UZAK METOT ÇAĞRIMI
REMOTE METHOD INVOCATION - RMI

Mehmet AKPOLAT

Hacettepe Üniversitesi Bilgisayar Mühendisliği Bölümü
Lisans Öğrencisi

mehmet.akpolat@gmail.com

İçindekiler

1. Önsöz

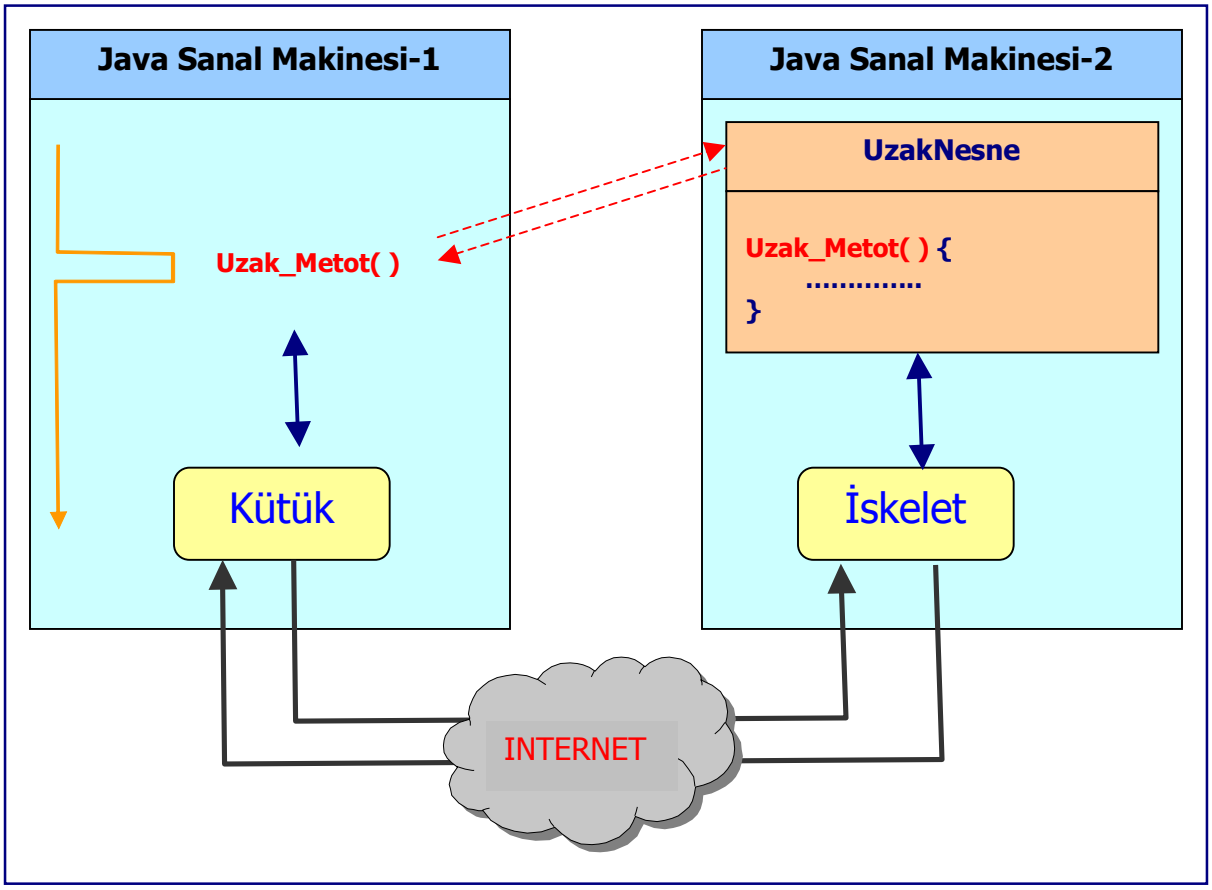
Son yıllarda bilişim dünyasında yapılan dev hamlelere ulusumuzun da ayak uydurmaya çalışması, sevindirici olmasına karşın bu alanda yazılmış olan Türkçe kaynak eksikliği belirgin bir biçimde hissedilmektedir.İşte, Uzak Metot Çağrımı (Remote Method Invocation-RMI) ile alakalı olarak tamamen ana dilimizde hazırlanmış bu kaynak, az da olsa bu konudaki eksikliği kapatmak amacını taşımaktadır.

2. Uzak Metot Çağrımı(*RMI*) Nedir?

Uzak Metot Çağrımı(*RMI*), ilk olarak *JDK™ 1.1* ile gündeme gelmiş ve ağ programlamayı daha yüksek seviyelere taşımıştır. Uzak Metot Çağrımı 'nın kullanımının nispeten kolay olmasının yanında fark edilir güçte teknoloji ler arasında gösterilebilir. Uzak Metot Çağrımı, ortalama Java geliştiricilerini dağıtık işlemler dünyası adı verilen yeni bir paradigma ile tanıştırmaktadır. Uzak Metot Çağrımı, dağıtık sistemlerdeki klasik Uzak Yordam Çağırma (*Remote Procedure Call-RPC*) mantığına benzer şekilde, uzaktaki bir bilgisayar üzerinde çalışan "uzak nesne" nin metotlarını çağırabilmek olarak bilinmektedir.Diğer dağıtık programlama arayüzlerinin(Arayüz Tanımlama Dili(*IDL*), Uzak Yordam Çağırma(*RPC*), vb.) aksine Java Uzak Metot Çağrımı(*RMI*), Java programlama diline özgüdür(*language specific*).

Uzak Metot Çağrımı, dağıtılmış bir ortamda yerel çalışıyormuşcasına imkanlar sağlayan, uzaktan nesnelere ileti göndermeye yarayan temel bir teknoloji olarak görülebilir.Büyük çaplı yazılımlar başarılarını Uzak Metot Çağrımı 'na borçludurlar. Bu tür büyük çaptaki yazılımlarda Java *RMI* teknolojisi kullanıldığından, istemci ve sunucu da Java programlama dili ile kodlanacağı için dağıtık programlama tümüyle Java programlama dilinde gerçekleşmektedir.Böylelikle Java programlama dilinin özellikleri ve avantajları dağıtık programlamada da kullanılabilmiş olmaktadır. Java *RMI* ağ işlemleri için kullanıcıya soket veya streamlere göre daha üst düzeyde bir arayüz sunmaktadır. Bu nedenle, *RMI* ile dağıtık programlama yapmak soket ve stream kullanımına göre daha az karmaşıktır. Programcı açısından bakıldığında, *RMI* kullanıldığında istemci/sunucu uygulamaların geliştirilmesi sırasında ağ işlem alt düzeydeki ayrıntıları ile uğraşmak gerekmemektedir.

Çizim 1'de Java Sanal Makinesi-1 üzerinde bulunan bir Java uygulamasının (veya Java *applet* 'inin) Java Sanal Makinesi-2 üzerinde bulunan UzakNesne isimli bir nesnenin *Uzak_Metot()* isimli bir metodunu çağrımı görülmektedir.



- Çizim 1 : RMI ile Uzak Metot Çağrımı -

* Dökümanın bundan sonraki bölümlerinde Uzak Metot Çağrımı ifadesinin yerine kolaylık sağlanması açısından *RMI* ifadesi kullanılacaktır.

3. Uzak Metot Çağrımı'nın Amacı Nedir?

RMI tasarımcılarının başlıca amacı uygulama geliştiricilerinin dağıtılmış olmayan programlarda kullanılan Java sözdizim ve semantik kurallarını dağıtılmış programlarda da kullanabilmesini sağlamaktır.Bunu sağlamak amacıyla tasarımcılar tek bir Java Sanal Makinesi(*JVM*) içeren sistemlerdeki sınıf ve nesnelerin çalışmasını dikkatli bir biçimde birden çok Java Sanal Makinesini içeren dağıtık işlem ortamları için modellemişlerdir.

RMI mimarisi, nesnelerin davranışlarını, aykırı durumların nasıl ve ne zaman meydana geldiğini, belleğin nasıl yönetildiğini ve son olarak parametrelerin uzak metotlara nasıl aktarıldığını ve bu metotlardan nasıl döndürüldüğünü tanımlar.

