Escalonamento Concorrente para Replicação Máquina de Estados

Lorenzo Bragagnolo e Matheus Homrich

Escola Politécnica – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Porto Alegre – RS – Brasil

Abstract. *This article has the goal to study the scalability in the context of SMR(State Machine Repltication) and deepen the knowledge about hypotheses for concurrent staggering. SMR is a technique used to grow the availability of distributed systems, consisting in a set of deterministic replicas that begin in the same state and, receiving the same entries in the same order, maintaining it's replicas consistent.*
Beside the fact that it is very used, the classic technique does not scale with the growth of the flow rate with the core in a replica. This way, many strategies for concurrent staggering in SMR commands were studied.
This work studies the efforts of the research group DDC(Dependable Distributed Computing) in this sense and elaborates hypotheses for the growth of concurrency in this staggering.

Resumo. *Este trabalho tem por objetivo o do estudo de escalonamento aplicado a replicação máquina de estados (RME) e aprofundamento em hipóteses para o escalonamento concorrente. RME é uma técnica utilizada para aumentar a disponibilidade de sistemas distribuídos, consistindo em um conjunto de réplicas determinísticas que iniciam em mesmo estado, e, recebendo as mesmas entradas na mesma ordem, mantendo-se réplicas consistentes.*
Apesar de muito utilizada, a técnica clássica não escala vazão com o número de núcleos em uma réplica. Desta forma, diversas estratégias para o escalonamento concorrente de comandos em RME foram estudados.
Este trabalho estuda os esforços do grupo de pesquisa DDC (Dependable Distributed Computing) neste sentido e elabora hipóteses de aumento de concorrência neste escalonamento.

1. Introdução

Sistemas computacionais altamente disponíveis são cada vez mais necessários. Uma maior disponibilidade de um sistema pode ser alcançada através de replicação. A replicação de partes do sistema permite que mesmo se alguma parte esteja indisponível, o sistema continue operando.

A replicação ativa, ou Replicação Máquina de Estados (RME), foi proposta por Lamport[Lamport 1978] e aprofundada por Schneider [Schneider 1990], é um paradigma de replicação altamente empregado na indústria. Os princípios de RME são simples: se um conjunto de réplicas implementa o mesmo serviço determinístico, e se estas recebem as mesmas entradas, então irão gerar as mesmas saídas e atravessar os mesmos estados. Um serviço é determinístico se a sua saída e seu próximo estado depende somente do estado atual e da entrada fornecida. A recepção das mesmas entradas significa também

na mesma ordem. Isso pressupõe que todas réplicas recebem mensagens em ordem total. Esta funcionalidade é provida pela difusão atômica, que é equivalente ao problema do consenso.

Explorar arquiteturas multi-core implica na possibilidade de processamento concorrente nas réplicas. Como já colocado, isto pode ferir o determinismo das mesmas. Assim, a questão geral é como realizar processamento concorrente e determinístico nas réplicas. A literatura mostra diferentes abordagens. Há abordagens que introduzem sincronização/coordenação adicional entre as réplicas de forma que todas elas executam concorrentemente e deterministicamente, ou da mesma forma. Outras abordagens baseiam-se na observação, feita já em [Lamport 1978] de que diferentes requisições não conflitam, então sua ordem relativa não importa. Com isso, estas abordagens tratam de identificar conflitos entre requisições e processar na mesma ordem total somente as conflitantes, enquanto outras podem ser concorrentes.

Neste trabalho partimos do escalonamento concorrente para RME descrito em [Escobar et al. 2019] e desdobrado em [Pintor 2020] para investigar hipóteses de aumento de concorrência. Os resultados já atingidos são limitados à vazão da operação de inserção de novos comandos entregues pelo consenso em uma estrutura de controle de dependências. Este trabalho propõe melhorias neste aspecto.

Este texto se organiza da seguinte forma:

- uma introdução aos conceitos de RME;
- uma síntese do trabalho realizado em [Escobar et al. 2019];
- uma síntese do trabalho realizado em [Pintor 2020];
- Hipóteses Para o Aumento de Concorrência no Escalonamento de RME
- as propostas de melhorias;
- considerações finais.

2. RME

Replicação de Máquina de Estados é um conceito que consiste em possuir cópias do funcionamento do programa em diferentes máquinas, assim permitindo manter o mesmo estado para as diferentes máquinas, conseguindo garantir que a ação requisitada será executada, independente de ocorrer falhas em alguma das máquinas, como na Figura 1.

Enquanto RME oferece alta disponibilidade através da replicação, sua vazão não escala com o número de réplicas pois estas devem realizar a mesma computação. O determinismo necessário entre as réplicas é uma das razões para que o modelo seja sequencial em cada réplica, desta maneira o processamento concorrente interno gera diferentes estados em diferentes réplicas devido ao não-determinismo inerente à concorrência.

