50

분산 시스템

분산 시스템은 전세계의 구조를 바꿨다. 웹 브라우저가 지구 상 어딘가에 있는 웹 서버에 접속하면 **클라이언트/서버** 분산 시스템이라는 구조에 한 구성원이 된다. 구글이나 페이스북의 웹 서비스를 사용한다는 것은 하나의 기계를 사용하는 것이 아니다. 수천 대의 이루어진 기계들이 사이트의 특정 서비스를 제공하기 위해서 서로 협력하고 있다. 분산 시스템을 공부하는 것을 흥미롭게 만드는 것이 무엇인지 명확하게 알았을 것이다. 사실 이 주제만을 위한 독립된 수업이 있어야 한다. 여기에서는 그 중의 몇 가지 중요 주제들을 다루도록 한다.

분산 시스템을 개발할 때 몇 가지 새로운 도전거리가 생겨난다. 그 중 우리가 집중할 부분은 "**실패**"에 관한 것이다. 우리가 "완벽"한 시스템을 만드는 법을 모르기 때문에 (어쩌면 평생 모를지도) 기계와 디스크, 네트워크와 소프트웨어는 모두 때때로 고장이 난다. 하지만 웹 서비스를 개발할 때, 그 서비스는 사용자에게 절대 중지하지 않는 것처럼 보이기를 원한다. 어떻게 이 목적을 달성할 수 있을까?

핵심 질문: 구성 요소가 실패하더라도 동작하는 시스템을 어떻게 만들까

가끔씩 고장나는 부품들로 어떻게 항상 동작하는 시스템을 만들까? RAID에서 다루었던 주제들이 기억날 것이다. 하지만 여기서 다루는 문제는 좀 더 복잡하다. 해법마저도 복잡하다.

분산 시스템의 핵심 사안은 실패와 고장의 극복이다. 새로운 주제를 연구할 수 있는 기회가 생겼다. 기계는 고장난다. 하지만 기계 중 하나가 고장났다고 해서 전체 시스템이 정지한다는 것을 뜻하지는 않는다. 비록 개별 구성 요소들은 자주 고장나지만 기계들을 고장 없는 시스템처럼 보이도록 만들 수가 있다. 이것이 진정한 아름다움이자 분산 시스템의 가치이다. 구글과 페이스북 등을 포함하여 거의 모든 현대의 웹 서비스는 이러한 토대 위에서 동작한다.

또 다른 중요한 문제가 있다. 시스템 **성능**은 매우 중요한 요소이다. 분산된 시스템들을 연결하는 네트워크에서는 시스템 설계자들은 주어진 목적을 달성하는 데 있어 많은 신경을 써야 한다. 전송 메시지의 개수를 줄이고, 통신이 가능한 효율적이도록(낮은 지연 시간, 높은 대역폭) 만들어야 한다.

팁: 통신은 본질적으로 신뢰할 수 없다

거의 모든 상황에서 통신은 근본적으로 비신뢰성 작업으로 보는 것이 좋다. 비트 손상, 끊어진 링크나 멈춘 기계들 그리고 도착한 패킷들을 수용하는 버퍼의 고갈과 같은 것들은 모두 같은 결과에 도달한다. 패킷들이 목적지에 도달하지 못한다. 신뢰성 있는 서비스를 신뢰할 수 없는 네트워크 상에서 구현하기 위해서는 패킷 손실에 대응할 수 있는 기술들을 개발해야 한다.

마지막으로, **보안** 역시 매우 중요한 요소이다. 원격 사이트를 접속할 때, 접속한 사이트가 진짜 원했던 사이트인지를 확신할 수 있는지도 중요한 문제이다. 더 나아가 양자 간의 통신을 제 삼자가 변경하거나 도청할 수 없도록 만드는 것도 어려운 문제다.

여기서는 분산 시스템에서 새로이 등장하는 개념인 통신에 대해서 다루도록 하겠다. 질문은 다음과 같다. 분산 시스템에서 어떻게 한 기계가 다른 기계와 **통신**할 수 있을까? 먼저 메시지라고 하는 가장 기본적인 기법을 살펴보고 그것을 기반으로 하여 복합적인 기법들을 다루도록 하겠다. 앞에서 언급했듯이 실패가 우리의 주 관심사가 될 것이다. 통신 계층이 실패를 다루는 방법을 살펴보자.

50.1 통신의 기본

최신 네트워킹의 핵심 가정은 통신은 신뢰할 수 없다는 것이다. 광역 인터넷이던 인피니밴드(Infiniband)와 같은 근거리 고속 네트워크이던 상관없이 패킷들은 정기적으로 손실되거나, 손상되거나, 목적지에 도착하지 못할 수도 있다.

패킷 손실이나 손상에는 많은 이유가 있다. 때로는 전송 중에 전기적으로 또는 그와 유사한 문제로 비트가 반전된다. 어떤 경우에 네트워크의 링크나 패킷 라우터와 같은 시스템의 구성 요소 또는 원격 호스트 등이 고장 났거나 제대로 동작을 안할 수도 있다. 드물게는 네트워크 연결선이 사고로 잘렸을 수도 있다.

좀 더 중요한 원인은 네트워크 스위치, 라우터 또는 연결의 종단점에서 충분히 버퍼링을 할 수 없기 때문이다. 구체적으로, 모든 링크가 제대로 동작하고 모든 시스템의 구성 요소가(스위치와 라우터 그리고 종단의 호스트) 제대로 동작 중이라 하더라도 다음과 같은 이유로 패킷들을 잃어버릴 수 있다. 라우터에 패킷이 도착한다고 해 보자. 패킷이 처리되려면 라우터 내부 어딘가에 존재하는 메모리에 저장되어야 한다. 한 번에 많은 패킷들이 도착한다면 라우터의 메모리가 그 모든 패킷들을 다 수용할 수 없을 수 있다. 그 시점에 라우터가 내릴 수 있는 선택은 패킷을 포기하는(**drop**) 것이다. 이와 같은 현상은 호스트에서도 마찬가지로 발생한다. 많은 수의 메시지를 하나의 기계에 전송하면 그 기계의 자원은 쉽게 고갈될 수 있기 때문에 패킷들을 같은 식으로 잃어버리게 된다.

