Proposta de *Runtime* em Nuvem Baseada em Atores e Tolerância a Falhas por Logs para *Stream Processing* em C++

André L. Rodrigues[1], **Eduardo M. Martins**[1], **Dalvan Griebler**[1]

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

(andre.rodrigues99,e.martins01)@edu.pucrs.br,dalvan.griebler@pucrs.br

Resumo. *O Asynchronous Barrier Snapshotting (ABS) é um protocolo de tolerância a falhas, utilizado em sistemas de streaming. Entretanto, ele pode apresentar overhead de desempenho devido à algumas limitações de algoritmos de snapshot, como a necessidade de pausar e reiniciar toda a pipeline para recuperar o estado após a falha de um operador e a impossibilidade de escalar dinamicamente os operadores sem interromper a aplicação. O protocolo LOG.io surgiu como uma alternativa que resolve essas limitações apresentadas. Contudo, os resultados de desempenho obtidos indicaram que o LOG.io apresenta limitações em cenários com altas taxas de input de dados. Desse modo, este trabalho tem como objetivo implementar, no framework Resiflow, os protocolos ABS e LOG.io em uma runtime baseada em nuvem, a fim de analisar e comparar o impacto no desempenho de ambos sob diferentes cargas de dados e cenários de falha.*

1. Introdução

De acordo com Andrade et al. [Andrade et al. 2014], *stream processing* é um paradigma de computação voltado à coleta, processamento e análise de grandes volumes de dados heterogêneos gerados de forma contínua em tempo real. Diferentemente do tradicional paradigma de *batch processing*, que armazena e processa dados em intervalos pré-estabelecidos, *stream processing* atua no exato momento em que o dado chega ao sistema.

Como sistemas de *stream processing* são projetados para operar continuamente e por tempo indeterminado, reprocessar todos os dados após uma falha é inviável. Desse modo, a resiliência se torna um requisito essencial, garantindo que o sistema consiga se recuperar para um estado consistente. Caso contrário, corre-se o risco de produzir resultados incorretos ou defasados, dada a natureza *real-time* dessas aplicações. Nesse contexto, diversos protocolos foram propostos para oferecer essa funcionalidade.

O *Asynchronous Barrier Snapshotting* (ABS) é um dos protocolos de *snapshot* de estado da arte para sistemas de *stream processing*. Contudo, como descrito por Simon et al.[Simon et al. 2025], durante a recuperação de falhas, ele introduz *overhead* considerável, pois exige reiniciar a aplicação e reprocessar todos os dados desde o último *checkpoint*. Além disso, em sistemas maleáveis, o ABS não oferece suporte à escalabilidade dinâmica, o que implica em pausar a aplicação sempre que for necessário aumentar ou diminuir os recursos.

Protocolos baseados em logging são alternativa a essas limitações. Mais especificamente, o protocolo LOG.io é um mecanismo recente de tolerância a falhas baseado

em logs e capaz de realizar a recuperação de falha em nível de operador. Isso permite que possa recuperar o estado de um nodo falho sem a necessidade de interromper a aplicação, garantindo o processamento contínuo dos dados tanto durante a recuperação de falhas quanto ao escalar dinamicamente a aplicação. Contudo, apesar dessas vantagens, o LOG.io apresentou *overhead* de performance significativo quando submetido a cenários com altas cargas de dados.

Diante desse contexto, este trabalho apresenta a proposta de implementar uma *runtime* baseada em nuvem no *framework* ResiFlow, além de avaliar o desempenho da implementação dos protocolos ABS e LOG.io em C++, utilizando uma arquitetura baseada em atores neste ambiente. Para isso, será realizada uma análise de performance de ambos os protocolos em diferentes cenários de falhas e sob variadas cargas de dados.

