Universidade do Minho
Escola de Engenharia

Sistemas Distribuídos em Larga Escala

Mestrado Integrado em Engenharia Informática

Decentralized Timeline Service

A84961 Alexandre Ferreira
A84462 Alexandre Miranda
A85227 João Azevedo
A85315 Miguel Cardoso
A85729 Paulo Araújo

Braga,
Junho 2021

Abstract

Este projeto consiste na elaboração de uma *Timeline* (por exemplo: *Facebook*, *Twitter*, entre outros) com base num ambiente distribuído *Peer-to-peer*. Cada nó da rede tem uma identidade utilizada para identificar a origem de um certo conjunto de mensagens, publicadas na sua linha de tempo local. Para um dado nodo obter informação publicada por outros deverá subscrever os mesmos e o sistema deverá conseguir fornecer um participante da rede capaz de enviar essa informação, podendo este ser a fonte ou um subscritor da mesma. Por fim, o conteúdo partilhado deverá ser armazenado, de forma efémera, em subscritores de um dado nó fonte.

System model

Claramente, este sistema propõe um conjunto de desafios complicados de atingir num sistema distribuído em grande escala quando queremos garantir uma solução escalável e eficiente. O primeiro passo, passou pela definição dos requisitos associados a este sistema, requisitos esses que definem o modelo do sistema que estamos a considerar. Deste modo, consideramos os seguintes pressupostos:

1. Cada nodo na rede deve ter um identificador único, necessário para o processo de subscrição;
2. As mensagens publicadas por um nodo devem chegar a todos os seus subscritores ativos;
3. A linha temporal de cada nodo deve ser lhe apresentada segundo alguma ordem total;
4. Os nodos falham de forma graciosa, nomeadamente, através do *logout*;
5. A descentralização é apenas relativa aos *posts* na *Timeline*;
6. As publicações são guardadas nos nodos subscritores de forma efémera que, no nosso caso, representa o número máximo de mensagens por nodo (simula um limite de memória).

Uma solução baseada nestes pressupostos poderia passar pela utilização do conceito de *superpeers* onde estes seriam nodos como os outros mas com responsabilidades acrescidas. Alguns nodos ligar-se-iam a um *superpeer* e estes últimos comunicariam entre eles para fazer a distribuição das mensagens para os outros conjuntos de nodos. Apesar de ser uma implementação que já provou dar frutos, como por exemplo no *Gnutella* ou em outros sistemas *Peer-to-peer*, nós decidimos utilizar apenas nodos sem criar grupos de nodos ou algo que se pareça. Como não estamos a considerar falhas de nodos, isto é uma forma viável de resolver o problema.

Outro problema está presente na forma de como a informação a ser persistida seria acedida pelos nodos. Começamos por pensar num armazenamento distribuído, e com isso veio a ideia de utilizar uma das implementações de `DHT's` lecionadas, nomeadamente, o *Chord* e a *Kademlia*. O nosso problema identifica-se mais com a segunda opção, pois cada nodo poderia ser identificado da mesma forma que o conteúdo armazenado numa chave da *HashTable*. No entanto, não encontramos uma boa implementação desta `DHT` nas tecnologias que decidimos e estamos mais confortáveis em utilizar.

Assim sendo, a nossa solução baseia-se em dois componentes principais sendo eles um servidor central, responsável pelos serviços de autenticação e subscrição, e o nodo, um participante da rede que estará em contacto com outros nodos que subscreve e que o subscrevem. De esclarecer aqui que o serviço de autenticação incorpora todo o armazenamento de informação acerca dos diferentes utilizadores, servindo assim de substituto às referidas `DHT's`.

Our approach

A nossa abordagem apresenta, claramente, uma faceta centralizada e outra descentralizada. No primeiro caso, temos a então centralização da informação acerca dos nodos, das suas conexões ativas e ainda dos serviços de autenticação. No segundo caso, temos a total descentralização do armazenamento e propagação das mensagens das *Timelines* dos diferentes nodos.

Central server

A utilização deste servidor é fundamental para a manutenção da rede de nodos. Os novos nodos que entram no sistema não sabem nada sobre o estado atual da rede, nomeadamente, endereços `IP` e portas onde as diferentes subscrições e subscritores estão. Este servidor terá a tarefa de incorporar esses nodos na rede conectando-os de forma a que recebam e enviem as mensagens de forma correta. Em suma serve como porta de entrada para a rede.

