Programa de Pós-Graduação em Informática Aplicada - PPGIA

Doutorado em Ciência de Dados e Inteligência Artificial

Disciplina: M907 - Sistemas Distribuídos | Professor Dr. Nabor Mendonça
Antonio Marcos Aires Barbosa | Matrícula: 2016397

TRABALHO 01:

- Análise de viabilidade em disponibilidade

Exercício 1.1:

Deduza a fórmula matemática que calcula a disponibilidade de um serviço replicado em múltiplos servidores

A fórmula deve ser definida em termos dos seguintes parâmetros:

n – número de servidores ($n > 0$)
k – número mínimo de servidores disponíveis necessário para o serviço ser acessado de forma consistente ($0 < k \leq n$)
p – probabilidade de cada servidor estar disponível em um dado instante ($0 \leq p \leq 1$)

Exercício 1.2:

Implemente a(s) fórmula(s) derivada(s) no exercício 1.1 na sua linguagem de preferência, e utilize sua implementação para mostrar como a disponibilidade do serviço é afetada para diferentes valores de n, k e p

Utilize os valores para o caso sem replicação ($n = k = 1$) como base de comparação

Bônus:

Organize os resultados obtidos em uma tabela ou planilha, e visualize-os na forma de gráficos 2D

RESPOSTAS:

Derivação dos casos mais simples

Para os casos extremos, onde k = 1 e k = n:

1. Caso k = 1:

Nesse caso, o número mínimo de servidores necessários para o serviço ser acessado de forma consistente é igual a 1. Isso significa que, mesmo que apenas um servidor esteja disponível, o serviço pode ser considerado disponível.

Neste caso, o cálculo para a disponibilidade nesse caso é dada por:

$$A = 1 - (P(0))$$

Agora, vamos substituir P(0) na fórmula. Como k = 1, a probabilidade de falha é a probabilidade de todos os servidores estarem indisponíveis.

```
P(0) = C(n, 0) * p^0 * (1 - p)^(n - 0)
P(0) = 1 * 1 * (1 - p)^n
P(0) = (1 - p)^n
```

Portanto, a fórmula para a disponibilidade quando k = 1 (operação de consulta) é:

```
A = 1 - (1 - p)^n
```

Essa fórmula nos fornece a disponibilidade do serviço quando apenas um servidor precisa estar disponível.

2. Caso k = n:

Nesse caso, o número mínimo de servidores necessários para o serviço ser acessado de forma consistente é igual ao número total de servidores n. Isso significa que todos os servidores precisam estar disponíveis para que o serviço seja considerado disponível.

A fórmula para calcular a disponibilidade nesse caso é dada por:

```
A = 1 - (P(0) + P(1) + P(2) + ... + P(n-1))
```

Agora, vamos substituir P(x) na fórmula para cada valor de x de 0 a n-1. Como k = n, a probabilidade de falha é a probabilidade de menos de n servidores estarem disponíveis.

```
P(x) = C(n, x) * p^x * (1 - p)^(n - x)
```

Substituindo na fórmula de disponibilidade:

```
A = 1 - (P(0) + P(1) + P(2) + ... + P(n-1))
A = 1 - [C(n,0) * p^0 * (1-p)^(n-0) + C(n,1) * p^1 * (1-p)^(n-1) + ... + C(n, n-1)
* p^(n-1) * (1-p)^(n-(n-1))]
```

Essa fórmula nos fornece a disponibilidade do serviço quando todos os servidores precisam estar disponíveis.

Nestes casos extremos do cálculo da disponibilidade de um serviço replicado em múltiplos servidores, no caso k = 1 temos o que seria uma operação de consulta; e k = n temos uma operação de atualização;

Resposta generalizada:

A disponibilidade de um serviço replicado em múltiplos servidores pode ser calculada com a fórmula da probabilidade de falha, através do cálculo do complemento da disponibilidade temos a probabilidade de falha do serviço.

A probabilidade de falha é igual à probabilidade de que o número de servidores disponíveis seja menor que o número mínimo necessário (k).

