INF01151 - SISTEMAS OPERACIONAIS II N - Turma B - 2022/2

Universidade Federal do Rio Grande do Sul - UFRGS

Professor: Weverton Cordeiro

Brenda Streit Schussler - 00325353 - brendaschussler@gmail.com

Pedro Henrique Casarotto Rigon - 00325358 - pedrohenriquecasarotto@hotmail.com

Pedro Henrique Capp Kopper - 00314142 - phckopper@inf.ufrgs.br

Nikolas Cunha Chemale - 00288491 - nikolascunha123@gmail.com

Relatório Trabalho Prático Parte 1: Threads, Sincronização e Comunicação

- **Como foi implementado cada subserviço:**

Subserviço de Descoberta (Discovery Subservice): O subserviço de descoberta foi implementado usando a comunicação UDP Broadcast entre elementos de uma mesma rede. Dessa forma, o usuário que entrar como participante ficará responsável por enviar mensagens do tipo “DISCOVERY_TYPE” a cada 1 segundo contendo suas informações de *hostname, ip_address, mac_address* e *status*, e ficar aguardando uma resposta do tipo “CONFIRMED_TYPE” de seu Manager, na porta 4001. Por outro lado, o manager inicia o seu serviço de descoberta aguardando por mensagens UDP Broadcast do tipo “DISCOVERY_TYPE” na porta 4000, enviadas pelos participantes. Ao recebê-las, o manager adiciona as informações do participante na tabela de participantes incluindo um campo extra denominado, *time_control,* que será explicado posteriormente. Após cadastrar o participante, o Manager envia uma mensagem de confirmação do tipo “CONFIRMED_TYPE”, na porta 4001, para o endereço de ip que enviou a mensagem broadcast. Essa

mensagem contém as suas informações pessoais para que o participante saiba quem é o seu líder. Vale ressaltar que esse processo de envio de mensagens de descoberta é contínuo e ocorre a cada 1 segundo, mesmo que o líder já tenha sido reconhecido. Isso serve para que na próxima etapa do trabalho possamos introduzir um sistema de eleição e eventualmente mais de um líder por aplicação.

Subserviço de Monitoração (Monitoring Subservice): O serviço de monitoramento funciona de forma semelhante ao serviço de descoberta, no entanto há uma alteração entre os papéis ativos e passivos. Desse modo, no sub serviço de monitoração, o manager é responsável por envios de mensagem, através da porta 4002, do tipo “SLEEP_STATUS_TYPE” e após isso aguardará uma resposta dos participantes contendo as informações atualizadas. Essa resposta será do tipo “CONFIRMED_STATUS_TYPE”, através da porta 4003, e o mesmo usará funções implementadas no subserviço de Gerenciamento para atualizar os dados do participante na tabela de participantes. Já os participantes ficarão aguardando mensagens do tipo “SLEEP_STATUS_TYPE” na porta 4002 e ao recebê-la enviarão uma resposta contendo suas informações. Essa mensagem será do tipo “CONFIRMED_STATUS_TYPE” através da porta 4003, para o ip específico do manager.

Como foi anteriormente mencionado, há um campo extra na tabela de participantes, denominado *time_control.* Trata-se de um contador que é responsável por identificar quantos segundos o participante tem para enviar uma nova mensagem de monitoramento ou de descoberta, e é utilizado para verificar se o participante está ativo ou em modo sono (suspenso), e atualizar seu status corretamente. Ou seja, todos os participantes iniciam com o valor default de contador igual a 5, e a cada vez que enviam uma mensagem do tipo “DISCOVERY_TYPE” ou “CONFIRMED_STATUS_TYPE” esse valor é resetado, ganhando assim, mais 5 segundos para enviar uma nova resposta. A cada um

segundo é iterado por toda tabela de participantes decrementando em uma unidade, de cada participante, a variável *time_control* de modo que, se ela atingir o valor zero, é feita uma lista dos participantes que estão zerados, indicando que estes estão suspensos. Isto é, cada participante tem 5 segundos para enviar outra mensagem informando seu status ou será considerado inativo e seu status será alterado. Além disso, é nesse serviço que ocorre a remoção de participantes da tabela de participantes quando for o caso.

Subserviço de Gerenciamento (Management Subservice): O sub serviço de Gerenciamento controla o acesso à estrutura de participantes e possui os métodos responsáveis por alterar essa estrutura. Por se tratar de uma região da memória na qual as threads concorrem pelo acesso, foi necessário tomar os devidos cuidados relacionados com o paradigma de programação concorrente, com uso de diretivas para garantir a exclusão mútua dos métodos de alteração e leitura da tabela de participantes. Vale ressaltar que a estruturação do código foi realizada de forma a tornar idempotente a função que escreve na tabela de participantes. Para isso foi necessário uma função que procura se as informações do participante já se encontram na tabela e nesse caso apenas atualiza as informações. Isso é útil, uma vez que, foi implementado um *at least once* na comunicação entre os participantes distribuídos, pois o participante envia mensagens constantemente ao manager e o manager responde com suas informações. Logo, o participante continua o envio mesmo após receber o “ACK” de confirmação. Isto é feito como uma forma de facilitar, posteriormente, a implementação de múltiplos managers e o processo de eleição de um manager.

