Uma Implementação do Utilitário Dusers Baseada em *Threads*

Paulo Lício de Geus [*] — **paulo@dcc.unicamp.br**

Cesar Augusto de Carvalho Vannini [**] — **gutovan@dcc.unicamp.br**

Francisco Watzko Neto [**] — **watzko@dcc.unicamp.br**

Instituto de Computação
Universidade Estadual de Campinas
13081-970 Campinas, SP

Resumo

Este documento descreve a implementação do utilitário dusers (*domain rusers*) nos sistemas operacionais SunOS 4.1.3 e Solaris 2.4. O trabalho valeu-se do conceito de *threads* para atingir um maior paralelismo e performance da aplicação. Foram utilizados na implementação as bibliotecas de *threads* disponíveis nos respectivos sistemas.

Baseado nesse conceito, construímos uma ferramenta que apresenta ao mesmo tempo um processamento de controle centralizado e uma coleta de dados distribuída, evitando desta forma o uso desnecessário dos recursos de rede. Durante a implementação pudemos observar claramente as diferenças existentes quanto ao paradigma das bibliotecas de *threads*, relatando neste trabalho as nossas conclusões sobre o seu uso.

Abstract

This paper describes the implementation of the dusers utility (*domain rusers*) in the SunOS 4.1.3 and Solaris 2.4 operating systems. The work uses a multi-*thread*ed concept to achieve a greater parallelism and better performance. For this purpose we used the *threads* libraries available on those operating systems.

Based on this concept, we built a system tool that shows both a centralized control process and a distributed data collect, avoiding unnecessary use of net resources. During the implementation we could clearly observe the differences between the two *threads* libraries, reporting in this paper our conclusions about their use.

---

* - Professor Doutor do Instituto de Computação

** - Mestrando do curso de Pós-Graduação do Instituto de Computação

1. Introdução

Em uma rede de sistemas UNIX, onde várias pessoas estão logadas simultaneamente em diversas máquinas, freqüentemente necessita-se saber quem são essas pessoas. O utilitário rusers procura suprir esta necessidade, informando quem são as pessoas que estão logadas nas máquinas do segmento de rede em que foi executado, constituindo-se em uma limitação muito grande quando a rede é de maior porte. Quando se precisa saber esta informação sobre um domínio de médio ou grande porte, a única saída é a execução deste utilitário em cada segmento de rede distinto que o compõe, consumindo quantidades relevantes de tempo e processamento, sem mencionar a insatisfação do usuário.

Procurando resolver este problema, implementou-se o utilitário dusers, que tem por objetivo informar quem são as pessoas que estão logadas no domínio em que o utilitário foi executado.

2. Análise do Projeto

A primeira forma analisada para sua implementação foi baseada na existência de um *daemon* sendo executado permanentemente em todas as máquinas do domínio, como ilustra a FIGURA 1. Estes *daemons* atualizariam, com uma periodicidade pré-definida, uma base de dados permanente responsável pelo armazenamento das informações. Quando um usuário desejasse saber quem estava logado em todo o domínio, bastaria executar o utilitário dusers que leria esta base de dados central. Infelizmente, esta solução trivial apresenta alguns problemas: todas as máquinas do domínio teriam um *daemon*, levando a uma utilização permanente dos recursos computacionais existentes, gerando um uso ineficiente dos mesmos. Mesmo que o utilitário dusers não fosse utilizado por um longo período de tempo, a coleta das informações e o seu respectivo armazenamento continuariam sendo realizados. Por tratar-se de uma base de dados centralizada, a queda da máquina onde esta estivesse implicaria uma paralisação do serviço, pois nem o processo executado pelo cliente poderia ler desta, nem os *daemons* poderiam atualizá-la. Uma outra dificuldade apresentada por esta implementação seria a necessidade de monitoração do funcionamento dos *daemons* em todos os segmentos, contribuindo desta forma para o aumento da carga de trabalho do administrador. A periodicidade da atualização da base de dados seria também uma fonte de inconsistência, pois se um determinado *daemon* atualizasse a base de dados, e logo após um usuário daquele segmento encerrasse sua sessão, a base de dados ficaria desatualizada até que uma nova atualização fosse realizada.

