Tópicos em Redes de Computadores

PBFT: Sistemas Tolerantes a Intrusão

Prof. Elias P. Duarte Jr.
Universidade Federal do Paraná (UFPR)
Departamento de Informática
www.inf.ufpr.br/elias/topredes

Sumário

- Hoje vamos estudar um dos resultados mais importantes de sistemas distribuídos
- PBFT: Practical Byzantine Fault Tolerance
- Vamos definir o problema, o modelo de sistema
- Estudar o protocolo que representa a solução
- O artigo original:

Miguel Castro, Barbara Liskov, “Practical Byzantine Fault Tolerance,” *The 3rd OSDI*, pp. 1–15, 1999.

Replicação Distribuída

- O PBFT foi proposto para replicação distribuída
- A idéia: um servidor é replicado e múltiplos processos respondem pelo serviço
- Clientes fazem múltiplas requisições simultâneas
- Objetivo: manter a consistência das réplicas, mesmo que haja falhas bizantinas
- Solução: garantir que todas as réplicas executam todas as requisições na mesma ordem

Máquina de Estados (ME)

- Uma ME representação (modelo) de um processo
- Consiste de:
  - Um conjunto de estados do processo
  - Um conjunto de transições entre estados
  - Uma transição do “estado atual” para o próximo estado ocorre como consequência de um “evento”
    - um evento pode ser, por exemplo, a execução de uma operação pelo processo (digo, máquina de estado)
    - transições de um estado para o próprio são permitidas

Máquina de Estado Determinística

- Uma máquina de estado determinística só permite a ocorrência de 1 evento de cada vez
- A transição de estados causada pelo evento também é única, produz uma saída bem definida
  - logicamente a saída produzida depende do estado atual e da transição que ocorreu

Em Sistemas Assíncronos

- A máquina de estados no modelo temporal assíncrono pode demorar um tempo arbitrário para realizar a transição entre estados
  - e para produzir a saída correspondente, se houver
- Considerando que processos (máquinas de estado) podem sofrer falhas
- O modelo assíncrono sofre daquele problema crucial: é impossível distinguir um processo falho de um processo lento

Replicação Máquina de Estados

- Aplicação típica: replicação de servidores
- Por exemplo: pense em um grupo de servidores Web replicados
- Na Replicação Máquina de Estados – *State Machine Replication:*

  → *cada réplica é uma máquina de estados*

  → *todas as réplicas iniciam no mesmo estado*

  → *executam a mesma sequência de operações*

  → *desta forma: todas as transições são idênticas!*

Réplicas Consistentes

- A Replicação Máquina de Estados garante que todas as réplicas estarão sempre no mesmo estado
- Desta forma: se uma réplica falha, sem problemas! O serviço continua disponível!
- Até **f** réplicas podem falhar e o serviço continua disponível
- Além disso: aumenta o desempenho do serviço: ao invés de ter um servidor respondendo clientes, tem n servidores

Operações vão chegando

- Clientes geram requisições de operações que devem ser executadas pelo servidor
- As requisições devem carregar: o *id* do cliente (lógico), um *timestamp* (vamos explicar direitinho) e a operação a ser executada
- Considere que o servidor é réplicado usando RME
- Note que isso fica totalmente transparente para os clientes!
- Múltiplos clientes podem enviar múltiplas requisições

Múltiplos Clientes & Múltiplas Requisições

- Múltiplos clientes podem enviar múltiplas requisições
- Não podemos deixar que sejam executadas pelas réplicas do servidor em ordens diferentes!
- Deve haver um **consenso** sobre exatamente qual requisição deve ser executada de cada vez
- É justamente isso que o PBFT garante: todas as réplicas executam exatamente a mesma operação, mesmo que haja um máximo de **f** falhas bizantinas

PBFT: Modelo de Sistema

- O PBFT foi definido para um sistema assíncrono
- Considera que canais de comunicação podem perder mensagens
  - canais de comunicação podem perder, atrasar, duplicar mensagens e entregá-las fora de ordem
  - as mensagens no grupo de réplicas são transmitidas com multicast, daí precisa usar UDP (e não TCP)
- Falhas são bizantinas: arbitrárias
  - inclui crash e omissão
  - na motivação menciona invasão de sistemas e bug de software

