UNIVERSIDADE FEEVALE

VANIUS ROBERTO BITTENCOURT

PROTOCOLO THINCLIENT PARA APLICAÇÕES COMERCIAIS

Novo Hamburgo
2012

VANIUS ROBERTO BITTENCOURT

PROTOCOLO THINCLIENT PARA APLICAÇÕES COMERCIAIS

Trabalho de Conclusão de Curso
apresentado como requisito parcial
à obtenção do grau de Bacharel em
Ciência da Computação pela
Universidade Feevale

Orientador: Gabriel da Silva Simões

Novo Hamburgo
2012

AGRADECIMENTOS

Gostaria de agradecer a todos os que, de alguma maneira, contribuíram para a realização desse trabalho de conclusão, em especial:

A minha esposa que conviveu comigo diariamente, minha gratidão pelos momentos que compreendeu a minha ausência, de onde teve que ser mãe e pai ao mesmo tempo. Muito obrigado.

RESUMO

Para executar aplicações *desktop* em janelas remotamente no modelo cliente-servidor *thin client* não há um protocolo eficiente. As alternativas atuais se baseiam em transferência de detalhes da tela e geram tráfego excessivo dos eventos gerados pelo usuário. Este trabalho propõe um protocolo para representação visual de aplicações em janelas, de forma que são trafegados atributos dos componentes através de um formato de fácil entendimento e implementação. As soluções atuais foram estudadas e para solucionar problemas em comum é definida uma arquitetura e especificado um protocolo. Foi desenvolvido um protótipo para validar o protocolo e analisar seu desempenho. Através de experimentos é feita comparação do protocolo proposto com outras tecnologias e demonstrados os ganhos alcançados. É obtido um protocolo eficiente que permite ser executado sobre redes com baixa largura de banda e alta latência.

Palavras-chave: Protocolo.Cliente-servidor.Cliente leve.Aplicação.Janelas

ABSTRACT

To run desktop windowing applications remotely on client-server thin client model there is not an efficient protocol. The alternatives are based on current transfer details of the screen and generate excessive traffic of user-generated events. This paper proposes a protocol for visual representation of windowing applications, so that attributes of the components are transmitted through a format easily understood and implemented. Current solutions have been studied and to solve problems in common an architecture is defined and protocol is specified. A prototype was developed to validate the protocol and analyze its performance. Through experiments is made comparing the proposed protocol with other technologies and demonstrated the gains achieved. It obtained an efficient protocol that allows run over networks with low bandwidth and high latency.

Key words: Protocol.Client-server,Thin cliente.Application.Windows

LISTA DE FIGURAS

Figura 1.1 - Funcionamento de aplicações RDP (remote desktop) 17
Figura 1.2 - Funcionamento de aplicações HTML (web) 18
Figura 1.3 - Funcionamento de aplicações Flash (*rich internet application*) 19
Figura 2.1 - Funcionamento do modelo proposto 20
Figura 2.2 - Estrutura de fluxo da arquitetura 22
Figura 2.3 - Exemplo de comando e atributo em formato JSON 25
Figura 2.4 - Tecnologias do formato TBAP 25
Figura 2.5 - Especificação para formato de mensagem 26
Figura 2.6 - Três primitivas para comunicação síncrona 27
Figura 3.1 - Protocolo TLS/SSL 34
Figura 3.2 - Funcionamento do TBAP sobre o protocolo SSL 34
Figura 3.3 - Processos e *buffer* de mensagem 37
Figura 4.1 - Classe Message 42
Figura 4.2 – Estados de aceitação e processamento 44
Figura 4.3 – Fila de mensagens 45
Figura 4.4 – Conversão de fila para uma mensagem única 46
Figura 4.5 – Criação de componente pelo cliente 49
Figura 4.6 – Execução de mensagem recebida pelo cliente 50
Figura 4.7 – Execução de mensagem pelo cliente Swing 51
Figura 4.8 – Diagrama de classes para *framework* cliente 52
Figura 4.9 – Criação de componente na aplicação 54
Figura 4.10 – Padrão *Builder* adotado na API 55
Figura 4.11 – Utilização de *generics* na construção de componentes 55
Figura 4.12 – Declaração de componente utilizando recurso de *generics* 56
Figura 4.13 – Diagrama de classes do *framework* servidor 57
Figura 5.1 – Cenário de comparação entre protocolos 61
Figura 5.2 – Configuração de monitoramento do TrafMeter 62
Figura 5.3 – Aplicação de teste para avaliação 63
Figura 5.4 – Resultado de monitoramento com protocolo VNC 64
Figura 5.5 – Resultado de monitoramento com protocolo RDP 64
Figura 5.6 – Resultado de monitoramento com protocolo TBAP 65

Figura 6.1 – Código de aplicação de negócio 67
Figura 6.2 – Comportamento do cliente em diferentes sistemas operacionais 69
Figura 6.3 – Configuração de limite de velocidade no Trafmeter 70
Figura 6.4 – Desempenho do protocolo sobre conexão lenta 71

LISTA DE TABELAS

Tabela 1.1 – Comparações entre tecnologias *thin client* 19
Tabela 2.1 – Exemplo de fluxo de mensagens síncronas 28
Tabela 3.1 – Fatores críticos 32
Tabela 3.2 – Desempenho de redes 36
Tabela 5.1 – Resultado da comparação entre os protocolos 65
Tabela 6.1 - Exemplo de mensagens em formato JSON 68

LISTA DE ABREVIATURAS E SIGLAS

3G	3rd Generation
API	Application Programming Interface
ATM	Asynchronous Transfer Mode
BSON	Binary JavaScript Object Notation
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
IP	Internet Protocol
JSON	JavaScript Object Notation
LAN	Local Area Network
MAN	Metropolitan Area Network
MIT	Massachusetts Institute of Technology
MVC	Model-view-controller
RDP	Remote Desktop Protocol
RIA	Rich Internet Application
RPC	Remote Procedure Call
SSL	Secure Socket Layer
TBAP	Thin client Busness Application Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
VNC	Virtual Network Connection
W3C	World Wide Web Consortion
WAN	Wide Area Network
WPAN	Wireless Personal Area Network
WLAN	Wireless Local Area Network
WMAN	Wireless Metropolitan Area Network
WWAN	Wireless Wide Area Network
XAML	Extensible Application Markup Language
XML	Extensible Markup Language

SUMÁRIO

INTRODUÇÃO ____ **12**
1 SOLUÇÕES ATUAIS ____ **16**
1.1 Problema em comum ____ 16
1.2 *Desktop* remoto ____ 16
1.3 Aplicações *Web* ____ 17
1.4 Aplicações ricas ____ 18
1.5 Comparações ____ 19
2 PROPOSTA ____ **20**
2.1 Modelo computacional ____ 20
2.2 Arquitetura e protocolo para aplicação *thin client* visual ____ 21
2.3 Execução ____ 23
2.4 Fluxo de comunicação RPC ____ 23
2.5 Formato ____ 24
2.6 Nomenclatura de mensagens ____ 25
2.7 Fluxo de mensagens ____ 26
2.8 Transmissão de eventos ____ 28
2.9 Interface gráfica ____ 29
2.10Negociação ____ 30
3 FATORES CRÍTICOS DE SUCESSO ____ **31**
3.1 Tecnologias relacionadas ____ 32
3.2 Segurança ____ 32
3.3 Estabelecimento de conexão ____ 35
3.4 Conexão confiável ____ 36
3.5 Proteção contra ataques ____ 37
3.6 Capacidade de processamento do servidor e cliente ____ 38
3.7 Correta interpretação do protocolo ____ 38
4 IMPLEMENTAÇÃO DE PROTÓTIPO ____ **40**
4.1 Arquitetura e codificação ____ 40
4.2 Núcleo do protocolo ____ 40
4.3 Implementação cliente ____ 47
4.4 Implementação servidor ____ 52
4.5 Aspectos não codificados ____ 58
5 AVALIAÇÃO DO PROTOCOLO ____ **59**
5.1 Metodologia ____ 59
5.2 Comparação entre tecnologias atuais ____ 59
5.3 Cenário ____ 60
5.4 Execução ____ 61
5.5 Resultado ____ 63
6 EXPERIMENTAÇÕES ____ **67**
6.1 Aplicação de negócio ____ 67
6.2 Fluxo do protocolo ____ 68
6.3 Execução em rede heterogênea ____ 68
6.4 Teste em rede de baixa velocidade ____ 69
6.5 Resultados obtidos ____ 71
CONCLUSÃO ____ **73**
REFERÊNCIAS BIBLIOGRÁFICAS ____ **77**
APÊNDICE A – ESPECIFICAÇÕES DE NOMENCLATURAS ____ **79**

1. Negociação 79
2. Avisos e notificações 79
3. Atributos de componentes visuais 79
4. Eventos 80
APÊNDICE B - CÓDIGOS FONTES DO PROTÓTIPO 81
1. Pacote *Common* 81
2. Pacote *Client* 86
3. Pacote *Server* 96

INTRODUÇÃO

Existem vários meios e modelos de execução de programas, dentre os quais um dos mais utilizados 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, enquanto que o lado do servidor é responsável pelo processamento computacional e armazenamento de dados (ACHARYA, 2006). 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 ou uma extensão ao servidor, formando a aplicação. Neste conceito, o cliente pode executar localmente processamento e regras de negócio, utilizando 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 apenas 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 é tratado como "*thin client"* (SOSINSKY, 2009).