Uma variante desta solução também foi estudada: a utilização de apenas um *daemon* por segmento de rede, que também foi descartada por apresentar praticamente as mesmos inconvenientes de sua predecessora. Esta alternativa reflete também o uso do utilitário rusers do sistema, em modo remoto.

Utilizando-se máquinas estratégicas da rede, poderia-se facilmente escrever um *shell script* com a funcionalidade desejada. Contudo, limitações originais do rusers permanecem, tais como *time-outs* inadequados, esforço extra no *script* para se criar

alguma tolerância a falhas e conhecidos *bugs* de implementação, particularmente nas versões SunOS 4.x.

Outra variação cogitada, foi a utilização de um único *daemon*, responsável pela inicialização periódica de processos remotos encarregados da coleta de informações. Esta variante também mostrou-se inviável, já que a máquina responsável pela inicialização dos diversos processos remotos poderia ficar sobrecarregada, além de possibilitar o desperdício de processamento por longos períodos de tempo, entre outras imperfeições.

Analisando-se as diversas opções anteriores e suas respectivas deficiências, tentou-se vislumbrar uma solução que, se não as resolvesse totalmente, ao menos minimizasse-as. A seguir apresentamos uma abordagem de como resolver tais deficiências:

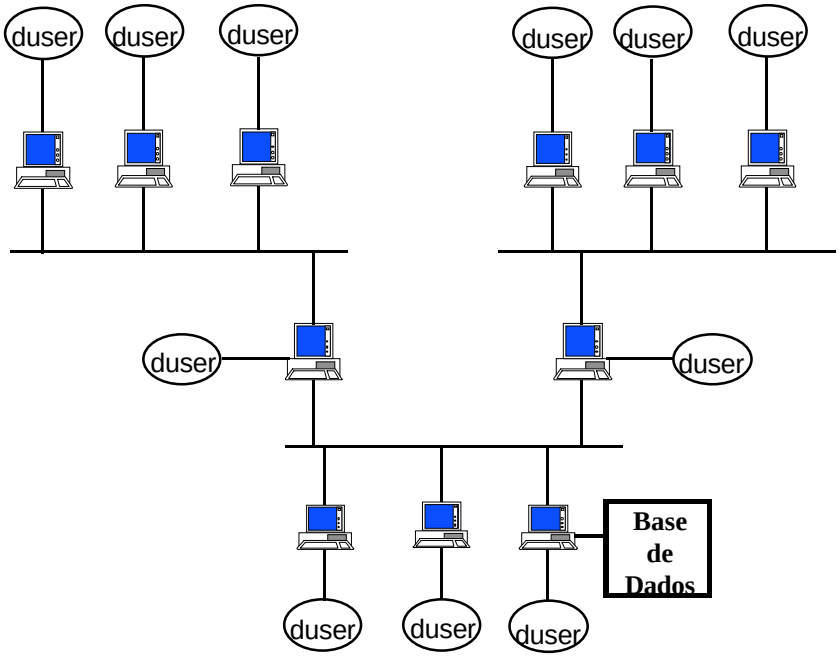

Figura 1 - Solução com *daemons* em todas as máquinas

- Inconsistência das informações contidas na base de dados. Procurando-se diminuí-la, resolveu-se implementar uma coleta de dados que fosse efetuada apenas quando da execução do utilitário dusers.

- Processamento e armazenamento desnecessários. Sua otimização implicaria em evitar o uso de *daemons*, realizando-se a coleta de dados e o seu respectivo armazenamento apenas quando da invocação do utilitário.

