TP1: Um sistema de log remoto (rmtlog)

Nome: Artur Henrique Marzano Gonzaga **Matrícula:** 2016006263

Nome: Tiago de Rezende Alves **Matrícula:** 2016006948

1 Introdução

A proposta desse trabalho prático é desenvolver uma biblioteca, composta de um cliente e um servidor, que realize a transmissão confiável e o armazenamento de logs entre dispositivos remotos. Desse modo, faz-se possível que aplicações em diferentes máquinas gerem mensagens de log e as enviem para um único dispositivo servidor, que as coleta e armazena. Além da proposta geral, os programas deveriam utilizar o *MD5* para verificação de integridade das mensagens transmitidas e simular um canal problemático corrompendo alguns dos MD5s. Por fim, visando a confiabilidade e eficiência da transmissão, solicitou-se a implementação do protocolo de janelas deslizantes de tamanho configurável aplicando o método das confirmações seletivas.

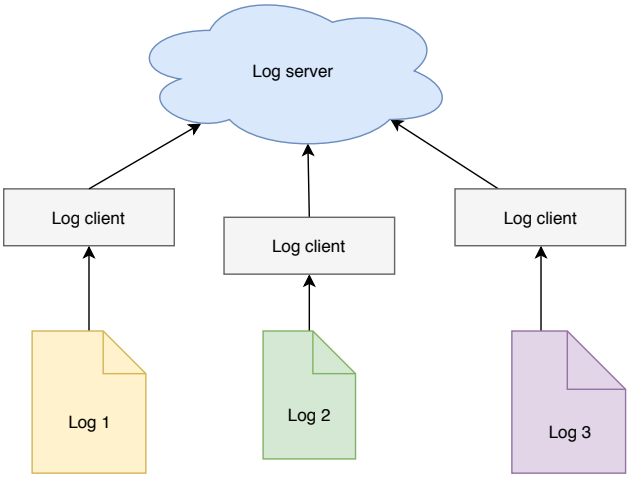

Figure 1: Esquema geral do sistema.

1.1 Protocolo da Janela Deslizante

Conforme solicitado, implementamos um cliente e servidor que se comunicam pelo protocolo de **janelas deslizantes com confirmações seletivas**. O objetivo principal desse protocolo é promover uma maior utilização de banda ao permitir que o transmissor envie um certo número de quadros sem esperar a imediata confirmação desses. O tamanho da janela corresponde ao número máximo de quadros nessa situação. Desse modo, o transmissor envia continuamente as mensagens, de modo a manter a diferença entre o número de sequência do último pacote transmitido e o número de sequência do pacote mais antigo sem confirmação menor que o tamanho da janela do transmissor. Devido a essa característica, um temporizador é associado a cada quadro da janela no momento de envio, responsável por reenviar o quadro caso sua confirmação não chegue dentro de um intervalo de tempo pré-determinado.

A *confirmação seletiva* se refere ao fato do receptor continuar recebendo outros quadros dentro de sua janela, mesmo que não sejam o primeiro da janela (próximo esperado), e enviando suas respectivas confirmações. Similarmente ao transmissor, a janela do receptor trava quando a diferença entre o número de sequência do último quadro confirmado e o número de sequência do quadro mais antigo ainda não recebido for maior ou igual ao tamanho da janela. Esse protocolo é robusto contra a chegada de quadros fora de ordem, pois o receptor os coloca na janela em ordem.

2 Detalhes de implementação

A linguagem C foi escolhida para implementação do trabalho a fim de proporcionar eficiência e um ajuste mais fino dos aspectos internos dos programas. Utilizamos a interface de sockets para facilitar a comunicação dos programas via UDP. O cliente foi implementado como segue:

1. As linhas do arquivo de *log* são progressivamente lidas, inseridas na janela e enviadas, com o número de sequência **partindo do 0**, até que a janela deslizante relativa ao cliente esteja completamente cheia ou o arquivo acabe.
2. Após a transmissão de cada mensagem é associado um temporizador a ela que, após uma quantidade de tempo especificada na entrada, retransmite a mensagem.
3. Logo após o primeiro enchimento da janela, o cliente bloqueia até que chegue a confirmação do primeiro quadro. Qualquer confirmação que chegue é registrada no seu quadro correspondente dentro da janela e tem o temporizador associado destruído.
4. Quando chega a confirmação do primeiro quadro, o cliente desliza a janela em uma unidade – ou seja, insere um quadro para a próxima linha de log no fim da janela e libera o espaço do primeiro quadro da janela, e volta ao passo 3.
5. Ao fim da leitura, o cliente espera a recepção das confirmações restantes da última janela, sempre removendo da janela o primeiro quando sua confirmação chega, até fechar a janela.

