Replicação Fracamente Coerente

Replicação Fracamente Coerente

- ~~Coerência forte~~
  - ~~Leituras observam sempre versão mais recente~~
  - ~~*Coerência sequencial, linearizabilidade, etc.*~~
- Alta disponibilidade
  - Qualquer acesso recebe uma resposta que não é “erro”
- Tolerância a partições
  - Sistema funciona mesmo na presença de partições de rede
  - Ou seja, apesar de um número arbitrariamente alto de mensagens se perderem ou atrasarem

Tentemos construir um protocolo de replicação fracamente coerente...

A + P

Modelo do sistema

- Múltiplas réplicas, múltiplos clientes
- Cada réplica tem um **id numérico**=0,1,2,3,...
- Sistema **assíncrono**, onde podem ocorrer partições de rede
- Rede assegura **entrega FIFO**

Operações sobre o sistema replicado

- *Queries (*leituras)

- *Updates*
  - Operações que modificam estado replicado
  - Podem ser operações **mais complexas** que simples *write*
    - Por exemplo: insert(element, map)

Esboço inicial de um protocolo: Interação cliente - réplica

- Cliente envia pedido de operação **a uma réplica *r***
  - Por exemplo, a mais "próxima"
  - Mesmo cliente pode contactar diferentes réplicas ao longo do tempo

- Réplica *r* que recebe o pedido do cliente:
  - Caso seja *update*, atribui-lhe um **identificador $u_{rep,seq}$**
    - *rep*: identificador da réplica
    - *seq*: número de sequência, incrementado para cada *update* emitido pela réplica
  - Executa o pedido localmente
    - Caso seja *update*, junta-o a um *update log* local
  - Retorna ao cliente
  - **Em *background*, propaga** o pedido às restantes réplicas

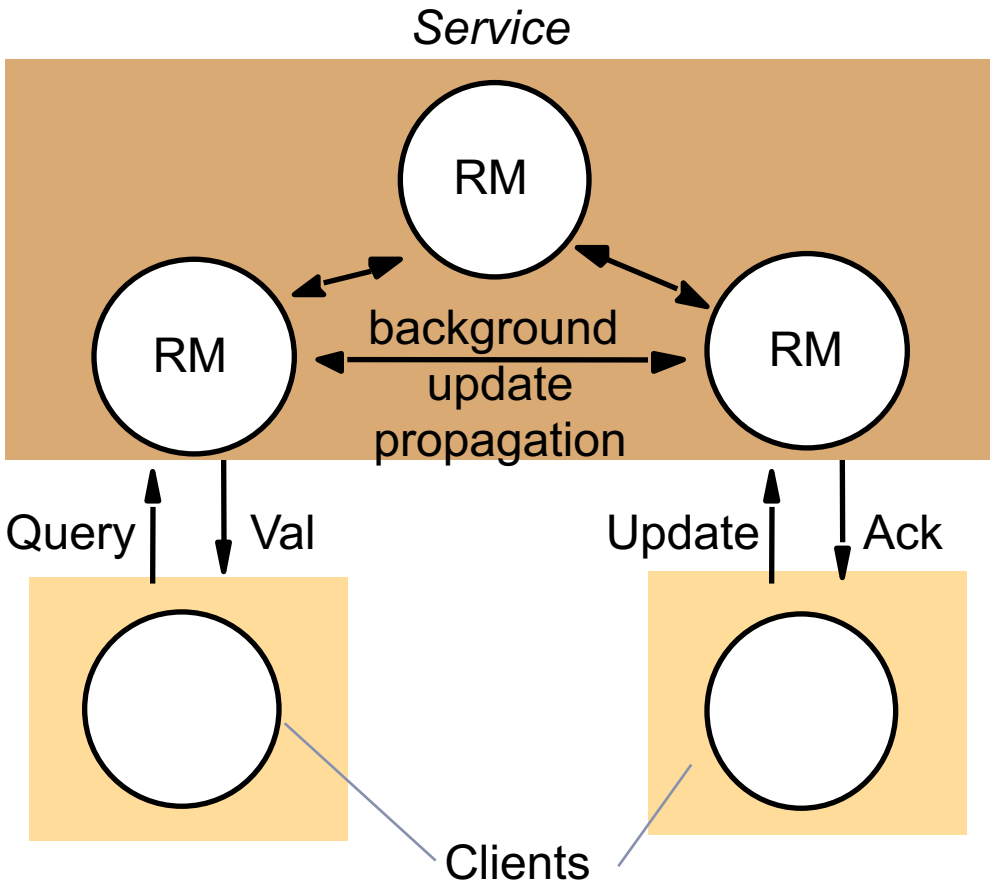
Esboço inicial de um protocolo: Propagação de *updates*

