



عنوان

پروژه طراحی سیستم های ایونت بیس و ارتباط از راه دور

رشته تحصیلی:

مهندسی برق

نیمسال دوم ۱۴۰۰ – ۱۴۰۱

فهرست مطالب

چکیده	۶
فصل اول - مفاهیم ابتدایی پروژه عملی	۷
۱-۱ تاریخچه نود جی اس	۷
۱-۲ سوکت و کتابخانه آن در نود چیست؟	۹
۱-۳ ایوند لوپ چیست؟	۱۶
۱-۴ آسینک / ایویت چیست؟	۱۸
فصل دوم - مفهوم کلی و چگونگی عملکرد رویداد گرایانه	۲۲
۲-۱ ساخت کلاس مربوط به مدیر رویداد	۲۲
۲-۲ نحوه ذخیره رویداد ها	۲۴
۲-۳ نحوه فراخوانی رویداد ها	۲۵
۲-۴ جمع بندی	۲۷
فصل سوم - کلاینت برای ارتباطات از راه دور	۲۸
۳-۱ پیش نیاز ها	۲۸
۳-۲ نحوه ارتباط اولیه کلاینت با سرور	۲۹
۳-۳ نحوه ارتباطات ثانویه	۳۱
۳-۴ سیستم ضربان قلب	۳۴
فصل چهارم - مرکز ارتباط سرویس ها و مسیر یابی	۳۵
۴-۱ پیشنهاد ها	۳۵
۴-۲ مدیریت ورود اولیه کلاینت	۳۷

۴-۳ دریافت دیتا از کلاینت و تسک بندی درخواست ها ۳۹

۴-۴ عمل انجام تسک ها ۴۰

فصل پنجم - خلاصه و نتیجه گیری ۴۶

۵-۱ نتیجه گیری ۴۶

فهرست شکل ها

- شکل ۱ - نحوه ارتباط سیستم ها در شبکه تی سی پی ۱۰
- شکل ۲ - مقایسه TCP و UDP ۱۲
- شکل ۲.۱ - مدل برنامه نویسی سوکت ۱۴
- شکل ۳ - شکل برنامه نویسی سوکت در نود ۱۶
- شکل ۴ - مدل نشان دهنده فاز های Eventloop ۱۷
- شکل ۵ - مدلی دیگر از ساختار EventLoop ۱۸
- شکل ۶ - نمونه کد عملی برای ترتیب اجرای کد ۱۹
- شکل ۷ - مدل پیاده شده async / await ۲۱
- شکل ۸ - ساختار اولیه کلاس EventsManager ۲۳
- شکل ۹ - ساختار اولیه توابع ثابت برای ذخیره رویداد ۲۴
- شکل ۱۰ - ساختار اولیه توابع ثابت برای فراخوانی رویداد ۲۳
- شکل ۱۱ - تست کلاس ساخته شده ۲۶
- شکل ۱۲ - پیشنهاد های کلاینت ۲۷
- شکل ۱۳ و ۱۴ - ایجاد اتصال اولیه به سرور توسط کلاینت ۳۰
- شکل ۱۵ - انجام درخواست های ورودی به کلاینت ۳۳
- شکل ۱۶ - سیستم ضربان قلب ۳۴
- شکل ۱۷ - پیشنهاد های سرور ۳۶
- شکل ۱۸ - مدیریت ورود اولیه کلاینت به سرور ۳۷
- شکل ۱۹ - مدیریت ورود ثانویه و رمز نگاری ۳۸

- شکل ۲۰ - اضافه کردن وظیفه به لیست ۳۹
- شکل ۲۱ - ترتیب بندی وظایف ۴۰
- شکل ۲۲ - آماده سازی انجام وظیفه ۴۱
- شکل ۲۳ - تکمیل وظیفه و مدیریت نتایج ۴۲
- شکل ۲۴ - آجکت مربوط به دیتا ۴۳
- شکل ۲۵ - نحوه مسیریابی دیتا ۴۴
- شکل ۲۶ - آفلاین بودن یوزر هدف ۴۵

چکیده

در دنیای امروزی ارتباط درست و سریع بین عناصر مختلف سیستم بحثی بسیار مهم محسوب می شود چرا که در صورت تحقق این امر کیفیت و عملکرد سیستم ها بهتر و تقسیم کار راحت تر صورت می گیرد در این پروژه ما قصد داریم تا با ساختن سیستمی جامع و ساده این مسئله را به نحو بسیار ساده حل کنیم ما در این پروژه با استفاده از مفاهیم ابتدایی شبکه و امنیت سعی می کنیم با ایجاد یک کتابخانه جامع و ساده به اپلیکیشن هایی که از این سیستم استفاده می کنند این اجازه را بدهیم که بدون در نظر گرفتن مکان جغرافیایی و بقیه سیستم های شبکه بتوانند به صورت مستقل در یک شبکه جامع و کامل با هم همکاری داشته باشند تا بتوانند کاستی هایی که در یکدیگر (مانند تحریم – محدودیت پردازش و ...) وجود دارد را جبران کنند و بدون توجه به هم کار خود را مستقل انجام داده و همدیگر را تکمیل کنند

تاریخچه نود جی اس

نود جی اس در ابتدا در سال ۲۰۰۹ توسط رایان دال (Ryan Dahl) نوشته شد. ۱۳ سال قبل از آن نیز اولین محیط توسعه جاوااسکریپت در سمت سرور با نام LiveWire Pro Web توسط نتاسکیپ معرفی شده بود. نسخه اولیه نود جی اس تنها از سیستم عامل لینوکس و مک او اس پشتیبانی می کرد. توسعه و نگهداری نود جی اس توسط رایان دال صورت می گرفت و سپس نیز توسط شرکت جوینت حمایت شد.

مشاهده نوار پیشرفت بارگذاری فایل در سایت فلیکر توسط رایان دال، الهام بخش ایده ساخت نود جی اس شد. در آن موقع حین بارگذاری فایل در سایت فلیکر، مرورگر وب نمی توانست تشخیص دهد که چه میزان از فایل بارگذاری شده و بنابراین برای نمایش روند پیشرفت بارگذاری، مرورگر مجبور بود به وب سرور درخواست دهد. رایان دال مشتاق راه ساده تری برای این کار بود.

رایان دال در سال ۲۰۰۹ به انتقاد از ضعف محبوب ترین سرور وب جهان یعنی آپاچی در زمینه رسیدگی به تعداد زیادی اتصال (تا ۱۰'۰۰۰ اتصال یا بیش تر) پرداخت و همچنین انتقاداتی را به متداول ترین روش کدنویسی یعنی برنامه نویسی ترتیبی (Sequential Programming) وارد کرد. در برنامه نویسی ترتیبی در مواجهه با کانکشن های همزمان یا کل فرایند برنامه موقتاً متوقف می شود یا بالا جبار مقداری زیادی از حافظه اصلی به پشته فراخوانی اختصاص می یابد.

رایان دال پروژه نود جی اس را در ۸ نوامبر سال ۲۰۰۹، در افتتاحیه همایش JSCond در اروپا به همگان معرفی کرد. نود جی اس تشکیل شده بود از موتور جاوااسکریپت وی ۸ (V8) گوگل به همراه یک حلقه رخداد (Event loop) و نیز یک رابط برنامه نویسی کاربردی سطح پایین برای ورودی/خروجی. ارائه پروژه توسط رایان دال در همایش JSConf با تشویق ایستاده حاضرین روبرو شد.

در ژانویه ۲۰۱۰، سامانه مدیریت بسته ان پی ام (npm) برای نود جی اس معرفی شد. ان پی ام فرایند انتشار و به اشتراک گذاری کد منبع کتابخانه های نود جی اس را آسان می کند و طراحی شده تا کار نصب، بروزرسانی و حذف کتابخانه های نرم افزاری را سهولت ببخشد.

در ژوئن ۲۰۱۱، مایکروسافت و جوینت با همکاری یکدیگر کار پیاده‌سازی نسخه بومی نود جی‌اس برای ویندوز را شروع کردند. اولین نسخه نود جی‌اس که از سیستم‌عامل ویندوز پشتیبانی می‌کرد در ژوئیه ۲۰۱۱ منتشر شد. در ژانویه ۲۰۱۲، رایان دال از مدیریت پروژه کناره‌گیری کرد و آن را به ایزاک اسلوتر (Isaac Schlueter) که همکار او و نیز به وجود آورنده ان‌پی‌ام نیز بود، واگذار کرد. در ژانویه ۲۰۱۴ اسلوتر نیز اعلام کرد که تیموتی جی فنتین (Timothy J. Fontaine) پروژه را رهبری خواهد کرد.

در دسامبر ۲۰۱۴ فدور اینداتنی (Fedor Indutny) انشعابی از نود جی‌اس را با نام آی‌او جی‌اس (io.js) شروع کرد. به خاطر اختلاف داخلی اعضای پروژه بر سر نظارت شرکت جوینت روی پروژه، آی‌او جی‌اس به عنوان یک پروژه^۸ جایگزین برای نود جی‌اس با سیاست «حاکمیت [متن] باز» به همراه یک کمیته فنی جداگانه به وجود آمد. برخلاف نود جی‌اس پدیدآورندگان آی‌او جی‌اس تصمیم گرفتند تا نسخه موتور وی‌ا۸ استفاده شده در پروژه، همواره بروز و مطابق آخرین نسخه آن باشد.

در فوریه ۲۰۱۵، قصد ایجاد بنیادی بی‌طرف با نام بنیاد نود جی‌اس اعلام شد. سپس در ژوئن همان سال، کمیته‌های هر دو پروژه رأی موافق به همکاری با یکدیگر تحت بنیاد نود جی‌اس دادند.

در سپتامبر ۲۰۱۵، نسخه ۰/۱۲ نود جی‌اس با نسخه ۳/۳ آی‌او جی‌اس ادغام شد و تحت نام نود (Node) با ورژن ۴/۰ منتشر شد. این ادغام ویژگی‌های استاندارد ES۶ موجود در موتور وی‌ا۸ و همچنین چرخه انتشار با پشتیبانی بلندمدت (Long-term support release cycle) را به نود جی‌اس اضافه کرد. از سال ۲۰۱۶ به بعد، وب‌سایت آی‌او جی‌اس در پیامی به بازدیدکنندگان پیشنهاد می‌کند تا در نتیجه ادغام دو پروژه و همچنین به خاطر عدم انتشار نسخه جدیدی از آی‌او جی‌اس، دوباره به استفاده از نود جی‌اس بازگردند.

سوکت چیست؟

قبل از پرداختن به سوکت و تعریف آن بهتر است چند موضوع مرتبط با این مفهوم را بشناسیم. آشنایی با این موضوعات، به شما درک بهتری از سوکت را ارائه می دهند.

آی پی آدرس

هر کامپیوتر یک آدرس آی پی دارد. این آدرس شمار های بی همتا است که از ۴ عدد ۸ بیتی تشکیل شده که با نقطه از یکدیگر جدا میشوند.

هنگامی که به یک شبکه وصل هستید و از یک IP برای ارتباط استفاده می کنید، شماره کامپیوتر شما را به شبکه معرفی می کند. وقتی سایتی را در مرورگر جستجو می کنید، درخواستی همراه IP آدرس شما به آن سایت فرستاده می شود. سرور آن سایت نیز اطلاعات درخواستی را به واسطه IP آدرس برای کامپیوتر شما ارسال می کند. در واقع IP آدرس در مسیریابی به سرور سایت موردنظر کمک می کند تا اطلاعات به محل درست خود برسند. زمانی که بخواهید چند صفحه از مرورگر خود را به طور همزمان باز کنید و چندین جستجو داشته باشید، دیگر IP آدرس به تنهایی پاسخگو نخواهد بود؛ در این مواقع بحث پورت مطرح می شود.