4. Dağıtık ve Dağıtık Olmayan Uygulamaların Karşılaştırılması

RMI 'nın mimarları dağıtık Java nesnelerin kullanımını yerel nesnelerin kullanımına benzetmeyi amaçlamışlardır.Bunu başarmış olsalar da ortaya çıkan bazı farklılıklar aşağıda Tablo 1 dahilinde açıklanmıştır.

	Yerel Nesne	**Uzak Nesne**
Nesne Tanımı	Bir yerel nesne Java sınıfı tarafından tanımlanır.	Uzak nesnenin davranışı Uzak(*Remote*) arayüzü sağlayan bir arayüz tarafından tanımlanır.
Nesne Gerçekleştirimi	Yerel nesnenin gerçekleştirimini Java sınıfı yapar.	Uzak nesnenin davranışları uzak arayüzün gerçekleştirimini yapan bir sınıf tarafından uygulanır.
Nesne Yaratılması	Yerel nesnenin yeni bir örneği new işleci ile yaratılır.	Uzak nesnenin yeni bir örneği sunucu bilgisayar üzerindeki new işleci yaratılır. İstemci direkt olarak yeni bir uzak nesne yaratamaz(Java 2 Uzak Nesne Aktivasyonu dışında).
Nesne Erişimi	Yerel nesneye bir nesne referans değişkeni ile ulaşmak mümkündür.	Uzak nesneye erişim, uzak arayüzün vekil kütük(*proxy stub*) gerçekleştirimini gösteren bir referans değişkeni ile mümkündür.

Referanslar	Tek Java Sanal Makinesi (*JVM*)içeren sistemlerde bir nesne referansı yığındaki bir nesneyi işaret eder.	Uzak referans, yerel yığındaki bir vekil nesneye(*stub*) bir işaretçidir. Bu vekil nesne(*stub*), uzak metotların gerçekleştirimini içeren uzak nesneye erişim bilgilerini içerir.
Aktif Referanslar	Tek Java Sanal Makinesi içeren sistemlerde, eğer bir nesneyi gösteren bir referans mevcut ise o nesne yaşıyor demektir.	Dağıtık ortamlarda uzak JVM iflas edebilir veya bağlantı kopabilir. Uzak bir nesneye belirli bir zaman periyoduyla kiralanmışcasına erişilebiliniyorsa, bu nesnenin aktif bir referansının olduğu. Eğer uzak nesneyi işaret eden tüm aktif referansları durdurulmuş veya kiralama süresi sona ermişse, bu referanslar artık çöp biriktiriciye(*garbage collector*) atılmaya hazır hale gelmiştir.
Sonlandırma	Eğer bir nesne *finalize()* metodunu gerçekleştirimini yapmışsa, bu metot nesne için ayrılan alanın çöp biriktiriciye (*garbage collector*) teslim edilmesinden önce çağırılır.	Eğer bir uzak nesne referansgösterilmeyen (*Unreferenced)* arayüzünün gerçekleştirimini yapmışsa, bu arayüzün metodu bütün referanslar durdurulduğunda çağırılır.
Çöp Biriktirici	Eğer bir nesnenin referanslarının tümü durdurulmuşsa, o nesne çöp biriktirici (*garbage collector*) için artık belirgin bir adaydır.	Dağıtık çöp biriktirici, yerel çöp biriktirici ile birlikte çalışır. Eğer ortada uzak referans kalmamışsa ve uzak nesnenin bütün yerel referansları durdurulmuşsa, belirtilen uzak nesne çöp biriktirici (*garbage collector*) için artık belirgin bir adaydır.
Aykırı Durumlar	Java derleyicisi uygulamaları tüm olası aykırı durumları ele almaya zorlamaktadır.	RMI bütün uygulamalarını olası tanımlı *RemoteException* sınıfı dahilindeki tüm aykırı durum nesnelerini göz önünde bulundurmalarını hususunda zorlar. Bundaki amaç dağıtık uygulamaların sağlamlığını(*robustness*) arttırmaktır.

- Tablo 1: Dağıtık ve Yerel Nesneler arasındaki farklılıklar -

5. Neden Uzak Metot Çağrımı?

Çünkü,

- ◇ Java *RMI* modelinin öğrenilmesi ve kullanılması kolaydır.
- ◇ Taşınabilirdir, bu nedenle farklı platformlarda çalışabilme yetisi vardır.
- ◇ Güvenilir dağıtık uygulama geliştirimini sağlamaktadır.
- ◇ Birçok kütüphane ve kaynağı beraberinde sağlamaktadır.
- ◇ Sunuculardan *applet* 'lere geri çağrımları sağlamaktadır..

6. Uzak Metot Çağrımı Mimarisi

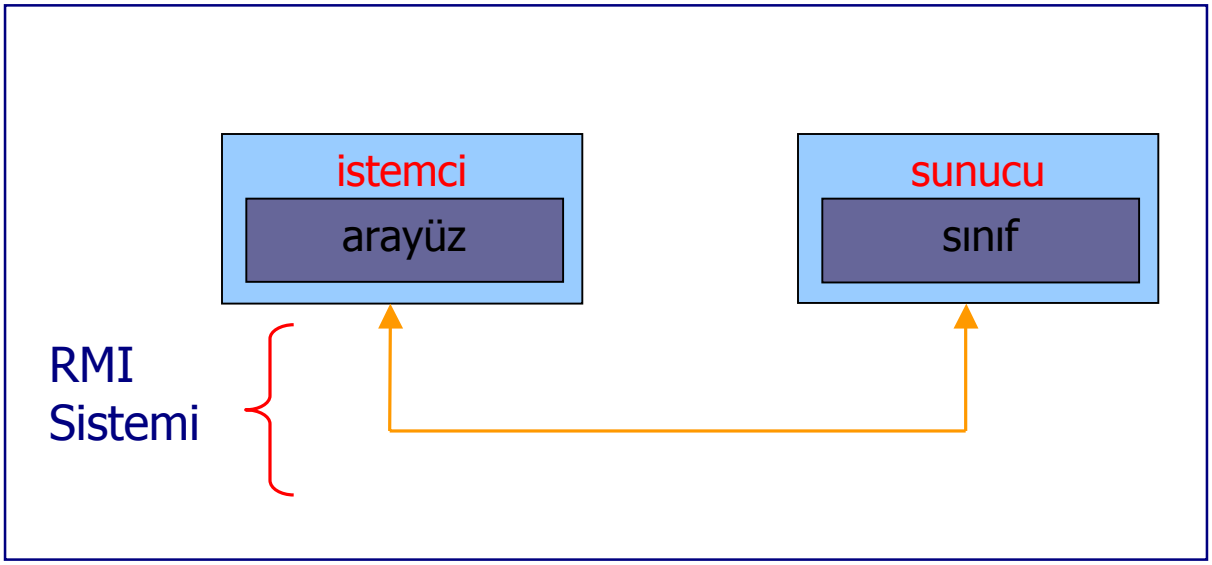
RMI mimarisinin tasarım amacı, Java programlama dilini ve yerel nesne modelini bünyesinde bütünleştirmiş yeni bir dağıtık nesne modelini ortaya koymaktı.Bu düşünce ile yola çıkan *RMI* mimarları, hazır olan Java mimarisinin güvenliğini ve sağlamlığını dağıtık işlem dünyasına taşımayı başarmışlardır.

6.1. Arayüzler – RMI 'nın Kalbi

RMI mimarisi önemli bir prensibi temel edinmiştir: *davranışın tanımı ve gerçekleştirimi birbirinden tamamen ayrı kavramlardır*. Bu düşünceden yola çıkarak *RMI*, belirli bir davranışın tanımını yapan kod parçası ile yine aynı davranışın gerçekleştirimini yapan kod parçasını ayrı Java Sanal Makineleri(*JVM*) üzerinde düşünmektedir.Bu istemcilerin belirli bir servisin tanımı ile ilgilendikleri ya da sunucuların servis hizmetlerini sağlamaya odaklandığı dağıtık sistemlerle uyuşmaktadır.

