4
Avaliações e resultados

Em engenharia, ao realizar uma medição, devemos observar o impacto que esta causa no sistema monitorado. Em outras palavras, qual interferência que a medição irá causar. A ferramenta elaborada por este trabalho interfere nas chamadas de sistema de principal utilização pelo sistema operacional. Por este motivo, para sabermos a real interferência que esta ferramenta causa, além dos testes funcionais, é imperativo realizar testes que avaliem o desempenho por meio da utilização intensiva de tais chamadas.

Dessa forma, a seção 4.1 abrirá o capítulo com a avaliação da degradação de desempenho que o código inserido no núcleo causa no sistema operacional. Já na seção 4.2, analisaremos o desempenho das ferramentas desenvolvidas para a interface núcleo usuário e, na seção 4.3, da GUI com o usuário. Na seção 4.4 serão avaliadas as funcionalidades da ferramenta, finalizando o capítulo na seção 4.5, com as considerações finais.

Os testes foram conduzidos no cluster de simulação de reservatórios da unidade de operação do Rio de Janeiro (UO-Rio) da Petrobras. O cluster é composto por 32 nós de processamento HP DL160, cada nó está equipado com dois processadores Intel X5570 (4 núcleos @ 2.93GHz), 48Gb de memória RAM DDR3 @ 1333Mhz, interconectados por rede ethernet gigabit (1Gbps) e InfiniBand 4X QDR (32Gbps), dois discos SAS de 160 gigabytes agrupados em configuração RAID 1 (espelhamento) por hardware usando controladoras SAS *(Serial Attached SCSI)*, e sistema operacional Red Hat Enterprise Linux 5 *patchlevel* 8. Para o propósito dos testes, foram alocados dois nós do referido cluster.

4.1
Desempenho do código inserido no núcleo

Para avaliar a degradação de desempenho causada pelo módulo inserido no núcleo, é necessário se aprofundar nos detalhes de como foram desenvolvidos os pontos críticos do código, especificamente, os lugares que lidam com grande concorrência inter-processos. Vale lembrar que tais regiões do código são habilitadas apenas quando o módulo está em funcionamento. Assim, conforme indicado na seção 3.2.1, é possível identificar, revisitando a figura 3.11, dois pontos no fluxograma que podem ser classificados nesta categoria:

i. **O ponto de entrada do desvio das chamadas de sistema**: Nesse ponto, todos os eventos de E/S precisam ser avaliados, através da

procura em uma tabela de dispersão pela identificação do processo, se serão monitorados.

ii. **Identificação do descritor de arquivo e gravação no buffer circular**: Dos processos monitorados, é necessário analizar o descritor de arquivos para concluir se o evento está direcionado a rede. Caso afirmativo, deve ser interceptado e salvo no buffer circular (figura 3.7).

Pode-se afirmar que o grau de utilização de (i) é proporcional à quantidade de eventos de E/S realizados. Também, é importante ressaltar que, na assinatura das chamadas de sistema que representam esses eventos (figura 4.1), o último parâmetro representa o tamanho do bloco (quantidade de bytes recebidos ou enviados) utilizado. Logo, de acordo com a figura 4.2, quanto maior o tamanho do bloco, menor a quantidade de eventos que serão utilizados para operar com uma mesma quantidade de bytes.

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Figura 4.1 Assinatura das chamadas de sistema *read*() e *write*()

$$\text{Eventos} = \frac{Tamanho\ do\ arquivo}{Tamanho\ do\ bloco}$$

Figura 4.2 Cálculo da quantidade de eventos necessários para ler um arquivo

Dessa forma, a cada ocorrência de um evento de E/S, o algoritmo descrito na figura 3.11 é percorrido. Assim, a utilização de blocos maiores, com o propósito de ler ou escrever uma mesma quantidade de bytes, representa menos repetições do algorítmo, e consequentemente, por executá-lo menos vezes, menor degradação de desempenho. Ainda que blocos maiores provoquem menos invocações do referido algoritmo, este, para constar, deve ser rápido o suficiente para, proporcionalmente, utilizar uma fração despresível do tempo total do evento.

Observando a figura 4.3, verificamos que ao encontrar a entrada do desvio, a requisição deve solicitar uma trava *(lock)*, realizar uma procura em uma tabela de dispersão, e liberar a trava. Apenas se houver a confirmação de que o processo deve ser monitorado, é invocada a execução da instrução que verifica se o evento é destinado à rede.

A opção pela utilização de uma tabela de dispersão teve como objetivo evitar alterar a estrutura do núcleo que representa um processo, que, na falta de outra forma mais apropriada, poderia ser alterada para abrigar um bit que representasse a seleção do processo para a monitoração. Entretanto, tal modificação poderia prejudicar a compatibilidade do código do núcleo modificado com os demais códigos existentes.

Assim, de modo a não haver alterações nas principais estruturas do núcleo, optou-se por utilizar uma tabela de dispersão indexada pelos números de identificação dos processos. O custo computacional de realizar uma procura em uma tabela

de dispersão é logarítmico, equivalente a $O(\log n)$. Portanto, utilizar esse tipo de procura, em relação à modificação de uma estrutura principal do núcleo, compensa a decisão.

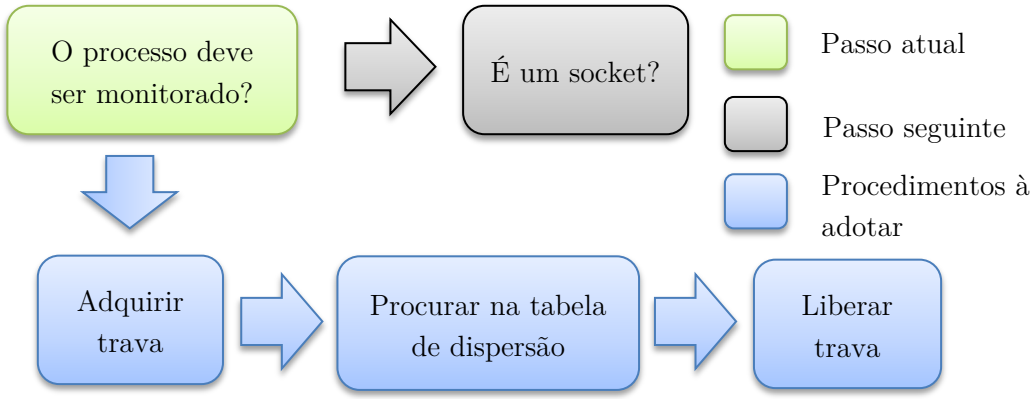

Figura 4.3 Diagrama do algoritmo aplicado na função de entrada

Entretanto, além do custo computacional, devido à possibilidade de concorrência, a tabela de dispersão requer um mecanismo que a proteja quando houver uma consulta simultânea a uma alteração. Dessa forma, o mecanismo indicado para esse fim foi o *kernel mutex*, devido à simplicidade de utilização e por forçar um reescalonamento do processo na ocasião de um conflito [50]. Dessa forma, se dois processos, ao acessarem a tabela de dispersão, entrarem simultaneamente na área da região crítica, um deles será reescalonado, liberando recursos para os demais processos que não estejam concorrendo por essa região. Assim, a primeira sequência de testes irá avaliar se, a cada evento de E/S, o impacto de entrar na região crítica e pesquisar na tabela de dispersão degrada o desempenho do sistema operacional significantemente. Há a expectativa de que a entrada na região crítica cause certa degradação de desempenho.

Para a próxima sequência de testes, de acordo com a figura 4.4, é necessário avaliar qual impacto causa a identificação do descritor de arquivos e a cópia dos parâmetros de entrada das chamadas de sistema para o buffer circular. A identificação do descritor de arquivos é realizada por intermédio da invocação de um método normalmente já utilizado pelo núcleo quando há comunicação com a rede, logo, a expectativa de degradação de desempenho dessa identificação é baixa.

Entretanto, apesar do algoritmo utilizado pelo buffer circular ser livre de intertravamentos, tal afirmação é verdadeira apenas para operações de leitura e escrita simultâneas. Quando há operações concorrentes de escrita, torna-se necessário a utilização de um mecanismo intertravamento. Dessa forma, para coordenar as operações de escrita no buffer, similar ao mecanismo utilizado na entrada do desvio, e com base na mesma justificativa, utilizou-se também um *kernel mutex*. Logo, a expectativa de degradação de desempenho, para as operações de escrita concorrentes no buffer circular, é igualmente significativa.

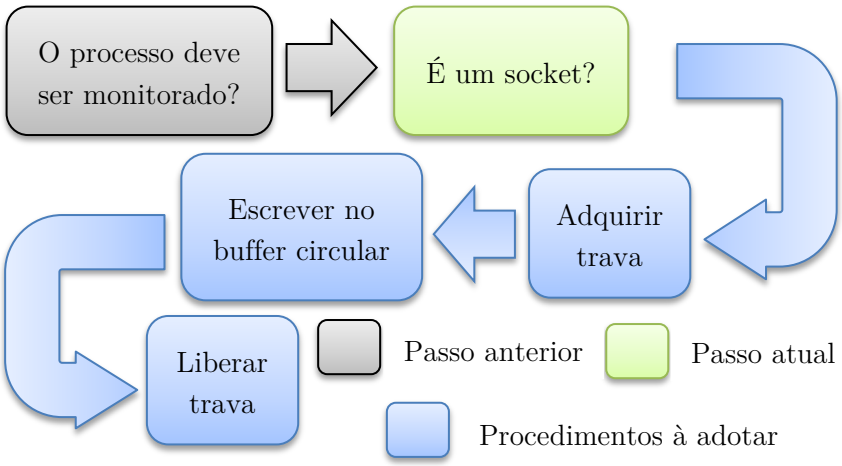

Figura 4.4 Diagrama do algoritmo de identificação do *socket*

4.1.1 Degradação no ponto de entrada

Para avaliar o desempenho no ponto de entrada do desvio, com o objetivo de medir a degradação dos eventos de E/S de uma forma geral, devemos analisar, principalmente, a região crítica que protege a tabela de dispersão responsável por identificar se o processo atual deve ser monitorado. Para tal, sabe-se, de antemão, que o núcleo atravessa a referida região crítica a cada evento de E/S.

De acordo com a figura 4.2, sabe-se que existe uma proporção entre a quantidade de eventos e tamanho do bloco escolhido. Sabe-se também que operações de leitura, em geral, são mais rápidas que operações de escrita. Assim, de modo a estressar a região crítica, podemos variar, para um arquivo de tamanho constante, o tamanho do bloco dos eventos de leitura. Paralelamente, para explorar a característica da região crítica de reescalonar processos conflitantes, podem-se lançar *threads* simultâneas com a finalidade de gerar eventos concorrentes de leitura. Para esse fim, foi desenvolvido um *script*[8] que realiza as seguintes atividades:

- Criar um arquivo temporário, preenchido por bytes nulos, de tamanho definido pelo usuário, que será removido ao fim do teste.
- Incluir-se na monitoração automaticamente.
- Criar uma barreira para sincronização.
- Criar o número de *threads* simultâneas definido pelo usuário. Cada *thread* é responsável por ler o arquivo temporário por completo usando blocos de tamanho definido pelo usuário.
- Aguardar a sinalização do usuário pela liberação da barreira.
- Contabilizar o tempo gasto pelas *threads*
- Excluir-se automaticamente da monitoração ao término da execução.

[8] Arquivo contendo comandos para execução em lote.

Com esse *script*, é possível estressar o sistema operacional o suficiente para conhecer quanto o ponto de entrada do desvio degradada o desempenho deste. Assim, foram selecionados blocos de 1, 10, 100, 1000 e 8192 bytes para ler um arquivo de 10.000.000 bytes, que equivalem a 10.000.000, 1.000.000, 100.000, 10.000 e a aproximadamente 1220 eventos de leitura respectivamente. O valor de 8192 bytes, escolhido devido à forma como é organizada a memória no Linux, equivale a duas páginas de memória. Nas seções subsequentes serão apresentados os resultados apenas, a discussão sobre eles virá na seção 4.1.1.4.

