# 4.
# Otimização em Nível Gerencial

Num processo de produção de cenas por meio de computação gráfica, pode-se perceber que o renderizador é capaz de entender apenas o processo da produção de apenas um frame no tempo. Nesse contexto, ele não pode avaliar a forma de melhor distribuir as cargas de trabalho e realizar o acompanhamento de todos os outros renderizadores.

Essa necessidade é atendida por um gerente de renderização, que é um sistema que irá coordenar todo o processo de produção da cena ou *take* e tomar iniciativas caso haja problemas.



### 4.1.1. Análise de Renderização

A renderização de uma cena ou *take* consiste em produzir uma sequência de *frames* que foram especificados previamente por um modelo 3D e um conjunto de metadados. De forma simplista, o processo de gerenciamento da renderização atual consiste em distribuir $N$ frames para serem produzidos por $M$ servidores (*nodes*), onde $N > M$. Isso estabelece a primeira restrição na qual o gerenciador deve cuidar, a restrição de recursos; assim, uma mesma máquina deve processar mais de um *frame* para a completude da cena.

Tal fato pode ser realizado de diversas maneiras, sendo que, essencialmente, todos os gerenciadores associam sequencialmente o processamento de cada *frame* a um *node* (ou nó), como uma fila. Assim, cada *node* acessa a fila de *frames* e inicia o processamento e, ao finalizar o trabalho, ele acessa novamente o conjunto de *frames* remanescente na

fila (figura 33) e continua o trabalho até extinguir todos os *frames*. Essa forma de gerenciamento tem a vantagem de manter todos os *nodes* ocupados e *nodes* mais potentes poderem processar mais *frames*, além de ser de simples gerenciamento.

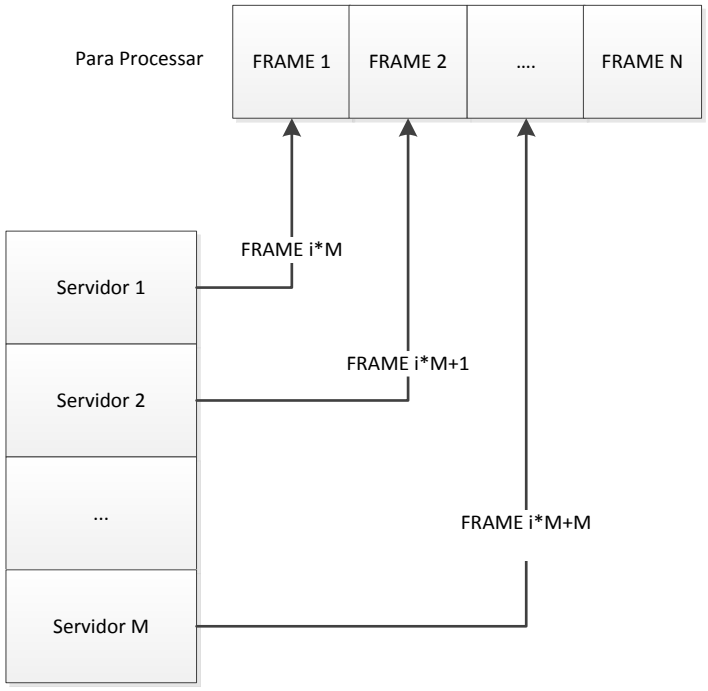


**Figura 33:** Processo convencional de distribuição de frames

No entanto, esse modelo é ineficiente, pois desconsidera totalmente as questões temporais da renderização, tornando o processo muito mais custoso e necessitando de maior tempo de geração de *caches*. Isso impacta diretamente na eficiência do renderizador, pois o mesmo fica sem referência de dados anteriores ou posteriores, além de ser muito baixa a probabilidade de um *node* renderizar *frames* próximos. Dessa forma, toda a otimização da renderização para a coerência temporal é perdida, e ainda as maquinas não são selecionadas de acordo com suas capacidades e muito menos são mapeados os *frames* de maior custo em máquinas de maior capacidade.

### 4.1.2. Inventário de Nós

A maioria dos gerenciadores utiliza o conceito de inventário de nós (*nodes*), que consiste em uma tabela descritiva de cada servidor disponível para atender uma demanda de renderização. No contexto geral, ao ser adicionado ao gerenciador, cada servidor deve passar um conjunto de informações a respeito de suas capacidades e equipamentos integrados. Uma vez adicionado, o gerenciador confia nesse parâmetro e pode ou não utilizá-lo para atender uma demanda.

O gerenciador proposto utiliza a mesma ideia de inventário, no entanto, considera-se que o perfil de capacidades do nó é dinâmico, ou seja, deve ser medido periodicamente. Além disso, alguns dados só podem ser mensurados durante a efetiva utilização do equipamento (transferências de memória, transferência em disco, transferências na rede), isso devido a dinamicamente os elementos mudarem suas características para melhor acomodar a demanda.

Um exemplo é a transferência de dados, pois durante a carga de dados, o *Storage* que atende ao conjunto de servidores sofre um pico de requisições e, normalmente, reduz a capacidade de transferência. Além disso, a cópia de dados na rede pode ser feita por diversos protocolos, como CIFS, NFS, FTP e HTTP, no entanto, cada servidor e cada sistema operacional possui uma resposta de performance diferente para cada protocolo, assim, o gerenciador deve perceber o custo associado a essas operações.

Para realizar a medição desses parâmetros, foi feito o uso dos contadores de performance do Microsoft Windows e Linux, sendo que nesse último, os dados são obtidos através da plataforma Mono (Mono, 2001), apesar de ser pouco explorado nesse trabalho (apenas nos testes com VCA e Embree utilizando Xeon Phi). Dessa maneira, periodicamente os nós enviam seus contadores para o gerenciador através do sistema de mensagens do mesmo (REST).

### 4.1.3. Arquitetura de Renderização em Cluster

Na maioria dos estúdios de produção de conteúdo virtual, os clusters de renderização, ou *Render Farms*, são muito comuns e utilizados, sendo o escalonamento de uso muito complexo, devido à alta demanda. Para atingir a melhor performance possível, os servidores não possuem virtualizações internas, e normalmente são máquinas de altíssima capacidade. Outra característica desses *clusters* é o conceito de geração e grupo.

O processo de compra desses servidores é, normalmente, realizado periodicamente, diferindo apenas alguns anos entre as compras. Assim, num mesmo *cluster*, existem diversas gerações de máquinas, com capacidades e funcionalidades diferentes. A consequência disso é a disparidade de performance de cada um desses servidores, ou seja, a consideração de que cada máquina do *cluster* pode atender os processos da mesma forma é inválida, tanto que, em muitos casos, gerações anteriores não conseguem renderizar cenas mais atuais devido à restrição de memória e outros problemas.

Esses problemas não são observados por todos os gerenciadores, apenas o sistema do Deadline[6] se previne quando observa uma máquina muito obsoleta, porém, essa observação só se dá de forma estática, ou seja, dado uma versão de um renderizador, o mesmo verifica as recomendações mínimas para atender. Nos demais gerenciadores, isso não é considerado, permitindo que máquinas incapazes de atender ao processo sejam utilizadas.

O custo desse descuido é a perda de tempo e recursos da renderização, pois uma vez que a máquina inicia o processo de renderização existem apenas três formas de finalização: término da renderização (*exit code* igual a 0); estouro de tempo de renderização (tempo escolhido para indicar que o processo de renderização está

[6] Deadline (Thinkbox, 2007) é um toolkit para administração de clusters de renderização.

paralisado ou sem ação); e falha na renderização (*exit code* diferente de 0). Assim, caso haja estouro do tempo ou falha na renderização, o *frame* é reposto na cabeça da fila, o que pode levar muitos minutos para acontecer, e, após o erro, essa máquina pegará outro *frame* que não será capaz de renderizar novamente.

Uma forma de resolver o problema é criar grupos e solicitar ao utilizador do gerenciador a escolha dos grupos que devem atender a demanda solicitada. No entanto, o utilizador não sabe se os servidores menos potentes atenderão à demanda, assim, ele terá que testá-los para saber. Um comportamento comum é a não utilização de servidores mais antigos, ou seja, o utilizador sempre escolhe os grupos de maior performance, deixando os de menor poder computacional ociosos.

Nesse contexto, elaborou-se a arquitetura do gerenciador como apresentado na figura 34.

