MCTA025-13 - SISTEMAS DISTRIBUÍDOS

RELÓGIOS VETORIAIS, EXCLUSÃO MÚTUA E ELEIÇÃO

Emilio Francesquini

06 de agosto de 2018

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

- Estes slides foram preparados para o curso de **Sistemas Distribuídos na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Estes slides foram adaptados daqueles originalmente preparados (e gentilmente cedidos) pelo professor **Daniel Cordeiro, da EACH-USP** que por sua vez foram baseados naqueles disponibilizados online pelos autores do livro “Distributed Systems”, 3ª Edição em: `https://www.distributed-systems.net`.

A relação “aconteceu-antes” (*happened-before*)

- se a e b são dois eventos de um mesmo processo e a ocorreu antes de b, então $a \rightarrow b$
- se a for o evento de envio de uma mensagem e b for o evento de recebimento desta mesma mensagem, então $a \rightarrow b$
- se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$

Nota:
Isso introduz uma noção de ordem parcial dos eventos em um sistema com processos executando concorrentemente.

Problema
Como fazemos para manter uma visão global do comportamento do sistema que seja consistente com a relação aconteceu-antes?

Solução
Associar um *timestamp* $C(e)$ a cada evento e tal que:

- **P1** se a e b são dois eventos no mesmo processo e $a \rightarrow b$, então é obrigatório que $C(a) < C(b)$
- **P2** se a corresponder ao envio de uma mensagem m e b ao recebimento desta mensagem, então também é válido que $C(a) < C(b)$

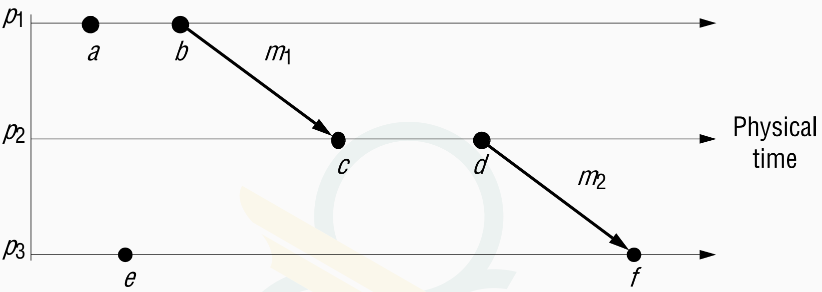
Solução

Cada processo P_i mantém um contador C_i local e o ajusta de acordo com as seguintes regras:

1. para quaisquer dois **eventos sucessivos** que ocorrer em P_i, C_i é incrementado em 1
2. toda vez que uma mensagem m for enviada por um processo P_i, a mensagem deve receber um *timestamp* $ts(m) = C_i$
3. sempre que uma mensagem m for recebida por um processo P_j, P_j ajustará seu contador local C_j para $\max\{C_j, ts(m)\}$ e executará o passo 1 antes de repassar m para a aplicação

Observações:

- a propriedade **P1** é satisfeita por (1); propriedade **P2** por (2) e (3)
- ainda assim pode acontecer de dois eventos ocorrerem ao mesmo tempo. Desempate usando os IDs dos processos.



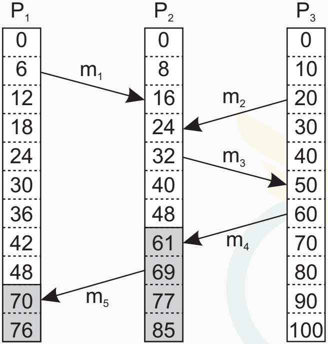
Fonte: CDKB

Exercício: O que se pode dizer sobre:

1. **a** e **b**?
2. **b** e **c**?
3. **a** e **f**?
4. **a** e **e**?

Relógios Vetoriais

Observação:
Relógios de Lamport **não** garantem que $C(a) < C(b)$ implica que a tenha realmente ocorrido antes de b:



Observação
Evento a: m_1 foi recebido em $T = 16$;
Evento b: m_2 foi enviado em $T = 20$.

Nota
Nós não podemos concluir que a precede temporalmente (precedência causal) b.

