Integrando Novos Modelos de Comunicação em Rede ao Módulo *CybCollaboration*

Paulo Vinícius de Farias Paiva

Orientadora:

Liliane dos Santos Machado

João Pessoa – PB

Março/2011

RESUMO

A Realidade Virtual (RV) tem como idéia principal o uso de ambientes tridimensionais aonde os usuários podem explorar e interagir em mundos virtuais levando-os a sentirem-se imersos nesta experiência. Sistemas de RV são caracterizados principalmente pelo elevado grau de realismo que oferecem ao simular ambientes e situações reais. Dentre os vários tipos de sistemas de RV existentes, os Ambientes Virtuais Colaborativos (AVCs) são ambientes virtuais que utilizam a capacidade da Internet em aproximar as pessoas e combinar as suas ações, contribuindo assim, na construção do conceito de trabalho colaborativo à distância e em tempo real. Utilizando técnicas avançadas de interação, os AVCs auxiliam os seus usuários, seja no treinamento e educação de novos profissionais, na avaliação destas interações como também no estabelecimento de novas parcerias para a execução de trabalhos, projetos e pesquisas em conjunto. Deste modo, a colaboração é utilizada para coordenar as interações entre os vários participantes no ambiente virtual. Uma vez que os AVCs são ambientes em rede, se torna imprescindível o uso de técnicas que melhorem a transmissão de dados diminuindo ao máximo as atenuações imprimidas pelos meios de transmissão. Para ambientes que requerem altas taxas de transmissão de dados como os AVCs com retorno de força, os protocolos de transmissão devem ser implementados de forma que os usuários tenham retornos táteis de boa qualidade. Este trabalho visou o desenvolvimento de melhoress práticas de programação em rede e integração de tais modelos ao framewrok CyberMed.

SUMÁRIO

Capítulo 1. Introdução

1.1. Motivação

Este trabalho visa relatar os estudos realizados acerca dos Ambientes Virtuais Colaborativos, baseados em Realidade Virtual (RV), bem como acerca de um *framework* com funções colaborativas e voltado para o desenvolvimento de aplicações médicas chamado CyberMed. Também são apresentados algumas arquiteturas e protocolos de comunicação em rede utilizados nestes ambientes. O módulo de colaboração deste *framework, c*hamado *CybCollaboration*, descreve um protocolo para a comunicação remota entre vários usuários, em tempo real, permitindo a realização de determinadas tarefas em conjunto. O módulo *CybCollaboration* suporta atualmente o desenvolvimento de aplicações colaborativas que usam dispositivos de interação homem-máquina, como os dispositivos hápticos, que exploram o sentido do tato dos usuários, os *trackers* que atuam como rastreadores de movimento corporal e mais recentemente as luvas de dados. A sub-categoria dos AVCs que inclue o uso de dispositivos hápticos são conhecidos como Ambientes Virtuais Hápticos Colaborativas (AVHCs). Entretanto, é conhecido que estas aplicações são muito sensíveis às transmissões via rede, devido às altas taxas de transmissão de dados pela rede, gerados pelo dispositivo, acarretando danos no nível de realismo da interação háptica, isto é, no sentido do toque dos usuários. Com o intuito de prover melhor performance de rede, bem como suportar maior número de usuários nos AVCs desenvolvidos com o CyberMed, sobretudo para aqueles que utilizam os dispositivos hápticos, foi observado a necessidade de realizar alterações no protocolo do módulo *CybCollaboration*. Deste modo, este trabalho verificou a necessidade de implementação de novos modos de transmissão para o módulo de colaboração do *framework* CyberMed, bem como o estudo do comportamento de rede das aplicações geradas. Deste modo, se tornaria possível a inclusão de múltiplos usuários colaborando em tempo real sem maiores impactos na aplicação.

Este relatório tem como objetivo apresentar os trabalhos inseridos no projeto INCT-MACC que foram realizados no período de Julho a Dezembro de 2011, junto ao grupo de pesquisa do laboratório labTEVE (UFPB) aonde as principais atividades foram o estudo de comunicações em redes, programação paralela, o estudo de arquiteturas de redes em AVCs, o desenvolvimento de modelos de simulação para análise de performance.

1.2. Ambientes Virtuais Colaborativos (AVCs)

Graças ao avanço das redes de computadores, e da Internet, o uso dos ambientes baseados em RV foi expandido com a possibilidade de interações com outros usuários à distância. Surgiram assim, os Ambientes Virtuais Colaborativos (AVCs) que são ambientes virtuais em rede que permitem aos usuários interagir uns com os outros em tempo real, visando à realização de uma tarefa em conjunto. A colaboração nestes ambientes permite que profissionais de diversas áreas interajam entre si, mesmo que localizados à distâncias

consideráveis. Alguns exemplos práticos como resultantes de tais interações são: projetos, pesquisas científicas, trabalhos de negócios, entretenimento, educação à distância, *design* colaborativo de produtos dentre diversas outras aplicações.

Dentro do contexto da medicina, os AVCs trazem grande importância na formação de novos profissionais da medicina, possibilitando colaboração entre estudantes e profissionais, melhorando a qualidade da aprendizagem de procedimentos médicos. Diferentes abordagens para o uso da colaboração estão sendo estudadas e Margery et al. [MAR99] divide a colaboração em três diferentes categorias:

• Nível 1 - Os usuários podem percebem-se multualmente através de seus *avatares*, mas não podem realizar modificações no ambiente virtual;

• Nível 2 – As modificações são realizadas individualmente e sequencialmente por cada usuário;

• Nível 3 - Os usuários modificam, simultaneamente, as mesmas ou diferentes propriedades de um objeto virtual presente no AV.

A colaboração de Nível 3, também conhecida como manipulação cooperativa de objetos [PAI10], pode ocorrer de diversas formas visto que um objeto virtual permite diferentes transformações sobre si (rotação, translação, cor, etc). Deste modo, múltiplas ações poderão ser combinadas desde que, antes de iniciada a colaboração, sejam pré-estabelecidas regras a fim de definir quais propriedades do objeto cada participante deverá alterar [PIN2], senão todas. É importante ressaltar que a comunicação em rede destes sistemas deve ser bem administrada e têm influência direta na qualidade da interação colaborativa. Deste modo, os AVCs também são considerados como Ambientes Virtuais em Rede, que são sistemas executados paralelamente por vários hosts através Internet, ou de uma rede de computadores qualquer, compartilhando informações do ambiente virtual tais como: propriedades dos objetos virtuais envolvidos, posições dos interadores, *avatares,* informações de rede como os endereços IP de cada usuário, dados que descrevem o estado corrente da aplicação [ZYD99].

1.2.1. Armazenamento e Distribuição do AV

Uma problemática nesses ambientes é a garantia de consistência das informações, ou seja, garantir que todos os usuários participantes da colaboração, estão presentes em um ambiente com as mesmas características independentemente do momento em que a conexão é estabelecida [ZYD99]. Sistemas de gerenciamento de banco de dados (SGBDs) podem ser utilizados em paralelo ao AVC de modo que o status do ambiente seja devidamente armazenado possibilitando aos usuários interagirem de forma contínua com o ambiente, que existirá independentemente de possuir usuários conectados. Existem algumas possibilidades quanto ao armazenamento e distribuição das cópias de um AV sendo as mais conhecidas a arquitetura distribuída, aonde cópias do AV são distribuídas entre todos os nós da rede e a centralizada, em que o mundo virtual encontra-se armazenado em uma base de dados localizada em um servidor. No primeiro caso, o AV é executado em um conjunto de máquinas clientes, sem memória compartilhada e independentemente das diferentes características de software e hardware.

Uma das maneiras de manter a consistência em um ambiente virtual colaborativo é conhecida como *replicação ativa* [ZYD99]. Neste modelo, cada usuário fica responsável por manter uma cópia local do AV e replicar as alterações realizadas localmente para os outros participantes remotos. Cada participante, ao receber novas notificações informando que o ambiente foi alterado, deve atualizar o seu ambiente imediatamente garantindo, portanto, o sincronismo e a consistência do ambiente. Um alto nível de sincronização entre os participantes se faz necessário, a fim de que o aspecto colaborativo não seja comprometido possibilitando assim, que cada um perceba as interações realizadas por todos os outros e no momento em que ocorram. Estas mensagens de atualização devem ser trocadas de acordo com um protocolo de rede específico já existente ou que seja implementado pelo desenvolvedor da aplicação. Uma desvantagem na utilização do mecanismo de replicação e de suas mensagens de estado, principalmente para os AVCs de grande escala, é o aumento na propagação de mensagens pela rede, visto a necessidade de replicação da base de dados principal para todos os clientes, sendo que estas mensagens muitas vezes são consideradas irrelevantes para certos clientes.

1.2.2. Comunicação em Rede

A comunicação entre os participantes de um AVC é realizada sobre diferentes tipos de arquiteturas em rede. Arquiteturas distribuídas são utilizadas em muitos dos AVCs existentes na atualidade. As mais conhecidas são:

a) Cliente/Servidor – O estado do AV é mantido em uma base de dados centralizada, isto é, em um ou vários servidores que mediam as interações dos clientes conectados com o AV. Tal arquitetura de rede é mais aconselhável para os ambientes com número determinado de usuários. Pelo fato do tráfego de rede ser imprevisível em ambientes virtuais dinâmicos de larga escala, os servidores tendenciosamente se tornam os "gargalos" do sistema, aumentando a latência do ambiente [SAL10].

b) Peer-to-peer Unicast – Modo não-hierárquico de comunicação. Cada usuário mantém e atualiza a sua própria cópia local sendo responsável por divulgar aos outros clientes as suas alterações locais. A comunicação em rede é feita mediante mensagens de *unicasting*, ou seja, o emissor envia diretamente ao receptor sem intermediários. Este modo de comunicação implica em ocupação da largura de banda à medida que o número de usuários venha a aumentar, sendo portanto, mais apropriado para os AVs com menor número de participantes [SAL10][ZYD99].

c) Peer-to-peer Multicast – O uso do protocolo de comunicação Multicast neste modelo possibilita maior seletividade dos receptores, ao contrário do Broadcast. O Multicast permite a divulgação de mensagens para múltiplos usuários divididos por grupos de receptores de determinados tipos de tráfego de dados. Deste modo, esta arquitetura de rede é eficiente para uso em ambientes virtuais colaborativos de larga escala uma vez que os diferentes tipos de tráfego de rede podem ser administrados com maior eficiência e otimizando o impacto causado com o aumento dos usuários [KEV03].

1.2.3. Comunicação em Rede por Multicast

Multicast é a entrega de informação para múltiplos destinatários simultaneamente usando a estratégia mais eficiente onde as mensagens só passam por um link uma única vez e somente são duplicadas quando o link para os destinatários se divide em duas direções [KEV03][ZYD99][HAL01]. Comunicações em Multicast permitem que o programador definam um socket UDP e envie a um grupo de pacotes de dados de uma única vez, isto é, em uma única chamada à função sendto(). E ainda o pacote é entregue para os vários receptores em potencial. Em vez de enviar N pacotes iguais separadamente a cada um dos N receptores, um pacote multicast pode ser enviado e irá atingir todos os N receptores, mesmo sendo N muito grande. A faixa especial de endereços IP é usado para criar um grupo lógico de receptores. Usando este endereço, o programador da aplicação tem a capacidade de enviar um ou um fluxo de pacotes para este endereço de destino e esperar que a rede tente entregar uma cópia do pacote para cada receptor no grupo multicast especificado. A comunicação multicast depende de funcionalidade adicional na rede para construir um "árvore de encaminhamento multicast" entre o aplicativo de envio e do grupo de receptores. O emissor da mensagem está localizado na "raiz" da árvore. Da raiz, um fluxo de pacotes flui para cima do tronco até alcançar os "ramos". Em cada ramo da árvore, a rede recebe um pacote de entrada e copia para cada um dos ramos de saída (Figura 1) [KEV03].

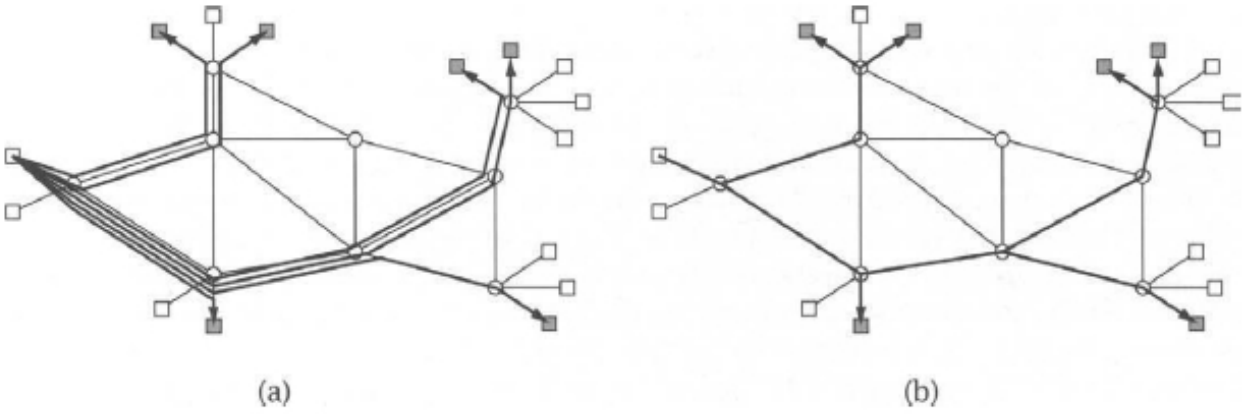

Figura 1. Diferentes graus de escalabilidade: (a) unicast e (b) multicast [KEV03]

Capítulo 2. Metodologia

Este Capítulo visa detalhar o estudo e aprendizado realizado sobre as principais ferramentas utilizadas ao longo do projeto que foram: a linguagem de programação C/C++ usada na implementação das alterações realizadas no *CybCollaboration*; o *framework* CyberMed e o simulador de redes OPNET Modeler (OM) usado nas simulações de rede.

2.1 Linguagens de Programação C e C++

Como requisito básico para implementação das alterações no módulo de colaboração do CyberMed, foir necessário o estudo e aprimoramento em técnicas de programação em C/C++, tais como o uso da alocação dinâmica, ponteiros, listas encadeadas, herança, polimorfismo, funções virtuais, *threads* e *sockets,* dentre outros conhecimentos. Neste tópico serão relatados os aprendizados de tal estudo.

