardan güncellemelere ilişkin "tamamlandı" yanıtı alınmadan yerel işlem teslim edilmez.

Nedensel tutarlılık (causal consistency): Birbirleriyle nedensel olarak ilişkili olmaları muhtemel olan yazma işlemleri tüm diğer süreçler tarafından aynı sırayla görülür. Fakat, birbirleriyle ilişkili olmayan eşzamanlı yazma işlemlerinin farklı süreçler tarafından farklı sırayla görülmeleri de söz konusudur.

PRAM tutarlılık: *Paralel rasgele erişim makinesi (Parallel Random Access Machine-PRAM)* tutarlılık modelinde, tekil bir süreç tarafından yapılan yazma işlemleri tüm diğer süreçler tarafından aynı sırada görülürken, farklı süreçler tarafından yapılan yazma işlemleri farklı sıralarda görülebilir.

Yukarıda sözü edilen tüm kuvvetli tutarlılık yöntemlerinde, ağ ya da kopya gecikmesi durumunda okuma ve yazma erişim istekleri gecikecektir. Yine, hata durumunda da istek gerçeklenmeyecektir.

5.2.2 Zayıf Tutarlılık Modelleri

Zayıf tutarlılık modellerinde, çok geniş alana yayılmış sistemlerde tutarlılık gereksinimlerinin gevşetilerek kullanılabilirlik ve başarımın arttırması mümkün olabilir. Buradan yola çıkarak kuvvetli tutarlılık yöntemlerine göre daha az sınırlamaları olan zayıf tutarlılık modelleri geliştirilmiştir. Bu kategoriye giren modeller aşağıdaki gibi sınıflandırılabilir:

Giriş tutarlılığı (entry consistency): Giriş tutarlılığı, bir kritik bölgenin ve kritik bölge içerisinde erişilen paylaşılan veriyi koruyan özel senkronizasyon değişkenlerinin sağladığı avantajları sunan bir tutarlılık yöntemidir. Girişten tutarlı bir sistemde, bir nesneye erişim sırasında süreç nesneyi mutlaka tutarlı olarak görür.

Diğer bir deyişle, erişim başladığı anda, nesne durumu mutlak suretle en son güncellenmiş olan nesnenin durumu ile aynıdır. Giriş tutarlılığıyla sağlanan model, bir çok paylaşılan bellekli paralel programlarda olduğu gibi, paylaşılan veriye erişimi koruyan kritik bölgeleri kullanır. Aksi takdirde geri dönen değerin tutarlı bir değer olmaması mümkündür. Model paralel erişimlerin gerektirdiğinden daha güçlü bir tutarlılık da sağlamaz. Bu nedenle, yüksek başarım istenen gerçeklemeler için uygun bir yöntemdir.

Delta tutarlılık: Delta tutarlılık modelinde nesnenin asıl kopyası üzerine yapılan bir güncellemeden belirli ve kısa bir zaman aralığı sonra, herhangi bir okuma işleminin sonucunda dönen değerin tutarlı olduğu garanti edilir.

Nihai tutarlılık (eventual consistency): Nihai tutarlılık modeline göre, herhangi bir okuma işleminin sonucu en son bilinen güncelleme ile tutarlıdır. Fakat, güncellemelerin yayılma zamanları tam olarak belli değildir. Güncellemeler ile ilgili bilgiler kopyaların bulunduğu alanlara dağıtılır ve güncellemeler tamamlandığında nihai olarak kopyaların tutarlılığı da sağlanmış olur.

Gevşek tutarlılık (loose consistency): Gevşek tutarlılık modeline göre, herhangi bir okuma işleminin nesnenin asıl kopyası ile tutarlı olmasına ilişkin bir garanti yoktur. Fakat, sistematik ve genelde manuel olarak çağrılan mekanizmalar tutarlılığı sağlar.

Periyodik tutarlılık: Bu yöntemde, kopyalar asıl kopya ile periyodik olarak senkronize edilirler. Daha sonra belli bir süre geçince tutarlılık tekrar bozulur. İstek üzerine nesnelerin tutarlılığının sağlanmasına ilişkin sistematik bir mekanizma yoktur.

Nesne erişim kategorileri ve tutarlılık modelleri kısaca incelendikten sonra, şimdi kontrol nesnesi içinde yer alan eşzamanlılık kontrolü ve tutarlılığı sağlama mekanizmaları bir arada incelenecektir.

5.3 Eşzamanlılık Kontrolü ve Tutarlılığı Sağlama

Kopyalı nesnelere erişim düzeni onlar üzerinde uygulanacak olan senkronizasyon mekanizmasının yapısını da belirler. Örneğin, herhangi bir uygulama aynı nesnenin birden fazla yazıcı tarafından güncellenmesine izin verirken, bir diğeri yalnızca-

okunabilir çoklu kopyalara sahip tek bir yazıcıya izin verebilir. İlk örnek çok sayıda eşzamanlı yazıcılara izin veren bir modeli gerektirirken, diğeri için daha basit ve daha kolay gerçeklenebilen *çoklu okuyucu tek yazıcı (multiple reader-single writer-MRSW)* senkronizasyon modeli yeterli olabilir. DBN nesne modelinde de MRSW yapısı kabul edilmiştir. Yani bir alt nesnenin farklı alanlardaki kopyalarına aynı anda yapılan çağrılardan en az biri yazma erişimli bir çağrı ise, bu çağrılar ardışıl olarak yürütüleceklerdir.

Alt nesne kopyaları arasında tutarlılığı sağlayan mekanizma için ardışıl tutarlılık modeli benimsenmiştir. Ardışıl tutarlılık yöntemi dağıtılmış nesne ortamlarında geniş bir kullanım oranına sahip, basit ve anlaşılır olan bir yöntemdir. Ayrıca, tutarlılık modelleri arasında yapısı MRSW senkronizasyon modeli ile birlikte gerçeklenmeye en uygun olan modeldir. Bu modele göre, kontrol nesnesi alt nesneyi erişim sırasında mutlaka tutarlı olarak görür.

Tutarlılık mekanizması içinde, bileşik nesne tasarımcısının seçimine bağlı iki farklı protokol sunulmuştur. Bunlar; *yazma-güncelleme* ve *yazma-geçersiz kılma* protokolleridir. Tasarımcı, sunulan protokollerden hangisinin kullanılacağına kontrol nesnesinin üretilmesi aşamasında karar verir ve kontrol nesnesinin sınıfı OSÜ tarafından bu tercihe bağlı olarak üretilir. Tasarımcı, uygulama gerektirdiği takdirde, başarımı yükseltmek amacıyla aynı DBN'nin farklı alt nesneleri için farklı protokoller de belirleyebilir.

Uygulanacak senkronizasyon ve tutarlılık mekanizmaları kontrol nesnesi içinde yer alan (Bkz. Şekil 4.8) ve ayrıntılı algoritmaları izleyen bölümlerde anlatılmış olan aşağıdaki metotlar tarafından gerçeklenirler.

```
public Object get_object(){...}
public void invalidate_object(){...}
public void update_object(Object sub_obj){...}
public Object copy_my_self(){...}
public void incoming_object(Object sub_obj){...}
public void give_object(int ref_dest,int kind){...}
public void ask_object(int kind){...}
public boolean ask_to_read(int ref_dest){...}
public void end_of_use(int ref_dest,int kind){...}
```

5.3.1 Eşzamanlılık Kontrolü ve Tutarlılık Algoritmaları

Alt nesne kopyalarına eşzamanlı olarak erişimin sağlanması ve kopyalar arasında tutarlılığın korunması için yürütülen işlemler, alt nesne üzerindeki tüm erişimleri denetleyen kontrol nesnesi tarafından yürütülür. Kontrol nesnesi eğer alt nesne üzerinde sahiplik konumunda ise, erişim denetimi işlemlerini kendi yürütür. Aksi takdirde, erişim isteğini uzak kontrol nesnesine iletir.

Bir alt nesnenin kopyaları üzerinde aynı anda yürütülmek istenen çağrılardan en az biri yazma erişimi ise, alt nesnenin bütünlüğünün bozulması muhtemeldir. Bundan dolayı, bir yazma erişimi tamamlanmadan, bir diğer yazma ya da okuma erişimi isteğine izin verilmez. Fakat, eğer yazma erişimli istek yok ise, çok sayıda alan üzerinde eşzamanlı okuma erişimleri yapılabilir. Alt nesne kopyaları arasında kurulan bu tip bir senkronizasyon denetimine *çoklu-okuyucu tek-yazıcı senkronizasyonu* adı verilir. Bu bölümde, söz konusu denetimleri gerçekleyen algoritmalar incelenecektir.

Tanımlamalar:

Bu bölümde, dağıtılmış bileşik nesneler üzerinde metot çağrıları yürüten uygulamalardan "istekçi" adı altında söz edilecektir. "Sunucu" adı ile de, her uygulamanın sistemle entegrasyonunu sağlayan "Dağıtılmış Bileşik Nesne Sunucusu" kastedilmektedir.

```
int         client_id;        // kontrol nesnesinin uzak erişimler için
                                 kullanacağı, bulunduğu alana ait
                                 istekçi tanımlayıcısı

int         obj_id;           // her türlü erişimde referans olarak
                                 kullanılan tekil nesne tanımlayıcısı

boolean     master;           // kontrol nesnesinin geçerli bir alt nesne
                                 kopyasına sahip olup/olmadığını
                                 gösteren mantıksal değişken

Vector  vctWaitingClients; // isteği karşılanamamış olan
                              istekçilerin beklemeye alındığı liste
```

```
Vector      vctReader;      // okuma erişiminde bulunan istekçilere
                               ait client_id bilgilerinin bulunduğu liste

int         nb_reader;      // okuma izni almış istekçilerin sayısı

boolean     is_writer;      // yazma izni almış istekçi olup/olmadığını
                               gösteren değişken, true ise, yazıcı var.

String      kind;           // erişim türü (R : okuma , W : yazma)
```

Alt nesne kopyaları üzerinde, "yazma-geçersiz kılma" ve "yazma-güncelleme" tutarlılık protokollerinden bileşik nesne tasarımcısı tarafından seçilen herhangi biri uygulanabilmektedir. Bu tercihin nasıl ifade edildiği, otomatik sınıf üretecinin yapısının incelendiği Bölüm 7'de açıklanmıştır. İzleyen bölümlerde her iki tutarlılık protokolüne ilişkin algoritma verilmiştir.

5.3.1.1 Yazma-Geçersiz Kılma Tutarlılık Protokolünü Kullanan Eşzamanlılık Kontrol Algoritması

Kopyalanmış bir alt nesnenin farklı alanlar üzerindeki kontrol nesnelerinden biri, o alt nesnenin sahibi olarak atanır. Sahiplik, ilk anda alt nesnenin yaratıldığı alanın kontrol nesnesinde iken, daha sonra alt nesne üzerinde yürütülen yazma erişimli çağrılar ile el değiştirir.

Algoritmaya geçmeden önce yazma-geçersiz kılma protokolü özetle incelenecek olursa; bu protokole göre, herhangi bir alanda bulunan kontrol nesnesinden gelen erişim isteği yazma türünde ise, alt nesnenin geçerli bir kopyasına sahip olan tüm diğer alanlardaki kontrol nesnelerine "*alt nesneyi geçersiz kıl*" çağrısı gönderilir ve yerel kopyaların artık güncel bir değer taşımadıkları bildirilir. Daha sonraki aşamalarda, kopyası geçersiz kılınmış bir alandan alt nesneye erişim isteği oluşunca, yerel kontrol nesnesi alt nesneye ait nesne tanımlayıcısı (*obj_id*) ve erişim isteği türü (*kind*) bilgileri ile, güncel bir kopya elde etmek üzere kendi sunucusuna başvurur Sunucu güncel bir kopyaya sahip olan kontrol nesnesini belirler ve ona ait sunucu aracılığı ile isteği söz konusu kontrol nesnesine iletir. Erişim isteği karşılandığı anda, güncel kopyanın alana yüklenmesi tamamlanmıştır. Bu amaçla gerçekleştirilen işlem

adımları aşağıda açıklanmıştır. Algoritma çoklu-okuyucu tek-yazıcı senkronizasyonunu kulanmaktadır.

Algoritmaya göre, kontrol nesnesi yerel alt nesne kopyası üzerinde bir metot çağrısı yürütmeden önce, `ask_object(kind)` çağrısı ile erişim için izin başvurusunda bulunur. Eğer metot çağrısı nesne durumu üzerinde değişiklik yapmayacak türden bir çağrı ise, çağrı parametresinin (`kind`) değeri R, aksi takdirde W olur. Bu çağrıdan geri dönüldüğünde, erişim için izin alınmıştır ve alt nesne üzerinde metot çağrısı yürütülür. `ask_object()` metodunun yapısı Şekil 5.2'de verildiği gibidir.

```
void ask_object(String kind) {
    if1 (sahiplik yerel kontrol nesnesinde)   // izin yerel olarak alınacak
        give_object(client_id, kind) ; //  yerel izin isteğinde bulun
     else1 {                                  //  sahiplik uzak kontrol nesnesinde
        if2 (yerel alt nesne kopyası geçerliliğini koruyor ve erişim isteği okuma)
            - okuma isteğini sahipliği elinde bulunduran uzak kontrol nesnesine ilet
        end-if2
        if3 (yerel kopya geçersiz veya okuma isteği için gelen yanıt olumsuz) {
            - bir thread yarat ve izin isteğini <obj_id, kind> çifti ile
              sahipliği elinde bulunduran uzak kontrol nesnesine ilet
            wait();                           //  yanıt gelene kadar bekle
            if4 (erişim isteği yazma) {
                master ← true;                //  sahiplik yerel kontrol nesnesinde
                nb_reader ← 0;                //  okuyucu sayısını sıfırla
                is_writer ← true;             //  yazıcının varlığını belirle
            } end-if4
        } end-if3
    } end-if1
} end-method
```

Şekil 5.2. `ask_object()` metodu içinde yürütülen işlem adımları

Alt nesneye erişim için yapılan izin çağrısı `ask_object()` metodunca ya yerel, ya da uzak kontrol nesnesinin `give_object()` metoduna aktarılır. Şekil 5.3'te algoritması verilen `give_object()` metodu, çoklu-okuyucu, tek-yazıcı çalışma düzenine uygun olan eşzamanlılık ve tutarlılıkla ilgili işlem adımlarını gerçekler.

```
void give_object(int ref_client, int kind) {
    if1 (gelen isteğin türü okuma) {          // okuma isteği için denetim yap
        if2 (yazıcı yok ve bekleyen istekler listesi boş) {
            - okuyucuların sayısını (nb_reader) bir arttır
            - yeni gelen istekçiyi okuyucular listesine (vctReader) ekle
            - alt nesneyi isteği yapan alana gönder
        } else2 isteği bekleyen istekçiler listesine (vctWaitingClients) ekle
        end-if2
    } else1    // gelen isteğin yazma isteği olması durumu
        if3 (yazıcı yok ve bekleyen istekler listesi boş) {
            is_writer ← true              // yazıcının varlığını belirle
            - okuyucular listesinde (vctReader) bulunan tüm istekçilere
              alt nesne kopyalarını geçersiz kılma mesajı gönder
            - okuyucular listesini boşalt
            - alt nesneyi istekte bulunan alana gönder
            sub_obj ← null;               // kendi kopyanı geçersiz kıl
            master ← false;               // sahipliği iptal et
        } else3 isteği ve türünü (ref_client, kind) bekleyen-istekçiler
                listesine ekle
        end-if3
    end-if1
} end-method
```

Şekil 5.3. `give_object()` metodu içinde yürütülen işlem adımları

Alt nesneye erişim için gerekli iznin alınması ile birlikte, o ana kadar yüklenmemişse bile, geçerli olan alt nesne kopyası isteği yapan alan üzerine yüklenir. Daha sonra, metot çağrısı "yerel olarak" yürütülür. Çağrının sonlanmasının ardından, çoklu-yazıcıyı tek-yazıcı eşzamanlılık kontrol algoritması gereği yerine getirilmesi gereken işlemleri `end_of_use()` metodu tarafından gerçeklenirler. Eğer alt nesnenin sahipliği yerel kontrol nesnesinde ise (`master` değişkeninin değeri "true"), kontrol nesnesinin `end_of_use(client_id, kind)` metodu yerel olarak çağrılır. Alt nesnenin sahipliğinin diğer bir alandaki uzak kontrol nesnesinde olması durumunda ise (`master` değişkeninin değeri "false"), yerel sunucu üzerinden uzak kontrol nesnesinin `end_of_use()` metoduna bir çağrı gönderilir. Söz konusu metot tarafından gerçeklenen işlem adımlarını içeren algoritma Şekil 5.4'te verilmiştir.

```
void end_of_use(int ref_client, int kind) {
    if1 (okuma işlemini tamamlamış bir okuyucu {
        - okuyucuların sayısını bir azalt
        if2 (okuyucu sayısı sıfır ve bekleyen istekçiler listesi boş değil) {
            // bu, en az bir yazma isteğinin sırada beklediğini gösterir
            - bekleyen istekçiler listesinin (vctWaitingClients)
              başındaki isteği çek              // bu bir yazıcıdır
            - okuyucular (vctReader) listesinde bulunan
              tüm istekçilere geçersiz kılma mesajı gönder
            - alt nesneyi ve bekleyen istekçiler listesini
              çekilen isteğin yapıldığı alana gönder
            - okuyucular listesini boşalt
            sub_obj ← null;                     // alt nesne kopyanı geçersiz kıl
            master ← false;                     // sahipliği iptal et
        } end-if2
    } else1 {                                   // yazma işlemini tamamlayan yazıcı
        is_writer ← false;
        if3 (bekleyen istekçiler listesi boş değil) {
            - bekleyen istekçiler listesinin başındaki isteği çek
            // çekilen istek bir okuyucu ya da bir yazıcı olabilir
            if4 (çekilen istek okuma isteği) {
                - alt nesneyi çekilen isteğin yapıldığı alana gönder
                - okuyucuların sayısını bir arttır
                while (bekleyen istekçiler listesinin başındaki
                                istek okuma isteği olduğu sürece) {
                    - bekleyen istekçiler listesinin başındaki isteği çek
                    - alt nesneyi çekilen isteğin yapıldığı alana gönder
                    - okuyucuların sayısını bir arttır
                } end-while
            else4 {                             // çekilen istek bir yazma isteğidir
                    - alt nesneyi ve bekleyen istekçiler listesini
                      çekilen isteğin yapıldığı alana gönder
                    sub_obj ← null;             // alt nesne kopyanı geçersiz kıl
                    master ← false;             // sahipliği iptal et
            } end-if4
        } end-if3
    } end-if1
} end-method
```

Şekil 5.4. `end_of_use()` metodu içinde yürütülen işlem adımları

Yukarıda verilen `give_object()` ve `end_of_use()` metotlarının içeriğinden de görülebileceği gibi, kontrol nesnesine alt nesnenin sahipliği de aktarılacaksa, bekleyen istekçiler listesi (`vctWaitingClients`) de alt nesne ile birlikte gönderilmektedir. Sunucu üzerinden uzak bir istekçinin kontrol nesnesine gönderilen alt nesne kopyası, o uzak kontrol nesnesinin `incoming_object()` metodu üzerinde yürütülen bir metot çağrısı ile iletilmektedir. `incoming_object()` metodun yürüttüğü işlem adımları Şekil 5.5'te verilmiştir.

```
void incoming_object(Object obj, Vector vct) {
      sub_obj ← obj;                  // gelen kopyayı alt nesne olarak ata
      vctWaitingClients ← vct;  // bekleyen istekler listesini oluştur
      if (alt nesne yapılan yazma isteği sonucunda gönderilmiş ise)
         master ← true;
      end-if
      notify(); //   wait() çağrısını yürüterek beklemeye alınmış olan
                    ask_object() metodunun kaldığı yerden
                    çalışmasına devam etmesini sağlar.
} end-method
```

Şekil 5.5. `incoming_object()` metodu içinde yürütülen işlem adımları

Yukarıda verilen `incoming_object()` metoduna ait algoritmadan da görülebileceği gibi, gelen alt nesne kopyası yazma isteğinin yanıtı olarak geliyorsa, `master` değişkeni "true" yapılarak sahiplik bu kontrol nesnesine aktarılır. Ayrıca, erişim denetimi de artık bu kontrol nesnesi tarafından yönetileceğinden, `vct` ile gelen liste de bekleyenler listesi (`vctWaitingClints`) olarak atanır.

5.3.1.2 Yazma-Güncelleme Tutarlılık Protokolünü Kullanan Eşzamanlılık Kontrol Algoritması

Algoritmaya geçmeden önce yazma-güncelleme protokolü özetle incelenecek olursa; bu protokole göre, herhangi bir alanda bulunan kontrol nesnesinden gelen erişim isteği yazma türünde ise, bu erişime izin verilirken, o anda alt nesnenin bir kopyasını içeren alanların listesi de istekte bulunan kontrol nesnesine gönderilir. Yazma erişimi iznini alan kontrol nesnesi alt nesne üzerinde çağrıyı yürüttükten sonra listede bulunan diğer alanlardaki kontrol nesnelerine alt nesnenin güncellenmiş kopyasını gönderir.

Yazma-geçersiz kılma ile yazma-güncelleme tutarlılık protokolleri arasındaki farklılıklar yalnızca `give_object()` ve `end_of_use()` metotlarındadır. Kontrol nesnesinin yazma-güncelleme tutarlılık protokolünü, gerçekleyen `give_object()` metodunun yapısı Şekil 5.6'da, `end_of_use()` metodunun yapısı da Şekil 5.7'de verilmiştir.

```
void give_object(int ref_client, int kind) {
    if1 (gelen isteğin türü okuma) {                    // okuma isteği için denetim yap
        if2 (yazıcı yok ve bekleyen istekler listesi boş) {
            - okuyucuların sayısını (nb_reader) bir arttır
            - yeni gelen istekçiyi okuyucular listesine (vctReader) ekle
            - alt nesneyi isteği yapan alana gönder
        } else2 isteği bekleyen istekçiler listesine (vctWaitingClients) ekle
        end-if2
    } else1     // gelen isteğin yazma isteği olması durumu
        if3 (yazıcı yok ve bekleyen istekler listesi boş) {
            is_writer ← true                            // yazıcının varlığını belirle
            - alt nesneyi ve okuyucular listesini istekte bulunan alana gönder
            sub_obj ← null;                             // alt nesne kopyanı geçersiz kıl
            master ← false;                             // sahipliği iptal et
        } else3 isteği ve türünü (ref_client, kind) bekleyen-istekçiler
                listesine ekle
        end-if3
    end-if1
} end-method
```

Şekil 5.6. Yazma güncelleme tutarlılık yöntemini kullanan algoritmanın `give_object()` metodu içinde yürütülen işlem adımları

```
void end_of_use(int ref_client, int kind) {
    if1 (okuma işlemini tamamlamış bir okuyucu {
        - okuyucuların sayısını bir azalt
        if2 (okuyucu sayısı sıfır ve bekleyen istekçiler listesi boş değil) {
            // bu, en az bir yazma isteğinin sırada beklediğini gösterir
            - bekleyen istekçiler listesinin (vctWaitingClients)
              başındaki isteği çek                    // bu bir yazıcıdır
            - alt nesneyi, bekleyen istekçiler ve okuyucular listeleri ile birlikte
              çekilen isteğin yapıldığı alana gönder
            master ← false;                           // sahipliği iptal et
        } end-if2
     } else1 {                                        // yazma işlemini tamamlayan yazıcı
        - okuyucular listesinde bulunan tüm istekçilere
          güncellenmiş olan alt nesne kopyasını gönder
        is_writer ← false;
        if3 (bekleyen istekçiler listesi boş değil) {
            - bekleyen istekçiler listesinin başındaki isteği çek
            // çekilen istek bir okuyucu ya da bir yazıcı olabilir
            if4 (çekilen istek okuma isteği) {
                - alt nesneyi çekilen isteğin yapıldığı alana gönder
                - okuyucuların sayısını bir arttır
                while (bekleyen istekçiler listesinin başındaki
                                istek okuma isteği olduğu sürece) {
                    - bekleyen istekçiler listesinin başındaki isteği çek
                    - alt nesneyi çekilen isteğin yapıldığı alana gönder
                    - okuyucuların sayısını bir arttır
                } end-while
             else4 {                                  // çekilen istek bir yazma isteğidir
                    - alt nesneyi, bekleyen istekçiler ve okuyucular listeleri ile birlikte
                      çekilen isteğin yapıldığı alana gönder
                    master ← false;                   // sahipliği iptal et
            } end-if4
        } end-if3
    } end-if1
} end-method
```

Şekil 5.7. Yazma güncelleme tutarlılık yöntemini kullanan algoritmanın `en_of_use()` metodu içinde yürütülen işlem adımları

Yazma-güncelleme tutarlılık protokolünün kullanıldığı kontrol nesnesinin `incoming_object()` metodu yazma-geçersiz kılma protokolünün kullanıldığı kontrol nesnesininkine göre küçük bir farklılık göstermektedir. Çünkü, yazma-güncelleme protokolünde, yazma izni verilen istekçi artık alt nesnenin sahipliğini de alacağı için, diğer kopyaların güncellenmesi işlemini de yürütecektir. Bu yüzden, üçüncü bir parametre olarak alt nesnenin kopyalarını içeren alanların tanımlayıcılarının bulunduğu `vctReader` listesi de çağrı sırasında yeni alana gönderilir. Bu durum Şekil 5.8'de gösterilmiştir.

```
void incoming_object(Object obj, Vector vct1, vct2) {
      sub_obj ← obj;
      vctWaitingClients ← vct1;
      vctAccesingServers ← vct2;
      if (alt nesne bir yazma isteğinin sonucunda gönderilmiş ise)
         master ← true;
      end-if
      notify();          // wait() çağrısını yürüterek beklemeye alınmış
                            olan ask_object() metodunun kaldığı
                            yerden çalışmasına devam etmesini sağlar.
} end-method
```

Şekil 5.8. Yazma güncelleme tutarlılık yöntemini kullanan algoritmanın `incoming_object()` metodu içinde yürütülen işlem adımları

6. DAĞITILMIŞ BİLEŞİK NESNE TABANLI ORTAM

Dağıtılmış ortamda müşterek olarak yürütülen grup çalışmalarını desteklemek amacıyla, büyük bir nesnenin daha alt seviyede çok sayıdaki küçük nesnenin bir araya gelmesi suretiyle oluştuğu ve grup içerisinde bu alt nesnelerin saydam bir şekilde paylaştırıldığı dağıtılmış bileşik nesne (DBN) modeli Bölüm 4'te incelenmişti. Bu bölümde de DBN modelini destekleyecek ve kullanıcıların merkezi bir ortamda çalışıyor görüntüsü altında dağıtılmış müşterek uygulamalar geliştirebilmelerine olanak sağlayacak olan Dağıtılmış Bileşik Nesne Tabanlı Ortam (DBNTO) ara katman yazılımının tasarım ve gerçekleme ayrıntıları incelenecektir.

6.1 DBNTO Tasarım Konuları

DBNTO tasarımında, nesne bileşimi yönteminden yola çıkarak, mevcut Java nesne modelinin kopyalanarak fiziksel olarak dağıtılmış paylaşılan nesnelere genişletilmesi için gerekli alt yapının nasıl geliştirilebileceği konusu üzerinde yoğunlaşılmıştır. Özellikle, internet üzerinde dağıtılmış müşterek grup çalışmalarının dağıtılmış bileşik nesneler kullanılarak kolaylıkla gerçeklenebilmesini sağlayacak uygun mekanizmalar aranmıştır.

Ele alınan ana tasarım konuları; fiziksel olarak dağıtılmış paylaşılan nesne yapısı için bir model oluşturma, sistemin dinamik olarak uyarlanabilmesi ve nesne yönetimi, şeklinde üç başlık altında incelenebilir.

6.1.1 Dağıtılmış Paylaşılan Nesne Yapısı İçin Bir Model Oluşturma

Kopyalanarak fiziksel olarak farklı alanlar üzerine dağıtılmış bir nesneyi gerçeklemenin bir yolu, ayrık parçaların bir üst seviye nesne içinde birleştirilerek, istekte bulunulan alanlarda bu üst seviye nesne ve sadece yapılan metot çağrısının gerektireceği ilgili alt nesnelerin kopyalarının çıkarılması şeklinde olabilir. Bu şekliyle bir dağıtılmış paylaşılan nesne, dağıtılmış bileşik nesne yapısı ile ifade

edilmiş olur. Bu durumda, bileşik nesne üzerindeki bir metot çağrısı alt seviyedeki bileşen nesneleri üzerinde yürütülecek bir veya bir dizi metot çağrısı şeklinde gerçeklenir.

6.1.2 Dinamik Uyarlama

Geniş alana yayılmış bir uygulamanın gerçeklenmesi için dağıtılmış bileşik nesneler kullanılacak ise, destek sistemi dinamik olarak yeni bileşenlerin eklenmesi veya var olanlardan bazılarının çıkarılması hususunda uygun mekanizmalar sağlamalıdır.

Dinamik uyarlama, büyük ölçekli geniş alana yayılmış uygulamalar için gerçeklenmesi güç bir özelliktir. Bu uygulamalar sadece uzun ömürlü uygulamalar olmayıp, aynı zamanda çok sayıda kullanıcı tarafından sürekli olarak kullanılan uygulamalardır.

Nesne bileşimi perspektifinden bakıldığında bu durum, kendi adres alanında dağıtılmış bileşik nesnenin bir kopyasına sahip olan bir uygulamanın, yeni bir "bileşen alt nesne" eklenmesi veya var olanlardan birinin çıkarılması gibi durumlardan haberdar edilmesi anlamına gelir.

Ayrıca, sisteme çalışma anında yeni uygulamalar eklenip, var olanların sistemi terk etmesi durumlarından diğerlerinin haberdar edilmeleri ve uygulamaların bu giriş/çıkışlardan etkilenmeden çalışmalarına devam edebilmeleri gerekir.

