Auto-ajuste de parâmetros em protocolo de comunicação em grupo: estudo de caso do Isis

Adnilson Costa Garrido Junior[1]
Orientador - Allan Edgard Silva Freitas
Grupo de Pesquisa em Sistemas Distribuídos, Otimização, Redes e Tempo-Real (GSORT)
Especialização em Computação Distribuída e Ubíqua
Instituto Federal de Educação, Ciência e Tecnologia da Bahia (IFBA)
Email: adnilsongarrido@gmail.com

Resumo
Adaptabilidade é um fator essencial em Sistemas Distribuídos, e tem sido um tópico cada vez mais discutido e estudado. A complexidade dos atuais sistemas computacionais, somado a ambientes cada vez mais dinâmicos, tem exigido que as aplicações se comportem de forma adaptável. O objetivo deste trabalho consiste em implementar o protocolo de comunicação em grupo, Isis, a fim de avaliar seu comportamento em diversos cenários. Bem como, realizar mudanças em seu funcionamento, a fim de torná-lo adaptável a carga do ambiente.

Abstract
Adaptability is a key issue in Distributed Systems, and has been a growingly discussed and studied topic. The complexity of the current computational systems, added to ever more dynamic environments, has required that the applications behave adaptively. The aim of this study is to implement the group communication protocol, Isis, in order to evaluate its behavior in different scenarious. As well make changes in its operation, making it adaptable to load the environment.

Palavras-chave: Adaptabilidade, Sistemas distribuídos, Comunicação em grupo, Protocolo, Isis.

SUMÁRIO

I. INTRODUÇÃO

No início dos anos noventa, com o fenômeno da Internet, a computação distribuída passou a ter definitiva relevância. E a proliferação da Computação Distribuída (ou dos Sistemas Distribuídos) sobre a Internet trouxe consigo uma crescente necessidade de comunicação entre ambientes computacionais heterogêneos (envolvem uma multiplicidade de recursos e tecnologias diferentes) [1]. Conforme [2], um caso particular de comunicação ocorre quando uma mensagem deve ser enviada a diversos nós (computadores) na rede. Para atender essa necessidade de envio de um para muitos, de forma transparente, e com confiabilidade, foi criado o novo paradigma chamado Comunicação em grupo.

Segundo [3], a comunicação em grupo é introduzida como a infra-estrutura (serviço) de comunicação que permite a implementação das diferentes técnicas de replicação, destacando-se a necessidade de primitivas de comunicação com propriedades de ordenação bem definidas para implementação dessas técnicas. Sendo assim, comunicação em grupo é, para [4], hoje um tema muito relevante em computação distribuída, surgindo ao longo dos anos como uma área com forte sinergia entre teoria e prática: comunicação em grupo é necessária para construção de sistemas distribuídos, e sua importância teórica é devido aos complexos problemas abordados na literatura.

Nesse contexto, contribuições teóricas podem ser definidas como as contribuições para abstrações e paradigmas, contribuições para modelos computacionais e especificação de problemas e, claro, contribuições para algoritmos. Em relação a aspectos práticos no contexto de comunicação em grupo, podemos citar a eficiência, a estrutura da arquitetura do sistema, a flexibilidade do sistema, exatidão, e claro entendimento das propriedades do sistema. Eficiência é claramente importante: é inapropriado ter um sistema de comunicação em grupo ineficiente. Algoritmos e paradigmas podem contribuir para a eficácia: um novo algoritmo, ou a aplicação de um novo paradigma, pode aumentar a eficiência. Captações e algoritmos podem contribuir para uma arquitetura limpa e flexível do sistema. [3]. Com isso, no desenvolvimento de bons protocolos de comunicação em grupo há aspectos teóricos e práticos, sendo o foco desse trabalho os aspectos práticos.

Há várias contribuições na área de desenvolvimento de protocolos de comunicação em grupo, dentre os quais podemos destacar o Isis, Amoeba e o *Timed Causal Block*. Este trabalho estudará aspectos do protocolo clássico do Isis, estático, propondo adaptabilidade.

Isis se tornou popular por ser executado em diversas plataformas, linguagens de programação e protocolos [5]. Além disso, o Isis serviu de base para o desenvolvimento do *JGroups* (*toolkit* para a comunicação em grupo confiável, escrito em Java) [6]. Um de suas características é ser estático, ou seja, o programador fica responsável em escolher a primitiva a utilizar: *ABCAST* (é uma primitiva de comunicação do tipo bloqueante e síncrona) ou *CBCAST* (é uma primitiva de comunicação do tipo não bloqueante e assíncrona). Essa escolha é feita levando-se em consideração o desempenho global do sistema e os requisitos da aplicação, por exemplo, uma dada aplicação, como a replicação ativa, baseada em entrega uniforme e ordem total pode requerer o uso de *ABCAST*. Porém, a confiabilidade do sistema não é levada em consideração na escolha da primitiva, outro problema que surge é o fato de a análise de desempenho, feita pelo programador, ser subjetiva.

O Isis é um dos protocolos mais populares e utilizado por diversos programadores, inclusive inexperientes, com isso, o desempenho do protocolo pode ser comprometido devido a má configuração.

Este trabalho tem como objetivo principal implementar o Isis e avaliar o seu desempenho, comparando sua performance com um número variado de processos e em diferentes cenários de configuração. Bem como propor uma adaptação do protocolo de acordo com a carga do ambiente.

Esse artigo está estruturado da seguinte forma. A Seção II faz uma breve discussão sobre sistemas distribuídos, além de apresentar sua relação com tolerância a falhas. A Seção III esclarece noções importantes sobre comunicação em grupo, abordando sua importância, e descrevendo modelos, primitivas e projetos para comunicação em grupo. Estes conceitos são importantes pois dão fundamentação teórica para compreender o restante do artigo. A Seção IV apresenta um conceito importante na área de sistemas distribuídos: Sistemas Adaptáveis. A Seção V apresenta o protocolo

utilizado como estudo de caso deste trabalho. Abordando, principalmente, suas primitivas de comunicação. A Seção VI apresenta e discute os resultados deste trabalho, mostrando alguns detalhes de como foi a implementação do protocolo, bem como as mudanças realizadas em seu funcionamento a fim de torná-lo adaptável. Além disso, é mostrado o ambiente de execução e as métricas utilizadas na análise de desempenho. Finalizando, a Seção VII faz um breve resumo, e entendimento, sobre todo o projeto, alem de discutir trabalhos futuros.

II. SISTEMAS DISTRIBUÍDOS

Os sistemas computacionais tem se tornado cada vez mais complexos. Como consequência, a probabilidade de problemas nesses sistemas tem aumentado ao longo dos anos. Para solucionar estes problemas, pesquisas têm sido realizadas com o objetivo de maximizar a confiabilidade e desempenho desses sistemas [6].

A redução do preço dos computadores e sua grande disseminação, a partir do surgimento de microcomputadores na década de 80, impulsionaram a criação de redes de computadores. Essas redes foram criadas, principalmente, devido à necessidade de compartilhamento de recursos entre os computadores. Inicialmente, pequenas redes locais foram sendo instaladas nas instituições, e, posteriormente, a partir de meados da década de 90 no Brasil, elas foram se interligando à Internet. Hoje, as redes de computadores estão em todos os lugares e a Internet pode ser entendida como um grande e disperso sistema distribuído que permite aos seus usuários acessarem serviços, tais como *www*, email e transferência de arquivos, entre muitos outros [7].

Um sistema distribuído é aquele em que os componentes localizados em computadores em rede se comunicam e coordenam suas ações apenas pela passagem de mensagens. Essa definição leva às seguintes características, significativas em sistemas distribuídos: simultaneidade entre os componentes, a falta de um relógio global e falhas independentes de componentes. Podemos citar vários exemplos de aplicações distribuídas modernas, como [6]:

- Pesquisa na web: indústria em crescimento desde a década passada. Dados recentes indicam que o número global de pesquisas aumentou para mais de 10 bilhões por mês. A tarefa de um motor de busca web é indexar todo o conteúdo do *World Wide Web*, que abrange uma ampla gama de estilos de informação, incluindo páginas web, fontes multimídia e livros digitalizados. Esta é uma tarefa muito complexa, pois as estimativas atuais indicam que a Web é composto de mais de 63 bilhões de páginas e um trilhão de endereços exclusivos. Tendo em vista que a maioria dos motores de busca analisa todo o conteúdo da web e, em seguida, realiza o processamento neste enorme banco de dados, logo, verifica-se que esta tarefa representa um grande desafio para os projetistas de sistemas distribuídos.

- Jogos *multiplayer online*: um número muito grande de usuários interagem através da Internet com um mundo virtual. Esta interação tem aumentado significativamente em termos de sofisticação. O número de jogadores também tem aumentado, com sistemas capazes de suportar mais de 50.000 jogadores *online*.

  A engenharia de jogos *online* representa um grande desafio na área de sistemas distribuídos, especialmente por causa da necessidade de tempo de resposta rápido. Outros desafios incluem a propagação em tempo real de eventos para muitos jogadores, e manter uma visão coerente do mundo compartilhado. Estes, portanto, são excelentes exemplos dos desafios enfrentados pelos projetistas de sistemas distribuídos.

- Sistemas de negociação financeira: o setor financeiro utiliza há muito tempo tecnologias de sistemas distribuídos, com sua necessidade, em particular, em ter acesso em tempo real a uma ampla gama de fontes de informações (por exemplo, preço de ações e tendências).

Em computação distribuída é notório que processos devem se comunicar entre si [2]. Tais processos podem formar um grupo, que nada mais é que um conjunto de processos que se comunicam entre si. Logo, quando uma mensagem é enviada ao grupo, todos os membros desse grupo recebem essa mensagem.

Sistemas distribuídos que implementam comunicação

em grupo podem ser divididos em duas categorias: grupos fechados, no qual somente integrantes do grupo podem se comunicar entre si; e grupos abertos, onde um processo fora do grupo pode enviar mensagem a este grupo. Tais sistemas se baseiam em difusão atômica, uma primitiva que garante que as mensagens sejam entregues de forma ordenada, e a todos os destinatários (que não tenham falhado), ou a nenhum[2].

Um sistema distribuído deve prover operação continuada, com apenas pequena queda de desempenho, mesmo na presença de qualquer tipo de falha. Apesar de se conhecer um bom conjunto de técnicas de tolerância a falhas, sua aplicação a sistemas distribuídos, principalmente os de tempo real, ainda é muito recente e seus resultados são muitas vezes insatisfatórios [8].

