Hướng dẫn Lập trình Mạng của Beej

Dùng Socket Internet

Brian “Beej Jorgensen” Hall (bản dịch tiếng Việt của Duc-Tam Nguyen)

Contents

Chapter 1

Giới thiệu

Này! Lập trình socket đang hành bạn? Đọc trang `man` mà chả hiểu ra làm sao? Bạn muốn làm mấy thứ ngầu trên mạng nhưng không có thời gian lội qua đống `struct` để biết có cần gọi `bind()` trước `connect()` không, v.v.

Mà đoán xem! Tôi đã lội qua cái đống đó rồi, và đang ngứa ngáy muốn kể cho mọi người nghe! Bạn đã đến đúng chỗ rồi. Tài liệu này sẽ giúp lập trình viên C tầm trung có đủ vũ khí để nắm được cái mớ lập trình mạng này.

Ồ và còn nữa: tôi đã theo kịp tương lai rồi (vừa kịp lúc thôi!) và đã cập nhật thêm IPv6 cho tài liệu! Đọc thôi!

1.1 Đối tượng đọc

Tài liệu này được viết theo dạng hướng dẫn, không phải tài liệu tham khảo đầy đủ. Nó phù hợp nhất với những ai mới bắt đầu với lập trình socket và đang cần chỗ bám để leo. Nó chắc chắn không phải là tài liệu *toàn diện và đầy đủ* về lập trình socket, theo bất kỳ nghĩa nào.

Nhưng hy vọng nó vừa đủ để bạn bắt đầu đọc hiểu được trang `man`... `:-)`

1.2 Nền tảng và trình biên dịch

Code trong tài liệu này được biên dịch trên máy Linux với trình biên dịch `gcc` của GNU. Tuy nhiên nó cũng biên dịch được trên hầu hết các nền tảng dùng `gcc`. Tất nhiên điều này không áp dụng nếu bạn lập trình trên Windows, xem phần về Windows bên dưới.

1.3 Trang chủ chính thức và sách in

Tài liệu gốc được đặt tại:

- `https://beej.us/guide/bgnet/`

Ở đó bạn cũng tìm thấy code ví dụ và các bản dịch sang nhiều ngôn ngữ.

Để mua bản in đóng bìa đẹp (có người gọi là "sách"), ghé vào:

- `https://beej.us/guide/url/bgbuy`

Tôi sẽ rất cảm ơn nếu bạn mua, vì nó giúp tôi duy trì lối sống viết tài liệu!

1.4 Ghi chú cho người dùng Solaris/SunOS/illumos

Khi biên dịch trên Solaris hay SunOS, bạn cần thêm một số cờ dòng lệnh để liên kết đúng thư viện. Chỉ cần thêm " `-lnsl -lsocket -lresolv` " vào cuối lệnh biên dịch, kiểu như:

```
$ cc -o server server.c -lnsl -lsocket -lresolv
```

Nếu vẫn còn báo lỗi, thử thêm `-lxnet` vào cuối. Tôi không biết chính xác cái đó làm gì, nhưng một số người cần đến nó.

Một chỗ khác có thể gặp vấn đề là khi gọi `setsockopt()`. Prototype trên Solaris khác với trên máy Linux của tôi, vì vậy thay vì:

```
int yes=1;
```

hãy dùng:

```
char yes='1';
```

Vì tôi không có máy Sun nên chưa tự kiểm tra, đây chỉ là những gì mọi người nói với tôi qua email.

1.5 Ghi chú cho người dùng Windows

Từ trước đến nay tài liệu này có truyền thống chê Windows khá nhiều, đơn giản là vì tôi không thích nó mấy. Nhưng rồi Windows và Microsoft cũng có nhiều cải thiện. Windows 10 kết hợp với WSL (xem bên dưới) tạo nên một hệ điều hành khá được. Không có gì nhiều để phàn nàn.

À, vẫn còn một chút. Ví dụ, tôi đang viết bài này (năm 2025) trên chiếc laptop 2015 từng chạy Windows 10. Cuối cùng nó chậm quá và tôi cài Linux lên. Rồi dùng từ đó đến giờ.

Còn nay lại có Windows 11 đòi hỏi phần cứng mạnh hơn Windows 10. Tôi không ưa điều đó chút nào. Hệ điều hành phải càng nhẹ càng tốt, không nên buộc bạn bỏ thêm tiền. CPU mạnh hơn là để chạy ứng dụng, không phải để chạy hệ điều hành! Và thêm vào đó, Microsoft biết bạn muốn gì, và cái bạn muốn là… quảng cáo! Đúng không? Ngay trong hệ điều hành! Bạn đang nhớ nó lắm phải không? Windows 11 đã có sẵn cho bạn rồi đó.

Vì vậy tôi vẫn khuyến khích bạn thử Linux[1], BSD[2], illumos[3] hay bất kỳ hệ Unix nào đó thay vì Windows.

Ừ mà cái bục diễn thuyết này chui vào đây lúc nào vậy?

Thôi, ai thích gì dùng nấy. Bạn dùng Windows cũng được, phần lớn nội dung trong tài liệu này vẫn áp dụng, chỉ cần sửa vài chỗ nhỏ.

Điều đầu tiên bạn nên cân nhắc là Windows Subsystem for Linux[4]. Đây cơ bản là cách cài một thứ gì đó kiểu máy ảo Linux trên Windows 10. Với nó, bạn biên dịch và chạy các chương trình trong tài liệu này nguyên vẹn không cần sửa gì.

Một lựa chọn khác là cài Cygwin[5], bộ công cụ Unix cho Windows. Nghe đồn rằng các chương trình này biên dịch được nguyên vẹn với Cygwin, nhưng tôi chưa tự thử bao giờ.

Còn nếu bạn muốn làm theo đúng kiểu Windows thuần túy, thật can đảm đấy! Và đây là việc bạn phải làm: chạy ra ngoài mua ngay một máy Unix! Không không, đùa thôi. Tôi phải thân thiện hơn với Windows ngày nay mà…

[1] https://www.linux.com/
[2] https://bsd.org/
[3] https://www.illumos.org/
[4] https://learn.microsoft.com/en-us/windows/wsl/
[5] https://cygwin.com/

Thôi được rồi. Vào việc thôi.

Đây là những gì bạn cần làm: đầu tiên, bỏ qua gần hết các file header hệ thống tôi đề cập trong tài liệu. Thay vào đó, include:

```
#include <winsock2.h>
#include <ws2tcpip.h>
```

`winsock2` là phiên bản "mới" (khoảng năm 1994) của thư viện socket trên Windows.

Đáng tiếc là nếu bạn include `windows.h`, nó tự động kéo theo header `winsock.h` cũ hơn (phiên bản 1), xung đột với `winsock2.h`! Vui thật.

Vì vậy nếu phải include `windows.h`, bạn cần định nghĩa một macro để nó *không* kéo theo header cũ:

```
#define WIN32_LEAN_AND_MEAN  // Say this...

#include <windows.h>         // And now we can include that.
#include <winsock2.h>        // And this.
```

Khoan đã! Bạn còn phải gọi `WSAStartup()` trước khi làm bất cứ điều gì với thư viện socket. Bạn truyền vào phiên bản Winsock muốn dùng (ví dụ phiên bản 2.2), rồi kiểm tra kết quả để chắc chắn phiên bản đó có sẵn.

Code trông kiểu này:

```
#include <winsock2.h>

{
    WSADATA wsaData;

    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        fprintf(stderr, "WSAStartup failed.\n");
        exit(1);
    }

    if (LOBYTE(wsaData.wVersion) != 2 ||
        HIBYTE(wsaData.wVersion) != 2)
    {
        fprintf(stderr,"Version 2.2 of Winsock not available.\n");
        WSACleanup();
        exit(2);
    }
```

Để ý lệnh gọi `WSACleanup()` ở đó. Đó là hàm bạn cần gọi khi dùng xong thư viện Winsock.

Bạn cũng cần báo cho trình biên dịch liên kết với thư viện Winsock, tên là `ws2_32.lib` cho Winsock 2. Trong VC++, vào menu `Project`, chọn `Settings...`, bấm tab `Link`, tìm ô "Object/library modules" và thêm "ws2_32.lib" vào danh sách.

Ít nhất tôi nghe vậy.

Sau khi xong xuôi, hầu hết các ví dụ trong tài liệu này đều dùng được, với vài ngoại lệ. Thứ nhất, bạn không thể dùng `close()` để đóng socket mà phải dùng `closesocket()`. Ngoài ra, `select()` chỉ hoạt động với socket descriptor, không hoạt động với file descriptor thông thường (như `0` cho `stdin`).

Ngoài ra còn có lớp socket `CSocket`. Xem tài liệu trình biên dịch của bạn để biết thêm.

Muốn biết thêm về Winsock, xem trang chính thức của Microsoft.

Cuối cùng, tôi nghe nói Windows không có system call `fork()`, mà một số ví dụ của tôi có dùng. Có thể bạn cần liên kết với thư viện POSIX nào đó, hoặc dùng `CreateProcess()` thay thế. `fork()` không nhận tham số nào, còn `CreateProcess()` nhận khoảng 48 tỷ tham số. Nếu không muốn đối mặt với điều đó, `CreateThread()` dễ nuốt hơn một chút, tiếc thay thảo luận về đa luồng thì nằm ngoài phạm vi tài liệu này rồi. Tôi chỉ viết được đến đây thôi!

Và còn một điều "cuối cùng" nữa, Steven Mitchell đã chuyển một số ví dụ[6] sang Winsock. Có thể xem thêm ở đó.

1.6 Chính sách email

Tôi thường sẵn sàng trả lời câu hỏi qua email, cứ viết thôi, nhưng tôi không đảm bảo sẽ hồi âm. Cuộc sống khá bận và đôi khi tôi thật sự không có thời gian trả lời. Khi đó tôi thường xóa email đi. Không có gì cá nhân cả, chỉ là không có thời gian cho câu trả lời chi tiết mà bạn cần thôi.

Thông thường, câu hỏi càng phức tạp thì khả năng tôi trả lời càng thấp. Nếu bạn thu hẹp vấn đề trước khi gửi, kèm theo đầy đủ thông tin liên quan (nền tảng, trình biên dịch, thông báo lỗi, và bất cứ thứ gì bạn nghĩ có thể giúp tôi debug), bạn sẽ có nhiều khả năng nhận được hồi âm hơn. Để biết thêm, đọc tài liệu của ESR: How To Ask Questions The Smart Way[7].

Nếu không nhận được hồi âm, hãy tiếp tục mày mò, cố tự tìm câu trả lời, rồi nếu vẫn bế tắc thì viết lại cho tôi kèm theo những gì bạn đã tìm được. Biết đâu thông tin đó đủ để tôi giúp được.

Dù tôi có nài nỉ bạn viết thế này hay thế kia, tôi chỉ muốn nói rằng tôi *thực sự* trân trọng mọi lời khen ngợi mà tài liệu này nhận được qua bao nhiêu năm. Đó là nguồn động lực thực sự, và nghe rằng nó được dùng vào việc tốt thì thật vui! `:-)` Cảm ơn mọi người!

1.7 Nhân bản tài liệu

Bạn hoàn toàn được phép nhân bản trang này, dù công khai hay riêng tư. Nếu bạn nhân bản công khai và muốn tôi đặt liên kết từ trang chủ, gửi cho tôi một dòng tại `beej@beej.us`.

1.8 Ghi chú cho người dịch

Nếu bạn muốn dịch tài liệu sang ngôn ngữ khác, liên hệ tôi tại `beej@beej.us` và tôi sẽ đặt liên kết đến bản dịch của bạn từ trang chủ. Bạn được phép thêm tên và thông tin liên lạc của mình vào bản dịch.

Tài liệu nguồn dùng encoding UTF-8.

Xin lưu ý các điều khoản giấy phép trong phần Bản quyền, phân phối và pháp lý bên dưới.

Nếu bạn muốn tôi lưu trữ bản dịch, cứ hỏi. Tôi cũng sẽ đặt liên kết nếu bạn tự lưu trữ. Cách nào cũng được.

1.9 Bản quyền, phân phối và pháp lý

[6] https://www.tallyhawk.net/WinsockExamples/
[7] http://www.catb.org/~esr/faqs/smart-questions.html

Chapter 2

Socket là gì?

Bạn cứ nghe nói về "socket" suốt, và chắc đang thắc mắc chính xác thì nó là cái gì. Thì nó là thế này: một cách để nói chuyện với các chương trình khác qua file descriptor chuẩn của Unix.

Cái gì?

Được rồi, chắc bạn từng nghe một hacker Unix nào đó tuyên bố, "Trời ạ, *mọi thứ* trong Unix đều là file!". Ý họ nói là khi các chương trình Unix làm I/O, chúng đọc hoặc ghi qua một file descriptor. File descriptor đơn giản là một số nguyên gắn với một file đang mở. Nhưng (điểm mấu chốt đây), "file" đó có thể là một kết nối mạng, một FIFO, một pipe, một terminal, một file thật nằm trên đĩa, hay gần như bất kỳ thứ gì khác. Mọi thứ trong Unix *đều là* file! Vì vậy khi bạn muốn nói chuyện với một chương trình khác qua Internet, bạn sẽ làm điều đó qua một file descriptor, tin đi là vừa.

"Vậy kiếm cái file descriptor cho giao tiếp mạng này ở đâu, hở ông Thông Thái?" có lẽ là câu cuối bạn định hỏi ngay lúc này, nhưng tôi vẫn sẽ trả lời: Bạn gọi đến system routine `socket()`. Nó trả về socket descriptor, và bạn giao tiếp qua nó bằng các lời gọi socket chuyên dụng `send()` và `recv()` (`man send`, `man recv`).

"Khoan đã!" có lẽ giờ bạn đang la lên. "Nếu đó là file descriptor, thì vì sao quỷ thần ơi tôi không dùng luôn `read()` và `write()` bình thường để giao tiếp qua socket?" Câu trả lời ngắn là, "Được chứ!" Câu trả lời dài hơn là, "Được, nhưng `send()` và `recv()` cho bạn nhiều quyền kiểm soát hơn đối với việc truyền dữ liệu."

Tiếp theo là gì? Thế này nhé: có đủ loại socket. Có địa chỉ DARPA Internet (Internet Socket), tên đường dẫn trên máy cục bộ (Unix Socket), địa chỉ CCITT X.25 (X.25 Socket bạn cứ yên tâm bỏ qua), và chắc còn nhiều loại khác tuỳ phiên bản Unix bạn chạy. Tài liệu này chỉ bàn đến loại đầu tiên: Internet Socket.

2.1 Hai loại internet socket

Gì cơ? Có hai loại Internet socket à? Đúng. Mà thôi, không. Tôi nói dối đấy. Có nhiều hơn, chỉ là tôi không muốn làm bạn sợ. Ở đây tôi chỉ nói về hai loại thôi. Ngoại trừ câu này, nơi tôi sẽ nói rằng "Raw Socket" cũng rất mạnh và bạn nên tìm hiểu thêm.

Thôi được rồi. Hai loại đó là gì? Một là "Stream Socket"; cái kia là "Datagram Socket", từ đây trở đi có thể được gọi là "`SOCK_STREAM`" và "`SOCK_DGRAM`" tương ứng. Datagram socket đôi khi được gọi là "connectionless socket" (socket phi kết nối). (Mặc dù chúng vẫn có thể `connect()` nếu bạn thật sự muốn. Xem `connect()` bên dưới.)

Stream socket là luồng giao tiếp hai chiều, có kết nối, và đáng tin cậy. Nếu bạn đẩy hai thứ vào socket theo thứ tự "1, 2", chúng sẽ đến đầu bên kia đúng theo thứ tự "1, 2". Và cũng không có lỗi. Tôi chắc chắn điều đó đến mức nếu ai đó dám cãi lại, tôi sẽ bịt tai lại và ngân nga *la la la la*.

Cái gì dùng stream socket? Chắc bạn nghe nói đến mấy ứng dụng `telnet` hay `ssh` rồi chứ? Chúng dùng stream socket. Mọi ký tự bạn gõ cần đến đúng theo thứ tự bạn gõ, đúng không? Ngoài ra, trình duyệt web dùng Hypertext Transfer Protocol (HTTP), giao thức này dùng stream socket để lấy trang web. Thật vậy, nếu bạn telnet vào một trang web ở port 80, gõ "`GET / HTTP/1.0`" rồi nhấn RETURN hai lần, nó sẽ quăng cả đống HTML vào mặt bạn!

> Nếu bạn không cài `telnet` và cũng không muốn cài, hoặc cái `telnet` của bạn kén chọn khi kết nối với client, tài liệu này kèm theo một chương trình giống `telnet` tên là `telnot`[a]. Nó đủ đáp ứng mọi nhu cầu trong tài liệu. (Lưu ý telnet thật ra là một giao thức mạng có đặc tả chuẩn[b], còn `telnot` thì không implement giao thức đó chút nào.)
>
> [a] https://beej.us/guide/bgnet/source/examples/telnot.c
> [b] https://tools.ietf.org/html/rfc854

Stream socket đạt được chất lượng truyền dữ liệu cao như vậy bằng cách nào? Chúng dùng một giao thức gọi là "Transmission Control Protocol", hay còn được biết đến với tên "TCP" (xem RFC 793[1] để biết thông tin cực kỳ chi tiết về TCP). TCP đảm bảo dữ liệu của bạn đến đúng thứ tự và không có lỗi. Có thể bạn đã nghe "TCP" trước đây, như là nửa ngon lành của "TCP/IP", trong đó "IP" viết tắt cho "Internet Protocol" (xem RFC 791[2]). IP chủ yếu lo việc định tuyến trên Internet và nhìn chung không chịu trách nhiệm về tính toàn vẹn dữ liệu.

Ngon. Còn Datagram socket thì sao? Vì sao chúng được gọi là connectionless? Nói chung chuyện là sao vậy? Vì sao chúng lại không đáng tin cậy? Vài sự thật cho bạn đây: nếu bạn gửi một datagram, nó *có thể* đến nơi. Nó *có thể* đến không đúng thứ tự. Nếu nó đến, dữ liệu trong packet sẽ không có lỗi.

Datagram socket cũng dùng IP để định tuyến, nhưng không dùng TCP; chúng dùng "User Datagram Protocol", hay "UDP" (xem RFC 768[3]).

Vì sao chúng là connectionless? Cơ bản là vì bạn không cần duy trì một kết nối mở như với stream socket. Bạn chỉ cần đóng gói một packet, gắn header IP với thông tin đích vào, rồi gửi đi. Không cần kết nối. Chúng thường được dùng hoặc khi TCP stack không có sẵn, hoặc khi vài packet rơi rụng đây đó không đồng nghĩa với tận thế. Ứng dụng mẫu: `tftp` (trivial file transfer protocol, em họ bé tí của FTP), `dhcpcd` (một DHCP client), game nhiều người chơi, streaming audio, gọi video, v.v.

"Khoan đã! `tftp` và `dhcpcd` được dùng để chuyển các chương trình nhị phân từ máy này sang máy khác! Dữ liệu không được phép mất nếu muốn chương trình còn chạy được sau khi đến nơi! Loại phép thuật đen tối gì vậy?"

Ồ, bạn người của tôi, `tftp` và các chương trình tương tự có giao thức riêng chạy trên UDP. Ví dụ, giao thức tftp quy định rằng với mỗi packet được gửi đi, bên nhận phải gửi lại một packet nói, "Tôi nhận được rồi!" (một packet "ACK"). Nếu bên gửi packet gốc không nhận được hồi âm trong, giả sử, năm giây, anh ta sẽ gửi lại packet cho đến khi cuối cùng nhận được ACK. Thủ tục xác nhận này rất quan trọng khi implement các ứng dụng `SOCK_DGRAM` đáng tin cậy.

Còn với các ứng dụng không cần độ tin cậy như game, audio, hay video, bạn chỉ việc mặc kệ mấy packet bị rớt, hoặc tìm cách bù trừ một cách khéo léo. (Dân chơi Quake sẽ nhận ra biểu hiện của hiện tượng này với thuật ngữ kỹ thuật: *cái lag trời đánh*. Từ "trời đánh" ở đây đại diện cho bất kỳ lời chửi thề cực kỳ tục tĩu nào.)

Vì sao lại dùng một giao thức nền không tin cậy? Hai lý do: tốc độ và tốc độ. Gửi đi rồi quên luôn thì nhanh hơn nhiều so với việc theo dõi cái gì đã đến nơi an toàn và đảm bảo thứ tự rồi đủ thứ chuyện. Nếu bạn đang gửi tin nhắn chat, TCP tuyệt vời; nếu bạn đang gửi 40 lần cập nhật vị trí mỗi giây của các người chơi trong thế giới game, có lẽ một hai cái bị rớt cũng không sao lắm, và UDP là lựa chọn tốt.

[1] https://tools.ietf.org/html/rfc793
[2] https://tools.ietf.org/html/rfc791
[3] https://tools.ietf.org/html/rfc768

2.2 Mấy thứ thấp cấp và lý thuyết mạng

Vì tôi vừa nhắc đến việc các giao thức xếp lớp lên nhau, đã đến lúc nói về cách mạng thật sự hoạt động, và chỉ cho bạn xem vài ví dụ về cách `SOCK_DGRAM` packet được dựng lên. Thực tế thì bạn có thể bỏ qua phần này. Tuy nhiên nó là kiến thức nền tốt để có.

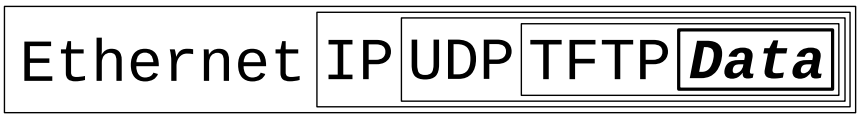

Figure 2.1: Data Encapsulation.

Này các cháu, đã đến lúc học về *Data Encapsulation*! Cái này rất rất quan trọng. Quan trọng đến mức bạn có thể sẽ được học nó nếu chọn môn mạng ở đây, trường Chico State `;-)`. Cơ bản, nó nói như sau: một packet được sinh ra, packet được bọc ("encapsulated") vào một header (và hiếm khi là một footer) bởi giao thức đầu tiên (ví dụ giao thức TFTP), rồi toàn bộ (kèm theo cả header TFTP bên trong) lại được bọc tiếp bởi giao thức kế tiếp (ví dụ UDP), rồi lại được bọc bởi cái tiếp nữa (IP), rồi cuối cùng bởi giao thức ở tầng phần cứng (vật lý) (ví dụ Ethernet).

Khi máy khác nhận được packet, phần cứng bóc header Ethernet ra, kernel bóc header IP và UDP ra, chương trình TFTP bóc header TFTP ra, và cuối cùng nó có được dữ liệu.

Giờ thì tôi mới có thể nói về cái *Layered Network Model* khét tiếng (hay còn gọi là "ISO/OSI"). Mô hình mạng này mô tả một hệ thống chức năng mạng có nhiều ưu điểm so với các mô hình khác. Ví dụ, bạn có thể viết các chương trình socket giống hệt nhau mà chẳng cần quan tâm dữ liệu được truyền đi về mặt vật lý thế nào (serial, thin Ethernet, AUI, gì cũng được), vì các chương trình ở tầng thấp hơn lo chuyện đó cho bạn. Phần cứng mạng thật sự và topology hoàn toàn trong suốt với lập trình viên socket.

Không dông dài nữa, tôi sẽ trình bày các tầng của mô hình đầy đủ. Nhớ cái này cho kỳ thi môn mạng nhé:

- Application
- Presentation
- Session
- Transport
- Network
- Data Link
- Physical

Physical Layer là phần cứng (serial, Ethernet, v.v.). Application Layer thì cách xa tầng vật lý gần như xa hết mức bạn có thể tưởng tượng, đó là nơi người dùng tương tác với mạng.

Mô hình này chung chung đến mức bạn có thể dùng nó làm sách hướng dẫn sửa xe hơi nếu thật sự muốn. Một mô hình xếp lớp nhất quán hơn với Unix có thể là:

- Application Layer (*telnet, ftp, v.v.*)
- Host-to-Host Transport Layer (*TCP, UDP*)
- Internet Layer (*IP và định tuyến*)
- Network Access Layer (*Ethernet, wi-fi, hay gì đó*)

Đến đây, chắc bạn đã thấy các tầng này tương ứng với việc đóng gói dữ liệu gốc như thế nào rồi.

Thấy cần bao nhiêu công đoạn để xây dựng một packet đơn giản chưa? Trời ạ! Và bạn phải tự gõ các header packet bằng "`cat`"! Đùa thôi. Với stream socket, tất cả việc bạn cần làm là `send()` dữ liệu ra. Với datagram socket, bạn chỉ cần đóng gói packet theo cách của mình rồi `sendto()` đi. Kernel xây dựng Transport Layer và Internet Layer giúp bạn, còn phần cứng lo Network Access Layer. Ôi, công nghệ hiện đại.

Thế là kết thúc chuyến ghé ngắn ngủi của chúng ta vào lý thuyết mạng. À đúng rồi, tôi quên nói với bạn tất cả những gì tôi muốn nói về định tuyến: không gì cả! Đúng vậy, tôi sẽ không nói về nó chút nào. Router

bóc packet ra đến header IP, tra bảng định tuyến, *bla bla bla.* Xem RFC về IP[4] nếu bạn thật sự thật sự quan tâm. Nếu bạn không bao giờ học về nó, thì cũng không sao, bạn vẫn sống được.

[4]https://tools.ietf.org/html/rfc791

Chapter 3

Địa chỉ IP, `struct`, và xử lý dữ liệu

Đến đoạn mà chúng ta được ngồi nói chuyện code cho khác đi chút.

Nhưng trước tiên, nói thêm tí về phần không phải code nhé! Tuyệt vời! Tôi muốn nói một chút về địa chỉ IP và port để chúng ta gọn được phần đó. Rồi tới chuyện sockets API lưu trữ và thao tác với địa chỉ IP cùng dữ liệu khác ra sao.

3.1 Địa chỉ IP, phiên bản 4 và 6

Ngày xưa ơi là xưa, hồi Ben Kenobi còn được gọi là Obi Wan Kenobi, có một hệ thống định tuyến mạng tuyệt vời tên là The Internet Protocol Version 4, còn gọi là IPv4. Địa chỉ của nó gồm bốn byte (hay còn gọi là bốn "octet"), và thường được viết dưới dạng "chấm và số", kiểu như: `192.0.2.111`.

Chắc bạn cũng thấy nó đâu đó rồi.

Thực tế là tính đến thời điểm viết bài này, gần như mọi site trên Internet đều dùng IPv4.

Mọi người, kể cả Obi Wan, đều vui vẻ. Mọi thứ đều ổn, cho đến khi một người hay dội nước lạnh tên Vint Cerf cảnh báo tất cả rằng chúng ta sắp cạn địa chỉ IPv4!

(Ngoài việc cảnh báo mọi người về Ngày Tận Thế IPv4 Đang Tới Trong Khói Lửa Đau Thương, Vint Cerf[1] còn nổi tiếng là Cha Đẻ Của Internet. Nên thực sự tôi cũng không ở vị trí đủ tầm để nghi ngờ phán đoán của ông.)

Cạn địa chỉ? Sao có thể thế được? Ý tôi là, có tới mấy tỷ địa chỉ IP trong một địa chỉ IPv4 32-bit. Chả lẽ thực sự có mấy tỷ máy tính ngoài kia?

Có.

Thêm nữa, lúc ban đầu, khi còn rất ít máy tính và ai cũng nghĩ một tỷ là con số lớn không tưởng, một số tổ chức lớn đã được cấp hào phóng hàng triệu địa chỉ IP để dùng riêng. (Như Xerox, MIT, Ford, HP, IBM, GE, AT&T, và một công ty nhỏ bé gọi là Apple, chỉ kể vài cái.)

Thực ra, nếu không có mấy giải pháp chữa cháy, chúng ta đã cạn từ đời nào rồi.

Nhưng giờ chúng ta đang ở cái thời mà ai cũng nói mỗi con người sẽ có một địa chỉ IP, mỗi máy tính, mỗi cái máy tính bỏ túi, mỗi cái điện thoại, mỗi cái đồng hồ đỗ xe, và (tại sao không) mỗi con chó con nữa.

Và thế là IPv6 ra đời. Vì Vint Cerf chắc là bất tử (kể cả phần xác có ra đi, lạy trời đừng, thì chắc ông cũng đã tồn tại dưới dạng một chương trình ELIZA[2] siêu thông minh nào đó lang thang trong Internet2), không

[1] https://en.wikipedia.org/wiki/Vint_Cerf
[2] https://en.wikipedia.org/wiki/ELIZA

ai muốn nghe ông nói lại "tôi đã bảo rồi" nếu chúng ta lại hết địa chỉ trong phiên bản tiếp theo của Internet Protocol.

Điều này gợi cho bạn cái gì?

Là chúng ta cần *rất nhiều* địa chỉ hơn. Không phải gấp đôi, không phải gấp một tỷ lần, không phải gấp nghìn nghìn tỷ lần, mà *nhiều gấp 79 TRIỆU TỶ TỶ lần số địa chỉ khả dĩ!* Xem đứa nào đòi cạn nữa nào!

Bạn sẽ hỏi, "Beej ơi, thật không? Tôi có mọi lý do để không tin mấy con số khổng lồ." Ờ, khác biệt giữa 32 bit và 128 bit nghe cũng không ghê gớm; chỉ thêm 96 bit thôi mà phải không? Nhưng nhớ là ta đang nói về lũy thừa: 32 bit biểu diễn khoảng 4 tỷ số (2^{32}), còn 128 bit biểu diễn khoảng 340 nghìn tỷ nghìn tỷ nghìn tỷ số (thật đấy, 2^{128}). Cỡ bằng một triệu Internet IPv4 cho *mỗi ngôi sao trong Vũ Trụ.*

Quên luôn cái kiểu chấm và số của IPv4 đi; giờ chúng ta có biểu diễn dạng hexa, mỗi cụm hai byte cách nhau bởi dấu hai chấm, kiểu như:

```
2001:0db8:c9d2:aee5:73e3:934a:a5ae:9551
```

Chưa hết! Rất nhiều lần bạn sẽ gặp địa chỉ IP có nhiều số 0, và bạn có thể nén chúng lại giữa hai dấu hai chấm. Bạn cũng có thể bỏ các số 0 đầu của mỗi cặp byte. Ví dụ, từng cặp địa chỉ sau là tương đương:

```
2001:0db8:c9d2:0012:0000:0000:0000:0051
2001:db8:c9d2:12::51

2001:0db8:ab00:0000:0000:0000:0000:0000
2001:db8:ab00::

0000:0000:0000:0000:0000:0000:0000:0001
::1
```

Địa chỉ `::1` là *địa chỉ loopback*. Nó luôn có nghĩa là "cái máy tôi đang chạy ngay bây giờ". Trong IPv4, địa chỉ loopback là `127.0.0.1`.

Cuối cùng, có một chế độ tương thích IPv4 dành cho địa chỉ IPv6 mà bạn có thể bắt gặp. Ví dụ muốn biểu diễn địa chỉ IPv4 `192.0.2.33` dưới dạng địa chỉ IPv6, bạn viết thế này: "`::ffff:192.0.2.33`".

Đang vui ra trò đấy.

Thực ra vui đến mức mấy Người Sáng Tạo Ra IPv6 đã lơ là bỏ đi cả nghìn tỷ nghìn tỷ địa chỉ để dành cho các mục đích dự trữ, nhưng nói thật, chúng ta có quá trời địa chỉ, ai thèm đếm làm gì nữa? Vẫn còn dư đủ cho mỗi người đàn ông, phụ nữ, trẻ em, chó con, và đồng hồ đỗ xe trên mỗi hành tinh trong thiên hà. Và tin tôi đi, mỗi hành tinh trong thiên hà đều có đồng hồ đỗ xe. Bạn biết điều đó là thật mà.

3.1.1 Subnet

Vì lý do tổ chức, đôi khi sẽ tiện nếu ta tuyên bố rằng "phần đầu của địa chỉ IP này tính đến bit đây là *phần network* của địa chỉ IP, còn phần còn lại là *phần host*".

Ví dụ, với IPv4, bạn có `192.0.2.12`, và ta có thể nói ba byte đầu là network còn byte cuối là host. Hay nói cách khác, ta đang nói về host `12` trên network `192.0.2.0` (để ý cách ta zero byte host).

Giờ đến phần thông tin lỗi thời hơn! Sẵn sàng chưa? Thời Thượng Cổ, có các "class" subnet, trong đó một, hai, hoặc ba byte đầu của địa chỉ là phần network. Nếu bạn may mắn có một byte cho network và ba byte cho host, bạn có tới 24 bit host trên network của mình (khoảng 16 triệu). Đó là network "Class A". Ngược lại là "Class C", với ba byte network và một byte host (256 host, trừ đi vài cái bị dự trữ).

Nên như bạn thấy, có rất ít Class A, một đống Class C, và một ít Class B ở giữa.

Phần network của địa chỉ IP được mô tả bằng thứ gọi là *netmask*, bạn AND bit với địa chỉ IP để lấy ra số network. Netmask thường trông kiểu như `255.255.255.0`. (Ví dụ với netmask đó, nếu IP của bạn là `192.0.2.12`, thì network của bạn là `192.0.2.12` AND `255.255.255.0` cho ra `192.0.2.0`.)

Tiếc là, hóa ra kiểu này không đủ tinh gọn cho nhu cầu cuối cùng của Internet; chúng ta cạn network Class C khá nhanh, và Class A thì thôi khỏi hỏi. Để khắc phục, Các Thế Lực Có Quyền Năng đã cho phép netmask dùng số bit tuỳ ý, không chỉ 8, 16, hay 24. Vậy nên bạn có thể có netmask kiểu `255.255.255.252`, nghĩa là 30 bit network và 2 bit host, cho phép bốn host trên network. (Lưu ý netmask *LUÔN* là một dãy bit 1 theo sau là một dãy bit 0.)

Nhưng dùng chuỗi số dài ngoằng kiểu `255.192.0.0` làm netmask thì cũng hơi bất tiện. Thứ nhất, người ta không có khái niệm trực quan đó là bao nhiêu bit, thứ hai, nó không gọn tí nào. Nên Kiểu Mới ra đời, và nó đẹp hơn nhiều. Bạn chỉ cần đặt một dấu gạch chéo sau địa chỉ IP, rồi theo sau là số bit network ở dạng thập phân. Như thế này: `192.0.2.12/30`.

Hoặc với IPv6, kiểu thế này: `2001:db8::/32` hay `2001:db8:5413:4028::9db9/64`.

3.1.2 Số port

Nếu bạn còn nhớ, tôi đã giới thiệu Mô hình Mạng Phân Lớp trong đó Internet Layer (IP) được tách khỏi Host-to-Host Transport Layer (TCP và UDP). Lướt lại đoạn đó trước khi qua đoạn tiếp theo nhé.

Hoá ra ngoài địa chỉ IP (tầng IP dùng), còn một địa chỉ nữa được TCP (stream socket) dùng, và tiện thể cả UDP (datagram socket) cũng dùng. Đó là *số port*. Nó là một số 16-bit, giống như địa chỉ cục bộ cho một kết nối.

Hãy nghĩ địa chỉ IP như địa chỉ đường của một khách sạn, và số port như số phòng. Cũng là một phép so sánh được; có khi lúc khác tôi sẽ nghĩ ra một phép liên quan đến ngành công nghiệp ô tô.

