UNIVERSIDADE FEDERAL DE SANTA CATARINA
Centro Tecnológico
Curso de Pós-Graduação em Ciência da Computação

DISSERTAÇÃO

UM MODELO DE APLICATIVO TUNELADOR DE DADOS SERIAIS SOBRE INFRA-ESTRUTURA TCP/IP

Fabrício Nees

João Bosco da Mota Alves
Orientador

Florianópolis, Maio de 2001

PÁGINA DE APROVAÇÃO

UM MODELO DE APLICATIVO TUNELADOR DE DADOS SERIAIS SOBRE INFRA-ESTRUTURA TCP/IP

Fabrício Nees

Esta Dissertação (Tese) foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação Área de Concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

_____

Fernando Álvaro Ostuni Gauthier, Dr.

Banca Examinadora

_____

João Bosco da Mota Alves, Dr.

_____

Luiz Fernando J. Maia, Dr.

_____

Malcon Tafner, Dr. (FURB)

RESUMO

A finalidade deste trabalho é apresentar um modelo de aplicativo capaz de trafegar dados de e para dispositivos seriais através de uma infra-estrutura de rede baseada nos protocolos TCP/IP, e mais especificamente no protocolo de rede IPv4 e de transporte TCP. Tal capacidade é normalmente conhecida como tunelamento. Os dados seriais a serem enviados podem ser representados de diferentes formas, escolhida quando de sua utilização, podendo ser essencialmente dados seriais no padrão RS-232, assim como estruturas de dados mais complexas, que serão traduzidas no aplicativo para dados seriais puros, que serão por sua vez enviados ao dispositivo serial.

O aplicativo é composto de um módulo de gerência, o qual compõe o servidor, e um módulo de tunelamento responsável pelo tráfego dos dados. Para validar a proposta foram desenvolvidos dois aplicativos separados, um para cada módulo.

ABSTRACT

Our goal at the present work is to present a software model which is capable of transmitting data to and from serial devices over a TCP/IP based network infrastructure, more precisely a IPv4 network and TCP transport protocol based infrastructure. Such a capability is usually known as tunneling. Serial data can be represented in several ways, chosen at the time of its use. It could most often be essentially RS232 standard, as well as more complex data structures, which will be translated at application time to pure serial data, which, in turn, will be turned to the serial device.

The software has a two modules approach. One module is the manager module, also known as the server module. Another one is named the tunneling module, which is the one responsible for the tunneling process itself.

SUMÁRIO

Lista de Figuras

1 INTRODUÇÃO

1.1 APRESENTAÇÃO DO TEMA E JUSTIFICATIVA

Com o advento da Internet e sua disponibilização no Brasil a partir da metade dos anos 90, várias foram as aplicações surgidas. Em um primeiro momento, dentro de universidades e centros de pesquisa, mas logo ganhando espaço dentro do setor de produção nacional, com profundos reflexos na vida de muitos cidadãos.

Entre as várias possibilidades de aplicações, uma nos parece de grande valia e de futuro promissor. Um grupo de estudos da Universidade Federal de Santa Catarina acabou por introduzir um novo conceito, a **Experimentação Remota** (WISINTEINER, 1999).

Uma das dificuldades encontradas nos estudos de experimentação remota, no que tange ao atendimento de demanda, está justamente no gargalo apresentado por sistemas (como nos casos atuais) monousuário. Alternativas há para resolver o problema. Pelo menos quanto à concepção e desenvolvimento de uma arquitetura de computação multiusuária. Em outras palavras, vários dispositivos estão, hoje, configurados com arquitetura mono-usuária, ou seja, quando um usuário usa um recurso, só a ele é possível esse uso. Torná-lo multiusuário, portanto, é de fundamental importância para uma maior racionalização de recursos.

Como podemos ver, a experimentação remota é uma verdadeira janela para o futuro, possibilitando o uso de equipamentos conectados a servidores remotos. Esses equipamentos podem ser simples, mas sempre com tendência de aumento de complexidade. Em sua grande maioria conectam-se a porta serial, pois é o padrão mais fácil de usar e principalmente o mais difundido.

Como dificuldade maior, está o caráter repetitivo de um servidor do equipamento e um cliente que devem comunicar-se através da internet usando TCP/IP. Quando da disponibilização de novas funções do dispositivo, pelo processo servidor, não só o protocolo

precisa ser revisto, mas também as características da conexão TCP, o que leva a mudanças no código de comunicação. Do lado do servidor, as funções devem ser implementadas para garantir a transmissão das novas funções ao equipamento.

Seria muito mais fácil se a preocupação se limitasse à inclusão das funções a serem disponibilizadas e da representação das mesmas no protocolo. Assim como seria melhor se pudéssemos ter acesso a todas as funções sem a necessidade do servidor conhecê-las primeiro. Uma solução para esse problema é especificar e implementar um aplicativo servidor de porta serial. Este trabalho pretende generalizar a comunicação entre o cliente e o equipamento sendo "servido", através de uma forma de generalização, acima tratada como um “aplicativo servidor de porta serial”.

Por ter uma função básica de trafegar dados seriais entre um dispositivo serial e um cliente remoto, que está conectado através de uma porta TCP, esse sistema, objeto principal deste trabalho, recebeu o nome de **tunelador** (**SerialPipe**) pois, a princípio, nem o cliente nem o periférico precisam ficar sabendo detalhes sobre o processo de transmissão dos dados.

Em verdade, o periférico não toma conhecimento do cliente remoto, e tudo que o cliente remoto precisa saber é que deve mandar seus dados para a porta especificada que o periférico vai recebê-los. O aplicativo SerialPipe é capaz de receber e enviar dados para uma porta TCP qualquer, uma serial qualquer existente no sistema (hw/so), um arquivo qualquer (fd) ou um console.

Sabendo de onde e para onde deve enviar os dados, ficará esperando em uma entrada e escrevendo na outra. Poderá ainda fazer as duas coisas simultaneamente, ou em turnos, ou ainda várias delas ao mesmo tempo. Por isso poderá desde simplesmente repassar dados seriais vindos de uma porta TCP para uma serial, até tomar decisões padrão quando controlando um robô complexo (por exemplo, que o robô espere uma confirmação após avisar que completou uma tarefa).

1.2 OBJETIVOS DA PESQUISA

1.2.1 Objetivo Geral

A principal meta desse trabalho é especificar e validar um modelo de aplicativo capaz de transmitir dados seriais encapsulados em uma infra estrutura baseada nos protocolos TCP/IP.

1.2.2 Objetivos Específicos

Como objetivos específicos esse trabalho apresenta:

a) Estudar as bases da experimentação remota;

b) Estudar o conceito de tunelamento;

c) Aprofundar os conhecimentos nos protocolos TCP/IP;

d) Especificar um modelo de aplicativo tunelador;

e) Implementar o modelo proposto.

1.3 ORGANIZAÇÃO DO TRABALHO

Esse documento está assim organizado. O capítulo 1 é esta introdução. A seguir, no capítulo 2, tratar-se-á dos componentes envolvidos no projeto, pois essa fundamentação teórico-prática é essencial para a compreensão do sistema proposto: o protocolo TCP/IP, a porta serial e os princípios de tunelamento. No capítulo 3 será feita a especificação do aplicativo a ser desenvolvido como parte principal desse projeto. No capítulo 4, detalhes da implementação, e limitações auto-impostas para a efetivação da implementação em tempo hábil. No capítulo 5, nossas conclusões e objetivos a serem alcançados com o permanente aprimoramento do aplicativo.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 REXLAB

Em 1996, o Prof. João Bosco da Mota Alves, ministrando uma disciplina sobre Automação de Processos no Curso de Pós-Graduação em Ciência da Computação - CPGCC - da Universidade federal de Santa Catarina - UFSC, propôs um desafio como trabalho para a turma. Tratava-se de fazer-se um estudo sobre os requisitos necessários ao projeto e desenvolvimento de um sistema que permitisse a realização de experimentos remotamente.

O entusiasmo com o qual os estudantes se dedicaram a tal empreitada foi determinante nas discussões e, posterior elaboração da proposta de criação de um laboratório que pudesse ser uma espécie de alavanca para tais experimentos. Não apenas o laboratório foi criado, o RExLab, como da idéia original resultou em uma dissertação de mestrado de Miguel Wisinteiner (WISINTEINER, 1999), com a inserção de um projeto piloto, ao qual foram adicionados mais dois outros semelhantes, hoje em funcionamento e que continuam a sofrer aperfeiçoamento.

Ampliando a dimensão de utilização da Internet, o RExLab - Laboratório de Experimentação Remota, cujo endereço de seu portal é http://rexlab.inf.ufsc.br representa, hoje, um poderoso instrumento de Pesquisa e Desenvolvimento (P&D), com vistas a aplicações relevantes em todas as áreas do conhecimento onde se faça necessária a monitoração e intervenção (controle).

Devido a grande demanda apresentada pelos usuários do RExLab e, principalmente, pelo fato da idéia ter ganho um mundo de aplicações não imaginadas em sua concepção, houve que se construir um portal para a difusão da informação. Não apenas as geradas no próprio RExLab, mas que o mesmo se fizesse presente como ferramenta de auxílio à produção do conhecimento e difusão da informação resultante, traduzidas em socialização do saber, em

geral. Em seu portal, três grandes tópicos são contemplados (podendo ainda ter seu universo ampliada, dependendo da demanda observada doravante, a saber:

a) **Sistemas de Computação e Robôs Inteligentes**

b) **Sistemas de Conhecimento**

c) **Acessibilidade e Tecnologias**

No primeiro, estão concentradas as informações referentes aos equipamentos e sistemas que envolvem computação e, em particular, as pesquisas e desenvolvimentos na área de Robôs Inteligentes, ou robôs que emulam comportamentos os quais seriam considerados inteligentes, caso fossem exibidos por animais.

No segundo, são disponibilizadas as informações que se reportam a áreas cujas pesquisas são consideradas Interdisciplinar, Multidisciplinar e Transdisciplinar. A rigor, é o RExLab a serviço de estudos e pesquisas que demandam a intervenção de várias disciplinas, como por exemplo, o caso da questão energética brasileira, a ocupação humana, a violência urbana e rural, o empreendedorismo como forma de organização e emancipação de comunidades carentes, a inserção social, o aprender a aprender (treinamento de fundamentos ou habilidades, necessários à filosofia) no ensino fundamental, etc.

Finalmente, o terceiro, que é dedicado a pessoas portadoras de necessidades especiais (antigamente conhecidas como pessoas portadoras de deficiência) que, como se sabe, além do preconceito generalizado, sofrem por terem pouca importância no mundo científico e tecnológico. O RExLab tem esse tópico como de igual importância aos outros dois.

Como recurso educacional, o RExLab desenvolve ferramentas de auxílio à educação e treinamento, que podem ser utilizadas em, praticamente, todas as áreas do conhecimento, permitindo a estudantes buscar informações no mundo real a partir de um computador remoto. Este fato introduziu este novo conceito, **Experimentação Remota** (WISINTEINER, 1999), o qual é uma extensão do acesso remoto e, diferentemente deste, permite que ações sejam executadas em dispositivos externos ao computador que está sendo acessado, ainda que controlados pelo mesmo.

O RExLab é, portanto, um sistema combinado de computadores (Servidores) e dispositivos externos (a serem monitorados e controlados), que permitem ao usuário remoto (Cliente) monitorar e controlar, em princípio, qualquer dispositivo conectado ao computador, como se o usuário estivesse na presença do mesmo. Podem ser conectados ao Servidor os mais diversos tipos de dispositivos (sensores, atuadores, etc.), e mesmo equipamentos de teste (osciloscópios, geradores de função, multímetros etc.), permitindo que o Cliente, ao conectar-se ao Servidor, execute ações de monitoração e de controle sobre os mesmos.

Como ilustração da utilização do modelo RExLab - Laboratório de Experimentação Remota, a Figura 1 indica, de forma generalizada, sua arquitetura de computação.

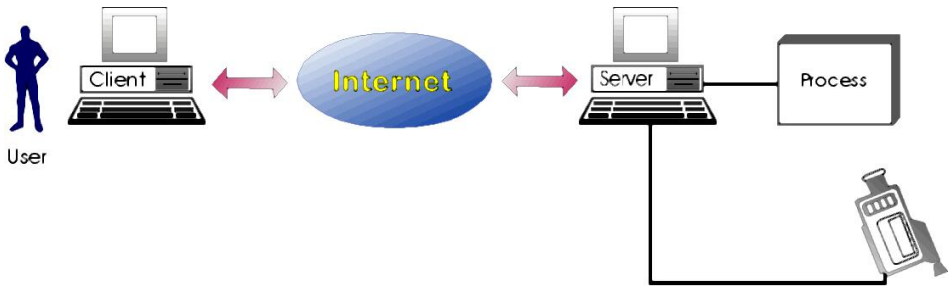

Figura 1 : Arquitetura básica do RExLab - Laboratório de Experimentação Remota

Os principais benefícios da utilização do Laboratório de Experimentação Remota no meio acadêmico são:

Maior utilização da rede de recursos disponíveis nas universidades brasileiras, públicas ou privadas, beneficiando, antes de tudo, a formação de profissionais com a qualificação demandada pelos desafios do novo milênio;

O mesmo permite que os usuários (professores, alunos e servidores, das instituições, e também qualquer pessoa com acesso a Internet) tenham acesso a recursos que os mesmos não dispõem, proporcionando a um grande número de pessoas a realização de experimentos reais de baixo custo, pois oferece o compartilhamento dos recursos, ampliando significativamente sua utilização.

Em princípio, duas instituições com equipamentos diferentes podem disponibilizá-los para uso nas duas instituições, por seus estudantes e pesquisadores, além do público, em geral. Evidentemente, que essa possibilidade irá depender do tipo de equipamento: se o mesmo pode

ter uso simultâneo por clientes distintos. Mesmo neste caso, a restrição é do equipamento e não do RExLab.

Experimentos reais podem ser realizados de qualquer lugar e a qualquer hora, desde que se tenha um acesso à Internet e depende, como dito, do tipo de recurso a ser disponibilizado.

Além disso, o desenvolvimento da tecnologia de Laboratórios de Experimentação Remota nos permite vislumbrar, em um futuro próximo, novas utilizações para a Internet, além das já conseguidas como, por exemplo:

- Exploração de lugares inacessíveis a seres humanos, tais como vulcões, oceanos e outros planetas.

- Manutenção, à distância, de máquinas equipamentos, em tempo menor e com menores custos.

As possibilidades expostas justificaram a criação de um projeto piloto de um Laboratório de Experimentação Remota (WISINTEINER, 1999), que é a integração de clientes com dispositivos no LabReX.

A experiência piloto tinha por objetivo, além de proporcionar contato com experimentação remota, permitir ao usuário executar um programa em linguagem Assembly para o microcontrolador da família 8051, por ele (cliente) desenvolvido. Trata-se, portanto, de um laboratório para complementar o ensino de microcontroladores, assunto seguramente presente no currículo da maioria dos cursos de Engenharia Elétrica/Eletrônica do país, além de presente no currículo de muitos cursos técnicos afins e de ser extremamente útil para profissionais das áreas de Engenharia de Produção e Ciência da Computação.

O Laboratório de Experimentação Remota (http://www.rexlab.inf.ufsc.br) é composto de:

1. Uma placa contendo o microcontrolador 8051 e outros componentes periféricos que permitem a comunicação do mesmo com o PC através de serial RS-232;

2. Um programa servidor (Lab-Rem-Servidor), que recebe informações do cliente (Lab-Rem-Cliente), as repassa ao 8051 pela porta serial e retorna ao cliente a reposta solicitada;

3. Um programa cliente (Lab-Rem-Cliente), que carrega o código binário do programa do usuário, o transfere ao servidor (Lab-Rem-Servidor) para ser executado e permite ao usuário solicitar a resposta que desejar;

O aluno que está estudando o microcontrolador da família 8051 pode, a partir do Lab-Rem-Cliente, conectar-se ao Lab-Rem-Servidor, carregar o programa que deseja testar e enviá-lo ao Servidor. O Servidor repassará o programa ao microcontrolador 8051, que o executa. O usuário tem, então, acesso aos resultados (quase todos os registradores e as posições de memória interna de 32 a 127), através do Lab-Rem-Cliente.