Subserviço de Interface (Interface Subservice): Decidimos implementar uma thread que é responsável exclusivamente pela interação com o usuário. Isso se deu devido ao fato do terminal ser um recurso compartilhado e ser necessário controlar o acesso a ele, a fim de evitar competição no acesso aos seus recursos. Para

garantir essa sincronização, a thread espera um semáforo que é disparado pelas threads que manipulam os dados. Assim que é notificado, o serviço atualiza a interface com base nos dados daquele instante e volta a esperar no semáforo. Além disso, a atualização também pode ser disparada ao pressionar a tecla C, entrando em modo comando. Nesse caso, a interface para de atualizar automaticamente e disponibiliza um campo para que o usuário digite o comando desejado. Caso seja um participante e ele receba um EXIT, envia uma mensagem de "PROGRAM_LEAVE" para o manager, indicando que quer ser removido da lista de participantes. Caso seja um manager que realiza o comando de WAKEUP <*hostname*>, é enviado um pacote de Wake-On-Lan para o hostname indicado. Ao receber um EOF ou rodar um comando com sucesso, a thread volta a atualizar automaticamente os dados.

- **Em quais áreas do código foi necessário garantir sincronização no acesso aos dados:**

O acesso crítico desse programa se encontra nas concorrentes leituras e escritas na tabela de participantes, uma vez que diversas threads concorrem pelo acesso à métodos do sub serviço de gerenciamento. A implementação dos métodos da estrutura de dados levou em conta a possibilidade de acesso concorrente, portanto, utilizamos mutex em todos os métodos inclusive no de print, uma vez que para printar é necessário que haja a coerência dos dados. O programa da forma que foi feito, valida as operações anteriores e faz uso de comparações para tornar as operações idempotentes, logo, tivemos que utilizar mutex em todos os métodos para garantir que apenas uma thread alterasse ou obtivesse informações sobre a estrutura de dados para não gerar falsos positivos nem sobrescrita de estruturas.

A sincronização na função de identificar participantes inativos tem uma particularidade quanto ao uso dos mutex. Portanto, o uso de mutex nessa função

foi necessário onde era obtido os participantes inativos, porém, antes de chamar a função para alterar os status dos participantes inativos, era necessário liberar o lock permitindo transferir a responsabilidade de gerir a fila de threads ao sistema operacional, uma vez que o método *remove* também faz uso de mutex. Ou seja, se fosse feita a chamada da função *remove* sem liberar o lock anteriormente adquirido, o programa iria tentar o acesso a uma seção crítica sem liberar a seção crítica na qual estava executando, resultando em um deadlock.

Além disso, no serviço de interface, na codificação do participante, é necessário garantir a sincronização quando um programa encerra para que o mesmo envie uma mensagem solicitando sua remoção da tabela e posteriormente a desconexão do programa. A interface do usuário deve printar dados consistentes, por isso fizemos uso de um semáforo binário entre os métodos da interface para garantir a exclusão mútua na apresentação das informações ao usuário.

- **Descrição das principais estruturas e funções que a equipe implementou;**

O nosso sistema, de forma simplificada, trata-se de uma constante comunicação entre elementos de uma mesma rede executando o mesmo programa. Sendo que, a depender do tipo de usuário (participante ou manager) ele irá consumir funções específicas para o papel que deve desempenhar. Portanto, a diferença entre um manager e um participante são as funções que eles consomem por meio das threads criadas e os papéis que eles desempenham na aplicação. Sendo o manager um gestor e o participante uma entidade, no geral, mais passiva. O programa possui duas estruturas de dados principais, sendo elas:

1. Estrutura Participant: No manager ela armazena todos os participantes da rede na qual é possível interagir e enviar comandos específicos. Nos participantes, é usada para armazenar as suas próprias informações

pessoais, facilitando a atualização de suas informações e o acesso entre as threads do programa.

2. Estrutura Packet: Estrutura na qual é realizada o envio de mensagens entre os participantes e o manager. Também foi implementado parcialmente a utilização de um padrão union permitindo o envio de estruturas de dados, como por exemplo a Participante, através do seu payload. No entanto, essa alteração estará 100% concluída apenas na fase 2 do trabalho, embora esteja em estado avançado de implementação.