PBFT: Continuando o Modelo

- O modelo considera que as falhas são “independentes”
  - a falha de um processo não altera a probabilidade de outro processo falhar
  - para isso: implementações distintas: *N-version programming*
  - e devem ser também administrados independemente
  - na prática: réplicas do mesmo processo ;-)

PBFT: Ainda o Modelo

- O PBFT considera que as mensagens trocadas entre os processos são sigilosas, mantém integridade e autenticação
- Toda mensagem é assinada da forma tradicional
  - calcula o hash da mensagem e criptografa com a chave privada (critptografia assimétrica)
  - uma mensagem assinada pelo processo i é escrita assim: <m>σi
  - o hash da mensagem m é chamado de D(m), D de “message digest”, sinônimo de hash

O Serviço

- Clientes fazem requisições para o serviço replicado
- São n réplicas, sendo n = 3f+1
- Faz referência ao Teorema de Lamport (vimos na aula passada) para justificar o limite
- Mas tem mais! Explicação intuitiva muito legal

Por que 3f+1 réplicas?

- Como são f falhos e eles podem omitir mensagens, o serviço tem que terminar com 2f+1 mensagens
- Mas pode ser que os que não mandaram mensagens eram sim corretos, pois o sistema é assíncrono!
- Assim, das 2f+1 mensagens recebidas, f podem ter vindo de processos maliciosos
- Sem problemas: mesmo assim, a maioria é de mensagens de processo corretos (f+1)

Propriedades: Safety

- *Safety*: o sistema replicado funciona exatamente como um sistema não replicado, centralizado com 1 único servidor
  - assim, as réplicas executam 1 operação atômica por vez, sequencialmente
  - *linearizability*
  - mesmo que haja até f réplicas falhas, as réplicas corretas continuam executando de forma consistente
  - além disso: o sistema tem controle de acesso dos clientes

Propriedades: Liveness

- Liveness: os clientes recebem respostas para suas requisições após um intervalo de tempo finito
- Apesar de que não há premissa sobre sincronização de relógios:
  - seja atraso(t) o tempo para transmitir uma mensagem
  - assume-se que atraso(t) não cresce mais rapidamente que t indefinidamente

PBFT: O Algoritmo

- O PBFT é um algoritmo de replicação máquina de estados
- Há um conjunto de réplicas R, que têm identificadores 0, 1, ..., |R|-1
- |R| = 3f+1, uma discussão muito interessante de que não deve ser maior!
  - se for maior: traz mais impacto no desempenho, sem ganho em termos das propriedades concretas

As Réplicas: Primária e Backups

- Uma das réplicas funciona como primária
- Pode ser qualquer uma (é um líder, na verdade)
- As outras réplicas são chamadas no artigo de backups, mas podemos chamar também de secundárias

PBFT: Visões

- Um conceito muito importante no PBFT
- Uma visão é a forma como uma réplica enxerga a composição do sistema como um todo
  - na prática: o conjunto de réplicas corretas
  - por exemplo, se há 4 réplicas e a 2 está falha: {0, 1, 3}
- Os processos (réplicas) mantém permanentemente uma visão atualizada da composição do sistema
- As visões são numeradas, 1ª visão, 2ª visão, ... e assim por diante
- Atenção: **apenas** quando o **primário** falha, muda a visão!

PBFT: O Algoritmo 1,2,3,4

1) O cliente envia uma requisição para o primário
2) O primário faz difusão da requisição p/ backups
3) Réplicas executam e retornam resposta ao cliente
4) O cliente espera receber f+1 resultados idênticos para definir o resultado da requisição

Premissas

- Todas as réplicas iniciam no mesmo estado
- As réplicas devem ser determinísticas
  - a execução de uma operação por uma réplica em um determinado estado e com um determinado conjunto de argumentos de entrada → SEMPRE produz o mesmo resultado
- Para garantir a safety do PBFT é necessário garantir que TODAS as réplicas executam TODAS as operações na mesma ordem, mesmo havendo **f** falhas

O Cliente

- Um cliente c requisita a execução da operação o:

  <REQUEST, o, t, c>σ_c
- t – timestamp da requisição, identifica de forma única a requisição → basta ser implementado como um contador local
- Todas as mensagens enviadas pelas réplicas ao cliente incluem o número da visão atual
- Assim o cliente determina quem é o primário – pode alternativamente enviar a requisição para todos