- Aumento da carga de trabalho do administrador de rede. A utilização de uma solução que exigisse o mínimo de configuração e a não utilização de *daemons* reduziria bastante esta característica.

Procurando atender a estas necessidades, foi implementada a seguinte solução: ao invocar o utilitário dusers, seria disparado um processo responsável pela criação de tantos *threads* quantas fossem as máquinas existentes no domínio, e cada um destes ficaria incumbido de coletar as informações sobre a máquina que lhe havia sido destinada e gravá-las em um arquivo temporário. O processo principal encarregaria apenas da verificação do término da execução dos *threads*. A FIGURA 2 ilustra esta opção.

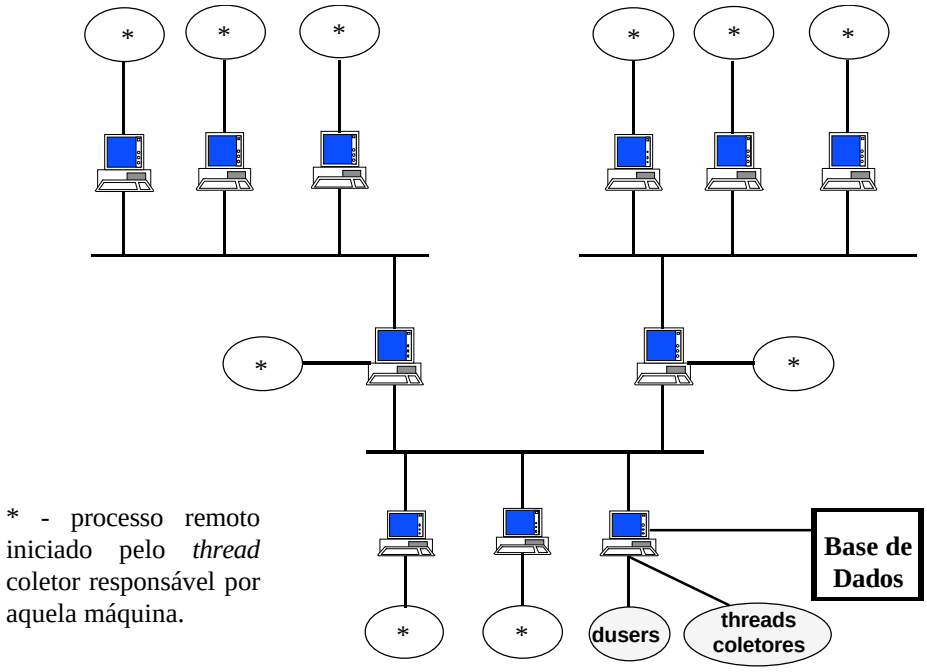

Figura 2 - Solução que utiliza o conceito de *threads* para a coleta das informações nas diversas máquinas de um domínio

3. Descrição da Implementação

O utilitário dusers teve a sua implementação feita para dois sistemas operacionais, a saber: SunOS 4.1.3 e Solaris 2.4 i.é. SunOS 5.4. Apesar das diferenças das bibliotecas *threads* e do próprio tratamento dos respectivos sistemas operacionais, o algoritmo para colher a informação necessária é único, e resumidamente segue o esquema abaixo:

- Conseguir o nome das máquinas do domínio. A obtenção do nome das máquinas do domínio utiliza um arquivo de configuração. Este arquivo era inicialmente criado a partir do resultado do comando `ypcat` *`host`*, disponível em ambientes NIS, *Network Information Service*, mas tal solução não se mostrou viável por existirem *hosts* que não estavam aptos a responder à requisição, gerando a criação desnecessária de *threads* e conseqüente desperdício de processamento. Decidiu-se então manter um arquivo de configuração fixo composto apenas dos *hosts* do domínio que eram capazes de responder à requisição.

- Criar um *thread* para cada máquina existente. A criação dos *threads* varia de acordo com as bibliotecas utilizadas, mas segue a mesma linha, independente do sistema.