패킷 손실은 네트워킹에서 근본적인 문제이다. 우리의 질문은 이렇다. 어떻게 대처해야 할까?

50.2 신뢰할 수 없는 통신 계층

간단한 방법은 아무런 조치도 취하지 않는 것이다. 어떤 응용 프로그램들은 패킷 손실 시 대응 방법을 가지고 있기 때문에 메시지 계층과 직접 통신하도록 하는 것이 이로울 때도 있다. 신뢰할 수 없는 계층에 대한 좋은 예 중 하나는 거의 모든 현대 시스템에 존재하는 **UDP/IP** 네트워크 스택을 들 수 있다. UDP를 사용하기 위해서는 **소켓** API를 이용하여 **통신 지점**(**Communication end point**)을 생성한다. 다른 편 기계(또는 같은 기계 내)의 프로세스들은 UDP **데이터그램**(**datagram**)을 원래의 프로세스로 전송한다(데이터그램은 최대 크기가 정해져 있는 고정 크기의 메시지이다).

그림 50.1과 그림 50.2는 UDP/IP로 구현되어 있는 간단한 클라이언트와 서버를 나타낸다. 클라이언트는 서버로 메시지를 보낼 수 있으며 서버는 받은 메시지에 답신으로 응답한다. 이 적은 양의 코드에 분산 시스템을 구현하는 데 필요한 모든 것이 담겨 있다!

UDP는 신뢰할 수 없는 통신 계층의 훌륭한 예제이다. 만약 사용한다면 패킷을 잃어버리는(떨어진) 경우들을 만나게 될 것이며 메시지는 목적지에 도달하지 못한다. 발신 측은 그렇기 때문에 손실에 대해서 전혀 알 수가 없다. 하지만, UDP가 모든 실패에 대해서 전혀 대비를 할 수 없다는 말은 아니다. 예를 들어 UDP는 **체크섬**을 포함하고 있기 때문에 일부 패킷 손상은 검출할 수 있다.

하지만 많은 응용 프로그램들은 그저 목적지로 데이터를 전송하기를 원할 뿐 패킷 손실에 대해서 걱정하고 싶어 하지 않기 때문에 우리는 더 많은 기법들을 준비해야 한다.

```
1  // 클라이언트 코드
2  int main(int argc, char *argv[]) {
3      int sd = UDP_Open(20000);
4      struct sockaddr_in addr, addr2;
5      int rc = UDP_FillSockAddr(&addr, "machine.cs.wisc.edu", 10000);
6      char message[BUFFER_SIZE];
7      sprintf(message, "hello world");
8      rc = UDP_Write(sd, &addr, message, BUFFER_SIZE);
9      if (rc > 0) {
10         int rc = UDP_Read(sd, &addr2, buffer, BUFFER_SIZE);
11     }
12     return 0;
13 }
14
15 // 서버 코드
16 int main(int argc, char *argv[]) {
17     int sd = UDP_Open(10000);
18     assert(sd > -1);
19     while (1) {
20         struct sockaddr_in s;
21         char buffer[BUFFER_SIZE];
22         int rc = UDP_Read(sd, &s, buffer, BUFFER_SIZE);
23         if (rc > 0) {
24             char reply[BUFFER_SIZE];
25             sprintf(reply, "reply");
26             rc = UDP_Write(sd, &s, reply, BUFFER_SIZE);
27         }
28     }
29     return 0;
30 }
```

〈그림 50.1〉 **UDP/IP 클라이언트/서버 코드**

```
1  int UDP_Open(int port) {
2      int sd;
3      if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) { return -1; }
4          struct sockaddr_in myaddr;
5          bzero(&myaddr, sizeof(myaddr));
6          myaddr.sin_family = AF_INET;
7          myaddr.sin_port = htons(port);
8          myaddr.sin_addr.s_addr = INADDR_ANY;
9          if (bind(sd, (struct sockaddr *) &myaddr, sizeof(myaddr)) == -1) {
10             close(sd);
11             return -1;
12         }
13     return sd;
14 }
15
16 int UDP_FillSockAddr(struct sockaddr_in *addr, char *hostName, int port) {
17     bzero(addr, sizeof(struct sockaddr_in));
18     addr->sin_family = AF_INET; // 호스트 바이트 오더
19     addr->sin_port = htons(port); // short, 네트워크 바이트 오더
20     struct in_addr *inAddr;
21     struct hostent *hostEntry;
22     if ((hostEntry = gethostbyname(hostName)) == NULL) { return -1; }
23         inAddr = (struct in_addr *) hostEntry->h_addr;
24         addr->sin_addr = *inAddr;
25     return 0;
26 }
27
28 int UDP_Write(int sd, struct sockaddr_in *addr, char *buffer, int n) {
29     int addrLen = sizeof(struct sockaddr_in);
30     return sendto(sd, buffer, n, 0, (struct sockaddr *) addr, addrLen);
31 }
32
33 int UDP_Read(int sd, struct sockaddr_in *addr, char *buffer, int n) {
34     int len = sizeof(struct sockaddr_in);
35     return recvfrom(sd, buffer, n, 0, (struct sockaddr *) addr,
36                     (socklen_t *) &len);
37     return rc;
38 }
```

〈그림 50.2〉 **간단한 UDP 라이브러리**

구체적으로 신뢰할 수 없는 네트워크 상에서 신뢰할 수 있는 통신 방법이 필요하다.

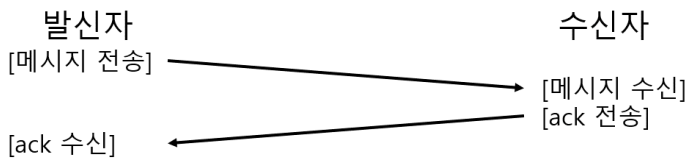
50.3 신뢰할 수 있는 통신 계층

신뢰할 수 있는 통신 계층을 만들기 위해서는 패킷 손실에 대응할 수 있는 새로운 메커니즘과 기술이 필요하다. 클라이언트가 불안한 연결을 통해 서버로 메시지를 전송하는 간단한 예를 살펴보자. 첫 번째 질문은 이렇다. 발신자는 수신자가 메시지를 수신했다는 것을 어떻게 알 수 있을까?

