Consenso com Participantes Desconhecidos em Memória Compartilhada

Cátia Khouri[1], Fabíola Greve[2], Sébastien Tixeuil[3]

[1]Departamento de Ciências Exatas – Universidade Estadual do Sudoeste da Bahia
Vitória da Conquista, BA – Brasil

[2]Departamento de Ciência da Computação – Universidade Federal da Bahia (UFBA)
Salvador, BA – Brasil

[3] UPMC Sorbonne Universités, Institut Universitaire de France
Paris – França

catia091@dcc.ufba.br, fabiola@dcc.ufba.br, Sebastien.Tixeuil@lip6.fr

Resumo. *O consenso é um problema fundamental para o desenvolvimento de soluções confiáveis em sistemas distribuídos dinâmicos, tais como sistemas P2P, grades oportunistas, redes móveis ad-hoc, etc. Diferentemente dos sistemas clássicos, onde o conjunto de participantes e suas identidades são conhecidos, nos sistemas dinâmicos não se sabe a priori quais são os pares com quem se pode colaborar, nem quantos deles estão disponíveis. Recentemente, condições necessárias e suficientes foram identificadas para a resolução do consenso nesse contexto, mas tomando-se por base o modelo de comunicação por passagem de mensagens. Nesse artigo, estendemos tais resultados e apresentamos protocolos para a resolução do consenso com participantes desconhecidos num modelo de memória compartilhada.*

Abstract. *Consensus is a fundamental problem for the development of reliable distributed systems deployed over dynamic networks, such as P2P, desktop grids, mobile ad-hoc networks, etc. Unlike classical networks, where the set of participants and their identities are known, in dynamic networks, the membership of the system and its cardinality are not known a priori. Recently, necessary and sufficient conditions have been identified to solve consensus in these environments, but for the message passing communication model. In this paper, we extend these recent results and present protocols able to solve consensus with unknown participants in the shared memory model.*

1. Introdução

Sistemas distribuídos modernos desenvolvidos sobre redes de sensores, redes móveis *ad hoc,* grades oportunistas, ou redes P2P não-estruturadas possibilitam a seus participantes acessarem serviços e informações independentemente de sua localização ou mobilidade. Isso é possível através de um desenho de sistema altamente dinâmico que dispensa a existência de uma autoridade administrativa centralizada ou uma infraestrutura estática. Questões relacionadas ao desenho de soluções confiáveis que lidam com a natureza de alto dinamismo e auto-organização de tais redes são um campo muito ativo de pesquisa.

Por outro lado, problemas de acordo são fundamentais em sistemas distribuídos confiáveis dos quais o mais importante é o do *consenso.* Neste, cada processo propõe

um valor e todos os processos corretos devem atingir uma decisão comum dentre os valores propostos. É bem estabelecido, entretanto, que não existem algoritmos determinísticos para resolver o problema do consenso em sistemas sujeitos a falhas por colapso (*crash*) mesmo que o número de processos faltosos seja limitado a um (Fischer et al 1985). Para lidar com essa limitação, uma abordagem é a utilização de detectores de falhas ou de líderes, os quais provêem os sistemas com o sincronismo extra necessário para contornar a impossibilidade do consenso em redes tradicionais (Chandra e Toueg 1996). Esses detectores podem cometer um número arbitrário de falhas, mas não comprometem as propriedades de corretude dos protocolos de consenso que os usa.

Num contexto de sistemas distribuídos clássicos, vários algoritmos de consenso foram propostos, inclusive tomando-se por base a abstração de detectores de falhas ou de líderes. Mas apenas recentemente alguns trabalhos foram propostos para resolução do consenso em sistemas distribuídos dinâmicos, e.g., Cavin et al (2004, 2005), Greve e Tixeuil (2007, 2010), Alchieri et al (2008), Delporte-Gallet e Faulconnier (2009).

Os trabalhos de Cavin et al (2004, 2005) e Greve e Tixeuil (2007, 2010) apresentam condições e protocolos para resolução do consenso em redes dinâmicas com participantes desconhecidos – FT-CUP (*consenso tolerante a falhas com participantes desconhecidos*). Ele mantém as mesmas propriedades do problema original, mas assume que os nós não têm consciência sobre o conjunto de processos no sistema (Π) nem sua cardinalidade (n). Entretanto, se alguma cooperação é esperada, os processos precisam obter algum conhecimento parcial sobre os demais. Para isso, esses trabalhos sugerem a abstração do *detector de participantes*, uma espécie de oráculo que fornece dicas sobre os processos participantes na computação. Baseado no grafo de conhecimento inicial formado pelos detectores de participantes define-se as condições de conectividade suficientes e necessárias desse grafo para resolver FT-CUP em ambientes assíncronos.

Cavin et al (2005) identificaram um detector de falhas perfeito (P) como um requisito de sincronismo necessário para resolver FT-CUP com os requisitos de conectividade mínimos sobre o grafo de conhecimento inicial. Mas P só pode ser implementado em um sistema síncrono. Assim, resolver FT-CUP em um cenário com conectividade de conhecimento mais fraca demanda condições de sincronismo mais fortes. Contudo, sincronismo forte compete com o alto dinamismo, descentralização total e auto-organização das redes de sensores e *ad-hoc*. Além disso, mesmo com um detector de falhas perfeito, quando se considera uma conectividade de conhecimento mínimo a versão uniforme do FT-CUP não pode ser resolvida em redes desconhecidas.

Greve e Tixeuil (2007) apresentam uma proposta alternativa para solucionar o FT-CUP. Eles consideram os requisitos minimais de sincronia, já identificados para a resolução do consenso no modelo clássico, e buscam os requisitos mínimos para a conectividade do grafo de conhecimento que possibilitam resolver o problema. O requisito mínimo de sincronia está representado pelo detector de falhas da classe $\Diamond S$ e detector de líder Ω. Por serem não-confiáveis, tais detectores são mais adequados para a modelagem de soluções em redes dinâmicas.

Por sua vez, grande parte dos algoritmos de consenso propostos é baseada no paradigma de troca de mensagens para comunicação entre processos. Entretanto, dada a importância e analogias entre sistemas de passagens de mensagens e de memória compartilhada, inclusive com relação aos requisitos mínimos para realização de consenso em ambientes assíncronos, alguns estudos foram dedicados a investigar o

problema do consenso em sistemas de memória compartilhada (Lo e Hadzilacos 1994, Guerraoui, R. e Raynal, M. (2007), Delporte-Gallet e Fauconnier 2009).