- Durante a criação de cada *thread*, o nome do *host* conseguido no arquivo de configuração, é passado como parâmetro.

- De posse desse parâmetro, o *thread* inicia uma chamada de sistema para requisitar as informações. A chamada executa o comando `finger` com os devidos argumentos, e a resposta é colocada num arquivo temporário, que é tratado pelo *thread* e apresentado ao usuário, finalizando a sua execução.

- Esperar o último *thread* apresentar sua resposta e terminar. O processo principal tem como tarefa o gerenciamento dos *threads* por ele criados.

A implementação efetuada tem um aumento da sua utilidade devido a uma série de parâmetros que podem ser passados ao utilitário quando da sua chamada. A TABELA 1 apresenta tais parâmetros. A referência sobre estes parâmetros também pode ser encontrada na *man page* também disponibilizada com o dusers [5].

Parâmetro	Função
(nenhum)	Apresenta o username de todos os usuários em todos os hosts do domínio especificados no arquivo de configuração
-a	Apresenta informações completas sobre os usuários: username, nome completo, TTY, idle time, quando, onde e a partir de que máquina está logado
-f [filename]	Assume que o(s) nome(s) do(s) host(s) a ser(em) pesquisados está(ão) em [filename]. Este arquivo deve conter apenas o nome de um host por linha e nenhuma linha em branco
-h (host1 host2 ...)	Faz com que sejam pesquisados apenas os hosts informados na linha de comando. Quando utilizado, deve sempre ser o último parâmetro a ser especifi-cado, caso contrário, todos os parâmetros posteriores a este serão desprezados
-n	Indica quais as máquinas do domínio que não estejam sendo utilizadas por nenhum usuário
-t (n)	O argumento n especifica o tempo em segundos que, supostamente, cada thread deverá permanecer no estado sleep antes que seja realizada uma nova tentativa de verificação da finalização do comando `finger`. Este parâmetro encontra-se disponível apenas na implementação do sistema operacional SunOS
-letra	Apresenta um resumo do modo de utilização (usage), já que qualquer outra letra não é reconhecida como parâmetro.

Tabela 1 - Parâmetros do Utilitário Dusers

Serão apresentados a seguir detalhes sobre as implementações realizadas nos sistemas operacionais SunOS e Solaris, utilizando-se das bibliotecas de *threads* providas em ambos os sistemas. Salienta-se aqui que essas bibliotecas apresentam diferenças não somente sintáticas, mas também semânticas, resultando em códigos fonte distintos.

3.1. Implementação em SunOS

A biblioteca *lightweight processes*, lwp, no SunOS é implementada em nível de usuário. O *kernel* não tem conhecimento da existência dos *threads*, que funcionam como co-rotinas, tratando a aplicação como uma aplicação normal [9]. Dessa forma, quando um *thread* é bloqueado pelo *kernel*, a aplicação toda também o é, prejudicando o paralelismo desejado [1]. Exatamente por este motivo, houve uma preocupação com o controle de escalonamento dos *threads* no SunOS.

A chamada de sistema para executar o comando `finger` (*system("finger...")*) [3, 4], por definição, bloqueia o processo até que a resposta da chamada retorne. Se cada *thread* estivesse sujeito a ser bloqueado, resultando no bloqueio também da aplicação, o resultado seria um tempo de resposta elevadíssimo, pois os *threads* executariam seqüencialmente e ficariam parados esperando a resposta dos sistemas.

Foi acrescentado o "&" ao final dos argumentos do `finger`, para que este fosse executado em *background* e retornasse o controle rapidamente para a aplicação. Nesse ponto, um outro *thread* qualquer é explicitamente invocado para execução. Após todos os *threads* serem disparados, é efetuado o teste da existência das respostas. Caso uma resposta exista, ela é filtrada e apresentada ao usuário; do contrário, o *thread* principal novamente invoca outro *thread* específico para a execução naquela máquina. Se a verificação da existência da resposta falhar por um determinado número de vezes, então o *thread* encerra a sua execução supondo que não haverá resposta da máquina (*time-out*).