Primário recebe requisição

- Após receber a requisição, o primário faz uma difusão atômica da requisição para todos os backups
- Após receber f+1 respostas corretas, a resposta enviada ao cliente é da seguinte forma:

  <REPLY, v, t, c, i, r>σi
- Sendo: v – visão, t – timestamp, c – cliente, i – réplica, r – resultado, σi – mensagem assinada por i

Estado de uma Réplica

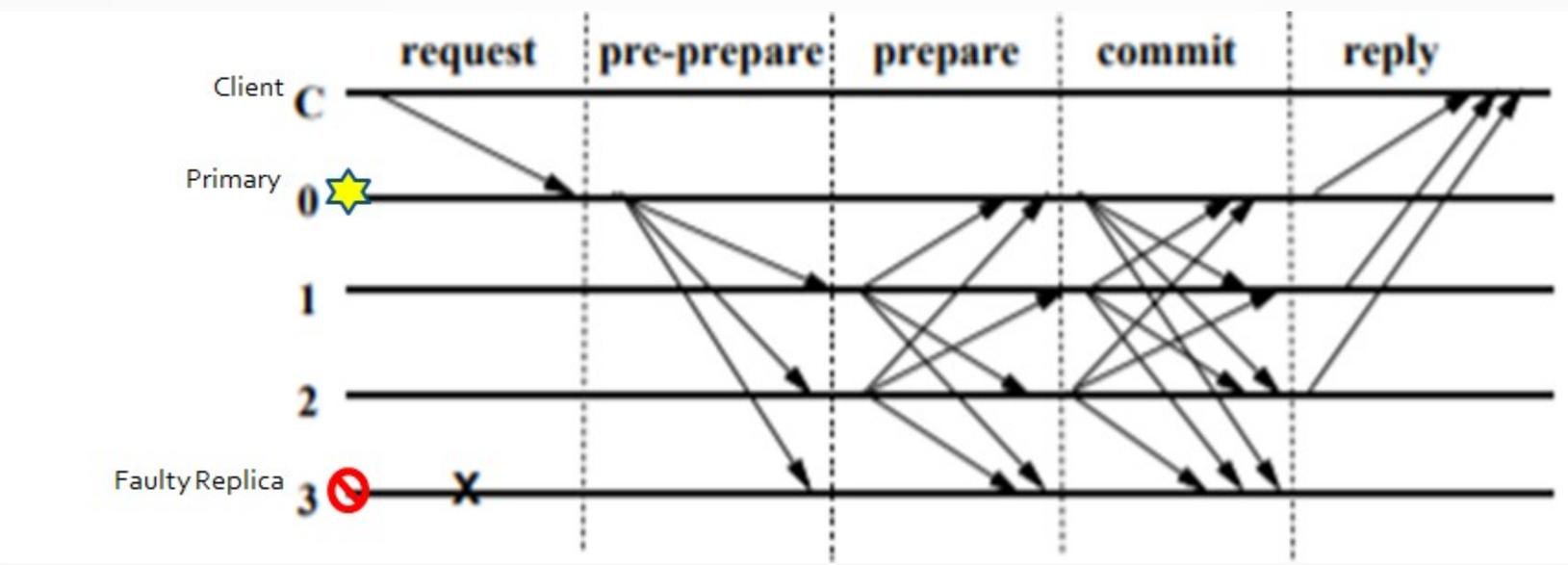
- Cada réplica mantém:
  - A visão atual, que é indentificada por um contador inteiro
  - Um **log** de mensagens que aceitou (requisições e outras tb)

A Comunicação Primário-Backups

- A comunicação do primário para as réplicas se dá através de difusão atômica (*atomic broacast*)
- Todas as mensagens são entregues por todas as réplicas na mesma ordem
- Logo após receber a requisição do cliente, o primário já comunica com os backups
- É possível otimizar o algoritmo de forma que ao invés de mandar 1 operação de cada vez, o primário manda várias

A Difusão Atômica

- O algoritmo definido no PBFT tem 3 fases:
  - pre-prepare
  - prepare
  - commit