Estudantes de microcontroladores podem, portanto, fazer experiências práticas com o microcontrolador 8051 mesmo sem dispor do componente. É importante salientar que trata-se de experiência real, não de simulação.

Assim como esse projeto piloto, outros já foram concluídos e estão em constante aprimoramento. Podemos citar:

- Bas52Rem - sistema que permite execução da linguagem BASIC-52 remotamente;

- For52Rem - sistema que permite a execução de um interpretador Forth para a família 8051 remotamente;

- RExPIC - sistema que sistema que permite ao usuário escrever um código fonte, em linguagem assembly, para o PIC, enviar este código para ser executado remotamente e ter acesso ao resultado da porta B através de imagem de LEDs conectados a esta porta. A imagem é capturada por uma câmara conectada à porta paralela do PC servidor e é acionada a cada 5 segundos.

- Microservidor - Dispositivo de baixo custo com a finalidade de baixar os custos das aplicações do RExLab (ainda em fase de desenvolvimento)

2.2 O PROTOCOLO TCP/IP

O Transport Control Protocol / Internet Protocol (TCP/IP) é o padrão de fato das redes de comunicação de dados nos dias de hoje. Desenvolvido inicialmente no Departament of Defense dos EUA, mais especificamente na Advanced Research Projects Agency (ARPA), para interligar diferentes redes de computadores que surgiram para otimizar recursos, mas não conversavam entre sí. (MOSCHOVITIS et al., 1999). A partir de 1984 passou a fazer parte integrante do sistema UNIX, onde era utilizado não apenas nas comunicações inter computadores mas também para comunicação entre processos do mesmo sistema.

Como protocolo de comunicação entre computadores, o TCP/IP logo tornou-se padrão para comunicação de longa distancia, devido ao grande uso de sistemas UNIX nas primeiras redes WAN, feitas principalmente entre universidades e órgãos governamentais americanos. Com a explosão da Internet, que pelos mesmos motivos (a presença do UNIX na maioria dos nós) adotou o TCP/IP como padrão, o protocolo popularizou-se também para redes locais.

Rapidamente ultrapassou outros protocolos para LAN, quase todos proprietários (vendor-specific) como IPX/SPX, Banyan VINES, NETBEUI, etc., assim como os usados em ambiente mainframe, devido a redução no número destes. Como a definição do TCP/IP é pública (através dos RFCs) e a implementação (em UNIX) tem seu código aberto, todos os vendedores de software de rede acabaram por incorporá-lo a seus produtos.

O termo TCP/IP, usado em um contexto mais amplo, refere-se a um conjunto grande de protocolos, que inclui protocolos que vão desde a fronteira de acesso ao meio até a camada de aplicação. Já a sigla TCP/IP faz na verdade referência a dois protocolos de camadas diferentes. O IP trabalha principalmente na camada de rede (do modelo OSI) e é praticamente onipresente nas redes que usam TCP/IP. Já o TCP trabalha na camada de transporte e é um protocolo orientado a conexão, com sua contraparte não orientada a conexão sendo a UDP.

O UDP coloca as informações que quer enviar na área de dados do protocolo e preenche o cabeçalho com os dados necessários. Entre eles o IP de origem e o IP de destino. Então simplesmente submete o pacote a camada IP, esperando que chegue ao destino. Controle de recepção, assim como resposta a uma eventual solicitação virão em outro pacote UDP controlado pela aplicação.

Já o TCP envia um datagrama pedindo uma conexão e aguarda uma resposta na mesma porta de origem, que, assim como todos os datagramas posteriores dessa conexão virá com o bit ACK setado. Daí para frente, solicitações e respostas serão controladas pelo aplicativo dentro da conexão já estabelecida e controlada pela camada de transporte (TCP).

Essas considerações, apesar de básicas, são importantes para analisarmos nossas escolhas e a segurança nelas contidas.

2.3 A PORTA SERIAL

2.3.1 Definição

Um dispositivo de entrada e saída é apenas uma forma de transmitir dados de e para o computador. Existem muitos tipos de dispositivos de e/s como: portas seriais, portas paralelas, controladores de HD, placas de rede, USB, etc. A maioria dos PCs têm uma ou duas portas seriais, cada uma tem um conector de nove pinos (algumas vezes vinte e cinco). Os programas podem enviar dados (Bytes) para o pino de transmissão e receber (Bytes) do pino de recepção. Os outros pinos são para terra e controle.

Muito além de conectores, as portas seriais convertem dados de paralelo para serial e mudam a representação elétrica dos dados. Dentro de um computador os bits de dados fluem em paralelo (usando vários fios ao mesmo tempo). O fluxo serial é uma seqüência de bits sobre um único fio (como no pino de conexão e recepção do conector serial). Para a porta serial criar tal fluxo ela deve converter dados de paralelo (dentro do computador) para serial no pino de transmissão (e vice e versa).

A maioria dos componentes eletrônicos de uma porta serial é encontrada em um chip ou parte de um chip, conhecido como UART.

2.3.2 Pinos e Fios

Os PCs antigos usavam conectores de 25 (vinte e cinco) pinos, mas apenas 9 (nove) dos pinos eram realmente utilizados. Assim a maioria dos computadores hoje usa apenas conectores de 9 pinos. O número mínimo de fios para usar em transmissões de duas vias é 3 (três), um para transmissão, outro para recepção de dados e um como terra. A voltagem de qualquer fio é medida de acordo com esse terra. Os outros fios são usados apenas para propósito de controle (sinalização) e não para enviar bytes. Para cada um destes sinais há um fio dedicado a ele. Alguns (ou todos) desses fios de controle são chamados "linhas de controle de modem". Fios de controle de modem estão ou no estado positivo de +12 volts ou no estado negativo de –12 volts. Um desses fios é para sinalizar o computador para parar de enviar bytes para a porta serial. De outro lado, outro fio sinaliza o dispositivo conectado à porta serial para parar de enviar bytes para o computador.

Se o dispositivo conectado é um modem outros fios podem dizer ao dispositivo para desligar o telefone ou dizer ao computador que uma conexão foi feita ou que o telefone está tocando.

2.3.3 RS 232

A porta serial (que é diferente da porta USB) é normalmente uma RS 232-C, EIA 232-D ou EIA 232. Sendo que, essas três, são praticamente a mesma coisa. O prefixo original RS (Recommended Standard – Padrão Recomendado) transformou-se em EIA (Eletronics Industries Association – Associação das Indústrias Eletrônicas) e depois EIA/TIA quando a EIA juntou-se com a TIA (Telecommunications Industries Association – Associação das Indústrias de Telecomunicações). O padrão EIA 232 provê, também, especificação para comunicação síncrona, mas o hardware para dar suporte a comunicações síncronas dificilmente existe no PC.

2.3.4 Endereços de I/O e IRQs

Como o computador precisa se comunicar com cada porta serial, o sistema operacional precisa saber que cada porta serial existe e onde ela está (seu endereço I/O). Ele também precisa saber qual o fio (número de IRQ) a porta serial deve usar para requerer serviço da CPU do computador. Ela requer serviço enviando uma interrupção nesse fio assim cada dispositivo de porta serial deve gravar em sua memória não volátil tanto o endereço de I/O quanto o seu IRQ.

Com o barramento PCI não funciona exatamente assim, pois o mesmo tem seu próprio sistema de interrupções. Mas como as BIOS de PCIs designam chips para mapear essas interrupções PCIs para IRQ elas se comportam exatamente como descrito acima exceto que o compartilhamento de interrupções é permitido.

Endereços de I/O não é o mesmo que endereços de memória. Quando um endereço de I/O é colocado no barramento de endereço do computador outro fio é energizado, isso avisa tanto a memória principal para ignorar esse endereço como avisa os dispositivos que tem endereço de I/O (como portas seriais) que devem ouvir o endereço para ver se corresponde ao seu. Se o endereço corresponde então o dispositivo lê os dados no barramento de dados.

2.3.5 Interrupções

Quando a porta serial recebe um número de bytes (pode ser setado para 1, 4, 8 ou 14) no seu buffer FIFO ela sinaliza a CPU para recebê-los, enviando um sinal elétrico, conhecido como interrupção, em um determinado fio, normalmente usado apenas por essa porta. Assim o FIFO espera por um número de bytes e então envia uma interrupção, no entanto essa interrupção também será enviada se houver uma demora inesperada para a chegada do próximo byte (conhecido como time out).

Assim se os bytes estão sendo recebidos vagarosamente (como quando alguém estiver digitando em um teclado de um terminal) pode haver uma interrupção enviada para cada byte recebido. Para alguns chips UART a regra é desta forma: se quatro bytes seguidos podem ser recebidos mas nenhum desses quatro aparecem, então a porta desiste de esperar por estes

bytes e envia uma interrupção para a CPU pegar os bytes que estão correntemente na FIFO. É claro que se a FIFO estiver vazia nenhuma interrupção será mandada. Cada condutor de interrupção dentro do computador tem um número identificador de IRQ e a porta serial deve conhecer cada condutor para o qual deve sinalizar.

Interrupções são enviadas quando a porta da serial precisa da atenção da CPU. É importante fazer isso de alguma forma no devido tempo, pois o buffer dentro da porta serial pode suportar apenas 16 (dezesseis) bytes de entrada (um nas portas mais antigas). Se a CPU falhar para remover tais bytes recebidos de imediato, então não haverá nenhum espaço livre para qualquer byte recebido a mais e o pequeno buffer poderá sofrer overflow.

Interrupções contêm um monte de informação. Indiretamente a interrupção por si só avisa o chip controlador de interrupção que certa porta serial precisa de atenção. O controlador de interrupção então sinaliza a CPU e a CPU roda um programa especial para servir aquela porta serial. Esse programa é chamado de rotina de serviço de interrupção (parte do software drive de serial).

Ela tenta descobrir o que aconteceu com a porta serial e então resolve o problema de como transferir bytes de ou para o buffer de hardware da porta serial. Este programa pode facilmente descobrir o que aconteceu, pois a porta serial tem os seus próprios registradores de endereço de I/O conhecidos pelo software drive de serial. Estes registradores contêm informação de status sobre a porta serial, o software lê estes registros e por inspecionar o seu conteúdo descobre o que aconteceu e toma a ação apropriada.

2.4 O PROTOCOLO SERIAL

2.4.1 Fluxo de Dados

Dados (bytes que representam letras, figuras, etc.) fluem de e para a porta serial. Esta é a forma mais comum que a unidade central de processamento usa para transferir dados para seus periféricos e para outros equipamentos em geral. O aspecto que recebe a maior atenção do usuário comum neste fluxo de dados é a velocidade. As velocidades de fluxo (como 56k) são incorretamente entendidas como velocidade média, mas porque são usualmente citadas apenas como velocidade ao invés de velocidade máxima de fluxo instantâneo. É importante entender que a velocidade média é normalmente menor que a velocidade especificada. Esperas (ou tempo IDLE) resultam numa velocidade média menor, essas esperas podem incluir longas esperas ou apenas um segundo devido ao controle de fluxo. Num outro lado pode haver pequenas esperas de vários microssegundos entre os bytes. Se o dispositivo conectado, direta ou indiretamente, a porta serial não pode aceitar a velocidade total da porta serial então a velocidade média deve ser reduzida. Veremos a seguir como.

2.4.2 Controle de Fluxo

Controle de fluxo significa a habilidade de regular o fluxo de bytes em um conector de um fio ou barramento. Ele inclui a possibilidade de parar e de recomeçar o fluxo sem a perda de bytes. O controle de fluxo é necessário para permitir uma mudança na velocidade de fluxo instantâneo de e para equipamentos conectados à porta serial.

Exemplo de controle de fluxo é o caso aonde conectamos um modem externo de 33.600 bps através de um pequeno cabo (o padrão RS232 limita o tamanho deste cabo) na porta serial. Quando o modem envia e recebe bytes sobre uma linha telefônica de 33.6 Kbits/s (BBS) ele não está fazendo qualquer compressão de dados ou correção de erros. Mas podemos configurar a velocidade da porta serial para 115 200 bits/s, e estaremos enviando dados do computador para a linha telefônica nesta velocidade.

Então o fluxo do computador para o modem sobre o pequeno cabo é de 115.200 bps e, no entanto, o fluxo do modem para a linha telefônica é de apenas 33.600 bps. Como um fluxo mais rápido (115.2) está indo para o modem do que aquele que está saindo dele, o modem está guardando o fluxo em excesso (81.6 Kbps aproximadamente) em um de seus buffers. Este buffer vai eventualmente sobrecarregar e estourar (sofrer um *overflow*) a não ser que a velocidade de 115.200 seja diminuída.

É então que aparece o controle de fluxo. Quando o buffer do modem está praticamente cheio o modem envia um sinal de parada para a porta serial, a porta serial passa o sinal de parada para o drive do dispositivo, o fluxo de 115.200 bps cessa e então o modem continua a enviar dados à taxa de 33.600 desfazendo-se daquilo que ele já tinha no seu buffer.

Como nada está entrando no buffer, a quantidade de bytes a ser enviada deverá diminuir. Quando praticamente nenhum byte restar no buffer, o modem envia um sinal de início para a porta serial, e o fluxo de 115.2 K do computador para o modem recomeça. Assim sendo, o controle de fluxo cria uma média de fluxo no cabo curto, nesse caso 33.6 K, que é significativamente menor que a velocidade de fluxo nominal de 115.2 K.

Esse é o controle de fluxo “start-stop” e esse foi um exemplo simples de controle de fluxo do computador para o modem, mas há também o controle de fluxo que é usado na direção oposta, de um modem para o computador. Cada direção de fluxo envolve 03 (três) buffers: (1) no modem, (2) no chip UART, que chamamos de FIFOs, (3) na memória principal gerenciada pelo drive de serial. O controle de fluxo protege todos os buffers, exceto os FIFOs, de explodirem (de sofrerem *overflow*).

Os pequenos buffers FIFO da UART não são protegidos dessa forma, mas, ao invés disso, dependem de uma resposta rápida às interrupções que emitem. FIFO significa “o primeiro que entra é o primeiro que sai”, que é a forma como ele lida com os bytes. Todos os três buffers usam a regra FIFO mas apenas um deles também usa isso como seu nome.

Essa é a essência do controle de fluxo, mas ainda existem outros detalhes.

2.4.3 Ausência de controle de fluxo

Os sintomas da ausência de controle de fluxo são blocos de dados faltando do total enviado. Isto se deve ao fato de que quando ocorre overflow são mais do que alguns bytes que sofrem o overflow e são perdidos. Normalmente centenas ou até milhares de bytes se perdem e todos em blocos contíguos.

2.4.4 Controle de fluxo por Hardware ou Software

Se possível o melhor é usar controle de fluxo por hardware, pois este usa dois fios dedicados para enviar sinais de stop e start. Como o hardware dos dois comunicantes está preparado e aguardando o sinal, a reação é imediata e não consome processamento extra.

Controle de fluxo por software usa os dois fios principais de envio e recebimento para enviar sinais de start e stop. Assim usa-se os caracteres ASCII de controle DC1 para start, e DC3 para stop. Eles são simplesmente inseridos no fluxo normal de dados. Portanto, como os comunicantes necessitam analisar o fluxo em busca do sinal, o desempenho geral do sistema diminui. O controle de fluxo por software não apenas é mais lento em reagir a um overflow, mas também dificulta o envio de dados binários, necessitando de precauções extras. Como dados binário provavelmente conterão DC1 e DC3, cuidados especiais devem ser tomados para distinguir entre um DC1 que significa stop e um DC1 que faz parte do código binário. O mesmo para DC3.

2.4.5 O caminho do fluxo de dados - Buffers

Normalmente há um par de buffers de 16-bytes na placa serial e um par de buffers maiores no dispositivo serial. Há ainda um terceiro par de buffers na memória principal de tamanho ainda maior. Consideremos o fluxo de saída, ou seja, o fluxo dos dados no dispositivo transmissor.

Quando uma aplicação manda dados para a porta serial eles são inicialmente armazenados no buffer de transmissão da memória principal. De tempos em tempos o driver do dispositivo serial retira alguns bytes do buffer de transmissão da memória e os coloca um a um no pequeno buffer (digamos 16 bytes) da placa serial de onde eles obrigatoriamente serão transmitidos ao dispositivo serial. Lá novamente eles são colocados em buffers, que podem ser grandes (1K já seria considerável).

