LNLS - LABORATÓRIO NACIONAL DE LUZ SÍNCROTRON

GUSTAVO CIOTTO PINTON

RELATÓRIO DE ESTÁGIO

Campinas, 17 de novembro de 2016

LNLS - LABORATÓRIO NACIONAL DE LUZ SÍNCROTRON

GUSTAVO CIOTTO PINTON

RELATÓRIO DE ESTÁGIO

Estágio realizado junto ao Grupo de Controle para o Laboratório Nacional de Luz Síncrotron - LNLS -, sob a orientação de José Guilherme Ribas Sophia Franco, desde de fevereiro de 2016.

José Guilherme Ribas Sophia Franco

James Francisco Citadini

Márcio Paduan Donadio

Campinas, 17 de novembro de 2016

Sumário

1 Introdução

Durante todo o ano de 2016, desenvolvi minhas atividades de estágio no Laboratório Nacional de Luz Sincroton (LNLS - CNPEM) no grupo de Controle, que é responsável por prover soluções de controle e sensoriamento 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 de controle que serão utilizadas no acelerador *Sirius*, que se encontra em fase de construção atualmente.

Nestes 10 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 na grande maioria das suas 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 a fim de disponibilizar um servidor NTP de *stratum* 1 à rede de controle. Além disso, entrei em contato com o desenvolvimento de algumas ferramentas relacionadas ao sistema de controle EPICS, tais como o *Control System Studio*, o arquivador *EPICS Archiver Appliance* e ao servidor de alarmes *BEAST*. As próximas seções são dedicadas aos detalhes de implementação destes e de outros problemas.

Cabe destacar que este relatório não segue rigorosamente o formato disponibilizado, dado que abrange muitos assuntos distintos e que o número de páginas é limitado a 20.

2 Implementação de um servidor NTP de *stratum* 1

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 provindas de tais fontes.

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

Neste relatório, serão discutidas as características da versão 4 do NTP, especificadas no RFC5905 [2], e sua aplicação no contexto do acelerador *Sirius*. 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 um comunicação. As variantes NTP são, portanto:

i. *server/client*: um cliente envia pacotes a um servidor requisitando sincronização, que responde utilizando o endereço contido nos respectivos pacotes. Nesta variante, servidores fornecem sincronização aos clientes, mas não aceitam sincronizações vindas deles. 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.

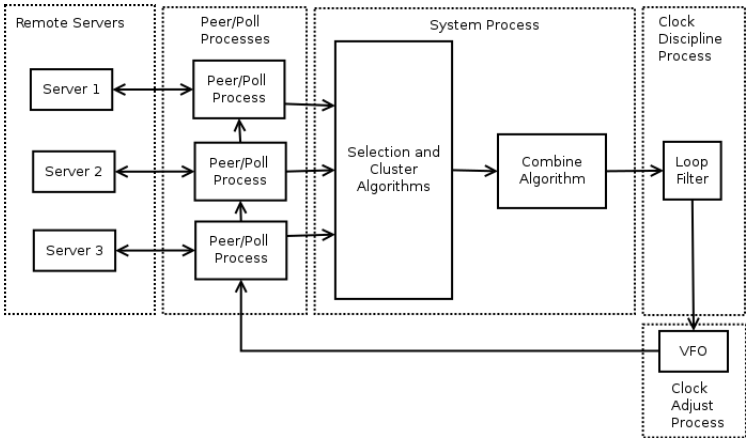


Figura 1: Implementação dos processos executados por um nó da rede. Extraída de [2].

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

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

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

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

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

   O *poll process* é responsável, por sua vez, por enviar pacotes aos servidores a cada intervalo de 2^τ segundos. τ varia de 3 a 17, resultando, assim, em intervalos de 8 segundos a 36 horas. O valor de τ pode variar durante a execução, sendo modificado pelo algoritmo regulador do relógio, que será discutido posteriormente.

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

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

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

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

2.3 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 [6]. 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, é necessário o uso do pulso de 1PPS, cuja finalidade é marcar o ínicio de um novo segundo. 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 microsegundos 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 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.

2.3.1 Descrição do *hardware*

No total, utilizou-se três modelos diferentes de GPS, sendo que dois deles tratam-se de *kits* contendo módulos GPS já integrados. Escolhemos, em um primeiro momento, os *kits Ultimate GPS Breakout* da *Adafruit* e *GPS Click* da *MikroElektronika*. O terceiro modelo, por sua vez, é o módulo receptador *CAM M8Q* da *ublox*, que foi posteriormente

incluído em uma placa também desenvolvida durante o estágio. As principais vantagens em relação a este último são o menor custo ($25 contra $39.95 e $49 dos dois *kits*) e maior independência no *design* do circuito de integração.