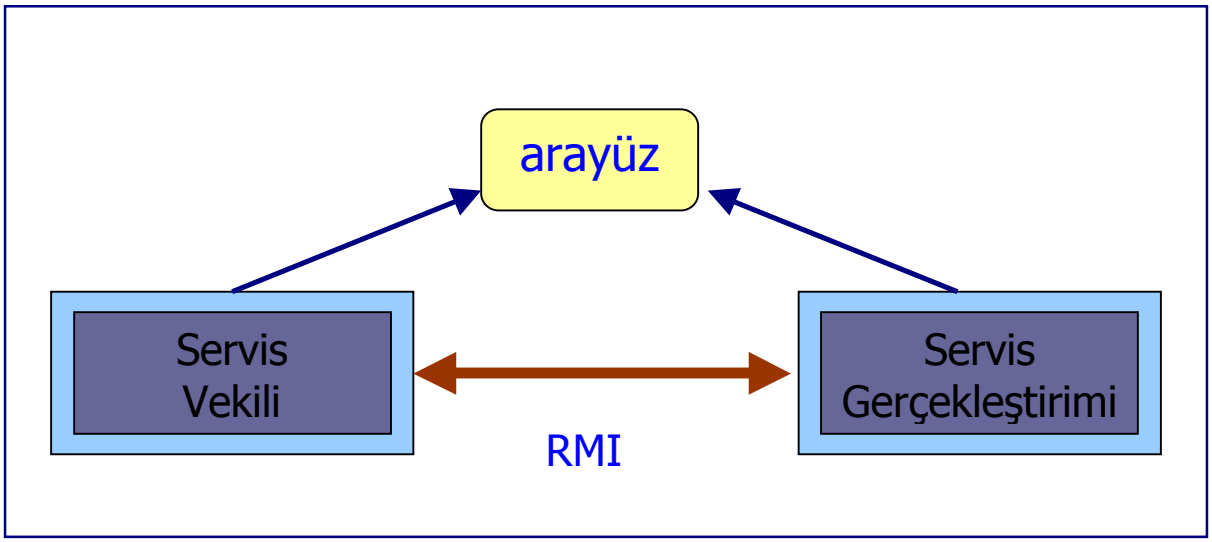
RMI 'da özellikle uzak servislerin tanımları Java arayüzleri(*interfaces*) kullanılarak kodlanmaktadır. Bu yapılan uzak servis tanımlarının gerçekleştirimi ise sınıflar (*classes*) içerisinde kodlanmaktadır. Böylelikle bu noktada arayüzlerin davranışları tanımladığını, sınıfların ise bu davranışların gerçekleştirimini yaptığını özetlermek yararlı olacaktır.

Aşağıda verilmiş olan Çizim 2, bize bu ayrımı daha somut olarak sunmak amacıyla verilmiştir.



- Çizim 2 : RMI 'da sınıf & arayüz kavramları -

Bahsettiğimiz gibi Java arayüzleri, çalıştırılabilir kod içermezler.Dolayısıyla *RMI*, uzak arayüzün(*remote interface*) gerçekleştirimini içeren iki adet sınıfı desteklemiştir. Birinci sınıf davranışın gerçekleştirimi içeren sınıftır ve sunucu üzerinde faaliyet gösterir. İkincisi sunucu tarafından sağlanan hizmetlere erişimde vekil(*proxy*) rolünü üstlenmektedir ve istemci tarafında faaliyet göstermektedir.Aşağıda verilmiş olan Çizim 3, bize belirtilen iki sınıfı akılda iyice somutlaştırmak amacıyla verilmiştir.



- Çizim 3 : Sağlanan servisler için Vekil & Gerçekleştirim sınıfları -

İstemci program metot çağrılarını çizim dahilinde belirtilen vekil(*proxy*) nesnelere yapar. *RMI*, yapılan bu çağrıyı gerçekleştirimin yapıldığı uzak sunucudaki Java Sanal Makinesine(*JVM*) yönlendirir. Gerçekleştirim tarafından döndürülen değerler önce vekil(*proxy*) ve vekil nesnesinden de metot çağrısını yapan istemciye yönlendirilir.

* Üst düzey *RMI* mimarisinin çalışma mekanizmasını basitçe açıkladıktan sonra şimdi bu mimariyi oluşturan alt katmanlara geçebiliriz.Eğer bu noktaya kadar bahsedilen hususlarda bir eksiklik hissediliyorsa, bu hussuların tekrar gözden geçirilmesi tavsiye edilir.

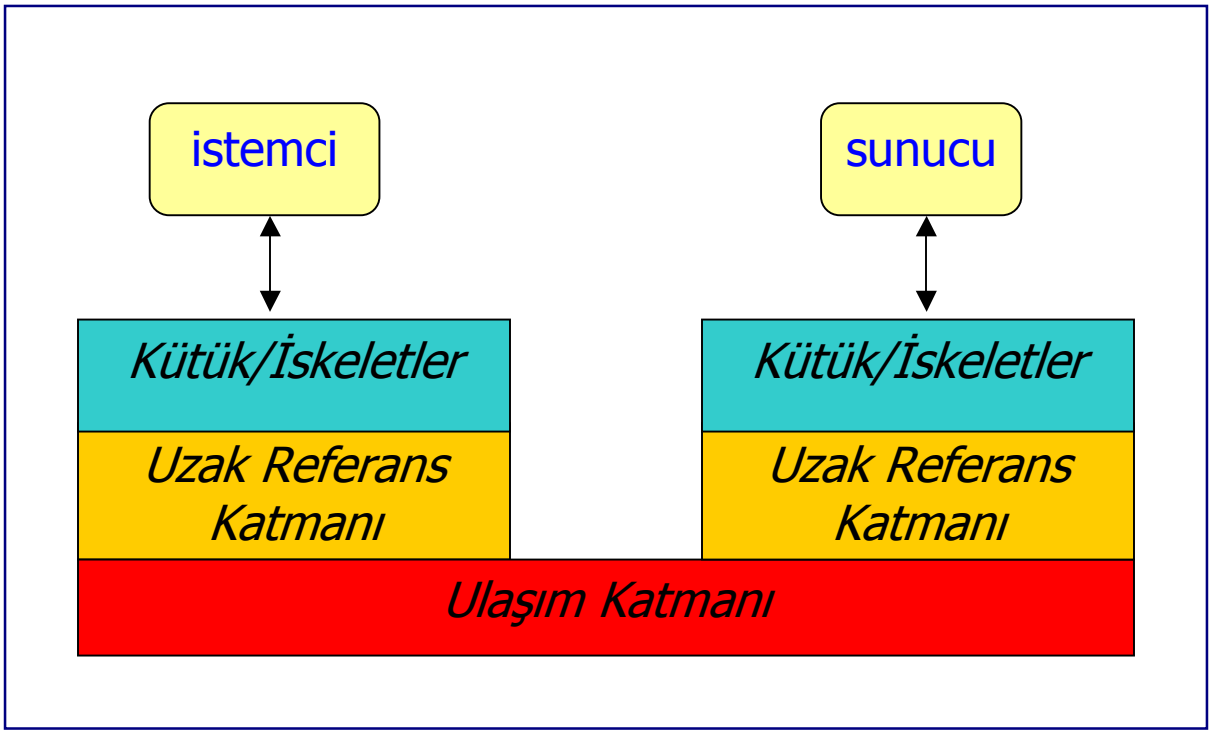
6.2. RMI Mimarisi Katmanları

RMI, gerçekleştirimi gereği esasında 3 soyutlama katmanından oluşmaktadır. İlk soyutlama katmanı olan Kütük&İskelet(*Stub&Skeleton*) Katmanı, istemci tarafından arayüz referans değişkenine yönelik yapılan metot çağrılarını keser ve bu çağrıları uzak *RMI* servisine yönlendirir.

Sonraki katman olan Uzak Referans(*Remote Reference*) Katmanı, istemcilerden yapılan referans istemlerini yorumlayıp, yönetme işlevlerine sahiptir.*JDK 1.1* 'de bu katman çalışan istemci ile ilgili uzak servis nesnesinin bire-bir olarak iletişimlerini sağlardı. *Java 2 SDK* 'da ise bu katman, Uzak Nesne Aktivasyonu(*Remote Object Activation*) kullanılarak uykudaki uzak servis nesnelerinin de aktivasyonunu desteklemek amacıyla geliştirilmiştir.

