6ª FEIRA DE INICIAÇÃO CIENTÍFICA E EXTENSÃO
FICE
05 e 06 de setembro

UMA INTERFACE DE SOFTWARE PARA A REDE DE SENSORES DO PROJETO ESTUFA

Tiago Possato[1], Mateus Zanini[2], Angelita Rettore de Araujo Zanella[3], Tiago Heineck[4]

INTRODUÇÃO

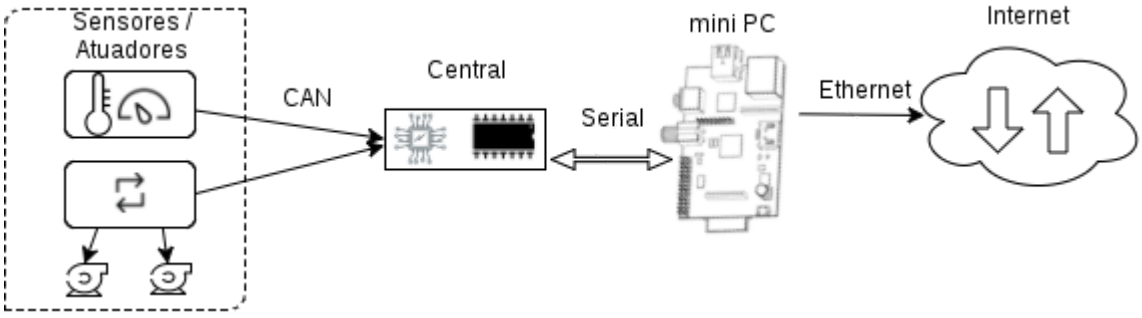
O Projeto Estufa é um projeto do NECTAR (**N**úcleo de **E**studos em **C**omunicações, **T**ecnologias e **A**nálises de **R**ede) do IFC Campus Videira. Tem como objetivo automatizar as estufas agrícolas do campus, permitindo o gerenciamento a partir de qualquer lugar através da Internet. Para isso, está sendo desenvolvida uma rede composta por dispositivos eletrônicos e computacionais que fazem desde a leitura das variáveis ambientais até o acionamento de atuadores, como bombas de irrigação. Nesta rede são utilizados três tipos diferentes de comunicação, sendo eles: (1) Protocolo CAN (***C**ontroller **A**rea **N**etwork*), para comunicação dos dispositivos finais, como sensores e atuadores com uma placa central; (2) Barramento serial UART (***U**niversal **A**synchronous **R**eceiver **T**ransmitter*), para comunicação serial entre a placa central e um mini computador de controle (atualmente é utilizado um Raspberry Pi) e; (3) Ethernet, para ligar o computador de controle com a Internet.

A Figura 01 apresenta uma visão geral da distribuição dos dispositivos neste projeto. Um ponto de destaque é que os dados dos sensores e atuadores podem ser recebidos a qualquer instante. Da mesma forma, o sistema de controle precisa enviar comandos para a rede sem aguardar a liberação de recursos. Sendo assim, o *software* que faz a interface entre a rede da estufa e o resto do sistema

[1] Aluno do IFC – Campus Videira. Curso de Bacharelado em Ciência da Computação.
E-mail: tiago.possato@yahoo.com.br
[2] Aluno do IFC – Campus Videira. Curso de Bacharelado em Ciência da Computação.
E-mail: mateus.zanini.gl@gmail.com
[3] Professora orientadora do IFC – Campus Videira. Curso de Bacharelado em Ciência da Computação.
E-mail: angelita@ifc-videira.edu.br
[4] Professor coorientador do IFC – Campus Videira. Curso de Bacharelado em Ciência da Computação.
E-mail: tiago.heineck@ifc.edu.br

precisa utilizar técnicas de programação paralela com o intuito de prover acesso simultâneo de leitura e escrita na porta serial.

Figura 01: Visão geral da distribuição dos dispositivos do Projeto Estufa



Fonte: Elaboração dos autores (2017).

Atualmente, o sistema funciona através de um *software* escrito na linguagem Python, que recebe e envia mensagens de e para a rede de sensores. Ele também efetua tarefas do sistema de controle, como disparo de alarmes. Na Figura 02 é apresentado o uso de recursos pelo sistema em Python em execução no computador, funcionando com uma única placa remota. É possível perceber que o uso de CPU pelo processo principal (PID 8680) é superior a 20%, justificando a necessidade de melhorar a performance da aplicação.

