
소켓 프로그래밍 HOWTO

릴리스 3.13.0

Guido van Rossum and the Python development team

10월 19, 2024

Contents

1	소켓	1
1.1	역사	2
2	소켓 만들기	2
2.1	IPC	3
3	소켓 사용하기	3
3.1	바이너리 데이터	4
4	연결 끊기	5
4.1	소켓이 죽을 때	5
5	비블로킹 소켓	5

저자
Gordon McMillan

개요

소켓은 거의 모든 곳에서 사용되지만, 가장 심하게 오해된 기술 중 하나입니다. 이것은 10,000피트 상공에서 본 소켓 개요입니다. 진짜 자습서는 아닙니다 - 여러분은 여전히 작동하도록 만들기 위해 해야 할 일이 있습니다. 세부 사항을 다루지는 않습니다만 (그것 것들이 많이 있습니다), 그것들을 적당히 사용하기에 충분한 배경을 줄 수 있기를 바랍니다.

1 소켓

INET (즉, IPv4) 소켓에 관해서만 이야기할 것이지만, 사용 중인 소켓의 99% 이상을 차지합니다. 또한, STREAM (즉, TCP) 소켓에 관해서만 이야기할 것입니다 - 여러분이 무엇을 하고 있는지 정말로 알고 있지 (그럴 때 이 HOWTO는 필요 없습니다!) 않다면, 다른 모든 것보다 STREAM 소켓으로 더 나은 동작과 성능을 얻을 수 있습니다. 소켓이 무엇인지에 대한 수수께끼뿐만 아니라 블로킹과 비 블로킹 소켓으로 작업하는 방법에 대한 힌트를 분명하게 하려고 합니다. 하지만 블로킹 소켓에 관해 이야기하는 것으로 시작할 것입니다. 비 블로킹 소켓을 다루기 전에 이것이 어떻게 작동하는지 알아야 합니다.

이러한 것들을 이해하는데 어려움을 주는 한 부분은 문맥에 따라 “소켓”이 여러 가지 미묘하게 다른 것을 뜻할 수 있다는 것입니다. 그래서 먼저, 대화의 끝점인 “클라이언트” 소켓과, 배전반 운영자와 비슷한 “서버” 소켓을 구별해 보겠습니다. 클라이언트 응용 프로그램(예를 들어, 여러분의 브라우저)은 “클라이언트” 소켓만 사용합니다; 이것이 대화하는 웹 서버는 “서버” 소켓과 “클라이언트” 소켓을 모두 사용합니다.

1.1 역사

IPC (Inter Process Communication, 프로세스 간 통신)의 다양한 형태 중에서, 소켓이 가장 많이 사용됩니다. 특정 플랫폼에서 다른 형태의 IPC가 더 빠를 가능성이 있지만, 크로스 플랫폼 통신의 경우 소켓이 유일한 게임의 법칙입니다.

They were invented in Berkeley as part of the BSD flavor of Unix. They spread like wildfire with the internet. With good reason — the combination of sockets with INET makes talking to arbitrary machines around the world unbelievably easy (at least compared to other schemes).

2 소켓 만들기

대충 말하면, 이 페이지로 연결되는 링크를 클릭하면 브라우저가 다음과 같은 작업을 수행합니다:

```
# create an INET, STREAMing socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# now connect to the web server on port 80 - the normal http port
s.connect(("www.python.org", 80))
```

connect가 완료되면, 소켓 *s*를 사용하여 페이지의 텍스트 요청을 보낼 수 있습니다. 같은 소켓으로 응답을 읽은 다음 파괴됩니다. 그렇습니다, 파괴됩니다. 클라이언트 소켓은 일반적으로 하나의 교환(또는 일련의 작은 교환 집합)에서만 사용됩니다.

