برنامه نویسی شبکه در یونیکس با ساکت های اینترنتی[1] در C/C++

افرادی که با زبان های برنامه نویسی کار کرده اند. مطمئنا یکی از بزرگترین دردسرهایشان برنامه نویسی ساکت یا شبکه بوده است. نکات و تکنیک های فوق تصوری در برنامه نویسی شبکه، این دنیای پهناور، وجود دارند. لذا در این مقاله سعی می کنیم مفاهیم پایه ای جهت برنامه شبکه در سیستم عامل یونیکس را بررسی کنیم. با کسب مهارت در برنامه نویسی در سیستم های تحت یونیکس یا لینوکس، می توان براحتی برنامه نویسی در سیستم های ویندوز را نیز تمرین و تجربه کرد.

۱. مقدمه

آیا با مباحث Socket Programming کار کرده اید؟ شما می خواهید برنامه نویسی اینترنتی انجام دهید، اما وقتی برای سیر در دریایی از دستورالعمل ها ندارید و نمی توانید بفهمید، آیا شما مجبور هستید bind() را قبل از connect() فراخوانی کنید یا خیر و ... . در این مقاله سعی می کنیم از رازها (!) و مشکلاتی که در امر برنامه شبکه پنهان هستند پرده برداری کنیم.

۱/۱ خواننده

این مقاله به عنوان یک اشکال زدا نوشته شده و نه به عنوان یک مرجع. این مقاله می توانید برای اشخاصی که تازه شروع به امر برنامه نویسی شبکه کرده اند مفید واقع شود، با این وجود که شامل راهنمایی های کامل در برنامه نویسی شبکه نیست.

۱/۲ پلاتفرم و کامپایلر

کدهای موجود در این مقاله تحت یک سیستم عامل لینوکس و با کامپایلر gcc کامپایل گشته اند. اما، باید روی هر پلاتفرمی که از gcc استفاده می کند کارکرد داشته باشند. بدیهی است که، اگر برای ویندوز برنامه نویسی می کنید کارکرد نخواهد داشت – قسمت برنامه نویسی ویندوز را ببینید.

۱/۳ یادداشت ها برای برنامه نویسان Solaris/SunOS

هنگام کامپایل برای Solaris و SunOS، شما باید سوئیچ های command-line بیشتری را تعیین کنید تا کتابخانه های صحیح را با هم اتصال یا لینک بدهید. برای انجام این کار، تنها عبارت "-lnsl -lsocket -lresolv" را به انتهای دستور کامپایل اضافه کنید، بصورت زیر:

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

اگر هنوز هم خطا گرفته می شود، عبارت "-lxnet" را نیز به انتهای آن دستور اضافه کنید. به شخصه نمی دانم که عبارت اخیر دقیقا چکاری انجام می دهد، اما نظر می رود که بعضی افراد به آن احتیاج داشته باشند. مکان دیگری که ممکن است در آن مشکلاتی پیدا کنید، در فراخوانی setsockopt() می باشد. پروتوتایپ (الگوی نخست) در سیستم عامل لینوکس من با این موارد تفاوت دارد، بنابراین به جای:

```
int yes=1;
```

عبارت زیر را وارد کنید:

---

[1] Internet Sockets

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

چون من سیستم عامل Sun را نداشتم، هیچ کدام از اطلاعات فوق را چک نکرده ام – لذا اینها تنها اطلاعاتی بودند که از طریق ایمیل از افراد پرسیده ام.

۱/۴ یادداشت ها برای برنامه نویسان ویندوز

جدیدا تنفر خاصی نسبت به ویندوز پیدا کرده ام و شما را به استفاده از لینوکس، BSD یا یونیکس تشویق می کنم. اگرچه، می توان این موارد را تحت ویندوز هم استفاده کرد. ابتدا، تقریبا تمام فایل های هدر سیستمی را که در زیر به آنها اشاره کرده ام، ایگنور کنید. تمام چیزهایی که نیاز به include کردن آنها در برنامه دارید، هدر زیر می باشد:

```
#include <winsock.h>
```

صبر کنید! شما قبل از اینکه کار دیگری با کتابخانه socket ها انجام دهید باید یک فراخوانی به WSAStartup() نیز انجام دهید. کدی که این کار را انجام می دهد، چیزی شبیه به زیر است:

```
#include <winsock.h>

{
  WSADATA wsaData;   // if this doesn't work
  //WSAData wsaData; // then try this instead

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

همچنین مجبور هستید تا به کامپیوتر بگویید که در کتابخانه Winsock اتصال برقرار کند که معمولا wsock32.lib یا winsock32.lib یا یک چنین چیزی نامیده می شود. تحت VC++، این کار را می توانید به صورت زیر انجام دهید:

منوی Project را باز کرده، Settings... را انتخاب کنید و روی برگه Link کلیک کنید. سپس بدنبال جعبه ای با عنوان "Object/library modules" بگردید. آیتم "wsock32.lib" را به آن لیست اضافه کنید.

سرانجام، شما احتیاج به فراخوانی WSACleanup() دارید. برای جزئیات این کار می توانید از help های آنلاین موجود در سطح اینترنت کمک بگیرید.

پس از انجام این کار، بایستی مثال های موجود در این اشکالزدا اساسا کارکرد داشته باشند (بجز چند استثناء). برای یک مورد، شما نمی توانید از close() برای بستن یک ساکت استفاده کنید – بجای آن باید از closesocket() استفاده کرد. همچنین، select() تنها با توصیفگران ساکت[۲] کاکرد دارد، نه توصیفگران فایل (مثل 0 برای stdin).

متاسفانه فکر می کنم، ویندوز فراخوانی سیستمی fork() را که در بعضی از مثال ها استفاده شده است نداشته باشد. شاید شما مجبور به لینک شدن در یک کتابخانه POSIX یا ... باشید تا موجب کارکرد آن گردید یا بجای آن می توانید از CreateProcess() استفاده کنید. fork() هیچ آرگومانی را نمی پذیرد و CreateProcess() در حدود ۴۸ میلیون آرگومان میپذیرد. به دنیای شگفت انگیز از برنامه نویسی Win32 خوش آمدید!!

[2] Socket Descriptor

۲. Winsock چیست؟

شما همیشه صحبت هایی حول "ساکت ها یا Socket" را شنیده اید و می شنوید، و شاید از کارهای آنها شگفت زده شده باشید. به هر حال، تعریف دقیقی نمی توان از یک ساکت داشت اما می توان آنرا نسبی به صورت زیر تعریف کرد:

یک ساکت، راهی برای صحبت کردن با دیگر برنامه هایی است که از توصیفگران یونیکسی استاندارد فایل استفاده می کنند.

شاید این جمله مقداری گنگ بنظر رسد، در جواب باید گفت شاید شنیده باشید که بعضی افراد می گویند: "همه در یونیکس، یک فایل می باشد!". چیزی که این افراد در مورد آن صحبت می کنند، این واقعیت است که هنگامی انجام هر گونه عملیات I/O (ورودی/خروجی) توسط برنامه های یونیکسی، اینکار از طریق خواندن یا نوشتن در یک توصیفگر فایل (file descriptor) انجام می شود. یک توصیفگر فایل، یک عدد صحیح وابسته با یک فایل باز شده[۳] می باشد. اما، این فایل می تواند یک ارتباط شبکه ای، یک FIFO، یک Pipe، یک ترمینال، یک فایل واقعی روی دیسک یا هر چیز دیگری باشد. تمام چیزها در یونیکس یک فایل است! بنابراین هنگامی که شما می خواهید با برنامه دیگری از طریق اینترنت ارتباط برقرار کنید، اینکار را از طریق یک توصیفگر فایل انجام خواهید داد. حال سول این است که کجا می توان این توصیفگر فایل را برای ارتباط شبکه ای بدست آورد؟

احتمالا این آخرین سوالی است که ذهن شما را آزار می دهد. شما یک فراخوانی را به روتین سیستمی socket() ایجاد می کنید (یا انجام می دهید – هر دو به یک معنی هستند). این روتین، توصیفگر ساکت را برمی گرداندو شما ازطریق آن با استفاده از فراخوانی های خاص ساکت send() و recv() ارتباط برقرار می کنید. ممکن است سوال کنید، "اگر این یک توصیفگر فایل باشد، چرا من نمی توانم تنها از فراخوانی های معمول read() و write() برای ارتباط از طریق یک ساکت استفاده کنم؟"

جواب این است که، شما می توانید، اما send() و recv()، کنترل بیشتری را روی انتقال داده ها انجام می دهند. اگرچه انواع زیادی از ساکت ها وجود دارند:

- DARPA Internet Addresses (ساکت های اینترنتی)
- نام های مسیر روی یک گره محلی (ساکت های یونیکس)
- آدرس های CCITT X.25 (ساکت های X.25 – که شما می توانید با اطمینان آنها را نادیده گرفته و از آنها رد شوید).
- و دیگر ساکت هایی که به نسخه و نوع یونیکسی که شما اجرا می کنید بستگی دارند.

این مقاله تنها با اولین مورد سروکار دارد، یعنی ساکت های اینترنتی یا Internet Socket ها.

۲/۱ دو نوع از ساکت های اینترنتی

دو نوع از ساکت های اینترنتی وجود دارند؟ بلی، اما در حقیقت خیر. برای اینکه از سردرگمی نجات پیدا کنید (!) ما فقط درباره دو نوع از آنها صحبت می کنیم. استثنای این جمله در جاهایی است که ذکر می شود، "ساکت های خام" یا "Raw Socket" ها هم در این مورد بسیار قدرتمند هستند و لذا به این موارد هم باید توجه داشت.

[3] Open File

به هر حال، دو نوع یاد شده از ساکت ها، "ساکت های انسدادی یا جریانی"[۴] و "ساکت های دیتاگرامی"[۵] هستند که از این پس، به ترتیب تحت عناوین "SOCK STREAM" و "SOCK DGRAM" یاد می شوند. ساکت های دیتاگرامی بعضی مواقع "ساکت های بدون اتصال"[۶] نیز نامیده می شوند (اگرچه، در صورتی که واقعا بخواهید می توانید با connect() متصل شد).

ساکت های جریانی، جریان های ارتباطی در متصل دو مرحله ای[۷] هستند. اگر شما دو شی را به ترتیب 1 و ۲ به خروجی ببرید، آنها در طرف مقابل به ترتیب 2 و ۱ می رسند. همچنین این ساکت ها خطای آزاد[۸] هستند (هر خطایی که انجام می دهید، خیالات ذهن خودتان هستند و نمی توان آنها را در اینجا توضیح داد)!

تصویر ۱ – کپسوله سازی داده ها

چه چیزهایی از ساکت های جریانی استفاده می کنند؟ شاید نام برنامه telnet را شنیده یا از آن استفاده کرده باشید. این برنامه از ساکت های جریانی استفاده می کند. تمام کاراکترهایی که شما تایپ می کنید باید به همان ترتیبی که تایپ شده اند در طرف مقابل دریافت شوند. همچنین، مرورگرهای وب از پروتکل HTTP استفاده می کنند که این پروتکل برای دریافت صفحات از ساکت های جریانی استفاده می کند. در واقع، اگر شما به یک وبسایت روی پورت ۸۰ تلنت کنید و عبارت "GET /" را تایپ کنید، کدهای HTML را برای شما جمع آوری و به شما برمی گرداند!

چطور ساکت های جریانی این سطح بالا از کیفیت انتقال داده را بدست آورده اند؟ آنها از یک پروتکل به نام "Transmission Control Protocol" یا "TCP" استفاده می کنند (می توانید RFC-793 را برای جزئیات بیشتر روی TCP ببینید). TCP اطمینان حاصل می کند که داده های شما یکی پس از دیگری (به ترتریب) و به صورت error-free برسد. حتما کلمه "TCP" را بعنوان نیمی از عبارت "TCP/IP" شنیده اید که نیمه ی IP خلاصه ی "Internet Protocol" می باشد (RFC-791 را ببینید). پروتکل اینترنت یا IP اصولا با مسیریابی اینترنتی سروکار داشته و اساسا مسئول درستی و بی نقص بودن داده ها[۹] نیست.

اکنون راجع به ساکت های دیتاگرامی می خواهیم صحبت کنیم و اینکه چرا آنها بدون اتصال و غیرقابل اطمینان هستند. در این راستا، باید چندین حقیقت را متذکر شد: اگر شما یک دیتاگرام را ارسال کنید، *ممکن است* به مقصد خود برسد یا ممکن است بدون ترتیب و خارج از نظم به مقصد برسد. اگر داده ها به این صورت به مقصد برسد، داده های درون بسته، error-free خواهند بود.

همچنین ساکت های دیتاگرامی از IP برای مسیریابی استفاده می کنند، اما از TCP استفاده نمی کنند. آنها از "User Datagram Protocol" یا "UDP" استفاده می کنند (RFC-768 را ببینید). این ساکت ها به این دلیل بدون اتصال هستند که شما مجبور به نگهداری یک ارتباط باز هنگام کارکردن با ساکت های جریانی نیستید. شما تنها یک بسته را ساخته، یک هدر IP با اطلاعات توضیحی به آن اضافه کرده و آن را به بیرون می فرستید. هیچ ارتباطی نیاز نیست (یعنی تنها ساختن بسته در این مورد نظر بوده و به باز بودن ارتباط توجهی نشده است). اساسا این ساکت ها برای انتقالات اطلاعات به صورت بسته-به-بسته استفاده می شوند، مثل، tftp، bootp و ... . چطور این برنامه ها در صورتی که حتی امکان از دست رفتن دیتاگرام ها وجود دارد، کار می کنند!؟

---

[4] Stream Socket
[5] Datagram Socket
[6] Connectionless Socket
[7] two-way connected communication streams
[8] error free
[9] Data Intergrity

در جواب باید گفت که هر برنامه پروتکل خودش را در بالای UDP دارد. برای مثال، پروتکل tftp می گوید: بازای هر بسته که فرستاده می شود، گیرنده باید یک بسته را پس بفرستد. این بسته به معنی دریافت شدن بسته می باشد (یک بسته ACK). اگر فرستنده حقیقی بسته، هیچ جوابی را دریافت ندارد، بمدت ۵ ثانیه صبر کرده و پس از آن بسته را مجددا می فرستد و این عمل تا زمان دریافت بسته ACK ادامه خواهد یافت. این رویه تصدیقی[10]، هنگام اجرا و کارکردن با application های از نوع SOCK_DGRAM بسیار مهم است.

۲/۲ تئوری شبکه

چون فقط طرح بندی پروتکل ها را ذکر کردیم، لذا اکنون زمان صحبت درباره چگونگی کارکرد شبکه ها می باشد و سپس چندین مثال از چگونه ساخته شدن بسته های SOCK_DGRAM را خواهیم دید. در واقع، در صورتی که با مفاهیم شبکه آشنایی دارید می توانید از این قسمت رد شوید.

اکنون، زمان صحبت درباره کپسوله سازی داده ها یا Data Encapsulation می باشد[11]. این مبحث بسیار بسیار مهم می باشد (در صورتی که در ایالات شیکاگو و ... واحد شبکه را بگیرید، یکی از مهم ترین و اولین مباحث بوده و روی آن تاکید بسیار می شود). اساسا، این عملیات روند زیر را طی می کند:

یک بسته ساخته شده، توسط اولین پروتکل (در اینجا پروتکل TFTP)، در یک هدر (و ندرتا در یک فوتر) پوشش دار شده (encapsulate می شود) – یعنی اطلاعاتی به آن می چسبد. سپس کل آن (بهمراه هدر TFTP)، مجددا بوسیله پروتکل بعدی (UDP) کپسوله سازی شده و دوباره توسط پروتکل بعدی (IP) کپسوله سازی خواهد گشت. در این لحظه بوسیله آخرین پروتکل در لایه فیزیکی یا سخت افزاری (Ethernet)، مجددا عملیات کپسوله سازی انجام خواهد گشت.

هنگامی که کامپیوتر دیگر بسته را دریافت می کند، سخت افزار آن، هدر Ethernet، هسته یا kernel، هدرهای UDP و IP و برنامه ی TFTP، هدر TFTP را کَنده و سرانجام به داده می رسد.

اکنون در انتهای کار، می توانم درباره یک مدل شبکه بندی صحبت کنم. این مدل شبکه بندی، سیستم از عاملیت شبکه ای را توصیف کرده که نسبت به دیگر مدل ها منفعت های بسیاری دارد. برای مثال، شما می توانید برنامه های ساکتی[12] بنویسید که دقیقا کارهای مشابه را بدون حمل کردن شرایط چگونگی انتقال فیزیکی (serial، thin Ethernet، AUI و ...) داده انجام دهد، چرا که برنامه های موجود در لایه های پایین تر این کار را برای شما انجام خواهند داد. سخت افزار حقیقی و همبندی یا توپولوژی حقیقی شبکه، برای برنامه نویس شبکه واضح نیست. در زیر لایه های مختلف این مدل را می بینید (مدل هفت لایه ی OSI). مطمئنا این مدل را از امتحانات خود در کلاس های شبکه بیاد می آورید:

Application

Presentation

Session

Transport

Network

Data Link

---

[10] Acknowledgement

[11] در سایت کروز چندین presentation ارائه کرده ام که در این راستا بسیار مفید هستند.

[12] منظور برنامه هایی که با شبکه کار می کنند

Physical

لایه Physical، سخت افزار شبکه می باشد (serial، Ethernet و ...). لایه Application چیزی شبیه به لایه فیزیکی خواهد بود – جایی است که *کاربران* با شبکه فعل و انفعال دارند. اگر شما می خواهید واقعا این مدل را استفاده کنید، این مدل به عنوان یک راهنمای خودکار در تعمیر شبکه بسیار کلی است. یک مدل طرح بندی سازگارتر برای یونیکس ممکن چیزی به صورت زیر باشد:

Application Layer (telnet, ftp, etc.)

Host-to-Host Transport Layer (TCP, UDP)

Internet Layer (IP and routing)

Network Access Layer (Ethernet, ATM, or whatever)

در این نقطه از زمان، شما احتمالا می توانید چگونگی رابطه این لایه ها را در کپسوله سازی داده های واقعی را ببینید. اکنون می خواهیم ببینیم که برای ساخت یک بسته ساده چقدر کار باید انجام شود؟

شما برای اینکار، مجبور هستید که خودتان در هدرهای بسته با استفاده از "cat" عمل نوشتن را انجام دهید. تمام کاری که برای ساکت های جریانی باید انجام دهید، فرستادن داده ها با استفاده از send() می باشد. تمام کاری که برای ساکت های دیتاگرامی باید انجام دهید، کپسوله سازی بسته به شیوه خودتان، و سپس ارسال آن با استفاده از sendto() است. هسته (kernel)، لایه انتقال و لایه اینترنت را برای شما می سازد و سخت افزار لایه دستیابی شبکه را برای شما مهیا می سازد. تکنولوژی مدرن!

در اینجا به انتهای خلاصه ای از تئوری شبکه می رسیم. تنها چیزی که باید در این جا متذکر شوم این است که درباره مسیردهی ما هیچ صحبتی به میان نخواهیم آورد. روتر بسته را به هدر IP چسبانده و با جدول مسیریابی خود مشورت کرده (!) و ... (برای اطلاعات بیشتر و بحث های حرفه ای تر، پس از ورود به فاروم گروه کروز و بدنبال آن تیم شبکه، سوال های خود را مطرح سازید).

۳. ساختمان ها (struct) و بررسی و حمل داده ها (Data Handling)

اکنون باید درباره برنامه نویسی صحبت کنیم. در این قسمت، انواع داده ای مختلفی که بوسیله واسط ساکت ها استفاده می شود را متذکر خواهیم شد. چرا که بعضی از آنها بسیار مهم هستند. ابتدا، یک مورد آسان: یک توصیفگر ساکت. یک توصیفگر ساکت، گونه زیر می باشد:

int

تنها یک عدد صحیح عادی! از اینجا، چیزها مرموز و عجیب می شوند. باید این نکته را بدانید که، دو نوع مدل برای مرتب سازی بایت[۱۳] وجود دارد:

با ارزش ترین و مهم ترین بایت در ابتدا (بعضی مواقع یک اکتت یا هشتایی هم نامیده می شود)، یا کم ارزش ترین بایت در ابتدا. مدل قبلی، "ترتیب بایت شبکه ای"[۱۴] نامیده می شود. بعضی ماشین ها، اعدادشان را ذاتا به صورت ترتیب بایت شبکه ای ذخیره می کنند و بعضی دیگر خیر. هنگامی که ذکر می شود که یک چیز باید در یک ترتیب بایت شبکه ای باشد، شما مجبور هستید تابعی (از قبیل htons()) را برای تغییر آن از ترتیب بایت میزبان[۱۵] فراخوانی کنید. اگر در مقاله ذکر نشده بود که از ترتیب بایت شبکه ای استفاده کنید، آنوقت باید مقادیر را به صورت ترتیب بایت میزبان رها کنید. بعضی مواقع ترتیب بایت شبکه، تحت

[13] Byte Ordering
[14] Network Byte Order
[15] Host Byte Order

عنوان "Big-Endian Byte Order" نیز شناخته می شود (در مقابل ترتیب بایت میزبان که به آن "Little-Endian Byte Order" می گوییم). این ساختمان اطلاعات آدرسی ساکت[16] را برای بسیاری از انواع ساکت ها نگه داری می کند:

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

sa_family می تواند چیزهای متفاوتی باشد، اما در این مقاله برای هر کاری که ما انجام می دهیم، به صورت AF_INET خواهد بود. sa_data حاوی یک آدرس مقصد و یک شماره پورت برای ساکت می باشد. این مورد نسبتا غیراداره شدنی است، چرا که شما نمی خواهید آدرس را در sa_data به صورت دستی قرار دهید (یا آدرس را pack کنید).

برای کار با struct_sockaddr، برنامه نویس ها یک ساختمان موازی[17] با نام **sturct_sockaddr_in** ایجاد کرده اند (که در آن in مخفف عبارت Internet می باشد):

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

این ساختمان، ارجاع دادن عناصر آدرس ساکت را آسان تر می کند. به خاطر داشته باشید که sin_zero (که برای pad کردن ساختمان به طول struct_sockaddr می باشد) بایستی به تمام صفر با تابع memset() تنظیم شود. همچنین، یک اشاره گر به یک ساختمان sockaddr_in را می توان به یک اشاره به یک ساختمان sockaddr و بعکس معین کرد (این مورد مهم بود!). پس اگرچه socket() یک ساختمان sockaddr* را می خواهد، اما شما هنوز می توانید از یک ساختمان sockaddr_in استفاده کنید و آنرا در آخرین دقیق قالب بندی و معین کنید (cast)! همچنین، به خاطر داشته باشید که sin_family در یک ساختمان sockaddr، با یک sa_family رابطه داشته و بایستی به صورت "AF_INET" تنظیم شود. سرانجام، sin_port و sin_addr بایستی به صورت ترتیب بایت شبکه ای باشند! اما شاید بپرسید، چگونه کل ساختمان، struct in_addr sin_addr می تواند به صورت ترتیب بایت شبکه ای باشد؟ این سوال، بررسی و آزمایش دقیقی از ساختمان struct in_addr احتیاج دارد:

