# = DAĞITIK SİSTEMLER =

Mantık Process'i : Aynı anda birden fazla işlem yapmak.

→ Paralel Hesaplama'da; Veriyi parçala → Parçaları farklı yapılarda işle → Sonuçları topla → Birleştir.

→ Paralel Hesaplama'nın gelişimi:

1-) İlk olarak işlemler farklı makinelerde yapıldı.

2-) Daha sonra paylaşımlı hafıza ile birden fazla CPU ile yapıldı.

→ Paylaşımlı hafızada paralel hesaplamanın avantajı ; Veriyi bölmüyoruz. Zaman kaybı az. Haberleşme iyi.

ÖRNEK: Anakart üzerinde 4 CPU 1 RAM olsun. İşlem RAM'le yüklenir. CPU'ların hepsi bunu görür, RAM'e giderek okurlar işlemi gerçekleştirirler. (Veri bölünmez)

→ Bir PC'de 4 CPU olsun RAM'in hafızası sınırlıdır. Çok büyük problemlerin çözümü için daha fazla hafızaya ihtiyaç vardır. Bu sebeple farklı bilgisayarlar kullanılır. Veri bölünerek PC'ler arasında paylaştırılır. Yani paylaşımlı hafıza yok. PC'ler, birbirlerinin hafızalarını göremediğinden, bir network ile aralarında bağlantı kurulur. Buna da dağıtık sistem denir.

→ Dağıtık hesaplama; birbirinden coğrafi olarak farklı yerlerde bulunan bilgisayarların büyük hesaplamalar yapmasıdır.

→ Dağıtık hesaplamada karşılaşılması beklenen problemler ;

1. Ağ problemi (Domain farklılığı)
2. ABD - Japonya bilgisayarı nasıl iletişim kuracak?
3. Hangi protokol kullanılacak?
4. Güvenlik nasıl sağlanacak?
5. Senkronizasyon nasıl sağlanacak?
6. Bilgisayar saati farklılıkları, Yerel saat farklılıkları.

→ Dağıtık sistemler bu problemlerin birçoğunu çözdü.

ÖRNEK: İnternet.

→ Dağıtık sistemlerin ütopik amacı; global bir işletim sistemi yapmak.

Dağıtık sistemlerin şuanki amacı ; dağıtık olan hesaplama makinelerini insanların etkin bir şekilde kullanmasını sağlamak.

→ Dağıtık sistemlerin daha gelişmişi ; Cloud computing = Bulut hesaplamadır.

→ Web Service : Yazılımı başkasına kullandırmak istiyorsak, bunu hizmet olarak sunmalıyız. Web service çalışma mantığı;

Web service tanımlama dökümanı → hangisini kullanmak istediğini seç → Parametre yollar → İşlem yapılır → Sonuç elde edilir

→ Bulut Bilişim'in görevi ; yazılımsal ve donanımsal kaynakları web servis olarak kullanmak (hızlı bir şekilde başlatma, durdurma, öldürme ...)

→ Sanallaştırma : Bir PC'de birden fazla PC'nin kurulması ve kullanılmasıdır. PC içine sanal makine (virtual box, vmware) kurulur ve bunlara da farklı OS'ler yüklenir. ( Linux-Ubuntu gibi)

→ Windows'un kullandığı Uzak Masaüstü Bağlantısı. Örneğin Ubuntu'da ssh ubuntu@10.47.7.6 komutu ile IP'si verilen bilgisayarın masaüstünü getirebiliriz.

## = PARALEL HESAPLAMA =

→ Sistemde tek CPU varsa, bir tane instruction vardır. Sadece multiprocessing yapılabilir. (ya da multithreading = multiprogramming)

→ Paralel Hesaplama için birden fazla CPU olmalı. Her CPU'da bir instruction çalışır. Böylece parçaları işlenen problem, eş zamanlı olarak çözülür.

→ Mevcut Sorun; Bir problemi çözebilmek için varolan kaynakların sınırlı, ihtiyaçların sınırsız olması. Bu sorunun çözümünü paralel hesaplama yaparak sağlanmaya çalışılmıştır.

→ Bir işin paralel bir şekilde çözülebilmesi için:

1. Problem bölünebilmelidir. Birbirini etkilemeyen bağımsız parçalar haline getirilebilir olmalıdır.

2. Instructionların eş zamanlı çalışacağı birden çok ortam olmalıdır. (Birden çok CPU)

→ Aynı anda hesaplama yaptırma olayı : Paralel Hesaplama.

→ Paralel hesaplama'da, bir sonraki adım, bir önceki adıma bağlıysa, problem paralelleştirilemez.

## Paralel Hesaplama Nerelerde Kullanılır?

1-) Atmosphere, Earth, Environment.

2-) Sensörlerden tüm verilerin toplanıp tahminler yapılması Ör: Okyanus üzerinden yarım saat içinde bir hava hadisesi olacaksa, bunun tahmin edilebilmesi. Ör: Yine deprem sensörlerinin sürekli takip edilebilmesi, işlenmesi ve tahminlerin yapılması.

3-) Nükleer enerji simülasyonları (atom fiziği, quantum fiziği)

4-) Photoshop programları Ör: Bank adama

5-) Makine öğrenmesi, petrol aramaları, veri tabanları, veri madenciliği, web search...

6-) Hastalık teşhisi.

## Paralel Hesaplama Niçin Kullanılır?

1-) Zaman ve paradan tasarruf etmek için. Çünkü kaynaklar pahalı ve sınırlıdır. Teorik olarak çok kaynak = çok hız olduğundan bu tasarruf önemli.

2-) Web Search yapmak için.

3-) Daha büyük problemleri çözmek için.

4-) Eş zamanlı çalışabilirliği sağlamak için (Concurrency)

5) Elde olmayan kaynakları kullanabilmek için. Problemi hem burada hem de başka bir PC'de çözülmesi için, başka PC'ye gönderme anlayışı.

NOT Dünya dışı varlıkları araştıran proje: SETI. Dünya'nın farklı yerlerindeki teleskoplar, sürekli uzayı taramaktadır. Yeni oluşumlar, yer değiştirmeler... ile ilgili veriler sürekli gelmekte. Bu verilerin işlenmesi gerekmektedir. Bu yüzden üniversite bir program yazıyor. Bu program PC'a kurulduğunda, sen o PC'i kullanmadığında devreye giriyor ve bu proje için çalışıyor. (Ekran koruyucusu değil de bu program çalışıyor)

## Seri Hesaplama Neden Tercih Edilmiyor?

1-) Elde yeteri kadar çok PC yok.

2-) Transmission Speeds: İletim hızı düşük (RAM'den okuma hızı ... düşük)

3-) Küçültmedeki limit. Bir board'a 60 nm'den daha küçük birim koyulamaz.

4-) Teknolojinin getirdiği kısıtlamalar: Diskin dönüş hızı, okuma-yazma hızı...

5-) Ekonominin " " : 10.000 $'lık CPU alıp evde gerçekleştiremeyiz.

→ Yeni teknoloji zengin ülkelerde olur. ABD ordusu.

→ Hesaplama gücü fazlaysa işlem yapma hızı da yüksek.

# Paralel Hesaplama Nasıl Yapılır?

1-) En basit şekilde : 1 Board üzerinde 4 CPU

2-) Komutları pipeline işletmek

3-) Multicore (çok çekirdekli) CPU'lar (Birden fazla hesaplama ünitesi)

## VON NEUMAN MİMARİSİ

→ Bir PC'de olması gereken parçaları tanımlamıştır. 4 temel ünite vardır. Hafıza, Control, ALU, I/O.

- hafıza : memory. kullanıcı, veri, instruction, OS'i tutar.
- control : kullanıcı ve hafızasındakini eşleştirip kontrol eder.
- ALU : hesaplama.
- I/O : kullanıcı ile etkileşim.

→ bu mimari Seri mimaridir.

→ ALU aynı anda sadece bir işlem yapar.

## FLYNN'IN SINIFLANDIRMASI

→ Paralel hesaplamayı sınıflandırmada iki boyut vardır. 1. data 2. instruction

→ Bunlardan oluşan 4 farklı kombinasyon:

1. SISD : Single Instruction Single Data
2. SIMD : " " Multiple "
3. MISD : Multiple " Single "
4. MIMD : " " Multiple "

### 1-) SISD :

1 komut CPU'ya alınır → 1 veri üzerinde işlem yapılır → Sonuç tahmin edilebilir.

✓ Paralel olmayan, sıralı, 1 CPU.
✓ Bildiğimiz klasik PC

Ör : Laptop

### 2-) SIMD :

1 komut çok CPU'ya alınır → tüm CPU'lar aynı anda aynı komutu işlerler. → Farklı veri işlenir.

✓ Data seviyesinde paralelleştirme.
✓ Birden fazla CPU ~yerel hafızalar
✓ Yaygın kullanılmazlar.