Todos 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 pelo *kernel* do sistema operacional da placa, dependendo, evidentemente, 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 todos provêem pulsos de largura superiores a $100ms$.

2.3.2 Configuração do servidor NTP

O primeiro passo para a configuração foi verificar se o *kernel* do *Linux* está provido do módulo *LinuxPPS*, responsável pela captura de pulsos 1PPS. Versões superiores à 2.6.34 já integram tal módulo por padrão [1]. A segunda etapa consistiu na configuração das portas de entrada e saída que serão usadas pela *BeagleBone Black* para se comunicar com o GPS. Ela foi realizada a partir da implementação de um arquivo de extensão *DTS*, cujo objetivo é modificar a função interna dos pinos da *Beagle*. 3 pinos, portanto, foram utilizados: 2 para o par *RX/TX* e um para a entrada PPS.

Em seguida, configurou-se os *daemons GPSD*, que implementa o protocolo de comunicação NMEA e realiza a troca de informação com o receptador GPS, e *NTPD*, responsável por ajustar o tempo do sistema, conforme seções anteriores. A comunicação entre estes dois processos é realizada por meio de um espaço de memória compartilhado disponível. É necessário certificar-se, ainda, que o *NTPD* foi compilado com o *driver ATOM*, que se encarrega do processamento dos pulsos PPS.

2.3.3 Implementação de variáveis EPICS

Duas implementações de servidores de variáveis EPICS foram escritas. A primeira, desenvolvida em *python*, extende as classes presentes no módulo *PCASpy* e utiliza os módulos `python-gps` e `ntplib` para comunicar-se com os *daemons GPSD* e *NTPD*, respectivamente. A segunda, por sua vez, consiste em uma ponte, escrita em C, entre o *Stream IOC* e as bibliotecas `libntpq` e `libgps`. De modo geral, o *Stream IOC* envia requisições, seguindo o protocolo BSMP, via um *socket unix* local para a *ponte*, que utiliza, por sua vez, as bibliotecas `libntpq` e `libgps` para obter os valores. Vale destacar que a `libntpq` não está disponível nos repositórios oficiais da maioria dos sistemas Linux atuais e, portanto, deve ser compilada separadamente a partir do código fonte do *NTPD*.

As variáveis EPICS geradas pelas implementações foram divididas em duas categorias principais, de acordo com a sua origem, isto é, se elas descrevem parâmetros do servidor NTP ou dados recebidos do receptor GPS. A tabela 1 resume as principais variáveis geradas pelo servidor e a 2, para o módulo GPS. A interface representada na figura 2 foi gerada a partir destas bibliotecas e utilizam as *PVs* descritas nas tabelas a seguir.

2.4 Construção de *capes* para a *BeagleBone Black*

Três *capes* foram implementados através do *Kicad*, um para cada receptador GPS. Tais arquivos podem ser encontrados no repositório do grupo.

Tabela 1: Variáveis definidas para o servidor *NTPD*.

Nome	Descrição
`NTP:Leap`	Indicação de um eventual *leap second* pendente a ser inserido ou removido.
`NTP:Stratum`	*Stratum* do servidor
`NTP:Refid`	ID de referência da fonte de sincronismo
`NTP:Offset`	*Offset* de tempo entre o relógio do sistema e da fonte de sincronismo
`NTP:Jitter`	Desvio padrão das medidas de *offset* mais recentes
`NTP:Precision`	Precisão do *clock* do sistema em *log2*
`NTP:Srcadr`	Endereço IP do servidor
`NTP:Version`	Versão do *daemon* NTPD e data de compilação
`NTP:Timestamp`	Diferença, em segundos, entre a data atual do servidor e a *unix epoch* (1 de Janeiro de 1970).

Tabela 2: Variáveis definidas para o receptor *GPS*.

Nome	Descrição
`GPS:Fix`	Indicação do tipo de *fix* (*No fix*, 2D ou 3D *fix*)
`GPS:Latitude`	Latitude medida pelo GPS
`GPS:Longitude`	Longitude medida pelo GPS
`GPS:Altitude`	Altitude medida pelo GPS
`GPS:Sattelites`	*String* contendo a identificação dos satélites usados para o *fix*
`GPS:Timestamp`	*Timestamp* fornecido, análogo à mesma variável do servidor NTP.