A. Tolerância a Falhas

Segundo [8], sistemas distribuídos são formados por um conjunto de processadores independentes. Esses sistemas se diferenciam de computadores paralelos pelo fraco acoplamento entre os nodos, ou seja, os elementos de um sistema distribuído não compartilham a mesma memória. Toda a interação deve ser feita por troca de mensagens através de canais de comunicação. Os nodos de um sistema distribuído também não têm acesso a um relógio global, portanto não possuem uma base de tempo comum para ordenação de eventos. Além disso, sistemas distribuídos são geralmente construídos com elementos não homogêneos e assíncronos.

Muitos tipos de falhas são freqüentemente encontrados na literatura de sistemas distribuídos. Conforme [9], as principais falhas encontradas são (estão ilustradas na figura 1):

- Falha do tipo *crash*: caracterizada por uma falha que causa a parada de um componente ou a perda de seu estado interno;

- Falha por omissão: caracterizada quando ocorre um componente não responde a determinadas entradas (abrange falhas de crash);

- Falha de temporização: caracteriza-se quando o componente responde muito cedo ou muito tarde. Conhecida também como falha de desempenho (engloba falha de omissão);

- Falha de resposta: caracteriza-se por computação incorreta, ou seja, o componente produz respostas incorretas para algumas entradas (não abrange as anteriores);

- Falhas bizantinas (arbitrárias ou maliciosas): é um conceito que abrange uma grande variedade de comportamentos faltosos, estes incluem programas que não seguem o protocolo correto, dados corrompidos, e até mesmo, comportamentos maliciosos que buscam violar a confiabilidade do sistema [10]. Para [9], é uma falha arbitrária que provoca um comportamento totalmente arbitrário e imprevisível do componente durante o defeito (engloba todas as classes de falhas).

Essas falhas, que podem ocorrer em sistemas distribuídos, afetam as trocas de mensagens entre processos. Tolerância a falhas em sistemas distribuídos pode ser alcançada com técnicas de: transação (introduzida no contexto de guardar periodicamente o estado da computação em um lugar estável) e replicação (permite a continuidade da execução em caso de falhas – mascaramento de falha). No que tange a replicação, suas duas principais técnicas são: replicação ativa e replicação passiva. Sendo a comunicação em grupo funciona uma camada de *middleware* entre a camada de transporte e a camada que implementa replicação [4].

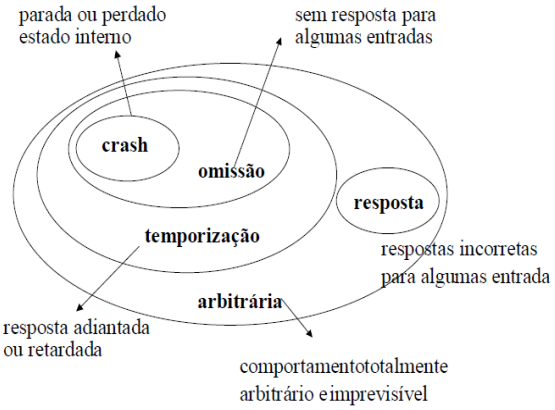

Figura 1- Modelo de falhas em sistemas distribuídos [8]

III.COMUNICAÇÃO EM GRUPO

A comunicação entre processos é um assunto de fundamental importância na construção de sistemas distribuídos. Os processos (componentes) devem se comunicar através de troca de mensagens. Uma chamada de procedimento remoto (CPR), ou *remote procedure call* (RPC), faz a comunicação entre duas partes: um processo cliente e um processo servidor [8]. Segundo [12], a seqüência de operações que ocorrem em uma chamada remota de procedimento são as seguintes (ver figura 2):

1. O cliente chama um procedimento local, chamado de *stub*. O *stub* do cliente constrói uma mensagem contendo o nome do procedimento a ser chamado e todos os parâmetros. O empacotamento requer serialização de todos os elementos de dados em formato de vetores de bytes;

2. Mensagens são enviadas do *stub* do cliente para o sistema remoto, através de *sockets*;

3. O *kernel* transfere as mensagens, através de algum protocolo, para o sistema remoto;

4. O *Stub* do servidor recebe as mensagens, desempacota os seus argumentos, e se necessário, converte-os a partir de um formato padrão de rede para uma forma específica de máquina;

5. A função no servidor retorna para o seu *stub* os resultados da função;

6. O *stub* do servidor converte os valores de retorno, se necessário, e empacota-os em uma ou mais mensagens de rede para enviar para o *stub* do cliente;

7. Através da rede, mensagens são enviadas de volta para o *stub* do cliente;

8. O *stub* do cliente lê as mensagens do kernel do local;

9. O *stub* do cliente, em seguida, retorna os resultados para a função do cliente, convertendo-os a partir da representação de rede para uma representação local, se necessário;

10. O código do cliente, em seguida, continua a sua execução.

A utilização de RPC é um exemplo de comunicação ponto a ponto, ou *unicast*. Entretanto, em certos tipos de sistemas é necessário que a comunicação seja de um para muitos.

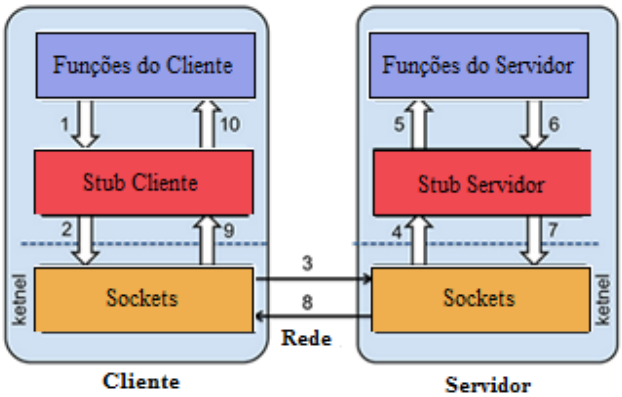

Figura 2- Etapas de uma chamada de procedimento remoto. Adaptado de [12]

A maioria dos atuais sistemas operacionais distribuídos são baseados em chamada remota de procedimento. Para muitas aplicações distribuídas e paralelas, no entanto, este modelo de comunicação emissor-receptor é inapropriado [13]. O que é necessário é que um processo qualquer possa enviar uma mensagem para um grupo de processos.

Tipicamente em um Sistema Distribuído, um número de computadores cooperam entre si para fornecerem um determinado serviço. O principal mecanismo utilizado em tal sistema é a troca de mensagens entre estes computadores através dos canais de comunicação. Muitas aplicações distribuídas (como sistemas de banco de dados replicados, *groupware*, etc.) envolvem comunicação freqüente entre os participantes do sistema. Nestes casos, uma abstração muito útil é a Comunicação em Grupo. Neste tipo de comunicação, um processo *p* envia urna mensagem *m* destinada a um grupo de outros processos encarregados de fornecerem um serviço potencialmente distribuído (ver figura 3) [14].

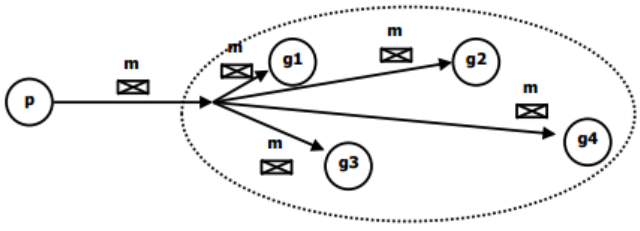

Figura 3- Processo *p* envia uma mensagem *m* ao grupo

Grupos são dinâmicos, eles podem ser criados e

destruídos. Processos podem entrar ou sair do grupo, e podem pertencer a vários grupos. Uma analogia para comunicação em grupo é o conceito de uma lista de discussão. Um remetente envia uma mensagem para uma parte da lista de discussão e vários usuários (membros da lista) recebem a mensagem. Grupos permitem que processos lidem com coleções de processos como uma abstração. Idealmente, um processo só deve enviar uma mensagem a um grupo e não precisa saber ou se importar quem são seus membros [15].

Comunicação em grupo é um serviço que provê estas primitivas. Um grupo nada mais é que o conjunto de processos com um identificador. Mensagens podem ser enviadas para os membros do grupo apenas referenciando o identificador do grupo. Comunicação em grupo é uma camada de *middleware* entre a camada de transporte e a camada que implementa replicação (ver figura 4).

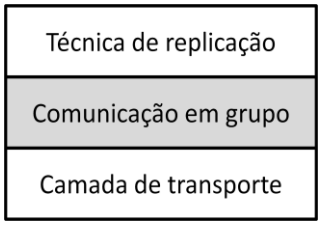

Figura 4 - Comunicação em grupo. Adaptado de [4]

A. Primitivas de comunicação em grupo

A comunicação em grupo torna-se cada vez mais necessário em sistemas distribuídos. Existem várias primitivas de comunicação, e cada uma possui diferentes garantias de ordenação e confiabilidade, dando suportes a diferentes requisitos de aplicações.

Em um sistema distribuído onde a comunicação entre os processos ocorre por difusão, seletiva ou não, a presença de falhas, tanto nas comunicações quanto nos processos, pode ocasionar perdas de mensagens, levando a inconsistências nos estados dos membros dos grupos. Sendo assim, as primitivas utilizadas para comunicação devem oferecer pelo menos confiabilidade e um nível de garantia em relação à ordenação das mensagens [16].

Nessa seção serão apresentados quatro tipos básicos de primitivas de comunicação em grupo:

- **Difusão confiável**

Esta primitiva garante que todas as mensagens enviadas a um grupo de processos serão recebidas por todos os nós não faltosos do grupo.

Seja *M* o conjunto de todas as mensagens envolvidas numa comunicação, e *G* um conjunto de processos, também chamado de grupo. A difusão confiável pode ser obtida através de duas primitivas: *multicast(G, m)* e *deliver(m)*, onde *m* é uma mensagem pertencente ao grupo M. As primitivas são definidas da seguinte forma:

- *multicast(G, m)*: A mensagem *m é* transmitida para todos os processos do grupo *G*.

- *deliver(m)*: A mensagem *m* é entregue.

Esta primitiva possui as seguintes propriedades:

- Validade: Se um processo correto executa *multicast(G, m)*, então algum processo correto em *G* eventualmente entregará *m* ou nenhum processo em *G* está correto.

- Acordo: Se um processo em *G* entrega uma mensagem *m*, então todos os processos corretos em *G* eventualmente entregarão *m.*

- Integridade: Para qualquer mensagem *m*, cada processo correto pertencente a *G* entrega *m* no máximo uma vez e apenas se *m* foi previamente difundida em *G* [16].