Informalmente, um sistema com memória compartilhada é um conjunto de processos executando que se comunicam através de um conjunto de células de memória compartilhadas sobre as quais há operações que podem ser executadas por um ou mais processos. Para cada célula existe um conjunto de valores possíveis de serem armazenados. Esse é o caso, por exemplo, das máquinas *multicore* atuais onde os processos compartilham uma única memória física. Um modelo semelhante também é encontrado em sistemas distribuídos, onde parte da memória de cada processador (e.g. registradores) é compartilhada de modo que um ou mais processos podem acessá-los. Nos sistemas distribuídos em que um nó pode ser um sistema *multicore* é possível vislumbrar o surgimento de modelos híbridos especialmente em redes auto-organizáveis em que os participantes não têm uma visão completa do sistema.

Este artigo estende o trabalho de Greve e Tixeuil (2007) e apresenta um novo conjunto de protocolos para a resolução do consenso uniforme em um sistema assíncrono, sujeito a falhas, com participantes desconhecidos, em que a comunicação entre os processos se dá através de memória compartilhada. Até onde sabemos, essa é a primeira proposta para resolução do FT-CUP em memória compartilhada. Além de fazer uma releitura do modelo de registradores compartilhados para um sistema assíncrono com grafos de conhecimento por conectividade, são propostos protocolos diferenciados que otimizam (em termos de complexidade e simplicidade) os anteriormente sugeridos para o modelo de troca de mensagens. Mais especificamente, esse artigo tem como contribuições: (1) o estudo do problema do FT-CUP para o modelo de memória compartilhada; (2) proposta de algoritmos originais apropriados para tal modelo e (3) que são mais eficientes (em latência e complexidade tempo/espaço) em comparação com os propostos anteriormente por Greve e Tixeuil (2007); (4) nova definição para o detector k-OSR com apresentação de propriedades mais fracas para a resolução FT-CUP.

De fato, arcabouços de memória compartilhada distribuída são bastante requisitados na implementação de serviços de armazenamento de dados distribuídos com requisitos de consistência e disponibilidade e que atendem a um grande número de clientes, tais como *data centers*, armazenamento em nuvem, etc. A disponibilidade é obtida através da replicação dos dados nos vários nós servidores, já a consistência, através da taxa de replicação (geralmente na maioria dos nós servidores) e do controle de operações de escrita e leitura da memória distribuída, bem como da reconfiguração do conjunto dos servidores. Num contexto de redes dinâmicas, devido a entradas, saídas e falhas dos nós, a reconfiguração (ou mudança de visões do conjunto de servidores) é de fundamental importância. A grande totalidade dos protocolos de reconfiguração usa o consenso como módulo de base para obter o acordo na mudança das visões (Aguilera et al. (2009). Assim, acreditamos que o protocolo FT-CUP aqui proposto, por considerar um conjunto de participantes desconhecidos e, sobretudo, ser voltado para memória compartilhada, será bastante útil no suporte ao desenvolvimento de tais soluções.

1.1. Trabalhos Relacionados

Cavin et al (2004, 2005) e Greve e Tixeuil (2007, 2010) são os únicos trabalhos de relevância a propor condições sobre a conectividade de conhecimento para resolver o consenso em uma rede com participantes desconhecidos sujeita a falhas (*crash*) para o modelo de troca de mensagens. Alchieri et al (2008) estudaram o problema do consenso

Bizantino tolerante a falhas – BFT-CUP, à luz da abstração do detector de participantes. Seguindo um caminho de protocolos e reduções similar a Greve e Tixeuil (2007) eles dão as condições necessárias e suficientes para resolver BFT-CUP em redes desconhecidas com o requisito adicional de que os processos comportam-se maliciosamente.

Delporte-Gallet e Faulconnier (2009) apresentam algoritmos para o consenso em um modelo híbrido com participantes desconhecidos e anônimos, considerando-se interações por memória compartilhada. Seu foco são sistemas anônimos onde, diferentemente do nosso modelo, processos não possuem identidade. Eles propõem soluções para sistemas eventualmente síncronos (em algum instante todos os processos se comunicam "em tempo") e para sistemas em que a partir de algum instante os mesmos processos se comunicam em tempo uns com os outros, mas não considera a abstração detector de participantes nem condições do grafo de conectividade.

Até onde sabemos, não foram propostas soluções para o consenso FT-CUP num sistema assíncrono voltadas para o modelo de memória compartilhada. Nesse sentido, nosso trabalho é pioneiro.

O resto do artigo está estruturado da seguinte maneira: A Seção 2 traz o modelo do sistema e alguns conceitos. A seção 3 apresenta a abstração detector de participante. Os protocolos encontram-se na Seção 4, a Seção 5 apresenta as conclusões e a Seção 6 as referências bibliográficas.

2. Preliminares

2.1. Modelo do Sistema

Considera-se um sistema distribuído que consiste de um conjunto finito de n >1 processos $\Pi = \{p_1, ..., p_n\}$, onde cada p_i pode estar consciente de apenas um subconjunto dos processos $\Pi_i \subseteq \Pi$. Π_i é um conjunto arbitrário definido de acordo com o detector de participantes (Seção 3.1). Um processo pode falhar por parada (*crashing*), i.e., parando prematura ou deliberadamente. Após parar, um processo não se recupera. Um processo que não falha é dito *correto* e um processo que falha é dito *faltoso*. O número máximo de processos que podem falhar, *f*, é conhecido por todos os processos. Processos se comunicam através da leitura e escrita em registradores regulares 1-escritor/n-leitores (1W/nR), os quais se comportam corretamente (problemas relacionados a registradores tolerantes a falha são discutidos em (Afek et al 1992). No modelo de participantes desconhecidos, o registrador R_i do processo p_i só pode ser escrito pelo próprio p_i e um processo p_i só pode ler o registrador R_j do processo p_j se $p_j \in \Pi_i$. Um registrador compartilhado modela uma forma de comunicação persistente onde o emissor é o escritor, o receptor é o leitor, e o estado do meio de comunicação é o valor do registrador. Um registrador 1W/nR regular, é um registrador no qual uma leitura não concorrente com uma escrita sempre pega o valor correto e quando uma leitura sobrepõe uma escrita pode obter o valor novo ou o antigo. (Lamport, 1986). Não fazemos suposições sobre o tempo que dura cada operação de leitura ou escrita, i.e., o sistema é assíncrono.