4.1.1.1
Desempenho de E/S referencial

Foram considerados como tempos de referência os valores obtidos com o módulo inativo. Por não existir demanda por recursos concorrentes ao teste, os valores obtidos correspondem ao máximo de desempenho que se pode desenvolver na máquina descrita no início do capítulo. A figura 4.5, representa cem amostras do tempo transcorrido, em segundos, entre o início e o fim da execução do *script*, sendo vinte amostras para cada bloco escolhido, usando 1, 2, 4 ou 8 *threads*. Os valores médios de tempo são apresentados na tabela 4.1.

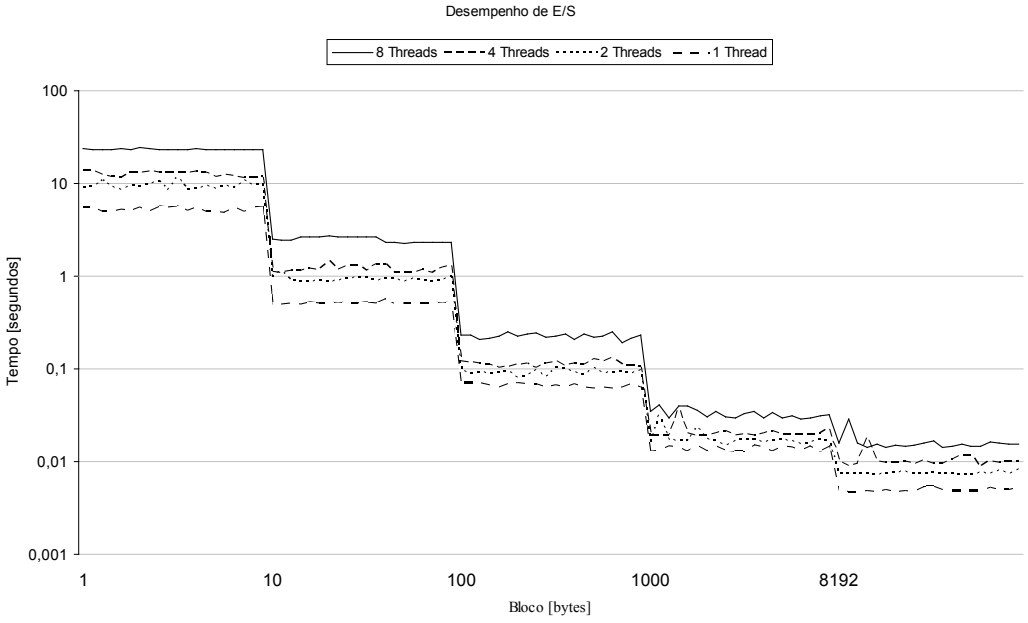

Figura 4.5 Desempenho de E/S sem carga extra

	1 byte	10 bytes	100 bytes	1000 bytes	8192 bytes
8 *threads*	23,286s	2,490s	0,227s	0,033s	0,016s
4 *threads*	12,639s	1,201s	0,114s	0,021s	0,010s
2 *threads*	9,550s	0,933s	0,092s	0,018s	0,008s
1 *thread*	5,271s	0,511s	0,066s	0,014s	0,005s

Tabela 4.1 Médias de desempenho obtidas sem carga extra

4.1.1.2
Desempenho de E/S com a monitoração ativada

Ao carregar o módulo e habilitar a monitoração, todos os processos, mesmo os não monitorados, devem passar pela região crítica que protege a tabela de dispersão. Logo, os valores obtidos correspondem ao desempenho de referência, apresentados na seção 4.1.1.1, acrescido da degradação de desempenho causada pela entrada na região crítica e pela pesquisa na tabela de dispersão. A figura 4.6, representa cem amostras do tempo transcorrido, em segundos, entre o início e o fim da execução do *script* com a monitoração habilitada, sendo vinte amostras para cada bloco escolhido, usando 1, 2, 4 ou 8 *threads*. Os valores médios de tempo obtidos são apresentados na tabela 4.2.

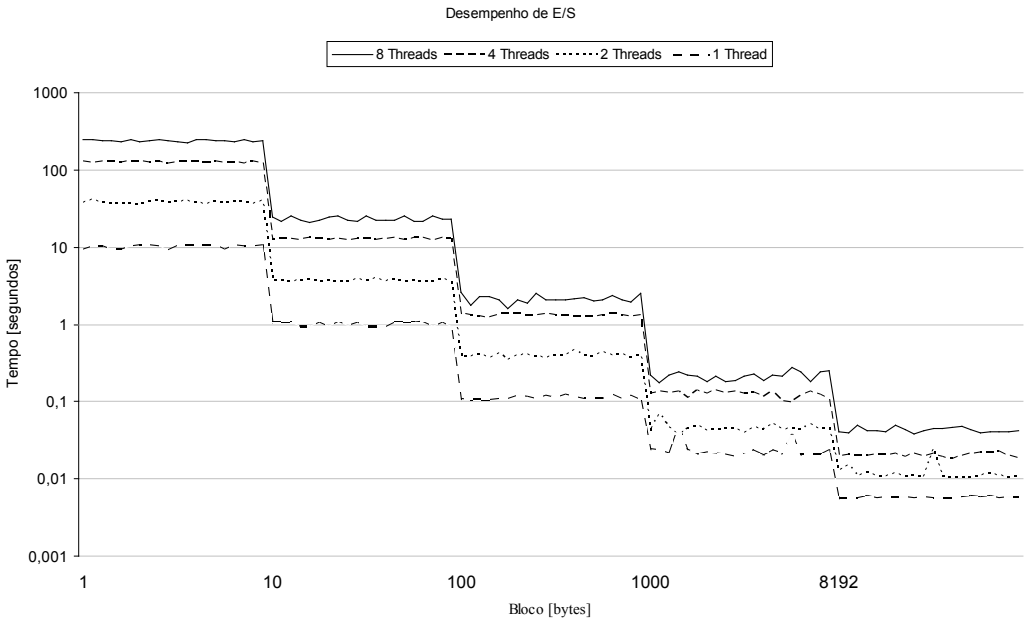

Figura 4.6 Desempenho de E/S com a monitoração habilitada

	1 byte	10 bytes	100 bytes	1000 bytes	8192 bytes
8 *threads*	239,904s	23,165s	2,138s	0,216s	0,043s
4 *threads*	126,753s	12,774s	1,315s	0,125s	0,020s
2 *threads*	37,998s	3,724s	0,398s	0,046s	0,012s
1 *thread*	10,105s	1,001s	0,111s	0,024s	0,006s

Tabela 4.2 Médias de desempenho com a monitoração habilitada

4.1.1.3
Desempenho de E/S do processo em monitoração

Quando um processo está presente na lista de monitoração, além de testar a presença do processo na tabela de dispersão, é necessário avaliar se esta é de fato uma comunicação de rede. Logo, os valores obtidos correspondem ao desempenho de referência, apresentados na seção 4.1.1.2, acrescido da soma da degradação de desempenho causada pela monitoração e pela avaliação do tipo de descritor. A figura 4.7, representa cem amostras do tempo transcorrido, em segundos, com o processo em teste sendo monitorado, entre o início e o fim da execução do *script,* sendo vinte amostras para cada bloco escolhido, usando 1, 2, 4 ou 8 *threads*. Os valores médios de tempo obtidos são apresentados na tabela 4.3.

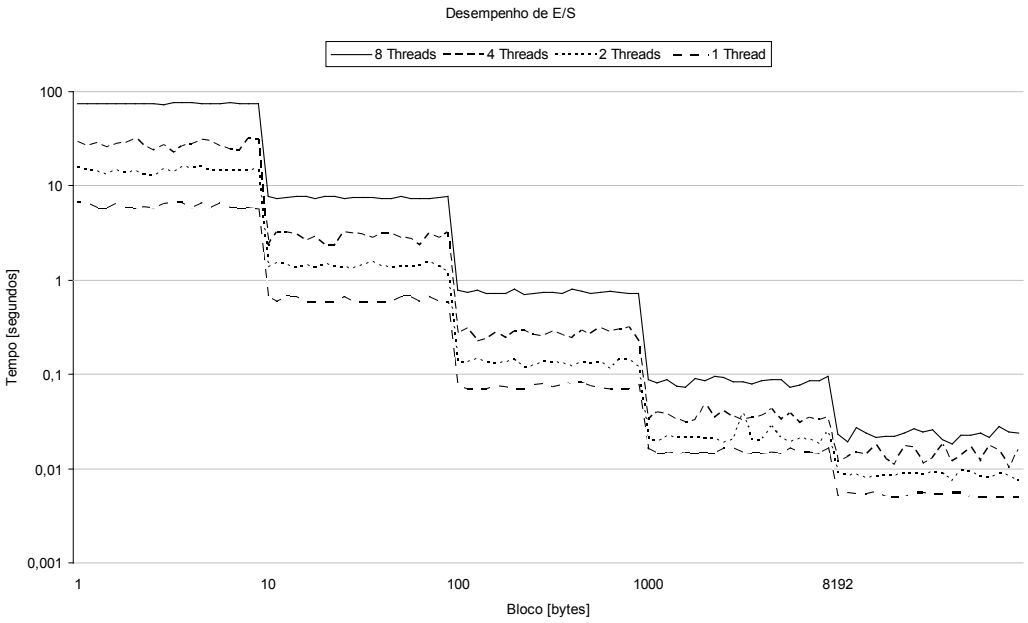

Figura 4.7 Desempenho de E/S do processo monitorado

	1 byte	10 bytes	100 bytes	1000 bytes	8192 bytes
8 *threads*	75,032s	7,513s	0,745s	0,085s	0,023s
4 *threads*	27,499s	2,869s	0,271s	0,036s	0,014s
2 *threads*	14,614s	1,419s	0,132s	0,022s	0,009s
1 *thread*	6,061s	0,614s	0,073s	0,015s	0,005s

Tabela 4.3 Médias de desempenho do processo monitorado

4.1.1.4
Avaliação dos resultados

As médias de tempo apresentadas na tabela 4.3 demonstram que o incremento de tempo nas operações de E/S, apresentado na tabela 4.4, dependendo do grau de concorrência e da quantidade de processo concorrentes, podem ser bem elevados. A justificativa para tais tempos pode ser atribuída, conforme discutido na seção 4.1.1, ao aumento da concorrência pela utilização da região crítica que protege a tabela de dispersão.

	1 byte	10 bytes	100 bytes	1000 bytes	8192 bytes
8 *threads*	930,3%	830,1%	843,0%	554,8%	169,6%
4 *threads*	902,8%	963,8%	1052,6%	502,8%	95,9%
2 *threads*	297,9%	299,2%	331,4%	158,1%	58,4%
1 *thread*	91,7%	95,8%	68,3%	72,3%	17,2%

Tabela 4.4 Incremento de tempo de E/S percentual da seção 4.1.1.2 / 4.1.1.1

Durante a condução dos testes, foram observadas baixa utlização de CPU (para o pior caso, 8 núcleos, blocos de 1 byte), conforme a figura 4.8, e baixa quantidade de processos concorrentes (também para o pior caso), conforme a figura 4.9. Esse comportamento, como já era esperado, indica a característica da região crítica de reescalonar processos concorrentes, ocasionando a degradação de desempenho percebida.

Entretanto, o incremento de tempo de E/S observado para o caminho completo, quando o processo se encontra em monitoramento, mostraram ser inferiores, conforme a tabela 4.5, aos tempos observados pelo caminho parcial, onde apenas a monitoração se encontra habilitada e sem processos em monitoramento. Avaliando as figuras 4.10 e 4.11, observamos, para o pior caso (8 núcleos e blocos de 1 byte), maior utilização de CPU e maior concorrência entre os processos que no caso anterior. Uma justificativa possível para tais resultados seria que caminho completo promove uma pequena dessincronia entre os processos, esta suficiente o bastante para evitar que entrem na região crítica simultaneamente, permitindo melhor utilização da CPU.