Definição
Dizemos que b pode depender causalmente de a se $ts(a) < ts(b)$ com:

- para todo k, $ts(a)[k] \leq ts(b)[k]$ e
- existe pelo menos um índice k' para o qual $ts(a)[k'] < ts(b)[k']$

Precedência vs. dependência

- Dizemos que a precede causalmente b
- b pode depender causalmente de a, já que há informação de a que pode ter sido propagada para b

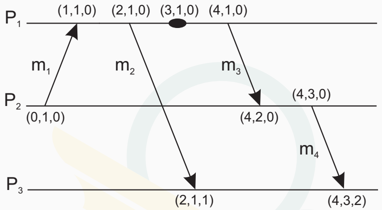
Relógios vetoriais foram criados para resolver as limitações de relógios de Lamport, *i.e.*, o fato de que eles **não garantem que se** $C(a) < C(b)$ **então** $a \to b$.

Solução: cada P_i mantém um vetor VC_i

- $VC_i[i]$ é o relógio lógico local do processador P_i
- se $VC_i[j] = k$, então P_i sabe que k eventos ocorreram em P_j.

Mantendo os relógios vetoriais

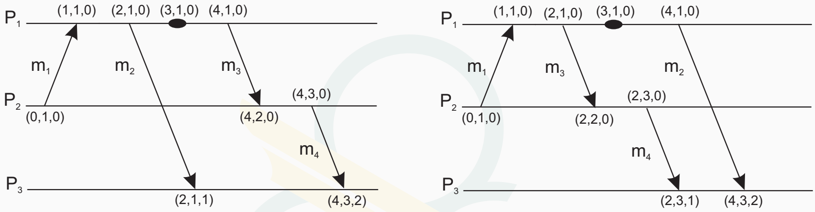
1. antes da execução de um evento, P_i executa $VC_i[i] \leftarrow VC_i[i] + 1$
2. quando o processo P_i enviar uma mensagem m para P_j, ele define o *timestamp* (vetorial) de m $ts(m)$ como sendo VC_i (após executar o passo 1)
3. no recebimento de uma mensagem m, o processo P_j define $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$



Análise

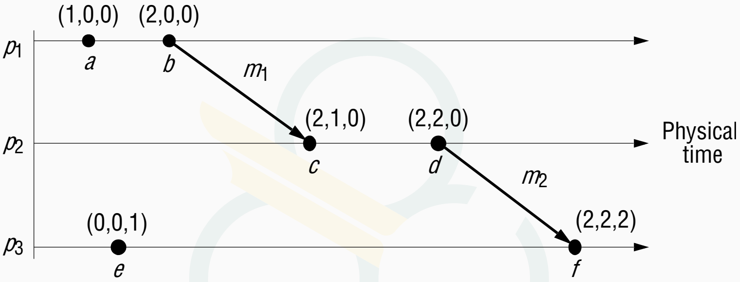
Situação	$ts(m_2)$	$ts(m_4)$	$ts(m_2) < ts(m_4)$	$ts(m_2) > ts(m_4)$	Conclusão
(a)	(2,1,0)	(4,3,0)	Sim	Não	m_2 pode preceder causalmente m_4, $m_2 \rightarrow m_4$

Suponha agora um atraso no envio de m_2:



Análise

Situação	$ts(m_2)$	$ts(m_4)$	$ts(m_2) < ts(m_4)$	$ts(m_2) > ts(m_4)$	Conclusão
(a)	(2,1,0)	(4,3,0)	Sim	Não	m_2 pode preceder causalmente m_4, $m_2 \rightarrow m_4$
(b)	(4,1,0)	(2,3,0)	Não	Não	m_2 e m_4 podem conflitar, $m_2 \parallel m_4$



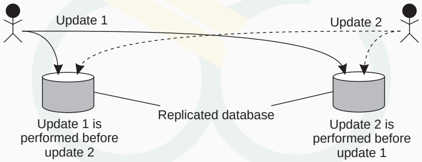
Fonte: CDKB

Exercício

1. O que pode ser dito sobre **a** e **f**?
2. O que pode ser dito sobre **c** e **e**?

Problema
Alguma vezes precisamos garantir que atualizações concorrentes em um banco de dados replicado sejam vistos por todos como se tivessem ocorrido na mesma ordem.