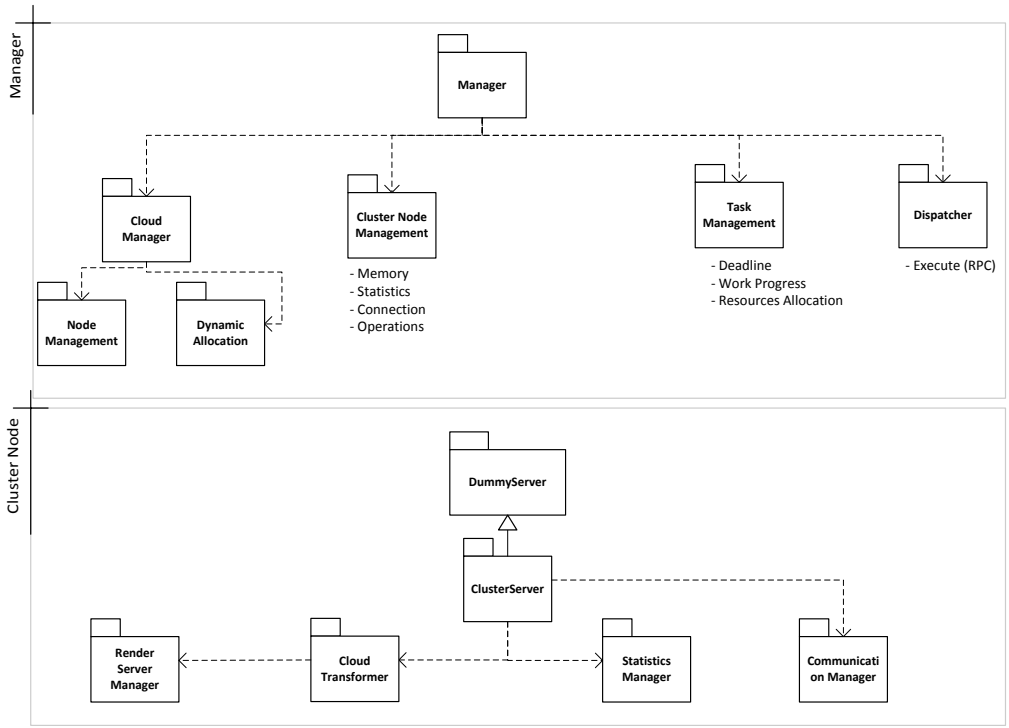


**Figura 34 :** Arquitetura de alto nível do Gerenciador e do Sistema de Controle dos Nós (imagem produzida)

A arquitetura proposta necessita de um serviço rodando no servidor de trabalho (*Cluster Node*). Esse serviço é o responsável por se comunicar e gerar os parâmetros de performance e controle para o servidor de gerenciamento (*Manager*). Essa comunicação utiliza o protocolo HTTP por meio de mensagens REST. O módulo de serviço dos nodes (*DummyServer*) é extremamente simples, cabendo ao mesmo apenas a comunicação inicial com o gerenciador. Nesse processo, ao se comunicar com o manager, o *DummyServer* recebe um arquivo *DLL* para ser copiado localmente através de HTTP. Esse arquivo é o *assembly* do servidor local que irá executar todo o controle necessário para atender ao gerenciador. Após o *download*, o arquivo é carregado como um *Assembly* e, através dos métodos de reflexão do C#, a interface é carregada e passa a assumir o controle das instruções do gerenciador.

Esse procedimento é necessário para manter todos os nós atualizados e de acordo com funcionamento do gerenciador, permitindo que seja alterado o sistema com ele ainda em operação. Isso é importante devido ao custo de parada e configuração de grandes clusters, bem como o uso em ambientes virtualizados remotos, como as nuvens. Todos os sistemas de gerenciamento de rendering analisados, na indústria e na literatura acadêmica, não realizam esse procedimento.

Uma vez carregado, o gerenciador local (*ClusterNode*) realiza o envio das informações como nome da máquina, endereços IPs, tipos de placas de rede, processadores, placas de vídeo, memória, capacidade de disco e dados do sistema operacional, como tipo, versão e updates realizados. Após essa comunicação inicial, o serviço inicia um processo de envio de informações de performance para o gerenciador (uso de *CPU*, memória disponível, banda de memória e outros). Ele ainda dispõe do caminho em disco e informações do renderizador, bem como é capaz de se comunicar com o mesmo usando a linha de comando ou o acesso de administração, que consiste em uma *thread* no renderizador que se comunica por meio de REST/HTTP com o serviço local.

### 4.1.4. Gerenciador

Todos os nós se comunicam com um servidor de controle central (*Manager*) que faz todo o controle da renderização. A estrutura interna do mesmo foi desenvolvida em C#.NET 4.5 e toda a comunicação é feita por pacotes REST/HTTP.

Para simplicidade de uso, o mesmo conta com um *WebSite* que apresenta uma interface de controle e submissão de dados (figura 35).

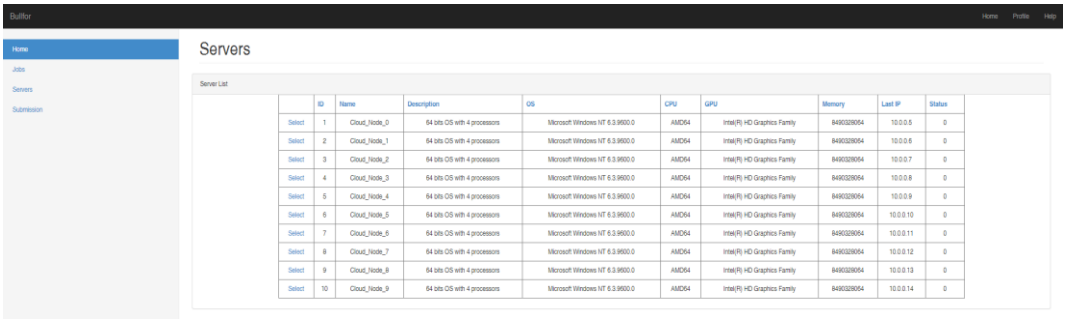


**Figura 35:** Interface Web do Gerenciador (imagem do sistema)

Na interface é possível visualizar as medições de cada servidor e o progresso do trabalho submetido. Internamente, os dados são armazenados em um banco de dados SQL Server ou SQL Azure, com uma estrutura de dados como a da figura 36.

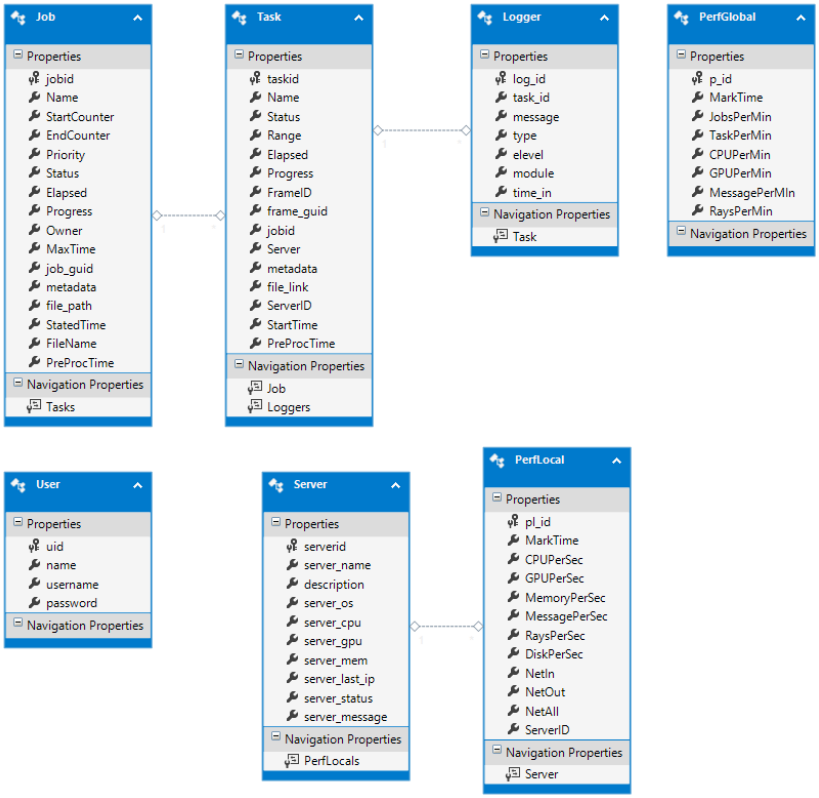


**Figura 36:** Estrutura de Dados do Servidores SQL (imagem produzida)

Através dessa interface é possível submeter uma tarefa para o gerenciador, especificando os campos da figura 37. Observa-se que o arquivo com os dados deve ser enviado, ou utilizar utilizar apenas o caminho de rede dos elementos de cena (nos casos em que há acesso direto ao *Storage*).

