3
Técnicas para processamento de Sockets em servidores

As técnicas explicadas neste capítulo tratam de formas eficientes para servidores processarem um grande número de sockets conectados. Os casos específicos de Windows e Unix são considerados.

3.1
Processando múltiplos clientes ao mesmo tempo

Uma maneira para se processar múltiplos clientes ao mesmo tempo é usar sockets bloqueantes com múltiplas threads. Nesse caso, em geral é usada uma thread por cliente, tratando de suas requisições e respostas. Por exemplo:

```
clientSocket = accept(listen,(sockaddr*)&clientAddr,
  sizeof(sockadd_in));
_beginthread(((void(*)(void*))ClientThread,STACK_SIZE,
  (void*)clientSocket);
...
void ClientThread(SOCKET fd)
{
  ...
  r = recv(fd,buffer,sizeof(buffer),0);
  send(fd,buffer,r,0);
}
```

Usando essa maneira em servidores multijogador em massa, pode-se chegar a um número elevado de threads criadas num mesmo servidor, o que traz um elevado gasto de processamento na troca de contexto das threads.

Outra maneira de se processarem múltiplos clientes é usando select [28]. select é uma função que bloqueia por um tempo controlado (podendo ser infinitamente), enquanto alguns eventos não ocorrem. Esta função

PUC-Rio - Certificação Digital Nº 0410842/CA

retorna o número de sockets a serem tratados quando qualquer um dos seguintes eventos ocorrerem:

- Um ou mais sockets estão prontos para leitura
- Um ou mais sockets estão prontos para escrita
- Um ou mais sockets possuem uma exceção pendente

Abaixo está um exemplo que verifica se vários sockets estão prontos para leitura:

```
while(true)
{
  memcpy(&readSet,&initReadSet,sizeof(initReadSet));

  select(maxfdp1,readSet,NULL,NULL,NULL);

  for(i=0 ; i < readSet.fd_count ; i++)
  {
    if(FD_ISSET(readSet.fd_array[i],readSet))
    {
      //o socket está pronto pra leitura
      recv(...)
    }
  }
}
```

onde FD_ISSET é uma macro que determina se um particular socket está pronto para leitura. A descrição detalhada das funções, macros e tipos acima pode ser encontrada em [28].

Existem algumas desvantagens no uso de select para processar um grande número de conexões. Primeiro há um limite no número de sockets que podem ser verificados para eventos de leitura, escrita e exceções. O número máximo de sockets que podem ser verificados para algum tipo de evento em um select é FD_SETSIZE. Em alguns sistemas operacionais, como 4.4BSD, o FD_SETSIZE é definido como:

```
#ifndef FD_SETSIZE
#define FD_SETSIZE 256
#endif
```

PUC-Rio - Certificação Digital Nº 0410842/CA

Em Windows esse valor é 64, mas ele pode ser alterado definindo-se FD_SETSIZE antes de incluir o arquivo winsock2.h. Outro problema é o custo de processamento dos eventos. A cada chamada de select, os conjuntos de sockets em espera de eventos de leitura, de escrita e de erros devem ser reinicializados. Isso é demonstrado no exemplo com a linha:

```
memcpy(&readSet,&initReadSet,sizeof(readSet));
```

onde readSet é reinicializado com os sockets de initReadSet.

Além disso, a cada retorno da função select, possui-se apenas o número de sockets que possuem eventos a serem tratados. Existe ainda a necessidade de procurar nos conjuntos de sockets, por aqueles com eventos. Num servidor multijogador em massa, onde pode haver mais de mil sockets em cada conjunto, essa busca pode acabar sendo custosa.

Uma outra função semelhante à select, porém não existente para Windows é a função poll, que possui o seguinte protótipo:

```
int poll(struct pollfd * fdarray, unsigned long nfds,
  int timeout);
```

onde fdarray é um ponteiro para o começo de um vetor de estruturas pollfd, nfds é o número de elementos nesse vetor e timeout é o número de milisegundos que a função bloqueia antes de voltar. A estrutura pollfd possui o seguinte formato:

```
struct pollfd
{
  int fd;           //o socket
  short events;     //eventos de interesse no socket
  short revents;    //eventos que ocorreram no socket
};
```

