Dağıtılmış Sistemler

Dağıtılmış sistemler dünyanın çehresini değiştirdi. Web tarayıcınız gezegendeki başka bir yerdeki bir **web sunucusuna** (**client/server)** bağlandığında, basit bir istemci/sunucu dağıtılmış sistem biçimi gibi görünen bir şeye katılmaktadır. Bununla birlikte, Google veya Facebook gibi modern bir web hizmetiyle iletişime geçtiğinizde, yalnızca tek bir makine ile etkileşim kurmuyorsunuz; perde arkasında, bu karmaşık hizmetler, her biri sitenin belirli hizmetini sağlamak için işbirliği yapan geniş bir makine koleksiyonundan (yani binlerce) oluşturulur. Bu nedenle, dağıtılmış sistemleri incelemeyi ilginç kılan şeyin ne olduğu açık olmalıdır. Gerçekten de, bütün bir sınıfa yakışır; Burada, ana konulardan sadece birkaçını tanıtıyoruz. Dağıtılmış bir sistem oluştururken bir dizi yeni zorluk ortaya çıkar. Odaklandığımız en önemli şey **başarısızlıktır** (**failure)**; makineler, diskler, ağlar ve yazılımlar zaman zaman başarısız oluyor, çünkü "mükemmel" bileşenleri ve sistemleri nasıl oluşturacağımızı bilmiyoruz (ve muhtemelen asla bilemeyeceğiz). Bununla birlikte, modern bir web hizmeti oluşturduğumuzda, bunun müşterilere hiç başarısız olmuyormuş gibi görünmesini isteriz; 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 arızalansa da, nadiren arıza yapıyormuş 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 husustur. Uzak bir siteye bağlanırken, uzaktaki tarafın söyledikleri kişi olduğundan emin olmak temel 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. Yani, dağıtılmış bir sistemdeki makineler birbirleriyle nasıl **iletişim**(**communication)** kurmalıdır? Mevcut en temel ilkellerle, mesajlarla başlayacağız ve bunların üzerine birkaç üst düzey ilkel oluşturacağız. Yukarıda söylediğimiz gibi, başarısızlık merkezi bir odak olacaktır: iletişim katmanları başarısızlıkları nasıl ele almalı?