Quando o driver do dispositivo cessa o fluxo de bytes de saída, ele cessa o envio dos bytes saindo do buffer da memória principal. O aplicativo pode continuar mandando bytes para esse buffer até que ele alcance a capacidade máxima. Apenas então ele será bloqueado e irá para de rodar até que algum espaço no buffer de memória para a porta serial seja liberado. Se não houver outros threads, o processo será bloqueado esperando recursos de hardware.

2.4.6 Exemplo de fluxo de dados

Em muitas situações, um caminho de transmissão envolve mais de um link, cada um com seu próprio controle de fluxo. Um exemplo seria um terminal texto conectado a um PC conectado a um BBS através de um modem. Um aplicativo roda no PC e fala com as duas seriais, uma conectada ao modem e a outra conectada ao terminal de texto. O texto digitado no terminal entra no PC pela primeira porta serial e vai para o aplicativo, então é enviado pela segunda porta serial para o modem e de lá para o BBS. Bytes sendo enviado do BBS para o terminal apresentar fazem o caminho inverso.

Vamos considerar um caso aonde a velocidade de apresentação do terminal não suporta a quantidade de bytes entrantes. Nesse caso o terminal gera um sinal stop, para controlar o fluxo. Consideremos ainda que esse stop chegue até o BBS.

Vamos analisar o fluxo desse sinal stop que pode ser de hardware em alguns links e de software em outros. Com o fluxo de bytes alto demais para a manipulação no terminal, este gera um sinal stop através de sua porta serial para a primeira porta serial no PC.

O driver do dispositivo detecta o sinal e para de enviar bytes de um buffer de porta serial de 8k na memória principal para o buffer de hardware de, digamos, 16 bytes da placa serial e portanto para o terminal. O aplicativo no entanto continua mandando dados para o buffer de 8k da porta serial.

Quando esse buffer de 8k da primeira porta serial atinge sua capacidade máxima, o aplicativo precisa parar de escrever nele. O aplicativo vai parar e aguardar, e, sendo assim, vai parar de ler dados do buffer de recepção de 8k da segunda porta serial conectada ao modem.

O fluxo vindo do modem continuará até que esse buffer de 8k também atinja sua capacidade máxima e envie um outro sinal stop para o modem. Então o buffer do modem para de enviar dados para a segunda porta serial (no meio do caminho está o pequeno buffer de hardware da segunda porta serial).

O buffer do modem também se esgota, e, caso o controle de erros esteja ativado, envia outro sinal stop para o modem no BBS. O buffer daquele modem deixa de ser consumido e também atinge o limite, causando outro sinal stop a ser enviado para a placa e porta serial a qual esta conectado. No servidor do BBS o buffer de porta serial na memória também atinge sua capacidade, causando o processo que está escrevendo a ser bloqueado temporariamente.

Assim, um sinal de stop de um terminal de texto parou um processo no servidor do BBS. Percebe-se que o sinal stop passou por 4 portas seriais e por um aplicativo no PC. Cada porta serial aqui possui dois buffers em uma determinada direção de fluxo, um de 8k na memória principal e outro de 16 bytes no hardware da placa serial.

O programa aplicativo pode possuir um buffer próprio, o que elevaria o número de buffers através dos quais os dados estão passando a onze. Os pequenos buffers de 16 bytes no hardware da placa serial não participam diretamente do controle de fluxo.

Caso a velocidade de manipulação de dados seja o gargalo no fluxo do BBS para o terminal desse exemplo, é provável que o sinal de stop do terminal realmente pare o aplicativo no BBS. No entanto, para que isso aconteça conforme demonstrado, é necessário que todos os buffers já estivessem próximos de suas capacidades máximas, ou que algum problema tenha ocorrido no terminal que o fizesse parar por tempo suficiente.

Em geral, em um determinado instante, notaríamos que alguns links estariam fluindo, enquanto outros estariam parados. Levaria alguns sinais stop do terminal para que um último stop parasse o aplicativo no BBS.

Caso o controle de erros dos modems estejam desativados, dados poderiam ser perdidos, mesmo antes de chegar ao gargalo do terminal e portanto de nada adiantaria o buffer do aplicativo rodando no PC.

2.4.7 Driver de Dispositivo

O driver de dispositivo da porta serial é o software que opera a porta serial. Esses drivers para portas no padrão RS232 já acompanham os sistemas operacionais em maior uso hoje em dia, e vêm ou como módulos ou como parte do kernel dos sistemas. Pode-se facilmente ver suas características no painel de controle do MS-Windows ou manuseá-los ainda mais facilmente em sistemas Unix.

Além desses, outros drivers específicos para os dispositivos conectados as portas podem ser encontrados e são muito utilizados. Esses últimos (que podem estar embutidos em aplicativos) conhecem as características do dispositivo conectado a porta e normalmente utilizam os drivers da porta para mandarem dados aos dispositivos.

2.5 TUNELAMENTO

> "Tunelamento - Encapsulamento do protocolo A dentro do protocolo B,
> de tal forma que A trata B como se esse fosse uma camada de enlace"
> (CALLON R.; HASKIN D., 1997)

2.5.1 Terminologia Adotada

Antes de iniciarmo-nos nos conceitos relativos ao processo de tunelamento propriamente dito, urge esclarecer a terminologia específica utilizada.

- Dados originais – dados que sofrem encapsulamento.
- Túnel – um caminho de envio (path) entre dois nós no qual o conteúdo dos datagramas são os dados originais.
- Protocolo de entrega – protocolo utilizado pelos nós do túnel.
- Túnel IPv4 – túnel que utiliza IP versão 4 como protocolo de entrega.
- Nó final do túnel – um nó aonde um túnel começa ou termina.
- Cabeçalho do túnel – o cabeçalho adicionado aos dados originais durante o encapsulamento. Ele estão indicando os nós finais do túnel como origem e destino.
- Pacote do túnel – um pacote que encapsula os dados originais.
- Ponto de entrada do túnel – o nó final do túnel aonde os dados originais são encapsulados.
- Ponto de saída do túnel – o nó final do túnel aonde os pacotes do túnel são descapsulados.

- MTU do túnel – o tamanho máximo dos dados originais sem necessitar de fragmentação, ou seja, o MTU do caminho entre o ponto de entrada do túnel e o ponto de saída do túnel menos o tamanho do cabeçalho do túnel.

- Limite de HOPs do túnel - Número máximo de nós que um pacote de túnel pode percorrer do ponto de entrada do túnel até o ponto de saída do túnel.

2.5.2 Tunelador Genérico

O uso de encapsulamento e desencapsulamento de dados é freqüentemente chamado de um tunelamento dos dados, e o encapsulador e o desencapsulador são então considerados como nós finais do túnel .

No caso geral de tunelamento nós temos:

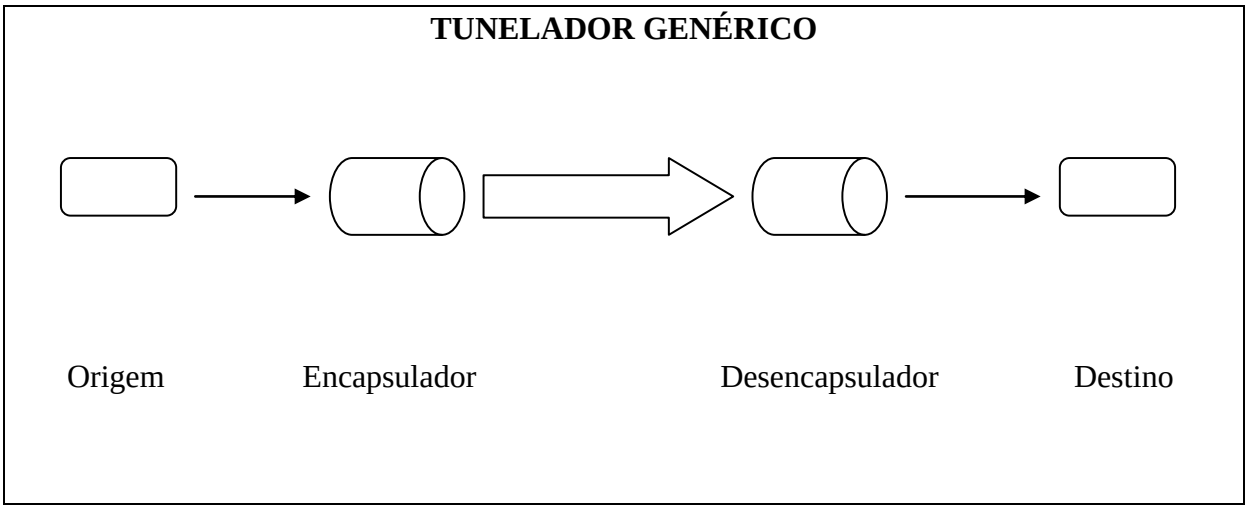

Figura 2 : Modelo genérico de um tunelador

Na Figura 2, podemos observar que em um tunelador genérico a origem, o encapsulador, o desencapsulador e o destino são tratadas como entidades separadas. O nó encapsulador é considerado o ponto de entrada do túnel, e o nó desencapsulador o ponto de saída do túnel. Em geral, pode também haver múltiplos pares origem-destino usando o mesmo túnel entre o encapsulador e o desencapsulador.

2.5.3 Tuneladores Específicos

Técnicas de Tunelamento já são largamente empregadas para ligar protocolos de camada de rede não-IP (e.g. AppleTalk, CLNP, IPX) sobre uma infra-estrutura roteada por IP versão 4 (IPv4). Tunelamento IPv4 é o encapsulamento de pacotes arbitrários dentro de datagramas IPv4 que são enviados sobre uma infra-estrutura IPv4 entre pontos finais de túnel. Para um protocolo tunelado, um túnel parece como um link de um único hop, isto é, roteamentos de dados que utilizam um túnel sobre uma infra-estrutura de camada de rede podem interoperar sobre ele como se fosse um link ponto a ponto. Túneis estáticos ponto a ponto podem ser estabelecidos entre um host e um roteador, ou entre dois hosts. Novamente, cada túnel ponto a ponto configurado manualmente é tratado como se fosse um simples link ponto a ponto.

No caso de um tunelador de dados seriais sobre IPv4, estaremos utilizando uma situação no mínimo curiosa, aonde dados de um nível mais baixo estarão sendo encapsulados sobre um protocolo de nível mais alto. Esta situação ainda não foi prevista por nenhuma RFC nem analisada pela IETF. As RFCs sobre tunelamento são, de outro modo, numerosas e variadas.

Existem RFCs sobre tunelamento de vários protocolos de rede sobre protocolos de enlace, assim como tunelamento de protocolos de nível de enlace sobre protocolos de nível físico e protocolos de rede sobre protocolos de rede como a “RFC2003: IP sobre IP” (PERKINS, C. 1996) e a “RFC2529: Transmission of IPv6 over IPv4 Domains without Explicit Tunnels” (CARPENTER; JUNG, 1999). Também existem RFCs para tunelamento genérico de um protocolo sobre outro como no caso da “RFC1701: Generic Routing Encapsulation – GRE” (S. Hanks, T. Li, D. Farinacci, P. Traina, 1994) e de um protocolo sobre IPv4 “RFC1702 Generic Routing Encapsulation over IPv4 networks” (S. Hanks, T. Li, D. Farinacci, P. Traina, 1994).

Há inclusive RFC de advertência quanto ao tunelamento recursivo de um protocolo sobre outro, mas essa situação foi estudada e abrange protocolos de nível de rede (IPX sobre IP, IP sobre NetBIOS, AppleTalk sobre IP, etc.) (Tsuchiya P., 1992).

No entanto, em nossa situação, não há continuidade da transferência do pacote muito além dos pontos de entrada e saída do túnel. Com essa afirmação restringimos um pouco o uso do túnel aqui proposto. Estamos trabalhando com dispositivos diretamente conectados, pois, caso contrário, insurgiremos em situações absurdas, como o controle de uma conexão SLIP a partir de um ponto remoto, o que apesar de teoricamente possível, exigiria um estudo mais aprofundado.

O presente projeto destina-se à aquisição de dados e controle de dispositivos ligados diretamente a porta serial do nó final do túnel. Devemos portanto restringir o uso de dados seriais usados como meio de enlace entre o dispositivo conectado e algum dispositivo remoto, até concluído o estudo de suas conseqüências.

3 ESPECIFICAÇÃO DO MODELO

3.1 CONSIDERAÇÕES INICIAIS

O desenvolvimento do projeto envolve um ambiente com três componentes: um dispositivo serial, o SerialPipe (SP) e um cliente remoto. Como do dispositivo serial e do ambiente remoto precisamos apenas de interfaces, vamos apenas relacionar alguns pontos chave que merecem nossa consideração quando da implantação. O verdadeiro escopo do projeto é o desenvolvimento do terceiro componente (SerialPipe) que integrará os outros dois. Procuraremos especificá-lo da forma mais granulada possível, para facilitar a implementação. Outros dois componentes são o sistema de arquivos e o console do sistema onde está rodando o SP, que podem ser usados tanto como entrada assim como saída do túnel.

Temos duas situações possíveis a serem consideradas. Enquanto no equipamento junto ao dispositivo serial sendo servido teremos sempre um SerialPipe encapsulando e desencapsulando os dados seriais em pacotes TCP, no lado do cliente remoto, podemos ou não ter um SerialPipe.

Em um serviço de tunelamento de serial **sem** um aplicativo SerialPipe como nó final de túnel na ponta do cliente, temos dois casos:

1. O cliente remoto crê que o dispositivo serial esta conectado diretamente;

2. O cliente remoto tem conhecimento do servidor.

No primeiro caso, onde o cliente remoto crê que o dispositivo serial esta conectado diretamente, temos ainda duas possibilidades. O cliente tem capacidade de redirecionar seu tráfego serial para um socket TCP/IP, ou essa função será feita pelo sistema operacional.

Na primeira possibilidade, quando o cliente tem a capacidade de redirecionar seu tráfego, precisamos, se necessário, criar o socket, estabelecendo a conexão e informá-lo para qual socket redirecionar. Se o cliente for capaz de criar o socket ele mesmo, essa etapa pode

ser dispensada, mas em ambos os casos, precisamos saber o IP e a porta do cliente remoto, para onde direcionar os dados seriais.

Na segunda possibilidade, quando a responsabilidade de redirecionar o tráfego serial recair sobre o sistema operacional, esse pode implementar um middleware com inteligência suficiente para conhecer o IP e porta, criar o socket e estabelecer a conexão, redirecionando ele mesmo todo tráfego endereçado, digamos, para a COM5/cuaa5 para aquele socket, ficando assim, completamente transparente para o software aplicativo cliente o fato de o dispositivo estar operando remotamente.

No segundo caso, onde o cliente remoto tem conhecimento do servidor, este deve ser capaz de descobrir qual o IP e a porta do servidor remoto, estabelecer a conexão, criando o socket, e somente então redirecionar seu tráfego para/de aquele socket. Nesse caso, pode-se utilizar um controle de fluxo próprio, liberando o "servidor de serial" para utilizar seu próprio controle de fluxo com o hardware serial, permitindo inclusive um controle de fluxo por hardware (RTS/CTS), pois caso contrário, o único controle de fluxo possível seria por software (XON/XOFF).

Esse é o único caso aonde o protocolo de aplicação do cliente não SerialPipe pode ser alterado, caso alguma inteligência exista no "servidor de serial". Se o cliente souber da existência do nó remoto, este pode concluir que a inicialização do dispositivo já foi feita quando do estabelecimento da conexão. Pode ainda, enviar apenas os dados pertinentes ao efetivo controle do dispositivo remoto, ou interpretar seletivamente os dados provenientes daquele.

Na situação em que **temos** um aplicativo SerialPipe como nó final de túnel na ponta do cliente, o mesmo pode fazer o papel de middleware, desde que o SO permita o uso da porta serial em modo promíscuo. Assim, o aplicativo tunelador pode capturar os dados enviados pela aplicação cliente diretamente do buffer de serial e jogá-los para o socket aberto, ou vice-versa.

