İçindekiler

1.Kavramlar

Soketler, aynı veya farklı hostlar üzerindeki süreçlerin haberleşmesini sağlayan bir haberleşme (interprocess communication) yöntemidir. Soket, soyut bir tanımla haberleşme uç noktalarıdır. POSIX'in sağladığı programlama API'si sayesinde programcı soketlere yazarken veya okurken yine `write()`,`read()` gibi sistem çağrılarını kullanabilir.

İstemci (Client): Hizmet isteyen soket programlara denir.

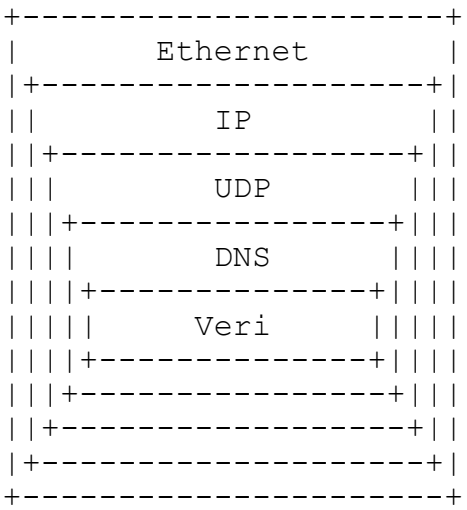
Sunucu (Server): Hizmet veren soket programdır.

Port: Bir bilgisayarda birden çok soket bulunabilir. Örneğin hem telnet soketi, hem de ftp soketi açık olabilir. Soketleri birbirinden ayırt etmek ve istemciyi sunucudaki uygun program ile buluşturmak için her soket programın PORT denilen bir numarası vardır. Örneğin FTP protokolünün port numarası 21, Telnet sunucunun port numarası ise 23'tür. Standart servislerin port numaraları /etc/services dosyasında tanımlıdır. 1-1024 arasındaki portlar yalnız root tarafından kullanılabilir, normal kullanıcılar bu portları bind edemezler.

IP Numarası: TCP/IP protokolü hostları, sadece kendisine ait olan bir IP numarası ile tanımlar. Bilgisayarlar birbiri ile IP numarasını kullanarak haberleşirler. Bir istemci soket program, önce IP numarasını kullanarak sunucunun bulunduğu bilgisayara, sonra PORT numarasını kullanarak hizmet istediği sunucu program ile temasa geçer. IPV4 standardına göre IP'ler 192.168.1.10 gibi [0-255].[0-255].[0-255].[0-255] formatına sahiptir. IPV6 standartı 128bit adres kullanır.

Sarmalama(Encapsulation): Her katman kendisine gelen pakete bir başlık ekler ve bir sonraki protokole aktarır. Buna encapsulation (sarmalama) denir. Karşı tarafta paket açılırken yine her katman kendisi ile ilgili başlığı açar. Örneğin fiziksel katman (mesela ethernet aygıtı) gönderen

sistemin fiziksel katmanının eklediği başlığı açar. Zaten diğer başlıkları yorumlayamaz. Aşağıda sarmalanmış bir paket görülmektedir.



1.1. UNIX Domain Soketleri

1.2. Internet Soketleri

1.3. Domain Soketleri ve Internet Soketleri

Aşağıdaki mail, FreeBSD core team üyesi Robert Watson tarafından gönderilmiştir.Yararlı bulduğum için aynen ekliyorum.Mail, UNIX domain soketleri ve internet soketlerinin karşılaştırılması hakkındadır.

On Fri, 25 Feb 2005, Baris Simsek wrote:

> What are the differences between impelementations of them at kernel level?

There are a few differences that might be of interest, in addition to the already pointed out difference that if you start out using IP sockets, you don't have to migrate to them later when you want inter-machine connectivity:

- UNIX domain sockets use the file system as the address name space. This means you can use UNIX file permissions to control access to communicate with them. I.e., you can limit what other processes can connect to the daemon -- maybe one user can, but the web server can't, or the like. With IP sockets, the ability to connect to your daemon is exposed off the current system, so additional steps may have to be taken for security. On the other hand, you get network transparency. With UNIX domain sockets, you can actually retrieve the credential of the process that created the remote socket, and use that for access control also, which can be quite convenient on multi-user systems.

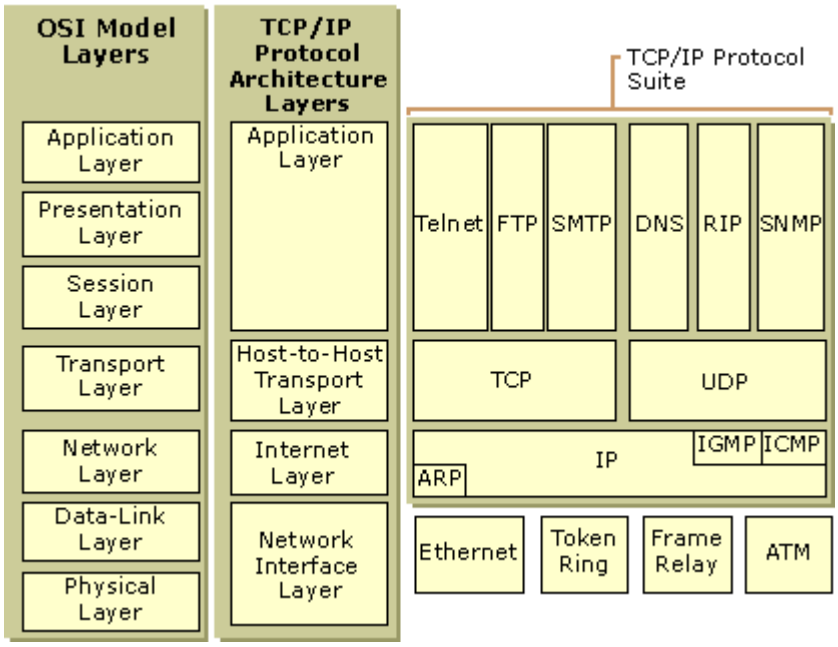
- IP sockets over localhost are basically looped back network on-the-wire IP. There is intentionally "no special knowledge" of the fact that the connection is to the same system, so no effort is made to bypass the normal IP stack mechanisms for performance reasons. For example, transmission over TCP will always involve two context switches to get to the remote socket, as you have to switch through the netisr, which occurs following the "loopback" of the packet through the synthetic

```
  loopback interface.  Likewise, you get all the overhead of ACKs, TCP
  flow control, encapsulation/decapsulation, etc.  Routing will be
  performed in order to decide if the packets go to the localhost.
  Large sends will have to be broken down into MTU-size datagrams, which
  also adds overhead for large writes.  It's really TCP, it just goes over
  a loopback interface by virtue of a special address, or discovering that
  the address requested is served locally rather than over an ethernet
  (etc).

- UNIX domain sockets have explicit knowledge that they're executing on
  the same system.  They avoid the extra context switch through the
  netisr, and a sending thread will write the stream or datagrams directly
  into the receiving socket buffer.  No checksums are calculated, no
  headers are inserted, no routing is performed, etc.  Because they have
  access to the remote socket buffer, they can also directly provide
  feedback to the sender when it is filling, or more importantly,
  emptying, rather than having the added overhead of explicit
  acknowledgement and window changes.  The one piece of functionality that
  UNIX domain sockets don't provide that TCP does is out-of-band data.  In
  practice, this is an issue for almost noone.

In general, the argument for implementing over TCP is that it gives you
location independence and immediate portability -- you can move the client
or the daemon, update an address, and it will "just work".  The sockets
layer provides a reasonable abstraction of communications services, so
it's not hard to write an application so that the connection/binding
portion knows about TCP and UNIX domain sockets, and all the rest just
uses the socket it's given.  So if you're looking for performance locally,
I think UNIX domain sockets probably best meet your need.  Many people
will code to TCP anyway because performance is often less critical, and
the network portability benefit is substantial.

Right now, the UNIX domain socket code is covered by a subsystem lock; I
have a version that used more fine-grain locking, but have not yet
evaluated the performance impact of those changes.  I've you're running in
an SMP environment with four processors, it could be that those changes
might positively impact performance, so if you'd like the patches, let me
know.  Right now they're on my schedule to start testing, but not on the
path for inclusion in FreeBSD 5.4.  The primary benefit of greater
granularity would be if you had many pairs of threads/processes
communicating across processors using UNIX domain sockets, and as a result
there was substantial contention on the UNIX domain socket subsystem lock.
The patches don't increase the cost of normal send/receive operations, but
due add extra mutex operations in the listen/accept/connect/bind paths.

Robert N M Watson
```

2.TCP/IP Protokolü

TCP/IP, "*Transaction Control Protocol and Internet Protocol*" için bir kısaltmadır. TCP/IP, DARPA(Defense Advanced Research Projects Agency) projesi olarak 1970'lerde Amerika'daki bazı devlet kurumlarını birbirine bağlamak amacı ile geliştirildi. Daha sonra BSD(Berkeley Software Distribution) UNIX'e eklenerek uygulama alanı buldu. TCP/IP'nin içerdiği protokollerin tanımları RFC(Request For Comments)'lerde yapılmıştır. RFC'lere ulaşmak için: http://www.faqs.org/rfcs/

TCP/IP 4 katmanlı bir modeldir: 1. Uygulama Katmanı 2. Taşıma Katmanı 3. Internet Katmanı 4. Ağ Erişim Katmanı. Her bir katman OSI modelindeki bir veya daha fazla katmanın işlevine sahiptir. 1982'de BSD UNIX ile TCP/IP uygulamaları geliştirildi.