Na atualidade existem dois modelos comuns de funcionamento de aplicativos cliente-servidor. No primeiro modelo, denominado "*desktop"*, a aplicação é executada na estação, ou seja, a aceitação de dados e as regras de negócio são processadas no computador do usuário. Por esse motivo, este modelo também é denominado *fatclient*. O servidor é utilizado principalmente para repositório dos dados (BIDGOLI, 2004, p. 27). Geralmente sua interface gráfica consiste em janelas que são exibidas de forma sobreposta, permitindo manter várias janelas visualizadas simultaneamente. Nesse modelo há componentes visuais elegantes, tais como listas, tabelas, *treeview*, etc. Por ser uma aplicação executada localmente a camada cliente deve estar instalada na estação ou centralizada em sua rede privada. No caso de uma rede de longa distância, há um problema em manter os clientes atualizados. Para isso é necessário ocorrer o *download* do cliente no início da execução – o quê pode ser uma restrição, dependendo do tamanho da aplicação.

No outro modelo, as aplicações são inteiramente executadas do lado do servidor, o cliente apenas exibe a interface. Esta interface é visualizada através páginas HTML. Com isso, qualquer estação com navegador de internet funciona como cliente. O armazenamento e execução das regras de negócio são feitas no servidor. Esse modelo é denominado "aplicação *web*" (DOOLEY, 2011, p. 53). O formato de documento HTML foi inicialmente desenvolvido para representação de textos – nunca o seu objetivo foi representar aplicações.

Por isso, para desenvolver aplicações neste modelo, é necessário uma série de ferramentas e aplicativos adicionais. Sem auxílio deles seu desenvolvimento é complexo (MURUGESAN; DESHPANDE, 2001, p. 5). Por padrão, o HTML não permite visualização em janelas. No servidor as aplicações podem ser desenvolvidas em diversas linguagens, mas a aplicação deve resultar em documentos HTML. Isso praticamente obriga o desenvolvedor *web*, além de dominar a linguagem da aplicação, dominar o formato HTML e prever o seu comportamento em diferentes navegadores. Para navegar entre as telas (páginas) ocorre a recarga da interface. Com objetivo de melhorar esta restrição surgiu a tecnologia Ajax (HOLZNER , 2008, p. 6), porém sua implementação pode ser tornar complexa (POWELL, 2008, p. 475). O protocolo HTTP é utilizado para fazer a transferência do documento HTML até o cliente. Este protocolo tem por característica não armazenar estado de conexão, pois é encerrada quando a transferência é concluída. Sendo assim, no lado do servidor, deve haver mecanismos para recuperar e armazenar os dados da sessão (MACDONALD, 2010, p. 258). O HTTP não possui suporte nativo para fluxo de dados de dois caminhos (ida e volta), com isso aplicações que usam comunicação bidirecional precisam utilizar conexões adicionais, podendo causar sobrecarga no servidor (FETTE; MELNIKOV, 2011). A vantagem deste modelo é que funciona como um *thin client*, sendo que não há preocupação em distribuir e atualizar os clientes. Praticamente todas as estações, de diferentes sistemas operacionais, possuem navegador de internet.

Na década de 80, quando surgiram os primeiros conceitos de telas gráficas, o MIT (*Massachusetts Institute of Technology*) criou um protocolo de comunicação entre terminais semelhante ao VT100, utilizado em terminais caracteres. O protocolo se denominava *System Window X,* ou simplesmente X (SCHEIFLER; GETTYS, 1992). 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 para ser utilizado em meios externos de alta latência, tal como a internet (MEERSMAN; TARI, 2005, p. 782).

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. Esta aplicação é executada no servidor e sua visualização é feita através do cliente, utilizando o *plugin* Adobe Flash, que está instalado em quase todos navegadores de internet (PISA, 2009). 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* (CAPTAINCASA, 2011). Assim como, estas existem outras soluções proprietárias, que aplicaram suas próprias soluções de servidor, cliente e protocolo de comunicação entre eles.

Na internet, muitas tecnologias e protocolos são padronizados, de forma aberta e livre. Para evitar o crescimento de tecnologias proprietárias o W3C está definindo modificações para padrão HTML, agora na versão 5 (PILGRIM, 2010). Está sendo definida característica do protocolo para visualizações ricas, tal como desenhos vetoriais e exibição de vídeo. O principal objetivo do HTML5 é ser uma alternativa ao Flash (HARRIS, 2011, p. 4). O HTML5 possui instruções básicas de desenho, mas não necessariamente para exibir janelas semelhantes às aplicações *desktop*. Para ter esse recurso deve ser desenvolvida uma aplicação que roda sobre o HTML5. A nova versão do HTML não terá recursos prontos específicos para exibição de aplicações, tal como o Adobe Flex possui.

Para contornar o problema de acessar aplicações *desktop* à distância, surgiram alguns protocolos para permitir acessar a tela do computador remotamente, tal como RDP (*Remote Desktop Protocol*) e VNC (*Virtual Network Computing*). Estes protocolos consistem em mapear a área da tela *desktop pixel* a *pixel*, fazendo a transferência da área modificada (CRAFT; BROOMES; KHNASER, 2002, p. 155). Então, se uma janela é movimentada ou minimizada, ocorre a tráfego da área, pois não há armazenamento das janelas em segundo plano. Por representar a tela do usuário, na prática, ocorre a exibição integral da aplicação no servidor, consumindo recursos para cada usuário conectado. Se houver um número elevado de usuários será necessário um servidor bastante robusto (HARWOOD, 2002, p. 159).

Atualmente, são perceptíveis as mudanças das tendências sobre desenvolvimento de aplicações. Percebe-se uma busca pelo resgate dos aspectos centralizados e leves adotados pelos terminais burros (MORIMOTO; KOVACH; ROBERTS, 2003, p. 483). Na internet existem diferentes protocolos de comunicação, mas não há um protocolo difundido que permita a execução de aplicações remotas com aparência em janelas. Igualmente não é encontrado protocolo que realiza o tráfego de atributos de objetos visuais ao invés do tráfego de *pixels*, executada por um cliente-leve independente de plataforma e sistema operacional.

Este trabalho tem por objetivo criar uma especificação de um protocolo para representação visual de aplicações, visando resolver muitos dos problemas citados anteriormente. A partir das possibilidades atuais é possível extrair um modelo e propor uma normalização. Para auxiliar sua aplicação será adotado um formato de padrão aberto e difundido, de melhor compreensão humana. Será possível desenvolver aplicações centralizadas em servidor, sendo que para o usuário será exibida somente a interface visual. Essa interface poderá ser semelhante a aplicações *desktop* convencional, fugindo das limitações de aplicações *web* com HTML. Depois de construídos componentes para interpretar o protocolo, o programador poderá focar seu esforço na regra de negócio da aplicação, e não com processo de execução da interface. Por ser um protocolo aberto, poderão ser construídos clientes para diversos ambientes para funcionar com qualquer aplicação. Da mesma forma do lado do servidor haverá a possibilidade de ser criado um servidor de aplicação que será responsável pelo tráfego do protocolo visual.

1 SOLUÇÕES ATUAIS

1.1 Problema em comum

Em uma rede local (LAN) as estações de trabalho acessam um servidor em comum. Geralmente em LAN as estações são semelhantes, ou seja, fazem parte de uma rede homogênea. A rede local possui um tráfego de alta velocidade e sua estrutura possui baixa complexidade. Neste cenário as aplicações *thin client* podem não trazer grandes benefícios.

Quando há a necessidade desta rede local comunicar com outra rede mais distante – e com isso formar uma rede maior (WAN) - surgem fatores tal como menor velocidade, maior latência e aumento da possibilidade uma rede heterogênea (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 16). Quanto maior o número de estações maior será a possibilidade de haver computadores com poder de processador defasado, além que será maior a necessidade de manutenção de *hardware* e *software*. Então quanto maior a rede maior será a necessidade por aplicações com baixa dependência da plataforma de execução. Neste cenário as aplicações *thin client* trariam grande melhoria, pois funcionariam de forma simplificada e equivalente para um número maior de configurações de estações.

Cada vez mais é comum a necessidade de executar aplicações através de um cliente leve. A internet, por se tratar de uma rede abrangência mundial, aumentou esta demanda significativamente. Hoje não somente computadores se conectam a internet, mas também *smartphones*, *tablets*, *videogames*, tocadores de BluRay, TVs, etc. Muitos destes equipamentos não possuem grande poder computacional (WASINGER, 2007, p. 25), então a execução através de uma camada leve é essencial.

A seguir serão tratadas tecnologias atuais que possuem este objetivo.

1.2 *Desktop* remoto

Uma das soluções atuais é a execução de *desktop* remoto. Neste tipo de tecnologia a interface do usuário é transmitida integralmente *pixel* a *pixel* para o cliente. A aplicação executada do lado servidor pode ser uma aplicação *desktop*, que funciona sobre esse mecanismo sem a necessidade de adaptação. As interações do usuário são transmitidas à aplicação ao usuário como se o usuário estive operando localmente no computador remoto. As operações de *mouse* e teclado são enviadas remotamente. Nesse cenário, apesar do usuário interagir em outro computador, os eventos são gerados no computador remoto.

Existem vários protocolos de *desktop* remoto, entre eles podemos citar o RDP. Este protocolo é baseado no padrão aberto ITU-T T.128 (STEWART, 2010, p. 395). Dentre alguns recursos deste padrão inclui a vídeo conferência, onde um servidor pode transmitir a mesma tela para vários clientes. A Figura 1.1 ilustra o funcionamento das tecnologias de *desktop* remoto.