6.1.3 Nesne Yönetimi

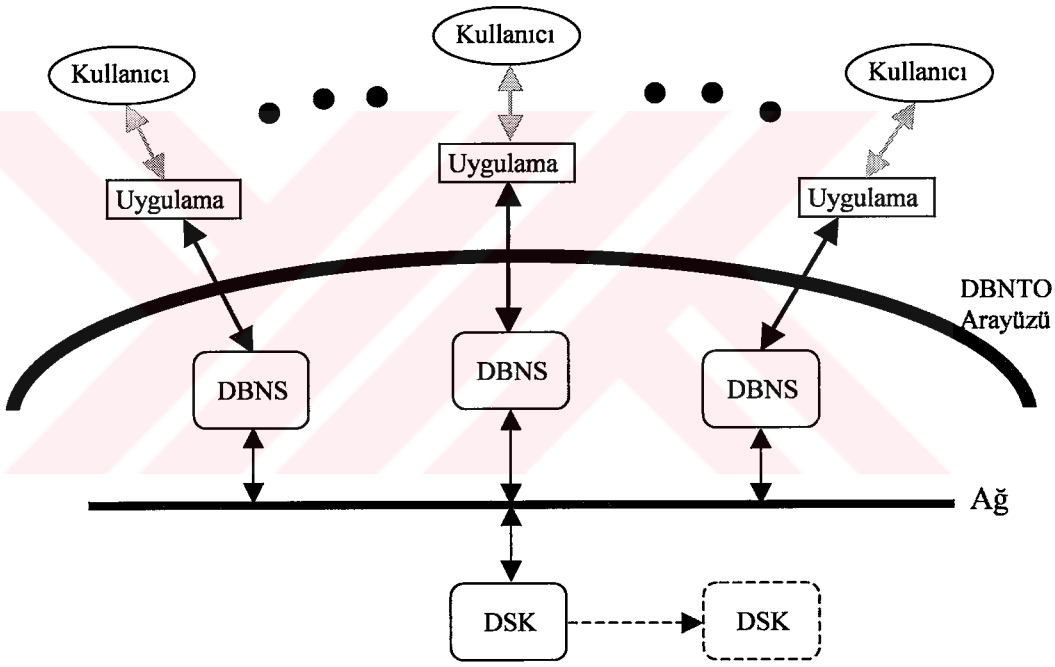
Nesne yönetimi konusunda ele alınması gereken ana problemler, *nesne adlandırma (object naming), dinamik olarak nesne bağlama (dynamic object binding),* farklı alanlar üzerinde kopyalanmış bulunan alt nesneler arasında tutarlılık yönetimi gibi konulardır. Önemli olan, bu konuların uygulama programcılarına saydam olarak gerçekleyebilecek çözümler bulmaktır.

Bir uzak adres alanındaki bileşik nesneye ait referansı dinamik olarak kendi adres alanına bağlayabilmek ve daha sonra bu nesnenin yerleşkesinin bulunabilmesi için her bir bileşik nesne sistem çapında tekil bir ad ile kayıt altına alınır.

Farklı alanlar üzerinde kopyalanmış bulunan bir alt nesnenin *eşzamanlı (concurrent)* erişimler sırasında bütünlüğünün sağlanabilmesi ve erişimin ardından kullanılan tutarlılık protokolü uyarınca diğer kopyaların da güncellenmesi veya geçersiz kılınması gibi işlemler sistem tarafından arka planda yürütülerek yerine getirilir.

6.2 Dağıtılmış Bileşik Nesne Ortamının Genel Yapısı

Dağıtılmış bileşik nesne modelini kullanan geniş alana yayılmış bir müşterek uygulamanın genel yapısı Şekil 6.1'de görülmektedir. Bu yapıda, dağıtılmış ortamda müşterek çalışmalar yürüten kullanıcılara ait uygulamalar DBNTO arayüzü üzerinden dağıtılmış bileşik nesneleri paylaşırlar.



Şekil 6.1. DBNTO sisteminin genel yapısı

Şekil 6.1'de yer alan her bir uygulama, *Dağıtılmış Bileşik Nesne Sunucusu (DBNS)* olarak adlandırılan, bir sistem birimi aracılığıyla dağıtılmış bileşik nesne ortamına erişir. Ayrıca, DBNS'lerin ortama giriş/çıkış işlemlerini denetleyen, onlara sistem çapında tekil bir tanımlayıcı atayan ve yeni yaratılan bir DBN'yi oluşturan her bir alt nesne için tekil bir nesne tanımlayıcısı sağlayan, bir *DBNTO Sistem Koordinatörü (DSK)* yer alır. Sistemin hata-hoşgörü oranını arttırmak için, Şekil 6.1'de kesikli çizgilerle gösterilmiş olan, ikinci bir DSK yedek sistem koordinatörü olarak görev

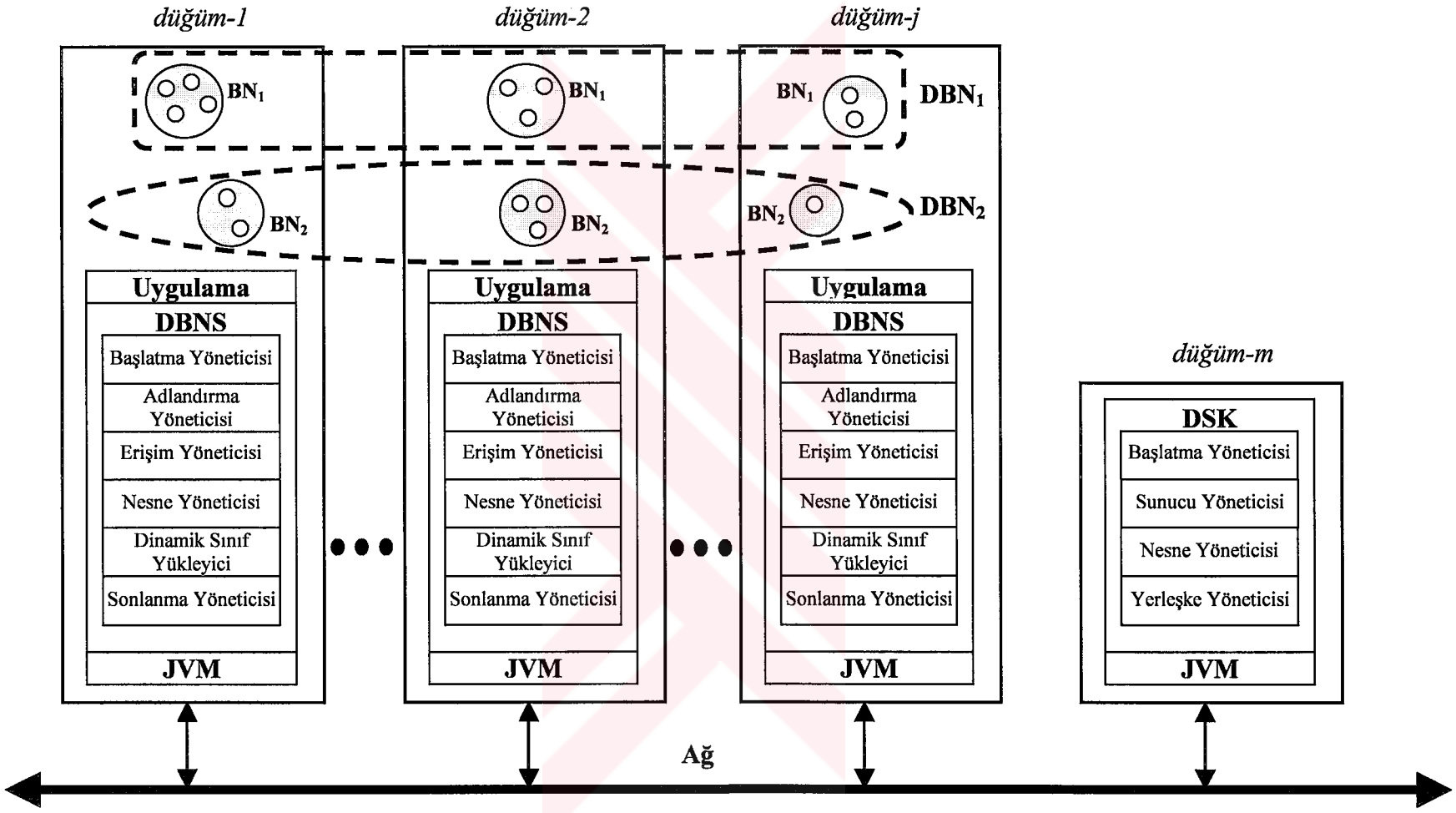
yapar. Herhangi bir şekilde asıl DSK'nin devre dışı kalması durumunda, yedek DSK üzerinden sistem problemsiz bir şekilde çalışmasına devam edebilmektedir.

Herhangi bir uygulama programı DBNS tarafından kendisine sunulan komut kümesini kullanarak dağıtılmış bileşik nesne ortamına giriş/çıkış yapma, yeni bir DBN yaratma ve kaydettirme, var olan bir DBN'nin sistemde *aranması (lookup)* ve kendi adres alanına yüklenmesi gibi işlemleri yürütür. DBN üzerinde bir metot çağrısı yürütüldüğünde, ilgili alt nesnenin/nesnelerin çağrının yapıldığı adres alanına taşınması, bütünlüğü bozmayacak şekilde çağrının gerçekleştirilmesi ve tutarlılığın sağlanmasına ilişkin ara işlem adımları da, bağlantı nesnesi, kontrol nesnesi ve ilgili DBNS'ler tarafından arka planda yürütülür.

Şekil 6.2'de iki farklı DBN'nin DBNTO üzerindeki dağılımı ve DBNTO yapısını oluşturan ana sistem elemanları olan dağıtılmış bileşik nesne sunucuları ve sistem koordinatörü ile bunların içinde yer alan sistem yöneticileri görülmektedir.

Bir bileşik nesne tekil bir adres alanı üzerinde bir uygulama tarafından yaratılıp sisteme kaydettirildikten sonra, diğer alanlardaki uygulamaların bu bileşik nesneyi kendi alanlarına yükleyip, nesne üzerinde metot çağrıları yürütmeleri sırasında bileşik nesnenin ilgili alt nesneleri de o alan üzerinde kopyalanırlar ve sonuçta Şekil 6.2'de görülen dağıtılmış bileşik nesne yapıları oluşur. Burada, BN_1 *düğüm-1* ve BN_2 *düğüm-2* üzerinde yaratılmış ve daha sonra, BN_1 *düğüm-2*'de üç alt nesnesi ile, *düğüm-3*'te iki alt nesnesi ile birlikte ve BN_2'de *düğüm-1*'de iki ve *düğüm-3*'te de bir alt nesnesi ile birlikte kopyalanmıştır.

Şimdi, DBNTO ara katman yazılımını oluşturan ana öğeler olan koordinatör ve sunucu birimlerinin yapıları ve işlevleri açıklanacaktır.



Şekil 6.2. DBNTO ve DBN yapısının genel görünüşü

6.2.1 DBNTO Sistem Koordinatörü

Dağıtılmış bileşik nesne ortamının hizmete hazır duruma getirilmesi için ilk olarak çalıştırılacak olan birim DBNTO Sistem Koordinatörü (DSK)'dür. Koordinatör ilk olarak daha sonra yaratılacak olan sunucuların da önceden bileceği belirli bir düğümde ve belirli bir *iskele (port)* numarası ile başlatılır.

Koordinatörün görevleri genel olarak şunlardır:

- Koordinatörün sunucular ile ve sunucuların da birbirleriyle doğrudan haberleşebilmeleri için, bir sunucunun bir diğerinden ayırt edilmesini sağlayacak, sistem çapında tekil, bir tanımlayıcı bilgisine ihtiyaç vardır. Bu amaçla, koordinatör ortama yeni katılan her bir sunucuya, ilk değeri sıfır olan basit bir tamsayı değişkenden türettiği, tekil bir *sunucu tanımlayıcısı (ser_id)* atar. Tanımlayıcı tek bir elden verildiği için, iki farklı sunucuya aynı değerin verilmesi söz konusu değildir. Bu yüzden, verilen tanımlayıcının sistem çapında tekil bir değer olması da garantilenir.

- Yeni yaratılan bir sunucunun erişim bilgilerini (düğüm adresi ve iskele numarası) sistemde var olan sunuculara ve eski sunucuların bilgilerini de yeni katılana aktararak, sunucular arasında doğrudan haberleşme alt yapısını hazırlar.

- Her bir alt nesneye sistem çapında tekil bir *nesne tanımlayıcısı (obj_id)* verir.

- DBN'lere ve onun alt nesnelerine ait *yerleşke* bilgilerini tutar.

Koordinatör içerisinde sistemin çalışmasını düzenleyen ve yukarıda sözü edilen görevleri yerine getirmesini sağlayan çeşitli yönetici birimleri bulunmaktadır. Bu birimler şu şekilde sınıflandırılabilir.

6.2.1.1 Başlatma Yöneticisi

Başlatma Yöneticisi (BY), koordinatör ilk olarak yaratıldığında bir kereye mahsus olmak üzere çalışır. Başlatma yöneticisi, koordinatörün dağıtılmış bileşik nesne ortamına daha sonra giriş yapacak olan sunuculara hizmet verebilecek duruma getirilmesi için gerekli olan işlem basamaklarını gerçekler. Bu amaçla, bulunduğu

düğümde önceden belirlenmiş bir iskele üzerinde bir *kayıt alanı (registry)* yaratır. Daha sonra, koordinatörü standart bir ad ile bu alana kaydeder. Böylece, ileride yaratılacak olan sunucular ilk işlem olarak önceden bilinen bu *<düğüm-adresi, iskele-numarası>* çiftinin tanımladığı kayıt alanı üzerinde yapacakları bir *arama (lookup)* işlemi ile koordinatöre erişim için gerekli olan bilgiyi alacaklardır.

6.2.1.2 Sunucu Yöneticisi

Sunucu Yöneticisi (SY), dağıtılmış bileşik nesne ortamına giriş yapan tüm sunuculara bir *sunucu tanımlayıcısı (ser_id)* verir. Sistemde daha sonraki tüm haberleşmelerde bu tanımlayıcı bilgisi kullanılır. SY, sistemdeki tüm sunuculara *ait <sunucu tanımlayıcısı, erişim nesnesi>* çiftini "sunucu erişim tablosu" adlı bir tabloda tutar. SY, <düğüm adresi, iskele numarası> çifti ile yeni bir sunucunun sisteme katılması durumunda;

- O sunucunun kayıt alanı üzerinde yapacağı bir arama işlemi ile elde edeceği erişim nesnesini sunucular tablosuna ekler ve yeni sunucuya bir *ser_id* verir.
- Yeni katılan sunucuya ait <sunucu tanımlayıcısı, erişim nesnesi> çiftini "sunucular" tablosunda yer alan diğer sunuculara ve tablodakileri de yeni katılan sunucuya aktararak, sistemdeki tüm sunucuların birbirleriyle doğrudan iletişim kurabilmeleri için gerekli alt yapıyı hazırlar.

6.2.1.3 Nesne Yöneticisi

Nesne Yöneticisi (NY)'nin iki görevi vardır. İlki, yeni yaratılmış olan bir DBN'yi oluşturan alt nesnelerin her birine ayrı ayrı, sistem çapında tekil bir *nesne tanımlayıcısı (ob_id)* vermektir. DBNTO sistemi içerisinde daha sonraki erişimlerde alt nesnelere bu tanımlayıcılar aracılığıyla erişilecektir. NY tüm alt nesnelere ait *ob_id* ve sahiplik bilgilerinin yer aldığı bir "alt nesne erişim tablosu" tutar. Bu tabloda, sistemde *obj_id* ile belirlenen bir alt nesnenin sahipliğinin hangi sunucu üzerinde bulunan kontrol nesnesinde olduğunu belirten *<obj_id, ser_id>* bilgi çiftleri yer alır.

NY'nin bir diğer görevi, DBN'yi oluşturan alt nesnelere yenilerinin eklenmesi veya var olanların çıkarılması durumlarında, o DBN'yi kullanmakta olan uygulamalara hizmet veren sunucuları değişiklikten haberdar etmektir. NY bu amaçla, bir DBN'ye kendi adres alanında sahip olan sunuculara ilişkin "DBN kullanıcıları tablosu" adlı bir tablo daha tutar. Bu tabloda her bir DBN'ye ait *<ad, DBN kullanıcıları listesi>* çifti yer alır.

6.2.1.4 Yerleşke Yöneticisi

Yerleşke Yöneticisi (YY) iki tip nesne için hizmet verir; DBN ve DBN'yi oluşturan alt nesneler. Sistemde var olan her bir DBN, kullanıcı tanımlı bir ad ve yaratıldığı alanın sunucusuna ait tanımlayıcı bilgisini içeren *<ad, ser_id>* çifti ile "DBN yerleşke tablosu"na kaydedilir. Daha sonra, bir DBN'yi yüklemek isteyen bir sunucunun yürütmüş olduğu arama işlemine karşılık, yerleşke yöneticisi DBN yerleşke tablosunda *ad*'a karşılık gelen sunucuya ait tanımlayıcı bilgisini geri döndürür.

Yerleşke yöneticisi, herhangi bir alt nesneye ait en geçerli kopyanın hangi alandaki kontrol nesnesinde bulunduğu bilgisini bir önceki bölümde sözü edilen "alt nesneler tablosu" üzerinde yapacağı bir arama işlemi ile belirler. Bu bilgiye "*sahiplik bilgisi*" adı verilir. Sahiplik bilgisi, alt nesne üzerinde en son güncelleme işlemi yürütmüş olan uygulamaya ait sunucunun kimlik bilgisidir. Farklı alanlardan gelen güncelleme istekleri sonucunda bir alt nesnenin sahipliği zaman içerisinde değişir. Yerleşke yöneticisinin bu konu ile ilgili olarak yürüttüğü işlemler;

- Herhangi bir sunucunun *obj_id* ile istekte bulunduğu, bir alt nesnenin sahipliğini elinde bulunduran kontrol nesnesinin sunucusuna ait *ser_id* değerini geri döndürmek,

- Bir alt nesnenin güncellenmesi durumunda, "alt nesneler tablosu" üzerinde *obj_id*'ye karşılık gelen *ser_id* değerini yenisi ile değiştirmektir.

Yerleşke yöneticisinin bir diğer görevi, dinamik sınıf yüklemesi için hizmet vermektir. Bir DBN sınıfının uzaktan dinamik olarak yüklenebilmesi için o sınıfın bir HTTP sunucusu ile kullanıma sunulması gerekir. Bu amaçla, yapılan işlemler Bölüm 6.2.2.5'te anlatılacaktır. Bu işlemler sırasında yerleşke yöneticisinin görevi,

uzaktan dinamik olarak sınıf yüklemesi yapmak isteyen bir sunucunun vereceği bir sınıf adına karşılık, o sınıfın yükleneceği HTTP sunucusunun adresini döndürmektir. Yerleşke yöneticisi bu amaçla "sınıf adresleri tablosu" adlı bir tablo tutar. Bu tabloda her bir DBN sınıfına ilişkin *<sınıf adı, sunucu adresi>* bilgi çifti yer alır. Uygulamalar yarattıkları farklı sınıftaki her bir DBN için, o DBN'nin türetildiği sınıf ait <sınıf adı, sunucu adresi> çiftini kendi sunucuları vasıtasıyla koordinatör üzerindeki sınıf adresleri tablosuna kaydettirir.

Koordinatör içinde yer alan yukarıda sözü edilen yöneticiler ve kullanılan veri yapıları ile sunuculara hizmet veren metot çağrıları Tablo 6.1'de görüldüğü gibidir.

Tablo 6.1. Koordinatör içinde bulunan yöneticiler, veri yapıları ve metot çağrıları

Yönetici Adı	Veri Yapısı	Metot Adı
Başlatma yöneticisi		DBNTO_system_coordinator(name)
Sunucu yöneticisi	sunucu erişim tablosu <ser_id, erişim nesnesi>	record_new_server(host, port) remove_server(ser_id)
Nesne yöneticisi	alt nesne erişim tablosu	register_new_control_object(ser_id) change_master(ser_id, obj_id) handle_get_objct(obj_id)
Yerleşke Yöneticisi	<obj_id, ser_id> DBN yerleşke tablosu <DBN_adı, ser_id> sınıf adresleri tablosu <sınıf adı, sunucu adresi>	get_control_object_owner(obj_id,ser_id,kind) register_DBN(dbn_name, ser_id) re_register_DBN(dbn_name, ser_id) lookup_DBN(dbn_name) add_new_DBN_user(dbn_name, ser_id) get_DBN_users_list(dbn_name) register_class(classname, classbase) load_class(classname) change_classbase(classname, classbase) find_class_server(classname)

6.2.2 DBNTO Sunucusu

Dağıtılmış bileşik nesne sunucusu (DBNS) uygulama programını DBNTO sistemine bağlayan ve sistemin karmaşıklığını ve ayrıntılarını kullanıcıdan gizleyen bir ara katman yazılımı olarak görülebilir. Bu amaçla, başlatılan her bir kullanıcı

uygulaması için bir sunucu yaratılır. Sunucu içerisinde sistemin çalışmasını düzenleyen çeşitli sistem yönetici birimleri bulunmaktadır.

6.2.2.1 Başlatma Yöneticisi

Başlatma Yöneticisi (BY), yeni yaratılan bir sunucunun sistemdeki diğer sunuculara ve kendi uygulamasına hizmet verebilecek duruma getirilmesi için gerekli olan işlem basamaklarını gerçekler. Bu amaçla;

- Yaratıldığı bilgisayar üzerinde boş bir iskele bulur ve bu iskele üzerinde bir kayıt alanı yaratır,
- Kendini önceden belirlenen standart bir ad ile bu alana kaydettirir,
- Kendini, düğüm adresi ve iskele numarası daha önceden belirlenen bir bilgisayar üzerinde yaratılmış olan koordinatöre kaydettirmek amacıyla, kayıt alanı üzerinde bir arama işlemi yaparak, koordinatöre erişim nesnesini elde eder.
- Koordinatörün sunucu yöneticisine bir çağrı göndererek (`record_new_server(host,port);`) kendini kaydettirdiği düğüm adresi ve iskele numarasını aktarır.

Daha önce de belirtildiği gibi, koordinatörün sunucu yöneticisi kendine gönderilen düğüm adresi ve iskele üzerindeki kayıt alanında yapacağı bir arama işlemi ile, o sunucuya erişim nesnesini elde eder ve yeni katılan sunucuya ait <ser_id, erişim nesnesi> çiftini diğer sunuculara da gönderir. Böylece, dağıtılmış bileşik nesne ortamının ana elemanları olan sunucular ve koordinatör birbirleriyle doğrudan haberleşebilme olanağına kavuşurlar.

6.2.2.2 Adlandırma Yöneticisi

Adlandırma Yöneticisi (AY), ait olduğu uygulama tarafından yaratılan DBN'lere ait ad bilgilerini yönetir. Bu amaçla, "DBN tablosu" adlı bir tablo tutar. Tabloda her bir DBN'ye ilişkin <*ad, nesne*> bilgi çifti yer alır. Adlandırma yöneticisi uygulama için iki tip hizmet sunar.

- Yeni bir DBN'nin tabloya kaydettirilmesi,
- Daha önce kaydettirilmiş olan bir DBN için tabloda arama yapma hizmetleridir.

Uygulama tarafından yaratılan bir DBN kullanıcı tanımlı bir *ad* ile kaydettirilmek istendiğinde, bu istek doğrudan sunucunun adlandırma yöneticisine gelir. Yönetici gelen *ad* ve kendi kimlik bilgisi ile koordinatörün yerleşke yöneticisine bir çağrı gönderir. Eğer bu ad altında daha önce bir DBN kaydedilmemiş ise, olumlu yanıt alınır ve bu yanıt üzerine <ad, nesne> çifti sunucu üzerinde kaydedilir.

Adlandırma yöneticisinin bir diğer görevi, uygulamasına bir DBN'yi kendi adres alanına bağlama konusunda hizmet vermektir. Bu amaçla, herhangi bir uygulama programı tarafından daha önce bir başka alan üzerinde yaratılmış olan bir DBN kullanılmak istendiğinde, uygulama tarafından bir arama komutu yürütülür. Bu komut adlandırma yöneticisi üzerinde bir metot çağrısına dönüşür. Daha sonra, adlandırma yöneticisi tarafından yürütülen işlemler şu şekildedir;

- Yönetici öncelikle, istekte bulunulan DBN'nin kayıtlı bulunduğu sunucunun kimlik bilgisini koordinatörün yerleşke yöneticisine yapacağı bir çağrı ile öğrenir,
- Daha sonra, ilgili sunucudan *ad*'a karşılık düşen DBN için istekte bulunarak nesneyi yükler.

6.2.2.3 Erişim Yöneticisi

Erişim Yöneticisi (EY), DBNTO sisteminde bulunan tüm sunucuların erişim bilgilerini tutar. Bu amaçla, her bir sunucuya ait *<ser_id, erişim nesnesi>* çifti "sunucu erişim tablosu" adlı bir tabloda tutulur. Yeni bir sunucunun sisteme eklenmesi veya var olan bir sunucunun sistemi terk etmesi durumlarında bu tablo üzerinde gerekli şekilde güncelleme işlemi yapılır. Tablo üzerinde sıklıkla yürütülen işlem, herhangi bir anda bir sunucuya çağrı yapılmak istendiğinde, *ser_id*'ye karşılık gelen erişim nesnesi için yapılan bir arama işlemidir.

6.2.2.4 Nesne Yöneticisi

Nesne Yöneticisi (NY) iki farklı nesne yapısı için hizmet verir:

- Bir DBN'yi oluşturan bileşen alt nesnelerinin sunucular arasında iletilmesi ve kopyalanması işlemleri,
- Sistemde var olan bir DBN'nin yapısında meydana gelen değişikliğin (DBN'ye yeni bir alt nesnenin eklenmesi veya var olanlardan bazılarının çıkarılması) onu kullanmakta olan diğer uygulamaların sunucularına bildirilmesi işlemidir.

Bir DBN'ye ait alt nesneler ve bunlara ilişkin kontrol nesneleri, nesne yöneticisi tarafından "alt nesneler tablosu" ve "kontrol nesneleri tablosu" adlı iki ayrı tabloda kaydedilirler. Kaydetme işlemi sırasında, her bir alt nesne için koordinatörden sistem çapında tekil bir *nesne tanımlayıcısı (obj_id)* alınır ve <obj_id, alt nesne> çifti alt nesneler tablosunda, <obj_id, kontrol nesnesi> çifti de kontrol nesneleri tablosunda kaydedilirler. obj_id bilgisi sadece o alt nesnenin değil, o alt nesneye erişimlerin sağlandığı bağlantı ve kontrol nesnelerinin de tanımlayıcısıdır. Nesne yöneticisi koordinatörden bir alt nesne için bir tanımlayıcı alırken, aynı anda koordinatör tarafından o alt nesneye ilişkin alan bilgisi de koordinatörün yerleşke yöneticisi tarafından kaydedilir. Böylece, bir çağrı yürütüldüğünde, eğer o çağrıya ilişkin kontrol nesnesi ve alt nesne çağrının yapıldığı alan üzerinde bulunmuyorsa, nesne yöneticisi aracılığıyla o alana yüklenmesi sağlanır. Bu işlemin nasıl gerçekleştirildiğine ilişkin ayrıntılar Bölüm 6.3.3'te açıklanmıştır.

Nesne yöneticisi tarafından sunulan bir diğer hizmet olan bir DBN'nin yapısında meydana gelen bir değişikliğin o DBN'yi kullanmakta olan uygulamalara bildirilmesi işleminde, nesne yöneticisi koordinatörün nesne yöneticisinden, kendi adres alanlarında o DBN'ye sahip olan sunucuların listesini alır. O sunucuların nesne yöneticilerine yapacağı çağrı ile değişikliği bildirir. Bu işlemlerin ayrıntıları da Bölüm 6.3.6.'da açıklanmıştır.

6.2.2.5 Dinamik Sınıf Yükleyici

Dinamik Sınıf Yükleyicisi (DSY), bir DBN'nin ait olduğu sınıf tanımlamasının çalışma anında uzak bir alan üzerinden dinamik olarak yüklenmesi işlevini yürütür.

Uzak bir alandan yüklenilmesine izin verilecek olan bir DBN ve ona ait bileşen nesnelere (her bir alt nesne ve onlara ait bağlantı ve kontrol nesneleri) ait sınıf dosyaları bir HTTP sunucusu vasıtasıyla hizmete sunulur.

DBN yaratılmadan önce, *<sınıf_adı, sunucu_adresi>* bilgi çifti (örneğin, Personal, http://guray.hho.edu.tr/my_classes/) uygulama programı tarafından yürütülen bir komutla (`register_class(Personal, http://guray.hho.edu.tr/my_classes/)`) önce sunucuya ve oradan da koordinatörün yerleşke yöneticisine aktarılır ve burada kaydedilir. Daha sonra, bir başka alandan ilgili DBN'yi yüklemek isteyen uygulamanın sınıf yükleme komutunu (`load_class("Personal");`) yürütmesi sonrası işlem adımları şu şekildedir:

Bu komut sunucunun dinamik sınıf yükleyicisi üzerinde bir metot çağrısına dönüşür (`load_class("Personal");`).

Dinamik sınıf yükleyici koordinatörün yerleşke yöneticisine yapacağı bir çağrı ile (`String classbase = get_class_server("Personal");`) uzaktan yükleme yapılacak olan HTTP sunucusunun adresini alır.

Sunucu, Java RMI'nın sunduğu *RMI sınıf yükleyici (RMIClassLoader)* mekanizmasını kullanarak `Class cl=RMIClassLoader.loadClass(classbase, "Personal");` çağrısı ile, dinamik olarak uzaktan sınıf yükleme işlemini gerçekleştirir.

Bir sınıf uzaktan dinamik olarak bu şekilde yüklendiğinde, o sınıf içinden erişilen tüm diğer sınıflar da daha sonra gerektiğinde otomatik olarak yüklenirler.

