# Um Protocolo Pessimista para Registro de Mensagens Baseado em um *Event Logger* Distribuído e Tolerante a Falhas

**Edson Tavares de Camargo**[1,2], **Fernando Pedone**[3], **Elias P. Duarte Jr.**[2]

[1]Universidade Tecnológica Federal do Parará - Campus Toledo (UTFPR)
CEP: 85902-490 – Toledo – PR – Brasil

[2]Universidade Federal do Paraná (UFPR) – Programa de Pós-Graduação em Informática
Caixa Postal 19081 – 81531-980 – Curitiba – PR – Brasil

[3]Università della Svizzera italiana (USI) – Faculty of Informatics
CH-6904 – Lugano – Switzerland

edson@utfpr.edu.br, fernando.pedone@usi.ch, elias@inf.ufpr.br

***Abstract.*** *Most message logging protocols rely on a centralized event logger in order to store recovery information i.e, the determinants. This centralized approach, besides representing a single point of failure, is a bottleneck for the performance of the message logging protocol. In this work, we present a pessimistic message logging protocol built with a fault-tolerant distributed event logger based on consensus. Both are implemented and evaluated. Results show that the distributed event logger outperforms the centralized counterpart and that the proposed protocol allows efficient application recovery.*

***Resumo.*** *A maioria dos protocolos de registro de mensagens conta com um event logger centralizado para armazenar informações de recuperação, isto é, os determinantes. Essa abordagem centralizada, além de apresentar um ponto único de falha, representa um gargalo para o desempenho dos protocolos de registro de mensagens. Neste trabalho, apresentamos um protocolo pessimista para registro de mensagens construído com um event logger distribuído e tolerante a falhas baseado em consenso. Ambos são implementados e avaliados. Resultados demonstram que o event logger distribuído tem desempenho superior ao da abordagem centralizada e que o protocolo proposto realiza a recuperação da aplicação eficientemente.*

## 1. Introdução

*Rollback-recovery* é uma técnica de tolerância a falhas tradicionalmente empregada em sistemas de alto desempenho e de longa duração [Egwutuoha et al. 2013]. O objetivo da técnica é restaurar o sistema a um estado consistente após uma falha [Elnozahy et al. 2002]. Para tanto, o estado de um processo é salvo periodicamente durante a sua execução e, perante uma falha, reiniciado a partir de um estado anterior, reduzindo assim a quantidade de trabalho perdido. O *checkpoint* é a ação de armazenar periodicamente o estado de um processo sem-falha em execução. Protocolos de registro de mensagens são uma categoria de *rollback-recovery* que, ao contrário da abordagem coordenada, não exigem a sincronização dos *checkpoints*. Além disso, na sua abordagem pessimista apenas o processo que falha é reiniciado. A abordagem garante uma

recuperação consistente a partir de *checkpoints* tomados independentemente em cada processo [Lifflander et al. 2014, Liu et al. 2013, Bouteiller et al. 2010].

Protocolos de registro de mensagens assumem que todos os eventos não-determinísticos que um processo executa podem ser identificados e a informação necessária para reaplicar cada evento durante a recuperação pode ser registrada em tuplas chamadas de *determinantes* [Elnozahy et al. 2002]. Considerando que os determinantes são salvos em uma entidade confiável, um processo em recuperação pode recriar deterministicamente o estado anterior à falha ao reaplicar os determinantes na sua ordem original. Consequentemente, uma tarefa crucial nos protocolos de registro de mensagens é salvar e recuperar de forma confiável os determinantes sem penalizar o desempenho da aplicação.

O componente responsável por armazenar de forma confiável os determinantes é chamado de *event logger*. O *event logger* recebe os determinantes dos processos da aplicação, armazena-os localmente, e notifica os processos da aplicação. Protocolos de registro de mensagens normalmente assumem que o *event logger* é uma entidade centralizada que não tolera falhas [Bouteiller et al. 2009, Ropars and Morin 2009, Bouteiller et al. 2005]. De fato, a falha de um *event logger* centralizado pode paralisar os processos da aplicação, uma vez que esses não mais conseguem salvar os determinantes.

O objetivo deste trabalho é apresentar um *event logger* distribuído e tolerante a falhas que tem desempenho comparável ou superior à abordagem centralizada, bem como um protocolo pessimista de registro de mensagens para interagir com o *event logger* proposto. Em particular, o *event logger* replicado não requer recursos extras (nodos físicos) em comparação a um *event logger* centralizado. Os *event loggers* replicados podem ser hospedados nos mesmos nodos dos processos da aplicação. Além disso, o *event logger* distribuído pode tolerar um número configurável de falhas.

Duas implementações do *event logger* proposto foram realizadas. Ambas são baseadas no algoritmo Paxos [Lamport 2001]. A primeira implementação se apoia em uma configuração tradicional do Paxos, chamadas de Paxos clássico. A segunda configuração é chamada de Paxos paralelo. As duas implementações são comparadas com uma implementação de um *event logger* centralizado. Um protocolo pessimista de registro de mensagens é implementado em MPI para interagir com o *event logger* proposto. O MPI é o padrão de *facto* para o desenvolvimento de aplicações paralelas e distribuídas de alto desempenho [Fagg and Dongarra 2000]. Entre as principais características do protocolo proposto estão as seguintes: a distinção entre eventos determinísticos e não-determinísticos em MPI; o emprego da abordagem de *checkpoint* não-coordenado; e a recuperação automática da aplicação perante falhas de processos. Os *event loggers* foram avaliados usando as aplicações LU e MG do *NAS Parallel Benchmark*, a aplicação AMG (*Algebraic MultiGrid*) e o algoritmo paralelo de Gusfield. O algoritmo de Gusfield também foi empregado para avaliar a recuperação. Resultados demonstram que o *event logger* baseado na abordagem Paxos Paralelo tem desempenho superior ao *event logger* centralizado e que o protocolo proposto realiza a recuperação da aplicação eficientemente.

Este trabalho segue organizado da seguinte forma. A Seção 2 apresenta os conceitos principais da técnica de *rollback-recovery*. A Seção 3 detalha o protocolo proposto, incluindo o *event logger* baseado em consenso. A Seção 4 aborda a implementação e a Seção 5 os resultados experimentais. Por fim, a Seção 6 apresenta a conclusão.