```
// Internet address (a structure for historical reasons)
struct in_addr {
    unsigned long s_addr; // that's a 32-bit long, or 4 bytes
};
```

اگر شما ina را طوری اعلان کرده اید که از نوع struct sockaddr_in باشد، آنوقت ina.sin_addr.s_addr به آدرس ۴ بایتی IP ارجاع دارد (در ترتیب بایت شبکه ای). به خاطر بسپارید که اگرسیستم شما هنوز از اتحاد[18] God-awful برای struct in_addr استفاده کند، شما هنوز می توانید آدرس ۴ بایتی IP را دقیقا به همان روشی که در بالا انجام شد، ارجاع دهید (این مسئله از #define ها ناشی می شود).

۳/۱ تبدیل قالب های اولیه

[16] Socket address Information
[17] Parallel
[18] Union

صحبت های بسیاری زیادی درباره تبدیل ترتیب بایت شبکه ای به میزبان شده است – اکنون زمان عمل است! دو گونه و نوع وجود دارند که شما می توانید به هم تبدیل کنید: short (دو بایت) و long (چهار بایت).

این توابع برای محیطهای بدون علامت (unsigned) نیز کار می کنند. شما می خواهید یک نوع short را از ترتیب بایت میزبان به شبکه تبدیل کنید. با "h" (خلاصه عبارت Host) شروع کرده، یک "to" به آن اضافه کرده و سپس بدنبال یک "n" (لاصه عبارت "Network") به آن اضافه می کنیم و سپس "s" (خلاصه "short") را به آن اضافه می کنیم. در نتیجه تبدیل نوع short از ترتیب بایت میزبان به شبکه به صورت زیر خواهد بود (بخوانید Host To Network Short):

h-to-n-s

پس کار تقریبا راحت است... شما می توانید هر ترکیبی از "n"، "h"، "s" و "l" را بکار ببرید، اما استثناهایی هم وجود دارد (که منطقی هم هستند). برای مثال، تابعی به نام stolh() (با خلاصه ای به صورت Short To Long Host) وجود ندارد. اما توابع زیر وجود دارند:

htons() -- "Host to Network Short"

htonl() -- "Host to Network Long"

ntohs() -- "Network to Host Short"

ntohl() -- "Network to Host Long"

شاید بپرسید، "اگر مجبور باشیم که ترتریب بایت را روی یک char تغییر دهیم، چکار باید بکنیم؟". آنگاه ممکن است فکر کنید که اصلا این امکان وجود ندارد. همچنین ممکن است فکر کنید که چون ماشین 68000 شما پیش از این از ترتیب بایت شبکه ای استفاده میکرده است، شما مجبور به فراخوانی htonl() روی آدرس های IP خود نیستید. ممکن است این فکر درست باشد، اما اگر شما سعی بر انتقال داده به ماشینی کنید که ترتیب بایت شبکه ای معکوسی دارد، در این صورت برنامه شما در کار خود موفق نخواهد بود. اما نباید زیاد بسته فکر کرد! ما در دنیای یونیکس هستیم! به خاطر داشته باشید که، بایت های خود را قبل از قرار دادن آنها روی شبکه بصورت ترتریب بایت شبکه ای در آورید.

آخرین نکته: چرا در یک struct sockaddr_in لازم است که sin_addr و sin_port به صورت ترتیب بایت شبکه ای باشند، اما در مورد sin_family احتیاج نیست؟

در جواب باید گفت که، sin_addr و sin_port به ترتیب در بسته در لایه های IP و UDP کپسوله شده اند. بنابراین، بایستی به صورت ترتیب بایت شبکه ای باشند. به هر حال، فیلدsin_family تنها بوسیله هسته استفاده می شود تا تعیین شود ساختمان حاوی چه نوعی از آدرس است، بنابراین باید به صورت ترتیب بایت میزبان باشد. همچنین، چون sin_family روی شبکه فرستاده نمی شود، لذا می تواند بصورت ترتیب بایت میزبان باشد.

۳/۲ آدرس های IP چگونگی کار کردن با آنها

خوشبختانه، گروهی از توابع وجود دارند که به شما اجازه دستکاری آدرس های IP را می دهند. احتیاجی نیست بصورت دستی آنها را درک کرده و با عملگر >> با آنها سروکله بزنید.

ابتدا، بیایید فرض کنیم که شما یک ساختمان ina از struct sockaddr_in دارید و یک آدرس IP بصورت "10.12.110.57" دارید که می خواهید اطلاعات را روی آن ذخیره کنید. تابعی که می خواهید از آن استفاده کنید یعنی inet_addr()، یک آدرس IP را که بصورت نمادگذاری عددی-نقطه ای می باشد، بصورت unsigned long تبدیل می کند. این تخصیص و عمل را می توان به صورت زیر انجام داد:

```
ina.sin_addr.s_addr = inet_addr("10.12.110.57");
```

به خاطر داشته باشید که inet_addr()، از قبل آدرس را بصورت ترتیب بایت شبکه ای برگشت می دهد – شما مجبور به فراخوانی htonl() نیستید. اکنون، کد فوق، زیاد مستحکم و قوی نیست، چرا که در آن error checking انجام می گردد. فرض کنید، inet_addr() یک 1- را به اشتباه برگرداند. اعداد باینری بیادتان آمد؟ دستور 1-(unsigned) تنها به آدرس IP، 255.255.255.255 ربط داده می شود! این آدرس را آدرس انتشاری یا Broadcast Address می نامیم. لذا همیشه به خاطر داشته باشید تا error checking را بصورت صحیح انجام دهید.

در حقیقت، یک واسط شفاف تر و واضح تر وجود دارد که شما می توانید بجای استفاده از inet_addr() از آن استفاده کنید: این واسط inet_aton() نام دارد ("aton" به معنی "Ascii TO Network" می باشد):

```
#include <ltsys/socket.h>
#include <ltnetinet/in.h>
#include <ltarpa/inet.h>

int inet_aton(const char *cp, struct in_addr *inp);
```

و در زیر یک کاربرد ساده در زمان بسته بندی (pack) یک strudt sockaddr_in را می بینید (هنگامی که به bind() و connect() رسیدیم، این مثال حس بیشتری را در شما برخواهد انگیخت!!).