48.1 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.

```
// 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`)**

Yönlendiricinin bu noktada sahip olduğu tek seçenek, bir veya daha fazla paketi bırakmaktır. Aynı davranış uç ana bilgisayarlarda da oluşur; tek bir makineye çok sayıda mesaj gönderdiğinizde, makinenin kaynakları kolayca tükenebilir ve bu nedenle yeniden **paket kaybı (drop)** ortaya çıkar.

Bu nedenle, paket kaybı ağda esastır. Böylece soru şu hale gelir: Bununla nasıl başa çıkmalıyız?

48.2 Unreliable Communication Layers

Basit bir yol şudur: bununla ilgilenmiyoruz. Bazı uygulamalar paket kaybıyla nasıl başa çıkılacağını bildiğinden, bazen temel, güvenilir olmayan bir **mesajlaşma katmanıyla**(**end-to-end argument)** iletişim kurmalarına izin vermek yararlı olabilir). 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: **A Simple UDP Library (`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 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 husustur: 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 (User Datagram Protocol)** kullanmak için bir işlem, bir iletişim uç noktası oluşturmak amacıyla yuva API'sini kullanır; diğer makinelerdeki (veya aynı makinedeki) işlemler, orijinal işleme UDP datagramları 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 hiçbir arızaya karşı koruma sağlamadığı anlamına gelmez. Örneğin UDP, bazı paket bozulma biçimlerini algılamak için bir **sağlama**

toplamı (checksum) içerir. 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 kaybını ele alacak bazı yeni mekanizmalara ve tekniklere ihtiyacımız var. Basit düşünelim.

Figure 48.3: **Message Plus Acknowledgment**

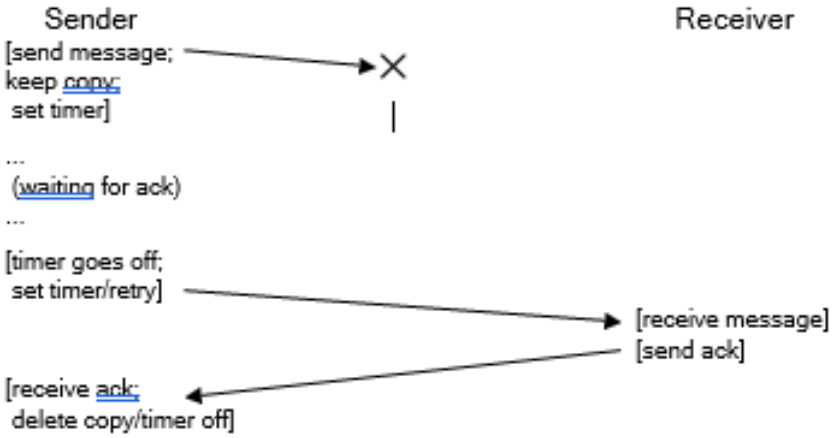

Figure 48.4: **Message Plus Acknowledgment: Dropped Request**

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 veya kısaca ack olarak bilinir. Fikir basit: gönderen, alıcıya bir mesaj gönderir; alıcı daha sonra alındığını onaylamak için kısa bir mesaj gönderir. Şekil 48.3, süreci göstermektedir.

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

Bu durumu halletmek için zaman aşımı 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(retry)**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 deneme**(**timeout/retry)** olarak adlandırmasına yol açtı; oldukça zeki kalabalık, bu ağ türleri, değil mi? Şekil 48.4 bir örneği göstermektedir.

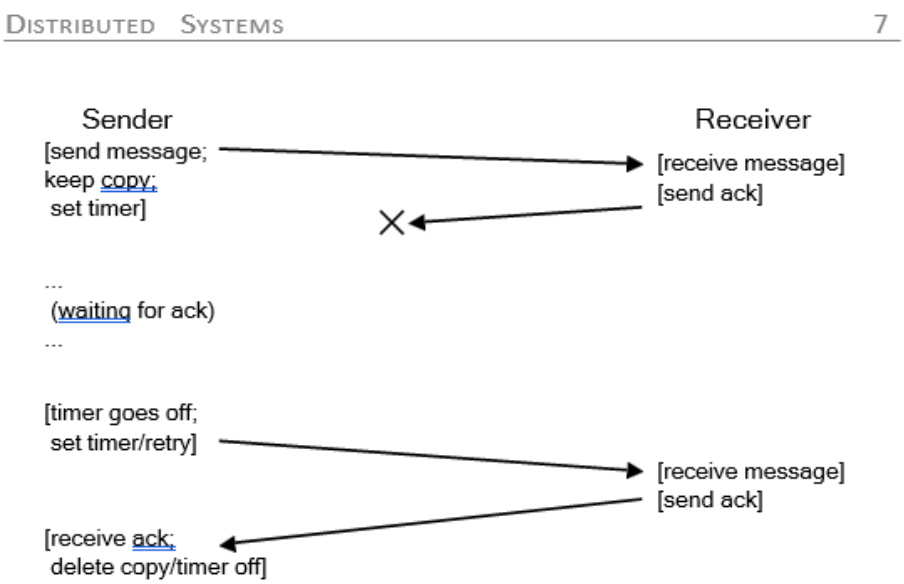

Figure 48.5: **Message Plus Acknowledgment: Dropped Reply**

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 her mesajın alıcı tarafından tam olarak **bir kez alındığını(exactly once)** da garanti etmek isteriz.

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. Bir mesaj gönderildiğinde, mesajın yanı sıra sayacın o anki 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, müşteri zaman aşımı değerini daha yüksek bir miktara, belki de orijinal değerin iki katına çıkarabilir.

Erken Aloha ağında öncülük edilen ve erken Ethernet'te [A70] benimsenen böyle bir üstel geri çekilme şeması, aşırı yeniden göndermeler nedeniyle kaynakların aşırı yüklendiği durumları önler. Sağlam sistemler, bu nitelikteki aşırı yükten kaçınmaya (**exponential back-off)** çalışır.

Alıcı, sayaç değerini, o göndericiden gelen mesajın kimliği için beklenen değer olarak kullanır. Alınan bir mesajın kimliği (N) alıcının sayacıyla (ayrıca N) eşleşirse, mesajı onaylar 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 kaybolursa, gönderen zaman aşımına uğrar ve N mesajını yeniden gönderir. Bu kez alıcının sayacı daha yüksektir (N + 1) ve bu nedenle alıcı bu mesajı zaten aldığını bilir. Böylece mesajı onaylar ama uygulamaya iletmez. Bu basit şekilde, tekrarlardan kaçınmak için dizi sayaçları kullanılabilir.

