INF01151 – Sistemas Operacionais II N
Prof. Weverton Cordeiro

Integrantes:

Eduardo André Leite | Matrícula: 00287684.

Henrique Borges Manzke | Matrícula: 00326970.

Pedro Arejano Scheunemann | Matrícula: 00335768.

1. Introdução.

Este relatório apresenta a implementação do projeto de "Gerenciamento de Sono" de estações de trabalho em um mesmo segmento de rede física. O objetivo deste serviço é garantir que as estações de trabalho, ao serem colocadas em modo de sono, ainda permitam que os colaboradores possam acordá-las remotamente para acessar serviços específicos, através de uma mesma estação líder.

A proposta do serviço se baseia no padrão de rede Wake-On-LAN (WoL), o que possibilita que as estações de trabalho sejam acordadas remotamente a partir do envio de um "pacote mágico" para a placa de rede da estação. Para implementar este serviço, o dividimos em três subserviços: descoberta, monitoramento e interface.

2. Implementação dos subserviços.

A depender da entrada do usuário, o programa tem 2 caminhos:

1. O que será executado caso iniciarmos a estação como manager, através de ./sleep_server manager.
2. O que será executado caso iniciarmos a estação como participante, através de ./sleep_server.

Tanto no fluxo de execução do manager quanto no do participante, criamos três threads, cada uma para lidar com um subserviço: descoberta, monitoramento e interface. A criação dessas threads, por parte do manager e do participante, estão nas funções guest_program() e manager_program(), respectivamente.

```
// Funções do participante
static void guest_program();
void* guest_discovery_service(void *arg);
void* guest_monitoring_service(void *arg);
void* guest_interface_service(void *arg);
```

```
// Funções do manager
static void manager_program();
void* manager_discovery_service(void *arg);
void* manager_monitoring_service(void *arg);
void* manager_interface_service(void *arg);
```

É interessante acompanhar a explicação de cada subserviço junto com o código, na função correspondente ao subserviço e ao tipo de estação.

2.1. Subserviço de descoberta.

A estação manager recebe e responde pacotes do tipo SLEEP_SERVICE_DISCOVERY que chegaram através de broadcast na porta PORT_DISCOVERY_SERVICE.

Aqui, ela atua como servidor. Depois de criar o descritor do socket e habilitar o recebimento de mensagens por broadcast, bindamos o socket na porta PORT_DISCOVERY_SERVICE e no IP INADDR_ANY (0.0.0.0). Por fim, temos um loop para esperar receber mensagens de participantes nesse endereço, e enviar de volta uma mensagem de confirmação. Os participantes descobertos são adicionados na tabela de participantes, e a tabela atualizada é printada no console. A tabela é uma variável compartilhada entre as threads do manager na memória, portanto, garantimos o acesso exclusivo a ela através de mutex.

A estação participante envia pacotes do tipo SLEEP_SERVICE_DISCOVERY para as outras estações por broadcast nas portas PORT_DISCOVERY_SERVICE e espera ser respondida pelo manager.

Aqui, ela atua como cliente. Depois de criar o descritor do socket e habilitar o envio de mensagens por broadcast, setamos a opção de timeout para 4 segundos. Temos um loop para enviar mensagens para todas as estações da rede nas portas PORT_DISCOVERY_SERVICE por broadcast, por isso o IP INADDR_BROADCAST (255.255.255.255), e esperar de volta uma mensagem de confirmação, por 4 segundos (timeout). Se essa confirmação não chega, enviamos de volta a mensagem por broadcast e esperamos novamente. Esse loop se repete até chegar a confirmação. O timeout aqui é importante para a estação manager poder executar o programa após uma estação participante já ter executado, e mesmo assim elas se reconhecerem.

2.2. Subserviço de monitoramento.

A estação participante recebe e responde pacotes do tipo SLEEP_STATUS_REQUEST que chegaram na porta PORT_DISCOVERY_SERVICE.

Aqui, ela atua como servidor. Depois de criar o descritor do socket e habilitar a opção REUSEADDR, bindamos o socket na porta PORT_MONITORING_SERVICE e no IP INADDR_ANY (0.0.0.0). Por fim, temos um loop para esperar receber mensagens do manager nesse endereço, e enviar de volta uma mensagem do tipo SLEEP_STATUS_REQUEST, sendo que a resposta enviada: ou é avisando que quer sair (STATUS_QUIT), caso o usuário tenha digitado EXIT no terminal, ou avisando que está acordada (STATUS_AWAKE). Na explicação da interface dará para entender melhor esta lógica.

A estação manager envia pacotes do tipo SLEEP_STATUS_REQUEST para as estações já conhecidas e processa a resposta.

Aqui, ela atua como cliente. Depois de criar o descritor do socket e habilitar a opção REUSEADDR, setamos a opção de timeout para 4 segundos. Temos um loop para enviar mensagens para todas as estações participantes que foram adicionadas na tabela, utilizando as informações armazenadas dos seus IPs e a porta convencionada PORT_MONITORING_SERVICE para enviar uma mensagem do tipo SLEEP_STATUS_REQUEST nesses endereços. Para cada participante, esperamos 4 segundos para receber a resposta (timeout). Se a resposta não chegar, atualizamos a entrada do participante na tabela para “asleep”. Se chegar, e for STATUS_AWAKE, atualizamos a entrada para “awake”, mas se for STATUS_QUIT, removemos o participante da tabela. Aqui, é garantido que a tabela só vai ser printada no console se houver atualização, caso contrário, não precisa printar de novo. Toda operação que envolve a tabela foi protegida com o uso de mutexes, para garantir o acesso exclusivo e evitar problemas de concorrência.