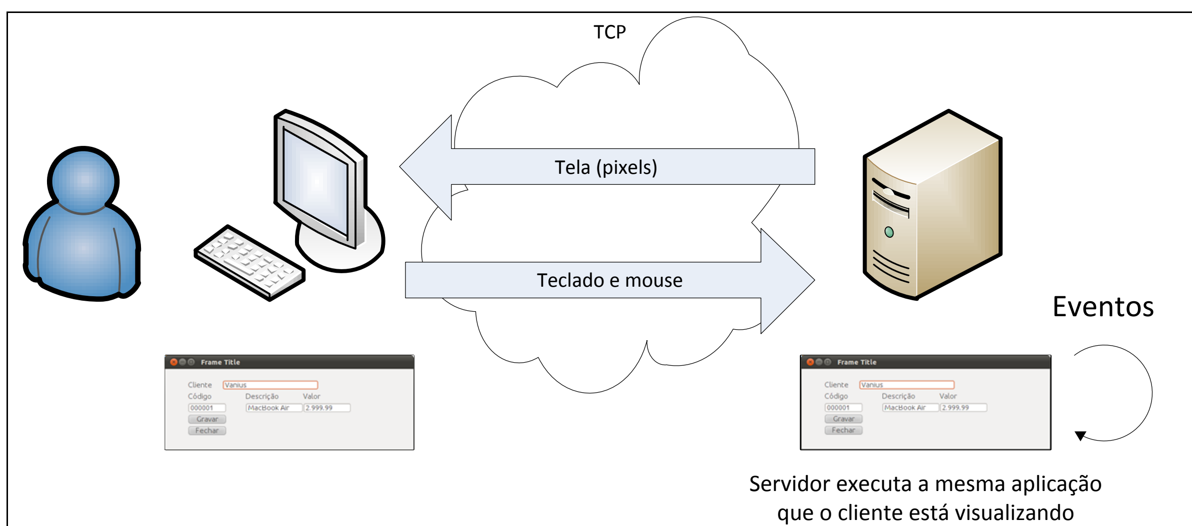

Figura 1.1 - Funcionamento de aplicações RDP (remote desktop)
Fonte: Autor

1.3 Aplicações *Web*

O HTML é o principal formato de aplicações via internet. Apesar de possuir proposta de funcionar em um cliente leve, para contornar restrições do HTML cada vez mais são transmitidas rotinas escritas em JavaScript embutidas dentro HTML. Estas rotinas são executadas pelo navegador, ou seja, há um processamento do lado cliente. Quando termina a transmissão do HTML a conexão é encerrada, ou seja, não há um fluxo contínuo durante a aceitação da tela com o usuário. Para evitar recarga do HTML validações de campos, tratamentos de eventos são processados no cliente. Muitas vezes para contornar esta situação os campos são validados somente ao final do preenchimento, ou seja, somente do *submit* do *form* (página). Na Figura 1.2 é demonstrado o funcionamento das aplicações *Web*.

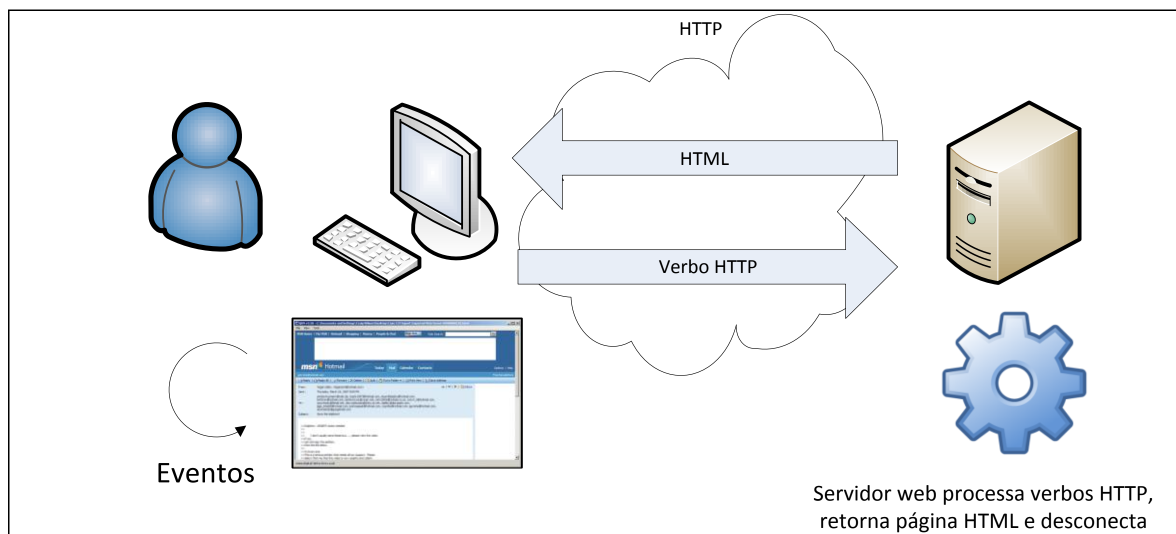

Figura 1.2 - Funcionamento de aplicações HTML (web)
Fonte: Autor

Existem outros formatos semelhantes ao HTML. Um deles é XAML, criado pela Microsoft para ser utilizado no *Silverlight*. Funciona semelhante ao HTML, porém foi criado para representar uma interface gráfica. Outro aspecto parecido ao HTML é que permite ter embutido rotinas do *framework* .Net. O XAML será o formato padrão de desenvolvimento para aplicações Metro para o Windows 8 e para o Windows Phone 7.

1.4 Aplicações ricas

Aplicações de formato RIA (*Rich Internet Application*) são executadas no lado cliente normalmente em navegadores de internet. Podem requisitar e transmitir informações para o servidor. Foram criadas inicialmente para criar recursos gráficos em página HTML, apesar disso permitem criação de sistemas remotos. A Figura 1.3 ilustra o funcionamento de aplicações RIA.

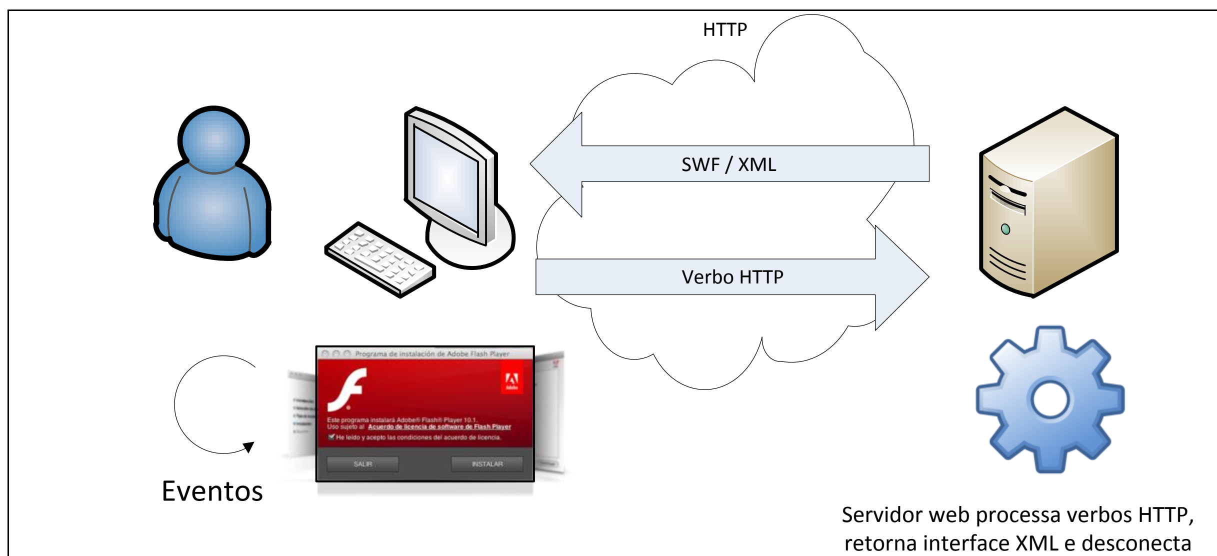

Figura 1.3 - Funcionamento de aplicações Flash (*rich internet application*)
Fonte: Autor

1.5 Comparações

Aplicações *web* e as tecnologias RIA possuem uma formatação para janela de aceitação de dados, porém uma janela de cada vez. Para estas soluções não há um fluxo de eventos que retornam ao servidor. Se for necessário o evento deve ser interceptado do lado do cliente e deve chamar uma função remota.

Soluções de *desktop* remoto transmitem ações de teclados e mouse para o servidor e lá serão gerados e processados os eventos. A Tabela 1.1 exibe um comparativo entre as soluções estudadas.

Tabela 1.1 – Comparações entre tecnologias *thin client*

Tecnologia	Processamento local	Local de processamento do evento	Representação visual
RDP	Não	Remoto	*Pixel map*
VNC	Não	Remoto	*Pixel map*
HTML	JavaScript	Local	HTML
Flash	ActionScript	Local	MXML, XML
Silverlight	.Net e JavaScript	Local	XAML

Fonte: Autor

2 PROPOSTA

Com base no estudo das soluções atuais é possível extrair aspectos positivos para compor um novo modelo de aplicação *thin client*. Tecnologias tal como Silverlight possuem a interface gráfica representada por um formato simplificado, contendo atributos de campos. E outras tecnologias tal como RDP possuem todo processamento no lado de servidor, inclusive o tratamento de eventos.

