48

Dağıtık Sistemler

Dağıtılmış sistemler dünyanın çehresini değiştirdi . Web tarayıcınız gezegende başka bir yerde bir web sunucusuna bağlandığında, **istemci / sunucu(client/server)** dağıtılmış sisteminin basit bir biçimi gibi görünen şeye katılıyor. Google veya Face-book gibi modern bir web servisiyle iletişime geçtiğinizde, yalnızca tek bir makineyle etkileşime girmiyorsunuz, ancak; sahnelerin arkasında, se karmaşık hizmetler büyük bir alandan inşa edilir. Her biri sitenin kısmi hizmetini sağlamak için işbirliği yapan makinelerin toplanması (yani binlerce). 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 çıkıyor Odaklandığımız en önemli şey **başarısızlık(failure)**; makineler, diskler, ağlar ve yazılımların hepsi zaman zaman başarısız olur, çünkü biz bunu yapmayız (ve muhtemelen asla başarısız olmaz) "Mükemmel" bileşenlerin ve sistemlerin nasıl oluşturulacağını bilir. Ancak, modern bir web hizmeti oluşturduğumuzda, müşterilere asla başarısız olmazmış gibi görünmesini isteriz; Bu görevi nasıl başarabiliriz?

DÖNÜM:
BİLEŞENLER BAŞARISIZ OLDUĞUNDA ÇALIŞAN SİSTEMLER NASIL OLUŞTURULUR Her zaman düzgün çalışmayan parçalardan nasıl bir çalışma sistemi kurabiliriz? Temel soru size RAID depolama dizilerinde tartıştığımız bazı konuları hatırlatmalıdır; Ancak buradaki sorunlar, çözümler gibi daha karmaşık olma eğilimindedir.

İlginçtir ki, başarısızlık dağıtılmış sistemlerin inşasında merkezi bir zorluk olsa da, aynı zamanda bir fırsatı temsil eder. Evet, makineler arızalanır; ancak bir makinenin arızalanması gerçeği, tüm sistemin arızalanması gerektiği anlamına gelmez. Bir dizi makineyi bir araya getirerek, bileşenlerinin düzenli olarak arızalanmasına rağmen, nadiren arızalanan bir sistem oluşturabiliriz . Bu gerçeklik, dağıtılan sistemlerin merkezi güzelliği ve değeridir ve neden Google, Facebook vb. dahil olmak üzere kullandığınız hemen hemen her modern web hizmetinin altında yatmaktadır.

İPUCU: İLETİŞİM DOĞASI GEREĞİ GÜVENİLMEZDİR
Neredeyse her koşulda, iletişimi temelde güvenilmez bir faaliyet olarak görmek iyidir. Bit bozulması, çalışmayan veya çalışmayan bağlantılar ve makineler ve gelen paketler için arabellek alanı olmaması aynı sonuca yol açar: paketler bazen hedeflerine ulaşmaz. Bu tür güvenilmez ağların üzerine güvenilir hizmetler oluşturmak için paket kaybıyla başa çıkabilecek teknikleri göz önünde bulundurmalıyız.

Başka önemli konular da var. Sistem **performansı(performance)** genellikle kritiktir; Dağıtılmış sistemimizi birbirine bağlayan bir ağ ile, sistem tasarımcıları genellikle verilen görevleri nasıl yerine getireceklerini, gönderilen mesaj sayısını azaltmaya ve iletişimi daha verimli hale getirmeye çalıştıklarını dikkatlice düşünmelidir. (düşük gecikme süresi, yüksek bant genişliği) mümkün olduğunca.

Son olarak, **güvenlik(security)** de gerekli bir husustur. Uzak bir siteye bağlanırken, uzak partide kim olduklarını söylediklerine dair bir güvenceye sahip olmak merkezi bir sorun haline gelir. Ayrıca, üçüncü tarafların diğer ikisi arasında devam eden bir iletişimi izleyememesini veya değiştirememesini sağlamak da bir zorluktur.

Bu girişte, dağıtılmış bir sistemde yeni olan en temel yönü ele alacağız: **iletişim(communication)**. Yani, dağıtılmış bir sistemdeki makineler birbirleriyle nasıl iletişim kurmalıdır? Mevcut en temel ilkellerle , mesajlarla başlayacağız ve bunların üzerine birkaç üst düzey ilkel inşa edeceğiz. Yukarıda söylediğimiz gibi, başarısızlık merkezi bir odak noktası olacaktır: iletişim katmanları başarısızlıkları nasıl ele almalıdır?

