



آذر ۱۳۹۷

CE-40695

## پیاده سازی یک شبکه P2P

استاد: دکتر مهدی جعفری

### ۱ چکیده

هدف این فاز از پروژه پیاده سازی یک شبکه peer to peer است که در فاز بعدی قرار است یک block chain application روی آن اجرا شود. در این شبکه:

□ یک root در نقش DNS Server وجود دارد.

□ تعدادی پیام جهت ارتباط node ها با یکدیگر و تشکیل یک گراف همبند بدون دور از آن ها (درخت) رد و بدل میشود.

□ ارتباط کاربر با شبکه از طریق یک واسط کاربری انجام میگردد.

جهت تست این شبکه، باید قابلیت اجرا شدن هم در نقش (DNS server) root و هم در نقش client را داشته باشید و در نقش و نحوه رد و بدل شدن پیامها و عملیات مختلف انجام شده در شبکه را مشاهده نمایید.

### ۲ مقدمه

P2P ساده شده عبارت Peer to Peer یا معادل فارسی آن همتا به همتا توزیعی از معماری شبکه های کامپیوتری است که در آن peer ها هم در نقش client و هم در نقش server ظاهر میشوند و به peer های دیگر میتوانند اطلاعات بفرستد یا از آنها اطلاعات بگیرند. در این شبکه ها برخلاف شبکه های client-server سرور مرکزی وجود ندارد تا باعث شود کلاینت ها برای دسترسی به منابع سرور از آن اطاعت کنند. Bitcoin, Tor, Torrent ... نمونه هایی از شبکه های P2P شناخته شده هستند. عملاً یک شبکه ی P2P نمی تواند وجود داشته باشد. و معمولاً یک سری node هستند که دارند maintain می کنند و اگر آنها نباشند به شبکه آسیب می رسد

### ۳ پروتکل پروژه

#### ۱.۳ نقش کلاینت ها

اولین قدم رجیستر شدن هر کلاینت در شبکه است. برای این کار کلاینت با فرستادن یک پیام رجیستر به root، خودش را رجیستر میکند. برای این کار کلاینت با فرستادن یک پیام رجیستر به روت خودش را در شبکه رجیستر میکند.

در ادامه هر کلاینت باید آدرس اولین همسایه خود را از DNS بگیرد. برای این کار کلاینت با فرستادن یک پیام در قالب advertise به روت شبکه درخواست یک آدرس میکند که در جواب DNS هم یک آدرس برمیگرداند. زمانی که از یک کلاینت که قبلاً advertise شده پیام advertise می آید، روت در جواب به آن، آدرس هیچکدام از همسایه های client را نمی فرستد و به جای آن آدرس جدید میفرستد.

کلاینت بعد از پیدا کردن اولین همسایه خود با فرستادن یک join packet به اطلاع آن میدهد که درخواست متصل شدن دارد.

در این شبکه همه موظف هستند که پیام های دریافتی را از طریق کانکشن ها به همه ارسال کنند.

بعد از متصل شدن به اولین همسایه، کلاینت میتواند پیام های خود را به صورت broadcast به شبکه ارسال کند.

بعد از اتصال به همسایه هر node موظف است هر ۴ ثانیه یک بار از طریق همسایه خود به روت یک پیام reunion hello ارسال کند.

روت انتظار دارد این پیام هر ۴ ثانیه یک بار از هر کلاینت به او برسد در غیر این صورت فرض میکند این کلاینت در شبکه وجود ندارد و

دیگر آن را advertise نمیکند.  
 هر کلاینتی که پیام reunion hello به دستش میرسد باید آدرس خود را به ته پیام بچسباند، field number of entities را آپدیت کند و به پدرش ارسال کند. و اگر هم reunion hello back رسید، آدرس خودش را از پیام بر میدارد، field number of entities را آپدیت میکند و به نفر بعدی میدهد.  
 لازم به ذکر است که در این شبکه عمق درخت حداکثر ۸ فرض شده است.

### ۲.۳ نقش روت

اولین وظیفه روت رجیستر کردن client ها در شبکه با جواب دادن به register request آن ها است.  
 در گام بعدی روت وظیفه دارد جواب advertise request کلاینت ها را با دادن آدرس یک peer، با فرض اینکه در گراف اتصالات تشکیل شده هیچ peer ای وجود ندارد که بیشتر از ۲ فرزند داشته باشد، به آن ها دهد.  
 آخرین وظیفه روت انتظار برای رسیدن یک reunion hello به ازای هر client در مدت زمان ۳۶ ثانیه و برگرداندن یک پیام reunion hello back به آنهاست. اگر reunion hello در این مدت نرسید، روت دیگر آن node و فرزندانش را advertise نمیکند.

### ۴ توضیحات پیاده سازی

در ادامه به معرفی وظایف اشیاء ساخته شده از هر کلاس و تابعهای موجود در آنها و همچنین نحوه گرفتن ورودی در آنها میپردازیم.