Na estrutura pollfd, fd é o descritor do socket, events especifica os eventos que se deseja verificar e revents armazena o estado do socket no retorno da função. O *flag* POLLIN pode ser usado para verificar se existem dados para serem lidos e POLLOUT pode ser usado pra verificar se dados podem ser escritos no socket. Esses não são os únicos *flags* que podem ser especificados, outros podem ser encontrados em [28]. A função poll retorna o número de descritores que precisam ser tratados. Abaixo está um exemplo do uso de poll:

PUC-Rio - Certificação Digital Nº 0410842/CA

```
pollfd client[MAX_CLIENTS];
client[0].fd = clientSocket;
client[0].events = POLLIN;
...
result = poll(clients,numClients,INFTIM);
int c = 0;
for(i=0 ; i < numClients, c < result ; i++)
{
  if(clients[i].revents & POLLIN)
  {
    c++;
    //o socket está pronto pra leitura
    recv(client[i].fd,buffer,sizeof(buffer),0);
  }
}
```

PUC-Rio - Certificação Digital Nº 0410842/CA

Uma vantagem do poll sobre o select é que não há necessidade de reinicializar o vetor de pollfd a cada chamada de poll. Isso ocorre porque o parâmetro de entrada é gravado em uma variável(events) e o retorno é gravado em outra variável(revents). No select, as variáveis de entrada recebem novos valores da função, havendo então a necessidade de reinicializá-las a cada chamada de select.

Outra vantagem do poll é que não há limite para o número de sockets sendo tratados. Na necessidade de tratar mais sockets é só aumentar o vetor de pollfd. No entanto, ainda há necessidade de buscar pelos sockets que tiveram eventos, de maneira semelhante à select, onde, a cada retorno, existe a possibilidade de que seja necessário passar por muitos sockets que não possuem eventos a serem tratados.

Existem outras maneiras consideradas mais eficientes para processar um grande número de conexões. A desvantagem dessas maneiras é que elas costumam ser específicas do sistema operacional; portanto, enquanto ganha-se com eficiência, perde-se em portabilidade.

3.2 Usando verificação constante em sockets não bloqueantes

Existe uma maneira de processar muitas conexões usando apenas a API do Sockets que pode se aplicar bem a algumas situações. Ela faz uso da verificação constante de sockets não-bloqueantes. Os sockets convencionais

são bloqueantes. Quando não há mensagem de um determinado cliente, chamar recv em seu socket causa um bloqueio até que uma mensagem do cliente chegue. Para colocar um socket em modo não-bloqueante, usa-se:

```
unsigned long enable = 1;
ioctlsocket(socket, FIONBIO, (unsigned long*)&enable);
```

Cada vez que é chamada uma função como recv, e ela está num momento em que deveria bloquear, é gerado em Windows o erro WSAEWOULDBLOCK e, em Unix, é gerado o erro EWOULDBLOCK.

Após colocado em estado não-bloqueante, para processar todos os sockets basta passar por eles todos chamando recv e send (caso haja mensagens a enviar). Aqueles sockets que não retornarem que deveriam bloquear, receberam ou enviaram dados.

Ficar constantemente verificando um socket e recebendo, a maior parte do tempo, a mensagem que a operação(recv ou send) deveria bloquear é um gasto desnecessário de recursos de sistema. Porém, caso um jogo use bastante a comunicação TCP e exista um número considerável de jogadores conectados, de modo que a cada vez que chega o momento de chamar um recv ou um send passou-se tempo suficiente para que haja uma elevada possibilidade da operação ser bem sucedida, essa maneira de processar sockets pode passar a possuir boa eficiência. Cada vez que um recv ou um send é chamado em um socket, e a operação é bem sucedida, não há gasto desnecessário de processamento.

Essa maneira de processar sockets possui algumas vantagens em comparação ao uso de select e poll, pois não há necessidade de se achar o socket que gerou algum evento. No momento em que é detectado que chegou uma mensagem, já se sabe de qual socket a mensagem veio.