No servidor, a janela de recepção funciona de forma ligeiramente diferente, uma vez que o dado que precisa de garantia de chegada é apenas aquele proveniente do cliente. O servidor opera da seguinte maneira:

1. O servidor recebe mensagens indefinidamente. A cada mensagem recebida, ele identifica o cliente por seu IP e porta de origem na lista de clientes e, caso não esteja na lista, insere um novo cliente.
2. Cada cliente na lista tem sua própria janela deslizante do tamanho especificado associada. Essa janela começa preenchida com mensagens vazias aguardando recepção.
3. Quando chega uma mensagem, se o número de sequência for inferior ao da primeira mensagem esperada, o servidor reenvia a confirmação, pois significa que alguma confirmação anterior não chegou ao cliente. Se o número de sequência estiver dentro da janela, ela é inserida na janela e a confirmação é enviada. Caso o número esteja além da janela, a mensagem é ignorada.
4. Após inserção de mensagens na janela, todas as mensagens que já chegaram mais à esquerda são escritas no arquivo e a janela é deslizada, liberando espaço à esquerda e alocando espaço à direita para as próximas mensagens que agora podem chegar. A janela só desliza quando a primeira mensagem chegar, mas pode deslizar tantas unidades quanto mensagens da janela a depender de quais mensagens já chegaram.

Em ambos os casos (cliente/servidor), o MD5 da mensagem ou da confirmação é calculado e verificado imediatamente após a recepção. Mensagens com MD5 incorreto são ignoradas ao chegarem no servidor, da mesma forma que confirmações com MD5 incorreto ao chegarem no cliente.

2.1 Casos anômalos

2.1.1 Mensagens candidatas

Suponha que o servidor receba duas mensagens de um mesmo cliente, com o mesmo número de sequência e ambas com suas hashes MD5 corretas, mas com conteúdos diferentes.

A operação de servidor que não trata esse problema seria a seguinte, a depender do caso:

- Se a primeira candidata a chegar se torna a próxima esperada antes que a segunda candidata chegue, o servidor escreve a primeira no arquivo, envia a confirmação e desliza a janela. Quando a segunda candidata chega, ele confere que o número de sequência já passou e reenvia a confirmação para esse número de sequência. No final, apenas a primeira foi escrita no arquivo.

- Se a primeira candidata não se tornar a próxima esperada e ficar esperando na janela até logo depois da chegada da segunda candidata, há um conflito e devemos decidir qual manter na janela.

Reconhecemos que não há como determinar a ordem relativa entre as mensagens candidatas. Além disso, no primeiro caso não há como saber, no momento de chegada do próximo quadro esperado, se chegará outro quadro com o mesmo número de sequência em um momento futuro, de tal forma que o problema só poderia ser detectado se o servidor mantivesse o MD5 associado com o número de sequência de todas as mensagens que já saíram da janela – o que vai contra a proposta de economia de espaço da janela. No segundo caso, se removermos a mensagem que já está na janela e descartarmos a outra candidata, a janela do cliente pode ficar dessincronizada com a do servidor, pois o cliente já recebeu a confirmação da mensagem que estava na janela (naquele momento não era possível saber que o conflito aconteceria) e não transmitirá a candidata novamente.

Se alterarmos os números de sequência a partir das mensagens problemáticas perdemos o controle da ordem dos quadros – o quadro é quem deve saber sua ordem de envio e o servidor não deveria "dar palpite", especialmente basedo em casos anômalos.

Outra opção seria criar novos tipos de mensagem que permitam coordenação entre cliente-servidor quando um problema desse tipo for identificado, mas isso teria que ser feito de forma *backwards-compatible* com o padrão anterior.

Nosso servidor opta pelo melhor custo-benefício e simplesmente **descarta uma candidata** se já houver uma mensagem na janela. Assim, em ambos os casos, a primeira *a chegar* – que não necessariamente é a primeira enviada – é a que será escrita no arquivo de log.

