

# MONITOR DE SAÚDE DO CLUSTER DA UFES

MARCELO MAGALHÃES DO CARMO

March 12, 2003

Projeto final de conclusão do curso  
de Engenharia de computação

**Aluno : Marcelo Magalhães do Carmo<sup>1</sup>,**  
**Orientador : Raul Henrique Lopes<sup>2</sup>,**

Departamento de Informática, Universidade Federal do Espírito Santo,  
29060-900, Vitória, Espírito Santo, Brazil

---

<sup>1</sup>e-mail: marcken@terra.com.br

<sup>2</sup>email: raulh@inf.ufes.br

# Contents

<b>Contents</b>	<b>1</b>
<b>1 Introdução</b>	<b>3</b>
1.1 Proposta de Trabalho . . . . .	3
<b>2 Clusters - <i>Conceitos básicos</i></b>	<b>4</b>
2.1 O que significa Cluster? . . . . .	4
2.2 O que são Nós do Cluster? . . . . .	4
2.3 O que são os Masters de um Cluster? . . . . .	4
2.4 Importancia dos Clusters . . . . .	4
2.4.1 Performance . . . . .	5
2.4.2 Disponibilidade . . . . .	5
2.4.3 Capacidade de Crescimento . . . . .	5
2.4.4 Compromisso entre Custo/Beneficio . . . . .	5
2.4.5 Compromisso entre Segurança/Eficiência . . . . .	6
<b>3 O monitor de saúde do Cluster</b>	<b>6</b>
3.1 Uma Visão geral do Sistema . . . . .	6
3.1.1 O Processo Master . . . . .	6
3.1.2 Os Processos nós . . . . .	8
3.1.3 Os Processos monitores . . . . .	8
3.1.4 O Protocolo HMON . . . . .	8
3.1.5 O Protocolo Heartbeat . . . . .	9
3.2 Detalhamento do processo Master . . . . .	9
3.3 Detalhamento dos Processos Nós . . . . .	10
3.4 Detalhamento dos processos Monitores . . . . .	11
3.5 Detalhamento do protocolo HMON . . . . .	12
3.5.1 Primitivas do protocolo HMON . . . . .	12
<b>4 Tecnologias utilizadas no projeto</b>	<b>12</b>
4.1 TCP/IP . . . . .	12
4.2 Sockets . . . . .	13
4.3 Protocolo Heartbeat . . . . .	14
4.4 IPC (Interprocess Communication) . . . . .	15
4.4.1 Passagem de mensagens . . . . .	16
4.4.2 Mecanismo de sincronismo . . . . .	16
4.4.3 Memória compartilhada . . . . .	16

## List of Figures

1	Visão geral do sistema . . . . .	7
2	Visão geral do sistema . . . . .	7
3	Visão geral do sistema . . . . .	10
4	Visão geral do sistema . . . . .	11
5	Visão geral do sistema . . . . .	13
6	Visão geral do sistema . . . . .	15

## List of Tables

# 1 Introdução

Cluster é a denominação utilizada para definir um conjunto de equipamentos trabalhando em conjunto para resolver um problema. Os Clusters são muito utilizados hoje em dia para conseguir uma grande performance ou disponibilidade de um sistema a um menor custo do que ter máquinas de grande porte. Mas, aumentando o número de equipamentos trabalhando em conjunto também se elevam os níveis de complexidade do sistema como um todo. Isso naturalmente requer uma solução para facilitar a administração de sistemas mais complexos. A motivação deste trabalho está em aplicar o conhecimento adquirido no decorrer do curso de graduação em um projeto concreto, com aplicação prática em algum setor. Visto que nos últimos anos temos observado que existe uma grande demanda de desenvolvimento na área de processamento paralelo e processamento em Cluster, resolvemos aplicar o conhecimento do curso de Engenharia de Computação em um projeto nesta área. Outra coisa que me motivou a realizar um projeto nessa área foi o interesse de entender um pouco mais de como funciona a tecnologia na qual eu tenho trabalhado, pois atualmente tenho trabalhado no monitoramento e suporte técnico de um Cluster, aonde roda um sistema de banco de dados de missão crítica do negócio da empresa a qual eu presto meus serviços. Dentre todas as áreas de interesse, a parte de monitoramento de um sistema em Cluster foi o que mais interessou para desenvolver o trabalho. No decorrer deste texto, estaremos explicando conceitos básicos sobre a tecnologia utilizada para desenvolvimento do projeto, e porque foram tomadas algumas decisões de projeto. Neste texto também foram incluídas referências úteis sobre o tema desenvolvido, e alguns pequenos tutoriais de utilização em um ambiente real do projeto desenvolvido. Também estaremos comentando a utilização de ferramentas de terceiros utilizadas no decorrer do projeto.

## 1.1 Proposta de Trabalho

A linha de raciocínio que deveremos seguir neste trabalho é a de conseguir um sistema que possa auxiliar um Cluster a atender as exigências de performance e custo, e ainda fornecer um certo nível de segurança. Com isso pretendemos criar um sistema de monitoramento de saúde do Cluster. O monitor de saúde do Cluster nada mais é do que um sistema de monitoramento gerenciável, que pode capturar qualquer informação do Cluster e transformá-la em dados úteis que podem ser interpretados facilmente e também pode ser integrado com um sistema de alerta de monitores. Além disto, a proposta inclui fazer um sistema em camadas, que podem ser separadamente reescritas ( caso necessário ) sem que todo o sistema seja afetado pela modificação. Um sistema de monitoramento de Cluster pode facilmente ser justificado, pois ele tem por objetivo :

- Diminuir o tempo de indisponibilidade do Cluster;
- Aumentar a eficiência;
- Diminuir o desperdício de recursos;
- Atuar pró-ativamente na manutenção do Cluster;
- Gerar relatórios de utilização, viabilizando um dimensionamento preciso do Cluster;
- Ter acesso a uma visão da performance instantânea e média do Cluster;
- Centralizar a administração.

