Relatório de Estágio

CNPEM - Centro Nacional de Pesquisa em Energia e Materiais

LNLS - Laboratório Nacional de Luz Síncroton

Gustavo **CIOTTO PINTON**
Campinas, 9 de março de 2016

Sumário

1 *Netwok Time Protocol* - NTPv4

1.1 Introdução

O protocolo NTP implementa diversas soluções que permitem a sincronização dos relógios dos computadores pertencentes a uma determinada rede. O protocolo utiliza diversas métricas, descritas nas próximas seções, a fim de determinar quais são as fontes mais seguras e consistentes para obter a melhor sincronização e uma maior precisão. Somadas a essas estatísticas, o NTP faz uso de algoritmos de seleção, *cluster* e combinação que garantem, por sua vez, a determinação dos servidores mais confiáveis a partir de um número finito de amostras provindas de tais fontes.

A troca de mensagens é feita através de pacotes UDP, sendo que o protocolo suporta tanto o IPv4 quanto o IPv6. Apesar do fato de que o protocolo UDP não oferece garantias de entrega e correção de eventuais erros ou duplicatas, o NTPv4 implementa mecanismos, tais como o *On-Wire protocol*, capazes de verificar a consistência dos dados contidos nos pacotes recebidos e, assim, agir corretamente em casos de perdas ou pacotes repetidos.

Neste relatório, serão discutidas as características da versão 4 do NTP, especificadas no RFC5905. Esta versão aprimora alguns aspectos da versão 3 (NTPv3) e adiciona algumas outras funcionalidades, como, por exemplo, a descoberta dinâmica de servidores (*automatic server discovery*), sincronização rápida na inicialização da rede ou depois de falhas (*burst mode*) e uso da criptografia *Public-key*.

1.2 Características do Protocolo

O primeiro aspecto importante do protocolo NTPv4 é a organização dos nós de uma rede. O NTP provê 3 tipos diferentes de variantes e 6 modos de associação, que identificam a função de cada nó que compõe um comunicação. As variantes NTP são, portanto:

i. *server/client*: um cliente envia pacotes a um servidor requisitando sincronização, que responde utilizando o endereço contido nos respectivos pacotes. Nesta variante, servidores fornecem sincronização aos clientes, mas não aceitam sincronizações vindas dos clientes. As associações entre os nós nesta variante são persistentes, ou seja, são criadas na inicialização do serviço e nunca são destruídas.

ii. *symmetric*: neste tipo de variante, um nó se comporta tanto como servidor como cliente, isto é, ele recebe e envia informações de sincronização ao outro nó. Associações deste tipo podem ser persistentes, conforme explicado no item anterior, ou temporárias, isto é, podem ser criadas a partir do recebimento de um pacote e eliminadas após um certo intervalo ou ocorrência de erro. No primeiro caso, adota-se uma associação *ativa*, enquanto que na segunda, adota-se uma *passiva*.

iii. *broadcast*: nesta variante, um servidor *broadcast* persistente envia pacotes que podem ser recebidos por diversos clientes. Quando um cliente recebe um pacote deste tipo, uma associação temporária do tipo *broadcast client* é criada e o cliente recebe sincronização até o fim de um intervalo ou ocorrência de um erro.

O protocolo oferece ainda uma funcionalidade que permite aos clientes descobrirem servidores disponíveis na rede para sincronização. Tal mecanismo é chamado de *Dynamic Server Discovery*, que provê dois tipos especiais de associação: *manycast server* e *manycast client*. Um cliente *manycast* persistente envia pacotes para endereços de *broadcast* ou *multicast* e, caso um *manycast server* receba tais pacotes, ele envia uma resposta a determinado cliente, que, por sua vez, mobiliza uma associação temporária com o respectivo servidor. A fim de descobrir os servidores mais próximos, os clientes enviam pacotes com TTL crescentes, até que o número mínimo de servidores descobertos seja atingido.

O segundo aspecto importante é a implementação dos processos que são executados em um sistema a fim de garantir as funcionalidades apresentadas acima. Cada nó da rede utiliza dois processos dedicados para cada servidor que provê sincronização, além de 3 outros dedicados para escolha dos melhores candidatos e ajuste do relógio. A figura 1 esquematiza a relação entre tais processos. As flechas representam trocas de dados entre processos ou algoritmos.

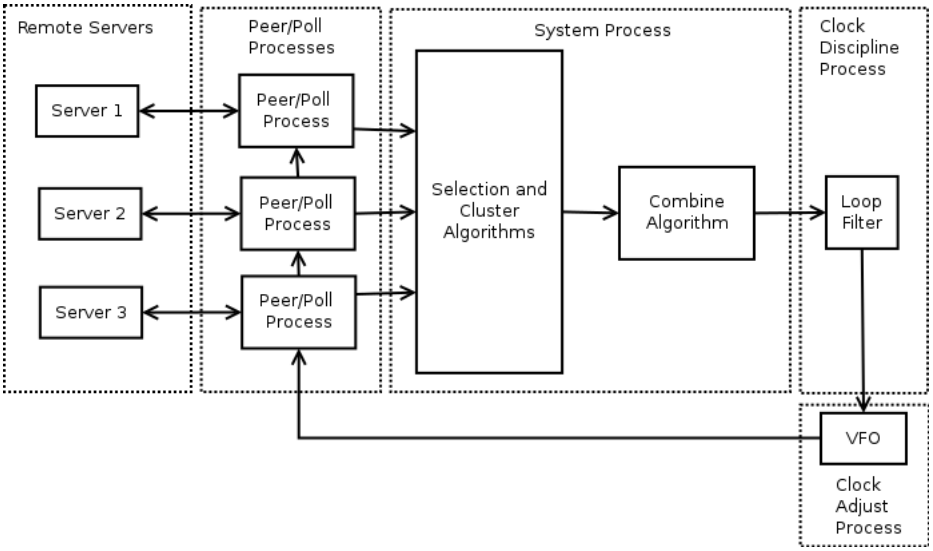

Figura 1: Implementação dos processos executados por um nó da rede.

Cada componente é, portanto, responsável por uma funcionalidade específica oferecida pelo NTP. Temos, assim:

i. *Remote servers*: servidores que fornecem sincronização aos nós da rede. Tais servidores podem pertencer à mesma rede às quais os clientes estão inseridos ou podem ser disponibilizados via Internet por organismos responsáveis por gerenciar e garantir que os relógios apresentem tempos consistentes.

