UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

ESCOLA POLITÉCNICA
DEPARTAMENTO DE ENGENHARIA ELETRÔNICA E DE COMPUTAÇÃO

PROJETO FINAL

Orgaperf
Monitor de Desempenho para o OPSC

Autor: _____
Luiz Felipe de Souza e Silva

Orientador de Projeto Final: _____
Antônio Cláudio Gómez de Sousa, M. Sc.

Examinador: _____

Examinador: _____

DEL
Maio/2008

Agradecimentos

Agradeço a Deus e aos meus pais por me darem a oportunidade de estudar e chegar à universidade. A toda a minha família e amigos pela paciência das várias vezes que não pude comparecer aos compromissos marcados devido ao curso de engenharia.

Agradeço ao meu professor e orientador, Antonio Claudio, pela enorme paciência em me orientar nesse projeto.

Agradeço ao pessoal da Orga Systems, em especial ao Paulo Jorge por seu total apoio a produção do Orgaperf, mesmo quando várias pessoas me instruíam o contrário, e por sua ajuda decisiva em por o software em ambiente de produção, ao Paulo Coelho, chefe dos engenheiros de serviços, pois foi por causa dele que surgiu toda a idéia do Orgaperf, ao Alberto Pereira por dar continuidade ao projeto depois do meu desligamento da empresa e ao Götz Schuchart, diretor de tecnologia da Orga Systems Brasil, por gentilmente ceder a utilização desse software para fins acadêmicos.

Resumo

Este projeto descreve um software de análise de desempenho para um sistema proprietário, o OPSC.

O OPSC é um sistema de tarifação de telefones pré-pago utilizado por empresas de telefonia móvel.

Palavras Chave

Engenharia de Software;
tarifação;
telefonia móvel;
análise de desempenho;

Sumário

Lista de Figuras

Lista de Tabelas

Glossário

Array: Vetor de dados, onde dado um índice, esse é somado a um ponteiro base e o resultado aponta para a área de memória onde está armazenado o dado desejado.

AWK: Linguagem de programação especializada em tratar textos

BSD Sockets: Trata-se da interface padrão por onde os programas que necessitam se comunicar com a rede "conversam" com o sistema operacional.

Buffer: Área de memória usada para armazenamento temporário de dados durante operações de I/O.

Datasource: Origem de dados; podem vir de inúmeras fontes, um banco de dados, um arquivo,...

Fork: Função especial em sistemas Unix® onde um processo pode criar uma cópia de toda a área de memória utilizada por ele e criar uma nova entrada no escalonador de processos, fazendo com que as duas cópias sejam executadas concorrentemente.

Hash: É um vetor associativo, onde a *string* dada como argumento do vetor leva a um identificador único que contem a área de memória onde está armazenado o dado desejado.

Hotfix: É uma atualização feita de forma emergencial que não passa pela equipe de qualidade.

Kernel: É a alma do sistema operacional, é o mediador entre os programas do usuário e o hardware do servidor.

Overhead: Consumo extra de recurso para realizar uma determinada operação.

Patch: É uma atualização feita a um software.

Patchlevel: É a versão da atualização do software. Também conhecido como revisão.

Pipe: Mecanismo de comunicação inter-processos onde um processo escreve em uma ponta e o outro processo lê o que foi escrito na outra ponta.

Regular Expressions: São padrões de reconhecimento de formações regulares de texto.

Semaphore: É um mecanismo de sincronização de processos concorrentes, como se fosse um sinal de trânsito. Um processo só pode proceder quando o semáforo permitir.

Short Message: Também conhecido como torpedo.

String: Dados em formato binário ou texto.

Thread: Também conhecido por processo leve, assim como o fork, cria outra entrada no escalonador de processos, mas não cria uma cópia da área de memória do processo original, os dois processos são executados utilizando a mesma área de memória.

Threshold: É um intervalo próximo a um determinado valor onde o valor medido ainda é considerado normal.

Trouble Ticket: É um pedido de auxílio de suporte à Orga System que é aberto pelo sistema interno da empresa.

Unix timestamp: Formato de armazenamento da data corrente em sistemas Unix® que corresponde à quantidade de segundos desde 1º de janeiro de 1970.

1. Introdução

Este capítulo dará uma breve introdução sobre o mercado de telefonia pré-paga, e onde a Orga Systems se encaixa nesse mercado, em seguida, no segundo capítulo, será descrita a metodologia utilizada pela Orga Systems para identificar problemas em seu principal produto.

O terceiro capítulo descreverá a proposta de uma nova ferramenta para ajudar nos diagnósticos realizados pelo suporte da Orga Systems, o quarto capítulo descreve o seu planejamento, seguido da análise no quinto capítulo e no sexto capítulo a opção de arquitetura utilizada. O sétimo capítulo mostra como foi realizado a implementação da ferramenta, seguido dos resultados no oitavo capítulo e finalizando com as conclusões no nono capítulo.

1.1. O Mercado de telefones móveis pré-pagos

O início das operações de celular pré-pago começou no início da década de 90 com a *Houston Cellular Telephone Company*. Mas dificuldades técnicas impediam que esse tipo de serviço viesse a se tornar o carro chefe das companhias telefônicas daquela época. Uma das dificuldades principais era como realizar o provisionamento do serviço, que só veio a ser vencida em 1994, quando foi patenteado o primeiro sistema de provisionamento para telefones pré-pagos.

Os celulares pré-pagos começaram a ser disponibilizados no Brasil em 1998 pela CTBC Celular e passaram a representar rapidamente a maioria dos celulares. Em agosto de 2003 73,65% dos celulares no Brasil eram pré-pagos. A Figura 1.1 apresenta a evolução crescente deste percentual.

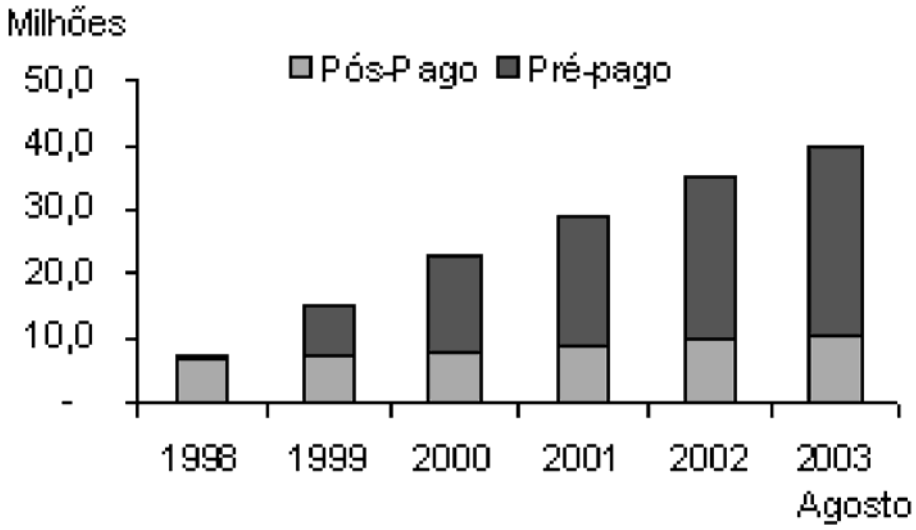


Figura 1.1 Evolução em números absolutos de telefones pré-pagos no Brasil.

Para a tarifação dos celulares pré-pagos era necessária uma técnica especial, diferenciada da técnica dos celulares pós-pagos. Em julho de 1998 a TIM contratou os serviços da Orga Systems, empresa multinacional com experiência na tarifação para celulares pré-pagos, e parceira tradicional da TIM na Europa. Assim a Orga Systems passou a apoiar a TIM na área de tarifação (billing) de serviços pré-pagos com tecnologia GSM. Esses serviços pré-pagos começaram a ser implantados em outras operadoras de celulares adquiridas pelo grupo italiano pela América Latina, como por exemplo, a *Telecom Personal Argentina* (TIM TPA).

Durante muito tempo, o trabalho foi prestado por uma equipe técnica estabelecida em território alemão, onde Orga Systems é sediada. Mas, em meados de 2004, houve a decisão estratégica de transferi-lo para o território brasileiro. Para isso, foi criada uma subsidiária no Rio de Janeiro que começou a montar uma equipe técnica para prestar o serviço. Um dos motivos que levou a essa decisão foi que em 1998 quando a Orga Systems iniciou suas operações no Brasil juntamente com a operadora TIM, existia apenas um sistema OPSC instalado cobrindo todo o Brasil. Conforme o aumento do número de aparelhos pré-pago aumentou também o número de sistemas OPSC instalados. Em 2006 já eram quatro sistemas OPSC em operação, cada um responsável por uma determinada região do Brasil, e os preparativos para um quinto sistema já estavam em andamento.

Além do aumento da quantidade de trabalho, havia também problemas com o fuso horário, que fazia com que o horário de trabalho na Alemanha raramente fosse compatível com o horário brasileiro e prejudicava o atendimento a um cliente tão importante quanto a TIM Brasil. Além disso, com uma subsidiária no Brasil poderia ser dada maior atenção a América Latina, onde a Orga Systems possui outros clientes também importantes. A Orga iniciou suas operações no Brasil com quatro funcionários e apenas a TIM Brasil como cliente. Hoje já são mais de setenta funcionários, entre eles, gerentes de projetos, desenvolvedores, especialistas em testes entre outros, com clientes na Argentina, Chile, Guatemala e Venezuela.

2. Descrição do modelo de trabalho anterior

A equipe da Orga Systems responsável pela manutenção das instalações dos sistemas da empresa na América latina é conhecida como *Service Engineers* ou engenheiros de serviços. Os engenheiros de serviços são responsáveis por auxiliar os clientes em instalações, atualizações e operação dos produtos criados pela empresa.

O produto principal da empresa é conhecido como OPSC (*Orga Prepaid Service Center*), que é um sistema de tarifação para telefones pré-pagos para redes GSM. A localização de sua função na rede GSM pode ser apreciada pela Figura 2.1.

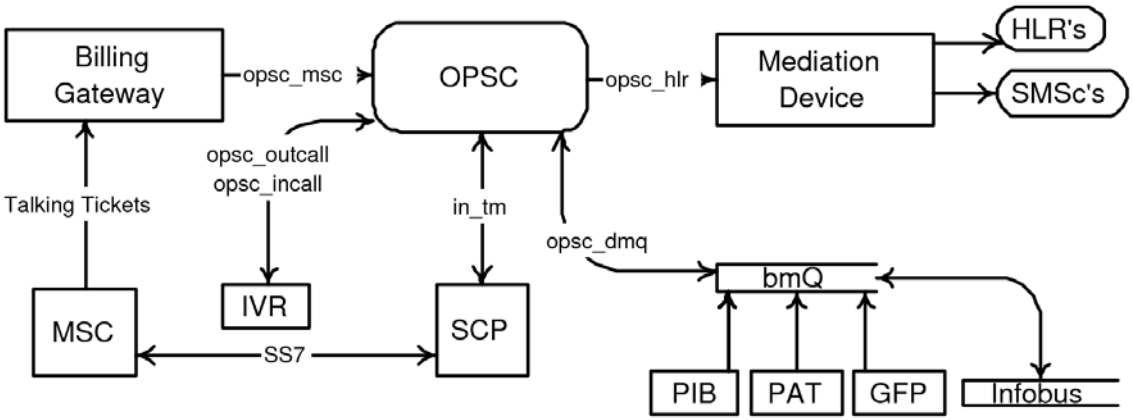


Figura 2.1 Arquitetura de tarifação simplificada de uma rede de telefonia móvel, representação macro blocos

Na Figura 2.1 estão representados por retângulos alguns elementos de rede, que abaixo serão descritos:

- **Billing Gateway** É o lugar onde os tickets assíncronos são armazenados, quer dizer, tarifados após o término da ligação. Essa forma de tarifação é também conhecida como *hot billing*.
- **MSC** *Master Station Controller*, também conhecida como central telefônica. Ao contrário da rede telefônica convencional onde existem várias centrais, na rede móvel, essas centras se reduzem a algumas unidades apenas. Geralmente uma por DDD, mas dependendo da quantidade de terminais instalados pode haver mais.

- **SCP** *Service Control Point*, é o ponto onde existe a integração da rede inteligente (*IN*) utilizando o protocolo SS7 com a rede onde o OPSC está ligado utilizando o protocolo IP sobre rede ethernet. SS7 é um protocolo padrão para comunicação e sinalização de centrais.
- **Mediation Device** A rede de mediação é a interface por onde o OPSC pode enviar comandos para as centrais. Por exemplo, um usuário consumiu todos os créditos do seu aparelho, o OPCS está apto a terminar a ligação do usuário, enviando um comando chamado *tear down* para a mediação e de lá para a central desejada, desconectando o terminal da ligação.
- **IVR** *Interactive Voice Response*, também conhecido pelos usuários de aparelho pré-pago por ligar para o usuário avisando que está com saldo baixo ou dando as boas vindas à operadora.
- **bmQ** *BEA message queue*, é a interface do OPSC com os demais sistemas da operadora, como promoções, recargas, *customer care*, ...

O OPSC por ser de um conjunto de softwares, é concebido dentro de um modelo de instalação onde a operadora fornece o hardware e o sistema operacional, que é de responsabilidade do departamento de Tecnologia da Informação da operadora e a Orga Systems, com o auxílio dos engenheiros de serviços, realiza a instalação do software e suas licenças.

A Orga Systems também oferece auxílio na configuração da integração com os outros sistemas da operadora, como interfaces de tarifação, mediação com a engenharia, serviços de enfileiramento de mensagem (message queues), pontos de controle de serviços (service control points), entre outros.

O modelo de operação mista estabelecida para o OPSC, onde o departamento de TI da operadora controla o hardware e o sistema operacional e a Orga Systems controla o software OPSC é reconhecidamente passível de desentendimentos/incompatibilidades por não ser só uma entidade responsável por todo o sistema e sim uma parceria entre as entidades Orga Systems e a operadora.

Trabalhar nesse modelo exige bastante cooperação e extrema confiança mútua, o que nem sempre é verdade, assim abrindo margem para uma entidade acusar a outra em caso de falha ou degradação do serviço. Dependendo da origem do problema, caso a Orga Systems seja apontada como causadora da falha ou degradação ela pode ser penalizada por quebra da qualidade de serviço.

Para que a Orga Systems possa garantir seus níveis de qualidade de serviço, é exigido da operadora um patamar mínimo de qualidade dos serviços por ela prestado, como por exemplo, tempo de recepção e transmissão de tickets e comandos de/para a plataforma de mediação, além de exigir que o hardware e o sistema operacional tenham recursos suficientes para comportar o OPSC.

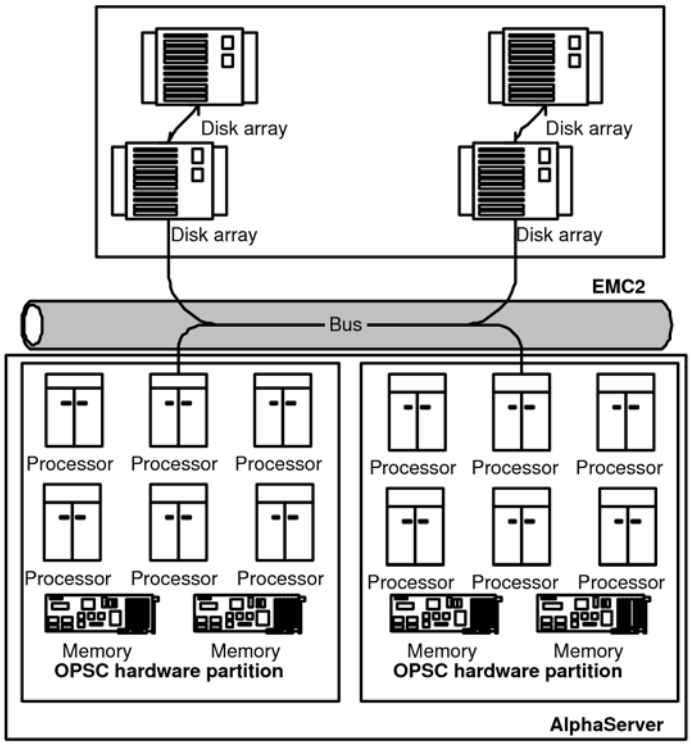


Figura 2.2 Partições de hardware AlphaServer + EMC^2.

O OPSC necessita de hardware baseado em processadores *Alpha* usando o sistema operacional Digital Unix® 4.0 ou HP/Compaq Tru64 5.1. Para reduzir os custos de TI, é prática comum as operadoras montarem o OPSC em servidores corporativos como o *AlphaServer* GS1280, que pode ter até 64 processadores e até 512Gb de RAM em conjunto com disk arrays EMC^2. Por ser um servidor corporativo, existe a possibilidade de existirem outros serviços da operadora nesse mesmo servidor.

Usando a arquitetura descrita, um servidor OPSC na verdade é uma partição de um servidor corporativo em conjunto com algumas partições de um servidor de armazenamento EMC[2].

Além das características arquiteturais de hardware que podem dar problemas, existem outros agentes que também podem influenciar problemas de desempenho, são os problemas de arquitetura de software.

O software OPSC, por ser um sistema chave, exerce função central nas regras de negócio da operadora. Assim, é de se esperar, que outros sistemas, de outras empresas integradoras ou parceiras da operadora também dependam do OPSC.

Porém, a implementação de alguns desses sistemas foi feita de forma tão integrada que necessita estar instalado no mesmo servidor onde o OPSC está instalado.

Se, por um exemplo, um desses sistemas se comporta mal e começa a usar de forma exagerada os recursos do servidor, o OPSC vai sentir os recursos do servidor cada vez mais raros e em conseqüência tem seu funcionamento degradado.

Dado o cenário macro de como funciona a operação do sistema OPSC, sempre que há um problema de desempenho no produto, este problema pode não estar diretamente relacionado à Orga Systems e sim a algum outro sistema. Em todos os casos, um pedido de auxílio ao suporte é criado através de um *trouble ticket* para a Orga Systems, onde um engenheiro de serviços é designado para verificar o problema.

Ao ser designado, o engenheiro de serviços deve realizar alguns procedimentos, como por exemplo, verificar os processos ativos, carga da máquina (load average), valor do uso de memória, além de outros procedimentos próprios para validar o funcionamento do OPSC e descartar qualquer falha ou a sua procurar a origem.

Porém, não é incomum não ter acesso ao servidor problemático no exato momento em que a degradação se iniciou ou às vezes até mesmo quando o problema já

foi normalizado. Não ter os dados históricos do comportamento do servidor deixa difícil ou até mesmo impossível diagnosticar a origem do problema.

Para solucionar a questão do diagnóstico dos problemas que aconteciam com o OPSC, os engenheiros de serviços foram divididos em times e cada time se tornou responsável por determinados clientes com o objetivo de realizar relatórios mensais sobre o comportamento do OPSC, de forma a prever e trabalhar proativamente frente a alguma anormalidade do sistema.

Esses relatórios seriam a base dos gerentes de projeto para provar junto aos clientes que o funcionamento do OPSC é sempre estável e que o problema possivelmente deve ser em algum outro software ou sistema interligado ao software OPSC.

Devido à necessidade de se escrever esses relatórios, cada time escreveu sua série de scripts para capturar dados de desempenho de cada sistema de forma não coordenada criando diferentes programas para realizar uma mesma tarefa.

Esses scripts armazenavam os dados de desempenho em um arquivo texto que continha aproximadamente um período de 24 horas de dados e armazenava os seguintes tipos de dados:

- Valores como consumo de CPU, quantidade de processos ativos, utilização de memória física e swap, e os 10 processos que mais consumiam CPU.
- Valores do comportamento interno do OPSC, como tickets recebidos:
    - Tickets recebidos pela rede inteligente, que é um caminho mais sensível à degradação de desempenho por serem 100% *online* e necessitar de tempo de resposta na ordem de milisegundos.
    - Tickets recebidos por *hot billing*, que é o caminho assíncrono.
    - Filas de comando internas do OPSC. Ex: fila para tickets que devem ser tarifados, fila do *mediation device* para envio de comandos à rede, fila do *IVR* para mensagens de boas vindas e alertas.

A data em que esse relatório era gerado era escolhida ao acaso durante o mês, onde o comportamento de um único dia era tomado como amostragem para o mês inteiro.

Os dados gerados por esses scripts demandavam uma quantidade enorme de trabalho, para alterá-los e reinterpretá-los, mês a mês, e os consolidá-los em uma planilha Excel.

Uma forma de minimizar o trabalho para realizar os relatórios de análise de desempenho precisava ser inventada, e foi com essa idéia que surgiu o propósito de criar o Orgaperf.

Outro motivo importante para a criação do Orgaperf é manter uma série histórica de dados de desempenho considerados importantes pelos engenheiros de serviços, deixando mais fácil o diagnóstico e a possibilidade de precisar exatamente quando o problema iniciou e quando voltou ao normal.

A operadora também detém esses dados através de um software chamado HP OpenView®, mas o grupo que administra o software não permite livre acesso as informações lá contidas e também procura dificultar todas as tentativas de obter-las, requerendo uma série de exigências burocráticas, pois só com o consentimento do gerente da área era possível pedir a emissão do relatório.

Quando os engenheiros da Orga conseguiam o relatório do HP OpenView®, o mesmo já se encontrava obsoleto, mostrando dados de semanas atrás e nem sempre exibiam a janela de tempo pedida, prejudicando a identificação do início ou do fim do problema.

