UNIVERSIDADE DE SÃO PAULO
ESCOLA POLITÉCNICA
DEPARTAMENTO DE ENGENHARIA MECÂNICA

PMC 581 : Projeto Mecânico II

Prof. Orientador: Marcos R. P. Barreto

SCADA em JAVA

9,0 (nove)

André Galhego Arantes - NUSP: 569231

São Paulo, 17 de Dezembro de 1998

Índice

1 Objetivo

O presente projeto tem como objetivo a inclusão de uma nova estrutura para o modelo tradicional de SCADA (*Supervisory Control and Data Acquisition*), introduzindo novas tecnologias e tendências que estão surgindo. Este projeto visa a implementação de um SCADA utilizando componentes que estão sendo desenvolvidos e que possuem o sistema operacional *Java OS* em seus *firmware*. Estes componentes ainda não estão disponíveis comercialmente, porém com este projeto espera-se estar adiantado quando os componentes estiverem disponíveis para plena utilização. Assim, serão feitos modelos em *software* dos mesmos para interagirem com o sistema desenvolvido, de modo que a substituição destes pelos itens físicos se dê de uma maneira suave e sem grandes disparidades.

O documento irá comparar o sistema convencional com o proposto, assim como explicar um pouco da linguagem Java e suas funcionalidades de maior relevância para o projeto, e a descrição do funcionamento dos componentes do sistema como um todo.

2. Scada Convencional

Usando uma definição geral, SCADA (Supervisory control and data acquisition) é um sistema de medida e controle que consiste em sensores para aquisição de estados e geração dos sinais correspondentes, CLP's que captam estes sinais para processar a lógica do sistema e gerar sinais de controle para os atuadores, e uma estação central de acúmulo de dados com terminais para monitoração dos processos. Um esquema simples para o entendimento desta definição se encontra na figura 2.1.

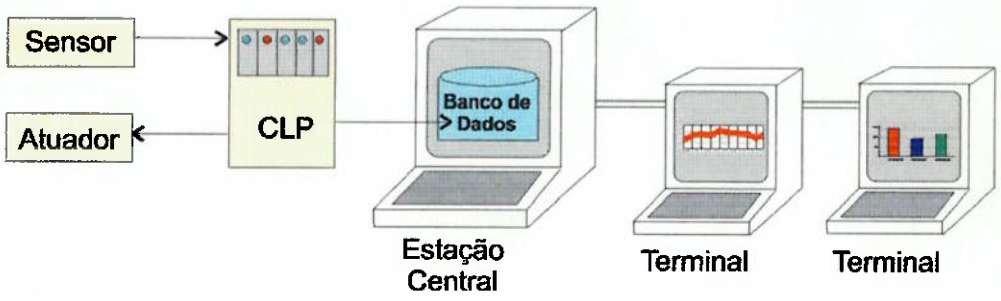

Figura 2.1. Esquema simples de um SCADA atual

O funcionamento deste sistema se dá da seguinte forma: os sensores possuem valores e disponibilizam estas informações para a CLP através de protocolos de camadas baixas de comunicação. Esta CLP processa estas informações e gera sinais de controle para modificar os valores dos atuadores, de modo a produzir o comportamento desejado do sistema. Os dados provenientes dos sensores também são traduzidos de uma forma que uma estação central (um computador) possa armazená-los em um banco de dados que contém as informações de todos os sensores. Cada terminal que esteja interessado na monitoração dos sensores, atualiza seus dados através de *polling*, ou seja, com uma freqüência pré-definida, acessam o banco de dados da estação central e coletam o dados referente aos sensores que deseja. Com estes novos dados, podem atualizar seus gráficos e disponibilizar as informações aos operadores.

A estação central e os terminais estão ligados geralmente através de uma LAN (*Local Area Network*), e é preciso que haja uma compatibilidade entre os aplicativos

dos terminais e do banco de dados gerado pela estação central. Logo, quaisquer desenvolvimento de um novo sistema para esta monitoração deve levar em conta as plataformas existentes.

Quando um novo sensor precisa ser instalado, é preciso adquirir mais um placa para a CLP, reprogramá-la para atualizar a lógica do processo, assim como reestruturar o banco de dados da estação central.

É claro que um SCADA envolve mais do que foi descrito acima, porém as informações são suficientes para que se entenda a proposta que será feita, assim como comparações mais específicas poderão ser feitas. Com isto, os pontos principais a serem ressaltados são:

- níveis de protocolos diferentes na estrutura, necessitando trabalhar os mesmos para garantir a disponibilização dos dados para todos os componentes;
- alta dedicação dos processadores dos terminais para efetuar o *polling* ;
- necessidade da compatibilidade dos sistemas de monitoração entre si e o banco de dados da estação central;
- alteração física da CLP para cada sensor ou atuador instalado.

3. Sistema Proposto

Uma vez que foi descrito o sistema convencional para a implementação de um SCADA, será agora descrita a proposta de mudança. Primeiramente, para um maior entendimento do projeto, será descrita sucintamente a linguagem Java, ressaltando os pontos principais relevantes para o esclarecimento da funcionalidade do sistema.

3.1. Linguagem Java

A linguagem Java foi desenvolvida com a intenção de ser um sistema facilmente programável sem muito treinamento profundo, e que alavancasse os padrões atuais de uso, ou seja, visando trocar programas 'gordos' e de arquitetura rígida por pequenos aplicativos específicos para determinadas tarefas. Ele foi projetado para suportar aplicações em rede, e é neste ambiente onde demonstra se usufrui mais intensamente de seus recurso.

Sua lógica é orientada a objetos, que visam representar algo real, com características e comportamento. Um objeto possui variáveis, que indicam seu estado, e métodos, que são as únicas maneiras de alterar suas variáveis, permitindo definir comportamentos do objeto. As variáveis formam o núcleo do objeto, enquanto os métodos circundam o núcleo, isolando-o de outros objetos no programa. Os objetos de software interagem e se comunicam entre si através de mensagens, que contêm os seguintes elementos:

- objeto para o qual a mensagem é endereçada
- nome do método a ser executado
- parâmetros requeridos pelo método

Assim, é possível perceber que fica mais simples trabalhar no desenvolvimento de sistemas, uma vez que o mesmo é formado por objetos bem definidos. É possível ter uma visão global do sistema, percebendo a interação entre os objetos, assim

como distribuir melhor o trabalho de desenvolvimento, quando for necessário desenvolver a arquitetura interna de cada objeto.

O Java possui as seguintes funcionalidades, que são essenciais para o projeto:

- **Multiplataforma** - a linguagem foi desenvolvida de maneira que o código gerado pudesse ser interpretado por quaisquer máquinas. A maneira encontrada para a realização disto foi a divisão da geração de códigos em duas etapas: geração de *bytecodes* e interpretação em *real-time*. Para o melhor entendimento, é preciso antes falar sobre a *Java Virtual Machine*. A JVM é uma idealização de máquina, para qual os programas irão ser compilados. Esta máquina é única, então sempre os códigos que forem escritos, independente de qual plataforma os estão gerando, serão compilados e formarão os *bytecodes*, que são os códigos a serem interpretados pela JVM. Cada plataforma deve possuir um interpretador Java dedicado à esta plataforma. Quando de desejar rodar um programa, os *bytecodes* serão interpretados em tempo real neste interpretador. Por exemplo: uma plataforma Machintosh possui um "interpretador Java para Machintosh" e uma UNIX, possui um "interpretador Java para UNIX". Assim, se um programa for feito em um PC com Windows, ao se compilar, serão gerados arquivos contendo os *bytecodes*. Estes arquivos podem ser distribuídos para os usuários de Machintosh ou UNIX, e quando forem rodar o programa, os respectivos interpretadores irão gerar os comandos para suas máquinas em tempo real.

- **RMI (*Remote Method Invocation*)** - é o maneira pela qual os objetos em Java conseguem comunicar entre si remotamente. Com isto, é possível que se faça um sistema distribuído e a comunicação dos componentes é feita remotamente, através de protocolos de nível mais alto (ex: TCP/IP). Juntamente com a característica multiplataforma da linguagem, este conceito faz com que se possa desenvolver um sistema a ser distribuído em uma rede sem levar em conta quais plataformas estão envolvidas na mesma.

- ***Java Beans*** - por ser uma linguagem orientada a objetos, o Java permite que se reutilizem os mesmos para aplicação distintas. Uma vez que o objeto foi devidamente encapsulado pelos seus métodos, é possível que o código possa ser reutilizado e o objeto irá manter sempre o mesmo comportamento. Muitos destes objetos são gráficos, e possuem características visíveis, como tamanho, cores, etc... Um *Bean* é um objeto que pode ser considerado como um componente gráfico, que pode ser visualizado e ter suas características modificadas visualmente durante a etapa de programação. Um exemplo simples pode ser o de um botão, que em um compilador visual, pode ser arrastado de um palheta de ferramentas até a janela que se está construindo, e pode ter suas características como cor, tamanho, legenda, alterados durante a modelagem da janela.

Com estas exposições, agora o sistema proposto pode ser perfeitamente entendido, assim como suas características desejadas poderão ser asseguradas.

3.2. Visão Geral do Sistema Proposto

O sistema proposto consiste basicamente da integração de novos componentes que vêm sido desenvolvidos, os quais possuem um sistema operacional denominado Java OS (que estaria presente possivelmente em *firmware*), e que podem ser utilizados de maneira a se aproveitar todos os benefícios que a linguagem Java pode oferecer. Como estes componentes não estão ainda disponíveis comercialmente, foram idealizados objetos que possuem características e comportamento o mais próximo possível com a realidade. Assim, uma posterior substituição de um objeto lógico um Java por um componente real com características em Java OS não terá grandes discrepâncias.

Um esquema geral da integração dos componentes pode ser visto conforme a figura abaixo:

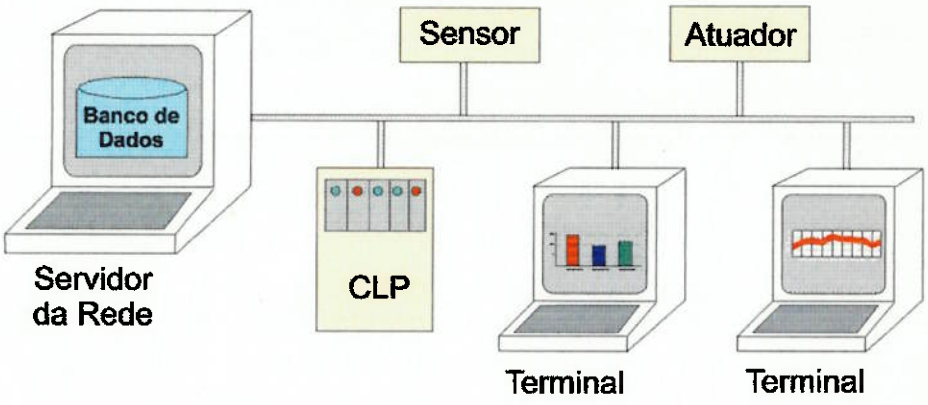

Figura 3.2.1. Esquema geral do sistema proposto

Basicamente podemos pensar em uma rede comum, onde há um servidor para o gerenciamento da mesma e computadores ligados ao backbone compartilhando as informações entre si. Esta comunicação é possível pois cada um possui um endereço, gerenciado pelo servidor, e usam de protocolos passar as informações. Agora, podemos pensar nos componentes que possuem Java OS também ligados nesta rede. A inserção deste equipamento se faz de forma similar à de um computador: seria conectado fisicamente à rede através de um hub, e teria seu endereço registrado no servidor. A partir deste momento, seria possível detectar a presença do mesmo na rede, pois ele já possui um nível de protocolo que permite a integração com os computadores ligados ao backbone.

Estes componentes podem ser considerados como objetos Java, devido ao seu *firmware*, e a comunicação com eles seria feita da mesma forma que foi descrito no item anterior, conforme ilustrado na figura 3.1. O grande ponto está na forma e em quais situações que entram em contato. Cada componente se registra no equipamento desejado, e este só mandará informações para os participantes de sua lista interna de registros, através do RMI. Por exemplo: quando um computador está interessado em obter a leitura de um sensor, ele entra em contato com o mesmo e se registra na lista que o sensor possui. Quando fosse necessário informar uma alteração por parte deste sensor, ele acessaria somente quem uma vez já se registrou nele e atualizaria por conta própria os dados que o computador possui. Pode-se ressaltar que neste caso os terminais podem voltar seu processamento

para outros fins que não a atualização, pois são passivos quando os sensores alteram seus dados.

O mesmo pode se estender aos atuadores: as CLP's se registram nos sensores relevantes à sua lógica e nos atuadores, alterando os dados deste quando conveniente. Convém ressaltar que estas CLP's na verdade se diferem das atuais. Podem possuir apenas uma placa de entrada e saída e se comunicam através de protocolos de camadas mais altas dos que os atualmente utilizados. Na verdade poderiam até ser considerados como computadores que estão com a lógica de controle do sistema armazenada.

Como pode ser visto, podemos ver diferenças importantes entre os dois tipos de SCADA mostrados. Em primeiro lugar, a estação central desaparece, e a divulgação dos dados se faz de forma mais direta. Há também um aumento da confiabilidade do sistema, uma vez que no SCADA convencional uma avaria nesta acarretaria na falta de monitoração em todos os terminais, e no sistema proposto estes estão completamente independentes. Pode-se notar também uma menor intensidade de processamento e uso da rede por parte dos terminais, uma vez que não se precisa mais de *polling* e as informação virão por parte dos sensores, que alteram diretamente os dados existentes, e somente quando necessário. A característica da linguagem Java de ser multiplataforma e podermos reutilizar objetos já definidos aumenta a flexibilidade do sistema e menos complexidade na implementação da rede de comunicação entre os componentes. Vale a pena também mencionar que a inserção de um novo componente se faz de uma maneira mais simples, e a escalabilidade do sistema está assegurada. Resumindo, o novo sistema possui os seguintes pontos:

- uniformidade no nível de protocolo ao longo de todo o sistema;
- baixa dedicação dos processadores dos terminais;
- quaisquer modificação no sistema pode ser feita sem a necessidade de se levar em conta as plataformas envolvidas;
- a inserção de um novo componente na rede só altera a lógica do sistema;
- os componentes lógicos desenvolvidos podem ser facilmente reutilizáveis e graficamente modeláveis.

