ARQUITETURA E PROTOCOLO PARA APLICAÇÃO VISUAL THIN CLIENT

Vanius Roberto Bittencourt

Instituto de Ciências Exatas e Tecnológicas - Universidade Feevale – Novo Hamburgo, RS. E-mail: vanius@gmail.com

RESUMO
Neste artigo são estudadas diferentes soluções para uma necessidade cada vez mais comum na atualidade – executar aplicações thin client visuais. Para solucionar problemas em comum é fornecida uma proposta aberta de arquitetura e protocolo para aplicação cliente-servidor thin client com visual em janelas.
Palavras-chave: aplicações thin client - Interface visual - Arquitetura cliente servidor

ARCHITECTURE AND PROTOCOL FOR APPLYING VISUAL THIN CLIENT

ABSTRACT
In this paper are studied different solutions to an increasingly common need today - to run applications thin client visuals. To solve common problems is provided a proposed open architecture and protocol for client-server application with visual thin client on windows.
Keywords: thin client applications - Visual interface - Client Server Architecture

INTRODUÇÃO

Existem vários meios e modelos de execução de programas, um dos mais úteis é o cliente-servidor. Neste modelo o programa é executado em dois pontos através de uma rede de computadores. O lado do cliente é responsável pela interface com o usuário e o lado do servidor, responsável pelo processamento computacional e armazenamento de dados. A comunicação e transferência de dados entre os lados são definidas através de um protocolo, geralmente na camada de aplicação.

O cliente pode ser uma parte fixa da aplicação como uma extensão ao servidor, formando a aplicação. Neste conceito, o cliente pode executar localmente processamento e regras de negócio e utiliza o servidor apenas como repositório de dados. Outra arquitetura para o modelo de cliente-servidor é o cliente apenas "representar" a execução da aplicação do servidor, de modo que este seja uma mera interface visual. Dentro deste contexto, o mesmo cliente pode representar e executar diferentes aplicações, já que o mesmo é contido na integra no servidor. Este conceito será tratado de *thin client*.

A interface visual de aplicações pode ser de vários tipos. Um dos primeiros a surgir foi a interface chamada caractere, sendo a tela formada por uma matriz normalmente de 80 colunas por 24 linhas e cada posição podendo exibir um caractere da tabela ASCII. Nesta interface a resolução da tela não é relevante, pois sempre são considerados os caracteres – e não é possível representar símbolos ou desenhos. Posteriormente, surgiu a interface gráfica, onde poderiam ser representados caracteres sem estar dentro de uma matriz, e também poderiam ser exibidos símbolos e outros desenhos. Através deste tipo de interface surgiu o conceito de aplicações em janela, onde formulários poderiam ficar dispostos lado a lado e um por cima de outro.

Este documento irá rever a evolução interface visual da arquitetura cliente-servidor. Será demonstrado que na atualidade não há uma padronização para este fim – as soluções de hoje se baseiam em extensões de tecnologias que não possuem esse propósito. O resultado é que existe grande dependência de rotinas extras e ferramentas proprietárias, tornando seu desenvolvimento bastante complexo. Será proposta uma nova tecnologia, baseada em uma especificação de protocolo.

EVOLUÇÃO DA ARQUITETURA CLIENTE-SERVIDOR E INTERFACE COM USUÁRIO

Na década de 70 e 80 era forte a utilização de clientes leves em sistemas Unix, os chamados "terminais burros". A interação com o usuário era em formato caractere (texto) onde eram trafegados apenas os caracteres da tela e eventos de teclados pelo usuário. Todo processamento ocorria no servidor. Para permitir uma flexibilidade da execução de diferentes clientes foram adotados padrões de comunicação entre o servidor e cliente. Um destes padrões foi o VT100, largamente utilizado e com variações (Parallax, 2010). Ainda na década de 80, surgiram os primeiros conceitos de telas gráficas, diferentes das telas caracteres. O MIT criou um protocolo de comunicação entre terminais semelhante ao VT100. O protocolo se denominava System Window X, ou simplesmente X (Scheifler; Gettys, 1986). Este protocolo foi proposto para ser independente de hardware e plataforma. Nos sistemas Unix este protocolo foi largamente adotado e é utilizado até hoje. Apesar de ser um protocolo para ser utilizado em uma arquitetura cliente-servidor, é praticamente um padrão de desenvolvimento de telas gráficas para aplicações *desktop*. Considerando a necessidade atual de soluções em aplicações gráficas na

internet, seu uso é questionável. Este protocolo gera muitas interações entre o cliente e servidor, que consiste em trafegar definições detalhadas da tela. Com isso, este protocolo é praticamente inviável de ser utilizado em meios externos e de velocidade variada, tal como a internet.