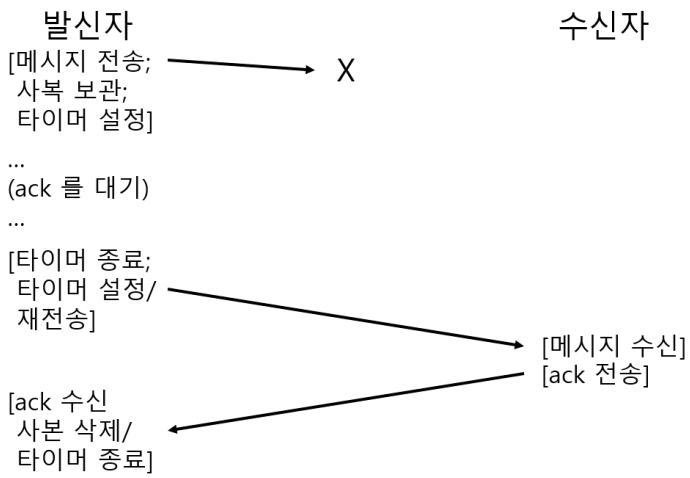
우리가 사용할 기술은 **확인**(**acknowledgement**) 또는 짧게 **ack**라고 하는 것이다. 개념은 간단하다. 발신자는 메시지를 수신자에게 보낸다. 수신자는 받았다는 것을 알리기 위해서 짧은 메시지를 다시 보낸다. 그림 50.3에 이 과정을 나타내고 있다.

발신자가 메시지가 도착했다는 ack를 받으면 수신자가 메시지를 잘 받았다는 것을 확신할 수 있다. 발신자가 ack를 못 받으면 어떻게 할까?

이와 같은 경우를 다루기 위해서 **타임아웃**(**timeout**)이라고 하는 추가적인 장치가 필요하다. 발신자가 메시지를 보낼 때 발신자는 타이머를 설정하여 일정 시간이 흐른



〈그림 50.3〉 **메시지와 확인(ack)**

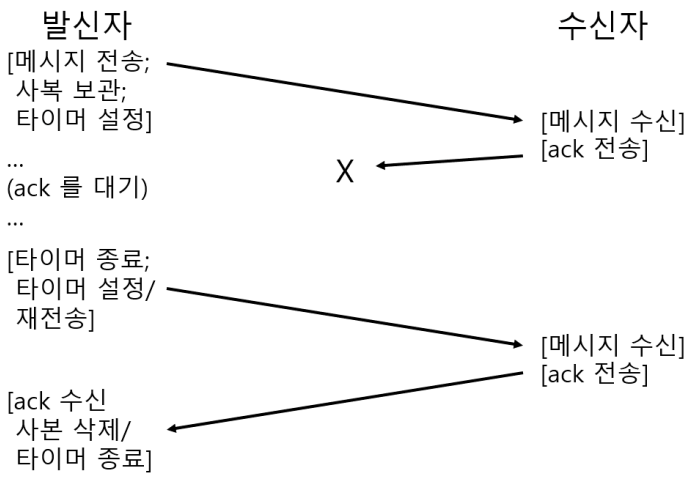


〈그림 50.4〉 **메시지와 확인(ack): 누락된 요청**

후에는 종료되도록 한다. 만약 그 시간 안에 ack를 받지 못한다면, 발신자는 메시지를 잃어버렸다고 판단한다. 발신자는 이번엔 전달되겠지 하는 희망을 갖고 똑같은 메시지의 전송을 재시도한다. 이것이 제대로 동작하려면 발신자는 **재전송**에 대비하여 메시지의 사본을 갖고 있어야 한다. 타임아웃과 재시도라는 기술이 조합되었기 때문에 일부는 이 방법을 **타임아웃/재시도** 방식이라고 부른다. 네트워크하는 사람들이 꽤 똑똑한 것 같지 않은가? 그림 50.4에서 이 방식을 보인다.

불행하게도, 현재 상태의 타임아웃/재시도로는 충분하지가 않다. 그림 50.5에서 패킷 손실이 문제를 만들어 내는 경우의 예를 들었다. 이 예제에서는 원래의 메시지가 손실되는 것이 아니라 ack 메시지가 손실되었다. 발신자 측에서 본다면 ack를 못 받은 상황과 마찬가지이기 때문에 타임아웃과 재시도 방식이 제대로 동작하는 것처럼 보인다. 하지만 수신자 측에서 보면 상당히 다르다. 이 경우 같은 메시지를 두 번 받았다! 어떤 경우에는 이런 상황이 발생하는 것이 괜찮을지는 모르겠지만, 일반적으로 그렇지가 않다. 파일을 내려받는 중에 어떤 패킷이 반복적으로 받아졌다면 어떻게 되겠는지 생각해 보라. 그러므로 신뢰성 있는 메시지 계층을 목표로 한다면 수신측도 각 메시지를 **정확히 한 번만** 받는다는 보장이 필요하다.

수신자가 중복된 메시지를 검출할 수 있으려면 발신자가 각 메시지를 구분해서 전송해야 하며 수신자는 각 메시지를 이전에도 받은 적이 있는지 파악할 수 있는 방법이



〈그림 50.5〉 **메시지와 확인(ack): 응답 누락**

필요하다. 수신자가 메시지를 중복해서 수신할 경우, 메시지에 대해 ack를 보내지만 응용 프로그램에게는 받은 데이터를 (아슬아슬하게) 전달하지는 않는다. 발신자는 ack를 받지만 메시지는 두 번 받지는 않으므로 위에서 언급한 것처럼 정확히 한 번씩 처리하는 시맨틱을 따르게 된다.

중복된 메시지를 검출하는 많은 방법들이 있다. 예를 들어, 발신자가 각 메시지를 위해 유일한 ID를 생성할 수 있으며 수신자는 지금까지 받은 모든 ID를 다 추적하도록 할 수 있다. 이 방법을 쓸 수는 있지만, 모든 ID들을 추적하려면 무한정의 메모리가 필요하기 때문에 엄두도 못 낼만큼 비싼 작업이 된다.