- P_1 adiciona R\$ 100 a uma conta (valor inicial: R\$ 1000)
- P_2 incrementa a conta em 1%
- Há duas réplicas



Resultado
Na ausência de sincronização correta,
réplica #1 $\leftarrow$ R\$ 1111, enquanto que na réplica #2 $\leftarrow$ R\$ 1110.

Observação

Agora é possível garantir que uma mensagem seja entregue somente **se todas as mensagens que as procederem por causalidade tiverem sido entregues**.

Multicasts ordenados por causalidade **são menos restritivos** do que multicasts *com ordem total*. Se duas mensagens não tem uma relação causal, então a ordem que elas serão entregues pode ser **diferente para cada um dos processos**.

Para garantir que as mensagens serão entregues seguindo a ordem causal:

Passos

1. P_i incrementa $VC_i[i]$ somente quando enviar uma mensagem;
2. P_j "ajusta" VC_j quando **entregar**[1] uma mensagem (mas não muda $VC_j[j]$): $VC_i[k] = \max\{VC_j[k], ts(m)[k]\}, \forall k$

[1] **Atenção**: as mensagens não são ajustadas quando são *recebidas*, mas sim quando elas são *entregues* à aplicação

Para garantir que as mensagens serão entregues seguindo a ordem causal:

Passos

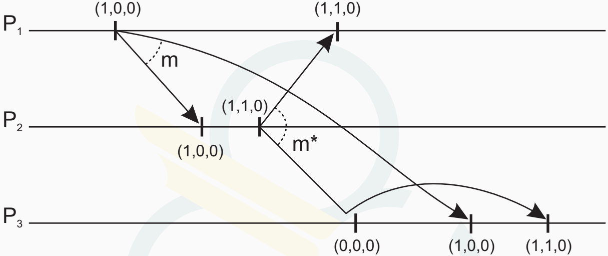
1. P_i incrementa $VC_i[i]$ somente quando enviar uma mensagem;
2. P_j "ajusta" VC_j quando **entregar**[1] uma mensagem (mas não muda $VC_j[j]$): $VC_i[k] = \max\{VC_j[k], ts(m)[k]\}, \forall k$

Além disto, P_j posterga a **entrega** de m até que:

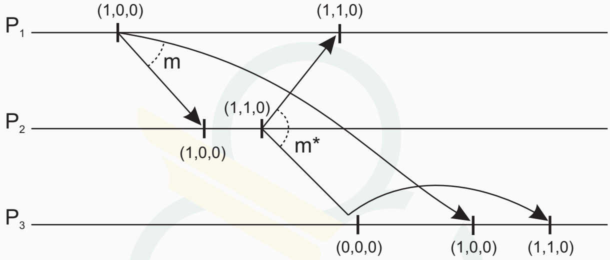
- $ts(m)[i] = VC_j[i] + 1$. (m é a próxima mensagem que P_j espera de P_i)
- $ts(m)[k] \leq VC_j[k]$ para $k \neq i$. (P_j já entregou todas as mensagens enviadas para P_i)

[1] **Atenção**: as mensagens não são ajustadas quando são *recebidas*, mas sim quando elas são *entregues* à aplicação

Exemplo



Exemplo



Exercício

Tome $VC_3 = [0, 2, 2]$, $ts(m) = [1, 3, 0]$ em P_1. Que informação P_3 tem e o que ele irá fazer quando receber m (de P_1)?

Algoritmos de Exclusão Mútua

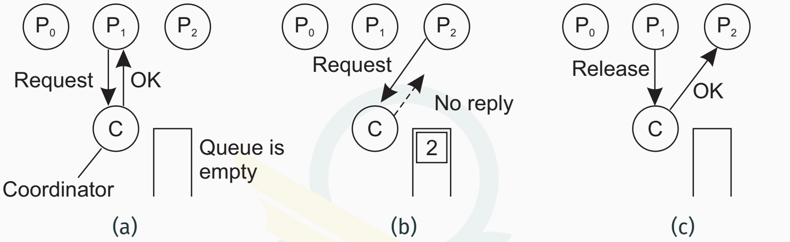
Problema
Alguns processos em um sistema distribuído querem acesso exclusivo a algum recurso.