2.5 Resultados

A figura 3 representa os resultados obtidos para o receptor *Adafruit Ultimate GPS Breakout*. 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`. O segundo trata-se da fonte ligada ao PPS, controlada pelo *driver* `ATOM` que tivemos que incorporar na compilação do *NTP* e que utiliza a API *timepps* para se comunicar ao *kernel*. A fonte representada por `LOCAL`, que obtém a data do próprio sistema, é considerada apenas se as duas anteriores não estiverem disponíveis.

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 6μs atrás do pulso de PPS, e o jitter, 0.006 ms.

A figura 4, por sua vez, exibe o resultado da execução do mesmo comando, porém para o receptor da *MikroE GPS Click*. Observa-se que, como no caso anterior, o servidor conseguiu sincronizar-se ao pulso de PPS. Enfim, a figura 5 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 receptadores da *Adafruit* e da *MikroE*, e o último, enfim, é um servidor externo, referenciado pelo endereço `ntp.cnpem.br`. Nota-se que a diferença entre as fontes PPS está na ordem de unidades de microsegundos e que a diferença entre elas e a externa é da ordem de 100μs. Os testes para o módulo da *ublox* não foram concluídos antes da redação deste relatório, portanto não serão incluídos.

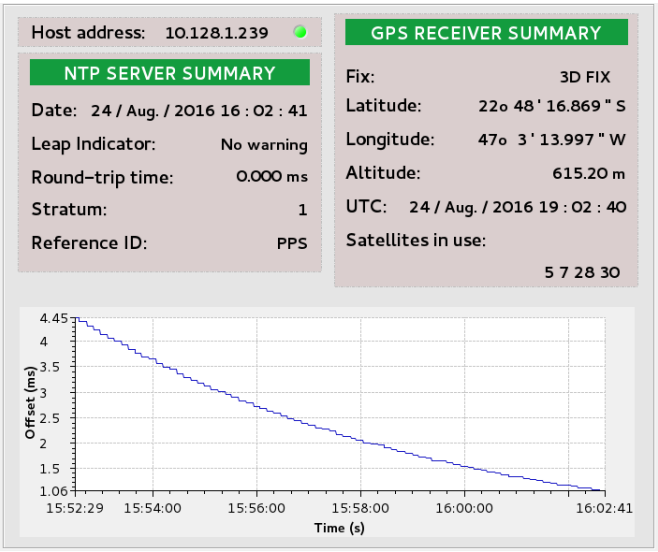


Figura 2: Interface construída para visualização das *PVs*.

```
     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 3: Comando `ntpq -p` na *BeagleBone* conectada ao *cape* com o receptor *Adafruit Ultimate GPS*.

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 imãs, 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*. 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 preeempção. Aplicações que rodam sobre *Linux* embarcado e que, portanto, compartilham recursos com outros processos, apresentam, geralmente, desempenho inferior. O sistema desenvolvido anteriormente