## 2. A Técnica de *Rollback-Recovery*

A técnica de tolerância a falhas *rollback-recovery* é frequentemente empregada para prover tolerância a falhas em aplicações de alto desempenho. Desse modo, as aplicações podem reiniciar a partir de um estado salvo previamente. A técnica assume um sistema distribuído, onde os processos da aplicação se comunicam através de uma rede e têm acesso a um dispositivo de armazenamento confiável que sobrevive a falhas [Elnozahy et al. 2002]. Periodicamente, os processos salvam informações de recuperação no dispositivo confiável durante a sua execução sem-falhas. Após a ocorrência de uma falha, a aplicação usa as informações de recuperação para reiniciar a sua computação a partir de um estado anterior. As informações de recuperação incluem os *checkpoints*, isto é, os estados dos processos participantes. Alguns protocolos também incluem o registro de eventos não-determinísticos, codificados em tuplas chamadas de *determinantes*. Basicamente, um estado global consistente é aquele em que se o estado de um processo reflete uma mensagem recebida, então o estado correspondente do emissor deve refletir o envio daquela mensagem [Kshemkalyani and Singhal 2011]. Um conjunto de *checkpoints* que corresponde a um estado consistente é chamado de *linha de recuperação*. O principal objetivo dos protocolos de *rollback-recovery* é restaurar o sistema a partir da mais recente linha de recuperação após uma falha. Os protocolos de *rollback-recovery* podem ser classificados em duas categorias: baseados somente no *checkpoint* (*checkpoint-based*) ou baseados em registro de mensagens (*log-based*), descritos a seguir.

### 2.1. *Rollback-Recovery* Baseado em *Checkpoints*

Os protocolos de *rollback-recovery* baseados em *checkpoints* dividem-se em duas abordagens: coordenada e não coordenada. Quando o *checkpoint* é realizado independentemente em cada processo, sem uma coordenação global, é chamado de não coordenado. A vantagem da abordagem não coordenada está em cada processo criar o seu *checkpoint* quando lhe é mais conveniente. Entretanto, com essa abordagem, um estado global consistente pode nunca ser atingido. Nesse caso, os *checkpoints* realizados tornam-se inúteis e devem ser descartados [Elnozahy et al. 2002, Kshemkalyani and Singhal 2011].

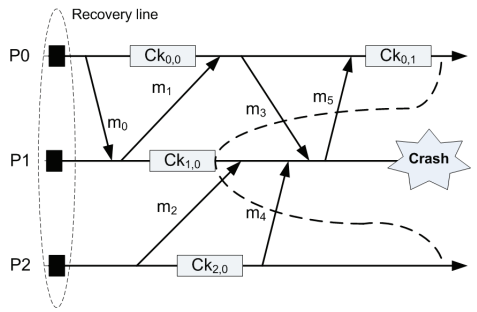


**Figura 1. *Checkpoint* não coordenado e um linha de recuperação.**

A Figura 1 apresenta um cenário no qual o mais recente conjunto de *checkpoints* (isto é, $Ck_{0,1}$, $Ck_{1,0}$, e $Ck_{2,0}$) não resulta em uma linha de recuperação. Isso se deve ao fato de que a mensagem $m_5$ é recebida por $P_0$ mas não enviada por $P_1$ — nesse caso, $m_5$ é chamada de uma *mensagem órfã* e $P_0$ um *processo órfão*. $P_0$ é então obrigado a retroceder a um *checkpoint* anterior (isto é, $Ck_{0,0}$). O problema entretanto persiste, pois

$m_1$ e $m_2$ precisam ser retransmitidas. Desta forma, a única linha de recuperação corresponde ao estado inicial da aplicação. Esse fenômeno é conhecido como *efeito dominó*. O *checkpointing* não coordenado é suscetível ao efeito dominó.

O *checkpointing* coordenado evita o efeito dominó. Os processos se sincronizam para realizar os *checkpoints* e, consequentemente, criar um estado global consistente [Kshemkalyani and Singhal 2011]. Embora a abordagem coordenada seja relativamente fácil de implementar, a sua execução impõe uma sobrecarga considerável à aplicação uma vez que os processos precisam se coordenar e salvar os seus estados simultaneamente no dispositivo de armazenamento. Além disso, mesmo que um único processo falhe, todos os processos precisam retroceder ao último *checkpoint*.

### 2.2. *Rollback-Recovery* Baseado em Registro de Mensagens

Os protocolos de *rollback-recovery* baseados em registro de mensagens empregam tanto *checkpoints* quanto o registro de eventos não-determinísticos com o objetivo de evitar as desvantagens das abordagens coordenada e não coordenada. Os protocolos de registro de mensagens assumem que todos os eventos não-determinísticos executados por um processo podem ser identificados e a informação necessária para reproduzir cada evento durante a recuperação pode ser registrada em determinantes [Kshemkalyani and Singhal 2011]. Um *evento* corresponde a um passo de comunicação ou um passo de computação de um processo. A maioria dos protocolos de registro de mensagens assume que a recepção das mensagens é o único evento não-determinístico. O registro de mensagens evita o efeito dominó do *checkpointing* não coordenado salvando todas as mensagens recebidas. Por exemplo, na Figura 1, as mensagens $m_2$, $m_4$ e $m_3$ recebidas pelo processo $P_1$ devem ser salvas, assim como os determinantes que contém a ordem de recepção das mensagens. Durante a recuperação somente o processo $P_1$ retrocede. Assim, o estado de $P_1$ eventualmente será o mesmo ao anterior à falha uma vez que as mensagens $m_2$, $m_4$ e $m_3$ são reaplicadas na mesma ordem.

Dependendo de como os determinantes são registrados, os protocolos de registro de mensagem podem ser classificados em pessimista, otimista ou causal [Elnozahy et al. 2002]. Na abordagem pessimista do registro de mensagens um processo primeiro armazena o determinante antes de entregar a mensagem. Apesar de simplificar a recuperação e a coleta de lixo, a abordagem pessimista gera uma sobrecarga durante a execução da aplicação: a aplicação precisa aguardar pelo armazenamento de cada determinante.