پورت

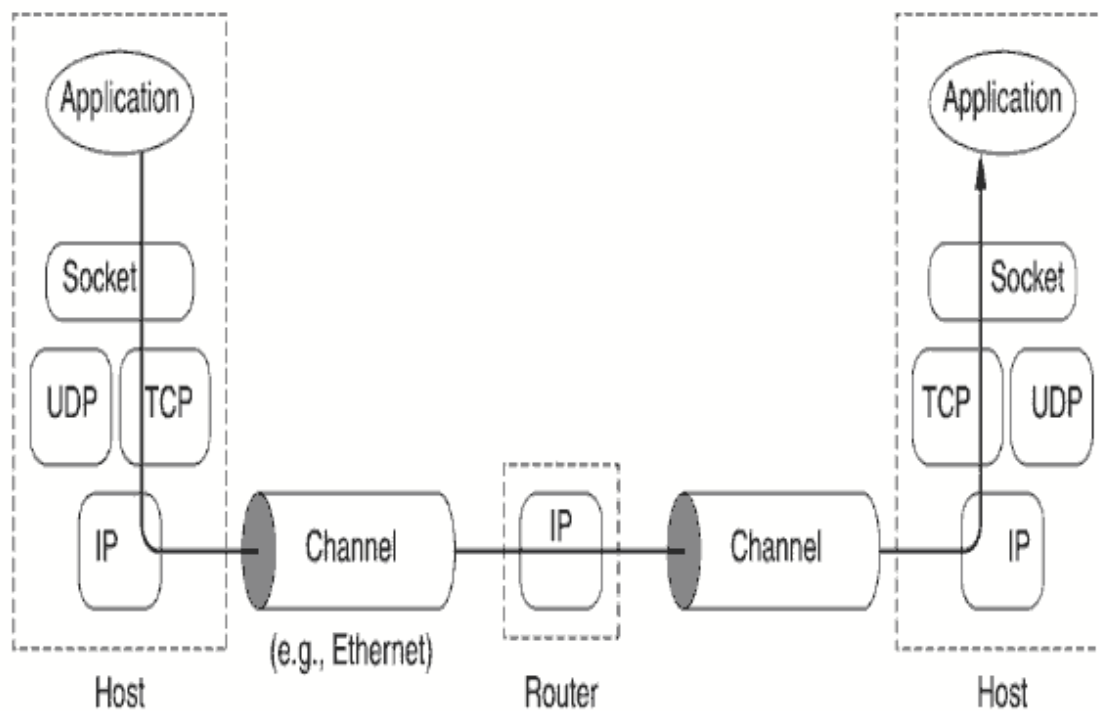
اینجا منظور از پورت، پورت منطقی است مانند سوکت که گفتیم دو تعریف منطقی و فیزیکی دارد. پورت فیزیکی نیز عملکردی مانند سوکت فیزیکی دارد. پورت منطقی عددی است برای شناسایی برنامه یا سرویسی خاص که قصد دسترسی به شبکه را دارد.

پورت و IP آدرس را می شود مانند شماره تلفن در نظر گرفت که IP آدرس کد شهر یا کشور است و پورت، باقی شماره تلفن در نظر گرفته می شود. کد شهر یا IP آدرس جهت شناسایی محدوده و منطقه تماس به کار می رود و قسمت باقیمانده یا پورت، شماره اختصاصی و بی همتایی است که تماس با آن برقرار میشود.

سوکت

به شکل ساده، سوکت ترکیبی از پورت و IP آدرس است. به تعبیر تخصصی تر، سوکت نقطه انتهایی یک ارتباط دو طرفه بین دو برنامه در حال اجرا در شبکه است. سوکت به یک عدد پورت متصل می شود تا لایه TCP شبکه بتواند برنامه موردنظر برای ارسال اطلاعات را تشخیص دهد. در مثال شماره تلفن، سوکت مانند گوشی تلفن است. به این شکل که شماره موردنظر و کد ناحیه را در گوشی وارد کرده و تماس را برقرار می کنید. زمانی که تماس پاسخ داده می شود، در واقع یک کانال ارتباطی بین شما و فردی که با او تماس گرفته اید، ایجاد می شود؛ به تعبیر ساده تر، کار سوکت ایجاد این کانال است. از طریق کانال ارتباطی ایجاد شده توسط سوکت، داده هایی در طول شبکه ارسال و دریافت میشوند. زبانی که دو برنامه به وسیله آن از این کانال با هم مکاتبه میکنند نیز پروتکل نام دارد.

TCP/IP network

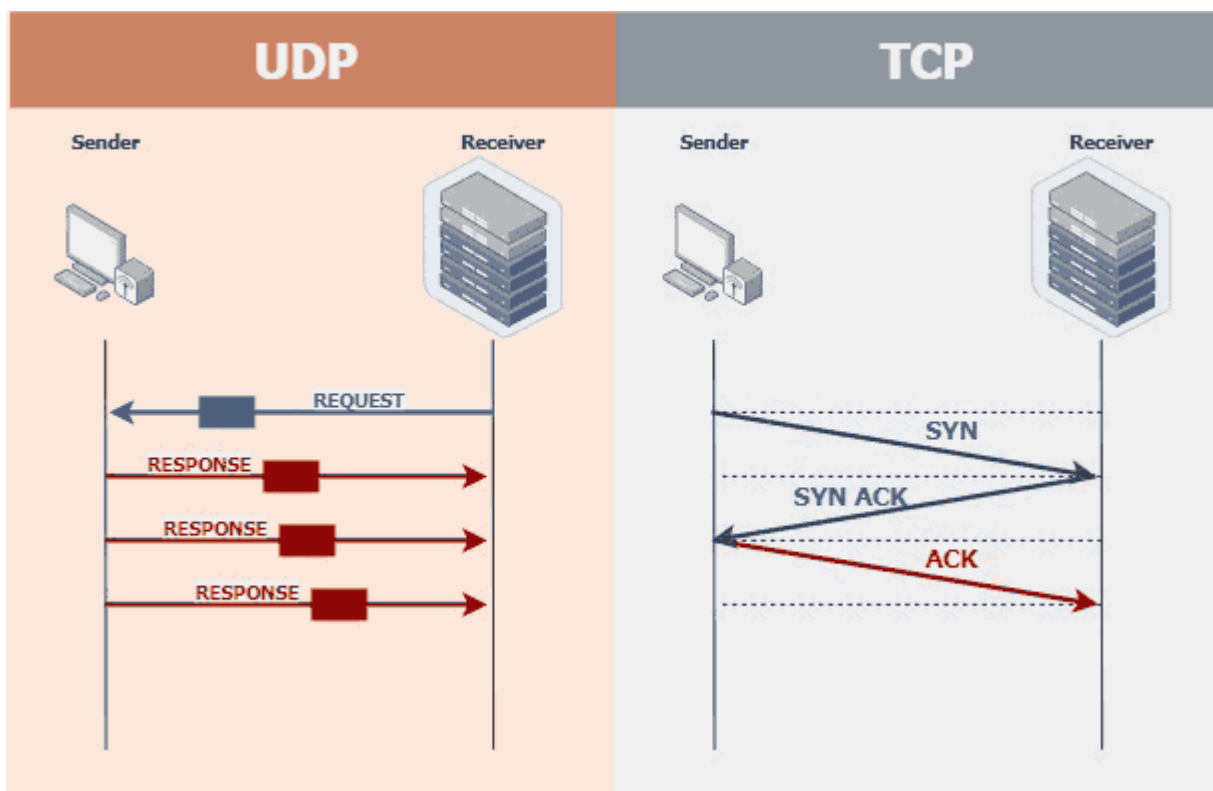


شکل ۱

این واژه مخفف Protocol Control Transmission و به معنی پروتکل کنترل انتقال می باشد. ارسال فایل‌های بزرگ در اینترنت به صورت یکپارچه صورت نمی گیرد. این فایل به قسمت‌های کوچکتر قابل کنترلی تقسیم می شود که هر یک به تنهایی به مقصد می رسند. به این تکه‌های کوچک packet گفته می شود و ما آنها را بسته می نامیم. این بسته‌ها تکه‌های خرد شده یک فایل بزرگ هستند. این بسته‌ها هر یک به تنهایی به مقصد موردنظر می رسند و هدف از این کار نیز کاهش ازدحام و ترافیک شبکه است. TCP از رسیدن تمام بسته‌ها به مقصد درست مطمئن می شود. سپس تصدیقی مبنی بر دریافت کامل بسته‌ها توسط مقصد ارسال می کند. اگر این تصدیق دریافت بسته‌ای به فرستنده نرسد، او متوجه مفقودی بسته شده و آن را دوباره ارسال می کند.

پروتکل‌های اینترنت

گفتیم زبان مشترکی که برنامه‌های متصل به شبکه، از طریق سوکت با آن ارتباط برقرار می کنند، پروتکل نام دارد. این پروتکل‌ها نیز متفاوت هستند. در واقع پروتکل‌ها چارچوب، قوانین و استانداردهایی هستند که جهت ایجاد ارتباط بین دو کامپیوتر در شبکه به کار میروند. IP آدرس که در قسمت سوکت آن را تعریف کردیم، در لایه IP شبکه اجرا می شود. پورت‌ها نیز در لایه transport یا حمل و نقل، به عنوان بخشی از سرگروه TCP یا UDP پیاده سازی میشوند.



شکل ۲

پروتکل IP / TCP که از آن در بخش سوکت سخن گفتیم نیز از دو نوع پورت TCP- port و UDP- port استفاده می کند. TCP برای کاربردهای بر پایه ارتباط طراحی شده و در ساختار آن، بررسی خطا و ارسال دوباره بسته های از دست رفته نیز وجود دارد. UDP برای کاربردهای بدون اتصال طراحی شده، بررسی خطا ندارد و بسته های از دست رفته را دوباره ارسال نمی کند. هر برنامه ای بسته به نوع اتصال یکی از پروتکل های لایه حمل و نقل UDP یا TCP را به کار می برد. گفتیم سوکت های شبکه از طریق پروتکل IP / TCP سازماندهی میشوند. بدین ترتیب، انواع سوکت ها نیز طبق دو نوع پورت این پروتکل تقسیم بندی می شوند.

