Sistema Descentralizado de Chat

PSD & SDGE

Daniel Pereira	Duarte Ribeiro	Filipe Pereira	Rui Lopes
PG55928	PG55938	PG55941	PG56009

Índice

Introdução

Este relatório tem como objetivo apresentar o trabalho prático desenvolvido durante as unidades curriculares de Paradigmas de Sistemas Distribuídos e Sistemas Distribuídos em Grande Escala. O documento detalha o processo de implementação, contando com a descrição de todos os componentes e estratégias utilizadas para superar os diversos desafios.

Cliente

Este componente define a interface de utilização do serviço, sendo que a lógica de comunicação com os demais componentes deverá estar completamente abstraída ao utilizador final. Como tal, o grupo decidiu implementar uma TUI[1] baseada em comandos.

```
[INFO] Welcome to Doge Chat!
[INFO] Running client with id: rui
...
[INFO] Available commands:
[INFO]  - /exit
[INFO]  - /help
[INFO]  - /send <message>
[INFO]  - /online
[INFO]  - /topic <topic>
[INFO]  - /logs <last> <userId?>
[INFO]  - /cancel
client>
```

Listagem 1: `mvn -pl client exec:java -Dexec.args="--name rui -t uminho -as 6667"`

Para simplificação foram removidos todos os logs de output (em Listagem 1) relativos à pesquisa do tópico, consequente agregação distribuída e interação inicial com o servidor de chat escolhido pelo cliente. Esse processo será agora explicado.

Ciclo de Vida

Quando o cliente inicia recebe como parâmetro o seu identificador (`--name`) e o tópico a que se quer juntar (`-t` ou `--topic`) - ou criar caso este não exista.

Primeiramente, este contacta o serviço de pesquisa (via DHT), visto que o tópico pode já ter sido criado por outro cliente. Caso o tópico exista, o serviço de pesquisa retorna o endereço de `C`[2] servidores de chat e, aleatoriamente, o cliente escolhe um para se conectar. Caso o tópico ainda não exista, deverá ser

[1] Terminal User Interface.
[2] Valor configurável, mas atualmente definido para 2.

criado e, portanto, o cliente contacta um servidor de agregação que iniciará um processo de agregação distribuída para descobrir quais os `C` servidores mais capazes de atender um novo tópico.

Após ter na sua posse o servidor de chat a que se deverá ligar, o cliente trata de estabelecer diversas conexões com o mesmo e anunciar-se perante a rede. Tal como iremos ver mais em baixo, um servidor de chat possui diversos sockets, seja para comunicar com outros servidores de chat, com clientes ou com o seu servidor de agregação.

Assim sendo, o cliente possuirá as seguintes conexões abertas com os restantes componentes do sistema:

Servidor de Chat

- Socket `PUSH` para envio de mensagens de chat ou de saída anunciada;
- Socket `REQ` para envio de pedidos síncronos, como listar utilizadores online no tópico (via `/online`) ou de aviso de entrada;
- Socket `SUB` para subscrever a mensagens do tópico atual no seu servidor de chat[3]
- Socket `gRPC` para realização de pedidos de logs via Reactive gRPC.

Servidor de Agregação

- Socket `REQ` para solicitar início de agregação ao servidor de agregação passado como argumento aquando da inicialização do programa (via `-as`).

Algo a ter em atenção é o facto de que os sockets ZeroMQ não são thread-safe, pelo facto de que não implementam mecanismos de exclusão, deixando isso para a camada aplicacional. No componente do cliente essa preocupação não existe[4].

Servidor de Pesquisa

- Socket `TCP` para comunicação com o nó da DHT passado como argumento aquando da inicialização do programa (via `-dht`)[5].

O grupo decidiu fazer uso de Protobuf para além do imposto pela utilização do `gRPC`. Isto é, para a comunicação via sockets ZeroMQ (seja aqui no cliente, seja no resto do sistema) fez-se uso de mensagens definidas e geradas via Protobuf[6]. Esta escolha permitiu que pudéssemos avançar mais rapidamente noutras vertentes do sistema, sem grande preocupação sobre o protocolo de mensagens e respetiva serialização.