Durante a década de 90, com a popularização do sistema operacional Windows, houve uma onda de surgimento de aplicações "gráficas", basicamente aplicações *desktop*. O conceito de aplicação "caractere" ou "texto" passou a ser considerado obsoleto. Surgiram tecnologias e linguagens tal como VB, Delphi, Visual C, etc. para o desenvolvimento dessas aplicações. Houve também uma forte evolução dos processadores – principalmente com o processador Pentium - e placas de vídeo. Outros conceitos, tal como "terminal burro", foram considerados desnecessários devido ao aumento de poder de processamento das estações. Com as aplicações visuais também se aumentou a dependência sobre o sistema operacional. Mesmo neste panorama de aplicações rodando em estações surgiram alguns protocolos para permitir interface gráfica remota, tal como RDP e VNC. Estes protocolos consistem em transferir toda área da tela, mapeando cada *pixel*.

Mas a partir do final da década de 90 houve a consolidação e massificação do uso da internet como meio de comunicação informatizado. Isso causou impacto em vários aspectos. Um deles foi o surgimento de aplicações via internet, utilizando o protocolo HTTP. Então houve uma nova onda, onde surgiram linguagens tal como PHP, ASP, JS, Python, etc. Com as novas aplicações via internet – chamadas de aplicações "*web*" – ganhou-se a flexibilidade de funcionar com diferentes sistemas operacionais e não depender de instalação na estação. As soluções *web*, por serem baseadas no formato HTML, são visualmente bastante limitadas, já que este formato foi desenvolvido para exibição de textos. Com o passar do tempo, o HTML foi sendo estendido, muitas vezes de forma irregular. Surgiram tecnologias para contornar restrições do HTML, tal como o Ajax, onde a página HTML não necessita ser totalmente recarregada. Mas a sua aplicação é bastante complexa, por isso surgiram diversas *frameworks* para auxiliar sua utilização. Com a necessidade de aplicações mais ricas surgiram soluções alternativas tal como o Flash, que inicialmente foi fortemente utilizada para desenhos de banners e outras visualizações gráficas. Com o passar do tempo verificou-se que essa tecnologia poderia ser utilizada para desenvolvimento de aplicações *web*. As aplicações *web* com gráficos avançados foram chamadas de *Rich Application*.

Visualizando a dificuldade em desenvolver aplicações visuais para internet, a Adobe – desenvolvedora do Flash – criou uma ferramenta chamada Flex, onde o programador desenvolve a aplicação em linguagem Java, é executada no servidor e sua visualização no cliente é feita através do *plugin* de Flash, que está instalado em quase todos navegadores de internet. Com isso o resultado é uma aplicação rica, semelhante a aplicações *desktop*, sem necessidade de domínio do formato HTML. Existem outras soluções para aplicações Java, tal como CaptainCasa, que pode tanto exibir a tela do cliente em navegador quanto em janelas *desktop*. Assim como estas existem outras soluções proprietárias, que aplicaram suas próprias soluções de servidor, cliente e protocolo de comunicação entre eles. Verifica-se que a estrutura dessas soluções e seu protocolo são distintos, porém pode-se extrair um modelo e propor uma normalização.

Na internet muitas tecnologias e protocolos são padronizados, de forma aberta e livre. Para evitar o crescimento muito grande de tecnologias proprietárias o W3C está definindo

modificações para padrão HTML, agora na versão 5 (Pilgrim, 2010). Estão sendo definidas características do protocolo para aplicações visuais e de vídeo.

São perceptíveis as mudanças das tendências sobre desenvolvimento de aplicações. Hoje há uma busca pelo resgate de aspectos interessantes adotados antes das aplicações *desktop* dos anos 90, que tendem a ser executadas em plataformas independentes e sobre clientes leves. Na internet existem diferentes protocolos de comunicação, mas nenhum permite a execução de aplicações remotas com aparência visual em janelas. As atuais aplicações *web* pelo protocolo HTTP necessitam diversos controles adicionais para o servidor e o cliente representarem os objetos de maneira equivalentes, já que a comunicação não é contínua - os estados da conexão não são preservados, chamados de *stateless*. Alguns controles incluem uso de *cookies* ou de *frameworks* facilitadores. A comunicação sempre deve ser feita pelo cliente ao servidor, onde este retorna informações de acordo com a passagem de parâmetros. Em nenhum momento o servidor inicia um envio informação ao cliente, sempre este deve requisitar. O cliente e o servidor devem ter um mecanismo para controlar a sessão de cada cliente e representação de seus objetos – o conceito chamado *RestFull*.