Podemos ainda capturar os dados provenientes de um dispositivo serial e enviá-lo para outro dispositivo que esteja em qualquer lugar atingido por uma rede TCP/IP, especialmente se ligado pela Internet. Esse seria em essência o verdadeiro túnel serial puro, aonde um dado

original serial trafega pela Internet utilizando o protocolo IP como protocolo de entrega. Os dispositivos, um origem e outro destino, estariam vendo o túnel como um único HOP, o ponto de entrada seria o SerialPipe rodando no equipamento aonde está conectado o dispositivo que origina os dados e o ponto de saída aquele rodando no equipamento para o qual os dados seriais destinam-se.

Nesse cenário encontra-se também a necessidade de observarmos algumas das maiores dificuldades a que nos referimos anteriormente, como a relação do MTU versus aplicação de tempo real e o sempre importante controle de fluxo. No entanto, esperamos que alguns testes possam provar que um dispositivo controlado por um, digamos, joystick será capaz de responder adequadamente ao controle por um joystick remoto. Isto deriva do fato de que dispositivos presentes no mundo real que dependem de outros, levam em conta conhecimento sobre estes. Como em aplicações com Inteligência Artificial, dentre outras, usa-se conhecimento sobre o dispositivo para simplificar o controle do mesmo. Quanto mais conhecimento se tem sobre um dispositivo a ser manipulado, mais torna-se possível simplificar as ações necessárias para tal.

3.2 DISPOSITIVO SERIAL

Pode ser qualquer periférico com comunicação serial. Entendemos um dispositivo serial como um dispositivo que reage com o mundo. Assim, ou ele executa tarefas de sua funcionalidade que recebe pela porta serial, ou ele recebe informações do mundo real e transmite-os pela interface serial.

Esse universo é enorme e inclui: modem, mouses, impressoras, um sem número de dispositivos de aquisição de sinais como GPSs e termostatos entre muitos outros.

Cada periférico possui seu próprio protocolo de comunicação. Os dados transmitidos ou a transmitir, possuem um significado diferente para cada função e o dispositivo conhece-o muito bem, pois faz parte de seu software básico.

Além de simples envio dos dados gerados no dispositivo e/ou recebimento de dados com tarefas básicas a executar, alguns dispositivos possuem um protocolo bem mais complexo (e.g. modem, impressora, robô com alguma complexidade de tempo real) e muitos incluem rotinas de inicialização. O protocolo também define a velocidade e os parâmetros de comunicação com o DTE. Conforme já vimos, os periféricos podem comunicar em diferentes velocidades.

Eles podem estar a qualquer distância da porta serial do DTE desde que a tecnologia de conexão suporte.

Estaremos no entanto utilizando dispositivos conectados pelo padrão RS-232, respeitando então suas respectivas restrições de protocolo, velocidade e distância.

3.3 CLIENTE REMOTO

Pode ser qualquer aplicativo, ou mesmo uma janela para entrada e saída de texto aberta diretamente no nó final do túnel, por onde pode-se mandar comandos diretamente para a serial ou receber dados diretamente dela.

Conforme a inteligência contida no lado do servidor de serial, o cliente precisa ter mais ou menos conhecimento do dispositivo serial. Como essa inteligência pode ser variável, para exemplificar vamos considerá-la nula, com dados trafegando da porta serial X sendo servida para a porta TCP Y. Nesse caso, tudo o que o cliente remoto precisa fazer é considerar aquela porta TCP Y do servidor como se fosse a porta serial local para a qual deseja mandar os dados, e desviar seu tráfego de dados para lá. Assim, todo o conhecimento do dispositivo serial deve estar no cliente, inclusive rotinas de inicialização e tratamento de erros.

Usamos esse cenário no modelo considerado acima, onde o sistema operacional funciona como middleware. Nele podemos interceptar os dados sendo transmitidos para a serial local e redirecioná-los para a porta Y, incorporando muitos dos aplicativos já existentes.

Por padrão, o lado cliente deve estabelecer a conexão na porta TCP pré determinada, pois normalmente o servidor já está esperando por ela. No entanto isso pode ser configurado por *plugin*s do servidor e o cliente deve ter conhecimento desses casos. Também o protocolo de comunicação pode ser alterado (como, por exemplo, para um nível superior) e o cliente precisará de cada vez menos esforço para enviar e/ou receber dados da serial remota.

3.4 O SERIALPIPE

Especificaremos as funções definidas para o aplicativo servidor como dois aplicativos dependentes em uma relação pai/filho. O processo pai será considerado o servidor de túnel serial ou processo gerente de túnel e o processo filho o servidor de serial ou ponto final de túnel. Basicamente, temos aqui um processo responsável por disponibilizar, iniciar, configurar, gerenciar, monitorar e finalizar os túneis serial. Enquanto o outro processo (filho) fica responsável por redirecionar o fluxo de dados entre os locais adequados. Este processo filho contém todas as funções específicas do tunelamento em si. O processo pai implementa as funções de configuração, gerência e monitoramento, conforme previsto. Essas tarefas passam a ser exercidas pelo processo pai, que denominamos processo residente ou *deamon*. Todas as outras tarefas são de responsabilidade do(s) processo(s) filho, que implementa(m) os túneis em si.

Vejamos então como processa-se o túnel, do seu estabelecimento ao seu término normal na linha do tempo.

Temos inicialmente apenas o gerente de túnel, que denominaremos por simplicidade de *deamon*. O *deamon* "escuta" em uma porta TCP específica e bem conhecida por todos os aplicativos que almejem utilizar-se dos recursos seriais do equipamento, onde este é executado. Quando um cliente – ou outro tunelador – necessitar das funções do dispositivo ligado ao equipamento rodando este *deamon*, ele deve conhecer seu endereço IP diretamente ou através do sistema de nomes distribuído (DNS). Deve, ainda, conhecer a porta padrão, na qual o *deamon* escuta. Então, ele conecta-se à porta TCP, conhecida como servidora de seriais, e solicita o uso de uma das seriais disponíveis. Solicitada a porta, e estando ela disponível, o *deamon* inicia um processo filho que irá fazer o encapsulamento e desencapsulamento e retorna ao solicitante a porta a ser utilizada como nó final do túnel. A partir desse momento, em um tunelador puro, todo dado dirigido para a porta TCP irá para a porta serial especificada e, portanto, para o dispositivo, e/ou, todo dado do dispositivo, e, portanto, na porta serial, irá para a porta TCP. Quando o cliente/nó remoto tiver sua necessidade satisfeita, deverá avisar o *deamon,* na porta de gerência, e este terminará o processo filho, deixando a porta livre para outro cliente.

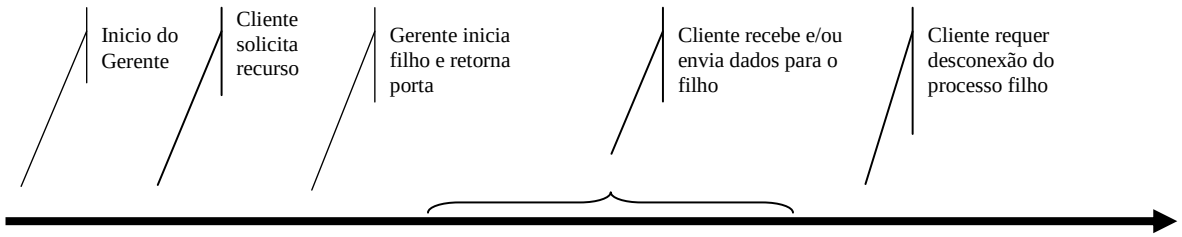

Figura 3 : Timeline de um Tunelamento

Especificaremos a seguir o processo filho, dividido em duas partes, as quais denominamos processo core (que implementa as funções básicas) e *plugin* de comunicação ou filtro (cuja implementação depende das características do dispositivo a ser servido e das funcionalidades exigidas pela aplicação cliente). Então, o processo pai, ou *deamon*, será especificado

3.4.1 Processo Core

Elemento principal e de entrada do processo filho, que implementa as tarefas independentes de dispositivos e de transmissão de dados por TCP.

Os dispositivos a serem utilizados, suas configurações e método de abertura, estão declaradas no *plugin* de comunicação. No entanto, é o processo core que executa a abertura e o fechamento dos dispositivos, pois conhece quais estão instalados e como abri-los no sistema operacional do servidor. Também é ele que envia os dados aos dispositivos, pois também sabe como proceder junto (ou não) ao SO.

A implementação deve levar em conta que parte da lógica e variáveis padrões estarão em outra parte, no *plugin* de comunicação.

Funções Básicas

Veremos, a seguir, as funções elementares, mas essenciais, para podermos trabalhar com os "dispositivos" TCP, porta serial, arquivo e console. Essas funções são o coração do SerialPipe, como o próprio nome core já diz, pois são funções dependentes do sistema operacional, e portanto do hardware, sendo muitas vezes implementadas pelo programador em baixo nível. Após sua implementação, o nível de abstração torna-se muito maior, permitindo que concentremo-nos nas funções principais para o bom funcionamento do túnel.

Função OpenSerial

Funcionalidade: Abre uma porta serial especificada. Também configura essa porta segundo os parâmetros necessários. Grande parte são passados do *plugin* específico ao dispositivo. Esses parâmetros podem ser alterados pelas interfaces de configuração. Faz o tratamento de erros da abertura da porta.

Parâmetros de entrada: Identificação da porta no SO; Velocidade da porta; Paridade da porta; Stop bits da porta.

Resultados: Identificador para a porta.

Função ReadSerial

Funcionalidade: Faz a leitura de um conjunto de n bytes da porta serial especificada. Pode ser blocking ou não-blocking. Trata erros de leitura / comunicação.

Parâmetros de entrada: Identificador da porta, número de bytes a serem lidos.

Resultados: Bytes lidos, número de bytes efetivamente lidos, código de erro.

Função WriteSerial

Funcionalidades: Faz a escrita de um conjunto de n bytes para a porta serial especificada. Trata erros de escrita / comunicação.

Parâmetros de entrada: Identificador da porta, bytes a serem escritos.

Resultados: Bytes transmitidos, código de erro.

Função CloseSerial

Funcionalidade: Fecha a porta serial especificada. Normalmente a porta fica aberta durante o tempo de vida do aplicativo. Trata erros que podem ocorrer no fechamento da porta.
Parâmetros de entrada: Identificador da porta serial.
Resultados: Código de erro/sucesso.

Função OpenTCP

Funcionalidade: Abre uma conexão TCP na porta especificada. Normalmente fica aguardando por uma conexão vinda do cliente, retornando quando estabelecida. Pode ser configurada para iniciar a conexão com o sistema remoto, precisando para isso daquele IP que assim como a porta, é passado do *plugin* específico. Esses parâmetros podem ser alterados pelas interfaces de configuração. É responsável pela segurança no nível de rede, conforme regras do tipo allow/deny por IP e nome. Faz o tratamento de erros da abertura da conexão TCP.
Parâmetros de entrada: Número da porta TCP local, número da porta TCP remoto (opcional).
Resultados: Identificador para a porta.

Função ReadTCP

Funcionalidade: Faz a leitura de um conjunto de n bytes da conexão TCP especificada. Pode ser blocking ou não-blocking. Trata erros de leitura / comunicação.
Parâmetros de entrada: Identificador da porta, número de bytes a serem lidos.
Resultados: Bytes lidos, número de bytes efetivamente lidos, código de erro.

Função WriteTCP

Funcionalidade: Faz a escrita de um conjunto de n bytes para a conexão TCP especificada. Trata erros de escrita/comunicação.
Parâmetros de entrada: Identificador da porta, bytes a serem transmitidos.
Resultados: Bytes transmitidos, código de erro.

Função CloseTCP

Funcionalidade: Fecha a conexão TCP especificada. Trata erros.
Parâmetros de entrada: Identificador da porta TCP.
Resultados: Código de erro/sucesso.

Função OpenFile

Funcionalidade: Abre um arquivo com o nome especificado. O modo de tratamento do arquivo (escrita/leitura) também é passado. Esses parâmetros podem ser alterados pelas interfaces de configuração. Faz o tratamento de erros da abertura do arquivo.
Parâmetros de entrada: Nome do arquivo a ser aberto, tipo de abertura (leitura/escrita/ambos).
Resultados: Identificador do arquivo, código de erro.

Função ReadFile

Funcionalidade: Faz a leitura de um conjunto de n bytes do arquivo especificado. Pode ser blocking ou não-blocking. Trata erros de leitura/arquivo.
Parâmetros de entrada: Identificador do arquivo, número de bytes a serem lidos.
Resultados: Bytes lidos, número de bytes efetivamente lidos, código de erro.

Função WriteFile

Funcionalidade: Faz a escrita de um conjunto de n bytes para o arquivo especificado. Trata erros de escrita/arquivo.
Parâmetros de entrada: Identificador do arquivo, número de bytes a serem escritos.
Resultados: Bytes gravados, código de erro.

Função CloseFile

Funcionalidade: Fecha o arquivo. Trata erros.
Parâmetros de entrada: Identificador do arquivo.
Resultados: Código de erro/sucesso.

Função ReadCon

Funcionalidade: Lê uma linha do console. Retorna todos os bytes até o <CR>. Trata erros de leitura/stdin.
Parâmetros de entrada: Não há.
Resultados: Bytes lidos, número de bytes lidos, código de erro.

Função WriteCon

Funcionalidade: Escreve n bytes no console. Trata erros de escrita/stdout.

Parâmetros de entrada: Identificador da porta, número de bytes a serem escritos.
Resultados: Bytes escritos, código de erro.

Configuração por Linha de Comando

Configurações na linha de comando serão passadas para o aplicativo quando ele for executado. Não está previsto gerenciamento e monitoramento pela linha de comando, sendo necessário reiniciar o aplicativo para mudar os parâmetros ou corrigir erros. Esses parâmetros são passados como opções de linha de comando.

Opções de linha de comando

Escolha da entrada e saída e suas opções - Escolha dos dispositivos de entrada e saída e suas opções, conforme tabela 1 (pág.44) . Caso não selecione-se todas as opções, serão usadas as padrões.

Escolha de *plugin* - Escolha de um dos *plugin*s existentes no sistema. Caso não selecione-se nenhum *plugin*, o padrão será usado.

Inicio e término do túnel - O início é na execução do aplicativo, e o fim do túnel apenas quando do término do aplicativo. Isso implica em iniciarmos o processo de tunelamento assim que rodarmos o aplicativo, utilizando os parâmetros da linha de comando ou os parâmetros padrões existentes no *plugin*.

Suporte a *Plugin*s de Comunicação

O processo de tunelamento deve contemplar a possibilidade de aprimorarmos o conhecimento do dispositivo sendo acessado. Com isto, podemos reduzir o número de bytes trafegados pelo túnel ao necessário para produzirmos o efeito desejável no dispositivo, ou, caso de aquisição de sinais deste, o número mínimo de bytes para conhecermos o estado do dispositivo em questão. O cliente remoto deve, em contrapartida saber deste aprimoramento do túnel, para que possa abstrair os bytes não relevantes, e aproveitar esta facilidade.

Este aprimoramento do conhecimento do dispositivo é, aqui, contemplado com o que chamamos de *plugin* de comunicação. Isto deve ser implementado com a capacidade do processo *core* de permitir que as funções de lógica de comunicação sejam adicionadas sob demanda ao processo responsável pelo túnel (o qual implementa o nó final do túnel propriamente dito). Isto pode ser feito com a inclusão de procedimentos no processo de tunelamento, herança dos métodos e atributos numa instância de túnel e várias outras formas, de acordo com a filosofia da linguagem de programação utilizada para a implementação.

O plugin de comunicação, quando não agrega nenhuma inteligência, procede a um roteamento simples de dados e é chamado de plugin padrão.

3.4.2 *Plugin* de comunicação ou filtros

É a parte do processo filho que conhece o protocolo específico do equipamento serial e o nível de abstração do cliente remoto. Aqui fica a especificação do protocolo de aplicação e a tradução para o protocolo serial que comanda o dispositivo. Ou seja, é feita a tradução de um para outro. Chamamos a isso de inteligência em uma referência ao grau de conhecimento do dispositivo que a implementação do túnel irá adquirir. Podemos ainda executar outras funções padrão que não necessitem da intervenção do usuário ou cliente remoto nem do dispositivo.

