Universidade Federal do Pampa

Autor: Anderson Schmitz Venturini

DESENVOLVIMENTO DE UM GERENCIADOR DINÂMICO DE MEMÓRIA MULTITHREAD PARA UM PROXY DE VIDEO SOB DEMANDA

Trabalho de Conclusão de Curso II

BAGÉ
2013

ANDERSON SCHMITZ VENTURINI

DESENVOLVIMENTO DE UM GERENCIADOR DINÂMICO DE MEMÓRIA MULTITHREAD PARA UM PROXY DE VIDEO SOB DEMANDA

Trabalho de conclusão de curso apresentado ao curso de graduação em Engenharia de Computação da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Computação.

Orientador: Prof. Msc. Bruno Silveira Neves

BAGÉ
2013

ANDERSON SCHMITZ VENTURINI

DESENVOLVIMENTO DE UM GERENCIADOR DINÂMICO DE MEMÓRIA MULTITHREAD PARA UM PROXY DE VIDEO SOB DEMANDA

Trabalho de conclusão de curso apresentado ao curso de graduação em Engenharia de Computação da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Computação.

Orientador: Prof. Bruno Silveira Neves

Dissertação defendida e aprovada em: 11 de maio de 2013.

Banca examinadora:

_____

Prof. Msc Bruno Silveira Neves
Orientador
UNIPAMPA

_____

Prof. Dr. Leonardo Bidese de Pinho
UNIPAMPA

_____

Prof.ª Dr.ª Ana Paula Lüdtke Ferreira
UNIPAMPA

Dedicatória

Dedico este trabalho especialmente à minha família, Pedro, Gelci e Mateus, que ensinaram o valor do estudo, trabalho e dedicação e que acima de tudo, sempre me apoiaram, deram carinho e amor.

Agradecimentos

Gostaria de agradecer aos amigos e colegas pelo incentivo e companheirismo conquistado nessa jornada. Ao professor Bruno, pela amizade, ajuda e orientação no desenvolvimento deste trabalho. A todos os professores do curso de Engenharia de Computação da UNIPAMPA que jamais mediram esforços para ajudar e transmitir seus conhecimentos. Por fim, a todos que, direta ou indiretamente, contribuíram para a realização deste trabalho.

Epígrafe

"Don’t complain about the problem - fix it."
Eddie R. Lawson

Lista de Figuras

Lista de Tabelas

Lista de Siglas

VsD	Video sob Demanda
VoD	*Video-on-Demand*
CDN	*Content Distribution Network*
QoS	*Quality of Service*
LRU	*Least Recently Used*
LFU	*Least Frequently Used*
TI	Tecnologia da Informação
VCR	*Video Cassette Recording*
CBR	*Constant Bit Rate*
HECo	*High Efficiency Computing*
CCVC	*Collapsed Cooperative Video Cache*
MCC	Memória Cooperativa Colapsada
CC	*Chunk-based Caching*
CARTE	*Current demAnd Rather Than futurE*
CPU	*Central Processing Unit*
SA	*Simulated Annealing*
IPCM	*Intel Performance Counter Monitor*
PMU	*Performance Monitor Unit*
Mbps	Megabits por segundo
API	*Application Programming Interface*
DDR	*Double Data Rate*
RAM	*Random Access Memory*
DMA	*Direct Memory Access*

Sumário

Resumo

Com a crescente utilização dos serviços de video sob demanda (VsD) na atualidade, faz-se necessária a utilização de técnicas que permitam o provimento do serviço com qualidade. Com este intuito são implantadas redes de distribuição de conteúdo (CDN) a fim de reduzir o tráfego de dados na dorsal da rede e a latência de resposta aos clientes. Para isso são implementados *proxies* na borda da rede. Entretanto, a eficiência e o custo dos *proxies* são cruciais para a sustentabilidade do serviço oferecido através da CDN. Assim, este trabalho propõe a implementação de uma solução multithread que busque melhores custos-benefícios quando comparada às implementações de *proxies* de VsD atuais executadas sobre servidores de propósito geral.

Palavras-chave: distribuidor; Video sob demanda; gerenciador de memória.

Abstract

With the growing utilization of the Video-on-Demand (VoD) services in the present, it is necessary the implementation of techniques that allow the provision of the service with quality. For this purpose, Content Distribution Networks (CDN) look to reduce the data traffic of the network's backbone and the response latency to the clients. Therefore, proxies are implemented in the frontier of the CDN. Furthermore, the eficiency and cost of the proxies are crucial to the sustainability of the service offered. So, we propose an implementation of a multithread solution that aims for a better cost-benefit when compared to the implementation of the current VoD proxies running on general purpose servers.

Key-words: proxy; Video-on-Demand; memory manager; hardware.

1 Introdução

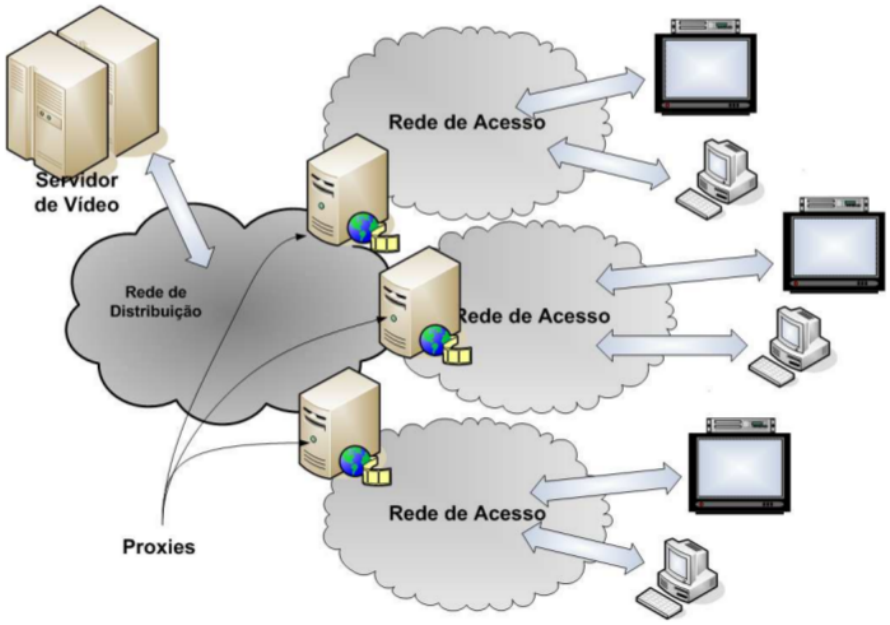
1.1 Visão Geral

Com o avanço da tecnologia na última década, os serviços de vídeo sob demanda (*VoD - Video-on-Demand*) têm colaborado com uma significante parte do aumento do tráfego de dados da Internet nos dias de hoje (SOUTO; CAVALCANTI; MARTINS, 2010). Uma das técnicas amplamente utilizadas para dar suporte a tal demanda é a utilização de redes de distribuição de conteúdo (*CDN - Content Distribution Network*) que buscam manter a qualidade de serviço (*QoS - Quality of Service*), agilizando o tempo de resposta aos clientes e reduzindo o tráfego de comunicação no *backbone* da rede. Para isto, são implementados servidores intermediários (*proxies*) na borda da CDN [Figura 1.1]. Estes *proxies* atuam como servidores secundários de menor porte, armazenando os dados mais acessados e entregando estas informações aos clientes ao invés do servidor principal.

Desta forma, a eficiência dos *proxies* de VoD é fundamental para o provimento do serviço com qualidade e para a implementação da CDN, uma vez que *proxies* eficientes possibilitam a redução dos custos de implantação de CDNs, através da redução do consumo da banda no *backbone* da rede e no servidor VoD, além do aumento do número de clientes suportados. Isto permite que o serviço seja provido aos clientes com preços mais acessíveis, o que tende a levar ao aumento da popularização do mesmo.

Escolher uma política de substituição de *cache* apropriada possui um papel crucial no design de um *proxy*, uma vez que é a chave para atingir uma alta eficiência em acertos de *cache* em função das requisições. Tipicamente, as estratégias de substituição de *cache* clássicas, como *LRU* e *LFU*, ainda são amplamente utilizadas em *proxies* devido as suas simplicidade, custos computacionais e desempenho aceitáveis (WU et al., 2012). Entretanto, estratégias de *caching* de *proxies* Web convencionais não dão o suporte necessário para serviços de *VoD* devido ao grande tamanho dos objetos multimídia e da alta demanda de banda requisitada. Assim, é necessário a implementação de algoritmos otimizados para este tipo de serviço, que sejam capazes de adaptarem-se às limitações e aos padrões de acesso dos clientes dinamicamente (como por

Figura 1.1 – Arquitetura genérica de um sistema de VoD.



Fonte: Bragato, 2006, p. 7.

exemplo, a variação da velocidade de transmissão e a interatividade dos clientes no decorrer dos vídeos, avançando, retrocedendo e pausando a execução) e às peculiaridades dos objetos multimídia (segmentação e estrutura de execução temporal existente em objetos de vídeo, por exemplo).

No decorrer dos últimos anos, diversos novos algoritmos de gerenciamento de memória foram desenvolvidos buscando aumentar a eficiência dos *proxies* voltados para *VoD* (LIU; XU, 2004) (SO-IN, 2005) (LI; CHEN, 2009) (HONG; VLEESCHAUWER; BACCELLI, 2010). Todavia, atualmente os *proxies* são inteiramente desenvolvidos em software, executando sobre arquiteturas tradicionais, como servidores de propósito geral. Assim, essas plataformas potencialmente apresentam grande desproporções dos recursos oferecidos, prejudicando a eficiência global do sistema e ocasionando o desperdício do investimento realizado para a implantação do serviço. Dentre as técnicas utilizadas para tornar possível o provimento do serviço às grandes massas de requisições está a utilização de *clusters* de *proxies* (CHEN et al., 2003)(CHANDRA; SAHOO, 2009)(MOGHAL; MIAN, 2003), porém, frente ao movimento recente de redução dos impactos ambientais causados pela infraestrutura necessária para o provimento de serviços de TI (como redução do consumo energético, busca por materiais sustentáveis, reciclagem de equipamentos, etc), esta estratégia demonstra-se ineficiente tendo em vista os gastos despendidos para a refrigeração e instalação e área consumida pelas fazendas de *clusters*.

Desta forma, este trabalho propõe uma nova implementação que vise a melhor relação custo-benefício para a aplicação alvo, através do desenvolvimento de um gerenciador dinâmico de memória *multithread* e de uma nova estratégia de substituição de memória que vise reduzir o consumo de banda no *link* do *proxy* e o servidor *VoD*.

1.2 Estrutura do Trabalho

Este trabalho está dividido em capítulos, mantendo uma ordem lógica e coerente. O primeiro capítulo, Introdução, além de uma visão geral do escopo do trabalho, também apresenta a problematização, objetivos (geral e específicos), metodologia geral e o cronograma seguido. No capítulo 2 apresenta-se o *proxy* de VoD desenvolvido, incluindo as estruturas de dados, módulos de programa, as estratégias de substituição de memória estudadas e a arquitetura paralela desenvolvida. No capítulo 3 é exibida a versão *multithread* paralela do *proxy* de VoD implementado. O capítulo 4 descreve o ambiente de simulação implementado para realizar a análise, com o intuito de abstrair os elementos complexos da rede que normalmente interagem com o *proxy*, tais como o servidor de vídeo primário e os clientes. A validação do sistema, avaliações experimentais e os resultados obtidos são apresentados e discutidos nos capítulos 5 e 6. No capítulo 7, são apresentados os comentários finais sobre todas as atividades realizadas, quanto aos resultados obtidos e propostas de trabalhos futuros.

1.3 Objetivos

1.3.1 Objetivo Geral

Este trabalho tem como objetivo principal o desenvolvimento de um gerenciador de memória multithread para um *proxy* de *VoD* em software que busque melhores custos-benefícios quando comparado às implementações de *proxies* atuais executadas sobre servidores de propósito geral.

1.3.2 Objetivos Específicos

Pontualmente, este trabalho propõe:

1. Estudar e implementar técnicas para:
    - *Caching* de objetos multimídia.

- Software de alto desempenho.
- Comparar o desempenho das estratégias de gerenciamento de memória.

2. Aprofundar estudos sobre Linguagem C.

1.4 Metodologia

O trabalho está estruturado com base nos seguintes passos:

Levantamento e estudo da literatura. Inicialmente será realizada a pesquisa e o estudo de bibliografias na área de gerenciamento de memória de *proxies* de VoD, de implementação de software de alto desempenho e de desenvolvimento utilizando-se da Linguagem C, com o intuito de aumentar o nível de conhecimento sobre os conceitos fundamentais para o desenvolvimento deste trabalho.

Estudo do gerenciador dinâmico de memória. Em seguida, o gerenciador de memória do *proxy* de VoD proposto por (ISHIKAWA, 2003) é estudado, como solução base para o desenvolvimento do trabalho, com o objetivo de identificar os módulos funcionais principais do sistema.

Revisão de independências entre módulos. Posteriormente, serão identificadas as dependências existentes entre estes módulos com o intuito de levantar subsídeos para o planejamento da implementação de blocos funcionais paralelos em software.

Desenvolvimento do gerenciador de memória paralelo em Linguagem C. Com o estudo dos módulos independentes realizado, será possível dar início ao desenvolvimento do algoritmo de gerenciamento dinâmico de memória paralelo utilizando-se a linguagem C. Paralelamente será feita a integração dos módulos implementados e a validação dos mesmos.

Geração de cargas de teste. Em seguida, serão gerados casos de teste que busquem representar diferentes cenários de carga de trabalho para um *proxy* VoD, como por exemplo, a variação da quantidade de memória do *proxy*, o intervalo de tempo entre novas requisições dos clientes, taxa de codificação, número máximo de clientes ativos e diferentes comportamentos para estes clientes.

Coleta e análise de resultados. Consecutivamente, estes cenários são utilizados como base para testar a eficiência do gerenciador de memória implementado, gerando dados como a taxa de hits de memória, a vazão de dados do sistema, entre outros. Por último, espera-se

também comparar os dados de desempenho do sistema, obtidos através da etapa de testes anterior, através das versões implementadas (serial e multithread), além dos resultados disponíveis na literatura.

1.5 Cronograma

A tabela 1.1 apresenta o cronograma proposto para a realização deste trabalho.

Tabela 1.1: Cronograma de Atividades.

Atividade	Semestre 1				Semestre 2			
	1	2	3	4	1	2	3	4
Levantamento e estudo bibliográfico	x	x			x	x		
Análise do gerenciador de memória do *proxy* de VoD		x						
Revisão de independência e paralelismo dos módulos		x	x					
Desenvolvimento do gerenciador de memória em software		x	x	x	x	x	x	
Integração dos módulos internos e debug		x	x	x	x	x	x	
Escrita do relatório de TCC I	x	x	x	x				
Apresentação do relatório de TCC I				x				
Geração das baterias de teste						x		
Coleta e análise de resultados						x	x	x
Escrita da monografia de TCC II					x	x	x	x
Apresentação da monografia de TCC II								x

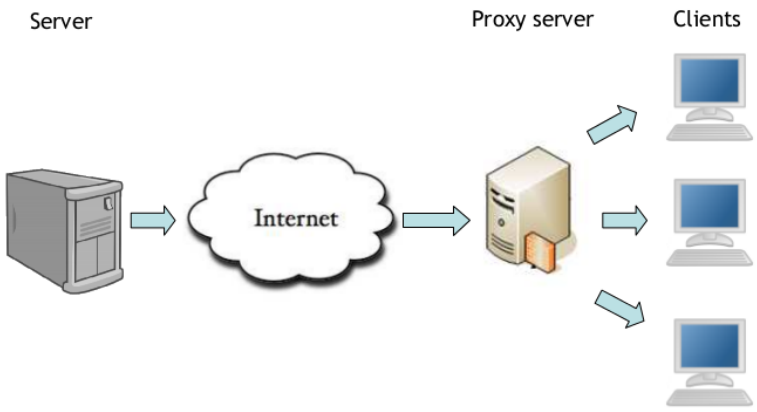
2 Proxy de VoD

O *proxy* ou distribuidor pode ser caracterizado como um servidor secundário de menor porte, posicionado na borda da rede próximo aos clientes, cuja função é a de replicar o conteúdo mais acessado do servidor primário. A inserção de *proxies* na rede, além de evitar o congestionamento na dorsal da rede e reduzir a latência de entrega do conteúdo requisitado pelos clientes, também amplia a capacidade de processamento e armazenamento dos servidores principais uma vez que se considera que o desempenho e capacidade dos *proxies* são somados aos dos servidores primários.

A eficiência de um *proxy* é diretamente proporcional ao uso do conteúdo replicado que ele consegue armazenar. Por isso, um *proxy* procura armazenar apenas o conteúdo mais popular, de modo a manter sua taxa de reaproveitamento a mais alta possível. Contudo, objetos multimídia apresentam um grande volume de dados, o que consome rapidamente a capacidade de armazenamento. A restrição do espaço de *caching* faz com que o *proxy* não armazene todos os objetos de vídeo requisitados pelos clientes. Uma solução é a alocação de apenas porções dos vídeos. Por exemplo, uma estratégia de *caching* pode armazenar porções de um objeto de vídeo de acordo com a frequência de acesso dos usuários (HSU; LI, 2011).

A estrutura de rede estudada é representada na Figura 2.1. Quando um cliente realiza uma requisição por um bloco de vídeo, esta requisição deve ser tratada primeiramente pelo *proxy*, verificando se o conteúdo requisitado está disponível em sua *cache*. Caso o conteúdo esteja disponível, então o mesmo será transmitido para o cliente diretamente, o que resulta em um '*hit*' de *cache*. Caso contrário, considera-se que houve uma falha de acesso (*cache miss*) e, nestas circuntâncias, o *proxy* requisita o bloco de vídeo não disponível em sua memória ao servidor principal. Uma vez que o bloco tenha sido enviado e replicado na memória do *proxy*, este poderá ser encaminhado ao respectivo cliente. Opcionalmente, o *proxy* é capaz de decidir se o conteúdo adquirido do servidor deve ser armazenado em memória. Se não houver espaço livre em memória, o algoritmo de reciclagem do *proxy* realiza o descarte de parte do conteúdo armazenado na memória de forma a liberar espaço para que o novo conteúdo proveniente do servidor possa ser alocado.

Figura 2.1 – Estrutura servidor-*proxy*-clientes.



Fonte: Wu, 2012, p. 90.

O gerenciador de memória a ser proposto por este trabalho basear-se-á na ideia proposta por (ISHIKAWA, 2003) devido aos resultados obtidos em relação aos trabalhos de estado da arte existentes na área e às peculiaridades de sua estratégia, tais como:

- Gerenciamento dinâmico que leva em consideração o histórico de acessos aos vídeos pelos clientes como critério para definição do conteúdo a ser mantido em memória.
- O modelo de preservação de blocos de vídeo em memória que mantém alocados os blocos de vídeo acessados por alguns clientes com base na demanda efetiva gerada por novos clientes que acessarão o mesmo conteúdo.
- Suporte a escalabilidade que prevê o resgate de recursos reservados em cenários de alta carga de trabalho, para atendimento a demanda crescente de clientes.

A partir de simplificações preliminares para a implementação do *proxy*, de modo a eliminar fatores que pudessem influenciar os dados obtidos e facilitar o desenvolvimento inicial do projeto, fizemos as seguintes suposições:

Entrega instantânea entre as entidades: desta maneira o tempo de transmissão do conteúdo entre o servidor principal, *proxy* e clientes não é levado em consideração, ou seja, os segmentos de vídeo são entregues instantaneamente, demorando apenas o tempo de escrita e leitura dos *buffers* do sistema.

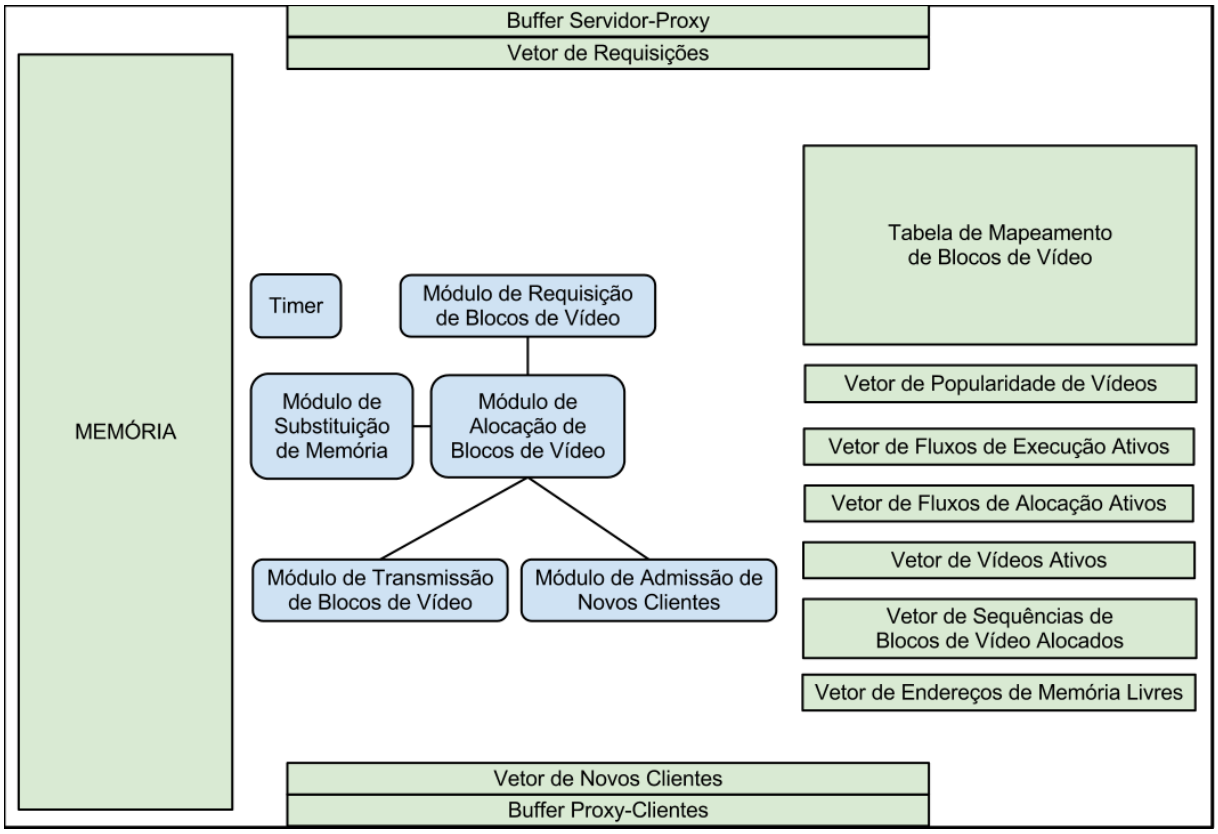
Não há interrupção ou desvio do fluxo de execução: os clientes não interrompem seus fluxos de execução enquanto estes não assistirem o vídeo por completo. Assim, sendo realizada a primeira requisição a partir de um ponto qualquer do vídeo, cada cliente requisita

segmento-a-segmento até o fim do vídeo. Ou seja, não são suportadas funcionalidades como de *VCR* (avançar, retroceder e pausar).