2. Background

2.1. Asynchronous Barrier Snapshotting (ABS)

O ABS é um protocolo de *snapshot* para sistemas de *dataflow* distribuídos. Derivado do clássico protocolo de *snapshot* proposto por Chandy-Lamport [Chandy and Lamport 1985], o ABS tem como objetivo solucionar duas das principais limitações presentes em outros protocolos de *snapshots*: o elevado impacto na performance e o alto custo de armazenamento.

Alguns protocolos de *snapshot* exigem a interrupção de toda a aplicação para capturar os *snapshots*, como é o caso do Naiad [Murray et al. 2013], por exemplo. Carbone et al. [Carbone et al. 2015] destaca que essa estratégia afeta tanto o *throughput* quanto custo armazenamento, pois toda a computação precisa ser bloqueada para obter o *snapshot*. Diferentemente, o ABS não interrompe o processamento: ele injeta marcadores de barreira no *pipeline*, que percorrem o grafo até os *sinks*, permitindo capturar o estado de forma incremental e sem suspender a execução.

Outra abordagem muito utilizada no protocolo proposto por Chandy–Lamport [Chandy and Lamport 1985], no qual todos os registros "em trânsito" nos canais de comunicação são persistidos como parte do *snapshot*. Isso cria *snapshots* significativamente maiores do que o necessário, pois incluem não apenas o estado dos operadores, mas também um log completo das mensagens que circulam na rede, gerando um aumento no custo de armazenamento e maior tempo de recuperação. O ABS, por sua vez, reduz esses custos ao não realizar a persistência dos estados dos canais dos grafos acíclicos. No caso de grafos cíclicos, ele armazena apenas um log mínimo correspondente aos *back-edges* dos *loops*, em vez de registrar todos os dados em transito.

Apesar das melhorias, o ABS ainda apresenta limitações que introduzem *overhead* considerável. Qualquer falha em um operador resulta na parada completa no *pipeline* de processamento. Além disso, o protocolo não oferece mecanismos para escalonamento dinâmico, o que impede o aumento ou a redução de recursos em tempo de execução. Para reescalar um operador, é necessário tirar um *snapshot* global, interromper a computação, ajustar o paralelismo configurado para aquele operador, reimplantar o pipeline com as novas configurações e, então, retomar a execução a partir do *snapshot* mais recente. Essas duas limitações resultam em *overhead* adicional, impactando a eficiência do sistema.

2.2. LOG.io

O LOG.io é um protocolo de recuperação por *rollback* baseado em logs, criado para sistemas de *stream processing* distribuídos. Ele registra, de forma incremental, as modificações do estado dos operadores e as dependências dos dados (*fine-grain data lineage*). O protocolo oferece suporte a grafos cíclicos e acíclicos, permite escalonamento dinâmico e possibilita que a recuperação de falhas seja realizada em nível de operador, de forma independente, sem interromper os operadores que permanecem ativos. Assim, ao restaurar apenas os operadores que falharam, o LOG.io busca reduzir tanto o *overhead* em tempo de execução quanto o tempo de recuperação.

Em comparação ao ABS, o LOG.io apresenta desempenho melhor ou equivalente sob cargas de dados baixas ou moderadas, destacando-se positivamente em cenários com falhas e demonstrando que as melhorias propostas atingem o objetivo esperado. Contudo, quando submetido a altas taxas de entrada de dados, o protocolo apresenta um *overhead* significativo. Conforme explicado por Simon et al.[Simon et al. 2025], essa degradação é provocada pelo custo do registro dos logs dos eventos (*fine-grained logging*), que se torna um gargalo à medida que a carga de dados aumenta.

2.3. ResiFlow

ResiFlow é um *framework* para *stream processing* projetado para dar suporte ao desenvolvimento de aplicações de *streaming* paralelas, distribuídas, resilientes e de alto desempenho em C++. Construído sobre a infraestrutura do MPI, o *framework* tem como objetivo simplificar a criação de aplicações complexas baseadas em grafos acíclicos direcionados.