Hoje não há uma implementação definitiva e padronizada para aplicações gráficas em janelas (*windowed*) de arquitetura cliente-servidor executadas por um cliente-leve, através de um protocolo aberto independente de plataforma e sistema operacional. Observa-se que o uso do HTML está sendo explorado na forma de aplicações comerciais, apesar de não ter sido criado para isso. Novas especificações do HTML irão permitir desenhos livres, tal como no Flash. Para permitir o mesmo visual em janelas deverá ser desenvolvida uma camada para isso.

PROPOSTA DE ARQUITETURA E PROTOCOLO PARA APLICAÇÃO *THIN CLIENT* VISUAL

O protocolo proposto será denominado TBAP – *Thinclient Business Application Protocol*. A proposta deste modelo é uma especificação aberta de protocolo responsável pela representação visual de aplicações simples, contendo formulários (janelas) para entrada e exibição de campos para aplicações empresariais (*business aplication*). Não será tentado contemplar visualização rica na íntegra (*rich applications*) tal como recursos de animações, vídeos, desenhos vetoriais ou desenhos livres (*draw canvas*). Os possíveis formatos de campos serão denominados componentes, sendo contemplados os de uso comum por aplicações deste segmento, tal como: texto não editáveis (*label*), caixa de texto editável (*edit*), caixa de texto com seleção de opções (*combobox*), caixa de marcação (*checkbox*), botão (*button*) e tabela (*grid*). Além disso, o protocolo também permitirá a utilização de caixa de diálogo de uso comum, tal como aviso, alerta, confirmação, entrada de texto (*inputbox*), abrir arquivo, salvar arquivo e imprimir.

Não haverá tráfego da imagem em *bitmap* da aplicação, serão transferidos somente os atributos básicos de formulários e controles, tal como posição, tamanho e características de aceitação. Com isso o tráfego será bastante reduzido, tornando-o eficiente. Os componentes visuais devem ter um identificador numérico, com isso, toda a comunicação contendo mudanças de atributos ou eventos deve ter o identificador a qual componente se refere. O servidor é responsável para gerar o número do identificador.

Este protocolo deve ser implementado sobre outro protocolo de transporte, tal como TCP puro. A arquitetura é dividida em partes (camadas) que funcionam de forma acopladas, podendo ter desenvolvimento independente, conforme ilustrado na figura 1.

Tanto no servidor quanto no cliente devem existir mecanismos de comunicação (transporte) que encapsulam o protocolo TBAP. No cliente deve ter uma camada que traduz todos os comandos do protocolo para a camada de representação visual. Esta camada visual pode ter várias implementações, para diferentes plataformas visuais. No servidor deve existir uma camada que gerencia as aplicações, tal como um servidor de aplicação. Esta camada faz a ligação dos métodos de aplicação com a representação visual, que é transferida para a camada de protocolo.

Os métodos de aplicação residem numa camada em comum às aplicações, funcionando como um *framework*. A camada de aplicação é onde a aplicação customizada é executada. Essa aplicação deve respeitar uma interface, que garante que ocorrerá a interoperabilidade com a camada base de aplicação.