Data structures

Para o armazenamento dos dados sobre um nodo tivemos que ter em atenção toda a informação pessoal e da sua rede para futura gestão. Posto isto, é necessário guardar uma lista dos `ids` de quem ele subscreve e ainda uma lista dos `ids` dos seu subscritores.

Os `ids` dos seu subscritores é fundamental para o que o central seja capaz de indicar a um outro nodo um local onde encontrar os seus *posts* caso ele esteja *offline*.

Os `ids` das suas subscrições, na realidade não são tão obrigatórias de guardar no central, uma vez que um nodo poderia guardá-las localmente e enviar mensagens a pedir conexões para cada uma delas aquando do *login*, no entanto isso geraria, no pior caso, muitas mensagens ou então uma mensagem muito pesada. Deste modo, o central já vai guardando essas informações tornando esse processo mais eficiente.

Por motivos de gestão de rede o servidor central guarda as conexões que cada nodo tem na mesma e por motivos de cálculo de disponibilidade é também guardado o tempo médio de sessão.

General use cases

- **Registo de um novo nodo:** Este processo é o mais simples, apenas insere um novo nodo na rede colocando-o como disponível (*online*) e sem quaisquer conexões ativas;
- ***Login* de um nodo:** Este processo é talvez o mais complexo pois incorpora um conjunto de tarefas distintas, para além de colocar o nodo como disponível (*online*):

  1. Registar uma nova sessão: Como referido anteriormente, o registo da sessão de um nodo permite-nos recolher estatísticas sobre o tempo médio de sessão que será útil no processo de eleição de nodos, que será explicado mais à frente;
  2. Gerir as suas subscrições: De forma geral, percorre-se a lista de subscrições do nodo e, para cada uma, elege-se um subscritor (ou o próprio nodo que é subscrito), atualmente disponível na rede, como fonte dos *posts* dessa subscrição. Através dessa fonte, o nodo poderá não só recuperar as mensagens que poderá ter perdido dessa subscrição como ainda receber os seus *posts* em tempo real;
  3. Gerir os seus subscritores: A partir da lista dos seus subscritores que estão *online* é gerada uma árvore de conexões para transmissão dos *posts* deste nodo. O algoritmo de construção da árvore será explicado em detalhe mais à frente.
  4. Atualizar conexões dos nodos: É atualizada a informação na base de dados sobre todas as novas conexões geradas a partir deste processo.

- ***Logout* de um nodo:** Este processo, para além de colocar o nodo como não disponível, faz ainda as seguintes tarefas:

  1. Atualização de estatísticas de sessão: Com base no tempo da sessão atual o servidor atualiza o tempo médio de sessão do nodo;
  2. Gerir as conexões que dependem deste nodo: Para todos os nodos que dependem deste para obtenção de informação de diferentes origens, o servidor central encarrega-se de eleger novos nodos que substituam este que está em processo de *logout*;
  3. Gerir os nodos que este depende: Posteriormente, é necessário remover este nodo de todos aqueles de quem ele depende;
  4. Atualizar conexões dos nodos: É atualizada a informação na base de dados sobre todas as novas conexões geradas a partir deste processo.

  Sabemos que o processo apresentado de gestão de conexões apósm um *logout* não é a única opção, pois poderíamos guardar isto localmente (em cada nodo) e notificávamos os dependentes através de um *post* de *logout* na *Timeline*. No entanto os nodos dependentes teriam de pedir ao central novas conexões podendo levar a maior tráfego na rede.

- **Subscrição:** Este processo é mais simples que o anterior tendo tarefas menos complexas pois apenas interferem com dois nodos. As tarefas a realizar são as seguintes:

  1. Eleição do nodo a conectar: Utilizando o mesmo algoritmo referenciado anteriormente, faz-se a eleição de um nodo capaz de propagar as mensagens para aquela subscrição. De notar que esse nodo poderá ser a fonte original ou um qualquer nodo subscritor que esteja *online*.
  2. Atualizar conexões dos nodos: É atualizada a informação na base de dados sobre todas as novas conexões geradas a partir deste processo.

Node election

A eleição de nodos poderia passar por uma escolha aleatória de um nodo subscritor associado a uma dada fonte. Esta escolha não tem em conta quaisquer métricas visto que não considera, por exemplo, a disponibilidade dos nodos, um fator muito importante em sistemas distribuídos, portanto, decidimos adotar esta métrica e uma outra para tornar este processo mais inteligente.