Saber pelo menos quanto tempo durou o problema é um dado importantíssimo também para os gerentes de projeto, pois este deve ser contabilizado nos relatórios de indisponibilidade do sistema.

3. Objetivos da nova ferramenta

O objetivo principal que a nova ferramenta deve contemplar é armazenar uma série histórica de dados de desempenho considerados importantes pelos engenheiros de serviços.

Entendem-se por dados de desempenho tudo o que é considerado por um engenheiro de serviços como uma informação importante para que sua análise sobre um determinado problema seja válida, tudo que possa balizar o laudo dado como real causa do problema e ajude a apontar possíveis soluções.

Além de dados que ajudem analisar problemas, outros dados que ajudem a compreender o comportamento do OPSC, indicando qual variação de um determinado recurso impacta na degradação do desempenho do sistema, ajudando a prever os requerimentos de hardware e software necessários para comportar a expectativa de expansão da base de terminais móveis a ser instalado no próximo ano.

Entre os dados de desempenho mais importantes, podemos citar.

- Utilização de CPU;
- Utilização de memória física e Swap;
- Utilização de recursos de I/O;
- Quantidade de *talking tickets* processados;
- Tamanho das principais *FIFO*s do OPSC;

Em uma situação ideal, para ajudar os gerentes de projeto e minimizar os trabalhos dos engenheiros de serviços, deveria existir um software que preenchesse todos os requisitos abaixo:

- Ser completamente automático;
- Capturar todos os dados necessários de análise de desempenho;
- Consolidar esses dados capturados;

- Gerar automaticamente as planilhas com os gráficos do que foi consolidado;
- Emitir, caso necessário, um alarme caso algum dos dados fugisse uma determinada folga do que é considerado comum (*threshold*);
- Ser de fácil instalação;
- Ser de fácil utilização;
- Não funcionar com privilégios de super-usuário (*root*);
- Ser multi-plataforma;
- Ser de fácil modificação, mesmo que em campo (no próprio cliente);
- Funcionar em *background*;
- Ser imperceptível para o cliente;
- Não concorrer com os recursos exigidos pelo software OPSC.

Procurando contemplar todos os requisitos da situação ideal, o Orgaperf foi proposto como um software único e flexível que onde todos os times pudessem instalá-lo nos sistemas em que eram responsáveis e unificar a força de trabalho e focar em dois propósitos, facilitar a criação do relatório de desempenho mensal e atender melhor aos clientes da Orga Systems.

Deve se deixar claro que a concepção desse software se baseia em um problema concreto, vivenciado diariamente pelos engenheiros de serviços e que necessitava de solução. Este projeto materializa a oportunidade de produzir algo baseado nessa experiência diária, do contato com o cliente e as instalações do OPSC por ele hospedadas.

Esse software deve ser escrito em linguagem script para ser fácil dar manutenção em campo e, preferencialmente, ser escrito em uma linguagem que todos os engenheiros de serviços estejam familiarizados, ser de fácil compreensão e exigir o mínimo esforço para se adaptar a novos sistemas.

4. Planejamento

Devido à forma em que o projeto foi conduzido, um questionário para o detalhamento do projeto, não foi realizado. Os requisitos do projeto foram levantados e analisados pelo mesmo engenheiro que realizou a implementação do software. Os riscos relacionados a essa atitude foram levados em conta na análise dos riscos

4.1. Escopo e objetivos do projeto

O Orgaperf visa coletar e armazenar dados de desempenho, assim ajustando a situação ideal à realidade do que pode ser entregue em tempo hábil. O que se deve esperar desse software é que, além de armazenar os dados históricos considerados importantes pelos engenheiros de serviços, possa:

- Funcionar em *background*;
- Funcionar de forma integrada ao OPSC;
- Ser de fácil instalação;
- Ser de fácil utilização;
- Não funcionar com privilégios de super-usuário (*root*);
- Ser imperceptível para o cliente;
- Ser de fácil modificação, mesmo que em campo (no próprio cliente);
- Não concorrer com os recursos exigidos pelo software OPSC;
- Evitar ter uma interface entre o software e os engenheiros de serviços complexa.

Dos objetivos do software, podemos verificar três macros blocos de funcionalidades que foram implementados:

- Captação;
- Armazenamento;
- Exibição.

Cada macro bloco foi implementado como um módulo, por ter um escopo de atuação bem definido, e cada módulo foi desenvolvido de forma estanque, por ter sido definida uma interface comum para a comunicação entre eles.

Uma das grandes vantagens de adotar a modelagem do software em módulos é que é possível realizar o desenvolvimento baseado em testes. Uma vez definindo o comportamento que cada módulo deve reproduzir, é possível construir um teste e desenvolver de forma a sempre passar nesse teste, aumentando a qualidade do software entregue e diminuindo o tempo de desenvolvimento, por não precisar de um período de testes estendido.

Mesmo não sendo um requisito explícito, cabe propor que o software foi modelado de forma a ter portabilidade, porque outras aplicações da Orga Systems rodam em outros sistemas operacionais como o Sun Solaris e o Linux.

4.2. Itens de Desempenho

O Orgaperf não tem requisitos grandes de desempenho, mas é imperativo que os recursos consumidos não concorram com o OPSC.

4.3. Ambiente de desenvolvimento

O ambiente de desenvolvimento é um servidor Compaq Proliant DL360 utilizando o sistema operacional HP Tru64 5.1B.

O servidor de desenvolvimento utilizou um hardware inferior ao de produção, mas com a mesma plataforma de hardware e o mesmo processador Alpha, a mesma versão de sistema operacional e o mesmo *patchlevel*.

O software OPSC instalado nesse servidor tem a mesma versão e os mesmos *patches* e *hotfixes* que o ambiente de produção.

Procurando reproduzir o ambiente de produção no ambiente de desenvolvimento da melhor forma possível, as bases de dados utilizadas pelo OPSC em produção foram reduzidas para que pudessem funcionar com os recursos do ambiente de desenvolvimento.

O ambiente de desenvolvimento contempla um compilador ANSI C para desenvolver softwares utilizando essa plataforma de hardware enquanto que no ambiente de produção não há compilador algum.

4.4. Método de análise

O objetivo da ferramenta, o software Orgaperf, é ser pequeno e simples.

Para diminuir a quantidade de abstrações e simplificar o projeto, o método a ser adotado foi à análise estruturada.

4.5. Gerenciamento de mudanças

Mudanças não são esperadas no desenvolvimento desse software, o desenvolvedor, como um engenheiro de serviços, terá pouca necessidade de alterá-lo uma vez terminado a especificação.

4.6. Estimativas

Para realizar as previsões de esforço do projeto, usou-se o software COCOMO II.1999.0, a opção de utilizar esse software partiu da extensa experiência do grupo que o criou em estimar projetos.

Para este sistema, o foram previstos 33 pontos por função e 3063 linhas de código usando uma linguagem de alto nível. Considerou-se que 2% de código seriam desperdiçados devido a erros e volatilidade dos requisitos.

A estimativa de tempo para implementação do projeto dada pelo COCOMO foi de 7.3 semanas de trabalho.

4.7. Cronograma do projeto

A tabela abaixo apresenta o cronograma do projeto, e a seguir são definidas as atividades do cronograma.

Tarefa	1 a 16/7/06	17 a 31/7/06	1 a 14/8/06	15 a 31/8/06	1 a 16/9/06	17 a 30/9/06
1	x					
2	x					
3		x				
4		x	x			
5				x	x	x
6						x
7						x

Tabela 4.1 Cronograma do Projeto

Atividades:

1. Planejamento, definição de escopo, estratégias de gestão de mudanças, ricos e elaboração do cronograma;
2. Modelagem conceitual, definição de interfaces;
3. Desenvolvimento do módulo de captura de dados e testes;
4. Desenvolvimento do módulo de armazenamento de dados e testes;
5. Desenvolvimento do módulo de exibição de dados e testes;
6. Publicação do software em ambiente de produção;
7. Ajustes do software em ambiente de produção.

4.8. Recursos do projeto

O software Orgaperf utilizou de recursos de apoio ao desenvolvimento, principalmente um compilador C e para a linguagem Perl. As duas já estão pré-instaladas no servidor de desenvolvimento.

Algumas bibliotecas foram utilizadas ao longo do projeto conforme for necessário, por exemplo, o RRDtool.

O recurso humano alocado ao projeto foi uma pessoa, mas dada a característica de desenvolvimento de módulos, mais pessoas poderiam ser alocadas para ajudar no desenvolvimento para diminuir o tempo de entrega do software.

Assim, a tabela de recursos do projeto é mostrada na Tabela 4.2.

Tarefa	Recurso	Dedicação	Início	Fim
Planejamento	Luiz Felipe	40 hrs	1/07/06	10/07/06
Modelamento	Luiz Felipe	40 hrs	11/07/06	16/07/06
Mod. Captura	Luiz Felipe	60 hrs	31/07/06	2/08/06
Mod. Armazenamento	Luiz Felipe	110 hrs	2/08/06	14/08/06
Mod. Exibição	Luiz Felipe	170 hrs	14/08/06	27/09/06
Publicação	Luiz Felipe	20 hrs	27/09/06	28/09/06
Ajustes	Luiz Felipe	30 hrs	28/09/06	30/09/06

Tabela 4.2 Alocação de recursos humanos

4.9. Estratégias para o Gerenciamento de Riscos

O software Orgaperf conta com uma interface muito simples para os usuários, os engenheiros de serviços da Orga System. Como todos têm amplo conhecimento em sistemas Unix® e a comunicação usa o comando kill, foi eliminado o risco de não possuir uma interface amigável com o usuário.

Mesmo sendo o OPSC um produto da Orga Systems, qualquer novo software instalado no servidor deve receber o aval da operadora, sinalizando a aceitação da instalação do novo módulo ou software. No caso do Orgaperf o seu desenvolvimento foi conduzido sem a certeza prévia de que a sua instalação seria bem recebida, caracterizando um sério risco a implementação em ambiente de produção.

Existia ainda outro risco importante. Um software desenvolvido por apenas uma pessoa impõe um alto risco no desenvolvimento e na entrega do software dentro do prazo esperado. Caso alguma eventualidade ocorresse o software estaria seriamente comprometido.

Os principais riscos do projeto estão listados na Tabela 4.3.

	Risco	Categoria	Impacto	Probabilidade
Produto	Aceitação	Usuário	Catastrófico	40%
Produto	Prazo	Pessoal	Grave	20%
Produto	Eventualidades	Pessoal	Grave	20%
Produto	Softwares não licenciados	Legal	Grave	1%
Processo	Complexidade maior que o esperado	Implementação	Grave	10%
Processo	Mudança de requisitos do software	Implementação	Grave	10%

Tabela 4.3 Análise de Riscos

A estratégia que foi adotada para mitigar os riscos de probabilidade moderada e alta foi:

- **Aceitação** A Orga Systems, através de um dos engenheiros de serviços mais influentes para o cliente, procederá à instalação e demonstrará as funcionalidades positivas que o produto tem a fornecer, procurando minimizar ao máximo o risco de aceitação.
- **Prazo** O método de desenvolvimento do software em módulos e, juntando desenvolvimento e teste garante, mesmo que o tempo estipulado no cronograma não seja suficiente, que a parte do software pronta seja demonstrada, assim sendo possível negociar mais prazo para a entrega do produto completo.
- **Eventualidades** A equipe de engenheiros de serviço estará acompanhando o desenvolvimento do software. Caso aconteça alguma eventualidade que impeça por completo a entrega do software, outro engenheiro de serviço pode se voluntariar a continuar o projeto.
- **Complexidade maior que o esperado** A base de funcionamento do software pode ser encontrado em outros softwares para outros sistemas

operacionais, logo o risco de se deparar com um problema de grande complexidade sem solução é baixo.

- **Mudança de requisitos do software** O foco do projeto está claramente definido, sabe-se exatamente o que se pretende com o software, porém uma vez apresentado ao cliente pode ser que este deseje mais funcionalidades, mas estas serão incorporadas apenas se forem julgadas absolutamente necessárias e de curto tempo de implementação.

5. Análise

O Orgaperf é uma ferramenta para a análise de desempenho para o OPSC, logo, o objetivo da ferramenta é captar, armazenar e exibir esses dados de desempenho para melhor auxiliar os engenheiros de serviços.

Neste capítulo vamos mostrar quais são os dados de desempenho considerados importantes, tomar decisões e explicar por quais motivos elas foram tomadas.

5.1. Captando dados de desempenho

Os dados de desempenho considerados importantes pelos engenheiros de serviços podem ser divididos em três grupos.

O primeiro grupo de dados são os que podem ser fornecidos pelo sistema operacional apenas, como uso de memória, CPU, paginação (*Swap*), I/O e quantidade de processos esperando na fila do escalonador de processos (*load average*).

O segundo grupo de dados são os que só podem ser fornecidos através dos arquivos *Protokoll* do OPSC, como a quantidade de tickets de tarifação processados e o tempo de resposta dos comandos enviados a mediação.

O terceiro grupo de dados são os que só podem ser medidos se forem observados à quantidade de arquivos em um diretório, que é o caso das filas *FIFO* do OPSC. Essas filas têm sempre a mesma estrutura, um arquivo *IN*, um arquivo *OUT*, e os arquivos com o conteúdo de cada entrada na fila, como pode ser observado na Figura 5.1.

Toda comunicação inter-processos do OPSC é feita por filas *FIFO*, medir o tamanho dessas filas é uma forma de verificar o desempenho de todo o sistema.

Para determinar o tamanho de uma fila basta subtrair o valor contido no arquivo *OUT* pelo valor do arquivo *IN*.

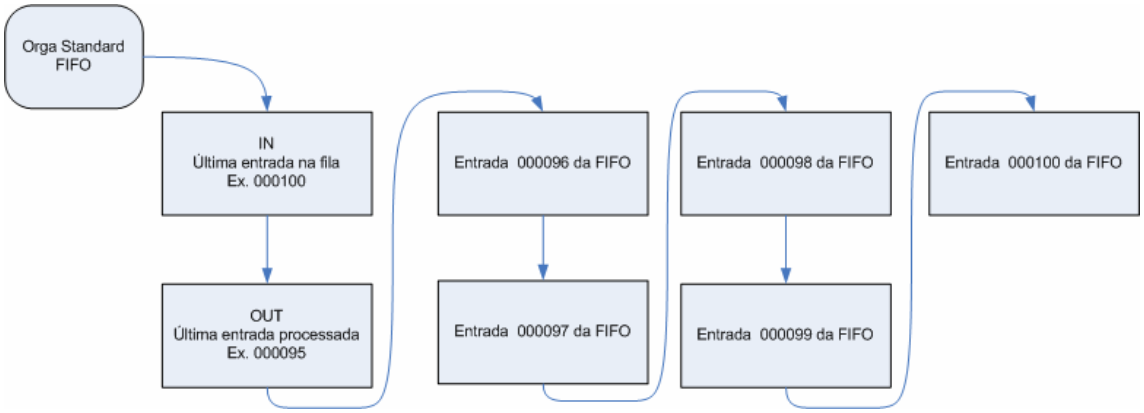


Figura 5.1 Estrutura de uma FIFO padrão da Orga

Desses três grupos de dados que podemos captar, podemos dizer que temos na verdade três famílias de dados:

- **Coleta de dados de desempenho básicos**
  - Uso de memória
  - Uso de paginação
  - Uso de CPU
  - Uso de recursos de I/O
  - Carga no escalonador de processos
- **Coleta de dados de desempenho do OPSC**
  - Tickets de tarifação processados
- **Coleta de dados de desempenho de FIFOs**
  - Tamanho das FIFOs internas

5.1.1. Dados de desempenho básicos

Dados como CPU, I/O e memória (física e virtual) são em sistemas Unix® amostrados usando os comandos *vmstat*, *swapon*, *ps*, *top*, *iostat* e *uptime*.

O *vmstat* dá amostras em intervalos regulares do uso global de CPU e memória de todo o sistema operacional, na Saída 5.1 temos o resultado desse comando.

```
> vmstat 1
Virtual Memory Statistics: (pagesize = 8192)
procs      memory        pages                            intr        cpu
r   w   u  act free wire fault  cow zero react  pin pout  in   sy  cs us  sy id
27  1K 345   2M 627K 489K   78G  21G  19G   10M  17G   7M  3K 191K 24K 17 12 71
24  1K 345   2M 626K 489K 21494  500  618     0  698    0 572 101K 12K 19 10 71
23  1K 345   2M 626K 489K 21780 7980 5582     0 6060    0  2K 112K 16K 17 10 73
24  1K 345   2M 625K 489K 20685 7971 4637     0 5724    0 750 126K 14K 19 10 71
25  1K 345   2M 624K 489K 20650 8134 5045     0 6036    0 427 97K 12K 19  8 73
22  1K 345   2M 625K 489K 19352 8688 5206     0 6233    0  1K 106K 15K 17 9 74
24  1K 345   2M 624K 489K 25772  888 1118     0  923    0 858 131K 16K 17 17 66
28  1K 345   2M 623K 489K 19969 9229 6514     0 7104    0 986 130K 19K 20 11 69
27  1K 345   2M 623K 489K 21929 8121 4738     0 5651    0 764 127K 15K 18 10 72
```

Saída 5.1 vmstat -1

Os campos mais importantes retornados pelo *vmstat* são, debaixo do grupo *cpu*, us (*user*), sy (*system*) e id (*idle*). *User* corresponde ao tempo de CPU gasto efetivamente código executado pelo usuário, *system* corresponde ao tempo de CPU gasto pelo *kernel*, ou seja, pelo próprio sistema operacional e *idle* corresponde a quanto tempo a CPU ficou de "folga".

Outro campo importante está debaixo do grupo *memory*, a limitação desse campo é que ele corresponde apenas à memória real, ou seja, física, aquela que está instalada no servidor, é dados em número de páginas. No Tru64 cada página corresponde a 8192 bytes ou oito kbytes.

Assim dentro desse grupo temos act (*active*), *free* e wire (*wired*). *Active* corresponde a páginas de memória atualmente em uso ou que foram utilizadas recentemente e não devem ser paginadas, *free* corresponde a páginas liberadas para serem alocadas e *wired* são páginas que não podem ser paginadas em hipótese alguma, como o caso de *shared memory*.

Essa saída do *vmstat* exibe os contadores para CPU conforme desejado, mas para realizar o cálculo da memória disponível são necessários mais dois contadores que são a quantidade de memória inativa e a quantidade de memória reservada para *buffers* do *kernel*.

Existe outro parâmetro que pode ser passado ao *vmstat* para que ele mostre detalhes da memória, é o caso do parâmetro '-P'.

```
> vmstat -P
       free pages = 591638
     active pages = 1064731
   inactive pages = 1359829
      wired pages = 488647
        ubc pages = 553714
        ==================
            Total = 4058559
```

Saída 5.2 vmstat -P

A Saída 5.2 já nos dá todos os contadores para fechar à matemática do uso de memória real, onde o (*free* + *active* + *inactive* + *wired* + *ubc* = memória total), mas não nos dá a informação para o uso de memória virtual.

O *swapon* serve para gerenciar a ativação da memória virtual, o uso pode ser alterado passado o parâmetro '-s', que nos fornece o uso da memória virtual, e assim podemos fechar a matemática geral do uso de memória física e a virtual.

```
> /sbin/swapon -s
Swap partition /dev/disk/dsk0h:
    Allocated space:      1343986 pages (10.25GB)
    In-use space:          221127 pages ( 16%)
    Free space:           1122859 pages ( 83%)


Total swap allocation:
    Allocated space:      1343986 pages (10.25GB)
    In-use space:          221127 pages ( 16%)
    Available space:      1122859 pages ( 83%)
```

Saída 5.3 swapon -s

A informação fornecida pelo *swapon*, como podemos ver na Saída 5.3, é bem direta, são fornecidos o uso e o espaço livre de todas as partições de *swap*.

Para obter a informação de utilização de I/O do disco temos que utilizar o *iostat*. Como parâmetro necessário, é preciso que seja informado qual disco deve ser monitorado. Existe o limite de até três discos poderem ser monitorados simultaneamente.

```
> iostat dsk0 1
     tty          dsk0         cdrom0           cpu
 tin tout    bps    tps    bps    tps  us ni sy id
   0   30    256     13      0      0   3  0  5 92
   0   52      0      0      0      0   0  0  0100
   0   52    304     28      0      0   0  0  1 99
```

Saída 5.4 iostat

Na Saída 5.4 sob o grupo *dsk0*, temos os dados de *bps* (bytes por segundo) e *tps*(transações por segundo) onde são mostrados as atividades correntes de I/O no dispositivo.

Um detalhe importante a ser observado, o *iostat* nos dá à mesma informação que conseguimos aproveitar do *vmstat* sem o parâmetro '-P'.

O *uptime* é o programa responsável por mostrar a carga no escalonador de processos.

```
> uptime
09:17  up 24 days, 15:39,  3 users,  load average: 0.09, 0.06, 0.06
```

Saída 5.5 uptime

Concluindo, podemos amostrar os dados básicos de desempenho através do *vmstat*, *iostat*, *swapon* e *uptime*, com eles conseguimos tudo que necessitamos para obter o básico dos componentes CPU, I/O e memória.

Incluir as informações que o *ps e o top* nos oferecem não pertence ao escopo do que o Orgaperf se propõe a fazer, que é armazenar dados baseados em séries de tempo.

