İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

HAREKETLİ NESNELER İÇİN DİNAMİK RMI

YÜKSEK LİSANS TEZİ

Müh. Göksel SARIKAYA
(504031512)

Tezin Enstitüye Verildiği Tarih : 3 Mayıs 2006
Tezin Savunulduğu Tarih : 14 Haziran 2006

Tez Danışmanı :	**Prof. Dr. Nadia ERDOĞAN (İ.T.Ü)**
Diğer Jüri Üyeleri:	**Prof. Dr. Bülent ÖRENCİK (İ.T.Ü)**
	Prof. Dr. Coşkun SÖNMEZ (Y.T.Ü)

HAZİRAN 2006

ÖNSÖZ

Tez çalışmalarım süresince göstermiş olduğu yakın ilgi, destek, anlayış ve yardımlarından dolayı tez danışmanım sayın Prof. Dr. Nadia Erdoğan'a, yardımlarından dolayı dostlarıma ve hayatımın her döneminde olduğu gibi yüksek lisans eğitimim ve tez çalışmam boyunca da bana destek olan aileme duyduğum minneti bir kez daha dile getirmeyi bir borç bilirim.

Mayıs 2006 Göksel SARIKAYA

İÇİNDEKİLER

Sayfa No

KISALTMALAR

CCM	CORBA Component Model, CORBA Bileşen Modeli
COM	Component Object Model, Bileşen Nesne Modeli
CORBA	Common Object Request Broker Architecture, Ortak Nesne İstem Aracısı Mimarisi
DCOM	Distributed Component Object Model, Dağıtık Bileşen Nesne Modeli
D_RMI	Dynamic RMI, Dinamik RMI
EJB	Enterprise Java Beans
JVM	Java Virtual Machine, Java Sanal Makine
MİB	Merkezi İşlemci Birimi
NYP	Nesneye Yönelik Programlama
RMI	Remote Method Invocation, Uzak Metot Çağrısı

ŞEKİLLER

Sayfa No

HAREKETLİ NESNELER İÇİN DİNAMİK RMI

ÖZET

DCOM, CORBA ve RMI gibi bileşen modellerinin kullanıldığı dağıtık sistem tasarımları uygulama gereği ağ üzerinde yer değiştiren hareketli nesnelere sıkça yer verirler. Bu tür uygulamalarda, çalışmanın kesilmeden sürdürülebilmesi için sistemin dinamik olarak yeniden yapılandırılmasına gereksinim duyulur. Ancak, RMI ve benzeri geleneksel bileşen modelleriyle sistemi dinamik olarak yeniden yapılandırma mümkün olamaz. Durağan bir yapıya sahip olan RMI modelinde, haberleşen nesneler arasındaki bağlantı nesnelerin üzerinde yer aldıkları makinelerin adresleri kullanılarak, sabit olarak kurulur. Bu nedenle, hareketli bir nesnenin konumu değiştikten sonra, haberleştiği tüm nesneler ile arasındaki bağlar kopar ve artık uzak çağrıların yürütülmesi mümkün olamaz. Bu durumda çalışma ancak yer değiştiren nesne ile diğer tüm nesneler arasındaki bağlantıların yeniden kurulmasından sonra başlayabilir. Birçok uygulama için bu tür bir kesinti kabul edilemez.

Bu tez çalışmasında, bir hareketli nesnenin ağ üzerinde yer değiştirmesinden sonra da haberleştiği diğer nesneler ile arasındaki bağların sürekliliğini sağlayan bir altyapı tasarlanmış ve gerçeklenmiştir. Dinamik RMI olarak adlandırılan bu altyapı sayesinde, nesneler yer değiştirdikten sonra sistem dinamik olarak yeniden yapılandırılmaktadır. Uygulamaya saydam olarak gerçekleştirilen bu işlemler sayesinde dağıtık uygulama herhangi bir kesinti veya müdahale ile karşılaşmadan çalışmasını sürdürebilmektedir. Örneğin, bir istemci–sunucu uygulamasında, istemci nesnelerinin sunucu metotlarına erişimlerinin saydam bir şekilde gerçekleşmesi beklenir. Klasik RMI kullanımında, sunucunun yer değiştirmesiyle saydamlık ilkesi çiğnenecektir çünkü istemcilerin sunucu nesnenin yeni adresini edinip bağlantının yeniden kurulması için girişimde bulunmaları gerekecektir. Ancak Dinamik RMI altyapısı bütün bu işlemleri kendiliğinden yerine getirir ve nesne hareketliliğinin haberleşme üzerindeki olumsuz etkisini ortadan kaldırır. Böylece, uygulamada yer alan tüm nesneler birbirleriyle konumlarından bağımsız olacak şekilde haberleşmeyi sürdürürler; hareketlilik çalışmayı hiç bir şekilde etkilemez. Dinamik RMI, uygulama katmanı ile Java RMI katmanı arasında yer alacak şekilde tasarlanmıştır.

Ayrıca, bir nesnenin birden fazla kez yer değiştirmesi sonucunda, kendisine ulaşabilmek için birden fazla makine üzerinden atlamalı, görece uzun bir yolu izleme gereği ortaya çıkabilir. Tez kapsamında geliştirilen bir algoritma ile bu yol tek adıma indirgenebilmekte ve sistem başarımına önemli bir katkı sağlanmaktadır. Bir örnek uygulama olarak "Ağda Yük Dengeleme" problemine hareketli nesneler ve Dinamik RMI kullanılarak bir çözüm üretilmiştir.

DYNAMIC RMI FOR MOBILE OBJECTS

SUMMARY

Distributed applications that utilize component models such as DCOM, CORBA and RMI usually involve mobile objects which migrate from host to host on the web. In those kinds of applications, the system needs to be dynamically reconfigured in order to continue execution without any disruption during runtime. However, with RMI, as well as other conventional component models, dynamic reconfiguration of the system at runtime is not possible. The RMI component model has a static structure, where all connections between components must be set up before runtime to enable communication. For this reason, if an object migrates to a new location at runtime, all connections between it and other objects are broken, and consequently, remote method invocation becomes impossible. In such a case, communication may restart only after all connections between the migrating object and all other objects with which it communicates are reconfigured. This results in a disruption in the runtime, usually not tolerable for most applications.

In this thesis, a middleware structure, named Dynamic RMI, is designed and implemented to provide uninterrupted connections among distributed components in the presence of migration. Dynamic RMI takes the necessary actions for the dynamic reconfiguration of the system after migration of mobile objects. These operations are carried out transparently to the application; therefore, using Dynamic RMI, distributed systems can continue their execution without any disruption or interference. For example, in a classical client-server application, client needs to invoke remote methods of a server transparently. With conventional RMI, if a server object migrates to another machine the transparency principle will be violated, as the clients need to acquire the new address of the server object and must reconfigure their connections accordingly. However, the Dynamic RMI middleware automatically performs operations essential for the reconfiguration of the system when an object migrates, without requiring any effort on the application side. As a result, objects can continue to communicate with other objects transparently and independently of their locations, and mobility doesn't affect system integrity. Dynamic RMI is implemented as a middleware layer between application layer and the Java RMI layer.

Further, as a mobile object moves from one location to another, this may result in a long path that traverses several nodes between client and server objects. The greater the number of nodes with forwarding references to the server object, the greater is the method invocation latency. Therefore, in this thesis, a path compression algorithm is also designed to shorten such paths between clients and servers. An application, *Network Load Balancing*, has been implemented to demonstrate the use of mobile objects and Dynamic RMI in a distributed application.

1 GİRİŞ

Günümüzde istemci – sunucu uygulamalarının ve dağıtık nesne tabanlı sistemlerinin bilgisayar dünyasının her alanında sıklıkla kullanıldığı görünmektedir. Özellikle Internet'in oldukça popüler hale gelmesiyle birlikte bu sistemlerin esnek ve ortama dinamik olarak kolayca uyum sağlayabilecek şekilde geliştirilmeleri gerekliliği ortaya çıkmıştır.

Dağıtık nesne tabanlı uygulamalarda sıkça kullanılan orta katman bileşen çerçeveleri CORBA [1], COM (Component Object Model, Bileşen Nesne Modeli) [2], DCOM (Distributed Component Object Model, Dağıtık Bileşen Nesne Modeli) [2] , RMI [3] (Uzak Metot Çağrısı) ve EJB [4] (Enterprise Java Beans)'dir. Bu orta katman platformlar, çoğunlukla statik (sabit) uygulamalara olanak sağlarken, dağıtık sistemlerin dinamik olarak çalışır halde tutulmalarına çok küçük bir oranda olanak sağlarlar. Bu durumda da, yukarıda sayılan bileşen modellerinden birisini kullanarak gerçekleştirilen dinamik uygulamalarda, dinamizmi sağlayacak olan ek işler ve yükler tamamen bileşen geliştiricilerin üzerine bırakılmıştır [5]. Örneğin, RMI kullanılarak gerçekleştirilen bir uygulamada istemciler ve sunucu arasında dinamik olmayan bir şekilde bir yapı oluşturulur. Bu yapıda tüm istemciler sunucu nesnesinin adresini kullanarak teker teker sabit bir şekilde bağlanmaktadırlar. Bu sabit yapı korunduğu sürece uygulamanın çalışması, istemcilerin sunucudan hizmet almaları mümkündür. Aksi halde, mesela çalışma anında sunucu nesnesinin yer değiştirmesi durumunda, istemciler sunucudan hizmet alamazlar, çalışmada hata oluşur ve sonlanır. Çalışmayı yeniden gerçekleştirmek için tüm istemci nesneleri ile sunucu nesnesinin yeni yeri arasında her biri için yeniden bağlantı kurulması gerekmektedir. Bu da uygulama için oldukça maliyetli bir işlem olacaktır.

Tez çalışması olarak geliştirilen Dinamik RMI sisteminde ise sözü edilen dinamik yapı bileşen geliştiriciye ve uygulamaya ek bir yük getirmeden tamamen sistemin kendisi tarafından sağlanmaktadır. Bu temel yapı içerisinde farklı sunucu sınıfların kullanabileceği ve içerisinde nesnelerin yer değiştirme işlemlerini gerçekleştiren metotların bulunduğu sunucu temel sınıfı bulunmaktadır. Böylece bileşen geliştirici

bu temel sınıfı kullanarak fazladan bir işlem yapmadan uygulamasını dinamik hale getirmiş olur.

Dinamik RMI (D_RMI) ile çalışma anında nesnelerin yer değiştirmesi, diğer nesnelerin çalışmalarını hiçbir şekilde etkilememektedir ve sistemin bütünlüğü bozulmamaktadır. Böylece, sistem kendisini daima güncel tutar, sunucu nesne yürütme anında yer değiştirmiş olsa bile istemci nesneler, normal akışları dışında bir işleme gereksinim duymadan, sunucudan hizmet almaya devam ederler.

D_RMI'nın sağladığı bu esneklik ve dinamizm, dağıtık sistemlerin performansını da olumlu şekilde etkilemektedir. Örneğin, dağıtık sistemlerde, makine yükü, hafıza yeterliliği ve ağ yükü düşünülmesi gereken faktörlerdir [5]. Dinamik RMI ile geliştirilen dağıtık bir uygulamada, çalışma zamanında bir sunucu nesneyi aşırı yüklü bir makineden, kısmen daha az yüklü bir makineye taşıyabiliriz. Sistemin kendisini dinamik olarak güncellemesi özelliği sayesinde de bu taşınma işleminden istemci nesneler etkilenmez ve çalışmada herhangi bir aksama veya hata oluşmaz.

Gerçeklenen çalışma kapsamında, nesnelerin ard arda yer değiştirme işlemlerinden sonra, istemci ile sunucu nesneleri arasında oluşabilecek zincir şeklindeki uzun yolun minimum mesafeye düşürülmesini sağlayacak bir algoritma da bulunmaktadır. Bu algoritma sayesinde, sunucu nesne yer değiştirme sırasında onlarca atlama yapmış olsa bile, istemcinin sunucu metodunu ilk çağrısından sonra, istemci nesnesi sunucu nesnesine sadece bir adımda erişecek şekilde yol indirgenir.

Hareketli nesneler için Dinamik RMI olanağı sağlayan altyapının tasarım ve gerçekleme ayrıntıları tez kitabı içerisinde, JAVA ve yaygın bir orta katman platformu olan RMI'nın anlatılmasının hemen ardından anlatılacaktır. Daha sonra, Dinamik RMI'nın kullanımına bir örnek olarak ağda yük dengeleme konulu bir uygulamaya yer verilecektir.

2 JAVA PROGRAMLAMA DİLİ

Java platformu, bilgisayar ağının (network) önemi hesaba katılarak ve aynı yazılımın birçok değişik bilgisayar ortamında veya değişik tür makinelerde çalışması fikri ile geliştirilmiş bir teknolojidir. Java teknolojisi kullanılarak aynı uygulamayı kişisel bilgisayarlarda, Microsoft Windows, X Window sistemleri, Motif, OS/2, Macintosh bilgisayarlarda, hatta cep telefonlarında gibi değişik ortamlarda çalıştırılabilmektedir. Java ile herhangi bir ortamda oluşturulan bir yazılım hiçbir değişiklik yapılmadan, hatta tekrar derlenmeye bile ihtiyaç duyulmadan başka bir ortamda rahatlıkla kullanılabilir. Java dışında bir dilde program geliştirmek için genelde izlenen yol şudur. Bir işletim sistemi seçilir, bu işletim sistemine uygun olarak program yazılır ve daha sonra diğer işletim sistemleri için bu programlar yeniden düzenlenir. Bu çalışma düzeni büyük miktarlarda emek ve maddi kaybı beraberinde getirir. Java bu sorunu ortadan kaldırıp interneti bilgisayarınız haline dönüştürecek programlama dilidir. Herhangi işletim sistemi ve donanıma sahip bilgisayarların bir web sayfasındaki Java diliyle yazılmış bir programı indirip kullanabilmesi mümkündür [6].

Java platformu hem programlama dili, hem de bir ortam olarak düşünülebilir. Programlama dili olarak, açık, nesneye yönelik (Object-oriented), güvenli, sağlam, internet için elverişli bir teknoloji diyebiliriz. Ortam olarak da işletim sistemi, veri tabanı (database), ve orta katman (middleware) teknolojileri bulmak mümkündür [7].

Java platformu üç ana gruba ayrılır:

- Standart Java
- Enterprise Java
- Tüketici için ve gömülü cihazlar için Java (embedded devices)

Aşağıda bu platformlar ve sahip oldukları özellikler listelenmiştir:

Standart Java:

Java 2 SDK (J2SE) , Java 2 Runtime Environment, Java Plug-in, Java Web Start, Java HotSpot Server Virtual Machine, Collections Framework, Java Foundation Classes (JFC) , Swing Components, Pluggable Look & Feel, Accessibility, Drag and Drop, Security, Java IDL, JDBC, JavaBeans, Remote Method Invocation (RMI) , Java 2D [7].

Enterprise Java:

Enterprise JavaBeans (EJB) Architecture, JavaServer Pages (JSP) , Java Servlet, Java Naming and Directory Interface (JNDI) , Java IDL, JDBC, Java Message Service (JMS) ,Java Transaction (JTA) , Java Transaction Service (JTS) , JavaMail, RMI-IIOP, Software Development Kit & Application Model, Java 2 SDK, Enterprise Edition (J2EE) , Sun BluePrints Design Guidelines for J2EE [7].

Tüketici için ve gömülü cihazlar için Java (embedded devices) :

Java 2 Platform, Micro Edition (J2ME technology) , Connected Device Configuration (CDC) , Connected Limited Device Configuration (CLDC) , C Virtual Machine (CVM) , K Virtual Machine (KVM) , PersonalJava, Java Card, JavaPhone API, Java TV API, Jini Network Technology, Mobile Information Device Profile (MIDP) [7].

2.1 Java'nın Tarihçesi

Java, James Gosling, Patrick Naughton, Chris Warth, Ed Frank ve Mike Sheridan tarafından Sun MicroSystems da 1991'de geliştirildi. İlk çalışan sürümü 18 ay aldı. İlk adı da OAK idi; fakat sonradan 1995 de JAVA olarak değiştirildi. Sonradan birçok insanın katkıları ile JAVA' nın şimdiki sürümü yaratıldı. Önceleri küçük elektronik makinelerinin kontrolünü yapabilecek platformdan bağımsız bir dil yaratma ihtiyacına yönelik oluşturulan JAVA, internetin hızlı gelişimi ile platformları birbirine bağlayan bir dil olarak lanse edildi. Her ne kadar C++ da bu iş için kullanılabilse de, her platformun kendisine has C++ derleyici (compiler) gereksinimi dolayısıyla ve bunun da çok pahalı bir işlem olması nedeniyle ihtiyaçlara ekonomik olarak cevap veremiyordu.

Bunun üzerine derlenmeden çalışabilecek bir programın daha ucuz ve verimli olacağı varsayılarak, JVM (Java Virtual Machine) oluşturuldu. JVM, JAVA kodlarını yorumlayarak çalışıyordu. JAVA kodlarının özelliği bytecode dediğimiz yapıdır. Bu Java uygulamalarına güvenlik ve taşınabilirlik (portability) sağlamaktadır. Çünkü JVM bu kodları yorumluyor ve üzerinde çalıştığı sisteme zarar verecek hiçbir kodu çalıştırmıyordu. Sonuçta JVM yorumlama yapan bir birim olarak performansı bir parça düşük olmasına rağmen, Internet dünyası için çok önemli bir adım oldu. JAVA' nın performans kaybını azaltmak amacıyla JIT Sun tarafından tasarlandı. Böylece JIT, JVM in bir parçası olarak ihtiyaç duyulan Java kodlarını anlık olarak derleyebiliyordu. JIT tüm kodu derleyemez, çünkü Java sadece yürütme anında yapılabilecek bazı kontroller içerdiğinden bu pek mümkün olamaz [8].

2.2 Java'nın Özellikleri

Java'nın yaygın olarak kullanılmasının başlıca nedenleri şunlardır:

2.2.1 Basitlik

Java öğrenmesi kolay bir dildir. Bir parça programlama geçmişi olan özellikle de C++ gibi nesne tabanlı bir dilde deneyimi olan kişi için Java'yı anlamak çok kısa bir zaman alır. Ayrıca yazılan kod da son derece basit ve anlaşılabilirdir.

2.2.2 Mimariden Bağımsız Olma

Java ile yazılan programlar her platforma çalışır. Programı başka platformlara aktarmak için güç sarf etmeye gerek yoktur. Bu sayede aynı anda, örneğin PC üzerinde Windows veya Linux ve Macintosh üzerinde MacOS ile çalışan kullanıcılar tamamen aynı program aracılığı ile etkileşimde bulunabilirler.

2.2.3 Nesneye Dayalı

Java Smalltalk'dan bu güne uzanan, "nesneye dayalılık" ilkesini zorunlu bir altyapı olarak kurmuştur. Bugün en yaygın kullanılan nesneye yönelik programlama dillerinden biri olan C++ dilinde nesne tabanlı kod ile nesne tabanlı olmayan kod bir arada yazılabilir; ancak Java'da bu mümkün değildir. Java'da yeni bir yapı nesneler ile oluşturulmak zorundadır. Bu yapı programcıyı nesne tabanlı düşünmeye zorlayacağından üretkenliği arttırır. Bu konuda C++ dan dahi daha fazla Nesneye Yönelik Programlama'ya (NYP) yönelik bir dildir. NYP' nin en önemli özelliği

soyutlamadır. Karmaşık yapıları daha kolay işleyip kullanabilmek için soyutlamaya ihtiyaç duymaktayız. Soyutlamanın en kolay yöntemi hiyerarşik soyutlamadır. Bu tip soyutlama kodlama için gereklidir. Proses tabanlı geleneksel sistemlerde oluşan veriler de bu tip bir soyutlama ile daha kolay yönetilebilir. NYP'nin üç temel prensibi şöyle sıralanabilir:

2.2.3.1 Sarma

Veri ve kodu bir araya tutmaya yarayan bir mekanizma olarak tanımlanabilir. Sarmanın temeli "Sınıf" (class) oluşturur. Sınıf bir nesnenin yapısını ve davranışlarını tanımlar. Nesneler, bağlı oldukları sınıfın birer örneği olarak tanımlanabilirler. Yani sınıf mantıksal bir yapı iken, nesne fiziksel bir gerçekliktir. Her sınıf metot ve değişkenlerden oluşur. Her metot ve değişkenin bir erişim kontrolü vardır: Herkese açık veya özel korumalı olabilir.