Soluções:

Baseado em permissão: um processo que quiser entrar na seção crítica (ou acessar um recurso) precisa da permissão de outros processos

Baseado em tokens: um *token* é passado entre processos. Aquele que tiver o *token* pode entrar na seção crítica ou passá-lo para frente quando não estiver interessado.

Use um coordenador



(a) Processo P_1 pede permissão ao coordenador para acessar o recurso compartilhado. Permissão concedida.

(b) Processo P_2 então pede permissão para acessar o mesmo recurso. O coordenador não responde.

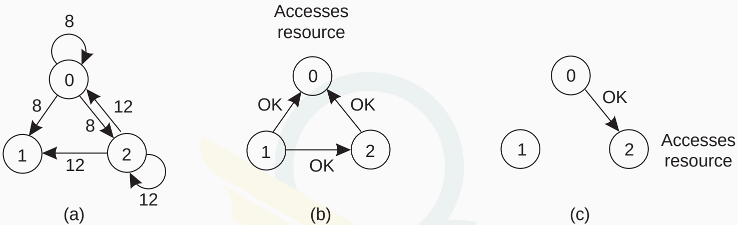
(c) Quando P_1 libera o recurso, avisa o coordenador, que então responde para P_2.

Princípio
Mesmo do Lamport, exceto que acks não são enviados. Ao invés disso, respostas (permissões) são enviadas quando:

- o processo receptor não tem interesse no recurso compartilhado; ou
- o processo receptor está esperando por um recurso, mas tem menos prioridade (a prioridade é determinada via comparação de timestamps)

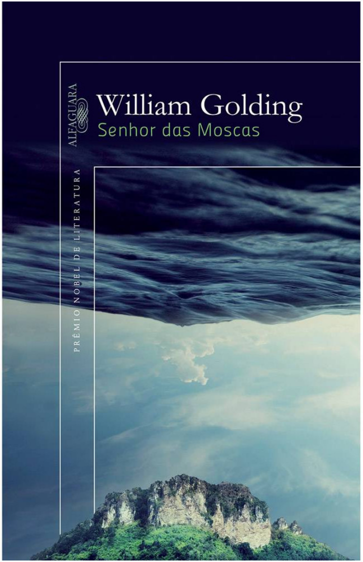
Em todos os outros casos, o envio da resposta é adiado, implicando a necessidade de alguma administração local.

Exemplo com três processos:



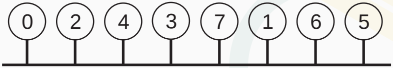
(a) dois processos (P_0 e P_2) querem acessar um recurso compartilhado ao mesmo tempo
(b) P_0 tem o menor *timestamp*; ele ganha
(c) quando P_0 terminar, também manda um OK; assim P_2 agora pode continuar

EXCLUSÃO MÚTUA BASEADA EM TOKEN

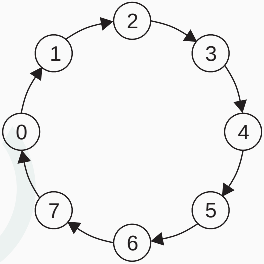


Fonte: Google Images

Ideia
Organizar os processos em anel lógico e passar um *token* entre eles. Aquele que estiver com o *token* pode entrar na seção crítica (se ele quiser).



(a)



(b)

Princípio
Assuma que todo recurso é replicado N vezes, com cada réplica associada a seu próprio coordenador $\Rightarrow$ acesso requer a maioria dos votos de $m > N/2$ coordenadores. Um coordenador sempre responde imediatamente a uma requisição.

Hipótese
Quando um coordenador morrer, ele se recuperará rapidamente, mas terá esquecido tudo sobre as permissões que ele deu.

Quão robusto é esse sistema?