48.1 İletişimle İlgili Temel Bilgiler

Modern ağın temel ilkesi, iletişimin eğlenceli ve günlük olarak güvenilmez olmasıdır. İster geniş alan Internet'te ister Infiniband gibi yerel bir yüksek hızlı ağda olsun, paketler düzenli olarak kaybolur, parçalanır veya başka bir şekilde hedeflerine ulaşamaz .

Paket kaybı veya bozulması için çok sayıda neden vardır. Bazen, iletim sırasında, bazı uçlar elektrik veya diğer benzer sorunlar nedeniyle ters çevrilir. Bazen, sistemdeki bir ağ iş bağlantısı veya paket yönlendirici veya hatta uzak ana bilgisayar gibi bir öğe, bir şekilde zarar görür veya başka bir şekilde düzgün çalışmaz ; ağ kabloları yanlışlıkla kopar, en azından bazen.

Bununla birlikte, daha temel olanı, bir ağ anahtarı, yönlendirici veya uç nokta içinde arabelleğe alma eksikliğinden kaynaklanan paket kaybıdır. Özellikle, tüm bağlantıların doğru çalıştığını ve sistemdeki tüm bileşenlerin (anahtarlar, yönlendiriciler, uç ana bilgisayarlar) beklendiği gibi çalıştığını garanti edebilsek bile, aşağıdaki nedenlerden dolayı kayıp hala mümkündür. Bir paketin bir yönlendiriciye ulaştığını hayal edin; paketin işlenmesi için, yönlendiricinin içinde bir yerde belleğe yerleştirilmesi gerekir . Bu tür birçok paket ulaşırsa

```
// İStemci kodu
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;
}

// Sunucu kodu
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;
}
```

Şekil 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, paketlerden bir veya daha fazlasını bırakmaktır . Aynı davranış uç ana bilgisayarlarda da oluşur ; tek bir makineye çok sayıda mesaj gönderdiğinizde , makinenin kaynakları kolayca bozulabilirve böylece paket kaybı tekrar ortaya çıkar.