- Feita em fundo, **assincronamente**

- Réplica A liga-se a réplica B
- A envia a B sequência de *updates*
  - Na ordem pela qual A os mantém no seu *log* local

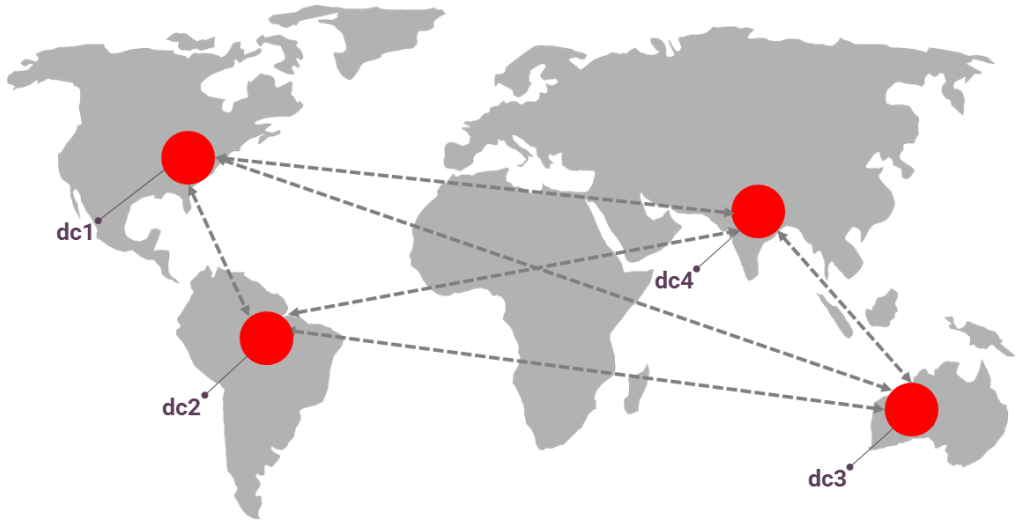
- Pode ser usado algum mecanismo de filtragem para evitar que A envie *updates* que B já conhece
  - Vamos omitir esse detalhe

Esboço inicial do protocolo: Visão geral



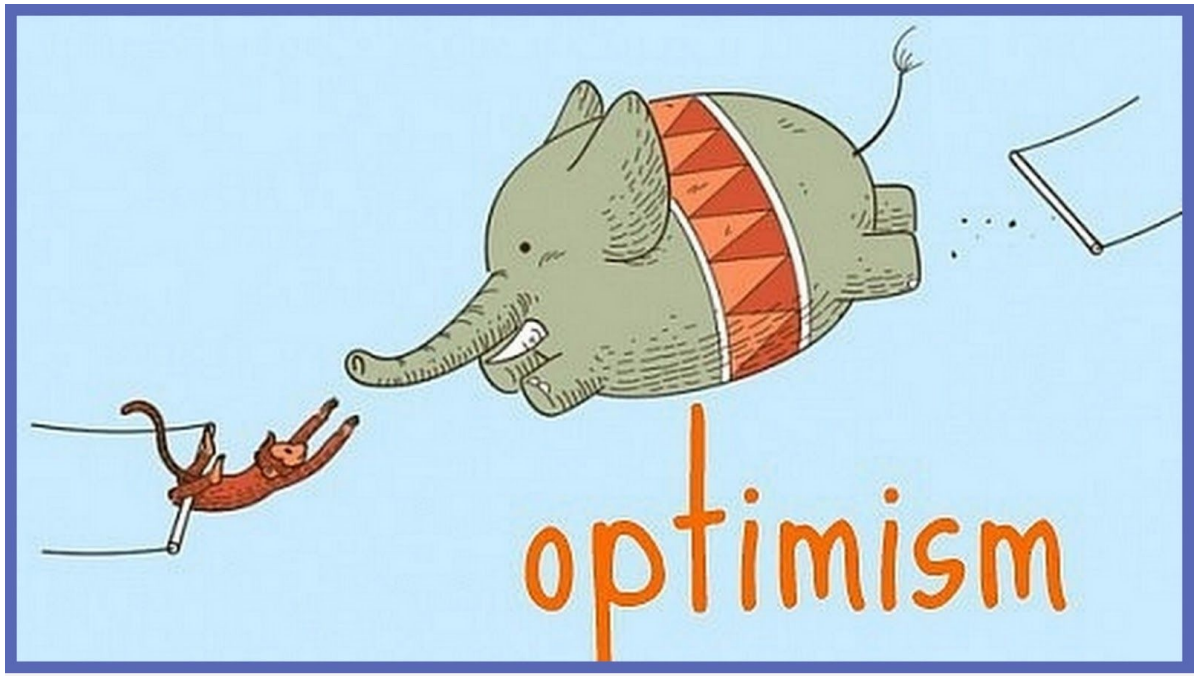
Que vantagens conseguimos?