   A fim de diferenciar os diversos servidores utilizados em relação ao seu grau de importância e confiabilidade, o protocolo NTP atribui um nível a cada *server*, chamado de *stratum*. Tal atributo vale 1 para servidores primários, 2 para servidores secundários e assim sucessivamente. À medida que o valor de *stratum* aumenta, a precisão diminui, dependendo do estado da rede. O valor máximo deste atributo é 15 e, portanto, são permitidos até 15 níveis hierárquicos. O valor 0 é reservado pelo protocolo para mensagens de controle e transmissão de estado entre nós. Tais mensagens são chamadas de pacotes *Kiss-o'-Death*.

ii. *Peer/poll processes*: quando um pacote transmitido por um servidor chega em um nó, o *peer process* é chamado. Tal processo então verifica se o pacote é consistente (*On-Wire protocol*, proteção contra perdas e duplicatas) e calcula algumas estatísticas usadas pelos demais processos. Tais estatíticas consistem em:

   - *offset* (θ): deslocamento de tempo do relógio do servidor em relação ao relógio do sistema;
   - *delay* (δ): tempo que o pacote necessita para percorrer toda a rede entre cliente e servidor;
   - *dispersion* (ϵ): erro máximo inerente à medida do relógio do sistema;
   - *jitter* (ψ): raiz do valor quadrático médio dos *offsets* mais recentes.

   O *poll process* é responsável, por sua vez, por enviar pacotes aos servidores a cada intervalo de 2^τ segundos. τ varia de 4 a 17, resultando, assim, em intervalos de 16 segundos a 36 horas. O valor de τ pode variar durante a execução, sendo modificado pelo algoritmo regulador do relógio, que será discutido posteriormente.

iii. *System process*: inclui algoritmos de seleção, clusterização e combinação que utilizam as diversas estatísticas obtidas de cada servidor para determinar os candidatos mais precisos e confiáveis à sincronização do relógio do sistema. As funções de cada algoritmo são, respectivamente:

   – determinar bons candidatos, isto é, determinar quais servidores possuem informações de sincronismo efetivamente importantes;
   – determinar os melhores candidatos dentro do conjunto de servidores julgados importantes no passo anterior;
   – computar estatísticas baseadas nos dados recolhidos dos servidores presentes no subconjunto escolhido pelo algoritmo de clusterição.

iv. *Clock discipline process*: responsável por controlar o tempo e frequência do relógio do sistema;

v. *Clock-adjust process*: roda a cada segundo para comunicar aos demais processos os resultados das correções realizadas no relógio do sistema.

1.3 Exemplo

A rede representada pela figura 2 foi proposta a fim de testar o funcionamento do protocolo NTP. As setas determinam as relações entre os nós e as direções representam quais nós fornecem e/ou recebem sincronização. Todos os computadores pertencem à mesma rede, isto é, assume-se que há um *router* ligando esses nós e responsável pela comunicação com a Internet.

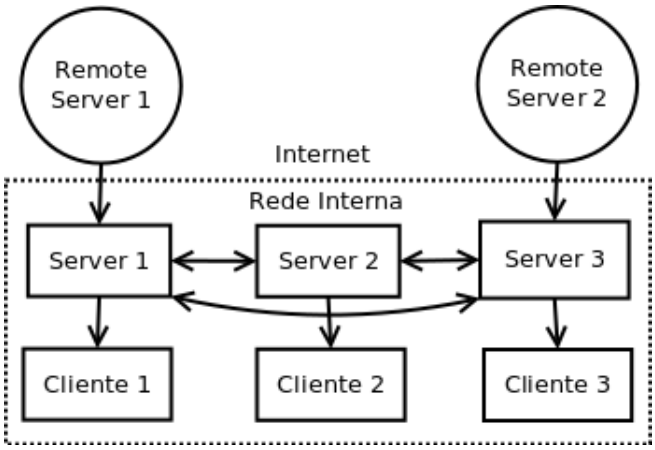

Figura 2: Topologia da rede de teste.

Têm-se, portanto, conforme a subseção anterior:

– associações cliente/servidor entre *remote servers* e *servers*, e entre *servers* e *clients*;

– associações simétricas ativas entre *servers*.

A fim de eliminar a necessidade de implementar essa rede fisicamente, utilizou-se o *software virtualbox*. Cada máquina presente na figura, com exceção dos *remote servers*, foi substituída por uma máquina virtual, cujo sistema operacional é o *Linux Debian 8.2.0 i386*. A escolha deste sistema foi baseada nas características previstas para as máquinas que comporão a infraestrutura do Sirius. Além disso, como pretende-se isolar completamente a rede, isto é, eliminar qualquer comunicação com a Internet, os *remote servers* serão substituídos pelos respectivos servidores. Dessa maneira, tais servidores utilizarão seus próprios relógios como fonte primária de sincronismo.

Foi adotado que a rede teria endereço `10.0.0.0/24`, os servidores possuiriam endereços do tipo `10.0.0.10x`, sendo x o dígito que caracteriza o servidor (1, 2 ou 3), e `10.0.0.0y1` para os clientes, sendo y o equivalente de x.

Após a configuração de rede das respectivas máquinas virtuais, a instalação do NTP pode prosseguir. Inicialmente, é necessário fazer a instalação do pacote `ntp` através do comando

```
$ sudo apt-get install ntp
```

Estão incluídos neste pacote, os programas `ntpd`, `ntpq` e `ntpqc`. `ntpd`, ou *NTP Daemon*, roda continuamente no sistema e é responsável pela troca de mensagens com os diversos servidores ou clientes, de acordo com as configurações, enquanto que `ntpq` e `ntpqc` (o *q* refere-se a *query*) são utilizados para verificar o estado das variáveis e alterar configurações do *daemon*.

Quando é iniciado, o *daemon* retira as suas configurações do arquivo `/etc/ntp.conf`. Cada nó da rede deve, portanto, configurar esse arquivo de acordo com as funções que desempenha.

A configuração dos clientes é simples: basta adicionarmos a linha

```
server 10.0.0.10y iburst
```

ao arquivo de configurações. A opção `iburst` é uma otimização fornecida pelo protocolo que agiliza a sincronização inicial. Essa opção faz com que o intervalo de envio de pacotes seja reduzido e a quantidade de pacotes enviados seja aumentada caso o servidor não esteja acessível.