Son katman olan Ulaşım(*Transport*) Katmanı, ağ üzerindeki makineleri birbirine bağlayan TCP/IP(*Transport Control Protocol/Internet Protocol*) tabanlı bağlantı prensibini temel almaktaktadır.

RMI mimarisinin katmanlı bir yapıya sahip olması sonucunda, mimarinin diğer katmanlarını değiştirmeden her bir katman üzerinde yeni eklemeler veya değişiklikler yapılabilir.Mesela Ulaşım(*Transport*) Katmanı, üst katmanları etkilemeden UDP/IP (*User Datagram Protocol/Internet Protocol*) katmanı ile değiştirilebilir.

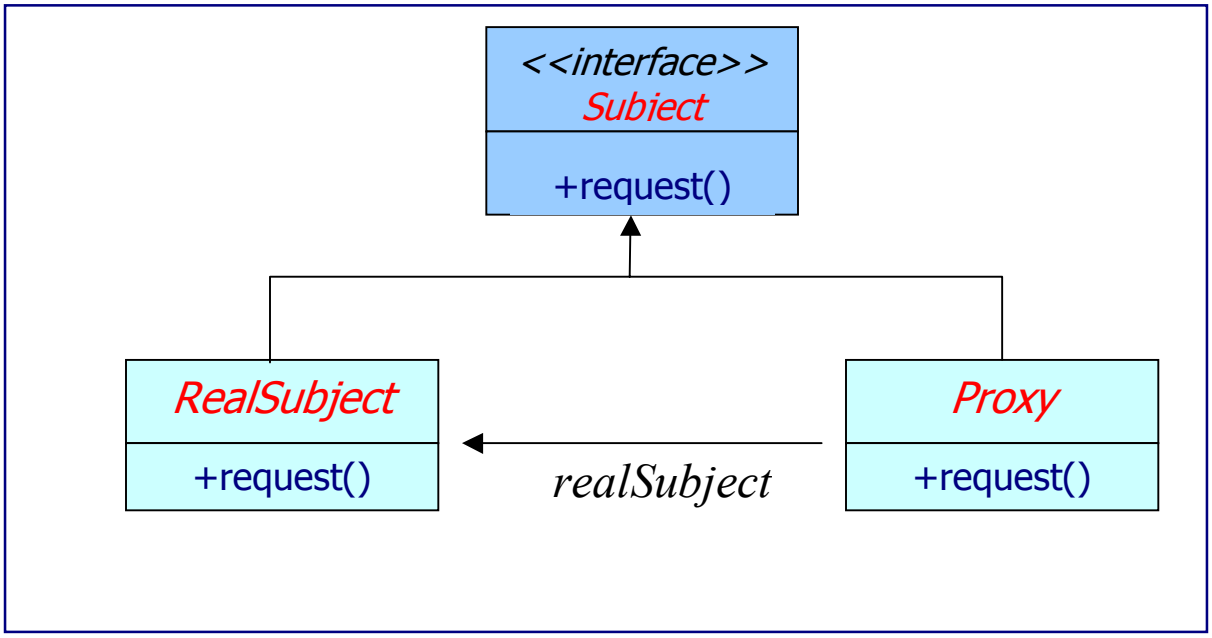


- Çizim 4 : RMI Mimarisi Katmanları -

6.2.1. Kütük & İskelet Katmanı

Bu katmanda *RMI*, *Design Patterns* (*Gamma, Helm, Johnson, Vlissides*) kitabında bahsi geçen *Proxy* tasarım örüntüsünü kullanmaktadır. *Proxy* tasarım örüntüsünde, belirli bir bağlamdaki bir nesne başka bağlamdaki bir vekil nesne(*proxy*) ile temsil edilebilmektedir. Vekil(*proxy*) nesne, katılımcı nesneler arasında metot çağrımlarını nasıl ileteceğini bilmektedir.

Aşağıda verilmiş olan Çizim 5 'de *Proxy* tasarım örüntüsüne ait sınıf diyagramını bulabilirsiniz. Bu sınıf diyagramı dikkatlice analiz edilecek olunursa, *RMI* mimarisinde kütüğün(stub) vekil(*proxy*) ; uzak servis gerçekleştirimini içeren sınıfın ise *RealSubject* rolünde olduğu anlaşılabilmektedir.



- Çizim 5 : Proxy Tasarım Örüntüsü -

İskelet(*skeleton*), *RMI* mimarisinde kullanılmak üzere düşünülmüş yardımcı bir sınıftır. İskelet, *RMI* bağlantısı üzerinden kütük(*stub*) ile nasıl iletişim kuracağını bilir. İskelet kütük ile konuşmasında, bağlantı üzerinden aldığı uzak metot çağrımı için gerekli parametreleri uzak servis gerçekleştirim nesnesine iletir. Bu gerçekleştirim nesnesi ise parametreleri alır ve işlem adımları sonucunda bulduğu verileri tekrar iskelete gönderir ve iskelet de bunu ilgili kütüğe iletir.

6.2.2. Uzak Referans Katmanı

Uzak Referans Katmanı, *RMI* bağlantılarının metot çağrım semantiklerini belirler ve destekler. Bu katman uzak gerçekleştirim nesnesine olan bağlantıyı temsil eden *RemoteRef* adında bir referans nesnesini sağlar.

Kütük(*stub*) nesneleri *RemoteRef* nesnesi içindeki *invoke( )* metodunu kullanarak uzak metot çağrımını iletirler. *RemoteRef* nesnesi uzak servisler için gönderilen metot çağrım semantiklerini anlar.

RMI 'nın *JDK 1.1* 'deki gerçekleştiriminde istemcilerin uzak servis gerçekleştirimlerine bağlanabilmeleri için tek bir yol mevcuttur: noktadan-noktaya(*point-to-point*) bağlantı.

Uzak servisin istemci tarafından kullanılabilmesi için öncelikle bu servisin sunucu tarafında yaratılması, eğer temel bir servis ise *RMI* sistemine kayıt(*register*) edilmiş olması gerekmektedir.

Java 2 SDK , *RMI* gerçekleştiriminde istemci-sunucu paradigmasına yeni bir semantik eklemiştir. Bu sürümde, *RMI* aktif edilebilir uzak nesneleri desteklemektedir. Vekile(*proxy*) aktif edilebilir uzak nesnelere erişim için bir metot çağrımı yapıldığı zaman, *RMI* uzak servis gerçekleştirim nesnesinin uykuda olup olmadığını belirler. Eğer nesne uykuda bulunmuşsa, *RMI* nesneyi yaratır ve önceki durum bilgilerini disk üzerindeki bir kütükten(*file*) geri yükler. Eğer nesne uykuda değil de hafızada hazır durumda ise, *JDK 1.1* 'deki uzak servis gerçekleştirim nesneleri gibi davranır.

Başka çeşit bağlantı semantikleri de mümkündür. Mesela, bire-çok yayın (*multicast*) bağlantısında bir vekil(*proxy*) nesne, metot isteğini birden çok gerçekleştirim nesnesine gönderir ve ilk gelen cevabı alır. Ayrıca ilerleyen yıllarda *Sun* 'ın RMI 'ya çağrım semantikleri alanında ekler yapması mümkündür.

6.2.3. Ulaşım Katmanı

Ulaşım Katmanı, Java sanal makineleri(*JVMs*) arasındaki bağlantıyı kurar. Tüm bağlantılar TCP/IP(*Transport Control Protocol/Internet Protocol*) protokolünü kullanan akış tabanlı ağ bağlantılarıdır.

İki farklı Java sanal makinesi aynı bilgisayarda çalışmasına rağmen birbirleriyle iletişim kuracakları zaman ana bilgisayarın TCP/IP ağ protokol yığıtını kullanırlar. TCP/IP protokolü kalıcılığın yanı sıra iki bilgisayar arasında IP ve port numaralarını kullanarak akış tabanlı bağlantı imkanı sağlar. Genellikle DNS(*Domain Name Server*)'in IP adresleri yerine kullanıldığını düşünürsek ; ortaya *cs.hacettepe.edu.tr* : 3249 ve *nakpolat.firat.edu.tr* : 4278 arasında bir TCP/IP bağlantısı çıkmış olur. *RMI* 'nın yayımlanmış güncel sürümünde tüm bilgisayardan bilgisayara bağlantılarda TCP/IP bağlantısı temel olarak kullanılmaktadır.