Ör : GPU.

3-) MISD : 1 den fazla CPU → Veriyi → Birden fazla → Her CPU
1 veri al CPU'da işle. farklı işlem yapacak.

✓ Kullanım alanı pratikte olmadığından çok kullanılmamış.

✓ Çok sayıda işlemci = instruction

ÖR: kriptografi, Filtreleme (Ses)

4-) MIMD : 1 den fazla CPU → Veriyi → Her bir kısmını
1 den fazla veri parçalara bir CPU'da
böl. çözümle.

✓ Her process bağımsız çalışır.

✓ Senkron-asenkron çalışabilir. İşlemin ne zaman bitecği belli değildir.

✓ Cluster'lar bunu kullanır.

ÖR: Supercomputer.

---

Paralel Hesaplama'daki Terimler — 07.10.2013

Task: Hesaplama yaparken birbirinden bağımsız çözülecek kısımlara (ayrık parçalara) task denir.

NOT Paralel Hesaplama'da en önemli şey: Paylaşımlı hafızadır. Bütün processorler aynı hafızayı kullanır. Ara sonuçlar hafızaya yazıldığında 2. processor hızlı olmalıdır. Dezavantaj: makinenin mimarisinin karmaşıklaşıyor olmasıdır. Çok çekirdekli yapının yaygınlaşmasıyla Paylaşımlı hafıza mimarisi de yaygınlaşmıştır

Shared Memory: Makine üzerinde tüm CPU'ların hafızanın her yerini okuyup yazabilmesidir.

SMP: Simetric Multi Processor: Tek adres uzayı, pek çok işlemci, hafıza uzayını birden çok kişi.

Distributed Memory: Birden fazla CPU 1 Board. Network ile CPU'lar birbirine bağlanır. Ortak bir iş olduğunda birbirlerinin hafızalarını kullanırlar. Network bağlantısı şart.

NOT Paralel hesaplama yapan parçalar birbiriyle haberleşmek. Haberleşme yolları: 1. Paylaşımlı hafıza, 2. Communications.

Communications: Normalde hafıza sadece kendi lokalidir. Hafızayı başka makinelere dahil etme. Zaman kaybının en fazla olduğu yerdir. En iyi durumda işlemin başında ve sonunda haberleşme gerektirir. Haberleşme maliyeti, Paralel Hesaplamadan yüksekse Paralel Hesaplama yapılmaz.

Senkronizasyon: Eş zamanlı çalışan parçalar senkronize olmalıdır. (Break pointler.)

**Granularity:** Hesaplama / Haberleşme

Hesaplama ↑, Haberleşme ↓ ise Coarse büyük parçalı; üstünkörü.
Hesaplama ↓, Haberleşme ↑ ise Fine küçük " ; hassas.

**Observed Speedup:** Hızlanma miktarı = Seri haberleşme / Paralel haberleşme

**Paralel Overhead:** Zaman olarak ekstra yükler = kayıp zaman.

→ Haberleşme zamanı: kayıp zaman, geri kazanılmaz.

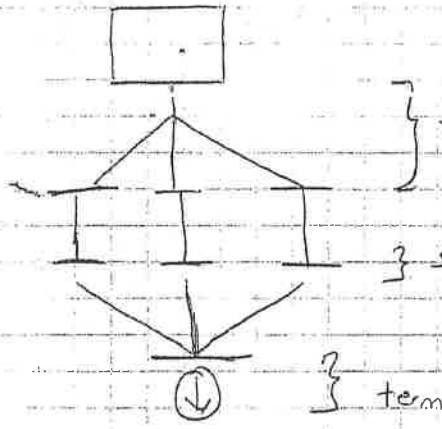
→ Task startup time: başlatma zamanı. İlk iterasyonda çok yükseldir. Daha sonra hazır kullanacağından kısalır. Kaçınılmaz zaman kaybı.

→ Senkronizasyon: 3 parça işi erken bitirir. 1 parça geç. Onu beklemek zorundasın ve bu seri yavaşlatır.

→ Veri haberleşmesi zamanı: Değişkenlerin ara değerlerini birbirine aktarırken gerçekleşen kayıplar.

→ Compiler, Operatörlerden gelen kayıplar.

→ Kütüphane, Tools, OS'den gelen "




→ Termination time: Sonucun alınıp yazılmasına kadar olan zaman kaybı.

**Massively Paralel** Büyük derecede paralelleştirme.

**Embarrasingly Paralel** Overhead yok. Sayısı çok değil.

**Scability:** Ölçeklenebilirlik. İhtiyaca göre büyütülüp / küçültmek.

Web Servicelerde dinamik. Sete çökme olayı. Gelen isteklerin sayısına göre kendini ayarlamak.
Donanımsal kaynakları artırınca Paralel Hesaplamanın hızı artıyorsa bu ölçeklenebilirliktir.

## Multicore Processors:

**Cluster Computing:** Herkesin satın alabileceği çok sayıda bilgisayarı bir araya getirerek paralel hesaplama yapmak.

**Supercomputing = High Performance Computing:** Çok büyük kaynakları olan bilgisayarlardır. RAM'ler TB, PB oluyor. Paralel, yüksek özellikli ve çok fazla sayıda donanımsal parçalara sahip. Büyük problemler için kullanılır. Supercomputing hızı saniyede yaptığı floating point işlem sayısı ile ölçülür. En büyük, yüksek hızlı supercomputer Çin'de dir.
ÖRNEK: Tianhe-2

## PARALEL HAFIZA MİMARİLERİ

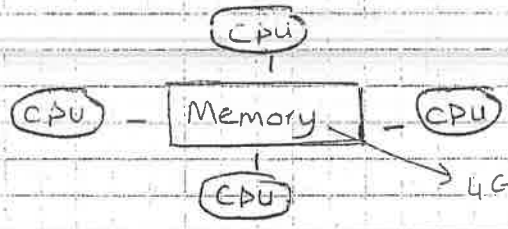
### 1-) PAYLAŞIMLI HAFIZA MİMARİSİ (SHARED MEMORY):

→ Bir CPU değişiklik yaptığında, diğer CPU'lar bundan haberdar oluyor.

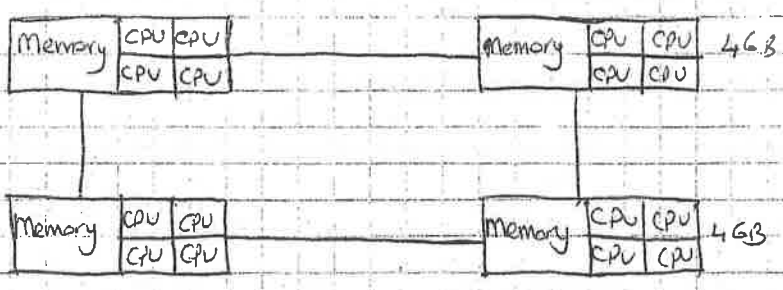
→ Tüm işlemciler paylaşımlı hafızaya ulaşabilir.

a) **Uniform Memory Access = UMA:** CPU özellikleri aynı. Bu CPU'ların hafızaya erişim süreleri aynı.

→ **Cache coherency:** Bir işlemcinin paylaşımlı hafıza bölgesini update ettiği durumda diğer CPU'ların bundan haberdar olmasıdır. Bu da tutarlılık sağlar.




b) **Nonuniform Memory Access = NUMA:** 4 farklı makine olsun. Her makine bir tane hafıza var. Bu 4 makineyi bir bus ile bağlayarak RAM ve CPU miktarını artırıyoruz.
Her makine kendi local hafızasına hızlı erişirken, diğerlerinin RAM'ine daha yavaş erişmektedir.




RAM miktarı: 16 GB. (4 katı)

CPU miktarı: 16. (4 katı)

**UMA - NUMA Arasındaki Fark:** UMA'da CPU'ların türü, yazma-okuma hızları, buslar... aynıdır. NUMA'da birbirinden bağımsız makineler buslarla birbirlerine bağlıdır. Kendi hafızalarına farklı hızlarla erişebilmektedirler, diğer makinelerin hafızalarına erişim süreleri de farklıdır. Kendi hafızaları ile birlikte diğer makinelerin hafızalarını da adresleyebilirler. Bu sebeple RAM miktarı artırılmış olur.

→ Cache Coherency gerçekleşiyorsa buna CCNUMA ya da CCUMA denir.

→ Her değişikliği CPU'ya bildirmek NUMA'da zordur. (16 tane CPU var) Bu yüzden NUMA, cache coherency'nin sağlanmasının gerekli olmadığı durumlarda kullanılır. Yani bir değişiklik başka verileri etkilemeyen durumlarda.