- Seja $p = \Delta t / T$ a probabilidade de que um coordenador morra e se recupere em um período Δt e que tenha uma esperança de vida T.
- A probabilidade $\mathbb{P}[k]$ de que k dos m coordenadores sejam resetados durante o mesmo intervalo é:

$$\mathbb{P}[k] = \binom{m}{k} p^k (1-p)^{m-k}$$

- f coordenadores resetam $\Rightarrow$ corretude é violada quando os coordenadores que não falharam são minoria: quando $m - f \leq N/2$ ou $f \geq m - N/2$
- A probabilidade de violação é $\sum_{m-N/2}^{N} \mathbb{P}[k]$.

Probabilidade de violação em função dos parâmetros

N	m	p	Violação
8	5	3 seg/hora	$< 10^{-15}$
8	6	3 seg/hora	$< 10^{-18}$
16	9	3 seg/hora	$< 10^{-27}$
16	12	3 seg/hora	$< 10^{-36}$
32	17	3 seg/hora	$< 10^{-52}$
32	24	3 seg/hora	$< 10^{-73}$

N	m	p	Violação
8	5	30 seg/hora	$< 10^{-10}$
8	6	30 seg/hora	$< 10^{-11}$
16	9	30 seg/hora	$< 10^{-18}$
16	12	30 seg/hora	$< 10^{-24}$
32	17	30 seg/hora	$< 10^{-35}$
32	24	30 seg/hora	$< 10^{-49}$

EXCLUSÃO MÚTUA: COMPARAÇÃO

Algorítimo	# msgs por entrada/saída	Atraso para entrar (em qde msgs)	Problemas
Centralizado	3	2	Morte do coordenador
Decentralizado	$2mk + m$, $k = 1,2,...$	$2mk$	*Starvation*, ineficiente.
Distribuído	$2 (n - 1)$	$2 (n - 1)$	Morte de qualquer
Token ring	1 à ∞	0 à $n - 1$	Perder token, proc. morrer

Algoritmos de eleição

Princípio
Um algoritmo precisa que algum dos processos assuma o papel de coordenador. A pergunta é: como selecionar esse processo especial dinamicamente?

Nota
Em muitos sistemas o coordenador é escolhido manualmente (ex: servidores de arquivos). Isso leva a soluções centralizadas com um ponto único de falha.

Perguntas

1. Se um coordenador é escolhido dinamicamente, até que ponto podemos dizer que o sistema será centralizado e não distribuído?
2. Um sistema inteiramente distribuído (ou seja, um sem um coordenador) é sempre mais robusto que uma solução centralizada/coordenada?

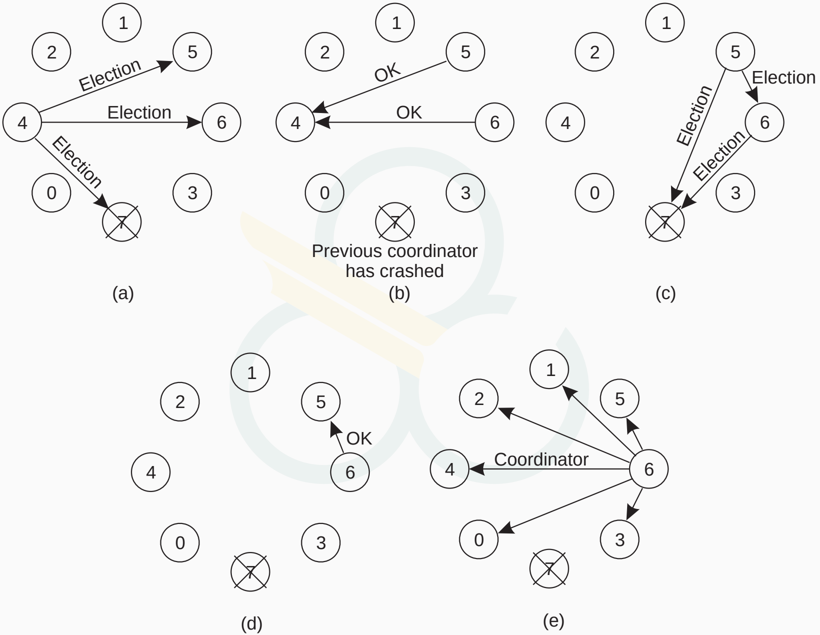
- Todos os processos possuem um `id` único
- Todos os processos conhecem os `id`s de todos os outros processos no sistema (mas eles não tem como saber se os nós estão funcionando ou não)
- A eleição significa identificar o processo de maior `id` que está funcionando em um dado momento