O *ps* e o *top* são capazes de fornecer qual o processo que mais consome CPU no momento corrente e geralmente esses processos não costumam ser os mesmos.

5.1.2. Dados de desempenho do OPSC

Os dados de desempenho do OPSC são obtidos através de arquivos de log, conhecido como arquivos de *Protokoll*.

O *Protokoll* guarda em um arquivo único todos os logs de todos os subsistemas do OPSC, lá é possível ter informações sobre a quantidade de tickets processados, tempo de resposta de comando enviados à plataforma de mediação, número de recargas realizadas, entre outras.

O que é mais importante extrair do *Protokoll* é a quantidade de tickets processados. Tickets processados equivalem à quantidade de ligações que foram tarifadas pelo OPSC.

O comportamento do volume de ligações segue um padrão ao longo do dia e ao longo da semana. Por exemplo, são esperados as seguintes variações de volume:

- Aumento próximo ao meio dia, por ser o horário onde as pessoas costumam se comunicar para se encontrar;
- Aumento próximo às 18 horas, por ser o horário que as pessoas começam a sair do trabalho.
- As sextas feiras o volume de ligações é maior que o considerado normal ao longo da semana.

Por esse raciocínio poderíamos validar se o OPSC está seguindo esta tendência e deduzir que seu funcionamento está dentro do esperado.

As linhas do *Protokoll* que são importantes obter são similares as que estão na Saída 5.6.

```
opsc_bil4   :2312
175104:I:INCDRMCC;MBLM2;9181429739;9137293467;20061223164821;37;22.1463;0.7630;
724029002742637;00024;Single;20061223175104;0.000000;0.0000;0.000000;2;1;VCFIXO
;0.0000;20061018114528;0.7630;0.000000;GSMTACS_0;14155808;0000000011011000000000
00000100000;N;N;900;NNNY00PF0148RD;;1;0;0;0;0;0;VCFIXORN2I30;0;0;0;0;0;0;0;0;
0;0;;;;;;
opsc_bill_sc:2312
175059:I:INCDRMOC;MSPO2;1981768170;1141583332;20061223174831;29;41.9129;0.6700;
724031104960573;00001;Single;20061223175059;none;011;1;0.000000;0.0000;0.000000
;2;1;VCFIXO;NNNY00PN0148NM;14155808;0.0000;0;20060203160656;0.6700;0.000000;GSM
TACS_0;00;    ;0;0;0;0;0;00000000110110000000000000100000;VCFIXOSP1;0;0;0;0;0;0
;0;0;0;0;;;;;;
```

Saída 5.6 Protokoll

Em cada linha do *Protokoll* temos detalhes sobre um determinado ticket, entre eles podemos destacar o:

- **INCDRMCC** Chamada a cobrar para o cliente, via *hot billing*
- **INCDRMOC** Chamada originada no cliente, via *hot billing*
- **INCDRMTC** Chamada terminada no cliente, via *hot billing*
- **INSS7CF** Chamada siga-me para o cliente, via *IN*
- **INSS7MCC** Chamada a cobrar para o cliente, via *IN*
- **INSS7MOC** Chamada originada no cliente, via *IN*
- **INSS7MTC** Chamada terminada no cliente, via *IN*
- **MOSC** *short message* originado no cliente
- **MTSC** *short message* terminado no cliente

Vale lembrar que *IN* trata-se da rede inteligente, nesse tipo de chama a operação de tarifar o cliente acontece em tempo real, quer dizer, à medida que o cliente fala ao telefone. *Hot billing* é a chamada que é tarifada apenas ao término da chamada. O que determina se vai ser feito uma chamada via *IN* ou *hot billing* é o saldo disponível no momento de executar a chamada.

5.1.3. Dados de desempenho de FIFOs

FIFO para o OPSC nada mais é que um diretório com determinados arquivos de controle, na Saída 5.7 mostramos o conteúdo de uma fila FIFO.

```
> find NetReqFIFO
NetReqFIFO
NetReqFIFO/IN
NetReqFIFO/OUT
NetReqFIFO/ANSTOSS
NetReqFIFO/FIFOCONFIG
NetReqFIFO/SERVER
NetReqFIFO/000001.dat
NetReqFIFO/000002.dat
```

Saída 5.7 Estrutura de uma FIFO

Para saber o tamanho de uma fila FIFO basta subtrair o valor do que está armazenado no arquivo *IN* pelo valor do arquivo *OUT*. Como podemos verificar na Saída 5.8.

```
$ echo $[ $(cat NetReqFIFO/IN ) - $(cat NetReqFIFO/OUT ) ]
0
```

Saída 5.8 Verificando o tamanho de uma FIFO

O comando em shell acima abre o arquivo *IN* e guarda o valor em memória, abre o arquivo OUT e guarda o valor em memória, subtrai o valor *IN* do valor *OUT* e resulta no tamanho da fila.

Por ser de primordial importância para o OPSC, as filas FIFO foram feitas de forma a serem realmente simples.

5.2. Como guardar esses dados de desempenho

Os dados de desempenho podem ser mantidos de várias maneiras, neste projeto foram levados em consideração duas bibliotecas para realizar tal trabalho, o RRDtool e a dupla SQLite/GDChart.

O RRDtool é uma biblioteca especializada em gerenciar dados baseados em séries de temporais, como tráfego de rede, temperatura, consumo de CPU, carga no

escalonador de processos, etc. Os dados são salvos em bases de dados circulares o que garante que a quantidade de recursos necessários são sempre constantes e reduzidos.

Outra forma de realizar a mesma tarefa do RRDtool mas de forma mais complicada seria utilizar um banco de dados relacional, de preferência de baixo requisito de recursos como o SQLite.

Porém, utilizar uma base de dados relacionais para salvar dados de desempenho, que podem ser consolidados ao longo do tempo, requer um esforço muito grande para ser implementado, pois necessitaria de um procedimento sendo executando em intervalos regulares para consolidar esses dados.

Além do esforço de implementação do procedimento de consolidação, seriam utilizados recursos extras para tal tarefa, e, para garantir sempre um tamanho compacto da base de dados, dados antigos seriam apagados e conseqüentemente a base de dados iria se fragmentar, precisando de manutenção regular.

Outro ponto que deve ser considerado é que o SQLite só realizaria o armazenamento de dados, necessitando de outro componente para gerar os gráficos com os dados armazenados.

Esses dados poderiam ser criados com pouco esforço utilizando a biblioteca GDChart, que é capaz de criar uma infinidade de gráficos em diversos formatos.

Somando todo esforço necessário entre utilizar o RRDtool ou a dupla SQLite e GDChart, optou-se por utilizar o RRDtool por ser uma biblioteca própria para solucionar o objetivo da ferramenta e por ter outras ferramentas parecidas a utilizando.

5.3. Sobre o RRDtool

Entende-se por *round robing* um *buffer* circular. *Buffer* circular é um *buffer* cujo conteúdo é escrito e lido de maneira circular, ou seja, a última posição do *buffer* é sucedida pela primeira. Assim, um algoritmo de leitura que leia a última posição do

buffer e necessite continuar lendo irá retornar ao início do *buffer* e proceder à leitura a partir daí. O mesmo vale para algoritmos de escrita, sendo que a escrita numa posição não-vazia provoca a perda do conteúdo original.

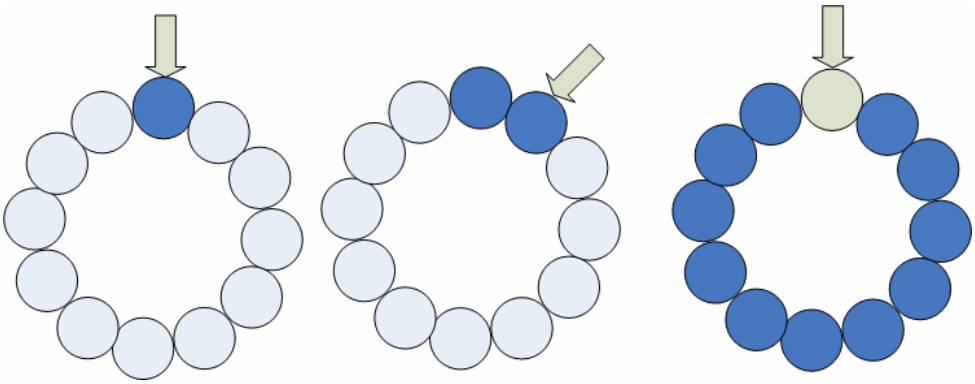


Figura 5.2 Funcionamento de um buffer circular

A Figura 5.2 está exibindo um ciclo de operação de um *buffer* circular, o estado inicial, a segunda operação e o que acontece quando o ponteiro para a posição corrente faz uma volta completa, apagando o campo com o dado mais antigo, sempre reutilizando os recursos mais antigos.

Um RRD ou *round robing database* não é diferente. Cada entrada na base de dados sobrescreve uma entrada mais antiga uma vez que se tenha fechado o ciclo, assim, construindo um círculo grande o suficiente para guardar a quantidade de dados esperado, a quantidade de memória alocada nunca cresce além do que já foi definido, reduzindo a quantidade de manutenção pra manter esse tipo de base de dados.

Um tipo de implementação de um RRD é a solução de Tobias Oetiker, com o software RRDtool. Esse software é uma generalização de outro software do mesmo autor de nome MRTG (*Multi Router Traffic Grapher*).

Um exemplo do uso do MRTG pode ser visto na Figura 5.3.

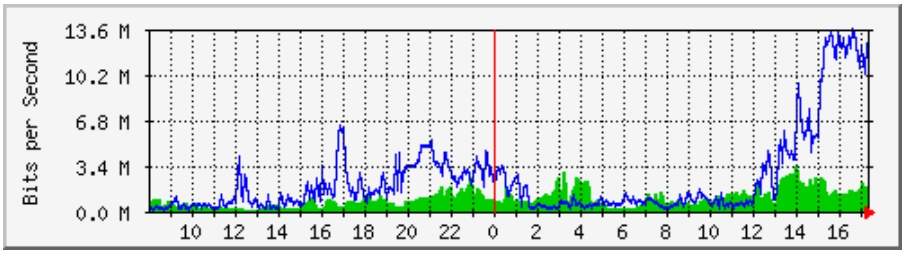


Figura 5.3 Utilização de um link usando o MRTG

O RRDtool é considerado uma generalização do MRTG por ser capaz de manipular qualquer dado que seja baseado no tempo e não apenas utilização de banda como o MRTG. Com o ele é possível guardar dados como a temperatura de um lugar, velocidade de um carro, fluxo de corrente elétrica, tudo ao longo do tempo, gerando séries temporais de dados.

A melhor forma de mostrar o que é o RRDtool é mostrando um exemplo do que ele é capaz de fazer. Vamos usar o exemplo de um carro, imagine que você está dirigindo um carro. As 12:05 você vê no odômetro do carro que o carro percorreu 12.345 quilômetros até o momento. As 12:10 você observa novamente e a leitura é 12.357. Isso significa que você percorreu 12 quilômetros em cinco minutos.

Foram percorridos 12 quilômetros que significam 12000 metros e essa distância foi percorrida em cinco minutos ou 300 segundos, a velocidade poderia ser calculada como 12000 metros / 300 segundos ou 40 metros/segundo.

Poderíamos também calcular a velocidade em quilômetros por hora: 12 vezes 5 é 1 hora, então se multiplicarmos 12 por 12 teríamos 144 quilômetros por hora.

A idéia de medir velocidade a partir da quilometragem e tempo não é nada diferente de medir a velocidade de um link medindo a quantidade de bytes recebidos ou enviados, velocidade de tickets processados a partir da quantidade total de tickets já processados, o que diferencia uma medida de outra é a forma de como coletar os dados.

Para armazenar os dados de quilometragem que foi observado acima, poderíamos criar um RRD usando o RRDtool da seguinte forma:

```
rrdtool create test.rrd            \
        --start 920804400          \
        DS:speed:COUNTER:600:U:U   \
        RRA:AVERAGE:0.5:1:24       \
        RRA:AVERAGE:0.5:6:10
```

Com esse comando foi criado um RRD chamado **test.rrd** que inicia ao meio dia de 7 de março de 1999 (essa data significa 920806440 segundos desde 1º. de janeiro de 1970 UTC, padrão *Unix timestap*). Esse RRD guarda uma única fonte de dados (DS: *datasource*) de nome **speed** que representa um contador.

Agora podemos adicionar dados no RRD da seguinte forma, <*unix timestamp*>:<*valor do contador*>, por exemplo:

```
rrdtool update test.rrd 920804700:12345 920805000:12357 920805300:12363
rrdtool update test.rrd 920805600:12363 920805900:12363 920806200:12373
rrdtool update test.rrd 920806500:12383 920806800:12393 920807100:12399
rrdtool update test.rrd 920807400:12405 920807700:12411 920808000:12415
rrdtool update test.rrd 920808300:12420 920808600:12422 920808900:12423
```

Existe mais de uma forma de conseguirmos descarregar os dados armazenados em um RRD, uma dessas formas e a mais utilizada é montar um gráfico com o *datasource* desejado. Por exemplo:

```
rrdtool graph speed.png                          \
        --start 920804400 --end 920808000        \
        DEF:myspeed=test.rrd:speed:AVERAGE       \
        LINE2:myspeed#FF0000
```

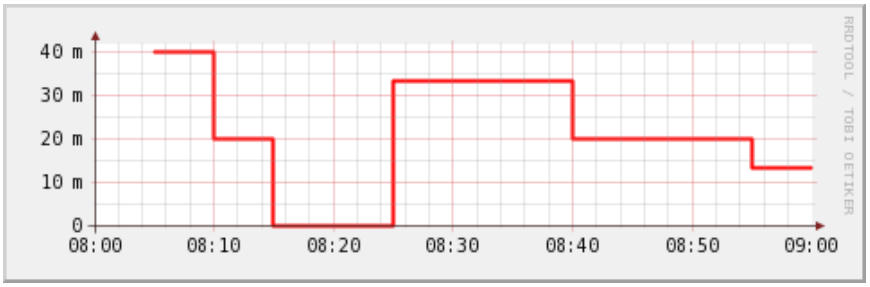


Figura 5.4 Resultado do processo anterior

Na Figura 5.4, temos o resultado do processo de descarga dos dados do RRD em forma gráfica, mas podemos usar mais recursos do RRD para dar mais riqueza ao resultado produzido.

Podemos realizar alguns cálculos com o *datasource* e em seguida por uma legenda no gráfico, por exemplo:

```
rrdtool graph speed2.png                    \
   --start 920804400 --end 920808000        \
   --vertical-label m/s                     \
   DEF:myspeed=test.rrd:speed:AVERAGE       \
   CDEF:realspeed=myspeed,1000,\*           \
   LINE2:realspeed#FF0000
```

O resultado esperado está na Figura 5.5.

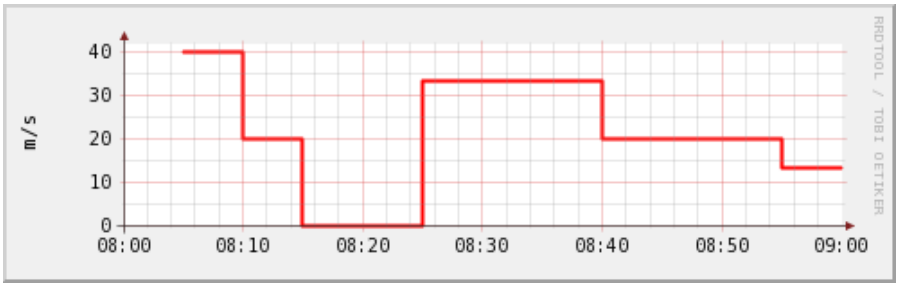


Figura 5.5 Gráfico com legenda

Podemos fazer o gráfico se comportar de maneira distinta em alguns pontos do gráfico, por exemplo:

```
rrdtool graph speed3.png                    \
     --start 920804400 --end 920808000      \
     --vertical-label km/h                  \
     DEF:myspeed=test.rrd:speed:AVERAGE     \
     "CDEF:kmh=myspeed,3600,*"              \
     CDEF:fast=kmh,100,GT,kmh,0,IF          \
     CDEF:good=kmh,100,GT,0,kmh,IF          \
     HRULE:100#0000FF:"Maximum allowed"     \
     AREA:good#00FF00:"Good speed"          \
     AREA:fast#FF0000:"Too fast"
```

O Resultado pode ser apreciado na Figura 5.6.

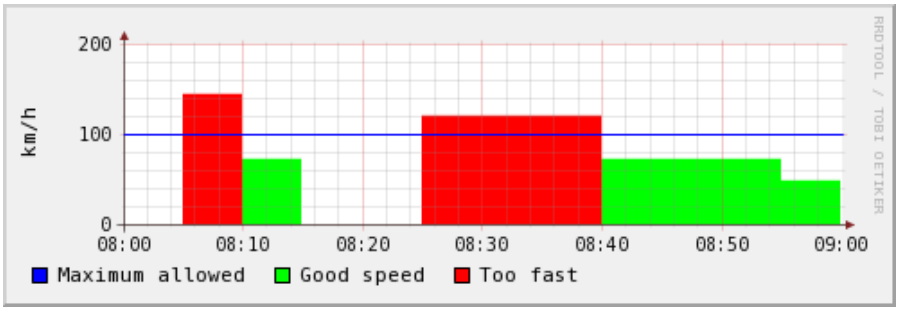


Figura 5.6 Melhorando a visualização

Virtualmente não existem limites no que pode ser feito nos gráficos que o RRDtool é capaz de criar, através de um simples *datasource* de velocidade é possível acrescentar uma quantidade grande de recursos para melhorar a leitura e a beleza do gráfico gerado. Vamos usar mais um exemplo:

```
rrdtool graph speed4.png                          \
   --start 920804400 --end 920808000              \
   --vertical-label km/h                          \
   DEF:myspeed=test.rrd:speed:AVERAGE             \
   "CDEF:kmh=myspeed,3600,*"                      \
   CDEF:fast=kmh,100,GT,100,0,IF                  \
   CDEF:over=kmh,100,GT,kmh,100,-,0,IF            \
   CDEF:good=kmh,100,GT,0,kmh,IF                  \
   HRULE:100#0000FF:"Maximum allowed"             \
   AREA:good#00FF00:"Good speed"                  \
   AREA:fast#550000:"Too fast"                    \
   STACK:over#FF0000:"Over speed"
```

O Podemos ver o resultado na Figura 5.7.

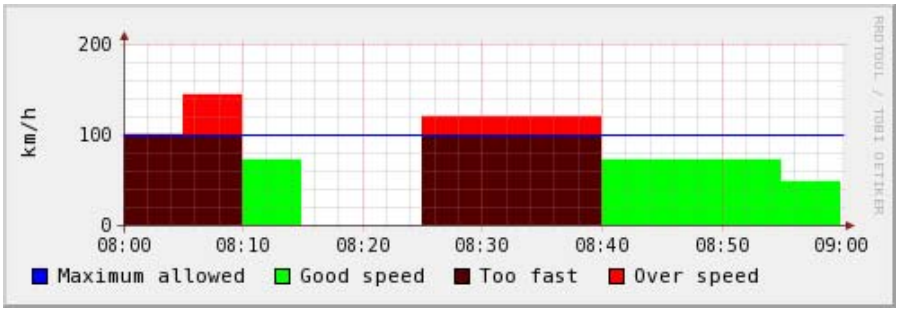


Figura 5.7 Melhorando ainda mais a visualização

5.4. Armazenando dados de desempenho com o RRDtool

Antes de armazenar dados em um RRD é preciso criar o RRD para armazenar esses dados, este é o momento onde deve existir uma análise bem detalhada sobre o que é preciso guardar.

Um RRD como explicado anteriormente, é um *buffer* circular, mas na verdade um RRD é um pouco mais complexo que um simples *buffer* circular, é como se fossem vários *buffers* circulares, onde cada entrada no buffer é um *datapoint*, ou seja, uma amostra do dado. Outro conceito dos RRDs é de consolidação dos dados, assim, cada *buffer* circular do RRDtool, chamado de RRA (*Round Robing Archive*), recebe *datapoints* dos RRAs anteriores. Esse conceito é mais bem descrito na Figura 5.8.

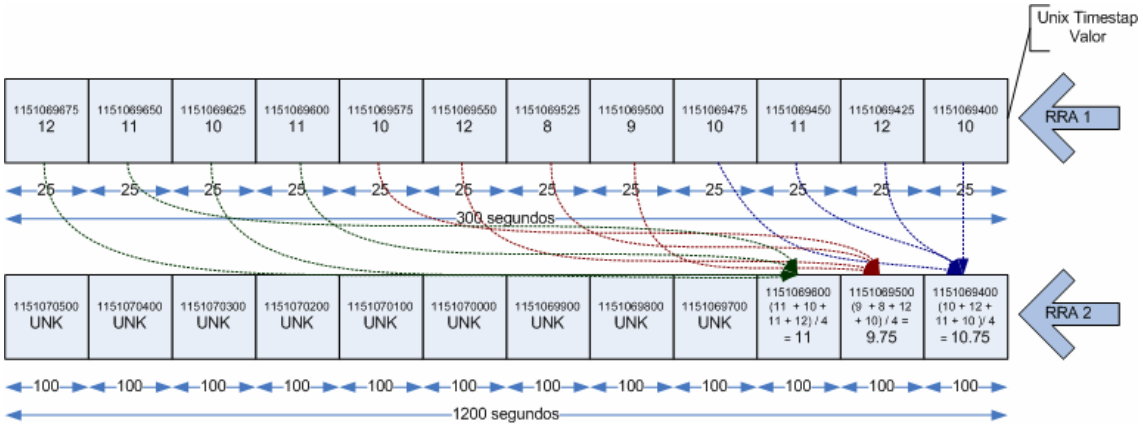


Figura 5.8 Estrutura de RRAs em um RRD

Na Figura 5.8 temos dois RRAs, o primeiro RRA tem um *step* de 25 segundos e tem 12 *rows*, o que totaliza nesse RRA 300 segundos de dados armazenados. O segundo

RRA tem um *step* de 100 segundos e tem 12 *rows*, mesmo número de *rows* do primeiro RRA, mas não é necessário ser sempre igual. O segundo RRA totaliza 1200 segundos de dados.