```
     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 4: Comando `ntpq -p` na *BeagleBone* conectada ao *cape* com o receptor *MikroE GPS Click*.

```
     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 5: Comando `ntpq -p` na *BeagleBone* cliente que obtêm sincronização de 3 servidores distintos.

pelo grupo utiliza a *PRU* para a recepção do sinal de sincronismo e ativa, posteriormente, seus nós escravos, conforme figura 6 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.

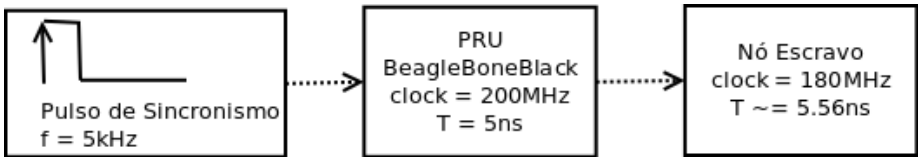


Figura 6: Esquema de sincronismo realizado pelo grupo de controle.

Nesta seção, será apresentada a alternativa descrita na figura 7, 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*.

3.2 Implementação

Esta subseção é dedicada a duas implementações 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* [7], 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 8 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*.

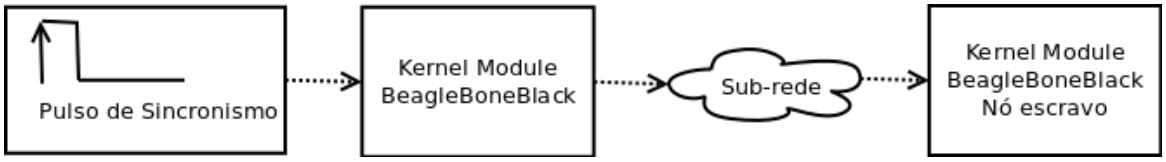


Figura 7: Sistema de sincronismo proposto.

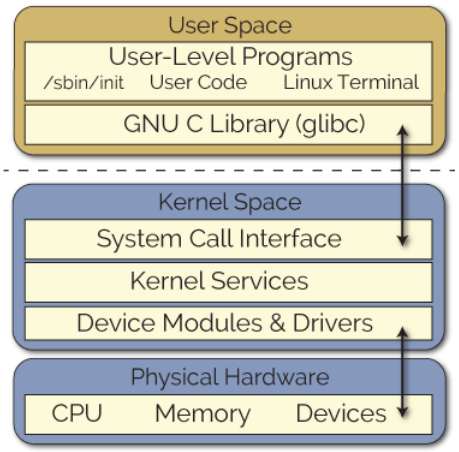


Figura 8: Comunicação entre *kernel* e *user spaces*. Extraída de [7].

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*, prepapar 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.

3.2.3 Implementação do *Kernel Module* de `BB1`

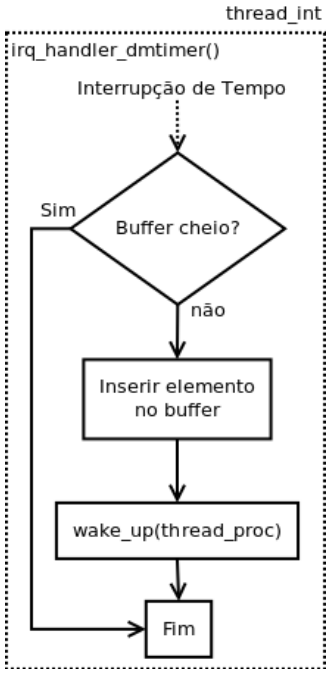
O *module* desenvolvido para a placa `BB1` instancia duas *threads*: uma para inicialização dos componentes e a outra para processamento e envio de pacotes *UDP*. No *kernel space*, cada *thread* é dada por uma estrutura do tipo `struct task_struct`, especificada na biblioteca `linux/kthread.h`, que provê também as funções `kthread_create` e `wake_up_process`. A primeira é reponsável 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 politíca 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*[5], isto é, que asseguram 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*. 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 9 resumem o funcionamento destas *threads*.

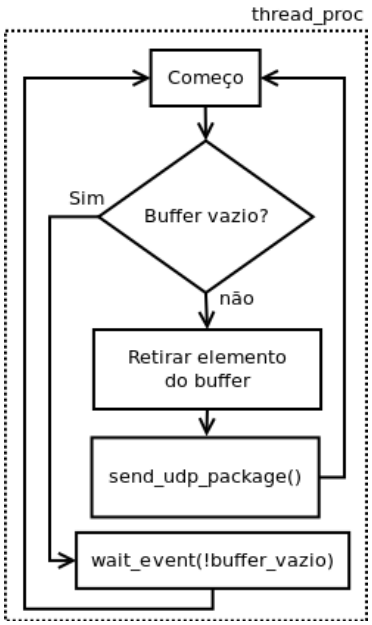
A figura 9a especifica o fluxograma da *thread_int* que preenche o *buffer* circular a cada interrupção do relógio e sinaliza para a *thread_proc* que ele não está mais vazio. A figura 9b evidencia que um pacote é enviado somente o *buffer* não estiver vazio. Caso contrário, a *thread_proc* se bloqueia 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* contém 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 frêquencia (*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.

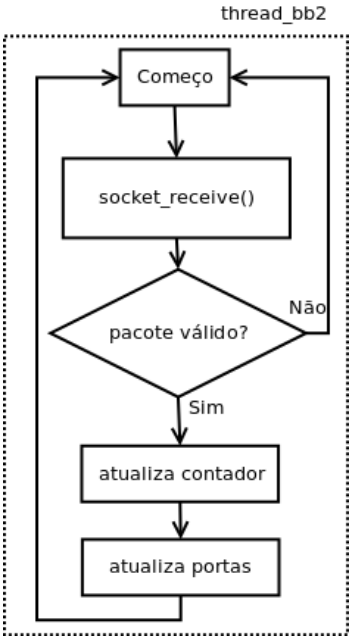
A função `omap_dm_timer_request()` retorna uma estrutura do tipo `struct omap_dm_timer` que representa um dos *timers* disponíveis no módulo. Há mais algumas variantes desta chamada, porém a principal é a `omap_dm_timer_request_specific(int timer_id)`, que recebe como parâmetro o *id* do *dmtimer* desejado. 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



(a) Fluxograma da *thread* que é interrompida pelo *timer*.



(b) Fluxograma da *thread* que envia pacotes à rede.



(c) Fluxograma da *thread* do *kernel module* em `BB2`.

Figura 9: Fluxogramas das *threads* que compõem `BB1` e `BB2`.

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 $f_{clock} = 32kHz$.

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

No nosso caso, desejamos interrupções a cada $t = 1ms$ com $PTV = 1$. A aplicação direta da equação 1 resulta em *TDLR* = 0xFFFFFFF7.