2.2.3.2 Kalıtım (Inheritance)

Bir objenin, diğer bir objenin özelliklerini aynen taşıma özelliğidir

2.2.3.3 Çokyüzlülük (Polymorphism)

Aynı metodun birden fazla ara yüz (interface) tarafından kullanılmasıdır [8].

2.2.4 Dağıtılmış Programlama

Java'da dağıtılmış programlama dil ile gelen bir özelliktir. Java'da geliştirme yapılırken Internet'te dağılma özelliği hep göz önüne alınır. RMI (Remote Method Invocation - Uzak Metot Çağırma) mekanizması sayesinde, sunucu istemcide çalışan Java programındaki bir metodu çağırıp geriye bir değer alabilir.

2.2.5 Çok Görevlilik (Multithreaded)

Çok görevlilik Java dilinin bir parçasıdır, ayrı bir kütüphane kullanılmasına gerek yoktur. Bu sayede, kolayca, program içerisinde programın birden fazla ufak kısımları arka planda, ya da aynı anda çalışabilir [9].

Çoğu programlama dilleri tek ipliklidir (single threaded). Tek iplikli ortamlarda; oluşturulacak uygulamanın bir anda sadece bir işle ilgileneceği prensibinden hareket edilir. Java' da ise (çok iplikli) aynı anda birden çok iş bir arada yürütülebilir. Bu düzen bir internet tarayıcısının çalışmasına benzetebiliriz; aynı anda internetten

müzik dinlenip sayfa aşağıya doğru taranırken, aynı anda internetten bir doküman da indirilebilir. Aslında yapılan iş işlemci zamanını iplikler arasında paylaştırmaktır. MİB çok hızlı bir şekilde iplikler arasında geçiş yaparak her iplikteki işin bir kısmını yapıp, bir diğer ipliğe geçer ve bu sürekli bir döngü halinde devam eder. Böylece tüm ipliklerdeki işler birlikte yapılır [6].

2.2.6 Hatasız kod

Java ile çalışma zamanında ortaya çıkan (runtime) hatalar çok zor gerçekleşir. Pek çok hata, kodun derlenmesi sırasında ortaya çıkar ve düzeltilir. Güçlü tip denetimi sayesinde bir nesnenin başka bir nesne yerine kullanılabilme durumları katı bir şekilde belirlenmiştir. Çalışma zamanında ortaya çıkabilecek hatalar (dosya bulunamaması, Internet bağlantısı kurulamaması) çerçevelenmiştir ve programcı aykırı durum yakalama mekanizması (Exception Handling) ile bu hataları dikkate almaya zorlanmıştır.

2.2.7 Güvenlik

Java programları çok çeşitli güvenlik seviyeleri ile birçok ihtiyaca cevap verecek şekilde çalışabilir. Bu özelliği sayesinde Internet'te bulunan Java appletleri güvenle çalıştırılabilir [9].

Doğal olarak kullanıcı (client) için bilmediği bir siteden uygulama indirip çalıştırması için o uygulamanın güvenli olduğundan emin olması gerekir. Girdiğiniz sayfada bulunan appletin bilgisayarınıza virüs bulaştırmasını veya başka bir saldırıyı önlemek için Java'da bir dizi mekanizma oluşturulmuştur. Java güvenlik mekanizması hiçbir appletin yerel makinedeki veya istemci üzerindeki veya uzaktaki bir dosya sistemini okumasına veya bu dosya sistemine bir şey yazmasına, sunucu dışındaki makineye bağlanmasına ya da bunların dışındaki tehlikeli işlemlere izin vermez. Tabi ki bu güvenlik kısıtlamaları javanın web uygulamaları olan appletler için geçerlidir. Yerel makinede çalışacak uygulamalar (applications) yerel diskten herhangi bir şey okuyabilir ve yerel diske yazabilirler [10].

2.2.8 Gürbüz

Java, hem derleme, hem de yürütme esnasında kontroller yapar. Örneğin, bellek yönetimi JVM tarafından yapılır ki böylelikle kullanıcı, artık kullanılmayan belleğin serbest bırakılması ile ilgilenilmek durumunda kalmaz [8].

2.3 JVM'nin Çalışma Düzeni

Java uygulamaları JVM (Java Virtual Machine) tarafından yorumlanır. JVM, işletim sisteminin bir üst düzeyinde yer alır. Bu sayede bir Java uygulaması değişik işletim sistemlerinde, herhangi bir değişiklik yapılmadan çalışabilir (Şekil 2.1).

Şekil 2.1: Platformlar

Java kodları Java derleyicisi tarafından Java'ya has bayt kodu (byte-code) formatında derlenir. Bayt kodu formatındaki Java kodu "Java yorumlayıcı" içeren herhangi işletim sistemi tarafından çalıştırılabilir. Java kodu her çalıştırıldığında yorumlayıcı tarafından makine diline çevrilir. Web sayfalarındaki appletler "bayt-kod" (class file) formatındadır. Sayfaya girildiğinde sayfadaki bayt-kod formatında derlenmiş applet internet tarayıcısının bir parçası olan JVM tarafından yerel makine koduna çevrilir ve appletin yerel bilgisayarda çalıştırabilmesini sağlar.
Java dinamik bir dil olduğundan Java kodunun makine diline çevrilip daha sonra kullanılmak üzere saklamasına izin verilmez. Java dilinde yazılmış bir program her çalıştırıldığında bayt-kod, yorumlayıcı tarafından makine diline çevrildiğinden Java yorumlanan bir dildir. Bu özelliğinden dolayı da C ve C++ ' a göre daha yavaş çalışır. Çünkü C ve C++' da kod doğrudan makine diline çevrilerek derlenir, dolayısıyla program her çalıştırıldığında kodun bir daha yorumlanmasına gerek yoktur. Bu özelliklerinden dolayı bu dillerin faklı donanım ve işletim sitemine sahip her tür makineye göre tekrar derlenmeleri gerekir [6].

Java'nın kaynak koddan çalıştırılmasına kadar geçirilen evreler:

Derleme Anında:

Şekil 2.2: Derleme Aşamaları

Çalışma Anında:

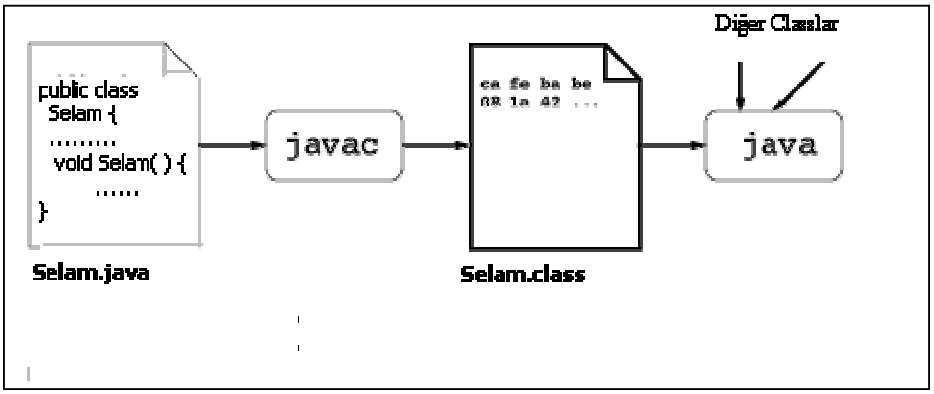
Şekil 2.3: Çalışma Zamanı

Bayt koduna çevrilen kaynak kod, JVM tarafından yorumlanır ve uygulama yürütülmeye başlanır.

Örnek Java Sınıfı:

```
public class Selam {
                public static void main(String args[]) {
                 System.out.println("Hello World !");
                        }
        }
```

Yukarıdaki örnek Java sınıfının derlenme ve çalışma aşaması şu şekilde gerçekleşmektedir.



Şekil 2.4: Java Sınıfının Çalışma Aşamaları

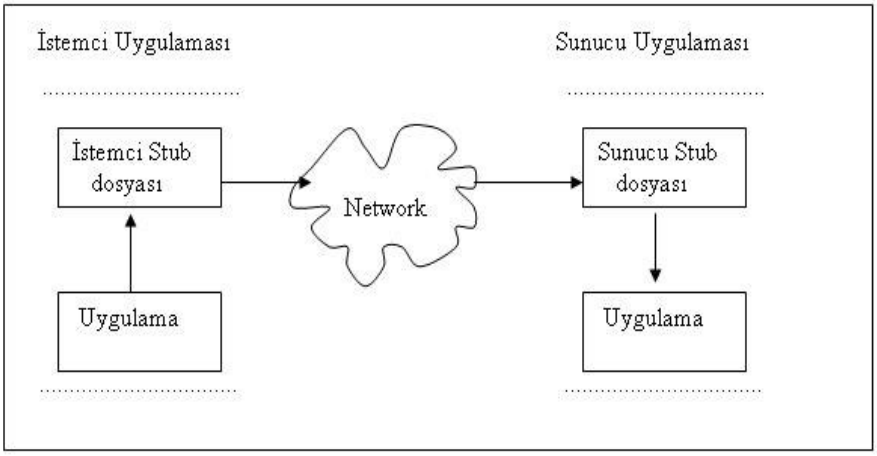
Yazılan Java kaynak kodları ilk önce derlenir (compile) daha sonra ise çalıştırılır. Java kaynak kodu içerisinde belirtilen her sınıf (class) için fiziksel olarak bir .class dosyası oluşturulur.

3 JAVA RMI (Uzak Nesne Çağrısı) (Remote Method Invocation)

Bu bölümde, Java RMI'nın yapısı, hangi katmanlardan oluştuğu, özellikleri anlatılacaktır. RMI'nın yapısı hakkında bu bilgiler verildikten sonra, standart kullanımının gösterimi örnek bir istemci sunucu uygulaması üzerinde anlatılacaktır.

3.1 Java RMI Nedir

Uzak Metot Çağrısı - Remote Method Invocation (RMI), platformdan bağımsız olarak farklı sanal Java makineleri üzerinde çalışan nesnelerin birbirleriyle iletişim kurup konuşabilecekleri bir Java mekanizmasıdır [11]. İnternet üzerinde farklı yerlerde bulunan ve TCP/IP ile bağlantısı sağlanmış sanal Java makineleri üzerindeki Java nesnelerinin, normal metot çağrıları kullanarak iletişimde bulunabilmesi için gerekli olan katmanları içermektedir. Java RMI kullanıldığında istemci ve sunucu da Java ile yazılacağı için dağıtılmış programlama tümüyle Java dilinde gerçekleşmektedir, böylelikle Java dilinin özellikleri ve avantajları dağıtılmış programlamada da kullanılabilmiş olmaktadır. Bu sayede ağ üzerinde bir makinede bulunan bir uygulama, başka bir makinede bulunan herhangi bir bileşenin bir metodunu çağırıp, o metottan dönen değeri elde edebilmektedir (Şekil 3.1) . Sonuç olarak, bu teknolojinin görevi ağ üzerinde dağıtılmış halde bulunan bileşenler ve uygulamaların haberleşmesini sağlamaktır.



Şekil 3.1: Network Üzerinden İstemcinin Sunucuya Ulaşması

Java RMI ağ işlem (networking) için kullanıcıya soket veya stream'lere göre daha üst düzeyde bir ara yüz sunmaktadır. Bu nedenle, RMI ile dağıtılmış programlama, soket ve stream kullanımına oranla daha az karmaşıktır. Programcı açısından bakıldığında ise, RMI kullanıldığında istemci/sunucu uygulamaların geliştirilmesi sırasında ağ işlem (networking) ayrıntıları (port ve soket ayarları) ile uğraşmak gerekmemektedir.

3.2 RMI Mimarisi

RMI sistemi; "Koçan / İskelet" (Stub / Skeleton) katmanı , "Uzak Referans" (Remote Reference) katmanı ve "İletim" (Transport) katmanı olmak üzere 3 katmandan oluşmaktadır.

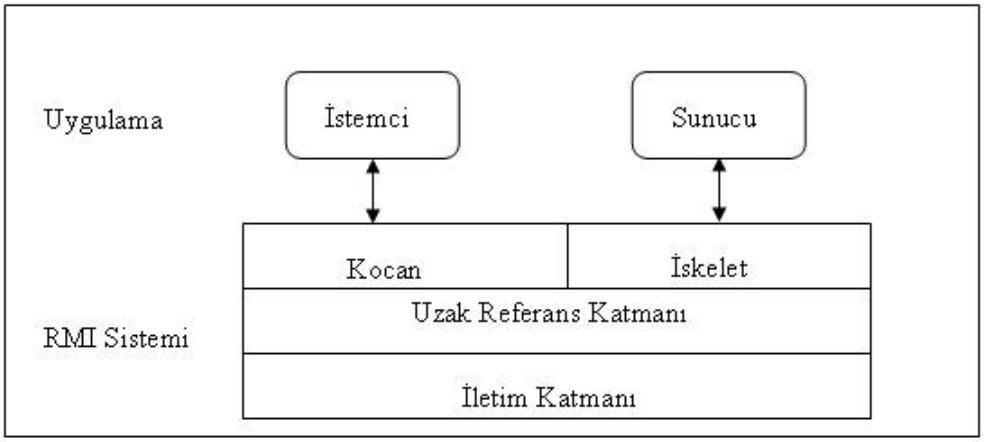
- Koçan / İskelet Katmanı :
  Bu katmanın istemci tarafındaki adı "koçan" , sunucu tarafındaki adı ise "iskelet"tir.

- Uzak Referans (Remote Reference) Katmanı :
  Uzaktan erişimdeki davranışların yönetildiği katmandır. Çağrının nasıl gerçekleşeceğini (unicast, multicast veya persistent olarak) istemci API kullanımını değiştirmeye gerek kalmadan belirlemekle sorumludur.

- İletim (Transport) katmanı :

  İki makine arasındaki bağlantı ayarlarının yapıldığı ve uzak nesne çağrısının takibinin yapıldığı katmandır (çağrının hedefi kim? vb.) [12].

Bu üç katman Şekil 3.2'de de görüldüğü gibi birbirlerinden ara yüz ve protokollerle tanımlanan sınırlarla ayrılmışlardır, bu yüzden birbirlerinden bağımsızdırlar ve üzerlerinde gerçekleşecek herhangi bir değişiklikte birbirlerini etkilememektelerdir. Örneğin , mevcut transport katmanı TCP tabanlı olabilir ve bu katman diğer katmanlarda bir değişikliğe yol açmadan UDP tabanlı olarak değiştirilebilir.



Şekil 3.2: RMI Sistemi Katmanları

Bu üç katman arasındaki ilişki şu şekilde gerçekleşmektedir :

Uzakta bulunan bir Java nesnesinin metotlarını çağırmak isteyen bir istemci ilk önce kendi tarafında bulunan "koçan" kod ile iletişime geçer. "Koçan" kod, isteği "Uzak Referans" katmanına, bu katman da "İletim" katmanına geçirir. Her iki bilgisayarın iletim katmanı aracılığıyla iletişime geçmesinden sonra, karşı taraftaki bilgisayarda istek önce "uzak referans" katmanına, oradan da "iskelet" koda iletilerek istenen servise ulaşılır. Servisin işlenmesinden sonra sunucunun elde ettiği sonuçlar da aynı yolu geriye doğru izleyerek istemciye iletilmektedir [11].

Aşağıda bu katmanların yapıları ve görevleri daha ayrıntılı olarak tanıtılmaktadır.

3.2.1 Koçan / İskelet Katmanı

Koçan / İskelet katmanı, uygulama katmanı ile RMI sistemi arasında bulunan ve bu iki katmanın ilişkisini sağlayan bir arayüzdür. İstemci, uzakta bulunan bir nesnenin metotlarını çağırmaya başladığı anda ilk olarak "koçan" kod ile iletişime geçecektir. İstemcinin uzaktaki nesne ile ilgili olarak elinde bulunan referans aslında yerel "koçan" koda olan referanstır. Karşı tarafta bulunan "iskelet" kod ise uzak nesnenin istenen metodunun gerçekleştiriminin çağrılması ve sonuçların elde edilmesinden sorumludur [12].

Burada “koçan”, istemci tarafındaki uzak nesnenin bir vekilidir. Ayrıca koçan, uzak nesnenin desteklediği ara yüzü (interface) gerçekler. İstemci tarafında çalışan bir koçan katmanının yaptığı işler sırasıyla şu şekildedir :

- Uzak referans katmanını çağırarak uzak nesneye çağrım işlemlerini başlatır.
- Nesne serilizasyon (nesnelerin byte stream'leri biçimine dönüştürülüp aktarılması) tekniği ile çağrıyı parametreleri ile birlikte paketler.
- Uzak referans katmanını “çağrıyı gerçekleştir” diyerek bilgilendirir.
- Karşı taraftan (sunucu tarafından) paket halinde gelen sonucu paketi çözerek elde eder.
- Uzak referans katmanını uzak nesne çağrısı tamamlandığı yönünde bilgilendirir.

İskelet ise, sunucu tarafında bulunan, ulaşılmak istenen uzak nesnenin metodunu çağıracak olan katmandır. İskeletin, bir RMI çağrısı sırasındaki gerçeklemesi gereken işler ise şunlardır :

- İstemciden gelen paketi açar ve parametreleri çözer.
- Uzak nesnenin istenen metoduna çağrıyı gerçekleştirir.
- Çağrılan metodun sonucunu alır.
- Sonucu paketleyerek, isteğin geldiği yoldan benzer biçimde karşı tarafa gönderir.

3.2.2 Uzak Referans (Remote Reference) Katmanı

Uzak referans katmanı, iletim katmanıyla koçan/iskelet katmanı ile bağlantı halindedir ve bu iki katman arasında bir köprü görevi görmektedir. Sunucu tarafında iletim katmanından gelen istekleri iskelet kodunun anlayacağı biçime dönüştürmektedir. İstemci tarafında ise koçan kodundan gelen çağrıları iletim katmanı biçemlerine dönüştürmektedir [12].

Ayrıca, metot çağrısı için, istemcinin koçan katmanından ve sunucunun iskelet katmanından bağımsız olarak uygulamaya özel bir uzak referans protokollerine sahiptir. Her bir uzak nesne uygulaması kendi uzak referans protokolünü aynı tabaka içerisinde alt sınıf olarak seçebilir. Böylece farklı metot çağrısı protokolleri tek bir uzak referans katmanı içerisinde bulunabilir. Bu uygulamalar için esneklik getirmektedir.

Bahsedilen bu özel protokollere örnek olarak :

- Uçtan uca metot çağrımı (Point-to-point invocation)
- Kopya nesne grubu çağrımı (Replicated object invocation)

verilebilir.

Metot çağrımı sırasında istemci ile sunucu kendi özel uzak referans işlemlerini gerçekleştirmektedir. Örneğin, bir uzak nesne, kopya nesne grubu çağrımı protokolüyle çalışmaktaysa, istemci tarafı tek bir nesneye çağrısı iletmek yerine her bir kopya nesne için metot çağrısını iletebilmektedir.

Bu protokoller istemci sunucu arasındaki bağlantının son derece kontrollü olmasını da sağlamaktadır. Örneğin, uzak erişim sırasında eğer uzak nesne ulaşılamaz durumdaysa yeniden bağlantı kurulmaya çalışılır.

3.2.3 İletim (Transport) Katmanı

Java RMI sistemin iletim katmanının görevleri; uzak bilgisayarın adres uzayına bağlantı kurmak, bağlantıyı yönetmek, kontrol etmek, bakımını yapmak ve bir bağlantı isteği gelip gelmediğini dinlemektir. Bu işlemler için Java soketleri kullanılmaktadır. Aynı zamanda iletim katmanında, aynı sanal makinede TCP , UDP gibi birden fazla protokol bir arada bulunabilmektedir. Bunun sebebi, RMI iletim katmanının bağlantı sırasında doğrudan uygulamayla iletişime girmeyip, sanal

makine ile bağlantı kurmasıdır. Aynı şekilde, TCP olan bir protokolü, UDP ile değiştirmek de RMI uygulamasındaki iletim katmanını etkilememektedir [12].

