



اصول مهندسی

اینترنت

HTTP



TCP/IP

OSPF

گردآوری و تألیف:
مهندس احسان ملکیان



HTTP

(۱) مقدمه

در فصول گذشته ساختار پروتکل‌های TCP و IP را بررسی کردیم و طریقه آدرس‌دهی ماشینها و پروسه‌های روی هر ماشین را بوسیله آدرس IP و آدرس پورت آموختیم. معمولاً پیاده‌سازی این پروتکلها توسط طراح هر سیستم عامل انجام و بعنوان جزئی از سیستم عامل همراه آن ارائه و نصب می‌شود. در سیستم عامل‌هایی مثل یونیکس یا MS-Windows که اصل^۱ برنامه آن در دسترس نیست این انعطاف وجود ندارد که بتوان در محیطهای آزمایشگاهی برنامه‌های TCP و IP یا پروتکل‌های مرتبط با آنها را تغییراتی داد و نتیجه تغییرات را بررسی و تحلیل کرد لذا نظر علاقمندان به این مورد را به سیستم عامل "لینوکس" جلب می‌نمائیم.

حال فرض می‌کنیم یک برنامه نویس بخواهد در یک محیط برنامه نویسی مثل C بگونه‌ای برنامه نویسی کند که محتویات یک فایل درون یک کامپیوتر راه دور را تغییر بدهد یا آنرا روی کامپیوتر خودش منتقل نماید یا فرض کنید یک برنامه نویس موظف شده است که یک محیط پست الکترونیکی خاص و با امکانات ویژه برای بکارگیری در یک محیط اداری طراحی نماید. برای طراحی چنین برنامه‌هایی که تماماً در لایه چهارم یعنی لایه کاربرد تعریف می‌شود برنامه نویس باید به نحوی با مفاهیم برنامه نویسی تحت شبکه آشنا باشد.

در این فصل اصول کلی برنامه نویسی شبکه و مفهوم "سوکت"^۲ را مورد بررسی قرار می‌دهیم و با مثالهای ساده روش نوشتن برنامه‌های کاربردی تحت پروتکل TCP/IP را تشریح خواهیم کرد. برای سادگی کار و همچنین ارائه دید عمیق، کدهایی که در این فصل ارائه می‌شوند به زبان C هستند که در محیط سیستم عامل لینوکس و با مترجم gcc به زبان ماشین ترجمه شده‌اند. در حقیقت این فصل نقطه آغازی است برای تمام برنامه نویسانی که به نحوی مجبور خواهند شد برنامه کاربردی تحت شبکه اینترنت بنویسند.

سنگ بنای تمام برنامه‌های کاربردی لایه چهارم مفهومی بنام سوکت است که این مفهوم توسط طراحان سیستم عامل یونیکس به زیبایی به منظور برقراری ارتباط برنامه‌های تحت شبکه و تبادل جریان داده بین پروسه‌ها ابداع شد و در این فصل باید مفهوم آنرا کالبد شکافی کنیم.

شاید شما این جمله را شنیده باشید "در دنیای یونیکس هر چیزی می‌تواند بصورت یک فایل تلقی و مدل شود". تمام عوامل و انواع ورودی و خروجیها (I/O) می‌تواند توسط سیستم فایل مدل شود. مثلاً چاپگر می‌تواند یک فایل باشد (مثلاً فایلی با نام PRN). حال وقتی سیستم عامل چاپگر را بصورت یک فایل استاندارد مدل کرده باشد شما با مفاهیمی که از فایلها و چگونگی بکارگیری آنها در محیط برنامه نویسی آموخته اید، برای راه اندازی چاپگر و چاپ یک متن، می‌توانید در برنامه خود عملیات ساده و در عین حال استاندارد زیر را انجام بدهید:

(الف) چاپگر را همانند یک فایل با نام استاندارد آن بصورت فایلی "فقط نوشتنی" باز می‌کنید. (با دستورات open() یا fopen()).

(ب) اگر نتیجه مرحله قبل موفقیت آمیز بود سیستم عامل یک مشخصه فایل^۳ بعنوان اشاره گر فایل

^۱ Source

^۲ Socket

^۳ File Descriptor

برمی گرداند.

ج) داده‌هایی که قرار است بر روی چاپگر ارسال شوند را با همان دستورات معمولی نوشتن در فایل (دستور معمولی write() یا fwrite() ، درون فایل باز شده از مرحله قبل می‌نویسید. د) پس از اتمام کار فایل را می‌بندید. (دستور close() یا fclose())

کلیت کاری که باید انجام بشود همین چند مرحله است و برنامه نویسی به هیچ عنوان درگیر ساختار چاپگر و اعمالی که برای راه اندازی و چاپ یک متن لازم است نخواهد شد. این وظائف را راه انداز چاپگر بعنوان بخشی از پوسته سیستم عامل بعهده دارد.

چهار مرحله‌ای که برای بکارگیری چاپگر معرفی شد دقیقاً میتواند برای نوشتن بر روی صفحه نمایش یا خواندن از آن مورد استفاده قرار گیرد ، فقط باید نام فایل صفحه نمایش "کنسول" (con) در نظر گرفته شود.

یونیکس قادر است تمام دستگاههای ورودی و خروجی را بعنوان فایل مدل نماید. بنابراین تمام عملیاتی که برنامه نویسی برای بکارگیری دستگاههای مختلف بایستی بداند و بکار بگیرد یکسان و ساده و شفاف خواهد بود. آیا می‌توانید گزاره های زیر را بپذیرید:

- چاپگر فایلی است فقط نوشتنی
- پوششگر^۱ فایلی است فقط خواندنی
- صفحه نمایش بعنوان کنسول فایلی است خواندنی و نوشتنی
- پورت سریال فایلی است خواندنی و نوشتنی
- یک فایل واقعی روی دیسک سخت فایلی است خواندنی و نوشتنی
- یک فایل واقعی روی دیسک فشرده فایلی است فقط خواندنی
- صف FIFO یا خط لوله^۲ در محیط یونیکس فایلهایی هستند خواندنی و نوشتنی

حال که ذهن شما این نکته را پذیرفت که هر نوع I/O در دنیای سیستم عامل بصورت یک فایل استاندارد قابل عرضه و مدل کردن است ، شما را با یک سوال کلیدی مواجه می‌کنیم :
"آیا ارتباط دو کامپیوتر روی شبکه و مبادله اطلاعات بین آن دو، ماهیت ورودی / خروجی (I/O) ندارد؟"

اگر جوابتان منفی است این فصل را رها کنید ولی اگر تردید دارید یا یقیناً جوابتان مثبت است تا انتها این فصل را دنبال نمایید.

اگر ساختار فایل را برای ارتباطات شبکه ای تعمیم بدهیم آنگاه برای برقراری ارتباط بین دو برنامه روی کامپیوترهای راه دور روال زیر پذیرفتنی است :

الف) در برنامه خود از سیستم عامل بخواهید تا شرایط را برای برقراری یک "ارتباط" با کامپیوتری خاص (با آدرس IP مشخص) و برنامه ای خاص روی آن کامپیوتر (با آدرس پورت مشخص) فراهم کند یا اصطلاحاً سوکتی را بگشاید. اگر این کار موفقیت آمیز بود سیستم عامل برای شما یک اشاره گر

^۱ Scanner
^۲ Pipe

فایل برمی‌گرداند و در غیر اینصورت طبق معمول مقدار پوچ (NULL) به برنامه شما برخواهد گرداند. (ب) در صورت موفقیت آمیز بودن عمل در مرحله قبل، به همان صورتی که درون یک فایل می‌نویسید یا از آن می‌خوانید، می‌توانید با توابع send() [یا write()] و recv() [یا read()] اقدام به مبادله داده‌ها بنمائید.

(ج) عملیات مبادله داده‌ها که تمام شد ارتباط را همانند یک فایل معمولی ببندید. (با تابع close()) برای آنکه در برنامه خود همانند فایل یک "اشاره گر ارتباط" را از سیستم عامل طلب کنید تا برایتان مقدمات یک ارتباط را فراهم کند بایستی تابع سیستمی socket() را در برنامه خود صدا بزنید. در صورتی که عمل موفقیت آمیز بود، یک اشاره‌گر غیر پوچ بر می‌گردد که از آن برای فراخوانی توابع و روالهای بعدی استفاده خواهد شد.

پس از این هرگاه از "سوکت باز" یا مبادله داده‌ها روی سوکت یاد کردیم منظورمان اشاره به یک ارتباط باز یا مبادله اطلاعات بین دو نقطه TSAP^۱ روی دو سیستم شبکه کامپیوتری می‌باشد. دقیقاً همانند فایلها که میتوان همزمان چندین فایل را در یک برنامه واحد باز کرد و روی هر یک از آنها (با استفاده از اشاره گر فایل) نوشت یا از آنها خواند، در یک برنامه تحت شبکه می‌توان بطور همزمان چندین ارتباط فعال و باز داشت و با مشخصه هر یک از این ارتباطها روی هر کدام از آنها مبادله داده انجام داد.

۲) انواع سوکت و مفاهیم آنها

اگر بخواهیم از نظر اهمیت انواع سوکت را معرفی کنیم دو نوع سوکت بیشتر وجود ندارد. (انواع دیگری هم هستند ولی کم اهمیت ترند). این دو نوع سوکت عبارتند از:

- سوکتهای نوع استریم که سوکتهای اتصال گرا^۲ نامیده می‌شود.
- سوکتهای نوع دیتاگرام که سوکتهای بدون اتصال^۳ نامیده میشود.

اگر عادت به پیش داوری دارید برای تمایز بین مفهوم این دو نوع سوکت، تفاوت بین مفاهیم ارتباط نوع TCP و UDP را مد نظر قرار بدهید. روش ارسال برای سوکتهای نوع استریم همان روش TCP است و بنابراین داده‌ها با رعایت ترتیب و مطمئن با نظارت کافی برخطاهای احتمالی مبادله می‌شوند. سوکتهای نوع دیتاگرام نامطمئن است و هیچگونه تضمینی در ترتیب جریان داده‌ها وجود ندارد.

اکثر خدمات و پروتکل‌هایی که در لایه چهارم تعریف شده‌اند و در فصول بعدی آنها را بررسی می‌کنیم نیازمند حفظ اعتبار و صحت داده‌ها و همچنین رعایت ترتیب جریان داده‌ها هستند. بعنوان مثال پروتکل انتقال فایل (FTP)، پروتکل انتقال صفحات ابرمتن (HTTP) یا پروتکل انتقال نامه‌های الکترونیکی (SMTP) همگی نیازمند برقراری یک ارتباط مطمئن هستند و طبعاً از سوکتهای نوع استریم بهره می‌برند.

^۱ Transport Service Access Point

^۲ Connection Oriented

^۳ Connectionless

همانگونه که قبلاً در مورد پروتکل TCP آموختیم پروتکلی است که داده‌ها را با رعایت ترتیب و خالی از خطا مبادله می‌نماید و پروتکل IP که در لایه زیرین آن واقع است با مسیریابی بسته‌ها روی شبکه سروکار دارد. سوکتهای نوع استریم دقیقاً مبتنی بر پروتکل TCP بوده و طبیعتاً قبل از مبادله داده‌ها باید یک اتصال^۱ به روش دست‌تکانی سه‌مرحله‌ای^۲ برقرار بشود.

سوکتهای نوع دیتاگرام مبتنی بر پروتکل UDP است و بدون نیاز به برقراری هیچ ارتباط و یا اتصال، داده‌ها مبادله میشوند و بنابراین تضمینی بر رسیدن داده‌ها، صحت داده‌ها و تضمین ترتیب داده‌ها وجود ندارد ولی با تمام این مشکلات باز هم در برخی از کاربردها مثل انتقال صدا و تصویر یا سیستم DNS که قبلاً آنرا بررسی کردیم مورد استفاده قرار می‌گیرد. تنها حسن این روش سرعت انتقال داده‌ها می‌باشد.

در حقیقت شما با استفاده از سوکتها میخواهید یک ابزار برای استفاده از پروتکلهای TCP یا UDP در اختیار داشته باشید.

” سوکت یک مفهوم انتزاعی از تعریف ارتباط در سطح برنامه‌نویسی خواهد بود و برنامه‌نویس با تعریف سوکت عملاً تمایل خود را برای مبادله داده‌ها به سیستم عامل اعلام کرده و بدون درگیر شدن با جزئیات پروتکل TCP یا UDP از سیستم عامل می‌خواهد تا فضا و منابع مورد نیاز را جهت برقراری یک ارتباط، ایجاد کند.“

۳) مفهوم سرویس دهنده / مشتری^۳

در برنامه نویسی شبکه این نکته قابل توجه است که هر ارتباطی دو طرفه می‌باشد یعنی عملاً ارتباط مابین دو پروسه تعریف می‌شود لذا طرفین ارتباط بایستی در لحظه شروع تمایل خود را برای مبادله داده‌ها به سیستم عامل اعلام کرده باشند. در هر ارتباط یکی از طرفین، شروع کننده ارتباط تلقی می‌شود و طرف مقابل در صورت آمادگی این ارتباط را می‌پذیرد. در صورت پذیرش ارتباط، مبادله داده‌ها امکان پذیر خواهد بود. اگر برنامه‌ای را که شروع کننده ارتباط است، ”برنامه مشتری“ بنامیم قاعدتاً برنامه‌ای که این ارتباط را می‌پذیرد (و منتظر آن بوده) ”سرویس دهنده“ نام خواهد گرفت.

تعریف عمومی: مشتری (Client) پروسه‌ای است که اصولاً نیازمند مقداری اطلاعات است. سرویس دهنده (Server) پروسه‌ای است که اطلاعاتی را در اختیار دارد و تمایل دارد تا این اطلاعات را به اشتراک بگذارد و منتظر می‌ماند تا یک متقاضی، واحدی از این اطلاعات را طلب کند و او آنرا تحویل بدهد.

