

Distributed Systems

Dağıtık/dağıtımlı sistemler dünyanın çehresini değiştirdi. Web tarayıcınız gezegenin başka bir yerindeki bir web sunucusuna bağlandığında, basit bir formda gözüken **istemci/sunucu(client/server)** dağıtık/dağıtımlı sisteme katılıyor. Google veya Facebook gibi modern bir web hizmetiyle iletişim kurduğunuz zaman, yalnızca tek bir makineyle etkileşimde bulunmazsınız; perde arkasında, istenilen karmaşık hizmetlerden her biri sitenin belirli hizmetini sağlamak için işbirliği yapan geniş bir makine koleksiyonundan (örneğin; binlerce sayıda olabilir.) inşa edilmiştir.

Dağıtık/dağıtımlı bir sistem kurarken bir takım yeni zorluklar ortaya çıkar. Bu zorluklardan en büyüğü **başarısızlıktır(failure)**; makineler, diskler, ağlar ve yazılımların tümü zaman zaman başarısız olur, çünkü bileşenlerin ve sistemlerin "mükemmel" bir şekilde nasıl oluşturulacağını bilemeyiz (ve muhtemelen hiçbir zaman da bilemeyeceğiz). Ancak, modern bir web hizmeti oluşturduğumuzda, istemcilere hiç başarısız olmayacakmış gibi görünmesini isteriz; bu görevi nasıl başarabiliriz?

The CRUX:

BİLEŞENLERİ BAŞARISIZ OLDUĞUNDA BİLE ÇALIŞAN SİSTEMLER NASIL İNŞA EDİLİR

Her zaman için doğru çalışmayan parçalardan oluşan bir sistemi nasıl kurabiliriz?

Bu basit soru size RAID depolama dizilerinde tartıştığımız bazı konuları hatırlatmalıdır; ancak buradaki sorunlar ve çözümler daha karmaşık olma eğilimindedir.

İlginç bir şekilde başarısızlık, Dağıtık/dağıtımlı sistemlerin inşasında temel bir zorluk olsa da, aynı zamanda bir fırsatı da temsil eder. Evet, makineler arızalanır; ancak bir makinenin arızalanması, tüm sistemin arızalanması gerektiği anlamına gelmez. Bir dizi makineyi bir araya getirerek, bileşenleri düzenli olarak arızalanmasına rağmen nadiren arızalanan bir sistem kurabiliriz. Bu gerçeklik, dağıtılmış sistemlerin merkezi güzelliği ve değeridir ve Google, Facebook vb. dahil olmak üzere kullandığınız hemen hemen her modern web hizmetinin temelini neden bu olduğunu gösterir.*

İPUCU: BAĞLANTI ÖZÜNDE GÜVENİLİR DEĞİLDİR

Hemen hemen her koşulda, bağlantıyı temelde güvenilmez bir faaliyet olarak görmek iyidir. Bit bozulması, kapalı veya çalışmayan bağlantılar ve makineler ve gelen paketler için arabellek alanının eksikliği bunların hepsi aynı sonuca yol açar: paketler bazen hedeflerine ulaşmaz. Bu tür güvenilmez ağların üzerine güvenilir hizmetler oluşturmak için paket kaybıyla başa çıkabilecek teknikleri düşünmeliyiz.

Başka önemli konular da vardır. Genellikle sistem performansı kritiktir; Dağıtık/dağıtımlı sistemimizi birbirine bağlayan bir ağ ile birlikte, sistem tasarımcıları gönderilen mesaj sayısını azaltmaya ve bağlantıyı olduğunca verimli hale getirmeye çalışarak (düşük gecikme süresi, yüksek bant genişliği) verilen görevleri nasıl gerçekleştireceklerini dikkatlice düşünmelidir.

Son olarak, **güvenlik(security)** de gerekli bir husustur. Uzak bir siteye bağlanırken, uzaktaki tarafın söylediği kişinin o olduğundan emin olmak temel bir sorun haline gelir. Ayrıca, üçüncü tarafların diğer iki kişi arasında devam eden bir iletişimi izleyememesini veya değiştirmemesini sağlamak da bir zorluktur.

Bu giriş kısmında, Dağıtık/dağıtımlı bir sistemde yeni olan en temel kavramı ele alacağız: **İletişim(communication)**. Yani, Dağıtık/dağıtımlı bir sistemdeki makineler birbirleriyle nasıl iletişim kurmalı? Mevcut en temel ilkel yollarla, mesajlarla başlayacağız ve bunların üzerine birkaç üst düzey yapılar oluşturacağız. Yukarıda söylediğimiz gibi, başarısızlık odak noktası olacaktır: iletişim katmanlarının hatalarını nasıl ele almalıyız?

48.1 İletişimin Temelleri

Modern ağ oluşturmanın temel ilkesi, iletişimin esasında güvenilmez olduğudur. İster geniş alan internetinde, ister Infiniband gibi yerel yüksek hızlı bir ağda olsun, paketler düzenli olarak kaybolur, bozulur veya başka bir şekilde hedeflerine ulaşmaz.

Paket kaybı veya bozulmasının birçok nedeni vardır. Bazen iletim sırasında bazı bitler elektriksel veya diğer benzer problemlerden dolayı ters çevrilir. Bazen, ağ bağlantısı veya paket yönlendirici veya uzak ana bilgisayar gibi bir sistemdeki bir öge bir şekilde hasar görür veya başka bir şekilde düzgün çalışmaz; çok fazla yaşanmasada nadiren ağ kabloları yanlışlıkla kopabilir. Ancak daha temel olanı, bir ağ anahtarı, yönlendirici veya uç nokta içinde arabelleğe alma eksikliğinden kaynaklanan paket kaybıdır. Spesifik olarak, tüm bağlantıların doğru çalıştığını ve sistemdeki tüm bileşenlerin (anahtarlar, yönlendiriciler, uç ana bilgisayarlar) beklediği gibi çalıştığını ve çalıştığını garanti edebilmek bile, aşağıdaki nedenden dolayı kayıp hala mümkündür. Bir yönlendiriciye bir paket geldiğini hayal edin; paketin işlenmesi için yönlendiricinin içinde bir yere belleğe yerleştirilmesi gerekir. Bu tür birçok paket gelirse;

```

// client code
int main(int argc, char *argv[]) {
    int sd = UDP_Open(20000);
    struct sockaddr_in addrSnd, addrRcv;
    int rc = UDP_FillSockAddr(&addrSnd, "cs.wisc.edu", 10000);
    char message[BUFFER_SIZE];
    sprintf(message, "hello world");
    rc = UDP_Write(sd, &addrSnd, message, BUFFER_SIZE);
    if (rc > 0)
        int rc = UDP_Read(sd, &addrRcv, message, BUFFER_SIZE);
    return 0;
}

// server code
int main(int argc, char *argv[]) {
    int sd = UDP_Open(10000);
    assert(sd > -1);
    while (1) {
        struct sockaddr_in addr;
        char message[BUFFER_SIZE];
        int rc = UDP_Read(sd, &addr, message, BUFFER_SIZE);
        if (rc > 0) {
            char reply[BUFFER_SIZE];
            sprintf(reply, "goodbye world");
            rc = UDP_Write(sd, &addr, reply, BUFFER_SIZE);
        }
    }
    return 0;
}

```

Figure 48.1: Example UDP Code (*client.c*, *server.c*)

İlk olarak, yönlendirici içindeki bellek tüm paketleri barındıramayabilir. Yönlendiricinin bu esnada sahip olduğu tek seçenek, bir veya daha fazla paketi **bırakmaktır(drop)**. Aynı davranış, bilgisayarlarda da oluşur; tek bir makineye çok sayıda mesaj gönderdiğinizde, makinenin kaynakları kolayca ezilmiş hale gelir ve bu nedenle yeniden paket kaybı ortaya çıkar.

Bu nedenle, paket kaybı ağ oluşturmada esastır. Böylece şu soru ortaya çıkıyor: bununla nasıl başa çıkmalıyız?