3.3 RMI Güvenlik

RMI sistemde, yerel bir yol (path) dan bir sınıfa bağlanılacaksa, bu sınıflar bir güvenlik yöneticisi (security manager) tarafından kontrol edilmez. Fakat bu sınıflara ağ üzerinden RMI sınıf yükleyici (RMI ClassLoader) tarafından bağlanılacaksa, ya bir güvenlik yöneticisi olmalıdır veya da bir aykırı durum uyarısı gönderilmelidir [12].

Bir Java programında yapılan işlemlerin güvenliğe göre denetlenebilmesi için ilk olarak güvenlik yöneticisi başlatılmalıdır. Başlatılan güvenlik yöneticisi, bağlanılan sınıfların standart Java güvenlik şartlarını sağlayıp sağlamadığını kontrol eder. Standart Java güvenlik şartları "RMISecurityManager" sınıfı içerisinde bulunmaktadır.

Uygulamalar kendi güvenlik yöneticisini tanımlamalı ya da RMISecurityManager' i kullanmalıdır. Eğer bunlardan birini kullanılmazsa, uygulama istediği sınıflara ağ kaynak kodlarından ulaşamayacaktır.

3.4 Artık Toplama (Garbage Collection)

Dağıtılmış sistemlerde, herhangi bir istemciye referans taşımayan uzak nesnelerin otomatik olarak silinmesi çok önemli bir ihtiyaçtır. Bu işlem, programcının, eriştiği uzak nesnelerin kayıtlarını tutma ihtiyacını ve hangi nesnenin artık kullanılmayacağı kontrolünü yapma gereğini ortadan kaldırmaktadır [12].

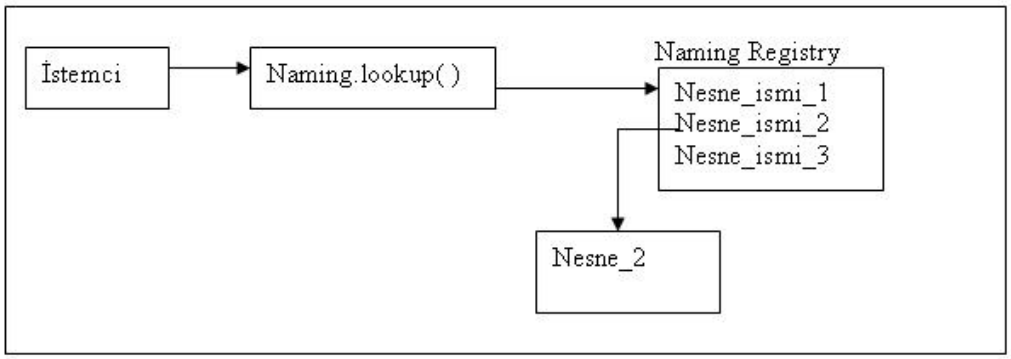
RMI'da artık toplama şu şekilde gerçekleşmektedir: RMI , çalışma zamanında o anda aktif olan tüm uzak nesne referanslarının kayıtlarını her bir sanal makinede tutmaktadır. Bir referans, Java sanal makinesine girince o referansın kayıtlarını tutan sayaç bir arttırılır. İlk referans sanal makineye girdiğinde referans ettiği nesne için sunucuya "referans ediliyor" mesajını gönderir. Yaşayan referans yerel sanal makinede "referans etmiyor" olarak bulunursa sayaç bir azaltılır. Sayaca ait son referans da silinirse, "referans edilmiyor" mesajı sunucuya gönderilir. Eğer, sunucuda bir nesnede "referans edilmiyor" etiketi bulunuyorsa, buna "zayıf referans" denir. Zayıf referans, Java sanal makinenin çöp toplayıcının o nesneyi silmesine

neden olur. Böylece, o nesne için çöp toplayıcı tarafından "finalize( )" metodu çağrılarak o nesnenin güvenli bir şekilde sonlanması sağlanır.

3.5 RMI İsimlendirme (Naming)

İstemcilerin uzaktaki nesnelerin metotlarını RMI ile çağrılabilmeleri için, söz konusu uzak nesnelere ilişkin referansları elde etmeleri gerekmektedir. RMI modelinde bu amaca yönelik olarak bir isim sunucu (name server) kullanılmaktadır. Sunucular uzak nesneleri java.rmi.Naming isimli bir sınıfta bulunan bind( ) metodunu kullanarak İsim Kayıtları (Naming Registry) servisine kaydettirmelidirler. İsimlendirme servisi, "rmiregistry" isimli bir komutu çalıştırarak sağlanmaktadır. İstemciler ise java.rmi.Naming isimli sınıfın lookup( ) metodunu kullanarak uzak nesnelerin hangileri olduğuna bakıp istenen uzak nesneye bir referans elde edebilmektedir.

Örneğin ; "nesne_ismi_2" uzak nesnesini Java sanal makinelerinde arayan bir istemcinin Naming.lookup işlemleri Şekil 3.3'te göründüğü gibi gerçekleşmektedir.



Şekil 3.3: İstemcinin Naming.lookup işlemleri

3.6 Dinamik Kod Yükleme ve Avantajı

RMI'nın en önemli özelliklerinden birisi, eğer alıcı sanal makinede çağırılan metodun ait olduğu nesnenin sınıfı tanımlanmamışsa, o nesnenin sınıfının bayt kodunu alıcı sanal makinenin indirmesidir. Nesnenin veri tipleri ve metotları daha önceden tek bir sanal makinede bulunmaktayken, RMI' daki bu yetenek sayesinde

bir başka sanal makineye (çoğunlukla uzak sanal makineye) gönderilmektedir. RMI nesneleri kendi gerçek tipleriyle başka sanal makinelere geçirildiği için, uzaktaki sanal makine içinde nesnenin metotları değişmeyecektir. Böylece, yeni tipler uzak sanal makinede tanımlanabilecek ve yeni metotlar eklenerek nesnenin metotları dinamik olarak geliştirilebilecektir.

3.7 Örnek RMI Uygulaması Gerçeklenmesi

RMI uygulamaları çoğunlukla istemci ve sunucu olmak üzere iki ayrı programdan oluşmaktadır. Tipik bir sunucu programı uzak nesneleri oluşturur ve bu metotlara sahiptir. Daha sonra, oluşturulan bu nesnelerin metotlarını istemcilerin çağırması için bekler [3].

Tipik bir istemci programı ise, sunucuda bulunan bir veya daha fazla uzak nesnenin uzak referansını alır ve metotlarını çağırır. Bu türden uygulamalara "Dağıtılmış nesne uygulaması" denir. RMI'ın görevi, istemci ile sunucunun haberleşmesini sağlamak ve bu ikisi arasında bilgi alışverişini gerçekleştirmektir.

Aşağıdaki uygulamada, bir Java RMI sunucusu ve istemcisi oluşturulması gösterilecektir. Örnekte, sunucu dört işlem yapabilen metotlara sahiptir. İstemci, yapmak istediği işlemi sayılarıyla beraber sunucuya bildirecek ve sonucu sunucudan alacaktır.

3.7.1 İstemci ve Sunucu Sınıflarının Oluşturulması

Bu bölümde genel olarak bir istemci ve sunucu yapısı nasıl oluşturulur adım adım anlatılacaktır.

3.7.1.1 Sunucu Arayüzünün Oluşturulması

RMI uygulamasında bulunması gereken 3 tanımlamadan birisi "Remote interface" denilen sunucu arayüzüdür. Bu arayüz uzak nesnede gerçekleştirimleri yapılacak olan metotların prototiplerini içermektedir. Uzak arayüz, java.rmi.Remote sınıfından türetilmeli ve içinde bulunan her metot java.rmi.RemoteException tanımlamalıdır. Parametre olarak geçirilmek istenen veya dönüş değeri olacak olan uzak nesneler de uzak arayüz türünde (sunucu sınıf türünde değil) tanımlanmalıdır.

Örnek uygulamamızda bulunan arayüz kodu Şekil 3.4 de görüldüğü şekildedir:

```
// DortIslemServerArayuz.java

import java.rmi.*;
public interface DortIslemServerArayuz extends Remote
{
        public int topla ( int x, int y ) throws RemoteException;
        public int cikar ( int x, int y ) throws RemoteException;
        public int carp  ( int x, int y ) throws RemoteException;
        public int bol   ( int x, int y ) throws RemoteException;
}
```

Şekil 3.4: Örnek Arayüz Tanımı

3.7.1.2 Sunucu Sınıfının Oluşturulması

Sunucu sınıf, uzak arayüzü gerçekleştirmeli (implements), constructor metodu tanımlamalıdır. Sunucu sınıfın ana (main) metodu içinde, uzak nesnenin bir örneği (instance) oluşturularak, RMI registry içine konulmalıdır.

Örnek uygulamamızda bulunan sunucu sınıfı kodu şu şekildedir:

```
// DortIslemServer.java

import java.util.*;
import java.rmi.registry.*;
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.net.URL;

public class DortIslemServer extends UnicastRemoteObject implements
DortIslemServerArayuz
{
private int sonuc;
public DortIslemServer () throws RemoteException {
        super( ) ;
        sonuc = 0;
 }
```

Constructor

Şekil 3.5: "DortIslemServer" Sunucu Sınıfı Tanımı

```
    public int topla ( int a , int b)
    {
            sonuc = a + b ;
            return ( sonuc ) ;
    }
    public int cikar ( int a , int b )
    {
            sonuc = a - b ;
            return ( sonuc );
    }
    public int carp ( int a, int b )
    {
             sonuc = a * b ;
             return ( sonuc );
    }
    public int bol ( int a , int b )
    {
             sonuc = a / b ;
             return ( sonuc );
    }

    public static void main(String args[])
    {
    try
     {
            DortIslemServer  s  =  new DortIslemServer ( ) ;
            Naming.rebind("server" , s);
            System.out.println(" Server Hazır ! ");
     }
    catch(Exception e)
    {
            System.out.println(" Server Error : "+e.getMessage());  }
     }
} // DortIslemServer.java sonu
```

Şekil 3.6: "DortIslemServer" Sunucu Sınıfı Tanımı (Devamı)

3.7.1.3 İstemci Sınıfının Oluşturulması

Sunucudan dört işlem servisini alacak olan örnek bir istemci programı örnek uygulamada şu şekilde düzenlenmiştir:

```
//İstemci.java
import java.rmi.*;
import java.net.*;
import java.net.URL;
import java.io.*;
import java.util.*;
import java.rmi.server.*;

public class İstemci
{
  public static void main (String args[])
   {
   try
   {
    DortIslemServerArayuz cl_object = (DortIslemServerArayuz)
Naming.lookup ("rmi://127.0.0.1/server");

    int alinan_sonuc ;

    alinan_sonuc = cl_object.topla ( 10 , 20 );
    System.out.println("Toplam sonucu :" + alinan_sonuc );

    alinan_sonuc = cl_object.cikar ( 20,10 );
    System.out.println("Cıkarma sonucu :" + alinan_sonuc );

    alinan_sonuc = cl_object.carp ( 5 , 10 );
    System.out.println("Carpma sonucu :" + alinan_sonuc );

    alinan_sonuc = cl_object.bol ( 100 , 10 );
    System.out.println("Bölme sonucu :" + alinan_sonuc );

   }
   catch(Throwable e)
   {
    System.out.println("Client Exception : " + e);
   }

  } // main sonu

} // İstemci sonu
```

Şekil 3.7: "İstemci" Sınıfı Tanımı

3.7.2 Dosyaların Derlenmesi , Koçan / İskelet dosyalarının oluşması

Arayüz, sunucu ve istemci kodlar yazıldıktan sonra şu şekilde derlenmelidir:

< javac DortIslemServerArayuz.java
< javac DortIslemServer.java
< javac İstemci.java

Bu komutlardan sonra java dosyalarının ".class" uzantılı derlenmiş halleri oluşmaktadır.

"Stub" ve "Skeleton" kodların oluşturulması için RMI derleyicisi (rmic komutu) kullanılır. Sunucu kod rmic ile derlenerek "stub" ve "skeleton" kodlar oluşur.

< rmic DortIslemServer

3.7.3 Derlenen Sunucu ve İstemci Programlarının Çalıştırılması

- Rmiregistry adı verilen uygulama çalıştırılır. Bu uygulama arka planda (windows98 ortamında sunucu tarafta ayrı bir Dos penceresi içerisinde) çalıştırılabilir. Eğer, sunucu ve istemci kod içinde belli bir port belirtilmedi ise 1099 varsayılan port olarak kullanılır. Eğer, sunucu kodda başka bir port kullanılacak ise, bu port numarası rmiregistry çalıştırılırken belirtilmelidir.
  < rmiregistry
  ya da port numarası belirtilmek istenirse
  < rmiregistry 5000

- Daha sonra ilk önce sunucu program çalıştırılır.
  < java DortIslemServer

- Sunucu program çalıştırıldıktan sonra ise sunucudan servis almak isteyen istemci program çalıştırılır.
  < java Istemci

4 İLGİLİ ÇALIŞMALAR

Bu bölümde, mevcut bileşen modeller ve dağıtık sistemlerin dinamik konfigürasyonu ile ilgili daha önce gerçeklenmiş olan çalışmalar anlatılmaktadır.

4.1 Mevcut Bileşen Modelleri

Bileşen modelleri içerisinde sıkça kullanılan belli başlı dört model vardır: COM / DCOM [2], EJB [4], CORBA [1] bileşen modeli (CCM) ve RMI [3].

COM (Bileşen Nesne Modeli), nesnelerin birbirleriyle birden fazla arayüz aracılığıyla konuşabilmesini sağlayan Microsoft tarafından geliştirilen bir modeldir. DCOM ise COM uygulamalarının dağıtık sistemlerde birbirleriyle haberleşmesini sağlayan uygulama katmanında çalışan bir protokoldür.

EJB (Enterprise JavaBeans), dağıtık Java uygulamalarının birbirleriyle ilişki içinde çalışmasını sağlayan sunucu merkezli bir bileşen modelidir. CCM de, EJB' ye benzer şekilde CORBA bileşenlerinin bir arada çalışmasını sağlayan bir modeldir. Her iki model de bir kap içerisinde bulunur ve dış dünya ile ilişkisi hep bu kap üzerinden olur. Bu modellerin gerçeklenmesi ise uygulama geliştiriciler yerine kap geliştiriciler tarafından yapılır.

RMI ise Java'nın bileşenlerinin farklı makinelerden birbirleriyle haberleşmesini sağlayan bir sistemdir. RMI'da da, yukarıda isimleri geçen tüm mevcut bileşen modülleri gibi, dinamik konfigürasyona olanak sağlayan bir yapı bulunmamaktadır. Modellerin tümü bileşen sistemleri için temel bir yapı oluşturmakta ancak bu yapının dinamik olarak değişmesine ve dağıtık çalışmanın değişikliklere rağmen sürdürülebilmesini izin verecek desteği sağlamamaktadır.

4.2 Dinamik Sistemler

Dağıtık sistemlerin çalışma zamanında dinamik konfigürasyonu üzerinde çalışan Chen [5,13,14], Konfigürasyon Yönetimi Etmeni, Sanal Koçan ve başlangıçtaki yerel veya uzaktan erişim hakkındaki çalışmalarıyla istemci ve sunucu arasında saydam bir

yapı oluşturmayı amaçlamıştır. Fakat nesnelerin taşınması hakkında bir çalışma yapılmamıştır. Bu konuda karşılaşılacak senaryolar ortaya koymamış ve nesnenin taşınması sırasında izlenmesi gereken adımlar hakkında fikir bildirmemiştir. Bunlara ek olarak, nesnelerin yer değiştirmesinden sonra sistemin verimli çalışması hakkında da öngörüde bulunulmamıştır.

Marco Avvenuti ve Alessio Vecchio [15] ise, nesnelerin mobil RMI'ya uygun olarak hangi yapılarda oluşturulacağı, nesnelerin kontrolünün ne tür yapılarla (CM Ajanı gibi) sağlanacağı üzerinde durulmamış, fakat nesnelerin farklı çalışma uzayı arasında nasıl yer değiştirebileceği anlatılmıştır. Nesnelerin yer değiştirmeleri sırasında hangi adımların izleneceği, istemci – sunucu arasındaki oluşan yolun kısaltılması için yöntemler açıklanmıştır.

Geleneksel dağıtık sistemlerde, nesnelerin yer değiştirmesinden sonra sistem bütünlüğünün devamı iki temel yöntemle sağlanmaya çalışılmıştır[14]. Bunlardan birincisi uzak nesnelerin referanslarının daima güncel tutulmasıdır. Örneğin, Holder [16]'in çalışmasında, izleyici mekanizması kullanmaktadır. Bu mekanizmada, nesne bir yerden başka bir yere taşınacağı zaman eski yerinde bir iz bırakır. Bu iz sayesinde, nesnenin yeni yeri bulunabilir ve bir istemci hareket halindeki nesneye erişmek istediği zaman bir veya birden fazla izleyiciyi geçmek zorundadır. Bu izleyici yöntemi sistem performansı için maliyetlidir. Çünkü nesne her bulunduğu yerde bir izleyici yaratacaktır ve bu izleyiciler üzerinden yapılan çağrılar performansı olumsuz etkilemektedir. Sistem bütünlüğünü ve dinamizmi sağlamaya yönelik ikinci yöntem ise, her bir metot çağrımından önce sunucu nesnenin referansının elde edilmesi yöntemidir. Bu yapıda, örneğin CORBA'da dinamik arayüz çağrısı sisteminde de olduğu gibi, istemci bir sunucu metodu çağrısı yapmadan önce sunucunun referansını sorgular ve o referansı aldıktan sonra çağrıyı gerçekler. Fakat kolaylıkla görüleceği gibi bu sistem de performans açısından oldukça maliyetlidir çünkü her defasında sunucunun referansını sorgulamak zaman alacaktır.

5 HAREKETLİ NESNELER İÇİN DİNAMİK RMI SİSTEMİ

Bu çalışmanın amacı, hareketli (yer değiştirebilen) nesnelerin sağlıklı bir şekilde çalışmalarını sağlayacak dinamik bir RMI yapısının oluşturulmasıdır. Standart olarak kullanılan RMI'da, uzak nesnelerin metotlarına erişim, dinamik olmayan bir şekilde önceden oluşturulmuş yapı korunduğu sürece gerçekleştirilmektedir. Çalışma zamanında nesnelerin yer değiştirmeleri durumunda, çalışmada hata oluşur ve sonlanır. Örneğin, istemci nesnesi, sunucu nesnesinin metotlarını kullanırken sunucu nesnesinin hangi adreste olduğunu bilir ve sunucu nesnesinin bu adresten ayrılması durumunda daha fazla hizmet alamaz. Yeniden çalışır hale gelmesi için istemci nesnelerin ve sunucu nesnesinin yeniden başlatılmaları ve birbirleri arasında yeniden bağlantı kurmaları gerekmektedir.

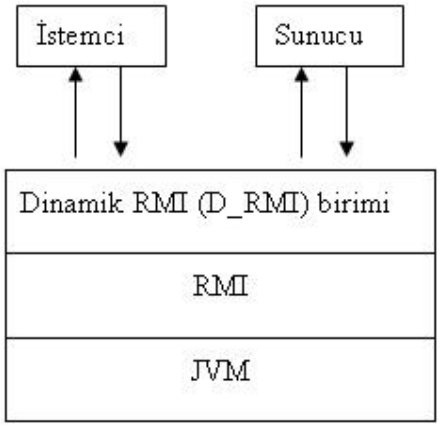
Hareketli nesneler için dinamik RMI sistemi ise, nesnelerin hareket etmelerine olanak sağlayacak bir yapı sağlamaktadır. Standart RMI'nın bu eksikliğini gidermektedir. Çalışma zamanında, istemci nesneler, sunucu nesne metotlarını sunucunun yerinden bağımsız olarak kullanmaktadırlar. Bu dinamik yapı sayesinde, sunucu nesneler yer değiştirebilirler, bu şekilde herhangi bir yer değiştirme, istemci nesnelerinin sunucudan hizmet almalarını engellemeyecektir. Çalışan sistem, dinamik olarak kendisini yenileyip çalışmayı devam ettirecektir.