En sık kullanılan güvenilir iletişim katmanı, TCP/IP veya kısaca TCP olarak bilinir. TCP, ağdaki tıkanıklığı [VJ88], çok sayıda bekleyen isteği ve diğer yüzlerce küçük ince ayar ve optimizasyonu işlemek için makine dahil olmak üzere yukarıda tanımladığımızdan çok daha fazla karmaşıklığa sahiptir. Merak ediyorsanız bunun hakkında daha fazlasını okuyun; daha da iyisi, bir ağ oluşturma kursu alın ve bu materyali iyi öğrenin.

48.4 İletişim Soyutlamaları

Temel bir mesajlaşma katmanı verildiğinde, şimdi bu bölümdeki bir sonraki soruya yaklaşıyoruz: Dağıtılmış bir sistem oluştururken hangi iletişim soyutlamasını kullanmalıyız?

Sistem topluluğu, yıllar boyunca bir dizi yaklaşım geliştirdi. Bir çalışma grubu, işletim sistemi soyutlamalarını aldı ve bunları şu şekilde genişletti:

Dağıtılmış bir ortamda çalışır. Örneğin, **dağıtılmış paylaşılan bellek (distributed shared memory) (**DSM) sistemleri, farklı makinelerdeki işlemlerin büyük, sanal bir adres alanını paylaşmasını sağlar [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 sistemidir. Bir makinede bir sayfaya erişildiğinde iki şey olabilir. İlk (en iyi) durumda, sayfa zaten makinede yereldir ve bu nedenle veriler hızla getirilir. İkinci durumda, sayfa şu anda başka bir makinededir. Bir sayfa hatası oluşur ve sayfa hatası işleyicisi başka bir makineye sayfayı getirmesi, istekte bulunan işlemin sayfa tablosuna yüklemesi ve yürütmeye devam etmesi için bir mesaj gönderir.

Bu yaklaşım, birkaç nedenden dolayı günümüzde 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 bazı bölümleri aniden kullanılamaz hale gelir. Adres alanınızın bazı bölümleri kaybolduğunda başarısızlıkla başa çıkmak zordur; "sonraki" işaretçinin adres alanının kaybolan bir bölümünü işaret ettiği bağlantılı bir liste hayal edin. Eyvah!

Diğer bir sorun performanstır. Kod yazarken genellikle 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 zorundaydı ve bu tür bir yaklaşımın amacının çoğunu boşa çıkardı. Bu alanda çok araştırma yapılmasına rağmen, çok az pratik etkisi oldu; bugün hiç kimse DSM kullanarak güvenilir dağıtılmış sistemler oluşturamıyor.

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(re- mote procedure call)** veya kısaca RPC fikrine dayanmaktadır [BN84]. 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. Sihrin geri kalanı, genel olarak iki parçadan oluşan RPC sistemi tarafından gerçekleştirilir: bir **saplama oluşturucu (stub gen- erator)**(bazen **protokol derleyici(protocol compiler)** olarak adlandırılır) ve **çalışma kitaplığı(run-time library).** Şimdi bu parçaların her birine daha ayrıntılı olarak göz atacağız.

Saplama Oluşturucu

Saplama 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 oluşturulur; bu RPC hizmetini kullanmak isteyen bir istemci programı, RPC'ler yapmak için bu istemci stub'ına bağlanır ve onu çağırır.

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:

- Bir **mesaj tamponu(Create a message buffer.)** oluşturun. Bir mesaj arabelleği genellikle belirli bir boyuttaki bitişik bir bayt dizisidir.

- **Gerekli bilgileri mesaj arabelleğine paketleyin(Pack the needed information into the message buffer).**