O processo de eleição adotado é simples, por cada *login* de um nodo registamos o tempo de início de sessão e, a cada *logout*, calculamos um resultado agregado do tempo médio de sessão. No entanto, o tempo de sessão poderia criar uma situação de *Preferential attachment* no nodo com mais tempo *online*, o que é, claramente, uma situação má em caso de falha e da carga total associada à propagação de mensagens pelo mesmo. Deste modo, decidimos utilizar um outro fator, a distância entre nodos, que é tido como um conceito abstrato dado pelo valor da diferença entre as portas, balanceando, assim, a "pontuação" atribuída a cada nodo candidato.

Assim, quando temos de eleger um nodo subscritor como *forwarder* de uma dada fonte original, temos em conta o seguinte algoritmo:

Algorithm 1: Calculate candidate node score

Result: score
score = candidade.upTime() / abs(candidate.port - subscriber.port)

Subscriptions tree

Este é outro processo referenciado na funcionalidade de *login* e que surge na necessidade de restabelecer uma árvore de conexões de subscritores a um nodo (subscrição) original quando esse nodo esteve *offline* e depois se autenticou.

Sabemos que num cenário onde temos muitos subscritores de um dado nodo X, que estabelecem, à partida, uma relação de ligação balanceada pela pontuação dos nodos candidatos, a falta desse nodo X causa a falta de quaisquer informação atualizada dessa fonte e, portanto, os nós são notificados que essa fonte já não está disponível. Entretanto, as ligações mantidas de subscritor para *forwarder* podem ter sido perdidas e novos nodos na rede podem ter entrado podendo ser mais favoráveis no processo de eleição.

Assim, temos a necessidade de construir a árvore de conexões ao nodo fonte do zero, estabelecendo o seguinte algoritmo aquando o *login* dessa fonte:

1. **Refazer a rede.** Remover quaisquer conexões estabelecidas com o nodo fonte ou com nodos intermédios;
2. **Ordenar as pontuações.** Calcular a pontuação de todos os nós subscritores ao nodo fonte (relação tempo de sessão por distância) e ordenar o resultado;
3. **Escolher os melhores candidatos.** Conectar os 50% melhores nodos à fonte de subscrição;
4. **Árvore *random*.** Criar uma árvore aleatória a partir daí, ou seja, para cada um dos outros 50%, escolher um nodo aleatório dos já conectados à rede e estabelecer conexão, adicionando o escolhido ao conjunto da próxima iteração;

Sabemos que para a criação desta árvore muitas alternativas poderiam ser implementadas, como por exemplo *Preferential Attachment*, no entanto, isto teria outras implicações na *performance*.

Nesse caso, no nível 2 da árvore ficariam, da mesma forma os 50% dos nodos com mais pontuação, mas depois não faria sentido utilizar o algoritmo anterior alterando o conceito de aleatoriadade para considerar pesos, pois iremos dar preferência para um *hub* que pode não ter qualquer relação de distância com o nodo a considerar no momento. Certo que esta última situação também poderá acontecer na implementação atual mas, de qualquer das formas, produz uma visão mais equilibrada das duas métricas consideradas na eleição de nodos.

Por outro lado, faria mais sentido fazer essa preferência de nodos por nível, ou seja, na primeira vez davamos mais peso ao nodo com maior pontuação e recalculávamos as novas pontuações para cada um dos nodos do primeiro nível. O algoritmo prosseguia tanto tempo quanto uma complexidade proporcional a O(N * N * log(N)) operações.

É preciso ter em atenção que a utilização de servidores centrais tem custos elevados, principalmente neste processo de construção das árvores e, portanto, queremos evitar, para 1000 nodos fazer log(1000) * 1000 * 1000 operações cálculos de pontuação e fazer apenas 1000, garantindo, ainda assim, uma solução equilibrada considerando o processo de eleição de nodos apresentado.

Nodes participating in the network

Este é o componente que tem mais relevância na visão descentralizada do sistema construído visto que a partilha da informação é feita entre nodos (*peers*), diretamente. As responsabilidades principais deste interveniente são de publicar mensagens na sua *Timeline* vindas dele próprio ou de outras fontes. Por outro lado, deve ser possível consultar a linha temporal presente em cada nodo, ordenada de forma total. Por fim, pressupõe-se um mecanismo de subscrição associado e comunicação constante com o servidor central para gestão da mesma.

Data structures

No componente nodo temos então que armazenar as informações relativa aos *posts* e que possibilite a propagação dos mesmos de forma ordenada. Assim sendo, cada nodo contém a sua *timeline* pessoal onde guarda todas as mensagens que publicou, a sua *timeline* total, onde tem não só os seu *posts* como também os das suas subscrições e ainda tem uma estrutura que mapeia as publicações de cada subscrição para que o acesso às mesmas seja facilitado.

