Relatório da Atividade RPC

Raiza Andrade de Freitas Tomazoni
Rafael Limeira de Castro Bueno
Rafael Luckwu Agripino

A atividade consiste em um serviço de listas remotas em Go exposto via RPC, no qual um servidor central mantém um mapa de listas inteiras em memória e oferece operações de Append, Remove, Size e Get para clientes remotos. O estado é protegido por **um sync.Mutex** para evitar condições de corrida **e é persistido em disco por meio de um log de operações e snapshots periódicos**, o que permite recuperar o conteúdo das listas após a reinicialização do processo. A concorrência foi exercitada principalmente com chamadas paralelas dentro de um mesmo cliente (várias goroutines fazendo Append e Remove na mesma lista), e o próprio desenho do sistema permite, em teoria, a execução simultânea de múltiplos processos clientes conectando-se ao mesmo servidor, ainda que não tenha sido desenvolvido um binário específico de “multicliente” para isso.
Durante o desenvolvimento, os principais desafios, além do pouco conhecimento acerca da linguagem Go, ficaram em torno do **controle de concorrência e da persistência**. Sem a trava (mu.Lock()/Unlock()), o acesso concorrente ao mapa de listas gera *data races* e risco de corrupção do estado; **com a trava, o sistema mantém consistência, mas cria um gargalo: todas as operações sobre qualquer lista passam pelo mesmo mutex e pelo mesmo processo servidor**, o que limita a escalabilidade quando o número de clientes ou de requisições cresce. Além disso, há vários pontos únicos de falha: um único processo servidor (se ele cair, o serviço para), um único disco para armazenar log e snapshot (qualquer problema de I/O ou corrupção de arquivo compromete a recuperação), e ausência de replicação ou mecanismo automático de *failover*. O tratamento de falhas é basicamente reativo: na reinicialização o servidor tenta carregar o snapshot e reaplicar o log, mas erros de leitura/escrita são apenas registrados em saída padrão.A solução desenvolvida implementa um serviço de listas inteiras acessível via RPC, no qual um único servidor central mantém todas as listas em memória e expõe operações remotas de *Append*, *Remove*, *Get* e *Size*. A consistência interna do estado foi garantida por meio de um mecanismo simples: o uso de um **único mutex global** (um sync.Mutex) que protege todo o mapa de listas. Dessa forma, qualquer operação — independentemente da lista envolvida — somente é executada quando o mutex está livre, evitando *data races* e corrupções de memória.

Embora funcional, essa estratégia utiliza **um mutex único para todo o serviço**, e não um mutex por lista ou mecanismo mais granular. Como consequência, na prática **não existe concorrência real entre listas diferentes**: todas as requisições RPC competem pela mesma trava. Portanto, mesmo que múltiplos workers ou processos clientes tentem acessar listas distintas, o servidor continuará atendendo

apenas uma operação por vez. Essa abordagem é suficiente para garantir consistência em um cenário simples, mas tem limitações severas de **desempenho** e **escalabilidade**.

Nos testes realizados com múltiplos workers simulando clientes concorrentes, observou-se que o servidor mantém a consistência mesmo sob alta pressão de chamadas, porém isso ocorre ao custo de maior **latência**. À medida que o número de workers aumenta, forma-se uma fila interna de operações aguardando o mutex global. Em outras palavras, quanto mais clientes simultâneos, maior o tempo de espera de cada operação, tornando o sistema obviamente inadequado para carga elevada ou para cenários com alto grau de paralelismo.

Além disso, a solução apresenta outro ponto crítico típico de sistemas monolíticos: **é composto por um único servidor**. Em caso de falha no processo ou indisponibilidade da máquina, todo o serviço deixa de funcionar, caracterizando um ***single point of failure***. Não há replicação, redundância, balanceamento de carga ou qualquer mecanismo distribuído capaz de manter o serviço ativo em caso de falhas.

Para que esse sistema fosse realmente adequado ao contexto de **Sistemas Distribuídos**, seria necessário repensar sua arquitetura. Uma solução escalável exigiria:

1. **Distribuição do estado em múltiplos servidores**

   — por exemplo, via *sharding*, onde cada servidor é responsável por um subconjunto das listas.

2. **Acesso concorrente granular**

   — utilizando mecanismos como um mutex por lista, ou um sync.RWMutex para permitir leituras paralelas.

3. **Replicação e coerência entre servidores**

   — o que demandaria algoritmos de consistência distribuída (p.ex. Raft, Paxos, Quorum Reads/Writes) para que múltiplas réplicas mantenham o mesmo estado mesmo diante de falhas ou atrasos de rede.

4. **Balanceamento de carga**

   — possibilitando que múltiplos clientes se conectem a diferentes servidores, reduzindo a contenção e a fila de requisições.

Essas soluções aumentariam significativamente a escalabilidade e a disponibilidade do serviço, mas trariam novos desafios em relação à **consistência**, que deixaria de ser garantida apenas por um mutex local e passaria a depender de protocolos distribuídos mais complexos.

Uma solução distribuída, contudo, traz novos trade-offs: ao espalhar o estado em múltiplos servidores ou ao adotar replicação assíncrona, ganhamos desempenho e disponibilidade, mas passamos a conviver com a possibilidade de leituras desatualizadas ou divergências temporárias entre réplicas, impactando a consistência global do sistema.