2.3. Subserviço de interface.

Na estação participante, é um serviço simples para verificar se o usuário digitou EXIT no terminal. Caso sim, seta uma flag que indica que ele deseja sair da tabela armazenada pelo manager. Essa flag é lida a todo tempo pelo subserviço de monitoramento do participante e indica se a mensagem de resposta será STATUS_EXIT ou STATUS_AWAKE, e por isso, aqui, também garantimos acesso exclusivo a ela através de mutex.

Na estação manager, é também um serviço simples para verificar se o usuário digitou WAKEUP <hostname> no terminal. Caso sim, verifica se o hostname está contido na tabela armazenada. Se tiver, envia um pacote WOL para o endereço MAC armazenado que corresponde a ele. Perceba: os outros subserviços que são responsáveis por printar a tabela de participantes no terminal, apenas quando esta for atualizada. Essa foi nossa ideia final e será melhor explicada no relato de problemas que encontramos.

3. Breve descrição das funções e estruturas auxiliares implementadas.

Como na comunicação UDP é necessário ter uma convenção entre cliente-servidor sobre o tamanho e tipo dos dados enviados, criamos uma estrutura para facilitar essa questão, e representar os pacotes de dados enviados, com dois campos: valor inteiro que indica o tipo do pacote (SLEEP_SERVICE_DISCOVERY ou SLEEP_STATUS_REQUEST), e um array de caracteres com 256 posições que representa o conteúdo do pacote, uma mensagem.

```
struct packet {
    uint16_t type;
    char payload[256];
};
```

Para representar as informações de um participante na rede, criamos uma estrutura contendo os campos pertinentes e que serão armazenados na tabela de participantes (nome de host, endereço MAC, endereço IP e status).

```
struct guest{
    string hostname;
    string mac_address;
    string ip_address;
    string status;
};
```

Para lidar de forma eficiente com a tabela de participantes e criar uma abstração na manipulação dos dados, criamos uma classe que oferece funções para adicionar e deletar participante, obter e atualizar status, retornar uma lista com todos IPs dos participantes, retornar o endereço MAC de um participante específico e printar a tabela.

```
class guestTable{

    public:
        guestTable(){}

        map<string, guest> guestList;

        void addGuest(guest g)
        {
        void deleteGuest(string ip_address)
        {
        void guestSlept(string ip_address)
        {
        void wakeGuest(string ip_address)
        {
        string getGuestStatus(string ip_address)
        {
        vector<string> returnGuestsIpAddressList()
        {
        string returnGuestMacAddress(string hostname)
        {
        void printTable() {
};
```

Além dessas estruturas e classes, criamos algumas funções auxiliares para dividirmos melhor as responsabilidades do programa, como:

– As funções, get_host_name(), get_mac_address() e get_ip_address(), que são responsáveis por coletar as informações necessárias da estação local.

– A função pack_discovery_sending_message() que apenas concatena o retorno destas três funções, empacotando a mensagem que o participante enviará por broadcast para ser recebida pela estação manager.

– A função parse_payload(), que faz o inverso. É usada pela estação manager, e serve para desempacotar o payload vindo de uma estação participante, decompondo em 3 strings (hostname, endereço MAC e endereço IP) e retornando na estrutura de um participante.

– A função send_wake_on_lan_packet() que envia um pacote mágico WOL para o endereço MAC recebido como argumento.

```
string get_host_name()
{

string get_mac_address()
{

string get_ip_address()
{

string pack_discovery_sending_message()
{

guest parse_payload(char* payload)
{

void send_wake_on_lan_packet(string mac_address)
{
```

4. Relato dos problemas encontrados durante a implementação e como estes foram resolvidos.

A parte realmente complicada do trabalho foi o começo, nenhum de nossos membros tinham experiência com trabalhos similares. Foi necessário aprofundar nosso conhecimento do conteúdo da disciplina e procurar os recursos necessários fora dela para uma implementação em C++ da especificação. A maior parte do tempo perdido foi estudando como implementar sockets UDP e a comunicação entre eles.

Demoramos para dar partida ao desenvolvimento. Felizmente, após o início não demorou tanto para termos algo bom em mãos e podermos testarmos. Durante a implementação, não encontramos nenhum problema sério que tenha nos custado tanto tempo. Nossos problemas incluíram:

1 - Ter confundido o endereço de localhost com o endereço de ip na função que pega o endereço da máquina. E em consequência disso, desenvolvemos código desnecessário que usava portas diferentes para cada máquina que foi posteriormente removido, por uma confusão de conceitos.

2 - Permitir que o usuário consiga interagir com a interface apesar das constantes atualizações da lista de participantes na estação manager feitas pelos subserviços de descoberta e monitoramento. Como usamos o mesmo terminal para ler entrada do usuário e printar a tabela, é difícil sincronizar as operações para ter um resultado perfeito, já que a cada vez que a tabela for printada, o console é limpo para reescrever, e se o usuário está digitando algo nesse momento, o conteúdo é perdido. Não podemos garantir o acesso exclusivo de nenhuma dessas operações ao terminal, já que: ou a tabela não vai ser atualizada em real-time, ou o usuário não vai conseguir digitar nada. Não chegamos a implementar uma solução sem falhas, o resultado final foi reduzir a frequência de prints da lista para apenas quando realmente houver atualizações (ou um participante adicionado, ou um status alterado), e deixar o usuário digitar concorrentemente. Com bad timing, ainda pode acontecer da entrada do usuário ser apagada antes de ser enviada, por acontecer uma atualização da tabela no mesmo momento.