Bu bilgi- çağrılacak işlev için bir tür tanımlayıcının yanı sıra işlevin ihtiyaç duyduğu tüm bağımsız değişkenleri içerir (örneğin, yukarıdaki örneğimizde, func1 için bir tamsayı). Tüm bu bilgileri tek bir bitişik arabelleğe koyma işlemine bazen argümanların **sıralanması(marshaling)** veya mesajın **seri hale(serialization)** getirilmesi denir.

- **Mesajı hedef RPC sunucusuna gönderin.(Send the message to the destination RPC server.)** RPC sunucusuyla iletişim ve sunucunun doğru çalışmasını sağlamak için gereken tüm ayrıntılar, aşağıda daha ayrıntılı olarak açıklanan RPC çalışma zamanı kitaplığı tarafından gerçekleştirilir.

- **Cevabı bekleyin(Wait for the reply).** İşlev çağrıları genellikle senkronize olduğundan, çağrı, tamamlanmasını bekleyecektir.

• **Dönüş kodunu ve diğer bağımsız değişkenleri paketinden çıkarın(Unpack return code and other arguments).** İşlev yalnızca tek bir dönüş kodu döndürürse, bu işlem basittir; ancak, daha karmaşık işlevler daha karmaşık sonuçlar (ör. bir liste) döndürebilir ve bu nedenle saplamanın bunları da açması gerekebilir. Bu adım aynı zamanda **xml'i objeye çevirme işlemi (unmarshaling**) veya seri hale getirme olarak da bilinir.

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

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

• **Mesajı paketinden çıkarın(Unpack the message)**. **Sıralamadan çıkarma(unmarshaling)** veya **seri durumdan(deserial-ization)** çıkarma adı verilen bu adım, gelen mesajdaki bilgileri alır. İşlev tanımlayıcısı ve bağımsız değişkenler çıkarılır.

• **Gerçek işlevi arayın(Call into the actual function).** Nihayet! noktaya ulaştık uzak işlevin gerçekte yürütüldüğü yer. RPC çalışma zamanı, kimlik tarafından belirtilen işlevi çağırır ve istenen bağımsız değişkenleri iletir.

• **Sonuçları paketleyin(Package the results)**. Dönüş bağımsız değişkenleri geri sıralanır tek bir yanıt arabelleğine.

• **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 tür eşzamanlı moda oluşturulur.Yaygın bir organizasyon bir **iş parçacığı havuzudur(thread pool).** 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 aşmaktadır. 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 kitabındaki mükemmel bölümü okuyun[SK09]. 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(end-to-end argument), 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. Ardından, bir süre geçtikten sonra istemci, sunucunun istek üzerinde hala 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 [BN84].

Birçok sistemin ele aldığı sorunlardan biri **bayt sıralamasıdır(byte ordering).** Bildiğiniz gibi, bazı makineler değerleri büyük endian sıralaması olarak bilinen şekilde depolarken, diğerleri küçük 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, mesajdaki her bilgi parçasının dönüştürülmesi gerekir. Böylece, endianness farkı küçük bir performans maliyetine sahip olabilir.

Son bir konu, iletişimin eşzamansız doğasını istemcilere gösterip göstermemek, böylece bazı performans iyileştirmelerini mümkün kılmaktır. Spesifik olarak, tipik RPC'ler eşzamanlı olarak yapılır, yani bir müşteri prosedür çağrısını yayınladığında, devam etmeden önce prosedür çağrısının geri dönmesini beklemesi gerekir. 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; 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 somebasic 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 thefew 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 (Kod)

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

SORULAR

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.

Server