Hareketli nesneler için dinamik RMI sistemi temel olarak 5 birimden oluşmaktadır:

- Dinamik RMI Birimi
- Sanal Koçan
- Nesne Yöneticisi
- İstemci Birimi
- Sunucu Birimi

5.1 Dinamik RMI Birimi

Yürütme sırasında yeniden konfigüre edilebilen dağıtılmış nesne sistemlerinde nesnelerin yaratılmalarından, yer değiştirmelerinden ve yer değiştirme sırasında sistemin bütünlüğünün korunmasından sorumlu olan yetkili ve ayrıcalıklı bir yapıya gerek duyulur. Bu çalışmada, söz konusu hizmetleri yerine getirmek üzere Dinamik RMI (D_RMI) Birimi olarak adlandırılan bir ara katman yazılımı geliştirilmiştir. Uygulama yazılımı ile RMI arasında yer alan bu birim (Şekil 5.1) D_RMI özelliğini destekleyecek olan altyapıyı oluşturan ve denetleyen temel unsurdur. Üzerinde çalışılan dağıtılmış nesne yapısının dinamik olarak yeniden düzenlenmesi, yeni bir nesnenin eklenmesi, var olan bir nesnenin silinmesi, bir nesnenin yer değiştirmesi, vb. gibi durumlarda etkinleşen bu birimin ana görevi, gerekli işlem ve güncelleme adımlarını yürüterek sistemin çalışmasını tutarlı ve güvenli bir şekilde devam ettirmektir.



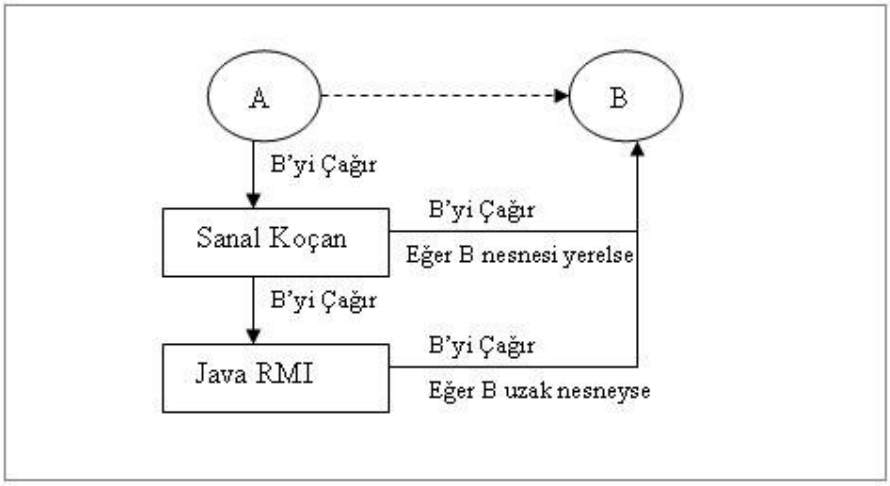
Şekil 5.1: Dinamik RMI Ara Katmanı

5.2 Sanal Koçan

Günümüzde dağıtılmış sistemler bileşenleri arasında iletişim mekanizması olarak genelde RPC veya RMI kullanılmaktadırlar. Bu yapılarda istemci nesnesi, sunucu nesnesinin referansını elde eder ve doğrudan erişim gerçekleştirir. Fakat sunucu nesne yer değiştirdiği zaman, istemci tarafında sahip olunan sunucu referansı artık geçersiz olmakta ve istemci bu referans ile sunucuya erişmek istediği zaman hata (aykırı durum) oluşmaktadır. Böyle bir durumda, istemcinin aykırı durumu önlemek

üzere, gerektiğinde sunucunun referansını güncellemesi veya çağrıyı yinelemesi gerekmektedir. Bu tür işlemler istemci tarafı için fazladan ek yük oluşturur. Ayrıca, bu tip bir çözüm, uygulamanın saydam olmasına engel olur. Buna karşılık, sistemden sorumlu başka bir mekanizma referans güncelleme işlemini üstlenecek olursa, saydam bir uygulama sağlanmış olunur [14]. Ayrıca, başarılı bir dinamik konfigürasyon için hedef bileşenler ve diğer bileşenler arasındaki ilişkiler sağlanmalı, gözlenmeli ve kontrol edilmelidir. Bileşenler arasındaki bu işlemleri yapacak olan yapı ise Java RMI'nin üzerinde gerçeklenen bir ara katman yazılımı ile sağlanabilir. Bu yapıda bileşenler (uzak nesneler) RMI'da olduğu gibi standart koçanlarla haberleşmeyip, Sanal Koçan denilen özel nesneler üzerinden haberleşirler. Bir Sanal Koçanın temel yapısı içerisinde sunucu nesnesine (yerel veya uzak) bir işaretçi tutulur. Sanal Koçan aynı zamanda işaretçisini tuttuğu sunucu nesnesinin tüm metotlarına da sahiptir. Böylece, herhangi bir istemci, erişmek istediği sunucunun metotlarına Sanal Koçan üzerinden erişebilir [13].

Şekil 5.2'deki örnekte, A nesnesi B nesnesine erişip bir metodunu çağırmak istiyor. Bu erişim isteğini Sanal Koçan üzerinden yapıyor. Sanal Koçan, B nesnesine yerel veya uzak nesne olmasına göre iki farklı yoldan erişiyor. Eğer, B nesnesi A nesnesi ile aynı makinede bulunuyorsa, yani B nesnesi yerel nesne ise doğrudan B nesnesine erişilir. Eğer, B nesnesi uzak nesneyse Sanal Koçan, Java RMI'yı kullanarak B nesnesine çağrı yapmaktadır.



Şekil 5.2: Sanal Koçan Üzerinden Sunucu Metodu Çağırma

Sanal Koçanın bir diğer özelliği de, temel fonksiyonları dışında, istenilen şekilde genişletilebilmesidir. Uygulamaya özgü bir dizi ek işlem sunucunun metotlarını çağırmadan önce veya metot çağrıları tamamlandıktan sonra gerçekleştirilebilir. Örneğin, metot çağrılarının gerçekleştiği zaman, gerçekleşme süresi, istemci ve sunucu isimleri gibi bilgilerin kayıt altına alınması isteniyorsa, metodu çağırmadan önce veya çağırdıktan sonra istenilen kayıt tutma işlemleri gerçekleştirilebilir. Bir diğer örnek olarak da metoda ait istatistiksel bilgiler tutulmak isteniyorsa, Sanal Koçandaki sunucu metot çağrıları sırasında gerekli eklemeler ve düzenlemeler yapılabilir.

Sanal Koçan kullanımı klasik RMI koçanının kullanımı ile aynıdır. Java RMI koçanı tarafından gerçeklenen arayüz (interface) aynı şekilde Sanal Koçan içinde de gerçeklenmektedir. Şekil 5.3 'deki program parçasında, Sanal Koçanın kod içerisinde nasıl kullanılacağı gösterilmiştir.

```
public void Test ( )
{
    String strBilgi ;
    TestArayuzu objSanalKocan =
(TestArayuzu)objMakineYoneticisi.SanalKocanAl("Sunucu1") ;

    Try
    {
        strBilgi = objSanalKocan.testMetot("Bu bir test cagrisidir.") ;
        System.out.println(strBilgi) ;
    }
    Catch ( Exception e )
    {
        e.printStackTrace( ) ;
    }
}
```

Şekil 5.3: Sanal Koçan Kullanma İçin Örnek Kod Parçası

Sonuç olarak Sanal Koçanın kullanımının getireceği yararları şu şekilde özetleyebiliriz :

a. Sanal Koçan, Nesne Yöneticisi tarafından dinamik olarak yüklenir ve kontrol edilir.

b. Sanal Koçan, sunucu çağrılarını otomatik olarak izleyebilir ve nesnelerin yer değiştirebilir duruma geçip geçmediğini kontrol edebilir.

c. Dinamik konfigürasyonda önemli bir konu olan nesneler arasındaki etkileşim, Sanal Koçanlar ile belirlenebilmektedir [14].

5.3 Nesne Yöneticisi

Nesne Yöneticisi, D_RMI Birimi işlemlerini yürütmekten sorumludur. Her bir yürütme ortamında (runtime environment) yönetimden sorumlu olan bir adet Nesne Yöneticisi yer alır. Çalışma ortamında bulunan ve Dinamik RMI yapısının o makine üzerinde düzgün ve güvenli bir şekilde çalışmasını sağlayan Nesne Yöneticisi, yeni nesnelerin çalışma ortamına (runtime environment'a) kaydedilmesinden, o makine üzerinde var olan nesnelerin çalıştırılıp, durdurulması ve yer değiştirmelerinden sorumludur.

Nesne Yöneticisi bu görevleri yerine getirirken iki adet listeden yararlanır. Bu listeler Yerel Nesne Listesi ve Sanal Koçan Listesidir. Bu listelerin Dinamik RMI sistemindeki fonksiyonları daha sonraki bölümlerde anlatılacaktır.

Nesne Yöneticisi çalışmaya başladığı zaman kendisini RMI Registry ile RMI İsimlendirme Servisi (Naming Service) üzerine kaydeder ve çalışır durumda beklemeye geçer. Farklı makineler üzerinde çalışan Nesne Yöneticileri birbirleri ile RMI üzerinden haberleşirler. Bu haberleşme ortamı nesnelerin yer değiştirmesi sırasında Nesne Yöneticilerinin birbirlerine denetim bilgilerini aktarmalarına olanak sağlar.

Herhangi bir nesne, bir makine üzerinde yerel olarak yaratılıp çalışmaya başladığında, öncelikle kendisini o makine üzerinde bulunan Nesne Yöneticisine

kaydeder. Kaydedilen her nesne için, Nesne Yöneticisinde bulunan Yerel Nesne Listesinde bir kayıt oluşturulur.

Sanal Koçanların yaratılıp, istemcilerin kullanımı için hazırlanması işlemi de Nesne Yöneticisi tarafından gerçekleştirilmektedir. Bir istemci, bir sunucuya ait olan hizmetleri (metotları) kullanmak istediği zaman, Nesne Yöneticisi ile iletişim kurar ve kendisine sunucuya ait olan bir Sanal Koçan döndürmesini bekler. Nesne Yöneticisi ise istenilen sunucuya ait olan Sanal Koçanı hazırlar ve istemciye Sanal Koçanın referansını gönderir. Aynı zamanda da, yaratılan bu Sanal Koçan nesnesi, Nesne Yöneticisi üzerinde bulunan Sanal Koçan Listesine kaydedilir. Bu liste, Sanal Koçanların güncellenmesinde ve dinamik konfigürasyon yenilenmesi işlemlerinde, Sanal Koçan nesnelerine ulaşmakta kullanılmaktadır.

5.3.1 Nesne Yöneticisi tarafından yürütülen temel işlemler

5.3.1.1 SunucununKocaniniAl

Bu metot istemci tarafından kullanılır. İstemci erişmek istediği bir sunucunun Sanal Koçanını bu metodu kullanarak elde eder. Parametre olarak erişilmek istenen sunucu nesnenin ismini alır. İşlemlerden geriye erişilmek istenen sunucunun referansını taşıyan Sanal Koçan nesnesi döndürülür.

```
public SanalKocanArayuzu SunucununKocaniniAl( String strSunucuAdi )
{
}
```

Yapılan İşler :

İlk önce, erişilmek istenilen sunucu nesnenin yerel veya uzak nesne olduğu belirlemek üzere, SunucuNesnesiYerelmi( ) metodu kullanılır. Bu metot, Nesne Yöneticisi üzerindeki nesnelerin bulunduğu Yerel Nesne Listesini tarayarak, erişilmek istenilen sunucu nesnenin aynı makine üzerinde bulunup bulunmadığını kontrol eder. Eğer, sunucu nesne Yerel Nesne Listesinde yer alıyorsa, yani yerel ortamda bulunuyorsa, sunucunun yerel referansı kullanılarak bir Sanal Koçan oluşturulur ve oluşturulan bu Sanal Koçan nesnesinin referansı istemciye gönderilir. Eğer sunucu nesnesi, Yerel Nesne Listesinde bulunmuyorsa, Java.RMI İsimlendirme Servisi (Naming Service) kullanılarak uzak nesne olarak aranılır. Elde edilen uzak

nesne referansını içeren bir Sanal Koçan oluşturulur ve oluşturulan bu Sanal Koçan istemciye gönderilir. Böylece, istemci artık sunucunun herhangi bir metodunu Sanal Koçan üzerinden çağırabilir.

Bu iki kontroller (Yerel Nesne Listesi ve RMI İsimlendirme Servisi) sonunda da sunucu nesnesine erişilemezse, istemciye "sıfır" (NULL) değeri döndürülür ve bu durum bir uyarı olarak kaydedilir.

Bu arada, elde edilen Sanal Koçan referansı ilerideki olası Sanal Koçan referans güncellemelerinde kullanılmak üzere bir listeye (Sanal Koçan Listesi) kaydedilir.

5.3.1.2 Nesnelerin yer değiştirmesi

Nesnelerin yer değiştirmesi, bir makine üzerinde bulunan bir nesnenin başka bir makineye taşınması işlemidir. Bu işleme nesnelerin göç etmesi adı da verilmektedir (object migration).

Nesnenin yer değiştirebilmesi için, taşınacak olan nesnenin üzerinde kayıtlı olduğu Nesne Yöneticisinin SunucuNesnesiniTasi ( ) metodu çağrılır. Bu metoda parametre olarak hedef makinenin (Nesne Yöneticisinin) adresi (URLsi) verilmektedir.

```
public void SunucuNesnesiniTasi ( String GidilecekMakineAdresi )
{
}
```

Örneğin, bir nesne Nesne Yöneticisi-1'in üzerinde yer aldığı makineden Nesne Yöneticisi-2'nin üzerinde yer aldığı makineye taşınacaksa:

```
String strHedefMakineAdresi = "rmi://169.45.6.1/NesneYoneticisi2"
objNesneYoneticisi1.SunucuNesnesiniTasi(strHedefMakineAdresi) ;
```

şeklinde bir metot çağrısı yapılır.

- void SunucuNesnesiniTasi (String GidilecekMakineAdresi)

"GidilecekMakineAdresi" parametresi kullanılarak Java RMI İsimlendirme Servisi üzerinden hedef makinenin (Nesne Yöneticisi nesnesinin) referansı alınır.

Bir sonraki adımda, hedef makinenin referansını elde eden kaynak Nesne Yöneticisi, hedef Nesne Yöneticisi üzerindeki YeniNesneYarat() metodunu çağırır.

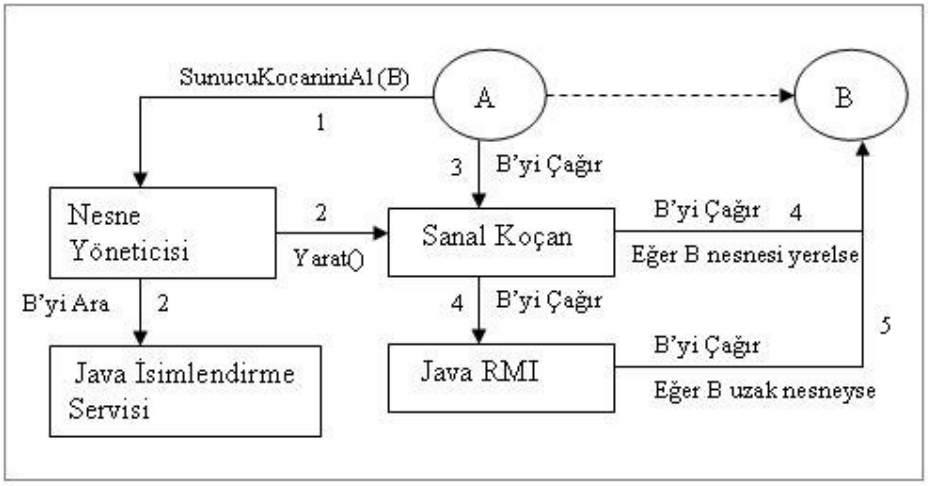
```
SanalKocanArayuzu YeniNesneYarat ( String AnaMakineAdresi )
{
}
```

Bu metot, taşınacak olan nesnenin, hedef makine üzerinde yaratılmasını sağlar.

YeniNesneYarat() metoduyla yeni yaratılan nesnenin referansı kaynak Nesne Yöneticisine dönüş parametresi olarak gönderilir. Bu referans bilgisi, yer değiştiren nesnenin hali hazırda kullanılan Sanal Koçanlarındaki referanslarının güncellenmesinde kullanılacaktır.

Yer değiştiren nesnenin yeni referansı, ReferansGuncelle( ) metodu kullanılarak, kaynak Nesne Yöneticisi üzerindeki Sanal Koçan tarafından tutulan sunucu nesnenin referansı olarak güncellenir. Böylece, Sanal Koçan üzerinden sunucuyu kullanan istemcinin istekleri sorunsuz şekilde karşılanmaya devam edilir. Sunucunun yeri ve yeni yerinin referansı taşınma sonucu değişmiştir, fakat bu değişiklik istemciye hissettirilmeden Sanal Koçan içerisinde tutulan sunucu referansının güncellenmesiyle sonuçlanmıştır. Sözü edilen tüm işlemler istemciye tamamen saydam bir şekilde gerçekleştirilir.

Bir nesnesinin (A istemci) Makine Yöneticisi yapısını kullanarak Dinamik RMI sistemini üzerinden başka bir nesnenin (B sunucusu) metodunu çağırması için yürütülen adımlar Şekil 5.4'de gösterilmiştir.



Şekil 5.4: Nesne Yönetici ile Sanal Koçan Üzerinden Nesne Metodu Çağırma

5.4 İstemci Uygulaması

İstemci uygulaması çalışmaya başladığı zaman, ilk önce üzerinde çalışacağı makineden sorumlu olan Nesne Yöneticisi nesnesinin referansını JAVA RMI ile alır. Daha sonra, metotlarını kullanmak istediği sunucunun referansını Nesne Yöneticisi üzerindeki SunucununKocaniniAl() metoduyla alır. SunucununKocaniniAl metodu, istemciye içerisinde sunucu nesneye ait referansın yer aldığı bir Sanal Koçan döndürür.

İstemci, Sanal Koçan nesnesini doğru bir şekilde alamaz ise, tekrar dener. İkinci kez de düzgün bir biçimde elde edemez ise çalışmasını sonlandırır.

Eğer, istemci Sanal Koçan nesnesini sorunsuz bir şekilde elde ederse, bu nesneyi kullanarak sunucunun tüm metotlarını çağırabilir.

Adım adım özetleyecek olursak , istemci:

- Üzerinde çalışacağı makineden sorumlu olan Nesne Yöneticisi referansını tutacak olan bir Nesne Yöneticisi nesnesi yaratır.
- Nesne Yöneticisinin referansını java.rmi.Naming.lookup ile elde eder.
- Nesne Yöneticisi nesnesinin referansını kullanarak, erişmek istediği sunucuya ait Sanal Koçan nesnesini elde eder.

- Edindiği Sanal Koçan nesnesi kullanarak istenilen sunucu metodu çağırılabilir (Örnek uygulamada sunucu üzerinde bulunan Dört İşlem() metodu çağrılıyor).