Vamos apresentar aqui as considerações gerais que devem ser seguidas quando da implementação dos plugins.

Protocolo serial

Como vimos, este é o protocolo que o dispositivo entende. Sendo a linguagem do dispositivo, a sua implementação é extremamente dependente do hardware e firmware daquele. No plugin de comunicação, o protocolo serial é portanto, aquele que é utilizado no lado do equipamento conectado a porta serial.

Pode ser desde o protocolo RS-232 puro, com o simples envio de caracteres (bytes) para o dispositivo, até o envio de vários caracteres para que o dispositivo execute uma função

complexa. Uma tarefa como “mover o braço mecânico 30 graus para a direita” pode ser implementada pelo dispositivo em uma função própria para esse fim, que exija apenas os graus como parâmetro, mas provavelmente será necessário enviar vários comando de alguns bytes cada pela porta serial para completar a tarefa. Também a leitura da posição atual pode ser feita automaticamente pelo plugin servidor para validar o movimento.

Algumas tarefas padrão pertinentes ao dispositivo serial podem ser executadas sem o conhecimento explícito do outro dispositivo ou cliente. Podemos citar uma inicialização do equipamento ligado a serial, que muitas vezes precisa estabelecer a comunicação e receber parâmetros de configuração para poder começar a receber comandos. Essa inicialização muitas vezes independe da aplicação e não há a necessidade de intervenção de mais ninguém além do computador servidor de serial. Por este motivo colocamos esses casos no protocolo serial, pois devem estar associadas ao dispositivo serial.

Podemos citar ainda o tratamento de alguns erros básicos ou que exijam resposta imediata, como no caso de um robô que encontre uma situação anômala cuja intervenção do equipamento mais próximo mostre-se essencial. Ilustramos esta situação na Figura 4.

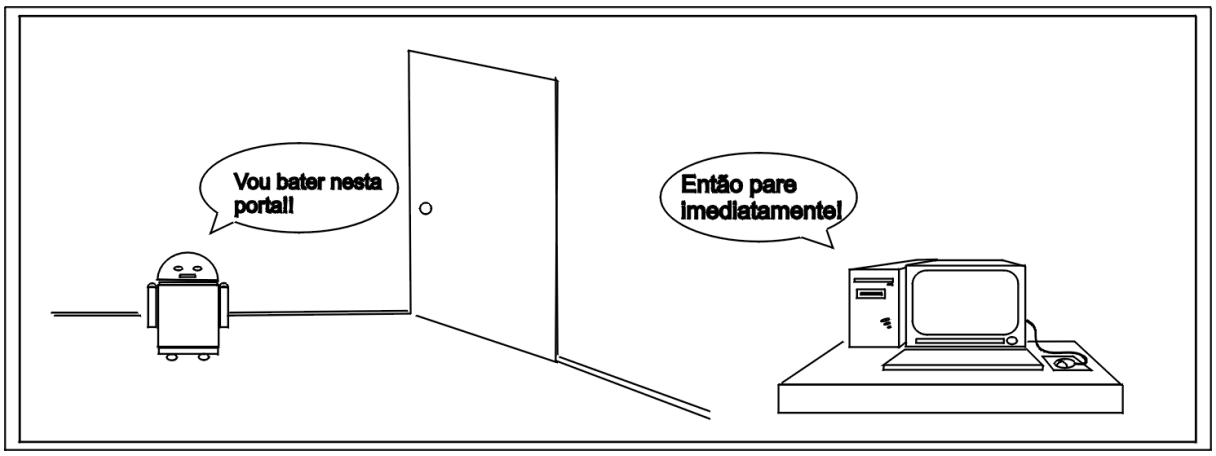

Figura 4: Exemplo de resposta padrão ao dispositivo

Neste caso, após a intervenção padrão do DTE, a situação seria então informada ao aplicativo cliente que encontra-se remoto, para providências corretivas, mas até lá, o lado do túnel ligado ao dispositivo deve conter essa inteligência imprescindível.

O módulo que trata do protocolo serial seria, portanto, em essência, funções que enviam um ou mais conjunto de bytes pela porta serial, representando algo que o dispositivo entenda, ou que receba uma série de bytes do dispositivo os quais passarão por uma tradução para o protocolo de aplicação, conforme a lógica de comunicação (ver adiante). Essas funções devem ser implementadas de acordo com o equipamento que se pretende controlar.

Protocolo de aplicação

É aquele que o SP utiliza no lado da porta TCP. Destina-se a implementar um protocolo para comunicação com o cliente remoto. Pode ser desde o recebimento de bytes crus que devem ser enviados para a serial, ou vice versa, até um protocolo de alto nível que receba e envie instruções genéricas e dados tratados. Pode-se receber todos os comandos necessários para mover um braço mecânico, assim como se pode receber apenas uma string que diga "mova o braço 30 graus para a direita".

Alguma troca de informações entre o servidor e o ponto remoto pode se dar sem que nenhum evento seja gerado no equipamento serial. Exemplo disso é checar se os dois lados estão funcionando bem. Caso haja uma situação de exceção os dois sistemas podem trocar essa informação apenas para setar variáveis de controle e não passar pela mesma situação novamente.

A colocação de buffers na porta TCP também é desejável. Algumas vezes o equipamento vai enviar uma quantidade maior de dados do que permite a qualidade da conexão TCP. Nesse caso até uma amostragem para envio pode ser necessária. Essa amostragem pode ser parte da lógica de comunicação, mas também pode ser automaticamente implementada pelo protocolo de aplicação.

Como pode-se ver, não só o protocolo serial é dependente do equipamento, mas também o de aplicação pode ser mudado para facilitar o envio e recebimento de dados pertinentes a ele.

Em essência, esse módulo implementa o protocolo de comunicação com o cliente remoto, sendo que aquele pode ser desde um telnet com console, como um aplicativo complexo. Conforme a inteligência vai sendo adicionada no *plugin*, o protocolo de comunicação com o cliente remoto muda, falando-se em um nível cada vez mais alto.

Lógica de Comunicação

É a parte principal do SP. É a lógica que sabe como o fluxo, entre o dispositivo serial e o cliente remoto, se dá. Como tanto a porta serial, como a conexão TCP, podem ser utilizadas para recebimento assim como para envio de dados, a lógica deve saber controlar o fluxo desses dados.

A comunicação dar-se-á de forma simplex, half-duplex, ou, ainda, full-duplex, dependendo do tipo de comunicação e do tipo de equipamento sendo controlado ou medido. Por exemplo, um medidor de temperatura que envia seus dados em intervalos fixos de tempo fará um comunicação simplex. Caso ele necessite de intervenção para o envio da temperatura, realizando-a apenas sob demanda, o fluxo será half-duplex, com a temperatura fluindo para o cliente após o envio da solicitação por aquele. Já no caso de um medidor programável, várias operações podem estar ocorrendo simultaneamente, entre programações e envios de dados, caracterizando uma comunicação full-duplex.

Também recomenda-se implementar, na lógica de comunicação, uma lista de prioridades, tanto para o sentido do fluxo, como para o envio dos dados em si, dependendo de algum evento específico.

A lógica entra em funcionamento assim que as conexões com os dispositivos estiverem estabelecidas. Com o túnel em funcionamento normal, ele só deverá perder o controle de seu processo quando o túnel for intencionalmente encerrado. Ou quando alguma

interrupção inesperada ocorrer. Ele deverá reportar-se então para seu processo pai acusando o código do erro, e este fará a interface com o usuário.

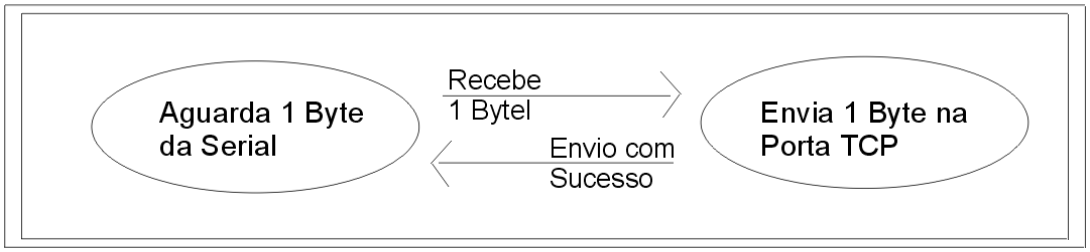

Figura 5: Exemplo de lógica de comunicação simples

Conforme podemos deduzir (vide figura 4), o plugin padrão efetuará uma lógica de comunicação muito simples, apenas roteando os bytes entre a porta serial e a porta TCP, em ambos os sentidos. Assim, todos os bytes trafegarão de forma transparente entre o cliente e o dispositivo, dando-lhes a impressão que estão diretamente conectados.

3.4.3 Processo Gerente de Túnel ou *Deamon*

Consiste do processo pai ou *deamon*. Ele escuta por requisições em uma porta TCP fixa, bem conhecida por quem precisa utilizar o dispositivo serial conectado ao servidor que está sendo por ele gerenciado. Este processo pode estar sempre rodando, tendo sido iniciado na inicialização da máquina, e assim fazer justiça ao nome deamon. Este é o modo que planejamos inicialmente para este processo, mas no entanto, ele pode também ser acionado por um aplicativo específico do sistema operacional (ex. tcpwrapers do UNIX) quando de uma tentativa de conexão na porta TCP supra citada ou pode ainda ser acionado manualmente pelo usuário gerente.

Como poemos ver na figura 5, assim que iniciado, o processo gerente passa a aguardar numa porta específica de gerência. É através dessa porta que um candidato a usuário remoto do dispositivo fará uma requisição para utilizar o dispositivo.

Recebida uma requisição, o gerente verificará então a disponibilidade do dispositivo solicitado, para confirmar se ele está conectado e disponível, ou seja, garantirá que ele não

está em uso, ou que é de uso simultâneo. Caso o dispositivo esteja disponível, o processo gerente iniciará um processo filho responsável pela implementação do túnel, como já vimos. No sucesso desta inicialização o processo gerente retornará ao cliente remoto, agora já efetivo, a porta de comunicação com o dispositovo solicitado. Em geral, na prática, o usuário remoto já sabe de antemão a porta em que o dispositivo serial será disponibilizado, mas de qualquer forma, esta porta lhe é informada.

No caso de erro em qualquer etapa acima, o processo gerente retorna ao estado de aguardando requisição, liberando, se necessário, o dispositivo serial relacionado.

Quando do término da utilização do dispositivo pelo cliente remoto, este, idealmente, enviará uma requisição de término de conexão, ao qual o processo gerente atende liberando o dispositivo serial, se necessário, e retornando uma confirmação ao cliente remoto. O processo gerente volta então a aguardar por alguma requisição de utilização.

É importante salientar que, caso o cliente remoto não possa proceder as requisições necessárias, uma interface local, assim como uma remota, estão disponíveis para o usuário interessado proceder a instanciação do túnel. Neste caso, após a utilização é importante que por questões de segurança do dispositivo, o túnel seja finalizado por uma das interfaces.

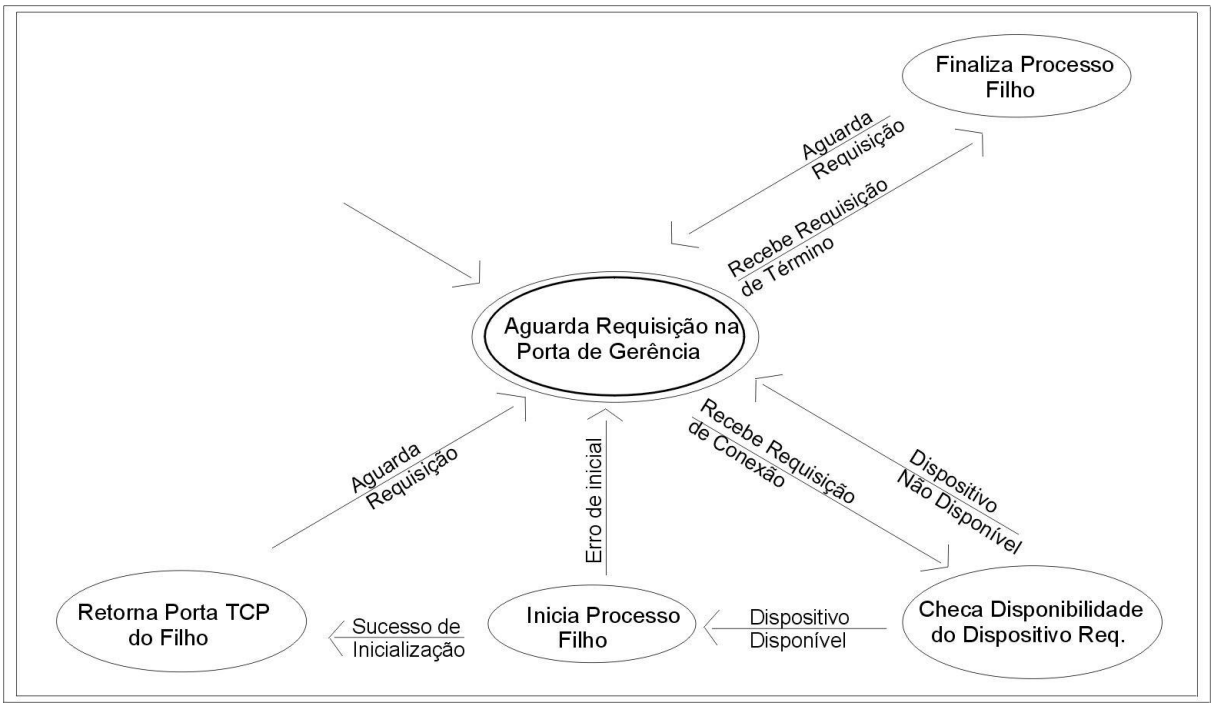

Figura 6: Diagrama de estados do processo gerente

Podemos dividir as funções deste processo em configuração, monitoramento e gerência. Isto porque enquanto a **Configuração** e **Monitoramento** requerem uma **Interface homem máquina**, a gerência pode ocorrer basicamente sem intervenção. Para isso, precisamos especificar para a **Gerência**, **Protocolos de requisição e liberação**, assim como **Tratamento de erros**. Essas destacadas são as sessões a seguir, e serão apresentadas na ordem em que foram destacadas neste parágrafo.

Configuração

São as funções que modificam de alguma forma uma ou mais instâncias de túnel. A maioria das modificações ocorrem antes do início do trafego de dados pelo túnel, através da passagem de parâmetros para o aplicativo filho antes de sua execução. Essas funções podem inclusive sobrepor os dados e a lógica contidas no *plugin*.

São as seguintes as funções de configuração:

Instanciação de Túnel:
Permite a escolha de um dos túneis ativos para gerência ou a criação de um novo túnel.
Temos como resultado o identificador do túnel a ser gerenciado. No caso de estarmos criando um túnel novo, temos como resultado principal o túnel instanciado.

Escolha das Entradas e Saídas:
Permite escolheremos quais as entrada e saída do nó final de túnel que estamos gerenciando.
Em geral, se estamos gerenciando um nó final de servidor de serial, sendo esta a situação mais usual, iremos escolher uma das opções como uma porta TCP e a outra como uma porta serial.
Já se estamos usando o tunelador do lado do cliente remoto, iremos escolher um dos lados como uma porta TCP e o outro conforme a necessidade. Nesta especificação, as opções são, além de uma porta serial, outra porta TCP, arquivo ou console.
Note-se que a utilização de outra porta serial no lado do cliente remoto serve não apenas para uma comunicação dispositivo a dispositivo, caso em que os dois lados são na prática servidores de serial. Esta utilização serve também para em alguns sistemas operacionais, como o Unix, garantirmos uma utilização transparente do do dispositivo por qualquer cliente

aplicativo, visto que nesses sistemas a serial é disponibilizada através do sistema de arquivos, permitindo assim um acesso promíscuo ao identificador de serial pelo aplicativo e pelo tunelador.

Escolha de *Plugin*:
Por padrão o plugin a ser carregado na criação de uma instância de túnel é o de roteamento simples. Neste, os dados serão simplesmente roteados entra o par de entradas/saídas do nó.
Esta função permite a escolha de outros *plugins* que estejam disponíveis.