2.1.1. Utilizando Ponteiros

Os ponteiros guardam endereços de memória. Deste modo, ao declararmos ponteiros, é informado ao compilador o tipo de variável que será apontada. Ponteiros do tipo *char* apontam para variáveis do tipo *char* e assim por diante.

a) Declarando um ponteiro e alterando o conteúdo apontado

É importante lembrar que o ponteiro obrigatoriamente deve ser inicializado (ou seja, apontado para alguma posição de memória) antes de ser usado! Para atribuirmos um valor a um ponteiro recém-criado basta que o igualemos a um valor de memória utilizando o operador & (que indica uma posição de memória) como a seguir:

```
int count=10;
int *pt;
pt=&count; //pt aponta para a posição de memória de count
```

Assim, através do ponteiro criado pt, podemos alterar o valor da variável count. O operador * neste caso indica que o conteúdo apontado por pt agora será 12, alterando assim o valor de count:

```
*pt=12;
```

b) **Incremento de posição de memória**

Ao incrementarmos um ponteiros ele passa a apontar para o próximo valor do mesmo tipo na memória. Por exemplo, se o ponteiro aponta para um inteiro e o incrementamos, ele passa a apontar para o próximo registro de um inteiro na memória. Deste modo, se a próxima posição de memória contém um float o ponteiro não apontará para este, esperando alcançar o endereço do próximo inteiro.

```
int *p;
p++; // aponta para a próxima posição de memória com um conteúdo inteiro
```

c) Incremento do conteúdo apontado

Para incrementar o conteúdo apontado deve-se utilizar o operador * como exibido a seguir:

```
int *p, n=10;
p=&n; // p aponta para 10
(*p)++; // Incrementa o conteúdo apontado por p, ou seja, n=11 agora
```

d) Soma/subtração de inteiros com ponteiros

```
p=p+15;

*(p+15); //usando o conteúdo do ponteiro 15 posições de memória adiante
```

e) Vetores representados como ponteiros

Ao declarar uma matriz *int m[5][5]*, o compilador calculará quantos bytes são necessários para armazená-lo realizando um cálculo semelhante a *5 * 5 * sizeof(int).* Após armazenado, o nome da matriz m torna-se um ponteiro para o primeiro elemento. Assim, pode-se entender que a notação m[índice] para acessar um elemento da matriz é idêntica a seguinte notação: *(m+índice). Como exemplo, temos a seguinte instrução:

```
nome_matriz[índice] == *(nome_matriz+índice) // em ambos nome_matriz é um ponteiro
```

Assim, fica claro o motivo de em linguagem C, a indexação de vetores começar com zero. Isto ocorre pelo fato de ao pegarmos o valor do 1° elemento de um vetor, desejarmos na verdade, ***nome_da_variável** e então devemos ter um índice igual à 0.

```
int m[5][5];

int *iterador, i, numElementos;

iterador = m[0]; // o ponteiro iterador aponta para o primeiro elemento da matriz m[0] ou *(m+0);
```

f) Ponteiros com função de vetores

Pode-se indexar a posição de ponteiros uma vez que podemos fazê-los apontar para a próxima posição de memória. Deste modo podemos fazer uma iteração em várias posições de memória, tal qual um array. Exemplo:

```
int vet[5] = {1, 2, 3, 4, 5};
int *p, i;
p = &vet;
for(i=0; i<5; i++)  printf("%d", p[i]);
```

No exemplo anterior, dentro do laço, poderíamos usar a notação seguinte para acessar os elementos do array vet: printf("%d", *(p+i));

g) Ponteiros representando Strings

Uma vez que o nome de vetores são ponteiros, o nome de uma string é um ponteiro para o tipo char (* char). No próximo exemplo são recebidos duas strings. A origem é copiada em cima da destino, caractere por caractere de forma a simular o comportamento da função strcpy():

```
void StrCpy(char *destino, char *origem) {
        while (*origem != '\0') { // enquanto não alcançar o final da string
                // pegue o caractere atual da string origem e armazene no caractere atual da string destino
                *destino = *origem;
                // Faça com que os ponteiros apontem para os próximos caracteres das strings
                origem++;
                destino++;
        }
}
```

Assim, esta função permite a cópia de uma string para outra.

h) Strings constantes

Toda string inserida no programa é colocada em um banco de strings criado pelo compilador. Neste banco, para cada string inserida no código é armazenado o endereço do início desta string. Com fim de evitar redundância de informação, ou seja, inserir várias strings iguais ao longo do código e para evitar a armazenagem na mesma posição de memória no banco de strings, podemos usar um ponteiro que aponte para esta string que será usada várias vezes:

```
char *strConstante = "String Constante";
```

2.1.2. Listas Encadeadas

Os vetores e matrizes ocupam um espaço contíguo de memória permitindo assim o acesso randômico, ou seja, acesso direto a qualquer um dos elementos a partir do ponteiro para o primeiro elemento (com índice 0) e que é caracterizado pelo nome do vetor. Nas listas encadeadas, para cada novo elemento inserido na estrutura, devemos alocar um espaço de memória para armazená-lo. Porém, os elementos não são armazenados de forma seqüencial uma vez que a alocação é dinâmica. Assim, faz-se necessário armazenar junto às informações da própria estrutura o ponteiro para o próximo elemento da lista, deixando-a encadeada e permitindo o acesso seqüencial [CEL04]. Comumente, a lista encadeada é representada por um ponteiro para o primeiro elemento (nó), como no exemplo a seguir:

```
// estrutura que representa um nó da lista
struct lista {
        int campo;
        struct lista* prox;
};
typedef struct lista Lista;

/* função de criação */
Lista* create(void) {
        return NULL;
}
```

a) Função de Inserção

Para cada elemento, devemos alocar dinamicamente a memória necessária para armazenar o novo elemento, guardar a sua informação e fazer com que ele aponte para o próximo elemento que é aquele que era o primeiro elemento (que apontava para NULL). Um exemplo seria a função insere() exibida a seguir:

```
/* Inserção no início da lista, e retorna o valor da lista */
Lista* insere (Lista* l, int c) {
        Lista* novo = (Lista *) malloc(sizeof(Lista));
        novo->campo = c;
        novo->prox = l;
return novo;
}

int main(void) {
```

```
        Lista* l;
        l = create();
        l = insere(l, 24);
    return 0;
    }
```

Uma forma de não dispensar a necessidade de atualização constante do valor da variável da lista (l), podemos passar o parâmetro da lista por referência para a função de inserção. Neste caso, os parâmetros das funções seriam do tipo ponteiro de ponteiro para lista (Lista** l) e o seu conteúdo poderia ser acessado/atualizado dentro da própria função [CEL04].

```
/* Inserção no início, e atualiza o valor da lista */
void insere (Lista** l, int c) {
    Lista* novo = (Lista*) malloc(sizeof(Lista));
    novo->campo = c;
    novo->prox = *l; // conteúdo de l, ou seja, estrutura apontada pelo ponteiro passado
    *l -> novo;
}
```

A chamada desta função no cliente seria:

```
    Lista *l = create();
    Insere(&l, 23);
```

b) Percorrendo a lista

Para percorrer uma lista, basta que seja realizado um *loop* aonde em cada interação o campo que aponta para o próximo é incrementado para a próxima posição de memória:

```
    /* imprimindo valores dos elementos */
    void imprime(Lista* l) {
        Lista* p; // iterador
        for(p=l; p != NULL; p = p->prox)
            printf("campo = %d ", p->campo);
    }
```

e) Função para busca de elemento

```
    função para busca de um elemento, e retorna NULL caso não seja encontrado */
    Lista* busca(Lista* l, int v) {
    Lista* p;
        for(p=l; p->NULL; p=p->prox) {
            if(p->campo == v)
              return p;
        }
```

```
        return NULL;
}
```

e) **Função para remoção de elementos**

```
        ista* remove (Lista* l, int ) {

        Lista* ant = NULL;
        Lista* p = l;

        while (p != NULL && p->campo != v) {
                ant = p;
                p = p->prox;
        }

        if (p == NULL) // verifica se achou o elemento
        return l;  // caso não, retorna a antiga lista

        if (ant == NULL) { // retira elemento do inicio
                l = p->prox;
        } else {
                ant->prox = p->prox; // retira elemento do meio da lista
        }
        free(p);
        return l;
```

2.1.3. Erros de Compilação

Nestem item, serão listados alguns erros de compilação exibidos pelos compiladores g++ (Unix) e do Dev-C++ (Windows) utilizados durante os estudos. Também serão listadas algumas descrições ou possíveis soluções:

Tabela 1. Alguns erros de compilação gerados pelo g++ e Dev-C++

Erros de compilação	Descrições e/ou soluções
switch quantity not an integer	Em C/C++ o switch só aceita caracteres ou inteiros
ISO C++ forbids declaration of ‘vector’ with no type	Usar o namespace std, e usar o operador ::
cannot allocate an object of type ‘Objeto’ because the following virtual functions are abstract.	Classes abstratas, isto é, que possuem ao menos um método virtual puro (sem implementação) não podem ser instanciadas.
'Class XX' has virtual functions but non-virtual destructor:	Para usar métodos virtuais puros, o destrutor de uma super classe deve ser público e virtual ou protected e não-virtual
ISO C++ Forbids Declaration Of 'List' With No Type	1- Possivelmente o header do tipo usado não foi incluído, ou o tipo não foi definido. 2- Ao usar objetos do tipo list, deve-se usar o namespace std, e usar o operador :: 3- Pode ser o retorno da função chamada que não foi definido
Multiple types in one declaration (erro na última linha do código)	Faltou ; em uma das classes
variable or field 'dtra' declared void	Erro quando o tipo do parâmetro de uma função não é especificado

OBS: Criar um array dentro de uma função e retornar um ponteiro para o 1° elemento e tentar acessá-lo ao terminar a execução da função dará erro, por que o espaço de memória já foi liberado. Deste modo, dentro da função deve-se criar um vetor e alocar dinamicamente a memória que ficará no *heap*.

2.2 CyberMed

O CyberMed é um *framework* de licença aberta, desenvolvido em C++, voltado para a criação de aplicações médicas baseadas em RV. Como todo *framework*, o CyberMed viabiliza ao programador, um desenvolvimento mais ágil de sistemas, além de oferecer a possibilidade de extensão do código fonte e inclusão de novos recursos. O CyberMed destaca-se, principalmente, pela ampla gama de recursos oferecidos pelos seus módulos. Dentre suas principais funcionalidades destacam-se:

- Visualização mono e estereoscópica;
- Interação através de dispositivos convencionais (mouse e teclado) e não-convencionais (háptico e de rastreamento);
- Uso de modelos tridimensionais deformáveis;
- Suporte à detecção de colisão;
- Suporte a aplicações de colaboração e integração de métodos de avaliação *online.*

Como pode ser visto na Figura 2, o CyberMed é constituído por dez módulos, dispostos em três camadas principais: *Application Engine, Core* e a *Utils.* A camada *Core* é responsável pelo controle dos estados internos, como aquisição, cálculo, armazenamento e acesso aos dados do sistema [MAC08]. A camada *Application Engine* provê um conjunto de métodos que auxiliam o usuário na construção de suas aplicações na inserção de recursos como a visualização, a colisão, a deformação, a avaliação e a interação háptica. Por fim, a camada *Utils* oferece uma série de facilidades na construção e edição de menus e na realização de operações matemáticas como cálculo de matrizes e operações lineares.

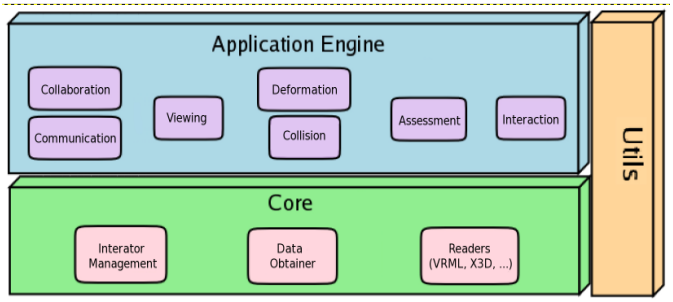

Figura 2. A arquitetura de camadas e módulos do CyberMed [PAI10]

Vale ressaltar que arquitetura do CyberMed possui suas camadas e módulos nomeados na língua inglesa visando uma maior padronização com as implementações internacionais. Além disso, o módulo mais

recente do CyberMed é o correspondente as tarefas colaborativas, ou seja, o componente denominado *Collaboration* cujo módulo, explicado na sessão posterior, é chamado de *CybNetwork*.

2.2.1.O Módulo de Colaboração (CybCollaboration)

Neste item, iremos apresentar um estudo detalhado das principais classes e métodos deste módulo de maneira que possamos entender como ocorrem as trocas de mensagens entre os hosts. Basicamente, pode-se dividir o módulo *CybCollaboration* em três partes principais. A primeira formada pelas classes *CybCollaboration*, que descreve o protocolo e troca de mensagens entre os participantes; a *CybCollaborationProperties* que contém propriedades e configurações como o tipo da comunicação em rede, e forma de manipulação dos objetos, dentre outras, e por fim a *CybCollaborationIntegrant*, que representa cada integrante com as seguintes informações: o nome, o endereço IP e o interador. Estas classes contêm informações e operações a serem utilizadas por todas as outras classes e podem ser vistas como o núcleo do módulo. A segunda diz respeito às colaborações relativas aos dispositivos de interação e que são sub-classes da classe principal *CybCollaboration*. As classes *CybMouseCollaboration*, *CybHapticCollaboration* e *CybTrackerCollaboration* tem informações básicas sobre colaboração com os respectivos tipos de dispositivos [SAL10]. A terceira parte se refere às colaborações derivadas. Esta é formada por classes que derivam das classes de colaboração básicas e ajustam as características para formar um ambiente de colaboração mais personalizado. Por enquanto, apenas a classe *CybHapticAssistedCollab* está implementada neste nível [SAL10][PAI10].

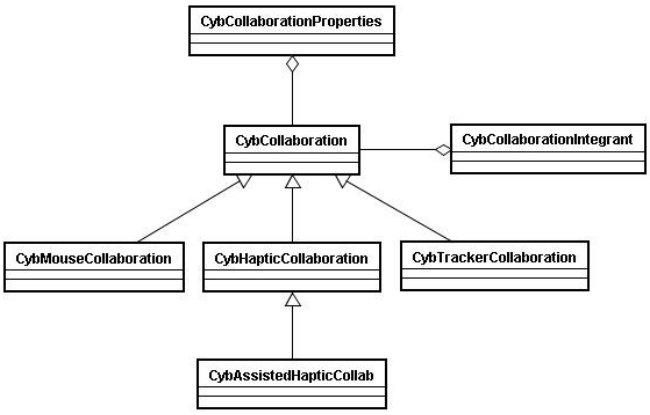

Figura 3: Estrutura das classes do módulo de Colaboração (CybCollaboration) do CyberMed [SAL10].

Atualmente o módulo está passando por adaptações para suportar colaborações com o uso de luvas de dados, ndo que uma aplicação de teste já foi desenvolvida com o auxílio do módulo (Figura 4).

Figura 4. AVC com luvas de dados desenvolvido para fins de testes

Antes de iniciadas as intervenções no módulo, com o intuito de aprimorarmos o conhecimento a respeito do protocolo definido neste módulo, foi necessário um estudo minucioso de seus principais métodos e dos relacionamentos entre todas as classes deste módulo. Nos próximo tópicos serão exibidas explicações detalhadas sobre tais classes.

a) Métodos da classe *CybCollaboration*

- CybCollaboration() – Atribue às instâncias do CybCore, do mouse, do socket, da thread de recepção de dados. O objeto de propriedades da colaboração (properties) é instanciado, passando para este as referências do socket, da lista de integrantes da colaboração e definindo, por padrão, o tipo de liberdade do AV como bloqueado (BLOCK_OBJECT_MANAGER).

- createCollaboration() – Método utilizado pelo "centralizador da colaboração" para iniciar a colaboração e permanecer em espera de requisições de participação dos clientes. Define a liberdade do AV no objeto de propriedades, instancia o objeto interador principal (*mainInterator*) que será monitorado pela colaboração. Depois, a colaboração e o objeto *properties* são cadastrados como ouvintes da thread de recepção de dados, que é logo inicializada.