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 entra 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 valido 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*, por exemplo `0xc0a80216` para `192.168.2.22`, `INADDR_SEND` é o endereço para o qual deseja-se enviar o datagramas 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 funçao `sock_release()`.

3.2.4 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 9c 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.

3.3 Utilizando o módulo PRU

Implementações alternativas para os *kernel modules* de *BB1* e *BB2* foram propostas usando o módulo PRU da *BeagleBone Black*. Para *BB1*, a configuração do *timer* e a verificação das interrupções geradas por ele e pelo pulso de sincronismo são feitas pela PRU. Quando um *trigger* é recebido, um sinal é enviado para um programa rodando no *user space* da *Beagle*, cujo propósito é, de fato, enviar pacotes *UDP* para *BB2*. É importante observar que a comunicação entre o processador e o módulo PRU é garantida via *hardware* e que uma API contendo funções de comunicação com a PRU também é fornecida pela comunidade.

Um processo semelhante foi realizada para *BB2*, isto é, substituiu-se o *kernel module* por um programa rodando na unidade de tempo real. Tal programa verifica, através de *polling* nos registradores do controlador de rede, se um novo pacote foi recebido. Se isto ocorreu, então um sinal é enviado a um outro programa no *user space*, responsável por tratar tal pacote. O tratamento de tal pacote não é realizado diretamente na PRU, visto que seria necessária a implementação de toda a pilha *UDP/IP*.

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 10a para a implementação contendo os *kernel modules* e na 10b para a implementação utilizando a PRU. Os desvios padrões, ou *jitters*, obtidos são, respectivamente, $5.7ms$ e $985.1\mu s$, muito distantes daquele adquirido pelo sistema construído pelo grupo de controle, no qual utilizou-se somente 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*. Nas nossas aplicações, utilizamos um *switch HP V1810G*, que apresenta um comportamento *não determinístico*, isto é, não é possível prever com certitude 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 3.2.3, 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.

(a) Captura de tela do osciloscópio para a implementação com *kernel modules*.

(b) Captura de tela do osciloscópio para implementação com PRU.

Figura 10: Resultados encontrados para as duas implementações.

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 incovenientes 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 do PROSAC e Implementação de Aplicações Clientes

4.1 Introdução

O *PROSAC* é um *firware* desenvolvido no grupo de controle, cujo principal propósito é receber requisões da sala de controle e acionar a respectiva placa do bastidor. Esse *firware* 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, um artigo foi publicado na PCaPAC 2016 e dois clientes foram implementados a fim de testar seu funcionamento: um em Java e outro, utilizando o *kit* de desenvolvimento *STM32F7 Discovery*.

4.2 Manutenação do *PROSAC*

O grupo de controle publicou na PCaPAC 2016 um artigo entitulado *UVX Control System: An Approach With BeagleBone Black* [4], em que fui um dos co-autores. Minha tarefa neste trabalho, em poucas palavras, foi ajustar a política