- **Difusão FIFO (First-In-First-Out)**

É uma difusão confiável com a propriedade de ordenação FIFO. Existem duas propriedades referentes à ordenação FIFO [16]:

- Ordenação FIFO Global: Se um processo difunde uma mensagem *m* antes de difundir *m',* então todos os processos corretos em *G* não entregam *m*' antes de entregar *m*.

- Ordenação FIFO Local: Se um processo difunde uma mensagem *m* em *G* antes de difundir *m'* em *G*, então todos os processos corretos em *G* não entregam *m*' antes de entregar *m*.

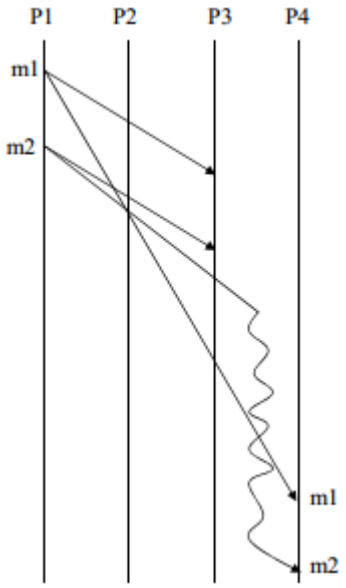

Figura 5- Difusão FIFO [17]

A figura 5 mostra um caso em que há difusão FIFO, note a relação FIFO é atendida tanto em P3 quanto em P4, pois em ambos os casos *m1* é entregue antes de *m2*.

- **Difusão Causal**

Esta primitiva é utilizada quando o contexto de uma mensagem *m* também dependa das mensagens entregues pelo seu remetente, ou seja, quando há a necessidade de um tipo de difusão que considere a ordem causal de eventos. Para [16], um evento *e* precede causalmente o evento *f* (denotado por *e -> f*), se e somente se:

1. O mesmo processo executa *e* e depois executa *f*, ou;
2. *e* é a difusão de uma mensagem e *f* é a entrega desta mensagem, ou;
3. Existe um evento *h*, tal que $e \rightarrow h$ e $h \rightarrow f$.

Observa-se, com essa definição, que a relação de precedência causal é transitiva e acíclica.

Segundo [16], uma difusão causal é uma difusão confiável e que satisfaz uma das seguintes propriedades:

- Ordenação Causal Global: seja uma mensagem *m* que precede causalmente uma mensagem *m'*, então qualquer processo correto não entrega *m'* antes de *m* ser entregue.
- Ordenação Causal Local: Se uma mensagem *m* é difundida em G, e esta precede causalmente uma mensagem *m'*, então qualquer processo correto em G não entrega *m'* antes de entregar *m*.

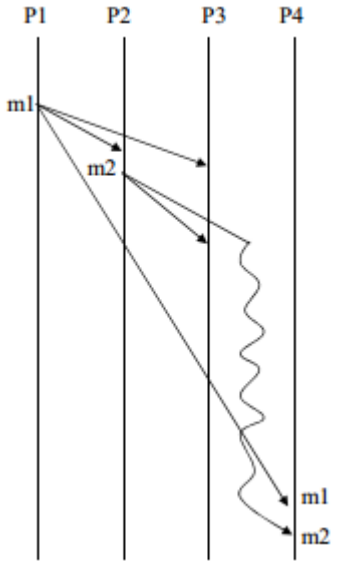

Figura 6- Difusão Causal [17]

- **Difusão Atômica**

Uma das mais importantes primitivas de comunicação em grupo é a difusão atômica, ou *atomic broadcast*. Esta primitiva é também conhecida como *total order broadcast*, ou simplesmente abcast. A primitiva garante que as mensagens sejam entregues de forma ordenada. Para especificar mais formalmente as propriedades do abcast, precisamos introduzir a seguinte notação:

- A difusão atômica de uma mensagem *m* para os membros de um grupo *G* é denotada por *abcast(G, m)*;
- A entrega de mensagem *m* é denotado por *adeliver (m)*.

Difusão atômica é definida como uma transmissão confiável e que satisfaz a seguinte propriedade:

- Ordem total uniforme: Se dois processos *p* e *q* entregam as mensagens *m* e *m'*, então *p* entrega *m* antes de *m'* se e só se *q* entrega *m* antes de *m'*.

É importante frisar a distinção entre as primitivas *abcast* e *adeliver* e envio/recepção (ver Figura 7). A semântica de enviar/receber é definida pela camada de

transporte. A semântica de *abcast/adeliver* é definido por *atomic broadcast*. Um protocolo de transmissão atômica usa a semântica de enviar/receber para fornecer a semântica de *abcast/adeliver* [4].

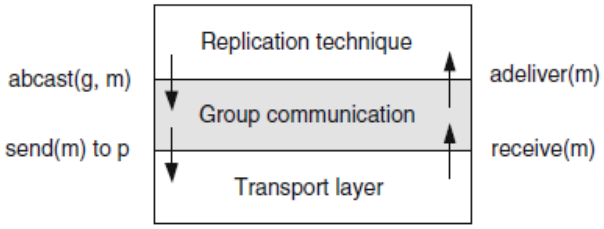

Figura 7- Enviar/receber versus abcast/adeliver[4]

B. *Modelos de Grupos*

Em relação a implementação, o modelo de comunicação entre os processos, que estão espalhados em redes locais ou de longa distância, é um dos principais fatores que diferenciam os sistemas distribuídos.

Segundo [4], os modelos de grupo de comunicação são:

- Grupo estático

  É um grupo no qual a quantidade de membros é constante ao longo do tempo. Apesar deste tipo de grupo ser simples, são muitos restritos. Por exemplo, se uma das replicas falhar, pode ser necessário substituí-la por uma nova réplica, a fim de manter o mesmo grau de replicação.

- Grupo dinâmico

  No grupo dinâmico os membros mudam ao longo do tempo. Para este tipo de grupo políticas de gerenciamento são necessárias para incluir e remover processos do grupo.

  Os processos no grupo estão sujeitos a diferentes tipos de falhas, sendo elas as falhas benignas e malignas. No primeiro o processo faz seu trabalho corretamente, ou não faz. Por exemplo, um erro no canal de comunicação é uma falha benigna. No segundo, também conhecido como falha bizantina, o processo ou canal se comporta de forma arbitrária. Um exemplo de falha benigna é a por *crash* (parada silenciosa), no qual o servidor pára de funcionar. No contexto de falhas do tipo *crash*, é feita a distinção entre *Crash-Stop* e *Crash-Recovery*. No primeiro o processo não tem acesso a uma área de armazenamento de estado, logo, um processo que falhe perde seu estado: após a recuperação, o processo é indistinguível de um processo recém-iniciado. No segundo, os processos possuem uma área de armazenamento de estado. Neste caso, um processo que falhe pode recuperar seu estado salvo mais recente.

- Combinação destes modelos

  Combinando estas dimensões têm-se vários modelos de grupos, como: *static crash-recovery groups* e *dynamic crashstop groups.*

C. *Projetos para comunicação em grupo*

Uma série de alternativas de projetos para comunicação em grupo estão disponíveis. Estas irão afetar a forma como os grupos se comportam e enviam mensagens. Algumas dessas alternativas são:

- Grupos abertos v*ersus* grupos fechados

  Com grupos fechados, apenas os membros do grupo podem enviar mensagem para o grupo. Isto é útil quando vários processos precisam se comunicar entre si na solução de um problema, tais como aplicações de processamento paralelo.

  O mecanismo alternativo é a de grupos abertos, onde os não-membros podem enviar uma mensagem para um grupo. Um exemplo do uso deste tipo de grupo é uma implementação de um servidor replicado (tal como um sistema de arquivo redundante) [15] (ver Figura 8).

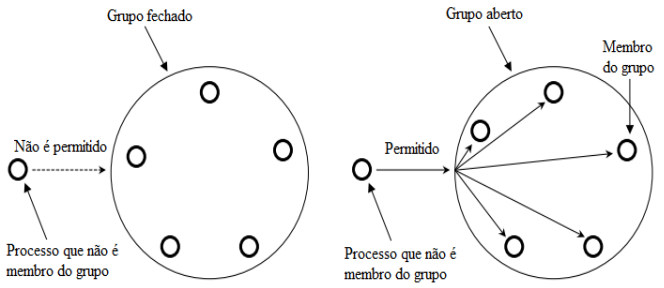

Figura 8- Grupos abertos versus grupos fechados

- Grupos igualitários (pares) *versus* grupos hierárquicos

Com os grupos igualitários, cada membro se comunica com o outro. Os benefícios são que este é um sistema descentralizado, simétrico, com nenhum ponto de falha. No entanto, a tomada de decisão pode ser complexa uma vez que todas as decisões devem ser tomadas coletivamente. A alternativa é grupos hierárquicos, em que um membro faz o papel de um coordenador de grupo. O coordenador toma decisões sobre quem vai executar uma requisição. A tomada de decisão é simplificada, pois é centralizada. A desvantagem é que este é um sistema centralizado, assimétrico e, portanto, tem um único ponto de falha [15] (ver Figura 9).

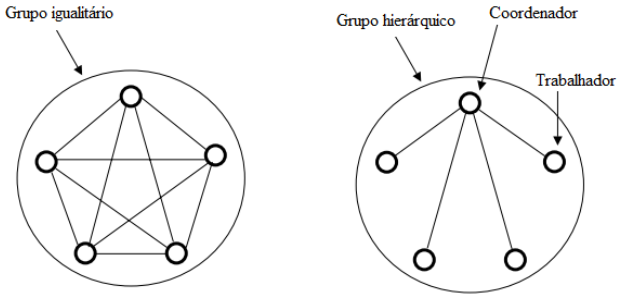

Figura 9- Grupos igualitários versus grupos hierárquicos

- Controle dos membros do grupo (servidor de grupo *versus* gerenciamento distribuído)

Se o controle de associação de grupo é centralizada, teremos um servidor do grupo que é responsável por obter todas as solicitações de adesão. Ele mantém um banco de dados de membros do grupo. Apesar de ser fácil de implementar, sofre com o problema de possuir um único ponto de falha. O mecanismo alternativo é o de gerir a associação de grupo de forma distribuída, onde todos os membros do grupo recebem mensagens anunciando novos membros (ou a saída de membros) [15].