```
struct sockaddr_in my_addr;

my_addr.sin_family = AF_INET;         // host byte order
my_addr.sin_port = htons(MYPORT);     // short, network byte order
inet_aton("10.12.110.57", &(my_addr.sin_addr));
memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct
```

واسط inet_aton()، برخلاف تقریبا تمام توابع مرتبط دیگر با ساکت، در صورت موفقیت، مقادیر غیر-صفر را برگشته داده و در صورت رخداد اشتباه در اجرا، مقدار صفر را بر می گرداند و آدرس در inp منتقل می شود. متاسفانه، پیاده سازی inet_aton() در تمام پلاتفرم ها اینگونه (یکسان) نیست و اگرچه استفاده از آن مقدم است، اما تابع معمول و قدیمی تر inet_addr() در مقاله استفاده می گردد.

به هر حال، شما اکنون می توانید آدرس های رشته ایIP [19] را به نمایش های باینریشان تبدیل کنید. آیا راه دیگری هم وجود دارد؟ اگر شما یک struct in_addr داشته باشید و بخواهید آن را به صورت نمادگذاری عددی-نقطه ای چاپ کنید، چطور؟ در این مورد، شما از تابع inet_ntoa() ("ntoa" بمعنی "Network TO Ascii" می باشد) به شکل زیر استفاده خواهید کرد:

```
printf("%s", inet_ntoa(ina.sin_addr));
```

این دستور آدرس IP را چاپ می کند. به خاطر داشته باشید که inet_ntoa() یک struct in_addr را بعنوان یک آرگومان می پذیرد، نه بیشتر. همچنین باید به خاطر سپرد که این تابع، یک اشاره گر به یک char (کاراکتر) را برمی گرداند. این اشاره گر به یک آرایه ایستای کاراکتری موجود درون inet_ntoa() اشاره می کند، بطوریکه هر بار که شما inet_ntoa() را فراخوانی می کنید، آخرین آدرس IP را که درخواست کرده بوده اید، جاینویسی می کند. برای مثال، قطعه برنامه زیر

```
char *a1, *a2;
.
.
a1 = inet_ntoa(ina1.sin_addr);  // this is 192.168.4.14
a2 = inet_ntoa(ina2.sin_addr);  // this is 10.12.110.57
```

[19] string IP Address

```
printf("address 1: %s\n",a1);
      printf("address 2: %s\n",a2);
```

خروجی زیر را تولید خواهد کرد:

```
address 1: 10.12.110.57
address 2: 10.12.110.57
```

اگر احتیاج به ذخیره آدرس دارید، با strcpy() آنرا در آرایه کاراکتری خود کپی کنید. تا همین جا برای این موضوع کافی است. بعدا، یاد خواهید گرفت که یک رشته مانند "whitehouse.gov" را به آدرس IP متناظرش تبدیل کنید (قسمت DNS را ببینید).

۴. فراخوانی های سیستمی یا Bust

در این قسمت ما فراخوانی هایی سیستمی را که به ما اجازه دستیابی به عاملیت شبکه ای از یک باکس یونیکس را می دهند بررسی می کنیم. هنگام فراخوانی یکی از این توابع، هسته بصورت اتوماتیک تمام کارها را برای شما انجام می دهد.

جایی که بسیاری از افراد در اینجا مشکل دارند، این است که به چه ترتیبی باید این چیزها را فراخوانی کرد. بدون این اطلاعات مهم، حتی بزرگترین اشکال زداهای برنامه نویسی شبکه برای شما سودی نخواهند داشت. لذا سعی کردیم تا فراخوانی های سیستم را دقیقا (تقریبا) به همان ترتیبی که شما در برنامه هایتان احتیاج دارید، ذکر کنیم.

۴/۱ socket() – گرفتن توصیفگر فایل!

بالاخره به تابع socket() رسیدیم. در این قسمت سعی می کنیم درباره فراخوانی سیستمی socket() گفت و گو کنیم. در زیر مثالی را می بینید:

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

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

اما واقعا این آرگومان ها چه چیزهایی هستند؟ ابتدا، domain باید "AF_INET" تنظیم شود درست مانند struct sockaddr_in (که در بالا ذکر شد). سپس نوع آگومان به هسته می گوید که این ساکت از چه نوعی است: SOCK_STREAM یا SOCK_DGRAM را می توان مورد استفاده قرار داد. سرانجام، تنها لازم است پروتکل را به "0" تنظیم کنیم تا socket() پروتکل صحیح را بسته به نوع تعیین شده انتخاب کند (توجه: دامین های بسیاری از آنهایی که من لیست کرده ام وجود دارند. انواع زیادتری از آنچه من لیست کرده ام وجود دارند. همچنین، باید ذکر کرد که یک راه "بهتر" برای گرفتن پروتکل وجود دارد. قسمت getprotobyname() را ببینید).

socket() بسادگی به شما یک توصیفگر ساکت را بر می گرداند که شما می توانید از آن در فراخوانی های سیستمی بعدی استفاده کنید و در صورتی که با خطا مواجه شود، -1 را برخواهد گرداند. متغیر عمومی errno به ارزش و مقدار خطا تنظیم می شود (مورد perror() را در زیر ببینید).

در بعضی مقاله ها، شما اشاره ای از "PF_INET" خواهید دید که بصورت طبیعی بندرت دیده می شود، اما برای شفاف شدن موضوع اندکی آنرا در اینجا توضیح می دهیم. چند وقت پیش، گمان می رفت که ممکن است یک خانواده آدرس[۲۰] (که "AF"

[20] a address family

در "AF_INET" مخفف آن است) چندین پروتکل را که توسط خانواده پروتکلیشان[21] ارجاع داده شده اند (که "PF" در "PF_INET" به آن اشاره دارد) پشتیبانی کند. در صورتی که چنین نبود! پس کار صحیح این است که از AF_INET در struct sockaddr_in خود و از PF_INET در فراخوانی خود به socket() استفاده کنید. اما اگر بخواهیم واقع گرایانه صحبت کنیم، باید گفت که شما می توانید از AF_INET در هرجایی استفاده کنید و به دلیل اینکه W. Richard Stevens در کتاب خود از آن استفاده کرده است، لذا ما هم در این مقاله از آن استفاده می کنیم. اما این ساکت چه مزیتی دارد؟ جواب این است که، این ساکت به خودی خود هیچ منفعتی ندارد و شما برای کارکرد آن باید فراخوانی های سیستمی بیشتری را درگیر کار کنید.

۴/۲ bind() – روی چه پورتی هستم؟

هنگامی که یک ساکت دارید، ممکن است مجبور به وابسته کردن (ربط دادن) آن ساکت با یک پورت روی ماشین محلی خود شوید (این کار معمولا هنگامی که شما با استفاده از listen() برای ارتباطات ورودی روی یک پورت خاص فالگوش ایستاده اید – MUD ها هنگامی که به شما می گویند به w.x.y.z با پورت 6969 تلنت کنید، این کار را انجام می دهند). شماره پورت توسط هسته جهت منطبق کردن (match کردن) یک بسته ورودی با یک توصیفگر ساکت از یک پروسه مشخص استفاده می شود. اگر شما می خواهید که فقط یک connect() داشته باشید، این کار غیرضروری است. به هر حال، برای افزایش اطلاعات هم که شده آنرا بخوانید. در زیر خلاصه ای برای فراخوانی سیستمی bind() می بینید:

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

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

socketfd توصیفگر فایل ساکتی[22] است که توسط socket() برگردانده شده است. my_addr یک اشاره گر به یک struct sockaddr است که حاوی اطلاعات درباره آدرس شما، نام، پورت و آدرس IP شما می باشد. addrlen را می توان بصورت sizeof(struct sockaddr) تنظیم کرد. اکنون به مثالی در این باب توجه کنید:

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

#define MYPORT 3490

main()
{
   int sockfd;
   struct sockaddr_in my_addr;

   sockfd = socket(AF_INET, SOCK_STREAM, 0); // do some error checking!

   my_addr.sin_family = AF_INET;         // host byte order
   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', 8); // zero the rest of the struct

   // don't forget your error checking for bind():
```

[21] protocol family
[22] socket file descriptor

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

تعدادی مسئله وجود دارند که باید آنها را بخاطر بسپارید:

myaddr.sin_port به صورت ترتیب بایت شبکه ای می باشد، پس بصورت my_addr.sin_addr.s_addr می باشد. نکته دیگر این است که هدر فایل ها ممکن است از یک سیستم به سیستم دیگر تفاوت داشته باشند. آخرین نکته هم این که، در قسمت bind()، باید ذکر کنم که می توان بعضی از فرآیندهای دریافت آدرس IP و/یا پورت خود را اتوماتیک کرد:

```
my_addr.sin_port = 0; // choose an unused port at random
my_addr.sin_addr.s_addr = INADDR_ANY;  // use my IP address
```

با تنظیم my_addr.sin_port به صفر، شما به bind() می گویید که برای شما پورت انتخاب کند. همچنین، با تنظیم my_addr.sin_addr.s_addr به INADDR_ANY، شما به آن می گویید که بصورت اتوماتیک با آدرس IP ماشینی که پروسه روی آن در حال اجراست، تعویض شود.

اگر باریک بینی خود را بکار انداخته باشید، خواهید فهمید که من INADDR_ANY را بصورت ترتیب بایت شبکه ای قرار نداده ام! بهرحال، من اطلاعاتی درونی داریم: INADDR_ANY در حقیقت صفر می باشد! حتی اگر شما بایت ها را بازآرایی کنید، یک مقدار صفر، ارزش صفر خود را حفظ خواهد کرد. به هر حال، اشخاص درک خواهند کرد که باید یک بعد موازی در جایی که INADDR_ANY آنجا وجود دارد باشد که در اینجا مقدار ۱۲ است و کد ما کار کرد نخواهد داشت:

```
my_addr.sin_port = htons(0); // choose an unused port at random
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);  // use my IP address
```

من تنها خواستم اشاره کنم که بسیاری از کدهایی که شما با آنها سروکار دارید، بدون مشکل با INADDR_ANY و از طریق htonl() اجرا نخواهند شد. Bind() همچنین در صورت رخداد خطا -1 را برمی گرداند و errno را به مقدار و ارزش خطا تنظیم می کند. نکته دیگری که در هنگام فراخوانی bind() باید بدان توجه داشت این است که: باید مراقب به شماره پورت ها نیز بود. تمام پورت های قبل از 1024، رزرو شده اند (مگر اینکه شما در سیستم superuser باشید)! شما می توانید هر شماره پورتی بعد از آن را تا ۶۵۵۳۵ داشته باشید (که این پورتها توسط برنامه های سیستمی یک سیستم عامل استفاده نمی گردند).

بعضی مواقع، ممکن است ملاحضه کرده باشید که در اجرای مجدد یک سرور، bind() در کار خود ناموفق بوده و پیامی به صورت "Address already in use." به شما داده خواهد شد. این وضعیت چه معنی دارد؟

در هسته، مقداری از ساکت اتصالی هنوز در حال کارکرد و وقت گذرانی است و در نتیجه پورت را در دست خود دارد. شما می توانید مقداری صبر کنید تا این حالت برطرف شده (برای مثال یک دقیقه یا ...)، یا کدی را به برنامه خود اضافه کنید تا به آن اجازه مجدد از یک پورت را بدهد، بمانند زیر:

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

   // lose the pesky "Address already in use" error message
   if (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1) {
      perror("setsockopt");
      exit(1);
   }
```

* آخرین نکته درباره bind(): مواقعی وجود دارد که شما مطلقا مجبور به فراخوانی آن نیستید. اگر شما با استفاده از connect() به یک ماشین ریموت وصل شده و به اینکه پورت محلی شما چیست توجهی ندشته باشید (که در مورد برنامه تلنت شما فقط به پورت

ریموت توجه دارید)، می توانید بسادگی connect() را فراخوانی کنید. این تابع چک می کند تا ببینید آیا ساکت غیر محدود[23] می باشد یا خیر و سپس در صورت لزوم، آنرا با استفاده از bind() به یک پورت محلی می چسباند (bind می کند).

۴/۳ تابع connect()

بیایید وانمود کنیم که ما یک برنامه Telnet هستیم. کاربر شما به شما دستور می دهد (درست مانند فیلم TRON) تا یک توصیفگر فایل ساکت[24] را بگیرید. شما موافقت کرده و socket() را فراخوانی می کنید. سپس، کاربر به شما می گوید که به "10.12.110.57" روی پورت "23" وصل شوید (پورت استاندارد تلنت). اکنون باید چکاری انجام دهید؟

اکنون شما باید connect() را بررسی کنید – چطور می توان به یک میزبان ریموت وصل شد. فراخوانی connect() به صورت زیر می باشد:

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

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

sockfd توصیفگر فایل ساکت موجود در مجاورت ما می باشد، که هنگامی که بوسیله فراخوانی socket() برگشت داده می شود، serv_addr یک struct sockaddr بوده که حاوی آدرس IP و پورت مقصد می باشد و addrlen را می توان به صورت sizeof(struct sockaddr) تنظیم کرد. اکنون یک مثال را با هم بررسی می کنیم:

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

#define DEST_IP   "10.12.110.57"
#define DEST_PORT 23

main()
{
   int sockfd;
   struct sockaddr_in dest_addr;   // will hold the destination addr
sockfd = socket(AF_INET, SOCK_STREAM, 0); // do some error checking!

   dest_addr.sin_family = AF_INET;          // host byte order
   dest_addr.sin_port = htons(DEST_PORT);   // short, network byte order
   dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
   memset(&(dest_addr.sin_zero), '\0', 8);  // zero the rest of the struct

   // don't forget to error check the connect()!
   connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
   …
```

مجددا، از چک کردن مقدار برگشتی از connect() اطمینان حاصل کنید – در صورت بروز خطا 1- را برگشت داده و متغیر را errno تنظیم می کند. همچنین، ملاحظه می کنید که ما bind() را فراخوانی نکردیم. اساس، ما هیچ توجهی به شماره پورت محلی خودمان نداریم، ما تنها به پورت ریموت توجه خواهیم داشت. هسته یک پورت محلی را برای محل انتخاب خواهد کرد و سایتی که ما به آن متصل می شویم، این اطلاعات را به صورت اتوماتیک از ما خواهد گرفت. هیچ نگرانی وجود ندارد!