웹 서버에서 일어나는 일은 좀 더 복잡합니다. 첫째, 웹 서버는 “서버 소켓”을 만듭니다:

```
# create an INET, STREAMing socket
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# bind the socket to a public host, and a well-known port
serversocket.bind((socket.gethostname(), 80))
# become a server socket
serversocket.listen(5)
```

알아 두어야 할 몇 가지 사항: 소켓을 외부 세계에서 볼 수 있도록 `socket.gethostname()`를 사용했습니다. `s.bind('localhost', 80)`이나 `s.bind('127.0.0.1', 80)`을 사용했다면, 여전히 “서버” 소켓을 가지게 되지만 같은 기계 내에서만 볼 수 있는 소켓을 갖게 됩니다. `s.bind('', 80)`은 시스템에 있는 모든 주소로 소켓에 연결할 수 있음을 나타냅니다.

두 번째로 주목해야 할 점: 낮은 번호의 포트는 일반적으로 “잘 알려진” 서비스(HTTP, SNMP 등)를 위해 예약되어 있습니다. 연습 중이라면 적당히 높은 번호(4 자릿수)를 사용하십시오.

마지막으로, `listen`에 대한 인자는 외부 연결을 거부하기 전에 최대 5개의 연결 요청을 큐에 넣기를 원하는 것을 소켓 라이브러리에 알립니다. 코드의 나머지 부분이 제대로 작성되었다면, 이것으로 충분합니다.

이제 우리는 포트 80에서 대기하는 “서버” 소켓을 가지고 있고, 웹 서버의 메인 루프를 입력할 수 있습니다:

```
while True:
    # accept connections from outside
    (clientsocket, address) = serversocket.accept()
    # now do something with the clientsocket
    # in this case, we'll pretend this is a threaded server
    ct = client_thread(clientsocket)
    ct.run()
```

실제로 이 루프가 작동할 수 있는 3가지 일반적인 방법이 있습니다 - `clientsocket`를 처리하기 위해 스레드로 보내거나, `clientsocket`를 처리할 새 프로세스를 만들거나, 비 블로킹 소켓을 사용하도록 이 응용 프로그램을 재구성하고, `select`를 사용하여 “서버” 소켓과 활성 `clientsocket`들 간에 다중화(multiplexing)합니다. 나중에 자세히 다룹니다. 지금 이해해야 할 중요한 점: 이것이 “서버” 소켓이 하는 전부입니다. 어떤 데이터도 보내지 않습니다. 어떤 데이터도 수신하지 않습니다. 단지 “클라이언트” 소켓을 생성할 뿐입니다. 각 `clientsocket`은 우리가 바인드 한 호스트와 포트로 `connect()`를 수행하는 다른 “클라이언트” 소켓에 대한 응답으로 만들어집니다. `clientsocket`를 만들자마자, 더 많은 연결을 기다리는 것으로

돌아갑니다. 두 개의 “클라이언트”는 자유롭게 대화를 나눌 수 있습니다 - 그들은 대화를 끝낼 때 재사용되는 어떤 동적으로 할당된 포트를 사용합니다.

2.1 IPC

한 기계의 두 프로세스 간에 빠른 IPC가 필요하다면, 파이프나 공유 메모리를 살펴야 합니다. AF_INET 소켓을 사용하기로 했다면, “서버” 소켓을 'localhost'에 바인드 하십시오. 대부분 플랫폼에서, 이것은 네트워크 코드의 두 개를 계층을 건너뛰는 지름길을 취할 것이고, 꽤 빨라집니다.

➡ 더 보기

multiprocessing은 교차 플랫폼 IPC를 고수준 API로 통합합니다.

3 소켓 사용하기

첫 번째로 주목해야 할 점은 웹 브라우저의 “클라이언트” 소켓과 웹 서버의 “클라이언트” 소켓은 같은 녀석이라는 것입니다. 즉, 이것은 “피어 투 피어(peer to peer)” 대화입니다. 또는 다른 방식으로 표현하면, 설계자로서, 대화를 위한 예절의 규칙이 무엇인지 결정해야 합니다. 일반적으로, connect하는 소켓이 요청이나 로그인을 보내 대화를 시작합니다. 그러나 이것은 설계상의 결정입니다 - 소켓의 규칙이 아닙니다.