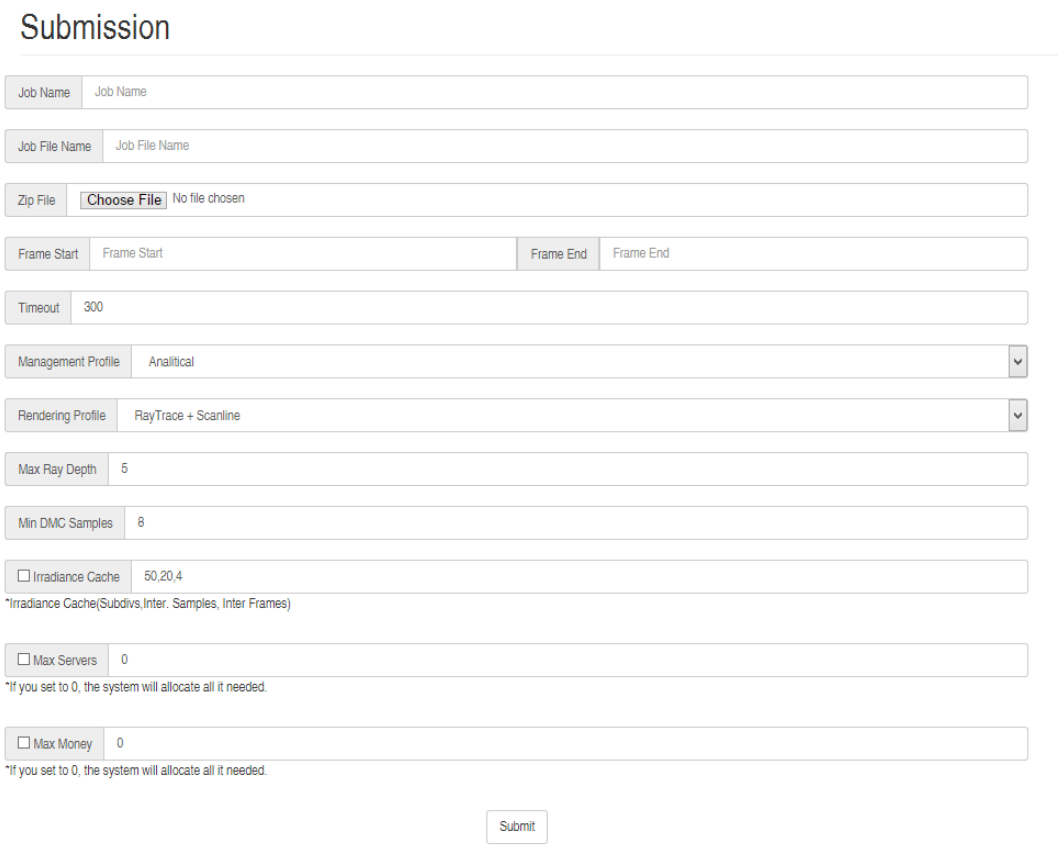


**Figura 37:** Campos de Submissão (imagem do sistema)

Uma vez enviado o material, o gerenciador inicia o processo de renderização de *frames*.

### 4.1.5. Processo de Inicialização de Renderização

Considerando que o tempo de renderização total é muito maior que o tempo da renderização em tempo real, o gerenciador inicia a renderização fazendo uma solicitação de dados para todos os *frames*, essa solicitação utiliza o sistema de filas comum dos gerenciadores. A figura 38 apresenta o fluxo de execução da inicialização da renderização.

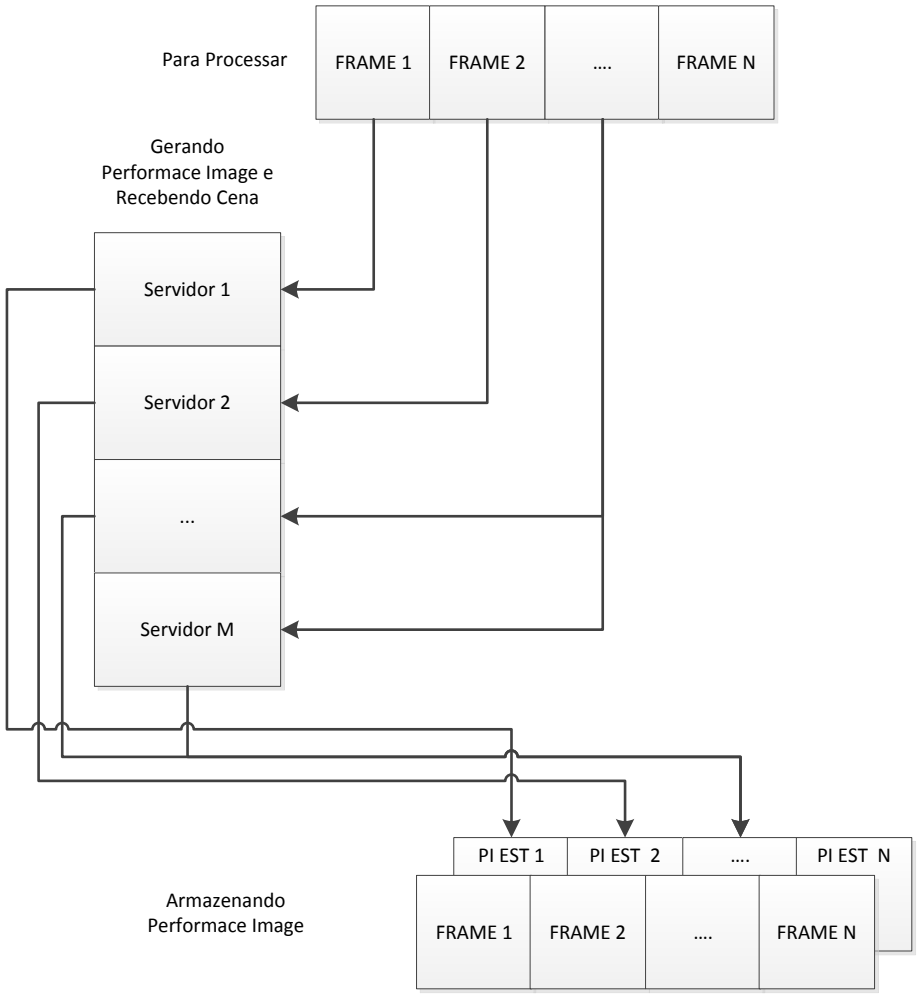


**Figura 38:** Inicialização da renderização de cenas (imagem produzida)

Percebe-se que cada nó realiza uma rasterização rápida para produzir a *Performance Image* estimada de cada nó, bem como o canal de velocidade de cada *frame*, esse processo é extremamente rápido. Essa etapa é executada pelo renderizador, através do comando *estimate* enviado via REST, que calcula as estimativas da renderização.

Uma vez recebida a estimativa de um *frame*, calcula-se o custo de memória estimado, que utiliza uma heurística simples, computando-se a quantidade de dados de texturas e modelos e combinando com as *Performace Images* (PI) (figura 19, capítulo 3). Assim, sabendo o nó que executará um determinado *frame*, o gerenciador realiza uma busca pelo *pixel* que possui o maior número de triângulos projetados $\boldsymbol{n_t}$ e outra busca

com o *pixel* de maior número de materiais utilizados $n_m$. Assim, uma estimativa para o consumo de memória é apresentada pela equação 12.

$$c_{memory}(f) = k_{est}\left[\sum_{i=0}^{n_m} c_{material}(i) \cdot sizeof\big(mat(i)\big) + n_t \cdot sizeof(Triangle)\right]$$

**Equação 12:** Estimativa de Consumo de Memória

Na equação 12, a função *sizeof* computa a quantidade de memória consumida pelo material, através da soma dos tamanhos dos mapas do material, e $k_{est}$ é um fator de correção, visto que a conta se trata de uma heurística para o real custo, o valor utilizado é 1,25, ou seja, 25% de erro.



### 4.1.6. Distribuição sem restrição temporal

Uma vez abastecido pelas estimativas do que será executado, o gerenciador inicia o processo de planejamento da renderização, levando em contas as premissas:

- Frames de maior custo devem ser executados por nós de maior poder de processamento.
- A coerência espacial deve ser utilizada.
- *Nodes* sem condições de atendimento de demanda devem ser descartados.

Nesse caso o gerenciador deve se comportar como os demais gerenciadores, garantindo o atendimento a todos os *frames*. No entanto, a distribuição é feita na forma de blocos, ou seja, cada nó deverá renderizar $n$ *frames* em sequência e depois será reescalado para atender mais *frames*, assim, quem decide quem fará quais *frames* é o gerenciador.

Para isso, o mesmo cria uma tabela de custo de renderização estimada (bloco de *frames*) e a ordena decrescentemente por custo. Da mesma forma, os nós são colocados em uma tabela ordenada pela

capacidade dos mesmos (inicialmente é usada a expressão: $Fator = Clock \cdot n_{cores}$), e é analisado se um nó possui requisitos mínimos para a renderização do *frame* (capacidade de memória, versão do renderizador, tipo de sistema operacional e acesso a pastas de rede). O próximo passo é a categorização de capacidades de cada *frame*, agregando *frames* que podem ser renderizados por *GPU* e *frames* que podem ser renderizados por *CPU* (figura 39).