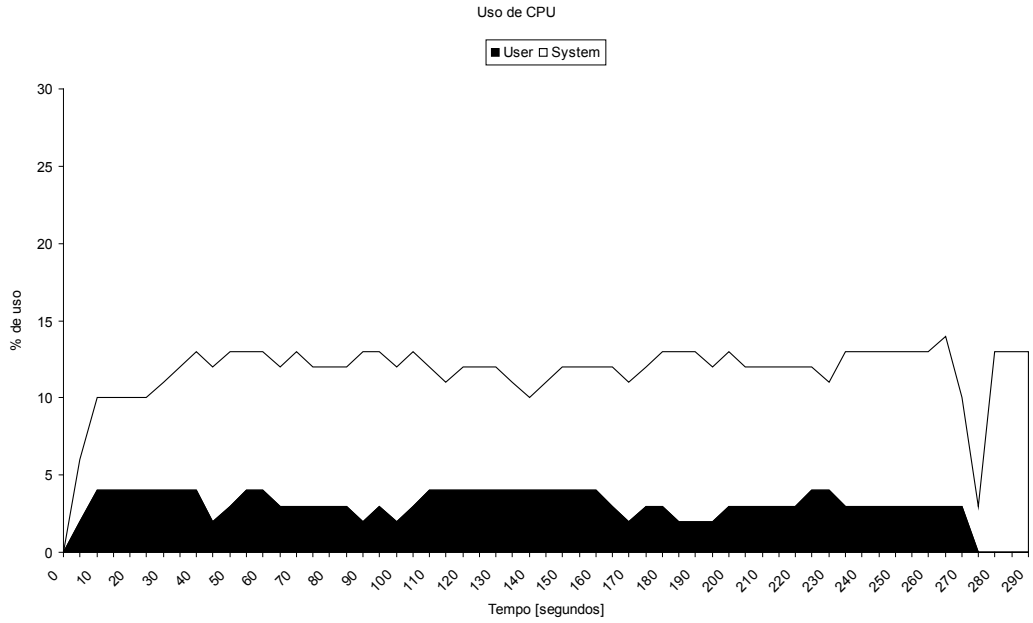

Figura 4.8 Utilização de CPU com os testes da seção 4.1.1.2

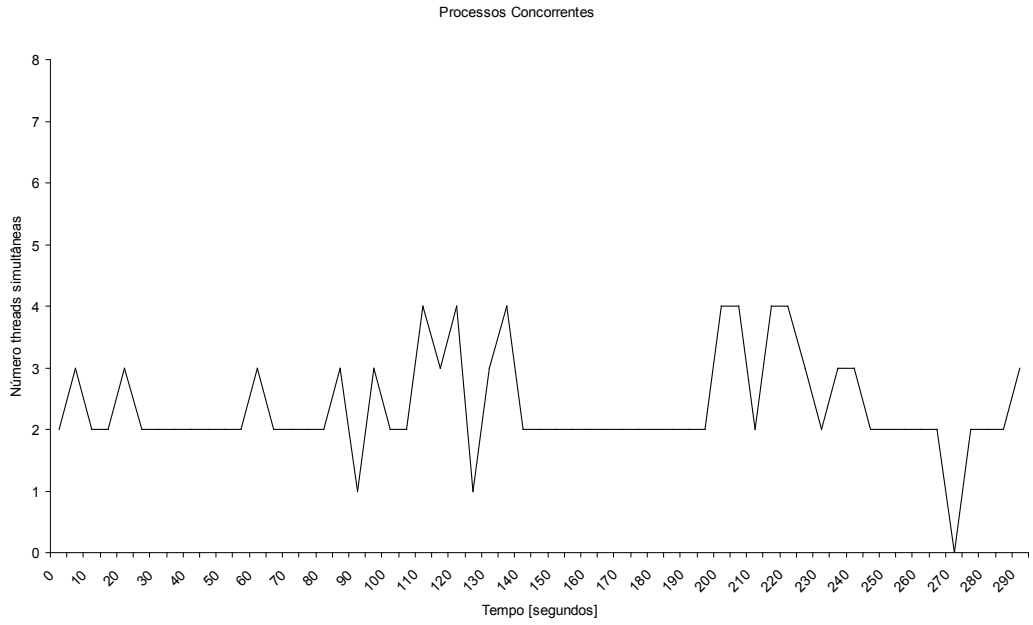

Figura 4.9 Processos concorrentes observados nos testes da seção 4.1.1.2

Ainda que o incremento de tempo de E/S, devido à monitoração, seja bem significante, chegando a até 1000% sobre o tempo de referência, é possível deduzir, observando as tabelas 4.4 e 4.5, que existe um gradiente de redução de tempo para quanto maior o bloco e menor a concorrência (menor quantidade de processos simultâneos). Podemos observar que esta tendência também se confirma para a diferença entre os testes, a tabela 4.6 demonstra o incremento de tempo diferencial percentual relativo entre os testes conduzidos na seção 4.1.1.2 e na seção 4.1.1.3.

	1 byte	10 bytes	100 bytes	1000 bytes	8192 bytes
8 *threads*	222,2%	201,7%	228,7%	156,9%	46,4%
4 *threads*	117,6%	139,0%	137,8%	72,2%	36,6%
2 *threads*	53,0%	52,1%	43,5%	23,6%	14,1%
1 *thread*	15,0%	20,2%	11,3%	8,8%	5,9%

Tabela 4.5 Incremento de tempo de E/S percentual da seção 4.1.1.3 / 4.1.1.1

	1 byte	10 bytes	100 bytes	1000 bytes	8192 bytes
8 *thread*	-68,7%	-67,6%	-65,1%	-60,8%	-45,7%
4 *threads*	-78,3%	-77,5%	-79,4%	-71,4%	-30,3%
2 *threads*	-61,5%	-61,9%	-66,7%	-52,1%	-27,9%
1 *thread*	-40,0%	-38,6%	-33,9%	-36,8%	-9,6%

Tabela 4.6 Diferença percentual relativa entre as tabelas 4.5 e 4.4

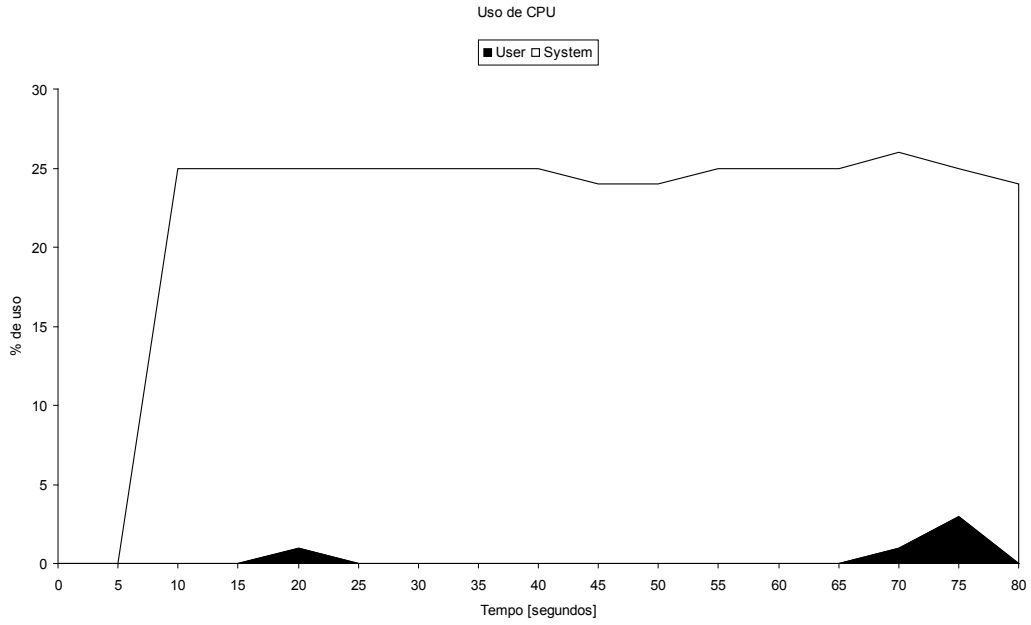

Figura 4.10 Utilização de CPU com os testes da seção 4.1.1.3

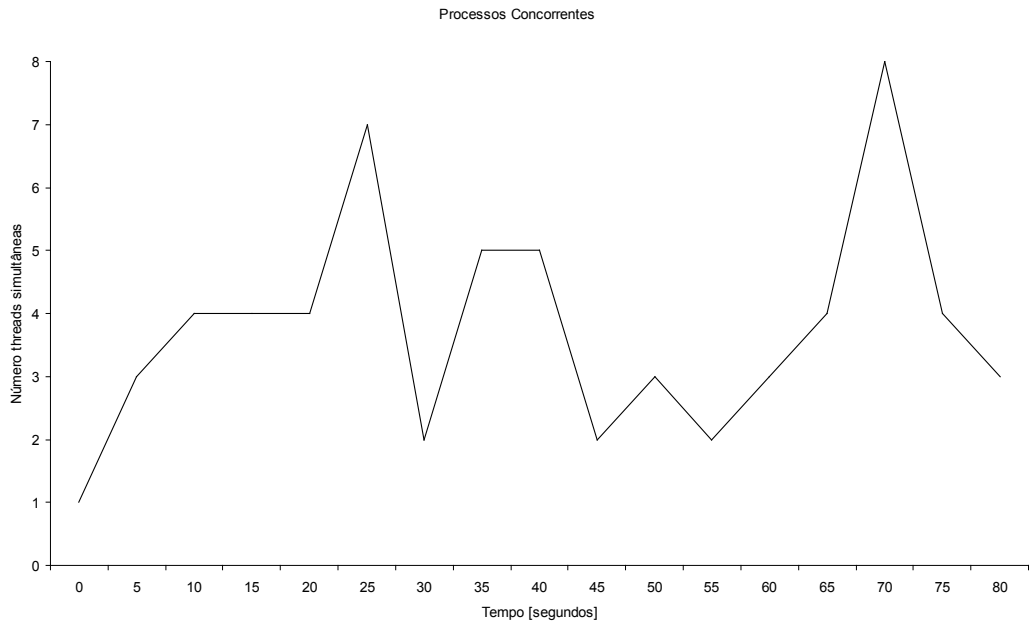

Figura 4.11 Concorrência dos processos nos testes da seção 4.1.1.3

4.1.2 Degradação ao gravar no buffer circular

Com o objetivo de avaliar a degradação de desempenho causada pela monitoração dos eventos que efetivamente realizam E/S de rede, devemos mensurar o desempenho da identificação do tipo de descritor de arquivo e da operação de escrita no buffer circular, para tal, foi desenvolvido um *script*, similar ao da seção 4.1.1, porém, com arquitetura cliente-servidor.

Para este teste, de modo a não medir eventos em duplicidade, apenas o servidor será monitorado, enquanto um nó, diferente daquele onde o servidor está sendo executado, será responsável por realizar eventos de escrita neste. O servidor deverá ler tais eventos, usando blocos de tamanho definido pelo usuário, e irá desempenhar as seguintes atividades:

- Incluir-se automaticamente na monitoração.
- Aguardar por conexões em uma porta definida pelo usuário.
- Instanciar a quantidade de processos filhos que foram definidos pelo usuário.
- Cada processo filho será responsável por gerir apenas uma conexão.
- Ao término da conexão, o processo que a atendeu finalizará e um novo processo filho será instanciado. Este irá aguardar nova conexão.
- Ao concluir um número definido pelo usuário de regenerações de processo filhos, o servidor finalizará, encerrando o serviço.
- Excluir-se automaticamente da monitoração ao término da execução

O cliente irá desempenhar as seguintes atividades:

- Criar uma barreira de sincronização
- Criar o número de *threads* definido pelo usuário.
- As *threads* se conectarão ao servidor definido pelo usuário, negociando qual a quantidade de bytes será enviada e o bloco de leitura que deve ser utilizado.
- Ao término da negociação com o servidor, todas as *threads* devem aguardar a liberação da barreira.
- Aguardar a sinalização do usuário pela liberação da barreira.
- Contabilizar o tempo gasto pelas *threads*

De maneira análoga a seção 4.1.1, os blocos selecionados foram de 1, 10, 100, 1000 e 8192 bytes para a transmissão de 10.000.000 bytes, que equivalem a 10.000.000, 1.000.000, 100.000, 10.000 e a aproximadamente 1220 eventos de leitura. Mas desta vez, o objetivo final será utilizar o caminho completo da monitoração, o que corresponde a armazenar dados no buffer circular. Nas seções subsequentes serão apresentados os resultados apenas, a discussão sobre eles virá na seção 4.1.2.4.