6.2.2.6 Sonlanma Yöneticisi

Sonlanma Yöneticisi (SY), uygulaması sona eren bir sunucunun DBNTO sisteminden tümüyle çıkarılması için gerekli olan işlemleri gerçekler. Sunucu bu maksatla koordinatöre ve diğer sunuculara DBNTO sistemini terk ediyor olduğunu bildiren bir çağrı gönderir. Ancak, o an sahipliği o sunucu üzerinde bulunan alt nesneler varsa, her bir nesnenin sahipliğinin alt nesneyi kullanmakta olan bir diğer sunucuya aktarılması gerekir (daha önce de belirtildiği gibi, alt nesne üzerinde en son güncelleme işlemi yürütmüş olan uygulamaya ait kontrol nesnesi, o alt nesnenin

sahibidir. Sahiplik zaman içerisinde bir sunucudan diğerine geçerek el değiştirir). Eğer sahipliğin verileceği sunucu o an geçerli bir kopyaya sahip değilse, o sunucu üzerinde bulunan alt nesne kopyasının da güncellenmesi gerekmektedir. Bu işlemler şu şekilde gerçekleştirilir:

- Çalışmakta olan bir uygulama sona erdirildiğinde, uygulamayı sonlandırma komutu sunucuya bir çağrı ile (`leave_system()`) bildirilir.
- Bu durum koordinatörün sunucu yöneticisine bir çağrı ile (`remove_server(ser_id)`) bildirilir.
- Koordinatör sonlanmak üzere olan sunucunun sahipliği altında bulunan alt nesne olup/olmadığını "alt nesne erişim tablosu"ndan kontrol eder ve bu durumdaki alt nesneleri sunucuya bildirir (`change_object_owner(obj_id)`).
- Sonlanma yöneticisi obj_id'ye karşılık gelen kendi alanındaki kontrol nesnesine bir çağrı yaparak (`Vector list = get_users_list()`), o an bu alt nesnenin bir kopyasına sahip sunuculara ait ser_id listesini alır.
- Sonlanma yöneticisi listeden çekilen bir sunucuya sahipliği aktarmak için gerekli olan çağrıyı (`change_master(ser_id,obj_id)`) yürütür. Çağrı sonlandığında koordinatörde yer alan alt nesne erişim tablosunda obj_id'ye karşılık gelen ser_id değeri yenisi ile değiştirilir.
- Sonlanma yöneticisi tüm sunuculara sistemden ayrılıyor olduğu çağrısını yapar (`remove_server(ser_id)`). Çağrıyı alan sunucular da kendi sunucu erişim tablolarından ilgili alanı silerler.

Bu işlemlerin yürütülmesine ait ayrıntılar Bölüm 6.3.5'te açıklanmıştır.

Dağıtılmış bileşik nesne sunucusu içinde yer alan yöneticiler ve kullanılan veri yapıları ile sunucunun koordinatöre ve yerel kontrol nesnesine hizmet veren metot çağrıları Tablo 6.2'de görüldüğü gibidir.

Tablo 6.2. Bir dağıtılmış bileşik nesne sunucusu içinde bulunan yöneticiler, veri yapıları ve metot çağrıları

Yönetici Adı	Veri Yapısı	Metot Adı
Başlatma yöneticisi		DBNTO_server(host) ya da DBNTO_server(host,port)
Adlandırma yöneticisi	DBN tablosu <ad, nesne>	Register(dbn_name, ser_id) Lookup(dbn_name) get_object(dbn_name, ser_id) re_register(dbn_name, ser_id) inform_object_change(dbn_name)
Erişim yöneticisi	sunucu erişim tablosu	get_port_number() add_new_server(ser_id) change_master(ser_id, obj_id)
Sonlanma yöneticisi	<ser_id, erişim nesnesi>	leave_system() remove_server(ser_id) change_object_owner(obj_id)
Nesne yöneticisi	alt nesneler tablosu <obj_id, alt nesne> kontrol nesneleri tablosu <obj_id, kontrol nesnesi>	register_new_control_object(ser_id) get_control_object (obj_id) ask_object(obj_id, kind) ask_object_read(obj_id) ask_right_to_read(obj_id , ser_id) record_comming(sub_obj, obj_id) record_leaving(obj_id) get_all_objects() invalide_all_readers(vctReader, obj_id) invalide_reader(obj_id) update_all(sub_obj, obj_id, vctReader) do_update(sub_obj, obj_id) end_reading(obj_id) release_object(obj_id) give_object(ser_id, kind , obj_id) send_to_client(ser_id, obj_id, sub_obj, vctWaitingClients, vctReader, kind) send_to_control_object(sub_obj, obj_id, vctWaitingClients, vctReader)
Uygulamaya sunulan metotlar		register_class(classname, classbase) load_class(classname) change_classbase(classname, classbase)

6.3 DBNTO Sisteminde Yürütülen DBN İşlemleri

DBNTO sistemi içerisinde çeşitli kullanıcı gruplarının, farklı uygulamaları çalışıyor olabilir. Bu gruplar çalışmaları sırasında DBNTO sisteminin aşağıda sıralanan altı temel hizmetini kullanırlar. Bu hizmetler sırasıyla;

- Yeni bir uygulamanın sisteme katılması,
- Yeni bir DBN'nin yaratılması ve kayıt ettirilmesi,
- DBN üzerinde bir metot çağrısının yürütülmesi,
- Uzak bir uygulamanın bir DBN'yi yüklemesi,
- Bir uygulamanın sona ermesi,
- Bir DBN'ye yeni alt nesne(ler) eklenmesi veya var olanlardan bir kısmının çıkarılması,

başlıkları altında incelenebilir.

6.3.1 Yeni Bir Uygulamanın Sisteme Eklenmesi

Bir DBNTO uygulamasının nasıl başlatıldığına ilişkin işlem adımları Şekil 6.3'te görülmektedir. Buradaki uygulama programı içerisinde

```
DBNTO_Server DBNTO_server = new DBNTO_Server();
```

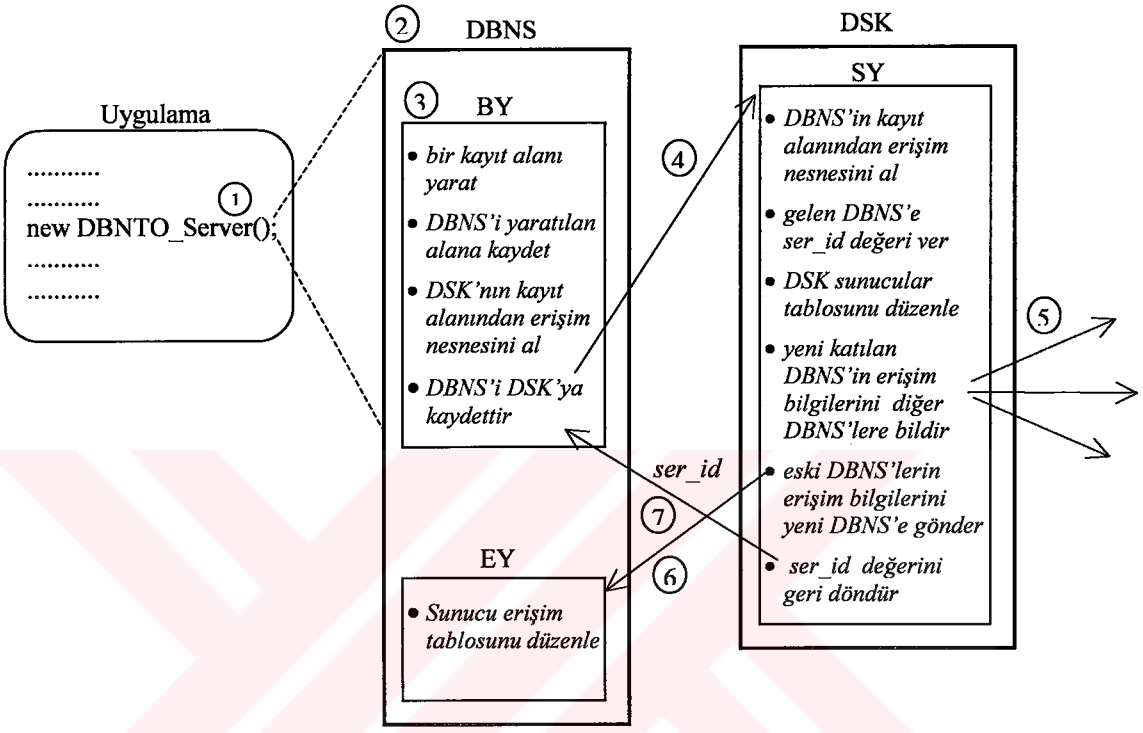
komutu yürütüldüğünde (1), o uygulamanın DBNTO sisteminin bir üyesi olmasını sağlayacak olan sunucu nesnesi yaratılmış olur (2). Sunucu yaratıldığında öncelikle, başlatma yöneticisi çalışır (3). Bu yönetici sırası ile şu işlemleri yürütür:

(i) Çalıştığı bilgisayarda ve boş olan bir iskele üzerinde bir kayıt alanı yarat.

(ii) Sunucuyu yaratılan kayıt alanına kaydettir.

(iii) Sunucuyu DBNTO sistem koordinatörüne kaydettir (4).

Koordinatörün hangi makinada ve iskele üzerinde kayıtlı olduğu belli olduğu için, sunucu bu <makine adresi, iskele numarası> çifti üzerinde yapacağı arama işlemi ile koordinatöre erişim nesnesini elde eder ve son olarak kendisinin kaydedilmiş olduğu <makine adresi, iskele numarası> çifti ile koordinatörün sunucu yöneticisine bir çağrı gönderir.



Şekil 6.3. Yeni bir uygulamanın DBNTO sistemine katılması

Koordinatörün çağrıyı alan SY sırasıyla şu işlemleri yürütür;

(i) Çağrı parametresi olarak gelen <makine adresi, iskele numarası> üzerinde bir arama işlemi ile DBNTO sistemine yeni katılan sunucuya erişim nesnesini al.

(ii) Yeni katılan sunucuya ait erişim bilgisini (ser_id, erişim nesnesi çifti) sisteme daha önce eklenmiş olan tüm sunuculara ilet (5).

(iii) Sisteme daha önce eklenmiş olan tüm sunucuların erişim bilgilerini yeni katılan sunucuya ilet (6).

(iv) Yeni sunucuya ait erişim bilgisini "sunucu erişim tablosu"na ekle.

(v) Yeni katılan sunucuya *ser_id* değerini döndür (7).

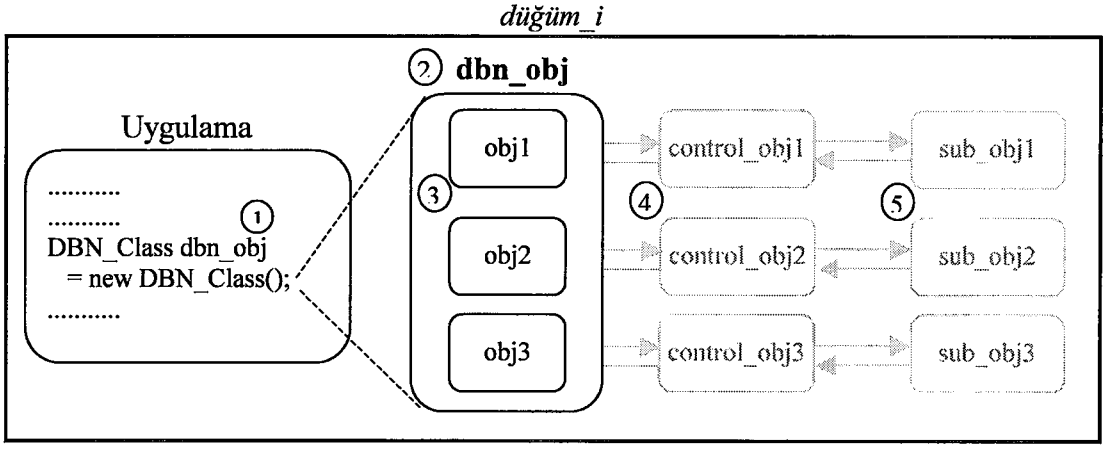
Bu işlemler tamamlandıktan sonra, sistemdeki tüm sunucular ve koordinatör birbirleriyle doğrudan haberleşebilecek duruma gelirler.

6.3.2 Bir DBN'nin Yaratılması ve Kayıt Ettirilmesi

Bir uygulama içerisinden *dbn_obj* adlı yeni bir DBN yaratıldığı farz edilsin. *dbn_obj* nesnesinin üç alt nesnesi olsun (*sub_obj1, sub_obj2, sub_obj3*). Bu alt nesnelere erişimi denetleyen üç tane de kontrol nesnesi olacaktır (*control_obj1, control_obj2, control_obj3*). Ayrıca, DBNTO sisteminde serbestçe kopyalanabilen bu nesneleri bağlama (binding) amacıyla üç tane de bağlantı nesnesi kullanılır (*obj1, obj2, obj3*). Daha önce Bölüm 4'de de belirtildiği gibi, *dbn_obj* dağıtılmış bileşik nesnesi içerisinden doğrudan erişilen nesneler bağlantı nesneleridir ve alt nesnelere erişim de yalnızca kontrol nesneleri üzerinden yapılabilir.

Örnekte yer alan *dbn_obj* nesnesi *DBN_Class* sınıfına, *obj(i)* nesneleri *Class(i)* sınıflarına, *control_obj(i)* nesneleri *Control_Class(i)* sınıflarına ve son olarak *sub_obj(i)* nesneleri de *Sub_Class(i)* sınıflarına aittirler. Nesneyi tasarlayan kullanıcı sadece *Sub_Class(i)* ve *DBN_Class* sınıflarını hazırlamakta, *Class(i)* ve *Control_Class(i)* sınıfları bir *Otomatik Sınıf Üreteci* tarafından olarak üretilmektedir.

Uygulama programcısı tarafından hazırlanan şekliyle *dbn_obj*'nin görüntüsü Şekil 6.4'te sunulduğu gibidir. Programcı *DBN_Class* sınıfının içine yalnızca bağlantı nesnelerinin üretileceği bağlantı sınıflarını (*Class(i)*) alır. Şekil6.4'te gösterilen silik renkli kısımlar *DBN_Class* sınıfının *kurucu (constuctor)* metodu çalıştıktan sonra otomatik olarak oluşur.



Şekil 6.4. Bir DBN'in yaratıldığı andaki şematik görünüşü

Şekil 6.4'teki uygulamanın içerisinde

```
DBN_Class dbn_obj = new DBN_Class();
```

komutunun yürütülmesi ile birlikte (1), *dbn_obj* nesnesi yaratılır (2). *dbn_obj* nesnesi yaratılırken ilk olarak, *DBN_Class*'ın kurucu metodu çalışır ve bu metot içinde de aşağıda belirtildiği gibi bağlantı nesnelerinin yaratılma işlemleri yürütülür (3).

```
Class1 obj1 = new Class1();
Class1 obj2 = new Class2();
Class1 obj3 = new Class3();
```

obj(i) nesnesi yaratıldığında bu nesnenin kurucu metodunun çalışması ile *control_obj(i)* nesnesi (4) ve *control_obj(i)* nesnesi yaratıldığında da bu nesnenin kurucu metodunun çalışması ile birlikte *sub_obj(i)* nesneleri yaratılır (5). Tüm bu işlemlerin nasıl gerçekleştiği ve bu sırada uygulama, sunucu ve koordinatör arasında gerçekleştirilen çağrılar Şekil 6.5'te görüldüğü gibidir (bu noktadan itibaren kolay anlaşılabilmesi için sadece *obj1* üzerindeki işlem adımları gösterilecektir, aynı işlemler *obj2* ve *obj3* için de geçerlidir).

dbn_obj nesnesinin kurucu metodunun çalışması ile birlikte *obj1* nesnesi yaratıldığında, ilk olarak obj1'in de kurucu metodu çalışır ve

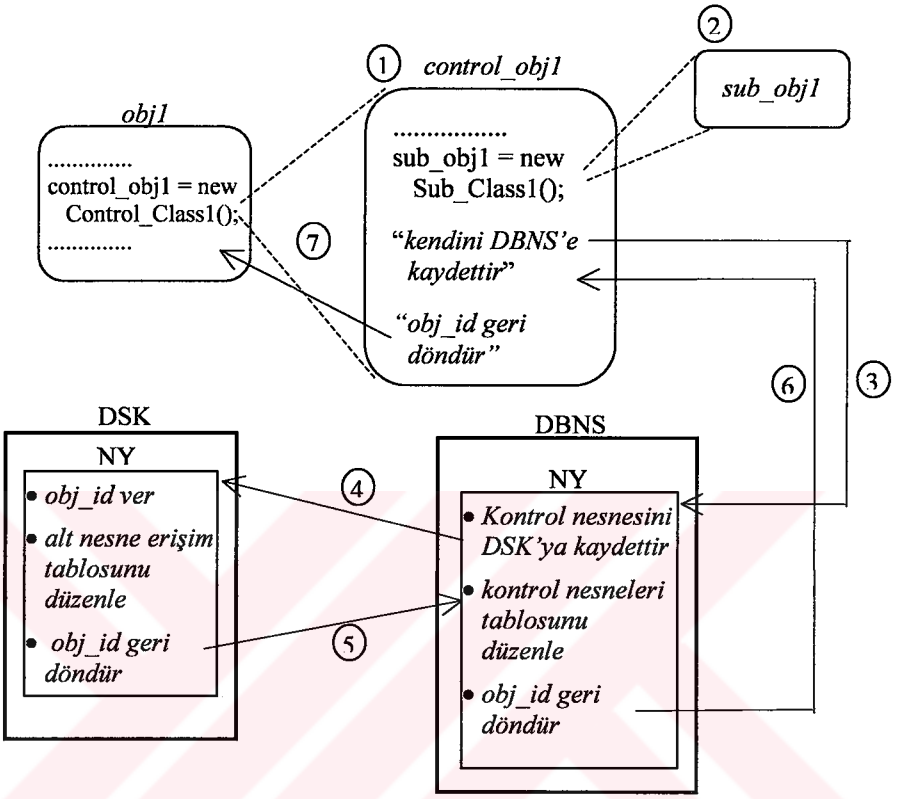
```
Control_Class1 control_obj1 = new Control_Class1();
```

komutunun yürütülmesi ile birlikte *sub_obj1* alt nesnesinin kontrol nesnesi olan *control_obj1* nesnesi yaratılır (Şekil 6.5 (1)). Bu andan itibaren işlem akışı *control_obj1* nesnesine geçecektir. Bu nesnesinin kurucu metodunun çalışması ile

```
Sub_class1 sub_obj1 = new Sub_Class1();
```

komutu yürütülür ve *sub_obj1* nesnesi yaratılır (2). Daha sonra yaratılan kontrol nesnesi ve alt nesne sunucuya kaydettirilir. Bu amaçla, *control_obj1* içinden sunucunun nesne yöneticisine bir çağrı gönderilir (3). Çağrıyı alan NY, kontrol nesnesini koordinatörün NY'sine kaydettirmek için bir çağrı üretir (4). Bu çağrı

üzerine koordinatörün NY'si gelen nesne için bir *obj_id* verir ve nesnenin ait olduğu sunucu ile en yeni kopyanın bulunduğu sunucu bilgilerini (başlangıçta her ikisi de aynıdır) kaydeder ve *obj_id* bilgisi sırası ile sunucuya (5), buradan *control_obj1*'e (6) ve son olarak *obj1*'e (7) geri döndürülür. Bu andan itibaren yapılacak tüm nesne işlemlerinde bu *obj_id* değeri referans olarak kullanılır.



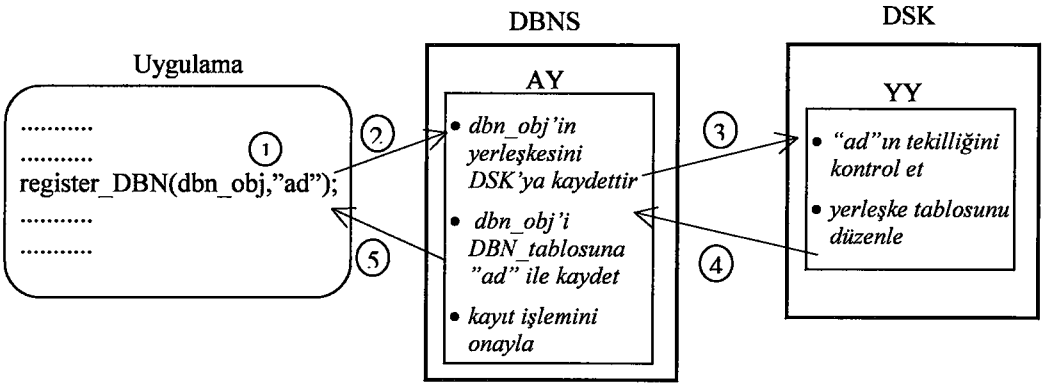
Şekil 6.5. Bir alt nesnenin yaratılması sürecindeki işlem adımları

Yukarıda anlatıldığı şekilde dağıtılmış bileşik nesne tek bir alan üzerinde bileşen nesneleri ile birlikte yaratıldıktan sonra, Şekil 6.6'da görüldüğü gibi, kullanıcı tanımlı bir *ad* ile kayıt ettirilir. Bu amaçla, uygulama programı içinden

```
register(dbn_obj,"ad");
```

komutu yürütüldüğünde (1), sunucunun adlandırma yöneticisi (AY)'ne bir çağrı yapılır (2). Çağrıyı alan AY öncelikle koordinatörün yerleşke yöneticisi (YY)'ne bir çağrı gönderir (3). Bu çağrı alındığında öncelikle bu isim altında daha önce bir başka nesnenin kaydedilip/kaydedilmediği kontrolü yapılır ve ardından nesnenin yerleşke bilgisi kaydedilir. (Böylece, ileriki aşamalarda *dbn_obj* bileşik nesnesini yüklemek isteyen bir sunucuya doğrudan nesnenin bulunduğu sunucunun *ser_id* değeri

döndürülecektir.) Eğer daha önce bu isim altında bir bileşik nesne kaydedilmemiş ise, sunucuya olumlu mesajı döndürülür (4). Sunucuya olumlu mesajın geri dönmesi ile birlikte, *dbn_obj* nesnesi *"ad"* isim bilgisi ile birlikte kaydedilir. Son olarak, kayıt işleminin başarılı/başarısız olarak tamamlandığı bilgisi geri döndürülür (5).

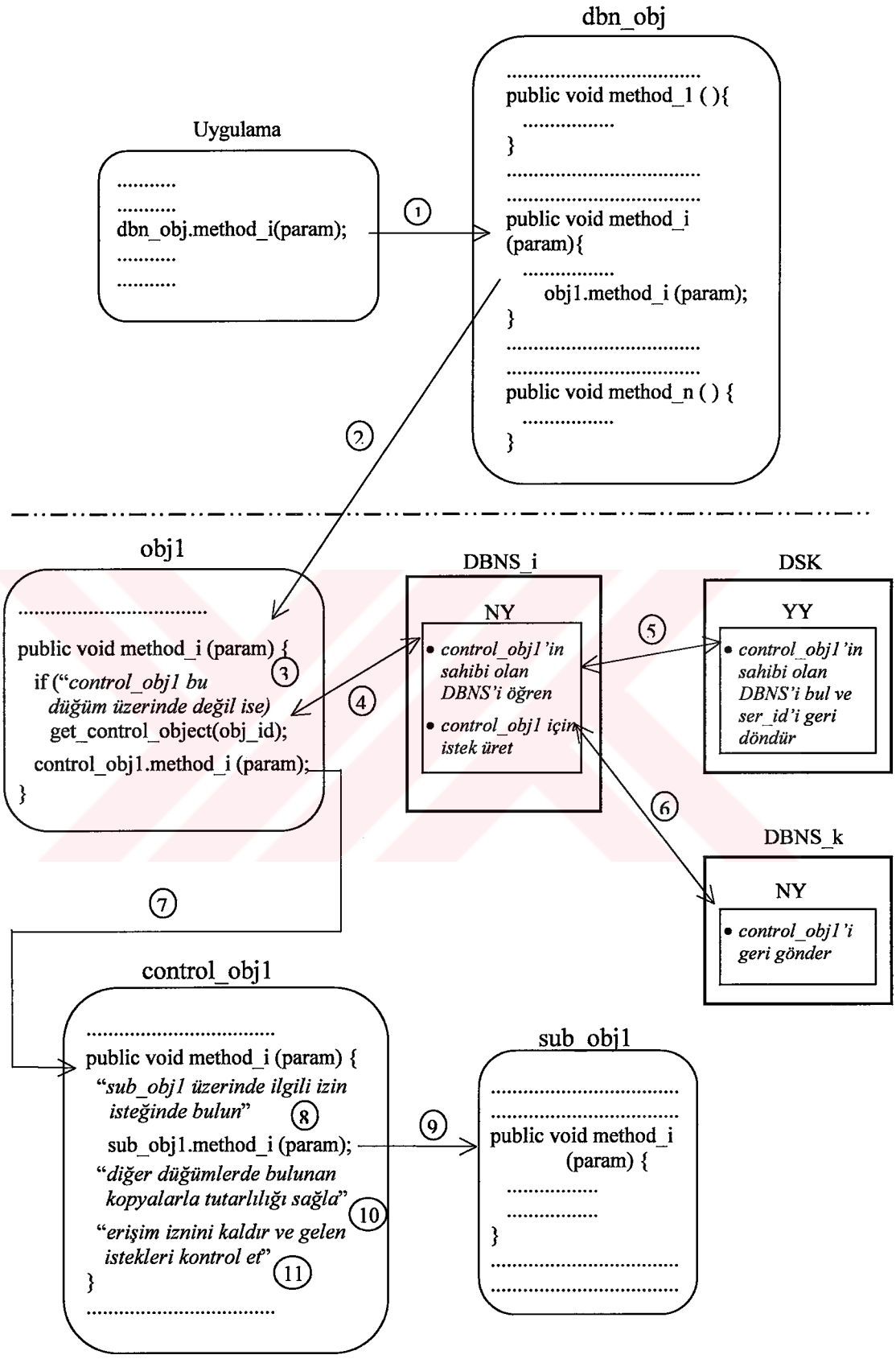


Şekil 6.6. Yeni yaratılan bir DBN'nin bir ad ile kaydedilmesi

6.3.3 DBN Üzerinde Bir Metot Çağrısının Yürütülmesi

Bu bölümde DBN üzerinde yapılan bir metot çağrısının nasıl gerçeklendiği incelenecektir. Eğer metot çağrısı DBN'nin üzerinde yaratıldığı yerel uygulamadan değil de, diğer bir alandaki uygulamadan yapılıyor ise, öncelikle DBN'nin sınıf yüklemesinin yapılması ve ardından da DBN'nin kendisinin yüklenmesi gerekir. Bu işlemlerin nasıl gerçeklendiği Bölüm 6.3.4'te açıklanacaktır.

Bir DBN üzerinde yürütülen metot çağrısının gerçeklenme adımları Şekil 6.7'de görülmektedir. Uygulama programı içinden bir önceki adımda yaratılmış olan *dbn_obj* nesnesi üzerinde bir metot çağrısı yürütüldüğünü (1) ve bu çağrının *sub_obj1* alt nesnesine ait bir çağrı olduğunu düşünelim. Bu durumda, çağrı doğrudan *sub_obj1* alt nesnesinin bağlantı nesnesi olan *obj1* üzerinde yürütülen bir çağrıya dönüşür (2).



Şekil 6.7. DBN üzerinde bir metot çağrısının gerçeklenmesi

Bir DBN istek üzerine bir diğer alan üzerinde kopyalandığında sadece bağlantı nesneleri ile kopyalanır. Yani, kontrol nesneleri ve alt nesneler kopyalanmaz. O alandan herhangi bir alt nesneye ait bir metot çağrısı gelmesi durumunda, alt nesneye erişimin sağlandığı kontrol nesnesi ve güncel alt nesne kopyası çağrının yapıldığı alana yüklenir. Bundan dolayı *obj1* içinde öncelikle kontrol nesnesinin o alana daha önceki çağrılar sonucunda yüklenip/yüklenmediği kontrolü yapılır (3). Eğer yüklenmemiş ise, sunucuya yapılan `get_control_object(obj_id);` çağrısı ile kontrol nesnesi ve alt nesne istenir (4). Çağrıyı alan sunucu, *obj_id* ile belirtilen kontrol nesnesinin şu anki sahibini (bir alt nesneyi en son güncellemiş olan alandaki kontrol nesnesi o alt nesnenin sahibidir) koordinatörün yerleşke yöneticisine bir çağrı göndererek öğrenir (5). Çağrı sonucu geri dönen *ser_id* bilgisi, güncel alt nesne kopyasına sahip olan sunucunun kimlik bilgisidir. Böylece, kontrol nesnesinin istekte bulunacağı sunucu belirlenmiş olur. Bu sunucunun nesne yöneticisine yapılan

```
get_control_object(obj_id);
```

metot çağrısı sonucunda *control_obj1* ve *sub_obj1* nesneleri yeni alana yüklenir (6). Daha sonra, *obj1* bağlantı nesnesi

```
control_obj1.method_i(param);
```

çağrısını yürütür (7).

sub_obj1 alt nesnesi dağıtılmış ortamda paylaşımlı kullanılan bir nesne olduğundan dolayı, *control_obj1* içinde öncelikle *sub_obj1* için gerekli erişim izni (okuma/güncelleme) alınır (8). Metot çağrısının yürütülmesinin ardından (9), eğer nesne durumunda bir değişiklik yapılmış ise, diğer nesne kopyaları (kullanılan tutarlılık yöntemine göre) geçersiz kılınır veya güncellenir (10). Son olarak, eğer bu arada çağrı isteğinde bulunmuş, fakat isteği, yürütülmekte olan istekle uyuşmadığından dolayı beklemeye alınmış olan diğer istekler de incelenerek, mümkün olanların yürütülmesi sağlanır (11).

6.3.4 Bir Uzak Uygulamanın Bir DBN'yi Yüklemesi

DBNTO sisteminde, bir uygulamanın bir başka uygulama tarafından daha önceden yaratılmış olan bir DBN üzerinde metot çağrısı yürütebilmesi için, kendi adres alanı içinde o DBN'ye ait kabuk nesneye sahip olması gerekmektedir. Bu da, o DBN'ye ait sınıf tanımlamalarının daha önceden çağrının yürütüleceği alanda yer almasını gerektirir. Farklı iki yaklaşım ile durum çözümlenebilir. İlki, farklı alanlarda yaratılmış olan DBN'lere ait sınıf tanımlarının uygulama başlatılmadan önce statik olarak yeni alana yüklenmesidir. Diğeri ise, söz konusu olan sınıf tanımlamalarının çalışma anında, dinamik olarak, uzaktaki alan üzerinden yeni alana yüklenmesidir. İlk yöntem kullanıcıya bir ek yük getirdiği ve saydamlık ilkesiyle çeliştiği için bu çalışmada ikinci çözüm yolu benimsenmiştir.

Bir uzak uygulamanın bir DBN'yi yüklemesi konusu dinamik olarak uzaktan sınıf yükleme ve nesne yükleme olarak iki alt başlık altında incelenecektir.

6.3.4.1 Dinamik Olarak Uzaktan Sınıf Yükleme

DBNTO sisteminde herhangi bir uygulamanın bir DBN'yi kullanabilmesi için, DBN ve bileşen nesnelerine ait sınıf tanımlamalarının çalışma anında dinamik olarak uzaktaki bir alandan yüklenebildiği yukarıda belirtilmişti. Bu yapının gerçeklenmesi Java RMI'nın sunduğu *dinamik sınıf yükleme (dynamic class loading)* özelliği ile sağlanmıştır.