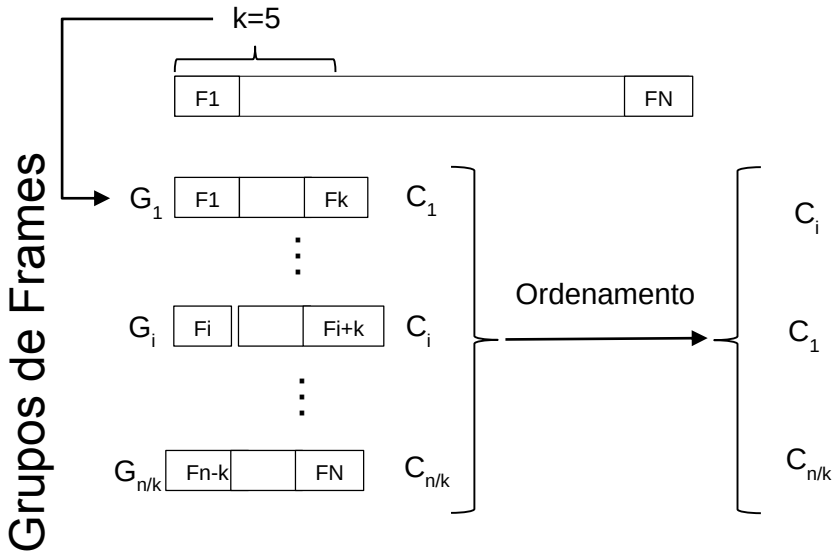


**Figura 39:** Classificação e categorização de frames (imagem produzida)

Nessa etapa, o gerenciador atribui os servidores de maior capacidade aos *frames* de maior custo, até preencher a associação de todos os servidores. Observa-se que os *frames* são produzidos não seguindo a ordem temporal e sim a ordem de custo, porém o sistema restringe a atribuição dos *frames*, sendo que um *frame f*, pertencente a um grupo *g*, só pode ser alocado para renderização se o *frame 0.2f* já estiver concluído (a menos que máquinas fiquem ociosas por essa decisão), ou seja, seja *f* = 100**,** f só pode ser iniciado se o *frame* 20 já estiver pronto.

Esse teste não é aplicado para os 20% dos *frames* iniciais. Isso se deve devido a uma observação de uso, pois foi percebido que ter uma sequência de *frames*, mesmo com alguns *frames* faltantes, da sequência solicitada é importante para a avaliação artística da animação, do contrário, essa avaliação necessita esperar a completude do renderização. Todavia, isso pode ser removido caso a pré-visualização não seja necessária.

Durante a execução da renderização os *frames* são executados sempre em números ímpares maiores do que um. Assim, decidiu-se que o sistema trabalha com blocos de cinco *frames*, sendo a ordem de renderização como mostrado na figura 40. O número de frames por bloco e a ordem foram estabelecidos arbitrariamente, baseados na experiência do autor observando o processo de geração.

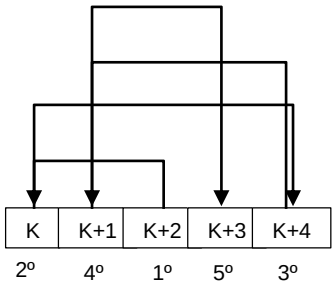


**Figura 40:** Ordem de execução do bloco de frames (imagem produzida)

Pode-se observar que a geração segue uma forma similar à busca binária. Isso é feito para melhorar as estimativas da *Performance Image* (PI) para o renderizador, de forma a utilizar da melhor maneira possível o aproveitamento e custos e compensação de *Performance Image*, discutido no capítulo 3.

Além disso, o gerenciador notifica os nós com os *frames* que já foram concluídos; assim, um nó pode utilizar a PI produzida por outro nó

que seja próxima. *Frames* com distâncias maiores que 10 (i.e, o dobro do número de frames) não são utilizados para compensação de PI.

Conforme os *frames* são finalizados, o gerenciador carrega as PIs e suas compensações entre *frames* e atualiza a estimativa de tempo total da renderização, bem como o custo dos *frames* e a capacidade mínima requerida por *frame*. Nesse processo a tabela ordenada dos custos dos *frames* é recomputada, bem como a tabela de capacidade dos nós (através dos dados de performance periodicamente enviados pelo *ClusterNode*), que nesse caso utiliza a equação 13.

$$Fator = Clock \cdot n_{cores} \cdot medium(CPUUsage) \cdot \left(\frac{1}{t_{memory}} + \frac{1}{t_{disk}} + \frac{1}{t_{network}}\right) \cdot \frac{t_{total}}{3}$$

**Equação 13:** Estimativa de performace dos servidores

No novo cálculo, os sistemas que apresentarem os melhores tempos de transferência são contemplados com um valor adicional à performance de transferência de dados e é compensado o uso do processador. De forma similar, são distribuídos os *frames* remanescente até a completude da renderização (figura 41).

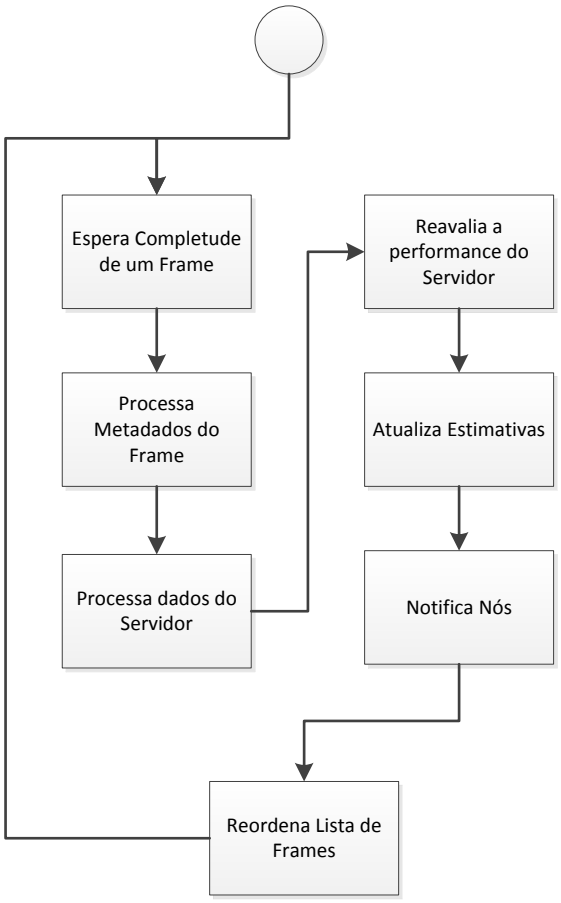


**Figura 41:** Fluxo de gerenciamento dos *frames* (sem restrições) (imagem produzida)

### 4.1.7. Distribuição com restrição temporal

O gerenciador pode utilizar suas estimativas para computar o tempo remanescente para o processamento dos *frames* através da computação do custo do *frame* com a potência do nó atribuído[7]. Assim, é possível aplicar uma restrição ainda mais complexa, a restrição temporal, nenhum gerenciador estudado apresentou uma solução para essa restrição, além disso, deve-se entender que a proposta é o atendimento de melhor esforço, ou seja, o renderizador tentará atender da melhor maneira possível. Nesse caso, existem duas restrições, a temporal e de recursos, uma vez que não é possível aumentar a capacidade de um *cluster* de

[7] No capítulo o cálculo de custo e tempo de renderização é feito por *core* e por GHz.

forma simples. O gerenciador pode utilizar grupos extras de máquinas, como workstations, dessa forma, as premissas são alteradas para:

- *Frames* de maior custo devem ser executados por nós de maior poder de processamento.
- A coerência espacial deve ser utilizada.
- Deve se utilizar o maior número de nós, se houver necessidade, o mesmo possuir compatibilidade de sistema e condições mínimas de execução.
- O tempo total deve atender o tempo solicitado quando a solicitação for suficiente para o refinamento das estimativas e otimizações dentro da qualidade mínima.

Nessas novas premissas, o fato de se utilizar o maior número de nós exclui os nós que não possuem versões compatíveis de renderizador e sistemas operacionais, mas inclui restrições de *hardware* como memória. Outra observação é que deve haver tempo suficiente para que o gerenciador possa ajustar as estimativas e tomar novas providências para atender o tempo de renderização, por exemplo, o tempo de renderização deve ser maior que o tempo de renderização do *frame* de maior custo.

Existem dois contextos da restrição imposta, quando o número de *frames* é menor que o número de servidores disponíveis e quando o número de *frames* é maior ou igual.

No primeiro contexto, o gerenciador inicialmente calcula o tempo total para a renderização da tarefa, atribuindo os *frames* aos nós ordenados (nós que não possuem problemas de memória). Caso ainda haja servidores, os *frames* remanescentes passam por um processo de divisão, seguindo o custo do mesmo. Essa divisão utiliza o algoritmo de BSP (*Binary Space Subdivision*) (figura 42b), traçando uma reta que divide o custo igualitariamente, isso gera duas mascaras de renderização (figura 42c). Esse processo continua até o número de subimagens atenda

a todos os servidores remanescentes (figura 42a). Os nós remanescentes são então ordenados e é computado o custo dos subframes.