2.2. Consenso

O problema do consenso é o problema de acordo mais importante em computação distribuída. Neste, cada processo p_i propõe um valor v_i e todos os processos têm que decidir por um mesmo valor v. Formalmente, o problema é definido pelas seguintes

propriedades de segurança: (*safety*): (1) validade – se um processo decide por um valor *v*, então *v* foi proposto por algum processo; (2) acordo uniforme – se um processo decide por um valor então todos os processos decidem pelo mesmo valor; e uma propriedade de vivacidade (*liveness*) (3) terminação – todos os processos corretos acabam por decidir por algum valor. (Chandra e Toueg 1996).

Numa rede desconhecida sujeita a falhas, consideram-se três variantes do problema: CUP (Consenso com Participantes Desconhecidos) – consenso em uma rede desconhecida livre de falhas; FT-CUP (CUP Tolerante a Falhas) – consenso numa rede desconhecida onde até *f* processos podem falhar; e FT-CUP Uniforme (CUP Uniforme Tolerante a Falhas) – consenso uniforme em rede desconhecida com até *f* falhas.

2.3. A abstração Ômega (Ω)

Um detector de falhas da classe ◊*S* pode ser visto como um oráculo que fornece dicas (não confiáveis) sobre processos parados. O oráculo líder Ω satisfaz à propriedade de liderança eventual: existe um instante após o qual todo processo correto sempre confia no mesmo processo correto no sistema (i.e., o mesmo líder). ◊*S* e Ω possuem o mesmo poder computacional e são as classes mais fracas de detectores que permitem resolver o consenso em redes conhecidas assíncronas com passagens de mensagens (Chandra et al 1996) ou com memória compartilhada (Delporte-Gallet et al 2004).

2.4. Notações de Grafos

Um grafo direcionado $G_{di}(V, E)$ é *k*-fortemente conectado se para qualquer par de nós (v_i, v_j), v_i pode atingir v_j por *k* caminhos disjuntos nos nós. G_{di} é fortemente conectado se k=1. No grafo acíclico direcionado obtido da redução de G_{di} a seus componentes fortemente conectados, um poço é um componente do qual não saem arestas. Pelo Teorema de Menger [Yellen and Gross 1998], sabe-se que o número mínimo de nós cuja remoção de $G_{di}(V, E)$ desconecta os nós v_i e v_j é igual ao número de caminhos disjuntos de v_i a v_j.

Observação 1. Se um grafo é *k*-fortemente conectado, a remoção de (*k*-1) nós deixa pelo menos um caminho entre qualquer par de nós (v_i, v_j).

3. Condições de Conectividade de Conhecimento para Consenso em Sistemas Sujeitos a Falhas

3.1. Detectores de Participante: Uma Abstração de Conectividade de Conhecimento

A maior parte dos trabalhos na literatura sobre consenso considera que Π é conhecido de todo processo no sistema. Em redes de sensores sem fio, essa suposição é claramente não realística uma vez que processos podem ser mantidos por diferentes autoridades administrativas, ter vários instantes de acordar, inicializações, taxas de falha, etc. É claro que algum conhecimento sobre os outros nós é necessário para executar qualquer algoritmo distribuído não trivial. Por exemplo, o uso de mensagens “Hello” para a vizinhança poderia ser uma forma possível de cada processo obter conhecimento sobre outros processos. A noção de detectores de participantes (PD) foi proposta por Cavin et al (2004) para prover os processos com este conhecimento inicial sobre os participantes do sistema. Eles podem ser vistos como oráculos distribuídos que fornecem informações sobre quais processos participam do sistema. Denotamos por *i*.PD o detector de

participantes do processo p_i. Quando requisitado por p_i, i.PD retorna um subconjunto de processos em Π. Esta informação pode evoluir entre requisições.

Definição 1: [*Detector de Participante*] O detector de participante i.PD do processo p_i retorna um conjunto $\Pi_i \subseteq \Pi$. Seja i.PD a requisição de p_i no tempo $t \in \mathcal{T}$. Esta requisição deve satisfazer as duas seguintes propriedades: (a) Acurácia da Informação. O detector de participantes não comete erros: $\forall p_i \in \Pi$, $\forall t \in \mathcal{T}$: i.PD(t) $\subseteq \Pi$. (b) Inclusão da Informação. A informação retornada pelo detector de participante é não decrescente com o tempo: $p_i \in \Pi$, $\forall t, t' \in \mathcal{T}, t' \geq t$: i.PD(t) $\subseteq i$.PD(t').

Para simplificar a notação, dizemos que $p_j \in i$.PD quando $p_j \in i$.PD(t) para algum t. Note que, diferentemente dos detectores de falhas, a informação fornecida pelos detectores de participantes não lidam com falhas. Assim, o subconjunto de processos retornado por i.PD pode conter processos corretos ou faltosos.

3.1.1. O Grafo da Conectividade do Conhecimento

O relacionamento de conhecimento fornecido pelo PD pode ser modelado como um grafo da conectividade do conhecimento G_{di} no sistema. G_{di} é direcionado uma vez que o conhecimento que é dado pelo PD não é necessariamente bidirecional (i.e., pode ocorrer $p_j \in i$.PD e $p_i \notin j$.PD). Além disso, uma vez que o conhecimento evolui com o tempo, G_{di} evolui com o tempo também. Seja $G_{di}(t)=(V, E_{di}(t))$ o grafo direcionado obtido da saída do PD no tempo $t \in \mathcal{T}$. Então $V=\Pi$ e $(p_i, p_j) \in E_{di}(t)$ sss $p_j \in i$.PD(t), i.e. p_i conhece p_j no tempo t. A propriedade da Inclusão da Informação é válida para G_{di} de modo que $\forall t_i \geq t_{i-1}$: $G_{di}(t_{i-1}) \subseteq G_{di}(t_i)$ significando que novos relacionamentos de conhecimento entre processos podem ser identificados no decorrer do tempo, mas não descartados. Similarmente, $G(t) = (V, E(t))$ é o grafo não direcionado obtido do PD em t, $V = \Pi$ e $(p_i, p_j) \in E(t)$ sss $p_j \in i$.PD ou $p_i \in j$.PD, assim, $\forall\ t_i \geq t_{i-1}$: $G(t_{i-1}) \subseteq G(t_i)$. Na abordagem seguida nesse artigo, não temos que precisar o instante em que G_{di} é composto. Assim, nós consideramos as seguintes definições:

Definição 2: [$G = (V, E)$: *Grafo da Conectividade do Conhecimento não Direcionado*] $G = G(t_i) \supseteq G(t_{i-1}) \supseteq ... \supseteq G(t_0)$ é a sequência de subgrafos não-direcionados geradores de G, tal que $V = \Pi$, $E(t_i) \supseteq E(t_{i-1})$, $t_i \geq t_0$.