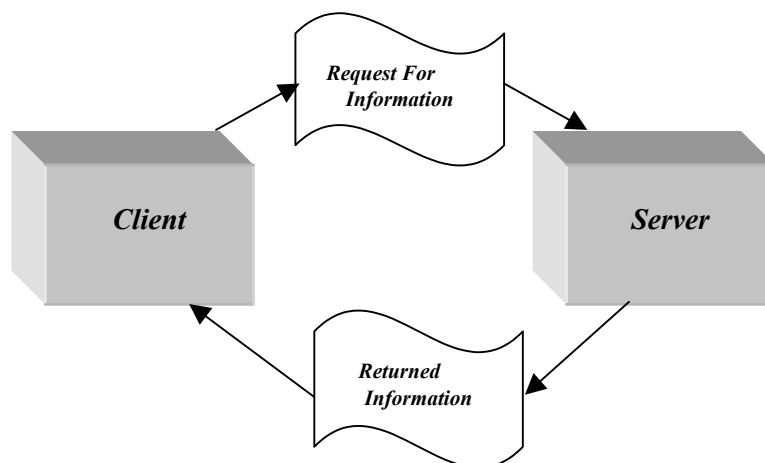
^۱ Connection
^۲ Tree Way Handshake
^۳ Client/Server

بعنوان مثال وقتی سخن از سرویس دهنده وب^۱ در میان است در یک عبارت ساده، منظور سیستمی است که اطلاعاتی را در قالب صفحات وب^۲ در اختیار دارد و در عین حال منتظر است که کسی تقاضای یکی از این صفحات را نموده و او این درخواست را اجابت کرده و داده‌های لازم را در پاسخ به این تقاضا ارسال نماید.

برنامه سمت سرویس دهنده^۳ برنامه‌ای است که روی ماشین سرویس دهنده نصب میشود و منتظر است تا تقاضائی مبنی بر برقراری یک ارتباط دریافت کرده و پس از پردازش آن تقاضا، پاسخ مناسب را ارسال نماید بنابراین در حالت کلی برنامه سرویس دهنده شروع کننده یک ارتباط نیست.

در طرف مقابل برنامه‌های سمت مشتری^۴ هستند که بنا بر نیاز، اقدام به درخواست اطلاعات می‌کنند. تعداد مشتریها روی ماشینهای متفاوت یا حتی روی یک ماشین می‌تواند متعدد باشد و لیکن معمولاً تعداد سرویس دهنده‌ها یکی است. (مگر در سیستم‌های توزیع شده که مورد بحث ما نیستند). برای مثال جلسه پرسش و پاسخی را در نظر بگیرید که یکنفر صاحب اطلاعات، پاسخگو و منتظر سوال است - سمت سرویس دهنده - در طرف دیگر تعدادی سوال کننده هستند که مختارند در رابطه با موضوع مورد بحث سوال نمایند - سمت مشتری -.

به نظر می‌رسد با دقت در مفهوم سرویس دهنده/ مشتری متقاعد بشوید که ساختار برنامه‌ای که در سمت سرویس دهنده در حال اجراست با برنامه‌ای که در سمت مشتری اجرا می‌شود، متفاوت خواهد بود.



شکل (۷-۱) ارتباط بین سرویس دهنده و مشتری

^۱ Web Server
^۲ Web Page
^۳ Server Side
^۴ Client Side

قبل از آنکه وارد مقوله برنامه نویسی سوکت بشویم بد نیست الگوریتم کل کاری که بایستی در سمت سرویس دهنده و همچنین در سمت مشتری انجام بشود، بررسی نمائیم :
برنامه شما در سمت سرویس دهنده به عملیات زیر نیازمند خواهد بود :

الف) یک سوکت را که مشخصه یک ارتباط است ، بوجود بیاورید. تا اینجا فقط به سیستم عامل اعلام کرده‌اید که نیازمند تعریف یک ارتباط هستید. در همین مرحله به سیستم عامل نوع ارتباط درخواستی خود را (TCP یا UDP) معرفی می‌نمائید. این کار در محیط سیستم عامل یونیکس توسط تابع سیستمی (socket) انجام می‌شود.

ب) به سوکتی که باز کرده‌اید یک آدرس پورت نسبت بدهید. این کار توسط تابع سیستمی (bind) انجام می‌شود و در حقیقت با این کار به سیستم عامل اعلام می‌کنید که تمام بسته های TCP (یا UDP) را که آدرس پورت مقصدشان با شماره مورد نظر شما مطابقت دارد ، به سمت برنامه شما هدایت کند. در حقیقت با این کار خودتان را بعنوان پذیرنده دسته ای از بسته های TCP یا UDP با شماره پورت خاص معرفی کرده‌اید. (دقت کنید که در برنامه سمت سرویس دهنده استفاده از دستور (bind) الزامی است)

ج) در مرحله بعد به سیستم عامل اعلام می‌کنید که کارش را برای پذیرش تقاضاهای ارتباط TCP شروع نماید. این کار توسط تابع سیستمی (listen) انجام می‌شود و چون ممکن است تعداد تقاضاهای ارتباط متعدد باشد باید حداکثر تعداد ارتباط TCP را که می‌توانید پذیرای آن باشید ، تعیین نمائید چرا که سیستم عامل باید بداند برای پذیرش ارتباطات TCP چقدر فضا و منابع شامل بافر در نظر بگیرد. دقت کنید که اعلام پذیرش تقاضاهای ارتباط به معنای پذیرش داده‌ها نیست بلکه فضای لازم را جهت ارسال و دریافت داده‌ها ایجاد می‌کنید. معمولاً تعیین تعداد ارتباطات TCP که میتواند بطور همزمان پذیرفته شده و به روش اشتراک زمانی^۱ پردازش شود ، در اختیار شماست ولی باید این تعداد کمتر از مقداری باشد که سیستم عامل بعنوان حداکثر تعیین کرده است. بازهم یادآوری می‌کنیم که سرویس دهنده می‌تواند بصورت همزمان، چندین ارتباط متفاوت با چندین برنامه روی ماشینهای متفاوت را بصورت باز و فعال داشته باشد. بعنوان یک مقایسه با سیستم فایل تعداد حداکثر ارتباط باز را تعداد فایلی تصور کنید که می‌تواند توسط برنامه شما بطور همزمان باز شود.

د) نهایتاً با استفاده از تابع (accept) از سیستم عامل تقاضا کنید یکی از ارتباطات معلق را (در صورت وجود) به برنامه شما معرفی کند. تابع (accept) نکات ظریفی دارد که به تفصیل بررسی خواهد شد.

ه) از دستورات (send) و (recv) برای مبادله داده‌ها استفاده نمائید.

و) نهایتاً ارتباط را خاتمه بدهید. این کار به دو روش امکان پذیر خواهد بود:

- قطع ارتباط دو طرفه ارسال و دریافت (توسط تابع (close)
- قطع یکطرفه یکی از عملیات ارسال یا دریافت (توسط تابع (shutdown)

در برنامه سمت مشتری بایستی اعمال زیر انجام شود :

^۱ Time sharing

الف) یک سوکت را که مشخصه یک ارتباط است، بوجود بیاورید. تا اینجا فقط به سیستم عامل اعلام شده است که نیازمند تعریف یک ارتباط هستید.

ب) در مرحله بعد لازم نیست همانند برنامه سرویس دهنده به سوکت خود آدرس پورت نسبت بدهید یعنی لزومی به استفاده از دستور `bind()` وجود ندارد چرا که برنامه سمت مشتری منتظر تقاضای ارتباط از دیگران نیست بلکه خودش متقاضی برقراری ارتباط با یک سرویس دهنده است. بنابراین در مرحله دوم به محض آنکه نیازمند برقراری ارتباط با یک سرویس دهنده شدید آن تقاضا را با استفاده از تابع سیستمی `connect()` به سمت آن سرویس دهنده بفرستید.

اگر مراحل دست تکانی سه مرحله ای را در برقراری یک ارتباط TCP بخاطر داشته باشید، دستور `connect()` عملاً متولی شروع و انجام چنین ارتباطی است.

مجدداً تاکید می‌کنیم از تابع `bind()` زمانی استفاده می‌شود که پذیرای ارتباطات TCP با شماره پورت خاصی باشید ولی در طرف مشتری چنین کاری لازم نخواهد بود چرا که برنامه سمت مشتری شروع کننده ارتباط است.

اگر عمل `connect()` موفقیت آمیز بود به معنای موفقیت در برقراری یک ارتباط TCP با سرویس دهنده است و می‌توانید بدون هیچ کار اضافی به ارسال و دریافت داده‌ها اقدام نمایید.

ج) از توابع `send()` , `recv()` برای ارسال یا دریافت داده‌ها اقدام نمایید.

د) ارتباط را با توابع `close()` یا `shutdown()` بصورت دوطرفه یا یکطرفه قطع نمایید.

پس از بررسی الگوریتم کلی برنامه های سمت سرویس دهنده و سمت مشتری وقت آن رسیده است که ساختمان داده‌ها و همچنین توابع و روالهای مورد نیاز در برنامه نویسی را با دقت بیشتری مورد بررسی قرار بدهیم.

۱۴) ساختمان داده‌های مورد نیاز در برنامه نویسی مبتنی بر سوکت

برای آغاز برنامه نویسی بهترین کار آنست که متغیرها و انواع ساختمان داده مورد نیاز در برنامه نویسی سوکت، تحت بررسی قرار بگیرد. (تمام قطعه کدها با C هستند)

اولین نوع داده، "مشخصه سوکت"^۱ است که همانند اشاره‌گر فایل، برای ارجاع به یک ارتباط باز مورد استفاده قرار می‌گیرد و یک عدد صحیح دوبایتی است یعنی با تعریف زیر، متغیر `a` می‌تواند مشخصه یک سوکت باشد:

```
int a;
```

دومین نوع داده برای برقراری ارتباط، یک استراکچر است که آدرس پورت پروسه و همچنین آدرس IP ماشین طرف ارتباط را درخود نگه می‌دارد. فعلاً در تعریفی ساده ساختار آن بصورت زیر است:

^۱ Socket Descriptor


```
struct sockaddr {
```

```
    unsigned short sa_family;    /* address family, AF_xxxx */
    char sa_data[14];           /* 14 bytes of protocol address */
};
```

● **sa_family** : خانواده یا نوع سوکت را مشخص می‌کند. در حقیقت این گزینه تعیین می‌کند که سوکت مورد نظر را در چه شبکه و روی چه پروتکلی بکار خواهید گرفت؛ لذا در سیستمی که با پروتکل‌های متفاوت و سوکت‌های متفاوت سروکار دارد، باید نوع سوکت درخواستی را تعیین کنید. فعلاً در کل این فصل که بحث ما شبکه اینترنت با پروتکل TCP/IP است خانواده سوکت را با ثابت AF_INET مشخص می‌کنیم. در مورد شبکه‌های دیگر مثل Appletalk این گزینه متفاوت خواهد بود.

● **sa_data** : این چهارده بایت مجموعه‌ای است از آدرس پورت، آدرس IP و قسمتی اضافی که باید با صفر پر شود و دلیل آنرا بعداً اشاره می‌کنیم.

همانگونه که اشاره شد این استراکچر بایستی آدرس پورت و آدرس IP را نگه دارد ولی در تعریف بالا چنین فیلدهائی مشاهده نمی‌شود. برای سادگی در برنامه نویسی، استراکچر دیگری معرفی میشود که دقیقاً معادل استراکچر قبلی است ولی تعریف متفاوتی دارد و شما می‌توانید از هر کدام به دلخواه بهره بگیرید:

```
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 */
};
```

● **sin_family** : همانند ساختار قبلی خانواده سوکت را تعیین می‌کند و برای شبکه اینترنت بایستی مقدار ثابت AF_INET داشته باشد.

● **sin_port** : این فیلد دو بایتی، آدرس پورت پروسه مورد نظر را مشخص می‌نماید.

● **sin_addr** : آدرس IP ماشین مورد نظر را مشخص می‌کند. این فیلد خودش یک استراکچر است که در ادامه تعریف خواهد شد، فقط بدانید که کلاً عددی صحیح، بدون علامت و چهاربایتی است.

● **sin_zero[8]** : این هشت بایت در کاربردهای مهندسی اینترنت کلاً باید مقدار صفر داشته باشد. دلیل وجود این فیلد، آنست که مفهوم سوکت برای تمام شبکه‌ها با پروتکل‌های متفاوت، بصورت معادل استفاده شده و بنابراین استراکچر فوق باید بگونه‌ای تعریف شود که برای تمام پروتکل‌های شبکه قابل استفاده باشد. در شبکه اینترنت فعلاً آدرس IP چهاربایتی و آدرس پورت دوبایتی است در حالی که در برخی دیگر از شبکه‌ها طول آدرس بیشتر است. بنابراین هنگامی که از استراکچر فوق در کاربردهای برنامه نویسی شبکه اینترنت بهره می‌گیرید این هشت بایت اضافی است ولی حتماً باید با تابعی مثل memset() تماماً صفر شود.

دقت نمایید که دو استراکچر قبلی دقیقاً معادلند و می‌توان در فراخوانی توابع، هر کدام از آنها را با تکنیک «تطابق نوع»^۱ بجای دیگری بکار برد ولی در مجموع استفاده از تعریف دوم راحت تر خواهد بود.

در تعریف استراکچر دوم یک استراکچر دیگر بنام `in_addr` تعریف شده که ساختار آن بصورت زیر است:

```
/* Internet IP address (a structure for historical reasons) */
```

```
struct in_addr {
    unsigned long s_addr;
};
```

این فیلد چهاربایتی برای نگهداری آدرس IP کاربرد دارد و تعریف آن بصورت فوق کمی عجیب به نظر می‌رسد چرا که می‌توانستیم در استراکچر قبلی بطور مستقیم آنرا `unsigned long` معرفی کنیم ولی بهر حال بصورت بالا تعریف شده است. برای مقداردهی به فیلدهای بالا می‌توانید از هر روشی که دلخواه شماست استفاده کنید ولیکن توابعی ساده برای این کار وجود دارند که در ادامه معرفی خواهند شد.

