فهرست

به نام یگانه برنامه‌نویس عالم

مقدمه مترجم

این نوشتار ترجمه ای است از مقاله Beej's Guide to Network Programming Using Internet Sockets که توسط آقای Brain Hall گردآوری و نوشته شده است. این ترجمه مربوط به نسخه ای از این مقاله است که در سال ۲۰۰۱ میلادی گردآوری شده و آخرین نسخه ای بود که مترجم موفق به دریافت آن شد.

به علت نثر خاص مولف مقاله و شوخی ها و الفاظ مطایبه آمیز موجود در آن، برگردان مقاله به فارسی اندکی با دشواری روبرو بود. بنابراین ممکن است بخش هایی از مقاله به نظر متفاوت بیاید.

مترجم تمامی تلاش خود را برای برگردان روان متن به کار گرفته است. در بعضی قسمت‌ها برای مفهوم شدن برخی عناوین متن افزوده هایی در متن آورده‌ایم (مثلا در متن اصلی داشتیم: socket() و ما در برگردان تابع socket() ترجمه کردیم) و یا توضیحاتی را برای تکمیل مباحث در پاورقی اضافه کرده‌ایم و برای رعایت امانت حرف –م– را به عنوان نشانه افزودنی مترجم قرار داده‌ایم. امید است مورد استفاده محققان و دانشجویان عزیز قرار گیرد. بدیهی است که هر گونه انتقاد و نظر سازنده در پیشبرد هر تحقیقی لازم خواهد بود.

در پایان لازم می دانم مراتب قدردانی و سپاسگزاری صمیمانه خود را از خواهر عزیزم خانم زهرا صابری که در ویرایش این مقاله مرا یاری دادند به جای آورم. و سپاس او را.

غلامرضا صابری تبریزی

آبان ۱۳۸۶

مخاطبین:

این مقاله مجموعه‌ای آموزشی(تور) است و می‌تواند برای اشخاصی که بـه دنبـال جـای پـایی در برنامـه نویسی Socket می‌گردند مفید واقع شود. گفتنی است این مقاله مطمئنا یک مرجع کامل در این زمینه به حساب نمی‌آید.

کامپایلر و پلتفرم مورد استفاده:

کدهای موجود در این مقاله در کـامپیوتری بـا سیـستم عامـل Linux و بـا اسـتفاده از کامپـایلر gcc کامپایل شده است. در واقع در هر کامپیوتر و سیستم عاملی که از gcc استفاده می‌کند می‌توان کدهای موجود در این مقاله را اجرا کرد. به یاد داشته باشید اگر می‌خواهید برای ویـنـدوز برنامـه نویـسی نماییـد این کدها به خاطر برخی مسائل کامپایل نخواهند شد. برای برنامـه نویـسی سـوکت تحـت ویـنـدوز مـی توانید به مراجع برنامه نویسی سوکت در ویندوز مراجعه نمایید البته ذکر این نکته را هم خـالی از لطـف نمیبینم که اکثر توابع در ویندوز و لینوکس یکسان است فقط در ویندوز چند کار اضافی دیگر هم بایـد انجام گیرد.

سوکت چیست؟

مسلما قبل از شروع کار با هر چیزی، شناخت آن می تواند بسیار مفید واقـع شـود. شـما حتمـا دربـاره سوکت‌ها شنیده‌اید و شاید از خود پرسیده باشید که سوکت‌ها واقعا چه هستند؟ خوب پاسخ سوال شـما این است: روشی برای تعامل با برنامه‌های دیگر از طریق توصیف‌کننده‌های استاندارد فایل در UNIX.
ممکن است بپرسید این به چه معناست؟

خوب ممکن است چنین جمله‌ای را از یک هکر یونیکس شنیده باشید: "یا عیسی مسیح! همه چیـز در یونیکس یک فایل است!!" چیزی که این شخص راجع به آن صحبت می کـرده در واقـع ایـن حقیقـت است که برنامه‌های یونیکس هر نوع عملیات ورودی و خروجی(I/O) را به واسطه خوانـدن یـا نوشـتن در یک توصیف‌کننده فایل(File Descriptor) انجام می‌دهند.

توصیف‌کننده فایل، یک عدد صحیح است که با یک فایل باز در ارتباط است. اما (نکته هم در همین اما است) آن فایل باز می تواند یک ارتبـاط شـبکه(Network Connection)، FIFO ، Pipe، ترمینال یا یک فایل عادی روی هارد دیسک باشد و یا هـر چیـز دیگـری کـه بـا ورودی و خروجـی در ارتباط است.

"همه چیز در یونیکس یک فایل است!" بنابراین وقتی شما مـی خواهیـد بـا برنامـه‌ای دیگـر از طریـق اینترنت ارتباط برقرار نمایید باید این کار را از طریق توصیف‌کننده های فایـل انجـام دهیـد. بهتـر اسـت باور کنید.

سوال: از کجا می توانم این توصیف‌کننده های فایل را برای ارتباطات شبکه‌ای به دست آورم؟ این ممکن است در حال حاضر آخرین سوال باقیمانده در ذهن شما باشد. من بدین گونه پاسخ شما را می‌دهم: شما یکی از روتین های سیستم به نام socket() را فراخوانی می‌نمایید. این فراخوان توصیف‌کننده سوکت را به شما (برنامه شما) بر می گرداند و شما از طریق این توصیف‌کننده و با استفاده از دو تابع send() و recv() اطلاعات را فرستاده و دریافت می‌نمایید.

حال ممکن است از خود بپرسید: "اگر مقدار بازگشتی تابع socket() یک توصیف‌کننده فایل است چرا نمی توان از توابع عادی read() و write() برای خواندن و نوشتن از و در آن استفاده کرد؟"

پاسخ کوتاه این است: "شما می توانید" و پاسخ بلندتر: "شما می توانید اما توابع send() و recv() کنترل بیشتری را روی انتقال اطلاعات در اختیار شما قرار می دهند."

اما می‌رسیم به انواع سوکت ها. سوکت‌ها بسیار متنوع هستند. برای مثال آدرس های اینترنتی DARPA (سوکت های اینترنتی)، نام مسیرها در یک گره محلی (سوکت های یونیکس)، CCITT x.25 (که می توانید از این نوع با اطمینان صرفنظر نمایید.) و بسیاری انواع دیگر از سوکت هایی که برخی از آنها به نوع یونیکسی که استفاده می کنید بستگی دارند. این مقاله فقط به نوع اول سوکت هایی که ذکر شد می پردازد و آن هم همان طور که حدس زده‌اید سوکت های اینترنت هستند.

دو نوع سوکت اینترنت:

دو نوع سوکت اینترنتی وجود دارد! آیا واقعا دو نوع سوکت اینترنتی وجود دارد؟ خوب در پاسخ به شما باید بگویم بله و خیر. خیر از این لحاظ که انواع بیشتری از سوکت های اینترنتی وجود دارند ولی من قصد گیج کردن شما را ندارم و بله از این لحاظ که در این مقاله فقط در مورد دو نوع از سوکت های اینترنتی بحث به میان می آورم. خوب وقت صحبت کردن در مورد این دو نوع فرا رسیده است. این دو نوع کدامند؟ یکی از آنها سوکت های جریان(Stream Sockte) و دیگری سوکت های دیتا گرام (Data Gram) هستند که از این قسمت به بعد به آنها به ترتیب SOCK_STREAM و SOCK_DGRAM نیز می گوییم. در برخی از مواقع به سوکت های دیتاگرام ،سوکت های فاقد ارتباط(Connectionless) نیز می گویند (البته این نامگذاری بدین معنی نیست که این سوکت‌ها نمی‌توانند متصل شوند، بلکه دلایل دیگری دارد که در ادامه به آنها خواهیم پرداخت).

سوکت‌های جریان، ارتباطات(جریانها)ی دوطرفه قابل اعتماد هستند؛ یعنی اگر شما دو آیتم را با ترتیب ۱و۲ از طریق این سوکت‌ها ارسال نمایید این اطلاعات در طرف دیگر با همین ترتیب می‌رسد. این نوع سوکت‌ها اصطلاحا بدون خطا یا Erro free هستند.

سوال: کاربرد سوکت‌های جریان چیست؟ خوب در پاسخ باید بگویم حتما تا به حال در باره telnet شنیده‌اید. این برنامه از سوکت های جریان استفاده می کند. تمامی کراکتر هایی که شما در این برنامه تایپ می کنید با همان ترتیبی که فرستاده شده‌اند به طرف دیگر می رسند. در ضمن مثال دیگری که در این زمینه می توان ذکر کرد مرورگرهای وب است که با استفاده از پروتکل HTTP به دریافت صفحات وب می پردازند. این پروتکل هم از سوکت های جریان برای دریافت صفحات وب استفاده می کند.

اما چگونه سوکت های جریان می توانند به چنین کیفیتی در انتقال اطلاعات دست پیدا کنند؟ دلیل این کیفیت استفاده سوکت های جریان از پروتکلی به نام پروتکل نظارت بر انتقال(TCP[1]) است. این پروتوکل اطمینان حاصل می کند که اطلاعات شما به صورت پی در پی و بدون خطا به مقصد برسد. حتما با دیدن TCP به یاد چیز دیگری هم افتاده اید: نیمه دیگری که همیشه با آن به کار می رود. نام این نیمه گمشده IP (Internet Protocol[2]) یا پروتکل اینترنت است. این پروتکل برای مسیر یابی به کار می رود و وظیفه‌ای در قبال بی نقصی در رسیدن داده‌ها بر عهده ندارد.

سوکت های DataGram: سوالی که ممکن است در ذهن شما در مورد این نوع سوکت‌ها پدید آمده باشد این است که چرا به این نوع سوکت‌ها فاقد اتصال(Connectionless) می گویند؟ چرا آنها قابلیت اطمینان پایین تری نسبت به سوکت های جریان دارند؟ خوب دلایل این امور بدین شرح است: اگر شما یک DataGram[3] را ارسال نمایید،ممکن است به مقصد برسد یا نرسد. این احتمال هم وجود دارد که به صورت نامنظم به مقصد برسد. البته اگر بسته به مقصد برسد از خطا مبرا خواهد بود. سوکت های DataGram برای مسیر یابی از IP استفاده می‌نمایند اما برای نظارت بر انتقال اطلاعات خود از TCP استفاده نمی کنند. بلکه به جای استفاده از TCP از پروتکل دیگری به نام UDP[4] (User DataGram Protocol) استفاده می کنند.