Fase Pre-Prepare

- Tudo começa na fase pre-prepare, em que o primário **p** envia uma mensagem

  $<\text{PRE-PREPARE}, v, n, d>\sigma_p$

- v – visão, n – número de sequência desta requisição para o primário, d – hash da mensagem (digest), mensagem assinada por p
- a mensagem m pode ser mandada junto a esta mensagem ou separada, o hash(m) = d permite conferir se é a mensagem correta

Fase Pre-Prepare no Backup

- Um backup aceita o pre-prepare se:

1) Assinaturas e hash estão corretos

2) O backup está na mesma visão v indicada

3) O backup não aceitou outro pre-prepare com a mesma visão, o mesmo número de sequência e um hash diferente (portanto m diferente)

4) O número de sequência está dentro de limites especificados (só para evitar números absurdos)

Do Pre-Prepare → Prepare

- Quando aceita o pre-prepare, o backup **i** entra na fase prepare
- Envia a seguinte mensagem para todas as outras réplicas (inclusive a primária):

  $\langle PREPARE, v, n, d, i \rangle \sigma_i$

- Esta mensagem é acrescentada ao log de i

Uma Réplica Aceita o Prepare

- Uma réplica (inclusive a primária) aceita o backup se:

1) A assinatura está correta
2) A réplica está na mesma visão indicada no prepare
3) O número de sequência da requisição (n) está na faixa especificada (só para evitar coisas malucas)

Tá Preparada

- Dizemos que uma réplica **i** está

  prepared(m, v, n, i)
- se tem armazenado no seu log:
  - a requisição m, na visão v, com num-seq n
  - **mais** 2f prepares com os mesmos dados vindos de 2f réplicas diferentes (a primária está também → 2f+1)
- As fases pre-prepare e prepare garantem que todas as réplicas entraram em acordo para o número de uma requisição dentro de uma visão

Se está preparada…

- Se a réplica i está prepared(m, v, n, i)
- Então nenhuma réplica j pode estar prepared(m’,v, n, j)
  - não vou ter 2 requisições com o mesmo número de sequência na mesma visão
  - na verdade basta que os hashes sejam diferentes, não as mensagens

Prepare → Commit

- Quando fica prepared(m,v,n,i) a réplica i difunde <COMMIT, v, n, D(m),i>σ_i
- Todas as réplicas recebem, inclusive primária
- Uma réplica aceita a mensagem de commit se:

1) A assinatura está correta
2) A réplica está na mesma visão indicada na mensagem
3) O num-seq está na faixa prevista

Tá Comitada :-P

- Dizemos que uma réplica está **committed(m,v,n)** se um conjunto qualquer de f+1 réplicas estiverem prepared(m,v,n,i) para todo i ∈ conjunto
- Definimos outra condição em que dizemos que uma réplica está **committed-local(m,v,n,i)** se ela está preparada e recebeu 2f+1 commits (inclusive o próprio)
- Observe que se committed-local é verdadeira, então comitted é verdadeira

Envio da Resposta ao Cliente

- Quando está committed-local, a réplica executa a operação indicada na requisição
- E envia a resposta para o cliente
- O cliente aceita uma resposta que recebeu de pelo menos f+1 réplicas diferentes
- Após executarem uma requisição com timestamp n, as réplicas não aceitam mais (simplesmente ignoram) requisições com timestamp < n

Conclusão

- Nesta aula estudamos o algoritmo de consenso bizantino PBFT
- Muito importante nos dias de hoje, inclusive no contexto de blockchains permissionadas

Obrigado!

Lembrando: a página da disciplina é:
https://www.inf.ufpr.br/elias/topredes

Garbage Collection

- O problema: o log não pode crescer infinitamente
- Quanto é possível retirar mensagens do log?
- Claramente (informalmente): mensagens de requisições antigas, para as quais já foram envias respostas ao cliente por f+1 réplicas corretas → podemos retirar, sob condições…
- Por outro lado: é importante manter uma “prova” que o estado está correto
- E provas de que requisições foram corretamente respondiddas

Provas → Checkpoints

- Gravar uma prova de cada requisição individual é muito caro!
- Solução: gravar provas periodicamente, por exemplo a cada 100 requisições
- O momento em que o PBFT vai gerar uma prova de correção é chamado de checkpoint
- Um checkpoint com uma prova (de que está tudo certo) é chamado de checkpoint estável

A Prova de Um Checkpoint

- Chegou a hora de fazer checkpoint (p.ex. juntaram 100 requisições comitadas)
- A réplica i envia para as outras réplicas:

  $\langle \text{CHECKPOINT}, n, d, i \rangle \sigma_i$
- Lembrando: n de num-seq; d de hash (digest)
- Completa a prova quando: são coletadas 2f+1 mensagens de CHECKPOINT de réplicas distintas e corretas
- O checkpoint então se torna estável

Agora sim: Garbage Collection

- Quando um checkpoint fica estável...
- ... podemos eliminar TODAS as mensagens de PRE-PREPARE, PREPARE e COMMIT com números de requisição <= n
- Podemos descartar do log também todos as mensagens referentes a CHECKPOINT anteriores
- Ou seja: basta guardar o último checkpoint estável e nada antes dele

Uma Réplica mantém Checkpoints

- Uma réplica mantém sempre:
  - O último checkpoint estável
  - Zero ou mais checkpoints ainda não estáveis, sendo construídos

O Intervalo do Nun-Seq

- Os números de sequência das requisições devem ficar entre h <= n <= H
- h: deve ser maior que o último checkpoint estável
- H: de ser maior que o número de requisições necessárias para iniciar um checkpoint
  - por exemplo, se faz checkpoint a cada 100 mensagens, então podemos fazer H = 200
  - a ideia é que as réplicas não fiquem paradas esperando para completar um checkpoint, seguem pra frente executando

Mudanças de Visão

- O que dispara uma mudança de visão?

Mudanças de Visão

- O que dispara uma mudança de visão?
- A falha da réplica primária
- São utilizados *timeouts* para detectar a falha
- Uma réplica dispara um timer assim que recebe uma requisição para executar
- Tudo tem que terminar antes do timeout soar!

Timeout!

- Quando ocorre um timeout, a réplica vai iniciar uma mudança da visão v, para a visão v+1
- Enquanto está fazendo a mudança de visão a réplica não aceita mensagens, exceto:
  - mensagens de checkpoint e referentes a mudança de visão (CHECKPOINT, VIEW-CHANGE, NEW-VIEW)

View Change!

- A réplica disparando a mudança de visão envia para as demais (broadcast):

  <VIEW-CHANGE, v+1, n, C, P, i>σ_i

- n: num-seq do último checkpoint estável
- C: as 2f+1 mensagens que provas que está correto
- P conjunto de conjuntos Pm, cada um com as mensagens preparadas em i para num-seq > n

Cada Conjunto Pm

- Pm contém a mensagem PRE-PREPARE +2f PREPAREs vindos de diferentes réplicas (mesma visão, mesmo n, assinaturas OK)
- O primário p da visão v+1 é pré-definido
  - para a visão v o primário pode ser v MOD |R|
- Quando este primário da visão v+1 recebe 2f mensagens válidas de VIEW-CHANGE, transmite por broadcast <VIEW-CHANGE, v+1, V, O>
  - V: as 2f msgs VIEW-CHANGE; O mostrado a seguir

O Conjunto O

- O é um conjunto de mensagens PRE-PREPARE (sem as requisições propriamente ditas, só o header), calculado da seguinte maneira:
  - Seja min-s o num-seq do último checkpoint estável em v
  - Seja max-s o maior num-seq de um PREPARE em v
  - O primário cria um PRE-PREPARE para cada requisição com num-seq tal que: min-s < num-seq <= max-s
  - Os PRE-PREPAREs são da nova visão:

    <PRE-PREPARE, v+1, n, d>σ_p

As Lacunas: no-op

- Podem haver lacunas
- Para algum num-seq não se localiza nenhuma operação (foi perdida, algo aconteceu ela não foi comitada e executada)
- Neste caso, o novo primário gera um PRE-PREPARE para a operação no-op

Primário Completa O

- O primário então acrescenta todas as mensagens de O ao seu log
- Agora tá tudo pronto:
- Inaugurada a nova visão v+1
- Vamos ver como fazem as réplicas secundárias (backups)

Réplicas Secundárias e O

- Cada réplica deve validar o conjunto O, da mesma forma que o primário fez
- Deve atualizar seus logs
- Enviar PREPAREs de cada mensagem de O por broadcast para as demais réplicas
- Acrescentar os PREPAREs ao seu log
- Pronto! Tá na visão v+1

As Requisições Anteriores

- Todas as requisições que estavam sendo executadas na visão v, tem que ser inteiramente re-executadas na visão v+1
  - São as requisições com num-seq entre min-s e max-s
- Se respostas foram enviadas anteriormente ao cliente, esta informação deve ser armazenada, para não mandar de novo!