4.1.2.1
Desempenho de E/S de rede referencial

Os testes conduzidos por esta sessão são similares aos testes conduzidos na seção 4.1.1.1, porém, medindo o desempenho das operações de E/S de rede. Os tempos adquiridos por este teste correspondem ao melhor resultado que o cluster utilizado pelo teste pode desempenhar. A figura 4.12 representa cem amostras do tempo transcorrido, em segundos, entre o início e o fim da execução do *script*, sendo vinte amostras para cada bloco escolhido, usando 1, 2, 4 ou 8 *threads*. Os valores médios de tempo são apresentados na tabela 4.7.

	1 byte	10 bytes	100 bytes	1000 bytes	8192 bytes
8 *threads*	7,140s	0,778s	0,689s	0,684s	0,686s
4 *threads*	7,011s	0,754s	0,348s	0,345s	0,345s
2 *threads*	7,083s	0,740s	0,174s	0,173s	0,174s
1 *thread*	7,008s	0,736s	0,098s	0,090s	0,088s

Tabela 4.7 Médias de desempenho de rede obtidas sem carga extra

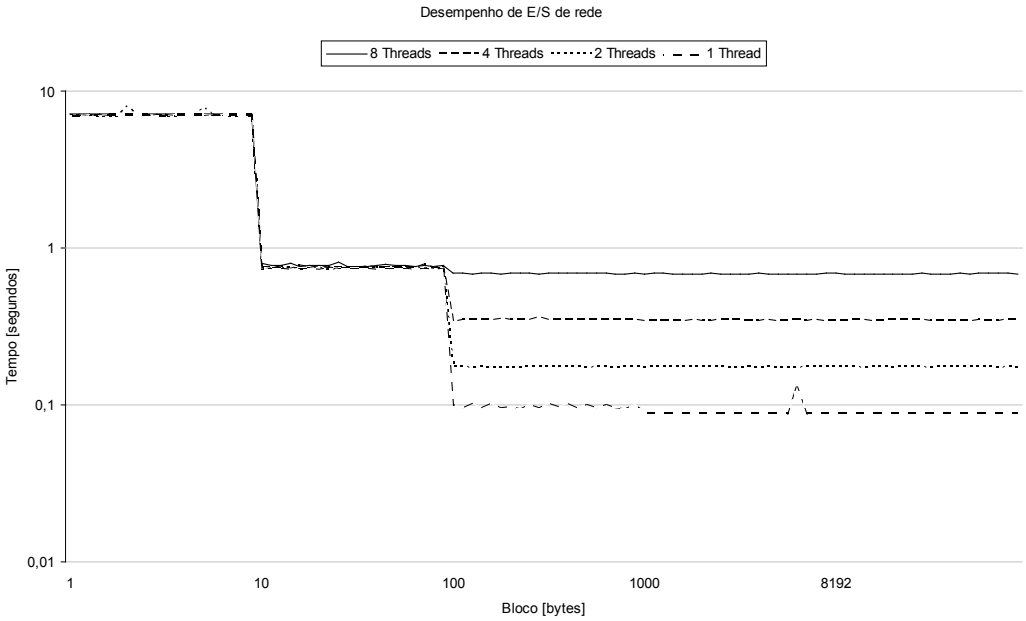

Figura 4.12 Desempenho de E/S de rede sem carga extra

4.1.2.2
Desempenho de E/S de rede com a monitoração ativada

Os testes conduzidos por esta seção são similares aos testes conduzidos na seção 4.1.1.2, porém, medindo o desempenho das operações de E/S de rede. A figura 4.13 representa cem amostras do tempo transcorrido, em segundos, entre o início e o fim da execução do *script*, sendo vinte amostras para cada bloco escolhido, usando 1, 2, 4 ou 8 *threads*. Os valores médios de tempo são apresentados na tabela 4.8.

	1 byte	10 bytes	100 bytes	1000 bytes	8192 bytes
8 *threads*	72,159s	7,330s	0,781s	0,688s	0,685s
4 *threads*	26,957s	2,765s	0,351s	0,345s	0,345s
2 *threads*	9,849s	1,091s	0,182s	0,174s	0,173s
1 *thread*	7,531s	0,788s	0,098s	0,088s	0,088s

Tabela 4.8 Médias de desempenho de rede obtidas com a monitoração habilitada

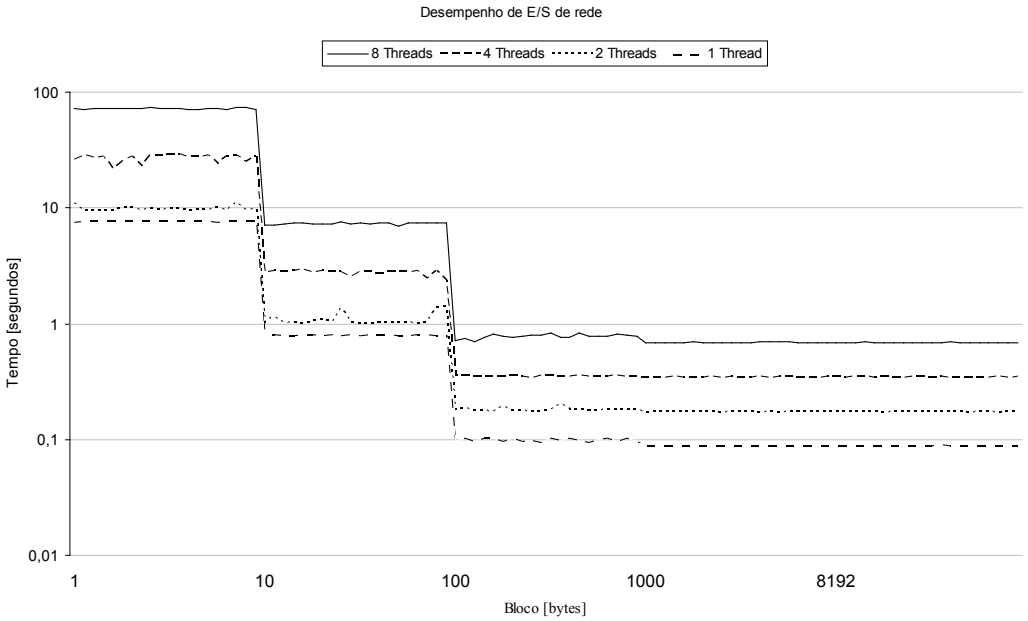

Figura 4.13 Desempenho de E/S de rede com a monitoração habilitada

4.1.2.3
Desempenho de E/S de rede do processo em monitoração

Os testes conduzidos por esta seção são similares aos testes conduzidos na seção 4.1.1.3, porém, medido o desempenho das operações de E/S de rede. A figura 4.14 representa cem amostras do tempo transcorrido, em segundos, entre o início e o fim da execução do *script*, sendo vinte amostras para cada bloco escolhido, usando 1, 2, 4 ou 8 *threads*. Os valores médios de tempo são apresentados na tabela 4.9.

	1 byte	10 bytes	100 bytes	1000 bytes	8192 bytes
8 *threads*	147,829s	14,748s	1,645s	0,686s	0,687s
4 *threads*	84,587s	9,251s	0,919s	0,344s	0,344s
2 *threads*	55,433s	5,852s	0,593s	0,173s	0,174s
1 *thread*	33,745s	3,414s	0,374s	0,088s	0,088s

Tabela 4.9 Médias de desempenho do processo monitorado

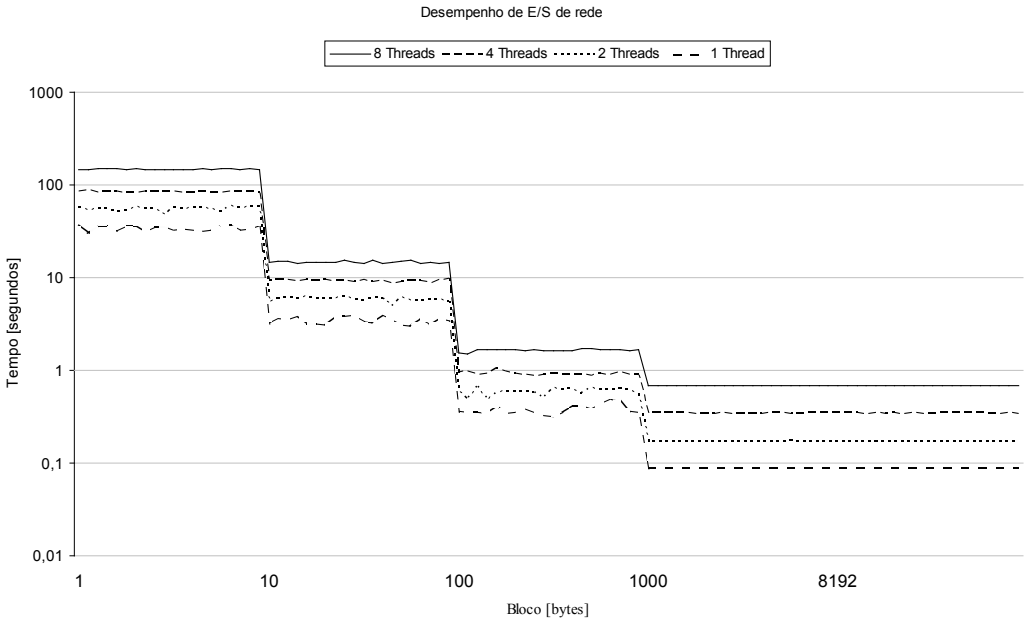

Figura 4.14 Desempenho de E/S de rede do processo monitorado

4.1.2.4
Avaliação dos resultados

Analisando as tabelas 4.10 e 4.11, observamos que os resultados apresentados para os testes de E/S de rede foram mais coerentes, na medida em que o incremento de tempo para o caminho parcial foi menor que para o caminho completo, em comparação aos resultados que foram apresentados na seção 4.1.1.

	1 byte	10 bytes	100 bytes	1000 bytes	8192 bytes
8 *threads*	910,6%	842,5%	13,3%	0,5%	-0,2%
4 *threads*	284,5%	266,8%	0,9%	0,0%	0,0%
2 *threads*	39,0%	47,4%	4,5%	0,1%	-0,1%
1 *thread*	7,5%	7,1%	0,6%	-2,2%	0,0%

Tabela 4.10 Incremento de tempo de E/S percentual da seção 4.1.2.2 / 4.1.2.1

	1 byte	10 bytes	100 bytes	1000 bytes	8192 bytes
8 *threads*	1970,4%	1796,1%	138,7%	0,3%	0,2%
4 *threads*	1106,5%	1127,0%	164,1%	-0,2%	-0,2%
2 *threads*	682,6%	690,9%	240,9%	0,0%	-0,1%
1 *thread*	381,5%	363,7%	282,2%	-2,1%	0,0%

Tabela 4.11 Incremento de tempo de E/S percentual da seção 4.1.2.3 / 4.1.2.1

Para justificar tal desempenho, e também melhor fundamentar a diferença observada na seção 4.1.1.4, podemos comparar as figuras 4.15 e 4.16, onde são presentados os gráficos de utilização de CPU ao longo dos testes das seções 4.1.2.2 e

4.1.2.3 para o pior caso (8 núcleos, blocos de 1 byte), nota-se que a média de utilização de CPU é aproximadamente 20% em ambos os gráficos, bem superior a média de aproximadamente 10% observada na figura 4.8 da seção 4.1.1.4. Provavelmente, a latência da rede influenciou a menor concorrência dos processos, impedindo que eles entrassem simultaneamente na região crítica e, consequentemente, que fossem reescalonados.

