INSTITUTO TECNOLÓGICO DE AERONÁUTICA
Laboratório de CES-33

Laboratório 3

Chat com Threads

Professora:
Cecília de Azevedo Castro César

Turma Comp-20
Claudio Felipe Lázaro da Silva – claudiosilva.cfls@gmail.com
Eric Toshio Endo Soares - eric.toshio.e.s@gmail.com

São José dos Campos – SP
Abril / 2019

1. Implementação

a. Ideia geral

A ideia desse laboratório é simular o servidor de um sistema de chat que utiliza uma thread diferente para cada cliente. O servidor consiste de um histórico das mensagens - implementado com lista ligada - onde cada nova mensagem é acrescentada ao final. A Figura 1 ilustra como seria o funcionamento para dois clientes.

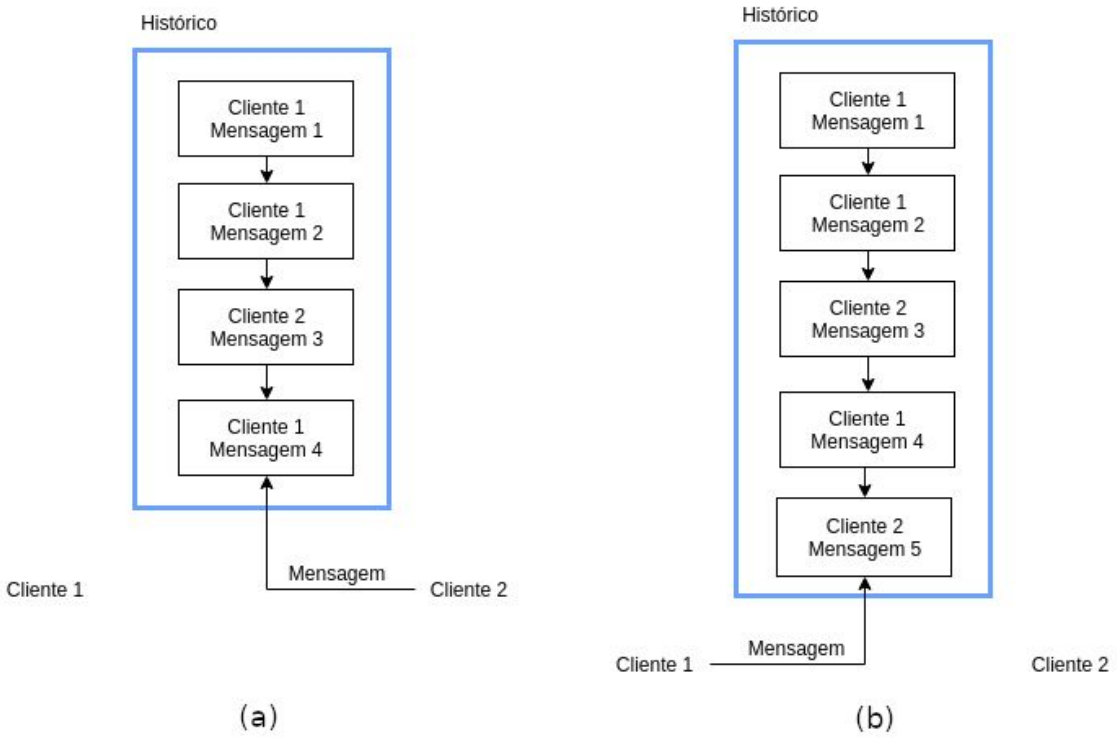

Figura 1. (a) Cliente 2 enviou uma mensagem e será escrita no final do histórico. (b) Cliente 1 enviou uma mensagem e será escrita “abaixo” da mensagem do Cliente 2.

Como a escrita é feita somente no final dessa lista, uma variável de interesse para o controle dessas operações deve ser um ponteiro apontando para o último bloco de mensagem adicionado. Assim, a função que insere o novo bloco recebe como parâmetro esse ponteiro para o final do histórico (chamado de *last*), adiciona um novo bloco e retorna um ponteiro para esse bloco para atualizar a variável *last*. Há outras formas de implementar essa operação, mas essa foi adotada para que que os efeitos da sincronização - ou a falta dela - sejam percebidos mais facilmente.

Além dos clientes, há outra função que executa em uma thread diferente. Essa função, chamada *auditor*, possui como objetivo percorrer todo o histórico e fazer a contagem do número de mensagens enviadas por cada cliente e compará-lo com o número de mensagens que cada cliente efetivamente enviou. Claramente,

discrepâncias entre esses valores evidenciam a falha no funcionamento do sistema e a razão da existência desse *auditor* é somente a de verificar essa falha.

