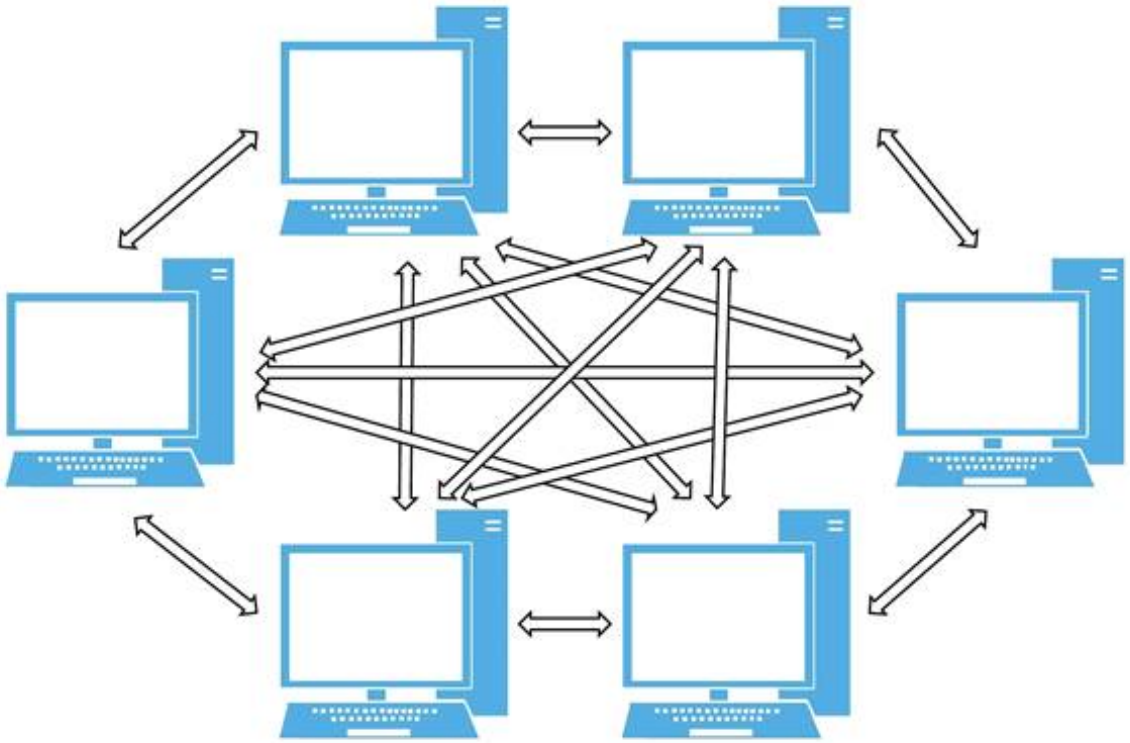
پروژه سوم درس شبکه‌های کامپیوتری

علیرضا سالمی
۸۱۰۱۹۶۴۸۰
رضا قنبری
۸۱۰۱۹۶۵۲۸

۳۱ تیر ۱۳۹۹

۱ مقدمه

در این پروژه قصد داریم یک شبکه نظیر به نظیر را به کمک زبان برنامه نویسی پایتون پیاده‌سازی نماییم. سپس با جمع آوری الاعات حاصل از اجرای شبکه برای مدت زمان مشخص ۵ دقیقه، نتایج را تحلیل کرده و گراف شبکه از دید هر گره را نیز رسم نماییم.



شکل ۱: شبکه نظیر به نظیر

در ادامه ابتدا این نوع از شبکه‌ها را بررسی کرده و سپس به شرح پیاده‌سازی و شبیه‌سازی انجام شده می‌پردازیم. **در ادامه ممکن است برای اشاره به این شبکه‌ها از کلماتی مانند نظیر به نظیر یا همتا به همتا اشاره شود که با هم معادل هستند و هر دو ترجمه کلمه peer to peer هستند.**

۲ شبکه‌های p2p

اگر جزء آن گروه از افرادی هستید که به طور مداوم از اینترنت استفاده می‌کنند، به احتمال زیاد بارها و بارها واژه p2p یا همان شبکه‌های نظیر به نظیر را در مقالات مختلف مشاهده کرده‌اید. قصد داریم که در ادامه به طور مختصر در باره این شبکه‌ها اطلاعاتی را بیان کنیم.

۱.۲ شبکه نظیر به نظیر به چه معناست؟

ظیر به نظیر یا p2p ه گونه خاصی از شبکه‌های کامپیوتری اشاره دارد که از یک معماری توزیع شده استفاده می‌کنند. به این معنا که همه کامپیوترها یا دستگاه‌های عضو این شبکه حجم کاری خود را در شبکه به اشتراک قرار می‌دهند. کامپیوترها یا دستگاه‌هایی که بخشی از یک شبکه نظیر به نظیر هستند peers نامیده می‌شوند. کامپیوترهای درون یک شبکه نظیر به نظیر هیچ‌گونه ارجحیتی نسبت به یکدیگر نداشته و همگی و با یکدیگر برابر هستند. کامپیوترهای درون یک شبکه نظیر به نظیر بدون آنکه به یک سیستم مدیریت متمرکز نیازی داشته باشند منابع را میان یکدیگری تقسیم می‌کنند. شبکه‌های نظیر به نظیر که به نام شبکه‌های همگرا نیز شهرت دارند همگی مجوزها و اختیاراتی یکسان با یگدیگر دارند. کامپیوترها می‌توانند در یک زمان در نقش سرور و کلاینت ظاهر می‌شوند. علاوه بر این، در یک شبکه همگرا منابع در دسترس میان کامپیوترها مختلف به اشتراک قرار می‌گیرد، بدون آنکه سرور مرکزی نقشی در این زمینه داشته باشد. پردازنده‌ها، فضای ذخیره‌سازی دیسک و پهنای باند از جمله منابعی هستند که در این شبکه‌ها به اشتراک قرار می‌گیرند. [۱]

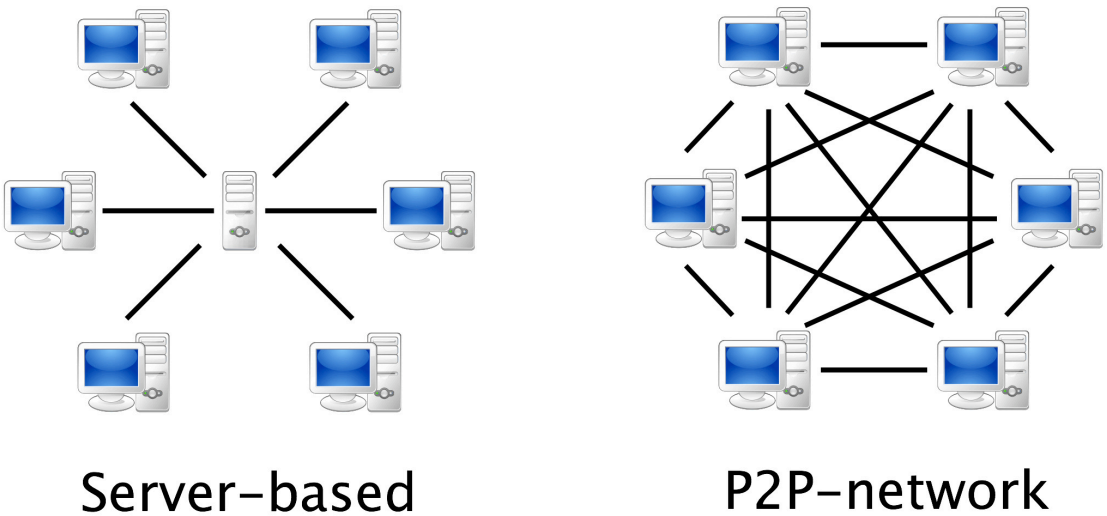
۲.۲ شبکه‌های نظیر به نظیر چه کاری انجام می‌دهند؟

شبکه‌های نظیر به نظیر ضمن آنکه برای به‌اشتراک‌گذاری منابع مورد استفاده قرار می‌گیرند، همچنین به کامپیوترها و دستگاه‌ها کمک می‌کنند در قالب یک گروه سرویس خاصی را ارائه کرده یا یک کار خاص را انجام دهند. با این وجود شبکه‌های فوق عمدتا به منظور به اشتراک‌گذاری فایل‌ها در اینترنت مورد استفاده قرار می‌گیرند. شبکه‌های p2p به دلیل آنکه به کامپیوترها اجازه می‌دهند به شبکه متصل شده و به‌طور همزمان فرآیند دریافت و ارسال فایل را انجام دهند ایده‌آل هستند.

۳.۲ مقایسه با معماری client/server

فرض کنید، مرورگر خود را باز کرده و برای دانلود یک فایل سایتی را باز می‌کنید. در این حالت، سایت به عنوان یک سرور کار کرده و کامپیوتر شما در نقش یک کلاینت فایل را دریافت می‌کند. این وضعیت مشابه جاده‌ای یک طرفه است. فایلی که شما دانلود می‌کنید ماشینی است که از نقطه A (سایت) به نقطه B (کامپیوتر شما) حرکت می‌کند. اگر همان فایل را از طریق یک شبکه نظیر به نظیر دانلود کنید، به‌طور مثال از طریق یک سایت تورنت به عنوان نقطه شروع، دانلود به صورت‌های مختلف انجام می‌شود. فایل در قالب بیت‌ها و بخش‌هایی که روی کامپیوترهای مختلف درون یک شبکه نظیر به نظیر قرار دارد دانلود می‌شود. در همان زمان ممکن است فایلی از کامپیوتر شما به سمت کامپیوترهایی که فایل را درخواست کرده‌اند ارسال شود. این وضعیت شبیه به یک جاده دو طرف است.

از طرف دیگر، در معماری client/server یک گره مرکزی وجود دارد که به همه درخواست‌ها رسیدگی می‌کند. اگر به هر دلیلی این گره مرکزی از کار بیافتند یا به اصطلاح down شود، در این صورت هیچ تبادلی در شبکه وجود نخواهد داشت اما در شبکه‌های p2p در صورت از کار افتادن یک گره، گره‌های دیگری وجود دارند که به درخواست‌ها پاسخ دهند.



شکل ۲: مقایسه معماری p2p و client/server

این شبکه‌ها به سختی از دسترس خارج می‌شوند. حتا اگر یکی از کامپیوترها خاموش شوند، کامپیوترهای دیگر همچنان به کار خود ادامه داده و ارتباط را برقرار می‌کنند. این شبکه‌ها تنها در صورتی از دسترس خارج می‌شوند که همه کامپیوترهای درون شبکه خاموش شوند. شبکه‌های نظیر به نظیر به شدت

[۱] https://www.shabakeh-mag.com/networking-technology/10495/

گسترش‌پذیر هستند. اضافه کردن کامپیوترها به سادگی امکانپذیر بوده و نیازی نیست پیکربندی خاصی روی سرور اعمال شود چون هر گره هم نقش سرور و هم نقش کلاینت را دارد.