4. Componentes

Uma vez esclarecida a visão geral do processo, os componentes serão detalhados para um melhor entendimento de seu funcionamento e da interação entre todos no sistema. As classes foram divididas em *packages*, que é uma maneira de dizer que várias classes pertencem a um certo agrupamento devido à semelhança de funções. Os *packages* definidos são os seguintes:

- **dados:** classes que não possuem métodos, apenas agrupam tipos de dados que irão ser passados remotamente entre os componentes do sistema

- **interfaces:** são as interfaces que indicam quais métodos poderão ser utilizados remotamente pela classe que o implementar. Assim, se uma classe que possui 4 métodos implementar uma interface remota que possui 1 método, só este poderá ser ativado remotamente, e os outros 3, somente localmente

- **sensores:** classes que determinam o objeto lógico sensor e sua interface gráfica

- **clps:** classes que determinam o objeto lógico CLP e sua interface gráfica

- **atuadores:** classes que determinam o objeto lógico atuador e sua interface gráfica

- **graphs:** *package* constituído por mais três *packages*:

    - **BarGraphs:** classes que determinam o objeto lógico BarGraph e sua interface gráfica em Bean
    - **TrendGraphs:** classes que determinam o objeto lógico TrendrGraph e sua interface gráfica em Bean

- **Janelas:** classes que servirão como hospedeiras dos Beans BarGraph e TrendGraph

- **borland:** esta *package* contém outras embutidas e basicamente conta com duas classes: *XYLayout* e *XYConstraints*. Estas duas classes foram desenvolvidas pela *Borland International, Inc.*, e fazem parte do conjunto de classes que compõem o produto *JBuilder*™. Estas são utilizadas em conjunto para se poder usar coordenadas absolutas em janelas para acrescentar componentes. A primeira indica que o *layout* aceitará cooredenadas absolutas, e o segundo será usado para acrescentar um componente em uma determinada posição (x,y) e podendo utilizar um espaço (largura, altura) na janela. Como provém de um pacote fechado, não será explicada junto com os outros componentes do projeto, porém na listagem que está no final do documento pode-se ter idéia de como utilizá-las.

A seguir, serão descritos os funcionamentos das classes que compõem as *packages*, para se esclarecer o comportamento de cada um e se poder ter uma melhor idéia da interação entre os componentes no sistema. Nestas descrições não serão incluídas as linhas de código das classes, pois estas se encontram na íntegra no final do documento, e possuem comentários bem detalhados para se poder acompanhar o raciocínio da estrutura. Assim, as descrições que serão feitas explicam melhor a dinâmica de cada objeto e quaisquer dúvida quanto à implementação da lógica pode ser tirada na listagem em anexo.

4.1 Package dados

4.1.1 Classe Dados

A classe *Dados* representa um agrupamento de variáveis que serão passadas entre objetos para efetuar a inscrição na lista de componentes. Assim, uma CLP irá passar um *Dados* para um sensor e este poderá efetuar sua inscrição na lista que possui. Esta classe é constituída pelos seguintes tipos de dados:

- Acao: conterá a informação do objeto que dirá se está querendo se registrar ou se remover da lista do objeto destinatário.
- Componente: indica o nome do objeto que está mandando as informações para o registro
- TipoAtualizacao: indica se o objeto cliente irá ser atualizado orientado a eventos ou por uma taxa de amostragem
- Intervalo: corresponde ao intervalo de tempo desejado pelo componente que irá ser atualizado por taxa de amostragem

4.1.2 Classe Ponto

Assim como a classe *Dados*, esta classe somente agrupa dados a serem passados, porém neste caso os dados dizem respeito ao valor que o sensor está passando para os componentes. Suas variáveis são:

- ValorMaximo: valor numérico do máximo valor atingido pelo componente, em sua unidade coerente
- ValorMinimo: valor numérico do mínimo valor atingido pelo componente, em sua unidade coerente
- ValorAtual: valor numérico do valor atual que define o estado do componente, em sua unidade coerente
- Tempo: horário correspondente ao ValorAtual que está sendo passado para o componente requisitante
- NomeComponente: nome que o componente que está passando os dados é reconhecido na rede

4.2. Package interfaces

4.2.1. Classe Atualizacoes

A interface *Atualizações* define o método *Atualizar* como remoto. Este método será utilizado por componentes que desejem que tenha seus dados atualizados remotamente por outro componente. Assim, o componente que implementar esta interface poderá receber uma classe do tipo *Ponto*, para poder atualizar seus dados, e retornará uma mensagem para o componente que o atualizou, dando continuidade ao processo.

4.2.2. Classe Inscricoes

A interface *Inscricoes* define o método *AlteraInscricao* como remoto. Este método será utilizado por componentes que possuem listas de componentes ligados nele, e através deste poderão inserir ou retirar suas inscrições desta lista.

O servidor (o dono da lista) receberá os dados do cliente empacotados pela classe *Dados*, podendo efetuar a inserção do componente na lista, assim como sua identificação e retirada da mesma, e retornará os seus valores ao cliente pela classes *Ponto*, que será utilizada pelo cliente para atualizar seu valor de imediato.

4.3. Package sensores

4.3.1 Visão Geral

Os sensores são componentes que estão efetivamente monitorando os eventos reais que estão ocorrendo e assumindo valores para representar estes eventos. Podem tanto ser digitais, onde fornecem informações com apenas 2 valores distintos, como analógicos, disponibilizando valores intermediários dentro de um intervalo.

Os sensores digitais são utilizados para monitoração de mudança de estados. Ou seja, somente irão enviar informações aos componentes neles inscritos quando seu valor for alterado, atualizando, então, os dados mostrados nos terminais supervisores. Já os sensores analógicos poderão ser usados em duas aplicações distintas. Podem atualizar os dados sobre seu valor com uma taxa de amostragem pré definida, independente dos eventos que estão ocorrendo. Isto serviria para uma monitoração que vise a visualização do comportamento do sensor em um determinado período de tempo. Outra forma de se utilizar este tipo de sensor seria para monitoração de variações. Assim como o sensor digital, só atualizaria os dados quando registrasse uma alteração do seu valor dentro de uma margem pré estabelecida, porém neste caso seria necessário também informar o novo valor atingido, ou o quanto o valor mudou.

Para o projeto, idealizou-se o sensor como um elemento que possui seu valor alterado por um agente externo, possui um lista de componentes registrados que é modificada pelos mesmos, e que só é ativo no que se diz respeito ao *broadcasting* do seu valor atual, ou seja, na atualização ativa dos dados dos componentes nele interessados. Para simular este agente externo, interface gráfica possui campos onde pode-se entrar manualmente com os valores desejados para simular a reação do sensor.

4.3.2 Classe SensorFrame

Esta classe é responsável pela interface gráfica do Sensor. Os métodos que definem a lógica do controle da lista de interessados, assim como suas atualização estão na classe Sensor. Ambas classes são inicializadas em cascata e trabalham em paralelo. Então, a separação se dá apenas no armazenamento dos arquivos, mas ambos estão ligados logicamente e seu funcionamento se dá como se fossem uma só, ficando transparente para o usuário esta separação.

Uma vez inicializada, esta classe fica com a seguinte aparência:

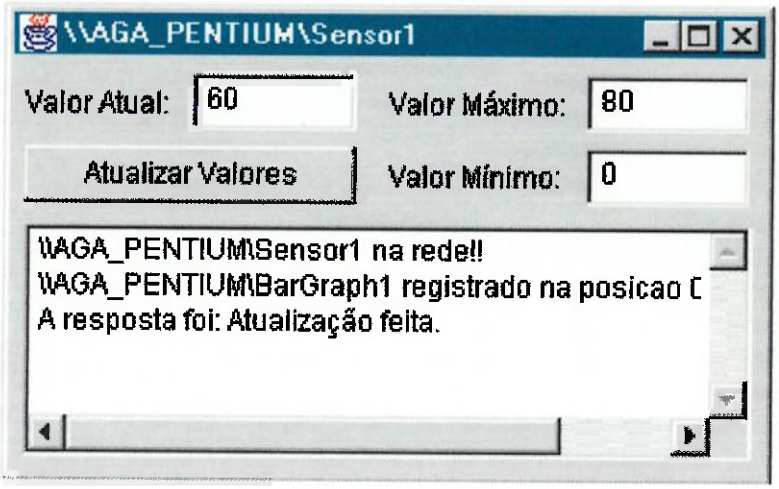

Figura 4.3.2.1. Interface gráfica do Sensor

Como se poder ver, os valores máximo, mínimo e atual do sensor podem ser entrados pelo usuário para simular o sensor. Com isto, utilizando o mesmo código, podem ser abertos vários sensores, e se estipular um fundo de escala diferente, assim como o comportamento na mudança dos valores atuais.

A área de texto indica mensagens que são impressas pelo Sensor e recebidas pelos componentes que entram em contato com o mesmo, para se ter uma monitoração dos processos que estão acontecendo.

O botão serve para atualizar os componentes que se registraram no sensor e estão interessados em ser atualizados por eventos. Assim, uma vez digitado o valor que se deseja nos campos, ao se apertar o botão, será ativada a classe Sensor, e somente a lista de componentes orientados a evento serão atualizados. Os outros, orientados a taxa de amostragem, estão sendo atualizados em *background* diretamente pela classe Sensor.

4.3.3. Classe Sensor

A classe *Sensor* contém a lógica principal de registros de componentes e envio de dados, assim como inicializa a classe SensorFrame, que é a sua GUI (Graphic User Interface).

Ao se iniciar esta classe, primeiramente o sensor será registrado na rede com um determinado nome, recuperado da entrada pelo prompt, para ser identificado. Este nome possui a localização do servidor onde se encontra. Por exemplo: entrando-se com o nome "//143.123.32.145/Sensor1" indica que o sensor vai ser reconhecido como "Sensor1" no servidor que possui o IP 143.123.32.145. Assim, o sensor está apto a ser procurado na rede para se poder efetuar inscrições e remoções em sua lista de interessados, assim como atualizar os dados dos mesmos.

O sensor, implementa a interface *Inscricoes*, e utiliza o método *AlteraInscricao* para que os componentes possam se inscrever remotamente no mesmo. O definição deste método é a seguinte:

```
public Ponto AlteraInscricao (Dados d)
```

Assim, o componente interessado em receber informações do sensor passa seus dados através da classe *Dados*, e recebe imediatamente o estado atual do sensor definido pela classe *Ponto*. Para efetuar a inscrição, o sensor possui duas listas que poderá incluir o componente: uma para orientados a evento, que serão atualizados ao se apertar o botão da interface gráfica; e outra para orientados a amostragem, que serão atualizados por *threads*.

Esta atualização ocorre da seguinte forma: um *thread* é um processo que após iniciado, ocorre em *background*, e podem ser instanciados vários ao mesmo tempo, sendo um distinto do outro através dos nomes que recebem. Assim, nesta classe, para cada componente que se inscreve na lista orientada a amostragem, será iniciado um *thread* que receberá o nome deste próprio componente e terá um taxa de atualização própria, conforme definida pela classe *Ponto* passada. Quando se deseja retirar o componente da lista, procura-se o *thread* que possui o nome do componente que requisitou a retirada, e se encerra a sua execução.

Desta maneira, o sensor pode agrupar componentes atualizados juntamente pelo acionamento do botão, ou por taxas de amostragem distintas, rodando em paralelo e que não interferem um no funcionamento do outro.

Para que o sensor consiga atualizar os componentes nele inscritos, estes devem implementar a interface *Atualizacoes*, pois assim possuirão o método *Atualizar* e o sensor poderá acioná-lo remotamente e passar os dados necessários para a atualização.

4.4. Package clps

4.4.1 Visão Geral

No sistema proposto, utilizamos a nomenclatura CLP para um componente que estaria com a lógica de controle do sistema armazenada, e que distribui ordens de comando para os atuadores para assegurar o comportamento previsto do processo. Mas este componente se difere das CLP's atuais, devido ao fato de poderem possuir apenas uma placa de entrada e saída para controlar inúmeros componentes, e não utilizarem a linguagem L para a construção de sua lógica interna. Conforme já dito, elas podem ser consideradas como computadores que possuem a lógica de controle definida por programação em nível mais alto.

Para o efetivo controle do processo, cada CLP's (no caso de se haver mais de uma) se registra somente nos sensores que possuem os dados necessários para o processamento das informações. São aproveitados os mesmos os métodos que estão sendo utilizados pelos outros componentes, uma vez que a informação que necessita do sensor é do mesmo tipo (valores máximo e mínimos, e atual). É também preciso que as CLP's se registrem nos atuadores, indicando que irão mandar sinais para a atualização dos valores dos mesmos.

Assim, com as informações disponibilizadas pelos sensores em que estão registradas, as CLP's processam os dados e geram sinais de controle para os atuadores. Conceitualmente, é isto o que se faz atualmente, porém o novo método proposto oferece uma interface menos complexa e mais flexível.

A CLP que será utilizada neste projeto foi modelada de uma maneira genérica, podendo a sua lógica de controle ser alterada sem interferir no resto do funcionamento da mesma, uma vez que ela foi completamente isolada em um método único. Em um primeiro momento, esta CLP alimenta uma lista que contém pares atuadores-sensores, ou seja, quais atuadores serão afetados quando determinado sensor mudar de estado. Uma vez feito isto, a CLP se registra nos sensores que são de interesse na continuidade de sua lógica de controle. A partir deste ponto, ela receberá os dados dos sensores e irá atualizando os estados desejados dos atuadores.

Toda esta parte de inscrição e atualização não será afetada pelo tipo de controle que a CPL irá executar, e pode-se utilizar o código geral, por exemplo, para controle PI ou para PID, bastando modificar apenas o método responsável pela "transformação" das entradas dos valores dos sensores em saídas para os atuadores.

4.4.2. Classe CLPFrame

Assim como o sensor, a CLP possui uma GUI para coordenar as suas atividades, que possui o seguinte aspecto:

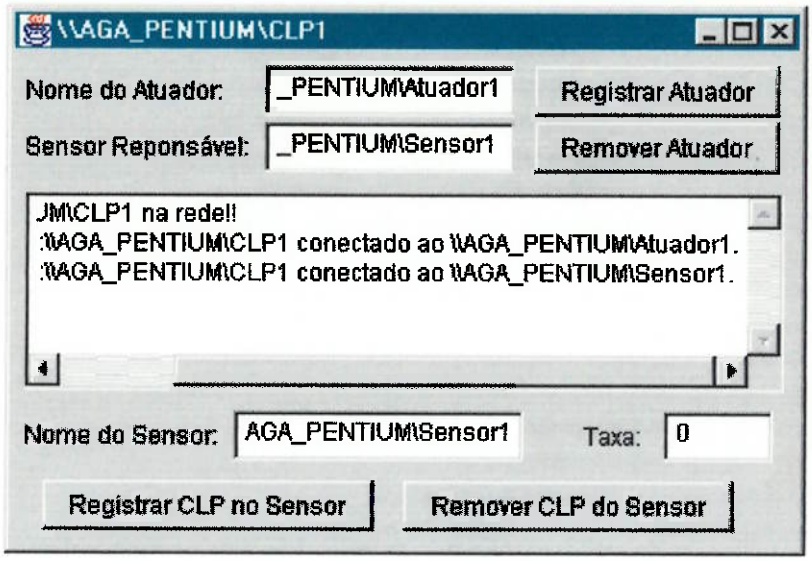

Figura 4.4.2.1. Interface gráfica da CLP

Os campos de cima serão utilizados para alimentar a lista "atuadores x sensores". Os nomes a serem entrados serão os nomes que os componentes são conhecidos na rede, e nesta etapa a CLP irá entrar em contato com o atuador para receber os seus valores máximos e mínimo, para poder utilizar em sua lógica de controle. Neste momento a CLP ainda não entrou em contato com os sensores.