Monitoramento

As funções de monitoramento permitem ao usuário gerente uma visualização do funcionamento do túnel. Apesar de imprescindíveis na implementação do tunelador, essas funções vão muito além, pois têm utilidade permanente.

Visualização do Console:
Permite a visualização do console nos casos em que este é escolhido como destino do nó final do túnel sendo monitorado.
Em alguns casos é imprescindível, como naqueles em que o sistema operacional é totalmente gráfico. Também, como a escolha de console implica na utilização da entrada e saída padrão do SO (stdin e stdout) e essas podem estar redirecionadas, a visualização dos dados neste ponto é de extrema importância para a verificação do funcionamento adequado.

Interceptação de E/S:
Mesmo nos casos em que não utilizamos o console, é importante a visualização dos dados que estejam trafegando pelo túnel, tanto na entrada como na saída do nó sendo monitorado.

Indicação de Funcionamento:
Uma indicação visual de que o túnel sendo monitorado está habilitado a receber e enviar dados.

Indicação de Fluxo de Dados:

Uma indicação visual de que dados estão passando pelo túnel num determinado momento.

Todas essas funções estão disponíveis nas interfaces gráficas, tanto na interface local como na interface remota via web.

Interface Gráfica

A interface gráfica existe no processo pai, portanto independente do processo de tunelamento, pois esse será controlado por um processo *core* filho. Supostamente rodando no mesmo equipamento do processo responsável pelo tunelamento, a interface permite uma total configuração das funções e opções dos dispositivos.

Escolha do túnel - Permite escolher qual instância vamos configurar, gerenciar e monitorar.

Escolha da entrada e saída e suas opções - Permite a escolha dos dispositivos de entrada e saída e suas opções, conforme a tabela 1 abaixo. Cada dispositivo possui seu próprio frame na janela principal, onde além dos dispositivos para seleção, estão as opções pertinentes a cada dispositivo.

Console - Quando da escolha do console como um dos dispositivos, temos ainda um frame separado para termos acesso a ele. Temos uma janela de saída representando o stdout e um box de entrada representando o buffer do stdin.

Visualização de entrada e saída - No mesmo frame dedicado a escolha de dispositivo e opções temos uma janela para monitoramento do tráfego de/para aquele dispositivo. Assim, mesmo sem escolhermos o console (stdin/stdout) como dispositivo, temos acesso aos dados trafegando pelo túnel para monitoramento.

Escolha de *plugin* - No menu principal da interface teremos uma janela para gerenciar o(s) *plugin*(s) a serem usados. Inicialmente teremos apenas uma instância de túnel, portanto escolheremos apenas um *plugin*.

Inicio e término do túnel - Esta função é na realidade, dentro do escopo deste trabalho, de gerenciamento. Botões de INICIAR e PARAR permitem iniciarmos e interrompermos o fluxo

de dados entre os dispositivos. Esses botões servem para manualmente iniciarmos um processo filho com as características selecionadas acima, e também para finalizarmos o processo responsável pelo tunelamento. Inicialmente precisaremos fazê-lo cada vez que trocarmos qualquer configuração de algum dos dispositivos ou o *plugin*.

Tabela 1: Opções de Dispositivos

Dispositivo	Opções
TCP	número da porta passivo ou ativo se ativo ip remoto
Serial	identificação da porta velocidade de transmissão numero de caracteres tipo de paridade numero de stop bits
Arquivo	nome do arquivo caminho do arquivo

Interface Remota

A interface remota será utilizada via browser. O gerente da conexão existe independente do processo de tunelamento, pois esse será controlado por um processo filho.

Ele será responsável pela transmissão dos dados de monitoramento, assim como pela recepção das ações solicitadas. Como se pode ver a seguir, essa interface tem todas as características da interface gráfica local, permitindo que todas as configurações sejam feitas também remotamente.

Escolha do túnel – Assim como na Interface Gráfica.

Escolha da entrada e saída e suas opções - Assim como na Interface Gráfica.

Console - Assim como na Interface Gráfica.

Visualização de entrada e saída - Assim como na Interface Gráfica.

Escolha de *plugin* - Assim como na Interface Gráfica.

Inicio e término do túnel - Assim como na Interface Gráfica.

Protocolo de administração - Este protocolo refere-se àquele de comunicação necessário para que um cliente de administração remoto possa programar, no processo gerenciador de túneis, as opções constantes de sua interface, conforme descritas acima. Apesar de uma solução proprietária parecer interessante para prototipação e segurança, uma solução aberta deve ser buscada, como reza a premissa do presente trabalho.

Assim sendo, definimos o protocolo HTTP como protocolo de comunicação e procedimentos baseados no servidor como agentes. Para isto, devemos ter funções no processo gerente que simulem um pequeno servidor HTTP, fornecendo páginas em HTML com *forms* específicos às funções supra mencionadas, e capacidade de interpretação dos resultados POST tal qual linguagens de implementação de CGIs.

A segurança passa a ser definida, no nível de aplicação, como resultado de login e logout em base de dados específica.

Gerenciamento

O gerenciamento trata da distribuição de recursos disponíveis, neste caso, dispositivos seriais. Esta é a função responsável pelo início e término de instâncias de túneis. O gerenciamento de túneis ou instância de túneis ocorre principalmente quando da solicitação de utilização de um determinado dispositivo serial disponível. Assim, quando ocorre a solicitação, deve se verificar se o dispositivo está disponível para compartilhamento e proporcioná-lo em caso positivo. Também é função do gerenciamento finalizar a utilização de um dado dispositivo serial.

O gerenciamento pode portanto ser exercido ou através das interfaces de configuração e monitoramento como já observamos anteriormente, ou dinamicamente de acordo com as solicitações de utilização de dispositivos.

Para o gerenciamento dinâmico das conexões, precisamos de um protocolo de requisição e liberação, assim como de um método de tratamento de erro. Opcionalmente, podemos ter um esquema de prioridades, controle de concorrência e outras funções típicas de um *deamon*. Estas opções, no entanto, deixamos para um trabalho futuro.

Protocolo de Requisição e Liberação

Definimos três primitivas: requer conexão, aceita conexão, requer desconexão, aceita desconexão.

Ele se dá conforme a tabela 2 abaixo:

Tabela 2: Protocolo de Requisição e Liberação

1. REMOTO ----------→ REQUER CONEXÃO (X) ----------→GERENTE
2. REMOTO <---------- ACEITA CONEXÃO (Y) <---------- GERENTE
3. REMOTO ----------→ REQUER DESCONEXÃO (X) ----------→GERENTE
4. REMOTO <---------- ACEITA DESCONEXÃO (X) <----------GERENTE

Onde:

X é o número do dispositivo ou porta requerido;

Y é a porta na qual o nó final do túnel estará esperando para a conexão de dados seriais.

- Antes do passo 2 o GERENTE inicia um processo filho esperando na porta Y.
- Opcionalmente, de acordo com a configuração do túnel, o processo tunelador filho irá abrir uma conexão de forma ativa em uma porta específica no IP de REMOTO.
- O GERENTE poderá responder com NÃO ACEITA CONEXÃO no lugar de ACEITA CONEXÃO no passo 2, caso não possa, por algum motivo, aceitá-la.
- Não existe NÃO ACEITO DESCONEXÃO, nem REQUER DESCONEXÃO por parte do GERENTE. Para identificar esses casos, o REMOTO deve implementar time-out. Caso o REMOTO seja outro GERENTE, isso dar-se-á na lógica de comunicação.

Tratamento de Erro

Caso o processo filho, reporte um erro, o gerente deverá, além de informar à interface ativa, tomar as medidas cabíveis. Aqui são descritos quatro casos essenciais, sendo vários outros tratamentos possíveis, podendo ser implementados em um trabalho futuro.

ERRO: Erro crítico reportado na abertura da porta serial.
TRATAMENTO: Não envia ACEITA CONEXÃO.

ERRO: Erro crítico reportado na abertura da porta TCP.
TRATAMENTO: Não envia ACEITA CONEXÃO.

ERRO: Erro reportado pela lógica de comunicação.
TRATAMENTO: Encerra conexão e disponibiliza serial.

ERRO: Erros críticos reportados no envio de dados.

TRATAMENTO: Encerra conexão e disponibiliza serial.

4 IMPLEMENTAÇÃO DO MODELO

4.1 A LINGUAGEM TCL/TK

Como muitas outras linguagens, entre as quais podemos citar PERL, PYTHON e PHP, a Tool Command Language (Tcl) surgiu logo após o advento da popularização da Internet, como ferramenta para desenvolvimento para aplicativos do lado do servidor (server side applications). Como essas outras linguagens, ela é fundamentalmente interpretada, apesar de existirem compiladores para o seu código. A extensão Tk foi desenvolvida especialmente para a linguagem Tcl pela mesma pessoa, John Outerhout, mas logo surgiram adaptações para várias outras linguagens. Tk possui os vários elementos gráficos necessários para se criar uma GUI. Sua facilidade em criar janelas e sua interface/biblioteca para várias linguagens logo chamou a atenção de um sem número de desenvolvedores de aplicativos.

Entre suas vantagens para com nosso trabalho, e consequentemente o motivo de sua escolha podemos citar:

Fácil Prototipação: por ser uma linguagem interpretada, a criação de protótipos é muito facilitada, pois basta um editor de textos para corrigirmos algum erro ou alterarmos o código quando melhor nos convier;

Multiplataforma: interpretadores TCL/TK existem para praticamente todas as plataformas difundidas e as quais temos acesso, incluindo FreeBSD, OpenBSD, NetBSD, e outros UNIX BSD, Linux, SCO Unix, e outros UNIX system V, Windows 95/98/NT, MacOS e outros sistemas Macintosh;

Extensa Biblioteca de Hardware: nossas necessidades de comunicação com portas seriais são facilmente supridas por funções pré-definidas em suas bibliotecas de hardware, facilitando em muito sua programação;

Extensa Biblioteca TCP/IP: novamente, nossas necessidades de comunicação com pilhas TCP/IP são facilmente supridas por funções pré-definidas em suas bibliotecas de protocolo de transporte/rede, facilitando em muito sua programação;

Interface Nativa com TK: a facilidade de programação de código TK permitiu a criação rápida de interfaces para visualização e testes. TK é considerados por poucos mas bons programadores de ferramentas de administração de redes americanas como superior ao controle de janelas oferecido pela linguagem Java™.

Código Aberto: sem dúvida um dos mais importantes paradigmas modernos, essa facilidade nos dá a opção de analisarmos em que parte da comunicação de baixo nível (por exemplo) se deu determinado problema. Bem melhor que uma janela de erro com um código incompreensível ou ainda uma "tela azul".

4.2 PROCESSO FILHO

Inicialmente, foram implementados as funções do processo core e testadas. Os testes obtiveram sucesso e as funções foram aprovadas para dar suporte ao aplicativo.

Em seguida, foram implementados *plugin*s básicos conforme especificado.

Esses *plugin*s foram implementados e testados como prova de conceito, na seguinte ordem:

1. serial->TCP. Com isso, podemos capturar dados aleatórios gerados por qualquer dispositivo serial (e.g. Mouse) e mostrá-los sem qualquer tratamento em uma tela de telnet à porta TCP. Os dados, apesar de difícil leitura, por serem crús, foram apresentados com sucesso.

2. TCP->serial. Assim foi possível enviar dados serial puros para um dispositivo serial. Para este teste utilizamos um microcontrolador com um protocolo simples, ligado a pequenos motores elétricos. O microcontrolador aguardava três valores pela serial e posicionava os eixos dos motores de acordo com os parâmetros enviados,

movimentando um braço mecânico com dois graus de liberdade. A este braço temos conectado uma pequena câmera de video, dando utilidade real à implementação.

Para testarmos com maior precisão, incorporamos então a interface local ao protótipo. A interface foi essencial para obtermos um conjunto ótimo de simulações.

Com isso, obtivemos um nó final de túnel. Apesar de encontrar-se inicialmente solitário na estrutura especificada, podia rodar como um aplicativo estanque e produzir o efeito desejável, isto é, tunelar dados serial dentro de pacotes TCP. Além disso, com ele somos capazes de configurar as características do túnel e gerenciar e monitorar o funcionamento do túnel em si.

Algumas funções foram implementadas conjuntamente com outras no nível de codificação para fins de teste. Isto é verdade especialmente no que se refere aos dispositivos de entrada/saída por console e por arquivo, que obtivemos substituindo o código original porta TCP ou porta serial. Isso deve-se a facilidade de alterarmos um código interpretado e à característica destes dispositivos de entrada e saida (ou stdin/stdout) e arquivo (ou fs) servirem especialmente para debug. No entanto, essas funções serão implementadas como função específicas, pois facilitam o tratamento de erro em uma instalação particular.

Ainda como característica desta fase devemos apontar que as funções de tratamento de serial acabaram sendo provisoriamente implementadas dentro do plugin, que por sua vez foi incorporado ao código do processo core.

Apresentamos a seguir as telas do processo de tunelamento em sí, chamado de processo filho, para uma perfeita compreensão do objetivo implementado e funcional.

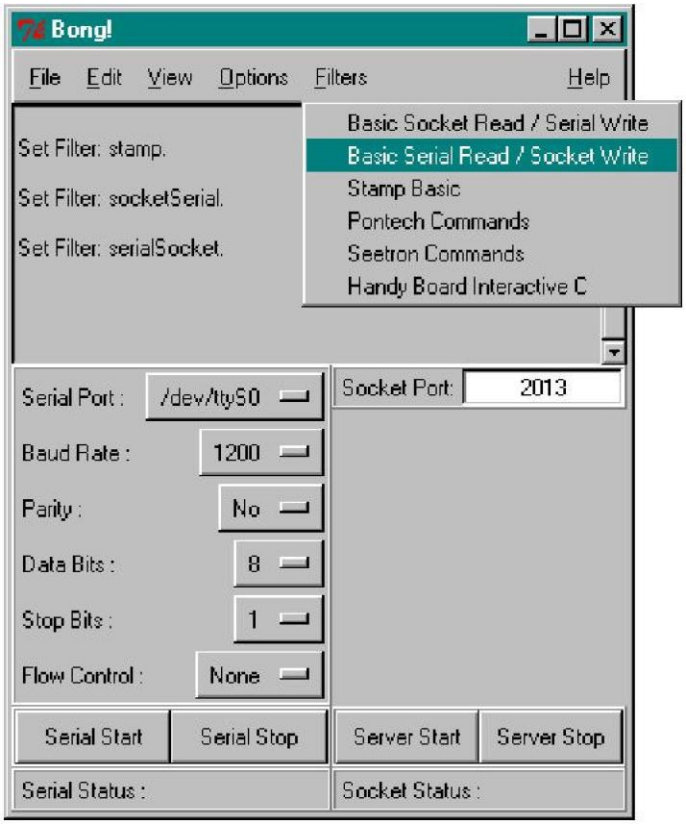

Figura 7: Escolha de um filtro

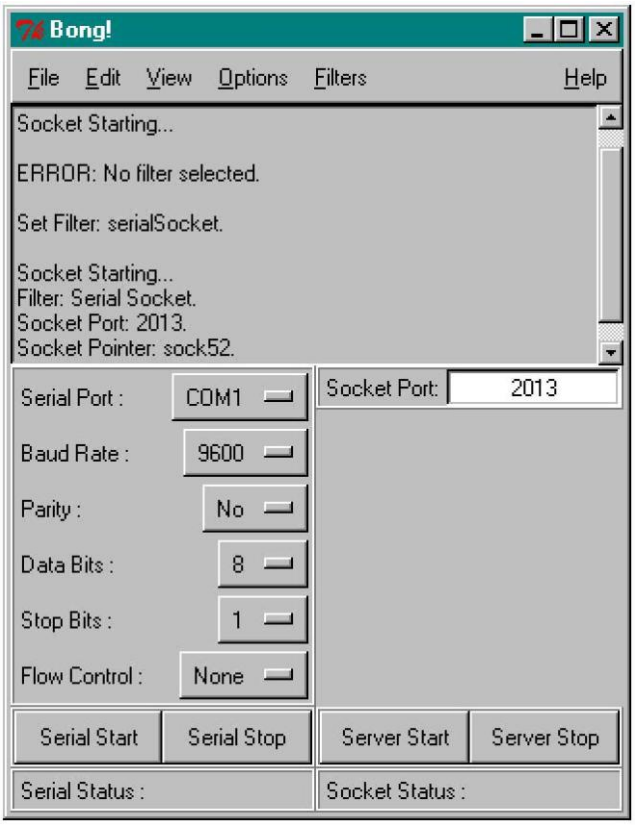

Figura 8: O tunelador ativo

4.3 PROCESSO PAI OU GERENTE

Inicialmente retiramos a interface local do nó final do túnel, implementando-a em um processo a parte.