IV. SISTEMA ADAPTÁVEL

Sistemas de distribuídos atuais operaram em ambientes altamente dinâmicos e de grande escala, como a Internet. Aplicações distribuídas são difíceis de serem construídas e mantidas, tendo em vista a complexidade e as características destes ambientes. Esta dificuldade é aumentada ainda mais quando o aplicativo tem requisitos de qualidade de serviço (QoS) relacionado a confiabilidade [18].

Um software adaptável normalmente é visto como um software que pode ter a sua configuração modificada em tempo de execução devido a alterações no seu ambiente. Mudanças na configuração com base em abordagens adaptativas podem ser utilizadas, através de mudanças nos padrões de comunicação adotados ou por novas necessidades da aplicação. A adaptabilidade pode envolver a troca de algoritmos e políticas em tempo de execução para adaptar a configuração do aplicativo para mudanças no ambiente do sistema.

Para [19], a iniciativa de computação autônoma da IBM visa sistemas computacionais capazes de se auto-gerenciarem, a tomada de decisão é feita a partir de políticas de alto nível, caracterizados por políticas auto-*, como auto-configuração, auto-cura (resiliência), auto-otimização, auto-proteção, dentre outras.

Um senso comum na literatura é que, para ser autônomico, os sistemas devem ser compostos por uma hierarquia de sub-sistemas autonômicos, desde o nível mais alto (nível de utilizador) até os componentes mais básicos.

Segundo [20], alguns sistemas de informação, principalmente devido a características de seu domínio, podem ser desenvolvidos de uma forma que os tornem mais configuráveis e adaptáveis. Assim, esses sistemas podem se alterar de uma maneira mais eficiente para satisfazer requisitos do seu domínio de informação, que vierem a surgir (ou se modificar) após o período de desenvolvimento desses sistemas.

Um sistema distribuído pode evoluir para se adaptar às novas exigências da aplicação ou mudanças no ambiente (devido a falhas, adição ou remoção de recursos de computação, dentre outros.). Portanto, ser adaptável é uma exigência, em primeiro lugar, para lidar com a natureza dinâmica de tais sistemas. Assim, sistemas adaptativos tem sido o foco de muitas pesquisas em áreas como a confiabilidade, sistemas de tempo real, engenharia de software, dentre outros. Por exemplo, um sistema tolerante a falhas é adaptável por definição, porque ele vai trabalhar corretamente apesar da falha de alguns dos seus componentes, ou seja, eles

são adaptáveis em relação a falhas de componentes. A natureza de adaptabilidade de um sistema distribuído em relação à sua capacidade autonômica, os sistemas distribuídos autonômicos podem ser classificados de acordo com o grau de controle que pode exercer sobre os mecanismos adaptativos relacionados, se houver, como segue abaixo. [19]

1. Não adaptativo: nenhuma adaptação é fornecida;

2. Adaptativo *offline*: a adaptação é possível, mas apenas antes da execução - não há adaptação em tempo de execução;

3. Adaptativo *on-line*: a adaptação é realizada em tempo de execução através de objetivos pré-definidos. Políticas não podem ser controladas em tempo de execução;

4. Adaptativo autonômico: políticas ou objetivos de adaptação podem ser modificadas em tempo de execução.

A fim de implantar sistemas distribuídos (SD) autonômicos, blocos básicos de SD devem permitir que aplicativos ou serviços da camada superior controlem seu comportamento através de certas políticas ou objetivos. As políticas ou objetivos podem especificar requisitos de QoS e/ou restrições ambientais, e é importante definir objetivos autônomos ou derivá-los de SLAs (*Service Level Agreement*) da camada superior. Esta abordagem é muito utilizada para implementar detectores de falhas autonômicos e comunicação em grupo autônomo que são importantes para construção de blocos de SD. No caso detector de falhas, o sistema de camada superior (um usuário ou sistema) pode controlar a QoS de detecção de falha sob os recursos disponíveis, tais como o tempo de detecção e precisão de detecção. No caso de comunicação em grupo, o tempo de bloqueio e sobrecarga de protocolo pode ser controlado. Nos dois casos (detecção de falhas e comunicação em grupo), blocos de construção são classificados como adaptativo autonômicos e, em último caso, apenas adaptativo on-line, como o seu objetivo - otimizando o rendimento - não é modificado durante a execução. Adaptativo online é uma forma mais fraca de um sistema distribuído autônomo [19].

V. ISIS

Segundo [13], *Isis* é um sistema desenvolvido a partir de um estudo de tolerância a falhas em sistemas distribuídos. O sistema possui um conjunto de técnicas de construção de software para sistemas distribuídos que tenha boa performance, seja robusto, apesar de falhas, e explore paralelismo. A abordagem básica é a de proporcionar um mecanismo de conjunto de ferramentas para a programação distribuída, através do qual um sistema distribuído é construído pela interligação de programas não distribuído bastante convencionais, utilizando ferramentas desenhadas a partir do kit. Ferramentas são incluídas para o gerenciamento de dados replicados, sincronização de sistemas distribuídos, automatizar recuperação, e reconfigurar dinamicamente um sistema para acomodar mudanças de cargas de trabalho. ISIS tornou-se muito bem sucedido: centenas de empresas e universidades atualmente empregam o kit de ferramentas em ambientes que variam de pregões financeiros a sistemas de telecomunicações.

Para [22], o Isis tem sido eficaz na resolução de problemas de consistência em sistemas distribuídos, cooperação em algoritmos distribuídos e tolerância a falhas. Dois aspectos do ISIS são chave para sua abordagem global:

- Uma implementação de grupos de processos virtualmente síncronos. Este grupo é composto por um conjunto de processos que cooperam para executar um algoritmo distribuído, replicar dados, fornecer serviço tolerante a falhas ou explorar distribuição.

- Uma coleção de protocolos *multicast* confiável no qual os processos e os membros do grupo interagem entre si e com outros grupos. Confiabilidade no ISIS engloba atomicidade, garantias de entrega ordenada e uma primitiva de no qual as alterações de membros no grupo são sincronizadas.

Isis suporta um modelo de execução chamado sincronismo virtual, e isso impõe obrigações de protocolo que geralmente não foram considerados em estudos anteriores. Além disso, Isis fornece um bom desempenho para uma ampla gama de padrões de comunicação, incluindo a RPC, multi-RPC, *streaming*

em ponto-a-ponto e a nível de grupo, *multicasts one-shot* emitidos em rápida sucessão em diferentes grupos, e também fornece um bom desempenho para diferentes tamanho de mensagens [23].

Segundo [5], o projeto Isis desenvolveu um toolkit para programação distribuída, e uma coleção de aplicativos de alto nível baseado nessas ferramentas. O interesse em ISIS para interoperabilidade levou-nos a utilizar o sistema para uma ampla gama de hardware, e oferecer interfaces para uma variedade de linguagens, como Fortran e Lisp.

O Isis é utilizado para desenvolvimento de sistemas distribuídos confiáveis, através de grupos de processos. Se um programa particular é necessário para manter um sistema disponível, o sistema introduz um grupo de programas cada um dos quais replicando o original. Para atualizar o dado gerenciado pelo programa replicado, o sistema envia uma mensagem ao grupo de processo. Cada membro reage atualizando sua réplica particular. Como todos os programas vêem as mesmas atualizações na mesma ordem, eles irão permanecer em estados mutuamente consistentes [21].

Isis implementa um modelo de execução que permite ao usuário ser rigoroso sobre os possíveis comportamentos do sistema, e desenvolver algoritmos no qual os membros do grupo se comportem de forma consistente em relação ao outro. Por consistência nos referimos a um conjunto de propriedades que permitem que os membros do grupo cooperem, por exemplo, para simular um único processo, para replicação de dados, para sincronizar as ações, subdividir tarefas, dentre outros. O modelo desenvolvido para essa proposta é a sincronia virtual, e pode ser considerado como uma adaptação de serialização transacional (transações concorrentes não podem interverir umas com as outras) em um ambiente caracterizado por grupos de processos, computação cooperativa, e comunicação em grupo [23].

A. Tipos de grupo

O Isis suporta quatro tipos de grupo, sendo eles:

- Grupo igualitário: Neste tipo de grupo os processos cooperam entre si, a fim realizar uma tarefa. Podendo, por exemplo, replicar dados, subdividir tarefas, monitorar um ao outro, e, de alguma forma, realizar uma ação de forma distribuída e coordenada;

- Grupo cliente e grupo servidor: nesse caso um grupo de processos atua como servidores para um conjunto potencialmente grande de clientes. Clientes interagem com servidor através de requisições e respostas, podendo a comunicação ser via RPC com um servidor escolhido, ou *multicast* para o grupo de servidores;

- Grupo de difusão: é um tipo de cliente-servidor no qual os servidores enviam mensagens a um conjunto de servidores e clientes. Nesse caso, clientes são passivos, ou seja, apenas recebem mensagens. Qualquer aplicação é envia mensagem para um grande número de clientes, como um pregão eletrônico, utiliza difusão atômica;

- Grupo hierárquico: surgem a partir da necessidade de grupos servidores maiores em um sistema distribuído. Este tipo de grupo é composto por três estruturas de grupos. Um grupo raiz que recebe solicitação de conexão de um determinado grupo, e uma aplicação, que interage somente com este subgrupo.

B. Sincronia Virtual

Segundo [25], Isis foi o primeiro a propor o modelo de sincronia virtual. Esse modelo perfaz os aspectos que levaram o Isis a ter sucesso. Sua abordagem torna possível para um processo inferir o estado e as ações dos processos remotos usando informações de estado local e eventos que têm sido observados no local. O uso dessa propriedade, pode-se muitas vezes levar a soluções elegantes e eficientes para os problemas que seriam muito difíceis de resolver e implementar [5]. A sincronia virtual provê uma forma de comunicação assíncrona, nesse modelo a ordenação das mensagens são feitas apenas em casos de dependência causal entre as mensagens.

C. Arquitetura

Segundo [23], Isis implementa diversas camadas de protocolo, usando uma arquitetura empilhada como o TCP/IP, ou da arquitetura OSI. A camada mais alta é a VSYNC (sincronia virtual, ver Figura 10), responsável pelo suporte do modelo de sincronia virtual de *n* grupos. Segundo [23], as camadas do Isis são:

- Camada de sincronia virtual

A camada mais alta é a VSYNC (sincronia virtual, ver Figura 10), responsável pelo suporte do modelo de sincronia virtual de *n* grupos.