TCP/IP 'nin en üstünde *RMI* tarafından kullanılan protokol JRMP(*Java Remote Method Protocol*)'dir. *JRMP*, kısmen iki versiyonu olan patentli ve akış tabanlı bir protokoldür. İlk versiyonu *JDK 1.1* 'in RMI sürümü ile yayımlandı ve bu versiyonunda sunucu tarafında iskelet(skeleton) sınıflarına ihtiyaç duyulmaktadır. İkinci versiyonu ise *Java 2 SDK* ile yayımlandı. Bu versiyonu performans açısından optimize edilmiş ve iskelet sınıflarının sunucu tarafında kullanımına gerek duyulmamıştır. Ayrıca *BEA Weblogic* ve *NinjaRMI* gibi bazı alternatif gerçekleştirimler, TCP/IP 'nin en üstünde protokol olarak *JRMP* 'yi kullanmayabilirler.

Sun ve *IBM*, RMI 'nın sonraki versiyonu için birlikte çalışmışlar ve sonuç olarak Java SDK sürüm 1.3 ile birlikte gelen *RMI-IIOP* versiyonunu yayımlamışlardır. *RMI-IIOP* versiyonunda istemci-sunucu arası iletişimin sağlanması için JRMP protokolü yerine OMG (Object Management Group) tarafından yayımlanmış olan IIOP(*Internet Inter-ORB Protocol*) 'nin kullanılması dikkat çekicidir.

OMG (Object Management Group), CORBA (*Common Object Request Broker Architecture)* gibi önemli bir satıcı-yansız(*vendor-neutral*) dağıtık nesne mimarisini tanımlamıştır ve 800 'den fazla üyesi bulunan bir gruptur. *CORBA* mimarisi dahilindeki istemci ve sunucular birbirleri ile iletişim kurarken *IIOP* protokolünü kullanırlar. Yeni *RMI-IIOP* versiyonu birçok özelliği ile RMI 'ya benzemektedir fakat örtüşmediği özellikler de vardır.

RMI 'nın ulaşım katmanı istemci ve sunucular arasında bağlantıyı kurmak amacıyla tasarlanmıştır. Ulaşım katmanı birden çok TCP/IP bağlantılarını kullanmayı tercih etmesine karşın bazı ağ ayarları bir istemci ile sunucu arasında sadece tek TCP/IP bağlantısına izin verirler. Bu nedenle ulaşım katmanı, tek TCP/IP bağlantısını kullanarak çoklama yöntemi ile birden çok sanal bağlantı imkanı sağlar.

7. Uzak Nesnelerin Adlandırılması

RMI mimarisinin bu noktaya kadar olan sunumu sırasında, önemli bir sorunun cevabının henüz verilmediğini fark etmiş olabilirsiniz: "*İstemci herhangi bir RMI uzak servisini nasıl bulacak?*". İşte bu sorunun cevabını bu bölümde alacaksınız.

İstemciler, uzak servisleri bulabilmek için bir çeşit isimlendirme(*naming*) ve dizin(*directory*) servisine başvururlar. Bu servis kısaca JNDI (*Java Naming and Directory Interface)* olarak adlandırılmıştır. Bu dolambaçlı bir mantığa benzeyebilir: *İstemci bir servisi yine bir servis kullanarak nasıl yerleştirebilecektir?* Gerçeği söylemek gerekirse durum bundan ibarettir. İsimlendirme ve Dizin Servisi(*JNDI*), herhangi bir organizasyonda herkes tarafından bilinen ana bilgisayar ve *port* numarasıyla çalışır.

RMI, *JNDI* ile birçok farklı dizin servisi kullanabilir.Ayrıca *RMI* kendine ait basit bir rehberi de barındırmaktadır: *rmiregistry* . Bu rehber, uzak servis nesnelerine ev sahipliği yapan ve servislerle alakalı tüm sorguları kabul eden her bir bilgisayar üzerinde faaliyet gösterir. Bu servis belirtilen bilgisayarlarda eğer bir *port* numarası değişikliği yapılmadıysa, 1099 nolu port üzerinde faaliyet gösterir.

Ev sahipliği yapan bir bilgisayar üzerinde çalışan sunucu program, uzak servisin gerçekleştirimini yapmış yerel bir nesne oluşturarak, ilgili uzak servisi yaratmış olur. Sonraki adım dahilinde yerel nesneyi *RMI* 'ya aktarır. RMI ise kendisine yerel bir nesne aktarılınca, istemcilerin ilgili servise bağlanıp istemde bulunmalarını beklemek için bir dinleme servisi(*listening service*) yaratır. Sunucu program ise yerel nesneyi aktardıktan sonra *RMI* 'nın yerel rehberine(*rmiregistry*) bu nesneyi herkese açık bir isimle kaydeder.

İstemci tarafında, *RMI* 'nın yerel rehberine(*rmiregistry*) *Naming* statik sınıfı kullanılarak erişim sağlanır. Bu statik sınıf, istemcilerin rehberde arama yapmada kullandıkları *lookup()* metodunu sağlamaktadır. *lookup()* metodu parametre olarak sunucu programa ev sahipliği yapan bilgisayar adını ve erişilmek istenen servis adını kabul etmektedir.

Bahsedilen parametrenin yapısı aşağıda verilmiştir:

rmi://<host_adi>[:<isimlendirme_servisi_port>]/<servis_adı>

host_adi → sunucu programa ev sahipliği yapan bilgisayar adı(yere ağlarda ayırt edilebilen bilgisayar adları veya Internet üzerindeki DNS adı olabilir)
isimlendirme_servisi_port → İsimlendirme servisi 1099 nolu port dışında bir port üzerinde çalışıyorsa,o port numarası
servis_adi → erişilmek istenen servis adı

8. Java RMI 'nın Kullanılması

Şimdi çalışan bir *RMI* sistemi oluşturarak deneyim sahibi olmanın vakti geldi. Şimdiye kadar anlatılanları somutlaştırmak amacıyla bir amortisman fiyat listesi uygulama örneği verilecektir.

Çalışan bir RMI sistemi çeşitli kısımlardan oluşmuştur:

- Uzak servislerin gerçekleştirimi
- Kütük(*stub*) ve iskelet(*Skeleton*)
- Uzak servislere ev sahipliği yapacak sunucu
- İstemcilerin uzak servisleri bulmasını sağlayan bir *RMI* isimlendirme servisi
- Sınıf kütüğü sağlayıcısı (http veya ftp sunucusu)
- Uzak servisleri kullanacak olan istemci

İlerleyen bölümlerde, basit bir *RMI* sisteminin nasıl yaratılacağı adım adım gösterilmektedir. Verilecek olan kaynak kodlar kullanılarak oluşturulacak kaynak kütükler tek bir alt dizinin altında tutulabilir.

RMI sisteminin önceden tasarlandığı düşünülürse, aşağıdaki adımlar takip edilerek RMI sisteminin oluşturulması tamamlanabilir:

1. Arayüzler için Java kodunu yazın ve derleyin.
2. Gerçekleştirimler için Java kodunu yazın ve derleyin.
3. Gerçekleştirim sınıflarından kütük(*stub*) ve iskelet(*skeleton*) sınıf kütüklerini(*files*) oluştur.
4. *RMI* istemci programı için gerekli Java kodunu geliştir.
5. Uzak servise ev sahipliği yapacak program için Java kodunu yaz.
6. *RMI* sistemini yükle ve çalıştır.

Yukarıda verilen işlem adımlarını bir amortisman fiyat listesi uygulaması oluşturmak üzere adım adım uygulamaya çalışalım. Bu örnek dahilinde istemci uzak nesneyi kullanarak yerel *fiyatListesi* nesnesini sunucudan ister ve bunun için ödünç verme süresini ve miktarını sunucuya gönderir. Bunun ardından, sunucu kendisine gönderilen parametrelere ek olarak sadece kendisinin bildiği ilgi oranı değerini kullanarak yerel *fiyatListesi* nesnesini yaratır.Daha sonra yaratılan nesne bit dizisi haline dönüştürülerek (*serialization*) istemciye gönderilir. Bu noktadan itibaren istemci elde ettiği nesneyi ister görüntüler, isterse de üzerinde değişiklik yapabilir. Yani, istemcinin artık kendine ait yerel *fiyatListesi* nesnesi vardır.