- addIntegrant() – Adiciona um integrante na cena local. É criado um interador remoto que é atribuído ao novo integrante, sendo este carregado e desenhado localmente. Atribui uma instância do interador remoto e carrega-o a partir do CybDataObtainer. O tipo do interador do objeto integrante é definido para assim ser adicionado à lista de integrantes.

- getRemoteInterator() – Método que utiliza-se do CybInteratorFactory para retornar um objeto interador remoto, de acordo com o tipo passado como parâmetro.

- joinCollab() – Cria o interador local, monta a estrutura CybNodeAddress com os dados do servidor, monta o pacote de requisição de participação e envia para o servidor. Depois a colaboração e o properties são cadastrados como ouvintes da thread de recepção de dados, que é logo inicializada

- eventPerformed() – Método que monitora os eventos de rede, sendo chamado no thread de recepção de dados. Recebe como parâmetro os dados enviados e o endereço de quem os enviou. De posse destas informações, o método faz os devidos processamentos locais ou emite respostas ao comunicante, de acordo com o protocolo de colaboração definido.

- processJoinRequestMsg(buffer, IPCliente, portaCliente) – Processa a requisição de participação de um cliente. Segue os seguintes passos:

    1. Monta e envia o pacote de resposta com os campos contendo o flag ACCEPT e o tipo de interador do servidor;

    2. O servidor recebe os dados enviados pelo novo participante (name, nameLen, interatorType) e monta o pacote sendMessage com base nestas inforamções: SCENE_CONFIG (flag), número de integrantes, interatorType (tipo do interador), nameLen (tamanho em bytes do nome), name, ipAddrLen (tamanho em bytes do IP), ipStr (IP do cliente), port.

    3. Envia-o para todos os antigos participantes e envia as propriedades da colaboração para o novo participante.

    4. Varre a lista de participantes pegando suas informações e montado um novo pacote SCENE_CONFIG enviando-os para o novato.

    5. Cria o novato localmente (no server).

- processSceneConfigMsg(buffer, ipStr, port) - Pega do primeiro pacote recebido no buffer, o campo do n° de integrantes para usá-lo para percorrer a lista de integrantes. Para cada um deles, cada campo é atribuído a uma variável local que servirá para instanciar os integrantes remotos localmente.

Obs: O restante das funções são basicamente para o processamento dos pacotes recebidos e que visam a alteração do estado do AV, tal como a translação/rotação de objetos virtuais ou dos interadores.

b) Métodos da Classe *CybCollaborationProperties*

- CybCollaborationProperties() – Seta a variável serverNode, e de acordo com o objManType define a cena como modificável ou não, utilizando-se do cybCore, e se o objeto estará livre ou não. O leader, por padrão é definido como falso. Este pode ser definido via o setLeader() nas subclasses que desejarem colaboração guiada. O atributo boleano objectFree define a possibilidade de alterar a cena com cybCore->setSceneEditable().

- eventPerformed(char* buffer, CybNodeAddress) - Recebe dados de um *buffer* que foi enviado pelo endereço *address.* Verifica o 1° flag (tipo da mensagem), que podem ser três:

    1. OBJECT_REQUEST_MANAGER – Requisição para manipular objeto. Flag enviado pelo método serverRequestObjectManager() no próprio objeto local de propriedades. O cliente envia para o servidor este flag para requisitar a posse do objeto. Então o servidor libera-o, definindo localmente que ele não mais irá possuí-lo com setObjectFree(false).

2. OBJECT_LEAVE_MANAGER – Requisição para liberar objeto. Flag enviado pelo método serverLeaveObjectManager(). O cliente envia para o servidor este flag para liberar um objeto. Então o servidor retoma a posse do objeto definindo-o como livre com setObjectFree(true).
3. COLLAB_PROP_DATA – Define o tipo da manipulação (objectManagerType). De acordo com o tipo de manipulação é definido a liberdade com setObjectFree(). Um cliente ao entrar na colaboração recebe o tipo de liberdade da mesma do servidor, que usa o sendCollaborationProperties() no método processJoinRequestMessage().

- requestObjectManager() – Caso o objeto estiver livre (isObjectFree()), uma requisição de manipulação (OBJECT_REQUEST_MANAGER) para todos os participantes, que irão travar os seus objetos locais. Logo após, é liberado o objeto local.
- sendCollaborationProperties() – O centralizador da colaboração envia o seu *objectManagerType* local para um novo participante.
- setObjectFree(bool free) – Libera a cena mediante uso do método cybCore->setSceneEditable().

2.3 OPNET Modeler

Este tópico visa descrever as principais funcionalidades e conceitos a cerca do simulador de eventos discretos OPNET Modeler (OM) utilizado nas simulações do CyberMed para análise de desempenho de rede. A versão utilizada nos estudos foi a 14.5, versão educacional. Alguns dos principais conceitos a saber são:

a) Módulos – Os módulos representam as partes de um objeto da rede (ou nó) onde os dados serão gerados e processados [OPN10], ou seja, eles são as unidades básicas que compõe um nó da rede. Existem assim, diferentes tipos de módulos para diferentes funções. Existem, por exemplo, os chamados módulos transmissores (*transmitters*) e os receptores (*receivers*), que representam *interfaces* de rede utilizados na entrada e na saída de pacotes, mantendo conexão externa com os *links* de transmissão de dados. Existem também, os módulos de enfileiramento (*queues*) e os geradores (*generators*) que como seus próprios nomes revelam, servem para o enfileiramento e a geração de pacotes respectivamente. Também existem os módulos processadores (*processors*) que permitem a definição de uma sequência lógica de execução para processamento dos pacotes que chegam ou são gerados pelo módulo local.

b) Procedimentos do Kernel OM e a linguagem Proto-C – Os procedimentos do kernel do OM permitem desde operações básicas, como a criação e a transmissão de pacotes, como também operações mais complexas como a utilização de protocolos de roteamento e envio de pacotes por multidifusão. Outras funções auxiliam na criação e carregamento de Funções de Densidade Probabilísticas (PDF – *Probability Density Function*), no tratamento de atributos de objetos, de processos, de interrupções, dentre outras utilidades. Foi designado o termo *Proto-C* para o código utilizado dentro do OM, que nada mais é do que a junção das funções

do kernel do OM com a linguagem C/C++ pura. O *Proto-C* tem como função principal a definição do Diagrama de Transição de Estados que compõe os Modelos de Processos dos nós. Dentro de um Modelo de Processos, os trechos de código *Proto-C* podem ser incluídos nos seguintes locais:

- *Enter Executive* – Código executado quando o módulo entra em um estado
- *Exit Executive* – Código executado quando o módulo sai de em um estado.
- *Function Block (FB)* – Código das funções utilizadas.
- *Header Block (HB)* – Cabeçalho do código-fonte. São definidas as macros condicionais, as constantes e inseridas as bibliotecas.
- *Transition Executive* – Código executado quando uma condição pré-determinada é satisfeita. Geralmente uma condição é definida por meio de macros e diretivas no Header Block (HB).

c) Modelo de Processos – Diagrama de Transição de Estados **(***State Transition Diagram* – STD), ou seja, representa o comportamento lógico de um módulo. Uma STD define os estados de um módulo e quais critérios ou condições deverão ser satisfeitas para que haja a transição entre estes estados. Para definição da STD, o OPNET utiliza os conceitos de Estado e Transição, determinando assim quais ações serão tomadas em conseqüência aos eventos disparados. Os módulos processadores são definidos por uma sequência lógica que é denominada Diagrama de Transição de Estados (STD - *State Transition Diagram*) e é estabelecida através do uso de estados. A STD pode ser definida em linguagem C/C++ e por meio de funções disponíveis pelo OM que são reconhecidas pelo seu *Kernel, e* são também conhecidas como procedimentos do kernel (*Kernel Procedures*).

d) Estados – Representam a condição atual de um módulo. Os tipos de estados são infinitos e fica a cargo do desenvolvedor a concepção destes. Por exemplo, um módulo pode estar em estado de espera por pacotes, ou em espera pela recuperação de um link defeituoso. Estados de inatividade, atividade, inicial são os mais freqüentes. No OM, cada estado é representado por um círculo com o seu nome ao centro, como pode ser visto na Figura 5. Com um duplo-clique acima do nome do estado, será aberta a janela onde devem ser inseridas as Executivas de Entrada (*Enter Executives*), que são aqueles comandos executados assim que o fluxo de execução entrar no estado atual. Da mesma maneira, para definir os comandos a serem executados na saída do estado, o desenvolvedor deve definir as Executivas de Saída (*Exit Executives*) também com um duplo-clique, porém abaixo do nome do estado. Os tipos de estados são:

- **Estado não-forçado (*Unforced State*) –** Estado que retorna o controle da simulação para o *Kernel* da mesma, logo após serem executadas as executivas de entrada (*Enter Executives*). Ou seja, o estado é pausado na metade do seu fluxo de execução que será retomado da próxima vez que o Modelo de Processos for invocado.

- **Estado forçado (*Forced State*) –** Estado que executa ambas as suas executivas (de entrada e saída) e depois transita o fluxo de execução para o próximo outro estado.

- **Transições –** Mudança entre estados causada por um evento ou condição pré-estabelecida. As transições podem ser condicionais, quando ocorrem mediante a satisfação de uma condição e as não-condicionais, quando não há nenhuma condição imposta para que a ocorra a transição.

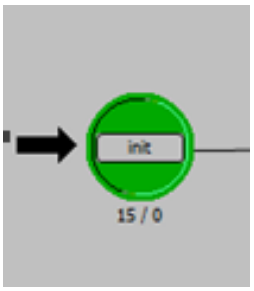

Figura 5. Estado de inicialização de um módulo

e) Eventos - As simulações do OM são possíveis graças aos eventos. Os Modelos de Processos respondem a eventos específicos e podem agendar a execução de novos eventos. Existem diferentes tipos de eventos e os mais comuns são as interrupções retornadas pelo *Kernel* da simulação.

- **Interrupção-** As interrupções são eventos disparados a fim de interromper o a execução do estado atual, permitindo que a simulação volte sua atenção para outras entidades e eventos do modelo. As interrupções podem ser geradas pela chegada ou saída de pacotes por uma das interfaces de rede, ou mesmo podem ser agendadas pelo próprio Modelo de Processos local, sendo neste caso denominadas de auto-interrupção (*self-interrupt*). Ao ser executado um evento qualquer, o módulo corrente é afetado e o *Kernel* da simulação passa o seu controle para o Modelo de Processos deste módulo via uma interrupção. O Modelo de Processos responde ao evento mudando de estado e executando o código associado, e por fim retorna o controle ao *Kernel* da Simulação. O modelo de processo ao entrar em um estado, primeiramente executa as suas Executivas de Entrada (*Enter Executives*) e caso o estado seja do tipo não-forçado, o modelo de processo encerra a execução e retorna o controle da mesma para a simulação.

f) Distribuições - As distribuições do OM são Funções Densidade de Probabilidade (*Probability Density Function - PDF*) e definem uma probabilidade sobre uma faixa de possíveis resultados e podem ser utilizadas para modelar o tempo de chegada de pacotes ou a probabilidade de transmissão de erros. Os parâmetros e estatísticas de um modelo, também são definidos e calculados com base em diferentes tipos de distribuições de valores, que são utilizados para gerar uma série de valores numéricos estocásticos. As distribuições são inicializadas através da função KP *op_dist_load()* que é utilizada para carregar um *set-up* inicial de valores de uma distribuição quando necessário [OPN10].

Capítulo 3. Comunicação em Rede

3.1. API *Berkeley Sockets*

De modo que pudéssemos entender o funcionamento do módulo de rede do CyberMed (CybNetwork) foi necessário um estudo de programação em redes utilizando soquetes. Deste modo, foi realizado um rápido estudo sobre comunicação em redes e implementados alguns exemplos de tais aplicações. A API utilizada foi a UNIX Berkeley Sockets. Esta API constitui uma biblioteca para desenvolvimento de aplicações, baseados na linguagem de programação C/C++, e que realizam comunicação entre processos locais ou remotos, mais comumente para comunicações através de uma rede de computadores. Os sockets permitem o envio e recepção de dados até que um dos processos em questão termine a ligação. Existem alguns tipos de sockets e os mais comuns são:

- *Stream sockets* (SOCK_STRM) - Implementam o envio de um fluxo contínuo de dados seguro, isto é, com garantia de entrega e integridade dos dados no lado receptor. Logo, eles representam o protocolo TCP/IP. Como exemplo, estes são bastante utilizados em comunicações de telnet e www, que exigem maior contorle sobre o fluxo de dados.
- *Datagram sockets* (SOCK_DGRAM)- Utilizado em uma comunicação mais veloz, porém insegura, sem a garantia de entrega e integridade dos pacotes. Representa o protocolo UDP/IP.
- *Raw sockets* (SOCK_RAW) – Processos de controle que gerenciam ou monitoram a infra-estrutura de rede.

Vale ressaltar que comunicações em rede baseadas no protocolo UDP são consideradas não-confiáveis uma vez que este protocolo não fornece um controle de fluxo maior como no caso do TCP, fornecendo apenas o serviço de multiplexação e demultiplexação. Não há noção de conexões, nem ordem de entrega, confiabilidade ou controle de congestionamento. Se o objetivo é enviar pacotes UDP, os pacotes podem ser simplesmente enviados no soquete a um endereço de host e porta especificados. Se o objetivo é receber pacotes UDP, o passo adicional de "ligação" a tomada tem de ser realizada. Esta etapa notifica a pilha de protocolo que deve começar a encaminhar pacotes destinados a essa porta para a aplicação. Além disso, como o TCP tem toda a sobrecarga de estabelecer e fechar conexões, UDP podem ser significativamente mais rápido e geralmente é bastante utilizado em aplicações de multimídia [HAL01].

3.1.1. Tipos de Dados

Algumas estruturas de dados especificas para comunicação em rede foram definidas pela biblioteca UNIX Berkeley Socket Interface. Algumas delas são:

in_addr - Representa o endereço IP.

```
struct in_addr {
        unsigned long s_addr; // 32 bits (4 bytes) para o endereço IP
```

}

sockaddr - Representa um *socket*. Para se comunicarem, dois *sockets* devem possuir o mesmo domínio e protocolo.

```
struct sockaddr{
        unsigned short sa_family; // Endereço do domínio
        char sa_data[14]; // Endereço do protocolo
}
```

sockaddr_in - Representa um *socket* para a Internet, e que geralmente deve ser convertido para o tipo *struct sockaddr.*

```
struct sockaddr_in {
        short int sin_family; // Endereço da família
        unsigned short int sin_port // Endereço da porta
        struct in_addr sin_addir; // Endereço IP
        unsigned char sin_zero[8]; // Usado para preencher as estruturas com 0
}
```

3.1.2. Trabalhando com IPs

inet_addr() – Função que converte um IP em um *long* que pode ser armazenado na estrutura do tipo *sockaddr* no campo s_addr. O long é retornado na notação Network Byte Order e não há necessidade de chamar a função htonl ("*Host to Network Long*"). Em caso de erro, -1 é retornado.

```
ina.sin_addr.s_addr = inet_addr("10.12.110.57");
```

inet_ntoa() – Imprime um IP na notação de pontos e números (ntoa = network to ascii**)**

```
printf("%s", inet_ntoa(ina.sin_addr));
```

3.1.3. Funções Básicas para Gerenciamento de Sockets

a) socket() – Criando um descritor de arquivo.

