48

Distributed Systems

Dağıtılmış sistemler **(Distributed systems)** dünyanın yüzünü değiştirdi. Web tarayıcınız gezegende başka bir yerde bir web sunucusuna bağlandığında, **client/server** dağıtılmış sisteminin **(Distributed systems)** basit bir biçimi gibi görünen şeye katılıyor. Google veya Facebook gibi modern bir web servisiyle iletişime geçtiğinizde, yalnızca tek bir makineyle etkileşime girmiyorsunuz; Perde arkasında, bu karmaşık hizmetler, her biri sitenin belirli hizmetini sağlamak için işbirliği yapan geniş bir makine koleksiyonundan inşa edilmiştir. Bu nedenle, dağıtılmış sistemleri incelemeyi ilginç kılan şeyin ne olduğu açık olmalıdır. Gerçekten de, bütün bir sınıfa layıktır; Burada, sadece ana konulardan birkaçını tanıtıyoruz.

Dağıtılmış bir sistem oluştururken bir dizi yeni zorluk ortaya çıkmaktadır. Odaklandığımız en önemli şey sorundur(**failure**); makineler, diskler, ağlar ve yazılımlar zaman zaman sorunlu olur, çünkü "mükemmel" bileşenlerin ve sistemlerin nasıl inşa edileceğini bilmiyoruz (ve muhtemelen asla bilemeyeceğiz). Ancak, ne zaman
modern bir web hizmeti oluşturuyoruz, müşterilere asla sorun olmuyormuş gibi görünmesini istiyoruz; bu görevi nasıl başarabiliriz?

Önemli Nokta:
BİLEŞENLER ARIZALI OLDUĞUNDA ÇALIŞAN SİSTEMLER NASIL OLUŞTURULUR
Her zaman doğru çalışmayan parçalardan çalışan bir sistemi nasıl kurabiliriz? Temel soru size RAID depolama dizilerinde tartıştığımız bazı konuları hatırlatmalıdır; ancak buradaki sorunlar, çözümler gibi daha karmaşık olma eğilimindedir.

İlginç bir şekilde, başarısızlık, dağıtılmış sistemlerin inşasında merkezi 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 toplayarak, bileşenleri düzenli olarak arızalanmasına rağmen nadiren arıza yapıyor gibi görünen bir sistem inşa edebiliriz. Bu gerçeklik, dağıtılmış sistemlerin merkezi güzelliği ve değeridir ve neden Google, Facebook vb. dahil olmak üzere kullandığınız neredeyse her modern web hizmetinin temelini oluşturmaktadır.

İPUCU: İLETİŞİM DOĞAL OLARAK GÜVENİLİR DEĞİLDİR

Hemen hemen her koşulda, iletişimi temelde güvenilmez bir faaliyet olarak görmek iyidir. Bit bozulması, çalışmayan bağlantılar ve makineler ve gelen paketler için arabellek alanının olmaması aynı sonuca yol açar: paketler bazen hedeflerine ulaşmaz. Bu tür güvenilir olmayan ağların üzerine güvenilir hizmetler inşa etmek için, paket kaybıyla başa çıkabilecek teknikleri düşünmeliyiz.

Diğer önemli sorunlar da var. Sistem performansı (**performance**) genellikle kritiktir; Dağıtılmış sistemimizi birbirine bağlayan bir ağ ile, sistem tasarımcıları genellikle verilen görevleri nasıl gerçekleştireceklerini dikkatlice düşünmeli, gönderilen mesajların sayısını azaltmaya ve iletişimi mümkün olduğunca verimli (düşük gecikme süresi, yüksek bant genişliği) yapmaya çalışmalıdır.

Son olarak, güvenlik (**security**) de gerekli bir hususdur. bağlanırken uzak bir site için, uzaktaki tarafın söyledikleri kişi olduğuna dair bir miktar güvenceye sahip olmak merkezi bir sorun haline gelir. Ayrıca, üçüncü tarafların diğer iki kişi arasında devam eden bir iletişimi izleyememesini veya değiştirememesini sağlamak da bir zorluktur.

Bu girişte, dağıtılmış bir sistemde yeni olan en temel özelliği ele alacağız: iletişim (**communication**). Yani, dağıtılmış bir sistemdeki makineler birbirleriyle nasıl iletişim kurmalıdır? Mevcut en temel ilkellerle, mesajlarla başlayacağız ve birkaç tane daha üst yapı oluşturacağız.
üstlerinde seviye ilkelleri. Yukarıda söylediğimiz gibi, başarısızlık merkezi bir odak olacaktır: iletişim katmanları başarısızlıkları nasıl ele almalı?

48.1 İletişim Temelleri (Communication Basics)

Modern ağ oluşturmanın temel ilkesi, iletişimin temelde güvenilmez olmasıdır. Geniş alan İnternet'te veya Infiniband gibi yerel alan yüksek hızlı ağda olsun, paketler düzenli olarak kaybolur, bozulur veya başka bir şekilde hedeflerine ulaşmaz.

Paket kaybının veya bozulmasının çok sayıda nedeni vardır. Bazen, iletim sırasında, elektriksel veya benzeri başka sorunlar nedeniyle bazı bitler ters çevrilir. Bazen, sistemdeki bir ağ bağlantısı veya paket yönlendirici veya hatta uzak ana bilgisayar gibi bir öğe bir şekilde zarar görür veya başka bir şekilde düzgün çalışmaz; ağ kabloları, en azından bazen, yanlışlıkla kopuyor.

Bununla birlikte daha temel olan, bir ağ anahtarı, yönlendirici veya uç nokta içinde ara belleğe alma eksikliğinden kaynaklanan paket kaybıdır. Spesifik olarak, tüm bağlantıların doğru çalıştığını ve sistemdeki tüm bileşenlerin (anahtarlar, yönlendiriciler, uç ana bilgisayarlar) beklendiği gibi çalışır durumda olduğunu garanti edebilsek bile, aşağıdaki nedenden dolayı kayıp yine de mümkündür. Bir yönlendiriciye bir paket ulaştığını hayal edin; paketin işlenmesi için yönlendirici içinde bir yere bellekte yerleştirilmesi gerekir. Bu tür birçok paket ulaşırsa

```
// 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: **Örnek UDP Kodu (client.c, server.c)**

bir kez, yönlendirici içindeki belleğin tüm paketleri barındıramaması mümkündür. Yönlendiricinin bu noktada sahip olduğu tek seçenek, bir veya daha fazla paketi bırakmaktır (**drop**). Aynı davranış uç ana bilgisayarlarda da oluşur; tek bir ana çok sayıda mesaj gönderdiğinizde

makine kaynakları kolayca tükenebilir ve bu nedenle paket kaybı yeniden ortaya çıkar.

Bu nedenle, paket kaybı ağda esastır. Soru böylece ortaya çıkıyor: bununla nasıl başa çıkmalıyız?

48.2 48.2 Güvenilir Olmayan İletişim Katmanları

Basit bir yol şudur: bununla ilgilenmiyoruz. Bazı uygulamalar paket kaybıyla nasıl başa çıkılacağını bildiğinden, bazen temel, güvenilir olmayan bir mesajlaşma katmanıyla iletişim kurmalarına izin vermek yararlı olabilir;(**end-to-end argument**) (see the **Aside** at end of chapter) Böyle güvenilmez bir katmanın mükemmel bir örneği bulunur.

```
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: **Basit Bir UDP Kitaplığı (udp.c))**

İPUCU: DÜRÜSTLÜK İÇİN KONTROL TOPLAMLARI KULLANIN

Sağlama toplamları, modern sistemlerde bozulmayı hızlı ve etkili bir şekilde tespit etmek için yaygın olarak kullanılan bir yöntemdir. Basit bir sağlama toplamı toplamadır: sadece bir yığın verinin baytlarını toplayın; tabii ki, temel döngüsel artıklık kodları (CRC'ler), Fletcher sağlama toplamı ve diğerleri [MK09] dahil olmak üzere daha birçok başka gelişmiş sağlama toplamı oluşturulmuştur.

Ağda, sağlama toplamları aşağıdaki gibi kullanılır. Bir makineden diğerine mesaj göndermeden önce mesajın baytları üzerinden bir sağlama toplamı hesaplayın. Ardından hedefe hem mesajı hem de sağlama toplamını gönderin. Hedefte alıcı, gelen mesaj üzerinden de bir sağlama toplamı hesaplar; Bu hesaplanan sağlama toplamı, gönderilen sağlama toplamı ile eşleşirse, alıcı, verilerin iletim sırasında muhtemelen bozulmadığına dair bir miktar güvence hissedebilir.