É importante salientar que a idéia do *step* é o mesmo que de um Δt, que é o infinitésimo de tempo. Os RRAs são alimentados em seqüência, quer dizer, um *step* do segundo RRA equivale a quatro *steps* do primeiro RRA, assim, quando quatro *steps* do primeiro RRA forem preenchidos, um *step* do segundo RRA será preenchido automaticamente. Quanto menor o *step* usado maior vai ser a velocidade que o RRD vai precisar ser atualizado e mais preciso vai ser a resolução do dado amostrado. Porém, cada *datapoint* consome cerca de 8 bytes, por exemplo, se for necessário armazenar um dado de 1 ano inteiro com um *step* de 1 segundo, gastaríamos:

3600 seg x 24 horas x 365 dias x 8 bytes = 252.288.000 bytes

Para reduzir esse tamanho temos duas opções: aumentar o *step* ou usar de mais de um *RRA*. O período onde é mais importante ter maior resolução é justamente o mais recente, quando mais tempo passar menos importante é saber qual foi à amostra em um determinado segundo e mais importa a média em um intervalo de tempo.

A origem do argumento do parágrafo anterior é que desejamos mostrar esses dados em forma de gráficos com determinadas janelas de tempo, por exemplo, últimos 15 minutos, última hora, último dia, última semana,...

Um gráfico com qualidade razoável para ser posto em um relatório pode ter 640x480 pixels, assim teríamos 640 infinitésimos para serem plotados no gráfico. Para um gráfico de 15 minutos teríamos 15 x 60 = 900 infinitésimos, logo um RRA de 900 posições com um *step* de 1 segundo é o necessário para não perder informação devido à interpolação.

Agora, se fosse necessário imprimir os dados da última hora, seria preciso um *step* de 3600 ÷ 640 =~ 5 segundos. Com esse cálculo bastam 640 infinitésimos com *step* de 5 segundos para preencher o gráfico sem perder informação devido à interpolação.

Assim, para economizar espaço, poderíamos ter um RRA que armazenasse os dados na forma mais precisa possível, mas em um espaço reduzido de tempo, digamos 30 minutos ou 1800 segundos, e outro RRA com o período do ano inteiro, mas com um *step* maior, digamos 300 segundos. Dessa maneira poderíamos refazer as contas da seguinte forma.

60 seg x 30 min x 8 = 14.400
+
(3600 seg x 24 horas x 365 dias)/300 x 8 bytes = 31536000 bytes
= 31.550.400 bytes

O resultado é que reduzimos, em muito, o espaço necessário para armazenar os dados de um ano inteiro. Mas mesmo assim é bem grande para o tamanho de uma base RRD com apenas um dado ou, na linguagem do RRDtool, *datasource* guardado. É possível ter vários *datasources* em um mesmo RRD, mas caso existisse outro *datasource* armazenado no RRD ele ficaria com o dobro do tamanho.

A melhor solução para reduzir o tamanho do arquivo RRD é utilizar vários RRAs no mesmo RRD, mas com diferentes *steps*. Cada RRA com foco em gerar o gráfico de uma determinada janela de tempo sem perder informação devido à interpolação de amostras. Para equilibrar a relação "tamanho x qualidade", chegamos à seguinte matemática:

Para a janela dos últimos 10 minutos
900 *datapoints* com *step* de 1 segundo = 15 minutos

Para a janela da última 1 hora
900 *datapoints* com *step* de 5 segundos = 1 hora e 15 minutos

Para a janela do último dia
900 *datapoints* com *step* de 150 segundos = 1 dia e 12 horas

Para a janela da última semana
900 *datapoints* com *step* de 900 segundos = 9 dias e 9 horas

Para a janela do último mês

900 *datapoints* com *step* de 4000 segundos = 41 dias e 15 horas

Para janela dos últimos 6 meses

900 *datapoints* com *step* de 24000 segundos = 8 meses e 9 dias

Para a janela do último ano

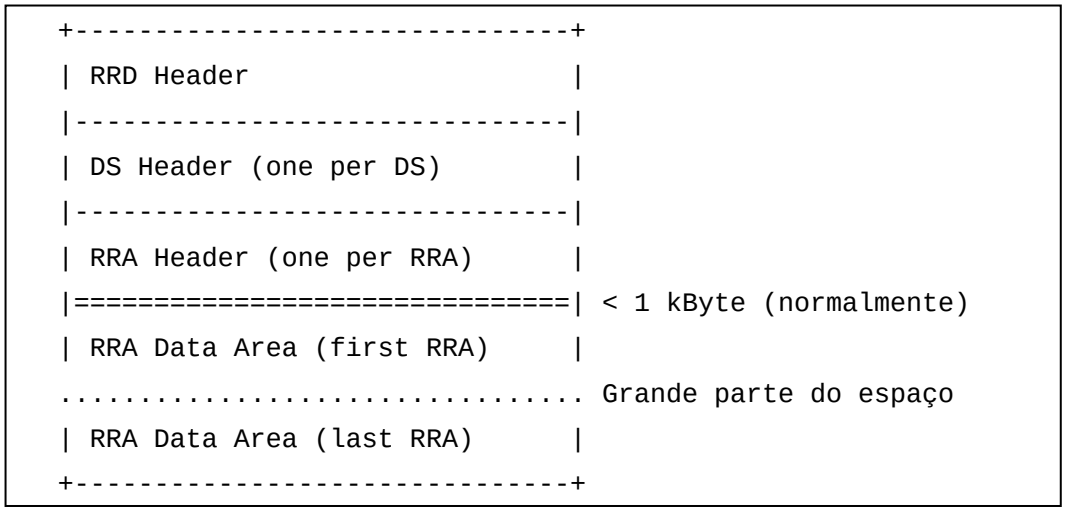
900 *datapoints* com *step* de 48500 segundos = 16 meses e 25 dias.

Os valores dos RRAs foram escolhidos de forma a ter uma margem de folga razoável para deslocar a janela de tempo um pouco mais pra trás do que foi originalmente estipulado.

A conta do consumo de espaço por *datasource* com todos esses RRAs é:

7 x 900 x 8 = 50.400 bytes.

Os cabeçalhos e de um RRD pode ser desprezados por serem de tamanho ínfimo em relação ao espaço ocupado pelos RRAs, podemos verificar a estrutura um RRD a seguir:



Saída 5.9 Estrutura de um RRD

Sendo os cabeçalhos por volta de 1kbyte, grande parte do espaço é ocupado pelos RRAs, assim, podemos tomar como base que é preciso de cerca de 51kbytes por *datasource* armazenado.

Por conter um *overhead* pequeno, sempre que possível, o objetivo será usar múltiplos RRDs com poucos *datasources* ao invés de um único RRD com muitos *datasources*. Assim caso um arquivo se corrompa, não será perdido tudo que foi amostrado.

Agora, com a estrutura do RRD definida, basta criar-los com o RRDtool *create* e utilizar o RRDtool *update* para inserir os dados amostrados.

5.5. Exibindo dados de desempenho com o RRDtool

Uma característica importante ao RRDtool é poder realizar a amostragem dos *datasources*, se preocupando primeiro em guardar o que é importante e, só depois, se preocupar em como exibir o que foi amostrado.

Realizar o processo de amostragem e armazenamento é a parte mais simples de todo o projeto, e o mais importante. Exibir o que está armazenado é onde requer mais paciência.

Nesse momento é importante voltar à seção 5.1, onde foi definido que seria armazenado no RRDtool.

- **Coleta de dados de desempenho básicos**
  - Uso de CPU
  - Uso de memória
  - Uso de paginação
  - Uso de recursos de Entrada/Saída
  - Carga sobre o escalonador de processos
- **Coleta de dados de desempenho do OPSC**
  - Tickets de tarifação processados

- **Coleta de dados de desempenho de FIFOs**
  - Tamanho das FIFOs internas

Um gráfico deve ser confeccionado de forma exibir de forma clara o que foi armazenado no RRD, nessa seção será proposto o formato de exibição de cada dado amostrado para auxiliar a posterior implementação. As figuras abaixo corresponderão às propostas de gráficos.

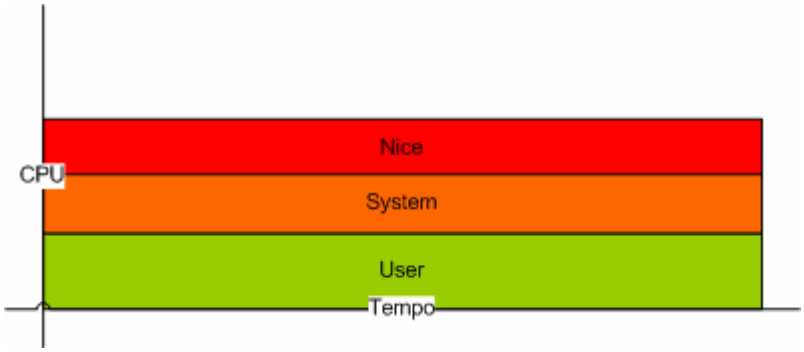


Figura 5.9 Proposta para exibir os dados de CPU

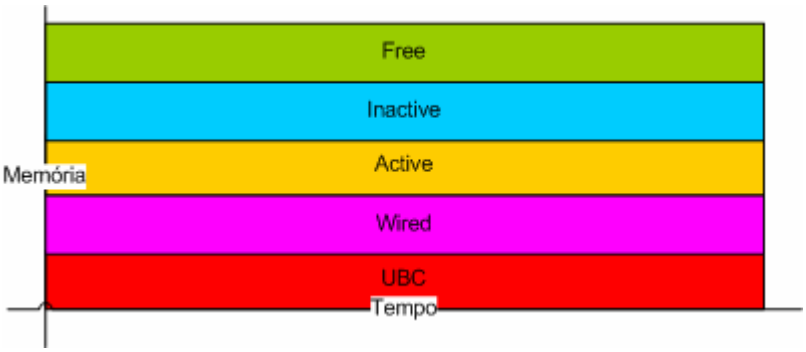


Figura 5.10 Proposta para exibir o uso de memória Física

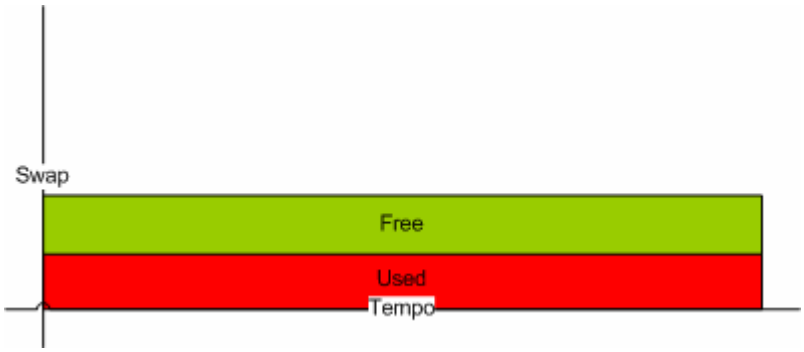


Figura 5.11 Proposta para exibir o uso de memória virtual

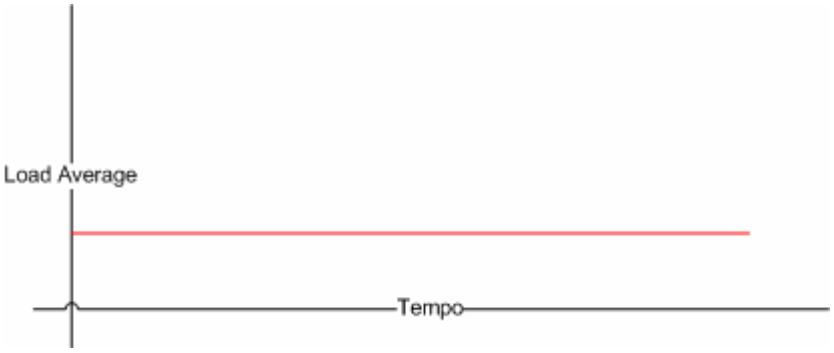


Figura 5.12 Proposta para exibir a carga sobre o escalonador de processos

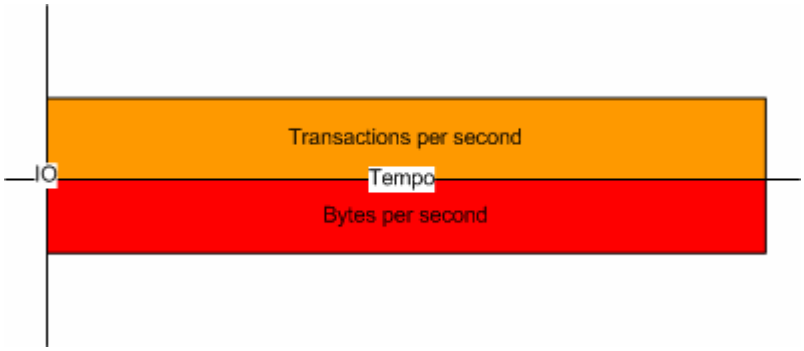


Figura 5.13 Proposta para exibir o uso de recursos de Entrada/Saída

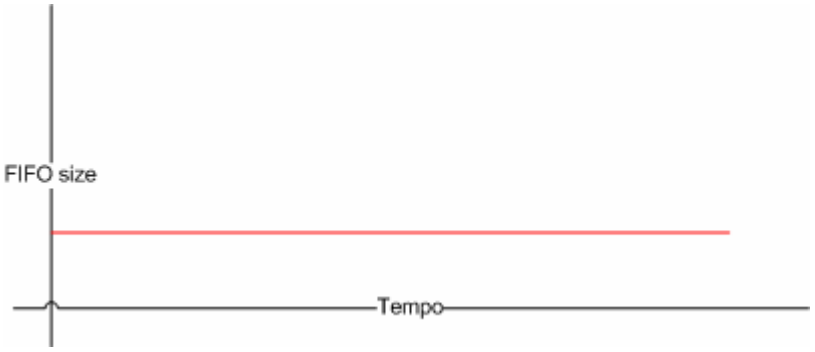


Figura 5.14 Proposta para exibir o tamanho das filas do tipo FIFO

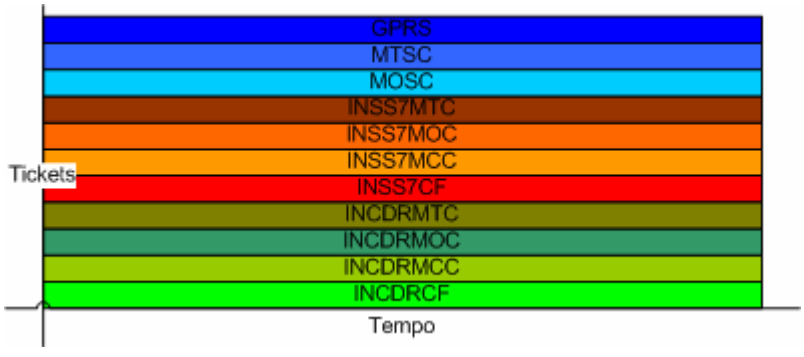


Figura 5.15 Proposta para exibir os tickets processados

6. Arquitetura

A decisão sobre qual arquitetura que um software irá utilizar é um ponto vital no desenvolvimento de um software. Essa etapa definirá a documentação preliminar por meio a qual o software será construído.

Esse é o momento onde definiremos as estruturas do sistema que abrange os componentes de software, as propriedades externas visíveis desses componentes e as relações entre eles.

6.1. Relações do Orgaperf com agentes externos

O software Orgaperf necessita interagir com o engenheiro de serviços, para exibir os dados que tem armazenado, e com a dupla sistema operacional/OPSC, para captar dados. Essa relação é mostrada na Figura 6.1.

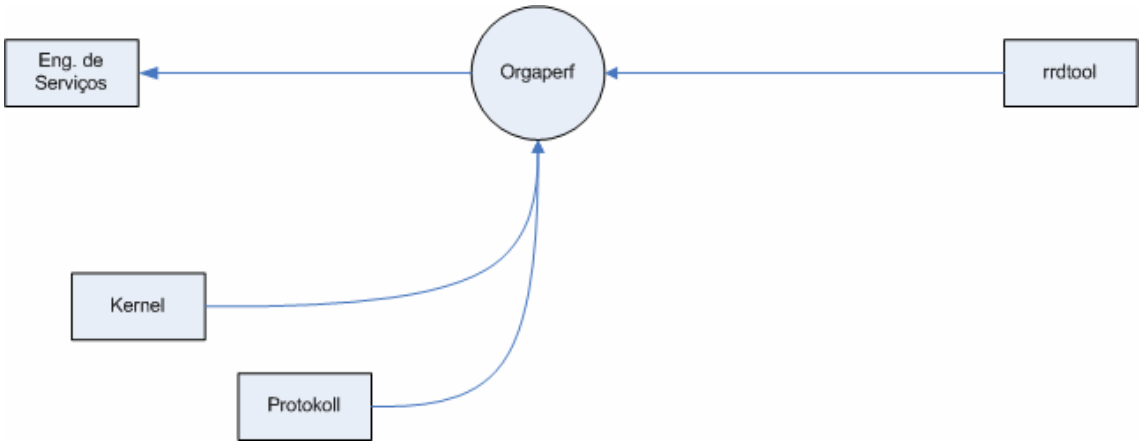


Figura 6.1 Relações do Orgaperf com agentes externos

O sistema operacional é intitulado como *kernel* e o OPSC como *Protokoll*. O *Protokoll* é um arquivo de log, que é reciclado baseado em tamanho, em tempo ou em ambos. Os dados de desempenho são armazenados no RRDtool e o este pode ser considerado como uma entidade responsável por guardar e gerenciar todos os dados de desempenho.

6.2. Principais funções do software Orgaperf

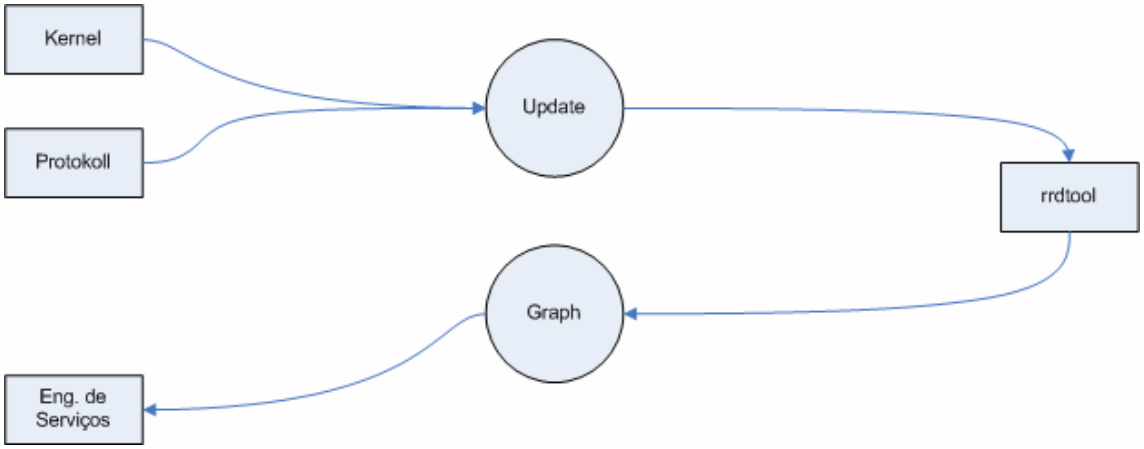


Figura 6.2 Principais funções do software Orgaperf

A Figura 6.2 mostra as duas principais funções realizadas pelo software. **Update** e **Graph**.

A função de Update é utilizada para inserir ou atualizar os dados de desempenho captados para dentro do RRDtool. Já a função Graph é produto de uma ação iniciada por um engenheiro de serviços que gera a reação do software de descarregar os dados de desempenho armazenados no RRDtool.

6.3. Porque a linguagem Perl

Perl é uma linguagem de programação madura e bastante conceituada no ambiente corporativo, originalmente concebida para tratamento de textos, foi criada por Larry Wall em 1987 e até hoje é continuamente desenvolvida e melhorada. Perl une a capacidade de interpretar textos do *AWK* e uma forma de programação estruturada parecida com a linguagem *C*, além de ser interpretada e ter suporte a várias plataformas, como praticamente todas as versões de Unix® e Windows®. Em Perl a alocação de memória é feita de forma dinâmica e transparente ao usuário e é uma das linguagens preferidas dos administradores de sistemas Unix®.