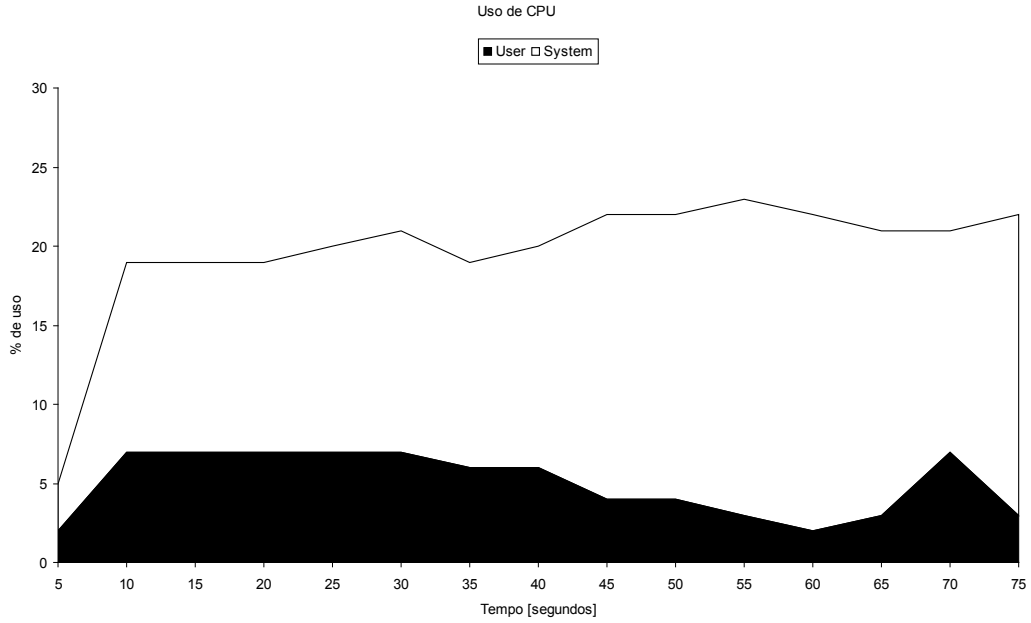

Figura 4.15 Utilização de CPU com o teste da seção 4.1.2.2

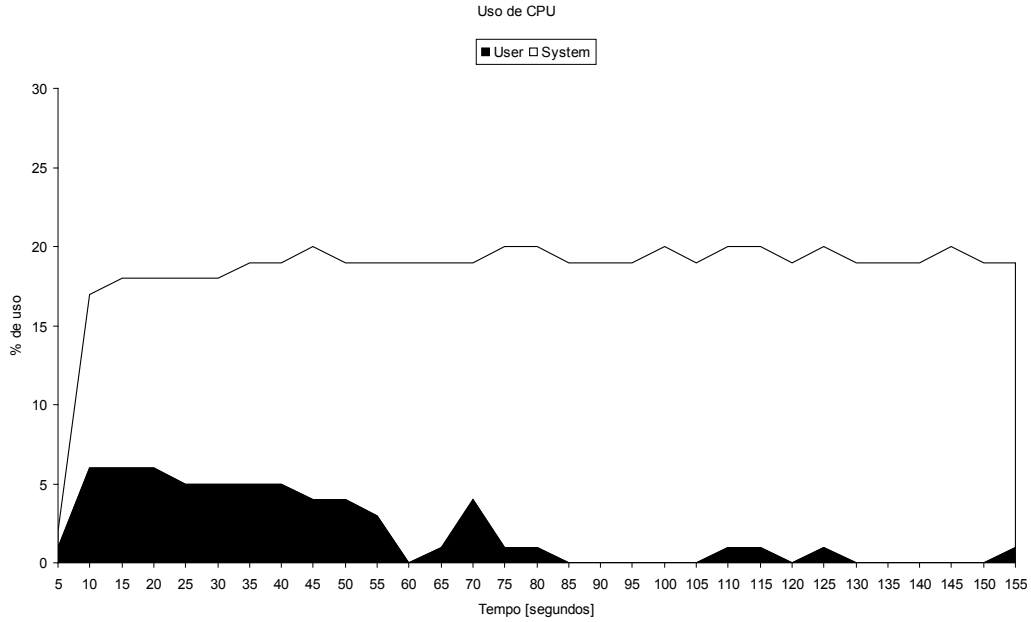

Figura 4.16 Utilização de CPU com o teste da seção 4.1.2.3

Um dos motivos para a diferença na utilização de CPU pelo usuário (tempo de *user*) entre a seção 4.1.2.2 e a seção 4.1.2.3 é devido ao caminho parcial evitar uma das regiões críticas do código, reduzindo a probabilidade de reescalonando, resultando em menor tempo de execução. Esta menor probabilidade permitiu uma média de aproximadamente quatro processos concorrentes para o caminho parcial, como observado na figura 4.17, enquanto o caminho completo, com maior probabilidade de resscalonamento, obteve uma média de aproximadamente três processos concorrentes, como observado na figura 4.18. Assim, podemos explicar o menor desempenho do caminho completo, principalmente, devido à gravação no buffer circular, por depender da entrada em mais uma região crítica para tal.

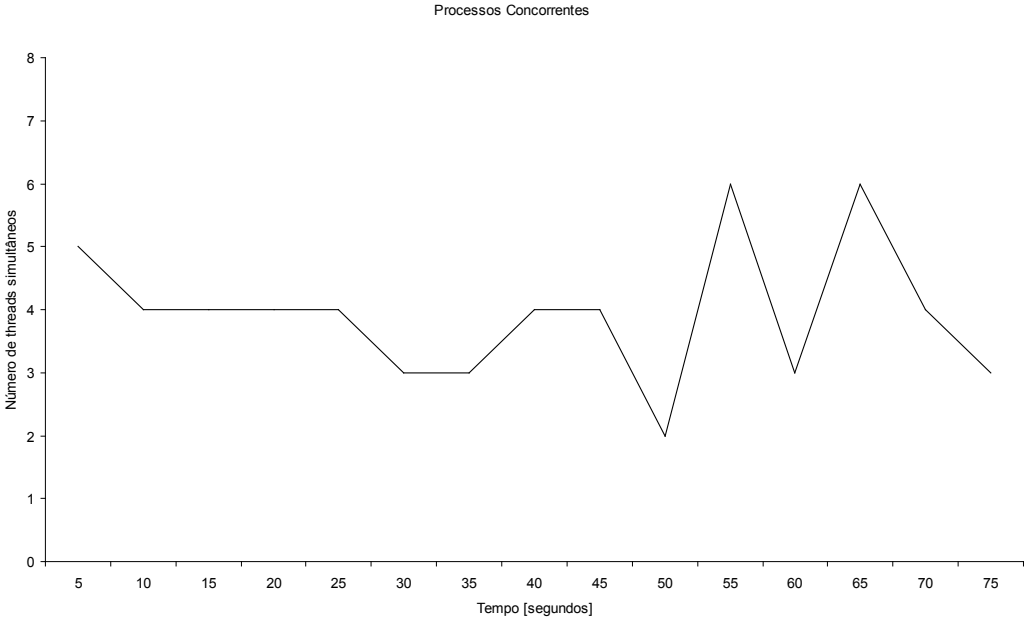

Figura 4.17 Concorrência dos processos do teste da seção 4.1.2.2

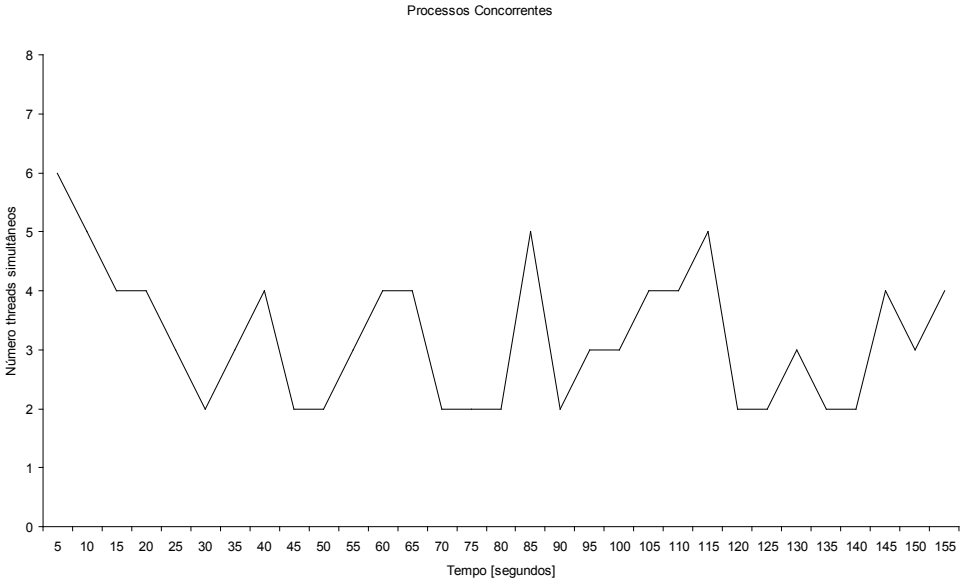

Figura 4.18 Concorrência dos processos do teste da seção 4.1.2.3

Como também observado na seção 4.1.1.4, a tabela 4.12 apresenta o mesmo gradiente de redução de incremento de tempo, onde o menor incremento é observado quanto maior o tamanho do bloco e menor concorrência. Os gradientes apresentados por esta seção e pela seção 4.1.1.4 representam um forte indício de que as regiões críticas, principalmente a que protege a tabela de dispersão, são as principais responsáveis pela degradação de desempenho observada nos testes.

	1 byte	10 bytes	100 bytes	1000 bytes	8192 bytes
8 *threads*	104,9%	101,2%	110,6%	-0,3%	0,4%
4 *threads*	234,8%	234,5%	161,7%	-0,2%	-0,3%
2 *threads*	462,8%	436,6%	226,2%	-0,1%	0,0%
1 *thread*	348,1%	333,0%	280,0%	0,1%	0,0%

Tabela 4.12 Diferença percentual relativa entre as tabelas 4.9 e 4.8

4.2 Avaliação do desempenho da interface núcleo-usuário

Os testes de desempenho da interface núcleo-usuário foram conduzidos utilizando replicações de uma transação GIOP, contendo uma requisição e uma resposta deste protocolo, que ocupam 588 bytes no total. Conforme indicado na seção 3.3, as informações adquiridas do núcleo pela interface são convertidas em arquivos no formato *libpcap*. O desempenho da conversão promovida por esta interface pode ser observado na tabela 4.13 e na figura 4.19. Com elas, podemos deduzir que o algoritmo utilizado pela interface consome tempo linear, de aproximadamente 77 microssegundos, para cada registro processado na máquina de teste. Assim, como cada evento, segundo a tabela 4.9, pode consumir cerca de 9 microssegundos, existe a possibilidade de que esta interface se torne um ponto de retenção de desempenho caso haja demanda considerável de E/S de rede pelos processos monitorados.