```python
import socket

# Create a UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind the socket to the port
server_address = ('localhost', 10000)
sock.bind(server_address)

while True:
    # Receive message from client
    data, address = sock.recvfrom(4096)

    # Send an acknowledgement to the client
    sock.sendto(b'ACK', address)
```

Client

```python
import socket

# Create a UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Send a message to the server
server_address = ('localhost', 10000)
message = b'Hello, server!'
sock.sendto(message, server_address)

# Receive the acknowledgement from the server
data, _ = sock.recvfrom(4096)
print(data)
```

Bunu iki farklı makinede çalıştırmak için, istemcide sunucu adresi olarak bir makinenin IP adresini ve sunucuda sunucu adresi olarak diğer makinenin IP adresini kullanmanız gerekecektir.

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.

UDP sunucusunu ve istemci kodunu bir iletişim kitaplığına dönüştürmek için, mesaj gönderme ve alma yöntemlerine sahip bir sınıf tanımlayabiliriz.

```python
import socket

class UDPCommunication:
    def __init__(self, address, port):
        self.address = address
        self.port = port
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    def send(self, message):
        self.sock.sendto(message, (self.address, self.port))

    def receive(self):
        data, _ = self.sock.recvfrom(4096)
        return data
```

Bu kitaplığı kullanmak için istemci ve sunucu kodu, ham soket çağrıları yerine UDPCommunication sınıfının gönderme ve alma yöntemlerini kullanacak şekilde yeniden yazılabilir.

Server

```python
import UDPCommunication

# Create a communication object and bind it to the port
server = UDPCommunication('localhost', 10000)

while True:
    # Receive message from client
    data = server.receive()

    # Send an acknowledgement to the client
    server.send(b'ACK')
```

Client

```python
import UDPCommunication

# Create a communication object
client = UDPCommunication('localhost', 10000)

# Send a message to the server
message = b'Hello, server!'
client.send(message)

# Receive the acknowledgement from the server
data = client.receive()
print(data)
```

Bu kitaplık, zaman aşımlarını ayarlama veya hataları işleme yöntemleri gibi gerektiğinde ek yöntemlerle genişletilebilir.

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, UDP ile gönderebileceğiniz en büyük tek mesaj olmalıdır. 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!

UDP iletişim kitaplığına zaman aşımı ve yeniden deneme işlevi eklemek için, yeniden denemelerle mesaj göndermek için bir yöntem ve alınan mesajları onaylamak için bir yöntem ekleyebiliriz. Gönderme yöntemi, gönderilmiş ancak henüz onaylanmamış mesajları takip etmeli ve belirli bir zaman aşımı süresi içinde onaylanmamış mesajları göndermeyi yeniden denemelidir. Alma yöntemi, alınan tüm mesajları onaylamalıdır.

```python
import socket
import time

class UDPCommunication:
    def __init__(self, address, port, timeout=1.0):
        self.address = address
        self.port = port
        self.timeout = timeout
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.unacknowledged_messages = {}

    def send(self, message):
        # Send the message and start a timer
        self.sock.sendto(message, (self.address, self.port))
        self.unacknowledged_messages[message] = time.time()

        # Wait for an acknowledgement or timeout
        while message in self.unacknowledged_messages:
            time.sleep(0.01)  # Sleep to avoid spinning and wasting CPU

            # Check if the message has timed out
            elapsed_time = time.time() - self.unacknowledged_messages[message]
            if elapsed_time > self.timeout:
                # Resend the message and reset the timer
                self.sock.sendto(message, (self.address, self.port))
                self.unacknowledged_messages[message] = time.time()

    def receive(self):
        data, _ = self.sock.recvfrom(4096)