2.1.2 Clientes inativos

Nosso servidor implementa um **temporizador para liberar espaço alocado para clientes inativos**. Um cliente que deixar de comunicar-se com o servidor por mais de 30 segundos é automaticamente removido. Dessa maneira, os requerimentos de recursos (espaço, processamento, etc) para execução do servidor dizem respeito apenas a quantos clientes simultâneos ele terá de suportar.

Caso esse temporizador não fosse implementado, o servidor teria de ser reiniciado esporadicamente para liberar os recursos que foram alocados para clientes que não estão mais presentes.

2.2 Bibliotecas & Portabilidade

Utilizamos a *librt* para termos acesso a *POSIX Timers* com as funções *timer_create*, *timer_settime* e *timer_delete*. No cliente, ela é utilizada para os temporizadores de retransmissão, enquanto no servidor é utilizada para os temporizadores de clientes inativos. Utilizamos a *libpthread* para criação de mutexes em recursos compartilhados. A *libcrypto* foi utilizada para cálculo dos MD5s (*openssl/md5.h*).

As funções *nonstandard htobe64* e *be64toh*, presentes na *glibc* desde a versão 2.9 (lançada em 2009), foram usadas para conversão entre *host order* e *network order* de valores de 64bits, como os números de sequência e *timestamps*.

2.3 Implementação das janelas

A janela do servidor foi implementada como uma lista encadeada e a janela do cliente foi implementada como um vetor de mensagens. Isso se deve à experimentação durante o desenvolvimento do trabalho – queríamos descobrir qual implementação era mais eficiente/adequada. Começamos com a a lista encadeada nas duas pontas e eventualmente trocamos a janela do cliente para um vetor. Embora a implementação por lista encadeada no cliente tenha tido como consequência um alto número de operações de memória (chamadas a *malloc/free*) e possivelmente degrade sob cargas extremas, observamos que as duas implementações tiveram performance similar para as cargas testadas. Atribuímos esse resultado ao fato de que, na implementação por lista encadeada, inserir as mensagens no envio tem custo constante, mas buscá-las para marcar as confirmadas tem custo linear, enquanto na implementação por vetor inserir as mensagens no envio tem custo linear (para arredar o vetor) e buscá-las para marcar as confirmadas tem custo constante. Como toda mensagem tem que ser eventualmente **inserida** e **confirmada**, a complexidade de tempo das duas implementações é equivalente.

2.4 Exclusão mútua

Utilizamos *mutexes* para garantir exclusão mútua em operações que causariam condições de corrida.

No servidor, uma *mutex* para a lista de clientes garante que o temporizador dos clientes não esteja removendo clientes inativos enquanto a lista está sendo percorrida, outro cliente está sendo adicionado ou a mensagem está sendo inserida na janela do cliente.

No cliente temos uma *mutex* associada ao temporizador de cada confirmação para garantir que o temporizador não seja deletado em resposta à chegada de uma confirmação ao mesmo tempo que está sendo armado novamente após a retransmissão. Também usamos uma *mutex* para manter a janela estável – a *thread* do temporizador não pode acessar a janela ao mesmo tempo em que a janela está sendo movida pela thread principal – e duas *mutexes* para valores do log de execução: uma para o número de mensagens transmitidas e outra para o número de mensagens transmitidas com erro, pois ambas as variáveis podem estar sendo incrementadas ao mesmo tempo pela *thread* principal enviando mensagens e por temporizadores retransmitindo mensagens.

3 Instruções para compilação e execução

Como o trabalho foi implementado em C, faz-se necessário a compilação do código fonte. Para tal, estando no diretório raíz do projeto, basta executar o *Makefile* incluído com o seguinte comando:

```
make all
```

Após a compilação, execute o programa conforme especificado:

```
./cliente <arquivot> <IP:port> <Wtx> <Tout> <Perrort>
./servidor <arquivor> <port> <Wrx> <Perrorr>
```

Dos parâmetros que correspondem ao cliente, o <*arquivot*> é o nome do arquivo contendo as linhas de log a serem transmitidas, <*IP:port*> é o IP e porto que o servidor está usando, <*Wtx*> é o tamanho da janela de transmissão, <*Tout*> é a quantidade em segundos do timeout para retransmissão das mensagens, <*Perrort*> é a probabilidade do cliente mandar uma mensagem corrompida. No servidor, <*arquivor*> é o nome do arquivo no qual serão escritas as linhas de log recebidas, <*port*> é o porto que o servidor irá receber as mensagens, <*Wrx*> é o tamanho da janela de recepção e <*Perrorr*> é a probabilidade do servidor enviar uma confirmação corrompida.