۴.۲ مزایای شبکه‌ها p2p

۱. down کزدن آن‌ها بسیار سخت است، حتی اگر یکی از گره‌ها خاموش شود، شبکه به کار خود ادامه می‌دهد.

۲. برای متوقف کردن آن‌ها باید تمام گره‌های موجود در شبکه را خاموش کرد.

۳. به شدت مقیاس پذیر و گسترش پذیر هستند.

۴. اضافه کردن همتایان جدید آسان است زیرا نیازی به انجام تنظیمات مرکزی بر روی سرور مرکزی نیست .

۵. وقتی صحبت از اشتراک فایل است ، هرچه شبکه نظیر به نظیر بزرگتر باشد ، سرعت آن بیشتر است .

۶. هرچه فایل‌ها بیشتر در شبکه گسترش پیدا کند و به دست گره‌های بیشتری برسد، سرعت دریافت فایل‌ها افزایش می‌یابد.

۷. امکان دریافت همزمان فایل‌ها از چند گره موجود است که سبب افزایش سرعت دریافت می‌شود.

۵.۲ معایب شبکه‌های p2p

۱. چون گره باید ه زمان به چند گره سرویس بدهد، کارایی سیستم برای کاربر کم می‌شود.

۲. چون سرور مرکزی وجود ندارد، امکان پشتیبان گیری از فایل‌های موجود در شبکه وجود ندارد.

۳. وظیفه ایجاد امنیت در سیستم مانند عدم انتقال فایل‌های نامعتبر یا آلوده به عهده هر کاربر است و میدیریت مرکزی وجود ندارد.

۴. مدیریت transactionها در این سیستم‌ها بسیار دشوار است.

۵. به علت عدم وجود سرور کنترل کننده مرکزی، امکان بسیاری از حملات به کامپیوترها موجود در شبکه بالا است.

۶.۲ آیا باید از شبکه‌های p2p استفاده کرد؟

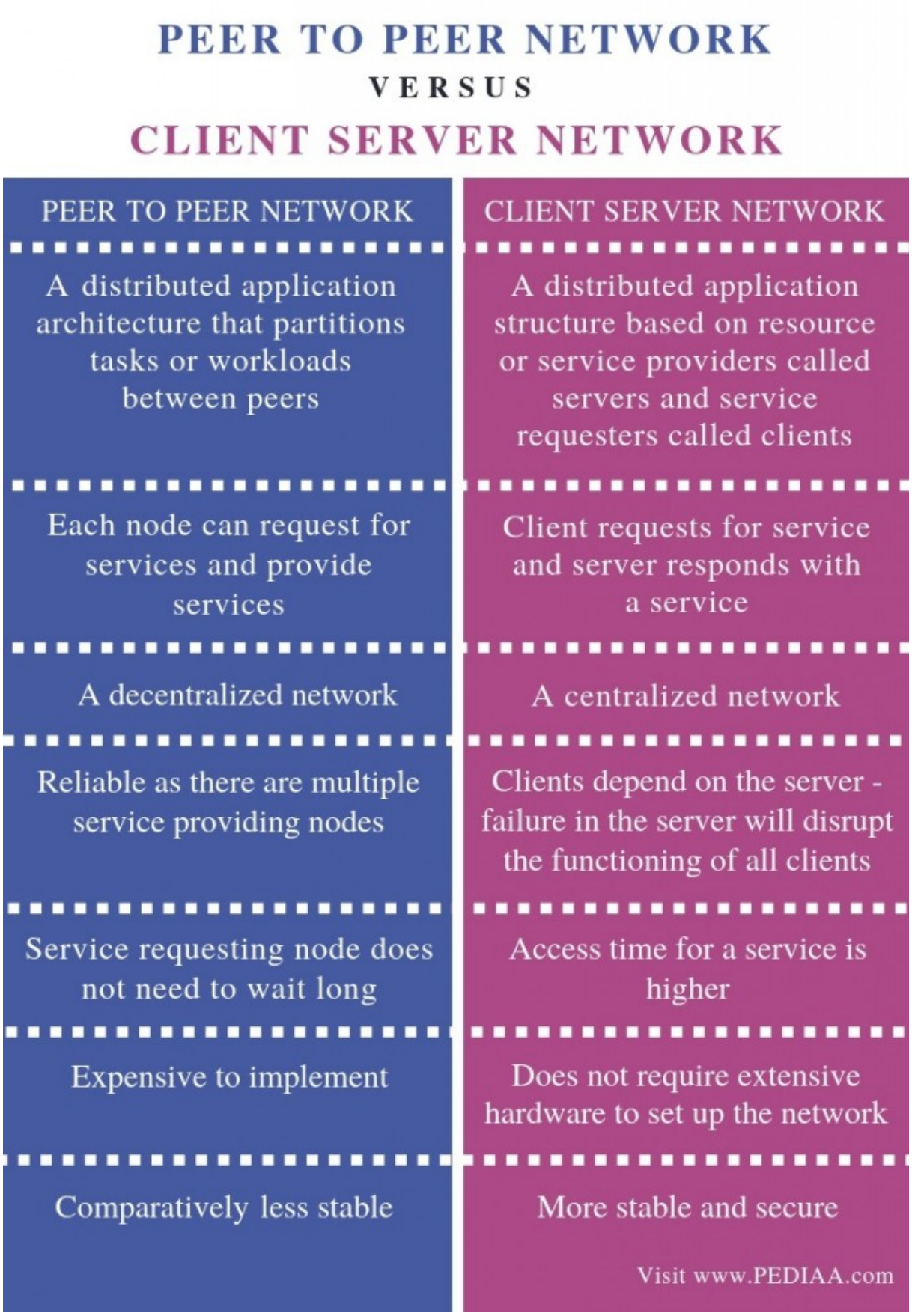
در دنیای تکنولوژی، به طور معمول هر معماری مزایا و معایب خاص خود را دارد. وجود این معایب و مزایای به این معناست که به طور معمول هیچ معماری بر دیگری برتری ندارد و باید با توجه به شرایط موجود و نوع استفاده از سیستم معماری مورد نظر را انتخاب و پیاده سازی کرد. ما در این‌جا از معایب و مزایای روش p2p سخن گفتیم، حال با توجه به نوع نیاز و ساختار مشکلی که قصد حل آن را داریم باید تصمیم گرفت که آیا از این نوع معماری بهره گرفت یا اجتناب کرد.
به طور کلی در ۴ مورد پیشنهاد می‌شود که از معماری نظیر به نظیر بهره گرفت:

۱. downloading services

۲. streaming services

۳. broadcasting services

۴. personal communication services



شکل ۳: مزایا و معایب دو نوع معماری p2p و client/server

۳ فرضیات پروژه

با توجه به ابهامات ایجاد شده در پزوژه، تعدادی فرض پایه برای انجام ءروژه در نظر گرفتیم و پیاده‌سازی‌ها بر اساس فرضیاتی که در ادامه ذکر می‌شود انجام شده است.

۱.۳ برخی تعاریف اولیه

۱. **همسایه دو طرفه : گرهی که هم شما به آن پیام می‌دهید و هم آن به شما پیام می‌دهد.**

۲. **همسایه تلاش‌شده : گرهی است که شما به آن پیام می‌دهید اما آن گره به شما پیام نمی‌دهد یا هنوز نداده است.**

۳. **همسایه یک طرفه : گرهی است که نه در لیست همسایه‌های تلاش‌شده و نه در لیست همسایه‌های دو طرفه قرار ندارد اما شما از آن پیام دریافت می‌کنید.**

۲.۳ فرایند پذیرش یک همسایه (ارتباط دو طرفه)

برای پذیرش یک همسایه که **ارتباطی دو طرفه است** دو حالت وجود دارد:

۱. **اگر شما شروع کننده ارتباط باشید (ارتباط در ابتدا یک طرفه است و در لیست همسایه‌های تلاش‌شده قرار می‌گیرد) ابتدا هر دو ثانیه یک پیام برای گره هدف ارسال می‌کنید. اگر گره هدف پاسخ شما را داد و شما در لیست همسایه‌های دو طرفه او بودید، ارتباط دو طرفه خواهد شد.**

۲. **اگر گره دیگری شروع کننده ارتباط باشد، هنگامی که یک پیام از گره دیگر دریافت کردید و تعداد همسایه‌های شما کمتر از حداکثر تعداد ممکن باشد، اورا به لیست همسایه‌های خود اضافه می‌کنید.**

۳.۳ فرایند انتخاب گره برای تلاش برای یافتن همسایه (ارتباط یک طرفه با قرار گیری در لیست همسایه تلاش‌شده)

هر گره هنگامی که تعداد همسایه‌هایش کمتر از حداکثر حد مجاز باشد باید تلاش کند که به گره‌های دیگر متصل شود تا شاید بتواند همسایه پیدا کند. فرایند به این صورت است که:

۱. **شرط این که گره اقدام به ایجاد ارتباط یک طرفه کند آن است که تعداد همسایه‌های آن از حداکثر تعداد همسایه‌های مجاز کمتر باشد.**

۲. **اگر شرط فوق بر قرار باشد، بلافاصله تعدادی از گره‌های دیگر که همسایه نباشند و در لیست همسایه تلاش‌شده قرار نداشته باشند انتخاب می‌شوند و تا زمانی که پاسخ نداده اند یا تعداد همسایه‌های ما به مقدار کافی نباشد هر دو ثانیه به آن‌ها یک پیام می‌فرستیم.** به این صورت و با این روش، به تمام گره‌هایی که همسایه ما نیستند هر دو ثانیه پیام داده می‌شود تا در صورت امکان همسایه ما شوند.

۳. **هنگامی که تعداد همسایه‌ها به اندازه کافی رسید، همه ارتباط‌های یک طرفه که ما شروع کننده هستیم لغو می‌شود و دیگر پیامی به آن ارتباط‌های یک طرفه ارسال نمی‌شود(لیست تهمسایه‌های تلاش‌شده خالی می‌گردد).**

۴.۳ فرایند غیر فعال شدن گره

هنگامی که یک گره غیر فعال می‌شود، **تمام همسایه‌های یک طرفه و دو طرفه خود را از دست می‌دهد. به این معنا که دیگر برای آن‌ها پیامی ارسال نمی‌کند و همچنین اگر پیامی دریافت کرد نیز پاسخ نمی‌دهد.** دقت کنید که همسایه‌ها نیز این گره را از لیست همسایه‌های خود خارج می‌کنند اما علت آن است که بیش از ۸ ثانیه به آن‌ها پیام نمی‌دهد زیرا مدت خاموش بودن ۲۰ ثانیه است.