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

48.2 Güvenilmez İletişim Katmanları

Basit bir yol şudur: bununla uğraşmıyoruz. Bazı uygulamalar paket kaybıyla nasıl başa çıkacaklarını bildiklerinden, bazen temel bir güvenilmez mesajlaşma katmanıyla, sıklıkla duyulan **uçtan uca argümanın** (**end-to-end argument**)bir örneği ile iletişim kurmalarına izin vermek yararlıdır (bölümün sonundaki Kenara bakın). 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);
}
```

Şekil 48.2: **Basit UDP Kitaplığı (udp.c)**

İPUCU: BÜTÜNLÜK İÇİN SAĞLAMA TOPLAMLARINI KULLANIN

Sağlama toplamları, modern sistemlerde bozulmayı hızlı ve etkili bir şekilde tespit etmek için yaygın olarak kullanılan bir yöntemdir. Basit bir sağlama toplamı eklemedir: yalnızca bir veri yığınının baytlarını toplayın; Elbette, temel döngüsel artıklık kodları (CRC'LER), Fletcher sağlama toplamı ve diğerleri dahil olmak üzere daha karmaşık sağlama toplamları oluşturulmuştur [MK09].

Ağda sağlama toplamları aşağıdaki gibi kullanılır. Bir makineden diğerine mesaj göndermeden önce, mesajın baytları üzerinden bir sağlama toplamı hesaplayın. Sonra hem mesajı hem de sağlama toplamını desti- nation'a gönderin. Hedefte, alıcı gelen iletinin üzerinde de bir sağlama toplamı hesaplar; Hesaplanan bu sağlama toplamı gönderilen sağlama toplamıyla eşleşirse, alıcı, iletim sırasında verilerin büyük olasılıkla bozulmadığına dair bir güvence hissedebilir.

Sağlama toplamları bir dizi farklı eksen boyunca değerlendirilebilir. Etkililik birincil hususlardan biridir: verilerdeki bir değişiklik, verilerde bir değişikliğe yol açar mı?toplamlarının hesaplanmasının genellikle pahalı olduğu anlamına gelir . Hayat, yine, mükemmel değil.

UDP/IP ağ yığınında bugün neredeyse modern sistemlerde mevcuttur. UDP kullanmak için, bir işlem bir **iletişim uç noktası(communication endpoint)** oluşturmak üzere **soketler(sockets)** API'sini kullanır; Diğer makinelerdeki (veya aynı makinedeki) işlemler, UDP **datagramlarını (datagrams)**orijinal işleme gönderir (datagram, maksimum boyuta kadar sabit boyutlu bir mesajdır).

Şekil 48.1 ve 48.2, UDP / IP üzerine kurulu basit bir istemci ve sunucuyu göstermektedir. İstemci sunucuya bir ileti gönderebilir ve sunucu da bir yanıtla yanıt verir. Bu az miktarda kodla, dağıtılmış systems oluşturmaya başlamak için ihtiyacınız olan her şeye sahipsiniz!

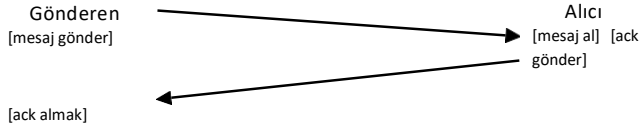
UDP, güvenilmez bir iletişim katmanının harika bir örneğidir. Bunu kullanırsanız, paketlerin kaybolduğu (düştüğü) ve dolayısıyla hedeflerine ulaşmadığı durumlarla karşılaşırsınız ; gönderen bu nedenle kayıptan asla haberdar edilmez. Ancak bu, UDP'nin herhangi bir arızaya karşı koruma sağlamadığı anlamına gelmez. Örneğin, UDP bazı paket bozulma biçimlerini algılamak için bir **sağlama toplamı(checksum)** içerir.

Ancak, birçok uygulama sadece bir hedefe veri göndermek istediğinden ve paket kaybı konusunda endişelenmediğinden, daha fazlasına ihtiyacımız var. Özellikle, güvenilmez bir ağın üstünde güvenilir iletişime ihtiyacımız var.

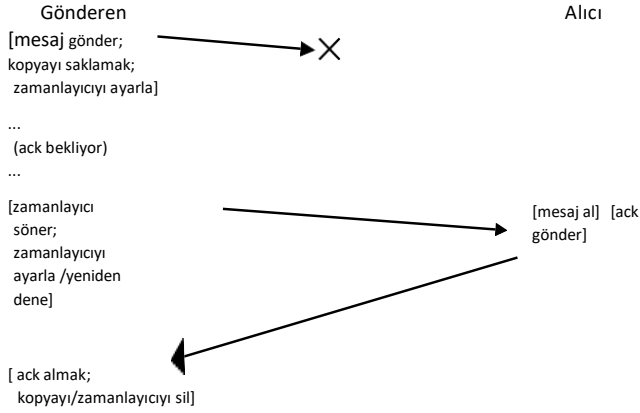
48.3 Güvenilir İletişim Katmanları

Güvenilir bir iletişim katmanı oluşturmak için, paket kaybını ele almak için bazı yeni mekanik anizmlere ve tekniklere ihtiyacımız var. Basit

bir şey düşünelim



Şekil 48.3: **Message Plus Teşekkür**



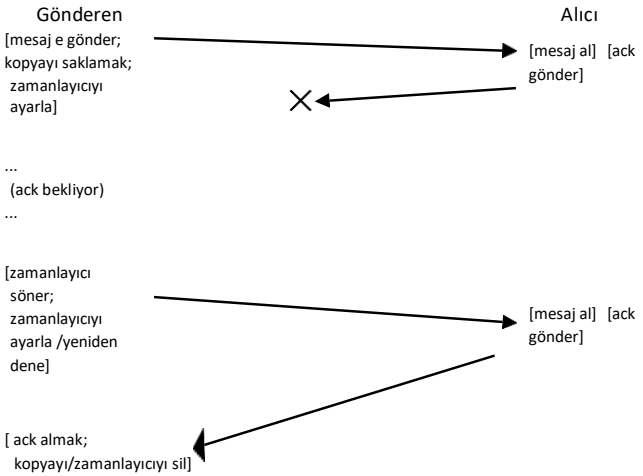
Şekil 48.4: **Message Plus Onay: Bırakılan İstek**

Bir istemcinin güvenilmez bir bağlantı üzerinden bir sunucuya ileti gönderdiği örnek . Cevaplamamız gereken ilk soru: Gönderen, alıcının mesajı gerçekten aldığını nasıl biliyor?

Kullanacağımız teknik, bir **teşekkür**(**acknowledgment**) veya kısaca **ack** olarak bilinir. Fikir basittir: 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 gerçekten orijinal bilgeyi aldığından emin olabilir. Ancak, gönderen bir onay almazsa ne yapmalıdır?