Giả sử bạn muốn có một máy tính vừa xử lý mail đến VÀ dịch vụ web, làm sao phân biệt hai dịch vụ trên một máy chỉ có một địa chỉ IP?

Ồ, các dịch vụ khác nhau trên Internet có các số port well-known khác nhau. Bạn có thể xem hết trong Bảng Port Khổng Lồ Của IANA[3] hoặc, nếu bạn dùng Unix, trong file `/etc/services`. HTTP (web) là port 80, telnet là port 23, SMTP là port 25, game DOOM[4] dùng port 666, vân vân. Port dưới 1024 thường được coi là đặc biệt, và thường đòi hỏi quyền đặc biệt từ OS để dùng.

Và tạm vậy thôi!

3.2 Byte Order

Theo Lệnh Của Vương Quốc! Sẽ có hai thứ tự byte, từ nay về sau được biết tới với tên gọi Chuối Lè và Hoành Tráng!

Tôi đùa thôi, nhưng thực sự một trong hai cái tốt hơn cái kia. `:-)`

Thật sự chả có cách nào nhẹ nhàng để nói, nên tôi cứ phun ra thẳng: máy tính của bạn có thể đã đang lưu byte ngược chiều sau lưng bạn. Tôi biết! Chả ai muốn phải nói ra.

Vấn đề là, mọi người trong thế giới Internet đã thống nhất chung rằng nếu bạn muốn biểu diễn số hex hai byte, chẳng hạn `b34f`, bạn sẽ lưu nó thành hai byte liên tiếp, `b3` rồi tới `4f`. Hợp lý, và như Wilford Brimley[5] sẽ nói với bạn, đây là Cách Làm Đúng Đắn. Số này, với đầu lớn đứng trước, được gọi là *Big-Endian*.

Khổ nỗi, *vài* máy tính rải rác đây đó trên thế giới, cụ thể là những máy chạy vi xử lý Intel hoặc tương thích Intel, lưu byte theo kiểu đảo ngược, nên `b34f` sẽ được lưu trong bộ nhớ dưới dạng hai byte liên tiếp `4f` rồi `b3`. Cách lưu này gọi là *Little-Endian*.

[3] https://www.iana.org/assignments/port-numbers
[4] https://en.wikipedia.org/wiki/Doom_%281993_video_game%29
[5] https://en.wikipedia.org/wiki/Wilford_Brimley

Nhưng khoan, tôi chưa xong chuyện thuật ngữ! Cái *Big-Endian* tỉnh táo hơn còn được gọi là *Network Byte Order* vì đó là thứ tự mà dân mạng chúng tôi khoái.

Máy của bạn lưu số theo *Host Byte Order*. Nếu là Intel 80x86, Host Byte Order là Little-Endian. Nếu là Motorola 68k, Host Byte Order là Big-Endian. Nếu là PowerPC, Host Byte Order là... ờ, tuỳ!

Nhiều lúc lúc xây gói tin hoặc điền struct dữ liệu, bạn sẽ cần chắc chắn các số hai và bốn byte của mình ở dạng Network Byte Order. Nhưng làm sao làm được điều đó nếu bạn không biết Host Byte Order gốc là gì?

Tin vui! Bạn cứ mặc định là Host Byte Order không đúng, rồi cứ đưa giá trị qua một hàm để chuyển về Network Byte Order. Hàm sẽ làm phép màu chuyển đổi nếu cần, và như vậy code của bạn khả chuyển giữa các máy với endianness khác nhau.

Được rồi. Có hai kiểu số bạn có thể chuyển đổi: `short` (hai byte) và `long` (bốn byte). Các hàm này cũng chạy được với biến thể `unsigned`. Giả sử bạn muốn chuyển một `short` từ Host Byte Order sang Network Byte Order. Bắt đầu bằng "h" cho "host", nối thêm "to", rồi "n" cho "network", và "s" cho "short": h-to-n-s, hay `htons()` (đọc: "Host to Network Short").

Đơn giản đến mức gần như quá đà...

Bạn có thể dùng mọi kết hợp của "n", "h", "s", và "l" mà bạn muốn, không tính mấy cái ngớ ngẩn. Ví dụ, KHÔNG có hàm `stolh()` ("Short to Long Host"), ít nhất là không có ở bữa tiệc này. Nhưng có:

Hàm	Mô tả
`htons()`	`h`ost `to` `n`etwork `s`hort
`htonl()`	`h`ost `to` `n`etwork `l`ong
`ntohs()`	`n`etwork `to` `h`ost `s`hort
`ntohl()`	`n`etwork `to` `h`ost `l`ong

Nói chung, bạn sẽ muốn chuyển các số sang Network Byte Order trước khi chúng ra đường dây, và chuyển về Host Byte Order khi chúng vào từ đường dây.

Sockets API không có biến thể 64-bit chuẩn, nhưng tôi có nói về các lựa chọn khác trong trang tham khảo `htons()`. Còn nếu bạn muốn làm việc với số thực dấu chấm động, xem phần Serialization, ở tít phía dưới.

Cứ coi các số trong tài liệu này là ở dạng Host Byte Order trừ khi tôi nói khác.

3.3 `struct`

Ồ, cuối cùng cũng tới. Tới lúc nói chuyện lập trình. Trong phần này, tôi sẽ giới thiệu các kiểu dữ liệu được sockets interface dùng, vì một số trong đó đúng là khó nhằn.

Đầu tiên là cái dễ: socket descriptor. Một socket descriptor là kiểu sau:

```
int
```

Chỉ là `int` bình thường.

Từ đây đi bắt đầu lạ hơn, nên cứ đọc tiếp và kiên nhẫn với tôi tí.

Struct đầu tiên của tôi™, `struct addrinfo`. Struct này là phát minh tương đối gần đây, dùng để chuẩn bị các struct địa chỉ socket cho các lần dùng sau. Nó cũng dùng trong tra tên máy và tra tên dịch vụ. Chỗ đó

sẽ dễ hiểu hơn khi chúng ta tới phần dùng thực tế, nhưng tạm biết rằng đây là một trong những thứ đầu tiên bạn gọi khi tạo kết nối.

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any"
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname

    struct addrinfo *ai_next;      // linked list, next node
};
```

Bạn sẽ điền struct này một chút, rồi gọi `getaddrinfo()`. Nó sẽ trả về con trỏ tới một linked list mới của các struct này đã được điền sẵn mọi thứ bạn cần.

Bạn có thể ép nó dùng IPv4 hoặc IPv6 bằng trường `ai_family`, hoặc để `AF_UNSPEC` để dùng cái nào cũng được. Như vậy tiện vì code của bạn có thể bất kể phiên bản IP.

Để ý rằng đây là linked list: `ai_next` trỏ tới phần tử kế tiếp, có thể có nhiều kết quả để bạn chọn. Tôi sẽ dùng kết quả đầu tiên chạy được, nhưng bạn có thể có nhu cầu kinh doanh khác; biết gì đâu mà nói, ông ơi!

Bạn sẽ thấy trường `ai_addr` trong `struct addrinfo` là con trỏ tới `struct sockaddr`. Đây là chỗ ta bắt đầu đi vào chi tiết cặn kẽ bên trong một struct địa chỉ IP.

Bạn thường không cần ghi trực tiếp vào những struct này; đa số trường hợp, một cuộc gọi tới `getaddrinfo()` để điền `struct addrinfo` giúp bạn là đủ. Tuy nhiên, bạn *sẽ* phải ngó vào bên trong các `struct` này để lấy các giá trị ra, nên tôi giới thiệu chúng ở đây.

(Thêm nữa, mọi code viết trước khi `struct addrinfo` ra đời đều đóng gói đống này bằng tay, nên bạn sẽ thấy nhiều code IPv4 ngoài đời làm đúng y vậy. Kiểu, trong các phiên bản cũ của tài liệu này chẳng hạn.)

Một số `struct` là IPv4, một số là IPv6, và một số cả hai. Tôi sẽ ghi chú cái nào là cái nào.

Dù sao thì, `struct sockaddr` giữ thông tin địa chỉ socket cho nhiều kiểu socket.

```
struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes of protocol address
};
```

`sa_family` có thể là nhiều thứ, nhưng với mọi thứ ta làm trong tài liệu này nó sẽ là `AF_INET` (IPv4) hoặc `AF_INET6` (IPv6). `sa_data` chứa địa chỉ đích và số port cho socket. Cái này khá khó chịu vì bạn chẳng muốn tỉ mẩn đóng gói địa chỉ vào `sa_data` bằng tay.

Để xử lý `struct sockaddr`, các lập trình viên tạo ra một cấu trúc song song: `struct sockaddr_in` ("in" cho "Internet") dùng cho IPv4.

Và *đây là đoạn quan trọng*: một con trỏ tới `struct sockaddr_in` có thể ép kiểu thành con trỏ tới `struct sockaddr` và ngược lại. Nên dù `connect()` cần `struct sockaddr*`, bạn vẫn cứ dùng `struct sockaddr_in` và ép kiểu ở phút cuối!

```
// (Chỉ IPv4, xem struct sockaddr_in6 cho IPv6)

struct sockaddr_in {
    short int          sin_family;  // Address family, AF_INET
    unsigned short int sin_port;    // Port number
    struct in_addr     sin_addr;    // Internet address
    unsigned char      sin_zero[8]; // Same size as struct sockaddr
};
```

Struct này giúp tham chiếu các thành phần của địa chỉ socket dễ dàng. Để ý rằng `sin_zero` (được đưa vào để đệm struct cho đủ chiều dài của `struct sockaddr`) nên được set toàn bộ về 0 bằng hàm `memset()`. Cũng để ý rằng `sin_family` tương ứng với `sa_family` trong `struct sockaddr` và nên được set là "`AF_INET`". Cuối cùng, `sin_port` phải ở *Network Byte Order* (bằng cách dùng `htons()` !)

Đào sâu thêm! Bạn thấy trường `sin_addr` là một `struct in_addr`. Cái gì đây? Ồ, không định kịch tính quá, nhưng đây là một trong những union đáng sợ nhất mọi thời đại:

```
// (Chỉ IPv4, xem struct in6_addr cho IPv6)

// Địa chỉ Internet (là một struct vì lý do lịch sử)
struct in_addr {
    uint32_t s_addr; // đây là int 32-bit (4 byte)
};
```

Chà! Ồ, nó *từng* là union, nhưng giờ có vẻ cái thời đó đã qua. May phúc. Vậy nếu bạn khai báo `ina` là `struct sockaddr_in`, thì `ina.sin_addr.s_addr` tham chiếu tới địa chỉ IP 4-byte (ở Network Byte Order). Lưu ý rằng kể cả nếu hệ thống của bạn vẫn còn dùng cái union trời đánh cho `struct in_addr`, bạn vẫn có thể tham chiếu địa chỉ IP 4-byte y hệt như tôi làm ở trên (nhờ mấy `#define`).

Còn IPv6 thì sao? Có các `struct` tương tự:

```
// (Chỉ IPv6, xem struct sockaddr_in và struct in_addr cho IPv4)

struct sockaddr_in6 {
    u_int16_t       sin6_family;   // address family, AF_INET6
    u_int16_t       sin6_port;     // port, Network Byte Order
    u_int32_t       sin6_flowinfo; // IPv6 flow information
    struct in6_addr sin6_addr;     // IPv6 address
    u_int32_t       sin6_scope_id; // Scope ID
};

struct in6_addr {
    unsigned char   s6_addr[16];   // IPv6 address
};
```

Để ý rằng IPv6 có một địa chỉ IPv6 và một số port, y như IPv4 có địa chỉ IPv4 và số port.

Cũng để ý là tôi sẽ chưa nói về trường IPv6 flow information hay Scope ID ngay bây giờ... đây chỉ là tài liệu khởi động thôi. `:-)`

Cuối cùng nhưng không kém phần quan trọng, đây là thêm một struct đơn giản, `struct sockaddr_storage`, được thiết kế đủ to để chứa cả struct IPv4 và IPv6. Bạn thấy đó, với một số lời gọi, đôi khi bạn không biết

trước nó sẽ điền `struct sockaddr` của bạn bằng địa chỉ IPv4 hay IPv6. Nên bạn truyền vào cấu trúc song song này, rất giống `struct sockaddr` nhưng to hơn, rồi ép kiểu về kiểu bạn cần:

```
struct sockaddr_storage {
    sa_family_t  ss_family;     // address family

    // tất cả dưới đây là padding, tuỳ implementation, bỏ qua:
    char      __ss_pad1[_SS_PAD1SIZE];
    int64_t   __ss_align;
    char      __ss_pad2[_SS_PAD2SIZE];
};
```

Điều quan trọng là bạn có thể nhìn address family trong trường `ss_family`, kiểm tra xem nó là `AF_INET` hay `AF_INET6` (cho IPv4 hay IPv6). Rồi bạn có thể ép kiểu nó về `struct sockaddr_in` hay `struct sockaddr_in6` nếu muốn.

3.4 Địa chỉ IP, Phần Hai

May cho bạn, có cả đống hàm cho phép bạn thao tác với địa chỉ IP. Không cần mày mò tính bằng tay rồi nhét vào một `long` bằng toán tử `<<`.

Đầu tiên, giả sử bạn có một `struct sockaddr_in ina`, và bạn có một địa chỉ IP "`10.12.110.57`" hoặc "`2001:db8:63b3:1::3490`" mà bạn muốn lưu vào đó. Hàm bạn muốn dùng, `inet_pton()`, chuyển một địa chỉ IP ở dạng số-và-dấu-chấm thành `struct in_addr` hoặc `struct in6_addr` tuỳ theo bạn chỉ định `AF_INET` hay `AF_INET6`. ("`pton`" là viết tắt của "presentation to network", bạn có thể gọi là "printable to network" cho dễ nhớ.) Chuyển đổi có thể thực hiện như sau cho IPv4 và IPv6:

```
struct sockaddr_in sa;   // IPv4
struct sockaddr_in6 sa6; // IPv6

inet_pton(AF_INET, "10.12.110.57", &(sa.sin_addr));
inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr));
```

(Ghi chú nhanh: cách cũ dùng một hàm tên `inet_addr()` hoặc một hàm khác tên `inet_aton()`; mấy cái này giờ lỗi thời và không chạy với IPv6.)

Đoạn code ở trên không vững chắc lắm vì không có kiểm tra lỗi. Đấy, `inet_pton()` trả về `-1` khi lỗi, hoặc `0` nếu địa chỉ bị hỏng. Nên hãy kiểm tra chắc chắn kết quả lớn hơn 0 trước khi dùng!

Rồi, giờ bạn có thể chuyển địa chỉ IP dạng chuỗi sang dạng nhị phân. Còn chiều ngược lại thì sao? Nếu bạn có `struct in_addr` và bạn muốn in nó ra dạng số-và-dấu-chấm? (Hoặc `struct in6_addr` mà bạn muốn dạng, ờ, "hex-và-dấu-hai-chấm".) Trong trường hợp này, bạn muốn dùng hàm `inet_ntop()` ("ntop" nghĩa là "network to presentation", bạn có thể gọi là "network to printable" cho dễ nhớ), kiểu như:

```
// IPv4:

char ip4[INET_ADDRSTRLEN];  // chỗ chứa chuỗi IPv4
struct sockaddr_in sa;      // giả sử nó đã được nạp gì đó rồi

inet_ntop(AF_INET, &(sa.sin_addr), ip4, INET_ADDRSTRLEN);
```

```
printf("The IPv4 address is: %s\n", ip4);


// IPv6:

char ip6[INET6_ADDRSTRLEN]; // chỗ chứa chuỗi IPv6
struct sockaddr_in6 sa6;    // giả sử nó đã được nạp gì đó rồi

inet_ntop(AF_INET6, &(sa6.sin6_addr), ip6, INET6_ADDRSTRLEN);

printf("The address is: %s\n", ip6);
```

Khi gọi, bạn sẽ truyền loại địa chỉ (IPv4 hoặc IPv6), địa chỉ, con trỏ tới chuỗi để chứa kết quả, và độ dài tối đa của chuỗi đó. (Có hai macro tiện lợi giữ kích thước chuỗi bạn cần để chứa địa chỉ IPv4 hoặc IPv6 lớn nhất: `INET_ADDRSTRLEN` và `INET6_ADDRSTRLEN`.)

(Thêm một ghi chú nhanh nữa nhắc lại cách cũ: hàm lịch sử để làm chuyển đổi này là `inet_ntoa()`. Nó cũng lỗi thời và không chạy với IPv6.)

Cuối cùng, các hàm này chỉ chạy với địa chỉ IP dạng số, chúng không làm DNS lookup trên nameserver với hostname như "`www.example.com`". Bạn sẽ dùng `getaddrinfo()` để làm việc đó, như bạn sẽ thấy sau.

3.4.1 Mạng Riêng (Hoặc Mạng Bị Ngắt Kết Nối)

Nhiều nơi có firewall giấu network của họ khỏi phần còn lại của thế giới để tự bảo vệ. Và nhiều khi, firewall còn dịch địa chỉ IP "nội bộ" thành địa chỉ IP "bên ngoài" (cái mà mọi người khác trên thế giới biết) bằng một quá trình gọi là *Network Address Translation*, hay NAT.

Đang thấy hồi hộp chưa? "Anh đang dẫn tôi đi đâu với mớ thứ kỳ quặc này?"

Ổ, thư giãn đi, mua cho mình một ly không cồn (hoặc có cồn), vì với người mới, bạn còn chả cần lo về NAT, vì nó được làm trong suốt cho bạn. Nhưng tôi muốn nói về network sau firewall phòng trường hợp bạn bắt đầu lú lẫn vì những con số network bạn nhìn thấy.

Chẳng hạn, tôi có firewall ở nhà. Tôi được công ty DSL cấp hai địa chỉ IPv4 tĩnh, vậy mà tôi có bảy máy trong mạng. Sao có thể? Hai máy không thể chia sẻ cùng một địa chỉ IP, không thì dữ liệu biết đi về máy nào!

Câu trả lời: chúng không chia sẻ cùng địa chỉ IP. Chúng đang ở trong một mạng riêng với 24 triệu địa chỉ IP được cấp. Chúng đều của riêng tôi. Ổ, tất cả của riêng tôi, ít ra ai đó bên ngoài nhìn vào thì thấy vậy. Đây là chuyện đang xảy ra:

Nếu tôi đăng nhập vào một máy từ xa, nó báo rằng tôi đang đăng nhập từ 192.0.2.33, đó là địa chỉ IP công khai mà ISP cấp cho tôi. Nhưng nếu tôi hỏi máy ở nhà địa chỉ IP của nó, nó trả lời 10.0.0.5. Ai đang dịch địa chỉ IP từ cái này sang cái kia? Đúng rồi, firewall đấy! Nó đang làm NAT!

`10.x.x.x` là một trong vài network được dự trữ, chỉ dùng trên các mạng hoàn toàn tách biệt, hoặc các mạng ở sau firewall. Chi tiết về các số network riêng nào có sẵn cho bạn dùng được nêu trong RFC 1918[6], nhưng vài cái thường thấy là `10.x.x.x` và `192.168.x.x`, trong đó `x` là 0 đến 255 thường thế. Ít gặp hơn là `172.y.x.x`, trong đó `y` chạy từ 16 đến 31.

Các mạng sau firewall NAT không *cần* phải thuộc một trong những mạng dự trữ này, nhưng thường là vậy.

(Chuyện vui! Địa chỉ IP bên ngoài của tôi thực ra không phải là `192.0.2.33`. Mạng `192.0.2.x` được dự trữ để làm địa chỉ IP "thật" giả tưởng cho dùng trong tài liệu, đúng y như tài liệu này! Ghê chưa!)

[6] https://tools.ietf.org/html/rfc1918

IPv6 cũng có mạng riêng, theo một nghĩa nào đó. Chúng sẽ bắt đầu bằng `fdXX:` (hoặc có thể trong tương lai `fcXX:`), theo RFC 4193[7]. Nhưng NAT và IPv6 nhìn chung không đi với nhau (trừ khi bạn làm cái trò gateway IPv6 sang IPv4 vốn vượt quá phạm vi tài liệu này). Về lý thuyết, bạn sẽ có quá trời địa chỉ dùng đến mức không cần tới NAT nữa. Nhưng nếu bạn muốn cấp địa chỉ cho chính mình trên một mạng không đi ra ngoài, đây là cách làm.

[7] https://tools.ietf.org/html/rfc4193

Chapter 4

Từ IPv4 nhảy sang IPv6

Nhưng tôi chỉ muốn biết phải đổi gì trong code để nó chạy được với IPv6! Nói luôn đi!

Được! Được!

Gần như mọi thứ ở đây đều là cái tôi đã nói ở phía trên, nhưng đây là phiên bản ngắn dành cho người không đủ kiên nhẫn. (Tất nhiên, còn nhiều hơn thế, nhưng đây là những gì áp dụng được trong phạm vi tài liệu này.)

1. Đầu tiên, cố gắng dùng `getaddrinfo()` để lấy toàn bộ thông tin `struct sockaddr`, thay vì đóng gói struct bằng tay. Làm vậy sẽ giữ cho code của bạn bất kể phiên bản IP, và cắt gọn được kha khá bước phía sau.
2. Chỗ nào bạn thấy mình đang hard-code thứ gì liên quan đến phiên bản IP, cố gắng gói lại trong một hàm trợ giúp.
3. Đổi `AF_INET` thành `AF_INET6`.
4. Đổi `PF_INET` thành `PF_INET6`.
5. Đổi các phép gán `INADDR_ANY` thành phép gán `in6addr_any`, có hơi khác một chút:

   ```
   struct sockaddr_in sa;
   struct sockaddr_in6 sa6;

   sa.sin_addr.s_addr = INADDR_ANY;  // use my IPv4 address
   sa6.sin6_addr = in6addr_any; // use my IPv6 address
   ```

   Ngoài ra, giá trị `IN6ADDR_ANY_INIT` có thể dùng như một initializer khi khai báo `struct in6_addr`, như thế này:

   ```
   struct in6_addr ia6 = IN6ADDR_ANY_INIT;
   ```

6. Thay vì `struct sockaddr_in`, dùng `struct sockaddr_in6`, nhớ thêm "6" vào tên trường nếu cần (xem `struct`s ở trên). Không có trường `sin6_zero`.
7. Thay vì `struct in_addr`, dùng `struct in6_addr`, nhớ thêm "6" vào tên trường nếu cần (xem `struct`s ở trên).
8. Thay vì `inet_aton()` hoặc `inet_addr()`, dùng `inet_pton()`.

9. Thay vì `inet_ntoa()`, dùng `inet_ntop()`.
10. Thay vì `gethostbyname()`, dùng `getaddrinfo()` xịn hơn.
11. Thay vì `gethostbyaddr()`, dùng `getnameinfo()` xịn hơn (dù `gethostbyaddr()` vẫn chạy được với IPv6).
12. `INADDR_BROADCAST` không còn chạy nữa. Dùng IPv6 multicast thay thế.

Et voilà!

Chapter 5

System call hoặc không gì cả

Đây là phần chúng ta đi vào các system call (và vài hàm thư viện khác) cho phép bạn chạm tới chức năng mạng của một máy Unix, hay bất kỳ máy nào có sockets API (BSD, Windows, Linux, Mac, vân vân). Khi bạn gọi một trong các hàm này, kernel nhảy vào làm hết công việc cho bạn, tự động như có phép.

Chỗ nhiều người kẹt nhất quanh đây là thứ tự gọi các thứ này. Ở đoạn đó, các trang `man` chả giúp được gì, chắc bạn cũng phát hiện ra rồi. Để cứu cái hoàn cảnh kinh dị đó, tôi đã cố xếp các system call trong các phần dưới đây theo *đúng* (xấp xỉ) thứ tự bạn sẽ cần gọi chúng trong chương trình.

Cộng thêm vài mẩu code mẫu rải rác, chút sữa và bánh quy (mà bạn sợ là phải tự lo), cùng một ít gan và lòng can đảm, và bạn sẽ bắn dữ liệu đi khắp Internet như Con Của Jon Postel!

(Xin lưu ý để ngắn gọn, nhiều đoạn code dưới đây không có kiểm tra lỗi cần thiết. Và chúng hay giả định rằng kết quả gọi `getaddrinfo()` thành công và trả về một phần tử hợp lệ trong linked list. Cả hai tình huống này đều được xử lý đàng hoàng trong các chương trình đứng độc lập, nên cứ lấy mấy cái đó làm mẫu.)

5.1 `getaddrinfo()`: Chuẩn bị phóng!

Đây là một con ngựa thồ thực thụ với khá nhiều tuỳ chọn, nhưng dùng thì thực ra đơn giản. Nó giúp chuẩn bị các `struct` bạn sẽ cần về sau.

Một chút lịch sử: ngày xưa người ta dùng một hàm tên là `gethostbyname()` để làm DNS lookup. Rồi bạn nạp thông tin đó bằng tay vào một `struct sockaddr_in`, và dùng nó trong các lời gọi.

May thay, giờ không cần thế nữa. (Cũng không đáng mơ ước, nếu bạn muốn viết code chạy được với cả IPv4 và IPv6!) Trong thời hiện đại, bạn có hàm `getaddrinfo()` làm đủ thứ thiện lành giùm bạn, bao gồm DNS lookup và tra tên dịch vụ, và còn điền luôn các `struct` bạn cần!

Xem thử cái coi!

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *node,   // e.g. "www.example.com" or IP
                const char *service,  // e.g. "http" or port number
                const struct addrinfo *hints,
                struct addrinfo **res);
```

Bạn đưa vào hàm này ba tham số đầu vào, và nó đưa lại cho bạn con trỏ tới một linked list kết quả là `res`.

Tham số `node` là tên host cần kết nối, hoặc một địa chỉ IP.

Tiếp theo là tham số `service`, có thể là số port, kiểu "80", hoặc tên một dịch vụ cụ thể (tìm trong Bảng Port Của IANA[1] hoặc file `/etc/services` trên máy Unix) kiểu "http" hay "ftp" hay "telnet" hay "smtp" hay gì tuỳ ý.

Cuối cùng, tham số `hints` trỏ tới một `struct addrinfo` mà bạn đã điền sẵn các thông tin liên quan.

Đây là một lời gọi ví dụ nếu bạn là server muốn lắng nghe trên địa chỉ IP của host, port 3490. Lưu ý nó chưa thực sự lắng nghe hay cấu hình mạng gì cả, chỉ chuẩn bị các struct để ta dùng sau:

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo;  // will point to the results

memset(&hints, 0, sizeof hints); // make sure the struct is empty
hints.ai_family = AF_UNSPEC;     // don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

if ((status = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
    fprintf(stderr, "gai error: %s\n", gai_strerror(status));
    exit(1);
}

// servinfo now points to a linked list of 1 or more
// struct addrinfos

// ... do everything until you don't need servinfo anymore ....

freeaddrinfo(servinfo); // free the linked-list
```

Để ý tôi set `ai_family` là `AF_UNSPEC`, tức là tôi không quan tâm xài IPv4 hay IPv6. Bạn có thể set `AF_INET` hoặc `AF_INET6` nếu muốn cụ thể một trong hai.

Bạn cũng thấy cờ `AI_PASSIVE` ở đó; nó bảo `getaddrinfo()` tự gán địa chỉ của local host vào các struct socket. Tiện vì bạn khỏi phải hard-code. (Hoặc bạn đặt một địa chỉ cụ thể vào làm tham số đầu của `getaddrinfo()` chỗ tôi đang để `NULL` ở trên.)

Rồi ta gọi. Nếu có lỗi (`getaddrinfo()` trả về khác không), ta có thể in ra bằng hàm `gai_strerror()`, như bạn thấy. Còn nếu mọi thứ đâu vào đấy, `servinfo` sẽ trỏ tới một linked list của các `struct addrinfo`, mỗi cái chứa một `struct sockaddr` nào đó để ta dùng sau này. Đỉnh!

Cuối cùng, khi ta xong việc với linked list mà `getaddrinfo()` đã tốt bụng cấp phát cho, ta có thể (và nên) giải phóng hết bằng một cú gọi `freeaddrinfo()`.

Đây là ví dụ nếu bạn là client muốn kết nối tới một server cụ thể, ví dụ "www.example.net" port 3490. Lại nữa, cái này chưa thực sự kết nối, chỉ chuẩn bị các struct để dùng sau:

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo;  // will point to the results
```

[1] https://www.iana.org/assignments/port-numbers

```
memset(&hints, 0, sizeof hints); // make sure the struct is empty
hints.ai_family = AF_UNSPEC;     // don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets

// get ready to connect
status = getaddrinfo("www.example.net", "3490", &hints, &servinfo);

// servinfo now points to a linked list of 1 or more
// struct addrinfos

// etc.
```

Tôi cứ nói mãi rằng `servinfo` là linked list với đủ loại thông tin địa chỉ. Viết nhanh một chương trình demo để khoe thông tin đó nào. Chương trình ngắn này[2] sẽ in địa chỉ IP của host nào bạn nhập trên dòng lệnh:

```
/*
** showip.c
**
** show IP addresses for a host given on the command line
*/

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <netinet/in.h>

int main(int argc, char *argv[])
{
    struct addrinfo hints, *res, *p;
    int status;
    char ipstr[INET6_ADDRSTRLEN];

    if (argc != 2) {
        fprintf(stderr,"usage: showip hostname\n");
        return 1;
    }

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;  // Either IPv4 or IPv6
    hints.ai_socktype = SOCK_STREAM;

    if ((status = getaddrinfo(argv[1], NULL, &hints, &res)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(status));
        return 2;
    }

    printf("IP addresses for %s:\n\n", argv[1]);
```

[2] https://beej.us/guide/bgnet/source/examples/showip.c

```
    for(p = res;p != NULL; p = p->ai_next) {
        void *addr;
        char *ipver;
        struct sockaddr_in *ipv4;
        struct sockaddr_in6 *ipv6;

        // get the pointer to the address itself,
        // different fields in IPv4 and IPv6:
        if (p->ai_family == AF_INET) { // IPv4
            ipv4 = (struct sockaddr_in *)p->ai_addr;
            addr = &(ipv4->sin_addr);
            ipver = "IPv4";
        } else { // IPv6
            ipv6 = (struct sockaddr_in6 *)p->ai_addr;
            addr = &(ipv6->sin6_addr);
            ipver = "IPv6";
        }

        // convert the IP to a string and print it:
        inet_ntop(p->ai_family, addr, ipstr, sizeof ipstr);
        printf("  %s: %s\n", ipver, ipstr);
    }

    freeaddrinfo(res); // free the linked list
    return 0;
}
```

Như bạn thấy, code gọi `getaddrinfo()` trên bất kỳ thứ gì bạn truyền vào dòng lệnh, hàm đó điền linked list được `res` trỏ tới, rồi ta duyệt list và in ra hoặc làm gì tuỳ thích.

(Có một đoạn hơi xấu xí chỗ ta phải đào vào các loại `struct sockaddr` khác nhau tuỳ phiên bản IP. Xin lỗi về chuyện đó! Tôi cũng không chắc có cách nào khéo hơn.)

Chạy thử nào! Ai chả thích screenshot:

```
$ showip www.example.net
IP addresses for www.example.net:

  IPv4: 192.0.2.88

$ showip ipv6.example.com
IP addresses for ipv6.example.com:

  IPv4: 192.0.2.101
  IPv6: 2001:db8:8c00:22::171
```

Giờ đã có cái đó trong tay, ta sẽ dùng kết quả từ `getaddrinfo()` để truyền sang các hàm socket khác, và rồi, cuối cùng, dựng được kết nối mạng! Đọc tiếp đi!

5.2 `socket()`: Lấy File Descriptor!

Chắc không trì hoãn được nữa, tôi phải nói về system call `socket()`. Đây là bản phân tích:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Nhưng các tham số này là gì? Chúng cho phép bạn nói rõ muốn loại socket nào (IPv4 hay IPv6, stream hay datagram, TCP hay UDP).

Ngày xưa người ta hard-code các giá trị này, và bạn vẫn hoàn toàn có thể làm thế. (`domain` là `PF_INET` hoặc `PF_INET6` , `type` là `SOCK_STREAM` hoặc `SOCK_DGRAM` , còn `protocol` có thể set là `0` để chọn protocol phù hợp cho `type` đó. Hoặc bạn có thể gọi `getprotobyname()` để tra protocol bạn muốn, "tcp" hay "udp".)

(Cái `PF_INET` này là họ hàng gần của `AF_INET` , cái mà bạn dùng khi khởi tạo trường `sin_family` trong `struct sockaddr_in` . Thực ra chúng gần nhau đến mức có cùng giá trị, và nhiều lập trình viên gọi `socket()` rồi truyền `AF_INET` làm tham số đầu thay vì `PF_INET` . Giờ lấy sữa và bánh quy ra đi, vì đến giờ kể chuyện. Ngày xửa ngày xưa, người ta tưởng rằng một address family (cái mà "AF" trong " `AF_INET` " là viết tắt) có thể hỗ trợ nhiều protocol, được gọi bởi protocol family của chúng (cái mà "PF" trong " `PF_INET` " là viết tắt). Chuyện đó không xảy ra. Và họ sống hạnh phúc bên nhau mãi mãi, Hết. Nên cách đúng nhất là dùng `AF_INET` trong `struct sockaddr_in` và `PF_INET` trong cú gọi `socket()` .)

Thôi, đủ rồi. Cái bạn thực sự muốn làm là dùng các giá trị từ kết quả gọi `getaddrinfo()` , nhét thẳng vào `socket()` như thế này:

```
int s;
struct addrinfo hints, *res;

// do the lookup
// [pretend we already filled out the "hints" struct]
getaddrinfo("www.example.com", "http", &hints, &res);

// again, you should do error-checking on getaddrinfo(), and walk
// the "res" linked list looking for valid entries instead of just
// assuming the first one is good (like many of these examples do).
// See the section on client/server for real examples.

s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

`socket()` chỉ trả về cho bạn một *socket descriptor* để dùng trong các system call sau, hoặc `-1` khi lỗi. Biến toàn cục `errno` được set thành giá trị của lỗi (xem trang man `errno` để biết thêm, và một ghi chú nhanh về việc dùng `errno` trong chương trình đa luồng).

Ổn, ổn, ổn, mà cái socket này được tích sự gì? Câu trả lời là bản thân nó chả được tích sự gì, bạn phải đọc tiếp và gọi thêm system call thì nó mới có nghĩa.

5.3 `bind()` : Tôi đang ở port nào?

Khi đã có một socket, có thể bạn sẽ phải gắn nó với một port trên máy local. (Thường làm vậy nếu bạn chuẩn bị `listen()` đợi kết nối tới trên một port cụ thể, game mạng nhiều người chơi làm vậy khi bảo bạn "kết nối tới 192.168.5.10 port 3490".) Số port được kernel dùng để ghép gói tin tới với socket descriptor của một tiến trình cụ thể. Nếu bạn chỉ định `connect()` (vì bạn là client, không phải server), cái này có lẽ không cần. Cứ đọc đi, cho vui.

Đây là tóm tắt của system call `bind()`:

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

`sockfd` là socket file descriptor do `socket()` trả về. `my_addr` là con trỏ tới một `struct sockaddr` chứa thông tin địa chỉ của bạn, cụ thể là port và địa chỉ IP. `addrlen` là độ dài tính theo byte của địa chỉ đó.

Phù. Hơi nhiều để nuốt trong một lần. Xem ví dụ bind socket vào host mà chương trình đang chạy, port 3490:

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, "3490", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// bind it to the port we passed in to getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);
```

Bằng việc dùng cờ `AI_PASSIVE`, tôi đang bảo chương trình bind vào IP của host đang chạy nó. Nếu bạn muốn bind vào một địa chỉ IP local cụ thể, bỏ `AI_PASSIVE` đi và đặt địa chỉ IP vào tham số đầu của `getaddrinfo()`.