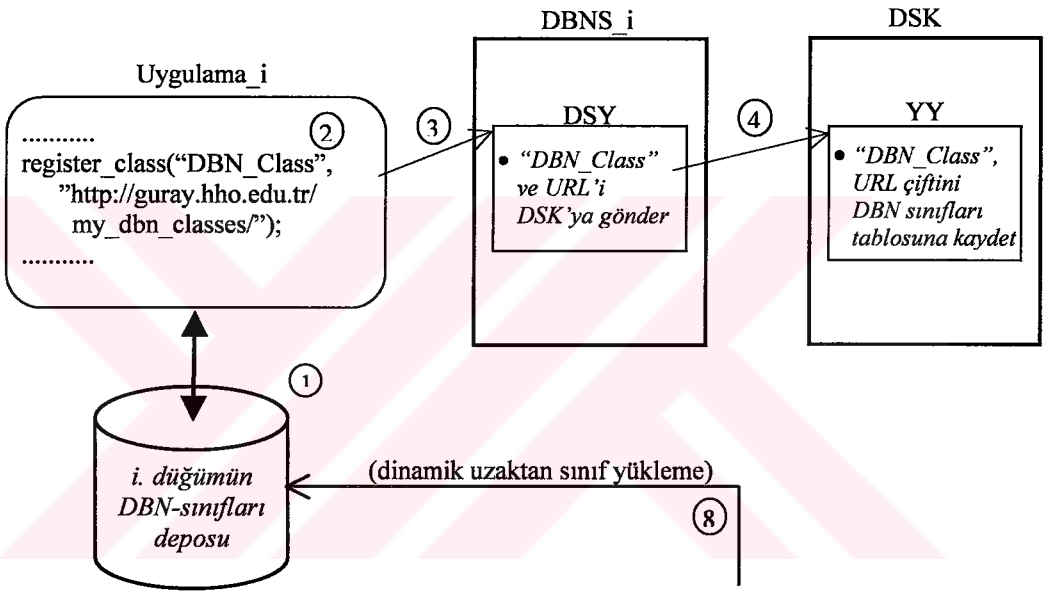
Dinamik sınıf yüklemesinin nasıl yapıldığına ilişkin işlem adımları Şekil 6.8'de görülmektedir. Şekil 6.8 (a) yeni bir DBN yaratıp, sisteme ekleyen bir uygulamanın DBN sınıfını sisteme tanıtmasına ilişkin işlem adımlarını gösterirken, Şekil 6.8 (b) daha sonra bu DBN'yi kendi adres alanına yüklemek isteyen bir başka uygulamanın sınıf yüklemesini nasıl yapacağına ilişkin işlem adımları göstermektedir.

Şekil 6.8 (a) incelendiğinde, DBNTO sistemi içinde yer alan bir uygulama tarafından yaratılan bir DBN'ye ve DBN içinde bulunan bileşen nesnelerine ait sınıf dosyaları bir HTTP sunucusu ile diğer alanlardaki uygulamalar için yüklenebilir hale getirilir (1). Bu noktadan itibaren konunun daha iyi anlaşılması için yukarıda verilen örnek tekrar ele alınacak ve *DBN_Class, Class(i), Control_Class(i) ve Sub_Class(i)* sınıf dosyaları, http://guray.hho.edu.tr/my_dbn_classes/ adresinden bir HTTP sunucusu ile

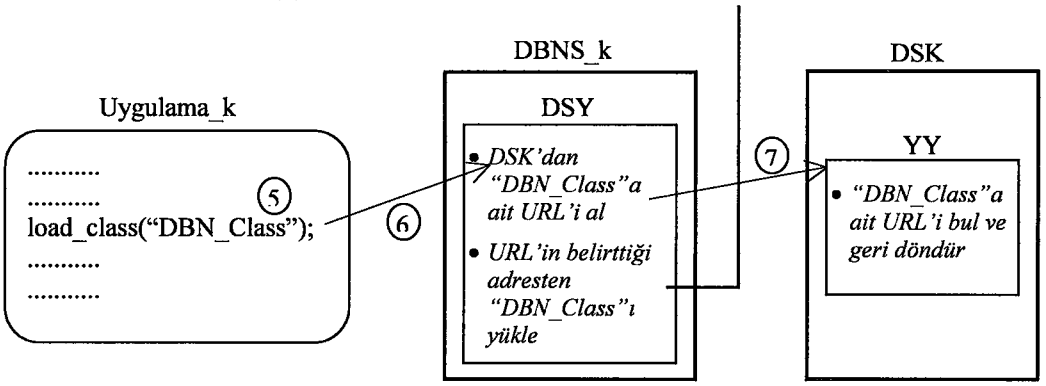
erişilebilir hale getirildiği varsayılacaktır. Bu durumda, uygulama programı içinde *DBN_Class* sınıfından *dbn_obj* kabuk nesnesi yaratılmadan önce, *DBN_Class* sınıf adı ve uzak kullanıma sunulduğu sunucu adresine ait <*sınıf_adı,URL*> çifti;

```
register_class("DBN_Class",
        "http://guray.hho.edu.tr/my_dbn_classes/");
```

komutu ile (2) sunucuya aktarılır (3). Sunucu gelen bilgiyi aynı şekliyle koordinatörün yerleşke yöneticisine iletir (4). Böylece, DBNTO sisteminde bulunan tüm DBN'lere ait sınıf kümelerine hangi sunucu üzerinden erişilebileceği bilgisi koordinatörde saklanır.



(a) : Yeni bir DBN sınıfının sisteme tanıtılması.



(b) : Dinamik olarak uzak bir alan üzerinden sınıf yüklemesinin yapılması.

Şekil 6.8. Bir DBN sınıfının sisteme tanıtılması ve uzak bir alandan dinamik olarak sınıf yüklenmesi

Herhangi bir zamanda, bir uygulama bir *dbn_obj* nesnesini yüklemek istediğinde, öncelikle *DBN_Class* sınıf tanımlamasını yüklemesi gerekir. Bu amaçla uygulama içerisinden,

```
load_class("DBN_Class");
```

komutu yürütüldüğünde (5), sınıf yükleme isteği sunucuya iletilir (6). Sunucu öncelikle, *DBN_Class* sınıfının yükleneceği HTTP sunucusunun adresini koordinatörden öğrenir (7) ve ardından, Java RMI'nın sunduğu dinamik sınıf yükleme mekanizmasını kullanarak yükleme işlemini gerçekleştirir (8). Daha sonra, *DBN_Class* içinden erişilen sınıflara ait yüklemeler gerektiğinde, bu yükleme işlemleri de dinamik sınıf yükleyici tarafından otomatik olarak aynı adresten yapılmaktadır.

Görüldüğü gibi, uzaktan sınıf yüklemesi gerçekleştirmek isteyen bir uygulamanın yalnızca yükleyeceği sınıfın adını vermesi yeterlidir. Geri kalan tüm ara işlemler sistem tarafından çözümlenmektedir.

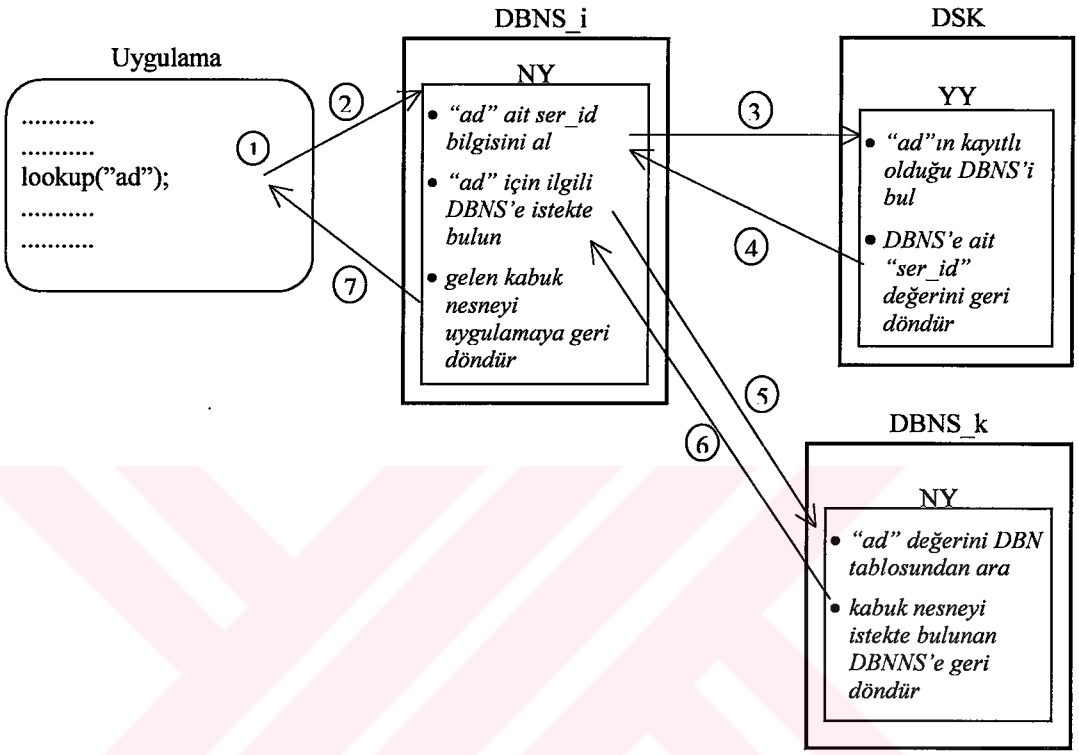
6.3.4.2 Bir DBN'nin Uzak Bir Uygulama Tarafından Yüklenmesi

Her hangi bir uygulamanın daha önceden yaratılmış olan bir DBN'yi kullanabilmesi için öncelikle, eğer yerel olarak sahip değilse, o DBN'nin türetildiği sınıfı, bir önceki bölümde anlatıldığı şekilde, yüklemesi gerekir. Daha sonra yer alması gereken, DBN'yi temsil eden kabuk nesnenin yüklenmesine ilişkin işlem adımları Şekil 6.9'da gösterilmiştir. Uygulama programcısı bu amaçla yapacağı yalnızca yüklemek istediği DBN'nin adını vererek bir arama işlemi gerçekleştirir. Uygulama programı içerisinden,

```
lookup("ad");
```

komutu yürütüldüğünde (1), bu komut sunucunun adlandırma yöneticisi üzerinde bir metot çağrısına dönüşür (2). Çağrıyı alan yönetici, öncelikle *"ad"* isimli DBN'nin hangi sunucuda kayıtlı olduğunu öğrenmek amacıyla koordinatörün yerleşke yöneticisine bir çağrı gönderir (3). Koordinatör *ad*'a karşılık gelen nesnenin kayıtlı olduğu sunucunun *ser_id* bilgisini yerleşke tablosundan bulur ve sunucuya gönderir

(4). Sunucu *ser_id*'ye karşılık gelen hedef sunucunun nesne yöneticisine bir çağrı ileterek *ad* için istekte bulunur (5). Çağrıyı alan hedef sunucu, *ad*'a karşılık gelen kabuk nesneyi kendi alanında kayıtlı bulunan DBN'ler tablosundan bularak, çağrıyı yapan sunucuya geri döndürür (6). Son olarak DBN sunucu tarafından uygulama programına aktarılır (7).

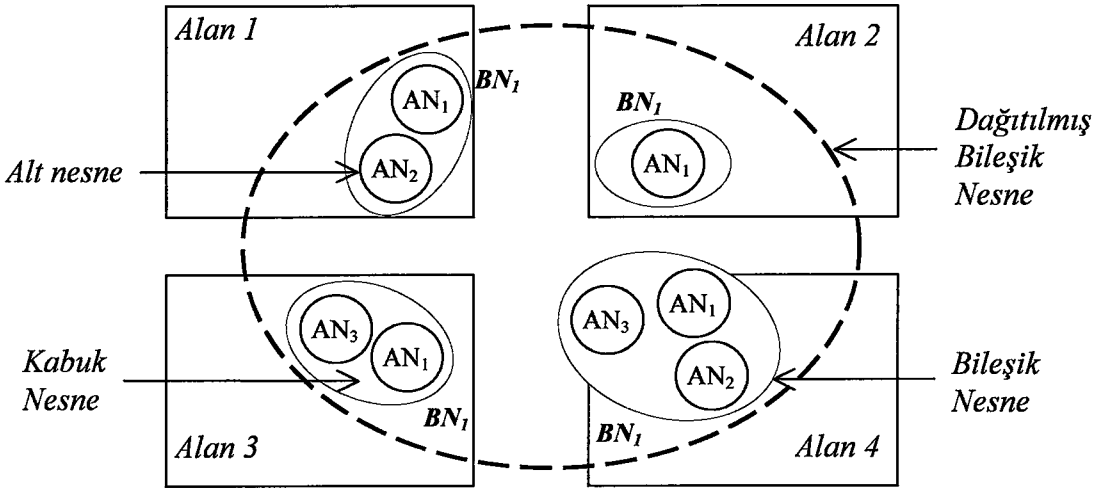


Şekil 6.9. Bir DBN'nin uzak bir alan üzerinden yüklenmesi

Yukarıda açıklanan işlem adımları sonucunda kabuk nesne ile birlikte, kontrol nesnelerini ve alt nesneleri bağlamak için kullanılan bağlantı nesneleri de yeni alana kopyalanırlar. Daha sonra, bu uygulama içinden yapılacak olan metot çağrılarına göre, yalnızca ilgili alt nesneler ve bunların kontrol nesneleri yeni alan üzerinde kopyalanacaktır (bu işlemlere ilişkin ayrıntılar Bölüm 4 ve Bölüm 5'te incelenmişti). Bu duruma göre bir DBN, Şekil 6.10'da da görüldüğü gibi, parçalı olarak kopyalı bir bileşik nesne görüntüsü kazanır

Şekil 6.10'daki dağıtılmış bileşik nesne, öncelikle *Alan 4* üzerinde üç alt nesnesi ile birlikte yaratılmış (BN_1); daha sonra, *Alan 1* üzerinde AN_1 ve AN_2 alt nesneleriyle, *Alan 2* üzerinde sadece AN_1 alt nesnesiyle ve *Alan 3* üzerinde de AN_1 ve AN_3 alt nesneleriyle kopyalanmıştır. Alt nesnelerin bu şekildeki dağılışları tamamıyla ilgili

alanlarda çalışmakta olan uygulamalardan yapılan metot çağrıları sonucu oluşmaktadır.



Şekil 6.10. Üç alt nesneye sahip bir DBN'nin dört alan üzerine farklı alt nesneleriyle kopyalanmış şematik görüntüsü

6.3.5 Bir Uygulamanın Sona Ermesi

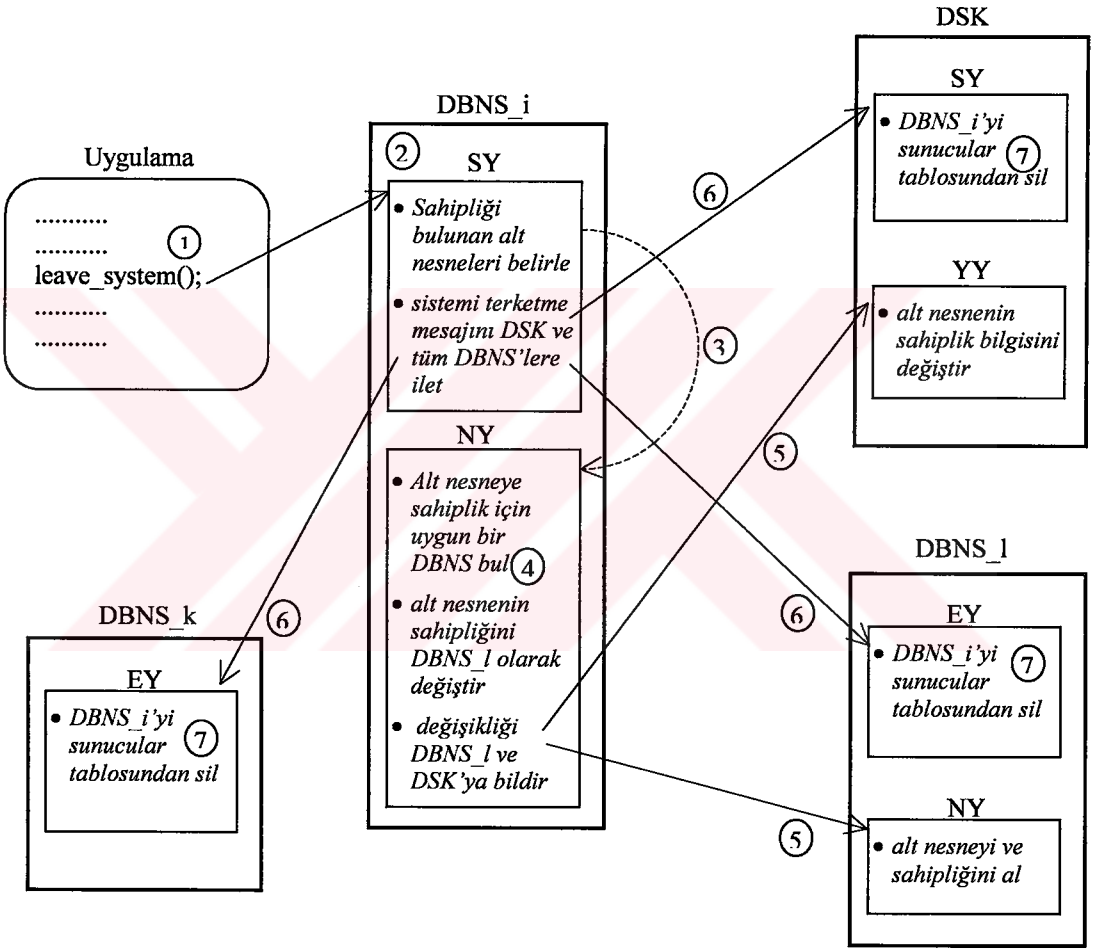
Dağıtılmış bileşik nesne ortamında bulunan bir uygulama sonlanırken, bu uygulamayı DBNTO sistemine bağlayan sunucunun da sistemden çıkarılması gerekmektedir. Bu amaçla, sunucunun diğer sunucuların ve koordinatörün kayıtlarından silinmesi yeterli değildir. Eğer sistem çapına yayılmış olan DBN alt nesnelerinden bazılarının sahiplik görevi o anda sistemi terk edecek olan sunucu üzerinde bulunuyorsa, o sunucu sahip olduğu her bir alt nesnenin sahiplik görevini sistemde ilgili alt nesneyi kullanmakta olan bir diğer sunucuya aktarmalıdır. Bu işlemin ardından DBNTO sistemini terk edecek olan sunucu, durumu tüm sunuculara ve koordinatöre göndereceği bir çağrı ile bildirerek sistemi terk edebilir.

Bir uygulamanın sona erdirilmesi ile ilgili yukarıda kısaca açıklanan işlem adımları Şekil 6.11'de gösterilmiştir. Buna göre, sistemi terk etmek isteyen uygulama,

```
leave_system();
```

komutunu yürüterek (1), isteğini sunucunun sonlanma yöneticisine iletir (2). Çağrıyı alan yönetici öncelikle kendi nesne yöneticisine göndereceği bir çağrı ile sahiplik

konumundaki alt nesnelerin belirlenmesini bildirir (3). NY, sahiplik hakkı o sunucuda bulunan alt nesneleri teker teker belirleyerek, her biri için belirlenen alt nesnenin bir kopyasına sahip diğer sunuculardan herhangi birini sahiplik için seçer (4) (örneğin DBNS_k). Daha sonra, DBNS_k'ye yapacağı bir çağrı ile alt nesnenin sahipliğini verir (5). Bu noktada, eğer kullanılmakta olan tutarlılık protokolü yapılan bir güncelleme sonucunda diğer kopyaların geçersiz kılınmasını gerektiren "yazma geçersiz kılma" protokolü ise ve DBNS_k da o anda geçersiz bir kopyaya sahip ise, güncel kopyayı DBNS_i'den ister.



Şekil 6.11. Bir uygulamanın DBNTO sistemini terk etmesine ilişkin işlem adımları

DBNS_i'ye ait nesne yöneticisinin yürüttüğü bu işlemler zinciri sahiplik konumundaki tüm alt nesneler için yapılır. Bu işlemlerin ardından, sonlanma yöneticisi DBNS_i'nin sistemden ayrıldığı mesajını tüm diğer sunuculara ve koordinatöre iletir (6). Bu mesajı alan sunucular ve koordinatör, DBNS_i'yi kendi sunucular tablolarından silerler (7).

6.3.6 Bir DBN'ye Yeni Alt Nesneler Eklenmesi veya Var Olanlardan Bir Kısmının Çıkarılması

Belirli zamanlarda bir DBN'yi oluşturan alt nesnelere yenilerinin eklenmesi veya var olan alt nesnelerden bazılarının çıkarılması gerekebilir. Bu şekildeki bir değişikliği yalnızca o DBN'yi yaratmış olan uygulamanın yapması söz konusudur. Çalışma sırasında böyle bir durum söz konusu olduğunda, öncelikle o uygulama kapatılır. Daha sonra DBN'nin türetildiği sınıf yapısı yeni duruma göre tekrar düzenlenir ve bu düzenlemenin ardından uygulama tekrar başlatılır.

DBNTO sisteminin normal çalışma şeklinde, bir bileşik nesneyi yaratan uygulamanın sistemden ayrılması, o bileşik nesnenin ve sisteme yayılmış olan bileşen nesnelerinin kopyalarının sistemden çıkarılmasını gerektirmez. Diğer uygulamalar o bileşik nesne üzerinde çalışmalarına devam edebilirler.

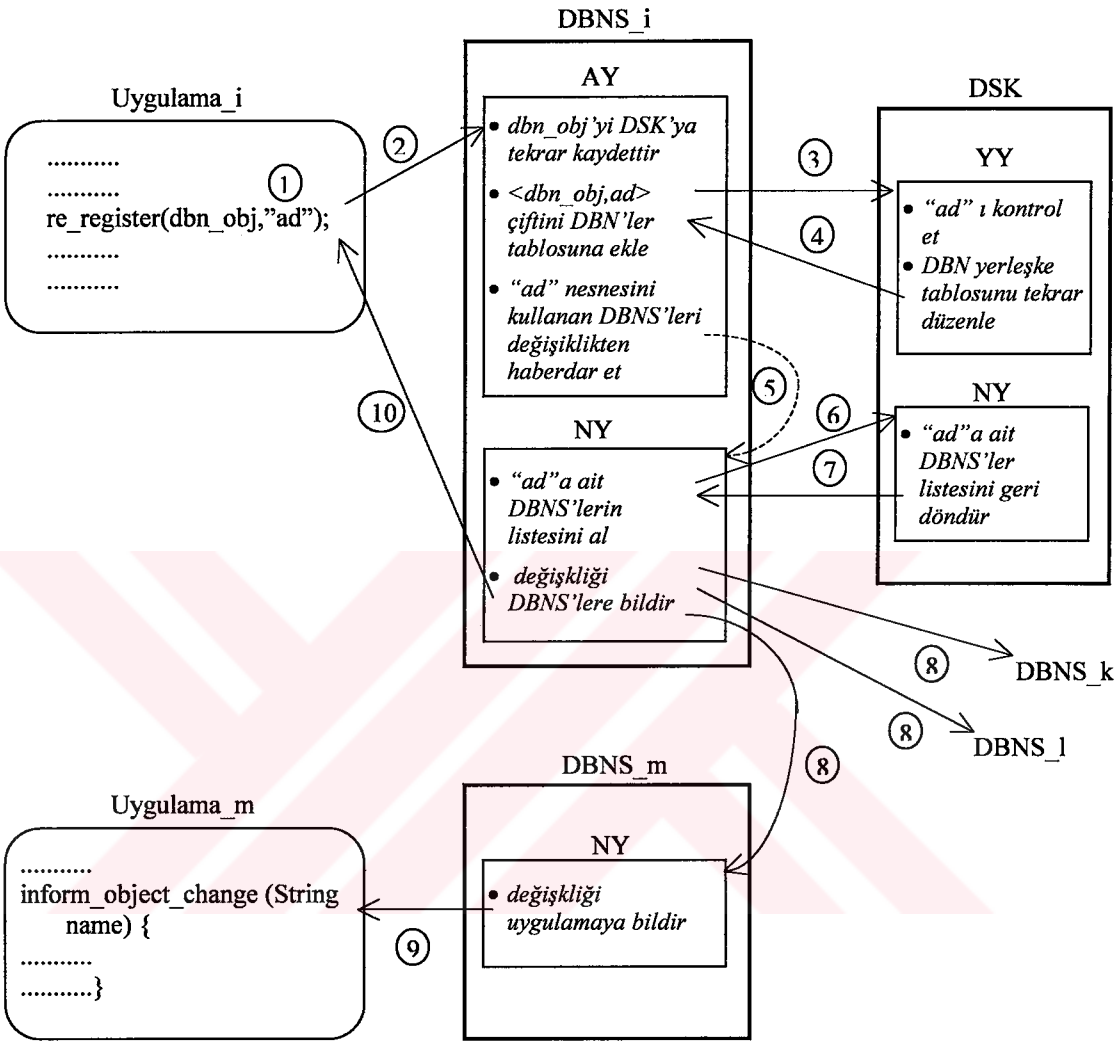
Eğer bir DBN'nin yapısında değişiklik yapılacak ise, uygulama yukarıda açıklandığı şekilde sonlandırılıp, DBN'nin türetildiği sınıf yapısının tekrar düzenlenme işleminin ardından uygulama tekrar başlatıldığında, değişikliğin ilgili DBN'yi kullanmakta olan diğer uygulamalara bildirilmesi gerekmektedir. Bu sürece ilişkin işlem adımları Şekil 6.12'de sunulduğu gibidir.

Şekil 6.12'den de görülebildiği gibi DBN'yi oluşturan alt nesneler üzerinde değişiklik yapan uygulama, yeni DBN'yi aynı isim altında kaydettirmek için normalde kullanılan komuttan farklı olarak

```
re_register(dbn_obj,"ad");
```

komutu ile (1) DBNS_i'nin adlandırma yöneticisine çağrı gönderir (2). Çağrıyı alan adlandırma yöneticisi, isteği koordinatörün yerleşke yöneticisine iletir (3). Yerleşke yöneticisi, gelen çağrının gerçekten daha önce bir başka *ser_id* bilgisi ile kaydedilmiş olan *"ad" isimli* DBN'ye ait olup olmadığını kontrol eder, "DBN yerleşke tablosunu" yeni duruma göre düzenler ve sonucu DBNS_i'ye bildirir (4). Bunun üzerine, DBNS_i <*dbn_obj, ad*> çiftini kendi DBN tablosuna kaydeder ve *ad* isimli DBN'yi kullanmakta olan diğer uygulamaların sunucularını değişiklikten haberdar etmek üzere kendi nesne yöneticisine bir çağrı gönderir (5). Çağrıyı alan nesne yöneticisi öncelikle, *ad* isimli DBN'yi kullanmakta olan sunucuların listesini

koordinatörden ister (6-7). Koordinatörden yanıt olarak gelen listedeki her bir sunucuya *ad* isimli DBN'nin yapısının değiştiği mesajını gönderir (8). Çağrıyı alan sunucular hizmet verdikleri uygulama programına değişiklik mesajını gönderirler (9).



Şekil 6.12. Bir DBN'nin iç yapısındaki bir değişikliğin diğer uygulamalara bildirilmesi

Değişiklik mesajını alan bir uygulama programı en basit şekliyle `inform_object_change()` metodu içinde,

```
load_class("DBN_Class");
lookup("ad");
```

komutlarını tekrar yürüterek, kabuk nesne ve alt nesnelere ait yeni sınıf tanımlamalarını yükler ve yeni bileşik nesneyi kendi adres alanına bağlayabilir.

Bununla birlikte, çok daha farklı bir yaklaşımlar da söz konusu olabilir. Bu, uygulamanın şekline ve bileşik nesnenin o alanda hangi amaçla kullanıldığına bağlı olarak değişebilecek bir durumdur.

7. OTOMATİK SINIF ÜRETECİ

7.1 Giriş

Dağıtılmış bileşik nesneyi oluşturan alt nesnelerin, erişim ve yönetimine ilişkin ayrıntılarını kullanıcıdan gizleyen bağlantı ve kontrol nesneleri DBN modelini oluşturan en önemli sistem elemanlarıdır.

Bağlantı nesnesi bir alt nesnenin uzak alanlar üzerinde kolaylıkla bağlanabilmesini sağlarken, kontrol nesnesi de alt nesne üzerinde yürütülen her türlü metot çağrısını denetimi altına alarak, gerekli senkronizasyon ve tutarlılığı sağlama işlemlerini yürütür.

DBN tasarımcısı, merkezi bir uygulamaya benzer şekilde, alt nesne sınıflarını ve onların oluşturduğu bileşik nesne yapısını kuran kabuk nesnenin sınıfını hazırlamakla yükümlüdür. Bölüm 4.4'te "Bir Bileşik Nesnenin Hazırlanması" başlığı altında anlatıldığı gibi, her bir alt nesneye ait bağlantı ve kontrol nesnelerinin üretileceği sınıflar, alt nesne sınıflarının arayüz tanımlamaları kullanılarak bir sınıf üreteci tarafından otomatik olarak üretilirler. Bu bölümde de, yapısı basit bir derleyiciyi andıran *Otomatik Sınıf Üretecinin (OSÜ)* tasarım ve gerçekleme ayrıntıları incelenecektir.

7.2 Sınıf Adları İçin Kabul Edilen Kurallar

Bağlantı ve kontrol nesneleri sınıfları otomatik olarak üretileceği için, hem bu sınıflara ait, hem de alt nesne sınıfları ve arayüz tanımlamalarına ait dosyaların aşağıda belirtilen kurallara uygun olarak adlandırılmaları gerekmektedir.

1. Alt nesnelere ait sınıf dosyaları "`Sub_`" öneki ile başlar. Örneğin `Sub_Account.java`, `Sub_Person.java` gibi.

2. Arayüz tanımlama dosyaları `Sub_` önekini içermeksizin "`_itf`" soneki ile biter. Bu durumda, `Sub_Account.java` sınıfı için yazılacak arayüz dosyası `Account_itf.java` olacaktır.

3. Bağlantı nesnesi sınıfı adı, alt nesne sınıf adından "`Sub_`" öneki çıkarılarak elde edilir. `Sub_Account.java` sınıfı için üretilen bağlantı nesnesi sınıfı `Account.java` olur.