Para [23], são algumas características dessa camada:

- Através do *compressed vector timestamps* adiciona informações de ordenamento nas mensagens. O *timestamp*, ou carimbo de tempo, cresce proporcionalmente ao número de processos (os que enviam mensagem) no grupo. Mesmo quando vários grupos são usados, o Isis não adiciona mais de um *timestamp* em cada mensagem.
- A entrega de mensagens pode ser adiada por questões de ordenamento e atomicidade. Há uma série de razões para se atrasar uma mensagem, como:

  1. Uma mensagem pode chegar antes de alguma outra que deveria precedê-la;

  2. A transmissão de uma mensagem pode ser adiada até que alguma outra mensagem seja enviada;

  3. A entrega de um *multicast* totalmente ordenado pode ser adiada até que a ordem de entrega seja conhecida, etc.

- Quando ocorre mudança na visão do grupo, há garantia de que cada mensagem seja entregue, de forma atômica, para todos os membros do grupo.

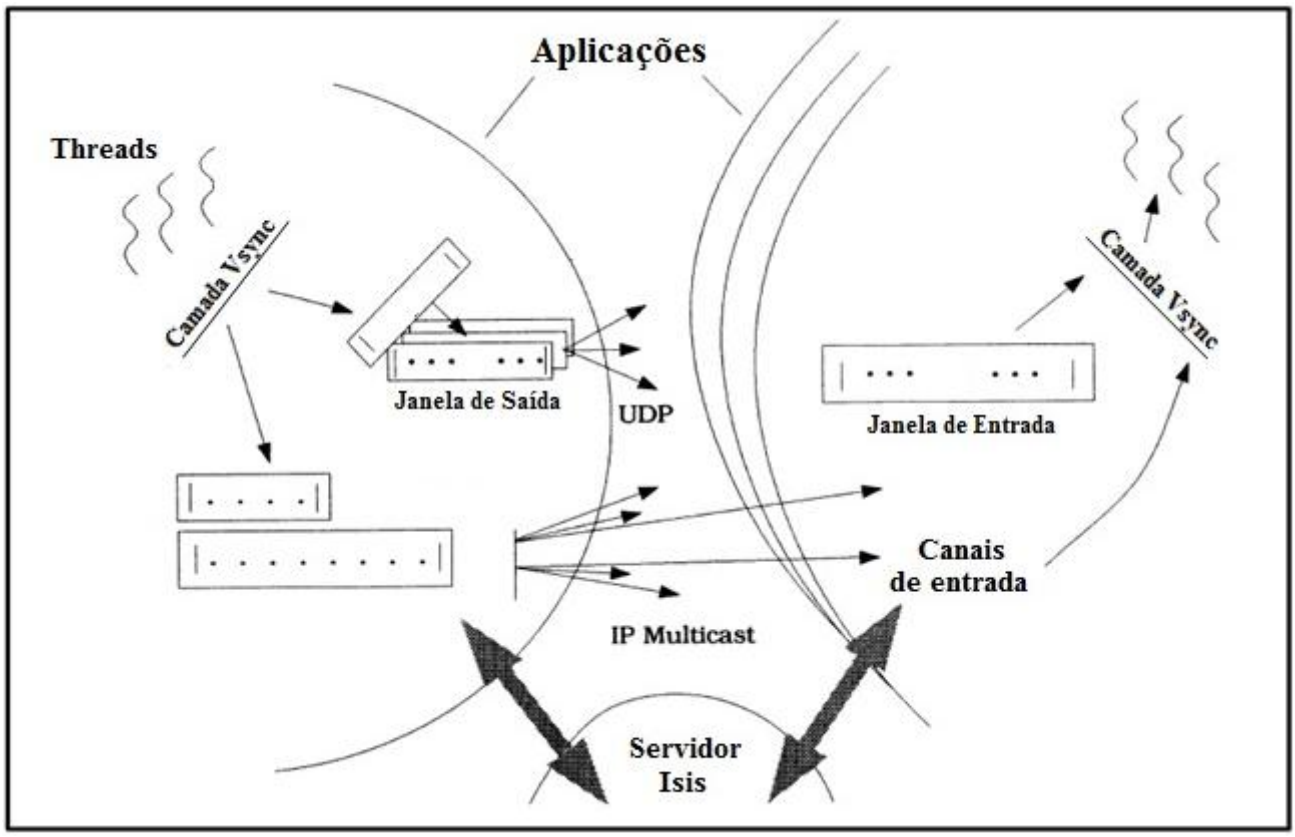

Figura 10- Arquitetura de comunicação do Isis. Adaptado de [23]

- Camada de transporte

  Abaixo da camada de sincronia virtual encontra-se a camada de transporte *multicast*. Esta camada é responsável pela entrega ordenada de mensagens, sem perda ou duplicação, a menos que o remetente falhe. Além de ser responsável por informar a camada VSYNC se a entrega foi realizada, ou não.

D. Primitivas

ISIS difere de outros sistemas baseados em comunicação em grupo, pois integra gerenciamento de mudanças no grupo, e por possuir duas primitivas de comunicação *multicast*, CBCAST e ABCAST [5]. *Multicast* é utilizado nas duas primitivas, porém fornecem padrões de comunicação diferentes.

Além do ABCAST e CBCAST, o Isis possui uma outra primitiva, chamado de GBCAST, sendo que os três possuem diferentes semânticas. ABCAST fornece comunicação síncrona e é utilizado na transmissão de mensagens a membros de um grupo. CBCAST fornece comunicação virtualmente síncrona e também é utilizado para o envio de mensagens. O GBCAST tem um pouco dos conceitos do ABCAST, exceto que este é utilizado para controlar a associação de processos a um grupo, ao invés de transmitir mensagens.

ABCAST

ABCAST é uma primitiva de comunicação do tipo síncrona e bloqueante. Esta primitiva faz uso do protocolo *two-phase commit*, ficando bloqueado até que os receptores confirmem o recebimento da mensagem e, com isso, todos os processos do grupo recebam a mensagem na mesma ordem – comunicação fracamente síncrona [24].

Segundo [24], o ABCAST funciona da seguinte forma:

- Um emissor, A, atribui um *timestamp* (um número seqüencial) a mensagem e a envia para todos os membros do grupo;

- Cada membro do grupo associa um *timestamp*, maior do que qualquer outro *timestamp* que foi enviado ou recebido, e manda de volta para A;

- Quando A recebe todas as mensagens de retorno, este escolhe o maior *timestamp* e manda um *commit* para todos os membros do grupo contendo o *timestamp* escolhido.

Com isso, há garantia de que a entrega das mensagens sejam realizadas na mesma ordem para todos os membros do grupo. Nesta primitiva, a performance do sistema é baseada na latência, pois, caso o receptor demore para responder ao processo A, o sistema perderá performance.

CBCAST

Embora o ISIS suporte uma ampla gama de protocolos *multicast*, a primitiva CBCAST é utilizada na maioria dos sistemas, na verdade, muitas ferramentas baseada no ISIS são basicamente invocações dessa primitiva [22].

A primitiva ABCAST apresentada na seção anterior sofreu modificações em seu funcionamento, a nova versão será apresentada nesta Seção. Ainda assim, sua nova versão é muito mais lenta do que CBCAST.

Segundo [24], pelo fato de o ABCAST ser complexo e caro, os projetistas do Isis criaram o CBCAST, esta primitiva do tipo assíncrona e não bloqueante, garante a entrega ordenada apenas para mensagens que são causalmente relacionados. Sendo utilizada na implementação de dependência causal e sincronia virtual. A primitiva funciona da seguinte forma:

- Assumindo um grupo com *n* membros, cada processo mantém um vetor com *n* posições, uma posição por membro do grupo.

- A i-ésima posição deste vetor armazena o número da última mensagem recebida na seqüência do processo *i*;

- Os vetores são gerenciados, em tempo de execução, pelo próprio Isis, e não pelos processos, e são inicializados com zero;

- Sempre que for enviar uma mensagem, o processo incrementa (+1) em sua posição no

vetor, e o vetor é enviado como parte da mensagem.

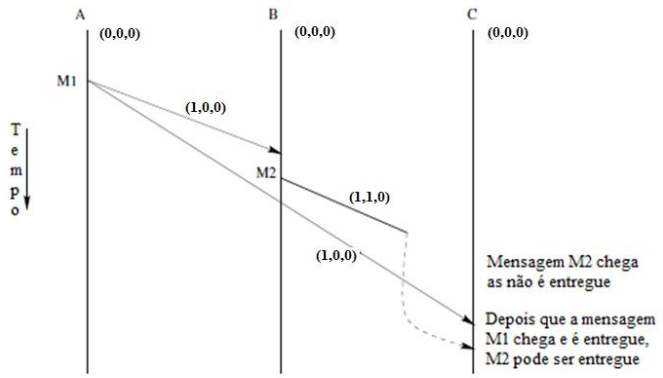

Figura 11- Entrega de mensagens utilizando o CBCAST. Adaptado de [24]

A Figura 11 exemplifica o funcionamento do CBCAST para um grupo com três processos [24]:

1. O processo A envia a mensagem *M* (contendo o vetor [1,0,0]) para B e C;

2. *M1* chega a B, é verificado que 0 foi a última mensagem enviada por A e que os outros elementos do vetor são iguais aos que B armazena. Logo, a mensagem é entregue a B.

3. Após *M1* ser recebida B, este envia a mensagem *M2* (contendo o vetor [1,1,0]) para C. Ocorre que a mensagem *M2* chega antes que *M1*. C verifica que B já recebeu uma mensagem de A antes de *M2* ter sido enviada. Então *M2* não é entregue a C até que a mensagem *M1* seja recebida.