`bind()` cũng trả về `-1` khi lỗi và set `errno` thành giá trị lỗi.

Rất nhiều code cũ đóng gói `struct sockaddr_in` bằng tay trước khi gọi `bind()`. Rõ ràng cái đó chỉ cho IPv4, nhưng thật ra chả có gì ngăn bạn làm điều tương tự với IPv6, chỉ là dùng `getaddrinfo()` thường dễ hơn. Dẫu sao, code cũ trông kiểu này:

```
// !!! THIS IS THE OLD WAY !!!

int sockfd;
struct sockaddr_in my_addr;

sockfd = socket(PF_INET, SOCK_STREAM, 0);

my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(MYPORT);     // short, network byte order
```

```
my_addr.sin_addr.s_addr = inet_addr("10.12.110.57");
memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);

bind(sockfd, (struct sockaddr *)&my_addr, sizeof my_addr);
```

Trong code trên, bạn cũng có thể gán `INADDR_ANY` vào trường `s_addr` nếu muốn bind vào IP local của mình (giống như cờ `AI_PASSIVE` phía trên). Phiên bản IPv6 của `INADDR_ANY` là một biến toàn cục `in6addr_any` được gán vào trường `sin6_addr` của `struct sockaddr_in6`. (Cũng có macro `IN6ADDR_ANY_INIT` bạn có thể dùng trong khởi tạo biến.)

Thêm một cái nữa phải để ý khi gọi `bind()`: đừng đi quá thấp với số port. Mọi port dưới 1024 đều ĐƯỢC DỰ TRỮ (trừ khi bạn là superuser)! Bạn có thể dùng bất cứ port nào trên đó, lên tới 65535 (miễn là chúng chưa bị chương trình khác dùng).

Đôi khi bạn sẽ để ý, bạn chạy lại server và `bind()` fail, báo "Address already in use." Nghĩa là sao? Ồ, một mẩu socket từng kết nối vẫn lảng vảng trong kernel, và nó đang giữ port. Bạn có thể chờ cho nó thoáng ra (khoảng một phút), hoặc thêm code vào chương trình để cho phép tái dùng port, kiểu này:

```
int yes=1;
//char yes='1'; // Solaris people use this

// lose the pesky "Address already in use" error message
setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof yes);
```

Một ghi chú nhỏ cuối cùng về `bind()`: có lúc bạn không nhất thiết phải gọi. Nếu bạn đang `connect()` tới một máy từ xa và không quan tâm local port của mình là bao nhiêu (như trường hợp `telnet`, bạn chỉ quan tâm remote port), bạn chỉ cần gọi `connect()`, nó sẽ kiểm tra xem socket đã được bind chưa, và sẽ `bind()` vào một local port chưa dùng nếu cần.

5.4 `connect()`: Này, bạn kia!

Giả sử vài phút thôi là bạn là ứng dụng telnet. Người dùng ra lệnh cho bạn (y như trong phim *TRON*) lấy một socket file descriptor. Bạn tuân theo và gọi `socket()`. Tiếp, người dùng bảo bạn kết nối tới "`10.12.110.57`" trên port "`23`" (port telnet chuẩn). Ối! Làm gì giờ?

May cho bạn, hỡi chương trình, bạn đang đọc phần về `connect()`, tức là làm sao kết nối tới một host từ xa. Nên đọc tiếp cho cuồng nhiệt! Không có thời gian để mất!

Cú gọi `connect()` như sau:

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

`sockfd` là socket file descriptor thân thiện hàng xóm của ta, do cú gọi `socket()` trả về, `serv_addr` là một `struct sockaddr` chứa port đích và địa chỉ IP, và `addrlen` là độ dài tính theo byte của cấu trúc địa chỉ server.

Mọi thông tin này có thể lấy được từ kết quả của `getaddrinfo()`, thế mới đỉnh.

Bắt đầu có nghĩa hơn chưa? Từ đây tôi không nghe được bạn, nên đành hy vọng là có. Xem ví dụ tạo kết nối socket tới "`www.example.com`", port `3490`:

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;

getaddrinfo("www.example.com", "3490", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// connect!

connect(sockfd, res->ai_addr, res->ai_addrlen);
```

Lại nữa, chương trình kiểu cũ tự điền `struct sockaddr_in` của mình để truyền cho `connect()`. Bạn có thể làm thế nếu muốn. Xem ghi chú tương tự trong phần `bind()` ở trên.

Nhớ kiểm tra giá trị trả về từ `connect()`, nó trả `-1` khi lỗi và set biến `errno`.

Cũng để ý ta không gọi `bind()`. Cơ bản, ta không quan tâm local port của mình; ta chỉ quan tâm đi đâu (remote port). Kernel sẽ chọn một local port giùm ta, và site ta kết nối tới sẽ tự nhận được thông tin đó. Khỏi lo.

5.5 `listen()`: Ai đó gọi tôi đi mà?

Rồi, đổi không khí tí. Nếu bạn không muốn kết nối tới một host từ xa thì sao. Ví dụ cho vui, bạn muốn chờ các kết nối tới và xử lý chúng theo cách nào đó. Quá trình có hai bước: trước tiên `listen()`, rồi `accept()` (xem dưới).

Cú gọi `listen()` khá đơn giản, nhưng cần giải thích tí:

```
int listen(int sockfd, int backlog);
```

`sockfd` là socket file descriptor quen thuộc từ system call `socket()`. `backlog` là số kết nối cho phép trong hàng đợi tới. Nghĩa là sao? Các kết nối tới sẽ chờ trong hàng đợi này cho đến khi bạn `accept()` (xem dưới), và đây là giới hạn bao nhiêu cái được phép xếp hàng. Đa số hệ thống âm thầm giới hạn con số này ở khoảng 20; bạn có thể an toàn với `5` hay `10`.

Lại như thường lệ, `listen()` trả `-1` và set `errno` khi lỗi.

Ừ, chắc bạn đoán được, ta cần gọi `bind()` trước khi gọi `listen()` để server chạy trên một port cụ thể. (Phải báo cho đám bạn biết kết nối vào port nào chứ!) Nên nếu bạn chuẩn bị lắng nghe kết nối tới, dãy system call bạn sẽ gọi là:

```
getaddrinfo();
socket();
bind();
```

```
listen();
/* accept() goes here */
```

Tôi để đây thay cho code mẫu, vì nó cũng tự giải thích rồi. (Code trong phần `accept()` dưới đây đầy đủ hơn.) Phần khó nhất của cả cái mớ này là cú gọi `accept()`.

5.6 `accept()`: "Cảm ơn đã gọi port 3490."

Sẵn sàng chưa, cú gọi `accept()` hơi kỳ kỳ! Chuyện xảy ra như vầy: ai đó xa tít tắp sẽ cố `connect()` tới máy bạn trên một port bạn đang `listen()`. Kết nối của họ sẽ được xếp hàng chờ được `accept()`. Bạn gọi `accept()` và bảo nó lấy kết nối đang chờ. Nó sẽ trả về cho bạn *một socket file descriptor mới toanh* để dùng cho kết nối đơn lẻ này! Đúng vậy, tự nhiên bạn có *hai socket file descriptor* với giá một! Cái gốc vẫn tiếp tục lắng nghe các kết nối mới, còn cái vừa tạo đã sẵn sàng để `send()` và `recv()`. Tới đích rồi!

Cú gọi như sau:

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

`sockfd` là socket descriptor đang `listen()`. Dễ thôi. `addr` thường là con trỏ tới một `struct sockaddr_storage` cục bộ. Đây là nơi thông tin về kết nối tới sẽ được đặt (và cùng với nó bạn xác định được host nào đang gọi mình từ port nào). `addrlen` là một biến `int` cục bộ, nên được set bằng `sizeof(struct sockaddr_storage)` trước khi địa chỉ của nó được truyền cho `accept()`. `accept()` sẽ không nhét quá bấy nhiêu byte vào `addr`. Nếu nó nhét ít hơn, nó sẽ đổi giá trị `addrlen` cho khớp.

Đoán xem? `accept()` trả `-1` và set `errno` khi có lỗi. Cá là bạn đoán ra rồi.

Như trước, đây là cả đống để nuốt một lần, nên có một mẩu code mẫu dưới đây cho bạn ngẫm:

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

#define MYPORT "3490"  // the port users will be connecting to
#define BACKLOG 10     // how many pending connections queue holds

int main(void)
{
    struct sockaddr_storage their_addr;
    socklen_t addr_size;
    struct addrinfo hints, *res;
    int sockfd, new_fd;

    // !! don't forget your error checking for these calls !!

    // first, load up address structs with getaddrinfo():

    memset(&hints, 0, sizeof hints);
```

```
    hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

    getaddrinfo(NULL, MYPORT, &hints, &res);

    // make a socket, bind it, and listen on it:

    sockfd = socket(res->ai_family, res->ai_socktype,
                                                res->ai_protocol);
    bind(sockfd, res->ai_addr, res->ai_addrlen);
    listen(sockfd, BACKLOG);

    // now accept an incoming connection:

    addr_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr,
                                                    &addr_size);

    // ready to communicate on socket descriptor new_fd!
    .
    .
    .
```

Lại nữa, để ý ta sẽ dùng socket descriptor `new_fd` cho mọi cú gọi `send()` và `recv()`. Nếu bạn chỉ nhận đúng một kết nối duy nhất, bạn có thể `close()` cái `sockfd` đang lắng nghe để chặn thêm kết nối tới cùng port, nếu bạn muốn.

5.7 `send()` và `recv()`: Nói với tôi đi, cưng!

Hai hàm này để giao tiếp qua stream socket hoặc datagram socket đã connect. Nếu bạn muốn dùng datagram socket bình thường chưa connect, bạn cần xem phần `sendto()` và `recvfrom()` bên dưới.

> Đây là điều có thể (hoặc không) mới với bạn: mấy cú này là các cú gọi *blocking*. Tức là `recv()` sẽ *block* cho tới khi có dữ liệu sẵn để nhận. "Mà 'block' là cái quái gì đã?!" Nghĩa là chương trình của bạn sẽ dừng ngay đó, trên cái system call đó, cho tới khi ai đó gửi bạn gì đó. (Thuật ngữ dân OS dùng cho "dừng" trong câu trên thực ra là *sleep*, nên tôi có thể dùng hai từ đó thay nhau.) `send()` cũng có thể block nếu thứ bạn đang gửi bị tắc ở đâu đó, nhưng hiếm hơn. Ta sẽ quay lại khái niệm này sau, và nói về cách tránh khi cần.

Đây là cú gọi `send()`:

```
int send(int sockfd, const void *msg, int len, int flags);
```

`sockfd` là socket descriptor bạn muốn gửi dữ liệu tới (cho dù là cái do `socket()` trả về hay cái lấy từ `accept()`). `msg` là con trỏ tới dữ liệu bạn muốn gửi, và `len` là độ dài dữ liệu đó theo byte. Cứ set `flags` bằng `0`. (Xem trang man `send()` để biết thêm về flags.)

Một đoạn code mẫu:

```
char *msg = "Beej was here!";
int len, bytes_sent;
.
.
.
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
.
.
.
```

`send()` trả về số byte thực sự được gửi đi. *Con số này có thể ít hơn số bạn bảo nó gửi!* Đúng rồi, đôi khi bạn bảo nó gửi cả một đống dữ liệu mà nó không kham nổi. Nó sẽ bắn đi được bao nhiêu dữ liệu thì bắn, và tin rằng bạn sẽ gửi nốt phần còn lại sau. Nhớ nhé, nếu giá trị `send()` trả về không khớp với `len`, bạn phải tự gửi nốt phần còn lại của chuỗi. Tin vui: nếu gói tin nhỏ (dưới khoảng 1K), *thường* nó sẽ gửi hết được cả lần. Lại nữa, `-1` được trả về khi lỗi, và `errno` được set thành mã lỗi.

Cú gọi `recv()` giống ở nhiều điểm:

```
int recv(int sockfd, void *buf, int len, int flags);
```

`sockfd` là socket descriptor để đọc, `buf` là buffer để đọc thông tin vào, `len` là độ dài tối đa của buffer, và `flags` lại có thể set bằng `0`. (Xem trang man `recv()` để biết về flags.)

`recv()` trả về số byte thực sự được đọc vào buffer, hoặc `-1` khi lỗi (với `errno` được set tương ứng).

Khoan! `recv()` có thể trả `0`. Chuyện này chỉ có một nghĩa duy nhất: đầu bên kia đã đóng kết nối với bạn! Trả về `0` là cách `recv()` báo cho bạn biết điều đó đã xảy ra.

Đấy, dễ mà, phải không? Giờ bạn đã có thể đưa dữ liệu qua lại trên stream socket! Yay! Bạn đã là Lập Trình Viên Mạng Unix!

5.8 `sendto()` và `recvfrom()`: Nói với tôi đi, kiểu DGRAM

"Tất cả nghe hay ho," tôi nghe bạn nói, "nhưng với datagram socket chưa connect thì sao?" Không vấn đề, amigo. Có ngay đây.

Vì datagram socket không gắn với một host từ xa, đoán xem mẩu thông tin nào ta cần đưa vào trước khi gửi gói? Đúng rồi! Địa chỉ đích! Đây là bức tranh:

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
           const struct sockaddr *to, socklen_t tolen);
```

Như bạn thấy, cú gọi này về cơ bản giống `send()` cộng thêm hai mẩu thông tin. `to` là con trỏ tới một `struct sockaddr` (có lẽ là một `struct sockaddr_in` hay `struct sockaddr_in6` hay `struct sockaddr_storage` bạn ép kiểu vào phút chót) chứa địa chỉ IP đích và port. `tolen`, sâu bên trong là một `int`, có thể chỉ cần set là `sizeof *to` hoặc `sizeof(struct sockaddr_storage)`.

Để có cấu trúc địa chỉ đích trong tay, bạn có thể lấy từ `getaddrinfo()`, hoặc từ `recvfrom()` dưới đây, hoặc tự điền bằng tay.

Y như `send()`, `sendto()` trả về số byte thực sự được gửi (lại nữa, có thể ít hơn số byte bạn bảo nó gửi!), hoặc `-1` khi lỗi.

Y hệt là `recv()` và `recvfrom()`. Tóm tắt của `recvfrom()` là:

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
             struct sockaddr *from, int *fromlen);
```

Lại nữa, cái này giống `recv()` cộng thêm vài trường. `from` là con trỏ tới một `struct sockaddr_storage` cục bộ sẽ được điền địa chỉ IP và port của máy nguồn. `fromlen` là con trỏ tới một `int` cục bộ, nên được khởi tạo bằng `sizeof *from` hoặc `sizeof(struct sockaddr_storage)`. Khi hàm trả về, `fromlen` sẽ chứa độ dài của địa chỉ thực sự được lưu trong `from`.

`recvfrom()` trả về số byte đã nhận, hoặc `-1` khi lỗi (với `errno` được set tương ứng).

Có một câu hỏi: tại sao ta dùng `struct sockaddr_storage` làm kiểu socket? Sao không `struct sockaddr_in`? Vì, bạn thấy đó, ta không muốn buộc mình vào IPv4 hay IPv6. Nên ta dùng `struct sockaddr_storage` chung chung, đủ to cho cả hai.

(Rồi... câu hỏi nữa: sao `struct sockaddr` không đủ to cho mọi địa chỉ? Ta còn ép kiểu cái `struct sockaddr_storage` chung chung về cái `struct sockaddr` chung chung! Có vẻ dư thừa phải không? Câu trả lời là, nó không đủ to, và tôi đoán đổi nó ở thời điểm này sẽ Rắc Rối. Nên người ta làm cái mới.)

Nhớ nhé, nếu bạn `connect()` một datagram socket, bạn có thể đơn giản dùng `send()` và `recv()` cho mọi giao dịch. Bản thân socket vẫn là datagram socket và gói tin vẫn dùng UDP, nhưng interface socket sẽ tự động thêm thông tin đích và nguồn giùm bạn.

5.9 `close()` và `shutdown()`: Biến khỏi mặt tôi đi!

Phù! Bạn đã `send()` và `recv()` dữ liệu cả ngày, và đã đủ rồi. Bạn sẵn sàng đóng kết nối trên socket descriptor của mình. Dễ ợt. Bạn chỉ cần dùng hàm `close()` file descriptor Unix thường dùng:

```
close(sockfd);
```

Cái này sẽ chặn mọi lần đọc và ghi tiếp tới socket. Ai đó cố đọc hay ghi socket ở đầu từ xa sẽ nhận được lỗi.

Phòng khi bạn muốn kiểm soát chút nữa cách socket đóng, bạn có thể dùng hàm `shutdown()`. Nó cho phép bạn cắt giao tiếp theo một hướng nhất định, hoặc cả hai (giống như `close()`). Tóm tắt:

```
int shutdown(int sockfd, int how);
```

`sockfd` là socket file descriptor bạn muốn shutdown, và `how` là một trong các giá trị sau:

`how`	Tác dụng
`0`	Không cho nhận thêm nữa
`1`	Không cho gửi thêm nữa
`2`	Không cho gửi lẫn nhận thêm nữa (giống `close()`)

`shutdown()` trả về `0` khi thành công, và `-1` khi lỗi (với `errno` được set tương ứng).

Nếu bạn chịu khó dùng `shutdown()` trên datagram socket chưa connect, nó chỉ làm socket không còn dùng được cho các cú gọi `send()` và `recv()` tiếp theo (nhớ là bạn có thể dùng chúng nếu đã `connect()` datagram socket của mình).

Lưu ý quan trọng, `shutdown()` không thực sự đóng file descriptor, nó chỉ đổi khả năng dùng của nó. Để giải phóng một socket descriptor, bạn cần dùng `close()`.

Không có gì cả.

(Trừ việc nhớ rằng nếu bạn dùng Windows và Winsock thì nên gọi `closesocket()` thay vì `close()`.)

5.10 `getpeername()`: Bạn là ai?

Hàm này dễ cực.

Dễ đến mức tôi suýt không cho nó nguyên một phần. Nhưng thôi cứ để đây.

Hàm `getpeername()` sẽ cho bạn biết ai ở đầu bên kia của một stream socket đã kết nối. Tóm tắt:

```
#include <sys/socket.h>

int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

`sockfd` là descriptor của stream socket đã kết nối, `addr` là con trỏ tới một `struct sockaddr` (hoặc `struct sockaddr_in`) sẽ giữ thông tin về đầu kia của kết nối, và `addrlen` là con trỏ tới một `int`, nên được khởi tạo bằng `sizeof *addr` hoặc `sizeof(struct sockaddr)`.

Hàm trả `-1` khi lỗi và set `errno` tương ứng.

Khi đã có địa chỉ của họ, bạn có thể dùng `inet_ntop()`, `getnameinfo()`, hoặc `gethostbyaddr()` để in ra hoặc lấy thêm thông tin. Không, bạn không thể lấy được tên login của họ. (Thôi được, được. Nếu máy kia chạy một ident daemon, thì làm được. Tuy nhiên, cái đó vượt quá phạm vi tài liệu này. Xem RFC 1413[3] để biết thêm.)

5.11 `gethostname()`: Tôi là ai?

Còn dễ hơn cả `getpeername()` là hàm `gethostname()`. Nó trả về tên của máy tính mà chương trình của bạn đang chạy. Tên này có thể được dùng bởi `getaddrinfo()` ở trên, để xác định địa chỉ IP của máy local.

Còn gì vui hơn? Tôi nghĩ ra được vài thứ, nhưng chúng không liên quan tới lập trình socket. Dẫu sao, đây là bản phân tích:

```
#include <unistd.h>

int gethostname(char *hostname, size_t size);
```

Các tham số đơn giản: `hostname` là con trỏ tới một mảng char sẽ chứa hostname sau khi hàm trả về, và `size` là độ dài theo byte của mảng `hostname`.

Hàm trả về `0` khi thành công, và `-1` khi lỗi, set `errno` như thường lệ.

[3] https://tools.ietf.org/html/rfc1413

Chapter 6

Nền tảng client-server

Thế giới này là thế giới client-server, cưng ơi. Gần như mọi thứ trên mạng đều xoay quanh các tiến trình client nói chuyện với các tiến trình server và ngược lại. Ví dụ như `telnet`. Khi bạn kết nối tới một host từ xa trên port 23 bằng telnet (client), một chương trình trên host đó (tên `telnetd`, server) bật dậy. Nó xử lý kết nối telnet tới, dựng cho bạn một prompt đăng nhập, vân vân.

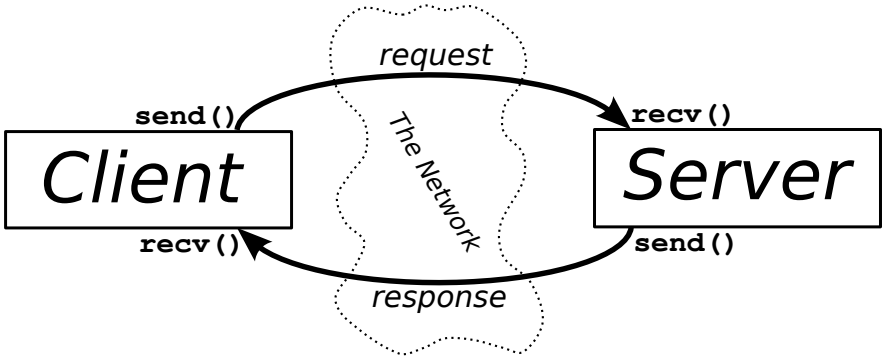

Figure 6.1: Tương tác client-server.

Việc trao đổi thông tin giữa client và server được tóm tắt trong sơ đồ ở trên.

Lưu ý cặp client-server có thể nói `SOCK_STREAM`, `SOCK_DGRAM`, hoặc bất cứ thứ gì khác (miễn là cùng nói một thứ). Vài cặp client-server hay gặp: `telnet` / `telnetd`, `ftp` / `ftpd`, hay `Firefox` / `Apache`. Mỗi lần bạn dùng `ftp`, có một chương trình từ xa tên `ftpd` đang phục vụ bạn.

Thường thì, một máy sẽ chỉ có một server, và server đó xử lý nhiều client bằng `fork()`. Quy trình cơ bản là: server chờ một kết nối, `accept()` nó, và `fork()` một tiến trình con để xử lý. Đó là điều server mẫu của chúng ta làm trong phần kế tiếp.

6.1 Server stream đơn giản

Tất cả những gì server này làm là gửi chuỗi "`Hello, world!`" đi qua một kết nối stream. Bạn chỉ cần chạy nó ở một cửa sổ, rồi telnet vào từ cửa sổ khác bằng:

```
$ telnet remotehostname 3490
```

trong đó `remotehostname` là tên máy bạn đang chạy server.

Code server[1]:

```
/*
** server.c -- a stream socket server demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

#define PORT "3490"  // the port users will be connecting to

#define BACKLOG 10   // how many pending connections queue will hold

void sigchld_handler(int s)
{
    (void)s; // quiet unused variable warning

    // waitpid() might overwrite errno, so we save and restore it:
    int saved_errno = errno;

    while(waitpid(-1, NULL, WNOHANG) > 0);

    errno = saved_errno;
}


// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(void)
{
    // listen on sock_fd, new connection on new_fd
    int sockfd, new_fd;
    struct addrinfo hints, *servinfo, *p;
    struct sockaddr_storage their_addr; // connector's address info
```

[1] https://beej.us/guide/bgnet/source/examples/server.c

```
    socklen_t sin_size;
    struct sigaction sa;
    int yes=1;
    char s[INET6_ADDRSTRLEN];
    int rv;

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE; // use my IP

    if ((rv = getaddrinfo(NULL, PORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // loop through all the results and bind to the first we can
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("server: socket");
            continue;
        }

        if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes,
                sizeof(int)) == -1) {
            perror("setsockopt");
            exit(1);
        }

        if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            perror("server: bind");
            continue;
        }

        break;
    }

    freeaddrinfo(servinfo); // all done with this structure

    if (p == NULL)  {
        fprintf(stderr, "server: failed to bind\n");
        exit(1);
    }

    if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
    }

    sa.sa_handler = sigchld_handler; // reap all dead processes
    sigemptyset(&sa.sa_mask);
```

```
    sa.sa_flags = SA_RESTART;
    if (sigaction(SIGCHLD, &sa, NULL) == -1) {
        perror("sigaction");
        exit(1);
    }

    printf("server: waiting for connections...\n");

    while(1) {  // main accept() loop
        sin_size = sizeof their_addr;
        new_fd = accept(sockfd, (struct sockaddr *)&their_addr,
            &sin_size);
        if (new_fd == -1) {
            perror("accept");
            continue;
        }

        inet_ntop(their_addr.ss_family,
            get_in_addr((struct sockaddr *)&their_addr),
            s, sizeof s);
        printf("server: got connection from %s\n", s);

        if (!fork()) { // this is the child process
            close(sockfd); // child doesn't need the listener
            if (send(new_fd, "Hello, world!", 13, 0) == -1)
                perror("send");
            close(new_fd);
            exit(0);
        }
        close(new_fd);  // parent doesn't need this
    }

    return 0;
}
```

Nếu bạn tò mò, tôi để code trong một hàm `main()` to (theo tôi cảm thấy) cho rõ mặt cú pháp. Bạn cứ tự nhiên tách ra thành các hàm nhỏ hơn nếu thấy dễ chịu hơn.

(Cả cái chuyện `sigaction()` này có thể mới với bạn, không sao. Đoạn code ở đó chịu trách nhiệm dọn dẹp các tiến trình zombie xuất hiện khi các tiến trình con đã `fork()` thoát ra. Nếu bạn đẻ ra cả đống zombie mà không dọn, ông quản trị hệ thống của bạn sẽ nhảy dựng lên.)

Bạn có thể lấy dữ liệu từ server này bằng client liệt kê ở phần tiếp theo.

6.2 Client stream đơn giản

Cậu này còn dễ hơn cả server. Tất cả những gì client này làm là kết nối tới host bạn ghi ở dòng lệnh, port 3490. Nó nhận chuỗi server gửi đi.

Source client[2]:

[2] https://beej.us/guide/bgnet/source/examples/client.c

```
/*
** client.c -- a stream socket client demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#include <arpa/inet.h>

#define PORT "3490" // the port client will be connecting to

#define MAXDATASIZE 100 // max number of bytes we can get at once

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct addrinfo hints, *servinfo, *p;
    int rv;
    char s[INET6_ADDRSTRLEN];

    if (argc != 2) {
        fprintf(stderr,"usage: client hostname\n");
        exit(1);
    }

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    if ((rv = getaddrinfo(argv[1], PORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // loop through all the results and connect to the first we can
```

```
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("client: socket");
            continue;
        }

        inet_ntop(p->ai_family,
            get_in_addr((struct sockaddr *)p->ai_addr),
            s, sizeof s);
        printf("client: attempting connection to %s\n", s);

        if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            perror("client: connect");
            close(sockfd);
            continue;
        }

        break;
    }

    if (p == NULL) {
        fprintf(stderr, "client: failed to connect\n");
        return 2;
    }

    inet_ntop(p->ai_family,
            get_in_addr((struct sockaddr *)p->ai_addr),
            s, sizeof s);
    printf("client: connected to %s\n", s);

    freeaddrinfo(servinfo); // all done with this structure

    if ((numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1) {
        perror("recv");
        exit(1);
    }

    buf[numbytes] = '\0';

    printf("client: received '%s'\n",buf);

    close(sockfd);

    return 0;
}
```

Để ý nếu bạn không chạy server trước khi chạy client, `connect()` trả về "Connection refused". Rất hữu ích.

6.3 Datagram socket

Ta đã đi qua cơ bản của UDP datagram socket ở phần thảo luận về `sendto()` và `recvfrom()` phía trên, nên tôi chỉ giới thiệu vài chương trình mẫu: `talker.c` và `listener.c`.

`listener` ngồi trên một máy chờ gói tin tới ở port 4950. `talker` gửi một gói tới port đó, trên máy được chỉ định, chứa bất cứ thứ gì người dùng gõ ở dòng lệnh.

Vì datagram socket không có kết nối và chỉ bắn gói tin ra không trung với thái độ thờ ơ về chuyện có đến nơi không, ta sẽ bảo client và server dùng cụ thể IPv6. Như vậy ta tránh được tình huống server lắng nghe trên IPv6 còn client gửi bằng IPv4; dữ liệu đơn giản sẽ không được nhận. (Ở thế giới TCP stream socket có kết nối, lệch kiểu này vẫn có thể xảy ra, nhưng lỗi `connect()` cho một address family sẽ khiến ta thử lại cái kia.)

Đây là source của `listener.c`[3]:

```
/*
** listener.c -- a datagram sockets "server" demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define MYPORT "4950"    // the port users will be connecting to

#define MAXBUFLEN 100

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(void)
{
    int sockfd;
    struct addrinfo hints, *servinfo, *p;
    int rv;
    int numbytes;
    struct sockaddr_storage their_addr;
    char buf[MAXBUFLEN];
```

[3] https://beej.us/guide/bgnet/source/examples/listener.c

```
    socklen_t addr_len;
    char s[INET6_ADDRSTRLEN];

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_INET6; // or set to AF_INET to use IPv4
    hints.ai_socktype = SOCK_DGRAM;
    hints.ai_flags = AI_PASSIVE; // use my IP

    if ((rv = getaddrinfo(NULL, MYPORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // loop through all the results and bind to the first we can
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("listener: socket");
            continue;
        }

        if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            perror("listener: bind");
            continue;
        }

        break;
    }

    if (p == NULL) {
        fprintf(stderr, "listener: failed to bind socket\n");
        return 2;
    }

    freeaddrinfo(servinfo);

    printf("listener: waiting to recvfrom...\n");

    addr_len = sizeof their_addr;
    if ((numbytes = recvfrom(sockfd, buf, MAXBUFLEN-1 , 0,
        (struct sockaddr *)&their_addr, &addr_len)) == -1) {
        perror("recvfrom");
        exit(1);
    }

    printf("listener: got packet from %s\n",
        inet_ntop(their_addr.ss_family,
            get_in_addr((struct sockaddr *)&their_addr),
            s, sizeof s));
    printf("listener: packet is %d bytes long\n", numbytes);
    buf[numbytes] = '\0';
    printf("listener: packet contains \"%s\"\n", buf);
```

```
    close(sockfd);

    return 0;
}
```

Để ý trong cú gọi `getaddrinfo()` cuối cùng ta dùng `SOCK_DGRAM`. Cũng lưu ý không cần `listen()` hay `accept()`. Đây là một cái thú vị của datagram socket chưa connect!

Tiếp theo là source của `talker.c` [4]:

```
/*
** talker.c -- a datagram "client" demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define SERVERPORT "4950"   // the port users will be connecting to

int main(int argc, char *argv[])
{
    int sockfd;
    struct addrinfo hints, *servinfo, *p;
    int rv;
    int numbytes;

    if (argc != 3) {
        fprintf(stderr,"usage: talker hostname message\n");
        exit(1);
    }

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_INET6; // set to AF_INET to use IPv4
    hints.ai_socktype = SOCK_DGRAM;

    rv = getaddrinfo(argv[1], SERVERPORT, &hints, &servinfo);
    if (rv != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // loop through all the results and make a socket
```

[4] https://beej.us/guide/bgnet/source/examples/talker.c

```
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("talker: socket");
            continue;
        }

        break;
    }

    if (p == NULL) {
        fprintf(stderr, "talker: failed to create socket\n");
        return 2;
    }

    if ((numbytes = sendto(sockfd, argv[2], strlen(argv[2]), 0,
             p->ai_addr, p->ai_addrlen)) == -1) {
        perror("talker: sendto");
        exit(1);
    }

    freeaddrinfo(servinfo);

    printf("talker: sent %d bytes to %s\n", numbytes, argv[1]);
    close(sockfd);

    return 0;
}
```

Và bấy nhiêu thôi! Chạy `listener` trên một máy, rồi chạy `talker` trên một máy khác. Xem chúng nói chuyện với nhau! Niềm vui lành mạnh cho cả nhà!

Bạn còn chả cần chạy server lần này! Bạn có thể chạy `talker` một mình, và nó vui vẻ bắn gói tin ra không trung, chúng biến mất nếu đầu kia không có ai sẵn `recvfrom()` chờ. Nhớ nhé: dữ liệu gửi qua UDP datagram socket không đảm bảo tới nơi!

Trừ một chi tiết nhỏ mà tôi đã nhắc nhiều lần ở trên: datagram socket đã connect. Tôi cần nói ở đây, vì ta đang ở phần datagram của tài liệu. Giả sử `talker` gọi `connect()` và chỉ định địa chỉ của `listener`. Từ lúc đó, `talker` chỉ có thể gửi tới và nhận từ địa chỉ được `connect()` chỉ định. Vì lý do đó, bạn không cần dùng `sendto()` và `recvfrom()`, cứ dùng `send()` và `recv()` cho xong.

Chapter 7

Một Vài Kỹ Thuật Hơi Nâng Cao

Mấy cái này thật ra không *nâng cao* gì cho lắm, nhưng nó đã ra khỏi phần căn bản mà chúng ta đã đi qua rồi. Thật ra, nếu bạn đã lê tới tận đây, bạn có thể tự cho mình là khá thành thạo phần căn bản của lập trình mạng Unix rồi đấy! Chúc mừng!

Vậy giờ chúng ta bước vào cái thế giới mới mẻ và rực rỡ của những thứ bí hiểm hơn về socket mà bạn có thể muốn tìm hiểu. Chiến thôi!

7.1 Blocking

Blocking. Bạn đã nghe về nó rồi, vậy nó thực chất là cái quái gì? Nói gọn, "block" là tiếng lóng dân kỹ thuật để chỉ "ngủ". Bạn chắc đã để ý rằng khi chạy `listener` ở phía trên, nó cứ ngồi đó chờ cho đến khi có gói tin đến. Cái xảy ra là nó đã gọi `recvfrom()`, chẳng có dữ liệu nào cả, nên người ta nói `recvfrom()` đã "block" (nghĩa là nằm ngủ ở đó) cho tới khi có dữ liệu.

Rất nhiều hàm bị block. `accept()` bị block. Toàn bộ họ hàng `recv()` đều bị block. Lý do chúng làm được vậy là vì chúng được phép làm vậy. Khi bạn tạo socket descriptor lần đầu bằng `socket()`, kernel đặt nó ở chế độ blocking. Nếu bạn không muốn một socket bị blocking, bạn phải gọi `fcntl()`:

```
#include <unistd.h>
#include <fcntl.h>
.
.
.
sockfd = socket(PF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
.
.
.
```

Bằng cách đặt socket ở chế độ non-blocking, bạn có thể "poll" socket để lấy thông tin một cách hiệu quả. Nếu bạn cố gắng đọc từ một socket non-blocking mà không có dữ liệu, nó không được phép block, nó sẽ trả về `-1` và `errno` được gán thành `EAGAIN` hoặc `EWOULDBLOCK`.

(Khoan, nó có thể trả về `EAGAIN` *hoặc* `EWOULDBLOCK`? Vậy phải kiểm tra cái nào? Đặc tả thật ra không chỉ định hệ thống của bạn sẽ trả về cái nào, nên để code chạy được trên mọi nơi, kiểm tra cả hai.)

Nói chung thì, kiểu polling này là ý tồi. Nếu bạn đặt chương trình vào một vòng lặp busy-wait để tìm dữ liệu trên socket, bạn sẽ ngốn CPU như thể nó miễn phí. Một giải pháp thanh lịch hơn để kiểm tra xem có

dữ liệu đang đợi được đọc hay không sẽ xuất hiện trong phần tiếp theo về `poll()`.

7.2 `poll()`: Synchronous I/O Multiplexing

Cái bạn thực sự muốn làm là bằng cách nào đó theo dõi *một đống* socket cùng lúc rồi xử lý những cái nào đã có dữ liệu sẵn. Như vậy bạn không cần phải liên tục poll tất cả mấy cái socket đó xem cái nào sẵn sàng đọc.

> *Xin lưu ý: `poll()` chậm kinh khủng khi số lượng kết nối cực lớn. Trong những tình huống đó, bạn sẽ có hiệu năng tốt hơn nếu dùng một event library như libevent[a], thư viện này cố gắng dùng phương pháp nhanh nhất có sẵn trên hệ thống của bạn.*
>
> [a] https://libevent.org/

Vậy làm sao tránh được polling? Một cách có chút trớ trêu là, bạn có thể tránh polling bằng cách dùng system call `poll()`. Nói gọn, chúng ta sẽ nhờ hệ điều hành làm hết phần việc bẩn cho mình, và chỉ cần báo cho chúng ta biết khi nào có dữ liệu sẵn sàng để đọc trên socket nào. Trong thời gian đó, process của chúng ta có thể nằm ngủ, tiết kiệm tài nguyên hệ thống.

Kế hoạch chung là giữ một mảng `struct pollfd` chứa thông tin về những socket descriptor nào chúng ta muốn theo dõi, và muốn theo dõi những loại sự kiện nào. Hệ điều hành sẽ block ở lời gọi `poll()` cho đến khi một trong những sự kiện đó xảy ra (ví dụ "socket sẵn sàng để đọc!") hoặc cho đến khi hết thời gian timeout mà người dùng đặt.

Tiện lợi ở chỗ, một socket đang `listen()` sẽ báo "sẵn sàng đọc" khi có một kết nối mới sẵn sàng để `accept()`.

Nói đủ rồi. Làm sao dùng cái này đây?