Para tal, foi feita uma proposta de um sistema em várias camadas, saindo da interface com o cliente até chegar na coleta dos dados dos monitores nos nós do Cluster. A princípio, a proposta de monitores que deveriam ser implementados seria de monitores de temperaturas dos processadores e das placas mães dos nós do Cluster, mas o sistema teria que ser também extensível para novos monitores serem implementados sem a modificação de nenhum componente além do novo monitor. A proposta também inclui um monitor de disponibilidade dos nós, que deve ser implementado afetando o mínimo possível à camada de comunicação. Além disso, o protocolo se propõe a ser simples e de fácil entendimento e implementação, já que ele deve ser divulgado para que outras pessoas possam também programar utilizando o protocolo e o sistema desenvolvidos neste trabalho.

## 2 Clusters - *Conceitos básicos*

Neste capítulo vamos abordar alguns conceitos básicos que nos levaram a algumas definições de qual linha de estudo seguiremos durante todo o trabalho. Vale a pena ressaltar que aqui serão colocadas definições em uma linguagem menos técnica, para um melhor entendimento e para futura apresentação de termos que serão utilizados no decorrer do texto.

### 2.1 O que significa Cluster?

Cluster é o nome comum dado atualmente a um grupo de máquinas que interagem para realizar uma única tarefa. Quando temos vários computadores juntos compartilhando recursos para realizar o mesmo trabalho, então chamamos esse grupo de computadores de Cluster (que do inglês significa "Grupo"). Um Cluster pode ser constituído de dois até N computadores, dependendo somente dos recursos de interação entre eles. Sendo assim, poderíamos pensar em um Cluster como várias máquinas trabalhando como uma única máquina para realizar uma única tarefa, mas nem sempre é assim. Hoje o conceito de Cluster é utilizado em vários pontos de vista diferentes, e não somente no ponto de criar um super computador com um grupo de várias máquinas juntas.

### 2.2 O que são Nós do Cluster?

Os nós de um Cluster são as máquinas que fazem parte do conjunto de máquinas, e que disponibilizam os seus recursos de processamento para realizar as tarefas submetidas ao Cluster. Normalmente os recursos dos nós são gerenciados por uma outra entidade, chamada Master.

### 2.3 O que são os Masters de um Cluster?

Os Masters são as máquinas que fazem parte do Cluster, compartilhando os seus recursos de processamento para realizar as tarefas do Cluster, mas também atuam em tarefas para administrar os recursos do Cluster. Normalmente, existem vários nós para serem gerenciados por um Master, mas podem existir configurações Multi-Master, aonde várias (ou todas) máquinas são gerentes do Cluster.

### 2.4 Importância dos Clusters

Atualmente as empresas de software e hardware vem dando uma grande importância a este conceito, pois na prática cada dia fica mais difícil e mais caro construir computadores com maior capacidade de processamento e armazenamento. Os Clusters tem por finalidade otimizar os recursos de processamento de várias máquinas para obter um desempenho próximo ao de um computador com maior capacidade. No entanto, é um erro achar que vários computadores juntos vão desempenhar uma tarefa igual a um único computador com a capacidade destes mesmos vários computadores, pois além de realizar a tarefa, o Cluster ainda deve se preocupar com o sincronismo dos dados entre os servidores. Os Clusters também podem ser vistos do ponto de ter sempre uma reserva de processamento em um caso de falha de um sistema. Quando temos vários computadores independentes trabalhando na mesma tarefa, podemos trabalhar a ideia de que se um destes computadores parasse de funcionar, os demais poderiam detectar esta anomalia e tratá-la, de tal forma que a tarefa que estava sendo executada antes desta anomalia continue a ser executada após a detecção do erro. Hoje em dia este recurso está sendo muito explorado pelas empresas de software e hardware, para tornar seus sistemas o mais confiáveis possível. Do ponto de vista de crescimento de processamento, em alguns casos existe a necessidade de em um determinado momento aumentar a escala de processamento. O Cluster também se propõe a solucionar o problema de escala, pois poderíamos agregar mais computadores ao grupo existente para aumentar o processamento, lembrando que o aumento de performance não é linear, e irá depender diretamente da tecnologia e dos algoritmos utilizados na implementação do Cluster. Esta técnica também é válida para um super computador, mas o custo de se construir um computador expansível é relativamente alto, e limitado, pois na teoria o crescimento de processamento sempre estaria limitado ao projeto do computador, enquanto que com um Cluster a escala estaria limitada a como os computadores interagem entre si e quantos

computadores existem no grupo. Então podemos ver um Cluster basicamente de três ângulos diferentes: Performance, Disponibilidade e Capacidade de crescimento. Nem sempre esses itens podem ser atingidos mutuamente, e normalmente é opção de projeto definir o que é mais importante dentre estes três ângulos básicos de visão.

#### **2.4.1 Performance**

Cluster que se propõem a serem otimizados para realização de cálculos intensos, aonde é necessário um grande poder de processamento são normalmente otimizados para performance. De um modo geral são sistemas que executam uma tarefa por um tempo determinado e seu objetivo é tornar este tempo finito o menor possível. Além disso, existe uma relação de importância dos dados envolvidos no processo de realização da tarefa e a necessidade de executar grandes esforços computacionais. Normalmente, quanto menos importante é o dado, maior é a otimização para performance, e quanto mais importante é o dado, menor é a otimização para performance e maior é a otimização para disponibilidade do serviço. Um bom exemplo de Cluster otimizado para performance é um Cluster que faz previsão do tempo. Os dados envolvidos normalmente são dados coletados de um satélite em tempo real. Isso faz com que o Cluster tenha que fazer um esforço computacional grande para analisar uma dada amostragem da tendência climática. Mas, se este mesmo Cluster ficar parado por alguns instantes, quando ele voltar poderá coletar novos dados de entrada e retomar ao processamento intenso.

#### **2.4.2 Disponibilidade**

Clusters que se propõem a serem otimizados para manter uma tarefa executando o maior tempo possível. Normalmente são sistemas que devem estar funcionando por tempo indeterminado e cujo objetivo é manter este sistema funcionando por um maior tempo possível sem interrupções. Um exemplo clássico deste sistema seria um servidor Web. Normalmente servidores Web não necessitam de esforço computacional grande, mas necessitam estar disponível 24h por dia, sete dias por semana.