Bu durumu ele almak için, **zaman aşımı(timeout)** olarak bilinen ek bir mekanizmaya ihtiyacımız var. Gönderen bir ileti gönderdiğinde, gönderen artık bir süre sonra kapanacak bir zamanlayıcı ayarlar. Bu süre zarfında herhangi bir onay alınmamışsa, gönderen iletinin kaybolduğu sonucuna varır . gönerici daha sonra göndermenin **yeniden denemesini(retry)** yapar ve bu sefer geçeceği umuduyla aynı mesajı tekrar gönderir. Bu yaklaşımın işe yaraması için, gönderenin tekrar göndermesi gerekebileceği ihtimaline karşı iletinin bir kopyasını yanında tutması gerekir. Zaman aşımı ve yeniden denemenin birleşimi, bazılarının yaklaşımı **zaman aşımı/yeniden deneme(timeout/retry)** olarak adlandırmasına neden olmuştur; oldukça zeki kalabalık, bu ağ türleri, değil mi? Şekil 48.4'te bir örnek gösterilmektedir.



Şekil 48.5: **Message Plus Teşekkür: Bırakılan Yanıt**

Ne yazık ki, bu formda zaman aşımı / yeniden deneme yeterli değildir . Şekil 48.5, soruna yol açabilecek bir paket kaybı örneği göstermektedir. Bu örnekte, kaybolan orijinal mesaj değil, onaydır. Gönderenin bakış açısından , durum aynı görünüyor: hiçbir ack alınmadı ve bu nedenle bir zaman aşımı ve yeniden deneme sırası geldi. Ancak alıcının bakış açısından, oldukça farklı: şimdi aynı mesaj iki kez alındı! Bunun iyi olduğu durumlar olsa da, genel olarak değildir; Bir dosyayı indirirken ne olacağını ve indirme işleminin içinde fazladan paketlerin tekrarlandığını hayal edin. Güvenilir bir mesaj katmanı hedeflediğimizde, genellikle her mesajın alıcı tarafından **tam olarak bir kez(exactly once)** alındığını garanti etmek isteriz.

Alıcının yinelenen mesaj iletimini algılamasını sağlamak için , gönderenin her mesajı benzersiz bir şekilde tanımlaması gerekir ve alıcının her mesajı önceden görüp görmediğini izlemek için bir yola ihtiyacı vardır. Alıcı yinelenen bir iletim gördüğünde, mesajı basitçe ack , ancak (kritik olarak) mesajı verileri alan uygulamaya *iletmez* . Böylece, gönderen ack'yi alır , ancak mesaj yukarıda belirtilen tam olarak bir kez semantikleri koruyarak iki kez alınmaz.

Yinelenen iletileri algılamanın sayısız yolu vardır. Örneğin, gönderen could her ileti için benzersiz bir kimlik oluşturur; alıcı şimdiye kadar gördüğü her kimliği izleyebilir. Bu yaklaşım işe yarayabilir, ancak tüm kimlikleri izlemek için sınırsız bellek gerektiren son derece maliyetlidir.

Çok az bellek gerektiren daha basit bir yaklaşım bu sorunu çözer ve mekanizma bir **dizi sayacı(sequence counter)** olarak bilinir. Bir sıra sayacı ile, gönderen ve alıcı, her iki tarafın da koruyacağı bir sayaç için bir başlangıç değeri (örneğin, 1) üzerinde anlaşır. Bir mesaj gönderildiğinde, ülkenin geçerli değeri mesajla birlikte gönderilir; Bu sayaç değeri (*N*), mesaj için bir kimlik görevi görür. İleti gönderildikten sonra, gönderen daha sonra 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 çalışır.Sağlam sistemler bu nitelikteki aşırı yüklenmeyi önlemek için çaba gösterir.

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 sayacını (*N* + 1'e) arttırır 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, yinelemeleri önlemek için sıra sayaçları kullanılabilir .

En yaygın kullanılan güvenilir iletişim katmanı TCP **/ IP** veya kısaca **TCP** olarak bilinir. TCP, ağdaki tıkanıklığı ele almak için makineler [VJ88], birden fazla olağanüstü istek ve yüzlerce diğer küçük ince ayar ve optimizasyon dahil olmak üzere yukarıda açıkladığımızdan çok daha karmaşıklığa sahiptir. 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 paylaşmasını sağlar [LH89]. Bu soyutlama, dağıtılmış bir hesaplamayı çok iş parçacıklı bir uygulamaya benzeyen bir şeye dönüştürür; 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 approach, birkaç nedenden dolayı bugün 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 kısımları kaybolduğunda başarısızlıkla başa çıkmak zordur; "Sonraki" bir işaretçinin adres alanının giden bir kısmına girdiği 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, böyle bir yaklaşımın amacının çoğunu yenerek, neredeyse hiç iletişim gerçekleşmeyecek şekilde organize etmek için çok dikkatli olmak zorundaydılar. 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 soyutlamaları, dağıtılmış sistemler oluşturmak için zayıf bir seçim olduğu ortaya çıkarken , programlama dili (PL) soyutlamaları çok daha mantıklıdır. En baskın soyutlama, **uzaktan prosedür çağrısı (re- mote procedure call)** veya kısaca **RPC** [BN84][1] fikrine dayanmaktadır.