2.1. Ağ Erişim Katmanı

Bu katman TCP/IP paketlerini fiziksel ağa bırakmak ve aynı zamanda fiziksel ağdan gelen paketleri almakla görevlidir. OSI modelindeki Fiziksel katman ve Veri-Bağ katmanının karşılığıdır.

2.2. Ağ Katmanı

Bu katman adresleme, paketleme ve yönlendirme fonksiyonlarını yerine getirir. IP, ARP, ICMP ve IGMP protokolleri, bu katmana ait çekirdek protokollerdir.

Internet Protocol (IP): Adres bilgilerini ve paket yönlendirme için bazı kontrol bilgilerini içerir. RFC 791'de tanımlanmış olup en önemli internet protokolüdür. İki önemli görevi vardır: 1. Ağlar arasında bağlantısız datagram dağıtımını yapmak, 2. Fregmantasyon ve veri katmanına yardımcı olarak değişik MTU(maximum-transmission unit) değerleri ile datagramları yeniden oluşturmak. IP paketinin başlık yapısı aşağıdaki gibi tanımlanmıştır:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Version: Kullanılan internet başlığının biçimini, versiyonunu gösterir.
IP Header Length (IHL): Datagram başlığının 32 bit olarak boyutunu gösterir. Doğru bir IP başlığı için başlık boyutu en az 5 olmalı.

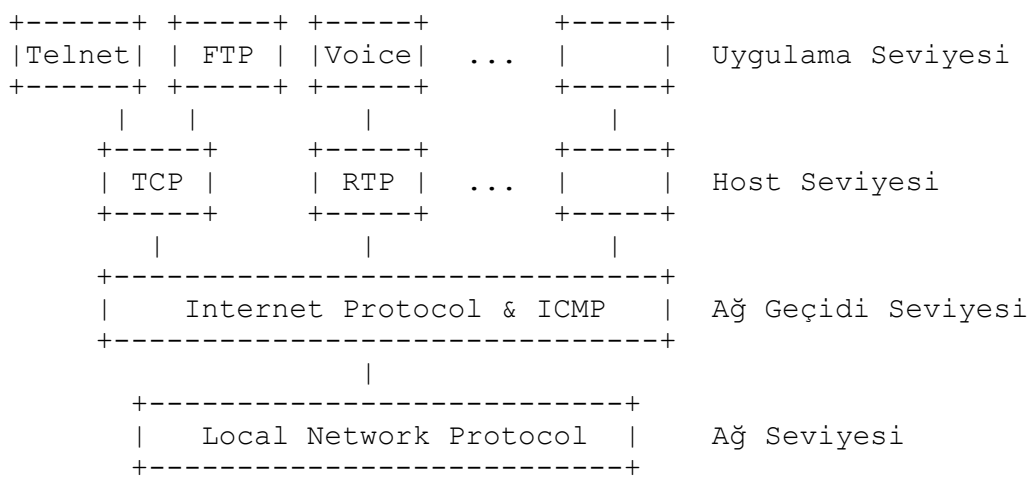
Type of Service: İstenilen hizmet kalitesi ile ilgili soyut parametreler sunar. Örneğin bazı ağlar, önceliği destekler. Trafiğin bir kısmına öncelik verilebilir.
Total Length: Başlık ve veri bilgisi ile birlikte toplam datagram boyutunu gösterir. 16 bittir, buradaki değer byte olarak gösterir. Yani IP paketi en fazla 64K boyutunda olabilir.
Identification: Gönderen tarafından yazılır. Datagram parçalarını biraraya getirmeye yardımcı olur.
Flags: Paketin parçalanabileceğini veya parçalanamayacağını gösterir.
Fragment Offset: Bu paketin datagram içerisinde nereye ait olduğunu tanımlar.
Time to Live: Sürekli azalan tam sayıdır. Paketin hangi noktada yok edileceğini belirtir. Paketin sonsuza tek ağda kalmasına engel olur.
Protocol: IP'nin işi bittikten sonra paketi hangi üst protokol alacağını gösterir.
Header Checksum: IP başlığının bozulmadığına emin olmak için tutulan değer
Source Address: Gönderen noktayı gösterir.
Destination Address: Alıcı noktayı gösterir.
Options: IP, güvenlik gibi değişik seçenekleri destekler.
Data: Üst katmana verilecek veriyi tutar.

2.3. Taşıma Katmanı

Bu katman transparan bir şekilde verinin hosttan hosta taşınmasını sağlar. Akış kontrolünü ve hata düzeltmeyi sağlar. Veri transferinin bittiğinden emin olur. TCP ve UDP protkolleri bu katmana aittir. Ağ katmanı bağlantı yönelimli (connection oriented) bağlantı sağlamaz. Taşıma katmanı bunu sağlar. Ağ katmanı ulaşan paketlerin, gönderildiği sırada ulaştığını da garanti etmez. Taşıma katmanı her paketi numaralandırarak bunu basitçe çözer. Hata oluştuğu durumda paketi yeniden ister. Böylece oluşabilecek hataların önüne kesilir.

Transmission Control Protocol (TCP): TCP, IP ortamında uçtan uca güvenli haberleşme sunan bağlantı yönelimli(connection oriented) bir protokoldür. RFC 793'de tanımlanmıştır. Uygulama katmanının hemen altında bulunur. Aynı zamanda süreçler arası haberleşme(interprocess communication) prokolüdür. İki süreç arasında sanal bir devre oluşturur. Telnet, TCP kullanan popüler uygulamalardan birisidir.

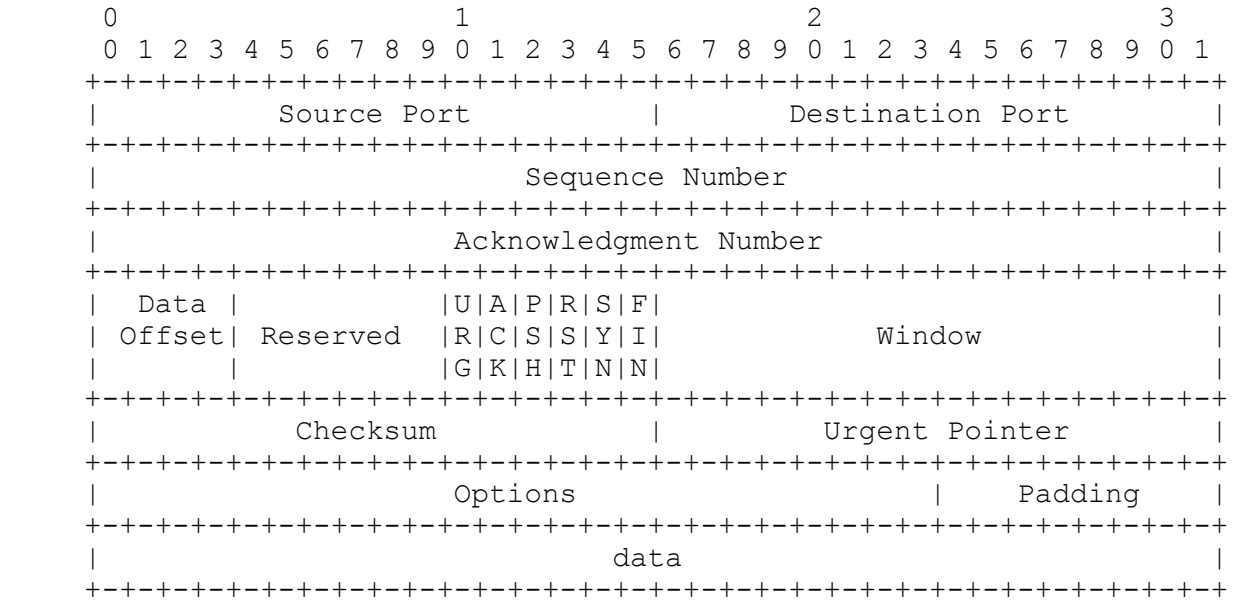
TCP, zarar görmüş, kaybolmuş veya sırası bozulmuş veriyi kurtarabilir. Aktarılan her sekizlik için sıra numarası tutar ve alıcı noktadan olumlu ACK(Acknowledge-aldığını bildirmek) bekler. Eğer ACK, bir zamanaşımı(timeout) süresi içerisinde gelmezse veri yeniden aktarılır. Alıcı taraf verileri sıralı almamış veya geciken ACK'lerden dolayı birden fazla almış olabilir. TCP bunları düzeltir. Her bir segmente bir kontrol toplamı(checksum) eklenerek alıcı tarafın aldığı verinin doğru olup olmadığını denetlemesi sağlanır. TCP'nin diğer protokoller ile ilişkisi:



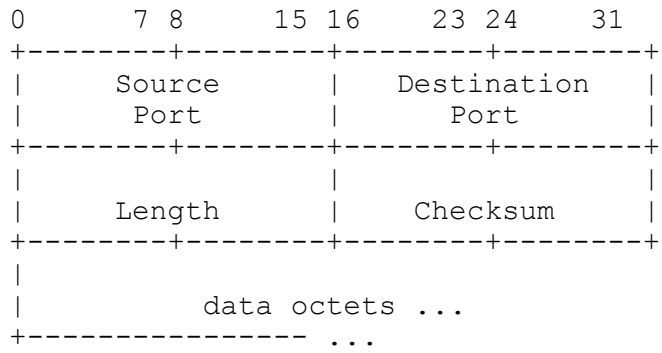
TCP, veri stream'lerini birbirinden ayırt etmek için port tanımlayıcı kullanır. Her TCP birbirinden bağımsız port tanımlayıcı sunar. Bu nedenle port tanımlayıcılar tek olmayabilir. Bu nedenle soket oluşturulurken internet adresi de kullanılır.

Bir bağlantı tamamen uçlardaki soketler arasında oluşur. Yerel bir soket pek çok dış soket ile bağlantı yapabilir. Bağlantı iki yönlü veri taşımada (full duplex) kullanılabilir.

TCP segmentleri internet datagramları olarak gönderilir. Çünkü altında IP protokolü vardır. TCP segmentleri, IP tarafından paketlenip gönderilir. TCP başlığı aşağıdaki gibidir:



User Datagram Protocol (UDP): RFC 768'de tanımlanmıştır. Bu protokol uygulamalar için en az protokol yükü ile haberleşme olanağı sağlar. Protokol işlemi gerçekleştirmeye yöneliktir. Dağıtım ve güvenliği temin etmez. Checksum değeri tutar ancak paket bozulmuşsa yeniden paketi dağıtmaz. Başlık yapısı şu şekildedir:



DNS ve tftp bu protokolü kullanan en popüler uygulamalardır.

Bu iki türün özellikleri ve aralarındaki temel farkları şöyle sıralayabiliriz:

1. Stream soketler verileri sıralı gönderir, datagram soketleri sıralı göndermeyebilir. (TCP protokolu, paketleri sıralı göndermeyi garanti eder. UDP garanti etmez. TCP paketlerin başlık bilgisinde sıra numarası vardır, UDP'de yoktur. TCP, her zaman sıradaki paketi ister. Örneğin 4 numaralı paket yerine 5 numaralı paket eline ulaşırsa karşı tarafa bunu bildirir ve 4'ü ister. 4'ü alınca da 5'ten önceye koyar.)

2. Stream soketler güvenlidir, Datagram soketler güvensizdir. (TCP protokolu güvenliği garanti eder, UDP garanti etmez. Çünkü TCP acknowledgement ile denetim yapar. Yani bir paketi gönderdiği zaman, karşı taraf paketi aldığını haber vermeden o paketi göndermiş saymaz kendini ve tekrar gönderir. ayrıca paketin doğru gidip gitmediğini anlamak için başlık bilgisinde checksum -kontrol bilgisi- tutar. UDP'de checksum tutar ancak checksum yanlışsa aynı paketi tekrar istemez.)

3. Stream soketler, işlem bitene kadar kesintisiz bir bağlantı kurar. Datagram soketler ise bağlantı kurmaz. Sadece veri göndereceği zaman bağlantı kurar ve işi bitince bağlantıyı koparır.

Bu iki arasındaki farkı anlatmak için postacı ve telefon benzetmesini vereceğim. Mektup insanlar arasında haberleşmeyi sağlayan bir yöntemdir. Postacı mektupları posta kutusuna bırakıp gider. Kişi ise mektupları müsait olduğu herhangi bir an (belki 1 saat sonra, belki 1 gün, belki 1 hafta) alır ve okur. Cevabını yine posta kutusuna atar ve postacı bir süre sonra mektupları alıp karşıya taşır. Telefon örneğinde ise, bir taraf diğer tarafa telefon açar. Aradaki bağlantı kurulduktan sonra insanlar bağlantı kopmadan karşılıklı konuşurlar. Posta örneğinde bağlantının sürekliliği gibi bir şey söz konusu değildi. Telefon görüşmesinde sözlerin sıralı gitmesi söz konusu. Yani sözler birbirine karışmaz. Ancak postada ise durum farklı. Mektuplar karşı tarafta sıralı okunmayabilir. Örneğin posta kutusunda 5 mektup birikince mektuplarını okur. Telefonda ise karşılıklı sürekli konuşulur ve söylenen karşıya iletilir.

UDP'nin bu kadar tez avantajına rağmen neden daha çok kullanıldığı bu şemalardan açıkça görülmektedir. TCP bir veri karşıya 6x32+Veri boyu kadar bir paket olarak gitmektedir. Yani her paket fazladan 192 bit başlık (header) bilgisi taşımaktadır. Oysa UDP paketleri 64 bitlik başlık (header) bilgisine sahiptir.

UDP kullanmanın en önemli nedeni az protokol yüküdür. Video sunucu gibi realtime veri akışı gerektiren bir uygulama için TCP fazla yük getirir ve görüntü realtime oynamaz. Bu nedenle multicast uygulamalarında Datagram soketler kullanılır. Ayrıca video ve ses görüntülerinde genelde az bir veri kaybı sesi veya görüntüyü bozmaz. Bu nedenle sıkı paket kontrolune gerek yoktur. Eğer iyi bir fiziksel bağlantınız varsa hata oranı düşük olacaktır ve bu nedenle TCP'nin yaptığı hatalı paket kontrol işlemleri fazladan yük olacaktir.

UDP her ne kadar kendisi paket güvenliğini denetlemese de bunu yazılımcı yapabilir. Örneğin TCP bir paketi gönderdiğinde karşı tarafın onu aldığını anlamak için acknowledgement bekler. UDP bunu yapmaz. Fakat bunu soket yazılımcısı yapabilir. Yazılımcı, gönderilen her pakete bir cevap bekleyerek bunu sağlar.