Por outro lado, a comunicação com o serviço de pesquisa é feita através de um protocolo muito simples suportado por JSON, que será visto mais em baixo. Dada esta heterogeneidade de middlewares (tanto ao nível do socket como do codec a adotar), o grupo decidiu criar uma abstração em Java, `Endpoint<M>`, capaz de receber um meio de transporte (subclasse de `AbstractTransport`) e um codec (subclasse de `MessageCodec<M>`). Esta classe tem a capacidade de registar handlers que serão executados aquando da receção assíncrona de mensagens na camada de transporte definida, caso o socket tenha a operação de `receive` implementada.

Desta forma, a criação de um socket ZeroMQ do tipo `PUSH` capaz de enviar mensagens geradas pelo Protobuf prima pela simplicidade de se instanciar uma classe da seguinte forma:

```
var pushSocket = new Endpoint<MessageWrapper>(
  new ZmqPushTransport(context.createSocket(SocketType.PUSH)),
  new ProtobufCodec()
)
```

Esta classe exporá, depois, métodos de `connectSocket` ou `disconnectSocket`.

[3] Isto pois, um servidor de chat poderá estar a servir vários tópicos.
[4] Mais tarde irão aparecer componentes onde essa preocupação existe.
[5] Definido para `127.0.0.1:8000` por omissão.
[6] Constam no mesmo ficheiro `.proto` que o serviço `gRPC`.

Caso o socket fosse do tipo `PULL` seria possível registar handlers de receção via um método `on`.

Ainda, no caso de ser um socket normal `TCP` capaz de enviar e receber mensagens via JSON, bastaria:

```
var dhtClient = new Endpoint<JsonNode>(
  new TcpTransport(new Socket(host, port)),
  new JsonCodec()
)
```

Em suma, esta abstração, aliada ao facto de que utilizamos Protobuf na maioria do sistema, permitiu, factualmente, ao grupo iterar mais rápido sobre as questões propostas pelo enunciado.

Threads Virtuais

Ao longo do desenvolvimento, quando possível, o grupo fez uso de threads virtuais do Java[7], seja no componente do Cliente, seja nos restantes. Isto é, todas as chamadas a `Thread` foram substituídas por `Thread.ofVirtual`. Ainda, durante o desenvolvimento com Reactive gRPC fez-se uso desta "fibers", tal como iremos detalhar no seguinte subcapítulo.

Este tipo de threads, se comparadas com as convencionais, têm inúmeras vantagens. Por exemplo, o "scheduling" é efetuado pelo próprio runtime da JVM e não do sistema operativo; além de gastarem muito menos memória.

Receção de Logs

A receção de logs, tal como pedido pelo enunciado, recorre à tecnologia Reactive gRPC. Essencialmente, quando o utilizador executa o comando `/logs` (com os respetivos argumentos) é feito um RPC `GetLogs` que irá retornar uma stream de `ChatMessage` - em termos reativos, isto quer dizer que tanto o cliente como o servidor de chat irão estar assentes sobre `Flowable`.

Uma vez que o cliente pode pedir uma quantidade exacerbante de logs, seria desagradável se a UI bloqueasse até que o RPC concluísse. Desta forma, optamos por utilizar a primitiva `subscribeOn`, visto que estaremos a receber dados vindos de um socket - assim, a própria subscrição é feita numa thread diferente da principal e permite que o cliente possa, logo de seguida, escrever outros comandos. Para além disso, utilizamos a primitiva `observeOn`, com um valor de backpressure igual a `32` - isto é, a própria receção dos logs também não atrapalhará a thread principal e, portanto, o cliente poderá visualizar mensagens ou escrever outros comandos enquanto o pedido atual de logs se encontra a ser feito.

Em ambos as chamadas, a `subscribeOn` e `observeOn`, fez-se uso de "fibers" instanciando um `Scheduler` que é passado como argumento. Este "scheduler" fará, depois, uso de um `ExecutorService` capaz de lançar threads virtuais para não interromper a execução normal do programa, tal como já mencionado.

Posto isto, não é possível a um cliente iniciar dois RPCs ao mesmo tempo, mas é lhe possível cancelar o "ongoing" RPC (via `/cancel`) e começar outro.