Na segunda etapa, que corresponde à utilização dos campos localizados na parte de baixo da janela, a CLP irá localizar o sensor desejado e se inscrever em sua lista. O campo "Taxa" será usado para entrar com o intervalo de amostragem

para a atualização dos dados. Caso se entre com zero, estará indicando que deseja ser atualizada por eventos.

A área central será utilizada para imprimir mensagens da própria CLP, assim como mensagens recebidas pelos atuadores e sensores, para se ter idéia do andamento do processo que está ocorrendo ao fundo.

4.4.3. Classe CLP

Nesta classe está o cerne do funcionamento da CLP. Assim como o sensor, o primeiro passo executado é a CLP se registrar na rede para poder ser encontrado para as atualizações desejadas. Uma vez feito isto, será feita a alimentação da lista "atuadores x sensores" conforme explicado no item anterior, assim como a inscrição nos sensores. Como esta classe implementa a interface *Atualizacoes*, poderá atualizada remotamente pelo sensor, para receber os dados necessários para dar continuidade à sua lógica de controle. Uma vez recebidos os dados dos sensores, é percorrida a lista de atuadores para encontrar quais serão afetados por esta mudança. Depois, passam-se os valores máximo e mínimo do atuador e os valores máximo, mínimo e atual do sensor. Isto basta para que possa se calcular o valor a ser atingido pelo atuador.

Como neste ponto de desenvolvimento do projeto não é possível atribuir uma dinâmica ao sistema, devido à falta dos componentes reais, não foi desenvolvida uma lógica de controle muito rebuscada. O cálculo feito considerou apenas uma fácil visualização do funcionamento do controle. Assim, a lógica apenas consiste em conservar a posição relativa do sensor para o atuador. Ou seja, se o sensor está a 40% do seu valor total, assim deve ficar o atuador. Novamente é frisado que apesar da lógica ser simples, a modificação por uma outra mais rebuscada está assegurada, devido ao isolamento do método do controle. Definindo que este precisa apenas das entradas e saídas mencionadas, sua generalidade está também assegurada.

4.5. Package atuadores

4.5.1 Visão Geral

Os atuadores são elementos que alteram seus valores de acordo com ordens de comando que recebem das CLP's, e com isto modificam o comportamento do sistema onde estão inseridos. Eles podem ser tanto digitais como analógicos, porém possuem o mesmo tipo de método, que diz apenas o valor que deve assumir, deixando a preocupação da variedade de valores válidos para a CLP.

Na modelagem, o atuador, tal qual na realidade, é um elemento completamente passivo. A CLP entrará em contato com o mesmo para obter seus valores máximo e mínimos e se encarregará de atualizar o seu valor atual para garantir sua lógica de controle. A única diferença que poderá ocorrer entre os diversos atuadores será a gama de valores que podem assumir. Como a interface gráfica possui campos onde o usuário pode entrar com os valores máximo e mínimo do atuador, pode-se utilizar o mesmo código e abrir vários sensores ao mesmo tempo, e atribuindo-se valores diferentes, pode-se simular um sistema com vários atuadores distintos instantaneamente.

4.5.2. Classe AtuadorFrame

Conforme visto acima, a interface gráfica de um atuador servirá apenas para entrar com seus valores máximo e mínimo, assim como verificar seu estado atual. Para uma melhor percepção visual da mudança de comportamento do atuador, utilizou-se a construção de um gráfico de barra que atinge a altura correspondente ao valor desejado com a cor vermelha, contrastando com o fundo branco. Seu formato se encontra na figura 4.5.2.1. Verifica-se que o valor atual é demostrado pela altura que o gráfico representa, assim como o seu valor por escrito e horário que a CLP mandou o comando para o mesmo. Ao ser inicializado, o atuador terá o valor mínimo de "0" e máximo de "100", valor atual "0" e horário de atualização "00:00:00".

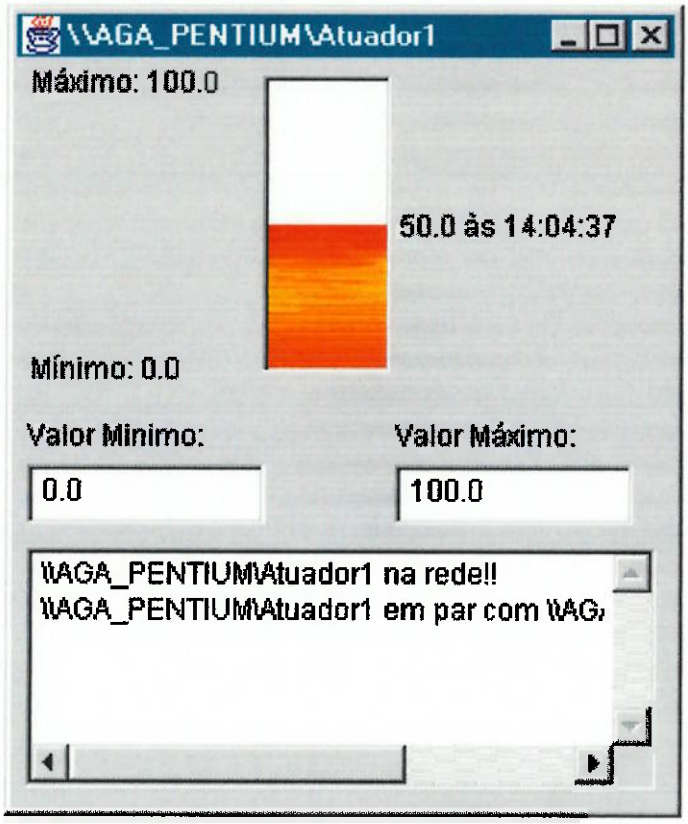

Figura 4.5.2.1.Interface gráfica do Atuador

Ao se modificar os valores do campo, quando a CLP se registrar no atuador, ela encarregará de atualizar sua construção para refletir estes valores desejados. Quando a CLP retirar o atuador de sua lista, o valores de inicialização voltarão. Assim, fica fácil identificar quando o atuador está inativo.

4.5.3. Classe Atuador

A classe Atuador implementa a interface *Inscricoes* e assim a CLP pode se inscrever remotamente. A inscrição aqui é diferente da que ocorre no sensor. Uma vez que o atuador é um elemento passivo e não irá fazer *broadcasting* de suas informações, a inscrição propriamente dita se resume apenas em informar ao componente que nele se inscreveu seus valores máximo e mínimo.

Esta classe também implementa a interface *Atualizacoes*, para poder ter seus dados atualizados remotamente pela CLP.

4.6. Package graphs

4.6.1 Visão Geral

Para a monitoração dos sensores serão utilizados componentes gráficos distintos em seu uso, e estão agrupados nesta *package*. Um deste componentes é o Trend Graph (gráfico de tendências), que plota os um número determinado de valores recebidos dos sensores analógicos em um determinado intervalo de tempo, mostrando a curva de valores que o componente assumiu durante este intervalo. O outro componente é o Bar Graph (gráfico de barras), que mostra apenas um valor instantâneo do sensor, assim como o horário de sua última atualização

Estes dois componentes foram desenvolvidos para serem usado como *Beans*, conforme descrito no item 3.1. Assim, são considerado como componentes gráficos a serem incluídos no desenvolvimento de *layouts*. Isto faz com que seja fácil o modelamento de janelas de monitoramento de acordo com o aplicação desejada. Pode-se acrescentar mais de um gráfico por janela, sendo apenas o espaço o fator delimitante. Como são componentes gráficos a serem inseridos em janelas, não podem ser inicializados isoladamente, portanto, foram feitas classes que estão na *package* Janelas, que simplesmente servem para abrigar estes Beans e poderem rodar nas janelas e interagir com o resto do sistema.

4.6.2. Package Janelas - Classes PrincipalBarGraph e PrincipalTrendGraph

Estas duas classes tem apenas o propósito de inicializar as classes JanelaBarGraph e JanelaTrendGraph, respectivamente, que irão abrigar os *Beans* BarGraph e TrendGraph. Ao se inicializar a classe, será recuperado do prompt o nome que o Bean irá ser reconhecido na rede, assegurando a individualidade de cada instância dos Beans que for aberta.

4.6.3. Package Janelas - Classes JanelaBarGraph e JanelaTrendGraph

Continuando a lógica das classes anteriores, estas duas irão abrir uma janela para poder abrigar os respectivos *Beans*, atribuir-lhes o nome recuperado do prompt e deixá-los funcionando.

4.6.4. Package BarGraphs - Classe BarGraphPanel

Uma vez abrigado pela classe JanelaBarGraph, o *Bean* BarGraph apresenta o seguinte aspecto:

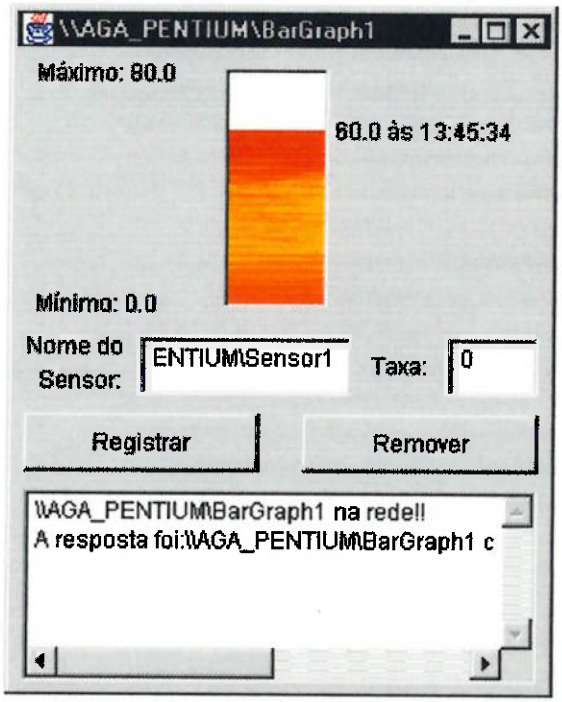

Figura 4.6.4.1. Interface gráfica do BarGraph

Os campos que possui se destinam à inscrição nos sensores, conforme descrito nas classes anteriores. No campo "Nome do Sensor" entra-se com o nome pelo qual o sensor é conhecido na rede; e no campo "Taxa", o intervalo de amostragem para a atualização do mesmo. Entrar com taxa nula indica que o BarGraph será atualizado por evento. A área de texto serve para a impressão de mensagens geradas pelo BarGraph e pelos sensores.

Ao se inscrever no sensor, o BarGraph recupera do sensor seus valores máximos e mínimos, e serão a base para as atualizações futuras. Cada atualização conta com o preenchimento do gráfico na altura correspondente, assim como escrita do valor e o horário que o sensor passou o dado para o BarGraph.

4.6.5. Package BarGraphs - Classe BarGraph

A classe BarGraph possui a lógica principal de inscrições e remoções do BarGraph nas listas dos sensores, assim como inicializa a GUI e põe o BarGraph na rede, para ser reconhecido pelo nome entrado pelo prompt que foi recuperado pela classe PrincipalBarGraph. Esta classe implementa a interface *Atualizacoes*, ou seja, possui um método chamado *Atualizar* que será chamado remotamente pelos sensores para terem seus dados sobre os mesmos atualizados. Uma vez recebidos estes dados, o gráfico BarGraphPanel correspondente será reconstruído para refletir este novo estado do sensor.

4.6.6. Package TrendGraphs - Classe TrendGraphPanel

Esta é a classe responsável pela interface gráfica do TrendGraph, e possui o seguinte aspecto:

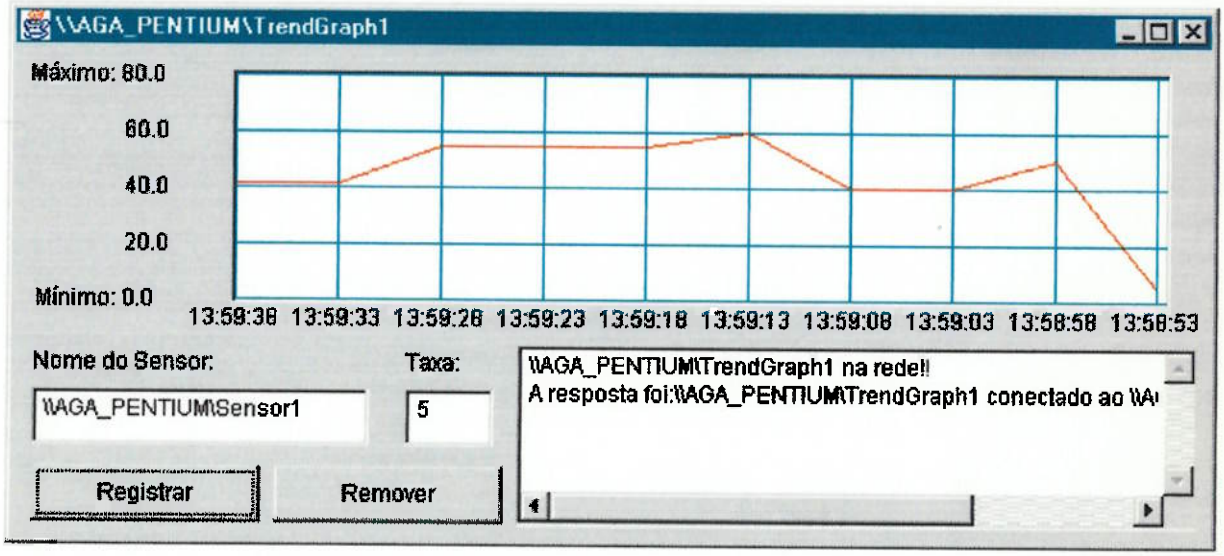

Figura 4.6.6.1. Interface gráfica do TrendGraph

Esta classe foi desenvolvida para ser um *Bean*, e possui duas características que podem ser modificadas quando em etapa de desenvolvimento da janela. A primeira é o número de colunas a serem mostradas. Dependendo da aplicação desejada, estipula-se o número de colunas que se deseja, e estas se distribuirão uniformemente ao longo da janela, sempre mantendo a largura original do gráfico. A outra característica que pode ser mudada é o número de linhas, funcionando da mesma maneira que o número de colunas, ou seja, a altura do gráfico sempre será mantida a mesma.

Os campos utilizados para inscrição e disposição de mensagens seguem o mesmo modelo que os descritos no BarGraphPanel, no item 4.6.4. As atualizações também seguem o mesmo padrão, porém aqui para cada novo ponto que se recebe, é preciso mover todos os outros pontos uma coluna para a direita, para depois poder acrescentar este novo ponto e retraçar a linha.

4.6.7. Package TrendGraphs - Classe TrendGraph

A estrutura geral do TrendGraph segue a mesma linha do BarGraph, descrita no item 4.6.5.