۵) مشکلات ماشینها از لحاظ ساختار ذخیره سازی کلمات در حافظه

در گذشته تفاوت ماشینهای نوع ^۲BE و نوع ^۳LE را بررسی کردیم و اشاره شد که در پروتکل TCP/IP ترتیب بایتها بصورت BE توافق شده است لذا وقتی قرار است برنامه شما روی ماشینی که ساختار LE دارد نصب شود ترتیب بایتهای ارسالی روی شبکه بهم خواهد خورد. بعنوان مثال وقتی که روی ماشینی از نوع LE دستور زیر اجرا می‌شود:

```
struct sockaddr_in as;
as.sin_port=0xB459;
```

چون بایت کم ارزش اول ذخیره می‌شود و بعد از آن بایت پر ارزش قرار می‌گیرد لذا پس از قرار گرفتن این دو بایت در بسته TCP آدرس پورت بصورت زیر (و قطعاً اشتباه) تنظیم خواهد شد:

59	B4
----	----

بنابراین وقتی قرار است برنامه نویس مقداری را درون فیلدی قرار بدهد که دو بایتی یا چهار بایتی است بایستی نگران نوع ماشین و ترتیب بایتها باشد. بهمین دلیل معرفی توابع زیر بعنوان ابزار کار برنامه نویس شبکه اینترنت ضروری است:

^۱ Casting

^۲ Big Endian

^۳ Little Endian

- htons() : تابع تبدیل کلمات دوبایتی به حالت BE
- htonl() : تابع تبدیل کلمات چهاربایتی به حالت BE
- ntohs() : تابع تبدیل کلمات دوبایتی از BE به حالت فعلی ماشین
- ntohl() : تابع تبدیل کلمات چهاربایتی از BE به حالت فعلی ماشین

برنامه نویس لازم است ساختار ماشین مورد استفاده جهت نصب نهایی برنامه اش را بداند تا در صورت LE بودن حتماً قبل از قرار دادن مقادیر در فیلدهای دوبایتی یا چهاربایتی از توابع فوق استفاده کند.

تذکر: فقط وقتی از توابع فوق استفاده می‌شود که نهایتاً فیلد مورد نظر در بسته TCP یا IP تنظیم شود. بعنوان مثال در استراکچر sock_addr_in در فیلد sin_family مقدار AF_INET که مقداری ثابت است قرار می‌گیرد و این فیلد فقط برای سیستم عامل تعریف شده و روی شبکه منتقل نخواهد شد لذا برای مقدار دهی به این فیلد لازم نیست از توابع فوق استفاده نمائیم. در ادامه با مثالهایی که خواهیم داشت با موارد استفاده توابع فوق آشنا می‌شویم.

۵-۱) مشکلات تنظیم آدرس IP درون فیلد آدرس

در مبحث آدرسهای IP آموختید که آدرسهای IP در قالب چهار فیلد هشت‌تایی^۱ دهی نوشته می‌شوند:

192.140.11.211

در حالی که در استراکچر sock_addr_in فیلد آدرس IP عددی است چهاربایتی که با یک عدد از نوع long پر می‌شود. بنابراین دو تابع زیر جهت انتساب آدرسهای IP با ساختار فوق‌الذکر کاربرد دارد:

- **تابع inet_addr() :** این تابع یک رشته کاراکتری بفرم "187.121.11.44" را گرفته و به یک عدد چهاربایتی با قالب BE تبدیل می‌کند. مثال:

```
struct sockaddr_in ina;
ina.sin_addr.s_addr=inet_addr("130.421.5.10")
```

در مثال بالا آدرس IP رشته‌ای است و پس از تبدیل به عددی چهاربایتی در قالب BE در فیلد مربوطه قرار می‌گیرد.

- **تابع inet_ntoa() :** این تابع عکس عمل تابع قبلی را انجام می‌دهد یعنی یک آدرس چهاربایتی در قالب BE را گرفته و آنرا بصورت یک رشته کاراکتری که آدرس IP را بصورت نقطه‌دار تعریف کرده، تبدیل می‌نماید. پارامتر ورودی تابع فوق از نوع struct in_addr و خروجی آن نوع رشته‌ای است. به مثال زیر دقت کنید:

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

در مثال فوق محتوای آدرس IP بصورت رشته‌ای نقطه دار و در مبنای ده روی خروجی چاپ خواهد

^۱ Octet

شد. مثلاً خروجی به فرم زیر است:

130.141.5.10

در بخشهای آتی چگونگی تبدیل آدرسهای حوزه بفرم www.ibm.com را به آدرس IP در محیط برنامه نویسی توضیح خواهیم داد. قبل از آن باید توابع لازم برای تعریف و برقراری ارتباط تعریف شوند.

۴) توابع مورد استفاده در برنامه سرویس دهنده (مبتنی بر TCP)

۴-۱) تابع socket()

فرم کلی این تابع بصورت زیر است:

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

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

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

- **domain** : این پارامتر نشان دهنده خانواده سوکت است و به نحوی که قبلاً اشاره شد در برنامه نویسی شبکه اینترنت ، با مقدار ثابت AF_INET تنظیم می شود.
- **type** : با این پارامتر نوع سوکت دلخواهتان را اعلام می کنید که می تواند نوع استریم یا از نوع دیتاگرام باشد. اگر سوکت دلخواهتان نوع استریم بود در فیلد type مقدار ثابت SOCK_STREAM قرار بدهید و اگر نوع دیتاگرام خواستید در آن مقدار SOCK_DGRAM تنظیم کنید.
- **protocol** : در این فیلد شماره شناسائی پروتکل مورد نظرتان را تنظیم می کنید که برای کاربردهای شبکه اینترنت همیشه مقدار آن صفر است. مقادیری که در فیلدهای اول و سوم قرار می دهید در برنامه نویسی تحت شبکه اینترنت همیشه ثابت خواهند بود.

مقدار بازگشتی توسط تابع socket() همان مشخصه سوکت است که از آن برای توابع بعدی استفاده خواهد شد (دقیقاً مثل اشاره گر یک فایل) لذا مشخصه سوکت بایستی تا زمانی که ارتباط خاتمه می یابد بدقت نگهداری شود.

اگر مقدار برگشتی تابع socket() ، ۱- باشد عمل موفقیت آمیز نبوده و روند کار نباید متوقف شود و شما بعنوان برنامه نویس موظفید حتماً خروجی این تابع را بررسی کنید چرا که عملیات بقیه توابع که در ادامه معرفی خواهند شد به خروجی همین تابع بستگی دارد. وقتی مقدار برگشتی تابع socket() مقدار ۱- باشد متغیر سراسری errno شماره خطای رخ داده می باشد. برای پردازش شماره خطا تابع سیستمی perror() می تواند استفاده شود که روش بکارگیری آن در مثالها آمده است. این دو متغیر و تابع نیاز به تعریف ندارد و سیستمی هستند.

۲-۶) تابع bind()

وقتی سیستم عامل برای شما یک سوکت باز کرد در حقیقت شما فقط سنگ بنای یک ارتباط را بنا نهاده‌اید ولی هنوز هیچ کاری برای مبادله داده‌ها انجام نشده است. تابع bind() که معمولاً در برنامه سمت سرورس دهنده معنا می‌یابد "عملی است جهت نسبت دادن آدرس پورت به یک سوکت باز شده". این تعریف احتمالاً ابهام دارد پس به تعریف ساده زیر دقت کنید:

از طریق تابع bind() از سیستم عامل خواهش می‌کنید که تمام بسته‌های TCP یا UDP و همچنین تقاضاهای ارتباط با شماره پورت خاص را به سمت برنامه شما هدایت نماید.

بعنوان مثال وقتی گفته می‌شود که پروتکل HTTP به پورت 80 گوش می‌دهد به این معناست که برنامه سرورس دهنده، تمام بسته‌های TCP را که وارد ماشین محلی می‌شوند و شماره پورت مقصد آنها 80 است، تحویل می‌گیرد و پردازش می‌نماید.

فرم کلی تابع bind() بصورت زیر است:

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

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

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

- **sockfd**: همان مشخصه سوکتی است که قبلاً با استفاده از تابع socket() باز کرده‌اید. در حقیقت شما می‌خواهید به سوکت باز شده یک آدرس پورت نسبت بدهید.
- **my_addr**: یک استراکچر که خانواده سوکت، آدرس پورت و آدرس IP ماشین محلی را در خود دارد. ساختار این استراکچر قبلاً تعریف شد.
- **addr_len**: طول استراکچر my_addr بر حسب بایت

برای آشنایی با چگونگی استفاده از توابع فوق به قطعه کد زیر دقت کنید:

```
#include <string.h>
```

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

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

```
#define MYPOR 3490
```

```
main()
```

```
{
```

```
int sockfd;
```

```
struct sockaddr_in my_addr;
```

```
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) != NULL){
```

```

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("132.241.5.10");
bzero(&(my_addr.sin_zero), 8);     /* zero the rest of the struct */

if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr))!=-1){
.
.
.

```

در مورد تابع `bind()` نکاتی وجود دارد که اشاره به آنها خالی از لطف نیست:

الف) در محیط یونیکس اگر فیلد آدرس پورت با مقدار صفر تنظیم شود آنگاه سیستم عامل در بین آدرسهای پورت از شماره ۱۰۲۴ تا ۶۵۵۳۵، یک شماره تصادفی انتخاب کرده و آنرا بعنوان شماره پورت در نظر می گیرد.

ب) برنامه کاربردی شما نباید شماره پورتی را برگزیند که بین صفر تا ۱۰۲۳ باشد چرا که این شماره پورتها برای سرویس دهنده های استاندارد و سرویس دهنده های یونیکس رزرو شده است و سیستم عامل اجازه استفاده از این شماره پورتها را به برنامه های کاربران نخواهد داد.

ج) در محیط یونیکس اگر فیلد آدرس IP را با مقدار ثابت `INADDR_ANY` تنظیم کنید، آنگاه سیستم عامل بصورت خودکار آدرس IP ماشین محلی شما را استخراج و در آن قرار خواهد داد.

د) نکته ای که ممکن است بر آن خرده بگیرید آن است که چرا در مقداردهی فیلدهای بالا سعی نکردیم با بهره گیری از توابع `htons()` آن را به حالت BE تبدیل کنیم در حالی که چنین کاری لازم می باشد. دلیل آن بسیار ساده است: هر دو مقدار صفر دارند و حالت صفر نیازی به تبدیل ندارد.

ه) اگر شماره پورتی که انتخاب می کنید برنامه دیگری قبل از شما برای خود رزرو کرده باشد یعنی آنرا در برنامه خود به سوکتی `bind()` کرده باشد آنگاه عمل `bind()` موفقیت آمیز نبوده و مقدار (۱-) به برنامه شما باز خواهد گشت. برای پردازش نوع خطا، متغیر سراسری `errno` شماره خطا و تابع `perror()` مشخصات خطا را بر می گرداند.

۳-۶) تابع `listen()`

این تابع فقط در برنامه سرویس دهنده معنا می یابد و در یک عبارت ساده اعلام به سیستم عامل برای پذیرش تقاضاهای ارتباط TCP است. به عبارت بهتر توسط این تابع به سیستم عامل اعلام می کنید که از این لحظه به بعد (یعنی زمان اجرای تابع) تقاضاهای ارتباط TCP ماشینهای راه دور با شماره پورت مورد نظران را به صف کرده و منتظر نگه دارد.

با توجه به آنکه ممکن است پس از راه اندازی برنامه سرویس دهنده، در لحظاتی چندین پروسه متفاوت بطور همزمان تقاضای برقراری ارتباط TCP به یک آدرس پورت بدهند بنابراین سیستم عامل

باید بداند که حداکثر چند تا از آنها را بپذیرد و ارتباط آنها را به روش دست تکانی سه مرحله ای برقرار نموده و آنها را در صف سرویس دهی قرار بدهد. توسط تابع `listen()` باید به سیستم عامل اعلام شود که حداکثر تعداد ارتباطات فعال و باز روی یک شماره پورت خاص چند تا باشد. فرم کلی تابع بصورت زیر است:

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

- **sockfd**: همان مشخصه سوکت است که در ابتدا آنرا ایجاد کرده‌اید.
- **backlog**: حداکثر تعداد ارتباطات معلق و به صف شده منتظر. در بسیاری از سیستمها مقدار backlog به ۲۰ محدود شده است.

همانند توابع قبلی در صورت بروز خطا مقدار برگشتی این تابع ۱- خواهد بود و متغیر `errno` شماره خطای رخ داده می‌باشد.

۴-۶) تابع `accept()`

این تابع اندکی مرموز به نظر می‌رسد و بایستی به مفهوم آن دقت شود: پس از آنکه تابع `listen()` اجرا شد تقاضای ارتباط TCP پروسه های روی ماشینهای راه دور (در صورت وجود) پذیرفته ، به صف شده و معلق نگاه داشته میشود. وقتی که تابع `accept()` اجرا می‌شود در حقیقت برنامه شما از سیستم عامل تقاضا می‌کند که از بین تقاضاهای به صف شده یکی را انتخاب کرده و آنرا با مشخصات پروسه طرف مقابل تحویل برنامه بدهد. بنابراین برنامه باید از بین ارتباطات معلق یکی را به حضور بطلبد تا عملیات لازم را انجام بدهد. بهمین دلیل سیستم عامل یک مشخصه سوکت جدید ایجاد کرده و آنرا به برنامه بر می‌گرداند. در اینجا شما یک سوکت جدید دارید. مشخصه سوکت اول که توسط تابع `socket()` بدست آمده و مشخصه سوکت دوم که با تابع `accept()` به برنامه شما برگشته است. تفاوت این دو سوکت در چیست؟

الف) از سوکت اول برای پذیرش یکی از ارتباطات معلق در دستور `accept()` استفاده می‌کنید. در حقیقت این سوکت مشخصه کل ارتباطات به صف شده منتظر میباشد.