Definição 3: [$G_{di} = (V, E_{di})$: *Grafo da Conectividade do Conhecimento*] $G_{di} = G_{di}(t_i) \supseteq G_{di}(t_{i-1}) \supseteq ... \supseteq G_{di}(t_0)$ é a sequência de subgrafos geradores de G_{di}, tal que $V = \Pi$, $E_{di}(t_i) \supseteq E_{di}(t_{i-1})$, $t_i \geq t_0$.

3.1.2. Classes de Detectores de Participantes

Com base nas propriedades dos grafos da conectividade do conhecimento G_{di} (G) gerados, diversas classes de PD podem ser obtidas. Algumas classes foram propostas em Greve e Tixeuil 2007. A classe que reúne as condições minimais para resolver FT-CUP num ambiente assíncrono sujeito a falhas, é a *k-OSR*, que será utilizada neste artigo.

Definição 3: ***k-Strong Conectivity* PD** (*k-SCO*). Esta classe de PD contém todos os G_{di} tais que G_{di} é k-fortemente conectado:

Definição 4: ***k-One Sink Redutibility* PD** (*k-OSR*). Esta classe de PD contém todos os G_{di} que satisfazem às seguintes condições:

1. o grafo da conectividade do conhecimento não direcionado G é *conectado*;
2. o grafo acíclico direcionado obtido pela redução de G_{di} a seus componentes k-forte-

mente conectados tem exatamente um poço;

3. G_{sink} é k-fortemente conectado.
4. $\forall\, p_i, p_j \in G_{di}$, tais que $p_i \notin G_{sink}$, $p_j \in G_{sink}$, existem k caminhos disjuntos nos nós de p_i a p_j.

A Figura 1 ilustra um grafo G_{di} induzido por um PD k-*OSR*, para $k = 2$. Note que há somente um componente poço e que todo componente G_i é 2-fortemente conectado.

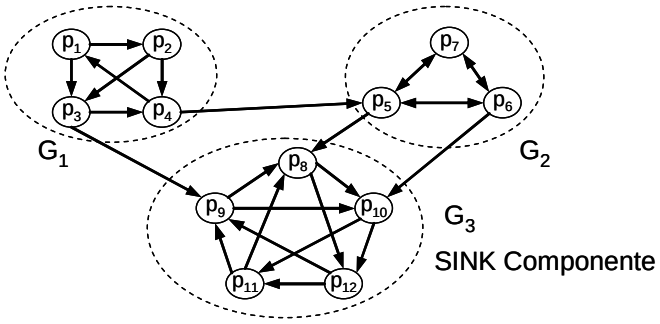

Figura 1. Grafo de conectividade do conhecimento induzido por um PD 2-*OSR*.

4. Protocolos para Resolver FT-CUP

A seguir, apresentamos os algoritmos para resolução do consenso com participantes desconhecidos no modelo de memória compartilhada. Para tanto, os processos do sistema irão se comunicar através de um registrador R do tipo 1W/nR. Nos algoritmos apresentados os processos se comunicam através do registrador compartilhado R. Denotamos por R_i ao conteúdo de R na posição i, referente aos valores dos campos armazenados por p_i. Cada processo p_i poderá ter acesso para "escrita" aos campos em R_i e poderá ter acesso para "leitura" aos campos em R_j para cada processo p_j de que p_i tem conhecimento. Cada registrador é composto dos campos: (i) *known*: lista de identidades de processos – representa o conjunto de processos conhecidos por p_i; (ii) *flag:* booleano que indica se p_i já concluiu o conhecimento do sistema; (iii) *decision*: armazena o valor decidido por p_i. As operações que os processos realizam sobre os registradores são as seguintes:

(i) **read** (R, v): operação de leitura sobre o registrador R retornando o valor v. Quando c denota um conjunto, a operação **read**(R, c) denota a leitura de cada elemento do conjunto em R e seu respectivo armazenamento em c. Assim, se p_j executa a operação **read**(R_i. *known*, c), o do conjunto de processos conhecidos por p_i será armazenado em na variável privada c, isto é, $c \leftarrow i.known$.

(ii) **write**(R, v): operação de escrita de v no registrador R. Da mesma forma que para o **read**(R, c), a operação **write**(R_i.*known*, c) denota $i.knownv \leftarrow c$.

(iii) **insert**(R, c) denota a escrita em R da união dos conjuntos c e R, isto é, **insert**(R_i.*known*, c) insere os elementos de $R_i.known \cap c$ em $R_i.known$.

4.1. Ampliando o conhecimento sobre os participantes do sistema

A informação fornecida pelo detector de participantes pode não ser suficiente para prover os processos com o conhecimento necessário para realizar o consenso. Por exemplo, se essa informação corresponde a um grafo completo de conhecimento, ele é suficiente, se é um grafo *k-OSR*, não. Desta maneira, propomos o protocolo COLLECT, um algoritmo completamente assíncrono que quando executado por todo processo p_i amplia seu conhecimento sobre os participantes do sistema através de uma prospecção

no grafo de conhecimento formado a partir da consulta ao detector de participantes.

4.1.1. O Algoritmo COLLECT

Cada processo p_i que executa COLLECT adquire conhecimento sobre os participantes do sistema, considerando que $f < k$ processos podem falhar. No início p_i só conhece a si próprio. Então, seu conhecimento passa a ser definido pelo que é fornecido pelo seu detector de participantes. Neste instante, o conjunto de conhecidos de p_i é composto pelo próprio p_i e os processos que estão a uma distância de comprimento 1 de p_i no grafo de conhecimento G_{di} definido pelo detector de participantes. A partir daí os processos colaboram trocando informações de modo a adicionar ao seu o conhecimento daqueles processos que eles já conhecem, i.e., fazem parte do seu *i.known*. O algoritmo contém um laço que cuida dessa colaboração. Cada vez que o laço é executado, os nós que estão a uma distância de 1, em G_{di}, de cada $p_j \in i.known$ são acrescentados a *i.known*, de modo que a cada execução do laço aumenta em 1 a distância de p_i aos nós que são acrescentados. Note-se que cada *i*.PD é chamado uma única vez no algoritmo, o que garante que a configuração inicial da conectividade do conhecimento do sistema é consistente para todos os nós do sistema e define um grafo comum $G_{di} = (V, E)$. Se PD $\in$ *k-SCO*, COLLECT retorna Π, se PD $\in$ *k-OSR*, ao final do algoritmo cada processo terá conhecimento de todos os nós de seu componente *k*-fortemente conexo, mais os nós alcançáveis de outros componentes (pelo menos os nós do componente poço).

