rmtlog: Um sistema de log remoto

João Victor M. Tamm

2 de Outubro de 2018

1 Introdução

Neste trabalho prático foi proposto o desenvolvimento de um sistema de armazenamento de logs remoto. Esse sistema consiste em duas partes, um servidor - que será responsável pelo armazenamento - e os clientes que são responsáveis por enviar as mensagens que pretende armazenar. De modo a tornar a transmissão segura e confiável foram introduzidos mecanismos de janela deslizante, confirmação seletiva e um código de verificação de erros.

2 Desenvolvimento

Como supracitado, o desenvolvimento do sistema foi dividido em duas partes, cliente e servidor. A linguagem utilizada foi *Python* em sua versão 3. Em seguida estão detalhadas cada uma das etapas do projeto.

2.1 Programa Cliente

O programa do cliente - emissor - é responsável por gerar as mensagens, empacotá-las e enviá-las ao servidor. Nesta etapa, foi determinado que as mensagens deveriam ser lidas de um arquivo, previamente determinado no momento da execução, no qual cada linha representa uma mensagem que deve ser empacotada e enviada de forma independente.

Tendo dito isso, a estratégia utilizada consitiu em modularizar o programa de forma que cada módulo tenha a sua própria responsabilidade, facilitando assim a manutenção e o desenvolvimento. Sendo assim, esta parte do sistema foi dividida em três arquivos: *cliente.py*, responsável por controlar o fluxo principal, *Transmitter.py* que implementa o transmissor e portanto monta e envia os pacote além de tratar as confirmações recebidas e o reenvio de pacotes e por último *ClientSlidingWindow.py* - que trata tudo que diz respeito à janela deslizante.

Em primeiro lugar, vamos descrever o funcionamento geral do programa para depois entrar em detalhes da implementação. Como dito anteriormente, o arquivo *cliente.py* é responsável primeiramente por criar uma instância do Transmissor e em seguida realizar a leitura do arquivo. A ideia implementada consiste na utilização de um loop que itera sobre o arquivo lendo linha por linha, montando e enviando o pacote, até que a janela deslizante seja preenchida por completo. Uma vez preenchida, o programa entra em um estado de espera ocupada que aguarda a confirmação do pacote que se encontra na primeira

posição da janela ser enviada corretamente pelo servidor. Ao receber este *ack* a espera se encerra e um novo pacote é inserido na janela e esse processo se estende até que o arquivo seja lido por completo. O primeiro desafio enfrentado foi evidenciado neste momento, ao realizar alguns testes foi possível perceber que o programa chegava ao fim da execução sem que todos os pacotes tivessem recebido seus respectivos *ack's*. Para solucionar tal imprevisto foi criado um loop que perdura enquanto existirem *ack's* a serem recebidos. No interior desse loop está uma chamada para a função responsável por tratar o recebimento das confirmações. Por último, a execução chega ao fim com a impressão do número de mensagens distintas enviadas, o número de pacotes transmitidos - incluindo retransmissões - o número de mensagens transmitidas com MD5 incorreto, e o tempo total de execução. A realização de testes no quais o cliente transmitia com altas taxas de erro evidenciaram que a variável responsável por contar a quantidade de envios não estava se comportando como o esperado. A solução está descrita na Subseção 2.3.

O Transmissor foi implementado através da criação de um classe para facilitar o processo de instanciação e controle ao longo da execução. Neste módulo, cada objeto possui como atributos uma instância do socket UDP para transmissão de dados sem conexão, a janela deslizante, uma tupla (HOST, PORT) com o endereço de envio, as variáveis contadoras descritas no parágrafo anterior, o tempo de timeout e a taxa de erro. Além destes atributos esta classe implementa os métodos necessários para montar o pacote, realizar o envio e o reenvio além daquele responsável por tratar os *ack's* recebidos. A montagem do pacote foi realizada com o auxílio de uma biblioteca denominada *struct*, que cria a formatação do pacote com os devidos tamanhos e byte order dos campos.