5. Considerações Finais

Com este projeto, espera-se estar um passo à frente no desenvolvimento de um novo ambiente para se trabalhar em SCADA. Todas as considerações feitas sobre os componentes lógicos que virão a ser substituídos por componentes físicos foram as mais genéricas e abertas possíveis, para se poder fazer uma transição segura e evitar quaisquer retrabalhos. A estrutura geral das classes e *packages* foi desenvolvida levando-se em consideração uma continuidade no desenvolvimento do projeto sem grandes dúvidas. O projeto como um todo está pronto para ser instalado em uma máquina que possui interpretador Java em poucos minutos, para se desprender mais tempo no desenvolvimento de sua estrutura do que em sua instalação.

Concluindo, o projeto possui um caráter generalista, aberto e de fácil implementação. Espera-se que com estas características seja possível ampliar sua aplicação, assim como acelerar seu desenvolvimento, para poder acompanhar as mudanças que estão ocorrendo e permanecer atual.

6. Estrutura dos Códigos

6.1 Classe Dados

```
//Classe Dados, que implementa a interface GrupoDados
//possibilitando o envio remoto de dados para inscrições

package dados;

import java.io.Serializable;

public class Dados implements Serializable {

  //Variáveis gerais da classe
  public String Acao;              //Define o se vai "registrar" ou
"remover"
  public String Componente;        //Nome do cliente
  public String TipoAtualizacao;   //Define o tipo de atualização(eventos ou
amostragem)
  public float Intervalo;          //Intervalo de atualização para
amostragem


  //Construtor principal da classe
  public Dados() {
         Acao = null;
         Componente = null;
         TipoAtualizacao = null;
         Intervalo = 0;
  }

  //Construtor com parametros já especificados
  public Dados(String a, String c, String t, float i) {
         Acao = a;
         Componente = c;
         TipoAtualizacao = t;
         Intervalo = i;
  }

}
```

6.2 Classe Ponto

```
// Classe Ponto, que agrupa valores das atualizações que os
// servidores passam para os componentes nele registrados

package dados;

import java.io.Serializable;

public class Ponto implements Serializable {

  //Variáveis gerais da classe
  public float ValorMaximo;       //Valor máximo atingido pelo componente
  public float ValorMinimo;       //Valor mínimo atingido pelo componente
  public float ValorAtual;        //Valor atual do componente
  public String Tempo;            //Horário associado ao valor passado
  public String NomeComponente;   //Nome do componente

  //Construtor principal
  public Ponto() {
         ValorMaximo = 0;
         ValorMinimo = 0;
         ValorAtual = 0;
         Tempo = "00:00:00";
         NomeComponente = null;
  }

  //Construtor para parametros já especificados
  public Ponto(float max, float min, float v, String t, String s) {
         ValorMaximo = max;
         ValorMinimo = min;
         ValorAtual = v;
         Tempo = t;
         NomeComponente = s;
  }

}
```

6.3. Classe Atualizacoes

```
//Interface Atualizacoes, que define o metodo Atualizar
//que permite que remotamente os valores dos clientes
//possam ser atualizados pelos servidores

package interfaces;

import java.rmi.Remote;
import java.rmi.RemoteException;
import dados.*;

public interface Atualizacoes extends Remote {

        Object Atualizar(Ponto p) throws RemoteException;

}
```

6.4. Classe Inscricoes

```
//Interface Inscrições, que define o método remoto a ser
//utilizado por servidores onde clientes irão se registrar
//e que recebe a classe Dados para o registro ou remoção
//e retorna a classe Ponto com os valores atuais do sensor

package interfaces;

import java.rmi.Remote;
import java.rmi.RemoteException;
import dados.*;

public interface Inscricoes extends Remote{

        Ponto AlteraInscricao(Dados d) throws RemoteException;

}
```

6.5. Classe Sensor

```
//Classe Sensor, que implementa a interface Inscricoes
//para que componentes possam se registrar remotamente

package sensores;

import java.rmi.*;
import java.rmi.server.*;
import interfaces.*;
import dados.*;

public class Sensor extends UnicastRemoteObject
                implements Inscricoes {

  //Variáveis gerais da classe
  static String nomesEventos [] = new String [11]; //Lista de clientes
atualizados por eventos
  Thread nomesTaxas []          = new Thread [11]; //Lista de clientes
atualizados por amostragem

  static String NomeSensorAtual = null; //Variável que recebe o nome do
sensor digitado no prompt
  static SensorFrame frame;             //Variável que guarda a referêcia à
GUI do sensor
  boolean packSensorFrame = false;      //Variável utilizada para validar a
GUI


  //Contrutor do Sensor, que chama a classe
  //SensorFrame, que é a GUI do Sensor
  public Sensor() throws RemoteException {

        super();  //Chama o construtor do UnicastRemoteObject

        //Bloco para validação da GUI
        frame = new SensorFrame(NomeSensorAtual);
        if (packSensorFrame)
           frame.pack();
        else
            frame.validate();
        frame.setVisible(true);

  }


  //Método que põe o Sensor na rede, para ser localizado remotamente pela
variável "nome"
  public static String ServicoSensor(String nome) {

        //Definição do Gerenciador de Segurança do RMI
        if (System.getSecurityManager() == null)
           System.setSecurityManager(new RMISecurityManager());

        //Coloca o sensor na rede como "nome"
        try {
            Inscricoes Sensor1 = new Sensor();
            Naming.rebind(nome, Sensor1);
```

```
        } catch (Exception e) {
                System.out.println("Excecao1 do " + NomeSensorAtual + "!!:
" + e.getMessage());
                e.printStackTrace();
        }

        //Retorna que o processo foi bem sucedido
        return (nome + " na rede!!");

  }


  //Método que implementa a interface Inscricoes,
  //que adiciona ou retira clientes da lista do Sensor
  public Ponto AlteraInscricao (Dados d) {

        Ponto valores = new Ponto();              //Variável que armazena
os dados do Sensor
        valores.NomeComponente = NomeSensorAtual; //Armazenamento do nome
do Sensor

        //Para Inscrição do Cliente
        if (d.Acao.compareTo("registrar")==0) {

           int i = 0;

           //Atualização orientada a evento
           if (d.TipoAtualizacao.compareTo("porevento")==0)
               while ( (i<10) & (nomesEventos[i]!= null) )
                     i++;

           //Atualização orientada a amostragem
           else  //Orientado a taxas de amostragem
                while ( (i<10) & (nomesTaxas[i]!=null) )
                      i++;

           //Tem lugar na lista para a inserção de um novo componente
           if (i<10) {

              //Inscrição do componente na lista orientada a evento
              if (d.TipoAtualizacao.compareTo("porevento")==0){
                 nomesEventos[i] = d.Componente;

              //Inscrição do componente na lista orientada a amostragem
              } else {
                 nomesTaxas[i] = new
Amostragem(d.Componente,(long)(d.Intervalo*1000));
                 nomesTaxas[i].start();
              }

              //Armazenamento dos valores do Sensor para ser passados para
o novo componente
              valores.ValorMinimo = new
Float(frame.CampoValorMinimo.getText()).floatValue();
              valores.ValorMaximo = new
Float(frame.CampoValorMaximo.getText()).floatValue();
              valores.ValorAtual  = new
Float(frame.CampoValorAtual.getText()).floatValue();
```

```
                //Impressão da resposta de sucesso de inclusão na janela do
sensor
                SensorFrame.AreaDeTexto.append(d.Componente+" registrado na
posicao "+i+" do sensor\n");

            //Não tem lugar na lista para novas inscrições
            } else {

                //Armazenamento de valores nulos para o componente pediu
o registro
                valores.ValorMinimo = 0;
                valores.ValorMaximo = 100;
                valores.ValorAtual  = 0;

                //Impressão da resposta de não inclusão na janela do
sensor
                SensorFrame.AreaDeTexto.append("Não há vagas para
"+d.Componente+".\n");
            }

            //Armazenamento do horário de atualização a ser passado para o
componente
            valores.Tempo = new
Horas().EmDigital(System.currentTimeMillis());
        }

        //Para Remoção do Cliente
        if (d.Acao.compareTo("remover")==0) {

           boolean encontrado = false;  //Variável que indica o estado de
procura do componente

           //Remoção do componente da lista orientada a evento
           if (d.TipoAtualizacao.compareTo("porevento")==0) {

              //A lista não está vazia
              if (nomesEventos[0]!= null) {

                 //Procura pelo nome na lista
                 int i = 0;
                 while ((nomesEventos[i+1]!=null) &
(d.Componente.compareTo(nomesEventos[i])!=0))
                       i++;

                 //O componente foi achado
                 if (d.Componente.compareTo(nomesEventos[i])==0) {

                    //O componentes da fila subirão de uma posição a
partir do retirado
                    int j = 0;
                    for (j=i; j<10; j++)
                        nomesEventos[j] = nomesEventos[j+1];

                    //Último lugar da lista componente fica vazio
                    nomesEventos[j] = null;

                    //Armazenamento de valores nulos para o componente que
pediu retirada
                    valores.ValorMinimo = 0;
                    valores.ValorMaximo = 100;
```

```
                    valores.ValorAtual  = 0;

                    //Impressão da resposta de sucesso de remoção na
janela do sensor
                    SensorFrame.AreaDeTexto.append(d.Componente + "
retirado da posicao "+i+" do sensor\n");

                    encontrado = true;  //Indica que o componente foi
encontrado
                }

            }

          //Remoção do componente da lista orientada a amostragem
          } else {

            //A lista não está vazia
            if (nomesTaxas[0]!= null) {

              //Procura pelo nome na lista
              int i = 0;
              while ((nomesTaxas[i+1]!= null) &
(d.Componente.compareTo(nomesTaxas[i].getName())!=0))
                  i++;

              //O componente foi achado
              if (d.Componente.compareTo(nomesTaxas[i].getName())==0) {

                //A atualização é terminada
                nomesTaxas[i].stop();
                nomesTaxas[i]= null;

                //Os componentes registrados sobem um lugar na lista
                int j = 0;
                for (j=i; j<10; j++)
                    nomesTaxas[j]= nomesTaxas[j+1];

                //Armazenamento de valores nulos para o componente que
pediu retirada
                valores.ValorMinimo = 0;
                valores.ValorMaximo = 100;
                valores.ValorAtual  = 0;

                //Impressão do resultado de sucesso na janela do
Sensor
                SensorFrame.AreaDeTexto.append(d.Componente + "
retirado da posicao "+i+" do sensor\n");

                encontrado = true; //Indica que o componente foi
encontrado
              }

            }

          }

          //O Componente não foi encontrado
          if (encontrado == false) {
```

```
                //Armazenamento de valores nulos para o componente que pediu
 retirada
                valores.ValorMinimo = 0;
                valores.ValorMaximo = 100;
                valores.ValorAtual  = 0;

                //Impressão do resultado de não encontro na janela do Sensor
                SensorFrame.AreaDeTexto.append("Não
 encontrado:"+d.Componente+".\n");
             }

             //Armazenamento de nulo para o componente que pediu a remoção
             valores.Tempo = "00:00:00";

          }

          //Retorna os valores armazenados
          return (valores);

    }


    //Método que atualiza os valores das classe registradas
    public static String Atualize(String cliente, Ponto p) {

          String resposta = null;              //Variável que armazena o
 texto de resposta
          p.NomeComponente = NomeSensorAtual; //Armazenamento do nome do
 sensor


          //Definição do Gerenciador de Segurança do RMI
          if (System.getSecurityManager() == null)
             System.setSecurityManager(new RMISecurityManager());

          //Procura pelo componente e atualiza o mesmo
          try {
              Atualizacoes objremoto = (Atualizacoes)
 Naming.lookup(cliente);
              resposta = (String) (objremoto.Atualizar(p));

          } catch (Exception e) {
                System.out.println("Excecao2 do " + NomeSensorAtual + "!!:
 " + e.getMessage());
                e.printStackTrace();
          }

          //Retorna o texto do resultado da atualização
          return("A resposta foi: " + resposta);

    }


    //Método para atualizar as classes orientadas a evento
    public static String AtualizaEvento(Ponto p){

          String resultado = null;  //Variável que armazena o texto de
 resposta

          //Armazenamento do horário de atualização
```

```
            p.Tempo = new Horas().EmDigital(System.currentTimeMillis());

            //Percorre a lista e atualiza os componentes registrados
            int i=0;
            while ( (i<10) & (nomesEventos[i]!= null) ){
                  resultado = Atualize(nomesEventos[i],p);
                  i++;
            }

            //Modifica o texto de resposta caso não haja componentes na lista
            if (resultado == null)
               resultado = "Não há componentes ligados no sensor.";

            //Retorna o texto do resultado da atualização
            return(resultado);
      }


      //Método principal do Sensor
      public static void main(String args[]) {

            NomeSensorAtual = args[0];                              //Recupera o
nome digitado no prompt
            String resposta = ServicoSensor(NomeSensorAtual);  //Liga o sensor
na rede
            SensorFrame.AreaDeTexto.append(resposta+"\n");      //Imprime
sucesso na janela do sensor

      }

}

//Thread de atualizacao das classes orientadas a taxas
class Amostragem extends Thread {

      long taxa;                  //Variável que armazena a taxa de amostragem
(em milisegundos)
      boolean Registro = true; //Variável que indica que está sendo feito o
registro na lista

      //Construtor principal da classe
      public Amostragem(String nome,long t) {

            taxa = t;         //Armazenamento da taxa de amostragem
            setName(nome);    //Dá um nome ao Thread para ser identificado

      }

      //Inicia o o processo de atualização
      public void run() {

            Ponto p = new Ponto();  //Inicializa o grupo de valores a ser
passado

            long TempoInicial = System.currentTimeMillis();  //Guarda o
tempo inicial

            //Loop geral da taxa de amostragem
            while (true) {
```

```
                    //Atualiza o valor dos clientes
                    p.ValorMinimo = new
Float(Sensor.frame.CampoValorMinimo.getText()).floatValue();
                    p.ValorMaximo = new
Float(Sensor.frame.CampoValorMaximo.getText()).floatValue();
                    p.ValorAtual  = new
Float(Sensor.frame.CampoValorAtual.getText()).floatValue();
                    p.Tempo = new Horas().EmDigital(TempoInicial);

                    //Esta iteração corresponde ao loop
                    if (Registro == true)
                       Registro = false;  //Indica que a próxima iteração
não corresponde ao registro

                    //Esta iteração não corresponde ao registro
                    else
                        Sensor.Atualize(this.getName(),p);  //Atualiza os
valores do componente

                    //Delay do restante do tempo de amostragem
                    try {
                        TempoInicial += taxa;
                        Thread.sleep(Math.max(0,TempoInicial-
System.currentTimeMillis()));
                    } catch (InterruptedException e) {
                            break;
                    }

            }

    }

}

//Classe horas, que retorna o valor de milisegundos para o formato desejado
class Horas {

      //Método que transforma o valor no formato HH:MM:SS
      public String EmDigital (long Horario) {

            String s   = ""+((int) ((Horario/1000)%60));
            String min = ""+((int) ((Horario/60000)%60));
            String h   = ""+((int) ((Horario/3600000)%24));

            //Bloco de inserção de zeros em números de 1 dígito
            if (s.length() == 1)
               s = "0" + s;
            if (min.length() == 1)
               min = "0" + min;
            if (h.length() == 1)
               h = "0" + h;

            return(h + ":" + min + ":" + s);  //Retorna o texto "HH:MM:SS"

      }

}
```