Figura 02: Uso dos recursos do computador

```
  PID CPU% MEM%   TIME+  Command
 8680 23.7  3.6  1:00.41 |  |- python3 /opt/estufa-central/interface/central/placaBase/central.py
 8720  1.9  3.6  0:04.12 |  |  |- python3 /opt/estufa-central/interface/central/placaBase/central.py
 8719  0.0  3.6  0:00.01 |  |  |- python3 /opt/estufa-central/interface/central/placaBase/central.py
 8716 14.7  3.6  0:34.96 |  |  `- python3 /opt/estufa-central/interface/central/placaBase/central.py
```

Fonte: Elaboração dos autores (2017).

Neste trabalho, o principal objetivo é de aperfeiçoar o desempenho do sistema, reescrevendo a interface de *software* na linguagem de programação C, utilizando recursos como *pthreads*, soquetes UNIX e banco de dados SQLite.

PROCEDIMENTOS METODOLÓGICOS

Para otimizar o *software*, optou-se pela utilização de técnicas de programação concorrente, que faz o uso de *threads* para gerenciar os recursos computacionais. Um *thread* é definido como um fluxo independente de instruções que podem ser executados pelo sistema operacional. Esse conceito pode ser entendido como uma função sendo executada de forma independente do programa principal. Cada *thread* possui sua própria pilha de memória e pode acessar áreas compartilhadas de memória (BARNEY, 2017). A interface de *software* desenvolvida faz uso da biblioteca *pthread* para criar funções independentes, com o objetivo de ler e escrever dados simultaneamente na porta serial do computador.

Quando uma mensagem é recebida da rede de sensores, ela passa por um tratamento inicial, onde são extraídos dados como o cabeçalho da mensagem e o valor que está sendo enviado. Após isso, é inserida em uma fila, podendo ser utilizada pelo resto da aplicação. As mensagens recebidas são armazenadas em um banco de dados SQLite, onde um *thread* específico executa esta ação, esvaziando a fila de entrada.

O SQLite é uma biblioteca GPL (Licença Pública Geral) que implementa um mecanismo de banco de dados SQL (*Structured Query Language*). Diferencia-se da maioria dos outros bancos por não necessitar de um processo exclusivo para o servidor. Além disso, ele lê e grava os dados diretamente em arquivos de disco, não deixando de oferecer suporte a múltiplas tabelas, índices, *triggers* e *views* (SQLITE, 2017?a, tradução nossa).

O SQLite pode ser utilizado em aplicações *multi-thread*, suportando três modos diferentes: (1) *single-thread*, onde as mutex são desabilitadas, fazendo com que o SQLite torne-se inseguro para ser usado em mais de um *thread* por vez; (2) *multi-thread*, no qual “[...] o SQLite pode ser usado com segurança por vários *threads*, desde que nenhuma conexão com o banco de dados seja usada simultaneamente em dois ou mais *threads*” e; (3) serializado, possibilitando que o SQLite seja usado com

segurança por vários *threads*, sem restrição (SQLITE, 2017?b, tradução nossa). O modo padrão, e utilizado neste trabalho, é o serializado.

As solicitações para a rede de sensores são enviadas por outro serviço, de mais alto nível, sendo que a *interface* desenvolvida possui um canal aberto, por onde as mensagens são recebidas e enviadas. Foi utilizado um soquete UNIX para esta finalidade. O *software* desenvolvido recebe as mensagens através deste soquete, insere em uma fila de saída e envia a solicitação pela porta serial.

Foi elaborado um algoritmo para garantir a entrega das mensagens. A implementação consiste em um *thread* que executa uma verificação periódica na fila de saída, re-enviando as mensagens em um intervalo predefinido. Toda vez que uma mensagem é recebida, o *thread* de recebimento de mensagens verifica na fila de saída se existe uma solicitação com a mesma assinatura. Em caso positivo, a mensagem é retirada da fila de saída. As mensagens possuem uma quantidade máxima de tentativas de reenvio, e quando essas tentativas são esgotadas a mensagem é retirada da fila de saída e um registro é armazenado.