Vídeos de mesmo tamanho e taxa de codificação constante: todos os vídeos utilizados possuem a mesma duração, taxa de fluxo de dados constante (*CBR*) e cada segmento de vídeo possui o mesmo tamanho em bytes de acordo com a taxa especificada.

A Figura 2.2 apresenta a visão geral do *proxy* de VoD desenvolvido utilizando-se a linguagem de programação C (KERNIGHAN; RITCHIE, 1988). O restante deste capítulo está organizado como segue: nas seções 2.1 e 2.3 são apresentadas, respectivamente, as estruturas de dados que compõem o sistema e os módulos de programa que desempenham o gerenciamento do *proxy*. A seção 2.2 descreve as estratégias de substituição de memória estudadas. A seção 2.4 relata o funcionamento passo-a-passo do algoritmo implementado.

Figura 2.2 – Estrutura do *proxy* de VoD.



Fonte: Arquivo pessoal.

2.1 Estruturas de Dados

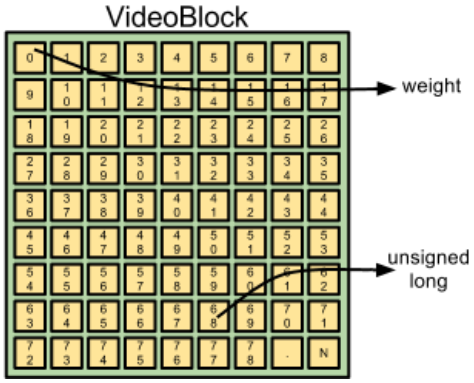
As estruturas de dados implementadas e necessárias para o funcionamento dos sistema são apresentadas a seguir:

***VideoBlock*:** representa um bloco de vídeo [Figura 2.3]. *VideoBlock* é a unidade de menor granularidade gerenciada pelo *proxy*. O tamanho de um bloco de vídeo em bytes pode ser obtido pela Equação (2.1). A estrutura contém um vetor do tipo *unsigned long*, *weight*, usado para manter a quantidade de dados transferida aos usuários igual a taxa de codificação especificada. A Equação (2.2) apresenta o tamanho do vetor *weight* em função da taxa de codificação, onde *sizeof(unsigned long)* = 8 bytes. Ou seja, para um vídeo codificado à *T Mbps* será criado e transmitido um objeto *VideoBlock* com peso igual à taxa de codificação *T*.

$$VideoBlock\_size\_B\ [bytes] = \frac{bitrate\ [Mbps]}{8bits} \times 1MB \tag{2.1}$$

$$VideoBlock \rightarrow weight = \frac{bitrate\ [Mbps]}{sizeof(unsigned\ long) \times 8bits} \times 1MB \tag{2.2}$$

Figura 2.3 – Exemplo da estrutura *VideoBlock*

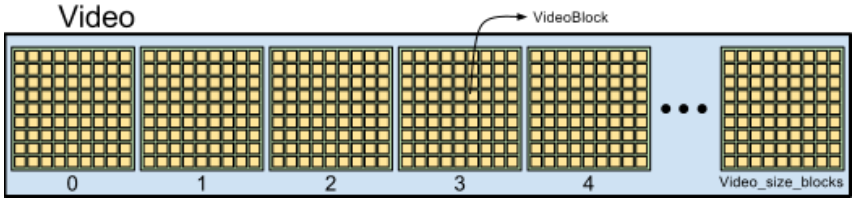


Fonte: Arquivo pessoal.

***Video*:** define um vídeo do acervo do *proxy* [Figura 2.4]. Um objeto *Video* é constituido de uma sequência de *video_size_blocks* blocos de vídeo contíguos, onde *video_size_blocks* é função do tamanho em bytes do vídeo e da taxa de codificação estipulada (Equação 2.3). A estrutura armazena informações como a ID do vídeo, tamanho em megabytes, tamanho em blocos de vídeo, posição de mapeamento na tabela de gerenciamento de blocos de vídeo, número de fluxos de execução e de alocação ativos, número de grupos de usuários contíguos, número de sequências de blocos de vídeo alocadas em memória.

$$video\_size\_blocks = \frac{video\_size\_B\ [bytes] \times 8bits}{bitrate\ [Mbps]} \times 1MB \tag{2.3}$$

Figura 2.4 – Exemplo da estrutura *Video* implementada.



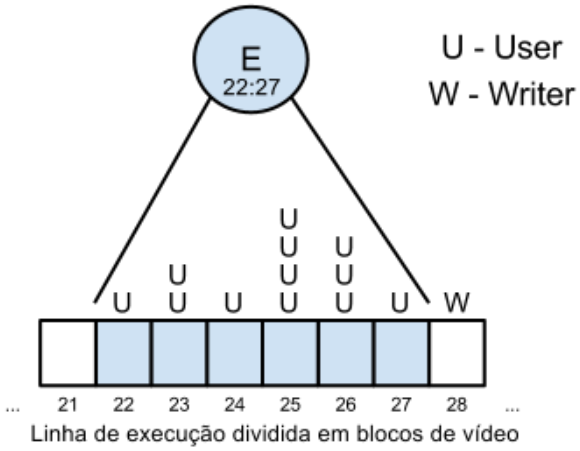
Fonte: Arquivo pessoal.

***User*:** representa um cliente do sistema servido pelo *proxy* (ao longo do desenvolvimento deste trabalho, usaremos alternadamente os termos usuário, cliente e fluxo de execução para nos referirmos ao mesmo ente). A estrutura contém informações como por exemplo, a ID do cliente, bloco de vídeo atual de execução, instante de chegada no sistema e ponteiros para a estrutura *Video* acessada e *Users* prévio e posterior na linha de tempo do vídeo.

***Writer*:** define um fluxo de alocação, cuja função é suprir a demanda de blocos de vídeo proveniente de um ou mais fluxos de execução prévios, agrupando informações referentes ao conteúdo que deve ser requisitado ao servidor principal caso este não esteja alocado em memória. O *Writer* armazena informações como ID do fluxo de alocação, bloco de vídeo atual sendo alocado, a prioridade de requisição do fluxo e ponteiros para a estrutura *Video* correspondente e *Writers* prévio e posterior na lista de *Writers* ativos (*Active_Writers_List*) em função de suas prioridades de requisição.

***UsersGroup*:** representa um grupo de fluxos de execução contíguos, de forma que a criação do mesmo possibilita que os blocos de vídeo alocados e que estão sendo transmitidos para seus respectivos fluxos de execução sejam reservados em memória, e assim permitam que os fluxos de execução prévios na linha de tempo do vídeo reaproveitem o conteúdo sem que seja necessário realizar uma requisição ao servidor principal. Outra vantagem existente com a criação de um *UsersGroup* e a possibilidade de reserva do conteúdo acessado é a necessidade de apenas um fluxo de alocação (*Writer*) para suprir a demanda de requisições gerada pelos clientes pertencentes a um mesmo grupo [Figura 2.5]. Um *UsersGroup* é delimitado pelos ponteiros inicial e final, armazena o número de clientes pertencentes ao grupo, o numero de blocos de vídeo reservados, possui ponteiros para a lista de clientes que pertencem ao grupo, para seu fluxo de alocação e para os nodos *UsersGroup* prévio e posterior na linha de tempo do vídeo e para os nodos prévio e posterior localizados na árvore de estruturas ativas (*Active_Structures_Tree*).

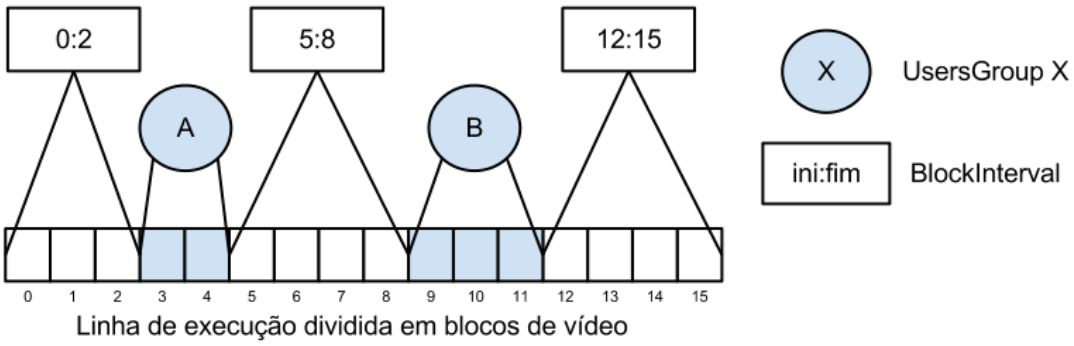
Figura 2.5 – Exemplo de *UsersGroup*



Fonte: Arquivo pessoal.

***BlockInterval*:** define um intervalo de blocos de vídeo entre dois *UsersGroups* existentes, ou entre um *UsersGroup* e a extremidade do vídeo (inicial ou final), ou entre as duas extremidades do vídeo (caso não existam grupos de clientes ativos). A Figura 2.6 apresenta a estrutura de um *BlockInterval*. *BlockIntervals* permitem que os blocos de vídeo alocados em memória sejam gerenciados no decorrer da execução, uma vez que fluxos de execução e de alocação deslocam-se na linha temporal do vídeo, executando, alocando, removendo e reservando blocos de vídeo. A estrutura é delimitada pelos ponteiros inicial e final, possui um ponteiro para o início da lista de *Sequences* pertencentes ao intervalo e ponteiros para os nodos *BlockInterval* prévio e posterior na linha de tempo do vídeo e nodos prévio e posterior localizados na *Active_Structures_Tree*.

Figura 2.6 – Exemplos de *BlockIntervals*

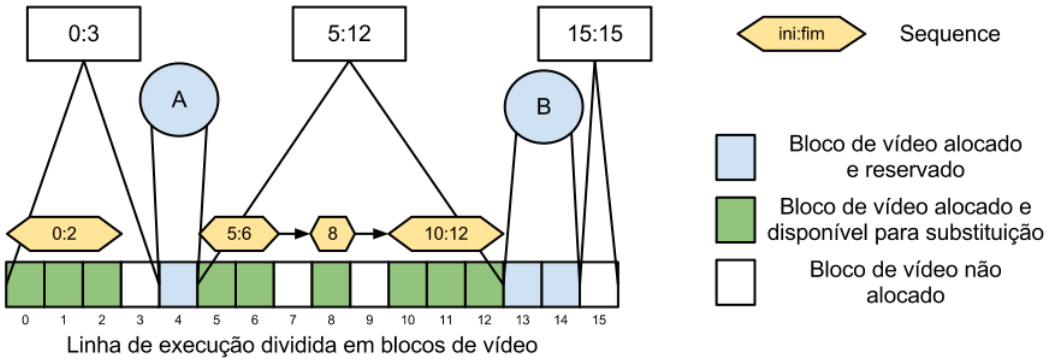


Fonte: Arquivo pessoal.

***Sequence*:** define um conjunto de um ou mais blocos de vídeo contíguos, alocados em memória e que não estão sendo acessados por um fluxo de execução ou de alocação (ou seja,

não encontram-se reservados) [Figura 2.7]. Desta maneira, o *proxy* pode selecionar este conjunto de blocos de vídeo para serem desalocados, liberando espaço em memória para a alocação de novas requisições. Um objeto *Sequence* é delimitado pelos blocos de vídeo inicial e final da sequência, número de blocos de vídeo contidos no conjunto, a prioridade de descarte em função da estratégia de substituição de memória utilizada, e ponteiros para nodos *Sequence* prévio e posterior dispostos em função da prioridade de substituição e para os nodos *Sequence* prévio e posterior na linha de tempo do vídeo.

Figura 2.7 – Exemplos de *Sequences*



Fonte: Arquivo pessoal.

***Memory*:** implementa a memória do *proxy*, implementada através de um vetor estático do tipo *VideoBlock* de *memory_size_blocks* posições, onde *memory_size_blocks* é calculado em função do tamanho em bytes da memória e do tamanho em bytes de um bloco de vídeo (2.4). Cada elemento do vetor *Memory* armazena um bloco de vídeo gerenciável pelo *proxy* que prontamente pode ser transferido aos clientes.

$$memory\_size\_blocks = \frac{memory\_size\_B\ [bytes]}{VideoBlock\_size\_B\ [bytes]} \tag{2.4}$$

***Blocks_Addresses_Table*:** define a tabela de mapeamento de blocos de vídeo, cuja função é mapear os blocos de vídeo dos vídeos servidos pelo *proxy* a endereços de memória (posições no vetor *Memory*). A utilização de uma tabela de mapeamento permite a rápida verificação se um bloco de vídeo encontra-se alocado ou não em memória. Isto pode ser feito através da leitura da posição VxB da tabela, referente ao vídeo mapeado para a linha V e ao bloco de vídeo com *offset* igual a B. Se o valor lido, *addr*, é maior ou igual a zero, então o bloco de vídeo está alocado na posição *addr* da memória. Caso contrário, o bloco de vídeo não está alocado em memória. A tabela é alocada estaticamente, com tamanho [*max_num_videos*, *video_size_blocks*], onde *max_num_videos* corresponde ao

número máximo de vídeos ativos suportados pelo *proxy* e *video_size_blocks* corresponde ao tamanho, em blocos de vídeo, de um vídeo qualquer do acervo (uma vez que todos os vídeos possuem o mesmo tamanho e mesma taxa de codificação).

***Videos_Popularity_Array*:** representa o vetor indireto de ordenação da popularidade dos vídeos servidos pelo *proxy*. Cada posição do vetor contém um ponteiro para a estrutura *Video* correspondente. O vetor possui comprimento *max_num_videos*. Os elementos são ordenados de forma crescente em função do número de clientes ativos em cada vídeo. Isto permite que vídeos com popularidades baixas, ou seja, com menos fluxos de execução ativos, sejam substituídos em priori a vídeos com grande quantidades de fluxos de execução ativos. Caso dois ou mais vídeos possuam a mesma quantidade de clientes ativos, o menor número de substituições de blocos de vídeos realizadas sobre cada vídeo é utilizada como métrica de desempate.

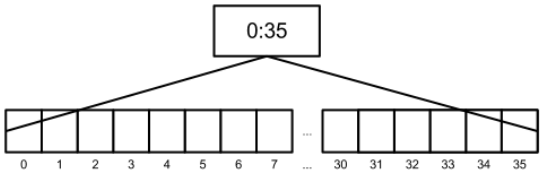
***Free_Addresses_Array*:** implementa o vetor de endereços de memória livres. O vetor possui *memory_size_blocks* elementos. Em conjunto com o contador de endereços livres em memória, *num_free_ addresses*, é possível gerenciar a taxa de desocupação da memória e quais os endereços de memória disponíveis. Conforme os blocos de vídeo são alocados em memória, endereços do vetor são consumidos e o contador *num_free_addresses* é decrementado. Quando a memória é reciclada e blocos são descartados, os endereços são adicionados ao vetor e o valor do contador é incrementado. A rotina de substituição de blocos de vídeo da memória é disparada quando um novo bloco de vídeo deve ser alocado e *num_free_addresses* é igual a '-1'.

***Active_Structures_Tree*:** define uma estrutura em forma de árvore que contém os *UsersGroups* e *BlockIntervals* ativos de um vídeo, dispostos em função dos seus limites iniciais e finais. Desta maneira, é possível verificar de forma eficiente se um novo cliente está localizado em um grupo de clientes já existente, compartilhando assim, o conteúdo já alocado, ou se é necessário criar um novo grupo de clientes e instanciar um novo fluxo de alocação para o mesmo. Inicialmente, quando não existe nenhum *UsersGroup* ativo em um vídeo, a estrutura contém apenas um *BlockInterval* contendo o vídeo por inteiro, como pode ser visto na Figura 2.8a. Conforme novas requisições chegam ao sistema, novos grupos de usuários são criados e ao passo que estes deslocam-se através da linha de execução do vídeo [Figura 2.8b], a árvore vai adquirindo sua forma, atualizando seus nodos incrementalmente de acordo com a execução dos fluxos de execução e dos fluxos de alocação ativos de cada vídeo [Figura 2.8c].

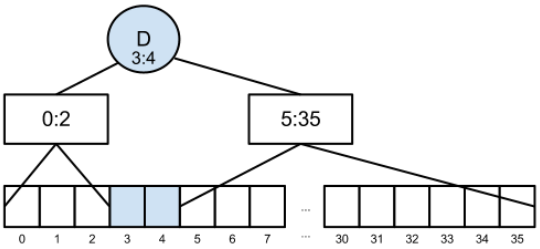
***Sequences_Priority_List*:** representa a lista de *Sequences* de um vídeo, cujos nodos estão dis-

Figura 2.8 – Exemplo de *Active_Structures_Tree*.

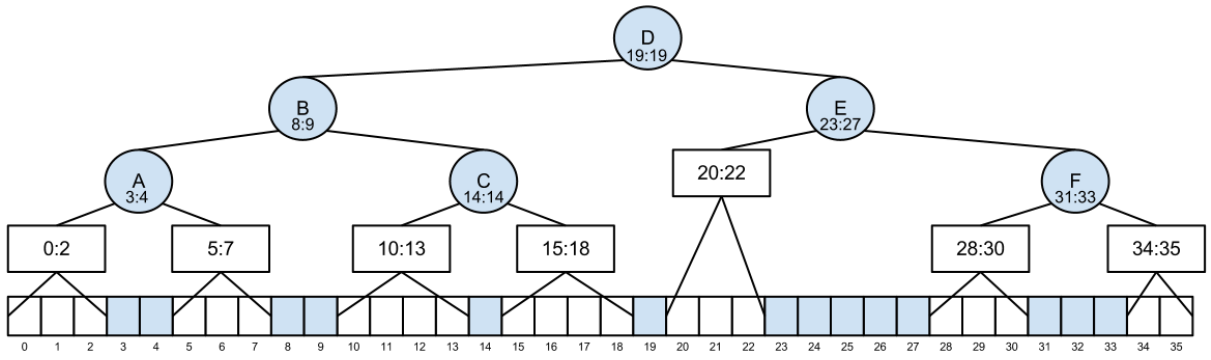
(a) Árvore de Estruturas Ativas Inicial.



(b) Árvore de Estruturas Ativas após inserção de um *UsersGroup*.



(c) Árvore de Estruturas Ativas após diversas inserções e rodadas de serviço executadas.



Fonte: Arquivo pessoal.

postos em ordem decresente em função da prioridade de substituição implementada. Desta forma, busca-se desalocar da memória as sequências de blocos de vídeo com a menor probabilidade de reuso no futuro de acordo com a estratégia de substituição escolhida.

***Active_UsersGroups_List*:** define uma lista de *UsersGroups* ativos de um vídeo, disposta em ordem crescente em função do bloco inicial. A lista de grupos de fluxos de execução contíguos possibilita percorrer os *UsersGroups* em sequência para servir os usuários internos.

***Active_Writers_List*:** representa a lista de *Writers* contendo todos os fluxos de alocação ativos do *proxy*. A lista pode ser disposta em função dos blocos correntes de alocação de cada *Writer* ou pelas suas prioridades de requisição. A prioridade de requisição, *request_priority*, é proporcional ao número de fluxos de execução ativos, contíguos, prévios ao fluxo de alocação que serão beneficiados caso o bloco de vídeo seja requisitado ao servidor principal e alocado.

***New_Users_Array*:** implementa o *buffer* de entrada da interface de rede com os clientes, armazenando todas as novas requisições de clientes que chegam a cada rodada de serviço. Se *New_Users_Array != NULL*, então existe pelo menos um novo fluxo de execução que pode ser admitido pelo *proxy*.

***Requisitions_Array*:** estrutura de dados que representa o *buffer* de saída da interface de rede com o servidor, armazenando todas as requisições realizadas pelo *proxy* ao servidor principal a cada rodada de serviço. São escritos no *buffer* a ID do vídeo e o *offset* do bloco de vídeo requisitado.

***Server_Proxy_Buffer*:** representa o *buffer* de entrada da interface de rede com o servidor. Armazena o conteúdo enviado pelo servidor principal, disponível para alocação em memória pelo *proxy*.

***Proxy_Users_Buffer*:** implementa o *buffer* de saída da interface de rede com os clientes, armazenando os blocos de vídeo transmitidos pelo *proxy* aos clientes. O envio do bloco de vídeo é realizado através da cópia do conteúdo armazenado na memória, no endereço de mapeado a partir da tabela de mapeamento para o *buffer* de saída.

2.2 Estratégias de Substituição de Memória

A estratégia para *caching* de vídeo em um *proxy* de *VoD* envolve o estudo de técnicas eficazes para gerenciamento da memória a fim de elevar a utilização do conteúdo armazenado, reduzindo o tráfego de comunicação com o servidor principal. Entretanto, o projeto de um *proxy* de *VoD* pode variar significativamente em complexidade, o que acaba por produzir impactos variados sobre o desempenho e a escalabilidade do sistema. Deste modo, a manutenção dos vídeos mais populares na memória do *proxy* tende a ser, em geral, o requisito mais importante, uma vez que a preservação de vídeos de baixa popularidade na memória do *proxy* faz com que recorrentemente o algoritmo de substituição seja executado para liberar espaço para alocação dos novos blocos de vídeo provenientes do servidor principal. Assim, quanto maior for o número de vídeos de baixa popularidade mantidos em memória, maior será o número de acessos ao servidor para transferência de blocos de vídeo não armazenados pelo *proxy*.

Existem diversas estratégias que buscam capturar a popularidade dos vídeos em um sistema de *VoD* (LIU; XU, 2004)(WU; YU; WOLF, 2001)(ISHIKAWA, 2003)(HONG; VLEESCHAUWER; BACCELLI, 2010)(YU et al., 2006)(VENKATRAMANI et al., 2002). Em razão disto, faz-se necessária a escolha de modelos de gerenciamento de memória que sejam possíveis de serem implementados no período de realização deste trabalho, que sejam capazes de serem validados frente às limitações de projeto e execução, e que apresentem resultados significativos aos modelos de estado da arte.

A escolha dos sistemas propostos por (ISHIKAWA, 2003) (*Collapsed Cooperative Video Cache*) e por (HONG; VLEESCHAUWER; BACCELLI, 2010) (*Chunk-based Caching*) levou em consideração diferentes aspectos como recência da publicação, documentação detalhada e aprimoramento diferencial dos trabalhos em relação às outras publicações existentes. Em adição, é apresentada uma solução desenvolvida pelo grupo de pesquisa e desenvolvimento HECo do curso de Engenharia de Computação da Universidade Federal do Pampa.