48.2 Güvenilir olmayan iletişim katmanları

Bu sorunun basit bir yolu şudur: onunla ilgilenmemek. Bazı uygulamalar paket kaybıyla nasıl başa çıkacaklarını bildiklerinden, bazen güvenilir temel bir mesajlaşma katmanı ile iletişim kurmalarına izin vermek yararlıdır; aşağıdaki kod herhangi birinin sıklıkla duyduğu uçtan uca argümanın(**end-to-end argument**) bir örneğidir. (bölümün sonundaki **kenara(Aside)** bakınız). Böyle güvenilir bir katmanın mükemmel bir örneğidir.

```

int UDP_Open(int port) {
    int sd;
    if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
        return -1;
    struct sockaddr_in myaddr;
    bzero(&myaddr, sizeof(myaddr));
    myaddr.sin_family      = AF_INET;
    myaddr.sin_port        = htons(port);
    myaddr.sin_addr.s_addr = INADDR_ANY;
    if (bind(sd, (struct sockaddr *) &myaddr,
              sizeof(myaddr)) == -1) {
        close(sd);
        return -1;
    }
    return sd;
}

int UDP_FillSockAddr(struct sockaddr_in *addr,
                     char *hostname, int port) {
    bzero(addr, sizeof(struct sockaddr_in));
    addr->sin_family = AF_INET;      // host byte order
    addr->sin_port    = htons(port); // network byte order
    struct in_addr *in_addr;
    struct hostent *host_entry;
    if ((host_entry = gethostbyname(hostname)) == NULL)
        return -1;
    in_addr = (struct in_addr *) host_entry->h_addr;
    addr->sin_addr = *in_addr;
    return 0;
}

int UDP_Write(int sd, struct sockaddr_in *addr,
              char *buffer, int n) {
    int addr_len = sizeof(struct sockaddr_in);
    return sendto(sd, buffer, n, 0, (struct sockaddr *)
                  addr, addr_len);
}

int UDP_Read(int sd, struct sockaddr_in *addr,
              char *buffer, int n) {
    int len = sizeof(struct sockaddr_in);
    return recvfrom(sd, buffer, n, 0, (struct sockaddr *)
                    addr, (socklen_t *) &len);
}

```

Figure 48.2: A Simple UDP Library (udp.c)

İpucu: Bütünlük için Checksums(Sağlama toplamları) kullanın
Checksums (Sağlama toplamları), modern sistemlerde bozulmayı hızlı ve etkili bir şekilde tespit etmek için yaygın olarak kullanılan bir yöntemdir. Basit bir Checksums(sağlama toplamı), toplamadır: sadece bir veri yığınının baytlarını toplayın; tabii ki sadece bu kadar değil, temel döngüsel artıklık kodları (CRC'ler), Fletcher Checksums (sağlama toplamı) ve diğerleri [MK09] dahil olmak üzere daha bir çok gelişmiş Checksums (sağlama toplamı) oluşturuldu.

Bilgisayar ağlarında sağlama toplamları aşağıdaki gibi kullanılır. Bir makineden diğerine mesaj göndermeden önce, mesajın baytları üzerinden bir checksums(sağlama toplamı) hesaplanır. Ardından hem mesajı hem de Checksums(sağlama toplamını)'ı hedefe gönderin. Hedefte, alıcı gelen mesaj üzerinden de bir Checksums(sağlama toplamı) hesaplar; bu hesaplanan sağlama toplamı gönderilen sağlama toplamı ile eşleşirse, alıcı verinin aktarım sırasında muhtemelen bozulmadığına dair bir güvence hissedebilir.

Checksums (Sağlama toplamları) bir dizi farklı yollarla değerlendirilebilir. Birincil husus etkililik: Verilerdeki bir değişiklik Checksums(sağlama toplamında)'da bir değişikliğe yol açar mı? Checksum(Sağlama toplamı) ne kadar güçlü olursa, verilerdeki değişikliklerin fark edilmemesi o kadar zor olur. Diğer önemli kriter performans: Checksums(sağlama toplamının)'ın hesaplanması ne kadar maliyetlidir? Ne yazık ki, etkinlik ve performans çoğu zaman birbirine göre tersdir, bu da yüksek kaliteli checksums(sağlama toplamlarının) hesaplanmasının genellikle maliyetli olduğu anlamına geliyor. Hayat, yine mükemmel değil.

Bugün neredeyse tüm modern sistemlerde **UDP/IP(Kullanıcı Veri Bloğu İletişim Kuralları)** ağ yığını bulunur, UDP'yi kullanmak için bir process, bir **iletişim uç noktası(communication endpoint)** oluşturmak için socket API'sini kullanır; diğer makinelerdeki (veya aynı makinedeki) processler, orijinal processe UDP **veri bloklarını(datagramları)** gönderir (bir datagram, belirli bir maksimum boyuta ulaşana kadar sabit boyutlu bir mesajdır).

Şekil 48.1 ve 48.2, UDP/IP üzerine kurulmuş basit bir istemci ve sunucuyu göstermektedir. İstemci, sunucuya bir mesaj gönderebilir ve sunucu bir yanıt verir. Bu az miktardaki kodla, Dağıtık/dağıtımlı sistemler oluşturmaya başlamak için ihtiyacınız olan her şeyi sahipsiniz!

UDP, güvenilir bir iletişim katmanının harika bir örneğidir. Eğer UDP kullanırsanız, paketlerin kaybolduğu (düşürüldüğü) ve dolayısıyla hedeflerine ulaşmadığı durumlarla karşılaşacaksınız; gönderen hiçbir zaman kayıptan haberdar edilmez. Ancak bu, UDP'nin hiçbir arızaya karşı koruma sağlamadığı anlamına gelmez. Örneğin, UDP, bazı paket bozulma biçimlerini algılamak için bir checksums(sağlama toplamı) içerir.

Ancak, birçok uygulama yalnızca bir hedefe veri göndermek istediğinden ve paket kaybı konusunda endişelenmediğinden dolayı, daha fazlasına ihtiyacımız var. Özellikle, güvenilir bir ağın üzerinde güvenilir iletişime ihtiyacımız var.

48.3 Güvenilir İletişim Katmanları

Güvenilir bir iletişim katmanı oluşturmak için paket kaybıyla başa çıkmak için bazı yeni mekanizmalara ve tekniklere ihtiyacımız var. Gelin bir düşünelim.

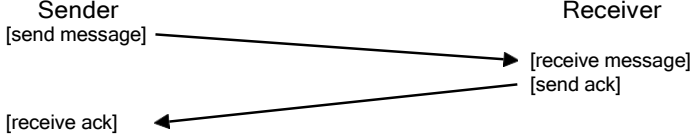


Figure 48.3: Message Plus Acknowledgment

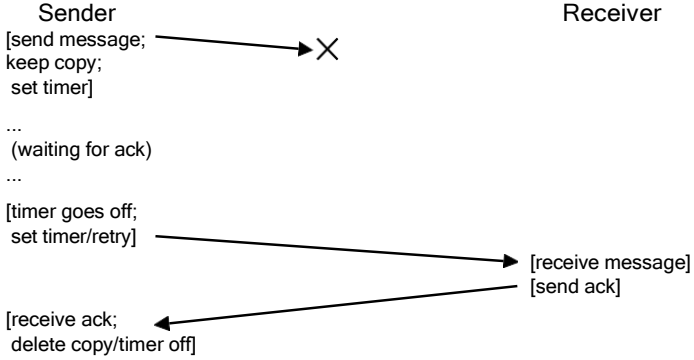


Figure 48.4: Message Plus Acknowledgment: Dropped Request

örnekte bir istemci, güvenilir bir bağlantı üzerinden bir sunucuya mesaj gönderir. Cevaplamamız gereken ilk soru: Gönderici, alıcının mesajı gerçekten aldığını nasıl biliyor?

Kullanacağımız teknik, bir alındığını bildirme(**acknowledgment**) veya kısaca kabul etmek(**ack**) olarak bilinir. Fikir basit: gönderen, alıcıya bir mesaj gönderir; alıcı daha sonra alındığını onaylamak için kısa bir mesaj gönderir. Şekil 48.3 süreci gösterir

Gönderici, mesajın bir onayını aldığında, alıcının gerçekten orijinal mesajı aldığından emin olabilir. Ancak, gönderici bir onay almazsa ne yapmalıdır?

Bu durumu ele almak için **zaman aşımı(timeout)** olarak bilinen ek bir mekanizmaya ihtiyacımız var. Gönderen bir mesaj gönderdiğinde, gönderen artık bir süre sonra kapanacak bir zamanlayıcı ayarlar. Bu süre içinde herhangi bir onay alınmadıysa, gönderen, mesajın kaybolduğu sonucuna varır. Gönderici daha sonra göndermeyi **yeniden dener(retry)** ve bu sefer dönüş yapılacağını umarak aynı mesajı tekrar gönderir. Bu yaklaşımın işe yaraması için gönderenin, tekrar göndermesi gerekmesi ihtimaline karşı mesajın bir kopyasını yanında tutması gerekir. Zaman aşımı ve yeniden deneme kombinasyonu, bazılarının yaklaşma **zaman aşımı/tekrar deneme(timeout/retry)** olarak adlandırmasına neden oldu; bu ağ türleri oldukça mantıklı, değil mi? Şekil 48.4 bir örneği göstermektedir.