۵.۳ فرمت پیام HELLo

در این‌جا چند فرض داریم:

۱. **تایپ پیام‌های ارسالی مقدار HELLO که خود نوع بسته است می‌باشد.**

۲. **در پیام ارسالی مقدار زمان آخرین پیام ارسال شده برابر با زمان کنونی است که می‌خواهد بسته را ارسال کند.**

۳. **لیست همسایه‌های ارسالی فقط و فقط لیست همسایه‌هایی است که ارتباط با آن‌ها دو طرفه است.**

۶.۳ ارتباط‌های یک طرفه ورودی

همان طور که قبلا نیز گفته شد:

۱. **اگر پیامی از غیر از همسایه‌ها بیاید و تعداد همسایه‌های ما کم باشد، به آن پاسخ داده و اورا به لیست همسایه‌ها اضافه می‌کنیم.**

۲. **اگر پیامی از غیر همسایه‌ها به ما ارسال شود و تعداد همسایه‌های ما به اندازه کافی باشد، صرفا ارسال کننده را به لیست کسانی که در تلاش برای ایجاد ارتباط با ما هستند یعنی لیست همسایه‌های یک طرفه اضافه می‌کنیم ولی پاسخی نمی‌دهیم. اینکار برای کشیدن توپولوژی شبکه مفید است. لازم به ذکر است اگر ۸ ثانیه پیامی از این گرهه دریافت نشد آن را از لیست یک طرفه‌های ورودی نیز حذف می‌کنیم. به این شکل همیشه توپولوژی بروز خواهد بود.**

۷.۳ دسترس پذیری یک گره

برای تعریف دسترس پذیری ما فرض کردیم که، **دسترس پذیری یک گره برابر است با مجموع تمام زمان‌هایی که گره مذکور در لیست همسایه‌های دوطرفه بوده تقسیم بر زمان کل اجرای شبیه‌سازی.**

۸.۳ توپولوژی شبکه از دید یک گره

در گرهی که قصد دارد توپولوژی از دید خود را بکشد اطلاعات **همسایه‌های دو طرفه خود، ارتباط‌های یک طرفه که خود آغازگر آن بوده، گره‌هایی که به این گره ارتباط یک طرفه دارند** را دارا است. علاوه بر این با توجه به ساختار پیام دریافتی از **همسایه‌های معتبر خود** اطلاعات مربوط به **همسایه‌های دوطرفه برای همسایه‌های معتبر خود** را نیز دارا است. بنابراین توپولوژی به این صورت است که:
شامل همه گره‌های موجود در لیست همسایه‌های دو طرفه و یک طرفه و تلاش‌شده که با گره فعلی به نحوی تعامل دارند یا گره فعلی با آن‌ها تعامل دارد.
همچنین نوع ارتباط بین گره‌ها به شکلی که برای گرهی که توپولوژی را می‌کشد مشخص باشد به کمک **یال‌های جهت دار** نمایش داده می‌شود. مشخص بودن به این معنا است که گره نام برده فقط از اطلاعات نام برده شده آگاه است و اطلاعات بیشتری ندارد.
نکته: فرض کردیم توپولوژی از دیدگاه یک گره خاموش معنا ندارد و هیچ ارتباطی نمایش داده نمی‌شود. به عبارت دیگر اطلاعاتی که نمایش داده می‌شود اصلا معتبر نیست و گره مذکور اطلاعی از سایر گره‌ها ندارد.

۹.۳ اتلاف پیام

فرض بر این است که نرخ اتلاف پیام نیز باید شبیه‌سازی گردد بنابراین با احتمال ۵ درصد یک پیام در مقصد پذیرفته نمی‌شود.

۱۰.۳ تعداد پیام‌های ارسالی و دریافتی و زمان ارسال

تعداد پیام‌های ارسالی و دریافتی فقط برای همسایه‌های دو طرفه محاسبه می‌شود. همچنین زمان ارسال پیام نیز زمان کنونی در نظر گرفته می‌شود.

۴ پیاده‌سازی برنامه شبیه‌ساز

برنامه سبیه‌ساز شامل دو بخش گره (Node) و main است. وظایف هر بخش در ادامه ذکر می‌شود.

۱.۴ پیاده‌سازی Node

۱.۱.۴ متغیرهای استفاده شده

در پیاده‌سازی یک گره از متغیرهای موجود در تصویر ۴ استفاده شده است.

- id : شناسه گره را نگهداری می‌کند.
- maxNeighborsCount : حداکثر تعداد همسایه‌های مجاز گره را نگهداری می‌کند.
- biNeighbors : لیست همسایه‌های دو طرفه را نگهداری می‌کند.
- attemptNeighbors : مجموعه گره‌هایی که ما به آن‌ها پیام ارسال کردیم اما پاسخ نداده اند. این متغیر برای رسم ارتباط‌های یک طرفه استفاده شده است.
- uniNeighbors : گره‌هایی که به گره فعلی پیام ارسال می‌کنند اما همسایه نیستند و گره فعلی نیز به آن‌ها پیام ارسال نمی‌کند. این متغیر برای رسم ارتباط‌های یک طرفه استفاده شده است.
- socket : سوکت UDP استفاده شده برای ارسال و دریافت را نگهداری می‌کند.

```python
def __init__(self,nodeId,baseTime,totalTime = 300,N=3):
        self.id = nodeId
        self.maxNeighborsCount = N
        self.biNeighbors = []
        self.attemptNeighbors = set()
        self.uniNeighbors = set()
        self.socket = socket.socket(socket.AF_INET,socket.
SOCK_DGRAM)
        self.socket.bind(('127.0.0.1',0))
        self.recvTimes = dict()
        self.sendTimes = dict()
        self.baseTime = baseTime
        self.recvTimers = dict()
        self.sendTimers = dict()
        self.uniNeighborsRecvTimers = dict()
        self.attemptNeighborsSendTimers = dict()
        self.totalTime = totalTime
        self.state = 'Active'
        self.sendHistory = dict()
        self.recvHistory = dict()
        self.neighborsOfNeighbors = dict()
        self.neighborsAvailability = dict()
        self.neighborsEntranceTime = dict()
```

شکل ۴: تابع init از Node

- recvTimes : زمان آخرین پیام دریافتی از هر گره را نگهداری می‌کند.
- sendTimes : زمان آخرین پیام ارسالی به هر گره را در خود ذخیره می‌کند.
- baseTime : در این پروژه تمامی زمان‌ها نسبت به زمان شروع اولیه محاسبه می‌شود و زمان‌ها نسبی است. این متغیر برای نگهداری زمان شروع است.
- recvTimers : چون در این پروژه در زمان‌های مشخص ۲ و ۸ ثانیه باید اقداماتی صورت بگیرد، این متغیر تایمرهایی که باید هر ۸ ثانیه فعال شوند و گره مناسب را از لیست گره‌ها خارج کنند نگهداری میکند.
- sendTimers : تایمرهای استفاده شده برای ارسال ۲ ثانیه یک بار پیام به همسایه‌های دو طرفه در این متغیر نگهداری می‌شود.
- uniNeighborsRecvTimers : این متغیر تایمرهای مربوط به کسانی که به گره فعلی پیام می‌دهند اما همسایه دو طرفه نیستند را نگهداری می‌کند و هدف آن این است که در پایان توپولوژی را دقیق‌تر رسم کند.
- attemptNeighborsSendTimers : تایمرهای مربوط به ارسال به همسایه‌های یک طرفه برای زمانی که تلاش می‌کنیم با آن‌ها همسایه دو طرفه شویم را نگهداری می‌کند.
- totalTime : کل زمان اجرا را نگهداری می‌کند.

- state : وضعیت گره مبنی بر فعال یا غیر فعال بودن را نگهداری می‌کند.
- sendHistory : متغیر تعداد پیام‌های ارسالی به هر گره را نگهداری می‌کند.
- recvHistory : این متغیر تعداد پیام‌های دریافتی از هر گره را نگهداری می‌کند.
- neighborsOfNeighbors : این متغیر همسایه‌های دو طرفه ارسالی از طرف هر همسایه خود را نگهداری می‌کند.
- neighborsAvailability : این متغیر مدت زمانی که هر گره در لیست همسایه‌های دو طرفه گره فعلی بوده است را نگهداری می‌کند.
- neighborsEntranceTime : زمان آخرین دفعه که یک گره وارد لیست همسایه‌های دو طرفه شده را نگهداری می‌کند.
- addresses : این متغیر آدرس تمام گره‌های موجود شامل ip و port را نگهداری می‌کند. (در تابع start دریافت می‌شود.)
- recvThread : این متغیر thread است که وظیفه دریافت پیام را بر عهده دارد. را نگهداری می‌کند. (در تابع start دریافت می‌شود.)
- selectThread : این متغیر thread است که وظیفه انتخاب همسایه جدید را بر عهده دارد. را نگهداری می‌کند. (در تابع start دریافت می‌شود.)

۲.۱.۴ تابع start

این تابع از بیرون صدا زده شده و باعث شروع به کار گره می‌شود. این تابع توسط بخش main شبیه‌ساز صدا زده می‌شود و فقط در ابتدای آغاز شبیه‌سازی استفاده می‌گردد. این تابع لیست کل آدرس‌ها را دریافت کرده و آدرس خود را حذف می‌کند سپس thread های دریافت و انتخاب همسایه را فعال می‌کند.

```
def start(self,addresses):
        addresses = addresses[:]
        addresses.remove(self.socket.getsockname())
        self.addresses = addresses
        self.recvThread = threading.Thread(target=self
.__recvMessage__,daemon=True)
        self.selectThread = threading.Thread(target=self
.__selectNewNeighbor__,daemon=True)
        self.recvThread.start()
        self.selectThread.start()
```

شکل ۵: تابع start

۳.۱.۴ تابع deActive

این تابع وظیفه دارد که گره را غیر فعال کند. این تابع توسط بخش main شبیه‌ساز صدا زده می‌شود و به مدت ۲۰ ثانیه گره فعلی را خاموش می‌کند. برای خاموش کردن یک گره، تمام تایمرهای فعال آن گره و همچنان تمام همسایه‌های یک طرفه و دو طرفه و گره‌های تلاش‌شده باید پاک شوند. سپس برای شروع به کار دوباره، یک تایمر برای ۲۰ ثانیه بعد تنظیم شده و گره مجددا راه اندازی می‌گردد. کد مربوط به این بخش در تصویر ۶ قابل مشاهده است.

۴.۱.۴ تابع ___active___

این تابع وظیفه فعال سازی مجدد گره را برعهده دارد و توسط خود گره به وسیله تایمر تنظیم شده در تابع deActive صدا زده می‌شود. تنها عملی که این تابع انجام می‌دهد آن است که وضعیت گره را به فعال تبدیل می‌کند. دقت کنید متغیر state برای توابع دیگر نقش کنترلی دارد و مقدار آن رفتار آن‌ها را تغییر می‌دهد. بنابراین فعال یا غیر فعال کردن گره به کمک تغییر مقدار آن امکان پذیر است. کد مربوط به پیاده‌سازی این تابع در تصویر ۷ قابل مشاهده است.

```
def deActive(self,time=20):
        self.state = 'deActive'
        self.attemptNeighbors.clear()
        self.uniNeighbors.clear()
        for timer in self.recvTimers.values():
            timer.cancel()
        for timer in self.sendTimers.values():
            timer.cancel()
        for timer in self.attemptNeighborsSendTimers.values():
            timer.cancel()
        for timer in self.uniNeighborsRecvTimers.values():
            timer.cancel()
        for neighbor in self.biNeighbors:
            self.__removeFromNeighbors__(neighbor)
        timer = threading.Timer(time,self.__active__)
        timer.start()
```

شکل ۶: تابع deActive

```
def __active__(self):
        self.state = 'Active'
```

شکل ۷: تابع ___active___

۵.۱.۴ ___selectNewNeighbor___

این تابع در thread ایجاد شده در تابع start اجرا می‌شود و هدف از اجرای آن، این است که اگر تعداد همسایه‌ها از تعداد مورد نظر کم‌تر بود، تعدادی گره را انتخاب کرده و تلاش کند به آن‌ها متصل شود. کد مربوط به این تابع در تصویر ۸ قابل مشاهده است.