```
#include <poll.h>

int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

`fds` là mảng thông tin (socket nào theo dõi cái gì), `nfds` là số phần tử trong mảng, còn `timeout` là timeout tính bằng milliseconds. Nó trả về số phần tử trong mảng đã có sự kiện xảy ra.

Hãy xem qua cái `struct` đó:

```
struct pollfd {
    int fd;         // the socket descriptor
    short events;   // bitmap of events we're interested in
    short revents;  // on return, bitmap of events that occurred
};
```

Vậy chúng ta sẽ có một mảng những cái đó, và sẽ đặt trường `fd` của mỗi phần tử bằng socket descriptor mà chúng ta quan tâm theo dõi. Rồi chúng ta sẽ đặt trường `events` để chỉ định loại sự kiện quan tâm.

Trường `events` là phép OR bit của các giá trị sau:

Macro	Mô tả
`POLLIN`	Báo cho tôi khi có dữ liệu sẵn sàng để `recv()` trên socket này.
`POLLOUT`	Báo cho tôi khi tôi có thể `send()` dữ liệu đến socket này mà không bị block.
`POLLHUP`	Báo cho tôi khi đầu bên kia đóng kết nối.

Khi đã có mảng `struct pollfd` sẵn sàng, bạn có thể truyền nó cho `poll()`, kèm theo kích thước mảng, cùng với giá trị timeout tính bằng milliseconds. (Bạn có thể chỉ định timeout âm để chờ mãi.)

Sau khi `poll()` trả về, bạn có thể kiểm tra trường `revents` để xem `POLLIN` hoặc `POLLOUT` có được bật không, cho biết sự kiện đó đã xảy ra.

(Thật ra bạn có thể làm nhiều hơn với `poll()`. Xem man page của `poll()`, ở phía dưới, để biết chi tiết.)

Đây là một ví dụ[1], chúng ta chờ 2.5 giây để có dữ liệu sẵn sàng đọc từ standard input, tức là khi bạn bấm `RETURN`:

```
#include <stdio.h>
#include <poll.h>

int main(void)
{
    struct pollfd pfds[1]; // More if you want to monitor more

    pfds[0].fd = 0;          // Standard input
    pfds[0].events = POLLIN; // Tell me when ready to read

    // If you needed to monitor other things, as well:
    //pfds[1].fd = some_socket; // Some socket descriptor
    //pfds[1].events = POLLIN;  // Tell me when ready to read

    printf("Hit RETURN or wait 2.5 seconds for timeout\n");

    int num_events = poll(pfds, 1, 2500); // 2.5 second timeout

    if (num_events == 0) {
        printf("Poll timed out!\n");
    } else {
        int pollin_happened = pfds[0].revents & POLLIN;

        if (pollin_happened) {
            printf("File descriptor %d is ready to read\n",
                    pfds[0].fd);
        } else {
            printf("Unexpected event occurred: %d\n",
                    pfds[0].revents);
        }
    }

    return 0;
}
```

Chú ý lại rằng `poll()` trả về số phần tử trong mảng `pfds` mà có sự kiện xảy ra. Nó *không* cho bạn biết *phần tử nào* trong mảng (bạn vẫn phải quét để tìm), nhưng nó có cho bạn biết có bao nhiêu phần tử có trường `revents` khác không (nên bạn có thể ngừng quét sau khi tìm được đủ số đó).

Có vài câu hỏi có thể nảy ra ở đây: làm sao thêm file descriptor mới vào tập hợp truyền cho `poll()`? Cho cái này, chỉ cần đảm bảo bạn có đủ chỗ trong mảng cho tất cả những gì bạn cần, hoặc `realloc()` thêm chỗ khi cần.

[1] https://beej.us/guide/bgnet/source/examples/poll.c

Còn việc xóa phần tử khỏi tập hợp thì sao? Cho cái này, bạn có thể sao chép phần tử cuối cùng trong mảng đè lên phần tử bạn đang xóa. Rồi truyền vào một số đếm nhỏ hơn một đơn vị cho `poll()`. Một cách khác là bạn có thể đặt trường `fd` thành một số âm và `poll()` sẽ bỏ qua nó.

Làm sao ráp tất cả lại thành một chat server mà bạn có thể `telnet` vào?

Cái chúng ta sẽ làm là khởi tạo một listener socket, rồi thêm nó vào tập file descriptor cho `poll()` theo dõi. (Nó sẽ báo sẵn-sàng-đọc khi có kết nối đi tới.)

Rồi chúng ta sẽ thêm các kết nối mới vào mảng `struct pollfd` của mình. Và chúng ta sẽ mở rộng nó linh động nếu hết chỗ.

Khi một kết nối bị đóng, chúng ta sẽ xóa nó khỏi mảng.

Và khi một kết nối sẵn-sàng-đọc, chúng ta sẽ đọc dữ liệu từ nó và gửi dữ liệu đó tới tất cả các kết nối khác để họ thấy được người khác gõ gì.

Hãy thử poll server này[2]. Chạy nó trong một cửa sổ, rồi `telnet localhost 9034` từ một số cửa sổ terminal khác. Bạn sẽ thấy được những gì bạn gõ trong một cửa sổ hiện ra ở những cửa sổ kia (sau khi bấm RETURN).

Không chỉ vậy, nếu bạn bấm `CTRL-]` rồi gõ `quit` để thoát `telnet`, server sẽ phát hiện việc ngắt kết nối và xóa bạn khỏi mảng file descriptor.

```
/*
** pollserver.c -- a cheezy multiperson chat server
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <poll.h>

#define PORT "9034"   // Port we're listening on

/*
 * Convert socket to IP address string.
 * addr: struct sockaddr_in or struct sockaddr_in6
 */
const char *inet_ntop2(void *addr, char *buf, size_t size)
{
    struct sockaddr_storage *sas = addr;
    struct sockaddr_in *sa4;
    struct sockaddr_in6 *sa6;
    void *src;

    switch (sas->ss_family) {
        case AF_INET:
            sa4 = addr;
```

[2] https://beej.us/guide/bgnet/source/examples/pollserver.c

```
            src = &(sa4->sin_addr);
            break;
        case AF_INET6:
            sa6 = addr;
            src = &(sa6->sin6_addr);
            break;
        default:
            return NULL;
    }

    return inet_ntop(sas->ss_family, src, buf, size);
}

/*
 * Return a listening socket.
 */
int get_listener_socket(void)
{
    int listener;     // Listening socket descriptor
    int yes=1;        // For setsockopt() SO_REUSEADDR, below
    int rv;

    struct addrinfo hints, *ai, *p;

    // Get us a socket and bind it
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;
    if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) != 0) {
        fprintf(stderr, "pollserver: %s\n", gai_strerror(rv));
        exit(1);
    }

    for(p = ai; p != NULL; p = p->ai_next) {
        listener = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol);
        if (listener < 0) {
            continue;
        }

        // Lose the pesky "address already in use" error message
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes,
                sizeof(int));

        if (bind(listener, p->ai_addr, p->ai_addrlen) < 0) {
            close(listener);
            continue;
        }

        break;
    }
```

```
    // If we got here, it means we didn't get bound
    if (p == NULL) {
        return -1;
    }

    freeaddrinfo(ai); // All done with this

    // Listen
    if (listen(listener, 10) == -1) {
        return -1;
    }

    return listener;
}

/*
 * Add a new file descriptor to the set.
 */
void add_to_pfds(struct pollfd **pfds, int newfd, int *fd_count,
        int *fd_size)
{
    // If we don't have room, add more space in the pfds array
    if (*fd_count == *fd_size) {
        *fd_size *= 2; // Double it
        *pfds = realloc(*pfds, sizeof(**pfds) * (*fd_size));
    }

    (*pfds)[*fd_count].fd = newfd;
    (*pfds)[*fd_count].events = POLLIN; // Check ready-to-read
    (*pfds)[*fd_count].revents = 0;

    (*fd_count)++;
}

/*
 * Remove a file descriptor at a given index from the set.
 */
void del_from_pfds(struct pollfd pfds[], int i, int *fd_count)
{
    // Copy the one from the end over this one
    pfds[i] = pfds[*fd_count-1];

    (*fd_count)--;
}

/*
 * Handle incoming connections.
 */
void handle_new_connection(int listener, int *fd_count,
        int *fd_size, struct pollfd **pfds)
{
    struct sockaddr_storage remoteaddr; // Client address
    socklen_t addrlen;
```

```
    int newfd;  // Newly accept()ed socket descriptor
    char remoteIP[INET6_ADDRSTRLEN];

    addrlen = sizeof remoteaddr;
    newfd = accept(listener, (struct sockaddr *)&remoteaddr,
            &addrlen);

    if (newfd == -1) {
        perror("accept");
    } else {
        add_to_pfds(pfds, newfd, fd_count, fd_size);

        printf("pollserver: new connection from %s on socket %d\n",
                inet_ntop2(&remoteaddr, remoteIP, sizeof remoteIP),
                newfd);
    }
}

/*
 * Handle regular client data or client hangups.
 */
void handle_client_data(int listener, int *fd_count,
        struct pollfd *pfds, int *pfd_i)
{
    char buf[256];    // Buffer for client data

    int nbytes = recv(pfds[*pfd_i].fd, buf, sizeof buf, 0);

    int sender_fd = pfds[*pfd_i].fd;

    if (nbytes <= 0) { // Got error or connection closed by client
        if (nbytes == 0) {
            // Connection closed
            printf("pollserver: socket %d hung up\n", sender_fd);
        } else {
            perror("recv");
        }

        close(pfds[*pfd_i].fd); // Bye!

        del_from_pfds(pfds, *pfd_i, fd_count);

        // reexamine the slot we just deleted
        (*pfd_i)--;

    } else { // We got some good data from a client
        printf("pollserver: recv from fd %d: %.*s", sender_fd,
                nbytes, buf);
        // Send to everyone!
        for(int j = 0; j < *fd_count; j++) {
            int dest_fd = pfds[j].fd;

            // Except the listener and ourselves
```

```
            if (dest_fd != listener && dest_fd != sender_fd) {
                if (send(dest_fd, buf, nbytes, 0) == -1) {
                    perror("send");
                }
            }
        }
    }
}

/*
 * Process all existing connections.
 */
void process_connections(int listener, int *fd_count, int *fd_size,
        struct pollfd **pfds)
{
    for(int i = 0; i < *fd_count; i++) {

        // Check if someone's ready to read
        if ((*pfds)[i].revents & (POLLIN | POLLHUP)) {
            // We got one!!

            if ((*pfds)[i].fd == listener) {
                // If we're the listener, it's a new connection
                handle_new_connection(listener, fd_count, fd_size,
                        pfds);
            } else {
                // Otherwise we're just a regular client
                handle_client_data(listener, fd_count, *pfds, &i);
            }
        }
    }
}

/*
 * Main: create a listener and connection set, loop forever
 * processing connections.
 */
int main(void)
{
    int listener;     // Listening socket descriptor

    // Start off with room for 5 connections
    // (We'll realloc as necessary)
    int fd_size = 5;
    int fd_count = 0;
    struct pollfd *pfds = malloc(sizeof *pfds * fd_size);

    // Set up and get a listening socket
    listener = get_listener_socket();

    if (listener == -1) {
        fprintf(stderr, "error getting listening socket\n");
        exit(1);
```

```
    }

    // Add the listener to set;
    // Report ready to read on incoming connection
    pfds[0].fd = listener;
    pfds[0].events = POLLIN;

    fd_count = 1; // For the listener

    puts("pollserver: waiting for connections...");

    // Main loop
    for(;;) {
        int poll_count = poll(pfds, fd_count, -1);

        if (poll_count == -1) {
            perror("poll");
            exit(1);
        }

        // Run through connections looking for data to read
        process_connections(listener, &fd_count, &fd_size, &pfds);
    }

    free(pfds);
}
```

Trong phần tiếp theo, chúng ta sẽ xem một hàm tương tự, cũ hơn, gọi là `select()`. Cả `select()` và `poll()` đều có chức năng và hiệu năng tương tự nhau, chỉ khác nhau ở cách dùng. `select()` có thể portable hơn một chút, nhưng có lẽ hơi cồng kềnh khi sử dụng. Chọn cái nào bạn thích nhất, miễn là nó được hỗ trợ trên hệ thống của bạn.

7.3 `select()`: Synchronous I/O Multiplexing, Kiểu Cổ Điển

Hàm này hơi lạ, nhưng rất hữu ích. Hãy tưởng tượng tình huống sau: bạn là một server, bạn muốn lắng nghe các kết nối mới đi tới đồng thời vẫn tiếp tục đọc từ những kết nối bạn đã có.

Không thành vấn đề, bạn nói, chỉ cần một `accept()` và vài cái `recv()` là xong. Khoan đã, anh bạn! Lỡ bạn đang block ở lời gọi `accept()` thì sao? Bạn sẽ `recv()` dữ liệu kiểu gì cùng lúc đó? "Dùng socket non-blocking đi!" Còn lâu! Bạn đâu muốn thành kẻ ngốn CPU. Vậy thì sao?

`select()` cho bạn quyền năng theo dõi nhiều socket cùng một lúc. Nó sẽ cho bạn biết cái nào sẵn sàng đọc, cái nào sẵn sàng ghi, và cái nào đã phát sinh exception, nếu bạn thực sự muốn biết cái đó.

> *Xin lưu ý: `select()`, dù rất portable, chậm kinh khủng khi số lượng kết nối cực lớn. Trong những tình huống đó, bạn sẽ có hiệu năng tốt hơn nếu dùng một event library như libevent[a], thư viện này cố gắng dùng phương pháp nhanh nhất có sẵn trên hệ thống của bạn.*
>
> [a] https://libevent.org/

Không dài dòng nữa, tôi sẽ giới thiệu tóm tắt về `select()`:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int numfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

Hàm này theo dõi các "tập hợp" file descriptor; cụ thể là `readfds`, `writefds`, và `exceptfds`. Nếu bạn muốn xem mình có thể đọc từ standard input và một socket descriptor nào đó, `sockfd`, thì chỉ cần thêm các file descriptor `0` và `sockfd` vào tập `readfds`. Tham số `numfds` nên được đặt bằng giá trị của file descriptor cao nhất cộng một. Trong ví dụ này, nó nên được đặt thành `sockfd+1`, vì chắc chắn nó cao hơn standard input (`0`).

Khi `select()` trả về, `readfds` sẽ bị sửa để phản ánh cái nào trong các file descriptor bạn đã chọn là sẵn sàng để đọc. Bạn có thể kiểm tra chúng bằng macro `FD_ISSET()`, ở phía dưới.

Trước khi đi xa hơn, tôi sẽ nói về cách thao tác với các tập hợp này. Mỗi tập thuộc kiểu `fd_set`. Các macro sau làm việc với kiểu này:

Hàm	Mô tả
`FD_SET(int fd, fd_set *set);`	Thêm `fd` vào `set`.
`FD_CLR(int fd, fd_set *set);`	Bỏ `fd` khỏi `set`.
`FD_ISSET(int fd, fd_set *set);`	Trả về true nếu `fd` nằm trong `set`.
`FD_ZERO(fd_set *set);`	Xóa toàn bộ phần tử khỏi `set`.

Cuối cùng, cái `struct timeval` lạ đời này là gì vậy? Nhiều khi bạn không muốn chờ vô tận để ai đó gửi dữ liệu. Có thể cứ mỗi 96 giây bạn muốn in "Still Going..." ra terminal mặc dù chẳng có gì xảy ra. Cái struct thời gian này cho phép bạn chỉ định khoảng thời gian timeout. Nếu thời gian bị vượt quá mà `select()` vẫn chưa tìm thấy file descriptor nào sẵn sàng, nó sẽ trả về để bạn có thể tiếp tục xử lý.

`struct timeval` có các trường sau:

```
struct timeval {
    int tv_sec;     // seconds
    int tv_usec;    // microseconds
};
```

Chỉ cần gán `tv_sec` bằng số giây cần chờ, và `tv_usec` bằng số microsecond cần chờ. Vâng, là _micro_second, không phải millisecond. Có 1.000 microsecond trong một millisecond, và 1.000 millisecond trong một giây. Như vậy, có 1.000.000 microsecond trong một giây. Tại sao lại là "usec"? Chữ "u" được vẽ trông giống chữ cái Hy Lạp μ (Mu) mà chúng ta dùng cho "micro". Ngoài ra, khi hàm trả về, `timeout` *có thể* được cập nhật để cho biết thời gian còn lại. Cái này tùy vào bản Unix bạn đang chạy.

Yay! Chúng ta có timer độ phân giải microsecond! Khoan đã, đừng tin vào điều đó. Chắc bạn sẽ phải chờ một phần khoảng timeslice tiêu chuẩn của Unix bất kể bạn đặt `struct timeval` nhỏ cỡ nào.

Một vài chuyện thú vị khác: Nếu bạn gán các trường trong `struct timeval` thành `0`, `select()` sẽ timeout ngay lập tức, về bản chất là poll toàn bộ file descriptor trong các tập của bạn. Nếu bạn đặt tham số `timeout` thành NULL, nó sẽ không bao giờ timeout, và sẽ chờ cho đến khi file descriptor đầu tiên sẵn

sàng. Cuối cùng, nếu bạn không quan tâm đến việc chờ một tập nào đó, bạn chỉ cần đặt nó thành NULL trong lời gọi `select()`.

Đoạn code sau[3] chờ 2.5 giây để có thứ gì đó xuất hiện trên standard input:

```
/*
** select.c -- a select() demo
*/

#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define STDIN 0  // file descriptor for standard input

int main(void)
{
    struct timeval tv;
    fd_set readfds;

    tv.tv_sec = 2;
    tv.tv_usec = 500000;

    FD_ZERO(&readfds);
    FD_SET(STDIN, &readfds);

    // don't care about writefds and exceptfds:
    select(STDIN+1, &readfds, NULL, NULL, &tv);

    if (FD_ISSET(STDIN, &readfds))
        printf("A key was pressed!\n");
    else
        printf("Timed out.\n");

    return 0;
}
```

Nếu bạn đang dùng terminal chế độ line-buffered, phím bạn bấm phải là RETURN, nếu không nó vẫn sẽ timeout.

Lúc này, một vài người trong các bạn có thể nghĩ đây là cách tuyệt vời để chờ dữ liệu trên datagram socket, và các bạn đúng: nó *có thể*. Một số Unix có thể dùng select theo kiểu này, một số thì không. Bạn nên xem man page địa phương của mình nói gì về chuyện này nếu muốn thử.

Một số Unix cập nhật thời gian trong `struct timeval` của bạn để phản ánh lượng thời gian còn lại trước khi timeout. Nhưng số khác thì không. Đừng trông cậy vào chuyện đó nếu bạn muốn portable. (Dùng `gettimeofday()` nếu bạn cần theo dõi thời gian đã trôi qua. Đáng tiếc, tôi biết, nhưng sự đời là vậy.)

Chuyện gì xảy ra nếu một socket trong tập đọc đóng kết nối? Trong trường hợp đó, `select()` sẽ trả về với socket descriptor đó được đánh dấu là "sẵn sàng đọc". Khi bạn thực sự `recv()` từ nó, `recv()` sẽ trả về `0`. Đó là cách bạn biết client đã đóng kết nối.

Một điểm thú vị nữa về `select()`: nếu bạn có một socket đang `listen()`, bạn có thể kiểm tra xem có

[3] https://beej.us/guide/bgnet/source/examples/select.c

kết nối mới hay không bằng cách đặt file descriptor của socket đó vào tập `readfds`.

Và đó, các bạn của tôi, là tổng quan nhanh về hàm `select()` đầy quyền năng.

Nhưng, theo yêu cầu đông đảo, đây là một ví dụ chi tiết. Không may, sự khác biệt giữa ví dụ đơn giản như bùn ở trên và cái này đây là đáng kể. Nhưng hãy xem qua, rồi đọc phần mô tả đi kèm sau đó.

Chương trình này[4] hoạt động như một chat server đa người dùng đơn giản. Khởi động nó trong một cửa sổ, rồi `telnet` vào ("`telnet hostname 9034`") từ nhiều cửa sổ khác. Khi bạn gõ gì đó trong một phiên `telnet`, nó sẽ xuất hiện ở tất cả phiên còn lại.

```
/*
** selectserver.c -- a cheezy multiperson chat server
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define PORT "9034"   // port we're listening on

/*
 * Convert socket to IP address string.
 * addr: struct sockaddr_in or struct sockaddr_in6
 */
const char *inet_ntop2(void *addr, char *buf, size_t size)
{
    struct sockaddr_storage *sas = addr;
    struct sockaddr_in *sa4;
    struct sockaddr_in6 *sa6;
    void *src;

    switch (sas->ss_family) {
        case AF_INET:
            sa4 = addr;
            src = &(sa4->sin_addr);
            break;
        case AF_INET6:
            sa6 = addr;
            src = &(sa6->sin6_addr);
            break;
        default:
            return NULL;
    }

    return inet_ntop(sas->ss_family, src, buf, size);
}
```

[4] https://beej.us/guide/bgnet/source/examples/selectserver.c

```
/*
 * Return a listening socket
 */
int get_listener_socket(void)
{
    struct addrinfo hints, *ai, *p;
    int yes=1;    // for setsockopt() SO_REUSEADDR, below
    int rv;
    int listener;

    // get us a socket and bind it
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;
    if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) != 0) {
        fprintf(stderr, "selectserver: %s\n", gai_strerror(rv));
        exit(1);
    }

    for(p = ai; p != NULL; p = p->ai_next) {
        listener = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol);
        if (listener < 0) {
            continue;
        }

        // lose the pesky "address already in use" error message
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes,
                sizeof(int));

        if (bind(listener, p->ai_addr, p->ai_addrlen) < 0) {
            close(listener);
            continue;
        }

        break;
    }

    // if we got here, it means we didn't get bound
    if (p == NULL) {
        fprintf(stderr, "selectserver: failed to bind\n");
        exit(2);
    }

    freeaddrinfo(ai); // all done with this

    // listen
    if (listen(listener, 10) == -1) {
        perror("listen");
        exit(3);
    }
```

```
    return listener;
}

/*
 * Add new incoming connections to the proper sets
 */
void handle_new_connection(int listener, fd_set *master, int *fdmax)
{
    socklen_t addrlen;
    int newfd;        // newly accept()ed socket descriptor
    struct sockaddr_storage remoteaddr; // client address
    char remoteIP[INET6_ADDRSTRLEN];

    addrlen = sizeof remoteaddr;
    newfd = accept(listener,
        (struct sockaddr *)&remoteaddr,
        &addrlen);

    if (newfd == -1) {
        perror("accept");
    } else {
        FD_SET(newfd, master); // add to master set
        if (newfd > *fdmax) {  // keep track of the max
            *fdmax = newfd;
        }
        printf("selectserver: new connection from %s on "
            "socket %d\n",
            inet_ntop2(&remoteaddr, remoteIP, sizeof remoteIP),
            newfd);
    }
}

/*
 * Broadcast a message to all clients
 */
void broadcast(char *buf, int nbytes, int listener, int s,
               fd_set *master, int fdmax)
{
    for(int j = 0; j <= fdmax; j++) {
        // send to everyone!
        if (FD_ISSET(j, master)) {
            // except the listener and ourselves
            if (j != listener && j != s) {
                if (send(j, buf, nbytes, 0) == -1) {
                    perror("send");
                }
            }
        }
    }
}

/*
```

```
 * Handle client data and hangups
 */
void handle_client_data(int s, int listener, fd_set *master,
                        int fdmax)
{
    char buf[256];    // buffer for client data
    int nbytes;

    // handle data from a client
    if ((nbytes = recv(s, buf, sizeof buf, 0)) <= 0) {
        // got error or connection closed by client
        if (nbytes == 0) {
            // connection closed
            printf("selectserver: socket %d hung up\n", s);
        } else {
            perror("recv");
        }
        close(s); // bye!
        FD_CLR(s, master); // remove from master set
    } else {
        // we got some data from a client
        broadcast(buf, nbytes, listener, s, master, fdmax);
    }
}

/*
 * Main
 */
int main(void)
{
    fd_set master;    // master file descriptor list
    fd_set read_fds;  // temp file descriptor list for select()
    int fdmax;        // maximum file descriptor number

    int listener;     // listening socket descriptor

    FD_ZERO(&master);    // clear the master and temp sets
    FD_ZERO(&read_fds);

    listener = get_listener_socket();

    // add the listener to the master set
    FD_SET(listener, &master);

    // keep track of the biggest file descriptor
    fdmax = listener; // so far, it's this one

    // main loop
    for(;;) {
        read_fds = master; // copy it
        if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
            perror("select");
            exit(4);
```

```
        }

        // run through the existing connections looking for data
        // to read
        for(int i = 0; i <= fdmax; i++) {
            if (FD_ISSET(i, &read_fds)) { // we got one!!
                if (i == listener)
                    handle_new_connection(i, &master, &fdmax);
                else
                    handle_client_data(i, listener, &master, fdmax);
            }
        }
    }

    return 0;
}
```

Chú ý rằng tôi có hai tập file descriptor trong code: `master` và `read_fds`. Tập đầu, `master`, giữ tất cả socket descriptor hiện đang được kết nối, cũng như socket descriptor đang lắng nghe kết nối mới.

Lý do tôi có tập `master` là vì `select()` thật sự *sửa* tập bạn truyền vào để phản ánh socket nào đang sẵn sàng đọc. Vì tôi phải theo dõi các kết nối từ lần gọi `select()` này qua lần gọi kế, tôi phải giữ chúng ở một nơi an toàn. Vào phút chót, tôi sao chép `master` sang `read_fds`, rồi mới gọi `select()`.

Nhưng chẳng phải điều đó có nghĩa là mỗi lần tôi có kết nối mới, tôi phải thêm nó vào tập `master` sao? Chuẩn! Và mỗi khi một kết nối đóng, tôi phải xóa nó khỏi tập `master` à? Vâng, đúng vậy.

Chú ý là tôi kiểm tra khi nào socket `listener` sẵn sàng đọc. Khi nó sẵn sàng, nghĩa là tôi có một kết nối mới đang chờ, và tôi `accept()` nó rồi thêm vào tập `master`. Tương tự, khi một kết nối client sẵn sàng đọc, và `recv()` trả về `0`, tôi biết client đã đóng kết nối, và tôi phải xóa nó khỏi tập `master`.

Nếu `recv()` của client trả về khác không, thì tôi biết đã có dữ liệu được nhận. Nên tôi lấy nó, rồi duyệt qua danh sách `master` và gửi dữ liệu đó đến tất cả các client đang kết nối còn lại.

Và đó, các bạn của tôi, là tổng quan không-hẳn-là-đơn-giản về hàm `select()` đầy quyền năng.

Một chú ý nhanh cho các fan Linux ngoài kia: đôi lúc, trong vài tình huống hiếm hoi, `select()` của Linux có thể trả về "sẵn-sàng-đọc" rồi thật ra lại không sẵn sàng đọc! Nghĩa là nó sẽ block ở `read()` sau khi `select()` bảo nó sẽ không block! Trời ạ, cái thằng! Cách khắc phục là bật cờ `O_NONBLOCK` trên socket nhận để nó trả về lỗi `EWOULDBLOCK` (mà bạn có thể an toàn bỏ qua nếu nó xảy ra). Xem trang tham khảo `fcntl()` để biết thêm về cách đặt socket ở chế độ non-blocking.

Thêm nữa, đây là một ghi chú bonus: có một hàm khác gọi là `poll()` hoạt động khá giống `select()`, nhưng với hệ thống quản lý tập file descriptor khác. Xem qua đi!

7.4 Xử Lý `send()` Một Phần

Còn nhớ hồi ở phần về `send()` phía trên, tôi đã nói rằng `send()` có thể không gửi hết số byte bạn yêu cầu chứ? Nghĩa là, bạn muốn nó gửi 512 byte, nhưng nó trả về 412. Chuyện gì xảy ra với 100 byte còn lại?

Chúng vẫn còn trong cái buffer nhỏ của bạn, đang đợi được gửi đi. Vì những hoàn cảnh ngoài tầm kiểm soát của bạn, kernel đã quyết định không gửi tất cả dữ liệu ra trong một đợt, và giờ, bạn ơi, đến lượt bạn phải đẩy dữ liệu đó ra.

Bạn có thể viết một hàm như thế này để làm việc đó:

```
#include <sys/types.h>
#include <sys/socket.h>

int sendall(int s, char *buf, int *len)
{
    int total = 0;        // how many bytes we've sent
    int bytesleft = *len; // how many we have left to send
    int n;

    while(total < *len) {
        n = send(s, buf+total, bytesleft, 0);
        if (n == -1) { break; }
        total += n;
        bytesleft -= n;
    }

    *len = total; // return number actually sent here

    return n==-1?-1:0; // return -1 on failure, 0 on success
}
```

Trong ví dụ này, `s` là socket bạn muốn gửi dữ liệu đến, `buf` là buffer chứa dữ liệu, và `len` là con trỏ trỏ tới một `int` chứa số byte trong buffer.

Hàm trả về `-1` khi có lỗi (và `errno` vẫn còn được gán từ lời gọi `send()`). Ngoài ra, số byte thực sự được gửi được trả về trong `len`. Nó sẽ là cùng một số byte bạn yêu cầu gửi, trừ khi có lỗi. `sendall()` sẽ cố hết sức, hổn hển thở dốc, để gửi dữ liệu ra, nhưng nếu có lỗi, nó sẽ báo lại cho bạn ngay.

Cho đầy đủ, đây là một lời gọi mẫu của hàm:

```
char buf[10] = "Beej!";
int len;

len = strlen(buf);
if (sendall(s, buf, &len) == -1) {
    perror("sendall");
    printf("We only sent %d bytes because of the error!\n", len);
}
```

Chuyện gì xảy ra ở đầu bên nhận khi chỉ một phần gói tin đến? Nếu các gói tin có độ dài biến đổi, làm sao bên nhận biết khi nào một gói kết thúc và một gói khác bắt đầu? Vâng, các tình huống đời thực là một cơn đau đầu kiểu hoàng gia nhức cả mông. Chắc bạn sẽ phải *đóng gói* (còn nhớ chuyện đó từ phần về đóng gói dữ liệu mãi tít đằng trước chứ?) Đọc tiếp đi!

7.5 Serialization: Cách Gói Dữ Liệu

Gửi dữ liệu dạng text qua mạng thì khá dễ, bạn đang thấy vậy, nhưng sẽ ra sao nếu bạn muốn gửi dữ liệu "nhị phân" như `int` hay `float`? Hóa ra bạn có một vài lựa chọn.

1. Chuyển con số thành text bằng hàm như `sprintf()`, rồi gửi text. Bên nhận sẽ phân tích text trở lại thành số bằng hàm như `strtol()`.
2. Cứ gửi dữ liệu thô, truyền một con trỏ trỏ tới dữ liệu cho `send()`.

3. Mã hóa con số thành một dạng nhị phân portable. Bên nhận sẽ giải mã.

Xem trước nhanh! Chỉ đêm nay thôi!

[*Màn sân khấu kéo lên*]

Beej nói, "Tôi thích Cách Ba ở trên nhất!"

[*HẾT*]

(Trước khi bắt đầu phần này một cách nghiêm túc, tôi phải nói với bạn rằng có các thư viện ngoài kia làm việc này, và tự cuộn tay làm lấy mà vẫn portable và không có lỗi là một thử thách đáng kể. Nên đi tìm hiểu và làm bài tập về nhà trước khi quyết định tự tay làm mấy thứ này. Tôi đưa thông tin vào đây cho ai tò mò muốn biết mấy thứ kiểu này hoạt động ra sao.)

Thật ra mọi cách ở trên đều có nhược và ưu điểm riêng, nhưng, như tôi đã nói, nhìn chung tôi thích cách thứ ba. Trước hết, hãy nói về một số nhược và ưu điểm của hai cách kia.

Cách thứ nhất, mã hóa các con số thành text trước khi gửi, có ưu điểm là bạn có thể dễ dàng in ra và đọc được dữ liệu đang chạy trên đường truyền. Đôi khi một giao thức dễ đọc cho người là rất tuyệt khi dùng trong tình huống không đòi hỏi nhiều băng thông, như với Internet Relay Chat (IRC)[5]. Tuy nhiên, nó có nhược điểm là việc chuyển đổi chậm, và kết quả hầu như luôn chiếm nhiều chỗ hơn con số gốc!

Cách hai: truyền dữ liệu thô. Cái này khá dễ (nhưng nguy hiểm!): chỉ cần lấy con trỏ trỏ tới dữ liệu muốn gửi, và gọi send với nó.

```
double d = 3490.15926535;

send(s, &d, sizeof d, 0);  /* DANGER--non-portable! */
```

Bên nhận lấy nó như sau:

```
double d;

recv(s, &d, sizeof d, 0);  /* DANGER--non-portable! */
```

Nhanh, đơn giản, còn gì để chê? Vâng, hóa ra không phải mọi kiến trúc đều biểu diễn `double` (hay `int` cũng vậy) với cùng bit representation hay cùng thứ tự byte! Code này rõ ràng là không portable. (Ê, có khi bạn không cần portable, trong trường hợp đó thì cái này nhanh và ngon.)

Khi đóng gói các kiểu số nguyên, chúng ta đã thấy họ hàng `htons()` giúp giữ mọi thứ portable bằng cách chuyển các con số sang Network Byte Order ra sao, và đó là Điều Đúng Đắn nên làm. Không may, không có hàm tương tự cho kiểu `float`. Mọi hy vọng đã mất sao?