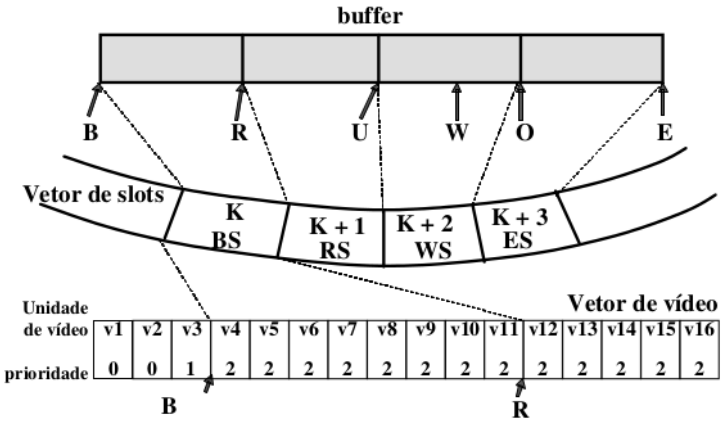
2.2.1 *Collapsed Cooperative Video Cache*

CCVC ou MCC é a estratégia proposta por (ISHIKAWA, 2003) na qual apresenta a arquitetura de um *proxy* de *VoD* inspirada no modelo de distribuição de vídeo baseado em *peer-to-peer*.

Como pode ser visto na Figura 2.9, o sistema mantém duas estruturas para cada vídeo armazenado na memória do sistema: o vetor de vídeo e o vetor de *slots*. O vetor de vídeo mapeia as unidades de vídeo que compõem um fluxo de vídeo completo sequencialmente. O vetor de

slots consiste de *slots* de tamanho fixo em unidades de tempo, que em conjunto compreendem toda a duração do vídeo. Estas duas estruturas possibilitam o gerenciamento entre os diversos *buffers* colapsados armazenados pelo sistema. Os buffers colapsados são *buffers* de vídeo dos usuários. Cada usuário possui um buffers colapsado armazenado em memória que pode ou não estar compartilhado com outros usuários.

Figura 2.9 – Projeção do *Ring buffer* sobre o vetor de vídeo



Fonte: Ishikawa, 2003, p. 42.

Os *buffers* de vídeo são compostos por unidades de vídeo (u.v.), onde esta representa um conjunto de frames que corresponde a menor quantidade de vídeo gerenciável pelo *proxy*. Para mapear as u.v na memória, cada unidade do vetor de vídeo contém um ponteiro para a localização da u.v. correspondente na memória, além de outras informações como o tamanho da u.v e um código de prioridade de descarte para uso do algoritmo de alocação de memória.

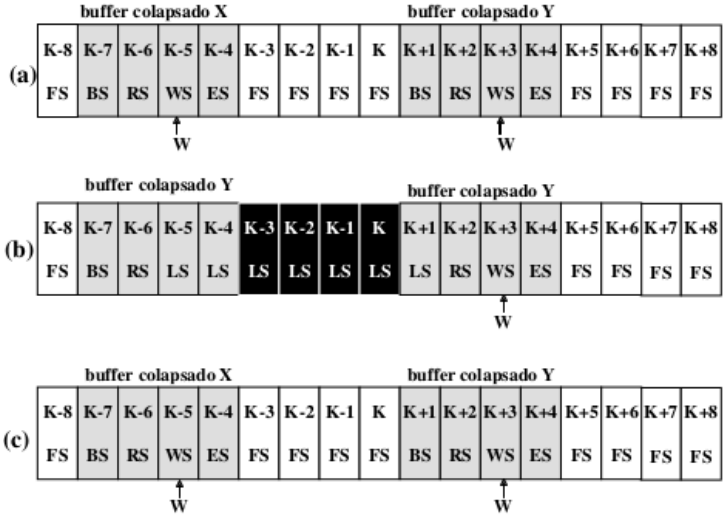
O vetor de *slots* serve para alocar *buffers* colapsados no *proxy*. Este pode ser dividido em 4 partes: BS, RS, WS e ES, correspondem respectivamente às áreas de leitura, risco de *underflow*, escrita e risco de *overflow*.

O vetor de *slots* gira de forma síncrona em sentido horário em relação ao vetor de vídeo na mesma velocidade da exibição do respectivo vídeo. À medida que gira, algumas unidades de vídeo entram e outras saem dos *ring buffers*. As que entram e têm seu conteúdo em memória são marcadas com prioridade 2 (a mais alta em todo sistema) e não podem ser descartadas. As que saem, passam a ter prioridade 1 (intermediária) e podem ser descartadas caso seja necessário. A prioridade 0 indica que a unidade de vídeo não está na memória.

O sistema implementa mecanismos para concatenação e divisão de *ring buffers* que possibilitam um controle de prioridade mais complexo. Deste modo, conforme mostra a Figura 2.10-a, se dois *ring buffers* estiverem separados temporalmente por alguns *slots* livres (FS), o

sistema converte estes *slots* em *slots* de ligação (LS), expandindo assim o *ring buffer* (Figura 2.10-b). A vantagem deste mecanismo é que o conteúdo potencialmente armazenado nos *slots* de ligação substituirá o fluxo regular de dados do servidor usado para abastecer a região de escrita WS do *ring buffer* temporalmente atrasado (a esquerda). Com isso, o ponteiro W de escrita do *ring buffer* a direita passará a alimentar todo o *ring buffer* expandido. Quando necessário, o sistema é capaz de resgatar algumas unidades de vídeo pertencentes aos *slots* de ligação para atender a uma nova demanda (Figura 2.10-c).

Figura 2.10 – Encadeamento de *buffers* colapsados



Fonte: Ishikawa, 2003, p. 47.

A estratégia leva em consideração a popularidade dos vídeos, assim como o tamanho das sequências de *slots*. Além disso, sequências de *slots* próximas do início do vídeo recebem prioridade menor de descarte, deste modo, tendem a permanecer alocadas em memória, acelerando a entrega do conteúdo aos clientes que requisitarem o vídeo. O mecanismo de resgate de unidades de vídeo possui uma complexidade NP-completa.

A função para o cálculo da prioridade de descarte de uma sequência S de *slots* de ligação é definida como:

$$F_s(d,f,t) = \frac{d}{min\{f,t\}} \tag{2.5}$$

Onde f é o número de *slots* que precisam ser liberados para alocar a demanda proveniente do servidor, t o tamanho da sequência de *slots* de ligação e d é a distância do início da sequência até o fim do vídeo .

O objetivo é minimizar a soma de F_s para as diversas sequências existentes. De acordo

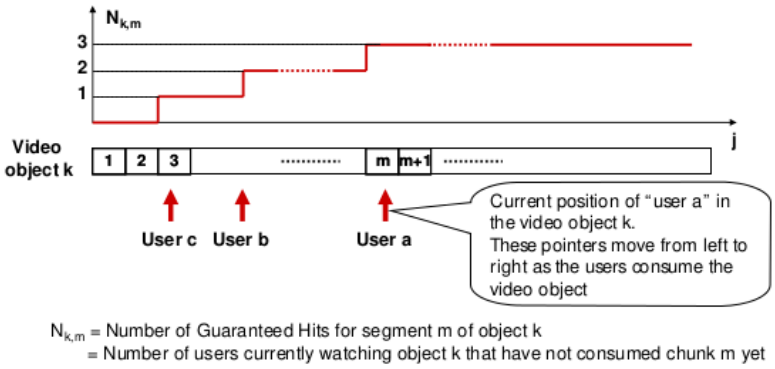
com o cálculo de F_s, quanto menor o valor obtido através da função maior a proximidade da sequência do final do vídeo o que leva a necessidade de reposição deste conteúdo por um fluxo regular do servidor por um período menor de tempo. Este mecanismo também foi projetado para não valorizar em excesso sequências muito longas, as quais podem estar distantes do final do vídeo. Estabelecendo como divisor o mínimo entre f e t, valoriza-se sequências que estejam próximas do fim e ao mesmo tempo possuam um tamanho maior ou igual a f.

2.2.2 *Chunk-based Caching*

Chunk-based Caching (CC), proposta por (HONG; VLEESCHAUWER; BACCELLI, 2010), apresenta o primeiro algoritmo a levar em consideração o número de clientes que estão acessando um vídeo do acervo para calcular a prioridade de descarte dos segmentos de vídeo em função da taxa de *hits* futura de cada segmento.

Como pode ser visto na Figura 2.11, a estratégia busca contabilizar os clientes que ainda não acessaram um segmento de vídeo como um valor de *hit* garantido, uma vez que os clientes tendem a seguir seu fluxo de execução, requisitando o segmento m, $m+1$, $m+2$, ...

Figura 2.11 – Disposição das prioridades de descarte dos segmentos de vídeo



Fonte: Hong, 2010, p. 3.

O sistema de prioridade de descarte (S_k) é baseado na contagem de requisições para cada vídeo. Cada vez que o vídeo k é requisitado S_k é incrementado por um valor A e todos os outros vídeo são decrementados em um valor B. Além disso, cada vídeo é dividido em M segmentos e cada segmento herda o valor S_k do vídeo ao qual pertence. Para cada segmento m do vídeo k, o valor $N_{k,m}$ acumula o número de *hits* garantidos para o segmento. Os valores $N_{k,m}$ são incrementados a cada nova requisição pelo vídeo k e o segmento $N_{k,m}$ é decrementado logo após o cliente consumir o segmento m.

Entretanto, este modelo desconsidera um limite temporal com respeito a distância entre

o ponteiro de leitura do cliente prévio e o *offset* de um segmento de vídeo particular, fazendo com que segmentos localizados no final do vídeo tenham um maior número de requisições associados aos mesmos, levando a permanência destes na memória do *proxy*.

Desta maneira, um ponto não considerado por Hong é a densidade de clientes localizada na proximidade do início de cada segmento de vídeo, o que pode justificar a preservação em memória dos blocos que, embora localizados no início do vídeo, possuem uma demanda maior associada a curto prazo.

2.2.3 *Current demAnd Rather Than futurE*

Neste trabalho é proposto um novo modelo denominado *CARTE*, cujo diferencial está no fato de que, em cenários de alta demanda, o *proxy* deveria realizar a melhor escolha para a utilização do seu conteúdo, considerando uma janela de tempo futura limitada por alguns segundos de execução, uma vez que, caso contrário, a reserva de recursos voltada à qualidade de serviço a longo prazo poderia comprometer a escalabilidade para manter a demanda atual. Em adição, diferentemente da *CCVC* que bloqueia os clientes ainda não admitidos pelo *proxy* baseado na indisponibilidade de recursos, o *CARTE* busca utilizar todo o recurso computacional disponível para servir o maior número de clientes possível. Consequentemente, quando necessário, alguns clientes podem ser redirecionados para um canal direto com o servidor principal, com o propósito de direcionar esforços para servir os clientes localizados em grupos que maximizam o rendimento do sistema a curto prazo.

Enquanto diversas estratégias reservam porções do vídeo com o intuito de prover uma melhor qualidade de serviço aos clientes futuros, onde em muitas vezes, chegam ao sistema muito tempo depois do atual tempo de execução, o *CARTE* busca utilizar de um melhor modo o conteúdo alocado para servir a real demanda a curto prazo existente. Além disso, a reserva de conteúdo pode levar a subutilização de recursos como memória e processador, uma vez que, em nenhum momento estes recursos são utilizados para vencer a demanda urgente pelo serviço de entrega. Idealmente, o processador deveria esforçar-se ao máximo para arranjar os fluxos de alocação de conteúdo para maximizar a produtividade do *proxy* (o que é proporcional ao número de clientes servidos pelo sistema), através do redirecionamento de requisições ao servidor principal e da implementação de uma política de alocação de memória baseada em prioridades.

A estratégia busca dividir o vídeo em blocos de vídeo, de maneira a gerenciar sequências de blocos de vídeo contíguas alocadas em memória. A prioridade de descarte, PS_s, de uma sequência de blocos de vídeo s é dada pela Equação (2.6).

$$PS_s = \frac{NC_s}{SZ_s} \times \frac{UI_s}{SZ_s} \tag{2.6}$$

Onde:

Tamanho da sequência (SZ_s): comprimento da sequência de blocos de vídeo s.

Número de clientes prévios (NC_s): total de clientes presentes dentro da janela de n blocos de vídeo precedentes à sequência s que potencialmente irão continuar a execução até a sequência alocada.

Fator de utilização (UI_s): representa o histório do número de blocos de vídeo acessados localizados dentro da sequência no tempo de execução atual.

O segundo elemento da Equação (2.6) (UI_s/SZ_s) somente é utilizado como fator de desempate entre sequências com a mesma prioridade de descarte. O valor de UI_s é calculado de acordo com a seguinte lógica:

início
  se $AEO_v < SP_s \;||\; ASO_v > EP_s$ **então**
    UIs = 0;
  senão
    se $ASO_v \geq SP_s$ **então**
      se $AEO_v > EP_s$ **então**
        $UI_s = EP_s - ASO_v$;
      senão
        $UIs = AEO_v - ASO_v$;
      fim
    senão
      se $AEO_v > EP_s$ **então**
        $UI_s = TS$;
      senão
        $UI_s = AEO_v - SP_s$;
      fim
    fim
  fim
fim

Algoritmo 1: Cálculo do Fator de Utilização (UI)

Tal que:

- SP_s corresponde ao *offset* do bloco inicial (*Starting Position*) da sequência s. Uma vez que as sequências são constantemente modificadas pelo deslocamento dos clientes no decorrer da linha de execução do vídeo a cada rodada de execução, faz necessário a atualização da posição da sequência de acordo com a localização dos fluxos de execução.
- EP_s representa o *offset* do bloco final (*Ending Position*) da sequência s, onde $EP_s = SP_s + SZ_s$.
- ASO_v informa, para um vídeo v, o *offset* médio do primeiro bloco de vídeo requisitado a partir do início do vídeo, calculado em função do ponto de entrada dos clientes, conforme a Equação (2.7). Onde TC_v corresponde ao número de clientes acessando o vídeo v e SO_{cv} corresponde ao *offset* de ponto entrada do cliente c que acessa o vídeo v ($SO_{cv} \geq 0$).
- Da mesma forma, a Equação (2.8) apresenta AEO_v que representa ao *offset* médio do último bloco de vídeo requisitado pelos clientes. EO_{cv} corresponde ao *offset* de ponto saída do cliente c que acessa o vídeo v (tal que $SO_{cv} \leq EO_{cv}$).

$$ASO_v = \sum_{1}^{TC_v} SO_{CV} \tag{2.7}$$

$$AEO_v = \sum_{1}^{TC_v} EO_{CV} \tag{2.8}$$

O algoritmo busca preservar em memória as sequências de blocos de vídeo com a maior demanda local em um curto período de tempo (representada pelo número de clientes existente dentro da janela de n blocos precedentes à sequência) e, ao mesmo tempo, favorece a preservação de pequenas sequências, uma vez que esta escolha permite uma melhor escalabilidade devido ao menor consumo de recursos de memória.

2.3 Módulos de Programa

O *proxy* tem como função principal realizar o atendimento às requisições de vídeo emitidas pelos clientes *VoD*, através do gerenciamento da entrada e saída dos blocos de vídeo em sua memória. Conforme apresentado na Figura 2.2, diferentes módulos de programa são necessários para dar suporte a esta funcionalidade principal, implementando operações específicas como a administração do envio de conteúdo aos usuários, a alocação e substituição de blocos de vídeo em memória, requisição dos blocos de vídeo ao servidor principal, o controle

da popularidade do acervo de vídeos, a admissão e rejeição de novos clientes e a finalização das seções de vídeo. Os principais módulos de programa implementados são apresentados a seguir:

Módulo de admissão de novos clientes: é responsável por verificar a existência de novas requisições de clientes no *buffer New_Users_Array* durante o início de cada rodada de serviço e realizar o controle de admissão destes clientes sob a condição de que existam recursos disponíveis, de maneira que seja possível criar e manter novos fluxos de execução e de alocação caso sejam admitidos.

Módulo de gerenciamento de estruturas internas: uma vez que novos clientes são adicionados ao *proxy* e outros finalizam suas execuções, é necessário atualizar as estruturas internas que identificam a localização dos clientes, gerenciar os fluxos de alocação, calcular a popularidade do acervo e a distribuição dos blocos de vídeo.

Módulo de verificação dos blocos de vídeo: o módulo executa os fluxos de alocação ativos existentes na *Active_Writers_List*, através da análise da tabela de mapeamento de blocos vídeo (*Blocks_Addresses_Table*), verificando se o conteúdo do vídeo requisitado encontra-se alocado em memória.

Módulo de requisição de blocos de vídeo ao servidor: caso o conteúdo requisitado pelos clientes não esteja alocado em memória, o módulo realiza o pedido dos blocos de vídeo ao servidor principal, através da escrita no *buffer Requisitions_Array*.

Módulo de alocação de blocos de vídeo: é responsável por alocar na memória do *proxy* o conteúdo requisitado ao servidor principal, disponível no *buffer Server_Proxy_Buffer*. Caso não existam endereços de memória disponíveis para serem alocados, o módulo de substituição de blocos de vídeo é executado.

Módulo de substituição de blocos de vídeo: o módulo analisa a distribuição dos usuários através dos vídeos e dos blocos de vídeo que estão alocados em memória e disponíveis a serem substituídos, a fim de reciclar a memória e remover o conteúdo com a maior prioridade de descarte, liberando espaço para a alocação de novos blocos de vídeo.

Módulo de transmissão de blocos de vídeo aos clientes: realiza o envio do conteúdo armazenado na memória do *proxy* aos clientes ativos através da escrita dos blocos de vídeo no *buffer Proxy_Users _Buffer*.

***Timer*:** responsável por realizar o sincronísmo do sistema, coordenando a execução das rodadas de serviço levando em consideração o tempo necessário para servir os clientes ativos,

alocar o conteúdo requisitado e gerenciar as estruturas internas. Para manter a compatibilidade com a taxa de codificação do vídeo, faz-se necessário a entrega dos blocos de vídeo requisitados dentro de um intervalo de tempo inferior a um segundo, ou seja, a duração máxima de cada rodada de serviço não deve ser superior a um segundo. Todavia, para fins de análise do desempenho do *proxy*, a próxima rodada de serviço não é inicializada enquanto a rodada corrente não for concluída. Isto possibilita que o tempo total necessário para completar cada rodada possa ser mensurado sem interferência entre as rodadas de serviço, o que viabiliza identificar precisamente quais condições de carga foram decisivas para a perda dos prazos de entrega.

2.4 Rotina de Serviço

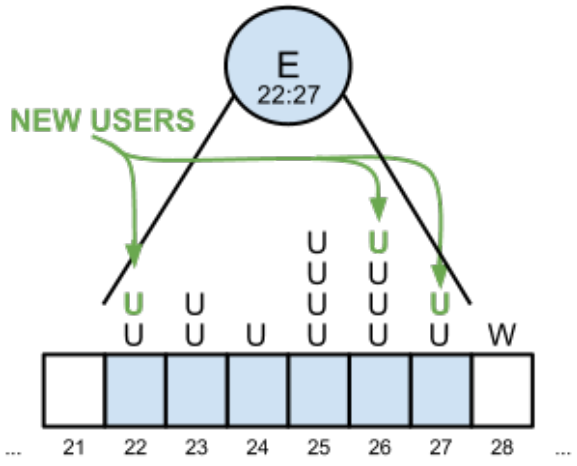
Abaixo é apresentada a rotina de execução do *proxy* implementado, exibindo passo-a-passo o gerenciamento do sistema, o fluxo de comunicação entre os módulos internos e a manipulação das estruturas de dados.

O gerenciamento do *proxy*, preliminarmente a qualquer tarefa, captura o valor do relógio da máquina hospedeira, registrando assim o instante de início de execução da rodada de serviço. Em seguida, verifica a existência de novas requisições no *buffer New_Users_Array*. Cada nova requisição existente representa um novo cliente (*User*) que chegou ao sistema. Caso o conteúdo do *buffer* seja diferente de nulo, então o módulo de admissão identifica a qual vídeo o novo cliente pertence. Em seguida, verifica se o vídeo requisitado já se encontra mapeado para uma posição na tabela de mapeamento de blocos de vídeo (*Blocks_Addresses_Table*). Se o valor de mapeamento for válido ($Video \rightarrow index \geq 0$), então o módulo verifica se é possível admitir o cliente tendo em vista a existência de recursos disponíveis. Senão, verifica se o número máximo de vídeos ativos servidos pelo *proxy* não foi ultrapassado, adiciona uma posição de mapeamento na tabela de mapeamento para o novo vídeo e incrementa o número de vídeos ativos. Caso o número máximo de vídeos ativos seja extrapolado, o novo cliente é descartado.

A admissão do novo cliente está condicionada a existência de recursos disponíveis ao *proxy*, ou seja, é necessário verificar de acordo com o ponto de entrada do novo cliente e das estruturas internas já existentes (*UsersGroups* e *BlockIntervals*), se será necessário criar um novo grupo de usuários e/ou instanciar um novo fluxo de alocação. Assim, verifica-se o ponto de entrada do novo cliente ($User \rightarrow current\_block$) no vídeo em relação aos grupos de clientes ativos contiguos já existentes. As possibilidades da localização do novo cliente e as decisões tomadas para tratar a admissão do mesmo são apresentadas a seguir:

Cliente localizado no interior de um *UsersGroup*: ou seja, *UsersGroup*→*ini_block* ≤ *User*→*current_block* ≤ *UsersGroup*→*end_block* [Figura 2.12]. Desta forma, o conteúdo requisitado já se encontra alocado em memória ou a inserção de um novo cliente previamente já realizou a instanciação de um novo fluxo de alocação (*Writer*) para o bloco de vídeo requisitado. Tendo em vista que não houve nenhuma modificação dos limites dos elementos *UsersGroup* e *BlockInterval* existentes, a estrutura disposta em árvore (*Active_Structures_Tree*) permanece intacta e o número de blocos de vídeo reservados pelo *proxy* não será alterado. Desta maneira, o novo cliente pode ser admitido, restando apenas adicioná-lo à lista de clientes ativos do grupo de usuários, incrementar o número de clientes admitidos (*num_clients_admitted*++), o número de clientes ativos do vídeo (*Video*→*num_active_clients*++), o número de clientes pertencentes ao *UsersGroup* (*UsersGroup*→*weight*++) e a prioridade do fluxo de alocação pertencente ao grupo de usuários (*UsersGroup*→*Writer*→*priority*++).

Figura 2.12 – Novo cliente localizado no interior de um *UsersGroup* existente.



Fonte: Arquivo pessoal.