Descrição. Inicialmente p_i requisita o seu PD para obter o conhecimento inicial do sistema, armazenando-o no campo *known* do registrador compartilhado, podendo portanto ser lido pelos outros processos (linhas 7, 9). Nesse instante, p_i será o único processo na variável privada ***updated*** (linha 10). Na fase de ampliação do conhecimento, cada processo p_i vai adicionar ao seu *i.known* a visão de todos os seus processos conhecidos. Para isso, enquanto houver participantes p_j conhecidos de p_i cujos conjuntos de participantes conhecidos (a menos daqueles faltosos) ainda não foram adicionados a *i.known* p_i continua a executar (linha 11). Então todo p_j nesta situação e que já recebeu retorno de *j*.PD, será adicionado a *updated* e terá seu conjunto de conhecidos adicionado a R_i*.known* (linhas 13-15). Quando p_i sai do while, ajusta seu flag para **true**, indicando que já terminou sua coleta (linha 16).

Lema 1. Na execução do Algoritmo 1 (COLLECT), a despeito de $f < k < n$ falhas: p_j é alcançável a partir de p_i em G_{di} por *k* caminhos disjuntos nos nós $\Leftrightarrow$ a partir de algum instante, $p_j \in i.known$, dado que p_i não termine antes.

Prova. [$\Rightarrow$] p_j é alcançável a partir de p_i em G_{di} por *k* caminhos disjuntos nos nós $\rightarrow p_j \in i.known$.

(A): No início de COLLECT, *i.known* = *i*.PD, então todo p_j alcançável a partir de p_i, que se encontra a uma distância=1 de p_i, está em *i.known*.

(B): Nos laços **while** (linha 11) e **for** (linha 12), p_i adiciona a *i.known* a visão de todo $p_j \in i.known$ (linhas 13-14), desde que p_j já tenha incluído *j*.PD em *j.known*. Além disso, p_i inclui *j* em *updated* (linha 15) (note que p_j permanece em |*i.known*| - |*updated*| até então). Assim, na primeira execução do while, uma vez que *j.known* = *j*.PD, temos que *i.known* = *i.known* $\cup$ *j*.PD para todo *j* que está em *i*.PD.

(C): Por suposição, existem pelo menos *k* caminhos distintos pelos quais p_i alcança p_j em G_{di}. Então, a falha de $f < k$ nós deixa pelo menos um caminho *P*: $p_i = p_0, p_1, ..., p_{y-1}$,

$p_y = p_j$ de nós corretos entre p_i e p_j. Vamos provar por indução sobre y que a partir de algum instante, $p_y \in i.known$. A base $y = 1$ é trivial (A). Assuma que a alegação é válida para todo p_y, $y > 1$. Uma vez que por indução $p_{y-1} \in i.known$, então em algum momento a visão de p_{y-1} será adicionada à visão de i. Como p_y é alcançável a partir de p_{y-1}, temos que $p_y \in (y\text{-}1).PD$ e portanto p_y estará na visão de p_{y-1} após a chamada de $(y\text{-}1).PD$. Logo p_y acabará por entrar na visão de i, isto é, $p_y \in i.known$.

[⇐] a partir de algum instante, $p_j \in i.known \rightarrow p_j$ é alcançável a partir de p_i em G_{di} por k caminhos disjuntos nos nós.

(A): Por construção, temos que no início de COLLECT, $i.known = i.known_0 = i.PD$. Seja $t = 1$ a primeira iteração do laço while (linha 11), e $i.known_t$ a visão de p_i ao final desta iteração. Temos que para todo $p_j \in i.known_0$, $j.PD$ será adicionado a $i.known_0$ de modo que $i.known_1 = i.known_0 \cup j.PD$. Assim, todo $p_y \in j.known_0$, o qual é alcançável a partir de j, será alcançável a partir de p_i através de p_j.

(B): Pela propriedade da inclusão da informação, temos que $i.known_0 \subseteq i.known_1 \subseteq ... \subseteq i.known_t$ .Vamos provar por indução sobre t que $p_t \in i.known_t$ é alcançável a partir de $p_i \in i.known_0$. A base $t = 1$ é trivial. Assuma que a alegação é válida para $t > 1$. Uma vez que por indução $p_{t-1} \in i.known_{t-1}$ é alcançável a partir de p_i; e $p_t \in i.known_{t-1}$, é alcançável a partir de p_{t-1}; então $p_t \in i.known_t$ é alcançável a partir de p_i através de p_{t-1}.

```
Algoritmo 1    COLLECT( )
constant:
(1)     f : int // upper bound on the number of crashes
variables:
(2)     pr: set of nodes
(3)     R_i.flag: boolean
(4)     R_i.known: set of nodes
(5)     updated: set of nodes
(6)     i, j: node

** All nodes **

INIT
(7)     pr = i.PD;
(8)     write (R_i.flag, false);
(9)     write (R_i.known, pr);
(10)    updated ={i} ; pr = ∅;

AMPLIAÇÃO DO CONHECIMENTO
(11)    while (|updated| < |R_i.known| - f):
(12)        for each j ∈ (R_i.known \ updated):
(13)            if (read(R_j.known, pr) ≠ ∅) then
(14)                insert(R_i.known, pr);
(15)                updated = updated ∪ j;
(16)    write(R_i.flag, true);
(17)    return;
```

Figura 2. Algoritmo 1 COLLECT( ): Constrói a visão do sistema para cada processo

Lema 2. Em um sistema $AS[\pi, f]$ estendido com PD *k-SCO*, o Algoritmo 1 (COLLECT) satisfaz às seguintes propriedades, a despeito de $f < k < n$ falhas: (i) Terminação: Todo nó correto p_i termina a execução. (ii) Corretude (*safety*): Todo nó correto p_i retorna

$i.known = \prod$.