de escalonamento e a prioridade das *threads* que compõem o *PROSAC*, de forma a obter o melhor desempenho possível nas tarefas síncronas realizadas pelo programa. As tarefas ditas *síncronas* são aquelas que necessitam ser sincronizadas a partir de um pulso de sincronismo externo. No *PROSAC*, tais tarefas consistem nas operações de ciclagem e rampa, isto é, a cada novo pulso recebido, o *PROSAC* é encarregado de transmitir um novo valor às placas que estiverem participando do processo. Devido às mesmas limitações de processamento de tempo real discuitidas na seção 3.2.3, o número de pulsos perdidos na nova versão do *PROSAC* era elevado para as frequências superiores a 500Hz, que tornava o seu uso proibitivo. Dessa forma, propus utilizarmos o *scheduler* denominado de *Completely Fair Scheduler* e ajustar a prioridade das *threads* de forma a garantir um maior tempo de CPU à *thread* responsável por processar os pulsos. A tabela 3 a seguir ilustra a taxa de pulsos perdidos para a configuração do *PROSAC* com o *Completely Fair Scheduler* conectado a duas placas LOCON de *12 bits*, ambas participando da ciclagem. Conforme tabela, as taxas de pulsos perdidos são muito inferiores aos pulsos totais recebidos, possibilitando, assim, seu uso no UVX.

Tabela 3: Resultados do *PROSAC* com o *Completely Fair Scheduler*.

Frequência (Hz)	Pulsos perdidos	Total	Razão
150	0	579.812	0
512	8	5.376.056	0.0001%
1000	18	6.050.225	0.0003%

4.3 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.

4.4 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 EPICS Archiver Appliance

5.1 Introdução

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

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

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

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

5.2 Instalação

A versão de Junho de 2016 do *archiver* foi instalada no *OPR23*, que se encontra na sala de controle, e pode ser acessada digitando-se o endereço `10.0.4.69` em qualquer *browser*. Atualmente, o arquivador possui 36 *PVs* conectadas, contendo, inclusive, algumas das variáveis geradas pelos receptadores GPS, descritos na seção 2.3.3. As próximas subseções descrevem algumas das modificações realizadas no *archiver*.

5.2.1 *Login* necessário

Um dos principais problemas do arquivador é que qualquer pessoa logada pode inserir ou remover variáveis do sistema. A fim de impedir que usuários não autorizados realizem tais ações, modificou-se o código das *appliances* para verificar se o usuário foi autenticado com sucesso. Para tal, instalamos um servidor LDAP no OPR23 e criamos uma página de *login*, acessada a partir de `http://10.0.4.69/login.html`. Quando o usuário aperta o botão *Ok*, uma requisição do tipo *POST* é enviada ao módulo *PHP* que roda no servidor. Esse módulo consulta o LDAP e retorna se o usuário foi autenticado ou não. Em caso de sucesso, uma nova requisição do tipo *POST* é enviada e capturada pelo módulo *management*, que, em seguida, inicia uma nova sessão para o usuário. Enfim, antes de qualquer operação de inserção ou remoção de variáveis, o módulo verifica se a sessão está definida e, caso esteja, autoriza a respectiva operação. Nossa intenção é integrar este sistema de *login* ao servidor LDAP do CNPEM no futuro.

Um usuário, denominado de *Anônimo*, com permissões básicas de leitura é disponibilizado por padrão e permite que qualquer pessoa consulte o *archiver* mesmo não estando autenticada.

