Documentação TP2 - Redes de Sensores Sem Fio

Leonardo Pessoa Miranda - Matrícula 2021039999

Introdução

Este projeto tem como objetivo simular, de forma simplificada, o comportamento de uma Rede de Sensores Sem Fio (RSSF). Em uma RSSF, diversos dispositivos distribuídos em um ambiente são capazes de coletar informações (como temperatura ou umidade) e transmiti-las, sem a necessidade de fios, para outros dispositivos ou para um servidor central. Esse tipo de rede é amplamente utilizado em aplicações de monitoramento ambiental, automação industrial, sistemas de smart home e até soluções biomédicas.

Neste trabalho, a rede simulada possui três possíveis tipos de sensores: temperatura, umidade ou qualidade do ar. Cada sensor é executado como um cliente em C, responsável por captar sua própria medição (inicialmente gerada de forma aleatória) e enviá-la periodicamente a um servidor. O servidor, também implementado em C, recebe essas leituras e as repassa somente aos sensores do mesmo tipo (implementando um modelo de publish/subscribe). Dessa forma, é como se existissem "tópicos": um tópico "*temperature*", outro "*humidity*" e outro "*air_quality*"; cada cliente assina apenas o tópico que corresponde ao seu tipo.

Além do envio periódico das medições, cada sensor escuta as leituras dos demais sensores do mesmo tipo. Ao receber uma nova medição, o sensor verifica sua distância em relação ao emissor e, se o emissor estiver entre os três mais próximos, recalcula o próprio valor medido por meio de uma fórmula de correção que leva em conta a diferença entre as medições e a distância entre os dois dispositivos. Essa troca contínua faz com que as leituras se ajustem gradualmente, simulando situações em que sensores próximos tendem a convergir para valores parecidos. Tudo isso acontece em um "grid" de 10x10 células, onde cada sensor tem uma coordenada fixa (x,y).

Para que o sistema seja robusto, o servidor utiliza multithreading: a cada conexão de um novo sensor, ele cria uma thread que trata especificamente daquela conexão. Isso permite suportar múltiplos clientes simultâneos sem travar a execução. Caso um cliente seja finalizado, o servidor detecta e avisa os demais clientes que aquele sensor saiu de operação, disparando uma medição "-1.0000". Então, cada cliente remove aquele sensor de sua base de dados e registra esse fato como action: removed.

Em linhas gerais, portanto, a solução funciona como segue:

1. Cliente (sensor) é inicializado com parâmetros de execução e gera a primeira medição, enviando-a ao servidor.
2. O servidor recebe a medição, registra-a em sua lista de sensores ativos e a encaminha em broadcast para todos os sensores do mesmo tipo.
3. Os demais sensores, ao receber essa nova medição, decidem se devem corrigir sua própria leitura usando a fórmula fornecida.
4. Ciclicamente, cada sensor envia novas medições (a cada 5, 7 ou 10 segundos, dependendo do tipo).
5. Caso o usuário encerre um sensor, o servidor notifica o restante da rede sobre a saída deste sensor, para que ninguém tente atualizar leituras ou considerar como vizinho um sensor que já não está mais ativo.

Desafios, Dificuldades e Imprevistos do Projeto

Durante o desenvolvimento, um dos primeiros desafios enfrentados foi garantir a correta inicialização do cliente com tratamento de erros em uma ordem de prioridade específica. O enunciado define claramente como proceder se faltam argumentos, se o parâmetro -type estiver ausente, se o tipo fornecido for inválido, se o parâmetro -coords estiver errado ou se as coordenadas estiverem fora do intervalo [0..9]. Foi necessário, portanto, estruturar o código de forma que, logo ao iniciar o cliente, cada uma dessas verificações fosse executada passo a passo, exibindo a mensagem de erro correspondente e o “Usage” na linha imediatamente seguinte. A dificuldade principal foi lembrar de interromper a execução do cliente tão logo fosse identificada a primeira falha, sem prosseguir para validações posteriores.

Outro ponto sensível envolveu o gerenciamento de múltiplas conexões e sincronização de dados no servidor. Como são criadas diversas threads, cada qual manipulando a lista de sensores ativos, precisou-se implementar um controle de acesso concorrente para evitar inconsistências. Utilizamos mutexes (travas de exclusão mútua) que são adquiridas antes de qualquer leitura ou escrita na lista de sensores e liberadas ao término da operação. Essa estratégia resolveu problemas de “race conditions” que surgiram em testes iniciais, onde duas threads tentavam atualizar a mesma estrutura simultaneamente, resultando em valores corrompidos ou travamentos.