#### **2.4.3 Capacidade de Crescimento**

Clusters que se propõem a serem flexíveis quanto ao seu poder de processamento. Normalmente, são sistemas que tem uma tendência de crescimento contínua, forçando o projetista do sistema a escolher uma estrutura que tenha uma capacidade de crescer junto com o sistema. Um exemplo simples desta necessidade é um banco de dados. Normalmente um banco de dados é modelado para uma quantidade de processamento e armazenamento inicial, mas naturalmente à medida que mais dados são inseridos a necessidade de processamento e armazenamento aumenta.

#### **2.4.4 Compromisso entre Custo/Benefício**

Clusters normalmente tem de enfrentar este problema, pois toda solução adotada para conseguir com que várias máquinas interajam tem um custo. Poderíamos desenvolver um sistema de hardware complexo, que intercomunicasse as máquinas para formar um Cluster, altamente rápido, seguro e que fosse expansível. Só que isso seria extremamente trabalhoso e caro, pois os algoritmos envolvidos na comunicação entre as máquinas são complexos, e quando nos inserimos segurança no sistema inevitavelmente colocamos também variáveis não controladas pelo sistema, que podem mudar de estado em um momento aleatório (como por exemplo, um disco rígido parar de funcionar por uma falha mecânica). Normalmente, as soluções levam em consideração o nível do investimento, a análise de risco do sistema e as exigências do cliente que irá adquirir a solução. Esses parâmetros normalmente são difíceis de serem medidos, e cabe ao projetista estimar esses dados para ter uma idéia de quem serão os seus clientes. Em alguns casos podemos chegar à conclusão que desenvolver um Cluster para uma determinada tarefa irá se tornar mais caro do que comprar um super computador para realizar a mesma tarefa, dependendo das exigências do projeto. Vale a pena lembrar que em termos de performance, aumentar o número de máquinas nem sempre é aumentar a performance, pois o crescimento não é linear, mas com o aumento de máquinas

após um determinado patamar à tendência e que a performance fique constante ou até mesmo que piore, dependendo da implementação do Cluster.

#### **2.4.5 Compromisso entre Segurança/Eficiência**

A cada dia que passa maior é a importância da segurança de sistemas. Segurança que deve ser interpretada tanto no sentido de segurança de dados quando no sentido de segurança de ter o sistema funcionando corretamente. Esse fator é geralmente um limitador de performance de sistemas computacionais, pois normalmente eles simplesmente agregam cargas altas de computação e quando falamos de Clusters esse fator nem sempre é desejado e pode ser admissível. Normalmente a segurança faz com que o sistema seja redundante, tanto na questão de dados transmitidos entre as máquinas para funcionamento do Cluster quanto na questão de equipamentos necessários para o processamento. A teoria é achar um equilíbrio entre esses dois fatores, atendendo obviamente as exigências do projeto.

### **3 O monitor de saúde do Cluster**

Neste capítulo estaremos comentando o trabalho, explicando o funcionamento de cada elemento e mostrando as tecnologias estudadas e utilizadas no desenvolvimento do trabalho. Estaremos entrando em detalhes de implementação e apresentando o motivo de algumas escolhas de projetos. Contudo, as documentações de APIs e códigos escritos e utilizados somente serão vistos posteriormente nos próximos capítulos

#### **3.1 Uma Visão geral do Sistema**

O monitor de saúde do Cluster é o sistema que possibilita o Administrador do Cluster saber o que esta acontecendo no Cluster em um determinado momento, fazer relatórios históricos do ambiente e ainda alertar caso aconteça alguma coisa de anormal ou prejudicial ao Cluster. Neste trabalho, estaremos abrangendo a criação de uma infra-estrutura para monitorar praticamente tudo o que se desejar monitorar nos nós do Cluster. Estaremos abordando um ambiente de Cluster com um único Master, e estaremos considerando que existe uma infra-estrutura privada de comunicação do Master com os demais nós do Cluster. Um esquema simplificado da infra-estrutura estudada seria representado como na figura 1.

O monitor de saúde teria um serviço rodando no Master e outro rodando em cada nó do Cluster. O serviço Master controla todos os serviços dos nós. Por sua vez, o serviço de um nó controla vários monitores, que enviam os seus dados para o serviço nó. A Lógica do sistema esta em ter uma base de dados distribuída com as informações dos monitores em cada nó respectivo, sendo atualizada com o Master somente por requisição do explícita do protocolo ou por configuração periódica. Podemos simplificar o modelo do sistema como sendo mostrado na Figura 2.

##### **3.1.1 O Processo Master**

O processo Master é quem controla os protocolos de comunicação entre o Master e o Nós, mantém atualizado o status dos nós via um protocolo chamado Heartbeat, e também atende as requisições dos clientes do cluster para disponibilizar as informações dos monitores. O processo Master é o Gateway centralizador de todas as informações de todos os nós, de tal forma que qualquer cliente que queira saber alguma informação sobre qualquer monitor em qualquer nó fará a requisição para o Master. O processo Master também é responsável pela integridade de toda a base de dados dos monitores de todos os nós gerenciados. O processo Master na verdade é subdivido em um processo central gerenciador de recursos, um processo que cuida de atender conexão dos nós, um processo que cuida da conexão dos clientes e um processo filho para cada nó conectado ao pai. De tal maneira, em um Cluster de 64 nós, teríamos 64 processos filhos rodando, um para cada filho conectado ao Master. Além disso, cada processo filho roda uma instancia independente do Heartbeat, e comunica o processo pai caso alguma coisa aconteça com o filho. Todas as modificações nas bases de dados dos nós são transmitidas para os processos filhos e tratadas individualmente, sendo que cabe ao processo central organizar a base e enviar as informações corretas os clientes quando requisitado. O processo Master foi implementado na linguagem de programação

Figure 1: Visão geral do sistema

Figure 2: Visão geral do sistema

Perl, utilizando a biblioteca padrão e alguns módulos escritos em C e compilados para Linux. Estes módulos podem ser facilmente portados para qualquer sistema POSIX (como Linux), e também para outros sistemas UNIX (como Solaris, FreeBSD, Digital UNIX, etc). Os detalhes de implementação serão descritos posteriormente, mas por enquanto vamos nos deter as funcionalidades macro do sistema.