O descritor de arquivo é um inteiro identificador usado fundamentalmente em sistemas UNIX onde as aplicações utilizam arquivos para realizarem os seus processos. A chamada a função socket() retorna o descritor do arquivo que será usado nas comunicações em rede. Em caso de sucesso, a maioria das funções de sockets retornam 0.

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- domain- Trata-se do domínio (escopo). O mais utilizado é o AF_INET, igualmente utilizado na estrutura sockaddr_in, e significa que o escopo é a Internet.
- type – Tipo do socket: SOCK_DGRAM (udp) ou SOCK_STREAM (tcp).

- protocol – Escolha 0 para que a função socket defina o protocolo baseado no tipo passado.

b) bind() – Definindo uma porta para comunicação

Caso um servidor deseje escutar por conexões de clientes, este deve ligar uma porta local ao seu IP, porta esta que será usada pelos clientes para se conectar, ou seja, os pacotes recebidos são direcionados a uma porta associada a um processo de socket local. Dessa forma, as portas possibilitam que sockets rodem em paralelo desde que se conectem a diferentes portas. OBS! As portas devem ser superiores a 1024, uma vez que as inferiores são reservadas.

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define MYPORT 3490

main() {
        int sockfd;
        struct sockaddr_in my_addr;

        sockfd = socket(AF_INET, SOCK_STREAM, 0); // do some error checking!
        my_addr.sin_family = AF_INET; // host byte order
        my_addr.sin_port = htons(MYPORT); // short, network byte order
        my_addr.sin_addr.s_addr = inet_addr("10.12.110.57");
        memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

        bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
                …
```

Obs: Existe uma forma de automatizar a escolha do conjunto porta + IP, que é mostrado a seguir:

```
// escolha randomica de uma porta
my_addr.sin_port = htons(0);

// usar o IP local (INADDR_ANY)
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

Obs: Um erro possível de ocorrer é o "Address already in use". Este erro acontece por conta da porta escolhida que está em uso, possivelmente por outro *socket* em aberto. Uma forma de tratar tal erro seria uma chamada à seguinte função:

```
int yes =1;
if (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1) {
    perror("setsockopt");
    exit(1);
}
```

c) connect() – Conectando-se em um servidor, ou host remoto.

#include <sys/types.h>

#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);

- sockfd- É o meu descritor local
- serv_addr – É uma estrutura do tipo sockaddr contendo o endereço/porta de destino do servidor.
- addrlen – sizeof(sockaddr).
-

```
#define DEST_IP "10.12.110.57"
#define DEST_PORT 23

main() {
        int sockfd;
        struct sockaddr_in dest_addr; // will hold the destination addr
        sockfd = socket(AF_INET, SOCK_STREAM, 0); // do some error checking!
        dest_addr.sin_family = AF_INET; // host byte order
        dest_addr.sin_port = htons(DEST_PORT); // short, network byte order

        dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
        memset(&(dest_addr.sin_zero), '\0', 8); // zero the rest of the struct

        connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));

        …
}
```

Obs: Geralmente o bind() é chamado no servidor. Note que neste código cliente não há necessidade de saber a porta local, e sim, apenas a porta de destino na qual o cliente irá se conectar.

d) listen() – Escutar e gerenciar conexões clientes

int listen(int sockfd, int backlog);

- backlog- Número máximo permito de conexões na fila de entrada.

Obs: Para chamar listen(), se faz necessário realizar chamada à função bind(), para conectar uma porta local ao socket. A ordem correta de criação de socket servidor com TCP/IP seria: socket() → bind() → listen() ... accept().

e) accept() – Aceitando conexões clientes

O host remoto tenta conectar-se a uma porta do servidor (que está em escuta). As requisições são enfileiradas em espera da chamada accept() do servidor. Accept() retornará um novo descritor de arquivo que será usado apenas para a nova conexão estabelecida. O original permanece em escuta e o novo é usado para envio/recepção de dados.

#include <sys/socket.h>

int accept(int sockfd, void *addr, int *addrlen);

- addr – Ponteiro para uma sockaddr_in local, que receberá as informações do host remoto.
- addrlen – Tamanho em bytes do addr;

```
#define MYPORT 3490 // the port users will be connecting to
#define BACKLOG 10   // how many pending connections queue will hold

main() {
        int sockfd, new_fd; // listen on sock_fd, new connection on new_fd
        struct sockaddr_in my_addr; // my address information
        struct sockaddr_in their_addr; // connector's address information
        int sin_size;

        sockfd = socket(AF_INET, SOCK_STREAM, 0); // do some error checking!
        my_addr.sin_family = AF_INET; // host byte order
        my_addr.sin_port = htons(MYPORT); // short, network byte order
        my_addr.sin_addr.s_addr = INADDR_ANY; // auto-fill with my IP
        memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

        bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));

        listen(sockfd, BACKLOG);

        sin_size = sizeof(struct sockaddr_in);
        new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);

        …
```

f) send() e recv() – Envio/recepção de dados em um *stream*

int send(int sockfd, const void *msg, int len, int flags);

- sockfd – Descritor do socket
- msg – Buffer de armazenamento dos dados a serem enviados
- len – Tamanho em bytes do buffer
- flags – Flag para opções de envio. Normalmente setado em 0.

```
…

char *msg = "Olá mundo socket!";

int len, bytes_sent;

len = strlen(msg);

bytes_sent = send(sockfd, msg, len, 0);

…
```

Obs: A função send() retorna o número de bytes enviados e caso ele não coincidir com o tamanho do buffer passado, cabe ao programador enviar o restante dos bytes que faltaram.

int recv(int sockfd, void *buf, int len, unsigned int flags);

- sockfd – Descritor do socket
- buf – Buffer de armazenamento dos dados a serem recebidos
- len – Tamanho em bytes do buffer
- flags – Flag para opções de envio. Normalmente setado em 0.

g) sendto() e recvfrom() – comunicando-se com Datagramas

A principal diferença das funções de transmissão/recepção de dados com *streams* confiáveis (SOCK_STREAM) em relação aos datagramas, está no fato de ser necessário a passagem de dois parâmetros adicionais que dizem respeito ao host de destino, que podem ser visto nas seguintes assinaturas:

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen);
int recvfrom(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen);
```

h) getpeername() - Obtendo informações do host remoto

#include <sys/socket.h>

int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);

- sockfd – Descritor local.
- addr – Estrutura que receberá os dados do socket remoto.
- addrlen – Tamnho em bytes da estrutura do socket remoto

i) gethostname() - Obtendo informações do host local

#include <unistd.h>

int gethostname(char *hostname, size_t size);

- hostname – Nome da máquina local
- size – Tamanho em bytes.

3.1.4. Exemplo de Aplicação em Rede com Sockets UDP/IP

Alguns exemplos de aplicações desenvolvidas com finalidade de estudo são apresentados a seguir . As Figuras 6 e 7 exibem 2 aplicações que se comunicam com sockets UDP/IP.

```c
#define LOCAL_SERVER_PORT 150
#define MAX_BUFFER_LEN 100

int main() {
    int sockfd;
    struct sockaddr_in localAddr; //  Armazena dados do host e liga-o a uma porta
    struct sockaddr_in remoteAddr; // Passada para a função de recebimento
    int remoteAddrLen = sizeof(remoteAddr);
    char buffer[MAX_BUFFER_LEN];

    // cria socket do tipo datagrama
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);

    // monta endereço local ...
    localAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    localAddr.sin_family = AF_INET;
    localAddr.sin_port = htons(LOCAL_SERVER_PORT);

    // liga-o a uma porta
    bind(sockfd, (sockaddr *) &localAddr, sizeof(localAddr));

    while(1) {
        memset(buffer, 0x0, MAX_BUFFER_LEN);
        recvfrom(sockfd, buffer, MAX_BUFFER_LEN, 0, (struct sockaddr *) &remoteAddr, &remoteAddrLen);
        printf("HOST: %s enviou: %s ", inet_ntoa(remoteAddr.sin_addr), buffer);
    }
}
```

Figura 6. Aplicação servidora que cria um socket UDP/IP e o mantém em escuta na porta 150

```c
#define REMOTE_SERVER_PORT 150
#define MAX_BUFFER_LEN 100

int main(int argc, char **argv) {
    int sockfd;
    struct sockaddr_in localAddr; //  Armazena dados do host e liga-o a uma porta
    struct sockaddr_in serverAddr; // Passada para a função de envio de dados
    struct hostent *h; // Armazena informações do host remoto
    char buffer[MAX_BUFFER_LEN];

    h = gethostbyname(argv[1]);
    int serverAddrLen = h->h_length;
    if (!h) printf("host desconhecido");

    // cria socket do tipo datagrama
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);

    // monta endereço do servidor remoto, baseando-se em h
    serverAddr.sin_family = h->h_addrtype;
    serverAddr.sin_port = htons(REMOTE_SERVER_PORT);
    memcpy((char*) &serverAddr.sin_addr.s_addr, h->h_addr_list[0], serverAddrLen);

    //   liga-o a uma porta
    bind(sockfd, (sockaddr *) &localAddr, sizeof(localAddr));

    sendto(sockfd, buffer, MAX_BUFFER_LEN, 0, (struct sockaddr *) &serverAddr, sizeof(serverAddr));
}
```

Figura 7. Aplicação cliente que cria um socket UDP/IP e envia dados para a aplicação servidora

Obs: A constante INADDR_ANY equivale ao endereço 0.0.0.0 e serve para conectar uma determinada porta à todas as interfaces locais

3.1.5. Comunicação em Rede com Multicast

Se uma aplicação quer enviar e receber tráfego Multicast entre essas redes distantes, é comum a criação de um túnel de Multicast. O tráfego de multidifusão é então encapsulado em pacotes de unicast e enviado via unicast para um host na rede remota que suporte o multicast. Uma vez recebido por esse host, os pacotes de multicast são recuperados de dentro dos pacotes unicast. Isso requer software ou hardware especializado em ambas extremidades da rede [HAL01]. Para enviar dados a um grupo de multicast, uma aplicação deve especificar um endereço IP deste grupo que faz parte da classe D, ou seja, varia de 224.0.0.2 a 239.255.255.255, e uma interface de rede local. Estes são atribuídos a seguinte estrutura: struct ip_mreq multiReq; e representa uma requisição IGMP para participação de um grupo.

a) Setando o escopo de rede do Multicast com o campo TTL (Time-to-live)

O campo TTL (time-to-live), presente no cabeçalho IP, conta o número de saltos entre sub-redes que um pacote realiza até alcançar certo destino. Deste modo, o campo é decrementado em cada roteador que passa até alcançar 0 no destino. Na implementação do multicast faz-se necessário a definição do escopo de rede através deste campo. Por padrão um pacote Multicast nunca será roteado para fora da sub-rede local se o campo TTL não for incrementado. O roteador com suporte a Multicast verifica o campo TTL. Caso 1, o pacote não é redirecionado para outra subrede, se maior que 1 o TTL é decrementado. Logo, o TTL quando é igual a 1 restringe o pacote a rede local e pode ser definido assim para testes locais, por exemplo como pode ser visto no trecho de código da Figura X. Para defini-lo, basta uma chamada a seguinte função:

int setsockopt(int socket, int level, int optName, void *optVal, unsigned int optLen)

- socket – Descritor do socket
- level – O parâmetro level pode receber os valores:
    - SOL_SOCKET: opções aplicadas a nível de socket
    - IPPROTO_TCP: opções aplicadas a nível de camada de transporte (TCP)
    - IPPROTO_IP: opções aplicadas a nível de camada de rede (IP)
- optName – Opções do socket
- optVal – O flag da opção definida
- optLen – O tamanho em bytes do flag passado

Nas aplicações de Multicast com escopo de redes locais, pode-se usar o seguinte código para setar o escopo de rede como LAN:

```
int sock; /* socket descriptor */
unsigned char ttl; /* time to live (hop count) */
int ttl_size;
```

```
…

ttl_size = sizeof(ttl);
if ((getsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, &ttl_size)) < O) {
        …
}
printf("Antigo valor TTL: %d\n",ttl);

ttl=S;
if ((setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, (void*) &ttl, ttl_size)) < O) {
        …
}
```

b) Definindo Interface de Loopback

A opção de socket IP_MULTICAST_LOOP pode ligar ou desligar a opção de *loopback* do multicast. Com o loopback habilitado, o emissor receberá os pacotes que ele enviar ao grupo de multicast, e caso esteja desabilitado ele não receberá suas próprias transmissões. O argumento de opção para IP_MULTICAST_LOOP é um unsigned char e o valor deve ser definido como 1 para habilitá-lo ou 0 para desabilitá-lo. Aqui está um exemplo simples que recupera o valor da opção *loopback* do multicast e define como o oposto do que era.

```
int sock;
unsigned char loopback;
int loopback_size;

// … crier socket aqui

loopback_size = sizeof(loopback);

// acessar o valor de loopback atual
if ((getsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loopback, &loopback_size)) < O) {
        perror("getsockopt() failed");
        exit(l);
}
if (loopback == O) {
        printf("Loopback is off\n");
        loopback = i;
} else {
        printf("Loopback is on\n");
        loopback = O;
}
/* setar o novo valor de loopback */
if ((setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loopback, loopback_size)) < O) {
        perror("setsockopt() failed");
exit(l);
}
```

c) Reutilizando sockets localmente

No multicast, o paradigma é um para muitos, e pode ser completamente válido para vários processos na mesma máquina receber os mesmos pacotes na mesma porta. A fim de contornar este problema, uma das duas opções de socket (ou SO_REUSEPORT ou SO_REUSEADDR) pode ser definida para indicar que mais de uma chamada à função bind() é permitido para o mesmo socket. A seguir pode ser visto um exemplo simples que define a opção tomada para permitir que uma porta seja reutilizada:

```
int sock;
int flag = I;

if ((setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &flag, sizeof(flag))) < O) {
        …
}
```

d) Enviando dados via Multicast

Os seguintes passos devem ser realizados para uma aplicação que deseja enviar dados a um grupo de multi-cast (Figura 8):

1. Cria estrutura com dados do grupo (IP e porta)
2. Para enviar os dados ao grupo não precisa dar o bind()
3. Enviar os dados ao grupo com a função sendto()

e) Recebendo dados via Multicast

Para receber dados Multicast uma aplicação deve realizar os seguintes passos (Figura 9):

1- Cria estrutura local e ligando-a à uma porta com a função bind()
2- Cria a estrutura de requisição IGMP passando o IP e a estrutura do grupo de multicast
3- Requisição de participação ao grupo
4- Recebe dados com recvfrom()

```c
#include <sys/types.h>
#include <sys/socket.h>
#define MAX_LEN 1000
#define PORT 9000
#define MULTICAST_IP "224.0.0.1"

int main() {
    int sockfd, msgLen;
    struct sockaddr_in multiAddr;
    unsigned char ttl = 1; //campo time-to-live
    char msg[MAX_LEN];

    // 1 - cria socket
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    // 2 - Define o escopo de rede
    setsockopt(sockfd, IPPROTO_IP, IP_MULTICAST_TTL, (void) &ttl, sizeof(ttl));
    // 3 - Monta estrutura do enderelo local multiAddr com o IP do grupo multicast
    multiAddr.sin_addr.s_addr = htonl(MULTICAST_IP);
    multiAddr.sin_family = AF_INET;
    multiAddr.sin_port = htons(PORT);
    // 4 - Envia dados em loop
    memset(msg, 0, sizeof(msg));

    while(fgets(msg, MAX_LEN, stdin)) {
        msgLen = strlen(msg);
        sendto(sockfd, msg, msgLen, 0, (struct sockaddr *) &multiAddr, sizeof(multiAddr));
    }
}
```

Figura 8. Aplicação que envia dados a um grupo Multicast