Assim, o modelo de RME, apesar de altamente difundido, requer avanços para suportar as crescentes demandas de vazão impostas a RME. Estes avanços têm acontecido, de forma pouco detalhada, em duas frentes: aumento da vazão intra-réplica tentando explorar arquiteturas multi-core; particionamento do estado da RME permitindo concorrência entre partições e assim aumento de vazão. Neste trabalho nos focamos no primeiro caso.

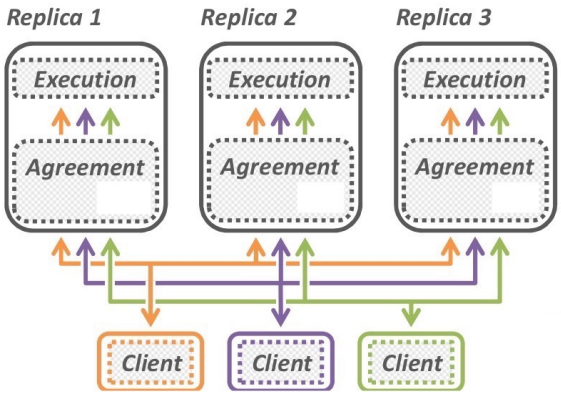


Figura 1. Arquitetura RME - adaptado de [Coyler]

3. Um GAD Livre de Bloqueio Para Escalonamento em RME

Em [Escobar et al. 2019] os autores[1] propuseram o uso de um grafo acíclico dirigido(GAD) como forma de organizar as requisições e registrar suas relações de dependência, permitindo assim o processamento concorrente de requisições independentes. O uso de GAD's para tal fim não é novo. Entretanto, observou-se que a detecção de conflito entre requisições exige que uma nova requisição seja comparada com todas as pendentes no grafo, esperando execução. Isto gera um custo significativo e, dependendo da forma como o GAD é acessado, esta estrutura representa uma região de contenção.

Assim, em [Escobar et al. 2019] foi proposto uma implementação livre de bloqueio para um GAD acessado concorrentemente. Esta estrutura mostrou vazão superior ao uso de um GAD com sincronização típica (bloqueio do grafo para operar). Em [Pintor 2020] foi realizado o estudo de uma implementação equivalente, com tecnologia mais recente, permitindo o uso de java para abstrair e encapsular as noções referentes ao funcionamento do GAD. Foi feito isso através de uma biblioteca do java chamada de *completable futures* [Urma and Fusco 2018]. Esta implementação mostrou-se de desempenho equivalente, entretanto mais simples de construir devido ao nível de abstração oferecido, dispensando a necessidade de tratar com técnicas para implementação lock-free.

Como alternativa à implementação proposta em [Escobar et al. 2019], pode-se fazer uso de estruturas de dados concorrentes através da biblioteca padrão da linguagem Java, sendo estas *CompletableFutures* e *ForkJoinPool*. Estas bibliotecas oferecem apoio para implementação do grafo de dependências, com seus nodos e arestas. Estas bibliotecas implementadas em java, nos permitem utilizar a concorrência utilizando um conceito chamado de *futures*, que consistem em lançar uma requisição caracterizada como assíncrona, ou seja, lança-se a requisição e não fica esperando pela resposta, mas sim, quando houver uma resposta, esta será recebida, garantindo dessa forma, uma melhor eficiência.

Esta biblioteca também nos fornece uma possibilidade de abstrair e encapsular a

[1]Do grupo de pesquisa *Dependable Distributed Computing* - PUCRS.

questão de dependências das tarefas, pois dentro da própria biblioteca existem funções específicas para a verificação de conflito, oferecendo dessa forma um maior nível de abstração.

4. Hipóteses Para Aumento de Concorrência no Escalonamento de RME

Em que pesa o aumento de vazão já conseguido nos trabalhos acima mencionados, estudos preliminares do grupo de pesquisa observam que o novo gargalo do sistema deixou de ser o GAD e passou a ser o processo sequencial de submissão de requisições para o GAD. Neste estudo investigamos implementações variadas que permitem inserções concorrentes no GAD já desenvolvido. Ou seja, vários processos concorrentes devem inserir requisições neste GAD, calculando todos os conflitos e dependências, e mantendo o GAD consistente.