انواع سوکت های شبکه

سوکت ها انواعی دارند که دو مورد از آنها طبق پروتکل های مورد استفاده دسته بندی شده اند. در ادامه تعریفی کوتاه از هر یک را ارائه خواهیم داد:

- سوکت های منطقی استریم که به Oriented Connection شهرت دارند.
- سوکت های دیتاگرام (Datagram).
- سوکت های خام
- سوکت های بسته متوالی

سوکت های استریم

این سوکت ها بر پایه پروتکل TCP بوده و با آن کار می کنند. به این شکل که قبل از تبادل اطلاعات باید یک اتصال مطمئن و قوی ایجاد شود. چنین اتصالی تضمین می کند که داده ها با نظم خاصی مبادله می شوند و داده ها به مقصد می رسند. از این نوع سوکت برای پروتکل هایی چون FTP, HTTP, SMTP استفاده می شود. پس از برقراری این ارتباط ایمن، می توان داده ها را از روی این سوکت ها خواند یا بر روی آنها نوشت. این نوع سوکت STREAM_SOCKET نام دارد. موارد استفاده سوکت های استریم در این سوکت که با پروتکل TCP کار می کند، داده ها به ترتیب، با نظارت بر بروز خطای احتمالی و به طور صددرصد تبادل می شوند. در این نوع سوکت ها، ثبت و ضبط داده هیچ محدودیتی ندارد. طبق موارد ذکر شده، پروتکل هایی که نیازمند این حد از حساسیت هستند، از این سوکت بهره می برند:

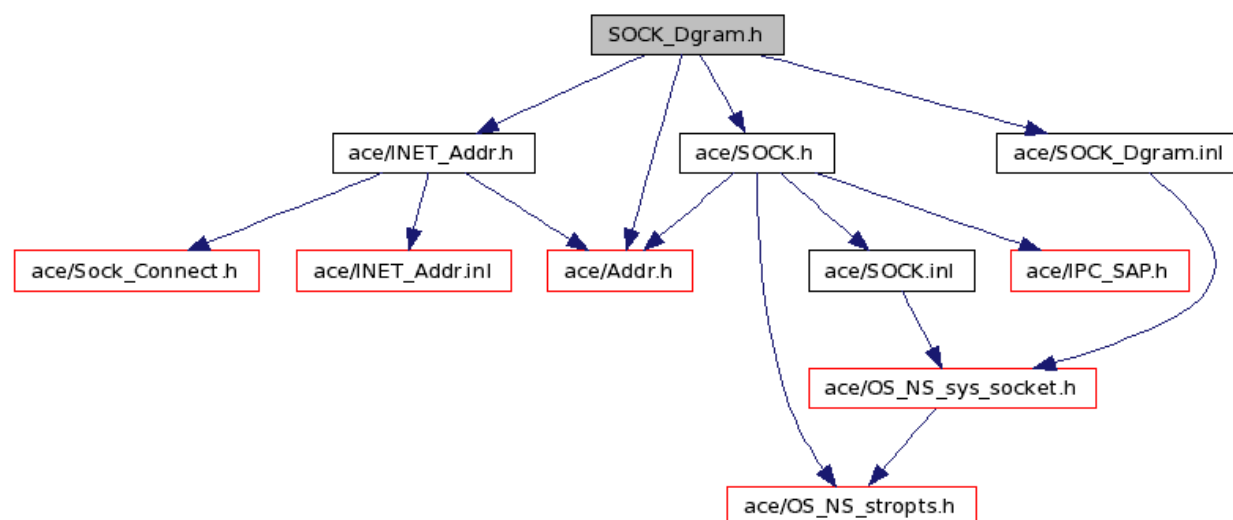
پروتکل انتقال فایل در اینترنت یا FTP

پروتکل انتقال صفحات ابرمتنی در اینترنت یا HTTP

پروتکل انتقال پست الکترونیکی یا SMTP

سوکت های دیتاگرام

این سوکت ها بر پایه پروتکل UDP یا Protocol Datagram User کار میکنند. در این نوع سوکت هیچ اتصالی از پیش ایجاد نمی شود. به دلیل اینکه اتصالی از قبل نیست، تضمینی هم برای انتقال صحیح داده ها، صحت آنها و رسیدن آنها به مقصد وجود ندارد.



شکل ۲.۱

در این پروتکل که به Less Connection نیز شهرت دارد، تنها مورد مهم سرعت انتقال و تبادل اطلاعات است و برای مبادله صوت و تصویر از آن استفاده می شود. این سوکت داده های تکراری را نیز دریافت کرده و DGRAM_SOCKET نام دارد سوکت های خام این سوکت ها با پروتکل ICMP یا Protocol Message یا Control Internet The کار می کنند. به طور معمول سوکت های خام، بر پایه دیتاگرام هستند. مشخصات دقیق این سوکت ها به رابط ارائه شده توسط پروتکل بستگی دارد. این سوکت ها برای کاربر عمومی در نظر گرفته نشده اند. سوکت های خام بیشتر برای افراد علاقه مند به توسعه پروتکل های ارتباطی جدید یا دسترسی به امکانات درونی و محرمانه پروتکل موجود، طراحی شده اند. این سوکت ها را فقط پرازنده های superuser می توانند استفاده کنند و RAW_SOCKET نام میگیرند.

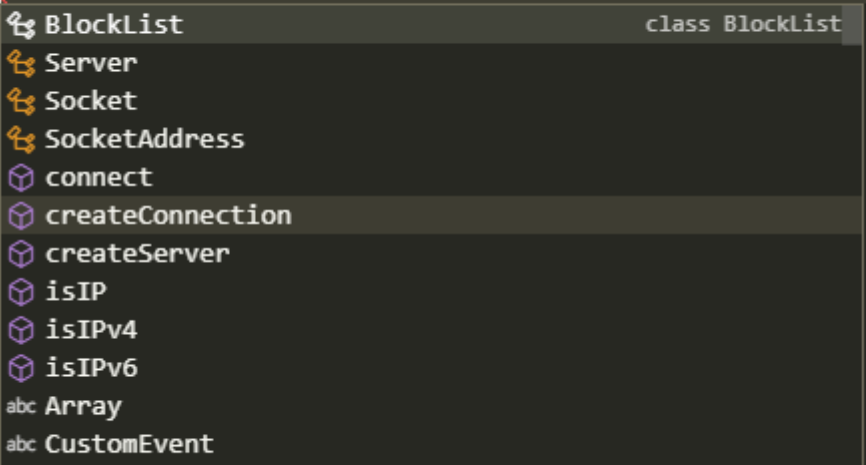
سوکت های بسته متوالی

این سوکت ها شبیه سوکت های استریم هستند با این تفاوت که ثبت و ضبط داده در این سوکت ها مرزبندی های مشخص دارد. این رابط فقط به عنوان بخشی از سیستم شبکه سوکت منطقی عمل می کند و این در بیشتر برنامه های کاربردی شبکه بسیار مهم است. سوکت های بسته متوالی به کاربر این امکان را می دهند تا سربرگ های پروتکل SPP یا Packet Sequence یا پروتکل IDP یا Datagram Internet را روی یک بسته یا گروهی از بسته ها دستکاری کند. این سوکت همچنین با نوشتن یک سربرگ نمونه برای هر اطلاعاتی که ارسال می شود و یا تعیین یک سربرگ مشخص برای تمام داده های خروجی، اجازه دستکاری را به کاربر می دهد. سوکت بسته متوالی همچنین کاربر را قادر می سازد تا سربرگ داده های ورودی را دریافت کند.

چگونه میتوان از سوکت در نود استفاده کرد؟

حال که میدانیم سوکت و انواع آن چیست و چه طور کار میکند برای استفاده از آن باید از کتاب خانه های آماده که زبان برنامه نویسی برای استفاده از شبکه در اختیار ما قرار میدهد استفاده کنیم. در مورد ما برای استفاده از این ویژگی باید از کتاب خانه Net استفاده کنیم برای این کار کافی است آن را به برنامه Import کنیم و در یک متغیر قرار دهیم البته ما در این پروژه از مدل ES6 که فایل های mjs هستن استفاده نمیکنیم و از مدل کلاسیک که استفاده از require هست از این کتابخانه استفاده میکنیم. برای وارد کردن آن به شکل تصویر زیر عمل میکنیم:

```
var socket = require("net");
socket.
```



شکل ۳

در این تصویر نحوه استفاده اولیه و متد های آن قابل مشاهده است

ایونت لوپ چیست؟

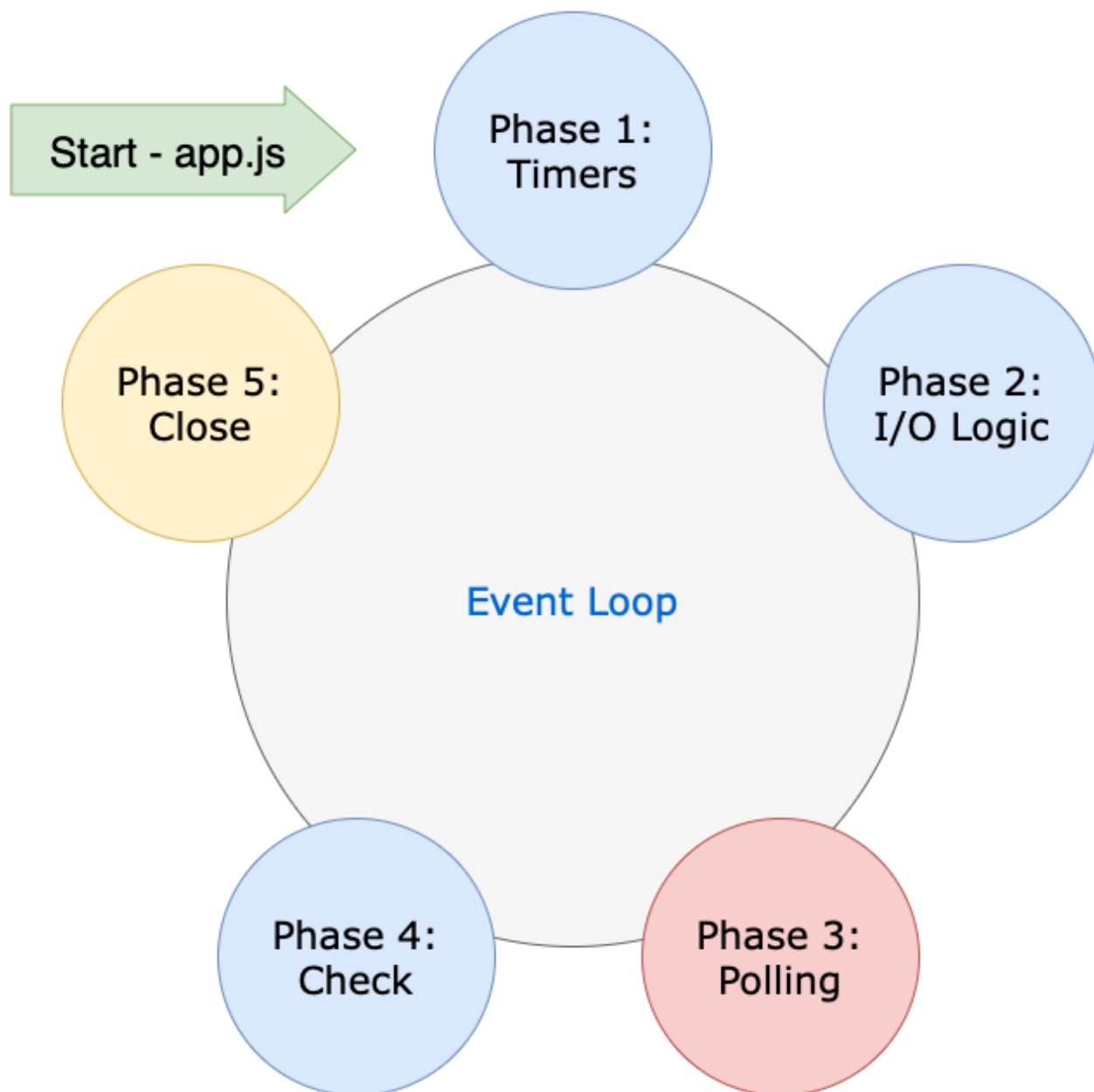
ایونت لوپ همان چیزی است که asynchronous programming را در جاوااسکریپت ممکن میکند.