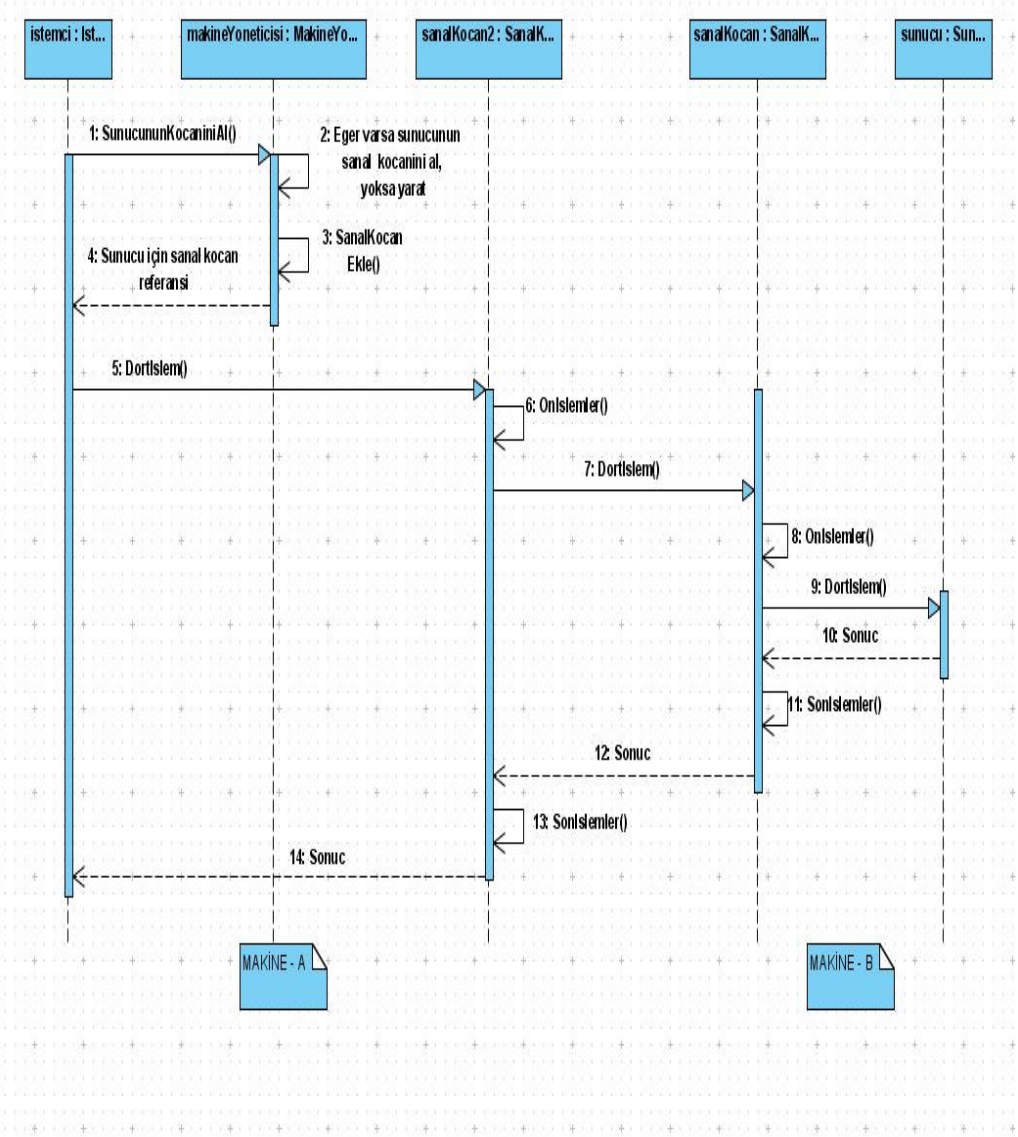
5.5 Sunucu Uygulaması

Sunucu uygulaması da çalışmaya başladıktan sonra ilk olarak üzerinde bulunduğu Nesne Yöneticisi nesnesinin referansını JAVA RMI İsimlendirme Servisi aracılığıyla alır. Daha sonra, Nesne Yöneticisi nesnesinde bulunan NesneKaydet() metodunu kullanarak, kendisini Nesne Yöneticisi üzerine kaydettirir. Bu kaydetme işlemi sırasında, sunucu nesnenin ismi ve referansı, Nesne Yöneticisi üzerinde bulunan Yerel Nesne Listesine eklenir. İstemci nesnesi Nesne Yöneticisi nesnesi üzerinden sunucunun referansını istediği zaman, bu liste kullanılarak sunucu nesnenin yerel bir nesne mi, yoksa uzak bir nesne mi olduğu kontrol edilir. Sunucu nesne, kendisini Nesne Yöneticisi üzerine kaydettikten sonra, herhangi bir nesne tarafından çağırılmaya hazır bir şekilde beklemeye geçer.

Adım adım özetleyecek olursak;

- Sunucu nesnesi yaratılır.
- Yaratılan sunucu nesne Java İsimlendirme Servisine kaydedilir. Yerel istemciler için bu işlemin bir önemi olmayacaktır, çünkü yerel istemciler sunucuya yerel referansı ile bağlanacaklardır. Fakat başka makinelerden sunucuya erişmek isteyen istemciler, RMI üzerinden bu sunucuya erişeceklerdir.
- Üzerinde çalışacağı makineden sorumlu Nesne Yöneticisi referansını tutacak olan Nesne Yöneticisi nesnesi yaratılır.
- Sunucu Nesne Yöneticisi nesnesinin referansını java.rmi.Naming.lookup ile alır.
- Sunucu nesne, kendisini, isim ve referansını vererek üzerinde çalışacağı makineye kaydettirir. Böylece, zamanla makine üzerinde çalışan nesnelerin isimlerinin ve referanslarının tutulduğu bir liste (Yerel Nesne Listesi) oluşturulur.

- Sunucu nesne, hazır halde, istemcilerden gelecek istekler için beklemeye geçer.



Şekil 5.5 İstemci Sunucu İletişimi Sıra Diyagramı

5.6 Nesnelerinin Yer Değiştirmeleri

Bir sunucu nesnesinin yer değiştirme şartlarının kontrolünden ve gözlenmesinden, yer değiştirme işleminin hangi makineye doğru olacağının belirlenmesinden ve yer değiştirme işleminin başlatılmasından, o sunucu nesnesi ile aynı makine üzerinde bulunan ve sunucuya ait olan Sanal Koçan sorumludur.

Sanal Koçan, ilgili sunucu nesnesinin yer değiştirmesine, yani yeni bir makineye taşınmasına karar verdikten sonra sunucu nesne üzerinde bulunan Taşın() metodunu çağırır.

Tasin metodunun prototipi:

```
public void Tasin ( String GidilecekMakineAdresi )
{
}
```

Tasin metodunun prototipinden de görüleceği gibi bu metot parametre olarak taşınılacak olan hedef makinenin adresini (String GidilecekMakineAdresi) almaktadır. Sunucu nesnesi gidilecek makinenin adresini aldıktan sonra kendi üzerinde bulunduğu makineden sorumlu olan Nesne Yöneticisinin SunucuNesnesiniTasi() metodunu çağırır.

SunucuNesnesiniTasi metodunun prototipi:

```
void SunucuNesnesiniTasi ( String GidilecekMakineAdresi )
```

Nesne Yöneticisi nesnesinin SunucuNesnesiniTasi metodu çağırılmasıyla birlikte "GidilecekMakineAdresi" kullanılarak, RMI İsimlendirme Kaydı (Naming Registry) servisi üzerinden hedef makineden sorumlu olan Nesne Yöneticisi nesnesinin uzak referansı elde edilmeye çalışılır. RMI İsim Kayıtları Servisinden, ilgili Nesne Yöneticisi uzak nesnesinin referansı elde edilemezse bu durum kayıt altına alınır ve taşınma işlemi durdurulur. Taşınılacak olan hedef makinenin Nesne Yöneticisi nesnesinin uzak referansı elde edilirse, taşınma işlemine devam edilir ve bu doğrultuda Nesne Yöneticisi uzak nesnesinin YeniNesneYarat() metodu çağrılır.

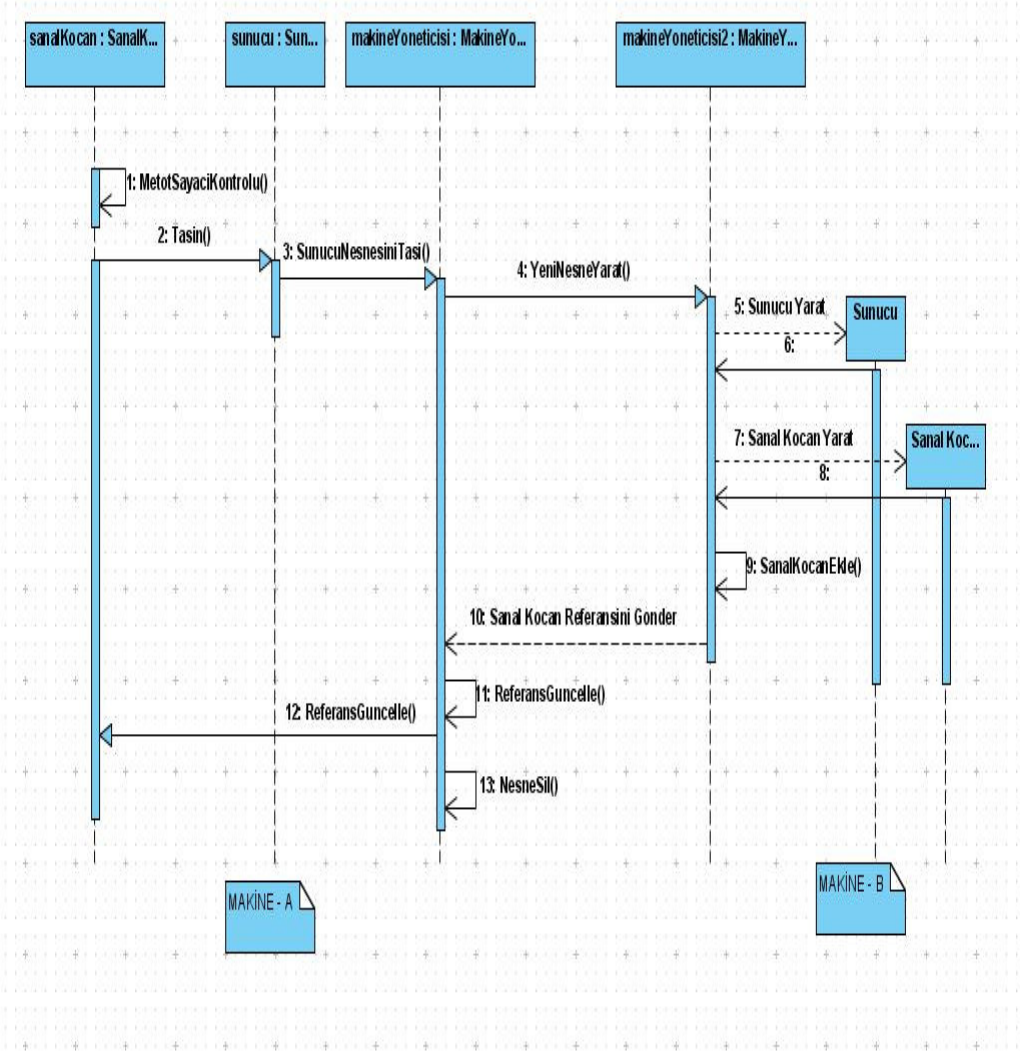
YeniNesneYarat metodunun prototipi:

SanalKoçan YeniNesneYarat (String AnaMakineAdresi)

YeniNesneYarat metodu, yeni bir sunucu nesnesi yaratır. Yeni sunucu nesnesini yarattıktan sonra, yarattığı nesneyi üzerinde bulunduğu makinenin Nesne Yöneticisinde bulunan NesneKaydet() metodunu kullanarak, makine üzerine kaydettirir. Bu kaydetme işlemi sırasında, sunucu nesnenin ismi ve referansı, Nesne Yöneticisi üzerinde bulunan Yerel Nesne Listesine eklenir. Daha sonra, ilgili sunucu nesnesine erişim için daha önceden yaratılmış olan kayıtlı bir Sanal Koçan nesnesi olup olmadığı kontrol edilir. Eğer, daha önceden yaratılmış bir Sanal Koçan nesnesi varsa, bu Sanal Koçan içerisinde bulunan ve sunucuya erişimi sağlayan referans işaretçisi güncellenir. Eğer önceden yaratılmış bir Sanal Koçan bulunmuyorsa, yaratılan sunucu nesnesinin referansı parametre olarak verilerek yeni bir Sanal Koçan nesnesi yaratılır. Bu yaratılan nesne, makine üzerindeki Sanal Koçan Listesine kaydedilir ve böylece YeniNesneYarat metodu işlemlerini tamamlamış olur. Bu metot geriye, içerisinde yeni oluşturulan sunucu nesnesinin referansını taşıyan Sanal Koçan nesnesinin işaretçisini döndürür.

Sunucu nesnesinin yer değiştirme işleminden önceki durumda üzerinde yer aldığı makinenin Nesne Yöneticisi, sunucuyu başarıyla hedef makineye taşıyıp göç işlemini tamamlamış sunucunun Sanal Koçanını elde ettikten sonra, kendi üzerinde bulunan sunucuya ait tüm Sanal Koçanların sunucuya erişim için kullandıkları işaretçilerini günceller. Artık eski makine üzerinde bulunan Sanal Koçanlar sunucuya eskiden olduğu gibi yerel referans ile erişemeyeceklerdir. Bunun yerine, yeni taşındığı makine üzerinde bulunan Sanal Koçan üzerinden sunucuya erişimi gerçekleştireceklerdir.

Taşınma işlemi yeni makine üzerinde tamamlandıktan sonra, eski makine üzerinde bulunan Yerel Nesne listesinden sunucunun kaydı silinir. Çünkü artık sunucu yeni bir makineye taşınmıştır ve eski makine üzerinde bulunmamaktadır.



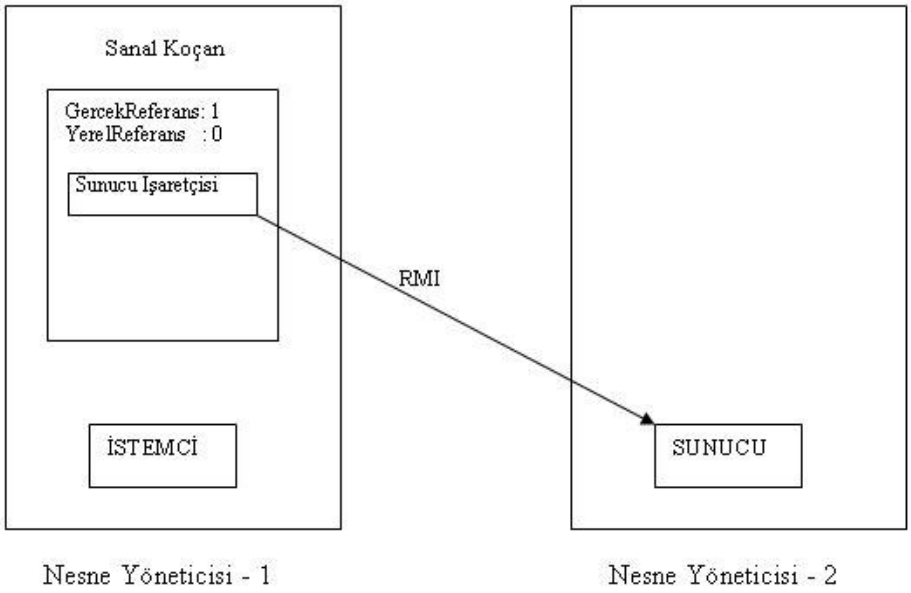
Şekil 5.6 Nesne Yer Değiştirme İşlemi Sıra Diyagramı

5.7 İstemci – Sunucu Arasındaki Yolun Kısaltılması

Bir istemci nesnesi sunucu nesnesinin herhangi bir metodunu kullanmak istediği zaman, üzerinde bulunduğu makinede yer alan Nesne Yöneticisi nesnesinin SunucununKocaniniAl( ) metodunu çağırıp sunucuya erişimi sağlayacak Sanal Koçanı elde etmekteydi. SunucununKocaniniAl metodu eğer sunucu nesnesi istemci nesnesiyle aynı makine üzerindeyse içerisinde sunucunun yerel referansının bulunduğu Sanal Koçanı istemciye göndermekteydi. Fakat istemci ile sunucu farklı

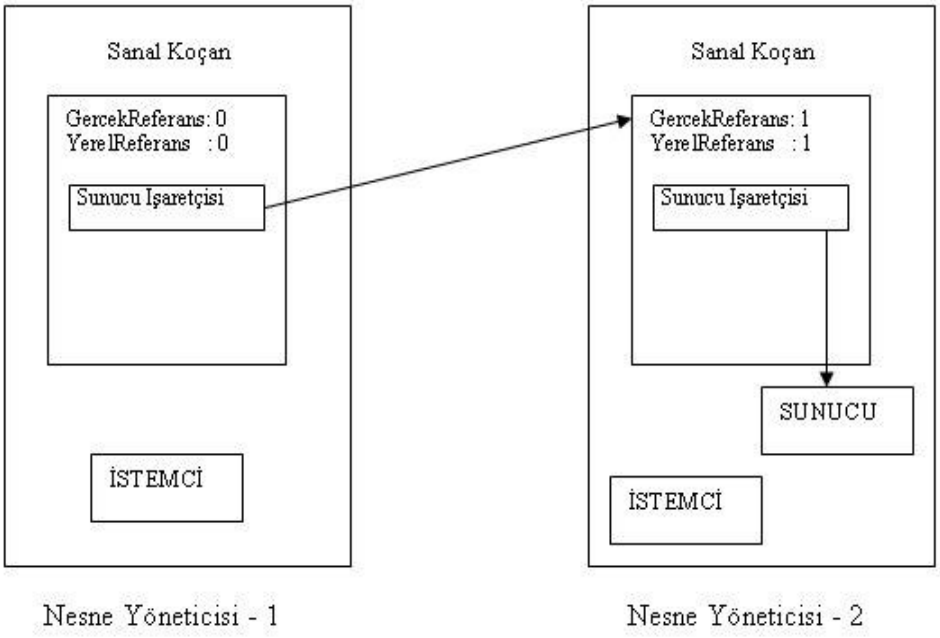
makineler üzerinde bulunuyorlarsa, istemci sunucuya erişimi başka makinelerde bulunan Sanal Koçanlar üzerinden gerçekleşmekteydi.

Örneğin, istemci ve sunucu farklı makinelerde yer alıyor ve sunucu üzerinde daha önceden yaratılmış bir Sanal Koçan nesnesi bulunmuyor ise, istemcinin kullandığı Sanal Koçan, Şekil 5.7'deki gibi sunucuya doğrudan RMI uzak referansı ile erişmektedir.



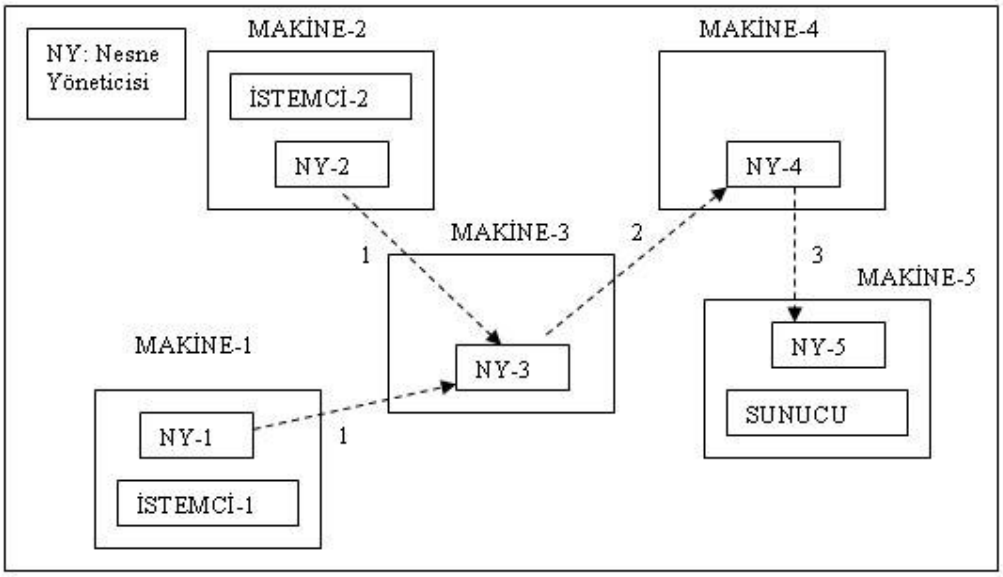
Şekil 5.7 Sunucuya RMI uzak nesne referansıyla erişme

Fakat istemci ve sunucu farklı makinelerde yer alıyor ve sunucunun bulunduğu makinede daha önceden sunucuya erişimde kullanılan Sanal Koçan bulunuyor ise, istemcinin kullandığı Sanal Koçan, sunucunun makinesinde bulunan Sanal Koçan nesnesine işaret etmektedir ve o nesne üzerinden sunucu metotlarına erişmektedir (Şekil 5.8).