### **3.1.2 Os Processos nós**

Os processos nós são aqueles que centralizam as informações dos monitores, que normalmente são os monitores locais da maquina (mas não obrigatoriamente deve ser, pois podemos ter um processo nó recebendo informações de um monitor que está em outra maquina). O processo nó é responsável por atender e catalogar os monitores que a ele querem enviar informações, e deve atender aos comandos do processo Master, repassando se necessário às requisições de atualização para os monitores. Os processos nós gerenciam um pequeno volume de dados, mas devem garantir a integridade dos mesmos e também devem garantir que dois monitores não podem concorrer pela mesma área de armazenamento de dados. Os processos nós ainda implementam uma instancia do Heartbeat com o processo Master, sendo que cabe a ele somente responder as requisições do processo Master no seu tempo limite. Os processos nós também disparam processos filhos para atender a cada requisição de comunicação com os monitores, mas ao invés do processo Master, estes processos filhos não verificam se os monitores estão ou não respondendo, somente repassam as informações enviadas pelos monitores para que o processo central possa gerenciar a base de dados. Os processos Nós foram implementados na linguagem de programação Perl, utilizando a biblioteca padrão e alguns módulos escritos em C e compilados para Linux. Estes módulos podem ser facilmente portados para qualquer sistema POSIX (como Linux), e também para outros sistemas UNIX (como Solaris, FreeBSD, Digital UNIX, etc). Os detalhes de implementação serão descritos posteriormente, mas por enquanto vamos nos deter as funcionalidades macro do sistemas.

### **3.1.3 Os Processos monitores**

São os processos que realmente capturam as informações da maquina. Enquanto os outros processos são virtualmente independentes da maquina que eles estão rodando, este está ligado física ou logicamente a maquina a qual ele pertence. Os monitores dependem também (muitas vezes) de informações muito especificas de hardware, normalmente fornecidas pelos fabricantes dos mesmos. Eles não armazenam nenhum dado, somente periodicamente atualizam os seus dados com os processos Nós. Estes processos também devem atender requisições urgentes de atualização, definidas no protocolo HMON. Este projeto se propôs a escrever monitores de temperatura das maquinas do cluster, e devido ao modo como se captura a temperatura ser muito especifico (dependendo do chipset da maquina) o sistema acabou tendo de ser escrito em C/C++. Foram criadas bibliotecas de comunicação e bibliotecas de erro, para acessar os processos nós e para salvar irregularidades com os monitores nos arquivos de log do sistema. Os monitores devem rodar em sistemas Linux, pois os drivers que fazem acesso direto ao endereço de memória do chipset estudado somente foi escrito para Linux (mas o projeto é OpenSorce, então é possível que se consiga portar para outros sistemas abertos como FreeBSD). Os detalhes dos chipsets estudados e do programa serão descritos posteriormente.

### **3.1.4 O Protocolo HMON**

O protocolo HMON é o quem permite a comunicação entre Master, Nós e Monitores. Nele estão definidas todas as consultas e requisições, e todos os Processos devem implementar o protocolo (ou pelo menos os comandos que atingem diretamente o nível de processo). Este protocolo foi descrito utilizando a linguagem XML, para fácil entendimento dos leitores e rápida implementação. O protocolo HMON será detalhado posteriormente, com exemplos em XML para facilitar o entendimento e para explicar o fluxo de comunicação desde o cliente até os monitores.



### 3.1.5 O Protocolo Heartbeat

O protocolo Heartbeat é uma ferramenta adicional ao sistema de monitoramento de Cluster, que pode ser configurado para ser desativado. O Heartbeat é uma forma de verificar se os processos nós ainda continuam respondendo, ou se por algum motivo anormal (falha de hardware ou de software) eles deixaram de responder. Em resumo, ele envia um Heartbeat (batida do coração) e espera o seu eco do sistema remoto. Funciona como um ping, mas no nível de transporte. Ele é mais eficiente do que o ping por que já está encapsulado no protocolo de transporte, não necessitando de nenhum outro cabeçalho adicional para tal, e ainda é um pacote que na implementação é marcado para prioridade alta, sendo que o ping (pacotes ICMP) são tratados como pacotes com prioridade baixa, podendo ser descartados em caso de sobrecarga da camada de comunicação. Outra vantagem do Heartbeat é que ele pode ser implementado em qualquer tipo de camada de comunicação, pois sua implementação pode ser substituída por uma implementação em outra camada de comunicação sem modificações no protocolo. Sua implementação será descrita posteriormente.

## 3.2 Detalhamento do processo Master

O processo Master foi desenvolvido utilizando a linguagem Perl (Practical Extraction and Report Language). Esta decisão de projeto foi tomada devido à praticidade com que um sistema pode ser escrito. Se fossemos escrever o mesmo sistema em C ou C++, gastar-se-ia o dobro do tempo, pois coisas como gerenciamento de memória alocada e gerenciamento de estrutura complexas teriam de ser implementadas. Como o objetivo do projeto não era fazer um sistema performático, mas funcional, então foi decidido por uma linguagem que proporcionasse um rápido desenvolvimento. Mas, como Perl também é uma linguagem que pode se tornar confusa se o programador usar de artifícios oferecidos pela linguagem, foi conduzida uma programação seguindo alguns padrões, como declaração explícita de variáveis, declaração explícita de funções e formatação clara das funções, seguindo os padrões de C e C++. O código também foi dividido em módulos para facilitar o entendimento de cada parte do código e foi utilizado (quando possível e legível) o conceito de orientação a objetos. Vale citar que outras linguagens foram consideradas no projeto, como Java, por exemplo. Mas mesmo Java não pode dar a praticidade de Perl, que pode facilmente importar uma API C ou C++ já escrita para o sistema e também não poderia dar a performance mínima requerida para um serviço de Cluster. Futuramente, um ótimo exercício seria reescrever todo o sistema em C++, para melhorar a performance e tornar o código puramente orientado a objetos, com classes bem definidas, mas a atual implementação já tem o código bem portátil e legível, tornando simples a sua estruturação em uma linguagem orientada a objetos. O motivo pelo qual não utilizamos orientação a objetos no sistema escrito em Perl é que a biblioteca escrita para orientação a objetos é um pouco confusa nesta linguagem, e torna o código menos legível do que utilizando funções. Toda a comunicação entre nós e clientes foi implementada utilizando o protocolo de comunicação TCP/IP, rodando sobre uma camada física Ethernet. Para fazer comunicação entre os clientes e nós, foi utilizado a biblioteca de sockets do UNIX, que em Perl é somente uma chamada de sistema para as APIs de sockets do sistema nativo. O motivo pela qual foi utilizado o protocolo TCP/IP é simples, ele vem sendo amplamente utilizado para comunicação no mundo todo, é de um custo baixo para moderado e tem uma vasta biblioteca de programação pronta. Nada impediria de reescrever o sistema utilizando outra arquitetura de comunicação, somente teríamos de reescrever alguns módulos do sistema, em especial os módulos de comunicação e de protocolo Heartbeat (tanto em Perl quanto que em C++). Internamente, o sistema funciona da como no esquema mostrado na Figura 3.