6.6. Classe SensorFrame

```
//Classe SensorFrame, que implementa a GUI do Sensor

package sensores;

import java.awt.*;
import java.awt.event.*;
import borland.jbcl.layout.*;
import interfaces.*;
import dados.*;

public class SensorFrame extends Frame implements ActionListener{

  //Variáveis gerais da classe atual
  XYLayout xYLayoutSensor     = new XYLayout();
  Button BotaoAtualizar       = new Button();
  Label TextoValorMaximo      = new Label();
  TextField CampoValorMaximo  = new TextField();
  Label TextoValorMinimo      = new Label();
  TextField CampoValorMinimo  = new TextField();
  Label TextoValorAtual       = new Label();
  TextField CampoValorAtual   = new TextField();
  static TextArea AreaDeTexto = new TextArea();

  //Construtor da classe atual
  public SensorFrame(String nome) {

    //Chama o método que define o layout da janela e inclui os componentes
acima
    try {

        MontaGUI(nome);

    } catch (Exception e) {
          System.out.println("Excecao1 do SensorFrame: " + e.getMessage());
          e.printStackTrace();
    }

  }


  //Construção da GUI
  public void MontaGUI(String NomeSensor) throws Exception{

        //Definição do layout geral da janela
        this.setLayout(xYLayoutSensor);
        this.setBackground(Color.lightGray);
        this.setSize(new Dimension(329, 205));
        this.setTitle(NomeSensor);

        //Definição das propriedades dos componentes da janela
        BotaoAtualizar.setLabel("Atualizar Valores");
        TextoValorMaximo.setText("Valor Máximo:");
        CampoValorMaximo.setText("80");
        TextoValorMinimo.setText("Valor Mínimo:");
        CampoValorMinimo.setText("0");
        TextoValorAtual.setText("Valor Atual:");
        CampoValorAtual.setText("40");
```

```
        //Inclusão dos itens na janela, com respectivas coordenadas e
tamanhos
        this.add(BotaoAtualizar,   new XYConstraints(  5, 38, 141,  24));
        this.add(TextoValorAtual,  new XYConstraints(  5,  7,  70,  25));
        this.add(CampoValorAtual,  new XYConstraints( 76,  7,  70,  24));
        this.add(AreaDeTexto,      new XYConstraints(  5, 71, 311, 100));
        this.add(TextoValorMaximo, new XYConstraints(160,  7,  75,  25));
        this.add(TextoValorMinimo, new XYConstraints(160, 38,  75,  25));
        this.add(CampoValorMaximo, new XYConstraints(245,  7,  70,  24));
        this.add(CampoValorMinimo, new XYConstraints(245, 38,  70,  24));

        //Seta que o botão do sensor e os botões da janela geram eventos
        BotaoAtualizar.addActionListener(this);
        this.addWindowListener(new WindowAdapter() {
             public void windowClosing(WindowEvent e) {
                  System.exit(0);
             }
      });

  }


  //Método para lidar com os eventos gerado pelo botão
  public void actionPerformed(ActionEvent e) {

        //Inicializacao do grupo de dados a serem passados
        Ponto p = new Ponto();

        //Envia os valores para o Sensor e imprime o resultado na janela
        if (e.getActionCommand().compareTo("Atualizar Valores")==0) {
           p.ValorMinimo = new
Float(CampoValorMinimo.getText()).floatValue();
           p.ValorMaximo = new
Float(CampoValorMaximo.getText()).floatValue();
           p.ValorAtual = new
Float(CampoValorAtual.getText()).floatValue();
           String resultado = Sensor.AtualizaEvento(p);
           AreaDeTexto.append(resultado+"\n");
        }

  }

}
```

6.7. Classe CLP

```
//Classe CLP, que implementa a interface Atualizacoes,
//para poder ter seus dados atualizados remotamente

package clps;

import java.rmi.*;
import java.rmi.server.*;
import interfaces.*;
import dados.*;

public class CLP extends UnicastRemoteObject
                 implements Atualizacoes {

  //Variáveis gerais da classe atual
  boolean packCLPFrame        = false;            //Variável para validação
da GUI
  static  CLPFrame frame      = new CLPFrame();   //Variável que
corresponde à GUI do CLP
  static  String NomeCLPAtual = null;             //Nome que o CLP vai ter

  static  String ListaAtuadores [][]    = new String [11][2]; //Lista do
par Atuador - Sensor
  static  float  ExtremosAtuadores [][] = new float [11][2];  //Lista dos
valores extremos dos atuadores


  //Contrutor do CLP, que chama a classe
  //CLPFrame, que é a GUI do CLP
  public CLP() throws RemoteException {

        super();               //Chama o construtor do UnicastRemoteObject

        //Bloco de validação da GUI
        if (packCLPFrame)
           frame.pack();
        else
            frame.validate();
        frame.setVisible(true); //Abre a janela

        //Inicialização das Listas
        for (int i=0; i<11; i++) {
            ListaAtuadores [i][0]=null;
            ListaAtuadores [i][1]=null;
        }

  }


  //Método que põe o CLP na rede, para ser localizado
  //remotamente, através da variável "nome"
  public static String ServicoCLP(String nome) {

        //Definição do Gerenciador de Segurança do RMI
        if (System.getSecurityManager() == null)
           System.setSecurityManager(new RMISecurityManager());
```

```
        //Coloca o CLP na rede como "nome"
        try {

            Atualizacoes CLP1 = new CLP();
            Naming.rebind(nome, CLP1);

        } catch (Exception e) {
                System.out.println("Excecao1 do " + nome + ": " +
e.getMessage());
                e.printStackTrace();
        }

        //Retorna texto para indicar sucesso do processo
        return (nome + " na rede!!");

  }


  //Método que chama a classe Sensor para se inscrever na sua lista
  public static String Registro(String nome, Dados d) {

        String resposta = null;          //Texto que o método retorna
        d.Componente    = NomeCLPAtual;  //Indica o nome da CLP que será
passado para o sensor

        //Definição do Gerenciador de Segurança do RMI
        if (System.getSecurityManager() == null)
           System.setSecurityManager(new RMISecurityManager());

        //Procura o sensor e efetua o registro
        try {
            Inscricoes objremoto = (Inscricoes) Naming.lookup(nome);
            Ponto ValorRecebido = objremoto.AlteraInscricao(d);
            resposta = AtualizarAtuador(ValorRecebido);

            //Definicao do texto a ser retornado
            if (d.Acao.compareTo("registrar")==0) {
               resposta = NomeCLPAtual + " conectado ao " + nome + ".";
            } else {
                  resposta = NomeCLPAtual + " retirado do " + nome + ".";
            }
        } catch (Exception e) {
                System.out.println("Excecao2 do "+NomeCLPAtual+": " +
e.getMessage());
                e.printStackTrace();
        }

        //O registro foi efetuado
        if (resposta != null)
           return("A resposta foi:" + resposta);

        //O registro não foi efetuado
        else
            return(nome + " não encontrado!");

  }


  //Método que chama a classe Atuador para pegar seus dados
```

```
  public static String RegistroAtuador(String atuador, String sensor,
String acao) {

          String resposta = null;                          //Texto que o método
retorna

          //Definição do Gerenciador de Segurança do RMI
          if (System.getSecurityManager() == null)
             System.setSecurityManager(new RMISecurityManager());

          //Registro do atuador na lista
          if (acao.compareTo("registrar")==0){
             try {
                 Inscricoes objremoto = (Inscricoes) Naming.lookup(atuador);
//Procura o atuador

                 Dados d = new Dados();  //Inicializa os dados a serem
passados
                 d.Componente = sensor;  //Envia o nome do sensor que faz
par com o atuador
                 d.Acao = "registrar";   //Indica que está se efetuando um
registro

                 Ponto AtuadorRecebido = objremoto.AlteraInscricao(d);
//Recebe os dados do atuador

                 //Procura um lugar na lista para incluir o atuador
                 int i=0;
                 while ( (i<10) & (ListaAtuadores[i][0]!= null) )
                       i++;

                 //Achou um lugar
                 if (i<10) {

                    //Insere os dados do atuador nas listas
                    ListaAtuadores[i][0] = atuador;
                    ListaAtuadores[i][1] = sensor;
                    ExtremosAtuadores[i][0] = AtuadorRecebido.ValorMinimo;
                    ExtremosAtuadores[i][1] = AtuadorRecebido.ValorMaximo;
                    resposta = NomeCLPAtual + " conectado ao " + atuador +
".";

                 //Não há vagas
                 } else {

                    resposta = "Não há vagas para " + atuador + ".";

                 }

             } catch (Exception e) {
                    System.out.println("Excecao3 do "+NomeCLPAtual+": " +
e.getMessage());
                    e.printStackTrace();
             }

          //Remoção do atuador na lista
          } else {

                //Lista não está vazia
                if (ListaAtuadores[0][0]!= null) {
```

```
                    //Procura o par Atuador - Sensor correspondente
                    int i=0;
                    while ( (i<10) &
((ListaAtuadores[i][0].compareTo(atuador)!=0) ||
(ListaAtuadores[i][1].compareTo(sensor)!=0))){
                          i++;
                    }

                    //Achou o atuador
                    if (i<10) {

                              try {

                                   //Os componentes da lista sobem de uma
posição

                                   int j = 0;
                                   for (j=i; j<10; j++) {
                                       ListaAtuadores[j][0] =
ListaAtuadores[j+1][0];

                                       ListaAtuadores[j][1] =
ListaAtuadores[j+1][1];

                                       ExtremosAtuadores[j][0] =
ExtremosAtuadores[j+1][0];

                                       ExtremosAtuadores[j][1] =
ExtremosAtuadores[j+1][0];
                                   }

                                   //O último valor da lista fica vazio
                                   ListaAtuadores[j][0] = null;
                                   ListaAtuadores[j][1] = null;
                                   ExtremosAtuadores[j][0] = 0;
                                   ExtremosAtuadores[j][1] = 0;

                                   //Imprime o resultado na janela da CLP
                                   resposta = atuador + " retirado do " +
NomeCLPAtual + ".";


                                   //Acessa o atuador para indicar a remoção
                                   Dados d = new Dados();
                                   d.Componente = sensor;
                                   d.Acao = "remover";
                                   Inscricoes objremoto = (Inscricoes)
Naming.lookup(atuador);
                                   Ponto AtuadorRecebido =
objremoto.AlteraInscricao(d);

                              } catch (Exception e) {
                                   System.out.println("Excecao4 do
"+NomeCLPAtual+": " + e.getMessage());
                                   e.printStackTrace();
                              }

                    //Não achou o atuador
                    } else {
                          resposta = "Não encontrado: " + atuador + ".";
                    }

               //A lista está vazia
               } else {
```

```
                        resposta = "A lista está vazia.";
                }

        }


        //Remoção efetuada com sucesso
        if (resposta != null)
           return("A resposta foi:" + resposta);

        //Remoção não foi efetuada
        else
             return(atuador + " não encontrado!");
  }


  //Método que implementa a interface Atualizacoes,
  //que permite que o sensor atualize remotamente o atuador
  public Object Atualizar(Ponto p) {

        String resposta = AtualizarAtuador(p);
        return(resposta);

  }


  //Método que atualiza os atuadores
  public static String AtualizarAtuador(Ponto p) {

        String resposta = null;  //Variável do texto a ser retornado

        //A lista não está vazia
        if (ListaAtuadores[0][0]!= null) {

           //Procura os atuadores influenciados pelo sensor em questão
           int i=0;
           while ((i<10) &
(ListaAtuadores[i][1].compareTo(p.NomeComponente)!=0))
                i++;

           //Achou o atuador
           if (i<10) {

              //Seta os valores a serem passados para o atuador
              String cliente = ListaAtuadores[i][0];
              p.ValorAtual = Controlador(p,i);
              p.ValorMinimo = ExtremosAtuadores[i][0];
              p.ValorMaximo = ExtremosAtuadores[i][1];


              //Definição do Gerenciador de Segurança do RMI
              if (System.getSecurityManager() == null)
                 System.setSecurityManager(new RMISecurityManager());

              //Efetua a atualização no atuador
              try {
                  Atualizacoes objremoto = (Atualizacoes)
Naming.lookup(cliente);
                  resposta = (String) (objremoto.Atualizar(p));
              } catch (Exception e) {
```

```
                        System.out.println("Excecao5 do " + NomeCLPAtual +
"!!: " + e.getMessage());
                        e.printStackTrace();
                }

            //Não há nenhum atuador que vai ser modificado devido ao sensor
            } else {
                    resposta = "Não há atuadores em par com
"+p.NomeComponente;
            }

        //A lista está vazia
        } else {
                resposta = "Não há atuadores em par com "+p.NomeComponente;
        }

        return("A resposta foi: " + resposta);

  }

  //Lógica do controlador -> recebe o valor do sensor em relação ao máximos
e mínimos
  //e calcula o valor correspondente para o atuador
  public static float Controlador(Ponto p, int i) {

        float porcentagem = (p.ValorAtual-p.ValorMinimo)/(p.ValorMaximo-
p.ValorMinimo);
        float ValorAtuador = (porcentagem*(ExtremosAtuadores[i][1]-
ExtremosAtuadores[i][0]))+ExtremosAtuadores[i][0];
        return (ValorAtuador);

  }

  //Método principal do CLP
  public static void main (String args[]) {

        NomeCLPAtual = args[0];                                    //Recebe o
nome do CLP do prompt
        String resultado = ServicoCLP(NomeCLPAtual);               //Coloca o
CLP na rede
        frame.setTitle(NomeCLPAtual);                              //Define o
título da janela
        frame.AreaDeTexto.append(NomeCLPAtual+" na rede!!\n"); //Imprime o
resultado

  }

}
```