Prova. Corretude (A): Por suposição, G_{di} é k-fortemente conectado. Sejam p_i, p_j, $p_y \in G_{di}$. Pelo teorema de Menger (*Observação 1*), considerando que $f < k < n$ nós podem falhar, temos:(B1) $p_i \rightsquigarrow p_j$ por um caminho de nós corretos porque p_j é alcançável a partir de p_i por pelo menos $k > f$ caminhos disjuntos nos nós; (B2) $\exists\ p_y$ correto, $p_j \in y.known$, porque $\forall\ p_j \in G_{di}$ pertencerá a pelo menos k conjuntos visão retornado pelo PD; uma vez que $f < k$, há pelo menos um p_y correto, o qual terá $p_j \in y$.PD e assim $p_j \in y.known$. (C): Pelo Lema 1, sabemos que uma vez que $p_i \rightsquigarrow p_j$ por um caminho de nós corretos, em algum momento $p_j \in i.known$.

Terminação Por construção, temos que a única condição de parada do algoritmo é $|updated| \geq |i.known| - f$ (linha 11). O conjunto *updated* representa os nós conhecidos de p_i cuja visão foi adicionada à visão de p_i; portanto $updated \subseteq i.known$. Inicialmente *updated* possui um elemento (linha 10) e vai aumentando de tamanho à medida que a visão de cada $p_j \in i.known$ é adicionada a $i.known$. Cada p_j que é inserido em *updated* é extraído de $i.known$. Logo, em algum instante $|updated| \geq |i.known| - f$ e p_i termina.

Lema 3. Em um sistema $AS[\pi, f]$ estendido com PD *k-OSR*, o Algoritmo 1 (COLLECT) satisfaz às seguintes propriedades, a despeito de $f < k < n$ falhas: Terminação: Todo nó correto p_i termina a execução. Corretude (*safety*): Todo nó correto p_i retorna um conjunto $i.known$: $p_i \in G_{sink} \Leftrightarrow i.known = G_{sink}$; e $p_i \notin G_{sink} \Leftrightarrow i.known \supsetneq G_{sink}$.

Prova. Corretude Caso 1: $p_i \in G_{sink}$ [$\Rightarrow$] $p_i \in G_{sink} \rightarrow i.known = G_{sink}$. G_{sink} é k-fortemente conectado, portanto $\forall\ p_i, p_j, p_y \in G_{sink}$: a *Observação 1* se mantém. Do Lema 1, p_j passa a pertencer a $i.known$. Suponha, por contradição, que $\exists\ p_j \in i.known$: $p_j \notin G_{sink}$. Como existe um caminho de p_i a p_j (Lema 1), $p_i \notin G_{sink}$, o que é uma contradição.

[$\Leftarrow$] $i.known = G_{sink} \rightarrow p_i \in G_{sink}$. Pelo Lema 1, $\forall\ p_i, p_j \in G_{sink}$, $\exists\ k$ caminhos em G_{di} que levam de p_i a p_j. Por suposição, $i.known = G_{sink}$, pela definição de G_{sink}, não existem caminhos de p_i a p_j, $p_j \notin G_{sink}$. Suponha, por contradição, que $p_i \notin G_{sink}$. Como $p_i \in i.known$, temos que $i.known \neq G_{sink}$, contradizendo nossa suposição.

Caso 2: $p_i \notin G_{sink}$ [$\Rightarrow$] $p_i \notin G_{sink} \rightarrow i.known \supsetneq G_{sink}$. Pela *Definição 4*, existem pelo menos k caminhos disjuntos entre quaisquer $p_i \notin G_{sink}$, $p_j \in G_{sink}$. Portanto $p_j \in i.known$ (Lema 1). Como $p_i \notin G_{sink}$ e $p_i \in i.known$, temos que $i.known \supsetneq G_{sink}$.

[$\Leftarrow$] $i.known \supsetneq G_{sink} \rightarrow p_i \notin G_{sink}$. Se $i.known \supsetneq G_{sink}$, $\exists\ p_j \in i.known$, $p_j \notin G_{sink}$. Pelo Lema 1, se $p_j \in i.known$, existem k caminhos de p_i a p_j. Pela definição de G_{sink}, não existe caminho de $p_i \in G_{sink}$ a $p_j \notin G_{sink}$, portanto, $p_i \notin G_{sink}$.

Terminação A prova segue direto da prova da propriedade de terminação do Lema 2.

4.2. O Algoritmo SINK

O algoritmo SINK (Figura 3a) usa o conhecimento adquirido pelos processos com o algoritmo COLLECT para determinar quais nós pertencem ao componente poço. Essa informação será usada no algoritmo CONSENSUS.

Descrição. Na fase inicial p_i chama COLLECT para construir sua visão do sistema, armazenando-a em $R_i.known$ (linha 8). Na fase de verificação, SINK checa o $R_j.known$ de cada p_j conhecido de p_i ($p_j \in R_i.known$), a menos de f (linha 9). Para cada j que ainda não tenha sido checado por p_i, desde que j tenha concluído COLLECT (linhas 10-11), o algoritmo verifica se p_i está em $R_j.known$ ($p_i \in R_j.known$). Em caso positivo, j é adicionado ao conjunto de nós checados e a verificação continua para os próximos $p_j \in R_i.known$ (linhas 12-15). Se para algum j, verifica-se que $i \notin R_j.known$, *então* já se sabe que i não está no poço, e SINK retorna **false** imediatamente (linhas 13-14). Caso p_i ainda não tenha terminado COLLECT (linha 11), p_j permanece como não checado e a verificação continua para os próximos p_j também não checados. A verificação é concluída de uma das duas maneiras: (a) p_i não está no poço, o que é detectado para algum j tq $i \notin R_j.known$ (linhas 13-14); ou (b) $i \in R_j.known$ para todo j conhecido de i, a menos de f (linha 16).

Lema 4. Em um sistema $AS[\pi, f]$ estendido com PD *k-OSR*, o Algoritmo 2 (SINK) satisfaz às seguintes propriedades, a despeito de $f < k < n$ falhas: Terminação: Todo nó correto p_i termina a execução decidindo se ele pertence ou não ao componente G_{sink}. Corretude (*safety*): $p_i \in G_{sink} \Leftrightarrow p_i$ retorna *true*; e $p_i \notin G_{sink} \Leftrightarrow p_j$ retorna *false*.

