PUC-Rio - Certificação Digital Nº 0410842/CA

2
Técnicas usando TCP,UDP e Sockets

A API do Sockets permite que aplicações realizem comunicação através de uma rede de computadores. Ela possibilita o uso dos protocolos de comunicação UDP e TCP para transferência de dados. WinSock é uma API específica para Windows que implementa as funcionalidades da API do Sockets porém com algumas extensões. Para mais informações sobre a API do Sockets e do WinSock veja [28, 9].

Este capítulo trata de técnicas eficientes ao usar TCP, UDP e a API do Sockets. Uma proposta de implementação de garantia de chegada sobre UDP também é apresentada.

2.1
Quando usar TCP e UDP

TCP é um protocolo que permite o envio de um fluxo de bytes de um ponto a outro de forma garantida e ordenada. Já UDP é um protocolo de envio de mensagens de forma não garantida e não ordenada. A falta de entendimento do funcionamento interno do TCP pode acarretar o seu uso de forma inadequada e causar, com isso, a impressão que TCP é um protocolo ineficiente.

É importante relatar que a diferença no tempo de chegada de uma mensagem enviada em TCP e uma enviada em UDP é pequena. O tempo a mais que TCP irá levar é influência do processamento local maior das mensagens. Outros fatores, explicados posteriormente, podem inserir um atraso maior no envio de uma mensagem TCP. Nesse caso, é assumido um momento em que nenhum desses fatores atrasa o envio.

Em trocas de mensagens não-urgentes, onde há necessidade de ordenação e garantia de chegada (por exemplo, no *login* de um MMORPG), TCP pode ser uma boa solução. No entanto, para movimentação em jogos multijogadores em massa, TCP normalmente não se torna uma boa solução. Os exemplos a seguir ilustram a situação:

- Exemplo 1 : Suponha-se que existam 2 outros jogadores no campo de visão de um cliente. Suponha-se que estes 2 jogadores tenham feito uma movimentação e o servidor tenha recebido a mensagem de movimentação dos 2 jogadores. Se o servidor usar TCP e enviar a posição do jogador 1 e depois a posição do jogador 2, o cliente recebendo as posições só informará à aplicação os bytes da posição do jogador 2 depois que receber os bytes da posição do jogador 1; mesmo que o cliente receba os bytes da posição do jogador 2 antes dos bytes da posição do jogador 1. Com isso, se um cliente receber logo a informação do jogador 2 e demorar para receber a informação do jogador 1, serão 2 jogadores muito dessincronizados com o servidor. Se a posição do jogador 2 fosse logo informada para a aplicação, apenas o jogador 1 estaria muito dessincronizado.
- Exemplo 2 : Suponha-se que um servidor acabou de receber uma mensagem de movimentação de um jogador e ele acabou de enviá-la para um determinado cliente. Antes do servidor receber o Ack por essa mensagem, uma nova mensagem de movimentação desse mesmo jogador é recebida e ele tem que enviar a nova mensagem. A informação de movimentação antiga do jogador é obsoleta; no entanto, TCP só fornecerá os bytes da mensagem nova para a aplicação cliente depois que fornecer os bytes da mensagem antiga. Nesse caso, o melhor seria que fosse possível descartar a mensagem obsoleta do conjunto de mensagens com garantia de chegada.

PUC-Rio - Certificação Digital Nº 0410842/CA

No exemplo 1, se faz necessário um protocolo de comunicação que forneça garantia de chegada às mensagens, mas sem ordenação. No exemplo 2, se faz necessário um protocolo de comunicação que forneça garantia de chegada apenas à mensagens mais recentes de uma determinada informação. Esses protocolos não correspondem nem a TCP nem a UDP. Logo eles precisam ser implementados. Nessa dissertação, esses protocolos são implementados em cima de UDP. Em seções posteriores, detalhes da implementação desses protocolos são explicados.

2.2 O Algoritmo de Naggle e o Ack Atrasado

Se num determinado jogo for verificado que há a possibilidade do uso de TCP, é importante o entendimento de 2 partes do TCP, o algoritmo de Naggle e a técnica do Ack atrasado [27].