Como descrito em [Bouteiller et al. 2010], é possível reduzir o número total de determinantes armazenados distinguindo os eventos determinísticos dos não-determinísticos. Um evento é determinístico quando a partir do estado atual existe somente um estado resultante possível para o evento. Se um evento pode resultar em estados diferentes, então é dito não-determinístico. A recepção de mensagens com uma identificação de emissor explícita é um evento-determinístico e não requer o seu registro. Ao contrário, quando se aguarda uma mensagem de um emissor desconhecido então a recepção é dita não-determinística. Conforme definido em [Cappello et al. 2010], muitas aplicações MPI contêm somente eventos de comunicação determinísticos. Porém, muitas importantes aplicações MPI são não-determinísticas. Além disso, os desenvolvedores geralmente incluem o não-determinismo na codificação para melhorar o desempenho da aplicação.

## 3. Um Protocolo para Registro de Mensagens Baseado em Consenso

O *event logger* desempenha um papel crucial nos protocolos de registro de mensagens [Bouteiller et al. 2005]. O *event logger* recebe os determinantes, armazena-os localmente, e notifica os processos da aplicação após armazená-los. Apesar do *event logger* exercer este grande impacto, muitos protocolos o implementam como um componente centralizado e incapaz de tolerar falhas [Ropars and Morin 2009, Bouteiller et al. 2006, Ropars and Morin 2010]. Uma vez que *event logger* precisa notificar os processos da aplicação ao salvar um determinante, um *event logger* centralizado facilmente se torna em um gargalo conforme aumenta o número de processos. Além disso, a falha do *event logger* pode paralisar a aplicação ou levá-la a um estado inconsistente durante a recuperação. Nesta seção apresentamos o modelo do sistema, a descrição do protocolo proposto e do serviço de *event logger* propriamente dito.

### 3.1. Modelo de Sistema

Considere um sistema representado por um grafo unidirecional completo $G = (V, E)$, o conjunto de vértices $V$ corresponde ao conjunto de processos e uma aresta $(i, j) \in E \mid i, j \in V$ representa a habilidade dos processos $i$ e $j$ de se comunicar diretamente, sem intermediários. O sistema é assíncrono com detectores de falhas. Um processo pode estar em um de dois estados: falho ou sem-falha. O modelo de falhas é o de parada com recuperação (*crash-recovery*). Os canais de comunicação são confiáveis e FIFO. Destaca-se que mensagens recebidas concorrentemente de diferentes emissores são entregues em qualquer ordem. Uma execução é uma sequência alternada de eventos e estados de processos. O efeito de um evento no estado de um processo conduz o processo a um novo estado. A ordem parcial entre os eventos é obtida através da relação *happened-before* [Kshemkalyani and Singhal 2011]. Os eventos são classificados em determinísticos e não-determinísticos.

Os eventos não-determinísticos de um processo são identificados e armazenados em determinantes. Um determinante de um processo $i$ é formado pelo seu identificador, o identificador do emissor da mensagem e o tipo de evento (por exemplo, se uma recepção ou verificação de mensagem recebida), ou seja: $det_i \leftarrow (det_{id}, id_j, event\_type)$. Inicialmente, $det_{id}$ é definido em 0 e é incrementado a cada novo determinante armazenado. Os processos têm acesso a um serviço de *event logger* (definido na Seção 3.3) para salvar, remover e recuperar os determinantes. Ao salvar um determinante, o *event logger* retorna ao processo o identificador do determinante. Um processo $i$ incrementa contadores $sent_i[j]$ e $recv_i[j]$ ao enviar ou receber uma mensagem do processo $j$, respectivamente. Os contadores são representados por vetores de tamanho $|V|$, onde cada posição no vetor é indexada pelo identificador do processo correspondente e inicializada em 0. O protocolo assume a abordagem *sender-based* [Johnson and Zwaenepoel 1987]: toda vez que o processo $i$ envia uma mensagem $msg$ ao processo $j$, uma cópia da mensagem é mantida na memória volátil de $i$. O identificador da mensagem é formado pelo $id$ do emissor e a sequência de envio, ou seja $msgs_i \leftarrow (id_j, sent_i[j], msg)$.

### 3.2. Descrição do Protocolo

Periodicamente, usando o seu relógio local, cada processo realiza um *checkpoint*, o qual é armazenado em um dispositivo confiável. O *checkpoint* inclui o estado do processo, as suas mensagens enviadas mantidas em memória volátil, o identificador do

último determinante submetido ao *event logger* e os contadores de mensagens enviadas e recebidas. Ou seja, o *checkpoint* em um processo $i$ tem o seguinte formato: $ck_i \leftarrow (p_i, msgs_i, det_{id}, sent_i[], recv_i[])$. Apenas o último *checkpoint* do processo é mantido. A cada *checkpoint* em $i$, todos os determinantes de $i$ mantidos até então no *event logger* podem ser removidos. Além disso, cada processo $j$ pode apagar as mensagens enviadas para $i$ mantidas em sua memória tal que $sent_j[i] \leq recv_i[j]$.

A recuperação de um processo ocorre da seguinte forma. Um novo processo $i$ é criado para substituir o processo falho. O processo $i$ lê o seu *checkpoint*, obtém os seus determinantes do *event logger* e solicita a cada processo $j$ que reenvie as mensagens de $i$ a partir da última mensagem recebida por $i$, ou seja, $\forall j \in V$, $i$ envia $recv_i[j]$ a $j$. Então, $j$ reenvia $msg$ tal que $sent_j[i] > recv_i[j]$. O processo $j$ também envia $recv_j[i]$ para que $i$ saiba qual foi a última mensagem que $j$ recebeu de $i$. As mensagens $sent_i[j] \leq recv_j[i]$ não serão reenviadas, pois $j$ as recebeu antes da falha de $i$.

O processo $i$ então reinicia sua computação a partir do seu *checkpoint*. Cada evento determinístico encontrado será reproduzido. Os eventos não-determinísticos serão substituídos pelas informações contidas nos determinantes lidos do *event logger*. Por exemplo, considere a falha de $P_1$ (Figura 1). Antes de falhar, os determinantes $det_1$, $det_2$ e $det_3$ correspondentes às mensagens $m_2$, $m_4$ e $m_3$ foram salvos no *event logger*. Em sua execução normal, $P_1$ aguarda mensagens de quaisquer emissores. Mas, em recuperação, e de posse dos determinantes $det_1$, $det_2$ e $det_3$ recuperados do *event logger*, $P_1$ aguarda as mensagens vindas de $P_2$, $P_2$ e $P_0$, nessa ordem. Uma vez que cada mensagem é reaplicada, a mensagem $m_5$ é então gerada e armazenada em memória volátil. A partir de então, a execução de $P_0$, $P_1$ e $P_2$ segue normalmente.