```c
#define MAX_LEN 1000
#define PORT 9000
#define MULTICAST_IP "224.0.0.1"

int main() {
    int sockfd, flag_on = 1;
    struct sockaddr_in multiAddr; // endereço local
    struct sockaddr_in remoteAddr; // para receber dados
    struct ip_mreq multiReq;
    char msg[MAX_LEN];
    int bufferLen;
    unsigned int remoteAddrLen;

    // 1 - cria socket
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    // 2 - Define o escopo de rede
    setsockopt(sockfd, IPPROTO_IP, IP_MULTICAST_TTL, (void) &ttl, sizeof(ttl));
    // 3 - Monta estrutura do enderelo local multiAddr com o IP do grupo multicast
    multiAddr.sin_addr.s_addr = htonl(MULTICAST_IP);
    multiAddr.sin_family = AF_INET;
    multiAddr.sin_port = htons(PORT);

    // 4- Liga o enderço a uma porta local
    bind(socket, (struct sockaddr *) &multiAddr, sizeof(multiAddr));

    // 5 - Constrói requisição IGMP
    multiReq.imr_multiAddr.s_addr = inet_addr(MULTICAST_IP);
    multiReq.imr_interface.s_addr = htonl(INADDR_ANY);

    // 6 - Evia a mensagem de requisição para participar do grupo multicast
    Setsockpt(sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP, (void*) &multiReq, sizeof(multiReq));

    while(1)    {
            recvfrom(sockfd, buffer, MAX_LEN, 0, (struct sockaddr*) &remoteAddr, &remoteAddrLen);
    }
}
```

Figura 9. Aplicação receptora de dados multicast

Capítulo 4. Programação Concorrente com *lpthreads*

Como requisito básico para o estudo da programação em rede, foi necessário o estudo de programação concorrente usando threads, uma vez que os ambientes virtuais colaborativos realizam múltiplas tarefas ao mesmo tempo. A biblioteca estudada foi a *lpthread* do UNIX, pelo fato de esta ser já bem difundida e ter sido utilizada na implementação da classe CybThread do CyberMed.

4.1. O que são Threads?

Uma *thread* pode ser definida como um fluxo independente de instruções que pode ser agendado para ser executado pelo sistema operacional. A implementação de threads e processos difere entre os sistemas operacionais, mas geralmente as *threads* estão contido dentro de um processo. Múltiplas *threads* podem existir dentro do mesmo processo e compartilhar recursos, como a memória, enquanto que os diferentes processos não compartilham desses recursos. Em particular, as *threads* de um processo compartilham futuras instruções deste último (seu código) e de seu contexto (os valores que suas variáveis referenciam em um dado momento). Por exemplo, para fazer uma analogia, múltiplas threads em um processo são como múltiplos cozinheiros que lêem ao mesmo tempo um livro de culinária e seguem suas instruções, não necessariamente da mesma página [POS01]. Desta forma as diferentes threads de um sistema servem para otimizar tarefas e muitas vezes economizando recursos de memória.

4.1.1. pthread_create – Criando Threads

O número máximo de *threads* que podem ser criadas por um processo é dependente da implementação e de como estas estarão utilizando os recursos de memória e processamento. Uma vez criados, as *threads* são pontuais, e podem criar outras *threads.* Não existe uma hierarquia implícita ou dependência entre threads.

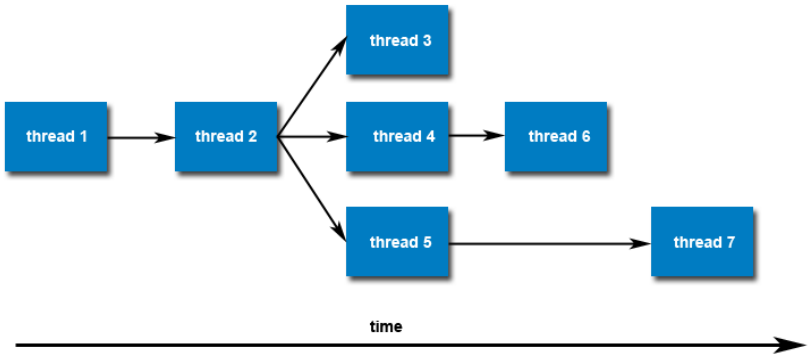

Figura 10. Linha de tempo da execução de um programa *multithreaded* [POS01]

```
#include <pthread.h>

int pthread_create(pthread_t thread, const pthread_attr_t attribute, void *(*start_routine)(void*), void arg);
```

- thread – Endereço de uma variável que identificará a thread
- attribute – Estrutura do tipo pthread_attr_t que contém os atributos de uma thread
- start_routine – O endereço da função que será executada pela thread
- arg – Os argumentos da função passada no campo anterior.

4.1.2. pthread_attr_t – Atributos de uma Thread

Por padrão, um *thread* é criado com determinados atributos e podem ser alterados pelo programador através do objeto de atributo da *thread* definido pela estrutura *pthread_attr_t*. Para manipulá-la o programador dispõe de algumas funções como a pthread_attr_init() e a pthread_attr_destroy() e são usados para inicializar e destruir este atributo, respectivamente. Outras rotinas são então utilizados para consulta e definição de atributos específicos. Dentre os atributos podemos citar os parâmetros de programação, o tamanho e endereço da pilha de execução dentre inúmeros outros. O atributo da *thread* também contém uma opção que define o escopo de agendamento das threads, isto é, se serão agendados pelo kernel do sistema operacional ou por outras *threads* e pode ser definido através com o seguinte comando:

```
status = pthread_attr_setscope(&attribute, PTHREAD_SCOPE_SYSTEM);
```

4.1.3. pthread_exit() – Encerrando Threads

Esta função permite que uma *thread* filha, ou seja, criada por uma thread pai, possa retornar dados que foram processados por ela. Para isso, basta que os valores sejam passados por parâmetro no fim da thread e assim que esta for encerrada, os dados serão retornados para a thread pai. A sua assinatura e parâmetros são:

```
#include <pthread.h>

void pthread_exit(void *value_ptr);
```

- value_ptr – ID da thread a ser encerrada

4.1.4. Passando argumentos para as Threads

A função pthread_create () permite que o programador passe argumentos para a função que executará a thread. Para casos em que múltiplos argumentos deverão ser passados, essa limitação é facilmente superada através da criação de uma estrutura que contenha todos os argumentos em seus campos, e depois passar um ponteiro para essa estrutura na chamada à função pthread_create(). Todos os argumentos devem ser passados por referência e transformados para um ponteiro do tipo (void *) por *casting*. A Figura 11, apresenta um exemplo de código em que uma estrutura é passada como argumento para várias threads. Neste exemplo, a função PrintHello() receberá uma estrutura do tipo thread_data que contém o ID da thread, um inteiro para contagem das threads e uma string que será exibida em uma mensagem. No main(), todas as threads são criadas em loop, preenchem suas estruturas do tipo thread_data e executam a função PrintHello().

```c
#include <pthread.h>
#include <iostream>
using namespace std;

#define NUM_THREADS 5

// estrutura que será passada para as threads struct
thread_data {
    int thread_id;
    int sum;
    char *message;
};

// Array de estruturas struct thread_data thread_data_array[NUM_THREADS];
void *PrintHello(void* threadarg) {
    struct thread_data *my_data;
    int taskid, sum;
    char* hello_msg;

    // Atribue os campos da estrutura recebida
    my_data = (struct thread_data*) threadarg;
    taskid = my_data->thread_id;
    sum = my_data->sum;
    strcpy(hello_msg,my_data->message);
    printf("Hello World! Sou thread ##%d!\n", taskid);
    printf("Sum = %d", sum);
    printf("Message = %s", hello_msg);
```

```
28
29 pthread_exit(NULL);
30 }
31
32 int main(int argc, char** argv) {
33     pthread_t threads[NUM_THREADS]; // vetor de threads
34     int rc; // codigo de retorno da função create()
35     long t;
36     int sum = 0;
37
38     for(t=0; t<NUM_THREADS; t++) {
39         sum += 1;
40         // Preenchimento dos campos da estrutura
41         thread_data_array[t].thread_id = t;
42         thread_data_array[t].sum = sum;
43         strcpy(thread_data_array[t].message, "thread");
44         printf("In main: creating thread %ld\n", t);
45
46         // Cria thread, passando a estrutura para a função PrintHello
47         rc = pthread_create(&threads[t], NULL, PrintHello, (void*) &thread_data_array[t]);
48         if(rc)
49                 printf("ERROR!! return code from pthread_create() is %d\n", rc);
50     }
51     // finaliza thread
52     pthread_exit(NULL);
53 return 0;
54 }
55
```

Figura 11. Programa que envia estruturas do tipo *thread_data* como parâmetro para a função PrintHello que executam as *threads*

4.1.5. pthread_join() – "Acoplando" Threads

O "acoplamento" entre threads é uma forma simples de sincronismo entre as threads e determina que uma thread pai, ou seja, a thread que cria as outras e que geralmente é a função main(), deva esperar o término da execução das threads filhas. Cada thread filha ao encerrar pode retornar os dados que foram processados por ela para a thread pai. Para isto, a thread pai deve fazer uma chamada à função *pthread_join()* que recebe dois parâmetros: o ID da thread a ser esperada e o endereço da variável que receberá o retorno da thread:

status = pthread_create(&vet[0], &attribute, run, (void*) 1L);

status = pthread_join(vet[0], &retorno);

Desacoplar uma thread pai da filha significa que será permitido que os seus recursos sejam liberados e portanto ela não pode ser "acoplada":

status = pthread_detach(vet[0]);

Por padrão as threads são "acopláveis" e a próxima instrução define que o estado da thread a ser criada será desacoplado, ou seja, que não será retornado nenhum dado para a thread pai.

```
status = pthread_attr_setdetachstate(&attribute, PTHREAD_CREATE_DETACHED);
```

Como exemplo, a Figura 12 exibe um programa em que cinco threads manipulam um espaço de memória compartilhado representado pela variável inteira *global*. A thread principal main() cria todas as outras e espera o encerramento das mesmas. Todas as threads executam a função threadBody() que executa os seguintes passos: 1- imprime uma mensagem contendo o seu identificador; 2- permace em estado de espera por três segundos; 3- incrementa a variável global; 4- imprime outra mensagem informando o término da thread; 5- por fim encerra a thread retornando o novo valor da variável global.

```c
#include <lpthread.h>
#include <iostream>
using namespace std;

#define NUM_THREADS 5

long global = 10; // variável global

// função que incrementa a variável global e a retorna para o main()
void* threadBody(void *id) {
    long x, tid = (long) id;
    printf("\nInicio da Thread %ld", tid);
    sleep(3);
    global++;
    printf("\nFim da Thread %ld, retornando global = %ld", tid, global);
    pthread_exit((void*)global);
};

int main (int argc, char *argv[]) {

    pthread_t threads[NUM_THREADS]; // Vetor de threads
    pthread_attr_t attr;     // atributo de thread
    long i, status;
    void *temp;

    // Inicializando o atributo das threads
    pthread_attr_init(&attr);
```

```c

    // Definindo o estado das threads como acopláveis
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    // Criação das threads
    for(i=0; i<NUM_THREADS; i++)    {
            printf("\nNo main: criando a thread-%ld", i);
            status = pthread_create(&threads[i], &attr, threadBody, (void*)i);
    }
    // Acoplando as threads ao main() e exibindo a variável global
    for(int i=0; i<NUM_THREADS; i++)    {
            pthread_join(threads[i], &temp);
            printf("\nNo main: resultado da thread %d = %ld", i, (long)temp);
            pthread_attr_destroy(&attr);
    }
    pthread_exit(NULL);
};
```

Figura 12. Programa exemplo de como utilizar a função pthread_join()

Capítulo 5. Provendo Liberdade de Comunicação em Rede para o *CybCollaboration*

Este Capítulo visa descrever o processo de modificações por qual passou o módulo de colaboração do CyberMed. O módulo anteriormente disponibilizava apenas comunicações em rede via *unicast* em conjunto do protocolo UDP/IP sem que fosse possibilitando ao programador uma escolha entre outras formas como o *multicast, broadcast* e o uso do protocolo de transmissão TCP/IP, por exemplo. O módulo também não dispunha de uma ferramenta que auxiliasse na monitoração do tráfego de rede gerado, como contabilização dos pacotes enviados e recebidos durante as colaborações. Deste modo, algumas alterações nas classes já existentes dos módulos de rede e colaboração foram necessárias, bem como a inclusão de novas classes.

5.1. Implementando o Multicast no *CybNetwork*

Para possibilitar o uso do protocolo multicast no módulo de colaboração tornou-se necessário a implementação de um nó que também possibilitasse o envio de dados por unicast, uma vez que determinadas mensagens são direcionadas apenas ao servidor da aplicação. Deste modo, a classe *CybMulticastNode* foi criada e adicionada ao módulo de rede, visto que este módulo ainda não previa suporte ao uso de Multicast em uma arquitetura ponto a ponto. Logo, esta classe visa simular um nó que deve pertencer a um grupo de multicast, sendo este capaz de enviar e receber dados via UDP/IP.

A *CybMulticastNode* herda da classe *CybUDPServer* possibilitando-a usar os métodos de transmissão/recepção de dados já definidos na super-classe. Em seu construtor, recebe a porta que será conectada: à estrutura do grupo multicast para transmitir dados ao grupo; e à estrutura referente ao socket UDP que contém o IP local para realizar o bind() e receber dados. Logo, como podemos observar, que ao instanciarmos um objeto da classe *CybMulticastNode* estamos automaticamente instanciando um objeto do tipo da super-classe *CybUDPServer* constituindo dois sockets, sendo um para envio de dados ao grupo multicast e o outro para o envio via unicast para um determinado host na rede. Alguns métodos essenciais para a comunicação via multicast foram acrescentados às antigas classes *CybMulticastServer* e *CybMulticastClient* bem como inseridos na nova classe anteriormente apresentada. Estes métodos são descritos a seguir:

- setTTL() – Define o campo do cabeçalho IP chamado TTL (time to live) que determina o tempo de vida do pacote em termos de saltos entre redes, ou seja, determina o escopo de rede (LAN, WAN etc...) da aplicação.
- joinGroup() – Requisição IGMP para se cadastrar em um grupo de multicast.
- leaveGroup() – Requisição para abandonar determinado grupo de multicast.

- setSockReusableOpt() – Permite a reutilização do mesmo socket para que a aplicação conecte-se a várias portas locais.
- setLoopbackOption() – Define se os dados enviados para um grupo de multicast serão retornados para o emissor. Recebe os inteiros 1, para habilitar e 0 para desabilitar tal opção.

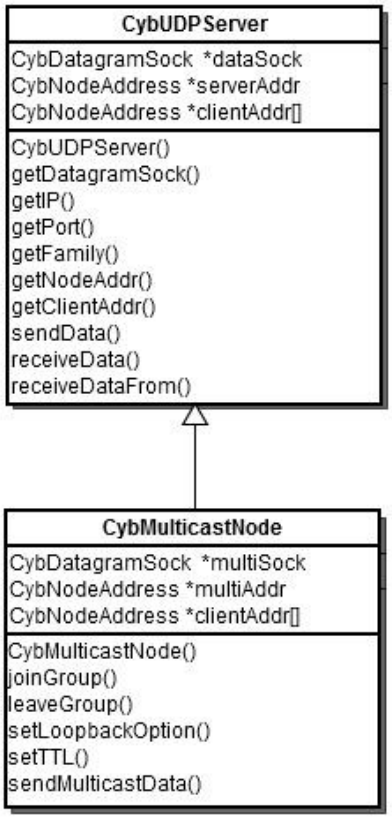

Figura 13. A nova classe *CybMulticastNode* foi acrescentada como subclasse de CybUDPServer de modo que seja possível o envio de dados também por unicast

5.2. Alterações no *CybCollaboration*

Este tópico visa descrever as alterações feitas no módulo de colaboração do CyberMed a fim de prover ao desenvolvedor a liberdade de escolha da comunicação em rede para as colaborações. Os modelos de comunicação em rede são disponibilizados pelo módulo de rede (CybNetwork) baseando-se em uma arquitetura distribuída, sem a presença de um servidor. Podem ser escolhidos os protocolos de comunicação Multicast, UDP/IP, TCP/IP e futuramente também o Broadcast.

Para que o desenvolvedor pudesse ter liberdade de escolha entre os protocolos citados anteriormente no CyberMed foi necessário alguns ajustes nas principais classes do módulo de colaboração (CybCollaboration). As classes alteradas foram a *CybCollaboration*, que implementa o protocolo de comunicação com base nas informações geradas pelos participantes e a *CybCollaborationProperties,* que

mantém os estados e propriedades tais como o estado de manipulação de objetos, do protocolo de rede, se a colaboração é do tipo livre ou guiada dentre outros fatores.

O socket que antes era definido rigidamente na classe da super-classe da colaboração passou a utilizar um objeto fábrica que retorna as instâncias em tempo de execução. Para tal fim, foi desenvolvida e adiciona a classe *CybNodeFactory* baseando-se no padrão de projeto *AbstractFactory* e tem como função instanciar, configurar e retornar um *socket* de acordo com o tipo de comunicação desejada em tempo de execução. Nesta classe, foi especificado o novo tipo de dado *TCommunicastionType*, de modo a referenciar o tipo de comunicação desejado:

```
typedef enum TCommunicationType {
      UNICAST_UDP = 0,
      UNICAST_TCP = 1,
      MULTICAST = 2,
      BROADCAST = 3,
} TCommunicationType;
```

Assim, o *socket* retornado é armazenado na referência da classe CybNode mantida como atributo nas classes da colaboração, e o tipo desejado é armazenado no novo atributo communicationType da classe *CybCollaborationProperties*, de modo que todo o módulo de colaboração tenha conhecimento do modo de comunicação em rede vigente.

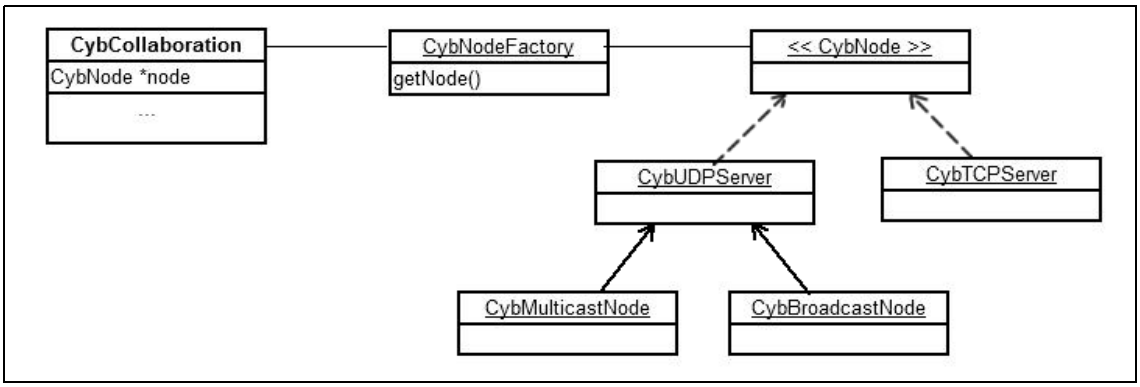

Figura 14. Nova forma de instanciação do *socket* usado na colaboração. Agora a classe *CybCollaboration* usa o método getNode() da *CybNodeFactory* para instanciar o socket em tempo de execução.

Deste modo, foi possível prover liberdade na escolha da comunicação de rede que antes estava programada rigidamente (com *sockets* UDP/IP) e provendo liberdade de escolha para o programador. Para tal fim, apenas torna-se necessário passar o tipo de comunicação desejada como parâmetro no construtor do objeto da colaboração *CybCollaboration*, como pode ser visto na Figura 17. A transmissão de dados é feita, baseando-se no tipo de comunicação em rede definido, e com chamadas ao método sendDataToAll(). Este método reccebe como parâmetros o buffer que contém o pacote, o tamanho em bytes do buffer e o tipo de comunicação em rede desejado, como pode ser visto em sua implementação a seguir:

```
1  void CybCollaboration::sendDataToAll(char *buffer, int bufferLen, int netCommunicationType) {
2
3      char type = buffer[0];
4
5      // Send data according to a Multicast network communication approach
6      if (netCommunicationType == MULTICAST) {
7           CybMulticastPeer *serverNode = (CybMulticastPeer*) node;
8           serverNode->sendMulticastData(buffer, bufferLen);
9           performance->countTrafficSent(type, bufferLen);
10
11     // Send data according to a Unicast network communication approach
12     } else if (netCommunicationType == UNICAST_UDP) {
13
14          CybUDPServer *serverNode = (CybUDPServer*) node;
15
16          for (it = integrants.begin(); it != integrants.end(); ++it) {
17               CybNodeAddress* addr = (*it)->getAddress();
18               try {
19                    serverNode->sendData(buffer, bufferLen, addr);
20               } catch (CybCommunicationException e) {
21                    e.showErrorMessage();
22               }
23                    performance->countTrafficSent(type, bufferLen);
24          }
25     }
26 }
```

Figura 15. Implementação da função sendDataToAll() pertencente à classe *CybCollaboration*, responsável pelo envio dos pacotes de acordo com o protocolo de rede escolhido

5.3. Monitorando o tráfego de rede

Para fins de futura avaliação de desempenho de rede nas aplicações colaborativas, foi criada a classe *CybCollaborationPerformance* que contabiliza todos os pacotes gerados, enviados e recebidos pelo módulo de colaboração, dividindo-os por tipo de mensagens e registrando o número total de pacotes/bytes enviados e recebidos. Assim, é possível a monitoração do tráfego de rede e a geração de um relatório ao término das colaborações contabilizando todo o tráfego de rede gerado e processado. Deste modo, uma instancia da classe de desempenho foi inserida na classe *CybCollaboration* e a contabilização do tráfego é realizada mediante chamadas aos principais métodos countTrafficSent() e countTrafficReceived() que recebem como parâmetros o tipo do pacote e o tamanho do buffer. Para a geração do relatório foi definido uma classe a parte. A classe *TrafficSniffer* constitui uma thread que aguarda a coleta de dados de rede durante determinado tempo, para logo em seguida apresentar o relatório de performance através de uma chamada ao método reportPerformance() do objeto de performance.

```
class TrafficSniffer : public CybThread {

private: CybCollaboration *collab;
         int collectTime;
         int time;

public: // Receive the collaboration object and time for monitoring the network traffic
         TrafficSniffer(CybCollaboration *collab, int collectTime) {
                 this->collab = collab;
                 this->collectTime = collectTime;
                 time = 0;
        }

        void run() {
                 this->setTime(1000);
                 cout << time << endl;

                 if (time > collectTime) {
                         collab->getCollaborationPerformance()->reportPerformance();
                         delete collab;
                 }
                 time++;
        }
};
```

Figura 16. Thread que monitora o tráfego de rede para fins de avaliação da performance das aplicações colaborativas

Para utilizar a função de geração de relatório para análises da performance de rede nas aplicações finais, basta criar uma instância da thread *TrafficSniffer* passando ao seu construtor o objeto da colaboração e o tempo de monitoramento e logo após inicializá-la:

a)
```
CybHapticCollaboration* cybCollaboration = new CybHapticCollaboration(5001, UNICAST_UDP);
TrafficSniffer *sniffer = new TrafficSniffer(cybCollaboration, 20);
sniffer->init();
...
```

b)
```
CybHapticCollaboration* cybCollaboration = new CybHapticCollaboration(5001, MULTICAST, "224.0.0.1");
TrafficSniffer *sniffer = new TrafficSniffer(cybCollaboration, 60);
sniffer->init();
...
```

Figura 17. Trecho de código para criar uma aplicação de colaboração háptica com comunicação em rede por (a) *udicast* ou (b) *multicast.* Ambas incluem a monitoração de tráfego de rede.

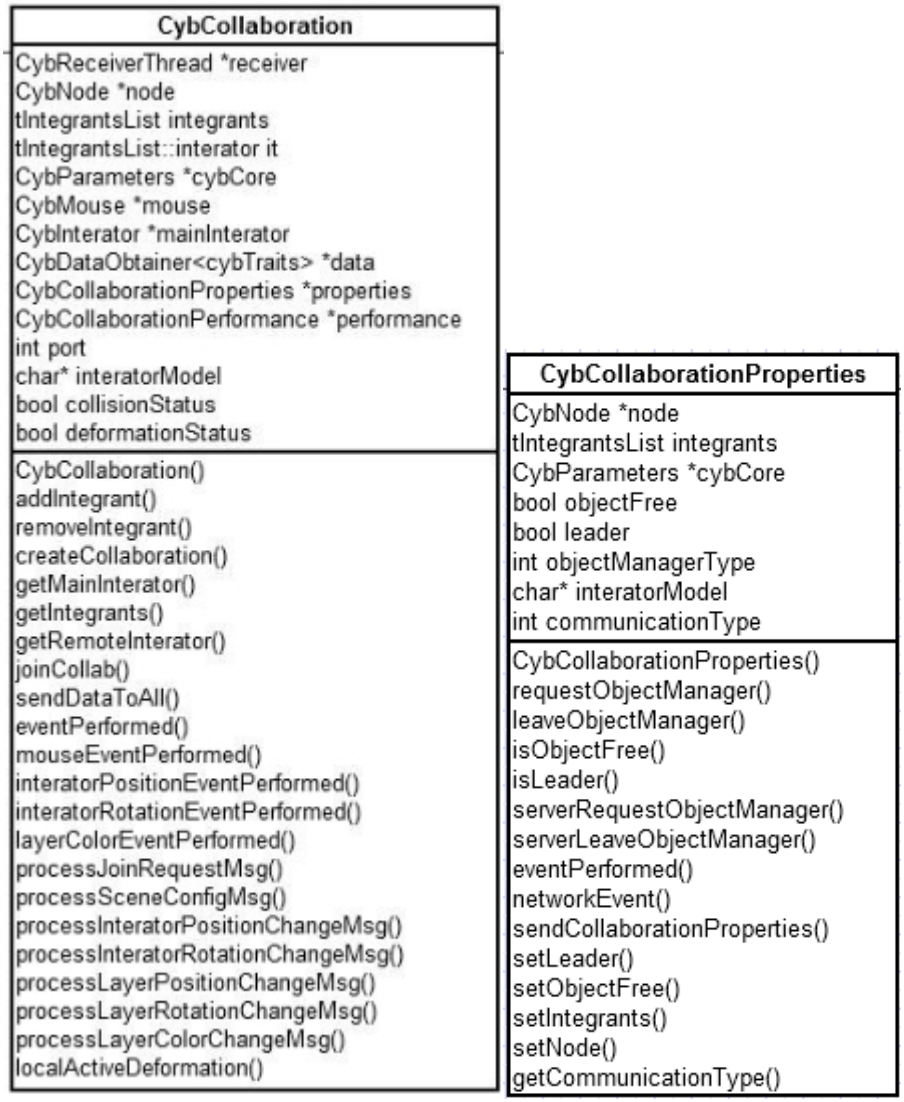

Figura 18. Alterações nas classes de colaboração*:* (a) inserção do método sendDataToAll() e dos atributos performance e node; (b) substituição do método setNode() pelo antigo setUDPNode() e inserção do método getCommunicationType() e do atributo communicationType

Por fim foram desenvolvidas aplicações de teste com o multicast e podem ser vistas na Figura 19, sendo a primeira delas com o uso do mouse e a segunda com dispositivos hápticos.

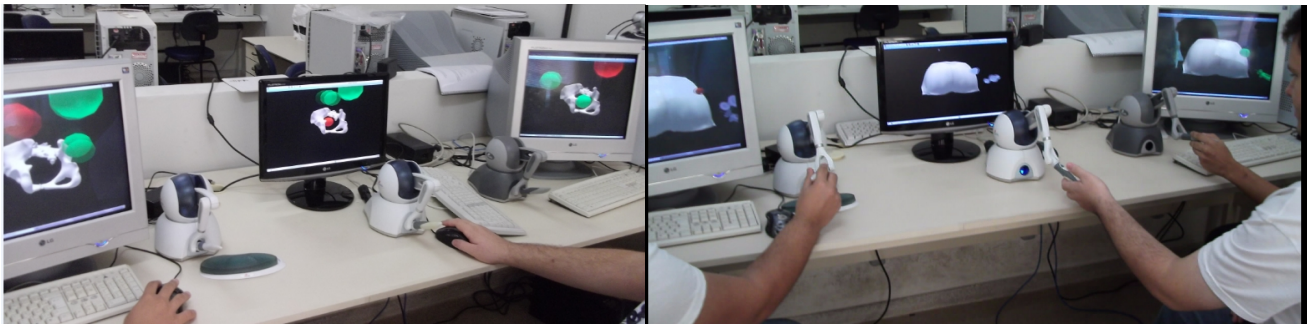

Figura 19. Aplicações de colaborações com Multicast com o uso do mouse e dispositivos hápticos

Capítulo 6. Avaliação de Desempenho do *CybCollaboration*

Para simular o comportamento de uma aplicação háptica desenvolvida a partir do *framework* CyberMed, foi necessário primeiramente o levantamento dos aspectos essenciais do seu módulo de colaboração *CybCollaboration*, bem como das mensagens enviadas pelo dispositivo háptico utilizado nos estudos. Os requisitos que foram escolhidos para representar a aplicação foram: o protocolo de colaboração utilizado pelo CyberMed e os pacotes envolvidos, as taxas de transmissão do PHANToM Omni e os protocolos para transmissão via Internet utilizados.

6.1. Descrição dos Modelos de Simulação

O modelo de nós criado para simular os participantes envolvidos é composto por quatro módulos descritos a seguir:

- *haptic* – Este módulo atua como gerador de pacotes. O seu modelo de processo chamado *Bursty_src_phantom_process*, atua mediante o estado inativo e o ativo, a fim de simular as rajadas de pacotes características destes dispositivos.

- *application* – Responsável pela simulação da aplicação, ou seja, o protocolo de colaboração do Cyber-Med responsável pelo acordo no estabelecimento e encerramento das conexões entre os clientes, envio, recepção e processamento dos pacotes de mídia háptica gerados pelo módulo anterior. Com esta finalidade, foi desenvolvido o modelo de processo *Cybermed-process.*

- *transmitter* – Módulo que representa uma interface de rede para envio de dados.

- *receiver* **–** Módulo que representa uma interface de rede para recepção de dados.

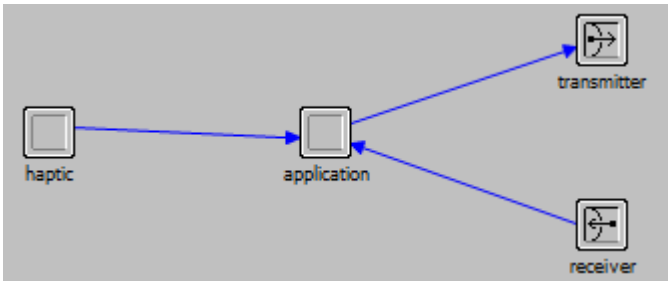

Figura 20. Modelo do nó dos clientes da colaboração háptica e seus respectivos módulos e *streams*

Como pode-se observar na Figura 1, foram definidos além dos módulos, os fluxos de pacotes (*Packet Streams*) de criação (GNR_STRM), entrada (RCV_STRM) e saída (OUT_STRM) de pacotes.

6.1.1.O processo do Módulo application (Cybermed-process)

O *Cybermed-process* é o modelo de processo, ou seja, é o Diagrama de Transição de Estados (STD) criado com o intuito de simular os principais estados e eventos envolvidos em uma aplicação colaborativa desenvolvida com o CyberMed. Este possui basicamente dois estados: inicial (*init)*, e o inativo (*idle).* O primeiro estado é do tipo forçado (*forced*), isto é, ele executa ambas as suas executivas (de entrada e saída), sem que ocorram pausas e logo após, ocorre a transição para o estado posterior. Desta forma, o estado *init* é executado apenas uma vez no início da execução do processo, e nele ocorre a inicialização de variáveis responsáveis pela coleta de estatísticas, a leitura dos atributos *Node Address* e *Leader,* que são utilizados na identificação e no estabelecimento da função (cliente ou líder) de cada nó respectivamente. Ainda no estado *init,* através da KP *op_stat_reg()*, são carregadas as variáveis que referenciam as estatísticas a serem coletadas e por fim há o agendamento de uma auto-interrupção (*self-interrupt*) que indicará o início da colaboração.