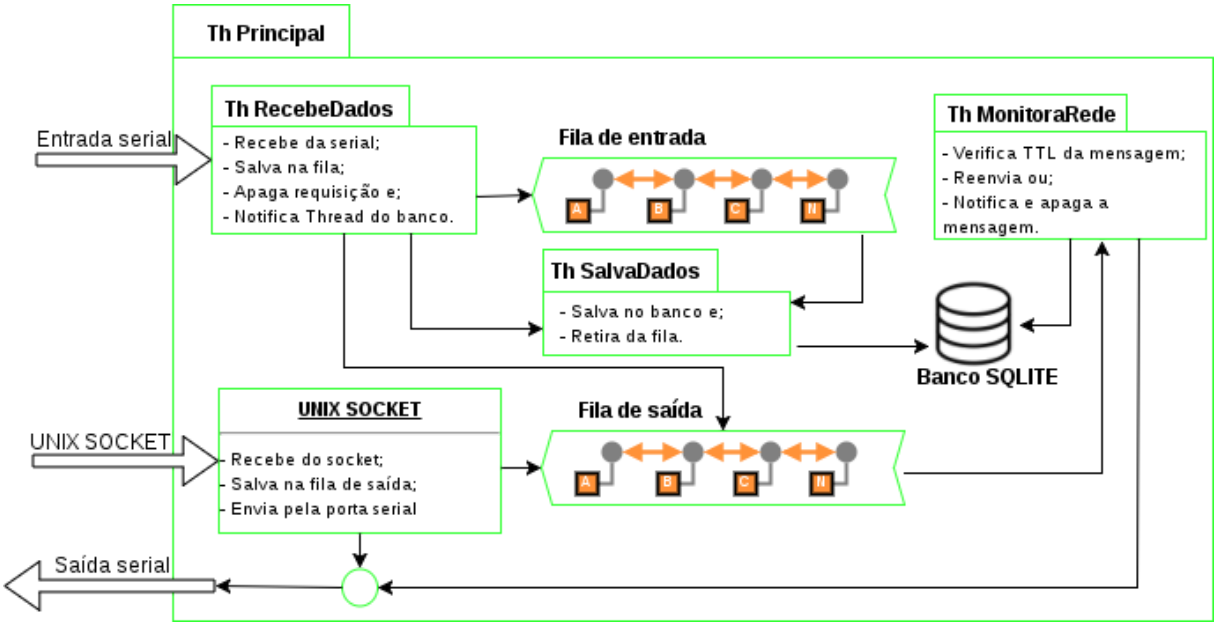
As filas utilizadas no *software* são estruturas de dados do tipo fila duplamente encadeada com cabeçalho. A estrutura do cabeçalho contém uma referência para o primeiro e para o último elemento da fila, um contador com a quantidade de elementos na fila e uma estrutura de um mutex. As regiões críticas das funções que manipulam as filas são protegidas pelo respectivo mutex, que implementa exclusão mútua em *threads*. Um algoritmo de exclusão mútua serve para garantir que regiões críticas de código não sejam executadas simultaneamente, protegendo estruturas de dados compartilhadas de modificações simultâneas (POIANI, 2012).

Segundo Venki (2011), “mutex e semáforos são recursos do *kernel* que fornecem serviços de sincronização”, também conhecidos como primitivas de sincronização. O uso de semáforos restringe o número máximo de usuários que podem consumir simultaneamente um recurso compartilhado. Já o uso de mutex serializa o acesso ao recurso, fazendo com que ele não possa ser executado simultaneamente por mais de um *thread* (WINQUIST, 2005). Um mutex pode ser entendido como um semáforo binário ou de tamanho um.

RESULTADOS E DISCUSSÕES

A implementação da interface de *software* ocorreu conforme a Figura 03. O *software* inicia os *threads* e fica aguardando requisições em um soquete UNIX. Quando uma requisição chega pelo soquete, ela é salva na fila de saída e enviada pela porta serial. Os dados provenientes da rede de sensores são recebidos por outro *thread* através da porta serial. Este, insere os dados na fila de entrada, apaga a requisição na fila de saída e notifica o *thread* do banco de dados, avisando que novos dados chegaram.