→ Değişikliğin bildirilmesi: Donanımın ortak kullanımını sağlayan CPU'lara bildirilmesi demektir. Ya da CPU'nun belli cycle aralığında ortak kullanılan bellek değeri kontrol etmesi gerekmektedir. Bu da NUMA'da çok maliyetlidir.

## NUMA Avantajları:

✓ RAM miktarı artar
✓ Adres uzayı artar
✓ Veri paylaşımı yüksektir
✓ Çok büyük problemleri parçalara böler ve kısa sürede çözümler.
✓ Aradaki bus hızlıdır.

## NUMA Dezavantajları:

✓ Ölçeklenebilirlik zor.
✓ Makinenin local RAM yönetimi kolay, local olmayanların yönetimi zor.
✓ Makine sayısı arttıkça işlemler karmaşıklaşır, ölçeklemek zorlaşır.
✓ CPU yönetimi zor.
✓ Bunlar için program yazmak zor.

# 2-) DAĞITIK HAFIZA MİMARİSİ (DISTRIBUTED MEMORY):

→ Her processor kendi local hafızasını kullanabilir, diğer makinelerin hafızasına erişemez.

→ Global adres uzayı yoktur.

→ Bir local değişikliği diğerlerine bildirmeye gerek yoktur (cache coherency yok)

→ Bir CPU, diğerinin bilgisine erişmek istediğinde bus aracılığıyla istekte bulunur. İstenen veri bus'a bırakılır. İsteyen veriyi bustan alır.

→ İletişim ortamı Ethernet, İnternet, Fiber ... olabilir.

## Avantaj:

✓ Ciddi anlamda ölçeklenebilir.
✓ Makine sayısı arttıkça hesaplama sayısı artar.
✓ Bus hızı, iletişim ve hesaplama hızını belirler.
✓ Maliyeti düşüktür.

## Dezavantaj:

✓ Veri haberleşmesi ile ilgili işleri programcının düşünüp yazması gerekir.
✓ Global değişkenlerin senkronizasyonu zordur.
✓ Bir global değişkende, değişiklik yapıldığında diğerlerine bildirmek gerekir. Cache coherency olmadığından bu zahmetlidir. Bu da ekstra yük anlamına gelir.

3-) HİBRİT DISTRIBUTED : (SMP + Distributed)

→ Dağıtık hafıza kullanarak çok hızlı işlem yapmak.

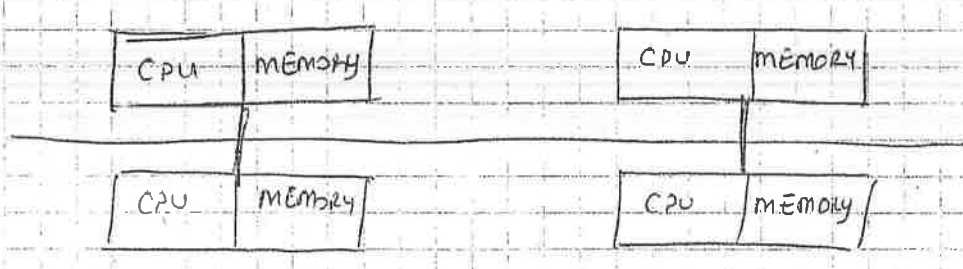
→ İki hafıza mimarisinin olumlu yönleri birleştirilmiştir.

→ Her bir shared kendi RAM ve CPU'sunu başkasına kullandırabiliyor.

→ NUMA'dan farkı; NUMA'da ortak hafıza var, burada yok..

→ Hızlı dağıtık ile ölçeklenebilir.

NOT DISTRIBUTED MEMORY'NİN ŞEKLİ




Global değişkenler sık kullanılıyorsa bus yoğun olur. Kapasite aşılır ve başka bir işlem bus. yapamazsın.

Hibrit'in Avantajı: Her bir makine, diğerinin hafıza adres uzayını tutmuyor.

" Dezavantajı: Bir makineye ulaşmak için hepsi uyarılır. Bus gereksiz kalabalık.

21.10.2013

PARALEL HESAPLAMA - 2

→ Paralel Hesaplama C ile yapılır. Sebebi: Önemli ölçüde makine diline yaklaşmanın gerekliliğidir.

→ Paralel Hesaplamada, hafıza bölgesinin adresini direkt verip veriye ulaşabiliyoruz.

Paralel Hesaplama Modelleri

1. Paylaşımlı Hafıza Modeli
2. Message Passing Modeli
3. Data Paralel Model
4. Thread Modeli
5. Hibrid Modeli

**Data Paralel Model :** Veriyi paralelleştireceksek önce problemin bağımsız alt problemlere bölünüp bölünmediğine bakarız. Eğer ayrılabiliyorsa veri paralelleştirme yapanız.

**Paylaşımlı Hafıza Modeli :**
- Aynı Hafıza kullanımı
- Program yazmak kolay.
- Klasik programlamadan farkı paylaşımın olması.

Problem parçalara bölünür → Her parçayı 1 CPU yönetir → Her bir parçaya ötekı CPU'lar erişebilir. → Bir değişiklik yaptığında ötekı CPU'lar haberdar olur.

- Semafor, kilit kullanılarak paylaşılan veri korunur.
- Yapılan modeller; paylaşımlı yaptığın problemi derleyen derleyiciler.

**ÖRNEK :** Sunun derleyicisi intel'de çalışmaz. Çünkü mimarileri ve adreslemeleri farklıdır.
- Lock ve Semaphore kullanımı da donanımla ilgilidir.

**Thread Model :**
- Process'lerin global adres uzayları kullanılır.
- Kendisine ait hafızası vardır.
- Thread, Processle bilgisayar gibi bakar.
- Paralel Hesaplamayı mimariden (donanımdan) ayırır.
- Processin içinde çalışır, o mimaride tek process gibi görünür.
- Thread = Process parçacığı.
- Thread oluşturma, process oluşturmadan hızlıdır.
- Process için PCB oluşturulur, thread için genel yok.
- Çok process büyük kayıp, çok thread büyük hız.
- Threadler, processlerin global adres uzayında haberleşir. Böylece kayıp zaman yoktur. Avantajı: threadler arası haberleşmeyi programcı düşünmez.

**ÖRNEK:** POSIX threadli. C'de çalışır. (Eski)
C'de thread yazmak zor, Javada kolay.

**ÖRNEK:** Open MP, Open Message Passing: C, C++'ta çalışır.

**NOT** Thread paralel hesaplama sayılmaz.

**Message Passing Modeli** Yaygın kullanılır.
- Dağıtık mimarileri ve farklı CPU'ları var.

Problem parçalara bölünür → Parçalar CPU'ya gönderilir → CPU'da sonuçlar hesaplanır → Sonuçlar toplanır.

- Problem haberleşme
- İster shared memory kullan, ister message passing, Lokal memory'de problem yok. Birbirleriyle haberleşmede problem var.
- Mesaja ne istediğini yaz yolla, nereye yollayacağı adresi bilmiyor.
- Bunun yapılabilmesi için her iki CPU da message passing olmalı.
- Veri gönderildiğinde karşıda alıcı olmalı. Hatta dinlemeli, anında cevap vermeli.

**Problem :** Programcı message passing ile ilgili şeyleri bilmeli.
Başka makinenin adres uzayı sözkonusu olduğu için karmaşık yapı.

**ÖRNEK** MPI = message Passing Interface
open MPI = C üzerinde herkes kullanır.

# Paralel Programlama Modelleri

30.10.2013

1-) Paylaşımlı Hafıza 2-) Mesajlaşma = Temel olanlar

**Mesajlaşma Modeli:** ← Message passing

→ Bağımsız bilgisayarların her birinin CPU ve hafızaları var. Hepsi kendi işini yapıyor. Aynı anda birçok iş yapılmakta. Veri paylaşımı gerektiğinde CPU'ların lokal hafızaları var, işlemler burada yapıyorlar ara sonuçların haberinin verilmesinde de mesaj gönderiyorlar.

→ Mesaj gönderme yöntemleri

1- Ortak bus varsa bunun üzerinden.

2- Yoksa network üzerinden.

→ Paylaşımlı hafızada veriyi yazma-okuma nanosaniyeler mertebesinde, mesajlaşmalı milisaniyeler.

→ Mesajlaşma modeli hız gerektiren işlemlerde kullanılmaz, çok büyük işlemlerde kullanılır.

→ Paylaşımlı modelde özel PC'ye (donanımsal altyapıya) ihtiyaç var. Bu modelde yok.

→ Mesajlaşma modeli en popüler paralel programlama modelidir.

→ Sıradan PC üzerine kurulur, C, Fortran gibi dillerde rahatça kullanılabilir.

→ Fizik, nükleer enerji simülasyonlarında kullanılır.

→ Program yazılır → döngü paralel hale getirilir → mesajlaşma C, Fortran kütüphaneleri ile gerçekleştirilir.