### 3.3. O Serviço Proposto de *Event Logger* Baseado em Consenso

O serviço de *event logger* proposto é baseado em consenso. O consenso é uma abstração fundamental na computação distribuída tolerante a falhas. O consenso pode ser usado para construir um serviço de *event logger* usando a abordagem de máquina de estado [Schneider 1990]. A replicação máquina de estado regula como os comandos devem ser propagados e executados pela réplicas para que o serviço seja consistente. No nosso caso, os comandos são solicitações para salvar um determinante, propagados e executados pelas réplicas do *event logger*. A propagação dos comandos possui dois requisitos: (i) cada réplica sem-falha deve receber cada comando e (ii) duas réplicas quaisquer não podem discordar na ordem dos comandos recebidos e executados. Se a execução é determinística, então as réplicas alcançarão o mesmo estado e produzirão a mesma saída ao executar a mesma sequência de comandos.

Paxos é um algoritmo de consenso que garante a propriedade de segurança do consenso em um sistema assíncrono sujeito a falhas e a propriedade de progresso sob suposições de assincronia fraca. No Paxos os processos podem assumir os seguintes papéis distintos: *proposers*, *acceptors* e *learners*. Os *proposers* propõe um valor, os *acceptors* escolhem um valor e os *learners* aprendem o valor decidido. Um único processo pode assumir qualquer uma dessas funções, e múltiplas funções simultaneamente. Paxos é ótimo em termos de resiliência [Lamport 2006]: para tolerar $f$ falhas ele requer $2f + 1$ *acceptors*. Um processo *coordenador* ainda pode ser escolhido para evitar que os *proposers* concorram indefinidamente por um valor. Com a presença de um coordenador, um comando pode ser aprendido em três passos de comunicação [Lamport 2001].

### 3.3.1. *Event Loggers* Baseados em Paxos Clássico e Paxos Paralelo

Dois procolos baseados em Paxos Clássico e Paxos Paralelo são propostos para construir o *event logger* replicado. A Figura 2 apresenta as três abordagens.

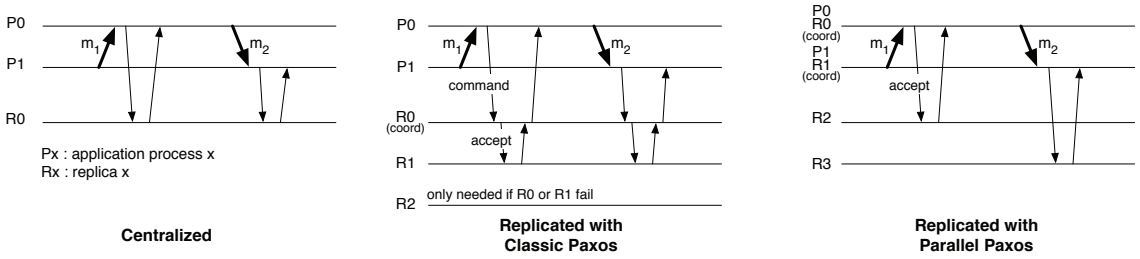


**Figura 2. Três implementações de um *event logger*.**

Os processos da aplicação são representados por $P0$ e $P1$. As mensagens $m_1$ e $m_2$ são mensagens trocadas entre os processos da aplicação. Cada mensagem recebida pelo processo da aplicação gera uma requisição ao *event logger*, representado por $Rx$ em cada abordagem. A abordagem centralizada conta apenas com um único *event logger* ($R_0$) e assim não tolera falhas. As abordagens baseadas em Paxos contam com 3 *event loggers* replicados e assim podem tolerar uma falha (isto é, $f = 1$). Considera-se que os coordenadores do Paxos executaram previamente a primeira fase do protocolo e podem seguir com a segunda fase ao receber um comando.

O Paxos clássico baseia-se na replicação máquina de estado clássica. Os processos da aplicação são *proposers* e as réplicas de *event logger* são os *acceptors* e os *learners*. Cada processo da aplicação submete comandos ao coordenador, um processo entre os *acceptors* para registrar os determinantes. O coordenador recebe o comando, executa o Paxos para registrar o comando em um quórum de réplicas e responde ao processo da aplicação (ver Figura 2). Por exemplo, um determinante é recebido pelo coordenador ($R_0$), que também é um *acceptor*. O coordenador executa diretamente a segunda fase do Paxos. Ao receber a resposta de um quórum de *acceptors*, no caso $R_0$ e $R_1$, o consenso é finalizado e $R_0$ responde ao processo $P0$. Em "boas execuções" (isto é, na ausência de falhas de processos) um determinante é registrado em quatro passos de comunicação e $2f + 2$ troca de mensagens ponto-a-ponto. Em contraste, um *event logger* centralizado pode registrar eventos após dois passos de comunicação e duas trocas de mensagens.

No segundo protocolo, Paxos Paralelo, há um sequência separada de execuções do Paxos associada a cada processo da aplicação. Isso significa que cada processo possui seu próprio conjunto de réplicas o que permite certas otimizações. Primeiro, uma vez que cada processo tem sua própria sequência de execuções do Paxos, o processo não compete com outros processos da aplicação nas execuções de Paxos e assim não há necessidade de um único coordenador que recebe requisições de diferentes processos. Em boas execuções, o processo é o único *proposer* em sua sequência de Paxos. Uma segunda otimização é a seguinte: ao usar diferentes conjuntos de réplicas o desempenho não é mais limitado pelo o que o coordenador e os *acceptors* podem suportar. Terceiro, é possível co-alocar os processos da aplicação e o *acceptor*-coordenador no mesmo processador. Este esquema pode registrar um determinante após dois passos de comunicação e $2f$ mensagens.