Implementamos então o protocolo de requisição e liberação. Estas funções foram testadas com um par de processos gerente. Nesta fase ficou evidente a necessidade de um timeout durante a negociação, que foi então implementado.

Feita a negociação do recurso, a função do gerente é o disparo do processo filho. Essa função foi implementada e a ação de monitoramento feita através de criação de um arquivo no diretório de trabalho do sistema operacional. Tal arquivo é atualizado pelo processo filho regularmente. Caso o arquivo fique além de um limite pré estabelecido sem atualização o processo gerente finaliza e reinicia o processo filho.

Apresentamos a seguir as telas do módulo gerente, que demonstram a implementação das funções especificadas, assim como apresentam o módulo principal do tunelador, totalmente funcional.

Figura 9: O Gerente

Figura 10: Gerente aguardando conexão

Figura 11: Cliente a conectar-se

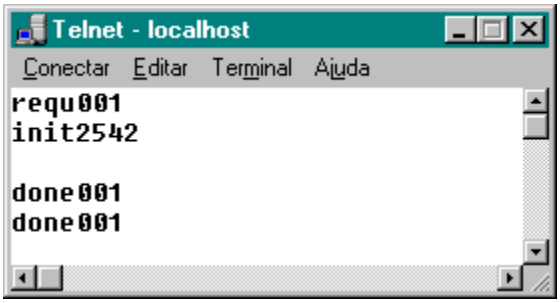

Figura 12: Requisições de Conexão e Desconexão

5 CONCLUSÃO

Devemos ressaltar que o presente trabalho resultou em um protótipo completamente funcional. Sua demonstração acadêmica foi acompanhada por demonstrações em eventos e feiras. Seu código aberto foi modificado por colegas e segue em estudos de viabilidade da idéia no mercado.

Após terminarmos os testes com o protótipo, tiramos duas conclusões básicas:

- as funcionalidades e, portanto, a utilidade de um tunelador de serial são de grande valia, não só para as necessidades específicas que nos levaram a sua implementação, como para todos os aplicativos que de alguma forma precisam comunicar-se com dispositivos seriais remotos;

- melhorias ainda devem ser feitas para lançá-lo como um produto digno de uma licença de uso público.

A aquisição de sinais de dispositivos remotos, por si só, justifica um aprofundamento nessa filosofia de acesso a dispositivos remotos, visto que conexões TCP multicast são por definição promíscuas. Assim sendo, clientes facilmente implementados de monitoramento de equipamentos remotos, podem ser utilizados por vários estudantes ao mesmo tempo, permitindo uma disseminação das facilidades implantadas e justificando investimentos.

No entanto, a multiplicidade de aplicações é bem maior. Nossa pequena câmera de video é movida por dois servo-motores acionados por uma controladora com interface serial. Podemos destacá-la como uma aplicação em que dados seriais são usados para provocar efeitos no mundo real. Para a movimentação de um dos servos é necessário o envio de alguns bytes para a controladora. Em ambos os casos acima, alguns bytes trafegando de ou para a porta serial são suficientes para alcançarmos nosso objetivo. Em outros casos, há a necessidade de enviarmos e recebermos dados seriais, como para controlar um carro

remotamente, aonde além do envio de uma mudança de direção deve haver um constante recebimento da posição do objeto.

A flexibilidade do tunelador também é importante. No caso da câmera, oito bytes são enviados, mas apenas dois têm significado para o movimento: qual servo movimentar e a posição desejada. Portanto, é preferível enviarmos apenas esses dois bytes, caso o cliente remoto seja capaz de fazê-lo. Caso contrário, no qual o cliente remoto age como se a controladora estivesse conectada diretamente a ele, o envio de todos os bytes se faz necessário. Portanto, o túnel depende essencialmente das capacidades do cliente remoto.

O caso ideal de utilização do tunelador é como middleware entre o aplicativo e o sistema operacional. Caso o sistema operacional permita a utilização de seriais em modo promíscuo, isto é, caso permita que mais de um processo tenha acesso ao buffer da serial, basta iniciarmos o tunelador no lado do cliente remoto, configurando-o para recuperar os dados da mesma porta serial utilizada pelo aplicativo e então iniciar a conexão com o outro ponto final do túnel. Caso o sistema operacional não permita, deve-se desenvolver um redirecionador que capture os dados do aplicativo antes do dele e envie-os para o tunelador, simulando para ambos uma porta serial.

O controle de fluxo deve ser melhor endereçado, pois sua utilização é importante para aplicações complexas. Esse controle de fluxo deve ser implementado necessariamente como controle por software e nossa sugestão é que o tunelador interprete o controle de fluxo contido no túnel e traduza-o para o controle de fluxo já existente entre ele e o dispositivo físico. A utilização de buffers grandes no tunelador, assim como as restrições aos tipos de dispositivos conectados permitiram que nossa implementação funcionasse sem maiores problemas.

Esperamos que o presente trabalho continue a ser aperfeiçoado, com a implementação mais fiel da especificação e os desenvolvimentos recém apresentados, essenciais para um funcionamento adequado em qualquer aplicação. O desenvolvimento de funções core para outros dispositivos, em particular dispositivos USB, torna-se cada vez mais necessário, visto que estes últimos têm tornado-se cada vez mais freqüentes no mercado.

6 REFERÊNCIA BIBLIOGRÁFICA

Telecommunications Industry Association. EIA-232-E Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange. (1991)

FARINACCI, D.; LI, T.; HANKS, S.; MEYER, D.; TRAINA, P. RFC2784: Generic Routing Encapsulation (GRE). Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc2784.txt> Acesso em: 2000

MOSCHOVITIS, CHRISTOS J. P. et al. The History of Internet. New York: MTM Publishing, 1999.

CARPENTER, B.; JUNG, C.. RFC 2529: Transmission of IPv6 Packets over IPv4. Hampshire, Santa Clara: The Internet Society, 1999.

WOODBURN, R. A.; MILLS, D. L. RFC1241: Scheme for an internet encapsulation protocol: Version 1. Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc1241.txt>. Acesso em: 2000

TSUCHIYA, P. RFC1326: Mutual Encapsulation Considered Dangerous. Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc1326.txt>. Acesso em: 2000

CALLON R.; HASKIN D. RFC 2185: Routing Aspects Of IPv6 Transition Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc2185.txt>, 1997

HANKS, S.; LI T.; FARINACCI D.; TRAINA P. RFC1701: Generic Routing Encapsulation (GRE). Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc1701.txt>. Acesso em: 2000

HANKS, S.; LI T.; FARINACCI D.; TRAINA P. RFC1702: Generic Routing Encapsulation (GRE) over IPv4 networks. Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc1702.txt>. Acesso em: 2000

SIMPSON, W. RFC1853: IP in IP Tunneling. Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc1853.txt>. Acesso em: 2000

PERKINS, C. RFC2003: IP Encapsulation within IP. Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc2003.txt>. Acesso em: 2000

PERKINS, C. RFC2004: Minimal Encapsulation within IP. Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc2004.txt>. Acesso em: 2000

HAMZEH, K.; Pall, G.; VERTHEIN, W.; TAARUD, J.; LITTLE, W.; ZORN, G. RFC2637: Point-to-Point Tunneling Protocol. Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc2637.txt>. Acesso em: 2000

ROMKEY, J. L. RFC1055: Nonstandard for transmission of IP datagrams over serial lines: SLIP. Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc1055.txt>. Acesso em: 2000

MCGREGOR, G. RFC1332: The PPP Internet Protocol Control Protocol (IPCP). Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc1332.txt>. Acesso em: 2000

SIMPSON, W. RFC1331: The Point-to-Point Protocol (PPP) for the Transmission of Multi-protocol Datagrams over Point-to-Point Links. Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc1331.txt>. Acesso em: 2000

SIMPSON, W. RFC1548: The Point-to-Point Protocol (PPP). Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc1548.txt>. Acesso em: 2000

SIMPSON, W.; Ed. RFC1570: PPP LCP Extensions. Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc1570.txt>. Acesso em: 2000

SIMPSON, W.; Ed. RFC1661: The Point-to-Point Protocol (PPP). Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc1661.txt>. Acesso em: 2000

SCHNEIDER, K.; VENTERS S.. RFC1963: PPP Serial Data Transport Protocol (SDTP). Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc1963.txt>. Acesso em: 2000

REYNOLDS, J.; POSTEL J. RFC1700: Assigned Numbers. Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc1700.txt>. Acesso em: 2000

HAMZEH, K et al. RFC2637: Point-to-Point Tunneling Protocol . Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc2637.txt>. Acesso em: 2000

BRADEN, R. RFC1122: Requirements for Internet hosts - communication layers. Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc1122.txt>. Acesso em: 2000

TURNER, G.; KOMARINSKI, M. F. Remote Serial Console HOWTO. Disponível em: <http://www.linux.org/docs/ldp/howto/Remote-Serial-Console-HOWTO/index.html>. Acesso em: 2000

LAWYER, David S. Serial HOWTO. Disponível em: <http://www.linux.org/docs/ldp/howto/Serial-HOWTO.html>. Acesso em: 2000

Open Source Technology Group. Tcl SourceForge Project. Disponível em: <http://tcl.sourceforge.net/>. Acesso em: 2000

ANEXO I - CÓDIGO FONTE DO TUNELADOR

A seguir apresentamos o código fonte deste que é um processo tunelador funcional, porém simplificado, o qual usaremos como base para todo o desenvolvimento futuro. O código é aqui disponibilizado por tratar-se de um código de domínio público, que deve ser, quando finalizado, publicado sob licença GPL (Gnu Public License) e aberta a todo e qualquer uso. Uma versão deve ser publicada ainda como código aberto, podendo também ser modificado conforme as necessidades de quem utilizar.