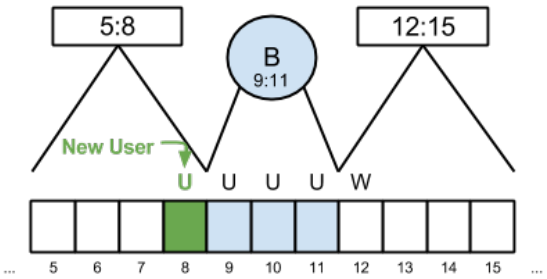
Cliente localizado em um bloco de vídeo contíguo à esquerda de um *UsersGroup*: a nova requisição esta localizada no bloco de vídeo à esquerda do limite inferior de um *UsersGroup*, tal que (*User*→*current_block* == *UsersGroup*→*ini_block*−*1*). Existem dois possíveis casos a serem tratados:

Bloco de vídeo requisitado alocado: se o bloco de vídeo requisitado, contíguo ao *UsersGroup*, está alocado em memória [Figura 2.13a], então o novo cliente após deslocar seu fluxo de execução permanecerá contíguo ao *UsersGroup* e extenderá o limite inferior do grupo [Figura 2.13b]. Assim, se o novo cliente for admitido então o número de blocos de vídeo reservados em memória aumentará. Desta maneira, é necessário

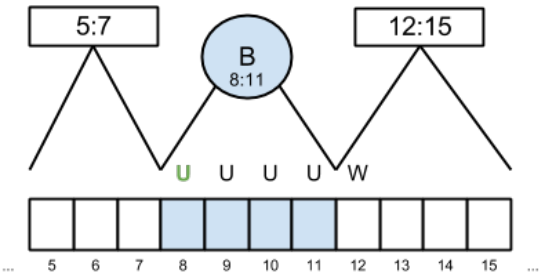
verificar se o *proxy* possui recursos disponíveis ($num\_available\_addresses > 0$) e então decrementar o número de endereços de memória disponíveis ($num\_available\_addresses--$), adicioná-lo à lista de clientes ativos do grupo de usuários, incrementar o número de clientes admitidos ($num\_clients\_admitted++$), o número de clientes ativos ($Video \rightarrow num\_active\_clients++$), o número de blocos de vídeo reservados ($Video \rightarrow num\_reserved\_blocks++$), o comprimento e o número de clientes pertencentes ao *UsersGroup* ($UsersGroup \rightarrow length++$ e $UsersGroup \rightarrow weight++$), a prioridade do fluxo de alocação pertencente ao *UsersGroup* ($UsersGroup \rightarrow Writer \rightarrow priority++$) e atualizar o limite superior do *BlockInterval* anterior e o limite inferior do *UsersGroup*. Caso contrário, rejeita-se o novo cliente e incrementa-se o número de clientes rejeitados ($num\_clients\_rejected++$).

Figura 2.13 – O novo cliente, localizado à esquerda de um *UsersGroup*, extende o limite inferior do *UsersGroup* existente.

(a) Bloco de vídeo requisitado, contíguo ao limite inferior do *UsersGroup*, está alocado.



(b) Novo cliente extende o limite inferior do *UsersGroup* existente.



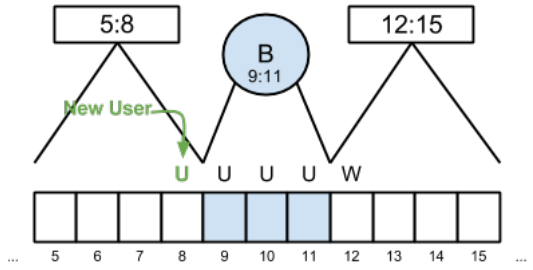
Fonte: Arquivo pessoal.

Bloco de vídeo requisitado não alocado: se o bloco de vídeo requisitado, contíguo ao *UsersGroup*, não estiver alocado em memória [Figura 2.14a] então não é garantido que o novo cliente permanecerá contíguo ao grupo, pois, este terá que aguardar pelo menos uma rodada de serviço até que um *Writer* aloque o bloco de vídeo requisitado [Figura 2.14c]. Desta maneira, será necessário criar um novo *UsersGroup* [Figura 2.14b], causando o aumento do número de endereços de memória reservados. Logo, se o *proxy* possuir recursos disponíveis, então admite-se o novo cliente, decrementa-se o número de endereços de memória disponíveis, incrementa-se o número de clientes admitidos, o número de clientes ativos, o número de blocos de vídeo reservados, o número de grupos de usuários ativos ($Video \rightarrow num\_active\_usersgroups++$), o número de fluxos de alocação ativos ($Video \rightarrow num\_active\_writers++$), cria-se um novo grupo de usuários, um novo fluxo de alocação e um *BlockInterval* intermediário aos dois *UsersGroups*. Caso contrário, rejeita-se o novo cliente e incrementa-se

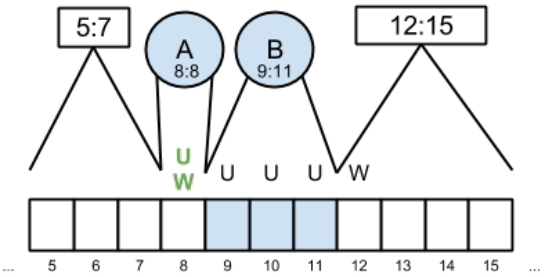
o número de clientes rejeitados.

Figura 2.14 – O novo cliente, localizado à esquerda de um *UsersGroup*, não permanece contíguo e cria um novo grupo de usuários.

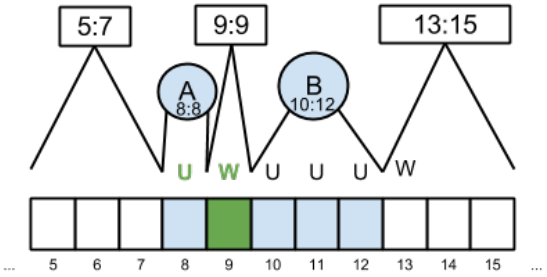
(a) Bloco de vídeo requisitado, contíguo ao limite inferior do *UsersGroup*, não está alocado.



(b) Novo cliente cria um novo *UsersGroup*, pois, não é garantido que o mesmo permanecerá contíguo ao *UsersGroup* existente.



(c) Novo cliente aguarda uma rodada enquanto o *Writer* aloca o bloco de vídeo requisitado.



Fonte: Arquivo pessoal.

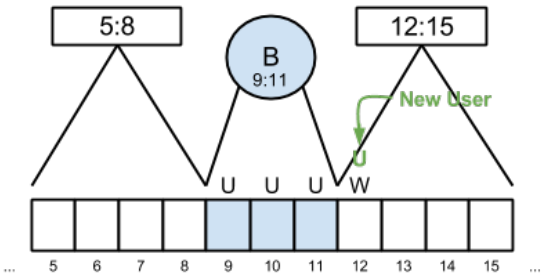
Cliente localizado em um bloco de vídeo contíguo à direita de um *UsersGroup*: o novo cliente está localizado no bloco de video contíguo à direta do limite superior de um *UsersGroup* (*User→current_block == UsersGroup→end_block+1*). Existem dois possíveis casos a serem tratados:

***UsersGroup* anterior desloca-se:** se o bloco de vídeo do limite superior do *UsersGroup* estiver alocado em memória [Figura 2.15a], então o grupo de clientes anterior ao novo cliente irá avançar e manter-se-á contíguo ao novo cliente, fazendo com que o mesmo extenda o limite superior do grupo [Figura 2.15b]. Assim, se o novo cliente for admitido então o número de blocos de vídeo reservados em memória aumentará. Desta maneira, é necessário verificar se o *proxy* pode ostentar o novo fluxo de execução e então decrementar o número de endereços de memória disponíveis, adicioná-lo à lista de clientes ativos do grupo de usuários, incrementar o número

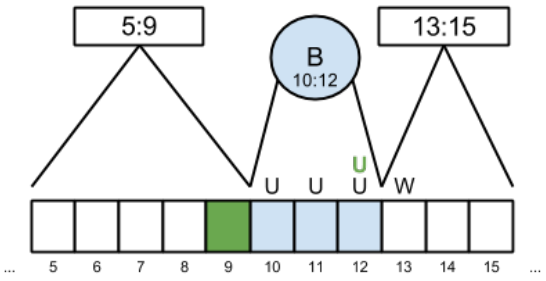
de clientes admitidos, o número de clientes ativos, o número de blocos de vídeo reservados, o comprimento e o número de clientes pertencentes ao *UsersGroup*, a prioridade do fluxo de alocação pertencente ao *UsersGroup* e atualizar o limite inferior do *BlockInterval* posterior e o limite superior do *UsersGroup*. Caso contrário, rejeita-se o novo cliente e incrementa-se o número de clientes rejeitados.

Figura 2.15 – O novo cliente, localizado à direita de um *UsersGroup*, extende o *UsersGroup* existente.

(a) *UsersGroup* anterior possui todos os blocos internos alocados, logo, irá avançar o fluxo de execução de todos os clientes internos.



(b) Novo cliente extende o limite superior do *UsersGroup* existente.



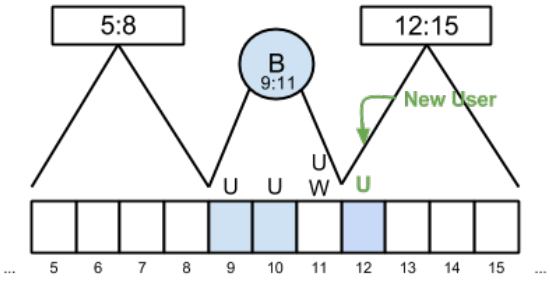
Fonte: Arquivo pessoal.

***UsersGroup* anterior não se desloca:** se o bloco de vídeo do limite superior do *UsersGroup* não estiver alocado em memória [Figura 2.16a], então o grupo de usuários anterior ao novo cliente não irá avançar [Figura 2.16c], aguardando até que o *Writer* aloque o conteúdo, não havendo garantia de que o novo cliente permancerá contíguo ao grupo de usuários. Desta maneira, será necessário criar um novo *UsersGroup* [Figura 2.16b], causando o aumento do número de endereços de memória reservados. Logo, se o *proxy* possuir recursos disponíveis, então admite-se o novo cliente, decrementa-se o número de endereços de memória disponíveis, incrementa-se o número de clientes admitidos, o número de clientes ativos, o número de blocos de vídeo reservados, o número de grupos de usuários ativos, o número de fluxos de alocação ativos, cria-se um novo grupo de usuários, um novo fluxo de alocação e um *BlockInterval* intermediário aos dois *UsersGroups*. Caso contrário, rejeita-se o novo cliente e incrementa-se o número de clientes rejeitados.

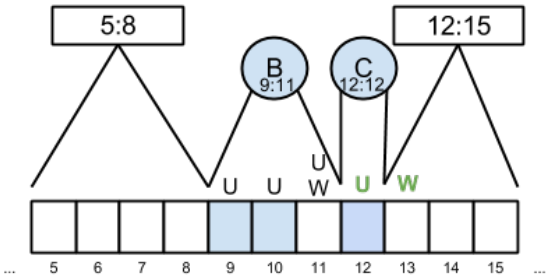
Cliente localizado entre dois *UsersGroups* separados por um bloco de vídeo: a admissão do novo cliente neste caso, está sujeita ao conjunto dos processos anteriores (admissão de um cliente contíguo aos limites de um *UsersGroup*), com a adição do caso unificado onde, se ambos os grupos de usuários deslocarem-se e o bloco requistado pelo novo cliente estiver alocado em memória [2.17a], fazendo com que o novo cliente também desloque-se, então

Figura 2.16 – O novo cliente, localizado à direita de um *UsersGroup*, não permanece contíguo e cria um novo grupo de usuários.

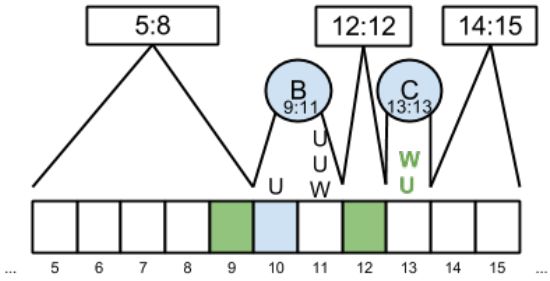
(a) *UsersGroup* anterior não possui o bloco interno superior alocado, logo, não irá deslocar o limite superior.



(b) Novo cliente cria um novo *UsersGroup*, pois, não é garantido que o mesmo permanecerá contíguo ao *UsersGroup* existente.



(c) *UsersGroup* anterior não possui o bloco interno superior alocado, logo, não irá deslocar o limite superior.



Fonte: Arquivo pessoal.

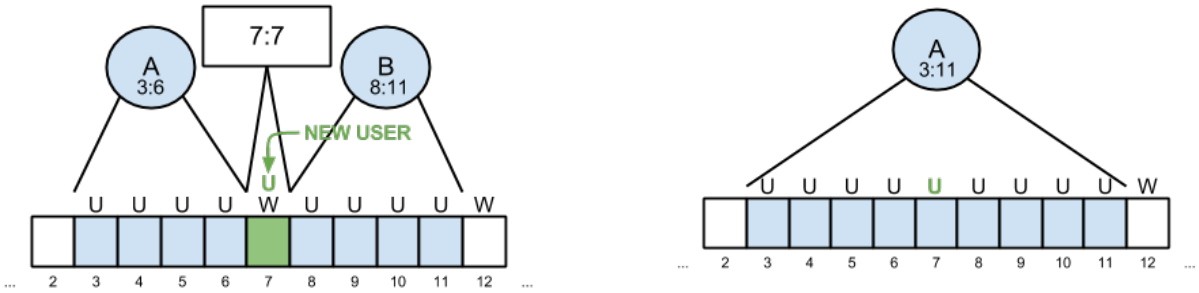
o *BlockInterval* existente entre os *UsersGroups* será excluído e os grupos de usuários serão unidos, como mostra a Figura 2.17b. Deste modo, se o *proxy* possuir recursos disponíveis, então deve-se admitir o novo cliente, decrementar o número de endereços de memória disponíveis, o número de fluxos de alocação ativos (*Video→num_active_writers−−*), o número de grupos de usuários ativos (*Video→num_active_usersgroups−−*), incrementar o número de clientes admitidos, o número de clientes ativos, o número de blocos de vídeo reservados, remover o *BlockInterval* intermediário e o *UsersGroup* que se encontra mais longe da raiz da *Active_Structures_Tree*, atualizar as informações do outro grupo de usuários, acumulando os dados de ambos os grupos. Como consequência, o fluxo de alocação que servia o grupo de clientes anterior é removido, tendo em vista a existência de um fluxo de alocação mais a frente da linha de execução que proverá o conteúdo requisitado a todos os fluxos de execução pertencentes ao novo e maior *UsersGroup*. Caso contrário, rejeita-se o novo cliente e incrementa-se o número de clientes rejeitados.

Cliente localizado no interior de um *BlockInterval*: neste caso não exitem *UsersGroups* com

Figura 2.17 – Exemplo de nova requisição localizada entre dois *UsersGroups* separados por um bloco de vídeo.

(a) Exemplo de nova requisição entre dois *UsersGroups* separados por um bloco de vídeo.

(b) União dos *UsersGroups* previamentes separados.



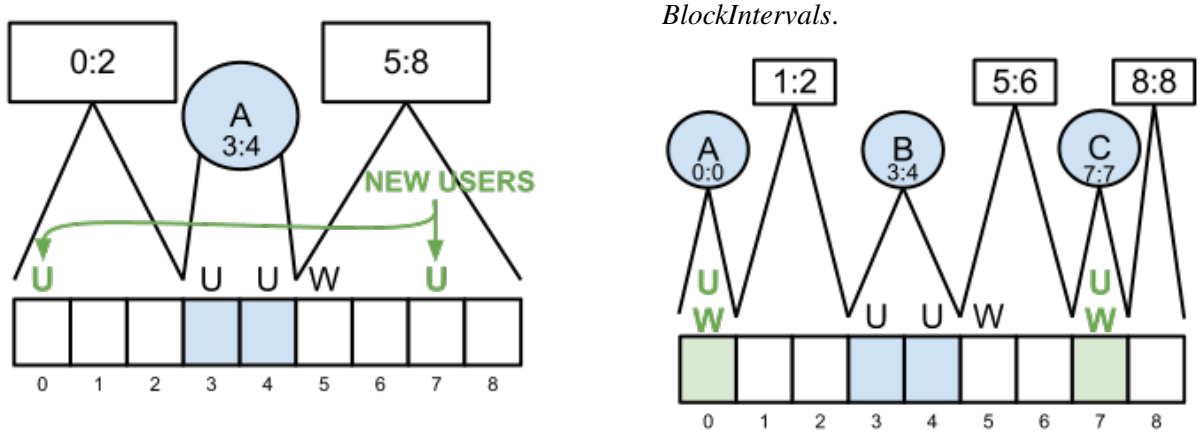
Fonte: Arquivo pessoal.

limites contíguos ao novo fluxo de execução [2.18a]. Assim, faz-se necessário a criação de um novo *UsersGroup* e *BlockInterval*. Novamente, verifica-se se o *proxy* possui recursos disponíveis, então admite-se o novo cliente, decrementa-se o número de endereços de memória disponíveis, incrementa-se o número de clientes admitidos, o número de clientes ativos, o número de blocos de vídeo reservados, o número de grupos de usuários ativos, o número de fluxos de alocação ativos, cria-se um novo grupo de usuários, um novo fluxo de alocação e um *BlockInterval* [2.18b]. Caso contrário, rejeita-se o novo cliente e incrementa-se o número de clientes rejeitados.

Figura 2.18 – Exemplo de novas requisições não contíguas a um *UsersGroup*.

(a) Admissão de novos fluxos de execução.

(b) Atualização das estruturas internas. Criação de novos *UsersGroups* e *BlockIntervals*.



Fonte: Arquivo pessoal.

Uma vez adicionados os novos clientes ao sistema, deve-se atualizar o vetor de popularidade dos vídeos (*Videos_Popularity_Array*) e executar os fluxos de execução dos clientes.

Assim, o módulo tem acesso à lista de grupos de usuários ativos (*Active_UsersGroups_List*) e realiza a execução de cada um dos fluxos de execução de cada grupo. Primeiramente, salva-se os *offsets* de início e fim do *UsersGroup* prévio a execução do grupo para que posteriormente possa ser verificado se o mesmo deslocou-se no decorrer do vídeo. Se o bloco de vídeo requisitado pelo cliente estiver alocado em memória então deve-se incrementar o número de *hits* de memória (*num_memory_hits*++) e transferir o conteúdo ao cliente. A transferência é realizada através da cópia do conteúdo da memória endereçado pela tabela de mapeamento de blocos para o *buffer Proxy_Users_Buffer*. Ao concluir a transferência, o ponteiro de bloco de leitura corrente do usuário é incrementado (*User*→*current_block*++), proporcionando o avanço do fluxo de execução sobre o vídeo. Caso contrário, se o conteúdo não estiver alocado em memória, então incrementa-se o número de *misses* de memória (*num_memory_misses*++) e o cliente aguarda no bloco corrente até que o mesmo seja alocado pelo fluxo de alocação pertencente ao *UsersGroup*.

Se o cliente finalizou a sua execução, transferindo o último bloco de vídeo do vídeo requisitado, então deve-se removê-lo do *UsersGroup* ao qual pertence e decrementar o número de clientes pertencentes ao grupo (*UsersGroup*→*weight*−−) e o número de usuários ativos do vídeo (*Video*→*num_active_users*−−). Caso um grupo de usuários deixe de existir (*UsersGroup*→*weight*==*0*) então remove-se o grupo de usuários da *Active_UsersGroups_List* e da *Active_Strutuctures_Tree* e decrementa-se o número de grupos de usuários pertencentes ao vídeo (*Video*→*num_active_usersgroups*−−).

Após executar todos os clientes pertencentes ao *UsersGroup*, obtém-se os novos *offsets* de início e fim do grupo de usuários. Com posse dos limites do grupo pré e pós-execução, é possível verificar se o grupo deslocou-se através do vídeo, se seu comprimento foi modificado (caso usuários não tenham encontrado o bloco de vídeo alocado em memória e tenham que aguardar, reduzindo ou aumentando o tamanho do *UsersGroup*) e se o *UsersGroup* anterior sobrescreveu ou tornou-se contíguo ao *UsersGroup* atual (caso este não tenha se deslocado). Assim, a rotina de atualização de *UsersGroups* e *BlockIntervals* e de suas sequências de blocos de vídeo alocadas é executada.

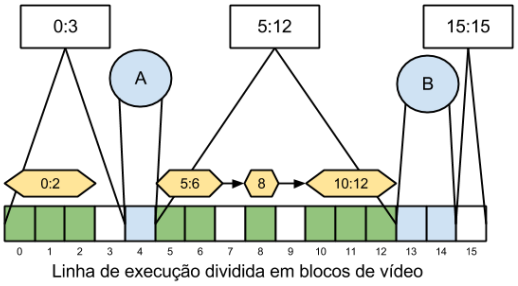
A rotina de atualização de *UsersGroups* e de *BlockIntervals* busca inicialmente atualizar as sequências de blocos de vídeo alocadas em memória pertencentes ao *BlockInterval* precedente ao *UsersGroup* que acabou de ser executado. Primeiramente, verifica-se se as sequências de blocos de vídeo existentes foram sobrescritas pelo deslocamento do *UsersGroup* anterior (*PrevUsersGroup*→*end_block* >= *BlockInterval*→Sequence→ini_block), atualizando o limite da sequência sobrescrita ou removendo a mesma (caso se tratasse de uma sequência de com-

primento unitário, cuja redução do comprimento faz com que a mesma deixe de existir). Em seguida, a rotina verifica se o *UsersGroup* atual deslocou e deixou um bloco de vídeo alocado como rastro. Nesse caso deve-se verificar se existia uma sequência de blocos de vídeo contígua ao bloco de vídeo deixado como rastro (sequência de blocos de vídeo <10:12> pertencente ao *BlockInterval* {6:13} da Figura 2.19b), e então atualizar seus limites [2.19c], ou se deve ser criado uma nova sequência correspondendo apenas ao bloco de vídeo deixado como rastro (sequência de blocos de vídeo <4> do *BlockInterval* {0:4} na 2.19c). Então, a rotina verifica se os *UsersGroups* atual e anterior tiveram seus limites sobrescritos ou tornaram-se contíguos, ocasionando a união dos dois grupos de usuários. A união busca manter na *Active_Structures_Tree* o *UsersGroup* mais próximo da raiz (menor valor de *UsersGroup→height*), assim, todos os usuários pertencentes ao grupo de usuários mais distante são movidos, os *offsets* dos limites, o número de clientes e a prioridade do fluxo de alocação pertencentes ao *UsersGroup* são atualizados. Por último, a rotina atualiza as prioridades de descarte das sequências de blocos de vídeo alocadas em memória do *BlockInterval* anterior de acordo com a estratégia de substituição definida. Se *CARTE* ou *CC* forem definidos, então calcula-se o número de clientes prévios a cada sequência. Senão, se *CCVC* foi definido então atualiza-se apenas as sequências que tiveram seus tamanhos ou *offsets* de início alteradas.