Sağlama toplamları bir dizi farklı eksen boyunca değerlendirilebilir. Etkililik birincil bir hususdur: Verilerdeki bir değişiklik, sağlama toplamında bir değişikliğe yol açar mı? Sağlama toplamı ne kadar güçlüyse, verilerdeki değişikliklerin fark edilmemesi o kadar zor olur. Performans diğer önemli kriterdir: sağlama toplamının hesaplanması ne kadar maliyetlidir? Ne yazık ki, etkinlik ve performans genellikle çelişkilidir, bu da yüksek kaliteli sağlama toplamlarının hesaplanmasının genellikle pahalı olduğu anlamına gelir. Hayat yine mükemmel değil.

bugün neredeyse tüm modern sistemlerde bulunan **UDP/IP** ağ yığınında. UDP'yi kullanmak için bir işlem, bir iletişim uç noktası oluşturmak amacıyla yuva (**sockets**) API'sini kullanır; diğer makinelerdeki (veya aynı makinedeki) işlemler (**communication endpoint**), orijinal işleme UDP datagramları(**datagrams**) gönderir (bir datagram, maksimum boyuta 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 daha sonra bir yanıtla yanıt verir. Bu az miktarda kodla, dağıtılmış sistemler oluşturmaya başlamak için ihtiyacınız olan her şeye sahipsiniz!

UDP, güvenilmez bir iletişim katmanına harika bir örnektir. Kullanırsanız, paketlerin kaybolduğu (düştüğü) ve dolayısıyla hedeflerine ulaşamadığı durumlarla karşılaşırsınız; gönderici bu nedenle kayıptan asla haberdar edilmez. Ancak bu, UDP'nin aşağıdakilere karşı koruma sağlamadığı anlamına gelmez:

herhangi bir başarısızlık. Örneğin, UDP bazılarını (**checksum**) tespit etmek için bir sağlama toplamı içerir.

paket bozulması biçimleri.

Bununla birlikte, birçok uygulama yalnızca bir hedefe veri göndermek istediğinden ve paket kaybı konusunda endişelenmediğinden, daha fazlasına ihtiyacımız var. Özellikle, güvenilmez bir ağ 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. Basit düşünelim

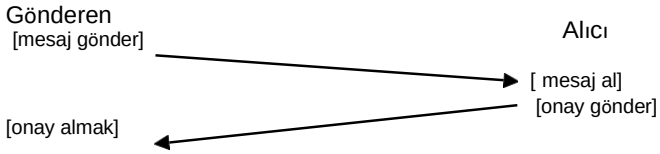


Figure 48.3: **Mesaj Artı Onay**

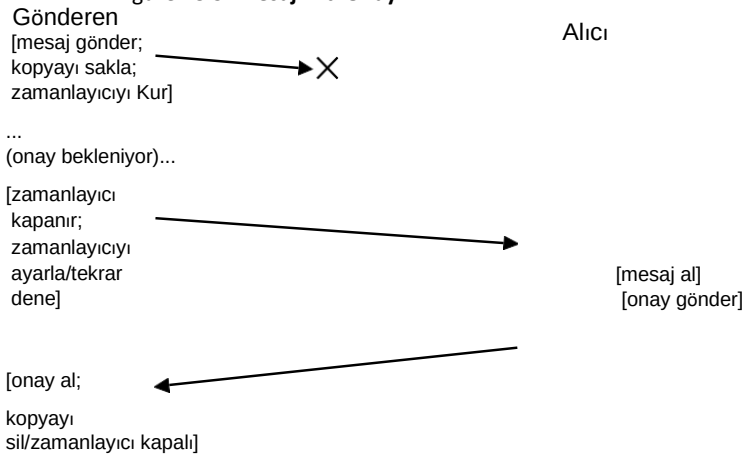


Figure 48.4: **Mesaj Artı Alındı: Bırakılan İstek**

bir istemcinin güvenilir olmayan bir bağlantı üzerinden bir sunucuya mesaj gönderdiği örnek. Cevaplamamız gereken ilk soru: Gönderen, alıcının mesajı gerçekten aldığını nasıl biliyor?

Kullanacağımız teknik, bir teşekkür(**acknowledgment**)veya kısaca **ack** olarak bilinir. Fikir basit: gönderen, alıcıya bir mesaj gönderir; alıcı daha sonra onay için kısa bir mesaj gönderir.

fiş. Şekil 48.3, süreci göstermektedir.

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

Bu durumu halletmek için zaman aşımı (**timeout**) olarak bilinen ek bir mekanizmaya ihtiyacımız var. Gönderen bir mesaj gönderdiğinde, gönderen artık belirli bir süre sonra kapanacak bir zamanlayıcı ayarlar. Bu süre içinde herhangi bir onay alınmazsa, gönderen mesajın kaybolduğu sonucuna varır. Gönderen daha sonra, göndermeyi yeniden dener ve bu sefer geçeceğini umarak aynı mesajı tekrar gönderir. Bu yaklaşımın işe yaraması için gönderenin, yeniden göndermesi gerekebileceği ihtimaline karşı iletinin bir kopyasını yanında bulundurması gerekir. Zaman aşımı ve yeniden deneme kombinasyonu, bazılarının yaklaşım zaman aşımı/yeniden (**timeout/retry**) deneme olarak adlandırmasına yol açtı; oldukça zeki kalabalık, bu ağ türleri, değil mi? Şekil 48.4 bir örneği göstermektedir

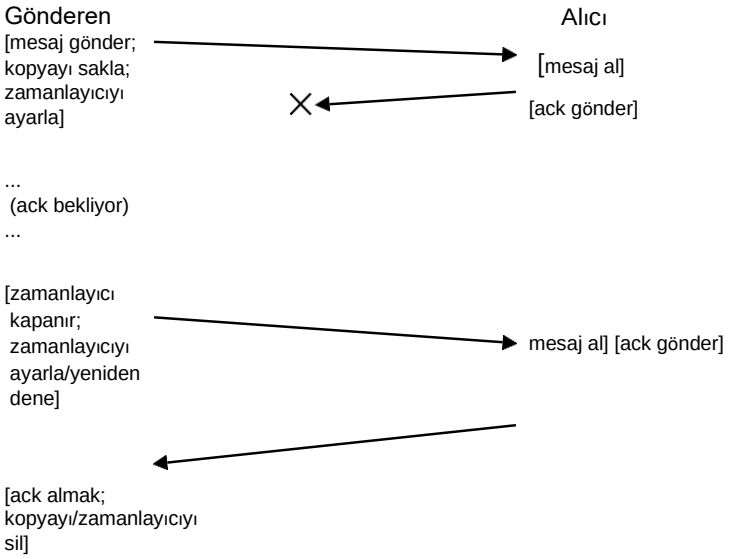


Figure 48.5: **Message Plus Acknowledgment: Dropped Reply**

Ne yazık ki, bu formda zaman aşımı/yeniden deneme yeterli değil. Figür

48.5, sorunlara yol açabilecek bir paket kaybı örneğini gösterir. Bu örnekte, kaybolan orijinal mesaj değil, onaydır. Gönderenin bakış açısından durum aynı görünüyor: herhangi bir onay alınmadı ve bu nedenle bir zaman aşımı ve yeniden deneme sırası var. Ancak alıcının bakış açısıyla durum oldukça farklıdır: şimdi aynı mesaj iki kez alınmıştır! Bunun uygun olduğu durumlar olsa da, genel olarak öyle değildir; Bir dosya indirirken ne olacağını hayal edin ve indirme işleminde fazladan paketler tekrarlanır. Bu nedenle, güvenilir bir mesaj katmanı hedeflediğimizde, genellikle şunu da garanti etmek isteriz:

her mesajın alıcı tarafından tam olarak bir kez (**exactly once**)alınmasıdır.

Alıcının yinelenen mesaj iletimini algılamasını sağlamak için, göndericinin her mesajı benzersiz bir şekilde tanımlaması ve alıcının her mesajı daha önce görüp görmediğini takip edecek bir yola ihtiyacı vardır. Alıcı yinelenen bir iletim gördüğünde, mesajı kontrol eder, ancak (kritik olarak) mesajı verileri alan uygulamaya iletmez. Böylece, gönderen onay alır, ancak mesaj iki kez alınmaz, yukarıda belirtilen tam olarak bir kez semantik korunur.

Yinelenen mesajları algılamanın 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 gerektirdiği için çok maliyetlidir.