→ En popüler message passing: MPI.

→ Data açısından veriyi paralelleştirme:

* Paralel hesaplamanın önemli bir kısmı veri üzerinde gerçekleşir.
* Veriyi böl. → eş zamanlı olarak paralel işle. * Verinin paralelleştirilmesinde, veriyi birbirinden bağımsız işler haline getirmek önemli.

→ Verinin paralelleştirilmesinde;

Veri kapsamlı olarak incelenir. Bir parçayı bekleyen parçalar varsa bu paralelleştirilemez.

ÖRNEK: 1, 2, 3, ..., 999, 1000 elemanlarından oluşan bir dizinin her bir elemanına 4 eklemek istiyoruz. Bunu: 1-25 arasını işle, 26-50 arasını işle, ..., 961-1000 arası. Böylece döngü parçalanmış olur. Program aşağıdaki gibidir.

```
do i=1,25                    do i=26,50
A(i) = B(i) * delta ,        A(i) = B(i) * delta ...
end do                       end do
```

Bunu elle yapacak dursam thread'le yapmış olurum. MPI ile yaparsam; do i=1,25 satırının yanına PF (paralel for) (paraleldir) şeklinde bir işaret koyarsak, o onlar ve yapar. (C'de)

Bu programın paralel kısmı bu döngüdür.

Paralelleştirme

Otomatik — Manuel

Problem nitelikleri çok farklı olduğundan önemli oranda limitlidir.

Bazısı çok paralelleştirilebilir, bazısı az.

Buna geliştirici karar verir.

→ Haberleşmede CPU iş yapmaz, yatar → kayıp
→ Taskların birbirini beklemesi → kayıp
→ Haberleşme bus'ı çok çalıştırır. Çok fazla mesajlaşma. Bir süre sonra bus'ın önemli bir kısmı haberleşmeye ayrılıyorsak program kilitlenir. Geri kalan kısım veri işleme için yeterli olmaz.

**Latency**: Gecikme.

**Bant Genişliği**: Birim zamanda gönderilen bit miktarı/veri. Çok fazla haberleşme Bant genişliğini doldurur, Latency'ye sebep olur.

→ Haberleşme açık bir şekilde, programcı kontrolünde MPI'da yapılır.

**Data Paralel Model**: Programcı mesajlaşmayı görmez.

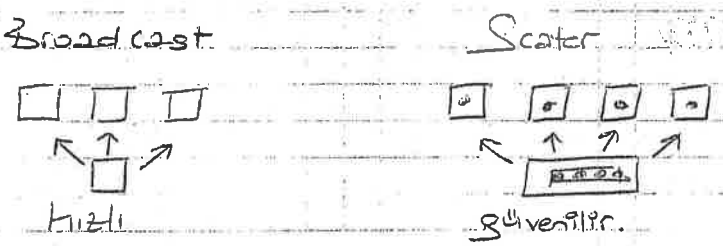
Haberleşme

**Senkron**
İki tarafın anlaşarak haberleşmesi.
Dezavantajı:
- Blocking
- Karşı tarafın mesajı aldığı bilinene kadar hiçbir iş yapılmaz.

**Asenkron**
mesajı karşıya yollar, başka bir işe bakar.
karşı taraf mesajı aldığını söyler.
Söylenmezse tekrar gönderilir.
blocking yok → hand shake

**Broadcast**: Mesaj herkese gönderilir. Herkes kendini ilgilendiren kısmı alır. (Hub)

**Scatter**: Mesajı sahibine teslim etme. Güvenli. Haberleşme zor ve pahalı.




2 şekilde gerçekleştirilir.

Point to point (Birebir)
Call (Bir → Herkes)

# Dağıtık Sistemlerin Tanımı

→ Kullanıcılara tek bir sistem gibi görünen fakat birbirinden bağımsız bilgisayarlardan oluşan sistem. (bu bilgisayarlar coğrafik olarak farklı yerlerdedir.)

→ Paralel Hesaplamada iş genellikle donanım-mimariye düşmektedir. Dağıtık sistemde böyle bir şart yok. Aralarındaki fark budur.

→ Dağıtık sistemi çalıştırabilmek için haberleşme ve iş bölümünü sağlayan katman; Orta sistem katmanı (middleware).

→ Paralel hesaplamada tüm PC'ler aynı olmalı. Çünkü bir işi paylaştırdığımızda farklı zamanlarda bitmesin, senkronize çalışsın diye. Ama Dağıtık sistemlerde farklı PC'ler vardır ve birbirinden bağımsız çalışırlar.

→ Middleware sayesinde kullanıcılar, sistemin geri kalan parçalarından haberdar oluyorlar. ÖRNEK: Torrentler. Dağıtık dosya sistemleridir. Böylece sistemin bir parçası oluyorsun.

→ Middleware:
- × Sayıt
- × OS üzerine kurulu
- × Üye olduğum dağıtık sistemin yöneticisinin adresi
- × PC'ler bunu kullanarak sistemin geri kalan kısmından haberdar olurlar.

→ Bir dağıtık sistemi kaynaklardan verimli bir şekilde faydalanmak için kullanırız. Tek PC değil, birçok PC bir araya getirilerek tek PC gibi görünür.

→ Middleware'in sağladığı diğer bir fayda; uygulama çalıştırılırken sistem üzerinde çalıştırmayı sağlar. Tek PC'da değil, tüm PC'lerde.

→ Her zaman tüm PC'ler kullanılmayabilir. Middleware sistemdeki tüm PC'lerin o anki durumuna göre yük dağılımı yapar. Herhangi bir PC'ye bir şey olursa diğerleri onun kısmını üstlenir, Sistem, bir nokta bir şey olduğunda çökmez.

→ Sistem public veya private olabilir. Kullanacağın yere göre sistem özellikleri değişir. ÖRNEK: Veritabanı çöktüğünde verilere ulaşabilmelisin; ama bunu yapmak kolay değil ve pahalı.

Distributed Hash: Middleware kullanarak sistemin bir parçası üyesi olmak. Böylece diğer bilgilerden haberin oluyor. Diğerlerinin kaynaklarına erişebiliyorsun. Artık tek PC değil, hepsi olıyorsun.

## Çözülmesi Gereken Dağıtık Sistem Problemleri:

1-) Veritabanı taşındığında, server çöktüğünde ne olacak?
2-) 2 Nolu PC, işlem yaparken " " "
3-) Sistemin çalışması için çöken'in görevleri diğerleri nasıl yapacak?
4) Çöken'in yedeği nasıl alınacak?

# Dağıtık Sistem Modelleri

1-) Transparency: Sistem özelliklerini müşteriden gizlemek.

a) Erişimin gizlenmesi: Bilgisayarları gizlemek, arka plandaki kaynakları gizlemek, kaynakların yerleri özellikleri --- gizlemek istenebilir.
Dağıtık Sistemlerde amaç, kullanıcıya tek PC izlenimi vermek.
DENEK: grid altyapısı, sanal kaynakların global OS gibi kullanılması.

* kaynaklara erişim güvenliği: Sistemin güvenliği açısından.

* kaynakların konumunun gizlenmesi.

* kaynak taşınmasının gizlenmesi Hızlı erişim için tüm kaynaklar dağıtılır. Herhangi bir yerde problem olduğunda sistem durmasın diye taşıdığın yerden devam edersin.

* verinin yedeklenmesi = replication

* concurrency = eş zamanlı çalışma Bir kaynağa çok çok kişi girebilir. Güvenlik açısından kimse bunu bilmemelidir.

* failure = sistemdeki hataların gizlenmesi. Güvensizlik oluşmasın diye müşteriye hata gösterilmez.

2-) Flexibility = Esneklik: Sistem esnek bir şekilde büyütülüp küçültülebiliyor olunmalı.
micro kernel metodu
middleware metodu
flexibility (kaynak ekle-çıkar ; esnek olmalı)

3-) Güvenlik: Sistem aksamadan çalışmalı. Sisteme güvenebilmeliyim.

4-) Performans: Yüksek performanslı hesaplama.

5-) Ölçeklenebilirlik: İhtiyaç oldukça sistem kaynağını devreye alabilmek. Bir kişi için çalışan sistem, 1 milyon kişi için de çalışabilmeli. Bunu otomatik yapmak gerekir.
Bulut Bilişim sayesinde otomatik ölçekleme iyi bir şekilde yapılmaktadır.
DENEK: 50 GB bant trafiğin var ama yetersiz. Çünkü web sayfana çok istek geliyor. Server'ı 100'e 150'ye çıkarsan da yetmiyor. Burada otomatik ölçekleme gerekir. Youtube, Facebook gibi. Yükü kontrol edecek olan server'ı otomatik almayı sağlar.
Ölçeklenebilirlikteki Sorun: Merkezi server yaklaşımı. Herkesin aynı yere erişmesi ile meydana gelen yoğunluk. (kullanılan ilk model : Server-client)
Küçük organizasyonlar için doğru bir işlem. Büyükler için dicentralized server yapılır. Cluster oluşturuyoruz. Bir işi örneğin 3 makineye ver. Gelen istekleri dağıtıcı her gelen isteği sıradakine ver. (Round Robin gibi bir algoritma kullanılır) 3 kat performans sağlanır.
Buradaki problem ise; Algoritmaları, programları, veritabanlarını --- dicenter yapamıyoruz. Paralelleşmeye uygun olmuyor.

