



Relatório de Estágio

CNPEM - Centro Nacional de Pesquisa em Energia e Materiais

LNLS - Laboratório Nacional de Luz Síncroton

Gustavo **CIOTTO PINTON**
Campinas, 23 de agosto de 2016

Sumário

1	Introdução	2
2	<i>Netwok Time Protocol</i> - NTPv4	2
2.1	Introdução	2
2.2	Características do Protocolo	2
2.3	Exemplo	4
2.4	Aplicação ao sistema de controle do <i>Sirius</i>	7
2.5	Construção de um <i>cape</i> para a <i>BeagleBone Black</i>	9
2.6	Resultados	10
3	Sincronização via <i>Ethernet</i> com a <i>BeagleBone Black</i>	11
3.1	Introdução	11
3.2	Implementação	12
3.3	Utilizando o módulo PRU	17
3.4	Resultados	17
3.5	Alternativa de solução: <i>Industrial Ethernet</i>	18
4	Manutenção e Implementação de Clientes PROSAC	19
4.1	Introdução	19
4.2	Manutenção do cliente <i>PROSAC</i> escrito em <i>Java</i>	19
4.3	Implementação para o <i>kit STM32F7 Discovery</i>	19
5	Servidores de variáveis EPICS	20
5.1	Introdução	20
5.2	Utilizando a biblioteca PCASpy	20
5.3	<i>EPICS StreamDevice</i>	21
6	EPICS Archiver Appliance	22
6.1	Introdução	22
6.2	Instalação	22
6.3	Uso do CS Studio no monitoramento	28
6.4	Acessando a <i>appliance</i> com <i>Python</i>	29
7	Best Ever Alarm System Toolkit - BEAST	30
7.1	Introdução	30
7.2	Instalação	32
7.3	Uso da interface <i>Eclipse</i> como <i>Alarm Client GUI</i>	36

1 Introdução

Durante o primeiro semestre de 2016, desenvolvi minhas atividades de estágio no Laboratório Nacional de Luz Síncrotron (LNLS - CNPEM) no grupo de Controle, que é responsável por prover soluções de controle e sensoramento de diversos equipamentos utilizados no acelerador de partículas, como fontes de tensão e corrente, bombas de vácuo e sensores de temperatura, por exemplo. É responsabilidade do grupo, igualmente, planejar e desenvolver as futuras ferramentas que serão utilizadas no acelerador *Sírius*, que se encontra em fase de construção atualmente.

Nestes 5 meses, entrei em contato com o desenvolvimento de *software* e *hardware* para sistemas embarcados, especialmente para Linux embarcado, já que o laboratório pretende utilizar a *BeagleBone Black* para futuras implementações. Entre os principais projetos, pude desenvolver módulos para o *kernel* do Linux, programas para o módulo *realtime* da Beagle e integrar um receptor GPS a esta mesma placa. As próximas seções são dedicadas aos detalhes de implementação destes e de outros problemas.

Cabe destacar que este relatório trata-se de uma versão ainda não completa, visto que algumas atividades deverão ser finalizadas e, portanto, documentadas somente no segundo semestre de 2016.

2 Network Time Protocol - NTPv4

2.1 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 providas 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*.

2.2 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 uma 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.

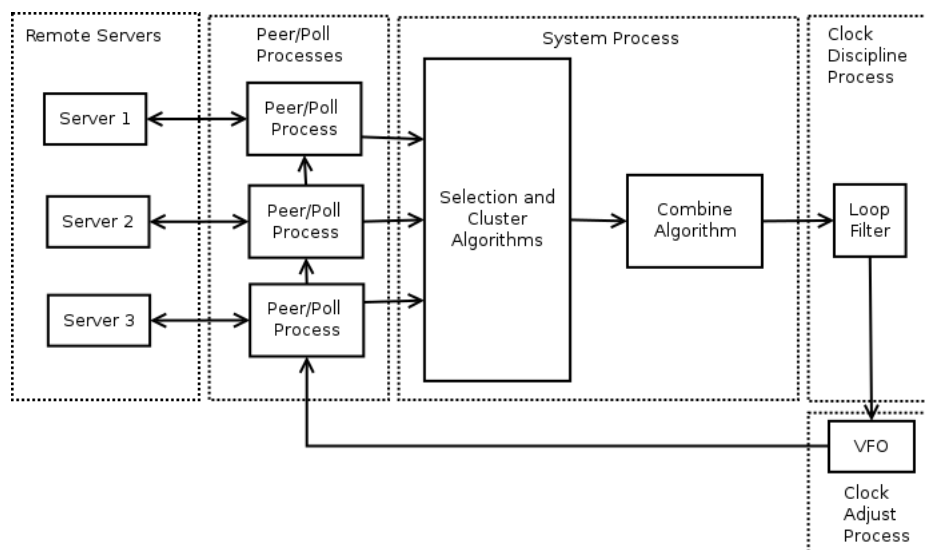


Figura 1: Implementação dos processos executados por um nó da rede.

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 estatísticas 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 clusterização.

- 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.

2.3 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.

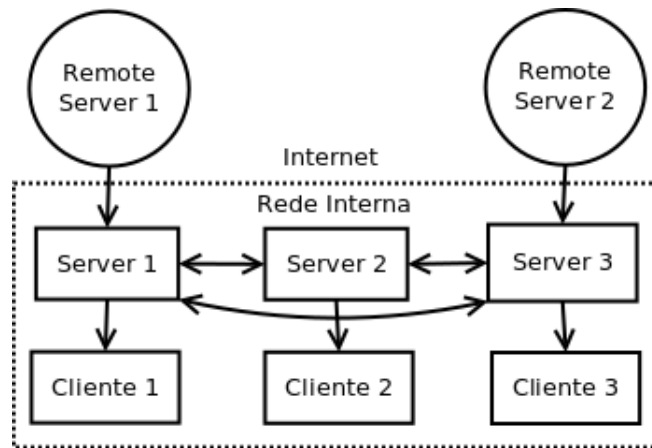


Figura 2: Topologia da rede de teste.

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 *** 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 *falseickers*). *refid* representa o *id* da referência 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 obtém 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*.

2.4 Aplicação ao sistema de controle do *Sirius*

Considerando que o sistema de controle do *Sirius* será composto por uma vasta quantidade de *Beagles* conectadas, é de extrema importância que a data e hora de todas elas estejam corretamente sincronizadas entre si, a fim de garantir a coerência entre os diversos *logs* que serão gerados na execução dos programas. Conforme discutido na subseção anterior, um servidor NTP é capaz de fornecer sincronismo a um número variável de clientes, desde que eles estejam configurados a obter sincronização deste servidor.

Diversos institutos fornecem, através da Internet, relógios de referência, isto é, servidores chamados de *stratum 0*, obtidos a partir de equipamentos como *GPS*, relógios atômicos ou outros *radio clocks*. Para a rede de controle do *Sirius*, que pretende ser totalmente isolada de redes externas, tal solução não pode ser obviamente utilizada. Sendo assim, propõe-se a implementação de um servidor NTP *stratum 0* a partir de um *GPS receiver*, ligado a uma *BeagleBone Black* dedicada, que hospedará o servidor.

GPS receivers são capazes de fornecer, além do posicionamento e velocidade, data e hora no padrão *UTC* e um pulso, chamado de *1PPS*, com frequência de 1Hz e precisão na ordem de $50ns$. Pode-se imaginar que somente o horário fornecido é suficiente para a implementação de um bom servidor NTP. Entretanto, os *delays* com que tal informação é recebida e transmitida pelo *receiver* não são constantes e variam conforme temperatura. Dessa forma, a fim de obter um tempo preciso, um pulso de *1PPS*, que marca o início de um novo segundo e cujo *jitter* pertence à ordem de centenas de nanosegundos, também é fornecido. A combinação destes dois últimos, aliados a um sistema operacional com *kernel* habilitado ao tratamento de tais pulsos, permite a implementação de um servidor com acurácia da ordem de $1\mu s$ em relação ao horário *real*. É importante destacar que o *PPS* deve ser combinado com alguma outra fonte de tempo, uma vez que ele não é capaz de dizer o horário efetivo, isto é, horas, minutos e segundos, mas sim somente o início de um novo segundo.

As próximas subseções visam descrever o *hardware* utilizado e a configuração do servidor NTP.

2.4.1 Descrição do *hardware*

Dois módulos GPS de fabricantes distintos foram comprados:

- i. *Adafruit Ultimate GPS Breakout*.
- ii. *MikroE GPS Click - u-blox LEA6S*

Ambos fornecem um pino de saída *1PPS* e comunicação serial assíncrona *RX/TX (UART)*. Foi utilizado também o multivibrador mono estável *74HC123*, a fim de obtermos um pulso *1PPS* mais largo na entrada do pino da *BeagleBone Black*. Tal pulso é fornecido por alguns receptores com uma largura de apenas $10\mu s$, que pode não ser capturado pela placa, dependendo da utilização da sua CPU. Sendo assim, utilizando-se um resistor de $1M\Omega$ e um capacitor de $1\mu F$, obtém-se uma largura de aproximadamente $500ms$. Para os receptores listados acima, porém, o uso do multivibrador não é necessário, visto que provêm pulsos de largura superiores a $100ms$.

2.4.2 Configuração do servidor NTP

O primeiro passo para a implementação é a preparação do *kernel* do *Linux* para o tratamento de pulsos *1PPS*. Isso pode ser realizado pela compilação do *kernel* (ver subseção 3.2.6) habilitando esta opção ou a partir da instalação do pacote *pps-tools* a partir do comando `apt-get install`.

A segunda etapa consiste na configuração das portas de entrada e saída que serão usadas pela *BeagleBone Black* para se comunicar com o GPS. Conforme referência [1], adotaremos que os pinos P9-11 e P9-13 serão utilizados, respectivamente, como *TX* e *RX*, e P9-12, como entrada do pulso

1PPS. O arquivo de *device tree overlay* especificando tais comportamentos pode ser obtido a partir de <http://tinyurl.com/oeo2ccu>. Após compilado e carregado, duas novas interfaces estarão disponíveis, sendo elas `/dev/ttyO4` e `/dev/pps0`. Para habilitar o carregamento deste *overlay* após o *boot*, é necessário editar os arquivos `/boot/uEnv.txt` e `/etc/default/capemgr` com, respectivamente, `cape_enable=capemgr.enable_portno=overlay_GPS` e `CAPE=overlay_GPS`.

A comunicação com o *GPS receiver* é realizado por meio do *daemon GPSD*. Após instalá-lo, crie o arquivo `/lib/systemd/system/gpsd.service` e adicione as seguintes linhas:

```
[Unit]
Description=GPS (Global Positioning System) Daemon
Requires=gpsd.socket

[Service]
ExecStart=/usr/sbin/gpsd -n -N /dev/ttyO4

[Install]
Also=gpsd.socket
```

O *daemon* pode ser iniciado com

```
systemctl start gpsd.service
```

O programa `gpsmon`, provido no pacote `gpsd-clients`, permite verificar se a conexão entre a *BeagleBone* o módulo GPS está correta.