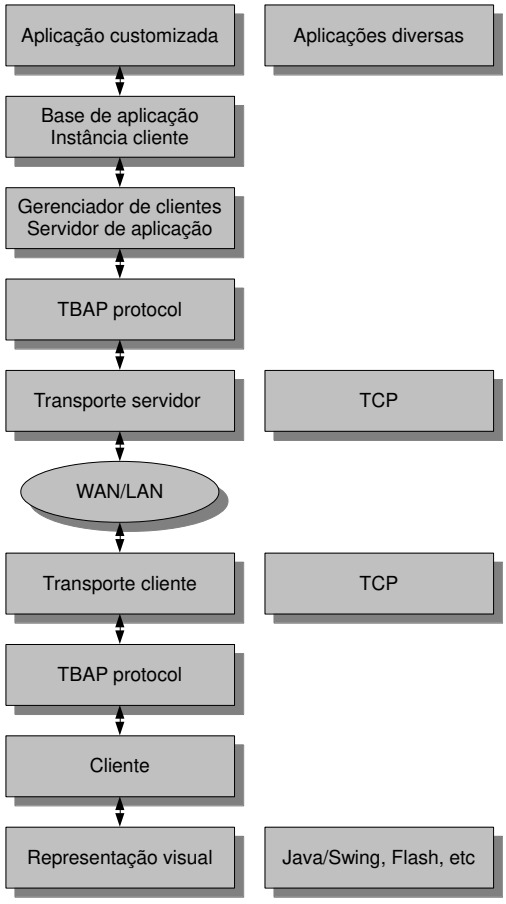


Figura 1. Estrutura de fluxo da arquitetura

Fluxo de comunicação

O cliente deve receber parâmetros de localização do servidor, tal como IP e porta. Além disso, deve ter o nome da aplicação que o servidor deve executar, assim como parâmetros adicionais. O cliente acessa sua camada de comunicação para fazer a conexão com o servidor, onde passa o nome da aplicação e parâmetros adicionais.

O servidor recebe a conexão e cria uma instância da camada de comunicação com o cliente. Esta camada recebe o nome da aplicação que deve executar. O servidor verifica se o nome está registrado, caso afirmativo, é criado instância da aplicação no servidor. O fluxo pode ocorrer *fullduplex* (ambos os sentidos simultaneamente, cliente-servidor).

A comunicação para o protocolo poderá ocorrer em forma de chamadas, tal como *Remote Procedure Call* (RPC). O protocolo permite comandos tal como criar componente, troca de atributo, etc. Para evitar tráfegos e espera (*leadtime*) para cada comando, o protocolo permite que vários comandos sejam empilhados em uma única chamada. Com isso, a construção de uma janela com vários controles poderá ser feita em uma única comunicação do servidor ao cliente, onde serão transmitidos os atributos dos controles e o cliente apenas deverá retornar o sucesso do processamento uma vez. A conexão de utilização tem a proposta de ser contínua (*statefull*) e de via dupla (*fullduplex*), diferente do exercido pelo HTTP nos sistemas *web* HTML.

Para possibilitar o uso destes aspectos, o protocolo proposto será executado sobre o protocolo TCP "puro" (Farley, 1998). A comunicação TCP deve ser iniciada pelo cliente, que conecta com um servidor através de uma identificação IP. No servidor deve existir um serviço de escuta de porta, que para cada nova conexão de cliente mantenha a conexão e permita novas conexões, através de um

mecanismo de *fork*. Para cada cliente conectado, o servidor deve criar um novo objeto para representar e gerenciar esta comunicação de forma independente a cada cliente.

O TCP permite tráfego de texto de ambos os sentidos, a qualquer momento, enquanto houver conexão. A escuta dos textos oriundos da conexão deve estar numa *thread* já que a mesma pode ocorrer simultaneamente a processamentos do cliente ou servidor. Através dos textos transmitidos serão definidas regras e formatos que devem respeitar o protocolo - com isso será possível uma flexibilidade da execução tanto do servidor quanto do cliente, independente de hardware ou plataforma, desde que seja respeitada a especificação das mensagens e seja sobre o protocolo TCP.

Formato

Os textos transmitidos do protocolo devem respeitar um formato em que seja possível identificar campos e valores. Para isso será adotado o formato JSON (*JavaScript Object Notation*), que em comparação ao XML se demonstra mais simplificado (Miller; Vandome; McBrewster, 2009). Cada mensagem do protocolo pode conter vários atributos, conforme figura 2.

```
{
"mensageName":"propertyChange",
  "array" : [
        {
"property1" : "1" },

        {
"property2" : "2" }  ] }
```

Figura 2. Comando e atributos do protocolo em formato JSON

O formato JSON é muito utilizado em rotinas escritas em JavaScript e Java J2EE, porém existem bibliotecas de manipulação para quase todas as linguagens e diferentes plataformas.

O protocolo funcionará através de mensagens de comando que são enviadas para o computador remoto, onde são interpretas e executadas no conceito de RPC. Um dos atributos da mensagem deve ser o "nome", que contém o objetivo da mensagem. Além do nome do objetivo da mensagem ela pode possuir campos de parâmetros, que são diferentes entre si, ou seja, cada tipo de mensagem possui seus parâmetros com nomes estabelecidos pela regra do protocolo. Cada fluxo pode ser uma mensagem do protocolo, ou conter várias mensagens. O fluxo de protocolo será a representação em formato texto UTF8 de um objeto JSON.

Interface gráfica

A visualização da aplicação do lado cliente será em formato de janelas, que podem ser exibidas simultaneamente. Cada janela deve possuir um identificador, definido pela aplicação no lado do servidor. E cada janela pode conter vários componentes visuais. O componente deve possuir um identificador único, inclusive dentre componentes de outras janelas.