Finalmente, mais tarde iremos ver outro tipo de preocupações que foram necessárias aquando da implementação do serviço no lado do servidor de chat.

Troca de Servidor de Chat

Quando o servidor de chat a que um cliente está conectado morre (ou fica incontactável, por algum motivo) este deverá conseguir trocar para outro servidor de chat que atende o mesmo tópico.

Obviamente, caso não existam mais servidores de chat a suportar o tópico o cliente não consegue prosseguir. De facto, a migração de tópicos entre servidores de chat é um processo que envolve algumas questões complexas - logicamente, ficou de fora.

[7] Disponíveis a partir da versão 21.

Esta funcionalidade de troca baseia-se na criação de um `ZMonitor`[8] que fica "attached" ao socket do tipo `REQ`, um dos sockets que liga um cliente a um servidor. Esse monitor fica configurado para enviar "heartbeats" de X em X tempo e detetar, então, quando a outra ponta (servidor de chat) deixa de responder.

Servidor de Chat

Este componente é responsável pelo correto roteamento de mensagens entre diferentes servidores de chat e, por conseguinte, entre clientes interessados num dado tópico. Ainda, deverá oferecer um serviço de transmissão de logs (como visto no capítulo anterior) e persistir os utilizadores online naquele momento.

Entrega Causal de Mensagens

Um dos requisitos do projeto é que seja garantida a entrega causal de mensagens. Num serviço de chat isto faz especialmente sentido, uma vez que uma mensagem A que seja enviada depois de uma mensagem B (possivelmente uma reposta) nunca aparecerá antes de A.

No entanto, não estamos perante um algoritmo que pretende, à partida, garantir uma ordem total e, portanto, quando duas mensagens forem concorrentes terá de existir um critério arbitrário de desempate. Este critério é ativado apenas quando as mensagens são efetivadas no log do servidor de chat e, como tal, será de esperar que mensagens concorrentes apareçam, no imediato, numa ordem diferente em diferentes clientes, mas sempre com a garantia de que o log final inevitavelmente ficará exatamente igual para todos os servidores de chat e clientes.

De facto, se olharmos para serviços de chat convencionais, tais como: WhatsApp ou Discord, é fácil encontrar cenários em que as mensagens enviadas, no imediato, aparecem em ordens diferentes para diferentes utilizadores (devido a serem concorrentes), mas, tal como no nosso sistema, o log final acaba por convergir para um mesmo valor[9].

Assim sendo, basta criar um "vector clock" por tópico e utilizar um algoritmo de "causal delivery", que realiza "buffering" das mensagens recebidas até às mesmas estarem prontas para serem entregues, tal como estudado nas aulas teóricas.

Naturalmente, um "vector clock" possui desvantagens. Por exemplo, não é fácil lidar com a saída de nós (neste caso, servidores de chat) - isto pois não sabemos exatamente quando é que deixarão de surgir mensagens que possam estar causalmente dependentes do servidor que morreu. Uma alternativa seria usar "interval tree clocks", mas, por simplicidade, o grupo decidiu não avançar com a ideia.

Gestão de Utilizadores Online

A entrada e saída de utilizadores deverá ser propagada entre servidores de chat, mas com a garantia de que o estado de utilizadores online convirja para o mesmo valor em todos os nós. É de esperar que um CRDT resolva o problema.

Desta forma, o grupo decidiu fazer uso de um CRDT do tipo `ORSet` em modo state-based. A escolha de um CRDT deste tipo prende-se no facto de que não poderão existir dois utilizadores com o mesmo identificador e, portanto, um set adequa-se.

Ainda, a implementação do mesmo em modo state-based dá-nos garantias que não teríamos numa implementação operation-based. Ora, um CRDT operation-based não se irá comportar bem perante uma rede em que podem existir mensagens duplicadas, mensagens perdidas ou partições de rede. No caso de um CRDT em modo state-based estes fatores, ao nível da rede, não afetam o seu funcionamento, nem é necessária a utilização de um sistema de entrega causal. Como tal, esta natureza mais robusta do CRDT inclinou o grupo para o modo state-based.

[8] Classe do próprio ZeroMQ em Java, `jeromq`.