- Alta **disponibilidade**, mesmo na presença de **partições de rede**
- Acessos podem ser respondidos imediatamente pela réplica **mais próxima** do cliente
  - Coordenação com o resto das réplicas feita em ***background***
- Vantagens muito importantes em **sistemas geo-replicados**



Desvantagens: coerência fraca

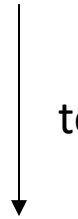
- Réplicas podem ter **vistas desatualizadas ou divergentes**
  - Réplica **não espera por coordenação** global antes de responder ao cliente
  - As réplicas **não** executam necessariamente as operações na mesma **ordem**



Desvantagens: coerência fraca (II)

- Temporariamente, é possível a uma **réplica não ter os *updates*** que foram recentemente emitidos por outras réplicas
  - Esta situação é resolvida quando as réplicas propagarem os *updates* entre si
- É possível uma réplica **ordenar** certos *updates* de forma **diferente** de outra réplica
  - Exemplo:

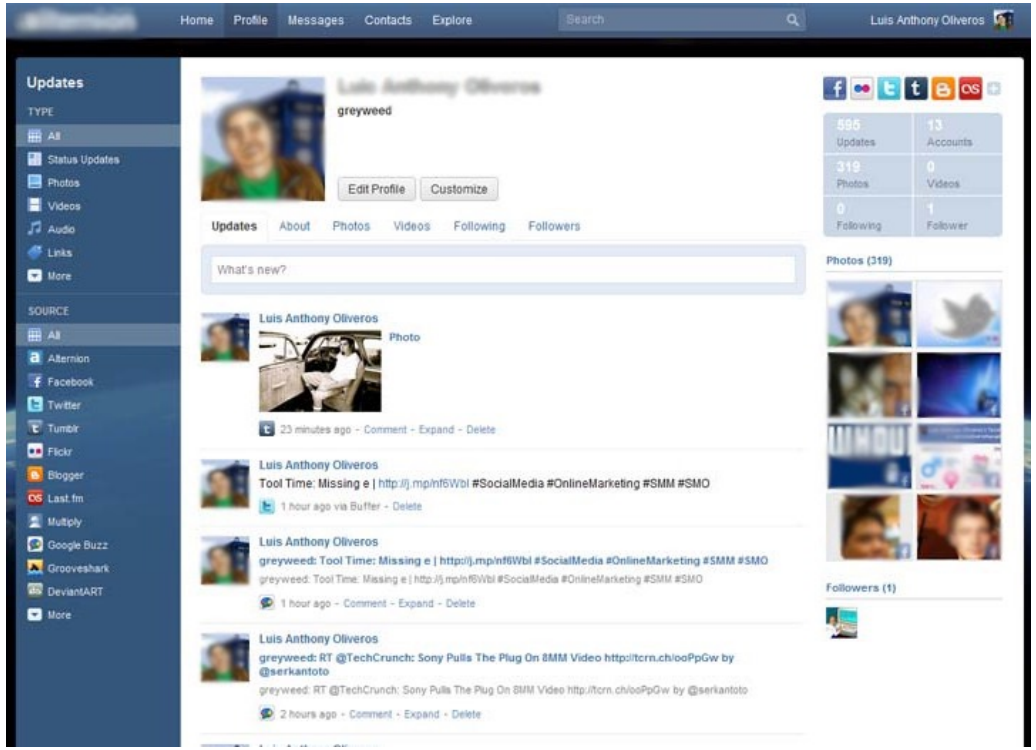
Réplica 0	Réplica 1
$u_{0,1}$	$u_{1,1}$ $u_{1,2}$
$u_{0,1}$ $u_{1,1}$ $u_{1,2}$	$u_{1,1}$ $u_{1,2}$ $u_{0,1}$



Alguma aplicação aceitaria estas anomalias de incoerência?