A probabilidade de que exatamente x servidores estejam disponíveis em um dado instante é dada pela multiplicação de:

```
combinação de n dado x,
multiplicada pela probabilidade de cada servidor estar disponível (p) elevada a x,
multiplicada pela probabilidade de cada servidor estar indisponível (1-p) elevada à
diferença entre n e x (n-x).
```

Matematicamente, a fórmula para essa probabilidade é dada por:

$$P(x) = C(n, x) * p^x * (1 - p)^(n - x)$$

Onde:

- C(n, x) é o número de combinações de n objetos escolhendo x (coeficiente binomial), calculado por C(n,x) = n!/(x!*(n-x)!)
- p^x representa a probabilidade de cada servidor estar disponível elevada a x
- (1 - p)^(n - x) representa a probabilidade de cada servidor estar indisponível elevada a (n - x)

A probabilidade de falha do serviço (F) é dada pelo somatório das probabilidade de falhas de cada quantidade x servidores estarem disponíveis:

```
F = P(0) + P(1) + P(2) + ... + P(k-1)
```

Onde P(x) é a probabilidade de exatamente x servidores estarem disponíveis em um dado instante, calculada como:

```
P(x) = C(n,x) * p^x * (1-p)^(n-x)
```

Em termos dos diferentes cenários possíveis em relação à disponibilidade dos servidores replicados, essa representação pode ser calculada pelo somatório das probabilidades de falha do serviço. Na forma de somatório, o valor de x varia de 0 a k-1, e cada termo P(x) é calculado com base nos coeficientes binomiais (C(n, x)), a probabilidade de cada servidor estar disponível (p) e a probabilidade de cada servidor estar indisponível (1 - p). Portanto, a probabilidade de falha (F) pode ser escrita como:

$$F = P(0)+P(1)+P(2)+ ... + P(k-1)$$

Ou seja:

$$F = \sum[C(n,x)*p^x*(1-p)^(n-x)], \text{ para x de [0 a k-1]}$$

onde:

```
F = probabilidade de falha (F)
n – número de servidores (n > 0)
k – número mínimo de servidores disponíveis necessário para o serviço ser acessado
de forma consistente (0 < k ≤ n)
p – probabilidade de cada servidor estar disponível em um dado instante (0 ≤ p ≤ 1)
```

# #

Implementando em Python

In [1]:
```python
# !pip install seaborn
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from matplotlib.ticker import FuncFormatter
from matplotlib.lines import Line2D

def calculate_availability(n, k, p):
    if n <= 0 or k <= 0 or k > n or p < 0 or p > 1:
        raise ValueError("Valores inválidos para os parâmetros.")

    def calculate_probability(x):
        return np.math.comb(n, x) * p**x * (1 - p)**(n - x)

    probabilities = [calculate_probability(x) for x in range(k)]
    availability = 1 - sum(probabilities)
    return availability

def plot_availability_curve(n_values, k_values, p_values, unit_cost):
    sns.set(style="whitegrid", rc={"axes.grid.axis": "y", "axes.grid.which": "major"})

    # Definir uma paleta de cores para atribuir cores diferentes às curvas de acordo com k
    colors = sns.color_palette("husl", len(k_values))

    fig, ax1 = plt.subplots(figsize=(12, 11))
    ax2 = ax1.twinx()

    lines = []  # Armazenar as linhas da legenda adicional

    for p in p_values:
        for i, k in enumerate(k_values):
            availabilities = []
            total_costs = []
            try:
                for n_val in n_values:
                    availability = calculate_availability(n_val, k, p)
                    availabilities.append(availability * 100)  # Exibindo em formato percentual
                    total_costs.append(n_val * unit_cost / 1000)  # Exibindo em k$
            except ValueError as e:
                print(f"Erro: {str(e)}")
                availabilities.append(float('nan'))
                total_costs.append(float('nan'))

            line = ax2.plot(n_values, availabilities, label=f'p = {p}, k = {k}', color=colors[i])
            lines.append(line[0])  # Adicionar a linha à lista de linhas da legenda adicional

            # Adicionar rótulos dos pontos de dados nas curvas
            for x, y in zip(n_values, availabilities):
                ax2.annotate(f"({k}, {x})", (x, y), xytext=(0, -10), textcoords="offset points",
                             ha='center', va='top', fontsize=8)

        # Adicionar rótulos nas bases internas das colunas
        for x, y in zip(n_values, total_costs):
            ax1.annotate(f"${y:.1f}k", (x, y), xytext=(0, -30), textcoords="offset points",
                         ha='center', va='top', fontsize=14)

        ax1.bar(n_values, total_costs, width=0.8, alpha=0.3, label='Custo (k$)', color='cyan', zorde

    ax1.set_ylabel('Custo total com servidores (Unidade: 1.000 USD/mês)')
    ax2.set_xlabel('Número total de servidores (n)')
    ax2.set_ylabel('Disponibilidade do Serviço (%)')  # Exibindo em formato percentual

    ax2.set_title('Curva de Disponibilidade e Custo Total')

    # Configurar escala logarítmica inversa no eixo vertical
    ax2.set_ylim(10, 102)

    # Formatar rótulos do eixo vertical
    def percent_formatter(x, pos):
        return f"{int(x)}%"
    ax2.yaxis.set_major_formatter(FuncFormatter(percent_formatter))
```

```
    # Criar legenda original
    legend1 = ax2.legend(loc='upper left', bbox_to_anchor=(0, 0.9))

    # Adicionar a legenda adicional
    legend_labels = [f'k = {k}' for k in k_values]
    legend2 = ax2.legend(lines, legend_labels, loc='upper left', bbox_to_anchor=(0, 1))

    # Adicionar as duas legendas ao mesmo tempo
    ax2.add_artist(legend1)
    ax2.add_artist(legend2)

    # Ajustar o layout do gráfico para evitar sobreposição de elementos
    plt.subplots_adjust(left=0, right=0.95, top=0.98, bottom=0.1)

    # Adicionar espaço para as legendas antes do eixo esquerdo
    ax1.set_xlim(min(n_values)-2, max(n_values)+1)

    plt.show()
```