4. Kontrol nesnesi sınıfı adı da, bağlantı nesnesi sınıf adına "`Control_`" öneki eklenerek elde edilir. `Sub_Account.java` sınıfı için üretilen kontrol nesnesi sınıfı Control_`Account.java` olur.

Sınıf adlarının uymaları gereken kurallar yalnızca bileşik nesne tasarımcısını (kural 1 ve 2) ve sınıf üretici programı (kural 3 ve 4) ilgilendirir. Diğer kullanıcılar ise, zaten bileşik nesnenin dağıtılmış ortamdaki parçalı yapısını görmedikleri ve doğrudan bileşik nesneyi oluşturan kabuk nesne üzerinde metot çağrıları yürüttükleri için, kendilerine saydam olarak gerçeklenen ayrıntıların muhatabı değillerdir.

7.3 Sınıfların Üretilmesi İçin Kullanıcı Arayüzü

Bağlantı ve kontrol nesnelerine ait sınıflar `Class_Generator.java` isimli bir Java programı aracılığı ile üretilirler. Alt nesne sınıfına ait arayüz tanımlaması ve kontrol nesnesi sınıfı içinde kullanılacak olan tutarlılık protokolü türü programın giriş parametrelerini oluştururlar.

Alt nesne kopyalarına uygulanacak tutarlılık protokolünün türünü belirlemek üzere,

- Bir güncelleme işleminin sonrasında diğer alt nesne kopyaları için geçersiz kılma protokolü uygulanacaksa, "`Invalidate`",

- kopyaları güncelleme protokolü uygulanacaksa, "`Update`",

sözcüklerinden biri parametre olarak kullanılır.

Alt nesne sınıfına ait arayüz tanımlaması eğer programın çalıştırıldığı dizin içinde yer alıyor ise, aşağıda görüldüğü gibi doğrudan dosya adı parametre olarak verilir.

```
java Class_Generator Account_itf.java Update
```

veya,

```
java Class_Generator Account_itf.java Invalidate
```

Aksi durumda, kök dizinden itibaren arayüz tanımlamasının bulunduğu dizine kadar olan yol ve dosya adı aşağıdaki gibi belirtilir:

```
java Class_Generator
      c:/my/Java/Classes/Account_itf.java Update
```

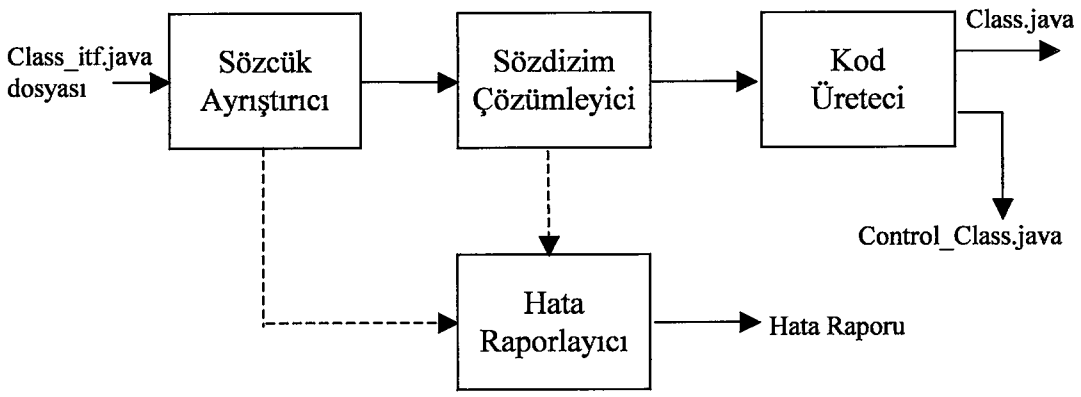
veya

```
java Class_Generator
      c:/my/Java/Classes/Account_itf.java Invalidate
```

Program çalışmasını başarı ile tamamlaması halinde, bağlantı nesnesi sınıfı için `Account.java` ve kontrol nesnesi sınıfı için `Control_Account.java` sınıf dosyaları `Account_itf.java` dosyasının bulunduğu dizin içinde yaratılırlar.

7.4 Otomatik Sınıf Üretecinin Genel Yapısı

Sınıf üretecinin içerisinde Şekil 7.1'de gösterildiği gibi, basit bir *Sözcük Ayrıştırıcı, Sözdizim Çözümleyici, Kod Üreteci* ve *Hata Raporlayıcı* bulunmaktadır.



Şekil 7.1. Otomatik Sınıf Üretecini oluşturan ana elemanlar

Sözcük ayrıştırıcı, giriş parametresi olarak gelen ve örnek olarak kabul ettiği herhangi bir alt nesne sınıfına ait arayüz tanımlaması (örneğin, `Account_itf.java` Bkz. Şekil 7.2) içinde yer alan tüm anahtar sözcükleri (`public`, `interface`, `void`, `int`, vs.), arayüz, metot ve parametre isimlerini (`Account`, `deposit`, `withdraw`, `amount`, vs.) ve özel karakterleri (`;` , `(` `)` `{` `}` vs.) birbirinden ayırarak, bunların Java programlama dili içinde bulunmasına izin verilen *sözcük (token)*'ler olup olmadıklarını kontrol eder. *Sözcük* ile, dilde bir anlam ifade eden karakter veya karakter katarından oluşan bir ifade kastedilmektedir.

Sözdizim çözümleyici, sözcük çözümleyici tarafından ayıraçlarla ayrılmış olan sözcük ve karakterlerin birbirlerine göre dizilişlerinin OSÜ'nün belirlediği (Şekil 7.2'de verilen örnekte olduğu gibi) ve Java RMI yapısında kullanılan arayüz tanımlamasına göre küçük bir farklılık arz eden (metot isimlerinin başına `R_` ve `W_` öneklerinin getirilmesi), yapıya uygun olup olmadığını kontrol eder.

```
public interface Account {
    public void W_deposit (float amount);
    public void W_withdraw (float amount);
    public float R_balance ();
}
```

Şekil 7.2. Örnek bir arayüz tanımlaması: `Account_itf.java`

Kod üreteci, bağlantı nesnesi ve kontrol nesnesi sınıflarında bulunması gereken standart kod parçaları ile arayüz içinde yer alan metot tanımlamalarına göre, üretilecek olan kod parçalarını içeren iki adet sınıf dosyasını (`Account.java`, `Control_Account.java`) üretir.

Yukarıda ardışıl olarak yürütülüyormuş gibi açıklanan sözcük çözümleme, sözdizim çözümleme ve kod üretme aşamaları, ilerleyen bölümlerde de açıklanacağı gibi, tümüyle birlikte yürütülen aşamalardır.

Hata raporlayıcı ünite, sözcük ayrıştırma ve sözdizim çözümleme aşamalarında oluşacak olan hata durumlarını uygun hata mesajları ile çıkışa aktarır. Örneğin,

anahtar sözcüklerin yanlış yazılması (`public` yerine `publc` gibi), arayüz tanımlaması içinde bulunmaması gereken bir özel karakterin kullanılması (:, ?, $ gibi) ya da kullanılan metot ve parametre isimlerinin Java programlama dilinde tanımlayıcılar için belirlenen kurallara uymaması (`hesap-bakiyesi`, `para-çekme` gibi) halinde, sözcük ayrıştırıcının belirlediği bir hata oluşur. Benzer şekilde, sözcüklerin sıralamalarının arayüz tanımlama kurallarına uymaması halinde (`interface public Account` gibi) sözdizim çözümleyicinin belirlediği bir hata oluşur.

7.4.1 Sözcük Ayrıştırıcı

Sözcük çözümleyici ünitesinin çalışması iki aşamalıdır. İlk aşamada sınıf üreticisini çalıştırmak için terminalden girilen Java komutu çözümlenir. İkinci aşamaya ise, komut satırında yer alan parametre listesinde hataya rastlanmaması durumunda geçilir ve ilk parametrenin belirlediği arayüz tanımlama dosyası açılır ve içerdiği kod parçasında yer alan sözcüklerin ayrıştırılması işlemi başlar.

Sözcük ayrıştırıcının sınıf üretecini çalıştıran komutu çözümlemek üzere yürüttüğü algoritma Şekil 7.3'te verilmiştir.

Kullanıcı, sınıfları üretmek amacıyla terminalden,

```
java Class_Generator dizin/dosya_adı tutarlılık_türü
```

şeklinde bir komut girdiğinde öncelikle, Şekil 7.3'te verilen algoritmadan görülebileceği gibi `dizin` ve `dosya_adı` birbirinden ayrılır. İkinci olarak, söz konusu dizinde dosyanın var olup olmadığı ve hemen ardından, tutarlılık türünü belirleyen anahtar sözcüğün doğru olarak girilip girilmediği (`Update/Invalidate`) kontrol edilir. Herhangi bir hatalı durumla karşılaşılmadığı takdirde, ikinci aşamayı başlatmak üzere `dosya_adı` ile belirtilen dosya açılır.

```
String param1 ← ilk parametre
int i ← ilk parametre içinde yer alan en son "/" karakterinin indisi
if (i = -1) {
    dizin ← "."
    dosya ← ilk parametre
} else {
    String dir ← ilk parametrenin i. indise kadar olan bölümü
    String dosya ← ilk parametrenin i. indisten sonraki bölümü
} end-if
if (dosya mevcut değil)
    hatayı rapor et (dosya mevcut değil)
end-if
param2 ← ikinci parametre
if (param2 = null)
    hatayı rapor et (tutarlılık protokolü türü parametresi mevcut değil)
else if (param2 != "Update") and (param2 != "Invalidate")
        hatayı rapor et (tutarlılık protokolü türü parametresi yanlış)
    end-if
end-if
dosya ile belirtilen arayüz tanımlama dosyasını aç
```

Şekil 7.3. Terminalden girilen komutu analiz eden kod parçası

İkinci aşama, arayüz tanımlama dosyasının içindeki sözcüklerin ayrıştırılmalarını içerir. Bu amaçla yapılan ilk işlem, anahtar sözcükler, tanımlayıcı adları (sınıf adı, metot adı ve parametre adları) ve özel karakterlerin arasına ayıraçlar konarak, her bir sözcüğün ayrı ayrı belirlenmesi işlemidir. Ayıraç olarak bir arayüz dosyası içinde var olan " `\t\n\r;{},()` " karakterleri kullanılır. Ayırma işlemi için Java'nın standart sınıf kütüphanesi içinde bulunan `StringTokenizer` sınıf yapısı kullanılır. Bu sınıf yapısı verilen bir karakter katarı içindeki sözcükleri, yine verilen ayıraç karakterlerini kullanarak birbirinden ayırır. Örneğin, aşağıda verilen Java kodu

"str" karakter katarının içinde bulunan sözcükleri parametre olarak verilen ayıraç-listesi'ne göre ayırarak "st" sözcükler listesine atar.

```
StringTokenizer st =
        new StringTokenizer(str," \t\n\r;{},()", true);
```

StringTokenizer(..) içindeki "true" parametresi ayıraç-listesi'nde yer alan karakterlerin de birer sözcük olarak belirlenmelerini sağlar. Eğer, bu parametre "false" değerini taşıması halinde, bu durumda ayıraç-listesi'nde yer alan karakterler birer sözcük olarak değerlendirilmezler.

Sözcük ayrıştırıcı, sözdizim çözümleyici ile birlikte çalışır. Sözdizim çözümleyiciden gelen bir istek üzerine, sıradaki sözcüğün doğruluğunu kontrol eder. Bu çalışma düzenini gerçekleyen algoritma Şekil 7.4'te verilmiştir. Eğer sözcüklerin bulunduğu st sözcükler listesinin içi boş ise, dosyadan yeni bir satır okunur, bu satırda yer alan sözcükler birbirlerinden ayrılır ve st'ye atanır. Daha sonra, st sözcükler listesinden bir sözcük çekilir. Eğer çekilen bu sözcük " ", "\t", "\n" veya "\r" değil ise, çağrı dönüş parametresi olarak geri döndürülür. Aksi takdirde, ayıraç karakterinden farklı bir sözcük elde edene kadar, yeni bir sözcük çekilme işlemi sürdürülür. st listesi boşaldığında dosyadan yeni bir satır okunarak aynı işlemler tekrarlanır.

Sonuç olarak, sözcük çözümleme aşamasında yürütülen işlemler kısaca özetlenecek olursa;

- Terminalden girilen sınıf üretecini çalıştıran komutta yer alan dizin ve dosya adlarının ayrıştırılması,
- Dosya adı ile belirtilen dosyanın fiziksel olarak var olup olmadığının kontrolü,
- Dosyanın açılıp, içindeki sözcüklerin teker teker belirlenmesi,

işlemleridir.

```
Class Token {
   Public String get() {
      StringTokenizer st;
      String tok, str;
      while (true) {
         if (st boş) {
            str ← f dosyasından bir satır oku
            if (str = "")
               return "";
            end-if
            st = new StringTokenizer(line,
                    " \t\n\r;{},()!",true);
            continue;
         } end-if
         tok = st.nextToken();
         if (tok = " ") or
            (tok = "\t") or
            (tok = "\n") or
            (tok = "\r")
           continue;
         end-if
         return tok;
      } end-while
   } end-method
} end-class
```

Şekil 7.4. Sözcükleri birbirinden ayırarak sözdizim çözümleyiciye gönderen algoritma

7.4.2 Sözdizim Çözümleyici

Sözdizim çözümleyici, bağlantı ve kontrol nesne sınıflarının üretilmesi amacıyla kullanılan arayüz tanımlama dosyası içinde bulunan sözcüklerin diziliş sıralarının kurallara uygun olup olmadığını denetleyen birimdir.

Sözdizim çözümleyici sözcüklerin doğru sıralanışına ilişkin bilgileri bir gramer yapısından öğrenir. Bir dilin yapısını oluşturan kurallar zincirine gramer adı verilir ve bir gramer çok sayıda türetim kuralının bir araya gelmesi ile oluşur. Türetimler, bir dil içinde yer alan her türlü komut ve tanımlamalara ilişkin yazım kurallarını belirlerler. Türetim, bir ifadenin uygun bir açılım ile yer değiştirerek, daha farklı ifadeler şeklinde yazılmasıdır [38].

Her gramerin bir başlangıç simgesi vardır ve aksi belirtilmedikçe gramer içindeki ilk türetimin sol tarafında yer alan simge, o gramerin başlangıç simgesidir. Eğer bu simgeden başlanmak suretiyle, bir dizi türetimler sonunda kaynak program koduna ulaşılabiliyorsa, sözdizim çözümleme işlemi başarı ile tamamlanmış demektir. Aksi durumda, bir hata oluşmuştur ve uygun bir hata mesajı ile çözümleme işlemi bitirilir.

7.4.2.1 Arayüz Yazım Grameri

Sözdizim çözümleyicinin çalışma düzeni arayüz tanımlama dosyası içinde yer alan sözcüklerin dizilişlerinin doğruluğunun kontrol edilebilmesi için, öncelikle arayüz yazımına ilişkin kurallar zincirini içeren bir gramer yapısının oluşturulmasını gerektirmektedir. Bu amaçla, Şekil 7.5'te verilen türetimlerden oluşan bir gramer yapısı tanımlanmıştır.

1. list_of_itf → ∈ | itf list_itf
2. itf → **"public"** **"interface"** **id** "{" list_methods "}"
3. list_methods → ∈ | method list_methods
4. method → **"public"** return_type **id** "(" list_param1 ") ;"
5. list_param1 → ∈ | list_param2
6. list_param2 → param | param "," list_param2
7. param → param_type id
8. param_type → **"int"** | **"String"** | **"boolean"** | **id**
9. return_type → ∈ | **"void"** | **"int"** | **"String"** | **"boolean"** | **id**

Şekil 7.5. Arayüz yazımının sözdizim çözümlemesi için tanımlanan gramer yapısı

Şekil 7.5'te koyu renkli olarak belirtilenler ifadeler arayüz yapısı içerisinde bulunmasına izin verilen anahtar sözcükler olup, daha ayrıntılı bir yazım şekli bulunmayan ifadelerdir. ∈ (boş katar) türetim işaretinin (→) solunda yer alan ifadenin bir başka ifadeye açılmasının zorunlu olmadığını göstermektedir. **id** ise, arayüz, metot, parametre ve oluşturulan sınıf adlarını ifade etmektedir.

7.4.2.2 Sözdizim Çözümlemesi İşlemi

Söz dizim çözümleyicinin gerçeklenmesinde *Yinelemeli iniş (Recursive descent)* yöntemi benimsenmiştir. Yinelemeli iniş yukarıdan-aşağıya *(top-down)* bir sözdizim çözümleme yöntemidir ve çözümleme sürecinde bir dizi yinelemeli yordam çağrıları ile işlem yapar. Yinelemeli iniş yöntemi, tüm diğer yukarıdan-aşağıya sözdizim çözümleyicilerde de olduğu gibi, bir türetim içinde en solda yer alan *sözcük-olmayan (nonterminal)* ifadenin uygun bir türetim ile yer değiştirmesi ilkesine dayanır. Bu amaçla, gramerde yer alan her bir sözcük-olmayan ifade için bir yordam yazılması gerekir [38].

Sözdizim çözümlemesi için "`Parsing`" adlı bir sınıf yapısı oluşturulmuştur. `Parsing` sınıfı aşağıda açıklanacak olan metotları ile, arayüz tanımlama dosyasında yer alan sözcüklerin, Şekil 7.5'te verilen gramer yapısına uygun olarak yerleşip yerleşmediklerini denetler. Bu amaçla, `Parsing` sınıfında gramerde yer alan her bir türetim kuralına karşılık, o türetimi gerçekleyen bir metot yer almaktadır.

`Parsing` sınıfı içinde, algoritması Şekil 7.4'te verilen `Token` sınıfından bir nesne (`token`) yaratılır. Bu şekilde, daha sonraki aşamalarda `token` nesnesinin `get()` metoduna yapılacak çağrılara (`token.get()`) karşılık, daha önce belirlenmiş olan kurallara göre (" \t\n\r;{},()!") birbirlerinden ayrılmış olan sözcüklerin sıra ile alınması sağlanacaktır.

Sözdizim çözümleme işlemi `Parsing` sınıfı içinde bulunan ve kodu Şekil 7.6'da verilen `analyze()` metoduna yapılan çağrı ile başlar. `analyze()` `kendi` içinden gramerin başlangıç simgesini tanımlayan türetimi gerçekleyen `analyze_list_itf()` metoduna (Bkz. Şekil 5.7) bir çağrı üreterek, çözümleme işlemini başlatır.

```
public void analyze() {
    if (!analyze_list_itf())
       System.out.println("Error in interface analysis");
    else {
       System.out.println("Analysis Completed Successfully");
    }
}
```

Şekil 7.6. `analyze()` metodu

```
private boolean analyse_list_itf() {
    tok = token.get();
    if (tok.equals("")) return true;
    if (!analyse_itf()) return false;
    return ana_li_itf();
}
```

Şekil 7.7. `analyze_list_itf()` metodu

Şekil 7.7'de verilen `analyze_list_itf` metodu ile gramerin başlangıç simgesini tanımlayan

`list_of_itf → ∈ | itf list_of_itf`

türetim kuralı denetlenir. Öncelikle, arayüz dosyasının boş olma durumu,

`list_of_ itf → ∈`

ile denetlenir. Eğer `token.get()` metodunun dönüş parametresi "" (boş katar) ise, sözcük ayrıştırıcı hiçbir sözcük bulamamıştır ve bu durum arayüz dosyasının boş olduğu manasına gelmektedir. Aksi takdirde, ikinci seçenek olan,

`list_of_itf → itf list_of_itf`

ile çözümleme işlemine devam edilir. Bu amaçla, türetimin açılımında ilk simge olan `itf`'yi gerçekleyen ve kodu Şekil 7.8'de verilen `analyze_itf()` metodu çağrılır.

```
private boolean analyze_itf() {
   if (!tok.equals("public")) {
      System.out.println("\n line "+token.line_no()+
                  ": keyword public expected.");
      return false;
   }
   tok = token.get();
   if (!tok.equals("interface")) {
      System.out.println("\n line "+token.line_no()+
                  ": keyword interface expected.");
      return false;
   }
   tok = token.get();
   if (!is_identifier(tok)) {      /* for the interface identifier */
      System.out.println("\n line "+token.line_no()+
                  ": interface identifier expected.");
      return false;
   }
   class_name = tok;
   tok = token.get();
   if (!tok.equals("{")) {
      System.out.println("\n line "+token.line_no()+
                  ": left bracket expected.");
      return false;
   }
   tok = token.get();
   if (!analyse_list_methods()) return false;
   if (!tok.equals("}")) {
      System.out.println("\n line "+token.line_no()+
                  ": right bracket expected.");
      return false;
      }
   return true;
}
```

Şekil 7.8. `analyze itf()` metodu

`analyze_itf()` metodu giriş sözcüklerinin gramerin ikinci türetim kuralı olan,

```
itf → "public" "interface" id "{"
          list_of_methods "}"
```

türetimine uygunluğunu denetler. `itf` tanımına göre, eğer ilk iki sözcük sırasıyla "**public**" ve "**interface**" ise, daha sonra gelen sözcüğün bir tanımlayıcı olup, Java'da kullanıcı tanımlayıcılarının yazımında uyulması gereken kurallara uygun olarak yazılmış olması gerekir. Bu amaçla, kodu Şekil 7.9'da verilmiş olan `is_identifier()` metoduna bir çağrı gönderilerek, tanımlayıcı sınaması yapılır. Çağrı dönüş değeri `"true"` ise, sözcük bir tanımlayıcıdır ve sınıf adı olarak atanır. Aksi halde, söz dizim çözümleme işlemi uygun bir hata mesajıyla sonlandırılır.

```
private boolean is_identifier(String s) {
   char c;
   int i;
   if (!Character.isJavaIdentifierStart(s.charAt(0)))
      return false;
   i = 1;
   while (true) {
      try {
         c = s.charAt(i);
         if (Character.isJavaIdentifierPart(s.charAt(i))) {
            i++;
            continue;
         }
      } catch (StringIndexOutOfBoundsException  ex) {
            return true;
      }
   }
}
```

Şekil 7.9. Kullanıcı tanımlayıcılarının doğruluğunu test eden `is_identifier()` metodu

Sınıf adının belirlenmesinin ardından, izleyen sözcük "{" ise, bu durumda ikinci türetim kuralında sırada yer alan `list_of_methods` ve } ifadeleri, arayüz dosyasında bir metot tanımlamaları listesi ve bunları izleyen "}" sözcüğü ile sonlanma olup olmadığının test edilmesini gerektirir. Bu amaçla, `analyze_itf()` içinden kodu Şekil 7.10'da verilen `analyze_list_methods()` metoduna çağrı yapılarak, metot tanımlamalarının çözümlenme işlemine geçilir.

```
private boolean analyse_list_methods() {
   if (tok.equals("}")) return true;
   if (!analyse_method()) return false;
   return ana_list_methods();
}
```

Şekil 7.10. `analyse_list_methods()` metodu

`analyze_list_methods()` metodu içinde gramerde yer alan üçüncü türetim kuralı olan,

`list_methods` → ∈ | `method list_methods`

kuralına uygunluk denetlenir. Buna göre öncelikle,

`list_methods` → ∈

türetimi uyarınca eğer gelen sözcük "}" ise, bu "{ }" arasının boş olduğunu, yani arayüz dosyasının içinde hiçbir metot tanımlamasının bulunmadığını gösterir. Aksi durumda, kuralın ikinci türetimi olan,

`list_methods` → `method list_methods`

türetimi uyarınca, bir metot ve onu izleyen metotlar kümesinin varlığı denetlenir. Bu amaçla, gramerde yer alan dördüncü türetim kuralı olan,

`method` → **"public"** `return_type` **id**
"(" `list_param1` **")"** **";"**

türetimine uygunluğu denetleyen, kodu Şekil 7.11'de verilen `analyze_method()` metoduna bir çağrı yapılır.

```
private boolean analyze_method() {
   if (!tok.equals("public")) {
      System.out.println("\n line "+token.line_no()+
            ": public method expected.");
      return false; }
   tok = token.get();
   if (!analyze_return_type()) {
      System.out.println("\n line "+token.line_no()+
            ": return type expected.");
      return false; }
   String s=tok.substring(0,2);
   tok=tok.substring(2);
   if (s.equals("W_")) kind=1;
   else  if (S.equals("R_")) kind=0;
         else { System.out.println("\n line +
            token.line_no()+ ": W_ or R_  expected.");
            return false; }
   if (!is_identifier(tok)) {  /* for the method name */
      System.out.println("\n line "+token.line_no()+
            ": method identifier expected.");
      return false; }
   if (!tok.equals("(")) {
      System.out.println("\n line "+token.line_no()+
            ": left parenthesis expected.");
      return false; }
   tok = token.get();
   if (!analyze_list_param1()) return false;
   if (!tok.equals(")")) {
      System.out.println("\n line "+token.line_no()+
            ": right parenthesis expected.");
      return false; }
   tok = token.get();
   if (!tok.equals(";")) {
      System.out.println("\n line "+token.line_no()+
            ": ';' expected.");
      return false; }
}
```

Şekil 7.11. `analyse_method()` metodu

`analyze_method()` metodu gelen sözcüklerde gramerin dördüncü türetim kuralı içindeki yapının varlığını arar. Öncelikle, "**public**" anahtar sözcüğü ve ardından gelen sözcüğün bir geri dönüş parametresi olması gerekir. Bu amaçla, kodu Şekil 7.12'de verilen `analyze_return_type()` metodu çağrılarak sıradaki sözcüğün dokuzuncu türetim kuralı olan

```
return_type → ∈ | "void" | "int" |
              "String" | "boolean" | id
```

türetimine uygunluğu denetlenir. Bu çağrıdan başarı ile dönülmüş ise, bir sonraki sözcük bir metot adı olmalıdır ve metot adının tanımlayıcı kurallarına uygunluğunu denetlemek için, `is_identifier()` metodu çağrılır. Bu çağrıdan da olumlu yanıt alınmış ise, daha sonra gelen sözcük "(" ve ardından bir parametre listesi yer almalıdır.

```
private boolean analyze_return_type() {
   param_value=tok;
   if (tok.equals("void")) void_meth = true;
      else void_meth = false;
   if (vct_fragment.contains(tok))
      fragment_return = true;
      else fragment_return=false;
   if (tok.equals("void") || tok.equals("int") ||
         tok.equals("boolean") || tok.equals("String")) {
      tok = token.get();
      basic_return = true;
      return true;  }
   if (!is_identifier(tok)) {          /* for a type identifier */
      System.out.println("\n line "+token.line_no()+
            ": return type identifier expected.");
      return false;
   }
   return true;
}
```

Şekil 7.12. `analyze return type()` metodu

Gramerde yer alan beş, altı ve yedinci türetim kuralları parametre listesinin oluşturulmasını içeren kurallardır. Gelen sözcüklerin bu kurallara uygunluğunu denetlemek için `analyze_method()` metodu içinden kodu Şekil 7.13'te verilen `analyze_list_param1()` metodu çağrılır. Çağrıdan başarı ile dönülmesi durumunda, metot analizi için son olarak ")" ve ";" sözcüklerinin gelip gelmediği incelenir.

```
private boolean ana_list_param1() {
   if (tok.equals(")")) return true;
   return analyse_list_param2();
}
```

Şekil 7.13. `analyse_list_param1()` metodu

Şekil 7.13'te verilen `analyze_list_param1()` metodu, sıradaki sözcüğünün gramerde yer alan beşinci türetim kuralı olan,

```
list_param1  →  ∈  |  list_param2
```

türetimine uygunluğunu denetler. Eğer sırasındaki sözcük ")" ise, metot çağrı parametresi içermiyor demektir. Aksi durumda, türetimin ikinci kısmı olan,

```
list_param1  →  list_param2
```

türetimine uygunluğu kontrol etmek için kodu Şekil 7.14'te verilen `analyze_list_param2()` metoduna çağrı yapılır.

```
private boolean analyze_list_param2() {
   if (!ana_param()) return false;
   if (!tok.equals(",")) return true;
   return analyse_list_param2();
}
```

Şekil 7.14. `analyze_list_param2()` metodu

`analyze_list_param2()` metodu, gramerde yer alan altıncı türetim kuralı olan

```
list_param2 → param | param "," list_param2
```

türetimine uygunluğu denetler. Bu amaçla, öncelikle ilk parametre tanımlamasının doğruluğunun kontrolü için kodu Şekil 7.15'te verilen `analyze_param()` metoduna bir çağrı yapar. Çağrıdan başarı ile dönülmüşse ve sıradaki sözcük de "," ise, bu çağrı parametrelerinin birden fazla olduğunu gösterir. Bu durumda metot tekrar kendini çağırır ve aynı işlemler diğer parametre tanımlamaları için de devam ettirilir. Aksi durumda, çağrı içinde yalnızca bir parametre var demektir.

Şekil 7.15'te verilen `analyze_param()` metodu, sıradaki sözcüklerin gramerde yer alan yedinci türetim kuralı olan,

```
param → param_type id
```

türetimine uygunluğunu denetler.

```
private boolean analyze_param() {
   if (!analyze_param_type()) return false;
   if (!is_identifier(tok)){    /* for the parameter identifier */
      System.out.println("\n line "+token.line_no()+
         ": parameter identifier expected.");
      return false;
   }
   tok = token.get();
   return true;
}
```

Şekil 7.15. `analyze_param()` metodu

Bir parametre tanımı, başta parametre tipi ve hemen onu izleyen parametre adı ile yapıldığı için, `analyze_param()` metodu öncelikle sıradaki sözcüğün bir tip olup olmadığını denetlemek amacıyla, kodu Şekil 7.16'da verilen `analyze_param_type()` metoduna bir çağrı gönderir. Çağrıdan başarı ile

dönülmüşse, sıradaki sözcüğün bir tanımlayıcı olup olmadığını kontrol etmek için `is_identifier()` metodunu çağırır. Bu çağrıdan da başarı ile dönülmüşse, parametre tanımlaması doğru olarak yapılmış demektir.

```
private boolean analyze_param_type() {
   if (tok.equals("void") || tok.equals("int")
       || tok.equals("boolean")
       || tok.equals("String")) {
   basic_param = true;
   tok = token.get();
   return true;
   }
   if (!is_identifier(tok)){       /* for a type identifier */
      System.out.println("\n line "+token.line_no()+
        ": parameter type identifier expected.");
      return false;
   }
   basic_param = false;
   tok = token.get();
   return true;
}
```

Şekil 7.16. `analyze_param_type()` metodu

Şekil 7.16'da verilen `analyze_param_type()` metodu, sıradaki sözcüğün standart bir parametre tipi ya da kullanıcı tanımlı bir sınıfa ait olup olmadığını, gramerde yer alan sekizinci türetim kuralı olan,

`param_type` → **"int"** | **"String"** | **"boolean"** | **id**

türetimine göre denetler.