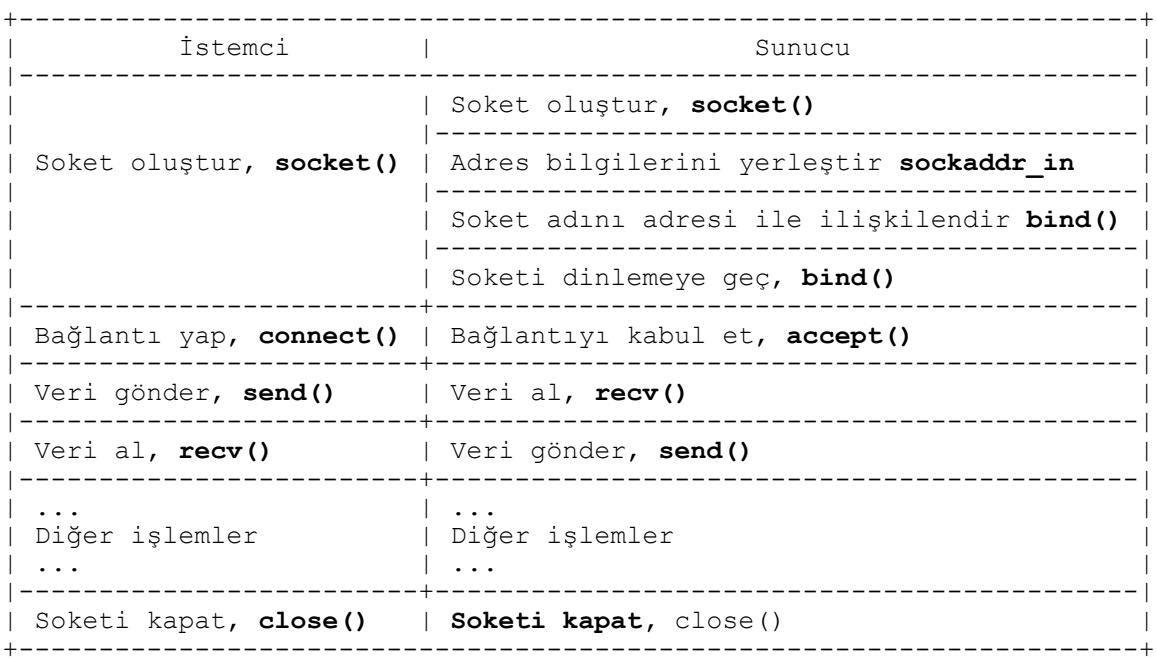
2.4. Uygulama Katmanı

TCP/IP protokolünün en üstünde yer alır. Taşıma katmanının sağladığı UDP ve TCP protokollerini kullanarak veri aktarımı yapabilirler. Telnet, FTP, SMTP, HTTP uygulama katmanı protokolleridir.

3. Uygulamadaki Temel Prensipler

Soketler her zaman iki uca sahiptir: Alıcı ve gönderici. Bütün mesajlar ve protokol gereği olan başlıklar nihayetinde fiziksel katmandan, mantıksal 1 ve 0'a karşılık gelen elektriksel sinyaller olarak gönderilir.

Soket program ya istemci, yada sunucudur. Programları daha karışık olmakla beraber bazı soket programlar her iki görevi de yapmaktadır. Sunucu program ile istemci program arasında çalışma olarak bazı farklar vardır. Aşağıdaki tablo her iki tarafta olayların nasıl gittiğini göstermektedir:



3.1. Soket Türleri

Tanımlı pek çok soket türü vardır. Ancak u dökümanda en çok kullanılan 3 türden bahsedilecektir.

SOCK_STREAM: Sıralı, güvenli, iki yollu, bağlantı yönelimli (connection oriented) veri akışı sağlar. Veri alışverişinden önce bağlantı (connect) yapılmış olmalı. Paketin kaybolmadığını veya birden çok gelmediğini uygulama katmanına garanti eder. Kabul edilebilir bir süre içerisinde veri transferi sağlanamazsa bağlantı yok varsayılır.

SOCK_DGRAM: Datagram bağlantısı sağlar. Datagramlar, bağlantısız (connectionless), güvenilir olmayan paketlerdir.

SOCK_RAW: TCP/IP ye raw olarak (herhangi bir format ve sınır olmadan) ulaşmayı sağlar.

Raw soketler başlı başına bir konu olup detaylı bilgi için Murat Balaban'ın yazdığı http://www.acikkod.org/yayingoster.php?id=34 dökümanından faydalanabilirsiniz. Datagram ve stream soketleri 2.3. Taşıma Katmanı kısmında tartışıldı.

3.2. Veri Dönüşümleri

İnsanların soldan sağa veya sağdan sola alfabelere sahip olmaları gibi işlemciler de byte'ları saklarken önemli byte'ın solda veya sağda olmasına göre sınıflandırılır. Buna endianness da denir. Arap rakamlarında olduğu gibi (İngilizce veya Türkçede kullandığımız rakamlar) önemli byte'ın solda olduğu sıralamaya big-endian denir. Önemli byte'ın en sağda olduğu sıralama ise Little Endian olarak adlandırılır.

Bütün işlemciler kendi sıralamasını seçmiştir. i386 ve klonu olan işlemciler little endian'dır. Sun Sparc, Motorola 68K ve PowerPC big endian kullanır. Java Sanal İşlemcisi (Java VM) de big endian kullanır.

Farklı iki işlemcisi olan makineler birbirileri ile haberleşecekleri zaman (IPC), bu veri dönüşümünü yapmazlar ise haberleşemezler.

Ağ protokolleri de kendi sıralamasını seçmelidir. Aksi takdirde iki farklı mimarideki bilgisayar IPC yaparak birbirileri ile haberleşecekleri zaman anlaşamayacaklardır. TCP/IP big endian sıralamasını kullanır. Bunun anlamı şu: Herhangi bir paket (IP adresi, paket uzunluğu, kontrol değeri gibi) gönderileceği zaman en önemli byte'ı önce gönderilir ve alınır.

İki tür byte sıralaması vardır. En önemli byte'ın önde geldiği sıralama ki buna *Network Byte Sıralaması* denir ve önemli byte'ın sonra geldiği dıralama. Buna da *Host Byte Sıralaması* denir. Bu ikisi arasındaki dönüşümler aşağıdaki dört fonksiyon tarafından yapılmaktadır:

```
#include <netinet/in.h>

uint32_t htonl(uint32_t hostlong); /* Host to Network Long */
uint16_t htons(uint16_t hostshort); /* Host to Network Short */
uint32_t ntohl(uint32_t netlong); /* Network to Host Long */
uint16_t ntohs(uint16_t netshort); /* Network to Host Short */
```

Eğer iki host da little endian ise veri transferinden önce network byte sırasına çevirilmeli. Alınan tarafta tekrar little endian'a çevrilir.

Şu senaryoyu dikkatle inceleyelim inceleyelim: Intel bir makine internet üzerinden SPARC makine ile haberleşecek. Veri olarak 192.168.1.254 gibi bir IP adresini düşünelim. Intel işlemci little endian olarak bunu 0xFE01A8C0(254 1 168 192) şeklinde tutar ve FE-01-A8-C0 sırasıyla bunu gönderir. SPARC makine bunu aynı sırada alacak ve bu sırada kullanacak. Çünkü big endian kullanır ve önemli olan en başta gelir. Dolayısıyla SPARC bu IP adresini 254.1.168.192 olarak algılar.

Programcı hostlarda hangi sıralamanın kullanıldığını bilmek zorunda değil. Yukarıdaki dönüştürücü fonksiyonlar kendi işlemcileri ile ilgili dönüşümleri otomatik olarak yapacaktır.

Veriler ağ üzerinde network byte sırası ile dolaşacaktır. Veriyi alan host makina ntohl ile bu veriyi kendi anladığı sıraya çevirecektir.

3.3. Sistem Çağrıları

Adres ataması:

Gerek sunucu, gerekse istemci internet adres bilgilerini tutmak için sockaddr_in yapısını kullanır. Bu yapının açılımı şu şekildedir:

```
struct sockaddr_in {
          short           sin_family;
          unsigned short  sin_port;
          struct in_addr  sin_addr;
          char            sin_zero[8];
}
```

Bu yapı içerisinde bir başka yapı daha vardır. Bu yapı in_addr yapısıdır ve açılımı şöyledir:

```
struct in_addr {
  union
  {
        struct { u_char s_b1, s_b2, s_b3, s_b4;
        } S_un_b;
        struct { u_short s_w1, s_w2; } S_un_w;
        u_long S_addr;
  }
  S_un;
}
```

Bu yapıları doğrudan kullanmayacağız; o nedenle rahat olun. Ancak ilk yapının temel yapımız olduğunu unutmayın. Bu yapı içerisinde IP adreslerini ve port numaralarını saklayacağız.

İnsanlar bir bağlantı gerçekleştirecekleri zaman genelde host isimlerini kullanırlar. Oysa internetteki

hiçbir makine host ismini kullanmaz. "cclub.ktu.edu.tr" bir host ismidir. 193.140.168.77 bu hostun sahip olduğu IP adresidir. Host isimlerini IP adreslerine çeviren sistemlere DNS (Domain Name Server) denir. İşletim sistemi bize DNS işlemlerini yapmak için kütüphane sunmaktadır. IP dönüşümü yapmak için kütüphane çağrılarından gethostbyname() 'i kullanabiliriz.

```
#include <netdb.h>
struct hostent* host;
host = gethostbyname("cclub.ktu.edu.tr");
```

Soket Oluşturma:

İki programın haberleşmesi için öncelikle her iki tarafta da soket açılması gerekir.

int socket(int domain, int type, int protocol);

socket() fonksiyonunun ikinci parametresi soketin tipidir. Buraya:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM
SOCK_PACKET

gibi soket türlerini yazabiliriz. İlk ikisini yukarıda anlattım. Diğerlerinin ne anlama geldiği şu aşamada önemli değildir. Yalnızca biz SOCK_STREAM tipini kullandığımız için ikinci parametre olarak bunu verdiğimizi bilin. Yani bu bir stream sunucu soket uygulamasıdır.