A janela deslizante(*ClientSlidingWindow.py*) foi criada por meio do uso de classes também pelo mesmo motivo citado anteriormente. Nesta parte do sistema foi implementada uma lista de objetos da classe *SlidingWindowElement*. Esta última estrutura foi criada para armazenar o pacote com o MD5 correto, o número de sequência para facilitar a procura na lista, um atributo booleano que registra se o pacote já recebeu a confirmação e por último um Timer que implementa o timeout para reenvio do pacote. O Timer foi criado utilizando o objeto de mesmo nome da biblioteca *threading*. Voltando ao módulo principal da janela, devido à necessidade de se deslizá-la a lista foi criada utilizando *collections.deque* uma vez que é possível determinar um tamanho máximo e a cada elemento inserido ele entra na última posição removendo o primeiro. Sendo assim, foi definido que o tamanho desta lista seria o tamanho máximo da janela. A classe armazena também o tamanho atual da janela com o intuito de saber se ela se encontra cheia ou não. Os métodos implementados consistiram apenas em inserir e procurar na lista além de verificar se todos os pacotes que se encontram na janela já receberam o ack.

Conforme definido a execução desta parte do sistema deve ser feita rodando o seguinte comando: **python3 cliente.py arquivo IP:port Wtx Tout Perror**

2.2 Programa Servidor

O programa do servidor - receptor - é responsável por receber as mensagens de log e armazená-las em ordem em um arquivo. O servidor é capaz também de receber conexões de vários clientes simultaneamente e manter a ordem dos logs por cliente, ou seja, as mensagens de cada cliente será armazenada em ordem

porém não é necessário uma ordenação dos clientes.

A estratégia adotada foi simplesmente criar um loop infinito que fica constantemente recebendo conexão dos clientes. A implementação realizada seguiu os mesmos padrões de modularização utilizados pelo cliente, resultando assim nos arquivos: *servidor.py* que controla o fluxo de execução, *Receiver.py* que é responsável pela criação do receptor que trata o recebimento das mensagens e envio dos ack's, *ServerSlidingWindow.py*, responsável por criar a janela deslizante do servidor, que é ligeiramente diferente da descrita anteriormente e por fim *Client.py* - que cria uma estrutura para os clientes.

A ideia central que permeia esta fase da implementação consiste basicamente em criar um loop que aguarda que clientes mandem mensagens. Uma vez recebida a mensagem é verificado se o cliente já se conectou antes ou não de forma a definir se é necessário criar uma instância para este cliente ou não. Toda mensagem recebida, caso ela esteja dentro dos limites da janela do cliente, é salva na janela para que ela posteriormente possa ser escrita no arquivo. No final de cada uma das requisições a janela do cliente é percorrida verificando se existem logs em ordem para serem armazenados. Caso tenha estes são escritos no arquivo de saída e a janela é deslizada.

O Receptor, como dito anteriormente, foi criado através do uso de classes. Os atributos necessários para um total funcionamento foram um socket UDP que fica escutando por novas mensagens, uma lista de objetos responsável por armazenar os clintes - detalhes especificados abaixo - a taxa de erro de tranmissão e um mutex para realizar acesso na lista de clientes por exclusão mútua. Os métodos utilizados foram os responsáveis por tratar o recebimento da mensagem, por enviar o ack, inserir um cliente na lista e procurar um cliente na lista. Cabe aqui dar uma ênfase maior naquele que foi necessária uma atenção maior e que corresponde a essência do sistema, o que trata a recepção de pacotes. Primeiramente, a função fica esperando o recebimento dos dados que, assim que recebidos, o *header* (quatro primeiros campos) é desempacotado, com o uso da mesma biblioteca utilizada para empacotar, para facilitar a manipulação. Feito isso, uma verificação é realizada para ver se o cliente já enviou mensagens anteriormente ou não, caso o mesmo não exista na lista ele é inserido. Por fim o MD5 é verificado sobre os dados que foram enviados antes de serem empacotados. Na situação em que o pacote não foi corrompido é necessário checar a situação do número de sequência do pacote em relação à janela, tendo três casos possíveis:

> (i) **Menor que o primeiro da janela**: O ack é reenviado para manter sincronia entre cliente e servidor;
>
> (ii) **Maior que o último da janela**: O pacote é ignorado uma vez que este se encontra fora do limite da janela;
>
> (ii) **Dentro dos limites da janela**: Mensagem é armazenada na janela do cliente juntamente com o número de sequência.

Vale a pena aqui ressaltar que, caso dois pacotes cheguem com o mesmo número de sequência e hashes MD5 corretos porém com o conteúdo diferente exitem duas possibilidades de resultados:

> (i) Quando o segundo pacote chega depois do primeiro mas o primeiro

ainda não foi salvo no arquivo o último sobrescreverá a mensagem que foi recebida anteriormente;

(*ii*) Na situação em que o último chega e o primeiro já foi salvo no arquivo o que chegou depois será ignorado e apenas um ack é enviado atestando que um pacote com aquele número de sequência já foi recebido.