8.1. Arayüzü Oluşturma

RMI uygulaması geliştirirken yapılması gereken ilk adım, uzak arayüzün tanımının yapılmasıdır. Bu uzak arayüz, uzak nesneleri kullanarak hangi metot ve değişkenler için istemde bulunabileceğini belirtir. Ama genellikle bu arayüzlerin tanımlarında sadece kullanılabilecek metotlar bulunur.

Bu uzak arayüz *RMI* paketini kullanma(*import*) ihtiyacını duymaktadır. Arayüz sınıfının tanımında, bu sınıfın *java.rmi.Remote* sınıfına erişebileceğini(*extends java.rmi.Remote*) belirtmesi gerekir ve tanımı verilen her bir metot *java.rmi* paketi içinde tanımlı *RemoteException* sınıfı türündeki aykırı durumları göz önünde bulundurabilmelidir.Örnek RMI sistemimizin arayüz tanımlama kütüğü olan *Hesapla.java* aşağıda verilmiştir.

```
/**************************************************
                        Hesapla.java
 **************************************************/
import java.lang.*;
import java.rmi.*;

//  istemcilerin uzak nesneler ile alakalı istemleri için ilgili
// metot tanımlarını içeren uzak arayüz sınıfı
public interface Hesapla extends java.rmi.Remote
{
   // verilen miktar ve süre parametrelerini kullanarak fiyat hesaplaması yapan uzak metot
   public fiyatListesi amortismanFiyatListesi( float miktar,int sure ) throws
                                        java.rmi.RemoteException;
   // ilgi oranı bilgisini görüntüleyen uzak metot
   public void ilgiOraniGoster( ) throws java.rmi.RemoteException;
}
```

8.2. Arayüz Gerçekleştirimini Oluşturma

Arayüz tanımlama kütüğü olan *Hesapla.java* oluşturulduktan sonra sıra sunucu tarafında bu arayüzü destekleyecek olan gerçekleştirim kütüğünü oluşturmaya geldi. Aşağıda arayüze sunucu tarafında gereken desteği verecek olan *HesaplaImp.java* arayüz gerçekleştirim kütüğünün örneği verilmiştir.

```
/*****************************************************
                        HesaplaImp.java
 *****************************************************/
import java.rmi.*;
import java.rmi.server.*;

// uzak arayüzde tanımları verilmiş olan uzak metotların
// gerçekleştirimlerini içeren gerçekleştirim sınıfı
public class HesaplaImp extends UnicastRemoteObject implements Hesapla
{
   private float ilgiOrani; // ilgi oranı

   public HesaplaImp(float ilgiOrani) throws java.rmi.RemoteException
   {
      this.ilgiOrani = ilgiOrani; // ilgi oranını alınan değerle kur
   }

   // verilen miktar ve süre parametrelerini kullanarak fiyat hesaplaması yapan uzak metot
   public fiyatListesi amortismanFiyatListesi ( float miktar, int sure ) throws
                                                                  java.rmi.RemoteException
   {
      System.out.println("Amortisman Fiyat Listesi....");
      return  new fiyatListesi( ilgiOrani, miktar, sure );
   }

   // ilgi oranı bilgisini görüntüleyen uzak metot
   public void ilgiOraniGoster ( ) throws java.rmi.RemoteException
   {
      System.out.println("Su anki ilgi orani :" + ilgiOrani ); // ilgi oranı bilgisini görüntüle
   }
}
```

Dikkatle inceleyecek olursak arayüz gerçekleştirim kütüğü *java.rmi* ve *java.rmi.server* paketlerini kullanma ihtiyacı hissetmektedir. Ayrıca uzak istemcileri desteklemek için gerçekleştirim sınıfı olan *HesapImp*, *UnicastRemoteObject* sınıfına erişebilmelidir(*extends*). *HesapImp* sınıfı *RMI* için istemci-sunucu ve bire-bir bağlantısını yönetir.

Günümüzde *MultiCastRemoteObject* sınıfı bulunmamaktadır fakat bu sınıfı yayımlanmış olan bazı *JDK 1.2* 'lerde bulmak mümkündür. Ayrıca *JDK 1.1* 'de kendimize ait *MultiCastRemoteObject* sınıfları oluşturarak bire-çok yayın yapabilen uzak istemcileri desteklememize izin verilmektedir.

Son olarak arayüz gerçekleştirim sınıfı olan *HesaplaImp* sınıfının önceden tanımlanan *Hesapla* arayüz sınıfının gerçekleştirimini yaptığı belirtilmelidir ve bunun için " *implements Hesapla* " ifadesi eklenmelidir.

8.3. Nesneleri Dizi Haline Getirme(*Object Serialization*)

amortismanFiyatListesi( ) metodu sunucu tarafında bir mesaj görüntüleme işlevini yerine getirir ve yeni bir yerel *fiyatListesi* nesnesi yaratarak istemci tarafına gönderir. Yerel *fiyatListesi* nesnesi dizi haline dönüştürülüp sıralanarak veri akımları halinde istemciye gönderilecek olan nesnelerdir. Şimdi artık uzak nesnelerin dizi haline dönüştürülmesinin tartışılmasına geçebiliriz.

Eğer uzak arayüzleri kullanarak yerel nesneleri gönderiyorsak, yerel sınıfın tanımını dizi haline dönüştürülebilir(*serializable*) olarak yapmak gerekir. Dikkat edecek olursak aşağıda *fiyatListesi* sınıfının tanımında bu işlemin *implements serializable* deyimi ile yapıldığı görülmektedir. Burada *serializable* arayüzünün gerçekleştirimini *fiyatListesi* sınıfının yapması amaçlanmamaktadır. Çünkü, bunu Java kendisi yönetmektedir. Eğer bize ait *fiyatListesi* sınıfının tanımında *externalizable* arayüzünün gerçekleştirimini yapacağımızı belirtirsek, *fiyatListesi.java* kütüğü içerisinde serialize/deserialize metotlarının gerçekleştirimini yapmak orunda kalırız. Bunun amacı *fiyatListesi* sınıfının kendine ait verileri dizi haline getirme işini kendisinin yürütmesidir. Eğer tanımında dizi halinde gönderilebilir ifadesi(*implements serializable*) bulunmayan yerel nesneyi göndermek istersek, Java istemci ya da sunucu tarafında sıralama(*marshaling*) hususundaki aykırı durumu saptar ve bildirir.

Ayrıca bir sınıfı dizi olarak gönderilebilir(*serializable*) olarak tanımlarken dikkatli olmak gerekir , çünkü Java bu sınıftan kalıtım yolu ile elde edilen sınıfları düzleştirmeye(*flatten*) çalışacaktır. Bu tür sınıfları dizi olarak göndermeye çalışmaktan kaçınılmalıdır.

```
/*****************************************************
                        fiyatListesi.java
 *****************************************************/
import java.lang.*;
import java.util.*;
import java.io.*;

// gerekli hesaplamaları yaparak fiyat listesini oluşturmayı sağlayan sınıf tanımı
public class fiyatListesi implements Serializable
{
  // fiyat listesi hesaplamada kullanılacak olan sınıf değişkenleri
  float  toplamOdunc;
  float  miktar;
  float  ilgiOrani;
  int  sure;

  // ilgili sınıf değişkenlerini alınan parametre değerlerine kurar
  public fiyatListesi ( float ilgiOrani, float miktar, int sure )
  {
     System.out.println("Fiyat Listesi Yaratildi....");
     this.ilgiOrani = ilgiOrani;
     this.miktar   = miktar;
     this.sure = sure;
     toplamOdunc = miktar + (miktar / ilgiOrani);
  }

  // uygun hesaplama algoritmasını uygulayarak bulduğu fiyat listesi verilerini görüntüler
  public void goster( )
  {
     int   kuponNumarasi = 0;
     float bakiye = toplamOdunc;
     float aylikOdeme = 0;

     System.out.println("Hesaplamada kullanilan veriler :");
     System.out.println("          Ilgi Orani       [%" + ilgiOrani + "]" );
     System.out.println("          Miktar     [$" + miktar + "]" );
     System.out.println("          Sure    [ " + sure + "]" );
     System.out.println("          Toplam Odunc  [$" + toplamOdunc + "]" );
     System.out.println( "\nAylik Odeme  Odeme Miktar   Kalan Bakiye");
     System.out.println( "------- ----------------------   ----------------");
```

```
        while( bakiye > 0 )
        {
            kuponNumarasi ++;
            aylikOdeme = toplamOdunc / sure;
            if( bakiye < aylikOdeme )
            {
                aylikOdeme  = bakiye;
                bakiye = 0;
            }
            else
            {
                bakiye = bakiye - aylikOdeme;
            }

            System.out.println( kuponNumarasi + " " + aylikOdeme + " " + bakiye );
        }
    }
}
```