Şekil 5.8: Sunucuya Sanal Koçan Üzerinden Erişim

İstemci nesnesi, sunucu nesnesine ilk erişiminde doğrudan sunucu nesnesinin bulunduğu makineyle iletişim halindedir ve arada başka bir makine bulunmamaktadır (İdeal durumdadır) . Fakat sunucu nesnesinin yürütme zamanında yer değiştirmesiyle, yani farklı makinelere taşınmasıyla birlikte istemci nesnesinin sunucu nesnesine erişimi doğrudan tek bir atlamayla (ideal durum) değil de, birden çok makine üzerinden atlamalar yaparak gerçekleşecektir (ideal olmayan durum). Sonuçta, istemcinin sunucuya erişiminin mümkün olabilmesi için birden fazla nesne ve makine üzerinden geçişlerin yapılması gerekecektir (Şekil 5.9). Sunucuya ulaşmak için izlenmesi gereken bu yolun uzaması çağrı zamanını da önemli ölçüde uzatacaktır. Bu nedenle, uzak çağrılarda olası bu tür dallanmaları önleyecek bir çözüm gereklidir.



Şekil 5.9: İstemci-Sunucu Arasındaki Oluşabilecek Atlamalı Yol

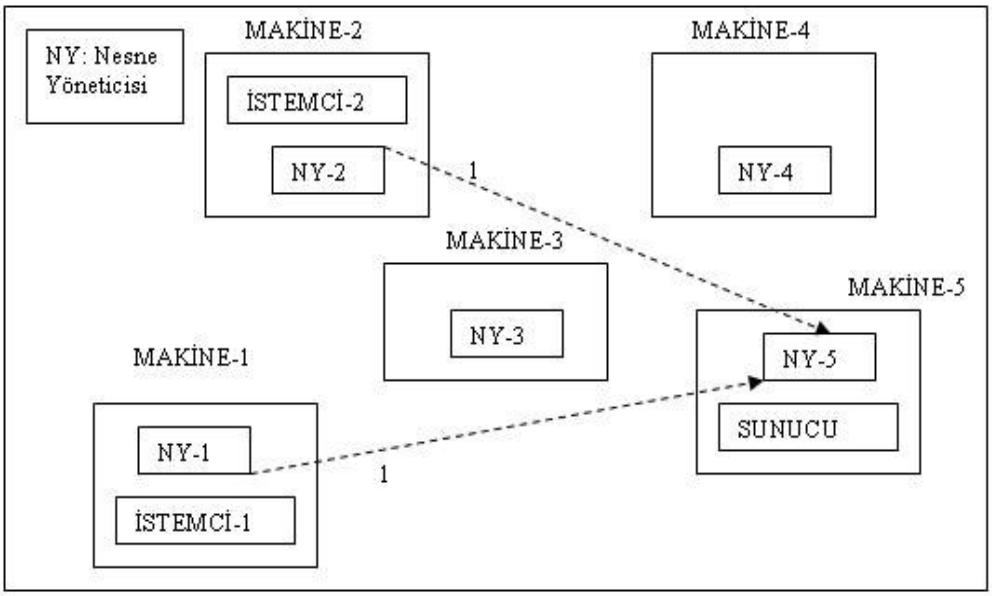
Yapılan çalışmada, bu tür dallanma ve atlamaların önlenmesi ve izlenmesi gereken yolun kısaltılması amacıyla, Sanal Koçanlar içerisine yeni bir metot eklenmiştir. Geliştirilen "SunucuyaDogrudanErisenVSReferansiAl" metodu sunucu çağrıları tamamlandıktan sonra, sunucuya ulaşma rotasının güncellenmesi işlemlerini yerine getirir.

SunucuyaDogrudanErisenVSReferansiAl metot prototipi:

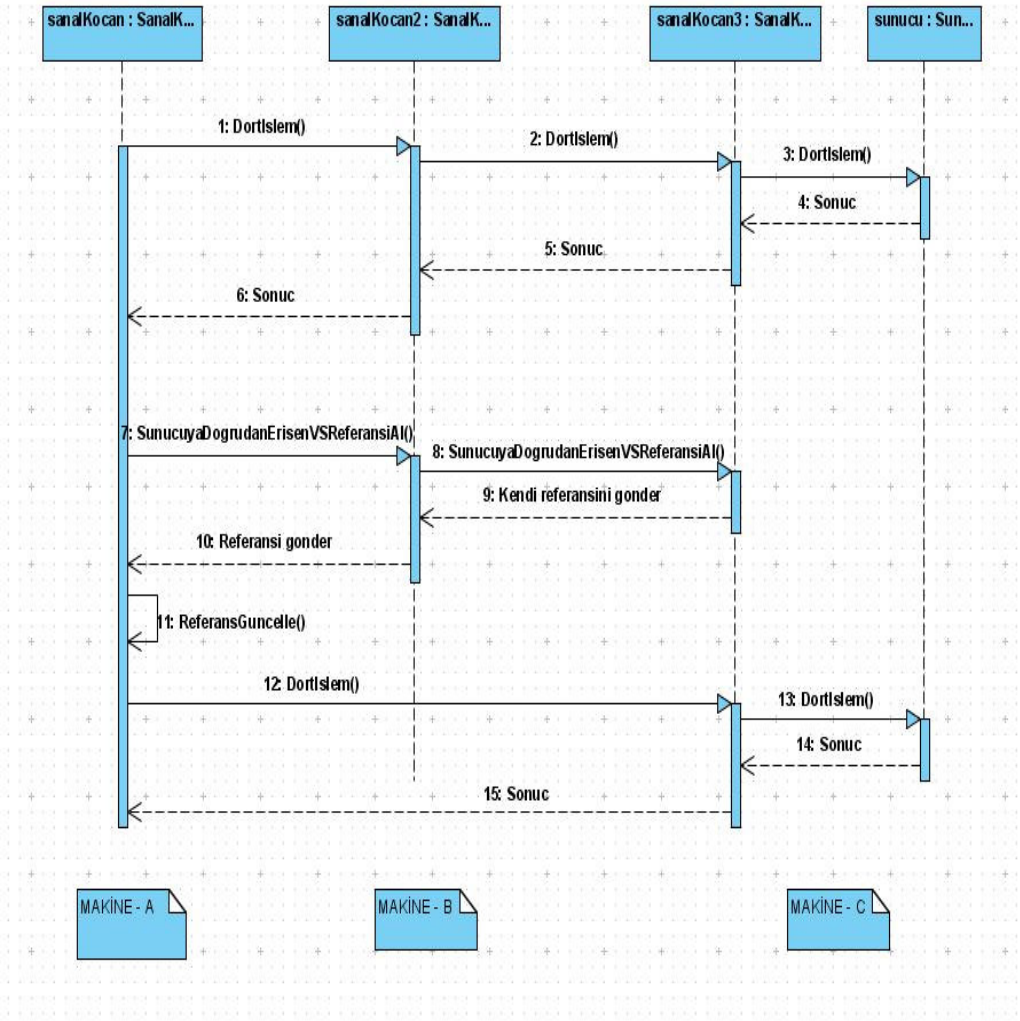
```
public SanalKocanArayuzu SunucuyaDogrudanErisenVSReferansiAl()
{
}
```

Bu metot, kendisini sunucuya ulaştıracak olan yolu izleyerek, arada yer alan Sanal Koçanları geçip, sunucuya doğrudan erişime izin verecek Sanal Koçan nesnesinin referansını elde eder ve bu referansı geri taşıyacak şekilde geldiği yolu izleyerek geri döner. Geri dönüş sırasında, sunucu nesnenin referansını elde eden ara makinelerdeki Sanal Koçanlar da kendi işaretçilerini güncellerler. Böylece, bu işlem tamamlandıktan sonra, tüm Sanal Koçanlar sunucuya bir atlama ile erişmelerini sağlayacak referansa sahip olurlar ve hiçbir aktarma veya atlama yapmadan, doğrudan sunucu nesnesine erişebilirler. Referans güncellenmesi işlemlerinden sonra

istemci nesneler ve sunucu arasındaki yeni rota aşağıdaki Şekil 5.10'da görüldüğü gibi olacaktır.



Şekil 5.10: Yol Kısaltma Algoritmasından Sonraki İstemci-Sunucu Arasındaki Yol



Şekil 5.11: İstemci – Sunucu Arasındaki Yolun Kısaltılması Sıra Diyagramı

5.8 Dinamik RMI Sisteminin Gerçeklenmesi

D_RMI gerçekleyen ara katman yazılımı 3 ana birimden oluşur: SanalKoçan, Nesne Yöneticisi ve Sunucu. Her birini oluşturan modüller ve gerçekleme ayrıntıları aşağıda sırasıyla anlatılmıştır.

5.8.1 SanalKocan Birimi Modülleri

Sunucu Yolu Kısaltma Modülü

Bu modül, sunucunun yer değiştirme işlemlerinden dolayı, istemci ile arasında zincir şeklindeki oluşan yolu kısaltan modüldür. Bu modülün kullandığı ana metot SunucuyaDogrudanErisenVSReferansiAl metodudur.

```
public SanalKocanArayuzu SunucuyaDogrudanErisenVSReferansiAl()
{
}
```

Bu metot, istemcinin sahip olduğu Sanal Koçandan başlayarak, sunucuya doğrudan ulaşan Sanal Koçana kadar uzanan yol üzerindeki tüm Sanal Koçanlar üzerinde zincirleme şekilde çağrılır. Metot, üzerinde bulunduğu Sanal Koçanın "m_bGercek" isimli boolean değişkenini kontrol eder. Eğer, bu değişken "doğru(true)" ise, Sanal Koçan doğrudan sunucu nesnesine erişiyor demektir. Aksi takdirde, Sanal Koçan bir başka Sanal Koçana işaret ediyor ve sunucuya dolaylı yoldan erişiyor demektir. Metot içinde, eğer "m_bGercek" değişken değerinin "doğru(true)" olduğu belirlenirse, sunucuyla aynı makine üzerinde bulunan ve ona doğrudan ulaşan Sanal Koçan bulunmuştur; o Sanal Koçan nesnesinin referansı metoddan geriye döndürülür ve işlem tamamlanmış olur. Ancak, eğer "m_bGercek" değişkeni "yanlış(false)" değeri taşıyorsa, Sanal Koçanın işaret ettiği, yol üzerinde bir sonraki adımda yer alan Sanal Koçanın SunucuyaDogrudanErisenVSReferansiAl metodu çağrılır ve böylece sunucunun üzerinde yer aldığı makineye yaklaşmak için bir adım daha ilerlenmiş olunur.

Bu metodun özyineli çağrı zincirinin, geri dönüşler tamamlanarak sonlanmasından sonra, sunucuya doğrudan erişen Sanal Koçan nesnesinin referansı, sunucuya giden yol zinciri üzerinde yer alan tüm Sanal Koçanlar tarafından elde edilmiş olur. Bu referansı elde eden Sanal Koçanlar, üzerlerinde taşıdıkları işaretçileri yeni referans ile güncelleyerek sunucuya en kısa yoldan ulaşabilecekleri işaretçilere sahip olurlar.

Referans Güncelleme Modülü

Bu modül, Sanal Koçan nesnesi üzerinde tutulan ve sunucuya erişmede kullanılan işaretçinin güncellenmesinde etkin olur. Referans güncelleme işlemi, sunucu nesnenin yer değiştirmesi işlemi sonrasında ve sunucuya giden yolun kısaltılması işleminde gerçekleştirilir. Referansın güncellenmesi için, Sanal Koçan nesnesinde bulunan ReferansGuncelle metodu kullanılır.

```
public void ReferansGuncelle( Object YeniReferans, boolean
bYeniReferansGercekmi , boolean bYeniReferansYerelmi )
```

Bu metot, referansı yeni değeriyle günceller. Ayrıca, referansın gerçek sunucuya mı, yoksa bir başka Sanal Koçana mı işaret ettiğini belirleyen m_bGercek değişkeni ile sunucu nesnesinin yerel mi, yoksa bir uzak nesne mi olduğunu gösteren m_bLokal değişkenlerinin değerlerini de uygun şekilde günceller.

Metot Çağrıları Denetim Modülü

Bu modüller ana yapının bir parçası olmayıp, Dinamik RMI yapısını kullanımına örnek olarak gerçeklenmiş olan "ağda yük dengeleme" uygulaması için gerek duyularak eklenmiş modüllerdir. Sanal Koçan biriminin, her uygulamanın gereksinimlerine uygun olarak ne şekilde genişletilebileceği konusunda iyi bir örnek oluştururlar. Yük dengeleme kararları verebilmek için, belirli bir zaman diliminde, sunucu nesnesinin hangi istemci nesneleri tarafından kaçar kere çağrıldığı bilgileri kayıt altına alınmalıdır. Bu nedenle, her bir istemci ve kullandığı metot için ayrı sayaçlar tutulmuştur.

Bu amaçla kullanılan listeler ve değişkenler;

String [] IstemciIsimListesi : Metotları kullanan istemcilerin isimlerini tutan dizidir.

String [] IstemciURLListesi : Metotları kullanan istemcilerin üzerlerinde yer aldıkları makinelerin adreslerini tutan dizidir.

int m_intKayitliIstemciSayisi : Sunucunun metotlarını kullanan farklı istemcilerin sayısını tutan tam sayı tipinden değişkendir.

int [] MetotSayaclari : Her bir istemcinin, metotları kaçar kere çağırdığını tutan dizidir.

Bir istemci nesnesi, sunucunun bir metodunu çağırdığı zaman, çağrı işlemi, sunucuyla aynı makine üzerinde bulunan Sanal Koçan üzerinden gerçeklenir. Böylelikle, metot çağrılarıyla ilgili olarak oluşturulmak istenen kayıtlar tutulması, Sanal Koçan nesnesi üzerinden sunucu metodu çağrısı gerçekleşmeden önce veya sonra, uygun işlem adımlarını yürütecek olan bir özel metodun etkinleştirilmesiyle

mümkün olur. Örnek uygulamada, istemci, sunucu metodunu çağırdığı zaman Sanal Koçan üzerinde bulunan MetotSayaciArtir metodu ile çağrılan sunucu metoduna ait sayaç bir arttırılır; böylece istemci tarafından bir çağrı daha yapıldığı kayıt altına alınmış olur.

```
void MetotSayaciArtir(String IstemciAdi, String IstemciURL )
{
}
```

Eğer, istemci, sunucunun metodunu ilk defa kullanıyorsa, MetotSayaciArtir fonksiyonu içerisinde ilgili istemci adına yeni bir kayıt açılır. Bu işlemle, Istemci İsim Listesi'ne metot çağrısını yürüten istemcinin ismi eklenir, IstemciURLListesi'ne istemcinin üzerinde çalıştığı makinenin adresi eklenir, m_intKayitliIstemciSayisi değişkeni bir arttırılır ve son olarak da metot sayacı bir arttırılır.

Eğer istemci, sunucunun metodunu daha önceden çağırmışsa, ilk çağrıyı izleyen tüm metot çağrılarında MetodSayaciArtir fonksiyonu içerisinde ilgili istemciye ait metot sayacına ulaşılır ve yeni bir kayıt açılma işlemi yapılmaksızın sadece metot sayacı arttırılır.

Metot Sayacı kontrol işlemi ise, MetotSayaciKontrolu() metodu ile gerçeklenir. Bu metot, herhangi bir sayacın önceden belirlenmiş olan metot sayacı üst limitini aşıp aşmadığını, belirlenen sayıdan fazla metot çağrısı yapan bir istemcinin var olup olmadığını belirler. Eğer üst limiti geçen bir istemci bulunuyorsa, bu istemcinin üzerinde bulunduğu makinenin adresi "m_strTasinilacakAdres" değişkenine aktarılır. Böylece, sunucunun fazla metot çağrısı yapan istemci ile aynı makineye taşınması işlemlerinin ilk adımı atılmış olunur.

5.8.2 Nesne Yöneticisi Dosyası Modülleri

Sunucu Koçanını Alma Modülü

Bu modül, istemcinin sunucu nesnesinin bir metodunu ilk defa kullanmak istediği zaman devreye girer. İstemci nesnesi, sunucunun metodunu kullanacağı zaman,

üzerinde bulunduğu makinenin (Nesne Yöneticisinin) SunucununKocaniniAl(String) metodunu çağırır.

Prototip:

```
Public SanalKocanArayuzu SunucununKocaniniAl( String strSunucuAdi)
{
}
```

Metodun prototipinden de görüldüğü gibi, metot parametre olarak istemcinin erişmek istediği sunucunun adını kabul etmektedir. Sunucu adını alan Nesne Yöneticisi, öncelikle söz konusu sunucunun bir yerel nesne mi, yoksa bir uzak nesne mi olduğuna karar vermek üzere SunucuNesnesiYerelmi(String) isimli metodu çağırır. Bu metot, o makine üzerinde bulunan Yerel Nesne Listesi'ni tarar ve erişilmek istenilen sunucu nesnesinin bu listede olup olmadığı araştırılır.

Eğer, aranan sunucu nesnesi Yerel Nesne Listesi'nde bulunuyorsa, sunucunun yerel referansı alınır ve bu referans kullanılarak Sanal Koçan nesnesi yaratılır. Yaratılan bu Sanal Koçan nesnesi istemciye gönderilir ve sunucu koçanını elde etme işlemi tamamlanmış olunur.

Eğer, aranan sunucu Yerel Nesne Listesi'nde bulunmuyorsa, bu sefer Java RMI İsimlendirme Kayıtları (Naming Registry) servisi kullanılarak, sunucunun uzak referansı elde edilmeye çalışılır. Sunucunu uzak referansı elde edilemezse SunucununKocaniniAl metodu, istemciye sıfır (null) değerini gönderir ve sunucunun referansını elde edemediğini bildirir. Bunun üzerine istemci, SunucununKocaniniAl metodunu bir kere daha çağırır ve yeniden dener. Çünkü bir kere yapılan çağrı başarısız olursa, kalıcı ve ciddi bir hata olup olmadığından emin olmak için ikinci defa çağrı tekrarlanır. Yeniden deneme sonucunda da başarısız olunursa istemci çalışmasını sonlandırır.

SunucununKocaniniAl metodunda, sunucu nesnesinin uzak referansı RMI İsimlendirme Kayıtları (Naming Registry) servisinden doğru bir şekilde elde edilirse, sunucunun başka bir makine üzerinde bulunduğu anlaşılır. Bunun üzerine, sunucunun üzerinde daha önceden yaratılmış bir Sanal Koçan nesnesi bulunup bulunmadığı kontrol edilir. Eğer, böyle bir nesne bulunuyorsa bu nesnenin referansı alınır ve bu referans kullanılarak Sanal Koçan yaratılır. Böylece, istemcinin

kullandığı Sanal Koçan sunucunun makinesinde bulunan Sanal Koçana işaret eder ve sunucu metotları çağrılabilir. Fakat eğer sunucunun makinesinde daha önceden yaratılmış bir Sanal Koçan nesnesi bulunmuyorsa, istemciye göndermek üzere, doğrudan sunucunun uzak referansına işaret eden yeni bir Sanal Koçan nesnesi yaratılır.

Bu işlemler sonucunda, yeni yaratılan Sanal Koçan nesnesi, makine (Nesne Yöneticisi) üzerinde bulunan Sanal Koçan Listesine kaydedilir ve istemciye, sunucuya erişimde kullanmak üzere gerekli olan Sanal Koçan nesnesini gönderilir.

Sunucu Nesnesini Taşıma Modülü

Bu modül, sunucunun bir makineden bir başka makineye taşınması sırasında kullanılan modüldür. Bu işlemlerden sorumlu olan metot SunucuNesnesiniTasi (String GidilecekMakineAdresi) metodudur. Bu metot ve sunucu nesnesinin taşınması sırasında gerçekleşen işlemler Bölüm 5.6'da yer alan "Nesnelerin Yer Değiştirmesi" bölümünde anlatılmıştır.

Yeni Sunucu Nesnesi Yaratma Modülü

Bu modül, sunucu nesnenin bir makineden taşınıp yeni makineye aktarılması sırasında hedef makine üzerinde gerçeklenen işlemleri kapsamaktadır. Hedef makine, sunucunun taşınması sırasında üzerinde yeni bir sunucu nesnesi yaratır ve gerekli kayıt işlemlerini de geçekleştirerek sunucunun, istemcilerin kullanımı için hazır hale gelmesini sağlar. Bu modülün kullandığı temel metot YeniNesneYarat (String AnaMakineAdresi) metodudur. Bu metodun işleyişi Bölüm 5.6'da anlatılmıştır.