Az bellek gerektiren daha basit bir yaklaşım bu sorunu çözer ve mekanizma sıra sayacı (**sequence counter**.) olarak bilinir. Bir sıra sayacıyla, gönderici ve alıcı, her iki tarafın da koruyacağı bir sayaç için bir başlangıç değeri (örneğin, 1) üzerinde anlaşırlar. Her mesaj gönderildiğinde, geçerli akım

mesajın yanında sayaç değeri de gönderilir; bu sayaç değeri (N), mesaj için bir kimlik işlevi görür. Mesaj gönderildikten sonra, gönderen değeri artırır (N+1'e).

İPUCU: ZAMAN AŞIMI DEĞERİNİ AYARLARKEN DİKKATLİ OLUN

Tartışmadan muhtemelen tahmin edebileceğiniz gibi, zaman aşımı değerini doğru ayarlamak, mesaj gönderimlerini yeniden denemek için zaman aşımlarını kullanmanın önemli bir yönüdür. Zaman aşımı çok küçükse, gönderici gereksiz yere iletileri yeniden gönderir, böylece gönderici ve ağ kaynakları üzerinde CPU zamanı boşa harcanır. 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, bu nedenle, paket kaybını algılamak için yeterince uzun süre beklemek ama daha fazla beklemek değildir.

Bununla birlikte, gelecek bölümlerde göreceğimiz gibi, dağıtılmış 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. Doğruysa, istemciler farklı bir uyarlamalı şekilde yeniden deneyebilir; örneğin, ilk zaman aşımından sonra, bir müşteri zaman aşımı değerini daha yüksek bir miktara, belki de iki katına çıkarabilir.

orijinal değer kadar yüksek. Böyle bir üstel geri çekilme (**exponential back-off**) planı, pio-

Erken Aloha ağında gerekli olan ve erken Ethernet'te [A70] benimsenen, aşırı yeniden gönderme nedeniyle kaynakların aşırı yüklendiği durumlardan kaçınır. Sağlam sistemler, bu nitelikteki aşırı yükten kaçınmaya çalışır.

Alıcı, sayaç değerini, gönderenden gelen iletinin kimliği için beklenen değer olarak kullanır. Alınan bir adaçayının (N) kimliği alıcının sayacı (ayrıca N) ile eşleşiyorsa, mesajı açar ve uygulamaya iletir; bu durumda, alıcı bu mesajın ilk kez alındığı sonucuna varır. Alıcı daha sonra artar- sayacını (N + 1'e) getirir ve bir sonraki mesajı bekler. Ack kaybolursa, gönderen zaman aşımına uğrar ve N mesajını yeniden gönderir. Bu kez, alıcının sayacı daha yüksektir (N + 1) ve böylece alıcı bu mesajı zaten aldığını bilir. Böylece mesajı acks eder, ancak uygulamaya iletmez. Bu basit şekilde, sıra sayaçları kopyaları önlemek için kullanılabilir. En yaygın kullanılan güvenilir iletişim katmanı **TCP/IP** veya kısaca **TCP** olarak bilinir. TCP, ağdaki tıkanıklığı ele alan makineler [VJ88], birden fazla bekleyen istek ve yüzlerce başka dahil olmak üzere yukarıda açıkladığımızdan çok daha fazla karmaşıklığa sahiptir. küçük ince ayarlar ve optimizasyonlar. Merak ediyorsanız bu konuda daha fazla bilgi edinin; daha da iyisi, bir ağ kursuna katılın ve bu materyali iyi öğrenin.

48.4 İletişim Soyutlamaları

Temel bir mesajlaşma katmanı göz önüne alındığında, şimdi bu bölümdeki bir sonraki soruya yaklaşıyoruz: dağıtılmış bir sistem oluştururken iletişimin hangi soyutlamasını kullanmalıyız? Sistem topluluğu yıllar içinde bir dizi yaklaşım geliştirdi. Bir grup çalışma, işletim sistemi soyutlamalarını aldı ve bunları genişletti. dağıtılmış bir ortamda çalışmak. Örneğin, dağıtılmış paylaşılan bellek (**distributed shared memory** (**DSM**))sistemleri, farklı makinelerdeki işlemlerin büyük, sanal bir adres alanını [LH89] paylaşmasını sağlar. Bu soyutlama dağıtılmış bir çok iş parçacıklı bir uygulamaya benzeyen bir şeye hesaplama; tek fark, bu iş parçacıklarının aynı makine içindeki farklı

işlemciler yerine farklı makinelerde çalışmasıdır. Çoğu DSM sisteminin çalışma şekli, işletim sisteminin sanal bellek sisteminden geçer. Bir makinede bir sayfaya erişildiğinde, iki şey olabilir. İlk (en iyi) durumda, sayfa makinede zaten yereldir ve bu nedenle veriler hızlı bir şekilde alınır. İkinci durumda, sayfa şu anda başka bir makinededir. Bir sayfa hatası oluşur ve sayfa hatası işleyicisi, sayfayı getirmesi, istekte bulunan işlemin sayfa tablosuna yüklemesi ve yürütmeye devam etmesi için başka bir makineye bir ileti gönderir. Bu yaklaşım günümüzde çeşitli nedenlerden dolayı yaygın olarak kullanılmamaktadır. DSM için en büyük sorun, başarısızlığı nasıl ele aldığıdır. Örneğin, bir makine arızalanırsa düşünün; o makinedeki sayfalara ne olur? Dağıtılmış hesaplamanın veri yapıları tüm adres alanına yayılmışsa ne olur? Bu durumda, bu veri yapılarının bazı kısımları aniden kullanılamaz hale gelir. Adres alanınızın bazı bölümleri kaybolduğunda başarısızlıkla başa çıkmak zordur; "Sonraki" işaretçinin adres alanının gitmiş olan bir bölümünü işaret ettiği bağlantılı bir liste hayal edin. Yikes! Diğer bir sorun ise performanstır. Kod yazarken genellikle belleğe erişimin ucuz olduğu varsayılır. DSM sistemlerinde, bazı erişimler ucuzdur, ancak diğerleri sayfa hatalarına ve uzak makinelerden pahalı getirmelere neden olur. Bu nedenle, bu tür DSM sistemlerinin programcıları, hesaplamaları neredeyse hiç iletişim gerçekleşmeyecek şekilde organize etmek için çok dikkatli olmak zorunda kaldılar ve bu da böyle bir yaklaşımın amacının çoğunu yendi. Bu alanda çok fazla araştırma yapılmasına rağmen, çok az pratik etki vardı; Bugün hiç kimse DSM'yi kullanarak güvenilir dağıtılmış sistemler kurmuyor.

48.5 Uzaktan Yordam Çağrısı (RPC)

İşletim sistemi soyutlamalarının dağıtılmış sistemler oluşturmak için kötü bir seçim olduğu ortaya çıkarken, programlama dili (PL) soyutlamaları çok daha anlamlıdır. En baskın soyutlama, uzaktan prosedür çağrısı veya kısaca RPC [BN84]1 fikrine dayanmaktadır.

Uzaktan yordam çağrı paketlerinin hepsinin basit bir amacı vardır: uzak bir makinede kod yürütme sürecini, yerel bir işlevi çağırmak kadar basit ve anlaşılır hale getirmek. Böylece müşteriye bir prosedür çağrısı yapılır ve bir süre sonra sonuçlar döndürülür. Sunucu, dışa aktarmak istediği bazı yordamları tanımlar. Büyünün geri kalanı

genellikle iki parçadan oluşan RPC sistemi tarafından işlenir: bir saplama oluşturucu (bazen protokol derleyici olarak adlandırılır) ve çalışma zamanı kitaplığı.

Şimdi bu parçaların her birine daha ayrıntılı olarak göz atacağız

---

1Modern programlama dillerinde bunun yerine uzak yöntem çağırma diyebiliriz (RMI), ama bu dilleri tüm süslü nesneleriyle kim seviyor?

Taslak Oluşturucu (Stub Generator)

Taslak üretecinin işi basittir: işlev bağımsız değişkenlerini ve sonuçlarını otomatikleştirerek mesajlara dönüştürme zahmetinden bazılarını ortadan kaldırmak. Çok sayıda fayda ortaya çıkar: kişi, bu tür kodları elle yazarken meydana gelen basit hatalardan tasarım gereği kaçınır; ayrıca, bir saplama derleyicisi bu tür bir kodu optimize edebilir ve böylece performansı artırabilir.