Um *event logger* implementado com Paxos Paralelo apresenta o mesmo número

de passos de comunicação de um *event logger* centralizado, com a vantagem de tolerar um número configurável de falhas e apresentar desempenho escalável. Quando configurado para tolerar uma falha ($f = 1$), o número de trocas de mensagens é o mesmo de um *event logger* centralizado. Além disso, para economizar recursos, *acceptors* não co-alocados com os processos da aplicação podem ser hospedados em um mesmo nodo físico. Por exemplo, um único nodo pode hospedar todos esses *acceptors*. Nesse caso, o Paxos Paralelo usa o mesmo número de nodos requerido pelo abordagem centralizada.

Ao se recuperar de uma falha, um processo de aplicação deve recuperar todos os seus determinantes. Na abordagem centralizada, o processo da aplicação contacta o *event logger*. Com a abordagem replicada, isso é feito contactando um quórum de *acceptors*.

## 4. Implementação

Esta seção descreve as implementações do protocolo de registro de mensagens pessimista em MPI e dos *event loggers*. A implementação do protocolo de registro de mensagens proposto se apoia na especificação ULFM (*User Level Failure Mitigation*). A ULFM é o mais recente esforço do MPI-Fórum para padronizar a semântica de tolerância a falhas em MPI [Bland et al. 2013], pois, tradicionalmente, as aplicações MPI abortam toda a aplicação se um único processo falhar. Em [Gamell et al. 2014] é apresentado o esquema que o MPI em conjunto com a ULFM emprega para lançar novos processos ao detectar um processo falho. A comunicação entre os processos em MPI é FIFO e implementada através do protocolo TCP.

A implementação do protocolo de registro de mensagens foi encapsulada em uma biblioteca para facilitar o seu uso por uma aplicação MPI. A seguir destacamos brevemente as principais primitivas. A primitiva `framework_init()` inicia o interceptador de eventos (descrito mais adiante) e as estruturas de dados necessárias, como os contadores de mensagens enviadas e recebidas. Essa primitiva também define parâmetros como, por exemplo, a periodicidade do *checkpointing* e o seu local de armazenamento. No entanto, a sua principal função é classificar o processo MPI como um processo original ou um novo processo em recuperação. A primitiva `aloc_data()` define quais variáveis da aplicação serão incluídas no *checkpointing*. Se o processo MPI for um novo processo em recuperação, a primitiva `recovery()` é responsável por restaurar o *checkpoint* do processo, solicitar os seus determinantes do *event logger* e orquestrar a recuperação conforme definido na Seção 3. A primitiva `chkp()` é responsável por realizar o *checkpointing* do processo. A primitiva `garbage_colletor()` realiza a limpeza periódica das mensagens antigas mantidas em memória. A biblioteca foi programada em linguagem C.

A interface PMPI [MPI Forum 2015] é utilizada para interceptar as chamadas MPI de forma transparente para o desenvolvedor. Dessa forma, é possível inserir código antes e depois de cada chamada MPI. Os eventos não-determinísticos são detectados de acordo com o refinamento proposto em [Bouteiller et al. 2010]. Toda primitiva de recepção, como `MPI_Recv`, que aguarda uma mensagem de uma fonte qualquer (`MPI_ANY_SOURCE`) gera um evento não-determinístico. Assim também acontece com primitivas que verificam se há uma mensagem pronta para ser recebida, como a `MPI_Probe`. Além das primitivas de recepção, as primitivas não bloqueantes que inspecionam se o recebimento da mensagem foi concluído como, por exemplo, `MPI_Waitsome`, `MPI_WaitAny`, `MPI_Test` também geram eventos não-

determinísticos. Cada mensagem enviada é copiada para a memória e armazenada em uma tabela *hash*. A biblioteca khash[1] foi usada como implementação de uma tabela *hash*. A implementação PMPI foi realizada em C.

Um interceptador de eventos, implementado através de uma *thread*, é responsável por submeter e recuperar os determinantes do *event logger*. Ao iniciar, cada processo MPI instancia um interceptador que se conecta ao *event logger*. O interceptador pode ser configurado para submeter determinantes de forma síncrona ou assíncrona. No modo síncrono, após submeter um determinante, o processo da aplicação aguarda a confirmação do *event logger* antes de prosseguir. No modo assíncrono o processo da aplicação pode prosseguir antes de receber a confirmação. Por padrão, todos os resultados foram obtidos usando o modo síncrono. O protocolo pessimista permite atrasar o recebimento das confirmações até o envio de uma mensagem a outro processo. Essa otimização foi aplicada na implementação do protocolo proposto. Foram implementadas três versões de *event loggers*: centralizado, baseado no Paxos Clássico e baseado no Paxos Paralelo. O interceptador e os *event loggers* foram implementados em C usando a biblioteca libevent versão 2.022[2]. A biblioteca libpaxos[3] versão 3 foi utilizada para desenvolver os *event loggers* baseado em Paxos.

## 5. Resultados Experimentais

Os resultados experimentais avaliam os *event loggers* propostos, incluindo o procedimento de recuperação. Os *event loggers* baseados em consenso são comparados à implementação do *event logger* centralizado. A latência e a vazão, em termos do número de determinantes armazenados por segundo, para cada um dos *event loggers* são apresentados em quatro execuções MPI: a aplicação AMG[4] (Algebraic Multigrid Solver), os *kernels* LU (Lower-Upper Gauss-Seidel solver) e MG (Multi-Grid solver) do NAS *parallel benchmark* versão 3.2.[5], e o algoritmo de árvores de cortes de Gusfield[6]. O algoritmo de Gusfield também foi empregado para avaliar a recuperação. As aplicações foram executadas usando a implementação OpenMPI/ULFM 1.1[7]. Os experimentos foram conduzidos em um *cluster* dedicado que consiste de 40 nodos, cada um com dois processadores Intel(R) Quad-Core Xeon L5420 2.5 GHz e 8 Gbytes de RAM. Os nodos estão conectados através de uma rede *Gigabit Ethernet*.