O algoritmo de Naggle consiste em não enviar uma mensagem, se ela for pequena e houver alguma mensagem enviada que não recebeu Ack. Uma mensagem é pequena se ela é menor que o tamanho de um segmento máximo que pode ser enviado ao outro ponto. Suponha-se que um servidor quer enviar 2 posições de jogadores para um cliente. Neste caso, o servidor envia a primeira posição com um send() e depois a segunda posição com outro send(). A segunda posição só será enviada depois que o servidor receber um Ack pela primeira posição, o que pode demorar muitos milisegundos numa WAN. O ideal, nesse caso, é que o algoritmo de Naggle fosse desligado e que a própria aplicação agregasse as 2 posições numa só mensagem, reduzindo o número de bytes usados em cabeçalhos (20 bytes para TCP e 20 bytes para IP).

Para desligar o algoritmo de Naggle em WinSock deve-se usar:

```
BOOL b = TRUE;

setsockopt(Socket,IPPROTO_TCP,TCP_NODELAY,(char*)&b,
  sizeof(BOOL));
```

Outro conceito importante sobre TCP é o Ack atrasado(*Delayed Ack*). Toda mensagem recebida por TCP não é imediatamente avisada como recebida (i.e. respondendo com um Ack). Ao invés disso, TCP espera normalmente em torno de 200 milisegundos na esperança que apareça alguma mensagem na direção do Ack de modo que o Ack possa ser enviado junto à mensagem (*Piggyback*).

Sendo assim, no exemplo dado, o tempo que demoraria para a segunda mensagem ser enviada seria todo o tempo de ida e volta de uma mensagem entre o cliente e o servidor (*Round Trip Time*) + 200 milisegundos, caso o cliente não tivesse nenhuma mensagem para enviar para o servidor.

É importante considerar essas 2 partes do TCP, pois falhar em fazer isso pode gerar uma comunicação ineficiente.

2.3
Reduzindo cópias de buffer em WinSock

Em WinSock, é possível fazer envio e recebimento de dados de maneira assíncrona, usando operações *overlapped I/O*. As funções que permitem essas operações, quando em modo *overlapped*, retornam imediatamente e, em algum momento posterior, recebe-se o evento de término da operação.

PUC-Rio - Certificação Digital Nº 0410842/CA

Para fazer uso dessas operações é preciso se criar um socket com o flag WSA_FLAG_OVERLAPPED habilitado. Para isso, usa-se:

```
Socket = WSASocket(AF_INET,SOCK_STREAM,0,NULL,0,
  WSA_OVERLAPPED)
```

pode-se usar também:

```
Socket = socket(AF_INET,SOCK_STREAM,0);
```

Usando a função socket(), o flag WSA_FLAG_OVERLAPPED é habilitado automaticamente. Depois da criação do socket, usam-se funções, tais como WSASend, WSASendTo, WSARecv e WSARecvFrom, que recebem como um dos parâmetros, uma variável do tipo WSAOVERLAPPED zerada. Como por exemplo:

```
WSAOVERLAPPED sndOp;

ZeroMemory(&sndOp,sizeof(sndOp));

WSASend(Socket,DataBuff,1,&BytesSent,Flags,&sndOp,NULL);
```

Uma vantagem do uso de operações overlapped está na possibilidade de se fazer com que o WinSock use os buffers da aplicação diretamente, em vez de usar buffers próprios para operações I/O [15]. Isso remove transferências de memória entre o WinSock e a aplicação nas chamadas de I/O.

Para fazer o WinSock usar os buffers da aplicação, basta zerar os tamanhos dos buffers de envio e de recebimento do socket:

```
int Zero = 0;

setsockopt(Socket,SOL_SOCKET,SO_SNDBUF,(char*)&Zero,
  sizeof(Zero));

setsockopt(Socket,SOL_SOCKET,SO_RCVBUF,(char*)&Zero,
  sizeof(Zero));
```

No presente trabalho, foi realizado um teste[1] para comparar o envio de mensagens UDP de forma convencional (sendto() bloqueante) e o envio

[1]Todos os testes realizados neste trabalho levantam dados através da execução de aplicações em Windows. É possivel que processos do próprio sistema operacional causem variações nos gráficos.