[9] Isso pode ser comprovado se sairmos e voltarmos a entrar numa sala de chat, no caso do Discord, por exemplo. O que envolve um "refetch" do log.

Serviço `gRPC` de Logs

Este é o serviço responsável pela emissão de logs quando pedidos de clientes surgem. O serviço está assente na tecnologia Reactive gRPC e, portanto, baseia-se no paradigma reativo.

```
message LogRequestMessage {
  string topic = 1;
  int32 last = 2;
}

message UserLogRequestMessage {
  string topic = 1;
  int32 last = 2;
  string userId = 3;
}

service LogService {
  rpc GetLogs (LogRequestMessage) returns (stream ChatMessage);
  rpc GetUserLogs (UserLogRequestMessage) returns (stream ChatMessage);
}
```

Listagem 2: Definição do serviço `gRPC`

A Listagem 2 mostra-nos a definição do serviço `gRPC` ao nível do Protobuf. Essencialmente, existem dois RPCs disponíveis: um para solicitar os logs (totais ou parciais) do tópico atual do cliente e outro para solicitar os logs (também totais ou parciais) do tópico atual do cliente, mas para um utilizador específico.

Durante a implementação do serviço, o grupo teve em atenção dois pormenores.

Primeiramente, o facto de que podem estar a surgir mensagens o log enquanto o utilizador está a consumir - levando a um erro de `ConcurrentModificationException`. Assim, quando o cliente realiza um RPC e está pronto a consumir é feito um "snapshot" do estado dos logs e essa é a "stream" que é enviada para o cliente.

Depois, é importante perceber que no ciclo de vida de um consumidor (neste caso, o cliente) é possível que o consumo só comece a acontecer passado algum tempo da invocação do RPC. Desta forma, a chamada ao método `Flowable.fromIterable`, que transforma a lista normal de Java numa stream reativa, é encapsulada numa chamada a `Flowable.defer` que atrasa a criação do "snapshot" até ao cliente começar a consumir. Ou seja, garantimos que o cliente vê os dados com a maior frescura possível, ainda que limitada à restrição imposta pelo ponto anterior.

Em relação ao primeiro ponto, uma possível solução seria a utilização de uma "cold stream" que emite eventos de adição quando um log é adicionado. No entanto, a stream criada no RPC teria, também, ela de ser "cold", isto pois os logs são enviados a pedido. No entanto, uma vez que não controlamos a forma como funciona o Reactive gRPC, o grupo viu-se impossibilitado de aplicar esta técnica.

Comunicação com Outros Componentes

Aqui, descrevemos todos os meios de comunicação utilizados para as diversas comunicações, com o exterior, por um servidor de chat.

Cliente

- Socket `PULL` para receber mensagens vindas dos clientes;
- Socket `PUB` para publicar mensagens vindas de outros servidores de chat nos clientes interessados;
- Socket `gRPC` para disponibilização do serviço de logs feito em Reactive gRPC.

Servidor de Agregação

- Socket `REP` * para responder a pedidos síncronos por parte de clientes ou do seu servidor de agregação.

Servidor de Chat

- Socket `SUB` com o objetivo de subscrever a mensagens vindas de outros servidores de chat;
- Socket `PUB` com o objetivo de publicar mensagens vindas dos clientes (via `PULL`) em outros servidores de chat.

O socket `REP` mencionado é utilizado tanto por clientes como pelo servidor de agregação do servidor de chat em questão. Uma vez que um servidor de chat pode ter múltiplos clientes a querer realizar pedidos síncronos (por exemplo, via comando `/online`) ficamos bastante limitados a um simples padrão de pedido-resposta sem concorrência.

Assim sendo, o grupo optou por implementar uma extensão ao padrão de pedido-reposta, tal como estudado nas aulas teóricas. Nesse sentido, existirá um socket `ROUTER` capaz de comunicar com um socket `DEALER` (via "proxy" do ZeroMQ), sendo que este último irá tratar de distribuir os pedidos por diversos sockets `REP` à escuta no endereço "bound" pelo socket `DEALER`, que no caso do nosso sistema é do tipo `inproc`. Atualmente, o número de "workers" do tipo `REP` está definido para o total de processadores existente na máquina que está a executar o servidor de chat.