Centralized Services = Mainframe : Güçlü bilgisayarlardır. Hem web, hem application'ları tutuyor. Supercomputer gibi çalışır. Bankalarda kullanılır. business logic üzerinde etkisi. Client'ler erişir. Veritabanına bağlıdır. Çok büyük ve yeteneklidir. Temel problemi: ölçekleme (biraz daha ram, biraz daha cpu artır) yok. Yetersiz kalması durumunda da 2. bir mainframe alınması, ihtiyacın çok çok üzerinde olacaktır.

Client-Server Modeli : Roller açık bir şekilde tanımlıdır. Mainframele göre daha esnektir. Yeni sunucu ekleyip cluster yapabiliriz. Fazlalaştıkça yönetimi zorlaşır. Server hizmet sunar, client hizmet ister, hizmet alır.
ÖRNEK: İnternet.
Clusterdaki ölçekleme network ile ve sizin yeteneğinize bağlıdır. Network trafiği fazlalaştıkça yönetim zorlaşır. Serverlar arası haberleşme çok yer kaplar.

Peer to Peer Modeli : Client-Server'daki limit olayının önüne geçilmesi için dir. Roller kesin olarak ayrılmamıştır. Dosya paylaşım sistemindeki dosyalar PC'lere gönderilir. Dosyayı yollayan server olur, server olan aynı zamanda diğerlerinden dosya çeker ve client olur. Oldukça ölçeklenebilir bir modeldir.
1. Problem: Bir uygulamayı parçalayamıyorsun.
2. Problem: Hizmet sunumu için uygun değil.

Kalın ve İnce İstemciler: PC'lerin pahalı olduğu dönemlerde çok popülerdi. Basit bir network kartı vardır. Üzerinde hesaplama, depolama ... yapılır. Yalnızca ince istemci merkezi sunucuya istek gönderir, sonuçlar da thin client'a gönderilir. Hala kullanılan bir model.
ÖRNEK: Windows'un uzak masaüstü olayı.

## ÇOK KATMANLI MİMARİ

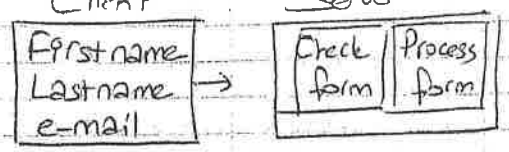
→ Özellikle web uygulamalarında (web & application)
→ Mainframe'lerde tek server'da
→ Client-Server'ın ilk mod tek server'da
→ 2 katmanlıda; 1-katman: web sunucusu 2.katman: Veritabanı.
→ 3 katmanlıda; 1. katman: işi göster 2. katman: işi yap 3. katman: çalıştır.
→ Yaygın kullanımı 3 katmanlıdır.
→ Sunucular arka planda dağıtılır.
→ Sunucuyu mysql ya da oracle'la bağla, konfigüre et ve sunucu ile bunları konuştur.
→ Bunlar merkezi modele yaklaşıyor. Merkezi modele replice edip cluster'a dağıtabiliriz. Bu zordur. Merkezi olmayan model kullanmalısın ama bunun da problemleri var:
* Hiçbir PC, sistemin tamamı hakkında kesin bir şeyler bilmez. (En yakını ve kendi bilgilerini tutar = dosya paylaşımında) Nedeni ise, gereksiz yük taşımak istenmemesidir.
→ Makinelerden birinin bozulması sistemi etkilemez.
ÖRNEK: Nosql veritabanı.
ÖRNEK: Dağıtık dosya sistemleri.

Scaling: ölçekleme için kullanılan metodlar;

1-) Process işini Server'da yap

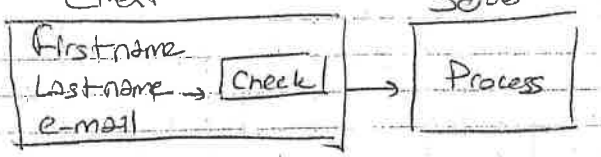
2-) Process işini Client'te yap.

1-)




her bir formu, server tek tek kontrol eder. Yanlışı geri bildirir.

2-)




kontrol olayını client yapar. Server meşgul edilmez.

⇒ İş mümkün olduğunca client'te yapılmalıdır. ⇐

DİKKAT: Name Spaceler - DNS Serverlar: Her networkte DNS server vardır. En yakındaki kullanılır. Resolve olması istenen isim DNS server'a parçalı gider → ilgilenebilirlikle ilgili güzel örnek.

## Dağıtık Sistemler için Varsayımlar:

1-) Network reliable olduğu varsayılır. Ağın güvenlidir, diyerek sistem tasarlıyoruz.

2-) Ağ güvenilirdir, diyerek işi yapıyoruz.

3-) Network homojendir (ama değildir. fiber'de hız farklıdır, bakır'da farklı)

4-) Topolojiler aynıdır. ( " " )

5-) Latency = 0 dır. (ağ gecikmesi sıfır sayılır, ama zamanında teslim ve zamanında cevap verme olmaz.)

6-) Bant genişliği sonsuzdur (ama değil.)

7-) Veri taşıma maliyeti sıfırdır (ama zaman veya paraya mal olur)

8-) Network yöneticileri aynıdır (farklıdır. cihaz yönetimleri farklıdır.)

## Dağıtık Sistem Türleri

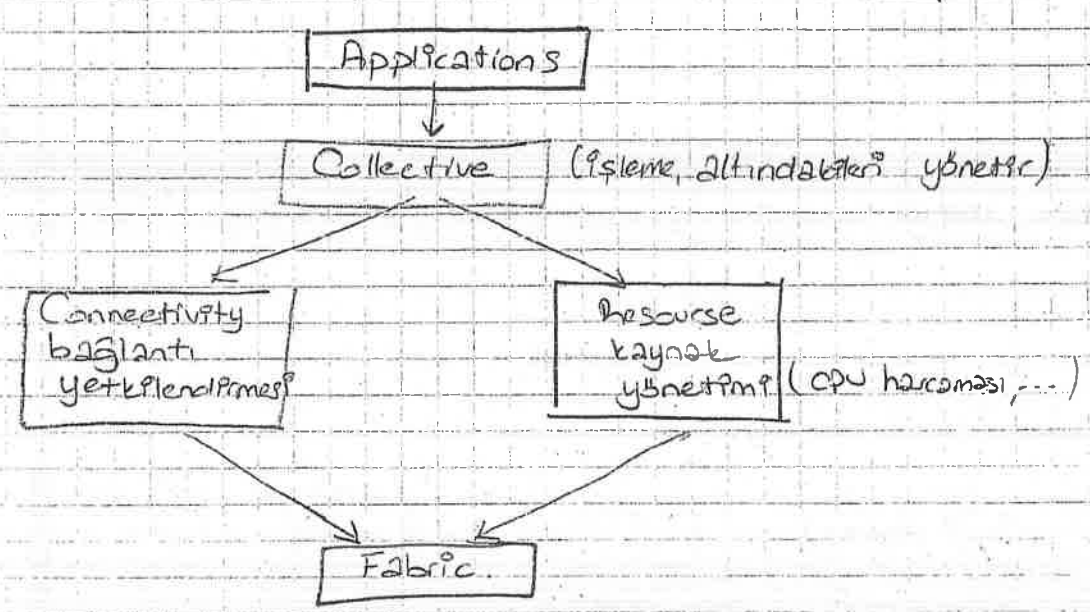
1-) Cluster = Küme:

→ PC'ler birbirinin aynısıdır. Yüksek hesaplama kapasitesine sahiptirler.
→ Kendi aralarında ağ ile bağlıdır. (Bu ağ hesaplama için değil)
→ Bunları yönetmek için ayrı bir ağ vardır.
→ Hesaplama için kullanılan ağ (fiber) = high speed network
→ Yönetim için " " = standard network
→ Hesaplamayı hızlandırmak için bir tane yönetici node vardır. İşi parçalar ve sonuçları toplamada görevlidir. (master - slave)
→ Tüm PC'leri bilir.
→ Diğer node'lar paralel hesaplamanın özelliklerini taşır.
→ Coğrafik olarak farklı konumdaki PC'lerde yapılmaz.
→ Dağıtıklığı da kendi localinde'dir.