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

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

Saplama üreteci bunun gibi bir arayüz alır ve birkaç farklı kod parçası üretir. İstemci için, arabirimde belirtilen işlevlerin her birini içeren bir istemci saplaması (**client stub**) oluşturulur; bir müşteri programı

Bu RPC hizmetini kullanmak isteyenler, RPC'ler yapmak için bu istemci stub'ına bağlanır ve onu arar.

Dahili olarak, istemci saplamasındaki bu işlevlerin her biri, uzaktan yordam çağrısını gerçekleştirmek için gereken tüm işi yapar. İstemciye, kod yalnızca bir işlev çağrısı olarak görünür (örneğin, müşteri func1(x)'i çağırır); func1() için istemci saplamasındaki kod dahili olarak şunu yapar:

- İleti arabelleği oluşturun(**Create a message buffer**). İleti arabelleği genellikle yalnızca belirli bir boyuttakibitişik bir bayt dizisidir.
- Gerekli bilgileri mesaj arabelleğine paketleyin (**Pack the needed information into the message buffer**). Bu bilgi, çağrılacak işlev için bir tür tanımlayıcı içerir.
- fonksiyonun ihtiyaç duyduğu tüm bağımsız değişkenlerin yanı sıra (örneğin, yukarıdaki örneğimizde, func1 için bir tamsayı). Tüm bu bilgileri tek bir bitişik arabelleğe koyma işlemine bazen argümanların sıralanması veya mesajın seri hale getirilmesi denir.
- Mesajı hedef RPC sunucusuna gönderin(**Send the message to the destination RPC server**). RPC sunucusuyla iletişim ve bunun için gereken tüm ayrıntılar
- düzgün çalışır, aşağıda daha ayrıntılı açıklanan RPC çalışma zamanı kitaplığı tarafından işlenir
- Cevabı bekleyin(**Wait for the reply**). İşlev çağrıları genellikle senkronize(**synchronous**) olduğundan, çağrı tamamlanmasını bekler.
- Dönüş kodunu ve diğer bağımsız değişkenleri paketinden çıkarın(**Unpack return code and other arguments.**). İşlev yalnızca tek bir dönüş kodu döndürürse, bu işlem basittir; fakat,
- daha karmaşık işlevler daha karmaşık sonuçlar (örneğin bir liste) döndürebilir ve bu nedenle saplamanın bunları da açması gerekebilir.

Bu adım aynı zamanda **unmarshaling** veya **deserialization** olarak da bilinir.

- arayana dönün(**Return to the caller**). Son olarak, müşteri saplamasından müşteri koduna geri dönün

Sunucu için kod da üretilir. Sunucu üzerinde yapılan işlemler şu şekildedir:

• Mesajı paketinden çıkarın(**Unpack the message**). Sıralamadan çıkarma veya seri durumdan çıkarma (**unmarshaling** or **deserial- ization**) adı verilen bu adım, gelen mesajdaki bilgileri alır. Bu işlev tanımlayıcısı ve bağımsız değişkenler çıkarılır.
• Gerçek işlevi arayın (**Call into the actual function**). Nihayet! Uzak işlevin fiilen yürütüldüğü noktaya ulaştık. RPC çalışma zamanı kimlik tarafından belirtilen işlevi çağırır ve istenen bağımsız değişkenleri iletir.
• Sonuçları paketleyin (**Package the results**). Dönüş bağımsız değişkenleri, tek bir yanıt arabelleğine geri sıralanır.
• Cevabı gönderin (**Send the reply**). Cevap sonunda arayana gönderilir.

Bir taslak derleyicide dikkate alınması gereken birkaç önemli konu daha vardır. İlki karmaşık argümanlardır, yani karmaşık bir veri yapısı nasıl paketlenir ve gönderilir? Örneğin, write() sistem çağrısı çağrıldığında, üç bağımsız değişken iletilir: bir tamsayı dosya tanıtıcısı, bir tampon işaretçisi ve kaç baytın (işaretçiden başlayarak) yazılacağını gösteren bir boyut. Bir RPC paketi bir işaretçi iletilirse, bu işaretçiyi nasıl yorumlayacağını bulması ve doğru eylemi gerçekleştirmesi gerekir. Genellikle bu, iyi bilinen türler aracılığıyla (örneğin, RPC derleyicisinin anladığı, belirli bir boyuttaki veri yığınlarını iletmek için kullanılan bir arabellek t) veya veri yapılarına daha fazla bilgi ekleyerek, derleyiciyi etkinleştirerek gerçekleştirilir. hangi baytların serileştirilmesi gerektiğini bilmek için.
Bir diğer önemli konu da eş zamanlılık açısından sunucunun organizasyonudur. Basit bir sunucu, istekleri basit bir döngüde bekler ve her isteği teker teker işler. Ancak, tahmin edebileceğiniz gibi, bu büyük ölçüde verimsiz olabilir; bir RPC çağrısı engellenirse (örn. G/Ç'de), sunucu kaynakları boşa harcanır. Bu nedenle, çoğu sunucu
bir çeşit eşzamanlı moda. Yaygın bir organizasyon bir iş parçacığı (**thread pool**) havuzudur.
Bu organizasyonda, sunucu başladığında sınırlı sayıda iş parçacığı oluşturulur; bir mesaj geldiğinde, bu çalışan iş parçacığından birine gönderilir, bu da daha sonra RPC çağrısının işini yapar ve sonunda yanıt verir; bu süre zarfında, bir ana iş parçacığı diğer istekleri almaya devam eder ve belki de bunları diğer çalışanlara gönderir. Böyle bir organizasyon, sunucu içinde eşzamanlı yürütmeyi mümkün kılar ve böylece kullanımını artırır; RPC çağrılarının artık doğru çalışmasını sağlamak için kilitleri ve diğer senkronizasyon ilkelerini kullanması gerekebileceğinden, çoğunlukla programlama karmaşıklığında standart maliyetler de ortaya çıkar.

Çalışma zamanı kitaplığı

Çalışma zamanı kitaplığı, bir RPC sistemindeki ağır yüklerin çoğunu işler; 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, uzak bir hizmetin nasıl bulunacağıdır. Bu adlandırma sorunu, dağıtılmış sistemlerde yaygın bir sorundur ve bir anlamda mevcut tartışmamızın kapsamını aşar.

siyon. En basit yaklaşımlar, örneğin mevcut internet protokolleri tarafından sağlanan ana bilgisayar adları ve port numaraları gibi mevcut adlandırma sistemleri üzerine kuruludur. Böyle bir sistemde, istemci, kullanmakta olduğu bağlantı noktası numarasının yanı sıra istenen RPC hizmetini çalıştıran makinenin ana bilgisayar adını veya IP adresini bilmelidir (bir bağlantı noktası numarası, yalnızca gerçekleşen belirli bir iletişim faaliyetini tanımlamanın bir yoludur. aynı anda birden fazla iletişim kanalına izin veren bir makinede). Protokol paketi daha sonra paketleri sistemdeki herhangi bir başka makineden belirli bir adrese yönlendirmek için bir mekanizma sağlamalıdır. Adlandırmayla ilgili iyi bir tartışma için başka bir yere bakmanız gerekecek, örn., İnternette DNS ve ad çözümlemesi hakkında bir şeyler okuyun veya daha iyisi Saltzer ve Kaashoek'in [SK09] kitabındaki mükemmel bölümü okuyun.

Bir istemci, belirli bir uzak hizmet için hangi sunucuyla konuşması gerektiğini öğrendiğinde, sonraki soru, RPC'nin hangi aktarım düzeyi protokolünün üzerine inşa edilmesi gerektiğidir. Spesifik olarak, RPC sistemi TCP/IP gibi güvenilir bir protokol mü kullanmalı yoksa UDP/IP gibi güvenilir olmayan bir iletişim katmanı üzerine mi inşa edilmelidir?

Naif bir şekilde seçim kolay görünüyor: Açıkça bir talebin uzak sunucuya güvenilir bir şekilde iletilmesini istiyoruz ve açık bir şekilde güvenilir bir şekilde yanıt almak istiyoruz. Bu nedenle TCP gibi güvenilir aktarım protokolünü seçmeliyiz, değil mi?

Ne yazık ki, RPC'yi güvenilir bir iletişim katmanının üzerine inşa etmek, performansta büyük bir verimsizliğe yol açabilir. Yukarıdaki tartışmadan, güvenilir iletişim katmanlarının nasıl çalıştığını hatırlayın: onaylar artı zaman aşımı/yeniden deneme ile. Böylece, istemci sunucuya bir RPC isteği gönderdiğinde, arayan kişinin isteğin alındığını bilmesi için sunucu bir onayla yanıt verir. Benzer şekilde, sunucu istemciye yanıtı gönderdiğinde, sunucunun yanıtın alındığını bilmesi için istemci yanıtı onaylar. Güvenilir bir iletişim katmanının üzerinde bir istek/yanıt protokolü (RPC gibi) oluşturarak, iki "ekstra" mesaj gönderilir.