O sucesso do Perl também pode ser explicado por sua capacidade de ser estendido usando módulos, um incontável número de módulos de extensão de Perl estão disponíveis no CPAN (*Comprehensive Perl Archive Network*), e mantidos por uma legião de usuários. Perl também admite criar ligações com outras linguagens através do Perl XS (*eXternal Subroutine*), onde é possível usar bibliotecas disponíveis em outras linguagens, principalmente na linguagem *C*, e as utilizar dentro de scripts Perl de forma transparente.

Outra característica decisiva para escolher Perl como a linguagem ideal para construir o Orgaperf é a facilidade em interagir com as funções básicas do Unix®, como for exemplo, o uso de *fork*s, *threads*, *semaphores*, *shared memory*, acesso a arquivos, comunicação por *BSD Sockets* (TCP/IP) entre outras funções. Perl simplifica grande parte da complicação encontrada em *C* e adiciona a comodidade de ter seus próprios mecanismos de alocação e liberação de memória, garantindo a eliminação dos vazamentos de memória.

Perl também possui uma das melhores implementações do algoritmo *hash* (vetores associativos) do mercado e o *overhead* de execução de um script em Perl, isto é, razão de tempos de execução de um programa compilado e um programa interpretado, é bastante aceitável.

Outro fator determinante para a escolha da linguagem Perl é por vir instalado por padrão nos servidores onde o OPSC funciona. O Perl é instalado por padrão nas instalações do Digital Unix 4.0 e do Tru64 5.1.

6.4. Arquitetura baseada em processo monitor

O software Orgaperf está baseado em uma arquitetura modular, onde cada módulo é independente e é responsável por uma determinada função.

Cada módulo é na verdade um processo Unix®, e a decisão arquitetural escolhida foi implementar o software usando um árvore invertida, onde um módulo pai instância módulos filho, como descrito na Figura 6.3.

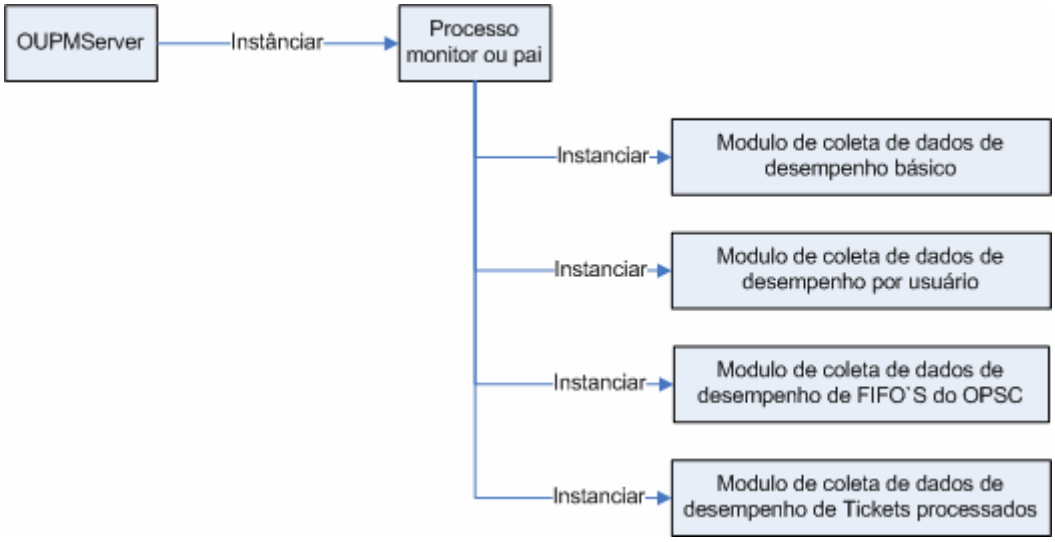


Figura 6.3 Arquitetura baseado em processos

O OUPMServer (Orga Unified Process Management) é um serviço de inicialização de *daemons* (serviços Unix®), o início do processo de inicialização do OPSC é a execução da proctable (arquivo com uma lista de processos) pelo OUPMServer. Ele é responsável por iniciar cada serviço e manter-los ativo, caso seja um serviço intermitente (executa, processa e termina), o OUPMServer pode ser considerado como um serviço de *scheduler*, que reinicializa o serviço em intervalos regulares de tempo.

O OUPMServer também é responsável por canalizar a saída dos programas por ele iniciados para o arquivo de *Protokoll*, assim, escrever no *Protokoll* é meramente escrever na saída padrão (*stdout*).

A função do módulo pai é, além de instanciar os módulos filhos, reiniciar um módulo filho caso algo de errado aconteça com ele, como mostra a Figura 6.4.

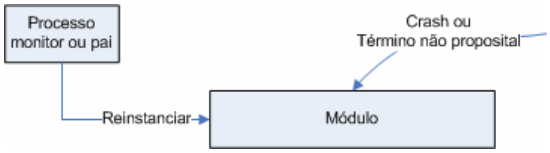


Figura 6.4 Relação módulo pai - módulo filho

O processo pai, uma vez sinalizado por um engenheiro de serviços para descarregar os dados de desempenho, cria um cópia do seu processo e faz chamadas a função *graph* do RRDtool para os diferentes contadores de desempenho, como mostra a Figura 6.5.

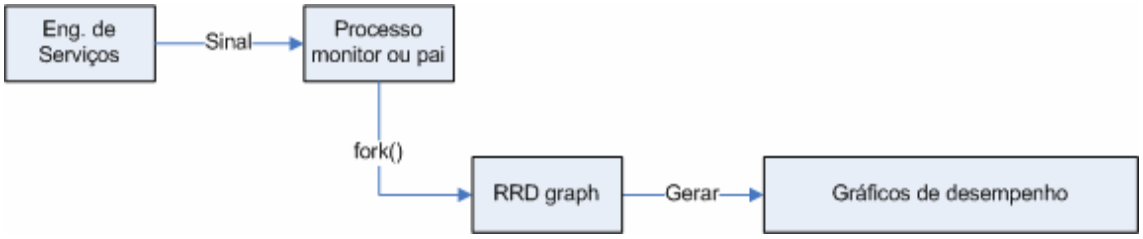


Figura 6.5 Descarga dos dados de desempenho

A finalização do software OPSC é iniciada enviando um pedido de término ao OUPMServer. Ele se encarrega de gerenciar o término de cada processo por ele instanciado. Uma vez que o software Orgaperf recebe o pedido de término do OUPMServer através do sinal apropriado, esse sinal é enviado aos módulos filhos e quando os filhos terminam o módulo pai sinaliza o término ao OUPMServer, como demonstrado na Figura 6.6.

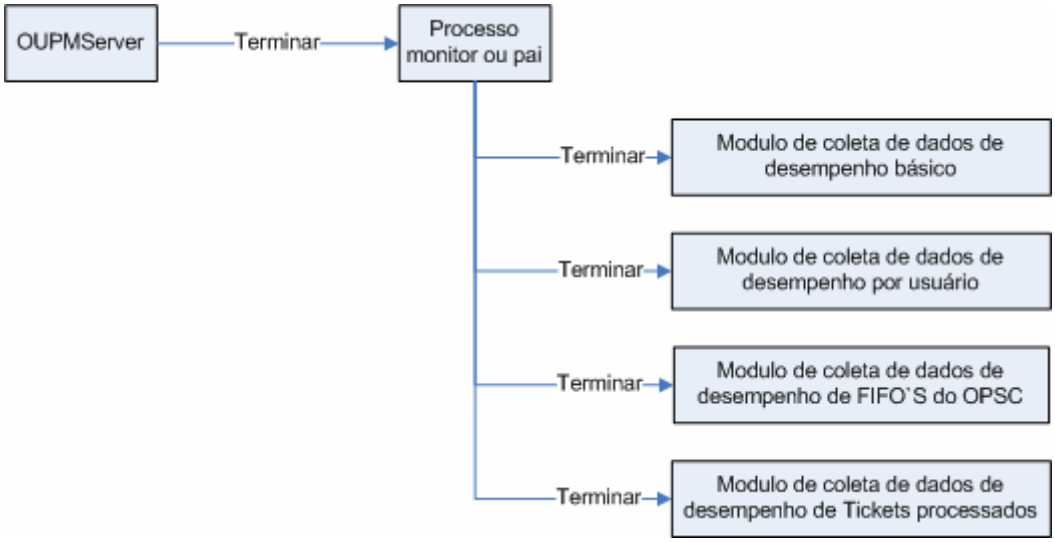


Figura 6.6 Sinalização de término

6.5. Processo de atualização de RRDs

Uma vez instanciados, os módulos filhos tem por tarefa captar os dados de desempenho de sua competência e os atualizar. Esses dados são gravados em arquivos RRD por intermédio do RRDtool, como demonstrado na Figura 6.7.

Cada módulo tem um timer próprio que dispara a cada segundo, gerando um evento de atualização dos dados de desempenho sob seu domínio.

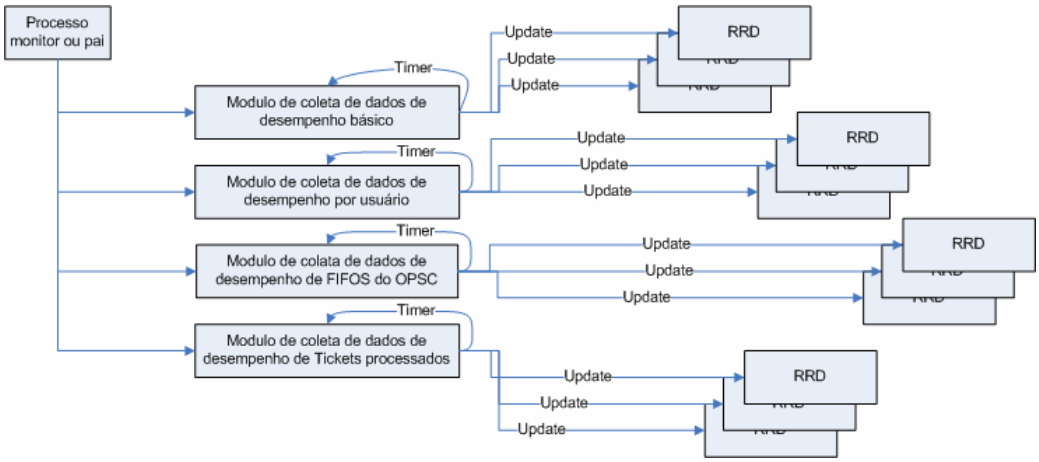


Figura 6.7 Processo de atualização dos RRDs

7. Implementação

Com o processo de análise completado, já existem bases para continuar o projeto e realizar a implementação do Orgaperf. O processo de implementação será dividido nas seguintes fases:

- **Módulo de coleta de dados de desempenho básicos**
  - Amostragem e modelagem gráfica
    - Uso de CPU
    - Uso de memória
    - Uso de paginação (*Swap*)
    - Uso de recursos de Entrada/Saída
    - Carga no escalonador de processos (*load average*)
- **Módulo gerenciador de módulos**
  - Bases para integração com o *OUPMServer*
  - Verificação de aptidão de funcionamento (*sanity check*)
    - Bases RRD criadas e acessíveis
    - Diretórios de trabalho com permissão correta
  - Instanciar filhos na inicialização
  - Procedimento para geração dos gráficos
  - Finalizar os filhos ao término
- **Módulo de coleta de dados de desempenho do OPSC**
  - Amostragem e modelagem gráfica
- **Módulo de coleta de dados de desempenho de FIFOs**
  - Amostragem e modelagem gráfica

7.1. Considerações prévias

Antes do início da implementação do Orgaperf, foi reconsiderada a questão de usar processos externos para realizar a amostragem dos dados. Foi verificado que tal procedimento aumentaria o uso do processador e memória.

Inicialmente foi idealizado utilizar, por exemplo, o *vmstat* para amostrar os dados de utilização da CPU. Para isso o *vmstat* seria instanciado através de um *pipe*, que é uma forma de comunicação inter processos, e seria realizada a leitura desse *pipe*, e a cada linha gerada pelo *vmstat* seria feito uma inserção na base RRD correspondente.

Ocorre que instanciar um processo via *pipe* adiciona-se um peso ao sistema operacional em relação a:

- Mudanças de contexto;
- Interpretar os dados vindos do *pipe*.

Se o Orgaperf fosse um processo de funcionasse esporadicamente, os custos menores de tempo de implementação decorrentes a adoção dessa prática justificaria o aumento do uso de recursos. Mas como será de uso constante, o esforço de estudar uma maneira de evitar os custos maiores de recursos se justifica.

Para evitar utilizar o *vmstat*, *iostat*, *uptime* e o *swapon*, é preciso avaliar como esses programas se comunicam com o sistema operacional para conseguir os dados que são exibidos no console. Como não é possível ter acesso ao sistema operacional, foi necessário observar os cabeçalhos das funções exportadas pelo kernel nos pacotes de desenvolvimento. Esses cabeçalhos ficam no diretório */usr/include*. Após algum tempo de procura foi possível observar algumas coisas interessantes, como podemos verificar na Saída 7.1.

```
#define TBL_SYSINFO      12  /* (no index) */
/*
 *      TBL_SYSINFO data layout
 */
struct tbl_sysinfo {
        long    si_user;        /* User time */
        long    si_nice;        /* Nice time */
        long    si_sys;         /* System time */
        long    si_idle;        /* Idle time */
        long    si_hz;
        long    si_phz;
        long    si_boottime;   /* Boot time in seconds */
        long wait; /* Wait time */
        int     si_max_procs;  /* rpb->rpb_numprocs */
```

Saída 7.1 Cabeçalho /usr/include/sys/table.h

Verificando no Google sobre essa estrutura, foi possível descobrir como ela pode ser utilizada.

```
#include <stdio.h>
#include <sys/table.h>

struct tbl_sysinfo t;
table(TBL_SYSINFO, 0, &t, sizeof(t));

printf ("%d %d %d %d\n", t.si_user, t.si_nice, t.si_system,
t.si_idle);
```

Saída 7.2 Utilizando a função table no Tru64

Ao compilar e executar esse programa, a sua saída correspondia à quantidade de microsegundos utilizados por cada classe, *user*, *nice*, *system* e *idle*. Estas são as mesmas informações fornecidas pelo *vmstat*, porém, sem precisar dele.

A função table() é a chave como todos os processos de análise de desempenho funcionam, com essa mesma função é possível realizar o mesmo trabalho do *swapon* e do *iostat*.

O próximo desafio é como embutir a funcionalidade da função table() no Perl, com conseguir amostrar os dados de desempenho sem precisar chamar programas externos.

7.2. Perl XS

Perl XS é uma interface onde um programa escrito em Perl pode chamar uma sub-rotina em C, sua utilização é bastante desejável quando é necessário invocar funções que utilizam bastante CPU ou memória, ou quando é necessário ter uma interface de baixo nível com o hardware, ou utilizar uma função em C já existente.

Uma vez que um módulo XS é instalado, ele pode ser carregado por um programa Perl como se fosse outro módulo qualquer, assim tornando a função em C disponível para uso pelo código Perl.

Embutir a função table() usando XS é uma forma ideal de evitar chamadas aos programas externos *vmstat*, *iostat*, *uptime* e *swapon*. Porém, como é desejável que o Orgaperf seja portável para outras plataformas, exportar a função table() para dentro do Perl não é a melhor solução, para este caso, o ideal é criar uma camada de abstração para a função table().

Para implementar essa camada de abstração, como o interesse é obter a amostragem do uso de CPU, memória física e virtual, *load average*, e uso de I/O por dispositivos, o melhor é criar funções que chamem da função table() e que retornem os dados de interesse. Assim, para substituir cada programa externo, foram criadas funções que os substituíssem. Na Saída 7.3 é demonstrado como o *vmstat* foi re-implementado.

```
void
cpuinfo ()
      INIT:
            struct tbl_sysinfo t;
            int err;

      PPCODE:
            err = table(TBL_SYSINFO, 0, &t, 1, sizeof(t));
            if (err == -1)
                  XSRETURN_UNDEF;

            EXTEND(SP, 4);
            PUSHs(sv_2mortal(newSViv(t.si_user)));
            PUSHs(sv_2mortal(newSViv(t.si_sys)));
            PUSHs(sv_2mortal(newSViv(t.si_nice)));
            PUSHs(sv_2mortal(newSViv(t.si_idle)));
```

Saída 7.3 Função em Perl XS para substituir o *vmstat*

A estrutura retornada tem todos os elementos para a matemática do uso de CPU, que é *user* + *system* + *nice* + *idle* = 100%. Para utilizar essa função XS em um código Perl é tão simples quanto mostra a Saída 7.4.

```
#/usr/bin/perl
use Orgaperf;

print “CPU “ . join(":", ORGAPerf::cpuinfo()) . “\n”;
```

Saída 7.4 Usando a função XS implementada

Executando esse código temos a Saída 7.5.

```
      CPU 57820277:97469349:0:1982228015
```

Saída 7.5 Resultado da execução da Saída 7.4

O resultado corresponde à quantidade de microsegundos utilizados por cada classe de execução. O resultado pode não ser amigável a primeira vista, mas é perfeito para ser armazenado no RRDtool.

Da mesma forma que foi re-implementado o comportamento do vmstat, variando o primeiro argumento da função table() é devolvido uma estrutura de dados com o que foi pedido. Assim, montando o par argumento x estrutura retornada, foram re-implementados os seguintes comandos externos:

- ***iostat***, argumento TBL_DKINFO
  - A estrutura retorna quantidade de requisições de I/O feitas, tamanho da fila de requisições de I/O e o tempo médio que o *kernel* esperou uma resposta do dispositivo.
- ***vmstat –P***, argumento TBL_VMSTATS
  - A estrutura retorna todas as variáveis para realizar a matemática da memória física, *active*, *inactive*, *wire*, *ubc cache* e *free.*
- ***swapon***, argumento TBL_SWAPINFO
  - A estrutura retorna a quantidade livre e em uso de *swap*.
- ***uptime***, argumento TBL_LOADAVG
  - A estrutura retorna a carga no escalonador, com a média dos últimos 5 minutos, média do último minuto e a carga corrente.

7.3. Módulo de coleta de dados de desempenho básicos

Com a camada de abstração em XS criada, não é mais preciso abrir *pipes* e executar programas externos e nem mais é preciso interpretar a saída desses programas para armazenar no RRDtool.

A implementação agora é uma tarefa mais simples, a Saída 7.6 mostra como foi implementado esse módulo.

```
sub CollectBasicRRDs () {
  my %ioinfo;

  my @entries = grep (/^ioinfo/, keys %filename);
    foreach my $entry (@entries) {
      my $dskname = (split (/-/, $entry))[1];
      my $dsknum = ORGAPerf::lookupdsknum ($dskname);
      $ioinfo{$entry} = $dsknum;
  }
    do {
      foreach my $dsk (keys %ioinfo) {
         my $info = ORGAPerf::monitordsknum($ioinfo{$dsk});
         RRDs::update ($filename{$dsk}->[0], "N:" . join(":", @$info))
           or &Log("E", "CollectBasicRRDs", RRDs::error(), __LINE__);
      }
      RRDs::update ($filename{'cpuinfo'}->[0],    "N:" . join(":", ORGAPerf::cpuinfo()))
         or &Log("E", "CollectBasicRRDs", RRDs::error(), __LINE__);
      RRDs::update ($filename{'realmeminfo'}->[0],"N:" . join (":", ORGAPerf::meminfo()))
        or &Log("E", "CollectBasicRRDs", RRDs::error(), __LINE__);
      RRDs::update ($filename{'swapmeminfo'}->[0],"N:" . join (":", ORGAPerf::vminfo()))
        or &Log("E", "CollectBasicRRDs", RRDs::error(), __LINE__);
      RRDs::update ($filename{'loadavg'}->[0],    "N:" . ORGAPerf::loadinfo())
        or &Log("E", "CollectBasicRRDs", RRDs::error(), __LINE__);
       sleep 1;
    } while (1);
}
```

Saída 7.6 Módulo de coleta de dados de desempenho básico

No início do código, verificamos quais discos precisam ter suas atividades monitoradas, configurável em um arquivo de configuração que será explicado mais a frente, e logo em seguida entramos no *loop* de amostragem.

Inicialmente no *loop* é realizado a operação de *update* do RRDtool com os dados que obtidos com a função XS responsável por substituir o *iostat*, para cada disco. A função join do Perl serve para juntar um *array* ligando cada elemento por uma *string* pré-determinada, que nesse caso é o caractere "**:**". Assim, a função *update* receberá como parâmetro uma *string*, por exemplo: N:10:20:30. O caractere **N** na função *update* é um símbolo coringa, que representa o *Unix timestamp* corrente.

Em seguida são alimentados os dados sobre o consumo de CPU, memória física, memória virtual e carga no escalonador de processos.

Como foi demonstrado anteriormente sobre o RRDtool, é virtualmente ilimitado o que podemos fazer com os gráficos, a Figura 7.1 representa de forma bem similar o que foi proposto na análise.