2-) Grid Mimarisi:

→ Dünya'nın her yerindeki PC'lan kullanabilir.
→ Cluster'a göre karmaşık ve yeteneklidir.
→ Sanal organizasyon yapısını kullanır. 7 farklı ünite kullanılarak Türkiye sanal organizasyonunu oluşturur.

tr sanal org → Avrupadaki ülkelerin sanal org → Yönetici Node.




Fabric Layer: Yöneten katman.

Connectivity : Erişim / hangi protocol yetki.

Resource : Kaynak durumu / ne kadar CPU kullanıyor.

Collective : Connectivity ve Resource'u yönetir.

** Grid, hizmet odaklı mimarinin ilk adımı.

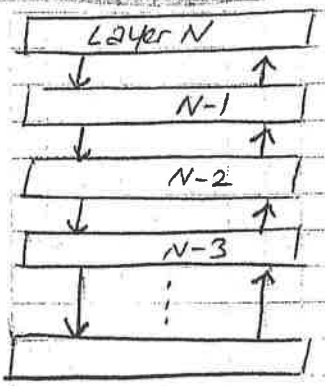
# ARCHITECTURAL STYLES:

→ 4 tane önemli mimari stil vardır.

× Katmanlı × Veri Merkezli

× Nesne Tabanlı × Olay Tabanlı.

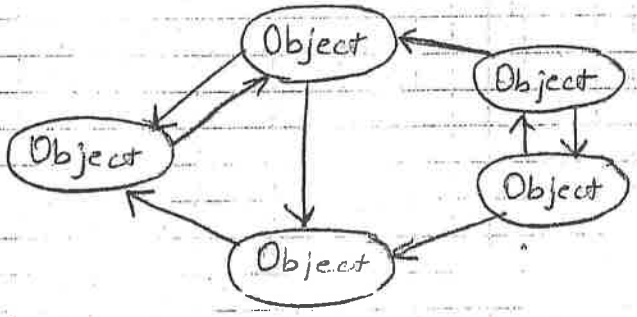
## Katmanlı:




→ Bir üst katman gerekli veriyi sıkartır, gerekli yerlere haberleşmeyi sağlar.

→ Katmanlar arası haberleşme tek yönlüdür. İstekler yukarıdan, cevaplar aşağıdan geliyor.

## Nesne tabanlı



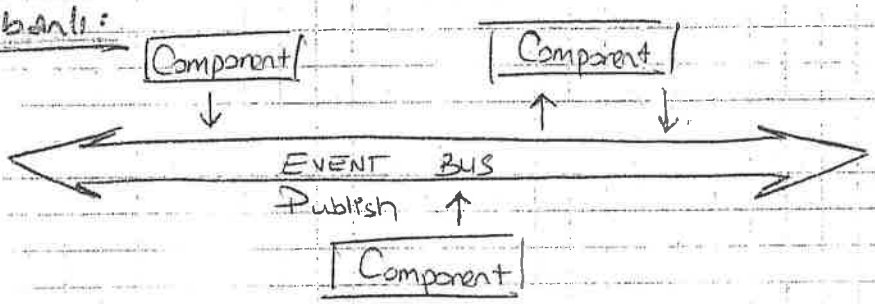

→ Elimizdeki nesneyi Java'da farklı programda çağırabiliyorduk

→ Buradaki durum ise; olağanlık olarak istenen kaynakları farklı PC'lere kullandırmaktır.

→ Beklenen ise; başka makine üzerindeki kaynakları kullanabilmek.

* Yeniden kullanılabilirlik: Client-Server mantığı.

## Olay Tabanlı:




→ Veri paylaşımı için geliştirilmiştir.
→ Bir olay olduğunda mesaj üretilir → Event bus'ta bir veri her yere dağıtılır. İlgili component bakıyor, kendisinimse alıyor, değilse bırakıyor.
→ Publish ile; bir component yayın yapıyor. diğerleri üye. Üyeler yayını dinliyor. İlgili üye ilgili veriyi alıyor. Publish yapan kişi, üyelerden habersizdir. Üretici (publish) mesajının ulaşıp ulaşmadığını bilmez.
→ Event bus bir çeşit servisdir.
→ Çok önemli işlerde, mesaj yerine ulaşmalıysa mutlaka subscribe'lar kullanılır.

Publish subscribe lar kullanım alanı olarak diğerlerinden farklıdır. Avantajlı fazladır. mesaj üretiminin fazla olduğu yerlerde kullanılır.
Önemli yazılım örneği : Javanın event kütüphanesi.

Problem: Publish yapan subscribe lardan haberdar değil.

Decopled inspace ; uzayda birbirinden bağımsız, gevşek bağlı.
quality services ; çok hassas sistemlerde bunu yükle.

bus = broker = aslında server broker mesajın gittiğinden emin olamaz.

Hassas sistemlerde extra bağlantı yapılmalı, iş karmaşıklaşır → Problem ;;

Problem : Kayıp mesajlar. Her bir subscribe gelen mesajın numarasını bus üzerinden üreticiye gönderir. Bus yoğunlaşır, bus olmaktan çıkar, Server olur.

Veri Merkezli :

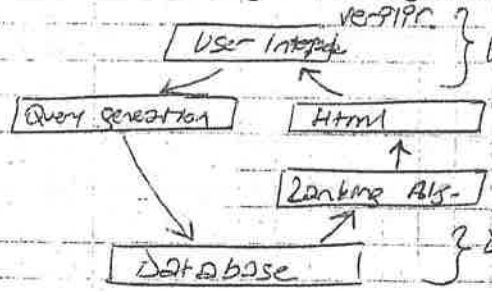
→ Bileşenler arası haberleşmeyi veri üzerinden yapar.
→ İşler dosyalar üzerinden gerçekleşir.
→ Mesaj paylaşımlı dosyaya yazılır.
→ İhtiyacı olan o dosyayı alır, mesajı çeker.

## Bu mimarileri Nasıl Gerçekleştiriyoruz?

### Centralized Architectures

Client - Server modeli.
Server, bir şeyi sunabilmek için başka server'a ihtiyaç duyuyorsa client durumundadır.
Bağlantısız bağlantı var. Aradaki bağlantının güvenilir olduğunu varsayıyoruz.
İsteği yap, unut, hatta meşgul etme.

İşler katmanlara bölünmüştür:
1- User Interface Layer; Arayüz
2- Business Logic ; İş yapılan yer
3- Database Layer; Sorgulara cevap verilir.

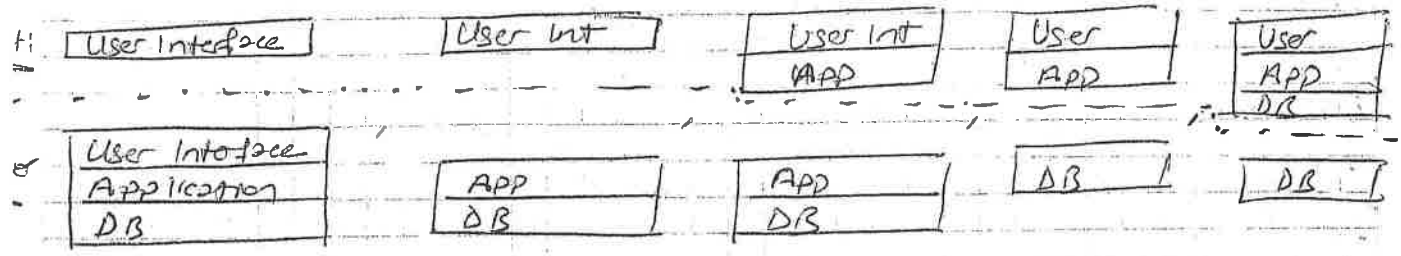



İsteğe göre Server'a ya da client'a iş yaptırılabilir. Avantajı ; server'a yük binmemesi.

### Decentralized Architectures

Bir makine hem server hem de client olabilir.
Makineler arasındaki fark net değil.
Peer to Peer network
Distributed hash table ; makine ile dosyalar birbirini eşleştirir. Hangi makinede hangi dosyanın tutulduğu bu tabloda kayıtlıdır.

Alternatif Client Server Organizasyonları:




## INTERCEPTORS:

→ Çok karmaşık bir programım var. Parametre alacak → Çalışacak → Sonucu döndürecek. Uzaktaki bilgisayarın bunu çalıştırması için araya farklı katmanlar koyulmalıdır. (Middleware mobilde çalışmaz)
→ Interceptor istek geldiğinde dosomething ile çağrıyı alıyor. İçinden isteği alıkarıyor. Middleware bu isteği görüyor ve gerekli yere isteği gönderip işletiyor. ve Geriye döndürüyor.
→ Böylece modülerlik sağlanır.

## ADAPTIVE YAKLAŞIM

→ Her şeye uyumlu uzun yıllar süren teknoloji olmak.
→ Kendini otomatik ayarlıyor.
→ 3 yöntemi var.