ب) از سوکت دوم برای دریافت و ارسال اطلاعات روی یکی از ارتباطات معلق استفاده می‌کنید. این سوکت مشخصه یکی از ارتباطات به صف شده می‌باشد.

فرم کلی تابع به صورت زیر است :

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

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

- **sockfd**: مشخصه سوکت است که در ابتدا با تابع `socket()` بدست آمده است.
- **addr**: اشاره گر به استراکچری است که شما آنرا بعنوان پارامتر به این تابع ارسال می‌کنید تا سیستم عامل پس از پذیرش یک ارتباط معلق آدرس پورت و آدرس IP طرف مقابل ارتباط را در آن به برنامه شما برگرداند. ساختار این استراکچر قبلاً

معرفی شد.

• **addrlen** : طول استراکچر `addr` بر حسب بایت

مقدار برگشتی این تابع یک مشخصه سوکت است که در روالهای بعدی مورد استفاده قرار می‌گیرد. اگر مقدار برگشتی (-۱) باشد خطائی رخ داده است که شماره آن خطا در متغیر سراسری `errno` قابل بررسی است.

مثال ناتمام زیر برای روشن شدن کلیت کار بسیار سودمند خواهد بود:

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

#define MYPOR 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 sockfd, 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;

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) != NULL) {

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

        if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) != -1) {

            listen(sockfd, BACKLOG);

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


بار دیگر تأکید می‌کنیم که برای ارسال یا دریافت داده‌ها بایستی از سوکت جدیدی که مشخصه آن توسط تابع `accept()` برمی‌گردد، استفاده کنید.

۴-۵) توابع `recv()` و `send()`

این دو تابع در برنامه سمت سرور و برنامه سمت مشتری قابل استفاده بوده و برای مبادله داده‌ها کاربرد دارند. فرم کلی دو تابع به صورت زیر است:

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

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

- **sockfd**: مشخصه سوکتی که از تابع `accept()` بدست آمده است.
- **msg**: محلی در حافظه (مثل آرایه یا استراکچر) که داده‌های ارسالی از آنجا استخراج شده و درون فیلد داده^۱ از یک بسته TCP قرار گرفته و ارسال می‌شوند.
- **len**: طول داده‌های ارسالی یا دریافتی بر حسب بایت
- **flag**: برای پرهیز از پیچیدگی بحث در این مورد توضیح نمی‌دهیم. فقط در آن صفر بگذارید.
- **buf**: این پارامتر در تابع `recv()` آدرس محلی در حافظه است که داده‌های دریافتی در آنجا قرار گرفته و به برنامه باز گردانده می‌شود.

مقدار برگشتی این دو تابع در صورت بروز هر گونه خطا ۱- خواهد بود ولی در صورت برگشت یک عدد مثبت، تعداد بایتهای ارسالی یا دریافتی را بر حسب بایت مشخص می‌کند. دقت کنید که ممکن است تعداد بایتهای ارسالی یا دریافتی با تعدادی که در متغیر `len` تقاضا داده‌اید یکسان نباشد. بعنوان مثال فرض کنید شما در متغیر `len` مقدار ۱۰۰۰ قرار داده‌اید ولی مقداری که تابع برگردانده است ۲۰۰ باشد. در این صورت ۸۰۰ بایت از کل داده‌های ارسالی (یا دریافتی) باقی مانده است که برنامه شما باید تکلیف آنها را مشخص کند.

توصیه: در هر مرحله سعی کنید حجم داده‌هایی که توسط تابع `send()` ارسال می‌کنید حول و حوش یک کیلو بایت باشد.

نکته: توابع `send()`، `recv()` فقط برای ارسال و دریافت روی سوکتهای نوع استریم کاربرد دارد ولی اگر می‌خواهید به روش UDP و با سوکتهای دیتاگرام داده‌هایتان را ارسال کنید اندکی صبر کنید؛

۴-۶) توابع `close()` و `shutdown()`

تا زمانی که نیاز داشتید می‌توانید یک ارتباط را باز نگاه‌داشته و داده ارسال یا دریافت نمائید ولیکن همانند فایلها هرگاه نیازتان برطرف شد باید ارتباط را ببندید.

فرم کلی تابع `close()` بصورت زیر است:

`close(int sockfd);`

^۱ Payload

- **sockfd**: مشخصه سوکت مورد نظر. این سوکت همان مشخصه ای است که تابع `accept()` برگردانده است. دقت کنید که اگر `sockfd` مشخصه ای باشد که توسط تابع `socket()` برگشته است تمام ارتباطات معلق و منتظر نیز بسته خواهد شد.
- ارتباطی که توسط تابع `close()` بسته می شود دیگر برای ارسال و دریافت قابل استفاده نخواهد بود.
- هر گاه سوکتی را ببندید در حقیقت یکی از ارتباطات TCP را بسته اید و سیستم عامل می تواند بجای آن تقاضای ارتباط دیگری را قبول کرده، برای پردازش به صف ارتباطات معلق اضافه کند.
- راه دیگر بستن یک سوکت تابع `shutdown()` می باشد که فرم کلی آن بصورت زیر است:

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

- **sockfd**: مشخصه سوکت مورد نظر
- **how**: روش بستن سوکت که یکی از سه مقدار زیر را می پذیرد:
 - ♦ **مقدار صفر**: دریافت داده را غیر ممکن می سازد ولی سوکت برای ارسال داده، همچنان باز است. سیستم عامل بافر ورودی^۱ مربوط به آن سوکت را آزاد می کند.
 - ♦ **مقدار ۱**: ارسال داده را غیر ممکن می سازد در حالی که سوکت برای دریافت داده ها همچنان باز است. سیستم عامل بافر خروجی^۲ مربوط به آن سوکت را آزاد می کند.
 - ♦ **مقدار ۲**: ارسال و دریافت را غیر ممکن کرده سوکت کاملاً بسته می شود. این حالت دقیقاً همانند تابع `close()` عمل می نماید.
- همانند توابع قبلی در صورت بروز خطا مقدار برگشتی این توابع ۱- خواهد بود و متغیر سراسری `errno` شماره خطا را برای پردازش مشخص می کند.

۷) توابع مورد استفاده در برنامه مشتری (مبتنی بر پروتکل TCP)

- تا اینجا توابعی که معرفی شدند توابع پایه ای بودند که در سمت سرویس دهنده به نحوی استفاده می شوند. حال باید ببینیم در سمت مشتری چه توابعی مورد استفاده قرار می گیرند:
- الف)** ابتدا دقیقاً مانند برنامه سرویس دهنده یک سوکت بوجود بیاورید. برای اینکار از تابع `socket()` که در بخش قبلی معرفی شد استفاده کنید. تا اینجا هیچ تفاوتی برای بکارگیری این تابع در سمت سرویس دهنده و سمت مشتری وجود ندارد.
- ب)** در هنگام نیاز مستقیماً تقاضای برقراری ارتباط را به سمت سرویس دهنده بفرستید و آنقدر منتظر شوید تا این تقاضا پذیرفته شود. این عمل توسط تابع `connect()` انجام می شود که در ادامه توضیح داده خواهد شد.
- ج)** از توابع `send()` و `recv()` برای ارسال و دریافت داده ها استفاده کنید.

 \Inbound buffer
 \Outbound buffer

د) نهایتاً ارتباط ایجاد شده را توسط تابع `close()` یا `shutdown()` ببندید.

۷-۱) تابع `connect()`

برای برقراری ارتباط با یک سرویس دهنده از تابع `connect()` استفاده می‌شود و در صورتی که برنامه سرویس دهنده روی ماشین مورد نظر اجرا شده باشد و توابع `listen()` و `accept()` در برنامه فراخوانی شده باشند آنگاه نتیجه تابع `connect()` موفقیت آمیز خواهد بود. فرم کلی تابع `connect()` به صورت زیر است:

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

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

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

- **sockfd**: مشخصه سوکتی است که با فراخوانی تابع `socket()` بدست آمده است.
- **serv_addr**: استراکچری از نوع `sockaddr` است که قبلاً معرفی شد. در این استراکچر آدرس IP ماشین مقصد و آدرس پورت برنامه مقصد تعیین خواهد شد.
- **addrlen**: اندازه استراکچر قبلی را بر حسب بایت معرفی می‌کند و می‌توان براحتی در این پارامتر مقدار `sizeof(struct sockaddr)` قرار داد.

به این نکته دقت کنید که شما آدرس پورت خودتان را تنظیم نمی‌کنید بلکه سیستم عامل بطور خودکار یک شماره پورت تصادفی برای شما انتخاب می‌کند و مقدار این شماره برای برنامه سمت مشتری اصلاً مهم نیست چرا که وقتی شما به یک سرویس دهنده متصل می‌شوید و سرویس دهنده این تقاضا را می‌پذیرد پاسخ سرویس دهنده به همان آدرس پورتی خواهد بود که سیستم عامل برای سوکت انتخاب کرده است. در حقیقت برنامه شما بعنوان شروع کننده ارتباط، آدرس پورت خود را نیز به طرف مقابل اعلام می‌کند. در مقابل آدرس پورت برنامه سرویس دهنده قطعاً باید ثابت و مشخص باشد تا برنامه مشتریها بتوانند ارتباط را شروع نمایند. در صورت عدم موفقیت در برقراری یک ارتباط TCP مقدار برگشتی این تابع ۱- خواهد بود و متغیر `errno` شماره خطای رخ داده می‌باشد.

مثال ناتمام زیر تا حدودی این دیدگاه را به شما عرضه می‌کند:

```
#include <string.h>
```

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

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

```
#define DEST_IP "132.241.5.10"
```

```
#define DEST_PORT 23
```

```
main() {
```

```

int sockfd;
struct sockaddr_in dest_addr; /* will hold the destination addr */

if (( sockfd = socket(AF_INET, SOCK_STREAM, 0))!=NULL) {

    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);
    bzero(&(dest_addr.sin_zero), 8);        /* zero the rest of the struct */

    if ((connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr)))!=1) {
        .
        .
        .
    }
}

```

۸) ارسال و دریافت به روش UDP با سوکتهای دیتاگرام

توابع ارسال، دریافت و پذیرش برای سوکتهای نوع استریم کاربرد دارد. حال باید دید که به چه صورت می‌توان ارسال و دریافت را به روش UDP روی سوکتهای نوع دیتاگرام انجام داد.

• برنامه سمت سرویس دهنده

الف) یک سوکت از نوع دیتاگرام ایجاد کنید. این کار با فراخوانی تابع `socket()` با پارامتر `SOCK_DGRAM` انجام می‌شود.

ب) به سوکت ایجاد شده آدرس پورت مورد نظر را نسبت بدهید. (با تابع `bind()`)
ج) بدون هیچ کار اضافی می‌توانید منتظر دریافت داده‌ها بشوید. (تا موقعی که داده‌ای دریافت نشود ارسال معنی نمی‌دهد). وقتی داده‌ای دریافت و پردازش شد آدرس برنامه مبدا (آدرس IP و پورت) مشخص شده و ارسال امکان پذیر خواهد بود.

ارسال و دریافت روی سوکتهای نوع دیتاگرام بوسیله توابع `recvfrom()` و `sendto()` انجام می‌شود.
د) نهایتاً سوکت ایجاد شده را ببندید.

• برنامه سمت مشتری

الف) یک سوکت از نوع دیتاگرام ایجاد کنید. (با تابع `socket()` و پارامتر `SOCK_DGRAM`)
ب) هر گاه نیاز شد بدون هیچ کار اضافی داده‌هایتان را به سمت سرویس دهنده ارسال نمایید. تا وقتی که به سمت سرویس دهنده ارسال نداشته باشید، دریافت داده‌ها معنا نمی‌دهد چرا که شما برای سرویس دهنده شناخته شده نیستید مگر اینکه داده‌ای را ارسال نمائید. ارسال و دریافت را تا زمانی که نیاز است انجام بدهید.

ج) سوکت ایجاد شده را ببندید.

فرم کلی تابع ارسال داده مبتنی بر سوکتهای دیتاگرام بصورت زیر است:

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

- **sockfd** : مشخصه سوکت دیتاگرام که با تابع socket() بوجود آمده است.
- **msg** : آدرس محل قرارگرفتن پیام در حافظه که داده‌های ارسالی بایستی از آنجا استخراج شده و درون یک بسته UDP قرار گرفته و ارسال شود.
- **len** : طول پیام ارسالی بر حسب بایت
- **flags** : برای پرهیز از پیچیدگی بحث فعلاً آنرا صفر در نظر بگیرید.
- **to** : استراکچری از نوع sockaddr که قبلاً ساختار آنرا مشخص کردیم. در این استراکچر باید آدرس IP مربوط به ماشین مقصد و همچنین شماره پورت سرویس دهنده تنظیم شود.
- **tolen** : طول استراکچر sockaddr است که به سادگی می‌توانید آنرا به مقدار sizeof(struct sockaddr) تنظیم نمایید.

مقدار برگشتی این تابع همانند تابع send() تعداد بایتی است که سیستم عامل موفق به ارسال آن شده است. دقت کنید که اگر مقدار برگشتی (-1) باشد خطائی بروز کرده که می‌توانید شماره خطا را در متغیر سراسری errno بررسی نمایید. باز هم تکرار می‌کنیم دلیلی ندارد تعداد بایتی که تقاضای ارسال آنها را داده‌اید با تعداد بایتی که ارسال شده یکی باشد. بنابراین حتماً این مورد را در برنامه خود بررسی کرده و همچنین تقاضاهای ارسال در هر مرحله را نزدیک یک کیلو بایت در نظر بگیرید.