Bu nedenle birçok RPC paketi, UDP gibi güvenilir olmayan iletişim katmanları üzerine kuruludur. Bunu yapmak, daha verimli bir RPC katmanı sağlar, ancak RPC sistemine güvenilirlik sağlama sorumluluğunu da eklemez. RPC katmanı, yukarıda açıkladığımız gibi zaman aşımı/yeniden deneme ve onayları kullanarak istenen sorumluluk düzeyine ulaşır. Bir tür sıra numaralandırma kullanarak, iletişim katmanı her bir RPC'nin tam olarak bir kez (arıza olmaması durumunda) veya en fazla bir kez (arızanın ortaya çıkması durumunda) gerçekleşmesini garanti edebilir.

Diğer sorunlar

Bir RPC çalışma zamanının da işlemesi 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 başarısızlık olarak görünebilir, bu nedenle yeniden denemeyi tetikleyebilir ve bu nedenle burada biraz dikkat edilmesi gerekebilir. Bir çözüm, açık bir onay kullanmaktır.

Kenara: UÇTAN UCA TARTIŞMA

Uçtan uca argüman, bir sistemdeki en yüksek düzeyin, yani genellikle "sondaki" uygulamanın, katmanlı bir sistem içinde belirli işlevlerin gerçekten uygulanamadığı tek yerel konum olduğunu öne sürer. tamamlandı. Dönüm noktası niteliğindeki makalelerinde [SRC84], Saltzer ve ark. Bunu mükemmel bir örnekle tartışın: iki makine arasında güvenilir dosya aktarımı. 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 ağ veya diskteki daha düşük düzeydeki güvenilir makineler böyle bir garanti sağlamaz.

Kontrast, güvenilir dosya aktarımı sorununu sistemin alt katmanlarına güvenilirlik ekleyerek çözmeye çalışan bir yaklaşımdır. Örneğin, güvenilir bir iletişim protokolü oluşturduğumuzu ve bunu güvenilir dosya aktarımımızı oluşturmak için kullandığımızı varsayalım. İletişim protokolü, bir gönderici tarafından gönderilen her baytın, örneğin zaman aşımı/tekrar deneme, alındı bildirimleri ve sıra numaraları kullanılarak alıcı tarafından sırayla alınacağını garanti eder. Ne yazık ki, böyle bir protokol kullanmak güvenilir bir dosya aktarımı sağlamaz; 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 edilmiş olsa da, dosya aktarımımız sonuçta güvenilir değildi. Güvenilir bir dosya aktarımı oluşturmak için uçtan uca güvenilirlik kontrolleri yapılmalıdır; örneğin, tüm aktarım tamamlandıktan sonra, dosyayı alıcı diskte tekrar okuyun, bir sağlama toplamı hesaplayın ve bu sağlama toplamını dosyanınkiyle karşılaştırın gönderen üzerinde.

Bu kuralın doğal sonucu, bazen daha düşük katmanlara sahip olmanın ekstra işlevsellik sağlamasının gerçekten de sistem performansını artırabilmesi veya başka bir şekilde bir sistemi optimize edebilmesidir. Bu nedenle, bir sistemde daha düşük bir seviyede bu tür makinelere sahip olmayı göz ardı etmemelisiniz; bunun yerine, genel bir sistem veya uygulamada nihai kullanımı göz önüne alındığında, bu tür makinelerin faydasını dikkatlice düşünmelisiniz.

(alıcıdan gönderene) yanıt hemen üretilmediğinde; bu, istemcinin sunucunun isteği aldığını bilmesini sağlar. Sonra, bir süre sonra

geçti, istemci sunucunun hala istek üzerinde çalışıp çalışmadığını periyodik olarak sorabilir; sunucu "evet" demeye devam ederse, istemci mutlu olmalı ve beklemeye devam etmelidir (sonuçta, bazen bir prosedür çağrısının yürütülmesinin tamamlanması uzun zaman alabilir).

Çalışma zamanı ayrıca, tek bir pakete sığabilecek olandan daha büyük, büyük bağımsız değişkenlere sahip prosedür çağrılarını da işlemelidir. Bazı alt düzey ağ protokolleri, bu tür gönderici tarafı parçalanmasını (**fragmentation**) (daha büyük paketlerin daha küçük paketler halinde) ve alıcı tarafında yeniden birleştirmeyi (**reassembly**)(daha küçük parçaların daha büyük bir mantıksal bütün halinde) sağlar; değilse, RPC çalışma zamanının bu tür bir işlevi kendisi uygulaması gerekebilir. Ayrıntılar için Birrell ve Nelson'ın makalesine bakın

Birçok sistemin ele aldığı sorunlardan biri bayt sıralamasıdır(**byte ordering**). Bildiğiniz gibi, bazı makineler değerleri big endian sıralaması olarak bilinen şekilde depolarken, diğerleri **little endian** sıralaması kullanır**. Big endian**, baytları (diyelim ki bir tamsayının) en önemli bitlerinden en önemsiz bitlerine kadar, Arap rakamlarına çok benzer şekilde saklar; küçük endian tam tersini yapar. Her ikisi de sayısal bilgileri depolamanın eşit derecede geçerli yollarıdır; Buradaki soru, farklı endianness'e sahip makineler arasında nasıl iletişim kurulacağıdır.

RPC paketleri genellikle mesaj formatlarında iyi tanımlanmış bir endianlık sağlayarak bunu halleder. Sun'ın RPC paketinde, **XDR** (**eXternal Data Representation**) katmanı bu işlevi sağlar. Bir mesaj gönderen veya alan makine, XDR'nin endianlığıyla eşleşirse, mesajlar beklendiği gibi gönderilir ve alınır. Bununla birlikte, iletişim kuran makinenin farklı bir sonu varsa, her bir bilgi parçası

mesajın dönüştürülmesi gerekir. Böylece, endianness farkı küçük bir performans maliyetine sahip olabilir.

Son bir konu, iletişimin eşzamansız (**synchronously**) doğasını istemcilere gösterip göstermemek, böylece bazı performans iyileştirmelerini mümkün kılmaktır. Spesifik olarak, tipik RPC'ler eşzamanlı (**asynchronously**) olarak yapılır, yani bir müşteri prosedür çağrısını yayınladığında, prosedür çağrısının geri dönmesini beklemesi gerekir.

devam etmeden önce. Bu bekleme uzun olabileceğinden ve istemcinin yaptığı başka işler olabileceğinden, bazı RPC paketleri bir RPC'yi eşzamansız olarak çağırmanıza olanak tanır. Eşzamansız bir RPC verildiğinde, RPC paketi isteği gönderir ve hemen geri döner; the

müşteri daha sonra diğer RPC'leri aramak veya diğer yararlı hesaplamalar gibi diğer işleri yapmakta özgürdür. İstemci bir noktada eşzamansız RPC'nin sonuçlarını görmek isteyecektir; bu nedenle RPC katmanını geri çağırarak, bekleyen RPC'lerin tamamlanmasını beklemesini söyler, bu noktada geri dönüş bağımsız değişkenlerine erişilebilir.

48.6 Özet

Yeni bir konunun, dağıtılmış sistemlerin ve onun ana sorununun ortaya çıktığını gördük: artık sıradan bir olay olan arızanın nasıl ele alınacağı. Google'ın içinde dedikleri gibi, yalnızca masaüstü makineniz olduğunda başarısızlık nadirdir; binlerce makinenin olduğu bir veri merkezinde olduğunuzda, her zaman arıza oluyor. Herhangi bir dağıtılmış sistemin anahtarı, bu başarısızlıkla nasıl başa çıkacağınızdır.

İletişimin herhangi bir dağıtılmış sistemin kalbini oluşturduğunu da gördük. Bu iletişimin ortak bir soyutlaması, istemcilerin sunucular üzerinde uzaktan aramalar yapmasını sağlayan uzaktan yordam çağrısında (RPC) bulunur; RPC paketi, yerel bir prosedür çağrısını yakından yansıtan bir hizmet sunmak için zaman aşımı/yeniden deneme ve onay dahil olmak üzere tüm kanlı ayrıntıları işler.