اما چرا آنها فاقد اتصال هستند؟ اساسا این بدان دلیل است که شما در این نوع سوکت‌ها مانند سوکت‌های جریان نیازی به پیگیری برقراری ارتباط بین مبدا و مقصد ندارید. شما فقط یک بسته (Packet) ایجاد می‌نمایید و اطلاعات مبدا و مقصد و داده های لازم را در آن قرار می‌دهید و آن را

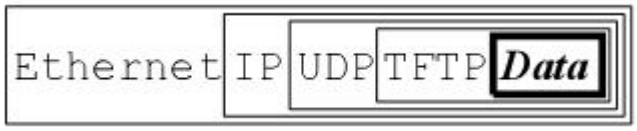
[1] براي اطلاعات بيشتر در زمينه TCP مي توانيد به RFC 793 مراجعه نماييد.
[2] براي اطلاعات بيشتر در مورد IP مي توانيد به RFC 791 مراجعه نماييد.
[3] بسته‌اي كه شامل آدرس مبدا و مقصد و داده‌ها است.
[4] براي اطلاعات بيشتر در مورد UDP مي توانيد به RFC 768 مراجعه نماييد.

ارسال می‌نمایید. این نوع ارتباطات معمولا برای ارسال بسته های اطلاعاتی کاربرد دارد. از برنامه‌هایی که از این نوع سوکت استفاده می کنند، tftp,bootp را می‌توان نام برد.

ممکن است از خود بپرسید با وجود این که بسته‌های DataGram می‌توانند گم شوند و نرسند این برنامه‌ها چگونه کار می‌کنند؟ خوب در پاسخ باید بگویم هر یک از این برنامه‌ها قراردادهای خود را دارند. به عنوان مثال قرارداد(Protocol) tftp می‌گوید برای هر بسته‌ای که فرستاده شد یک رسید باید بازگشت داده شود؛ مبنی بر این که طرف دیگر بسته را دریافت کرده است[5]. اگر پیغام "دریافت شد" در یک بازه زمانی خاص مثلا ۵ ثانیه نرسد، فرستنده تا زمانی که رسید را دریافت نکرده بسته را دوباره ارسال می‌کند. این نوع آگاهی از رسید بسته‌ها در موقع کار با SOCK_DGRAMها بسیار حائز اهمیت است.

ریزه کاری‌های سطح پایین و تئوری شبکه:

تا بدین مرحله از کار فقط درباره قراردادها (Protocols) بحث کردیم. حال زمان آن فرا رسیده که طرز کار شبکه‌ها را با هم بررسی نماییم و چند مثال در مورد نحوه ساخت بسته های SOCK_DGRAM ارائه دهیم. در صورتی که قبلا واحدی را در مورد شبکه گذرانده‌اید یا اطلاعاتی در مورد نحوه کار شبکه‌ها دارید می‌توانید از این بخش صرفنظر نمایید اما خواندن آن را به کسانی که پیش زمینه زیادی ندارند پیشنهاد می‌کنم.



شکل (۱)

در این قسمت به کپسوله‌سازی[6] داده‌ها (Data Encapsulation) می‌پردازیم. مثالی که در این بخش ارائه می‌دهیم یک بسته برنامه tftp است. اساسا یک بسته، مجموعه‌ای است از داده‌ها. برای فرستادن داده‌ها باید آنها را به صورت بسته در آوریم .برای این که بتوانیم داده‌های خود را به بسته‌ای نظیر شکل(۱) تبدیل نماییم باید مراحلی را پشت سر بگذاریم. این مراحل به این شرح هستند: ۱- ابتدا بسته ایجاد می‌شود. ۲- بسته (داده‌ها) در یک هدر (سرآیند) و به ندرت در یک فوتر (پا صفحه) توسط

[5] به این نوع تاییدها اصطلاحا ACK(Aknowledge) مي گويند.
[6] كپسوله سازي يعني تلفيق داده هايي با داده هاي ديگر.

بالاترین پروتکل (در این مثال tftp) برای اولین بار قرار داده می‌شود. ۳- پس از این مرحله بار دیگر تمامی بسته (اعم از هدر و فوتر) دوباره توسط پروتکل بعدی بسته‌بندی می‌شود. (UDP) ۴- در این مرحله باز هم تمامی بسته در یک هدر دیگر (IP) قرار می‌گیرد و در آخر هم توسط پروتکل پایین‌ترین لایه (فیزیکی) بسته‌بندی می‌شود و سپس فرستاده می‌شود. زمانی که بسته به مقصد می‌رسد تمامی این مراحل به صورت بر عکس انجام می‌شوند تا بسته دوباره به دست برنامه در سمت گیرنده برسد. مراحل باز شدن بسته: ابتدا سخت افزار (لایه فیزیکی) هدر Ethernet را می‌گشاید و سپس هسته هدر IP و UDP را می‌گشاید و در نهایت برنامه tftp سمت گیرنده هدر tftp را باز می‌کند و داده‌های موجود در آن را دریافت می‌نماید.

حال که مراحل بسته‌بندی را با هم بررسی نمودیم بهتر است نگاهی هم به انواع لایه‌های موجود بیندازیم و عملکرد هر یک را به صورت مختصر بررسی نماییم. در این قسمت می‌خواهیم درباره مدل لایه‌ای شبکه صحبت کنیم. این مدل یک سیستم از عاملیت‌های شبکه را توصیف می‌نماید که نسبت به سایر مدل‌ها مزایای بسیاری دارد. به عنوان مثالی برای یکی از این مزایا می‌توان گفت شما می‌توانید با استفاده از این مدل لایه‌ای برنامهٔ کار با سوکتی را بنویسید که بدون توجه به نوع شبکه مثل: Ethernet ، AUI و.... با انواع مختلف شبکه‌ها به راحتی کار کند، بدون این که نگرانی درباره طریقه انتقال اطلاعات داشته باشید. چرا که برنامه‌های سطح پایین تر به این مطالب رسیدگی می‌کنند.
همان طور که می‌دانید یکی از مدل‌های پراستفاده، مدل ISO است که شبکه را به هفت لایه به صورت زیر تقسیم می‌کند:

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

در مدل فوق لایه فیزیکی همان سخت افزار (Ethrnet, Serial ,...) است و لایه کاربردی هم جایی است که کاربر به صورت مستقیم با آن در ارتباط است. مدل دیگری که برای تقسیم بندی شبکه وجود دارد مدل TCP/IP است که به صورت زیر می‌باشد:

- Application Layer (*telnet, ftp, etc.*)
- Host-to-Host Transport Layer (*TCP, UDP*)
- Internet Layer (*IP and routing*)
- Network Access Layer (*Ethernet, ATM, or whatever*)

مشاهده می‌نمایید که برای فرستاده شدن، بسته باید از مراحل گوناگونی عبور نماید. اما به لطف پیشرفت علم و تکنولوژی تمامی کاری که برای کار کردن با یک سوکت جریان باید انجام دهید فرستادن اطلاعات با تابع send() است و تمامی کاری که برای کار با سوکت های DataGram باید انجام دهید بسته بندی اطلاعات به روش خودتان و فرستادن آن با تابع sendto() است. هسته، اطلاعات لازم برای لایه انتقال و اینترنت را به صورت خودکار برای شما ایجاد می کند و اطلاعات لازم برای لایه دسترسی به شبکه هم توسط سخت افزار ایجاد می گردد.

در پایان این بخش توضیح کوتاه خود را با ذکر این نکته خاتمه می‌دهم که در مورد IP دانستن همین نکته بس که این پروتکل توسط مسیریاب‌ها مورد استفاده قرار می گیرد و در مسیریابی کاربرد دارد.

ساختارهای (struct) مورد استفاده و رسیدگی به داده ها:

خوب، حالا زمان صحبت درباره برنامه‌نویسی است. در این بخش انواع داده های مورد استفاده توسط رابط سوکت را پوشش خواهم داد.

برای شروع یک مورد بسیار ساده را بررسی می‌نماییم. توصیف کننده سوکت، توصیف کننده‌ای از نوع int است؛ بله یک متغیر int ساده.

ممکن است از این قسمت به بعد مطالب کمی پیچیده شوند. بنابراین بهتر است مطالب را به ترتیب نوشته‌شده بخوانید و از آنها صرفنظر ننمایید. همان طور که می‌دانید دو نوع ترتیب در قرار گیری بایتها در یک عدد داریم : ۱- بایت پر اهمیت در ابتدا باشد و ۲- بایت کم اهمیت در ابتدا باشد. به روش اول Network Byte Order[۷] می گویند. برخی از ماشین‌ها اعداد خود را در این قالب ذخیره

[7] به این روش اصطلاحا Big Endian نیز مي گويند که توسط ريز پردازنده هاي شرکت موتورولا استفاده مي شود. در اين روش يك عدد مبناي شانزده مثل A0B2 به صورت A02B ذخيره مي شود(بايت با ارزش در همان ابتدا باقي مي ماند). البته اين روش در ماشين هاي بزرگ هم کاربرد دارد. (Microsoft Computer Dictionary). – م -

می‌نمایند و برخی نیز از این روش استفاده نمی‌کنند. به روش دوم اصطلاحا Host Byte Order[۸] می‌گویند. وقتی می‌گوییم چیزی باید به صورت Network Byte Order باشد، باید تابعی را فراخوانی نمایید تا آن را از حالت Host Byte Order به حالت Network Byte Order در آورد، اما زمانی که چیزی از Network Byte Order به میان نیاوردیم شما نیز ملزم به انجام کار خاصی نیستید.