작은 양의 메모리를 사용하면서 이 문제를 해결하는 좀 더 간단한 방법은 **순서 카운터(sequence counter)**라고 하는 방법이다. 순서 카운터를 사용하기 위해서 발신자와 수신자가 양쪽에서 관리할 어떤 시작 값(예, 1)에 서로 동의한다. 메시지를 보내면서 현재의 카운터 값을 함께 전송한다. 이 카운터 값 (N)은 메시지의 ID 역할을 한다. 메시지가 전송된 후에 발신자는 값을 ($N+1$)로 증가한다.

수신자는 이 카운터 값을 발신자 측으로부터 도착하는 메시지의 예상 ID의 값으로 사용한다. 만약 수신한 메시지의 ID (N)이 수신자의 카운터 (마찬가지로 N)와 동일하다면 메시지에 대해 ack를 보내고 메시지를 응용 프로그램에 전달한다. 이 경우에 수신자는 이 메시지가 처음 받은 것이라고 결론을 내린다. 수신자는 이후에 카운터를 ($N+1$로) 증가시키고 다음 메시지를 기다린다.

ack가 손실이 된 경우 발신자는 타임아웃으로 인해 메시지 N을 재전송할 것이다. 이번에는 수신자의 카운터가 ($N+1$)로 크기 때문에 발신자는 메시지를 이미 받았었다는 것을 안다. 그러므로 메시지에 대해서 ack를 전송하지만 메시지를 응용 프로그램에게 전달하지는 *않는다*. 이런 간단한 방식의 순서 카운터를 사용하여 중복 수신을 피할 수 있다.

가장 흔히 사용되는 신뢰할 수 있는 통신 계층은 **TCP/IP** 또는 짧게 **TCP**라고

팁: 무결성을 위해 체크섬을 사용하자

체크섬은 손상을 빠르게 그리고 효율적으로 검출하기 위해서 현대 시스템에서 흔히 사용하는 기법이다. 간단한 체크섬은 덧셈 방식이다. 데이터의 바이트 청크를 그냥 더하는 것이다. 물론, 다른 더 정교한 체크섬들이 개발되었다. 그 중에는 순환 중복 코드(CRC)와 Fletcher 체크섬이 있으며 그 외에도 더 있다 [MK09].

네트워킹의 체크섬은 다음과 같이 사용된다. 한 기계에서 다른 기계로 메시지를 전송하기 전에 메시지의 체크섬을 계산한다. 그리고 메시지와 체크섬을 목적지로 전송한다. 목적지에서 수신자는 받은 메시지로 체크섬 계산을 똑같이 한다. 만약 계산된 체크섬이 보낸 체크섬과 일치한다면 수신자는 전송 중에 데이터가 손상되지 않았다는 것을 확신할 수 있다.

체크섬은 여러 다른 기준으로 평가될 수 있다. 그 중에 효율성이 주된 고려 사항이다. 데이터가 변경되면 체크섬도 변경되는가? 강력한 체크섬일수록 데이터의 변경에 즉시 영향을 받는다. 성능 또한 다른 중요한 평가 기준이다. 체크섬의 계산 비용이 얼마나 되는가? 불행하게도 효율성과 성능은 서로 상충된다. 고품질의 체크섬은 대부분 복잡한 계산 과정을 수반한다. 인생이란, 언제나 그랬듯이, 완벽하지 않다.

부른다. TCP는 위에서 설명한 것보다 훨씬 더 정교한 기법으로 이루어져 있다. 네트워크의 혼잡도 관리 기법과 다중의 대기 중 요청들을 지원하며 수백 가지의 작은 수정과 최적화 기법들을 포함하고 있다 [Jac88]. 궁금하다면 관련 문헌들을 더 읽어보기 바란다. 더 좋은 방법은 네트워크 수업을 듣고 관련 내용을 숙지하는 것이다.

50.4 통신 추상화

기본 메시징 계층을 다루었으니 이제 새로운 질문을 해 보자. 분산 시스템을 구현하는 데 필요한 통신 개념은 무엇일까?

시스템 분야의 공동체는 수년에 걸쳐서 여러 기법들을 개발하였다. 어떤 부류는 운영체제 개념을 확장시켜서 분산 환경에 적용하였다. 예를 들어, **분산 공유 메모리(distributed shared memory, DSM)** 시스템은 하나의 프로세스가 서로 다른 기계들 위에서 커다란 가상 주소 공간을 공유할 수 있도록 하였다 [LH89]. 이 과정을 통해 분산된 연산이 마치 멀티 쓰레드 응용 프로그램처럼 보이도록 만들었다. 유일한 차이가 있다면 이 쓰레드들이 하나의 기계의 다른 프로세서들에서 실행되는 것이 아니라 다른 기계에서 실행된다는 것이다.

대부분의 DSM 시스템은 운영체제의 가상 메모리 시스템 기반으로 동작한다. 페이지가 접근되었을 두 가지 경우가 일어날 수 있다. 첫 번째(최선)는 페이지가 이미 기계 내에 있어서 빠르게 데이터를 가져올 수 있는 경우다. 두 번째는 페이지가 현재 다른 기계에 있는 경우이다. 이때는 페이지 폴트가 발생한다. 페이지 폴트 핸들러는 다른 기계에게 메시지를 보내 페이지를 달라고 요청하고, 그 결과를 프로세스의 페이지 테이블에 삽입한 후 실행을 계속한다.

팁: 타임아웃 값 설정에 주의를 기울이라

설명에서 짐작했을 수도 있겠지만 타임아웃을 사용하여 메시지 전송을 재시도하는 데 있어서 중요한 것은 타임아웃 값을 제대로 정하는 것이다. 만약 타임아웃이 너무 작다면 발신자는 불필요하게 메시지를 재전송하여 발신자의 CPU 시간과 네트워크 자원을 낭비하게 된다. 만약 타임아웃이 너무 크다면, 발신자는 너무 긴 시간을 기다렸다가 재전송을 하여 발신자 측의 성능이 줄어들게 된다. 단일 클라이언트와 서버 측면에서 "올바른" 값이란 것은 패킷 손실 여부를 알 수 있을 정도까지만 길고 그 이상은 기다리지 않도록 하는 값이다.