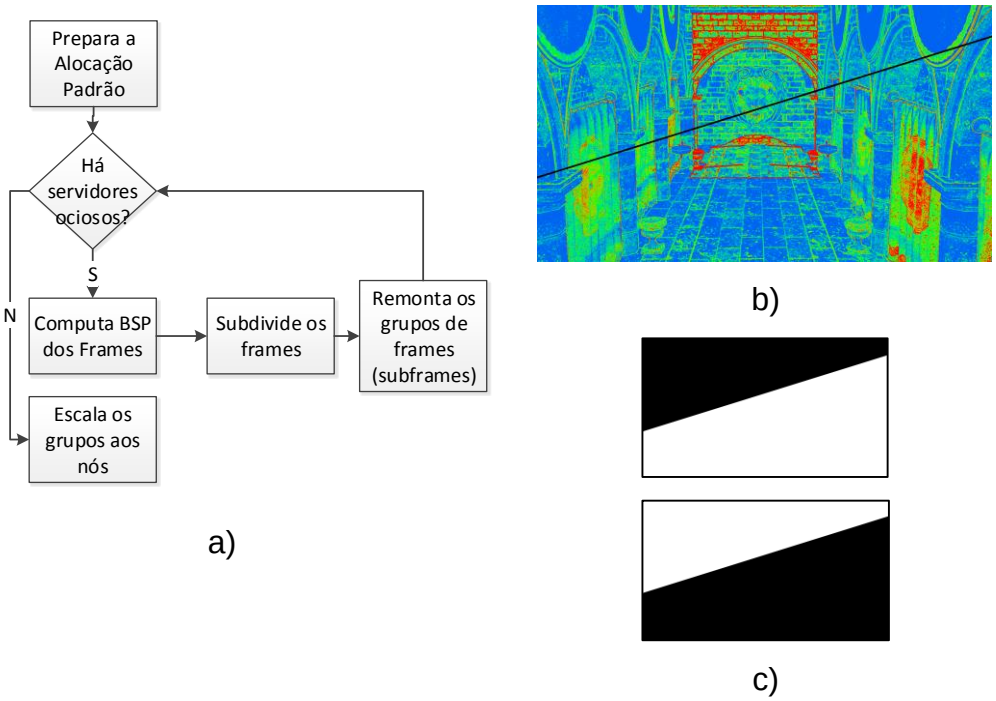


**Figura 42:** a) Esquemático do algoritmo de subdivisão de frame. b) Imagem de Performance sendo subdividida pelo BSP. c) Máscaras de renderização (branco habilita renderização, preto impede renderização)

O *subframe* é renderizado assumindo um *frame* retangular (figura 42b), no entanto, todo bloco da *quadtree* que pertencer integralmente à área não pertencente ao *subframe* em renderização, ou seja, pertence a parte preta da máscara, é descartado (não computado). Porém há uma pequena área em comum, 10 pixels de diâmetro na borda da divisão do BSP, a qual é usada para compor as imagens usando um operador *Blend*. Quando um *subframe* é finalizado, sua composição é feita pelo gerenciador, no módulo de gerenciamento de nós (operações). Essa estratégia é similar ao *Map-Reduce* (Dean et al., 2008).

Durante a distribuição dos *subframes*, é construída uma tabela com os *frames*, na qual cada um possui uma lista ordenada por capacidade dos servidores capazes de processá-lo. A escolha é feita por ordem de

custo do *frame* e pode haver servidores que ainda não sejam capazes de renderizar um *subframe*; nesse caso, esses servidores são descartados. Ao final desse processo, verifica-se se o tempo de renderização será atendido e, se o mesmo for atendido, o gerenciador realiza apenas o acompanhamento da renderização, reestimando o tempo a cada *frame* entregue (o cálculo só é refeito por *frame*, e não por *subframe*).

Caso o tempo não seja atendido em qualquer momento das estimativas do gerenciador, o mesmo entra no segundo contexto. Nesse caso, o número de *frames* é maior que o número de nós necessários para atender a demanda; assim, a forma de atender o tempo necessário é restringir a qualidade da imagem produzida. Isso é realizado de duas formas:

- **Redução de Amostras:** Nesse caso, nos processos de amostragem de dados, o sistema reduz pela metade a quantidade de amostras. Assim, o gerenciador espera reduzir pela metade o tempo de renderização. Essa redução pode ser maior ou menor, mas assume-se que o gerenciador terá tempo para reacomodar caso a redução não atenda.
- **Alteração de Técnicas:** Desse modo, o renderizador altera a forma de renderizar, trocando os modos de renderização. Assim, o sistema de *Shader* pode trocar os elementos de fundo que usam *Ray Tracing* ou *Path Tracing* para a rasterização; reduzindo, dessa maneira, o tempo de renderização. Nesse modo, o tempo de renderização só pode ser avaliado depois da renderização efetiva, cabendo ao gerenciador reestimar a condição de tempo remanescente.

Na primeira técnica, quando o gerenciador avalia a necessidade de reduzir a qualidade de um nível *i* para um nível *i – 1*, o mesmo avalia duas hipóteses. A primeira consiste em verificar o tempo que sobra com a alteração de qualidade e se é possível recomputar os *frames* já renderizados em qualidade *i* (somando os tempos gastos dos mesmos

divididos pela exponencial de base dois da diferença de nível). Caso seja viável, ou seja, se o tempo remanescente de toda a computação de *frames*, subtraído de 120% do tempo necessário para a renderização dos *frames* já recomputados com o novo nível, for positivo, o gerenciador reinsere os *frames* na fila e redistribui os mesmos. Uma vez que o gerenciador rebaixa um nível, o mesmo não pode subir novamente durante a renderização. Esse processo é recomputado a cada *frame* renderizado.

Quando se trata da segunda técnica mencionada acima, o processo realiza a troca das técnicas e reinicia a renderização trocando as técnicas. Quando é feita a recomputação do tempo remanescente, caso não seja atendido, o gerenciador sustenta as trocas de técnicas e realiza os rebaixamentos de qualidades de acordo com a primeira técnica (se for possível).



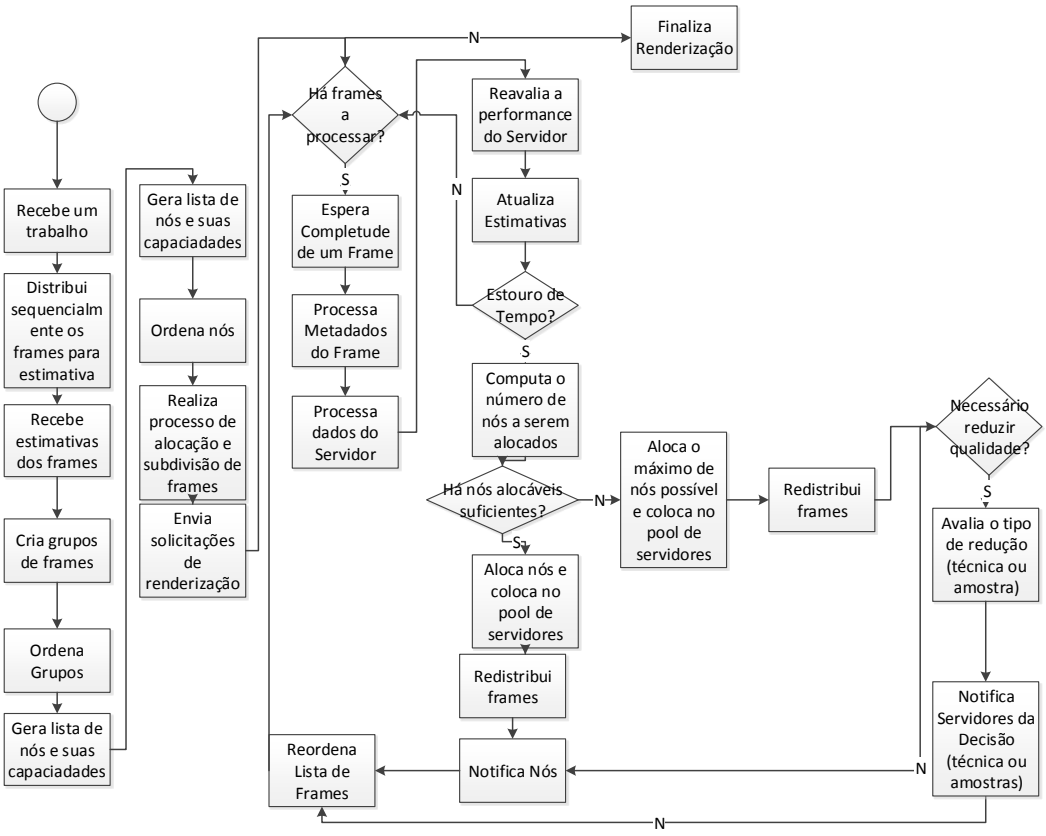


**Figura 43:** Algoritmo de gerenciamento com restrição temporal (imagem produzida)

Na figura 43 é mostrado esquematicamente o processo de gerenciamento.