این تابع تا زمانی که زمان شبیه‌سازی پایان نیافته باشد اجرا می‌شود و هنگامی که تعداد همسایه‌های گره از تعداد مشخص که ۳ است کمتر شد همسایه جدید انتخاب می‌کند که در لیست همسایه‌های دوطرفه و یک طرفه خود نباشد سپس به آن پیام ارسال می‌کند. این ارسال پیام با فراخوانی تابع ___sendHelloMessagePeriodicallyToAttemptNeighbors___ اتفاق می‌افتد که درون آن بعدتر توضیح داده خواهد شد. فعلا کافی است بدانیم که هر ۲ ثانیه پیامی به گره انتخاب شده ارسال می‌گردد. این عمل تا زمانی اتفاق می‌افتد که تعداد همسایه‌ها به تعداد مشخص ۳ برسد. هر گرهی که انتخاب شود به لیست همسایه‌های یک طرفه اضافه می‌شود.

```
def __selectNewNeighbor__(self):
        while time.time()-self.baseTime < self.totalTime:
            while (self.state == 'deActive' or len(self.biNeighbors) == self.maxNeighborsCount) and time.time() -
self.baseTime < self.totalTime:
                continue
            if not (time.time()-self.baseTime < self.totalTime):
                break
            addr = random.choice(self.addresses)
            if(not addr in self.biNeighbors) and (not addr in self.attemptNeighbors) and (len(self.biNeighbors) <
self.maxNeighborsCount):
                self.attemptNeighbors.add(addr)
                self.__sendHelloMessagePeriodicallyToAttemptNeighbors__(addr)
```

شکل ۸: تابع ___selectNewNeighbor___

۶.۱.۴ تابع ___sendHelloMessagePeriodicallyToBiNeighbors___

این تابع در تایمرهایی که به صورت دوره‌ای [۲] پیام ارسال می‌کند استفاده می‌شود. کد مربوط به این تابع در تصویر ۹ قابل مشاهده است. **تابع ___sendHelloMessagePeriodicallyToAttemptNeighbors___ نیز وجود دارد که فرایند ارسال به همسایه‌های تلاش‌شده را بر عهده دارد ولی چون مکانیزم کاری هر دو یکسان است، فقط یکی را توضیح می‌دهیم و دیگری نیز مشابه است. کد مربو به این تابع در تصویر ۱۰ قابل مشاهده است.**

```
def __sendHelloMessagePeriodicallyToBiNeighbors__(self,addr):
        if self.state == 'deActive':
            return
        self.sendTimes[addr] = time.time() - self.baseTime
        self.sendHistory[addr] = self.sendHistory.get(addr,0) + 1
        self.socket.sendto(self.__helloMessage__(addr),addr)
        self.sendTimers[addr] = threading.Timer(2,self.__sendHelloMessagePeriodicallyToBiNeighbors__,(addr,))
        self.sendTimers[addr].start()
```

شکل ۹: تابع ___sendHelloMessagePeriodicallyToBiNeighbors___

```
def __sendHelloMessagePeriodicallyToAttemptNeighbors__(self,addr):
        if self.state == 'deActive':
            return
        self.sendTimes[addr] = time.time() - self.baseTime
        self.socket.sendto(self.__helloMessage__(addr),addr)
        self.attemptNeighborsSendTimers[addr] = threading.Timer(2,self.__sendHelloMessagePeriodicallyToAttemptNeighbors__,(addr,))
        self.attemptNeighborsSendTimers[addr].start()
```

شکل ۱۰: تابع ___sendHelloMessagePeriodicallyToAttemptNeighbors___

در این تابع ابتدا زمان ارسال پیام به گره مورد نظر برابر با زمان کنونی قرار می‌گیرد (پیش تر گفتیم که زمان نسبی است) سپس پیام با فراخوانی تابع ___helloMessage___ ایجاد شده و ارسال می‌گردد. سپس تعداد پیام‌های ارسالی بروزرسانی می‌گردد و یک تایمر برای ۲ ثانیه بعد روی همین تابع تنظیم می‌گردد. این بخش باعث می‌شود که تابع به صورت دوره‌ای اجرا گردد.

۷.۱.۴ تابع ___helloMessage___

این تابع وظیفه پیاده‌سازی پیام HELLO برای ارسال به سایر گره‌ها را بر عهده دارد. کد مربوط به این تابع در تصویر ۱۱ آمده است.

[۲] priodically

```
def __helloMessage__(self,addr):
        hello = {
            'id' : self.id,
            'ip' : self.socket.getsockname()[0],
            'port' : self.socket.getsockname()[1],
            'type' : 'HELLO',
            'neighbors' : self.biNeighbors,
            'last_send_time' : self.sendTimes.get(addr,math.nan),
            'last_recv_time' : self.recvTimes.get(addr,math.nan)
        }
        return json.dumps(hello).encode()
```

شکل ۱۱: تابع ایجاد پیام HELLO

در این پیام id و ip و port و زمان آخرین دریافت و ارسال به گره مقصد و همچنین نوع پیام که HELLO است قرار می‌گیرد و سپس به فرمت json تبدیل شده و برای ارسال آماده می‌گردد.

۸.۱.۴ تابع ___removeFromNeighbors___

این تابع توسط تایمرهایی که برای ۸ ثانیه تنظیم شده‌اند اجرا می‌شود و ظیفه آن، این است که همسایه‌ای که بعنوان ورودی داده می‌شود را از لیست همسایه‌های دو طرفه خارج کند. کد مربوط به این تابع در تصویر ۱۲ آمده است.

```
def __removeFromNeighbors__(self,addr):
        sendTimer = self.sendTimers.get(addr,None)
        if not (sendTimer is None):
            sendTimer.cancel()
        if addr in self.biNeighbors:
            self.biNeighbors.remove(addr)
        end = time.time()
        self.neighborsAvailability[addr] = self.neighborsAvailability.get(addr,0) + (end - self.neighborsEntranceTime.get(addr,end-
self.baseTime) -  self.baseTime)
        if self.neighborsEntranceTime.get(addr,None):
            del self.neighborsEntranceTime[addr]
```

شکل ۱۲: تابع ___removeFromNeighbors___

در این تابع ابتدا تایمر ارسال مربوط به گرهی که می‌خواهیم حذف کنیم گرفته می‌شود و سپس غیر فعال می‌گردد. سپس گره مورد نظر از لیست گره‌های دو طرفه حذف می‌شود. سپس با استفاده از زمان اضافه شدن به لیست همسایه‌ها که ذخیره شده کمک می‌گیریم و مدت زمانی که همسایه بوده را به کل مدت زمان همسایه بودن می‌افزاییم. این عمل برای محاسبه دسترس پذیری مورد استفاده قرار می‌گیرد. در پایان نیز زمان ورودی قبلی حذف می‌شود تا روی ورودهای بعدی تاثیر نگذارد. **تابع ___removeFromInputUniNeighbors___ نیز وجود دارد که وظیفه آن این است که اگر کسی که به شما پیام می‌دهد ولی به دلیل تکمیل همسایه‌های شما درون لیست همسایه‌های شما نیست، به مدت ۸ ثانیه پیام نداد، از لیست گره‌هایی با این ویژگی خارج شود. این تابع برای آن است که در پایان توپولوژی بهتری از شبکه کشیده شود. کد مربوط به این تابع در تصویر ۱۳ آمده است.**

```
def __removeFromInputUniNeighbors__(self,addr):
        if addr in self.uniNeighbors:
            self.uniNeighbors.remove(addr)
```

شکل ۱۳: تابع ___removeFromInputUniNeighbors___

۹.۱.۴ تابع ___recvMessage___

این تابع وظیفه دریافت پیام‌های ارسالی به گره کنونی را دارد. کد مربوط به این تابع در تصویر ۱۴ قابل مشاهده است.

```
def __recvMessage__(self):
        while time.time() - self.baseTime < self.totalTime:
            try:
                self.socket.settimeout(1)
                msg , addr = self.socket.recvfrom(65535)
                msg = json.loads(msg.decode())
                recvNeighbors = [tuple(x) for x in msg['neighbors']]
                shouldMissMessage = random.choices([True,False],[0.05,0.95],k=1)
                if shouldMissMessage[0]:
                    continue
                if self.state == 'deActive':
                    continue
                myAddr = self.socket.getsockname()
                if (addr in self.attemptNeighbors) and (myAddr in recvNeighbors) and (len(self.biNeighbors) <
self.maxNeighborsCount):
                    self.__addToBiNeighborsWhenIsInAttemptNeighbors__(addr)
                    self.__processMsg__(msg,addr)
                elif (addr in self.biNeighbors) and (myAddr in recvNeighbors):
                    self.__processMsg__(msg,addr)
                elif (addr not in self.attemptNeighbors) and (addr not in self.biNeighbors) and (len(self
.biNeighbors)<self.maxNeighborsCount):
                    self.__addToBiNeighborsWhenIsNotInAttemptNeighbors__(addr)
                    self.__processMsg__(msg,addr)
                elif (addr not in self.biNeighbors) and len(self.biNeighbors) == self.maxNeighborsCount:
                    self.__addToUniNeighbors__(addr)
                if len(self.biNeighbors) == self.maxNeighborsCount:
                    self.attemptNeighbors.clear()
                    for t in self.attemptNeighborsSendTimers.values():
                        t.cancel()
            except:
                continue
        self.__terminate__()
```

شکل ۱۴: تابع ___recvMessage___

این تابع در تابع start و روی thread مربوط به دریافت پیام اجرا می‌گردد. همانگونه که مشاهده می‌کنید این تابع تا زمانی که زمان شبیه‌سازی پایان نیافته اجرا می‌شود. همچنین هنگامی که گره در وضعیت غیر فعال باشد، اجرای این تابع متوقف شده و در حلقه گیر می‌کند. **برای شبیه‌سازی نرخ از دست دادن پیام، فرض کردیم که با احتمال ۵ درصد پیام دریافتی را بررسی نکند. به این ترتیب این نرخ از دست رفتن پیام شبیه‌سازی گردیده است. علت انتخاب شبیه‌سازی به این شکل آن بود که در صورت پروژه ذکر شده که فرستنده از دست ر فتن پیام خود را متوجه نمی‌شود پس بهترین مکان برای پیاده‌سازی آن سمت گیرنده است.** پس از آن که بسته دریافت شد و تلف نشد، تعداد بسته‌های دریافتی از فرستنده را بروز رسانی می‌کنیم. سپس ۴ حالت روی خواهد داد:

۱. اگر فرستنده پیام مورد نظر در لیست همسایه‌های تلاش‌شده ما قرار داشته باشد و ما در لیست همسایه‌های دو طرفه او قرار داشته باشیم که به معنای آن است که به درخواست ما پاسخ داده و تعداد همسایه‌های ما کمتر از تعداد حداکثر مجاز باشد، که در این‌جا مقدار آن ۳ است، تابع

___addToBiNeighborsWhenIsInAttemptNeighbors___ و سپس تابع ___processMsg___ اجرا می‌گردد. هر یک از این توابع در ادامه توضیح داده خواهند شد.