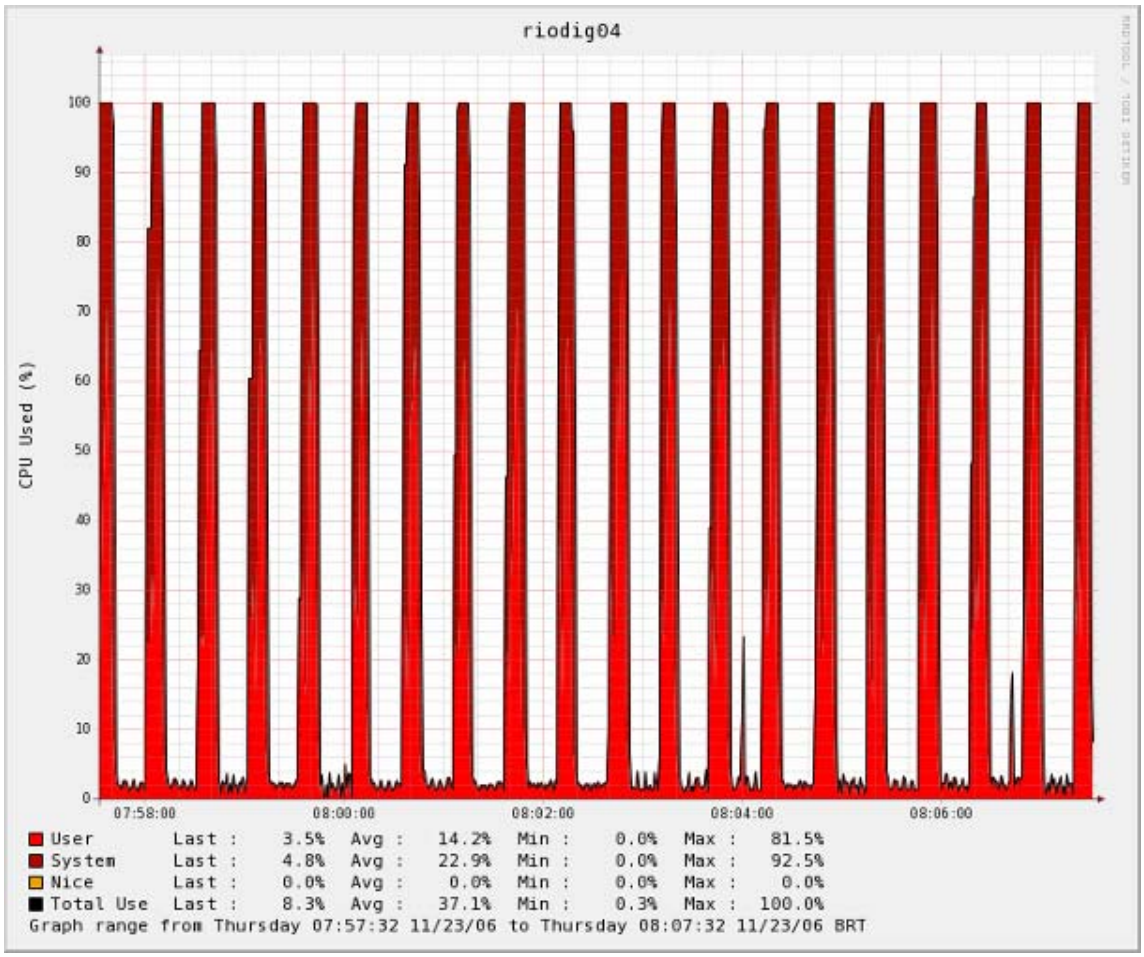


Figura 7.1 Gráfico de consumo de CPU

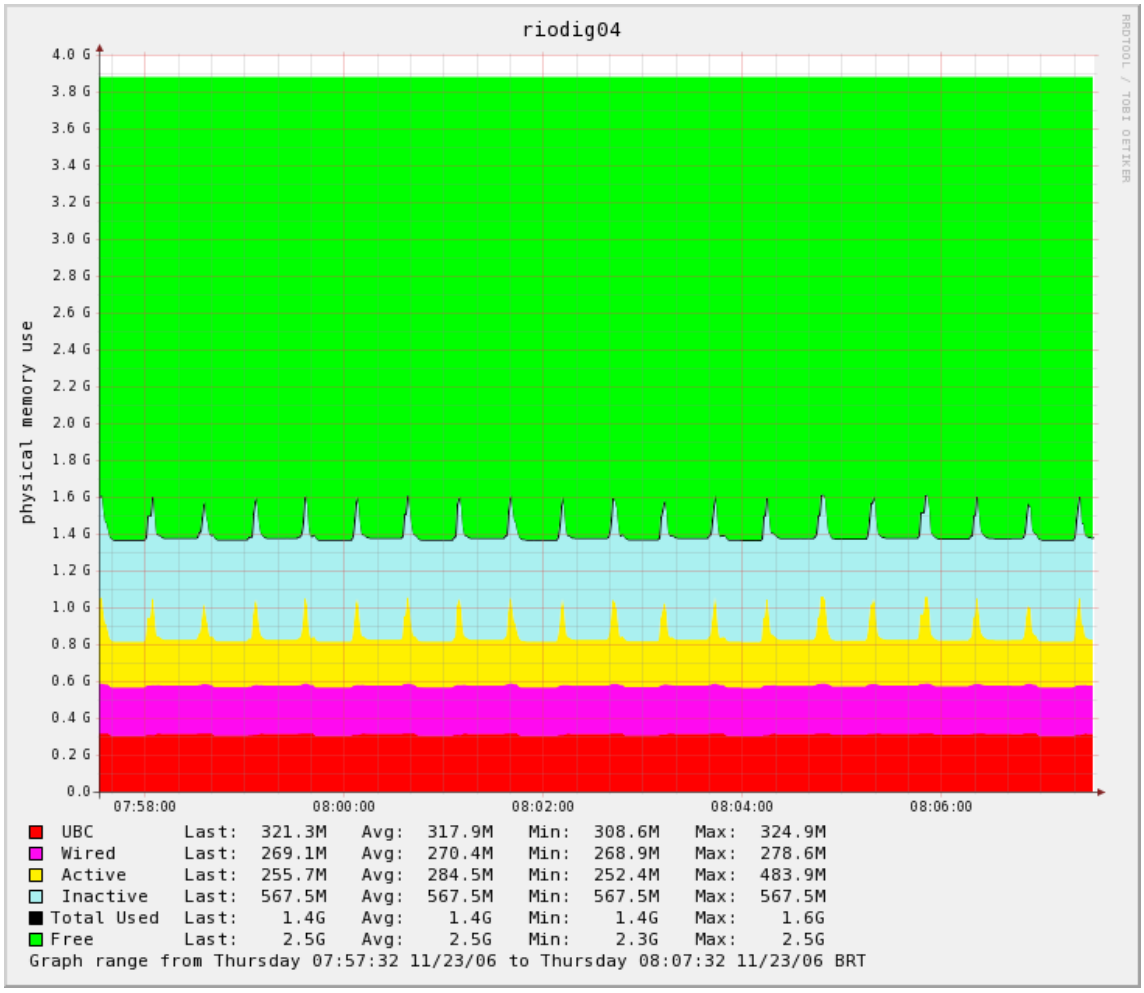


Figura 7.2 Utilização da memória física

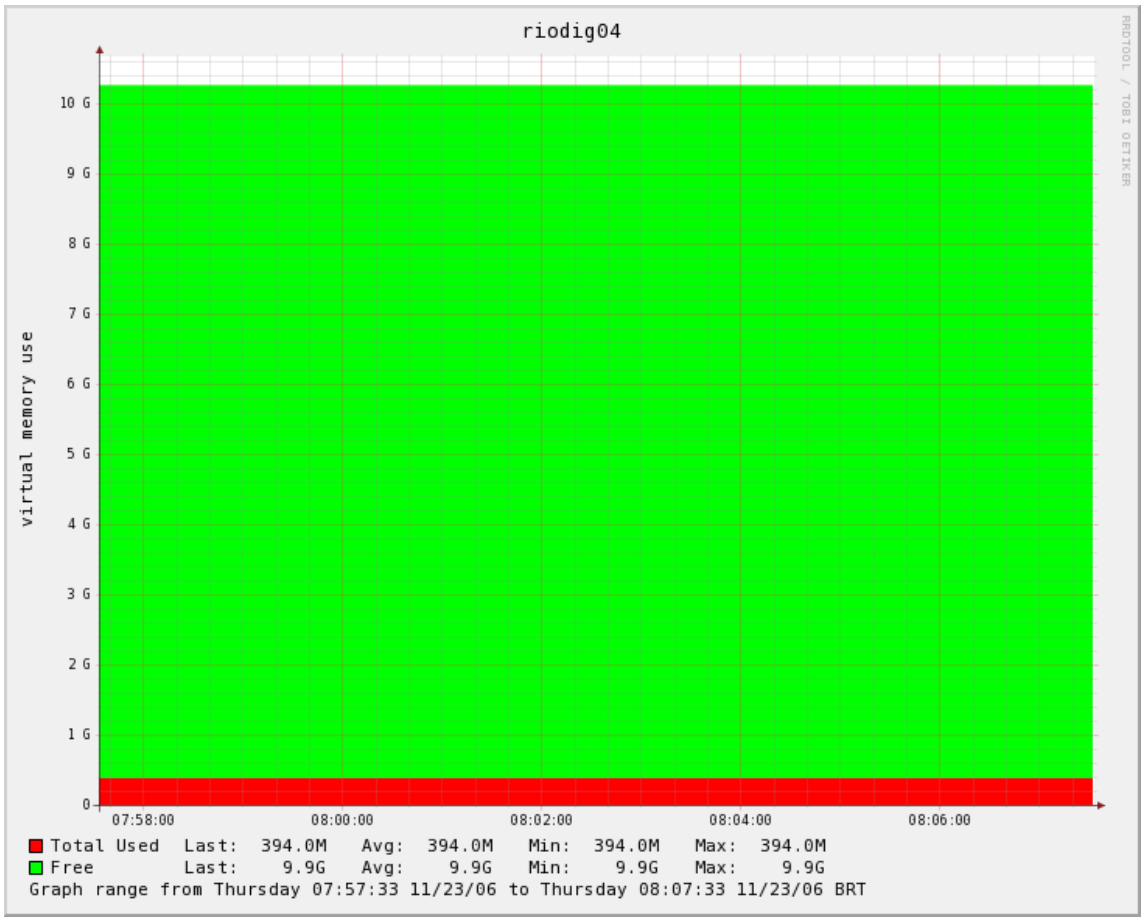


Figura 7.3 Utilização de paginação (*swap*)

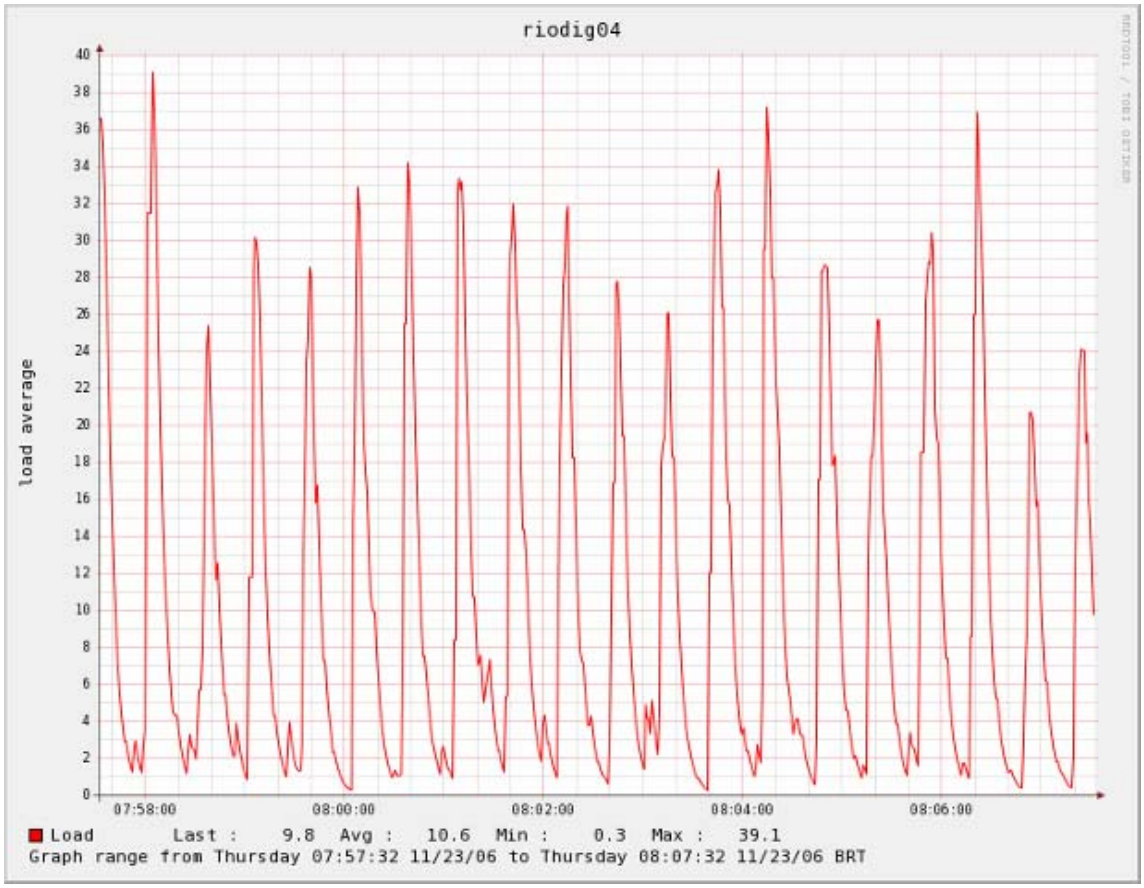


Figura 7.4 Carga no escalonador de processos (*load average*)

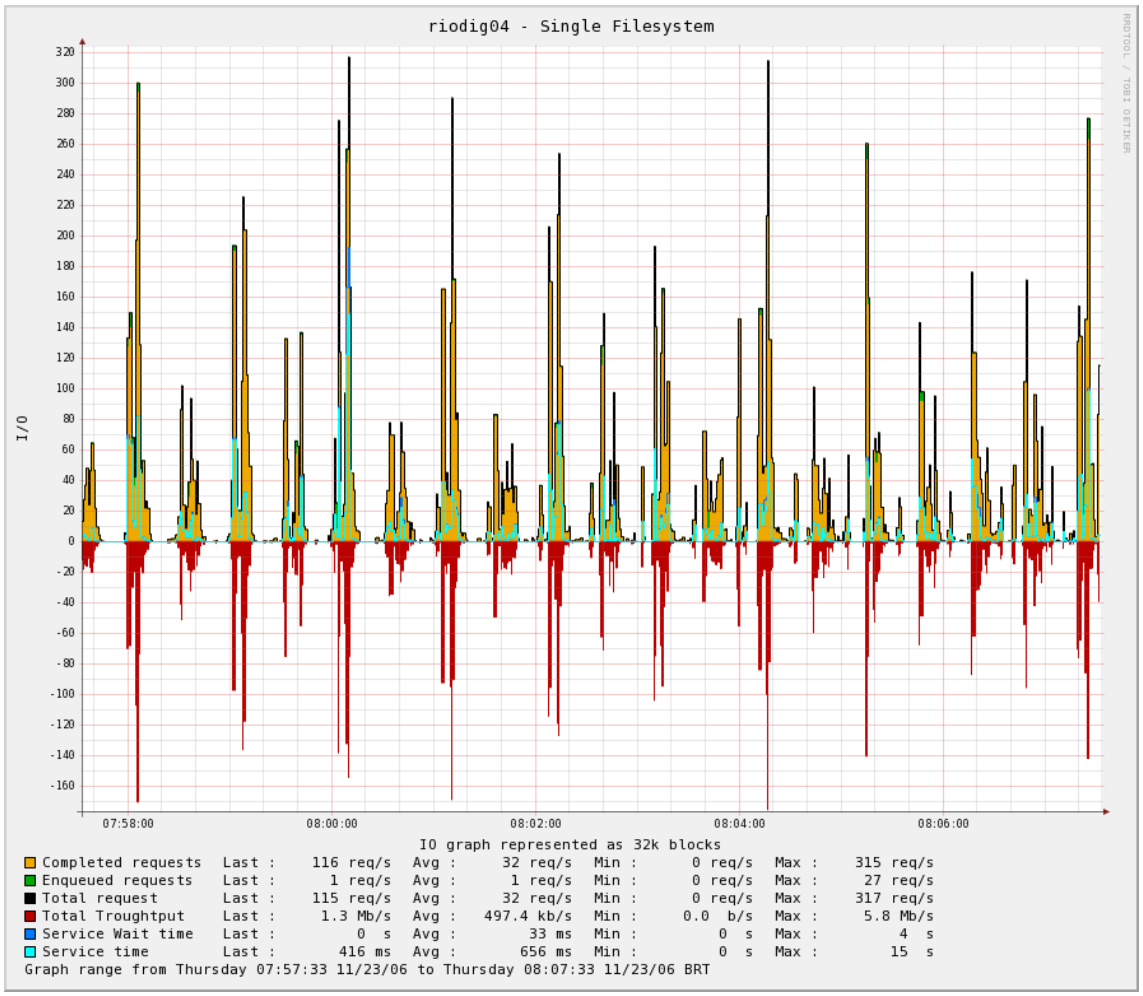


Figura 7.5 Utilização de I/O (disco rígido)

A Figura 7.5 mostra um dado muito importante que não aparecia na saída do programa *iostat*, que é o *service time*, tempo que uma requisição demora a ser respondida, e o *service wait time*, tempo que a requisição ficou aguardando na fila para ser processada.

Esses números só estão disponíveis via função table() e não existe uma ferramenta que venha com o sistema operacional que os apresente.

7.4. Módulo gerenciador de módulos

Com o primeiro módulo pronto e funcionando, agora é preciso criar uma forma de gerenciá-lo e adequar qualquer mensagem de log que seja exibida nos padrões que o OPSC espera.

O OPSC também é composto por vários subsistemas, e cada um é instanciado pelo gerenciador *OUPMServer* (Orga Unified Process Management Server). A primeira tarefa desse gerenciador de módulos do Orgaperf é se encaixar perfeitamente na inicialização do OPSC feita pelo *OUPMServer*.

O *OUPMServer* quando instância um processo, ele espera que, enquanto este estiver sendo executado, este não se desacople do terminal (entre em background e libere o *prompt*, por exemplo). Quando ocorre o desacoplamento do terminal, o *OUPMServer* considera que o processo terminou e, de acordo com a configuração, reinicia o processo após um determinado tempo.

Outra funcionalidade do *OUPMServer* é capturar tudo que é escrito no terminal e direcionar para o log central, o *Protokoll*. Para que não ocorram problemas com os programas de terceiros no momento de interpretar o *Protokoll*, a saída que o Orgaperf deve imprimir no console deve satisfazer o modelo da Saída 7.7.

```
orgaperf    :2603 154811:1:MAIN:Initialized MAIN as pid 453236
orgaperf    :2603 154811:1:MAIN:Initialized CollectBasicRRDs as pid 452971
orgaperf    :2603 154811:1:MAIN:Initialized CollectPerProcRRDs as pid 451434
orgaperf    :2603 154811:1:MAIN:Initialized CollectFIFOs as pid 449733
```

Saída 7.7 Formação esperada do log pelo *OUPMServer*

A primeira coluna deve ser o nome do processo que está imprimindo o log, a segunda coluna, data e hora, a terceira coluna deve informar se a mensagem significa inicialização (**1**), informação (**I**), aviso (**W**) ou um erro (**E**), a quarta coluna diz qual o nome do módulo interno e a quinta coluna o aviso que deve ser dado. Para obedecer a essa regra, foi criada uma função de log, para que sempre que seja necessário informar alguma coisa no *Protokoll*, a função se encarrega de formatar a mensagem da maneira correta.

Uma vez satisfeitas as interfaces de comunicação com o *OUPMServer*, podemos definir como será realizado o processo de configuração dos módulos do OPSC. De forma a simplificar a implantação de um arquivo de configuração, este é um pequeno

script em Perl. Seu código é enxertado no código principal do Orgaperf em tempo de execução, um exemplo de um arquivo de configuração pode ser verificado na Saída 7.8.

```
$$hostname = 'riodig04';

@$PROCS = qw /CollectBasicRRDs CollectCDRPerType CollectFIFOs /;

%$filename = (
  # Machine statistics
  'cpuinfo'        => [ $DBDIR . "/cpuinfo.rrd" ],
  'realmeminfo'    => [ $DBDIR . "/realmeminfo.rrd" ],
  'swapmeminfo'    => [ $DBDIR . "/swapmeminfo.rrd" ],
  'loadavg'        => [ $DBDIR . "/loadavg.rrd" ]

  # IO, they must ALWAYS begin with 'ioinfo-'
  'ioinfo-dsk0' => [ $DBDIR . "/ioinfo-dsk0.rrd", 'Single Filesystem' ],

  # FIFOs, they must ALWAYS terminate with 'FIFO'
  'BillingRecord001FIFO'  => [ $DBDIR . "/bgrd001fifo.rrd", 'MOSC MTSC GRPS' ],

  # CDR Statistics
  'cdrinfo'     => [ $DBDIR . "/cdrinfo.rrd" ],
);
```

Saída 7.8 Exemplo de arquivo de configuração

No arquivo de configuração especificamos as seguintes diretivas:

- **hostname**: É o nome que deve ser posto no cabeçalho dos gráficos para identificar a qual servidor pertence.
- **PROCS**: Especifica qual módulo desejamos habilitar, são eles:
  - **CollectBasicRRDs**: Coleta os dados de CPU, IO, uso de memória e carga no escalonador de processos.
  - **CollectFIFOs**: Coleta o tamanho das FIFOs internas.
  - **CollectCDRPerType**: Coleta as estatísticas dos tickets de tarifação processados.
- **filename**: Vetor associativo apontando qual RRD armazenará qual dado de desempenho.