مراحل اجرای یک نرم افزار node را در نظر بگیریم... به این صورت است:

(فرض کنیم یک app بسیار ساده با تنها یک thread (و یک event loop) داریم. به طور معمول بیش از یک thread -و نتیجتاً بیش از یک event loop- خواهیم داشت)

1. هنگام launch شدن یک thread ساخته میشود
2. سپس event loop درون آن ترِد initialized میشود
3. تمامی code برنامه اجرا می شود (هنوز اجرای event loop آغاز نشده)
4. سپس event loop اجرا می شود (و تا ابد ادامه دارد)

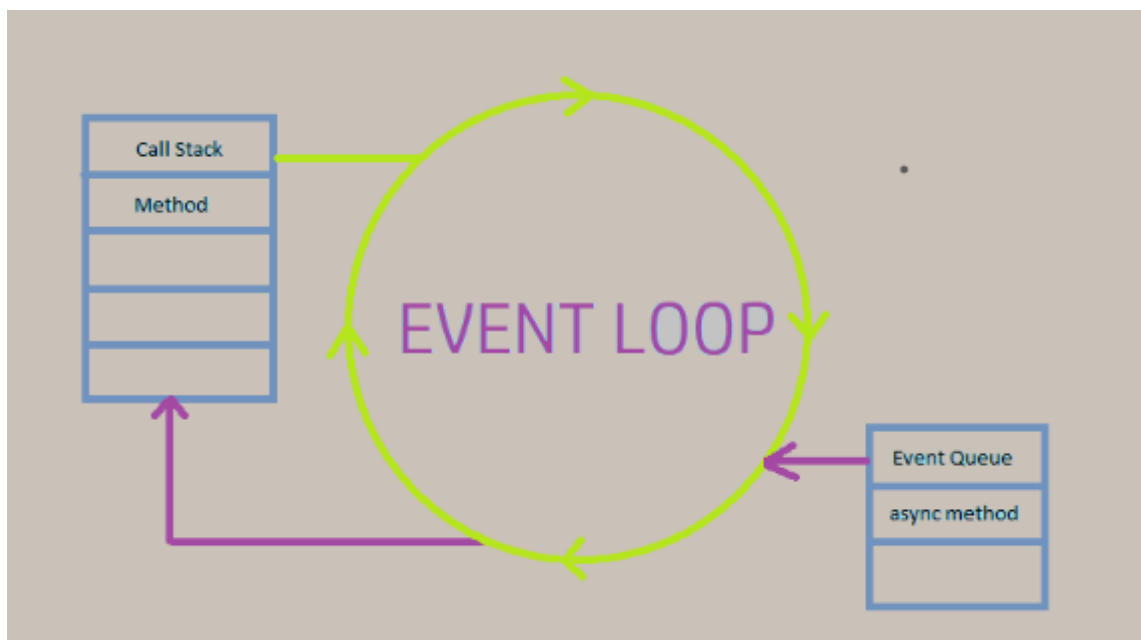
به هر iteration از event loop یک tick می‌گوییم که از 5 فاز (در واقع بیش از 5 فاز است) تشکیل شده:



شکل ۴

گفتیم ابتدا قسمت‌های synchronous برنامه اجرا می‌شود و سپس (زمانی که stack از همه‌ی این قسمت‌های sync خالی شد) event loop اجرا می‌شود.

کارِ event loop (این 5 فاز) به طور ساده این است که هرگاه stack خالی بود اولین تسک از task queue را به stack بدهد.



شکل ۵

برای درست کار کردن با event loop باید بدانیم که موارد زیر (عناصر async مان) کجای آن اند:

```
setTimeout ( callback, expTime )  
setInterval ( callback, expTime )  
setImmediate ( callback )  
process.nextTick( callback )  
promise // some promise instance
```

- تابعِ callback دو تابعِ setTimeout و setInterval بعد از اینکه زمانشان expire شد در اولین فرصت (همینطور که حلقه میچرخد) در فاز Timers اجرا میشود.

- در مورد `setImmediate` تابع `cb` اش در فاز `Check` اجرا میشود (در واقع قبل از رفتن به `tick` بعدی).
- در `event loop` هر `nextTick` و `promise` یک میکروتسک محسوب میشود که اجرای آنها در بین فاز های مختلف هندل میشود. برای مثال در ابتدای ورود به یک `tick` ایونت لوپ بررسی میکند که آیا میکروتسکی وجود دارد و اگر وجود داشته باشد ابتدا آن اجرا میشود. به همین دلیل در نمونه کد زیر ابتدا این دو لاگ میشوند:

```

1  const TIME_OUT = 10;
2  console.log('START');
3
4  setImmediate(() => {
5      console.log('setImmediate 1');
6      process.nextTick(() => { console.log('next tick 2') });
7  });
8
9  setTimeout(() => {
10     Promise.resolve().then(() => console.log("promise 2"));
11     setTimeout(() => console.log('setTimeout 3') , TIME_OUT);
12     setImmediate(() => console.log('setImmediate 2'));
13     process.nextTick(() => console.log('next tick 3'));
14
15     console.log('setTimeout 1');
16 }, TIME_OUT);
17
18 process.nextTick(() => console.log('next tick 1'));
19 Promise.resolve().then(() => console.log("promise 1"));
20
21 console.log('the code ENDS, now EVENT_LOOP starts...');
```

شکل ۶

خروجی به این صورت خواهد بود :

START

the code ENDS, now EVENT_LOOP starts...

promise 1

next tick 1

setImmediate 1

next tick 2

setTimeout 1

next tick 3

promise 2

setImmediate 2

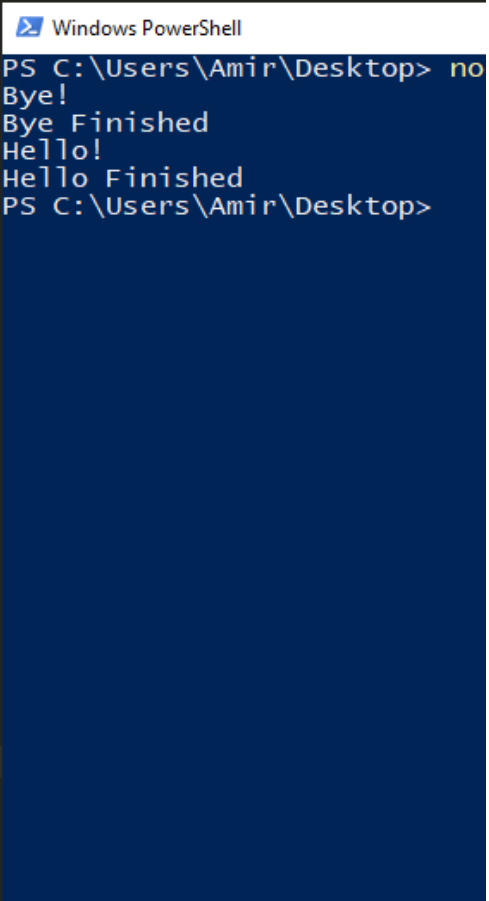
setTimeout 3

یک جمله که خیلی شنیده میشود این است که "don't block the event loop". این مستقیماً به پرفورمنس برمیگردد. مثلاً برای اینکه event loop برای انجام یک عملیات زمان بر، هندل کردن request ها - که در Polling phase انجام میشود - را معطل نکند (میدانیم ایونت لوپ سینگل ترد است) میتوانیم آن عملیات را توسط setImmediate که در Check phase و بعد از Polling اجرا شود، اجرا کنیم. اینگونه در هر tick اول request ها هندل شده و بعد آنکاره زمانبر انجام میشود..

روی هم رفته باید بدانیم چطور تسک ها را queue کنیم که پرفورمنس بهتری بگیریم!

نمونه دیگر از ایونت لوپ و عدم تداخل در دستورات:

```
1
2  const waitAndTalk = (ms, message) => {
3      return new Promise(res => {
4          setTimeout(() => {
5              console.log(message);
6              res();
7          }, ms);
8      })
9  }
10
11  const bruh = async (ms, message) => {
12      await waitAndTalk(ms, message)
13  }
14
15
16  ( async () => {
17
18      await bruh(5000, "Hello!");
19      console.log(`Hello Finished`);
20
21  })();
22
23
24
25
26  ( async () => {
27
28      await bruh(2000, "Bye!");
29      console.log(`Bye Finished`);
30
31  })();
```



شکل ۷

ساخت کلاس مدیریت رویداد

برای اینکه به این موضوع بپردازیم ابتدا باید مفهوم کلاس در نود را متوجه شویم :

در برنامه نویسی شی گرا ، کلاس یک قالب کدنویسی توسعه پذیر برای ایجاد اشیا است. کلاس مقادیر اولیه را برای وضعیت (متغیرهای عضو) و پیاده سازی رفتار (توابع یا متدهای عضو) فراهم می کند. در یک کلاس ، مجموعه ای از خصوصیت ها (Property) و متدهایی تعریف می شوند که برای همه اشیا از آن نوع مشترک هستند. برای مثال، یک کلاس می تواند یک خودرو باشد که دارای خصوصیت های «رنگ» و «داشتن چهار چرخ» باشد.

در برنامه ما کلاس ما شامل متود ها و لیست پراپرتی ها از مدل ثابت هستند این به این معنا است که شما برای استفاده از آنها لازم نیست یک کپی از کلاس بسازید و برای هر قسمت از برنامه یک نسخه جدا داشته باشید و تمام قسمت های برنامه از یک دیتای مشترک استفاده میکنند.

برای ساخت این کلاس به صورت زیر عمل میکنیم:

```
85  class EventManager {  
86  
87  }  
88  
89  
90  module.exports = EventManager;  
91
```

شکل ۸

در این قسمت ما به گذاشتن کلمه کلاس و اسم کلاسمون اون رو تعریف میکنیم و ادامه ساختار کلاس در بین کورلی بریس ها تعریف می شود

در انتها برای این که بتوانیم در فایل های دیگر به این کلاس دسترسی داشته باشیم آن را به لیست ماژول های نود اضافه میکنیم چرا که استفاده اصلی از این کلاس در قسمت های دیگر برنامه خواهد بود (به دلایل خوانا بودن بیشتر کد و قابلیت دیباگ راحل تر)

حال که کلاس ما آماده است باید ساختار درون آن را ایجاد کنیم برای این سیستم ما نیاز داریم به ۲ پراپرتی یا خصوصیت که برای اهداف زیر هستند:

هدف اول) ذخیره ساختار رویداد ها در داخل یک شیء که به فرمت جیسون یا نشانه گذاری شیء جاوا اسکریپت که در ادامه برنامه فراخوانی آن ها برای ما آسان تر باشد و سرعت و دقت لازم در این سیستم برای ما وجود داشته باشد.

هدف دوم) ذخیره سازی رویداد های تکرار شونده که مانند بالا در فرمت جیسون ذخیره میشوند ولی برخلاف مدل بالا بدون نیاز به فراخوانی تکرار میشوند این فرایند پس از اجرا به عنوان خروجی به ما تایع پایان تکرار برمیگرداند که به این معناست که بعد از فراخوانی این تایع رویداد تکرار شونده ما دیگر اجرا نمی شود این ساختار زمانی به کار می آید که شما میخواهید فرایندی را بدون نیاز به مدیریت دستی و پشت هم اجرا کنید این سیستم از تایمر های تکرار شونده استفاده میکنید که در بخش تایمر های ایونت لوپ در هر دور اجرا میشود که در این باره در فصل قبل مفصل توضیح داده شده است.