`Parsing` sınıfından üretilen bir nesnenin `analyze()` metoduna yapılan çağrı ile başlayan sözdizim çözümleme işlemi, arayüz dosyasında yer alan sözcüklerin Şekil 7.5'te verilen gramer yapısına uygunluğu kontrol etmek için, yukarıda yapıları açıklanan, bir dizi metodun iç içe çağrılması ile devam eder. `analyze()` metoduna

sonuçta parametre olarak "`true`" değerinin dönmesi, sözdizim çözümleme işleminin başarı ile sonuçlandığını, bir başka deyişle, arayüz tanımlama dosyasında bir yazım hatasının olmadığını gösterir.

Daha önce de belirtildiği gibi, sözdizim çözümleme ve kod üretme işlemleri aynı anda yürütülürler. Bu nedenle, sözdizim çözümleme işlemi sonlandığı anda, bağlantı ve kontrol nesnelerinin türetileceği sınıf dosyaları da üretilmiş olur.

7.4.3 Kod Üreteci

Kod üretici ünitenin görevi, sözdizim çözümleme işlemi sırasında arayüz dosyasından elde edilen `sınıf_adı` (örneğin `Account`) için, arayüz dosyasının bulunduğu dizin içinde bağlantı ve kontrol nesnesi sınıflarının türetileceği `Account.java` ve `Control_Account.java` dosyalarını yaratmaktır.

Bu amaçla, `Class_Generator` sınıfı içinde, yapısı Şekil 7.17'de verilen `OutputText` adlı bir sınıf yapısı oluşturulmuştur. `OutputText` sınıfı rasgele erişimli dosyalar üzerinde dosya yaratma, dosyaya veri girme ve kapatma gibi temel işlemleri sağlayan metotlar içermektedir.

Sözdizim çözümleme işlemi sırasında (`Parsing` içinde `analyze_itf()` metodu yürütülürken) "**public**" ve "**interface**" anahtar sözcüklerinden sonra gelen tanımlayıcının `sınıf_adı` olarak belirlenmesinin ardından, `sınıf_adı.java` ve `Control_Sınıf_adı.java` dosyaları aşağıdaki şekilde yaratılırlar.

```
OutputText out1, out2;   /* Bu tanımlama Parsing altında yapılır. */
out1 = new OutputText(dir + "/" + class_name + ".java";
out2 = new OutputText(dir+"/Control_"+class_name+ ".java";
```

Sözdizim çözümlemenin ilerleyen aşamalarında, `out1` ve `out2` nesnelerinin `print()` ve `println()` metotlarına yapılacak olan çağrılarla, hem bağlantı, hem de kontrol nesnesi sınıflarının içermesi gereken standart (metot tanımlamalarından bağımsız) ve metot tanımlamalarına göre değişen java kodları dosyalara yazılmaktadır. Örneğin, metot tanımlamalarından bağımsız olan java kodlarının dosyaya yazılması ile ilgili olarak, Şekil 7.18'de bağlantı nesnesi sınıfının,

Şekil 7.19'da da kontrol nesnesi sınıfının kurucu metotlarının dosyaya nasıl yazıldıkları gösterilmiştir.

```
class OutputText {
   public String fname;
   RandomAccessFile file;
   public OutputText(String f) {
      fname = f;
      try { file = new RandomAccessFile(f,"rw");
      } catch (IOException ex) {
            System.out.println("OutputText: "+ex);
      }
   }
   public void print(String s) {
      try { file.writeBytes(s);
      } catch (IOException ex) {
            System.out.println("OutputText: "+ex);
      }
   }
   public void println(String s) {
      try { file.writeBytes(s);
            file.writeBytes("\n");
      } catch (IOException ex) {
            System.out.println("OutputText: "+ex);
      }
   }
   public void close() {
      try { file.close();
      } catch (IOException ex) {
            System.out.println("OutputText: "+ex);
      }
   }
}
```

Şekil 7.17. `OutputText` sınıfının yapısı

```
out1.println("public " + class_name+"()  { ");
out1.println("  try { ");
out1.println("     control_object = new
                        Control_" + class_name + "();");
out1.println("     object_id=control_object.get_id();");
out.println("   } ");
out.println("   catch (Exception ex) { }");
out.println(" } ");
```

Şekil 7.18. Bağlantı nesnesi sınıfının kurucu metodunun dosyaya yazılışı

```
out2.println("  public Control_" + lass_name+"() { ");
out2.println("     sub_obj=new Sub"+class_name+"();");
out2.println("     try {");
out2.println("        my_client_number =
                        dcobe_client.get_client_id();");
out2.println("     add_new_client(my_client_number);");
out2.println("        object_id =
                        dcobe_client.register_new_object
                        (this,sub_obj);");
out2.println("     } catch (Exception ex) {}");
out2.println("  } ");
```

Şekil 7.19. Kontrol nesnesi sınıfının kurucu metodunun dosyaya yazılışı

Bağlantı ve kontrol nesneleri sınıflarında bulunması gereken standart kod parçaları Şekil 7.18 ve Şekil 7.19'da gösterildiği gibi dosyalara yazıldıktan sonra, metot çağrılarına ilişkin kod parçalarının yazımına, ilgili metot tanımlamalarının kurallara uygun olarak yapılıp yapılmadığının denetlendiği, sözdizim çözümlemedeki `analyze_list_methods()` ve izleyen metot çağrıları içinde devam edilir. Bu amaçla, bir metot tanımlamasının gramerde yer alan dördüncü kural olan,

method → "**public**" return_type **id** "(" list_param1 ");"

türetimine uygunluğu saptanırken, aynı zamanda, geri dönüş parametresi tipi, metot adı ve parametre listesi de ayrı ayrı belirlenir. Daha sonra, bağlantı nesnesinin kontrol nesnesi üzerinde yürüteceği her bir çağrının metot yapısını gerçekleyecek olan kod Şekil 7.20'de ve kontrol nesnesinin de alt nesne üzerinde yürüteceği aynı çağrıya ilişkin metot yapısını gerçekleyecek kod Şekil 7.21'de verildiği gibi sınıf dosyalarına yazılır.

```
out1.print("  public ");
out1.print(return_value + " ");
out1.print(method_name + "(");
out1.println(parameter_body + ") {");
out1.println("   if (control_object == null) {");
out1.println("      try {");
out1.println("         control_object = (Control_" +
class_name + ")dcobe_client.get_control_object(object_id);");
out1.println("      } catch (Exception ex) {}");
out1.println("   }");
if (!void_meth) out1.print("      return ");
   else out1.print("     ");
out1.print("     control_object." + meth_name + "(");
out1.println(invoke_body + ") ;");
out1.println("}");
```

Şekil 7.20. Bağlantı nesnesi sınıfında yer alan bir çağrı metodunun dosyaya yazılışı

```
out2.print("  public ");
out2.print(return_type + " ");
out2.print(method_name + "(");
out2.println(parameter_body + ") {");
if (kind==1) { //Writing mode
   out2.println("     ask_object(W); ");
   else out2.println("   ask_object(R); ");
if (!void_meth) out2.print("     return ");
   else out1.print("    ");
out2.println("   sub_obj." + meth_name + "(" +
                     invoke_body + "); ");
out2.println("     try {     ");
out2.println("        if (master == true)
                   end_of_use(my_client_number,W);  ");
out2.println("        else
dcobe_client.release_object(object_id);");
out2.println("     } ");
out2.println("     catch (Exception ex) {  ");
out2.println("        ex.printStackTrace();     ");
out2.println("     } ");
if (!void_meth) out2.println("   return ob;");
else out2.print("    ");
out2.println("   }    ");
```

Şekil 7.21. Kontrol nesnesi sınıfında yer alan bir çağrı metodunun dosyaya yazılışı

Yukarıda anlatılan şekilde bağlantı ve kontrol nesneleri sınıflarında yer alan standart kodlar ile metot yapılarına bağlı kodların dosyalara yazılmasının ardından,

```
out1.close();
out2.close();
```

çağrıları ile dosyalar kapatılarak kod üretme aşaması tamamlanır.

7.4.4 Hata Raporlayıcı

Otomatik olarak sınıf üretme işleminin sözcük çözümleme ve sözdizim çözümleme aşamalarında hatalı durumlar oluşabilir. Daha önce de belirtildiği gibi, anahtar sözcüklerin yanlış yazımı, java'da geçersiz sayılan tanımlayıcı adları kullanımı ya da java'da bulunmayan özel karakterlerin bulunması gibi durumlarda hatalar oluşabileceği gibi, sözcüklerin birbirlerine göre diziliş sıralarının Şekil 7.5'te verilen gramer kurallarına uymaması durumunda da hatalar oluşabilir. Bölüm 7.4.4'te verilen kod parçalarından da görülebileceği gibi, hatalı durumlar arayüz dosyasında hatanın oluştuğu satır numarası ve hataya ilişkin açıklayıcı bir raporlama ile kullanıcıya bildirilmektedir. Örneğin, aşağıda verilen kod parçası "**interface**" anahtar sözcüğünün yanlış kullanımı sonucu oluşan bir hatayı raporlamaktadır.

```
if (!tok.equals("interface")) {
      System.out.println("\n line "+token.line_no()+
                  ": interface expected.");
```

Yukarıdaki kontrolün sonucunda ekranda

```
line 1: interface expected.
```

şeklinde bir mesaj görüntülenecektir.

Benzer şekilde, aşağıda verilen kod parçası da sözcük olarak "}" gelmesi gereken yerde bir başka sözcük ile karşılaşıldığı için ortaya çıkan hatayı raporlamaktadır.

```
if (!tok.equals("}")) {
      System.out.println("\n line "+token.line_no()+
                  ": right bracket expected.");
```

Yukarıdaki kontrolün sonucunda ekranda

```
line 6: right bracket expected.
```

şeklinde bir mesaj görüntülenecektir.

Hata mesajlarının üretilmesinin ardından sınıf üretici programın çalışması sonlanmaktadır. Kullanıcı hatayı düzelttikten sonra, `Class_Generator.class` dosyasını tekrar çalıştırarak sınıf üretme işlemine yeniden başlar.

8. BİR DAĞITILMIŞ BİLEŞİK NESNE UYGULAMASI : INTERNET ÜZERİNDEN MÜŞTEREK KİTAP YAZMA ORTAMININ OLUŞTURULMASI

Bu bölümde, dağıtılmış bileşik nesne modelini temel alarak geliştirilen, "internet üzerinden müşterek kitap yazma ortamının oluşturulması" uygulaması tanıtılacaktır. Internet üzerinden müşterek kitap yazımı, bir çeşit *bilgisayar-destekli müşterek yazışma ortamı (computer-supported collaborative writing environment-CSCWE)* uygulamasıdır. CSCWE yeni bir konu olmayıp, özellikle son on yıl içerisinde literatürde bu konu üzerine yapılmış bir çok akademik ve endüstriyel çalışmalar bulunmaktadır (RIBIS [39] ve SEPIA [40]).

Bir müşterek yazışma alanı, yazarlar arasında sağlanan işbirliğinin türüne göre iki kategoriye ayrılır; bölümlenen ve aynı anda paylaşılan çalışma. İlk kategoriye göre, üzerinde çalışılacak olan görev birbirinden bağımsız alt görevlere bölünerek yazarlar arasında paylaştırılır. İkinci kategoriye göre ise, çok sayıdaki yazar bulundukları yerlerden aynı göreve, aynı anda katılabilirler [41]. Her iki yaklaşımın da kendilerine göre çeşitli avantaj ve dezavantajları bulunmakla birlikte, CSCW perspektifinden bakıldığında, bu bölümde incelenecek uygulama ilk kategoriye girmektedir. Buna göre, bilgisayar destekli müşterek yazışma ortamı uygulamalarının amacı, fiziksel olarak birbirlerinden uzak alanlarda bulunan kullanıcıların, düz yazı, grafik ya da çoğul ortam dokümanlarını aynı anda paylaşarak kullanabilmelerini sağlamaktır [42].

8.1 Internet Üzerinden Müşterek Kitap Yazma Uygulaması

Geliştirilen internet üzerinden müşterek kitap yazma uygulamasının amacı, coğrafik olarak birbirlerinden uzak birden fazla yazar için bir ortak bir geliştirme ortamı sunmaktır. Bölüm 6'da incelenen DBNTO sistemi böyle bir dağıtılmış ortam için gerekli alt yapıyı sunmaktadır. Yazarların üzerinde müşterek çalışacakları kitap bir nesne olarak düşünüldüğünde, kitap nesnesinin aslında tekil bir büyüklük olmayıp,

çok sayıdaki birbirinden bağımsız bölümlerden ve her bir bölümünde kendi içinde alt bölümlerden oluştuğu parçalı bir yapıya sahip olduğu görülür. Kitap nesnesinin bu parçalı görüntüsü, onun bir "dağıtılmış bileşik nesne" olarak modellenebilmesinin mümkün olabileceğini göstermektedir.

Bu amaçla, bir kitapta yer alan her bir bölüm ve alt bölümler ile önsöz, içindekiler, kaynaklar ve fihristin ayrı birer alt nesne ve üzerinde çalışılan kitapta da bu alt nesnelerin bir araya gelmesinden oluşan bir bileşik nesne olarak gerçeklenmiştir. Böylece, yazarların tek bir nesneye erişiyor görüntüsü altında, aslında yalnızca kendilerini ilgilendiren alt nesneye erişmeleri ile, hem birden fazla yazarın aynı anda kitabın farklı bölümleri üzerinde çalışabilmeleri, hem de iletim sırasında ağ üzerinden aktarılacak veri miktarının daha az olması sağlanmıştır. Paralelliğin artması ve ağ iletim sürelerinin kısalması olarak kazanılan bu avantajlar, doğal olarak çalışma ortamının başarım ve erişilebilirliğin de artmasını sağlamıştır.

8.2 Uygulamadaki Kısıtlamalar

Geliştirilen uygulamada kitabın tamamıyla düz yazı tabanlı olduğu kabul edilmiştir. Buna göre kitap içerisinde, şekil, resim ve grafikler yer almayacaktır. Getirilen bu kısıtlama dağıtılmış bileşik nesne modeli veya destekleme ortamından kaynaklanan bir kısıtlama olmayıp, yalnızca uygulamanın gerçekleme aşamasını hızlandırma amacıyla alınan bir karardır. Daha sonra da açıklanacağı gibi, yazışma için Java'da gerçeklenen basit bir düz yazı editörü kullanılmıştır. Ancak, uygulama örneğin MS Word editörünü kullanacak şekilde geliştirildiği takdirde, bu editörünün sağladığı tüm olanaklar kullanılabilecektir.

8.3 Uygulamanın Gerçekleme Ayrıntıları

Internet üzerinden müşterek kitap yazma uygulamasında, bir kitabın içinde yer alan önsöz, içindekiler, kaynaklar, fihrist ve ana bölümleri oluşturan her bir alt bölüm ayrı birer parça olarak düşünülmüş ve bu parçalar birer alt nesne olarak tasarlanmıştır. Daha sonra, bu alt nesneleri bir kabuk nesne içinde birleştiren "`Kitap`" isimli bir bileşik nesne sınıfı oluşturulmuştur. Daha önceki bölümlerde örnek olarak verilen dağıtılmış bileşik nesne uygulamalarının aksine, bu uygulamada `Kitap` sınıfından

türetilen bir "kitap" bileşik nesnesi başlangıçta kitabın bölümlerine ilişkin hiç bir alt nesnesi olmaksızın yaratılmaktadır. Daha sonra, yazarlar kendilerine sunulan kullanıcı arayüzünü kullanarak kitabın bölümlerinin oluşturulması için gereken işlemleri yürütmektedirler. Yani, çalışma sırasında kitap bileşik nesnesine yeni alt nesneler eklenip, çıkarılmaktadır.

8.3.1 Tanımlamalar

Tanımlamalara geçmeden önce, bu uygulama için adlandırma ile ilgili olarak tüm yazarların uyması gereken kuralların verilmesi faydalı olacaktır. Bu kurallara göre; kitabı oluşturan önsöz, içindekiler, kaynaklar ve fihrist bölümleri için, "Önsöz", "İçindekiler", "Kaynaklar" ve "Fihrist", anahtar sözcükleri, bölüm adları için "Bölüm_1", "Bölüm_3" şeklinde "Bölüm_" anahtar sözcüğünü hemen izleyen bir bölüm numarası yer alır. Alt bölüm adları için ise, "2_2", "3_2_1" şeklinde bölüm numarası ve alt bölümlerin numaraları birbirlerinden "_" karakteriyle ayrılarak verilir. Buna göre, bir bölüm adı ile Bölüm_1 şeklinde bir ana bölüm belirtilebileceği gibi, 1_2_1 şeklinde bir alt bölüm de belirtilebilir.

Uygulama başlatıldığı anda kitap bileşik nesnesi hiç bir alt nesnesi olmaksızın yaratılacağına göre, daha sonra eklenecek olan bölüm/alt bölümlere ait bağlantı nesnelerinin referanslarının kitap nesnesi içinde bir şekilde tutulması gerekmektedir. Ayrıca, yaratılan bölümlere ait bağlantı nesnelerinin diğer alanlardaki uygulamalara da aktarılmaları gerekir. Bu amaçla, "kitap_erişim" adlı bir alt nesne kullanılmıştır. Bu alt nesne içinde "erişim" adlı bir tablo bulunmaktadır. Tabloda kitapta var olan tüm bölüm/alt bölümlere ait <bölüm_adı, bağlantı nesnesi> bilgi çiftleri tutulmaktadır. kitap_erişim alt nesnesi erişim tablosuna yeni bir kayıt ekleme, var olan bir kaydı silme ve tabloda arama yapma temel işlemleri için Şekil 8.1'de verilen metotları sunar.

Yazarlar çeşitli zamanlarda kitabın içindekiler bölümünü görmek isteyebilirler. Bundan dolayı, kitapta var olan tüm bölümlere ilişkin <bölüm_adı, bölüm_başlığı> bilgi çifti "plan" adlı bir tabloda "kitap_planı" alt nesnesi içinde tutulmaktadır. kitap_planı alt nesnesi kitaba yeni bir bölüm eklendiğinde, var olan bir bölüm çıkarıldığında, bölüme ilişkin bölüm numarası ya da

bölüm başlığı değiştiğinde, `plan` tablosu üzerinde ekleme, silme ve güncelleme işlemlerini yapan Şekil 8.2'deki metotları sunar.

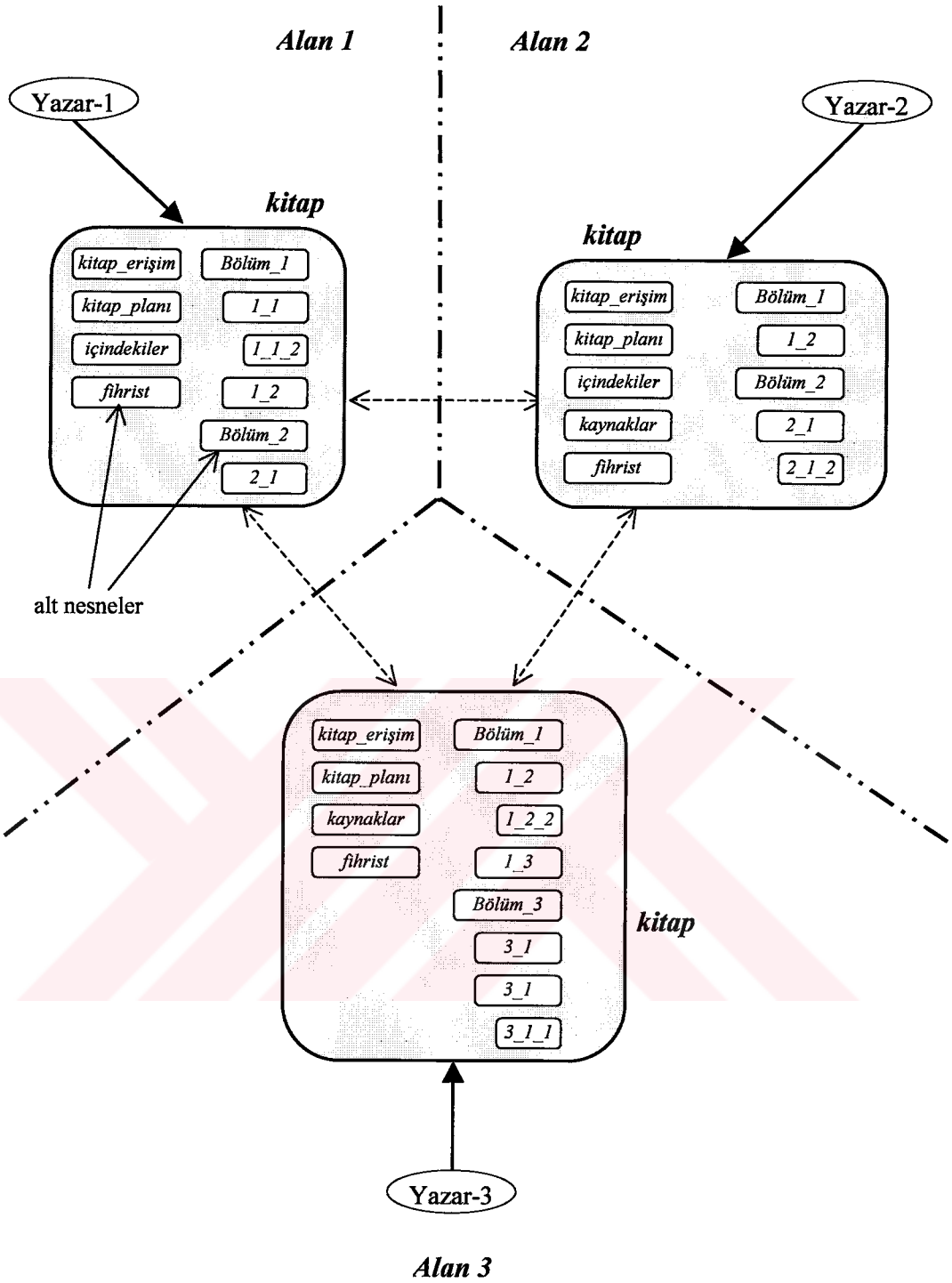
```
boolean    ekle(String bölüm_adı,
                        Bölümler bağlantı_nesnesi)
boolean    sil(String bölüm_adı)
boolean    güncelle(String eski_bölüm_adı,
                        String yeni_bölüm_adı)
Bölümler   ara(String bölüm_adı)
```

Şekil 8.1. `kitap_erişim` alt nesnesi tarafından sunulan metotlar

```
boolean    ekle(String bölüm_adı, String bölüm_başlığı)
boolean    sil(String bölüm_adı)
boolean    güncelle_ad(String eski_bölüm_adı,
                        String yeni_bölüm_adı)
boolean    güncelle_başlık(String bölüm_adı,
                        String yeni_bölüm_başlığı)
Vector     getir()
```

Şekil 8.2. `kitap_planı` alt nesnesi tarafından sunulan metotlar

Yukarıda açıklanan `kitap_erişim` ve `kitap_planı` alt nesneleri ile birlikte, `kitap` bileşik nesnesinin yapısı Şekil 8.3'te verildiği gibi olmaktadır. `kitap_erişim` ve `kitap_planı` alt nesneleri `kitap` bileşik nesnesinin yönetimine ilişkin alt nesneler olup, kitabı oluşturan bölümlere ait alt nesnelerden farklıdırlar. Bu nesneler `Kitap` sınıfının kurucu metodu içinde, `kitap` bileşik nesnesi ile birlikte yaratılırlar. Bundan dolayı, daha sonra `kitap` nesnesini kendi adres alanına yükleyen her bir uygulamaya otomatik olarak yüklenirler.



Şekil 8.3. Üç yazar tarafından paylaşılan `kitap` bileşik nesnesi

8.3.2 Yazarlara Sunulan Kullanıcı Arayüzü

Bu bölümde, yazarlara bir grafik kullanıcı arayüzü (graphical user interace-GUI) üzerinden sunulan `kitap` bileşik nesnesinde yer alan metotların yapısı incelenecektir. Tanımlamaları Şekil 8.4'teki gibi olan bu metotların işlevleri Tablo 8.1'de verildiği gibidir.

```
boolean   bölüm_ekle(String bölüm_adı)

boolean   bölüm_ekle(String bölüm_adı,
                     String bölüm_başlığı)

boolean   bölüm_sil(String bölüm_adı)

Editör    bölüm_oku(String bölüm_adı)

Editör    bölüm_yaz(String bölüm_adı)

boolean   bölüm_adını_değiştir(String eski_bölüm_adı,
                     String yeni_bölüm_adı)

boolean   bölüm_başlığını_değiştir(String bölüm_adı,
                     String yeni_bölüm_başlığı)

Editör    içindekileri_getir()

void      fihriste_ekle(String anahtar_sözcük,
                     String bölüm_adı)

Vector    fihristte_ara(String anahtar_sözcük)

boolean   fihristten_sil(String anahtar_sözcük)

boolean   fihristten_sil(String anahtar_sözcük,
                     String bölüm_adı)
```

Şekil 8.4. `kitap` bileşik nesnesinin kullanıcı arayüzünde yer alan metotlar

Tablo 8.1. `Kitap` bileşik nesne sınıfında yer alan metotların işlevleri

Metot Adı	İşlevi
bölüm_ekle	Kitaba yeni bir bölüm/alt bölüm ekler (önsöz ve kaynaklar eklemek için de aynı metot kullanılır).
bölüm_sil	Var olan bir bölüm/alt bölümü kitaptan siler.
bölüm_oku	Daha önceden kitaba eklenmiş olan bir bölüm/alt bölümü yalnızca okuma düzeninde yazara getirir.
bölüm_yaz	Daha önceden kitaba eklenmiş olan bir bölüm/alt bölümü güncelleme düzeninde yazara getirir.
bölüm_adını_değiştir	Eski bölüm adını yenisi ile değiştirir.
bölüm_başlığını_değiştir	Kitapta var olan bir bölümün başlığını değiştirir.
içindekileri_getir	"içindekiler" bölümünü oluşturur ve yazara getirir.
fihriste_ekle	Verilen anahtar sözcüğü ve geçtiği bölüm/alt bölüm adını fihriste ekler.
fihristten_sil	Verilen anahtar sözcüğü fihristten ya tamamen siler, ya da yalnızca belirtilen bölüm/alt bölüm adını siler
fihristte_ara	Verilen anahtar sözcüğü fihristte arayarak, hangi bölüm/alt bölümlerde geçtiği bilgisini yazara getirir.

8.3.3 Uygulamanın Başlatılması

Bir uygulamanın DBNTO sistemine giriş yapması, yeni bir dağıtılmış bileşik nesne yaratması ve onu bir isimle kaydettirmesi veya daha önceden var olan bir dağıtılmış bileşik nesneyi kendi adres alanına yükleyerek onun üzerinde çağrılar yürütmesine ilişkin izlenen işlem adımları Bölüm 6.3'te ayrıntılı olarak incelenmişti. Şimdi, bir yazarın sisteme girişi üzerine yürütülen işlemler onları gerçekleyen komutlar düzeyinde sunulacaktır.

Bir uygulama DBNTO sistemine giriş için

```
new DBNTO_Server();
```

komutunu yürütür. Yeni bir kitap yazma uygulaması başlatılacak ise,

```
(1)  Kitap kitap = new Kitap();
(2)  register_class(Kitap,
         "http://sınıf/tanımlamalarının/bulunduğu/adres")
(3)  register_DBN(kitap, "Bilgisayar Ağları");
```

komutları yürütülür. Bu komutlardan, (1) "Kitap" sınıfından bir "kitap" bileşik nesnesi yaratır. (2) uzaktan dinamik yüklemeyi sağlamak üzere sınıf tanımlamalarının bulunduğu URL'yi DBNTO sistem koordinatörüne kaydettirir. (3) kitap nesnesini "Bilgisayar Ağları" adı ile kaydettirir. Ya da kitap nesnesi zaten daha önce bir başka uygulama tarafından yaratılmış ve sisteme kaydedilmiş ise, yeni katılan uygulama,

```
(1)  load_class(Kitap);
(2)  Kitap kitap = lookup("Bilgisayar Ağları");
```

komutlarını yürütür. (1) Kitap sınıfını dinamik olarak uzak alandan yükler ve (2) kitap nesnesine ilişkin kabuk nesneyi uygulamanın adres alanına yükler.

8.3.4 Yazarların Gerçekleştirebildikleri Bileşik Nesne İşlemleri

Yazarların kendilerine sunulan grafik kullanıcı arayüzü üzerinden ürettikleri istekler, Şekil 8.4'te verilen kitap bileşik nesnesinin metotları üzerinde yürütülen çağrılara dönüşerek işlenirler. Tablo 8.1'de işlevleri kısaca açıklanan bu metotların yapıları izleyen bölümlerde açıklanmıştır.

8.3.4.1 Kitaba Yeni Bir Bölüm Ekleme

Kitaba yeni bir bölüm eklemek isteyen bir yazar bölüm_ekle metodu üzerinde bir çağrı yürütür. bölüm_ekle metodunun iki farklı gerçeklemesi vardır. Bunlar;

```
boolean   bölüm_ekle(String bölüm_adı) ve
boolean   bölüm_ekle(String bölüm_adı,
                     String bölüm_başlığı)
```

şeklindedir. `Kitap` bileşik nesnesi arayüzündeki metotların sayılarını arttırmamak için önsöz ve kaynaklar için farklı ekleme, çıkarma, okuma ve değiştirme metotları yaratılmamış, onlar da birer bölüm olarak düşünülmüştür. Ancak, bölümlerde farklı olarak, bölüm adı ile birlikte bölüm başlığı da yer aldığı için, aynı metodun iki farkı gerçeklemesi oluşturulmuştur. Buna göre, bir yazarın ürettiği istek,

```
bölüm_ekle("Önsöz")
```

metodu üzerinde bir çağrıya dönüştüğünde, eğer daha önce yaratılmamış ise, "Önsöz" isimli bir alt nesne yaratılır ve yazarın ekranında, "Önsöz.txt" dosya adıyla bir düz yazı editörü açılır. Yazar yazma işlemini tamamladığında, saklayarak editörü kapatır.

Yazarın ürettiği istek eğer aynı metodun diğer gerçeklemesi üzerinde, örneğin;

```
bölüm_ekle("Bölüm_1" , "Giriş")
```

şeklinde bir çağrıya dönüşürse, eğer daha önce yaratılmamış ise, bir üst paragrafta anlatılan işlemlerin aynısı "Bölüm_1" için de yapılır.