(1) Separation of concerns: İhtiyaç olan şeyleri birbirinden ayır. Veriyi ayır, işlemleri ayır, filtrelemeyi ayır, İşi katmanlara ayır... Bunu yaparsan her bir parçayı kendi içinde geliştir.

(2) Computational reflection: Yansıtma yöntemiyle Runtime değişiklik yapılır. Java'da vardır. (Çalışan program üzerinde yapacağına, nesneyi kullanarak yapasın) Bu, ortam şartlarına göre programı adapte etmenizi sağlar.

(3) Component base design: Tasarım yaparken bileşenlere ayır. Orijinal program iş mesajları ayır ya da database ile hesaplamayı ayır.

→ Bu 3'ünü yapamadığımızdan adaptive sistemler başarılı çalışamamıştır.

# PROCESS & THREAD

**Process :** Yürütülmekte, hafızada çalışmakta olan program parçasına process denir.

Elimizde bir program var. Bu programa ait process'i RAM'e verdik. İşletim sistemi bunun global değişkenlerini tanımalı, fonksiyon oluşturmalı.... Process'in çalışacağı adres aralığı belirlenir. Process Kontrol Bloğu oluşturularak belirli bir adres aralığı burada tanımlanır.

Değerlerin RAM'e yüklenmesi ve bunların sırasıyla işlenmesi, her Process için bir process kontrol bloğunun oluşturulması, bazı algoritmalar kullanılarak (round robin gibi) tek CPU'nun birden fazla CPU gibi çalışmasının sağlanması ... gerçekleştirilir.

**Avantajları :**

→ Kaynakların daha verimli kullanılmasını sağlar.
→ CPU, RAM ... yüksek oranda kullanılmasını sağlar.
→ Birden fazla process çalışabilir. Dolayısıyla bir işin birden fazla process'i olabilir.

**Dezavantajları :**

→ Her bir process'i oluşturmak için özel hafıza bilgeleri oluşturulur (stack)
→ Her bir process için PCB oluşturulur.
→ Input / Output işlemleri yapılır.
→ Bu işlemler zaman alıcıdır.
→ Masraflıdır.

Bu dezavantajları gidermek için thread'ler geliştirilmiştir. Thread, processi oluşturan en küçük birimlerdir. O.S. kaynaklarının kod bölümünü ve veri bölümünü paylaşımlı bir şekilde kullanarak processlerin daha verimli çalışmasına yardım eder.
Processler'i threadlere bölersiz. Threadler daha ucuz ve daha kolay oluşturulur. Process pek çok thread'in toplamıdır.

Thread Çeşitleri

1) User Seviyesi : Programcı hiçbir kütüphaneye ihtiyaç duymadan yalnız kendi çalıştığı seviyede oluşturulan thread'lerdir. Fakat input/output durumunda verimsizdir. (Bu durumlarda user seviyesinde processin tamamı bloke olabilir)

2) Kernel Seviyesi : Her process için birden fazla thread kullanılır. Burada bir kesme veya dağın geldiğinde kernel thread kullanılmaktadır.

3) Hibrit : User + Kernel, hem user seviyesinde hem de kernel seviyede işlem gören threadlerdir.

4) Lightweight Process : Solaris'te uygulanıyordu, artık kullanılmıyor. User'da çalışan threadler LWP'ye atanır. Atanan LWP Input/output gerektirirken, user maddeki işine devam eder.

Threadler paralel işlem yapabilmeyi sağlarlar. Bunun için paralel kütüphaneler kullanılır. Paralel çalışan yapıyı programlama dilleri sağlamaktadır.

Threadler MPI kullanmadan paralelleştirmeyi sağlarlar. Zaten MPI ve passing message kullanımı masraflı ve zordur. MPI ile bağımsız processlerin haberleşmesi çok zor. İşte threadler bu haberleşmeyi gerçekleştirmektedirler. Aktif çalışan bir processin aynı anda yapması gereken işlemleri yapmasını sağlar. Bunu yaparken de processin hafızasını kullanır. Böylece thread, paylaşımlı hafıza yöntemi ile haberleşmektedir. Bu da önemli oranda paralelleştirmeyi sağlamakta.

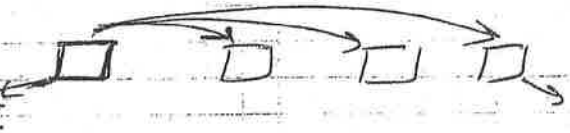
## Thread ve Process

- Process sistem tarafından belirlenen kaynakları kullanır, Ama thread ait olduğu processin kaynaklarını kullanır.
- Process işlem yaparken özel bir hafıza alanı kullanır. Thread o an yürütülen işlemin alanını kullanır ve çok daha az kaynak kullanır.
- Threadleri processler, processleri kullanıcılar başlatır.
- Threadler ait oldukları processin tüm kaynaklarına erişebilir. Avantajı; özel hafıza yönetimine ihtiyaç yok. Dezavantajı; senkronizasyonu sağlamak zordur. Aynı hafızayı kullandıkları için biri diğerinin kaynaklarını görebilir ve kullanabilir.
- Aynı program birçok kez çalıştırıldığında, her seferinde birbirinden farklı processler olur. ve her seferinde farklı PCB oluşturulur. Thread'de böyle bişey yok.
- ***** Multithreading'in bulunması dağıtık sistemi geliştirdi *****

## = MULTITHREADED SERVERS =

Threadlerin en çok kullanıldığı yerler serverlardır. Serverlar Clientlerin isteklerine cevap vermektedirler. Serverdan beklenen tüm Clientlere cevap vermesidir. Yani bir isteğe cevap verirken, aynı anda gelen başka istekleri de değerlendirmelidir.

Bu işlem pratikte yapılamaz; ancak Server birçok thread ile çalışabilir. Serverda birçok thread olsun. Başta bekleyen thread gelen isteğe cevap verir. Bu durumda Round Robin gibi bir schedular kullanılır.




⇒ Round Robin

Buradaki problem; Örneğin 10 istek karşısında 10 thread cevap veriyor. 11. istek geldiğinde ne olacak? Çok fazla thread üretilse, bu sefer de programlanması zor olur. Ne yapılacak?

Çözüm; dispatcher thread'in varlığıdır. Dispatcher thread Clientten gelen isteği Worker thread'e iletir. Kendisi ise ağı dinlemeye devam eder. Network haberleşmesi yapan tek thread = dispatcher thread.

Workerdan beklediği şey ise gelen isteği değerlendirmesi. Worker thread, gelen isteği işler ve sonucu geri gönderir. Tüm workerlar bu işlemi yapıyorlar. Bu, multiprocessing server kullanımıdır. Şu anda kullanılan yöntem de budur. Yani özetle; kaynaklar sınırlı, istekler sınırsız. Bu nedenle thread kullanılmıştır.

Olaya Client tarafından bakarsak; burada tarayıcı kullanılır. Tarayıcı sayfayı yüklerken; örn: css, javascript... yüklenmesini görüyoruz. Eskiden single threading kullanılıyordu. Sayfa en başından yavaş yavaş ve parça parça yükleniyordu. Örn: önce 1 resim, sonra 2. resim... Network haberleşmesinde 1 tane thread var. Server, veriler buna byte byte gönderiyor. Bu thread de bunu yavaş yavaş kullanıcıya gösteriyor.

Tarayıcılar html sayfasını yüklerken;

- html in header'ında yüklenmesi gerekenlere bakar.
- 1 tane thread hızlıca (gerekli css ---) bunları getirir. (Serverdan) ve tarayıcının geçici hafızasına yazar.
- Tarayıcı, burada resim olduğunu görüyorsa her resim için bir tane worker thread çalışır. Aynı anda birçok thread aracılığıyla, kullanıcı tarafındaki isteklere hızlı bir şekilde cevap verilir.

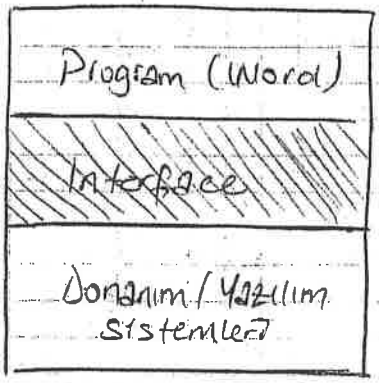
Özetle; Process; 1 tane program var → Bunu çalıştıracaz → Bu derlenecek → binary kodu olacak → Ramle yükle → OS, processin çalışması için kaynak veriyor. Hafıza verdi, global değişkenlerini tuttu, adres uzayında nereleri kullanacak onu söyledi, Process'in erişeceği dosyayı söylüyor → Process OS'in kendisini CPU'ya vermesini bekliyor → Aynı anda birden fazla process'i sisteme yükleyebiliyoruz → 1 tane processor'ü 10 taneymiş gibi kullanabiliyor.