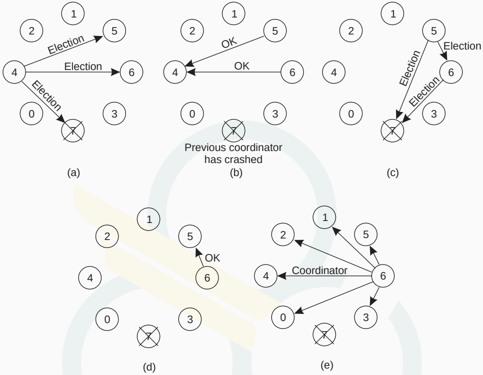
Princípio
Considere N processos $\{P_0, \ldots, P_{N-1}\}$ e seja $id(P_k) = k$. Quando um processo P_k perceber que o coordenador não está mais respondendo às requisições, ele começa uma nova eleição:

1. P_k envia uma mensagem `ELECTION` para todos os processos com identificadores maiores que o seu: $P_{k+1}, P_{k+2}, \ldots, P_{N-1}$.
2. Se ninguém responder, P_k ganha a eleição e se torna o coordenador
3. Se um dos nós com maior id responder, esse assume[2] a eleição e o trabalho de P_k termina.

[2] O maior sempre ganha, por isso o nome de “algoritmo do valentão”. 😜

ALGORITMO DE ELEIÇÃO — “BULLY”

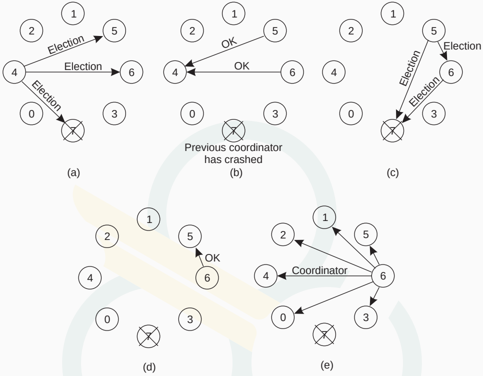




Cuidado

Estamos assumido algo importante aqui. O quê?

ALGORITMO DE ELEIÇÃO — “BULLY”



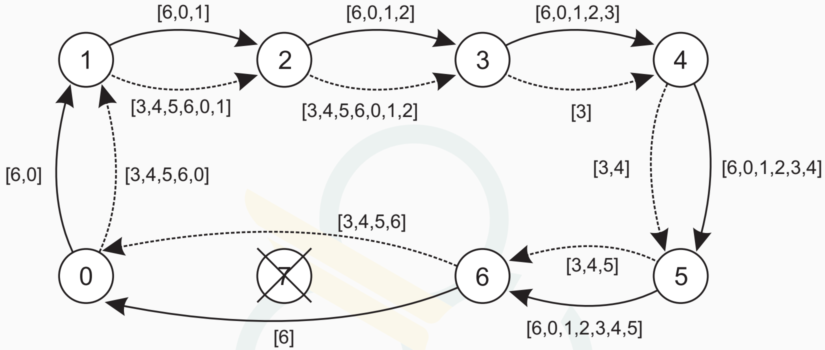
Cuidado

Estamos assumido algo importante aqui. O quê?

Assumimos que a comunicação é confiável

Princípio
As prioridades dos processos são obtidas organizando-os em um anel (lógico). Processos com prioridade mais alta devem ser eleitos como coordenador.