`bölüm_ekle` metodu eklenecek alt bölümler için de çağrılabilir. Örneğin;

```
bölüm_ekle("2_1" , "İletişim Ağının Yapısı")
```

çağrısı ile bölüm adı "2_1", bölüm başlığı "İletişim Ağının Yapısı" olan yeni bir alt nesne yaratılır.

Yeni bir bölüm/alt bölümün yaratılması ile, `kitap_erişim` alt nesnesinin,

```
ekle(bölüm_adı, bağlantı_nesnesi)
```

metodu ve `kitap_planı` alt nesnesinin,

```
ekle(bölüm_adı, bölüm_başlığı)
```

çağrılarak yeni eklenen bölüme ilişkin bilgiler hem yerel, hem de uzak alanlardaki `erişim` ve `plan` tablolarında kaydedilir.

8.3.4.2 Var Olan Bir Bölümün Kitaptan Silinmesi

Kitapta var olan bir bölümü silmek isteyen bir yazar `bölüm_sil` metodu üzerinde bir çağrı yürütür. `bölüm_sil` metodunun yapısı,

```
boolean   bölüm_sil(String bölüm_adı)
```

şeklindedir ve daha önce yaratılmış olan bir bölüm veya alt bölümün kitaptan çıkarılması için çağrılır. Örneğin,

```
bölüm_sil("2_1")
```

şeklinde bir çağrı geldiğinde yürütülen işlem basamakları şunlardır:

- `kitap_erişim` alt nesnesinin,

  ```
  ara("2_1")
  ```

  metodu çağrılarak, parametre olarak gelen bölüm adı "2_1" aranır.

- Çağrı sonucu geri dönen değer `null` ise, daha önce böyle bir bölüm eklenmemiş demektir. Bu durumda geriye `false` değeri döndürülür.

- Çağrı sonucu geri dönen değer `null` değil ise, bölüm `2_1`'e karşılık gelen bağlantı nesnesi geri gelmiştir.

- Bu durumda, bağlantı nesnesinin "`sil`" metodu çağrılır. Bu çağrı kontrol nesnesinin "`sil`" metodu üzerinde bir metot çağrısına dönüşür.

- Kontrol nesnesi `ask_object(W)` çağrısı ile, yazma amacıyla alt nesneye erişme isteğinde bulunur.

- Yazma senkronizasyonu denetimlerinden sonra iznin gelmesiyle birlikte, kontrol nesnesi içinde yer alan alt nesneye ilişkin değişkene "`null`" atanır. Ardından, kontrol nesnesinin `end_of_use()` metodu diğer alanlarda da aynı işlemin yapılması için DBNS'ye gerekli mesajı iletecektir.

- Bölümün silinmesinin ardından, `kitap_erişim` ve `kitap_planı` alt nesnelerinin

  ```
  sil(bölüm_adı)
  ```

  metotları çağrılarak, silinen bölüme ait bilgi bloklarının bu alt nesnelerde tutulan tablolardan da çıkarılmaları sağlanır.

Bu arada, Tablo 8.1'de verilen `Kitap` sınıfı arayüzünde bulunan ve bölümler üzerinde işlem yapan metotlardan, `bölüm_oku` ve `bölüm_yaz` dışındakiler kitap planı üzerinde değişiklik yapan metotlar olduklarından, bu metotlara yapılan çağrılar sonucu, kitabın içindekiler bölümü de değişecektir. Bu yüzden, "`içindekiler_değişti`" adlı bir değişken bu durumu denetlemek için kullanılmaktadır. Başlangıç değeri `false` olan bu değişkenin değeri kitaba yeni bir bölüm eklendiğinde `true` olarak değiştirilir. İlerleyen bölümlerde açıklanacak olan `içindekileri_getir()` metodu çağrılıp, `İçindekiler` alt nesnesinin güncel bir kopyası oluştuğunda, bu değişkenin değeri yine `false` olarak değiştirilecektir.

8.3.4.3 Bir Bölümün Okunması

Kitapta var olan bir bölümü okumak isteyen bir yazar `bölüm_oku` metodu üzerinde bir çağrı yürütür. `bölüm_oku` metodunun yapısı,

```
Editör    bölüm_oku(String bölüm_adı)
```

şeklindedir ve var olan bir bölüm veya alt bölümün yazarın ekranında görüntülenmesi için çağrılır. `Editör` sınıfı Bölüm 8.2'de sözü edilen ve ayrıntıları ilerleyen bölümlerde açıklanacak olan düz yazı editörü için gerçeklenen sınıftır. Burada kısaca değinilecek olursa, `Editör` sınıf yapısı, yazarların yeni bir bölüm yaratma, okuma ve güncelleme işlemleri sırasında kullanacakları editörü gerçeklemektedir.

Bir bölümün okunması amacıyla örneğin,

```
bölüm_oku("2_1")
```

şeklinde bir çağrı geldiğinde yürütülen işlem basamakları şunlardır:

- `kitap_erişim` alt nesnesinin,

  ```
  ara("2_1")
  ```

  metodu çağrılarak, parametre olarak gelen bölüm adı "2_1" aranır.

- Çağrı sonucu geri dönen değer `null` değil ise, bölüm `2_1`'e karşılık gelen bağlantı nesnesi geri gelmiştir.

- Bağlantı nesnesinin `oku()` metodu çağrılır. Bu çağrı kontrol nesnesinin `oku()` metodu üzerinde bir metot çağrısına dönüşür.

- Kontrol nesnesi `ask_object(R)` çağrısı ile, okuma amacıyla alt nesneye erişme isteğinde bulunur.

- Okuma senkronizasyonu denetimlerinden sonra iznin gelmesiyle birlikte, "2_1.txt" adlı düz yazı tipindeki dosya açılarak `Editör` editöründe görüntülenir.

- Yazarın editörü kapatma işlemi ile okuma işlemi sonlandırılır.

8.3.4.4 Bir Bölüm Üzerinde Güncelleme Yapılması

Kitapta var olan bir bölümü güncellemek isteyen bir yazar `bölüm_yaz` metodu üzerinde bir çağrı yürütür. `bölüm_yaz` metodunun yapısı,

```
Editör     bölüm_yaz(String bölüm_adı)
```

şeklindedir ve var olan bir bölümün ekranda görüntülenmesi ve yazarın gerekli güncelleme işlemlerini yapması için çağrılır. Yazarın güncelleme işlemini bitirmesinin ardından editörü kapatması ile birlikte yazma işlemi tamamlanır.

Bir bölüm üzerinde değişiklik yapılması amacıyla örneğin,

```
bölüm_yaz("2_1")
```

şeklinde bir çağrı geldiğinde yürütülen işlem basamakları şunlardır:

- `kitap_erişim` alt nesnesine yapılan `ara()` metot çağrısı ile, bölüm `2_1`'e ilişkin bağlantı nesnesi alınır.

- Bağlantı nesnesinin `yaz()` metodu çağrılır. Bu çağrı kontrol nesnesinin `yaz()` metodu üzerinde bir metot çağrısına dönüşür.
- Kontrol nesnesi `ask_object(W)` çağrısı ile, yazma amacıyla alt nesneye erişme isteğinde bulunur.
- Yazma senkronizasyonu denetimlerinden sonra iznin gelmesiyle birlikte, "`2_1.txt`" adlı düz yazı tipindeki dosya açılarak editörde görüntülenir.
- Yazma işleminin tamamlanmasının ardından, yazarın editörü kapatma işlemi ile birlikte `2_1.txt`dosyası son haliyle saklanır ve diğer alanlarda bulunan alt nesne kopyaları güncellenir.

8.3.4.5 Bir Bölüm Adının Değiştirilmesi

Bir yazar daha önce, örneğin "`Bölüm_2`" adı ile oluşturulmuş bir bölümün adını "`Bölüm_3`" şeklinde değiştirmek isteyebilir (tabii ki, o anda kitapta "`Bölüm_3`" adıyla bir bölüm olmamalıdır, varsa bu çağrıdan `false` değeri ile geri dönülür.). Böyle bir değişiklik durumunda, "`2_`" ile başlayan tüm alt bölüm adlarının da ("`2_1`", "`2_1_2`", ... gibi) değiştirilmesi gerekir. Benzer şekilde, eğer bölüm adı değiştirme isteği bir alt bölüm adı için geliyorsa (örneğin, "`2_1`" yerine "`2_3`"), o alt bölümün adı ile başlayan diğer alt bölümlerin adlarının da değiştirilmesi gerekir. Bu amaçla, bir bölüm ya da alt bölümün adını değiştirmek isteyen bir yazar, `bölüm_adını_değiştir` metodu üzerinde bir çağrı yürütür. `Bu` metodun yapısı aşağıdaki gibidir.

```
boolean   bölüm_adını_değiştir(String eski_bölüm_ad,
                               String yeni_bölüm_ad)
```

Bir bölüm adının değiştirilmesi amacıyla örneğin,

```
bölüm_adını_değiştir("Bölüm_3", "Bölüm_2")
```

şeklinde bir çağrı geldiğinde yürütülen işlem basamakları şunlardır:

- Parametre olarak gelen her iki bölüm adı `kitap_erişim` alt nesnesinde aranır.

- Eski bölüm adının tabloda olmaması ya da değiştirilmek istenen ad ile daha önceden kaydedilmiş bir bölümün var olması durumunda, geriye `false` değeri döndürülür.
- Aksi durumda, bölüm adının analizine geçilerek, eski ve değiştirilmek istenen adların bölüm/alt bölüm adı oldukları belirlenir.
- Buna göre, `kitap_erişim` alt nesnesinin,

  ```
  güncelle(eski_bölüm_adı, yeni_bölüm_adı)
  ```

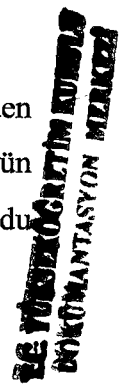
  ve `kitap_planı` alt nesnesinin,

  ```
  güncelle_ad(eski_bölüm_adı, yeni_bölüm_adı)
  ```

  metotları çağrılarak, `erişim` ve `plan` tablolarında ilgili bölüm adı ve o bölüm adı ile başlayan tüm diğer alt bölüm adları değiştirilir.
- "`içindekiler_değişti`" değişkeninin değeri `true` olarak değiştirilir.

8.3.4.6 Bir Bölüm Başlığının Değiştirilmesi

Kitaptaki bir bölümün veya alt bölümün tekrar düzenlenmesi sonucu daha önceden verilmiş olan bölüm başlığının değiştirilmesi gerekebilir. Bu amaçla, bir bölümün başlığını değiştirmek isteyen bir yazar, `bölüm_başlığını_değiştir` metodu üzerinde bir çağrı yürütür. Bu metodun yapısı aşağıdaki gibidir.



```
boolean   bölüm_başlığını_değiştir(
        String bölüm_adı, String yeni_bölüm_başlığı)
```

Bir bölüm başlığını değiştirilmesi amacıyla örneğin,

```
bölüm_başlığını_değiştir(
     "Bölüm_5", "Bilgisayar Ağlarında Güvenlik")
```

şeklinde bir çağrı geldiğinde yürütülen işlem basamakları şunlardır:

- `kitap_planı` alt nesnesinin,

  ```
  güncelle_başlık("Bölüm_5" ,
                  "Bilgisayar Ağlarında Güvenlik")
  ```

  metodu çağrılarak, ilgili bölüme yeni bir başlık atanır.

- "içindekiler_değişti" değişkeninin değeri true yapılır.

8.3.4.7 Kitabın İçindekiler Bölümünün Yazara Geri Döndürülmesi

Yazarlar tarafından yürütülen, bölüm_oku ve bölüm_yaz dışındaki tüm metot çağrıları kitabın planında değişiklik yapan çağrılardır. Bundan dolayı, yazarlar belirli anlarda kitap planının o anki mevcut şeklini görmek isteyebilirler. Benzer şekilde, müşterek yazışma ortamına yeni katılan yazar da kitap planının mevcut durumunu görmek isteyecektir. Bu amaçla, yazarlar içindekileri_getir adlı metoda bir çağrı yürüterek bu bilgiyi alabilirler. içindekileri_getir metodunun yapısı aşağıdaki gibidir.

```
Editör    içindekileri_getir()
```

Bu çağrı geldiğinde yürütülen işlem basamakları şunlardır:

- Daha önce oluşturulmamış ise, "içindekiler" adlı bir alt nesne oluştur. Bu alt nesne "içindekiler.txt" adlı yeni bir dosya açacaktır.
- Alt nesne daha önceden oluşturulmuşsa ve "içindekiler_değişti" değişkeninin değeri false ise, en son oluşturulduğundan bu yana kitap planında herhangi bir değişiklik meydana gelmemiş demektir. Bu durumda, "içindekiler" alt nesnesinin "görüntüle" metoduna bir çağrı gönderilerek, "içindekiler.txt" dosyasını yazarın ekranında görüntülenir.
- Aksi durumda, dosyanın içeriği tekrar oluşturulmalıdır. Bu amaçla;
  - kitap_planı alt nesnesinin,

    ```
    getir()
    ```

    metodu çağrılarak, plan tablosunda yer alan <bölüm_adı, bölüm_başlığı> bilgi çiftleri bir diziye alınır.
  - <bölüm_adı, bölüm_başlığı> bilgi çiftleri, bölüm adına göre sıralanır.
  - Sıralı <bölüm_adı, bölüm_başlığı> bilgi çiftleri "içindekiler" alt nesnesinin ekle metoduna yapılacak çağrı ile "içindekiler.txt" dosyasına yazılır. (Bu metot çağrısı sonlandığında tüm diğer alanlardaki

"içindekiler" alt nesneleri DBNTO sistemi tarafından otomatik olarak güncellenecektir.)

- "içindekiler_değişti" değişkeninin değeri false olarak değiştirilir.
- "içindekiler" alt nesnesinin "oku" metoduna bir çağrı gönderilerek, "içindekiler.txt" dosyası yazarın ekranında görüntülenir.

8.3.4.8 Fihrist üzerinde yürütülen işlemler

Fihrist üzerinde yürütülen işlemler, Tablo 8.1'de verildiği gibi, fihriste yeni bir anahtar sözcük ekleme, var olan bir anahtar sözcüğün fihristten tamamen çıkarılması ya da anahtar sözcüğün geçtiği bir bölüm adının çıkarılması ve fihristte bir anahtar sözcüğün aranmasıdır.

Fihrist bölümünü içeren Fihrist alt nesnesi kitap bileşik nesnesi ile birlikte yaratılmaktadır. Bu alt nesne, içinde <anahtar_sözcük, bölüm_adları_listesi> şeklinde kayıtlar içeren bir tabloyu yönetmektedir. Tablo üzerinde yukarıda verilen metotların gerektirdiği, ekleme, silme ve arama işlemleri yürütülür. Yazarlar bu işlemleri yapısı Şekil 8.5'te sunulan metotlar üzerinde çağrılar yürütmek suretiyle gerçekleştirirler.

```
void      fihriste_ekle(String anahtar_sözcük,
                        String bölüm_adı)
Vector    fihristte_ara(String anahtar_sözcük)
boolean   fihristten_sil(String anahtar_sözcük)
boolean   fihristten_sil(String anahtar_sözcük,
                        String bölüm_adı)
```

Şekil 8.5. Fihrist alt nesnesi tarafından sunulan metotlar

Bu bölümde, yukarıda verilen metotlarla yazarlara sunulan işlemlerin ayrıntıları açıklanacaktır.

Fihriste anahtar sözcük ekleme: Bir yazar fihriste bir anahtar sözcük eklemek amacıyla örneğin,

```
fihriste_ekle("haberleşme katmanı", "3_2_1")
```

şeklinde bir metot çağrısı yürüttüğünde;

- Anahtar sözcük tabloda aranır.
- Eğer `haberleşme katmanı` anahtar sözcüğü daha önce tabloya eklenmiş ise, karşılık gelen `bölüm_adları_listesi`'ne `3_2_1` (daha önce eklenmemiş ise) eklenir.
- Tabloda anahtar sözcüğe rastlanamamış ise,

  yeni bir `bölüm_adları_listesi` yaratılır ve `3_2_1` listeye eklenir.
- <`haberleşme katmanı`, `bölüm_adları_listesi` > tabloya eklenir.

Bir anahtar sözcüğü fihristte arama: Bir yazar fihriste bir anahtar sözcüğü aramak amacıyla örneğin,

```
fihristte_ara("haberleşme katmanı")
```

şeklinde bir metot çağrısı yürüttüğünde;

- Anahtar sözcük tabloda aranır.
- Eğer `haberleşme katmanı` anahtar sözcüğü daha önce tabloya eklenmiş ise, karşılık gelen `bölüm_adları_listesi` geri döndürülür.
- Tabloda anahtar sözcüğe rastlanamamış ise, geriye `null` değeri döndürülür.

Bir anahtar sözcüğün fihristten çıkarılması: Bir yazar fihristten bir anahtar sözcüğü tamamıyla çıkarmak amacıyla örneğin,

```
fihristten_sil("haberleşme katmanı")
```

şeklinde bir metot çağrısı yürüttüğünde;

- Anahtar sözcük tabloda aranır.

- Eğer `haberleşme katmanı` anahtar sözcüğü tabloda bulunursa, ilgili kayıt tablodan çıkarılır ve `true` değeri geri döndürülür.
- Tabloda anahtar sözcüğe rastlanamamış ise, `false` değeri geri döndürülür.

Bir bölüm adının fihristten çıkarılması: Bir yazar fihristten bir anahtar sözcüğe karşılık gelen bölüm adını çıkarmak amacıyla örneğin,

```
fihristten_sil("haberleşme katmanı")
```

şeklinde bir metot çağrısı yürüttüğünde;

- Anahtar sözcük tabloda aranır.
- Eğer `haberleşme katmanı` anahtar sözcüğü tabloda bulunursa, karşılık gelen `bölüm_adları_listesi`'nde ilgili bölüm adı aranır. Bulunursa, listeden çıkarılır ve `true` değeri geri döndürülür. Bulunamazsa, `false` değeri geri döndürülür.
- Tabloda anahtar sözcüğe rastlanamamış ise, `false` değeri geri döndürülür.

8.3.5 Kitabın Bölümlerini Oluşturan Alt Nesnelerin Yapısı

Daha önce de belirtildiği gibi, kitabı oluşturan parçalar; önsöz, içindekiler, her bir bölüm ve alt bölümler ile kaynaklar birer alt nesne olarak tasarlanmıştır. Kitap bölümlerinin yönetimi için "`Bölümler`" adlı bir sınıf yapısı oluşturulmuştur. Kitap sadece düz yazı metinlerini içerdiği için, oluşturulan alt nesne sınıfı, `Bölümler`, kendi içinde bir düz yazı metin dosyası ve bu dosya üzerinde sunulan basit dosya işlemleri ile, bölüm yazarlarına ait ad-soyad bilgileri, bölüm hakkında yazar görüşleri ve en son güncelleme tarihi gibi bilgileri yöneten metotlardan oluşmaktadır. `Bölümler` sınıfı içinde yer alan metotlar Şekil 8.6'da verilmiştir.

Bileşik nesne sınıfı `Kitap` ve alt nesne sınıfı `Bölümler` ile birlikte, basit bir editörü gerçekleyen `Editör` adlı bir sınıf yapısı daha oluşturulmuştur. `Editör` sınıfı kitaba yeni bir bölümün eklenmesi, var olan bir bölümün içeriğinde değişiklikler yapılması ve bir bölümün okunması için gerekli ortamı sağlamaktadır.

```
boolean     dosya_yarat(String dosya_adı)
void        dosya_oku(String dosya_adı)
void        dosya_güncelle(String dosya_adı)
void        son_güncelleme_tarihi()
Date        son_güncelleme_tarihini_getir()
void        yazar_listesine_ekle()
Editör      yazar_listesini_getir()
Editör      yazar_görüşlerine_ekle()
Editör      yazar_görüşlerini_getir()
```

Şekil 8.6. `Bölümler` alt nesne sınıfının bileşik nesne sınıfına sunduğu arayüz

Şekil 8.6'da yer alan bölümlere ilişkin alt nesne metotlarından yalnızca ilk üçü `Kitap` bileşik nesne sınıfına sunulan arayüzde yer alırlar. `son_güncelleme_tarihi()` ve `yazar_listesine_ekle()` metotları `dosya_yarat()` ve `dosya_güncelle()` metotlarının içinden çağrılır. Geri kalan dört metot olan, `son_güncelleme_tarihini_getir()`, `yazar_listesini_getir()`, `yazar_görüşlerine_ekle()` ve `yazar_görüşlerini_getir()`, yazarın herhangi bir bölüm üzerinde çalışıyorken editör penceresinden sunulan bir menüden seçeceği istekler doğrultusunda yürütülecek metotlardır.

Herhangi bir bölüm üzerinde güncelleme yapmış olan yazarların listesi, `Vector` tipinde `yazar_listesi` adlı bir değişkende tutulmaktadır. Her güncelleme işleminin ardından eğer daha önce eklenmemiş ise, yazara ait ad-soyad bilgisi `yazar_listesi`'ne eklenir. Ayrıca, yazarların bölüm üzerindeki görüşlerini belirtmeleri için düz yazı metni tipinde bir dosya tutulur. Bu dosya bölüm adının başına "`Görüşler_`" anahtar sözcüğü eklenerek elde edilir. Örneğin `Bölüm_1` adlı bir bölüm için görüşler dosyası, `Görüşler_Bölüm_1.txt` şeklindedir.

Şekil 8.6'da yer alan metotların işlevleri Tablo 8.2'de açıklandığı gibidir.

Tablo 8.2. `Bölümler` alt nesne sınıfında yer alan metotların işlevleri

Metot Adı	**İşlevi**
dosya_yarat	Verilen isimde "`txt`" uzantılı içi boş bir dosya editörde açılır. Yazarın editörü kapatması ile birlikte dosya saklanır.
dosya_oku	Verilen isimdeki dosya editöre getirilir. Editör kapatıldığında dosya saklanmaz.
dosya_güncelle	Verilen isimdeki dosya editöre getirilir. Editör kapatıldığında dosya üzerinde yapılan değişikliklerle saklanır.
son_güncelleme_tarihi	Yeni bir bölüm yaratıldığında ya da var olan bir bölüm güncellendiğinde, dosyanın kapatılma işleminden hemen sonra tarih bilgisini saklar.
son_güncelleme_tarihini_getir	Okuma ya da güncelleme amacıyla bir bölümün üzerinde çalışan bir yazarın isteği doğrultusunda, en son güncelleme tarihini getirir.
yazar_listesine_ekle	Yeni bir bölüm yaratıldığında ya da var olan bir bölüm güncellendiğinde, yazardan ad-soyad bilgilerini almak üzere, dosya_yarat ve dosya_güncelle metotlarının içinden çağrılır.
yazar_listesini_getir	Okuma ya da güncelleme amacıyla bir bölümün üzerinde çalışan bir yazarın isteği doğrultusunda, o ana kadar bölüm üzerinde çalışmış olan yazarların listesini getirir.
yazar_görüşlerine_ekle	Yazar bu metodu üzerinde çalıştığı bölüm hakkındaki görüşlerini belirtmek üzere çağırır.
yazar_görüşlerini_getir	Yazar bu metodu üzerinde çalıştığı bölüm hakkında diğer yazarlar tarafından belirtilen görüşleri listelemek üzere çağırır.

Yazarların kendilerine sunulan grafik kullanıcı arayüzü üzerinden ürettikleri istekler, bağlantı ve kontrol nesneleri üzerinden, Tablo 8.2'de verilen metotlara yönlendirilen

alt nesne çağrılarına dönüşürler. Bu metotlara yapılan çağrılar sonucunda yürütülen işlem adımları aşağıda sunulmuştur.

Dosya yaratma: Bileşik nesneyi oluşturan kabuk nesne üzerinde yürütülen,

```
bölüm_ekle("Bölüm_1" , "Giriş")
```

şeklindeki bir metot çağrısı ile bir alt nesne yaratıldıktan sonra, alt nesne üzerinde,

```
yarat("Bölüm_1")
```

şeklinde bir metot çağrısı yürütülür. Bunun sonucunda `yarat` metodu,

```
Editör("Bölüm_1"+".txt")
```

çağrısı ile yeni bir dosyayı editörde açar. Yazar, yazma işlemini tamamlayıp, editörü kapattığı anda;

- "`Bölüm_1.txt`" adlı düz yazı tipindeki dosya saklanır.
- Sistemden tarih bilgisi alınarak, son güncelleme tarihi olarak saklanır.
- Yazardan ad-soyad bilgisi alınarak, yazar_listesine eklenir.

Dosyadan okuma: Bileşik nesneyi oluşturan kabuk nesne üzerinde yürütülen

```
bölüm_oku("Bölüm_1")
```

şeklindeki bir metot çağrısı, ilgili alt nesne üzerinde,

```
oku("Bölüm_1")
```

şeklinde bir metot çağrısına dönüşür. Bunun sonucunda `oku` metodu,

```
Editör("Bölüm_1"+".txt")
```

çağrısı ile editörde `Bölüm_1.txt` dosyasını açar. Yazar, bu sırada ekranındaki pencerede açılan metin üzerinde düzeltme işlemleri yapsa bile, editörü kapattığında bu değişiklikler saklanmaz.

Dosya güncelleme: Bileşik nesneyi oluşturan kabuk nesne üzerinde yürütülen

```
bölüm_yaz("Bölüm_1")
```

şeklindeki bir metot çağrısı, ilgili alt nesne üzerinde,

```
yaz("Bölüm_1")
```

şeklinde bir çağrısına dönüşür. Bunun sonucunda `yaz` metodu,

```
Editör("Bölüm_1"+".txt")
```

çağrısı ile editörde `Bölüm_1.txt` dosyasını açar. Yazar ekranındaki pencerede açılan metin üzerinde düzeltme işlemlerini yapar. Editörü kapattığı anda;

- "`Bölüm_1.txt`" dosyası son haliyle saklanır.
- Sistemden tarih bilgisi alınarak, son güncelleme tarihi olarak saklanır.
- Yazardan ad-soyad bilgisi alınarak, eğer daha önce eklenmediyse, yazar_listesine eklenir.

Yukarıda açıklanan üç işlem sırasında yazar üzerinde çalıştığı bölümle ilgili olarak, son güncelleme tarihini görme, yazar listesini veya yazar görüşlerini alma gibi isteklerinin oluşması durumunda editördeki menüyü kullanarak bu işlemleri gerçekleştirebilir. Bu bilgiler açılacak yeni bir editörde görüntülenecektir.

Yazar bölümle ilgili bir görüş eklemek için istekte bulunduğunda, daha önce yaratılmamışsa yeni, yaratılmışsa var olan "`Görüşler_"+Bölüm_adı+".txt`" dosyası yazarın ekranında görüntülenir. Editörün kapatılması ile dosyanın son hali saklanır. Son güncelleme tarihini ve yazarlar listesini alma çağrıları da benzer şekilde yürütülürler. Son güncelleme tarihi ve listeden alınan yazarların ad-soyad bilgileri editörde görüntülenirler.

9. DENEYSEL SONUÇLAR VE DEĞERLENDİRME

Bu bölümde, tezde önerilen dağıtılmış bileşik nesne modelinin başarımını benzer özellikler taşıyan diğer sistemlerle karşılaştırmak amacıyla gerçekleştirilen testler ve elde edilen sonuçlar incelenecektir.

Dağıtılmış bileşik nesne modelinin başarımı değerlendirmek amacıyla sistem yapısı, çalışma ortamı ve uygulama alanı bakımından kendisi ile benzerlik gösteren ve Bölüm 2.2.3'te açıklanan "bölünmüş nesne modeli" sınıfına giren çalışmalar ile karşılaştırılabilir. Ancak, literatürde sözü edilen çalışmalar üzerinde uygulanmış testler ve bu testlere ilişkin sonuçlar ve değerlendirmeleri bulunmamaktadır. Gerçekçi bir karşılaştırma yapabilmek için, karşılaştırılacak olan sistemlerin aynı platform üzerinde, aynı şartlarda ve aynı kriterlerle test edilmeleri gerekir. Aksi takdirde, elde edilecek sonuçlar yanlış değerlendirmelerin yapılmasına neden olabilir. Bu amaçla, DBNTO sistemi yalnızca, Bölüm 3.6'da anlatılan Java'nın RMI yapısı [22] ile karşılaştırılmıştır. Ölçümler aynı test ortamında, aynı koşullarda ve aynı bileşik nesne örneği üzerinde yapılmıştır.

9.1 Test Ortamı

DBNTO ve Java RMI için yapılan tüm başarım ölçümleri, özellikleri Tablo 9.1'de verilen yapı üzerinde gerçekleştirilmiştir. Yerel alan ağının iş yükü günün değişik saatlerinde farklılıklar gösterdiği için ölçümler de günün farklı saatlerinde 10'ar kez tekrarlanarak uygulanmıştır. İzleyen bölümlerde verilecek olan ölçüm değerleri 10'ar kez uygulanmış olan bu testlere ait değerlerin aritmetik ortalamaları alınarak elde edilmiş olan sonuçlardır.

Her ne kadar testler yerel alan ağı üzerinde gerçekleştirilmiş olsa da, internet üzerinde yapılacak bir uygulamada, dağıtılmış bileşik nesne ortamına ait temel işlemlerde bir farklılık olmayacaktır. Yalnızca, müşterek uygulamaların birbirleri

arasındaki fiziksel uzaklıktan kaynaklanan iletim gecikmesi değişebilecek, tüm diğer parametreler sabit kalacaktır.

Tablo 9.1. Başarım ölçümlerinde kullanılan sistemin özellikleri

Özellik	Açıklama
Kullanılan ağ tipi	Yerel alan ağı (100 Mbit Ethernet).
Ağ işletim sistemi	Windows NT 2000.
Bilgisayar özellikleri	Pentium III 800 MHz işlemci, 256 MB RAM, Windows 2000 işletim sistemi.
Kullanılan bilgisayar adedi	5

9.2 Testlerin Uygulandığı Bileşik Nesnenin Yapısı

DBNTO'nun yapısı alt nesne işlemleri üzerine kurulduğu için, bir bileşik nesnenin içinde yer alan alt nesnelerin sayısının artması o bileşik nesne üzerindeki metot çağrılarının başarımını düşürecek bir etken değildir.