A configuração do NTP é ligeiramente mais complicada, visto que é necessário recompilá-lo de maneira a habilitar o uso do *driver* responsável por manipular o pulso 1PPS. Para tal, faça o *download* do código fonte em <http://www.ntp.org/downloads.html> e extraia seu conteúdo para o diretório de preferência. Em seguida, entre neste diretório e execute o comando abaixo. As *flags* `--enable-ATOM` e `--prefix` especificam, respectivamente, a ativação do *driver* de gerenciamento do 1PPS e o diretório onde o NTP será instalado.

```
./configure --enable-ATOM --prefix=/usr/local/ --enable-linuxcaps
```

O comando `make install` instalará os binários no diretório especificado. O pacote `libcap-dev` é necessário e, portanto, deve ser instalado via `apt-get install`. Enfim, a última etapa consiste em atualizar os arquivos `ntpd.service` e `ntpdate.service` para que os serviços sejam inicializados assim que a *BeagleBone* for ligada. Para tal, implementei um *script bash*, encontrado no repositório do grupo, que realiza automaticamente esta tarefa.

2.4.3 Implementação de variáveis EPICS

A fim de tornarmos disponíveis as variáveis EPICS que possam refletir o estado dos receptores e do servidor NTP, utilizamos algumas bibliotecas em *Python*. O servidor de variáveis é implementado através do módulo *PCASpy* (consultar seção 5) e de suas classes, e a comunicação com o servidor e o *receiver* através das bibliotecas `python-gps` (`apt-get install python-gps`) e `ntplib`.

Para o servidor NTP, as seguintes variáveis estão disponíveis:

- `NTP:OnOff`: indica se o servidor está ativo.
- `NTP:Address`: endereço IP do servidor na rede.
- `NTP:Timestamp`: número de segundos entre a data do servidor e 01 Janeiro de 1970.
- `NTP:Day`, `NTP:Month`, `NTP:Year`, `NTP:Hour`, `NTP:Minute` e `NTP:Second`: conversão de `NTP:Timestamp` nos respectivos campos.

- *NTP:Stratum*: *stratum* do servidor.
- *NTP:Leap*: ocasionalmente, as autoridades adicionam ou retiram 1 segundo do horário UTC, com o intuito de torná-lo mais próximo do horário solar. Esta variável pode assumir 4 estados, de acordo com a RFC5905. Quando vale 0, significa que nenhuma mudança é necessária e quando vale 1 ou 2, um segundo é, respectivamente, adicionado ou retirado. Ela pode conter também o valor 3, que representa que o relógio do servidor está dessincronizado.
- *NTP:Version*: versão no NTP. Por padrão, sempre utilizamos a 4.
- *NTP:Roundtrip*: tempo que o pacote de requisição leva para chegar e retornar da fonte, em milissegundos.
- *NTP:Reference*: *string* de 4 caracteres que identifica a fonte de sincronismo atualmente utilizado pelo servidor.

Para os receptores GPS, por sua vez, são disponibilizadas:

- *GPS:Fix*: *enum* que indica o tipo de *fix* obtido pelo GPS. 0 indica que o receptor não pôde obter nenhum *fix*, 1 que um *2D fix* foi obtido e 2, um *3D fix*.
- *GPS:Latitude*, *GPS:Longitude* e *GPS:Altitude*: informações de posicionamento.
- *GPS:UTC:Timestamp*: horário UTC em formato *timestamp* retornado pelo GPS. Da mesma maneira, são disponibilizadas variáveis individuais contendo campos, tais como dia, mês e ano, obtidos a partir deste *timestamp*.
- *GPS:UTC:Satellites*: vetor com os *PRN* dos satélites utilizados no cálculo do *fix*.

A interface abaixo foi gerada a partir destas bibliotecas e utilizam as *PVs* descritas acima:


Host address:	10.0.6.60	
NTP SERVER SUMMARY		
Date:	29 / July / 2016 11 : 01 : 21	
Leap Indicator:	No warning	
Round-trip time:	0.000 ms	
Stratum:	1	
Reference ID:	PPS	
GPS RECEIVER SUMMARY		
Fix:	3D FIX	
Latitude:	22o 48 ' 17.881 " S	
Longitude:	47o 3 ' 16.577 " W	
Altitude:	620.50 m	
UTC:	29 / July / 2016 14 : 01 : 20	
Satellites in use:	1 3 8 9 11 14 16 22 23 26 31	

Figura 5: Interface construída para visualização das *PVs*.

2.5 Construção de um *cape* para a *BeagleBone Black*

Dois *cares* foram implementados através do *Kicad*, um para cada receptor GPS. Tais arquivos podem ser encontrados no repositório do grupo.

2.6 Resultados

A figura 6 representa os resultados obtidos para o receptor *Adafruit*. Tal tabela foi obtida através do comando `ntpq`, sendo que as colunas representam informações relativas às fontes de sincronismo que tal servidor utiliza, tais como *stratum*, *delay*, *offset* e *jitter*, cujos conceitos já foram explorados anteriormente.

Destaca-se a presença de duas fontes específicas, SHM e PPS. A primeira, cuja representação é a abreviação de *SHared Memory*, obtém seus valores de hora e data por meio de um segmento compartilhado de memória, acessado igualmente pelo utilitário `gpsd`. Em outras palavras, tal segmento é o canal de comunicação utilizado pelo servidor NTP para comunicar-se com o `gpsd`, que, por sua vez, implementa a troca de dados com o respectivo receptor. O segundo trata-se da fonte ligada ao PPS, controlada pelo *driver* ATOM que tivemos que incorporar na compilação do *NTP*. A fonte representada por LOCAL, que obtém a data do próprio sistema, é considerada apenas se as duas anteriores não estiverem disponíveis.

remote	refid	st	t	when	poll	reach	delay	offset	jitter
LOCAL(0)	.LOCL.	10	l	204m	64	0	0.000	0.000	0.000
*SHM(0)	.GPS.	0	l	4	8	377	0.000	-22.871	39.990
oPPS(0)	.PPS.	0	l	3	8	377	0.000	-0.006	0.006

Figura 6: Resultado do comando `ntpq -p` na *BeagleBone* conectada ao *cape* com o receptor *Adafruit Ultimate GPS*.

Conforme esperado, o servidor conseguiu sincronizar-se ao início de um novo segundo (PPS) com precisão e *offset* na ordem de unidades de microsegundos. Na tabela representada acima, por exemplo, o *offset* é $-0.006ms$, isto é, o horário do servidor está $6\mu s$ atrás do pulso de PPS, e o jitter, $0.006ms$.

A figura 7, por sua vez, exibe o resultado da execução do mesmo comando, porém para o receptor da *MikroE*. Observa-se que, como no caso anterior, o servidor conseguiu sincronizar-se ao pulso de PPS.

remote	refid	st	t	when	poll	reach	delay	offset	jitter
LOCAL(0)	.LOCL.	10	l	107m	64	0	0.000	0.000	0.000
*SHM(0)	.GPS.	0	l	2	8	377	0.000	-137.48	2.448
oPPS(0)	.PPS.	0	l	1	8	377	0.000	0.000	0.002

Figura 7: Resultado do comando `ntpq -p` na *BeagleBone* conectada ao *cape* com o receptor *MikroE GPS Click*.

Enfim, a figura 8 representa a saída do comando executado em uma *Beagle* que utiliza 3 servidores para obter o horário. Os dois primeiros (10.0.6.63 e 10.0.6.60) são os endereços das placas ligadas, respectivamente, aos receptores da *Adafruit* e da *MikroE*, e o último, enfim, é um servidor externo, referenciado pelo endereço `ntp.cnpem.br`.

remote	refid	st	t	when	poll	reach	delay	offset	jitter
*10.0.6.63	.PPS.	1	u	2	8	377	0.216	0.002	0.040
+10.0.6.60	.PPS.	1	u	4	8	377	0.206	0.011	0.011
LOCAL(0)	.LOCL.	10	l	100m	64	0	0.000	0.000	0.000
+a.st1.ntp.br	.ONBR.	1	u	5	8	377	3.109	-0.154	0.013

Figura 8: Resultado do comando `ntpq -p` na *BeagleBone* cliente que obtém sincronização de 3 servidores distintos.

Nota-se que a diferença entre as fontes PPS está na ordem de unidades de microssegundos e que a diferença entre elas e a externa é da ordem de $100\mu s$. Isso indica que o fato de que a externa pertence a uma rede externa ao CNPEM, há a introdução de um atraso no horário fornecido por ela.

3 Sincronização via *Ethernet* com a *BeagleBone Black*

3.1 Introdução

A sincronização dos diversos componentes presentes no sistema de controle de um acelerador de partículas é um fator fundamental para seu bom funcionamento. Os sistemas que necessitam sincronismo, como as fontes de corrente para os ímãs, controladores de feixe e de injeção, por exemplo, devem exercer as suas respectivas funções em momentos especificados por pulsos de sincronismo, que podem ser enviados, por exemplo, por geradores de sinais espalhados pela infraestrutura local. A precisão com que estes equipamentos recebem tais pulsos depende de diversas variáveis como, por exemplo, o tipo de canal de comunicação utilizado no envio do pulso e a interface de entrada de dados que eles possuem. A precisão pode ser obtida através da medição do *jitter*, ou desvio-padrão, do *delay* calculado entre o envio do pulso e a sua recepção no equipamento. O *Sirius*, por exemplo, possui especificações para diversos sistemas de sincronismo, que variam de acordo com a necessidade de precisão dos equipamentos e das funções desempenhadas por eles. Para casos mais graves, como o da bomba de elétrons do acelerador linear (*LINAC*), os *jitters* não podem ultrapassar os $50ps$ e soluções especiais devem ser exploradas, como a implementação de uma rede *WhiteRabbit*, que está sendo desenvolvida e estudada por outros laboratórios no mundo todo.

Um sistema de sincronismo para as fontes de corrente já foi implementado pelo grupo de controle, utilizando as unidades *Programmable Realtime Unit*, ou simplesmente *PRU*, presentes na *BeagleBone Board*. Tais unidades apresentam *clock* de 200MHz, núcleos de memória própria e compartilhada, e módulos dedicados, como o *Enhanced GPIO*, que favorecem a implementação de aplicações *realtime*, em que a precisão é uma necessidade importante. Soluções implementadas para a *PRU* não estão sujeitas a fatores que degradam a precisão como o compartilhamento de *CPU* com outros processos e preempção. Aplicações que rodam sobre *Linux* embarcado e que, portanto, compartilham recursos com outros processos, apresentam, geralmente, desempenho inferior. O sistema desenvolvido utiliza a *PRU* para a recepção do sinal de sincronismo e ativa, posteriormente, seus nós escravos, conforme figura 9 abaixo. Esse sistema obteve um *jitter* da ordem de 15ns, representando, assim, um valor próximo do ótimo, uma vez que os ciclos de processamento de cada um dos componentes estão próximos de 5ns.

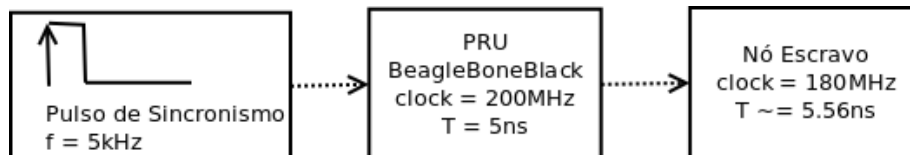


Figura 9: Esquema de sincronismo realizado pelo grupo de controle.