PUC-Rio - Certificação Digital Nº 0410842/CA

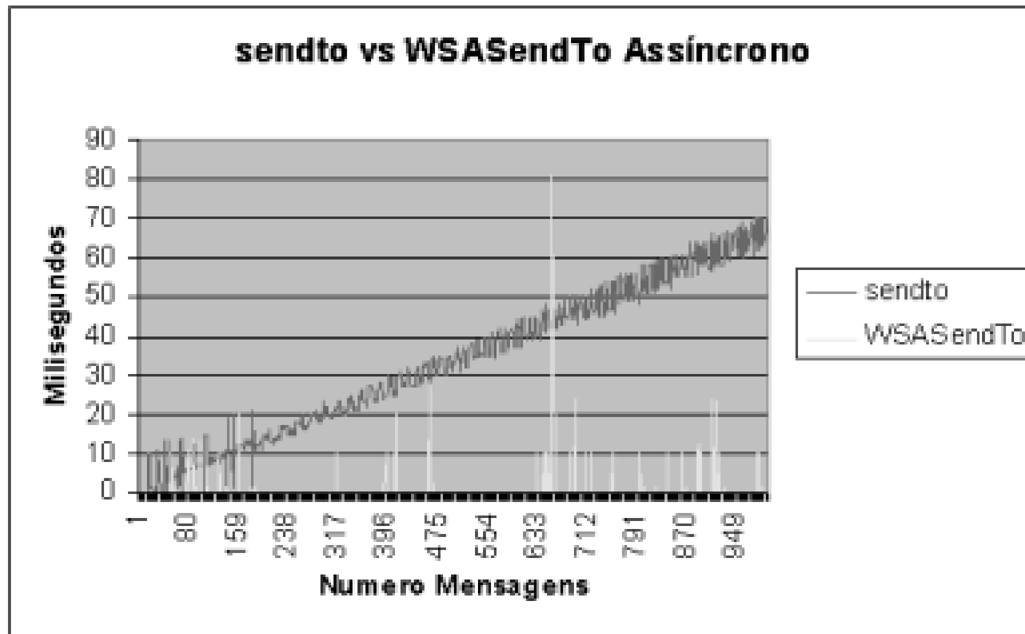

Figura 2.1: Comparação de sendto bloqueante com WSASendTo assíncrono usando buffer da aplicação

PUC-Rio - Certificação Digital Nº 0410842/CA

de mensagens UDP de forma assíncrona (usando *IO Completion Ports*), com o SO_SNDBUF colocado em 0 (Fig. 2.1). Na figura 2.2 é mostrado o mesmo teste só que SO_SNDBUF não é colocado em 0, ou seja, o buffer do WinSock é usado.

A principal conclusão que se pode tirar destes testes é que o envio assíncrono usando *IO Completion Ports* é consideravelmente mais eficiente do que o envio síncrono bloqueante. Pelos dois testes também pode-se ver que o uso de buffers da aplicação fornece um ganho em eficiência. No teste usando o buffer da aplicação, WSASendTo é em média 892% mais rápido enquanto no teste usando o buffer do WinSock, WSASendTo é em média 825% mais rápido.

2.4 Usando sockets UDP conectados

Conectar um socket UDP traz algumas vantagens. Ao chamar connect() em um socket UDP nenhuma mensagem é enviada, apenas o endereço e a porta remotos são gravados no socket. Com isso é possível passar a usar apenas send() em vez de sendto().

Em alguns sistemas operacionais (ex: Implementações do BSD como o SunOS 4.x), chamar sendto() faz o kernel conectar o socket usado, enviar a mensagem e desconectar o socket a seguir. Em [21] é verificado que