Cada componente possui propriedades tal como texto, largura, posição, etc. A maioria dos componentes representará uma informação visível ao usuário, que será denominado "campo". Estes componentes também podem permitir o usuário alterar a informação, tal como texto do campo. As propriedades podem ser consultadas e alteradas pela aplicação do servidor. Para isso nas mensagens deve ser informado o identificador do componente. Algumas propriedades são alteradas somente via aplicação, nunca pelo usuário. Outras propriedades podem ser ditadas se podem ou

não ser alterada pelo usuário, tal como comportamento “somente leitura”.

Ações do usuário na interface geram mensagens ao servidor, denominados “eventos”. Os eventos possuem o nome da ação e o id do componente. O evento também pode conter a valor do campo alterado pelo usuário.

Fluxo de mensagens, retornos e eventos

As mensagens podem ser síncronas ou assíncronas, dependendo se o tipo da mensagem necessita de um retorno. Nos casos das mensagens síncronas o lado oposto ao cliente-servidor retorna uma mensagem de resposta. As mensagens síncronas devem aguardar obrigatoriamente o retorno, ficando num estado de aguardo. A mensagem de retorno pode conter diferentes campos de retorno.

Mensagens síncronas possuem um parâmetro com um identificador de fluxo, sendo que a mensagem de resposta retorna o mesmo identificador de fluxo. Com isso podem ocorrer diferente fluxos de comando e resposta paralelamente. Uma mensagem síncrona não deve impedir o tráfego de outras mensagens. Sugere-se a implementação de *threads* como mecanismo de processamento paralelo para a recepção e tratamento de mensagens.

Cada mensagem recebida deve gerar uma *thread* de processamento. O mecanismo de recepção de mensagem deve funcionar numa *thread* diferente das executadas pelo tratamento das mensagens. Ao ser enviada uma mensagem, se a mesma for síncrona, então a thread em execução deve entrar em estado de espera (*wait*). Ao receber uma mensagem de resposta, a rotina de recepção deve identificar e recuperar qual a *thread* está em aguardo, causando a notificação (*notify*) – com isso continuar a execução. Para isso recomenda-se um mecanismo de mapa associativo chave-valor com o identificador de fluxo e *threads*.

Com estas características de controle de fluxo será possível empilhar estados de comandos e respostas. Um exemplo dessa situação é quando queremos interagir com o usuário dentro de um processamento de evento:

Cliente swing visível ao usuário		Servidor remoto executando aplicação de negócio
[thread 1 inicia] Usuário clica no ícone de fechar a janela. O cliente swing envia evento síncrono, questionando à aplicação de negócio se pode fechar a janela através do evento “canClose()”. [thread 1 espera]	->	[thread 1 inicia] A aplicação de negócio recebe o evento, porém necessita que usuário confirme a operação.
[thread 2 inicia] O cliente swing recebe mensagem “inputBox()” e exibe janela de confirmação . O cliente swing não responde ao servidor enquanto o usuário não interagir.	<-	Então a aplicação não responde no primeiro momento o evento “canClose()”, primeira envia mensagem síncrona para abrir janela de diálogo para o usuário “inputBox()” e aguarda resposta. [thread 1 espera]
Após o usuário confirmar é respondia a mensagem para servidor “inputBox() = yes”. [thread 2 finaliza]	->	[thread 1 acorda] A aplicação recebe a resposta de confirmação pelo usuário “inputBox() = yes”
[thread 1 acorda] O cliente swing recebe a resposta “canClose()=yes” então fecha a janela visível ao usuário. [thread 1 finaliza]	<-	Então agora pode responder a mensagem para o cliente “canClose()=yes”. [thread 1 finaliza]

Execução

O servidor e o cliente devem ser programas estáticos que permitam executar diferentes aplicações. Para isso, no servidor deve ser possível localizar as aplicações através de um mecanismo de "aplicações registradas". As aplicações executadas no servidor devem implementar uma interface que permita ser registrada ao servidor. Com isso o servidor possuirá uma lista de aplicações, que a conexão do cliente deverá indicar.

A aplicação executada no servidor deve acessar a interface com o usuário através de classes *proxy*, que apenas serve como um meio de transporte para classes reais que serão instanciadas no lado do cliente (Freeman; Freeman; Sierra; Bates, 2004).

Quando o cliente conecta do lado do servidor é criada uma *thread* para processar a aplicação. O cliente, ao ser iniciado, deve transmitir ao servidor qual aplicação deve executar no servidor. O servidor busca as aplicações registradas e, quando encontrado, é instanciado o objeto, sendo executado dentro da *thread* criada para este cliente.