6.8. Classe CLPFrame

```
//Classe que implementa a GUI do CLP

package clps;

import java.awt.*;
import java.awt.event.*;
import borland.jbcl.layout.*;
import interfaces.*;
import dados.*;

public class CLPFrame extends Frame implements ActionListener{

  //Variáveis gerais da classe
  XYLayout  xYLayoutCLP             = new XYLayout();
  Label     TextoComponenteAtuador  = new Label();
  Label     TextoSensor             = new Label();
  TextField CampoComponenteAtuador  = new TextField();
  TextField CampoSensor             = new TextField();
  Button    BotaoRegistrarAtuador   = new Button();
  Button    BotaoRemoverAtuador     = new Button();
  Label     TextoComponenteSensor   = new Label();
  Label     TextoTaxaSensor         = new Label();
  TextField CampoComponenteSensor   = new TextField();
  TextField CampoTaxaSensor         = new TextField();
  Button    BotaoRegistrarSensor    = new Button();
  Button    BotaoRemoverSensor      = new Button();
  static    TextArea AreaDeTexto    = new TextArea();

  //Construtor da classe atual
  public CLPFrame() {

    //Chama o método que define o layout da janela e inclui os componentes
acima
    try {

        MontaGUI();

    } catch (Exception e) {
          System.out.println("Excecao1 do CLPFrame: " + e.getMessage());
          e.printStackTrace();
    }

  }


  //Construção da GUI
  public void MontaGUI() throws Exception{

        //Definição do layout geral da janela
        this.setLayout(xYLayoutCLP);
        this.setBackground(Color.lightGray);
        this.setSize(new Dimension(400, 275));

        //Definição das propriedades dos componentes da janela
        TextoComponenteAtuador.setText("Nome do Atuador:");
        CampoComponenteAtuador.setBackground(Color.white);
```

```
        TextoSensor            .setText("Sensor Reponsável:");
        CampoSensor            .setBackground(Color.white);
        BotaoRegistrarAtuador .setLabel("Registrar Atuador");
        BotaoRemoverAtuador    .setLabel("Remover Atuador");
        TextoComponenteSensor .setText("Nome do Sensor:");
        CampoComponenteSensor .setBackground(Color.white);
        TextoTaxaSensor        .setText("Taxa:");
        CampoTaxaSensor        .setText("0");
        BotaoRegistrarSensor   .setLabel("Registrar CLP no Sensor");
        BotaoRemoverSensor     .setLabel("Remover CLP do Sensor");
        AreaDeTexto            .setBackground(Color.white);

        //Inclusão dos itens na janela, com respectivas coordenadas e
tamanhos
        this.add(TextoComponenteAtuador, new XYConstraints(  5,   7, 115,
25));
        this.add(CampoComponenteAtuador, new XYConstraints(125,   7, 125,
25));
        this.add(TextoSensor,            new XYConstraints(  5,  35, 115,
25));
        this.add(CampoSensor,            new XYConstraints(125,  35, 125,
25));
        this.add(BotaoRegistrarAtuador,  new XYConstraints(260,   7, 122,
25));
        this.add(BotaoRemoverAtuador,    new XYConstraints(260,  35, 122,
25));
        this.add(TextoComponenteSensor,  new XYConstraints(  5, 180, 100,
25));
        this.add(CampoComponenteSensor,  new XYConstraints(110, 180, 145,
25));
        this.add(TextoTaxaSensor,        new XYConstraints(285, 180,  40,
25));
        this.add(CampoTaxaSensor,        new XYConstraints(325, 180,  55,
25));
        this.add(BotaoRegistrarSensor,   new XYConstraints( 15, 215, 165,
25));
        this.add(BotaoRemoverSensor,     new XYConstraints(195, 215, 165,
25));
        this.add(AreaDeTexto,            new XYConstraints(  5,  70, 380,
100));

        //Seta que os botões da CLP e os botões da janela geram eventos
        BotaoRemoverSensor  .addActionListener(this);
        BotaoRegistrarSensor.addActionListener(this);
        BotaoRemoverAtuador  .addActionListener(this);
        BotaoRegistrarAtuador.addActionListener(this);
        this.addWindowListener(new WindowAdapter() {
             public void windowClosing(WindowEvent e) {
                  System.exit(0); } });

  }


  //Método para lidar com os eventos gerados pelos botões
  public void actionPerformed(ActionEvent e) {

        Dados d = new Dados();  //Inicializacao do grupo de dados a serem
passados

        //Ação do botão RegistrarSensor
```

```
            if (e.getActionCommand().compareTo("Registrar CLP no Sensor")==0)
 {

                //Envia os valores para a CLP e imprime o resultado na janela
                d.Acao = "registrar";
                d.Intervalo = new
 Float(CampoTaxaSensor.getText()).floatValue();

                if (d.Intervalo > 0)
                   d.TipoAtualizacao = "poramostragem";
                else
                   d.TipoAtualizacao = "porevento";

                String resposta =
 CLP.Registro(CampoComponenteSensor.getText(),d);
                AreaDeTexto.append(resposta+"\n");

            }

            //Ação do botão RemoverSensor
            if (e.getActionCommand().compareTo("Remover CLP do Sensor")==0) {

                //Envia os valores para a CLP e imprime o resultado na janela
                d.Acao = "remover";
                d.Intervalo = new
 Float(CampoTaxaSensor.getText()).floatValue();

                if (d.Intervalo > 0)
                   d.TipoAtualizacao = "poramostragem";
                else
                   d.TipoAtualizacao = "porevento";

                String resposta =
 CLP.Registro(CampoComponenteSensor.getText(),d);
                AreaDeTexto.append(resposta+"\n");
            }

            //Ação do botão RegistrarAtuador
            if (e.getActionCommand().compareTo("Registrar Atuador")==0) {

                //Envia os valores para a CLP e imprime o resultado na janela
                String acao = "registrar";
                String sensor = CampoSensor.getText();
                String atuador = CampoComponenteAtuador.getText();
                String resposta = CLP.RegistroAtuador(atuador,sensor,acao);
                AreaDeTexto.append(resposta+"\n");
            }

            //Ação do botão RemoverAtuador
            if (e.getActionCommand().compareTo("Remover Atuador")==0) {

                //Envia os valores para a CLP e imprime o resultado na janela
                String acao = "remover";
                String sensor = CampoSensor.getText();
                String atuador = CampoComponenteAtuador.getText();
                String resposta = CLP.RegistroAtuador(atuador,sensor,acao);
                AreaDeTexto.append(resposta+"\n");
            }
    }
}
```

6.9. Classe Atuador

```
//Classe Atuador, que implementa a interface Atualizacoes, para poder ter
//seus dados atualizados remotamente; e a interface Inscricoes, para que
//componentes possam se registrar remotamente

package atuadores;

import java.rmi.*;
import java.rmi.server.*;
import interfaces.*;
import dados.*;

public class Atuador extends UnicastRemoteObject
                  implements Atualizacoes, Inscricoes {

  //Variáveis gerais da classe atual
  boolean packAtuadorFrame        = false;              //Variável para
validação da GUI
  static  AtuadorFrame frame      = new AtuadorFrame(); //Variável que
corresponde à GUI do Atuador
  static  String NomeAtuadorAtual = null;               //Nome que o
Atuador vai ter


  //Construtor do Atuador, que chama a classe
  //AtuadorFrame, que é a GUI do Atuador
  public Atuador() throws RemoteException {

        super();              //Chama o construtor do UnicastRemoteObject

        //Bloco de validação da GUI
        if (packAtuadorFrame)
           frame.pack();
        else
            frame.validate();
        frame.setVisible(true); //Abre a janela

  }


  //Método que põe o Atuador na rede, para ser localizado
  //remotamente, através da variável "nome"
  public static String ServicoAtuador(String nome) {

        //Definição do Gerenciador de Segurança do RMI
        if (System.getSecurityManager() == null)
           System.setSecurityManager(new RMISecurityManager());

        //Coloca o Atuador na rede como "nome"
        try {

            Atualizacoes Atuador1 = new Atuador();
            Naming.rebind(nome, Atuador1);

        } catch (Exception e) {
                System.out.println("Excecao1 do " + nome + ": " +
e.getMessage());
                e.printStackTrace();
```

```
            }

            //Retorna texto para indicar sucesso do processo
            return (nome + " na rede!!");

    }


    //Método que retorna os valores do atuador
    public Ponto AlteraInscricao (Dados d) {

            Ponto p = new Ponto();  //Inicializa os dados a serem passados

            //Efetua o registro
            if (d.Acao.compareTo("registrar")==0){

                //Armazena os dados referentes ao atuador e imprime o resultado
na janela
                p.NomeComponente = (NomeAtuadorAtual);
                p.ValorMinimo = new
Float(frame.CampoValorMinimo.getText()).floatValue();
                p.ValorMaximo = new
Float(frame.CampoValorMaximo.getText()).floatValue();
                frame.AreaDeTexto.append(NomeAtuadorAtual+" em par com
"+d.Componente+"!\n"); //Imprime o resultado

            //Efetua a remoção
            } else {

                    //Imprime o resultado na janela
                    frame.AreaDeTexto.append(NomeAtuadorAtual+" não está em par
com "+d.Componente+"!\n"); //Imprime o resultado
            }

            //Envia os dados para a CLP
            return(p);

    }


    //Método que implementa a interface Atualizacoes,
    //que permite que o sensor atualize remotamente o gráfico
    public Object Atualizar(Ponto p) {

            //Envia os dados para a GUI e imprime o resultado na janela
            frame.Construir(p);
            return("Atualização feita.");

    }


    //Método principal do Atuador
    public static void main (String args[]) {

            NomeAtuadorAtual = args[0];
//Recebe o nome do Atuador do prompt
            String resultado = ServicoAtuador(NomeAtuadorAtual);
//Coloca o Atuador na rede
            frame.setTitle(NomeAtuadorAtual);
//Define o título da janela
```

```
            frame.AreaDeTexto.append(NomeAtuadorAtual+" na rede!!\n");
    //Imprime o resultado

      }

    }
```

6.10. Classe AtuadorFrame

```
    //Classe que implementa a GUI do Atuador

    package atuadores;

    import java.awt.*;
    import java.awt.event.*;
    import borland.jbcl.layout.*;
    import interfaces.*;
    import dados.*;

    public class AtuadorFrame extends Frame {

      //Variáveis gerais da classe
      XYLayout  xYLayoutAtuador      = new XYLayout();
      Label     TextoValorMaximo     = new Label();
      Label     TextoValorMinimo     = new Label();
      TextField CampoValorMinimo     = new TextField();
      TextField CampoValorMaximo     = new TextField();
      static    TextArea AreaDeTexto = new TextArea();

      //Variáveis gerais para o gráfico impresso
      static float MaxValor = 100;          //Valor máximo do atuador na unidade
    coerente
      static float MinValor = 0;            //Valor máximo do atuador na unidade
    coerente
      float ValorReal       = 0;            //Valor atual do atuador na unidade
    coerente
      int   ValorPixel      = 0;            //Valor atual do atuador em pixel
      int   altura          = 120;          //Altura em pixel do gráfico
      int   largura         = 50;           //Largura em pixel do gráfico
      String Tempo          = "00:00:00";   //Horário inicial a ser impresso


      //Construtor da classe atual
      public AtuadorFrame() {

        //Chama o método que define o layout da janela e inclui os componentes
    acima
        try {

            MontaGUI();

        } catch (Exception e) {
                System.out.println("Excecao1 do AtuadorFrame: " +
    e.getMessage());
                e.printStackTrace();
        }

      }
```

```
  //Construção da GUI
  public void MontaGUI() throws Exception{

        //Definição do layout geral da janela
        this.setLayout(xYLayoutAtuador);
        this.setBackground(Color.lightGray);
        this.setSize(new Dimension(285, 340));

        //Definição das propriedades dos componentes da janela
        CampoValorMaximo .setBackground(Color.white);
        TextoValorMaximo .setText("Valor Máximo:");
        CampoValorMaximo .setText(""+MaxValor);
        AreaDeTexto      .setBackground(Color.white);
        TextoValorMinimo .setText("Valor Minimo:");
        CampoValorMinimo .setText(""+MinValor);

        //Inclusão dos itens na janela, com respectivas coordenadas e
tamanhos
        this.add(TextoValorMinimo, new XYConstraints(  5, 145, 100,  25));
        this.add(CampoValorMinimo, new XYConstraints(  5, 170, 100,  25));
        this.add(TextoValorMaximo, new XYConstraints(160, 145, 100,  25));
        this.add(CampoValorMaximo, new XYConstraints(160, 170, 100,  25));
        this.add(AreaDeTexto,      new XYConstraints(  5, 205, 265, 100));

        //Seta que os botões da janela geram eventos
        this.addWindowListener(new WindowAdapter() {
             public void windowClosing(WindowEvent e) {
                  System.exit(0); } });

  }


  //Método que calcula o valor em pixel da altura do gráfico e associa ao
horário de atualização
  public void Construir(Ponto p) {

        MinValor   = p.ValorMinimo;
        MaxValor   = p.ValorMaximo;
        ValorReal  = p.ValorAtual;
        Tempo      = p.Tempo;
        ValorPixel = (int) (((ValorReal-MinValor)/(MaxValor-
MinValor))*altura);
        repaint();

     }


  //Método que constrói o gráfico
  public void paint(Graphics g) {

        int x = 110;  //Localização em x do gráfico a ser impresso
        int y = 32;   //Localização em y do gráfico a ser impresso

        //Construção do efeito 3D do gráfico
        MontaMoldura(x,y,g);

        //Preenchimento da altura do gráfico
        g.setColor(Color.red);
```

```
            g.fillRect(x, y+(altura-ValorPixel), largura, ValorPixel);

            //Textos de máximo, mínimo e valor atual
          g.setColor(Color.black);
            g.drawString("Máximo: " + CortaCasas(MaxValor), x-100, y+5);
            g.drawString("Mínimo: " + CortaCasas(MinValor), x-100,
y+altura+5);
          g.drawString(CortaCasas(ValorReal) + " às " + Tempo, x+55,
y+(altura-ValorPixel)+5);

    }


    //Método que monta o efeito 3D do gráfico
    public void MontaMoldura (int x, int y, Graphics g){

            //Linha cinza
            g.setColor(Color.gray);
            g.drawLine(x-2, y+altura, x-2        , y-2);
            g.drawLine(x-2, y-2     , x+largura+1, y-2);

            //Linha branca
            g.setColor(Color.white);
            g.drawLine(x-2        , y+altura+1, x+largura+1, y+altura+1);
            g.drawLine(x+largura+1, y+altura+1, x+largura+1, y-2);

            //Linha preta
            g.setColor(Color.black);
            g.drawLine(x-1, y+altura, x-1        , y-1);
            g.drawLine(x-1, y-1     , x+largura-1, y-1);

            //Fundo branco
            g.setColor(Color.white);
            g.fillRect(x, y, largura, altura);
    }


    //Método que corta as casas do valor impresso na tela
    public static String CortaCasas(float Valor) {

            int Inteiro = (int) (Valor);
            int Casas = (int) ( ( Valor - ((float) (Inteiro)) ) * 100 );
            float Resultado = ((float) (Inteiro)) + (((float) (Casas))/100);
            return ("" + Resultado);

    }

}
```

6.11 Classe PrincipalBarGraph

```
//Classe PrincipalBarGraph, que recebe o nome do prompt, e
//inicializa a janela que hospedará o BarGraph

package graphs.Janelas;

public class PrincipalBarGraph {

  boolean packFrame = false;  //Variável de validação da GUI


  //Construtor da classe atual
  public PrincipalBarGraph(String nome) {

        //Inicializa a janela que contém o BarGraph
        JanelaBarGraph frame = new JanelaBarGraph(nome);

        //Valida a janela
        if (packFrame)
           frame.pack();
        else
            frame.validate();
        frame.setVisible(true);

  }

  //Método principal da classe
  static public void main(String[] args) {

    //Inicializa a classe com o nome recebido pelo prompt
    new PrincipalBarGraph(args[0]);

  }

}
```

