

Departamento de Ciência da Computação – Universidade de Brasília (UnB)
Brasília – DF – Brasil
Teleinformática e Redes 2 -

Manoel Vieira C Neto
Matrícula 180137816
vieiranetoc@gmail.com

Abstract

1. Objetivos

2. Introdução

3. Algoritmo ABR

O algoritmo foi implementado em python na plataforma PyDash e foi baseado no PANDA[1]. Esse algoritmo é dividido em 4 etapas.

A proposta desse algoritmo é baseado no fato de que nem sempre a taxa de transferência é uma medida justa para o compartilhamento de largura de banda. Desse modo, o algoritmo propõe que sejam feitos pequenos incrementos na banda ao mesmo tempo que se prepara para reduzir a taxa no caso de congestionamentos na rede. Esses incrementos são feitos toda vez que o cliente solicita um novo segmento do vídeo para ser baixado.

3.1. PyDash

A implementação do algoritmo depende de como está estruturado o PyDash. Na plataforma, o algoritmo é criado no arquivo r2a_panda.py com uma classe de nome R2A_Panda. Essa classe possui os seguintes atributos:

- `throughputs` - é uma lista com a taxa de transferência de todos segmentos obtidos no repositório.
- `calc_throughputs` - é uma lista com a taxa de transferência alvo de todos os segmentos.
- `smooth_throughputs` - é uma lista com a taxa de transferência filtrada de todos os segmentos.
- `request_time` - é utilizado para obter o tempo necessário para que a transferência de um segmento seja feito.

- `inter_request_time` - é uma lista contendo o tempo entre a requisição de 2 segmentos consecutivos ou o tempo necessário para a transferência de um segmento, podendo esse tempo ser o real ou o alvo.
- `qi` - é uma lista contendo todas as qualidades disponíveis para ser transferido, obtidas a partir do arquivo mpd.
- `seg_duration` - é um valor que corresponde ao tamanho de um segmento em segundos.
- `selected_qi` - é uma lista contendo a qualidade selecionada para cada segmento transferido.

A duração de um segmento é um fator importante para definir quais serão transferidos. Desse modo, o valor foi definido para ser sempre de 1s e, consequentemente, o valor de `seg_duration` é 1. Os outros valores são obtidos pela aplicação do algoritmo e serão apresentados nas subseções seguintes.

Além dos atributos da classe, os métodos são importantes para o funcionamento adequado e eles são:

- `handle_xml_request` - faz a requisição para transferir o arquivo mpd do repositório para o cliente.
- `handle_xml_response` - recebe o resultado da requisição do arquivo mpd.
- `handle_segment_size_request` - faz a requisição para transferir um novo segmento do vídeo quando solicitado pelo cliente.
- `handle_segment_size_response` - recebe a resposta da requisição de transferência de um segmento do vídeo.

3.2. Estimativa da banda compartilhada

A estimativa da banda compartilhada é a primeira etapa do algoritmo.

Normalmente, a taxa de transferência do segmento atual do vídeo ($x[n]$) é usada para obter o bitrate do segmento que deve ser baixado. Essa taxa pode ser obtida pela seguinte equação:

$$x[n] = x[n-1] = \frac{r[n-1] \cdot \tau}{T[n-1]} \quad (1)$$

Em que:

- $n-1$ representa o segmento anterior.
- n representa o segmento atual.
- τ a duração de um segmento.
- $r[n-1]$ representa o bitrate do segmento anterior.
- $T[n-1]$ representa o tempo necessário para recuperar do repositório o segmento anterior.

Contudo, como nem sempre essa taxa é uma medida justa, o algoritmo determina a taxa média de transferência alvo. Essa taxa, apesar de não representar o valor real na rede, é utilizado para obter o bitrate e também gerenciar o tempo entre as requisições de segmentos. A seguir é apresentada a equação para obter essa taxa:

$$X[n] = x[n-1] + T[n-1] \cdot k \cdot (w - \max(0, X[n-1] - x[n-1] + w)) \quad (2)$$

Em que:

- X representa a taxa de transferência alvo.
- k representa a taxa de convergência de prova.
- w representa o incremento de prova do bitrate.