Uzaktan yordam çağrısı paketlerinin hepsinin basit bir amacı vardır: uzak bir makinede kod yürütme işlemini yerel bir işlevi çağırmak kadar basit ve basit hale getirmek. Böylece, bir müşteriye bir prosedür çağrısı yapılır ve bir süre sonra sonuçlar döndürülür. Sunucu sadece dışa aktarmak istediği bazı rutinleri tanımlar. Sihrin geri kalanı, genel olarak iki parçadan oluşan RPC sistemi tarafından ele alınır: bir **saplama oluşturucu(stub generator)** (bazen **protokol derleyicisi(protocol compiler)** olarak adlandırılır) ve **çalışma zamanı kütüphanesi(run-time library)**. Şimdi bu parçaların her birine daha ayrıntılı olarak bakacağız.

[1] Modern programlama dillerinde, bunun yerine **uzaktan yöntem çağırma** diyebiliriz. (**RMI**), ama bu dilleri, tüm süslü nesneleriyle kim sever ki ?

Saplama Jeneratörü

Saplama üretecinin işi basittir: paketleme işlevi argümanlarının ve sonuçlarının bazı acılarını otomatik hale getirerek mesajlara dönüştürmek. Çok sayıda fayda ortaya çıkar: Kişi, tasarım gereği, bu tür bir kodu elle yazarken meydana gelen basit hatalardan kaçınır; dahası, bir saplama derleyicisi belki de bu tür kodları optimize edebilir ve böylece performansı artırabilir.

Böyle bir derleyiciye giriş, bir sunucunun istemcilere vermek istediği çağrılar kümesidir. Kavramsal olarak, bu kadar basit bir şey olabilir :

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

Saplama üreteci böyle bir arayüz alır ve birkaç farklı kod parçası oluşturur. İstemci için, arabirimde belirtilen işlevlerin her birini içeren bir **istemci saplaması(client stub)** oluşturulur ; Bu RPC hizmetini kullanmak isteyen bir istemci programı, bu istemci saplaması ile bağlantı kurar ve RPC'leri yapmak için 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, istemci func1(x) çağırır); dahili olarak, func1() için istemci saplamasındaki kod şunu yapar:

- **İleti arabelleği oluşturun(Create a message buffer).** İleti arabelleği genellikle yalnızca belirli bir boyuttaki bitişik bir bayt dizisidir.
- **Gerekli bilgileri ileti arabelleğine paketleyin(Pack the needed information into the message buffer).** Bu information, 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 işlemi bazen argümanların **sıralanması(marshaling)** veya mesajın **serileştirilmesi(serialization)** olarak adlandırılır.
- **İletiyi hedef RPC sunucusuna gönderin(Send the message to the destination RPC server).** RPC sunucusuyla iletişim ve düzgün çalışması için gereken tüm ayrıntılar, aşağıda daha ayrıntılı olarak açıklanan RPC çalışma zamanı kitaplığı tarafından işlenir.
- **Yanıtı bekleyin(Wait for the reply).** İşlev çağrıları genellikle **eşzamanlı(synchronous)**olduğundan, çağrı tamamlanmasını bekleyecektir.
- **Dönüş kodunu ve diğer bağımsız değişkenleri açın(Unpack return code and other arguments).** İşlev yalnızca tek bir dönüş kodunu yeniden döndürürse, bu işlem basittir; ancak, daha karmaşık işlevler daha karmaşık sonuçlar (örneğin, bir liste) döndürebilir ve bu nedenle saplamanın bunları da açması gerekebilir. Bu adım, **sırasız bırakma(unmarshaling)** veya **seri** durumdan çıkarma(**deserialization**) olarak da bilinir.
- **Arayana geri dönün(Return to the caller).** Son olarak, sadece müşteri saplamasından geri dönün istemci koduna yerleştirin .

Sunucu için kod da oluşturulur. Sunucu üzerinde atılan adımlar aşağıdaki gibidir:

- **İletinin paketini açın(Unpack the message). Sırayı kaldırma(unmarshaling)** veya **seri(deserialization)** durumdan çıkarma olarak adlandırılan bu adım, gelen iletideki bilgileri alır. İşlev tanımlayıcısı ve bağımsız değişkenler ayıklanır.