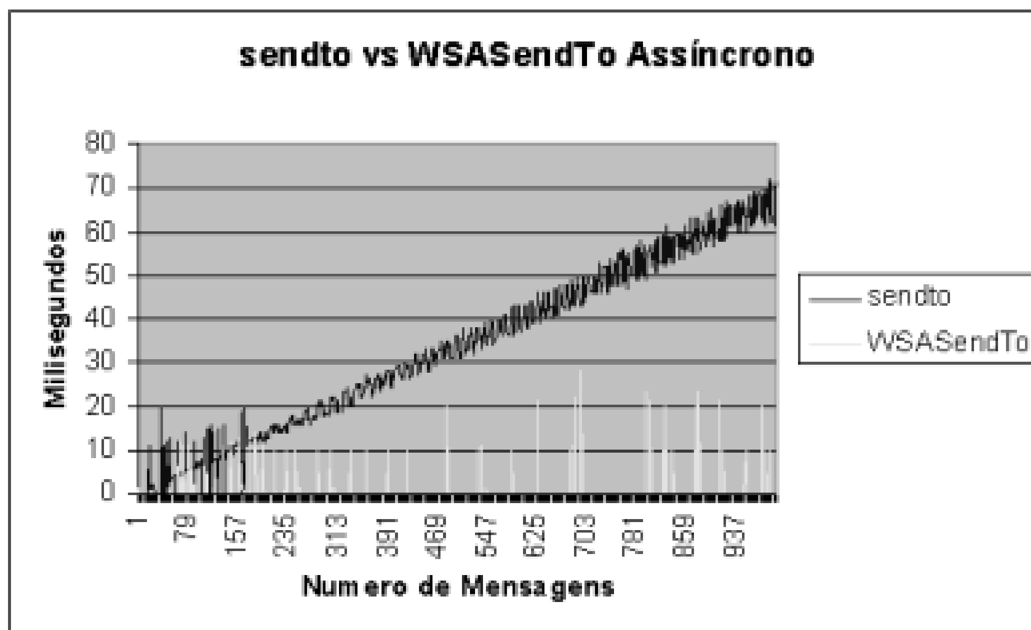

Figura 2.2: Comparação de sendto bloqueante com WSASendTo assíncrono usando buffer do WinSock

PUC-Rio - Certificação Digital Nº 0410842/CA

conectar e desconectar o socket dessa maneira custa quase 1/3 o tempo de processamento do envio de uma mensagem.

Além dessa melhoria de desempenho, conectar um socket UDP permite receber erros assíncronos de mensagens enviadas. Se, por exemplo, uma mensagem UDP for enviada para um determinado endereço, e nenhuma aplicação estiver recebendo mensagens na porta de destino, uma mensagem ICMP port-unreachable(porta inalcançável) é retornada. A única forma de receber essa mensagem é tendo um socket UDP conectado.

No presente trabalho, foram feitos testes comparando o envio de sockets UDP normais (usando sendto) e sockets UDP conectados (usando send). A figura 2.3 mostra os resultados do teste. Os testes mostram que ambos os casos conseguiram valores bastante próximos. O envio de mensagens usando sockets UDP conectados foi em média 4.71 % mais rápido. Em alguns testes de envio de 1000 mensagens, os sockets UDP conectados foram entre 12 a 22% mais rápidos que o envio usando sockets não conectados.

2.5 Enviando múltiplos buffers em Sockets

Em diversos momentos se faz necessária a criação de uma mensagem com cabeçalho e corpo, como por exemplo:

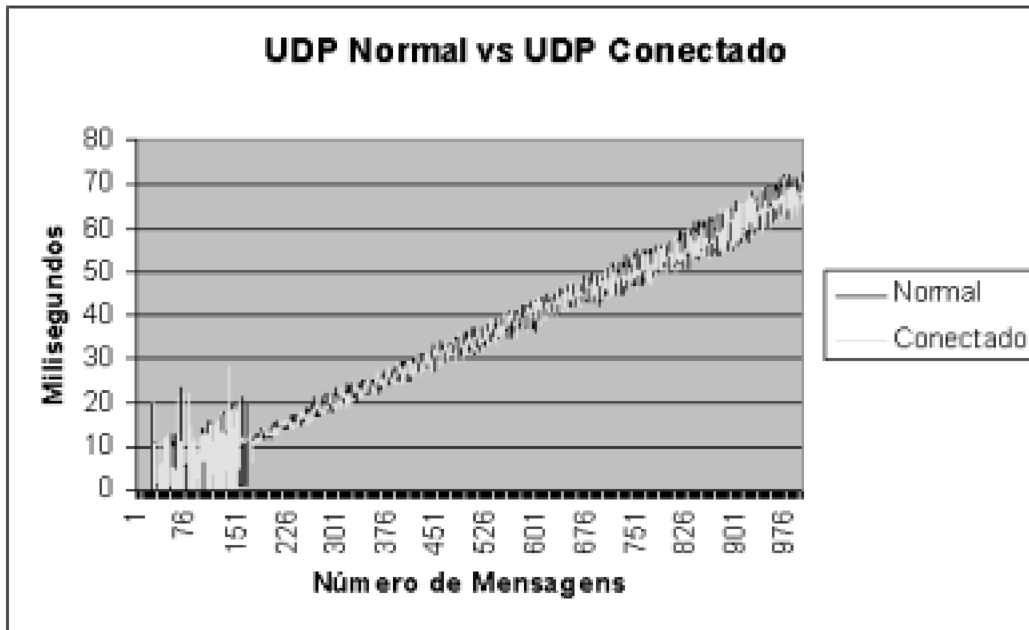

Figura 2.3: Comparação de envio de mensagens usando sockets UDP normais com sendto() e conectados com send()