Registros	100	1.000	10.000	100.000	1.000.000
	0,008s	0,082s	0,733s	7,502s	73,721s

Tabela 4.13 Médias de desempenho da interface

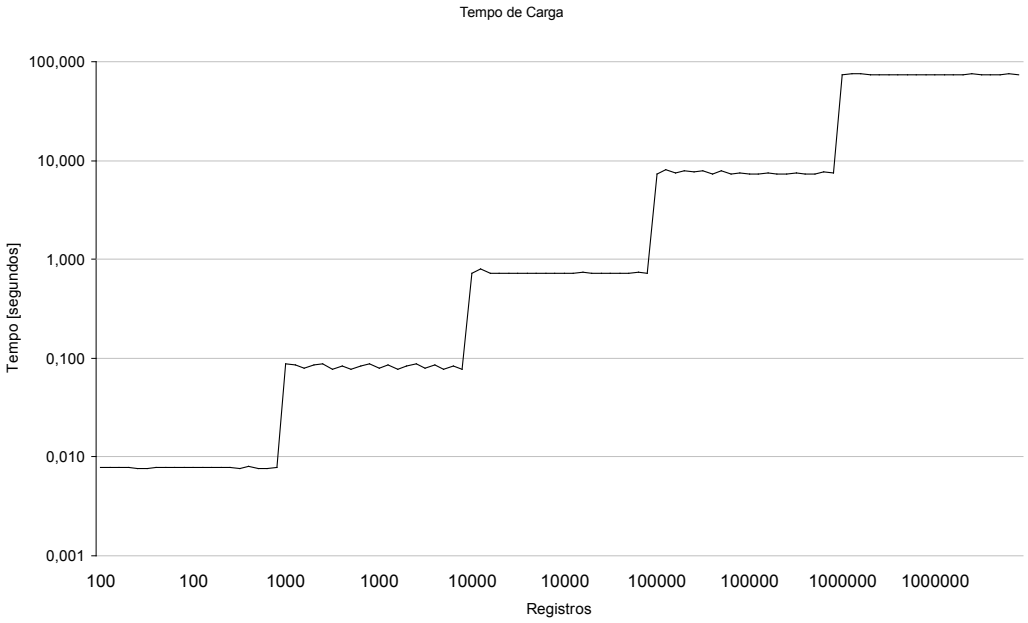

Figura 4.19 Desempenho da transformação núcleo → usuário

4.3 Avaliação do desempenho da interface GUI

Apesar de ser possível, mas não necessário e nem recomendável, executar a interface GUI na mesma máquina onde está sendo realizada a monitoração, é importante conhecer as limitações da ferramenta. Como anunciado na seção 3.4, o *Wireshark* foi à interface GUI escolhida para realizar a análise dos arquivos no formato *libpcap* produzidos pela interface núcleo-usuário.

O desempenho de carga dos arquivos *libpcap* pelo *Wireshark* mostrou, segundo a tabela 4.14 e a figura 4.20, degradação de desempenho exponencial, o que significa que conforme a quantidade de registros a serem carregados aumenta, o tempo de carga do arquivo aumenta exponencialmente. Dessa forma, foi constatado empiricamente, que para atingir tempos de carregamento razoáveis, é imperativo que os arquivos a serem analisados contenham em torno de 40.000 registros e que não ultrapassem mais que uma dezena de megabytes. Tais medidas garantem que a carga seja realizada em aproximadamente 1 minuto.

Registros	100	1.000	10.000	100.000	1.000.000
	0,171s	0,264s	2,551s	307,534s	44720,554s

Tabela 4.14 Médias de desempenho do *Wireshark*

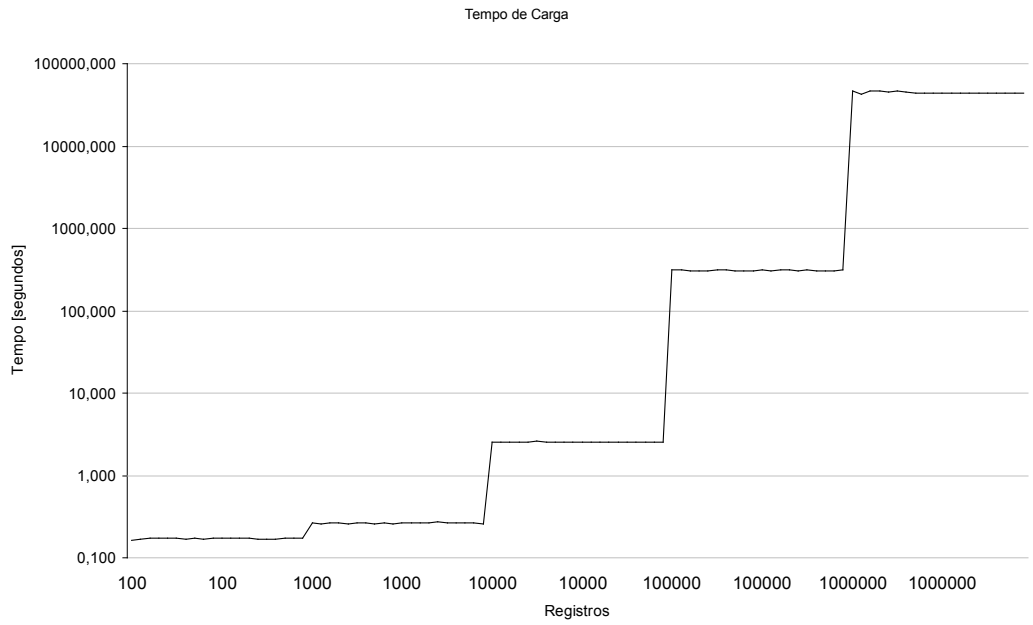

Figura 4.20 Desempenho de carga de registros do *Wireshark*

De modo a remediar o desempenho inferior para a carga arquivos com número elevado de registros, o *wireshark* conta com ferramentas que permitem fracionar arquivos *libpcap* grandes, que demandam muito tempo para o carregamento, em arquivos menores, de tempo de carga menor. E também, se necessário, há ferramentas para juntar arquivos menores de modo a formar um arquivo maior. O objetivo destas ferramentas é trabalhar com arquivos que contenham toda a informação que se deseja analisar, e que, de modo a ser confortável manipulá-los, se possa carrega-los em tempo razoável.

4.4 Avaliação funcional

O objetivo desta seção é mostrar o funcionamento da ferramenta proposta por este trabalho e indicar como preparar o ambiente para o seu uso, utilizando exemplos que simulam tarefas realizadas por aplicações reais. Por exemplo, conectar em um servidor de mensagens para enviar um e-mail, realizar transações utilizando a linguagem Java e o *middleware* CORBA através do uso de invocações dinâmicas, e do *middleware* CORBA como infraestrtutura de comunicação em um pregão eletrônico.

O uso de invocações dinâmicas tem como objetivo mostrar o funcionamento da ferramenta ainda que não exista um contrato formal sobre o formato das mensagens a serem trocadas entre cliente e servidor, pois tais mensagens só serão negociadas em tempo de execução pelo software. Tal característica requer que a ferramenta interprete o protocolo CORBA sem a utilização de tal contrato como referência, o que dificulta sua interpretação.

O uso da ferramenta em um pregão eletrônico é demostrado por dois sistemas, tal duplicidade tem como objetivo mostrar a independência da ferramenta

quanto à utilização de *middlewares* CORBA de fornecedores diferentes. O primeiro exemplo utiliza a versão embutida na linguagem Java, o segundo, mais elaborado, utiliza a versão fornecida pelo fabricante Micro Focus conhecido como Orbacus [51], e utiliza a arquitetura de envio de notificações disponível neste *middleware* para atualizar os clientes sobre os lances ofertados pelos demais.

4.4.1 Preparação do ambiente para testes

Para que o ambiente esteja preparado para realizar a monitoração dos softwares a serem testados, devemos preparar as camadas de monitoração, transformação e análise como descrito na seção 3.5. Para preparar a camada de monitoração, é necessário primeiro carregar o módulo de interceptação, conforme descrito na figura 4.21.

```
# insmod lkm/intercept.ko
# dmesg | tail -n 2

interceptor[main.c:init:55]: Kernel interceptor compiled on Apr  8 2013@08:55:27
interceptor[cdev.c:createdevfsfile:134]: Registered character device major 251 minor 0
```

Figura 4.21 Carregamento do módulo de interceptação de chamadas de sistema

Em seguida, é necessário criar o vínculo de comunicação entre a camada de transformação e monitoração, conforme descrito na figura 4.22. Os números do *major* e *minor* são informados pelo módulo com o comando *dmesg*[9], conforme mostrado na figura 4.21.

```
# mknod /dev/interceptor c 251 0
```

Figura 4.22 Criação do vínculo de comunicação núcleo-usuário

Uma vez carregado o módulo e criado o vínculo de comunicação, é necessário iniciar a interface de transformação, para que as informações capturadas sejam transferidas do núcleo para um arquivo. A interface pode ser iniciada conforme apresentado na figura 4.23.

```
# perl parser.pl /dev/interceptor
```

Figura 4.23 Instanciação da camada núcleo-usuário

Ao carregar o módulo, preparar o vínculo de comunicação e iniciar a camada de transformação, a interface de gerência da monitoração está apta a ser manipulada conforme descrito na seção 3.2.3. Dessa forma, podemos informar qual o número de identificação do processo desejamos monitorar para a camada de monitoração.

[9] Comando do Linux que mostra as últimas mensagens produzidas pelo núcleo.

A qualquer momento podemos utilizar a interface GUI, descrita na seção 3.4, para visualizar o que a monitoração foi capaz de interceptar. Podemos iniciar a interface GUI conforme indicado na figura 4.24.

```
# wireshark tshark.log
```

Figura 4.24 Instanciação da interface GUI

4.4.2 Conexão a um servidor de mensagens

Este exemplo irá comparar o resultado da captura de pacotes utilizando um capturador de pacotes *(packet sniffer)* e a captura realizada utilizando a ferramenta proposta por este trabalho. Na captura obtida pelo capturador de pacotes, como mostra a figura 4.25, foi utilizado um filtro de captura, onde apenas os pacotes destinados e provindos do endereço e da porta utilizada pelo servidor de mensagens foram capturados. Nela podemos verificar que há informações sobre o protocolo TCP que não são necessárias para análise da troca de mensagens entre o cliente de envio e o servidor de recepção de mensagens.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.219.128	10.29.237.57	TCP	74	51898 > 25 [SYN] Seq=0 Win=5840 Len=0 MSS=1460 SACK_PERM=1 TSval=22764070 TSec
2	0.000910	10.29.237.57	192.168.219.128	TCP	60	25 > 51898 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
3	0.000923	192.168.219.128	10.29.237.57	TCP	54	51898 > 25 [ACK] Seq=1 Ack=1 Win=5840 Len=0
4	0.007024	10.29.237.57	192.168.219.128	SMTP	154	S: 220 monitor2.un-rio.petrobras.com.br ESMTP Sendmail 8.13.8/8.11.6; Thu, 28 F
5	0.007052	192.168.219.128	10.29.237.57	TCP	54	51898 > 25 [ACK] Seq=1 Ack=101 Win=5840 Len=0
6	0.007114	192.168.219.128	10.29.237.57	SMTP	60	C: QUIT
7	0.007206	10.29.237.57	192.168.219.128	TCP	60	25 > 51898 [ACK] Seq=101 Ack=7 Win=64240 Len=0
8	0.008040	10.29.237.57	192.168.219.128	SMTP	117	S: 221 2.0.0 monitor2.un-rio.petrobras.com.br closing connection
9	0.008047	10.29.237.57	192.168.219.128	TCP	60	25 > 51898 [FIN, PSH, ACK] Seq=164 Ack=7 Win=64240 Len=0

Figura 4.25 Captura utilizando o *Wireshark*

Já a captura de mensagens realizada pela ferramenta, como mostra a figura 4.26, por capturar apenas os eventos de E/S, não contem todas as informações sobre o protocolo TCP. Porém, a omissão dessas informações simplifica a análise da troca de mensagens entre cliente e servidor, além de economizar a utilização de filtros de captura, porque, em substituição ao filtro, bastou selecionar o identificador do processo cliente.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.219.128	10.29.237.57	SMTP	152	C: 220 monitor2.un-rio.petrobras.com.br ESMTP Sendmail 8.13.8/8.11.6; Thu, 28
2	0.001205	10.29.237.57	192.168.219.128	SMTP	58	S: QUIT
3	0.001990	192.168.219.128	10.29.237.57	SMTP	115	C: 221 2.0.0 monitor2.un-rio.petrobras.com.br closing connection

Figura 4.26 Captura utilizando a ferramenta

Nas figuras 4.27 e 4.28, utilizou-se a opção do menu rápido “Follow TCP Stream” para certificar que o conteúdo das mensagens obtidas pelos dois métodos foram iguais.