2.1 Modelo computacional

Apesar da proposta deste trabalho ser um protocolo de comunicação, é fundamental especificar comportamentos e funcionalidades das partes do cliente e servidor.

O cliente deve exibir a GUI (*graphical user interface*) através de meta-dados que representam atributos de janelas e componentes visuais. O cliente não executará processamento específico, somente fará o processamento da interface gráfica e interceptação de ações do usuário na forma de eventos. Validações de aceitação de campos devem ser processadas do lado do servidor. Os eventos serão transmitidos respeitando um modelo com formato específico, e não em formato "cru" tal como posição do *mouse* e código de teclas pressionadas. A Figura 2.1 demonstra o funcionamento do protocolo proposto.

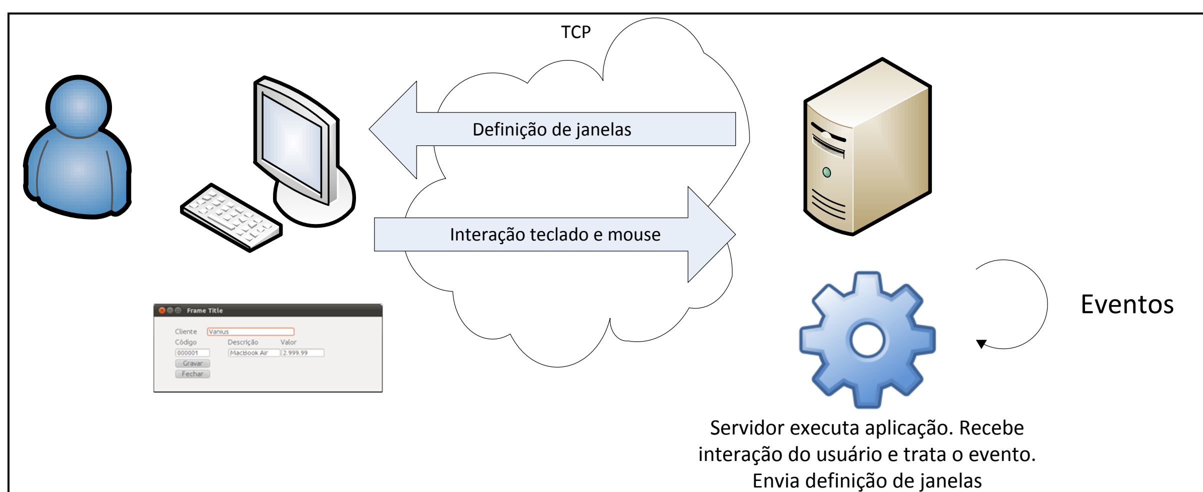

Figura 2.1 - Funcionamento do modelo proposto
Fonte: Autor

O servidor deverá permitir conexão de vários clientes e permitir executar diferentes aplicações. A transmissão será feita ponto-a-ponto, não prevendo a transmissão *multicast*, tal

como vídeo conferência. Diferente de aplicações ricas e páginas HTML não necessitará processamento do lado do cliente para validações ou tratamento de eventos.

2.2 Arquitetura e protocolo para aplicação *thin client* visual

O protocolo proposto será denominado TBAP – *Thin client 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.

No modelo OSI este protocolo seria enquadrado como um protocolo da camada de aplicação (TANENBAUM; STEEN, 2006, p. 123), então 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 2.2.

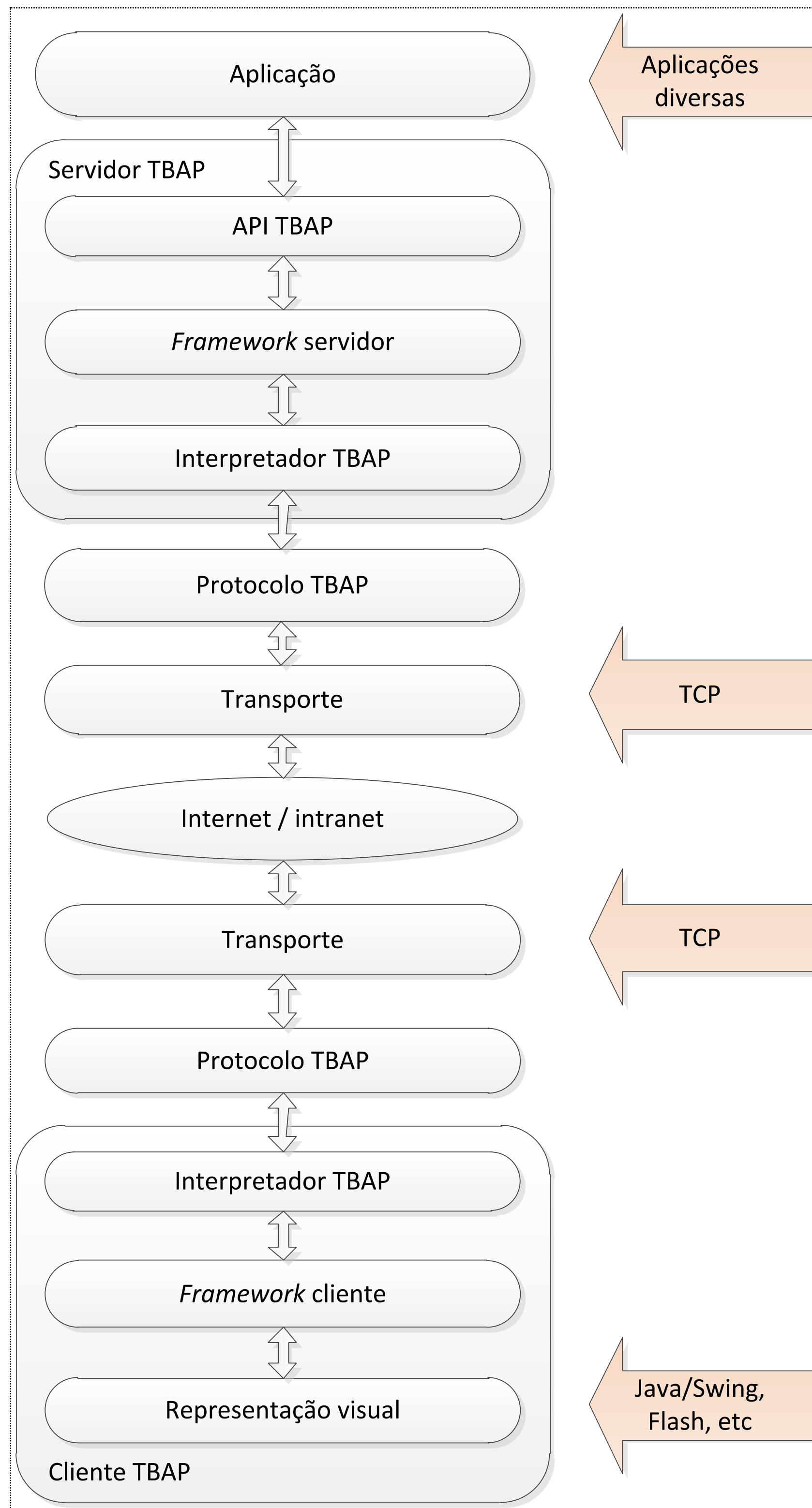

Figura 2.2 - Estrutura de fluxo da arquitetura
Fonte: Autor

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.

2.3 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.

2.4 Fluxo de comunicação RPC

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 ocorrerá em forma de chamadas, tal como *Remote Procedure Call* (RPC) (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 186). 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 (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011 p. 107). 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.

2.5 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*) (CROCKFORD, 2006), que em comparação ao XML se demonstra mais simplificado (MILLER; VANDOME; MCBREWSTER, 2009). Cada mensagem do protocolo pode conter vários atributos, conforme Figura 2.3.

```
{ "mensageName":"propertyChange",
   "array" : [{ "property1" : "1" },
              { "property2" : "2" }    ] }
```

Figura 2.3 - Exemplo de comando e atributo em formato JSON
Fonte: Autor

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. As mensagens são enviadas de modo assíncrono, com isso é assumida uma conexão orientada a mensagens (TANENBAUM; STEEN, 2006, p. 159). 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. As tecnologias envolvidas pelo formato TBAP são ilustradas pela Figura 2.4.

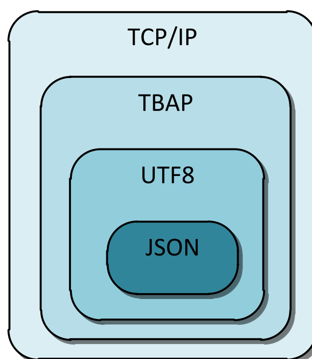

Figura 2.4 - Tecnologias do formato TBAP
Fonte: Autor

2.6 Nomenclatura de mensagens

As mensagens devem ser textos contendo informações divididas entre em pares chaves-valor, representadas em formato JSON. As chaves das informações serão a descrição do objetivo da mensagem e descrição dos parâmetros. Estas descrições textuais simbolizam o

funcionamento do protocolo e possuem conteúdo constante. As formas das descrições devem respeitar as regras:

- Palavras em inglês
- Letras em maiúsculo
- Mais de uma palavra devem ser delimitadas por *underscore* "_"
- Não utilizar siglas
- Descrições de eventos devem ter o prefixo "ON_"

Os valores devem ser textos respeitando a codificação UTF8.