IMPLEMENTAÇÃO

Para avaliar o modelo proposto foi desenvolvido protótipo que implementa as funcionalidades de servidor, cliente e aplicação de negócio. Foi utilizada a linguagem Java SE 7 e ambiente de desenvolvimento Netbeans IDE 7.0. A comunicação de rede foi feita através da classe *Socket* que manipula TCP puro. A interface gráfica utilizada é a *Swing*. Para manipular objetos JSON foi utilizada a biblioteca JSON-lib.

As classes criadas foram agrupadas em três pacotes, que podem gerar bibliotecas distintas: *server*, *client* e *common*. As bibliotecas *server* e *cliente* representam respectivamente os módulos distintos para executar no lado do servidor e do lado do cliente. Estas bibliotecas publicam classes para abstrair a manipulação do protocolo, fornecendo uma API para o desenvolvimento da aplicação de negócio e possíveis clientes visuais para outras plataformas.

A biblioteca *common* contém o núcleo de recepção e tratamento de mensagens do protocolo, utilizadas igualmente pelos lados servidor e cliente. Através do tráfego de texto pelo TCP são serializados e desserializados objetos que representam as mensagens. Há o controle de fluxo e *threads* para mensagens síncronas. Também existe mecanismo para agrupar mensagens em lote e fundir mensagens semelhantes a fim de melhorar a eficiência de tráfego.

A versão beta deste projeto protótipo, que implementa o protocolo TBAP nas camadas cliente e servidor, está disponível com codinome "felinelayer" em http://code.google.com/p/feline-layer/source/browse/

Foi criada uma aplicação de negócio exemplo que se utiliza da API para manipulação do protocolo. A aplicação é executada do lado servidor. Com isso foi simulado seu uso prático e bastante próximo ao objetivo do artigo. Abaixo o código fonte:

```
public class Sample extends br.com.felinelayer.server.application.Application {
   private Frame f;
   public Sample() {
      setApplicationName( "Sample" );
   }
   @Override
   protected void doStart() {
      String nome = showInput( "Digite seu nome:" );
      showMessage( "Bem vindo " + nome + "!" );
      beginBatch();
      f = addFrame().setWidth( 520 ).setHeight( 200 );
      f.addLabel().setCol( 1 ).setText( "Cliente" );
      f.addEdit().setCol( 3 ).setText( nome ).setWidth( 200 );
      f.addRow();
      f.addLabel().setCol( 1 ).setText( "Código" );
      f.addLabel().setCol( 5 ).setText( "Descrição" );
      f.addLabel().setCol( 10 ).setText( "Valor" );
```

```
        f.addRow();
        f.addEdit().setCol( 1 ).setText( "000001" ).setWidth( 60 );
        f.addEdit().setCol( 5 ).setText( "MacBook Air" ).setWidth( 80 );
        f.addEdit().setCol( 10 ).setText( "2.999,99" ).setWidth( 60 );
        f.addRow();
        f.addButton().setCol( 1 ).setText( "Gravar" ).setWidth( 80 );
        f.addRow();
        f.addButton().setCol( 1 ).setText( "Fechar" ).setWidth( 80 );
        endBatch();
    }
    @Override
    protected void doMessageReceived( Message message ) {
        String eventName = "";
        if (message.isPropertyExists( Protocol.EVENT_NAME) )
            eventName = message.getValueByName( Protocol.EVENT_NAME );
        if ( eventName.equals( Protocol.EVENT_BEFORE_CLOSE ) ) {
            String resposta = showInput( "Pode fechar? (\"sim\")" );
            if ( resposta.equalsIgnoreCase( "sim" ) )
                message.setValueByName( Protocol.EVENT_BEFORE_CLOSE_CANCLOSE, Protocol.YES );
            else
                message.setValueByName( Protocol.EVENT_BEFORE_CLOSE_CANCLOSE, Protocol.NO );
        }
    }
}
```

Na figura 3 é observada a janela que foi transmitida pela aplicação servidora, exibida do lado cliente. O resultado foi bastante semelhante nos sistemas operacionais Windows 7 e Linux Ubuntu 11.04.