Figura 4.27 Stream do sniffer

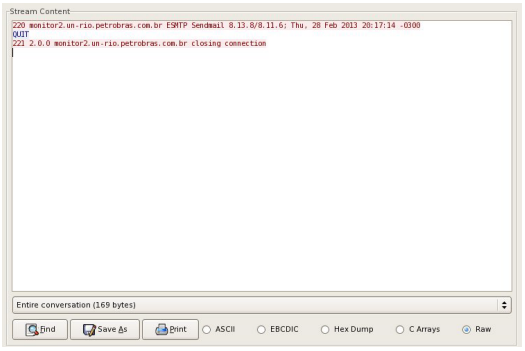

Figura 4.28 Stream da ferramenta

A captura realizada pela ferramenta mostrou produzir menos pacotes que o capturador de pacotes, porém, as informações contidas na captura realizada pela ferramenta são iguais às capturadas pelo capturador de pacotes embutido no *wireshark*.

4.4.3 Troca de mensagens CORBA utilizando DII/DSI

Este exemplo tem como finalidade mostrar que a ferramenta é capaz de analisar a comunicação entre componentes CORBA mesmo quando não há uma interface IDL descrevendo como será estruturada a troca de mensagens entre os componentes. Em outras palavras, quando o software realiza a comunicação entre componentes através da utilização das interfaces DSI *(Dynamic Skeleton Interface)* e DII *(Dynamic Invocation Interface)* existentes no *middleware*. Esta interface permite que o software realize e receba requisições CORBA definidas em tempo de execução.

Neste exemplo, mesmo não havendo necessidade para tal, para efeitos de demonstração do potencial da ferramenta, foi forjada uma IDL equivalente à estrutura de comunicação entre os componentes do exemplo. Esta foi adaptada para funcionar como extensão e carregada no *wireshark*, utilizando a técnica descrita na seção 3.4. O resultado deste teste pode ser comprovado na figura 4.29, onde o pacote é corretamente decodificado e exibido na árvore de decodificação de protocolos.

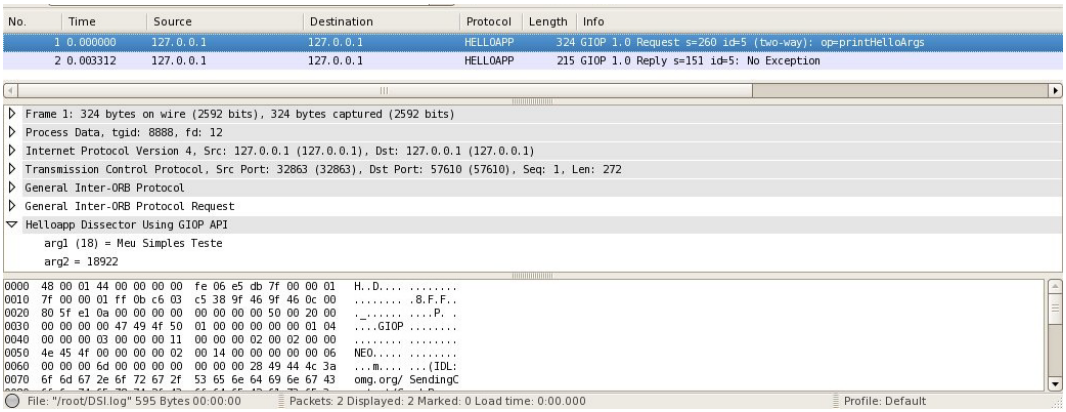

Figura 4.29 Folha obtida com a carga da IDL forjada

Dessa forma, com a IDL carregada no analisador de protocolos, podemos realizar pesquisas nos pacotes capturados utilizando, como fonte de dados, características contidas dentro do pacote GIOP antes inacessíveis, como mostrado na figura 4.30. Nesta figura, o campo representado por *arg2*, que corresponde a um inteiro, é utilizado como fonte de dados para um filtro condicional, onde este argumento deve ser maior ou igual a dois.

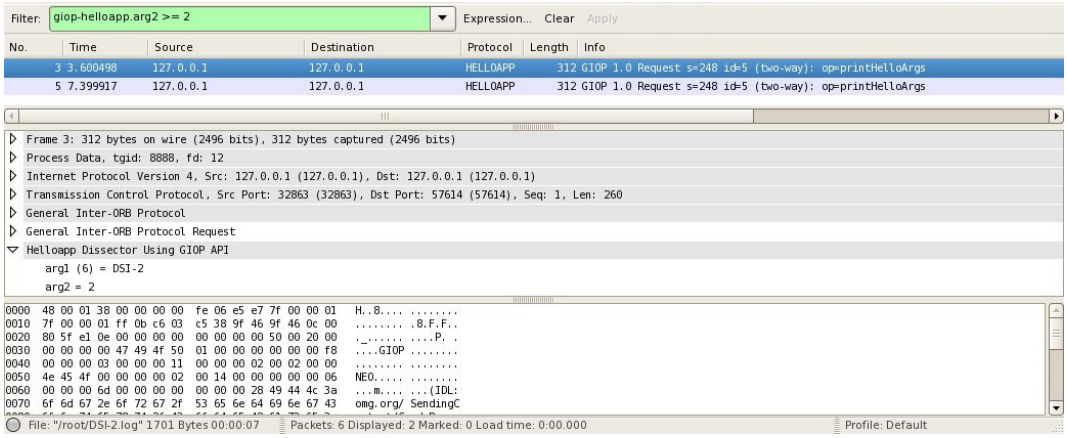

Figura 4.30 Pesquisa utilizando características do pacote GIOP

4.4.4 Gerenciador de pregão eletrônico

O software de gerenciamento de pregão eletrônico é um projeto de pesquisa do professor Murshed, da Upper Iowa University. O software, de sua autoria, foi obtido em [52]. Seu diagrama de funcionamento pode ser avaliado na figura 4.31, onde o *Auction Server* é o agente centralizador de requisições. Além do servidor, existem dois perfis de clientes, os compradores *(bidders)*, que realizam ofertas, e o vendedor *(seller)*, que cadastra e vende itens no pregão.

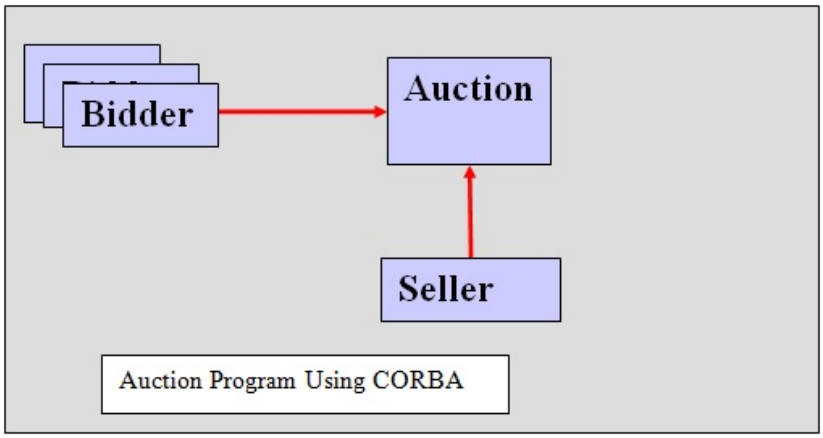

Figura 4.31 Diagrama do software de gerenciamento de pregões

A topologia de componentes utilizada pelo software deixa claro que, de forma a observar tudo o que está acontecendo no software, basta capturar as transações

intermediadas pelo *Auction Server*. Dessa forma, a ferramenta foi habilitada no servidor onde foi executado o *Auction Server*. O resultado da captura de pacotes pode ser avaliado na figura 4.32.

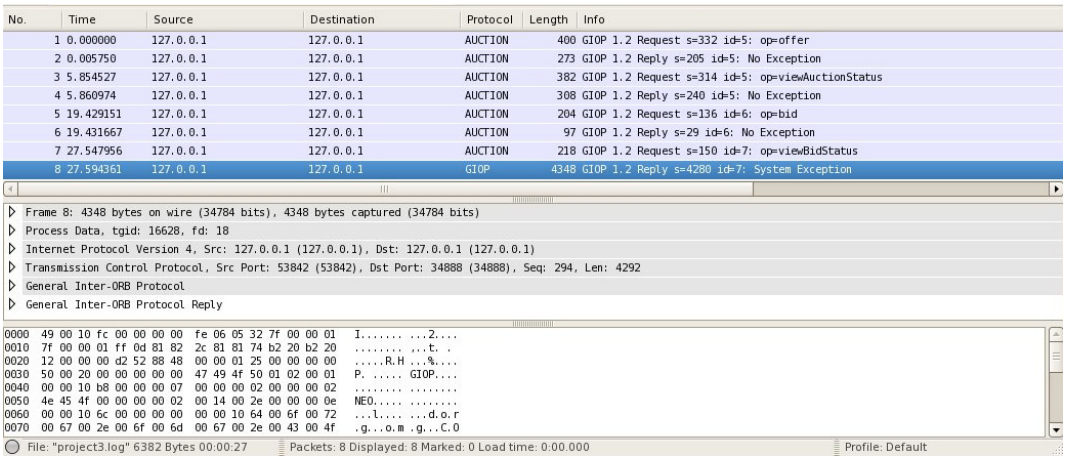

Figura 4.32 Resultado da monitoração do *Auction Server*

Durante a avaliação do software, foi forçada uma condição de erro durante a interpretação de uma sequência de caracteres. Esta condição culminou em um defeito inesperado no software, defeito este capturado pela monitoração. Entretanto, o erro ocorrido não foi mostrado ao usuário. O *Auction Server*, como forma de depuração, enviou um despejo de pilha *(stack dump)* para o cliente que invocou o método de forma incorreta, porém, o cliente não mostrou o referido erro para o usuário, demonstrando um dos usos possíveis da ferramenta.

A figura 4.32 mostra o erro mencionado como um pacote GIOP contendo uma exceção de sistema *(System Exception)*. Dentro deste pacote podemos encontrar o despejo da pilha e o seu conteúdo pode ser avaliado na figura 4.33. Dessa forma, observando o pacote listado na linha 7 da janela de resumo, exibido na figura 4.32, já temos informações suficientes para reproduzir este defeito. Reenviando ao servidor o conteúdo transportado por este pacote será possível reproduzir o defeito de forma mais fácil, facilitado seu processo de correção.

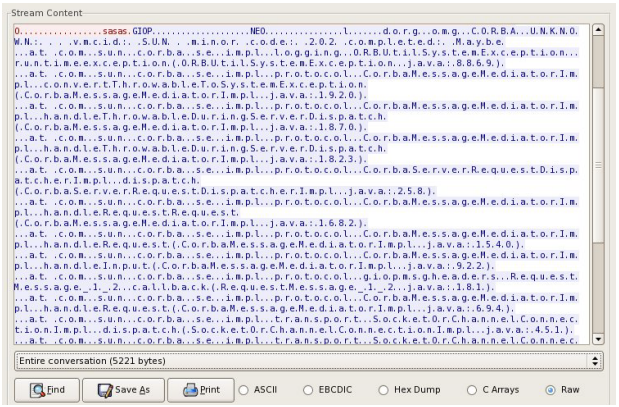

Figura 4.33 Conteúdo da exceção de sistema encapsulada no pacote GIOP

4.4.5
Gerenciador de pregão eletrônico com notificações

Para exemplificar o uso dos serviços de nomes (CosNaming) e de notificações (CosNotify) de CORBA, [53] disponibilizou um software de pregão eletrônico com notificações para, através do uso de bibliotecas que simplificam seu uso, mostrar a utilização dos referidos serviços. A interface do software mostra uma janela com uma lista de itens a serem vendidos, o item que está sendo leiloado, o valor do lance atual, as mensagens enviadas pelo leiloeiro aos licitantes e uma caixa de diálogo para informar o valor do lance que se deseja efetuar, conforme a figura 4.34.