خوب به سراغ بررسی ساختارهای مورد استفاده در سوکت نویسی می‌رویم. اولین ساختاری که بررسی می‌کنیم struct sockaddr است. این ساختار اطلاعات آدرس سوکت را برای انواع مختلفی از سوکتها نگه‌داری می‌کند:

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

sa_family: می‌تواند شامل موارد گوناگونی باشد، اما در این مقاله فقط از AF_INET استفاده می‌شود.

sa_data: شامل شماره پورت و آدرس مقصد برای سوکت است.

برای کار با ساختار sockaddr برنامه‌نویسان یک ساختار همسان آن تولید کرده اند: sockaddr_in (in برای Internet) که به صورت زیر است:

```
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
};
```

ساختار فوق ارجاع به المان‌های سوکت را بسیار ساده کرده است. نکته‌ای که در اینجا باید به آن توجه کنید این است که فیلد sin_zero[] حتما باید توسط تابع memset() با صفر پر شود.در ضمن این نکته را هم به یاد داشته باشید که یک اشاره‌گر به ساختار sickaddr_in می‌تواند به یک اشاره‌گر از نوع sockaddr تبدیل شود و بالعکس. بنابراین اگر در جایی یک آرگومان از نوع sockaddr* مورد نیاز بود می‌توان از یک اشاره‌گر به ساختار sockaddr_in هم استفاده کرد. در ضمن به یاد داشته باشید sin_family در ساختار sockaddr_in ، مطابق با sa_family در

[8] به این روش اصطلاحا Little Endian مي گويند. اين روش توسط پردازنده هاي Intel مورد استفاده قرار مي گيرد. و در آن بايت با ارزش عدد در انتهاي آن قرار مي گيرد مثلا عددي مثل A02B به صورت 2BA0 ذخيره مي‌شود. (Microsoft Computer Dictionary). – م -

ساختار sockaddr است و باید به AF_INTE ست شود. در نهایت این نکته را هم به یاد داشته باشید که sin_port و sin_addr باید به صورت Network Byte Order باشند.

تبدیل‌های محلی:

در این قسمت می‌خواهیم به طریقه تبدیل بایت‌ها از Host Byte Order به Network Byte Order و بالعکس بپردازیم. دو نوع داده وجود دارد که می‌توانید آنها را تبدیل نمایید : short(دو بایتی) و long (چهار بایتی). توابع مذکور در این قسمت روی انواع بدون علامت (unsigned) عمل می‌کنند. حال تصور کنید می‌خواهیم یک نوع short را از Host Byte Order به Network Byte Order تبدیل کنیم. با حرف h به نشانه میزبان (Host) شروع می‌کنیم. به دنبال آن to و سپس حرف n به نشانه شبکه(Network) و s را به نشانه short می‌آوریم. نتیجه می‌شود: h-to-n-s یا htons() بخوانید Host to Network Short. همان طور که مشاهده می‌نمایید بسیار ساده است. شما می‌توانید از هر ترکیب با معنی از حروف n,h,s,l استفاده نمایید، همان طور که گفتم ترکیب بامعنی مثلا چیزی به نام stoln()(Short to Long Host) نداریم. ترکیبات بامعنی به شرح زیر هستند:

- `htons()` – "Host to Network Short"
- `htonl()` – "Host to Network Long"
- `ntohs()` – "Network to Host Short"
- `ntohl()` – "Network to Host Long"

در نهایت به این سوال پاسخ می‌دهم که: چرا sin_addr و sin_port باید به صورت Network Byte Order باشند، اما sin_family احتیاجی به این کار ندارد؟ پاسخ: فیلدهای sin_addr و sin_port در لایه‌های IP[9] و UDP[10] کپسوله‌سازی می‌شوند. بنابراین باید به صورت Network Byte Order باشند. چرا که در شبکه(خارج از ماشین محلی) مورد استفاده قرار می‌گیرند. اما sin_family فقط توسط هسته(Kernel) مورد استفاده قرار می‌گیرد تا بداند که ساختار شامل چه نوع آدرسی است. بنابراین باید به صورت Host Byte Order باشد. در ضمن فیلد sin_family به بیرون از ماشین محلی(روی شبکه) منتقل نمی‌شود.

[9] منظور همان لایه Internet در مدل TCP/IP است. – م -
[10] لایه Host to Host در مدل TCP/IP. – م -

آدرس های IP و طریقه کار با آنها:

خوشبختانه یک دسته از توابع برای کار کردن با آدرس‌های IP وجود دارند و احتیاجی نیست شما عملیات روی آدرس‌ها را به صورت دستی انجام دهید(با عملگر شیفت).

در ابتدا فرض کنید شما یک ساختار از نوع sockaddr_in به نام ina دارید و می‌خواهید یک آدرس IP مانند "10.12.110.57" را در ساختار خود ذخیره نمایید. تابعی که برای انجام این عمل استفاده می‌شود inet_addr() نام دارد که یک آدرس IP را که مجموعه‌ای از اعداد و نقطه است، به صورت یک آرایه کراکتری دریافت می‌کند و آن را به یک عدد بدون علامت از نوع long تبدیل می‌کند. عمل انتساب فیلد ساختار می‌تواند به شکل زیر انجام گیرد:

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

به یاد داشته باشید تابع inet_addr() آدرس را به صورت Network Byte Order باز می‌گرداند و شما احتیاجی به فراخوانی تابع htonl() ندارید. عمل انتساب بالا از نظر کد نویسی دچار اشکالاتی است؛ چرا که قبل از انتساب مقدار بازگشتی تابع inet_addr() آن را برای خطاها بررسی نمی‌نماید. تابع inet_addr() در صورت مواجه با خطا مقدار ۱- را باز می‌گرداند، اعداد باینری را به خاطر می‌آورید نتیجه تبدیل 1- (unsigned) این مقدار مطابق آدرس IP "255.255.255.255" است بله آدرس Broadcast!! به یاد داشته باشید که قبل از انتساب حتما خطاها را بررسی نمایید.

راه دیگری که می‌توانید برای تبدیل آدرس های رشته‌ای به آدرس های عددی از آن استفاده نمایید و نسبت به تابع inet_addr() عملکرد بهتری دارد استفاده از تابع inet_aton() است. (aton مخفف Ascii To Network است):

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in_addr
*inp);
```

مثالی از به کار گیری تابع inet_aton() به شرح زیر است:

```
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
```

تابع فوق بر خلاف دیگر توابع، هنگام کار با سوکت‌ها، در صورت موفقیت مقدار غیر صفر و در صورت شکست مقدار صفر را باز می‌گرداند. متاسفانه برخی از سکو های کاری از این تابع پشتیبانی نمی‌نمایند. بنابراین در این مقاله از inet_addr() استفاده شده است. اما من شخصا استفاده از تابع inet_aton() را به شما توصیه می‌نمایم.

خوب تا اینجا یاد گرفتید آدرسهای رشته‌ای IP را که شامل نقطه است به نمایش دودویی در آورید. اما اگر یک ساختار از نوع ockaddr_in داشته باشید و بخواهید اعداد موجود در آن را به صورت نمایش با نقطه در آورید باید چه کاری انجام داد؟ خوب من به شما می‌گویم. در اینجا باید از تابع inet_ntoa() استفاده نمایید، همان طور که ممکن است حدس زده باشید ntoa مخفف Network To Ascii است به عنوان مثالی در این زمینه به کد زیر توجه نمایید:

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

عمل فوق باعث چاپ شدن آدرس IP می‌شود. توجه نمایید که تابع inet_ntoa() یک ساختمان از نوع struct in_addr را به عنوان آرگومان دریافت می‌کند نه یک عدد long را. هم چنین به خاطر داشته باشید این تابع یک اشاره‌گر به کراکتر را باز می‌گرداند.

پس از استخراج آدرس موجود و تبدیل آن به نمایش کراکتری، اگر احتیاج به ذخیره آدرس داشتید می‌توانید از تابع strcpy() استفاده نمایید.

فراخوان های سیستمی(System Calls):

در این قسمت به بررسی فراخوان‌های سیستمی که ما را قادر می‌سازند از عاملیات شبکه در یونیکس استفاده کنیم می‌پردازیم. زمانی که شما یکی از این توابع را فراخوانی می‌نمایید هسته عملیات را در دست گرفته و بخش اعظمی از آن را به صورت خودکار برای شما انجام می‌دهد. جایی که اکثر مردم در آن گیر می‌کنند ترتیب فرا خوانی این توابع است؛ چرا که در این مورد صفحات راهنما(man page) نمی‌تواند به آنها کمکی بکند. به همین خاطر در اینجا تمام تلاش خود را بر این امر معطوف داشته‌ام که توابع را به همان ترتیبی که فراخوانی می‌شوند به شما معرفی نمایم.

تابع socket() : دریافت توصیف‌کننده فایل:

سرفایل‌های مورد نیاز و شکل کلی این تابع به صورت زیر است:

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

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

اما این آرگومان‌ها چه چیزهایی هستند؟ در ابتدا آرگومان domain باید مانند ساختمان sockaddr_in با AF_INET مقدار دهی شود. در وحله بعدی آرگومان type قرار دارد. این آرگومان به هسته می‌گوید از چه نوع سوکتی می‌خواهیم استفاده کنیم که می‌تواند SOCK_STREAM و SOCK_DGRAM باشد. (توجه نمایید که آرگومان‌های domain و type می‌توانند مقادیر بسیار دیگری را نیز بپذیرند که من از ذکر آنها در اینجا خودداری نموده ام. (برای اطلاعات بیشتر می‌توانید به صفحات راهنمای تابع socket() مراجعه نمایید). در نهایت آرگومان protocol را با صفر مقداردهی نمایید تا به socket() اجازه دهید پروتکل صحیح را بر اساس type انتخاب نماید(البته راه بهتری هم برای گرفتن protocol وجود دارد (می‌توانید برای اطلاعات بیشتر صفحات راهنمای تابع getprotobyname() را مشاهده نمایید).

تابع سوکت یک توصیف‌کننده سوکت را باز می‌گرداند که می‌توانید از آن در فراخوان‌های سیستمی بعدی استفاده نمایید، این تابع در صورت مواجهه با خطا، مقدار ۱- را باز می‌گرداند و مقدار متغیر عمومی errno را هم به مقدار خطا، ست می‌نماید(به صفحات راهنمای تابع perror() مراجعه نمایید).

تابع bind(): من روی چه پورتی هستم؟

زمانی که شما یک سوکت دارید ممکن است مجبور شوید آن سوکت را با یک پورت روی ماشین محلی خود مرتبط سازید. شماره پورت توسط هسته برای ایجاد تطابق میان یک بسته(بسته رسیده) با یک توصیف‌کننده سوکت مربوط به یک پردازش مورد استفاده قرار می‌گیرد(در زمانی که از listen() استفاده می‌نمایید)، اما اگر فقط قصد استفاده از connect() را دارید انجام این کار غیر ضروری است.

سرفایل های مورد نیاز و شکل کلی تابع bind() به صورت زیر است:

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

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

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

sockfd توصیف کننده فایل سوکت است که توسط socket() باز گردانده شده است. myaddr یک اشاره گر به ساختار sockaddr است که اطلاعاتی درباره آدرس IP، شماره پورت و... را در بر دارد. مقدار addrlen هم باید برابر با طول ساختمان myaddr بر حسب بایت باشد. بنابراین می‌توانید آن را برابر با sizeof(struct sockaddr) قرار دهید.

مثالی از کاربرد bind():

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

نکته قابل توجه در مثال فوق مقدار myaddr.sin_port است که باید به صورت Network Byte Order باشد. نکته قابل توجه دیگر فایل‌های سرآیند(Header) هستند که ممکن است از

سیستمی به سیستم دیگر متفاوت باشند. به همین علت بهتر است برای اطمینان صفحات راهنمای سیستم خودتان را چک نمایید.

آخرین موضوعی که در ارتباط با تابع bind() باید متذکر شوم این است که اعمالی از قبیل فهمیدن آدرس IP خودتان و شماره پورت می‌توانند به صورت خودکار انجام پذیرند به مثال زیر توجه نمایید:

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

همان طور که در مثال فوق مشاهده نمودید، با ست کردن مقدار فیلد myaddr.sin_port به مقدار صفر، در واقع به تابع bind() اعلام می‌کنید شماره پورت را خودش برای شما انتخاب نماید. به همین ترتیب با ست کردن فیلد myaddr.sin_addr.s_addr با ثابت INADDR_ANY به تابع BIND() اعلام می‌کنید مقدار این فیلد را با آدرس IP ماشینی که برنامه در آن در حال اجرا است پر کند.

اگر کمی با دقت به مثال بالا توجه کرده باشید حتما متوجه این نکته شده‌اید که بنده مقادیر فوق را به صورت Network Byte Order تبدیل نکرده ام، دلیل این امر در این است که در واقع مقدار ثابت INADDR_ANY صفر است و مقدار صفر حتی در صورت دوباره چیده شدن بیت‌ها باز هم صفر باقی می‌ماند. با این حال در برخی از ماشین‌ها مقدار مذکور برای ثابت INADDR_ANY مقدار ۱۲ است و کد فوق به درستی کار نخواهد کرد. بنابراین برای این که کد شما قابلیت حمل داشته باشد و بتواند در هر ماشینی به راحتی اجرا شود بهتر است کد فوق را به صورت زیر بنویسید:

```
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
```

تابع bind() در صورت عدم موفقیت، مقدار ۱- را باز می‌گرداند و مقدار متغیر errno را نیز به مقدار خطا ست می‌نماید.

نکته دیگری که باید در این قسمت متذکر شوم شماره پورت هایی است که استفاده می‌نمایید؛ چرا که شماره های ۱۰۲۴ به پایین تماما رزرو شده هستند(مگر این که superuser باشید). شما می‌توانید

هر شماره دیگری را بالای این عدد استفاده نمایید تا حداکثر ۶۵۵۳۵ (البته در صورتی که توسط برنامه‌های دیگر در حال استفاده نباشد).

تابع connect():

در این قسمت به بررسی تابع connect() می‌پردازیم. از این تابع زمانی استفاده می‌شود که بخواهیم به یک میزبان راه دور متصل شویم. سر فایل‌های مورد نیاز و شکل کلی این تابع به صورت زیر است:

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

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

sockfd همان توصیف‌کننده فایل سوکت است. serv_addr یک ساختار از نوع sockaddr است که شامل آدرس IP و پورت میزبان راه دور است و addrlen هم سایز ساختار sockaddr بر حسب بایت است. بنابراین می‌توانید آن را با sizeof(struct sockaddr) مقداردهی نمایید. به مثال زیر توجه نمایید:

```
#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));
.
.
.
```

بهتر است مقدار بازگشتی تابع connect() را برای بررسی خطاها چک نمایید. این تابع نیز در صورت مواجهه با شکست مقدار ۱- را باز می‌گرداند و مقدار متغیر errno را نیز متعاقبا مقداردهی می‌نماید.

در مثال بالا به این نکته توجه داشته باشید که تابع bind() فرا خوانی نشده است؛ چراکه اساسا شماره پورت کامپیوتر محلی در اینجا اهمیتی ندارد و نکته حائز اهمیت آدرس میزبان راه دور است، در چنین مواقعی هسته به صورت خودکار یک شماره پورت برای ما انتخاب می‌نماید و کامپیوتر مقصد به صورت اتوماتیک این اطلاعات را دریافت خواهد کرد.

تابع listen():

خوب زمان این است که بحث را کمی تغییر دهیم. اگر نخواهیم به یک میزبان راه دور متصل شویم چه باید بکنیم؟ برای مثال ممکن است بخواهید برای یک یا چند درخواست(ارتباط) منتظر بمانید و پس از دریافت ارتباط آنها را به شکل خاصی اداره نمایید. خوب انجام چنین اعمالی دو مرحله دارد: مرحله اول گوش دادن listen() برای دریافت ارتباطات است و مرحله دوم پذیرش accept() ارتباطات.

تابع listen() بسیار ساده است و احتیاج به توضیح زیادی ندارد، شکل کلی این تابع به صورت زیر است:

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

sockfd یک توصیف‌کننده فایل سوکت است که توسط تابع socket() باز گردانده می‌شود، backlog هم تعداد ارتباطاتی است که می‌توانند در صف ارتباطات در انتظار باشند. اما صف

ارتباطات به چه معنی است؟ خوب در واقع این صف است که ارتباطات (connection) در آن برای رسیدگی به درخواستشان منتظر می‌مانند تا زمانی که مورد پذیرش accept() قرار بگیرند و عدد backlog تعداد ارتباطاتی را مشخص می‌کند که می‌توانند در این صف قرار بگیرند.

طبق معمول تابع listen() در صورت مواجهه با شکست، مقدار ۱- را باز می‌گرداند و مقدار errno هم متعاقبا مقداردهی می‌شود.

خوب همان طور که ممکن است تا به حال حدس زده باشید قبل از فراخوانی تابع listen() باید تابع bind() را فراخوانی نماییم . در غیر این صورت هسته یک پورت تصادفی را برای گوش دادن انتخاب می‌نماید. بنابراین اگر مایل به گوش دادن به یک پورت خاص برای دریافت ارتباطات هستید ترتیب فراخوانی توابعی که تا به حال ذکر کرده ایم به صورت زیر است:

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

تابع accept():

ممکن است از خود بپرسید چه زمانی باید این تابع را فراخوانی کرد؟ چه چیزی باید اتفاق بیفتد؟ خوب پاسخ شما اینجاست: فرض کنید شخصی از مسافتی بسیار دور سعی در connect() شدن به پورتی دارد که شما در حال listen() کردن به آن هستید. در ابتدا آن شخص در یک صف قرار می‌گیرد و منتظر می‌ماند. سپس مورد پذیرش accept() قرار می‌گیرد، تابع accept() یک توصیف‌کننده فایل سوکت جدید را برای شما باز می‌گرداند تا به این ارتباط جدید از طریق آن رسیدگی نمایید. درست است شما دو توصیف‌کننده سوکت برای یک ارتباط دارید، اولی هنوز در حال گوش دادن به پورت مورد نظر شما است و دومی آماده ارسال send() و دریافت recv() اطلاعات.

سرفایل‌های مورد نیاز و شکل کلی تابع accept() به صورت زیر است:

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

int accept(int sockfd, void *addr, int *addrlen);
```

sockfd همان توصیف‌کننده سوکتی است که در حال گوش دادن listen() به آن هستید، addr یک اشاره‌گر است به ساختاری از نوع sockaddr_in. این ساختار اطلاعات را درباره طرف

دیگر ارتباط در خود نگه داری می‌نماید، addrlen هم طول ساختار addr است و می‌تواند به صورت sizeof(struct sockaddr_in) مقداردهی شود.

این تابع نیز در صورت مواجهه با شکست، مقدار ۱- را باز می‌گرداند و مقدار errno نیز متعاقبا به مقدار خطا ست خواهد شد.

به مثالی در مورد تابع accept() توجه نمایید:

```
#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);
.
.
.
```

توجه داشته باشید که از توصیف‌کننده new_fd برای تمامی فراخوانهای send() و recv() استفاده خواهیم کرد، بنابراین اگر شما فقط قصد دریافت یک ارتباط را دارید می‌توانید sockfd را برای جلوگیری از تعداد بیشتری ارتباط close() نمایید[11].

توابع send() و recv():

این توابع برای مخابره کردن[12] از طریق سوکت‌های جریان و یا سوکت‌های DataGram متصل شده (Connected) به کار می‌روند. اگر مایل به استفاده از سوکت‌های DtaGram غیر متصل هستید باید از توابع sendto() و recvfrom() استفاده نمایید که در ادامه مقاله مورد بررسی قرار می‌گیرند.