Suponha que um conjunto de processos P se comunicam através de *broadcast* para um conjunto de todos os processos do sistema. Isto é, $\forall m$: $dests(m) = P$. Com CBCAST, cada processo p recebe mensagens enviadas a ele, atrasa a entrega, se necessário, e depois entrega em uma ordem consistente e com causalidade [22]:

$$m \rightarrow m' \text{-»} \forall p: deliver(m) \rightarrow deliver(m')$$

Segundo [22], o CBCAST garante entrega ordenada causal e livre de *deadlock*. O algoritmo geral para entregar uma mensagem é o seguinte:

1. Antes de enviar *m*, o processo p_i incrementa $VT(p_i)[i]$ e carimbo *m*;

2. Ao receber a mensagem *m* enviada por p_i e com carimbo $VT(m)$, processo $p_j \neq p_i$ atrasa a entrega até que:

$$\forall\, k: 1 \ldots n = \begin{cases} VT(m)[k] = VT(p_j)[k] + 1, & k = i \\ VT(m)[k] \leq VT(p_j)[k], & k \neq i \end{cases}$$

3. Quando a mensagem *m* é entregue, $VT(P_j)$ é atualizado a seguinte forma:

$$\forall\ k: 1 \ldots n: VT(p_j)[k] = max(VT(p_j)[k], VT(m)[k])$$

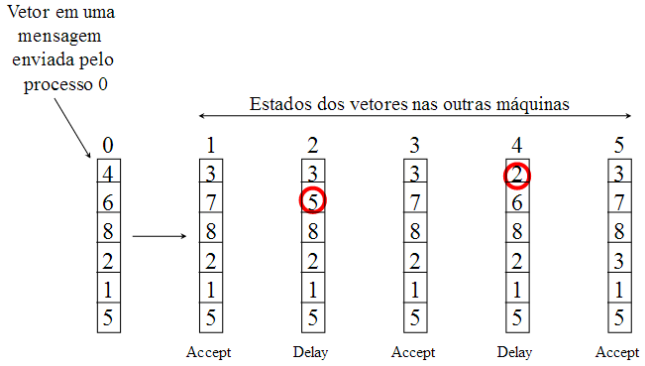

Figura 12- Vetores usando CBCAST

A Figura 12 exemplifica o funcionamento deste algoritmo com seis processos:

.

- ✓ O processo 0 envia uma mensagem *m* com o vetor [4, 6, 8, 2, 1, 5] para os outros membros do grupo;

- ✓ O processo 1 verifica que possui as mesmas mensagens que o processo 0, então a mensagem pode ser entregue para o processo 1;

- ✓ O processo 2 verifica que não recebeu a mensagem 6 enviada pelo processo 1, então a mensagem 4 do processo 0 não pode ser entregue, então esta aguarda;

- ✓ O processo 3 verifica que possui as mensagens que o processo 0 e ainda a mensagem 7 vinda do processo 1, e que não chegou ao processo 0, então a mensagem é aceita;
- ✓ O processo 4 não recebeu a mensagem 3 do processo 0 ainda, a mensagem 4 tem que aguardar;
- ✓ O processo 5 pode receber a mensagem 4.

Protocolo ABCAST Causal

A primitiva CBCAST pode ser facilmente estendida para trabalhar de forma totalmente ordenada. Isto foi proposto por Birman [22], se tornando a nova versão do ABCAST. Podemos notar que é incomum, para o ABCAST, garantir que essa ordenação total adapte-se com a causalidade. Por exemplo, um processo *p* transmite, de forma assíncrona, uma mensagem *m* usando ABCAST, após isto, envia uma mensagem *m'* usando CBCAST, e que algum receptor de *m'* envia *m''* usando ABCAST. Aqui nos temos $m \rightarrow m' \rightarrow m''$, mas *m* e *m''* são transmitidos por processos diferentes. A solução proposta por Birman sempre entrega *m* antes de *m''*. Esta propriedade é muito importante: sem ela poucos algoritmos poderiam usar ABCAST assíncrono de forma segura, e os atrasos introduzidos pelo bloqueio até que o protocolo realize entrega ordenada pode ser significativo [22].

VI. AVALIAÇÃO DE DESEMPENHO

Como vimos na Seção V, Isis é um protocolo, ou *toolkit*, desenvolvido na *Cornell University*, sendo baseado em duas primitivas de comunicação em grupo. Como resultado deste trabalho, o protocolo foi modificado a fim de se adaptar as necessidades da aplicação. Para isso, foi necessário obter resultados de execuções em diferentes cenários, analisá-los e realizar mudanças com base nestes dados. Após estas mudanças, foi imprescindível colher novos resultados para diferentes cenários.

Desempenho e confiabilidade são fundamentais em sistemas distribuídos, sendo um fator diferencial entre os protocolos de comunicação em grupo: no contexto de desempenho em comunicação em grupo, o foco deste trabalho é a vazão e latência; E no contexto confiabilidade o foco é a garantia de entrega causalmente ordenada, ou totalmente ordenada.

A. *Métricas*

Esta seção visa demonstrar quais foram os passos utilizados para desenvolver a nova versão do Isis (adaptável). Ao final, é feita uma discussão sobre os resultados, comparando, com isso, as duas versões: a estática e a adaptável.

Para analisar o desempenho das duas versões, foram utilizadas duas métricas:

- ✓ Vazão: definida por quantidade de mensagens em um período. Por exemplo, a quantidade de mensagens entregues em cinco minutos. Neste artigo, é verificada a quantidade total de mensagens entregues por todos os processos em um período de cinco minutos, ou seja, a soma da quantidade de mensagens entregues por cada processo;
- ✓ Latência: definida como o atraso desde o envio até a entrega. Por exemplo, caso determinado processo entregue *n* mensagens e o somatório da latência de cada mensagem seja dada por *totalLatAcum*, a latência média será dada por:

$$Latência\ média = \frac{totalLatAcum}{n}$$

B. *Ambiente de Execução*

O Isis adaptável foi avaliado em três diferentes cenários, sendo que em cada cenário foi utilizado um número *n* de processos, divididos em duas máquinas físicas (ver Figura 13 - *Arquitetura do Ambiente de Validação*). O número de processos utilizados foram:

- ✓ 10 processos (5 em cada máquina);
- ✓ 30 processos (15 em cada máquina);

- ✓ 50 processos (25 em cada máquina).

Como ilustrado na Figura 13, a arquitetura do ambiente de execução é composta por duas máquinas físicas, cada uma executando o mesmo número de processos. O serviço executa em apenas uma máquina, pois só deve existir um processo que controle o tempo de execução do protocolo, bem como a entrada de processos no grupo. As duas máquinas utilizadas na validação deste trabalho possuem as mesmas características:

- ✓ Máquina Dell com processador Intel® Core™ i5-3550 3.30GHz e 6MB de Cache L3;
- ✓ 8 GB de memória RAM (DDR3);
- ✓ 500 GB de espaço de disco rígido;
- ✓ Sistema Operacional Ubuntu, versão 13.10;
- ✓ *Java Virtual Machine 6 Update 23*;

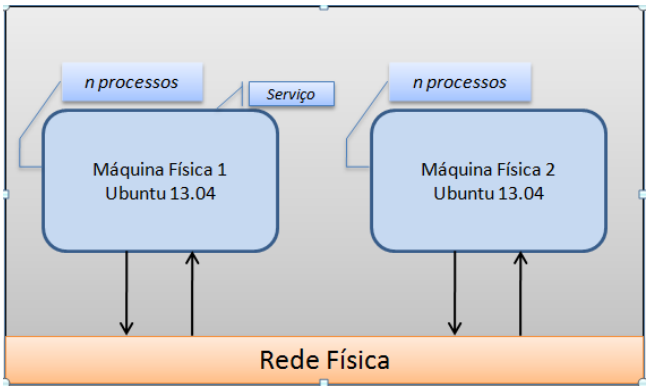

Figura 13 - Arquitetura do Ambiente de Validação

C. *Adaptabilidade no Isis*

O protocolo Isis pode ser executado com diferentes tamanhos de *buffer* e duas formas de entrega: ABCAST e CBCAST. Estes parâmetros determinam configurações distintas de execução que serão analisadas em diferentes cenários.

Como o objetivo de analisar o comportamento do Isis, foi realizado diversas execuções em diferentes cenários. Com os resultados obtidos foi possível verificar, por exemplo, quais cenários o Isis tem melhor desempenho. Cada cenário varia da seguinte forma:

- ✓ Número de processos: o número de processos no grupo varia entre dez, vinte, trinta, quarenta e cinqüenta processos;
- ✓ Tamanho do *buffer*: o tamanho do *buffer* varia entre cinco, dez e quinze;
- ✓ Primitiva: a primitiva varia entre o ABCAST e CBCAST.

Todos os possíveis cenários (combinações das variáveis acima) foram avaliados, obtendo, com isso, um total de trinta cenários diferentes. Estes cenários estão ilustrados nas tabelas Tabela *1*, Tabela *2* e Tabela *3*. Nestas tabelas, vazão é o número total de mensagens entregues em um período de tempo, enquanto latência é a média entre o tempo de envio e o tempo de entrega das mensagens.

Analisando os dados obtidos, verificou-se, obviamente, que a latência e vazão são influenciadas pela primitiva utilizada, ou seja, nestas tabelas fica claro que o CBCAST é muito mais rápido que ABCAST. Além disso, é possível verificar, baseado nos resultados, que a latência e vazão possuem relação com o tamanho do *buffer*. Com isso, quando o tamanho de *buffer* é pequeno (por exemplo, 5) a latência é menor e a vazão é maior, o contrário ocorre quando o valor do *buffer* possui um tamanho relativamente grande (por exemplo, 15).

	ABCAST (vazão)	**CBCAST (vazão)**	**ABCAST (latência)**	**CBCAST (latência)**
10 processos	4216	6443	3777	1622
20 processos	22746	25568	2727	811
30 processos	54315	57630	2476	544
40 processos	93024	101932	3130	415
50 processos	147220	160735	4087	249

Tabela 1 - Resultados da execução do Isis com tamanho do *buffet* igual a 5

	ABCAST (vazão)	CBCAST (vazão)	ABCAST (latência)	CBCAST (latência)
10 processos	4160	6470	4689	3062
20 processos	22112	25736	3607	1291
30 processos	55602	57725	3488	1104
40 processos	90658	102407	4865	575
50 processos	141619	161425	5607	644

Tabela 2 - Resultados da execução do Isis com tamanho do *buffet* igual a 10

	ABCAST (vazão)	CBCAST (vazão)	ABCAST (latência)	CBCAST (latência)
10 processos	3676	5953	5329	4662
20 processos	20346	23718	4407	1624
30 processos	51993	53204	3996	1840
40 processos	82122	94839	5025	882
50 processos	138155	149728	6007	1124

Tabela 3 - Resultados da execução do Isis com tamanho do *buffet* igual a 15

Nossa proposta de adaptabilidade visa variar, em tempo de execução, o tipo de primitiva e/ou tamanho do *buffer* (entre 1 e 15) a fim de suprir as necessidades da aplicação. No presente trabalho será considerada como necessidade da aplicação a latência máxima aceitável (*lma*). A variação da primitiva e/ou tamanho do *buffer* funciona da seguinte forma:

1. A latência máxima aceitável deve ser definida;
2. O protocolo é inicializado com a primitiva ABCAST e tamanho do *buffer* igual a 15;
3. A latência média (latência acumulada/total de mensagens entregues) é recalculada a cada 20 mensagens entregues;
4. Se a latência média for maior que *lma* e:
   - O tamanho do *buffer* for maior que um, então é decrementado uma unidade do valor do seu tamanho;
   - O tamanho do *buffer* é igual a um e o tipo de primitiva utilizada é ABCAST, o tamanho passa a ser 15 e a primitiva CBCAST.
5. Se a latência média for menor que 80% de *lma* e:
   - O tamanho do *buffer* for menor que 15, então o tamanho do buffer é incrementado em uma unidade;
   - O tamanho do *buffer* for maior, ou igual, a 15, e o tipo de primitiva utilizada seja CBCAST, tamanho passa a ser um e a primitiva ABCAST.