Nesta seção, será apresentada a alternativa descrita na figura 10, que utiliza o protocolo *Ethernet* e equipamentos padrões na implementação deste tipo de rede, como *switches*, para o envio dos *triggers* de sincronismo, e a sua respectiva *performance*.

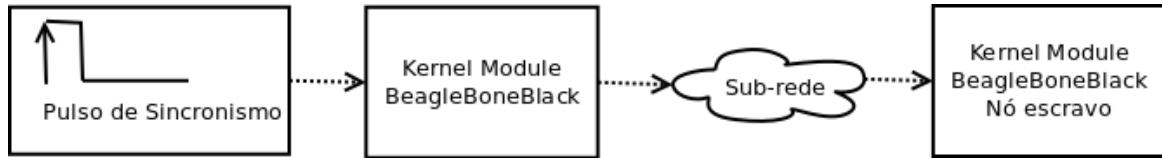


Figura 10: Sistema de sincronismo proposto.

3.2 Implementação

Esta subseção é dedicada à implementação do sistema de sincronismo proposto.

3.2.1 Loadable Kernel Modules

A principal diferença entre os dois sistemas apresentados na subseção anterior é que, no segundo, não utilizamos a unidade *PRU* da *BeagleBone Black*. Em seu lugar, implementamos módulos do *kernel* do *Linux*, os chamados *Loadable Kernel Modules (LKM)*. Tais módulos são mecanismos que permitem a adição ou remoção de código do *Linux kernel* em tempo de execução e, por esta razão, são ideias para a implementação de *device drivers*, cujo principal propósito é realizar a comunicação com o *hardware* disponível.

Sendo essencialmente parte do *kernel*, os *LKMs* são executados no *kernel space* e, por isso, possuem endereços de memória e *APIs* próprios, que são, por sua vez, separados daqueles disponíveis no *user space*. Este último representa o *space* em que, na maioria das vezes, escrevemos e executamos as aplicações em C. A figura 11 resume as interações entre os dois espaços: o *user space* comunica-se com *kernel space* através de *system calls*, que, por sua vez, acessa o *hardware* através de *drivers* específicos. O *kernel space* possibilita o tratamento de interrupções, que serão utilizadas neste exemplo, ao contrário do *user space*.

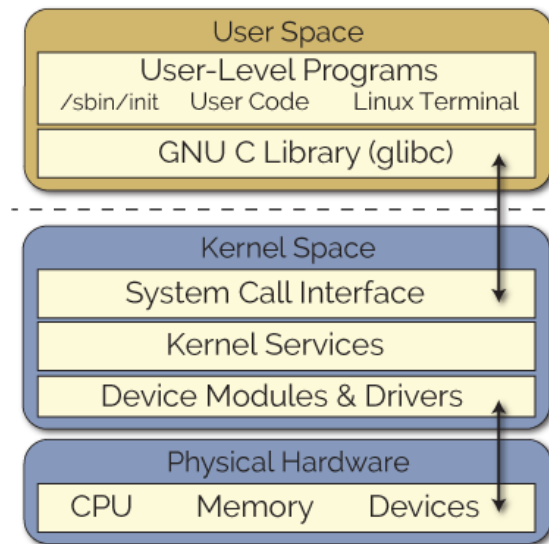


Figura 11: Comunicação entre *kernel* e *user spaces*.

3.2.2 Descrição das características do projeto

De maneira geral, serão desenvolvidas duas aplicações, uma para a *Beagle BB1* que recebe o *trigger* de sincronismo e prepara o pacote *UDP* para envio e a outra, para a placa *BB2*, que é responsável pela recepção do respectivo pacote e tratamento. A aplicação que recebe o *trigger* mantém um contador interno e o transmite em intervalos definidos de tempo, para que o nó escravo possa também manter um contador de mesmo tipo. Dessa forma, o nó escravo pode determinar se um pacote foi perdido durante a transmissão e corrigir seu contador.

Em termos de programação, as aplicações apresentam as seguintes características:

- *Kernel Module* em *BB1*: um *timer* gera interrupções a cada 1ms. Na rotina de tratamento desta interrupção, um registro contendo os dados a serem enviados via *Ethernet* é adicionado em um *buffer* circular. Uma outra *thread* é responsável por retirar estes elementos do *buffer*, preparar os pacotes *UDP* e enviá-los.
- *Kernel Module* em *BB2*: uma *thread* é instanciada e espera pacotes *UDP*. Quando recebe, atualiza seu contador e seus pinos de saída.

Por questões de simplicidade, o módulo *PWM* da placa que envia os pacotes *UDP* (*BB1*) será utilizado como gerador dos *triggers* de sincronismo. A sua configuração será descrita na próxima subseção.

3.2.3 Configuração do *PWM* com o *Device Tree Overlays*

Os mecanismos de *Cape Manager* e *Device Tree Overlays* permitem a modificação do sistema, como, por exemplo, configuração da função dos pinos, ativação e carregamento de módulos em tempo de execução a partir de programas sendo executados no *user space*.

Sendo assim, a fim de modificarmos o pino P9_22 (*header P9*, pino 22) para que atue como uma saída do módulo *PWM*, é necessário escrevermos um arquivo de extensão *.dts* (*device tree source*), contendo as especificações que desejamos adotar. Um arquivo deste tipo é disponível em <http://tinyurl.com/ntkl2w7> e contem todas as possíveis configurações de todos os pinos da *BeagleBone Black*. Como vamos utilizar somente as linhas referentes ao P9_22, podemos copiá-las para um arquivo separado e adotá-lo em seguida. Após obtermos tal arquivo, é necessário compilá-lo a partir do comando *dtc* e incluir o resultado no diretório */lib/firmware/*.

Para carregar o *overlay*, configuramos duas variáveis de ambiente:

```
$ export SLOTS=/sys/devices/bone_capemgr.9/slots
$ export PINS=/sys/kernel/debug/pinctrl/44e10800.pinmux/pins
```

Executamos, então, os comandos abaixo, em que *pwm_P9_22* referencia o arquivo gerado após compilação. *duty_ns* e *period_ns* são utilizados, respectivamente, para configurar o intervalo de tempo em que a onda permanecerá em nível lógico alto e o seu período total. Para as próximas seções, vamos adotá-los como, respectivamente, *2ms* e *70ms*. Um *script bash*, que pode ser encontrado no repositório do grupo, foi escrito e automatiza o processo abaixo.

```
$ echo pwm_P9_22 > $SLOTS
$ echo pwm > /sys/devices/ocp.3/P9_22_pinmux.12/state
$ cd /sys/class/pwm/
$ echo 0 > export
$ cd pwm0/
$ echo 2000000 > duty_ns
$ echo 70000000 > period_ns
```

3.2.4 Implementação do *Kernel Module* de **BB1**

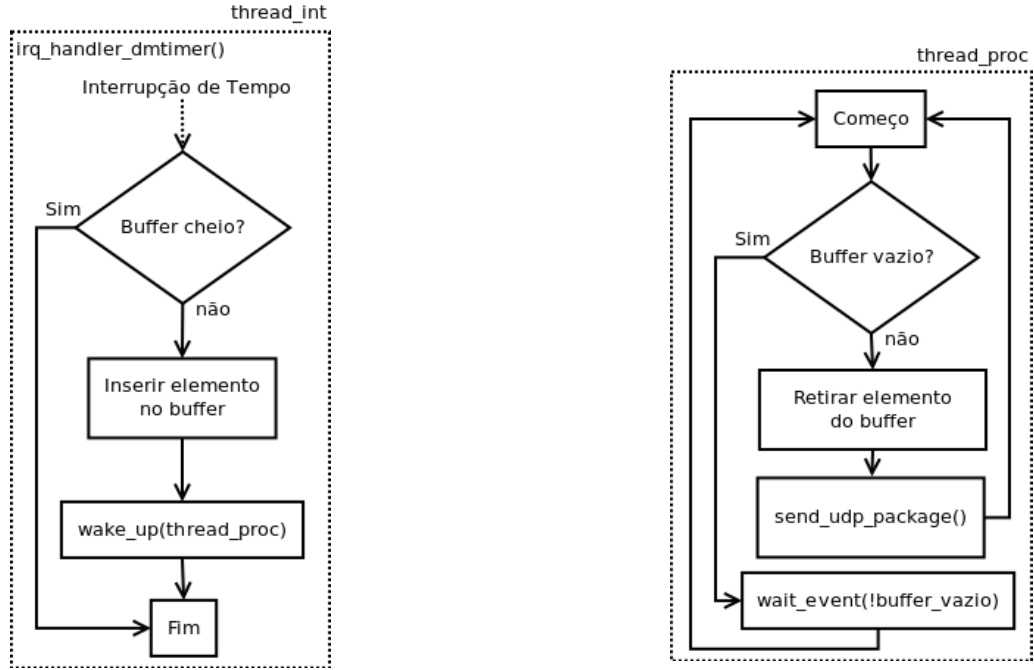
O *module* desenvolvido para a placa *BB1* instancia duas *threads*: uma para inicialização dos componentes e a outra para processamento e envio de pacotes *UDP*. No *kernel space*, cada *thread* é dada por uma estrutura do tipo *struct task_struct*, especificada na biblioteca *linux/kthread.h*, que provê também as funções *kthread_create* e *wake_up_process*. A primeira é responsável por alocar os recursos necessários à execução de uma *thread* e a segunda, por permitir a sua execução. É possível alterar a prioridade e a *policy* de uma *thread* através da função *sched_setscheduler*,

disponível em `linux/sched.h`. Esses dois parâmetros são usados pelo *Linux scheduler* para decidir qual dos processos interrompidos deterá o processador após a interrupção ou bloqueio do processo em curso de execução. O *Linux* fornece duas políticas de *real-time*, `SCHED_FIFO` e `SCHED_RR`, que possuem prioridade superior à política comum, `SCHED_NORMAL`. Portanto, processos cujas *scheduler classes* são do tipo *real-time* sempre são escolhidos antes daqueles de política comum. É necessário observar que, no *kernel* padrão do *Linux*, estas classes não são capazes de garantir um comportamento dito *hard real-time*, isto é, que assegurem o cumprimento de qualquer requisito dentro de um certo limite. As *real-time scheduling policies* fornecem, portanto, um comportamento de *soft real-time*, ou seja, o *kernel* fará o máximo possível para atender às requisições, mas não pode prometer sempre cumpri-las. Porém, um *patch*, chamado `RT_PREEMPT` e disponível em <http://tinyurl.com/2fxb3sd>, foi desenvolvido para garantir características de *hard real-time*. Sua instalação está comentada nas próximas subseções. Quando uma tarefa do tipo `SCHED_FIFO` assume a *CPU*, ela continua a ser executada até que ela seja bloqueada ou que ela conceda explicitamente o processador a outro processo, podendo rodar indefinidamente caso nenhuma dessas condições seja atendida. Somente uma tarefa `SCHED_FIFO` ou `SCHED_RR` de maior prioridade pode interromper sua execução. Para as tarefas do tipo `SCHED_RR`, o princípio é o mesmo, porém elas são submetidas a um intervalo de execução, sendo que, ao fim deste intervalo, tais tarefas perdem os recursos de processamento e entram na fila circular de espera. Em outras palavras, a política `SCHED_RR` é semelhante à `SCHED_FIFO`, exceto no tempo que é permitido ao uso de *CPU*. No nosso caso, ambas as *threads* terão políticas de `SCHED_FIFO`, mas aquela que é interrompida pelo *timer* e *GPIOs* possuirá maior prioridade. Em adição, a segunda *thread* (aquela responsável pelo envio de pacotes) desiste da *CPU* quando o *buffer* circular está vazio e só é “acordada” quando uma interrupção de tempo ocorrer (o *buffer* volta a ser preenchido na função de tratamento desta interrupção). Os fluxogramas da figura 12 abaixo resumem o funcionamento destas *threads*.