Đừng sợ! (Bạn có sợ lúc đó không? Không à? Không chút nào?) Có thứ chúng ta có thể làm: chúng ta có thể pack (hoặc "marshal", hoặc "serialize", hoặc một trong cả ngàn triệu cái tên khác) dữ liệu thành một định dạng nhị phân đã biết mà bên nhận có thể unpack ở đầu bên kia.

"Định dạng nhị phân đã biết" là gì? Chúng ta đã thấy ví dụ `htons()` rồi, nhỉ? Nó đổi (hoặc "mã hóa", nếu bạn muốn nghĩ theo cách đó) một con số từ bất kỳ định dạng nào của máy chủ sang Network Byte Order. Để đảo ngược (giải mã), bên nhận gọi `ntohs()`.

Nhưng chẳng phải tôi vừa nói xong là không có hàm nào như thế cho các kiểu phi số nguyên khác sao? Đúng vậy. Tôi có nói. Và vì không có cách chuẩn nào trong C để làm điều này, đây là một tình thế khó nhằn (một câu đùa gratuitous dành cho các fan Python của tôi).

Điều cần làm là đóng gói dữ liệu vào một định dạng đã biết và gửi nó qua đường truyền để giải mã. Ví dụ, để pack `float`, đây là một thứ nhanh và bẩn với nhiều chỗ để cải thiện[6]:

[5] https://en.wikipedia.org/wiki/Internet_Relay_Chat
[6] https://beej.us/guide/bgnet/source/examples/pack.c

```
#include <stdint.h>

uint32_t htonf(float f)
{
    uint32_t p;
    uint32_t sign;

    if (f < 0) { sign = 1; f = -f; }
    else { sign = 0; }

    // whole part and sign
    p = ((((uint32_t)f)&0x7fff)<<16) | (sign<<31);

    // fraction
    p |= (uint32_t)(((f - (int)f) * 65536.0f))&0xffff;

    return p;
}

float ntohf(uint32_t p)
{
    float f = ((p>>16)&0x7fff); // whole part
    f += (p&0xffff) / 65536.0f; // fraction

    if (((p>>31)&0x1) == 0x1) { f = -f; } // sign bit set

    return f;
}
```

Đoạn code trên là một cài đặt khá ngây thơ lưu một `float` trong một số 32-bit. Bit cao nhất (31) được dùng để lưu dấu của số ("1" nghĩa là âm), và bảy bit kế tiếp (30-16) được dùng để lưu phần nguyên của `float`. Cuối cùng, các bit còn lại (15-0) được dùng để lưu phần lẻ của số.

Cách dùng khá thẳng thắn:

```
#include <stdio.h>

int main(void)
{
    float f = 3.1415926, f2;
    uint32_t netf;

    netf = htonf(f);  // convert to "network" form
    f2 = ntohf(netf); // convert back to test

    printf("Original: %f\n", f);        // 3.141593
    printf(" Network: 0x%08X\n", netf); // 0x0003243F
    printf("Unpacked: %f\n", f2);       // 3.141586

    return 0;
}
```

Ổ mặt tích cực, nó nhỏ, đơn giản và nhanh. Ổ mặt tiêu cực, nó dùng không gian không hiệu quả và dải giá trị bị hạn chế nghiêm trọng, thử lưu một số lớn hơn 32767 vào đấy xem, nó sẽ không vui đâu! Bạn cũng có

thể thấy trong ví dụ trên rằng vài chữ số thập phân cuối cùng không được bảo toàn chính xác.

Chúng ta có thể làm gì thay thế? *Chuẩn* để lưu các số dấu chấm động được gọi là IEEE-754[7]. Hầu hết máy tính dùng định dạng này nội bộ cho việc làm toán dấu chấm động, nên trong các trường hợp đó, nói chính xác ra, không cần chuyển đổi gì. Nhưng nếu bạn muốn source code của mình portable, đó là một giả định bạn không nhất thiết có thể đặt.

Hoặc bạn có thể đặt? Rất có khả năng hệ thống của bạn là IEEE-754, giống như khả năng cao nó là số bù hai cho số nguyên. Nên nếu bạn biết mình có cái đó, bạn chỉ cần truyền dữ liệu qua đường truyền (dù bạn cần sửa endianness bằng `htonl()` hoặc hàm phù hợp, `float` cũng có endianness). Và đây là cái `htons()` cùng đồng bọn làm trên các hệ thống big-endian, nơi không cần chuyển đổi.

Nhưng phòng trường hợp bạn đang ở trên hệ thống không phải IEEE-754, đây là đoạn code mã hóa `float` và `double` sang định dạng IEEE-754[8]. (Chủ yếu thôi, nó không mã hóa NaN hay Infinity, nhưng có thể sửa lại để làm được.)

```
#define pack754_32(f) (pack754((f), 32, 8))
#define pack754_64(f) (pack754((f), 64, 11))
#define unpack754_32(i) (unpack754((i), 32, 8))
#define unpack754_64(i) (unpack754((i), 64, 11))

uint64_t pack754(long double f, unsigned bits, unsigned expbits)
{
    long double fnorm;
    int shift;
    long long sign, exp, significand;

    // -1 for sign bit
    unsigned significandbits = bits - expbits - 1;

    if (f == 0.0) return 0; // get this special case out of the way

    // check sign and begin normalization
    if (f < 0) { sign = 1; fnorm = -f; }
    else { sign = 0; fnorm = f; }

    // get the normalized form of f and track the exponent
    shift = 0;
    while(fnorm >= 2.0) { fnorm /= 2.0; shift++; }
    while(fnorm < 1.0) { fnorm *= 2.0; shift--; }
    fnorm = fnorm - 1.0;

    // calculate the binary form (non-float) of the significand data
    significand = fnorm * ((1LL<<significandbits) + 0.5f);

    // get the biased exponent
    exp = shift + ((1<<(expbits-1)) - 1); // shift + bias

    // return the final answer
    return (sign<<(bits-1)) | (exp<<(bits-expbits-1)) | significand;
}
```

[7] https://en.wikipedia.org/wiki/IEEE_754
[8] https://beej.us/guide/bgnet/source/examples/ieee754.c

```
long double unpack754(uint64_t i, unsigned bits, unsigned expbits)
{
    long double result;
    long long shift;
    unsigned bias;

    // -1 for sign bit
    unsigned significandbits = bits - expbits - 1;

    if (i == 0) return 0.0;

    // pull the significand
    result = (i&((1LL<<significandbits)-1)); // mask
    result /= (1LL<<significandbits); // convert back to float
    result += 1.0f; // add the one back on

    // deal with the exponent
    bias = (1<<(expbits-1)) - 1;
    shift = ((i>>significandbits)&((1LL<<expbits)-1)) - bias;
    while(shift > 0) { result *= 2.0; shift--; }
    while(shift < 0) { result /= 2.0; shift++; }

    // sign it
    result *= (i>>(bits-1))&1? -1.0: 1.0;

    return result;
}
```

Tôi đặt vài macro tiện dụng ở trên cùng để đóng gói và mở gói các số 32-bit (có thể là `float`) và 64-bit (có thể là `double`), nhưng hàm `pack754()` có thể được gọi trực tiếp và bảo nó mã hóa `bits` bit dữ liệu (trong đó `expbits` bit được dành cho phần mũ của số chuẩn hóa).

Đây là cách dùng mẫu:

```
#include <stdio.h>
#include <stdint.h> // defines uintN_t types
#include <inttypes.h> // defines PRIx macros

int main(void)
{
    float f = 3.1415926, f2;
    double d = 3.14159265358979323, d2;
    uint32_t fi;
    uint64_t di;

    fi = pack754_32(f);
    f2 = unpack754_32(fi);

    di = pack754_64(d);
    d2 = unpack754_64(di);

    printf("float before : %.7f\n", f);
```

```
    printf("float encoded: 0x%08" PRIx32 "\n", fi);
    printf("float after  : %.7f\n\n", f2);

    printf("double before : %.20lf\n", d);
    printf("double encoded: 0x%016" PRIx64 "\n", di);
    printf("double after  : %.20lf\n", d2);

    return 0;
}
```

Đoạn code trên cho ra output:

```
float before : 3.1415925
float encoded: 0x40490FDA
float after  : 3.1415925

double before : 3.14159265358979311600
double encoded: 0x400921FB54442D18
double after  : 3.14159265358979311600
```

Một câu hỏi khác bạn có thể có là làm sao đóng gói `struct`? Không may cho bạn, compiler được tự do nhét padding khắp nơi trong `struct`, và điều đó nghĩa là bạn không thể gửi nguyên cục đó qua đường truyền một cách portable trong một đợt. (Bạn có đang chán nghe "không thể làm cái này", "không thể làm cái kia" không? Xin lỗi! Để dẫn lời một người bạn của tôi, "Bất cứ khi nào có gì trục trặc, tôi luôn đổ lỗi cho Microsoft." Cái này có thể không phải lỗi của Microsoft, đành nhận vậy, nhưng phát biểu của bạn tôi hoàn toàn đúng.)

Quay lại chuyện chính: cách tốt nhất để gửi `struct` qua đường truyền là đóng gói từng trường độc lập rồi mở gói chúng thành `struct` khi đến đầu bên kia.

Nghe làm nhiều việc quá, bạn đang nghĩ vậy. Vâng, đúng thế. Một điều bạn có thể làm là viết một hàm trợ giúp để giúp đóng gói dữ liệu cho bạn. Sẽ vui lắm! Thật mà!

Trong sách *The Practice of Programming*[9] của Kernighan và Pike, họ cài đặt các hàm kiểu `printf()` tên là `pack()` và `unpack()` làm chính xác chuyện này. Tôi sẽ link tới chúng, nhưng có vẻ mấy hàm đó không có trên mạng cùng với phần còn lại của source sách.

(*The Practice of Programming* là một cuốn đọc xuất sắc. Zeus cứu một con mèo con mỗi khi tôi giới thiệu nó.)

Tới đây, tôi sẽ thả một gợi ý về một cài đặt Protocol Buffers bằng C[10] mà tôi chưa từng dùng, nhưng trông hoàn toàn tử tế. Các lập trình viên Python và Perl sẽ muốn xem qua các hàm `pack()` và `unpack()` của ngôn ngữ mình để hoàn thành cùng chuyện đó. Còn Java có cái interface Serializable to đùng có thể dùng theo cách tương tự.

Nhưng nếu bạn muốn tự viết tiện ích đóng gói của mình trong C, mánh của K&P là dùng variable argument list để tạo các hàm kiểu `printf()` để dựng các gói tin. Đây là phiên bản tôi tự nấu lên[11] dựa trên đó mà hy vọng sẽ đủ để cho bạn ý tưởng về cách một thứ như vậy có thể hoạt động.

(Code này tham chiếu đến các hàm `pack754()` ở trên. Các hàm `packi*()` hoạt động giống họ hàng `htons()` quen thuộc, ngoại trừ việc chúng pack vào một mảng `char` thay vì một số nguyên khác.)

[9] https://beej.us/guide/url/tpop
[10] https://github.com/protobuf-c/protobuf-c
[11] https://beej.us/guide/bgnet/source/examples/pack2.c

```
#include <stdio.h>
#include <ctype.h>
#include <stdarg.h>
#include <string.h>

/*
** packi16() -- store a 16-bit int into a char buffer (like htons())
*/
void packi16(unsigned char *buf, unsigned int i)
{
    *buf++ = i>>8; *buf++ = i;
}

/*
** packi32() -- store a 32-bit int into a char buffer (like htonl())
*/
void packi32(unsigned char *buf, unsigned long int i)
{
    *buf++ = i>>24; *buf++ = i>>16;
    *buf++ = i>>8;  *buf++ = i;
}

/*
** packi64() -- store a 64-bit int into a char buffer (like htonl())
*/
void packi64(unsigned char *buf, unsigned long long int i)
{
    *buf++ = i>>56; *buf++ = i>>48;
    *buf++ = i>>40; *buf++ = i>>32;
    *buf++ = i>>24; *buf++ = i>>16;
    *buf++ = i>>8;  *buf++ = i;
}

/*
** unpacki16() -- unpack a 16-bit int from a char buffer (like
**                ntohs())
*/
int unpacki16(unsigned char *buf)
{
    unsigned int i2 = ((unsigned int)buf[0]<<8) | buf[1];
    int i;

    // change unsigned numbers to signed
    if (i2 <= 0x7fffu) { i = i2; }
    else { i = -1 - (unsigned int)(0xffffu - i2); }

    return i;
}

/*
** unpacku16() -- unpack a 16-bit unsigned from a char buffer (like
**                ntohs())
*/
```

```
unsigned int unpacku16(unsigned char *buf)
{
    return ((unsigned int)buf[0]<<8) | buf[1];
}

/*
** unpacki32() -- unpack a 32-bit int from a char buffer (like
**                ntohl())
*/
long int unpacki32(unsigned char *buf)
{
    unsigned long int i2 = ((unsigned long int)buf[0]<<24) |
                           ((unsigned long int)buf[1]<<16) |
                           ((unsigned long int)buf[2]<<8)  |
                           buf[3];
    long int i;

    // change unsigned numbers to signed
    if (i2 <= 0x7fffffffu) { i = i2; }
    else { i = -1 - (long int)(0xffffffffu - i2); }

    return i;
}

/*
** unpacku32() -- unpack a 32-bit unsigned from a char buffer (like
**                ntohl())
*/
unsigned long int unpacku32(unsigned char *buf)
{
    return ((unsigned long int)buf[0]<<24) |
           ((unsigned long int)buf[1]<<16) |
           ((unsigned long int)buf[2]<<8)  |
           buf[3];
}

/*
** unpacki64() -- unpack a 64-bit int from a char buffer (like
**                ntohl())
*/
long long int unpacki64(unsigned char *buf)
{
    unsigned long long int i2 =
        ((unsigned long long int)buf[0]<<56) |
        ((unsigned long long int)buf[1]<<48) |
        ((unsigned long long int)buf[2]<<40) |
        ((unsigned long long int)buf[3]<<32) |
        ((unsigned long long int)buf[4]<<24) |
        ((unsigned long long int)buf[5]<<16) |
        ((unsigned long long int)buf[6]<<8)  |
        buf[7];
    long long int i;
```

```
    // change unsigned numbers to signed
    if (i2 <= 0x7fffffffffffffffu) { i = i2; }
    else { i = -1 -(long long int)(0xffffffffffffffffu - i2); }

    return i;
}

/*
** unpacku64() -- unpack a 64-bit unsigned from a char buffer (like
**                ntohl())
*/
unsigned long long int unpacku64(unsigned char *buf)
{
    return ((unsigned long long int)buf[0]<<56) |
           ((unsigned long long int)buf[1]<<48) |
           ((unsigned long long int)buf[2]<<40) |
           ((unsigned long long int)buf[3]<<32) |
           ((unsigned long long int)buf[4]<<24) |
           ((unsigned long long int)buf[5]<<16) |
           ((unsigned long long int)buf[6]<<8)  |
           buf[7];
}

/*
** pack() -- store data dictated by the format string in the buffer
**
**   bits |signed   unsigned   float   string
**   -----+----------------------------------
**      8 |   c        C
**     16 |   h        H         f
**     32 |   l        L         d
**     64 |   q        Q         g
**      - |                              s
**
**  (16-bit unsigned length is automatically prepended to strings)
*/

unsigned int pack(unsigned char *buf, char *format, ...)
{
    va_list ap;

    signed char c;              // 8-bit
    unsigned char C;

    int h;                      // 16-bit
    unsigned int H;

    long int l;                 // 32-bit
    unsigned long int L;

    long long int q;            // 64-bit
    unsigned long long int Q;
```

```
    float f;                    // floats
    double d;
    long double g;
    unsigned long long int fhold;

    char *s;                    // strings
    unsigned int len;

    unsigned int size = 0;

    va_start(ap, format);

    for(; *format != '\0'; format++) {
        switch(*format) {
        case 'c': // 8-bit
            size += 1;
            c = (signed char)va_arg(ap, int); // promoted
            *buf++ = c;
            break;

        case 'C': // 8-bit unsigned
            size += 1;
            C = (unsigned char)va_arg(ap, unsigned int); // promoted
            *buf++ = C;
            break;

        case 'h': // 16-bit
            size += 2;
            h = va_arg(ap, int);
            packi16(buf, h);
            buf += 2;
            break;

        case 'H': // 16-bit unsigned
            size += 2;
            H = va_arg(ap, unsigned int);
            packi16(buf, H);
            buf += 2;
            break;

        case 'l': // 32-bit
            size += 4;
            l = va_arg(ap, long int);
            packi32(buf, l);
            buf += 4;
            break;

        case 'L': // 32-bit unsigned
            size += 4;
            L = va_arg(ap, unsigned long int);
            packi32(buf, L);
            buf += 4;
            break;
```

```
        case 'q': // 64-bit
            size += 8;
            q = va_arg(ap, long long int);
            packi64(buf, q);
            buf += 8;
            break;

        case 'Q': // 64-bit unsigned
            size += 8;
            Q = va_arg(ap, unsigned long long int);
            packi64(buf, Q);
            buf += 8;
            break;

        case 'f': // float-16
            size += 2;
            f = (float)va_arg(ap, double); // promoted
            fhold = pack754_16(f); // convert to IEEE 754
            packi16(buf, fhold);
            buf += 2;
            break;

        case 'd': // float-32
            size += 4;
            d = va_arg(ap, double);
            fhold = pack754_32(d); // convert to IEEE 754
            packi32(buf, fhold);
            buf += 4;
            break;

        case 'g': // float-64
            size += 8;
            g = va_arg(ap, long double);
            fhold = pack754_64(g); // convert to IEEE 754
            packi64(buf, fhold);
            buf += 8;
            break;

        case 's': // string
            s = va_arg(ap, char*);
            len = strlen(s);
            size += len + 2;
            packi16(buf, len);
            buf += 2;
            memcpy(buf, s, len);
            buf += len;
            break;
        }
    }

    va_end(ap);
```

```
    return size;
}

/*
** unpack() -- unpack data dictated by the format string into the
**             buffer
**
**   bits |signed   unsigned   float   string
**   -----+----------------------------------
**      8 |   c        C
**     16 |   h        H         f
**     32 |   l        L         d
**     64 |   q        Q         g
**      - |                               s
**
**  (string is extracted based on its stored length, but 's' can be
**  prepended with a max length)
*/
void unpack(unsigned char *buf, char *format, ...)
{
    va_list ap;

    signed char *c;              // 8-bit
    unsigned char *C;

    int *h;                      // 16-bit
    unsigned int *H;

    long int *l;                 // 32-bit
    unsigned long int *L;

    long long int *q;            // 64-bit
    unsigned long long int *Q;

    float *f;                    // floats
    double *d;
    long double *g;
    unsigned long long int fhold;

    char *s;
    unsigned int len, maxstrlen=0, count;

    va_start(ap, format);

    for(; *format != '\0'; format++) {
        switch(*format) {
        case 'c': // 8-bit
            c = va_arg(ap, signed char*);
            if (*buf <= 0x7f) { *c = *buf;} // re-sign
            else { *c = -1 - (unsigned char)(0xffu - *buf); }
            buf++;
            break;
```

```
        case 'C': // 8-bit unsigned
            C = va_arg(ap, unsigned char*);
            *C = *buf++;
            break;

        case 'h': // 16-bit
            h = va_arg(ap, int*);
            *h = unpacki16(buf);
            buf += 2;
            break;

        case 'H': // 16-bit unsigned
            H = va_arg(ap, unsigned int*);
            *H = unpacku16(buf);
            buf += 2;
            break;

        case 'l': // 32-bit
            l = va_arg(ap, long int*);
            *l = unpacki32(buf);
            buf += 4;
            break;

        case 'L': // 32-bit unsigned
            L = va_arg(ap, unsigned long int*);
            *L = unpacku32(buf);
            buf += 4;
            break;

        case 'q': // 64-bit
            q = va_arg(ap, long long int*);
            *q = unpacki64(buf);
            buf += 8;
            break;

        case 'Q': // 64-bit unsigned
            Q = va_arg(ap, unsigned long long int*);
            *Q = unpacku64(buf);
            buf += 8;
            break;

        case 'f': // float
            f = va_arg(ap, float*);
            fhold = unpacku16(buf);
            *f = unpack754_16(fhold);
            buf += 2;
            break;

        case 'd': // float-32
            d = va_arg(ap, double*);
            fhold = unpacku32(buf);
            *d = unpack754_32(fhold);
            buf += 4;
```

```
            break;

        case 'g': // float-64
            g = va_arg(ap, long double*);
            fhold = unpacku64(buf);
            *g = unpack754_64(fhold);
            buf += 8;
            break;

        case 's': // string
            s = va_arg(ap, char*);
            len = unpacku16(buf);
            buf += 2;
            if (maxstrlen > 0 && len > maxstrlen)
                count = maxstrlen - 1;
            else
                count = len;
            memcpy(s, buf, count);
            s[count] = '\0';
            buf += len;
            break;

        default:
            if (isdigit(*format)) { // track max str len
                maxstrlen = maxstrlen * 10 + (*format-'0');
            }
        }

        if (!isdigit(*format)) maxstrlen = 0;
    }

    va_end(ap);
}
```

Và đây là chương trình demo[12] của đoạn code ở trên, nó pack một ít dữ liệu vào `buf` rồi unpack ra các biến. Chú ý rằng khi gọi `unpack()` với tham số string (format specifier "`s`"), khôn ngoan thì nên đặt một giới hạn độ dài tối đa ở phía trước nó để ngăn chặn buffer overrun, ví dụ "`96s`". Hãy cẩn thận khi unpack dữ liệu bạn nhận được qua mạng, một kẻ xấu có thể gửi các gói tin được dựng sai cách nhằm tấn công hệ thống của bạn!

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

// If you have a C23 compiler
#if __STDC_VERSION__ >= 202311L
#include <stdfloat.h>
#else
// Otherwise let's define our own.
// Varies for different architectures! But you're probably:
typedef float float32_t;
```

[12] https://beej.us/guide/bgnet/source/examples/pack2.c

```
typedef double float64_t;
#endif

int main(void)
{
    uint8_t buf[1024];
    int8_t magic;
    int16_t monkeycount;
    int32_t altitude;
    float32_t absurdityfactor;
    char *s = "Great unmitigated Zot!  You've found the Runestaff!";
    char s2[96];
    int16_t packetsize, ps2;

    packetsize = pack(buf, "chhlsf", (int8_t)'B', (int16_t)0,
            (int16_t)37, (int32_t)-5, s, (float32_t)-3490.6677);
    packi16(buf+1, packetsize); // store packet size for kicks

    printf("packet is %" PRId32 " bytes\n", packetsize);

    unpack(buf, "chhl96sf", &magic, &ps2, &monkeycount, &altitude,
            s2, &absurdityfactor);

    printf("'%c' %" PRId32" %" PRId16 " %" PRId32
            " \"%s\" %f\n", magic, ps2, monkeycount,
            altitude, s2, absurdityfactor);
}
```

Dù bạn tự cuộn tay code lấy hay dùng của người khác, có một bộ các thủ tục đóng gói dữ liệu chung là ý hay, để hạn chế bugs, thay vì pack từng bit bằng tay mỗi lần.

Khi đóng gói dữ liệu, định dạng nào là tốt để dùng? Câu hỏi hay. Rất may, RFC 4506[13], the External Data Representation Standard, đã định nghĩa các định dạng nhị phân cho cả đống kiểu khác nhau, như kiểu floating point, kiểu số nguyên, mảng, dữ liệu thô, vân vân. Tôi đề nghị bạn tuân thủ theo đó nếu bạn định tự cuộn dữ liệu lấy. Nhưng không bắt buộc. Cảnh Sát Gói Tin không đang đứng ngay ngoài cửa nhà bạn đâu. Ít nhất, tôi *nghĩ* là họ không.

Dù gì đi nữa, mã hóa dữ liệu bằng cách này hay cách khác trước khi gửi nó đi là cách làm đúng đắn!

7.6 Đứa Con Trai Của Đóng Gói Dữ Liệu

Đóng gói dữ liệu thực sự nghĩa là gì? Trong trường hợp đơn giản nhất, nó nghĩa là bạn sẽ dán lên đó một header với thông tin nhận diện hoặc độ dài gói tin, hoặc cả hai.

Header của bạn nên trông ra sao? Thì, nó chỉ là một ít dữ liệu nhị phân đại diện cho bất cứ gì bạn thấy cần để hoàn thành dự án của mình.

Wow. Nghe mơ hồ ghê.

Được rồi. Ví dụ, giả sử bạn có một chương trình chat nhiều người dùng `SOCK_STREAM`. Khi một người dùng gõ (“nói”) gì đó, có hai thông tin cần được truyền về server: cái gì được nói và ai đã nói.

Tới đây ổn chứ? “Vấn đề ở đâu?”, bạn đang hỏi.

[13] https://tools.ietf.org/html/rfc4506

Vấn đề là các tin nhắn có thể có độ dài khác nhau. Một người tên "tom" có thể nói "Hi", còn người khác tên "Benjamin" có thể nói "Hey guys what is up?"

Nên bạn `send()` tất cả thứ này tới các client khi nó đến. Luồng dữ liệu ra của bạn trông như thế này:

```
t o m H i B e n j a m i n H e y g u y s w h a t i s u p ?
```

Và cứ thế. Làm sao client biết khi nào một tin nhắn bắt đầu và một tin khác kết thúc? Bạn có thể, nếu muốn, làm cho tất cả tin nhắn có cùng độ dài và chỉ cần gọi `sendall()` mà chúng ta đã cài đặt, ở trên. Nhưng như vậy phí băng thông! Chúng ta không muốn `send()` 1024 byte chỉ để "tom" nói "Hi".

Vì vậy chúng ta *đóng gói* dữ liệu vào một cấu trúc header và gói tin nhỏ. Cả client và server đều biết cách pack và unpack (đôi khi được gọi là "marshal" và "unmarshal") dữ liệu này. Đừng nhìn bây giờ, nhưng chúng ta đang bắt đầu định nghĩa một *giao thức* mô tả cách client và server giao tiếp!

Trong trường hợp này, giả sử user name có độ dài cố định 8 ký tự, padding bằng `'\0'`. Rồi giả sử dữ liệu có độ dài biến đổi, tối đa 128 ký tự. Hãy xem thử một cấu trúc gói tin mà chúng ta có thể dùng trong tình huống này:

1. `len` (1 byte, unsigned), tổng độ dài của gói tin, đếm cả user name 8 byte và dữ liệu chat.
2. `name` (8 byte), tên người dùng, NUL-padded nếu cần.
3. `chatdata` (*n* byte), chính dữ liệu, không quá 128 byte. Độ dài của gói tin nên được tính bằng độ dài của dữ liệu này cộng 8 (độ dài của trường name ở trên).

Tại sao tôi chọn giới hạn 8 byte và 128 byte cho các trường? Tôi bịa ra từ không khí, giả định chúng sẽ đủ dài. Có thể, dù vậy, 8 byte là quá hạn chế với nhu cầu của bạn, và bạn có thể có trường name 30 byte, hoặc bất cứ gì. Chọn lựa là của bạn.

Dùng định nghĩa gói tin ở trên, gói tin đầu tiên sẽ gồm thông tin sau (ở hex và ASCII):

```
   0A     74 6F 6D 00 00 00 00 00      48 69
(length)  T  o  m    (padding)        H  i
```

Và gói thứ hai tương tự:

```
   18     42 65 6E 6A 61 6D 69 6E      48 65 79 20 67 75 79 73 20 77 ...
(length)  B  e  n  j  a  m  i  n       H  e  y     g  u  y  s     w  ...
```

(Độ dài được lưu ở Network Byte Order, dĩ nhiên. Trong trường hợp này, nó chỉ có một byte nên không quan trọng, nhưng nói chung bạn sẽ muốn tất cả số nguyên nhị phân của mình được lưu ở Network Byte Order trong các gói tin.)

Khi bạn gửi dữ liệu này, bạn nên an toàn và dùng một lệnh tương tự `sendall()` ở trên, để bạn biết tất cả dữ liệu đã được gửi, kể cả khi cần nhiều lời gọi `send()` để đưa hết ra.

Tương tự, khi bạn nhận dữ liệu này, bạn cần làm thêm một ít việc. Để an toàn, bạn nên giả định rằng bạn có thể nhận được một phần gói tin (ví dụ có khi chúng ta nhận được "`18 42 65 6E 6A`" từ Benjamin ở trên, nhưng chỉ nhận được chừng đó trong lời gọi `recv()` này). Chúng ta cần gọi `recv()` lặp đi lặp lại cho đến khi gói tin được nhận đầy đủ.

Nhưng làm sao? Thì, chúng ta biết tổng số byte cần nhận để gói tin hoàn chỉnh, vì con số đó được dán ở đầu gói tin. Chúng ta cũng biết kích thước gói tin tối đa là 1+8+128, tức 137 byte (vì đó là cách chúng ta định nghĩa gói tin).

Thật ra có vài thứ bạn có thể làm ở đây. Vì bạn biết mỗi gói tin bắt đầu bằng độ dài, bạn có thể gọi `recv()` chỉ để lấy độ dài gói tin. Rồi sau khi có nó, bạn có thể gọi nó lần nữa chỉ định chính xác độ dài còn lại của gói tin (có thể lặp lại để lấy hết dữ liệu) cho đến khi có gói tin hoàn chỉnh. Ưu điểm của cách này là bạn

chỉ cần một buffer đủ lớn cho một gói tin, còn nhược điểm là bạn phải gọi `recv()` ít nhất hai lần để lấy hết dữ liệu.

Một lựa chọn khác là chỉ cần gọi `recv()` và nói rằng số byte bạn sẵn sàng nhận là số byte tối đa trong một gói tin. Rồi bất cứ gì bạn nhận được, dán nó vào cuối buffer, và cuối cùng kiểm tra xem gói tin có hoàn chỉnh chưa. Dĩ nhiên, bạn có thể nhận được một phần của gói tin kế tiếp, nên bạn cần có chỗ cho phần đó.

Cái bạn có thể làm là khai báo một mảng đủ lớn cho hai gói tin. Đây là mảng công tác nơi bạn sẽ dựng lại các gói tin khi chúng đến.

Mỗi lần bạn `recv()` dữ liệu, bạn sẽ append nó vào work buffer và kiểm tra xem gói tin đã hoàn chỉnh chưa. Tức là, số byte trong buffer lớn hơn hoặc bằng độ dài được chỉ định trong header (+1, vì độ dài trong header không bao gồm byte cho chính độ dài đó). Nếu số byte trong buffer nhỏ hơn 1, gói tin rõ ràng là chưa hoàn chỉnh. Bạn phải làm trường hợp đặc biệt cho chuyện này, vì byte đầu tiên là rác và bạn không thể dựa vào nó để lấy đúng độ dài gói tin.

Khi gói tin đã hoàn chỉnh, bạn có thể làm gì với nó tùy ý. Dùng nó rồi xóa khỏi work buffer.

Hú! Bạn có đang tung hứng hết mấy thứ đó trong đầu không? Đây là cú đấm thứ hai trong combo một-hai: bạn có thể đã đọc qua phần cuối của một gói tin và sang gói kế trong một lời gọi `recv()` duy nhất. Tức là, bạn có work buffer với một gói tin hoàn chỉnh, và một phần chưa hoàn chỉnh của gói tin kế tiếp! Chết tiệt. (Nhưng đây là lý do bạn làm work buffer đủ lớn để chứa *hai* gói tin, phòng khi chuyện này xảy ra!)

Vì bạn biết độ dài của gói tin đầu tiên từ header, và bạn đã theo dõi số byte trong work buffer, bạn có thể trừ ra và tính được bao nhiêu byte trong work buffer thuộc về gói tin thứ hai (chưa hoàn chỉnh). Khi đã xử lý xong gói đầu tiên, bạn có thể xóa nó khỏi work buffer và dời phần gói thứ hai chưa hoàn chỉnh xuống đầu buffer để mọi thứ sẵn sàng cho lời gọi `recv()` kế tiếp.

(Một số độc giả sẽ chú ý rằng việc thật sự dời phần gói thứ hai chưa hoàn chỉnh về đầu work buffer mất thời gian, và chương trình có thể được code để không cần làm vậy bằng cách dùng circular buffer. Không may cho số còn lại trong các bạn, một cuộc thảo luận về circular buffer vượt ra ngoài phạm vi bài viết này. Nếu vẫn tò mò, tóm lấy một cuốn sách cấu trúc dữ liệu và đi từ đó.)

Tôi chưa bao giờ nói là dễ đâu nhé. À ừ, tôi có nói nó dễ. Và nó dễ mà; bạn chỉ cần luyện tập thôi, rồi khá nhanh nó sẽ tự đến với bạn một cách tự nhiên. Tôi thề bằng thanh kiếm Excalibur đấy!

7.7 Gói Tin Broadcast: Hello, World!

Tới giờ, hướng dẫn này nói về việc gửi dữ liệu từ một máy sang một máy khác. Nhưng có thể, tôi khẳng định, rằng bạn có thể, với đúng quyền hạn, gửi dữ liệu tới nhiều máy *cùng một lúc*!

Với UDP (chỉ UDP, không phải TCP) và IPv4 chuẩn, chuyện này được làm qua một cơ chế gọi là *broadcasting*. Với IPv6, broadcasting không được hỗ trợ, bạn phải dùng kỹ thuật thường là vượt trội hơn gọi là *multicasting*, mà đáng tiếc tôi sẽ không bàn tới lúc này. Nhưng thôi đừng mơ mộng về tương lai nữa, chúng ta đang kẹt trong hiện tại 32-bit.

Khoan đã! Bạn không thể chạy đi broadcast lung tung được; bạn phải đặt tùy chọn socket `SO_BROADCAST` trước khi có thể gửi một gói tin broadcast ra mạng. Nó giống như mấy cái nắp nhựa nhỏ người ta đậy lên công tắc phóng tên lửa vậy! Quyền năng trong tay bạn lớn tới mức đó đấy!

Nhưng nghiêm túc nhé, có một nguy hiểm khi dùng gói tin broadcast, đó là: mọi hệ thống nhận được gói tin broadcast phải bóc hết các lớp vỏ hành đóng gói dữ liệu cho đến khi tìm ra dữ liệu được gửi đến port nào. Rồi nó bàn giao dữ liệu hoặc vứt đi. Trong cả hai trường hợp, đó là nhiều việc cho mỗi máy nhận gói tin broadcast, và vì đó là tất cả các máy trên mạng local, có thể rất nhiều máy làm rất nhiều việc không cần thiết. Khi game Doom mới ra, đây là một lời than phiền về network code của nó.

Giờ, có hơn một cách lột da mèo[14]... khoan đã. Có thật là có hơn một cách lột da mèo không? Câu thành

[14]Nói cho rõ, tôi yêu mèo. Chúng là nhất. Tôi đã có nhiều người bạn mèo yêu quý qua năm tháng. Dù tôi thừa nhận một số người phản đối câu thành ngữ hình tượng rùng rợn này, với nguồn gốc từ nguyên đã thất lạc theo thời gian, tôi nghĩ phần này của hướng

ngữ kiểu gì vậy? Ồ, tương tự, có hơn một cách gửi một gói tin broadcast. Vậy, đi vào phần thịt và khoai tây của vấn đề: bạn chỉ định địa chỉ đích cho một tin nhắn broadcast ra sao? Có hai cách phổ biến:

1. Gửi dữ liệu tới địa chỉ broadcast của một subnet cụ thể. Đây là network number của subnet đó với tất cả các bit một được bật ở phần host của địa chỉ. Ví dụ, ở nhà mạng của tôi là `192.168.1.0`, netmask là `255.255.255.0`, nên byte cuối của địa chỉ là số host của tôi (vì ba byte đầu, theo netmask, là network number). Nên địa chỉ broadcast của tôi là `192.168.1.255`. Trên Unix, lệnh `ifconfig` thật ra sẽ cho bạn tất cả dữ liệu này. (Nếu bạn tò mò, logic bitwise để lấy địa chỉ broadcast của mình là `network_number` OR (NOT `netmask`).) Bạn có thể gửi loại gói tin broadcast này tới mạng remote cũng như mạng local, nhưng bạn có rủi ro gói tin bị router của đích đến vứt đi. (Nếu họ không vứt đi, thì một con smurf ngẫu nhiên nào đó có thể bắt đầu làm ngập LAN của họ bằng traffic broadcast.)
2. Gửi dữ liệu tới địa chỉ broadcast "toàn cục". Đây là `255.255.255.255`, còn gọi là `INADDR_BROADCAST`. Nhiều máy sẽ tự động AND bitwise cái này với network number của bạn để chuyển nó thành địa chỉ broadcast của mạng, nhưng một số thì không. Tùy thôi. Router không chuyển tiếp loại gói tin broadcast này ra khỏi mạng local của bạn, khá là trớ trêu.

Vậy chuyện gì xảy ra nếu bạn thử gửi dữ liệu trên địa chỉ broadcast mà không đặt tùy chọn socket `SO_BROADCAST` trước? Hãy khởi động mấy chương trình `talker` và `listener` ngon lành cũ và xem chuyện gì xảy ra.