A descoberta dinâmica de novos sensores se mostrou igualmente desafiadora. Conforme um sensor é iniciado, ele passa a enviar medições que chegam a todos os clientes do mesmo tipo. Para cada sensor remoto, o cliente local armazena *(x,y)* e a última medição conhecida em uma lista interna, mas só utiliza efetivamente as leituras

dos três mais próximos para corrigir sua própria medição. Descobrir se aquele sensor remoto recém-chegado pertence ao “top 3” exige o cálculo da distância euclidiana e a comparação com os demais sensores conhecidos. Em versões iniciais, a dificuldade era remover sensores “caídos” (que saíram do sistema) ou sensoriar corretamente a influência de sensores muito distantes. As soluções adotadas envolveram verificar sempre se *measurement* == *-1.0000* para descartar o sensor da lista local e, em toda recepção de mensagem, recalcular a distância para checar se o emissor segue como vizinho relevante.

A questão do recebimento da própria mensagem também gerou dúvidas. No modelo publish/subscribe, o sensor acaba recebendo de volta as medições que ele mesmo enviou, já que todos os dispositivos do mesmo tipo, inclusive ele, estão inscritos no tópico. O enunciado determina que, se as coordenadas do sensor emissor coincidirem com as do sensor receptor, deve-se registrar action: same location e descartar a atualização. Em um primeiro momento, essa repetição de logs podia parecer erro, mas, após testes, constatou-se que estava de acordo com o funcionamento normal do broadcast.

Houve também imprevistos relacionados a IPv4 e IPv6. A documentação pedia que o servidor aceitasse parâmetros para configurar se rodaria em v4 ou v6, o que exigiu ler o argumento (ex.: “v4” ou “v6”) e ajustar os parâmetros de *getaddrinfo* e *bind*. Da mesma forma, o cliente precisaria descobrir se o IP fornecido (127.0.0.1 ou ::1) corresponde a IPv4 ou IPv6. Em alguns testes, esquecia-se de colocar “v6” no servidor e, ao rodar o cliente com endereço ::1, a conexão não se estabelecia. A solução incluiu instruir claramente no código (ou na própria linha de comando) que, caso o servidor esteja em modo v6, o cliente também deve usar um IP v6.

Vale destacar o tratamento das mensagens exibidas no terminal. O servidor imprime logs no formato determinado para cada medição recebida. Já o cliente, além de imprimir tipo, coordenadas e valor da medição, deve adicionar “action:” seguido de “same location”, “correction of <valor>”, “not neighbor” ou “removed”, dependendo das circunstâncias. Foi um desafio lembrar de inserir uma linha em branco adicional ao final, conforme orientado, para melhorar a formatação e evitar confusões entre os logs. Houve ainda a necessidade de arredondar as correções a quatro casas decimais, indicando valores negativos com um “-” na frente caso a diferença fosse negativa.

Por fim, a atualização dos valores mostrou-se uma fonte de erros se o clamp (limite mínimo e máximo) não fosse aplicado após cada nova medição corrigida. Durante os testes, identificamos que, sem esse tratamento, o sensor podia exceder o intervalo estabelecido no enunciado, por exemplo, podendo chegar a 41 ou 42 ºC, fugindo da faixa de 20 a 40. Para corrigir, passamos a aplicar o clamp imediatamente depois de

recalcular a medição, garantindo que o valor se mantenha no intervalo válido. Isso foi essencial para não quebrar a lógica do sistema e respeitar o contexto de cada tipo de sensor.

Conclusões

A implementação final, apesar dos desafios de concorrência, sincronização, descoberta dinâmica e tratamento apropriado das mensagens, apresentou o comportamento esperado. Cada cliente inicia com seus parâmetros validados, gera uma medição aleatória dentro do intervalo adequado e passa a enviar essas medições ao servidor em intervalos periódicos (5, 7 ou 10 segundos, dependendo do tipo). O servidor controla as conexões por meio de threads, catalogando sensores em listas separadas para cada tipo, e faz o broadcast das mensagens de modo que somente clientes do mesmo tipo as recebam. O tratamento de desconexão (Ctrl + C) também foi contemplado, com o servidor enviando uma leitura de -1.0000 para sinalizar a remoção de um sensor.

Esse trabalho comprova a viabilidade de simular redes de sensores com publish/subscribe e correção de medições baseada na proximidade. A sincronização por mutexes, o cálculo de top 3 vizinhos e a formatação de logs conforme o enunciado foram fontes de dificuldades, mas também de aprendizado valioso sobre programação em rede com C e sobre arquiteturas distribuídas que necessitam de processos robustos de comunicação.