Distributed Map/Reduce

Relatório

João Nogueira – 89262
Tiago Melo – 89005
Licenciatura em Engenharia Informática
Computação Distribuída

Solução Base

- Optámos pela utilização de **threads** para lidar com o envio, receção e processamento de mensagens. Em runtime, um coordenador cria uma Thread para o “trabalho” que ele executa (delegação de tarefas), que só é iniciada após a divisão do texto em blobs e tokens e cria uma Thread por cada nova conexão, dedicada unicamente às mensagens trocadas com a mesma.
- O **worker** consiste numa implementação de um lifecycle que, essencialmente, só recebe mensagens e executa RPCs, dependendo da “task” da mensagem. A implementação dos algoritmos map e reduce foi feita de maneira relativamente simples, visto que a performance não foi uma das nossas principais prioridades.
- A comunicação entre coordenadores e workers é feita através de TCP Sockets e as mensagens são codificadas em JSON, onde cada dicionário tem um campo “task” e, dependendo do valor do mesmo, há campos adicionais.
- Para além das mensagens estabelecidas no protocolo, achámos relevante a adição das tarefas “heartbeat”, “backup_reg”, “backup_ack”.
- Foi seguido o protocolo do enunciado no sentido que: o worker envia uma mensagem do tipo register, o coordinator regista o novo worker utilizando um **id interno**, o coordinator envia map requests até esgotar os blobs e o coordinator envia reduce requests até ter apenas um item. Esta solução funciona visto que o reduce é feito com “dois passos para frente e um para trás”. Isto é, fazemos pop() duas vezes, esperamos pela resposta e damos append(resposta).

Solução distribuída

- Sempre que um worker se regista, o coordernador gera um id interno, que será util mais tarde para recuperar mensagens perdidas. O worker é adicionado a um array de workers e é notificado com o endereço do coordenador de backup, caso exista.
- A partir deste momento, o coordenador envia requests sempre que tem um worker disponível. Estes são guardados numa Queue, o que permite que haja um bloqueio do processo sempre que não há workers dispníveis. Essencialmente o coordenador: divide o texto passado como argumento em blobs, “tokeniza” as blobs usando a função tokenizer dada, envia map_requests destes tokens até todos estarem mapeados (dando append a todas as respostas a **listList**), envia reduce_requests até **listList** ter apenas um item. Novamente, isto funciona graças ao uso de uma recv_queue que permite que a lista diminua de comprimento 1 unidade a cada iteração.
- O(s) worker(s) respondem aos reduce requests sempre com uma sorted list. Esta lista é ordenada usando locale PT, seguindo o exemplo do output fornecido.
- Foi seguida uma lógica semelhante à de uma aplicação cliente-servidor, onde o coordenador atua como um servidor e os workers como a client counterpart.

Solução distribuída robusta

- Consideramos que a maneira de como os workers foram desenvolvidos permite uma boa escabilidade da solução. Por conseguinte, um dado coordenador permite uma conexão com um número indefinido de workers e a distribuição de tarefas é feita de acordo com o mesmo, não havendo pre-alocação nem hardcode das mesmas.
- Desenvolvemos o coordenador de forma a suportar um backup. Isto é, quando um coordenador se liga, tenta conectar-se ao endereço passado como argumento. Se conseguir, assume que é o master e aguarda que haja workers disponíveis. Caso contrário (se o endereço estiver ocupado), tenta conectar-se ao coordenador que o está a ocupar. Regista-se como backup e envia uma heartbeat, que sinaliza não só que está vivo, mas também que está pronto a receber mensagens sobre o estado.
- A sincronização entre coordenadores é apenas funcional quando ambos se ligam antes do trabalho do primeiro começar. Isto é, ambos se iniciam antes do primeiro worker. Posto isso, o master enviará ao backup as mensagens que recebe dos workers e o segundo processa-las-á de maneira a ter o seu estado atualizado e, caso o master morra, consiga retomar onde ele morreu, seja no map seja no reduce.
- A nossa solução apresenta as limitações de não permitir que um coordenador se ligue como backup a “meio” do trabalho e que um coordenador que se ligue após o backup entrar em serviço não assume que é backup, visto que a porta estava livre e tudo correu bem.

Tolerância a falhas

- Quando um worker se conecta a um coordenador, é imediatamente notificado do endereço do backup (caso exista) e caso o master morra, tenta conectar-se ao endereço fornecido. Caso não lhe tenha sido enviado nenhum endereço de backup, tenta conectar-se ao endereço do master durante um minuto, até dar timeout e desistir.
- Apesar de não o suportarmos em termos de sincronização, quando um backup se conecta, o master dá broadcast a todos os workers do backup que acabou de aceitar.
- Não há eleição de coordenadores visto que temos simplesmente uma regra de “o primeiro a conectar-se ao port dos argumentos vence”.
- Para além do master, o sistema permite a morte de 1 worker quando só existe um worker a trabalhar. Ou então a morte do primeiro worker a ser iniciado. Isto graças a uma tabela de pedidos que regista o ultimo pedido que foi a feito a cada worker. Assim, quando um dado worker morre basta acedermos a qual o pedido que foi perdido e adicioná-lo a uma lista de pedidos não processados (**unprocs**) que, após a **listList** ser reduzida a uma única entrada, será processada até já não ter entries.

Performance

- Consideramos que as seguintes alterações ao programa trariam bons resultados e uma melhor performance:
    > Estudo de qual o tamanho ideal de blobs (que não compremeta a tolerância de falhas mas que permita o envio de mensagens largas de forma a minimizar o número de envios)
    > Implementação de um algoritmo reduce com tempo de execução O(log(n)*n)
    > Map e Reduce requests a serem feitos sincronamente, de forma a evitar percorrer a mesma lista duas vezes.
    > Merging sorted lists corretamente

- Contudo, foram implementados diversos protocolos que permitem uma solução robusta e elegante, nomeadamente:
    > Workers que tentam reconectar se a coordenadores quando estes morrem
    > Sincronismo de backups que não implica o envio de mensagens demasiado grandes
    > Tolerância à morte de workers através de uma tabela de pedidos