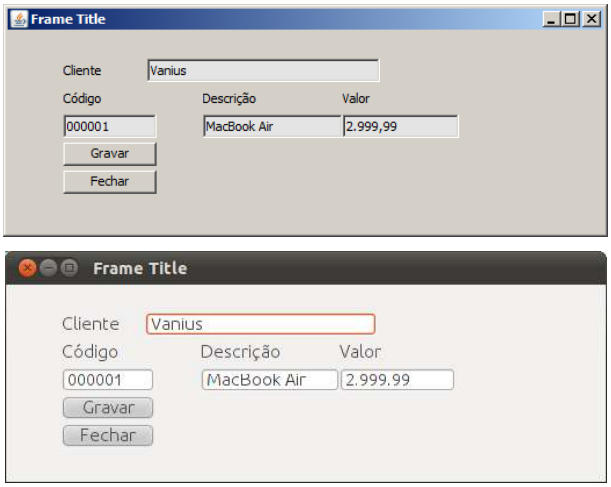


Figura 3. Janela que foi definida no lado do servidor é exibida do lado cliente

Abaixo um fragmento do tráfego das mensagens em formato JSON, entre cliente-servidor, com registro data/hora e sentido de fluxo enviado ou recebido (*sent* ou *received*):

Servidor e aplicação de negócio		Cliente Swing
05:57:20:045 Sent : {"MessageName":"array","array":[{"ComponentHeight":"200","ID":"1","ComponentClass":"Frame","ComponentWidth":"520","MessageName":"CreateComponent"},{"ComponentText":"Cliente","ParentFrameID":"1","ComponentTop":"24","ID":"2","ComponentClass":"Label","ComponentLeft":"48","MessageName":"CreateComponent"},{"ComponentText":"Vanius","ParentFrameID":"1",...]}	->	05:57:20:051 Received : {"MessageName":"array","array":[{"ComponentHeight":"200","ID":"1","ComponentClass":"Frame","ComponentWidth":"520","MessageName":"CreateComponent"},{"ComponentText":"Cliente","ParentFrameID":"1","ComponentTop":"24","ID":"2","ComponentClass":"Label","ComponentLeft":"48","MessageName":"CreateComponent"},{"ComponentText":"Vanius","ParentFrameID":"1",...]}
05:57:25:406 Received : {"EventName":"EventBeforeClose","EventComponentID":"1","CanClose":"Yes","InquiringId":"1","MessageName":"Event"}	<-	05:57:25:405 Sent : {"EventName":"EventBeforeClose","EventComponentID":"1","CanClose":"Yes","InquiringId":"1","MessageName":"Event"}
05:57:28:902 Sent : {"EventName":"EventBeforeClose","EventComponentID":"1","CanClose":"Yes","InquiredId":"1","MessageName":"Event"}	->	05:57:28:903 Received : {"EventName":"EventBeforeClose","EventComponentID":"1","CanClose":"Yes","InquiredId":"1","MessageName":"Event"}

Com a implementação do protótipo foi possível observar:

- Êxito na transferência de atributos dos objetos visuais para o cliente e eventos gerados para o servidor, inclusive em rede heterogênea.
- Foi possível ter uma aplicação de negócio executada remotamente com aparência de aplicação desktop.
- Fácil implementação na aplicação de negócio através do uso de APIs que abstraem o protocolo e a comunicação.
- Não houve necessidade de utilização conhecimentos em HTML, XML ou Java Script.
- Diferente do HTML nativo não há comportamento de *refresh* integral para atualizar campos.
- Por utilizar protocolo TCP *statefull* então não há necessidade da aplicação de negócio ter mecanismo de persistência RestFull.
- Não há tráfego de *pixels* da janela da aplicação, inclusive para alterar campos e posição da janela.
- A mesma biblioteca de servidor e de cliente podem executar diferentes aplicações de negócio simultaneamente, inclusive em sistemas operacionais diferentes.
- Pelo fato do protocolo utilizar padrões abertos e difundidos, tal como TCP e textos UTF8 representando objetos JSON, poderia existir servidor e cliente implementados em diferentes plataformas.

CONCLUSÃO

Este artigo estudou o uso em conjunto do modelo cliente-servidor *thin client* com interface gráfica com janelas. O seu uso pode resultar em uma solução bastante flexível quanto à plataforma de execução e uma boa experiência de uso por parte do usuário. Hoje não há uma padronização de tecnologia para este fim – existem várias tentativas de expandir protocolos atuais e cada vez mais surgem ferramentas ou rotinas proprietárias para alcançar este objetivo.