Üçüncü parametresi protocol tipini belirler. Hali hazırda 5 protokol türü vardır :

AF_UNIX (UNIX internal protocols)
AF_INET (ARPA Internet protocols)
AF_ISO (ISO protocols)
AF_NS (Xerox Network Systems protocols)
AF_IMPLINK (IMP "host at IMP" link layer)

Soket ile ilgili tanımlamaların başlık dosyaları `<sys/types.h>` ve `<sys/socket.h>` dir.

Veri Alış-verişi:

Birbiri ile bağlantısı yapılmış iki soket arasında artık veri alış-verişine başlayabiliriz. Buradaki veriler karşıya taşınması gereken gerçek veriler olabileceği gibi, karşıya yapması gereken işleri bildiren bir komut listesi olabilir. Her şey, iki program arasında tarafınızdan tanımlanmış protokol çerçevesinde olacaktır.

write() ve read() fonksiyonları aynı zamanda dosya işlemleri için de kullanılır. send() ve recv() ise bu fonksiyonların soketler için özelliştirilmiş halidir.

```
int send(int s, const void *msg, int len, unsigned int flags);
int recv(int s, const void *msg, int len, unsigned int flags);
```

4. Sunucu Soket Programı

4.1. Sistem Çağrılarının Kullanımı

Bu kadar teorik bilgiden sonra bir uygulama yapmaya geçebiliriz. 2222 numaralı portta çalışan basit bir sunucu programı üzerinde olayları inceleyelim. Bundan sonra üzerinde çalışacağımız kodlar sunucu için yazılmaktadır. İstemci program olarak standart telnet programını kullanacağız. Mimari olarak sunucu yazılımların, istemci yazılımlardan farklı olduğunu unutmayın. Aşağıdaki kodları gcc C derleyicisi kullanarak yazdım. Ancak bu kodlar tamamen standart olup Windows altındaki Visual C derleyicisi ile de çalışabilir. Değişik platformlarda sorun yatmasın diye kod içerisinde Türkçe karakter kullanmadım.

Önce port numaramızı belirleyelim:

```
#define PORT 2222
```

Sıra soket oluşturmada:

```
int sockfd;
sockfd=socket(AF_INET,SOCK_STREAM,0);
if (sockfd<0) {
  perror("socket");
  exit(1);
}
```

Burada verdiğim kodlar tek başına çalışmaz. Daha kolay anlaşılsın diye programdan parçalar bir araya getiriyorum. En sonunda çalışabilir kodun tamamını vereceğim. Bu işlemlere başlamadan önce bazı header (.h) dosyalarının yüklenmesi gerekir. Bunları da kodun tamamında görebilirsiniz.

Yukarıdaki kodu incelersek: Soket oluşturma işlemini socket() fonksiyonu ile yapıyoruz.

socket() fonksiyonu geriye bir tamsayı değer döndürür. Hata oluşumunda bu değer -1'dir. Eğer hata oluşmazsa geri dönen değer soketin tanımlayıcı numarasıdır. Bu numara en başta bahsettiğimiz gibi dosya tanımlayıcısıdır. Bu numarayı kodda görüldüğü gibi sockfd değişkenine atatım. Bu işlemden sonra soketi takip etmek için artık sockfd değişkenini kullanacağım. Oluşturulan her soket bu şekilde ayrı bir numara alır.

Hata kontrolü yapmaya özen gösterin. Aksi halde zamanızın büyük bir kısmını programdaki hataları ayıklamak ile geçirirsiniz.

Şimdi port numarasının atamasını yapalım:

```
if(bind(sockfd,(struct sockaddr *)&my_addr,sizeof(struct sockaddr)) ==
-1) {
  perror("bind");
  exit(1);
}
```

bind() fonksiyonu soketi isimlendirir. socket() ile oluşturulmuş bir soket için isim uzayında (bellekte) vardır ancak bir isme sahip değildir. Bind işlemi belirtilen port numarasını ve gerekli sistem kaynaklarını işletim sisteminden ister. Eğer bu işlemi yapmazsanız işletim sistemi port havuzundan herhangi bir port atar. Port numarasını belirlemezseniz, diğer insanlar sizin programla haberleşecek programlar yazamaz. bind() fonksiyonu aynı zamanda sizi bir veya daha fazla ağ

arabirim kartından (Network Interface Card) yalıtır. Ağ üzerindeki her host bir NIC'a sahiptir ve her NIC en az bir IP adresine sahiptir. addr.sin_addr değeri olarak INADDR_ANY yazmakla işletim sistemine host üzerinde bulunan mümkün bütün IP adreslerinden bağlantı kabul edeceğinizi söylüyorsunuz. Bunu istemiyorsanız makinenin sahip olduğu IP numaralarından birini, hizmet vermek üzere seçebilirsiniz. Eğer üzerinde bulunduğunuz makine bir firewall ise belirlenmiş bir IP'yi kullanmak yararınıza olacaktır. Bu şekilde firewall'un yalnızca bir yüzü hizmet sunacaktır.

```
if (listen(sockfd, 5) == -1) {
  perror("listen");
  exit(1);
}
```

bind() işleminden sonra sunucumuz artık istemcileri beklemeye başlayabilir. İstemcilerin kabul edilebilmesi için sunucunun öncelikle dinlemeye geçmesi gerekir. Bunun için listen() fonksiyonunu kullanır. Bu çağrı, soketin özelliklerinde bazı değişiklikler yapar. Örneğin soketi dinleme moduna geçer ve bu soketi veri taşıması için kullanamazsınız. listen(), çağrıldıktan sonra parametre olarak belirttiğiniz kadar bir bekleme kuyruğu oluşturur. Yukarıda parametre olarak 5 verdik. Bunun anlamı: Sunucu mevcut bağlantıya hizmet verirken gelecek ilk 5 istek bekleme kuyruğuna konulacak. Sunucu sırayla bunlara hizmet verecek.

Artık sunucu bağlantı kabul etmek için soketi dinlemeye almıştır. Herhangi bir istemcinin bağlantı isteğini accept() çağrısı ile yakalayacağız. Kabül fonksiyonu programı G/Ç bekleme durumuna sokar. Dolaysıyla programımız işletim sistemi tarafından bloke edilir ve bağlantı isteği geldiğinde tekrar uyandırılır.

```
int accept(int s, struct sockaddr *addr, int *addrlen);
```

Kabül işlemi, gelen isteği yeni bir soket ile karşılar. Çünkü mevcut soketimiz dinleme durumundadır ve veri taşıma işlemi gerçekleştiremez.

Şimdi programımızın tamamlandığını ve cclub.ktu.edu.tr hostunda çalıştığını varsayalım.

```
cclub:~> telnet cclub.ktu.edu.tr 2222
```

Geçici olarak telnet programını istemci olarak kullandım. "telnet" i bir echo istemcisi gibi düşünün. Sunucu ne gönderirse onu ekrana basar. 2222 portuna bağlantı isteği gönderdiğimizde sunucu programımız bize yeni bir soket açıp bağlantımızı kabul edecek. Telnet ile sunucumuzun konuşmaması için hiçbir neden yok.

İstemcilerin adres bilgileri, bağlantı kabülü sırasında alınır. accept() fonksiyonun son parametresi bir girdi değil çıktıdır. Bağlantı sırasında bu yapı doldurulup programa döndürülür. Biz programa dönen adres bilgilerini kullanarak programlarımıza biraz renk katabiliriz. Örneğin, istemcinin adresini alıp bir yasak dosyasında var mı diye kontrol ederiz. Eğer varsa bağlantıyı red ederiz. Bağlantıları kısıtlayan /etc/hosts.deny ve serbestlik tanıyan /etc/hosts.allow dosyalarını hatırlayın. Sanki bir şeyler hatırlar gibi olduk değil mi?

```
int fd, client_size;
struct sockaddr_in client_addr;
client_size = sizeof(struct sockaddr_in);
fd = accept(sockfd, (struct sockaddr *)&client_addr, &client_size));
printf("Merhaba %s",inet_ntoa(client_addr.sin_addr));
```

Sunucumuza bir bağlantı yapıldığında sunucu ekrana "Merhaba 193.140.168.54" gibi bir şey yazacak. Gördüğünüz gibi bağlantı yapanın adresini de aldık.