Para o *server 2*, é necessário adicionarmos as duas linhas seguintes.

```
peer 10.0.0.101
peer 10.0.0.103
```

Essas duas linhas criam associações simétricas ativas entre o *server 2* e os outros servidores. Dessa maneira, eles poderão trocar informações de sincronização entre si.

Para os servidores 1 e 3, além das duas linhas contendo a opção `peer`, é necessário também adicionar as duas linhas abaixo:

```
server 127.127.0.0
fudge 127.127.0.0 stratum 1
```

A primeira opção configura o relógio local do sistema como uma fonte de sincronização, enquanto que a segunda aumenta a sua hierarquia. Se o atributo `stratum` vale 1, logo a prioridade do respectivo servidor torna-se máxima.

Para iniciar o *daemon*, basta executar o comando abaixo em cada máquina.

```
$ sudo /etc/init.d/ntp restart
```

Os sistemas levam alguns minutos para se sincronizarem. Para verificar o estado das conexões e da sincronização, utiliza-se o programa `ntpq` através do comando

```
$ ntpq -p
```

A opção `-p` lista todos os nós utilizados para sincronização do relógio local. Para o *Server 1*, espera-se uma saída parecida com a figura 3 (não há garantias que seja idêntica, visto que os algoritmos que determinam as fontes que serão utilizadas para sincronização são baseados em fatores que podem variar dependendo do estado da rede).

```
lnls@lnls:~$ ntpq -p
     remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
 LOCAL(0)        .LOCL.           1 l   71m   64    0    0.000    0.000   0.000
+10.0.0.102      10.0.0.103       3 u  335 1024  377    0.365   38.989  11.778
*10.0.0.103      LOCAL(0)         2 u   27   64  376    1.969    5.589 2099.90
```

Figura 3: Resultado do comando `ntpq -p` no *Server 1*.

O caracter $\star$ indica quais dos servidores está sendo usado como fonte de sincronismo e o + indica os outros possíveis candidatos válidos (os chamados *truechimers*). É possível também encontrar o caracter −, representando servidores que provêm fontes de sincronismo inválidas (chamadas também de *falsetickers*). `refid` representa o *id* da refêrencia de sincronismo do respectivo servidor. Por exemplo, o nó `10.0.0.103` usa como referência o relógio `LOCAL(0)`, que o próprio relógio da máquina. `st` é o valor do *stratum*. Conforme esperado, servidores secundários, como `10.0.0.103`, que possuem somente uma fonte de sincronismo, têm esse atributo setado para 2. `when` corresponde ao tempo, em segundos, da última troca de pacotes e `poll` é o tempo em segundos até o próximo envio. `reach`, em representação octal, indica se as 8 últimas tentativas de comunicação foram bem-sucedidas ou não. Esse atributo funciona como um registrador que é deslocado para a esquerda a cada nova comunicação. Se ela for bem-sucedida, o *bit* mesmo significativo é setado para 1, senão, para 0. As demais colunas são as estatísticas comentadas na subseção anterior.

A figura 4 é a saída do comando `ntpq -p` no cliente conectado no *server 1*. Conforme esperado, ele obtem corretamente a sincronização deste servidor.

```
lnls@lnls:~$ ntpq -p
     remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
*10.0.0.101      10.0.0.103       3 u  193  512  377    0.317  -16.632 1453.58
```

Figura 4: Resultado do comando `ntpq -p` no *Client 1*.

1.4 Aplicação ao projeto *Sirius*

2 EPICS Archiver Appliance

2.1 Introdução

O EPICS Archiver Appliance, desenvolvido pelo instituto americano *National Accelerator Laboratory (SLAC)*, é capaz de monitorar e arquivar um grande número de váriaveis, as chamadas *PVs*, geradas por servidores EPICS presentes na rede. O sistema fornece também opções de configuração de um largo conjunto de parâmetros referentes ao armazenamento e monitoramento. Uma *appliance* é composta basicamente por quatro módulos distintos, sendo eles:

- *Management*: provê as ferramentas necessárias para a gerência da *appliance*. Permite, por exemplo, adicionar ou remover *PVs* à lista de variáveis a serem arquivadas;
- *Engine*: realiza a integração entre os módulos;
- *Data Retrieval*: módulo responsável por recuperar os dados das *PVs* arquivadas;
- *ETL*: responsável por extrair os dados e tranformá-los a fim de que as aplicações possam processá-los posteriormente;

A figura 5 esquematiza o modo de funcionamento do *EPICS Archiver Appliance*.

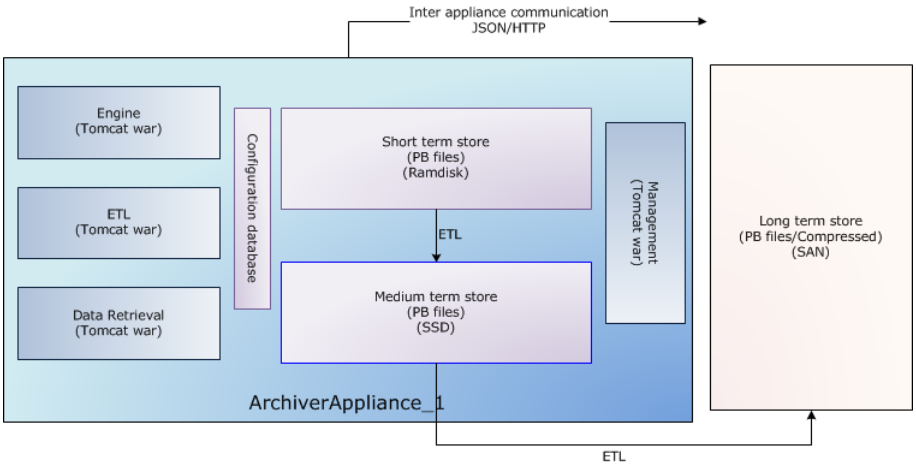

Figura 5: Modo de funcionamento de uma *appliance*

O instituto desenvolvedor da aplicação sugere que cada módulo seja lançada em sua própria instância *Tomcat*. Em adição, ele propõe a divisão da unidade de armazenamento em 3 outras unidades, de acordo com a frequência que os dados são salvos. Essas unidades são divididas *short-term*, *medium-term* e *long-term storage*, cujas frequências de armazenamento são, respectivamente, a cada hora, diária e anual. Essas configurações podem ser modificadas através da modificação de arquivos específicos, explicados nas próximas subseções.