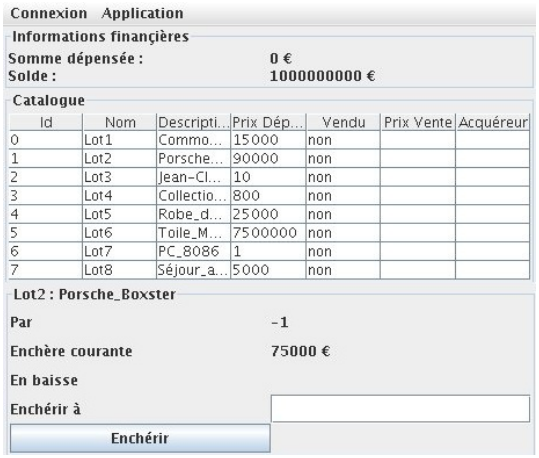

Figura 4.34 Interface do software de pregão eletrônico com notificações

A topologia de componentes utilizada por esse software é similar ao do software testado na seção 4.4.4, onde existe um componente centralizador de requisições. Entretanto, as tentativas, em um primeiro momento, de monitorar o software foram infrutíferas. Análises posteriores mostraram que o middleware CORBA utilizado pelo software, o Orbacus, diferente do fornecido pelo fabricante da linguagem Java, não utiliza as chamadas de sistema *read* e *write*, e sim as chamadas *recv* e *send*.

Apesar de serem equivalentes, *recv* e *send* são mais apropriadas para utilização por *sockets* conectados, como os utilizados na maioria das comunicações via rede, por permitir, em alguns casos, um melhor ajuste das propriedades de comunicação destes. Mesmo equivalentes, por se localizarem em posições diferentes no vetor de chamadas, correspondem a chamadas de sistemas diferentes das interceptadas pela ferramenta.

Para contornar esta situação, poderíamos alterar o código carregado no núcleo de modo a incluir na monitoração as chamadas *recv* e *send* ou injetar uma biblioteca dinâmica capaz de transformar as chamadas *recv* em *read* e *send* em *write*. Pela simplicidade, a última opção foi à escolhida. Mesmo com a alteração promovida pela injeção da biblioteca, não houve prejuízo no funcionamento do software, dessa forma, o resultado da monitoração pode ser avaliado na figura 4.35.

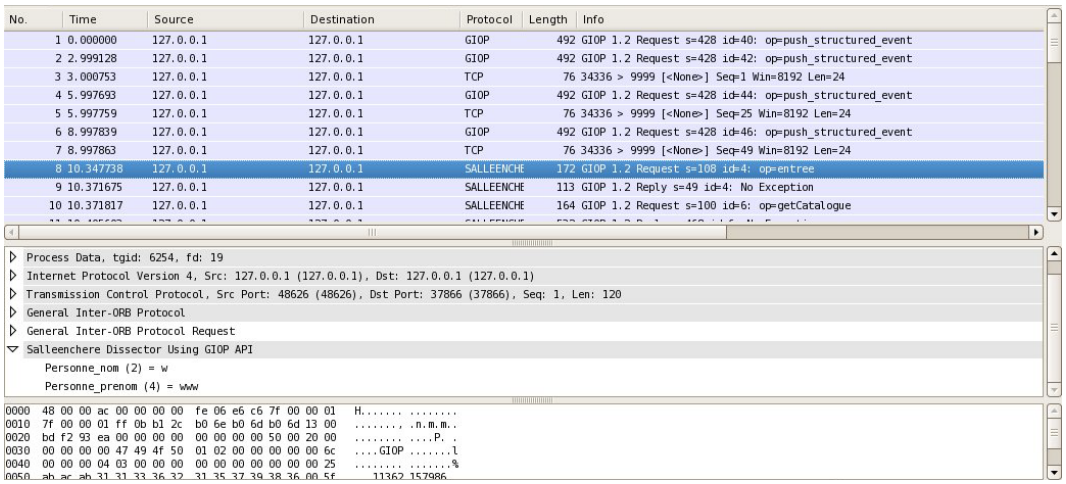

Figura 4.35 Monitoração do software de pregão eletrônico com notificações

Este exemplo também serviu para mostrar como o *wireshark* trata mensagens segmentadas, pois, ao receber uma requisição, o *middleware*, previamente, não dispõe do tamanho integral da requisição. Por este motivo, este é obrigado a ler uma pequena[10] quantidade de bytes, equivalente ao tamanho do cabeçalho GIOP. Dentro deste cabeçalho, há a informação do tamanho completo da requisição. Só assim o *middleware* é capaz de complementar a leitura anterior, completando o restante da requisição, sendo assim capaz de formar a requisição completa.

Logo, a leitura de tal requisição gera dois eventos para o sistema operacional, um fixo, para a leitura do cabeçalho GIOP, e um segundo evento, na sequência, para o restante da requisição. As linhas 3 e 4 da janela de resumo exibida na figura 4.35 demonstram tal comportamento, na primeira leitura, o analisador identifica o pacote como sendo um pacote TCP genérico, e, em seguida, na leitura seguinte, faz a desfragmentação do pacote, o reconhecendo assim como um pacote GIOP. No caso apresentado, a IDL do software já tinha sido carregada e a estrutura da requisição pôde ser completamente decodificada.

Apesar de serem menos frequentes que os casos onde capturadores de pacotes são utilizados, tais linhas adicionais podem prejudicar a compreensão da troca de mensagens entre os componentes, por esse motivo, é possível aplicar um filtro que ignore os fragmentos TCP observados, como pode ser avaliado na figura 4.36.

[10] Para o Orbacus, este valor corresponde a 76 bytes.

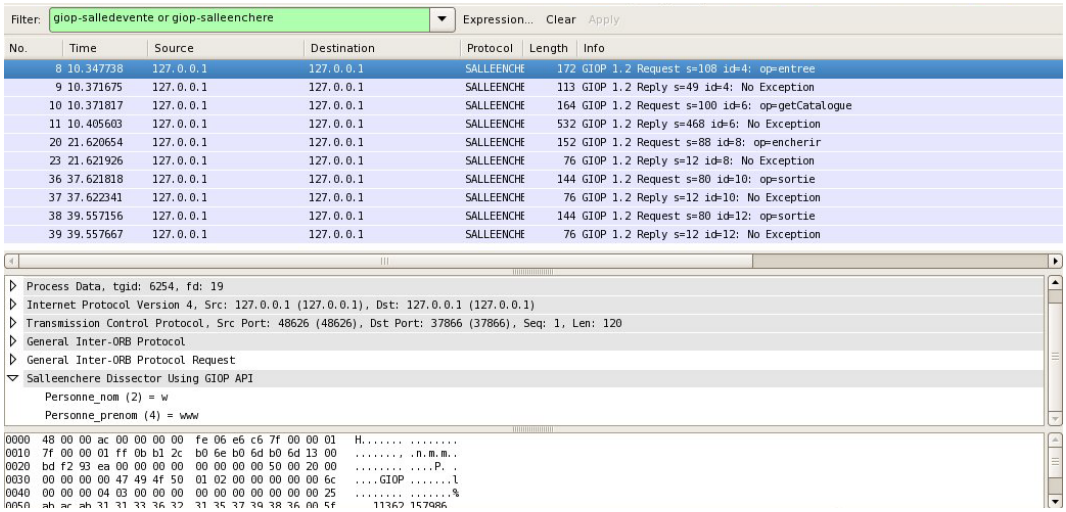

Figura 4.36 Aplicação de filtros para remoção das entradas TCP

4.5 Considerações finais

Funcionalmente, resumindo suas características, a ferramenta elaborada por este trabalho possui os mesmos recursos disponíveis nos analisadores de protocolos. Porém, apresenta uma ferramenta de captura de pacotes mais eficiente no sentido de restringir a captura do tráfego de rede a apenas aos processos selecionados pelo usuário. Dessa forma, se não for possível identificar, de antemão, a porta de comunicação quer será utilizada pelo software devido a este atender em portas designadas dinamicamente, a captura seria possível utilizando à proposta deste trabalho, entretanto, não seria possível realizar esta mesma captura utilizando capturadores comuns, por dependerem do conhecimento prévio de qual porta devem capturar.

Entretanto, como foi indicado na seção 4.1, foi notável o baixo rendimento da ferramenta em situações de elevada concorrência. Porém, é da região crítica, por estar localizada no ponto de entrada das funções de desvio, que surge a hipótese que melhor explica a baixa quantidade de processos concorrentes e os diferentes perfis de uso observados para os acessos a arquivos locais e a *sockets* de rede. Esta região crítica, por reescalonar os processos que a aguardam e obrigar a todos os processos que a atravessem, tem as todas às características que a justificam, como melhor hipótese, ser a responsável pela degradação de desempenho observada pela ferramenta.

É possível que, ao invés de reescalonar o processo, se a região crítica realizasse uma espera ocupada *(busy wait)*, seria possível perceber sensível melhora de desempenho. Neste caso, com a utilização da espera ocupada, o processo não haveria de esperar por um novo intervalo de tempo[11] do escalonador de processos para conseguir prosseguir com a realização do evento de E/S.

[11] Para o Linux, o intervalo de tempo do escalonador é 1/100 segundos. [54]

Ainda que a espera ocupada seja de uso mais fácil, sua utilização não corresponde a uma boa solução para este tipo de problema. Outra solução possível seria a troca desta proteção por um mecanismo de intertravamento estilo leitores e escritores. Neste tipo de mecanismo, os processos de leitura podem ser concorrentes, mas caso um processo de escrita solicite a trava, este irá aguardar o término da utilização da trava por todos os processos de leitura para, só assim, conseguir adquiri-la. Em seguida, todos os processos subsequentes devem aguardar a trava de escrita finalizar para poderem readquirir a trava de leitura ou de gravação. O ganho com este tipo de intertravamento se daria por permitir maior quantidade de processos de leitores concorrentes, devida a necessidade de pesquisas à tabela de dispersão, que escritores, devido às alterações em tal tabela por conta de inclusões ou exclusões de identificadores de processos.

Ainda que mais eficiente que uma região crítica, tal mecanismo ainda requer o uso de intertravamentos. Uma solução mais complexa, porém mais eficiente, seria o uso de contadores de referência para a tabela de dispersão. Assim, sempre que houver a necessidade de alteração da tabela, uma nova cópia desta seria alocada e alterada. Para a troca, a cópia da tabela de dispersão com as alterações aplicadas seria colocada no lugar da anterior, atomicamente, assim que o contador de referências da tabela anterior atingisse o valor zero. Esta solução seria dependente de atualizações atômicas de ponteiros e de incrementos e decrementos atômicos de contadores, porém, este seria um algoritmo livre de intertravamentos.

O uso de qualquer uma das soluções acima apresentadas levaria a um melhor desempenho da ferramenta em situações de elevada concorrência. Já para as demais camadas, o desempenho delas só poderia ser melhorado se houvessem algoritmos paralelos que substituíssem os algoritmos atualmente em uso.

No caso da interface núcleo-usuário, o algoritmo que realiza o cálculo de verificação de cabeçalhos TCP é o que provavelmente tem o maior custo computacional. De forma a mitigar esse efeito, seria possível realizar tal cálculo utilizando uma segunda linha de processamento *(thread)* para, consequentemente, diminuir o tempo de processamento por pacote. Outra possibilidade seria eliminar esta camada, realizando toda a transformação realizada por esta no núcleo, reaproveitando os algoritmos de verificação de cabeçalhos por ele disponibilizados.

Já para o caso da interface GUI com o usuário, por não limitar o desempenho da monitoração, a busca por um melhor desempenho é menos crítica. Entretanto, ainda assim, há estudos sendo conduzidos pelos desenvolvedores do *wireshark* por algoritmos paralelos que possam acelerar a carga dos arquivos *libpcap*. Enquanto isso, segmentar os arquivos de captura ainda é a solução indicada para reduzir o tempo de carga dos mesmos.