```
1     pk_count = 0;
2     haptic_pkts = 0;
3
4     op_ima_obj_attr_get(op_id_self(), "Node Address", &src_addr);
5     op_ima_obj_attr_get(op_id_self(), "Leader", &leader);
6
7     // Gera número inteiro aleatório para identificar os pacotes de alteração da cena
8     distr = op_dist_load("uniform_int", 3, 6);
9
10    // Carregamento das estatisticas
11    ete_delay = op_stat_reg("ETE Delay", OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
12    pk_cnt_stathandle = op_stat_reg("packet count", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
13    haptic_pkt_stathandle = op_stat_reg("haptic count", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
14
15    op_intrpt_schedule_self (op_sim_time(), 0);
```

Figura 21. Executiva de Entrada do estado init

Já o estado *idle* é do tipo não-forçado (*unforced*), ou seja, após ser executada a sua Executiva de Entrada, o estado entra em um período de inatividade, até que interrupções disparadas por eventos de rede sejam retornadas, como aquelas causadas pela detecção de dados nos fluxos de entrada/saída ou pela geração de pacotes. As macros condicionais GENERATE, RECEIVED e CREATE_COLLAB estabelecem as condições necessárias para que ocorram as transições entre os estados e execução de determinadas funções. A primeira delas indica quando os pacotes gerados pelo módulo *haptic* chegam ao módulo corrente (*application)* realizando assim, uma chamada à função *send()* que é responsável pelo envio destes pacotes para a rede. A segunda macro, detecta quando os pacotes são recebidos pelo módulo receptor, ou seja, quando os pacotes são recebidos pelo nó, sendo executada logo em seguida a função *rcv()*, responsável pelo processamento dos pacotes. Por fim, a última macro detecta a auto-interrupção agendada pelo estado *init,* para que seja iniciada a aplicação colaborativa através da chamada à função *joinRequest(),* que representa uma solicitação para participação por parte de um cliente.

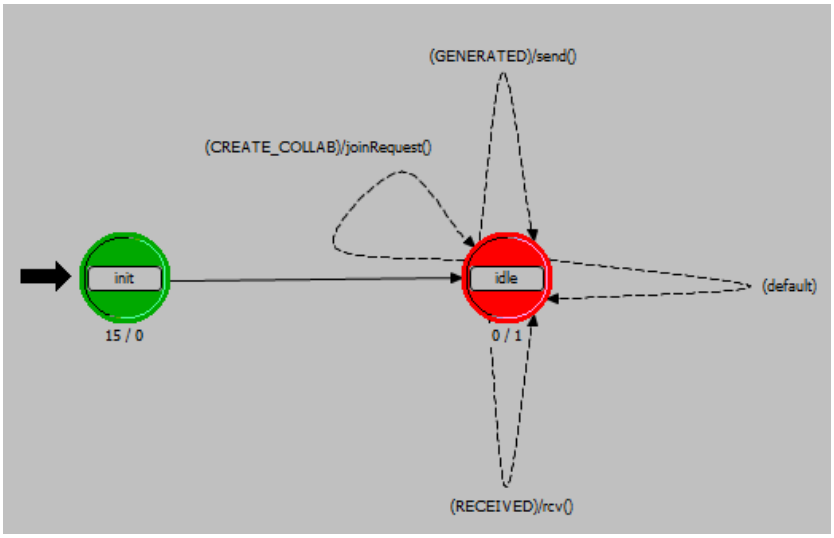

Figura 22. Diagrama de Estados do Modelo de Processo *Cybermed-process*

A função *send()* recebe os pacotes gerados pelo fluxo de pacotes GNR_STRM, entre o módulo gerador *haptic* e o módulo *application, a*tualiza a variável *haptic_pkts* que contabiliza o número de pacotes recebidos gerados pela interface háptica. Depois são gerados aleatoriamente os números inteiros, entre os valores 3 e 6, que preencherão o campo *msg_type* indicando as mensagens de alteração do ambiente virtual, como alteração da posição/rotação dos interadores ou das camadas dos objetos virtuais. Estes pacotes possuem tamanho entre 18 e 20 Bytes. Por fim, cada pacote gerado é replicado para todos os nós participantes, simulando assim o comportamento de replicação ativa do CyberMed.