Uma vez o Orgaperf configurado, é preciso antes de instanciar seus módulos, verificar se o ambiente está apropriado para que eles funcionem sem problemas.

Para que o funcionamento de um módulo não encontre problemas é preciso ter certeza que:

1. O diretório onde os RRDs são armazenados está OK;
2. O diretório onde os gráficos serão gerados está OK;
3. Os RRDs estão devidamente criados.

Os passos um e dois são bem triviais, bastando verificar se os diretórios existem e se estão com a permissão apropriada, se não tiver sucesso, abortar a execução do programa e emitir o erro no console. O *OUPMServer* se encarregará de avisar ao engenheiro de serviços que existe um problema.

O passo três verifica o arquivo de configuração por quais RRDs precisam estar criados, cada tipo (*cpuinfo*, *meminfo*, *swapinfo*, *ioinfo*, *cdrinfo*,*FIFOs*) tem uma estrutura diferente, por isso na configuração é preciso especificar qual tipo do RRD e a qual arquivo **.rrd** ele está associado.

Por questões arquiteturais, existem duas exceções nos tipos de RRD, quando um tipo começa por *ioinfo-*, sabemos que se trata de monitorar um disco físico, em seguida virá o nome do dispositivo. No arquivo de configuração, ioinfo-dsk0, significa que desejamos monitorar o disco físico dsk0. O outro campo é uma descrição sobre o que é o sistema de arquivos, essa descrição será impressa no gráfico para facilitar a identificação.

A outra exceção é quando desejamos monitorar FIFOs, sempre que um nome termina em FIFO, como no caso da *BillingRecord001FIFO* é que desejamos monitorar a fila **BillingRecord001**. O outro campo é uma identificação, essa identificação será impressa no gráfico para facilitar a identificação.

Com as bases RRD criadas, acessíveis e operacionais se torna possível iniciar os módulos filhos. Para cada filho o módulo pai faz uma cópia de si próprio usando *fork*() e executa a sub-rotina correspondente ao módulo.

Assim, um módulo filho nada mais é que uma cópia do processo pai rodando uma sub-rotina específica. Como vimos na seção anterior, onde explicamos o

funcionamento do módulo de coleta de dados de desempenho básicos, essas sub-rotinas entram em um *loop* infinito, sempre alimentado os RRDs usando a função *update*.

A tarefa do módulo gerenciador é após instanciar os processos filhos é verificar se eles estão sempre ativos, caso por algum motivo um processo filho morra, o sistema operacional envia um sinal ao processo pai, onde ele automaticamente re-instancia o processo que morreu. Outra funcionalidade do módulo gerenciador é acionada por um engenheiro de serviços. Quando ele deseja obter os gráficos de desempenho das informações armazenadas pelo Orgaperf em seus RRDs, ele envia um sinal usando o comando *kill* do sistema operacional, como podemos ver na .

```
kill -HUP `cat orgaperf.pid`
```

Esse comando instrui o Orgaperf para gerar todos os gráficos de desempenho nos últimos 10 minutos, outros sinais correspondem a outras janelas de tempo, como listados abaixo:

- HUP: Últimos 10 minutos
- USR1: Última hora
- USR2: Último dia
- WINCH: Última semana

Apesar dos RRDs estarem preparados para mostrar janelas maiores que uma semana, essas são as janelas mais utilizadas pelos engenheiros de serviços no dia-a-dia. Janelas maiores, de até 18 meses, podem ser obtidas de outra forma, reutilizando o código Perl que gera os gráficos do Orgaperf.

Quando um sinal é recebido pelo Orgaperf para gerar os gráficos de desempenho, uma linha é impressa no *Protokoll* em caráter informativo, uma cópia do processo pai é realizada via um *fork*() e executa a sub-rotina de criação dos gráficos.

```
orgaperf  :2603 154812:I:sighup: Genarating last 10 minutes graphs as pid 12153
```

Essa sub-rotina chama a função *graph* do RRDtool e o RRDtool se encarrega de evitar problemas de concorrência com os arquivos RRD, pois existem outros processos gravando nesses arquivos realizando operações de *update*.

A implementação foi realizada de forma só o pai gerar os gráficos, pois provou deixar o código dos módulos filhos enxuto e simples de manter. A sub-rotina que gera os gráficos do Orgaperf é a maior em quantidade de linhas e a mais complicada de manter.

Ao término do processo, os gráficos ficam disponíveis em <*diretório de instalação do Orgaperf*>/img.

A última tarefa do módulo gerenciador é quando o OUPMServer requisitar o seu termino do Orgaperf, este informa aos filhos para terminarem, aguarda o termino de cada um, e em seguida termina. Essa seqüência de finalização deve ser respeitada para que as bases RRD não se corrompam por término não apropriado.

7.5. Módulo de coleta de dados de desempenho do OPSC

Esse módulo é responsável por pesquisar o *Protolkoll* e procurar informações relevantes ao desempenho do OPSC. O dado mais importante que podemos retirar do *Protokoll* é a quantidade de CDRs (tickets de tarifação) processados até o momento. Como um tarifador, se em algum ponto da cadeia de processamento houver um problema, a quantidade de tickets processados será afetada. Armazenando a quantidade de tickets processados no RRD, havendo um problema, ele será evidenciado quando for analisado o gráfico de tickets processados.

O objetivo desse módulo é ficar "lendo" o *Protokoll* à medida que ele vai crescendo, procurando por linhas apropriadas e as contabilizando.

```
do {
  while (my $line = $fd->getline) {
      if ($line =~ /^opsc_bil[a-zA-Z0-9_ ]+:[0-9 ]+:I:([A-Z0-9]+);/) {
        if (defined ($tkts{$1})) {
          $tkts{$1}++;
        } else {
          $tkts{'OTHERS'}++;
        }
      }
    }

    my $rrdupd = 'N';
    foreach $key (@cdrtypes) {
      $rrdupd .= ":$tkts{$key}";
    }

    RRDs::update ($filename{'cdrinfo'}->[0], $rrdupd) or &Log("E",
"CollectCDRPerType", RRDs::error(), __LINE__);

    my $pos = $fd->tell;

    if ($lastpos == $pos) {
      $file = &CollectCDRPerType_openfile ($fd, $dir, $file);

      $lastpos = $fd->tell;
    } else {
      $lastpos = $pos;
    }

    sleep 1;
  } while ( 1 );
```

Saída 7.9 Implementação do módulo de amostragem de CDRs

A Saída 7.9 mostra como foi realizada a implementação desse módulo, onde para cada linha, é realizada uma procura baseado em um *regular expression* por um padrão, caso esse padrão seja detectado é feito a contabilização dessa linha como sendo um dos tickets já conhecidos (*INCDRMOC*, *INCDRMCC*, *INSS7MOC*,...). As linhas são lidas do arquivo até o momento que se recebe um *EOF* (fim do arquivo) do Perl, aí é feita a atualização do RRD com as informações coletadas.

Após esperar um segundo, uma nova tentativa de leitura é feita, caso um novo EOF seja recebido, é feito uma procura por um novo arquivo de *Protokoll*, pois o atual possivelmente já foi fechado e provavelmente já existe um novo. Como o *Protokoll* é um arquivo texto que está sempre crescendo e muda de nome após uma determinada condição (tempo ou tamanho), é feito uma checagem para verificar se estamos ao final do arquivo e devemos procurar o próximo.

Com o que foi amostrado, o gráfico resultante é bem similar ao que foi definido no capítulo anterior, como podemos observar na Figura 7.6.

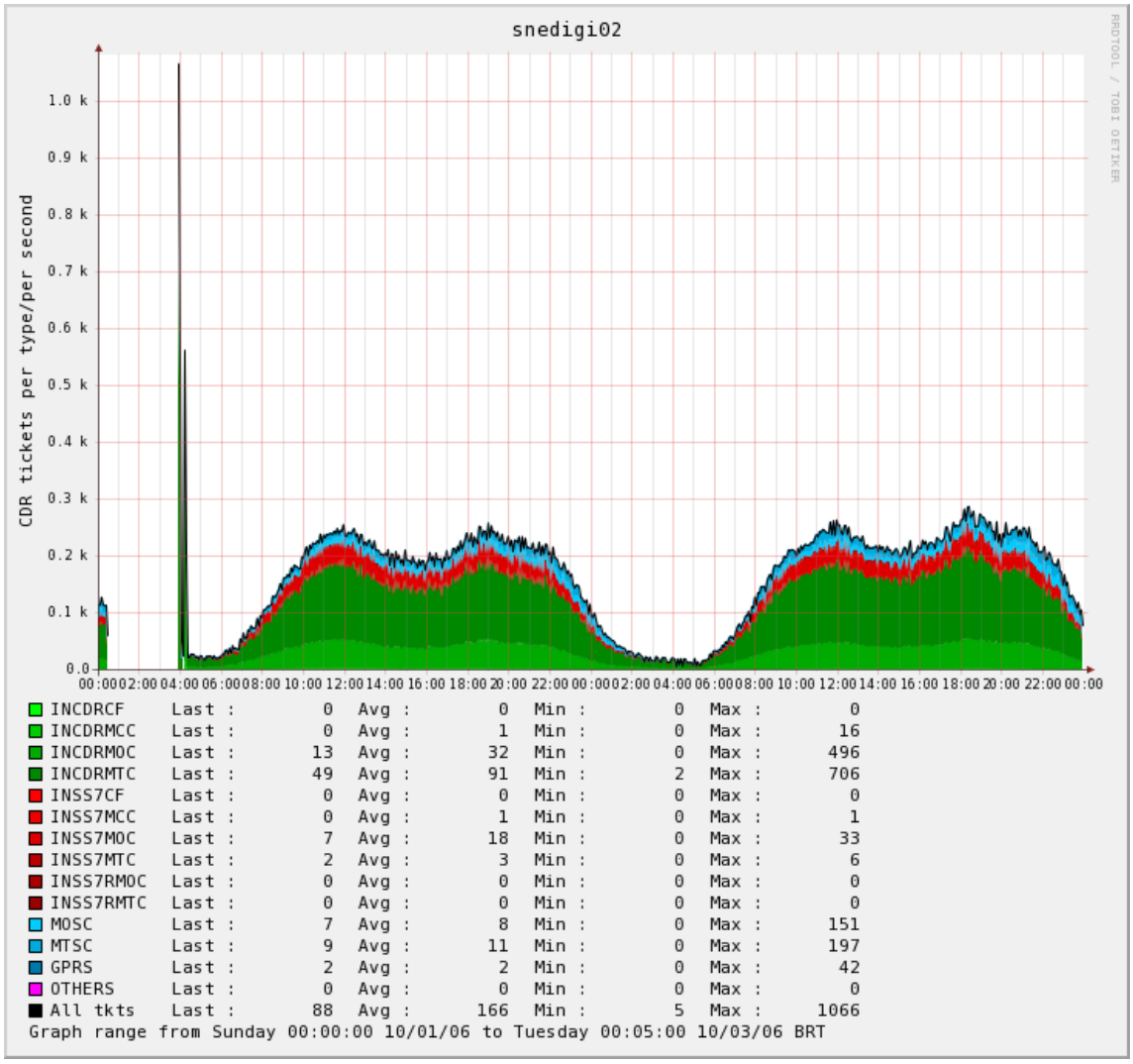


Figura 7.6 Gráfico de CDRs processados

7.6. Módulo de coleta de dados de desempenho de FIFOs

As FIFOs no OPSC são a alma de toda a comunicação inter-processos do sistema. O processo para medir o tamanho de uma FIFO padrão Orga depende de operações de abertura e leitura de arquivos, logo, para não influenciar a amostragem de FIFOs com outras amostragens, foi decidido criar um módulo responsável só para as elas.

Para garantir um menor custo de I/O, a operação de amostrar o tamanho de uma FIFO foi implementado usando XS, como mostra a Saída 7.10.

```
SV *
showfifoqueue (fifo)
  INPUT:
    char             *fifo;

  INIT:
    char              buf[BUFSZ];
    FILE             *p;
    int               out, in;

  CODE:
    snprintf (buf, BUFSZ, "%s/%s/IN", orgadata, fifo);
    p = fopen (buf, "r");
    if (p == NULL)
      XSRETURN_UNDEF;

    fscanf(p,"%d", &out);
    fclose (p);

    snprintf (buf, BUFSZ, "%s/%s/OUT", orgadata, fifo);
    p = fopen (buf, "r");
    if (p == NULL)
      XSRETURN_UNDEF;

    fscanf(p,"%d", &in);
    fclose (p);

    RETVAL = newSViv (out - in);

  OUTPUT:
    RETVAL
```

Saída 7.10 Amostrando o tamanho de uma FIFO usando XS

Usando na Saída 7.11 mostramos o código XS no Perl para atualizar os RRDs apropriados.

```
sub CollectFIFOs () {
  my @fifos = grep (/FIFO$/, keys %filename);

  do {
    foreach my $key (@fifos) {
      RRDs::update ($filename{$key}->[0], "N:" . ORGAPerf::showfifoqueue($key))
              or &Log("E", "CollectFIFOs", RRDs::error(), __LINE__);
    }

    sleep 1;
  } while ( 1 );
}
```

Saída 7.11 Processo de atualização do RRD das FIFOs

Cada FIFO configurada no arquivo de configuração é amostrada e o resultado armazenado em seu próprio RRD. Com o que foi amostrado, o gráfico resultante é bem similar ao que foi definido no capítulo anterior, como podemos observar na Figura 7.7

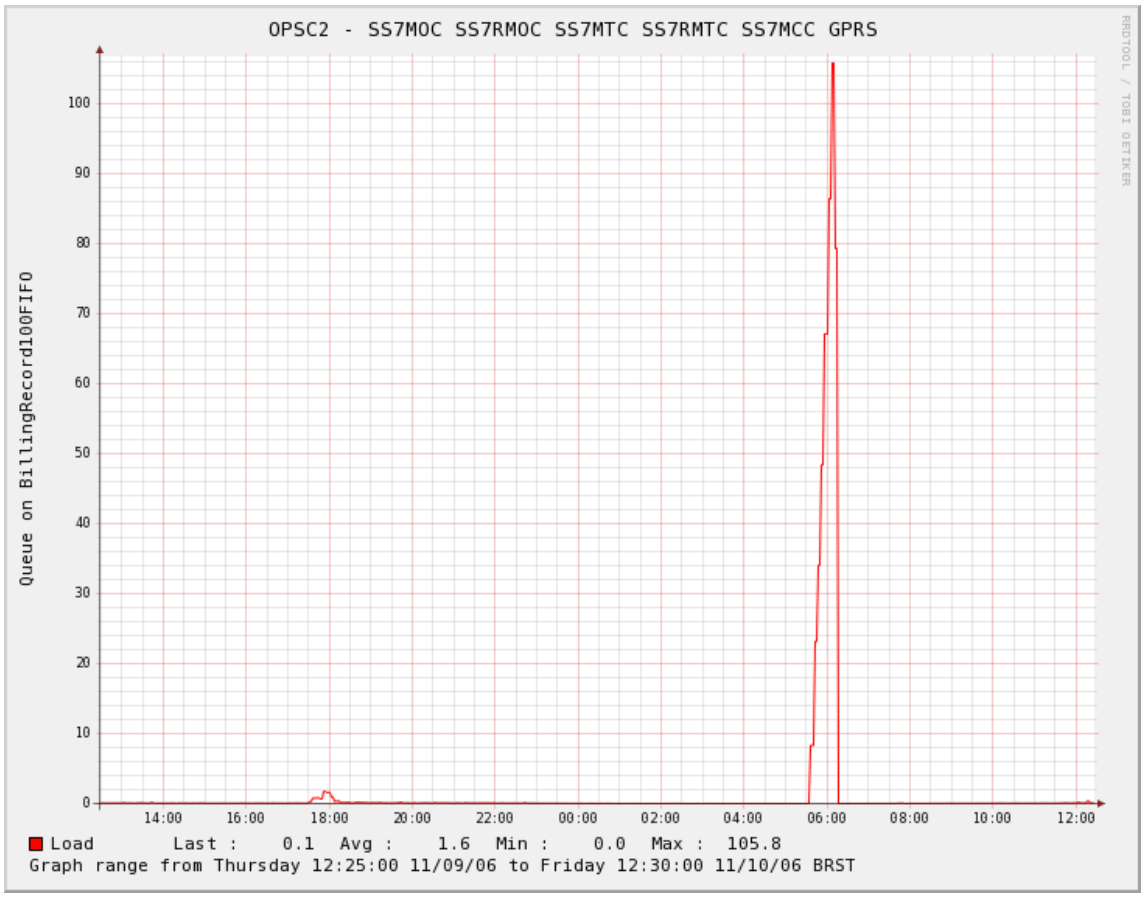


Figura 7.7 Gráfico de utilização da BillingRecordFIFO001

8. Resultados

Logo durante as primeiras semanas de funcionamento na TIM Brasil, o Orgaperf já mostrou cumprir o seu objetivo. Armazenar dados de desempenho ao longo do tempo seria de grande valor para os engenheiros de serviços.

Já podíamos ver, de forma clara, o que já era um consenso comum entre os engenheiros de serviços, que a carga entre os quatro OPSC não era aproximadamente igual.

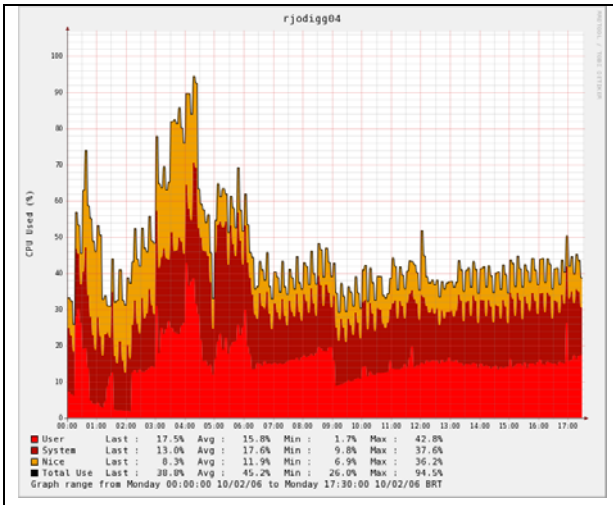
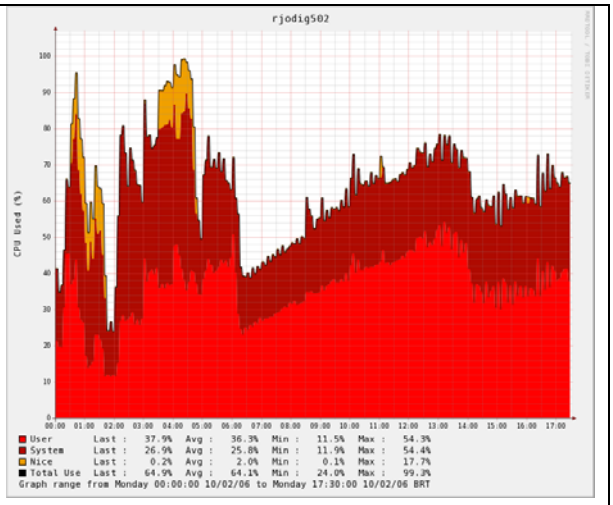
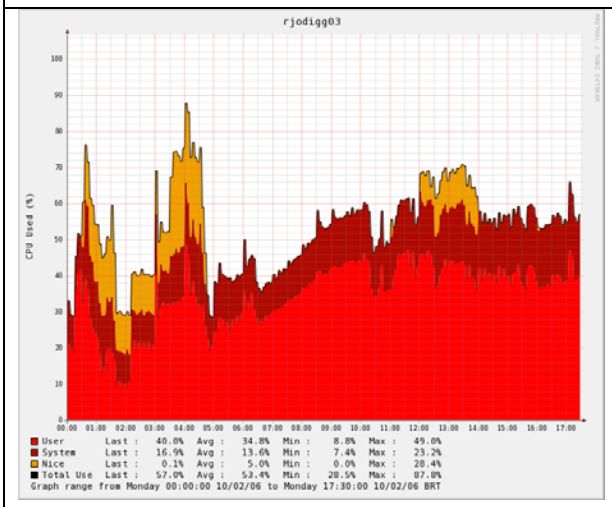
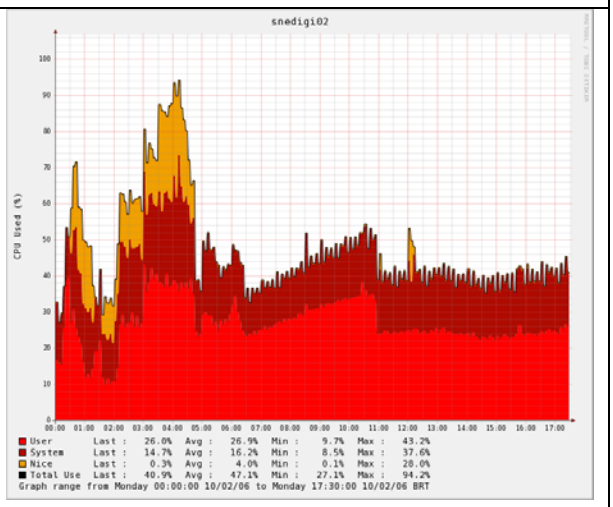
	
OPSC1	**OPSC2**
	
OPSC3	**OPSC4**

Tabela 8.1 Utilização de CPU dos quatro OPSCs

Pelos gráficos exibidos na Tabela 8.1 , chegamos à conclusão que o OPSC2, com 64% de consumo de CPU em média, e em seguida o OPSC3, com 53% de consumo de CPU em média, eram os mais carregados.