Requisições Faltantes em 1 Réplica

- É possível que alguma réplica esteja sem alguma das requisições que estão sendo re-executadas
- Ao receber V, a réplica pode identificar se perdeu alguma requisição e de qual réplica pode obter a informação

Corretude

- Um outline das provas apenas
- Lembrando as duas âncoras das provas em Sistemas Distribuídos
  - *Safety*: propriedade segundo a qual nada de ruim vai acontecer
    - Todas as réplicas concordam com a mesma ordem de commits
  - *Liveness*: algo de bom vai acontecer
    - Os clientes recebem respostas para suas requisições após um intervalo de tempo finito

Prova da Safety: Outline

- Vimos que se prepared(m, v, n, i) é verdade
- Então prepared(m’, v, n, i) é falso se m ≠ m’
- O ponto é garantir que se tudo estiver acontecendo e uma mudança de visão se torna necessária, a corretude é mantida
- O protocolo de mudança de visão garante que todas as réplicas entram em acordo sobre os num-seqs de todos os commits executados

Continuando a Safety

- Veja que se uma réplica comitou, isso significa que pelo menos f+1 réplicas corretas estiveram prepared(m, v, n, i); seja R1 este conjunto
- Nenhuma réplica correta aceita um pre-prepare para uma nova visão v+1 sem antes ter executado direitinho a new-view
- Assim, existe um conjunto R2 de 2f+1 réplicas que executaram new-view
- Como |R| = 3f+1, os conjuntos R1 e R2 têm interseção em pelo menos 1 réplica, digamos k
- Pelo menos k garante que todas as requisições preparadas em v são tratadas na mudança de visão para v+1

Agora a Liveness

- Liveness: os clientes recebem respostas para suas requisições após um intervalo de tempo finito
- Assim é importante que os timeouts sejam precisos o suficiente
- Apesar de que não há premissa sobre sincronização de relógios:
  - seja atraso(t) o tempo para transmitir uma mensagem
  - assume-se que atraso(t) não cresce mais rapidamente que t indefinidamente

Otimizações

- É possível aplicar 3 otimizações para reduzir o custo do PBFT em termos de comunicação
- 1ª Otimização: ao invés de todas as réplicas enviarem a resposta completa ao cliente
  - apenas 1 envia, as demais enviam só o hash
  - se as repostas são muito grandes (arquivos inteiros, por exemplo) pode valer a pena

2ª Otimização

- Permite reduzir em alguns casos os momentos em que se esperam mensagens de 5 para 4
- Se uma réplica entra no estado prepared para uma determinada execução, e todas as requisições anteriores já comitaram
- As réplicas podem enviar respostas tentativas ao cliente
- Neste caso o cliente tem que aguardar 2f+1 respostas, pois daí sabe que comitou
- Se não receber 2f+1: aguarda f+1 definitivas

3ª Otimização

- Esta otimização só pode ser feita para operações de leitura
- Veja que não podemos sair repondendo imediatamente todas as requisições deste tipo ao cliente:
  - podem estar acontecendo operações de escrita concorremente
  - não mandar dados temporários/contraditórios ao cliente!

Completando a 3ª Otimização

- Assim, após completar o commit de todas as operações sendo executadas quando chega a operação de leitura
- É possível executá-la e enviar diretamente uma resposta tentativa ao cliente
- Mais uma vez, o cliente precisa coletar 2f+1 respostas
- Mais uma vez, se não der certo, aguarda f+1 definitivas

Conclusão

- Nesta aula estudamos o algoritmo de consenso bizantino PBFT
- Depois estudamos o processo de checkpointing, mais a coleta de lixo
- Em seguida vimos o protocolo de troca de visão
- E um outline das provas de correção
- Por fim as 3 otimizações de comunicação
- Muito importante nos dias de hoje, inclusive no contexto de blockchains permissionadas

Obrigado!

Lembrando: a página da disciplina é:
https://www.inf.ufpr.br/elias/topredes

Obrigado!

Lembrando: a página da disciplina é:
https://www.inf.ufpr.br/elias/topredes