### 4.1.8. Arquitetura de Renderização em Nuvem

Uma forma estudada de se atender a picos de demandas de processamento é o uso de computação em nuvens. No entanto, existem diversos aspectos que ainda dificultam o uso da nuvem na renderização. Os gerenciadores de renderização foram concebidos para operarem em ambientes de *clusters*. Portanto, a única forma de integrá-los com a nuvem é dar a aparência de extensão de rede para a nuvem. Assim, utiliza-se uma arquitetura de conexão como a da figura 44.

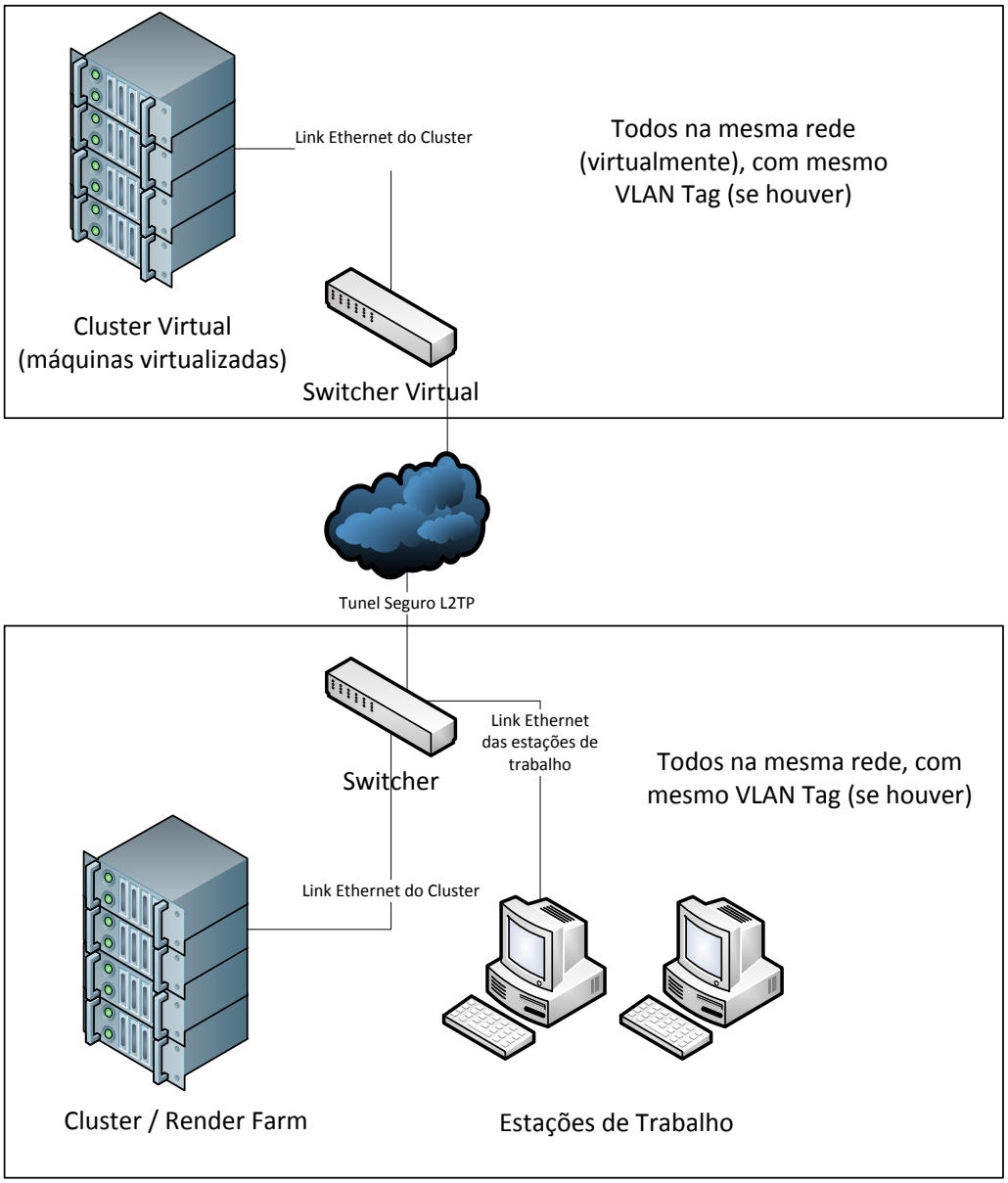


**Figura 44:** Arquitetura convencional de transbordo para a nuvem

No entanto, essa arquitetura possui alguns problemas, principalmente no Brasil, pois os maiores centros de computação em nuvem não possuem instalações no Brasil. Assim, para utilizar a nuvem, de forma a atender a forma de operação dos gerenciadores de renderização, perde-se desempenho com o uso da VPN para garantir a segurança da comunicação de dados, além de não ser possível realizar o alocamento dinâmico de nós. Outro ponto é que os dados necessitam

estar num *storage*, o que força a cópia de dados através da rede de comunicação, visto que os renderizadores tradicionais não compreendem o conceito de *blob*.

Nesse sentido, um dos maiores gargalos da computação em nuvem para a renderização é a rede, pois os dados necessários para a produção dos *frames* são muito grandes e poucas instituições/empresas podem dispor de links dedicados para a comunicação eficiente.

Assim, a utilização de *cloud computing* pela arquitetura da figura 44 é apenas uma extensão remota do *cluster*.

O gerenciador do presente trabalho considera o uso da *cloud* pública para o transbordo de carga de renderização. Assim, através dos módulos de computação nas nuvens, o gerenciador é capaz de utilizá-la de forma adequada.

Dentre as principais características da computação nas nuvens é a capacidade do serviço alterar dinamicamente a capacidade de processamento. Dessa forma, a alocação dinâmica de nós torna viável o transbordo de processamento para atender demandas temporais. Esse processo pode ser feito através das APIs da Amazon e da Microsoft[8].



### 4.1.9. Sincronização de Bases

Outra questão crucial sobre a computação na nuvem é o conceito de *Blob* e a forma com que os dados são armazenados nos servidores. Esse conceito difere da forma com que os *storages* trabalham. Assim, o processo de gravação e o acesso aos dados não podem ser vistos como um acesso direto a um arquivo em disco ou em um servidor de armazenamento.

[8] Deve-se notar que a conta associada deve possuir a capacidade de aumento de demanda de acordo com a *SLA* do contrato.

Dessa forma, o gerenciador e o *ClusterNode* são responsáveis por traduzir e processar as requisições de arquivo do renderizador, de forma que para o mesmo, o acesso e gravação seja transparente.

Assim, quando um *ClusterNode* é carregado como um nó da nuvem, o mesmo sabe que será necessário acessar os dados num *blob*, copiar para uma pasta local no nó e iniciar a renderização. Por isso é fundamental que um nó na nuvem utilize ao máximo a coerência temporal. Além disso, a gravação dos resultados e metadados deve ser copiada para o *blob* de saída.

Quando o gerenciador solicita a renderização de um *frame* em um nó da nuvem, o mesmo altera os caminhos no arquivo de cena. Ao receber o comando, o *ClusterNode* realiza o *download* (se necessário) dos arquivos, e altera os caminhos do arquivo de cena, e inicia a renderização da mesma maneira que um servidor local. Ao finalizar a renderização, o *ClusterNode* copia os dados e metadados para o *blob* de saída e notifica o gerenciador. Esse gerenciador, por sua vez, copia os resultados do *blob* de saída para o *Storage* local.

Para que esse procedimento ocorra, é necessário que as bases de dados estejam sincronizadas, ou seja, os dados locais devem ser iguais aos dados remotos. A solução encontrada para atender a essa demanda foi a utilização do sistema da Cloudberry Lab, chamado Cloud Backup for Windows Server. Esse produto é capaz de manter uma pasta local sincronizada com um *blob* em uma *cloud* pública. Assim, uma vez que seja necessário o acesso por um nó remoto, o mesmo poderá utilizar os dados presentes no *blob* (figura 45).