Cada mensagem deve ter um nome definido pela chave "MESSAGE_NAME". Cada mensagem pode ter nenhum ou vários pares chaves e valor de parâmetros. No caso de mensagens síncronas a mensagem pode ser respondida com nenhum ou vários pares chave-valor de retorno. A Figura 2.5 ilustra o formato que a mensagem deve ter.

```
{"MESSAGE_NAME": "<Nome da mensagem>";
<"NOME_PARAMETRO_1">: "<Valor parâmetro 1>";...
<"NOME_PARAMETRO_N">: "<Valor parâmetro N>;
<"NOME_RETORNO_1">: "<Valor retorno 1>";...
<"NOME_ RETORNO _N">: "<Valor retorno N>}
```

Figura 2.5 - Especificação para formato de mensagem
Fonte: Autor

As especificações da nomenclatura das mensagens, parâmetros e retornos estão descritas na seção "APÊNDICE A".

2.7 Fluxo de mensagens

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 (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 187). Conforme a Figura 2.6 deve obedecer as três primitivas: executar operação, receber requisição e enviar resposta. A mensagem de retorno pode conter diferentes campos de retorno.

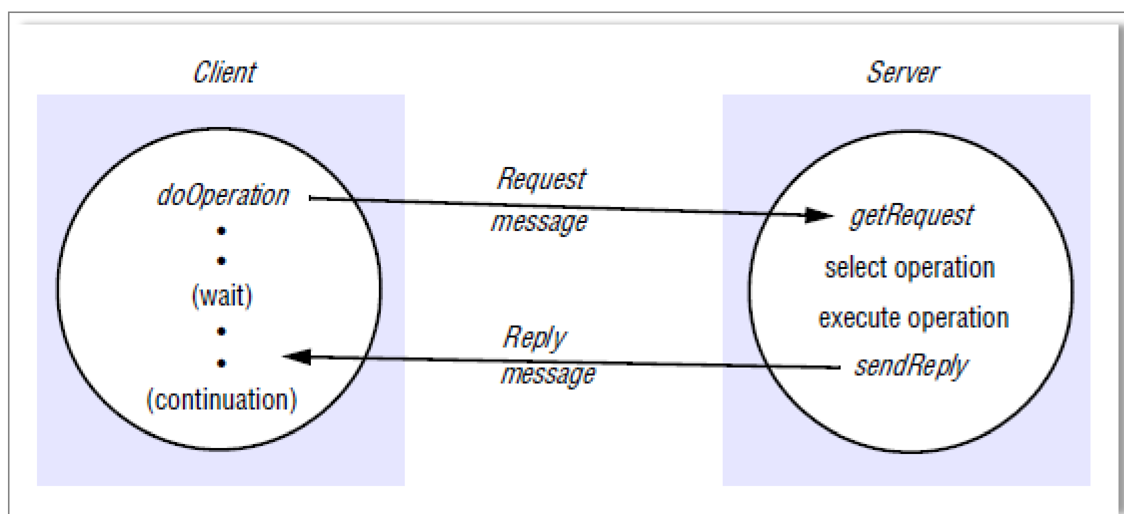

Figura 2.6 - Três primitivas para comunicação síncrona
Fonte: COULOURIS; DOLLIMORE; KINDBERG; BLAIR (2011) , p. 187

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:

Tabela 2.1 – Exemplo de fluxo de mensagens síncronas

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]

Fonte: Autor

2.8 Transmissão de eventos

Ações do usuário na interface geram mensagens ao servidor, denominados “eventos”. Os eventos possuem o nome da ação e o identificador do componente. O evento também pode conter a valor do campo alterado pelo usuário, além de parâmetros específicos para cada tipo de ação.

Tipos de eventos:

- Tecla pressionada
- Clique do mouse
- Troca de foco

A maneira que será tratada os eventos gerados pelo usuário é crucial para o protocolo ser eficiente. Sempre que uma mensagem é enviada existe o tempo para chegar ao outro lado e ser processada. Este tempo é denominado latência (COULOURIS; DOLLIMORE;

KINDBERG; BLAIR, 2011, p. 83). Em aplicações *desktop*, puramente executadas no computador local, o tempo de envio de comandos de eventos são desprezíveis, pois o processamento é feito localmente.

A representação visual da aplicação é definida no servidor, porém sua exibição e acessibilidade são feita pelo cliente. Ou seja, as ações do usuário causam o efeito no cliente se necessário posteriormente é transmitido evento ao servidor. Isto significa, por exemplo, o que usuário digita é visualizado imediatamente no cliente, assim como quando executa o troco de foco. Estas operações não necessitam de comunicação ao servidor, diferentemente do que ocorre em tecnologias tal como RDP e VNC, pois se o cliente digita é transmitida a tecla ao servidor, o servidor insere a tecla, atualiza a tela e a retransmite ao cliente.

No contexto do protocolo proposto, os eventos devem ser trafegados até outro computador, então o tempo de envio deve ser considerado para definir os tipos de eventos que o protocolo irá utilizar. Eventos muito repetitivos, que geram muitas transmissões em poucos segundos, devem ser evitados.

Exemplos de eventos que devem ser evitados são: movimentação do mouse sobre componentes (sem cliques), redimensionamento de janela e digitação em caixa de texto.

Serão poucos os eventos de teclados transmitidos ao servidor. O texto digitado em campos será transmitido como mudança do atributo de texto do componente, e não a cada tecla pressionada. O texto modificado deve ser transmitido quando ocorrer algum outro evento ao servidor. Por exemplo: o usuário preenche seu nome de *login* e pressiona botão de confirmação. Quando o botão for pressionado então deve ser transmitido evento de alteração ao campo, contendo todo o texto do campo. O mesmo deve ocorrer se o usuário retirar o foco do campo. As teclas que terão seus eventos transmitidos imediatamente serão as teclas Enter, Esc e de funções F1 até F12, além de combinações que contenham CTRL ou ALT.

Para evitar fluxo de eventos desnecessários, a aplicação do lado do servidor pode definir quais eventos devem ser gerados para cada componente pelo cliente. Essa indicação pode ser feita a qualquer momento, inclusive no instante da criação do componente.

2.9 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”.

2.10 Negociação

O processo de início execução da aplicação remota é denominado “negociação”. Nessa etapa mensagens podem ser trocadas entre cliente e servidor de forma que possam ajustar a execução ou impedi-la.

Depois de criada conexão o cliente deve passar para o servidor o nome da aplicação e parâmetros de execuções. Como etapa consecutiva o cliente e servidor podem requisitar informações sobre configurações de execução, tal como a versão do cliente, resolução da tela, sistema operacional e plataforma de execução.

No processo de negociação podem ocorrer requisições de recursos disponíveis entre cliente e servidor. Com isso podem ser implementados novos recursos e os lados podem funcionar de maneira alternativa para contemplar a execução em modo de compatibilidade. Com isso pode-se tornar o protocolo flexível para ser estendido.

Após receber o nome da aplicação e parâmetros o servidor pode fazer as requisições para o cliente. Ao concluir todas as requisições o servidor deve enviar mensagem de conclusão para o cliente. Então nesse momento o cliente pode fazer as suas requisições ao servidor. Quando o cliente concluir então deve enviar mensagem ao servidor. Nesse momento o servidor pode iniciar a execução da aplicação e por consequência a aplicação aparecerá para o usuário no lado do cliente.

3 FATORES CRÍTICOS DE SUCESSO

O protocolo tem por objetivo transferir uma representação visual de um aplicativo de um computador a outro. Esta representação visual é um fluxo de informação que se transporta através diversas tecnologias, portanto o protocolo funciona interagindo com outros meios computacionais, fazendo parte de um sistema maior. Estas tecnologias se relacionam de forma que ocorre uma forte dependência, pois uma tecnologia implementa e abstrai uma camada inferior (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 92).

Os fatores críticos das tecnologias envolvidas são enumerados para compreensão dos riscos possíveis ao protocolo. Com isso pode-se prever determinados comportamentos para evitar ou mitigar problemas, garantindo o seu correto funcionamento. Alguns fatores devem ter seu comportamento exercido a nível mínimo esperado. Estas condições serão estabelecidas como pré-requisitos para o protocolo fornecer um resultado esperado.

O correto funcionamento deve ser dado pelo cumprimento dos seguintes itens (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 49):

- Conectividade e continuidade
- Largura de banda
- Latência

A possibilidade de um ponto alcançar outro ponto, sem interferência de qualquer barreira, é objetivo da conectividade. Continuidade é a intenção em manter o fluxo de informação entre estes pontos, de forma ininterrupta e sem interferência. Os dados transmitidos devem ser entregues em tempo hábil para não afetar experiência do usuário.

A latência - o tempo que a informação demora em se deslocar até o destino - deve ser mínima. Porém, além deste tempo, os dados como um todo devem se deslocar em velocidade sufuciente para não comprometer o tempo total da transmissão. A largura de banda dita a quantidade de bytes que podem ser trafegados em uma fração de tempo.

O fato de o protocolo ser o meio para um sistema distribuído implica que a usabilidade do sistema deve ser transparente ao usuário, ou seja, é conveniente não ser perceptível o acoplamento dos componentes envolvidos (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 49).

3.1 Tecnologias relacionadas

Cada camada possui seus próprios fatores críticos para o correto funcionamento, porém como são inter-relacionados os fatores das tecnologias envolvidas devem ser considerados para o sucesso do trabalho, tal como indicado na Tabela 3.1.