A prioridade dos *threads* também é um fator decisivo na forma de execução do dusers. Se eles têm baixa prioridade, após a sua criação são colocados numa fila de espera para a execução. Com a prioridade muito alta, os *threads* executariam seqüencialmente, como procedimentos, invalidando o paralelismo. Buscando resolver tais dificuldades, colocou-se os *threads* com prioridade baixa para execução e, logo após sua criação, são explicitamente escalonados através da função `lwp_yield()`[2].

3.2. Implementação em Solaris

Na arquitetura *multi-threaded* do Solaris, a implementação dos *threads* se dá de forma híbrida, ou seja, uma parte é como biblioteca de usuário (*threads*), e a outra é implementada em nível de *kernel* (*lightweight processes*). Os *lightweight processes*, *lwp*, funcionam como processadores virtuais, gerenciam uma determinada quantidade de *threads* e são bloqueados pelo *kernel* quando um de seus *threads* é bloqueado [6, 8, 9]. Sendo assim, se cada *lwp* gerenciar apenas um *thread*, e sendo este bloqueado, não resultará em prejuízo ao paralelismo como um todo.

A chamada de sistema que executa o comando `finger` não precisa conter o "&" para que este seja executado em *background*, pois quando este for bloqueado pelo *kernel* existirão outros *threads* em outros *lwp* prontos para execução. O escalonamento é feito diretamente pelo *kernel*, não necessitando haver chamadas explicitas de escalonamento no código.

Quando a chamada de sistema retorna a aplicação, o *thread* bloqueado é automaticamente reescalonado, iniciando então a filtragem e apresentação dos resultados.

Contudo, outros cuidados devem ser tomados. A função `thr_exit()`[6, 7], quando executada pelo último *thread* da aplicação, encerra não apenas o *thread*, mas também a aplicação, já que o programa principal não é considerado um *thread* [6]. Se o primeiro *thread* criado executar tal função antes da criação de outros *threads*, haveria um comportamento não desejado da aplicação, encerrando-a prematuramente. Modificou-se a estrutura do programa para que houvesse um *thread* principal, responsável pela criação e gerenciamento dos demais *threads*. Este somente executará `thr_exit()` após checar que todas as máquinas responderam aos processos chamados por seus *threads* disparadores ou foram consideradas inoperantes com relação ao comando `finger` utilizado. Garante-se portanto o final de execução do utilitário de maneira controlada.

4. Comparação das Implementações

As implementações efetuadas em ambos os sistemas operacionais UNIX apresentaram grandes diferenças não apenas em relação às variações nas implementações das bibliotecas de *threads* utilizadas, mas também no que se refere à política que as rege. A seguir, apresentaremos tais diferenças.

A biblioteca de *threads* utilizada no sistema operacional SunOS 4.1.3 permite que o usuário efetue um controle explícito sobre o escalonamento dos *threads*, gerenciando a execução dos mesmos através de procedimentos que se encarregam de interromper a execução de um *thread* e executar a invocação de um outro. Além disso, cabe ao programador saber quando um *thread* ficará bloqueado devido, por exemplo, a uma chamada de sistema. Caso seja possível evitar esta situação, o programador deve providenciar o escalonamento de outro *thread*. Um exemplo desta situação foi a necessidade da utilização do "&" no comando `finger`, forçando sua execução em *background*.

Por sua vez, a biblioteca de *threads* utilizada no sistema operacional Solaris 2.4 representa um grande avanço em relação à implementação na versão do SunOS 4.1.3. Aqui é fornecida uma biblioteca bem mais inteligente na qual o programador preocupa-se principalmente com a criação dos *threads*, cabendo ao sistema operacional o total gerenciamento dos mesmos. Agora o próprio *kernel* encarrega-se de saber que um determinado *thread* bloqueou devido a uma solicitação de serviço ao sistema operacional e providenciar seu escalonamento automaticamente. A TABELA 2 lista as diferenças básicas entre as diferentes implementações.

