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

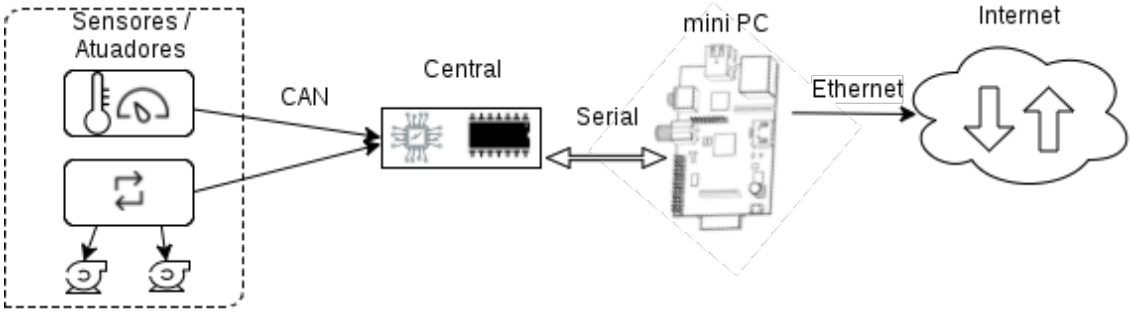
Autores: Tiago POSSATO[1], Mateus ZANINI[1], Tiago HEINECK[2]
Identificação autores: [1]Aluno IFC - Campus Videira; [2] Orientador IFC - Campus Videira.

Introdução

O Projeto Estufa é um projeto do NECTAR (Núcleo de Estudos em Comunicações, Tecnologias e Análises de Rede) 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 tanto, está sendo desenvolvida uma rede composta por dispositivos eletrônicos e computacionais que fazem desde a leitura das variáveis ambientais até o acionamento de bombas de irrigação. Nesta rede são utilizados três tipos diferentes de comunicação, sendo eles: (1) Protocolo CAN (Controller Area Network), para comunicação dos dispositivos finais, como sensores e atuadores com uma placa central; (2) Barramento serial UART (Universal Asynchronous Receiver Transmitter), 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.

Na Figura 01 é apresentada 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 precisa utilizar técnicas de programação paralela com o intuito de prover acesso simultâneo de leitura e escrita na porta serial.

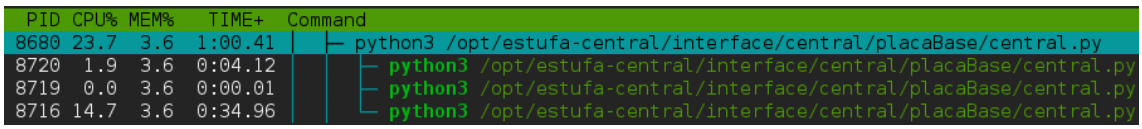
Figura 01: Visão geral da distribuição dos dispositivos do Projeto Estufa



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 dos recursos do computador executando o sistema em Python e funcionando com uma placa remota. É possível perceber que o uso de CPU pelo processo principal (PID 8680) é superior a 20%, justificando a necessidade de buscar alternativas para melhorar a performance da aplicação.

Figura 02: Uso dos recursos do computador



Fonte: Elaboração dos autores, 2017.

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

Material e Métodos

Para otimizar o software, optou-se pela utilização de técnicas de programação concorrente, que faz o uso de *threads* para gerenciar os recursos computacionais.

Uma *thread* é definida como um fluxo independente de instruções que podem ser executadas pelo sistema operacional. Esse conceito pode ser entendido como uma função sendo executada de forma independente do programa principal. Uma *thread* possui sua própria pilha de memória e pode acessar áreas de memória compartilhada (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, sendo que uma *thread* específica 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 pois não necessita 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 uma *thread* por vez; (2) multi-*thread*, no qual "[...] o SQLite pode ser usado com segurança por várias *threads*, desde que nenhuma conexão com o banco de dados seja usada simultaneamente em duas ou mais *threads*" e; (3) serializado, possibilitando que o SQLite seja usado com segurança por várias *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 UNIX, insere em uma fila de saída e envia a solicitação pela porta serial.

Um soquete é um canal de comunicação de duas vias que pode ser usado para comunicação em uma grande variedade de domínios. Os soquetes são amplamente utilizados para comunicação entre processos em máquinas diferentes, utilizando as redes de computadores como canal de troca de mensagens. Um soquete UNIX, por outro lado, é um caminho para um arquivo, onde dois sistemas heterogêneos podem trocar dados em um mesmo computador (HALL, 2015).

Foi elaborado um algoritmo básico para garantir a entrega das mensagens. A implementação consiste em uma *thread* que executa uma verificação periódica na fila de saída, reenviando as mensagens com um intervalo predefinido. Toda vez que uma mensagem é recebida, a *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, sendo que quando essas tentativas são esgotadas, a mensagem é retirada da fila de saída e um *log* 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 uma *thread* (WINQUIST, 2005). Um mutex pode ser entendido como um semáforo binário, ou de tamanho um.