Tabela 3.1 – Fatores críticos

Camada	Fator crítico
Aplicação executada no servidor	Correto desenvolvimento da aplicação de modo a não gerar excesso de processo, de forma que prejudique o desempenho do servidor.
Servidor TBAP	Proteção contra ataques e invasões
Servidor e cliente protocolo TBPA	Correta interpretação do protocolo TBAP e coerência de versões. Permitir uso de porta TCP alternativa Segurança dos dados trafegados
Interface gráfica do cliente TBPA	Cliente deve permitir correta visualização, tal ter como resolução apropriada, número de cores, etc
Plataforma de execução	Deve estar instalado corretamente para permitir servidor ou cliente ser executado com desempenho adequado
Sistema operacional	Sistema operacional deve estar opto para executar o servidor ou cliente. Cliente possui requisitos mínimos de interface
Hardware de servidor e cliente	Deve haver capacidade de processamento e armazenamento para executar a aplicação. Disponibilidade e estabilidade
Adaptador de rede, switches e outros dispositivos de rede.	Capacidade para transferir os dados sem interrupção e com qualidade mínima
Roteadores e firewall	A configuração de acesso não deve bloquear a utilização do protocolo

Fonte: Autor

3.2 Segurança

A especificação do protocolo TBAP idealiza o uso de mensagens texto de fácil entendimento humano. O principal objetivo desta característica é auxiliar o desenvolvimento de aplicações cliente e servidor. As mensagens representam dados da visualização de uma

aplicação; potencialmente conterão informações pertinentes à regra de negócio da aplicação assim como dados de interesse exclusivo aos usuários finais.

Um aspecto que deve ser levado em consideração é o da segurança dos dados transmitidos. Depois de estabelecida a conexão entre os pontos - e garantido o fluxo de troca de informações – é importante assegurar que a informação emitida seja lida e interpretada somente pelo destinatário (TANENBAUN; STEEN, 2006, p. 377).

Por haver diversas tecnologias envolvidas nem todas as camadas implementam controle de segurança. O protocolo TBAP, por ser um protocolo de aplicação, funciona sobre as demais camadas de enlace e físico. As camadas inferiores funcionam de forma transparente e indiferente ao funcionando de protocolos superiores, ou seja, não é possível um protocolo de aplicação interferir no funcionando de protocolos inferiores. As camadas de enlace físico, tal como Ethernet, não prevê obrigatoriamente este tipo de necessidade (OSTROVSKY, 2008, p. 21). Não é possível o protocolo de aplicação certificar que níveis inferiores sejam seguros.

A criptografia – mecanismo de segurança para troca de informações eletrônicas – não é utilizada em todas as situações, pois para ocorrer o “em baralhamento” da mensagem é necessário passar por uma transformação através um determinado algoritmo (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 74). Este requisito de processamento computacional deve ser levado em consideração (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 501), pois pode representar uma restrição. A criptografia pode afetar a velocidade do processamento das mensagens.

A camada TCP, utilizada pelo protocolo TBAP, não fornece qualquer nível segurança ou criptografia (GENCO, 2008, p. 182). Os dados trafegados não são alterados e podem ser acessados da mesma forma que foram transmitidos. Através de ferramentas de monitoração de dados, tal como *sniffer*, as informações podem ser interceptadas e lidas (RACHGHARE, 2009, p. 235).

Para utilizar o TCP de modo seguro é necessário haver uma camada superior que o implementa, conforme ilustrado na Figura 3.1. Para permitir uma comunicação segura o protocolo TBAP prevê seu funcionamento sobre o protocolo criptográfico o TLS/SSL. O protocolo TLS funciona sobre o TCP e permite que outros serviços o utilizem (TANENBAUM; STEEN, 2006, p. 548).

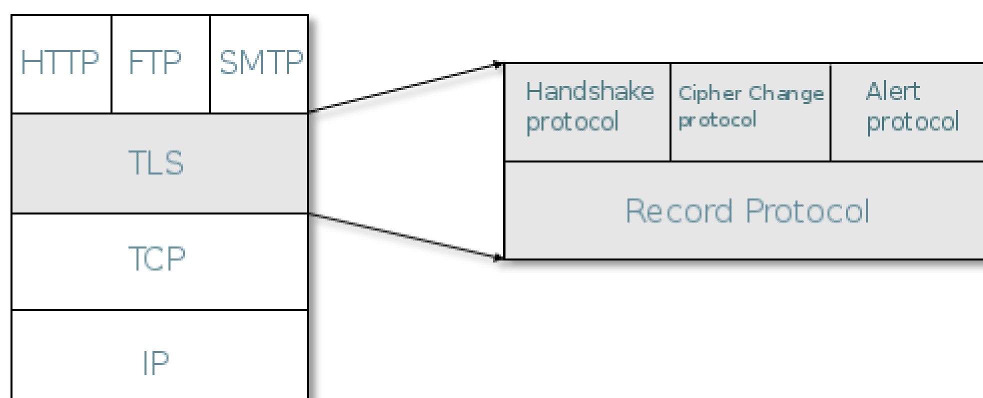

Figura 3.1 - Protocolo TLS/SSL
Fonte: http://pt.wikipedia.org/wiki/Transport_Layer_Security

Em suma, o protocolo opera diretamente sobre o TCP, porém permite o uso do protocolo SSL como um mecanismo opcional de segurança. Nesta situação o TBAP utiliza o SSL como protocolo de transporte, conforme ilustrado pela Figura 3.2.

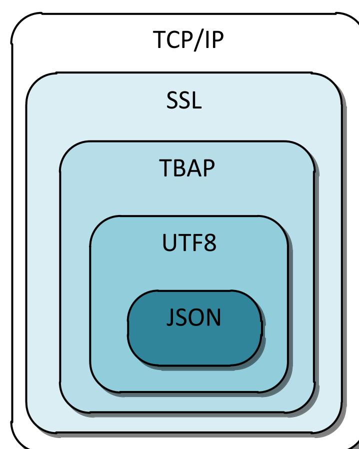

Figura 3.2 - Funcionamento do TBAP sobre o protocolo SSL
Fonte: autor

Esta característica não é obrigatória, pois se pode optar em utilizar o protocolo TBAP sobre uma estrutura onde camadas inferiores são protegidas, neste caso pode não ser necessário onerar um processamento de segurança adicional. Um exemplo desta situação é a utilização de uma rede privada virtual (VPN), que pode ser protegida através de autenticação (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 128).

3.3 Estabelecimento de conexão

A conectividade entre o servidor e cliente - dentre vários fatores - depende da disponibilidade e permissão de acesso sobre os endereços IPs e portas envolvidas. No caminho percorrido pelo fluxo de dados podem estar envolvidos diversos periféricos, cada um sujeito a algum controle de bloqueio. O uso de serviços de *firewall* pode ser aplicado em diversos níveis, muitas vezes pode ser difícil de certificar que o fluxo através de uma dada porta irá ocorrer de forma correta (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 125).

O protocolo TBAP utiliza a porta TCP padrão 1379, com isso é conveniente observar que seu uso não é previsto nos atuais ambientes de rede. Possivelmente esta porta estará sujeito a algum tipo de bloqueio, pois uma das estratégicas de proteção é liberar somente as portas utilizadas por algum serviço do local. Devido a esse aspecto é permitido o protocolo TBAP funcionar através de alguma porta alternativa, mesmo que esta porta seja registrada para outro protocolo ou serviço.

Em situações que pode haver bloqueios de portas é recomendável o servidor abrir uma porta com um número de algum serviço registrado conhecido, tal como a porta 80, utilizado por servidor de páginas *web* HTTP (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 393). Por ser um serviço de uso comum normalmente o acesso a páginas de internet é permitido (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 483). Este tipo de alternativa é utilizado por outros serviços, tal como o protocolo de voz por IP Skype, onde possui a porta padrão 443, porém o cliente tenta conectar através de portas alternativas, inclusive a porta 80 (GOUGH, 2005, p. 157).

O serviço de TBAP pode funcionar de forma correta utilizando a porta 80, porém deve-se atentar que neste caso o mesmo computador que executar o serviço de servidor do protocolo TBAP não poderá executar outro serviço de servidor HTTP na porta 80. O servidor deve estar preparado para receber requisições indevidas de navegadores de internet (clientes HTTP), onde esta situação deve ser detectada e uma resposta padrão de servidor *web* indisponível no formato HTTP deve ser enviada como retorno.

3.4 Conexão confiável

Para a comunicação ser confiável cada mensagem enviada deve ser corretamente entregue e a mesma mensagem não pode sofrer modificação (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 71).

Um dos benefícios do protocolo TCP é que possui controle de detecção de erros (*checksum*) e ordenação de entrega de pacotes (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 155). Graças a estas características o TCP garante o recebimento dos dados – caso a comunicação física não for interrompida. Por funcionar sobre o TCP o protocolo TBAP não tem a necessidade de aplicar controle de verificação de *checksum*.

O protocolo proposto tem um dos objetivos ser eficiente, no sentido ter baixa necessidade de consumo de recurso de rede. Esta característica permitirá funcionar em ambientes limitados. O TBAP é recomendado para operar sobre rede que oferecem largura de banda mínima de 0,05 Mbps (ou 50 Kbps) e latência máxima de 500 milissegundos. Segundo a tabela de desempenho de rede do autor (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 86), Tabela 3.2, estas características são adequadas para ser executada sobre as principais redes existentes.