۲. اگر فرستنده پیام مورد نظر در لیست همسایه‌های دو طرفه ما قرار داشته باشد و ما نیز در لیست همسایه‌های دو طرفه او قرار داشته باشیم، تابع ___processMsg___ اجرا می‌گردد. این تابع در ادامه توضیح داده خواهد شد.

۳. اگر فرستنده پیام مورد نظر در لیست همسایه‌های تلاش‌شده و دو طرفه نبود، کسی است که به ما پیام می‌دهد و در تلاش است ما را به لیست همسایه‌های دوطرفه خود اضافه کند پس اگر ما نیز تعداد همسایه‌هایمان کم باشد باید اورا به لیست همسایه‌های دو طرفه خود اضافه کنیم. به این منظور باید تابع ___addToBiNeighborsWhenIsNotInAttemptNeighbors___ و ___processMsg___ اجرا کردد. پیاده‌سازی آن در ادامه شرح داده می‌شود.

۴. اگر فرستنده پیام مورد نظر در لیست همسایه‌های تلاش‌شده و دو طرفه نبود، کسی است که به ما پیام می‌دهد و در تلاش است ما را به لیست همسایه‌های دوطرفه خود اضافه کند اما اگر ظرفیت ما کامل باشد و نتوانیم همسایه دو طرفه جدید بپذیریم، به پیام پاسخ نمی‌دهیم اما برای کشیدن بهتر توپولوژی، باید این گره را به لیست گره‌هایی که همسایه یک طرفه یا دو طرفه ما نیستند ولی به ما پیام می‌دهند اضافه کنیم. به این منظور تابع ___addToUniNeighbors___ اجرا می‌گردد. پیاده‌سازی این تابع در ادامه توضیح داده می‌شود.

دقت کنید در هر حلقه اگر تعداد همسایه‌ها به اندازه حداکثر تعداد مجاز یعنی ۳ رسیده باشد، لیست همسایه‌های تلاش‌شده (آن‌هایی که ما به آن‌ها پیام می‌دادیم ولی آن‌ها جواب نمی‌داند) خالی می‌شود و تایمرهای ارسال پیام به آن‌ها نیز لغو می‌گردد. در پایان برای اینکه شبیه‌سازی خاتمه یابد، تابع ___terminate___ فراخوانی می‌گردد. کد مربوط به این تابع در تصویر ۱۵ قابل مشاهده است. این تابع ابتدا زمان کنونی را از زمان ورود هر گره که در لیست گره‌های دو طرفه است کم می‌کند و با زمان دسترس بودن اضافه می‌کند. سپس تمامی تایمرهایی که هنوز فعال هستند را غیر فعال می‌کند. به این شکل شبیه‌سازی برای این گره پایان می‌یابد.

```
def __terminate__(self):
        end = time.time()
        for addr in self.biNeighbors:
            self.neighborsAvailability[addr] = self.neighborsAvailability.get(addr,0)  + (end - self
.neighborsEntranceTime.get(addr,end-self.baseTime) - self.baseTime)
            if self.neighborsEntranceTime.get(addr,None):
                del self.neighborsEntranceTime[addr]
        for timer in self.recvTimers.values():
            timer.cancel()
        for timer in self.sendTimers.values():
            timer.cancel()
        for timer in self.uniNeighborsRecvTimers.values():
            timer.cancel()
        for timer in self.attemptNeighborsSendTimers.values():
            timer.cancel()
```

شکل ۱۵: تابع ___terminate___

پیاده‌سازی تابع ___addToBiNeighborsWhenIsInAttemptNeighbors___ در تصویر ۱۶ آمده است. در این تابع ابتدا مقدار دسترس پذیری برابر با مقدار قبلی (در صورتی که قبلا نیز همسایه بوده باشد) یا برابر با مقدار صفر قرار داده می‌شود. سپس گره به لیست همسایه‌های دو طرفه افزوده می‌شود و از لیست همسایه‌های تلاش‌شده خارج شده و زمان ورودش به لیست همسایه‌های دو طرفه نیز ثبت می‌گردد. سپس تایمر مربوط به همسایه تلاش‌شده بودن غیر فعال می‌شود. در پایان نیز تابع مربوط به ارسال دوره‌ای پیام برای آن فراخوانی می‌گردد.

پیاده‌سازی تابع ___addToBiNeighborsWhenIsNotInAttemptNeighbors___ در تصویر ۱۷ آمده است. چون در این حالت قبلا در لیست همسایه‌های تلاش‌شده نبوده است، فقط کافی است آن را به دو طرفه‌ها افزود و سپس زمان ورود را ذخیره کرد و تابع مربوط به ارسال دوره‌ای پیام را فراخوانی کرد.

پیاده‌سازی تابع ___processMsg___ در تصویر ۱۸ آمده است. در این تابع ابتدا تایمری که مربوط به عدم دریافت پیام برای مدت ۸ ثانیه بود لغو می‌شود سپس زمان دریافت پیام ذخیره شده و پس از آن یک تایمر جدید برای ۸ ثانیه بعد فعال می‌گردد که اگر پیامی نیامد، از لیست همسایه‌های دو طرفه حذف گردد. در پایان نیز لیست همسایه‌های موجود در پیام استخراج شده و نگهداری می‌شود و تعداد پیام‌های دریافتی از همسایه‌ها یکی زیاد می‌شود.

پیاده‌سازی تابع ___addToUniNeighbors___ در تصویر ۱۹ آمده است. در این تابع فرستنده پیام به لیست گره‌هایی که به ما پیام میفرستند اما همسایه ما نیستند یعنی گره‌های یک طرفه افزوده می‌شود، سپس اگر تایمری داشته باشد که برای مدت ۸ ثانیه پیام نفرستد آن را حذف کند نیز غیر فعال می‌شود و دوباره، تایمر جدیدی تنظیم می‌گردد. همانطور که پیش‌تر نیز ذکر کردیم، این لیست فقط و فقط برای رسم بهتر توپولوژی در پایان اجرا استفاده می‌گردد و کاربرد دیگری ندارد. بنابراین اگر قرار بر رسم توپولوژی نباشد می‌توان به راحتی این‌ها را حذف کرد.

```
def __addToBiNeighborsWhenIsInAttemptNeighbors__(self,addr):
        self.neighborsAvailability[addr] = self.neighborsAvailability.get(addr,0)
        self.biNeighbors.append(addr)
        self.neighborsEntranceTime[addr] = time.time() - self.baseTime
        self.attemptNeighbors.remove(addr)
        if addr in self.attemptNeighborsSendTimers.keys():
            self.attemptNeighborsSendTimers[addr].cancel()
        self.__sendHelloMessagePeriodicallyToBiNeighbors__(addr)
```

شکل ۱۶: تابع ___addToBiNeighborsWhenIsInAttemptNeighbors___

```
def __addToBiNeighborsWhenIsNotInAttemptNeighbors__(self,addr):
        self.biNeighbors.append(addr)
        self.neighborsEntranceTime[addr] = time.time() - self.baseTime
        self.__sendHelloMessagePeriodicallyToBiNeighbors__(addr)
```

شکل ۱۷: تابع ___addToBiNeighborsWhenIsNotInAttemptNeighbors___

```
def __processMsg__(self,msg,addr):
        neighbors = [tuple(x) for x in msg['neighbors']]
        if self.socket.getsockname() in neighbors:
            self.recvHistory[addr] = self.recvHistory.get(addr,0) + 1
        timer = self.recvTimers.get(addr,None)
        if not (timer is None):
            timer.cancel()
        self.recvTimes[addr] = time.time() - self.baseTime
        self.recvTimers[addr] = threading.Timer(8,self.__removeFromNeighbors__,(addr,))
        self.recvTimers[addr].start()
        self.neighborsOfNeighbors[addr] = [tuple(x) for x in msg['neighbors']]
```

شکل ۱۸: تابع ___processMsg___

```
def __addToUniNeighbors__(self,addr):
        self.uniNeighbors.add(addr)
        timer = self.uniNeighborsRecvTimers.get(addr,None)
        if timer:
            timer.cancel()
        self.uniNeighborsRecvTimers[addr] = threading.Timer(8,self.__removeFromInputUniNeighbors__,args=(addr,))
        self.uniNeighborsRecvTimers[addr].start()
```

شکل ۱۹: تابع ___addToUniNeighbors___

۱۰.۱.۴ تابع report

این تابع برای دریافت گزارش عملکرد گره در پایان شبیه‌سازی است که توسط بخش main شبیه‌ساز صدا زده می‌شود. کد مربوط به این بخش در تصویر ۲۰ آمده است. این تابع باید اطلاعات لازم از جمله توپولوژی را نیز ایجاد نماید. برای ساخت توپولوژی، مجموعه تمام گره‌های شناخته شده شامل گره‌های

موجود در لیست همسایه‌های دو طرفه و همسایه‌های یک طرفه و تلاش‌شده می‌باشد. مجموعه یال‌ها شامل موارد زیر است:

۱. برای هر همسایه دو طرفه، دویال از گره فعلی به همسایه و از همسایه به گره فعلی را اضافه می‌کنیم.

۲. برای همسایه‌های یک طرفه یک یال از گره همسایه به گره فعلی اضافه می‌کنیم.

۳. برای گره‌هایی که همسایه تلاش‌شده هستند یک یال از گره فعلی به آن گره اضافه می‌کنیم.

۴. برای همسایه‌های همسایه‌های دو طرفه که در پیام‌ها دریافتی از آن‌ها وجود دارد، برای گره‌هایی که در لیست گره‌های همسایه دو طرفه با ما هستند، یک یال از همسایه‌ی همسایه‌ما به آن و برعکس اضافه می‌کنیم.