No modo normal, o trabalho imprime apenas as mensagens lidas (no cliente) e as mensagens recebidas (no servidor). Existe ainda a possibilidade de executar o trabalho em modo *debug*, no qual o conteúdo ds mensagens não é mostrado e, ao invés, são impressos na tela metadados a respeito das mensagens recebidas e enviadas, clientes, janela de transmissão ou recepção, dentre outras. Para tal, estando no diretório do projeto, execute o *Makefile* com o seguinte comando:

```
make debug
```

Após a compilação, execute o cliente e o servidor da mesma forma descrita acima.

4 Testes e análises de desempenho

Diferentes testes e análises foram feitos com o intuito de se verificar a corretude e desempenho do trabalho pŕatico. Todos os testes foram realizados em uma máquina rodando Arch Linux com Intel Core i5 e 6GB de RAM.

4.1 Um cliente, um servidor

4.1.1 Efeito dos erros

A fim de aferir o efeito dos erros de transmissão e de confirmação na eficiência da implementação, relacionamos a probabilidade de erro com o número de retransmissões de mensagens e com o tempo total de execução.

Esses testes foram realizados para temporizações variando entre $t_{out} := \{1, 2, 3, 4, 5\}$ segundos, janelas de tamanho 100 nas duas pontas, um log com *966* linhas (obtido do *kernel ring buffer* com o *dmesg*) e probabilidades de erro de 0 a 0.7, variando com resolução 0.04. Os valores mostrados são a média aritmética de 4 valores medidos para cada experimento. O debugging foi desabilitado para evitar atrasos de I/O.

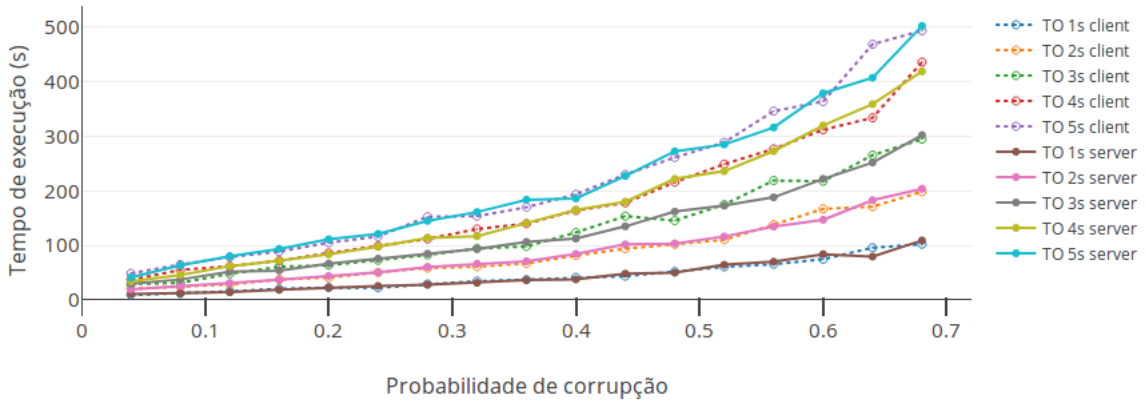

Figure 2: Tempo de execução por probabilidade de corrupção para corrupções no cliente (mensagem original) e no servidor (confirmação).

Para referência, sem corrupção em nenhum dos lados, um cliente nessa configuração de teste leva em média 128 milissegundos para concluir a operação, independentemente do valor do *timeout* (pois nenhuma mensagem precisa ser retransmitida), o que parece um tempo razoável. Na implementação de referência (do professor), o mesmo teste leva em média 620 milissegundos.