برای پیاده سازی این دو مورد به صورت زیر عمل میکنیم:

```
15
16  class EventManager {
17
18      static Events = {};
19      static RepetitiveEvents = {};
20
21
```

شکل ۸

نحوه ذخیره رویداد ها:

برای اینکه بتوانیم رویداد ها را ذخیره کنیم باید از ساختار ذخیره داده که در بخش قبلی ساخته شد استفاده کنیم این ساختار در رم سیستم ذخیره میشود و در زمان فراخوانی فرقی با یک تابع عادی ندارد. برای اینکه ذخیره این ساختار با استفاده از توابع زیر به هدف خود میرسیم.

```
14     static listen = (eventName, func) => {  
15         EventManager.Events[eventName] = func;  
16     }  
17  
18  
19     static repeat = (time, func) => {  
20  
21         var timer = setInterval(() => {  
22             func();  
23         }, time);  
24         return () => {  
25             clearInterval(timer);  
26         }  
27     }  
28
```

شکل ۹

در این حالت همان طور که در بالا گفته شد ۲ تابع رویداد و رویداد های تکرار شونده در اینجا تعریف شده است.

در اینجا به عنوان ورودی تابع یک اسم برای رویداد که بتوان آن را به اسم فراخوانی کرد و ورودی بعدی تابع خام اون رویداد است البته در اینجا ما به ورودی های تابع کاری نداریم و در آینده در هنگام اجرای تابع ورودی های آن را به صورت آرایه ای از ورودی ها به آن خواهیم داد با این کار ما میتوانیم تعداد ورودی های هر تابع را مدیریت کنیم و نگرانی در این مورد نداشته باشیم.

نحوه فراخوانی رویداد ها

برای اینکه بتوانیم از رویداد های تعریف شده استفاده کنیم باید توابعی مربوط به این رویداد ها تعریف شود. از آنجا که سیستم ما دارای ۲ نوع عملیات فرا خوانی هست ما نیز نیاز به ۲ نوع تابع داریم:

(۱) تابع سنکرون

(۲) تابع آسنکرون

تفاوت این ۲ تابع برای این است که در مدل اول ما به نتیجه اتفاقی که می افتد کاری نداریم (البته مفهوم سنکرون یعنی اینکه برنامه ما تا زمان تمام نشدن عملیات تابع منتظر می ماند) و فقط از این تابع ها جهت تغییر در برخی قسمت های برنامه استفاده میکنیم. ورودی این توابع مشخص هستند و خروجی ندارند (تابع مدل پوچ هستند)

در توابع آسنکرون همانطور که اینونت لوپ کار میکند ما از مفهومی به اسم پرامیس یا قول استفاده میکنیم که از مدل میکرو عملیات ها محسوب میشوند و پس از تمام شدن لوپ فعلی و قبل از شروع لوپ بعدی اجرا میشوند خوبی استفاده از این سیستم این است که ما میتوانیم از ایونت های متعدد به صورت موازی در برنامه استفاده کنیم و نتیجه را بعدا به سیستم تحویل دهیم این قابلیت باعث میشود که تعداد توابع قابل پردازش ما در بازه زمانی مشخص افزایش چشمگیری داشته باشد.

تابع آخر که مربوط به پاک سازی رویداد هاست برای وقتی استفاده میشود که شما یک رویداد شناور میسازید (رویداد شناور زمانی هست که رویداد شما در حین اجرا بودن برنامه ساخته میشود نه در ابتدای ران شدن) بعد از استفاده دیگر به آن نیازی نیست و ما نیاز داریم آن را پاک کنیم چرا که اگر این کار را انجام ندهیم پس از مدتی مموری سیستم ما با کمبود ظرفیت رو به رو خواهد شد و از جهتی دیگر چون اسم هر رویداد یکبار مصرف است احتمال تداخل در رویداد ها و عملکرد آن ها ممکن خواهد بود.

برای فراخوانی رویداد ها به صورت زیر توابع رویدادی را میسازیم:

```
31 static call = (eventName, ...args) => {
32     if(EventsManager.Events[eventName] != undefined)
33     {
34         var func = EventsManager.Events[eventName];
35         func(...args);
36     }
37     else
38     {
39         throw new Error(`Unknown event name: ${eventName}`)
40     }
41 }
42
43 static callAsync = (eventName, ...args) => {
44     return new Promise((res, rej) => {
45         if(EventsManager.Events[eventName] != undefined)
46         {
47             var func = EventsManager.Events[eventName];
48             res(func(...args));
49         }
50         else
51         {
52             rej(`Unknown event name: ${eventName}`)
53         }
54     })
55 }
56
57 static delete = (eventName) => {
58     if(EventsManager.Events[eventName] != undefined)
59     {
60         delete EventsManager.Events[eventName];
61     }
62 }
63
64 }
65
```

شکل ۱۰

جمع بندی

در نهایت یک مدل استفاده ساده برای دریافت و تحلیلی یک سیستم ساده کامند (مثلا در ربات تلگرام) به صورت زیر خواهد بود:

```
78
79
80 Windows PowerShell
81 PS C:\Users\Amir\Desktop> node '.\EventManager.js'
82 [ '/command', 'arg1', 'arg2', 'arg3' ]
83 ChatID: 231515 - Command: /command - Arg1: arg1 - Arg2: arg2 - Arg3: arg3
84 PS C:\Users\Amir\Desktop>
85
86
87
88
89
90 EventsManager.listen("Bot:OnCommandWithParams", (chatid, command, arg1, arg2, arg3) => {
91   console.log(`ChatID: ${chatid} - Command: ${command} - Arg1: ${arg1} - Arg2: ${arg2} - Arg3: ${arg3}`)
92 })
93
94 var link = "/command arg1 arg2 arg3"
95 console.log(link.split(/\s/))
96 setTimeout(() => {
97   EventsManager.call('Bot:OnCommandWithParams', 231515, ...link.split(/\s/))
98 }, 2000);
```

شکل ۱۱

پیش نیاز ها

برای اینکه برنامه ما بتواند به درستی کار کنید نیاز داریم تا از بعضی از کتاب خانه های پیشفرض پلتفرم نود استفاده کنیم برای این منظور آن ها را در ابتدای فایل برنامه قرار می دهیم، کتاب خانه هایی که نیاز داریم به شکل زیر خواهند بود:

۱. Net Socket

۲. Crypto

۳. EventsManager

۴. Node-RSA

هر یک از کتاب خانه های بالا به کار منحصر به فرد خود را دارا میباشند و با کمک آن ها ساخت برنامه برای ما ممکن خواهد بود.

- ۱- نت سوکت قلب برنامه ماست، ما به وسیله نت سوکت توسط پروتکل تی سی پی به سرور اصلی وصل شده و با آن ارتباط رفت و برگشتی داریم این ارتباط تا زمانی که نیاز هست حفظ شده و قطع نمی شود همینطور تا زمانی که این ارتباط برقرار است از ارتباط مجدد یوزر با اسم مشابه جلوگیری خواهد شد
- ۲- کریپتو به ما اجازه میدهد تا متن های تصادفی واقعی بسازیم در صورتی که از کریپتو استفاده نکنیم این متن ها تصادفی واقعی نخواهند بود و افرادی که قصد به هم زدن امنیت سیستم را داشته باشند با ابزار مناسب قادر به حدس زدن سشن آیدی های ما خواهند بود.
- ۳- ایونت منیجر همان کتابخانه داخلی قبلی است که در قسمت قبل ساختیم با استفاده از این کتابخانه به قلب برنامه دسترسی داریم و میتوانیم ایونت هایی که در قسمت های دیگر برنامه نوشته شده اند را با دیتایی که از سمت سرور دریافت میشود فراخوانی کرده و در صورت نیاز نتیجه را برگردانیم.
- ۴- نود آر اس ای یکی از کتابخانه های مفید نود است که با استفاده از آن میتوانیم ارتباط بین سرور و کلاینت را توسط کلید عمومی سرور رمزگذاری کنیم و مسیر ارتباط بین یوزر ها را امن کنیم این کار باعث میشود تا ما بدون ترس دیتا های مهم و حساس را جا به جا کنیم.

```

2  const net = require('net');
3  const crypto = require("crypto");
4  const EventsManager = require('./EventManager');
5  const rand = (length=16) => {return crypto.randomBytes(length).toString("hex")}
6  var RSACryptor = require("./RSA.js");
7

```

شکل ۱۲

نحوه ارتباط اولیه کلاینت با سرور:

برای ایجاد این ارتباط ابتدا به آیپی ، پورت و پسوردی که میخوایم وصل بشیم را به صورت ورودی به کلاینتمون بدیم همینطور آیدی اون کلاینت که باید حتما منحصر به فرد باشد را نیز می دهیم چرا که در غیر این صورت سرور کلاینت ما را قبول نخواهد کرد.

بعد از انجام اتصال انکودینگ ارتباط را به یو تی اف ۸ تنظیم میکنیم همینطور تعداد ماکزیمم شنونده ها که در کلاینت زیاد مهم نیست ولی اگر ارتباط سنگین شود مهم میشود و همینطور ماکزیمم تایم اوتی که برای کلاینت در نظر داریم ۵ ثانیه است گرچه میتواند بیشتر نیز شود ولی مقدار آپتیمال آن ۵ ثانیه است. حال بعد از انجام اتصال اولین بسته اطلاعاتی که برای سرور میفرستیم آیدی و کلید عمومی خودمان است که بدون کد گذاری فرستاده میشود البته میتوان از کلید از پیش اشتراک گذاشته شده این ارتباط را نیز امن کرد و یا آیدی را بعدا فرستاد ولی نسخه اولیه برنامه به این صورت عمل نمی کند، بعد از این که این دیتا رو فرستادیم سرور نیز منتظر یک پیام از سمت ماست و پیام های دیگر اهمیتی داده نمیشوند در صورتی که مشکلی وجود نداشته باشد (آیدی در سرور نباشد) سرور نیز برای ما کلید عمومی خود را میفرستد (این نیز بدون کد گذاری است گرچه میتواند کد گذاری داشته باشد چون کلید عمومی کلاینت در دسترس است ولی صرف نظر شده است) بعد از دریافت دیتا از سمت سرور با توجه به اینکه نود ساختار اسنکرون دارد همه دیتا هایی که در یک بازه به سمت کلاینت یا سرور ارسال میشوند در کنار هم وصل و با هم به برنامه داده میشوند برای همین ما نیاز داریم از یک جدا کننده که در اینجا سمی کلون هست استفاده کنیم تا پیام ها را متوجه شویم از آنجا که دیتا در فرمت جیسون هستند و انکود میشوند این کار برای ما مشکلی ایجاد نخواهد کرد، حال که این موضوع را میدانیم دیتا های ورودی را بدون پردازش رها کرده و فقط اولین پکت ورودی را مورد بررسی قرار میدهیم چرا که پکت جواب سرور برای ورود همین خواهد بود (سرور در صورتی که ما وصل نباشیم دیتا های مربوط به ما را دراپ میکند و به بقیه یوزر ها اجازه انجام فعالیت بر روی ما را نمیدهد ولی این مکانیزم ایجاد شده که اگر در همون تایم دیتای ورودی داشتیم بتوانیم آن را نیز پردازش کنیم) حال که جواب سرور را گرفته ایم رمزی که برای استفاده از سرور

باید داشته باشیم را رمز گذاری کرده (با استفاده از کلید عمومی سرور که در اختیار داریم) و برای سرور میفرستیم قبل از این اتفاق سرور که بر روی اتصال ما ، ما را پردازش کرده است حال برای یک بار منتظر پکت پسورد از سمت ما خواهد بود کلاینت ما نیز بعد از ارسال پکت پسورد به صورت تک باره منتظر جواب سرور خواهد بود که آیا سرور قبول کرده است یا خیر دی صورتی که سرور قبول کرده باشد کلاینت ما کلید عمومی سرور را در خود ذخیره و برای ارتباط های بعدی از آن استفاده میکنی حال کلاینت ما به صورت نامحدود به ورودی ها گوش میدهد در این مرحله کلاینت ما دیگر به مرحله فرستادن پسورد و کلید کاری ندارد و اگر موردی از این شکل برایش ارسال شود ، آن را به شکل یک درخواست عادی پردازش میکند و مشکلی پیش نخواهد آمد.