### 5.1. Os *Event Loggers*

Primeiramente, o desempenho dos *event loggers* foram avaliados através de uma simples aplicação MPI onde cada processo submete um novo determinante ao *event logger* ao receber a confirmação de que o determinante submetido anteriormente está armazenado. Nessa aplicação, os processos MPI não trocam mensagens entre si e os determinantes têm o tamanho de 50 bytes. A Figura 3 apresenta a vazão e a latência (ambas em milissegundos) conforme o número de processos aumenta até 128. Nesse experimento há um nodo

[1] http://attractivechaos.awardspace.com/khash.h.html
[2] http://libevent.org/
[3] https://bitbucket.org/sciascid/libpaxos
[4] https://codesign.llnl.gov/amg2013.php
[5] https://www.nas.nasa.gov/publications/npb.html
[6] http://www.deinfo.uepg.br/ jcohen/parallel-cuttree.html
[7] https://bitbucket.org/icldistcomp/ulfm/downloads

dedicado que hospeda o *event logger* centralizado. O *event logger* baseado em Paxos clássico conta com um *proposer* em um nodo dedicado e três *acceptors* (isto é, $f = 1$) e três *learners*; cada *acceptor* e *learner* executa em 1 nodo dedicado. A configuração do Paxos Paralelo é a seguinte: cada sequência de Paxos usa um *proposer*, que também é *learner*, e três *acceptors* (isto é, $f = 1$). O *proposer/learner* está hospedado junto ao processo MPI. As sequências de *acceptors* estão em três nodos dedicados, com cada *acceptor* da sequência em um nodo. Os processos MPI estão executando nos 40 nodos do *cluster*.

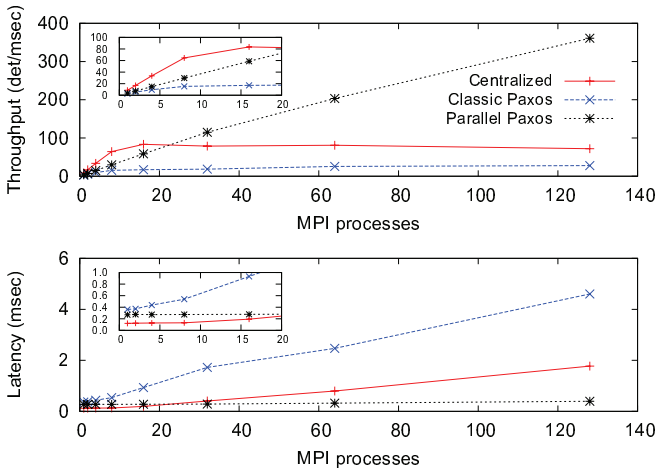


**Figura 3. Vazão e latência para três as abordagens de *event loggers*.**

A Figura 3 apresenta a vazão e latência para os três *event loggers* implementados. O pequeno gráfico interno é um zoom dos 20 primeiros processos. O *event logger* centralizado atinge a vazão máxima de 83 determinantes por milissegundo (det/ms) com 16 processos. O *event logger* baseado em Paxos Clássico alcança a vazão máxima de 28 det/ms com uma latência de 4,3 ms com 128 processos MPI. Já o *event logger* baseado em Paxos Paralelo nunca chega a saturar nesse experimento: a vazão aumenta proporcionalmente ao número de processos e a latência permanece aproximadamente constante, abaixo de 4 ms. Com 128 processos, a abordagem baseada em Paxos Paralelo tem 5 vezes a vazão da abordagem centralizada e uma latência muito menor. A seguir apresentamos resultados de desempenho dos *event loggers* coletados durante a execução de 4 aplicações MPI (Figura 4). A primeira barra corresponde ao desempenho da aplicação sem o uso de qualquer *event logger* e sem registrar qualquer tipo de evento. A segunda barra indica os resultados para a abordagem centralizada e as duas últimas para os *event loggers* baseados em Paxos Clássico e Paxos Paralelo, respectivamente. Para os três primeiros resultados (AMG, LU e MG) os *event-loggers* são configurados conforme descrito no início desta seção. Foram usados 16, 32, 64 e 128 processos MPI, exceto para o último gráfico (Algoritmo de Gusfield) que foi executado com 4, 8, 16 e 32 processos MPI.

A aplicação AMG possui o seguinte não-determinismo em seu código: todas as chamadas `MPI_Iprobe` usam a marcação `MPI_ANY_SOURCE` e somente uma chamada `MPI_Recv`, entre muitas, usa tal marcação. A aplicação também possui primitivas de verificação `MPI_Test` e `MPI_Testall`. Embora as primitivas de verificação gerem muitos eventos não-determinísticos, um determinante somente é gerado quando a mensagem está pronta para ser recebida. O número de vezes em que a mensagem não estava