A figura 12a especifica o fluxograma da *thread_int* que preenche o *buffer* circular a cada interrupção do relógio e sinaliza para a *thread_proc* que ele não está mais vazio. A figura 12b evidencia que um pacote é enviado somente o *buffer* não estiver vazio. Caso contrário, a *thread_proc* se bloqueia esperando uma interrupção do relógio. As funções `wake_up` e `wait_event` estão disponíveis na biblioteca `linux/wait.h`. A segunda deve receber como parâmetro uma fila, que representa a estrutura de dados onde serão colocados os processos que esperam pelo evento. Essa fila é do tipo `wake_queue_head_t` e pode ser criada estaticamente, via a *macro* `DECLARE_WAITQUEUE()`, ou dinamicamente, via a função `init_waitqueue_head()`.

Os processadores da família *Sitara AM335x* da *Texas Instruments* possuem um módulo, chamado de *DMTimer*, composto por 7 *timers* configuráveis. Cada um dos *timers* contem um contador de 32 *bits* crescente com capacidade de *auto reload* e geração de interrupção ao atingir seu valor máximo (`0xFFFFFFFF`), isto é, em caso de *overflow*. Além disso, é possível configurar um divisor de frequência (*prescaler*) e escolher o sinal de *clock* que será usado pelo componente. No *kernel space*, a biblioteca `plat/dmtimer.h` fornece as principais operações para configuração do módulo.

A função `omap_dm_timer_request()` retorna uma estrutura do tipo `struct omap_dm_timer` que representa um dos *timers* disponíveis no módulo. Há mais algumas variantes desta chamada, porém a principal é a `omap_dm_timer_request_specific(int timer_id)`, que recebe como parâmetro o *id* do *dmtimer* desejado. É necessário notar, porém, que talvez seja necessário a configuração de um *overlay* para a configuração dos pinos utilizados pelo respectivo *timer*. Em seguida, executamos a chamada `omap_dm_timer_set_source`, que configura o sinal de *clock* de entrada. Utilizamos o *clock* de 32kHz, portanto passamos a constante `OMAP_TIMER_SRC_32_KHZ` como parâmetro à função. `omap_dm_timer_set_prescaler` permite definir o valor do divisor de frequência e `omap_dm_timer_set_load_start`, o valor que será carregado no contador quando um *overflow* ocorrer. Tais definições dependem do intervalo desejado entre interrupções e são dadas pela equação 1, em que *TDLR* é o valor a ser carregado no contador, *PTV*, o *prescaler* e $f_{clock} = 32kHz$.



(a) Fluxograma da *thread* que é interrompida pelo *timer*. (b) Fluxograma da *thread* que envia pacotes à rede.

Figura 12: Fluxogramas das *threads* que compõem BB1.

$$t = (0xFFFFFFFF - TDLR + 1) * \frac{1}{f_{clock}} * 2^{PTV+1} \quad (1)$$

No nosso caso, desejamos interrupções a cada $t = 1ms$ com $PTV = 1$. A aplicação direta da equação 1 resulta em $TDLR = 0xFFFFFFFF7$.

O último passo é ativar a interrupção em caso de *overflow* e associar a ela uma função de tratamento. Para configurá-la, utilizamos três funções:

- `omap_dm_timer_get_irq()`: retorna o *id* da interrupção associada ao *timer* passado como parâmetro;
- `omap_dm_timer_set_int_enable()`: habilita interrupção no módulo. Recebe como parâmetro uma constante que define o evento que deve ser associado à interrupção. No nosso caso, como queremos que uma interrupção seja lançada após *overflow* do contador, passamos o valor `OMAP_TIMER_INT_OVERFLOW`;
- `request_irq()`, da biblioteca `linux/interrupt.h`: recebe como parâmetro o *id* da interrupção, a sua função tratadora e o tipo de função. Este último é uma constante definida na biblioteca e vale `IRQF_TIMER`. No nosso caso, criamos uma função chamada `dmtimer_irq_handler()` que retorna uma estrutura do tipo `irqreturn_t`. Para comunicar ao sistema operacional que a interrupção foi corretamente tratada, tal função deve retornar `IRQ_HANDLED`. Além desta constante, a função deve atualizar o registrador de *status* do módulo realizando uma escrita. Isso é feito através da função `omap_dm_timer_write_status()`, com o parâmetro `OMAP_TIMER_INT_OVERFLOW`.

Enfim, iniciamos o *timer* com a chamada de `omap_dm_timer_start()`.

A próxima etapa é a configuração dos pinos de entrada e saída. A biblioteca `linux/gpio.h` fornece as funções necessárias para definirmos a direção (entrada ou saída) e o nível lógico a ser atribuído a um respectivo pino. Utilizamos dois pinos de entrada e saída nesta aplicação:

- GPIO_48 (P9_15): pino de entrada que recebe o *trigger* de sincronismo. A cada borda de subida detectada, uma interrupção é lançada e algumas *flags* responsáveis por armazenar a ocorrência do evento são atualizadas. Quando a *thread_proc* prepara o próximo pacote, ela consulta essas *flags* e, caso esteja setadas, adiciona ao respectivo pacote o *trigger* recebido;
- GPIO_68 (P8_15): pino de saída que tem seu valor alterado quando um *trigger* de sincronismo foi detectado. Utilizado para testes da aplicação.

Para verificar se um pino pode ser utilizado, chamamos a função `gpio_is_valid`, cujo parâmetro é o *id* do respectivo pino. Se o retorno for diferente de 0, o *id* é válido e podemos continuar a configurá-lo. Em seguida, executamos, em sequência, `gpio_request()`, `gpio_direction_output()` - para saída - ou `gpio_direction_input()` - para entrada - e `gpio_export()`. Tais funções são responsáveis por, respectivamente, reservar o pino, configurar sua direção e exportá-lo, permitindo ou não que outras aplicações o utilizem. O pino de entrada requer adicionalmente que uma interrupção seja relacionada a ele. Para tal, repetimos o procedimento realizado para o *timer*, isto é, obtemos o *id* da interrupção através de `gpio_to_irq()` e associamos uma função tratadora com `request_irq()`. Entretanto, ao invés de enviarmos o parâmetro `IRQF_TIMER` para esta última, passamos `IRQF_TRIGGER_RISING`, que lançará as interrupções quando bordas de subida forem detectadas. Em relação aos pinos de saída, utiliza-se `gpio_set_value()` para modificar o valor de tensão imposta na respectiva saída, cujo *id* é passado como parâmetro à função. Os recursos utilizados por um pino de entrada ou saída são liberados a partir das chamadas `gpio_unexport()` e `gpio_free()`.

Enfim, o último aspecto a ser discutido neste módulo é a implementação do envio de pacotes *UDP*. Três bibliotecas devem ser importadas a fim de realizar essa comunicação: `linux/netdevice.h`, `linux/ip.h` e `linux/in.h`. O primeiro passo é a criação do *socket* de comunicações, que é realizado pela chamada à função `sock_create()`, passando como parâmetros algumas constantes e a estrutura `struct socket` que representa um *socket* no *kernel space*. As constantes especificam o tipo do respectivo *socket*, sendo que, para aplicações *UDP*, utilizamos `AF_INET`, `SOCK_DGRAM` e `IPPROTO_UDP`, que especificam, respectivamente, o formato dos endereços (endereços *Internet Protocol v4*), as camadas de transporte e rede dos pacotes a serem enviados. Em seguida, definimos seu endereço e a porta a partir da função `connect()`, que pode ser acessada através do atributo `ops` de `struct socket`. Além do *socket* criado, ela recebe como parâmetro um ponteiro para uma variável do tipo `struct sockaddr`, que, por sua vez, possui três atributos importantes: `sin_family`, `sin_addr.s_addr` e `sin_port`. Recebem, respectivamente, `AF_INET`, `htonl(INADDR_SEND)` e `htons(CONNECT_PORT)`, em que `htonl()` e `htons()` são funções pré-definidas que convertem valores de *host order* para *network byte order*, `INADDR_SEND` é o endereço, por exemplo `0xc0a80216` para `192.168.2.22`, para o qual deseja-se enviar o datagrama e `CONNECT_PORT` é a porta. `connect()`, além de definir tais características, também ativa a conexão no *socket*. O envio de pacotes é realizado por `sock_sendmsg()`, que recebe dois parâmetros: as estruturas `struct socket` e `struct msghdr`, que contem o conteúdo da mensagem. Enfim, para desalocar o *socket*, chama-se a função `sock_release()`.

Por fim, a compilação do arquivo `.c` é realizada por meio de um *Makefile*, cujo conteúdo é semelhante ao código abaixo, em que `timer_kernel_module` é o nome do arquivo do módulo e `$(shell uname -r)` retorna a versão dos *linux headers* instalados na *Beagle*.

```
obj-m+=timer_kernel_module.o

all:
    make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) clean
```

A adição do módulo compilado (arquivo `.ko`) é realizado através do comando `insmod` e a sua remoção, a partir de `rmmmod`. O arquivo de *log* pode ser utilizado para eventual *debug*, sendo acessado a partir de `/var/log/kern.log`.

3.2.5 Implementação do *Kernel Module* de BB2

Uma só *thread* é instanciada no *kernel module* de BB2. Ela aguarda a recepção de pacotes *UDP* e os processa. Ela é criada da mesma forma descrita na subseção anterior e tem sua *policy* modificada para `SCHED_FIFO`, a fim de obter tratamento de *real-time*. O fluxograma da figura 13 destaca a execução do módulo. Quando um pacote é recebido, há uma verificação de sua integridade e, caso seja válido, o contador e as portas de *gpio* são atualizadas conforme conteúdo. Observa-se que o processo de inicialização de tais portas segue o mesmo princípio que o descrito em **Implementação do *Kernel Module* de BB1**. A criação do *socket*, no entanto, é ligeiramente diferente: ao invés da função `connect()`, utiliza-se `bind()`, que atribui um endereço e porta ao respectivo *socket*, passado como parâmetro através de uma estrutura do tipo `struct sockaddr`, que é inicializada com as funções `htonl` e `htons`, destacadas anteriormente. A recepção dos pacotes é feita via `sock_recvmsg()`, que bloqueia a *thread* até que um datagrama seja detectado.

A compilação pode ser feito através de um *Makefile* e a adição e remoção do módulo, via os comandos `insmod` e `rmmmod`.