در موارد فوق اگر گره خاموش باشد، گره‌های شناسایی شده در لحظه آخر فقط خودش است و هیچ یالی نیز وجود ندارد. سپس برای هر یک از گره‌هایی که تا به حال با آن‌ها همسایه دو طرفه بودیم، اطلاعاتی شامل ip و port و تعداد پیام‌های ارسالی و دریافتی و مدت زمانی که در دسترس بوده تقسیم بر کل زمان را قرار می‌دهیم. در پایان نیز همسایه‌های کنونی را در لیست جداگانه قرار داده می‌دهیم. سپس آن‌ها را در فایل json ذخیره می‌کنیم.

```
def report(self,address="./json_output/"):
        with open(address+'report_node_'+str(self.id)+".json",'w') as file:
            vertex = set(self.biNeighbors+[self.socket.getsockname()]+list(self.attemptNeighbors)+list(self
.uniNeighbors))
            for neighbor,biNeighbors in self.neighborsOfNeighbors.items():
                if neighbor in self.biNeighbors:
                    vertex = vertex.union(set(biNeighbors))
            edges = set()
            for neighbor in self.biNeighbors:
                edges.add((self.socket.getsockname(),neighbor))
                edges.add((neighbor,self.socket.getsockname()))
            for neighbor in self.attemptNeighbors:
                edges.add((self.socket.getsockname(),neighbor))
            for neighbor in self.uniNeighbors:
                edges.add((neighbor,self.socket.getsockname()))
            for addr,biNeighbors in self.neighborsOfNeighbors.items():
                for neighbor in biNeighbors:
                    if neighbor in vertex and addr in self.biNeighbors:
                        edges.add((addr,neighbor))
                        edges.add((neighbor,addr))
            nodeReport = {
                'state' : self.state,
                'address' : self.socket.getsockname(),
                'all_neighbors' : [
                    {
                        'ip' : addr[0],
                        'port' : addr[1],
                        'send_to_count' : self.sendHistory.get(addr,0),
                        'recv_from_count' : self.recvHistory.get(addr,0),
                        'availability' : availability/(time.time()-self.baseTime)
                    } for addr,availability in self.neighborsAvailability.items()
                ],
                'current_neighbors' : list(set(self.biNeighbors)),
                'topology' : {
                    'vertex' : list(vertex) if self.state == 'Active' else [self.socket.getsockname()],
                    'edges' : list(edges) if self.state == 'Active' else []
                }
            }
            json.dump(nodeReport,file)
```

شکل ۲۰: تابع report

۲.۴ main

این قسمت از شبیه‌ساز عملا نقش کنترلر برای اجرا و متوقف و فعال و غیر فعال کردن گره‌ها را دارا است. در پیاین نیز این بخش از شبیه‌ساز است که توپولوژی گرافیکی را رسم می‌نماید.

۱.۲.۴ تابع prepare

این تابع وظیفه اماده‌سازی اولیه برای اجرای شبیه‌سازی را بر عهده دارد. این آماده‌سازی شامل ایجاد پوشه‌های مورد نیاز برای ذخیره نتایج شبیه‌سازی است. کد مربوط به این تابع در تصویر ۲۱ است.

```
def prepare():
    if not os.path.exists('./json_output'):
        os.makedirs('./json_output')
    if not os.path.exists('./network_graphs'):
        os.makedirs('./network_graphs')
```

شکل ۲۱: تابع prepare

۲.۲.۴ تابع runNetwork

این تابع وظیفه مدیریت و اجرای شبکه در زمان پیاده‌سازی را بر عهده دارد. کد مربوط به این تابع در تصویر ۲۲ آمده است. این تابع ابتدا زمان کنونی را به عنوان زمان آغاز شبیه‌سازی (زمان پایه) نگهداری کرده و سپس ۶ تا گره ایجاد می‌کند و به هرکدام id و زمان پایه را می‌دهد. سپس از هرکدام آدرس شامل ip و port را دریافت کرده و با فراخوانی start آدرس‌ها را به گره‌ها می‌دهد و آن‌ها را اجرا می‌کند. سپس برای ۱۰ ثانیه بعد یک فراخوانی تنظیم می‌کند که یک گره به صورت تصادفی غیر فعال شود. پیاده سازی این بخش را جلوتر بررسی می‌کنیم. در پایان نیز منتظر می‌ماند تا همه گره‌ها و thread آن‌ها با thread اصی ملحق شود و بعد از آن گزارش را برای هر یک از گره‌ها دریافت می‌کند.

برای انتخاب تصادفی یک گره و غیر فعال کردن آن تابع deActiveNodeRandomly فراخوانی می‌شود. کد مربوط به این تابع در تصویر ۲۳ قابل مشاهده است. در این تابع ابتدا یک گره که وضعیت آن، active باشد انتخاب شده و تابع deActive روی آن صدا زده می‌شود. سپس یک تایمر برای ۱۰ ثانیه بعد تنظیم می‌شود تا یک گره دیگر غیر فعال گردد. دقت کنید خود گره وظیفه دارد بعد از زمان مشخص ۲۰ ثانیه خود را فعال نماید به همین دلیل فراخوانی برای فعال کردن دوباره گره در این جا تنظیم نمی‌شود.

۳.۲.۴ تابع drawGraphs

این تابع وظیفه دارد فایل‌های json ایجاد شده را بخواند و گراف معادل آن‌ها را رسم نماید. کد مربوط به این تابع در تصویر ۲۴ آمده است. در این تابع به کمک کتابخانه کار با گراف networkx گراف مربوط به شبکه از دید گره تولید کننده گراف را تولید می‌کند. در گراف‌های حاصل، یال‌های یک طرفه از گره اول به گره دوم به معنای وجود ارتباط در جهت گره اول به گره دوم است. همچنین وجود یال دوگانه به معنای وجود ارتباط دو طرفه است. **نکته دارای اهمیت آن است که گره‌هایی که بارنگ زرد مشخص شده گرهی است که گراف را رسم کرده و این گراف حاصل تصور آن گره از شبکه است. همچنین گره قرمز رنگ گرهی است که خود رسم کننده گراف است اما در حالت غیر فعال قرار دارد. بنابراین این نمودار فقط شامل یک گره است و بقیه گره‌ها حضور ندارند زیرا گره رسم کننده غیر فعال است و اطلاعات آن معنا ندارد و باطل است.**

```python
def runNetwork():
    baseTime = time.time()
    nodes = [Node(i,baseTime,300) for i in range(6)]
    addresses = [node.socket.getsockname() for node in nodes]
    for node in nodes:
        node.start(addresses)
    timer = threading.Timer(10,deActiveNodeRandomly,(nodes,baseTime))
    timer.start()
    for node in nodes:
        node.selectThread.join()
        node.recvThread.join()
    for node in nodes:
        node.report()
```

شکل ۲۲: تابع runNetwork

```python
def deActiveNodeRandomly(nodes,baseTime,totalTime=300):
    node = None
    while node is None:
        node = random.choice(nodes)
        if node.state == 'deActive':
            node = None
    node.deActive()
    if time.time() - baseTime < totalTime:
        timer = threading.Timer(10,deActiveNodeRandomly,(nodes,baseTime))
        timer.start()
```

شکل ۲۳: تابع deActiveNodeRandomly

```python
def drawGraphs():
    options = {
        'node_color': 'pink',
        'node_size': 10000,
        'width': 2,
        'arrowstyle': '-|>',
        'arrowsize': 10,
        'font_size': 10,
        'edge_color':'green',
        'with_labels':True,
    }
    for i in range(6):
        graph = nx.DiGraph()
        json_inp = json.load(open('./json_output/report_node_'+str(i)+'.json'))
        vertices = [tuple(vertex) for vertex in json_inp['topology']['vertex']]
        edges = []
        for x in json_inp['topology']['edges']:
            temp = []
            for y in x:
                temp.append(tuple(y))
            edges.append(tuple(temp))
        plt.rcParams["figure.figsize"] = (20,10)
        graph.add_nodes_from(vertices)
        graph.add_edges_from(edges,weight=1)
        current_node_color = 'yellow' if json_inp['state'] == 'Active' else 'red'
        colors = ['pink' if x!=tuple(json_inp['address']) else current_node_color for x in graph.nodes()]
        options['node_color'] = colors
        pos = nx.spring_layout(graph,scale=5,k=5)
        nx.draw(graph,pos,**options)
        # plt.show()
        plt.savefig('./network_graphs/graph_node_'+str(i)+'.png',format="PNG")
        plt.clf()
```

شکل ۲۴: تابع drawGraphs

۵ نتایج شبیه‌سازی

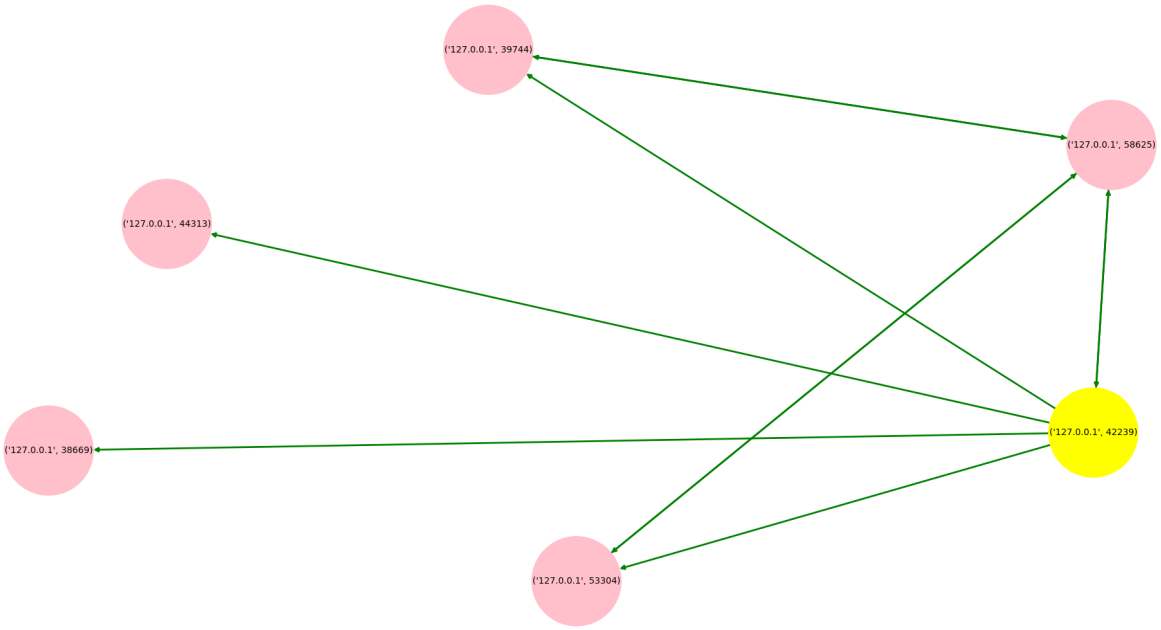
نتایج شبیه‌سازی در تصاویر زیر آورده شده است. در باره هرکدام نیز تحلیل مختصری بیان شده است. دقت کنید همه فایل‌های خروجی به فرمت json در پوشه json_output قرار گرفته است. همچنین گراف برای هر گره در پوشه network_graphs قرار گرفته است. قبل از ورود به بررسی گره‌ها ابتدا فرمول زیر را تعریف می‌کنیم که به طور تقریبی برقرار است.

$$availability(nodeX) \simeq \frac{send\,to\,count(nodeX) \times 2}{300} \quad (۱)$$

برای بررسی این که چرا رابطه فوق صحیح است، یک گره هرگاه گرهی در همسایگی اش باشد هر دو ثانیه یک پیام به گره همسایه خود ارسال می‌کند. بنابراین اگر گره x همسایه گره مورد نظر باشد، به طور تقریبی هر دو ثانیه یک پیام به او ارسال می‌گردد. از طرفی تا زمانی که گره در لیست همسایه‌ها باشد این رویداد برقرار است، پس به طور تقریبی مدت زمانی که گره مورد نظر با گره x همسایه است برابر با تعداد پیام‌های ارسالی به آن گره ضرب در ۲ است. در پایان با تقسیم این عدد بر کل زمان شبیه سازی که ۳۰۰ ثانیه است درصد دسترس پذیری گره x برای گره کنونی محاسبه می‌گردد.