Bir RPC paketini gerçekten anlamanın en iyi yolu elbette kendiniz kullanmaktır. Sun'ın rpcgen saplama derleyicisini kullanan RPC sistemi daha eskidir; Google'ın gRPC'si ve Apache Thrift, modern yaklaşımlardır. Birini deneyin ve tüm yaygaranın ne hakkında olduğunu,görün.

References

[A70] "The ALOHA System — Another Alternative for Computer Communications" by Nor- man 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.*

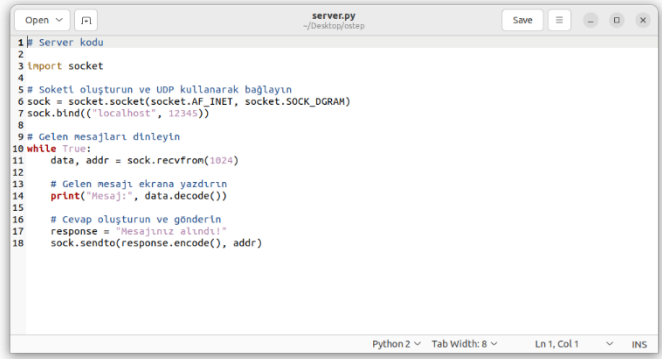
Ödev (Code)

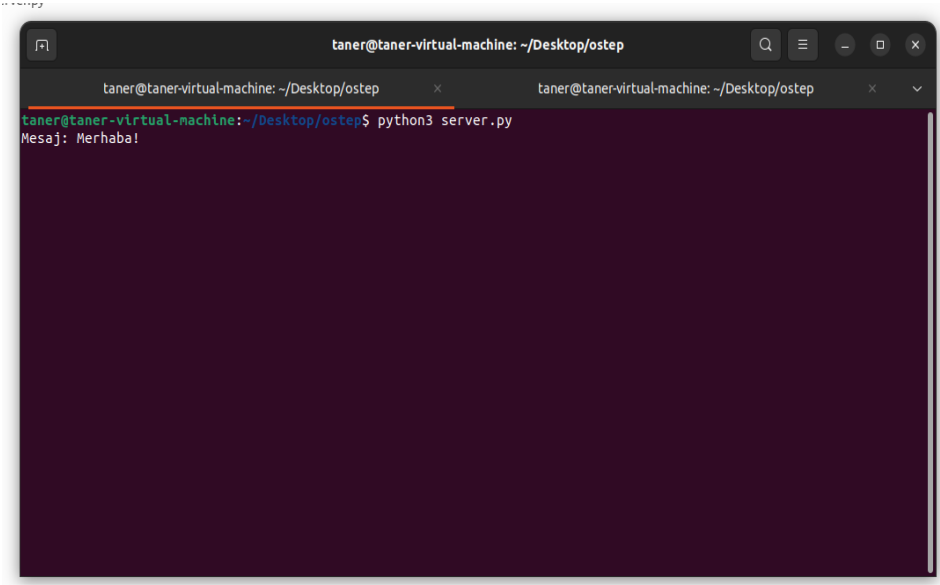
Bu bölümde, bunu yapma görevine aşina olmanız için bazı basit iletişim kodları yazacağız. İyi eğlenceler!

Sorular

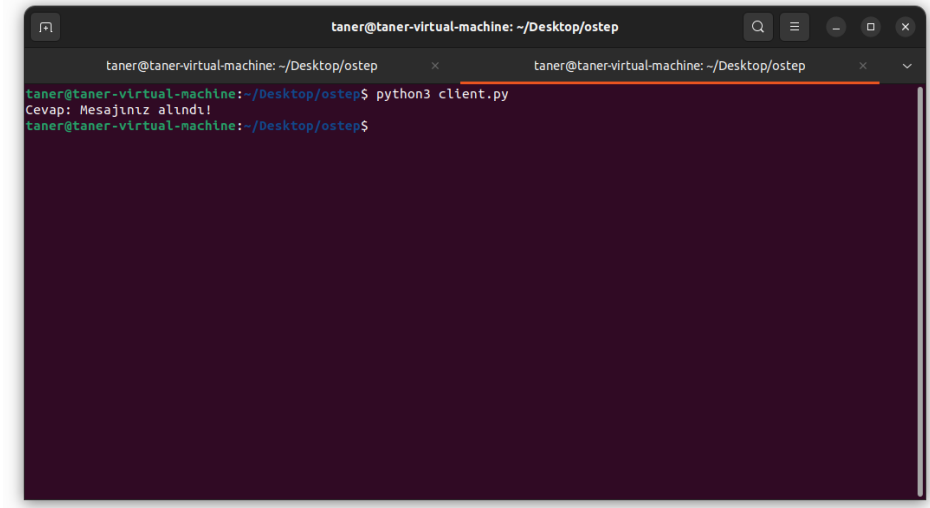
1. 1. Bölümde verilen kodu kullanarak basit bir UDP tabanlı sunucu ve istemci oluşturun. Sunucu, istemciden mesajlar almalı ve bir onay ile yanıt vermelidir. Bu ilk denemede herhangi bir yeniden iletim veya sağlamlık eklemeyin (iletişimin kusursuz çalıştığını varsayın). Bunu test için tek bir makinede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

Merhaba arkadaşlar öncelikle bu kodlar bu kodlar, bir istemci ve bir sunucu arasında bir UDP bağlantısı kurarak haberleşmeyi göstermektedir. İstemci(Client), sunucuya(surver) "Merhaba!" mesajını gönderir ve sunucu bu mesajı alır, cevaplar ve cevabı istemciye gönderir. Bu örnek, UDP protokolünün nasıl kullanılabileceğini göstermektedir.





Bu iki resimde de görüyoruz ki udp sistemimiz system içinde birbirinden haber alabiliyor.(python kodları ile)



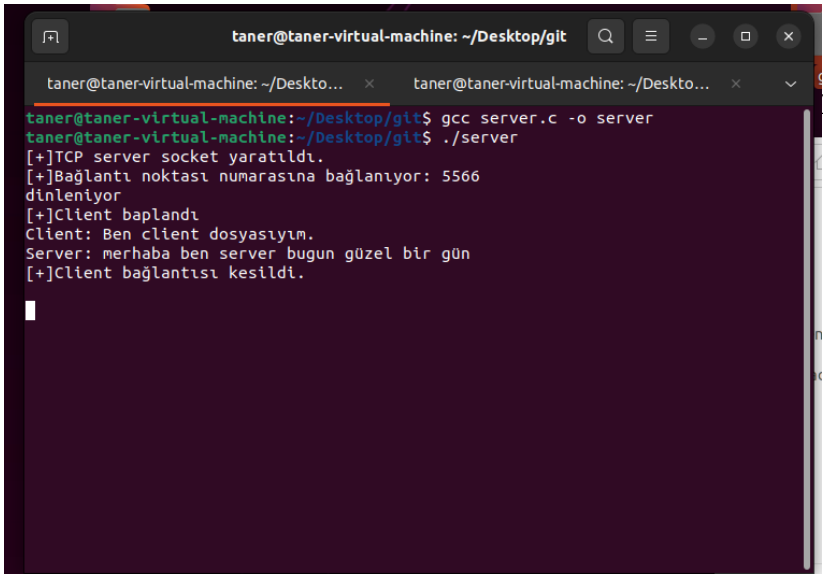
Şimdi de c kodları ile çıktılarımızı göreceğiz

Client Kodu

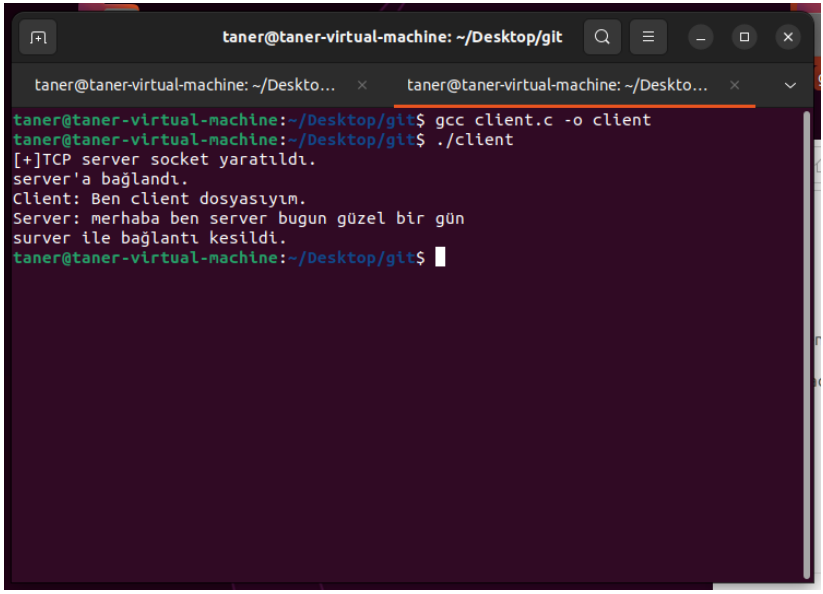
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