## SANALLAŞTIRMA :

→ OS kullanabileceği kaynak miktarı bellidir. Bunun yerine aynı donanım üzerine de birden fazla işletim sistemi kurabilirim.

→ "Donanımdan bağımsız olarak, yazılımı donanım üzerinde çalıştırmak"

→ Program donanıma bağlı olarak çalışır. Ör: derleme işlemi donanımdan bağımsız olamaz. Bu yüzden sanallaştırma kolay bir iş değildir.




→ Win. OS. üzerine Word'ü kurduk. Bunun bazı ihtiyaçları var. (Process haberleşmesi için intel process communication sağlama, RAM, Stack ...) Bunların hepsini OS sağlar. Word hiçbir zaman donanımı görmez. Her zaman OS'in interfacelini görür. (API) Bu işlem bir nebze sanallaştırma işlemidir.

→ Sanallaştırma birden fazla katmanda yapılabilir.

★ Donanım ve yazılım arasında sanallaştırma yapmak için makinenin anlayacağı instructionları sağlaması gerekir. Bunun için iki yöntem kullanılır.

1) Sistem çağrılarını yalnız OS yapabilir.

2) Kütüphane çağrılarını yapabilmek için bir interface gerekir.

★ Interrupt ; Sadece OS donanıma ulaşabilir, çünkü başkaları erişirse kaos olur. Ama donanımı OS dışındaki programlar tabii kullanabilir.

★ Kütüphaneler ; Donanımın OS tarafından kullanılabilmesi için driverlar kullanılır. OS istediğinde interrupt oluşturur ve o driver'ı kullanır. Biz OS aracılığı ile kullanırız. Bunun için de kütüphanelere ihtiyaç var.

Yani ; donanımı kullanmayı sağlayacak Interface → Öncelikli instructionlar

↓

Driverlar.

## Standartların Avantajları

1-) Uluslararasıdır. Herkes kullanabilir.
2-) Binary değil text formattadır.
3-) Programlama dillerine bağlı değildir.
4-) Basittir.
5-) Ucuzdur.
6-) Kurumsal uygulama gerçekleştirilebilir.
7-) Kiralama olayı ile geliştirilebilir.

ÖRNEK: Berkeley Üni. hizmetleri.

NOT xmethods.com birçok web servisi var. (UDDI directory)

~~UDDI~~ ile webservis paylaşımı sözleşmesi ~~yok~~

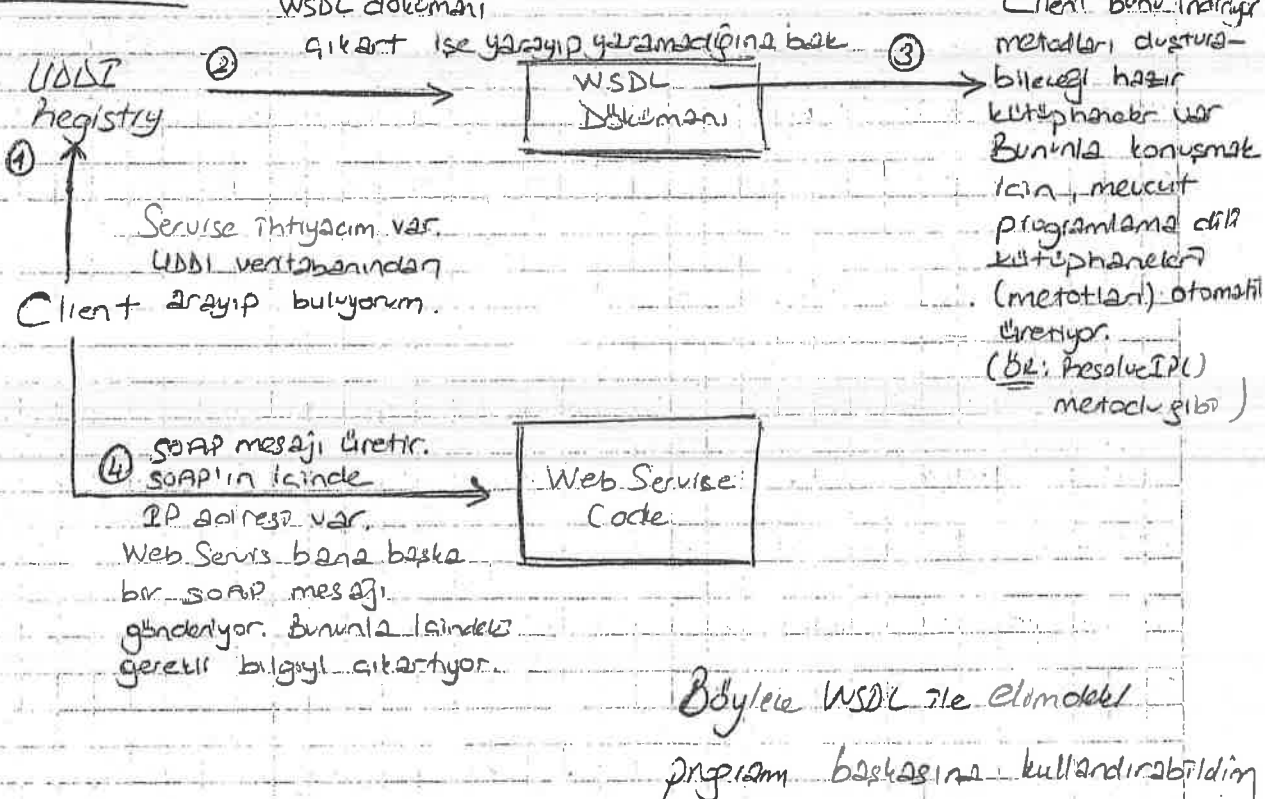
Ör:

| Publisher | Style | ServisNama |
|---|---|---|
| Paylaşan kişi | RPC DOC | Servisin adı |

ÖRNEK: IP adresini alıp, kime ait olduğunu söyleyen bir web servisi.
Para birimi dönüştüren bir web servisi

---

## Nasıl Çalışır?

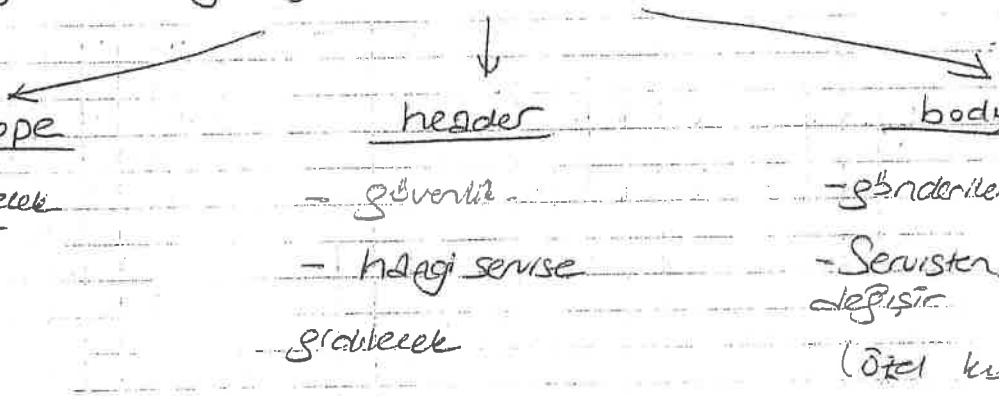



Böylece WSDL ile elimdeki program başkasına kullandırabildim

Bu 4 standart çok önemli.

**XML ;** Esnek bir dil olduğu için, istenilen veri türü ifade edilebilir.

**SOAP ;** Mesajı içinde taşıyan zarf. Sorgu yada cevap yazılır. Zarfa adres yazılır. İlgili yere gönderilir. SOAP kısımları




**WSDL ;** XML tagları sınırsız değildir.

Metotlar burada tanımlanır. (parametre, metot, operation...)

Bunu indiren kendi kodlarını yazabilsin diye tanımlama yapmalıyız.

Bir de kodlar hangi makinelerde çalışıyor, bunu söyler. Makineler WSDL'deki adresi indiriyor, gerekli bilgileri alıyor. Cevabı XML dökümanı şeklinde döndürüyor.

## Mesajı Nasıl Gönderiyoruz ?

Form doldurulduktan sonra XML dökümanı üretilir (Bu SOAP) Bunu makine yapıyor. Java kodu ile kendim de yapabilirim. Ama elimde 1000 tane IP adresi olsa bunu for ile oluşturabilirim. Tek tek yazmam. İşte böyle durumlarda web servisler kullanılır.

**UDDI ;** Veritabanı gibidir.

- İnsanların adı olarak kullandığı
- Şirketlerin kullandığı
- Paralı (Satılan)

Registry'ler vardır.