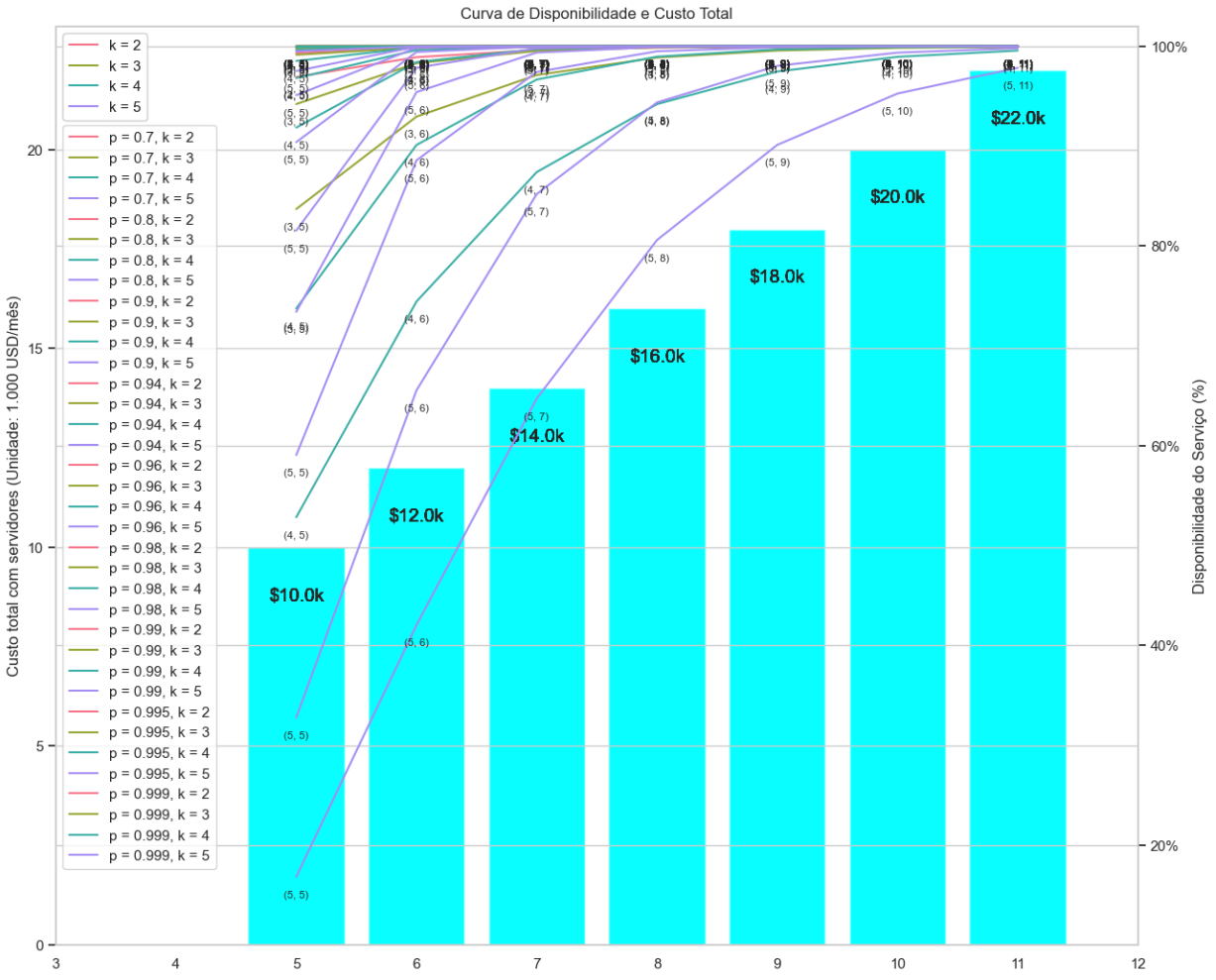
Plotagem

In [2]:
```
# Valores para o número total de servidores (n) e mínimos de servidores necessários (k)
n_values = range(5,12)
unit_cost = 2000

k_values = range(2,6)

# Lista de valores de p
p_values = [0.7, 0.8, 0.9, 0.94, 0.96, 0.98, 0.99, 0.995, 0.999]

plot_availability_curve(n_values, k_values, p_values, unit_cost)
```



==========

Leitura suplementar sobre resiliência

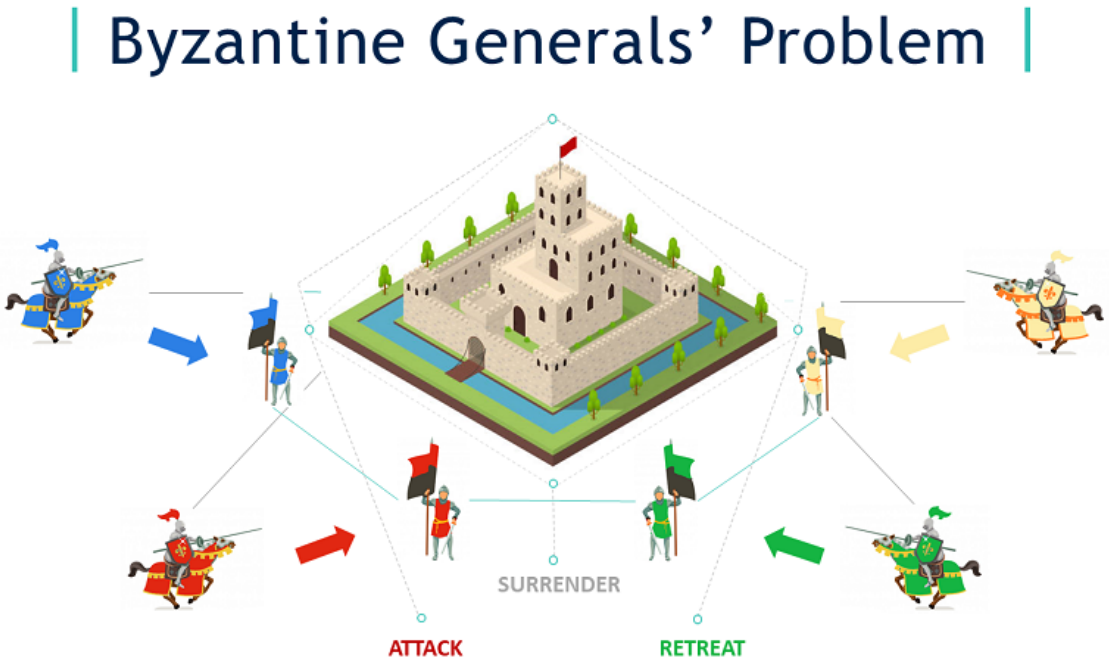
O problema de disponibilidade em sistemas distribuídos tem semelhanças com o problema do consenso distribuído, inicialmente descrito por Edsger W. Dijkstra em um artigo de 1974 intitulado "Self-stabilizing systems in spite of distributed control". (https://dl.acm.org/doi/pdf/10.1145/361179.361202)

Na abordagem de Dijkstra, o problema do consenso distribuído consiste em encontrar um valor de consenso entre um grupo de processos em um sistema distribuído, onde cada processo pode ter diferentes valores iniciais e falhar de forma arbitrária durante a execução.

Mais propriamente dito no tema de Sistemas Distribuídos, em 1980, tal problema foi apresentado como o problema dos Generais Bizantinos, conforme proposto pelo matemático e cientista da computação Leslie Lamport, em 1982. Remontando à um contexto histórico antigo, onde a confiança entre generais no campo de guerra era tratada, visualizando a dificuldade e possibilidade de um dos generais não receber a mensagem.