- **Gerçek işlevi çağırın(Call into the actual function).** Nihayet! Noktaya geldik uzak işlevin gerçekten yürütüldüğü yer. RPC çalışma zamanı , kimlik tarafından belirtilen işlevi çağırır ve istenen bağımsız değişkenleri iletir.
- **Sonuçları paketleyin(Package the results).** Geri dönüş argüman(lar)ı geri sıralanır
  tek bir yanıt arabelleğine.
- **Yanıtı gönderin(Send the reply).** Yanıt 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. 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 kütüphanesi, bir RPC sistemindeki ağır yüklerin çoğunu üstlenir; çoğu performans ve güvenilirlik sorunu burada ele alınmaktadır. Şimdi böyle bir çalışma zamanı katmanı oluşturmadaki bazı büyük zorlukları tartışacağız.

Üstesinden gelmemiz gereken ilk zorluklardan biri, bir re-mote hizmetinin nasıl bulunacağıdır. Bu **adlandırma(naming)** sorunu , dağıtılmış sistemlerde yaygın bir sorundur ve bir anlamda mevcut tartışmamızın kapsamının ötesine geçer. En basit yaklaşımlar, mevcut internet protokolleri tarafından sağlanan ana bilgisayar adları ve bağlantı noktası numaraları gibi mevcut adlandırma sistemlerine dayanır. Böyle bir sistemde, istemci, istenen RPC hizmetini çalıştıran makinenin ana bilgisayar adını veya IP adresini ve kullandığı bağlantı noktası numarasını bilmelidir (bir bağlantı noktası numarası, Bir makinede gerçekleşen ve aynı anda birden fazla iletişim kanalına izin veren belirli iletişim faaliyetleri). Protokol paketi daha sonra paketleri sistemdeki diğer herhangi bir makineden belirli bir adrese yönlendirmek için bir mekanizma sağlamalıdır. Adlandırma hakkında iyi bir tartışma için, başka bir yere bakmanız gerekir, örneğin, İnternet'te DNS ve ad çözünürlüğü hakkında bilgi edinin ya da daha iyisi , Saltzer ve Kaashoek'in kitabındaki mükemmel bölümü okuyun. [SK09].

Bir istemci,belirli bir re-mote hizmeti için hangi h sunucusuyla konuşması gerektiğini öğrendikten sonra, bir sonraki soru, RPC'nin hangi aktarım düzeyi protokolünün üzerine inşa edilmesi gerektiğidir. Özellikle, RPC sistemi TCP / IP gibi güvenilir bir pro-tocol kullanmalı mı yoksa UDP / IP gibi güvenilmez bir iletişim katmanı üzerine mi kurulmalıdır?

Naif bir şekilde seçim yapmak kolay görünüyor: Açıkçası bir yeniden görevin uzak sunucuya güvenilir bir şekilde teslim edilmesini istiyoruz ve açıkça güvenilir bir şekilde yanıt almak istiyoruz. Bu nedenle TCP gibi güvenilir transport 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: onaybelirlemeleri artı zaman aşımı / yeniden deneme ile. Böylece, istemci sunucuya bir RPC isteği gönderdiğinde, sunucunun bir onayla yanıt verdiği, böylece arayanın isteğin alındığını bildiği belirtilir. Benzer şekilde, sunucu yanıtı istemciye gönderdiğinde, istemci yanıtı ack, böylece sunucu yanıtın alındığını bilir. Güvenilir bir iletişim katmanının üzerine bir istek/yanıt protokolü (RPC gibi) oluşturularak , iki "ekstra" mesaj gönderilir.

Bu nedenle, birçok RPC paketi, UDP gibi katmanlardaki güvenilmez com-municatiüzerine inşa edilmiştir . Bunu yapmak daha verimli bir RPC katmanı sağlar, ancak RPC sistemine güvenilirlik sağlama sorumluluğunu da ekler. RPC katmanı, zaman aşımı / yeniden deneme yaparak istenen sorumluluk seviyesine ulaşır ve yukarıda tarif ettiğimiz çok şeyi kabul eder. İletişim katmanı, bir tür sıra numaralandırma kullanarak, her RPC'nin tam olarak bir kez (arıza olmaması durumunda) veya en fazla bir kez (hatanın ortaya çıktığı durumda) gerçekleşmesini garanti edebilir.

Diğer Sorunlar

Bir RPC çalışma zamanının da ele alması gereken başka sorunlar da vardır. Örneğin, uzaktan aramanın tamamlanması uzun zaman

aldığında ne olur? Zaman aşımı makinemiz göz önüne alındığında, uzun süredir devam eden bir uzaktan arama, bir müşteriye başarısızlık olarak görünebilir, böylece yeniden denemeyi tetikleyebilir ve bu nedenle burada biraz bakım yapılması gerekebilir. Çözümlerden biri , 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.

