Universidade Federal do Pará

Instituto de Ciências Exatas e Naturais

Faculdade de Computação

Disciplina: Laboratório de Sistemas Distribuídos

Professor: Dionne Monteiro

Equipe:

Antonio Henrique de Assis de Matos - 202204940016

Daniel Diniz da Silva - 202104940004

Isabel dos Reis Camarão - 202204940028

Relatório - Sockets em Linux

Equipe Java

Nessa atividade, objetivamos implementar uma aplicação que seja capaz de lidar com mais de uma requisição. Para isso, analisou-se possíveis soluções, de forma que o servidor atenda ilimitadas requisições de clientes. Primeiramente, observamos a opção de thread por conexão, em que cada cliente que solicita sua conexão cria uma nova thread para tratar da requisição, e se mantém até o fim da comunicação. Esse modelo tem como ponto positivo um paralelismo inato, haja vista que para cada cliente ocorre uma operação separadamente. Entretanto, esse modelo não é adequado para uma quantidade alta de requisições simultâneas, como indica Harold (2013) ao afirmar que a criação de uma thread para cada cliente tende a se tornar ineficiente à medida que o número de usuários cresce, resultando em queda de desempenho. Portanto, o modelo thread por conexão é ideal para aplicações menores, em que o foco é a simplicidade por não necessitar de tanta robustez.

Além dessa solução, é possível aplicar o conceito de “fork”. Nesse modelo, é criado uma cópia do servidor para dedicar um processo exclusivo ao tratamento de cada nova requisição. Esse modelo tem como ponto positivo a robustez e o isolamento de memória, haja vista que uma falha crítica no atendimento a um cliente não afeta o servidor principal nem as demais conexões. Entretanto, essa arquitetura apresenta um custo computacional elevado, como indica Tanenbaum (2016) ao afirmar que a criação e o gerenciamento de novos processos consomem significativamente mais recursos do sistema do que *threads*, gerando um *overhead* que pode esgotar a memória. Portanto, o modelo de múltiplos processos é ideal para aplicações onde a estabilidade e a segurança são prioritárias, apesar de custar maior consumo de hardware.

Após o estudo desses conceitos, foi realizada a implementação dos modelos de thread e fork em código na linguagem C. Para a abordagem threads, o código estabelece um servidor que, após configurar a rede e entrar em modo de espera, inicia um ciclo contínuo para aceitar conexões. Diferente de uma abordagem sequencial, para cada cliente aceito, o programa aloca dinamicamente um espaço de memória para identificar a sessão e dispara imediatamente um fluxo de execução paralelo. Isso delega a troca de mensagens a uma função separada, permitindo que o processo principal fique livre instantaneamente para atender novas requisições.

Além disso, o algoritmo prioriza a eficiência no gerenciamento de recursos do sistema operacional. Ao iniciar a execução paralela, utiliza-se um mecanismo de desvinculação que permite que a memória e os recursos da *thread* sejam recuperados automaticamente assim que o atendimento ao cliente é finalizado, sem bloquear o servidor. Essa estratégia é fundamental para garantir que o sistema permaneça estável e não sofra com vazamento de memória ou travamentos, mesmo ao processar diversas conexões simultâneas.

Ademais, foi desenvolvido o código em C para o método fork. A implementação realizada adota a estratégia de criar processos independentes para gerenciar múltiplas conexões simultâneas. O servidor mantém um ciclo constante de monitoramento e, ao identificar um novo cliente, gera uma cópia automática de si mesmo. Dessa forma, o processo principal fica livre instantaneamente para aguardar novas solicitações, enquanto a cópia criada assume a responsabilidade total de processar a comunicação com o usuário atual.

Em relação à organização, o código separa claramente as tarefas entre os processos para economizar recursos. A cópia encerra o acesso ao canal de espera geral e dedica-se apenas à troca de mensagens, finalizando sua execução logo após o envio dos dados. Simultaneamente, o processo principal fecha a conexão com o cliente específico, delegando essa função à cópia. Esse isolamento garante que o sistema permaneça estável, pois uma falha no atendimento individual não interfere no funcionamento do servidor central.Seguido da implementação dos algoritmos, foram realizados testes de estresse para cada código. Para a validação experimental, os modelos foram submetidos a cargas de trabalho de 500, 1.000, 5.000 e 10.000 requisições. A análise quantitativa dos tempos de resposta evidencia a superioridade de escalabilidade das *threads*. Nos testes iniciais com 500 conexões, o modelo de *threads* registrou 0,159s, apresentando o dobro de performance em relação ao modelo de *fork* (0,323s). Embora em 1.000 requisições os tempos tenham se mantido próximos (1,20s para *thread* e 1,06s para *fork*), a disparidade se acentuou drasticamente na carga de 5.000 requisições: enquanto o servidor de *threads* processou a demanda em 3,13s, a implementação baseada em processos exigiu 6,98s, demonstrando o alto custo computacional da criação excessiva de processos.