Em um ambiente composto por diversos servidores EPICS e milhares de variáveis a serem monitoradas, tal como o *Sirius*, um sistema capaz de automizar e agilizar o armazenamento e recuperação de dados se torna fundamental para o monitoramento de eventuais problemas. Sendo assim, as próximas seções são dedicadas à instalação e exploração dos recursos disponíveis nesta aplicação.

2.2 Instalação

Esta subseção é dedicada às etapas necessárias para a instalação do EPICS Archiver Appliance.

2.2.1 Instalação das dependências

A aplicação necessita, além da própria base de bibliotecas EPICS, da instalação do *java-jdk 8* e de servidores *MySQL* e *Apache Tomcat*. A instalação do *Apache HTTP Server* é opcional e deve ser utilizada somente se o balanceamento de requisições entre as *appliances* for pretentida. Nesse caso, vamos utilizar o módulo *mod_proxy_balancer*.

i. **Instalação do EPICS**: é necessário inicialmente fazer o *download* e compilar as bibliotecas EPICS. Escolha um diretório de acordo com sua preferência (referenciado nesse documento por `$EPICS_DIR`) e execute os comandos abaixo. Alguns erros podem ocorrer na execução do comando `make`, resultantes da falta de algumas bibliotecas no sistema. Faça as respectivas instalações e repita o processo.

```
$ cd $EPICS_DIR
$ wget http://www.aps.anl.gov/epics/download/base/baseR3.14.12.5.tar.gz
$ tar -xvzf baseR3.14.12.5.tar.gz
$ rm baseR3.14.12.5.tar.gz
$ cd base-3.14.12.5
$ make
```

Adicione as seguintes variáveis de ambiente ao arquivo `.bashrc` do seu usuário (geralmente encontrado em `/home/user/`). Esse arquivo de configuração é consultado toda vez que um usuário executa um novo *shell*.

```
export PATH=$EPICS_DIR/base-3.14.12.5/bin/linux-x86_64:$PATH
export EPICS_BASE=$EPICS_DIR/base-3.14.12.5
export EPICS_HOST_ARCH=linux-x86_64
```

Enfim, atualize a sessão com

```
$ source /home/user/.bashrc
```

Para testar a instalação, use o comando `caget` com alguma variável disponível na sua rede.

```
$ caget variavel
```

ii. **Instalação do Java 8**: execute o comando abaixo para instalar o *Java Development Kit*.

```
$ sudo apt-get install openjdk-8-jdk
```

O EPICS Archiver Appliance necessita de duas variáveis de ambiente referentes ao *Java*. Adicione-as no arquivo *.bashrc*, após as variáveis de ambiente EPICS.

```
export JAVA_HOME="/usr/lib/jvm/java-8-openjdk-amd64"
export JRE_HOME="/usr/lib/jvm/java-8-openjdk-amd64/jre"
```

iii. **Instalação do MySQL Server**: para instalar um servidor *MySQL*, basta executar o comando abaixo. Durante o processo de instalação, será requisitada a senha para o usuário *root* do servidor.

```
$ sudo apt-get install mysql-server
```

iv. **Instalação do Apache Tomcat**: defina um diretório onde os arquivos serão instalados e execute os comandos abaixo. Esse diretório será referenciado pela variável de ambiente `$TOMCAT_DIR`.

```
$ cd $TOMCAT_DIR
$ wget http://ftp.unicamp.br/pub/apache/tomcat/tomcat-8/v8.0.32/bin/apache-tomcat
   -8.0.32.tar.gz
$ tar -xvzf apache-tomcat-8.0.32.tar.gz
$ rm apache-tomcat-8.0.32.tar.gz
$ cd apache-tomcat-8.0.32/
```

Defina a variável de ambiente `$CATALINA_HOME`, responsável por referenciar o diretório *apache-tomcat-8.0.32/* recém criado. Adicione a seguinte linha no arquivo `.bashrc` logo abaixo das variáveis de ambiente adicionadas na instalação do *Java*.

```
export CATALINA_HOME = $TOMCAT_DIR/apache-tomcat-8.0.32
```

v. **Instalação do Apache HTTP Server**: Para instalar o servidor *Apache*, faça o *download* do pacote.

```
$ cd $INSTALL_DIR
$ wget http://ftp.unicamp.br/pub/apache//httpd/httpd-2.4.18.tar.gz
$ tar -xvzf httpd-2.4.18.tar.gz
$ rm httpd-2.4.18.tar.gz
$ cd httpd-2.4.18
```

Algumas dependências devem ser instaladas:

```
$ sudo apt-get install libapr1-dev libaprutil1-dev
```

Será necessário instalar também a `libpcre`:

```
$ cd $INSTALL_DIR
$ wget ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/pcre-8.37.tar.gz
$ tar -xvzf pcre-8.37.tar.gz
$ rm pcre-8.37.tar.gz
$ cd pcre-8.37
$ ./configure --prefix=$INSTALL_DIR/pcre
$ make
$ make install
```

Finalmente, execute os *scripts* para instalar o *Apache http*:

```
$ cd $INSTALL_DIR/httpd-2.4.18
$ ./configure --prefix=$INSTALL_DIR/httpd --with-pcre=$INSTALL_DIR/pcre
$ make
$ make install
```

Para habilitar o módulo `mod_proxy_balancer`, descomente as seguintes linhas no arquivo `$INSTALL_DIR/httpd/conf/httpd.conf`:

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
LoadModule lbmethod_byrequests_module modules/mod_lbmethod_byrequests.so
```

Adicione as linhas abaixo no mesmo arquivo. Quando uma requisição for realizada pelo endereço `localhost/lnls-archiver`, um dos endereços dentro da *tag* `<Proxy>` será chamado. No nosso exemplo, colocamos o endereço do módulo `retrieval` de apenas uma *appliance*.

```
<Proxy "balancer://appl">
    BalancerMember "http://localhost:11998/retrieval"
</Proxy>
ProxyPass "/lnls-archiver" "balancer://appl"
ProxyPassReverse "/test" "balancer://appl"
```

Inicie o servidor com o comando:

```
$ sudo $INSTALL_DIR/httpd/bin/apachectl start
```

O servidor *Apache* funciona agora como um divisor de carga entre as diversas *appliances* da rede.

2.2.2 Instalação do EPICS Archiver Appliance

Tendo instalado todas as dependências, é necessário ainda configurar alguns parâmetros e instalar os pacotes referentes ao arquivador. Para tal, crie uma pasta onde serão instalados os pacotes. Esse diretório será referenciado por `$INSTALL_DIR` neste documento.

i. Crie uma pasta e faça o *download* dos pacotes do arquivador, segundo os comandos abaixo.