5.8.3 Sunucu Dosyası Modülleri

Taşınma Modülü

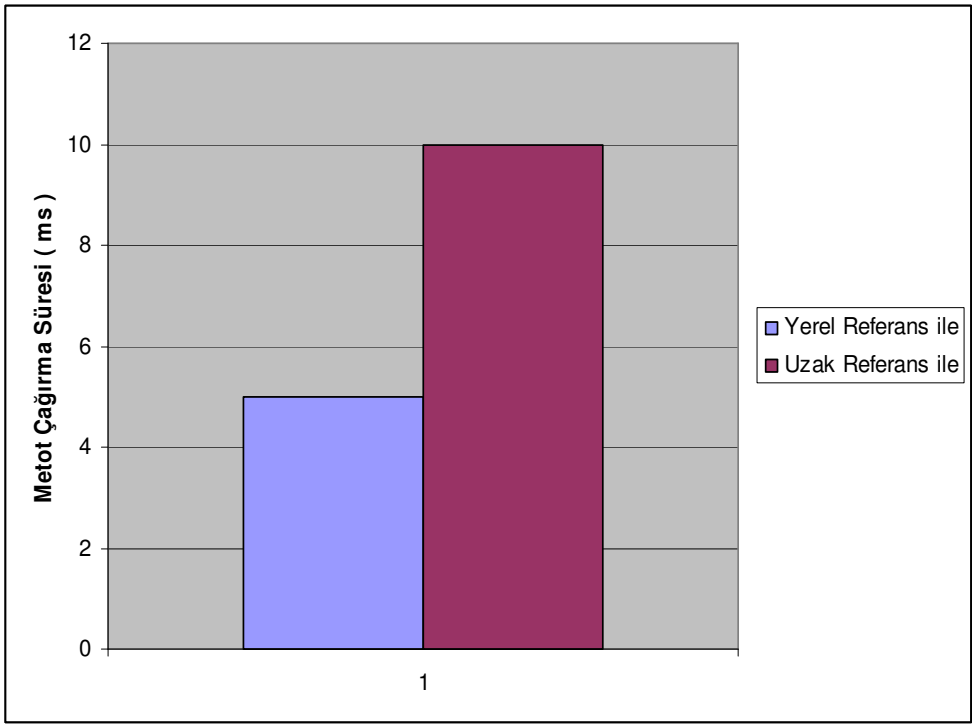
Bu modül, sunucunun bir makineden başka bir makineye taşınması sırasında kullanılan modüldür. Sunucu dosyası içerisinde bulunan Tasin (String GidilecekMakineAdresi) metodu taşınma işleminden sorumlu olan metottur. Bu metodun yaptığı iş ve Tasin metodundaki adımlar Bölüm 5.6'da anlatılmıştır.

5.9 Testler ve Ölçüm Sonuçları

Bu bölümde, D_RMI uygulamasının performansının, uzak veya yerel nesne metot çağrılarındaki gecikmelerin çeşitli senaryolarda ölçülmesi için yapılan deneyler anlatılacaktır. Yapılan deneyler bir adet diz üstü bilgisayar ve bir adet PC üzerinde gerçeklenmiştir. Diz üstü bilgisayarın sistem özellikleri, Pentium 1.8GHz Centrino işlemci, 1.00GB RAM'dir. PC'nin sistem özellikleri ise AMD Athlon XP 2200+ 1.8GHz işlemci, 256MB RAM'dir. Her iki bilgisayar üzerinde de Windows XP işletim sistemi vardır.

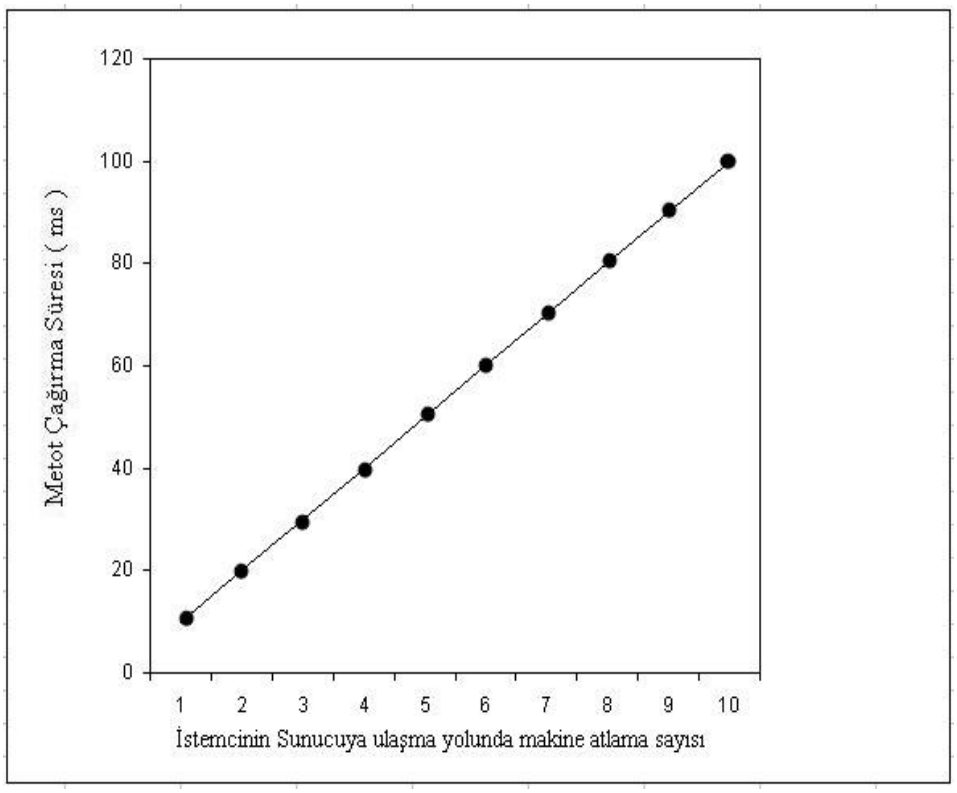
Bölüm 6'da anlatıldığı gibi D_RMI kullanımını göstermek için bir istemci-sunucu uygulaması olan "Ağ'da Yük Dengeleme" gerçekleştirilmiştir. D_RMI üzerinde gerçekleştirilen testler bu uygulama üzerinde yapılmıştır.

Aşağıdaki Şekil 5.12 'de görüldüğü üzere, sunucu ve istemci nesneleri aynı makine üzerindeyken, istemci nesnesinin sunucu üzerindeki DortIslem metodunu çağırması sunucu nesnesinin yerel referansı üzerinden yapılırken 5 ms. gibi bir süre zarfında gerçekleşmektedir. Sunucu ve istemci nesneleri farklı makinelerdeyken, istemci ve sunucu nesnesi birbirleriyle Sanal Koçanları üzerinden sadece bir adımda haberleşiyorlarken, uzak referans ile metot çağrısı 10 ms.'de gerçekleşmektedir.



Şekil 5.12 Yerel referans ve Uzak referans ile metot çağrılarındaki ölçülen süre

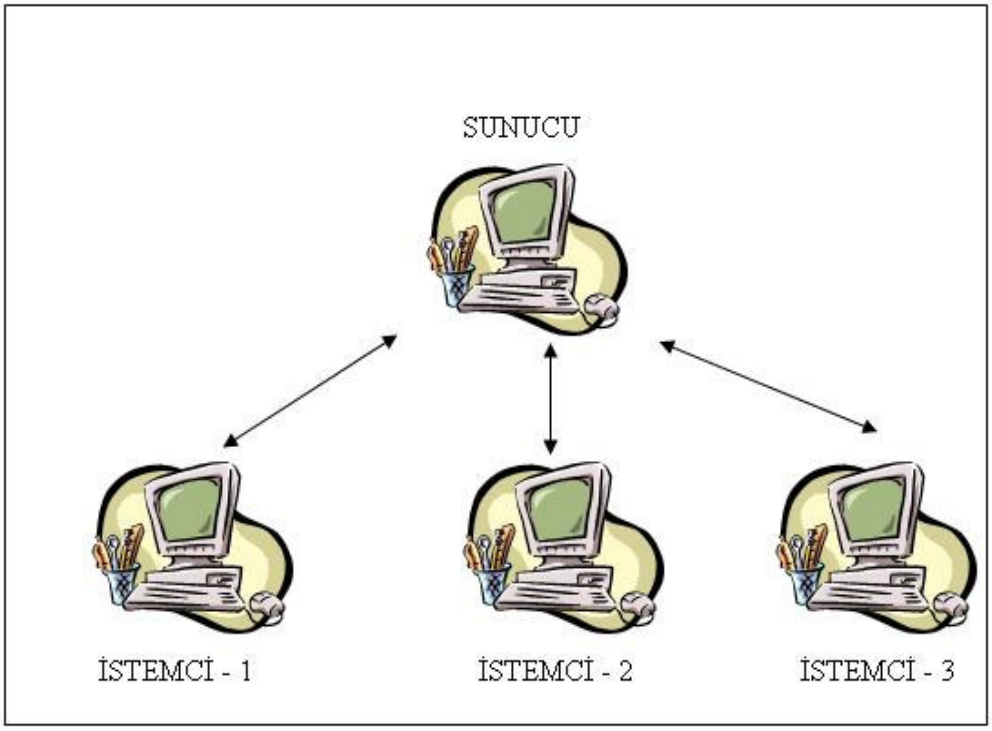
Dinamik RMI'da Bölüm 5.7'de anlatıldığı üzere yer değiştirmelerden doğabilecek istemci ve sunucu arasındaki uzun yolun kısaltılması için bir algoritma geliştirilmiştir. Bu algoritma, istemcinin sunucu metodunu çağırmasından sonra çalıştırılmaktadır ve istemci ile sunucu arasında birden fazla makineyi içine alan bir yol mevcutsa bu yolun sadece bir adıma düşürülmesi sağlanmaktadır. Bu algoritmanın çalıştırılmasının ardından istemci nesnesi sunucu metotlarını doğrudan hedef makineye ulaşarak tek bir adımda gerçekleştirmektedir, bu metot çağırma süresi de Şekil 5.12'de belirtildiği gibi 10ms. olarak ölçülmüştür. Fakat bu algoritma olmasaydı, istemci nesnesi, sunucu nesnesinin metoduna ulaşabilmek için üzerinden atlama yaptığı her bir makine için 10 ms. daha fazla bir süreye ihtiyaç duyacaktır. Şekil 5.13'de görüldüğü üzere , istemci , sunucu metodunu çağırmak için 2 makine atlıyorsa metot çağırma süresi 20ms. , 3 makine atlıyorsa 30 ms. , 5 makine atlıyorsa 50 ms. ve 10 makine atlıyorsa 100 ms. olarak ölçülmüştür.



Şekil 5.13 İstemcinin Sunucu metodu çağırması aşamasında aradaki makine sayısı ve süre grafiği

6 UYGULAMA – AĞDA YÜK DENGELEME

Mobil RMI sistemini ve özelliklerini göstermek için seçilen uygulama "Ağda Yük Dengeleme" uygulamasıdır. Bu uygulamada, bir adet sunucu ve birden fazla istemci nesneleri bulunmaktadır. Sunucu nesne, istemci nesnelere hizmet vermektedir ve tüm bu istemci nesneleri ve sunucu nesne farklı makinelerde bulunabilmektedir (Şekil 6.1) .



Şekil 6.1: Dağıtık Sistemlerde Sunucu ve Istemciler

Eğer, sunucu ile istemci nesneler aynı makine üzerinde bulunuyorsa birbirleriyle yerel olarak haberleşmektedirler. İstemci, sunucunun metotlarını normal bir yerel nesnenin metodunu çağırıyormuş gibi kullanabilmektedir.

Eğer, sunucu ile istemci nesneler farklı makineler üzerinde bulunuyorlarsa temelde Java RMI kullanılarak uzak metot çağrımı gerçekleştirilmektedir. Fakat Dinamik

RMI'da, normal RMI'dan farklı olarak RMI'nın standart yaratılan koçanları kullanılmamaktadır. Onun yerine daha esnek olan, yeni özellikler ve işlevler ekleyebildiğimiz Sanal Koçanlar kullanılmaktadır.

Dinamik RMI'da bir istemcinin sunucu metotlarını kullanabilmesi için sunucunun o an hangi makine üzerinde bulunduğunu bilmesine gerek yoktur. İstemciler, Sanal Koçanlar üzerinden saydam bir şekilde sunucunun metotlarına erişebilmektedir. Bu sebeple, sunucu, dinamik olarak bir makineden başka bir makineye, istemcilerden bağımsız olarak yer değiştirebilmektedir. Sunucunun hangi makinede olduğu bilgisine sahip olmaması veya bilinen bir makineden bir başkasına yer değiştirmiş olması, istemci nesnenin sunucuya erişmesini ve metotlarını kullanmasını engellemeyecektir.

Gerçeklenen uygulamada, sunucu nesnesi, istemci nesnelerine dört işlem hizmeti vermektedir. İstemci nesneleri, sunucunun DortIslem metodunu çağırabilmektedirler.

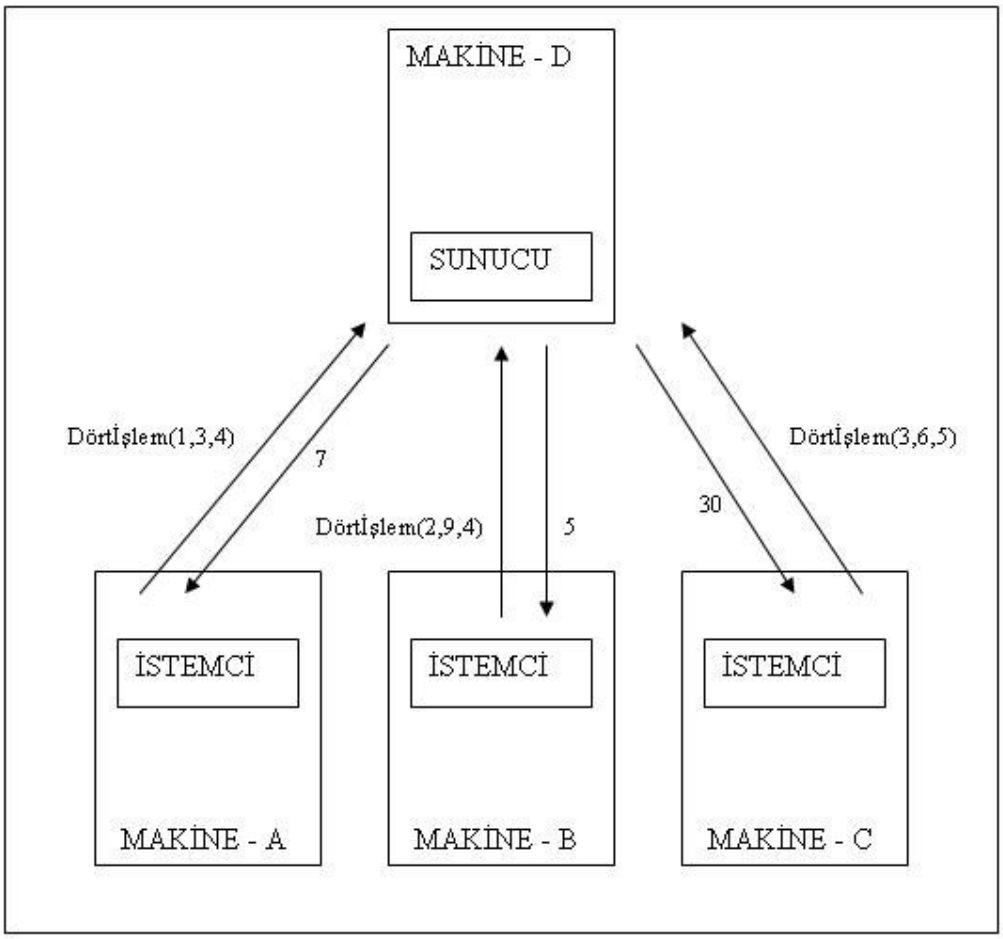
```
int DortIslem(int IslemTipi, int X1, int X2, String ClientName, String ClientURL )
{
}
```

Bu metot, "IslemTipi" değişkeninin aldığı değere göre, X1 ve X2 değişkenleri üzerinde dört işlemden birini uygulayıp sonucunu geri döndürmektedir.

IslemTipi değişkeni:
1 ise toplama,
2 ise çıkarma,
3 ise çarpma,
4 ise bölme işlemi yapılmaktadır.

X1 ve X2 parametre değişkenleri işlemin uygulanacağı tam sayıları göstermektedirler.

Şekil 6.2'de farklı makineler üzerinde bulunan istemcilerin, yine farklı bir makinede (Makine – D) bulunan sunucu nesnesinin dört işlem metodunu çağırarak sonucu elde etmeleri gösterilmiştir.

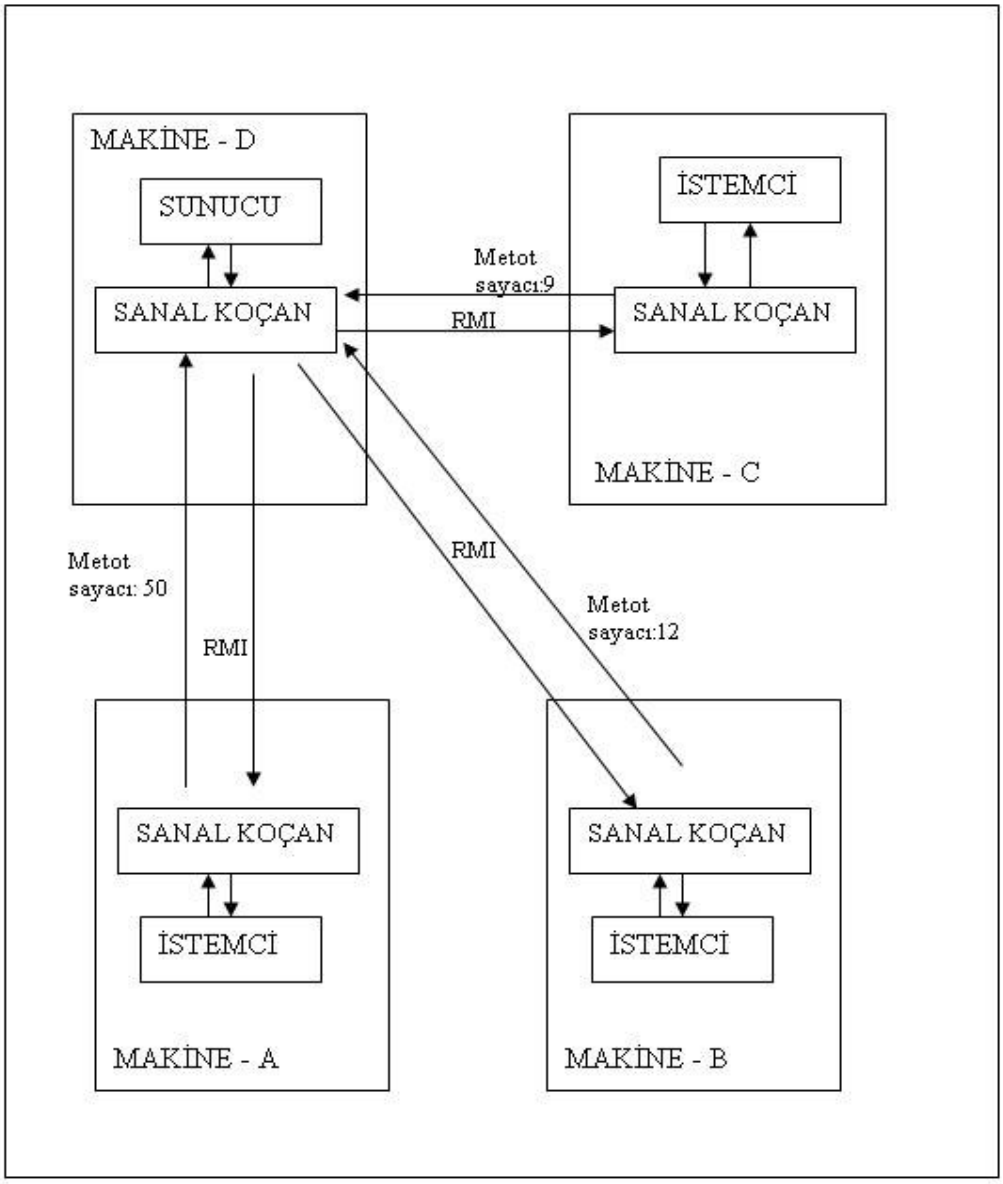


Şekil 6.2: Farklı Makinelerdeki İstemci-Sunucu Haberlesmesi

Yük Dengeleme işlemi ise ağ üzerindeki veri yoğunluğunu mümkün olan en aza indirmeyi amaçlamaktadır. Geliştirilen uygulamada, farklı makinelerde bulunan istemciler, sunucu nesnesinin yerel veya uzak nesne referansını kullanarak dört işlem metodunu çağırıp hizmet almaktadırlar. Herhangi bir istemci nesnesi, sunucuyu yoğun bir şekilde kullanıyorsa, sunucu nesnesinin istemcinin bulunduğu makineye taşınmasına karar verilir. Bu taşınma işlemi sonrasında sunucuyu yoğun olarak kullanan istemci sunucuya yerel olarak ulaşmaya ve sunucudan hizmet almaya başlayacaktır. Böylece, ağ üzerinde Java RMI'yı kullanarak bir ağ yoğunluğuna sebep olmayı bırakacaktır. Mobil RMI sayesinde de, sunucuyu kullanan diğer istemciler, sunucunun bu yer değiştirmesinden etkilenmeyecek, üzerlerinde herhangi

bir değişiklik yapmaya gerek olmadan çalışmalarına ve sunucu hizmeti almaya devam edeceklerdir.