8.4. Kütük(*stub*) ve İskelet(*skeleton*) Oluşturma

Arayüz ve gerçekleştirim kütüklerinin yaratılmasının ardından şimdi kütük (stub) ve iskelet(skeleton) kodlarını oluşturma işlemine geçebiliriz. Bu işlem JDK taarfından sağlanan *rmic* derleyicisini kullanarak gerçekleştirilebilir. Bu derleyici uzak servis gerçekleştirim sınıfı kütüğü üzerinde çalıştırılır.

```
> rmic HesaplaImp
```

Komut satırında çalışma dizinimiz içine kadar ilerleyerek geldikten sonra yukarıdaki biçimi ile rmic derleyicisini çalıştırırsak, bulunduğunuz dizin altında *HesaplaImp_Stub.class* ve eğer Java 2 SDK kullanıyorsak *HesaplaImp_Skel.class* kütükleri oluşturulacaktır.

JDK 1.1 'deki RMI derleyicisi olan *rmic* için seçenekler aşağıda belirtilmiştir

Kullanım : **rmic <seçenekler> <sınıf isimleri>**

<*seçenekler*> aşağıdaki opsiyonları içermektedir:

-keep	: arada üretilen kaynak kütüklerin silinmemesini belirtir.
-keepgenerated	: -keep ile aynı işleve sahiptir.
-depend	: tarihi geçmiş kütükleri özyineli olarak yeniden derlemeyi sağlar.
-nowarn	: derleme işlemi sırasında uyarı üretilmemesini bildirir.
-verbose	: derleyicinin yaptığı işlemler hakkındaki mesajları görüntüler.
-classpath <yol>	: derlenecek Java girdi kütüklerinin hangi *yol* takip edilerek bulunabileceği belirtilir.
-d <directory>	: derleyicinin derleme sonrası oluşturduğu *.class* uzantılı kütükleri nereye yerleştireceğini belirtir.
-J<runtime flag>	: Java yorumlayıcısına gerekli argümanları aktarmada kullanılır.

Java 2 platformu ile gelen yeni *rmic* derleyicisine ilkine nazaran 3 yeni opsiyon eklenmiştir:

-v1.1	: JDK 1.1 'in kütük(stub) protokolüne uygun kütük (stub) ve iskeletler(skeleton) oluşturmak için kullanılır.
-vcompat	: Hem JDK 1.1 'in hem de Java 2 'nin kütük(stub) protokollerine uygun kütük(stub) ve iskeletler (skeleton) oluşturmak için kullanılır.
- v1.2	: Sadece Java 2 'nin kütük(stub) protokolüne uygun kütük(stub) ve iskeletler(skeleton) oluşturmak için kullanılır.

8.5. İstemci Tarafını Oluşturma

Şimdi sıra uzak nesneleri kullanacak olan istemci tarafı uygulamasını oluşturmaya geldi. Aşağıda verilmiş olan *hesapIstemci.java* kaynak kütüğü, istemci tarafı uygulamasını zihinlerde somutlaştırma amacıyla verilmiştir.

```
/*****************************************************
                        hesapIstemci.java
 *****************************************************/
import java.util.*;
import java.net.*;
import java.rmi.*;
import java.rmi.RMISecurityManager;

// uzak arayüz sınıfının sağladığı uzak metotları çağırarak fiyat listesi isteminde
// bulunacak olan istemci sınıfı
public class hesapIstemci
{
    public static void main( String args[] )
    {
        // gerekli görülen yerel değişken tanımları
        Hesapla hs = null;
        fiyatListesi fliste = null;
        Float miktar = 0;
        int sure = 0;
        String sunucuAdi = "www.cs.hacettepe.edu.tr";

        System.setSecurityManager( new RMISecurityManager());
        if( args.length >= 1 ) // eğer bir sunucu adı argüman olarak girilmişse
          sunucuAdi = args[0]; // girilen sunucu adını sakla

        try
        {
          System.out.println("hesapIstemci baslatiliyor...");

          // eldeki sunucu adını ve tanımlı uzak servis adını kullanarak url oluştur
          String url = new String( "//"+ sunucuAdi + "/HesapServisi");
          System.out.println("hesapSunucu Lookup: url =" + url);
          // url bilgisini kullanarak uzak nesneye erişmeye çalış
          hs = (Hesapla)Naming.lookup( url );

          if( hs != null ) // aranan uzak nesne bulunduysa,
          {
            // Sunucudan aldığın şu anki ilgi oranı bilgisini görüntüle
            hs.ilgiOraniGoster();

            // Sunucudan alınan ilgi oranını kullanarak amortisman fiyat listesini hesapla
            miktar   = (float)3600.60;
            sure = 23;
            fliste  = hs.amortismanFiyatListesi( miktar, sure );
```

```
            // Hesaplanan fiyat listesini görüntüle
            fliste.goster();
         }
         else // uzak nesne bulunamadıysa, ilgili uyarı mesajını görüntüle
         {
             System.out.println("Istekte bulunulan uzak nesne bulunamadi!!!");
         }
      }
      catch( Exception exc ) // herhangi bir hata oluşursa yakala ve
      {
          System.out.println("İstemci: Hata olustu!!!"); // uyarı mesajını görüntüle
          exc.printStackTrace();
          System.out.println(exc.getMessage());
       }
    }
}
```

İstemci için yazılan kod parçasında, *java.rmi* paketi ile bu pakette yer alan *RMISecurityManager* sınıfının kullanılacağı belirtilir. İstemcinin bu noktada yapması gereken ilk icraat, sistem ile alakalı olarak bir güvenlik yöneticisinin(*security manager*) kaydını gerçekleştirmektir. *RMI* paketi kullanıcılarına güvenlik yöneticisi hizmetini sağlamakla birlikte kod geliştiricileri isterlerse kendilerine ait güvenlik yöneticisini oluşturup kaydını yapabilirler. Eğer sistemle alakalı olarak bir güvenlik yöneticisinin kaydı yapılmamışsa, Java sadece yerel sınıfların isteklerine izin verecektir ve bu da dolayısıyla dağıtık işlemin amacına ters düşecektir.

Eğer istemci için Java uygulaması yerine bir Java *applet* kodlayacaksanız, tarayıcınız tarafından önceden sizin için bir güvenlik yöneticisinin kayıt işlemi yapılacaktır. Dolayısıyla kodladığınız *applet* için ayrı bir güvenlik yöneticisine gerek kalmayacaktır.

Güvenlik yöneticisinin kayıt edilmesinin ardından, sunucu adından ve istekte bulunulan uzak nesne adından oluşan parametre bilgisine ihtiyaç duyulacaktır. Bu istemcinin *rmiregistry* vasıtasıyla uzaktaki nesneyi aramasını sağlayacaktır. İstemci *Naming* sınıfı dahilindeki *lookup()* metodunu çağırarak, sunucudan uzak nesne referansını döndürmesi için istekte bulunmuş olur. *Naming* sınıfının *lookup()* metodu aracılığıyla döndürülen nesne, uzaktaki gerçek arayüzün kalıbı olarak görülebilir.