(alıcı vegönderenden) yanıt hemen oluşturulmadığında; bu, istemcinin sunucunun isteği aldığını bilmesini sağlar. Daha sonra, bir süre geçtikten sonra, istemci periyodik olarak sunucunun hala istek üzerinde çalışıp çalışmadığını sorabilir; Sunucu g "evet" demeye devam ederse, istemci mutlu olmalı ve beklemeye devam etmelidir (sonuçta, bazen bir prosedür çağrısının yürütülmesinin tamamlanması uzun zaman alabilir).

Çalışma zamanı, yordam çağrılarını tek bir pakete sığabilecek olandan daha büyük bağımsız değişkenlerle de işlemelidir. Bazı alt düzey ağ protokolleri, bu tür gönderen tarafı **parçalanması(fragmentation)** (daha büyük paketlerin daha küçük paketler kümesine) ve alıcı tarafı **yeniden montajını(reassembly)** (daha küçük parçaların daha büyük bir mantıksal bütüne dönüştürülmesi) sağlar ; Aksi takdirde, RPC çalışma zamanının such işlevselliğinin kendisini uygulaması gerekebilir. Ayrıntılar için Birrell ve Nelson'ın makalesine bakınız [BN84].

Birçok sistemin ele aldığı bir sorun, **bayt sıralamasıdır(byte ordering)**. Bildiğiniz gibi , bazı makineler **değerleri big(big endian)** endian sıralaması olarak bilinen şeyde depolarken, diğerleri **küçük endian(little endian)** sıralamayı kullanır. Big endian, Arap rakamları gibi, en önemli bitlerden en az önemli bitlere kadar baytları (örneğin, bir tamsayı) depolar ; little endian tam tersini yapar. Her ikisi de nümerik formasyonda depolamanın eşit derecede geçerli yollarıdır; buradaki soru, *farklı* endianlığa sahip makineler arasında nasıl iletişim kurulacağıdır.

RPC paketleri genellikle ileti biçimlerinde iyi tanımlanmış bir endi-anness sağlayarak bunu ele alır. Sun'ın RPC paketinde, **XDR** (**sonsuz Veri Gösterimi**)(**eXternal Data Representation**) katmanı bu işlevselliği sağlar. Bir mesaj gönderen veya alan makine XDR'nin endianness'iyle eşleşiyorsa, mesajlar beklendiği gibi gönderilir ve alınır. Bununla birlikte, iletişim kuran makinenin farklı bir endianlığı varsa, mesajdaki her bilgi parçası dönüştürülmelidir. Bu nedenle, endyanlılıktaki farkın küçük bir performans maliyeti olabilir.

Son bir konu, iletişim iletişiminin zaman uyumsuz doğasını istemcilere gösterip göstermeyeceği, böylece bazı performans optimizasyonlarının sağlanıp sağlanmayacağıdır. Özellikle, tipik RPC'ler **eşzamanlı olarak(synchronously)** yapılır, yani bir istemci yordam çağrısını verdiğinde, devam etmeden önce yordam çağrısının dönmesini beklemesi gerekir. Bu bekleme süresi uzun olabileceğinden ve istemcinin yapabileceği başka işleri olabileceğinden, bazı RPC paketleri zaman **uyumsuz olarak(asynchronously)** RPC çağırmanıza olanak tanır. Zaman uyumsuz bir RPC verildiğinde, RPC paketi isteği gönderir ve hemen geri döner; İstemci daha sonra diğer RPC'leri çağırmak veya diğer yararlı hesaplamalar gibi başka işler yapmakta özgürdür . İstemci bir noktada zaman uyumsuz RPC'nin sonuçlarını görmek isteyecektir; Bu nedenle, RPC katmanına geri çağırır ve bekleyen RPC'lerin tamamlanmasını beklemesini söyler, bu noktada dönüş bağımsız değişkenlerine erişilebilir.

48.6 Özet

Yeni bir konunun, dağıtılmış sistemlerin ve ana sorununun tanıtımını gördük: şimdi sıradan bir olay olan arızanın nasıl ele alınacağı. Google'ın içinde dedikleri gibi, sadece masaüstü makineniz olduğunda, başarısızlık nadirdir; binlerce makineye sahip bir veri merkezindeyken , arıza her zaman oluyor. Herhangi bir dağıtılmış sistemin anahtarı, bu başarısızlıkla nasıl başa çıkacağınızdır.

İletişimin, dağıtılan herhangi bir sistemin kalbini oluşturduğunu da gördük . Bu iletişimin yaygın bir soyutlaması, istemcilerin sunucularda uzaktan arama yapmalarını sağlayan uzaktan yordam çağrısında (RPC) found'dur; RPC paketi , teslim etmek için zaman aşımı/yeniden deneme ve onaylama dahil olmak üzere tüm kanlı ayrıntıları işler. yerel yordam çağrısını yakından yansıtan bir hizmet.