5.2.2 Mudanças no estilo

Alguns arquivos *css* (*Cascading Style Sheet*) foram modificados para refletir melhor o esquema de cores do laboratório. As imagens do *logo* também foram modificadas.

5.3 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 `pbraw://10.0.4.69/lnls-control-archiver`. 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 `Cnt:MikroE:*`, todas as variáveis arquivadas para o receptor GPS da seção 2.3.3 poderão ser acessadas. Clique com o botão direito na variável desejada e acesse `Process Variable > Data Browser`.

5.4 Acessando a *appliance* com *Python*

A *appliance* pode ser acessada através de requisições *JSON* realizadas por um módulo escrito em *Python*, por exemplo. 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 11 representa o resultado da implementação.

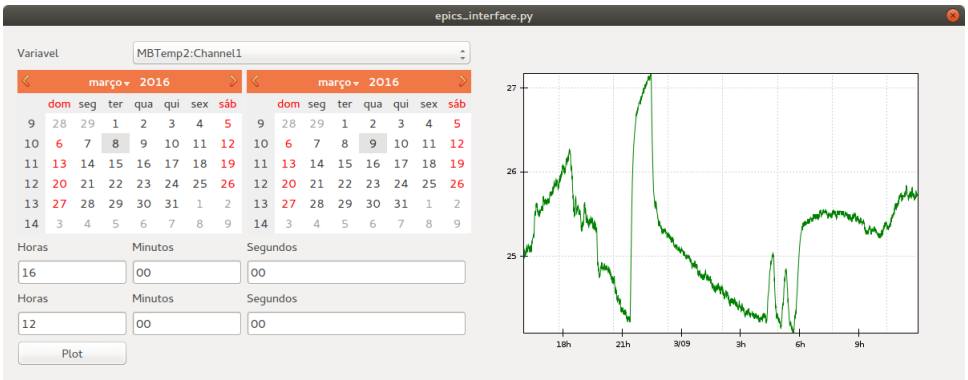


Figura 11: Interface *Qt* implementada em *python*.

6 Best Ever Alarm System Toolkit - BEAST

6.1 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 12, logo abaixo.

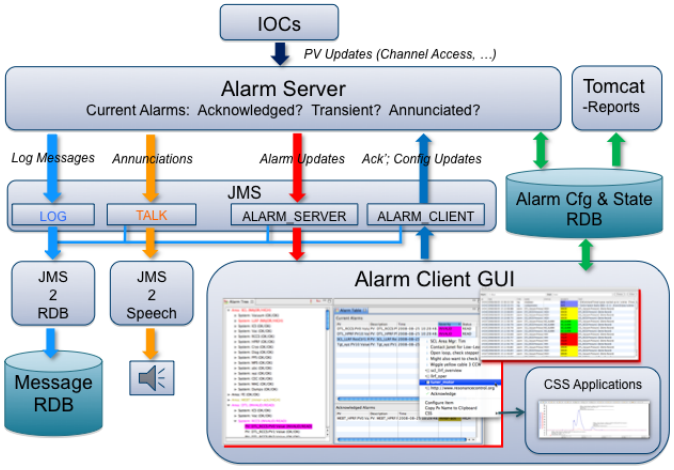


Figura 12: Implementação do sistema de monitoramento de alarmes *BEAST*. Extraída de [3].

É possível distinguir diversos componentes na figura acima:

i. *Alarm server*: o servidor é responsável por tarefas fundamentais no monitoramento de alarmes. Cabe a ele a leitura da configuração dos alarmes armazenada no banco de dados *Alarm Cfg & State RDB*, conectar-se às respectivas variávies, monitorar suas mudanças de estado e gerar alarmes quando necessário e desativá-los logo que um operador toma conhecimento do problema. Esse módulo permite que um número variável de clientes se conecte

a ele. Uma variável pode adotar duas configurações distintas, sendo elas *latch* e *annunciate*. Para a primeira, o servidor mantém o alarme de maior gravidade, mesmo que o estado da variável não seja atualmente errôneo, até que ele seja reconhecido manualmente pelo operador. A fim de impedir um grande volume de alarmes gerados, é possível habilitar as opções *delay*, que aciona o alarme somente se o estado errôneo da variável se mantiver durante o intervalo de tempo especificado, e *count*, que aciona o alarme se tal estado for detectado mais vezes que o valor especificado. A segunda configuração ativa o alarme somente quando a *PV* apresentar valor inválido, sendo que ele é desativado logo que tal variável voltar à sua faixa de operação esperada.

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

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

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

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

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

É possível, a partir de qualquer uma das *views* presentadas acima, acessar outros recursos, como, por exemplo, gráficos e valores atuais das variáveis desejadas. Para isso, basta apertar com o botão direito acima da *PV* e apertar em *Process variables*. A implementação fornece, ainda, suporte para autenticação de usuários via *LDAP* ou *JAAS*. Caso seja a escolha, somente usuários autorizados podem alterar as configurações de alarmes ou reconhecê-los.

6.2 Instalação