4.1. Refinando o Escopo do Experimento

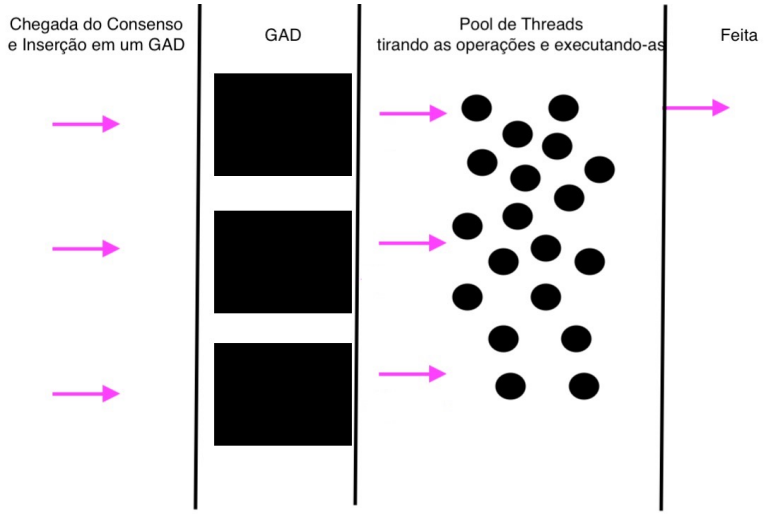
Para uma análise mais detalhada do problema, separamos somente a estrutura de grafo proposta em [Pintor 2020], desacoplando da plataforma de replicação, temos a implementação inicial da seguinte forma:

```
private static void taskScheduleParallelism(
int nThreads,
MessageContextPair msg,
PooledScheduler sch){

    ForkJoinPool forkJoinPool = null;
    try {
        forkJoinPool = new ForkJoinPool(nThreads);
        forkJoinPool.submit(() -> sch.schedule(msg)).get();

    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    } finally {
        if (forkJoinPool != null) {
            forkJoinPool.shutdown();
        }
    }

}
```

Para mais informações sobre a implementação do que foi citado anteriormente, procure [Pintor 2020] e [Escobar et al. 2019]

4.2. Proposta de Modificação

Para investigar o processamento concorrente de inserções no grafo, construímos um sistema com um pool de *worker threads*, assim como no original, e introduzimos um conjunto de threads clientes que fazem requisições concorrentes de inserção nesta estrutura. Anteriormente, estas requisições eram efetuadas sequencialmente por ocasião da entrega vinda do protocolo de consenso. Apresentamos aqui de forma detalhada o procedimento de inserção proposto para a estrutura concorrente e, também o código referente ao aumento de concorrência para o GAD.




```
PooledScheduler sch = new PooledScheduler(nt, cf);
sch.setExecutor(PooledScheduler::executeRequest);

for (int i=1; i<=nc; i++){
    new ClientThread(i, sch, nr).start();
}
```

Dessa forma, estamos lançando várias requisições através de threads clientes, onde ainda necessitamos da implementação de adição da tarefa no grafo.

```
private synchronized List<CompletableFuture<Void>> addTask(
Task newTask
) {
List<CompletableFuture<Void>> dependencies = new LinkedList<>()
ListIterator<Task> iterator = scheduled.listIterator();

while (iterator.hasNext()) {
    Task task = iterator.next();
    if (task.future.isDone()) {
        iterator.remove();
        continue;
    }
    if (conflict.isDependent(task.request, newTask.request)) {
        dependencies.add(task.future);
    }
}
scheduled.add(newTask);
return dependencies;
}
```

Note que foi introduzida a palavra reservada *synchronized* na declaração da função por motivos de garantir que as tarefas que possuem dependências sejam sequencializadas, porém o agendamento das tarefas é lançado de forma concorrente no trecho de código anterior.

5. Conclusão e Trabalhos Futuros

O código proposto está funcional, as *threads* clientes estão enviando as requisições de forma concorrente para a *pool de threads*, onde é verificado se existe conflito e, caso não exista conflito, a tarefa é executada dessa forma:

```
    private void submit(Task newTask,
        List<CompletableFuture<Void>>dependencies) {
    if (dependencies.isEmpty()) {
        pool.execute(() -> execute(newTask));
    } else {
        after(dependencies).thenRun(() -> {
            execute(newTask);
        });
    }
}
```

O teste de *benchmarking* do código pode ser implementado ainda, dessa forma, o próximo passo é validar a melhora de desempenho do sistema através do acoplamento do algoritmo implementado e, após isso, poderia-se dar um segundo passo no sentido de garantir a ordem total dos comandos conflitantes, através de mecanismos já existentes, pois isso só é garantido no nível de thread e não no nível de paralelismo total.
Uma sugestão de biblioteca para averiguar o *benchmark* é a JMH(Java Microbenchmark Harness), a qual nos possibilita o teste de operações feitas por segundo, o que nos daria um parâmetro para comparação.

Referências

Coyler, A. Hybrids on steroids: Sgx-based high-performance bft.

Escobar, I. A., Alchieri, E., Dotti, F. L., and Pedone, F. (2019). Boosting concurrency in parallel state machine replication. page 228–240.

Lamport, L. (1978). Time, clocks, and the ordering of events in a ditributed system. *Communications of the ACM*, pages 558–565.

Pintor, E. (2020). Explorando concorrência em aplicações chave-valor com replicação de máquinas de estado pintor, e. (2020a). parallel smr library. https://github.com/erickpintor/ parallel-smr.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, page 299–319.

Urma, R.-G., M. A. and Fusco, M. (2018). *Modern Java in Action. Manning Publications.*