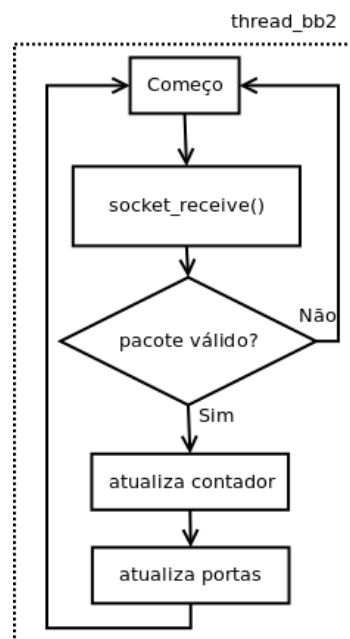


Figura 13: Fluxograma da *thread* do *kernel module* em BB2.

3.2.6 Modificando o *Linux* para *hard real-time*

Seção a ser completada no decorrer das atividades.

3.3 Utilizando o módulo PRU

Seção a ser completada no decorrer das atividades.

3.4 Resultados

Para avaliar os resultados, conectamos o *PWM* ao canal 3 do osciloscópio e configuramos o *trigger* para detectar subidas de borda deste canal. O módulo de BB2 foi configurado para modificar o estado

de um pino de saída, ao qual conectamos ao canal 2, a fim de gerar um simples pulso quando um pacote contendo um *trigger* de sincronismo for recebido. Dessa forma, podemos avaliar o *delay* entre a criação do evento inicial e de sua recepção no nó escravo, assim como o *jitter* associado. Para tal, utilizamos a ferramenta *Histograma* disponibilizada no equipamento e a configuramos para adquirir medidas relacionadas ao canal 2, isto é, pulsos gerados pela *Beagle BB2*. Modificamos a opção *Tempo de Persistência* do *Display Forma de onda* para infinito, de forma a garantir que a tela do osciloscópio mantenha os pulsos gerados em instantes passados. Enfim, habilitamos o funcionamento do osciloscópio, obtendo o resultado presente na figura 14.

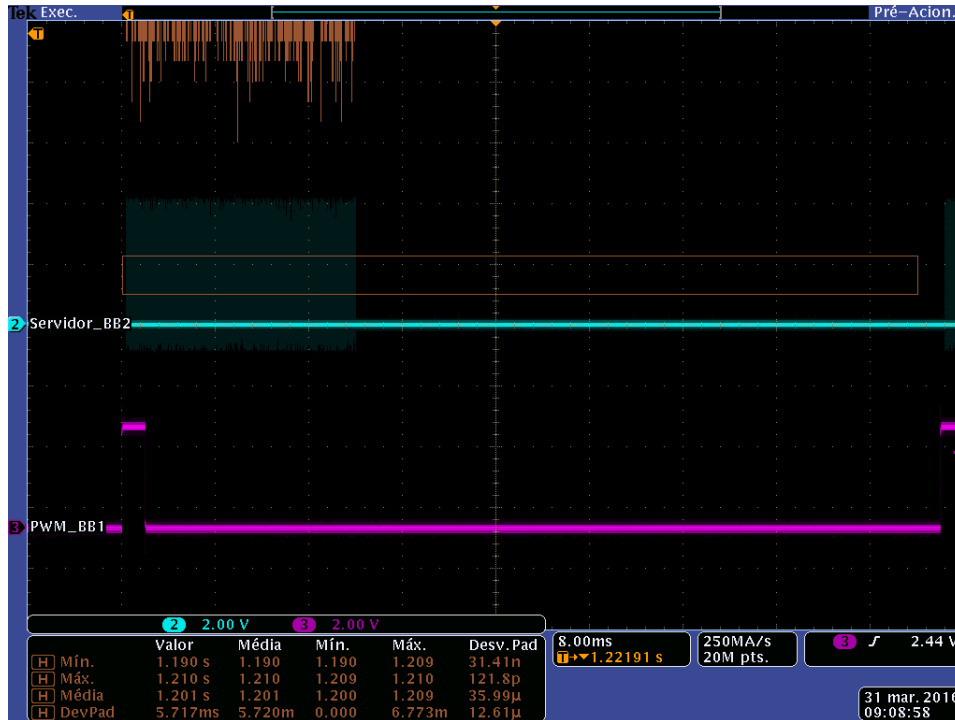


Figura 14: Captura de tela do osciloscópio.

O desvio padrão, ou *jitter* obtido, é da ordem de $5.7ms$, muito distante daquele adquirido pelo sistema construído pelo grupo de controle, no qual utilizou-se a *PRU* e obteve-se um valor próximo de $15ns$. Tal diferença é explicada, basicamente, por dois fatores: o canal utilizado para envio da mensagem e o componente de processamento da *Beagle*. Na nossa aplicação, utilizamos um *switch HP V1810G*, que apresenta um comportamento *não determinístico*, isto é, não é possível prever com certeza o tempo necessário para que um pacote seja entregue a um determinado *host*. O segundo aspecto a ser considerado é que aplicações rodando na *PRU* não dividem recursos com outros processos, ao contrário de aplicações que rodam em *embedded linux*. Na seção 3.2.4, comentamos que, apesar do *Linux* fornecer políticas de preempção *real-time*, o comportamento chamado de *hard real-time* não poderia ser alcançado pelo *kernel*. Em oposição, as unidades *PRUs* da *Beagle* foram desenvolvidas para este propósito.

3.5 Alternativa de solução: *Industrial Ethernet*

Alguns protocolos foram desenvolvidos a fim de alcançar exigências de *hard real-time* sobre redes *Ethernet*. Um deles é o *EtherCAT (Ethernet for Control Automation Technology)*, desenvolvido pela empresa alemã *Beckhoff*, que utiliza as unidades de processamento *real-time (PRUs)*. Um dos inconvenientes desta aplicação em relação à *BeagleBone Black* é que o seu processador, *Sitara AM3358*, não suporta este protocolo, sendo que os únicos da mesma família que o suportam são *AM3357* e *AM3359*. A *Texas Instruments* fornece um *kit* de desenvolvimento com o último, chamado de *AM3359 Industrial Communications Engine*, e uma biblioteca desenvolvida para o uso deste protocolo, *SYS/BIOS Industrial Software Development Kit (SDK)*.

Além do *EtherCAT*, outras alternativas para comunicação *real-time* sobre *Ethernet* e que são suportadas pelo processador AM3358 são *Ethernet/IP*, *PROFINET RT/IRT* e *PROFIBUS*.

4 Manutenção e Implementação de Clientes PROSAC

4.1 Introdução

O *PROSAC* é um *firmware* desenvolvido no grupo de controle, cujo principal propósito é receber requisiões da sala de controle e acionar a respectiva placa do bastidor. Esse *firmware* apresenta, atualmente, suporte a diversas placas que são usadas atualmente no *UVX* para controle de fontes e monitoramento de sensores. Exemplos de placas operadas pelo *PROSAC* são a *LOCON* de 12 e 16 *bits*, a *STATFNT* e a *DIGINT*.

Considerando a sua importância no contexto do sistema controle, dois clientes foram implementados a fim de testar seu funcionamento: um em Java e outro, utilizando o *kit* de desenvolvimento *STM32F7 Discovery*.

4.2 Manutenção do cliente *PROSAC* escrito em Java

O cliente *PROSAC* implementado em Java foi desenvolvido em 2011 por Bruno Martins para testes iniciais do *PROSAC*. Desta forma, casos mais complexos, como placas desenvolvidas posteriormente ao cliente, produziam exceções que interrompiam o programa. As seguintes modificações foram realizadas a fim de corrigir tais problemas:

- i. Função `processCommand` da classe `Client`: foi adicionado uma condição para verificar se a quantidade de *bytes* recebidos do *PROSAC* é a mesma que a esperada. Tal condição encontra-se no laço `for` do bloco `default` do `switch`.
- ii. Modificação de forma que os *status* das placas apareçam em uma janela distinta daquela onde o painel de comando está inserido. A classe `Boards` foi substituída pela `BoardsFrame`.
- iii. Suporte às placas *Statfnt*, *Digint*, *Rux* 12 *bits* bipolar e *Mux* 16 *bits*, e implementação das respectivas interfaces gráficas.
- iv. Correção nas escalas dos gráficos para as placas monopolares de 12 e 16 *bits*.
- v. Para a operação de rampa, o *PROSAC* necessita da ordem com que as placas aparecem no bastidor, não seus *IDs*. Por exemplo, se duas placas estiverem presentes, cujos *IDs* são 5 e 19, e quisermos executar uma rampa na 5, temos que enviar sua posição do bastidor, isto é 0.

4.3 Implementação para o *kit STM32F7 Discovery*

O *kit STM32F7 Discovery* oferece diversos recursos como interface *Ethernet*, *I2C*, *UART* e tela *LCD Touch* capacitiva. Os principais passos na implementação desta solução foram:

- i. Configuração de plugins para desenvolvimento na *IDE Eclipse*.
- ii. Configuração da interface *OpenOCD*, responsável pela comunicação entre placa e computador.
- iii. Implementação de um projeto na aplicação *STMCubeMX* para inicialização dos pinos dos módulos que serão utilizados no projeto.
- iv. Adição dos *middlewares FreeRTOS* e *LwIP* ao projeto.
- v. Correção do `stm32f7_hal_conf.h` com a definição dos registradores corretos do módulo *PHY*.

vi. Criação de uma interface gráfica, capaz de reconhecer eventos de *touch*.

A lógica utilizada nesta aplicação foi adaptada do cliente *Java* e, portanto, apresenta os mesmos comportamentos. Duas *threads* são criadas, sendo que uma envia requisições de leitura de dados a todo momento ao *PROSAC* e a outra, comandos requisitados pelo usuário, como ciclagem e rampa.

5 Servidores de variáveis EPICS

5.1 Introdução

EPICS, do inglês *Experimental Physics and Industrial Control System*, é um conjunto de ferramentas e aplicações que permitem monitorar e controlar sistemas que possuem, além de milhares de variáveis, uma larga e complexa rede ligando centenas de computadores e equipamentos. Tal sistema já é utilizado em diversos outros síncrotons e laboratórios no mundo todo e o seu desenvolvimento é realizado cooperativamente entre tais organizações. O principal intuito é proporcionar aos operadores de sistemas de grande porte uma visão global dos diversos módulos, facilitando, desta maneira, a identificação e previsão de problemas, assim como a escolha da melhor ação corretora. Entre as ferramentas oferecidas, destaca-se, por exemplo, arquivadores de variáveis, monitores de alarmes e suporte para construção de *interfaces* gráficas específicas para determinadas aplicações, que serão discutidos nas próximas seções deste documento.

Atualmente, no *UVX*, o controle é realizado através de módulos independentes, muitas vezes implementados em linguagens de programação distintas, que fornecem soluções específicas para um determinado equipamento e baixo grau de interação com outros programas. A identificação de problemas é, portanto, dificultada e depende muito mais de experiências anteriores dos operadores. Sendo assim, a implementação de *EPICS* para o sistema controle do *Sirius* está sendo considerada fortemente. Além disso, o acelerador linear de elétrons, que está sendo atualmente desenvolvido por uma empresa chinesa, a ser utilizado neste novo síncroton também oferecerá uma *interface* de comunicação com total compatibilidade com aplicações *EPICS*.