Principais funções implementadas:

- **add_participant**: Adiciona o participante na tabela de participantes do manager se o mesmo não estiver na tabela. Caso contrário as informações dele são atualizadas e *time_control* é “resetado”. A operação é feita de forma idempotente porque dados duplicados são permitidos pois não geram informações duplicadas, apenas atualização dos dados.
- **check_asleep_participant**: Função que verifica a cada 1 segundo se o participante está se comunicando com o sistema através do envio de mensagens. Para fazer esse controle ele utiliza um decrementador da variável *time_control* (toda vez que o participante se comunica com o manager essa variável é resetada para o valor default 5, que significa algo como “você tem 5 segundos para mandar uma nova mensagem ou seu status será atualizado para *asleep*”. Dessa forma, a função check_asleep_participant decrementa o *time_control* de todos os participantes e verifica se algum é igual a 0, caso positivo atualiza os status para asleep.
- **listen_discovery**: Manager consome essa função através de uma thread criada que fica aguardando por mensagens do tipo “DISCOVERY_TYPE” e

respondendo através do envio de mensagens do tipo “CONFIRMED_TYPE” para o ip do participante remetente.

- **manager_start_monitoring_service**: Realiza o controle dos usuários acordados e dormindos por meio do envio de mensagens do tipo “SLEEP STATUS TYPE” e aguarda a mensagens do tipo “CONFIRMED_STATUS_TYPE” para atualizar a variável de controle time_control. O tratamento da variável de controle (time_control), como já explicado, é feito através de decrementos e testes periódicos do seu valor.
- **listen_Confirmed_monitoring**: Mensagem de confirmação enviada pelo participante é recebida pelo manager para confirmar que está acordado e respondendo a comandos.
- **sig_handler**: Realiza o tratamento dos sinais recebidos, realizando a saída graciosa e descadastramento do manager em caso de interrupção ou encerramento do programa.
- **exit_control**: Controla o decremento periódico da variável de controle “time_control” responsável por identificar participantes dormindo.
- **user_interface_manager_thread**: Função consumida por thread que controla a interface do manager.
- **listen_Confirmed**: Função executada pelo participante responsável por receber e tratar mensagens do tipo “CONFIRMED_STATUS_TYPE”.
- **participant_start**: Função responsável por inicializar algumas informações de um participante e iniciar a chama da função de envio de mensagens de descoberta.
- **listen_monitoring**: Função responsável por receber e tratar mensagens do manager de monitoramento.
- **user_interface_participant_thread**: Função responsável por controlar a interface do participante.

- **Explicar o uso das diferentes primitivas de comunicação:**

Foi utilizado neste programa a criação de threads que consomem funções específicas e a troca de mensagens utilizando o mecanismo UDP configurado para enviar via Broadcast. Além disso, foram utilizadas primitivas para garantir a exclusão mútua através de mutex e semáforos, principalmente na região de acesso concorrente, que é a tabela de participantes.

Para a sincronização da interface do usuário, foi utilizado um semáforo, que permite que as threads dos serviços de descoberta e gerenciamento notifiquem que uma atualização deve ser exibida na tela.

Por fim, utilizamos a interface de *signals* disponibilizada pelo Linux para tratar os sinais SIGINT e SIGTERM, que permitem que o participante saia graciosamente da aplicação quando encerrado via shell ou morto, respectivamente.

- **Descrição dos problemas que a equipe encontrou durante a implementação e como estes foram resolvidos (ou não):**

Encontramos problemas para implementar a atualização dos status dos participantes inativos da lista de participantes. Considerando que a função de remover participante que estava sendo chamada requer acesso aos dados da tabela de participantes que estavam sendo editados pela função que a chamava, ocorria um deadlock e o programa não executava corretamente. Esse problema foi resolvido utilizando mutex para a edição dos dados na função, que foi liberado logo após essas edições e antes da chamada da função de remover participantes, que também possui um mutex em sua implementação.

Como objetivo estendido, também nos propusemos a implementar a serialização do payload através de Protocol Buffers, conforme visto em aula. Apesar

de termos conseguido realizar a implementação funcional de algumas mensagens, optamos por deixar essa melhoria para a segunda parte do trabalho, visto que ela demandará uma reescrita substancial do código de rede.

Tutorial de como Rodar o programa WakeUP_Computer:

Para a compilação e execução elaboramos um script .sh para automatizar as tarefas de compilação e execução. Aqui está o tutorial de como rodar o programa via script localizado no arquivo ZIP em anexo e como nome “run_program.sh”:

1. Abra o terminal no Ubuntu (pressionando "Ctrl + Alt + T" no teclado).
2. Navegue até a pasta onde o script "run_program.sh" está salvo.
3. Verifique se o script "run_program.sh" tem permissão para ser executado. Se você ainda não concedeu permissão, use o seguinte comando para dar permissão de execução:

   chmod +x run_program.sh

4. Agora você pode executar o script com ou sem o parâmetro "manager". Para executar o script como participante, basta digitar o seguinte comando:

   ./run_program.sh

5. Se você quiser executar o script como "manager", basta digitar o seguinte comando:

   ./run_program.sh manager