Java RMI ve DBNTO sistemi üzerinde yapılan test ölçümleri sırasında basit bir bileşik nesne yapısı kullanılmıştır. Bileşik nesnede bir kişinin kimlik ve araba bilgileri iki ayrı alt nesne içinde tutulmaktadır. Bu amaçla, her iki alt nesne için `Person.java` ve `Car.java` sınıfları, bileşik nesne için ise `Personal.java` sınıfı oluşturulmuştur. `Person.java` sınıf tanımlaması Şekil 9.1'de, `Car.java` sınıf tanımlaması Şekil 9.2'de ve `Personal.java` sınıf tanımlaması da Şekil 9.3'te sunulmuştur.

Java RMI'da tüm uzak metot çağrıları, sonuçta nesnenin bulunduğu bilgisayar üzerinden yerel olarak yürütüldüğü için, alt nesnelerin küçük taneli seçilmiş olmaları Java RMI'ya bir avantaj ya da dezavantaj sağlamamaktadır.

```
public class Sub_Person implements
                              java.io.Serializable {
      String      lastname;
      String      firstname;

      public Sub_Person() {
            lastname = new String("");
            firstname = new String("");
      }

      public void write_lastname(String name) {
            lastname = name;
      }

      public String read_lastname() {
            return lastname;
      }

      public void write_firstname(String name) {
            firstname = name;
      }

      public String read_firstname() {
            return firstname;
      }
}
```

Şekil 9.1. Kişi nesnesinin türetildiği `Person.java` sınıf tanımlaması

```
public class Sub_Car implements
                              java.io.Serializable {
      String      car_name;
      int         year;

      public void write_carname(String name) {
            car_name = name;
      }

      public void write_date(int date) {
            year = date;
      }

      public int read_date() {
            return year;
      }
}
```

Şekil 9.2. Araba nesnesinin türetildiği `Car.java` sınıf tanımlaması

```
public class Personal implements java.io.Serializable {
      Person     person;
      Car        car;
      String     lastname;
      String     firstname;
      String     carname;
      int        cardate;

      public Personal() {
            person = new Person();
            car = new Car();
            lastname = new String("");
            firstname = new String("");
            carname = new String("");
            cardate = 0;
      }

      public void put_lastname(String name) {
            person.put_lastname(name);
      }

      public String get_lastname() {
            lastname = person.get_lastname();
            return lastname;
      }

      public void put_firstname(String name) {
            person.put_firstname(name);
      }

      public String get_firstname() {
            firstname = person.get_firstname();
            return firstname;
      }

      public void put_car_name(String carname) {
            car.put_name(carname);
      }

      public void put_model_year(int date) {
            car.put_date(date);
      }

      public int get_model_year() {
            cardate = car.get_date();
            return cardate;
      }
}
```

Şekil 9.3. Kabuk nesnenin türetildiği `Personal.java` sınıf tanımlaması

9.3 Java RMI'dan Elde Edilen Test Sonuçları

Java RMI'nın sunduğu dağıtılmış nesne yapısı, Bölüm 2.2.2'de anlatılan, vekil tabanlı nesne modeli sınıfına girmektedir. RMI'da uzaktan metot çağrısı yapılacak olan nesnenin bir sunucu bilgisayar üzerine yerleştirilmesi gerekir. Bu nesneye uzak erişim için hem sunucu, hem de istekçi bilgisayar üzerinde birer vekil nesnesi bulunur (stub, skeleton). Ayrıca, uzak erişimin sağlanacağı nesne bulunduğu bilgisayarda, kullanıcı tanımlı bir "`ad`" ile kaydedilir. Bu nesne üzerinde uzaktan metot çağrısı yürütmek isteyen diğer alanlardaki istekçi programlar, sunucu bilgisayarın kayıt alanı üzerinde nesnenin adını vererek yapacağı bir arama işlemi ile, istekçi taraf vekil nesnesini (stub) kendi alanlarına yüklerler. Bu andan itibaren nesne üzerinde uzak metot çağrıları yürütülebilir. Java RMI'da metot çağrısı sırasında aktarılan ve geri dönen parametrelerin boyutları önem kazanmaktadır. Bu değerlerin büyük olması çağrının sonlanma zamanını doğrudan etkileyecektir.

Java RMI üzerinde uygulanan testlerde, yukarıdaki paragrafta anlatıldığı gibi, yaratılan uzak nesne öncelikle yerel sunucu bilgisayar üzerinde kaydedilmiştir. Daha sonra, diğer bilgisayarlarda bulunan istekçi programlar sunucunun kayıt alanı üzerinde nesne adını vererek yaptıkları arama işlemi ile vekil nesnesini kendi alanlarına yüklemişler ve uzak nesne üzerinde çağrılar yapmışlardır. Bu işlemler için yapılan ölçümlerde elde edilen sonuçlar Tablo 9.2'de sunulmuştur.

Tablo 9.2. Java RMI üzerinde yapılan testlerden elde edilen sonuçlar

İşlem	**Süre**
Uzak nesnenin sunucu bilgisayarda bir ad ile kaydedilmesi	2.17 ms
İstekçi programın sunucu bilgisayarın kayıt alanından yaptığı arama işlemi ile vekil nesneyi yüklemesi	1.75 ms
Bir uzak metot çağrısı (parametreli)	0.49 ms
Bir uzak metot çağrısı (parametresiz)	0.38 ms

Tablo 9.2'de verilen işlemlerden uzak nesnenin kaydedilmesi işlemi sunucu tarafında ve uzak nesneye ait vekil nesnesinin yüklenmesi işlemi de istekçi tarafında birer kez yürütülen işlemlerdir. Ancak, istekçinin uzak nesne üzerinde yürüteceği her bir metot çağrısı tablodan da görülebildiği gibi hemen hemen 0.5 ms sürmektedir. Aynı nesne üzerinde yerel olarak yürütülen metot çağrılarına ilişkin ölçüm sonuçları ise, parametreli bir çağrı için 610 ns ve parametresiz bir çağrı için 600 ns olarak ölçülmüştür. Buradan çıkan sonuç; bir yerel metot çağrısının sonlanma süresi ile uzak metot çağrısının sonlanma süresi arasında yaklaşık olarak on bin kat fark olduğudur.

9.4 DBNTO'dan Elde Edilen Test Sonuçları

Tezde önerilen dağıtılmış bileşik nesne ortamında aynı testleri uygulayabilmek için, öncelikle Şekil 9.2 ve Şekil 9.3'te verilen alt nesne sınıflarının arayüz tanımlamaları kullanılarak bağlantı ve kontrol nesnelerinin sınıf tanımlamaları üretilir ve ardından bileşik nesne yapısı oluşturulur (ayrıntılı bilgi için; bkz. Bölüm 7 ve Bölüm 4.4).

Bileşik nesne yaratıldıktan sonra;

- uygulama programı tarafından sunucu ve koordinatör bilgisayar üzerinde kullanıcı tanımlı bir ad ile kaydedilmesi,
- bileşik nesne üzerinde metot çağrısı yürütmek isteyen bir müşterek uygulamanın yürüteceği bir arama çağrısı ile bileşik nesneye ilişkin kabuk nesne ve bağlantı nesnelerini kendi alanına yüklemesi,
- eğer daha önce yüklenmemiş ise, kontrol nesnesi ve alt nesnelerin çağrının yapılacağı alana yerel olarak yüklenmeleri,
- bir okuma veya yazma çağrısının yürütülmesi, kullanılan tutarlılık protokolüne göre geçersiz kılma veya güncelleme işlemleri,

sürelerini içeren bilgiler Tablo 9.3'te sunulmuştur.

Tablo 9.3. DBNTO üzerinde yapılan testlerden elde edilen sonuçlar

İşlem	Süre
Yaratılan bileşik nesnenin kullanıcı tanımlı bir ad ile sunucu ve koordinatör üzerinde kaydedilmesi	1.25 ms
Müşterek bir uygulamanın bileşik nesneye ilişkin kabuk nesne ve bağlantı nesnelerini kendi alanına yüklemesi	3.15 ms
Eğer bir alt nesne üzerinde ilk çağrı yürütülüyorsa, kontrol nesnesi ve alt nesnenin istekçinin bilgisayarına yüklenme süresi	2.65 ms
Okuma erişimli metot çağrısı için; • Yerel alandan izin alma süresi • Uzak alandan izin alma süresi • Okuma çağrısının tamamlanması	 20 ns 0.75 ms 600 ns
Yazma erişimli metot çağrısı için; • Yerel alandan izin alma süresi • Uzak alandan izin alma süresi; • Geçersiz kılma tutarlılık protokolü uygulanıyor ise min. • Geçersiz kılma mesajı gönderilecek ilave her bir alan için • Güncelleme tutarlılık protokolü kullanılıyor ise • Yazma çağrısının tamamlanması	 20 ns 0.75 ms 0.72 ms 0.75 ms 610 ns
Eğer güncelleme tutarlılık protokolü kullanılıyor ise, bir yazma çağrısının sonlanmasının ardından, • Uzak bir alt nesnenin güncellenmesi • Güncelleme yapılacak ilave her bir alan için	 1 ms 0.95 ms

9.5 Elde edilen Test Sonuçlarının Değerlendirmesi

Tablo 9.3'te verilen test sonuçları incelendiğinde, bir bileşik nesneye ilişkin kabuk nesnenin kaydedilme ve müşterek bir başka alandan yüklenmesi işlemlerinde elde

edilen 1.25 ms kaydetme ve 3.15 ms uzaktan yükleme sürelerinin Java RMI'daki sonuçlar ile hemen hemen aynı olduğu görülmektedir. Java RMI'da istekte bulunulan alana uzak nesneye ilişkin erişim nesnesi, DBNTO'da ise bileşik neneye ilişkin kabuk nesne ve bağlantı nesneleri yüklenir. Sonuçta bu işlemler her uygulama için yalnızca bir kez yürütüleceğinden dolayı sistem başarımı üzerinde önemli bir etkisinin olmadığı açıktır.

Yürütülen bir DBNTO metot çağrısı, eğer henüz o alanda bir kopyası çıkarılmamış olan bir alt nesneye ilişkin ise, bu durumda o alt nesne ve ilgili kontrol nesnesinin birer kopyası çağrının yapıldığı alanda oluşturulacaktır. Kontrol nesnesi ilgili alanda kopyalandığı andan itibaren, dağıtılmış bileşik nesne uygulaması sonlanana kadar o alanda kalacaktır. Bundan dolayı, kontrol nesnesi ve alt nesnenin yüklenme sürecinin sistem başarımı üzerinde önemli bir etkisi yoktur.

Java RMI'da metot çağrısı sırasında aktarılan ve geri dönen parametrelerin büyüklüğü doğrudan çağrının sonlanma zamanını etkilemektedir. Örneğin, iki matrisi çarpıp, sonuç matrisini geri döndüren bir uzak nesne düşünüldüğünde, her çağrı için parametre olarak iki matris uzak alana ağ üzerinden aktarılacak ve sonuç matrisi de geri döndürülecektir. Oysa, DBNTO sisteminde tüm çağrılar yerel olarak yürütüldüğünden, aynı işlevi gerçekleştiren bir DBN'de parametre büyüklükleri sistem başarımını etkileyen bir etken olmayacaktır.

DBNTO sisteminin özellikle okuma çağrılarının yazma çağrılarına oranla ağırlıklı olduğu uygulamalarda çok iyi bir performans vereceği görülmektedir. Yerel bir Java metot çağrısı 370 ns'de sonlanırken, bir DBNTO çağrısı da 600 ns'de ve Java RMI çağrısı da yaklaşık 500.000 ns'de sonlanmaktadır. Burada ortaya çıkan sonuç oldukça dikkat çekicidir.

DBNTO sistemi yazma çağrılarının ağırlıklı olduğu bazı özel uygulamalarda da iyi sonuç vermektedir. Örneğin, belirli zaman aralıklarında bazı alanların diğerlerine oranla daha sık yazma işlemi yürütmeleri durumunda, alt nesne üzerinde erişim izni doğrudan yerel kontrol nesnesinden geleceğinden, yine Java RMI'ya oranla çok daha iyi bir başarım elde edilecektir.

10. SONUÇ VE ÖNERİLER

Bu tez çalışmasında, dağıtılmış nesneye-yönelik sistemler için uzak nesne modeline alternatif olarak kopyalamaya dayalı yeni bir model olan "dağıtılmış bileşik nesne" yöntemi ve bu yöntemi destekleyecek bir ara katman yazılımı olan "dağıtılmış bileşik nesne tabanlı ortam" sunulmuştur [4-6]. Önerilen sistem yaklaşımı, farklı alanlarda bulunan çok sayıdaki kullanıcı arasında bilgi paylaşımına ve müşterek iş yürütmeye imkan sağlayan uygulamalar için paylaşılan nesneler kümesi olarak düzenlenmiş, yeni bir dağıtılmış haberleşme ortamı sunmaktadır.

Günümüzde bir çok dağıtılmış nesneye-yönelik sistem uzak-nesne modelini benimsemiştir. Fakat, uzak-nesne modelinin özellikle ölçekleme konusunda bazı dezavantajları bulunmaktadır. Uzak-nesne modelindeki gibi desteklenen bir dağıtım saydamlığı, paylaşılan bir nesnenin kullanımını ve gerçeklenmesini basitleştirmekle birlikte, bilgili tasarımcıların dağıtımın getirdiği avantajdan yararlanmalarını da önler. Paylaşılan bir nesneye ait verinin veya kodun, çok sayıdaki alan üzerine başarım, kullanılabilirlik, koruma ve yük dengeleme amacıyla dağıtılmasına gereksinim duyulabilir. Dağıtılmış uygulamaların ölçeklenebilirliği ve başarımı için yerellik büyük önem taşır. Paylaşılan nesnelerin farklı alanlara kopyalanması etkin yerel erişim ve kullanılabilirlik açısından önemli yararlar sağlar. Ancak, uzak-nesne modeli bu konuda herhangi bir destek sağlamamaktadır.

Dağıtılmış ortamda müşterek yürütülen grup çalışmalarının özelliği, genelde paylaşılan nesnelerin büyük nesneler olmasıdır. Bu nedenle, kullanıcılara yine aynı şekilde büyük bir nesneyi paylaşıyor görüntüsü sunarak, daha alt seviyede paylaşımın sağlandığı bir nesne modeli geliştirilmiştir. Önerilen modelde, tasarım aşamasında büyük bir nesnenin çok sayıdaki alt nesnelerin birleşiminden oluşan bir bileşik nesne olarak yaratılması ve istekte bulunulan uzak alanlar üzerinde nesnenin sadece yapılan metot çağrısının gerektirdiği ilgili alt nesneleriyle kopyalanması öngörülmüştür. Bu arada, ortaya çıkan parçalı nesne yapısının istekçilere

gösterilmeden, onların yine tekil bir nesne ile işlem yapıyorlarmış görüntüsü altında erişimlerini gerçeklemelerini sağlayacak bir saydam model amaçlanmıştır.

Dağıtılmış bileşik nesne modeline göre, dağıtılmış ortamdaki uygulamalar bir bileşik nesneyi oluşturan alt nesneler üzerinden haberleşirler ve birbirleri ile etkileşimde bulunurlar. Bu amaçla, paylaşılan her bir bileşik nesne bir arayüz üzerinden kullanılabilen bir metotlar kümesi sunar. Nesneler pasiftir. Etkinlik, nesneleri paylaşan ve onların metotları üzerinde eşzamanlı çağrılar yürütebilen uygulama programları tarafından sağlanır.

Önerilen modelde, dağıtılmış bileşik nesneyi oluşturan alt nesnelerin farklı alanlar üzerinde rahatlıkla bağlanabilmesi amacıyla bir bağlantı nesnesi ve alt nesne kopyalarının senkronizasyon ve tutarlılığının sağlanabilmesi amacıyla da bir kontrol nesnesi her bir alt nesnenin önüne eklenmiştir. Bağlantı nesnesi ile dinamik bağlanma sağlanırken, kontrol nesnesi ile, dağıtılmış bileşik nesnenin alt nesneleri bazında düzenlenebilen çeşitli tutarlılık ve senkronizasyon mekanizmaları kurulabilmektedir. Ayrıca, tüm bu gerçekleme ayrıntıları kullanıcıya saydam olarak gerçekleştirilmektedir.

Bağlantı ve kontrol nesnelerinin üretildikleri sınıf tanımlamaları alt nesne sınıflarının arayüz tanımlamalarından yararlanılarak bir sınıf üreteci tarafından otomatik olarak üretilmektedir. Otomatik sınıf üretme işleminde tasarımcıya düşen görev, sadece bir alt nesne metodundaki erişim düzeninin hangi tipte (okuma/yazma) olduğunu belirlemek ve arayüz tanımlama dosyasını ona göre hazırlamaktır.

Dağıtılmış ortamdaki nesnenin alt nesnelere bölünüp, fiziksel olarak farklı adres alanlarında kopyalanmasını öngören DBN modeli ile kazanılan yararlar şu şekilde sıralanabilir:

- Kopyalama sonucu hataya hoşgörülü sistemlerin oluşturulabilmesi.
- Paylaşılan bir nesnenin bölünmesi sonucu, alt nesnelere paralel erişimin en yüksek düzeye çıkarılabilmesi.
- Yalnızca alt nesnelerin kopyalanması sonucu, ağda taşınan veri miktarının azalması.

- Yalnızca hedeflenen alt nesnelerin kopyalanması sonucu, verimli bellek kullanımı.

- Tutarlılık mekanizmalarının alt nesneler bazında düzenlenmesi sonucu, güncelleme amacıyla ağda aktarılacak veri miktarının azalması.

- Farklı alanlar üzerinde kopyalanmış olan alt nesnelere erişimi denetlemek amacıyla "çoklu-okuyucu, tek-yazıcı" eşzamanlılık kontrol yönteminin uygulanması.

- Tutarlılığı sağlama amacıyla, aynı DBN'nin farklı alt nesneleri için "yazma-geçersiz kılma" veya "yazma-güncelleme" tutarlılık protokollerinden herhangi birinin kullanılabilmesi.

10.1 DBNTO Ara Katman Yazılımının Sağladığı Katkılar

Dağıtılmış bileşik nesne modelini destekleyen ara katman yazılımının sağladığı katkılar şu şekilde özetlenebilir:

- Bir dağıtılmış bileşik nesnenin merkezi bir ortamda, tek bir bilgisayar üzerinde çalışacakmış görüntüsü altında basit bir şekilde tasarlanabilmesi.

- Bileşik nesneye, dağıtılmış ortamda kullanılabilme özelliğini kazandıran bağlantı ve kontrol nesnelerinin üretildiği sınıfların bir sınıf üreteci tarafından otomatik olarak üretilebilmesi.

- Bileşik nesnenin istekte bulunulan alanlara dinamik olarak yayılması. Eğer daha önce yüklenmemiş ise, bileşik nesneyi ifade eden kabuk nesne ile alt nesnelere ait sınıf tanımlamalarının istekçi alana dinamik olarak yüklenmesi.

- Bileşik nesneyi oluşturan alt nesnelere yenilerinin eklenmesi veya var olanlardan bazılarının bileşik nesneden çıkarılabilmesi gibi uyarlama işlemlerinin yürütme sırasında dinamik olarak yapılabilmesi.

- Yürütme sırasında bileşik nesnenin yalnızca ihtiyaç duyulan ilgili alt nesnelerinin istekçi alanlar üzerinde kopyalanması.

- Tutarlılık mekanizmasının alt nesneler bazında düzenlenebilmesi.
- Uygulanan testlerden elde edilen sonuçlardan da görülebileceği gibi, metot çağrılarının yerel olarak yürütülmesi sonucu, DBNTO sisteminin başarımını artması.
- Java dilinde kod üretebilen bir kullanıcının yeni bir kaç komut ile rahatlıkla sisteme adapte olabilmesidir.

10.2 Geliştirme Önerileri

Tezde önerilen dağıtılmış bileşik nesne modelinin özellikle internet üzerinde müşterek çalışmayı destekleyen CSCW uygulamaları için uygun bir yapı olduğu görülmektedir. Bu çalışmada, çok sayıdaki istekçi arasında paylaşımlı olarak kullanılan dağıtılmış bileşik nesnenin alt nesneleri arasında nesne bazında uygulanabilen, "yazma-geçersiz kılma" ve "yazma-güncelleme" şeklinde iki tutarlılık protokolü sunulmuştur.

Daha önce de belirtildiği gibi, değişik uygulama türlerinin isterlerini karşılamak üzere önerilebilecek, sistem başarımını arttırmaya yönelik bir çok farklı tutarlılık protokolü mevcuttur. Geliştirilen çalışma ortamını, diğer protokolleri destekleyecek şekilde zenginleştirmek DBN modelinin uygulama alanını genişletecektir. Tasarım aşamasında bu nokta dikkate alınarak esnek bir yapı oluşturulmaya çalışılmıştır. Öyle ki, yeni protokollerin desteklenmesi için, yalnızca kontrol nesnesinin yapısına bazı eklemelerin yapılması yeterlidir. DBN modeli ve DBNTO ara katman yazılımında herhangi bir değişiklik yapılmasına gerek yoktur. Her tür protokolün gerçeklenmesi için gerekli alt hizmetler zaten sunulmaktadır.

Kontrol nesnesinin yaratıldığı sınıf tanımlaması otomatik sınıf üreteci tarafından üretildiği için, söz konusu eklemeler yalnızca sınıf üretecine yapılacaktır. Bu amaçla, yeni tutarlılık protokolleri belirlenecek ve onların gerçeklemelerine ilişkin kod parçaları uygun yerlere eklenecektir.

KAYNAKLAR

[1] **Eddon, G., Eddon, H.,** 1998. Inside Distributed COM, *Microsoft Press,* Redmond, WA.

[2] **Rosenberry D., Kennedy D. and Fisher G.,** 1992. Understanding DCE, *O'Reilly and Associates*, Sebastopol , California.

[3] **Object Management Group,** 1999. The Common Object Request Broker: Architecture and Specification. Revision 2.3.1. OMG Document formal/99-10-07, Object Management Group, Framingham, MA, October.

[4] **Yılmaz, G. ve Erdoğan, N.,** 2000. Geniş Alanlı Ağlar İçin Bir Dağıtılmış Kopyalı Nesne Modeli, IEEE SIU-2000 8. Sinyal İşleme ve Uygulamaları Kurultayı, 14-17 Haziran, Belek, Antalya.

[5] **Yılmaz, G. ve Erdoğan, N.,** 2001. İnternet Ortamında Ortaklaşa İş Yürütmeyi Destekleyen Yeni Bir Nesne Modeli ve Programlama Arayüzü, I. Ulusal İletişim Sempozyumu, 17-21 Ekim, Ankara.

[6] **Yılmaz, G. and Erdoğan, N.,** 2001. A New Distributed Composite Object Model For Collaborative Computing, Proc. 16th International Symposium on Computer and Information Sciences (ISCIS XVI), 5-7 November, Antalya, Turkey.

[7] **Gosling, J., Joy B. and Steele, G.,** 1996. The Java language Specification, *Addison-Wesley Developers Press,* Sunsoft Java Series.

[8] **Neuman, B.C.,** 1994. Scale in Distributed Systems, In *Readings in Distributed Computing Systems,* IEEE Computer Society Press.

[9] **Baentsch, M., Baum, L., Molter, G., Rothkugel, S. and Sturm, P.,** 1997. Enhancing the Web's Infrastucture: From Caching to Replication, *IEEE Internet Computing,* 1(2):18-27, March.

[10] **Dingle, A., Partl, T.,** 1996. Web Cache Coherence, *Computer Networks ISDN Systems,* 28(7-11):907-920.

[11] **Snyder, A.,** 1993. The Essence of Objects: Concepts and Terms, *IEEE Software*, pp. 31-42, January.

[12] **Shapiro, M., Dickman, P. and Plainfosse, D.,** 1992. SSP Chains: Robust, Distributed References Supporting Acylic Garbage Collection, Rapport de INRIA, no 1799 – Broadcast Technical Report, No 1, November.

[13] **Sunderam, V.,** 1990. PVM: A Framework for Parallel Distributed Computing, *Concurrency: Practice and Experience*, 24(4), pp. 315-339, December.

[14] **Snir, M., Otto, S., Huss-Lederman, S., Walker, D. and Dongarra, J.,** 1996. MPI: The Complete Reference, *MIT Press*, Cambridge, MA.

[15] **Levy, E. and Silberschatz, A.,** 1990. Distributed File Systems: Concepts and Examples, *ACM Computing Surveys*, 21(4), pp. 321-375, December.

[16] **Mitchell, J.G., Gibbson, J.J., Hamilton, G. and Kessler, P.B.,** An Overview of the Spring System, In Proc. Compcon Spring 1994, IEEE, February.

[17] **Black, A., Hutchinson, N., Jul, E., Levy, H. and Carter, L.,** 1987. Distribution and Abstract Types in Emerald, *IEEE Transaction On Software Engineering*, vol. SE-13, no. 1, January.

[18] **Caughey, S.C., Parrington, G.D. and Shrivastava, S.K.,** 1993. SHADOWS: A Flexible Support System for Objects in Distributed Systems, IWOOOS-93, North Carolina.

[19] **Xerox Group,** Interlanguage Unification, ftp://parcftp.parc.xerox.com/pub/ilu/ilu.html

[20] **Grimshaw, A. and Wulf, W.,** 1997. The Legion Vision of A Worldwide Virtual Computer, *Communication of the ACM*, 40(1), pp. 39-45, January.

[21] **Globus Group,** Globus Project and Approach, http://www.globus.org/globus/information.html.

[22] **SUN Microsystems, Inc.,** 2000. Java Remote Method Invocation Specification. http://java.sun.com/products/jdk/1.3/docs/guide/rmi/

[23] **Hamilton, G., Powell, M. and Mitchell, J.,** 1993. Subcontract: A Flexible Base for Distributed Programming, In Proc. ACM 14th Symposium on Operating System Principles, Asheville, N.C., December.

[24] **Makpangou, M., Gourhant, Y., LeNarzul, J.P. and Shaphiro, M.,** 1994. Fragmented Objects for Distributed Abstractions, In T.L. Casavant and M. Singhal, (eds.), *Readings in Distributed Computing Systems,* pp. 170-186, IEEE Computer Society Press.

[25] **Steen, M.V., Homburg, P. and Tanenbaum, A., 1999.** Globe: A Wide-Area Distributed System, *IEEE Concurrency, 7(1),* pp. 70-78, January-March.

[26] **Steen, M.V., Hauck, F.C. and Tanenbaum, A.,** 1996. A Model for Worldwide Tracking of Distributed Objects, In Proc. TINA'96 Conference, Heidelberg, Germany, September.

[27] **Pacull, F., Sandoz, A. and Schiper, A.,** 1994. Duplex: A Distributed Collaborative Editing Environment in Large Scale, Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW).

[28] **Lugeon, J.C. and Pacull, F.,** 1995. A Light Weight Name Service and Its Use within a Collaborative Editor, In. Proc of IFIP Working Conference on Information Systems for Decentralized Organizations, Tranheim, Norway, August.

[29] **Hauck, F., Becker, U., Geicer, M., Meier, E., Rastofer, U. And Steckermeier, M., 1998.** AspectIX: An Aspect-Oriented and CORBA-Compliant ORB Architecture, Technical Report TR-14-08-08, IMMD IV, Univ. Erlange-Nürnberg, September.

[30] **Shapiro, M., Gourhant, Y., Habert, S., Mosseri, L., Ruffin, M. and Valot, C.,** 1989. SOS: An Object-Oriented Operating System - Assesment and Perspectives, *Computing Systems*, 2(4) pp.287-338, December.

[31] **Hagimont, D., Chevalier, P.Y., Freyssinet, A., Krakowiak, S., Lacourte, S., Mossierre J. and Rousset, X.,** 1994. Persistent Shared Object Support in the Guide System: Evaluation & Related Work, 9th Conference on Object-Oriented Programming, System, Languages and Applications (OOPSLA), Portland, October.

[32] **Homburg, P., Steen, M.V. and Tanenbaum, A.S.,** 1996. An Architecture for A Scalable Wide Area Distributed System, In Proc. 7. ACM SIGOPS European Workshop, Connemara, Ireland, September.

[33] **Mosberger, D.,** 1993. Memory Consistency Models, *Operating Systems Review,* 17(1): 18-26, January.

[34] **Lamport, L.,** 1978. Time, Clocks And The Ordering Of Events in A Distributed System, *Communications of the ACM,* 21(7): 558-565.

[35] **Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A. and Hennessy, J.,** 1990. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, *Computer Architecture News,* 18(2): 15-26, June.

[36] **Ahamad, M., Bazzi, R., John, R., Kohli, P. and Neiger, G.,** 1992. The Power of Processor Consistency, Tech. Report GIT-CC-92/34, Georgia Institute of Technology, Atlanta, GA 30332-0280, USA.

[37] **Lamport, L.,** How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Program, *IEEE Transaction on Computers,* C-28(9): 690-691, September.

[38] **Aho, A., Ravi, B. and Ullman, J.D.,** 1991. Compilers, Principles, Techniques and Tools, *Addison-Wesley Publishing Company,* Massachusetts.

[39] **Rein, G. L. and Ellis, C. A.,** 1993. RIBIS: A Real-Time Group Hypertext System, *Int. Journal of Man-Machine Studies*, pp. 339-367, August.

[40] **Haake, J. M. and Wilson, B.,** 1992. Supporting Collaborative Writing of Hyperdocuments in SEPIA, Proceedings of the CSCW Conference, pp. 138-146, November.

[41] **Sharples, M.,** 1993. Adding a Little Structure to Collaborative Writing, In *CSCW in Practice: An Introduction and Case Studies*, Diaper and C. Sanger (Eds.), pp. 51-68, Springer-Verlag, Germany.

[42] **Chang, K., Gong, Y., Dollar, T., Gajiwala, S., Lee, B., and Wear, A. W.,** 1995. On Computer Supported Collaborative Writing Tools for Distributed Environments, Proceedings of the ACM Computer Science Conference, pp. 222-229.

ÖZGEÇMİŞ

20 Ekim 1970 tarihinde İstanbul'da doğan Güray YILMAZ, 1987 yılında İnönü Endüstri Meslek Lisesi Elektrik Bölümünden mezun olmuştur. İstanbul Teknik Üniversitesi Elektrik-Elektronik Fakültesi Kontrol ve Bilgisayar Mühendisliği Bölümünden 1991 yılında mezun olmuş, aynı üniversitenin Fen Bilimleri Enstitüsü Kontrol ve Bilgisayar Mühendisliği Anabilim Dalı'ndaki yüksek lisans öğrenimini 1995 yılında tamamlamıştır. 1995 yılında aynı programda doktora öğrenimine başlamıştır. 1991 yılından bu yana Hava Harp Okulu Bilgisayar Mühendisliği Bölümünde öğretim üyesi olarak görev yapmaktadır.