Finalmente, ao contrário do componente Cliente, neste componente teve de existir o cuidado em não permitir que um socket ZeroMQ fosse utilizado por diferentes threads ao mesmo tempo. Por exemplo, o socket `PUB` cujo objetivo é publicar noutros servidores de chat, é utilizado tanto quando são recebidas mensagens vindas de clientes como quando um cliente acabou de entrar ou sair da rede (para atualização do estado de utilizadores online).

Posto isto, o grupo fez uso de uma `BlockingQueue` do Java para onde as diferentes threads escrevem e de onde outra thread lê em FIFO e publica no socket. Cada elemento desta queue é simplesmente um par `(tópico, mensagem)`. Estando esta instanciada como uma `LinkedBlockingQueue` também não existe a possibilidade de encher e serem perdidas mensagens.

Servidor de Agregação

A cada servidor de agregação está associado um servidor de chat. Isto pois, o objetivo deste componente é unicamente a procura de quais os melhores `C` servidores de chat para atender a um tópico novo.

CYCLON

Outro dos requisitos presente no enunciado é o de manter o grafo da rede como sendo "aleatório". Isto é, os vizinhos de um dado nó devem ir alterando, implicando, assim, que um nó passa por várias zonas da rede durante o seu tempo de vida.

Para tal, foi implementado o algoritmo CYCLON, cujas propriedades melhor se encaixam no perfil deste serviço. Como é sabido, este algoritmo, para além de fornecer uma boa gestão na troca de vizinhos entre os nós, através "shuffles", também facilita a entrada de novos nós através de um "random walk".

Neste processo, dado um novo nó na rede, ele requisita `l` random walks através do seu nó introdutor. Esta funcionalidade permite que um nó recém-chegado se torne praticamente indistinguível de nós que já realizaram diversos shuffles. Para além disso, o CYCLON também se destaca na gestão de nós incontactáveis, acabando por descartar aqueles com os quais não consegue realizar um "shuffle", rapidamente eliminando da rede nós que se permanecem incontactáveis durante bastante tempo.

No entanto, importa referir que uma pequena limitação na nossa implementação se deve ao facto de termos utilizado sockets `PUSH` / `PULL` (um socket `PULL` por cada nó e um socket `PUSH` por cada vizinho de cada nó). Sendo estes sockets do tipo `PUSH`, não existe a possibilidade de implementar timeouts de envio. Ou seja, ao iniciarmos um "shuffle", não temos como determinar se um vizinho se encontra incontactável antes de prosseguir com o algoritmo; apenas é enviado o pedido de "shuffle" e o vizinho é removido da cache. Além disso, não é garantido que um nó que iniciou um "shuffle" responda a um pedido de "shuffle" de um terceiro nó, o que permite a troca de vizinhos repetidos caso o nó com que foi iniciado o primeiro "shuffle" responda imediatamente.

A única solução para tal seria utilizar sockets `REQ` / `REP` para a realização de "shuffles", de forma a saber de imediato quando um vizinho está incontactável, através do uso de timeouts, e utilizar sockets `PUSH` / `PULL` para realizar os "random walk", visto que estes não exigem resposta imediata por parte dos intermediários.

A respeito desse pequeno detalhe, o funcionamento do algoritmo permanece praticamente perfeito, onde todos os nós são vizinhos de muitos outros nós na rede durante o decorrer do programa.

Agregação Distribuída

O enunciado pede que a agregação distribuída seja realizada via gossip. Tendo em conta que o grafo é de natureza unidirecional (imposta pelo CYCLON) recorrer a uma agregação baseada em gossip pode não resultar em todos os casos - isto pois, não é garantido que as mensagens fluam pela rede toda facilmente e que voltem ao nó agregador rapidamente. Pode ainda acontecer de zonas do grafo convergirem primeiro que outras, dificultando ou impossibilitando que o resultado chegue ao nó agregador.

Assim sendo, qualquer que seja o algoritmo a empregar estaremos sempre limitados, dada a própria natureza do grafo. De qualquer forma, o algoritmo implementado baseia-se no Extrema Propagation, estudado nas aulas teóricas. Com algumas diferenças, no entanto.