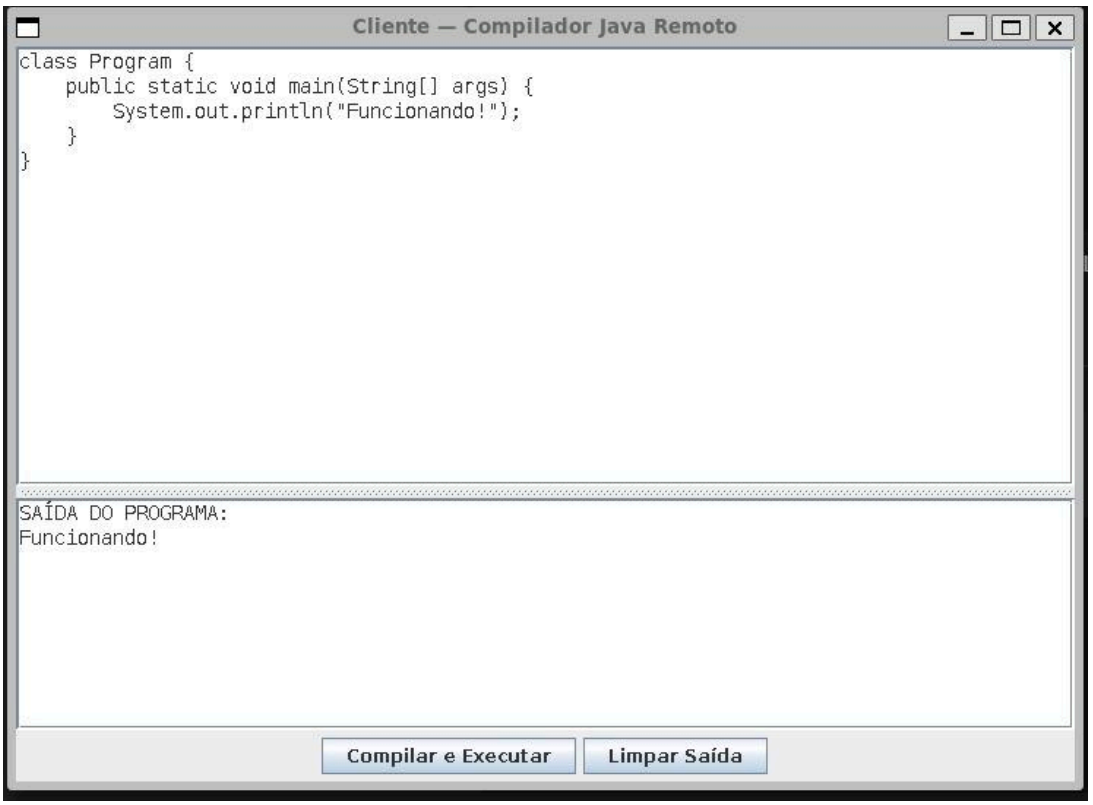
Nº de Requisições	**Tempo - Threads (seg)**	**Tempo - Fork (seg)**
500	0.159048890	0.323964302
1.000	1.200854160	1.062787817
5.000	3.136329570	6.982104613

10.000	7.227046530	Falha

Além da performance temporal, a execução com a carga máxima de 10.000 requisições revelou os limites operacionais de cada arquitetura. Nesse cenário crítico, o servidor de *threads* manteve a estabilidade e concluiu o processamento em 7,22s. Em contrapartida, o modelo de *fork* entrou em colapso e falhou antes de concluir a operação, indicando uma exaustão dos recursos do sistema operacional, como memória ou descritores de processos. Esses dados comprovam que, embora funcional para baixas demandas, o modelo de processos torna-se inviável em cenários de alta concorrência, onde o modelo de *threads* demonstra maior robustez. Entretanto, é relevante considerar que os resultados podem variar de acordo com as especificações de hardware de onde são testados.

A partir da implementação em C e os testes de estresse, foi desenvolvido um servidor, para compilação, execução e retorno de erros e saída de programas, em Java. Para isso, foi desenvolvida uma arquitetura Cliente-Servidor dividida em dois componentes principais. O Servidor, projetado para operar na porta 5000, utiliza o conceito de *multithreading* para garantir que múltiplos usuários possam compilar seus códigos simultaneamente. O fluxo de processamento consiste em receber o código-fonte enviado pela rede, salvá-lo temporariamente em disco como Program.java e invocar o compilador do sistema operacional através de comandos de execução externa. O servidor é capaz de capturar tanto os fluxos de erro de compilação quanto a saída padrão da execução, devolvendo ao cliente o resultado exato do processamento ou as mensagens de falha.

Por fim, foi implementado um programa com interface para executar as requisições do cliente. Pelo lado do usuário, o Cliente foi construído com uma interface gráfica baseada na biblioteca Swing, oferecendo um ambiente amigável com áreas distintas para edição de código e visualização de respostas. A comunicação ocorre via *socket TCP*, onde o cliente envia o texto do programa linha a linha, seguido de um marcador especial (__END__) que sinaliza ao servidor o fim da transmissão. Após o envio, a interface aguarda a resposta processada pelo servidor e a exibe na tela, permitindo que o usuário visualize se o seu código foi compilado e executado com sucesso ou se precisa de correções.



Referências

TANENBAUM, Andrew S.; BOS, Herbert. *Sistemas Operacionais Modernos*. 4. ed. São Paulo: Pearson, 2016.

Harold, E. R. *Java Network Programming.* 4. ed. O'Reilly Media, 2013.