Vale ressaltar que como o foco do laboratório é analisar o comportamento do servidor ao lidar com múltiplos clientes, não foi implementada a interface do cliente. Ao invés disso, implementou-se bots que simulam os clientes. Dessa forma, foi possível testar o servidor em situações extremas onde ocorrem muitas mensagens simultaneamente. Além disso, facilita fazer uma comparação técnica entre diferentes tipos de servidor.

b. Informações técnicas

Para fins de análise, foram implementadas três formas de servidores, todos utilizando a mesma ideia de lista ligada. Um servidor sequencial e dois servidores com threads, onde um possui sistema de sincronização e o outro não. O servidor sequencial se encontra no arquivo *servidor_sequencial.cpp* e os servidores com threads no arquivo *servidor_thread.cpp*. Para ativar ou desativar o sincronismo basta modificar o valor de *SYNCHRONISM.*

A linguagem de programação utilizada foi o C/C++ junto com as bibliotecas *pthread.h* e *semaphore.h* para implementação das threads e semáforos. A fim de tornar o código mais legível optou-se por utilizar o paradigma de programação orientada a objetos (POO). Tanto no código sequencial quanto o de threads foram criadas as classes *Server*, que faz todo o papel do servidor. Abaixo uma breve descrição de seus métodos.

Servidor sequencial:

- *Server::execute_test*: executa o teste de acordo com os valores definidos nas configurações.

Servidor de threads:

- *Server::execute_test*: executa o teste de acordo com os valores definidos nas configurações.
- *Server::infinite_execution*: cria os clientes de acordo com as definições, porém eles enviam mensagem indefinidamente. Além disso cria-se um auditor que apresenta a situação da lista de mensagens.

Obs.: a compilação deve ser realizada utilizando o *g++* com a flag de compilação *-lpthread* .

2. Execução, testes e análises de resultados

a. Execução

Segue abaixo imagens ilustrando o servidor em funcionamento com 2 clientes e 1 auditor, rodando em modo *infinite_execution*.