Tabela 3.2 – Desempenho de redes

Rede	Exemplo	Alcance	Largura (Mbps)	Lantência (ms)
LAN	Ethernet	1–2 Kms	10–10.000	1–10
WAN	IP routing	worldwide	0,010–600	100–500
MAN	ATM	2–50 Kms	1–600	10
Internetwork	Internet	worldwide	0,5–600	100–500
WPAN	Bluetooth	10–30m	0,5–2	5–20
WLAN	WiFi	0.15–1.5 Km	11–108	5–20
WMAN	WiMAX	5–50 Km	1,5–20	5–20
WWAN	3G	1–-5 Km	0,010-2	100–500

Fonte: COULOURIS; DOLLIMORE; KINDBERG; BLAIR (2011)

O TBAP permite uma latência alta, pois a atualização de tela é feita pelo cliente – para boa parte das operações do usuário não há a necessidade de transmitir ao servidor e aguardar retorno para atualizar a visualização.

Porém, assim como todos meios de comunicação, pode ocorrer interrupção ou desempenho ser afetado (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 21).

Por diversos fatores a transmissão TCP pode ser interrompida. Neste caso é recomendado haver um mecanismo de reconexão em um nível superior, continuando a transmissão através de uma nova conexão TCP (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 60).

As mensagens enviadas devem permanecer em um *buffer* enquanto não houver certeza que a mesma chegou ao destino (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 68). Deve existir um registro do controle de fluxo que represente as mensagens enviadas e recebidas. Cada mensagem possui um registro único, tal como um identificar incrementado. Conforme a Figura 3.3, cada canal de comunicação possui *buffer* para saída e entrada. O destinatário indica o registro da mensagem recebida, neste momento o remetente pode retirar esta mensagem do *buffer*. No momento da reconexão servidor e cliente trocam os registros da última mensagem processada. Se alguma mensagem não foi corretamente entregue então neste momento há a retransmissão da mensagem.

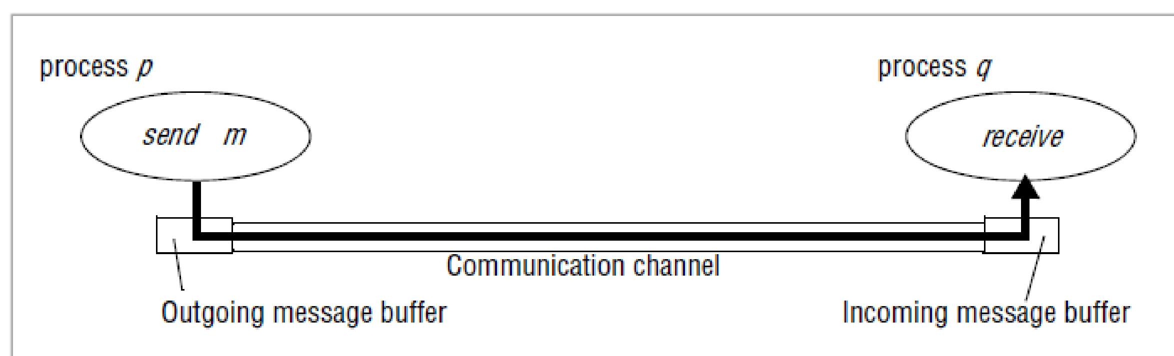

Figura 3.3 - Processos e *buffer* de mensagem
Fonte: COULOURIS; DOLLIMORE; KINDBERG; BLAIR (2011)

3.5 Proteção contra ataques

Todo serviço que funciona como um servidor mantém aberta uma porta definida para realizar a comunicação com diversos clientes. Esta porta aberta deve aceitar requisições remotas normalmente prevendo ser uma tentativa de conexão de cliente, porém este caminho pode estar com visibilidade pública e nem sempre pode se garantir que serão realizadas conexões regulares. O aplicativo servidor do protocolo TBAP deve prever a possibilidade de ocorrer conexões com intenções de prejudicar o seu correto funcionamento.

Uma das tácticas para causar dano a um servidor é enviar muitas requisições (TANENBAUM; STEEN, 2006, p. 427), onde um determinado computador envia diversas

mensagens de forma que causa uma sobre carga. O servidor deve estar preparado para verificar se está ocorrendo sucessivas trocas de mensagens desnecessárias e ou de formato inválido. Caso contrário o servidor pode ocupar tempo demasiado para responder estas mensagens indevidas, deixando de operar para os clientes válidos. Primeiramente o servidor deve registrar o endereço de IP de cada cliente, com isso se for detectado um abuso de mensagens errôneas então deve desconectar e o endereço deve ser considerado como bloqueado, ignorando futuras conexões.

3.6 Capacidade de processamento do servidor e cliente

Deve haver recursos de *hardware* e *software* para servidor e cliente operar integralmente de forma que não comprometa a interpretação do protocolo e execução da aplicação. O sistema operacional e a plataforma de execução devem estar preparados para carga de execução esperada. Os recursos do servidor devem possuir capacidade para executar determinadas aplicações, prevendo o número nominal de conexões simultâneas assim como as necessidades de memória e processamento da aplicação. O cliente deve permitir exibir a aplicação corretamente, para isso o cliente deve ser executado em ambiente que a interface gráfica esteja configurada de forma adequada, incluindo resolução, número de cores, layout (retrato ou paisagem).

3.7 Correta interpretação do protocolo

Entre os computadores que se comunicam - além da correta comunicação física e do protocolo de transporte TCP - deve ocorrer comunicação coerente respeitando as regras do protocolo TBAP. Cada computador que se comunica pode possuir localmente a sua lógica de interpretar o protocolo, porém devem funcionar de forma equivalente entre os dois lados. Entre outras palavras, as duas pontas devem compreender a mesma especificação lógica e o resultado deve ser o mesmo do esperado pelo outro lado. Deve ser respeitada a especificação do formato JSON e UTF8. Os nomes de mensagens, assim como seus atributos, devem estar previstas como palavras reservadas.

Após a elaboração da especificação das mensagens do protocolo, deve-se evitar alterá-la. Se for alterada a especificação não se pode garantir que todos os computadores estarão atualizados. Um caso de risco é um dos computadores não compreender – ou de forma errônea – alguma mensagem, causando uma situação de incompatibilidade. Para isso o protocolo deve prever a informação de versão atual e de compatibilidade entre versões.

Ambos os lados devem informar a versão do TBAP que interpretam. Também deve ser comunicado o seu nome e versão do seu *build*. Isto permite, por exemplo, que um cliente se adapte a um tipo de servidor.

4 IMPLEMENTAÇÃO DE PROTÓTIPO

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 (HAROLD, 2004). A interface gráfica utilizada é a Swing. Para manipular objetos JSON foi utilizada a biblioteca JSON-lib.

4.1 Arquitetura e codificação

Foram criadas interfaces para definir padrões de desenvolvimento para o servidor e cliente. As classes de manipulação do protocolo interagem com estas interfaces. Com isso obtém-se flexibilidade para alterar e estender o código (FREEMAN; FREEMAN; SIERRA, 2004).

As classes criadas foram agrupadas em três pacotes, que podem gerar bibliotecas distintas: *server*, *client* e *common*. As bibliotecas *server* e *client* 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. As especificações do protocolo, tal como nome de mensagens, são definidas em uma classe para esse fim. As mensagens são manipuladas por uma classe que abstrai a utilização do formato JSOSN. 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/

4.2 Núcleo do protocolo

O protocolo TBAP utiliza o TCP para transferir suas mensagens, que são representadas por textos no formato JSON, entretanto no protótipo construído a sintaxe e

formato do JSON são abstraídos através do uso de objetos da API. Com isso os desenvolvedores não necessitariam compreender o funcionamento do JSON. Para fornecer uma programação de alto nível foi criada a classe *Message*, que representa uma mensagem ou um conjunto de mensagens – conforme código fonte ilustrado pela Figura 4.1.

Cada mensagem pode ter várias propriedades, por isso seguem o padrão par chave-valor, ou seja, cada nome de propriedade é único e cada nome possui um valor associado. Por ter esta característica a classe *Message* utiliza objetos JSON de forma natural. Todas as mensagens têm ao menos a propriedade *name* em comum, que se comporta como um identificador dentre as mensagens.

As especificações de palavras reservadas do protocolo TBAP são definidas como constantes públicas da classe final *Protocol*. São declarados os nomes de propriedades das mensagens assim como valores de propriedades que possuem algum significado.

```
1.  public final class Message {
2.
3.      private JSONObject jsonObj;
4.
5.      public Message() {
6.          setFromString( "" );
7.      }
8.
9.      public Message( String name ) {
10.         setFromString( "" ).setName( name );
11.     }
12.
13.     public Message setFromString( String fullString ) {
14.         try {
15.             if ( fullString.isEmpty() )
16.                 jsonObj = new JSONObject();
17.             else
18.                 jsonObj = new JSONObject( fullString );
19.         } catch ( JSONException ex ) {
20.             Logger.getLogger( Message.class.getName() ).log( Level.SEVERE, null, ex
    );
21.         }
22.         return this;
23.     }
24.
25.     public String getValueByName( String name ) {
26.         try {
27.             return jsonObj.getString( name );
28.         } catch ( JSONException ex ) {
29.             Logger.getLogger( Message.class.getName() ).log( Level.SEVERE, null, ex
    );
30.         }
31.         return "";
32.     }
33.
34.     public Message setObjectByName( String name, Object value ) {
35.         try {
36.             jsonObj.put( name, value );
37.         } catch ( JSONException ex ) {
38.             Logger.getLogger( Message.class.getName() ).log( Level.SEVERE, null, ex
    );
39.         }
40.         return this;
41.     }
42.
43.     public Message setValueByName( String name, String value ) {
44.         return setObjectByName( name, value );
45.     }
46.
47.     @Override
48.     public String toString() {
49.         return jsonObj.toString();
50.     }
51.     ...
```

Figura 4.1 - Classe Message
Fonte: autor

O envio e recebimento de objetos *Message* são realizados pelo objeto *Communication*, que faz o uso do objeto Java Socket que realiza comunicação TCP. O Socket realiza transferência de texto *String*, então a aplicação deve transformar sua estrutura de dados

para sequencia de bytes (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 158). Devido a este fato o objeto *Communication* faz a transformação de *String* para objeto *Message* e vice-e-versa. Para essa processo de serialização e desserialização são utilizados os métodos *toString* e *setFromString* da classe *Message*. Esses métodos permitem recuperar e transmitir integralmente todos os atributos de uma mensagem.