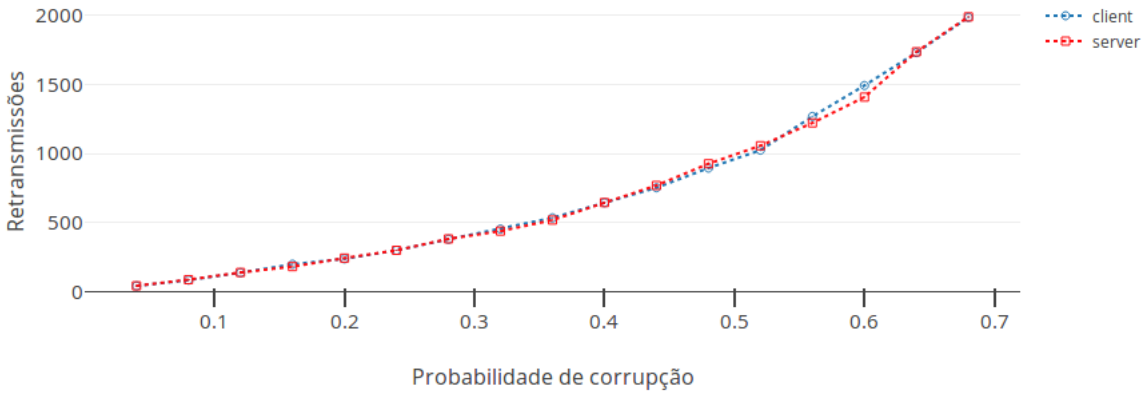

Figure 3: Número de retransmissões por probabilidade de corrupção para corrupções no cliente (mensagem original) e no servidor (confirmação).

Ao analisar o gráfico da figura 2, nota-se em primeira instância que há uma notória tendência crescente do tempo de execução relativo ao aumento da probabilidade de erro, acentuada pelo aumento do *timeout* relativo ao cliente. Além disso, o tempo de execução é indiferente à origem da corrupção, já que o número de mensagens a serem retransmitidas é o mesmo, conforme mostrado na figura 3. A probabilidade de uma mensagem enviada ter de esperar um *timeout* para ser retransmitida é $P_{err} = P_c + (1 - P_c) * P_s$, onde P_c é a probabilidade da mensagem corromper e P_s é a probabilidade da confirmação corromper. Essa é a probabilidade da mensagem **ser corrompida no cliente, portanto ignorada pelo servidor** ou dela **não ser corrompida no cliente e a confirmação corromper no servidor**. Definir a probabilidade de algum dos eventos (P_c ou P_s) como 0 define a probabilidade geral P_{err} de uma mensagem qualquer ter de ser retransmitida como a probabilidade do outro evento.

4.2 Múltiplos clientes

Realizamos testes com até 2500 clientes rodando em *background*, com corrupção nas duas pontas. Utilizamos um *log* para cada cliente, onde as linhas são sequencialmente numeradas de 1 a 100 e marcadas com o número de seus clientes. Utilizamos essas marcações para conferir a ordenação, para cada cliente, das linhas do arquivo de saída do servidor.

Até por volta de 1500 clientes o log do servidor esteve tanto completo quanto ordenado em relação a cada cliente. Para mais clientes, no entanto, começamos a receber mensagens de erro **do sistema** na shell rodando os clientes e os logs deixaram de estar completos, embora ainda ordenados a nível de clientes. Isso se deve a limitação de recursos (e não à capacidade do servidor, que não fomos capazes de testar ao extremo), uma vez que rodamos todos os clientes na mesma máquina. Experimentações com os limites de recursos (*ulimit*) elevaram esse limite para 2500. Especulamos que o servidor seja capaz de lidar com números ainda maiores de clientes simultâneos.

4.3 Logs grandes

A fim de confirmar a robustez do trabalho, realizamos 20 testes com um log com 1.000.000 linhas, corrupção de 0.001 nas duas pontas, *timeout* de 1 segundo e janelas de tamanhos diferentes – 5000 no cliente e 4900 no servidor. Todos os testes rodaram com sucesso, escrevendo o arquivo completo e em ordem, com tempo de execução médio de 716.495s e desvio padrão de 14.698s.

5 Conclusão

Consideramos que o trabalho foi concluído com sucesso apesar de dificuldades relativas à linguagem de escolha. O trabalho foi essencial para fixar conhecimentos sobre enquadramento, confiabilidade, camada de transporte e funcionamento de redes em geral. Acreditamos que o requerimento de suportar múltiplos clientes tornou a atividade bem mais trabalhosa do que esperávamos para um segundo trabalho da disciplina, mas também gerou discussões proveitosas a respeito de *threads*, sincronização, condições de corrida, etc, que contribuíram para a compreensão de problemas e soluções recorrentes no design de aplicações em rede.