```
$ mkdir $INSTALL_DIR/epics-archiver-appliance/
$ cd $INSTALL_DIR/epics-archiver-appliance/
$ wget
https://github.com/slacmshankar/epicsarchiverap/releases/download/v0.0.1
    _SNAPSHOT_26-January-2016/archappl_v0.0.1_SNAPSHOT_26-January-2016T18-03-47.
    tar.gz
$ tar -xvzf archappl_v0.0.1_SNAPSHOT_26-January-2016T18-03-47.tar.gz
$ rm archappl_v0.0.1_SNAPSHOT_26-January-2016T18-03-47.tar.gz
```

Nesta pasta estão os arquivos *.war* usados para lançar os módulos e alguns *scripts* sugeridos pelo desenvolvedor para auxiliar na instalação.

ii. Crie uma pasta onde as instâncias *Tomcat* de cada módulo serão armazenadas e crie o arquivo *lnls_appliances.xml*. Vamos chamar essa pasta de `epics-appliances/`.

```
$ mkdir $INSTALL_DIR/epics-appliances/
$ nano $INSTALL_DIR/epics-appliances/lnls_appliances.xml
```

Copie as seguintes linhas para esse arquivo. Se a configuração do *Apache httpd* não foi realizada, então o endereço de `<data_retrieval_url>` deve ser igual ao endereço de `<retrieval_url>`. Senão:

```
<appliances> <appliance>
     <identity>lnls_epics_appliance</identity>
     <cluster_inetport>localhost:12000</cluster_inetport>
     <mgmt_url>http://localhost:11995/mgmt/bpl</mgmt_url>
     <engine_url>http://localhost:11996/engine/bpl</engine_url>
     <etl_url>http://localhost:11997/etl/bpl</etl_url>
     <retrieval_url>http://localhost:11998/retrieval/bpl</retrieval_url>
     <data_retrieval_url>http://localhost/lnls-archiver</data_retrieval_url>
   </appliance>
 </appliances>
```

Esse arquivo especifica os endereços e portas de cada um dos quatro módulos, bem como o nome da *appliance* (`lnls_epics_appliance`). Os números das portas foram escolhidos aleatoriamente, mas, por sugestão do desenvolver, deve-se adotar uma sequência crescente a partir da porta do módulo `mgmt`, que, no nosso caso, vale 11995. `<data_retrieval_url>` é o endereço que deverá ser utilizado a fim de enviar e obter requisões HTTP e JSON, conforme figura 5 e `<cluster_inetport>` é a combinação `TCPIP address:port` usado para a comunicação entre as *appliances*. Por fim, adicione as váriaveis de ambiente ao arquivo `.bashrc`.

```
export ARCHAPPL_APPLIANCES=$INSTALL_DIR/epics-appliances/lnls_appliances.xml
export ARCHAPPL_MYIDENTITY="lnls_epics_appliance"
```

O desenvolvedor sugere também trocar a porta padrão usada pelo *Apache Tomcat* pela porta usada pelo módulo `mgmt`, isto é, 11995, e comentar as linhas sobre o uso do conector AJP. Para isso, siga a sequência abaixo.

```
$ nano $CATALINA_HOME/conf/server.xml
### Edite a porta de forma a obter
[...]
<Connector port="11995" protocol="HTTP/1.1"
               connectionTimeout="20000"
               redirectPort="8443" />
[...]
### Comente o trecho
<!-- Define an AJP 1.3 Connector on port 8009
    <Connector port="8009" protocol="AJP/1.3" redirectPort="8443" /> -->
```

Execute o *script* disponibilizado pelo desenvolvedor para criar as 4 instâncias do *Apache Tomcat*.

```
$ source /home/user/.bashrc
$ cd $INSTALL_DIR/epics-archiver-appliance/install_scripts
$ ./deployMultipleTomcats.py $INSTALL_DIR/epics-appliances
```

Serão criadas 4 pastas dentro de `$INSTALL_DIR/epics-appliances`, uma para cada módulo.

iii. É possível também alteração as configurações de *log*. O desenvolvedor sugere que sejam executados os seguintes comandos:

```
$ nano $CATALINA_HOME/lib/log4j.properties

### Arquivo log4j.properties
```

```
# Set root logger level and its only appender to A1.
log4j.rootLogger=ERROR, A1
log4j.logger.config.org.epics.archiverappliance=INFO
log4j.logger.org.apache.http=ERROR

# A1 is set to be a DailyRollingFileAppender
log4j.appender.A1=org.apache.log4j.DailyRollingFileAppender
log4j.appender.A1.File=arch.log
log4j.appender.A1.DatePattern='.'yyyy-MM-dd

# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n
```

iv. O próximo passo é definir o arquivo `policies.py`. Esse arquivo especifica as regras de armazenagem que serão utilizadas pelo arquivador. O arquivo fornecido pelo desenvolver define, por exemplo, três modalidades de armazenamento, que foram exemplificadas na subseção **Introdução**. É possível ajustar neste arquivo parâmetros que alteram a taxa de armazenamento de uma PV dependendo de sua taxa de aquisição, por exemplo. Para um primeiro instante vamos usar esse arquivo fornecido, que está disponível na pasta `WEB-INF/classes` dentro do arquivo `mgmt.war` em `$INSTALL_DIR/epics-archiver-appliance/`. Copie-o para `$INSTALL_DIR/epics-appliances/` e renomeie-o para `lnls_policies.xml`. Enfim, defina variáveis de ambiente usada por este arquivo.

```
export ARCHAPPL_POLICIES=$INSTALL_DIR/epics-appliances/lnls_policies.py
export ARCHAPPL_SHORT_TERM_FOLDER=$INSTALL_DIR/epics-storage
export ARCHAPPL_MEDIUM_TERM_FOLDER=$INSTALL_DIR/epics-storage
export ARCHAPPL_LONG_TERM_FOLDER=$INSTALL_DIR/epics-storage
```

As três últimas variáveis devem ser corretamente configuradas de acordo com a disponibilidade de equipamentos, por exemplo. No nosso caso, um único diretório é usado para os três tipos de armazenamento, sendo ele `epics-storage/`.

v. A próxima etapa é a criação de tabelas usadas pelo EPICS Archiver Appliance. Vamos criar uma base chamada `lnls_appliance_database` e o usuário `lnls_user`, com todos os direitos de acesso a ela. No nosso caso, a senha do usuário é `app1`.