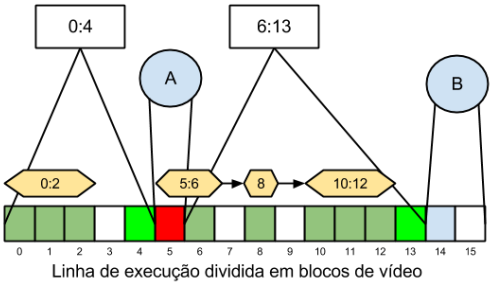
Após executar todos os *UsersGroups* ativos, o *proxy* inicia a execução de todos os fluxos de alocação ativos (*Writers*) em ordem de maior à menor prioridade de requisição contidos na lista de fluxos de alocação ativos (*Active_Writers_List*). Primeiramente é verificado se o conteúdo requisitado encontra-se alocado em memória através da leitura do elemento correspondente ao bloco de vídeo na tabela de mapeamento de blocos de vídeo ($addr = Blocks\_Addresses\_Table$ [*Writer→video→index*][*Writer→current_block*]). Se $addr \geq 0$, então o conteúdo requisitado encontra-se em memória e pode ser transferido aos clientes, restando apenas incrementar o ponteiro de alocação do *Writer* (*Writer→current_block*++), deslocando o fluxo de alocação. Caso contrário, o *proxy* requisita o bloco de vídeo ao servidor através da escrita no *buffer* de saída da interface de rede com o servidor (*Requisitions_Array*), informando a *ID* do vídeo e o *offset* do bloco de vídeo correspondente. Se o limite de banda de comunicação entre o servidor e o *proxy* foi ativado (*#define LIMITED_SERVER_COMMUNICATION*), cuja função é regular o tráfego de dados entre as entidades e consecutivamente, o número de blocos de vídeo enviados, é necessário verificar se o conteúdo requisitado foi transmitido pelo servidor. Caso o bloco de vídeo tenha sido enviado, o mesmo será armazenado no *buffer Server_Proxy_Buffer* e deverá ser repassado ao módulo de alocação em memória. Em adição, incrementa-se o número de requisições atendidas (*num_served_requisitions*++). Senão, incrementa-se o númro de requisições rejeitadas pelo servidor (*num_rejected_requisitions*++) e o fluxo de alocação permanece

Figura 2.19 – Exemplo de atualização de sequências de blocos de vídeo.

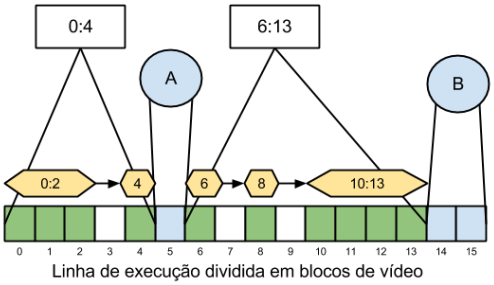
(a) Estado inicial dos *UsersGroups*.



(b) *UsersGroups* deslocam-se, deixando para trás blocos de vídeo alocados como rastro.



(c) Atualização das sequência de blocos de vídeo através da atualização do limite superior da sequência contígua e da criação de uma nova sequência contendo o bloco de vídeo deixado como rastro.



Fonte: Arquivo pessoal.

no bloco corrente até que seja possível alocar o conteúdo em uma próxima rodada de serviço.

O módulo de alocação de blocos de vídeo verifica se existem endereços de memória livres que possam ser utilizados para alocar os novos blocos de vídeo requisitados. Se $num\_free\_addresses \geq 0$, então existe pelo menos um endereço livre que pode ser utilizado. Assim, o *proxy* remove um endereço de memória a partir do vetor de endereços livres ($addr = Free\_Addresses\_Array[num\_free\_addresses]$), decrementa o número de endereços livres ($num\_free\_addresses--$), aloca o bloco de vídeo, atualiza a tabela de mapeamento de blocos de vídeo ($Blocks\_Addresses\_Table[Writer \rightarrow video \rightarrow index][Writer \rightarrow current\_block] = addr$). Caso contrário, é necessário executar o módulo de substituição de blocos de vídeo em memória, na tentativa de remover algum conteúdo da memória e assim, liberar espaço para alocar novos blocos de vídeo.

A remoção de blocos de vídeo da memória leva em consideração, em um primeiro momento, a popularidade dos vídeos obtida através do vetor de popularidade de vídeos (*Vi-*

deos_Popularity_ Array). À medida que os novos clientes são admitidos ao sistema, o módulo de admissão de novas requisições atualiza a popularidade do vídeo requisitado para refletir a quantidade de clientes que assistem ao vídeo. Assim, quanto maior o número de fluxos de execução ativos em um determinado vídeo, menor a probabilidade deste vídeo ter algum dos seus blocos descartado. Em seguida, o *proxy* acessa a lista de *Sequences* alocadas e disponíveis a serem removidas da memória (*Sequences_Priority_List*) na tentativa de desalocar os blocos de vídeo com as maiores prioridades de substituição. Se houver uma sequência de blocos de vídeo disponível, então o *proxy* deve adicionar os endereços de memória do conteúdo a ser removido ao vetor de endereços de memória livres, incrementar o valor de *num_free_addresses* de acordo com o tamanho da sequência, cancelar os valores de mapeamento na *Blocks_Addresses_Table* dos blocos de vídeo removidos e incrementar o número de substituições em memória realizadas (*num_memory_substitutions*++). Em seguida, o novo conteúdo é alocado em memória utilizando um dos endereços liberados. Caso não existam *Sequences* disponíveis no interior do vídeo selecionado, o vídeo com a popularidade imediatamente superior é selecionado, repetindo-se o processo através de todos os vídeos ativos. Perante a possibilidade de que nenhum bloco de vídeo possa ter sido descartado da memória para liberar espaço para conteúdo a ser alocado, o *proxy* contabiliza um erro de alocação de conteúdo (*num_memory_allocation_errors*++), o que acarreta no descarte do bloco de vídeo requisitado e a permanência do *Writer* no bloco corrente até que seja possível alocar o conteúdo em uma próxima rodada de serviço.

Ao concluir o processo de alocação com êxito, o módulo incrementa o ponteiro de bloco corrente do *Writer* (*Writer*→*current_block*++), deslocando o fluxo de alocação. Se o *Writer* chegou ao fim do vídeo, remove-se o mesmo da lista de fluxos de alocação ativos e decrementa-se o número de *Writers* ativos (*Video*→*num_active_writers*−−).

Finalmente, o *proxy* captura novamente o valor do relógio da máquina hospedeira, registrando o instante de conclusão da rodada de serviço. Através da diferença entre os valores de relógio obtidos (pré e pós à execução) é possível calcular o tempo de execuçao da rodada de serviço. Se o tempo consumido for inferior a um segundo, o *proxy* adormece pelo tempo restante até que se complete um segundo. Por outro lado, quando o tempo de serviço for maior ou igual a um segundo, o *proxy* não adormece e executa prontamente a próxima rodada de serviço. Quando o tempo de serviço da rodada é superior a um segundo então o *proxy* não foi capaz de atender a todos os clientes ativos do sistema dentro do prazo, ou seja, não foi possível transferir um bloco de vídeo a cada segundo a todos os fluxos de execução, o que pode levar a interrupção momentânea do fluxo de exibição do vídeo por parte do cliente. Para amenizar tais efeitos, normalmente são implementados *buffers* do lado do cliente.

Ao final da rodada de serviço, o *proxy* verifica se o número de rodadas a serem executadas foi vencido ($service\_round == max\_service\_rounds$) e encerra a execução. Caso contrário, incrementa-se o contador de rodadas de serviço ($service\_round$++) e inicia-se uma nova rodada.

3 Proxy de VoD Paralelo

Tendo em vista a alta carga de execução a qual os *proxies* de *VoD* são submetidos, podemos facilmente imaginar que o desempenho de um software sequencial e uniprocessado não será alto o suficiente para atender à crescente demanda de requisições em horários de pico do serviço, principalmente para vídeos de alta definição. Com esse intuito, implementamos um modelo *multithreaded*, permitindo assim, distribuir a carga de execução sobre diferentes *threads* de processamento e, consequentemente, sobre os diferentes núcleos do processador da arquitetura utilizada. Para chegar ao modelo implementado, buscamos identificar os módulo internos do *proxy* que exigiam uma maior parcela do processamento para gerenciar o sistema e, consecutivamente, uma maior parcela do tempo de serviço do *proxy*.

A estratégia de paralelização implementada é apresentada em 3.1, descrevendo as modificações e soluções desenvolvidas para dar suporte a nova arquitetura. Em 3.2 e 3.3 são descritos, respectivamente, o algoritmo de distribuição de carga e o algoritmo de distribuição de memória.

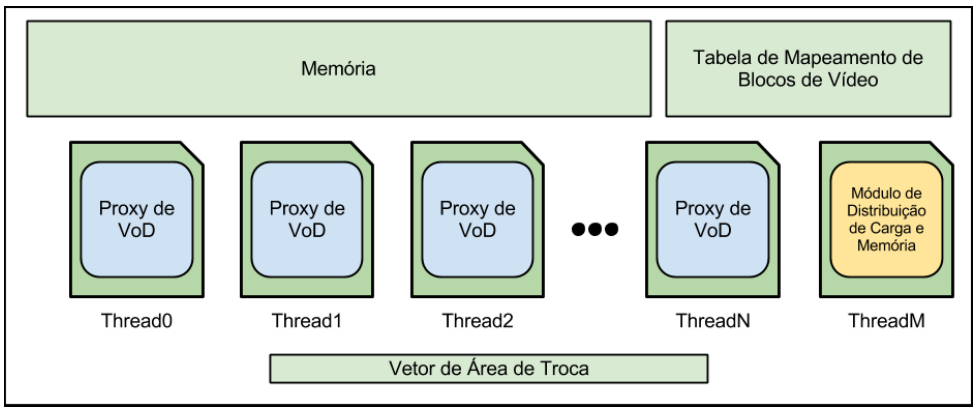
3.1 Estratégia de paralelização

Em um primeiro estudo, concluímos que a construção do modelo paralelo através do mapeamento de clientes para *threads* individuais seria algo inviável, uma vez que com o crescimento do número de requisições, o número de *threads* criadas seria muito alto, o que aumentaria a concorrência pela *CPU* e a intensidade das trocas de contexto, ocasionando a redução do desempenho. Da mesma forma, a criação de *threads* para servir os clientes ativos de cada vídeo também se torna inviável para um acervo de vídeos numeroso. Adicionalmente, buscamos desmembrar os módulos internos do *proxy*, de maneira que pudéssemos admitir novos clientes, servir os clientes ativos e gerenciar a memória concorrentemente, através de *threads* paralelas para cada conjunto de módulos. Todavia, devido ao modo de funcionamento do *proxy*, onde, a admissão de novos clientes modifica o estado atual das estruturas de dados internas, como as sequências de blocos de vídeo alocadas em memória e os limites dos *UsersGroups*, assim como

a transferência do conteúdo aos clientes ativos acarreta no deslocamento dos *UsersGroups* e a modificação dos *BlockIntervals* ao mesmo tempo em que o módulo de gerenciamento de memória busca alocar o conteúdo requisitado, substituindo blocos de vídeo da memória e alterando grande parte das estruturas de dados internas, o controle necessário para sincronizar tais operações impossibilitou que tal solução gerasse resultados satisfatórios, uma vez que o tempo gasto para garantir a integridade dos dados a cada operação aumentava significativamente conforme o número de clientes ativos e o número de substituições de memória aumentavam.

Desta maneira, buscando manter o sistema escalável, foi desenvolvido uma solução em que *proxies* de *VoD* operam paralelamente sobre diferentes conjuntos de vídeos e, consequentemente, sobre diferentes grupos de clientes ativos, implementados por *threads* independentes, compartilhando recursos, tais como, a memória principal, tabela de mapeamento de blocos de vídeo, o vetor de troca de endereços (*Addresses_Swap_Array*) e o vetor de cargas (*Load_Set_Array*).

Figura 3.1 – Arquitetura paralela do *proxy* de *VoD*



Fonte: Arquivo pessoal.

A arquitetura do sistema paralelo implementado é apresentada na Figura 3.1. Os *proxies* são homogêneos, ou seja, possuem as mesmas estruturas internas e fluxo de execução. A rotina de serviço executada permanece praticamente a mesma apresentada na seção anterior, com a diferença da inserção de chamadas às rotinas de verificação da carga de execução e de gerenciamento da porção de memória disponível a cada um dos *proxies*.

Uma vez que os *proxies* operam sobre vídeos diferentes, mapeados para suas respectivas linhas na tabela de mapeamento de blocos de vídeo e que um vídeo não é servidor por mais de um *proxy* simultaneamente, garante-se que os acessos às estruturas por uma *thread* são independentes e não alteram o conteúdo utilizado pelas demais *threads*, mantendo a integridade dos dados armazenados nas estruturas de gerenciamento (*Active_Structures_Tree, Ac-*

tive_Sequences_List, etc) sem a necessidade de implementar um controle de sincronismo para tais operações.

A distribuição de memória entre os *proxies* através de endereços exclusivos e, num primeiro momento, uniforme para todos os *proxies*, visa permitir que estes acessem e modifiquem a memória concorrentemente, sem que seja necessário implementar macanismos de bloqueio de acesso (para garantir que nenhum outro fluxo de alocação ou execução de outra *thread* esteja acessando um mesmo endereço), uma vez que um *proxy* realiza operações de alocação e substituição sobre endereços de memória de conhecimento restrito apenas ao mesmo.

Foi implementada uma *thread* auxiliar, *ManagerThread*, responsável pelo gerenciamento da distribuição de carga de execução e da quantidade de memória disponível entre os *proxies*. Esta *thread* obtém informações do estado atual de processamento dos *proxies* periodicamente no decorrer da execução, como tempos de serviço e número de clientes ativos, e toma providências quanto a redistribuição da carga existente através do rearranjo dos vídeos servidos pelos *proxies* e da distribuição de memória pela da movimentação de endereços através do vetor *Memory_Swap_Array*.

3.2 Distribuição de carga

A distribuição de carga é a estratégia utilizada para arranjar a demanda igualmente entre todos os *proxies* de *VoD* com o propósito de diminuir a latência do sistema e aumentar a taxa de utilização do mesmo de acordo com um parâmetro de distribuição escolhido. A distribuição de carga é composta de três fases: obtenção de informações (os dados atuais de carga de cada *proxy* são obtidos), tomada de decisão (estimativa da carga total atual e verificação da condição que possa disparar a movimentação da carga) e movimentação de requisições (migração da carga dos nodos altamente carregados para os nodos mais ociosos) (CHANDRA; SAHOO, 2009).

Algoritmos de distribuição de carga podem ser caracterizados como estáticos ou dinâmicos. Técnicas de distribuição de carga estáticas são técnicas simples uma vez que novas requisições são distribuídas para os nodos por uma estratégia fixa. Estas estratégias são de fácil implementação, mas não são capazes de adaptarem-se às situações de execução. Técnicas de gerenciamento de carga dinâmicas são estratégias adaptativas que podem reagir a mundanças esporádicas no sistema. As requisições de vídeos são distribuídas de acordo com a carga dos nodos disponíveis. Em sistemas de distribuição de carga dinâmicos, os valores de carga precisam ser atualizados em pequenos intervalos de tempo. Isto por sua vez aumenta o tráfego de

comunicação na rede para a transmissão de informações de carga. Assim, um compromisso entre *overhead* de comunicação e a relevância das informações deve ser realizado (VINAY et al., 2011). Analogamente, em um algoritmo *multithread* a frequência de comunicação e atualização entre as *threads* deve ser controlada para que não interfira no desempenho total do sistema, reduzindo o tempo de processamento útil devido às trocas de contexto e ao controle de acesso às regiões de memória compartilhadas.

Algoritmos de distribuição de carga estáticos e dinâmicos ainda podem ser classificados como centralizados e distribuídos. A solução centralizada adota um nodo como o responável pela obtenção das informações dos nodos e da tomada de decisão pela redistribuição da carga. Métodos centralizados podem suportar um maior número de clientes e possuem um menor *overhead* de comunicação mas são menos confiáveis, tendo em vista o único ponto de falha que pode comprometer todo o sistema. Estratégias distribuídas permitem que mais de um nodo do sistema coordene as decisões de distribuição de carga. Entretanto, são mais custosos para obter e gerenciar as informações dinâmicas do sistema.

O algoritmo de distribuição implementado baseou-se numa estratégia dinâmica e centralizada, utilizando o método probabilístico *Simulated Annealing* (*SA*) ou Recozimento Simulado desenvolvido por (KIRKPATRICK; GELATT JR.; VECCHI, 1983) e (CERNÝ, 1985) para encontrar uma distribuição de carga satisfatória.

Annealing é o processo utilizado para fundir um metal, onde este é aquecido a uma temperatura elevada e em seguida é resfriado lentamente, de modo que o produto final seja uma massa homogênea. Esta técnica é utilizada em problemas de otimização combinatória, $min_x\ f(x),\ x \in S$, onde f : S $\rightarrow \mathbb{R}$, S finito. Neste contexto, o processo de otimização é realizado por níveis, simulando os níveis de temperatura no resfriamento. Em cada nível, dado um ponto $u \in S$, vários pontos na vizinhança de u são gerados e o correspondente valor de f é calculado. Cada ponto gerado é aceito ou rejeitado de acordo com uma certa probabilidade. Esta probabilidade de aceitação decresce de acordo com o nível do processo, ou equivalentemente, de acordo com a temperatura (TROSSET, 2001).

O algoritmo 2 apresenta os passos realizados no processo de *SA*. Neste algoritmo, $T_k \in \mathbb{R}^*_+$ representa a temperatura no nível k e L_k o número de pontos que serão gerados neste nível. Inicialmente, T_0 e L_0 são fixados, e um ponto inicial, u, é escolhido em S.

início
  Inicializar $u,\ T_0,\ L_0$;
  $k \leftarrow 0$;
  enquanto $k < k_{max}$ **faça**
    para $l \leftarrow 1$ **até** L_k **faça**
      Gerar w de $V(u)$;
      se $f(w) \leq f(u)$ **então**
        $u = w$;
      fim
      se $random(0,1] < exp(\frac{f(u)-f(w)}{T_k})$ **então**
        $u = w$;
      fim
    fim
    $k++$;
    Calcular L_k e T_k;
  fim
fim

Algoritmo 2: *Simulated Annealing*

A ideia do algoritmo é inicialmente aceitar quase todas as transições propostas, a fim de escapar de um minimizador local (para isso, T_0 deve ser suficientemente grande) e em seguida aceitar com probabilidade cada vez menor os pontos que pioram o valor da função objetivo, sendo que no limite $T_k \rightarrow 0^+$, só serão aceitos os pontos que melhoram o valor da função objetivo (HAESER; RUGGIERO, 2011).

A distribuição da carga pode se dar através da normalização do total de um dos seguintes parâmetros:

- número de clientes ativos atendidos por cada *proxy*;
- número de fluxos de alocação executados por cada *proxy*;
- número de grupos de clientes contíguos existentes em cada *proxy*;
- número de sequências de blocos de vídeo alocadas em cada *proxy*.
- tamanho médio dos grupos de clientes contiguos em função do número de grupos de clientes em cada *proxy*

- tamanho médio das sequências de blocos de vídeo em função do número de sequências alocadas em cada *proxy*

Uma vez que buscamos distribuir a carga de execução através dos *proxies* da forma mais igualitária possível, a função de energia que rege o sistema busca rearranjar os vídeos de maneira a reduzir a variância do parâmetro de distribuição selecionado para cada *thread*, conforme apresentado no algoritmo 3.

```
início
    unsigned int i, j;
    double α, avg, total_dev;
    avg = 0;
    total_dev = 0;
    /* Calcula o peso da carga de cada proxy                    */
    para i ← 1 até NUM_THREADS faça
        S[i].load_parameter = 0;
        para j ← 1 até s[i].num_videos faça
            S[i].load_parameter += S[i].load_parameter[j];
        fim
        /* Acumula o total do parametro de carga                */
        avg += S[i].load_parameter;
    fim
    /* Calcula a media da carga dos proxies                     */
    avg = avg/NUM_THREADS;
    /* Calcula a variancia da distribuicao                      */
    para i ← 1 até NUM_THREADS faça
        α = |avg − S[i].load_parameter|;
        total_dev += α²;
    fim
    /* Retorna a energia da distribuicao calculada              */
    /* (variancia acumulada)                                    */
    return avg_dev;
fim
```

Algoritmo 3: Cálculo de energia da carga distribuída

Por exemplo, utilizando-se como parâmetro de distribuição de carga o número de clientes ativos em cada *proxy*, seja S_i o conjunto de clientes ativos dos vídeos servidos pelo *proxy* i, e U_j o número de clientes ativos do vídeo j ($U_j \geq 0$), tal que $S_i = \{U_a, U_b, U_c, ..., U_n\}$.

$S_0 = \{258, 106, 59, 45, 63\} = 531$ clientes ativos.
$S_1 = \{150, 75, 65, 51, 50\} = 391$
$S_2 = \{139, 82, 55, 35, 39\} = 350$
$S_3 = \{113, 78, 65, 48, 36\} = 340$

A variância total desta distribuição é igual a 23306. Após executar o algoritmo de distribuição de carga obtemos o sequinte estado:

$S_0 = \{258, 106,\ 50\} = 414$ clientes ativos.
$S_1 = \{150,\ 75,\ 59,\ 78,\ 39\} = 401$
$S_2 = \{139,\ 51,\ 55,\ 45,\ 63,\ 35\} = 388$
$S_3 = \{113,\ 82,\ 48,\ 65,\ 65,\ 36\} = 409$

A nova variância do sistema passou a ser de 386 e o tempo de serviço necessário para encontrar a solução foi igual a 0.000406 segundos. Ou seja, com a nova distribuição de vídeos teríamos uma configuração muito mais estável, o que elevaria a taxa de utilização dos *proxies* a um custo de processamento muito baixo. A capacidade de podermos selecionar diferentes parâmetros como base para a distribuição de carga possibilita-nos observar quais os impactos no gerenciamento de memória nos diferentes cenários.

3.3 Distribuição de memória

Com a distribuição de carga implementada, imaginou-se que os *proxies* poderiam ter a capacidade de aumentar ou diminuir a região de memória disponível de acordo com as cargas que estes estivessem submetidos. Ou seja, disponibilizar uma porção maior da memória principal às *threads* com maiores volumes de carga de execução, fazendo um melhor uso do conteúdo alocado em memória e, uma menor porção às *threads* com baixa carga de execução.

Em um primeiro momento decidimos optar por uma divisão de recursos proporcionais à carga de serviço, facilmente calculada tendo conhecimento do número total de clientes ativos de todos os *proxies* e do número de clientes servidos por cada *proxy*. Todavia, chegamos a conclusão que a distribuição de memória deveria ser independente e configurável, e que, da mesma maneira que a distribuição de carga, fosse baseada em parâmetros que refletissem diferentes estados dos *proxies*. Sendo assim, a Equação (3.1) apresenta o cálculo da porção de memória que uma *thread i* tem acesso em função do seu parâmetro de distribuição de memória selecionado ($mem\_dist\_parameter_i$) e do valor total do parâmetro acumulado de todas as *threads* ($total\_mem\_dist\_parameter$). "$new\_memory\_share_i$" é dado em endereços de memória.