Essa equação é implementada no método `handle_segment_size_request` com o código:

```
x = abs((w - max((0, self.calc_throughputs[-1]
- self.throughputs[-1] + w))
) * k * self.inter_request_time[-1] + self.
calc_throughputs[-1])
```

Em que x corresponde a $X[n]$ e os atributos usados com `[-1]` correspondem ao último elemento de uma lista, no caso os valores obtidos com o segmento anterior. A função `abs()` é responsável por garantir que não ocorram valores negativos para a taxa obtida.

3.3. Filtragem da taxa de transferência alvo

Depois que a taxa é obtida com a equação 2, é usada uma função EWMA de suavização de modo a obter uma taxa filtrada $y[n]$. O objetivo em obter isso, é para não existir anomalias, pontos fora da curva da função. A aplicação dessa função é apresentada a seguir:

$$y[n] = y[n-1] - T[n-1] \cdot \alpha \cdot (y[n-1] - X[n]) \quad (3)$$

Em que:

- $y[n]$ representa a taxa filtrada para o segmento atual.
- $y[n-1]$ representa a taxa filtrada do segmento anterior.
- $T[n-1]$ representa o tempo necessário para recuperar do repositório o segmento anterior.
- α representa a taxa de convergência da suavização.
- $X[n]$ representa a taxa de transferência alvo obtida na equação 2.

Essa equação é implementada no método `handle_segment_size_request` da classe com o código:

```
y = abs(-alfa * (self.smooth_throughputs[-1] -
x) * self.inter_request_time[-1] + self.
smooth_throughputs[-1])
```

Em que y corresponde a $y[n]$. A função `abs()` é responsável por garantir que não ocorram valores negativos para a taxa obtida.

3.4. Quantizar e identificar bitrate do segmento

A taxa filtrada é quantizada de modo que possa ser usada para determinar o bitrate. Porém, não é somente feita a quantização, também é aplicada uma função de deadzone para evitar que ocorram saltos entre níveis adjacentes de bitrate. Essa função é feita da seguinte forma:

$$r[n] = \begin{cases} r_{up}, & \text{se } r[n-1] < r_{up} \\ r[n-1], & \text{se } r_{up} \leq r[n-1] \leq r_{down} \\ r_{down}, & \text{caso contrário.} \end{cases} \quad (4)$$

Em que:

- r_{up} corresponde a $y[n] \cdot (1 - \epsilon)$.
- r_{down} corresponde a $y[n]$.
- $r[n-1]$ corresponde ao bitrate do segmento anterior.
- ϵ corresponde a uma margem de multiplicação.

Definido o valor de $r[n]$, é obtido o bitrate disponível mais próximo desse valor. O resultado é o bitrate que será solicitado ao repositório para ser transferido.

Essa função de deadzone e a quantização do bitrate são implementadas no método `handle_segment_size_request` da classe com o código:

```

selected_rup = self.qi[0]
selected_rdown = self.qi[0]

rup = y * (1 - E)
rdown = y

for i in self.qi:
    if rup > i:
        selected_rup = i
    if rdown > i:
        selected_rdown = i

if len(self.selected_qi) == 0:
    self.selected_qi.append(selected_rdown)
elif self.selected_qi[-1] < selected_rup:
    self.selected_qi.append(selected_rup)
elif selected_rup <= self.selected_qi[-1] <
    selected_rdown:
    self.selected_qi.append(self.selected_qi[-1])
else:
    self.selected_qi.append(selected_rdown)

msg.add_quality_id(self.selected_qi[-1])

```

Em que:

- rup corresponde ao r_{up} .
- rdown corresponde ao r_{down} .
- selected_rup corresponde à quantização de r_{up} .
- selected_rdown corresponde à quantização de r_{down} .