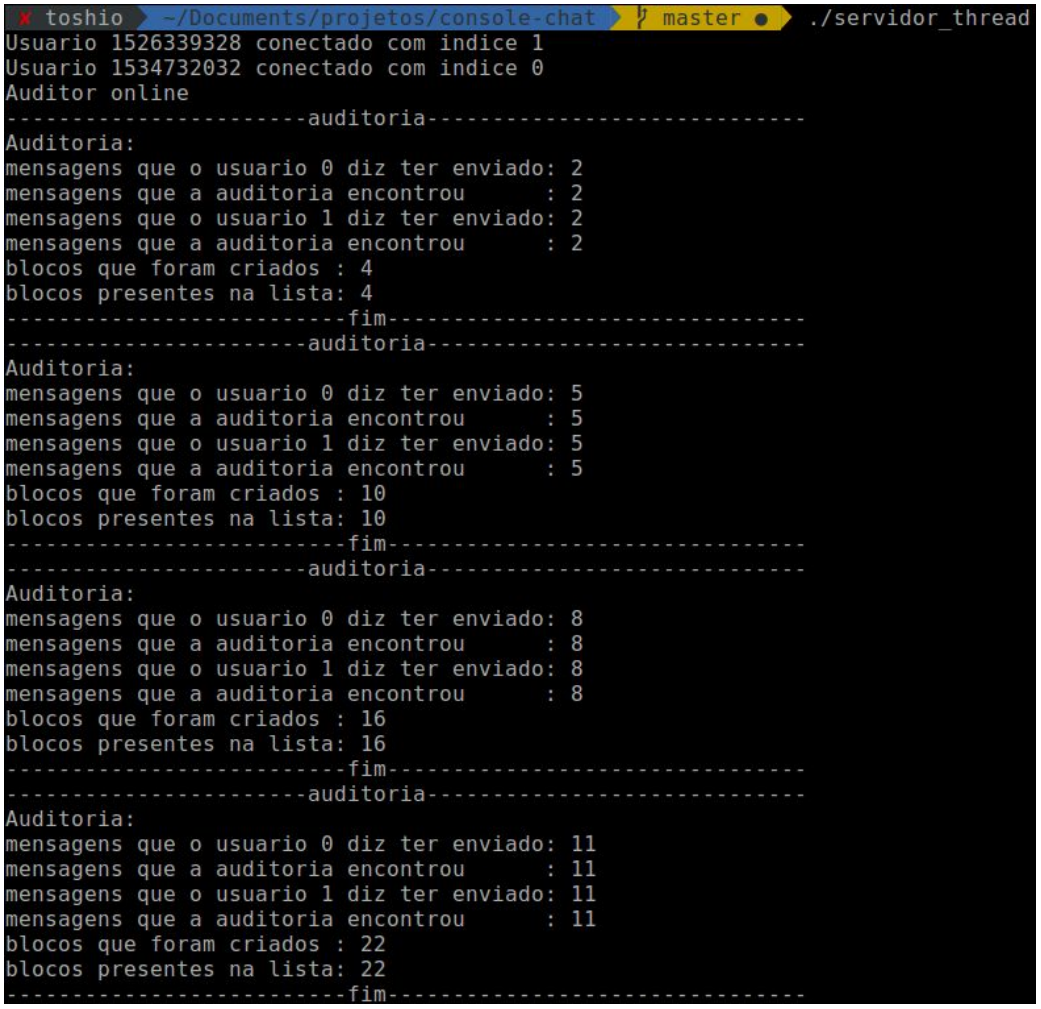

Figura 2. Servidor de threads com sincronização em modo *infinite_execution.*

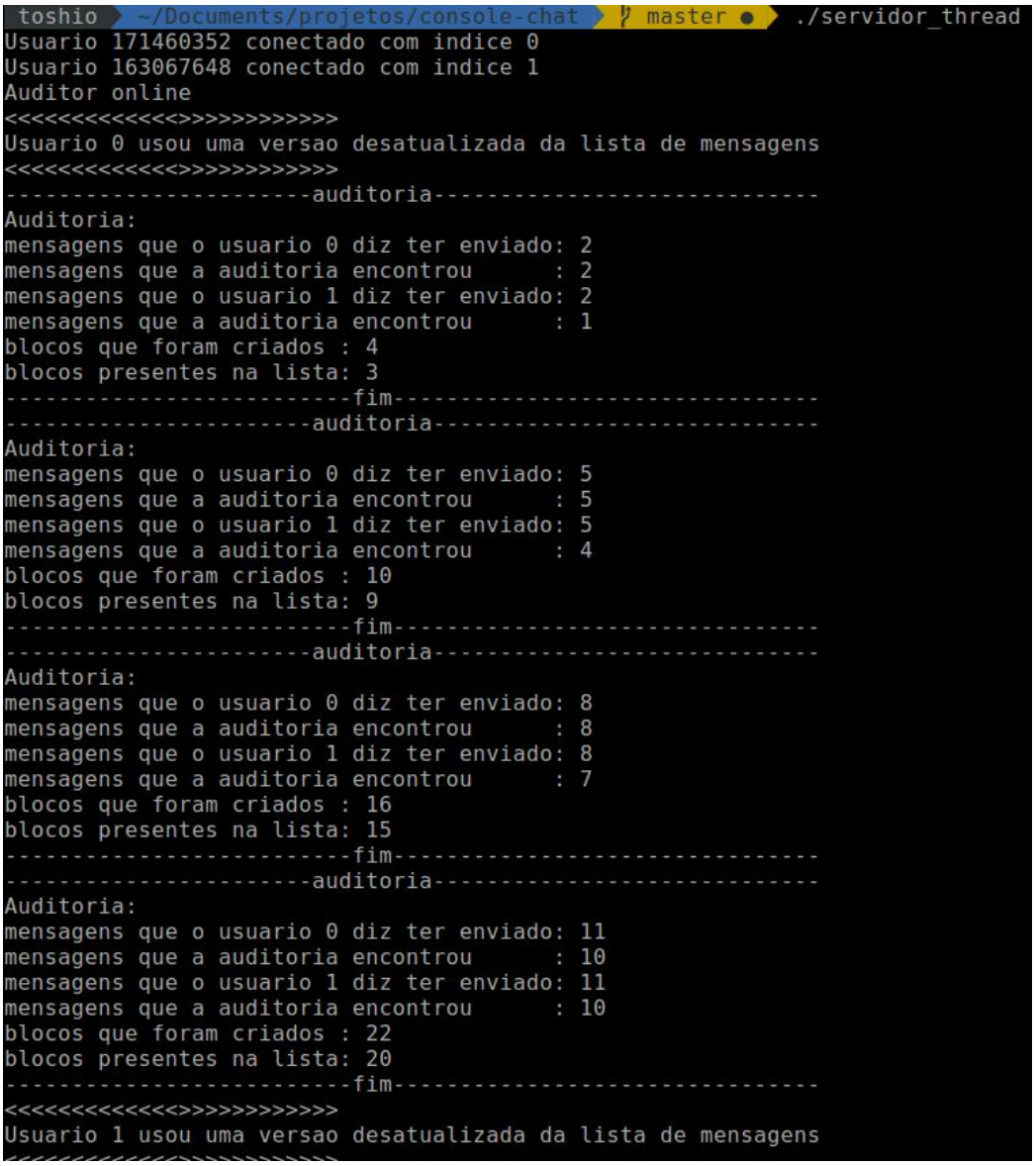

Figura 3. Servidor de threads sem sincronização em modo *infinite_execution.*

Como esperado, a execução do servidor com sincronização não teve nenhuma perda de mensagem. Enquanto isso, no servidor sem sincronização ocorreram a perda de duas mensagens.