O Processo central é quem inicia os demais processos. Ele fica encarregado de iniciar e finalizar todos os processos do sistema. Ele também é quem faz log de falhas no sistema e que carrega os arquivos de configurações iniciais do programa. O processo atende nós inicia uma instancia de comunicação, que no caso da implementação deste projeto é um socket que fica escutando em uma porta especificada pelo arquivo de configuração do sistema. Quando uma conexão chega na porta determinada ele inicia um novo processo para atender a requisição de conexão do novo nó. Ele também é responsável por limpar a memória utilizada pelos processos que ele cria. O processo atende clientes faz a mesma coisa que o atende nós, mas para os clientes do monitor de saúde do Cluster. Os processos de comunicação com os nós implementam fazem as chamadas as APIs do protocolo HMON para fazer a comunicação com os nós

Figure 3: Visão geral do sistema

do Cluster. Eles também iniciam uma instancia do protocolo Heartbeat na conexão com o nó e ficam constantemente monitorando a conexão remota. Caso ocorra algum erro de conexão, eles terminam enviando uma mensagem via protocolo IPC para o processo atende nós. Quando eles recebem novas informações dos monitores de um nó, eles realizam uma atualização na base de dados de monitores do sistema. Para fazer tal atualização, é utilizado um mecanismo de lock, para evitar acessos de escrita concorrentes na base de dados. Os processos de comunicação com os clientes são quem recebe e envia as informações requisitadas pelo cliente. Eles implementam as consultas na base de dados, e como os clientes não podem fazer atualizações na base, eles somente fazem um lock compartilhado da base para saber informações da mesma. O cliente também pode requisitar atualizações na base, o que irá fazer com que todos os processos encaminhem esta informações para os nós para reenviarem suas informações atualizadas para o processo Master. Todas as informações, incluindo a base de dados, são transmitidas de processo para processo ou acessadas via IPC ( Interprocess Communication ). Este protocolo permite que processos na mesma maquina possam trocar informações, compartilhar recursos e avisar uns aos outros dos seus estados e emitir sinais para troca de estados uns dos outros. Tornarei a comentar este protocolo estudados com mais detalhes posteriormente.

### 3.3 Detalhamento dos Processos Nós

O processo que roda nos nós dos Cluster foi implementado utilizando a linguagem Perl (Practical Extraction and Report Language), pelos mesmo motivos já citados no desenvolvimento do Processo Master. Toda comunicação entre nós e clientes, e entre nós e monitores foi implementada no protocolo TCP/IP, utilizando a API de Socket do UNIX. Os motivos já foram citados também no desenvolvimento do Processo Master. Os processos nós normalmente fazem uma conexão com o processo Master e aceitam varias conexões de vários monitores diferentes. Os monitores não necessariamente necessitam de estar localizados localmente na maquina aonde o processo nó esta rodando, podendo o processo nó gerenciar monitores remotos (com a restrição de não ter controles como Heartbeat para verificar a continuidade do serviço de monitor). Internamente os processos nós funcionam seguindo o seguinte esquema da Figura 4.

O processo central é o responsável por ler o arquivo de configuração, iniciar a conexão com o processo Master e iniciar o processo Atende Monitores. Além disso, o processo central é responsável por terminar os demais processos e gerar logs de erros para o sistema. O processo atende monitores é responsável por

Figure 4: Visão geral do sistema

iniciar um socket que fica escutando em uma porta registrada no arquivo de configuração, sendo que para cada processo monitor que abrir uma conexão será iniciado um processo para atender essa conexão. Os processos de comunicação com os Monitores são quem realizam a troca de informações do protocolo, e que atualizam a base de dados dos monitores. Quando um novo monitor abre uma conexão, o processo de comunicação pede as configurações do monitor, verifica se ele já existe na base de dados e envia a notificação para o monitor de qual será o seu identificador. Cada monitor então é identificado com um identificador único que é composto pelo identificador no nó mais o identificador no monitor. Toda a comunicação entre nó e master e nó e monitores é feita via o protocolo HMON, e toda comunicação entre os processos é feita via IPC, incluindo a comunicação interna dos processos com a base de dados compartilhada. A documentação sobre os protocolos e APIs utilizadas será descrita posteriormente neste documento.

### 3.4 Detalhamento dos processos Monitores

O único processo monitor desenvolvido foi o processo de monitoramento de temperatura de CPU. O processo utiliza uma biblioteca de sistema chamada `sensors.h` que faz chamada direta ao hardware da máquina. O chipset que nos testamos para monitorar a temperatura da CPU foi o VT82C686A. Este chipset que foi utilizado para construir um monitor funcional de teste tem um driver escrito para o Kernel do Linux versão 2.4.18. O driver foi escrito no projeto `lm-sensors`, que pode ser encontrado no site [www.lm-sensors.nu](http://www.lm-sensors.nu). Este driver já vem incluso nas versões da distribuição RedHat a partir da versão 7.3 (que já vem com o Kernel 2.4.18 ou superior). Todo o monitor foi escrito em C/C++, sendo que os drivers devem ser iniciados com acesso privilegiado (de `system`) e o processo do monitor deve rodar com acesso de super usuário (`root`). Foram desenvolvidas bibliotecas em C/C++ para facilitar com que novos monitores fossem escritos para plataforma de monitoramento de saúde do Cluster. A funcionalidade do monitor em si é bem simples, tendo ele somente que abrir uma conexão com um processo nó, receber a requisição de informações, enviar as suas configurações para o processo nó e esperar a nova requisição de informações. O próprio monitor que decide de quanto em quanto tempo ele será incomodado pelo processo nó. É ele que também decide se vai enviar dados continuamente ou somente enviar dados de uma média de coletas após um tempo de amostragem. No caso do monitor de temperatura, é enviado um valor instantâneo da temperatura periodicamente, aonde o período pode ser configurado no programa do

processo monitor.