#### ۱.۴ Peer

Stream موجود در constructor این کلاس، یک server\_ip و یک server\_port را ورودی میگیرد که ip و port همان سرور خودمان هستند. مقدار parent چنانچه ما root باشیم none است و در غیر این صورت هنگامی که درخواست join را ارسال میکنیم مشخص میشود. Packets هم همان بسته هایی است که دریافت کرده و باید به آن ها رسیدگی کنیم. Neighbours همان بچه های ما هستند. هر peer یک UserInterface و یک PacketFactory هم دارد. اگر root باشیم، به network\_nodes نیاز داریم که تمامی node های این شبکه را در بر بگیرد؛ همچنین به registered\_nodes جهت ثبت node هایی که به واسطه ما در شبکه register کرده اند احتیاج داریم. اگر هم root باشیم، root را مشخص کرده و به stream هم root را اضافه میکنیم (یعنی یک register\_node در root اضافه میکنیم). پیاده سازی تابع start\_user\_interface به این صورت است که تنها user\_interface را اجرا میکند. پس از آن، برنامه منتظر دستور کاربر میماند. دستورات کاربر به فرمت زیر هستند:

new\_register\_packet :1 □

new\_advertise\_packet :2 □

send\_broadcast\_packet :3 □

که بر اساس عدد وارد شده توسط کاربر، عملیات نظیر شده به آن در تابع handle\_user\_interface\_buffer اجرا میشود. در تابع read\_in\_buf، stream را میخوانیم، بسته ها را میسازیم و به آنها رسیدگی میکنیم و در همان لحظه این بافر را پاک میکنیم تا به یک پیام دو بار رسیدگی نکنیم. پس از آن به user interface رسیدگی میکنیم و هر بسته ای که احتیاج داریم بفرستیم را میفرستیم و برای یک دوره زمانی، استراحت (sleep!) میکنیم. برای فرستادن یک broadcast packet، از تابع send\_broadcast\_packet استفاده میکنیم. تابعی به اسم handle\_packet وجود دارد که برای بررسی بستهها مورد استفاده قرار میگیرد. به عنوان نمونه، طول هر بسته در آن چک می‌شود. در تابع check\_registered یک آدرس ورودی میگیریم و چک میکنیم که آن node در بین registered\_node ها وجود دارد یا خیر. در تابع handle\_advertise\_packet اول چک میکنیم که نوع بسته دریافت شده چیست؛ اگر از نوع request باشد، باید حتما root باشیم تا بتوانیم آن را بررسی کنیم (در غیر این صورت، آن را دور می ریزیم). در این صورت، اگر register شده بود، از یکی از همسایه های آن را پیدا میکنیم تا به آن advertise request بفرستیم.

#### ۲.۴ Stream

در تابع constructor این کلاس ابتدا با متد is\_valid فرمت های IP و Port را چک میکنیم تا مطمئن شویم به صورت همان فرمت مورد نظر هستند. server\_in\_buf همان بافری هست که روی سرور نوشته میشود و هر چند وقت یک بار بایستی چک شود. Callback Function (cb) نیز پیام های جدید را به ته server\_in\_buf میچسباند (append) و در نهایت Ack برمیگرداند. این Ack باعث میشود هر جا که سوکتی وسط کار قطع شود بفهمد قطع شده است. سپس tcpserver را میسازیم، در یک thread قرار میدهم و آن را اجرا میکنیم. self.nodes تمام نود هایی هستند که درون ما هستند.  
 get\_server\_address آدرس سرور را با آن فرمتی که میخواهیم به ما میدهد.  
 clear\_in\_buf بافر سرور را پاک میکند.  
 add\_node نود اضافه میکند.

`remove_node` نود مشخص شده را از آرایه پاک میکند و سپس متد `close` نود را اجاره میکند.  
`get_node_by_server` آی پی و پورت سرور یک نود را میگیرد و نود را برمیگرداند. سپس با `parse` کردن آن را به فرمت مد نظر تبدیل میکند.  
`add_message_to_out_buffer` با گرفتن یک آدرس و پیام نود را پیدا میکند و در `out_buffer` مینویسد.  
`read_in_buf` وظیفه دارد `read_in_buf` را برگرداند.  
`send_message_to_node` بافرهای توی نود را با استفاده از کال کردن تابع `send_message` خودش ارسال میکند.  
`send_out_buf_messages` پیام تمامی نود ها را ارسال میکند.

## ۳.۴ Node

در `constructor` این آبجکت ابتدا `IP/Port` سرور با `parse` شدن به فرمت مورد نظر در می آیند. `out_buff` بافری هست که قراره روی کلاینتش بنویسیم برود. با `is_register_connection` چک میکنیم رجیستر هست یا خیر. در آخر یک `try/catch` برای سوکت کلاینت قرار میدهیم تا اگر نودی در آن وسط `deattach` شد `exception` بخورد و از `out_buffer` پاک مشود.  
`send_message` به ازای هر بافر یک `self.client.send` میکند و اگر `Ack` برگشت یعنی پیام ارسال شده است.