4.2. sunucu.c

```
#include <stdio.h>
#include <string.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/wait.h>

#define PORT 3333
#define LISTQUEUE 5

main(int argc, char *argv[]) {
  int sockfd, new_fd;
  struct sockaddr_in server_addr, client_addr;
  int client_size;
  char buffer[1024];

  printf("%s %d portu uzerinde calismaya basladi...\n\n",argv[0],PORT);

  if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    perror("socket");
    exit(1);
  }

  memset(&server_addr, 0, sizeof(server_addr));
  server_addr.sin_family = AF_INET;
  server_addr.sin_port = htons(PORT);
  server_addr.sin_addr.s_addr = INADDR_ANY;

  if(bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) {
    perror("bind");
    exit(1);
  }
  if(listen(sockfd, LISTQUEUE) != 0) {
    perror("listen");
    exit(1);
  }

  while(1) {
    client_size = sizeof(struct sockaddr_in);
      if((new_fd=accept(sockfd,(struct  sockaddr*)&client_addr,&client_size))  ==
-1) {
      perror("accept");
      exit(1);
    }
    printf("%s sunucumuza baglandi...\n",inet_ntoa(client_addr.sin_addr));

    memset(&buffer, 0, sizeof(buffer));
    strcpy(buffer,"Merhaba ");
    strcat(buffer,(char *) inet_ntoa(client_addr.sin_addr));
    strcat(buffer," :)\n");

    if(send(new_fd,&buffer, strlen(buffer), 0) == -1) {
      perror("send");
      exit(1);
    }
    if(recv(new_fd,&buffer, strlen(buffer)-1, 0) == -1) {
      perror("recv");
      exit(1);
    }
    printf("Alinan yanit: %s\n", buffer);
    close(new_fd);
  }
  close(sockfd);

  return 0;
}
```

Evet. İşte harika dünyaya adım attık ve "Merhaba" dedik. Bunu ilk denediğimde gözlerime inanamamıştım. Başka bir bilgisayardaki kişiye merhaba demiştim. O zamanlar chat programları bu kadar populer değildi. Henüz mirc yokken, irc'a müptela olduğumuz günler. Bu gerçekten harika bir şey.

Şimdi programımızı linux altında "GNU C Compiler" gcc ile nasıl derleyeceğimizi görelim.

```
cclub:~> gcc -o sunucu1 sunucu.c
```

Bu komutla sunucu.c dosyasına yazdığımız kodları derleyip sunucu1 isimli çalışabilir dosyayı ürettik. Bakalım neler oluyor:

```
cclub:~> ./sunucu1
./sunucu1 2222 portu uzerinde calismaya basladi...
193.140.168.54 sunucumuza baglandi...
```

Tahmin edeceğiniz gibi bunlar sunucunun ekranına yazanlar. "193.140.168.54 sunucumuza baglandi..." mesajı, ceng hostundan bağlantı yapıldığında ekrana basılmıştır. Peki istemcinin ekranına neler dönüyor:

```
ceng:~# telnet 193.140.168.77 2222
Trying 193.140.168.77...
Connected to 193.140.168.77.
Escape character is '^]'.
Merhaba 193.140.168.54 :)
```

Merhaba 193.140.168.54 :) mesajı, sunucu tarafından istemcimize (telnet) gönderilen mesajdır. Onun üstündeki mesajları merak etmeyin. Onların bizimle ilgisi yok. Telnet'in ürettiği mesajlardır. Windows makineden "Başlat->Çalıştır" kullanarak aynı komutu verebilirsiniz.

5. İstemci Soket Programı

Şimdi istemci bir soket programın genel yapısına bakalım. Başlangıç aşamasında işlemler sunucu program ile aynı olacaktır. Ancak asıl iş yapılan kısımda hem mimari, hem de fonksiyonlar değişeek. Çünkü bir taraf hizmet verme, öteki taraf ise hizmet alma durumunda olacaktır. Şimdi geliştireceğimiz istemci yukardaki sunucu için geliştirilmemiştir. Bu programımız 3333 numaralı portta çalışsın. Bu istemci program için bir de sunuu program yazdım. Sunucu programın çalışmasını yukarıda anlattığımdan tekrar bu program için de anla. Bu programımız 3333 numaralı portta çalışsın. Bu istemci program için bir de sunuu program yazdım. Sunucu programın çalışmasını yukarıda anlattığımdan tekrar bu program için de anlatmayacağım. Doğrudan kodunu vermekle yetineceğim.

Öncelikle port numaramızı belirleyelim:

#define PORT 3333

Sunucunun çalıştığı bilgisayar istemciye parametre olarak verilecek:

```
int main(int argc, char *argv[])
{
  if (argc != 2) {
    printf("Kullanimi : %s hostname ",argv[0]);
    exit(1);
  }
}
```

Sunucunun Belirlenmesi:

Parametre olarak alınan host ismini çözümlüyoruz. Adres yapısında olan server_addr değişkenine PORT numarasını ve IP adresini yazıyoruz.

```
struct hostent *h_name;

if(argc > 1) {
  h_name = gethostbyname(argv[1]);
}
else {
  printf("Kullanimi: %s hostname ",argv[0]);
  exit(1);
}

bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = *(u_long *) h_name->h_addr;
serv_addr.sin_port = htons(PORT);

printf("Sunucu adresi: %s ",inet_ntoa(serv_addr.sin_addr));
```

Sunucuya Bağlanma:

Artık hizmetkarımıza, yani sunucumuza bağlanıp selam verme vakti geldi.

```
printf("Mesajinizi girin: ");
fgets(buf, sizeof(buf)+1, stdin);

if((send(sockfd, buf, sizeof(buf), 0)) <= 0) perror("send");
if((recv(sockfd, buf, sizeof(buf), 0)) <= 0) perror("recv");
printf("Sunucu'dan gelen mesaj: %s", buf);

shutdown(sockfd, 2);
close(sockfd);
```

Aynen sunucu programda olduğu gibi soket açtık. Soket türümüz yine SOCK_STREAM. Sonra connect()fonksiyonunu kullanarak açtığımız soket üzerinden bir bağlantı yaptık. connect() fonksiyonu sys/types.h başlık dosyasında aşağıdaki gibi tanımlanmıştır:

```
int connect(int s, const void *addr, int addrlen);
```

İlk parametre soket tanımlayıcısı, ikinci parametre bağlantı kurulacak sunucunun adres bilgilerini içeren server_addr yapısı, son parametre ise adres yapısının boyutudur. Artık sunucumuzla aramızda stream bir soket bağlantısı kurmuş olduk. İstemci tarafında connect() yapıldığı zaman, sunucu tarafında accept() yapılır. Yani yapılan bağlantı kabul edilir. Daha önce yazdığımız sunucu programdaki accept() işlemi, telnet programının connect() isteğini karşılıyordu. Şimdi bizim istemimiz bu isteği gönderecek. Dikkat ederseniz ilkel bir telnet programı yazıyoruz.

Sunucuya Bilgi Gönderme:

Sonraki kısmı, anlaşılması açısından basit tuttum. Ekrandan bir mesaj alıp bunu send() fonksiyonu ile sunucuya gönderiyoruz. Bu fonksiyon, dosyalara yazmak için kullandığımız write() fonksiyonu gibidir. send() ile sokete veri gönderiyoruz. Soket sunucu ile bağlantılı olduğundan veri, sunucuya ulaşır. Sunucu ise her send() isteğine recv() ile karşılık verir. send() yada recv() işlemleri her iki tarafta da yapılabilir. Sunucudan gelen her send() isteğine karşı da istemci de bir recv() vardır. Örneğin ftp programında karşılık send() ve recv() işlemleri vardır. Siz önce istemci tarafından 'get dosya' komutunu göndererek dosya isteğinizi belirtiyorsunuz. Sunucu bu komutu okuyup

yorumluyor ve size dosyayı ftp protokolune uygun paketler halinde gönderiyor. Artık siz okumaya başlıyorsunuz. Paketleri alıp birleştirip yerel diske yazıyorsunuz. Bu aşamadan sonra istediğinizi yaptırabilirsiniz.