شکل کلی تابع send() به صورت زیر است:

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

sockfd توصیف‌کننده سوکتی است که مایل به فرستادن اطلاعات از طریق آن هستید(که یا توسط accept() و یا توسط socket() باز گردانده شده است). msg اشاره‌گر به اطلاعاتی است که می‌خواهید بفرستید و len طول اطلاعات شما بر حسب بایت است، آرگومان flag را هم به مقدار صفر ست نمایید (برای اطلاعات بیشتر در مورد این آرگومان به صفحات راهنمای تابع send() مراجعه نمایید).

به مثالی از تابع send() توجه نمایید:

```
char *msg = "Beej was here!";
int len, bytes_sent;
.
.
```

[11] منظور نویسنده این است که در حالي که شما از new_fd براي ارسال و دريافت داده‌ها استفاده مي نماييد sockfd هنوز در حال گوش دادن به پورت براي دريافت ارتباطات جديد است. بنابراين در صورتي كه مايل به دريافت و پذيرش ارتباطات ديگري نيستيد بهتر است sockfd را مسدود ((close) نماييد. – م -

[12] منظور از مخابره کردن (communicate) همان ارسال و دریافت داده‌ها است.

```
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
.
.
.
```

تابع send() در صورت موفقیت تعداد بایت‌هایی را که موفق به فرستادن آنها شده است باز می‌گرداند، این مقدار ممکن است کمتر از مقداری باشد که شما برای فرستادن به آن داده اید؛ چراکه ممکن است در برخی مواقع مقدار اطلاعاتی که به تابع می‌دهید برای آن قابل اداره و در دست گیری نباشد. در چنین مواقعی تابع حداکثر مقداری را که می‌تواند می‌فرستد و ادامه روند فرستادن داده‌ها را به شما محول می‌نماید. خبر خوب در اینجا این است که اگر اطلاعات شما نسبتا کم باشد (تا ۱ کیلوبایت) تابع send() به احتمال زیاد با یکبار فرستادن، آن را ارسال خواهد نمود.

این تابع نیز در صورت مواجهه با خطا مقدار ۱- را باز می‌گرداند و مقدار errno را نیز به مقدار خطا ست می‌نماید.

تابع recv() از بسیاری از جوانب شبیه به تابع send() است. شکل کلی این تابع به صورت زیر است:

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

sockfd توصیف‌کننده سوکتی است که اطلاعات از طریق آن دریافت می‌شوند. buf یک بافر است که اطلاعات دریافت شده در آن قرار می‌گیرند، len هم حداکثر سایز بافر است، flag هم باید به صفر ست شود (برای اطلاعات بیشتر در مورد آرگومان flag می‌توانید از صفحات راهنمای تابع recv() استفاده نمایید). تابع recv() در صورت موفقیت تعداد بایت‌های دریافت شده را باز می‌گرداند و در صورت مواجهه با خطا مقدار ۱- را باز می‌گرداند. در چنین مواقعی مقدار متغیر errno به مقدار خطا ست خواهد شد.

صبر کنید. این تابع در شرایطی مقدار صفر را باز می‌گرداند و این فقط یک معنی می‌تواند داشته باشد: طرف مقابل ارتباط را قطع کرده است و این مقدار بازگشتی توسط تابع recv() برای آگاه کردن شما از این مسئله است.

توابع sendto() و recvfrom() (سوکت های Datagram):

در این قسمت توابع مربوط به سوکت‌های Datagram را مورد بررسی قرار می‌دهیم. خوب برای شروع حدس بزنید در مواقعی که از سوکت های Datagram استفاده می‌کنید و به یک میزبان راه

دور متصل نیستید باید قبل از ارسال چه بخشی از اطلاعات را به تابع بدهیم؟ درست است آدرس مقصد.

شکل کلی تابع sendto() به صورت زیر است:

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

همان‌طور که مشاهده می‌نمایید این تابع شبیه به send() است و فقط در دو آرگومان با send() متفاوت است. to یک اشاره‌گر به ساختمان sockaddr است که شامل آدرس IP و پورت کامپیوتر مقصد است، tolen هم می‌تواند به راحتی به صورت sizeof(struct sockaddr) مقداردهی شود. مانند تابع send()، این تابع نیز در صورت موفقیت تعداد بایت‌های فرستاده شده را باز می‌گرداند(که ممکن است کمتر از سایز مورد نظر شما باشد) و در صورت مواجهه با خطا مقدار ۱- را باز می‌گرداند.

حال به بررسی تابع recvfrom() می‌پردازیم. شکل کلی این تابع به صورت زیر است:

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

این تابع نیز شبیه به recv() است و فقط در تعدادی از آرگومان‌ها با recv() متفاوت است. from یک اشاره‌گر به ساختمانی از نوع sockaddr است که با آدرس IP و پورت ماشین مبدا پر می‌شود، fromlen یک اشاره‌گر به int است که می‌تواند به صورت sizeof(struct sockaddr) مقداردهی شود، زمانی که تابع کار خود را به اتمام می‌رساند fromlen شامل طول آدرسی است که در from ذخیره شده است.

این تابع نیز در صورت موفقیت تعداد بایت هایی را که دریافت کرده است باز می‌گرداند و در صورت مواجهه با خطا مقدار ۱- را باز می‌گرداند و مقدار متغیر errno را نیز به مقدار خطا ست می‌نماید.

به یاد داشته باشید اگر یک سوکت Datagram را connect() نمایید می‌توانید از توابع send() برای فرستادن و recv() برای دریافت داده‌های خود استفاده نمایید، در این حالت خود

سوکت هنوز یک سوکت Datagram است و بسته‌ها هنوز هم برای انتقال از UDP استفاده می‌نمایند اما رابط سوکت اطلاعات مبدا و مقصد را به صورت خودکار به بسته‌ها اضافه می‌نماید.[۱۳]

توابع close() و shutdown():

خوب شما تمامی طول روز را با توابع send() و recv() به فرستادن و دریافت اطلاعات سپری کرده‌اید و حال می‌خواهید کار خود را پایان دهید (ارتباط را قطع نمایید)، خوب انجام این کار بسیار ساده است می‌توانید انجام این کار را به توابع عادی کار با فایل‌ها در یونیکس واگذار نمایید:

```
close(sockfd);
```

فراخوانی تابع فوق سبب می‌شود از این نقطه به بعد از هرگونه خواندن و یا نوشتن بر روی sockfd جلوگیری به عمل آید و هر گونه سعی در هر دو طرف باعث بروز یک خطا خواهد شد، چرا که ارتباط قطع شده است.

اما ممکن است در مواقعی بخواهید کنترل بیشتری روی نحوه قطع شدن ارتباط داشته باشید و ارتباط یک طرف و یا در مواقعی هر دو طرف(مثل close()) را قطع نمایید. در چنین مواقعی می‌توانید از تابع shutdown() استفاده نمایید. شکل کلی این تابع به صورت زیر است:

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

در این قالب کلی sockfd توصیف‌کننده فایل سوکتی است که می‌خواهید shutdown() نمایید و how می‌تواند شامل یکی از موارد زیر باشد:

- `0` – Further receives are disallowed
- `1` – Further sends are disallowed
- `2` – Further sends and receives are disallowed (like `close()`)

0: باعث می‌شود دریافت اطلاعات غیرممکن شود.

۱: باعث می‌شود ارسال اطلاعات غیرممکن شود.

۲: باعث می‌شود هم دریافت و هم ارسال غیرممکن گردد(مانند close()).

[13] چراکه اگر شما سوکت را connect() ننمایید مجبور خواهید بود آدرس فرستنده و گیرنده را به صورت دستي به توابع sendto() و recvfrom() بدهید اما در این حالت این اطلاعات به صورت خودکار اضافه مي شوند. – م -

این تابع در صورت موفقیت مقدار صفر و در صورت شکست مقدار ۱- را باز می‌گرداند و مقدار متغیر errno هم به مقدار خطا ست می‌شود.

اگر شما قصد استفاده از shutdown() را روی سوکت های بدون ارتباط Datagram دارید به یاد داشته باشید این عمل سوکت را برای send() و یا recv()های بعدی غیر ممکن می‌سازد.

توجه به این نکته نیز ضروری است که shutdown() در حقیقت توصیف‌کننده، سوکت را نمی‌بندد و فقط نوع استفاده آن را تغییر می‌دهد و برای آزاد سازی یک توصیف‌کننده سوکت باید از close() استفاده نمایید.

تابع getpeername() :

این تابع به شما می‌گوید چه کسی در سر دیگر ارتباط قرار دارد. سر فایل‌های مورد نیاز و شکل کلی این تابع به صورت زیر است:

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

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

sockfd توصیف‌کننده سوکت مرتبط است، addr اشاره‌گری است به ساختاری از نوع sockaddr یا sockaddr_in که اطلاعاتی را درباره طرف دیگر ارتباط نگه‌داری می‌نماید و addrlen اشاره‌گری به int است که می‌تواند به صورت sizeof(struct sockaddr) مقداردهی شود.

تابع فوق در صورت مواجهه با خطا مقدار ۱- را باز می‌گرداند و مقدار errno نیز به مقدار خطا ست خواهد شد.

هنگامی‌که شما آدرس مقصد را داشته باشید می‌توانید با تابع inet_ntoa() و یا getpeername() اطلاعاتی را راجع به آن به دست آورید، البته با استفاده از این توابع شما نمی‌توانید نام ورودی(logi name) میزبان مقصد را به دست آورید[۱۴].

[14] براي اطلاعات بيشتر در زمينه و به دست آوردن نام ورودي (login name) به RFC 1413 مراجعه نماييد.