O protocolo TBAP se diferencia do Ajax em uma série de aspectos. O Ajax não é um protocolo, e sim um conjunto de tecnologias. Foi criado para contornar restrição do HTML que obriga o *refresh* e recarga integral do HTML para atualizações (Holzner, 2008). Ou seja, serve para incrementar uma tecnologia e não tem por objetivo ser uma tecnologia inovadora e alternativa. Por depender do protocolo HTTP, o Ajax faz requisições de sentido único, quando requisita transferência de conteúdo através de funções em JavaScript (*XmlHttpRequest*). O Ajax utiliza linguagens de marcação XML e HTML, que não representam necessariamente uma visualização de tela em janelas. Sua utilização pode ser complexa (Powell, 2008), por isso seu uso é sempre em atrelado à *frameworks*. É utilizado para desenvolvimento de sistemas que resultam em páginas HTML com transporte sobre o HTTP, que não é a proposta deste trabalho.

A proposta deste artigo também não se assemelha às soluções de aplicações ricas. Dentre as soluções atuais podemos citar Flash, Silverlight e JavaFX. As três tecnologias estão fortemente associadas ao uso de linguagem e *framework* específicos. Outro aspecto em comum é que são utilizadas para criação de desenhos vetoriais em *canvas*. Ou seja, inicialmente não tem objetivo para aplicações com janelas e campos. O tráfego entre o servidor e cliente não possuem especificação aberta na forma de protocolo. O HTM5 - que é a tecnologia de aplicação rica com padrão aberto e independente de linguagem - permite o desenho de *canvas*, porém igualmente não tem objetivo de aplicações em janelas e campos. É baseada na linguagem de marcação HTML4, que permite entrada de

campos, mas nativamente não permite hierarquias de janelas.

O objetivo do trabalho é a especificação de regras para o fluxo de informações entre o cliente e o servidor. As informações transferidas seriam a representação da interface do usuário em forma de janelas e eventos de interação gerados pelo usuário. A proposta se distancia de tecnologias de *desktop* remoto tal como RDP, pois pretende que o tráfego seja de fácil entendimento e implementação, contendo atributos de campos ao invés de detalhes com *pixels*.

A especificação aberta de um protocolo, assim como um rascunho de arquitetura, poderá facilitar o desenvolvimento de aplicação *desktop* remotas para vários programadores. Se for adotado poderão surgir clientes para várias plataformas e ambientes, com isso, ampliar as possibilidades. É possível vislumbrar cenários em que uma mesma aplicação pode ser executa em diversos dispositivos, sem a necessidade de ter que se preocupar com cada tipo de *hardware* ou sistema operacional. A intenção é que o desenvolvimento de aplicações para utilizar estes recursos seja bastante simples, sem uso de rotinas proprietárias ou estendendo protocolos existentes.

Um fator que pode apresentar dificuldade para a utilização do protocolo é a falta de clientes instalados nos computadores. Isto pode ser contornado através do desenvolvimento de um *middleware* que rodará sobre uma tecnologia comumente distribuída, que funcionará com um cliente. Um exemplo para isso é um cliente baseado em Flash ou HTML5, que faria a comunicação com o servidor TBAP e teria sua interface rodando em praticamente qualquer navegador de internet.

Alguns aspectos, apesar de não mencionados, são importantes e necessitam um estudo aprofundado no futuro. Um deles é a segurança, que deve ser implementada no protocolo através de criptografia RSA, usando protocolo SSL ou TLS. Para aplicações comerciais também não se pode deixar de citar a importância de relatórios e recursos de impressão. Deverá ser proposta uma extensão do protocolo para geração e visualização de documentos textos, provavelmente utilizando alguma especificação aberta já existente. Necessidades não visuais em comuns para aplicações comerciais podem ser contempladas, tal como uso de *cache* local para busca de dados - a fim de evitar tráfego repetitivo ao servidor.

REFERÊNCIAS

Farley, Jim (1998), **Java distributed computing**, O'Reilly.

Freeman, Eric , Freeman, Elisabeth , Sierra, Kathy , Bates, Bert (2004), **Head First design patterns**, O'Reilly.

Harold, Elliotte Rusty (2004), **Java network programming**, O'Reilly, 3rd edition.

Holzner, Steven (2008), **Ajax: A Beginner's Guide**, McGraw-Hill Prof Med/Tech.

Miller, Frederic P. ; Vandome, Agnes F. , McBrewster, John , (2009), **JSON**, VDM Publishing House Ltd.

Parallax, Inc (2010), **Programming and customizing the multicore propeller microcontroller: the official guide**. p. 379.

Pilgrim, Mark (2010), **HTML5: Up and Running**, O'Reilly & Google Press.

Powell, Thomas A., (2008), **Ajax: the complete reference**, McGraw-Hill Prof Med/Tech.

Scheifler, Robert, Gettys, Jim, (1986), The X Window System, In ACM Transactions on Graphics, v.5, n.2, p.79–109. http://dx.doi.org/10.1145/22949.24053