```
$ talker 192.168.1.2 foo
sent 3 bytes to 192.168.1.2
$ talker 192.168.1.255 foo
sendto: Permission denied
$ talker 255.255.255.255 foo
sendto: Permission denied
```

Vâng, nó không vui chút nào... vì chúng ta đã không đặt tùy chọn socket `SO_BROADCAST`. Đặt cái đó, và giờ bạn có thể `sendto()` tới bất cứ đâu bạn muốn!

Thật ra, đó là sự *khác biệt duy nhất* giữa một ứng dụng UDP có thể broadcast và một cái không thể. Vậy hãy lấy ứng dụng `talker` cũ và thêm một đoạn đặt tùy chọn socket `SO_BROADCAST`. Chúng ta sẽ gọi chương trình này là `broadcaster.c` [15]:

```
/*
** broadcaster.c -- a datagram "client" like talker.c, except
**                  this one can broadcast
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define SERVERPORT 4950    // the port users will be connecting to
```

dẫn được phục vụ tốt nhất bằng việc dùng nó.

[15] https://beej.us/guide/bgnet/source/examples/broadcaster.c

```
int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in their_addr; // connector's address info
    struct hostent *he;
    int numbytes;
    int broadcast = 1;
    //char broadcast = '1'; // if that doesn't work, try this

    if (argc != 3) {
        fprintf(stderr,"usage: broadcaster hostname message\n");
        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) {  // get the host info
        perror("gethostbyname");
        exit(1);
    }

    if ((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    // this call is what allows broadcast packets to be sent:
    if (setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &broadcast,
        sizeof broadcast) == -1) {
        perror("setsockopt (SO_BROADCAST)");
        exit(1);
    }

    their_addr.sin_family = AF_INET;     // host byte order
    their_addr.sin_port = htons(SERVERPORT); // network byte order
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(their_addr.sin_zero, '\0', sizeof their_addr.sin_zero);

    numbytes = sendto(sockfd, argv[2], strlen(argv[2]), 0,
             (struct sockaddr *)&their_addr, sizeof their_addr);

    if (numbytes == -1) {
        perror("sendto");
        exit(1);
    }

    printf("sent %d bytes to %s\n", numbytes,
        inet_ntoa(their_addr.sin_addr));

    close(sockfd);

    return 0;
}
```

Cái gì khác biệt giữa cái này và tình huống UDP client/server “bình thường”? Không có gì! (Ngoại trừ việc

client được phép gửi gói tin broadcast trong trường hợp này.) Vậy, cứ chạy chương trình UDP `listener` cũ trong một cửa sổ, và `broadcaster` trong một cửa sổ khác. Giờ bạn có thể làm tất cả những send mà đã thất bại ở trên.

```
$ broadcaster 192.168.1.2 foo
sent 3 bytes to 192.168.1.2
$ broadcaster 192.168.1.255 foo
sent 3 bytes to 192.168.1.255
$ broadcaster 255.255.255.255 foo
sent 3 bytes to 255.255.255.255
```

Và bạn sẽ thấy `listener` phản hồi rằng nó đã nhận được gói tin. (Nếu `listener` không phản hồi, có thể là vì nó được bind vào một địa chỉ IPv6. Thử đổi `AF_INET6` trong `listener.c` thành `AF_INET` để ép IPv4.)

À, cái này hơi phấn khích đấy. Nhưng giờ khởi động `listener` trên một máy khác bên cạnh bạn cùng mạng sao cho bạn có hai bản đang chạy, mỗi máy một bản, và chạy `broadcaster` lần nữa với địa chỉ broadcast của bạn... Ê! Cả hai `listener` đều nhận được gói tin mặc dù bạn chỉ gọi `sendto()` một lần! Ngầu!

Nếu `listener` nhận được dữ liệu bạn gửi trực tiếp tới nó, nhưng không nhận được dữ liệu trên địa chỉ broadcast, có thể là vì bạn có một firewall trên máy local đang chặn các gói tin. (Đúng vậy, Pat và Bapper, cảm ơn các bạn đã nhận ra trước tôi rằng đó là lý do code mẫu của tôi không chạy. Tôi đã bảo các bạn là tôi sẽ nhắc tên các bạn trong hướng dẫn, và đây các bạn. Vậy đó, *nyah*.)

Lại nữa, hãy cẩn thận với gói tin broadcast. Vì mọi máy trên LAN đều bị ép xử lý gói tin dù nó có `recvfrom()` hay không, nó có thể tạo khá nhiều tải cho toàn bộ mạng máy tính. Chúng chắc chắn là thứ cần dùng tiết kiệm và đúng lúc.

Chapter 8

Những Câu Hỏi Thường Gặp

Tôi kiếm những header file đó ở đâu?

Nếu bạn chưa có chúng trên hệ thống, thì chắc bạn không cần chúng. Kiểm tra sách hướng dẫn cho nền tảng cụ thể của bạn. Nếu bạn đang build cho Windows, bạn chỉ cần `#include <winsock.h>`.

Tôi phải làm gì khi `bind()` báo "Address already in use"?

Bạn phải dùng `setsockopt()` với tùy chọn `SO_REUSEADDR` trên socket đang lắng nghe. Xem phần về `bind()` và phần về `select()` để có ví dụ.

Làm sao lấy danh sách socket đang mở trên hệ thống?

Dùng `netstat`. Kiểm tra `man` page để biết chi tiết đầy đủ, nhưng bạn sẽ có output tốt chỉ bằng cách gõ:

```
$ netstat
```

Khó khăn duy nhất là xác định socket nào gắn với chương trình nào. `:-)`

Làm sao xem routing table?

Chạy lệnh `route` (trong `/sbin` trên hầu hết Linux) hoặc lệnh `netstat -r`. Hoặc lệnh `ip route`.

Làm sao chạy chương trình client và server nếu tôi chỉ có một máy tính? Tôi không cần một mạng để viết chương trình mạng à?

May cho bạn, hầu như mọi máy đều có triển khai "thiết bị" mạng loopback nằm trong kernel và giả vờ là một card mạng. (Đây là interface được liệt kê là "`lo`" trong routing table.)

Giả sử bạn đang login vào một máy tên "`goat`". Chạy client trong một cửa sổ và server trong cửa sổ khác. Hoặc khởi động server ở chế độ nền ("`server &`") và chạy client trong cùng cửa sổ. Cái được của loopback device là bạn có thể `client goat` hoặc `client localhost` (vì "`localhost`" có khả năng được định nghĩa trong file `/etc/hosts` của bạn) và bạn sẽ có client nói chuyện với server mà không cần mạng!

Nói gọn, không cần thay đổi gì trong code để nó chạy được trên một máy đơn lẻ không nối mạng! Hoan hô!

Làm sao biết đầu bên kia đã đóng kết nối?

Bạn có thể biết vì `recv()` sẽ trả về `0`.

Làm sao cài đặt tiện ích "ping"? ICMP là gì? Tôi tìm hiểu thêm về raw socket và `SOCK_RAW` ở đâu?

Tất cả câu hỏi về raw socket của bạn sẽ được trả lời trong sách UNIX Network Programming của W. Richard Stevens. Cũng vậy, xem trong thư mục `ping/` con trong source code của UNIX Network Programming của Stevens, có sẵn online[1].

Làm sao thay đổi hoặc rút ngắn timeout của một lời gọi `connect()`?

Thay vì đưa bạn chính xác cùng câu trả lời mà W. Richard Stevens sẽ đưa, tôi sẽ chỉ bạn tới `lib/connect_nonb.c` trong source code UNIX Network Programming[2].

Tóm tắt là bạn tạo một socket descriptor bằng `socket()`, đặt nó thành non-blocking, gọi `connect()`, và nếu mọi thứ suôn sẻ `connect()` sẽ trả về `-1` ngay lập tức và `errno` sẽ được gán thành `EINPROGRESS`. Rồi bạn gọi `select()` với bất kỳ timeout nào bạn muốn, truyền socket descriptor vào cả tập đọc lẫn tập ghi. Nếu nó không timeout, nghĩa là lời gọi `connect()` đã hoàn thành. Lúc này, bạn sẽ phải dùng `getsockopt()` với tùy chọn `SO_ERROR` để lấy giá trị trả về từ lời gọi `connect()`, giá trị đó sẽ bằng không nếu không có lỗi.

Cuối cùng, có lẽ bạn sẽ muốn đặt socket trở lại chế độ blocking trước khi bắt đầu truyền dữ liệu qua nó.

Chú ý rằng cách này có thêm cái được là cho phép chương trình của bạn làm việc khác trong lúc đang connect. Ví dụ, bạn có thể đặt timeout thấp, như 500 ms, và cập nhật một chỉ báo trên màn hình mỗi lần timeout, rồi gọi `select()` lần nữa. Khi bạn đã gọi `select()` và timeout, ví dụ 20 lần, bạn biết đã đến lúc bỏ cuộc với kết nối này.

Như tôi đã nói, xem source của Stevens để có ví dụ tuyệt vời tuyệt đối.

Làm sao build cho Windows?

Trước hết, xóa Windows và cài Linux hoặc BSD. `};-)`. Không, thật ra, chỉ cần xem phần về build cho Windows ở phần giới thiệu.

Làm sao build cho Solaris/SunOS? Tôi cứ bị lỗi linker khi cố biên dịch!

Lỗi linker xảy ra vì mấy cái máy Sun không tự động compile chung với các thư viện socket. Xem phần về build cho Solaris/SunOS ở phần giới thiệu để có ví dụ về cách làm việc này.

Tại sao `select()` cứ thoát ra khi có signal?

Signal có xu hướng làm cho các system call đang bị block trả về `-1` với `errno` được gán thành `EINTR`. Khi bạn đặt một signal handler bằng `sigaction()`, bạn có thể đặt cờ `SA_RESTART`, được cho là sẽ khởi động lại system call sau khi nó bị ngắt.

Đương nhiên, cái này không phải lúc nào cũng hiệu quả.

Giải pháp ưa thích của tôi cho chuyện này liên quan đến một câu lệnh `goto`. Bạn biết chuyện này khiến các giáo sư của bạn cực kỳ khó chịu, nên cứ làm đi!

```
select_restart:
if ((err = select(fdmax+1, &readfds, NULL, NULL, NULL)) == -1) {
    if (errno == EINTR) {
        // some signal just interrupted us, so restart
        goto select_restart;
    }
    // handle the real error here:
    perror("select");
}
```

[1] http://www.unpbook.com/src.html
[2] http://www.unpbook.com/src.html

Chắc rồi, bạn không *cần* dùng `goto` trong trường hợp này; bạn có thể dùng cấu trúc khác để điều khiển. Nhưng tôi nghĩ câu lệnh `goto` thật ra sạch hơn.

Làm sao cài đặt timeout cho một lời gọi `recv()`?

Dùng `select()`! Nó cho phép bạn chỉ định tham số timeout cho các socket descriptor mà bạn đang muốn đọc từ đó. Hoặc, bạn có thể gói toàn bộ chức năng vào một hàm duy nhất, như thế này:

```
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>

int recvtimeout(int s, char *buf, int len, int timeout)
{
    fd_set fds;
    int n;
    struct timeval tv;

    // set up the file descriptor set
    FD_ZERO(&fds);
    FD_SET(s, &fds);

    // set up the struct timeval for the timeout
    tv.tv_sec = timeout;
    tv.tv_usec = 0;

    // wait until timeout or data received
    n = select(s+1, &fds, NULL, NULL, &tv);
    if (n == 0) return -2; // timeout!
    if (n == -1) return -1; // error

    // data must be here, so do a normal recv()
    return recv(s, buf, len, 0);
}
.
.
.
// Sample call to recvtimeout():
n = recvtimeout(s, buf, sizeof buf, 10); // 10 second timeout

if (n == -1) {
    // error occurred
    perror("recvtimeout");
}
else if (n == -2) {
    // timeout occurred
} else {
    // got some data in buf
}
.
.
.
```

Chú ý rằng `recvtimeout()` trả về `-2` trong trường hợp timeout. Sao không trả về `0`? Nếu bạn còn nhớ, giá trị trả về `0` trên lời gọi `recv()` nghĩa là đầu bên kia đã đóng kết nối. Nên giá trị trả về đó đã có chỗ, và `-1` nghĩa là "lỗi", nên tôi chọn `-2` làm chỉ báo timeout của mình.

Làm sao mã hóa hoặc nén dữ liệu trước khi gửi qua socket?

Một cách dễ để mã hóa là dùng SSL (secure sockets layer), nhưng cái đó vượt ra ngoài phạm vi hướng dẫn này. (Xem dự án OpenSSL[3] để biết thêm.)

Nhưng giả sử bạn muốn cắm vào hoặc tự cài đặt hệ thống nén hay mã hóa của mình, thì đó chỉ là chuyện nghĩ về dữ liệu của mình như đang chạy qua một chuỗi bước giữa hai đầu. Mỗi bước thay đổi dữ liệu theo một cách nào đó.

1. server đọc dữ liệu từ file (hoặc ở đâu đó)
2. server mã hóa/nén dữ liệu (bạn thêm phần này)
3. server `send()` dữ liệu đã mã hóa

Giờ hướng ngược lại:

1. client `recv()` dữ liệu đã mã hóa
2. client giải mã/giải nén dữ liệu (bạn thêm phần này)
3. client ghi dữ liệu ra file (hoặc ở đâu đó)

Nếu bạn định nén và mã hóa, nhớ nén trước. `:-)`

Miễn là client đảo ngược đúng những gì server làm, dữ liệu sẽ ổn ở cuối bất kể bạn thêm bao nhiêu bước trung gian.

Vậy tất cả những gì bạn cần làm để dùng code của tôi là tìm vị trí giữa chỗ dữ liệu được đọc và chỗ dữ liệu được gửi (bằng `send()`) qua mạng, và nhét vào đó một đoạn code làm việc mã hóa.

Cái "`PF_INET`" mà tôi cứ thấy là gì vậy? Nó có liên quan đến `AF_INET` không?

Có, có liên quan đấy. Xem phần về `socket()` để biết chi tiết.

Làm sao viết một server nhận lệnh shell từ client và thực thi chúng?

Để đơn giản, giả sử client `connect()`, `send()` và `close()` kết nối (tức là không có system call nào theo sau mà client không kết nối lại).

Quy trình mà client làm theo là:

1. `connect()` tới server
2. `send("/sbin/ls > /tmp/client.out")`
3. `close()` kết nối

Trong lúc đó, server xử lý dữ liệu và thực thi nó:

1. `accept()` kết nối từ client
2. `recv(str)` chuỗi lệnh
3. `close()` kết nối
4. `system(str)` để chạy lệnh

Coi chừng! Cho server thực thi những gì client bảo thì chẳng khác gì cho quyền truy cập shell từ xa, và người ta có thể làm nhiều trò với tài khoản của bạn khi kết nối vào server. Ví dụ, trong ví dụ trên, lỡ client gửi "`rm -rf ~`" thì sao? Nó xóa sạch mọi thứ trong tài khoản của bạn, thế đấy!

Nên bạn khôn ra, và bạn ngăn client dùng bất cứ gì trừ một vài tiện ích bạn biết là an toàn, như tiện ích `foobar`:

[3] https://www.openssl.org/

```
if (!strncmp(str, "foobar", 6)) {
    sprintf(sysstr, "%s > /tmp/server.out", str);
    system(sysstr);
}
```

Nhưng bạn vẫn chưa an toàn, đáng tiếc: lỡ client nhập "`foobar; rm -rf ~`" thì sao? Điều an toàn nhất cần làm là viết một thủ tục nhỏ đặt ký tự escape ("`\`") trước tất cả ký tự không phải chữ và số (bao gồm cả khoảng trắng, nếu phù hợp) trong các tham số cho lệnh.

Như bạn thấy, bảo mật là vấn đề khá lớn khi server bắt đầu thực thi những thứ client gửi.

Tôi gửi cả đống dữ liệu, nhưng khi `recv()`, nó chỉ nhận được 536 byte hoặc 1460 byte mỗi lần. Nhưng nếu tôi chạy trên máy local, nó nhận toàn bộ dữ liệu cùng lúc. Chuyện gì đang xảy ra?

Bạn đang chạm đến MTU, kích thước tối đa mà môi trường vật lý có thể xử lý. Trên máy local, bạn đang dùng thiết bị loopback có thể xử lý 8K hoặc hơn không thành vấn đề. Nhưng trên Ethernet, vốn chỉ có thể xử lý 1500 byte kèm header, bạn chạm giới hạn đó. Qua modem, với MTU 576 (lại kèm header), bạn chạm giới hạn còn thấp hơn.

Bạn phải đảm bảo toàn bộ dữ liệu đang được gửi, trước hết. (Xem hàm `sendall()` để biết chi tiết.) Khi bạn chắc chuyện đó, thì bạn cần gọi `recv()` trong vòng lặp cho đến khi tất cả dữ liệu của bạn được đọc.

Đọc phần Đứa Con Trai Của Đóng Gói Dữ Liệu để biết chi tiết về việc nhận đầy đủ các gói tin dùng nhiều lời gọi `recv()`.

Tôi dùng máy Windows và không có system call `fork()` hay bất kỳ kiểu `struct sigaction` nào. Phải làm sao?

Nếu chúng tồn tại ở đâu đó, chúng sẽ nằm trong các thư viện POSIX có thể đã đi kèm với compiler của bạn. Vì tôi không có máy Windows, tôi thật sự không thể cho bạn câu trả lời, nhưng tôi nhớ mang máng là Microsoft có một lớp tương thích POSIX và đó là nơi `fork()` có thể nằm. (Và có thể cả `sigaction` nữa.)

Tìm trong phần help đi kèm VC++ từ khóa "fork" hoặc "POSIX" xem có manh mối gì không.

Nếu cái đó hoàn toàn không chạy, vứt hết cái `fork()`/`sigaction` và thay bằng thứ tương đương của Win32: `CreateProcess()`. Tôi không biết cách dùng `CreateProcess()`, nó nhận cả tỷ tham số, nhưng chắc nó được bao phủ trong tài liệu đi kèm VC++.

Tôi ở sau firewall, làm sao cho người ngoài firewall biết địa chỉ IP của tôi để họ có thể kết nối tới máy tôi?

Đáng tiếc, mục đích của firewall là ngăn người ở ngoài firewall kết nối tới máy bên trong firewall, nên cho phép họ làm vậy về cơ bản là bị coi là vi phạm bảo mật.

Không có nghĩa là mọi thứ đều thua cuộc. Một là, bạn vẫn thường có thể `connect()` qua firewall nếu nó đang làm kiểu masquerading hoặc NAT hay gì đó tương tự. Chỉ cần thiết kế chương trình sao cho bạn luôn là người chủ động khởi tạo kết nối, và bạn sẽ ổn.

Nếu cái đó không thỏa đáng, bạn có thể nhờ mấy ông sysadmin đục một lỗ trên firewall để người ta có thể kết nối tới bạn. Firewall có thể forward tới bạn qua phần mềm NAT của nó, hoặc qua proxy hay gì đó tương tự.

Xin lưu ý rằng một lỗ thủng trên firewall không phải chuyện đùa. Bạn phải đảm bảo mình không cho kẻ xấu truy cập vào mạng nội bộ; nếu bạn là người mới, khó hơn nhiều để làm phần mềm an toàn so với tưởng tượng của bạn.

Đừng làm sysadmin của bạn giận tôi. `;-)`

Làm sao viết một packet sniffer? Làm sao đặt Ethernet interface của tôi vào chế độ promiscuous?

Cho những ai chưa biết, khi một card mạng ở "chế độ promiscuous", nó sẽ chuyển TẤT CẢ gói tin cho hệ điều hành, không chỉ những gói tin có địa chỉ đến máy cụ thể này. (Chúng ta đang nói về địa chỉ tầng Ethernet ở đây, không phải địa chỉ IP, nhưng vì Ethernet ở tầng dưới IP, tất cả địa chỉ IP thực chất cũng được forward luôn. Xem phần Chuyện Nhảm Cấp Thấp và Lý Thuyết Mạng để biết thêm.)

Đây là cơ sở cách một packet sniffer hoạt động. Nó đặt interface vào chế độ promiscuous, rồi OS nhận mọi gói tin đi qua trên dây. Bạn sẽ có một loại socket nào đó để đọc dữ liệu này.

Đáng tiếc, câu trả lời cho câu hỏi này khác nhau tùy nền tảng, nhưng nếu bạn Google từ khóa, ví dụ, "windows promiscuous ioctl" chắc bạn sẽ tới được đâu đó. Cho Linux, có vẻ có một chủ đề Stack Overflow hữu ích[4] nữa.

Làm sao đặt giá trị timeout tùy chỉnh cho một socket TCP hoặc UDP?

Cái này tùy hệ thống của bạn. Bạn có thể tìm trên mạng `SO_RCVTIMEO` và `SO_SNDTIMEO` (để dùng với `setsockopt()`) xem hệ thống của bạn có hỗ trợ chức năng như vậy không.

Trang man của Linux đề nghị dùng `alarm()` hoặc `setitimer()` thay thế.

Làm sao biết port nào có sẵn để dùng? Có danh sách số port "chính thức" không?

Thường thì đây không phải vấn đề. Nếu bạn đang viết, ví dụ, một web server, thì nên dùng port 80 nổi tiếng cho phần mềm của mình. Nếu bạn đang viết một server chuyên biệt của riêng mình, thì chọn một port ngẫu nhiên (nhưng lớn hơn 1023) và thử.

Nếu port đã được dùng, bạn sẽ bị lỗi "Address already in use" khi cố `bind()`. Chọn port khác. (Nên cho phép người dùng phần mềm của bạn chỉ định một port khác qua file config hoặc tùy chọn dòng lệnh.)

Có một danh sách số port chính thức[5] được duy trì bởi Internet Assigned Numbers Authority (IANA). Chỉ vì cái gì đó (lớn hơn 1023) có trong danh sách đó không có nghĩa là bạn không thể dùng port đó. Ví dụ, DOOM của Id Software dùng cùng port với "mdqs", bất kể cái đó là gì. Tất cả những gì quan trọng là không ai khác *trên cùng một máy* đang dùng port đó khi bạn muốn dùng nó.

[4] https://stackoverflow.com/questions/21323023/
[5] https://www.iana.org/assignments/port-numbers

Chapter 9

Man Pages

Trong thế giới Unix, có một đống sách hướng dẫn. Chúng có những phần nhỏ mô tả từng hàm riêng lẻ mà bạn có sẵn để dùng.

Dĩ nhiên, "manual" sẽ là từ quá dài để gõ. Ý tôi là, không ai trong thế giới Unix, kể cả tôi, thích gõ nhiều đến thế. Thật ra tôi có thể nói dài tràng giang đại hải về chuyện tôi thích ngắn gọn đến mức nào, nhưng thay vào đó tôi sẽ ngắn gọn và không làm bạn chán với những bài diễn văn lê thê về chuyện tôi cực kỳ kinh ngạc ưa chuộng sự ngắn gọn đến cỡ nào trong hầu hết mọi hoàn cảnh ở tính tổng thể trọn vẹn của chúng.

[Tiếng vỗ tay]

Cảm ơn. Ý tôi muốn nói là, các trang này được gọi là "man page" trong thế giới Unix, và tôi đã đưa vào đây biến thể cắt gọn của riêng tôi để bạn đọc thư giãn. Vấn đề là, nhiều trong số các hàm này tổng quát hơn nhiều so với tôi tiết lộ, nhưng tôi chỉ sẽ trình bày các phần liên quan đến Lập Trình Socket Internet.

Nhưng khoan! Đó chưa phải là tất cả những gì sai với man page của tôi:

- Chúng không đầy đủ và chỉ trình bày phần căn bản từ hướng dẫn.
- Có rất nhiều man page khác ngoài đời thực hơn cái này.
- Chúng khác với những cái trên hệ thống của bạn.
- Các file header có thể khác cho một số hàm nhất định trên hệ thống của bạn.
- Tham số hàm có thể khác cho một số hàm nhất định trên hệ thống của bạn.

Nếu bạn muốn thông tin thật, kiểm tra man page Unix cục bộ của bạn bằng cách gõ `man gì_đó`, trong đó "gì_đó" là thứ bạn cực kỳ quan tâm tới, ví dụ "`accept`". (Tôi chắc Microsoft Visual Studio có thứ gì đó tương tự trong phần help của họ. Nhưng "man" tốt hơn vì nó ngắn gọn hơn "help" một byte. Unix lại thắng!)

Vậy, nếu chúng thiếu sót như thế, tại sao lại đưa chúng vào Hướng Dẫn? Có vài lý do, nhưng lý do tốt nhất là (a) những phiên bản này được nhắm cụ thể vào lập trình mạng và dễ tiêu hóa hơn bản thật, và (b) những phiên bản này có ví dụ!

À! Và nói về ví dụ, tôi có xu hướng không đưa tất cả phần kiểm tra lỗi vào vì nó thật sự làm tăng độ dài của code. Nhưng bạn tuyệt đối nên kiểm tra lỗi gần như mỗi khi bạn gọi bất kỳ system call nào trừ khi bạn hoàn toàn 100% chắc chắn nó sẽ không thất bại, và có lẽ bạn vẫn nên làm vậy kể cả khi đó!

9.1 `accept()`

Nhận một kết nối đi tới trên socket đang lắng nghe

Synopsis

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

Description

Khi bạn đã mất công lấy một socket `SOCK_STREAM` và cấu hình nó để nhận kết nối đi tới với `listen()`, rồi bạn gọi `accept()` để thực sự có được một socket descriptor mới dùng cho các giao tiếp tiếp theo với client vừa kết nối.

Socket cũ mà bạn đang dùng để lắng nghe vẫn còn đó, và sẽ được dùng cho các lời gọi `accept()` tiếp theo khi chúng đến.

Tham số	Mô tả
`s`	Socket descriptor đang `listen()`.
`addr`	Cái này được điền địa chỉ của bên đang kết nối tới bạn.
`addrlen`	Cái này được điền `sizeof()` của struct trả về trong tham số `addr`. Bạn có thể yên tâm bỏ qua nó nếu bạn giả sử mình nhận được một `struct sockaddr_in`, điều mà bạn biết vì đó là kiểu bạn đã truyền vào cho `addr`.

`accept()` thường sẽ block, và bạn có thể dùng `select()` để dòm socket descriptor đang lắng nghe trước để xem nó có "sẵn sàng đọc" không. Nếu có, thì có một kết nối mới đang đợi được `accept()`! Yay! Hoặc, bạn có thể đặt cờ `O_NONBLOCK` trên socket đang lắng nghe bằng `fcntl()`, và khi đó nó sẽ không bao giờ block, thay vào đó nó chọn trả về `-1` với `errno` được gán thành `EWOULDBLOCK`.

Socket descriptor do `accept()` trả về là một socket descriptor thực thụ, đang mở và đang kết nối tới host remote. Bạn phải `close()` nó khi dùng xong.

Return Value

`accept()` trả về socket descriptor vừa kết nối, hoặc `-1` nếu lỗi, với `errno` được gán phù hợp.

Example

```
struct sockaddr_storage their_addr;
socklen_t addr_size;
struct addrinfo hints, *res;
int sockfd, new_fd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, MYPORT, &hints, &res);
```

```
// make a socket, bind it, and listen on it:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);
listen(sockfd, BACKLOG);

// now accept an incoming connection:

addr_size = sizeof their_addr;
new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);

// ready to communicate on socket descriptor new_fd!
```

See Also

`socket()`, `getaddrinfo()`, `listen()`, `struct sockaddr_in`

9.2 `bind()`

Gắn socket với một địa chỉ IP và số port

Synopsis

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

Description

Khi một máy remote muốn kết nối tới chương trình server của bạn, nó cần hai mẩu thông tin: địa chỉ IP và số port. Lời gọi `bind()` cho phép bạn làm đúng chuyện đó.

Đầu tiên, bạn gọi `getaddrinfo()` để nạp một `struct sockaddr` với thông tin địa chỉ đích và port. Rồi bạn gọi `socket()` để có một socket descriptor, rồi bạn truyền socket và địa chỉ vào `bind()`, và địa chỉ IP cùng port được gắn vào socket một cách thần kỳ (dùng phép thuật thật sự)!

Nếu bạn không biết địa chỉ IP của mình, hoặc bạn biết mình chỉ có một địa chỉ IP trên máy, hoặc bạn không quan tâm địa chỉ IP nào của máy được dùng, bạn có thể chỉ cần truyền cờ `AI_PASSIVE` vào tham số `hints` của `getaddrinfo()`. Cái này làm gì? Nó điền phần địa chỉ IP của `struct sockaddr` bằng một giá trị đặc biệt báo cho `bind()` biết rằng nó nên tự động điền địa chỉ IP của host này.

Cái gì cái gì? Giá trị đặc biệt nào được nạp vào địa chỉ IP của `struct sockaddr` để làm nó tự động điền địa chỉ bằng host hiện tại? Tôi sẽ nói cho bạn biết, nhưng nhớ là chuyện này chỉ khi bạn đang điền `struct sockaddr` bằng tay; nếu không, dùng kết quả từ `getaddrinfo()`, như trên. Ở IPv4, trường `sin_addr.s_addr` của struct `sockaddr_in` được gán thành `INADDR_ANY`. Ở IPv6, trường `sin6_addr` của struct `sockaddr_in6` được gán từ biến toàn cục `in6addr_any`. Hoặc, nếu bạn đang khai báo một `struct in6_addr` mới, bạn có thể khởi tạo nó bằng `IN6ADDR_ANY_INIT`.

Cuối cùng, tham số `addrlen` nên được gán bằng `sizeof my_addr`.

Return Value

Trả về không nếu thành công, hoặc `-1` nếu lỗi (và `errno` sẽ được gán phù hợp).

Example

```
// modern way of doing things with getaddrinfo()

struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, "3490", &hints, &res);

// make a socket:
// (you should actually walk the "res" linked list and error-check!)

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// bind it to the port we passed in to getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);
```

```
// example of packing a struct by hand, IPv4

struct sockaddr_in myaddr;
int s;

myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(3490);

// you can specify an IP address:
inet_pton(AF_INET, "63.161.169.137", &(myaddr.sin_addr));

// or you can let it automatically select one:
myaddr.sin_addr.s_addr = INADDR_ANY;

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&myaddr, sizeof myaddr);
```

See Also

`getaddrinfo()`, `socket()`, `struct sockaddr_in`, `struct in_addr`

9.3 `connect()`

Kết nối một socket tới server

Synopsis

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *serv_addr,
            socklen_t addrlen);
```

Description

Khi đã dựng được một socket descriptor bằng lời gọi `socket()`, bạn có thể `connect()` socket đó tới một server remote bằng system call tên gọi rất đúng bản chất là `connect()`. Tất cả những gì bạn cần làm là truyền cho nó socket descriptor và địa chỉ của server bạn muốn làm quen. (À, và độ dài của địa chỉ, thứ thường được truyền cho các hàm kiểu này.)

Thông thường thông tin này đi kèm như kết quả của lời gọi `getaddrinfo()`, nhưng bạn có thể tự điền `struct sockaddr` của mình nếu muốn.

Nếu bạn chưa gọi `bind()` trên socket descriptor, nó sẽ tự động được bind vào địa chỉ IP của bạn và một port local ngẫu nhiên. Chuyện này thường ổn với bạn nếu bạn không phải server, vì bạn không thực sự quan tâm port local của mình là gì; bạn chỉ quan tâm port remote là gì để có thể đặt nó vào tham số `serv_addr`. Bạn *có thể* gọi `bind()` nếu bạn thực sự muốn socket client của mình nằm trên một địa chỉ IP và port cụ thể, nhưng chuyện này khá hiếm.

Khi socket đã `connect()`, bạn tự do `send()` và `recv()` dữ liệu trên nó tùy ý.

Ghi chú đặc biệt: nếu bạn `connect()` một socket UDP `SOCK_DGRAM` tới một host remote, bạn có thể dùng `send()` và `recv()` cũng như `sendto()` và `recvfrom()`. Nếu bạn muốn.

Return Value

Trả về không nếu thành công, hoặc `-1` nếu lỗi (và `errno` sẽ được gán phù hợp).

Example

```
// connect to www.example.com port 80 (http)

struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;

// we could put "80" instead on "http" on the next line:
getaddrinfo("www.example.com", "http", &hints, &res);
```

```
// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// connect it to the address and port we passed in to getaddrinfo():

connect(sockfd, res->ai_addr, res->ai_addrlen);
```

See Also

`socket()`, `bind()`

9.4 `close()`

Đóng một socket descriptor

Synopsis

```
#include <unistd.h>

int close(int s);
```

Description

Sau khi bạn đã dùng xong socket cho bất kỳ âm mưu điên rồ nào bạn đã bày ra và bạn không muốn `send()` hay `recv()` hay, nói thẳng, làm *bất cứ gì khác* với socket này, bạn có thể `close()` nó, và nó sẽ được giải phóng, không bao giờ dùng lại nữa.

Đầu bên kia có thể biết chuyện này xảy ra bằng một trong hai cách. Một: nếu đầu bên kia gọi `recv()`, nó sẽ trả về `0`. Hai: nếu đầu bên kia gọi `send()`, nó sẽ nhận signal `SIGPIPE` và send() sẽ trả về `-1` và `errno` sẽ được gán thành `EPIPE`.

Người dùng Windows: hàm bạn cần dùng tên là `closesocket()`, không phải `close()`. Nếu bạn thử dùng `close()` trên socket descriptor, có thể Windows sẽ nổi giận... Và bạn sẽ không thích nó khi nó nổi giận đâu.

Return Value

Trả về không nếu thành công, hoặc `-1` nếu lỗi (và `errno` sẽ được gán phù hợp).

Example

```
s = socket(PF_INET, SOCK_DGRAM, 0);
.
.
.
// a whole lotta stuff...*BRRRONNNN!*
.
.
.
```

```
close(s);  // not much to it, really.
```

See Also

`socket()`, `shutdown()`

9.5 `getaddrinfo()`, `freeaddrinfo()`, `gai_strerror()`

Lấy thông tin về một tên host và/hoặc service, rồi nạp một `struct sockaddr` với kết quả.