O problema, também conhecido como Bizantine Fault Tolerance (BRT), foi formalizado e descrito por Lamport, Robert Shostak e Marshall Pease no artigo "The Byzantine Generals Problem" publicado na revista "ACM Transactions on Programming Languages and Systems" com amplas aplicações em sistemas distribuídos e criptografia.

O Problema estudado por Dijkstra é semelhante ao problema dos Generais Bizantinos de Lamport, no sentido de que, ambos envolvem a coordenação de diferentes partes em um sistema distribuído, sujeitas a falhas e a possibilidade de comportamento malicioso, porém, enquanto o problema dos Generais Bizantinos se concentra na decisão entre duas opções, o problema do consenso distribuído é geralmente formulado como o processo de chegar a um consenso em um único valor. São necessárias soluções criativas para garantir a confiabilidade e a integridade do sistema distribuído.



Problema dos generais bizantinos (LAMPORT, 1980).

```
Um general comandante deve enviar uma ordem para seus n - 1 tenentes-generais de
modo que
IC1. Todos os tenentes leais obedecem à mesma ordem.
IC2. Se o general comandante for leal, todo tenente leal obedecerá à ordem que ele
enviar.
```

As condições IC1 e IC2 são chamadas de condições de consistência interativa. Observe que, se o comandante for leal, IC1 decorre de IC2. No entanto, o comandante não precisa ser leal.

Para resolver nosso problema original, o i-ésimo general envia seu valor de v(i) usando uma solução para o Problema dos Generais Bizantinos para enviar a ordem "use v(i) como meu valor", com os outros generais atuando como tenentes.

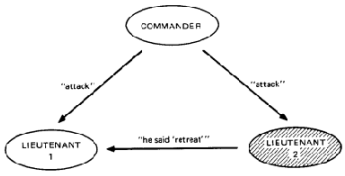


Fig. 1. Lieutenant 2 a traitor.

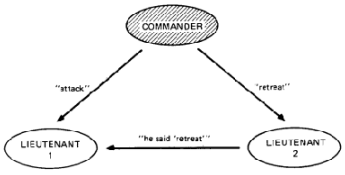


Fig. 2. The commander a traitor.

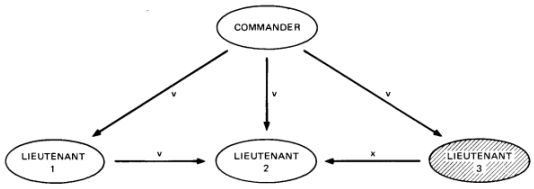


Fig. 3. Algorithm OM(1); Lieutenant 3 a traitor.

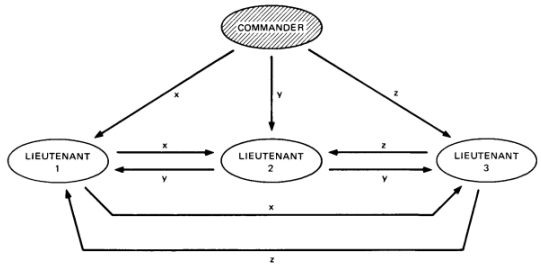


Fig. 4. Algorithm OM(1); the commander a traitor.

fonte: (LAMPORT, 1980) disponível em: SRI International https://lamport.azurewebsites.net/pubs/byz.pdf

Tipos de soluções para BTP (LAMPORT, 1980):

Sem solução possível:

> Com apenas três generais, nenhuma solução pode funcionar na presença de um único traidor.

Uma solução com mensagens orais:

Lamport demonsta que, para uma solução do Problema dos Generais Bizantinos usando mensagens orais, para lidar com m traidores, deve haver pelo menos 3m+1 generais. Primeiro, Lamport mostra a solução que funciona para 3m+1 ou mais generais. Onde uma "mensagens oral" se refere à capacidade de cada general deve executar algum algoritmo que envolve o envio de mensagens para os outros generais, e assumimos que um general leal executa corretamente seu algoritmo. A definição de uma mensagem oral está incorporada nas seguintes suposições que fazemos para o sistema de mensagens dos generais:

> A1. Cada mensagem enviada é entregue corretamente.
> A2. O destinatário de uma mensagem sabe quem a enviou.
> A3. A ausência de uma mensagem pode ser detectada.
>
> TEOREMA 1. Para qualquer m, o Algoritmo OM(m) satisfaz as condições IC1 e IC2 se houver mais de 3m generais e, no máximo, m traidores.