BEAST necessita de uma base de dados relacional, de uma implementação *JMS* e de uma instalação do *Eclipse* contendo todos os *plugins* dos quais o servidor de alarmes depende. O último só é necessário para exportar o produto a partir do código fonte. Depois de ter sido exportado, a versão do *Eclipse* pode ser descartada. Utilizamos o *MySQL*, *Apache ActiveMQ* e a versão *Eclipse Luna for RCP and Plugin Development* para os requisitos acima. O servidor de alarmes encontra-se no OPR23, assim como o arquivador.

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

Para utilizar 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`. Se tudo foi configurado corretamente, já é possível acessar os alarmes configurados através do componentes comentados no item iv da seção 6.1. A figura 13 representa alguns alarmes que configuramos no grupo de Controle. Destaca-se a variável `Cnt:MikroE:GPS:Fix`, criada na seção 2.3.3.

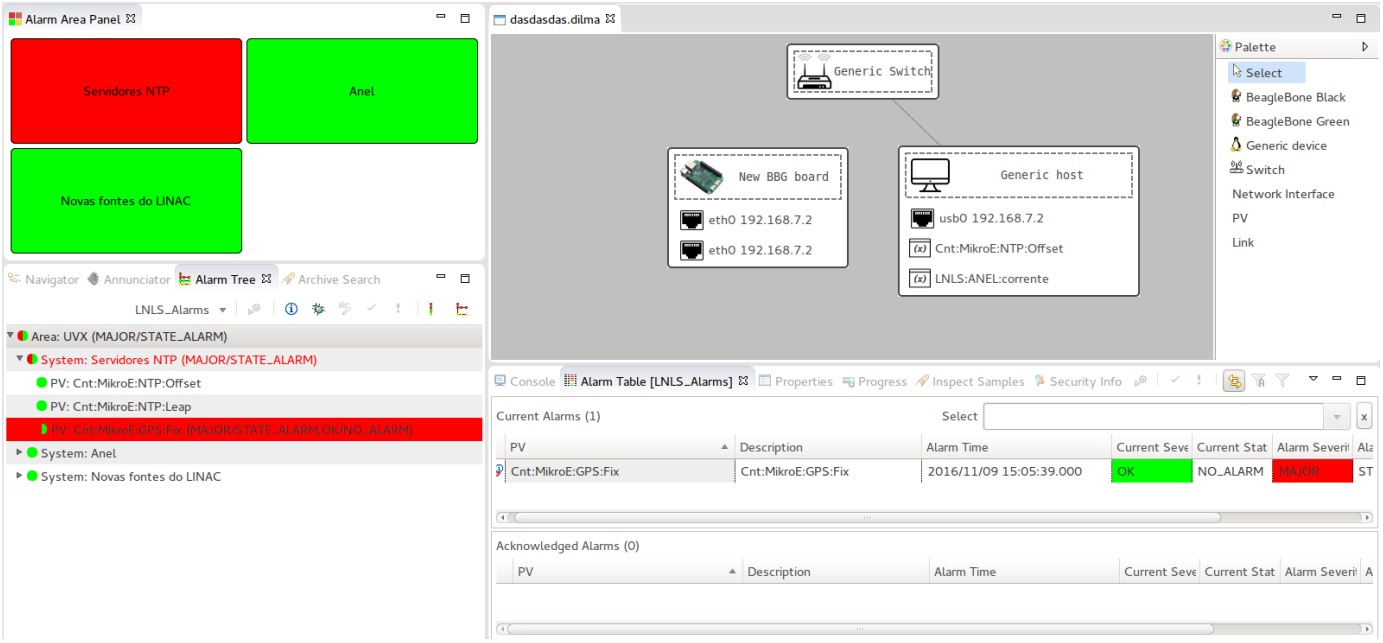


Figura 13: Cliente *BEAST* baseado no *Control System Studio*.

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

O *Eclipse* provê suporte para projetos do tipo *Plugin Project*, específicos para a criação de novos produtos e *plug-ins*. Sendo assim, foi possível exportar a configuração atual do *Eclipse* como um novo produto, que chamaremos de *LNLStudio*. Isso foi realizado através da seleção de apenas os *plugins* que são importantes para o grupo, tais como aqueles ligados ao *BEAST* e ao construtor de interfaces *OPI Builder*. Em relação a outros produtos originados do *Control System Studio*, o *LNLStudio* proporciona uma maior liberdade de configuração e modificação de parâmetros como, por exemplo, a possibilidade de autenticação a um servidor LDAP. Além disso, o último *LNLStudio* foi construído a partir da versão 4.1.4 do CSS, sendo mais recente do que muitos dos demais produtos disponíveis. A figura 13 foi gerada a partir do *LNLStudio*.

7 Conclusão

Durante este ano de estágio, entrei em contato com diversos conteúdos distintos, variando de aplicações de baixo nível até a interfaces gráficas. De maneira geral, as atividades desenvolvidas no grupo de controle contribuíram tanto para meu desenvolvimento técnico quanto ao desenvolvimento pessoal. Julgo também que a orientação de todos do laboratório foi fundamental para que eu pudesse aproveitar ao máximo as atividades propostas. Tive igualmente a oportunidade de aprender conceitos que não foram diretamente dados no meu curso de graduação, muitos ligados à física de operação do acelerador.

Referências

[1] Linuxpps installation. `http://linuxpps.org/wiki/index.php/LinuxPPS_installation`. Acessado em 08 de Novembro de 2016.

[2] Mills D and Martin J. Network time protocol version 4: protocol and algorithm specification. *Request for Comments RFC 5905*, Junho 2010.

[3] Kay Kasemir, Xihui Chen, and Katia Danilova. Best ever alarm system toolkit. `http://www.aps.anl.gov/epics/meetings/2010-10/14/BEAST_EpicsMeeting2010.pdf`. Acessado em 11 de Setembro de 2016.

[4] S. Lescano, A. R. D. Rodrigues, E. P. Coelho, G. C. Pinton, J. G. R. S. Franco, and P. H. Nallin. Uvx control system: An approach with beaglebone black. *Personal Computers and Particle Accelerator Controls*, 2016.

[5] Robert Love. *Linux Kernel Development*. Pearson Education, Inc., 3rd edition, 2010.

[6] Gary E. Miller and Eric S. Raymond. Gpsd time service howto. `http://www.catb.org/gpsd/gpsd-time-service-howto.html`. Acessado em 08 de Novembro de 2016.

[7] Derek Molloy. Writing a linux kernel module — part 1: Introduction. `http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/`. Acessado em 12 de Abril de 2016.