Relativamente à ordenação de mensagens, o nodo deve manter o contador (*id*) da sua última mensagem postada e ainda o contador (*id*) da última mensagem recebida de cada subscrição. A forma como estes campos são utilizados será explicada mais à frente.

General use cases

- **Registo no sistema:** O nodo comunica com o servidor central indicando a sua *Network* e ainda o seu identificador, recebendo então uma resposta com a confirmação de que está *online* ou, em caso de algo correr mal, que nada aconteceu;
- ***Login* no sistema:** Da mesma forma que na funcionalidade anterior, o nodo comunica ao servidor central a sua *Network* (pode ter alterado entretanto), recebendo de volta as portas às quais se deve conectar para, por um lado, receber as publicações das suas subscrições e, por outro lado, para recuperação da *Timeline*. De notar que, antes de processar o pedido de autenticação, recupera-se a base de dados local deste nodo;
- ***Logout* no sistema:** Comunicação com o central de que este nodo já não está *online*. Quando receber confirmação do servidor central, persiste o seu estado e termina o programa graciosamente.
- **Subscrição de nodos:** Este pedido exige uma comunicação com o central para obtenção da porta do nodo eleito a partilhar informação acerca daquela subscrição;
- **Gestão da *Timeline*:** Constitui um conjunto de funcionalidades de atualização, manutenção e recuperação de mensagens na linha temporal de um nodo e, por isso, pode ser dividido da seguinte forma:

    1. Publicação de conteúdo: Cada nodo tem a possibilidade de adicionar uma mensagem à sua *Timeline* local que será armazenada de forma persistente e será publicada para todos os subscritores diretos;
    2. Receção e *forwarding*: Aquando a receção de uma mensagem num paradigma de *pub-sub*, este adiciona-a à sua timeline local (efémera) e publica-a para todos os possíveis subscritores, mesmo que não exista um nodo interessado na mesma (com a seleção da tecnologia feita a publicação só é feita caso haja interesse mas isso é nos invisível);
    3. Recuperação da timeline: Após receber a resposta do *login* e processar a mesma, completando as conexões que deve efetuar, o nodo envia a cada uma das suas conexões uma mensagem a pedir todas as mensagens que o nodo tem sobre a fonte origem daquela conexão. Este pedido leva um relógio incorporado correspondente ao *id* da última mensagem recebida daquela origem antes de o recém autenticado ter feito *logout* da última vez. Desta forma o nodo alvo deste pedido apenas vai enviar os *posts* que são posteriores a este relógio. Pode acontecer que este nodo recetor do pedido não tenha todas as mensagens que foram enviadas pelo nodo fonte no período em que o nodo original esteve adormecido devido à efemeridade presente no sistema. Neste caso o nodo tem que se contentar e ignorar a possível perda de algumas mensagens. Depois de receber a resposta de todas as fontes, o nodo reconstrói a *timeline* ordenando causalmente todos os *posts*.
    4. Ordernação das publicações e estruturas de dados: Este processo é especialmente importante quando falamos de linhas temporais que têm de ser mantidas numa ótica causal.

- **Estrutura de um *Post***: Uma publicação tem então o conteúdo da mensagem, o *id* de quem a publicou e ainda um *vector clock*. Esse vetor teria então o *id* da mensagem associado à mensagem publicada e ainda os *id* das última mensagens de cada subscrição da fonte aquando da publicação deste *post*.

- **Ordenação de mensagens**: Aqui tentamos que todos os nodos vejam a mesma ordem das mensagens tal como o *publisher* vê, ou seja, um *post* de um certo nodo só pode ser, possivelmente, causado por um outro *post* feito por uma subscrição do *publisher* ou por ele mesmo. Assim sendo, tendo os relógios das subscrições e também da própria origem, nodos que partilhem subscrições com este *publisher* conseguirão obter uma certa ordenação causal nas mensagens das subscrições comuns. Como é óbvio, nem todos têm as mesmas subscrições e nesses casos os posts serão considerados concorrentes sendo o critério de desempate por *id* do nodo que publicou.

- **Problemas com esta implementação**: Devido à dinâmica de subscrições podemos originar situações paradoxais, ou seja, publicações que eram concorrentes com aquele *publisher* podem passar a não ser nos *posts* futuros, podendo gerar um paradoxo na ordenação:
  * Temos 3 mensagens (a,b e c) em que, segundo o nosso algoritmo, a é depois de b e antes de c e ainda b é depois de c. Isto é possível no caso de a e b serem causalmente relacionados, porque partilham um ou mais id's de nodos nos seus *vector clocks*, a e c acontece o mesmo em que o resultado dessa causalidade é c anteceder a, mas c pode ser concorrente com b já que não partilha quaisquer relógios com ele. Aplicando o nosso algoritmo, c pode ser depois de b por ter um *id* de *publisher* maior. Desse modo temos então um paradoxo e como estamos a usar ordenação por inserção, a ordem pode variar consoante chegada das mensagens.