فرم کلی تابع دریافت داده مبتنی بر سوکتهای دیتاگرام بصورت زیر است:

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

- **sockfd** : مشخصه سوکت دیتاگرام که با تابع socket() بوجود آمده است.
 - **buf** : آدرس محلی از حافظه که سیستم عامل داده‌های دریافتی را در آن محل قرار خواهد داد.
 - **len** : طول پیامی که باید دریافت شود (بر حسب بایت)
 - **from** : استراکچری است از نوع sockaddr که قبلاً ساختار آن بررسی شد و سیستم عامل آنرا با مشخصات آدرس IP و آدرس پورت برنامه مبدا تنظیم و به برنامه شما برمی‌گرداند.
 - **flag** : آنرا به صفر تنظیم کنید.
 - **len** : طول استراکچری است که سیستم عامل آنرا برگردانده است.
- مقدار برگشتی این تابع نیز تعداد بایتی است که دریافت شده است. این پارامتر برای پردازش داده‌های دریافتی اهمیت حیاتی دارد.

۹) توابع مفید در برنامه نویسی شبکه

بغیر از توابع سیستمی معرفی شده توابع دیگری هم هستند که برای برنامه نویسی شبکه بسیار مفید و کارآمد هستند. در ادامه برخی از مهمترین آنها را تشریح خواهیم کرد:

۹-۱) تابع `getpeername()`

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

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

با استفاده از این تابع می‌توانید هویت طرف مقابل، شامل آدرس IP و آدرس پورت پروسه طرف مقابل ارتباط را استخراج نمایید. پارامترهای این تابع بصورت ذیل تعریف شده است:

- **sockfd**: مشخصه سوکت مورد نظر
- **addr**: استراکچری است از نوع `sockaddr` که قبلاً ساختار آن تعریف شده است. این استراکچر توسط سیستم عامل با آدرس IP و آدرس پورت طرف مقابل پر خواهد شد.
- **addrlen**: طول استراکچر `sockaddr`

در صورت عدم موفقیت تابع فوق مقدار برگشتی (-۱) خواهد بود و در متغیر `errno` شماره خطا برای بررسی نوع خطا تنظیم خواهد شد.

نکته ای که ممکن است برنامه نویس فراموش کند آن است که ترتیب آدرس IP و آدرس پورت بصورت BE است و اگر ماشین شما از نوع LE است باید حتماً آنرا از طریق توابعی که در بخش ۵ به آن اشاره شد تبدیل کنید.

۹-۲) تابع `gethostname()`

این تابع نام ماشینی را که برنامه شما روی آن اجرا می‌شود، بر خواهد گرداند. این نام یک رشته کاراکتری معادل با نام نمادین ماشین است نه آدرس IP آن (مثلاً `www.ibm.com`). فرم کلی تابع بصورت زیر است:

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

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

- **hostname**: یک آرایه از کاراکترها (یا عبارت بهتر یک رشته کاراکتری) است که پس از بازگشت تابع نام ماشین در آنجا ذخیره خواهد شد.
- **size**: طول رشته کاراکتری بر حسب کاراکتر

اگر مقدار برگشتی (-۱) باشد خطائی بروز کرده و مقدار `errno` همانند قبل شماره خطا را نگه می‌دارد ولی اگر تابع فوق موفق عمل کند مقدار برگشتی صفر خواهد بود.

۳-۹) بکارگیری سیستم DNS برای ترجمه آدرسهای حوزه

قبلاً در مورد سیستم DNS و طرز عملکرد آن بحث شد. در اینجا وقت آن فرا رسیده است که بتوانید در محیط برنامه نویسی تقاضای ترجمه نام حوزه^۱ یک سرویس دهنده را به این سیستم ارائه کرده و نتیجه را در برنامه خود استفاده نمایید.

مثالهای کوچکی که تا اینجا داشته‌ایم همگی برای برقراری یک ارتباط با ماشین خاص مستقیماً از آدرس IP آن استفاده می‌کردند و لیکن فرض کنید که شما بخواهید برنامه ای بنویسید که کاربر بتواند آدرس نام حوزه یک سرویس دهنده را بعنوان آدرس مقصد وارد نماید. تابعی که در این مورد بکار می‌آید دارای فرم کلی زیر است:

```
#include <netdb.h>
```

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

● **name** : رشته کاراکتری نام حوزه یک سرویس دهنده

مقدار برگشتی تابع ، آدرس استراکچری است از نوع hostent که ساختار آن بصورت زیر تعریف شده 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]
```

● **hname** : نام رسمی ماشین (برای شبکه اینترنت این رشته نام حوزه خواهد بود مثلاً (www.ibm.com

● **h_aliases** : نام مستعار ماشین (این رشته با ۱۰ ختم می‌شود)

● **h_addrtype** : خانواده آدرس (همانگونه که اشاره شد در شبکه اینترنت این فیلد مقدار AF_INET خواهد داشت).

● **h_length** : طول آدرس بر حسب بایت

● **h_addr_list** : یک رشته کاراکتری که در آن آدرس IP مربوط به ماشین سرویس دهنده قرار

^۱ Domain Name

دارد. این رشته با ۱۰ ختم می‌شود.

دقت کنید که در تابع بالا در صورت موفقیت آمیز بودن، یک اشاره گر به استراکچر بر می‌گرداند و در غیر اینصورت مقدار NULL برخواهد گشت و برخلاف توابع قبلی متغیر errno تنظیم نخواهد شد و بجای آن متغیر سراسری herror که متغیری سیستمی است تنظیم می‌شود و در ضمن تابع سیستمی error() برای کشف نوع خطا بکار گرفته می‌شود.

برای رفع ابهاماتی که در این تابع وجود دارد طرح یک مثال ضروری به نظر می‌رسد. به برنامه کوچک و اجرائی زیر دقت نمایید :

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.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;
}
```

نام این برنامه getip است که یک آدرس نام حوزه را بعنوان ورودی دریافت کرده و نتیجه ترجمه آن را به آدرس IP و بقیه مشخصات را روی خروجی چاپ می‌کند. نکات زیر درمورد برنامه بالا ارزش بازگوئی دارد:

الف) طریقه بکارگیری برنامه فوق بدینصورت است که نام برنامه را روی خط فرمان تایپ کرده و سپس در جلوی آن نام حوزه را با یک فاصله خالی نوشته و کلید Enter را فشار می‌دهید. مثال :

```
$ getip www.ibm.com
```

ب) آدرس IP معادل با آدرس نام حوزه در متغیر h_addr_list واقع است و هر چند که بصورت یک رشته است که با کد ۱۰ ختم می‌شود ولی برای شبکه اینترنت که آدرسهای IP فعلاً چهار بایتی هستند شما فقط به چهار بایت اول آن که بصورت BE ذخیره شده‌اند نیازمندید. در برنامه فوق برای تبدیل آدرس چهاربایتی به حالت رشته ای نقطه دار بفرم (مثلاً 213.190.140.187) از تابع inet_ntoa() برای چاپ روی خروجی بهره گرفته شده است.

ج) عمل "تطبیق نوع" در تابع inet_ntoa() به آن دلیل بوده است که طبق تعریف اصلی متغیر h_addr بصورت رشته معمولی تعریف شده ولی در تابع inet_ntoa() آرگومان ورودی آن یک استراکچر از نوع in_addr است که در ابتدای فصل ساختار آن تعریف شد و چهاربایتی است. بنابراین مجبوریم با عمل "تطبیق نوع" سازگاری پارامتر ورودی را تضمین کنیم ولی در عمل اتفاق خاصی نمی‌افتد.

۱۰ برنامه های نمونه

پس از معرفی توابع ساده برای برنامه نویسی شبکه دو مثال ساده به شما کمک می‌کند تا با بررسی آنها اشکالات و ابهامات خود را رفع نمائید.

۱۰-۱) مثالی از مبادله اطلاعات به روش TCP مبتنی بر سوکتهای استریم

در مثال اول یک سیستم ساده مبتنی بر مفهوم سرویس دهنده /مشرتی بررسی میشود که مطابق با آنچه گفته شد در دو برنامه مجزا باید نوشته شود : برنامه سمت سرویس دهنده و برنامه سمت مشتری. این مثال از سوکتهای نوع استریم استفاده می‌کند یعنی مبادله داده مبتنی بر روش TCP است. در ابتدا برنامه سمت سرویس دهنده را بررسی می‌نمائیم:

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

#define MYPOR 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;

if ((sockfd = socket(AF_INET, SOCK_STREAM, 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; /* auto-fill with my IP */
bzero(&(my_addr.sin_zero), 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);
}

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 */
        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 */
    while(waitpid(-1, NULL, WNOHANG) > 0); /* clean up child processes */
}
}
```

عملی که این برنامه ساده انجام می‌دهد آن است که هرگاه برنامه سمت مشتری با این برنامه و شماره پورت ۳۴۹۰ ارتباط برقرار کند پیغام "Hello, world!\n" را دریافت خواهد کرد. بنابراین برنامه سمت سرویس دهنده دقیقاً پس از `accept()` کردن یک ارتباط بدون هیچ پردازش خاصی رشته چهارده بایتی فوق را برای طرف مقابل فرستاده و سوکت متناظر را خواهد بست.

برنامه تا رسیدن به دستور `while(1)` نیاز به توضیح خاصی ندارد چرا که فقط یک سوکت نوع استریم ایجاد شده و به این سوکت آدرس پورت ۳۴۹۰ نسبت داده شده و با تابع `listen()` اجازه داده شده تا حداکثر ده ارتباط معلق پذیرفته شود و سپس وارد حلقه بینهایت شده است. پس از آنکه برنامه وارد حلقه `while(1)` شد ابتدا اولین ارتباط معلق (در صورت وجود) پذیرفته شده و مشخصه سوکت جدید برای آن ایجاد شده و به برنامه برگردانده می‌شود.

پس از این کار یک فراخوان سیستمی یونیکس به نام `fork()` برای ایجاد یک پروسه فرزند انجام می‌شود. بد نیست برای آشنایی بیشتر در این مورد توضیحی ارائه نمایم:

`fork()` تنها راه ایجاد یک پروسس جدید در محیط یونیکس است و وظیفه آن ساختن یک پروسس تکراری دقیقاً یکسان با پروسس اولیه شامل تمام مشخصه های فایل، رجیسترها و منابع دیگر است. پس از اجرای `fork()` پروسس اولیه و پروسس نسخه برداری شده راه جداگانه ای را در پیش خواهند گرفت. از آنجائیکه `fork()` بعنوان داده‌های پدر برای ساختن فرزند نسخه برداری می‌شوند، همه متغیرها در زمان `fork()` مقادیر یکسان دارند اما پس از آغاز پروسه فرزند تغییرات بعدی در هر کدام از آنها تاثیری بر روی دیگری نخواهد گذاشت (متن برنامه که غیر قابل تغییر است بین پدر و فرزند به اشتراک گذاشته می‌شود). تابع سیستمی `fork()` یک مقدار برمی‌گرداند که برای پروسس فرزند برابر صفر و برای پروسس پدر شناسه پروسه فرزند^۱ خواهد بود. با استفاده از `pid` بازگشتی می‌توان فهمید که بین دو پروسس کدامیک فرزند و کدام پدر است.

بنابراین در برنامه فوق به ازای هر ارتباط که پذیرفته می‌شود یک پروسه جدید که بعد از تابع سیستمی `fork()` شروع می‌شود بعنوان پروسه فرزند تولید شده و همانند دیگر پروسسها بصورت اشتراک زمانی از سیستم عامل سرویس می‌گیرد.

دلیل آنکه در برنامه فوق از این روش استفاده شده آن است که ارتباطات معلق بروش `Polling` پردازش نشوند بلکه بصورت همروند اجرا گردند. این کار باعث می‌شود که هر گونه تاخیر در یکی از ارتباطات بقیه را با تاخیر مواجه نکند بلکه به ازای هر ارتباط معلق یک پروسه فرزند ایجاد شود و همه در یک سطح بصورت اشتراک زمانی سهمی از زمان CPU را دریافت کرده و اجرا شوند. هر پروسه فرزند که به اتمام رسید یک پروسه فرزند جدید برای ارتباطی جدید ساخته می‌شود.

تابع `waitpid(-1, NULL, WNOHANG)` پروسه پدر را به حالت تعلیق خواهد برد تا زمانی که تمام پروسسهای فرزندش به اتمام برسند.

حال به برنامه سمت مشتری دقت نمایید. این برنامه با توجه به توضیحاتی که تا اینجا ارائه شده احتیاج به توضیح ندارد. برنامه سمت مشتری زمانی موفق عمل خواهد کرد که قبل از اجرای آن برنامه سمت سرویس دهنده اجرا شده باشد.

^۱ pid (Process Identifier)

```
#include <stdio.h>
#include <stdlib.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);
    bzero(&(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, 0)) == -1) {
        perror("recv");
    }
}
```

```

    exit(1);
}
buf[numbytes] = '\0';
printf("Received: %s",buf);

close(sockfd);

return 0;
}

```

۲-۱۰) مثالی از مبادله اطلاعات به روش UDP مبتنی بر سوکتهای دیتاگرام

ابتدا برنامه سمت سرورس دهنده را ارائه می‌نمائیم. این برنامه در سمت سرورس دهنده منتظر دریافت بسته‌ها باقی می‌ماند و هر گاه بسته‌ای را از یک مشتری دریافت کرد به همراه آدرس آن بر روی خروجی نمایش خواهد داد. برنامه نیاز به توضیح خاصی ندارد.

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
#define MYPOR 4950 /* the port users will be connecting to */
#define MAXBUFLEN 100

main()
{
    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(MYPOR); /* short, network byte order */

```

```

my_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */
bzero(&(my_addr.sin_zero), 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, 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);
}

```

در سمت مشتری ، برنامه رشته‌ای را که بعنوان آرگومان دریافت کرده ، مستقیماً برای سرویس دهنده ارسال می‌کند. بعنوان مثال اگر برنامه را با نام talker.c نوشته و سپس کامپایل و بصورت زیر در خط فرمان اجرا نماییم:

```
$ talker www.hserver.edu hello
```

رشته hello توسط برنامه به سمت سرویس دهنده ارسال خواهد شد و برنامه سمت سرویس دهنده طبق توضیحی که ارائه شد آنرا روی خروجی چاپ خواهد کرد.