        # Acknowledge the received message
        self.sock.sendto(b'ACK', (self.address, self.port))
        return data

    def acknowledge(self, message):
        # Remove the message from the unacknowledged messages list
        if message in self.unacknowledged_messages:
            del self.unacknowledged_messages[message]
```

İstemci ve sunucu kodu, bu yeni yöntemleri kullanmak için yeniden yazılabilir. İstemci, bir mesaj göndermek için gönderme yöntemini çağırmalı ve bir onay alana kadar engellemeli ve sunucu bir mesaj almak için alma yöntemini çağırmalı ve ardından onaylamak için onaylama yöntemini çağırmalıdır.

Server

```
import UDPCommunication

# Create a communication object and bind it to the port
server = UDPCommunication('localhost', 10000)

while True:
    # Receive message from client
    data = server.receive()

    # Acknowledge the received message
    server.acknowledge(data)
```

Client

```
import UDPCommunication

# Create a communication object
client = UDPCommunication('localhost', 10000)

# Send a message to the server and wait for an acknowledgement
message = b'Hello, server!'
client.send(message)
```

Bu uygulama, onaylanmayan mesajları ve bunlara karşılık gelen zaman damgalarını depolamak için bir sözlük kullanır ve zaman aşımı süresi içinde onaylanmayan mesajları göndermeyi yeniden dener.

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.

UDP iletişim kitaplığına büyük mesajların aktarılması için destek eklemek için, büyük mesajları daha küçük parçalara bölerek ve tek tek göndererek göndermek için bir yöntem ekleyebiliriz. Sunucu tarafında, kitaplık parçaları orijinal büyük mesajda bir araya getirebilir.

```
import socket
import time

class UDPCommunication:
    def __init__(self, address, port, timeout=1.0, fragment_size=4096):
        self.address = address
        self.port = port
        self.timeout = timeout
        self.fragment_size = fragment_size
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.unacknowledged_messages = {}
        self.received_fragments = {}

    def send(self, message):
        # Divide the message into fragments
        message_id = hash(message)  # Generate a unique ID for the message
        fragments = [message[i:i+self.fragment_size] for i in range(0,
len(message), self.fragment_size)]

        # Send each fragment and start a timer
        for i, fragment in enumerate(fragments):
            self.sock.sendto(fragment, (self.address, self.port))
            self.unacknowledged_messages[(message_id, i)] = time.time()

        # Wait for all fragments to be acknowledged or time out
        while self.unacknowledged_messages:
            time.sleep(0.01)  # Sleep to avoid spinning and wasting CPU

            # Check if any fragments have timed out
            for key in list(self.unacknowledged_messages.keys()):
                elapsed_time = time.time() - self.unacknowledged_messages[key]
                if elapsed_time > self.timeout:
                    # Resend the fragment and reset the timer
                    message_id, fragment_index = key
                    fragment = fragments[fragment_index]
                    self.sock.sendto(fragment, (self.address, self.port))
                    self.unacknowledged_messages[key] = time.time()
```

```
    def receive(self):
        data, _ = self.sock.recvfrom(self.fragment_size)

        # Extract the message ID and fragment index from the received data
        message_id, fragment_index, fragment = data[:8], data[8:12], data[12:]

        # Acknowledge the received fragment
        self.sock.sendto(b'ACK', (self.address, self.port))

        # Store the received fragment
        if message_id not in self.received_fragments:
            self.received_fragments[message_id] = {}
        self.received_fragments[message_id][fragment_index] = fragment

        # Check if all fragments have been received
        if len(self.received_fragments[message_id]) == len(fragments):
            # Assemble the fragments into the original message
            message = b''.join(self.received_fragments[message_id][i] for i in
range(len(fragments
```

5. Yukarıdakileri yüksek performansla tekrar yapın. 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.

UDP iletişim kitaplığının performansını artırmak için, birden çok parçayı sendto'ya tek bir çağrıda gönderebilir ve alıcı tarafında doğru şekilde yeniden birleştirildiğinden emin olmak için her parçayı bir sıra numarasıyla işaretleyebiliriz