istemci.c

Aşağıdaki bir SMTP sunucuya bağlanıp smtp komutları gönderir ve cevaplarını alır. Tam olarak bir SMTP istemci değildir. SMTP protokolunun detaylari için RFC 821'e bakabilirsiniz.

```
#include <stdio.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/wait.h>
#include <sys/socket.h>

#define SMTP_PORT 25

int main(int argc, char *argv[])
{
  struct hostent *h_name;
  int sd;
  char cmd[512];
  struct sockaddr_in serv_addr;

  printf("checkd - SMTP sunucuyu kontrol eder.\n");
  printf("http://www.acikkod.org/\n\n");

  if(argc < 2) {
    printf("Kullanımı: %s hostname\n",argv[0]);
    exit(1);
  } else {
    h_name = gethostbyname(argv[1]);
  }

  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = *(u_long *) h_name->h_addr;
  serv_addr.sin_port = htons(SMTP_PORT);

  printf("> Soket açılıyor... ");
  if( (sd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    printf("hata!\n");
    perror("socket");
    exit(1);
  } else printf("tamam\n");

  printf("> %s bağlanılıyor... ",inet_ntoa(serv_addr.sin_addr));
  if(connect(sd,(struct sockaddr*)&serv_addr,sizeof(serv_addr)) < 0) {
    printf("hata!\n");
    perror("connect");
    exit(1);
  } else printf("tamam\n");

  printf("> Karşılama mesajı alınıyor... ");
  memset(cmd, 0, 256);
  if((recv(sd, cmd, 256, 0)) <= 0) {
    printf("hata!\n");
    perror("recv");
    exit(1);
  } else printf("tamam\n");

  printf("  %s\n", cmd);
```

```
  printf("> Selam veriliyor... ");
  memset(cmd, 0, 256);
  strcpy(cmd, "helo world\n");
  if((send(sd, cmd, 256, 0)) <= 0) {
    printf("hata!\n");
    perror("send");
    exit(1);
  } else printf("tamam\n");

  printf("> Cevap alınıyor... ");
  memset(cmd, 0, 256);
  if((recv(sd, cmd, 256, 0)) <= 0) {
    printf("hata!\n");
    perror("recv");
    exit(1);
  } else printf("tamam\n");

  printf("  %s\n\n", cmd);

  printf("SMTP sunucunuz çalışıyor.\n\n");

  return 0;
}
```

6. Gelişmiş G/Ç

6.1. Bloksuz G/Ç

Giriş/Çıkış (I/O); bir dosyaya, pipe'a, terminale veya ağ aygıtına yazmak veya bu aygıtlardan okumak gibi işlemleri içermektedir.

Okunacak veri hazır değilse veya yazılacak veri o an kabul edilmiyorsa bu işlemleri yapan süreçler bloklanacaktır. Bloksuz G/Ç (Nonblocking I/O), verinin veya aygıtın hazır olmaması gibi durumlarda bloklanmayı tamamen ortadan kaldıran bir özelliktir. Bu konunun detayları dökümanın kapsamı dışındadır. (man 2 fcntl)

6.2. select()

Soket program aynı anda birden fazla soketten veri okumak veya yazmak durumunda kalabilir. Bunu tek soket tanımlayıcı ile sağlayamazsınız. Çünkü soketiniz blok durumuna geçtiğinde (örneğin accept(), bağlantı gelene kadar programın blok olmasına neden olur) kodunuzun geri kalan kısmı çalışmaz, bloktan çıkmayı bekler.

Şöyle bir senaryoyu hayal edelim. İki adet soket tanımlayıcı var ve bu ikisi ile karşı uçtan dosya veri alacaksınız. 1. uç henüz veriyi hazırlamadı ancak 2. uç hazırladı varsayalım. Eğer program 1. soket tanımlayıcı ile ilk önce 1. uçtan veri çekmeye çalışırsa blok olacaktır. 2. ucun verisi hazır olduğu halde veri alınamayacaktır. Oysa select() ile bu iki uç kontrol edilip hazır olan uçtan -senaryomuzda 2. uç- veri okunsa idi program blok olmayacak ve verisi hazır olan işlerini yapmaya devam edecekti. Bu şekilde bloklanma riski taşımayan bir program yazılabilir.

Tek bir süreç içerisinde bloksuz G/Ç kullanarak bu sorun çözülebilir. Bütün dosya tanımlayıcılar bloksuz G/Ç yapacak şekilde set edilir. Eğer veri hazır değilse read() hemen sonlanır. Aynı işlemi ikinci dosya tanımlayıcısı için de yaparız. Belli bir süre sonra tekrar ilk tanımlayıcıyı kontrol ederiz. Buna '*polling*' deniliyor. Bunun dezavantajı, gereksiz yere CPU zamanı harcamaktadır.

Aynı problemi çözmek için kullanılabilecek bir diğer teknik de "*asenkron G/Ç*". Bu yöntemde, dosya tanımlayıcımız G/Ç için hazır olduğunda çekirdek, G/Ç yapacak süreci haberdar edecektir. Ancak burada standart problemleri ve sinyalleri işleme (signal handling) ile ilgili problemler vardır.

Bunlardan daha iyi bir teknik ise G/Ç çoğullama (I/IO multiplexing) olarak adlandırılmaktadır. İlgilendiğimiz tanımlayıcıların eklendiği bir küme vardır ve tanımlayıcılardan biri G/Ç için hazır

olmadıkça çıkmayan bir sistem çağrısı yapılır. Sistem çağrısından çıkıldığında hangi tanımlayıcıların G/Ç için hazır olduğu sorulabilir.

select birden fazla soket tanımlayıcının durumunu takip eden ve BSD4.2 ile gelen bir sistem çağrısıdır. Burada şunu belirtmeliyim ki, select() yalnızca soketler için değil genel olarak dosya ve G/Ç işlemleri için kullanılan bir sistem çağrısıdır.

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>


int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct
timeval *timeout);
```

n: En büyük soket tanımlayıcının bir fazlası (tanımlayıcılar 0'dan başladığından)
readfds: Okumak için izlenecek (hazır olup olmadığı bakılacak) soket tanımlayıcı
writefds: Yazmak için izlenecek soket tanımlayıcı
exceptfds: İstisnai durumlar için izlenecek soket tanımlayıcı

timeval, aşağıdaki şekilde bir veri yapısıdır:

```
struct timeval {
  long tv_sec; /* seconds */
  long tv_usec; /* microseconds */
};
```

timeout eğer NULL olarak verilirse bir dosya hazır olana kadar blokda bekler. Hazır olmazsa sonsuza kadar bekleyecektir. Tabi sinyal(signal) gonderilirse sonlanır. Eğer sinyal ile sonlandırılırsa select() -1 döndürecektir ve errno=EINTR olacaktır.

'0' a setlenirse hiç beklemez. Bütün belirtilen tanımlayıcılar test edilir ve çağrıdan hemen çıkılır. Bu, select içinde blocklanmadan birden fazla tanımlayıcıyı test etmek için kullanılır.

0'dan büyük bir değere setlenirse o değer kadar bekler. Belirtilen tanımlayıcılardan biri hazır olduğunda veya zamanaşımına uğradığında sistem çağrısı return yapar.

Gözetlenecek tanımlayıcılar tanımlayıcı setine (descriptor set) eklenir. Tanimlayıcı seti, fd_set veri yapısı şeklinde kayıt edilmiş veridir. fd_set, her tanımlayıcı için bir bit tutar ve geçerli boyutu 1024 bittir (sys/types.h içinde tanımlı). Set üzerinde işlem yapmak için bazı makrolar tanımlanmıştır:

fd_set aşağıdaki gibi tanımlanır:

```
fd_set dset;
```

Seti sıfırlamak için:

```
FD_ZERO(&dset);
```

Takip etmek istediğimiz tanımlayıcıları eklemek için:

```
FD_SET(fd, &dset);
```

select() seti değiştirmektedir. select sistem çağrısından sonra bir tanımlayıcının hala sette olup olmadığını test etmek için:

```
if (FD_ISSET(fd, &dset)) {
   ...
}
```

Bir tanımlayıcıyı setten çıkartmak için:

```
FD_CLR(fd, &dset);
```

Bu bilgiler ışığında bir uygulama geliştirebiliriz. Standart giriş (stdin), dosya tanımlayıcısı 0 olan bir dosyadır. Eğer bir PC kullanıyorsanız klavye standart giriştir. Aşağıdaki kod, standart girişi izlemektedir. Eğer standart giriş okumak için hazırsa (bunun anlamı klavyeden birşey yazıp enter'a basmışsak) hazır olduğunu ekrana basacak. FD_ISSET ile de verinin hazır olduğunu göreceğiz (Hazır olduğunda FD_ISSET, true olacaktır. Eğer hazır değilse false olacaktır.).