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 yaratıldı.\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'a bağlandı.\n");

  bzero(buffer, 1024);
  strcpy(buffer, "Ben client dosyasıyım.");
  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("surver ile bağlantı kesildi.\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 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 yaratıldı.\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'a bağlandı.\n");

  bzero(buffer, 1024);
  strcpy(buffer, "Ben client dosyasıyım.");
  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("surver ile bağlantı kesildi.\n");

  return 0;
}
```



Burada da C kodu ile verilen cıktıları görüyoruz



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

Client kodu:

```
// 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;
}
```

Surver kodu:

```
// 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;
}
```

Kütüphane kodu:

```
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);
}
```

Bu C dilinde yazılmış bir UDP kliyent-sunucu kodu parçacığıdır. Bu kod parçacığında, bir UDP soketi oluşturulup, bir porta bağlanılır ve bu soket üzerinden veri gönderme ve alma işlemleri gerçekleştirilir.

İlk olarak, UDP_Open() fonksiyonu bir port numarası alır ve bu porta bağlanacak bir UDP soketi oluşturmaya çalışır. Eğer soket oluşturulamazsa, fonksiyon -1 değerini döndürür. Eğer soket oluşturulursa, fonksiyon soketin dosya tanımlayıcısını (sd değişkeni) döndürür.

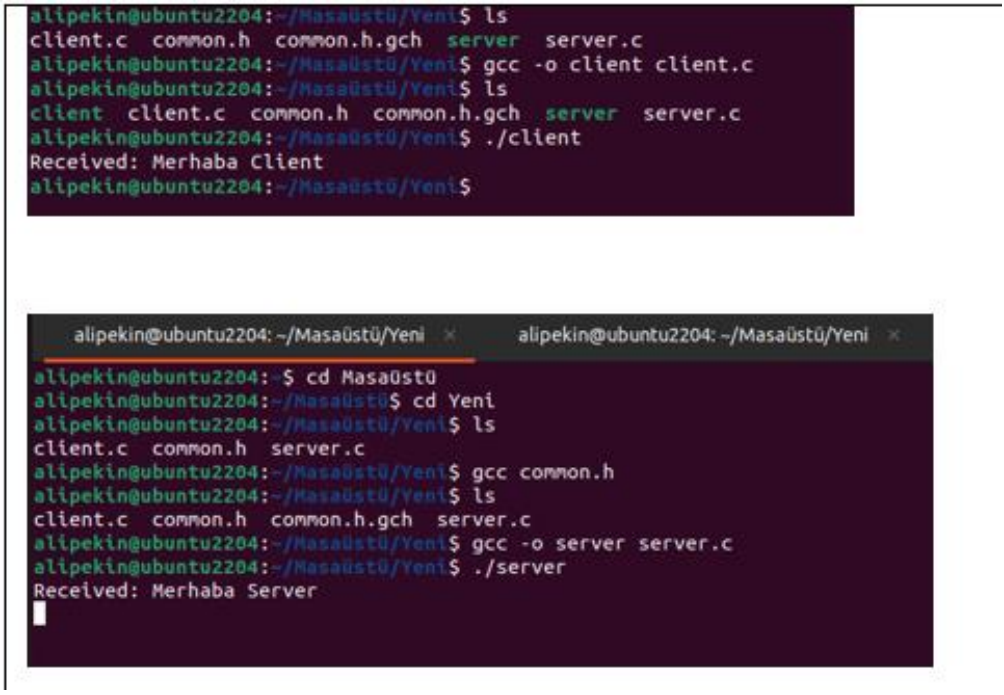
Sonra, UDP_FillSockAddr() fonksiyonu bir sockaddr_in yapısı ve bir host adı alır ve bu yapıda belirtilen host adına ait IP adresini doldurmaya çalışır. Eğer host adına ait IP adresi bulunamazsa, fonksiyon -1 değerini döndürür.

UDP_Write() fonksiyonu ise bir soket dosya tanımlayıcısı, sockaddr_in yapısı, bir veri tamponu ve tamponun uzunluğu alır ve bu veri tamponunu verilen soket üzerinden verilen sockaddr_in yapısına göndermeye çalışır. Fonksiyon, gönderilen veri miktarını döndürür.

UDP_Read() fonksiyonu ise bir soket dosya tanımlayıcısı, sockaddr_in yapısı, bir veri tamponu ve tamponun uzunluğu alır ve bu soket üzerinden veri okumaya çalışır. Okunan veri, verilen tampona yazılır ve fonksiyon, okunan veri miktarını döndürür.

Bu kod parçacığında, soketlerin timeout değerleri ayarlanmamıştır ve soketlerin recvfrom() fonksiyonları sonsuza kadar bekleyebilir. Eğer zaman aşımı olmasını istiyorsanız, soketlerin timeout değerlerini ayarlayarak zaman aşımını ele alabilirsiniz.

Server ve client programları çalıştırıldıktan sonra ekran görüntüleri



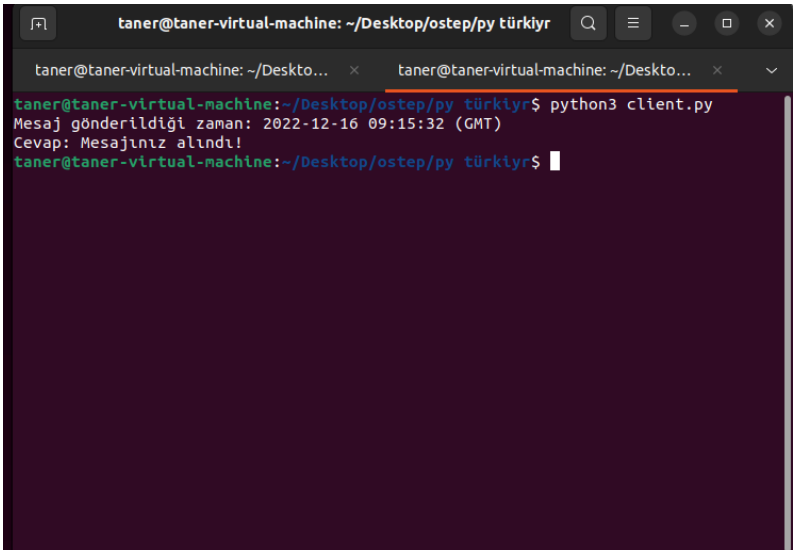
3. Gelişmekte olan iletişim kitaplığınıza zaman aşımı/yeniden deneme şeklinde güvenilir iletişim ekleyin. Özellikle, kütüphaneniz göndereceği herhangi bir mesajın bir kopyasını almalıdır. Gönderirken, bir zamanlayıcı başlatmalıdır, böylece mesajın gönderilmesinden bu yana ne kadar zaman geçtiğini takip edebilir. Alıcıda, kitaplık alınan mesajları onaylamalıdır. Gönderme istemcisi gönderirken engellemeli, yani geri dönmeden önce mesaj onaylanana kadar beklemelidir. Ayrıca süresiz olarak yeniden göndermeyi denemeye istekli olmalıdır. Maksimum mesaj boyutu, en büyük mesaj boyutu olmalıdır.
UDP ile gönderebileceğiniz tek mesaj. Son olarak, bir onay gelene veya iletim zaman aşımına uğrayana kadar arayanı uyku moduna alarak zaman aşımını/tekrar denemeyi verimli bir şekilde gerçekleştirdiğinizden emin olun; CPU'yu döndürüp boşa harcamayın!

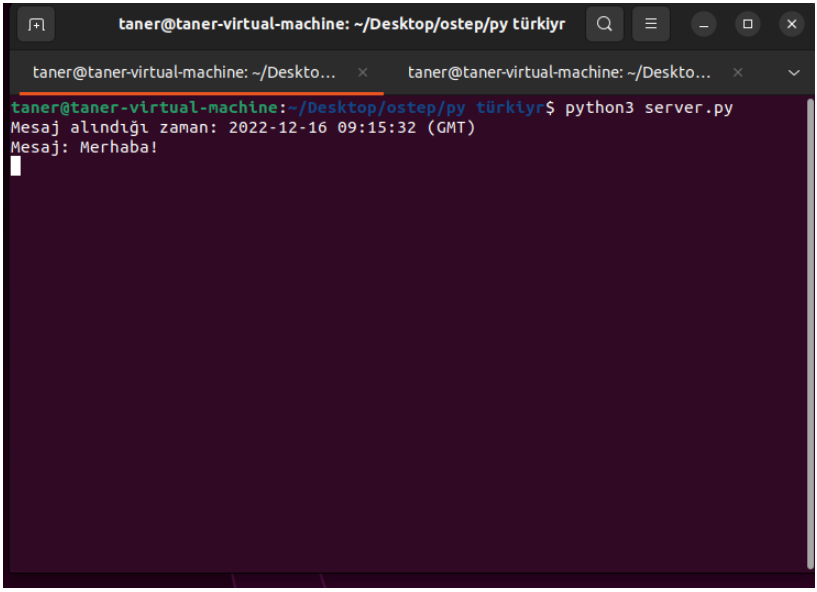
Öncelikle burada bizim ihtiyacımız olan time methodunu kullanarak systemin mesaj gönderdiği anın saniyesini kaydedip sonrasında ise normal zamandan cıkarak system zamanını buluyoruz aşağıda ise bu kodların çıktısını görebilirsiniz