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/wait.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);
    bzero(&(their_addr.sin_zero), 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;
}
```

(۱۱) بلوکه شدن^۱ پرونده‌های تمت شبکه

مفهوم بلوکه شدن یک پرونده از مباحث طراحی سیستم عامل است که نمی‌توان در اینجا کاملاً آنرا تشریح کرد ولی نکاتی از آن را که به مبحث برنامه‌نویسی سوکت مرتبط است توضیح می‌دهیم.

در یک عبارت ساده دستورات ورودی / خروجی یک پرونده در حال اجرا را متوقف کرده و تا زمانی که ورودی / خروجی آن کامل نشود و مجدداً از سیستم عامل برش زمانی دریافت نکند متوقف خواهد ماند. توابع `recv()` و `recvto()` و `accept()` از همین دسته هستند (یعنی به نوعی ورودی / خروجی محسوب می‌شوند) و بالطبع برنامه‌هایی که این توابع را اجرا نمایند توسط سیستم عامل بلوکه خواهند شد و تا کامل شدن عملیات ورودی / خروجی، بلوکه باقی می‌مانند.

بعنوان مثال تابع `accept()` یکی از ارتباطات معلق و به صف شده TCP را به برنامه شما تحویل می‌دهد. حال وقتی هیچ ارتباط معلق وجود ندارد یعنی هیچ ماشینی تقاضای برقراری ارتباط نداده است این تابع منجر به بلوکه شدن برنامه می‌شود تا زمانی که تقاضایی برسد، در این حالت سیستم عامل برنامه بلوکه شده را احیا کرده و اجرا می‌نماید. این روش کلاً بسیار مفید و کارآمد است و لیکن راهی وجود دارد که سیستم عامل پس از اجرای این تابع (وبقیه توابع) برنامه شما را بلوکه نکند. برای اینکار از فراخوان سیستمی `fcntl()` به نحوی که در مثال بعدی آمده است استفاده کنید. در این حالت بعد از فراخوانی توابع `accept()` یا `recv()` چه موفقیت آمیز و چه ناموفق برنامه شما بلوکه نخواهد شد بلکه خود برنامه نویس موظف است در برنامه خود امکان پذیرش ارتباط یا دریافت داده‌ها را بررسی نماید. در حقیقت این روش همان روش سرکشی^۲ است که در محیطهای چند کاربره روش مناسبی محسوب نمیشود چرا که در این روش برنامه شما در یک حلقه بی‌نهایت وقت CPU را گرفته و پشت سرهم سوکتها را سرکشی می‌نماید.

دقت کنید که اگر نتیجه `accept()` مقدار (-۱) باشد، برنامه شما توسط سیستم عامل بلوکه نمی‌شود و بنابراین اگر سعی کنید داده‌ای را دریافت یا ارسال کنید با خطای سیستمی و قطع نامتعادل^۳ برنامه مواجه خواهید شد. در برنامه‌هایی که بدین نحو نوشته می‌شوند، آزمایش مقدار برگشتی تابع `accept()` برعهده برنامه‌نویس خواهد بود.

برنامه‌نویسی تحت شبکه، ابزارهای بهتر و قوی‌تری نسبت به زبان معمولی C دارد، ولیکن ارائه مفاهیم سوکت و توابع لازم برای برنامه‌نویسی تحت شبکه، با استفاده از زبان C، مفاهیم را بهتر و بنیادی‌تر آموزش می‌دهد، زیرا برای ارائه مفهوم سوکت و برنامه‌نویسی تحت شبکه با زبانهای شیء‌گرا، مجبور خواهیم بود حجم بسیار زیادی از کدهای یک شیء را در زبانی مثل جاوا بررسی و تحلیل کنیم.

در بخش آتی سعی خواهد شد ضمن معرفی زبان جاوا، قابلیت‌های شبکه‌ای این زبان، بصورت فهرست‌وار مرور شود.

^۱ Blocking

^۲ Polling

^۳ Abnormaly Ending

۱۲) امکانات زبان JAVA در برنامه‌نویسی شبکه

۱-۱۲) مقدمه

جاوا یک زبان برنامه‌نویسی شیء‌گراست که می‌توان گفت بطور مستقیم از C و C++ گرفته شده است و اهدافی مثل "عدم وابستگی به ماشین اجرا"^۱، که C++ در عمل نتوانست بدان دست یابد را به نحو زیبایی پیاده‌سازی کرده است. یعنی بدون هیچ دغدغه‌ای می‌توان بر روی یک ماشین مبتنی بر سیستم عامل MS-Windows برنامه‌ای به زبان جاوا نوشت و آنرا بر روی ماشینی مبتنی بر یونیکس اجرا کرد. این قابلیت در واقع به نوعی نیاز شبکه اینترنت محسوب می‌شد و باعث شد تا جاوا در دوران اوج زبان C++، ناگهان نگاهها را معطوف خود کند و همانند وب در عرض چند سال به ابزاری مطمئن برای برنامه‌نویسی شبکه تبدیل شود.

بزرگترین ضعف برنامه‌های نوشته شده به زبان C++، آن دسته از "اشکالات پنهان"^۲ است که در اثر آزادی برنامه‌نویس در مدیریت حافظه و کار با اشاره‌گرها، در برنامه پدید می‌آید. جاوا با حذف اشاره‌گرها و تقبل مدیریت حافظه، این دو ضعف را برطرف کرد و به یک زبان برنامه‌نویسی امن مبدل شد.

هنگامیکه مهندسین شرکت سان توجه خود را به پروژه **گرین**^۳ معطوف کردند تا برای لوازم الکترونیکی این شرکت نرم‌افزار پیشرفته بسازند، دریافتند که کامپایلرهای C و C++ برای اینکار نارسایی دارند، از اینرو به فکر خلق زبانی جدید افتادند که در ابتدا آاک Oak نام گرفت و پس از مدتی به جاوا تغییر نام داد. به دنبال این پیشرفت، شرکت سان برای جاوا یک مرورگر ساخت که می‌توانست در محیط وب قطعه برنامه‌های جاوا را اجرا کند.

جاوا زبانی است ساده، ایمن، قابل حمل، شیء‌گرا، توانمند در حمایت از برنامه‌های "چند ریسمانی"^۴، با "معماری ختشی"^۵، که با زبانهای C و C++ تفاوتی دارد. این تفاوتها را می‌توان در موارد زیر خلاصه کرد:

- **اشاره‌گرها**: همانطور که قبلاً نیز اشاره شد در جاوا اشاره‌گری وجود ندارد، این درحالیست که در C/C++ می‌توان از اشاره‌گر استفاده کرد. این امر باعث می‌شود نتوان حافظه را بخوبی مدیریت کرد؛ هرگونه استفاده نامناسب در بکارگیری اشاره‌گرها در برنامه‌های C/C++، می‌تواند حداقل برنامه را متوقف کند.

- **استراکچرها و یونیون‌ها**^۶: در زبان C++ سه نوع از "انواع داده"^۷ وجود دارد: کلاس، استراکچر و یونیون، در حالیکه جاوا فقط شامل کلاس است. در یک زبان برنامه‌نویسی شیء‌گرا مثل C++ وجود "کلاس"، برنامه‌نویس را از داده‌هایی نظیر استراکچر و یونیون بی‌نیاز می‌کند، ولی C++

^۱ Platform Independence

^۲ Bug

^۳ Green

^۴ Multithread

^۵ Achitecture-neutral

^۶ Union

^۷ Data Type

برای سازگاری با C مجبور بود از آنها پشتیبانی کند در حالی که در جاوا هیچ الزامی در تعریف آنها وجود نداشت.

- **توابع:** جاوا هیچ تابعی ندارد، چون شیء‌گرا است و برنامه‌نویس را مجبور به استفاده از متدهای کلاس^۱ می‌کند، در حالیکه در C++ به غیر از کلاس، توابع نیز تعریف شده‌اند، که چندان با مفهوم شیء‌گرایی مطابقت ندارد.

- **وراثت چندگانه^۲:** وراثت چندگانه به این معناست که یک کلاس را از چند کلاس دیگر مشتق کنیم که این عمل در جاوا براحتمی امکان‌پذیر است، در حالیکه در C/C++ این کار بسیار مشکل بوده و باعث پیچیدگی و خطا می‌شود.

- **رشته‌ها:** در جاوا، رشته‌ها را بعنوان اشیاء کلاس اولیه داریم در حالیکه در C/C++ ساختاری شیء‌گرا برای پشتیبانی رشته‌های متنی نداریم.

- **دستور goto:** در C/C++ این دستور کمابیش استفاده می‌شود ولی در جاوا اگرچه این دستور جزو کلمات کلیدی است ولی استفاده از آن پشتیبانی نمی‌شود. عدم پشتیبانی از دستورات پرش غیرساختاریافته، باعث کاهش خطا در جاوا شده است.

- **Operator overloading:** در جاوا بر خلاف C/C++ از توانایی تغییر عملکرد اپراتورها پشتیبانی نمی‌شود تا پیچیدگی زبان کمتر شود.

- **تبدیل خودکار نوع^۳:** در زبان C/C++ شما می‌توانید یک متغیر را از نوعی مثل float تعریف کنید و سپس مقداری مثل int به آن نسبت بدهید، ولی اگر این عمل را در زبان جاوا انجام دهید فوراً با پیغام خطا مواجه خواهید شد. اینگونه سخت‌گیرها، امنیت ذاتی جاوا را افزایش چشمگیر داده است.

- **آرگومانهای خط فرمان^۴:** C/C++ دو پارامتر argc و argv را به برنامه ارسال می‌کند که تعداد آرگومانهای ذخیره شده در argv را مشخص می‌کند، در حالیکه argv یک اشاره‌گر به آرایه‌ای از کاراکترهاست. با حذف اشاره‌گرها در جاوا، به جای argv از args استفاده شد؛ [0]args اولین پارامتر خط فرمان است.

- برای تاکید بیشتر تکرار می‌شود که مشکل عمده C/C++ اشاره‌گرها و مدیریت حافظه است در حالیکه مدیریت حافظه در جاوا بصورت خودکار انجام می‌شود. برای مشخص شدن قضیه به این نکته دقت کنید که وقتی در C++ یک بلوک حافظه یا یک کلاس را new() می‌کنید، خودتان موظف به آزادسازی آن هستید و انجام ندادن این کار یک اشکال محسوب می‌شود. پاکسازی حافظه از اشیاء بی‌مصرف برعهده خود جاوا است. کار با آرایه‌ها در جاوا بسیار آسانتر و مطمئن‌تر است چون آرایه‌ها در این زبان، عضوی از یک کلاس می‌باشند.

- C++ از اصول شیء‌گرایی به موازات برنامه‌نویسی به سبک قدیم حمایت می‌کند که این حالت

^۱ Method

^۲ Multiple Inheritance

^۳ Automatic Conversion

^۴ Command-line Arguments

در جاوا وجود ندارد و جاوا صد در صد شیء گراست.

● کامپایلر جاوا یک برنامه نوشته شده را به کدهای اجرایی یک ماشین خاص مثل IBM یا Apple تبدیل نمی کند ، بلکه آنرا به کدهای اجرایی یک ماشین فرضی به نام JVM^۱ ترجمه می کند که مختص به هیچ پردازنده ای نیست ، بلکه زبان اسمبلی یک ماشین مجازی است. به کدهای اجرایی این ماشین مجازی "بایت کد"^۲ گفته می شود. بنابراین نتیجه ترجمه یک برنامه جاوا یک فایل میانی حاوی بایت کد است. هر ماشین که بخواهد یک برنامه جاوا را اجرا کند موظف است از "مفسر زمان اجرای جاوا" استفاده کند تا دستورات مجازی JVM به کدهای اجرایی واقعی از یک ماشین تبدیل شود. هر ماشین برای خودش JVM خاص دارد. بدین گونه جاوا فارغ از ساختار ماشین ، ترجمه و اجرا می شود. نحوه اجرای برنامه های کاربردی جاوا در یک ماشین بصورت زیر است:

برنامه های کاربردی جاوا
اشیاء جاوا (JAVA Objects)
ماشین مجازی جاوا (JVM)
سیستم عامل

ماشین مجازی جاوا (JVM) دارای پنج بخش مهم زیر می باشد :

● **مجموعه دستورات بایت کد** : بایت کدها به عنوان دستورالعملهای اجرایی (ولی مجازی) شامل دو قسمت عملوند و عملگر می باشند. هر نوع داده اولیه در جاوا یک بایت کد مخصوص به خود دارد. این دستورالعملهای مجازی توسط JVM به یک یا چند دستورالعمل اجرایی از یک ماشین تبدیل می شوند.

● **مجموعه رجیسترها** : مجموعه رجیسترها در ماشین مجازی جاوا ، همگی ۳۲ بیتی هستند.

● **پشته**^۳ : پشته در JVM دقیقاً مثل پشته هایی است که در دیگر زبانهای برنامه نویسی برای ارسال پارامتر به توابع و ذخیره متغیرهای محلی^۴ از آن استفاده می شود. هر متود از یک کلاس در زبان جاوا برای خود پشته ای دارد که متغیرهای محلی متود ، محیط اجرای آن و پارامترهای ارسالی به متود ، در این پشته قرار می گیرند.

● **فضای کاری**^۵ : فضای کاری برنامه JVM ، قسمتی از حافظه است که برنامه نویس قادر به دخالت در آن نیست ، بلکه خود کامپایلر ، حافظه این قسمت را مدیریت می کند و در صورت لزوم فضایی را تخصیص داده یا آنرا آزاد می کند. یعنی دست برنامه نویس از دسترسی غیر مجاز به حافظه کوتاه شده و عمل تخصیص و آزادسازی فضای مورد نیاز حافظه به کامپایلر محول شده است.

● **فضای ذخیره سازی متدها** : فضای متدهای JVM یک فضای ۸ بیتی است که در قسمت خاصی از حافظه ذخیره می شود.

^۱ Java Virtual Machine

^۲ Bytecode

^۳ Stack

^۴ Local Variable

^۵ Work Space

۲-۱۲) انواع داده در جاوا

جاوا ۸ نوع داده اصلی دارد که در جدول (۲-۷) فهرست شده‌اند. هر نوع، یک اندازه مشخص بر حسب بایت دارد. برخلاف زبان C که برای داده نوع صحیح بسته به نوع معماری ماشین، ۱۶، ۳۲ یا ۶۴ بیت در نظر گرفته می‌شود، زبان جاوا برای این نوع داده فقط ۳۲ بیت در نظر می‌گیرد که این نکته مزایایی برای زبان جاوا به وجود می‌آورد. یکی از این مزایا آن است که باعث می‌شود برنامه روی انواع ماشینهای ۱۶، ۳۲ و ۶۴ بیتی به یک شکل کار کند و نتیجه واحد ارائه دهد.

نوع	اندازه	توضیح
byte	1 Byte	یک عدد صحیح علامت‌دار با محدوده -۱۲۸ تا +۱۲۷
short	2 Byte	یک عدد صحیح علامت‌دار دو بایتی
int	4 Byte	یک عدد صحیح علامت‌دار چهار بایتی
long	8 Byte	یک عدد صحیح علامت‌دار هشت بایتی
float	4 Byte	یک عدد اعشاری چهار بایتی با استاندارد IEEE
double	8 Byte	یک عدد اعشاری هشت بایتی با استاندارد IEEE
boolean	1 Bit	یک پرچم تک بیتی که دو حالت True یا False دارد
char	2 Byte	یک تک‌کاراکتر یونی‌کد (دو بایت)

جدول (۲-۷) انواع داده اصلی در جاوا

در زبانهای شیء‌گرا نظیر جاوا هر چیزی یک شیء است. هر شیء دارای مجموعه‌ای از رفتار^۱ و صفات^۲ است و یکسری متود^۳ نیز برای دسترسی به اشیاء وجود دارد. رفتار تنها راه برای این است که به شیء بگوییم چه عملی را انجام دهد. برای اینکه رفتار یک شیء را بسازید باید ابتدا یک متود ایجاد کنید که این متودها شبیه به توابع در دیگر زبانها می‌باشند. برخلاف C++، جاوا توابعی که خارج از کلاسها و جداگانه تعریف شده باشند، ندارد. اگر یک شیء را تعریف کردید می‌توانید تعیین کنید که چه برنامه‌هایی و با چه سطحی از دسترسی به این شیء می‌توانند وجود داشته باشند. یک برنامه جاوا، شامل یک یا چند بسته^۴ می‌باشد که هر کدام از این بسته‌ها خود شامل تعاریف کلاسها هستند و توسط بقیه برنامه‌ها قابل استفاده می‌باشند.

اشیاء در جاوا می‌توانند به صورت پویا و در حین اجرای برنامه تولید شوند. هر کلاس می‌تواند زیر کلاس یا کلاس والد (Super Class) داشته باشد. این کلاس همیشه از کلاس والد خصوصیات و

^۱ Behavior
^۲ Attributes
^۳ Method
^۴ Package

رفتارها و روشها را به ارث می برد . البته همیشه نمی توان به متغیرهای داخلی کلاس والد دسترسی مستقیم داشت. دسترسی مستقیم به متغیرهای کلاس والد ، به شرطی امکان پذیر است که آن متغیرها بصورت public تعریف شده باشند.

حال به ارائه مثالی می پردازیم که مفهومی از شیء گرایسی را در خود دارد. در ارائه مثال فرض کرده ایم که با اصول ++C آشنایی دارید. در این مثال ، یک بسته (package) داریم که دو کلاس را مشخص می کند. یک عدد مختلط را در نظر بگیرید؛ می دانید که این نوع عدد شامل دو قسمت حقیقی و موهومی^۱ است:

```
class ComplexNumber {
// یک شیء برای متغیر مختلط ایجاد می شود.
protected double re, im;
// تعریف قسمتهای حقیقی و موهومی ( این قسمتها در خارج از کلاس در دسترس نیستند و مخفیند )
// پنج متود زیر متغیرهای پنهان بالا را استفاده و پردازش می کنند.
public void Complex (double x, double y) { re=x; im=y;}
public double Real() {return re;}
public double Imaginary() { return im; }
public double Magnitude ( ) { return Math.sqrt (re*re+im+im); }
public double Angle() {return Math.atan(im/re);}
class test {
// تعریف یک کلاس جدید برای استفاده از کلاس بالا
public static void main (String aras[ ] ) {
ComplexNumber C;
// تعریف یک شیء از نوع متغیر مختلط با تعریف بالا
C=new ComplexNumber();
// شیء از نوع متغیر مختلط در حافظه تولید می شود.
C. Complex (3.0, 4.0);
// مقداردهی اولیه به یک شیء از نوع متغیر مختلط
System.out.println ("The magnitude of C is " + C. Magnitude() );
}
}
```

در این مثال هر شیء شامل دو متغیر re و im می باشد که هر دو اعداد اعشاری ۶۴ بیتی هستند. این متغیرها نمی توانند توسط کلاسهای دیگر دستکاری شوند. (یعنی قابل دسترسی نیستند ، چون از نوع protected تعریف شده اند.) اگر این متغیرها را از نوع public تعریف می کردیم برای هر بسته در هر جا این متغیرها قابل رویت و دسترسی بودند و این حالت مطلقاً مفید نیست.

^۱ Real & Imaginary Part

حال به مثالی دیگر توجه کنید:

```
class Factorial {
public static void main(int arge, String args[ ]) // بدنه برنامه اصلی
long i, f, lower=1, upper=20; // تعریف چهار متغیر صحیح چهار بایتی
    for (i=lower ; i<=upper ; i<=upper; i++) { // C++ مشابه با زبان
        F= factorial(i); //f=i!
        System.out.println (i+ " " +f); // print i and t
    }
}

static long factorial (long k) { // تعریف یک تابع خود فراخوان فاکتوریل
    if (k ==0)
        return 1; // 0! = 1
    else
        return k * factorial (k-1); // k! = k+ (k-1)! }
}
```

در مثال بالا ابتدا کلاس اصلی برنامه به نام Factorail تعریف شده و سپس تابع factorial به عنوان متودی از این کلاس تعریف گردیده است. در متود اصلی کلاس برنامه که همیشه main نام دارد، متود factorial فراخوانی شده است.

۱۲-۳ اپلت Applet

اپلت ریزبرنامه یا برنامه کوچکی است که درون یک صفحه وب قرار می‌گیرد و روی یک سرویس‌دهنده اینترنت قابل دسترسی بوده و به عنوان بخشی از یک سند وب بر روی ماشین مشتری اجرا می‌شود. (البته به شرطی که مرورگر مجهز به مفسر جاوا^۱ باشد می‌توان آن را مشاهده کرد). اپلت‌ها با برچسب APPLET درون صفحه وب تعریف می‌شوند ولی فایل‌های خارجی به حساب می‌آیند. چون اپلت جهت استفاده در محیط وب نوشته می‌شود لذا کمی پیچیده‌تر از یک برنامه معمولی است. هنگامیکه می‌خواهید یک اپلت را در صفحه وب قرار دهید باید ابتدا تمام کلاسهای مورد نیاز آن اپلت را بسازید، سپس آنرا کامپایل کرده و بعد با استفاده از زبان HTML یک صفحه وب ساخته و اپلت را درون صفحه وب تعریف کنید. چون اپلتها دارای خط فرمان نیستند، برای فرستادن آرگومانهای متفاوت باید از برچسب <APPLET> استفاده کنیم.

دو راه برای اجرای یک اپلت وجود دارد:

- اجرا نمودن اپلت داخل یک مرورگر سازگار با جاوا مثل Netscape Navigator
- استفاده از Applet Viewer که این برنامه، اپلت را خارج از مرورگر و در یک پنجره، اجرا می‌کند، که برای اشکال‌زدائی از اپلتها راهی سریع و آسان محسوب می‌شود.

^۱ Java enabled browser

اپلت یک برنامه اجرایی است و برای اجرا در محیط مرورگر در نظر گرفته شده تا قابلیت‌هایی که صفحات وب ندارند از طریق آنها فراهم شود. اپلتها به همراه صفحات وب برای کاربران وب، ارسال و روی ماشین کاربر اجرا می‌شود. این برنامه اجرایی نباید عمداً یا سهواً قادر باشد صدمه‌ای به سیستم کاربر وارد کند؛ لذا اپلتها در مقایسه با برنامه‌های معمولی که به زبان جاوا نوشته می‌شوند، دارای محدودیت‌های زیر است:

- اپلت جز در موارد محدود و تحت نظارت شدید و آنهم برای خواندن، قادر به دسترسی به سیستم فایل نیست.
- اپلت قادر به فراخوانی و اجرای هیچ برنامه‌ای روی ماشین اجراکننده خود نیست.

در برنامه‌های کاربردی، بدنه برنامه اصلی با بلوک `main()` شروع می‌شود و در نهایت با علامت `}` پایان می‌پذیرد، ولی اپلتها در جاوا متود `main()` ندارند. صورت کلی بدنه یک اپلت به شکل زیر تعریف می‌شود:

```
public class Example extends java.applet.Applet {
...
}
```

با این تعریف، کلاس `Example`، کلاس از پیش تعریف شده `Applet` را به ارث برده و تمام مقدماتی را که یک اپلت برای فعل و انفعال با مرورگر دارد، فراهم می‌آورد. چندین اپلت می‌توانند بصورت مستقل در یک صفحه وب (روی مرورگر) اجرا شوند.

وقتی یک کلاس را به صورت اپلت تعریف می‌کنید، چندین متود اصلی و بنیادی را به ارث می‌برد که این متودها به خودی خود کاری انجام نمی‌دهند. برای آنکه یک اپلت عملیاتی شود، برنامه‌نویس باید این متودها را "باطل و دوباره‌نویسی"^۱ کند. پنج مورد از حیاتی‌ترین این متودها عبارتند از:

```
init() start() stop() destroy() paint()
```

دو متودی که بیش از بقیه احتیاج به دوباره‌نویسی دارند متودهای `init()` و `paint()` می‌باشند.

- متود `paint()` یکی از مهم‌ترین متودهای هر اپلتی است که برنامه‌نویس بدان احتیاج دارد. هر چیزی که بخواهد در پنجره خروجی اپلت نمایش داده شود، با استفاده از این متود امکان‌پذیر خواهد بود. متود `paint()` فقط یک آرگومان می‌پذیرد و بازنویسی آن بصورت زیر است:

```
public void paint(Graphics screen) {
// display statements go here
}
```

در مثال بالا آرگومان ورودی متود، یک شیء گرافیکی است. کلاس `Graphics` از اشیایی تشکیل شده که می‌توانند همه صفات و رفتارها را که نیاز است به عنوان متن، گرافیک و بقیه اطلاعات روی

^۱ Override

پنجره ، نمایش داده شوند کنترل کنند. اگر شما از یک شی Graphics در اپلتان استفاده می کنید ، دستور import را قبل از تعریف class در ابتدای فایل برنامه بیاورید:

```
import java.awt.Graphics;
```

- متود **init()** فقط یکبار و آنهم هنگام بارگذاری اپلت در مرورگر ، اجرا می شود؛ بنابراین متود **init()** در واقع برای تنظیم کردن و مقداردهی اولیه به اشیاء و متغیرهایی که در طول اجرای اپلت مورد نیازند استفاده می شود. به عنوان یک پیشنهاد ، این متود جایی مناسب برای تنظیم نوع قلم (فونت) ، رنگ قلم و رنگ پس زمینه صفحه می باشد.

- متود **start()** : با این متود ، اپلت راه اندازی شده و آغاز به کار خواهد کرد. اگر اپلت با استفاده از متود **stop()** موقتاً متوقف شده باشد ، با این متود از سرگرفته می شود. عملیاتی که برای راه اندازی یک اپلت مورد نیاز است ، در این متود دوباره نویسی می شود.

- متود **stop()** : هنگامی که این متود صدا زده شود ، اجرای اپلت موقتاً متوقف خواهد شد. زمانیکه کاربر یک صفحه وب (شامل اپلت) را رها می کند و به سراغ صفحه ای دیگر می رود ، این متود بطور خودکار فراخوانی می شود. (البته می توان این متود را به صورت مستقیم صدا زده و اپلت را متوقف کرد.) برنامه نویس تمهیدات لازم برای توقف اپلت را با دوباره نویسی این متود ، فراهم می آورد.

- متود **destroy()** : این متود درست برخلاف متود **init()** ، به منظور پایان دادن به اجرای اپلت فراخوانی می شود. برنامه نویس موظف است کارهایی را که باید در هنگام خاتمه اپلت انجام شود ، در این قسمت دوباره نویسی کند.

مثلاً فرض کنید بخواهیم یک اپلت بنویسیم که در محیط مرورگر اجرا شده و پیغام ساده Hello Web! بر روی پنجره آن نمایش یابد. چنین اپلتی فقط نیاز به بازنویسی متود **paint()** دارد تا بتواند روی پنجره خروجی پیغام را نمایش بدهد:

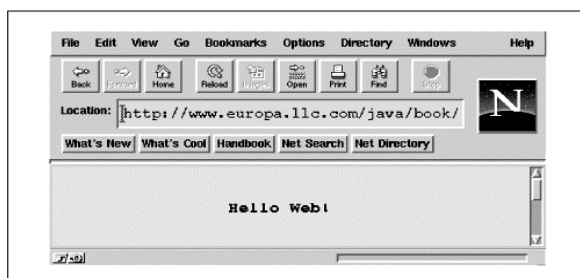
```
import java.awt.Graphics;
public class HelloWeb extends java.applet.Applet {
    public void paint( java.awt.Graphics gc ) {
        gc.drawString("Hello Web!", 125, 95);
    }
}
```

برای اجرای یک اپلت ، لازم است صفحه وبی ایجاد کنید که اپلت را بارگذاری کند. برای ایجاد یک صفحه وب ، یک فایل جدید روی ویرایشگر معمولی باز کرده و پس از نوشتن یک صفحه وب ساده همانند زیر ، آنرا با پسوند **.html** ذخیره کنید. سپس آنرا در محیط مرورگر تان باز کنید:

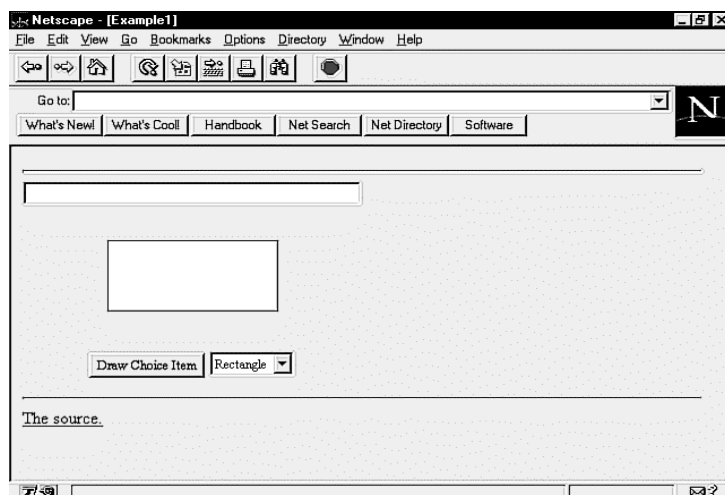