- Há aplicações em que:
  - Acesso a dados ligeiramente desatualizados não é erro sério
  - A ordem de execução dos *updates* não é importante para a aplicação
    - Ou seja, o estado obtido executando o mesmo conjunto de *updates* por ordens diferentes é **igual** ou **equivalente**

Exemplo de aplicação: *feed* de rede social



- Publicações e comentários
- Operações de atualização:
  - Publicar
  - Comentar
- O estado da aplicação resulta da aplicação cumulativa de atualizações

Atualizações do *feed* social

- publicar(“Tiago”, “Boa notícia: …”)
- publicar(“João”, “Bom dia”)
- publicar(“Pedro”, “Grande jogo”)
  - comentar(“Paulo”, “Pois foi”)

- Resultado 1:
  - João: Bom dia
  - Pedro: Grande jogo
    - Paulo: Pois foi
  - Tiago: Boa notícia: …

Atualizações (por outra ordem)

- publicar(“Tiago”, “Boa notícia: …”)
- publicar(“João”, “Bom dia”)
- publicar(“Pedro”, “Grande jogo”)
  - comentar(“Paulo”, “Pois foi”)

- Resultado:
  - Tiago: Boa notícia:…
  - João: Bom dia
  - Pedro: Grande jogo
    - Paulo: Pois foi

A ordem das publicações pode mudar – não é essencial para a aplicação

Mas o comentário só faz sentido depois da publicação (relação causal)

Conseguimos melhorar o nosso protocolo para prevenir algumas anomalias?

Anomalia 1: **Leituras incoerentes** pelo mesmo cliente

Anomalia 1:
Leituras incoerentes pelo mesmo cliente

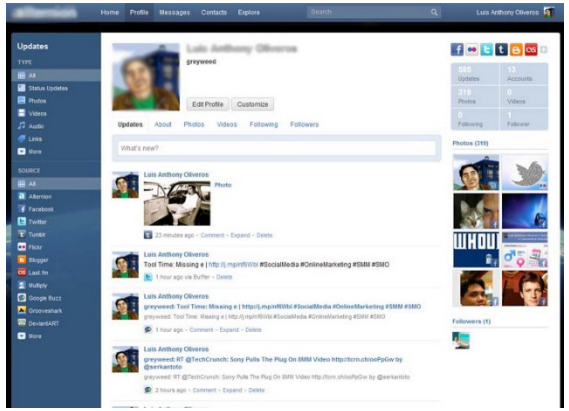
- Um cliente executa uma leitura e o valor observado reflete um *update u* (entre outros *updates*)

- Posteriormente, o mesmo cliente lê **de outra réplica** e obtém valor que não reflete o *update u*
  - *Possível porque as réplicas não têm de estar todas no mesmo estado*
  - *Algumas podem não estar atualizadas*

- Como **prevenir** esta anomalia?

No exemplo do *feed* social…

- A anomalia 1 é um mesmo cliente ver uma publicação que depois desaparece…



Anomalia 1:
Leituras incoerentes pelo mesmo cliente

- O que pretendemos:
  - Cada cliente **manter estado** sobre os *updates* que já observou em leituras anteriores
  - Ao invocar leitura sobre uma réplica, **verificar** se o conjunto de *updates* atual da réplica inclui o conjunto de *updates* já observados pelo cliente

- Como implementar isto de forma **eficiente**?