하지만 앞으로 다룰 장들에서 보게 되겠지만, 분산 시스템에는 대체적으로 하나 이상의 클라이언트와 서버가 있다. 여러 클라이언트가 하나의 서버로 전송하는 시나리오에서는 서버에 걸린 오버헤드의 정도를 나타내는 지표로 서버 측의 패킷 손실률을 사용할 수 있다. 서버가 오버헤드에 걸린 상황이라면 클라이언트는 다른 적응적 방식으로 재시도할 수 있다. 예를 들어 클라이언트는 첫 번째 타임아웃 이후에 값을 더 크게 증가시켜서 원래 값의 두 배가 되도록 할 수도 있다. 이와 같은 **지수적 백오프(exponential back-off)** 기법은 재전송으로 인해서 자원이 고갈되는 상황을 피하기 위해 초기의 Aloha 네트워크에서 개발되었고 초기 이더넷에 적용되었다 [Abr70]. 강인한 시스템들은 오버헤드를 줄이기 위해 이런 류의 노력을 한다.

이와 같은 기법은 현재에 여러 가지 이유로 대중적으로 사용되고 있지 않다. DSM의 가장 큰 문제는 실패를 처리하는 방식에 있다. 예를 들어, 기계가 고장났다고 가정하자. 그 기계의 페이지는 어떻게 되겠는가? 분산 연산의 자료 구조가 전체 주소 공간에 퍼져 있다면 어떻게 될까? 이 경우에는 자료 구조의 일부가 갑자기 사용불가 상태가 된다. 주소 공간의 일부가 사라지는 경우를 해결하기는 쉽지 않다. 연결 리스트에서 다음을 가리키는 포인터가 사라진 주소 공간을 가리키고 있다고 생각해 보자. 아이쿠! 이보다 더 큰 문제는 성능이다. 일반적으로 코드를 작성할 때 가정은 메모리 접근은 매우 빠르다고 가정한다. DSM 시스템에서는 어떤 메모리 접근은 빠르지만, 어떤 것들은 페이지 폴트가 발생하기 때문에 원격 기계에서 페이지를 가져와야 한다. DSM 시스템 프로그래머들은 통신이 발생하지 않도록 상당한 주의를 기울여 코드를 작성해야 한다. 그러면 사실 DSM을 쓸 필요가 없어진다. 이 분야에서 수많은 연구가 진행되었지만 실제 영향은 적었다. 이제는 누구도 DSM을 사용하여 신뢰할 수 있는 분산 시스템을 만들지 않는다.

50.5 Remote Procedure Call (RPC)

분산 시스템을 구현하는 데 있어서 운영체제의 개념들을 활용하는 것은 좋지 않은 선택이었고, 대신 프로그래밍 언어 (PL) 차원의 개념이 더 이치에 맞는다는 것을 알게 되었다. 가장 지배적인 개념은 **Remote Procedure Call** 또는 짧게 **RPC**라고 부르는

것에 기반하고 있다 [BN84][1]

Remote Procedure Call 은 간단한 목표를 갖고 있다. 원격 기계에서의 코드 실행을 로컬 내의 함수를 부르는 것처럼 간단하게 복잡하지 않게 만드는 것이다. 클라이언트는 프로시저 호출을 하고 잠시 후에 결과를 리턴받는다. 서버는 공지할(export) 루틴을 정의한다. 나머지는 RPC 시스템이 두 부분으로 나누어 담당한다. 바로 **스텁 생성기(stub generator, 또는 프로토콜 컴파일러(protocol compiler)**라고 함)와 **런타임 라이브러리(run-time library)**이다. 이제 그 각각의 부분에 대해서 상세하게 다뤄보도록 하겠다.

스텁 생성기

스텁 생성기(Stub Generator)가 하는 일은 간단하다. 함수의 인자들을 묶는 불편함을 없애고 자동적으로 메시지를 만드는 것이 스텁 생성기의 일이다. 많은 장점들이 있다. 수작업으로 그런 코드를 작성할 경우 발생할 수 있는 작은 실수들을 막아 주며, 더 나아가 스텁 컴파일러가 그런 코드를 최적화할 수도 있기 때문에 성능을 개선할 수도 있다.

서버가 클라이언트에게 공지할 프로시저들의 집합을 컴파일러에 입력으로 전달한다. 개념 상으로는 다음과 같이 아주 간단하게 구성될 수 있다.

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

스텁 생성기는 이런 종류의 인터페이스를 사용하여 몇 개의 다른 코드를 생성해 낸다. 클라이언트 용으로 인터페이스에 명시된 함수들로 구성된 **클라이언트 스텁(client stub)**을 생성한다. 클라이언트 프로그램이 RPC 서비스를 이용하기 원하면 클라이언트 스텁을 생성하여 호출한다.

클라이언트 스텁의 각각의 함수들이 원격 프로시저 호출을 위한 일을 처리한다. 클라이언트 코드는 함수 호출처럼 보인다(예, 클라이언트가 `func1(x)`를 호출). 클라이언트 스텁의 `func1()`의 코드는 다음과 같은 일을 한다.

- **메시지 버퍼 생성.** 일반적으로 메시지 버퍼는 특정 크기의 배열이다.
- **메시지 버퍼에 필요 정보를 병합.** 이 정보에는 함수 식별자와 함수 인자들이 포함된다(예, 앞에서의 예제에서는 `func1`의 `int` 형 하나). 버퍼에 이 정보를 담는 과정을 인자들을 **합병하기(marshaling)** 또는 메시지를 **직렬화하기(serialization)**라고 표현하기도 한다.
- **RPC 서버에 메시지 전송.** RPC 서버와 통신 그리고 동작하는 데 필요한 모든 상세 동작은 RPC 런타임 라이브러리가 담당한다. 구체적인 내용은 아래와 같다.

1) 현대의 프로그래밍 언어들에서는 원격 메소드 호출(**remote method invocation, RMI**)이라고 부르기도 한다. 그런데, 현란한 객체투성인데, 누가 이런 언어들을 좋아한단 말인가?

- **응답 대기.** 함수 호출은 대부분 **동기식(synchronous)**이기 때문에 완료가 될 때까지 대기를 한다.
- **리턴 코드와 인자 풀기.** 함수가 하나의 리턴 코드만 리턴하면 이 프로세스는 단순하다. 하지만 복잡한 함수의 경우 좀 더 복잡한 결과를 리턴할 수도 있다(예, 리스트). 그러므로 스텁의 내용들을 다 풀어내야 할 필요가 있다. 이 과정을 **합병해제하기** 또는 **역직렬화**라고 부른다.
- **호출자에게 리턴하기.** 클라이언트 스텁으로부터 최종적으로 클라이언트 코드를 리턴한다.