```
import socket
import time

class UDPCommunication:
    def __init__(self, address, port, timeout=1.0, fragment_size=4096):
        self.address = address
        self.port = port
        self.timeout = timeout
        self.fragment_size = fragment_size
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.unacknowledged_messages = {}
        self.received_fragments = {}

    def send(self, message):
        # Divide the message into fragments
        message_id = hash(message)  # Generate a unique ID for the message
        fragments = [message[i:i+self.fragment_size] for i in range(0,
len(message), self.fragment_size)]

        # Send the fragments in groups
        for i in range(0, len(fragments), self.fragment_size):
            # Encode the message ID and sequence numbers into the data
            data = message_id.to_bytes(8, 'big')
            for j, fragment in enumerate(fragments[i:i+self.fragment_size]):
                data += j.to_bytes(4, 'big')
                data += fragment

            # Send the data and start a timer
            self.sock.sendto(data, (self.address, self.port))
            self.unacknowledged_messages[message_id] = time.time()

        # Wait for all fragments to be acknowledged or time out
        while message_id in self.unacknowledged_messages:
            time.sleep(0.01)  # Sleep to avoid spinning and wasting CPU

            # Check if the message has timed out
            elapsed_time = time.time() -
self.unacknowledged_messages[message_id]
            if elapsed_time > self.timeout:
                # Resend the fragments and reset the timer
                self.sock.sendto(data, (self.address, self.port))
                self.unacknowledged_messages[message_id] = time.time()

    def receive(self):
        data, _ = self.sock.recvfrom(self.fragment_size)

        # Extract the message ID and sequence numbers from the received data
        message_id = int.from_bytes(data[:8], 'big')
        fragment_indexes = [int.from_bytes(data[i:i+4], 'big') for i in range(8,
len(data), self.fragment_size+4)]
        fragments = [data[i:i+self.fragment_size] for i in range(12, len(data),
self.fragment_size+4)]

        # Acknowledge the received fragments
        self.sock.sendto(b'ACK', (self.address, self.port))

        # Store the received fragments
        if message_id
```

6. Son bir uygulama zorluğu: sıralı teslimat ile eşzamansız mesaj gönderme. Diğer bir deyişle, müşteri birbiri ardına bir mesaj göndermek için gönder'i tekrar tekrar çağırabilmelidir; alıcı alıcıyı aramalı ve her mesajı güvenilir bir şekilde sırayla almalıdır; gelen birçok mesaj gönderici aynı anda uçuşta olabilmelidir. Ayrıca, bir müşterinin bekleyen tüm mesajların onaylanmasını beklemesini sağlayan bir gönderen tarafı araması ekleyin.

Bir yaklaşım, gönderen iş parçacığının bir mesaj kuyruğuna mesaj göndermesi ve alıcı iş parçacığının aynı sıradan gelen mesajları almasıyla gönderen ve alıcı için ayrı bir iş parçacığına sahip olmak olacaktır. İleti kuyruğu, iletilerin gönderildikleri sırayla teslim edilmesini sağlar.

Gönderenin onaylanacak tüm bekleyen mesajları beklemesini sağlamak için, alındılar için ayrı bir mesaj kuyruğu kullanabilir ve devam etmeden önce gönderen iş parçacığının uygun sayıda onayın alınmasını beklemesini sağlayabilirsiniz.