Timestamps vetoriais (*vector clocks*)

- Assumindo sistema em que:
  - Cada réplica tem um id=0,1,2,3,...
  - Cada *update* emitido por uma réplica tem um número sequencial

- ***Timestamp* vetorial** representa uma **versão**, resultante da execução cumulativa de um conjunto de *updates*
  - *Timestamp* vetorial tem uma entrada para cada réplica
  - Cada entrada denota o número sequencial do *update* mais recente emitido por essa réplica

Exemplo de *Timestamp* vetoriais

- Exemplo com três réplicas: R0, R1, R2
  - Réplica conhece estes *updates*: $u_{0,1}$ $u_{1,1}$ $u_{1,2}$ $u_{2,1}$ $u_{1,3}$

- *Timestamp* da versão obtida pela execução destes *updates*: <1,3,1>
  - Da réplica 0 recebemos todos os updates até ao 1
  - Da réplica 1 recebemos todos os updates até ao 3
  - Da réplica 2 recebemos todos os updates até ao 1

Relembrando as operações com *Timestamps* vetoriais

- Comparar versões
  - se v1[i] ≥ v2[i] para **todas** as entradas, então a versão de v1 é mais recente que v2

- Sincronizar réplicas:
  - Se réplica A propaga para B os *updates* que B possivelmente não tinha ainda, a versão de B passa a ser dada por **vB[i] = max(vA[i], vB[i])**, para todas as entradas

Anomalia 1:
Leituras incoerentes pelo mesmo cliente

- O que pretendemos:
  - Cada cliente manter **estado** sobre os *updates* que já observou em leituras anteriores
  - Ao invocar leitura sobre uma réplica, **verificar** se o conjunto de *updates* atual da réplica inclui o conjunto de *updates* já observados pelo cliente
- Como implementar isto de forma **eficiente**?
  - Cada réplica mantém *valueTS*, um ***timestamp* vetorial** que representa o conjunto de *updates* conhecidos pela réplica
  - *valueTS* é retornado ao *cliente* junto com cada valor lido
  - Cliente mantém *prevTS*, o *timestamp* vetorial da última leitura que leu
  - Em cada leitura, o cliente **verifica se *valueTS* ≥ *prevTS***

Anomalia 2: **Violação da causalidade** entre operações

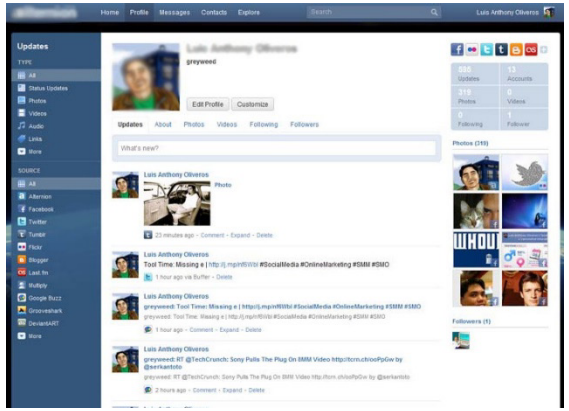
Anomalia 2:
Violação da causalidade entre operações

- Cliente *j* lê valor resultante do *update* anterior feito por cliente *i*
- **Em reação**, cliente *j* emite outro *update*
  - Ou seja, o *update* de *j* ***depende causalmente*** do *update* de *i*
- Uma réplica recebe apenas o *update* de j, ou recebe ambos na ordem inversa

- Como **prevenir** esta anomalia?

No exemplo do *feed* social…

- A anomalia 2 é aparecer um comentário sem a publicação correspondente…



Anomalia 2:
Violação da causalidade entre operações

- Cada *update* feito por um cliente leva um *timestamp prevTS*
  - **Indica a versão** da qual o *update* depende causalmente

- Ao receber o *update* + *prevTS*, réplica **verifica se *valueTS≥prevTS***
  - Se sim, executa o *update* (tal como definido antes)
  - Se não:
    - Coloca o *update* numa **fila** de *updates* pendentes
    - Sempre que novos *updates* forem executados localmente, verifica se já é possível verificar os *updates* pendentes
    - *Updates* pendentes também são propagados entre réplicas