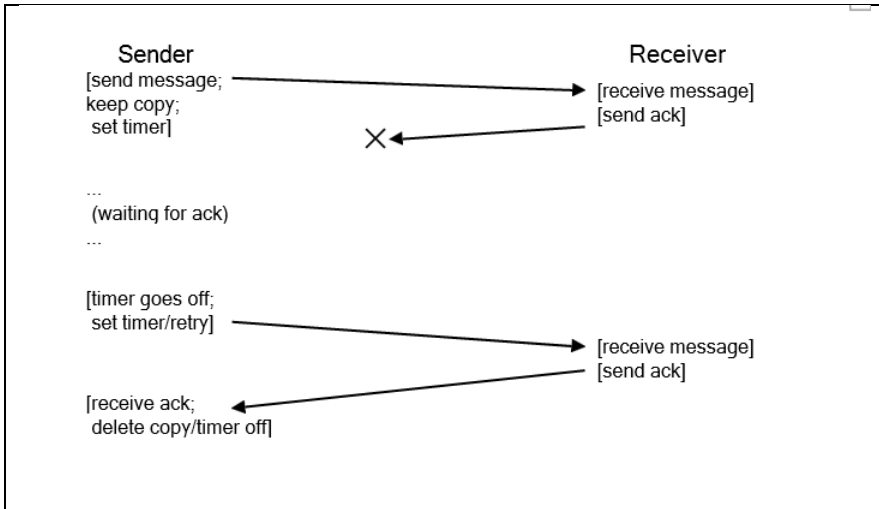


Figure 48.5: **Message Plus Acknowledgment: Dropped Reply**

Figür 48.5, soruna yol açabilecek bir paket kaybı örneğini gösterir. Bu örnekte, kaybolan mesaj orijinal değil, ancak onaylanmış. Gönderici açısından durum aynı görünüyor: hiçbir onay alınmadı ve bu nedenle bir zaman aşımı ve yeniden deneme yapılması gerekiyor. Ancak alıcının bakış açısından durum oldukça farklıdır: şimdi aynı mesaj iki kez alınmıştır! Bunun uygun olduğu durumlar olsa da, genel olarak uygun değil; bir dosya indirirken ne olacağını ve indirme işleminde fazladan paketlerin tekrarlandığını hayal edin. Bu nedenle, güvenilir bir mesaj katmanı hedeflediğimiz zaman, genellikle her mesajın alıcı tarafından **tam olarak bir kez(exactly once)** alındığını garanti etmek isteriz.

Alıcının yinelenen mesaj iletimini algılamasını sağlamak için, gönderenin her mesajı benzersiz bir şekilde tanımlaması gerekir ve alıcının her mesajı daha önce görüp görmediğini izlemek için bir yola ihtiyacı vardır. Alıcı bir kopya iletim gördüğünde, sadece mesajı kabul eder, ancak (önemli kısım) mesajı verileri alan uygulamaya iletmez. Böylece, gönderen onay alır, ancak mesaj iki kez alıcı tarafından alınmaz, yukarıda belirtilen tam olarak bir kez(exactly once) kavramı korunur.

Yinelenen mesajları tespit etmenin sayısız yolu vardır. Örneğin, gönderen her mesaj için benzersiz bir kimlik oluşturabilir; alıcı gördüğü her kimliği takip edebilir. Bu yaklaşım işe yarayabilir, ancak tüm kimlikleri izlemek için sınırsız bellek gerektirdiğinden oldukça maliyetlidir.

Daha az bellek gerektiren daha basit bir yaklaşım bu sorunu çözer. Bu yaklaşım mekanizma **sıra sayacı(quence counter)** olarak bilinir. Bir sıra sayacıyla, gönderici ve alıcı, her iki tarafın da koruyacağı bir sayaç için bir başlangıç değeri (örneğin, 1) üzerinde anlaşır. Bir mesaj gönderildiğinde, mesajla birlikte sayacın mevcut değeri de gönderilir; bu sayaç değeri (N), mesaj için bir kimlik görevi görür. Mesaj gönderildikten sonra, gönderen değeri artırır ($N + 1$ 'e).

İPUCU: ZAMAN AŞIMI DEĞERİNİ AYARLARKEN DİKKATLİ OLUN

Başlıktan da tahmin edebileceğiniz gibi, zaman aşımı değerini doğru ayarlamak, mesaj gönderimlerini yeniden denemek için zaman aşımını kullanmanın önemli bir yönüdür. Zaman aşımı çok küçükse, gönderen gereksiz yere mesajları yeniden gönderecek ve böylece gönderici ve ağ kaynakları üzerinde CPU zamanını boşa harcayacaktır. Zaman aşımı çok büyükse, gönderici yeniden göndermek için çok uzun süre bekler ve bu nedenle göndericide algılanan performans düşer. Tek bir istemci ve sunucu açısından "doğru" değer, paket kaybını algılamak için yeterince beklemek, ancak fazla beklemek değil.

Ancak, ileriki bölümlerde göreceğimiz gibi, dağıtık/dağıtımlı bir sistemde genellikle tek bir istemci ve sunucudan daha fazlası vardır. Birçok istemcinin tek bir sunucuya gönderme yaptığı bir senaryoda, sunucudaki paket kaybı, sunucunun aşırı yüklendiğinin bir göstergesi olabilir. Eğer bu doğruysa, istemciler farklı bir uyarlanabilir tarz ile yeniden deneyebilir; örneğin, ilk zaman aşımından sonra, bir müşteri zaman aşımı değerini daha yüksek bir miktara, belki de orijinal değer iki katı kadar artırabilir. Erken Aloha ağında öncülük edilen ve erken Ethernet'te [A70] benimsenen böyle bir üstel geri çekilme(exponential back-off) şeması, kaynakların aşırı yeniden gönderimler tarafından aşırı yüklendiği durumlardan kaçınır. Sağlam sistemler, bu türden aşırı yüklenmeyi önlemeye çalışır.

Alıcı, o göndericiden gelen mesajın kimliği için beklenen değer olarak sayaç değerini kullanır. Alınan mesajın kimliği (N) alıcının sayacıyla (ayrıca N) eşleşirse, mesajı alır ve uygulamaya iletir; bu durumda alıcı, bu mesajın ilk kez alındığı sonucuna varır. Alıcı daha sonra sayacını artırır ($N + 1$ 'e) ve bir sonraki mesajı bekler.

Onay kaybedilirse, gönderen zaman aşımına uğrar ve N mesajını yeniden gönderir. Bu sefer alıcının sayacı daha yüksektir ($N+1$) ve bu nedenle alıcı bu mesajı zaten aldığını bilir. Böylece mesajı alır ancak uygulamaya iletmez. Bu basit şekilde, tekrarları önlemek için dizi sayaçları kullanılabilir.

En yaygın olarak kullanılan güvenilir iletişim katmanı, **TCP/IP(Geçiş kontrol protokolü/İnternet protokolü)** veya kısaca **TCP(Geçiş kontrolü protokolü)** olarak bilinir. TCP, ağdaki sıkışıklığı işlemek için makineler kullanır[VJ88], birden çok bekleyen istek ve yüzlerce başka küçük ince ayar ve optimizasyon dahil olmak üzere, yukarıda tanımladığımızdan çok daha fazla gelişmişliğe sahiptir. Merak ediyorsanız daha fazlasını okuyun; daha da iyisi, bir ağ kurma kursuna katılın ve bu materyali iyi öğrenin.

48.4 İletişim Soyutlamaları

Temel bir mesajlaşma katmanı verildiğinde, bu bölümdeki bir sonraki soruya yaklaşıyoruz: dağıtık/dağıtımlı bir sistem kurarken hangi iletişim soyutlamasını kullanmalıyız?

Sistem topluluğu, yıllar içinde bir dizi yaklaşım geliştirdi. Bir çalışma grubu, işletim sistemi soyutlamalarını aldı ve bunları dağıtılmış bir ortamda çalışacak şekilde genişletti. Örneğin, dağıtılmış paylaşılan bellek (DSM) sistemleri, farklı makinelerdeki süreçlerin büyük bir sanal adres alanıdır [LH89]. Bu soyutlama, dağıtılmış bir hesaplamayı çok iş parçacıklı bir uygulamaya benzeyen bir şeye dönüştürür; tek fark, bu iş parçacıklarının aynı makine içinde farklı işlemciler yerine farklı makinelerde çalışmasıdır.

Çoğu DSM sisteminin çalışma şekli, işletim sisteminin sanal bellek sistemi aracılığıyla. Bir makinede bir sayfaya erişildiğinde iki şey olabilir. İlk (en iyi) durumda, sayfa makinede zaten yereldir ve bu nedenle veriler hızlı bir şekilde getirilir. İkinci durumda, sayfa şu anda başka bir makinededir. Bir sayfa hatası oluşur ve sayfa hatası işleyicisi sayfayı getirmesi, onu talep eden sürecin sayfa tablosuna kurması ve yürütmeye devam etmesi için başka bir makineye bir mesaj gönderir.

Bu yaklaşım günümüzde pek çok nedenden dolayı yaygın olarak kullanılmamaktadır. DSM için en büyük sorun, başarısızlığı nasıl ele aldığıdır. Örneğin, bir makinenin arızalandığını düşünün; o makinedeki sayfalara ne olur? Dağıtılmış hesaplamanın veri yapıları tüm adres alanına yayılırsa ne olur? Bu durumda, bu veri yapılarının parçaları aniden kullanılamaz hale gelir. Adres alanınızın bir kısmı kaybolduğunda başarısızlıkla başa çıkmak zordur; Bir "sonraki" işaretçisinin, giden adres alanının bir bölümünü gösterdiği bağlantılı bir liste hayal edin. Evet!