```
/*
 * select.c - I/O multiplexing
 *
 * Baris Simsek, <simsek at acikkod org>
 * http://www.acikkod.org
 * 07/07/2004
 *
 */

#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int
main(void) {
  fd_set dset;
  struct timeval tv;
  int ret;

  FD_ZERO(&dset);
  FD_SET(0, &dset); /* stdin'i gozlemeye aldik */
  tv.tv_sec = 10; /* 10 sn giris icin bekle */
  tv.tv_usec = 0;

  ret = select(1, &dset, NULL, NULL, &tv);

  if (ret == -1)
    perror("select()");
  else if (ret) {
    printf("Standart input okumak için hazir.\n");
    if(FD_ISSET(0, &dset))
      printf("Standart input icin dset true\n");
  }
  else {
    printf("10 saniye icinde standart giristen veri girilmedi.\n");
    if(!FD_ISSET(0, &dset))
      printf("Standart input icin dset false\n");
  }
  return 0;
}
```

7. Sık Sorulan Sorular

7.1. Kitap önerebilir misiniz?

W. Richard Stevens'ın "UNIX Network Programming - Volume 1" kitabı bu alanda başlıca referans kitaptır.

7.2. Soketler Nasıl Çalışır?

Soketler (özelikle connection oriented soketler) dosyalar veya PIPE gibi çalışır. Pipe'dan farkı iki yönlü olmasıdır. Dosyalardan farkı ise beklediğiniz kadar veri okuyamayabilir veya istediğiniz kadar veri yazamayabilirsiniz.

7.3. select() veri hazır dediği halde, 0 byte neden okunur?

select() verinin hazır olduğunu söyledikten sonra karşı taraf bağlantıyı koparmıştır. Bu da read() 'in 0 döndürmesine neden olur.

7.4. Soket seçeneklerini nasıl değiştiririm?

int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);

```
int flag = 1;
int result = setsockopt(sock, /* socket affected */
                        IPPROTO_TCP, /* seçeneği TCP seviyesinde set et. */
                        TCP_NODELAY, /* seçenek */
                        (char *) &flag,
                        sizeof(int)); /* seçenek değerinin büyüklüğü */
```

Seçenekler hakkında detay için "man 2 setsockopt"

7.5. SO_KEEPALIVE seçeneği neden kullanılır?

Oturum açmış iki bilgisayar arasında belli bir süre (RFC1122 de 2 saat olarak tanımlandı) veri alışverişi olmazsa, karşı tarafı yoklamak için (ACK isteyerek) kullanılır. Zira karşı taraf ulaşılamaz durumda olabilir. Bu durumu algılamak için kullanılır.

7.6. Soketi tam olarak nasıl kapatırım?

Soket kapatıltıktan sonra "netstat -na" ile bakıldığında sisteminizde TIME_WAIT'de duran soketler hala gözükebilir. Bu normaldir. Çünkü TCP, bütün verinin karşılıklı transfer edildiğinden emin olmak istiyor. Soket kapatıldığında her iki taraf da başka veri transferi olmayacağı konusunda anlaşmış demektir. Böyle bir anlaşmadan sonra socket rahatlıkla kapatılabilir. Ancak burada iki problem sözkonusu olacak. Bunlardan biri son ACK'in ulaşıp ulaşmadığı bilinemeyecek. (Bu ACK için tekrar ACK istense bunun da ulaşıp ulaşmadığı belli olmayacak. Kısır döngü olur.). Eğer bu ACK ağda kaybolmuş ise sunucu bunu bekliyor olacaktır. Bir diğer problem de eğer bir paket router'in birinde herhangi bir nedenle bekliyorsa ve alıcı taraf bunu belli bir süre içerisinde alamamışsa paketi yeniden talep edecektir. Ancak diğer paket gerçekte kaybolmamıştır ve belli bir süre ağda yeniden ortaya çıkacaktır. Bu kaybolma ve yeniden ortaya çıkması süresi içerisinde bağlantı koparsa ve aynı host aynı porttan yeni bir bağlantı açarsa göndereceği paketin sıra numarası ağdaki ile üst üste binecektir. Çünkü eski oturumdan kalma bir paket yeni oturumda transfer edilmiştir. Bundan kurtulmak için TIME_WAIT durumu ortadan kalkmadan yeni bir oturum açmamalı.

Bütün bunlar düşünüldüğünde TIME_WAIT'in programcı için bir yardımcı olduğu anlaşılır. Ancak TIME_WAIT olduğu sürece programınız aynı soketi yeniden bind() edemeyecektir. 7.4. anlatıldığı şekilde SO_REUSEADDR seçeneği set edilerek bu sorunu çözebilirsiniz. Öte yandan TIME_WAIT'te bekleyen soketler bir süre sonra (Linux'lerde bu 60 sn.dir.) close() olacaktır. sysctl ile bu süre değiştirilebilir.

close() doğru kapatma yöntemi ise de shutdown() daha kullanışlıdır. Çünkü tek yönlü soketi kapama olanağı da sunar.

int shutdown(int s, int how);

İkinci parametre ile kapa yönünü verebilirsiniz:

SHUT_RD: Veri alımı kesilecektir.
SHUT_WR: Veri gönderimi kapatılacaktır.
SHUT_RDWR: İki yönlü veri alışverişi durdurulacaktır. (close)

close(), o süreç için soketi kapatır ancak eğer socketi başka bir süreçle paylaşıyorsa socket hala açık duracaktır. shutdown() bütün süreçler için soketi kapatır.

7.7. String halindeki bir adresi internet adresine nasıl çevirim?

```
struct in_addr *atoaddr(char *address) {
  struct hostent *host;
  static struct in_addr saddr;

  /* Önce IP formatında deniyoruz. */
  saddr.s_addr = inet_addr(address);
  if (saddr.s_addr != -1) {
    return &saddr;
  }

  /* IP formatında değilse FQDN olarak deniyoruz. */
  host = gethostbyname(address);
  if (host != NULL) {
    return (struct in_addr *) *host->h_addr_list;
  }
  return NULL;
}
```

7.8. Soketlerde dinamik buffer kullanmanın bir yolu var mı?

Soketten okuyacağınız veri miktarı belli olmadığında böyle bir ihtiyaç doğuyor. Bu durumda malloc () ile mümkün olan en büyük tampon belleği ayırırsınız. Okunan verinin büyüklüğüne göre tampon bellek realloc() ile yeniden boyutlandırılır. Zaten pek çok UNIX'de malloc() fiziksel bellekten yer ayırmaz. Sadece adres uzayını belirler. Tampona veri yazdığınızda gerçek bellek sayfaları kullanılır. Bu nedenle büyük buffer ayırmakla gereksiz kaynak kullanımına neden olunmaz.

7.9. "address already in use" hatasını neden alırım?

Port kullanılıyordur veya sunucu bir programı henüz sonlandırdınız ve socket TIME_WAIT'dedir. İkinci durum için 7.4. ve 7.5. sorularının çözümlerine bakınız. Birinci durum için aynı portta çalışan diğer socket programı durdurmanız gerekmektedir.

7.10. Programımı nasıl daemon yapabilirim?

En kolay yolu inetd ile kullanmanızdır. Diğer bir yöntem ise fork() ederek isteklere cevap vermekdir. Detay icin http://www.enderunix.org/docs/daemontr.html Programin temel iskeleti asagidaki yapiya cevirilmeli:

```
ret = fork ();
if (ret == -1) { /* fork hata verdi */
  perror ("fork()");
  exit (3);
}
if (ret > 0) exit(0); /* Ana süreç çıkar */
if (ret == 0) { /* Alt süreç devam eder */
  close (STDIN_FILENO);
  close (STDOUT_FILENO);
  close (STDERR_FILENO);
  if (setsid () == -1)  exit(1);
  /* Alt sürece ait işler */
}
```

7.11. Aynı anda birden fazla soketi nasıl dinlerim?

select() kullanın. Hangi socket veri için hazır ise onun, kullanmanıza olanak sağlar. 6.2. select() bölümüne bakınız.

7.12. 1024 ten küçük portları neden bind edemiyorum?

Güvenlik nedenleri ile 1024'ten küçük portları yalnızca yetkili kullanıcı (root) açabilir.

9. Belge Tarihçesi

VI.sürümde "7. Sık Sorulan Sorular" bölümü eklendi. (1 Ağustos 2004)

V.sürümde Soket Türleri ve Veri Dönüşümleri başlıkları eklendi. Sarmalama(Encapsulation) tanımı eklendi. (8 Temmuz 2004)

IV.sürümde Gelişmiş G/Ç başlığı eklendi. (28 Haziran 2004)

III.sürümde RFC'ler gözden geçirilerek yeniden düzenlenmiştir. İlk sürümlerdeki bazı yanlışlar giderildi. (5 Haziran 2004)

II.sürüm, Linux Kullanıcıları Derneği Liste üyeleri için yeniden gözden geçirildi. (2001)

İlk sürüm: 10 Kasım 1998 (KTÜ Bilgisayar Klubü Dergisi için yazıldı.)

Hataları çekinmeden bana bildirebilirsiniz. b$