Conforme citado acima, uma estrutura foi criada para armazenar o cliente que enviou mensagens. Cada cliente é um objeto e possui sua janela própria e seu endereço.

A janela deslizante desenvolvida nesta etapa do projeto é ligeiramente diferente daquela que foi criada para a parte do cliente, uma vez que é necessário armazenar as mensagens e depois escrevê-las em ordem no arquivo de saída. Criou-se portanto, utilizando a mesma biblioteca *collections*, uma estrutura que é inicializada no tamanho da janela com todas as posições preenchidas com seus devidos números de sequência e mensagens vazias. Sendo assim uma janela de tamanho cinco é inicializada com os números de zero a quatro e a cada vez que a mesma desliza uma nova posição com o sequence number correto é criada.

Conforme definido a execução desta parte do sistema deve ser feita rodando o seguinte comando: **python3 servidor.py arquivo port Wrx Perror**

2.3 Mutex

O uso de locks para realizar operações mutuamente exclusivas se mostrou necessário uma vez ao longo do desenvolvimento foram detectadas algumas situações nas quais condições de corrida existiam. No servidor foi criado um mutex para controlar o acesso à lista de clientes, para que não ocorresse alterações simultâneas. Já no cliente, como citado anterior foi criado um lock para gerenciar o acesso às variáveis contadoras para que não houvesse conflito na hora de incrementá-las.

3 Análise de desempenho

A análise de desempenho foi realizada observando os impactos do aumento da taxa de erro nos envios das mensagens. De modo a obter um resultado mais fiel foi adotada uma estratégia de, para cada taxa de erro - que varia de 0.0 a 0.7, executar o programa quatro vezes e pegar a média dos resultados. Feito isso foram plotado dois gráfico utilizando uma janela de tamanho 100 e um arquivo de 966 linhas. O primeiro deles é a relação entre o número de retransmissões e a taxa de erro. Já o segundo demonstra a relação entre a taxa de erro e o tempo de execução.

Figura 1: Impacto do aumento da taxa de erro no número de retransmissões.

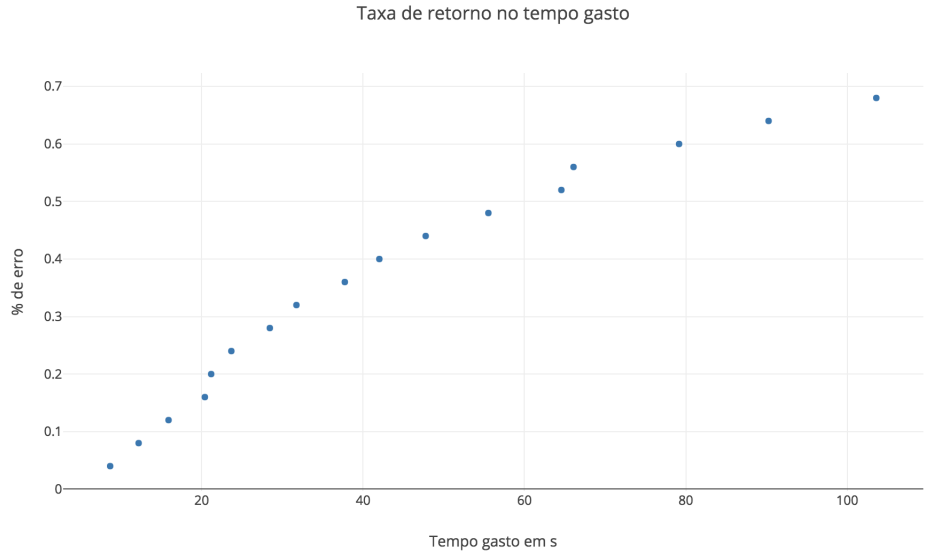

Figura 2: Impacto do aumento da taxa de erro no tempo de execução.

4 Conclusão

O trabalho contribuiu positivamente para o crescimento como cientista da computação. Foi possível colocar em prática os conceitos vistos em sala de aula como janela deslizante, transmissão UDP e a criação de meios confiáveis. Os problemas identificados foram descritos ao longo deste documento e foram solucionados consultando a internet sem grandes dificuldades.