Resultados e discussão

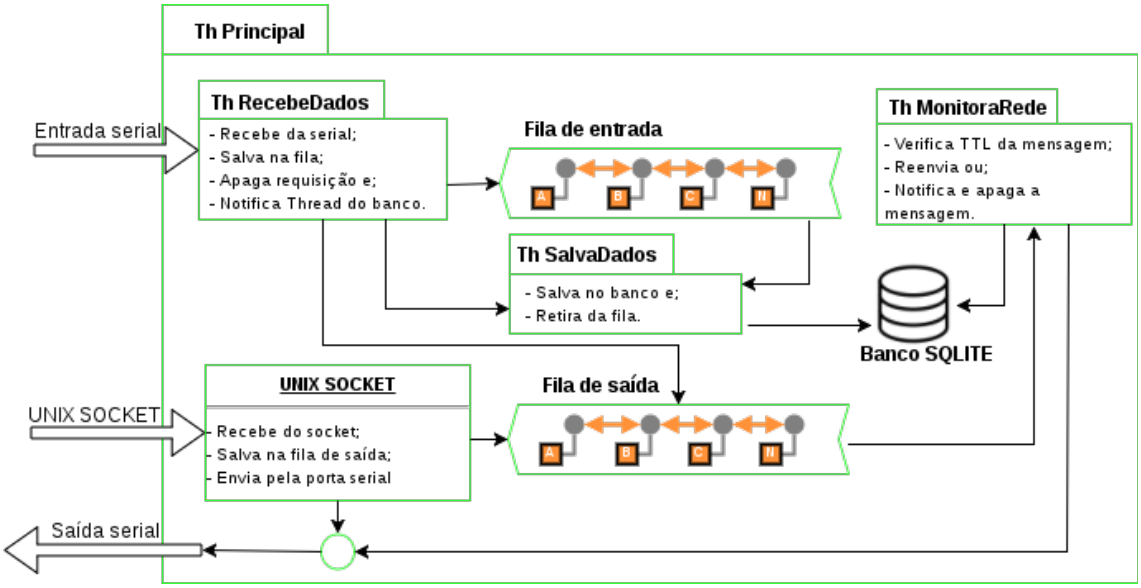
A implementação da interface de *software* ocorreu conforme a Figura 03. O *software* inicia as *threads* e fica aguardando requisições no soquete UNIX. Quando uma requisição chega pelo soquete, ela é salva na fila de saída e enviada pela porta serial. Os dados provenientes da rede de sensores são recebidos por uma *thread* através da porta serial. Ela insere os dados na fila de entrada, apaga a requisição na fila de saída e notifica a *thread* do banco de dados de que uma nova mensagem foi recebida. A notificação é feita utilizando uma variável de condição *pthread_cond_t*.

As variáveis de condição são recursos que permitem a sincronização entre *threads* através de um valor global. Sem a utilização destas variáveis, as *threads* necessitariam verificar se uma dada variável tem um valor específico, consumindo

recursos constantemente. Uma variável de condição obtém os mesmos resultados sem a necessidade de estar sempre verificando o valor (SOUZA, 2007?).

A *thread* que salva os dados no banco esvazia a fila de entrada e insere cada registro no banco de dados SQlite. De outro lado, a *thread* que monitora a rede verifica as mensagens na fila de saída periodicamente, enviando novamente a mesma mensagem.

Figura 03: Diagrama geral do programa



Fonte: Elaboração dos autores, 2017.

Conclusão

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. 2017. **POSIX Threads Programming** (https://computing.llnl.gov/tutorials/pthreads/). Acesso: 11/05/17.

HALL, Brian “Beej Jorgensen”. 2015. **Beej's Guide to Unix IPC (**http://beej.us/guide/bgipc/output/html/multipage/index.html). Acesso: 20/05/17.

POIANI, Thiago Henrique. 2012. **Mutexes, Monitores e Semáforos (**https://pt.slideshare.net/thpoiani/mutexes-monitores-e-semforos). Acesso: 09/05/17.

SOUZA, Orlando. 2007?. **Threads** (http://www.dei.isep.ipp.pt/~orlando/so2/threads.htm). Acesso: 22/05/17.

SQLITE. 2017?a. **About SQLite** (http://www.sqlite.org/about.html). Acesso: 09/05/17.

SQLITE. 2017?b. **Using SQLite In Multi-Threaded Applications** (http://www.sqlite.org/threadsafe.html). Acesso: 09/05/17.

VENKI. 2011. **Mutex vs Semaphore** (http://www.geeksforgeeks.org/mutex-vs-semaphore/). Acesso: 11/05/17.

WINQUIST, Niclas. 2005. **Mutex vs. Semaphore, what is the difference?** (http://niclasw.mbnet.fi/MutexSemaphore.html). Acesso: 15/05/17.