



پیاده سازی یک شبکه P2P

استاد: دکتر مهدی جعفری

۱ مقدمه

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 یک دفعه آن را به عنوان server و n دفعه به عنوان client آن را اجرا کنید و نحوه رد و بدل شدن پیامها و عملیات مختلف انجام شده در شبکه را مشاهده نمایید. ساختار کلی کلاسهای این کد به صورت زیر است:

۲ اشیا

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

۱.۲ 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. 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 احتیاج داریم که تمامی node های این شبکه را در بر میگیرد؛ همچنین به registered_nodes جهت ثبت node هایی که به واسطه ما در شبکه register کرده اند احتیاج داریم. اگر هم root نباشیم، 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:

□ 1: new_register_packet

□ 2: new_advertise_packet

□ 3: send_broadcast_packet

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_bufun 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 حتماً `root` باشیم تا بتوانیم آن را بررسی کنیم (در غیر این صورت، آن را دور می‌ریزیم). در این صورت، اگر `register` شده بود، از یکی از همسایه‌های آن را پیدا میکنیم تا به آن `request` `advertise` بفرستیم.

۲.۲ Stream

در تابع `constructor` این کلاس ابتدا با متد `is_valid` فرمت‌های `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` `(cb)` نیز پیام‌های جدید را به `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 مسازیم، در یک `thread` قرار میدهیم و آن را اجرا میکنیم. `self.nodes` تمام نودهایی 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 نودها را ارسال میکند.

۲.۲ Node

در `constructor` این آبجکت ابتدا `IP/Port` سرور با `parse` شدن به فرمت مورد نظر در می‌آیند. `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 نشود. `send_message` به ازای هر بافر یک `self.client.send` میکند و اگر `Ack` برگشت یعنی پیام ارسال شده است.