Figura 03: Diagrama geral do programa



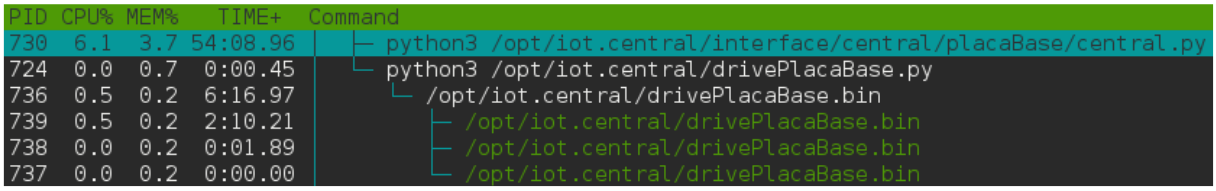
Fonte: Elaboração dos autores (2017).

O *thread* que salva os dados esvazia a fila de entrada e insere cada registro no banco de dados SQlite. De outro lado, o *thread* que monitora a rede é responsável por verificar as mensagens na fila de saída periodicamente, enviando novamente a mesma mensagem. Optou-se por utilizar um *thread* para salvar os dados no banco, pois podem existir demoras nesse processo e alguma possível mensagem que chegue pela porta serial pode ser perdida. A notificação entre o *thread* que recebe

os dados e o que salva no banco é feita utilizando uma variável de condição *pthread_cond_t*.

Para avaliação do desempenho da interface, foi montado um ambiente de testes com as mesmas características do ambiente de produção. A Figura 04 apresenta a utilização dos recursos do computador com o novo *software*. Nota-se que o processo com PID 730, que executa as mesmas tarefas que o processo apresentado na Figura 02 (PID 8680), está consumindo significativamente menos recursos de processamento.

Figura 04: Uso de recursos com a nova interface



Fonte: Elaboração dos autores (2017).

CONSIDERAÇÕES FINAIS

A interface desenvolvida mostrou-se mais eficiente do que a implementação original em Python. Testes realizados em ambiente controlado indicam que a utilização de recursos foi reduzida a níveis aceitáveis para a utilização em computadores com baixa capacidade de processamento e memória.

O algoritmo para reenvio de mensagens fornece mais segurança na comunicação, permitindo que as camadas superiores do controle do Projeto Estufa possam verificar problemas de comunicação e avisar os usuários do sistema.

Para os trabalhos futuros são considerados três pontos, sendo eles: (1) implantação do *software* em modo de produção, possibilitando análise do desempenho e identificação da necessidade de correções; (2) refinamento do algoritmo de reenvio de mensagens e; (3) utilização de recursos nativos do banco de dados, como *triggers*, para efetuar o disparo de alarmes na aplicação.

REFERÊNCIAS

BARNEY, Blaise. **POSIX Threads Programming**. 2017. Disponível em: <https://computing.llnl.gov/tutorials/pthreads/>. Acesso em: 11 maio 2017.

HALL, Brian “Beej Jorgensen”. **Beej's Guide to Unix IPC**. 2015. Disponível em: <http://beej.us/guide/bgipc/output/html/multipage/index.html>. Acesso em: 20 maio 2017.

POIANI, Thiago Henrique. **Mutexes, Monitores e Semáforos**. 2012. Disponível em: <https://pt.slideshare.net/thpoiani/mutexes-monitores-e-semforos>. Acesso em: 09 maio 2017.

SQLITE. **About SQLite**. 2017?a. Disponível em: <http://www.sqlite.org/about.html>. Acesso em: 09 maio 2017.

SQLITE. **Using SQLite In Multi-Threaded Applications**. 2017?b. Disponível em: <http://www.sqlite.org/threadsafe.html>. Acesso em: 09 maio 2017.

VENKI. **Mutex vs Semaphore**. 2011. Disponível em: <http://www.geeksforgeeks.org/mutex-vs-semaphore/>. Acesso em: 09 maio 2017.

WINQUIST, Niclas. **Mutex vs. Semaphore, what is the difference?**. 2005. Disponível em: <http://niclasw.mbnet.fi/MutexSemaphore.html>. Acesso em: 15 maio 2017.