در تصاویر زیر مراحل گفته شده را به صورت کد مشاهده میکنید:

```
16
17 var client = net.createConnection({port:port, host:ip}, () => {
18
19     client.setEncoding("utf8")
20     client.setMaxListeners(2500);
21     client.setTimeout(5000)
22
23     console.log(`Connection to RPC server successful, setting up info ...`);
24
25
26     // Use secret for public key sending
27
28     var sendObject = {
29         id:id,
30         pubKey:RSAEncryptor.getSelfPublicKey()
31     }
32
33     // client.write(`${JSON.stringify(sendObject)};`);
34     client.write(`${JSON.stringify(sendObject)}`);
35
36
37 });
```

شکل ۱۳

```

39 client.once("data", (cResponse) => {
40
41     var ReponseData = cResponse.toString();
42     var _ResponseIndex = ReponseData.indexOf(";");
43     var ResponseObject = JSON.parse(ReponseData.substr(0, _ResponseIndex))
44     if(ResponseObject.status)
45     {
46
47         if(ResponseObject.serverKey != undefined)
48         {
49             var authObject = {
50                 secret:secret
51             }
52
53
54             client.write(`${RSAEncryptor.encryptData(ResponseObject.serverKey, JSON.stringify(authObject))}`);
55
56             client.once("data", (serverAuthObject) => {
57
58                 var ServerAuthReponseData = serverAuthObject.toString();
59                 var _ServerAuthReponseData = ServerAuthReponseData.indexOf(";");
60                 var AuthObjectResponse = JSON.parse(ServerAuthReponseData.substr(0, _ServerAuthReponseData))
61
62                 if(AuthObjectResponse.status)
63                 {
64                     console.log(`Connection accepted!`)
65                     ServerKey = ResponseObject.serverKey;
66
67                     var _chunk = "";
68                     var d_index = -1;
69
70                     client.on("data", async (data) => {
71

```

شکل ۱۴

نحوه ارتباطات ثانویه

برای اینکه کلاینت بتواند از سمت سرور دیتا دریافت کنید باید در داخل کانال تی سی پی ایجاد شده اقدام به گوش کردن کند و در صورت ورود پکت جدید دیتا را آنالیز و عمل کند بدیهی است که این کار باید به دقت و سرعت کافی انجام شود و نتیجه به سرعت برگردد البته سرعت بازگشت به برنامه نویس اصلی که از کتابخانه استفاده میکند مربوط است ولی سیستم ما باید بتواند بخش خودش را به سرعت هندل کند.

برای این کار ما بر روی کلاینتمون منتظر دیتای جدید میمونیم و همونطور که قبلا گفته شده بود در نود دیتا ها بافر میشوند و در صورت وجود درخواست های پی در پی این دیتا ها با هم به عنوان یک دیتا وارد میشوند برای همین میبایست توسط یک سمی کالون از هم جدا شده و سپس در کلاینت ما از هم جدا شوند و هر کدام به ترتیب رسیدگی شوند البته این فرایند دریافت اسنکرون است و سیستم را بلاک نخواهد کرد.

در ابتدای دریافت دیتا ابتدا از هم جدا می شوند و سپس به ترتیب بررسی و انجام می شوند، دیتا های دریافت شده از ۲ مدل هستند، دیتا های بازگشتی و بدون بازگشت.

دیتا های بدون بازگشت دیتا هایی هستند که در برنامه ما اجرا میشوند و نتیجه آن ها برای ما مهم نیست و فقط میخواهیم اتفاقی انجام شود، بدیهی است در صورتی که در برنامه ارور داشته باشیم توانایی تشخیص ارور را در حین اجرای برنامه نخواهیم داشت و فقط اگر لاگ داشته باشیم بعدا متوجه آن میشویم.

دیتا های بازگشتی دیتا هایی هستند که پس از اجرا نتیجه آن ها برای ما مهم هستند این مدل از دیتا ها به صورت آسنکرون اجرا میشوند و سرور نیز در زمان ارسال این مدل دیتا ها به ما برای جواب صبر میکند، البته در سرور زمان خاصی برای تایم اوت شدن این صبر تعریف نشده که این میتواند مشکل داشته باشد ولی در نسخه فعلی برنامه برای ما مهم نیست ولی در آینده درست خواهد شد. زمانی که جواب درخواست در خود برنامه به نتیجه رسید، برنامه با توجه به آیدی که از سمت سرور بر مبنای شناسه نتیجه درخواست دریافت کرده آن را برای سرور میفرستد و سرور نیز جواب را به فرستنده تحویل میدهد.


```

70 client.on("data", async (data) => {
71
72     // console.log(`DATA IN :::: ${data}`)
73
74     _chunk += data.toString();
75     d_index = _chunk.indexOf(';');
76
77     // console.log(`DATA CHUNK: ${_chunk}`)
78
79     while (d_index > -1)
80     {
81         try
82         {
83             var JSONData = _chunk.substring(0,d_index);
84             // console.log(JSONData)
85             // console.log(encryptor.decrypt(encryptor.keyStore, JSONData))
86             var inData = "";
87             inData = JSON.parse(RSAEncryptor.decryptData(RSAEncryptor.getSelfPrivateKey(),
88             // console.log(inData)
89
90             // var inData = JSON.parse(data)
91             if(inData.from != undefined)
92             {
93                 // we need to call some registered local events
94                 if(inData.rpc)
95                 {
96                     var response = await EventManager.callAsync(inData.name, ...inData.ar
97
98                     var res = {};
99                     res.body = response;
100                     res.rid = inData.rid;
101                     res.id = inData.from;
102
103                     client.write(`${RSAEncryptor.encryptData(ServerKey, JSON.stringify(res
104                 }
105             else
106             {
107                 /* console.log("=====")
108                 console.log(inData)
109                 console.log(inData.name)
110                 console.log(inData.args)
111                 console.log("=====") */
112                 if(inData.name != undefined && inData.args != undefined) EventManager
113             }
114         }
115     }

```

شکل ۱۵

سیستم ضربان قلب

سیستم ضربان قلب برای این ایجاد شده است که اگر کلاینت ما پشت نت باشد و مدت زیادی ساکت بماند احتمال بریک شدن ارتباط وجود دارد برای همین بعد از تمام شدن مراحل اولیه و تثبیت کانکشن، هر ۱۰ ثانیه به سرور دیتا بیخود به سرور فرستاده میشود، از این امر برای تشخیص خطا در ارتباط نیز میتوان استفاده کرد اما دی حال حاضر تنها استفاده از آن تامین استمرار ارتباط است.

```
if(heartbeat_service == null)
{
    heartbeat_service = setInterval(() => {
        console.log(`Sending heartbeat ...`)

        var requestBody = {
            beat:true
        }

        // console.log(JSON.stringify(requestBody))
        // console.log(encryptor.encrypt(encryptor.keyStore, JSON.stringify(requestBody)))

        client.write(`${RSAEncryptor.encryptData(ServerKey, JSON.stringify(requestBody))};`, (err) => {
            if(err != undefined) console.log(`ERROR CALL REMOTE: ${err}`)
            else console.log(`Heartbeat response OK.`)
        })
    }, 10000);
}
```

شکل ۱۶

فصل چهارم - مرکز ارتباط سرویس ها و مسیر یابی

در این فصل از پروژه ما به معرفی قسمت اصلی برنامه یعنی سرور آن میپردازیم. در سرور عملیات اصلی مسیر یابی و انتقال دیتا از یوزری به یوزر دیگر و همینطور امنیت شبکه و رمزنگاری های هر یوزر به صورت مجزا اتفاق می افتد، به دلیل اینکه یوزر های ما ممکن است آپی استاتیک نداشته باشند سرور تنها سورسی است که از آن میتوانند یکدیگر را با نام فراخوانی کنند و به هم دیتا بفرستند.

پیشنیاز ها

قسمت پیشنیاز های سرور نیز همانند کلاینت است ولی تعداد آنها کمتر است برای مثال ما در سرور به ایونت منیجر نیازی نداریم چرا که ما برای سرور برنامه نویسی نمیکنیم و فقط از آن استفاده میکنیم.

برخی از پیشنیاز های سرور به شکل زیر می باشند:

1. Net Socket

2. Crypto

3. RSA Encryption

علاوه بر پیشنیاز های کتابخانه ای ما از تعدادی متغیر مهم در سیستم نیز استفاده میکنیم این متغیر ها به برنامه اجازه میدهند تا یوزر ها و تسک هایی که باید انجام دهند را مدیریت کنند که به صورت زیر هستند:

آرایه clients : این آرایه ذخیره کننده یوزر های سرور ورودی سرور است ما به وسیله این آرایه میتوانیم تشخیص دهیم که چه کسی در سرور با چه نامی حضور دارد و سریع ترین راه دسترسی به آن ها چیست، اگر یوزری در این آرایه نباشد به معنای این است که سرور آن را نمیشناسد به علاوه، زمانی که از سیستم ضربان قلب استفاده میشود در داخل این آرایه دینای مربوط به زمان هر یوزر به روز شده و در صورت به روز نبودن زمان ضربان یوزر ها آن ها از لیست ما حذف خواهند شد.

آرایه task : این آرایه زمانی استفاده میشود که سرور ما باید کار جدیدی را انجام دهد، اگرچه سرور ما به صورت آسنکرون کار میکند ولی همچنان تک هسته است و قسمت هایی از آن باعث بلاک شدن ترد اصلی و عدم توانایی پاسخگویی ایونت لوپ به درخواست های جدید میشود، شاید با خود بگویید که برنامه باید مولتی

ترد باشد، بله این حرف شما درست است و در آینده نیز این اتفاق خواهد افتاد ولی فرایند مولتی تردینگ در نود کمی متفاوت است و برنامه ها با هم مموری به اشتراک نمیگذارند برای همین هر ترد از برنامه تنها پورت سرور را به اشتراک میگذارند و این فرایند مسیر یابی برای سرور با غیر ممکن میکند (البته راه هایی هست برای درست کردن این مشکل ولی پیچیده هستند و زمان بر) برای همین ما در حال حاضر مجبور هستیم هر درخواست را تسک بندی کنیم و با اضافه کردن این درخواست ها به لیست تسک ها و ساخت ایونت لوپ خودمون (تایمری که هر 10 میلی ثانیه فعال میشود و در صورت اتمام درخواست فعلی بعدی را اجرا میکنید) آن ها را مدیریت کنیم. استفاده از این متود باعث میشود در صورتی که تعداد زیادی درخواست همزمان بر روی سرور وارد شد، سرور به راحتی بتواند به همه پاسخ دهد.