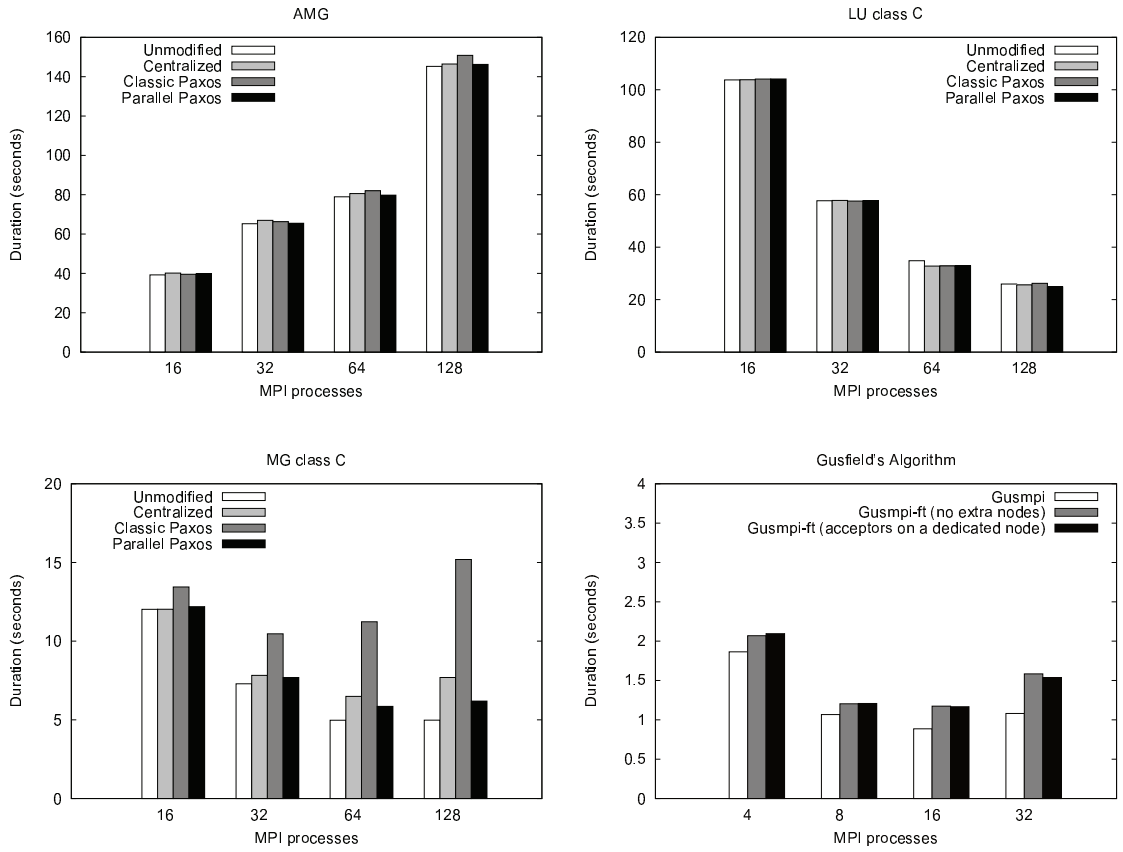


**Figura 4. Quatro aplicações MPI usando os *event loggers* propostos.**

pronta para ser recebida é incluído no determinante. Considerando todos os eventos gerados, os eventos não-determinísticos da aplicação AMG correspondem a menos de 1% do total. No entanto, há picos onde a vazão atinge até 70 det/ms. A Figura 4 (AMG) apresenta os resultados obtidos. O pior resultado foi obtido usando a abordagem do Paxos Clássico para 64 e 128 processos, houve uma sobrecarga de aproximadamente 3,8%. A abordagem centralizada apresentou uma sobrecarga de aproximadamente 2% para 16, 32 e 64 processos. O *event-logger* usando o Paxos Paralelo teve a menor sobrecarga, abaixo de 1,3% para todos os números de processos, sendo inclusive melhor que o centralizado.

Os *kernels* LU e MG são os únicos da versão 3.2 do NAS que apresentam eventos não-determinísticos. Em ambos, os únicos eventos não-determinísticos são `MPI_Recv` com a marcação `MPI_ANY_SOURCE`. No NAS, o tamanho das entradas para os experimentos é classificado em classes. Resultados da classe C são apresentados porque como o tamanho da entrada da classe C é menor do que da classe D, a taxa de comunicação é maior e, consequentemente, maior é a carga nos *event-loggers*. Enquanto o *kernel* LU gera menos de 1% de eventos não-determinísticos, no kernel MG quase 100% dos eventos são não-determinísticos, considerando somente as recepções. Embora as aplicações MG e AMG resolvam problemas similares, a MG possui muito mais não-determinismo em seu código. Na verdade, o não-determinismo no código é frequentemente uma escolha do desenvolvedor. Conforme apresenta a Figura 4, os *event-loggers* virtualmente não impactam no desempenho do LU. Ao contrário, o desempenho do MG é impactado pelos *event loggers*. Enquanto a abordagem usando o Paxos clássico apresenta uma sobrecarga

de 125% e 200% para 64 e 128 processos, o *event logger* centralizado mostra uma sobrecarga de 31% e 55% para a mesma execução. O *event logger* usando o Paxos Paralelo tem uma sobrecarga de 17,71% e 24,26% para 64 e 128 processos, respectivamente.

O algoritmo de Gusfield, último gráfico da Figura 4, foi executado com 4, 8, 12 e 32 processos MPI para uma entrada de 2000 vértices e 21.990 arestas. Esta é uma aplicação do tipo mestre-escravo. Somente o mestre gera eventos não-determinísticos, todas as suas recepções usam a marcação `MPI_ANY_SOURCE`. Nesse experimento foram avaliados a aplicação sem modificações e com a abordagem Paxos Paralelo. Porém, há duas configurações distintas do Paxos Paralelo. Na primeira, cada sequência de *acceptor* usa 3 nodos dedicados, como nos experimentos anteriores. Na segunda configuração, os *acceptors* são co-alocados com os processos da aplicação e, assim, não há nodo extra usado. Como é possível perceber, não houve diferença significativa de sobrecarga entre as duas configurações de Paxos Paralelo nesse experimento.

### 5.2. Recuperação da Aplicação

O protocolo proposto, incluindo a recuperação, foi avaliado usando o algoritmo de Gusfield. Todas as primitivas descritas na seção 4 são inseridas no código. O algoritmo foi levemente modificado para lidar com cenários de falha e recuperação. Basicamente, o algoritmo de Gusfield executa cálculos de fluxos máximos em um laço de repetição. O mestre solicita que seus escravos calculem uma parte do problema e ao receber o resultado de um escravo envia uma nova parte do problema a esse escravo. A primitiva `chkp()`, por exemplo, foi inserida no fim do laço de repetição. Já a primitiva `recovery` é inserida antes do laço. Foram incluídos no *checkpointing*, entre outros, os vetores que guardam os valores calculados a cada iteração.

A Figura 5 apresenta a execução do algoritmo para 4, 8, 16, 32 e 64 processos em uma entrada com 1000 vértices e 99.900 arestas. O *event logger* baseado na abordagem Paxos Paralelo foi empregado na execução. A primeira barra corresponde à execução do algoritmo original, isto é, sem modificações. A segunda barra apresenta a sobrecarga causada pelo registro dos determinantes no *event logger*. A terceira barra diz respeito a um cenário de falha e recuperação do mestre. Nessa execução houve a inserção de falhas aleatórias durante a execução. A quarta barra se refere à falha e recuperação dos escravos. Durante a execução um dos escravos falhava aleatoriamente. A quinta barra representa a adição ao cenário de falha do mestre de um *checkpointing* durante a metade da execução. A sexta barra adiciona a esse último cenário a coleta de mensagens antigas, configurado para executar pelo menos uma vez durante a execução.