Abaixo está um exemplo de processamento de mensagens recebidas dos clientes usando sockets não-bloqueantes:

```
for(i = 0 ; i < numClients ; i++)
{
  r = recv(clientSocket[i],buffer,sizeof(buffer),0);

  if(r != SOCKET_ERROR)
  {
    processMesssage(i,buffer,r);
  }
  else
```

PUC-Rio - Certificação Digital Nº 0410842/CA

```
  {
    error = WSAGetLastError();
    if(error == WSAEWOULDBLOCK)
    {
      continue;
    }
    else
    {
      onRecvError(i,error);
    }
  }
}
```

3.3
Usando IO Completion Ports de Windows

PUC-Rio - Certificação Digital Nº 0410842/CA

A tecnologia *IO Completion Ports* do Windows permite processar milhares de conexões mantendo um bom desempenho [17]. Esta tecnologia é para servidores com elevada taxa de tráfego e é sugerida como a maneira de melhor desempenho para processar um grande número de sockets em Windows [9].

Como mencionado anteriormente, Winsock possui funções *overlapped* que funcionam de maneira assíncrona. Essas funções são úteis para operações que demoram para completar, pois permitem a aplicação voltar logo a processar outras tarefas enquanto a operação é completada internamente no sistema operacional. As funções *overlapped* são usadas em conjunto os *IO Completion Ports.*

Segundo [17], *Completion Ports* é a única maneira que realmente permite uma alta escalabilidade. Essa maneira é otimizada para o funcionamento interno do sistema operacional.

Um *completion port* é uma lista em que o sistema operacional coloca notificações de operações *overlapped* terminadas. A aplicação cria algumas threads para receber essas notificações, idealmente uma thread por processador, para evitar troca de contexto.

Para criar um completion port usa-se a seguinte função:

```
HANDLE CreateIoCompletionPort(HANDLE FileHandle,
                              HANDLE ExistingCompletionPort,
                              DWORD CompletionKey,
                              DWORD NumberOfConcurrentThreads);
```

Essa função é usada para dois motivos:

- Para criar o objeto *completion port*
- Para associar um SOCKET ao *completion port*

Quando se deseja criar o *completion port*, o primeiro parâmetro deve ser definido como INVALID_HANDLE_VALUE, ExistingCompletionPort deve ser definido como NULL e CompletionKey é ignorado. O último parâmetro, NumberOfConcurrentThreads, especifica quantas threads podem executar concorrentemente num *completion port*. Colocando o valor 0 nesse parâmetro permite que um número de threads igual ao número de processadores executem concorrentemente. Ou seja, permite a execução de uma thread por processador. Um exemplo dessa criação é o seguinte:

```
HANDLE completionPort =
  CreateIoCompletionPort(INVALID_HANDLE_VALUE,
                         NULL,
                         0,
                         0);
```

Quando deseja-se associar um socket ao *completion port*, o próprio socket deve ser passado no primeiro parâmetro. No segundo passa-se o *handle* do *completion port* criado. O terceiro parâmetro consiste num valor que é passado no momento em que uma thread pega uma notificação de uma operação completa de um socket. Esse parâmetro pode ser usado para guardar um ponteiro para uma área de memória com informação relacionada ao socket por exemplo. Exemplo :

```
CreateIoCompletionPort((HANDLE)socket,
                       completionPort,
                       (DWORD) clientDataPointer,
                       0);
```

Depois que um socket está associado ao *completion port*, toda operação *overlapped* do socket gera eventos no *completion port*. Para pegar esses eventos de operações terminadas, deve-se usar a função GetQueuedCompletionStatus que possui o seguinte protótipo:

```
BOOL GetQueuedCompletionStatus(HANDLE completionPort,
                               LPDWORD numberOfBytesTransferred,
                               LPDWORD completionKey,
                               LPOVERLAPPED * overlapped,
                               DWORD milliseconds);
```

PUC-Rio - Certificação Digital Nº 0410842/CA

O primeiro parâmetro é o *completion port*, o qual se deseja adquirir notificações. numberOfBytesTransferred recebe o número de bytes transferidos na operação de Entrada/Saída. completionKey recebe o valor passado no momento em que o socket é associado ao *completion port*. overlapped recebe um ponteiro para a estrutura OVERLAPPED passada no momento da chamada de uma função *overlapped*. milliseconds especifica por quantos milisegundos a função deve bloquear, o valor INFINITE deixa a função bloqueada eternamente. Se a função retorna com o valor diferente de 0, ela conseguiu adquirir uma notificação de uma operação bem sucedida. Se ela retorna 0 com overlapped igual a NULL, é possivel que a função tenha retornado por timeout. Para verificar esse fato, basta chamar GetLastError() e verificar se o retorno é WAIT_TIMEOUT. Se a função retorna 0 e overlapped não é NULL, a função remove da lista do *completion port* uma notificação de operação com falha.