```
1   void send() {
2
3   Packet *pkt_aux = 0;
4   int msg_type, i=0;
5
6   FIN(send());
7
8      // Recepção dos pacotes gerados pelo Phantom
9      pkt = op_pk_get(GNR_STRM);
10
11     // Contabilização dos pacotes recebidos e gravação de estatísticas
12     ++haptic_pkts;
13     op_stat_write(haptic_pkt_stathandle, haptic_pkts);
14
15     /* Geração de inteiros aleatórios entre 3-6, para ndicação dos tipos
16     de pacotes de alteração do AV */
17     msg_type = (int)op_dist_outcome(distr);
18
19     // Montagem dos pacotes de alteração do AV
20     op_pk_nfd_set(pkt, "msg_type", msg_type);
21     op_pk_nfd_set(pkt, "x_position", 1.0);
22     op_pk_nfd_set(pkt, "y_position", 1.0);
23     if (msg_type == 5 || msg_type == 6) {
24         op_pk_nfd_set(pkt, "layer_id", 1.0);
25     }
26
27     // Envio do pacotes para todos os participantes
28     for(i=0; i<numParticipants; i++) {
29        pkt_aux = op_pk_copy(pkt);
30        op_pk_nfd_set(pkt_aux, "dest_address", i);
31        op_pk_send(pkt_aux, OUT_STRM);
32     }
33     op_pk_destroy(pkt);
34  FOUT;
35  }
```

Figura 23. Função *send()*, responsável pela transmissão de pacotes para a rede

A função rcv() simula a recepção e o processamento dos pacotes envolvidos na aplicação. Nesta função, encontra-se a transcrição das partes mais essenciais do protocolo de colaboração do CyberMed e a forma como este trata cada um de seus pacotes. Primeiramente são recebidos os pacotes do GNR_STRM e calculadas algumas estatísticas como o atraso ponto-a-ponto global de pacotes e a quantidade total de pacotes recebidos. Para cada pacote recebido, há primeiramente a verificação do tipo de sua mensagem a partir do *flag msg_type,* para posteriormente ocorrer um tratamento específico para cada uma das mensagens possíveis. Os pacotes contendo a mensagem JOIN_REQ serão processados apenas pelo nó eleito líder da colaboração, ou seja, aquele que mediará o estabelecimento de todas as conexões. Este deve receber um pedido de entrada, coletar os dados do novo participante e emitir um pacote de aceitação (ACCEPT), para que logo em seguida, sejam enviadas ao novo participante, as informações sobre o AV de todos os participantes que já estejam na colaboração. Estes pacotes são do tipo SCENE_CONFIG e possuem as informações sobre o AV de cada nó. O processamento destes pacotes é realizado de forma simbólica uma vez que, deve ser levado em conta apenas o tráfego de rede sendo assim, os processamentos locais indiferentes.

6.2. Simulando AVCs com comunicação por Unicast

Após a execução de algumas simulações de teste, foi verificada a necessidade de aperfeiçoamento do processo de encaminhamento de pacotes (dentro dos *switches*) a fim de suportar a execução de novas simulações com maior número de usuários. Com tal objetivo, alguns estudos e exercícios foram realizados em busca de alternativas para o encaminhamento de pacotes. A solução escolhida foi o uso dos objetos de sub-redes do simulador. As sub-redes do OPNET são objetos que visam agrupar um conjunto de nós (pré-definidos ou customizados pelo desenvolvedor) e são identificadas por um identificador numérico. Para o uso adequado, um novo campo foi inserido nos pacotes utilizados a fim de identificar a sub-rede de destino. Um novo objeto (roteador) foi criado para possibilitar a interligação entre as várias sub-redes. A Figura 1 exibe a estrutura dos cenários com 100 usuários, agrupados em 10 sub-redes e interligadas pelo roteador. O roteador possui conexão com o nó *background_traffic_gen*, responsável pela geração de tráfego de fundo, a fim de emular o comportamento de uma rede não-dedicada, onde além do tráfego natural da rede, há a interferência de outros tipos de aplicação que rodam em paralelo. Este gerador foi interconectado ao roteador possibilitando-o de gerar dados e enviá-los a todas as sub-redes conectadas. O tráfego de fundo foi carregado aleatoriamente e gerado com carga de 6 a 10 % do tráfego da aplicação do *CybCollaboration*. Estes valores foram variados com o intuito de simular o comportamento variante do fluxo de dados em redes não-dedicadas, como o caso da Internet.

Os links envolvidos utilizaram os padrões Ethernet (10 Mbps), para as conexões com o roteador, e o Gigabit Ethernet (1 Gbps), para as conexões internas das sub-redes. Os cenários escolhidos procuraram simular o comportamento da aplicação até que um limite aceitável de degradação do ambiente virtual fosse extrapolado, permitindo-nos assim, reconhecer o número máximo de usuários suportado pelo protocolo em cenários semelhantes. Baseando-se nos primeiros resultados obtidos, as novas simulações envolveram 20, 30,

40, 80 e 100 usuários colaborando em tempo-real, em arquitetura unidirecional (*unidirectional tele-haptics*) e bidirecional (*bidirecional tele-haptics*).

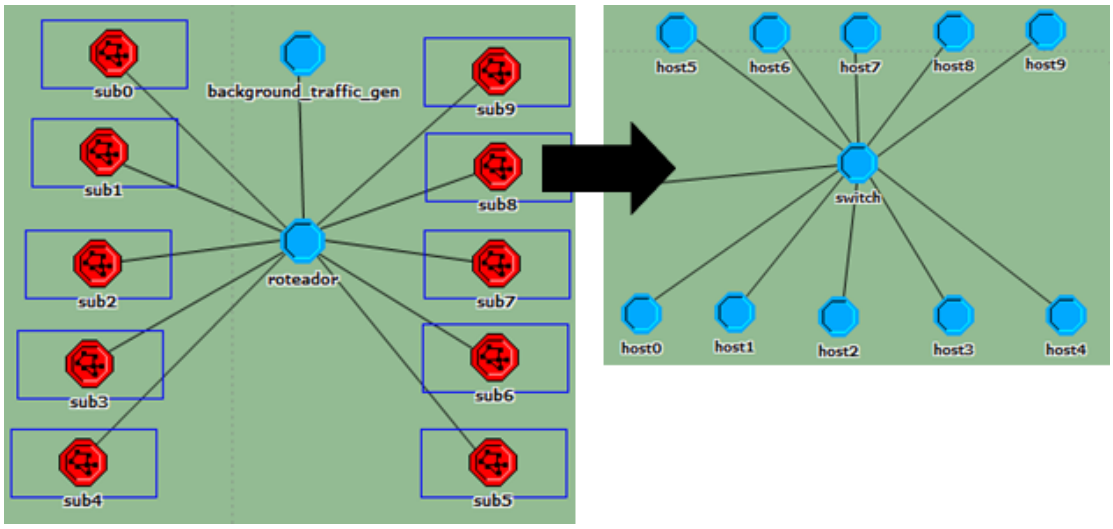

Figura 24. Topologia constituída de 10 sub-redes (cada uma com 10 usuários), 1 roteador e 1 gerador de tráfego de fundo.

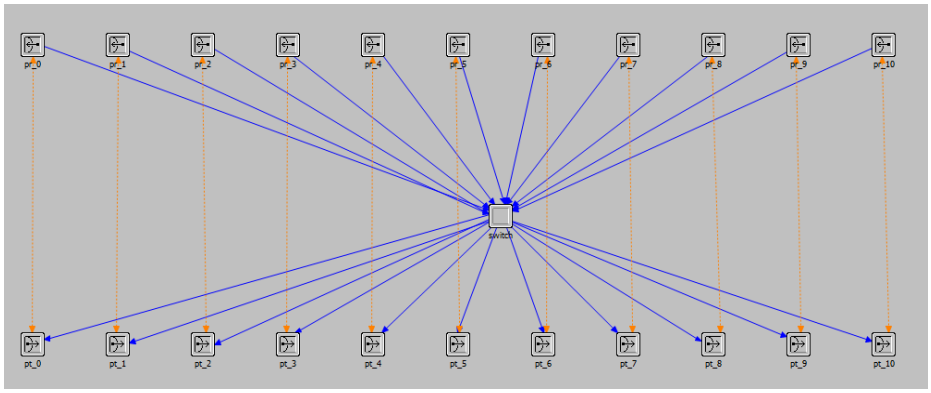

Figura 25. Modelo de roteador composto por 10 portas e 1 módulo processador.

Baseando-se em algumas referências presentes na literatura [MAR07][MAR08], tomamos como base os estudos realizados por Marshall et al. em [MAR07], para a análise de nossos resultados os requisitos QoE (*Quality of Experience*). Os requisitos necessários tomados como base de nossas avaliações estão disponíveis em Marshall et. al., 2008 (Referência I). Este trabalho estuda a transmissão de tráfego de mídia háptica ao longo de um rede de melhor esforço IP e de uma rede IP com o DiffServ habilitado. Requisitos estabelecidos:

- Delay < ~50ms
- Packet loss < 10%

De acordo com Marshall et. al., a forma como cada requisito de QoS interfere na experiência tátil do usuário pode ser percebida como os seguintes danos no AV:

- Delay - Pode reduzir o poder da força sentida pelo usuário;
- Packet loss - Afeta a percepção que o usuário tem do peso de um objeto virtual. O atraso pode também tirar a sincronização entre as diferentes cópias do ambiente virtual expalhadas na rede.

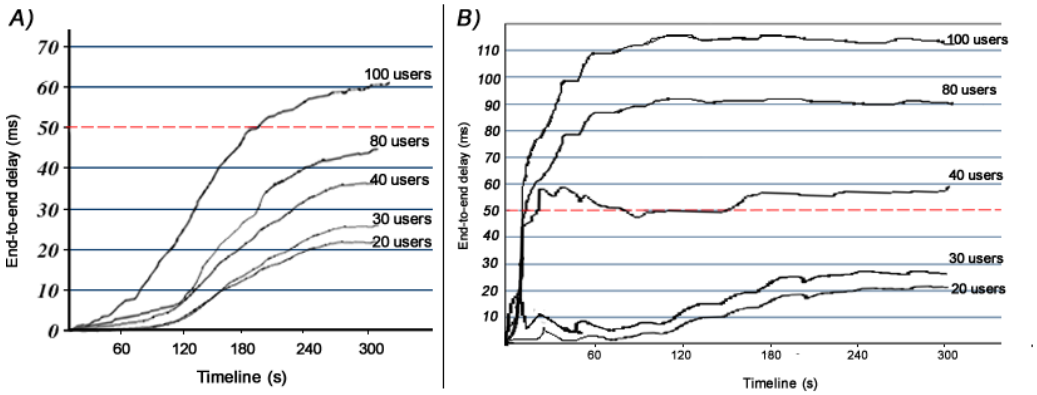

Figure 2. Perda de pacotes: (A) colaboração de tutoria; (B) colaboração concorrente

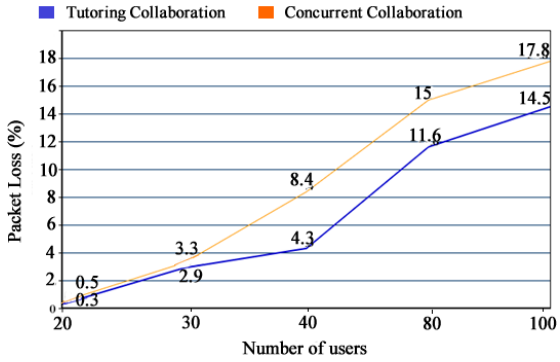

Figure 3. Perda de pacotes para os cenários de colaboração com unicast

Tabela 2. Resultados dos Cenários de Colaboração Concorrente

	Colaboração Concorrente (Fluxo bidirecional)	Colaboração de Tutoria (Fluxo unidirecional)
n° de usuários	Atraso (ms) | perda (%)	Atraso (ms) | perda (%)
20	22,5 | 0,5	21 | 0,3
30	28 | 3,3	26,5 | 2,9
40	57 | 8,4	38 | 4,3
80	90 | 15	40 | 11,6
100	115 | 17,8	60 | 14,5

Inicialmente, para os cenários que envolvem 20 e 30 usuários, o desempenho da aplicação permaneceu estável. Os valores aproximados de atraso fim a fim, para tais cenários foram: 22,5 e 28 ms. A porcentagem de pacotes descartados ou perdidos na rede apresentou-se dentro do limite aceitável (10%) com os valores

aproximados de 0,5 e 3,3 %. No cenário de 40 usuários, o atraso fim a fim alcançou uma média de aproximadamente 57 ms, o que possivelmente caracteriza o início da instabilidade na percepção tátil dos usuários, como também pode caracterizar a dessincronização entre as várias cópias do AV espalhadas entre os nós participantes. Porém, a perda de pacotes neste cenário de 40 usuários, ainda permanece em valor aceitável de 8,4 %, não afetando assim, a experiência de força sentida pelos usuários. Para os cenários de 80 e 100 usuários, os requisitos de QoE são extrapolados com valores de 90 e 115 ms de atraso de pacotes e 15 e 17,8 % de perda de pacotes, tornando a colaboração concorrente instável para cenários com número de usuários aproximado, igual ou superior a 80.

6.1.2. Resultados dos Cenários de Colaboração de Tutoria

Os resultados das colaborações de tutoria, que são aquelas que possuem fluxo de dados unidirecional, seguiram o padrão esperado: O tráfego enviado pelo nó tutor permaneceu aproximadamente (*n-1)* vezes maior do que o tráfego recebido em cada um dos nós clientes, sendo *n* o número total de participantes na colaboração. Tomando como exemplo o cenário de 20 usuários, o tráfego total enviado pelo nó tutor deve ser 19 (20 -1) vezes maior que o tráfego recebido em cada um dos nós. Foi observado, a partir dos resultados obtidos, que a colaboração de tutoria é executada sem nenhum dano na experiência dos usuários até 40 usuários participantes, sendo 39 deles estudantes e 1 tutor. Os valores aproximados obtidos para o atraso médio de pacotes foram 21, 26,5 e 38 ms e 0,3, 2,9 e 4,3 % para a perda de pacotes nos cenários com 20, 30 e 40 participantes. O desempenho da aplicação tem um dano mínimo de 11,6% (apenas 1,6% a mais do que a taxa permitida) na experiência tátil dos usuários. Porém este desempenho pode ser otimizado abaixando-se a taxa de envio do dispositivo háptico, visto que o atraso de pacotes para este cenário ainda permanece estável (40 ms). O mesmo não ocorre para o cenário com 100 usuários participantes, que alcança os valores de 60 ms de atraso e 14,5% de perda de pacotes, caracterizando degradação tanto no AV como na percepção tátil dos usuários.

Capítulo 7. Considerações Finais e Trabalhos Futuros

O presente trabalho apresentou uma documentação das atividades relacionadas ao projeto do INCT-MAC (Medicina Assistida por Computação Científica). As atividades, realizadas no laboratório LabTEVE dentro de um período de seis meses, envolveram pesquisa, desenvolvimento e simulação do *framework* CyberMed. Durante a pesquisa foi constatado que devido a pouca ou ausência de tentativas em quantificar ou qualificar os requisitos de QoS para a colaboração háptica [MAR08], esses tipos de requisitos ainda não estão bem definidos e podem mudar de acordo com experiências e estudos específicos. Como visto anteriormente, o módulo *CybCollaboration* não possuía suporte a aplicações com grande número de usuários devido o seu modelo de comunicação em rede, sendo agora possível utilizar também o protocolo multicast. Simulações experimentais foram realizadas e os resultados apresentados, para os cenários com comunicação por unicast. O modelo para a simulação multicast precisou de ajustes e atualmente encontra-se em fase de coleta dos dados para futuras

análises e comparações com o desempenho das mesmas aplicações quando utilizam o modo de transmissão por unicast. Como primeiros resultados, foi observado uma ótima melhora uma vez que o atraso de rede diminui consideravelmente quando utilizado o multicast. Atualmente, o módulo CybCollaboration também está em fase adaptação para prover suporte às colaborações com o uso de luvas de dados.

7.1. Cronograma de Atividades

As atividades realizadas seguiram o seguinte cronograma

Tabela 4. Cronograma de atividades

Atividades	VII	VIII	IX	X	XI	XII
Implementação dos modelos de simulação unicast no OM						
Estudo da linguagem C/C++						
Estudo de programação em rede e concorrente: sockets e threads						
Pesquisa sobre o multicast e implementação de protótipos						
Estudo dos módulos CybNetwork e CybCollaboration						
Modificações nos módulos de rede e colaboração						
Modificações nos módulos de rede e colaboração						
Implementação da simulação em Multicast / Escrita do relatório final						

7.1. Atividades Realizadas

Dentre o espaço de 6 meses, atividades de várias naturezas foram desenvolvidas dentro do projeto e alguns resultados foram obtidos, tais como:

- Reuniões mensais no LabTEVE;

- Trabalho aceito para publicação:
    - Paiva, Paulo V. F. ; Machado, Liliane S. ; Oliveira, Jauvane C. . An Experimental Study on CHVE's Performance Evaluation. Studies in Health Technology and Informatics, 2012.
- Apresentação interna de trabalhos no LabTEVE, apresentado os seguintes trabalhos:
    - “Programação paralela com CUDA“

8. Referências

[LAB10] LabTEVE - Site Oficial do Laboratório de Tecnologias para o Ensino Virtual e Estatística. *Página Principal.* João Pessoa - PB, 2010. Disponível em: < http://www.de.ufpb.br/~labteve/>. Acessado em novembro de 2010.

[MAR99] D. Margery et al. (1999), “A General Framework for Cooperative Manipulation in Virtual Environments”, Springer, Vol..44, No. 7, pp. 79-85.

[PIN02] M. Pinho et al. (2002), “Cooperative Object Manipulation in Immersive Virtual Environments: Framework and Techniques”, *Proceedings of the ACM VRST’02,* Hong Kong*,* pp. 171-178.

[RUD02] R. Ruddle et al. (2002) “Symmetric and Asymetric Action Integration During Cooperative Object Manipulation in Virtual Environments”, *ACM Transactions on Computer-Human Interaction,* Hong Kong*,* pp. 285-308.

[BRO95] W. Broll et al. (1995), “Interacting in Distributed Collaborative Virtual Environments”, *IEEE VRAI'95 - Virtual Reality Annual International Symposium*, IEEE Computer Society Press, pp.148-155.

[SOU03] Souayed, R. et al. (2003), “Haptic Virtual Environment Performance Over IP Networks: A case study”, *Processings of the Seventh IEEE DS-RT’03*.

[ZYD99] Singhal, S., Zyda, M., (1999), “Networked Virtual Environments: Design and Implementation”, *ACM Press/Addison-Wesley Publishing Co*, New York, USA.

[SKE06] Duval, T. et al. (2006), “SkeweR: a 3D Interaction Technique for 2-User Collaborative Manipulation of Objects in Virtual Environments”, *Proceedings of 3DUI’06,* Virgínia, USA*,* pp. 69-72.

[PAI10] Paiva, V. P., Machado, S. L. (2010), “Um Estudo sobre Manipulação Cooperativa em Ambientes Virtuais Colaborativos”, WRVA’10, São Paulo.

[SEN10] Site Oficial da *Sensable Technologies.* Disponível em: < http://www.sensable.com/>. Acessado em novembro de 2010.

[MAC08] Machado, L. S. et. al. (2008), “Desenvolvimento Rápido de Aplicações de Realidade Virtual Utilizando Software Livre”, *Livro de Minicursos do SVR’08*, pp. 5-33.

[SUN06] Sung, M. Y. et al. (2006), “Experiments for a Collaboratiove Haptic Virtual Reality”, *Proceedings of IEEE ICAT’06*, Hangzhou, pp. 174-179.

[BAS00] Basdogan. C. et al. (2000) , “An Experimental Study on the Role of Touch in Shared Virtual Environments”. *ACM Transactions on Computer-Human Interaction, Vol. 7,* pp. 443-460.

[MEN10] Mendes, L. L. “OPNET: Tutorial Básico”. Disponível em: < http://www.opnet.com/>. Acessado em novembro de 2010.

[OPN10] Site Oficial da OPNET Technologies. *Página Principal.* 2010. Disponível em: < http://www.opnet.com.> Acessado em novembro de 2010.

[SAL10] Sales, B. R. A. (2010), “Colaboração em Sistemas de Realidade Virtual voltados ao Treinamento Médico: um Módulo para o Framework CyberMed”, *Dissertação de Mestrado*, UFPB, João Pessoa – JP.

[KAM02] Carlos Kamienski et al. (2002), “Simulando a Internet: Aplicações na Pesquisa e no Ensino.” *Anais do XXII Congresso da SBC (JAI).* :, v. 2, p. 33-86.

[MAE07] Maestrelli, M. (2007), “Dificuldades em Simular a Internet”. Disponível em <http://cbpfindex.cbpf.br/publication_pdfs/nt00307.2008_01_04_14_25_50.pdf>. Acessado em novembro de 2010.

[MOR09] Moreira,V. (2009), “Avaliação do desempenho da comunicação de dados baseada na IEC 61850 aplicada a refinarias de petróleo”. Disponível em: <http://www2.ee.ufpe.br/instrumentacao/monografias/Vinicius_Moreira_PROMINP_II.pdf>. Acessado em novembro de 2010.

[POR10] Portnoi, M., Araújo, R. G. B., Brito, S. F. (2010) “Network Simulator – Visão Geral da Ferramenta de Simulação de Redes”

[ERI04] Eric Freeman et al. (2004) “Head First Design Patterns”, O’Reilly Media.

[TAN03] A. S. Tanenbaum, “Redes de Computadores”. Editora Campus, 4 rd ed., 2003

[DEV02] P. Dev, D. Harris, D. Gutierrez, A. Shah, and S. Senger, “End-to-end performance measurement of internet based medical applications,” in Proceedings of the Annual Symposium of the American Medical Informatics Association, pp. 205–209, San Antonio, Tex, USA, November 2002.

[MAR07] K. M. Yap, A. Marshall, W. Yu, G. Dodds, Q. Gu, and R. T. Souayed, “Characterising distributed haptic virtual environment network traffic flows,” in Proceedings of the4th IFIP Conference on Network Control and Engineering for QoS, Security and Mobility, pp. 297–310, Lannion, France, November 2007.

[SOU04] R. T. Souayed, D. Gaiti, W. Yu, G. Dodds, and A. Marshall, “Experimental study of haptic interaction in distributed virtual environments,” in Proceedings of Eurohapics, pp. 260–266, Springer, Munich, Germany, June 2004.

[MAR08] Alan Marshall, Kian Meng Yap, and Wai Yu, “Providing QoS for networked peers in distributed haptic virtual environments,” *Advances in Multimedia*, vol. 2008, Article ID 841590, 14 pages, 2008.

[GAM05] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (2005). “Padrões de Projeto: Soluções reutilizáveis de software OO”. Porto Alegre: *Bookman*.

[MAC08] Machado, L.S.; Souza, D.F.L.; Souza, Leandro C. ; Moraes, Ronei M. (2008) “Desenvolvimento Rápido de Aplicações de Realidade Virtual e Aumentada Utilizando Software Livre.” In: Veronica

Teichrieb; Fátima Nunes; Liliane Machado; Romero Tori. (Org.) *Realidade Virtual e Aumentada na Prática*. Porto Alegre: SBC, p. 5-33.

[SAL10] Sales, B.R.A.; Machado, L.S.; Moraes, R.M. (2010) “Colaboração Interativa para Sistemas de Realidade Virtual voltados ao Ensino e Treinamento Médico.” In: *Congresso Brasileiro de Informática em Saúde (CBIS 2010)*, Porto de Galinhas, Brazil. CDROM.

[CEL04] Waldemar Celes, Renato Cerqueira, Jose Lucas Rangel (2004) “Introdução a Estrutura de Dados - Renato Cerqueira“.

[POS11] Posix Thread Programming: <https://computing.llnl.gov/tutorials/pthreads/> Acessado em Dezembro de 2011

[CAR06] Richard H. Carver and Kuo-Chung Tai, (2006), “Modern Multithreading”, Wiley.

[HAL01] Brian “Beej” Hall (2001) “Beej’s Guide to Network Programming: Using Internet Sockets”

[KEV03] Makofske David and Almeroth Kevin (2003), “Multicast Sockets: Practical Guide for Programmers”, Practical Guide Series

ANEXO

(Paper submetido ao MMVR)