```
$ mysql -u root -p
>> CREATE DATABASE lnls_appliance_database;
>> GRANT ALL ON lnls_appliance_database.* TO 'lnls_user'@localhost IDENTIFIED BY
'app1';
```

Acesse o servidor *mysql* com esse usuário e execute o *script* de criação das tabelas disponibilizado pelo desenvolvedor:

```
$ mysql -u lnls_user -p
>> USE lnls_appliance_database
>> source
$INSTALL_DIR/epics-archiver-appliance/install_scripts/archappl_mysql.sql
```

vi. É necessário fazer o *download* do conector *MySQL* usado pelo *Apache Tomcat*.

```
$ wget
https://dev.mysql.com/get/Downloads/Connector-J/mysql-connector-java-5.1.38.tar.
    gz
```

Abra este arquivo e extraia `mysql-connector-java-5.1.38-bin.jar` para o diretório `$CATALINA_HOME/lib/`. Abra o arquivo *conf/context.xml* e adicione dentro de `<Context>`:

```
$ nano $CATALINA_HOME/conf/context.xml

<Context ...>
        <Resource   name="jdbc/archappl"
              auth="Container"
              type="javax.sql.DataSource"
              factory="org.apache.tomcat.jdbc.pool.DataSourceFactory"
              username="lnls_user"
              password="app1"
              testWhileIdle="true"
              testOnBorrow="true"
              testOnReturn="false"
              validationQuery="SELECT␣1"
              validationInterval="30000"
              timeBetweenEvictionRunsMillis="30000"
              maxActive="10"
              minIdle="2"
              maxWait="10000"
              initialSize="2"
              removeAbandonedTimeout="60"
              removeAbandoned="true"
              logAbandoned="true"
              minEvictableIdleTimeMillis="30000"
              jmxEnabled="true"
              driverClassName="com.mysql.jdbc.Driver"
              url="jdbc:mysql://localhost:3306/lnls_appliance_database"
         />
</Context>
```

vii. É necessário copiar e extrair todos os arquivos *.war* presentes em `epics-archiver-appliance/` para os respectivos diretórios `webapps/`. Para tal, definimos:

```
export DEPLOY_DIR=$INSTALL_DIR/epics-appliances
export WARSRC_DIR=$INSTALL_DIR/epics-archiver-appliance
```

E executamos o seguinte trecho para extrair os arquivos *.war*:

```
$ source /home/user/.bashrc

$ pushd ${DEPLOY_DIR}/mgmt/webapps && rm -rf mgmt*; cp ${WARSRC_DIR}/mgmt.war .;
    mkdir mgmt; cd mgmt; jar xf ../mgmt.war; popd;
$ pushd ${DEPLOY_DIR}/engine/webapps && rm -rf engine*; cp ${WARSRC_DIR}/engine.
    war .; mkdir engine; cd engine; jar xf ../engine.war; popd;
$ pushd ${DEPLOY_DIR}/etl/webapps && rm -rf etl*; cp ${WARSRC_DIR}/etl.war .;
    mkdir etl; cd etl; jar xf ../etl.war; popd;
$ pushd ${DEPLOY_DIR}/retrieval/webapps && rm -rf retrieval*; cp ${WARSRC_DIR}/
    retrieval.war .; mkdir retrieval; cd retrieval; jar xf ../retrieval.war; popd;
```

viii. Enfim, lance os 4 *Tomcats*:

```
$ source /home/user/.bashrc

$ export CATALINA_BASE=${DEPLOY_DIR}/mgmt;${CATALINA_HOME}/bin/catalina.sh start
$ export CATALINA_BASE=${DEPLOY_DIR}/engine;${CATALINA_HOME}/bin/catalina.sh
    start
$ export CATALINA_BASE=${DEPLOY_DIR}/etl;${CATALINA_HOME}/bin/catalina.sh start
$ export CATALINA_BASE=${DEPLOY_DIR}/retrieval; ${CATALINA_HOME}/bin/catalina.sh
    start
```

ix. O arquivador é acessado através da URL

```
http://localhost:11995/mgmt/ui/index.html
```

2.3 Uso do CS Studio no monitoramento

O *CS Studio* pode ser usado para monitorar a *appliance*. Para isso, entre em `Edit > Preferences` e acesse o item `CSS Applications > Trends > Data Browser`. No campo *Archive Data Server URLs*, adicione o endereço contido em `<data_retrieval_url>` do arquivo configurado no item ii da subseção 2.2.2, substituindo o protocolo `http://` por `pbraw://`. Escreva qualquer *Server alias*. Na tabela *Default Archive Data Sources*, adicione o mesmo endereço e aperte *Ok* para salvar as alterações.

É necessário alterar a perspectiva do *CS Studio*. Acesse `Windows > Open Perspective` e escolha **Data Browser**. Na aba *Archive Search*, escreva a *URL* configurada anteriormente e no campo *Pattern*, escreva o nome das variáveis arquivadas que deseja monitorar. Por exemplo, se escrevermos `MTTemp*`, todas as variáveis arquivadas com este início poderão ser acessadas. Clique com o botão direito na variável desejada e acesse `Process Variable > Data Browser`.

2.4 Acessando a *appliance* com *Python*

A *appliance* pode ser acessada através de requisições *JSON* realizadas por um módulo escrito em *Python*, por exemplo. Abaixo, está apresentada uma classe que foi escrita a fim de obter dados e informações de variáveis arquivadas.