De maneira simples, um sistema *EPICS* é composto por diversos servidores e clientes espalhados pela rede, capazes, respectivamente, de publicar ou obter os estados dos diversos dispositivos pertencentes à estrutura. Tais estados são chamados de variáveis, ou somente PVs, e podem assumir vários tipos (binário, inteiro, *float* . . .), faixas de operação e taxas de aquisição que dependem principalmente da maneira de como os equipamentos proveêm tais dados. Cabe aos servidores (chamados também de *Input/Output Controllers* ou *IOCs*), portanto, realizar as operações de escrita ou leitura no *mundo real* ou obter tais informações de outros dispositivos, como osciloscópios e sensores, e publicá-las aos clientes. A comunicação entre clientes e servidores é realizado através do protocolo de rede *Channel Access*, implementado para redes que oferecem grande largura de banda, capaz de garantir o atendimento de especificações de *soft real-time* (consultar seção 3.2.4 para mais informações).

As próximas subseções serão dedicadas ao detalhamento de duas possíveis implementação de servidores de variáveis. A primeira utiliza a biblioteca *pcaspy* de *Python*, enquanto que a outra, um módulo chamado *StreamDevice*.

5.2 Utilizando a biblioteca PCASpy

O módulo *PCASpy* (PCAS do inglês *Python Channel Access Server*) fornece classes que realizam tanto a comunicação com um cliente *Channel Access* quanto com os dispositivos medidores. Esse módulo oferece quatro classes, porém, em princípio, utiliza-se apenas duas, sendo elas *SimpleServer* e *Driver*.

A classe `SimpleServer` abstrai os detalhes da comunicação entre servidor e clientes e redireciona requisições de leitura e escrita para um objeto da classe `Driver`, que por sua vez, se comunica com o dispositivo medidor. Um objeto da classe `SimpleServer`, portanto, é a ponte entre os clientes e tais equipamentos. Entre seus métodos, destaca-se:

- `createPV(prefix, pvdb)`: registra e torna disponível as variáveis contidas no dicionário `pvdb`. A documentação do módulo na *Internet* contem os campos válidos para configuração de uma *PV*.
- `process(time)`: o servidor processa requisições dos clientes a cada *time* segundos.

A classe `Driver` deve ser redefinida a fim de refletir os detalhes de comunicação e protocolo. Para tal, através de mecanismos de herança, redefine-se os métodos `read()` e `write()`, que realizam, respectivamente, a leitura e escrita de variáveis. Além destas duas funções, essa classe mantém um registro interno dos valores das variáveis. Para modificá-lo, dois outros métodos são usados, sendo eles `getParam()` e `setParam()`. Enfim, a função `updatePV()` é chamada para comunicar ao objeto `SimpleServer` que uma *PV* foi modificada.

5.3 EPICS StreamDevice

O EPICS *StreamDevice* oferece suporte para a comunicação com dispositivos que podem ser controlados através do envio de *strings*, isto é, cadeias de *bytes*. Dessa forma, dispositivos que possuem interface *serial* de dados, como os padrões RS-232 e RS-485, podem se comunicar com um *StreamDevice* e fornecer os valores para as diversas variáveis EPICS configuradas.

A configuração de um *StreamDevice* é realizada a partir de arquivos ditos do tipo *protocolo*. Tais arquivos especificam os formatos e os padrões que são entendidos pelo dispositivo e determinam, portanto, os *bytes* que devem ser enviados para cada tipo de operação. Muitas opções de comandos e formatos são disponíveis nesta ferramenta, sendo que as principais serão detalhadas nas próximas subseções. As variáveis que são disponibilizadas à rede são, por sua vez, especificadas no arquivo *database*, que é lido durante a inicialização de um *StreamDevice*. As operações de leitura e escrita destas respectivas variáveis devem estar presentes em um dos arquivos de *protocolo*.

A comunicação das mensagens para o meio físico propriamente dita não é realizada pelo *StreamDevice*. Por padrão, tal ferramenta possui uma *interface* com um *driver* de comunicação assíncrona chamado *asynDriver*, que deve ser instalado separadamente. De modo prático, o módulo *StreamDevice* é responsável por converter os *bytes* recebidos e enviados em variáveis EPICS, enquanto que o *asynDriver* realiza a comunicação física com os dispositivos. É importante notar que é o uso do *asynDriver* não é imposto pelo *StreamDriver*: é possível, portanto, substituí-lo por qualquer outro *driver* de comunicação caso desejado.

As próximas subseções são dedicadas ao processo de instalação e configuração de um *StreamDevice* para a bomba de vácuo *Agilent 4UHV*, que será utilizada no projeto *Sirius*.

5.3.1 Instalação do StreamDevice

A primeira etapa consiste na instalação do *asynDriver*. Para isso, é necessário seguir os seguintes itens abaixo. Supõe-se que a biblioteca de funções EPICS já foi instalada conforme será apresentado na seção 6.2.1. O diretório onde os arquivos do *asynDriver* serão instalados será referenciado por `$INSTALL_DIR/asyn4-29/`.

- Faça o *download* do pacote através de <http://tinyurl.com/hy3sra7> e descompacte-o.
- Execute o arquivo `Makefile` através do comando `make`. Os diversos que compõem o *driver* serão compilados nesta fase.

O *StreamDevice* está disponível a partir do *link* <http://tinyurl.com/z8rkc7y>.

6 EPICS Archiver Appliance

6.1 Introdução

O EPICS *Archiver Appliance*, desenvolvido pelo instituto americano *National Accelerator Laboratory (SLAC)*, é capaz de monitorar e arquivar um grande número de variáveis, 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 transformá-los a fim de que as aplicações possam processá-los posteriormente;

A figura 15 esquematiza o modo de funcionamento do *EPICS Archiver Appliance*.

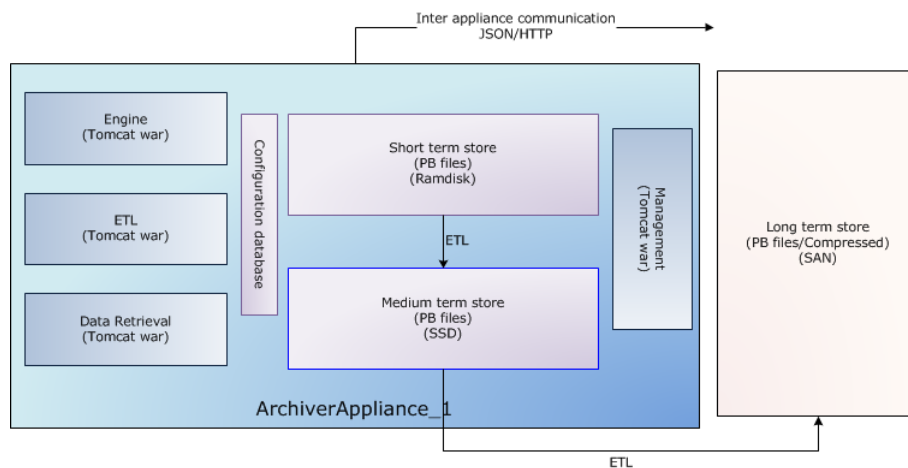


Figura 15: Modo de funcionamento de uma *appliance*

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 automatizar 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.

6.2 Instalação

Esta subseção é dedicada às etapas necessárias para a instalação do EPICS Archiver Appliance. Neste tutorial serão apresentadas dois métodos: um que utiliza os repositórios oficiais do Ubuntu 16.04 para obter os pacotes e o outro, os servidores dos próprios provedores dos serviços.

6.2.1 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 pretendida. 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=${JAVA_HOME}/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. No nosso caso, senha escolhida foi `toor`.

```
$ 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://tinyurl.com/zewzg72
$ tar -xvzf apache-tomcat-8.5.4.tar.gz
$ rm apache-tomcat-8.5.4.tar.gz
$ cd apache-tomcat-8.5.4/
```


Defina a variável de ambiente `{CATALINA_HOME}`, responsável por referenciar o diretório *apache-tomcat-8.5.4/* 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-8.5.4
```

No *Ubuntu* 16.04, a instalação pode ser realizada com o comando

```
sudo apt-get install tomcat8
```

E defina a variável `{CATALINA_HOME}` como

```
export CATALINA_HOME = /usr/share/tomcat8
```

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

```
$ cd $INSTALL_DIR
$ wget http://tinyurl.com/z8gnxmk
$ tar -xvzf httpd-2.4.23.tar.gz
$ rm httpd-2.4.23.tar.gz
$ cd httpd-2.4.23
```

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-2.4.23
$ ./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.

No *Ubuntu* 16.04, a instalação é realizada através do comando abaixo. A configuração deste módulo é um pouco distinta daquela documentada anteriormente, sendo que é necessário copiar os arquivos `proxy-balancer.*`, `proxy.*`, `proxy-http.*`, `status.*` e `lbmethod_byrequest.*` da pasta `mods-available/` para `mods-enabled/`. Os dois últimos diretórios encontram-se em `/etc/apache2/`. Enfim, a configuração dos *proxies* deve ser realizada no arquivo `proxy.conf` da mesma maneira que foi explicada anteriormente.

```
$ sudo apt-get install apache2
```

O servidor é iniciado, por sua vez, através de

```
$ sudo systemctl start apache2.service
```

6.2.2 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/archappl_v0.0.1_SNAPSHOT_22-June-2016T10-25-27
$ cd $INSTALL_DIR/archappl_v0.0.1_SNAPSHOT_22-June-2016T10-25-27
$ wget http://tinyurl.com/zom7pbw
$ tar -xvzf archappl_v0.0.1_SNAPSHOT_22-June-2016T10-25-27.tar.gz
$ rm archappl_v0.0.1_SNAPSHOT_22-June-2016T10-25-27.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, dentro dela, o arquivo `lnls_appliances.xml`. Tal pasta será chamada de `lnls-control-archiver_v0.0.1_22-June-2016T10-25-27`.

```
$ mkdir $INSTALL_DIR/lnls-control-archiver_v0.0.1_22-June-2016T10-25-27
$ cd $INSTALL_DIR/lnls-control-archiver_v0.0.1_22-June-2016T10-25-27
$ nano 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_control_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_control_appliance`). Os números das portas foram escolhidos aleatoriamente, mas, por sugestão do desenvolvedor, deve-se adotar uma sequência crescente a partir da porta do módulo `mgmt`, que, no nosso caso, vale 11995. É importante observar que `localhost` deve ser substituído pelo endereço que será utilizado para referenciar os *servlets* remotamente. `<data_retrieval_url>` é o endereço que deverá ser utilizado a fim de enviar e obter requisições HTTP e JSON, conforme figura 15 e `<cluster_inetport>` é a combinação TCP/IP `address:port` usado para a comunicação entre as *appliances*. Por fim, adicione as variáveis de ambiente ao arquivo `.bashrc`.

```
export ARCHAPPL_APPLIANCES=$INSTALL_DIR/lnls-control-archiver_v0.0.1_22-June-2016
    T10-25-27/lnls_appliances.xml
export ARCHAPPL_MYIDENTITY=lnls_control_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/archappl_v0.0.1_SNAPSHOT_22-June-2016T10-25-27/install_scripts
$ ./deployMultipleTomcats.py $INSTALL_DIR/lnls-control-archiver_v0.0.1_22-June
    -2016T10-25-27
```

Serão criadas 4 pastas dentro de `${INSTALL_DIR}/lnls-control-archiver_v0.0.1_22-June-2016T10-25-27`, 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 desenvolvedor 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 `$INSTALL_DIR/archappl_v0.0.1_SNAPSHOT_22-June-2016T10-25-27`. Copie-o para `$INSTALL_DIR/lnls-control-archiver_v0.0.1_22-June-2016T10-25-27` e renomeie-o para `lnls_policies.py`. Enfim, defina variáveis de ambiente usadas por este arquivo.

```
export ARCHAPPL_POLICIES=$INSTALL_DIR/lnls-control-archiver_v0.0.1_22-June-2016
T10-25-27/lnls_policies.py
export ARCHAPPL_SHORT_TERM_FOLDER=$INSTALL_DIR/lnls-archiver-storage/STS
export ARCHAPPL_MEDIUM_TERM_FOLDER=$INSTALL_DIR/lnls-archiver-storage/MTS
export ARCHAPPL_LONG_TERM_FOLDER=$INSTALL_DIR/lnls-archiver-storage/LTS
```

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 `lnls-archiver-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 controle.