PUC-Rio - Certificação Digital Nº 0410842/CA

```
struct MsgPos
{
  int msgID;
  float x,y,z;
};
```

No exemplo acima, o cabeçalho é msgID, que identifica o tipo da mensagem, e o corpo é x, y, z, que definem uma posição no espaço 3D. Uma forma de enviar uma mensagem desse tipo é a seguinte:

```
MsgPos Msg;
...
send(Socket,(const char*)&Msg,sizeof(Msg),0);
```

Quando deseja-se enviar uma mensagem cujo corpo é variável (uma mensagem de chat por exemplo, onde o cabeçalho identifica a mensagem e informa o seu tamanho e o corpo é o texto da mensagem) pode-se fazer uso da seguinte implementação:

```
char Msg[MSG_SIZE];
Cabecalho c;
c.msgID = MSG_CHAT;
c.size = strlen(MsgChat); //MsgChat é a mensagem de chat
```

```
memcpy(Msg,&c,sizeof(c));
memcpy(Msg+sizeof(c),MsgChat,c.size);

send(Socket,Msg,sizeof(c)+c.size,0);
```

Nesse caso, a mensagem é construída em um buffer da aplicação para depois ser copiada para o buffer de envio do socket. Um forma mais eficiente seria usar as funções para cópias de múltiplos buffers, de maneira a fazer a construção da mensagem no próprio buffer de envio do socket. Em WinSock, isso é feito da seguinte maneira:

```
Cabecalho c;
c.msgID = MSG_CHAT;
c.size = strlen(MsgChat);

WSABUF MsgBuff[2];
MsgBuff[0].buf = &c;
MsgBuff[0].len = sizeof(c);

MsgBuff[1].buf = MsgChat;
MsgBuff[1].len = c.size;

WSASend(Socket,MsgBuff,2,&BytesSent,Flags,NULL,NULL);
```

No caso acima, usa-se WSABUF para construir a mensagem dentro do buffer de envio do WinSock. WSABUF possui o seguinte formato:

```
typedef struct _WSABUF {
  u_long      len;
  char FAR *  buf;
} WSABUF, FAR * LPWSABUF;
```

onde len é o tamanho do buffer e buf é o ponteiro para o buffer.

A função WSASend possui o seguinte formato:

```
int WSAAPI WSASend(SOCKET s, LPWSABUF buf, DWORD cnt,
  LPDWORD sent, DWORD flags, LPWSAOVERLAPPED ovl,
  LPWSAOVERLAPPED_COMPLETION_ROUTINE func);
```