```
import time
import urllib2
import json

class JsonRequester ():

    def __init__(self, data_retrieval_url, mgmt_url):
        self.data_retrieval_url = data_retrieval_url
        self.mgmt_url = mgmt_url

    def json_request_variables(self, variables_prefix):

        url_json = self.mgmt_url + 'bpl/getPVStatus?pv=' + variables_prefix
        req = urllib2.urlopen(url_json)
        data = json.load(req)

        return data

    def json_request_data(self, variable, from_date, to_date):

        retrieval_url = self.data_retrieval_url + "/data/getData.json?"
        pv_name = ("pv=" + variable).replace(':', '%3A')

        to_date =   ("&to=" + time.strftime("%Y-%m-%dT%H:%M:%S",to_date) +
                                    ".000Z").replace(':', '%3A')
        from_date = ("&from=" + time.strftime("%Y-%m-%dT%H:%M:%S",from_date) +
                                    ".000Z").replace(':', '%3A')

        url_json = retrieval_url + pv_name + from_date + to_date

        req = urllib2.urlopen(url_json)
        data = json.load(req)

        secs = [x['secs'] for x in data[0]['data']]
        vals = [x['val'] for x in data[0]['data']]

        return secs, vals
```

O método construtor recebe 2 *strings* como parâmetros. *data_retrieval_url* e *mgmt_url* estão contidos no arquivo *lnls_appliances.xml* e representam, respectivamente, os endereços dos *servlets* de obtenção de dados e gerenciamento da *appliance*. A primeira *url* será usada para recuperar os dados e a segunda, para obter as informações relativas às variáveis arquivadas.

O método *json_request_variables* é responsável por retornar informações de uma ou várias variáveis, cujo nome (no caso de uma pesquisa de uma única variável) ou prefixo (parte comum ao nome de diversas variáveis) é passado como parâmetro. Para tal, utiliza o método `getPVStatus`, que é disponível no *servlet* de gerenciamento e acessível via a *url* `mgmt_url`. Esse método também recebe como parâmetro nomes ou prefixos de variáveis, especificados após o trecho `pv=` na requisição *json*. Para recuperar todas as variáveis arquivadas por uma *appliance* que comecem com o prefixo `MBTemp`, por exemplo, bastar utilizarmos `pv=MBTemp*` na requisição. Uma vez construída a *url*, é necessário utilizar as bibliotecas *python* `urllib2` e `json`, que realizam a comunicação com os *servlets*.

O método *json_request_data* retorna os dados arquivados para uma determinada variável, passada como parâmetro. Além dela, essa função recebe dois outros valores, sendo eles objetos do tipo *time*, cuja implementação reside no módulo `time`. Esses parâmetros representam, por sua vez, as fronteiras do intervalo de tempo para o qual se deseja recuperar os dados, sendo que `from_date` é a data mais antiga e `to_date`, a mais recente. Se `to_date` vale `None`, então o sistema o interpreta como o tempo no qual a chamada da função foi feita. Os dados são recuperados através do método `getData`, disponível no *servlet* `retrieval` da *appliance*. Antes de realizarmos a requisição, é necessário traduzir os objetos *time* para o formato aceito pela servidor. Sendo assim, utiliza-se o método `strftime` do módulo `time` que retorna a representação *string*, segundo a máscara especificada (no nosso caso, `%Y-%m-%dT%H:%M:%S`), do objeto passado como parâmetro. A requisição é, enfim, realizada através dos mesmos métodos que foram usados na função anterior.

O servidor envia todos os dados, isto é, valores e datas do respectivo evento, em único vetor de dicionários. Por este motivo, é necessário processarmos essa estrutura antes de retorná-la ao programa que chamou o método. Os valores são acessíveis pelo índice `val` e seu tipo depende da aplicação. A data dos eventos relativos aos valores são recuperados pelo índice `secs` e são números inteiros que contém o número de segundos desde uma data de referência (1 de Janeiro de 1970) usada no *servlet*. É necessário notar que as datas retornadas estão em *UTC*, portanto é exigido que tais valores sejam convertidos para a o fuso local.

Uma interface gráfica foi implementada, usando os módulos *Qt*, a fim de testarmos a comunicação. Ela possui um gráfico, onde serão mostrados os dados recuperados, uma caixa de opções, que possui todas as variáveis arquivadas na *appliance*, e componentes para seleção das datas de início e fim do intervalo desejado. A figura 6 representa o resultado da implementação.

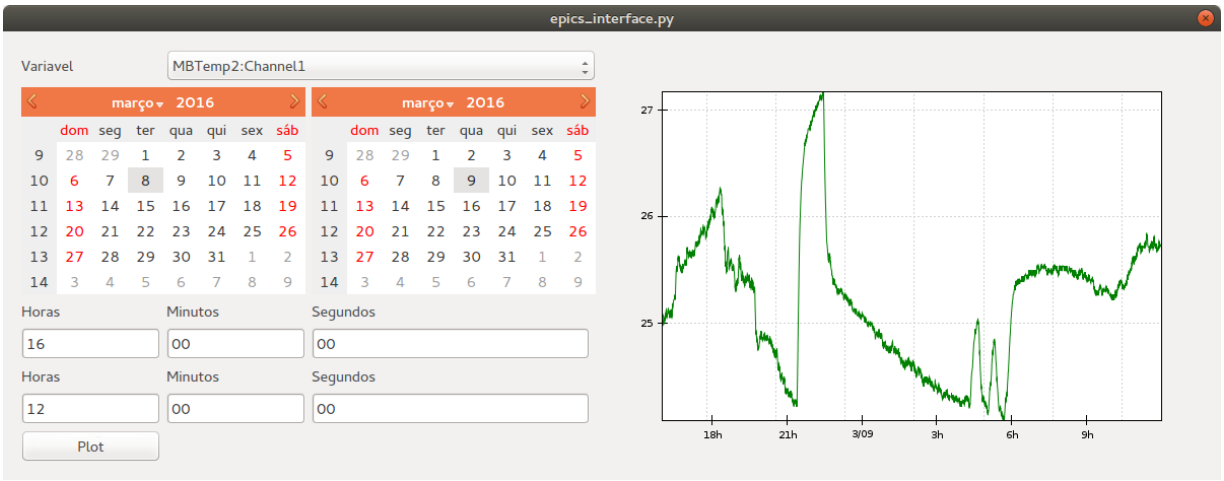

Figura 6: Interface *Qt* implementada em *python*.

3 Best Ever Alarm System Toolkit - BEAST

3.1 Introdução