```python
# İstemci kodu

import socket
import time

# Soketi oluşturun ve UDP kullanarak bağlayın
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Mesajı server'a gönderin
message = "Merhaba!"
send_time = time.time() # mesajın gönderildiği zamanı alın
sock.sendto(message.encode(), ("localhost", 12345))

# Cevabı alın ve ekrana yazdırın
data, addr = sock.recvfrom(1024)
send_time_tuple = time.gmtime(send_time) # Unix zamanını UTC koordinatif dilimine dönüştürün
formatted_send_time = time.strftime("%Y-%m-%d %H:%M:%S (%Z)", send_time_tuple) # UTC zamanını istediğiniz şekilde biçimlendirin
print("Mesaj gönderildiği zaman:", formatted_send_time) # mesajın gönderildiği zamanı yazdırın
print("Cevap:", data.decode())

```

```python
# Server kodu

import socket
import time

# Soketi oluşturun ve UDP kullanarak bağlayın
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(("localhost", 12345))

# Gelen mesajları dinleyin
while True:
    data, addr = sock.recvfrom(1024)
    receive_time = time.time() # mesajın alındığı zamanı alın

    # Gelen mesajı ekrana yazdırın
    receive_time_tuple = time.gmtime(receive_time) # Unix zamanını UTC koordinatif dilimine dönüştürün
    formatted_receive_time = time.strftime("%Y-%m-%d %H:%M:%S (%Z)", receive_time_tuple) # UTC zamanını istediğiniz şekilde biçimlendirin
    print("Mesaj alındığı zaman:", formatted_receive_time) # mesajın alındığı zamanı yazdırın
    print("Mesaj:", data.decode())

    # Cevap oluşturun ve gönderin
    response = "Mesajınız alındı!"
    sock.sendto(response.encode(), addr)
```

Kod Çıktıları





4. Kitaplığınızı daha verimli ve özelliklerle dolu hale getirin. İlk olarak, çok büyük mesaj aktarımını ekleyin. Spesifik olarak, ağ maksimum mesaj boyutunu sınırlasa da, kitaplığınız keyfi olarak büyük boyutta 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 iletmelidir.

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 sonr:

```
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 bir parçayı birer birer göndermek yerine, çok sayıda parçayı hızlı bir şekilde göndermelisiniz, böylece ağın çok daha fazla kullanılmasına izin vermelisiniz. Bunu yapmak için, alıcı tarafındaki yeniden montajın 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 değikende 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);
```

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 uçuşta olabilmelidir. Ayrıca, bir istemcinin bekleyen tüm iletilerin onaylanmasını beklemesini sağlayan bir gönderen tarafı araması ekleyin.

CEVAP1
Asenkron mesaj gönderme için birkaç farklı yöntem vardır. Bu uygulamada, aşağıdaki yöntemlerden birini kullanabilirsiniz:
1)Sockets: TCP veya UDP sockets kullanarak asenkron mesaj göndermeyi gerçekleştirebilirsiniz. Bu yöntemde, istemci bir soket oluşturur ve sunucuya mesaj gönderir. Sunucu, mesajları alır ve güvenilir bir şekilde sırayla işler. Bu yöntem, mesajların güvenilir bir şekilde alınmasını sağlar ancak gönderici tarafından beklenen tüm iletilerin onaylanmasını bekleme özelliğini eklemek için ek bir yapı gerekebilir.
2)Message Queue: Mesaj kuyrukları, bir mesaj gönderen ve alıcının birbirlerine bağlı olmadığı asenkron mesaj gönderme için kullanılabilir. Bu yöntemde, istemci bir mesaj oluşturur ve sunucuya gönderir. Sunucu, mesajı alır ve güvenilir bir şekilde sırayla işler. Bu yöntem, mesajların güvenilir bir şekilde alınmasını ve gönderici tarafından beklenen tüm iletilerin onaylanmasını bekleme özelliğini de sağlar.
3)Web Services: Asenkron mesaj gönderme için web servisleri de kullanılabilir. Bu yöntemde, istemci bir web isteği gönderir ve sunucu, isteği işledikten sonra cevap verir. Bu yöntem, mesajların güvenilir bir şekilde alınmasını ve gönderici tarafından beklenen tüm iletilerin onaylanmasını bekleme özelliğini de sağlar ancak biraz daha karmaşık bir yapı gerektirir.

CEVAP2
Eşzamansız mesaj göndermeyi gerçekleştirmek için, aşağıdaki adımları takip edebilirsiniz:

İstemci ve sunucu arasında bir ağ bağlantısı oluşturun. Bu, Python dilinde "sockets" API'sini kullanarak yapılabilir.

İstemci, sunucuya mesaj göndermek için "send" işlevini kullanır. Bu işlev, bağlantı üzerinden veri gönderir ve mesajları eşzamansız olarak göndermek için bir iş parçacığı kullanılabilir.

Sunucu, mesajları almak ve işlemek için "recv" işlevini kullanır. Bu işlev, bağlantı üzerinden veri alır ve mesajları sırayla işlemek için bir iş parçacığı kullanılabilir.

Gönderen, mesajları doğru sırayla teslim etmek için her iletiye bir sıra numarası ekler ve alıcı bu sıra numarasını kullanarak mesajları doğru sırada yeniden birleştirir.

Alıcı, gönderici tarafından onaylanmayan mesajları yeniden iletmek için bir mekanizma uygular ve alıcı, her mesajı aldığını doğrulamak için onay mesajları gönderir.

Gönderici, onaylanacak tüm bekleyen mesajları beklemek için bir bekleme işlevi uygular ve alıcı, bekleyen tüm mesajların işlenmesini sağlamak için bir "flush" işlevi uygular.

Bu adımlar, eşzamansız mesaj göndermeyi gerçekleştirmeyi ve güvenilir bir şekilde teslim etmeyi sağ

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?

Bant genişliğini ölçmek için, bir makineye bir dizi veri gönderir ve bu verinin ne kadar sürede gönderildiğini ölçersiniz. Bu verinin boyutunu bölüp, gönderim süresine bölmeniz, verinin bir saniyedeki gönderim hızını verir. Bu değer, bant genişliğini ölçmek için kullanılabilir.

Gecikmeyi ölçmek için, bir makineden diğerine bir paket gönderirsiniz ve alıcı tarafından ne zaman alındığını ölçersiniz. Bu paketin gönderildiği zaman ile alındığı zaman arasındaki fark, paketin gecikme süresini verir.

Rakamların makul olup olmadığını belirlemek için, beklentilerinizi bilmeniz gerekir. Örneğin, bir ağın bant genişliği hakkında bir fikriniz olmalı ve gönderim hızının bu beklentinin altında olup olmadığını kontrol etmelisiniz. Aynı şekilde, gecikme beklentileriniz olmalı ve gecikme süresinin bu beklentinin altında olup olmadığını kontrol etmelisiniz. Bu beklentilerinizi belirlemek için, ağın fiziksel özelliklerini, cihazların özelliklerini ve ağ trafiğini dikkate almanız gerekir.

Eğer rakamlar beklentilerinizin altında ise, bir sorun olabilir ve bu sorunu çözmek için kodunuzu inceleyebilir ve ağın yapısını ve özelliklerini değerlendirebilirsiniz. Eğer rakamlar beklentilerinizin üstünde ise, kodunuzun iyi çalıştığını ve ağın iyi performans gösterdiğini söyleyebilirsiniz.