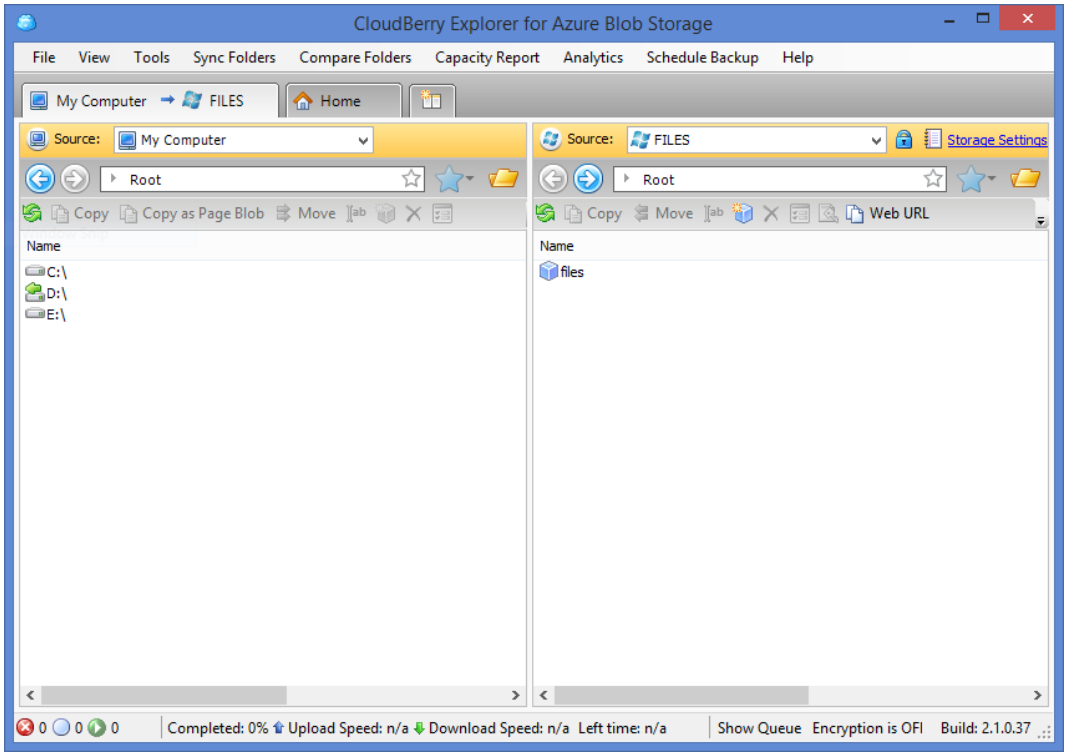


**Figura 45:** Cloudberry, sistema de sincronização de bases.

Outro fato importante é a forma com que os dados são sincronizados, pois existem arquivos que são muito grandes e haveria uma grande tempo desperdiçado com o envio do mesmo de forma completa. Assim, uma solução é o envio por blocos, ou seja, só são enviados os blocos dos arquivos que efetivamente foram alterados, reduzindo o tempo de envio, sendo esse tamanha definido entre 256 KBytes a 2 MBytes (optou-se por 512 KBytes).

### 4.1.10. Distribuição com restrição temporal

A grande diferença da utilização de *cloud computing* para atender a restrição temporal é que pode-se ter a condição de número de nós maior que o número de *frames* e, nesse caso, a situação é similar ao primeiro contexto da seção 4.2.4.

É importante notar que mesmo que se tenha o número de servidores igual ao número de *frames*, pode-se ter uma extrapolação do tempo de renderização. Além disso, as máquinas adicionadas sempre atendem às condições mínimas de operação, pois são alocadas de forma

a cumprirem esses requisitos. Na prática, o gerenciador de renderização sempre aloca a melhor configuração de máquina virtual disponível no sistema gestor da *cloud* pública.

Dessa forma, o processo de distribuição difere um pouco do apresentado em 4.2.4. No contexto de *cloud*, uma vez computado o custo de cada *frame* e constatado que o tempo total será maior do que o tempo total disponível, o gerenciador computa o número de nós a serem alocados considerando a configuração disponível (normalmente 8 cores com 32 Gb de memória RAM). Para dimensionar esse valor, o processo de distribuição segue a forma já discutida, ou seja, utilizando a associação das tabelas de custos. Após a computação do tempo remanescente é verificado se existe algum *frame* que o tempo de renderização estimado fique 25% maior que o tempo total; nesse caso, o gerenciador realiza a subdivisão de todos os *frames* nessa condição. Após isso, o cálculo de tempo de renderização é refeito e, caso o problema persista, os *frames* vão sendo subdivididos até que a condição seja resolvida.

Assim, não existe *frame* que tomará mais do que 25% do tempo total para ser produzido. Agora, o número de nós necessários para finalizar a renderização a tempo pode ser obtido pela fórmula da equação 13:

$$n_{nodes} = 1{,}2 \cdot \frac{\left(\sum t_{frames} - \sum_{i,j} t_{render}\left(node_i, frame_j\right)\right)}{\sum_j t_{render}\left(node_{cloud}, frame_j\right) + t_{allocation}}$$

**Equação 14:** Estimativa do número de servidores a serem alocados na nuvem

A equação 14 computa, no numerador, o tempo remanescente do atendimento dos nós locais, ou seja, o tempo total subtraído do tempo de processamento dos nós locais. Esse tempo é dividido pela soma do tempo gasto pelos nós da nuvem para renderizar os *frames*, ou

*subframes*, remanescentes na fila, acrescido do tempo de alocação na nuvem, que é aproximadamente cinco minutos. Observa-se que o valor é aumentado de 20%; isso para que, caso seja necessário mais subdivisões, já se tenham máquinas disponíveis para o atendimento, visto que o tempo de alocação dessas máquinas é alto.

Uma observação importante é que tanto em 4.2.4, quanto em 4.3.2, o cálculo do tempo remanescente não leva em conta que quando, um *frame* é finalizado, a máquina executora ficará disponível. O motivo dessa não inclusão é para permitir que o gerenciador tenha uma margem para realizar mais subdivisões e possa ter mais nós disponíveis para atender a demanda temporal. Apesar de ser um desperdício de processamento, essa área de manobra é necessária para que haja a possibilidade de atendimento.

Dessa maneira, quando um nó fica disponível por ter finalizado o *frame*, o gerenciador realiza um processo de reacomodação, que consiste em subdividir um frame em processamento. Isso é feito de maneira simplificada, escolhendo-se os *frames* de maior custo e solicitando a renderização de metade dos blocos ainda não iniciados pelo nó em execução. Nesse caso, o nó recebe uma máscara das áreas que não deve iniciar a renderização; e observa-se que a PI do *frame* ainda não pode ser usada, pois o *frame* não está completo. Essa restrição é removida com o uso da aceleração apresentada no capítulo 5.

A reacomodação só é realizada se a estimativa de finalização estiver ultrapassando o limite estabelecido. A figura 46 mostra o fluxo de execução da distribuição.

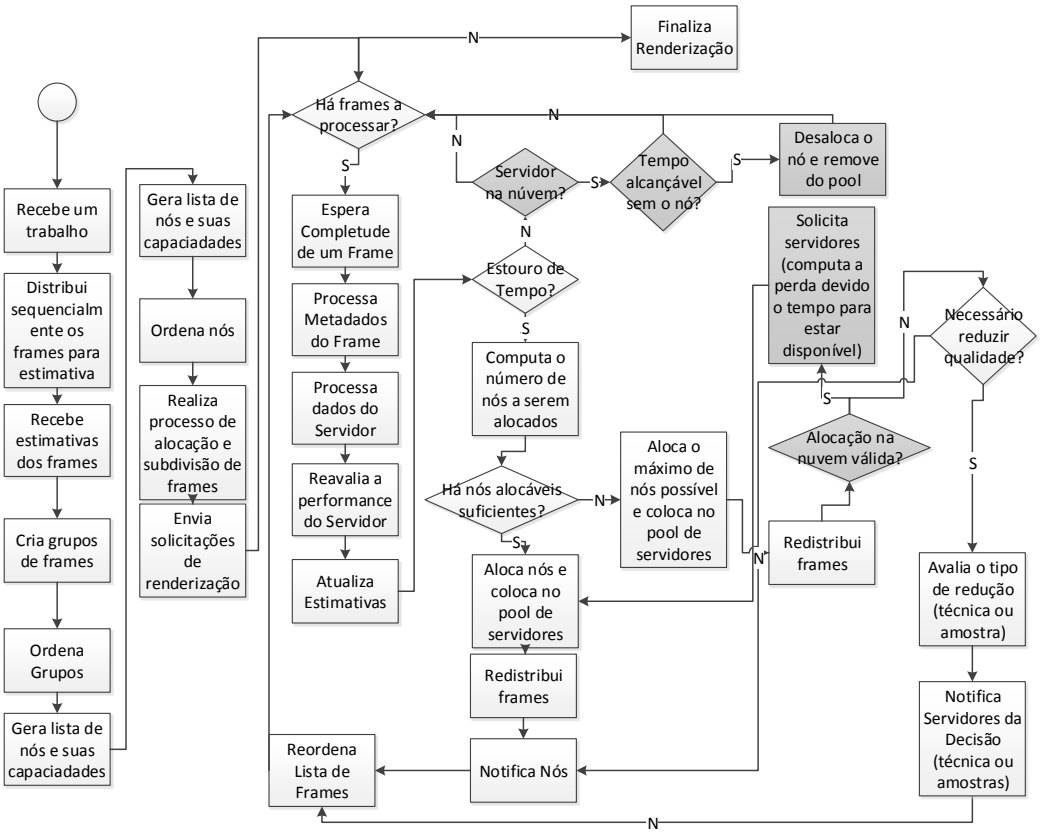


**Figura 46:** Processo de distribuição na nuvem com restrição temporal

### 4.1.11. Distribuição com restrição de custos

Outro ponto associado a *cloud computing* é o custo de execução, normalmente calculado em horas por *core*. Esse custo é associado à alocação da máquina virtual, ao consumo elétrico e de ar condicionado, bem como os custos de *software*.