Uma forma de fazer uso dessas funções é mostrada no pseudo-código abaixo:

```
Criar um completion port
Pegar o número de processadores do sistema
Para cada processador do sistema
  Criar uma thread de processamento de notificações
Criar um socket com flag overlapped
Associar o socket ao completion port
Fazer uma operação overlapped com o socket
...
Thread de processamento de notificações
  Repetir infinitamente
    Pegar uma notificação do completion port
    Se a notificação foi bem sucedida
      Processar notificação
```

PUC-Rio - Certificação Digital Nº 0410842/CA

Às vezes se faz necessário que a notificação da operação completa venha com alguma informação a mais como, por exemplo, se a operação foi de send ou de recv, ou para qual cliente essa operação foi realizada. Para enviar essas informações, podemos fazer uso do ponteiro para a estrutura OVERLAPPED retornado ao pegar a notificação.

Para fazer isso, deve-se criar uma estrutura como a abaixo:

```
struct Operation
{
```

```
  OVERLAPPED overlapped;
  int type;
};
```

Quando se for o momento de fazer uma operação *overlapped*, deve-se preencher a estrutura de acordo com a seguinte operação:

```
...
Operation op;

op.type = SEND_OPERATION;

ZeroMemory(&op.overlapped, sizeof(OVERLAPPED));

WSASend(socket,
        buffer,
        1,
        &BytesSent,
        Flags,
        &op.overlapped,
        NULL);
```

No código apresentado, é gravada, na estrutura criada, a operação que está sendo efetuada. Quando a função GetQueuedCompletionStatus retornar, basta converter o ponteiro do tipo LPOVERLAPPED para o tipo criado(Operation), para poder acessar seus dados. O código abaixo mostra como processar na thread de notificações, essas informações adicionais:

```
DWORD WINAPI WorkerThread(LPVOID server)
{
  HANDLE completionPort = GetCompletionPort(server);
  DWORD bytesTransferred;
  SOCKET socket;
  Operation * op;
  BOOL b;

  while(true)
  {
    b = GetQueuedCompletionStatus(completionPort,
                                  &bytesTransferred,
```

PUC-Rio - Certificação Digital Nº 0410842/CA

```
                                              (DWORD*)&socket,
                                              (OVERLAPPEP**)&op,
                                              INFINITE);

    if(b != FALSE)
    {
      OnOperationError(op);
      break;
    }

    switch(op->type)
    {
    case SEND_OPERATION:
      Tratar operação de envio ...
      break;
    case RECV_OPERATION:
      Tratar operação de recebimento ...
      break;
    }
  }
}
```

PUC-Rio - Certificação Digital Nº 0410842/CA

Segundo [15], *IO Completion Ports* é o suporte do Windows NT para o desenvolvimento de servidores com boa escalabilidade. Caso a plataforma Windows se demonstre interessante para executar o servidor de MMORPG, essa é a maneira principal a se usar para processar sockets. [9] é uma boa fonte de informação sobre *IO Completion Ports*.

As figuras 3.1, 3.2 e 3.3 mostram o desempenho de servidores usando múltiplas threads, select e IOCP respectivamente. A figura 3.4 mostra todos os resultados em um único gráfico. O teste realizado compara os desempenhos dos servidores na conexão de clientes e na troca de 10 mensagens TCP, ou seja, é calculado o tempo que demora para um cliente conectar e trocar 10 mensagens com o servidor.

O servidor multithread cria uma thread a cada novo cliente e possui o pior desempenho com o tempo médio para conexão e troca de mensagens de 271.90 milisegundos. O servidor select tem seu FD_SETSIZE alterado para 1000, permitindo-o ir a até 1000 clientes. Seu tempo médio é de 91.48 milisegundos. O servidor IOCP possui o melhor desempenho com o tempo médio de 52.92 milisegundos.