6.12 Classe JanelaBarGraph

```
//Classe JanelaBarGraph, que chama serve como hospedeira para a classe
//BarGraph, que está sendo incluída como um Bean

package graphs.Janelas;

import java.awt.*;
import java.awt.event.*;
import borland.jbcl.layout.*;
import graphs.BarGraphs.*;

public class JanelaBarGraph extends Frame {

  //Variáveis gerais da classe
  XYLayout xYLayout1     = new XYLayout();
  String titulo          = null;
```

```
  BarGraphPanel janela;  //Variável que indica o BarGraph que vai ser
adicionado à janela

  //Construtor da classe atual
  public JanelaBarGraph(String nome) {

    titulo = nome;                  //Define o título da janela (o nome do
BarGraph)
    janela = new BarGraphPanel();  //Inicializa o novo BarGraph
    janela.Conecta(titulo);        //Faz o BarGraph se conectar na rede

    //Chama o método que define o layout da janela
    try {

         MontaGUI();

    } catch (Exception e) {
            System.out.println("Excecao1 do JanelaBarGraph: " +
e.getMessage());
            e.printStackTrace();
    }

  }


  //Construção da GUI
  public void MontaGUI() throws Exception{

        //Definição do layout geral da janela
        this.setLayout(xYLayout1);
        this.setBackground(Color.lightGray);
        this.setSize(new Dimension(285, 360));
        this.setTitle(titulo);

        //Seta que os botões da janela geram eventos
        this.addWindowListener(new WindowAdapter() {
             public void windowClosing(WindowEvent e) {
                  System.exit(0);
             }
        });

        //Adiciona o BarGraph na janela
        this.add(janela);

  }

}
```

6.13. Classe BarGraphPanel

```
//Classe que implementa a GUI do BarGraph, que está acertada para ser usada
como Bean

package graphs.BarGraphs;

import java.awt.*;
import java.awt.event.*;
import borland.jbcl.layout.*;
import interfaces.*;
import dados.*;
import java.io.Serializable;

public class BarGraphPanel extends    Panel
                           implements ActionListener,
                                      Serializable {

  //Variáveis gerais da classe
  XYLayout  xYLayoutBarGraph        = new XYLayout();
  Button    BotaoRegistrar          = new Button();
  Button    BotaoRemover            = new Button();
  Label     TextoComponente         = new Label();
  Label     TextoComponente2        = new Label();
  Label     TextoTaxa               = new Label();
  TextField CampoTaxa               = new TextField();
  TextField CampoComponenteServidor = new TextField();
  static    TextArea AreaDeTexto    = new TextArea();

  //Variáveis gerais para o gráfico impresso
  float MaxValor   = 100;         //Valor máximo do atuador na unidade
coerente
  float MinValor   = 0;           //Valor máximo do atuador na unidade
coerente
  float ValorReal  = 0;           //Valor atual do atuador na unidade
coerente
  int   ValorPixel = 0;           //Valor atual do atuador em pixel
  int   altura     = 120;         //Altura em pixel do gráfico
  int   largura    = 50;          //Largura em pixel do gráfico
  String Tempo     = "00:00:00"; //Horário inicial a ser impresso

  BarGraph BarGraphMaster;  //Variável que seta o BarGraph que vai
controlar este gráfico

  //Construtor da classe atual
  public BarGraphPanel() {

    //Chama o método que define o layout da janela e inclui os componentes
acima
    try {

        MontaGUI();

    } catch (Exception e) {
          System.out.println("Excecao1 do BarGraphPanel: " +
e.getMessage());
          e.printStackTrace();
    }
```

```
    }

    //Método que chama o BarGraphMaster e conecta na rede como "nome"
    public void Conecta(String nome) {

      try {

            BarGraphMaster = new BarGraph(this, nome);

      } catch (Exception e) {
            System.out.println("Excecao1 do BarGraphPanel: " +
e.getMessage());
            e.printStackTrace();
      }

    }

    //Construção da GUI
    public void MontaGUI() throws Exception{

            //Definição do layout geral da janela
            this.setLayout(xYLayoutBarGraph);
            this.setBackground(Color.lightGray);
            this.setSize(new Dimension(285, 360));

            //Definição do layout geral da janela
            CampoComponenteServidor.setBackground(Color.white);
            BotaoRegistrar          .setLabel("Registrar");
            TextoComponente         .setText("Nome do");
            TextoComponente2        .setText("Sensor:");
            BotaoRemover            .setLabel("Remover");
            AreaDeTexto             .setBackground(Color.white);
            TextoTaxa               .setText("Taxa:");
            CampoTaxa               .setText("0");

            //Inclusão dos itens na janela, com respectivas coordenadas e
tamanhos
            this.add(TextoComponente,          new XYConstraints(  5, 147,  60,
15));
            this.add(TextoComponente2,         new XYConstraints( 12, 165,  50,
15));
            this.add(CampoComponenteServidor, new XYConstraints( 65, 150, 110,
30));
            this.add(TextoTaxa,                new XYConstraints(185, 150,  35,
30));
            this.add(CampoTaxa,                new XYConstraints(225, 150,  47,
30));
            this.add(BotaoRegistrar,           new XYConstraints(  5, 190, 122,
30));
            this.add(BotaoRemover,             new XYConstraints(150, 190, 122,
30));
            this.add(AreaDeTexto,              new XYConstraints(  5, 230, 267,
100));

            //Seta que os botões do gráfico geram eventos
            BotaoRemover  .addActionListener(this);
            BotaoRegistrar.addActionListener(this);
```

```
  }

  //Método que calcula o valor em pixel da altura do gráfico e associa ao
horário de atualização
  public void Construir(Ponto p) {

        ValorReal  = p.ValorAtual;
        MinValor   = p.ValorMinimo;
        MaxValor   = p.ValorMaximo;
        Tempo      = p.Tempo;
        ValorPixel = (int) (((ValorReal-MinValor)/(MaxValor-
MinValor))*altura);
        repaint();

     }

  //Método que constrói o gráfico
  public void paint(Graphics g) {

        int x = 110;  //Localização em x do gráfico a ser impresso
        int y = 12;   //Localização em y do gráfico a ser impresso

        //Construção do efeito 3D do gráfico
        MontaMoldura(x,y,g);

        //Preenchimento da altura
        g.setColor(Color.red);
        g.fillRect(x, y+(altura-ValorPixel), largura, ValorPixel);

        //Textos de máximo, mínimo e valor atual
      g.setColor(Color.black);
        g.drawString("Máximo: " + CortaCasas(MaxValor), x-100, y+5);
        g.drawString("Mínimo: " + CortaCasas(MinValor), x-100,
y+altura+5);
      g.drawString(CortaCasas(ValorReal) + " às " + Tempo, x+55,
y+(altura-ValorPixel)+5);

  }

  //Método que monta o efeito 3D do gráfico
  public void MontaMoldura (int x, int y, Graphics g){

        //Linha cinza
        g.setColor(Color.gray);
        g.drawLine(x-2, y+altura, x-2        , y-2);
        g.drawLine(x-2, y-2     , x+largura+1, y-2);

        //Linha branca
        g.setColor(Color.white);
        g.drawLine(x-2        , y+altura+1, x+largura+1, y+altura+1);
        g.drawLine(x+largura+1, y+altura+1, x+largura+1, y-2);

        //Linha preta
        g.setColor(Color.black);
        g.drawLine(x-1, y+altura, x-1        , y-1);
        g.drawLine(x-1, y-1     , x+largura-1, y-1);
```

```
        //Fundo branco
        g.setColor(Color.white);
        g.fillRect(x, y, largura, altura);
    }


    //Método para lidar com os eventos gerados pelos botões
    public void actionPerformed(ActionEvent e) {

        Dados d = new Dados();  //Inicializacao do grupo de dados a serem
passados

        //Ação do botão 'Registrar'
```

SCADA em Java - Pág. 61

```
        //Fundo branco
        g.setColor(Color.white);
        g.fillRect(x, y, largura, altura);
    }


    //Método para lidar com os eventos gerados pelos botões
    public void actionPerformed(ActionEvent e) {

        Dados d = new Dados();  //Inicializacao do grupo de dados a serem
passados

        //Ação do botão 'Registrar'
        if (e.getActionCommand().compareTo("Registrar")==0) {

            //Envia os valores para o BarGarphMaster e imprime o resultado
na janela
            d.Acao = "registrar";
            d.Intervalo = new Float(CampoTaxa.getText()).floatValue();

            if (d.Intervalo > 0)
               d.TipoAtualizacao = "poramostragem";
            else
               d.TipoAtualizacao = "porevento";

            String resposta =
BarGraphMaster.Registro(CampoComponenteServidor.getText(),d);
            AreaDeTexto.append(resposta+"\n");

        }

        //Ação do botão 'Remover'
        if (e.getActionCommand().compareTo("Remover")==0) {

            //Envia os valores para o BarGarphMaster e imprime o resultado
na janela
            d.Acao = "remover";
            d.Intervalo = new Float(CampoTaxa.getText()).floatValue();

            if (d.Intervalo > 0)
               d.TipoAtualizacao = "poramostragem";
            else
               d.TipoAtualizacao = "porevento";

            String resposta =
BarGraphMaster.Registro(CampoComponenteServidor.getText(),d);
            AreaDeTexto.append(resposta+"\n");
        }
```

6.14. Classe BarGraph

```
//Classe BarGraph, que implementa a interface Atualizacoes,
//para poder ter seus dados atualizados remotamente

package graphs.BarGraphs;

import java.rmi.*;
import java.rmi.server.*;
import interfaces.*;
import dados.*;

public class BarGraph extends UnicastRemoteObject
                  implements Atualizacoes {

  //Variáveis gerais da classe atual
  static  BarGraphPanel frame      ;        //Variável que corresponde à
GUI do BarGraph
  static  String NomeBarGraphAtual = null;  //Nome que o BarGraph vai ter
  static  int numeroDeRegistros    = 0;     //Variável que armazena o
número de registros feitos


  //Contrutor do BarGraph, que chama a classe
  //BarGraphPanel, que é a GUI do BarGraph
  public BarGraph(BarGraphPanel frame1, String nome) throws RemoteException
{

        super();                        //Chama o construtor do
UnicastRemoteObject

        frame = frame1;                 //Recebe o BarGraphPanel enviado
        NomeBarGraphAtual = nome;       //Armazena o nome do BarGraphPanel
enviado

        //Bloco que põe o BarGraph na rede, para ser localizado
remotamente, através do "nome"

        //Definição do Gerenciador de Segurança do RMI
        if (System.getSecurityManager() == null)
           System.setSecurityManager(new RMISecurityManager());

        //Coloca o BarGraph na rede como "nome"
        try {

            Naming.rebind(NomeBarGraphAtual, this);

        } catch (Exception e) {
                System.out.println("Excecao1 do " + NomeBarGraphAtual + ":
" + e.getMessage());
                e.printStackTrace();
        }

        //Imprime o resultado na janela do BarGraphPanel
        frame.AreaDeTexto.append(NomeBarGraphAtual+" na rede!!\n");

}
```

```
    //Método que chama a classe Sensor para se inscrever na sua lista
    public static String Registro(String nome, Dados d) {

            String resposta = null;                 //Texto que o método retorna
            Ponto p         = new Ponto();          //Valores que irá receber
 para atualização
            d.Componente    = NomeBarGraphAtual; //Indica o nome do BarGraph
 que será passado para o sensor

            //Definição do Gerenciador de Segurança do RMI
            if (System.getSecurityManager() == null)
               System.setSecurityManager(new RMISecurityManager());

            //Procura o sensor e efetua o registro
            try {
                Inscricoes objremoto = (Inscricoes) Naming.lookup(nome);
                p = objremoto.AlteraInscricao(d);

                //Definicao do texto a ser retornado
                if (d.Acao.compareTo("registrar")==0) {
                   resposta = NomeBarGraphAtual+" conectado ao " + nome + ".";
                   numeroDeRegistros = numeroDeRegistros + 1;

                   frame.Construir(p);  //Atualiza o aspecto do gráfico

                } else {
                      resposta = NomeBarGraphAtual+" retirado do "+nome+".";
                      numeroDeRegistros = numeroDeRegistros - 1;

                      //Verifica se vai zerar o gráfico
                      if (numeroDeRegistros==0)
                         frame.Construir(p);
                }

            } catch (Exception e) {
                  System.out.println("Excecao2 do "+NomeBarGraphAtual+": " +
e.getMessage());
                  e.printStackTrace();
            }

            //Retorna texto para indicar sucesso do processo
            if (resposta != null)
               return("A resposta foi:" + resposta);

            //Sensor não foi encontrado
            else
                 return(nome + " não encontrado!");
     }


     //Método que implementa a interface Atualizacoes,
     //que permite que o sensor atualize remotamente o gráfico
     public Object Atualizar(Ponto p) {

            frame.Construir(p);
            return("Atualização feita.");
     }
}
```

6.15. Classe PrincipalTrendGraph

```
//Classe PrincipalTrendGraph, que recebe o nome do prompt, e
//inicializa a janela que hospedará o TrendGraph

package graphs.Janelas;

public class PrincipalTrendGraph {

  boolean packFrame = false;  //Variável de validação da GUI

  //Construtor da classe atual
  public PrincipalTrendGraph(String nome) {

        //Inicializa a janela que contém o TrendGraph
        JanelaTrendGraph frame = new JanelaTrendGraph(nome);

        //Valida a janela
        if (packFrame)
           frame.pack();
        else
            frame.validate();
        frame.setVisible(true);
  }

  //Método principal da classe
  static public void main(String[] args) {

    //Inicializa a classe com o nome recebido pelo prompt
    new PrincipalTrendGraph(args[0]);

  }

}
```

6.16. Classe JanelaTrendGraph

```
//Classe JanelaBarGraph, que chama serve como hospedeira para a classe
//BarGraph, que está sendo incluída como um Bean

package graphs.Janelas;

import java.awt.*;
import java.awt.event.*;
import borland.jbcl.layout.*;
import graphs.TrendGraphs.*;

public class JanelaTrendGraph extends Frame{

  //Variáveis gerais da classe
  XYLayout xYLayout1 = new XYLayout();
  String titulo = null;

  TrendGraphPanel janela;  //Variável que indica o BarGraph que vai ser
adicionado à janela

  //Construtor da classe atual
```

```
    public JanelaTrendGraph(String nome) {

      titulo = nome;
      janela = new TrendGraphPanel();
      janela.Conecta(titulo);

      //Chama o método que define o layout da janela
      try {

          MontaGUI();

      } catch (Exception e) {
              System.out.println("Excecao1 do JanelaTrendGraph: " +
e.getMessage());
              e.printStackTrace();
      }

    }

    //Construção da GUI
    public void MontaGUI() throws Exception{

          //Definição do layout geral da janela
          this.setLayout(xYLayout1);
          this.setBackground(Color.lightGray);
          this.setSize(new Dimension(650, 290));
          this.setTitle(titulo);

          //Seta que os botões da janela geram eventos
          this.addWindowListener(new WindowAdapter() {
               public void windowClosing(WindowEvent e) {
                    System.exit(0);
               }
          });

          //Adiciona o TrendGraph na janela
          this.add(janela);

    }

}
```