b. Testes

Foram implementadas diferentes tipos de servidor, dentre elas: sequencial, threads com sincronização e threads sem sincronização. A fim de fazer uma comparação técnica entre elas, realizou-se um teste onde o servidor teria que lidar com dois clientes que enviaram 100 mensagens com o tempo de 1 segundo entre cada envio. Com isso foi possível obter o resultado apresentado na tabela 1.

Tabela 1. Comparação entre os tipos de servidor.

Tipo de servidor	Mensagem perdidas	Tempo de execução (s)
Sequencial	0	200
Threads sem sincronização	X	100
Threads com sincronização	0	100

Na tabela, X representa um valor incerto. Isso porque como será explicado abaixo, o servidor implementado utilizando threads sem sincronização é extremamente imprevisível. Durante os testes foram obtidos casos nessa situação em que ao final do envio só foram salvos no histórico de conversa uma única mensagem. Enquanto em outros, teve mais de 120 mensagens salvas no histórico. Dessa forma, torna-se inviável fazer uma análise quantitativa para o valor de X, já que sua variância é muito grande.

c. Resultados

A partir das informações obtidas nos testes, pode-se concluir que utilizar threads sem sincronização é impraticável, pois ocorrem muitas falhas ao longo de sua execução. Por outro lado, utilizar servidor de thread com sincronização ou sequencial são funcionais.

Dadas as circunstâncias, onde cada cliente espera um tempo antes de enviar uma mensagem, o desempenho com threads foi bem superior se comparado com o sequencial. Isto porque o tempo de espera no sequencial e somado, enquanto no com threads eles são executados ao mesmo tempo.

Durante a execução do servidor de threads sem sincronização ocorreram muitas falhas que causaram resultados inesperados. Dentre eles destacou-se a perda de algumas mensagens específicas e até a inutilização da função de adição de mensagem a lista. Na seção de análise de resultados será explicado melhor essas falhas.

d. Análise de resultados

A Figura 4 ilustra o que ocorre quando obtêm-se essa falha descrita nos resultados. Na Figura 4(a) há o caso em que os dois clientes tentam escrever ao mesmo tempo e por causa disso a variável *last*, apontando para o final, é enviada para as duas threads com o mesmo valor. O servidor irá criar os dois blocos solicitados, fato ilustrado na Figura 4(b).

Por conta da diferença de tempo entre as execuções, *last* será atualizado pela última thread a terminar, bem como a definição do “pai” deste último bloco. A Figura 4(c) ilustra esse último fato, mostrando que um dos blocos criados não pertence ao histórico e não poderá mais ser recuperado. A Figura 4(d) mostra outro caso possível em que *last* aponta para um dos blocos mas o penúltimo bloco aponta para o outro, encerrando a cadeia. Nesse último caso o histórico não será mais atualizado e toda mensagem nova será adicionada em uma cadeia inacessível.

Esse tipo de situação é comum quando se trabalha com threads e é conhecida como condição de corrida. Trata-se de um erro de sincronismo causado quando dois ou mais threads, no caso os clientes, tentam modificar um mesmo recurso, no caso o histórico de conversa. Na forma como foi implementado essa falha implica no acúmulo de memória alocada que não é utilizada, mas em outras implementações essa mesma condição poderia implicar na sobrescrita de valores, por exemplo.