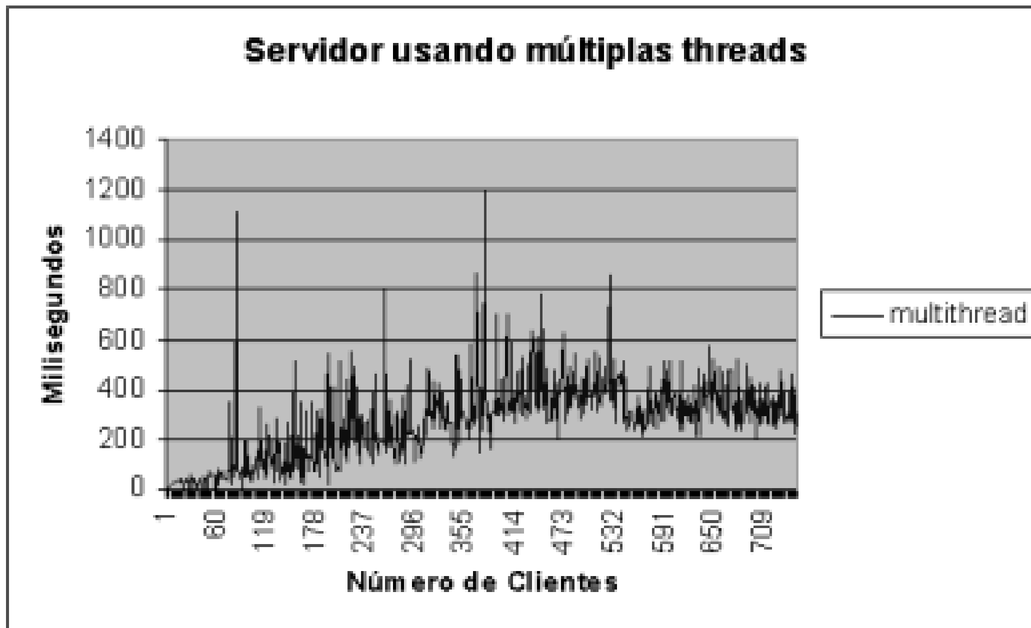

Figura 3.1: Performance de um servidor usando múltiplas threads

PUC-Rio - Certificação Digital Nº 0410842/CA

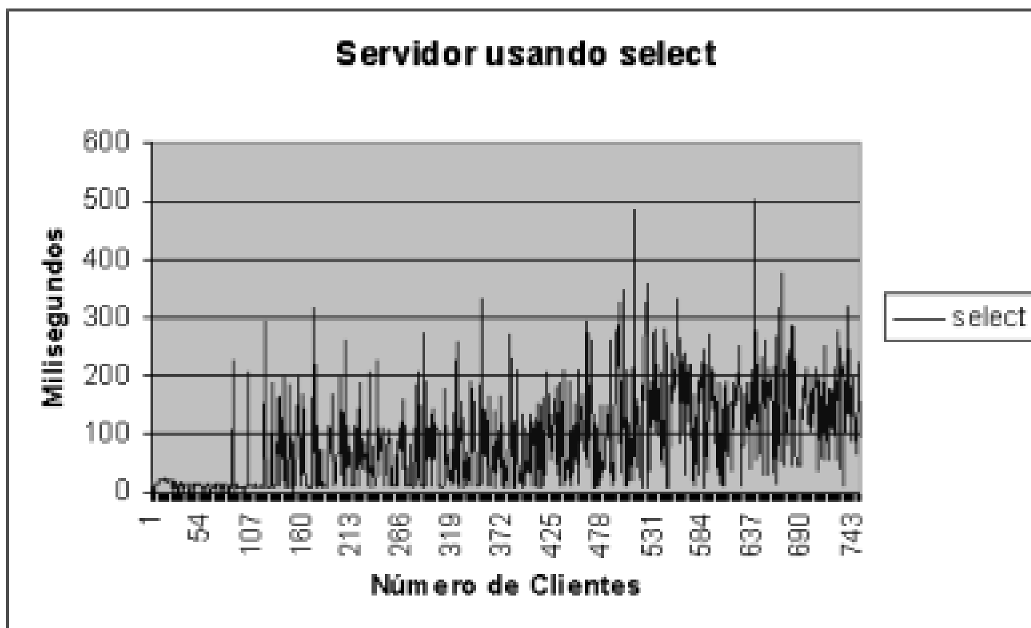

Figura 3.2: Performance de um servidor usando select