O parâmetro de distribuição de memória pode se representado através de um dos seguintes parâmetros:

- número de clientes ativos atendidos por cada *proxy*;

- número de fluxos de alocação executados por cada *proxy*;
- número de grupos de clientes contíguos existentes em cada *proxy*;
- número de sequências de blocos de vídeo alocadas em cada *proxy*;
- tamanho médio dos grupos de clientes contiguos em função do número de grupos de clientes em cada *proxy*;
- tamanho médio das sequências de blocos de vídeo em função do número de sequências alocadas em cada *proxy*;
- número de vídeos ativos em cada *proxy*.

$$new\_memory\_share_i = \frac{memory\_size\_blocks \times mem\_dist\_parameter_i}{total\_mem\_dist\_parameter} \tag{3.1}$$

Por exemplo:

Utilizando como parâmetro de distribuição de memória o número de fluxos de alocação ativos em cada um dos 4 *proxies* em execução, logo, $mem\_dist\_parameter_i$ representa o número de *Writers* ativos do *proxy i*. Sejam NWA_i o número de *Writers* ativos executados pelo *proxy i*, FMA_i e NFM_i a fração de memória atual e a nova fração de memória do *proxy i*, respectivamente. O tamanho total da memória em blocos de vídeo (*memory_size_blocks*) é 4000. Inicialmente cada *proxy* possui a mesma quantidade de memória, logo, $FMA_x = 4000/4 = 1000$ endereços de memória.

$NWA_0 = mem\_dist\_parameter_0 = 100$ *Writers* ativos.
$NWA_1 = mem\_dist\_parameter_1 = 150$
$NWA_2 = mem\_dist\_parameter_2 = 140$
$NWA_3 = mem\_dist\_parameter_3 = 250$
$\rightarrow \quad total\_mem\_dist\_parameter = 640$

Aplicando-se a Equação do cálculo da fração de memória de cada *proxy* obtemos:

$$NFM_i = \frac{memory\_size\_blocks \times mem\_dist\_parameter_i}{total\_mem\_dist\_parameter}$$

$$NFM_0 = \frac{4000 \times 100}{640} = 625 \qquad NFM_1 = \frac{4000 \times 150}{640} = 937$$
$$NFM_2 = \frac{4000 \times 140}{640} = 875 \qquad NFM_3 = \frac{4000 \times 250}{640} = 1563$$

Após o cálculo do novo número de endereços de memória que a *thread* terá acesso, é necessário realizar a liberação ou obtenção de endereços e a troca de informações dentre os *proxies*. Isto é realizado através de um vetor com a funcionalidade de área de troca de endereços (*Addresses_Swap_Array*), onde os *proxies* com novo número de endereços de memória menores que o número de endereços atual ($new\_memory\_share_i < current\_memory\_share_i$) liberam blocos de vídeo alocados em memória e inserem estes endereços (que até o momento eram de conhecimento restrito ao mesmo) no vetor de troca para que os *proxies* com novo número de endereços maiores que os atuais os utilizem.

Assim:

$NFM_0 - FMA_0 = |\ 625 - 1000| = 375$ endereços a serem liberados;
$NFM_1 - FMA_1 = |\ 937 - 1000| = \ \ 63$ endereços a serem liberados;
$NFM_2 - FMA_2 = |\ 875 - 1000| = 125$ endereços a serem liberados;
$NFM_3 - FMA_3 = |1563 - 1000| = 563$ endereços a serem obtidos.

A liberação de endereços de memória por um *proxy* busca, primeiramente, remover os endereços de memória livres (armazenados no vetor de endereços disponíveis para alocação, *Free_Addresses_Array*), adicioná-los ao vetor de troca e incrementar o número de endereços liberados ($num\_released\_addresses{+}{+}$). Se o número de endereços liberados não for o suficiente ($num\_released\_addresses < current\_memory\_share_i - new\_memory\_share_i$) então o *proxy* executa a rotina de reciclagem de memória, na tentativa de remover blocos de vídeo alocados e repassar os endereços ao vetor de troca. Caso não seja possível obter o número de endereços a serem liberados pela *thread* devido a quantidade de clientes ativos e, consequentemente, devido ao número de blocos de vídeo reservados, então a *thread* atualiza o novo valor da porção de memória disponível com a diferença do número de endereços liberados ($new\_memory\_share_i = current\_memory\_share_i - num\_released\_addresses$). Assim, o *proxy* permanece com o novo tamanho de memória calculado até que a próxima execução do módulo de distribuição de memória seja disparada e tente modificar a porção de memória novamente.

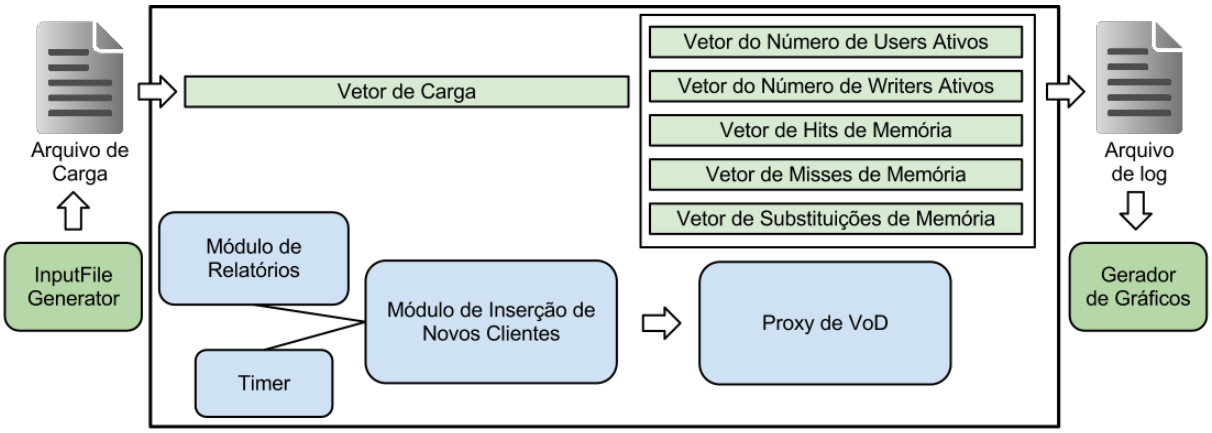
Analogamente, a obtenção de endereços de memória por um *proxy* se dá através da cópia de endereços armazenados no vetor de troca para o seu vetor de endereços de memória livres. A cada endereço extraído do vetor, incrementa-se o número de endereços obtidos e o número de endereços de memória livres do *proxy* ($num\_gathered\_addresses{+}{+}$ e $num\_free\_addresses{+}{+}$). Caso todos os endereços armazenados no vetor de troca tenham sido removidos e adicionados ao vetor de endereços livres do *proxy* e mesmo assim, não tenha sido obtido o número de endereços

necessários para completar a nova porção de memória da *thread* ($num\_gathered\_addresses < new\_memory\_share_i - current\_memory\_share_i$), então a *thread* atualiza o novo valor da porção de memória disponível com a diferença do número de endereços obtidos ($new\_memory\_share_i = current\_memory\_share_i + num\_gathered\_addresses$). Assim, o *proxy* permanece com o novo tamanho de memória calculado até que a próxima execução do módulo de distribuição de memória seja disparada e tente modificar a porção de memória novamente.

4 Simulador

A implementação de um simulador advém do fato de que a validação de um *proxy* de vídeo em condições reais de execução requer a interação com um ambiente complexo do qual fazem parte os clientes, a rede de distribuição e o conjunto de servidores de conteúdo. Na maioria das vezes, estes elementos não estão disponíveis em quantidades e tipos adequados de modo a possibilitar a avaliação experimental do sistema. Além disso, a implementação de um simulador vem ao encontro do interesse em centralizar em uma única plataforma uma variedade de recursos que viabilize a extração de dados como tempo de serviço, consumo de CPU, taxa de *hits* e de substituições de memória, de modo que esta plataforma pudesse ser ainda portável a ponto de permitir sua execução em qualquer computador, afim de que se pudesse comparar os desempenhos obtidos em diferentes máquinas.

Figura 4.1 – Estrutura do Simulador



Fonte: Arquivo pessoal.

A Figura 4.1 ilustra graficamente o conjunto de estruturas que formam o simulador, o qual foi inteiramente implementado em linguagem C para prover total compatibilidade com o *proxy* de VoD desenvolvido, descrito no capítulo 3. O restante deste capítulo está organizado como segue: nas seções 4.1 e 4.2 são apresentadas, respectivamente, as estruturas de dados implementadas que compõem o simulador e os módulos de programa que desempenham as funcionalidades do mesmo. A construção e os modos de execução do simulador são descritos

em 4.3. Finalmente, em 4.4 é relatado o funcionamento passo-a-passo do algoritmo implementado.

4.1 Estruturas de Dados

A seguir são descritas as estruturas de dados que compõem o simulador.

Arquivo de carga de simulação: arquivo de texto que contém a carga de execução do sistema durante o período de simulação. Como pode ser visto na Figura 4.2, o arquivo apresenta a listagem de todas as requisições que serão realizadas ao *proxy*, contendo a ID do cliente, o vídeo requisitado, o instante em que a requisição é emitida ao *proxy* a partir do início da simulação (dado em rodadas de serviço), e o *offset* do bloco de vídeo inicial requisitado.

Figura 4.2 – Exemplo de arquivo de carga de simulação.

RequisiçãoID	VideoID	TempoChegada	BlocoVideoInicial
0	1	0	0
1	1	0	0
2	1	1	0
3	2	1	0
4	2	10	0
5	3	5	0
6	2	7	0
...	...	...	...
N-2	1	100	0
N-1	2	200	0
N	1	150	0

Fonte: Arquivo pessoal.

O arquivo de carga de simulação é gerado a partir de um algoritmo auxiliar (*InputFileGenerator*), de acordo com os parâmetros de carga passados, como por exemplo, o número máximo de clientes, o número máximo de vídeos, o intervalo de realização de novas requisições, a distribuição de vídeos característica (aleatória ou distribuição Zipf), entre outros. A utilização de diferentes parâmetros leva a geração de cargas de simulação com diferentes características, como a dispersão dos clientes no decorrer dos vídeos e através da linha temporal de execução e do número de clientes ativos em cada vídeo. Tais conFigurações causam impactos diretos na taxa de *hits* de memória e no tempo de serviço do *proxy*.

Vetor de carga inicial: implementado através de um vetor estático do tipo *User*, contendo todos os clientes carregados do arquivo de carga de simulação, ordenados em função do

tempo de chegada no sistema. A cada rodada de serviço do simulador, o módulo de inserção de novos clientes verifica se existem requisições que devem ser realizadas ao *proxy* no instante atual ($User_x \rightarrow arrive\_time == execution\_round$). Se existir, adiciona os novos clientes ao buffer de novas requisições do *proxy* (*New_Users_Array*). Uma vez que apenas o simulador possui conhecimento de todas as requisições da carga de simulação, as decisões tomadas pelos *proxies* são baseadas apenas nos clientes ativos do sistema, tal como ocorre em um sistema real.

Vetores de dados: vetores estáticos responsáveis por armazenar os dados obtidos durante a simulação, como por exemplo, o tempo de execução de cada rodada de serviço dos *proxies*, o número de *hits*, *misses* e substituições de memória, assim como o número de *Users* e *Writers* ativos em cada rodada de serviço. Além disso, são armazenados o número de requisições realizadas o servidor principal e o número de impossibilidades de alocação em memória.

Arquivo de relatório: representa um arquivo de texto exportado a partir do simulador, contendo os dados obtidos no decorrer da simulação e armazenados nos vetores de dados.

4.2 Módulos de Programa

Os principais módulos de programa implementados que constituem o simulador são apresentados a seguir:

Módulo de inserção de novos clientes: módulo implementado através de uma *thread*, responsável por analisar o vetor de carga a cada segundo e inserir as requisições com tempo de chegada ao sistema igual a rodada de serviço corrente no vetor de novos clientes (*New_Users_Array*) dos *proxies*.

***Proxies* de VoD:** representam os *proxies* de VoD descrito no capítulo 3. Cada instância de um *proxy* é implementado por uma *thread*, o que possibilita que os *proxies* e o módulo de inserção de novas requisições executem paralelamente e independentemente.

Módulo de Profiling: rotina responsável por extrair os dados de execução dos *proxies* no decorrer da simulação e exportá-los para o arquivo de relatório.

***Timer*:** é responsável por realizar o sincronísmo do sistema, coordenando a verificação e inserção de novas requisições a cada segundo transcorrido de acordo com a rodada de serviço.

Módulo do IPCM: é o módulo de *profiling* implementado utilizando a ferramenta disponibilizada pela Intel, *IPCM* (DEMENTIEV; WILLHALM; BRUGGEMAN, 2012) e apenas é ativado quando a arquitetura hospedeira pertence às famílias de processadores Xeon e Core da Intel. O *IPCM* provê rotinas em C++ para estimar a utilização de recursos internos das CPUs, como número de instruções executadas, acertos em cache nível L2 e L3, etc. Com o propósito de obter uma análise mais precisa da utilização dos recursos de CPU, o IPCM utiliza-se dos dados dinâmicos obtidos a partir das unidades de monitoramento de performance (*PMU*).

4.3 Compilação e Modos de Execução

A compilação do simulador dá-se através de um arquivo *Makefile* disponibilizado juntamente com os arquivos fonte, facilitando e automatizando o processo. A construção é dada pelo comando abaixo:

" make {SimulationMode}_{MemorySubstitutionStrategy} [parameters] "

SimulationMode define o modo de simulação em função dos dados extraídos e apresentados no decorrer da execução e dos recursos oferecidos para depuração. Os modos disponíveis são apresentados abaixo:

***Profiling*:** versão de extração de dados do simulador. Apresenta todos os recursos disponíveis para a captação de dados de execução, como:

- tempo de serviço;
- número de clientes ativos;
- número de fluxos de alocação ativos;
- número de vídeos ativos;
- número de grupos de usuários contíguos ativos;
- número de novos clientes admitidos;
- número de novos clientes rejeitados;
- número de *hits* de memória;
- número de *misses* de memória;
- número de substituições de memória;

- número de erros de alocação de memória;
- tamanho da memória;
- número de requisições servidas pelo servidor principal;
- número de requisições não servidas pelo servidor principal;
- dados de *profiling* do *IPCM*;
  - número de ciclos de CPU executados;
  - número de instruções executadas;
  - número de instruções executadas por ciclo de clock;
  - número de bytes lidos a partir da memória;
  - número de bytes escritos para a memória;
  - número de *hits* de Cache nível L2;
  - número de *misses* de Cache nível L2;
  - número de ciclos de CPU perdidos devido a um *miss* de Cache nível L2;
  - número de *hits* de Cache nível L3;
  - número de *misses* de Cache nível L3;
  - número de ciclos de CPU perdidos devido a um *miss* de Cache nível L3;

***Log*:** versão de *log* do sistema, capaz de exportar todas as ações realizadas pelo *proxy* durante as rodadas de serviço.

***Debug*:** versão voltada à depuração do simulador. Adiciona a funcionalidade de depuração ao sistema, exibindo informações adicionais no decorrer da execução, como os estados dos vetores de *Users*, *Videos*, *Writers*, *UsersGroups*, *BlockIntervals* e *Sequence*, da tabela de mapeamento de blocos e do vetor de popularidade de vídeos.

A estratégia de substituição de memória utilizada pelo *proxy* é especificada através de *MemorySubstitutionStrategy*. Os valores permitidos para *ProxyMode* são *CARTE*, *CC* e *CCVC*, e representam as estratégias apresentadas na seção 2.2.

Através dos parâmetros de configuração opcionais é possível estipular:

***NUM_THREADS*:** o número de *Service Threads* em execução e consequentemente, o número de *proxies* de VoD paralelos. $NUM\_THREADS \in \mathbb{N}^*$.

***DEMANDSPACE*:** tamanho do espaço de demanda prévio à sequência de blocos de vídeo alocada em memória, cuja prioridade de descarte está sendo calculada. O valor é dado

em unidades de bloco de vídeo. *DEMANDSPACE* $\in \mathbb{N}^*$. Válido apenas para o modo *CARTE*.

***attenuationFactor*:** valor do fator de atenuação aplicado na função de cálculo da prioridade de descarte de uma sequência de blocos de vídeo alocada em memória. Desta maneira, a Equação 2.6 passa a ser calculada conforme apresenta a Equação 4.1. *attenuationFactor* $\in \mathbb{R}_+^*$. Válido apenas para o modo *CARTE*. Este fator busca reduzir o impacto do tamanho das sequências sobre a prioridade de descarte, de maneira que o número de clientes seja o elemento principal no cálculo da prioridade.

$$PS_s = \frac{NC_s}{|\log(SZ_s \times attenuationFactor)|} \times \frac{UI_s}{|\log(SZ_s \times attenuationFactor)|} \tag{4.1}$$

***LOAD_BALANCE*:** este parâmetro permite aos *proxies* ativarem ou desativarem a funcionalidade de distribuição de carga entre os nodos em execução. *NUM_THREADS* > 1 e *LOAD_BALANCE* $\in \mathbb{P}$, $\mathbb{P} = \{on, off\}$. Se a funcionalidade for desativada, a rotina de distribuição de carga não será executada e os *proxies* irão atuar sobre um acervo de vídeo estático. Caso contrário, se a funcionalidade for ativada então é necessário definir qual o parâmetro utilizado no cálculo de distribuição de carga:

- ***LOAD_PARAM_NUM_ACTIVE_USERS*:** distribuição de carga em função do número de clientes ativos em cada *proxy*
- ***LOAD_PARAM_NUM_ACTIVE_WRITERS*:** distribuição de carga em função do número de fluxos de alocação ativos em cada *proxy*
- ***LOAD_PARAM_NUM_ACTIVE_USERSGROUPS*:** distribuição de carga em função do número de grupos de clientes contíguos ativos em cada *proxy*
- ***LOAD_PARAM_SEQUENCES_LENGTH*:** distribuição de carga em função do comprimento médio das sequências de blocos de vídeo alocadas em cada *proxy*
- ***LOAD_PARAM_USERSGROUPS_LENGTH*:** distribuição de carga em função do tamanho médio dos grupos de clientes contiguos em função do número de grupos de clientes em cada *proxy*;
- ***LOAD_PARAM_SEQUENCES_LENGTH*:** distribuição de carga em função do tamanho médio das sequências de blocos de vídeo em função do número de sequências alocadas em cada *proxy*;

***LIMITED_SERVER_COMMUNICATION*:** ativa ou desativa o limite de comunicação com o servidor principal, restringindo o número de fluxos de alocação servidos a cada rodada

de serviço. Se *LIMITED_SERVER_COMMUNICATION* for definido, então deve-se estipular a banda de comunicação máxima disponível (*MAX_SERVER_BANDWIDTH*) em Mbps. Tal que, *LIMITED_SERVER_COMMUNICATION* $\in \mathbb{P}$, $\mathbb{P} = \{on, off\}$ e *MAX_SERVER_BANDWIDTH* $\in \mathbb{R}_{+}^{*}$. Caso contrário, todas as requisições realizadas ao servidor principal pelos fluxos de alocação serão atendidas e o sucesso da alocação do conteúdo será função apenas da existência de espaço disponível em memória.

***MEMORY_DISTRIBUTION*:** ativa ou desativa a funcionalidade de distribuição de memória entre os *proxies*. *NUM_THREADS* > 1 e *MEMORY_DISTRIBUTION* $\in \mathbb{P}$, $\mathbb{P} = \{on, off\}$. Se *MEMORY_DISTRIBUTION* não for definido então a rotina de distribuição de memória não será executada e as memórias dos *proxies* serão estáticas e terão o tamanho $MEMORY\_SIZE\_MB/NUM\_THREADS$ calculado no início da simulação. Caso contrário, se a funcionalidade for ativada então é necessário definir o parâmetro utilizado no cálculo de distribuição de memória:

- ***MEM_PARAM_NUM_ACTIVE_USERS*:** distribuição de memória em função do número de clientes ativos em cada *proxy*.
- ***MEM_PARAM_NUM_ACTIVE_WRITERS*:** distribuição de memória em função do número de fluxos de alocação ativos em cada *proxy*.
- ***MEM_PARAM_NUM_ACTIVE_VIDEOS*:** distribuição de memória em função do número de vídeos ativos em cada *proxy*.
- ***MEM_PARAM_NUM_ACTIVE_USERSGROUPS*:** distribuição de memória em função do número de grupos de clientes contíguos ativos em cada *proxy*.
- ***MEM_PARAM_SEQUENCES_LENGTH*:** distribuição de memória em função do comprimento médio das sequências de blocos de vídeo alocadas em cada *proxy*.
- ***MEM_PARAM_USERSGROUPS_LENGTH*:** distribuição de memória em função do tamanho médio dos grupos de clientes contiguos em função do número de grupos de clientes em cada *proxy*;
- ***MEM_PARAM_SEQUENCES_LENGTH*:** distribuição de memória em função do tamanho médio das sequências de blocos de vídeo em função do número de sequências alocadas em cada *proxy*;

***SAFE_RESOURCE_CONSUMING*:** ativa ou desativa a funcionalidade de restringir a admissão de novos clientes de acordo com a existência de recursos disponíveis no *proxy*. Conforme novos clientes são admitidos, novos *UsersGroups* são criados, ocasionando a reserva de blocos de vídeo em memória e reduzindo a quantidade de endereços de memória

disponíveis a serem alocados e substituidos. Quando o número de clientes é suficientemente alto, a ponto de reservar todos os endereços de memória do *proxy*, a admissão de novos clientes fará com que alguns fluxos de alocação futuros não consigam encontrar endereços de memória disponíveis para alocarem novos blocos de vídeo. Isso tende a elevar a taxa de *misses* de memória do *proxy*, uma vez que os clientes avançarão seus fluxos de execução e não encontrarão o conteúdo requisitado alocado em memória, necessitando aguardar no bloco corrente até que o fluxo de alocação correspondente ao seu *UsersGroup* consiga um endereço disponível para alocar o bloco de vídeo requisitado. Entretanto, isto permite que um número maior de clientes sejam servidos pelo *proxy*, a custos acessíveis. *SAFE_RESOURCE_CONSUMING* $\in \mathbb{P}$, $\mathbb{P} = \{on, off\}$.

***PRINT*:** ativa ou desativa a exibição de informações da simulação no *buffer* de saída padrão do sistema. *PRINT* $\in \mathbb{P}$, $\mathbb{P} = \{on, off\}$.

***TIMING*:** permite que o sistema seja avaliado sem que seja necessário marcar a execução do sistema segundo-a-segundo, ou seja, os *proxies* não necessitam aguardar pelo vencimento de um segundo para dar partida a nova rodada de serviço. *TIMING* $\in \mathbb{P}$, $\mathbb{P} = \{on, off\}$.

***FUNCIONAL*:** modo de execução funcional. Ativa ou desativa as transferências de dados, ou seja, o conteúdo alocado em memória não é copiado para os *buffers* de saída dos clientes e os blocos de vídeos disponíveis no *buffer* de entrada do *proxy* não são copiados para a memória. Isto permite verificar qual o tempo de execução exclusivo para realizar o gerenciamento do *proxy*, sem o gasto de processamento com a cópia de conteúdo aos clientes. *FUNCTIONAL* $\in \mathbb{P}$, $\mathbb{P} = \{on, off\}$.