이제 통신에 사용할 두 별의 동사가 있습니다. send와 recv를 사용하거나, 클라이언트 소켓을 파일류로 변환한 후 read와 write를 사용할 수 있습니다. 후자는 자바가 소켓을 제공하는 방식입니다. flush를 소켓에 사용해야 한다고 경고하는 것 외에는, 여기에 대해서는 언급하지 않을 것입니다. 이것들은 버퍼된 “파일”이며, 일반적인 실수는 어떤 것을 write하고는 응답을 read하는 것입니다. flush가 없으면, 요청이 여전히 출력 버퍼에 남아있을 수 있으므로 응답을 영원히 기다리게 될 수 있습니다.

이제 소켓의 주요 걸림돌에 도달했습니다 - send와 recv는 네트워크 버퍼에서 작동합니다. 이것들은 여러분이 넘겨준 모든 바이트를 처리하지 않을 수 있습니다, 그들의 주 관심사는 네트워크 버퍼를 처리하는 것이기 때문입니다. 일반적으로, 연관된 네트워크 버퍼가 채워지거나(send) 비워지면(recv) 반환됩니다. 그런 다음 처리 한 바이트 수를 알려줍니다. 메시지가 완전히 처리될 때까지 다시 호출하는 것은 여러분의 책임입니다.

recv가 0바이트를 반환하면, 다른 쪽이 연결을 닫았거나 닫고 있다는 뜻입니다. 이 연결에서 더는 데이터를 받지 못합니다. 영원히. 데이터를 성공적으로 보낼 수는 있습니다; 나중에 이것에 대해 더 이야기하겠습니다.

HTTP와 같은 프로토콜은 하나의 전송에만 소켓을 사용합니다. 클라이언트는 요청을 보낸 다음 응답을 읽습니다. 그게 전부입니다. 소켓은 버려집니다. 이는 클라이언트가 0바이트를 수신하여 응답의 끝을 감지할 수 있음을 뜻합니다.

그러나 추가 전송을 위해 소켓을 재사용할 계획이라면, 소켓에는 EOT (End of Transfer, 전송의 끝)가 없다는 것을 알아야 합니다. 반복합니다: 소켓 send 또는 recv가 0바이트를 처리한 후 반환되면 연결이 끊어진 것입니다. 연결이 끊어진 것이 아니라면, 소켓은 (당분간) 읽을 것이 아무것도 없다는 것을 알려주지 않을 것이므로, recv에서 영원히 기다릴 수 있습니다. 이것에 대해 조금 더 생각해보면, 소켓의 근본적인 진실을 깨닫게 될 것입니다: 메시지는 고정 길이거나 (역), 구분자로 표시되거나 (어깨를 으쓱), 얼마나 긴지 표시하거나 (훨씬 낫다), 연결을 닫아서 끝내야 합니다. 선택은 전적으로 여러분의 것입니다. (하지만 어떤 방법이 다른 것보다 올바릅니다).

연결을 끝내기를 원하지 않는다고 가정하면, 가장 간단한 해결책은 고정 길이 메시지입니다:

```
class MySocket:
    """demonstration class only
    - coded for clarity, not efficiency
    """

    def __init__(self, sock=None):
        if sock is None:
            self.sock = socket.socket(
                socket.AF_INET, socket.SOCK_STREAM)
```

(다음 페이지에 계속)

```

else:
    self.sock = sock

def connect(self, host, port):
    self.sock.connect((host, port))

def mysend(self, msg):
    totalsent = 0
    while totalsent < MSGLEN:
        sent = self.sock.send(msg[totalsent:])
        if sent == 0:
            raise RuntimeError("socket connection broken")
        totalsent = totalsent + sent

def myreceive(self):
    chunks = []
    bytes_recd = 0
    while bytes_recd < MSGLEN:
        chunk = self.sock.recv(min(MSGLEN - bytes_recd, 2048))
        if chunk == b'':
            raise RuntimeError("socket connection broken")
        chunks.append(chunk)
        bytes_recd = bytes_recd + len(chunk)
    return b''.join(chunks)