PUC-Rio - Certificação Digital Nº 0410842/CA

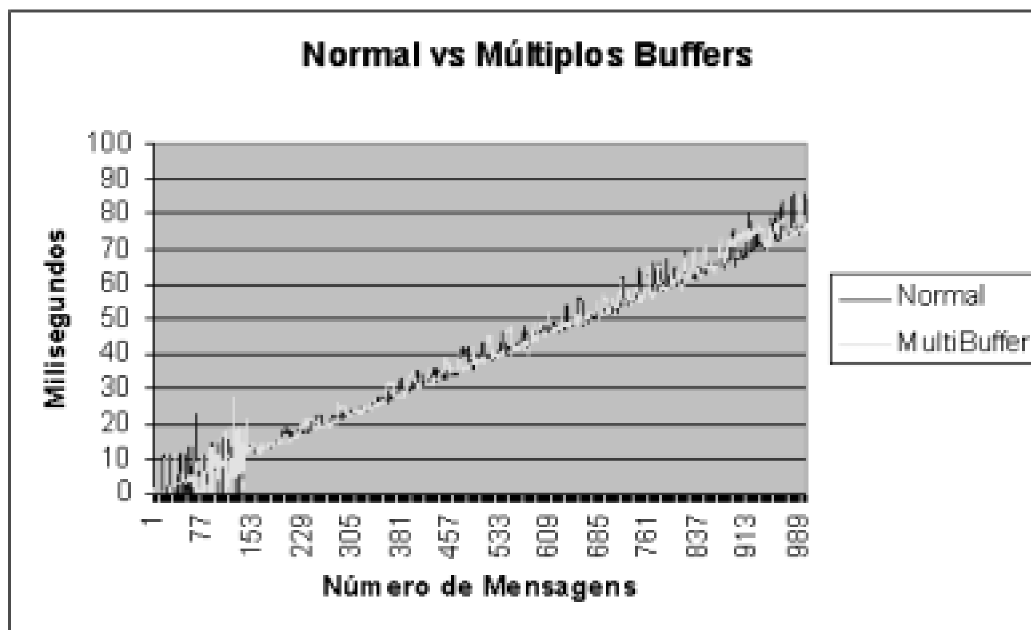

Figura 2.4: Comparação de envio de mensagens usando sendto() e WSASendTo() usando 2 buffers

PUC-Rio - Certificação Digital Nº 0410842/CA

onde s é o descritor do socket, buf é um vetor de WSABUF, cnt é o número de elementos no vetor, sent é o número de bytes enviados, flags é um parâmetro similar ao flags usado para o *send* convencional, *ovl* e *func* são usados para operações overlapped. Se a função termina imediatamente, ela retorna 0, caso contrário ela retorna SOCKET_ERROR.

Unix possui funções similares como:

```
ssize_t writev(int fd, const struct iovec * iov, int cnt);
```

onde fd é o descritor do socket, iovec é um vetor de buffers para envio e cnt é o numero de buffers. A estrutura iovec possui o seguinte formato:

```
struct iovec {
  char * iov_base; //endereço do buffer
  size_t iov_len;  //tamanho do buffer
};
```

A figura 2.4 mostra um teste do envio de mensagens UDP, com cabeçalho e corpo dinâmico, comparando o uso de sendto() com WSASendTo() usando 2 buffers.

Examinando a figura 2.4 pode-se ver que as duas formas de envio estão equilibradas em seus valores. O envio usando múltiplos buffers foi em média 9.47 % mais rápido. Em testes realizados do envio de 1000 mensagens, o envio usando múltiplos buffers foi entre 12 a 19% mais rápido.

2.6 Implementando Garantia de chegada em UDP

Em muitos casos, um protocolo que garanta a chegada das mensagens enviadas, sem obrigação de ordenação, se faz necessário. É o caso da lógica de movimentação mencionada na seção 2.1. A implementação aqui descrita é desenvolvida sobre UDP.

A maneira usual de se implementar garantia de chegada de mensagens é usando temporizadores e retransmissão. Quando uma mensagem UDP é enviada, um temporizador é acionado. Passado um tempo limite, o temporizador expira e a mensagem UDP é reenviada. Quando o outro ponto recebe a mensagem, ele responde com um Ack, informando ao ponto de envio sobre o recebimento da mensagem. Quando um ponto recebe um Ack de uma mensagem enviada, ele remove a mensagem de sua lista de retransmissão.

Um problema importante na implementação desse protocolo é a escolha do tempo para retransmissão de uma mensagem. Neste trabalho, esse tempo é denominado de RTO(*Retransmission Timeout*). Para uma boa escolha de um RTO é importante se levar em consideração o tempo estimado que uma mensagem leva para ir e voltar (que neste trabalho é chamado de SRTT) e a variação estimada entre os tempos de ida e volta das mensagens (que neste trabalho é chamado de VARRTT).

Para calcular o SRTT pode-se usar:

```
delta = RTT - SRTT
SRTT = SRTT + (delta / RTTINF)
```

RTT(round trip time) é o tempo efetivo que demora para uma mensagem ir e voltar entre 2 pontos. No código demonstrado, é calculada a diferença entre o valor estimado de RTT(SRTT) e o RTT de fato. Após esse cálculo, o SRTT aumenta ou diminui uma fração dessa diferença dependendo se o RTT for maior ou menor que o SRTT. RTTINF é uma variável que controla o quanto SRTT é influenciado por um RTT diferente.

Para calcular o VARRTT pode-se usar:

```
VARRTT = VARRTT + ((| delta | - VARRTT) / VARRTTINF)
```

A variação estimada dos tempo de ida e volta de mensagens(VARRTT) é influenciada por uma fração da diferença entre a variação calculada(delta) e a variação estimada(VARRTT). VARRTTINF é uma variável que controla o quanto VARRTT é influenciada por uma variação diferente.