Outro benefício importante trazido pelo Orgaperf foi saber por que o OPSC3, mesmo tendo menos carga que o OPSC2, era sensivelmente bem mais lento que o OPSC3. Pelo gráfico de utilização de I/O podemos reparar que o tempo de serviço das requisições (*Service Time*) é bem alto.

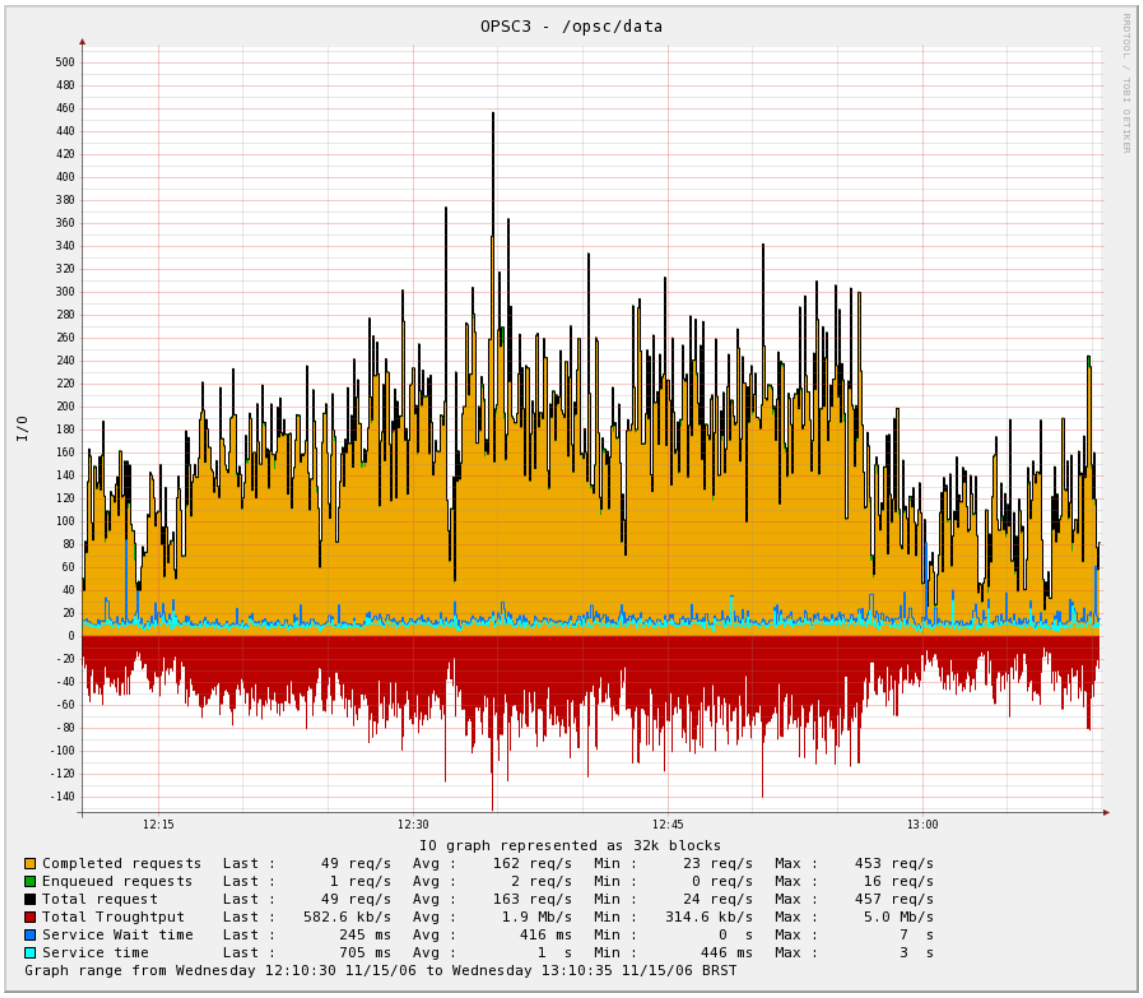


Figura 8.1 Utilização de I/O no OPSC3

Pela média, uma requisição de I/O levava um segundo para ser respondida, podendo levar até três segundos.

Após a apresentação desse fato, esta partição foi mais bem distribuída e o tempo de serviço baixou bastante, deixando o sistema funcionando bem melhor.

Algumas semanas mais tarde, um impacto sério de desempenho foi notado no OPSC2, sem motivos aparentes.

Recorrendo aos gráficos do Orgaperf, reparamos algumas coisas:

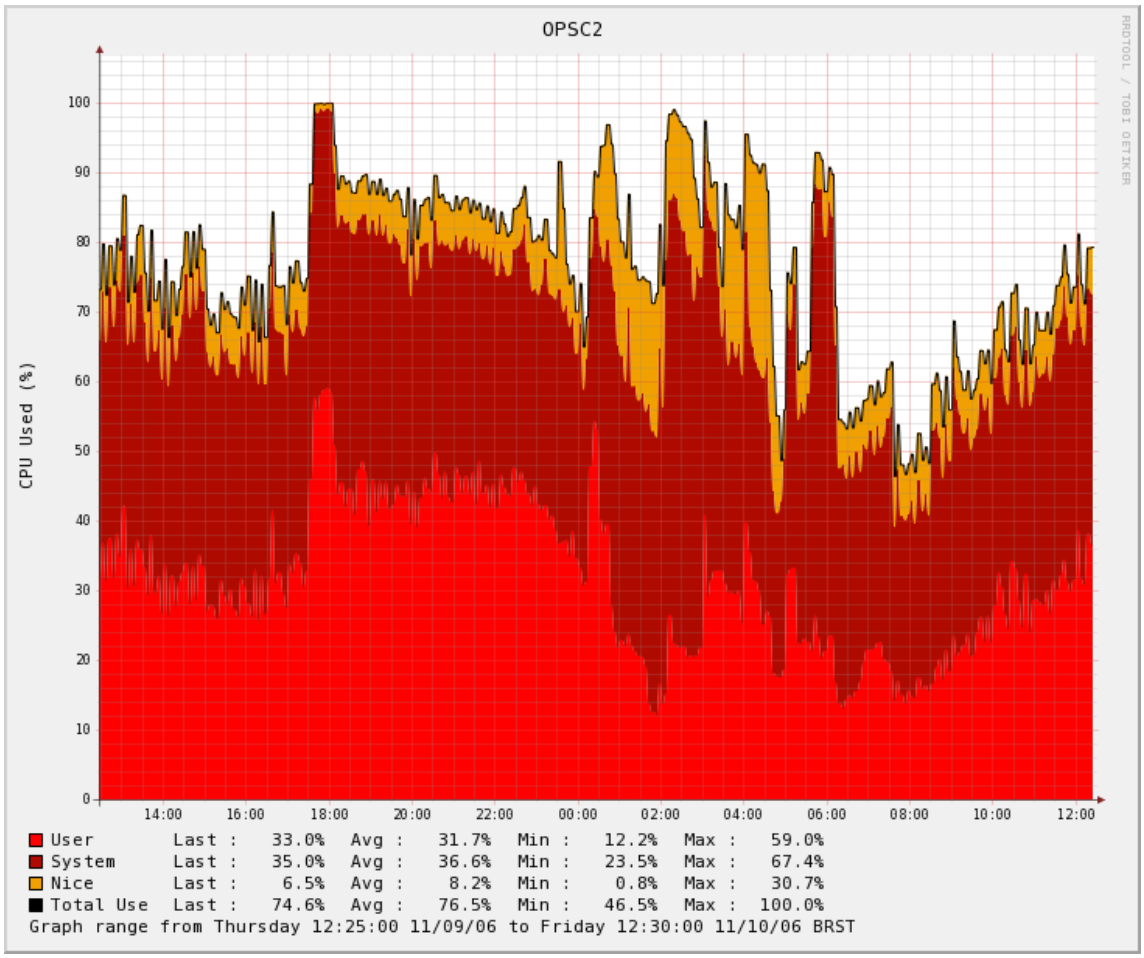


Figura 8.2 Utilização de CPU no OPSC 2

A utilização de CPU atingiu 100% durante o período de 17 horas e 30 minutos até pouco depois das 18 horas.

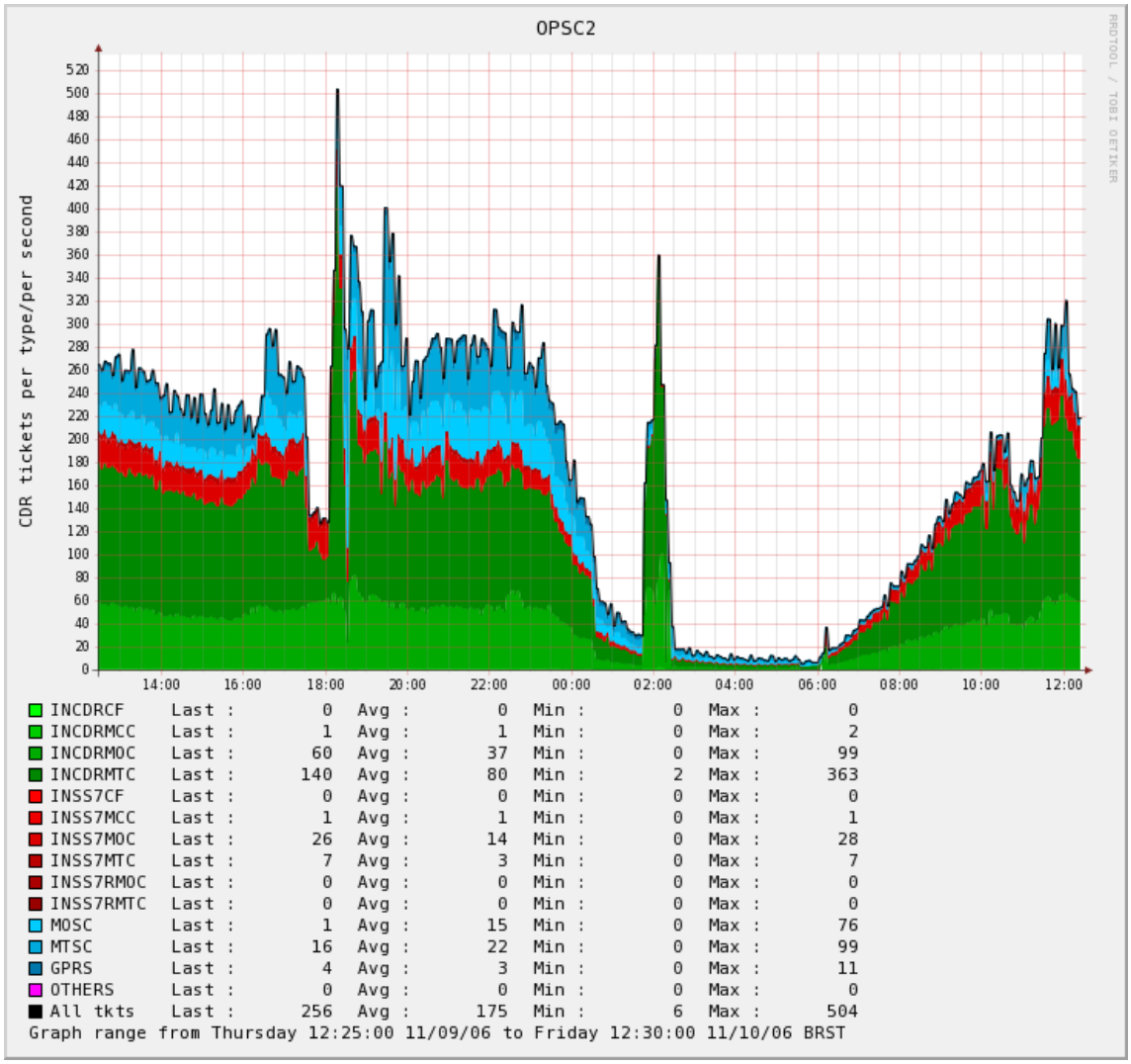


Figura 8.3 CDRs processados pelo OPSC2

No mesmo intervalo, como mostra a Figura 8.3, a quantidade de tickets processados caiu drasticamente, mostrando uma clara perda de receita no intervalo.

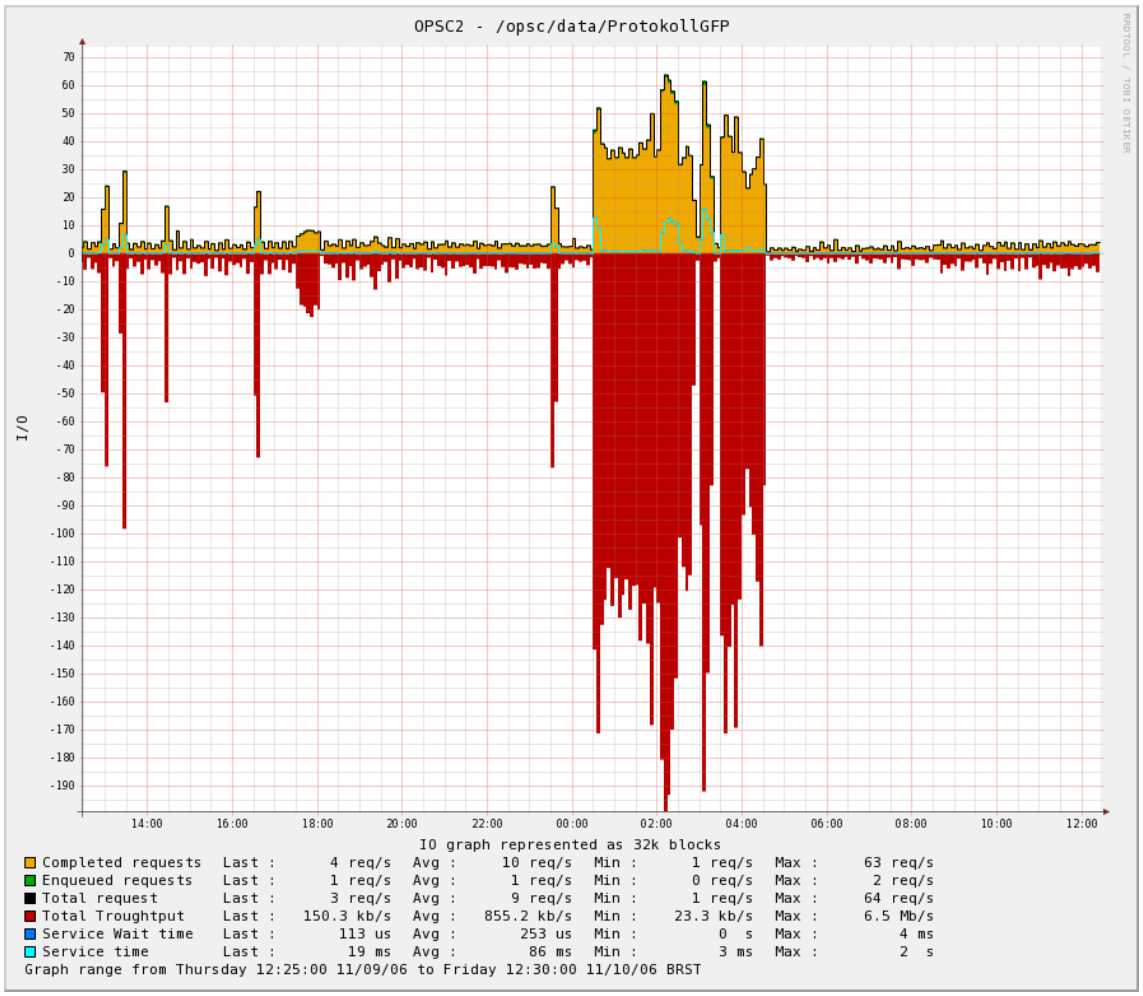


Figura 8.4 Utilização de I/O no OPSC2

E, no mesmo intervalo foi notado um comportamento estranho em uma das partições monitoradas, a partição onde eram armazenados os relatórios do GFP. GFP é um sistema que garante que as promoções e bonificações sejam aplicadas aos clientes apropriadamente.

Ao confrontar esses resultados com a equipe que prestava suporte ao GFP foi descoberto que na mesma hora que ocorreu a degradação de desempenho foi executado um script de caráter emergencial para realizar uma pesquisa sobre um cliente em especial. O erro foi executar esse script em um servidor de produção e ainda por cima em um horário não apropriado. A área foi alertada a não repetir o procedimento.

Nesse caso, os engenheiros de serviços puderam determinar com exatidão o início e o fim do problema, saber o efeito do problema e a sua causa, alertar ao cliente sobre como evitar sua reincidência e tirar a culpa da Orga sobre a degradação de

desempenho. Em longo prazo, o Orgaperf permitiu determinar melhor a projeção de recursos necessários e alocar melhor a distribuição de números entre os OPSCs.

Os ganhos não foram percebidos apenas em ambiente de produção. Com o Orgaperf os times de desenvolvimento e qualidade também souberam tirar proveito da nova ferramenta. O time de desenvolvimento agora tinha mais uma ferramenta para detectar vazamentos de memória e quantizar quanto recurso seria necessário para aplicar mais uma regra de negócio ao código do OPSC. O time de qualidade pôde verificar se mudanças feitas no OPSC pelo time de desenvolvimento iriam impactar no desempenho do OPSC em produção.

Com todos esses bons resultados em um curto período de funcionamento, o Orgaperf recebeu atenção especial da Orga Systems e prontamente foi proposta a instalação em outros clientes. A necessidade para portar o Orgaperf para outros ambientes logo surgiu, e assim o porte para o Digital Unix 4.0 e Linux foram feitos.

Rapidamente os clientes da América Latina de responsabilidade da Orga Systems Brasil passaram a ser monitorados pelo Orgaperf e hoje em dia, dois anos após a primeira instalação, o Orgaperf está sendo instalados nos clientes da Orga Systems Alemanha, Itália e Portugal em seus clientes pela Europa.

9. Conclusão

O método de análise estruturada foi ideal para o Orgaperf, onde a simplicidade era um ponto muito importante.

A opção pelo uso da linguagem Perl possibilitou, através de sua interface de extensão XS, buscar a forma mais eficiente de conseguir atingir o objetivo de não consumir muitos recursos, interfaceando diretamente com o *kernel* quando apropriado e simplificando o corpo do software com sua linguagem script, tornando-o fácil de alterar e simples de entender.

O RRDtool se mostrou uma biblioteca muito confiável e, após certo ponto da curva de aprendizado, simples de usar, extremamente flexível, e seu desempenho e em nenhum momento deixou a desejar. O Orgaperf é amplamente baseado nessa biblioteca e todo o sucesso da ferramenta pode ser atribuída ao uso do RRDtool.

As ferramentas de desenvolvimento do Tru64, entre elas o compilador C, o interpretador Perl e os programas auxiliares para compilar código XS para Perl, não apresentaram problema algum na plataforma *Alpha*, foi um processo de desenvolvimento bem tranqüilo, sem maiores surpresas.

Mesmo sendo um completo sucesso, o Orgaperf peca por não ter uma interface gráfica capaz de comparar resultados de vários servidores simultaneamente, forçando os engenheiros de serviços a entrar em nas máquinas de interesse para obter os gráficos de desempenho, uma a uma e olhar lado a lado em seus desktops os resultados. Uma proposta para o futuro do Orgaperf, além de suportar mais plataformas (ex. Sun Solaris), é criar outra ferramenta para auxiliar no trabalho de obter os gráficos de desempenho e auxiliar na comparação entre vários desses gráficos.

Esse projeto permitiu usar o conhecimento obtido nas disciplinas de engenharia de software, interfaceamento e integração de sistemas e, principalmente, os conhecimentos obtidos nos laboratórios de física experimental. O uso de gráficos para exibir séries temporais é abordado amplamente em física experimental II.

Os conhecimentos adquiridos durante a construção do Orgaperf foram muito importantes para completar a formação acadêmica do autor, por ser um exemplo prático de como transformar em solução um problema cotidiano e o tornar tão bem aceito a ponto de ser incorporado como um produto da Orga Systems.

10. Bibliografia

[1] Telefonia pré paga, wikipedia, http://en.wikipedia.org/wiki/Pay_as_you_go_%28phone%29, acessado em 2 de abril de 2008.

[2] Telefonia pré-paga no Brasil, Portal Teleco, http://www.teleco.com.br/comentario/com11.asp, acessado em 2 de abril de 2008.

[3] Pressman, Roger S., "Software Engineering", McGraw-Hill, 2002

[4] COCOMO - http://sunset.usc.edu/research/COCOMOII/ 2002

[5] Jr., Décio, "Perl: Guia de Referencia Rápida", Novatec Editora, 2000

[6] Siever, Elen, "Perl: Guia Completo", O'Reilly, 1999

[7] Wall, Larry, "Programming Perl", O'Reilly, 1999

[8] Schildt, Herbert, "C Completo e Total", Osborne, 1997

[9] PerlXS, Perl Documentation Project, http://perldoc.perl.org/perlxs.html, acessado em 29 de abril de 2008.

[10] Oetiker, Tobias, RRDtool, http://oss.oetiker.ch/rrdtool/doc/index.en.html, acessado em 29 de abril de 2008.