[23] unbound
[24] Socket file descriptor

۴/۴ listen() – چه کسی مرا صدا می زند؟

اگر شما نخواهید به یک هاست ریموت متصل شوید، چه می شود!؟ شاید، بنابه دلایلی، بخواهید برای ارتباطات ورودی[۲۵] صبر کرده و به طریقی با آنها کار کنید. این فرآیند دو مرحله دارد: نخست، باید فالگوش ایستاد (با استفاده از listen()) و سپس ارتباط را پذیرفت (با استفاده از accept()). فراخوانی استماع[۲۶] بسیار راحت است، اما به توضیح اندکی نیاز دارد:

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

sockfd یک توصیف فایل ساکت معمول از فراخوانی سیستمی socket() می باشد. backlog تعداد ارتباطاتی است که در صف های ورودی ارتباطات به آنها اجازه برقرار شدن را می دهیم. شاید بپرسید این مورد به چه معنی است؟ ارتباطات ورودی تا زمانی که شما آنها را accept() کنید، در این صف منتظر می مانند و تعداد ارتباطات تعیین شده بستگی به محدودیت ایجاد صف ها دارد. بسیاری از سیستم ها این تعداد را در حوالی ۲۰ کاهش می دهند؛ ممکن است شما بخواهید این عدد را بین ۵ تا ۱۰ قرار دهید. مجددا و طبق معمول، باید گفت که listen() در صورت بروز خطا 1- را برگردانده و errno را تنظیم می کند.

همان طور که ممکن است تا به حال متوجه شده باشید، ما قبل از فراخوانی listen() باید bind() را فراخوانی کنیم و در غیراین صورت هسته ما را به فالگوش ایستادن روی یک *پورت تصادفی* سوق می دهد! بنابراین، اگر شما می خواهید برای ارتباطات ورودی فالگوش بایستید، ترتیبی از فراخوانی های سیستمی که باید انجام شود به صورت زیر خواهند بود:

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

۴/۵ accept() – "از گوش دادن روی پورت ۳۴۹۰ تشکر می کنم."

اکنون پس از فالگوش ایستادن، آماده دریافت ارتباط (با accept() هستیم). فرآیند رخدادی به صورت زیر خواهد بود: شخصی از راه دور سعی بر connect() شدن به ماشین شما روی یک پورت مشخص که شما روی آن listen() کرده اید دارد. ارتباط او به صف درآمده و برای accept() شدن منتظر خواهد ماند. شما accept() را فراخوانی کرده و اجازه دریافت ارتباط را می دهید. سپس به شما یک توصیفگر فایل ساکت جدید را برگشت داده تا برای این ارتباط واحد استفاده کنید! درست است! شما دو توصیفگر فایل ساکت به ارزش یکسان دارید! توصیفگر اصلی هنوز برای پورت شما فالگوش ایستاده است و توصیفگری که جدیدا ساخته شده است برای send() و recv() آماده به کار می شود. فراخوانی به صورت زیر می باشد:

```
#include <sys/socket.h>
int accept(int sockfd, void *addr, int *addrlen);
```

sockfd، توصیفگر فالگوشی ساکت (برای listen()) می باشد. addr معمولا یک اشاره گر به یک struct sockaddr_in محلی خواهد بود. اینجا جایی است که اطلاعات مربوط به ارتباط ورودی به آنجا خواهد رفت (و با آن شما می توانید تعیین کنید که کدام هاست و از چه پورتی، شما را فراخوانی می کند). Addrlen یک متغیر محلی صحیح است که باید قبل از انتقال آدرسش به accept، به صورت sizeof(struct sockaddr_in) تنظیم شود. Accept بیشتر از آن تعداد بایت های زیاد در addr قرار نمیدهد.

---

[25] incoming connections

[26] فالگوش ایستادن – گوش دادن – Listen Call

اگر تعداد کمتری قرار دهد، مقدار addrlen را جهت انعکاس (reflect) آن، تغییر خواهد داد. چه حدسی می زنید؟ accept() در صورت بروز خطا، -1 را برگردانده و errno را تنظیم می کند. همانند قبل، یک قطعه کد ساده را برای مطالعه قرار می دهیم:

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

#define MYPORT 3490    // the port users will be connecting to

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

main()
{
   int sockfd, new_fd;  // listen on sock_fd, new connection on new_fd
   struct sockaddr_in my_addr;    // my address information
   struct sockaddr_in their_addr; // connector's address information
   int sin_size;

   sockfd = socket(AF_INET, SOCK_STREAM, 0); // do some error checking!

   my_addr.sin_family = AF_INET;         // host byte order
   my_addr.sin_port = htons(MYPORT);     // short, network byte order
   my_addr.sin_addr.s_addr = INADDR_ANY; // auto-fill with my IP
   memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

   // don't forget your error checking for these calls:
   bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));

   listen(sockfd, BACKLOG);

   sin_size = sizeof(struct sockaddr_in);
   new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
   .
   .
   .
```

مجددا به خاطر بسپارید که ما از توصیفگر ساکت new_fd برای تمام فراخوانی های send() و recv() استفاده می کنیم. اگر مایل به دریافت تنها یک ارتباط هستید، باید sockfd در حال شنود را close() کرده تا از ارتباطات ورودی روی آن پورت جلوگیری به عمل آورید.

۴/۶ send() و recv() – با من صحبت کن!

دو تابع برای ارتباط روی ساکت های جریانی یا ساکت های متصل شده ی دیتاگرامی وجود دارد. اگر می خواهید از ساکت های دیتاگرامی غیر اتصالی و منظم[۲۷] استفاده کنید، باید قسمت sendto() و recvfrom() را در زیر ببینید.

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

[27] Regular unconnected datagram sockets

sockfd توصیفگر ساکتی است که شما می خواهید اطلاعات را به آن بفرستید (چه توصیفگری باشد که توسط socket() برگشت داده شده و چه توصیفگری باشد که با accept() دریافت کرده اید). msg یک اشاره گر به داده هایی است که می خواهید بفرستید و len، طول و اندازه آن داده به بایت می باشد. تنها فلگ ها را به ۰ تنظیم کنید. یک مثال نمونه را در زیر می بینید:

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

send()، تعداد بایت هایی که عملا ارسال شده اند را بر می گرداند – ممکن است کمتر از تعدادی باشد که شما قصد فرستادن آنها را داشته اید! بعضی مواقع شما به این تابع می گویید که یک قطعه اطلاعاتی بزرگ را بفرستد و امکان حمل آن قطعه اطلاعاتی نمی باشد. در این صورت این تابع تا جایی که امکان دارد داده ها را ارسال کرده و برای ارسال باقیمانده داده ها به شما اعتماد می کند. به خاطر بسپارید که اگر مقدار برگشتی توسط send() با مقدار موجود در len مطابقت نداشته باشد، در این صورت برعهده شماست تا باقیمانده رشته را ارسال کنید. خبر مهم این است که: اگر بسته کوچک باشد (کمتر از 1 KB)، در این صورت این تابع احتمالا برای ارسال یکبارگی تمام داده ها اقدام خواهد کرد. مجددا باید گفت که در صورت بروز خطا، مقدار -1 برگشت داده شده و errno به شماره خطا تنظیم خواهد گشت. فراخوانی recv() زیر را ببینید:

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

sockfd توصیفگر فایلی است که خواندن از آن انجام می شود، buf، بافری است که اطلاعات باید از آن خوانده شوند، len طول ماکزیمم برای بافر بوده و فلگ ها را می توان مجددا به ۰ تنظیم کرد. recv() تعداد بایت هایی که عملا از بافر خوانده شده است را برمی گرداند و در صورت بروز خطا مقدار -1 را برگشت خواهد داد و errno را تنظیم می کند.

اما یک نکته وجود دارد! recv() می تواند ۰ را برگشت دهد و این تنها می تواند به یک معنی باشد: طرف ریموت ارتباط را بر روی شما بسته است! مقدار برگشتی 0 از recv() راهی است تا شما بفهمید این مسئله اتفاق افتاده است. به هر حال، در این نقطه از مقاله، شما می توانید داده ها را روی ساکت های جریانی ارسال و دریافت بدارید! اکنون شما یک برنامه نویس شبکه برای یونیکس[۲۸] شده اید!!

۴/۷ sendto() و recvfrom() – به سبک DGRAM با من صحبت کن!

ساکت های دیتاگرامی به یک میزبان ریموت متصل نیستند، حدس بزنید چه اطلاعاتی را قبل از فرستادن یک بسته باید ارسال بداریم؟ بلی، آدرس مقصد! در زیر یک ساختار نمونه را می بینیم:

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

[28] Unix Network Programmer :D ;)

همان طور که می بینید، این فراخوانی اساسا همان فراخوانی به send() بعلاوه ی دو قطعه اطلاعاتی دیگر می باشد. To یک اشاره گر به یکstruct sockaddr می باشد که حاوی آدرس IP و پورت مقصد می باشد. Tolen را می توان به sizeof(struct sockaddr) تنظیم کرد.

درست مانند send()، sendto() نیز تعداد بایت هایی که در عمل فرستاده شده اند را بر می گرداند (که ممکن است کمتر از تعداد بایت هایی باشد که شما دستور به ارسال آنها داده اید) و در صورت بروز خطا نیز مقدار 1- برگشت داده خواهد شد. recv() و recvfrom() تقریبا با هم مشابه می باشند. ساختاری خلاصه از recvfrom() به صورت زیر است:

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

این ساختار نیز، درست مانند recv() با چندین فیلد بیشتر می باشد. from یک اشاره گر به یک struct sockaddr محلی می باشد که با آدرس IP و پورت ماشین اصلی پر خواهد گشت. fromlen اشاره گری به یک int محلی می باشد که باید با sizeof(struct sockaddr) آنرا مقدار اولیه داد. هنگامی که توابع از اجرایشان برمی گردند، fromlen حاوی طول واقعی آدرسی است که در آن ذخیره گشته است. recvfrom() تعداد بایت های دریافت شده را بر می گرداند و در صورت بروز خطا مقدار 1- برگشت داده خواهد شد (و متعاقبا تنظیم errno).

به خاطر داشته باشید که اگر شما یک ساکت دیتاگرامی را متصل کنید (با استفاده از connect()، آنگاه می توانید از send() و recv() برای تمام انتقالات خود استفاده کنید. ساکت خودش هنوز یک ساکت دیتاگرامی است و بسته ها هنوز از UDP استفاده می کنند، اما واسط ساکت[۲۹] به صورت اتوماتیک مقصد و اطلاعات منبع را برای شما اضافه می کند.

۴/۸ توابع close() و shutdown()

پس از مدتی انتقال با send() و recv() ممکن است بخواهید ارتباط را روی توصیفگر فایل خود ببندید که بسیار ساده است. تنها می توانید تنها از توصیفگر فایل یونیکسی منظم[۳۰] تابعِ close() استفاده کنید:

```
close(sockfd);
```

این تابع از هر گونه خواندن و نوشتن اضافی را روی ساکت جلوگیری می کند. هر شخص که سعی بر نوشتن یا خواندن از ساکت روی گره ریموت کند، یک خطا دریافت خواهد کرد. اگر می خواهید اندکی کنترل بیشتری روی بستن ساکت ها داشته باشید، می توانید از تابع shutdown() استفاده کنید. این تابع به شما اجازه می دهد که ارتباط را در یک جهت مشخص یا در هر دو جهت (درست مثل کاری که close() انجام می دهد) قطع کنید:

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

sockfd توصیفگر فایل ساکتی است که شما می خواهید آنرا خاموش (shutdown) کنید و how یکی از موارد زیر خواهد بود:

0 -- Further receives are disallowed

1 -- Further sends are disallowed

2 -- Further sends and receives are disallowed (like close())

shutdown() در صورتی که کار خود را با موفقیت انجام دهد مقدار 0 را برمی گرداند و در صورت بروز خطا 1- را بر می گرداند (همراه با تنظیم errno). اگر شما قصد دارید که از shutdown() روی ساکت های دیتاگرامی متصل نشده[۳۱] استفاده کنید،

---

[29] socket interface
[30] regular Unix file descriptor
[31] unconnected datagram sockets

در این صورت ساکت را برای فراخوانی های بعدی send() و recv() غیرقابل دسترس می سازد (به خاطر داشته باشید که پس از قطع ارتباط، هنوز می توانید از این دو تابع استفاده کنید که قبل از آن باید ساکت دیتاگرامی خود را connect() کنید).

بسیار مهم است که به خاطر داشته باشید که shutdown() در عمل توصیفگر فایل را نمی بندد – تنها قابلیت استفاده آنرا تغییر می دهد. برای آزاد کردن یک توصیفگر ساکت، شما نیاز به استفاده از close() دارید (در مقاله های بعدی راجع به close() بحث خواهیم کرد).

۴/۹ getpeername() – شما چه کسی هستید؟

این تابع آنقدر ساده می باشد که معمولا در مقاله ها به آن هیچ توجه و بخش مجزایی تخصیص داده نمی شود. تابع getpeername() به شما خواهد گفت که چه کسی در انتهای دیگر یک ساکت جریانی متصل (شده)[۳۲] می باشد. در زیر ساختار این تابع را می بینید:

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

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

sockfd، توصیفگر ساکت جریانی اتصال یافته می باشد، addr، یک اشاره گر به یک struct sockaddr (یا یک struct sockaddr_in) می باشد که اطلاعاتی را مربوط به طرف دیگر ارتباط نگهداری می کند و addrlen، یک اشاره گر به یک int می باشد که باید با sizeof(struct sockaddr) آنرا مقدار اولیه داد. تابع در صورت بروز خطا مقدار 1- را برگشت می دهد (و errno را تنظیم خواهد کرد). هنگامی که شما آدرس آنها را بدست می آورید، می توانید از inet_ntoa() یا gethostbyaddr() برای چاپ یا دریافت اطلاعات بیشتر استفاده کنید. اما، خیر! شما همیشه نمی توانید login name آنها را بدست آورید (در صورتی که کامپیوتر موجود در طرف دیگر در حال اجرای یک دیامون ident باشد، این کار امکان پذیر خواهد بود. به هر حال، این مورد خارج از اهداف این مقاله است. RFC-1413 را برای اطلاعات بیشتر چک کنید).

۴/۱۰ gethostname() – من که هستم!؟

تابع ساده تر از getpeername()، تابع gethostname() می باشد. این تابع نام کامپیوتری که برنامه شما روی آن در حال اجراست بر می گرداند. این نام را می توان توسط gethostbyname() مورد استفاده قرار داد و در نتیجه آدرس IP ماشین محلی را تعیین کرد. شاید فکر کنید که چه چیزهای دیگری می تواند به جالب بودن کار اضافه کنند! من هم در همین فکر هستم، اما این موارد به Socket Programming مربوط نخواهند شد. به هر قسمت زیر را ببینید:

```
#include <unistd.h>

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

آرگومان ساده هستند: hostname، اشاره گری به یک آرایه کاراکتری است و بمحض بازگشت تابع، حاوی نام میزبان خواهد شد و size، طول آرایه hostname به بایت می باشد. تابع در صورت اجرا و تکمیل موفقیت آمیز مقدار 0 را برگشت داده و در صورت بروز خطا مقدار 1- را برگشت خواهد داد (و errno را تنظیم خواهد کرد).