D. Implementação

A implementação realizada neste trabalho foi baseada na versão do Isis proposta por Birman [22]. A Figura 14 apresenta o diagrama de classes referente a implementação do Isis adaptável.

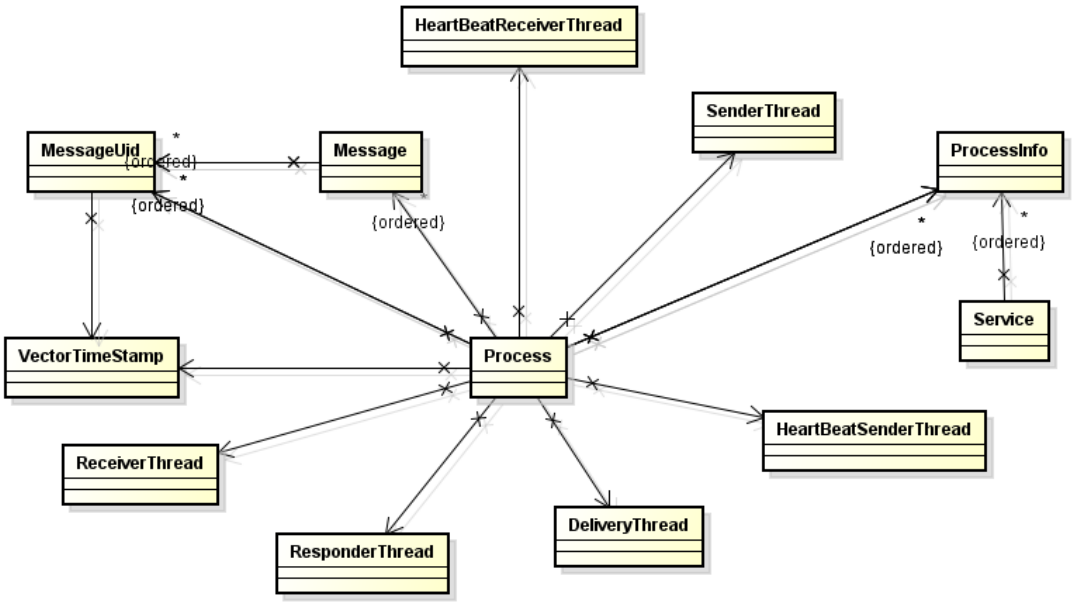

Figura 14 - Diagrama de Classes da Solução de Implementação do Isis

- *Service*

Responsável por controlar a entrada de processos no grupo e enviar uma mensagem, da visão inicial do grupo, a cada processo, ou seja, cada processo saberá quais processos fazem parte do grupo antes de iniciar a comunicação.

Além disso, a classe *Service* é responsável por controlar o tempo de comunicação entre os processos, que neste trabalho foi de cinco minutos, enviando, após esse tempo, uma mensagem "QUIT", para que os processos finalizem a execução.

- *ProcessInfo*

Esta classe define as informações (atributos) de um processo, como *host*, *pid* e portas.

- *SenderThread*

Classe associada a um processo, responsável pelo envio de mensagens aos outros processos que fazem parte do grupo.

- *ReceiverThread*

Classe associada a um processo, responsável por receber mensagens enviadas por processos que fazem parte do grupo, e inseri-la no *buffer* de mensagens.

- *ResponderThread*

Classe associada a um processo, responsável por receber mensagens do tipo "*REJOIN*", "*LEAVE*", ou "*QUIT"*, enviadas pela classe *Service.*

- *DeliveryThread*

Classe associada a um processo, responsável por entregar as mensagens que estão no *buffer* do processo. Nesta classe é definido alguns atributos importantes, como o tempo máximo de latência aceitável, total de mensagens consumidas e o instante de mudança entre o ABCAST e CBCAST.

- *VectorTimeStamp*

Classe que lida com o processamento de um *VectorTimeStamp*. Por exemplo, comparando dois vetores.

- *Message*

Classe que define as propriedades de uma mensagem, como, *messageType* (ABCAST, CBCAST ou SETS_ORDER). Além de armazenar informações necessárias para tomada de decisão do protocolo adaptado, como: *sendTime, deliveryTime, latency.*

- *MessageUid*

Classe que define atributos que tornam uma mensagem única. A classe *Message* possui um atributo (identificador) do tipo *MessageUid.*

- *HeartBeatSenderThread*

Classe associada a um processo, responsável por verificar se algum processo do grupo falhou (verificando se o *socket* está conectado ou ocorreu *timeout*). Além disso, é responsável por transmitir qualquer mudança (visão) no grupo.

- *HeartBeatReceiverThread*

Classe associada a um processo, responsável por receber e atualizar sua visão do grupo.

- *Process*

Definição do processo, é aqui que o processo é instanciado e suas *threads* são inicializadas. Quando o processo é inicializado, este envia uma mensagem contendo suas informações (id, *host* e porta) para o *service*. Com isso, o *service* adiciona este processo a visão do grupo.

E. Resultados

O Isis adaptável foi executado de acordo com o ambiente de execução descrito anteriormente, ou seja, com as mesmas condições que a versão estática do Isis.

Os gráficos a seguir apresentam o comportamento da execução do Isis adaptável no período de 300 segundos (5 minutos). A primitiva ABCAST foi definida com valor 1, enquanto CBCAST foi definida com valor 2.

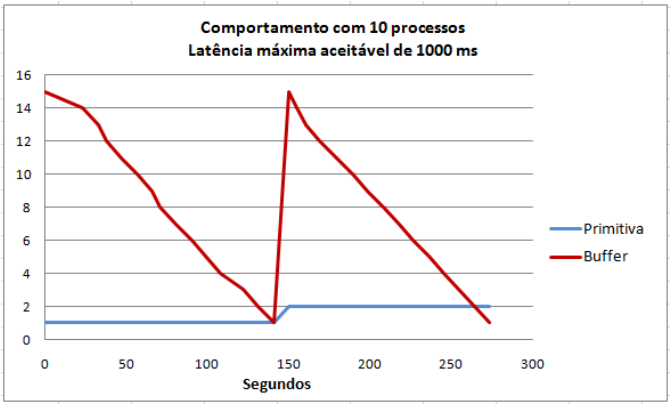

Gráfico 1 - Comportamento do Isis Adaptável em um cenário com 10 processos e Latência máxima aceitável de 1000 ms

O Gráfico 1 apresenta o comportamento do Isis executando com 10 processos e latência máxima aceitável (*lma*) de 1000 milisegundos. Verifica-se que a partir do tempo 273 segundos o tipo de primitiva (CBCAST) e o tamanho do *buffer* (1) se mantêm constantes.

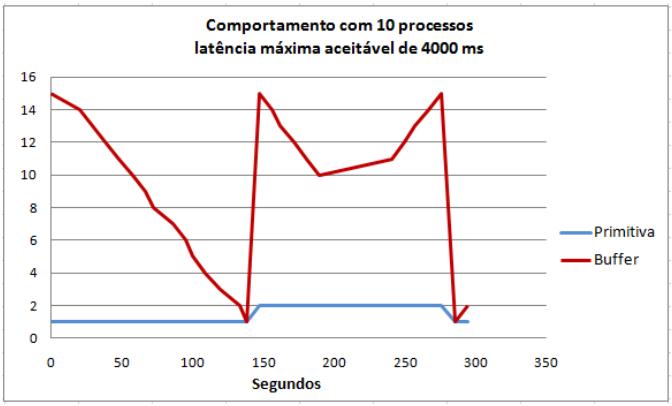

Gráfico 2 - Comportamento do Isis Adaptável em um cenário com 10 processos e Latência máxima aceitável de 4000 ms

O Gráfico 2 apresenta o comportamento do Isis executando com 10 processos e latência máxima aceitável de 4000 milisegundos. Com a limitação máxima de 4000 ms, muito maior do que 1000 ms (do Gráfico 1) verifica-se um comportamento menos constante em relação ao Gráfico 1, visto que o protocolo tende a manter a latência média entre 80% e 100% da *lma*.

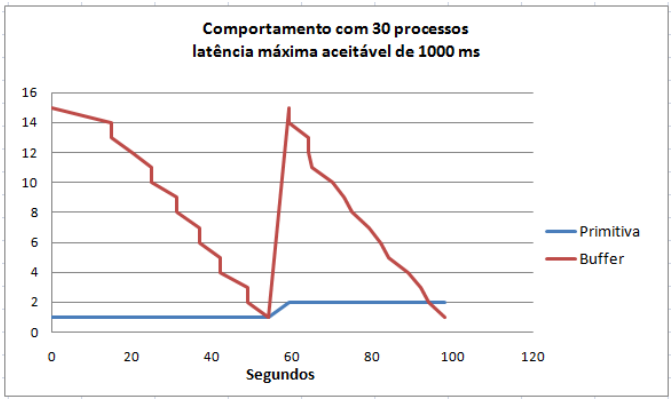

Gráfico 3 - Comportamento do Isis Adaptável em um cenário com 30 processos e Latência máxima aceitável de 1000 ms

O Gráfico 3 apresenta o comportamento do Isis em um cenário com 30 processos e latência máxima aceitável de 1000 milisegundos. Nota-se um

comportamento parecido com o do Gráfico 1. A principal diferença é que neste cenário, a primitiva (CBCAST) e tamanho do *buffer* (1) se mantêm constantes no tempo 98 s, e no primeiro cenário isso ocorre em 273 s. Isto se deve ao fato de que, com mais processos (neste caso 30), a quantidade de mensagens entregues por um período *p* de tempo é maior, logo, a latência média é recalculada em períodos menores de tempo.