۱.۵ گره صفر

فایل json مربوط به این گره دارای نام report_node_0 است. همچنین تصویر گراف مربوط به این گره در فایل graph_node_0 قرار دارد. این گراف را در تصویر ۲۵ مشاهده می‌کنید.



شکل ۲۵: گراف شبکه از دید گره صفر

خود گره با رنگ زرد مشخص شده است. در ادامه مطابق جدول داده شده بررسی می‌شود ایا نرخ دسترس پذیری معقول است یا خیر.

host number	send to count	calculated availability	predicted availability
2	14	0.0961	0.112
3	11	0.0856	0.088
4	8	0.0660	0.064

Table 1: comparison of predicted and calculated availability

همانگونه که مشاهده می‌کنید مقدار دسترس پذیری محاسبه شده و پیشبینی شده به صورت تقریبی برابر هستند. برای بررسی بیشتر نیز میتوان تعداد فرستاده شده و دریافت شده هر دو گره با یک دیگر را بررسی کرد که این عدد نیز باید نزدیک باشد البته چون مقدار پنج درصد از پیام‌ها drop می‌شوند پس امکان وجود تفاوت در مقدار ارسالی و دریافتی وجود دارد. جدول ۲ مقدایر ارسالی و دریافتی به هر گره، و مقدار ارسالی و دریافتی آن‌ها به گره درحال بررسی را نشان می‌دهد.

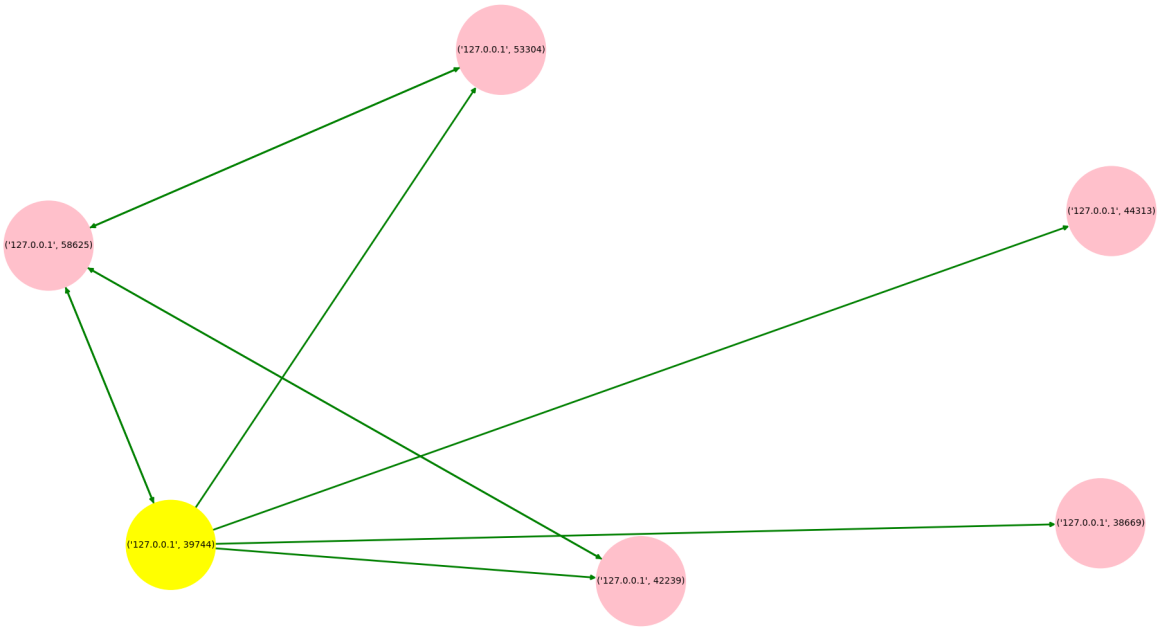
host	send node	receive node	receive current node	send current node
2	10	13	6	14
3	5	4	5	11
4	12	8	9	8

Table 2: statistics for node 0

همانگونه که مشاهده می‌کنید این اعداد با یک دیگر سازگار هستند، یعنی تعداد ارسال یکی از دیگری بزرگتر یا مساوی است و همچنین اختلاف زیادی بین اعداد نیست و اختلاف موجود نیز بخاطر اتلاف پیام‌ها در مسیر است. همچنان همسایه‌های کنونی این گره از تصویر ۲۵ قابل مشاهده است. گره کنونی یک همسایه دارد و در تلاش است با بقیه گره‌ها نیز همسایه شود. همچنین با استفاده از اطلاعات همسایه‌های همسایه خود متوجه شده است که یک گره که به آن درخواست داده، با همسایه او، همسایه است.

۲.۵ گره یک

فایل json مربوط به این گره دارای نام report_node_1 است. همچنین تصویر گراف مربوط به این گره در فایل graph_node_1 قرار دارد. این گراف را در تصویر ۲۶ مشاهده می‌کنید.



شکل ۲۶: گراف شبکه از دید گره یک

خود گره با رنگ زرد مشخص شده است. مشاهده می‌کنید که گره یک همسایه دو طرفه دارد و برای اینکه لیست همسایه‌های خود را کامل کنید، به سایر گره‌ها ارتباط یک طرفه برقرار کرده تا اگر جواب دادند همسایه دو طرفه شوند. همچنین با اطلاعات ارسال شده به عنوان همسایه از هر همسایه مشخص شده که همسایه خود با دوتا از گره‌هایی که به آن‌ها درخواست داده با هم همسایه هستند. در ادامه مطابق جدول داده شده بررسی می‌شود ایا نرخ دسترس پذیری معقول است یا خیر.

host	send to count	calculated availability	predicted availability
2	18	0.1258	0.12
3	26	0.1920	0.208
4	59	0.4664	0.472

Table 3: comparison of predicted and calculated availability

همانگونه که مشاهده می‌کنید مقدار دسترس پذیری محاسبه شده و پیشبینی شده به صورت تقریبی برابر هستند. برای بررسی بیشتر نیز میتوان تعداد فرستاده شده و دریافت شده هر دو گره با یک دیگر را بررسی کرد که این عدد نیز باید نزدیک باشد البته چون مقدار پنج درصد از پیام‌ها drop می‌شوند پس امکان وجود تفاوت در مقدار ارسالی و دریافتی وجود دارد. جدول ۴ مقدایر ارسالی و دریافتی به هر گره، و مقدار ارسالی و دریافتی آن‌ها به گره درحال بررسی را نشان می‌دهد.

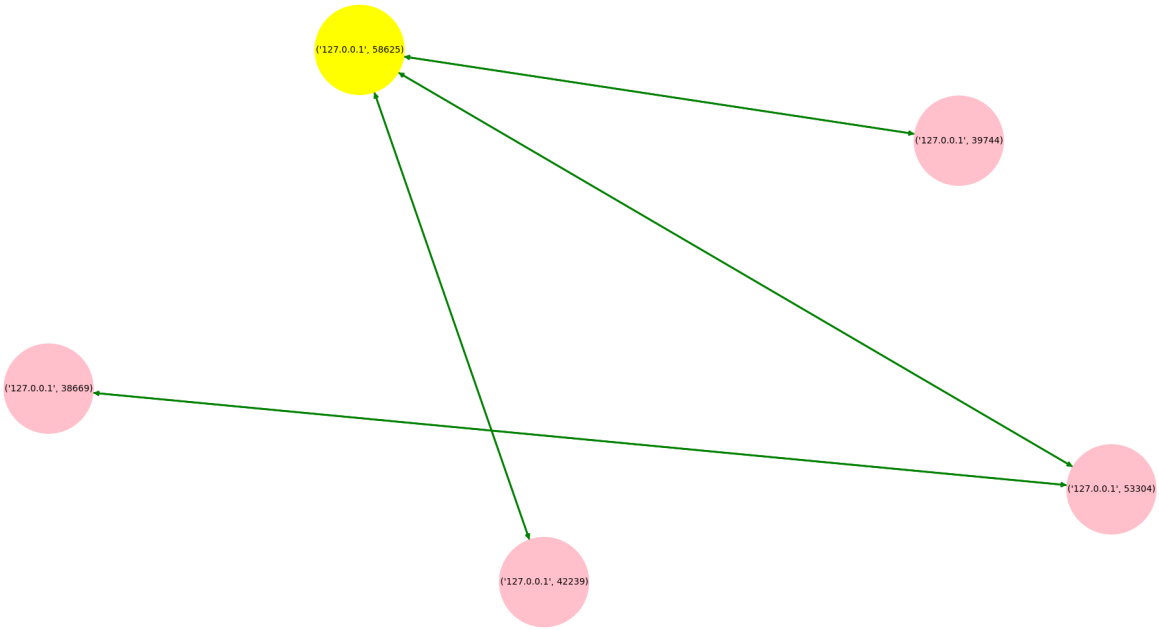
host	send node	receive node	receive current node	send current node
2	38	36	35	53
3	20	17	20	26
4	60	54	57	59

Table 4: statistics for node 0

همانگونه که مشاهده می‌کنید این اعداد با یک دیگر سازگار هستند، یعنی تعداد ارسال یکی از دیگری بزرگتر یا مساوی است و همچنین اختلاف زیادی بین اعداد نیست و اختلاف موجود نیز بخاطر اتلاف پیام‌ها در مسیر است. همچنین همسایه‌های کنونی این گره از تصویر ۲۶ قابل مشاهده است که یک همسایه دارد.

۳.۵ گره دو

فایل json مربوط به این گره دارای نام report__node__2 است. همچنین تصویر گراف مربوط به این گره در فایل graph__node__2 قرار دارد. این گراف را در تصویر ۲۷ مشاهده می‌کنید.