Architecture and technologies

A arquitetura deste sistema tem vindo a ser descrita como uma comunicação *request-reply*, segundo um modelo assíncrono, para serviços de autenticação e subscrição com o servidor central e uma comunicação *publisher-subscriber*, para os *posts* da *Timeline* entre nodos. No entanto, existem mais alguns tipos de comunicação considerados e, portanto, nesta secção, iremos dar destaque a todos esses tipos e tecnologias adotadas.

Começando pelo servidor central, este tem duas funções principais, um serviço para receber pedidos dos nodos e outro para poder notificar nodos acerca de mudanças na rede:

- *Event-driven* `central` *service*: Utiliza-se aqui a *package* `Atomix` para a construção e registo dos *handlers* para comunicação assíncrona em sistemas distribuídos. Mais propriamente, em todos os pedidos e respostas enviados aos nodos;

- *ZeroMQ* `notification (push)` *Pipeline*: Este padrão de *message oriented middleware* permite-nos distribuir as notificações para os servidores de `PULL` associados a cada nodo. Este canal é utilizado para notificar um nodo de que um *provider* ficou *offline* e que se deve conectar a outro nodo para receber as mensagens que antigamente vinham desse *provider*.

De seguida, o servidor de cada nodo requer mais complexidade em termos do número de serviços que tem à sua disposição, mas recorrendo ao mesmo conjunto de tecnologias:

- *Event-driven* `node` *service*: Assim como no caso anterior, é necessário registar todos os *handlers* das respostas do central e enviar pedidos de forma assíncrona e, portanto, é usado, novamente, a *package* `Atomix`;

- *ZeroMQ* `notification (pull)` *socket*: Necessário para a recessão das notificações *push* vindas do central;

- *ZeroMQ* `subscriber` *socket*: Este *socket* assina todas as suas subscrições definidas e processa cada mensagem da seguinte forma: faz *push*, através de uma comunicação `inproc` para a sua *Timeline* local e para a *thread* do seu *socket publisher*;

- *ZeroMQ* `publisher` *socket*: Este *socket*, tal como nome diz, recebe as mensagens vindas do seu `sub socket` (*posts* de subscrições) e da *GUI* (*posts* seus) e publica-as, mesmo que não exista subscritor

algum. Aqui ter escolhido *ZeroMQ* foi importante pois esta tecnologia consegue perceber se tem subscritores ativos e evita enviar mensagens em vão;

- *Timeline* `local`: recebe mensagens vindas do seu `socket sub` ou da *GUI* e adiciona-as às estruturas de dados de forma ordenada;

A escolha destas tecnologias vem ajudar a cumprir os pressupostos definidos para este modelo de sistema visto que, por um lado, nos permite cumprir alguns dos padrões de *publisher-subscriber* inerentes a este tema e também cumpre os requisitos de um sistema assíncrono onde a programação é, maioritariamente, baseada em eventos.

Possible improvements & conclusion

Como conlusão de todo o trabalho realizado percebemos que a construção de um sistema como este, em grande escala, tem inúmero desafios atracados e o que acaba por ser solução para umas situações pode ter impacto na *performance* noutras, acabando por ser um *trade-off* entre consistência da informação das *Timelines* e a possiblidade de particionamento da rede ou inclusão de pressupostos de dinamismo de subscrições entre *peers*, o que vem suportar a definição associada ao teorema de `CAP`.

Se prosseguíssemos com a implementação das duas facetas, centralizada e descentralizada, recorrendo a servidores centrais para gestão da rede seria, claramente, necessário considerar questões de tolerância a "faltas não graciosas", questões excluídas da nossa implementação.

Uma abordagem totalmente descentralizada também tem as suas implicações visto que as mensagens de *login*, *logout* seriam propagadas na rede através dos vizinhos dos nodos, assim como as publicações, o que poderia trazer implicações na *performance* da rede como um todo, para além das questões de ordenação de mensagens e já por isso é que todas estas questões normalmente são difíceis de manter em grande escala, pois questões como acordos distribuídos ou comunicação em grupo passam a ser relevantes como se verifica, por exemplo, num sistema *Blockchain*.