- qualquer processo pode iniciar a eleição ao enviar uma mensagem de eleição ao seu sucessor. Se um sucessor estiver indisponível, a mensagem é enviada ao próximo sucessor
- se uma mensagem for repassada, o remetente se adiciona na lista. Quando a mensagem voltar ao nó que iniciou, todos tiveram a chance de anunciar a sua presença
- o nó que iniciou circula uma mensagem pelo anel com a lista de nós “vivos”. O processo com maior prioridade é eleito coordenador



- As linhas contíguas mostram as mensagens da eleição iniciada por P_6
- As linhas pontilhadas se referem a eleição iniciada por P_3

ELEIÇÃO DE UM SUPERPEER

Como escolher um nó para ser um superpeer de forma que:

- nós normais acessem o superpeer com pouca latência
- superpeers sejam distribuídos homogeneamente por toda a rede de *overlay*
- seja mantida uma fração pré-definida de superpeers em relação ao número total de nós
- cada superpeer não deve ter que servir a mais de um número fixo de nós normais

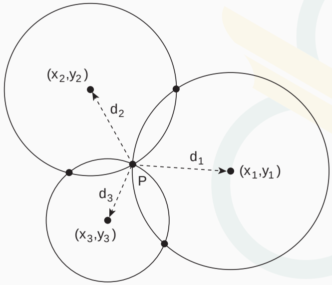
DHTs
Reserve uma parte do espaço de IDs para os superpeers. **Exemplo:** se S superpeers são necessários em um sistema que usa identificadores de m-bits, reserve os $k = \lceil \log_2 S \rceil$ bits mais à esquerda para os superpeers. Em um sistema com N nós, teremos, em média, $2^{k-m}N$ superpeers.

Roteamento para superpeers
Envie uma mensagem para a chave p para o nó responsável por p AND $\underbrace{11\cdots11}_{k}\underbrace{00\cdots00}_{m-k}$.

Sistemas de localização

Problema:
Em um sistema distribuído de grande escala onde os nós estão dispersados ao longo de uma rede de área ampla (*wide-area network*), frequentemente precisamos levar em consideração as noções de proximidade ou distância. Para isso, precisamos determinar a localização (relativa) de um nó.

Observação
Um nó P precisa de $d+1$ pontos de referência para calcular sua posição em um espaço d-dimensional. Considere o caso bidimensional:



Solução:
P precisa resolver um sistema de três equações com duas incógnitas (x_P, y_P):

$$d_i = \sqrt{(x_i - x_P)^2 + (y_i - y_P)^2}$$

Problema
Mesmo assumindo que os relógios dos satélites são precisos e estão sincronizados:

- leva algum tempo até que o sinal chegue ao receptor
- o relógio do receptor pode estar totalmente descompassado em relação ao satélite

- Δ_r: defasagem desconhecida do relógio do receptor
- x_r, y_r, z_r: coordenadas desconhecidas do receptor
- T_i: timestamp da mensagem do satélite i
- $\Delta_i = (T_{agora} - T_i) + \Delta_r$: *atraso medido* da mensagem enviada pelo satélite i.
- *distância **medida*** do satélite i: $c \times \Delta_i$ (c é a velocidade da luz)
- A distância **real** é:

$$d_i = (T_{agora} - T_i) \times c$$

logo:

$$d_i = c\Delta_i - c\Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

Observação
4 satélites $\Rightarrow$ 4 equações com 4 incógnitas (Δ_r sendo uma delas)

Ideia básica

- Assuma a existência de um banco de dados com as coordenadas de *access points* (APs) conhecidos
- Assuma que podemos estimar a distância até um AP
- Então: com três APs detectados, podemos calcular uma posição

Wardriving: localizando os pontos de acesso

- Use um dispositivo WiFi com um receptor GPS e se mova ao longo de uma área enquanto grava os pontos de acesso
- Calcule o centroide: assuma que um ponto de acesso AP foi detectado em N locais diferentes $\{\vec{x_1}, \vec{x_2}, \ldots, \vec{x_N}\}$ (cujas coordenadas foram capturadas com o GPS)
- Calcule a localização do AP como sendo $\vec{X_{AP}} = \frac{\sum_{i=1}^{N} \vec{x_i}}{N}$.

Problemas:

- acurácia de cada ponto $\vec{x_i}$ detectado pelo GPS
- um *access point* tem uma faixa de transmissão que não é uniforme
- o número de pontos da amostra (N) pode ser muito pequeno