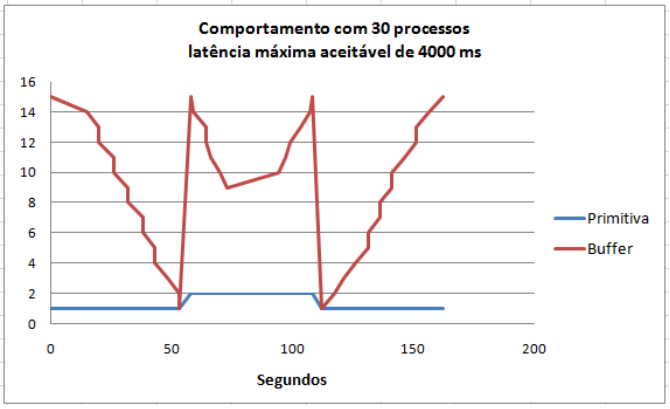

Gráfico 4 - Comportamento do Isis Adaptável com 30 processos e Latência máxima aceitável de 4000 ms

O Gráfico 4 apresenta o comportamento do Isis em um cenário com 30 processos e latência máxima aceitável de 4000 milisegundos. Seu comportamento é parecido com o cenário 2, e a explicação é similar a apresentada no cenário 3, ou seja, a quantidade de mensagens entregues em um determinado período de tempo é maior quando se há um número maior de processos no grupo, logo, a latência média é recalculada um número maior de vezes em relação ao cenário 2.

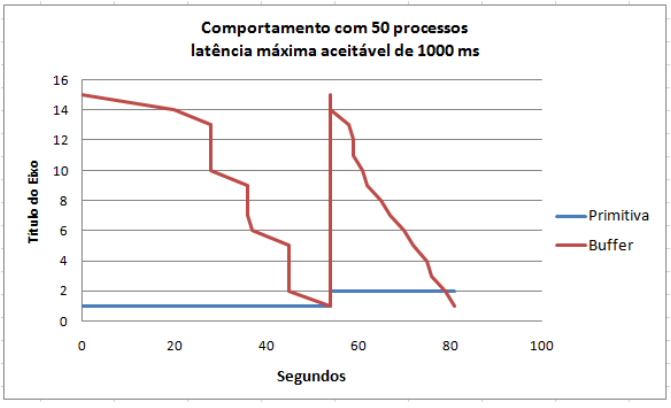

Gráfico 5 - Comportamento do Isis Adaptável em um cenário com 50 processos e Latência máxima aceitável de 1000 ms

O Gráfico 5 apresenta o comportamento do Isis em um cenário com 50 processos e latência máxima aceitável de 1000 milisegundos. Seu comportamento é parecido com os cenários 1 e 3. A diferença é que, no tempo 81s a primitiva e o tamanho do *buffer* se mantêm constantes até o fim da execução.

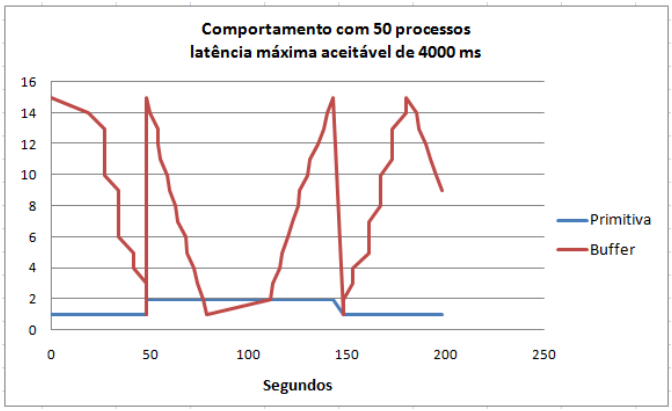

Gráfico 6 - Comportamento do Isis Adaptável em um cenário com 50 processos e Latência máxima aceitável de 4000 ms

O Gráfico 6 apresenta o comportamento do Isis em um cenário com 50 processos e latência máxima aceitável de 4000 milisegundos. Nota-se neste gráfico que há grande variação das grandezas. Isso se deve ao fato de o grupo possuir um número relativamente grande de processo e o valor *lma* ser grande (se comparado aos cenários que possuem valor 1000 ms).

A Tabela 4 - *Resultado da Execução do Isis Adaptável referente aos seis gráficos* apresenta os valores da vazão e latência média associada a cada cenário. Nota-se que no cenário 1 a latência média ficou muito acima do *lma*. Como este grupo possui um número relativamente pequeno de processos (dez), então a quantidade de mensagens entregues em um período *t* de tempo é muito menor em relação com grupos com trinta, ou cinqüenta, processos. Com isso, a latência média é recalculada em períodos maiores de tempo.

	Vazão	Latência
Gráfico 1	6327	2628
Gráfico 2	6491	2926
Gráfico 3	56494	835
Gráfico 4	55692	3733
Gráfico 5	156653	842
Gráfico 6	155854	3941

Tabela 4 - Resultado da Execução do Isis Adaptável referente aos seis gráficos

VII. CONSIDERAÇÕES FINAIS

O presente trabalho propôs implementar e avaliar o desempenho do protocolo Isis, em um estudo comparativo do efeito na latência da primitiva em uso e do tamanho do buffer. Estes parâmetros são utilizados para o desenvolvimento de uma versão adaptável do Isis que, de acordo com um limiar de latência máxima aceitável, configura o tamanho do buffer e a primitiva a ser utilizada.

O desenvolvimento de algoritmos com comportamento adaptável permite que a escolha adequada dos valores de parâmetros de configuração seja feita on-line e não em tempo de projeto, se adaptando a variações do ambiente.

Como trabalho futuro, o serviço de comunicação em grupo pode ser avaliado em uma plataforma de computação em nuvens, em que a escolha dos parâmetros pode definir custo na nuvem (gasto de CPU, memória, uso de rede etc.) e o objetivo proposto (ex: minimizar a latência) deve atender a um dado nível de requisitos de uso da nuvem.

BIBLIOGRÁFIA BÁSICA

[1] MACEDO, R. Computação Distribuída. http://www.lasid.ufba.br/publicacoes/artigos/ComputacaoDistribuida.pdf, acessado em abril de 2014.

[2] GOURLAR, A. (2002) Sistemas Distribuídos e Comunicação em Grupo

[3] SHIPER, A. Practical Impact of Group Communication Theory. Ecole Polytechnique Fédérale de Lausanne (EPFL).

[4] Shiper, A. (2006). Group Communication: From Practice to Theory, 117–136. Ecole Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland.

[5] BIRMAN, K. P.; COOPER, R. 1991. The ISIS Project: Real Experience with a Fault Tolerance Programming System. Operating System Review (April), 103-107. ACM/SIGOPS European Workshop on Fault-Tolerance. Techniques in Operating Systems, Bologna, Italy.

[6] COULOURIS, G. et al. Distributed Systems: Concepts and Design. Fifth Edition. Addison-Wesley, 2011.

[7] OLIVEIRA, R.; FRAGA, J.; MONTEZ, C. (2002). Programação em Sistemas Distribuídos. X Escola de Informática da SBC-Sul – ERI.

[8] Weber, T. S. Um roteiro para exploração dos conceitos básicos de tolerância a falhas.

[9] PAVAN, W. Tolerância a Falhas e Reflexão Computacional num Ambiente Distribuído. 2000. Dissertação – Programa de Pós-Graduação – Instituto de Informática – Universidade Federal do Rio Grande do Sul.

[10] PINHEIRO, G. Tuplebiz: Um espaço de tuplas distribuído e com suporta a transações resilientes a falhas bizantinas. 2012. Dissertação – Programa de Pós-Graduação – Instituto de Informática – Universidade Federal do Rio Grande do Sul.

[11] ZAMPIERI, A. Posicionamento de Réplicas em Sistemas Distribuídos. 2001. Dissertação – Programa de Pós-Graduação – Instituto de Informática – Universidade Federal do Rio Grande do Sul.

[12] Krzyzanowski, P. (2012). Remote Procedure Calls. http://www.cs.rutgers.edu/~pxk/417/notes/03-rpc.html, acessado em abril de 2014.

[13] The ISIS Project. http://www.cs.cornell.edu/info/projects/isis/, acessado em novembro de 2013.

[14] NETO, A. Protocolos para Difusão Confiável de Mensagens em Grupos de Comunicação. 1997. Dissertação – Instituto de Computação – Universidade Estadual de Campinas.

[15] Krzyzanowski, P. (2009). Group Communication. Rutgers University.

[16] Bessani, A. (2002) O Padrão Umiop eomo Base Para Comunicação de Grupo Confiável em Sistemas Distribuídos de Larga Escala. Dissertação-Universidade Federal de Santa Catarina.

[17] Lung, L. Computação Distribuída I. http://www.inf.ufsc.br/~frank/INE5418/Lau/Aula1e2-5418.pdf, acessado em maio de 2014.

[18] Gorender, S., Macêdo, R. J. A., and Raynal, M. (2007). An adaptive programming model for fault-tolerant distributed computing. IEEE Trans. on Dependable and Secure Computing.

[19] Macêdo, R. J. A. A Vision on Autonomic Distributed Systems.

[20] ALMEIDA, W.; FERREIRA, M. 2005. Persistir os Metadados dos Modelos de Objetos de Sistemas Distribuídos e Adaptáveis

[21] BIRMAN, K. P.; RENESSE, R. 1996. Software for Reliable Networks. Scientific American, may

[22] Birman, K., Shiper, A., Stephenson, P. (1991). Lightweight Causal and Atomic Group Multicast, 272-314.

[23] BIRMAN, K. P.; CLARK, T. 1994. Performance of the ISIS Distributed Computing Toolkit. Technical Report TR-94-1432, Dept. of Computer Science, Cornell University.

[24] Tanenbaum, A. S. Distributed Operating Systems, Prentice-Hall (1995). http://www.e-reading.ws/book.php?book=143358, maio de 2014.

[25] BIRMAN, K. P. 1993. The Process Group Approach to Reliable Distributed Computing. Communications of the ACM, december, v. 36, n. 12.

Adnilson Costa Garrido Junior nasceu em Salvador, Bahia e graduou-se no curso de Ciência da Computação pela Universidade Federal da Bahia (UFBA) em 2011. Atua na área de desenvolvimento de software na Cia de Processamento de Dados do Estado da Bahia – PRODEB desde 2013 e atualmente cursa a especialização em Computação Distribuída e Ubíqua no Instituto Federal de Educação, Ciência e Tecnologia da Bahia (IFBA).