Em um ambiente composto por centenas de milhares de variáveis EPICS, como o que será implementado no *Sirius*, a necessidade de um sistema capaz de monitorar quais variáveis encontram-se em estados errôneos torna-se imprescindível. Sendo assim, o monitor de alarmes *BEAST*, do inglês *Best Ever Alarm System Toolkit* e desenvolvido pelo laboratório americano *Oak Ridge National Laboratory*, representa uma solução capaz de gerenciar e controlar os alarmes gerados pelos servidores EPICS disponíveis na rede. Tal sistema é implementado em *Java* e é baseado no ambiente gráfico de desenvolvimento *Eclipse*. A arquitetura do sistema está represetada na figura 7, logo abaixo.

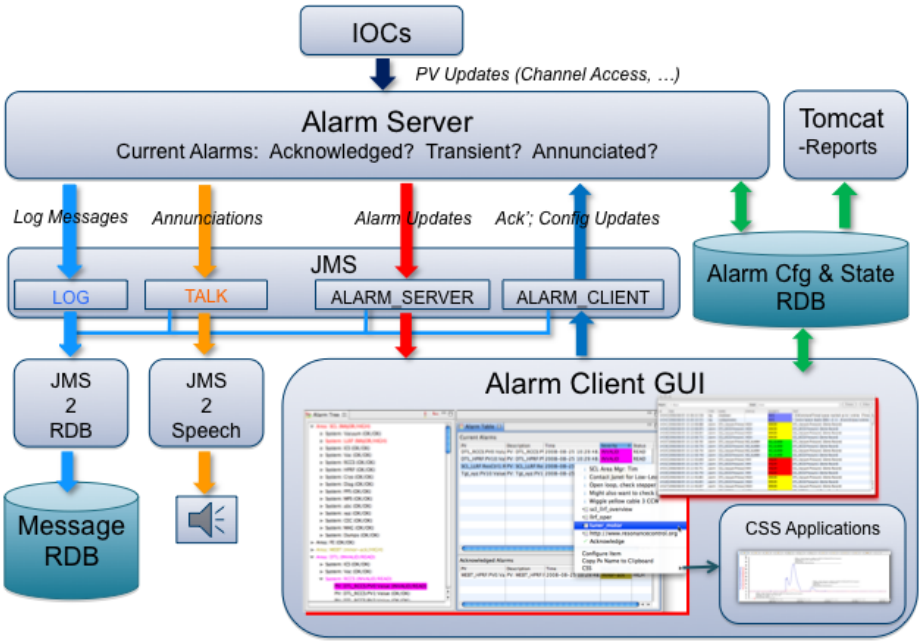

Figura 7: Implementação do sistema de monitoramento de alarmes *BEAST*.

É possível distinguir diversos componentes na figura acima:

i. *Alarm server*: o servidor é responsável por tarefas fundamentais no monitoramento de alarmes. Cabe a ele a leitura da configuração dos alarmes armazenada no banco de dados *Alarm Cfg & State RDB*, conectar-se às respectivas variávies, monitorar suas mudanças de estado e gerar alarmes quando necessário e desativá-los logo que um operador toma conhecimento do problema. Esse módulo permite que um número variável de clientes se conecte a ele. Uma variável pode adotar duas configurações distintas, sendo elas *latch* e *annunciate*. Para a primeira, o servidor mantém o alarme de maior gravidade, mesmo que o estado da variável não seja atualmente errôneo, até que ele seja reconhecido manualmente pelo operador. A fim de impedir um grande volume de alarmes gerados, é possível habilitar as opções *delay*, que aciona o alarme somente se o estado errôneo da variável se mantiver durante o intervalo de tempo especificado, e *count*, que aciona o alarme se tal estado for detectado mais vezes que o valor especificado. A segunda configuração ativa o alarme somente quando a *PV* apresentar valor inválido, sendo que ele é desativado logo que tal variável voltar à sua faixa de operação esperada.

ii. *Alarm Cfg & State RDB*: banco de dados relacional, como um servidor *MySQL* por exemplo, onde serão armazenadas as configurações de alarme e o atual estado de todos os alarmes.

iii. *Java Message Service - JMS*: utilizado para a comunicação entre diferentes módulos. No projeto, foi empregada a implementação realizada pelo *Apache Software Foundation* chamada de *Apache ActiveMQ*. O sistema utiliza 4 *topics* distintos, sendo eles:

- *ALARM_SERVER*: utilizado pelo servidor para publicar atualizações nos estados dos alarmes de acordo com a configuração de cada variável.
- *ALARM_CLIENT*: permite que clientes notifiquem atualizações de configuração e reconhecimento de alarmes.
- *TALK*: dedicado para anunciar mensagens.

iv. *Alarm Client GUI*: baseado na interface gráfica do *Eclipse*, oferece três opções de monitoramento:

- *Alarm table*: mostra os alarmes em duas tabelas distintas contendo aqueles reconhecidos (*acknowledged alarms*) e aqueles que ainda estão acionados (*active alarms*).
- *Alarm tree*: essa opção de visualização oferece uma visão hierarquica dos alarmes, sendo organizada, do nível mais alto para o menor, em áreas, sistemas, subsistemas e variáveis. Oferece opções para configurar, remover ou adicionar variáveis no nível desejado. O estado do alarme de cada item é mostrado por uma cor e por uma anotação, sendo composta por três sentenças entre parênteses, que representam respectivamente a gravidade atual (*current severity*), a maior gravidade detectada anteriormente (*alarm severity*) e o estado atual do alarme (*alarm status*). É sincronizada diretamente ao banco *Alarm Cfg & State RDB*, o que implica que uma mudança realizada é rapidamente detectada pelo servidor e pelos demais clientes.
- *Alarm area panel*: indicação gráfica do estado do sistema.

É possível, a partir de qualquer uma das *views* presentadas acima, acessar outros recursos, como, por exemplo, gráficos e valores atuais das variáveis desejadas. Para isso, basta apertar com o botão direito acima da *PV* e apertar em *Process variables*.

A implementação fornece, ainda, suporte para autenticação de usuários via *LDAP* ou *JAAS*. Caso seja a escolha, somente usuários autorizados podem alterar as configurações de alarmes ou reconhecê-los.

v. *Web reports*: é fornecido também um conteúdo *Web* capaz de gerar relatórios, calcular estatísticas (como, por exemplo, totais diários, variáveis que mais disparam alarmes, intervalos de tempo que os alarmes permanecem mais ativos em média *etc.*) e monitorar os alarmes. Assim como os módulos do *EPICS Archiver Appliance*, as páginas devem estar hospedadas em um servidor *Tomcat*.

3.2 Instalação

BEAST necessita de uma base de dados relacional, de uma implementação *JMS* e de um ambiente gráfico baseado no *Eclipse*. Utilizaremos, respectivamente, o *MySQL*, *Apache ActiveMQ* e a versão *Eclipse Luna for RCP and Plugin Development*.

Referências bibliográficas

Netwok Time Protocol - NTP

- shash