Bir diğer sorun ise performans. Genellikle kod yazarken belleğe erişimin ucuz olduğu varsayılır. DSM sistemlerinde bazı erişimler ucuzdur, ancak diğerleri sayfa hatalarına ve uzak makinelerden pahalı alımlara neden olur. Bu nedenle, bu tür DSM sistemlerinin programcıları, hesaplamaları neredeyse hiç iletişim olmayacak şekilde organize etmek için çok dikkatli olmak zorundaydılar ve bu tür bir yaklaşımın amacının çoğunu boşa çıkardılar. Bu alanda çok fazla araştırma yapılmış olmasına rağmen, çok az pratik yapıldı; bugün hiç kimse DSM kullanarak güvenilir dağıtılmış sistemler kurmuyor.

48.5 Uzaktan Prosedür Çağrısı (RPC)

İşletim sistemi soyutlamaları, dağıtık/dağıtımli sistemler oluşturmak için kötü bir seçim olsa da, programlama dili (PL) soyutlamaları çok daha anlamlıdır. En bilinen soyutlama, **uzaktan prosedür çağrısı(remote procedure call)** veya kısaca **RPC** fikrine dayanır [BN84]1.

Uzaktan prosedür çağrısı paketlerinin hepsinin basit bir amacı vardır: uzak bir makinede kod yürütme sürecini yerel bir işlevi çağırarak kadar basit ve anlaşılır hale getirmek. Böylece bir müşteriye bir prosedür çağrısı yapılır ve bir süre sonra sonuçlar döndürülür. Sunucu basitçe dışa aktarmak istediği bazı rutinleri tanımlar. Sihrin geri kalanı, genel olarak iki parçaya sahip olan RPC sistemi tarafından gerçekleştirilir: bir **kütük oluşturucu(stub generator)** (bazen **protokol derleyici(protocol compiler)**) olarak adlandırılır) ve çalışma zamanı kitaplığı ile. Şimdi bu parçaların her birine daha ayrıntılı olarak bakacağız.

¹In modern programming languages, we might instead say **remote method invocation (RMI)**, but who likes these languages anyhow, with all of their fancy objects?

Stub Generator(kütük oluşturucu)

Stub generator'ün işi basittir: paketleme işlevi argümanlarının ve sonuçlarının bazı sıkıntılarını otomatikleştirerek mesajlara dönüştürmek. Sayısız faydası ortaya çıkar: Tasarım gereği, bu tür kodların elle yazılmasında meydana gelen basit hataları engeller; ayrıca, bir stub generator bu kodu optimize edebilir ve böylece performansı iyileştirebilir.

Böyle bir derleyicinin girdisi, basitçe, bir sunucunun istemcilere vermek istediği çağrılar kümesidir. Kavramsal olarak, aşağıdaki kod kadar basit bir şey olabilir:

```
interface {
    int func1(int arg1);
    int func2(int arg1, int arg2);
};
```

Stub generator bunun gibi bir arabirim alır ve birkaç farklı kod parçası üretir. İstemci için, arabirimde belirtilen işlevlerin her birini içeren bir **istemci kütüğü(client stub)** oluşturulur; bu RPC hizmetini kullanmak isteyen bir istemci programı, bu client stub ile bağlantı kurar ve RPC'ler yapmak için onu çağırır.

Dahili olarak, client stub'daki bu işlevlerin her biri, uzaktan prosedür çağrısını gerçekleştirmek için gereken tüm işi yapar. İstemciye kod sadece bir işlev çağrısı olarak görünür (örneğin, istemci func1(x)'i çağırır); dahili olarak, func1() client stub'daki kod şunu yapar:

- **Bir mesaj arabelleği oluşturur.** Bir mesaj arabelleği genellikle sadece belirli bir boyuttaki bitişik bir bayt dizisidir.
- **Mesaj arabelleğine gerekli bilgileri paketler.** Bu bilgi, çağrılacak işlev için bir tür tanımlayıcının yanı sıra işlevin ihtiyaç duyduğu tüm argümanları içerir (örneğin, yukarıdaki örneğimizde, func1 için bir tamsayı). Tüm bu bilgileri tek bir bitişik ara belleğe koyma süreci, bazen argümanların **sıralanması(marshaling)** veya mesajın **serileştirilmesi(serialization)** olarak adlandırılır.
- **Mesajı hedef RPC sunucusuna gönderir.** RPC sunucusuyla iletişime geçmesi ve doğru çalışması için gereken tüm ayrıntılar, aşağıda daha detaylı olarak açıklanan RPC çalışma zamanı kitaplığı tarafından gerçekleştirilir.
- Cevabı bekler.İşlev çağrıları genellikle eşzamanlı(synchronous) olduğundan , aramanın tamamlanmasını bekleyecektir.
- **Dönüş kodunu(return code) ve diğer bağımsız değişkenleri paketinden çıkarır.** Fonksiyon sadece tek bir dönüş kodu döndürüyorsa, bu işlem basittir; ancak, daha karmaşık fonksiyonlar daha karmaşık sonuçlar (örneğin bir liste) döndürebilir ve bu nedenle stubın bunları da açması gerekebilir. Bu adım aynı zamanda **düzensiz(unmarshaling)** veya **seri durumdan çıkarma(deserialization)** olarak da bilinir.
- **Çağrıya geri dönülür.** Son olarak, client stub'dan (istemci saptamasından) geri dönülür müşteri koduna girilir.

Sunucu için ayrıca kod da oluşturulur. Sunucuda gerçekleşen adımlar aşağıdaki gibidir:

- Mesajı paketinden çıkarır. **Düzensizlik(unmarshaling)** veya **seri durumdan çıkarma(deserialization)** olarak adlandırılan bu adım, gelen mesajdaki bilgileri alır.Fonksiyon tanımlayıcısı ve bağımsız değişkenler ortaya çıkarılır.
 - Gerçek işlevi çağırılır. Sonunda! Uzak işlevin yürütüldüğü yerdeki noktaya ulaştık. RPC çalışma zamanı,ID tarafından belirtilen fonksiyonu çağırır ve istenen bağımsız değişkenleri iletir.
- Sonuçlar paketlenir. Dönüş argüman(lar)ı tek bir yanıt arabelleğine geri sıralanır .
- Cevap gönderilir. Cevap nihayet mesajı gönderen kişiye gönderilir.

Bir stub derleyicisinde dikkate alınması gereken birkaç önemli konu daha vardır. Birincisi, karmaşık argümanlardır, yani, karmaşık bir veri yapısı nasıl paketlenir ve gönderilir? Örneğin, write() sistem çağrısı çağrıldığında, üç argüman iletilir: bir tamsayı dosya tanımlayıcısı, bir arabellek için bir işaretçi ve kaç bayt (işaretçiden başlayarak) yazılacağını belirten bir boyut. Bir RPC paketine bir işaretçi iletilirse, bu işaretçiyi nasıl yorumlayacağını bulması ve doğru eylemi gerçekleştirebilmesi gerekir. Genellikle bu, ya iyi bilinen türler (örneğin, RPC derleyicisinin anladığı, belirli bir boyutta veri parçalarını iletmek için kullanılan bir arabellek) aracılığıyla veya veri yapılarına daha fazla bilgi ekleyerek, derleyiciyi etkinleştirerek gerçekleştirilir. hangi baytların serileştirilmesi gerektiğini bilmek.

Bir diğer önemli konu ise sunucunun eşzamanlılık açısından organizasyonudur. Basit bir sunucu, istekleri basit bir döngüde bekler ve her isteği birer birer işler. Ancak, tahmin edebileceğiniz gibi, bu son derece verimsiz olabilir; bir RPC çağrısı bloke olursa (örneğin, G/Ç'de), sunucu kaynakları boşa harcanır. Bu nedenle, çoğu sunucu eşzamanlı biçimde oluşturulur. Kullanılan ortak bir organizasyon vardır,bu bir **iş parçacığı havuzudur(thread pool)**. Bu organizasyonda, sunucu başladığında sonlu bir dizi iş parçacığı oluşturulur; bir mesaj geldiğinde, daha sonra RPC çağrısının işini yapan ve sonunda yanıt veren bu çalışan iş parçacıklarından birine gönderilir; bu süre boyunca, bir ana iş parçacığı diğer istekleri almaya devam eder ve bunları diğer çalışan iş parçacıklarına gönderir. Böyle bir organizasyon, sunucu içinde eşzamanlı yürütmeyi mümkün kılar, böylece kullanımını artırır; Standart maliyetler de, yoğunlukla programlama karmaşıklığında ortaya çıkar, çünkü RPC çağrılarının artık doğru çalışmasını sağlamak için kilitleri ve diğer senkronizasyon ilkelerini kullanması gerekebilir.

Çalışma Zamanı Kitaplığı

Çalışma zamanı kitaplığı, bir RPC sistemindeki ağır işlemlerinin çoğunu gerçekleştirir; çoğu performans ve güvenilirlik sorunu burada ele alınmaktadır. Şimdi böyle bir çalışma zamanı katmanı oluşturmanın bazı önemli zorluklarını tartışacağız.