```
<html>
<head> </head>
<body>
<applet code=HelloWeb width=300 height=200>
</applet>
</body>
</html>
```

در قطعه کد بالا، پارامتر width، طول پنجره و پارامتر height، ارتفاع پنجره نمایش اپلت را مشخص می‌کند. (در فصل دهم در مورد صفحات وب بیشتر خواهیم آموخت.) خروجی اپلت بالا به صورت شکل (۷-۳) خواهد بود. به عنوان مثالی دیگر در شکل (۷-۴)، یک اپلت با پنج عامل زیر در محیط مرورگر نشان داده شده است:

- ۱- فیلد ورود داده‌های متنی TextField ۲- پانل Panel ۳- منوی انتخاب Choice menu
 ۴- صفحه ترسیم گرافیک Canvas ۵- کلید فشاری Button



شکل (۷-۳) خروجی یک اپلت نمونه در محیط مرورگر



شکل (۷-۴) خروجی یک اپلت نمونه با پنج عامل کنترل در محیط مرورگر

```
import java.awt.*;
import java.lang.*;
import java.io.*;
import java.applet.*;

// This program illustrates a simple applet with a TextField,
// Panel, Button, Choice menu, and Canvas.

public class Example1 extends Applet {
    TextField tf;
    DrawCanvas c;
    Button drawBtn;
    Choice ch;
    // Add the Components to the screen..
    .
    public void init() {
        // Set up display area...
        resize(300,200);
        setLayout(new BorderLayout());

        // Add the components...
        // Add the text at the top.
        tf = new TextField();
        add("North",tf);

        // Add the custom Canvas to the center
        c = new DrawCanvas(this);
        add("Center",c);

        // Create the panel with button and choices at the
        bottom...
        Panel p = new Panel();
        drawBtn = new Button("Draw Choice Item");
        p.add(drawBtn);
        // Create the choice box and add the options...
        ch = new Choice();
        ch.addItem("Rectangle");
        ch.addItem("Empty");
        ch.addItem("Text");
        p.add(ch);
        add("South",p);
    }

    // Handle events that have occurred
    public boolean handleEvent(Event evt) {
        switch(evt.id) {
            // This can be handled
            case Event.ACTION_EVENT: {
                if(evt.target instanceof Button) {
                    // Repaint canvas to use new choices...
                    c.repaint();
                } // end if
                return false;
            }
        }
    }
}
```

```

default:
return false;
}
}
// Return the current choice to display...
public String getChoice() {
return ch.getSelectedItemAt();
}

// Return the text in the list box...
public String getTextString() {
return tf.getText();
}
}

// This is a custom canvas that is used for drawing
// text, a rectangle, or nothing...
class DrawCanvas extends Canvas {
Example1 e1app;
// Constructor - store the applet to get drawing
info...
public DrawCanvas(Example1 a) {
e1app = a;
}
// Draw the image per the choices in the applet...
public synchronized void paint (Graphics g) {
// Get the current size of the display area...
Dimension dm = size();
// Draw based on choice...
String s = e1app.getChoice();
// Calculate center coordinates...
int x,y,width,height;
x = dm.width/4;
y = dm.height / 4;
width = dm.width / 2;
height = dm.height / 2;
// Paint a rectangle in the center...
if (s.compareTo("Rectangle") == 0) {
// Draw the rectangle in the center with colors!
g.setColor(Color.blue);
g.drawRect(x,y,width,height);
g.setColor(Color.yellow);
g.fillRect(x + 1,y + 1,width - 2,height - 2);
} // end if
// Get the text in the applet and display in the
middle...
if (s.compareTo("Text") == 0) {
String displayText = e1app.getTextString();
g.setColor(Color.red);
g.drawString(displayText,x,y + (height/2));
}
}
}
}

```

۴-۱۲) امکانات جاوا برای برنامه‌نویسی سوکت

`java.net` یک بسته از بسته‌های جاوا است که شامل کلاس‌هایی برای کار با شبکه، سوکتها و URLهاست. در این بسته دو نوع کلاس سوکت استریم برای برنامه‌نویسی شبکه تعریف شده است:

- کلاس `Socket`: کلاسی جهت برقراری ارتباط و مبادله داده در سمت مشتری است.
- کلاس `ServerSocket`: کلاسی جهت تعریف ارتباط و مبادله داده در سمت سرویس‌دهنده است.

ابتدا کلاس `Socket` را مورد بررسی قرار می‌دهیم. در این کلاس چندین متود تعریف شده که مهمترین آنها در زیر معرفی می‌شوند:

`Socket(String host, int port)`

`Socket(InetAddress address, int port)`

`synchronized void close()`

`InputStream getInputStream()`

`OutputStream getOutputStream()`

متودهای اول و دوم در حقیقت متودهای "سازنده"^۱ کلاس نوع `Socket` هستند که دارای دو تعریف مجزا بوده و می‌توانند به دو صورت استفاده شوند:

الف) `Socket(String host, int port)`: برای ایجاد کلاس سوکت از طریق این متود، مستقیماً آدرس نام حوزه یک ماشین و شماره پورت سرویس‌دهنده مورد نظر در آرگومانهای آن مشخص می‌شود. اگر از این متود برای خلق یک سوکت استفاده شود، قبل از بوجود آمدن آن، نام حوزه بصورت خودکار توسط DNS ترجمه شده و آدرس IP آن باز خواهد گشت و در صورت موفقیت‌آمیز نبودن این عمل، ادامه کار ممکن نخواهد بود.

مثال:

```
try {
    Socket sock = new Socket("cs.wustl.edu", 25);
}
catch ( UnknownHostException e ) {
    System.out.println("Can't find host.");
}
catch ( IOException e ) {
    System.out.println("Error connecting to host.");
}
```

ب) `Socket(InetAddress address, int port)`: برای ایجاد کلاس سوکت از طریق این متود، آدرس

^۱ Constructor

IP یک ماشین و شماره پورت سرویس دهنده مورد نظر در آرگومانهای آن مشخص می شود. آدرس IP به صورت یک رشته مثل "192.34.221.108" به متود ارسال می شود.

مثال :

```
try {
    Socket sock = new Socket("128.252.120.1", 25);
}
catch ( UnknownHostException e ) {
    System.out.println("Can't find host.");
}
catch ( IOException e ) {
    System.out.println("Error connecting to host.");
}
```

ج) متود `close()` : این متود ضمن ختم ارتباط TCP بصورت دو طرفه ، سوکت را بسته و منابع تخصیص داده شده را آزاد خواهد کرد.

د) متودهای `getInputStream()` و `getOutputStream()` برای آنست که بتوان استریمهای ورودی و خروجی نسبت داده شده به سوکت را بدست آورده و بتوان از آن خواند یا در آن نوشت.^۱ به مثال زیر دقت کنید:

```
try
{
    Socket server = new Socket("foo.bar.com", 1234);
    InputStream in = server.getInputStream();
    OutputStream out = server.getOutputStream();
    // Write a byte
    out.write(42);
    // Say "Hello" (send newline delimited string)
    PrintStream pout = new PrintStream( out );
    pout.println("Hello!");
    // Read a byte
    Byte back = in.read();
    // Read a newline delimited string
```

^۱ دقت کنید که مفهوم استریم ورودی و خروجی (Input Stream/Output Stream) در زبانهای شیئی گرا را با مفهوم سوکتهای نوع استریم اشتباه نکنید. `InputStream` و `OutputStream` دو کلاس از کلاسهای جاوا (یا C++) هستند.

```

DataInputStream din = new DataInputStream( in );
String response = din.readLine();
server.close();
}
catch ( IOException e ) { }

```

در این مثال پس از ایجاد یک سوکت ، تلاش می‌شود تا با ماشینی با نام حوزه foo.bar.com و شماره پورت ۱۲۳۴ ارتباط TCP برقرار شود. در صورت موفقیت آمیز بودن این عمل ، استریمهای ورودی و خروجی ایجاد شده برای سوکت ، بدست می‌آید (در استریمهای in و out) تا بتوان روی آنها عملیات ورودی / خروجی انجام داد. نهایتاً پس از ارسال و دریافت ، از طریق این استریمها سوکت بسته می‌شود.

حال به کلاس سوکتی که در سمت سرویس‌دهنده باید ایجاد شود و برخی متوذهای آن ، دقت کنید :

```

ServerSocket( int port)
ServerSocket(int port, int count)
synchronized void close()
Socket accept()

```

متوذهای اول و دوم به گونه‌ای که از ظاهر آنها پیداست ، متوذهای سازنده هستند و پارامتر port ، آدرس (شماره) پورتهای است که باید در سمت سرویس‌دهنده به سوکت مقید (bind) شود. در متود دوم پارامتر count زمان انتظار برای گوش دادن به پورت جهت برقراری ارتباط است. در **ServerSocket** ، بطور درونی و خودکار عمل گوش دادن به پورت (listen) ، انجام می‌شود. متود **accept** دقیقاً طبق مفهومی که در ابتدای فصل عنوان شد ، یکی از ارتباطات معلق را برای پردازش به برنامه هدایت می‌کند. مقدار برگشتی این متود ، مشخصه یک شیء سوکت است که می‌توان استریمهای ورودی/خروجی آنرا بدست آورده و روی آنها ارسال یا دریافت داشت. در مثال زیر که متناظر با برنامه مثال قبلی است (یعنی در این دو مثال یکی سرویس‌دهنده و دیگری مشتری است) ، پس از ایجاد یک سوکت و مقید کردن شماره پورت ۱۲۳۴ ، یکی از ارتباطات معلق (در صورت وجود) پذیرفته شده و پس از عملیات ارسال و دریافت ، آن ارتباط بسته خواهد شد.

```

// Meanwhile, on foo.bar.com...
try {
    ServerSocket listener = new ServerSocket( 1234 );

    while ( !finished ) {
        Socket aClient = listener.accept(); // wait for connection
    }
}

```

```

InputStream in = aClient.getInputStream();
OutputStream out = aClient.getOutputStream();

// Read a byte
Byte importantByte = in.read();

// Read a string
DataInputStream din = new DataInputStream( in );
String request = din.readLine();

// Write a byte
out.write(43);

// Say "Goodbye"
PrintStream pout = new PrintStream( out );
pout.println("Goodbye!");

aClient.close();
}

listener.close();
}
catch (IOException e) { }

```

از بین متودهای متنوعی که در کلاس Socket تعریف شده، دو متود زیر در برخی از کاربردها بسیار مفیدند:

- **getport()**: این متود که متعلق به کلاس Socket است، شماره پورت انتخاب شده برای سوکت را بر می گرداند.

- **getHostName()**: این متود نام ماشین (نام نمادین) متناظر با یک سوکت را بر می گرداند. این دو تابع زمانی مفید است که در سمت سرویس دهنده، برنامه بخواهد با پذیرفتن یک ارتباط و دریافت شیء Socket متناظر با آن، هویت طرف مقابل ارتباط را تشخیص بدهد.

کلاسهای Socket و ServerSocket در جاوا بسیار ساده هستند و براحتی می توان آنها را مورد استفاده قرار داد. (در مورد سوکتهای دیتاگرام فعلاً مطلبی را مطرح نمی کنیم. در صورت تمایل به مراجع فصل مراجعه نمایید.)

۱۳) مراجع این فصل

مجموعهٔ مراجع زیر می‌توانند برای دست آوردن جزئیات دقیق و تحقیق جامع در مورد مفاهیم معرفی شده در این فصل مفید واقع شوند.

1. **Beej's Guide to Network Programming Using Internet Sockets, 1998.**
<http://www.ects.csuchico.edu/~beej/guide/net>
2. **Unix Network Programming, volumes 1-2**, W. Richard Stevens. Prentice Hall.
3. **Internetworking with TCP/IP**, Comer D.E., Prentice-Hall, 1996.
4. **Exploring Java, Patrick Niemeyer & Joshua Peck**; 1-56592-184-271-9, 2nd Edition July 1997 (est.)
5. **Java 1.2 Unleashed, Jamie Jaworski**, Macmillan Computer Publishing, 1998.
6. **Java By Example, Clayton Walnum**, Copyright© 1996 by Que® Corporation