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

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/archappl_v0.0.1_SNAPSHOT_22-June-2016T10-25-27/install_scripts/
archappl_mysql.sql
```

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

```
$ wget http://tinyurl.com/jf3cadl
```

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="controle"
        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>

```

Note que modificamos os campos `username`, `password` e `url` de acordo com a instalação no item v.

- vii. É necessário copiar e extrair todos os arquivos `.war` presentes em `archappl_v0.0.1_SNAPSHOT_22-June-2016T10-25-27` para os respectivos diretórios `webapps/`. Para tal, definimos:

```

export DEPLOY_DIR=$INSTALL_DIR/lnls-control-archiver_v0.0.1_22-June-2016T10-25-27
export WARSRC_DIR=$INSTALL_DIR/archappl_v0.0.1_SNAPSHOT_22-June-2016T10-25-27

```

E executamos o *script* `lnls-archiver-extract-wars.sh`, presente no repositório do grupo, para extrair os arquivos `.war`.

- viii. Enfim, os 4 *Tomcats* são lançados através do serviço *systemd*, que pode ser obtido através do repositório *git* do grupo, conforme comando abaixo. Tal serviço deve ser *instalado* do diretório `/lib/systemd/system`.

```
$ sudo systemctl start lnls-archiver.service
```

- ix. O arquivador é acessado através da URL (`localhost` pode ser substituído pelo endereço IP que referencia o *host*)

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

6.2.3 Debugging

A lista abaixo apresenta algumas situações que foram encontradas durante a instalação.

- Se após o reinício do arquivador todos os dados referentes às variáveis forem perdidos é necessário checar todos os arquivos `conf/context.xml` de cada um dos *tomcats* e verificar que a configuração do servidor *MySQL* foi realizada corretamente.
- Uma regra geral é verificar se todas as variáveis de ambiente foram corretamente configuradas.

6.3 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 6.2.2, 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`.

6.4 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 16 representa o resultado da implementação.

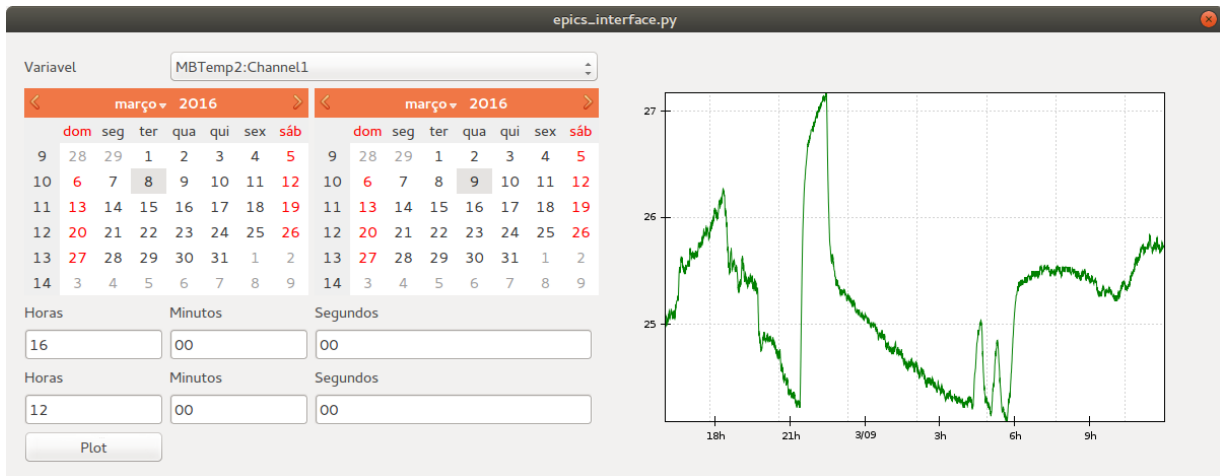


Figura 16: Interface *Qt* implementada em *python*.

7 Best Ever Alarm System Toolkit - BEAST

7.1 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 17, logo abaixo.

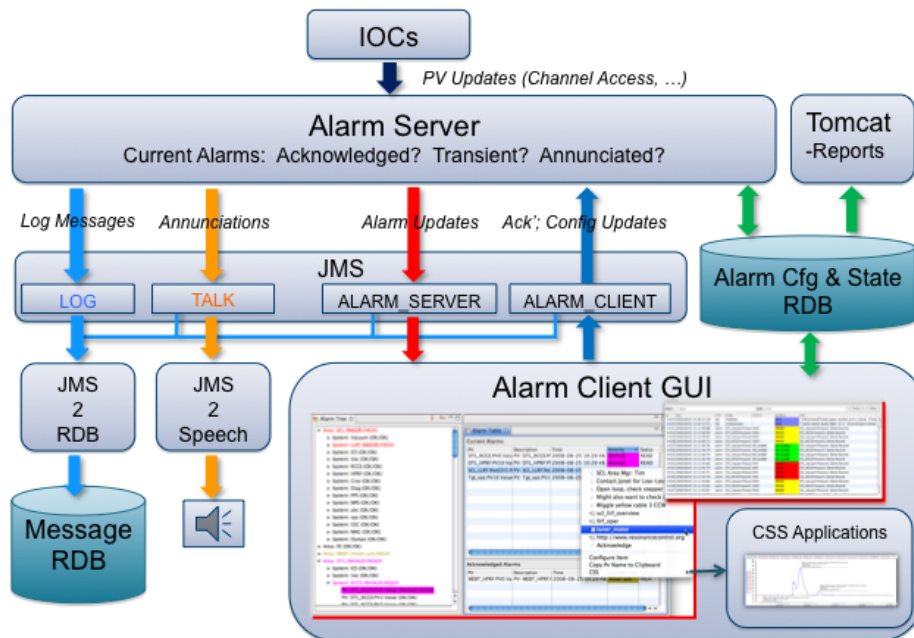


Figura 17: Implementação do sistema de monitoramento de alarmes *BEAST*.

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 variáveis, 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* apresentadas 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*.

7.2 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*. Uma sugestão de instalação é apresentada abaixo.

- i. É necessário inicialmente fazer o *download* do código fonte do *CS Studio* para obter o produto relacionado ao *Alarm server*. Para isso, acesse o repositório do projeto no *GitHub* e extraia os arquivos em um diretório. Usaremos aqui o mesmo caminho especificado por `$INSTALL_DIR`, utilizado também na seção 6.

```
$ cd $INSTALL_DIR/
$ wget https://github.com/ControlSystemStudio/cs-studio/archive/master.zip
$ unzip cs-studio-master.zip
$ rm cs-studio-master.zip
$ cd cs-studio-master/
```

Na pasta `cs-studio-master/`, estão contidos todos os arquivos com os códigos-fonte do *CS Studio*, porém só utilizaremos o diretório `applications/`, que é o onde está a implementação do *Alarm server*.

- ii. Antes de executar o servidor, é necessário primeiro configurar o banco de dados com as tabelas utilizadas por ele. Para isso, diversos *scripts* de instalação são disponibilizados pelo desenvolvedor em `applications/alarm/alarm-plugins/org.csstudio.alarm.beast/dbd`. Vamos usar parte do conteúdo contido em `ALARM.MYSQL.sql`, uma vez que este arquivo adiciona algumas entradas nas tabelas, a fim de testar a aplicação. Abra este arquivo e comente as linhas a partir de `INSERT INTO ALARM.ALARM_TREE VALUES (3, 2, 'System', now());`. As linhas acima desta sentença criam a hierarquia, que também nos será útil. Execute, então, os seguintes comandos:

```
>> CREATE DATABASE lnls_alarms;
>> GRANT ALL ON lnls_alarms.* TO 'lnls_alarm_user'@'%' IDENTIFIED BY 'controle';
>> exit
$ mysql -u lnls_alarm_user -p
>> source applications/alarm/alarm-plugins/org.csstudio.alarm.beast/dbd/ALARM_MYSQL.sql;
```

Verifique que as tabelas foram criadas corretamente com:

```
>> SHOW TABLES;
```

- iii. A próxima etapa é configurar o servidor de mensagens *JMS*. Será necessário fazer o *download* do pacote *Apache ActiveMQ*.

```
$ cd $INSTALL_DIR/
$ wget http://tinyurl.com/htw5tdx
$ tar -xvzf apache-activemq-5.13.0-bin.tar.gz
$ rm apache-activemq-5.13.0-bin.tar.gz
$ cd apache-activemq-5.13.0/
```

O diretório `conf/` contém um arquivo, `activemq.xml`, com as configurações do servidor. É possível, por exemplo, modificar a porta, que por padrão é 61616, ou adicionar autenticação via *jaas*. Para tal, adicione as seguintes linhas neste arquivo.

```
<plugins>
  <jaasAuthenticationPlugin configuration="PropertiesLogin" />
</plugins>
```

Neste caso, *PropertiesLogin* é o nome da configuração contido no arquivo `login.config`, que especifica os *plugins* necessários e os arquivos onde estarão armazenados os usuários, suas senhas (`users.properties`) e grupos (`groups.properties`). Por fim, inicie o servidor com

```
$ cd $INSTALL_DIR/apache-activemq-5.13.0/bin/linux-x86-64/
$ ./activemq start
```

No *Ubuntu* 16.04, a instalação do `activemq` pode ser feita através dos repositórios oficiais, por meio do comando `apt-get install activemq`. Os arquivos de configuração encontram-se, por sua vez, em `/etc/activemq`.

- iv. Tendo instalado os componentes responsáveis pela comunicação entre módulos e base de dados de configurações, podemos iniciar o servidor. Para isso, vamos utilizar a distribuição *Luna* do *Eclipse for RCP and RAP Developers*, por apresentar melhor compatibilidade com os pacotes gráficos do *CS Studio*. Extraia e inicie a aplicação.

```
$ cd $INSTALL_DIR/
$ wget http://tinyurl.com/jbmmue7
$ tar -xvzf eclipse-rcp-luna-SR2-linux-gtk-x86_64.tar.gz
$ rm eclipse-rcp-luna-SR2-linux-gtk-x86_64.tar.gz
$ cd eclipse/
$ ./eclipse
```

Importe o projeto que contém o servidor de alarmes. Para tal, entre em `File > Import ...> General > Existing Projects into Workspace` e escolha o diretório `$INSTALL_DIR/applications/alarm/`. Dentro deste projeto, abra o arquivo `AlamServer.product` contido em `alarm-plugins > org.csstudio.alarm.beast.server`. Espera-se que uma tela parecida com a figura 18 seja aberta.