Num processo real de uso de computação em nuvem, existe sempre uma restrição de custo, ou seja, um valor máximo para os gastos com a alocação dinâmica de nós.

Dessa forma, o gerenciador deve ser capaz de trabalhar com uma restrição de custos, o que representa uma restrição de recursos, recaindo na situação de 4.2.4, pois dado um valor máximo de gastos $max_{\$}$, o número máximo de nós alocáveis é limitado.

O processo de distribuição segue muito similar ao apresentado em 4.3.2, no entanto, a cada alocação de nó na nuvem é feito o desconto em $max_{\$}$ até que o número de *frames*, ou *subframes*, seja atendido ou o orçamento disponível acabe. Observa-se que o custo de cada alocação é calculado como $custo_{\text{nó cloud}} \cdot \left(t\left(node_{cloud}, frame_j\right) + 5min\right)$.

Quando a reavaliação revela a insuficiência para o atendimento da restrição temporal, inicia-se o processo de redução de qualidade apresentado em 4.2.4. Um ponto importante é que a cada alocação, automaticamente o nó adicionado tem que ser desalocado após a completude do frame escalado; isso é uma simplificação para atender a demanda de custo. A figura 47 ilustra o processo.

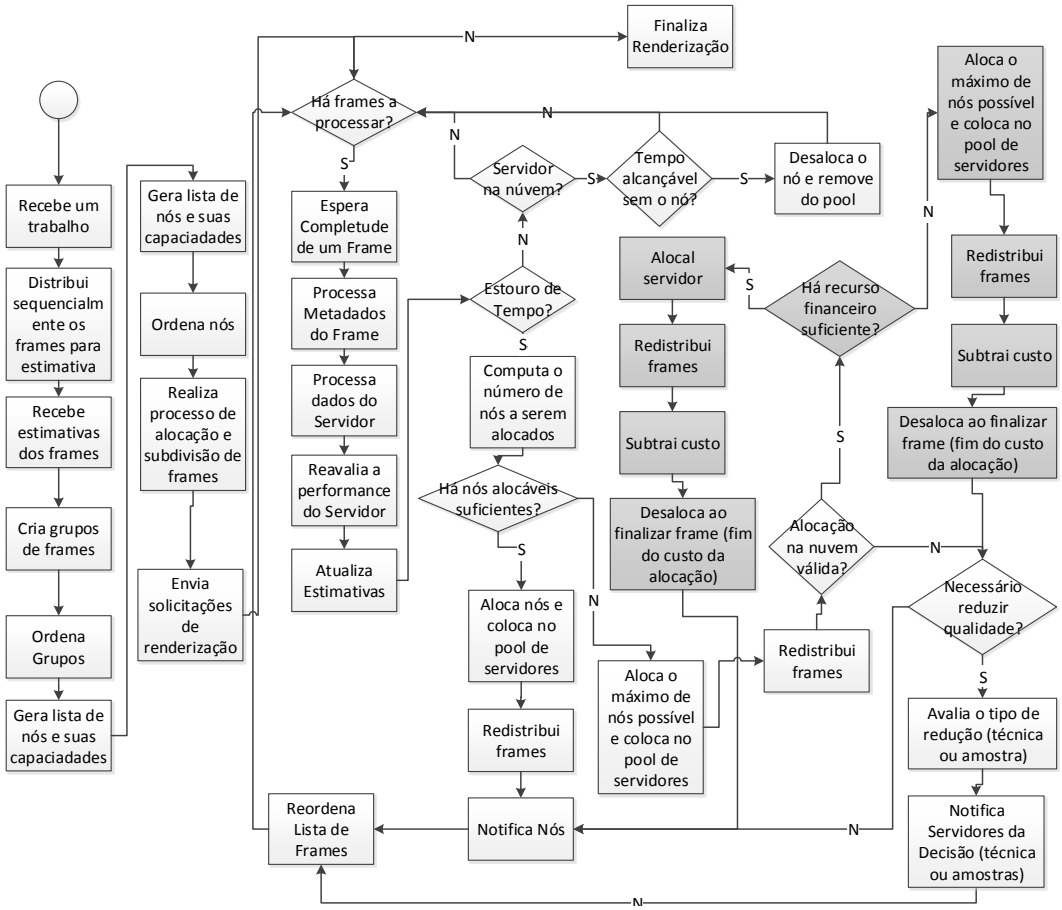


**Figura 47:** Alocação de nós com restrição de custo (imagem produzida)

### 4.1.12. Múltiplos Trabalhos

Por fim, todo o processo apresentado pelo sistema de gerenciamento é capaz de coordenar e gerenciar apenas um trabalho, porém, é comum o envio de múltiplas cenas e *takes*. Assim, além de coordenar o processo de renderização individual de cada cena, torna-se necessário atender as demandas de várias cenas.

O gerenciador trata esse problema de duas maneiras:

- **Prioridade:** Nesse modo, tarefas de maior prioridade são executas primeiro. Ao fim da execução de uma tarefa, a próxima tarefa em nível de prioridade é executada e é computado o tempo remanescente da mesma, descontado o gasto pela tarefa anterior. O usuário é quem especifica a prioridade.
- **Mais Curto Primeiro (EDF – Earliest Deadline First):** Nesse modo, as tarefas são ordenadas por *deadline*, assim, as tarefas que possuem menor tempo remanescente são executadas primeiro. O processo é iterativamente recomputado e atualizado, uma vez que, para gerar o *deadline*, o gerenciador deve fazer a estimativa de tempo total de computação.

A maneira baseada em prioridade é a encontrada em todos os sistemas de gerenciamento de renderização, no entanto, a mesma não se compromete com prazos, necessitando de um árbitro para avaliar o correto uso das prioridades.

Por outro lado, a metodologia de *deadline* é a utilizada por sistemas operacionais de tempo real (Tanenbaum et al, 2004), os quais garantem o atendimento de uma demanda, mesmo tendo que chavear entre tarefas. Assim, a estratégia de EDF é o modelo utilizado por padrão pelo gerenciador (figura 48).

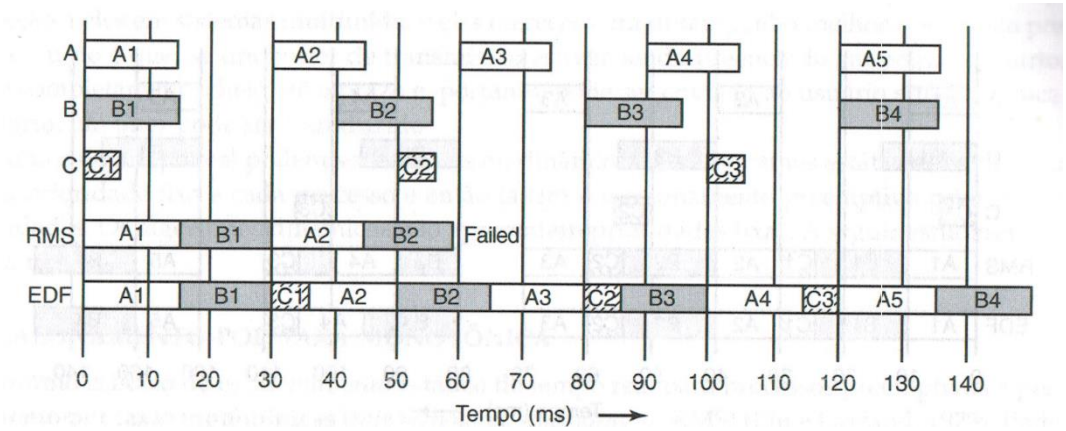


**Figura 48:** Exemplo de escalonamento usando o EDF – Earliest Deadline First, extraído de (Tanenbaum et al, 2004)

Observa-se que, em qualquer metodologia, uma vez que nós do cluster fiquem ociosos, o escalonador inicia o processamento de outra tarefa. Ainda, o escalonamento pode comprometer ainda mais a complexidade para atender uma demanda, pois o tempo remanescente de uma tarefa é afetado pela outra tarefa. Existem possibilidades de estudo de modelos de escalonamento das tarefas de forma a minimizar o estouro de tempo ou atender de uma forma mais otimizada. Porém, esse tratamento não é abordado por este trabalho.

Portanto, o gerenciamento de renderização é uma tarefa muito complexa e que tem papel fundamental no tempo final de renderização. Ainda, o processo de escalonamento tanto de tarefas, quanto de *frames* pode ser melhorado. Todavia, o capítulo 5 mostra algumas melhorias que podem reduzir ainda mais o tempo de renderização.