Yukarıda bahsedilen parametre bilgisinin yapısı aşağıda verilen örnektekine benzemektedir:

rmi://www.cs.hacettepe.edu.tr/uzakNesne
ya da
// www.cs.hacettepe.edu.tr/uzakNesne

Eğer istemci uzak nesne referansını başarıyla almışsa, bu noktada artık elde ettiği nesnenin metotlarını çağırabilir. Örneğimize dönecek olursak, istemci ilgi oranını görüntülemesi çağrısını yapar ve amortisman fiyat listesi için istekte bulnur. Eğer uzak arayüz gerçekleştirim sınıfı başarılı olursa, istemci fiyat listesi nesnesinin yerel bir kopyasını elde etmiş olur. Bu aşamadan itibaren elde ettiği fiyat listesi nesnesinin metotlarını çağırabilir ve hatta bu nesne üzerinde değişiklikte bulunabilir. Ayrıca buradaki yerel nesne, istemciye ait özel bir kopyadır ve sunucunun nesne üzerinde yapılacak değişikliklerden haberi olmaz.

8.6. Sunucu Tarafını Oluşturma

Aşağıda verilmiş olan *hesapSunucu.java* kaynak kütüğünün istemci için verilen kaynak kütüğe benzediği fark edilebilir.

```
/**************************************************
                    hesapSunucu.java
 **************************************************/
import java.util.*;
import java.rmi.*;
import java.rmi.RMISecurityManager;

public class hesapSunucu
{
   public static void main( String args[] )
   {
      System.setSecurityManager( new RMISecurityManager());
      try
      {
          System.out.println("hesapSunucu baslatiliyor…");
          // uzak nesneyi 4.5 ilgi oranı değeri ile yarat
          HesaplaImp hsi = new HesaplaImp( 4.5 );
          // ve yaratılan uzak nesneyi HesapServisi adı altında kayıt işlemini tamamla
          Naming.rebind("HesapServisi", hsi );
          System.out.println("Sunucu istemleri bekliyor…");
       }
```

```
        catch( Exception exc ) // herhangi bir hata oluşursa yakala ve
        {
            System.out.println("Sunucu: Hata olustu!!!"); // uyarı mesajını görüntüle
            exc.printStackTrace();
            System.out.println(exc.getMessage());
         }
    }
}
```

Sunucu da istemci gibi güvenlik yöneticisini dikkate almaktadır. Sunucu tam anlamıyla bir güvenlik yöneticisi ile kayıt olduktan sonra, sunucu uzak arayüzün gerçekleştirim sınıfı olan *HesaplaImp* nesnesinin bir örneğini yaratır. İşte, sunucunun aktardığı hakiki nesne gerçekleştirim nesnesidir. Sunucu *rmiregistry* kullandığı için gerçekleştirim nesnesinin örneği belirli bir adla ilişkilendirilerek ileride istemcinin bu ad için arama yapması sağlanmış olur. İstemci sunucudan referans isteminde bulunduğu zaman sunucu uzak nesnenin güncel olmasa da var olan referansını gönderir. İstemci ise elde ettiği güncel olmayan referansı kullanmaya kalktığında, sunucu bir aykırı durumu tespit eder (*çünkü referansı elde edilen nesne doğru nesne değildir*). Bu durumu kontrol altında tutmak istersek *bind( )* metodu yerine *rebind()* kullanılabilir.Bu yolla her defasında yeni bir sunucu hizmeti başlatılır ve *lookup()* metodu ile güncel nesnenin elde edilmesi sağlanmış olur.

Bu *RMI* sisteminde istediğimiz kadar nesne aktarabiliriz, fakat anlaşılırlığın sağlanması açısında verilen örnekte yalnızca bir nesne aktarılmaktadır. Ek olarak, bu tasarımda bir fabrika sınıfı (*factory class*) kullanarak uzak nesnelerin referanslarını döndürmek mümkündür. Normalde sadece bir kayıtçıya ihtiyaç duyulmasına karşın, Java farlı portlar üzerinde faaliyet gösteren birden çok kayıtçıya izin vermektedir. Bu durumda istemci ilgili port üzerindeki doğru kayıtçıya ulaşabilmek için doğru *lookup()* metodunu kullanması yeterlidir.

8.7. Sistemi Yükle ve Başlat

Öncelikle istemci ve sunucu kodlarının derlenmesi gerekir ve bunun için kod kütüklerinin bulunduğu dizine kadar komut satırında ilerlenir ve java derleyicisi(javac) aşağıda verilen parametrelerle kullanılarak istemci ve sunucu için .class uzantılı kütükler elde edilmelidir.

```
> javac hesapIstemci.java
> javac hesapSunucu.java
```

Yukarıdaki işlemlerin ardından belirtilen dizin altında istemci ve sunucu için sırasıyla hesapIstemci.class ve hesapSunucu.class kütüklerinin oluşturulduğu gözlemlenecektir.

Şu noktada tarif edilen tüm adımların başarıyla tamamlandığı kabul edilirse artık basit RMI uygulamamızı çalıştırmaya geçebiliriz. Bunun için ilk olarak *RMI* kayıtçısının(*rmiregistry*) sunucu tarafında başlatılması gerekir. Ama öncelikle kayıtçının(*registry*) size ait sunucu sınıflarını ait oldukları dizinde bulabilmesi için *CLASSPATH* değişkeninin kurulu olduğundan emin olunmalıdır. Aşağıda verildiği biçimiyle *rmiregistry* kullanılırsa, önceden tanımlanmış olan 1099 nolu port kullanılacaktır. Eğer farklı bir port numarası üzerinde çalışılmak isteniyorsa bu port numarası belirtilmelidir:

```
> rmiregistry
```

ya da

```
> rmiregistry 1092
```

Şimdi sıra istemcilerden gelecek olan istemlere karşılık vermek amacıyla sunucunun çalışır hale getirilmesine geldi. Aşağıda gösterildiği biçimiyle sunucuyu çalışır hale getirmek mümkündür:

```
> java hesapSunucu
```

Sunucunun çalışmaya başlamasının ardından sunucunun istemcilerden gelecek olan istemleri beklediğini, konsolda görüntülenecek

```
hesapSunucu baslatiliyor...
Sunucu istemleri bekliyor...
```

mesajları vasıtasıyla anlayabiliriz.

Son adım olarak başlatılan sunucu üzerindeki istenilen servisler için hizmet alabilmek için istemcinin başlatılması gerekir. Aşağıda gösterildiği biçimiyle istemciyi çalışır hale getirmek mümkündür:

```
> java hesapIstemci www.cs.hacettepe.edu.tr
```

İstemcinin çalışmaya başlamasının ardından sunucuya *ilgiOranı* 'nı görüntülemesi ve uzak nesne referansını göndermesi için istemci istemde bulunacaktır. İstemine yanıt alan istemci ise sunucudan gönderilen *fiyatListesi* nesnesinin içeriğini görüntüler.

9. Alternatif Gerçekleştirimler

Bu doküman sadece *RMI* mimarisini ve bu mimari için *Sun* firmasının gerçekleştirimini içermektedir. Doğal olarak bilişim dünyasında farklı gerçekleştirimler de mevcuttur:

- NinjaRMI
  University of California'da inşa edilmiş olan bu gerçekleştirimde RMI 'nın JDK 1.1 sürümü yapılan ilavelerle birlikte desteklenmektedir.
- BEA Weblogic Server
  Bu gerçekleştirim RMI, Microsoft COM, CORBA ve EJB (*Enterprise JavaBeans*) gibi birçok servisi destekleyen yüksek performansa sahip bir uygulama sunucusudur.
- Voyager
  ObjectSpace firmasının bir ürünü olan bu gerçekleştirimde, *RMI* ile birlikte patentli olan DOM, CORBA, EJB, Microsoft DCOM ve işlem servisleri(*transaction services*) net bir şekilde desteklenmektedir.

10. Yararlanılan Kaynaklar

- **http://bornova.ege.edu.tr/~erdur/SYT_2003_2004_Ders01.ppt**
- **http://bornova.ege.edu.tr/~ogurcan/courses/2004_2005/fall/JavaRMI.pdf**
- **http://www.edm2.com/0601/rmi1.html**
- **http://java.sun.com/developer/onlineTraining/rmi/RMI.html**
- **http://www.di.unipi.it/~giangi/CORSI/LPRA/LECTURES/JRMI2.pdf**
- **ftp://ftp.cs.hun.edu.tr/pub/dersler/BIL3XX/BIL341_YL-I/0X-04/2003/exp3/**