۴/۱۱ DNS – تو بگو "whitehouse.gov"، من میگم "198.137.240.92"!

[32] connected datagram socket

واژه DNS مخفف عبارت "Domain Name Service" می باشد. در مخلص کلام، یک آدرس خوانا و قابل حفظ برای انسان و مربوط به یک آدرس IP یک سرور خاص که به شما آدرس IP را می دهد (بنابراین می توانید از آن در کنار bind()، connect()، sendto() یا هر چیزی که احتیاجش را دارید استفاده کنید). به این صورت، هنگامی که شخص عبارت زیر را وارد می کند:

```
$ telnet whitehouse.gov
```

تلنت می تواند بفهمد که نیاز به connect() شدن به آدرس "198.137.240.92" است. اما چگونه این کار انجام می شود؟ شما از تابع gethostbyname() استفاده خواهید کرد:

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

همان طور که می بینید، این عبارات، یک اشاره گر به یک struct hostent را بر می گرداند. طرح این ساختمان به صورت زیر می باشد:

```
struct hostent {
    char    *h_name;
    char    **h_aliases;
    int     h_addrtype;
    int     h_length;
    char    **h_addr_list;
};
#define h_addr h_addr_list[0]
```

در زیر توضیح فیلدهای موجود در struct hostent را می بینید:

h_name -- Official name of the host.

h_aliases -- A NULL-terminated array of alternate names for the host.

h_addrtype -- The type of address being returned; usually AF_INET.

h_length -- The length of the address in bytes.

h_addr_list -- A zero-terminated array of network addresses for the host. Host addresses are in Network Byte Order.

h_addr -- The first address in h_addr_list.

gethostbyname() returns a pointer to the filled struct hostent, or NULL on error. (But errno is not set--h_errno is set instead.)

اما چگونه این ساختمان مورد استفاده قرار می گیرد؟ این تابع را بسیار راحت از چیزی که به نظر می آید می توانیم بکار ببریم. در زیر یک برنامه نمونه را می بینید:

```
/*
** getip.c -- a hostname lookup demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
```

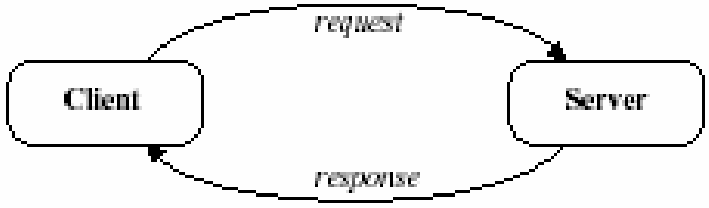
```
#include <netdb.h>
      #include <sys/types.h>
      #include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
   struct hostent *h;

   if (argc != 2) {  // error check the command line
      fprintf(stderr,"usage: getip address\n");
      exit(1);
   }

   if ((h=gethostbyname(argv[1])) == NULL) {  // get the host info
      herror("gethostbyname");
      exit(1);
   }

   printf("Host name  : %s\n", h->h_name);
   printf("IP Address : %s\n", inet_ntoa(*((struct in_addr *)h->h_addr)));

  return 0;
}
```

با gethostbyname()، شما نمی توانید از perror() برای چاپ کردن پیام خطا استفاده کنید (چرا که errno استفاده نشده است). در عوض تابع **herror()** را فراخوانی کنید! این تابع تقریبا سرراست و بدون دردسر می باشد. شما تنها رشته ای را که حاوی نـام ماشـین مـی باشـد ("whitehouse.gov") بـه gethostbyname() انتقـال مـی دهیـد و سـپس اطلاعـات را از struct hostent برگشت داده شده دریافت می کنید.

تنها نقطه کور در اینجا می تواند چاپ کردن آدرس IP باشد، در قطعه کـد بـالا، h->h_addr، آرایـه کـاراکتری مـی باشـد (یعنی یک char* می باشد)، اما inet_ntoa() می خواهد یک struct in_addr را به آن انتقال دهیم. بنابراین ما h->h_addr را در قالب یک struct in_addr* در آورده و سپس آنرا جهت دریافت داده ها (اطلاعات) مرجع زدایی می کنیم (dereference).

۵. پشت صحنه ارتباطات کلاینت–سرور

تقریبا همه تعامل های روی شبکه، از گفت و گوی پردازش های طرف کلاینـت بـا پـردازش هـای طـرف سـرور و بعکـس نشأت می گیرند. برای مثال، برنامه Telnet را در نظر بگیرید. هنگامی که با تلنت (کلاینت) به یک میزبان ریمـوت روی پـورت ۲۳ متصل می شوید، یک برنامه روی هاست (با نام telnetd، سرور) شروع به فعالیت می کند. این برنامه در طـرف سـرور بـا ارتباطـات ورودی تلنت سروکار دارد و به عنوان اولین عکس العمل برای شما یک login prompt یا ... را ظاهر می کند.



تصویر ۲ - فعل و انفعال بین کلاینت و سرور

تبادل اطلاعات بین کلاینت و سرور در تصویر ۲ خلاصه شده است. به خاطر داشته باشید که زوج کلاینت-سرور با SOCK_STREAM، SOCK_DGRAM یا یک چیز دیگر (و مادامی که آنها با همان چیز با هم گفت و گو دارند) با یکدیگر صحبت می کنند. چند مثال مناسب از زوج کلاینت-سرور، telnet/telnetd، ftp/ftpd یا bootp/bootpd و ... می باشند. هر هنگامی که شما از ftp استفاده می کنید، یک برنامه ریموت در کامپیوتر سرور به نام ftpd وجود دارد که به شما خدمات دهی می کند. اغلب، یک سرور (در اینجا به معنی خدمتگذار یا ... تجسم کنید) روی یک ماشین وجود دارد و آن سرور با استفاده از fork() با چندین کلاینت سروکار خواهد داشت. روتین و روال اساسی به صورت زیر می باشد:

سرور برای یک ارتباط صبر و سپس آنرا accept() کرده و یک پروسه فرزند را جهت کارکردن با آن، انشعاب می دهد (یا عمل Fork را با fork() انجام می دهد). در قسمت بعدی، سرور ما عملیات ذکرشده را انجام می دهد.

۵/۱ یک سرور جریانی نمونه

تمامی کاری که سرور ما در این قسمت انجام می دهد این است که رشته "Hello,World!\n" را روی یک ارتباط جریانی (stream) ارسال می کند. برای آزمایش برنامه شما می توانید آنرا در یک پنجره اجرا کرده و از یک پنجره دیگر با استفاده از دستور زیر به آن تلنت کنید:

```
$ telnet remotehostname 3490
```

که در اینجا remotehostname نامی ماشینی است که شما برنامه سرور را روی آن اجرا کرده اید. در زیر کد سرور (برنامه خدمتگذار ما در این مثال) را آورده ایم:

```
/*
** 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 <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

#define MYPORT 3490    // the port users will be connecting to

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

void sigchld_handler(int s)
{
   while(wait(NULL) > 0);
}

int main(void)
{
   int sockfd, new_fd;  // listen on sock_fd, new connection on new_fd
```

```
struct sockaddr_in my_addr;    // my address information
        struct sockaddr_in their_addr; // connector's address information
        int sin_size;
struct sigaction sa;
int yes=1;

if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    perror("socket");
    exit(1);
}

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

my_addr.sin_family = AF_INET;         // host byte order
my_addr.sin_port = htons(MYPORT);     // short, network byte order
my_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr))
                                              == -1) {
    perror("bind");
    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);
}

while(1) {  // main accept() loop
    sin_size = sizeof(struct sockaddr_in);
    if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr,
                                        &sin_size)) == -1) {
        perror("accept");
        continue;
    }
    printf("server: got connection from %s\n",
                                inet_ntoa(their_addr.sin_addr));
    if (!fork()) { // this is the child process
        close(sockfd); // child doesn't need the listener
        if (send(new_fd, "Hello, world!\n", 14, 0) == -1)
            perror("send");
        close(new_fd);
        exit(0);
    }
```

```
            close(new_fd);  // parent doesn't need this
                }

        return 0;
    }
```

ما کد را برای دلایل واضح، در یک تابع main() بزرگ آورده ایم . لذا شما می توانید طبق نیاز و علاقه خود آنـرا بـه توابـع کوچکتر تقسیم کنید. تابع به تابع sigaction() نیز قبل از این برخورد نداشته ایم. کدهای موجـود در آن بـرای جمـع آوری (reap) پروسه های زامبی که به صورت پروسه های خروجی فرزندِ fork() شده نمودار شده اند. اگر شما زامبی های زیـادی اعمـال کنیـد و آنها را جمع نکنید، مسائلی جانبی رخ خواهد داد. شما می توانید داده ها را از این سرور با استفاده از کلاینتِ لیست شـده در قـسمت بعد دریافت کنید.

۵/۲ یک کلاینت جریانی نمونه

این برنامه راحت تر از برنامه سرور می باشد. تمام کاری که برنامه کلاینت انجام می دهد، متصل شدن به هاستی اسـت کـه شما در command line تعیین کردید (روی پورت 3490). کلاینت پس از اتصال رشته ای کـه سـرور ارسـال مـی دارد را دریافـت خواهد کرد. کد سورس برای کلاینت:

```
/*
 ** 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>

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

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

int main(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct hostent *he;
    struct sockaddr_in their_addr; // connector's address information

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

    if ((he=gethostbyname(argv[1])) == NULL) {  // get the host info
        perror("gethostbyname");
        exit(1);
```

```
    }

        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
      perror("socket");
      exit(1);
    }

    their_addr.sin_family = AF_INET;    // host byte order
    their_addr.sin_port = htons(PORT);  // short, network byte order
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(&(their_addr.sin_zero), 8);  // zero the rest of the struct

    if (connect(sockfd, (struct sockaddr *)&their_addr,
                                sizeof(struct sockaddr)) == -1) {
      perror("connect");
      exit(1);
    }

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

    buf[numbytes] = '\0';

    printf("Received: %s",buf);

    close(sockfd);

    return 0;
  }
```

به خاطر داشته باشید که اگر قبل از اجرای کلاینت، سـرور را اجـرا نکـرده باشـید، تـابع connect() **پیـام** "Connection Refused" **را برگشت خواهد داد.**

۵/۳ ساکت های دیتاگرامی

در اینجا تنها چندین برنامه نمونه به نام های talker.c **و** listener.c **را عنوان می کنـیم.** listener **روی یـک ماشـین جهـت یک بسته ورودی روی پورت ۴۹۵۰ منتظر می شود.** talker **یک بسته را به آن پورت، روی آن ماشـین خـاص ارسـال مـی کنـد کـه حاوی چیزهایی است که کاربر در** command line **وارد می کند. در زیر سورس کد** listener.c **را می بینید:**

```
/*
** 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>
```

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

#define MAXBUFLEN 100

int main(void)
{
    int sockfd;
    struct sockaddr_in my_addr;    // my address information
    struct sockaddr_in their_addr; // connector's address information
    int addr_len, numbytes;
    char buf[MAXBUFLEN];

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    my_addr.sin_family = AF_INET;         // host byte order
    my_addr.sin_port = htons(MYPORT);     // short, network byte order
    my_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
    memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

    if (bind(sockfd, (struct sockaddr *)&my_addr,
                                sizeof(struct sockaddr)) == -1) {
        perror("bind");
        exit(1);
    }

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

    printf("got packet from %s\n",inet_ntoa(their_addr.sin_addr));
    printf("packet is %d bytes long\n",numbytes);
    buf[numbytes] = '\0';
    printf("packet contains \"%s\"\n",buf);

    close(sockfd);

    return 0;
}
```

به خاطر داشته باشید که در فراخوانیمان به socket() ما بالاخره از SOCK_DGRAM استفاده می کنیم. همچنین به خاطرداشته باشیدکه احتیاجی به listen() یا accept() نیست! این یکی از جنبه ها و نکات در استفاده از ساکت های دیتاگرامی غیر اتصالی می باشد! در زیر نیز، سورس کد talker.c را می بینید (بهصورت ترتیب بایت شبکه ای باید باشد):

```
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(&(their_addr.sin_zero), '\0', 8); // zero the rest of the struct

    if ((numbytes=sendto(sockfd, argv[2], strlen(argv[2]), 0,
        (struct sockaddr *)&their_addr, sizeof(struct sockaddr))) == -1) {
```