Uma solução com mensagens assinadas:

Como se pode ver no cenário da figura acima, é a capacidade dos traidores de mentir que torna o Problema dos Generais Bizantinos tão difícil. Lamport explica que o problema torna-se mais fácil de resolver se pudermos restringir essa capacidade. Uma maneira de fazer isso é permitir que os generais enviem mensagens assinadas que não podem ser falsificadas. Mais precisamente, adicionamos a A1-A3 o A4

> (a) A assinatura de um general leal não pode ser falsificada, e qualquer alteração no conteúdo de suas mensagens assinadas pode ser detectada.
> (b) Qualquer pessoa pode verificar a autenticidade da assinatura de um general.

```
TEOREMA 2. Para qualquer m, o Algoritmo SM(m) resolve o Problema dos Generais
Bizantinos se houver, no máximo, m traidores.
```

Vias de comunicação ausentes:

Para todas as soluções acima é assumdido que um general pode enviar mensagens diretamente para todos os outros generais. Para tratar casos de vias de comunicação ausente Lamport por fim remove essa suposição. É suporto pelos autores que barreiras físicas impõem algumas restrições sobre quem pode enviar mensagens para quem.

Consideram os generais formam os nós de um grafo não direcionado simples, 2 finito G, onde um arco entre dois nós indica que esses dois generais podem enviar mensagens diretamente um ao outro. Agora estendemos os algoritmos OM(m) e SM(m), que assumem que G é completamente conexo, para grafos mais gerais.

```
TEOREMA 3. Para qualquer m > 0 e qualquer p >= 3m, o Algoritmo OM(m, p) resolve o
Problema dos Generais Bizantinos se houver, no máximo, m traidores.

TEOREMA 4. Para qualquer m e d, se houver no máximo m traidores e o subgrafo de
generais leais tiver diâmetro d, então o Algoritmo SM(m + d - 1) (com a modificação
acima) resolve o Problema dos Generais Bizantinos.
```

Conclusões sobre Sistemas Confiáveis (LAMPORT, 1980)

Após apresentar as várias soluções para o Problema dos Generais Bizantinos e mostrar como elas podem ser usadas na implementação de sistemas computacionais confiáveis, Lamport observa que essas soluções são computacionalmente dispendiosas, tanto no tempo quanto no número de mensagens necessárias.

Os algoritmos OM(m) e SM(m) requerem caminhos de mensagem de comprimento até m+1. Em outras palavras, cada tenente pode ter que esperar pelas mensagens que se originaram no comandante e foram retransmitidas por meio de outros tenentes. Lamport afirma que Fischer e Lynch mostraram que isso deve ser verdade para qualquer solução que possa lidar com traidores, de modo que nossas soluções são ótimas nesse aspecto. Nossos algoritmos para um grafo que não está completamente conectado requerem caminhos de mensagem de comprimento até m+d, onde d é o diâmetro do subgrafo de generais leais. Os autores defenderm que isso também seja ótimo.

Os algoritmos OM(m) e SM(m) envolvem o envio de até (n-1)(n-2) ... (n-m-1) mensagens. O número de mensagens separadas necessárias certamente pode ser reduzido pela combinação de mensagens. Também pode ser possível reduzir a quantidade de informações transferidas, mas isso não foi estudado em detalhes no artigo analisado de Lamport.