شکل ۲۷: گراف شبکه از دید گره یک

خود گره با رنگ زرد مشخص شده است. همچنین گره دارای ۳ همسایه دو طرفه است و با اوجه به اطلاعات یکی از همسایه‌ها، یک گره همسایه برای آن همسایه تشخیص داده است.

host	send to count	calculated availability	predicted availability
0	10	0.0712	0.08
1	38	0.2813	0.304
3	9	0.0646	0.072
4	19	0.1931	0.152
5	11	0.0592	0.088

Table 5: comparison of predicted and calculated availability

همانگونه که مشاهده می‌کنید مقدار دسترس پذیری محاسبه شده و پیشبینی شده به صورت تقریبی برابر هستند. برای بررسی بیشتر نیز میتوان تعداد فرستاده شده و دریافت شده هر دو گره با یک دیگر را بررسی کرد که این عدد نیز باید نزدیک باشد البته چون مقدار پنج درصد از پیام‌ها drop می‌شوند پس امکان وجود تفاوت در مقدار ارسالی و دریافتی وجود دارد. جدول ۶ مقادیر ارسالی و دریافتی به هر گره، و مقدار ارسالی و دریافتی آن‌ها به گره درحال بررسی را نشان می‌دهد.

همانگونه که مشاهده می‌کنید این اعداد با یک دیگر سازگار هستند، یعنی تعداد ارسال یکی از دیگری بزرگتر یا مساوی است و همچنین اختلاف زیادی

host	send node	receive node	receive current node	send current node
0	14	6	13	10
1	53	35	36	38
3	11	8	8	9
4	23	18	19	19
5	4	5	4	11

Table 6: statistics for node 0

بین اعداد نیست و اختلاف موجود نیز بخاطر اتلاف پیام‌ها در مسیر است. همچنین همسایه‌های کنونی این گره از تصویر ۲۷ قابل مشاهده است که سه همسایه دارد.

۴.۵ گره سه

فایل json مربوط به این گره دارای نام report_node_3 است. همچنین تصویر گراف مربوط به این گره در فایل graph_node_3 قرار دارد. این گراف را در تصویر ۲۸ مشاهده می‌کنید.

شکل ۲۸: گراف شبکه از دید گره یک

این گره چون خاموش است، گراف معادل آن هیچ گره دیگری را نمی‌شناسد(در حال حاضر) و خود نیز قرمز رنگ (به معنای خاموش) است. در ادامه مطابق جدول داده شده بررسی می‌شود ایا نرخ دسترس پذیری معقول است یا خیر.

host	send to count	calculated availability	predicted availability
0	5	0.0572	0.04
1	20	0.2379	0.16
2	11	0.0921	0.088
5	4	0.0274	0.032

Table 7: comparison of predicted and calculated availability

همانگونه که مشاهده می‌کنید مقدار دسترس پذیری محاسبه شده و پیشبینی شده به صورت تقریبی برابر هستند. برای بررسی بیشتر نیز میتوان تعداد فرستاده شده و دریافت شده هر دو گره با یک دیگر را بررسی کرد که این عدد نیز باید نزدیک باشد البته چون مقدار پنج درصد از پیام‌ها drop می‌شوند پس امکان وجود تفاوت در مقدار ارسالی و دریافتی وجود دارد. جدول ۸ مقدایر ارسالی و دریافتی به هر گره، و مقدار ارسالی و دریافتی آن‌ها به گره درحال بررسی را نشان می‌دهد.

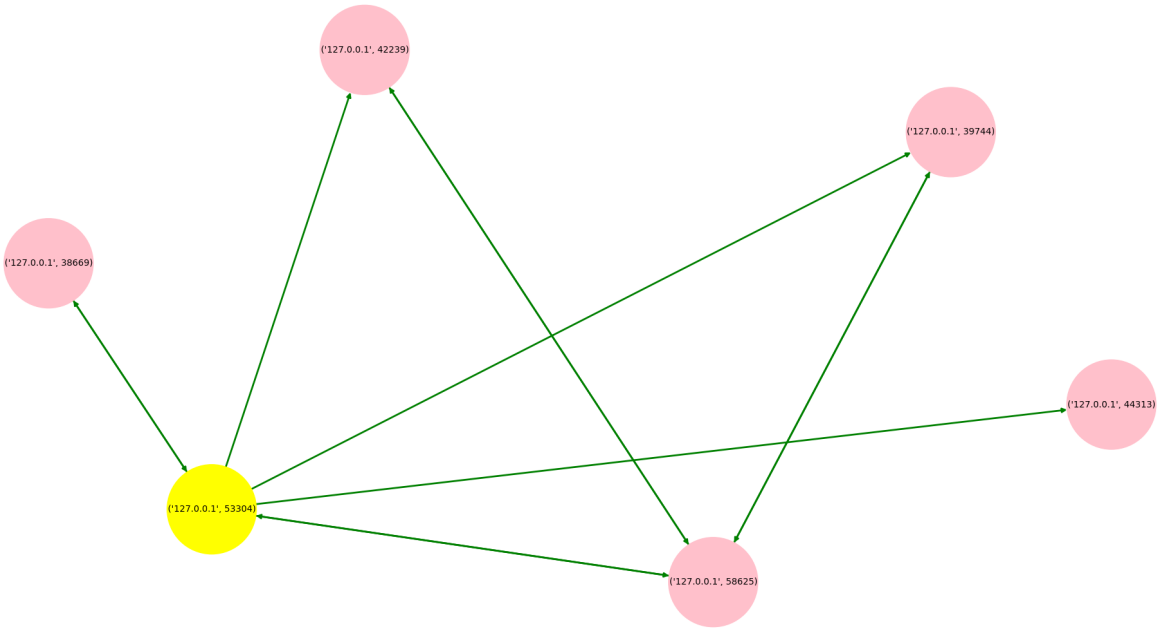
همانگونه که مشاهده می‌کنید این اعداد با یک دیگر سازگار هستند، یعنی تعداد ارسال یکی از دیگری بزرگتر یا مساوی است و همچنین اختلاف زیادی بین اعداد نیست و اختلاف موجود نیز بخاطر اتلاف پیام‌ها در مسیر است.

host	send node	receive node	receive current node	send current node
0	11	5	4	5
1	26	20	17	20
2	9	8	8	11
5	5	1	1	4

Table 8: statistics for node 0

۵.۵ گره چهار

فایل json مربوط به این گره دارای نام report__node__4 است. همچنین تصویر گراف مربوط به این گره در فایل graph__node__4 قرار دارد. این گراف را در تصویر ۲۹ مشاهده می‌کنید.



شکل ۲۹: گراف شبکه از دید گره یک

خود گره فعلی با رنگ زرد مشخص شده است.همانطور که قابل مشاهده است، این گره دو همسایه دو طرفه دارد و در تلاش است تا همسایه سوم را نیز بیابد بنابراین به گره‌های دیگر ارتباط یک طرفه برقرار کرده است. همچنین با کمک داده‌های مربوط به همسایه‌ی همسایه‌های خود، دانسته که چه گره‌هایی با کدام همسایه‌های گره اصلی، همسایه هستند. در ادامه مطابق جدول داده شده بررسی می‌شود ایا نرخ دسترس پذیری معقول است یا خیر.

host	send to count	calculated availability	predicted availability
0	12	0.0897	0.0959
1	60	0.4759	0.48
2	23	0.1670	0.184
5	13	0.0993	0.104

Table 9: comparison of predicted and calculated availability

همانگونه که مشاهده می‌کنید مقدار دسترس پذیری محاسبه شده و پیشبینی شده به صورت تقریبی برابر هستند. برای بررسی بیشتر نیز میتوان تعداد فرستاده شده و دریافت شده هر دو گره با یک دیگر را بررسی کرد که این عدد نیز باید نزدیک باشد البته چون مقدار پنج درصد از پیام‌ها drop می‌شوند پس امکان وجود تفاوت در مقدار ارسالی و دریافتی وجود دارد. جدول ۱۰ مقدایر ارسالی و دریافتی به هر گره، و مقدار ارسالی و دریافتی آن‌ها به گره درحال بررسی را نشان می‌دهد.

همانگونه که مشاهده می‌کنید این اعداد با یک دیگر سازگار هستند، یعنی تعداد ارسال یکی از دیگری بزرگتر یا مساوی است و همچنین اختلاف زیادی بین اعداد نیست و اختلاف موجود نیز بخاطر اتلاف پیام‌ها در مسیر است.

host	send node	receive node	receive current node	send current node
0	8	9	8	12
1	59	57	54	60
2	19	19	18	23
5	16	13	12	13

Table 10: statistics for node 0

۶.۵ گره پنج

فایل json مربوط به این گره دارای نام report_node_5 است. همچنین تصویر گراف مربوط به این گره در فایل graph_node_5 قرار دارد. این گراف را در تصویر ۳۰ مشاهده می‌کنید.

شکل ۳۰: گراف شبکه از دید گره یک

این گره چون خاموش است، گراف معادل آن هیچ گره دیگری را نمی‌شناسد(در حال حاضر) و خود نیز قرمز رنگ (به معنای خاموش) است. در ادامه مطابق جدول داده شده بررسی می‌شود ایا نرخ دسترس پذیری معقول است یا خیر.

host	send to count	calculated availability	predicted availability
1	4	0.0299	0.032
2	4	0.0324	0.032
3	5	0.0296	0.04
4	16	0.1290	0.128

Table 11: comparison of predicted and calculated availability

همانگونه که مشاهده می‌کنید مقدار دسترس پذیری محاسبه شده و پیشبینی شده به صورت تقریبی برابر هستند. برای بررسی بیشتر نیز میتوان تعداد فرستاده شده و دریافت شده هر دو گره با یک دیگر را بررسی کرد که این عدد نیز باید نزدیک باشد البته چون مقدار پنج درصد از پیام‌ها drop می‌شوند پس امکان وجود تفاوت در مقدار ارسالی و دریافتی وجود دارد. جدول ۱۲ مقدایر ارسالی و دریافتی به هر گره، و مقدار ارسالی و دریافتی آن‌ها به گره درحال بررسی را نشان می‌دهد.

همانگونه که مشاهده می‌کنید این اعداد با یک دیگر سازگار هستند، یعنی تعداد ارسال یکی از دیگری بزرگتر یا مساوی است و همچنین اختلاف زیادی بین اعداد نیست و اختلاف موجود نیز بخاطر اتلاف پیام‌ها در مسیر است. **در این‌جا یک نکته مهم قابل بیان است. دقت کنید گره ۵ گره ۱ را همسایه خود می‌داند و تعداد ۴ پیام ارسال کرده است، اما چون گره ۱ یا پیام‌هارا دریافت نکرده یا هنوز بررسی نکرده است، گره ۱ گره ۵ را همسایه خود نمی‌داند و فقط همسایه تلاش شده خود می‌داند زیرا به او پیام برای همسایه شدن ارسال کرده اما پاسخ ارسالی یا دریافت نشده یا بررسی نشده است. علت این امر آن است که زمان شبیه سازی محدود بوده و پیش از بررسی پاسخ گره ۵ به گره ۱ توسط گره ۱ شبیه‌سازی پایان یافته است. از طرفی علت دیگر نیز**

host	send node	receive node	receive current node	send current node
1	-	-	0	4
2	11	4	5	4
3	4	1	1	5
4	13	12	13	16

Table 12: statistics for node 0

می‌تواند خاموشی گره ۵ باشد. اگر شبیه سازی برای مدتی طولانی انجام شود، شبکه کاملا همگرا شده و این خطاها روی نخواهد داد.