Prova. Terminação Cada processo p_i em SINK irá checar todo processo p_j em sua visão local ($i.known$) retornado por COLLECT (linha 10). Se p_j já tiver terminado COLLECT, p_i verifica se está presente em $j.known$ (linhas 11-13) e em caso positivo insere p_j em seu conjunto de nós checados (*checked*) (linha 15). Inicialmente, $checked = \{p_i\}$, mas à medida que o algoritmo prossegue, uma vez que $f < k < n$, *checked* vai sendo ampliado até que uma das duas condições é satisfeita: (A) $|checked|$ atinge $|i.known| - f$: esta situação ocorre quando p_i é encontrado em todo $p_j \in checked$; então $|checked|$ atinge o tamanho máximo e SINK retorna *true*. (B) p_i não é encontrado em algum p_j checado e, constatando que não pertence ao componente poço, retorna *false*.

Corretude. Caso 1: $p_i \in G_{sink}$ [$\Rightarrow$] $p_i \in G_{sink} \rightarrow p_i$ retorna *true*. (A): $\forall\ p_i \in G_{sink} \Leftrightarrow i.known = G_{sink}$ (Lema 3); portanto, $\forall\ p_j, p_y \in G_{sink}$, $p_j \in y.known$. Além disso, pela *Definição 4*, existem pelo menos k caminhos disjuntos de $p_j \notin G_{sink}$ a $p_i \in G_{sink}$, logo, $p_i \in j.known$, $\forall\ j \in G_{di}$. (B): Por construção, e de (A), a condição da linha 13 sempre será falsa, impedindo que a linha 14 seja executada, o que forçará a saída do algoritmo somente após checar todos os $p_j \in i.known \backslash checked$, retornando *true*.

[$\Leftarrow$] p_i retorna $true \rightarrow p_i \in G_{sink}$. Suponha que $p_i \notin G_{sink}$. Então $\exists\ p_j\ (\in G_{sink})$ tal que $p_i \notin j.known$ e o algoritmo retornará *false* (linhas 13-14), contradizendo nossa suposição.

Caso 2: $p_i \notin G_{sink}$ [$\Rightarrow$] $p_i \notin G_{sink} \rightarrow p_i$ retorna *false*. A prova vem direto da prova da Caso 1, parte 2: se $p_i \notin G_{sink}$, então $\exists\ p_j\ (\in G_{sink})$ / $p_i \notin j.known$ e o algoritmo retorna false.

[$\Leftarrow$] p_i retorna $false \rightarrow p_i \notin G_{sink}$. O algoritmo só retorna false quando checar algum $p_j \in i.known$ para o qual $p_i \notin j.known$ (linhas 13-14). Já vimos que $\forall p_i \in G_{sink}$, para todo $p_j \in i.known$, $p_i \in j.known$ (Lema 3), então se $p_i \notin j.known$, $p_i \notin G_{sink}$.

4.3. O Algoritmo CONSENSUS

O algoritmo CONSENSUS Figura 3b) é executado para decidir por um valor dentre os propostos. Como apenas os nós do componente poço são alcançáveis por todos os demais nós do grafo G_{di}, apenas estes nós participam da decisão. Uma vez que estes nós

compartilham a mesma visão do sistema, é possível resolver o consenso usando o detector Ω, desde que haja pelo menos a maioria de nós corretos no componente poço. Após a decisão ter sido tomada, os nós que estão nos demais componentes fortemente conectados podem tomar conhecimento do valor decidido e decidir pelo mesmo valor.

Algoritmo 2 SINK ( )

```
constant:
(1)   f : int // upper bound on the number of crashes
variables:
(2)   pr: set of nodes
(3)   R_i.flag: boolean
(4)   R_i.known: set of nodes
(5)   checked: set of nodes
(6)   i, j: node

** All nodes **

INIT
(7)   pr = ∅; checked = {i};
(8)   COLLECT( );

VERIFICAÇÃO
(9)   while (|checked| < |R_i.known| - f):
(10)      for each j ∈ (R_i.known \ checked):
(11)          if (R_j.flag = true) then
(12)              read (R_j.known, pr);
(13)              if (i ∉ pr) then
(14)                  return false; end if
(15)              checked = checked ∪ {j}; end if
(16)  return true;
```

Algoritmo 3 CONSENSUS( )

```
constant:
(1)   f : int // upper bound on the number of crashes
input:
(2)   value
variables:
(3)   R_i.decision: value
(4)   R_i.known: set of nodes
(5)   d: value
(6)   in_the_sink: boolean
(7)   j: node

** All nodes **
task MAIN: {Main Consensus Task}
(8)   R_i.decision = ⊥; d = ⊥;
(9)   in_the_sink = SINK( );
(10)  if (in_the_sink) then
(11)      fork AGREEMENT;
(12)  else
(13)      fork GETDECISION;

** Node in Sink **
task AGREEMENT: {Underling Consensus}
(14)  Consensus.propose(value);
(15)  upon Consensus.decide(v);
(16)  write (R_i.decision, v);
(17)  return (R_i.decision);

** Node Not in Sink **
task GETDECISION: {Getting Decision}
(18)  while (d = ⊥):
(19)      for each j ∈ R_i.known:
(20)          read( R_j.decision, d);
(21)          if (d ≠ ⊥) then
(22)              write (R_i.decision, d);
(23)  return (R_i.decision);
```

Figura 3 (a) Algoritmo 2 – SINK()– Determina se p_i está no componente poço. (b): Algoritmo 3 – CONSENSUS() –): Realiza o consenso

Descrição. Na fase inicial todos os nós chamam o procedimento SINK para construir sua visão do sistema e verificar se ele pertence ou não ao componente poço (linha 9). Os nós no componente poço então começam a tarefa AGREEMENT a fim de decidir por um valor *v* enquanto que os nós dos demais componentes começam a tarefa GETDECISION para obter a decisão tomada (linhas 10-13). Para tomar a decisão os nós no poço executam um protocolo que usa o detector Ω e realizam o consenso (linha 14). Assim que cada p_i decide disponibiliza sua decisão em $R_i.decision$. Os nós fora do componente poço executam a tarefa GETDECISION para pegar a decisão tomada. Inicialmente a decisão de todo processo $p_i \notin$ poço é nula (⊥ denota um valor default que

não pode ser decidido por um processo.)(linhas 8, 18). p_i tenta pegar a decisão no registrador de um de seus processos conhecidos. Assim que ele encontra uma decisão válida, escreve-a em $R_i.decision$.