A aplicação da equação de deadzone adiciona o bitrate escolhido na lista selected_qi e, assim, o último elemento da lista é adicionado à mensagem para ser feita a requisição. Após isso, a requisição é feita.

3.5. Tempo entre requisições de segmentos

A última etapa do algoritmo consiste em obter o tempo entre requisições alvo, ou seja, não corresponde necessariamente ao tempo que uma transferência é feita, mas quanto tempo seria necessário caso a taxa correspondesse ao alvo ($\hat{T}[n]$). Além disso, o algoritmo propõe a inclusão do tamanho do buffer relacionado a um tamanho mínimo dele nesse tempo de modo que em certas ocasiões o tempo para a próxima requisição possa ser maior se o buffer for superior ao definido como valor mínimo. Isso pode ser observado na seguinte equação:

$$\hat{T}[n] = \frac{r[n] \cdot \tau}{y[n]} + \beta \cdot (B[n-1] - B_{min}) \quad (5)$$

Em que:

- β corresponde à taxa de convergência do buffer do cliente.
- $B[n-1]$ corresponde ao buffer do cliente após o recebimento do segmento anterior.

- B_{min} corresponde ao tamanho mínimo do buffer.

Além disso, também foi calculada a taxa real de transferência ($\tilde{T}[n]$) usando a equação:

$$\tilde{T}[n] = \frac{r[n] \cdot \tau}{x[n]} \quad (6)$$

Com essas duas taxas calculadas, o tempo entre requisições foi definido como:

$$T[n] = \max(\hat{T}[n], \tilde{T}[n]) \quad (7)$$

Esse é o $T[n]$ utilizado na equação 2. Na implementação isso é calculado no método `handle_segment_size_response` da classe com o código:

```

target_inter_time = msg.get_bit_length() * self
    .seg_duration / self.calc_throughputs[-1] +
    beta * (
        B - buffer_min)
actual_inter_time = time.perf_counter() - self.
    request_time

self.inter_request_time.append(max((
    target_inter_time, actual_inter_time)))

```

Em que:

- target_inter_time corresponde ao $\hat{T}[n]$.
- beta corresponde ao β .
- B corresponde ao tamanho do buffer.
- buffer_min corresponde ao tamanho mínimo do buffer.
- actual_inter_time corresponde ao $\tilde{T}[n]$.

3.6. Inicialização

No início do algoritmo alguns valores não estão definidos e por isso eles são determinados antes da aplicação do algoritmo.

A taxa de transferência alvo e a filtrada são iguais à real no momento da inicialização e são obtidas usando o tamanho do arquivo mpd e o tempo necessário para transferir esse arquivo. No código elas são obtidas no método `handle_xml_response` da seguinte forma:

```

if len(self.throughputs) == 0:
    self.throughputs.append(msg.get_bit_length()
        / t)
    self.calc_throughputs.append(msg.
        get_bit_length() / t)
    self.smooth_throughputs.append(msg.
        get_bit_length() / t)

```

Como também não há um tamanho para o buffer ainda, o valor usado para obter o tempo alvo entre requisições é obtido com a equação:

$$B_0 = B_{min} + (1 - \frac{r_0}{X_0}) \cdot \frac{\tau}{\beta} \quad (8)$$

Isso é implementado no método `handle_segment_size_response` da classe com o código:

```
if len(self.throughputs) == 1:  
    B = (1 - msg.get_bit_length() /  
         self.calc_throughputs[0]) * self.  
        seg_duration / beta + buffer_min
```

4. Resultados

5. Discussão e Conclusões

Referências

- [1] Z. Li, X. Zhu, J. Gahm, R. Pan, H. Hu, A. C. Begen, and D. Oran. Probe and adapt: Rate adaptation for http video streaming at scale.