No algoritmo original existe um parâmetro `nonews`, que é comparado com `T` a cada ronda, onde cada ronda é dada pelas respostas diretas dos vizinhos de um nó (via "reliable broadcast"). Caso esse `T` seja atingido naquela ronda significa que a convergência foi atingida naquele nó. Aqui, os vizinhos de um nó irão estar constantemente a trocar, dada a aplicação do CYCLON e, portanto, o `T` que definimos tem uma noção mais "global" e contabiliza respostas recebidas de qualquer vizinho naquele dado momento, sem que exista uma variável de `nonews`.

Depois, para mitigar a probabilidade de outras zonas do grafo, que não a do agregador, convergirem antes (impossibilitando uma resposta para o cliente) o grupo definiu que `T` será o parâmetro no caso do nó agregador e `2*T` será o parâmetro no caso de ser qualquer outro nó.

Outro problema existente é o facto de que a entrada de novos nós dificulta que a convergência seja atingida, visto que estes terão, na teoria, um servidor de chat melhor e irão fazer com que o `T` seja "resetado" para os nós com que comunicarem. Uma possível solução para este problema seria os nós deixarem-se afetar apenas por mensagens que já passaram por `X` nós (número de "hops"), aumentando a probabilidade da agregação terminar antes da mensagem ser "confiável".

Finalmente, e apesar de no enunciado referenciar que a execução de uma agregação é um processo raro, o grupo tomou a iniciativa de implementar suporte para diferentes agregações (diferentes tópicos) acontecerem na rede.

Notificação de Término

Quando o nó agregador converge e, portanto, a agregação termina, do seu ponto de vista, este tem o trabalho de notificar todas as partes interessadas.

Desta forma, o servidor de agregação trata, primeiramente, de notificar sincronamente cada um dos servidores de chat escolhidos; depois de notificar a DHT (inserindo lá o tópico como chave com a respetiva lista dos `C` servidores de chat escolhidos) e, finalmente, notifica o cliente que criou o tópico. Só assim garantimos que o cliente não consegue enviar mensagens antes de todas as partes estarem cientes de que o tópico existe.

Comunicação com Outros Componentes

Tal como no capítulo anterior, descrevemos todos os meios de comunicação utilizados para as diversas comunicações, com o exterior, por um servidor de agregação.

Servidor de Agregação

- Socket `PULL` para receber pacotes vindos de outros servidores de agregação que o têm como vizinho.

Servidor de Chat
- Socket `REQ` para realizar pedidos de estado ao seu servidor de chat (passado via argumento `-cs`).

Servidor de Pesquisa
- Socket `TCP` para comunicação com o nó da DHT passado como argumento aquando da inicialização do programa (via `-dht`).

Cliente
- Socket `REP` para responder a pedidos de agregação por parte de clientes.

Servidor de Pesquisa

Este componente é responsável por integrar um serviço de pesquisa que funciona como uma DHT. Essencialmente, cada chave corresponderá a um tópico de chat existente e cada valor ao conjunto de `C` servidores que atendem aquele tópico.

De facto, existem inúmeras estratégias diferentes para a construção de uma DHT, tais como: Consistent Hashing, Chord, Kelips, Tapestry, etc. O grupo optou por seguir uma abordagem baseada em Consistent Hashing com algumas otimizações por cima.

Este serviço possui uma simples interface de utilização, sendo que apenas disponibiliza duas operações para o exterior: pesquisar por um tópico e criar (ou atualizar) um tópico.

Funcionamento

Uma vez que a estratégia pela qual o grupo optou foi Consistent Hashing, no pior dos casos uma pesquisa envolve apenas um salto - isto acontece quando o nó que o cliente contacta não tem a chave e, portanto, o pedido terá de ser redirecionado.

Visto que os nós se conhecem uns aos outros (na totalidade) o redirecionamento de um pedido é simples e implica apenas fazer a mensagem chegar ao processo Erlang que toma conta da conexão com o nó que tem a chave requisitada pelo cliente. Esperando, depois, obviamente, pela resposta, que será reencaminhada para o cliente solicitante, via TCP.