SunOS	Solaris
Threads somente em nível de usuário, funcionam como co-rotinas em um processo comum	Threads híbridos, parte em nível de usuário, parte em nível de kernel
Serviço de threads disponível com acessório de programação	Evolução do serviço de threads, deixando de ser um acessório para tornar-se utilitário oferecido pelo sistema
Programador é obrigado a controlar explicitamente o escalonamento dos threads	Programador não se preocupa com o escalonamento dos threads, pois isto é feito diretamente pelo kernel
Chamadas de sistema bloqueiam a aplicação como um todo	Chamadas de sistema bloqueiam apenas o lwp e os threads por ele gerenciados
Implementação mais complexa devido a preocupação com o paralelismo	Implementação mais fácil e lógica, devido ao maior controle por parte da biblioteca de threads
Performance aceitável	Performance superior à implementação em SunOS

Tabela 2 - Diferenças das Implementações de *Threads* nos Sistemas Distintos

Com relação às outras soluções consideradas inicialmente, constatou-se que a adotada apresentava um bom desempenho principalmente pela rapidez com que são executados os *threads*. A não existência de uma base de dados centralizada e permanente eliminou não apenas a necessidade do armazenamento permanente de informações, mas também a dependência da disponibilidade do utilitário em relação a uma determinada máquina que agora só depende daquela onde é invocado o utilitário. A utilização de *threads* permitiu um elevado grau de paralelismo, e a execução do utilitário de forma local fez com que o consumo de processamento ocorresse principalmente na máquina local, diminuindo bastante esta penalização no restante do sistema.

5. Conclusão

Baseando-se no que foi apresentado, constata-se que houve uma grande evolução na biblioteca de *threads* do sistema operacional SunOS 4.1.3 para o Solaris 2.4, destacando-se o alto grau de independência que o segundo oferece em relação ao primeiro, o que possibilita uma diminuição da carga de trabalho do programador pois este passa a preocupar-se principalmente com a solução do seu problema e não com detalhes de gerenciamento de recursos do sistema.

A dificuldade encontrada para configurar a ferramenta foi resolvida através da utilização de um arquivo de configuração que, apesar de representar uma tarefa a mais a ser executada pelo administrador, representa uma saída pouco trabalhosa se comparada com as outras soluções encontradas.

Com relação à ferramenta desenvolvida, esta revelou-se bastante útil por não apenas resolver o problema inicial, que era o de saber quem estava logado em um

domínio num determinado instante, mas também prover diversas outras informações através das várias opções por ela oferecidas.

O desempenho apresentado pelo utilitário implementado é bastante bom, com destaque para a implementação do sistema operacional Solaris 2.4, onde a rapidez de resposta é marcante.

6. Referencia Bibliográfica

[1] SunOS 4.1 AnswerBook, “LightWeight Processes”, capítulo 2

[2] SunOS 4.1 AnswerBook, “LightWeight Processes Library”.

[3] SunOS 4.1 AnswerBook, “C Library Functions”.

[4] SunOS 4.1 AnswerBook, “User Commands”.

[5] SunOS AnswerBook, “The -man Macro Package”, 3.1-3.9; “Reference Manual Pages”, págs. 191-202.

[6] Solaris 2.4 AnswerBook, “Guide to Multi-thread Programming”, capítulos 1, 2, 3, 7.

[7] Solaris 2.4 AnswerBook, “System Services”, capítulo 4.

[8] M. L. Powell et al, “SunOS Multi-*thread* Architecture”, USENIX Winter 1991, págs. 1-14.

[9] Tanenbaum, Andrew S., Modern Operating Systems, Prentice Hall 1992.