Lema 5. O Algoritmo 3 (CONSENSUS) resolve FT-CUP uniforme num sistema $AS[\pi, f]$ estendido com *k-OSR* e Ω, a despeito de $f < k < n$ falhas, assumindo uma maioria de nós corretos no componente poço de $G_{di} \in$ *k-OSR*.

Prova. Validade. Fica garantida já que a decisão tomada é por um valor proposto pelos nós do poço (linha 14) através de um protocolo indulgente clássico (Chandra e Tueg 1996), o qual resolve o consenso com a maioria dos nós corretos no poço (pelo menos $f + 1$). Terminação Para provar que todo nó correto p_i decide, vamos mostrar que p_i termina executando a linha 17 ou 23 de CONSESNSUS. A partir da tarefa MAIN p_i pode tomar um de dois caminhos: Caso 1: se $p_i \in G_{sink}$, executa as linhas 10 e 11 e chama um algoritmo indulgente clássico para realizar o consenso (linha 14). Em decorrência da propriedade de terminação deste algoritmo, p_i toma uma decisão (linha 15), publica esse resultado em $R_i.decision$ e retorna sua decisão para a aplicação. Caso 2: se $p_i \notin G_{sink}$, executa as linhas 12 e 13 e tenta tomar sua decisão baseado no valor decidido por algum nó correto. Assim p_i começa a ler $R_j.decision$, para todo $p_j \in i.known$ (linhas 19, 20). Assim que encontrar um valor válido, decide por este valor e retorna-o para a aplicação (linhas 21-23). Uma vez que pelo menos f +1 nós corretos do poço terão decidido, em algum instante, p_i lerá um valor decidido de $R_j.decision$: ou de $p_j \in$ poço, já que $\forall\, p_j \in G_{sink}$, $p_j \in i.known$ (Lema 3); ou de um dos componentes que tenha decidido antes dele.

Acordo Uniforme O acordo uniforme é garantido pela propriedade de acordo uniforme do algoritmo indulgente utilizado. Assim, todo nó no poço recebe o mesmo valor de decisão (linha 15) e escreve esse valor em seu registrador compartilhado (linha 16). Assim os nós que não estão no poço lerão o mesmo valor, independente de qual seja o registrador lido e também decidirão pelo mesmo valor (linhas 18-23).

5. Conclusão

Neste artigo apresentamos um conjunto de protocolos para a resolução do consenso em um sistema assíncrono sujeito a falhas com participantes desconhecidos, considerando-se o modelo de memória compartilhada. O trabalho se baseia no uso de detectores de participantes os quais provêem os processos com um conhecimento inicial sobre quais processos participam do sistema. Tal conhecimento pode ser modelado como um grafo da conectividade do conhecimento, cujas propriedades definem a classe do detector de participantes. Nós mostramos que nossos algoritmos funcionam corretamente para sistemas munidos com detectores de participantes da classe k-OSR.

Atualmente, as arquiteturas multi-core e o recente paradigma de *software transaction memory* são uma das aplicações mais promissoras para protocolos de memória compartilhada. Os resultados trazidos pelo nosso trabalho são um passo inicial para que soluções práticas, voltadas para esses ambiente reais, além de redes dinâmicas, possam vir a ser obtidas. Como trabalhos futuros pretendemos propor o algoritmo de consenso subjacente, implementar os protocolos num ambiente real e avaliá-los experimentalmente.

6. Referências

Afek, Yehuda; Greenberg, David S.; Merritt, Michael e Taubenfeld, Gadi. (1992). Computing with faulty shared memory. *In Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*. pp 47-58.

Aguilera, M. K., Keidar, I., Malkhi, D., and Shraer, A. (2011). Dynamic atomic storage without consensus. *Journal of ACM*, 58:7:1–7:32.

Alchieri, E. A. P.; Bessani, A. N.; Fraga, J. S.; e Greve, F. (2008). Byzantine consensus with unknown participants. In *12th International Conference on Principles Of Dsstributed* Systems, pages 22–40.

Cavin, D; Sasson, Y e Schiper, A. (2004). Consensus with unknown participants or fundamental self-organization. *In Proc. 3rd Int. Conf. AD-NOC Networks & Wireless (ADHOC-NOW)*, pages 135–148, Vancouver, Springer-Verlag.

Cavin, D; Sasson, Y e Schiper, A. (2005). Reaching agreement with unknown participants in mobile self-organized networks in spite of process crashes.In *Research Report IC*/2005/026, EPFL.

Chandra, T. e Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *In Journal of the ACM*, 43(2):225–267.

Chandra, T.; Hadzilacos, V. e Toueg, S. (1996). The weakest failure detector for solving consensus. *In Journal of the ACM*, 43(4):685–722.

Delporte-Gallet, C. e Fauconnier, H. (2009).Two Consensus Algorithms with Atomic Registers and Failure Detector Ω. *In Lecture Notes in Computer Science, Distributed Computing and Networking*, Volume 5408/2009, pp. 251-262.

Delporte-Gallet, et al (2004). The weakest failure detectors to solve certain fundamental problems in distributed computing. *In: 23th ACM Symposium on Principles of Distributed Computing*.

Fischer, M. J.; Lynch N. A. e Paterson, M. D. (1985). Impossibility of distributed consensus with one faulty process. *In Journal of ACM*, 32(2):374–382.

Greve, Fabiola e Tixeuil, Sebastien. (2007). Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. *In DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 82–91, Washington, USA.IEEE Computer Society.

Greve, Fabiola e Tixeuil, Sebastien. (2010). Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. *Technical report*, INRIA, Paris, France.

Guerraoui, R. e Raynal, M. (2007). The Alpha of Indulgent Consensus. *In The Computer Journal*, Volume 50, nº 1. Oxford University Press, Reino Unido.

Lamport, L. (1986). On Interprocess Communication. *In Distributed Computing* Vol 1, pp 77-101.

Lo, W. e Hadzilacos, V. (1994).Using Failure Dtectors to Solve Consensus in Asynchronous Shared-Memory Systems. *In Lecture Notes in Computer Science, Distributed Algorithms*, Volume 857/1994, pp. 280-295.

Yellen, J. e Gross, J. Graph Theory and its Applications. CRC Press, 1998.