De acordo com o gráfico da Figura 5 é possível perceber que o tempo que a aplicação leva para se recuperar de forma consistente usando o protocolo proposto não é significante. Lembrando que o processo de recuperação inclui lançar um novo processo, recuperar os seus determinantes do *event logger*, ler o seu *checkpoint* e refazer a computação perdida. Tendo como base a aplicação sem qualquer modificação (primeira barra da Figura 5), quando o mestre se recupera de uma falha (terceira barra), a maior sobrecarga observada foi de 15,16% para 32 processos. Porém, quando o mestre realiza um *checkpointing* (quinta barra), a sobrecarga para esse mesmo caso cai para 13,2%. O mesmo ocorre no cenário de 64 processos, a sobrecarga com o mestre se recuperando cai de 9,42% para 8,62%. Ou seja, o tempo que o mestre leva realizar um *checkpoint* é compensado no processo de recuperação. O tempo de recuperação dos escravos (quarta barra)

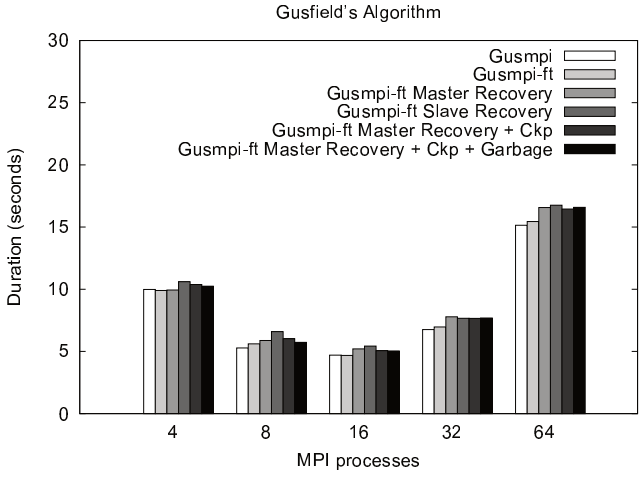


**Figura 5. Recuperação do algoritmo Gusfield.**

foi superior ao tempo de recuperação do mestre nesse experimento porque os escravos repetem a sua computação desde a última falha. Além disso, o mestre precisa se certificar que não deixou de receber mensagens enviadas por algum outro escravo devido a detecção da falha. É possível otimizar a implementação nesse cenário porque os escravos não precisam guardar um estado prévio. Nesse caso, ao detectar a falha de um escravo o mestre faria um *checkpoint* do seu estado e simplesmente lançaria um novo escravo que não precisaria refazer a sua computação. É possível observar também que ao adicionar a limpeza de mensagens antigas (última barra) não houve impacto significativo no tempo de execução (se comparado a penúltima barra).

## 6. Conclusão

Neste trabalhos apresentamos um protocolo pessimista para registro de mensagens construído com um *event logger* distribuído e tolerante a falhas baseado em consenso. Dois *event logger* baseados no algoritmo Paxos foram implementados e avaliados. Ao empregar o algoritmo de consenso Paxos, os *event logger* propostos garantem a propriedade de segurança mesmo se o sistema for assíncrono e o progresso apesar de falhas de processos. Nos experimentos realizados, usando as aplicações AMG, LU, MG e algoritmo de Gulsfield, o *event logger* baseado em Paxos Paralelo apresentou desempenho superior a abordagem centralizada. A implementação do protocolo foi encapsulada em uma biblioteca para facilitar o seu uso por uma aplicação MPI. O algoritmo de Gusfield foi levemente modificado empregando as primitivas implementadas. Resultados demonstram que perante falhas os processos da aplicação se recuperam eficientemente. Trabalhos futuros incluem avaliar o protocolo proposto em aplicações MPI de larga escala.

## Referências

Bland, W., Bouteiller, A., Hérault, T., Bosilca, G., and Dongarra, J. (2013). Post-failure recovery of MPI communication capability: Design and rationale. *International Journal of HPC Applications*, 27(3):244–254.

Bouteiller, A., Bosilca, G., and Dongarra, J. (2010). Redesigning the message logging model for high performance. *Concurrency and Computation: Practice and Experience*, 22(16):2196–2211.

Bouteiller, A., Collin, B., Herault, T., Lemarinier, P., and Cappello, F. (2005). Impact of event logger on causal message logging protocols for fault tolerant mpi. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 97–97.

Bouteiller, A., Hérault, T., Krawezik, G., Lemarinier, P., and Cappello, F. (2006). MPICH-V project: A multiprotocol automatic fault-tolerant MPI. *International Journal of HPC Applications*, 20(3):319–333.

Bouteiller, A., Ropars, T., Bosilca, G., Morin, C., and Dongarra, J. (2009). Reasons for a pessimistic or optimistic message logging protocol in MPI uncoordinated failure, recovery. In *Cluster*.

Cappello, F., Guermouche, A., and Snir, M. (2010). On communication determinism in parallel HPC applications. In *ICCCN*.

Egwutuoha, I. P., Levy, D., Selic, B., and Chen, S. (2013). A survey of fault tolerance mechanisms and checkpoint/restart implementations for hpc systems. *The Journal of Supercomputing*, 65(3):1302–1326.

Elnozahy, Alvisi, Wang, and Johnson (2002). A survey of rollback-recovery protocols in message-passing systems. *CSURV: Computing Surveys*, 34.

Fagg, G. E. and Dongarra, J. (2000). FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Recent advances in PVM and MPI*, LNCS.

Gamell, M., Katz, D., Kolla, H., Chen, J., Klasky, S., and Parashar, M. (2014). Exploring automatic, online failure recovery for scientific applications at extreme scales. In *SC*.

Johnson, D. B. and Zwaenepoel, W. (1987). Sender-based message logging. In *FTCS*.

Kshemkalyani, A. D. and Singhal, M. (2011). *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, Cambridge, UK.

Lamport (2001). Paxos made simple. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 32.

Lamport, L. (2006). Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125.

Lifflander, J., Meneses, E., Menon, H., Miller, P., Krishnamoorthy, S., and Kale, L. V. (2014). Scalable replay with partial-order dependencies for message-logging fault tolerance. In *CLUSTER*.

Liu, X., Xu, X., Ren, X., Tang, Y., and Dai, Z. (2013). A message logging protocol based on user level failure mitigation. In *ICA3PP*.

MPI Forum (2015). Document for a standard message-passing interface 3.1. Technical report, University of Tennessee, http://www.mpi-forum.org/docs/mpi-3.1.

Ropars, T. and Morin, C. (2009). Active optimistic message logging for reliable execution of MPI applications. In *Euro-Par*.

Ropars, T. and Morin, C. (2010). Improving message logging protocols scalability th-rough distributed event logging. In *Euro-Par*.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(3):299.