شیء RESPONSE_CODE : در این آبجکت به صورت پیشفرض تعدادی کد اررور تعریف شده که در زمان بروز خطا کلاینت مورد نظر را آگاه کند. این آگاهی کمک میکند که یوزر ما از وضعیت درخواست خود مطلع شود و بتواند با توجه به نتیجه فرایند اجرای کد خود را ادامه دهد (مثلاً ارور ایجاد کند یا دوباره تلاش کند)

```
1
2  const net = require('net');
3  const crypto = require("crypto");
4  const rand = (length=16) => {return crypto.randomBytes(length).toString("hex")}
5
6  var RSAEncryptor = require('./RSA.js')
7
8
9  var clients = [];
10 var tasks = [];
11 var RESPONSE_CODE = {
12   "REQUEST_FAILED": -1,
13   "REQUEST_SUCCESS": 1,
14   "REQUEST_ERROR": 0
15 }
16
```

شکل ۱۷

مدیریت ورود اولیه کلاینت:

اولین کاری که برای راه اندازی سرور در برنامه انجام میشود سخت سرور با فانکشن `net.createServer` است که به ما در صورت ورود کانکشن جدید آبجکت یوزر را به صورت کال بک باز میگرداند، این آبجکت یوزر را میتوان برای مدیریت یوزر فعلی استفاده کرد.

همونطور که در قسمت کلاینت توضیح داده شد در اولین مرحله ما در سرور تنها برای 1 باز از یوزر ورودی میخواهیم این ورودی تا زمانی که پردازش نشده باشد ورودی جدید را قبول نخواهیم کرد برای همین این ورودی باید همان دیتای مورد نیاز شناسایی ما باشد که شامل کلید عمومی و اسم یوزر خواهد بود. در ابتدا دیتای فرستاده شده باید فرمت دلخواه ما را داشته باشد برای همین ما سعی در بررسی خوانا بودن آن میکنیم در صورتی که مشکلی باشد یوزر قطع خواهد شد.

```
const server = net.createServer((client) => {  
  client.setEncoding("utf8");  
  
  console.log(`[${getTime()}] new connection from ${client.remoteAddress}`)  
  
  // Creating client object  
  client.once("data", (data) => {  
    // console.log(`Data after connection?`)  
  
    // var connectData = data.toJSON();  
  
    /* ar PreConnectData = data.toString();  
    var ConnectSplitIndex = PreConnectData.indexOf(";");  
    var connectData = JSON.parse(PreConnectData.substr(0, ConnectSplitIndex)); */  
  
    var connectData = undefined;  
  
    try  
    {  
      connectData = JSON.parse(data)  
      console.log(connectData)  
    }  
    catch(e)  
    {  
      console.log(`Connection Error ${e}\n${client.address().address} is dropped due to error.`)  
      client.destroy();  
      return;  
    }  
  
    // console.log(`[${getTime()}] Conenct Data: ${data} - ${connectData} - ${JSON.stringify(connectData)} - ${connectData.id}`)
```

شکل ۱۸

در ادامه سرور بررسی میکند که یوزر فعلی که میخواهد از سرور استفاده کنید آیا در حال حاضر در سرور وجود دارد یا خیر چرا که در صورتی که وجود داشته باشد عمل مسیر یابی دیتا به اشتباه انجام خواهد شد چرا که این اسم ها باید منحصر به فرد باشند. دقت شود که دیتای فرستاده شده که شامل آیدی است با آیدی های موجود

در سرور مقایسه میشود و در صورت نبود مشکل سرور اجازه عبور میدهد، حال گاهی ممکن است یوزر در سرور باشد ولی ضربانی نفرستاده باشد برای همین قبل از پس زدن یوزر جدید این نیز بررسی خواهد شد و در این صورت یوزر قبلی حتی اگر هنوز در حال استفاده باشد از سیستم خارج خواهد شد و جای خود را به یوزر جدید خواهد داد. زمانی که کانکشن تایید شود سرور استاتوس **true** را برای یوزر ارسال کرده و کلید عمومی خود را به همراه این درخواست برای یوزر ما ارسال خواهد کرد.

حال که یوزر ما از لحاظ کانکشن تایید شده باید از لحاظ اجازه ورود به سرور نیز تایید شود برای این امر کلاینت باید رمز سرور را توسط کلید عمومی ارسال شده رمز گذاری کنید و برای سرور بفرستد، سرور در این زمان نیز تنها منتظر یک پسورد از یوزر ماست و در صورتی که این یوزر چیزی برای سرور ما ارسال کند که به این امر مربوط نباشد از سرور بیروز خواهد افتاد.

بعد از دریافت پسورد و تایید آن یوزر ما به عنوان یوزر معتبر در سیستم میتواند فعالیت خود را آغاز کند البته لازم به ذکر است که در حال حاضر باگی وجود دارد که اگر یوزر دیتای به غیر از فرمت جیسون برای ما بفرستد سرور کرش خواهد کرد که حل آن زیاد سخت نبوده و در نسخه های آپدیت شده این سیستم در گیت هاب درست خواهد شد.

```
52
53   var clientObject = {
54     id:connectData.id,
55     pubKey:connectData.pubKey,
56     socketObject:client,
57     beat:new Date().getTime() + 10000
58   }
59
60   var isIdInUse = clients.find(clientObject => clientObject.id == connectData.id);
61   // console.log(isIdInUse)
62   if(isIdInUse)
63   {
64
65     if(isIdInUse.beat > new Date().getTime())
66     {
67       client.write(`${JSON.stringify({status:false})}`);
68       console.log(`[${getTime()}] Client ID in-use!`)
69       // client.destroy("client id in use")
70     }
71     else
72     {
73       clients.splice(clients.indexOf(isIdInUse), 1);
74       console.log(`[${getTime()}] expired client, new connection accepted!`)
75     }
76   }
77   else
78   {
79     client.write(`${JSON.stringify({status:true, serverKey:RSAEncryptor.getSelfPublicKey()})}`);
80   }
81
82   client.once("data", (authObject) => {
83
84     console.log(`AuthObject: ${authObject}`)
85
86     var _obtainedSecretObject = JSON.parse(RSAEncryptor.decryptData(RSAEncryptor.getSelfPrivateKey(), authObject));
87     var _obtainedSecret = _obtainedSecretObject.secret;
88
89     console.log(`S: ${secret}`)
90     console.log(`C: ${_obtainedSecret}`)
91
92     if(_obtainedSecret == secret)
93     {
94       clients.push(clientObject);
95       client.write(`${JSON.stringify({status:true})}`);
96
97       console.log(`[${getTime()}] new client registered, id: ${connectData.id}`);
98     }
```

شکل ۱۹

دریافت دیتا از کلاینت و تسک بندی درخواست ها

زمانی که یوزر ما در سرور تایید شد، سرور از سمت آن یوزر به دنبال دیتای ارسالی برای مسیر یابی آن دیتا خواهد بود بنابراین در هر ورودی دیتا آن ها را به لیست تسک ها اضافه کرده و آنها توسط سیستم لوپ ساخته شده مدیریت میشوند، تسک لیست به صورت زیر عمل میکند:

```
client.on("data", async (clData) => {  
  tasks.push({  
    client:client,  
    data:clData,  
    clObject:clientObject  
  })  
})
```

شکل ۲۰

بعد از اینکه انجام کار ها به تسک لیست واگذار شد در تسک لیست تایمری وجود دارد که خودش خود را تمدید میکند این تایمر به صورت بی نهایت این کار را انجام میدهد و در صورتی که تسکی در سیستم وجود داشته باشد عملکردش تا زمان تمام شدن آن تسک مختل شده و بعد از انجام آن تسک آن را پاک میکند و ادامه میدهد برای این کار لازم است این سیستم هر سری اولین عنصر آرایه تسک ها را بگیرد (چون از 0 به بالا عنصر ها چیده میشوند اولین عنصر زود ترین آن ها خواهد بود) و سپس آن را به تابع اجرای تسک که عمل اصلی را انجام میدهد واگذار کنید این تابع آسینک و یا پرامیس نیست (در صورت پرامیسی فای شدن این تابع عملکرد بهتری خواهیم داشت) و باید سرور صبر کنید تا عملکرد آن تمام شود.

عمل بررسی تسک ها به صورت زیر خواهد بود:

```
var prPre = () => {  
  
    if(tasks.length > 0)  
    {  
        var task = tasks[0];  
  
        console.log(`Doing Task: ${task}`);  
        processQueue(task)  
  
        tasks.splice(0, 1);  
    }  
  
    setTimeout(prPre, 10);  
}  
  
setTimeout(prPre, 10);
```

شکل ۲۱

عمل انجام تسک ها

بعد از اینکه تسک بندی ها انجام شد ما نیاز به تابعی داشتیم که عمل تسک ها را انجام دهد این تابع اسنکرون و پرامیس نبود (گرچه گفته شد در صورتی که بشه بهتر است اما انجام آن نیازمند تغییر در کد و عملکرد برنامه خواهد شد) برای همین هر تسک برای اجرا شدن نیازمند صبر برای تسک بعدی خواهد بود.

زمانی که تسک به تابع اجرا داده میشود ۳ مدل دیتا از دل تسک ما استخراج میشود:

- سوکت یوزر
- دیتای آماده فرستاده شدن
- دیتای فرستنده

این سه عدد دیتای نام بده شده دیتا های اصلی برای پردازش خواهند بود ، بعد از دریافت دیتا ها توسط تابع، عمل جدا سازی دیتا توسط سمی کلون انجام میشود تا دیتا های بافر شده این کلاینت با هم پردازش شوند دقت شود که این عمل از پیش برای ما تعریف نشده و عملاً باعث میشود تعداد تسک هایی که انجام میشوند در یک مرحله بیشتر از یک عدد باشد ولی برای این که درخواستی از دست ما نرود میبایست این کار انجام شود. بعد از پایان جدا سازی هر قسمت برای خودش به ترتیب این درخواست ها انجام میشن

```
const processQueue = (queObject) => {  
  
  var client = queObject.client;  
  var clData = queObject.data;  
  var clientObject = queObject.clObject  
  
  var _chunk = "";  
  var d_index = -1;  
  
  console.log(`DATA IN :::: ${clData}`)  
  
  _chunk += clData.toString();  
  d_index = _chunk.indexOf(';');  
  
  // console.log(`CHUNK: ${_chunk}`)  
  
  while (d_index > -1)  
  {  
    try  
    {  
      var JSONData = _chunk.substring(0,d_index);  
      var eventData = ""  

```

شکل ۲۲

ما دو مدل دیتای ورودی برای سرور داریم، دیتایی که برای خود سرور است (دیتای ضربان قلب) و دیتایی که باید مسیریابی شود. دیتای ضربان قلب تنها برای تمدید زمان کانکشن استفاده میشود و تنها دیتایی که دارد دیتایی به اسم beat است اگر این دیتا وجود داشته باشد ما میدانیم که ضربان ارسال شده است ولی اگر نباشد

میدانیم که دیتا برای مسیر یابی ارسال شده است. پس از این ورودی های دیتای دریافتی را جدا سازی میکنیم این ورودی ها عبارتند از:

- ریموت آیدی: شناسه درخواست فرستنده پیام نیاز در صورتی که پیام فرستاده شده از نوع نیازمند جواب باشد
- از سمت: شناسه خود فرستنده
- به : شناسه کسی که دیتا قرار است به او ارسال شود.
- تابع راه دور: تابعی که بر روی کسی که دیتتا را دریافت میکند اجرا میشود
- ورودی های تابع راه دور
- آر پی سی: آیا این درخواست نیازمند جواب است؟

```
207     var beat = eventData.beat;
208     if(beat)
209     {
210         console.log(`[${getTime()}] Heartbeat from ${client.localAddress} (${clientObject.id})`);
211         var ClientObjectForBeat = clients.find(clobj => clobj.id == clientObject.id);
212         if(ClientObjectForBeat != undefined)
213         {
214             ClientObjectForBeat.beat = new Date().getTime() + 10000;
215             ClientObjectForBeat.socketObject.write(`${RSAEncryptor.encryptData(clientObject.pu`);
216             client.setNoDelay();
217         }
218     }
219     else
220     {
221         var rid = eventData.rid; // Request ID for response to client if RPC was true
222         var from = clientObject.id;
223         var to = eventData.id;
224         var eventName = eventData.name;
225         var args = eventData.args;
226         var rpc = eventData.rpc;
227
228         if(eventName != undefined)
229         {
230             if(rpc)
231             {
232                 console.log(`[${getTime()}] Requesting async task from ${from} to ${to} for th`);
233             }
234             else
235             {
236                 console.log(`[${getTime()}] Requesting from ${from} to ${to} for the event nam`);
237             }
238         }
239         else
240         {
241             console.log(`[${getTime()}] Collecting async response from ${from} to ${to}.`);
242         }
243     }
```

شکل ۲۳

بعد از مشخص شدن ورودی ها آبجکت مربوط به ارسال دیتا به گیرنده را ایجاد میکنیم که شامل قسمت های زیر است:

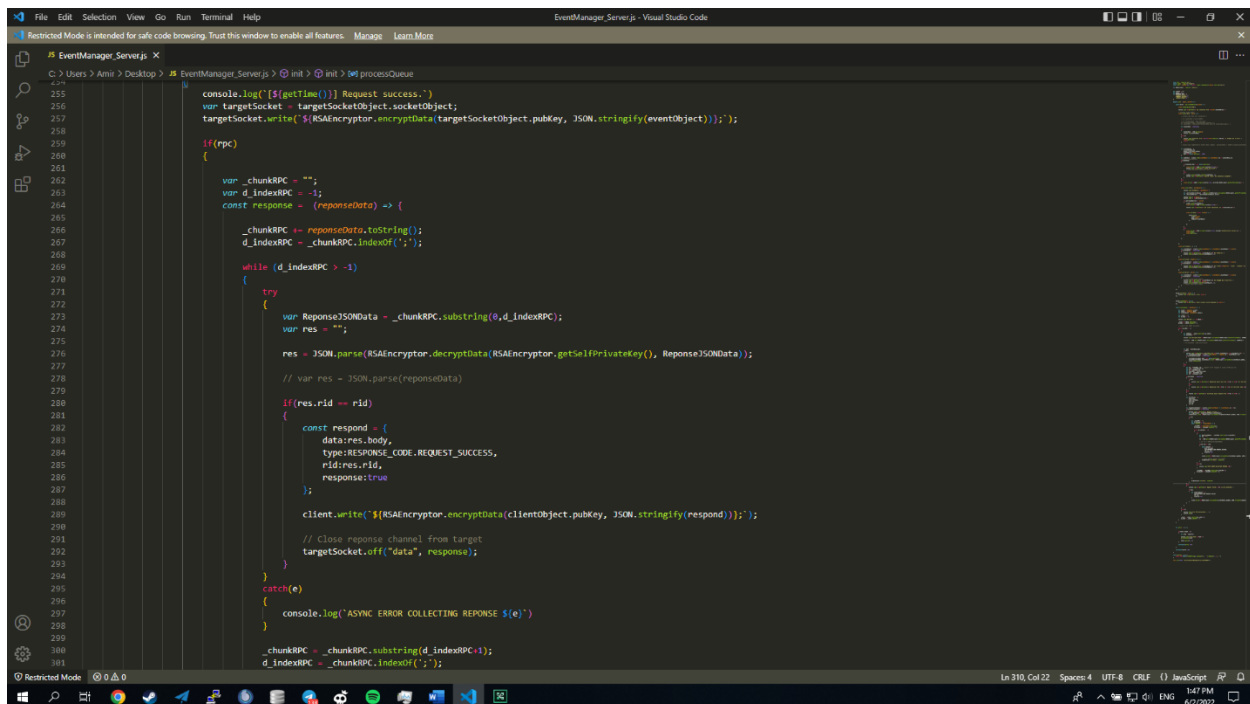
- فرستنده
- اسم تابع راه دور
- ورودی های تابع راه دور
- آر پی سی
- شناسه درخواست فرستنده

```
243  
244     var eventObject = {  
245         from:from,  
246         name:eventName,  
247         args:args,  
248         rpc:rpc,  
249         rid:rid  
250     }  
251
```

شکل ۲۴

سپس بررسی میشود که آیا اصلا گیرنده مورد نظر وجود دارد یا خیر در صورتی که وجود داشته باشد آبجکت مربوط به ارسال کد گذاری شده و سپس برای گیرنده ارسال میشود، اگر این دیتا نیازمند پاسخ باشد مدل پیشین دریافت دیتا برای گیرنده ایجاد میشود و سرور منتظر دریافت دیتا از گیرنده می شود شاید با خودتان بگویید این انتظار ممکن نیست ورودی جدید از سمت گیرنده برای فرستادن دیتا را مختل کند؟ (چرا که ما آسنکرون کار میکنیم و گیرنده میتواند در حین انجام در خواست فرستنده با بقیه قسمت ها در ارتباط باشد و خودش نیز فرستنده آن قسمت باشد) باید بگوییم که خیر این اتفاق نمی افتد فرضیه این است که سرور ما لیسنر (Listener) ها را شناسایی و دیتا را به همه آن ها میفرسند، تنها دیتایی واقعا پردازش میشود که اسم داشته باشد و دیتا های ورودی متفرقه اسم ندارند و فقط آیدی و بدنه دارند، این آیدی نیز از نوع آیدی درخواست است و برای سرور در مرحله اول قابل شناسایی نیستند پس به دلیل نبود گیرنده سرور آن ها را پردازش نمیکند برای همین این دیتا ها برای لیسنر پاسخ شناخته شده هستند و لیسنر عادی به آن ها واکنش

نشان نخواهد داد و فقط لیسر پاسخ آن ها شناسایی و در صورتی که به وی مربوط باشند پاسخ میدهد این پاسخ همزمان با لغو ادامه درباقتی ها خواهد بود، این لیسر ها نمیتوانند تک باره باشند چرا که در این صورت با هر ورودی از کار می افتند و دیگر قادر به جمع آوری پاسخ از سمت گیرنده نیستند.



```
255 console.log(`${getTime()} Request success.`);
256 var targetSocket = targetSocketObject.socketObject;
257 targetSocket.write(`${RSAEncryptor.encryptData(targetSocketObject.pubkey, JSON.stringify(eventObject))}`);
258
259 if(rpc)
260 {
261     var _chunkRPC = "";
262     var d_indexRPC = -1;
263     const response = (responseData) => {
264         _chunkRPC += responseData.toString();
265         d_indexRPC = _chunkRPC.indexOf(';');
266         while (d_indexRPC > -1)
267         {
268             try
269             {
270                 var ResponseJSONData = _chunkRPC.substr(0, d_indexRPC);
271                 var res = "";
272                 res = JSON.parse(RSAEncryptor.decryptData(RSAEncryptor.getSelfPrivateKey(), ResponseJSONData));
273                 // var res = JSON.parse(responseData)
274                 if(res.rid == rid)
275                 {
276                     const response = {
277                         data: res.body,
278                         type: RESPONSE_CODE_REQUEST_SUCCESS,
279                         rid: res.rid,
280                         response: true
281                     };
282                     client.write(`${RSAEncryptor.encryptData(clientObject.pubkey, JSON.stringify(response))}`);
283                     // Close response channel from target
284                     targetSocket.off("data", response);
285                 }
286             }
287             catch(e)
288             {
289                 console.log("ASYNC ERROR COLLECTING RESPONSE ${e}")
290             }
291             _chunkRPC = _chunkRPC.substr(d_indexRPC+1);
292             d_indexRPC = _chunkRPC.indexOf(';');
293         }
294     }
295 }
```

شکل ۲۵

در نهایت در صورتی که ما از فرد غیر متصل درخواست با جواب داشته باشیم این درخواست با خطا از سمت سرور ما را مطلع میسازد

```
else
{
    console.log(`${getTime()}` Request failed, ${to} is not connected.`)

    if(rpc)
    {
        var responseObject = {
            type:RESPONSE_CODE.REQUEST_FAILED,
            rid:rid,
            response:true
        }

        client.write(`${RSAEncryptor.encryptData(clientObject.pubKey, JSON.stringify(responseObject))}`);
    }
}
```

شکل ۲۶

فصل پنجم - خلاصه و نتیجه گیری

ما در این پروژه سعی کردیم که یکی از مشکلات برنامه نویسان برای برقرار ارتباط بین برنامه های نودی را برطرف کنیم با استفاده از این سیستم دیگر نیاز به ایجاد API های متعدد برای برقراری ارتباط بین سیستم ها نیست گرچه این سیستم ها نیاز به سرعت و پشتیبانی یوزر های زیادی را دارند ولی استفاده از این روش برای ارتباط های درون برنامه ای توصیه نمیشود.

ما با استفاده از این سیستم مفهوم مولتی تردینگ در برنامه را با تیکه کردن هر قسمت از برنامه به قسمتی مستقل و ارتباط مستقیم با توابع همدیگر ادا میکنیم چرا که در این صورت ما نیاز به استفاده از کتاب خانه های مولتر تردینگ ندارم و منابع خود را بیشتر مدیریت میکنیم هم چنان زمانی که قسمتی از برنامه از دسترس خارج شود بقیه قسمت ها همچنان داخل شبکه هستند و این سلامت شبکه ما را تامین میکند.

ما به خوبی سعی کردیم با استفاده از مفاهیم اسینک در نود اتفاقاتی که درون برنامه ما می افتد را مدیریت کنیم به این هدف که همه قسمت ها میتوانند بدون نگرانی درخواستی ارسال کنند و جواب آن را در سریع ترین زمان ممکن بگیرند.

از ما پنهان نیست که در درجه فعلی این کتابخانه مشکلات متعددی وجود دارد اگرچه این کتابخانه قابل استفاده در سطوح بزرگ است ولی همیشه امنیت و سرعت این سیستم مورد تهدید است و بهینه سازی بیشتری را میطلبد برای همین در حال حاضر فعلا در مدل آزمایشی در آن در سطح localhost برای طراحی بک اند سایت میتوان از آن استفاده کرد البته برای باز کردن پورت این سیستم میتوان از فایروال هایی استفاده کرد که فقط سرور های مربوطه را اجازه اتصال دهند.

در آینده برنامه است که با ایجاد تعامل بین سرور و یوزر های استفاده کننده، بتوان سرور را کانفیگ کرد و مسیر یابی ها را بهتر کنترل کرد، همینطور در زمان اتصال میتوان ار کلید از پیش به اشتراک گذاشته شده (PSK) برای ایجاد امنیت اولیه بین کلاینت ها و سرور استفاده کرد تا یوزر های بیرونی نتوانند آیدی ها را شناسایی کنند، همینطور سشن بندی نیز میتواند برای ما مفید باشد چرا که در این صورت با به روز رسانی سشن آیدی میتوان درخواست های اصلی از جعلی را شناسایی کرد.