서버 용 코드도 생성된다. 서버 측에서 수행되는 과정은 다음과 같다.

- **메시지 풀기.** 이 단계에서는 도착하는 메시지에서 **합병해제하기** 또는 **역직렬화**를 통해서 정보를 추출한다. 함수 식별자와 인자들이 추출된다.
- **실제 함수를 호출하기.** 마침내! 원격 함수가 실제로 실행되는 지점에 도달하였다. RPC 실행 시간 호출이 ID로 지정되어 있는 함수를 부르며 해당하는 인자를 전달한다.
- **결과를 통합.** 리턴 인자(들)는 응답 버퍼에 다시 합병한다.
- **응답 전송하기.** 응답이 마침내 호출자에게 전송된다.

스텁 컴파일러에서 고려해야 할 몇 가지 중요한 문제들이 있다. 첫 번째는 복잡한 구조의 인자나 다수의 인자를 전달하는 문제이다. 즉, 복잡한 자료 구조를 어떻게 패키지화해서 전송할 것인가? 예를 들어 `write()` 시스템 콜을 사용할 때는 세 개의 인자를 전달해야 한다. `int` 형의 파일 디스크립터와 버퍼의 포인터 그리고 써야 할 바이트를 나타내는 크기(포인터를 시작으로)를 전달해야 한다. 만약 RPC 패키지가 포인터를 전달했다면, 그 포인터를 어떻게 해석해야 하는지를 알아야 정확한 동작을 할 수 있다. 일반적으로 잘 알려진 데이터 형(예, RPC 컴파일러가 이해할 수 있는 `buffer_t`라는 자료 구조를 사용하여 특정 크기의 데이터 청크를 전달한다)을 사용하거나 자료 구조에 해석하는 방법에 대한 내용을 추가하여 컴파일러가 바이트의 어떤 부분을 직렬화할지 알 수 있도록 한다.

또 다른 문제는 병행성을 고려하여 서버를 구성하는 것이다. 단순한 서버는 간단한 반복문에서 요청을 대기하며 한 번에 한 요청씩 처리한다. 하지만, 예상한 대로 엄청나게 비효율적일 수 있다. 만약 RPC 호출이 차단되면(예, I/O를 기다리며) 서버의 자원이 낭비된다. 그러므로 대부분의 서버들은 병행성을 이용하여 구성된다. 흔한 구성 방식은 **쓰레드 풀(thread pool)**을 사용하는 것이다. 이 구성 방식은 서버를 시작할 때 정해진 수의 쓰레드들을 생성한다. RPC 호출이 도착하면 메인 쓰레드가 워커 쓰레드로 보낸다. 메인 쓰레드는 계속 RPC 호출을 받기 위해 대기한다. 또 다른 요청이 도착하면, 다시 다른 워커 쓰레드에게 전달한다. 이와 같은 방식으로 서버 내에서 병행 실행을 활용하여 활용률을 높일 수 있다. RPC 서버를 구성하는 데 필요한 프로그래밍 복잡도가 약간 늘어난다. RPC 호출의 올바른 동작을 위해 락이나 기타 동기화 기법들을 써야 하기 때문이다.

런타임 라이브러리

런타임 라이브러리는 RPC 시스템에서 대부분의 중요한 일을 책임진다. 성능과 신뢰성에 관한 대부분의 문제들을 처리하고 있다. 런타임 계층을 구현하는 데 필요한 핵심 사안 몇 가지를 다루도록 하겠다.

극복해야 하는 첫 번째 도전거리 중에 하나는 원격 서비스의 위치를 찾는 문제이다. **이름(naming)**에 관한 이 문제는 분산 시스템에서는 흔한 것이고 또 어떤 면에서는 현재 논의 주제에서 벗어난 것일 수도 있다. 가장 간단한 방법은 기존의 시스템을 활용하는 것이다. 다시 말해, 현재의 인터넷 프로토콜이 사용하는 호스트의 이름과 포트 번호를 활용하는 것이다. 그런 시스템에서 클라이언트는 RPC 서비스를 실행하기를 원하는 기계의 호스트 이름 또는 IP 주소 그리고 포트 번호를 반드시 한다(포트 번호는 기계에서 일어나는 통신 작업을 구별할 수 있는 방법이며, 이를 통해 여러 개의 통신 작업들이 동시에 진행될 수 있다). 그 다음, 패킷이 특정 주소로부터 시스템에 있는 임의의 다른 기계로 전달될 수 있는 메커니즘을 제공해야 한다. 이 주제와 관련하여서는 Grapevine 논문이나 인터넷의 DNS와 이름 해석 방법에 대해서 살펴보거나 또 다른 좋은 방법으로 Saltzer와 Kaashoek의 책에 훌륭하게 설명된 장을 읽어 보기를 바란다 [SK09].

클라이언트가 원격지의 서비스를 위해 어떤 서버와 소통할지 알아냈다면 그 다음의 질문은 RPC를 어떤 전송 계층(Transport Layer) 프로토콜 위에 만들지를 결정해야 한다. 구체적으로, RPC 시스템이 신뢰할 수 있는 TCP/IP와 같은 것을 사용할 것이냐 아니면 UDP/IP와 같은 신뢰할 수 없는 통신 계층 위에 만들 것이냐?

별 생각없이 선택하자면 간단해 보인다. 당연히 원격 서버에 요청이 확실하게 전달이 되기를 원하며 응답도 확실하게 받고 싶을 것이다. 그러므로 TCP와 같은 신뢰할 수 있는 전송 프로토콜을 사용해야 할 것이다. 그렇지 않은가?