4.4 Rotina de Serviço

O simulador possui uma interface para configuração de parâmetros da simulação. A sintaxe para execução do simulador via linha de comando é exibida a seguir:

" ./StartSimulator initializationFile.txt VideoBitRateMbps NumMaxUsers NumMaxVideos MemSizeMB VideoSizeMB NumSimulationRounds "

A descrição de cada parâmetro presente nesta sintaxe é apresentada como segue:

StartSimulator: arquivo executável do simulador.

Initializationfile.txt: arquivo de carga de simulação gerado pelo *InputFileGenerator*.

VideoBitRateMbps: determina a taxa de codificação dos vídeos gerenciados pelo *proxy* em Mbps, ou seja, o tamanho de cada bloco de vídeo e a quantidade de dados a serem transferidos por rodada de serviço a cada cliente.

NumMaxUsers: número máximo de clientes a serem carregados. Caso o número de clientes presentes no arquivo de inicialização seja maior que *NumMaxUsers*, todos os clientes excedentes serão ignorados.

NumMaxVideos: número máximo de vídeos disponíveis para acesso.

MemSizeMB: tamanho da memória do *proxy* em Megabytes.

VideoSizeMB: tamanho dos vídeos do acervo em Megabytes.

NumSimulationRounds: tempo limite de simulação em rodadas de serviço. Ao contrário do cenário real de funcionamento no qual o *proxy* tende a executar ininterruptamente, a simulação precisa ser interrompida para que os dados produzidos sejam exportados para arquivos texto. A estratégia de não exportar os dados para arquivos em disco em tempo de execução tem como objetivo não executar tarefas que envolvam acesso a dispositivos lentos (tais como barramentos de E/S, discos, memórias flash ou interfaces de rede), uma vez que isto poderia produzir uma interferência sobre o tempo de serviço do *proxy* durante a simulação em tempo real.

Primeiramente as estruturas internas do *proxy* são alocadas e inicializadas (tabela de mapeamento de blocos de vídeo, vetor de popularidade de vídeos, etc). Em seguida, o arquivo de carga de simulação passado como parâmetro é carregado para o sistema, transferindo as informações contidas no mesmo para o vetor de carga. Os clientes carregados são ordenados em função do instante de chegada ao sistema, a fim de agilizar o processo de busca por novos clientes pelo simulador a cada rodada de serviço.

Consecutivamente, as *threads* responsáveis por executarem o módulo de inserção de novos clientes e os *proxies* de vídeo são disparadas concorrentemente, de forma que a inserção de novas requisições a partir do vetor de carga do simulador para o vetor de novas requisições do *proxy* pelo módulo de inserção e a execução do *proxy* ocorram de forma independente. Para a implementação das *threads* foi utilizada a *API POSIX Threads* (*Pthreads*) (BUTENHOF, 1997), por apresentar um conjunto de interfaces para a programação paralela muito bem documentado, além de ser vastamente utilizado e especificar atributos distintos, como, política de escalonamento de tempo real, afinidade de CPU, captura de sinais de interrupção por *thread*, entre outros.

O módulo de inserção de novos clientes verifica a partir do início do vetor de carga, a existência de um cliente com tempo de chegada igual ao contador de rodadas de serviço (*execution_round*). Caso exista um novo cliente, este é adicionado ao vetor de novas requisições do *proxy* (*New_Users_Array*) e o cliente seguinte do vetor é verificado. Caso contrário, se o tempo de chegada ao sistema for maior que o valor do contador, o índice do vetor é salvo (*last_checked_user*) e o módulo executará a verificação por novos clientes a partir desta posição do vetor na próxima rodada de serviço.

Ao final da simulação, o módulo de relatórios exporta os dados obtidos no decorrer da execução e armazenados nos vetores de dados para um arquivo de texto em disco. Cada linha do arquivo de log exportado representa uma rodada de serviço.

A partir do arquivo de log exportado, é possível representar os dados obtidos em modo gráfico através do gerador de gráficos implementado em linguagem *Python* (ROSSUM, 1995), utilizando-se as bibliotecas *numpy* (ASCHER et al., 1999) e *matplotlib* (HUNTER, 2007).

5 Experimentos

Com o propósito de validar a arquitetura implementada foram realizados diversas simulações, buscando submeter o sistema a diferentes cargas de execução, a fim de garantir que o sistema se comportaria da maneira esperada. Desta maneira, são apresentados alguns dos resultados obtidos.

5.1 Teste de desempenho do algoritmo de distribuição de carga

Para analisar o desempenho do algoritmo de distribuição de carga foram executadas algumas simulações, com a finalidade de validar o algoritmo e verificar o tempo necessário para encontrar uma solução aceitável. Os experimentos foram executados sobre um servidor de propósito geral Intel Xeon E5530 (2.4GHz), com espaço de memória de 16GB (DDR3 RAM 1066MHz) e sistema operacional SUSE Linux Enterprise Server (3.0 kernel).

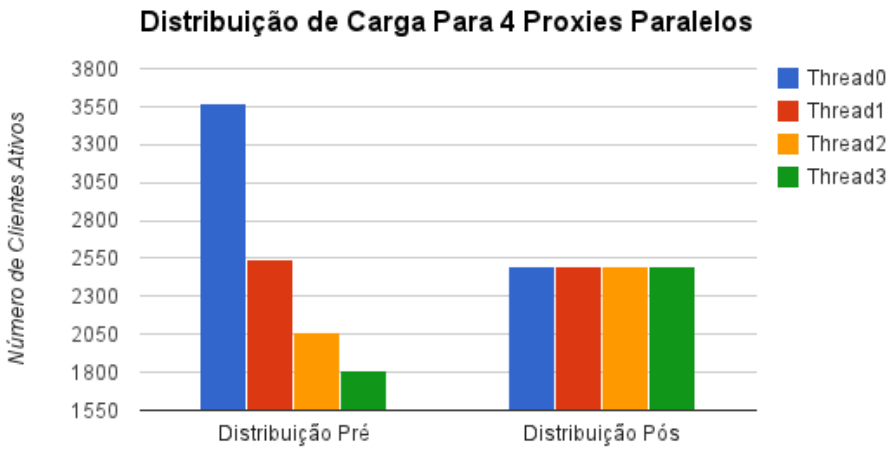
São apresentados resultados de distribuição de carga com 10000 clientes ativos, distribuídos através de 100 e 1000 vídeos, executando sobre 4, 6, 8 e 10 *threads* de serviço.

Tabela 5.1: Distribuição de carga para 4 *proxies* paralelos.

	Número de Vídeos	
	100	1000
Distribuição da Carga Pré	[3576] [2544] [2068] [1812]	[3292] [2435] [2138] [1993]
Energia Pré	1819680	1014541
Distribuição de Carga Pós	[2495] [2502] [2503] [2500]	[2465] [2465] [2465] [2463]
Energia Pós	38	3
Tempo de Execução (s)	0.001266	0.010526

Como pode ser visto nas tabelas 5.1, 5.2, 5.3 e 5.4, o tempo de execução do algoritmo de distribuição de carga é relativamente baixo, não impactuando no desempenho geral dos *proxies* tendo em vista que o módulo de distribuição de carga é disparado esporádicamente. Em adição, podemos verificar que o método de *SA* é capaz de encontrar uma solução próxima de

Figura 5.1 – Distribuição de 1000 vídeos para 4 *proxies* paralelos



Fonte: Arquivo pessoal.

Tabela 5.2: Distribuição de carga para 6 *proxies* paralelos.

	Número de Vídeos	
	100	1000
Distribuição da Carga Pré	[2916] [1921] [1524] [1329] [1204] [1106]	[2528] [1712] [1568] [1432] [1334] [1285]
Energia Pré	2288299	1061777
Distribuição de Carga Pós	[1881] [1625] [1622] [1621] [1628] [1623]	[1772] [1624] [1617] [1615] [1616] [1615]
Energia Pós	55157	19975
Tempo de Execução (s)	0.001327	0.011274

ótima [Figura 5.1] dentro de um tempo de execução aceitável, reduzindo a variância da distribuição de carga em até 330000 vezes e 8 vezes com 1000 vídeos ativos para 4 e 10 *proxies*, respectivamente. Todavia, devido a granularidade do algoritmo ser baseada sobre a distribuição de vídeos do acervo através dos *proxies*, podem ocorrer casos onde o número de clientes requisitando um vídeo específico seja muito maior que a popularidade média dos outros vídeos do acervo, o que pode impedir o aumento da produtividade do sistema, como pode ser visto na Figura 5.2.

Tabela 5.3: Distribuição de carga para 8 *proxies* paralelos.

	Número de Vídeos	
	100	1000
Distribuição da Carga Pré	[2590] [1620] [1264] [1057] [986] [924] [804] [755]	[2216] [1450] [1213] [1113] [1078] [987] [925] [882]
Energia Pré	2589858	1330784
Distribuição de Carga Pós	[1881] [1161] [1149] [1156] [1171] [1169] [1155] [1158]	[1404] [1214] [1206] [1214] [1204] [1214] [1202] [1206]
Energia Pós	455410	33584
Tempo de Execução (s)	0.000940	0.010614

Tabela 5.4: Distribuição de carga para 10 *proxies* paralelos.

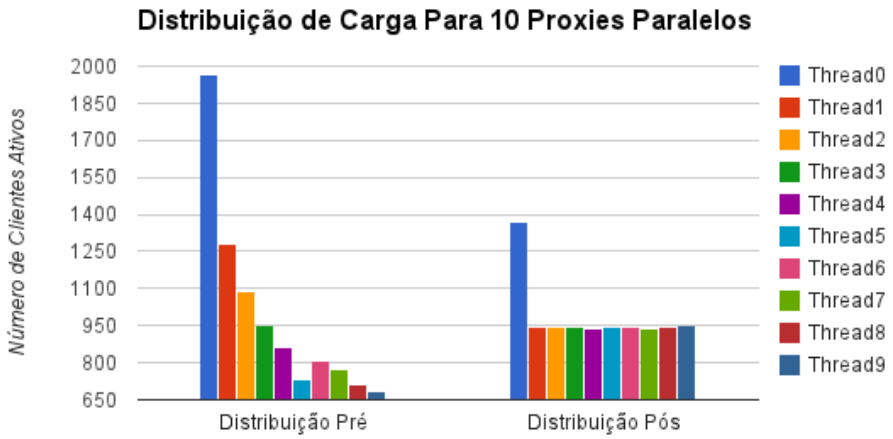
	Número de Vídeos	
	100	1000
Distribuição da Carga Pré	[2436] [1465] [1081] [899] [811] [760] [689] [690] [627] [542]	[1967] [1283] [1090] [950] [862] [733] [805] [776] [709] [685]
Energia Pré	2930118	1386258
Distribuição de Carga Pós	[1881] [936] [909] [908] [893] [888] [897] [896] [901] [891]	[1369] [945] [944] [943] [937] [945] [942] [938] [948] [949]
Energia Pós	864102	163118
Tempo de Execução (s)	0.001550	0.010839

5.2 Teste de desempenho do *proxy* paralelo em conjunto com a distribuição de carga

As tabelas 5.5 e 5.6 apresentam os dados relativos a um experimento com quatro *proxies* paralelos trabalhando em conjunto para servir 10000 clientes ativos, distribuídos através de 1000 vídeos, com uma taxa de codificação de 2Mbps, e com o módulo de distribuição de carga desativado e ativado, respectivamente.

Podemos identificar através das tabelas 5.5 e 5.6 que quando o algoritmo de distribuição de carga está desativado, ou seja, os vídeos não são realocados através dos *proxies* (*threads*), temos uma taxa de acertos de memória média ($CacheHit_{avg}$) de 86,72%, enquanto a variância média da distribuição de carga dos ($DistributionVariance_{avg}$) sobre os *proxies* é de 405,2 e o tempo de serviço médio ($ServiceTime_{avg}$) é igual a 0,4280 segundos (tempo de serviço máximo de 0,4723 segundos). Todavia, quando ativamos a funcionalidade de distribuição

Figura 5.2 – Distribuição de 1000 vídeos para 10 *proxies* paralelos



Fonte: Arquivo pessoal.

Tabela 5.5: Desempenho de 4 *proxies* paralelos sem distribuição de carga (2Mbps).

ThreadID	Tempo de Serviço Médio (s)	Users Ativos	Writers Ativos	Videos Ativos	Hits	Misses
[3]	0.395177	2026	1151	227	1727	299
[2]	0.414628	2170	1151	224	1871	299
[1]	0.430264	2480	1164	221	2138	342
[0]	0.472346	3324	1190	228	2967	357

de carga, $DistributionVariance_{avg}$ é aproximadamente zero, enquanto $CacheHit_{avg}$ sofre um aumento, sendo igual a 86,99% e $ServiceTime_{avg}$ passa para 0,4414 segundos (com um tempo de serviço máximo de 0,4541 segundos). Ou seja, conseguimos distribuir a carga através dos *proxies* ativos, de maneira a reduzir o maior tempo de execução, normalizar o processamento através das *threads* e, em adição, aumentar a taxa de *hits* de memória.

5.3 Teste de desempenho do algoritmo antigo vs. nova implementação

Durante o início da segunda fase de desenvolvimento do trabalho, buscamos modificar o algoritmo implementado de maneira a reduzir o tempo de execução das rodadas de serviço. Para isso, foram modificadas as rotinas de admissão de novos clientes, de gerenciamento das

Tabela 5.6: Desempenho de 4 *proxies* paralelos com distribuição de carga (2Mbps).

ThreadID	Tempo de Serviço Médio (s)	Users Ativos	Writers Ativos	Videos Ativos	Hits	Misses
[3]	0.422417	2499	1267	251	1971	528
[2]	0.444984	2503	1310	250	2143	360
[1]	0.444455	2500	1204	236	2149	351
[0]	0.454135	2498	843	163	2437	61

estruturas internas, de atualização dos *BlockIntervals* e das sequências de blocos de vídeo alocadas. As tabelas 5.7 e 5.8 apresentam os tempos de serviço médio e o número de ciclos de *CPU* médio gastos para uma simulação com 1 *proxy* de *VoD* servindo 6000 e 10000 usuários ativos, respectivamente, utilizando os algoritmos das duas fases do trabalho. Nestes exemplos buscamos isolar o desempenho do *proxy* estritamente ao processo de admissão, gerenciamento de estruturas internas, gerenciamento de memória e transferência de dados, removendo qualquer influência por parte da distribuição de carga e de memória.

Tabela 5.7: Desempenho do algoritmo antigo e novo para 6000 usuários ativos.

	Estratégias de Substituição					
	CARTE		CC		CCVC	
Algoritmo	Antigo	Novo	Antigo	Novo	Antigo	Novo
Tempo de Serviço (seg)	0.6997003	0.68138143	0.69891879	0.67822757	0.70107441	0.68076390
Número de ciclos de CPU	3734161430	3589700769	3732365603	3583654755	3738733490	3470052023

Dentre as modificações implementadas, podemos citar:

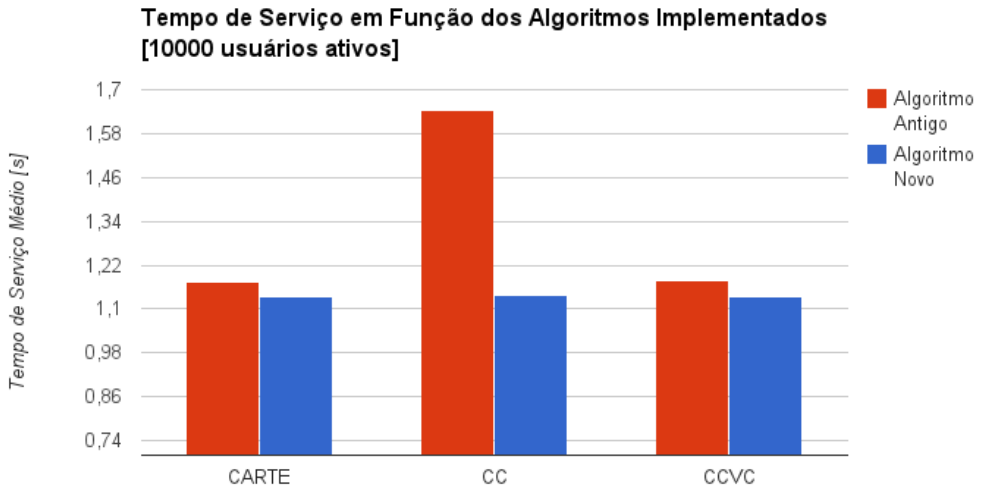
Redução do número de chamadas à rotina de atualização das sequências alocadas: tendo em vista o fluxo em que os módulos do *proxy* são executados (admissão de novos clientes →transferência de dados→gerenciamento de memória), foi possível disparar a rotina de atualização apenas no final de cada ciclo executado, removendo qualquer processo de cálculo de prioridades de substituição do módulo de admissão de novos clientes.

Agrupamento de instruções que não realizam acesso direto à memória: as instruções de processamento de dados que não envolvem o acesso direto à memória foram reunidas e dis-

Tabela 5.8: Desempenho do algoritmo antigo e novo para 10000 usuários ativos.

	Estratégias de Substituição					
	CARTE		CC		CCVC	
Algoritmo	Antigo	Novo	Antigo	Novo	Antigo	Novo
Tempo de Serviço (seg)	1.1740702	1.1333337	1.6436110	1.1378034	1.1779084	1.1322031
Número de ciclos de CPU	6223041170	5988015470	8756632080	6004348070	6214342390	6004701790

Figura 5.3 – Desempenho das duas versões do algoritmo implementado em função do tempo de serviço e da estratégia de descarte.



Fonte: Arquivo pessoal.

postas sequencialmente, prévias às instruções de escrita e leitura de memória, a fim de reduzir a frequência de acesso a mesma. Todavia, não é garantido que a sequência será mantida em linguagem de máquina após o compilador de programa ser executado.

Passagem de parâmetros por referência: todas as funções foram modificadas para tratar a passagem de parâmetros via referência, evitando realizar a cópia do conteúdo a ser processado.

Agrupamento de instruções que acessam a região de memória compartilhada: uma vez que a leitura e escrita às variáveis globais do sistema requerem acesso exclusivo por cada *thread* para garantir a integridade dos dados, as rotinas de distribuição de carga e de memó-

ria foram modificadas, reduzindo-se o número de chamadas às funções "*pthread_mutex_lock()*" e "*pthread_mutex_unlock()*" que realizam o controle de acesso e liberação da região de memória compartilhada, respectivamente.

Mapeamento de variáveis para registradores: variáveis de registradores são um caso especial de variáveis automáticas (variáveis de escopo local). Variáveis automáticas são alocadas em memória, contudo, o acesso de dados em memória é considerávelmente mais lento que a velocidade de processamento da *CPU*. Os computadores possuem pequenas quantidades de armazenamento contíguas às unidades de processamento da *CPU*, onde os dados podem ser acessados rápidamente. Estas células de armazenamento são chamadas registradores. Normalmente, o compilador de programas determina quais dados serão armazenados nos registradores da *CPU* em cada momento. Contudo, a linguagem C apresenta a classe de armazenamento '*register*', de maneira que o programador pode "sugerir" ao compilador quais variáveis deveriam ser armazenadas nos registradores do *CPU*, se possível, provendo certo controle sobre a eficiência de execução dos programas. Assim, algumas das variáveis locais que são repetidamente utilizadas no processamento de dados foram mapeadas para registradores.

Como pode ser visto na Figura 5.3, foi possível reduzir o consumo de *CPU* e o tempo de serviço do algoritmo previamente implementado apenas através da modificação das rotinas de programa do algoritmo implementado. Em especial para a estratégia *CC*, cujos algoritmos de atualização dos limites e das prioridades de descarte das sequências de blocos de vídeo alocadas em memória são mais custosos.

5.4 Teste de desempenho do *proxy* em função da variação do número de clientes servidos

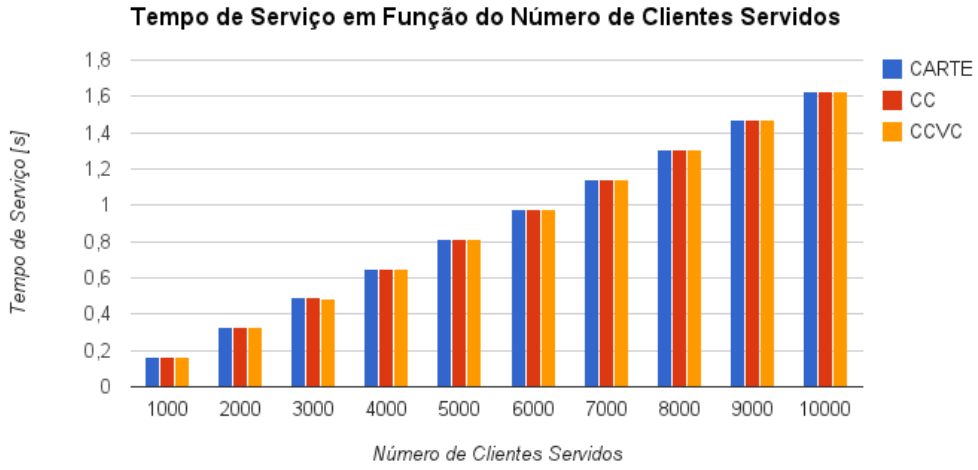
Com o propósito de verificar qual o impacto da carga de trabalho ativa sobre o desempenho do *proxy*, submetemos o mesmo a diferentes números de clientes servidos. A tabela 5.9 apresenta a variação do tempo de serviço do algoritmo em função do número de clientes servidos e da estratégia de substituição utilizada pelo *proxy*. A tabela 5.10 apresenta o número de instruções executadas pelo *proxy*, enquanto a tabela 5.11 exibe o número de bytes escritos em memória médio a cada rodada de serviço para cada uma das configurações. Tal que, NCS representa o número de clientes servidos.

Como podemos visualizar na Figura 5.4, o tempo de serviço é diretamente proporcional ao número de clientes servidos. Através da Figura 5.5 e da Tabela 5.11, podemos identificar

Tabela 5.9: Impacto do número de fluxos de execução servidos sobre o tempo de serviço.

Tempo de execução médio da rodada de serviço (s)			
NCS	Estratégias de Substituição		
	CARTE	CC	CCVC
1000	0.16360253220	0.16295422372	0.16320246779
2000	0.32632850508	0.32518133220	0.32623327796
3000	0.48914341355	0.48921516610	0.48896172881
4000	0.65126883728	0.65084862372	0.65171207118
5000	0.81478973220	0.81639363728	0.81394396271
6000	0.97810027457	0.97669868135	0.97544695593
7000	1.14201407458	1.14026853559	1.14024752881
8000	1.30744831186	1.30394564746	1.30647617966
9000	1.46846984407	1.46776876949	1.46726350508
10000	1.62785459661	1.62738876949	1.63130196610

Figura 5.4 – Tempo de serviço médio em função do número de clientes servidos e da estratégia de descarte.