PUC-Rio - Certificação Digital Nº 0410842/CA

Para as variáveis RTTINF e VARRTTINF são sugeridos os valores 8 e 4 [28]. O motivo principal é que ambos os valores são múltiplos de 2, o que torna possível se substituírem as divisões por shifts de bits, como por exemplo:

```
SRTT = SRTT + (delta >> 3); //dividir delta por 8

VARRTT = VARRTT + ((| delta | - VARRTT) >> 2); //dividir
  a diferença por 4
```

Para calcular o RTO, pode-se usar então:

```
RTO = SRTT + (VARRTT << VARINC)
```

O tempo para retransmissão de uma mensagem é calculado pela soma do tempo estimado de RTT com a variação estimada do RTT. Essa variação é influenciada por VARINC que multiplica o valor da variação. Um valor sugerido para VARINC é 2 , multiplicando VARRTT por 4 [28].

Outro fator importante é a variação do RTO a cada retransmissão. Segundo [8], a cada retransmissão, o RTO deve aumentar exponencialmente. Se um RTO é 2 segundos, e é feita uma retransmissão, o próximo RTO é de 4 segundos. Se outra retransmissão novamente for feita, o próximo RTO vai para 8 segundos. Ou seja, a cada retransmissão:

```
RTO = RTO << 1;
```

A retransmissão de mensagens pode ser implementada usando uma *thread*, que fica suspensa até o tempo de retransmissão mais próximo. Para facilitar a aquisição do momento de retransmissão mais próximo, as mensagens podem ser armazenadas de forma ordenada por RTO.

O envio de mensagens pode ser implementado da seguinte forma em pseudo-código:

```
EnviarMensagem(Msg)
Início
  EnviarMensagemUDP(Msg)
  Calcular o RTO da mensagem (usando RTO = SRTT +
    (VARRTT >> VARINC))
  Adicionar a mensagem à fila de retransmissão de
    maneira ordenada crescente
  Ativar o evento AtivaThread que faz thread de
```

PUC-Rio - Certificação Digital Nº 0410842/CA

```
    retransmissão recalcular o tempo suspenso
    baseado na primeira mensagem
Fim
```

A última parte do EnviarMensagem é criada porque é possível que o RTO da mensagem enviada agora possa ser o menor RTO da lista de retransmissão. Isso pode ocorrer num servidor em que um cliente possua o SRTT menor que os outros, o que contribui para um RTO menor.

A thread de retransmissão pode ser implementada da seguinte forma em pseudo-código:

```
ThreadRetransmissao()
Início
  Loop Infinito
    Se a lista está vazia
      TempoSuspenso = INFINITO
    Senão
      TempoSuspenso = Pegar o menor tempo da lista
        de retransmissão(é o da primeira mensagem)
        - TempoAtual

    Retorno = EsperarEvento(AtivaThread,TempoSuspenso)

    Se Retorno == tempo suspenso já passado
      EnviarMensagemUDP(Msg)
      Recalcular o RTO da mensagem (usando RTO = RTO >> 1)
      Reposicionar a primeira mensagem na lista de
        retransmissão
    Senão
      O retorno foi pela ativação do evento AtivaThread,
        desativar o evento
  Fim Loop
Fim
```

A função EsperarEvento é uma função que só retorna quando o evento AtivaThread é ativado ou quando já se esperou TempoSuspenso em milisegundos. Toda vez que a lista de retransmissões é alterada, durante o envio e recebimento de uma mensagem, é ativado o evento AtivaThread, fazendo com que a thread recalcule o seu TempoSuspenso.

O recebimento de mensagens pode se dar da seguinte forma em pseudo-código:

PUC-Rio - Certificação Digital Nº 0410842/CA

```
ReceberMensagem(Msg)
Início
  Se a mensagem for uma mensagem enviada de chegada garantida
    Enviar para o endereço da mensagem um Ack
    Repassar a mensagem para a aplicação
  Se a mensagem for um Ack
    Calcular o RTT da mensagem
    Calcular o SRTT e o VARRTT do ponto que retornou o Ack
    Remover a mensagem do Ack da lista de retransmissão
Fim
```

Implementar a garantia de chegada em mensagens UDP fornece uma ferramenta que pode ser aplicada em muitas situações. No caso já mencionado da movimentação, essa ferramenta se faz necessária para tratar de uma parte do protocolo. Mais informações sobre a implementação da garantia de chegada usando mensagens UDP podem ser encontradas em [28].

PUC-Rio - Certificação Digital Nº 0410842/CA