تابع gethostname() :

این تابع نام کامپیوتری را که برنامه در حال اجرا در آن است، باز می‌گرداند، پس از به دست آوردن این نام می‌توانید با استفاده از تابع gethostbyname() آدرس IP کامپیوتر خودتان را به دست آورید. فایل‌های سرآیند و شکل کلی تابع فوق به صورت زیر است:

```
#include <unistd.h>

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

hostname یک اشاره‌گر به کراکتر است که شامل نام میزبان خواهد بود و host هم اشاره‌گری است که تعداد کراکترهای نام میزبان را در خود دارد.
این تابع در صورت موفقیت مقدار صفر و در صورت عدم موفقیت مقدار ۱- را باز می‌گرداند و در صورت خطا مقدار errno نیز به مقدار خطا ست خواهد شد.

DNS (Domain Name Service):

خوب DNS همان طور که در بالا هم می‌توانید ملاحظه نمایید مخفف Domain Name Service یا سرویس نام دامنه است. به عنوان توضیحی مختصر درباره وصف DNS می‌توان گفت که شما یک نام قابل درک و فهم برای انسان (نام سایت) را به DNS می‌دهید و DNS آن را با آدرس IP تبدیل می‌نماید(سپس شما می‌توانید از bind()، connect() و... برای اتصال به آن سایت استفاده نمایید).
اما چگونه؟ خوب برای این کار شما باید از تابع gethostbyname() استفاده نمایید. سرفایل‌های مورد نیاز و شکل کلی این تابع به صورت زیر است:

```
#include <netdb.h>

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

همان طور که ملاحظه می‌نمایید این تابع یک اشاره‌گر به ساختاری از نوع hostent باز می‌گرداند. فیلدهای این ساختار به شرح زیر است:

```
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 این قسمت توضیح کوتاهی در مورد هر یک از فیلدهای این ساختار آورده‌ام:

h_name : نام میزبان.

h_aliases: آرایه‌ای شامل نام های مستعار برای میزبان که با کراکتر NULL خاتمه می‌یابد.

h_addrtype: نوع آدرسی که باز گردانده می‌شود. معمولا AF_INET.

h_length : طول آدرس بر حسب بایت.

h_addr_list: آرایه‌ای که با صفر به پایان می‌رسد و شامل آدرس های شبکه مربوط به میزبان است. آدرس‌ها در قالب Network Byte Order هستند.

h_addr: اولین آدرس موجود در h_addr_list.

تابع فوق در صورت موفقیت یک اشاره‌گر پر به ساختار hostent باز می‌گرداند و در صورت شکست مقدار NULL را باز می‌گرداند، البته در چنین مواقعی به جای مقدار errno متغیر دیگری به نام h_error با مقدار خطا ست می‌شود و برای چاپ خطا باید از herror() به جای perror() استفاده نمایید. به مثالی در زمینه استفاده از herror() توجه نمایید:

```
#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;
}
```

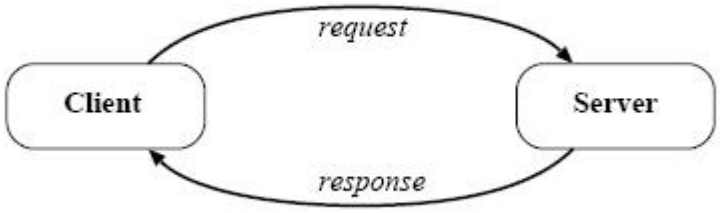
همان طور که گفته شد با تابع gethostbyname() نمی‌توانید از perror() برای چاپ خطاها استفاده نمایید(چرا که errno مورد استفاده قرار نگرفته است) و در عوض باید از herror() استفاده نمایید.

پس از استفاده از تابع gethostbyname() و دریافت اطلاعات ساده‌ترین راه برای چاپ آدرس IP موجود در فیلد h_addr ساختار hostent استفاده از inet_ntoa() است، اما تابع inet_ntoa() یک اشاره‌گر به ساختار in_addr را به عنوان آرگومان دریافت می‌نماید. بنابراین باید قبل از فرستادن این فیلد به تابع، آن را به اشاره‌گری از نوع ساختار in_addr تبدیل کرد، این نکته را می‌توانید در کد بالا هم مشاهده نمایید.

دورنمای مشتری-سرویس دهنده(Client-Server):

همه چیز در شبکه به نوعی در قالب سرویس‌دهنده و مشتری است. به عنوان مثال telnet را در نظر بگیرید. زمانی که شما به وسیله telnet به پورت ۲۳ یک میزبان راه دور متصل می‌شوید(مشتری) ، یک برنامه در آن میزبان به نام telnetd (سرویس‌دهنده) شروع به کار می‌کند و به ارتباطات دریافتی رسیدگی می‌نماید و یک عنوان (prompt) برای ورود در اختیار شما قرار می‌دهد.

تبادلات بین مشتری و سرویس‌دهنده در شکل ۲ به صورت خلاصه آورده شده است.



شکل(۲)

توجه نمایید که زوج مشتری-سرویس‌دهنده می‌توانند از طریق SOCK_STREAM و یا SOCK_DGRAM و یا هر چیز دیگری با هم در ارتباط باشند(البته از یک نوع). به عنوان مثال در زمینه برنامه‌های مشتری-سرویس‌دهنده می‌توان به: telnet/telnetd ، ftp/ftpd ، bootp/bootpd اشاره کرد.

اغلب فقط یک سرویس‌دهنده روی هر ماشین وجود دارد و این سرویس‌دهنده از طریق fork() کردن مشتری‌ها به چندین مشتری به صورت همزمان رسیدگی می‌نماید. به زبان ساده می‌توان گفت که یک سرویس‌دهنده برای ارتباط منتظر می‌ماند و پس از دریافت ارتباط آن را accept() کرده و یک پردازش فرزند را برای آن ارتباط fork() ایجاد می‌نماید تا بتواند به ارتباط رسیدگی کند. این اعمال کارهایی است که سرویس‌دهنده‌ای که در بخش بعدی ایجاد می‌کنیم انجام خواهد داد.

یک سرویس‌دهنده (server) ساده:

تمامی کاری که این سرویس‌دهنده انجام می‌دهد فرستادن رشته "Hello world\n" با استفاده از یک SOCK_STREAM است. تمامی کاری که شما برای امتحان کردن این سرویس‌دهنده باید انجام دهید اجرای آن در یک پنجره و اجرای telnet در پنجره‌ای دیگر و سپس telnet کردن به سرویس‌دهنده به صورت زیر است:

```
$ telnet remotehostname 3490
```

که در آن remotehost نام ماشینی است که درحال اجرای برنامه روی آن هستید.

کد برنامه به صورت زیر است(توجه کنید یک علامت \ در یک خط به معنای ادامه دار بودن آن در خطوط بعدی است):

```
/*
** 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,s
izeof(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;
}
```

یک مشتری ساده با استفاده از SOCK_STREAM :

این بخش، از بخش سرویس‌دهنده ساده تر است. تمامی کاری که این مشتری(سرویس گیرنده) انجام می‌دهد متصل شدن به سرویس‌دهنده‌ای است که شما در خط فرمان برای آن مشخص می‌نمایید. شماره پورت در این مثال ۳۴۹۰ است. پس از اتصال، مشتری رشته‌ای را که سرویس‌دهنده برای او می‌فرستد دریافت می‌نماید.

کد مشتری به شرح زیر است:

```
/*
** 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 یاد داشته باشید در صورتی که سرویس‌دهنده را قبل از مشتری اجرا ننمایید با خطای connection refused مواجه خواهید شد.

سوکتهای Datagram:

در این قسمت به ارائه یک مثال درباره سوکت های Datagram خواهم پرداخت. این مثال دارای دو برنامه است؛ اولی listener نام دارد که در یک ماشین مستقر می‌شود و منتظر دریافت یک بسته روی پورت ۴۹۵۰ می‌ماند و talker هم یک بسته را روی آن پورت می‌فرستد که شامل اطلاعاتی است که کاربر وارد کرده است.

کد منبع برای listener به صورت زیر است:

```
/*
** 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;
}
```

و کد منبع talker نیز به شرح زیر می‌باشد:

```
/*
** talker.c - a datagram "client" demo
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#define MYPORT 4950 // the port users will be
connecting to
int main(int argc, char *argv[])
{
int sockfd;
struct sockaddr_in their_addr; // connector’s
address information
struct hostent *he;
int numbytes;
if (argc != 3) {
fprintf(stderr,"usage: talker hostname
message\n");
exit(1);
}
if ((he=gethostbyname(argv[1])) == NULL) { // get
the host info
perror("gethostbyname");
exit(1);
}
if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) ==
-1) {
perror("socket");
exit(1);
```

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

تکنیک های نسبتا پیشرفته:

تکنیک هایی که از این بخش به بعد به کار می‌روند واقعا پیشرفته نیستند، بلکه خارج از بخش مقدماتی هستند و در واقع تا به همین جای کار هم شما بخش ابتدایی برنامه نویسی شبکه تحت یونیکس را پشت سر گذاشته‌اید، به شما تبریک می گویم.

بلوکه کردن[۱۵]:

در واقع Block یک معنای مجازی یا یک کنایه از خوابیدن (Sleep) است. ممکن است توجه کرده باشید زمانی که listener را اجرا می‌نمایید تا زمان دریافت یک بسته در انتظار باقی می‌ماند چراکه recvfrom() را فراخوانی می‌نماید و چون چیزی برای دریافت وجود ندارد می گوییم recvfrom() بلوکه شده است(در انتظار است)، تا زمانی که اطلاعاتی برسد. بسیاری از توابع به این صورت عمل می کنند. به عنوان مثال تابع accept() تمامی‌انواع recv() و.... دلیل بلوکه شدن این توابع این است که آنها اجازه این کار را دارا هستند. زمانی که شما یک توصیف کننده سوکت ایجاد

[15] Blocking (بلوکه کردن) در این متن همان طور که در ادامه گفته شده، به معني خوابيدن (Sleep شدن) پردازه‌ها است.

می‌نمایید هسته، آن را بلوکه می‌نماید. اگر نمی‌خواهید یک سوکت بلوکه شود باید از تابع fcntl() استفاده نمایید. سرفایل‌های مورد نیاز و شکل کلی این تابع به صورت زیر است:

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

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

با تغییر خصوصیت یک سوکت به non-blocking شما می‌توانید به صورت موثری از اطلاعات سوکت‌ها نمونه‌برداری نمایید. اگر شما سعی کنید یک سوکت non-blocking را بخوانید و اطلاعاتی برای خواندن موجود نباشد سوکت توانایی بلاک شدن را ندارد و مقدار ۱- باز گردانده خواهد شد و مقدار errno به EWOULDBLOCK ست خواهد شد.