불행하게도 신뢰할 수 있는 통신 계층 상에 RPC를 구현하면 성능이 엄청나게 안 좋아진다. 앞서 설명한 신뢰할 수 있는 통신 계층의 동작 방식을 기억해 보자. 확인(acknowledgement, ack)과 타임아웃/재시도 과정이 포함이 되어 있다. 그러므로 클라이언트가 RPC 요청을 서버로 보내면 서버는 ack로 응답하여 요청을 받았다는 것을 호출자가 알 수 있도록 한다. 유사하게 서버가 응답을 클라이언트에게 전송할 때 클라이언트가 서버에게 ack를 보내서 잘 받았다는 것을 알려야 한다. 요청/응답 방식의 프로토콜(RPC와 같은)을 신뢰할 수 있는 통신 계층 위에서 구현하면 두 개의 "추가" 메시지를 보내야 한다.

이러한 이유로 많은 RPC 패키지들은 신뢰할 수 없는 통신 계층인 UDP와 같은 것을 사용하여 구현한다. 그러면 효율적으로 RPC 계층을 만들 수 있지만 RPC 시스템이 신뢰성 담보를 책임져야 한다. RPC 계층은 앞서 설명했던 타임아웃/재시도 그리고 ack 방식을 충분히 활용하여 원하는 수준의 신뢰도를 달성한다. 통신 계층은 순서 번호 같은 것을 사용하여 각 RPC가 단 한 번만(실패가 전혀 없는 경우라면) 또는 많아야 한 번만(실패가 있었다면) 발생하도록 보장한다.

다른 문제들

RPC 런타임이 다루어야 할 다른 문제들이 있다. 예를 들면, 원격 호출이 완료까지 오랜 시간이 걸린다면 어떻게 해야 하는가? 타임아웃 방식을 사용하면 장 시간 동작하는 원격 호출은 클라이언트에게 실패한 것으로 보일 것이기 때문에 재시도를 하게 될 것이다. 그렇기 때문에 이 부분을 수정해야 한다. 한 가지 해법은 응답이 즉시 생성되지 않는 경우에는 먼저 ack를 보내는 것이다(수신자가 발신자에게). 클라이언트는 서버가 요청을 받았다는 것을 알 수 있다. 어느 정도 시간이 흐른 후, 클라이언트는 주기적으로 서버가 해당 요청을 처리 중인지 확인한다. 그때마다 서버가 "작업 중"이라고 한다면 클라이언트는 즐겁게 계속해서 대기하면 된다(프로시저 호출이 완료될 때까지 아주 오랜 시간이 걸리는 때가 있다).

런타임도 마찬가지로 한 패킷에 담을 수 있는 양보다 더 많은 수의 인자를 갖는 프로시저 호출들을 처리할 수 있어야 한다. 어떤 하부 계층의 네트워크 프로토콜은 이런 발신자 측 **분철**(**fragmentation**, 큰 패킷을 여러 작은 조각들로 나누는 것)과 수신자 측 **재조합**(**reassembly**, 작은 조각들을 전체가 되는 논리적 큰 청크로 만들기) 기법을 제공한다. 그렇지 않다면, RPC 런타임은 이와 같은 기능을 자체적으로 구현을 해야 한다. 상세 내용은 Birrell과 Nelson이 쓴 훌륭한 RPC 논문을 참고하자 [BN84].

많은 시스템들이 다루어야 하는 또 다른 문제는 **바이트 순서 표시**(**byte ordering**)에 관한 것이다. 이미 알겠지만, 어떤 기계들은 **빅 엔디안**(**big endian**) 순으로 값을 저장하고 어떤 기계들은 **리틀 엔디안**(**little endian**)을 쓰기도 한다. 빅 엔디안은 바이트를(int 형이라고 하자) 최상위 비트부터 최하위 비트까지 아라비아 숫자처럼 표기하지만 리틀 엔디안은 그 반대로 표현한다. 두 방식 모두 수치 정보를 저장하는 올바른 방식이다. 여기서의 질문은 어떻게 서로 *다른* 저장 방식을 사용하는 기계들끼리 통신을 하느냐이다.

RPC 패키지는 메시지 포맷에 잘 정의되어 있는 엔디안을 사용하는 것으로 이 문제를 해결한다. Sun의 RPC 패키지에서는 **XDR**(**eXternal Data Representation**, 외부 데이터 표현 방식) 계층이 그 기능을 제공한다. 만약 기계가 XDR의 엔디안을 준수하여 메시지를 발신 또는 수신하면, 기대한 대로 메시지를 발신하고 수신할 수 있다. 하지만, 만약 기계가 다른 엔디안을 사용하여 통신한다면, 메시지의 각 정보는 변환이 되어야 한다. 그러므로 엔디안의 차이는 약간의 성능 비용을 지불해야 한다.

마지막 문제는 클라이언트에게 비동기적 실행을 허가할 것인가의 문제이다. 그에 따른 성능 개선이 가능하다. 구체적으로, 어떤 RPC는 **동기적**으로 동작한다. 즉, 클라이언트가 프로시저 호출을 요청하면 그 결과가 리턴될 때까지 대기한다. 대기 시간이 길어질 수도 있다. 클라이언트가 다른 일을 처리할 수 있도록, 어떤 RPC 패키지는 RPC를 **비동기적**으로 호출하기도 한다. 비동기 RPC가 호출이 되면 RPC 패키지는 요청을 보내고 즉시 리턴한다. 클라이언트는 그 후에 다른 RPC를 호출한다거나 유용한 연산을 하는 식의 다른 작업을 자유롭게 진행할 수 있다. 어느 시점에 가서는 클라이언트가 비동기 RPC에 대한 결과를 원할 것이다. 그때 RPC를 호출하여, 현재 진행 중인 RPC를 대기토록 한다. 완료되면 리턴된 인자들을 접근할 수 있다.

여담: 단-대-단(end-to-end) 인자에 대한 논쟁

시스템의 최상위 계층, 즉 일반적으로 "가장 끝 단"에 있는 응용 프로그램이 계층적으로 되어 있는 시스템에서 궁극적으로 특정 기능이 실제로 구현될 수 있는 유일한 위치라는 것이 end-to-end 인자의 필요성을 설명한다. Saltzer 등이 쓴 그들의 랜드마크 논문에서 이 내용을 두 기계 간의 신뢰할 수 있는 파일 전송에 관한 훌륭한 예를 들어 주장하였다 [SRC84]. 기계 A에서 기계 B로 파일을 전송하기를 원하고 B에서 받는 바이트들이 A에서 전송한 바이트와 정확히 일치하기를 원한다면 그 내용을 "end-to-end" 검사를 통해 확인해야 한다. 네트워크나 디스크에 있는 하위 계층의 신뢰성 기법들은 그런 보장을 할 수가 없다.