```
        perror("sendto");
            exit(1);
        }

    printf("sent %d bytes to %s\n", numbytes,
                                inet_ntoa(their_addr.sin_addr));

    close(sockfd);

    return 0;
}
```

اکنون، listener را روی یک ماشین اجرا کنید و سپس talker را روی یک ماشین دیگر اجرا کنید و ارتباط آنها را مشاهده کنید! مستثنی از چند نکته کوچک، در بسیاری از اوقات در قبل راجع به ساکت های دیتاگرامی اتصالی[۳۳] صحبت کرده ام. اکنون، زمان صحبت کردن درباره این مسئله می باشد، چرا که ما در قسمت دیتاگرام مقاله قرار داریم. بیایید فرض کنیم که talker، تابع connect() را فراخوانی کرده و آدرس listener را مشخص می کند. از آن نقطه به بعد، talker ممکن است تنها با آدرسی که توسط connect() تعیین گشته است، ارسال و دریافت اطلاعات داشته باشد. به همین دلیل، شما مجبور به استفاده از sendto() و recvfrom() نیستید؛ می توانید بسادگی از send() و recv() استفاده کنید.

۶. تکنیک های پیشرفته

تکنیک های ارائه شده از این بخش، واقعا پیشرفته نیستند، اما از موارد پایه ای که در قبل بحث کردیم سطح بالاتری دارند. در حقیقت اگر شما این نکات را عمیقا درک کنید، در این صورت می توانید بهمگان (!) بگویید که موارد پایه ای برای برنامه نویسی شبکه در یونیکس را فرا گرفته اید! بنابراین در این قسمت درباره چیزهای محرمانه و از نظر افتاده ای بحث خواهیم کرد که ممکن است برخی علاقمند به فراگرفتن آنها باشند.

۶/۱ عملیات Blocking

Blocking ! ممکن است درباره این کلمه چیزهایی شنیده باشید! اما واقعا blocking به چه معنی است؟ در مخلص کلام، "Block" واژه ای مشتق شده از "Sleep" می باشد. شاید فهمیده باشید که هنگامی که listener (موجود در بالا) را اجرا می کنید، این برنامه آنقدر صبر می کند تا اینکه یک بسته را دریافت کند. در این هنگام recvfrom() فراخوانی شده بوده است و فرض کنیم که این بسته شامل هیچ اطلاعات و داده ای نبوده است، لذا به recvfrom() دستور "block" داده می شود (یعنی در اینجا توقف کن) تا داده هایی را دریافت کنی.

بسیاری از توابع عمل block را انجام می دهند: Accept() و تمام توابع recv(). دلیل اینکه آنها قدرت این کار را دارند این است که به آنها اجازه اینکار داده شده است. هنگامی که نخست توصیفگر فایل را با socket() ایجاد می کنید، هسته آنرا در حالت blocking تنظیم می کند. اگر شما نمی خواهید که یک ساکت قابلیت blocking را داشته باشد، در این صورت مجبور به فراخوانی fcntl() هستید:

```
#include <unistd.h>
#include <fcntl.h>
.
.
```

[33] connected datagram sockets

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
    fcntl(sockfd, F_SETFL, O_NONBLOCK);
    .
.
```

با تنظیم کردن یک ساکت در حالت non-blocking یا غیر بلاکی (یعنی بدون قابلیت blocking)، شما می توانید اطلاعات را برای ساکت تعیین وضعیت کنید (**poll کنید**). اگر سعی بر خواندن از یک ساکت غیر-بلاکی کنید و هیچ داده ای در آنجا وجود نداشته باشد، در این صورت ساکت اجازه block کردن را ندارد – ساکت مقدار 1- را برگشت داده و errno را به صورت EWOULDBLOCK تنظیم خواهد کرد.

بطور کلی، به هر حال این نوع از polling ایده مناسبی نیست. اگر شما برنامه را در یک حالت busy-wait برای داده ها موجود روی ساکت قرار دهید، در این صورت CPU Time را بشدت مصرف می کنید. یک راه حل عاقلانه تر برای چک کردن اینکه آیا داده ای در انتظار خوانده شدن وجود دارد یا خیر، در بخش بعدی در توضیحات select() آورده خواهد شد.

۶/۲ Synchronous I/O Multiplexing – select()

این تابع اندکی ناشناس و عجیب و غریب به نظر می رسد، اما بسیار کاربردی و مفید خواهد بود. وضعیت زیر را در نظر بگیرید: شما روی یک سرور هستید و می خواهید برای ارتباطات ورودی فالگوش بایستید و همچنین عملیات خواندن را از ارتباطاتی که قبلا داشته اید، ادامه دهید. هیچ مشکلی نیست، شما تنها از یک accept() و چندین recv() استفاده می کنید. اما کار به این سرعت هم نخواهد بود. اگر شما روی یک فراخوانی accept() عملیات بلاک را انجام دهید چطور؟ شما چطور می خواهید داده ها را در زمان یکسان با استفاده از recv() دریافت کنید؟ شاید بگویید راه حل، "استفاد از ساکت های غیر-بلاکی می باشد!" اما هیچ راهی وجود ندارد! چون در این صورت منابع بسیاری از CPU مصرف می شوند. پس، چه باید کرد؟

select() به شما این امکان را می دهد تا در آن واحد چندین ساکت را مانیتور کنید. این تابع به شما خواهد گفت که کدام ساکت ها برای انجام عملیات خواندن، کدام ساکت برای نوشتن آماده هستند و در کدام ساکت ها، استثنائات افزایش یافته است (منظور از استثنائات، روتین ها و توابعی است که خطاها را تشخیص داده و پس از تشخیص خطا هنوز هم قابل اجرا باشند). در زیر ساختاری از select() را می بینید:

```
#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);
```

تابع، "مجموعه ای" از توصیفگر های فایل را مانیتور می کند، readfds، writefds و exceptfds. اگر می خواهید ببینید که آیا امکان خواندن از از ورودی استاندارد و یک توصیفگر فایل (sockfd) وجود دارد، در این صورت به توصیفگرهای فایل، صفر را اضافه کرده و sockfd را به readfds تنظیم کنید. پارامتر numfds را بایستی به مقادیر بالاترین توصیفگر فایل بعلاوه یک، تنظیم کنید. در این مثال، این پارامتر بایستی به صورت sockfd+1 تنظیم شود، چرا که مطمئنا از بقیه ورودی های استاندارد (0) بزرگتر است.

هنگامی که **select()** از اجرا باز می گردد، **readfds** تغییر داده شده تا منعکس سازد کدام یک از توصیفگرهای فایلی که شما انتخاب کرده اید، برای خواندن آماده هستند. شما می توانید آنها را با ماکروی FD_ISSET() (موجود در زیر) آزمایش کنید.

قبل از عملیات اضافی، ما درباره چگونگی دستکاری کردن این مجموعه ها صحبت خواهیم کرد. هر مجموعه از نوع fd_set می باشد. ماکروهای زیر با این نوع، بکار گرفته شده اند:

FD_ZERO(fd_set *set) -- clears a file descriptor set

FD_SET(int fd, fd_set *set) -- adds fd to the set

FD_CLR(int fd, fd_set *set) -- removes fd from the set

FD_ISSET(int fd, fd_set *set) -- tests to see if fd is in the set

بعضی مواقع، شما نمی خواهید که همیشه برای ارسال داده ها از سوی یک شخص صبر کرد. شاید بخواهید هر ۹۶ ثانیه، عبارت "Still Going..." را در ترمینال چاپ کنید، اگرچه در این زمان هیچ اتفاقی نیافته است. این ساختار زمان به شما اجازه می دهد که دوره های timeout را مشخص کنید. اگر زمان از مقدار تعیین شده خود تجاوز کرد و select() هنوز هیچ توصیفگر فایلی را پیدا نکرده بود، در این صورت از اجرا بازگشت خواهد یافت (return خواهد شد) و در این صورت شما می توانید به ادامه عملیات خود در برنامه بپردازید. ساختمان زمانی (struct timeval)، فیلدهای زیر را دارد:

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

تنها tv_sec را به تعداد ثانیه هایی که می خواهید منتظر بمانید تنظیم کنید و tv_usec را به میکروثانیه های انتظار تنظیم کنید. بلی، این فیلد را باید به صورت میکوثانیه (یک میلیونیوم ثانیه) تنظیم کنید نه میلی ثانیه (یک هزارم ثانیه). هزار میکروثانیه در یک میلی ثانیه وجود دارند و هزار میلی ثانیه در یک ثانیه وجود دارند. بنابراین، یک میلیون میلی ثانیه در یک ثانیه وجود دارند. حال شاید بپرسید چرا این اسم به صورت"usec" می باشد؟

در جواب باید گفت که ما حرف "u" را برای تجسم سازی حرف μ (مو) در زبان یونانی مورد استفاده قرار داده ایم که به معنای میکرو یا یک میلیونیوم می باشد. همچنین، هنگامی که تابع بر می گردد، ممکن است timeout بروز شود تا زمان باقیمانده مشخص و نمایش داده شود. این مسئله به توزیع یونیکسی که اجرا می کنید بستگی دارد.

ما یک زمان سنج تحلیلگر میکروثانیه ای[۳۴] داریم! تکه یا قطعه های استاندارد زمانی در یونیکس در حدود ۱۰۰ میلی ثانیه می باشد، پس تنها ممکن است مجبور باشید که بازای این مدت صبر کنید و لذا باید گفت که هیچ مشکلی وجود ندارد که شما چقدر ساختمان زمانی یا struct timeval خود را کوچک تنظیم کرده اید.

اکنون چندین نکته جالب را با هم بررسی می کنیم: اگر شما فیلدهای موجود در struct timeval را به 0 تنظیم کنید، select() بسرعت و بلافاصله timeout می شود و لذا توصیفگرهای فایل، مجموعه های شما را poll می کند. اگر پارامتر timeout را به NULL تنظیم کنید، در این صورت هیچ گاه timeout فرا نخواهید رسید و تا زمانی که اولین توصیفگرفایل آماده شود، صبر خواهد کرد. آخرین نکته اینکه، اگر شما توجهی جهت صبرکردن برای یک مجموعه مشخص ندارید، در این صورت می توانید آنرا در فراخوانی به select() به مقدار NULL تنظیم کنید. قطعه کد زیر به مدت ۲,۵ ثانیه صبر خواهد کرد تا چیزی روی ورودی استاندارد ظاهر شود:

```
/*
** 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;
```

[34] microsecond resolution timer

```
    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;
}
```

اگر شما روی یک ترمینال خطی بافرشده[۳۵] هستید، کلیدی که باید بفشارید RETURN می باشد و در غیر این صورت، در هر حال timeout خواهد شد.

اکنون بعضی از شما ممکن است فر کنید که این یک راه عالی جهت انتظار برای داده ها روی یک ساکت دیتاگرامی می باشد – و نظر شما درست است: ممکن است باشد. بعضی Unix ها ممکن است از select در این حالت استفاده کنند و بعضی دیگر نه. شما باید ببینید که صفحه اصلی و محلی سیستم شما چه می گویید اگر قصد دارید که این عمل را انجام دهید.

بعضی یونیکس ها زمان را در struct timeval شما بروز کرده تا مدت زمانی که هنوز قبل از یک timeout باقیمانده است را منعکس کنند، اما بعضی یونیکس ها این کار را انجام نمی دهند. بنابراین اگر می خواهید یک برنامه نویس حرفه ای بوده و برنامه های Portable ارائه کنید، نمی توانید به این مورد تکیه کنید (اگر می خواهید زمان انقضایی را بسنجید، از gettimeofday() استفاده کنید. البته روش جالبی نیست و یکی از راه ها می باشد).

اگر یک ساکت در مجموعه ای که عمل خواندن را انجام می دهند (مجموعه خواندن (read))، ارتباط را ببندد چه اتفاقی می افتد؟ در این مورد، select() با آن مجموعه توصیفگر ساکت، بصورت "ready to read" برگشت داده می شود. هنگامی که شما عملا recv() را انجام می دهید، recv() مقدار 0 را برمی گرداند و این روشی که شما می توانید بفهمید که کلاینت ارتباط را بسته است. یک نکته مهم درباره select(): اگر شما ساکتی دارید که در حال listen() می باشد، می توانید چک کنید که آیا ارتباطی جدید با قرار دادن توصیفگر فایلِ آن ساکت، در مجموعه readfds وجود دارد یا خیر و این مروری سریع بر تابع select() می باشد.

اما، بعنوان یک دایمون محبوب، در اینجا یک مثال نسبتا عمیق و ریشه ای را عنوان می کنیم. متاسفانه، تفاوت بین مثال بالا و این مثال، مهم و قابل توجه است. اما پیشنهاد می شود یک نگاه اجمالی به کد داشته و سپس از آغاز دستورها را به همراه توضیحات همراه آنها بخوانید. این برنامه، مانند یک سرور چند-کاربره[۳۶] ی چت، عمل می کند. این برنامه را در یک پنجره اجرا کنید، سپس از چندین سیستم ویندوز دیگر به آن تلنت کنید ("telnet <hostname> 9034"). هنگامی که در یک نشست تلنت چیزی را تایپ می کنید، بایستی در تمامی دیگر نشست ها نمایش یابد.

```
/*
** 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>
```

[35] a line buffered terminal
[36] multi-user chat server

```
#include <netinet/in.h>
      #include <arpa/inet.h>

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

int main(void)
{
   fd_set master;   // master file descriptor list
   fd_set read_fds; // temp file descriptor list for select()
   struct sockaddr_in myaddr;     // server address
   struct sockaddr_in remoteaddr; // client address
   int fdmax;        // maximum file descriptor number
   int listener;     // listening socket descriptor
   int newfd;        // newly accept()ed socket descriptor
   char buf[256];    // buffer for client data
   int nbytes;
   int yes=1;        // for setsockopt() SO_REUSEADDR, below
   int addrlen;
   int i, j;

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

   // get the listener
   if ((listener = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
      perror("socket");
      exit(1);
   }

   // lose the pesky "address already in use" error message
   if (setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes,
                                          sizeof(int)) == -1) {
      perror("setsockopt");
      exit(1);
   }

   // bind
   myaddr.sin_family = AF_INET;
   myaddr.sin_addr.s_addr = INADDR_ANY;


       }
                  printf("selectserver: new connection from %s on "
                     "socket %d\n", inet_ntoa(remoteaddr.sin_addr), newfd);
               }
            } else {
               // handle data from a client
               if ((nbytes = recv(i, buf, sizeof(buf), 0)) <= 0) {
                  // got error or connection closed by client
                  if (nbytes == 0) {
                     // connection closed
                     printf("selectserver: socket %d hung up\n", i);
                  } else {
                     perror("recv");
                  }
                  close(i); // bye!
```

```
                    FD_CLR(i, &master); // remove from master set
                } else {
                    // we got some data from a client
                    for(j = 0; j <= fdmax; j++) {
                        // send to everyone!
                        if (FD_ISSET(j, &master)) {
                            // except the listener and ourselves
                            if (j != listener && j != i) {
                                if (send(j, buf, nbytes, 0) == -1) {
                                    perror("send");
                                }
                            }
                        }
                    }
                }
            } // it's SO UGLY!
        }
    }

    return 0;
}
```

توجه کنید که ما دو مجموعه توصیفگر فایل در کد داریم: master و read_fds. اولین مجموعه یعنی master، تمام توصیفگرهای ساکت را که در حال حاضر متصل شده اند و همچنین توصیفگر ساکتی را که برای ارتباطات جدید فالگوش ایستاده است، نگه داری می کند.