Synopsis

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *nodename, const char *servname,
                const struct addrinfo *hints,
                struct addrinfo **res);

void freeaddrinfo(struct addrinfo *ai);

const char *gai_strerror(int ecode);

struct addrinfo {
  int     ai_flags;          // AI_PASSIVE, AI_CANONNAME, ...
  int     ai_family;         // AF_xxx
  int     ai_socktype;       // SOCK_xxx
  int     ai_protocol;       // 0 (auto) or IPPROTO_TCP, IPPROTO_UDP

  socklen_t  ai_addrlen;     // length of ai_addr
  char   *ai_canonname;      // canonical name for nodename
  struct sockaddr  *ai_addr; // binary address
  struct addrinfo  *ai_next; // next structure in linked list
};
```

Description

`getaddrinfo()` là một hàm xuất sắc sẽ trả về thông tin về một tên host cụ thể (như địa chỉ IP của nó) và nạp một `struct sockaddr` cho bạn, lo hết các chi tiết linh kỉnh (như IPv4 hay IPv6). Nó thay thế các hàm cũ `gethostbyname()` và `getservbyname()`. Mô tả ở dưới có một đống thông tin có thể hơi ngợp, nhưng cách dùng thực tế khá đơn giản. Có thể đáng xem ví dụ trước.

Tên host mà bạn quan tâm đặt vào tham số `nodename`. Địa chỉ có thể là một tên host, như "www.example.com", hoặc một địa chỉ IPv4 hay IPv6 (truyền vào dạng chuỗi). Tham số này cũng có thể là `NULL` nếu bạn đang dùng cờ `AI_PASSIVE` (xem bên dưới).

Tham số `servname` về cơ bản là số port. Nó có thể là một số port (truyền vào dạng chuỗi, như "80"), hoặc nó có thể là tên service, như "http", "tftp", "smtp", "pop", vân vân. Tên service nổi tiếng có thể tìm thấy

trong IANA Port List[1] hoặc trong file `/etc/services` của bạn.

Cuối cùng, cho các tham số đầu vào, chúng ta có `hints`. Đây thật sự là nơi bạn định nghĩa những gì hàm `getaddrinfo()` sẽ làm. Xóa toàn bộ struct về không trước khi dùng bằng `memset()`. Hãy xem qua các trường bạn cần cấu hình trước khi dùng.

`ai_flags` có thể được gán thành nhiều thứ, nhưng đây là vài cái quan trọng. (Có thể chỉ định nhiều cờ bằng cách OR bitwise chúng lại với toán tử `|`.) Kiểm tra man page của bạn để có danh sách cờ đầy đủ.

`AI_CANONNAME` làm cho `ai_canonname` của kết quả được điền bằng tên canonical (thật) của host. `AI_PASSIVE` làm cho địa chỉ IP của kết quả được điền bằng `INADDR_ANY` (IPv4) hoặc `in6addr_any` (IPv6); điều này khiến lời gọi `bind()` tiếp theo tự động điền địa chỉ IP của `struct sockaddr` bằng địa chỉ của host hiện tại. Tuyệt vời cho việc dựng server khi bạn không muốn hardcode địa chỉ.

Nếu bạn có dùng cờ `AI_PASSIVE`, thì bạn có thể truyền `NULL` vào `nodename` (vì sau đó `bind()` sẽ điền nó cho bạn).

Tiếp tục với các tham số đầu vào, có lẽ bạn sẽ muốn gán `ai_family` thành `AF_UNSPEC`, báo cho `getaddrinfo()` tìm cả địa chỉ IPv4 lẫn IPv6. Bạn cũng có thể tự giới hạn mình ở một trong hai bằng `AF_INET` hoặc `AF_INET6`.

Kế tiếp, trường `socktype` nên được gán thành `SOCK_STREAM` hoặc `SOCK_DGRAM`, tùy vào loại socket bạn muốn.

Cuối cùng, cứ để `ai_protocol` ở `0` để tự động chọn kiểu protocol của bạn.

Giờ, sau khi bạn đã có tất cả thứ đó, bạn có thể *cuối cùng* gọi `getaddrinfo()`!

Dĩ nhiên, đây là nơi vui bắt đầu. `res` giờ sẽ trỏ tới một linked list của các `struct addrinfo`, và bạn có thể đi qua danh sách này để lấy tất cả địa chỉ khớp với những gì bạn đã truyền vào qua hints.

Giờ, có khả năng bạn sẽ có một vài địa chỉ không chạy được vì lý do này hay lý do khác, nên cái man page Linux làm là lặp qua danh sách gọi `socket()` và `connect()` (hoặc `bind()` nếu bạn đang dựng server với cờ `AI_PASSIVE`) cho đến khi thành công.

Cuối cùng, khi bạn đã dùng xong linked list, bạn cần gọi `freeaddrinfo()` để giải phóng bộ nhớ (nếu không nó sẽ bị rò rỉ, và Một Số Người sẽ nổi giận).

Return Value

Trả về không nếu thành công, hoặc khác không nếu lỗi. Nếu trả về khác không, bạn có thể dùng hàm `gai_strerror()` để có phiên bản in được của mã lỗi trong giá trị trả về.

Example

```
// code for a client connecting to a server
// namely a stream socket to www.example.com on port 80 (http)
// either IPv4 or IPv6

int sockfd;
struct addrinfo hints, *servinfo, *p;
int rv;

memset(&hints, 0, sizeof hints);
```

[1] https://www.iana.org/assignments/port-numbers

```
hints.ai_family = AF_UNSPEC; // use AF_INET6 to force IPv6
hints.ai_socktype = SOCK_STREAM;

rv = getaddrinfo("www.example.com", "http", &hints, &servinfo);
if (rv != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    exit(1);
}

// loop through all the results and connect to the first we can
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
        perror("socket");
        continue;
    }

    if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
        perror("connect");
        close(sockfd);
        continue;
    }

    break; // if we get here, we must have connected successfully
}

if (p == NULL) {
    // looped off the end of the list with no connection
    fprintf(stderr, "failed to connect\n");
    exit(2);
}

freeaddrinfo(servinfo); // all done with this structure
```

```
// code for a server waiting for connections
// namely a stream socket on port 3490, on this host's IP
// either IPv4 or IPv6.

int sockfd;
struct addrinfo hints, *servinfo, *p;
int rv;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // use AF_INET6 to force IPv6
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE; // use my IP address

if ((rv = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    exit(1);
}
```

```
// loop through all the results and bind to the first we can
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
        perror("socket");
        continue;
    }

    if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
        close(sockfd);
        perror("bind");
        continue;
    }

    break; // if we get here, we must have connected successfully
}

if (p == NULL) {
    // looped off the end of the list with no successful bind
    fprintf(stderr, "failed to bind socket\n");
    exit(2);
}

freeaddrinfo(servinfo); // all done with this structure
```

See Also

`gethostbyname()`, `getnameinfo()`

9.6 `gethostname()`

Trả về tên của hệ thống

Synopsis

```
#include <sys/unistd.h>

int gethostname(char *name, size_t len);
```

Description

Hệ thống của bạn có tên. Tất cả đều có. Cái này Unix hơn một chút so với phần mạng chúng ta đã nói, nhưng nó vẫn có chỗ dùng.

Ví dụ, bạn có thể lấy tên host của mình, rồi gọi `gethostbyname()` để tìm ra địa chỉ IP của mình.

Tham số `name` nên trỏ tới một buffer sẽ chứa tên host, và `len` là kích thước của buffer đó tính theo byte. `gethostname()` sẽ không ghi đè quá cuối buffer (nó có thể trả về lỗi, hoặc có thể chỉ ngưng ghi), và nó sẽ thêm `NUL` kết thúc chuỗi nếu có chỗ trong buffer.

Return Value

Trả về không nếu thành công, hoặc `-1` nếu lỗi (và `errno` sẽ được gán phù hợp).

Example

```
char hostname[128];

gethostname(hostname, sizeof hostname);
printf("My hostname: %s\n", hostname);
```

See Also

`gethostbyname()`

9.7 `gethostbyname()`, `gethostbyaddr()`

Lấy địa chỉ IP cho một hostname, hoặc ngược lại

Synopsis

```
#include <sys/socket.h>
#include <netdb.h>

struct hostent *gethostbyname(const char *name); // DEPRECATED!
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

Description

XIN LƯU Ý: hai hàm này đã được thay thế bởi `getaddrinfo()` và `getnameinfo()`! Đặc biệt, `gethostbyname()` không chạy tốt với IPv6.

Các hàm này ánh xạ qua lại giữa tên host và địa chỉ IP. Ví dụ, nếu bạn có "www.example.com", bạn có thể dùng `gethostbyname()` để lấy địa chỉ IP của nó và lưu vào một `struct in_addr`.

Ngược lại, nếu bạn có một `struct in_addr` hoặc một `struct in6_addr`, bạn có thể dùng `gethostbyaddr()` để lấy hostname. Hàm `gethostbyaddr()` *có* tương thích IPv6, nhưng bạn nên dùng hàm mới sáng bóng hơn là `getnameinfo()` thay thế.

(Nếu bạn có một chuỗi chứa địa chỉ IP ở dạng chấm-và-số mà bạn muốn tra hostname, bạn sẽ dùng `getaddrinfo()` với cờ `AI_CANONNAME` sẽ tốt hơn.)

`gethostbyname()` nhận một chuỗi như "www.yahoo.com", và trả về một `struct hostent` chứa hàng đống thông tin, bao gồm địa chỉ IP. (Thông tin khác là tên host chính thức, danh sách alias, kiểu địa chỉ, độ dài của các địa chỉ, và danh sách địa chỉ, đó là struct đa mục đích khá dễ dùng cho mục đích cụ thể của chúng ta một khi bạn hiểu cách.)

`gethostbyaddr()` nhận một `struct in_addr` hoặc `struct in6_addr` và đưa về cho bạn một tên host tương ứng (nếu có một), nên nó hơi kiểu ngược lại của `gethostbyname()`. Về tham số, dù `addr` là `char*`, thực chất bạn muốn truyền vào một con trỏ tới `struct in_addr`. `len` nên là `sizeof(struct in_addr)`, và `type` nên là `AF_INET`.

Vậy cái `struct hostent` được trả về này là gì? Nó có một số trường chứa thông tin về host đang nói.

Trường	Mô tả
`char *h_name`	Tên host canonical thật.
`char **h_aliases`	Danh sách alias có thể truy cập bằng mảng, phần tử cuối là `NULL`
`int h_addrtype`	Kiểu địa chỉ của kết quả, thật ra nên là `AF_INET` cho mục đích của chúng ta.
`int length`	Độ dài của địa chỉ tính theo byte, là 4 cho địa chỉ IP (phiên bản 4).
`char **h_addr_list`	Danh sách địa chỉ IP cho host này. Mặc dù đây là `char**`, thật ra nó là mảng ngụy trang của các `struct in_addr*`. Phần tử cuối của mảng là `NULL`.
`h_addr`	Một alias hay được định nghĩa cho `h_addr_list[0]`. Nếu bạn chỉ cần địa chỉ IP nào cũng được cho host này (đúng, host có thể có nhiều hơn một) chỉ cần dùng trường này.

Return Value

Trả về một con trỏ tới `struct hostent` kết quả nếu thành công, hoặc `NULL` nếu lỗi.

Thay vì `perror()` thông thường và mấy thứ bạn thường dùng để báo lỗi, các hàm này có kết quả song song trong biến `h_errno`, có thể in bằng các hàm `herror()` hoặc `hstrerror()`. Chúng hoạt động giống các hàm `errno`, `perror()`, và `strerror()` cổ điển mà bạn đã quen.

Example

```
// THIS IS A DEPRECATED METHOD OF GETTING HOST NAMES
// use getaddrinfo() instead!

#include <stdio.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
    int i;
    struct hostent *he;
    struct in_addr **addr_list;

    if (argc != 2) {
        fprintf(stderr,"usage: ghbn hostname\n");
        return 1;
    }

    if ((he = gethostbyname(argv[1])) == NULL) {  // get host info
        herror("gethostbyname");
        return 2;
    }
```

```
    // print information about this host:
    printf("Official name is: %s\n", he->h_name);
    printf("    IP addresses: ");
    addr_list = (struct in_addr **)he->h_addr_list;
    for(i = 0; addr_list[i] != NULL; i++) {
        printf("%s ", inet_ntoa(*addr_list[i]));
    }
    printf("\n");

    return 0;
}
```

```
// THIS HAS BEEN SUPERSEDED
// use getnameinfo() instead!

struct hostent *he;
struct in_addr ipv4addr;
struct in6_addr ipv6addr;

inet_pton(AF_INET, "192.0.2.34", &ipv4addr);
he = gethostbyaddr(&ipv4addr, sizeof ipv4addr, AF_INET);
printf("Host name: %s\n", he->h_name);

inet_pton(AF_INET6, "2001:db8:63b3:1::beef", &ipv6addr);
he = gethostbyaddr(&ipv6addr, sizeof ipv6addr, AF_INET6);
printf("Host name: %s\n", he->h_name);
```

See Also

`getaddrinfo()`, `getnameinfo()`, `gethostname()`, `errno`, `perror()`, `strerror()`, `struct in_addr`

9.8 `getnameinfo()`

Tra thông tin tên host và tên service cho một `struct sockaddr` đã cho.

Synopsis

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                char *host, size_t hostlen,
                char *serv, size_t servlen, int flags);
```

Description

Hàm này là ngược lại của `getaddrinfo()`, nghĩa là, hàm này nhận một `struct sockaddr` đã được nạp và tra tên cùng tên service trên đó. Nó thay thế các hàm cũ `gethostbyaddr()` và `getservbyport()`.

Bạn phải truyền vào một con trỏ tới `struct sockaddr` (thực chất có thể là `struct sockaddr_in` hoặc `struct sockaddr_in6` đã được cast) trong tham số `sa`, và độ dài của struct đó trong `salen`.

Tên host và tên service kết quả sẽ được ghi vào vùng được trỏ tới bởi các tham số `host` và `serv`. Dĩ nhiên, bạn phải chỉ định độ dài tối đa của các buffer này trong `hostlen` và `servlen`.

Cuối cùng, có vài cờ bạn có thể truyền, nhưng đây là vài cái hay. `NI_NOFQDN` sẽ làm cho `host` chỉ chứa tên host, không phải tên domain đầy đủ. `NI_NAMEREQD` sẽ làm hàm thất bại nếu không tìm được tên qua DNS lookup (nếu bạn không chỉ định cờ này và không tìm được tên, `getnameinfo()` sẽ đặt phiên bản chuỗi của địa chỉ IP vào `host` thay thế).

Như mọi khi, kiểm tra man page cục bộ của bạn để có thông tin đầy đủ.

Return Value

Trả về không nếu thành công, hoặc khác không nếu lỗi. Nếu giá trị trả về khác không, nó có thể được truyền cho `gai_strerror()` để có chuỗi dễ đọc. Xem `getaddrinfo` để biết thêm.

Example

```
struct sockaddr_in6 sa; // could be IPv4 if you want
char host[1024];
char service[20];

// pretend sa is full of good information about the host and port...

getnameinfo(&sa, sizeof sa, host, sizeof host, service,
            sizeof service, 0);

printf("   host: %s\n", host);    // e.g. "www.example.com"
printf("service: %s\n", service); // e.g. "http"
```

See Also

`getaddrinfo()`, `gethostbyaddr()`

9.9 `getpeername()`

Trả về thông tin địa chỉ về đầu remote của kết nối

Synopsis

```
#include <sys/socket.h>

int getpeername(int s, struct sockaddr *addr, socklen_t *len);
```

Description

Khi bạn đã `accept()` một kết nối remote, hoặc `connect()` tới một server, bạn giờ có cái gọi là *peer*. Peer của bạn đơn giản là máy tính bạn đang kết nối tới, được nhận diện bằng một địa chỉ IP và một port. Vậy...

`getpeername()` đơn giản trả về một `struct sockaddr_in` được điền thông tin về máy bạn đang kết nối tới.

Tại sao nó được gọi là "name"? Có nhiều loại socket khác nhau, không chỉ Internet Socket như chúng ta đang dùng trong hướng dẫn này, nên "name" là thuật ngữ tổng quát hay bao phủ mọi trường hợp. Trong trường hợp của chúng ta, "name" của peer là địa chỉ IP và port của nó.

Mặc dù hàm trả về kích thước của địa chỉ kết quả trong `len`, bạn phải nạp sẵn `len` bằng kích thước của `addr`.

Return Value

Trả về không nếu thành công, hoặc `-1` nếu lỗi (và `errno` sẽ được gán phù hợp).

Example

```
// assume s is a connected socket

socklen_t len;
struct sockaddr_storage addr;
char ipstr[INET6_ADDRSTRLEN];
int port;

len = sizeof addr;
getpeername(s, (struct sockaddr*)&addr, &len);

// deal with both IPv4 and IPv6:
if (addr.ss_family == AF_INET) {
    struct sockaddr_in *s = (struct sockaddr_in *)&addr;
    port = ntohs(s->sin_port);
    inet_ntop(AF_INET, &s->sin_addr, ipstr, sizeof ipstr);
} else { // AF_INET6
    struct sockaddr_in6 *s = (struct sockaddr_in6 *)&addr;
    port = ntohs(s->sin6_port);
    inet_ntop(AF_INET6, &s->sin6_addr, ipstr, sizeof ipstr);
}

printf("Peer IP address: %s\n", ipstr);
printf("Peer port      : %d\n", port);
```

See Also

`gethostname()`, `gethostbyname()`, `gethostbyaddr()`

9.10 `errno`

Giữ mã lỗi cho system call vừa gọi

Synopsis

```
#include <errno.h>

int errno;
```

Description

Đây là biến giữ thông tin lỗi cho nhiều system call. Nếu bạn còn nhớ, những thứ như `socket()` và `listen()` trả về `-1` khi lỗi, và chúng đặt giá trị cụ thể của `errno` để cho bạn biết lỗi nào đã xảy ra.

File header `errno.h` liệt kê một đống tên ký hiệu hằng cho các lỗi, như `EADDRINUSE`, `EPIPE`, `ECONNREFUSED`, vân vân. Man page cục bộ của bạn sẽ cho bạn biết mã nào có thể được trả về như là lỗi, và bạn có thể dùng chúng ở runtime để xử lý các lỗi khác nhau theo cách khác nhau.

Hoặc, thường gặp hơn, bạn có thể gọi `perror()` hoặc `strerror()` để có phiên bản dễ đọc của lỗi.

Một điều cần lưu ý, cho các fan đa luồng, là trên hầu hết hệ thống `errno` được định nghĩa theo cách thread-safe. (Nghĩa là, nó không thật sự là biến toàn cục, nhưng hành xử y như một biến toàn cục trong môi trường đơn luồng.)

Return Value

Giá trị của biến là lỗi mới nhất đã xảy ra, có thể là mã cho "thành công" nếu hành động vừa rồi thành công.

Example

```
s = socket(PF_INET, SOCK_STREAM, 0);
if (s == -1) {
    perror("socket"); // or use strerror()
}

tryagain:
if (select(n, &readfds, NULL, NULL) == -1) {
    // an error has occurred!!

    // if we were only interrupted, just restart the select() call:
    if (errno == EINTR) goto tryagain;  // AAAA! goto!!!

    // otherwise it's a more serious error:
    perror("select");
    exit(1);
}
```

See Also

`perror()`, `strerror()`

9.11 `fcntl()`

Điều khiển các socket descriptor

Synopsis

```
#include <sys/unistd.h>
#include <sys/fcntl.h>

int fcntl(int s, int cmd, long arg);
```

Description

Hàm này thường được dùng để làm file locking và các chuyện liên quan đến file, nhưng nó cũng có vài chức năng liên quan đến socket mà bạn có thể thấy hoặc dùng thỉnh thoảng.

Tham số `s` là socket descriptor bạn muốn thao tác, `cmd` nên được gán thành `F_SETFL`, và `arg` có thể là một trong các lệnh sau. (Như tôi đã nói, `fcntl()` còn nhiều hơn những gì tôi đang tiết lộ ở đây, nhưng tôi đang cố giữ tập trung vào socket.)

`cmd`	Mô tả
`O_NONBLOCK`	Đặt socket thành non-blocking. Xem phần về blocking để biết chi tiết.
`O_ASYNC`	Đặt socket làm I/O bất đồng bộ. Khi có dữ liệu sẵn sàng để `recv()` trên socket, signal `SIGIO` sẽ được raise. Ít khi thấy, và vượt ra ngoài phạm vi hướng dẫn. Và tôi nghĩ nó chỉ có trên một số hệ thống.

Return Value

Trả về không nếu thành công, hoặc `-1` nếu lỗi (và `errno` sẽ được gán phù hợp).

Các cách dùng khác nhau của system call `fcntl()` thật ra có giá trị trả về khác nhau, nhưng tôi không bao phủ chúng ở đây vì chúng không liên quan đến socket. Xem man page `fcntl()` cục bộ của bạn để biết thêm.

Example

```
int s = socket(PF_INET, SOCK_STREAM, 0);

fcntl(s, F_SETFL, O_NONBLOCK);  // set to non-blocking
fcntl(s, F_SETFL, O_ASYNC);     // set to asynchronous I/O
```

See Also

Blocking, `send()`

9.12 `htons()`, `htonl()`, `ntohs()`, `ntohl()`

Chuyển các kiểu số nguyên nhiều byte từ host byte order sang network byte order

Synopsis

```
#include <netinet/in.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Description

Chỉ để làm bạn thật sự không vui, các máy tính khác nhau dùng thứ tự byte khác nhau nội bộ cho các số nguyên nhiều byte (tức là mọi số nguyên lớn hơn một `char`). Hệ quả là nếu bạn `send()` một `short int` hai byte từ máy Intel sang máy Mac (trước khi chúng cũng biến thành Intel luôn), cái một máy tính nghĩ là số `1`, máy kia sẽ nghĩ là số `256`, và ngược lại.

Cách vượt qua vấn đề này là tất cả mọi người gạt bỏ khác biệt và đồng ý rằng Motorola và IBM đúng, còn Intel làm cách kỳ cục, và vì vậy tất cả chúng ta chuyển thứ tự byte của mình thành "big-endian" trước khi gửi ra. Vì Intel là máy "little-endian", đúng chính trị hơn là gọi thứ tự byte ưu tiên của chúng ta là "Network Byte Order". Vậy các hàm này chuyển từ thứ tự byte gốc sang network byte order và ngược lại.

(Chuyện này nghĩa là trên Intel các hàm này đảo tất cả byte, còn trên PowerPC chúng không làm gì vì các byte đã ở Network Byte Order rồi. Nhưng bạn vẫn luôn nên dùng chúng trong code, vì có ai đó có thể muốn build nó trên máy Intel và vẫn muốn mọi thứ chạy đúng.)

Lưu ý rằng các kiểu liên quan là số 32-bit (4 byte, có lẽ `int`) và 16-bit (2 byte, rất có thể `short`).

Có các biến thể 64-bit trên nhiều hệ thống. Xem hàm `htobe64()` [2] và họ hàng trong `<endian.h>` nếu bạn có (có vẻ MacOS thì không có). Và GCC có byte swapping built-ins[3] thậm chí lên tới 128 bit. Hoặc bạn có thể tự cuộn tay[4], nhưng chỉ thực sự làm swap nếu bạn đang ở trên máy little-endian!

Dù sao, cách các hàm này hoạt động là trước tiên bạn quyết định mình đang chuyển *từ* host (byte order của máy bạn) hay từ network byte order. Nếu "host", thì chữ đầu của hàm bạn sắp gọi là "h". Nếu không thì là "n" cho "network". Phần giữa tên hàm luôn là "to" vì bạn đang chuyển từ cái này "to" cái khác, và chữ áp chót cho biết bạn đang chuyển *sang* cái gì. Chữ cuối là kích thước dữ liệu, "s" cho short, hoặc "l" cho long. Vậy:

Hàm	Mô tả
`htons()`	`h` ost `to` `n` etwork `s` hort
`htonl()`	`h` ost `to` `n` etwork `l` ong
`ntohs()`	`n` etwork `to` `h` ost `s` hort
`ntohl()`	`n` etwork `to` `h` ost `l` ong

Return Value

Mỗi hàm trả về giá trị đã được chuyển.

[2] https://man.archlinux.org/man/htobe64
[3] https://gcc.gnu.org/onlinedocs/gcc/Byte-Swapping-Builtins.html
[4] https://beej.us/guide/bgnet/source/examples/htonll.c

Example

```
uint32_t some_long = 10;
uint16_t some_short = 20;

uint32_t network_byte_order;

// convert and send
network_byte_order = htonl(some_long);
send(s, &network_byte_order, sizeof(uint32_t), 0);

some_short == ntohs(htons(some_short)); // this expression is true
```

9.13 `inet_ntoa()`, `inet_aton()`, `inet_addr`

Chuyển địa chỉ IP từ chuỗi chấm-và-số sang `struct in_addr` và ngược lại

Synopsis

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

// ALL THESE ARE DEPRECATED!
// Use inet_pton() or inet_ntop() instead!

char *inet_ntoa(struct in_addr in);
int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
```

Description

Các hàm này bị deprecated vì chúng không xử lý IPv6! Dùng `inet_ntop()` hoặc `inet_pton()` thay thế! Chúng được đưa vào đây vì bạn vẫn có thể gặp chúng ngoài đời.

Tất cả các hàm này chuyển từ `struct in_addr` (một phần của `struct sockaddr_in` của bạn, có khả năng cao) sang một chuỗi ở định dạng chấm-và-số (ví dụ "192.168.5.10") và ngược lại. Nếu bạn có một địa chỉ IP được truyền qua command line hay gì đó, đây là cách dễ nhất để có `struct in_addr` để `connect()` tới, hoặc bất cứ gì. Nếu bạn cần quyền năng hơn, thử vài hàm DNS như `gethostbyname()` hoặc cố đảo chính *coup d'État* ở nước bản địa của bạn.

Hàm `inet_ntoa()` chuyển một địa chỉ mạng trong `struct in_addr` sang chuỗi định dạng chấm-và-số. Chữ "n" trong "ntoa" là "network", và "a" là "ASCII" vì lý do lịch sử (nên đó là "Network To ASCII", hậu tố "toa" có một người bạn tương tự trong thư viện C gọi là `atoi()` chuyển chuỗi ASCII sang số nguyên).

Hàm `inet_aton()` là ngược lại, chuyển từ chuỗi chấm-và-số sang một `in_addr_t` (là kiểu của trường `s_addr` trong `struct in_addr` của bạn).

Cuối cùng, hàm `inet_addr()` là hàm cũ hơn làm cơ bản cùng chuyện với `inet_aton()`. Về mặt lý thuyết nó bị deprecated, nhưng bạn sẽ thấy nó nhiều và cảnh sát sẽ không đến bắt bạn nếu bạn dùng nó.

Return Value

`inet_aton()` trả về khác không nếu địa chỉ hợp lệ, và trả về không nếu địa chỉ không hợp lệ.

`inet_ntoa()` trả về chuỗi chấm-và-số trong một buffer tĩnh bị ghi đè mỗi lần gọi hàm.

`inet_addr()` trả về địa chỉ dưới dạng `in_addr_t`, hoặc `-1` nếu có lỗi. (Đây là cùng kết quả nếu bạn thử chuyển chuỗi "`255.255.255.255`", là một địa chỉ IP hợp lệ. Đây là lý do `inet_aton()` tốt hơn.)

Example

```
struct sockaddr_in antelope;
char *some_addr;

inet_aton("10.0.0.1", &antelope.sin_addr); // store IP in antelope

some_addr = inet_ntoa(antelope.sin_addr); // return the IP
printf("%s\n", some_addr); // prints "10.0.0.1"

// and this call is the same as the inet_aton() call, above:
antelope.sin_addr.s_addr = inet_addr("10.0.0.1");
```

See Also

`inet_ntop()`, `inet_pton()`, `gethostbyname()`, `gethostbyaddr()`

9.14 `inet_ntop()`, `inet_pton()`

Chuyển địa chỉ IP sang dạng người đọc được và ngược lại.

Synopsis

```
#include <arpa/inet.h>

const char *inet_ntop(int af, const void *src,
                      char *dst, socklen_t size);

int inet_pton(int af, const char *src, void *dst);
```

Description

Các hàm này để xử lý địa chỉ IP dạng người đọc được và chuyển chúng sang biểu diễn nhị phân để dùng với nhiều hàm và system call. Chữ "n" là "network", và "p" là "presentation". Hoặc "text presentation". Nhưng bạn có thể nghĩ nó là "printable". "ntop" là "network to printable". Thấy chưa?

Đôi khi bạn không muốn nhìn vào một đống số nhị phân khi xem một địa chỉ IP. Bạn muốn nó ở dạng in đẹp đẽ, như `192.0.2.180`, hay `2001:db8:8714:3a90::12`. Trong trường hợp đó, `inet_ntop()` là dành cho bạn.

`inet_ntop()` nhận họ địa chỉ trong tham số `af` (hoặc `AF_INET` hoặc `AF_INET6`). Tham số `src` nên là con trỏ tới một `struct in_addr` hoặc `struct in6_addr` chứa địa chỉ bạn muốn chuyển thành chuỗi.

Cuối cùng `dst` và `size` là con trỏ tới chuỗi đích và độ dài tối đa của chuỗi đó.

Độ dài tối đa của chuỗi `dst` nên là bao nhiêu? Độ dài tối đa cho địa chỉ IPv4 và IPv6 là bao nhiêu? Rất may có vài macro giúp bạn. Các độ dài tối đa là: `INET_ADDRSTRLEN` và `INET6_ADDRSTRLEN`.

Lúc khác, bạn có thể có một chuỗi chứa địa chỉ IP ở dạng đọc được, và bạn muốn pack nó vào một `struct sockaddr_in` hoặc một `struct sockaddr_in6`. Trong trường hợp đó, hàm ngược lại `inet_pton()` là cái bạn cần.

`inet_pton()` cũng nhận họ địa chỉ (hoặc `AF_INET` hoặc `AF_INET6`) trong tham số `af`. Tham số `src` là con trỏ tới một chuỗi chứa địa chỉ IP ở dạng in được. Cuối cùng tham số `dst` trỏ tới nơi kết quả nên được lưu, có thể là `struct in_addr` hoặc `struct in6_addr`.

Các hàm này không làm DNS lookup, bạn sẽ cần `getaddrinfo()` cho cái đó.

Return Value

`inet_ntop()` trả về tham số `dst` nếu thành công, hoặc `NULL` nếu thất bại (và `errno` được gán).

`inet_pton()` trả về `1` nếu thành công. Nó trả về `-1` nếu có lỗi (`errno` được gán), hoặc `0` nếu đầu vào không phải địa chỉ IP hợp lệ.

Example

```
// IPv4 demo of inet_ntop() and inet_pton()

struct sockaddr_in sa;
char str[INET_ADDRSTRLEN];

// store this IP address in sa:
inet_pton(AF_INET, "192.0.2.33", &(sa.sin_addr));

// now get it back and print it
inet_ntop(AF_INET, &(sa.sin_addr), str, INET_ADDRSTRLEN);

printf("%s\n", str); // prints "192.0.2.33"
```

```
// IPv6 demo of inet_ntop() and inet_pton()
// (basically the same except with a bunch of 6s thrown around)

struct sockaddr_in6 sa;
char str[INET6_ADDRSTRLEN];

// store this IP address in sa:
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &(sa.sin6_addr));

// now get it back and print it
inet_ntop(AF_INET6, &(sa.sin6_addr), str, INET6_ADDRSTRLEN);

printf("%s\n", str); // prints "2001:db8:8714:3a90::12"
```

```
// Helper function you can use:

//Convert a struct sockaddr address to a string, IPv4 and IPv6:

char *get_ip_str(const struct sockaddr *sa, char *s, size_t maxlen)
{
    switch(sa->sa_family) {
        case AF_INET:
            inet_ntop(AF_INET,
                    &(((struct sockaddr_in *)sa)->sin_addr), s,
                    maxlen);
            break;

        case AF_INET6:
            inet_ntop(AF_INET6,
                    &(((struct sockaddr_in6 *)sa)->sin6_addr), s,
                    maxlen);
            break;

        default:
            strncpy(s, "Unknown AF", maxlen);
            return NULL;
    }

    return s;
}
```

See Also

`getaddrinfo()`

9.15 `listen()`

Báo một socket lắng nghe kết nối đi tới

Synopsis

```
#include <sys/socket.h>

int listen(int s, int backlog);
```

Description

Bạn có thể cầm socket descriptor của mình (tạo bằng system call `socket()`) và bảo nó lắng nghe kết nối đi tới. Đây là điều phân biệt server với client đấy các bạn.

Tham số `backlog` có thể nghĩa vài thứ khác nhau tùy hệ thống bạn đang dùng, nhưng nói chung nó là có thể có bao nhiêu kết nối đang chờ trước khi kernel bắt đầu từ chối các kết nối mới. Khi các kết nối mới đến, bạn nên nhanh chóng `accept()` chúng để backlog không đầy. Thử gán 10 hoặc gì đó, và nếu client của bạn bắt đầu bị "Connection refused" dưới tải nặng, tăng lên.

Trước khi gọi `listen()`, server của bạn nên gọi `bind()` để gắn mình vào một số port cụ thể. Số port đó (trên địa chỉ IP của server) sẽ là cái mà client kết nối tới.

Return Value

Trả về không nếu thành công, hoặc `-1` nếu lỗi (và `errno` sẽ được gán phù hợp).

Example

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, "3490", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype,
    res->ai_protocol);

// bind it to the port we passed in to getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);

listen(sockfd, 10); // set sockfd up to be a server socket

// then have an accept() loop down here somewhere
```

See Also

`accept()`, `bind()`, `socket()`

9.16 `perror()`, `strerror()`

In một lỗi dưới dạng chuỗi người đọc được

Synopsis

```
#include <stdio.h>
#include <string.h>   // for strerror()

void perror(const char *s);
char *strerror(int errnum);
```

Description

Vì rất nhiều hàm trả về `-1` khi lỗi và đặt giá trị của biến `errno` thành một số nào đó, sẽ tuyệt nếu bạn có thể dễ dàng in nó ra ở dạng có ý nghĩa với bạn.

Rất may, `perror()` làm điều đó. Nếu bạn muốn in thêm mô tả trước lỗi, bạn có thể trỏ tham số `s` tới đó (hoặc để `s` là `NULL` và sẽ không in gì thêm).

Nói gọn, hàm này nhận các giá trị `errno`, như `ECONNRESET`, và in chúng đẹp đẽ, như "Connection reset by peer."

Hàm `strerror()` rất giống `perror()`, chỉ khác là nó trả về một con trỏ tới chuỗi thông báo lỗi cho giá trị đã cho (bạn thường truyền biến `errno`).

Return Value

`strerror()` trả về một con trỏ tới chuỗi thông báo lỗi.