```

여기에 있는 전송 코드는 거의 모든 메시지전달 체계에서 사용할 수 있습니다 - 파이썬에서는 문자열을 보내고 `len()`를 사용하여 길이를 파악할 수 있습니다 (`\0` 문자가 포함되어 있어도). 더 복잡한 부분은 대부분 수신 코드입니다. (그리고 C에서도, 메시지가 `\0`을 포함하고 있을 때 `strlen`을 사용할 수 없다는 점을 제외하면 몹시 나쁘지는 않습니다.)

가장 쉬운 개선은 메시지의 첫 번째 문자를 메시지 유형의 표시자로 만들고, 유형이 길이를 결정하도록 하는 것입니다. 이제 두 개의 `recv`가 있습니다 - (적어도) 첫 번째 문자를 가져와서 길이를 조회할 수 있도록 하는 첫 번째와 나머지를 얻는 루프로 구성된 두 번째입니다. 구분자로 가기로 했다면, 임의의 체크 크기 (4096이나 8192는 네트워크 버퍼 크기와 종종 잘 맞습니다)로 수신하고 받은 내용에서 구분자를 검색하게 될 것입니다.

하나의 복잡성을 알아두어야 합니다: 여러분의 대화형 프로토콜이 (어떤 종류의 응답 없이) 여러 메시지를 연속적으로 보내는 것을 허락하고, `recv`에 임의의 체크 크기를 전달하면 다음 메시지의 시작 부분도 함께 읽는 일이 일어날 수 있습니다. 필요할 때까지 보관해 두어야 합니다.

메시지의 길이를 앞에 붙이면 (5자리 숫자라고 합시다) 더 복잡해집니다, (믿거나 말거나) 한 번의 `recv`로 5문자를 모두 얻을 수 없을 수 있기 때문입니다. 연습 중에는 이런 일이 일어나지 않을 것입니다; 하지만 네트워크 로드가 높으면, 두 개의 `recv` 루프를 사용하지 않는 한 여러분의 코드는 금방 망가지게 됩니다 - 길이를 결정하는 첫 번째와 메시지의 데이터 부분을 가져오는 두 번째입니다. 지저분합니다. 또한 `send`가 항상 한 번에 모든 것을 처리하지 못하는 것을 발견하게 될 것입니다. 그리고 이 단락을 읽었음에도, 결국 당신은 이 문제에 당하게 될 것입니다!

공간을 절약하고, 여러분의 성격을 단련하고, (그리고 제 경쟁적 지위를 유지하기 위해) 이러한 향상은 독자를 위한 연습으로 남겨둡니다. 이제 정리해봅시다.

3.1 바이너리 데이터

It is perfectly possible to send binary data over a socket. The major problem is that not all machines use the same formats for binary data. For example, [network byte order](#) is big-endian, with the most significant byte first, so a 16 bit integer with the value 1 would be the two hex bytes 00 01. However, most common processors (x86/AMD64, ARM, RISC-V), are little-endian, with the least significant byte first - that same 1 would be 01 00.

Socket libraries have calls for converting 16 and 32 bit integers - `ntohl`, `htonl`, `ntohs`, `htons` where “n” means *network* and “h” means *host*, “s” means *short* and “l” means *long*. Where network order is host order, these

do nothing, but where the machine is byte-reversed, these swap the bytes around appropriately.

In these days of 64-bit machines, the ASCII representation of binary data is frequently smaller than the binary representation. That's because a surprising amount of the time, most integers have the value 0, or maybe 1. The string "0" would be two bytes, while a full 64-bit integer would be 8. Of course, this doesn't fit well with fixed-length messages. Decisions, decisions.