دلیلی که ما در این کد مجموعه master را داریم این است که select() عملا مجموعه ای را که شما به آن منتقل می کنید تغییر داده تا منعکس سازد کدام ساکت ها برای عملیات خواندن آماده هستند. چون ما مجبور بودیم که ردپای ارتباطات را از یک فراخوانی به select() به فراخوانی بعدی، نگهداری کنیم، لذا باید این اطلاعات را در جایی ذخیره کنیم. آخرین گام اینکه ما master را در read_fds کپی می کنیم و سپس select() را فراخوانی می کنیم.

شاید بپرسید، آیا این مسئله به این معنی است که هر هنگامی که یک ارتباط جدید را دریافت می کنیم، مجبور به اضافه کردن آن به مجموعه master هستیم؟ در جواب باید گفت که بلی و همچنین هر هنگام که یک ارتباط بسته می شود، مجبور به حذف کردن آن از مجموعه master هستیم.

توجه کنید که ما بررسی می کنیم که ساکت در حال شنود، چه هنگام برای عملیات خواندن آماده است. هنگامی که این ساکت آماده شد، به این معنی است که ما یک ارتباط جدید دوره ای داریم و آنرا accept() کرده و آنرا به مجموعه master اضافه می کنیم. بهمین نحو، هنگامی که یک ارتباط کلاینت برای عملیات خواندن آماده است و recv() مقدار 0 را بر می گرداند، آنگاه می فهمیم که کلاینت ارتباط را بسته است و لذا باید آنرا از مجموعه master حذف کنیم.

اگر recv() در کلاینت، مقدار غیر صفری را برگشت دهد، در این صورت خواهیم فهمید که داده هایی دریافت شده است. لذا آن داده ها را گرفته و سپس به لیست master رفته و آن داده ها را به دیگر کلاینت های متصل ارسال می کنیم.

۶/۳ کارکردن با send() های ناتمام و بخش به بخش!

آیا توضیحاتی که در قبل درباره send() دادیم را هنگامی که send()تمام بایت هایی را که شما می خواهید ممکن است نفرستد، به یاد می آورید؟ مثلا، شما می خواهید ۵۱۲ بایت را بفرستید، اما این تابع ۴۱۲ را بر می گرداند. چه اتفاقی برای ۱۰۰ بایت باقیمانده افتاده است؟

آنها هنوز در بافر کوچک شما قرار دارند و جهت ارسال منتظر می باشد. ناشی از شرایطی که در ورای کنترل شماست، هسته تصمیم گرفته که تمام داده ها را در یک قطعه یا تکه داده ای نفرستد و اکنون، این بر عهده شماست که داده های باقیمانده را ارسال کنید. شما می توانید تابعی مانند زیر نیز بنویسید که این کار را برای شما انجام دهد:

```
#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
}
```

در این مثال، s ساکتی است که شما می خواهید داده ها را به آن ارسال کنید، buf بافری است که حاوی داده ها می باشد و len اشاره گر int بوده که حاوی تعداد بایت های موجود در بافر می باشد. تابع هنگام بروز خطا مقدار 1- را برگشت می دهد (و errno هنوز از فراخوانی به send() تنظیم شده است). همچنین تعداد بایت هایی که عملا فرستاده شده اند نیز در len برگشت داده می شوند. این تعداد، تعداد بایت هایی خواهد بود که شما قصد ارسال آنها را داشته اید، مگراینکه در این حین خطایی رخ داده باشد. برای تکمیل این بحث، در زیر یک فراخوانی نمونه به تابع را می بینید:

```
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);
}
```

هنگامی دریافت بخشی از یک بسته، چه اتفاقی برای طرف گیرنده می افتد؟ اگر بسته ها دارای طول متغیر باشند، گیرنده چطور می فهمد که چه موقع چه بسته تمام شده و چه موقع بسته ای دیگر شروع می شود؟ در جواب باید گفت که احتمالا مجبور به کپسوله سازی یا Encapsulate کردن بسته ها هستید (در ابتدای مقاله توضیح مختصری داده شد). اکنون جهت درک جزئیات بیشتر بخش زیر را بخوانید!

۶/۴ نظریه کپسوله سازی داده ها (Data Encapsulation)

کپسوله سازی داده ها به چه معنی است؟ در ساده ترین صورت، به این معنی است که شما یک هدر را به مقداری اطلاعات قابل تشخیص یا طول بسته یا هر دو می چسبانید. شاید سوال شود که هدرها ما به چه شکلی باید باشند؟ جواب این است که، در حقیقا یک هدر تنها مقدار داده باینری بوده که چیزهایی که شما برای تکمیل فرآیند ارتباط یا پروژه خود لازم می دانید را نمایش می دهند. البته این جمله مبهم است. جهت تفهیم، فرض کنیم که یک برنامه چت چندکاربره دارید که از SOCK_STREAM ها استفاده می کند. هنگامی که یک کاربر چیزی را تایپ می کند (می گوید)، نیاز به انتقال دو قطعه از اطلاعات به سرور می باشد: چه چیزی گفته شد (پیام) و چه کسی آنها را گفت (مبدا). اکنون مشکل چیست؟

مشکل این است که پیام ها می توانند طول های مختلفی داشته باشند. یک شخص با نام "tom" ممکن است بگوید: "Hi" و شخص دیگری با نام "Benjamin" ممکن است بگوید: "Hey guys What is up?". بنابراین، شما تمام این چیزها را به محض ورود، به کلاینت ها send() می کنید. جریان (stream) خروجی داده ها مانند زیر هستند:

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 ?

و بهمین صورت تا آخر. چگونه کلاینت می فهمد که چه هنگام یک پیام شروع شده و دیگری متوقف گشته است؟ یک راه این است که طول همه پیام ها را یکسان کرده و تابع sendall() را فراخوانی کنید. اما این کار پهنای باند را بهدر می دهد! ما نمی خواهیم 1024 بایت داده را send() که "Ashkan" بگوید "Hi". لذا ما داده ها را در یک هدر کوچک و ساختمان بسته ای کپسوله می کنیم. هم کلاینت و هم سرور می دانند که چطور این داده ها را pack و unpack کنند (بعضی مواقع نیز "marshal" و "unmarshal" می گویند). می خواهیم یک پروتکل را تعریف کنیم که چگونگی ارتباط یک کلاینت و سرور را شرح می دهد!

در این مورد، بیایید فرض کنیم که نام کاربر، طولی ثابتِ ۸ کاراکتری داشته که با '\0' خاتمه یافته و در کنار آن فرض می کنیم که داده ها دارای طول متغیری بوده و می توانند حداکثر ۱۲۸ کاراکتر باشند. اکنون به یک ساختمان بسته ای نمونه که ممکن است در این گونه شرایط مورد استفاده قرار گیرد، نگاه می کنیم:

len (1 byte, unsigned) -- The total length of the packet, counting the 8-byte user name and chat data.

name (8 bytes) -- The user's name, NUL-padded if necessary.

chatdata (n-bytes) -- The data itself, no more than 128 bytes.

طول بسته بایستی بصورتِ "طول این داده ها+ ۸" محسابه شود (که ۸، طول فیلد نام در بالا می باشد). چرا ما محدودیت های ۸-بایتی و ۱۲۸-بایتی را برای فیلدها انتخاب کردیم؟ بهرحال، ما بنابه حدس و شرایط مناسب حدس زدیم که این طول برای این دو فیلد کافی و مناسب هستند. اگرچه، شاید ۸ بایت برای شما بسیار کم باشد و بخواهید یک فیلد ۳۰ بایتی برای نام انتخاب کنید. به هر حال، این موارد بر عهده خودتان می باشد. با استفاده از این تعریف بسته که در بالا ذکر شد، اولین بسته، حاوی اطلاعات زیر خواهد بود (در حالت hex و ASCII):

```
 0A       74 6F 6D 00 00 00 00 00     48 69
(length)  T  o  m  (padding)          H  i
```

و دومین بسته به شکل زیر خواهد بود:

```
 14     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 ...
```

lengh (تعداد بایت های بسته)، در حالت ترتیب بایت شبکه ای ذخیره شده است که البته در این مورد، تنها یک بایت بوده و زیاد مهم نخواهد بود. اما در حالت کلی، شما باید تمام اعداد باینری صحیح خود را به صورت ترتیب بایت شبکه ای در بسته هایتان ذخیره کنید.

هنگامی که این داده ها را می فرستید باید ایمن باشید و از یک دستور، مشابه sendall() (در بالا) استفاده کنید، پس شما می دانید که تمام داده ها فرستاده شده اند، حتی اگر برای خارج شدن تمام آنها، چندین فراخوانی به send() نیاز باشد. همچنین، هنگام دریافت این داده ها، باید کار بیشتری انجام دهید. برای اینکه ایمن باشید، بایستی فرض کنید که ممکن است یک تکه بسته را دریافت کنید (مثلا ممکن است ما "00 14 42 65 6E" را از بنجامین دریافت کنیم، اما این تمام چیزی است که ما در این فراخوانی به recv() دریافت می کنیم). نیاز هست که چندین و چند بار recv() را فراخوانی کنیم تا اینکه بسته کاملا دریافت گردد. اما چگونه؟

ما تعداد بایت هایی را که برای دریافت کامل یک بسته نیاز به دریافت داریم می دانیم، چون این تعداد در جلوی بسته ضمیمه شده است. همچنین می دانیم که اندازه ماکزیمم بسته، 128+8+1 یا 137 بایت می باشد (چرا که ما بسته را اینگونه تعریف کرده ایم). کاری که ما می توانیم انجام دهیم این است که یک آرایه اعلان کنیم که به اندازه کافی برای دو بسته بزرگ باشد. این آرایه، آرایه کاری ما است و هنگامی که دریافت بسته ها، مجددا آنها را می سازیم. هر کدام که شما داده ها را recv() می کنید، باید آنرا به بافر کاری خود داده و چک کنید که آیا بسته کامل شده است یا خیر. روش کار به این صورت است که تعداد بایت های موجود در بافر، بزرگتر یا مساوی طول تعیین شده در هدر (+۱ – چرا که length موجود در هدر، بایتی متعلق به خودش، lengh را حساب نمی کند) می باشد. بدیهی است که اگر تعداد بایت های باقیمانده در بافر کوچکتر از ۱ باشد، بسته کامل نشده است. شما مجبور هستید که برای این مسئله، جعبه مخصوصی ایجاد کنید، چرا که اولین بایت، زباله ای (garbage) و ناخواسته و ... بوده و شما نمی توانید برای طول صحیح بسته به آن تکیه کنید. هنگامی که بسته کامل شد، از آن استفاده می کنید و سپس آنرا از بافر کاری خود حذف می کنید. البته برای کامل شدن این بحث باید یک کتاب حول ساختمان داده ها را بخوانید و از آن بعد می توانید براحتی موارد زیر را درک کنید (البته تمرین را نباید فراموش کرد).

۷. سوال و جواب (Q&A)

سوال: از کجا می توان فایل های هدر را گرفت؟

جواب: اگر شما از قبل آنها را روی سیستمتان ندارید، احتمالا به آنها احتیاج نخواهید داشت! راهنمای پلاتفرم خود را بررسی کنید. اگر شما با ویندوز کار می کنید، تنها نیاز به #include <winsock.h> می باشد.

سوال: هنگامی که bind() پیام "Address already in use" را گزارش می کند، چه باید کرد؟

جواب: شما مجبورید تا از setsockopt() با option یا گزینه ی SO_REUSEADDR، روی ساکت در حال شنود استفاده کنید. برای مثال قسمت bind() و select() را نگاه کنید.

سوال: چطور لیستی را از ساکت های باز روی سیستم بدست آورم؟

جواب: از netstat استفاده کنید. برای جزئیات بیشتر می توانید به مقالات و ... که در سایت های مختلفی آموزش داده شده اند رجوع کنید. اما خود دستور netstat $ خروجی های مناسب و خوبی را بدست شما خواهد داد. بهرحال، تنها نکته و حقه ای فرا گرفتن آن بسیار حائز اهمیت می باشد، این است که مشخص کنیم کدام ساکت با کدام برنامه مرتبط است!

سوال: چطور می توان جدول مسیریابی را نمایش داد؟

جواب: دستور route (که در بسیاری از لینوکس ها، در /sbin وجود دارد) یا دستور netstat –r را اجرا کنید.

سوال: اگر من تنها یک کامپیوتر داشته باشم، چطور می توانم برنامه ها کلاینت و سرور را اجرا کنم؟ آیا برای نوشتن برنامه های شبکه ای، نیاز به یک شبکه دارم؟

جواب: خوشبختانه، تقریبا روی تمام ماشین ها یک " device، وسیله یا ابزارِ" شبکه ای loopback پیاده سازی شده که در هسته قرار گرفته و خود را بعنوان یک کارت شبکه وانمود می کند (این واسطی است که در جدول مسیریابی به صورت "lo" لیست می شود – lo مخفف عبارت local). فرض کنید که شما در یک ماشین با نام "goat" لاگین شده اید. کلاینت را در یک پنجره و سرور را در پنجره ای دیگر اجرا کنید یا سرور را در background یا پشت صحنه ("server &") اجرا کرده و کلاینت را در همان پنجره اجرا کنید. نتیجه ی وسیله loopback روی سیستم این است که شما می توانید هم کلاینت goat یا کلاینت localhost باشید (چرا که احتمالا در فایل /etc/hosts، عبارت "localhost" تعریف شده است) و هم بدون شبکه (!)، از کلاینت با سرور به گفت و گو بپردازید! بطور خلاصه، برای اینکه کدها روی یک ماشین واحد غیرمتصل به شبکه کار کنند، هیچ تغییری در هیچ یک از کدها نیاز نیست!