البته نمونه برداری از سوکت‌ها با چنین روشی ایده خیلی خوبی نیست؛ چرا که اگر بخواهید در یک صف شلوغ به دنبال سوکتی که آماده خواندن است بگردید وقت CPU را هدر داده اید. یک راه حل دیگر هم برای انجام چنین اعمالی وجود دارد و آن هم استفاده از تابع select() است که در قسمت بعدی به آن پرداخته‌ایم.

select() ورودی و خروجی ترکیبی همزمان:

تابع select() توانایی نظارت کردن بر چندین سوکت را در یک زمان واحد به شما می‌دهد. این تابع به شما اطلاع می‌دهد کدام یک از آنها آماده خواندن و کدام یک آماده نوشتن است و کدام یک باعث ایجاد خطا شده است.

سرفایل‌های مورد نیاز و شکل کلی این تابع به صورت زیر است:

```
#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);
```

این تابع مجموعه‌ای از توصیف‌کننده‌ها را مورد بررسی قرار می‌دهد. در واقع readfds، writefds، و exceptfds. به عنوان مثال اگر می‌خواهید بدانید که آیا می‌توانید از ورودی استاندارد (stdin) و یک سوکت مثلا : sockfd اطلاعات را بخوانید، فقط کافی است مقدار صفر

(را برای توصیف‌کننده فایل) و توصیف‌کننده سوکت را به مجموعه readfds اضافه نمایید، مقدار numfds باید برابر با شماره بالاترین توصیف‌کننده + ۱ باشد که در اینجا برابر با sockfd+1 است؛ چرا که مطمئنا از صفر بزرگتر است.

پس از این که select() به کار خود خاتمه داد مقدار readfs متفاوت خواهد بود تا تابع بتواند سوکت‌هایی را که برای خواندن آماده هستند نشان دهد. سپس شما می‌توانید با استفاده از ماکروی FD_ISSETU() آنها[۱۶] را امتحان نمایید.

قبل از ادامه بحث می‌خواهم درباره دستکاری مجموعه‌های فوق صحبت کنم. هر کدام از این سه مجموعه از نوع fd_set هستند. ماکروهایی که در ادامه آمده‌اند برای دستکاری و تغییر این مجموعه‌ها به کار می‌روند:

FD_ZERO(fd_set *set) : یک مجموعه از توصیف‌کننده‌ها را پاک می‌کند.

FD_SET(int fd, fd_set *set) : توصیف‌کننده fd را به مجموعه set اضافه می‌نماید.

FD_CLR(int fd, fd_set *set) : توصیف‌کننده fd را از مجموعه set پاک می‌کند.

FD_ISSET(int fd, fd_set *set) : امتحان می‌کند تا ببیند آیا fd در set وجود دارد یا خیر.

تنها نکته پیچیده این است که ساختار timeval چیست؟ خوب در برخی از مواقع شما نمی‌خواهید تا ابد منتظر بمانید تا شخصی چیزی برای شما بفرستد. این ساختار شما را قادر می‌سازد تا یک بازه زمانی برای انتظار مشخص نمایید، در این حالت اگر زمان انتظار از زمان معین شده فراتر رود و تابع select() هیچ توصیف‌کننده آماده‌ای را پیدا نکرده باشد به کار خود خاتمه می‌دهد و شما می‌توانید ادامه روند پردازش را از سر بگیرید.

فیلدهای این ساختار به صورت زیر هستند:

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

فیلد tv_sec را به تعداد ثانیه‌های لازم برای انتظار و فیلد tv_usec را به تعداد میکروثانیه‌های لازم برای انتظار مقدار دهی نمایید. در ضمن زمانی که تابع به کار خود خاتمه دهد ممکن است مقدار

[16] منظور مجموعه هاي readfds,writefds,exceptfds است. – م -

آرگومان timeout را تغییر داده باشد تا نشان دهد هنوز زمانی باقی مانده است. این به نوع یونیکسی که در حال استفاده از آن هستید بستگی دارد.

نکته جالب دیگری که وجود دارد این است که اگر شما فیلدهای ساختار timeval خود را با 0 مقداردهی نمایید تابع select() به سرعت به تمامی مجموعه‌ها سرکشی کرده و کار خود را به اتمام می‌رساند. و اگر مقدار آرگومان timeout را به NULL ست کنید تابع select() تا آماده شدن اولین توصیف‌کننده منتظر می‌ماند.

قطعه کد زیر مدت ۲,۵ ثانیه برای ورودی استاندارد منتظر می‌ماند:

```
/** select.c - a select() demo
*/
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#define STDIN 0 // file descriptor for standard
input
int main(void)
{
struct timeval tv;
fd_set readfds;
tv.tv_sec = 2;
tv.tv_usec = 500000;
FD_ZERO(&readfds);
FD_SET(STDIN, &readfds);
// don't care about writefds and exceptfds:
select(STDIN+1, &readfds, NULL, NULL, &tv);
if (FD_ISSET(STDIN, &readfds))
printf("A key was pressed!\n");
else
printf("Timed out.\n");
return 0;
}
```

اگر از یک ترمینال با بافر خطی استفاده می‌نمایید باید کلید RETURN را فشار دهید. حال ممکن است بگویید انتظار برای دریافت اطلاعات از یک سوکت Datagram راه خوبی است. پاسخ شما درست است، بله می‌تواند باشد، چرا که برخی از انواع یونیکس‌ها اجازه استفاده از select() را با این روش می‌دهند و برخی خیر. بنابراین شما باید صفحات راهنمای مربوط به یونیکس خودتان را چک نمایید.

برخی از یونیکس‌ها زمان موجود در ساختار timeout را برای نشان دادن باقیمانده زمان تغییر می‌دهند و برخی این کار را انجام نمی‌دهند. اگر می‌خواهید برنامه شما قابلیت حمل داشته باشد باید از این نکات صرفنظر نمایید(برای مثال برای دانستن زمان سپری شده می‌توانید از gettimeofday() استفاده نمایید).

چه اتفاقی می‌افتد اگر یک سوکت که در مجموعه خواندن قرار دارد ارتباط را ببندد؟ در این حالت تابع select() سوکت را در لیست آماده خواندن قرار می‌دهد، و سپس زمانی که شما اطلاعات را از آن دریافت می‌نمایید recv() این تابع مقدار صفر را به شما باز می‌گرداند. این راهی است که در چنین مواقعی می‌توانید از قطع شدن ارتباط مطلع شوید.

نکته جالب دیگر درباره select() این است که اگر شما سوکتی داشته باشید که در حال listen() شدن است می‌توانید با قرار دادن توصیف‌کننده آن سوکت در مجموعه readfds از وجود ارتباطات جدید آگاه شوید.

مثال زیر مانند یک سرویس‌دهنده چت (Chat Server) چند منظوره عمل می‌نماید. برای امتحان آن باید برنامه را در یک پنجره اجرا نمایید و سپس در چند پنجره مجزا به آن telnet کنید (telnet hostname 9034) و سپس متنی را در یکی از پنجره‌ها وارد نمایید. خواهید دید که متن به همه پنجره‌ها فرستاده خواهد شد.

کد برنامه به صورت زیر است:

```
/*
** selectserver.c - a cheezy multiperson chat
server
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#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;
myaddr.sin_port = htons(PORT);
memset(&(myaddr.sin_zero), '\0', 8);
if (bind(listener, (struct sockaddr *)&myaddr,
sizeof(myaddr)) == -1) {
perror("bind");
exit(1);
}
// listen
if (listen(listener, 10) == -1) {
perror("listen");
exit(1);
}
// add the listener to the master set
```

```
FD_SET(listener, &master);
// keep track of the biggest file descriptor
fdmax = listener; // so far, it’s this one
// main loop
for(;;) {
read_fds = master; // copy it
if (select(fdmax+1, &read_fds, NULL, NULL, NULL)
== -1) {
perror("select");
exit(1);
}
// run through the existing connections looking
for data to read
for(i = 0; i <= fdmax; i++) {
if (FD_ISSET(i, &read_fds)) { // we got one!!
if (i == listener) {
// handle new connections
addrlen = sizeof(remoteaddr);
if ((newfd = accept(listener, (struct sockaddr
*)&remoteaddr,
&addrlen)) == -1) {
perror("accept");
} else {
FD_SET(newfd, &master); // add to master set
if (newfd > fdmax) { // keep track of the maximum
fdmax = newfd;
}
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 نمایید من در مثال فوق دو مجموعه توصیف‌کننده فایل دارم: master و readfs. اولی(master) تمامی توصیف‌کننده‌هایی را که در حال حاضر متصل هستند و توصیف کننده‌هایی را که در حال گوش داده شدن هستند در خود نگه داری می‌نماید. این بدان دلیل است که select() مجموعه هایی را که به آن می‌دهید تغییر می‌دهد تا سوکت‌هایی را که آماده هستند منعکس نماید، و برای این که بتوانم بین یک فرا خوانی select() و فراخوانی بعدی ارتباطات موجود را نگه داری نمایم باید آنها را در جایی دیگر ذخیره می‌کردم، سپس در آخرین لحظه master را در readfs کپی نموده و تابع select() را فراخوانی نموده ام. و این بدین معناست که هرگاه یک ارتباط جدید داشتم آن را به مجموعه master اضافه کرده‌ام و هر گاه یک ارتباط قطع می‌شده آن را از master حذف کرده ام.

توجه کنید من سوکت listener را چک کرده‌ام تا ببینم آماده خواندن است یا خیر و هر گاه آماده خواندن بود بدین معنی است که یک ارتباط جدید داریم. پس آن را accept() کرده و سپس به مجموعه master اضافه نموده‌ام. و در زمانی که یک ارتباط آماده خواندن است و تابع recv() مقدار صفر را باز می‌گرداند فهمیده‌ام که ارتباط قطع شده است و باید آن را از لیست master کسر نمایم. و اگر تابع recv() یکی از مشتری‌ها مقدار غیر صفری باز گرداند به معنی دریافت اطلاعات است. بنابراین آن را برای تمامی مشتری های دیگر از میان لیست master فرستاده‌ام.

رسیدگی به چندین ارسال :

به یاد دارید که در بخش های ابتدایی گفته شد تابع send() ممکن است نتواند تمامی اطلاعاتی را که به آن داده‌اید در یک بار ارسال نماید؟ اما چاره این کار چیست؟ خوب شما می‌توانید تابعی مانند مثال بنویسید تا انجام این کار را برای شما به عهده بگیرد.

به مثال زیر توجه نمایید:

```
#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 مثال فوق s سوکتی است که می‌خواهید اطلاعات را به آن ارسال نمایید و buf بافری است که اطلاعات در آن قرار دارد و len هم اشاره‌گری به int است که طول داده های موجود در بافر را در خود نگه داری می‌کند. تابع فوق در مواجهه با خطا مقدار ۱- را باز می‌گرداند. و تعداد بایت هایی را که می‌فرستد نیز در len قرار می‌دهد. این تعداد با تعداد درخواستی شما یکی است، مگر این که خطایی رخ دهد.