PUC-Rio - Certificação Digital Nº 0410842/CA

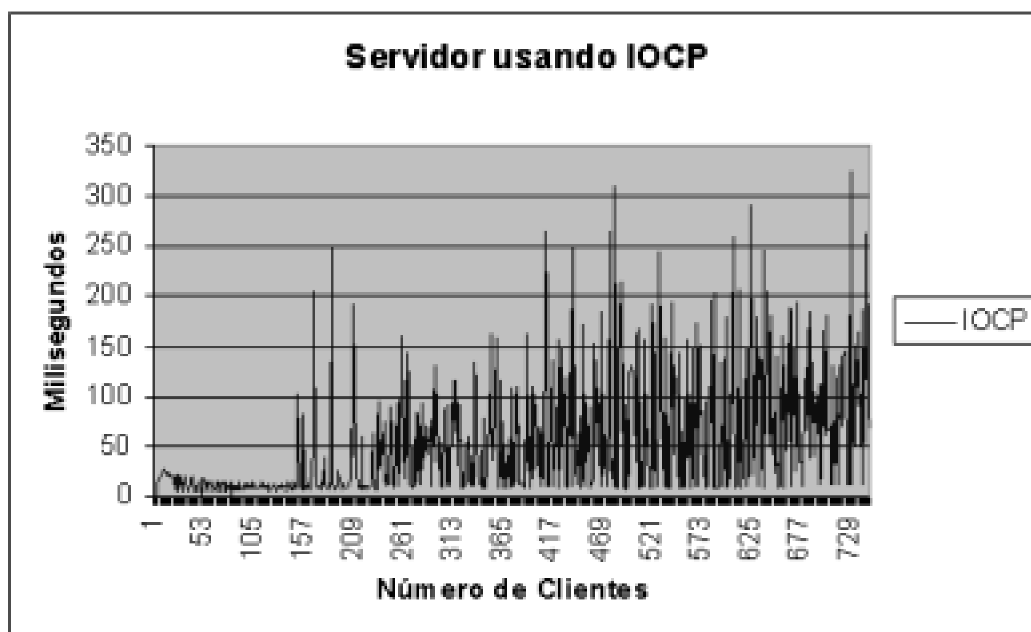

Figura 3.3: Performance de um servidor usando IOCP

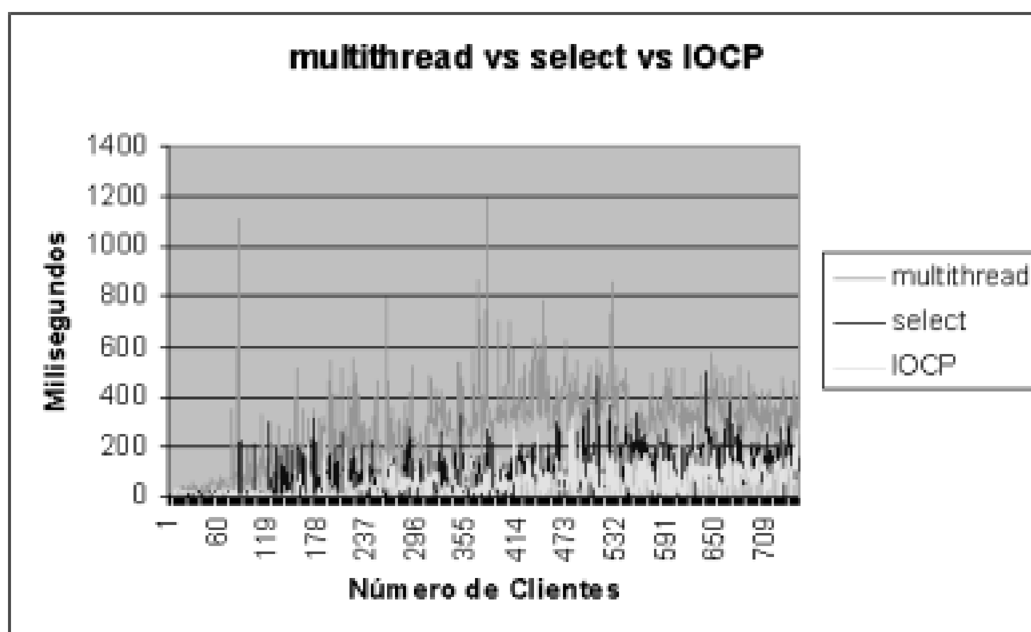

Figura 3.4: Comparação de performance entre servidores usando múltiplas threads, select e IOCP