Bir RPC paketini gerçekten anlamanın en iyi yolu elbette bir tanesini kendiniz kullanmaktır. Sun'ın RPC sistemi, saplama derleyicisi `rpcgen` 'i kullanarak, daha eski bir sistemdir; Google'ın gRPC'si ve Apache Thrift'i aynı şeyi üstlenen modern yaklaşımlardır. Birini deneyin ve tüm yaygaranın neyle ilgili olduğunu görün.

References

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

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

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

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

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

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

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

Ödev (Kod)

Bu bölümde, bunu yapma görevini öğrenmenizi sağlamak için bazı basit iletişim kodları yazacağız . İyi eğlenceler!

Soru

1. Bölümde sağlanan kodu kullanarak, basit bir UDP tabanlı sunucu ve istemci oluşturun. Sunucu istemciden ileti almalı ve bir onayla yanıt vermelidir. Bu ilk denemede, herhangi bir yeniden iletim veya sağlamlık eklemeyin (iletişimin mükemmel çalıştığını varsayalım). Bunu test için tek bir makinede çalıştırın; Daha sonra iki farklı makinede çalıştırın .
2. Kodunuzu bir **iletişim kitaplığına(communication library.)** dönüştürün. Özellikle,
kendi API'niz, çağrı gönderip alın ve gerektiğinde diğer API çağrılarıyla. İstemcinizi ve sunucunuzu, ham soket çağrıları yerine kitaplığınızı kullanacak şekilde yeniden yazın.
3. Gelişen iletişim li- brary'nize, **zaman aşımı / yeniden deneme(timeout/retry)** şeklinde güvenilir iletişim ekleyin . Özellikle, kitaplığınız göndereceği herhangi bir iletinin bir kopyasını oluşturmalıdır. Gönderirken, bir zamanlayıcı başlatmalıdır , böylece mesajın gönderilmesinden bu yana ne kadar zaman geçtiğini izleyebilir . Alıcıda, kütüphane alınan mesajları kabul etmelidir. İstemci gönderme sırasında **engellemeli**, yani geri dönmeden önce mesaj onaylanana kadar beklemelidir. Ayrıca, istemeden göndermeyi yeniden denemeye istekli olmalıdır. Maksimum ileti boyutu, UDP ile gönderebileceğiniz en büyük tek iletinin boyutu olmalıdır. Son olarak, bir ack gelene veya iletim zaman aşımına uğrayana kadar arayanı uyku moduna geçirerek zaman aşımı/yeniden deneme işlemini verimli bir şekilde gerçekleştirdiğinizden emin olun ; CPU'yu *döndürmeyin* ve boşa harcamayın!
4. Kitaplığınızı daha verimli ve özelliklerle dolu hale getirin. İlk olarak, çok büyük mesaj aktarımı ekleyin. Özellikle, ağ maksimum imum ileti boyutunu sınırlasa da, kütüphaneniz keyfi olarak büyük boyutlu bir mesaj almalı ve istemciden sunucuya aktarmalıdır. İstemci bu büyük mesajları parçalar halinde sunucuya iletmelidir; sunucu tarafı kütüphane kodu, alınan parçaları bitişik bütün halinde birleştirmeli ve tek bir büyük arabelleği bekleyen sunucu koduna geçirmelidir.
5. Yukarıdakileri tekrar yapın, ancak yüksek performansla. Her parçayı birer birer göndermek yerine, hızlı bir şekilde birçok parça göndermelisiniz, böylece ağın çok daha fazla kullanılmasına izin vermelisiniz. Bunu yapmak için, aktarımın her bir parçasını

dikkatlice işaretleyin, böylece alıcı tarafındaki yeniden montaj mesajı karıştırmaz .

6. Son bir uygulama zorluğu: sıralı teslimatla zaman uyumsuz ileti gönderme. Yani, istemci birbiri ardına mesaj göndermek için tekrar tekrar gönder çağırabilmelidir ; alıcı re-ceive'ı aramalı ve her mesajı sırayla, güvenilir bir şekilde almalıdır; birçok mesaj gönderen aynı anda uçuşta olabilmelidir. Ayrıca, bir istemcinin tüm bekleyen mes-sage'lerin onaylanmasını beklemesini sağlayan bir gönderen tarafı çağrısı ekleyin .

7. Şimdi, bir acı noktası daha: ölçüm. Yaklaşımlarınızın her birinin bant genişliğini ölçün; iki farklı makinede ne kadar veri aktarabilirsiniz, hangi oranda? Ayrıca gecikmeyi de ölçün: tek paket gönderme ve onaylama için, ne kadar çabuk biter? Son olarak, sayılarınız makul görünüyor mu? Ne bekliyordunuz? Bir sorun olup olmadığını veya kodunuzun iyi çalışıp çalışmadığını bilmek için beklentilerinizi nasıl daha iyi ayarlayabilirsiniz?