سوال: چطور می توانیم بگوییم که طرفِ ریموت (طرفین موجود در ارتباط بین کلاینت و سرور)، ارتباط را بسته است؟

جواب: در صورتی که recv() مقدار 0 را برگشت دهد، ارتباط بسته شده است.

سوال: چطور می توان یک ابزار "ping" را پیاده سازی کرد؟ ICMP چیست؟ کجا می توان اطلاعات بیشتری راجع به ساکت های خام (raw) و SOCK_RAW بدست آورد؟

جواب: تمام سوالات شما درباره ساکت های خام در سری کتاب های UNIX Network Programming تالیف W. Richard Stevens جواب داده شده اند.

سوال: چطور فایل های Solaris/SunOS را بسازیم (یا build کنیم – پسوند exe.)؟ هنگام کامپایل، خطاهای اتصالی یا لینکی دریافت می کنم.

جواب: خطاهای لینکی (یا خطاهای برنامه linker – linker errors)، به این دلیل اتفاق می افتند که سیستم های Sun، بصورت خودکار در کتابخانه های ساکت کامپایل نمی گردند.

سوال: چرا select() عبور و مرور سیگنال را نگه داری می دارد (منظور: چطور با این تابع، سیگنال سیر خود را دارد و منحرف و ... نمی شود) ؟

جواب: سیگنال ها سبب می شوند که فراخوانی های سیستمی بلاک شده[۳۷]، مقدار 1- را برگشت داده و errno را به EINTR تنظیم کنند. هنگامی که شما یک نگهدارنده سیگنال (signal handler) را با sigaction() نصب می کنید، می توانید فلگ SA_RESTART را تنظیم کنید که سبب می شود بعد از انقطاع، فراخوانی سیستم راه اندازی مجدد شود. در حالت طبیعی، این مورد همیشه کارکرد نخواهد داشت. راه حل زیرکانه ای که برای این مشکل پیشنهاد می شود، استفاده از یک دستور goto می باشد.

```
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");
   }
```

[37] blocked system calls

مطمئنا احتیاجی به استفاده از goto در این مورد ندارید؛ شما می توانید از ساختارهای دیگر برای کنترل آن استفاده کنید. اما مطمئنا جمله goto برای همگان واضح تر می باشد.

سوال: چطور می توانم یک عملیات timeout را در یک فراخوانی به recv() پیاده سازی کنیم؟

جواب: از select() استفاده کنید! این تابع به شما اجازه می دهد تا پارامتر timeout را برای توصیفگرهای فایلی که می خواهید روی آنها عمل خواندن را انجام دهید تنظیم کنید. در غیر این صورت، می توانید کل عاملیت را در یک تابع واحد پوشش دهید، مانند زیر:

فرض کنید که کلاینت connect() شده، داده ها را send() کرده و ارتباط را close() می کند (هیچ فراخوانی سیستمی اضافی، بدون اتصال مجددا کلاینت وجود ندارد). فرآیندی که کلاینت دنبال می کند به صورت زیر می باشد:

* connect() شدن به سرور
* دستور send("/sbin/ls > /tmp/client.out")
* close() کردن ارتباط

ضمنا، سرور داده را بررسی کرده و آنرا اجرا می کند:

* accept() کردن ارتباط از کلاینت
* دستور رشته ای recv(str)
* close() کردن ارتباط
* system(str)، جهت اجرای دستور

آگاه باشید! اینکه سرور مجبور باشد هر چیزی که کلاینت می گوید اجرا کند، شبیه این است که یک remote shell را در اختیار آن قرار داده باشیم و در این صورت افراد می توانند هنگام اتصال به سرور، روی اکانت شما کارهای مختلفی انجام دهند. برای نمونه، در مثال بالا، اگر کلاینت عبارت "rm -rf ~" را ارسال کند، چه اتفاقی می افتد؟ جواب واضح است، هر چیزی که در اکانت شما وجود داشته باشد، پاک خواهد شد! لذا باید ذکاوت بخرج دهیم و بغیر از ابزاری که می دانیم ایمن هستند، کلاینت را از اجرا باز داریم، مانند ابزار foobar:

```
if (!strcmp(str, "foobar")) {
   sprintf(sysstr, "%s > /tmp/server.out", str);
   system(sysstr);
}
```

اما شما هنوز هم نا امن هستید: اگر کلاینت عبارت "foobar; rm -rf ~" را وارد کند چه می شود؟ امن ترین چیز این است که یک روتین (قاعده یا روال) کوچک بنویسید که یک کاراکتر escape (یا "\") را در جلوی تمام کاراکترهای غیر الفایی (که در اقتضاء، شامل space ها هم می شود) در آرگومان های یک دستور قرار دهد.

سوال: من مقدار زیادی از داده ها را می فرستم، اما هنگامی که آنها را recv() می کنم، تنها ۵۳۶ بایت یا ۱۴۶۰ بایت در یک زمان دریافت می کنم. اما اگر برنامه را روی ماشین محلی خودم اجرا کنم، تمام داده ها را در یک زمان دریافت می کنم. مشکل چیست؟

جواب: داده های شما در حد MTU شده است – بیشترین اندازه برای داده که رسانه فیزیکی می تواند حمل کند. روی ماشین محلی، شما از ابزار loopback استفاده می کنید که می تواند 8K (یا بیشتر) را بدون هیچ مشکلی حمل کند. اما روی Ethernet که می تواند تنها ۱۵۰۰ بایت با یک هدر را حمل کند، شما از آن حد تجاوز خواهید کرد. روی یک مودم با MTU 576 (بهمراه هدر)، شما حد و مرز کمتری هم خواهید داشت! ابتدا، مجبور هستید تا اطمینان حاصل کنید که تمام داده ها فرستاده می شود (پیاده سازی

تابع sendall() را برای جزئیات بیشتر ببینید). هنگامی که از این مسئله حصول اطمینان کردید، آنگاه تا خواندن شدن تمام داده های خود، باید recv() را در یک حلقه loop فراخوانی کنید.

سوال: من روی ویندوز برنامه نویسی می کنم و هیچ فراخوانی سیستمی fork() یا هیچ نوعی از struct sigaction وجود ندارد، چه کاری باید انجام دهم؟

جواب: این موارد در کتابخانه های POSIX هستند که ممکن است با کامپایلر شما پیوند خورده باشند. به نظر می رسد که Mirrosoft یک لایه سازگاری با POSIX دارد و ممکن است تابع fork() در آنجا رار داشته باشد (و یا حتی sigaction). می توان برای اطلاعات بیشتر VC++ را برای کلمات "fork" و "POSIX" سرچ کرد. در صورتی که به هیچ وجه به جواب نرسیدید، از fork()/sigaction صرف نظر کرده و آنرا با معادل Win32 خود جایگزین کنید: **CreateProcess()**.

۸. منابع

۸/۱ کتب

Unix Network Programming, volumes 1-2 by W. Richard Stevens. Published by Prentice Hall. ISBNs for volumes 1-2: 013490012X[47], 0130810819[48].

Internetworking with TCP/IP, volumes I-III by Douglas E. Comer and David L. Stevens. Published by Prentice Hall. ISBNs for volumes I, II, and III: 0130183806[49], 0139738436[50], 0138487146[51].

TCP/IP Illustrated, volumes 1-3 by W. Richard Stevens and Gary R. Wright. Published by Addison Wesley. ISBNs for volumes 1, 2, and 3: 0201633469[52], 020163354X[53], 0201634953[54].

TCP/IP Network Administration by Craig Hunt. Published by O'Reilly & Associates, Inc. ISBN 1565923227[55].
Advanced Programming in the UNIX Environment by W. Richard Stevens. Published by Addison Wesley. ISBN 0201563177[56].

Using C on the UNIX System by David A. Curry. Published by O'Reilly & Associates, Inc. ISBN 0937175234. *Out of print.*

۸/۲ ارجاع ها در وب

BSD Sockets: A Quick And Dirty Primer[57] (has other great Unix system programming info, too!)
The Unix Socket FAQ

Client-Server Computing

Intro to TCP/IP[60] (gopher)

Internet Protocol Frequently Asked Questions

The Winsock FAQ

۸/۳ RFC ها

RFC-768–The User Datagram Protocol (UDP)
RFC-791–The Internet Protocol (IP)
RFC-793–The Transmission Control Protocol (TCP)
RFC-854–The Telnet Protocol
RFC-951–The Bootstrap Protocol (BOOTP)
RFC-1350–The Trivial File Transfer Protocol (TFTP)

۸/۴ یادداشت ها

1. http://www.ecst.csuchico.edu/~beej/guide/net/
2. http://tangentsoft.net/wskfaq/
3. http://www.tuxedo.org/~esr/faqs/smart-questions.html
4. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/send.2.inc
5. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/recv.2.inc
6. http://www.rfc-editor.org/rfc/rfc793.txt
7. http://www.rfc-editor.org/rfc/rfc791.txt
8. http://www.rfc-editor.org/rfc/rfc768.txt
9. http://www.rfc-editor.org/rfc/rfc791.txt
10. http://www.rfc-editor.org/rfc/rfc1413.txt
11. http://www.ecst.csuchico.edu/~beej/guide/net/examples/getip.c
12. http://www.ecst.csuchico.edu/~beej/guide/net/examples/server.c
13. http://www.ecst.csuchico.edu/~beej/guide/net/examples/client.c

14. http://www.ecst.csuchico.edu/~beej/guide/net/examples/listener.c

15. http://www.ecst.csuchico.edu/~beej/guide/net/examples/talker.c
16. http://www.ecst.csuchico.edu/~beej/guide/net/examples/select.c
17. http://www.ecst.csuchico.edu/~beej/guide/net/examples/selectserver.c
18. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/htonl.3.inc
19. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/htons.3.inc
20. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/ntohl.3.inc
21. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/ntohs.3.inc
22. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/inet_aton.3.inc
23. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/inet_addr.3.inc
24. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/inet_ntoa.3.inc
25. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/socket.2.inc
26. http://linux.com.hk/man/showman.cgi?manpath=/man/man7/socket.7.inc
27. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/bind.2.inc
28. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/connect.2.inc
29. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/listen.2.inc
30. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/accept.2.inc
31. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/send.2.inc
32. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/recv.2.inc
33. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/sendto.2.inc
34. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/recvfrom.2.inc
35. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/close.2.inc
36. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/shutdown.2.inc
37. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/getpeername.2.inc
38. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/getsockname.2.inc
39. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/gethostbyname.3.inc
40. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/gethostbyaddr.3.inc
41. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/getprotobyname.3.inc
42. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/fcntl.2.inc
43. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/select.2.inc
44. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/perror.3.inc
45. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/gettimeofday.2.inc
46. http://www.amazon.com/exec/obidos/redirect-home/beejsguides-20
47. http://www.amazon.com/exec/obidos/ASIN/013490012X/beejsguides-20
48. http://www.amazon.com/exec/obidos/ASIN/0130810819/beejsguides-20
49. http://www.amazon.com/exec/obidos/ASIN/0130183806/beejsguides-20
50. http://www.amazon.com/exec/obidos/ASIN/0139738436/beejsguides-20
51. http://www.amazon.com/exec/obidos/ASIN/0138487146/beejsguides-20
52. http://www.amazon.com/exec/obidos/ASIN/0201633469/beejsguides-20
53. http://www.amazon.com/exec/obidos/ASIN/020163354X/beejsguides-20
54. http://www.amazon.com/exec/obidos/ASIN/0201634953/beejsguides-20
55. http://www.amazon.com/exec/obidos/ASIN/1565923227/beejsguides-20
56. http://www.amazon.com/exec/obidos/ASIN/0201563177/beejsguides-20
57. http://www.cs.umn.edu/~bentlema/unix/
58. http://www.ibrado.com/sock-faq/
59. http://pandonia.canberra.edu.au/ClientServer/
60. gopher://gopher-chem.ucdavis.edu/11/Index/Internet_aw/Intro_the_Internet/intro.to.ip/
61. http://www-iso8859-5.stack.net/pages/faqs/tcpip/tcpipfaq.html
62. http://tangentsoft.net/wskfaq/
63. http://www.rfc-editor.org/
64. http://www.rfc-editor.org/rfc/rfc768.txt
65. http://www.rfc-editor.org/rfc/rfc791.txt
35. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/close.2.inc
36. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/shutdown.2.inc
37. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/getpeername.2.inc
38. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/getsockname.2.inc
39. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/gethostbyname.3.inc
40. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/gethostbyaddr.3.inc
41. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/getprotobyname.3.inc

42. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/fcntl.2.inc
43. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/select.2.inc
44. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/perror.3.inc
45. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/gettimeofday.2.inc
46. http://www.amazon.com/exec/obidos/redirect-home/beejsguides-20
47. http://www.amazon.com/exec/obidos/ASIN/013490012X/beejsguides-20
48. http://www.amazon.com/exec/obidos/ASIN/0130810819/beejsguides-20
49. http://www.amazon.com/exec/obidos/ASIN/0130183806/beejsguides-20
50. http://www.amazon.com/exec/obidos/ASIN/0139738436/beejsguides-20
51. http://www.amazon.com/exec/obidos/ASIN/0138487146/beejsguides-20
52. http://www.amazon.com/exec/obidos/ASIN/0201633469/beejsguides-20
53. http://www.amazon.com/exec/obidos/ASIN/020163354X/beejsguides-20
54. http://www.amazon.com/exec/obidos/ASIN/0201634953/beejsguides-20
55. http://www.amazon.com/exec/obidos/ASIN/1565923227/beejsguides-20
56. http://www.amazon.com/exec/obidos/ASIN/0201563177/beejsguides-20
57. http://www.cs.umn.edu/~bentlema/unix/
58. http://www.ibrado.com/sock-faq/
59. http://pandonia.canberra.edu.au/ClientServer/
60. gopher://gopher-chem.ucdavis.edu/11/Index/Internet_aw/Intro_the_Internet/intro.to.ip/
61. http://www-iso8859-5.stack.net/pages/faqs/tcpip/tcpipfaq.html
62. http://tangentsoft.net/wskfaq/
63. http://www.rfc-editor.org/
64. http://www.rfc-editor.org/rfc/rfc768.txt
65. http://www.rfc-editor.org/rfc/rfc791.txt

تالیف: سعید بیکی (cephexin@secumania.net)