Example

```
int s;

s = socket(PF_INET, SOCK_STREAM, 0);

if (s == -1) { // some error has occurred
    // prints "socket error: " + the error message:
    perror("socket error");
}

// similarly:
if (listen(s, 10) == -1) {
    // this prints "an error: " + the error message from errno:
    printf("an error: %s\n", strerror(errno));
}
```

See Also

`errno`

9.17 `poll()`

Kiểm tra sự kiện trên nhiều socket cùng lúc

Synopsis

```
#include <sys/poll.h>

int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

Description

Hàm này rất giống `select()` ở chỗ cả hai đều theo dõi các tập file descriptor để có sự kiện, như dữ liệu đi tới sẵn sàng để `recv()`, socket sẵn sàng để `send()` dữ liệu tới, dữ liệu out-of-band sẵn sàng để `recv()`, lỗi, vân vân.

Ý tưởng cơ bản là bạn truyền một mảng `nfds` cái `struct pollfd` trong `ufds`, cùng với một timeout tính theo millisecond (1000 millisecond một giây). `timeout` có thể âm nếu bạn muốn chờ mãi. Nếu không có sự kiện nào xảy ra trên bất kỳ socket descriptor nào trước khi timeout, `poll()` sẽ trả về.

Mỗi phần tử trong mảng `struct pollfd` đại diện cho một socket descriptor, và chứa các trường sau:

```
struct pollfd {
    int fd;         // the socket descriptor
    short events;   // bitmap of events we're interested in
    short revents;  // after return, bitmap of events that occurred
};
```

Trước khi gọi `poll()`, nạp `fd` bằng socket descriptor (nếu bạn gán `fd` thành một số âm, `struct pollfd` này bị bỏ qua và trường `revents` được gán thành không) rồi dựng trường `events` bằng cách OR bitwise các macro sau:

Macro	Mô tả
`POLLIN`	Báo cho tôi khi có dữ liệu sẵn sàng để `recv()` trên socket này.
`POLLOUT`	Báo cho tôi khi tôi có thể `send()` dữ liệu tới socket này mà không bị block.
`POLLPRI`	Báo cho tôi khi có dữ liệu out-of-band sẵn sàng để `recv()` trên socket này.

Khi `poll()` trả về, trường `revents` sẽ được dựng như một phép OR bitwise của các trường trên, cho bạn biết descriptor nào thật sự đã có sự kiện đó xảy ra. Thêm nữa, các trường khác này có thể xuất hiện:

Macro	Mô tả
`POLLERR`	Đã có lỗi trên socket này.
`POLLHUP`	Đầu remote của kết nối đã cúp máy.
`POLLNVAL`	Có gì đó sai với socket descriptor `fd`, có thể nó chưa khởi tạo?

Return Value

Trả về số phần tử trong mảng `ufds` đã có sự kiện xảy ra; số này có thể là không nếu timeout đã xảy ra. Cũng trả về `-1` nếu lỗi (và `errno` sẽ được gán phù hợp).

Example

```
int s1, s2;
int rv;
char buf1[256], buf2[256];
struct pollfd ufds[2];

s1 = socket(PF_INET, SOCK_STREAM, 0);
s2 = socket(PF_INET, SOCK_STREAM, 0);
```

```
// pretend we've connected both to a server at this point
//connect(s1, ...)...
//connect(s2, ...)...

// set up the array of file descriptors.
//
// in this example, we want to know when there's normal or
// out-of-band (OOB) data ready to be recv()'d...

ufds[0].fd = s1;
ufds[0].events = POLLIN | POLLPRI; // check for normal or OOB

ufds[1].fd = s2;
ufds[1].events = POLLIN; // check for just normal data

// wait for events on the sockets, 3.5 second timeout
rv = poll(ufds, 2, 3500);

if (rv == -1) {
    perror("poll"); // error occurred in poll()
} else if (rv == 0) {
    printf("Timeout occurred! No data after 3.5 seconds.\n");
} else {
    // check for events on s1:
    if (ufds[0].revents & POLLIN) {
        recv(s1, buf1, sizeof buf1, 0); // receive normal data
    }
    if (ufds[0].revents & POLLPRI) {
        recv(s1, buf1, sizeof buf1, MSG_OOB); // out-of-band data
    }

    // check for events on s2:
    if (ufds[1].revents & POLLIN) {
        recv(s1, buf2, sizeof buf2, 0);
    }
}
```

See Also

`select()`

9.18 `recv()`, `recvfrom()`

Nhận dữ liệu trên socket

Synopsis

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recv(int s, void *buf, size_t len, int flags);
ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);
```

Description

Khi bạn đã có socket dựng lên và đang kết nối, bạn có thể đọc dữ liệu đi tới từ đầu bên kia bằng `recv()` (cho socket TCP `SOCK_STREAM`) và `recvfrom()` (cho socket UDP `SOCK_DGRAM`).

Cả hai hàm đều nhận socket descriptor `s`, một con trỏ tới buffer `buf`, kích thước (tính theo byte) của buffer `len`, và một tập `flags` điều khiển cách các hàm hoạt động.

Ngoài ra, `recvfrom()` nhận thêm một `struct sockaddr*`, `from` sẽ cho bạn biết dữ liệu đến từ đâu, và sẽ điền `fromlen` bằng kích thước của `struct sockaddr`. (Bạn cũng phải khởi tạo `fromlen` bằng kích thước của `from` hoặc `struct sockaddr`.)

Vậy những cờ kỳ diệu nào bạn có thể truyền vào hàm này? Đây là vài cái, nhưng bạn nên kiểm tra man page cục bộ của mình để biết thêm và cái gì thật sự được hỗ trợ trên hệ thống của bạn. Bạn OR bitwise chúng lại với nhau, hoặc chỉ gán `flags` thành `0` nếu muốn nó là `recv()` vani bình thường.

Macro	Mô tả
`MSG_OOB`	Nhận dữ liệu Out of Band. Đây là cách lấy dữ liệu đã được gửi cho bạn với cờ `MSG_OOB` trong `send()`. Ở đầu nhận, signal `SIGURG` sẽ được raise báo cho bạn rằng có dữ liệu khẩn. Trong handler cho signal đó, bạn có thể gọi `recv()` với cờ `MSG_OOB` này.
`MSG_PEEK`	Nếu bạn muốn gọi `recv()` "chỉ để giả bộ", bạn có thể gọi với cờ này. Cái này sẽ cho bạn biết có gì đang đợi trong buffer khi bạn gọi `recv()` "thật" (tức là *không có* cờ `MSG_PEEK`). Nó giống bản xem trước cho lời gọi `recv()` kế tiếp.
`MSG_WAITALL`	Bảo `recv()` không trả về cho đến khi đã nhận được toàn bộ dữ liệu bạn chỉ định trong tham số `len`. Nó sẽ bỏ qua ý muốn của bạn trong hoàn cảnh cực đoan, ví dụ nếu một signal ngắt lời gọi hoặc nếu có lỗi xảy ra hoặc nếu đầu remote đóng kết nối, vân vân. Đừng giận nó.

Khi bạn gọi `recv()`, nó sẽ block cho đến khi có dữ liệu để đọc. Nếu bạn không muốn block, đặt socket thành non-blocking hoặc kiểm tra bằng `select()` hay `poll()` để xem có dữ liệu đi tới không trước khi gọi `recv()` hoặc `recvfrom()`.

Return Value

Trả về số byte thật sự đã nhận (có thể ít hơn số bạn yêu cầu trong tham số `len`), hoặc `-1` nếu lỗi (và `errno` sẽ được gán phù hợp).

Nếu đầu remote đã đóng kết nối, `recv()` sẽ trả về `0`. Đây là cách thường dùng để xác định đầu remote đã đóng kết nối chưa. Bình thường là tốt, cưng!

Example

```
// stream sockets and recv()

struct addrinfo hints, *res;
int sockfd;
char buf[512];
int byte_count;

// get host info, make socket, and connect it
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
getaddrinfo("www.example.com", "3490", &hints, &res);
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
connect(sockfd, res->ai_addr, res->ai_addrlen);

// all right! now that we're connected, we can receive some data!
byte_count = recv(sockfd, buf, sizeof buf, 0);
printf("recv()'d %d bytes of data in buf\n", byte_count);
```

```
// datagram sockets and recvfrom()

struct addrinfo hints, *res;
int sockfd;
int byte_count;
socklen_t fromlen;
struct sockaddr_storage addr;
char buf[512];
char ipstr[INET6_ADDRSTRLEN];

// get host info, make socket, bind it to port 4950
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_DGRAM;
hints.ai_flags = AI_PASSIVE;
getaddrinfo(NULL, "4950", &hints, &res);
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);

// no need to accept(), just recvfrom():

fromlen = sizeof addr;
byte_count = recvfrom(sockfd, buf, sizeof buf, 0, &addr, &fromlen);

printf("recv()'d %d bytes of data in buf\n", byte_count);
printf("from IP address %s\n",
    inet_ntop(addr.ss_family,
        addr.ss_family == AF_INET?
            ((struct sockaddr_in *)&addr)->sin_addr:
            ((struct sockaddr_in6 *)&addr)->sin6_addr,
        ipstr, sizeof ipstr);
```

See Also

`send()`, `sendto()`, `select()`, `poll()`, Blocking

9.19 `select()`

Kiểm tra xem các socket descriptor có sẵn sàng đọc/ghi không

Synopsis

```
#include <sys/select.h>

int select(int n, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

FD_SET(int fd, fd_set *set);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

Description

Hàm `select()` cho bạn cách kiểm tra đồng thời nhiều socket xem chúng có dữ liệu đang đợi được `recv()` không, hoặc bạn có thể `send()` dữ liệu cho chúng mà không bị block không, hoặc có exception nào xảy ra không.

Bạn điền tập socket descriptor của mình bằng các macro, như `FD_SET()` ở trên. Khi đã có tập, bạn truyền nó vào hàm qua một trong các tham số sau: `readfds` nếu bạn muốn biết khi nào bất kỳ socket nào trong tập sẵn sàng để `recv()` dữ liệu, `writefds` nếu bất kỳ socket nào sẵn sàng để `send()` dữ liệu, và/hoặc `exceptfds` nếu bạn cần biết khi nào có exception (lỗi) xảy ra trên bất kỳ socket nào. Bất kỳ hoặc tất cả tham số này có thể là `NULL` nếu bạn không quan tâm đến loại sự kiện đó. Sau khi `select()` trả về, giá trị trong các tập sẽ bị thay đổi để cho biết cái nào sẵn sàng đọc hoặc ghi, và cái nào có exception.

Tham số đầu tiên, `n`, là socket descriptor có số cao nhất (đều là `int`, nhớ chứ?) cộng một.

Cuối cùng, `struct timeval`, `timeout`, ở cuối, cái này cho bạn bảo `select()` kiểm tra các tập này bao lâu. Nó sẽ trả về sau khi timeout, hoặc khi có sự kiện xảy ra, cái nào đến trước. `struct timeval` có hai trường: `tv_sec` là số giây, cộng thêm `tv_usec`, số microsecond (1.000.000 microsecond một giây).

Các macro trợ giúp làm như sau:

Macro	Mô tả
`FD_SET(int fd, fd_set *set);`	Thêm `fd` vào `set`.
`FD_CLR(int fd, fd_set *set);`	Bỏ `fd` khỏi `set`.
`FD_ISSET(int fd, fd_set *set);`	Trả về true nếu `fd` nằm trong `set`.
`FD_ZERO(fd_set *set);`	Xóa toàn bộ phần tử khỏi `set`.

Lưu ý cho người dùng Linux: `select()` của Linux có thể trả về "sẵn-sàng-đọc" rồi thật ra không sẵn sàng đọc, khiến lời gọi `read()` theo sau bị block. Bạn có thể khắc phục bug này bằng cách bật cờ `O_NONBLOCK` trên socket nhận để nó trả lỗi với `EWOULDBLOCK`, rồi bỏ qua lỗi này nếu nó xảy ra. Xem man page của `fcntl()` để biết thêm về cách đặt socket thành non-blocking.

Return Value

Trả về số descriptor trong tập nếu thành công, `0` nếu đã đến timeout, hoặc `-1` nếu lỗi (và `errno` sẽ được gán phù hợp). Ngoài ra, các tập bị sửa để cho biết socket nào sẵn sàng.

Example

```
int s1, s2, n;
fd_set readfds;
struct timeval tv;
char buf1[256], buf2[256];

// pretend we've connected both to a server at this point
//s1 = socket(...);
//s2 = socket(...);
//connect(s1, ...)...
//connect(s2, ...)...

// clear the set ahead of time
FD_ZERO(&readfds);

// add our descriptors to the set
FD_SET(s1, &readfds);
FD_SET(s2, &readfds);

// since we got s2 second, it's the "greater", so we use that for
// the n param in select()
n = s2 + 1;

// wait until either socket has data ready to be recv()d
// (timeout 10.5 secs)
tv.tv_sec = 10;
tv.tv_usec = 500000;
rv = select(n, &readfds, NULL, NULL, &tv);

if (rv == -1) {
    perror("select"); // error occurred in select()
} else if (rv == 0) {
    printf("Timeout occurred! No data after 10.5 seconds.\n");
} else {
    // one or both of the descriptors have data
    if (FD_ISSET(s1, &readfds)) {
        recv(s1, buf1, sizeof buf1, 0);
    }
    if (FD_ISSET(s2, &readfds)) {
        recv(s2, buf2, sizeof buf2, 0);
    }
```

```
}
```

See Also

`poll()`

9.20 `setsockopt()`, `getsockopt()`

Đặt các tùy chọn khác nhau cho một socket

Synopsis

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(int s, int level, int optname, void *optval,
               socklen_t *optlen);
int setsockopt(int s, int level, int optname, const void *optval,
               socklen_t optlen);
```

Description

Socket là thứ khá có thể cấu hình. Thật ra, chúng có thể cấu hình đến mức tôi thậm chí sẽ không bao phủ hết ở đây. Có lẽ dù sao cũng tùy hệ thống. Nhưng tôi sẽ nói về phần cơ bản.

Rõ ràng, các hàm này lấy và đặt các tùy chọn nhất định trên một socket. Trên máy Linux, tất cả thông tin về socket nằm trong man page cho socket ở phần 7. (Gõ: "`man 7 socket`" để có hết mấy món ngon này.)

Về tham số, `s` là socket bạn đang nói đến, level nên được gán thành `SOL_SOCKET`. Rồi bạn đặt `optname` thành tên bạn quan tâm. Lại nữa, xem man page của bạn để có tất cả tùy chọn, nhưng đây là vài cái vui nhất:

`optname`	Mô tả
`SO_BINDTODEVICE`	Bind socket này vào tên thiết bị ký hiệu như `eth0` thay vì dùng `bind()` để bind nó vào địa chỉ IP. Gõ lệnh `ifconfig` trên Unix để xem tên thiết bị.
`SO_REUSEADDR`	Cho phép socket khác `bind()` vào port này, trừ khi đã có một socket đang lắng nghe tích cực bind vào port đó. Cái này giúp bạn vượt qua những thông báo lỗi "Address already in use" khi bạn thử khởi động lại server sau khi crash.
`SO_BROADCAST`	Cho phép socket UDP datagram (`SOCK_DGRAM`) gửi và nhận các gói tin được gửi đến và từ địa chỉ broadcast. Không làm gì, *KHÔNG LÀM GÌ!!*, với socket TCP stream! Hahaha!

Về tham số `optval`, nó thường là con trỏ tới một `int` cho biết giá trị đang nói. Cho boolean, không là false, khác không là true. Và đó là sự thật tuyệt đối, trừ khi nó khác trên hệ thống của bạn. Nếu không có tham số nào cần truyền, `optval` có thể là `NULL`.

Tham số cuối cùng, `optlen`, nên được gán thành độ dài của `optval`, có lẽ là `sizeof(int)`, nhưng thay đổi tùy tùy chọn. Lưu ý rằng trong trường hợp `getsockopt()`, đây là con trỏ tới một `socklen_t`, và

nó chỉ định kích thước tối đa của đối tượng sẽ được lưu trong `optval` (để ngăn buffer overflow). Và `getsockopt()` sẽ sửa giá trị của `optlen` để phản ánh số byte thật sự đã đặt.

Cảnh báo: trên một số hệ thống (đặc biệt là Sun và Windows), tùy chọn có thể là `char` thay vì `int`, và được gán, ví dụ, thành giá trị ký tự `'1'` thay vì giá trị `int` `1`. Lại nữa, kiểm tra man page của bạn để có thông tin thêm với "`man setsockopt`" và "`man 7 socket`"!

Return Value

Trả về không nếu thành công, hoặc `-1` nếu lỗi (và `errno` sẽ được gán phù hợp).

Example

```
int optval;
int optlen;
char *optval2;

// set SO_REUSEADDR on a socket to true (1):
optval = 1;
setsockopt(s1, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof optval);

// bind a socket to a device name (might not work on all systems):
optval2 = "eth1"; // 4 bytes long, so 4, below:
setsockopt(s2, SOL_SOCKET, SO_BINDTODEVICE, optval2, 4);

// see if the SO_BROADCAST flag is set:
getsockopt(s3, SOL_SOCKET, SO_BROADCAST, &optval, &optlen);
if (optval != 0) {
    print("SO_BROADCAST enabled on s3!\n");
}
```

See Also

`fcntl()`

9.21 `send()`, `sendto()`

Gửi dữ liệu ra qua socket

Synopsis

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int s, const void *buf, size_t len, int flags);
ssize_t sendto(int s, const void *buf, size_t len,
               int flags, const struct sockaddr *to,
               socklen_t tolen);
```

Description

Các hàm này gửi dữ liệu tới một socket. Nói chung, `send()` được dùng cho socket TCP `SOCK_STREAM` đã kết nối, còn `sendto()` được dùng cho socket datagram UDP `SOCK_DGRAM` không kết nối. Với socket không kết nối, bạn phải chỉ định đích đến của gói tin mỗi lần gửi, đó là lý do tham số cuối của `sendto()` định nghĩa gói tin đang đi đâu.

Với cả `send()` và `sendto()`, tham số `s` là socket, `buf` là con trỏ tới dữ liệu bạn muốn gửi, `len` là số byte muốn gửi, và `flags` cho phép bạn chỉ định thêm thông tin về cách dữ liệu được gửi. Gán `flags` thành không nếu bạn muốn nó là dữ liệu "bình thường". Đây là vài cờ hay dùng, nhưng kiểm tra man page `send()` cục bộ của bạn để biết thêm:

Macro	Mô tả
`MSG_OOB`	Gửi như dữ liệu "out of band". TCP hỗ trợ cái này, và đó là cách báo cho hệ thống nhận biết rằng dữ liệu này có độ ưu tiên cao hơn dữ liệu thường. Bên nhận sẽ nhận signal `SIGURG` và có thể nhận dữ liệu này mà không cần nhận hết phần dữ liệu thường còn lại trong hàng đợi trước.
`MSG_DONTROUTE`	Đừng gửi dữ liệu này qua router, chỉ giữ nó trong local.
`MSG_DONTWAIT`	Nếu `send()` sẽ block vì traffic đi ra đang bị tắc, làm nó trả về `EAGAIN`. Cái này giống như "bật non-blocking chỉ cho lần send này". Xem phần về blocking để biết chi tiết.
`MSG_NOSIGNAL`	Nếu bạn `send()` đến host remote không còn đang `recv()`, bạn thường sẽ nhận signal `SIGPIPE`. Thêm cờ này ngăn signal đó bị raise.

Return Value

Trả về số byte thật sự đã gửi, hoặc `-1` nếu lỗi (và `errno` sẽ được gán phù hợp). Lưu ý rằng số byte thật sự đã gửi có thể ít hơn số bạn yêu cầu gửi! Xem phần về xử lý `send()` một phần để có hàm trợ giúp vượt qua chuyện này.

Ngoài ra, nếu socket đã bị đóng bởi bất kỳ bên nào, process gọi `send()` sẽ nhận signal `SIGPIPE`. (Trừ khi `send()` được gọi với cờ `MSG_NOSIGNAL`.)

Example

```
int spatula_count = 3490;
char *secret_message = "The Cheese is in The Toaster";

int stream_socket, dgram_socket;
struct sockaddr_in dest;
int temp;

// first with TCP stream sockets:

// assume sockets are made and connected
//stream_socket = socket(...
//connect(stream_socket, ...

// convert to network byte order
temp = htonl(spatula_count);
// send data normally:
```

```
send(stream_socket, &temp, sizeof temp, 0);

// send secret message out of band:
send(stream_socket, secret_message, strlen(secret_message)+1,
        MSG_OOB);

// now with UDP datagram sockets:
//getaddrinfo(...
//dest = ... // assume "dest" holds the address of the destination
//dgram_socket = socket(...

// send secret message normally:
sendto(dgram_socket, secret_message, strlen(secret_message)+1, 0,
       (struct sockaddr*)&dest, sizeof dest);
```

See Also

`recv()`, `recvfrom()`

9.22 `shutdown()`

Dừng các lần send và receive tiếp theo trên socket

Synopsis

```
#include <sys/socket.h>

int shutdown(int s, int how);
```

Description

Đó! Tôi chịu hết nổi rồi! Không cho `send()` thêm nữa trên socket này, nhưng tôi vẫn muốn `recv()` dữ liệu trên đó! Hoặc ngược lại! Làm sao tôi làm được chuyện này?

Khi bạn `close()` một socket descriptor, nó đóng cả hai phía của socket cho đọc và ghi, và giải phóng socket descriptor. Nếu bạn chỉ muốn đóng một phía hoặc phía kia, bạn có thể dùng lời gọi `shutdown()` này.

Về tham số, `s` rõ ràng là socket bạn muốn thực hiện hành động này, và hành động đó là gì có thể chỉ định qua tham số `how`. `how` có thể là `SHUT_RD` để cấm thêm các `recv()`, `SHUT_WR` để cấm thêm các `send()`, hoặc `SHUT_RDWR` để cấm cả hai.

Lưu ý rằng `shutdown()` không giải phóng socket descriptor, nên bạn vẫn phải cuối cùng `close()` socket kể cả khi nó đã bị shut down hoàn toàn.

Đây là system call ít khi dùng.

Return Value

Trả về không nếu thành công, hoặc `-1` nếu lỗi (và `errno` sẽ được gán phù hợp).

Example

```
int s = socket(PF_INET, SOCK_STREAM, 0);

// ...do some send()s and stuff in here...

// and now that we're done, don't allow any more sends()s:
shutdown(s, SHUT_WR);
```

See Also

`close()`

9.23 `socket()`

Cấp phát một socket descriptor

Synopsis

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Description

Trả về một socket descriptor mới mà bạn có thể dùng để làm chuyện gì đó socket-kiểu. Đây thường là lời gọi đầu tiên trong quá trình đồ sộ viết một chương trình socket, và bạn có thể dùng kết quả cho các lời gọi tiếp theo tới `listen()`, `bind()`, `accept()`, hay nhiều hàm khác.

Trong cách dùng thông thường, bạn lấy giá trị cho các tham số này từ lời gọi `getaddrinfo()`, như trong ví dụ bên dưới. Nhưng bạn có thể tự điền bằng tay nếu thật sự muốn.

Tham số	Mô tả
`domain`	`domain` mô tả loại socket bạn quan tâm. Tin tôi đi, cái này có thể là nhiều thứ, nhưng vì đây là hướng dẫn về socket, nó sẽ là `PF_INET` cho IPv4, và `PF_INET6` cho IPv6.
`type`	Tham số `type` cũng có thể là nhiều thứ, nhưng bạn sẽ có lẽ gán nó thành `SOCK_STREAM` cho socket TCP đáng tin (`send()`, `recv()`) hoặc `SOCK_DGRAM` cho socket UDP nhanh không đáng tin (`sendto()`, `recvfrom()`). (Một kiểu socket thú vị khác là `SOCK_RAW` có thể dùng để dựng gói tin bằng tay. Khá ngầu.)
`protocol`	Cuối cùng, tham số `protocol` cho biết protocol nào dùng với một kiểu socket nhất định. Như tôi đã nói, ví dụ, `SOCK_STREAM` dùng TCP. Rất may cho bạn, khi dùng `SOCK_STREAM` hoặc `SOCK_DGRAM`, bạn chỉ cần gán protocol thành 0, và nó sẽ tự động dùng protocol phù hợp. Nếu không, bạn có thể dùng `getprotobyname()` để tra số protocol phù hợp.

Return Value

Socket descriptor mới để dùng trong các lời gọi tiếp theo, hoặc `-1` nếu lỗi (và `errno` sẽ được gán phù hợp).

Example

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;     // AF_INET, AF_INET6, or AF_UNSPEC
hints.ai_socktype = SOCK_STREAM; // SOCK_STREAM or SOCK_DGRAM

getaddrinfo("www.example.com", "3490", &hints, &res);

// make a socket using the information gleaned from getaddrinfo():
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

See Also

`accept()`, `bind()`, `getaddrinfo()`, `listen()`

9.24 `struct sockaddr` và đồng bọn

Các struct để xử lý địa chỉ internet

Synopsis

```
#include <netinet/in.h>

// All pointers to socket address structures are often cast to
// pointers to this type before use in various functions and system
// calls:

struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes of protocol address
};


// IPv4 AF_INET sockets:

struct sockaddr_in {
    short            sin_family;   // e.g. AF_INET, AF_INET6
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr, below
    char             sin_zero[8];  // zero this if you want to
```

```
};

struct in_addr {
    unsigned long s_addr;          // load with inet_pton()
};


// IPv6 AF_INET6 sockets:

struct sockaddr_in6 {
    u_int16_t       sin6_family;   // address family, AF_INET6
    u_int16_t       sin6_port;     // port number, network order
    u_int32_t       sin6_flowinfo; // IPv6 flow information
    struct in6_addr sin6_addr;     // IPv6 address
    u_int32_t       sin6_scope_id; // Scope ID
};

struct in6_addr {
    unsigned char   s6_addr[16];   // load with inet_pton()
};


// General socket address holding structure, big enough to hold
// either struct sockaddr_in or struct sockaddr_in6 data:

struct sockaddr_storage {
    sa_family_t  ss_family;     // address family

    // all this is padding, implementation specific, ignore it:
    char      __ss_pad1[_SS_PAD1SIZE];
    int64_t   __ss_align;
    char      __ss_pad2[_SS_PAD2SIZE];
};
```

Description

Đây là các struct cơ bản cho tất cả syscall và hàm xử lý địa chỉ internet. Bạn sẽ thường dùng `getaddrinfo()` để điền các struct này, rồi sẽ đọc chúng khi cần.

Trong bộ nhớ, `struct sockaddr_in` và `struct sockaddr_in6` chia sẻ cùng phần đầu struct với `struct sockaddr`, và bạn có thể tự do cast con trỏ của một kiểu sang kiểu kia mà không gây hại gì, trừ khả năng tận thế vũ trụ.

Nói đùa thôi về chuyện tận thế vũ trụ... nếu vũ trụ thực sự tận thế khi bạn cast một `struct sockaddr_in*` sang `struct sockaddr*`, tôi hứa với bạn đó là trùng hợp thuần túy và bạn thậm chí không nên lo về nó.

Vậy, với chuyện đó trong đầu, nhớ rằng mỗi khi một hàm nói rằng nó nhận `struct sockaddr*` bạn có thể cast `struct sockaddr_in*`, `struct sockaddr_in6*`, hoặc `struct sockaddr_storage*` của mình sang kiểu đó một cách dễ dàng và an toàn.

`struct sockaddr_in` là struct được dùng với địa chỉ IPv4 (ví dụ "192.0.2.10"). Nó chứa họ địa chỉ (`AF_INET`), một port trong `sin_port`, và địa chỉ IPv4 trong `sin_addr`.

Cũng có trường `sin_zero` trong `struct sockaddr_in` mà một số người quả quyết phải được gán thành

không. Người khác không quả quyết gì về nó (tài liệu Linux thậm chí không nhắc đến nó), và gán nó thành không có vẻ không thực sự cần thiết. Vậy, nếu bạn thích, cứ gán nó thành không bằng `memset()`.

Giờ, cái `struct in_addr` đó là một con quái vật lạ trên các hệ thống khác nhau. Đôi khi nó là một `union` điên rồ với đủ loại `#define` và nhảm nhí khác. Nhưng cái bạn nên làm là chỉ dùng trường `s_addr` trong struct này, vì nhiều hệ thống chỉ cài đặt mỗi trường đó.

`struct sockaddr_in6` và `struct in6_addr` rất giống, chỉ là chúng được dùng cho IPv6.

`struct sockaddr_storage` là struct bạn có thể truyền cho `accept()` hoặc `recvfrom()` khi bạn đang cố viết code độc lập với phiên bản IP và bạn không biết địa chỉ mới sẽ là IPv4 hay IPv6. Struct `sockaddr_storage` đủ lớn để chứa cả hai kiểu, khác với `struct sockaddr` gốc nhỏ.

Example

```
// IPv4:

struct sockaddr_in ip4addr;
int s;

ip4addr.sin_family = AF_INET;
ip4addr.sin_port = htons(3490);
inet_pton(AF_INET, "10.0.0.1", &ip4addr.sin_addr);

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip4addr, sizeof ip4addr);
```

```
// IPv6:

struct sockaddr_in6 ip6addr;
int s;

ip6addr.sin6_family = AF_INET6;
ip6addr.sin6_port = htons(4950);
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &ip6addr.sin6_addr);

s = socket(PF_INET6, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip6addr, sizeof ip6addr);
```

See Also

`accept()`, `bind()`, `connect()`, `inet_aton()`, `inet_ntoa()`

Chapter 10

Tài Liệu Tham Khảo Thêm

Bạn đã đi được tới đây, và giờ bạn đang gào lên đòi thêm! Còn có thể đi đâu nữa để học thêm về mấy thứ này?

10.1 Sách

Với mấy cuốn sách giấy kiểu cũ mà bạn cầm được trên tay, hãy thử một số cuốn hay ho dưới đây. Các liên kết này dẫn tới affiliate link của một hiệu sách nổi tiếng, cho tôi chút hoa hồng kha khá. Còn nếu bạn chỉ đơn giản là thấy rộng rãi, bạn có thể paypal donate vào `beej@beej.us`. `:-)`

Unix Network Programming, tập 1-2 của W. Richard Stevens. Xuất bản bởi Addison-Wesley Professional và Prentice Hall. ISBN cho tập 1-2: 978-0131411555[1], 978-0130810816[2].

Internetworking with TCP/IP, tập I của Douglas E. Comer. Xuất bản bởi Pearson. ISBN 978-0136085300[3].

TCP/IP Illustrated, tập 1-3 của W. Richard Stevens và Gary R. Wright. Xuất bản bởi Addison Wesley. ISBN cho các tập 1, 2, và 3 (và bộ 3 tập): 978-0201633467[4], 978-0201633542[5], 978-0201634952[6], (978-0201776317[7]).

TCP/IP Network Administration của Craig Hunt. Xuất bản bởi O'Reilly & Associates, Inc. ISBN 978-0596002978[8].

Advanced Programming in the UNIX Environment của W. Richard Stevens. Xuất bản bởi Addison Wesley. ISBN 978-0321637734[9].

10.2 Tham Khảo Trên Web

Trên web:

BSD Sockets: A Quick And Dirty Primer[10] (Có cả thông tin về lập trình hệ thống Unix nữa!)

The Unix Socket FAQ[11]

[1] https://beej.us/guide/url/unixnet1
[2] https://beej.us/guide/url/unixnet2
[3] https://beej.us/guide/url/intertcp1
[4] https://beej.us/guide/url/tcpi1
[5] https://beej.us/guide/url/tcpi2
[6] https://beej.us/guide/url/tcpi3
[7] https://beej.us/guide/url/tcpi123
[8] https://beej.us/guide/url/tcpna
[9] https://beej.us/guide/url/advunix
[10] https://cis.temple.edu/~giorgio/old/cis307s96/readings/docs/sockets.html
[11] https://developerweb.net/?f=70

TCP/IP FAQ[12]

The Winsock FAQ[13]

Và đây là một số trang Wikipedia liên quan:

Berkeley Sockets[14]

Internet Protocol (IP)[15]

Transmission Control Protocol (TCP)[16]

User Datagram Protocol (UDP)[17]

Client-Server[18]

Serialization[19] (đóng gói và mở gói dữ liệu)

10.3 RFCs

RFCs[20], hàng thật đây! Đây là những tài liệu mô tả các số được gán, API lập trình, và các giao thức được dùng trên Internet. Tôi đã đưa vào đây liên kết tới một vài cái cho bạn thưởng thức, nên lấy một xô bỏng ngô và đội mũ suy nghĩ vào:

RFC 1[21], RFC đầu tiên; nó cho bạn hình dung về "Internet" trông như thế nào ngay khi nó vừa ra đời, và một cái nhìn thoáng qua về việc nó được thiết kế từ con số không ra sao. (RFC này đã hoàn toàn lỗi thời, hiển nhiên rồi!)

RFC 768[22], User Datagram Protocol (UDP)

RFC 791[23], Internet Protocol (IP)

RFC 793[24], Transmission Control Protocol (TCP)

RFC 854[25], Giao thức Telnet

RFC 959[26], File Transfer Protocol (FTP)

RFC 1350[27], Trivial File Transfer Protocol (TFTP)

RFC 1459[28], Internet Relay Chat Protocol (IRC)

RFC 1918[29], Phân bổ địa chỉ cho mạng Internet riêng

RFC 2131[30], Dynamic Host Configuration Protocol (DHCP)

[12] http://www.faqs.org/faqs/internet/tcp-ip/tcp-ip-faq/part1/
[13] https://tangentsoft.net/wskfaq/
[14] https://en.wikipedia.org/wiki/Berkeley_sockets
[15] https://en.wikipedia.org/wiki/Internet_Protocol
[16] https://en.wikipedia.org/wiki/Transmission_Control_Protocol
[17] https://en.wikipedia.org/wiki/User_Datagram_Protocol
[18] https://en.wikipedia.org/wiki/Client-server
[19] https://en.wikipedia.org/wiki/Serialization
[20] https://www.rfc-editor.org/
[21] https://tools.ietf.org/html/rfc1
[22] https://tools.ietf.org/html/rfc768
[23] https://tools.ietf.org/html/rfc791
[24] https://tools.ietf.org/html/rfc793
[25] https://tools.ietf.org/html/rfc854
[26] https://tools.ietf.org/html/rfc959
[27] https://tools.ietf.org/html/rfc1350
[28] https://tools.ietf.org/html/rfc1459
[29] https://tools.ietf.org/html/rfc1918
[30] https://tools.ietf.org/html/rfc2131

RFC 9110[31], Hypertext Transfer Protocol (HTTP)

RFC 2821[32], Simple Mail Transfer Protocol (SMTP)

RFC 3330[33], Các địa chỉ IPv4 dùng cho mục đích đặc biệt

RFC 3493[34], Basic Socket Interface Extensions cho IPv6

RFC 3542[35], Advanced Sockets Application Program Interface (API) cho IPv6

RFC 3849[36], Tiền tố địa chỉ IPv6 dành riêng cho tài liệu

RFC 3920[37], Extensible Messaging and Presence Protocol (XMPP)

RFC 3977[38], Network News Transfer Protocol (NNTP)

RFC 4193[39], Unique Local IPv6 Unicast Addresses

RFC 4506[40], External Data Representation Standard (XDR)

IETF có một công cụ trực tuyến khá hay để tìm kiếm và duyệt RFC[41].

[31] https://tools.ietf.org/html/rfc9110
[32] https://tools.ietf.org/html/rfc2821
[33] https://tools.ietf.org/html/rfc3330
[34] https://tools.ietf.org/html/rfc3493
[35] https://tools.ietf.org/html/rfc3542
[36] https://tools.ietf.org/html/rfc3849
[37] https://tools.ietf.org/html/rfc3920
[38] https://tools.ietf.org/html/rfc3977
[39] https://tools.ietf.org/html/rfc4193
[40] https://tools.ietf.org/html/rfc4506
[41] https://tools.ietf.org/rfc/

Index