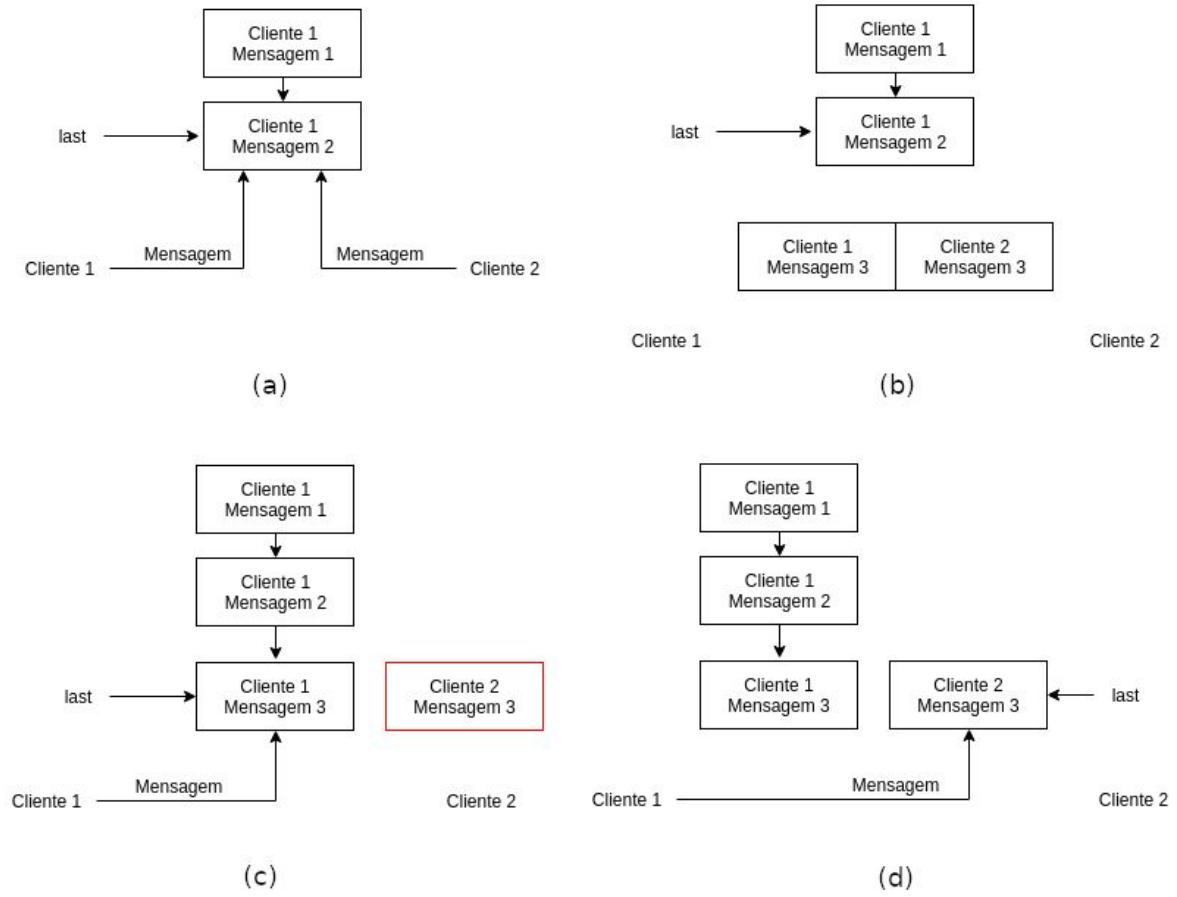

Figura 4. Ilustração do que pode ocorrer quando há concorrência entre as threads de clientes.

3. Limitações encontradas

O chat criado não tem como objetivo ser uma aplicação funcional, mas apenas uma simulação do seu funcionamento para ilustrar a condição de corrida entre as threads. Por conta disso, a aplicação é limitada em alguns aspectos, como a falta de interface para o cliente enviar mensagem e até mesmo o servidor que não mostra as mensagens recebidas.

Nesse último ponto é possível discutir sobre um problema adicional: quando escrever na tela as mensagens recebidas. Parece razoável que a mensagem recebida seja escrita sempre que um novo bloco seja criado, o que nesse caso daria a impressão de que não havia nenhum problema com a aplicação pois todas as mensagens recebidas seriam escritas. Contudo, supondo que a aplicação fosse encerrada e novamente aberta, algo que “forçaria” a recuperação do histórico, nem todas as mensagens apareceriam para os clientes devido ao problema citado.

4. Conclusões

A implementação do sistema de chat permitiu compreender melhor o funcionamento de threads e um dos riscos associados ao seu uso: condição de corrida. Diante desse problema, foi possível entender como o uso da ideia de semáforos é útil para resolver essa falha de sincronização.

Além disso, a comparação entre versão com threads e versão sequencial possibilitou ver que o uso de threads pode ser mais vantajoso desde que seja bem implementado para evitar as falhas de sincronização. A execução simultânea das threads possibilita que cada função possa executar de forma independente compartilhando os recursos disponíveis ao processo e cumprir o objetivo do código enquanto a forma sequencial também pode atender o funcionamento esperado mas possui desempenho inferior.