대비되는 접근법은 신뢰할 수 있는 파일 전송 문제를 해결하기 위해 시스템의 하위 계층에 신뢰성을 더하는 것이다. 예를 들어 신뢰할 수 있는 통신 프로토콜을 구성하였고 그것을 이용하여 신뢰할 수 있는 파일 전송 방법을 만든다고 하자. 타임아웃/재시도와 ack 그리고 순차 번호 등을 사용하는 통신 프로토콜의 경우 발신자가 전송하는 모든 바이트가 수신자에게 순서대로 전달될 수 있도록 할 것이다. 불행하게도 그런 프로토콜을 사용하더라도 신뢰성 있는 파일 전송을 할 수 없다. 통신을 하기 이전에 이미 발신자의 메모리에서 일부 바이트가 손상되었다고 해 보자. 또는 수신자가 디스크에 데이터를 기록할 때 안 좋은 일이 발생했다고 해 보자. 그러한 경우들에는 네트워크를 통해 신뢰할 수 있는 데이터가 전송되었다 하더라도 파일 전송은 근본적으로 신뢰할 수 없는 것이 된다. 신뢰할 수 있는 파일 전송을 만들기 위해서는 단-대-단 신뢰성 검사를 포함하여야 한다. 예를 들면, 전체 전송이 완료되면 수신자 측 디스크에서 파일을 다시 읽어서 체크섬을 계산하고, 발신자 측의 파일의 체크섬과 비교를 해야 한다.

이 금언에 따른 귀결은 하위 계층에서 추가적인 기능을 제공한다면, 때로는 시스템의 성능을 개선하거나 또는 시스템의 최적화를 이룰 수 있다는 것이다. 그러므로 시스템의 하위 계층에 그런 기법들을 포함하는 것을 제외시켜서는 안 된다. 오히려 기법들을 사용하는 전체 시스템이나 또는 응용 프로그램을 고려하여 그러한 기법들의 활용도를 신중하게 생각해 보아야 한다.

50.6 요약

분산 시스템이라는 새로운 주제와 실패와 오류를 다루는 법에 대한 소개를 하였다. 구글에서 말하듯이 데스크탑 기계만 사용할 때 실패 오류가 드물지만 수천 대의 기계로 이루어진 데이터 센터에 있으면 늘 상 겪는 것이 실패 오류이다. 분산 시스템의 핵심은 그 실패 오류들을 대처하는 것이다.

어느 분산 시스템이든 그 핵심은 통신의 형태로 나타난다. 통신의 흔한 개념은 원격 절차 호출(RPC)이며 이를 통해 클라이언트가 서버들로 원격 호출을 요청할 수 있다. 절차 호출이 마치 로컬에서 처리되는 것처럼 보이는 서비스를 제공하기 위해서 RPC 패키지는 타임아웃/재시도와 ack와 같은 상세한 세부 내용을 처리한다.

RPC 패키지를 정말 제대로 이해하는 최선의 방법은 당연하겠지만 직접 사용해보는 것이다. Sun의 RPC 시스템인 스텁 컴파일러 `rpcgen`은 흔히 사용되는 것이며 Linux를 포함하여 요즈음 시스템에서 많이들 사용하는 것이다. 사용해 보고 골치 아픈 이유를 한 번 살펴보라.

참고 문헌

[Abr70] **"The ALOHA System. Another Alternative for Computer Communications"**
Norman Abramson
The 1970 Fall Joint Computer Conference
ALOHA 네트워크는 지수적 백오프와 재전송과 같은 네트워킹의 기본 개념들을 개척하였다. 그리고 그 개념들은 수년 동안 공유 버스 이더넷에서 통신의 기틀을 형성하였다.

[BN84] **"Implementing Remote Procedure Calls"**
Andrew D. Birrell and Bruce Jay Nelson
ACM TOCS, Volume 2:1, February 1984
모든 사람들이 기반으로 하는 RPC 시스템의 기초이다. 그렇다. 제록스 파크의 친구들이 일구어 낸 또 하나의 선구자적인 노력의 산실이다.

[Jac88] **"Congestion Avoidance and Control"**
Van Jacobson
SIGCOMM '88
클라이언트가 네트워크의 혼잡을 인지하려면 어떻게 조정되어야 하는지를 설명한 선구자적 논문이다. 인터넷을 구성하는 핵심적인 기술 중의 하나임에 분명하다. 시스템을 진지하게 생각하는 사람과 Van Jacobson의 친척들이라면 꼭 읽어야 하는 글이다. 왜냐하면 친척들의 글들은 다 읽어야 하기 때문이다.

[LH89] **"Memory Coherence in Shared Virtual Memory Systems"**
Kai Li and Paul Hudak
ACM TOCS, 7:4, November 1989
가상 메모리를 통한 소프트웨어 기반의 공유 메모리에 대한 소개이다. 아주 흥미로운 개념이 아닐 수 없지만 끝까지 좋은 개념으로 남지는 못하였다.

[MK09] **"The Effectiveness of Checksums for Embedded Control Networks"**
Theresa C. Maxino and Philip J. Koopman
IEEE Transactions on Dependable and Secure Computing, 6:1, January '09
기본적인 체크섬 기법에 대한 잘 설명된 개론으로서 성능과 강인성에 대한 비교가 포함 되어 있다.

[SK09] **"Principles of Computer System Design"**
Jerome H. Saltzer and M. Frans Kaashoek
Morgan-Kaufmann, 2009
시스템에 대한 훌륭한 책으로서 모든 책장에 꼽혀 있어야 할 책이다. 우리가 살펴본 작명에 대해서 손에 꼽힐 정도로 아주 멋지게 서술하였다.

[SRC84] **"End-To-End Arguments in System Design"**
Jerome H. Saltzer, David P. Reed, and David D. Clark
ACM TOCS, 2:4, November 1984
계층과, 추상화 그리고 컴퓨터 시스템에서 기능이 궁극적으로 어느 위치에 있어야 하는지에 대해 아름답게 서술하였다.