برای آشنایی با طرز فراخوانی تابع فوق به مثال زیر توجه نمایید:

```
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);
}
```

اما سوالی که در اینجا پیش می‌آید این است که در مواقعی که اطلاعات به صورت قطعه به قطعه فرستاده می‌شوند در سمت گیرنده چه اتفاقی می‌افتد؟ اگر بسته‌ها در سایز های متفاوتی باشند ابتدا و انتهای هر یک از کجا مشخص می‌شود؟ در چنین مواقعی باید اطلاعات را بسته بندی نماییم. در بخش بعد چگونگی این کار را مورد بررسی قرار می‌دهیم.

بسته بندی داده‌ها:

بسته‌بندی داده‌ها به چه معنا است؟ خوب در ساده‌ترین حالت بسته‌بندی شما یکسری اطلاعات کنترلی (هدر) را که می‌تواند شامل طول بسته، اطلاعات شناسایی یا هر دو باشد به بسته اضافه می‌نمایید. اما هدر شما باید به چه صورتی باشد؟ در واقع هدر شما فقط شامل یکسری اطلاعات دودویی است که شما احساس کرده‌اید برای انجام پروژه تان ضروری است. به عنوان مثال فرض کنید که یک برنامه چت چند کاربره دارید که از SOCK_DGRAMها استفاده می‌نماید، در چنین حالتی زمانی که کاربر چیزی را تایپ می‌کند دو نوع اطلاعات باید به سرویس‌دهنده فرستاده شود: پیامی که نوشته شده و نام کاربری که پیام را نوشته است. خوب ممکن است از خود بپرسید مشکل چیست؟ پاسخ شما این است که پیام‌ها می‌توانند در سایز های متفاوتی باشند. به عنوان مثال شخصی به نام Tom ممکن است بنویسد "Hi" و شخصی به نام Benjamin بنویسد: "Hey guys! what is up?". اگر شما این داده‌ها را به همان شکلی که ارسال شده‌اند به مقصد بفرستید چیزی شبیه به متن زیر خواهید داشت:

```
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 ?
```

در چنین حالاتی یک مشتری از کجا باید بداند یک پیام از کجا شروع و در چه نقطه‌ای پایان یافته است؟ یکی از راه حل های موجود این است که طبق یک فرض تمامی پیام‌ها را یک اندازه بگیریم، اما چنین راه حلی فقط باعث به هدر رفتن پهنای باند می‌شود. چرا که نمی‌خواهیم مثلا ۱۰۲۴ بایت را به Tom اختصاص دهیم و او هم فقط پیغام Hi را ارسال نماید. به همین دلیل بهتر است داده‌ها را در یک بسته که شامل یک یا چند هدر است قرار دهیم، در ضمن هر دو طرف سرویس گیرنده و سرویس‌دهنده طریقه بسته بندی و باز کردن بسته‌ها را می‌دانند. شاید به نظر نیاید ولی ما در حال تعریف یک پروتکل بین مشتری و سرویس‌دهنده هستیم.

خوب در این مثال اجازه دهید فرض کنیم نام هر کاربر می‌تواند تا حداکثر ۸ کراکتر باشد که مکان های خالی آن با \0 پر می‌شوند و فرض کنیم طول داده‌ها تا ۱۲۸ کراکتر متغیر است. خوب بیایید تا اینجا نگاهی به ساختار بسته خود بیندازیم:

۱- طول بسته(len) : ۱بایت (بدون علامت) : که شامل ۸ کراکتر برای نام کاربر و تا حداکثر ۱۲۸ کراکتر برای پیام است.

۲- نام کاربر(name) : نام کاربر است.

۳- پیغام چت (chat data) که n بایت است و نباید از ۱۲۸ کراکتر بیشتر شود. طول این داده‌ها باید محاسبه شده و با ۸ کراکتر نام کاربر جمع شود.

با توجه به تعریف بالا در مورد هر بسته، بسته اول به صورت زیر خواهد بود(مبنای ۱۶ و کد اسکی):

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

و بسته دوم:

```
  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  ...
```

در این مثال طول(len) باید به صورت Network Byte Order باشد. برای فرستادن این بسته‌ها باید مطمئن شوید که تمامی اطلاعات فرستاده می‌شوند؛ حتی اگر مجبور شوید چندین بار اطلاعات را بفرستید. برای این کار می‌توانید از تابعی مثل sendall() که در مثالهای قبل نوشتیم و یا چیزی شبیه به آن استفاده نمایید.

پس از دریافت اطلاعات نیز باید کارهای اضافی انجام دهید؛ برای اطمینان باید فرض کنید ممکن است" بخشی" از یک بسته را دریافت کنید. برای این کار باید تابع recv() را آنقدر فرا خوانی نمایید تا تمامی بسته را دریافت کنید. اما چگونه می‌توان این کار را انجام داد؟ خوب ما تعداد کل بایت هایی را که باید دریافت کنیم می‌دانیم؛ چون آنها را در ابتدای هر بسته جاسازی نموده‌ایم. در ضمن از حداکثر سایز هر بسته نیز مطلع هستیم : ۱ + ۸ + ۱۲۸؛ در واقع ۱۳۷ بایت برای هر بسته. تنها کاری که شما باید انجام دهید تعریف آرایه‌ای است که به اندازه دو بسته جا داشته باشد. این آرایه در واقع همان بافر کاری شما است. هر بار که بسته‌ای را دریافت می‌کنید باید آن را در این بافر قرار دهید و در

صورتی که طول بسته بزرگتر یا مساوی با طول مشخص شده در هدر بود(۱+ چرا که فیلد طول شامل اندازه خود فیلد نیست) بسته کامل است اما اگر طول داده های موجود در بافر کمتر از ۱ بود فیلد طول از دست رفته است و نمی توان برای طول بسته به آن استناد نمود. در صورتی که بسته به صورت کامل دریافت شود می توانید عمل مورد نظر خود را با آن انجام دهید.

البته در مواردی ممکن است شما با یک بار فراخوانی recv() دو بسته را به صورت تکه تکه دریافت نمایید. در چنین مواقعی چون طول بسته اول را از طریق هدر آن می‌دانید می‌توانید تا انتهای بسته اول از بافر خود بخوانید سپس بسته اول را از بافر پاک نموده و بسته دوم را به ابتدای بافر انتقال دهید و شروع به کار روی بسته دوم نمایید. بعضی از خواننده ها ممکن است متوجه این نکته شده باشند که اعمال فوق(جا به جایی بسته ها) در بافر زمان گیر است و می توان با تکنیک هایی نظیر بافر های حلقوی این مشکل را حل نمود اما متاسفانه بحث در مورد این تکنیک ها خارج از حوصله این مقاله است.

من هرگز نگفتم که یادگرفتن برنامه نویسی سوکت ساده است. ... خیلی خوب گفتم!! و راحت است. فقط باید تمرین کنید و قسم می خورم که یاد بگیرید.

در پایان امیدوارم استفاده لازم را از این مقاله(ترجمه) ببرید. نظرات و انتقادات خود را می توانید با شماره ۰۹۳۵-۲٦٦۹۳٦۷ با بنده در میان بگذارید یا با آدرس ایمیل بنده مکاتبه نمایید و یا در بخش نظرات وبلاگ که در پایین تمامی صفحات آمده است مراجعه نمایید.

منابع دیگر

برای اطلاعات بیشتر می‌توانید به منابع زیر رجوع نمایید:

- صفحه راهنمای مربوط به هریک از توابع ذکر شده در مقاله برای مبتدیان.
- کتابها:

Unix Network Programming, volumes 1-2 by W. Richard Stevens. Published by Prentice Hall. ISBNs for volumes
1-2: 013490012X[46], 0130810819[47].
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[48], 0139738436[49], 0138487146[50].
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[51], 020163354X[52], 0201634953[53].
TCP/IP Network Administration by Craig Hunt. Published by O'Reilly & Associates, Inc. ISBN 1565923227[54].
Advanced Programming in the UNIX Environment by W. Richard Stevens. Published by Addison Wesley. ISBN
0201563177[55].
Using C on the UNIX System by David A. Curry. Published by O'Reilly & Associates, Inc. ISBN 0937175234. *Out of print.*

- مراجع وب:

BSD Sockets: A Quick And Dirty Primer[56] (has other great Unix system programming info, too!)
The Unix Socket FAQ[57]
Client-Server Computing[58]
Intro to TCP/IP[59] (gopher)
Internet Protocol Frequently Asked Questions[60]
The Winsock FAQ[61]

- در خواست برای توضیحات (RFC):

RFCs[62]–the real dirt:
RFC-768[63]–The User Datagram Protocol (UDP)
RFC-791[64]–The Internet Protocol (IP)
RFC-793[65]–The Transmission Control Protocol (TCP)
RFC-854[66]–The Telnet Protocol
RFC-951[67]–The Bootstrap Protocol (BOOTP)
RFC-1350[68]–The Trivial File Transfer Protocol (TFTP)