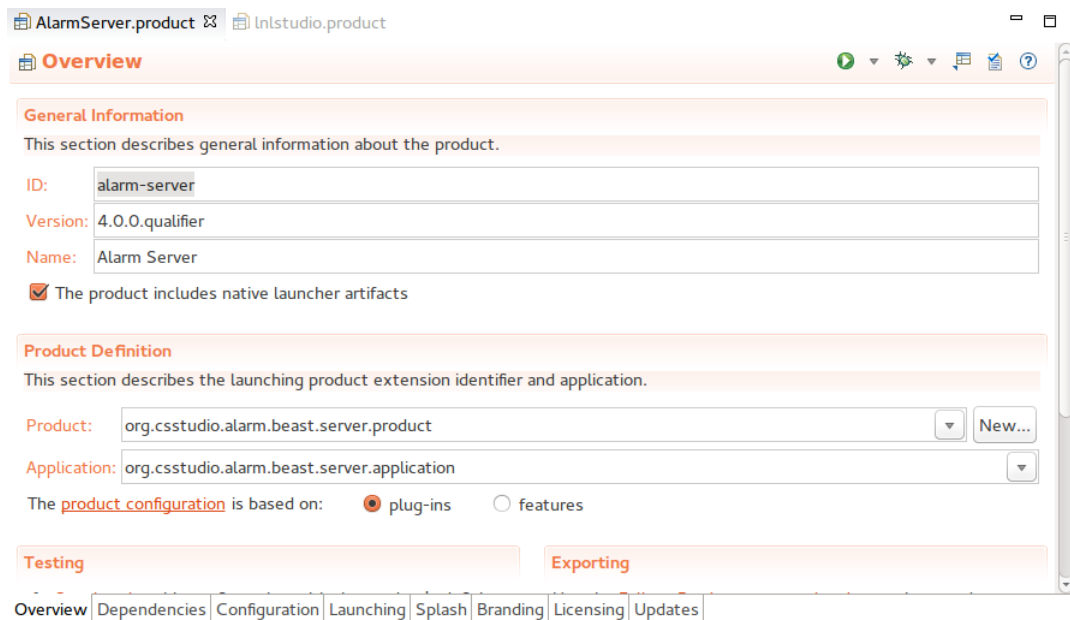


Figura 18: Configurações do produto `AlarmServer.product`.

Ainda não é possível executá-la, visto que este produto requer algumas dependências. Para incluí-las à instalação do *Eclipse*, acesse `Help > Install New Software... > Add...` e adicione a url <http://download.controlsystemstudio.org/updates/4.1>. Não é necessário realizar o *download* de todos os pacotes, somente os especificados na tabela 1 abaixo. A instalação também adiciona os componentes que permitem o acesso à *appliance*, conforme explicado na seção 6.3.

Tabela 1: *Plugins* necessários para o *AlarmServer*.

CS-Studio Applications		
Alarm Handler Tools	Alarm Handler UI	Appliance Archiver Reader (x2)
Application Utilities	Archive Reader RDB Feature	Archive Tools Feature
CS-Studio Epics v3 Support	CS-Studio RAP Utilities	Data Browser
Data Browser OPI Widget	ea4 Archiver Reader	Operator Interface Builder (BOY)
CS-Studio Core		
Core Auth Feature	Core Base Feature	Core Platform plugins
Core Platform RAP Plugins	Core UI Feature	Core Utility Feature
CS-Studio Command-line Execution Services Support	CS-Studio Common Data Layer (pvmanager)	CS-Studio DAL
CS-Studio Epics v4 Support	CS-Studio Extras Support	PVManager Autocomplete Feature
Maven osgi-bundles		
antlr	EPICS Plug-in	Graphene
JSON support for file data-source	JSR 353 API	JSR 353 Default Provider
org.antlr.runtime	org.epics.graphne	org.epics.util
org.epics.vtype	org.epics.vtype-json	org.python.jython
pvAccess - Java	pvData - Java	PVManager Core
PVManager EPICS Channel Access Support	PVManager EPICS PVAccess Support	PVManager Execute Support

PVManager File Support	PVManager Local PV Support	PVManager Simulated PV Support
PVManager System Support	PVManager VType Support	

O produto necessita também do *plug-in* `com.ibm.icu.base` contido no repositório *Orbit*, acessado através do *link* <http://tinyurl.com/zu8ybkj>. Selecione somente o pacote *International Components for Unicode for Java (ICU4J)* e instale-o.

O servidor já pode ser executado, porém ele utilizará uma configuração padrão para conectar-se ao *MySQL* e ao *JMS*. A fim de modificar tal configuração, editamos o arquivo *plugin_customization.xml*, presente no mesmo diretório. Esse arquivo deve refletir as instalações realizadas anteriormente.

Entre na aba *Launching* e adicione o argumento `-pluginCustomization` no campo *Program Arguments*. Tal argumento deve conter o caminho do arquivo *plugin_customization.xml*, conforme figura abaixo.

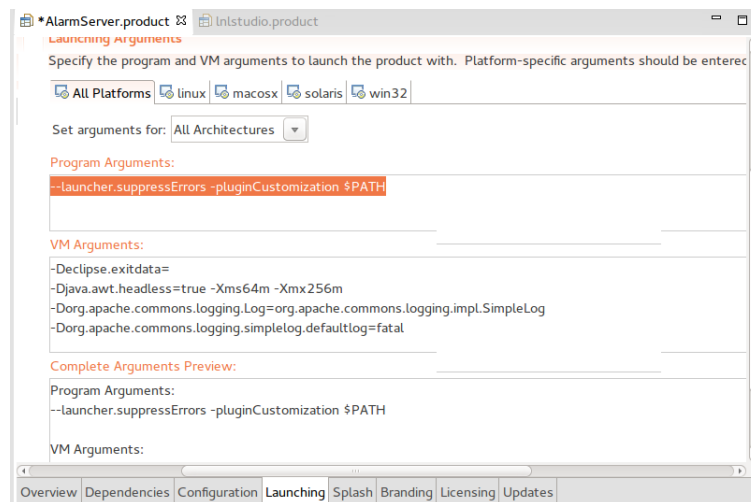


Figura 19: *Launching view* do produto `AlarmServer.product`. Substituir `$PATH` pelo caminho completo do arquivo *plugin_customization.xml*.

Execute, enfim, o produto apertando no ícone verde no canto superior direito. O servidor se encarregará da criação dos *topics* na aplicação *JMS*. O resultado esperado está representado na figura 20. *Annunciator* é o nome dado à raiz da hierarquia criada no item ii (verificar conteúdo do arquivo `ALARM.MYSQL.sql`) e define, assim, quais tópicos serão utilizados na comunicação.

Para uma execução independente do cliente do *Eclipse*, é possível exportar o produto através da ferramenta *Eclipse Product export wizard*, que pode ser acessado através do ícone próximo ao botão verde de *Launching*.

```
OpenJDK 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed in 8.0
2016-03-10 11:21:08.732 INFO [Thread 1] org.csstudio.alarm.beast.server.Application (start) - Alarm Server 4.1.1.201509181557 start
Alarm Server 4.1.1.201509181557
Configuration Root: Annunciator
Database URL:      jdbc:mysql://localhost:3306/ALARM
JMS URL:          failover:(tcp://localhost:61616)
JMS Server Topic: Annunciator_SERVER
JMS Client Topic: Annunciator_CLIENT
JMS Talk Topic:   Annunciator_TALK
JMS Global Topic: GLOBAL_SERVER
EPICS Addr. List: 127.0.0.1
```

Figura 20: Resultado da execução do produto *AlarmServer*.

7.3 Uso da interface *Eclipse* como *Alarm Client GUI*

Para configurar o *Eclipse* como um cliente do servidor de alarmes, é necessário configurá-lo para que ele acesse os servidores *MySQL* e *JMS*. Isso é realizado através de Window > Preferences > CSS Applications > Alarm > Alarm System. Modifique os campos¹ *url*, *username* e *password*, conforme configurado anteriormente, e reinicie o *Eclipse*. Se tudo foi configurado corretamente, já é possível acessar os alarmes configurados através dos componentes comentados no item iv da seção 7.1. Tais componentes são acessados a partir de Window > Show View > Other > CSS.

7.3.1 *Debugging*

Alguns erros foram encontrados durante a instalação:

- i. *A interface não salva corretamente as senhas do banco de dados e do JMS*: para resolver este problema basta redefinir as senhas que o *Eclipse* utiliza para criptografar as senhas salvas. Acesse Window > Preferences > General > Security > Secure Storage e redefina as senhas contidas na tabela *Master password providers*.
- ii. *Não é possível adicionar, remover ou configurar variáveis a partir da Alarm Tree View*: conforme explicado na seção **Introdução**, o *BEAST* oferece suporte à autenticação e autorização de usuários. Sendo assim, somente usuários cadastrados são permitidos de alterar ou reconhecer alarmes. Como a rede do *Sirius* é prevista para ser isolada, tal serviço não será necessário. Portanto, podemos configurar o *Eclipse* para dar permissão completa a qualquer *user*. Para tal, devemos modificar o plugin `org.csstudio.security`, de extensão `.jar` presente no diretório `$INSTALL_DIR/eclipse/plugin`. Abra esse arquivo com o *Archive Manager* (em sistemas *linux*), modifique o campo `alarm_config` para `.*` no arquivo `authorization.conf` e reinicie o *Eclipse*. Deve ser possível alterar configurações de alarmes a partir de agora.

7.3.2 Obtendo um *LNLStudio* a partir da configuração do *Eclipse*

É possível exportar a configuração atual do *Eclipse* como um novo produto, que chamaremos de *LNLStudio*. Para isso, criamos um novo *Plug-in Project* e adicionamos um novo arquivo do tipo *Product Configuration*. Neste arquivo, definimos nome, *id*, versão e, no campo *Product Definition*, definimos *Product* como `lnlstudio.product` e *Application*, `org.eclipse.ui.ide.workbench`. Na aba *Dependencies*, é preciso especificar quais *plug-ins* (ou *features*) são necessários ao produto. Por questão de simplicidade, adicionamos todos aqueles que estão instalados (*Add...* e *Ctrl+A* para selecionar todos). Para configurar a *splash screen*, clique na aba *Splash* e adicione uma *progress bar*, uma *progress message* e um arquivo chamado *splash.bmp* ao diretório do projeto. Essa imagem deve possuir dimensões 455x295.

O produto pode ser lançado através do ícone verde presente no canto superior da tela.

Para exportar este produto, clique em *Eclipse Product export wizard* na seção *Exporting* da aba *Overview* e escolha o endereço e formato (arquivo `.zip` ou pasta) para o qual ele deve ser exportado. Para executar, entre na pasta criada (ou extraída do arquivo) e execute o aplicativo `eclipse`.

¹*RDB* significa *Relational Database*, como o banco de dados *MySQL*.