4 연결 끊기

엄밀히 말하면, `close`를 사용하기 전에 소켓에 `shutdown`을 사용해야 합니다. `shutdown`은 반대편 소켓에 대한 권고입니다. 전달하는 인자에 따라, “더는 보내지 않을 것이지만, 여전히 들을 겁니다” 나 “듣고 있지 않습니다, 즐거웠습니다!”를 뜻할 수 있습니다. 그러나, 대부분 소켓 라이브러리는 프로그래머가 이 예절을 무시하는 방식으로 사용되었고 일반적으로 `close`가 `shutdown()`; `close()`와 같습니다. 따라서 대부분 상황에서 명시적인 `shutdown`은 필요하지 않습니다.

`shutdown`을 효과적으로 사용하는 한 가지 방법은 HTTP와 비슷한 교환에서입니다. 클라이언트는 요청을 보낸 다음 `shutdown(1)`을 수행합니다. 그러면 서버에 “이 클라이언트는 전송을 완료했지만 계속 받을 수 있습니다.”라고 말하게 됩니다. 서버는 0바이트의 수신으로 “EOF”를 감지할 수 있습니다. 요청을 완료했다고 가정할 수 있습니다. 서버가 응답을 보냅니다. `send`가 성공적으로 완료되면, 클라이언트는 여전히 수신 중입니다.

파이썬은 자동 `shutdown`을 한 걸음 더 나아가서, 소켓이 가비지 수집될 때 필요하면 자동으로 `close`를 수행한다고 말합니다. 그러나 이것에 의존하는 것은 매우 나쁜 습관입니다. `close`를 하지 않고 소켓이 사라지면, 반대편 끝은 여러분이 단지 느려지고 있다고 생각하면서 무한정 멈출 수 있습니다. 제발 완료되면 소켓을 `close`해 주세요.

4.1 소켓이 죽을 때

아마도 블로킹 소켓 사용에 관한 최악의 경우는 상대방이 (`close`를 수행하지 않고) 갑자기 다운되었을 때 일어나는 일입니다. 소켓이 멈출 수 있습니다. TCP는 신뢰성 있는 프로토콜이며, 연결을 포기하기 전에 아주 오랜 시간 동안 기다립니다. 스레드를 사용하고 있다면, 스레드 전체가 실질적으로 죽습니다. 이것에 대해 당신이 할 수 있는 일이 별로 없습니다. 블로킹 읽기를 수행하는 동안 록을 잡는 것과 같은 어리석은 짓을 하지 않는 한, 스레드는 자원을 많이 소비하지 않습니다. 스레드를 죽이려고 하지 마십시오 - 스레드가 프로세스보다 효율적인 부분적인 이유는 자원 재활용과 관련된 오버헤드를 회피한다는 것입니다. 즉, 스레드를 죽이면 전체 프로세스가 엉망이 될 가능성이 있습니다.

5 비 블로킹 소켓

앞의 내용을 이해했다면 소켓을 사용하는 방법에 대해 알아야 할 대부분을 이미 알고 있습니다. 거의 같은 방식으로 같은 호출을 계속 사용합니다. 여러분이 올바르게 사용하기만 한다면, 여러분의 앱을 거의 완전해질 겁니다.

파이썬에서, `socket.setblocking(False)`를 사용하여 비 블로킹으로 만듭니다. C에서는, 더 복잡하지만 (한가지 예를 들면, BSD 계열의 `O_NONBLOCK`과 POSIX 계열의 거의 같은 `O_NDELAY` 중에서 선택해야 합니다; `O_NDELAY`는 `TCP_NODELAY`와는 완전히 다른 것입니다) 똑같은 아이디어입니다. 소켓을 만든 후에, 하지만 사용하기 전에 이것을 수행합니다. (실제로는, 여러분이 괴짜라면, 계속 변경할 수 있습니다.)