Şekil 6.3'de A, B ve C makinelerinde bulunan istemciler, D makinesi üzerinde bulunan sunucunun metodunu kullanmaktalardır. Şekilde de görüldüğü gibi, istemciler, sunucu metotlarına ile Dinamik RMI'nın gelişmiş bir özelliği olan "Sanal Koçan"ları kullanarak erişmektelerdir. Uygulamada, her bir istemci için , sunucu metodunu kaçar kere çağırdıklarını tutan metot sayaçları bulunmaktadır. A makinesinde bulunan istemci, sunucu metodunu önceden belirlenmiş bir süre içerisinde 50 kere, B makinesinde bulunan istemci 12 kere, C makinesinde bulunan istemci ise 9 kere çağırmıştır. Bu sonuçlardan, A makinesindeki istemcinin, sunucu nesnesinin metodunu yoğun bir şekilde kullandığı görülmektedir.

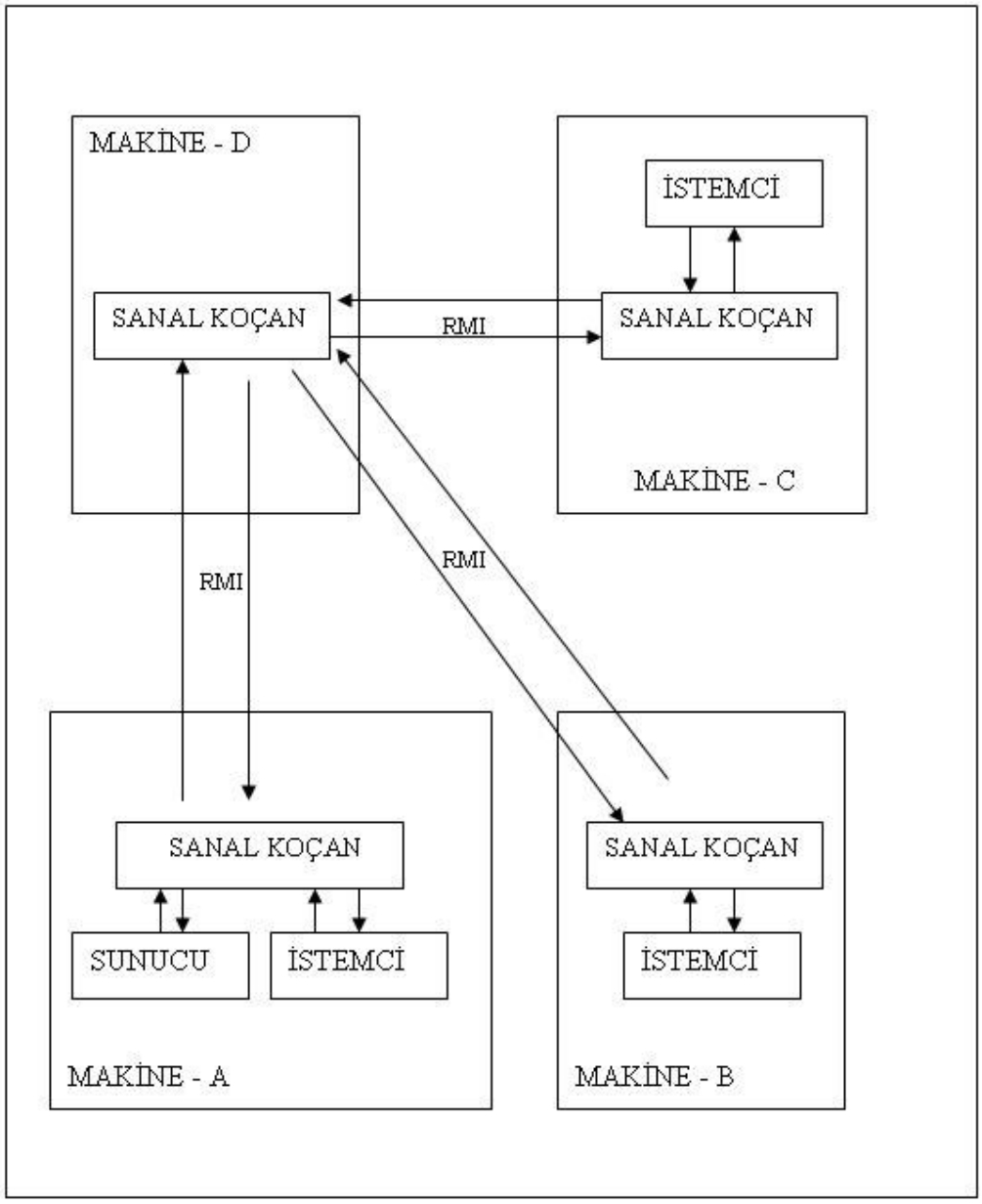


Şekil 6.3: İstemcilerin Sunucuyu Kullanmaları

Mobil RMI sistemini kullanarak gerçekleştirilen bu ağ dengeleme uygulamasına göre, çalışma anında, D makinesinde bulunan sunucu A makinesine taşınmalıdır. Böylece, D makinesinde bulunan sunucu nesnesini yoğun bir şekilde kullanan istemcinin ağ üzerinde yarattığı veri alışverişi yoğunluğu engellenmiş olunur. Sunucunun da A makinesine taşınmasıyla birlikte, A makinesinde bulunan istemci, sunucuya yerel referans ile erişebilecektir ve sunucu nesnesinin metotlarını Java RMI kullanmaksızın çağırabilecektir. Bu da ağ üzerindeki yoğunluğu dengeleyecek ve daha etkin bir çalışmayı sağlayacaktır. B ve C makinelerindeki istemciler içinse

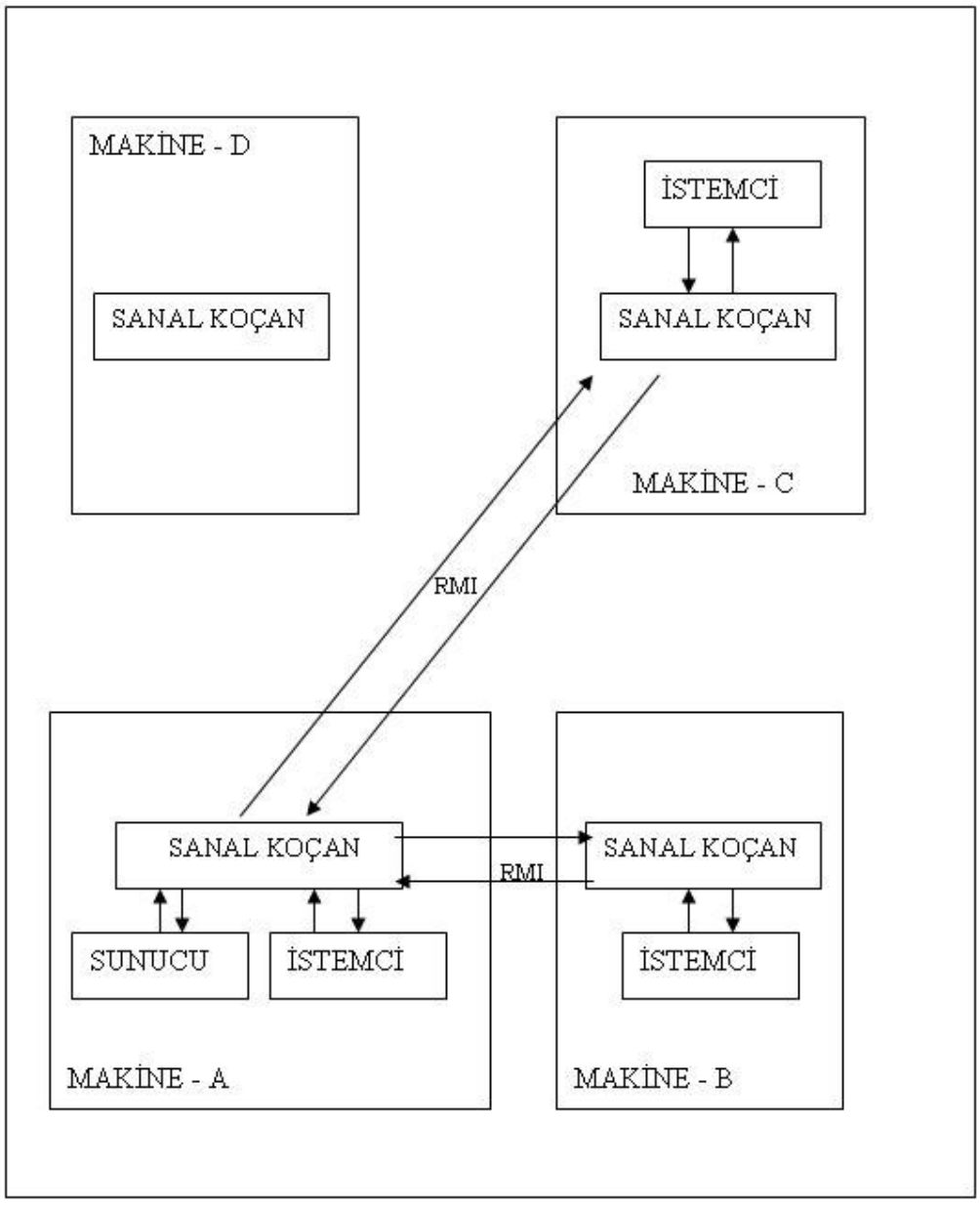
değişen bir şey olmamaktadır. B ve C makinesi üzerindeki istemciler önceki durumda D makinesindeki sunucu nesnesinin metotlarını Java RMI kullanarak "Sanal Koçan" kullanarak çağırmaktalardı. Yeni durumda ise, herhangi bir yeniden düzenleme yapmalarına gerek olmadan yine D makinesi üzerinde bulunan "Sanal Koçan"ı kullanarak sunucu metotlarına erişmeye devam edeceklerdir. Tek fark olarak, yer değiştirmeden sonra D makinesi üzerinde bulunan "Sanal Koçan" sunucuya yerel olarak doğrudan erişmeyecektir. Bunun yerine, A makinesi üzerinde bulunan "Sanal Koçan"a erişerek sunucu metotlarını kullanabileceklerdir. Tüm bu işlemler, Mobil RMI sistemi tarafından, istemciden habersiz, istemciye tamamen saydam bir şekilde gerçekleştirilmektedir.

D makinesinde bulunan sunucu nesnesinin A makinesine taşındıktan sonraki durum aşağıdaki Şekil 6.4'de gösterilmiştir.



Şekil 6.4: Sunucu Yer Değiştirdikten Sonra İstemci-Sunucu Bağlantıları

Yer değiştirme işlemi gerçeklendikten sonra, "Sunucu-İstemci Yolu Kısaltılması" işlemiyle aradaki nesnelerin ve makinelerin temizlenmesi, böylece sunucu ile istemci nesnesi arasındaki yolun kısaltılması sağlanır. Sunucu nesnesinin D makinesinden A makinesine taşınmasıyla birlikte, B ve C makinesinde bulunan istemciler gereksiz yere D makinesi üzerinden sunucuya ulaşmaktadırlar. "Sunucu-İstemci Yolu Kısaltılması" işleminden sonra B ve C makinelerinde bulunan istemciler doğrudan A makinesi ile haberleşeceklerdir. Yeni durumdaki istemci ve sunucu ilişkileri Şekil 6.5'de gösterilmektedir.



Şekil 6.5: Yol Kısaltma Algoritmasından Sonraki Yeni İstemci-Sunucu Bağlantıları

7 SONUÇ

Bu tez, Java RMI gibi dağıtık sistemlerde bileşenlerin haberleşmesini sağlayan yapıların bir takım ek modüller yardımı ile nesnelerin hareketlerinden sonra da çalışmalarını dinamik olarak sürdürebilmelerinin başarılı bir şekilde sağlandığını göstermektedir. Bahsedilen bu ek modüller, uygulama katmanı ile Java RMI katmanı arasında bulunacak ve çalışacak olan Dinamik RMI (D_RMI) ara katmanı bünyesinde bulunmaktadır.

Tez kapsamında 5. bölümde açıkça görüldüğü gibi, D_RMI çalışan dağıtık sistemlerin, nesne yer değiştirmelerinden sonra da çalışmalarını sürdürecek bir yapı sunmaktadır. Bu yapı içerisinde RMI'daki normal koçanlar gibi uzak nesne metotlarına erişim için arayüz görevi gören Sanal Koçan'lar bulunmaktadır ve bu Sanal Koçanlar aynı zamanda da çalışan dağıtık sistemin dinamik konfigürasyonu sırasında sistemi izleme, kontrol etme, ara işlemler gerçekleştirme görevlerini de yerine getirirler. Buna ek olarak, Nesne Yöneticisi modülü de nesnelerin hareket etmesinin ardından geçersiz olan uzak nesnelerin mevcut referanslarını yenileriyle güncellemektedir. Böylece, yer değiştiren nesne ile haberleşen diğer nesneler için herhangi bir işlem yapmaya gerek kalmadan dağıtık sistemin çalışmasının devamı sağlanmış olur. Bunlardan başka, Nesne Yöneticisi , çalışma anında otomatik olarak bileşenlerin uzak veya yerel olup olmama durumlarını kontrol eder ve gerekli değişiklikleri yapar. Mesela, aynı makinede bulunan iki bileşenden birisinin başka bir makineye geçmesinin ardından iki nesne birbirleri için otomatik olarak uzak nesne haline getirilecektir. Böylece, sadece iki nesnenin fiziksel olarak farklı yerlerde olma durumunda RMI kullanılır. Aksi takdirde iki nesnenin birbirleriyle haberlesmeşinde RMI kullanımına gerek yoktur, yerel referans üzerinden birbirleriyle konuşurlar. D_RMI'nın bu özelliği çalışma performansı açısından önemli bir avantaj sağladığı açıktır.

Bu çalışmada, D_RMI yapısı içerisinde, nesnelerin yer değiştirmelerinden sonra birbirleriyle konuşan iki nesnenin (istemci ve sunucunun) arasındaki mesafenin minimuma düşürülmesini sağlayan bir de algoritma geliştirilmiştir. Hareketli bir

nesnenin yer değiştirmelerinden sonra, bu nesneyi kullanan diğer nesnelerin ilgili nesneye ulaşmaları için birden fazla makine üzerinde atlama yapmaları gerekebilmektedir. Bu da uzak nesnenin metodunu çağırma zamanındaki gecikmenin atlama sayısıyla doğru orantılı bir şekilde artması anlamına gelecektir. İdeal durum istemci nesnesinin sunucu nesnesinin metodunu sadece tek bir hamlede çağırabilmesidir. D_RMI içerisinde geliştirilen nesneler arasındaki yol kısalmasını sağlayacak algoritma, kaynak nesne ile hedef nesne arasındaki mesafeyi bir adıma düşürmektedir. Böylece, metot çağırma zamanı minimuma çekilerek performans arttırılmaktadır.

Sonuç olarak, bu tez çalışması ile dağıtık sistemlerdeki nesnelerin hareket etmelerinden sonra sistemin çalışması dinamik olarak sağlanmaktadır. Bu dinamizm dağıtık sistem uygulamalarında sıkça kullanılan Java RMI üzerinde bir ara katman olan D_RMI tarafından sağlanmıştır. Böylece, D_RMI'yı kullanmak isteyen uygulama geliştiricileri, yeni bir programa dili öğrenmelerine gerek olmadan, Java RMI kullanarak içerisinde hareketli nesnelerin bulunduğu dinamik uygulamalar geliştirebileceklerdir.

KAYNAKLAR

[1] **Sun Microsystems.** 1999. Introduction to CORBA, http://java.sun.com/developer/onlineTraining/corba/corba.html.

[2] **Microsoft**, 2006. COM / DCOM, http://www.microsoft.com/com.

[3] **Sun Microsystems, Inc.**, 2006. Core Java, Java Remote Method Invocation (Java RMI) . http://java.sun.com/products/jdk/rmi/index.jsp

[4] **Sun Microsystems.,** 2001**.** Enterprise JavaBeans, http://java.sun.com/products/ejb/index.html.

[5] **Chen, X. , Simons, M.**, 2002. A Component Framework for Dynamic Reconfiguration of Distributed Systems. J. Bishop (Ed.): CD 2002, LNCS 2370, pp. 82.96.

[6] **Soğukpınar,A.**, 2001. Java'nın Hikayesi. http://www.programlama.com/sys/c2html/view.php3?DocID=454

[7] **Altıntaş, A.B.**, 2004. Java Kitap Projesi. http://www.kodcu.com

[8] **TechRepository.com** , 2004. http://www.techrepository.com

[9] **Biltek Dergisi** – IEEE ODTÜ Ögrenci Kolu , Mart 2002.

[10] **Gülaş, Y.** , 2003. Çoklu Görevcik Kullanımı. http://www.programlama.com/sys/c2html/view.php3?DocID=543&PF=1

[11] **Daconta, M.C. , Saganich, A. , Monk, E.** , 1999. Java 2 and JavaScript for C and C++ Programmers –Revised Edition.

[12] **Java Remote Method Invocation Specification**, 1997. Revision 1.4, JDK.

[13] **Chen, X.** , 2002. Dependence Management for Dynamic Reconfiguration. Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE'02).

[14] **Chen, X.** , 2002. Extending RMI to Support Dynamic Reconfiguration of Distributed Systems. Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)

[15] **Avvenuti, M. , Vecchio, A.**, 2005. MobileRMI: Upgrading Java – Remote Method Invocation towards mobility. *Softw. Pract. Exper.* 2005; **35**:939–975

[16] **Holder, O., Ben-Shaul, I. and Gazit, H.**, 1999. System Support for Dynamic Layout of Distributed Applications. In: Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99), Austin, TX, USA, pages 403-411, 1999.

[17] **Türk Dil Kurumu Bilgisayar Terimleri Karşılıklar Kılavuzu,** http://tdk.org.tr/bilterim/

[18] **Bilişim Sözlüğü,** http://www.tbd.org.tr/index.php?module=dict, 2004

ÖZGEÇMİŞ

Göksel Sarıkaya: 08 Mart 1981'de İstanbul, Kadıköy'de doğdu. Lise eğitimini Rıfat Canayakın Lisesi'nde tamamlayarak 1998 yılında İstanbul Teknik Üniversitesi Bilgisayar Mühendisliği bölümüne girdi. Lisans eğitimini 2003 yılında tamamladı ve aynı yıl İstanbul Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği programında Yüksek Lisans eğitimine başladı. 2003 yılından bu yana SIEMENS A.Ş. bünyesinde yazılım projelerinde mühendis olarak çalışıyor.