6.17. Classe TrendGraphPanel

```
//Classe que implementa a GUI do TrendGraph, que está acertada para ser
usada como Bean

package graphs.TrendGraphs;

import java.awt.*;
import java.awt.event.*;
import borland.jbcl.layout.*;
import interfaces.*;
import dados.*;
import java.io.Serializable;

public class TrendGraphPanel extends Panel
                             implements ActionListener,
                                        Serializable {

  //Variáveis gerais da classe
  XYLayout  xYLayoutTrendGraph      = new XYLayout();
  Button    BotaoRegistrar          = new Button();
  Button    BotaoRemover            = new Button();
  Label     TextoComponente         = new Label();
  Label     TextoTaxa               = new Label();
  TextField CampoTaxa               = new TextField();
  TextField CampoComponenteServidor = new TextField();
  static    TextArea AreaDeTexto    = new TextArea();

  //Variáveis gerais para o gráfico impresso
  float MaxValor   = 100;        //Valor máximo do atuador na unidade
coerente
  float MinValor   = 0;          //Valor máximo do atuador na unidade
coerente
  float ValorReal  = 0;          //Valor atual do atuador na unidade
coerente
  int   ValorPixel = 0;          //Valor atual do atuador em pixel

  int    nColunas   = 10;                    //Número de colunas do gráfico
  int    nLinhas    = 5;                     //Número de linhas do gráfico
  int    xPoints[]  = new int[nColunas];     //Coordenadas x das colunas
  int    yPoints[]  = new int[nColunas];     //Coordenadas y das linhas
  String tempos[]   = new String[nColunas]; //Horários associados às
colunas

  int x       = 118;         //Cooredenada x do gráfico
  int y       = 15;          //Cooredenada y do gráfico
  int altura  = 120;         //Altura em pixel do gráfico
  int largura = 500;         //Largura em pixel do gráfico
  int xcoluna;               //Distância em x entre uma coluna e outra
  int ylinha ;               //Distância em y entre uma linnha e outra
  boolean bean = true;       //Variável que indica se estão sendo feitas
modificações por Bean

  TrendGraph TrendGraphMaster;  //Variável qeu seta o TrendGraph que vai
controlar este gráfico

  //Construtor da classe atual
  public TrendGraphPanel() {
```

```
    //Chama o método que define o layout da janela e inclui os componentes
acima
    try {

        MontaGUI();

    } catch (Exception e) {
          System.out.println("Excecao1 do TrendGraphFrame: " +
e.getMessage());
          e.printStackTrace();
    }


  }


  //Método que chama o TrendGraphMaster e conecta na rede como "nome"
  public void Conecta(String nome) {

    try {

         TrendGraphMaster = new TrendGraph(this, nome);

    } catch (Exception e) {
          System.out.println("Excecao1 do BarGraphPanel: " +
e.getMessage());
          e.printStackTrace();
    }

  }


  //Propriedades a serem ajustadas por Bean
  //Recepção do número de colunas
  public int getnColunas() {
         return nColunas;
  }

  //Reconstrução do gráfico com o novo número de colunas
  public void setnColunas(int novoNumero) {
         nColunas = novoNumero;
         bean = true;
         repaint();
  }


  //Recepção do número de linhas
  public int getnLinhas() {
         return nLinhas;
  }

  //Reconstrução do gráfico com o novo número de linhas
  public void setnLinhas(int novoNumero) {
         nLinhas = novoNumero;
         bean = true;
         repaint();
  }
```

```
    //Construção da GUI
    public void MontaGUI() throws Exception{

            //Definição do layout geral da janela
            this.setLayout(xYLayoutTrendGraph);
            this.setBackground(Color.lightGray);
            this.setSize(new Dimension(650, 290));

            //Definição do layout geral da janela
            CampoComponenteServidor.setBackground(Color.white);
            BotaoRegistrar          .setLabel("Registrar");
            TextoComponente         .setText("Nome do Sensor:");
            BotaoRemover            .setLabel("Remover");
            AreaDeTexto             .setBackground(Color.white);
            TextoTaxa               .setText("Taxa:");
            CampoTaxa               .setText("0");

            //Inclusão dos itens na janela, com respectivas coordenadas e
 tamanhos
            this.add(TextoComponente,          new XYConstraints( 10, 160,  99,
 20));
            this.add(CampoComponenteServidor, new XYConstraints( 10, 184, 180,
 30));
            this.add(TextoTaxa,                new XYConstraints(210, 160,  42,
 20));
            this.add(CampoTaxa,                new XYConstraints(210, 184,  47,
 30));
            this.add(BotaoRegistrar,           new XYConstraints( 10, 225, 122,
 30));
            this.add(BotaoRemover,             new XYConstraints(140, 225, 122,
 30));
            this.add(AreaDeTexto,              new XYConstraints(270, 160, 366,
 96));

            //Seta que os botões do gráfico geram eventos
            BotaoRemover  .addActionListener(this);
            BotaoRegistrar.addActionListener(this);

    }


    //Método que calcula o valor em pixel da altura do gráfico e associa ao
 horário de atualização
    public void Construir(Ponto p) {

            ValorReal  = p.ValorAtual;
            MinValor   = p.ValorMinimo;
            MaxValor   = p.ValorMaximo;
            ValorPixel = (int) (((ValorReal-MinValor)/(MaxValor-
MinValor))*altura);

            //Desloca os pontos para a direita, para incluir o novo ponto
            for (int i=nColunas-1;i>0;i--){
                yPoints[i] = yPoints[i-1];
                 tempos[i] =  tempos[i-1];
            }

            tempos[0] = p.Tempo;                    //Guarda o horário do ponto
atual
```

```
            yPoints[0] = y+altura - ValorPixel; //Põe o ponto atual na altura
certa do gráfico
            repaint();                          //Reconstrói o gráfico

        }


    //Método que constrói o gráfico
    public void paint(Graphics g) {

            xcoluna = largura / (nColunas-1);  //Definição da largura das
colunas
            ylinha  = altura / (nLinhas-1);    //Definição da altura das
linhas

            //Inicializa as coordenadas dos pontos se for a primeira vez que
            //o gráfico está sendo construído com o número de colunas e linhas
            if (bean == true) {
               for (int i=0; i < nColunas; i++) {
                   xPoints[i] = x + xcoluna*i;
                   yPoints[i] = y + altura;
                   tempos[i] = "00:00:00";
               }
               bean = false;
            }

            //Construção do efeito 3D do gráfico
            MontaMoldura(g);

            //Tracado da linha
            g.setColor(Color.red);
            g.drawPolyline(xPoints,yPoints,nColunas);

            //Textos de máximo, mínimo, intermediários
            g.setColor(Color.black);
            g.drawString("Máximo: "+MaxValor, x-110, y+5);
            g.drawString(" Mínimo: "+MinValor, x-110, y+altura+5);
          for (int i=1;i<=nLinhas-2;i++){
              g.drawString(CortaCasas(MinValor + ((MaxValor-
MinValor)/(nLinhas-1))*i), x-60, y+5+(ylinha*(nLinhas-1-i)));
            }

            //Textos dos tempos
            for (int i=0; i < nColunas; i++)
                 g.drawString(tempos[i], (x-25)+i*xcoluna, y+altura+15);

    }


    //Método que monta o efeito 3D do gráfico
    public void MontaMoldura (Graphics g){

            int alt  = altura + 1;
            int larg = largura + 1;

            //Linha cinza
            g.setColor(Color.gray);
            g.drawLine(x-2, y+alt, x-2     , y-2);
            g.drawLine(x-2, y-2  , x+larg+1, y-2);
```

```
            //Linha branca
            g.setColor(Color.white);
            g.drawLine(x-2     , y+alt+1, x+larg+1, y+alt+1);
            g.drawLine(x+larg+1, y+alt+1, x+larg+1, y-2);

            //Linha preta
            g.setColor(Color.black);
            g.drawLine(x-1, y+alt ,x-1     , y-1);
            g.drawLine(x-1, y-1   ,x+larg-1, y-1);

            //Fundo branco
            g.setColor(Color.white);
            g.fillRect(x, y, largura, altura);

            //Grid azul-claro
            g.setColor(Color.cyan);
            for (int i=0; i<nColunas; i++) {
                g.drawLine(x+i*xcoluna, y     , x+i*xcoluna   , y+altura);
            }
            for (int i=0; i<nLinhas; i++) {
                g.drawLine(x     , y+i*ylinha, x+largura, y+i*ylinha);
            }

    }


    //Método para lidar com os eventos gerados pelos botões
    public void actionPerformed(ActionEvent e) {

            Dados d = new Dados();  //Inicializacao do grupo de dados a serem
passados

            //Ação do botão 'Registrar'
            if (e.getActionCommand().compareTo("Registrar")==0) {

               //Envia os valores para o TrendGarphMaster e imprime o
resultado na janela
               d.Acao = "registrar";
               d.Intervalo = new Float(CampoTaxa.getText()).floatValue();

               if (d.Intervalo > 0)
                  d.TipoAtualizacao = "poramostragem";
               else
                  d.TipoAtualizacao = "porevento";

               String resposta =
TrendGraphMaster.Registro(CampoComponenteServidor.getText(),d);
               AreaDeTexto.append(resposta+"\n");
            }

            //Ação do botão 'Remover'
            if (e.getActionCommand().compareTo("Remover")==0) {

               //Envia os valores para o TrendGarphMaster e imprime o
resultado na janela
               d.Acao = "remover";
               d.Intervalo = new Float(CampoTaxa.getText()).floatValue();

               if (d.Intervalo > 0)
                  d.TipoAtualizacao = "poramostragem";
```

```
            else
               d.TipoAtualizacao = "porevento";

            String resposta =
 TrendGraphMaster.Registro(CampoComponenteServidor.getText(),d);
            AreaDeTexto.append(resposta+"\n");

            bean = true;  //Indica que a próxima vez é preciso inicializar
 os pontos
         }

   }


   //Método que corta as casas do valor impresso na tela
   public static String CortaCasas(float Valor) {

         int Inteiro = (int) (Valor);
         int Casas = (int) ( ( Valor - ((float) (Inteiro)) ) * 100 );
         float Resultado = ((float) (Inteiro)) + (((float) (Casas))/100);
         return ("" + Resultado);

   }

}
```

6.18. Classe TrendGraph

```
//Classe TrendGraph, que implementa a interface Atualizacoes,
//para poder ter seus dados atualizados remotamente

package graphs.TrendGraphs;

import java.rmi.*;
import java.rmi.server.*;
import interfaces.*;
import dados.*;

public class TrendGraph extends UnicastRemoteObject
                  implements Atualizacoes {

  //Variáveis gerais da classe atual
  static  TrendGraphPanel frame      ;        //Variável que corresponde à
GUI do TrendGraph
  static  String NomeTrendGraphAtual = null;  //Nome que o TrendGraph vai
ter
  static  int numeroDeRegistros      = 0;     //Variável que armazena o
número de registros feitos


  //Contrutor do TrendGraph, que chama a classe
  //TrendGraphFrame, que é a GUI do TrendGraph
  public TrendGraph(TrendGraphPanel frame1, String nome) throws
RemoteException {

        super();                //Chama o construtor do UnicastRemoteObject

        frame = frame1;                 //Recebe o TrendGraphPanel enviado
```

```
            NomeTrendGraphAtual = nome;    //Armazena o nome do BarGraphPanel
    enviado

            //Bloco que põe o TrendGraph na rede, para ser localizado
    remotamente, através do "nome"

            //Definição do Gerenciador de Segurança do RMI
            if (System.getSecurityManager() == null)
               System.setSecurityManager(new RMISecurityManager());

            //Coloca o TrendGraph na rede como "nome"
            try {

                Naming.rebind(NomeTrendGraphAtual, this);

            } catch (Exception e) {
                    System.out.println("Excecao2 do " + NomeTrendGraphAtual +
    ": " + e.getMessage());
                    e.printStackTrace();
            }

            //Imprime o resultado na janela do TrendGraphPanel
            frame.AreaDeTexto.append(NomeTrendGraphAtual+" na rede!!\n");

      }


      //Método que chama a classe Sensor para se inscrever na sua lista
      public static String Registro(String nome, Dados d) {

            String resposta = null;                  //Texto que o método
    retorna
            Ponto p         = new Ponto();           //Valores que irá receber
    para atualização
            d.Componente    = NomeTrendGraphAtual; //Indica o nome do
    TrendGraph que será passado para o sensor

            //Definição do Gerenciador de Segurança do RMI
            if (System.getSecurityManager() == null)
               System.setSecurityManager(new RMISecurityManager());

            //Procura o sensor e efetua o registro
            try {
                Inscricoes objremoto = (Inscricoes) Naming.lookup(nome);
                p = objremoto.AlteraInscricao(d);

                //Definicao do texto a ser retornado
                if (d.Acao.compareTo("registrar")==0) {
                   resposta = NomeTrendGraphAtual + " conectado ao " + nome +
    ".";

                   numeroDeRegistros = numeroDeRegistros + 1;

                   frame.Construir(p);  //Atualiza o aspecto do gráfico

                } else {
                      resposta = NomeTrendGraphAtual + " retirado do " + nome
    + ".";

                      numeroDeRegistros = numeroDeRegistros - 1;

                      //Verifica se vai zerar o gráfico
```

```
                    if (numeroDeRegistros == 0) {

                        //Loop para zerar o(s) ponto(s)
                        for (int i=0;i<frame.nColunas;i++)
                            frame.Construir(p);
                    }
            }

        } catch (Exception e) {
                System.out.println("Excecao1 do "+NomeTrendGraphAtual+": "
+ e.getMessage());
                e.printStackTrace();
        }

        //Retorna texto para indicar sucesso do processo
        if (resposta != null)
           return("A resposta foi:" + resposta);

        //Sensor não foi encontrado
        else
             return(nome + " não encontrado!");

  }


  //Método que implementa a interface Atualizacoes,
  //que permite que o sensor atualize remotamente o gráfico
  public Object Atualizar(Ponto p) {

        frame.Construir(p);
        return("Atualização feita.");

  }

}
```

7. Bibliografia

- LINDEN, Peter van der. *Just Java*. São Paulo: Makron Books, 1998-07-02
- VALLE, André, GUIMARÃES Cláudia. *Java Manual de Introdução*. Rio de Janeiro: Axcel Books do Brasil, 1996
- VANHELSUWÉ, Laurence. *Mastering JavaBeans*. San Francisco: Sybex, 1997

Homepages:

- http://www.java.sun.com/ - Web site oficial da Linguagem Java e tópicos relacionados à mesma, pela Sun Microsystems
- http://www.javasoft.com/docs/books/tutorial/index.html - Web site contendo um Tutorial sobre os principais aspectos da linguagem Java, pela Sun Microsystems
- http://www.iinet.net.au/~ianw/ - Web site sobre informação de SCADA