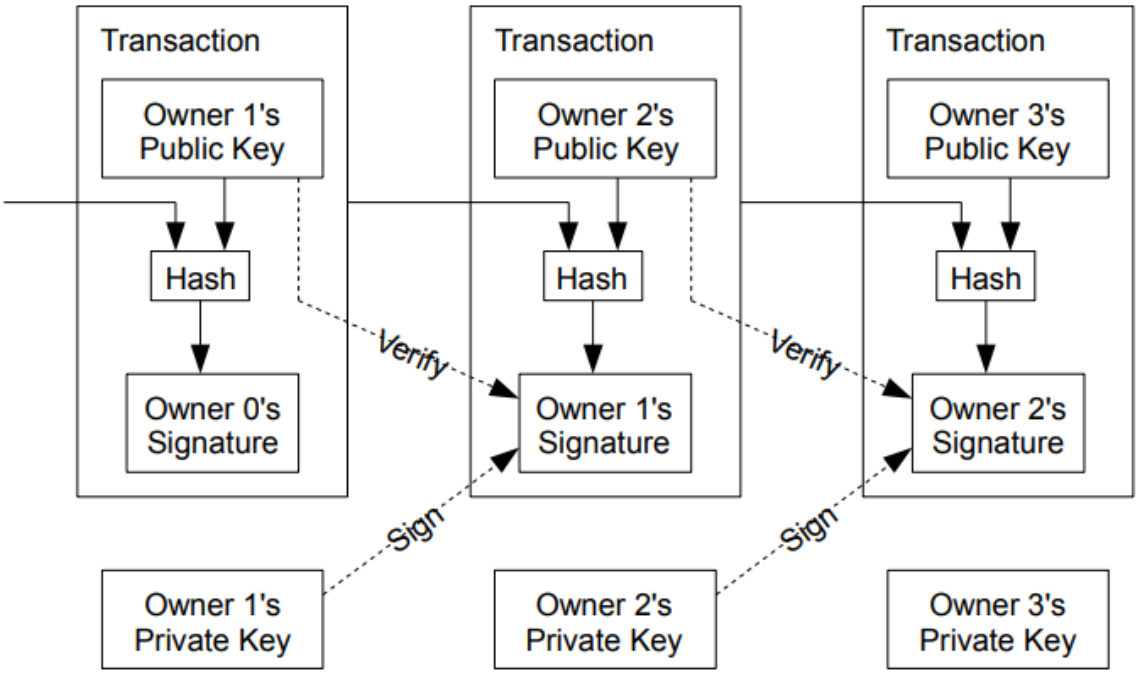
Os autores encerram por defender que alcançar a confiabilidade diante do mau funcionamento arbitrário é um problema difícil, com solução inerentemente cara. A única maneira de reduzir o custo é fazer suposições sobre o tipo de falha que pode ocorrer. Por exemplo, muitas vezes assume-se que um computador pode não responder, mas nunca responderá incorretamente. No entanto, quando é necessária uma confiabilidade extremamente alta, tais suposições não podem ser feitas e é necessário o custo total de uma solução de generais bizantinos.

Aplicações atuais (Wikipedia)

Tanto o artigo de Dijkstra como o artigo de Leslie Lamport, discutem como encontrar um valor de consenso, uma estratégia em comum, mesmo que algumas partes possma apresentar falhas, na metáfora de Lamport seriam generais desonestos, mensagens falsas ou falhas nas mensagens. Lamport propôs um algoritmo baseado em mensagens assinadas digitalmente para resolver o problema.

Já mais modernamente, Satoshi Nakamoto, em seu artigo "Bitcoin: A Peer-to-Peer Electronic Cash System" (disponível em: https://bitcoin.org/bitcoin.pdf), propôs um sistema de consenso distribuído para transações financeiras em um contexto de rede de computadores distribuídos. O sistema de Nakamoto ficou conhecido

como blockchain, que além da criptografia (já abordada por Lamport na forma de mensagens assinadas digitalmente), mas traz como principais inovações a combinação de elementos como a prova de trabalho e incentivos econômicos para alcançar um consenso descentralizado sobre a ordem das transações e a integridade do registro das transações.



Fonte: Nakamoto

Resumo das abordagens

Podemos concluir que os três artigos tratam do problema, pensam em como gerar consenso em um contexto distribuído, porém as abordagens são diferentes.

O algoritmo proposto por Lamport é baseado em mensagens assinadas digitalmente e assume que uma maioria dos generais é honesta e confiável. Já o sistema de Nakamoto é vai além dessa assinatura, baseado-se em criptografia, prova de trabalho e incentivos econômicos para garantir a integridade do registro das transações o autor conclui seu artigo com "We have proposed a system for electronic transactions without relying on trust". Dados esses novos elementos a rede blockchain pode ser aplicada a uma gama de aplicações de consenso distribuído, como registros eletrônicos de propriedade, votação eletrônica, controle de tráfego em cidades inteligentes, IoT e muito mais.

Bibliografia complementar

Lamport, L.; Shostak, R.; Pease, M. (1982). "The Byzantine Generals Problem" (PDF). ACM Transactions on Programming Languages and Systems. 4 (3): 387–389. CiteSeerX 10.1.1.64.2312.

Castro, M.; Liskov, B. (2002). "Practical Byzantine Fault Tolerance and Proactive Recovery". ACM Transactions on Computer Systems. Association for Computing Machinery. 20 (4): 398–461. CiteSeerX 10.1.1.127.6130. doi:10.1145/571637.571640. S2CID 18793794.