Arquitetura

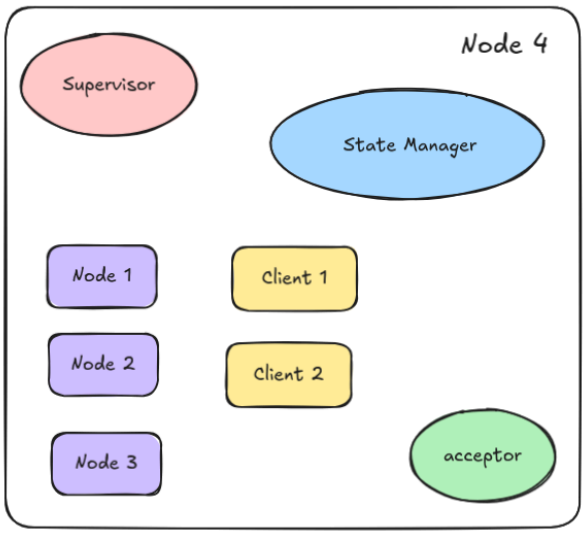

Figura 1: Arquitetura de um nó da DHT

Como apresentado na Figura 1, um nó da DHT pode ser dividido em 3 processos principais.

- O supervisor, que vai monitorizar todos os processos do nó;
- O state manager que guarda todo o estado relativo ao nó (chaves, valores, composição do anel e outro estado que seja necessário);

- O acceptor que aceita conexões de outros nós e de clientes.

Sempre que uma conexão é recebida ou iniciada pelo próprio nó, é criado um novo processo para tratar dessa conexão que vai ser, também ele, monitorizado pelo supervisor.

Nós Virtuais

Para garantir um balanceamento de carga mais uniforme e evitar que um único nó se torne um "hotspot" no anel, cada nó físico da DHT assume `P` (no nosso caso 3) posições lógicas no anel.

Quando o nó é inicializado, o state manager gera múltiplos "hashes" distintos, derivados do identificador do nó ("node4#1", ..., "node4# `P` "). Desta forma, em vez de ocupar apenas um ponto único, o nó físico passa a cobrir vários segmentos do espaço de "hash", o que faz com que os pedidos se distribuam de modo mais homogéneo mesmo quando há poucos participantes. Além disso, ao fragmentar o seu espaço de responsabilidade em vários intervalos menores, cada nó virtual reduz o impacto de entradas de nós, que envolve transferir menos dados por nó.

Entrada na DHT

Se um nó A se quiser juntar à DHT: primeiramente, envia uma mensagem com o seu endereço e nome ao nó 1 que é o ponto de entrada do sistema. O nó 1 irá responder com a composição atual do anel (endereços de outros nós e as suas posições); o nó A procede a calcular os seus nós sucessores e envia um pedido de transferência a esses nós com o respetivo intervalo de chaves que lhe correspondem.

Após receber resposta a todos os pedidos de transferência, o nó A inicia conexão com todos os nós da DHT (se a conexão ainda não existir) e envia as suas posições (dada a utilização de nós virtuais) no anel para poder, de facto, entrar no sistema e começar a servir pedidos. É neste momento que os restantes nós passam a saber da existência do nó A e, mesmo os nós que já conhecem o nó A, só poderão redirecionar pedidos para ele depois de receberem as suas posições virtuais.

Conclusão

Em suma, o presente relatório demonstrou a complexidade inerente ao desenho e implementação de um sistema descentralizado de chat, evidenciando a importância de certos mecanismos para sistemas distribuídos em grande escala, bem como a multitude de tecnologias existentes e capazes de ilustrar os mais diferentes paradigmas de sistemas distribuídos.

O grupo considera que o trabalho desenvolvido cumpre com todos os requisitos propostos pelo enunciado, bem como tenta ir além em alguns aspetos. Acima de tudo, foi possível aprender e aplicar conhecimento que outrora ainda era teórico. No entanto, alguns aspetos, nomeadamente o funcionamento do servidor de agregação, nos seus mais diversos aspetos, poderiam ter sido mais refinados e melhor executados. De qualquer forma, dada a dimensão do projeto e as restrições temporais existentes, o grupo está orgulhoso do que desenvolveu.