### 3.5 Detalhamento do protocolo HMON

O protocolo HMON foi definido para ser simples, prático e de fácil aprendizagem. Ele tem poucas primitivas, e foi totalmente implementado utilizando a linguagem XML para definição do protocolo. O protocolo é constituído de primitivas de consulta e informações. As primitivas de consultas foram todas definidas na API HMON\_PROTOCOL.pm, desenvolvida para Perl, e as informações são o retorno das primitivas de consultas. Quando um cliente, ou o master ou o nó querem fazer uma consulta, eles chamam a primitiva de consulta e enviam para o destinatário. O destinatário avalia a consulta em sua base de dados local e retorna a árvore XML que é referente à primitiva de consulta desejada. Caso algum a primitiva inclua uma flag de atualização, a consulta é reescrita e passada adiante, até chegar ao seu ultimo destino que é o monitor. Uma consulta de atualização de todo o Cluster percorrerá todos os nós e todos os monitores para serem atualizados naquele momento, e uma busca por um monitor específico somente será enviada para o nó do monitor e do nó encaminhada para este monitor requisitado. O protocolo foi feito para atender a demanda de uma base de dados distribuída, sem a necessidade de constantemente atualizar todos os monitores.

#### 3.5.1 Primitivas do protocolo HMON

- Primitiva Node -
- Primitiva Master -
- Primitiva Config -
- Primitiva Monitor -

## 4 Tecnologias utilizadas no projeto

Neste Capítulo, estaremos fazendo uma análise das tecnologias estudadas e utilizadas na implementação do trabalho. Estaremos discutindo sobre os tópicos citados anteriormente como IPC e Sockets, e estaremos exemplificando e apresentando a teoria envolvida no trabalho.

### 4.1 TCP/IP

Toda a parte de comunicação entre os processos distribuídos neste trabalho foram utilizando a pilha TCP/IP como base. Este documento, no entanto não terá como foco explicar nos detalhes o protocolo, mas exemplificar como funciona, e mostrar alguns conceitos chave utilizados, principalmente no protocolo Heartbeat implementado utilizando flags específicas e não triviais do protocolo TCP/IP. O TCP foi formalmente definido na RFC 793. Com o passar do tempo, vários erros foram detectados e muitos requisitos mudaram em algumas áreas. Cada máquina compatível com o TCP tem uma entidade de transporte TCP, que pode ser um processo de usuário (como os processos Master e Node) ou parte do kernel que gerencia fluxos e interfaces (o caso da pilha TCP do Sistema Operacional). Os pacotes TCP tem no máximo 64 Kbytes, mas normalmente o tamanho do pacote não ultrapassa o tamanho da MTU Ethernet, de 1500 Bytes. Quando um datagrama IP, contendo um pacote TCP, chega a uma máquina, eles são enviados a pilha TCP do SO da máquina, que tratará o pacote e identificará o processo de destino que tem uma ligação com aquele pacote. Esta ligação para o sistema operacional é um file descriptor, que funciona como uma SAP (Service Access Point) para o aplicativo poder enviar e receber informações encapsuladas no protocolo TCP. A camada de rede IP por si só não pode oferecer qualquer garantia de que os pacotes serão entregues corretamente e na ordem certa, por isso, cabe ao TCP administrar o fluxo de dados que são enviados e recebidos para garantir que os dados não serão perdidos no meio do caminho, ou que chegaram em uma ordenação incorreta. O TCP ao receber os pacotes terá que reorganizá-los, e se for necessário requisitar a retransmissão de pacotes perdidos ou defeituosos, tornando

Figure 5: Visão geral do sistema

a transmissão de dados confiável e garantindo os requisitos que o IP somente não pode oferecer. Vale lembrar que além do protocolo de transporte TCP ainda existe o protocolo UDP, mas este é simplesmente uma extensão do protocolo IP, com um cabeçalho para endereçar o processo de destino na máquina requisitada, não oferecendo nenhuma outra grande vantagem sobre o IP simplesmente. O protocolo TCP define um cabeçalho com vários parâmetros, dos quais nós estaremos comentando a cerca de alguns parâmetros importantes utilizados no trabalho, que normalmente não são utilizados em aplicativos comuns que utilizam o protocolo TCP/IP. O cabeçalho se apresenta da seguinte forma, descrita na Figura 5.

O Objetivo, como já dito, não será de explicar campo a campo, mas mostrar as funcionalidades utilizadas no trabalho. Vamos olhar para os campos URG e Urgent Pointer. Esses os parâmetros do cabeçalho do TCP são utilizados para sinalizar quando um dado urgente está para ser enviado, e qual a sua posição dentro da janela de dados recebidos. Quando o transmissor envia um dado com a flag URG (que é um bit que deve ser marcado como um), o receptor sabe que um dado urgente está para ser enviado, então o campo Urgent Pointer é lido para verificar em que posição do fluxo está o dado urgente. Neste momento, a pilha TCP vai enviar um sinal de interrupção no processo que está com esta conexão ativa, e o processo interromperá o seu fluxo de execução normal para poder atender a requisição urgente do transmissor. Esse é princípio básico que foi utilizado para implementar o protocolo Heartbeat, não tendo assim que utilizar outro protocolo nem desviar o fluxo de sinal para outra conexão. Com essa solução, um processo pode executar o seu funcionamento tranquilamente, e somente se preocupar com atender as requisições de sinal de vida quando for interrompido pelo sinal de URG, enviado pela pilha TCP. Como os dados urgentes estão dentro dos pacotes TCP de dados normais, não é necessário enviar pacotes de controle específicos para obter uma sinalização de vida do processo remoto, economizando assim (na maioria dos casos) banda de rede. As demais funcionalidades do TCP/IP serão descritas no próximo parágrafo, aonde falaremos sobre Sockets, que são (no UNIX) a forma de acesso mais comum ao protocolo TCP/IP.