Martins, Rolando; Gandhi, Rajeev; Narasimhan, Priya; Pertet, Soila; Casimiro, António; Kreutz, Diego; Veríssimo, Paulo (2013). "Experiences with Fault-Injection in a Byzantine Fault-Tolerant Protocol". Middleware 2013. Lecture Notes in Computer Science. Vol. 8275. pp. 41–61. doi:10.1007/978-3-642-45065-5_3. ISBN 978-3-642-45064-8. ISSN 0302-9743. S2CID 31337539.

Outras soluções para o problema dos Generais Bizantinos (Página Wikipedia)

Em 1999, Miguel Castro e Barbara Liskov introduziram o algoritmo "Practical Byzantine Fault Tolerance" (PBFT), que fornece replicação de máquina de estado bizantina de alto desempenho, processando milhares de solicitações por segundo com aumentos de latência abaixo de milissegundos.

Após o PBFT, vários protocolos BFT foram introduzidos para melhorar sua robustez e desempenho. Por exemplo, Q/U, HQ, Zyzzyva, e ABSTRACTs, ([28][29][30][31], respectivamente) abordaram as questões de desempenho e custo; enquanto outros protocolos, como Aardvark[32] e RBFT,[33] abordaram seus problemas de robustez.

Além disso, o Adapt[34] tentou fazer uso dos protocolos BFT existentes, alternando entre eles de maneira adaptativa, para melhorar a robustez e o desempenho do sistema à medida que as condições subjacentes mudam.

Além disso, foram introduzidos protocolos BFT que aproveitam componentes confiáveis para reduzir o número de réplicas, por exemplo, A2M-PBFT-EA[35] e MinBFT.[36]

1. Abd-El-Malek, M.; Ganger, G.; Goodson, G.; Reiter, M.; Wylie, J. (2005). "Fault-scalable Byzantine Fault-Tolerant Services". ACM SIGOPS Operating Systems Review. Association for Computing Machinery. 39 (5): 59. doi:10.1145/1095809.1095817.

2. Cowling, James; Myers, Daniel; Liskov, Barbara; Rodrigues, Rodrigo; Shrira, Liuba (2006). HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation. pp. 177–190. ISBN 1-931971-47-1.

3. Kotla, Ramakrishna; Alvisi, Lorenzo; Dahlin, Mike; Clement, Allen; Wong, Edmund (December 2009). "Zyzzyva: Speculative Byzantine Fault Tolerance". ACM Transactions on Computer Systems. Association for Computing Machinery. 27 (4): 1–39. doi:10.1145/1658357.1658358.

4. Guerraoui, Rachid; Kneževic, Nikola; Vukolic, Marko; Quéma, Vivien (2010). The Next 700 BFT Protocols. Proceedings of the 5th European conference on Computer systems. EuroSys. Archived from the original on 2011-10-02. Retrieved 2011-10-04.

5. Clement, A.; Wong, E.; Alvisi, L.; Dahlin, M.; Marchetti, M. (April 22–24, 2009). Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults (PDF). Symposium on Networked Systems Design and Implementation. USENIX. Archived (PDF) from the original on 2010-12-25. Retrieved 2010-02-17.

6. Aublin, P.-L.; Ben Mokhtar, S.; Quéma, V. (July 8–11, 2013). RBFT: Redundant Byzantine Fault Tolerance. 33rd IEEE International Conference on Distributed Computing Systems. International Conference on Distributed Computing Systems. Archived from the original on August 5, 2013.

7. Bahsoun, J. P.; Guerraoui, R.; Shoker, A. (2015-05-01). "Making BFT Protocols Really Adaptive". Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International: 904–913. doi:10.1109/IPDPS.2015.21. ISBN 978-1-4799-8649-1. S2CID 16310807.

8. Chun, Byung-Gon; Maniatis, Petros; Shenker, Scott; Kubiatowicz, John (2007-01-01). "Attested Append-only Memory: Making Adversaries Stick to Their Word". Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles. SOSP '07. New York, NY, USA: ACM: 189–204. doi:10.1145/1294261.1294280. ISBN 9781595935915. S2CID 6685352.

9. Veronese, G. S.; Correia, M.; Bessani, A. N.; Lung, L. C.; Verissimo, P. (2013-01-01). "Efficient Byzantine Fault-Tolerance". IEEE Transactions on Computers. 62 (1): 16–30. CiteSeerX 10.1.1.408.9972. doi:10.1109/TC.2011.221. ISSN 0018-9340. S2CID 8157723.