3.4 Usando kqueue em Unix

kqueue é uma interface que permite filtrar eventos interessantes para uma aplicação [28]. Esses eventos podem ser os seguintes: o momento em que existe dados a serem lidos num socket; o momento quando um socket está pronto para ser escrito; timeouts; e outros mais. Para criar uma kqueue deve-se usar a função:

```
int kqueue(void);
```

Essa função retorna um novo descritor para uma kqueue. A função kevent é usada para tanto registrar eventos de interesse como para determinar se algum evento ocorreu. Seu protótipo é o seguinte:

```
int kevent(int kq,
           const struct kevent * changelist,
           int nchanges,
           struct kevent * eventlist,
           int nevents,
           const struct timespec * timeout);
```

changelist e nchanges descrevem as mudanças a serem feitas nos eventos de interesse e podem ser NULL e 0 caso nenhuma mudança seja desejada. Quaisquer eventos de interesse que foram acionados são retornados em eventlist; nevents armazena o número de eventos do vetor eventlist. A função retorna o número de eventos que foram acionados. timeout pode ser NULL para bloquear ou pode-se especificar o tipo timespec para definir o tempo a bloquear na função.

kevent possui o seguinte formato:

```
struct kevent
{
  uintptr_t ident;
  short     filter;
  u_short   flags;
  u_int     fflags;
  intptr_t  data;
  void    * udata;
};
```

PUC-Rio - Certificação Digital Nº 0410842/CA

ident armazena o descritor do socket, filter identifica o tipo de evento de interesse. EVFILT_READ pode ser usado para identificar o interesse em saber quando o socket pode ser lido, e EVFILT_WRITE pode ser usado para especificar interesse em saber quando o socket pode ser escrito. flags identificam a ação a se fazer. EV_ADD serve para adicionar um novo evento, ED_DELETE serve para remover um evento. Os outros dados não são usados nesta dissertação e podem ser ignorados.

Existe uma macro que pode ser usada para preencher eventos. Essa macro possui o seguinte prototipo:

```
void EV_SET(struct kevent *kev,
            uintptr_t ident,
            short filter,
            u_short flags,
            u_int fflags,
            intptr_t data,
            void * udata);
```

Abaixo está um exemplo do uso de kqueue:

```
...
int kq;
struct timespec time;
struct kevent evread,evwrite;

EV_SET(&evread, socket1, EVFILT_READ, EV_ADD, 0, 0, NULL);
EV_SET(&evwrite, socket2, EVFILT_WRITE, EV_ADD, 0, 0, NULL);

kq = kqueue();
time.tv_sec = time.tv_nsec = 0;

kevent(kq, &evread, 1, NULL, 0, &time);
kevent(kq, &evwrite, 1, NULL, 0, &time);

struct kevent events[2];
int nevents = 2;

while(true)
{
  nev = kevent(kq, NULL, 0, events, nevents, NULL);
```

PUC-Rio - Certificação Digital Nº 0410842/CA

```
  for(i = 0 ; i < nev ; i++)
  {
    if(events[i].filter == EVFILT_READ)
    {
      recv((int)events[i].ident, buffer, sizeof(buffer), 0);
    }
    else
    {
      ...o socket pode ser escrito, enviar os dados que
        precisam ser enviados
    }
  }
}
```

No código mostrado, o evento evread para leitura e o evwrite de escrita são registrados. Quando o kevent bloqueante de dentro do loop retorna, nev possui o número de eventos retornados. Cada evento então é tratado e verifica-se o tipo de filtro do evento. Caso o filtro seja de leitura são lidos os dados do socket. Se o filtro for de escrita pode-se enviar alguma mensagem que precise ser enviada para o socket.

kqueue tem a vantagem que já retorna os sockets que tiveram algum evento ativado. Em algumas aplicações, a substituição do uso do select ou poll pelo uso de kqueue trouxe melhorias no desempenho [11]. Mais informações sobre esse tópico encontram-se em [28] e [11].

PUC-Rio - Certificação Digital Nº 0410842/CA