Üstesinden gelmemiz gereken ilk zorluklardan biri, bir uzak hizmetin nasıl bulunacağıdır. Bu **adlandırma sorunu(naming)**, dağıtık/dağıtımli sistemlerde yaygın bir sorundur ve bir bakıma şu anki tartışmamızın kapsamını aşmaktadır. En basit yaklaşımlar, örneğin mevcut internet protokolleri tarafından sağlanan ana bilgisayar adları ve bağlantı noktası numaraları gibi mevcut adlandırma sistemleri üzerine kuruludur. Böyle bir sistemde, istemci, istenen RPC hizmetini çalıştıran makinenin ana bilgisayar adını veya IP adresini ve ayrıca kullandığı bağlantı noktası numarasını bilmelidir (bir bağlantı noktası numarası, gerçekleşen belirli bir iletişim etkinliğini tanımlamanın yalnızca bir yoludur). Bir makinede, aynı anda birden fazla iletişim kanalına izin verir). Protokol paketi daha sonra sistemdeki herhangi bir başka makineden paketleri belirli bir adrese yönlendirmek için bir mekanizma sağlamalıdır. Adlandırma hakkında iyi bir tartışma için, başka yerlere de bakmanız gerekir, örneğin, İnternette DNS ve ad çözümleme hakkında bilgi edinir veya daha iyisi Saltzer ve Kaashook'un [SK09] adlı kitabındaki mükemmel bölümü okuyun.

İstemci belirli bir uzak hizmet için hangi sunucuyla konuşması gerektiğini öğrendiğinde, sıradaki soru hangi taşıma düzeyi protokolünün RPC üzerine kurulacağıdır. Spesifik olarak, RPC sistemi TCP/IP gibi güvenilir bir protokol kullanmalı mı yoksa UDP/IP gibi güvenilmez bir iletişim katmanı üzerine mi kurulmalıdır?

Mantıklı olarak seçim kolay görünebilir: açıkçası, bir talebin uzak sunucuya güvenilir bir şekilde teslim edilmesini istiyoruz ve açıkçası güvenilir bir şekilde bir yanıt almak istiyoruz. Bu yüzden TCP gibi güvenilir taşıma protokolünü seçmeliyiz değil mi?

Ne yazık ki, güvenilir bir iletişim katmanının üzerine RPC oluşturmak, performansta büyük bir verimsizliğe yol açabilir. Yukarıdaki tartışmadan, güvenilir iletişim katmanlarının nasıl çalıştığını hatırlayın: bildirimler ve zaman aşımı/tekrar deneme ile. Böylece, istemci sunucuya bir RPC isteği gönderdiğinde, sunucu bir alındı ile yanıt verir, böylece arayan kişi isteğin alındığını bilir. Benzer şekilde, sunucu istemciye yanıtı gönderdiğinde, istemci, sunucunun yanıtın alındığını bilmesi için yanıt verir. Güvenilir bir iletişim katmanının üzerine bir istek/yanıt protokolü (RPC gibi) oluşturularak, iki "ekstra" mesaj gönderilir.

Bu nedenle, birçok RPC paketi, UDP gibi güvenilmez iletişim katmanlarının üzerine inşa edilmiştir. Bunu yapmak daha verimli bir RPC katmanı sağlar, ancak RPC sistemine güvenilirlik sağlama sorumluluğunu da ekler. RPC katmanı, yukarıda anlattığımız gibi zaman aşımı/tekrar deneme ve alındı bildirimlerini kullanarak istenen sorumluluk düzeyine ulaşır. Bir tür sıra numaralandırma kullanarak, iletişim katmanı her RPC'nin tam olarak bir kez (hata olmaması durumunda) veya en fazla bir kez (arızanın ortaya çıkması durumunda) gerçekleşmesini garanti edebilir.

Diğer Sorunlar

Bir RPC çalışma zamanının ele alması gereken başka sorunlar da vardır. Örneğin, bir uzak aramanın tamamlanması uzun sürdüğünde ne olur? Zaman aşımı makinemiz göz önüne alındığında, uzun süredir devam eden bir uzaktan arama, istemciye bir başarısızlık olarak görünebilir, bu nedenle yeniden denemeyi tetikleyebilir ve bu nedenle burada biraz bakıma ihtiyaç duyulabilir. Çözümlerden biri açık bir onay kullanmaktır.

Aside: UÇTAN UCA ARGÜMAN

Uçtan uca argüman(end-to-end argument), bir sistemdeki en yüksek seviyenin, yani genellikle "en sondaki" uygulamanın, eninde sonunda belirli işlevlerin gerçekten etkilenebileceği katmanlı bir sistem içindeki tek yerel ayardır. Dönüm noktası makaleleri [SRC84], Saltzer ve ark. Bu konuyu mükemmel bir örnekle tartışır: iki makine arasında güvenilir dosya transferi. A makinesinden B makinesine bir dosya aktarmak ve B'de biten baytların A'da başlayan baytlarla tamamen aynı olduğundan emin olmak istiyorsanız, bunu "uçtan uca" kontrol etmeniz gerekir. ; örneğin ağda veya diskte daha düşük seviyeli güvenilir makineler böyle bir garanti sağlamaz.

Karşıtlık, sistemin alt katmanlarına güvenilirlik ekleyerek güvenilir dosya aktarımı sorununu çözmeye çalışan bir yaklaşımdır. Örneğin, güvenilir bir iletişim protokolü oluşturduğumuzu ve onu güvenilir dosya aktarımımızı oluşturmak için kullandığımızı varsayalım. İletişim protokolü, gönderici tarafından gönderilen her baytın alıcı tarafından, örneğin zaman aşımı/yeniden deneme, alındı bildirimleri ve sıra numaraları kullanılarak sırayla alınacağını garanti eder. Böyle bir protokol kullanmak ne yazık ki güvenilir bir dosya transferi yapmaz; Daha iletişim gerçekleşmeden gönderici belleğindeki baytların bozulduğunu veya alıcı verileri diske yazarken kötü bir şey olduğunu hayal edin. Bu durumlarda, baytlar ağ üzerinden güvenilir bir şekilde teslim edilse bile, dosya transferimiz sonuçta güvenilir değildi. Güvenilir bir dosya aktarımı oluşturmak için, uçtan uca güvenilirlik kontrolleri dahil edilmelidir, örneğin, tüm aktarım tamamlandıktan sonra, dosyayı alıcı diskte tekrar okuyun, bir sağlama toplamı hesaplayın ve bu sağlama toplamını gönderen üzerindeki dosyayla karşılaştırın.

Bu konunun doğal sonucu olarak, bazen daha düşük katmanlara sahip olmak ekstra işlevsellik sağlar, sistem performansını iyileştirebilir veya başka bir şekilde bir sistemi optimize edebilir. Bu nedenle, bir sistemde bu tür makinelerin daha düşük bir seviyede bulunmasını kötü olarak görmeyin; bunun yerine, genel bir sistem veya uygulamada nihai kullanımı göz önüne alındığında, bu tür makinelerin faydasını dikkatlice düşünmelisiniz.

(alıcıdan gönderene) yanıt hemen oluşturulmadığında; bu, istemcinin sunucunun isteği aldığını bilmesini sağlar. Ardından, belirli bir süre geçtikten sonra, istemci düzenli aralıklarla sunucunun istek üzerinde çalışıp çalışmadığını sorabilir; sunucu "evet" demeye devam ederse, istemci mutlu olmalı ve beklemeye devam etmelidir (sonuçta, bazen bir prosedür çağrısının yürütmeyi bitirmesi uzun zaman alabilir).

Çalışma zamanı, aynı zamanda, tek bir pakete sığabilecek olandan daha büyük, büyük argümanlarla prosedür çağrılarını da ele almalıdır. Bazı daha düşük seviyeli ağ protokolleri, bu tür gönderici tarafı **parçalanmasını(fragmentation)** (daha büyük paketlerin bir dizi daha küçük pakete) ve alıcı tarafı **yeniden birleştirmesini(reassembly)** (daha küçük parçaların daha büyük bir mantıksal bütün halinde) sağlar; eğer böyle değilse, RPC çalışma zamanının bu tür işlevselliği kendisinin uygulaması gerekebilir. Ayrıntılar için Birrell ve Nelson'ın makalesine bakın [BN84].

Birçok sistemin ele aldığı bir konu, **bayt sıralamasıdır(byte ordering)**. Bildiğiniz gibi, bazı makineler değerleri **büyük endian(big endian)** sıralaması olarak bilinen şekilde depolarken, diğerleri **küçük endian(little endian)** sıralamasını kullanır. Big endian, baytları (örneğin bir tamsayı) en önemli bittten en az anlamlı bite kadar saklar, Arap rakamlarına çok benzer; küçük endian tam tersini yapar. Her ikisi de sayısal bilgileri saklamanın eşit derecede geçerli yollarıdır; Buradaki soru, farklı endianness makineleri arasında nasıl iletişim kurulacağıdır.

