ChatMQTT

Mateus Azor Frutuoso, Thailha Roanni Schimit Cavalheiro

O projeto *ChatMQTT* nasce da tentativa de criar uma aplicação de chat inteiramente baseada no protocolo *MQTT*, utilizando *C* como linguagem principal e explorando tanto suas capacidades de baixo nível quanto os recursos avançados do padrão *MQTT*. Embora o usuário visualize apenas um menu textual em um terminal, por trás dessa simplicidade existe uma arquitetura rica, organizada em módulos independentes e na interação coordenada de diversas conexões *MQTT*, algumas persistentes, outras efêmeras, capazes de comunicar usuários, grupos e fluxos de controle de maneira confiável. A estrutura não é apenas funcional: ela foi pensada para demonstrar como, com o uso apropriado de mensagens retidas, *QoS* alto e tópicos hierárquicos bem definidos, é possível montar um sistema distribuído sem necessidade de serviços externos ou armazenamento local.

A espinha dorsal desse sistema é o arquivo *main.c*, que orquestra toda a experiência do usuário e funciona como centro da aplicação. Ele oferece a interface de linha de comando, interpreta entradas, coordena menus, dispara *publishers* e *subscribers* e gerencia a criação de *threads* paralelas. O *main* mantém ainda diversas listas de estado, (implementadas como *LinkedLists* com *mutexes*), para armazenar usuários conhecidos, grupos, eventos históricos, conversas, pendências e confirmações. Sempre que o programa precisa listar usuários, grupos ou eventos, o *main* cria conexões temporárias através do *subscriber*, consulta mensagens retidas, atualiza as listas internas e devolve ao usuário uma visão coerente da realidade do sistema. Isso significa que o *main* não apenas exibe a interface, mas também sincroniza constantemente o estado local com o estado distribuído mantido no *broker*.

Um dos módulos mais importantes é *agent.c*, responsável por uma conexão *MQTT* permanente que escuta o tópico exclusivo de controle do usuário, denominado *[USERNAME]_Control*. O *agent* funciona como um despachante inteligente: ele recebe mensagens de controle, interpreta seu tipo e executa a transformação apropriada conforme definido em *constants.c*. Por exemplo, quando chega uma mensagem *USER_REQUEST:Alice* no tópico de controle, o *agent* move essa informação para o subtópico *REQUESTS/* do usuário, guardando-a como mensagem retida para garantir que nenhuma solicitação se perca. Quando uma solicitação é aceita, o *agent* a converte em um evento de histórico, modificando o nome do tipo para *USER_REQUEST_ACCEPTED*, e publica isso em *HISTORY/*. É o *agent*, portanto, que materializa a lógica geral do sistema: ele transforma intenções em estado persistente e mantém coerência entre o que o usuário solicita, o que acontece e aquilo que fica guardado como memória distribuída.

A publicação de mensagens é tratada por *publisher.c*, um módulo que encapsula a complexidade de abrir uma conexão *MQTT*, configurar *callbacks* e publicar mensagens de forma síncrona. O projeto usa duas variações: o *publisher* padrão, que opera com *cleanSession = 1* e é ideal para operações rápidas e isoladas, e o *publisher* "sujo", *publisherDirty()*, que usa *cleanSession = 0* para preservar o estado da sessão e evitar que o *broker* apague informações importantes sobre o cliente. Esse módulo é estruturado para evitar que o restante da aplicação precise lidar com detalhes de baixo nível, como *flags* de conclusão, reconexões, *sleeps* e configurações *MQTTAsync*, tudo isso é encapsulado, permitindo que o *main* apenas invoque a função desejada e receba um retorno simples de sucesso ou falha.

O complemento natural do *publisher* é o módulo *subscriber.c*, que oferece vários modos de assinatura: leitura pontual de mensagens retidas, conexões persistentes para conversas em tempo real e assinaturas de sessão suja para manter estado ao longo de múltiplas interações. Assim como no *publisher*, o *subscriber* abstrai todos os detalhes relacionados à criação de cliente, *callbacks*, reconexão automática, gerenciamento de sessão e tratamento de chegada de mensagens. Para conversas em tempo real, (núcleo funcional do chat), existe o *subscriberConversation()*, que cria

uma *thread* dedicada capaz de escutar continuamente o tópico de conversa enquanto o usuário estiver digitando ou aguardando novas mensagens.

A sustentação estrutural desse fluxo contínuo de informação é fornecida pelo módulo *messages.c*, que implementa listas encadeadas *thread-safe* com funções de enfileiramento, desenfileiramento, busca e impressão formatada. Cada parte do sistema interage com essas listas para comunicar eventos. Para evitar conflitos entre múltiplas *threads*, cada lista emprega *mutexes* que travam e destravam o acesso a cada operação crítica. Esse módulo também oferece formatações convenientes, permitindo que conversas, históricos e grupos sejam apresentados ao usuário em formato legível, apesar de internamente serem apenas estruturas de texto.

Combinando esses módulos, o projeto constrói uma infraestrutura baseada no modelo *MQTT* que organiza seus estados através de diversos tipos de tópicos. Os principais são: *USERS/*, que guarda o status dos usuários; *GROUPS/*, que mantém a estrutura e os membros de cada grupo; *CHATS/*, que representa conversas reais em tempo de execução; e *[USER]_Control/*, subdividido em *REQUESTS/* e *HISTORY/*. Ao contrário do que ocorre em aplicações tradicionais, que normalmente dependem de bancos de dados ou serviços paralelos, essa arquitetura usa mensagens retidas para armazenar o estado global.

A lógica dos eventos segue uma transformação bem definida: mensagens como *USER_REQUEST*, *GROUP_REQUEST*, *USER_ACCEPTED* e *GROUP_ACCEPTED* são reinterpretadas pelo *agent* e movidas para os tópicos internos conforme sua natureza, solicitações vão para *REQUESTS/*, confirmações e rejeições vão para *HISTORY/*. Esse fluxo claro e sistemático permite que o sistema ofereça consistência na interpretação dos eventos por todos os clientes conectados. Com isso, o usuário pode navegar pelo menu, aceitar pedidos, rejeitar solicitações, criar conversas e administrar grupos sem nunca interagir diretamente com tópicos *MQTT* brutos: tudo é abstraído pelas camadas superiores.

Naturalmente, o projeto possui limitações e pontos que podem ser melhorados. A ausência de *TLS* e autenticação torna a comunicação vulnerável em ambientes não controlados; a dependência do *clientID* ser igual ao nome do usuário impede múltiplas instâncias simultâneas; e a memória totalmente volátil significa que o estado interno se perde ao reiniciar a aplicação. Mesmo assim, a arquitetura demonstra uma robustez significativa ao combinar modularidade, sessões persistentes, listas *thread-safe* e uma lógica consistente de transformação de eventos. O uso calculado de sessão limpa vs. sessão suja, *QoS* elevado e mensagens retidas mostra um domínio profundo dos recursos do *MQTT* e uma preocupação constante com a confiabilidade do sistema.

Ao final, o *ChatMQTT* se destaca não apenas como um cliente *MQTT* com interface textual, mas como uma arquitetura completa, modular, extensível e distribuída. Ele demonstra como é possível utilizar o ecossistema *MQTT* para muito mais do que telemetria: pode-se construir autenticação, grupos, conversas persistentes, histórico, reconstrução de estado e comunicação bidirecional em tempo real, tudo isso operando sobre uma base leve, eficiente e compatível com praticamente qualquer *broker MQTT* existente.