동작의 주요 차이점은 `send`, `recv`, `connect` 및 `accept`가 아무것도 하지 않고 반환될 수 있다는 것입니다. 여러분에게는 (물론) 많은 선택지가 있습니다. 반환 코드와 에러 코드를 확인하면서 일반적으로 자신을 미치게 만들 수 있습니다. 믿기지 않는다면, 한번 시도해보십시오. 여러분의 앱은 커지고, 버그가 많으며 CPU를 소진할 겁니다. 그러니 뇌사에 이르게 할 해결책은 건너뛰고 올바르게 해 보십시오.

`select`를 사용하십시오.

C에서, `select` 코딩은 상당히 복잡합니다. 파이썬에서, 이것은 달콤한 조각이지만, 여러분이 파이썬에서 `select`를 이해한다면 C에서도 거의 문제가 없을 만큼 C 버전이 아주 가깝습니다:

```
ready_to_read, ready_to_write, in_error = \
    select.select(
```

(다음 페이지에 계속)

```
potential_readers,
potential_writers,
potential_errs,
timeout)
```

`select`로 세 개의 리스트를 전달합니다: 첫 번째는 읽으려는 모든 소켓을 포함합니다; 두 번째는 쓰려는 모든 소켓, 그리고 마지막으로 (보통 비어있는데) 에러를 검사하려는 소켓들입니다. 소켓이 둘 이상의 리스트에 들어갈 수 있음에 유의해야 합니다. `select` 호출은 블로킹이지만, 시간제한을 지정할 수 있습니다. 이것은 일반적으로 민감한 작업입니다 - 달리할 좋은 이유가 없다면 긴 제한 시간(가령 1분)을 주십시오.

반환 값으로, 세 개의 리스트를 얻게 됩니다. 실제로 읽을 수 있고, 쓸 수 있고, 에러가 있는 소켓이 들어 있습니다. 이 리스트는 각기 여러분이 전달한 해당 목록의 부분집합(비어있는 것도 가능합니다)입니다.

소켓이 반환된 읽기 가능한 리스트에 있다면, 그 소켓에 대한 `recv`가 무언가를 반환하리라는 것을 알 수 있습니다. 쓰기 가능한 리스트도 마찬가지입니다. 무언가를 보낼 수 있습니다. 아마 당신이 원하는 전부는 아니겠지만, 무언가는 아무것도 아닌 것보다 낫습니다. (사실, 합리적으로 건강한 소켓은 쓰기 가능 상태로 반환될 것입니다 - 단지 네트워크 송신 버퍼 공간을 사용할 수 있음을 뜻합니다.)

“서버” 소켓이 있다면, `potential_readers` 리스트에 넣으십시오. 읽기 가능한 리스트에 등장하면, `accept`가 (거의 확실하게) 작동합니다. 다른 곳으로 `connect`하는 새 소켓을 만들었으면, `potential_writers` 리스트에 넣으십시오. 쓰기 가능한 리스트에 등장하면, 연결되었을 확률이 높습니다.

사실, `select`는 블로킹 소켓에서도 편리할 수 있습니다. 블록 할지 판단하는 한 가지 방법입니다 - 소켓은 버퍼에 무엇인가가 있으면 읽기 가능으로 반환됩니다. 그러나, 이것은 상대방이 완료했는지 아니면 단지 다른 일로 바쁜 것인지를 결정하는 문제에는 여전히 도움이 되지 않습니다.

이식성 경고: 유닉스에서, `select`는 소켓과 파일 모두에서 작동합니다. 윈도우에서 이런 시도를 하지 마십시오. 윈도우에서 `select`는 소켓에서만 작동합니다. 또한, C에서 많은 고급 소켓 옵션은 윈도우에서 다르게 동작함에 유의하십시오. 사실, 윈도우에서 저는 보통 소켓에 스레드를 사용합니다 (아주 잘 작동합니다).