## 4.2 Sockets

Sockets é a API (Application Program Interface) utilizada nos sistemas UNIX para se ter acesso à comunicação utilizando o protocolo TCP/IP. Na verdade, a API é bem mais ampla, permitindo utilizar tanto UDP, quando TCP e quanto IP puro, mas nós vamos nos reter a parte da API utilizada na comunicação

TCP/IP utilizada no trabalho. A API de sockets se originou como parte do sistema operacional BSD UNIX. O trabalho foi financiado por uma bolsa do governo americano, através da qual a University of California em Berkeley desenvolveu e distribuiu uma versão de UNIX que continha protocolos de ligação inter-redes TCP/IP. Muitos vendedores de computadores portaram o sistema BSD para seu hardware, e o usaram como base em seus sistemas operacionais comerciais. Como vários fabricantes divergiram na forma como implementavam as suas próprias versões de sockets, foi se padronizado novamente os sockets com o padrão POSIX (que atualmente é o utilizado nas distribuições atuais do sistema operacional Linux) que é usado agora como referência para todas as implementações. Como os sockets foram originalmente desenvolvidos como parte do sistema operacional UNIX, eles empregam muitos conceitos encontrados em outras partes do UNIX. Em particular, sockets são integrados com a E/S - uma aplicação se comunica através de uma rotina similar ao socket que forma um caminho para a aplicação transferir dados para um arquivo. Deste modo, compreender sockets exige que se entenda as facilidades de E/S do UNIX. UNIX usa um paradigma open-read-write-close para toda E/S; o nome é derivado das operações de E/S básicas que se aplicam tanto a dispositivos como a arquivos. Por exemplo, um aplicativo deve primeiro chamar open para preparar um arquivo para acesso. O aplicativo então chama read ou write para recuperar dados do arquivo ou armazenar dados no arquivo. Finalmente, o aplicativo chama close para especificar que terminou de usar o arquivo. Quando um aplicativo abre um arquivo ou dispositivo, a chamada open retorna um descritor, um inteiro pequeno que identifica o arquivo; o aplicativo deve especificar o descritor ao solicitar transferência de dados (isto é, o descritor é um argumento para o procedimento de read ou write). Por exemplo, se um aplicativo chama open para acessar um arquivo de nome teste, o procedimento de abertura poderia retornar o descritor 4. Uma chamada subsequente para write que especifica o descritor 4 fará com que sejam escritos dados no arquivo teste; o nome de arquivo não aparece na chamada para write. A comunicação de sockets usa também a abordagem de descritor. Antes de um aplicativo poder usar protocolos para se comunicar, o aplicativo deve solicitar ao sistema operacional que crie um socket que será usado para comunicação. O aplicativo passa o descritor como argumento quando ele chama procedimentos para transferir dados através da rede e o aplicativo não precisa especificar detalhes sobre o destino remoto cada vez que transfere dados. O socket funciona então como um túnel, entre aplicativos, aonde os dados escritos por um aplicativos são lidos pelo outro e vice-versa. Essa abordagem de comunicação simplificando o protocolo de comunicação em um dispositivo de E/S (como um arquivo) tem a limitação que no UNIX não existem distinções entre arquivos binários e arquivos textos, então o socket também não faz esse tipo de distinção, cabendo ao programador saber o início e o fim de seus dados enviados e recebidos. As principais funções da API utilizadas no trabalho foram socket, listen, accept, close, shutdown, connect, send e recv. Neste trabalho, apesar de existirem todas as funções da API C para Perl, foi utilizada a API Orientada a Objetos de Perl para acessar as funções do socket, pois desta forma o código pode se tornar um pouco mais simples.

### 4.3 Protocolo Heartbeat

Este protocolo consiste em fornecer a um socket a funcionalidade de monitorar a conexão existente, em afetar o trânsito normal de informações da aplicação. Vale lembrar que apesar de na API de sockets estar implementada uma funcionalidade de KEEPALIVE, mas esta funcionalidade não permite a parametrização do tempo que o subsistema do API socket deve esperar entre as tentativas de manter a conexão ativa. Sendo o protocolo TCP/IP um protocolo orientado a conexão, mas desenvolvido para trabalhar em ambientes de redes heterogêneas, suportando inclusive falhas temporárias de comunicação, ele por si só não é muito eficiente para verificar a continuidade da conexão, dependendo de timers externos definidos pelo programador. Como normalmente este tipo de checagem é trabalhosa, e não é necessariamente a funcionalidade principal do programa (mas muitas vezes é bastante desejada), foi-se implementada uma API para este protocolo, de tal forma que dado um socket previamente estabelecido, somente é necessário iniciar o Server e o cliente do protocolo Heartbeat e ele monitorará a conexão sem que o sistema seja interferido em seu fluxo normal. Os processos do protocolo são disparados via interrupções de software do sistema no programa (Signals do UNIX) que interrompem a execução normal do programa e executam as rotinas de atendimento dos sinais de aviso de conexão, que neste protocolo são chamados de Beats. Os Beats são um byte marcados no cabeçalho do TCP/IP como mensagens urgentes

Figure 6: Visão geral do sistema

(URG flag) e tem prioridade sobre os dados normais. Além disso, quando um dado urgente é recebido, um Signal URG é disparado contra o processo que está com a conexão aberta (e foi devidamente configurada para tal). Desta forma o programa não precisa de código adicional para receber e enviar dados de sinalização entre as aplicações. Além disso, o protocolo Heartbeat também requer que o Server tenha um timer e um contador, para ter um número máximo de tentativas de conexão e um intervalo entre elas. Um esquema do funcionamento do protocolo pode ser visto na Figura 6.