O principal motivo de apresentá-lo aqui é para garantir uma cópia de segurança não eletrônica.

```
#!/usr/local/bin/wish

# To Do List:
#############
# status messages
# connection checking
# implement other menu functions
# implement other filter specifics
# implement config defaults
# implement control-c

# Filters Check List:
#####################
# proc initConfig
# proc setFilter
# proc startSocket
# proc acceptFilter
# proc handleFilter

####################
proc initConfig {} {
        global c w

        # Config Hash - Defaults
        ########################################################

        set c(socketPort)                       {2013}

        set c(serialPort)                       {COM1}
        set c(serialBaud)                       {9600}
        set c(serialParity)             {No}
        set c(serialData)                       {8}
        set c(serialStop)                       {1}
```

```
    set c(serialFlow)                   {None}

set c(bufferSize)           {1000}
    set c(buffer)                   {}

    set c(serialPointer)            {}
    set c(socketPointer)            {}

# Titles
    ######################################################

    set c(serialConfigPortLabel)        {Serial Port :}
    set c(serialConfigBaudLabel)            {Baud Rate :}
    set c(serialConfigParityLabel)      {Parity :}
    set c(serialConfigDataLabel)        {Data Bits :}
    set c(serialConfigStopLabel)        {Stop Bits :}
    set c(serialConfigFlowLabel)        {Flow Control :}

    set c(serialStartButton)            {Serial Start}
    set c(serialStopButton)                 {Serial Stop}
    set c(socketStartButton)            {Server Start}
    set c(socketStopButton)                 {Server Stop}

    set c(socketPrompt)                 {Bong! 1.0 >}

    set c(serialStatusTitle)            {Serial Status :}
    set c(socketStatusTitle)            {Socket Status :}

    set c(serialStatus)                   {Disconnected}
    set c(socketStatus)                 {Disconnected}

# Filters
    ######################################################
```

```
    set c(filters,socketSerial)         {0}
    set c(filters,serialSocket)         {0}
    set c(filters,stamp)              {0}
    set c(filters,pontech)             {0}
    set c(filters,seetron)            {0}
    set c(filters,handyboard)           {0}

# Pontech
    ########################################################

    set c(pontech,servo1)               {0}
    set c(pontech,servo2)               {0}
    set c(pontech,servo3)               {0}
    set c(pontech,servo4)               {0}
    set c(pontech,servo5)               {0}
    set c(pontech,servo6)                  {0}
    set c(pontech,servo7)               {0}
    set c(pontech,servo8)               {0}

    # Widget Hash
    ########################################################

    set w(mainFrameTitle)                   {Bong!}
    set w(mainFrameGeometry)                {+695+315}
    #set w(mainFrameGeometry)                       {+695+00}

# Frames
    ########################################################

    set w(mainFrame)                        {.}
    set w(menuFrame)                               {.menuFrame}
    set w(displayFrame)                     {.displayFrame}
    set w(serialFrame)                      {.serialFrame}
    set w(socketFrame)                      {.socketFrame}
```

```
# Menu
    ######################################################

    set w(fileMenuButton)                   {.menuFrame.file}
    set w(editMenuButton)                   {.menuFrame.edit}
    set w(viewMenuButton)                   {.menuFrame.view}
    set w(optionsMenuButton)            {.menuFrame.options}
    set w(filtersMenuButton)            {.menuFrame.filters}
    set w(helpMenuButton)                   {.menuFrame.help}

    set w(fileMenu)                         {.menuFrame.file.menu}
    set w(editMenu)                         {.menuFrame.edit.menu}
    set w(viewMenu)                         {.menuFrame.view.menu}
    set w(optionsMenu)                    {.menuFrame.options.menu}
    set w(filtersMenu)                  {.menuFrame.filters.menu}
    set w(helpMenu)                         {.menuFrame.help.menu}

# Display
    ######################################################

    set w(displayText)                  {.displayFrame.text}
    set w(displayScroll)                {.displayFrame.scroll}

# Serial
    ######################################################

    set w(serialButtonFrame)            {.serialFrame.buttonFrame}
    set w(serialConfigFrame)            {.serialFrame.configFrame}
    set w(serialConfigPortFrame)
    {.serialFrame.configFrame.portFrame}
    set w(serialConfigBaudFrame)
    {.serialFrame.configFrame.baudFrame}
    set w(serialConfigParityFrame)      {.serialFrame.configFrame.parityFrame}
```

```
set w(serialConfigDataFrame)
{.serialFrame.configFrame.dataFrame}

set w(serialConfigStopFrame)
{.serialFrame.configFrame.stopFrame}

set w(serialConfigFlowFrame)
{.serialFrame.configFrame.flowFrame}

set w(serialStatusFrame)                {.serialFrame.statusFrame}


set w(serialConfigPortLabel)
{.serialFrame.configFrame.portFrame.label}

set w(serialConfigBaudLabel)
{.serialFrame.configFrame.baudFrame.label}

set w(serialConfigParityLabel)
{.serialFrame.configFrame.parityFrame.label}

set w(serialConfigDataLabel)
{.serialFrame.configFrame.dataFrame.label}

set w(serialConfigStopLabel)
{.serialFrame.configFrame.stopFrame.label}

set w(serialConfigFlowLabel)
{.serialFrame.configFrame.flowFrame.label}


set w(serialConfigPortOptions)
{.serialFrame.configFrame.portFrame.options}

set w(serialConfigBaudOptions)
{.serialFrame.configFrame.baudFrame.options}

set w(serialConfigParityOptions)
{.serialFrame.configFrame.parityFrame.options}

set w(serialConfigDataOptions)
{.serialFrame.configFrame.dataFrame.options}

set w(serialConfigStopOptions)
{.serialFrame.configFrame.stopFrame.options}

set w(serialConfigFlowOptions)
{.serialFrame.configFrame.flowFrame.options}


set w(serialStartButton)                {.serialFrame.buttonFrame.start}

set w(serialStopButton)                 {.serialFrame.buttonFrame.stop}


set w(serialStatusTitle)                {.serialFrame.statusFrame.title}
```

```
        set w(serialStatusLabel)                {.serialFrame.statusFrame.label}

    # Socket
        ########################################################

        set w(socketButtonFrame)                {.socketFrame.buttonFrame}
        set w(socketConfigFrame)                {.socketFrame.configFrame}
        set w(socketStatusFrame)                {.socketFrame.statusFrame}

        set w(socketLabel)                      {.socketFrame.configFrame.label}
        set w(socketEntry)                      {.socketFrame.configFrame.entry}

        set w(socketStartButton)                {.socketFrame.buttonFrame.start}
        set w(socketStopButton)                         {.socketFrame.buttonFrame.stop}

        set w(socketStatusTitle)                {.socketFrame.statusFrame.title}
        set w(socketStatusLabel)                {.socketFrame.statusFrame.label}
}

#####################
proc initDisplay {} {
        global c w

        # Window
        ########################################################

        wm geometry $w(mainFrame)       $w(mainFrameGeometry)
        wm title        $w(mainFrame)           $w(mainFrameTitle)

        # Frames
        ########################################################

        frame $w(menuFrame) -relief raised -bd 2
        frame $w(displayFrame)
```

```
        frame $w(serialFrame)
        frame $w(socketFrame)

        frame $w(serialButtonFrame) -relief ridge -bd 2
        frame $w(serialConfigFrame) -relief ridge -bd 2
        frame $w(serialStatusFrame) -relief ridge -bd 2

        frame $w(serialConfigPortFrame)
        frame $w(serialConfigBaudFrame)
        frame $w(serialConfigParityFrame)
        frame $w(serialConfigDataFrame)
        frame $w(serialConfigStopFrame)
        frame $w(serialConfigFlowFrame)

        frame $w(socketButtonFrame) -relief ridge -bd 2
        frame $w(socketConfigFrame) -relief ridge -bd 2
        frame $w(socketStatusFrame) -relief ridge -bd 2

        pack $w(menuFrame) -side top -fill x -expand 1
        pack $w(displayFrame) -side top -fill both -expand 1
        pack $w(serialFrame) $w(socketFrame) -side left -fill both -expand 1

        pack $w(serialConfigFrame) -side top -fill x
        pack $w(serialStatusFrame) -side bottom -fill x
        pack $w(serialButtonFrame) -side bottom -fill x

        pack $w(serialConfigPortFrame) $w(serialConfigBaudFrame)
$w(serialConfigParityFrame) \
            $w(serialConfigDataFrame) $w(serialConfigStopFrame)
$w(serialConfigFlowFrame) \
            -side top -fill both -expand 1

        pack $w(socketConfigFrame) -side top -fill x
        pack $w(socketStatusFrame) -side bottom -fill x
```

```
        pack $w(socketButtonFrame) -side bottom -fill x

        # Menus
        ########################################################

        menubutton $w(fileMenuButton) -menu $w(fileMenu) -text {File} -underline 0
        menubutton $w(editMenuButton) -menu $w(editMenu) -text {Edit} -underline 0
        menubutton $w(viewMenuButton) -menu $w(viewMenu) -text {View} -underline 0
        menubutton $w(optionsMenuButton) -menu $w(optionsMenu) -text {Options} -underline 0
        menubutton $w(filtersMenuButton) -menu $w(filtersMenu) -text {Filters} -underline 0
        menubutton $w(helpMenuButton) -menu $w(helpMenu) -text {Help} -underline 0

        menu $w(fileMenu) -tearoff 0
        menu $w(editMenu) -tearoff 0
        menu $w(viewMenu) -tearoff 0
        menu $w(optionsMenu) -tearoff 0
        menu $w(filtersMenu) -tearoff 0
        menu $w(helpMenu) -tearoff 0

        $w(fileMenu) add command -label {New} -command {}
        $w(fileMenu) add command -label {Open} -command {}
        $w(fileMenu) add command -label {Save} -command {}
        $w(fileMenu) add command -label {Exit} -command {exit}

        $w(editMenu) add command -label {Cut} -command {}
        $w(editMenu) add command -label {Copy} -command {}
        $w(editMenu) add command -label {Paste} -command {}

        $w(viewMenu) add command -label {Logs} -command {}

        $w(optionsMenu) add command -label {Config} -command {}
```

```
# Filters
    ########################################################

    $w(filtersMenu) add command -label {Basic Socket Read / Serial Write} -command {setFilter socketSerial}
    $w(filtersMenu) add command -label {Basic Serial Read / Socket Write} -command {setFilter serialSocket}
    $w(filtersMenu) add command -label {Stamp Basic} -command {setFilter stamp}
    $w(filtersMenu) add command -label {Pontech Commands} -command {setFilter pontech}
    $w(filtersMenu) add command -label {Seetron Commands} -command {setFilter seetron}
    $w(filtersMenu) add command -label {Handy Board Interactive C} -command {setFilter handyboard}

    $w(helpMenu) add command -label {About} -command {}

    pack $w(fileMenuButton) $w(editMenuButton) $w(viewMenuButton) $w(optionsMenuButton) $w(filtersMenuButton) -side left
    pack $w(helpMenuButton) -side right

    # Display
    ########################################################

    text $w(displayText) -yscrollcommand "$w(displayScroll) set" -background #c0c0c0 -height 10 -width 50 -setgrid 1
    scrollbar $w(displayScroll) -command "$w(displayText) yview"

    pack $w(displayScroll) -side right -fill y
    pack $w(displayText) -fill both -expand 1

    # Serial
    ########################################################

    button $w(serialStartButton) -command startSerial -text $c(serialStartButton) -width 10
```

```
        button $w(serialStopButton) -command stopSerial -text $c(serialStopButton) -width
10
        pack $w(serialStartButton) $w(serialStopButton) -side left -fill x -expand 1


        label $w(serialConfigPortLabel) -text $c(serialConfigPortLabel) -anchor w -justify left
-width 10
        label $w(serialConfigBaudLabel) -text $c(serialConfigBaudLabel) -anchor w -justify
left -width 10
        label $w(serialConfigParityLabel) -text $c(serialConfigParityLabel) -anchor w -justify
left -width 10
        label $w(serialConfigDataLabel) -text $c(serialConfigDataLabel) -anchor w -justify
left -width 10
        label $w(serialConfigStopLabel) -text $c(serialConfigStopLabel) -anchor w -justify
left -width 10
        label $w(serialConfigFlowLabel) -text $c(serialConfigFlowLabel) -anchor w -justify
left -width 10


        tk_optionMenu $w(serialConfigPortOptions) $c(serialPort) /dev/ttyS0 /dev/ttyS1
COM1 COM2
        tk_optionMenu $w(serialConfigBaudOptions) $c(serialBaud) 1200 2400 4800 9600
19200
        tk_optionMenu $w(serialConfigParityOptions) $c(serialParity) No Odd Even
        tk_optionMenu $w(serialConfigDataOptions) $c(serialData) 8 7 6
        tk_optionMenu $w(serialConfigStopOptions) $c(serialStop) 1 2
        tk_optionMenu $w(serialConfigFlowOptions) $c(serialFlow) None Hardware
{Xon/Xoff}


        pack $w(serialConfigPortLabel) -side left
        pack $w(serialConfigPortOptions) -side right
        pack $w(serialConfigBaudLabel) -side left
        pack $w(serialConfigBaudOptions) -side right
        pack $w(serialConfigParityLabel) -side left
        pack $w(serialConfigParityOptions) -side right
        pack $w(serialConfigDataLabel) -side left
        pack $w(serialConfigDataOptions) -side right
        pack $w(serialConfigStopLabel) -side left
        pack $w(serialConfigStopOptions) -side right
```

```
        pack $w(serialConfigFlowLabel) -side left
        pack $w(serialConfigFlowOptions) -side right

        label $w(serialStatusTitle) -text $c(serialStatusTitle) -anchor w -justify left -width 10
        label $w(serialStatusLabel) -textvariable $c(serialStatus) -anchor e -justify right -width 10

        pack $w(serialStatusTitle) -side left -fill x -expand 1
        pack $w(serialStatusLabel) -side left -fill x -expand 1

        # Socket
        ##########################################################

        button $w(socketStartButton) -command startSocket -text $c(socketStartButton) -width 10
        button $w(socketStopButton) -command stopSocket -text $c(socketStopButton) -width 10
        pack $w(socketStartButton) $w(socketStopButton) -side left -fill x -expand 1

        label $w(socketLabel) -text "Socket Port:"  -anchor w -justify left -width 10
        entry $w(socketEntry) -textvariable c(socketPort) -width 5 -justify center

        pack $w(socketLabel) -side left
        pack $w(socketEntry) -side left -fill both -expand 1

        label $w(socketStatusTitle) -text $c(socketStatusTitle) -anchor w -justify left -width 10
        label $w(socketStatusLabel) -textvariable $c(socketStatus) -anchor e -justify right -width 10

        pack $w(socketStatusTitle) -side left -fill x -expand 1
        pack $w(socketStatusLabel) -side left -fill x -expand 1
}

#########################
```

```
proc setFilter {filter} {
        global c w

    foreach {index value} [array get c {filters,*}] {
          set c($index) {0}
    }

    incr c(filters,$filter)

        $w(displayText) insert end "\nSet Filter: $filter.\n"
        $w(displayText) yview end
        update idletasks
}

######################
proc startSocket {} {
        global c w

        $w(displayText) insert end "\nSocket Starting...\n"
        $w(displayText) yview end
        update idletasks

        if { [set c(filters,socketSerial)] } {

                set c(socketPointer) [socket -server acceptReadableSocket $c(socketPort)]

            $w(displayText) insert end "Filter: Socket Serial.\n"
            $w(displayText) insert end "Socket Port: $c(socketPort).\n"
            $w(displayText) insert end "Socket Pointer: $c(socketPointer).\n"

        } elseif { [set c(filters,serialSocket)] } {

                set c(socketPointer) [socket -server acceptWritableSocket $c(socketPort)]
```

```
            $w(displayText) insert end "Filter: Serial Socket.\n"
            $w(displayText) insert end "Socket Port: $c(socketPort).\n"
            $w(displayText) insert end "Socket Pointer: $c(socketPointer).\n"

        } elseif { [set c(filters,stamp)] } {

        } elseif { [set c(filters,pontech)] } {

                set c(socketPointer) [socket -server acceptPontech $c(socketPort)]

            $w(displayText) insert end "Filter: Pontech.\n"
            $w(displayText) insert end "Socket Port: $c(socketPort).\n"
            $w(displayText) insert end "Socket Pointer: $c(socketPointer).\n"

        } elseif { [set c(filters,seetron)] } {

        } elseif { [set c(filters,handyboard)] } {

    } else {
            $w(displayText) insert end "\nERROR: No filter selected.\n"
    }

    $w(displayText) yview end
    update idletasks

        vwait c(events)
}

################################################
proc acceptReadableSocket {sock clientIp port} {
        global c w

        fileevent $sock readable [list handleReadableSocket $sock $clientIp]
        fconfigure $sock -buffering none -blocking 0
```

```
        puts $sock $c(socketPrompt)
        $w(displayText) insert end "Client connected: $clientIp.\n"
        $w(displayText) yview end
        update idletasks
}

################################################
proc acceptWritableSocket {sock clientIp port} {
        global c w

        fileevent $sock writable [list handleWritableSocket $sock $clientIp]
        fconfigure $sock -buffering none -blocking 0

        puts $sock $c(socketPrompt)
        $w(displayText) insert end "Client connected: $clientIp.\n"
        $w(displayText) yview end
        update idletasks
}

################################################
proc acceptPontech {sock clientIp port} {
        global c w

        fileevent $sock readable [list handlePontech $sock $clientIp]
        fconfigure $sock -buffering none -blocking 0

        puts $sock $c(socketPrompt)

        $w(displayText) insert end "Pontech Client connected: $clientIp.\n"

        #$w(displayText) insert end "Client IP:$clientIp.\n"
        #$w(displayText) insert end "Socket:$sock.\n"
        #$w(displayText) insert end "Socket Pointer:$c(socketPointer).\n"
```

```
        $w(displayText) yview end
        update idletasks
}

##########################################
proc handleReadableSocket {sock clientIp} {
        global c w

        append c(buffer) [gets $sock]

        if {[eof $sock]} {
                close $sock
      set c(buffer) {}
                $w(displayText) insert end "\nERROR: EOF.\n"

        } elseif { [string length $c(buffer)] > $c(bufferSize) } {
      close $sock
      set c(buffer) {}
      $w(displayText) insert end "\nERROR: Buffer Overflow.\n"

   } elseif { [regexp {^(quit)[\n]*$} $c(buffer) match m1] } {
      close $sock
      set c(buffer) {}
      $w(displayText) insert end "Client Disconnected.\n"

   } elseif { [regexp {(.+)[\n]*$} $c(buffer) match m1] } {
     set c(buffer) {}
     $w(displayText) insert end "Client: $m1.\n"
   }

   $w(displayText) yview end
   update idletasks
}
```

```
##########################################
proc handleWritableSocket {sock clientIp} {
        global c w
}

##########################################
proc handlePontech {sock clientIp} {
        global c w

        append c(buffer) [gets $sock]

        if {[eof $sock]} {
                close $sock
      set c(buffer) {}
                $w(displayText) insert end "\nClient disconnected.\n"

        } elseif { [string length $c(buffer)] > $c(bufferSize) } {
                close $sock
      set c(buffer) {}
                $w(displayText) insert end "\nERROR: Buffer Overflow.\n"

   } elseif {[regexp {^([0-9]+):([0-9]+):([0-9]+):([0-9]+):([0-9]+):([0-9]+):([0-9]+):([0-9]+)[\n]*$} \
                $c(buffer) match c1 c2 c3 c4 c5 c6 c7 c8]} {

      # Buffer Action
      ########################################################

                if {$c1 != $c(pontech,servo1)} {
                        set c(pontech,servo1) $c1
                        puts $c(serialPointer) "BD1 SV1 M$c(pontech,servo1)"
                } elseif {$c2 != $c(pontech,servo2)} {
                        set c(pontech,servo2) $c2
```

```
                puts $c(serialPointer) "BD1 SV2 M$c(pontech,servo2)"
            } elseif {$c3 != $c(pontech,servo3)} {
                set c(pontech,servo3) $c3
                puts $c(serialPointer) "BD1 SV3 M$c(pontech,servo3)"
            } elseif {$c4 != $c(pontech,servo4)} {
                set c(pontech,servo4) $c4
                puts $c(serialPointer) "BD1 SV4 M$c(pontech,servo4)"
            } elseif {$c5 != $c(pontech,servo5)} {
                set c(pontech,servo5) $c5
                puts $c(serialPointer) "BD1 SV5 M$c(pontech,servo5)"
            } elseif {$c6 != $c(pontech,servo6)} {
                set c(pontech,servo6) $c6
                puts $c(serialPointer) "BD1 SV6 M$c(pontech,servo6)"
            } elseif {$c7 != $c(pontech,servo7)} {
                set c(pontech,servo7) $c7
                puts $c(serialPointer) "BD1 SV7 M$c(pontech,servo7)"
            } elseif {$c8 != $c(pontech,servo8)} {
                set c(pontech,servo8) $c8
                puts $c(serialPointer) "BD1 SV8 M$c(pontech,servo8)"
            }

        set c(buffer) {}
            $w(displayText) insert end "Client:$c1:$c2:$c3:$c4:$c5:$c6:$c7:$c8.\n"

            after 100
    }

    $w(displayText) yview end
    update idletasks
}

####################
proc stopSocket {} {
        global c w
```

```
        close $c(socketPointer)
        $w(displayText) insert end "Socket Stopped.\n"
        $w(displayText) yview end
        update idletasks
}

#####################
proc startSerial {} {
        global c w
        set c(serialPointer) [open $c(serialPort) {WRONLY}]
        fconfigure $c(serialPointer) -mode "9600,n,8,1"
        fconfigure $c(serialPointer) -buffering none
        fconfigure $c(serialPointer) -blocking 0
        $w(displayText) insert end "Serial Started: $c(serialPort):$c(serialPointer).\n"
        $w(displayText) yview end
        update idletasks
}

####################
proc stopSerial {} {
        global c w
        close $c(serialPointer)
        $w(displayText) insert end "Serial Stopped.\n\n"
        $w(displayText) yview end
        update idletasks
}

############
#   Main
############
initConfig
initDisplay
```