RPC paketleri bunu genellikle mesaj formatları içinde iyi tanımlanmış bir bütünlük sağlayarak halleder. Sun'ın RPC paketinde, **XDR (eXternal Data Representation- Harici Veri Temsili)** katmanı bu işlevi sağlar. Bir mesaj gönderen veya alan makine XDR'nin endianlığı ile eşleşirse, mesajlar beklendiği gibi gönderilir ve alınır. Bununla birlikte, iletişim kuran makinenin farklı bir özelliği varsa, mesajdaki her bilgi parçası dönüştürülmelidir. Bu nedenle, endianlık farkı küçük bir performans maliyetine sahip olabilir.

Son bir konu olarak, iletişimin asenkron doğasının istemcilere gösterilip gösterilmeyeceği ve böylece bazı performans optimizasyonlarının sağlanıp sağlanmayacağı konusu vardır. Spesifik olarak, tipik RPC'ler **eşzamanlı(synchronously)** olarak yapılır, yani bir müşteri prosedür çağrısını yayınladığında, devam etmeden önce prosedür çağrısının geri dönmesini beklemesi gerekir. Bu bekleme uzun olabileceğinden ve istemcinin yapabileceği başka işler olabileceğinden, bazı RPC paketleri bir RPC'yi **eşzamansız(asynchronously)** olarak çağırmanıza olanak tanır. Eşzamansız bir RPC yayımlandığında, RPC paketi isteği gönderir ve hemen geri döner; istemci daha sonra diğer RPC'leri çağırarak veya diğer hesaplamalar gibi diğer işleri yapmakta serbesttir. İstemci bir noktada eşzamansız RPC'nin sonuçlarını görmek isteyecektir; böylece RPC katmanını geri çağırır ve ona bekleyen RPC'lerin tamamlanmasını beklemesini söyler, bu noktada dönüş argümanlarına erişilebilir.

48.6 Özet

Yeni bir konuyu, dağıtılmış sistemlerin ve bunun ana sorununun giriş kısmını gördük: artık sıradan bir olay olan arızanın nasıl ele alınacağını öğrendik. Google'ın içinde dedikleri gibi, yalnızca masaüstü makineniz olduğunda, başarısızlık nadirdir; binlerce makinenin bulunduğu bir veri merkezindeyken, her zaman arıza meydana gelir. Herhangi bir dağıtılmış sistemin anahtarı, bu başarısızlıkla nasıl başa çıkacağınızdır.

Ayrıca, iletişimin herhangi bir dağıtılmış sistemin kalbini oluşturduğunu da gördük. Bu iletişimin ortak bir özeti, istemcilerin sunucular üzerinde uzaktan arama yapmalarını sağlayan uzaktan prosedür çağrısında (RPC) bulunur; RPC paketi, yerel bir prosedür çağrısını yakından yansıtan bir hizmet sunmak için zaman aşımı/yeniden deneme ve onay dahil tüm korkutucu ayrıntıları ele alır.

Bir RPC paketini gerçekten anlamamanın en iyi yolu elbette bir tanesini kendiniz kullanmaktır. Sun'ın stub derleyici rpcgen kullanan RPC sistemi daha eski bir sistemdir; Google'ın gRPC ve Apache Thrift'i daha modern yaklaşımlardır. Bunlardan birini deneyin ve tüm bu konuştuklarımızın ne hakkında olduğunu görün.

References

[A70] "The ALOHA System — Another Alternative for Computer Communications" by Norman Abramson. The 1970 Fall Joint Computer Conference. *The ALOHA network pioneered some basic concepts in networking, including exponential back-off and retransmit, which formed the basis for communication in shared-bus Ethernet networks for years.*

[BN84] "Implementing Remote Procedure Calls" by Andrew D. Birrell, Bruce Jay Nelson. ACM TOCS, Volume 2:1, February 1984. *The foundational RPC system upon which all others build. Yes, another pioneering effort from our friends at Xerox PARC.*

[MK09] "The Effectiveness of Checksums for Embedded Control Networks" by Theresa C. Maxino and Philip J. Koopman. IEEE Transactions on Dependable and Secure Computing, 6:1, January '09. *A nice overview of basic checksum machinery and some performance and robustness comparisons between them.*

[LH89] "Memory Coherence in Shared Virtual Memory Systems" by Kai Li and Paul Hudak. ACM TOCS, 7:4, November 1989. *The introduction of software-based shared memory via virtual memory. An intriguing idea for sure, but not a lasting or good one in the end.*

[SK09] "Principles of Computer System Design" by Jerome H. Saltzer and M. Frans Kaashoek. Morgan-Kaufmann, 2009. *An excellent book on systems, and a must for every bookshelf. One of the few terrific discussions on naming we've seen.*

[SRC84] "End-To-End Arguments in System Design" by Jerome H. Saltzer, David P. Reed, David D. Clark. ACM TOCS, 2:4, November 1984. *A beautiful discussion of layering, abstraction, and where functionality must ultimately reside in computer systems.*

[VJ88] "Congestion Avoidance and Control" by Van Jacobson. SIGCOMM '88. *A pioneering paper on how clients should adjust to perceived network congestion; definitely one of the key pieces of technology underlying the Internet, and a must read for anyone serious about systems, and for Van Jacobson's relatives because well relatives should read all of your papers.*

Homework (Code)

In this section, we'll write some simple communication code to get you familiar with the task of doing so. Have fun!

Questions

1. Bu bölümde sağlanan kodu kullanarak basit bir UDP tabanlı sunucu ve istemci oluşturun. Sunucu, istemciden mesajlar almalı ve bir onay ile yanıt vermelidir. Bu ilk denemede, herhangi bir yeniden iletim veya sağlamlık eklemeyin (iletişimin mükemmel çalıştığını varsayın). Bunu test için tek bir makinede çalıştırın; daha sonra iki farklı makinede çalıştırın.

Cevap:

Kodu oluşturmak için gerekli kütüphaneler import edildikten sonra server üzerinden belir bir ip ve port değerine göre bağlantı oluşturuldu.

Sonrasında aynı ip ve port numarasına bir client bağlanarak mesaj gönderdi ve server tarafı da gelen mesajı görüp geri cevap verdi.

Bağlantı için socket metodu kullanıldı.

Bir taraftan diğer tarafa mesaj göndermek için send metodu kullanıldı.

Bir taraftan diğer tarafın mesajını okumak için recv metodu kullanıldı.

Bağlantıda hata olup olmadığı da kontrol edilerek gerekli işlemler yapıldı.

Client Kodu:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

int main(){

    char *ip = "127.0.0.1";
    int port = 5566;

    int sock;
    struct sockaddr_in addr;
    socklen_t addr_size;
    char buffer[1024];
    int n;

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0){
        perror("[-]Socket hatası");
        exit(1);
    }
    printf("[+]TCP server socket oluşturuldu.\n");

    memset(&addr, '\0', sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port = port;
    addr.sin_addr.s_addr = inet_addr(ip);

    connect(sock, (struct sockaddr*)&addr, sizeof(addr));
    printf("Server bağlandı.\n");

    bzero(buffer, 1024);
    strcpy(buffer, "MERHABA BEN CLIENT'IM");
    printf("Client: %s\n", buffer);
    send(sock, buffer, strlen(buffer), 0);

    bzero(buffer, 1024);
    recv(sock, buffer, sizeof(buffer), 0);
    printf("Server: %s\n", buffer);

    close(sock);
    printf("Serverden ayrıldı.\n");

    return 0;
}

```

Server kodu:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

int main(){

    char *ip = "127.0.0.1";
    int port = 5566;

    int server_sock, client_sock;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_size;
    char buffer[1024];
    int n;

    server_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (server_sock < 0){
        perror("[~]Socket hatası");
        exit(1);
    }
    printf("[+]TCP server socket oluşturuldu.\n");

    memset(&server_addr, '\0', sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = port;
    server_addr.sin_addr.s_addr = inet_addr(ip);

    n = bind(server_sock, (struct sockaddr*)&server_addr, sizeof(server_addr));
    if (n < 0){
        perror("[~]Bind error");
        exit(1);
    }
    printf("[+]Port Numarasını bağlayın: %d\n", port);

    listen(server_sock, 5);
    printf("Clientler Dinleniyor...\n");

    while(1){
        addr_size = sizeof(client_addr);
        client_sock = accept(server_sock, (struct sockaddr*)&client_addr, &addr_size);
        printf("[+]Client bağlandı.\n");

        bzero(buffer, 1024);
        recv(client_sock, buffer, sizeof(buffer), 0);
        printf("Client: %s\n", buffer);

        bzero(buffer, 1024);
        strcpy(buffer, "MERHABA BU SERVER MESAJIDIR.IYI GUNLER.");
        printf("Server: %s\n", buffer);
        send(client_sock, buffer, strlen(buffer), 0);

        close(client_sock);
        printf("[+]Client ayrıldı. \n\n");
    }

    return 0;
}

```

Server ve client programları çalıştırıldıktan sonra ekran görüntüler:

Server bağlantısı:

```
alipek@ubuntu2204:~/Masaüstü/ödev1$ touch client.c
alipek@ubuntu2204:~/Masaüstü/ödev1$ ls
client.c
alipek@ubuntu2204:~/Masaüstü/ödev1$ touch server.c
alipek@ubuntu2204:~/Masaüstü/ödev1$ ls
client.c server.c
alipek@ubuntu2204:~/Masaüstü/ödev1$ ls
client.c server.c
alipek@ubuntu2204:~/Masaüstü/ödev1$ gcc -o server server.c
alipek@ubuntu2204:~/Masaüstü/ödev1$ ./ server
bash: ./: Bir izin
alipek@ubuntu2204:~/Masaüstü/ödev1$ ./server
[+]TCP server socket oluşturuldu.
[+]Port Numarasını bağlayın: 5566
Clientler Dinliyor...
```

Client bağlantısı:

```
alipek@ubuntu2204:~/Masaüstü/ödev1$ gcc client client.c
/usr/bin/ld: client bulunamadı: Böyle bir dosya ya da izin yok
collect2: error: ld returned 1 exit status
alipek@ubuntu2204:~/Masaüstü/ödev1$ gcc -o client client.c
alipek@ubuntu2204:~/Masaüstü/ödev1$ ls
client client.c server server.c
alipek@ubuntu2204:~/Masaüstü/ödev1$ ./client
[+]TCP server socket oluşturuldu.
Server bağlandı.
Client: MERHABA BEN CLIENT'IM
Server: MERHABA BU SERVER MESAJIDIR.IYI GUNLER.
Serverdan ayrıldı.
alipek@ubuntu2204:~/Masaüstü/ödev1$
```

Client bağlantısı bittikten sonra server gözünden olanlar:

```
alipek@ubuntu2204:~/Masaüstü/ödev1$ touch client.c
alipek@ubuntu2204:~/Masaüstü/ödev1$ ls
client.c
alipek@ubuntu2204:~/Masaüstü/ödev1$ touch server.c
alipek@ubuntu2204:~/Masaüstü/ödev1$ ls
client.c server.c
alipek@ubuntu2204:~/Masaüstü/ödev1$ ls
client.c server.c
alipek@ubuntu2204:~/Masaüstü/ödev1$ gcc -o server server.c
alipek@ubuntu2204:~/Masaüstü/ödev1$ ./ server
bash: ./: Bir izin
alipek@ubuntu2204:~/Masaüstü/ödev1$ ./server
[+]TCP server socket oluşturuldu.
[+]Port Numarasını bağlayın: 5566
Clientler Dinliyor...
[+]Client bağlandı.
Client: MERHABA BEN CLIENT'IM
Server: MERHABA BU SERVER MESAJIDIR.IYI GUNLER.
[+]Client ayrıldı.
```

2. Kodunuzu bir iletişim kitaplığına dönüştürün. Özellikle, kendi API'nizi yapın, gönderme ve alma çağrılarının yanı sıra gerektiğinde diğer API çağrıları yapın. Ham soket çağrıları yerine kitaplığınızı kullanmak için istemcinizi ve sunucunuzu yeniden yazın.

Server Kodu:

```
#include <stdio.h>
#include "common.h"

int main(int argc, char** argv) {
    int sd = UDP_Open(8000);
    struct sockaddr_in addrRcv;
    char msg[BUFFER_SIZE];
    while(1) {
        UDP_Read(sd, &addrRcv, msg, BUFFER_SIZE);
        printf("Received: %s\n", msg);
        sprintf(msg, "Merhaba Client");
        UDP_Write(sd, &addrRcv, msg, BUFFER_SIZE);
    }
    return 0;
}
```

Client kodu:

```
#include <stdio.h>
#include "common.h"

int main(int argc, char** argv) {
    int sd = UDP_Open(8080);
    struct sockaddr_in addrSend, addrRcv;
    UDP_FillSockAddr(&addrSend, "localhost", 8000);
    char msg[BUFFER_SIZE];
    sprintf(msg, "Merhaba Server");
    UDP_Write(sd, &addrSend, msg, BUFFER_SIZE);
    UDP_Read(sd, &addrRcv, msg, BUFFER_SIZE);
    printf("Received: %s\n", msg);
    return 0;
}
```

Kütüphane Kodu:

```

#include <netdb.h>
#include <netinet/in.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

#define BUFFER_SIZE 1024

int UDP_Open(int port) {
    int sd;
    if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) return -1;
    struct sockaddr_in myaddr;
    bzero(&myaddr, sizeof(myaddr));
    myaddr.sin_family = AF_INET;
    myaddr.sin_port = htons(port);
    myaddr.sin_addr.s_addr = INADDR_ANY;
    if (bind(sd, (struct sockaddr *)&myaddr, sizeof(myaddr)) == -1) {
        close(sd);
        return -1;
    }
    return sd;
}

int UDP_FillSockAddr(struct sockaddr_in *addr, char *hostname, int port) {
    bzero(addr, sizeof(struct sockaddr_in));
    addr->sin_family = AF_INET; // host byte order
    addr->sin_port = htons(port); // network byte order
    struct in_addr *in_addr;
    struct hostent *host_entry;
    if ((host_entry = gethostbyname(hostname)) == NULL) return -1;
    in_addr = (struct in_addr *)host_entry->h_addr;
    addr->sin_addr = *in_addr;
    return 0;
}

int UDP_Write(int sd, struct sockaddr_in *addr, char *buffer, int n) {
    int addr_len = sizeof(struct sockaddr_in);
    return sendto(sd, buffer, n, 0, (struct sockaddr *)addr, addr_len);
}

int UDP_Read(int sd, struct sockaddr_in *addr, char *buffer, int n) {
    int len = sizeof(struct sockaddr_in);
    return recvfrom(sd, buffer, n, 0, (struct sockaddr *)addr,
                    (socklen_t *)&len);
}

```

Server bir portnumber üzerinden server açmak için `UDP_Open` metodunu kullanıyor. Herhangi bir client bu servera bağlanmak için ise `UDP_FillSockAddr` metodunu kullanıyor ve açılan portnumber ile serverla bağlantıya geçiyor.

Burada client ve server bu kütüphaneye ait olan metodları kullanarak işlevlerini gerçekleştiriyorlar.

Client ya da server karşı taraftan gelen mesajı okumak için `UDP_Read` metodunu kullanırken bir mesaj göndermek için `UDP_Write` kullanılıyor.

Server ve client programları çalıştırıldıktan sonra ekran görüntüleri:

```
alipekin@ubuntu2204:~/Masaüstü/Yeni$ ls
client.c common.h common.h.gch server server.c
alipekin@ubuntu2204:~/Masaüstü/Yeni$ gcc -o client client.c
alipekin@ubuntu2204:~/Masaüstü/Yeni$ ls
client client.c common.h common.h.gch server server.c
alipekin@ubuntu2204:~/Masaüstü/Yeni$ ./client
Received: Merhaba Client
alipekin@ubuntu2204:~/Masaüstü/Yeni$
```

```
alipekin@ubuntu2204: ~/Masaüstü/Yeni × alipekin@ubuntu2204: ~/Masaüstü/Yeni ×
alipekin@ubuntu2204:~$ cd Masaüstü
alipekin@ubuntu2204:~/Masaüstü$ cd Yeni
alipekin@ubuntu2204:~/Masaüstü/Yeni$ ls
client.c common.h server.c
alipekin@ubuntu2204:~/Masaüstü/Yeni$ gcc common.h
alipekin@ubuntu2204:~/Masaüstü/Yeni$ ls
client.c common.h common.h.gch server.c
alipekin@ubuntu2204:~/Masaüstü/Yeni$ gcc -o server server.c
alipekin@ubuntu2204:~/Masaüstü/Yeni$ ./server
Received: Merhaba Server
```

Hem client hem server için bir kütüphane oluşturuldu ve bu kütüphaneden yararlanılarak bağlantı sağlanmış oldu.

3. Gelişmekte olan iletişim kitaplığınıza zaman aşımı/tekrar deneme şeklinde güvenilir iletişim ekleyin. Özellikle, kitaplığınız göndereceği herhangi bir mesajın bir kopyasını oluşturmalıdır. Gönderirken, bir zamanlayıcı başlatmalıdır, böylece mesajın gönderilmesinden bu yana ne kadar zaman geçtiğini izleyebilir. Alıcıda, kitaplık alınan mesajları onaylamalıdır. İstemci gönderimi gönderirken engellemeli, yani geri dönmenden önce ileti onaylanana kadar beklemelidir. Ayrıca süresiz olarak yeniden göndermeyi denemeye istekli olmalıdır. Maksimum mesaj boyutu, UDP ile gönderebileceğiniz en büyük tek mesajın boyutu olmalıdır. Son olarak, gönderecinin bir onay gelene veya iletim zaman aşımına uğrayana kadar uyku moduna alarak zaman aşımı/tekrar denemeyi verimli bir şekilde gerçekleştirdiğinizden emin olun; CPU'yu döndürmeyin ve boş harcamayın

Time.h kütüphanesi kullanılarak ne zaman mesajın gönderildiği bilgisine ulaşıldı.