```python
import threading
import queue

# Sender thread
def sender_thread(send_queue, ack_queue):
  # Send a message
  send_queue.put(("message 1", 1))

  # Wait for acknowledgment
  ack = ack_queue.get()
  assert ack == 1, "Unexpected acknowledgment"

  # Send another message
  send_queue.put(("message 2", 2))

  # Wait for another acknowledgment
  ack = ack_queue.get()
  assert ack == 2, "Unexpected acknowledgment"

# Receiver thread
def receiver_thread(send_queue, ack_queue):
  # Receive first message
  message, seq_num = send_queue.get()
  assert message == "message 1", "Unexpected message"

  # Send acknowledgment
  ack_queue.put(seq_num)

  # Receive second message
  message, seq_num = send_queue.get()
  assert message == "message 2", "Unexpected message"

  # Send acknowledgment
  ack_queue.put(seq_num)

# Create message queues
send_queue = queue.Queue()
ack_queue = queue.Queue()
```

```python
# Create and start sender and receiver threads
sender = threading.Thread(target=sender_thread, args=(send_queue, ack_queue))
receiver = threading.Thread(target=receiver_thread, args=(send_queue,
ack_queue))
sender.start()
receiver.start()

# Wait for threads to finish
sender.join()
receiver.join()
```

Bu kod iki iş parçacığı oluşturur: gönderen iş parçacığı ve alıcı iş parçacığı. Gönderici iş parçacığı, mesajları send_queue'ye

gönderir ve ack_queue'deki onayları beklerken, alıcı iş parçacığı send_queue'den mesajları alır ve onayları ack_queue'ye gönderir. İleti kuyrukları, iletilerin gönderildikleri sırayla teslim edilmesini sağlar ve onaylar, gönderenin iletilerin ne zaman alındığını bilmesini sağlar.

7. Şimdi bir acı nokta daha: ölçüm. Yaklaşımlarınızın her birinin bant genişliğini ölçün; iki farklı makine arasında ne kadar veriyi ne oranda aktarabilirsiniz? Gecikmeyi de ölçün: tek paket gönderimi ve alındısı 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 ayarlayabilirsiniz?

Yaklaşımlarınızın bant genişliğini ölçmek için iperf veya netcat gibi bir araç kullanabilirsiniz. Bu araçlar, iki makine arasında veri göndermenizi ve aktarılan veri miktarını ve aktarım hızını ölçmenizi sağlar.

Örneğin, iki makine arasındaki bant genişliğini ölçmek için iperf'i şu şekilde kullanabilirsiniz:

Sunucu makinesinde:

```
iperf -s
```

Sunucu makinesinde:

```
iperf -c <server_ip_address>
```

Bu, iki makine arasında bir bant genişliği testi başlatacak ve iperf, aktarılan veri miktarını ve aktarım hızını bildirecektir.

Gecikmeyi ölçmek için ping veya traceroute gibi bir araç kullanabilirsiniz. Bu araçlar, bir paketi başka bir makineye göndermenize ve paketin gönderilip alınması için gereken gidiş-dönüş süresini (RTT) ölçmenize olanak tanır.

Örneğin, iki makine arasındaki gecikmeyi ölçmek için ping'i şu şekilde kullanabilirsiniz:

```
ping <remote_ip_address>
```

Bu, uzak makineye bir dizi paket gönderecek ve her paket için RTT'yi bildirecektir.

Yaklaşımınızın bant genişliğini ve gecikme süresini ölçtüğünüzde, sonuçları beklentilerinizle karşılaştırabilirsiniz. Rakamlar makul görünüyorsa, kodunuz iyi çalışıyor olabilir. Rakamlar beklentilerinizden önemli ölçüde farklıysa bu, kodunuzla veya ağla ilgili bir soruna işaret ediyor olabilir.

Beklentilerinizi daha iyi belirlemek için, kullandığınız ağ için beklenen bant genişliğini ve gecikmeyi araştırabilirsiniz. Örneğin, yerel alan ağı (LAN) kullanıyorsanız, geniş alan ağı (WAN) kullandığınız duruma göre daha yüksek bant genişliği ve daha düşük gecikme bekleyebilirsiniz. Ağ bağlantısının türü (ör. kablolu veya kablosuz), makineler arasındaki mesafe ve ağda bant genişliği için rekabet eden diğer cihazların sayısı gibi faktörleri de göz önünde bulundurabilirsiniz.