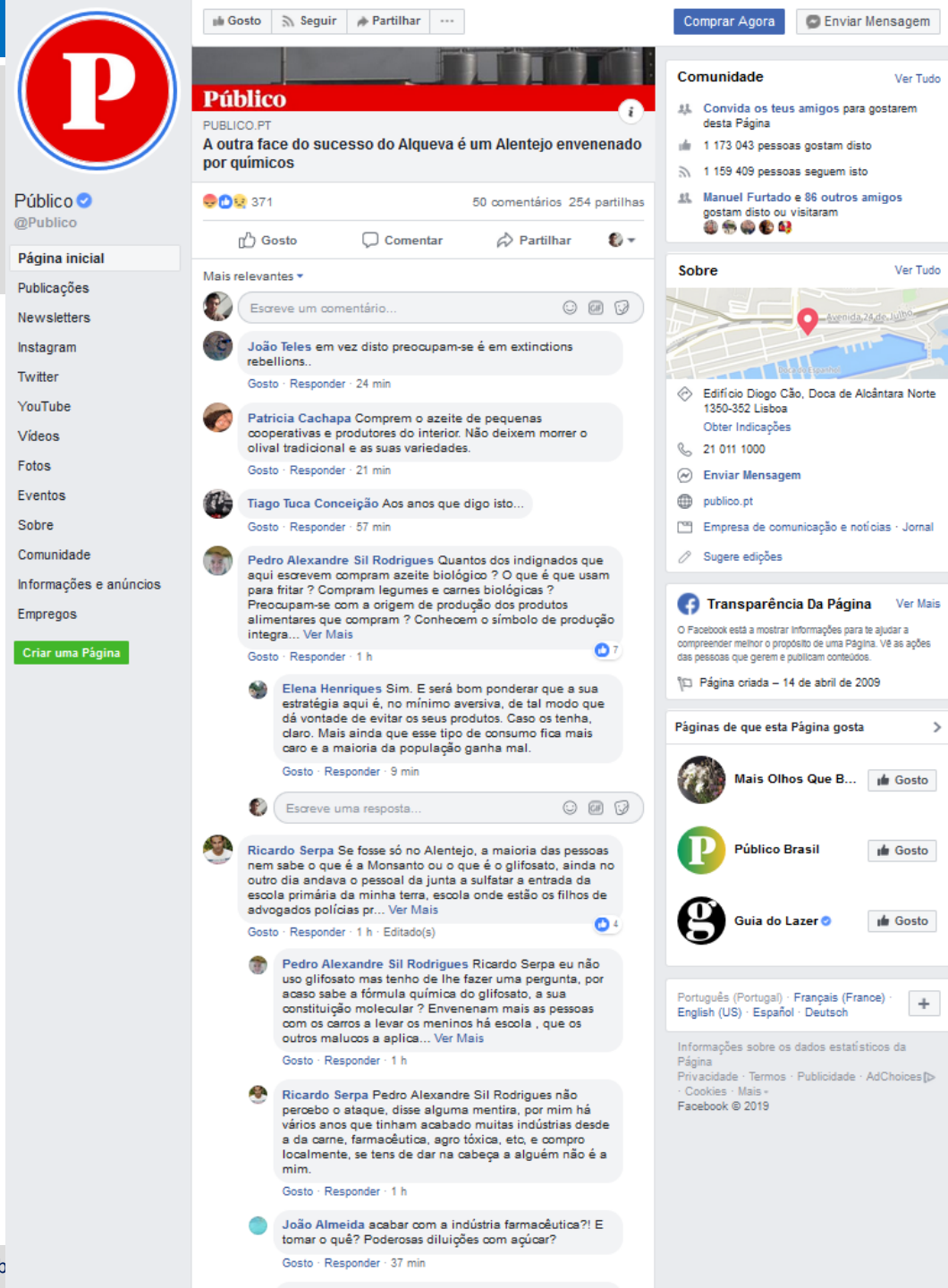
Conclusões

- Muitos protocolos de **replicação otimista** conseguem:
  - Prevenir muitas das anomalias de coerência que podem surgir com protocolos que oferecem garantias A+P
  - No nosso exemplo vimos como prevenir duas anomalias
  - No entanto, outras anomalias continuam a poder ocorrer (ao contrário de sistemas com coerência forte)

- Muitas aplicações **aceitam coerência fraca** para as estruturas de dados cuja coerência não é crítica

Aplicações geo-distribuídas que toleram coerência fraca

- Rede social online:
  - Publicações
  - Caixas de comentários
  - Conjunto de *Likes*
  - Notificações
  - etc.



Aplicações geo-distribuídas que toleram coerência fraca

- *Site* de compras:
  - Listas de *best sellers*
  - Carrinhos de compras
  - Preferências do cliente
  - Catálogo de produtos
  - etc.