Server Kodu:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <time.h>

#define BACKLOG 10

int main(int argc, char **argv){
    if(argc != 2){
        printf("Port Numarası giriniz.%s<port>\n", argv[0]);
        exit(0);
    }

    int port = atoi(argv[1]);
    printf("Port: %d\n", port);

    int n_client = 0;
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = INADDR_ANY;
    serverAddress.sin_port = htons(port);

    bind(sockfd, (struct sockaddr*)&serverAddress, sizeof(serverAddress));
    printf("[+]Bind\n");

    listen(sockfd, BACKLOG);
    printf("[+]Client'ler dinleniyor...\n");

    int i = 1;
    while(i){
        int client_socket = accept(sockfd, NULL, NULL);
        n_client++;
        time_t currentTime;
        time(&currentTime);
        printf("Client %d katıldığı zaman %s", n_client, ctime(&currentTime));
        send(client_socket, ctime(&currentTime), 30, 0);
    }

    return 0;
}
```


Client Kodu:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <time.h>

int main(int argc, char **argv){
    if(argc != 2){
        printf("Port Numarası giriniz.%s<port>\n", argv[0]);
        exit(0);
    }

    int port = atoi(argv[1]);
    printf("Port: %d\n", port);

    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    char response[30];
    struct sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = INADDR_ANY;
    serverAddress.sin_port = htons(port);

    connect(sockfd, (struct sockaddr*)&serverAddress, sizeof(serverAddress));
    printf("[+]Server'a bağlanıldı.\n");

    recv(sockfd, response, 29, 0);
    printf("Bağlanma Zamanı: %s", response);

    return 0;
}
```

Ekran görüntüleri:**Client tarafı:**

```
alipekin@ubuntu2204:~/Masaüstü/ödev3$ ./client 4444
Port: 4444
[+]Server'a bağlanıldı.
Bağlanma Zamanı: Thu Dec 8 20:22:16 2022
alipekin@ubuntu2204:~/Masaüstü/ödev3$
```

Server Tarafı

```
alipekin@ubuntu2204:~/Masaüstü/ödev3$ ^C
alipekin@ubuntu2204:~/Masaüstü/ödev3$ gcc -o server server.c
alipekin@ubuntu2204:~/Masaüstü/ödev3$ gcc -o client client.c
alipekin@ubuntu2204:~/Masaüstü/ödev3$ ./server 4444
Port: 4444
[+]Bind
[+]Client'ler dinleniyor...
Client 1 katıldığı zaman Thu Dec 8 20:22:16 2022
```

4. Kitaplığınızı daha verimli ve özelliklerle dolu hale getirin. İlk olarak, çok büyük mesaj aktarımı ekleyin. Spesifik olarak, ağ maksimum mesaj boyutunu sınırlasa da, kitaplığınız isteğe bağlı olarak büyük boyutlu bir mesaj almalı ve onu istemciden sunucuya aktarmalıdır. İstemci bu büyük mesajları parçalar halinde sunucuya iletmelidir; sunucu tarafı kitaplık kodu, alınan parçaları bitişik bütün halinde birleştirmeli ve tek büyük arabelleği bekleyen sunucu koduna geçirmelidir.

Client tarafından gönderilen mesajın boyutunu kendi istediğimiz maksimum boyuta bölüp her bir bölünen parçayı mesaj olarak göndermemiz gerekiyor.

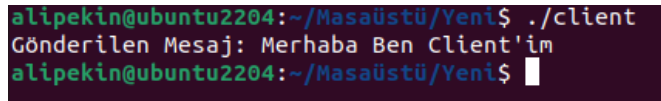
Client kodunda yapılan değişiklikler:

```
printf( "Gönderilen Mesaj: %s\n", msg );
char * token = strtok(msg, " ");
while( token != NULL ) {
    UDP_Write(sd, &addrSend, token, BUFFER_SIZE);
    token = strtok(NULL, " ");
}
```

Gönderilen mesajın bölünerek gönderilmesi sağlandı.

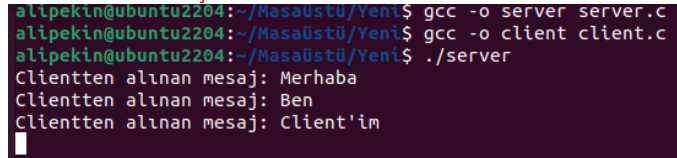
Ekran Görüntüleri:

Client çalıştırıldıktan sonra:



```
alipekin@ubuntu2204:~/Masaüstü/Yeni$ ./client
Gönderilen Mesaj: Merhaba Ben Client'im
alipekin@ubuntu2204:~/Masaüstü/Yeni$
```

Serverda Gözükme Şekli:



```
alipekin@ubuntu2204:~/Masaüstü/Yeni$ gcc -o server server.c
alipekin@ubuntu2204:~/Masaüstü/Yeni$ gcc -o client client.c
alipekin@ubuntu2204:~/Masaüstü/Yeni$ ./server
Clientten alınan mesaj: Merhaba
Clientten alınan mesaj: Ben
Clientten alınan mesaj: Client'im
```

5. Yukarıdakileri tekrar yapın, ancak yüksek performansla. Her parçayı birer birer göndermek yerine, çok sayıda parçayı hızlı bir şekilde göndermeli, böylece ağır çok daha fazla kullanılmasını sağlamalısınız. Bunu yapmak için, alıcı tarafındaki yeniden birleştirmenin mesajı karıştırmaması için aktarımın her bir parçasını dikkatlice işaretleyin.

Alıcı tarafında gelen mesajların tek tek değilde bir bütün şeklinde hızlı bir şekilde görülmesini sağlamak için gelen mesajların her birini bir bir değiştikende birleştirdikten sonra bu birleşen mesajı alıcıya gösterirsek bir bütün şeklinde görmüş olur.

Strcat fonksiyonu kullanarak gelen mesajları birleştirdim.

```
char dest[DEST_SIZE] = "";
while(1) {
    UDP_Read(sd, &addrRcv, msg, BUFFER_SIZE);
    strcat(dest, msg);
}
```

Ekran görüntüsü:

Client tarafı:

```
alipekin@ubuntu2204:~/Masaüstü/Yeni$ ./client
Gönderilen Mesaj: Merhaba Ben Client'im
alipekin@ubuntu2204:~/Masaüstü/Yeni$
```

Server tarafı:

```
alipekin@ubuntu2204:~/Masaüstü/Yeni$ ./server
Clientten alınan mesaj: Merhaba Ben Client'im
```

6. Son bir uygulama : sipariş teslimi ile asenkron mesaj gönderme. Yani, istemci art arda mesaj göndermek için art arda gönderici'yi arayabilmelidir; alıcı, her mesajı sırayla, güvenilir bir şekilde almalıdır; gelen birçok mesaj gönderici aynı anda uçta olabilmelidir. Ayrıca, bir istemcinin bekleyen tüm iletilerin onaylanmasını beklemesini sağlayan bir gönderen tarafı araması ekleyin.

Eşzamansız mesaj göndermeyi uygulamanın bir yolu, bir ağ bağlantısı oluşturmak için "sockets" API'sini kullanmak ve ardından bağlantı üzerinden veri göndermek ve almak için "gönder" ve "al" işlevlerini kullanmaktır. . Gönderen, mesajları eşzamansız olarak göndermek için ayrı bir iş parçacığı kullanabilir ve alıcı, mesajları sırayla almak ve işlemek için başka bir iş parçacığını kullanabilir.

İletilerin doğru sırayla teslim edilmesini sağlamak için, gönderen her iletiye bir sıra numarası ekleyebilir ve alıcı bu sıra numarasını mesajları doğru sırada yeniden birleştirmek için kullanabilir. Gönderici, alıcı tarafından onaylanmayan mesajları yeniden iletmek için bir mekanizma da uygulayabilir ve alıcı, her mesajı aldığını doğrulamak için onay mesajları gönderebilir.

Bir istemcinin onaylanacak tüm bekleyen mesajları beklemesini sağlamak için, gönderici bekleyen mesajların bir listesini tutabilir ve tüm mesajlar onaylanana kadar istemciyi bloke etmek için bir "bekle" işlevini kullanabilir. Alıcı, kontrolü istemciye geri vermeden önce bekleyen tüm mesajların işlenmesini sağlamak için bir "flush" işlevi de uygulayabilir.

7. Şimdi, bir acı nokta daha: ölçüm. Yaklaşımlarınızın her birinin bant genişliğini ölçün; iki farklı makine arasında hangi hızda ne kadar veri aktarabilirsiniz? Ayrıca gecikmeyi de ölçün: tek paket gönderimi ve alıntısı için ne kadar çabuk biter? Son olarak, rakamlarınız makul görünüyor mu? Ne bekliyordun? Bir sorun olup olmadığını veya kodunuzun iyi çalıştığını bilmek için beklentilerinizi nasıl daha iyi belirleyebilirsiniz?

Eğer aynı ağda birden fazla cihaz bulunuyorsa gecikme o kadar az olurken gönderilme hızı da o kadar hızlı olur. Bunu etkileyen bir diğer faktör makinelerin birbirine olan uzaklığı.