Neste trabalho o timer foi implementado utilizando a primitiva alarm do sistema, que quando chamada faz com que o sistema operacional envie um Signal ALRM fazendo com que periodicamente o sistema tenha que ser interrompido para executar as rotinas de envio dos Beats. No caso dos sistemas UNIX, quem estiver utilizando a API implementada neste trabalho deve lembrar que a função sleep internamente também utiliza esse recurso para sua implementação, tornando impraticável a sua utilização em conjunto com a API do Heartbeat desenvolvida neste trabalho. Contudo, ao invés de usar o sleep, existem outras soluções que podem produzir o mesmo efeito (por exemplo, utilizando a primitiva select). Esses detalhes de implementação estão documentados no código que está em anexo a este documento.

#### 4.4 IPC (Interprocess Communication)

Programas distribuídos geralmente envolvem alguma forma de comunicação entre processos. Essa comunicação foi padronizada no UNIX como IPC (ou Interprocess Communication). Quando uma aplicação é projetada para utilizar comunicação entre processos, normalmente ela é construída em pequenos pedaços, que executam tarefas distintas e trocam os resultados de suas tarefas entre os demais processos do programa. Normalmente estes programas individualmente tornam-se muito mais simples e eficientes do que um único programa que executa todas as tarefas juntas. Além disso, é mais simples a depuração dos programas e mais fácil de gerenciar equipes que devem desenvolver funcionalidades separadamente uma das outras. Mas o preço por esta comodidade é ter que controlar as informações que são distribuídas, e não mais centralizadas, que agora tem problemas de sincronização entre processos e também de acessos múltiplos a um único recurso compartilhado. Os IPCs no UNIX surgiu como uma idéia de padronizar a maneira como os processos enviavam mensagens entre si, e também tornar simples a comunicação entre processos como a leitura e escrita em um dispositivo de E/S. Vários fabricantes construirão as suas implementações de IPC, mas os mais aceitos foram o BSD e o System V. Para evitar que o IPC se tornasse

inviável, pois cada fabricante teria o seu, então se foi decidido por construir um novo padrão que deveria servir de referência para todas as implementações, que foi incluído no padrão POSIX (que atualmente é o suportado pelas distribuições de Linux). Como todo o trabalho foi desenvolvido utilizando a implementação do IPC para Linux, então vamos abordar somente este padrão. Para simplificar, o IPC pode ser dividido em 4 categorias distintas, que são elas:

- Passagem de mensagens (pipes, filas de mensagens, etc).
- Mecanismos de sincronismo (Mutex, semáforos, etc).
- Memória compartilhada
- Sockets

Como Sockets são um tipo de IPCs que já foram tratados posteriormente, então veremos somente os outros três primeiros tipos que foram utilizados no projeto.

#### **4.4.1 Passagem de mensagens**

Com IPC podemos enviar sinais de interrupção entre dois processos ou criar entre dois processos um túnel de comunicação. Os sinais simplesmente servem para avisar o processo remoto de algum evento externo a sua execução. Estes podem ser atendidos, interrompendo o fluxo normal do programa para executar um procedimento requerido pelo sinal ou simplesmente ignorado. Existem, no entanto, sinais que não podem ser ignorados (como o sinal de KILL, que requer do sistema operacional que a aplicação a qual o sinal foi enviado seja terminada imediatamente). Estes sinais são muito úteis para implementar um programa orientado a eventos externos, aonde o programa não executa nenhuma tarefa até ser avisado pelo IPC que existe alguma coisa a ser feita. Outra funcionalidade seria um procedimento que deve ser executado periodicamente, ou sem tempo previsto. Os túneis entre processos, mais conhecidos como pipes, são como os sockets, mas que fazem comunicação entre dois processos executando na mesma máquina sem a necessidade de um protocolo de rede ou transporte. Na verdade, o túnel criado entre dois processos funciona como um arquivo que pode ser escrito ou lido, e seus dados são lidos. Neste projeto os sinais foram utilizados na implementação dos protocolos de Heartbeat (sinais de URG e ALRM) e na comunicação entre os pais e os filhos (sinais de USR que avisam quando uma consulta foi requisitada pelo cliente e quando a resposta já foi atendida). Já os pipes, foram utilizados para controlar as saídas padrões dos processos pais e filhos.

#### **4.4.2 Mecanismo de sincronismo**

Os mecanismos de sincronismo são utilizados para ter-se um acesso ordenado aos recursos compartilhados via IPC. Caso eles não existam em um programa distribuído, teremos uma grande probabilidade de inconsistência dos dados que estão compartilhados entre os vários processos e constituem o programa. O método mais comum de sincronismo é o utilizado pelos semáforos, que foi neste trabalho utilizado para acessar uma área de memória compartilhada entre os processos do programa Master e entre os processos do programa Nó. Vale lembrar que os semáforos devem ser utilizados com cuidado, pois a má utilização de um semáforo pode levar um programa a um deadlock.

#### **4.4.3 Memória compartilhada**

Para termos uma flexibilidade dos dados que serão compartilhados entre processos, e para se ter uma maior velocidade no acesso aos mesmos, devemos utilizar o artifício de ter uma área de memória que pode ser vista por mais de um programa. Para termos tal facilidade utilizando IPC, no entanto, devemos nos certificar que nenhum processo estará lendo ou atualizando esta área enquanto um outro processo estiver atualizando simultaneamente. Como já dito, esse procedimento é conseguido através da utilização de semáforos e deve-se ter uma cautela especial na programação para evitar deadlocks. Neste projeto, foram utilizadas memórias compartilhadas para implementar os bancos de dados com as informações dos monitores, tanto nos programas nós quanto no programa Master. Este engenho fez com que vários



processos independentes pudessem atualizar uma mesma área de memória, tornando assim o programa modular e podendo realizar tarefas paralelamente no sistema operacional.

## References

- [1] Andrew S. Tanenbaum, Albert S. Woodhull, *Sistemas Operacionais*;
- [2] George Coulouris, Jean Dollimore, Tim Kindberg, *Distributed Systems, Concepts and Design - Second Edition*;
- [3] W. Richard Stevens, *UNIX Network Programming v 2 - Second Edition*;
- [4] W. Richard Stevens, *UNIX Network Programming v 1 - Second Edition*;