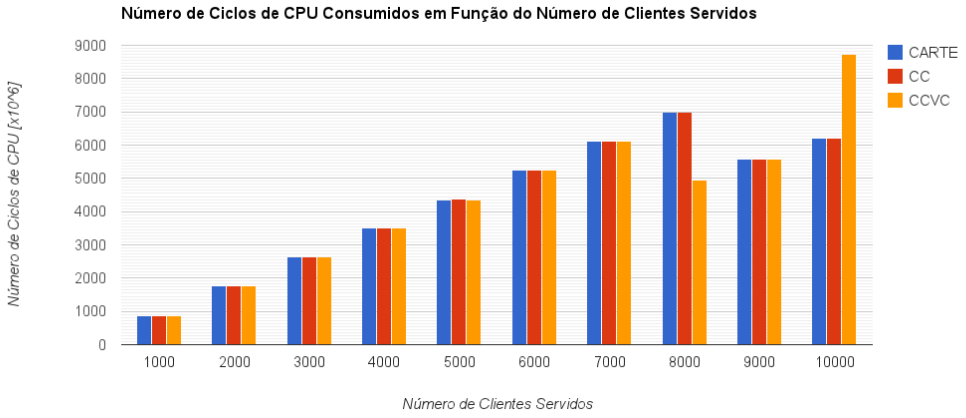
Fonte: Arquivo pessoal.

que o número de clientes tende a ser proporcional ao número de ciclos de CPU executados e ao tráfego de memória utilizado, respectivamente. Desta maneira, podemos estimar qual o consumo de recursos em função da carga cujos *proxies* estão submetidos.

Tabela 5.10: Impacto do número de fluxos de execução servidos sobre o número de instruções executadas pelo *proxy*.

Número médio de instruções executadas em uma rodada de serviço			
NCS	Estratégias de Substituição		
	CARTE	CC	CCVC
1000	332853690	333136505	333039357
2000	665290094	665574609	667950696
3000	997078399	997296714	999885497
4000	1332318526	1335978984	1331949458
5000	1664665417	1664673508	1664204847
6000	1999855735	1994947106	1997608488
7000	2327414061	2331068233	2331078416
8000	2666983942	2662524441	2662960501
9000	2990548779	2993241415	2991012329
10000	3337496906	3329982899	3326319993

Figura 5.5 – Número médio de ciclos de CPU consumidos em função do número de clientes servidos e da estratégia de descarte.



Fonte: Arquivo pessoal.

5.5 Teste de desempenho do *proxy* paralelo em função do grau de paralelismo

Configuração utilizada:

- Número máximo de clientes: 6000 clientes;
- Acervo: 20 vídeos de 2000MB cada;

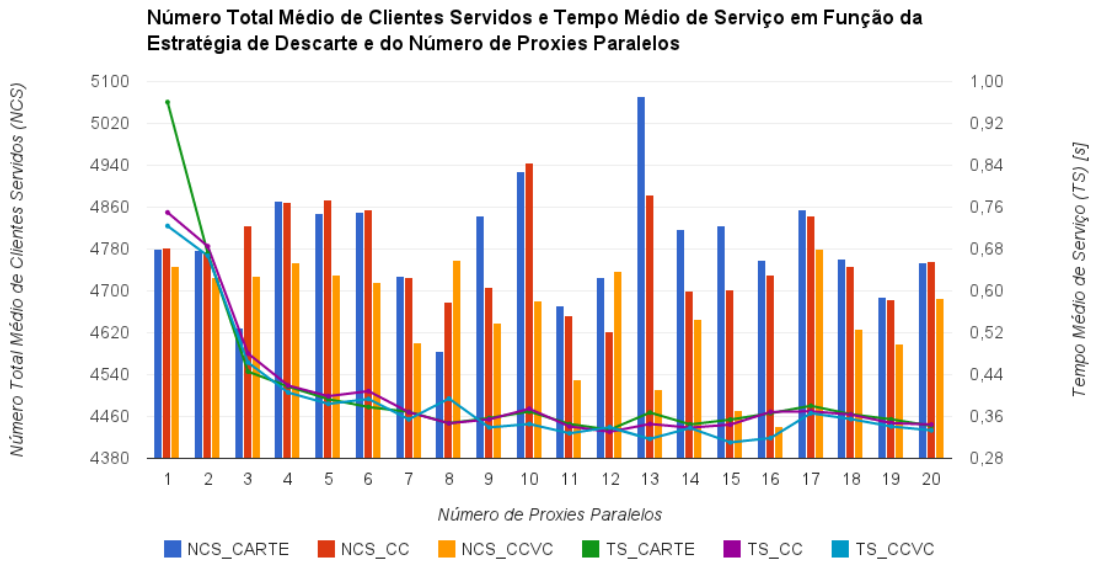
Tabela 5.11: Impacto do número de fluxos de execução servidos sobre o tráfego de escrita em memória do *proxy*.

Número médio de bytes escritos em memória em uma rodada de serviço			
NCS	Estratégias de Substituição		
	CARTE	CC	CCVC
1000	507945920	507842624	507818944
2000	1015686336	1015580800	1015559552
3000	1523362944	1523286784	1523289408
4000	2031121792	2031039424	2030996288
5000	2538713664	2538712384	2538717312
6000	3046649600	3046420288	3047329152
7000	3555052672	3554123200	3554199872
8000	4062021248	4061817600	4062023616
9000	4569507008	4569694848	4569530560
10000	5077166720	5077276864	5077277760

- Popularidade dos vídeos dada pela distribuição *Zipf*. Fator de dispersão: 0.270;
- Tamanho da Memória: 4000MB distribuída pelo número de *proxies* ativos;
- Taxa de codificação de vídeo de 8Mbps;
- *Offset* do início de requisição igual a 0 (início do vídeo);
- Limite de chegada de novas requisições igual a 2200 rodadas de serviço;
- Estratégias de substituição de blocos de vídeo: *CARTE* (tamanho do espaço de demanda: 10 blocos de vídeo), *CC* e *CCVC*;
- Número de *threads* variável;
- Distribuição de carga ativada;
- Limite de comunicação com o servidor principal desativado;
- Bloqueio de novos clientes devido à falta de recursos disponíveis desativado.

A tabela 5.12 apresenta o tempo de execução médio das rodadas de serviço de cada *proxy* utilizando a estratégia de substituição de memória *CARTE* em função do grau de paralelismo implementado.

Figura 5.6 – Número médio de clientes servidos e tempo médio de serviço em função do número de *proxies* paralelos e da estratégia de descarte.



Fonte: Arquivo pessoal.

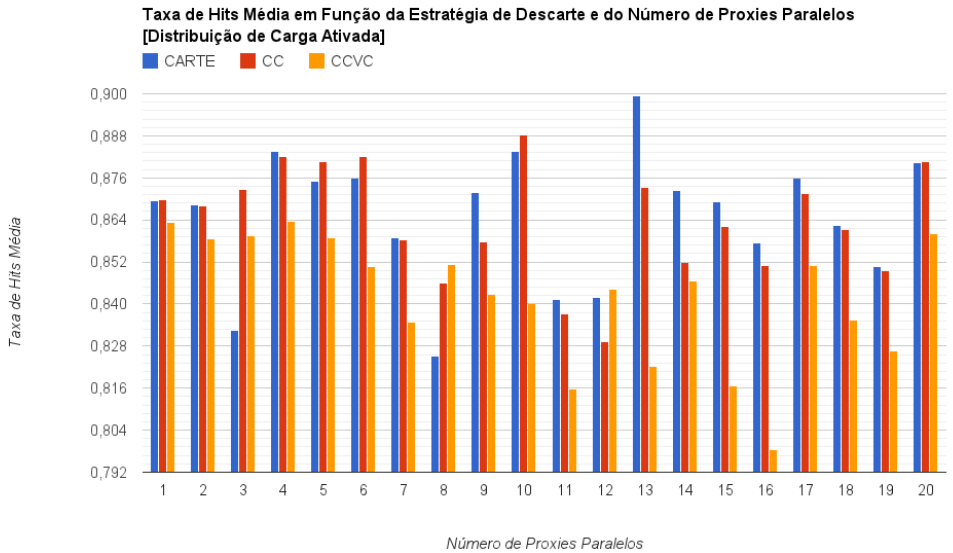
Como pode ser visto na Figura 5.6, que reúne os dados de tempo médio de serviço e número total médio de clientes servidos das três estratégias de descarte implementadas, o tempo de execução médio das rodadas de serviço para o CARTE decai até 0.346 segundos. Entretanto, se levarmos em consideração o número de clientes servidos e o tempo de serviço, identificamos que o melhor rendimento se dá para o CARTE quando o número de *threads* é igual a 13, 10 *threads* para o CC, e para a CCVC, 17 *threads*.

O aumento do número de *proxies* possibilitou uma taxa de *hits* de memória média máxima de 89,93% (13 *proxies*) em comparação aos 86,94% obtidos utilizando apenas 1 *proxy* para o *CARTE*, enquanto que para o *CC* a taxa média máxima foi de 88,85% (10 *proxies*) e 86,99% com 1 *proxy*. Todavia, a maior taxa de *hits* de memória média para o *CCVC* se deu com apenas 1 *proxy* ativo (86,32%) [FIgura 5.7].

Tabela 5.12: Impacto do número de *proxies* paralelos sobre o tempo de execução [CARTE].

Tempo de serviço médio da rodada de serviço (s)			
#*T*	Estratégias de Substituição		
	CARTE	CC	CCVC
1	0.960813	0.749955	0.724178
2	0.671816	0.685304	0.667283
3	0.445905	0.480590	0.463224
4	0.417253	0.419239	0.405937
5	0.393236	0.398882	0.384197
6	0.378108	0.408521	0.393515
7	0.368869	0.368324	0.354658
8	0.347289	0.347087	0.394376
9	0.357005	0.354618	0.339152
10	0.369540	0.374808	0.345756
11	0.345597	0.341051	0.327962
12	0.334817	0.330849	0.339605
13	0.367622	0.345754	0.317306
14	0.344865	0.338737	0.338104
15	0.354081	0.344543	0.310561
16	0.366336	0.368888	0.318876
17	0.380496	0.369959	0.366325
18	0.364344	0.363746	0.355091
19	0.354606	0.347521	0.341042
20	0.342291	0.344744	0.333603

Figura 5.7 – Taxa de *hits* de memória médio em função do número de *proxies* paralelos e da estratégia de descarte.



Fonte: Arquivo pessoal.

6 Resultados e Discussões

Observando os dados obtidos quanto a utilização do algoritmo *SA* como método de distribuição de carga entre os *proxies*, conseguimos identificar que este apresenta-se como uma ótima opção, tendo em vista seu custo computacional e o grau de satisfação quanto a distribuição normalizada obtida. Uma vez que o tempo de execução do algoritmo de distribuição de carga é bem inferior quando comparado ao limite de um segundo imposto para cada rodada de serviço, podemos garantir que o mesmo não impactuará no desempenho geral dos *proxies*. Em adição, com a redistribuição da carga através dos *proxies*, conseguimos reduzir a variância da distribuição de carga entre os mesmos, possibilitando que o sistema sirva um número maior de clientes simultaneamente sem estourar o tempo da rodada de serviço quando comparado ao mesmo modelo sem distribuição de carga. Entretanto, o grau de granularidade do algoritmo de distrobuição de carga não é capaz de distribuir a carga uniformemente entre todos os nodos quando a popularidade de alguns vídeos é muito maior que a popularidade média dos vídeos do acervo.

Adicionalmente, a implementação da rotina de distribuição de carga e memória permitiu um aumento da taxa de *hits* de memória, através da redução do número de vídeos populares servidos por uma mesma *thread*, distribuindo-os através de *threads* responsáveis por vídeos menos requisitados, reduzindo a concorrência direta por memória destes vídeos, descartando-se as sequências de vídeo alocadas de vídeos menos populares, aumentando o rendimento do sistema.

A recodificação das rotinas de execução do *proxy* e do simulador, utilizando de maneira mais adequada as estruturas internas, os modelos de computação e o fluxo de execução das rodadas de serviço, possibilitou uma melhoria na performance do algoritmo implementado, reduzindo o tempo de serviço das rodadas de serviço, a redução do número de instruções executadas e do número de ciclos de *CPU*.

A capacidade de estimar o tempo de serviço e o consumo de *CPU* e de memória de acordo com o número de clientes servidos é de grande importância, uma vez que permite-nos reservar recursos e prever o comportamento do sistema a longo prazo. Além disso, os dados

de tráfego de memória possibilitam identificar os possíveis gargalos criados devido à vazão máxima de memória (que é menor que a vazão de processamento total da *CPU*) quando um número maior de *threads* forem empregadas e o número de clientes servidos aumentar.

Os resultados obtidos com a variação do grau de paralelismo empregado pelo sistema exibiu a capacidade dos *proxies* de aumentarem o número de clientes servidos simultaneamente, reduzindo o tempo de serviço necessário para servir os clientes ativos, através do aumento do número de requisições tratadas paralelamente. Além de reduzir o tempo de serviço e de aumentar o rendimento do sistema, também fomos capazes de reduzir o número de requisições realizadas ao servidor principal, reduzindo o tráfego de dados através do *link* de comunicação entre os *proxies* e o servidor.

7 Conclusão

Este trabalho de conclusão de curso descreveu a implementação de um gerenciador dinâmico de memória de um *proxy* de *VoD*, complementado com o desenvolvimento de um ambiente de simulação para validar o software desenvolvido. Foram implementados quatro módulos em linguagem C: um gerenciador de memória de um *proxy* de *VoD*, um simulador de *proxies* de *VoD*, um gerador e editor de arquivos de simulação. Além disso, foi implementado um módulo utilizando a *API* de *profiling* disponibilizada pela Intel, *PCM*, para extrair dados de utilização de *CPU* e memória. Em adição, foram desenvolvidos *scripts* em linguagem *Python* para geração de gráficos dos dados de simulação exportados pela ferramenta.

Dentre os resultados obtidos apresentamos a implementação com sucesso do gerenciador dinâmico de memória paralelo do *proxy* de *VoD*, que apresenta um alto grau de configurabilidade, permitindo a adaptação do algoritmo para execução de diferentes cenários, como diferentes estratégias de descarte de vídeo, implementação de limite de banda de comunicação entre os *proxies* e o servidor de vídeos, a habilidade de distribuir a carga de clientes através dos nodos ativos, da variação do espaço de demanda pertencente aos *proxies* de acordo com a carga de serviço submetida, a implementação de rotinas de extração de dados de execução, entre outros. Em adição, desenvolvemos com sucesso o simulador multiplataforma de *proxies* de *VoD*, possibilitando verificar o desempenho do sistema sobre diferentes arquiteturas.

No decorrer do desenvolvimento do trabalho, conseguimos identificar os ganhos obtidos entre as versões parcial e final implementadas, cujo aumento de desempenho foi visível. Em adição, apresentamos a importância do algoritmo de distribuição de carga e de distribuição de memória implementado, que permitiram o aumento do rendimento e da produtividade dos *proxies*, através da normalização da carga de serviço entre as *threads*, de maneira a aumentar a quantidade de trabalho líquido suportada pelo sistema, além de aumentar a taxa de *hits* de memória, reduzindo o tráfego de comunicação com o servidor principal.

Ainda existe muito trabalho pela frente, tendo em vista a grande área de pesquisa que o assunto engloba, tais como, algoritmos de distribuição de carga e de memória, algoritmos

de substituição de memória, estratégias de paralelismo, entre outros. Em especial, sugere-se como pontos importantes para o prosseguimento do trabalho, a incorporação na estratégia *CARTE* de um mecanismo de adaptação dinâmica do espaço de demanda utilizado, que seja capaz de aumentar ou diminuir a região de cálculo da demanda das sequências conforme a disposição dos clientes no decorrer da linha temporal do vídeo, de maneira que o algoritmo seja capaz de identificar a disposição e os padrões de acesso dos clientes, buscando a redução do número de requisições realizadas ao servidor principal; a adição da capacidade dos *proxies* de aumentarem ou diminuírem o número de *threads* paralelas dinamicamente conforme a demanda existente, a fim de reduzir a taxa de ocupação da *CPU* e, em uma fase posterior de mapemento do sistema para hardware, da redução do consumo energético pela desativação dos núcleos ociosos; a busca por estratégias que maximizem a utilização da unidade de *DMA*, reduzindo a carga de execução sobre os processadores e desviando o fluxo de transferência de dados através do canal direto a partir da memória; a adicão da capacidade dos *proxies* de lidarem com vídeos de diferentes tamanhos e diferentes taxas de codificação, a fim de agregar mais recursos ao sistema; a capacidade dos *proxies* de adiantarem a execução de rodadas de serviço futuras, utilizando o tempo de serviço que resta de uma rodada de serviço atual, com o intuito de livrar a carga do sistema em momentos de pico de execução, onde o sistema não seria capaz de vencer a demanda em condições normais; e a modificação do tamanho do *buffer* do fluxo de alocação em relação ao fluxo de execução, afim de garantir certa *QoS*, nos casos onde a taxa de *hits* é baixa.

Referências Bibliográficas

ASCHER, D. et al. *Numerical Python*. Ucrl-ma-128569. Livermore, CA, 1999.

BUTENHOF, D. R. *Programming With Posix Threads*. Addison-Wesley, 1997. (Addison-Wesley Professional Computing Series). ISBN 9780201633924. Disponível em: <http://books.google.com.br/books?id=_xvnuFzo7q0C>.

CERNÝ, V. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, Springer Netherlands, v. 45, n. 1, p. 41–51, jan 1985. ISSN 0022-3239. Disponível em: <http://dx.doi.org/10.1007/BF00940812>.

CHANDRA, P. K.; SAHOO, B. Performance analysis of load balancing algorithms for cluster of video on demand servers. In: *Advance Computing Conference, 2009. IACC 2009. IEEE International*. [S.l.: s.n.], 2009. p. 408–412.

CHEN, H. et al. A new proxy caching scheme for parallel video servers. In: *Proceedings of the 2003 International Conference on Computer Networks and Mobile Computing*. Washington, DC, USA: IEEE Computer Society, 2003. (ICCNMC 03), p. 438. ISBN 0-7695-2033-2. Disponível em: <http://dl.acm.org/citation.cfm?id=950787.950997>.

DEMENTIEV, R.; WILLHALM, T.; BRUGGEMAN, O. *Intel Performance Counter Monitor - A better way to measure CPU utilization*. 2012. Disponível em: <http://software.intel.com/en-us/articles/intel-performance-counter-monitor>.

HAESER, G.; RUGGIERO, M. G. Aspectos teóricos de simulated annealing e um algoritmo duas fases em otimização global. *TEMA-Tendências em Matemática Aplicada e Computacional*, v. 9, n. 3, p. 395–404, 2011.

HONG, D.; VLEESCHAUWER, D. D.; BACCELLI, F. c. A chunk-based caching algorithm for streaming video. In: *NET-COOP 2010 - 4th Workshop on Network Control and Optimization*. Gent, Belgique: [s.n.], 2010. Session 05 : Streaming applications. Disponível em: <http://hal.inria.fr/inria-00597186>.

HSU, T.-H.; LI, Y.-H. A weighted segment-based caching algorithm for video streaming objects over heterogeneous networking environments. *Expert Syst. Appl.*, Pergamon Press, Inc., Tarrytown, NY, USA, v. 38, n. 4, p. 3467–3476, April 2011. ISSN 0957-4174. Disponível em: <http://dx.doi.org/10.1016/j.eswa.2010.08.134>.

HUNTER, J. D. Matplotlib: A 2d graphics environment. *Computing in Science and Engg.*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 9, n. 3, p. 90–95, may 2007. ISSN 1521-9615. Disponível em: <http://dx.doi.org/10.1109/MCSE.2007.55>.

ISHIKAWA, E. *Memoria Cooperativa para Distribuicao de Video sob Demanda*. Tese (Doutorado) — Universidade Federal do Rio de Janeiro, 2003.

KERNIGHAN, B. W.; RITCHIE, D. M. *The C Programming Language*. Second edition. Englewood Cliffs, New Jersey: Prentice-Hall, 1988.

KIRKPATRICK, S.; GELATT JR., . C. D.; VECCHI, M. P. Optimization by simulated annealing. *Science*, v. 220, n. 4598, p. 671–680, 1983. ISSN 0036-8075.

LI, J.; CHEN, Z. Sliding-window caching algorithm for streaming media server. In: *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*. New York, NY, USA: ACM, 2009. (ICIS 09), p. 1152–1159. ISBN 978-1-60558-710-3. Disponível em: <http://doi.acm.org/10.1145/1655925.1656135>.

LIU, J.; XU, J. A survey of streaming media caching. *IEEE Communications Magazine*, v. 42, n. 8, p. 88–94, August 2004.

MOGHAL, M. R.; MIAN, M. S. Effective load balancing in distributed video-on-demand multimedia system. In: *Multi Topic Conference, 2003. INMIC 2003. 7th International*. [S.l.: s.n.], 2003. p. 164–169.

ROSSUM, G. v. *Python reference manual*. P. O. Box 4079, 1009 AB Amsterdam, The Netherlands, April 1995. ii + 54 p. Disponível em: <http://www.python.org/doc/ref/ref-1.html>.

SO-IN, C. A survey of proxy caching mechanisms for multimedia data streams. 2005. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.128.1921>.

SOUTO, Á. A.; CAVALCANTI, D. B.; MARTINS, R. P. *Um plano nacional para banda larga: o Brasil em alta velocidade. Ministério das Comunicações*. 2010. Disponível em: <http://www.mc.gov.br/wp-content/uploads/2009/11/o-brasil-em-alta-velocidade1.pdf>.

TROSSET, M. What is simulated annealing? *Optimization and Engineering*, Springer Netherlands, v. 2, n. 2, p. 201–213, 2001. Disponível em: <http://dx.doi.org/10.1023/A:1013193211174>.

VENKATRAMANI, C. et al. Optimal proxy management for multimedia streaming in content distribution networks. In: *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*. New York, NY, USA: ACM, 2002. (NOSSDAV '02), p. 147–154. ISBN 1-58113-512-2. Disponível em: <http://doi.acm.org/10.1145/507670.507691>.

VINAY, A. et al. A comparative analysis of centralized and distributed dynamic load balancing algorithms for cluster based video-on-demand systems. In: *Proceedings of the International Conference 38; Workshop on Emerging Trends in Technology*. New York, NY, USA: ACM, 2011. (ICWET 11), p. 351–356. ISBN 978-1-4503-0449-8. Disponível em: <http://doi.acm.org/10.1145/1980022.1980099>.

WU, K.-L.; YU, P. S.; WOLF, J. L. Segment-based proxy caching of multimedia streams. In: *Proceedings of the 10th international conference on World Wide Web*. New York, NY, USA: ACM, 2001. (WWW 01), p. 36–44. ISBN 1-58113-348-0. Disponível em: <http://doi.acm.org/10.1145/371920.371933>.

WU, T. et al. Reuse time based caching policy for video streaming. In: *Consumer Communications and Networking Conference (CCNC), 2012 IEEE*. [S.l.: s.n.], 2012. p. 89–93. ISSN Pending.

YU, J. et al. A dynamic caching algorithm based on internal popularity distribution of streaming media. *Multimedia Systems*, Springer, p. 135–149, oct 2006. ISSN 0942-4962. Disponível em: <http://dx.doi.org/10.1007/s00530-006-0045-x>.