Uma das principais propostas da ResiFlow é apresentar ao desenvolvedor os principais conceitos de execução. Em vez de ocultar o comportamento do sistema por meio de abstrações de alto nível, o *framework* oferece acesso direto à execução dos operadores, aos canais de comunicação e ao estado da aplicação. Essa abordagem reduz o *overhead* introduzido por camadas de abstração e torna o comportamento do sistema mais transparente. Como resultado, a ResiFlow facilita a prototipação, a implementação e avaliação de diferentes abordagens para execuções paralelas e tolerância a falhas.

2.4. C++ Actor Framework (CAF)

Introduzido por Charousset et al. [Charousset et al. 2014], o C++ *Actor Framework* (CAF) foi projetado para fornecer um ambiente nativo e escalável para aplicações concorrentes de alto desempenho e sistemas distribuídos. O *framework* se baseia no modelo de atores, originalmente proposto por Hewitt et al. [Hewitt et al. 1973], o qual descreve programas computacionais como entidades independentes que trocam mensagens e realizam comunicação de maneira tolerante a falhas e transparente à rede [Armstrong 2003].

Nesse modelo, os atores são os componentes principais responsáveis pela concorrência e distribuição. As mensagens recebidas são armazenadas em uma fila de ordem FIFO e processadas sequencialmente. Cada ator pode: enviar um número finito de mensagens a outros atores; criar novos atores; e definir o comportamento que será adotado ao receber a próxima mensagem. Além disso, atores não compartilham estado e toda comunicação ocorre de forma estritamente assíncrona.

3. Proposta

Diante do contexto de que ter mecanismos eficientes de tolerância a falhas é um requisito essencial para sistemas de *stream processing*, e considerando as limitações apresentadas pelos protocolos ABS e LOG.io, a presente proposta de plano de estudo e pesquisa de mestrado tem como objetivo implementar uma *runtime* baseada em nuvem para o *framework* ResiFlow, sem modificar sua API de referência. Além disso, o trabalho tem como objetivo avaliar o desempenho das implementações dos protocolos ABS e LOG.io em C++, empregando uma arquitetura baseada em atores nesse ambiente. Para isso, será conduzida uma análise de performance de ambos os protocolos em diferentes cenários de falha e sob variadas cargas de dados.

Nesse sentido, a pesquisa busca responder a duas questões. A primeira consiste em entender como fornecer uma *runtime* escalável para execução em nuvem sem alterar a API de referência do ResiFlow. Parte-se da hipótese de que uma implementação baseada no modelo de programação orientado a atores utilizando o protocolo LOG.io pode oferecer maior escalabilidade em relação ao ABS. A segunda questão busca avaliar o impacto de desempenho de ambos protocolos na ResiFlow quando executados em ambiente de nuvem, partindo da hipótese de que o LOG.io apresenta desempenho inferior ao ABS em cenários sem falhas, mas proporciona maior *throughput* e menor tempo de indisponibilidade nos cenários em que falhas ocorrem.

Referências

Andrade, H. C., Gedik, B., and Turaga, D. S. (2014). *Fundamentals of stream processing: application design, systems, and analytics*. Cambridge University Press.

Armstrong, J. (2003). *Making reliable distributed systems in the presence of software errors*. PhD thesis.

Carbone, P., Fóra, G., Ewen, S., Haridi, S., and Tzoumas, K. (2015). Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603*.

Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75.

Charousset, D., Hiesgen, R., and Schmidt, T. C. (2014). Caf-the c++ actor framework for scalable and resource-efficient applications. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*, pages 15–28.

Hewitt, C., Bishop, P., and Steiger, R. (1973). Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance papers of the conference*, volume 3, page 235. Stanford Research Institute Menlo Park, CA.

Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. (2013). Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455.

Simon, E., Hoffmann, R. B., Alf, L., and Griebler, D. (2025). Log. io: Unified rollback recovery and data lineage capture for distributed data pipelines. *arXiv preprint arXiv:2512.16038*.