O objeto *Communication* possui método para enviar e receber mensagens. O exato momento de recebimento de texto é indeterminado, como pode ocorrer a qualquer instante então deve existir uma rotina em paralelo sempre disponível para recepção. Para isso é criada uma *thread* para escuta de tráfego TCP. Para o tratamento de evento de mensagem recebida é definida a interface *ICommunicationListener*. As camadas de servidor e cliente implementam esta interface de forma que são notificadas através chamadas reversas (*CallBack*) disparado pelo objeto Communication. Este comportamento é definido como o padrão de projeto *Observer* (FREEMAN; FREEMAN; SIERRA, 2004). Sempre que uma mensagem é recebida é iniciada uma nova *thread* para execução da interface *ICommunicationListener*. Com isso é garantida a recepção de novas mensagens mesmo se o tratamento de recepção anterior não for concluído (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 148).

O envio de mensagem é realizado pelo método *sendMessage* do objeto *Communication*. Este método envia mensagem de forma assíncrona, ou seja, a fluxo da aplicação segue independente da entrega. Contudo é interessante, em muitas circunstâncias, haver o envio de mensagem síncrona, que pode retornar informações do computador remoto. Para isso é utilizado o método *inquireMessage*. O fluxo de execução somente segue quando houver o retorna da mensagem. Para cada mensagem síncrona enviada são gerados e associados um número e um objeto identificador de controle, de forma que este objeto fica em estado de aguardo, funcionando como semáforo para a *thread* em execução.

As mensagens síncronas possuem o atributo *INQUIRING_ID* que contém o número identificador para a *thread* associada. É criada uma lista com os números e objetos de controles para as mensagens assíncronas, de modo que através do número possa ser recuperado o objeto de controle. Para a mensagem recebida é verificado se a mensagem possui o atributo *INQUIRING_ID*, neste caso é buscado a mensagem síncrona que a disparou e a mensagem recebida é armazenada uma lista a ser utilizado pelo método *inquireMessage*. Após o tratamento de recepção pela interface *ICommunicationListener* é retornado o objeto semáforo da mensagem e este tem estado alterado para executar novamente. O método *inquireMessage* segue o fluxo e retorna a resposta associada a mensagem enviada. Esta

sequencia pode ocorrer de forma sequencial e concomitante, de forma que a lógica geral da aplicação possa ser empilha e desempilhada. Isso é possível pelo fato do tratamento de cada mensagem ser processada em thread separada e haver controle de listas associativas de mensagens e respostas.

A aplicação de negócio, executado do lado do servidor, geralmente terá processamento disparado por alguma interação do usuário. Após a inicialização da aplicação o fluxo de execução da aplicação irá aguardar eventos disparados pela interface do cliente. Neste contexto é assumido que aplicação possui dois estados: em processamento e em aceitação com usuário - conforme ilustrado pela Figura 4.2. Consecutivamente após qualquer processamento da aplicação o usuário irá interagir, portanto as alterações de interface realizadas pelo processamento não necessitam ser imediatamente transmitidas - podem ser transmitidas posteriormente, porém ainda antes da interação do usuário.

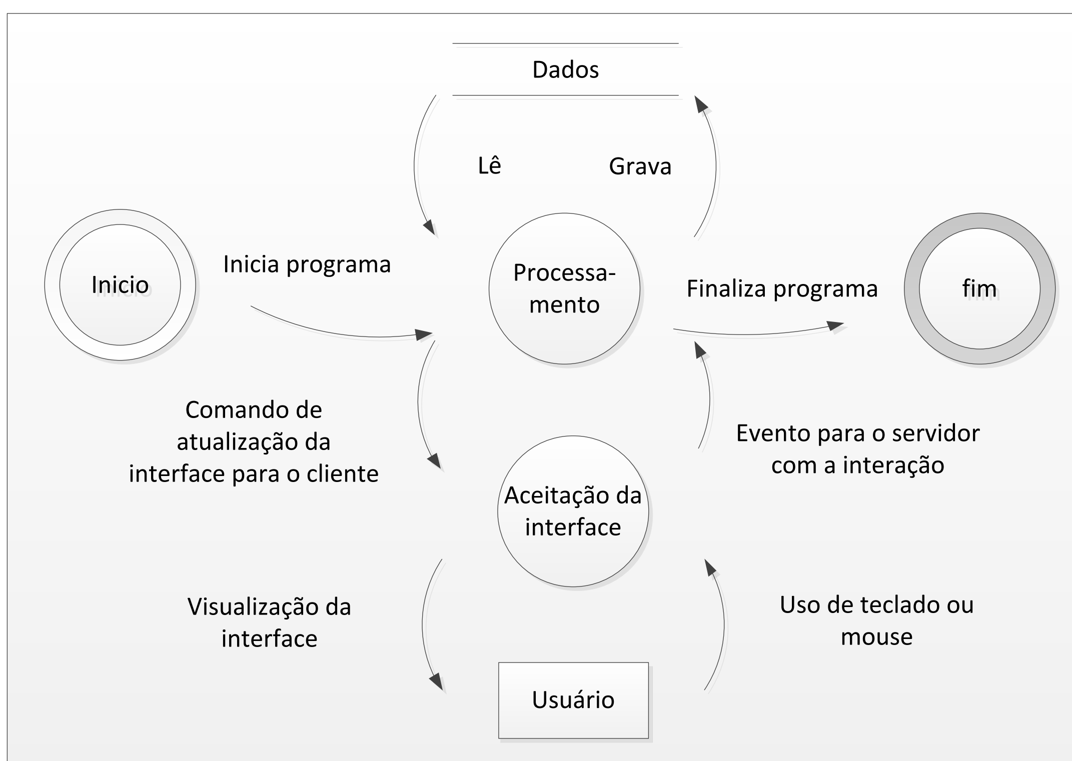

Figura 4.2 – Estados de aceitação e processamento
Fonte: autor

Várias alterações realizadas durante um processamento podem ser aglutinadas como uma única grande alteração. Isto permite que seja feita uma única transmissão que pode afetar diversas alterações de propriedades visuais, causando economia do fluxo de dados. Um exemplo desta situação ocorre quando aplicação cria o componente numa linha de código e

nas linhas seguintes define as propriedades – as definições de propriedades serão inseridas na mesma mensagem de criação. Para permitir este recurso foi desenvolvido um controle de fila de mensagens e outro controle para acumular mensagens por componente, conforme demonstrado no código fonte da Figura 4.3.

Enquanto ocorre processamento o comando *sendMessage* armazena as mensagens para posterior envio. A fila de mensagens deve ser enviada ao final do processamento ou se for causado uma interação com o usuário, tal como uma mensagem de confirmação. Cada evento é disparado por uma *thread* separada, então é possível haver processamento de rotinas distintas simultaneamente. Devido a este fato o controle de fila deve ter um objeto de fila para cada *thread*, de modo que cada uma tenha suas próprias filas de mensagens.

```
1.      private HashMap<Thread, ArrayList<Message>> _batch = new HashMap<>();
2.
3.      private synchronized ArrayList<Message> batch() {
4.          ArrayList<Message> result = _batch.get( Thread.currentThread() );
5.          if ( result == null ) {
6.              result = new ArrayList<>();
7.              _batch.put( Thread.currentThread(), result );
8.          }
9.          return result;
10.     }
11.
12.     public synchronized Communication sendMessage( Message message ) {
13.         if ( inBatch ) {
14.             addBatch( message );
15.             return this;
16.         }
17.         sendMessage( message.toString() );
18.         return this;
19.     }
```

Figura 4.3 – Fila de mensagens
Fonte: autor

A fila de mensagem é definida como um vetor de mensagens tal como *ArrayList<>* (FLANAGAN, 2005, p. 758). Esta fila, contendo várias mensagens, pode ser unificada como uma única mensagem. Na especificação do TBAP esta mensagem é definida pelo nome *ARRAY*. Para facilitar esta transformação a classe *Message* possui uma função estática *arrayToMessage*. Esta mensagem de vetor é transmitida e recebida pelo outro lado. Para identificar se a mensagem é um vetor é disponibilizada a função *isArray*. Se necessário a mensagem pode ser desmontada e obtidas as mensagens contidas através da função *getArray*. Estas características são demonstradas pelo código fonte da Figura 4.4.

```
1.      private static String ARRAY = "ARRAY";
2.
3.      public boolean isArray() {
4.          return isPropertyExists( ARRAY );
5.      }
6.
7.      public static Message arrayToMessage( ArrayList<Message> messages ) {
8.          JSONArray ja = new JSONArray();
9.          try {
10.             for ( Message c : messages )
11.                 ja.put( new JSONObject( c.toString() ) );
12.         } catch ( JSONException ex ) {
13.             Logger.getLogger( Message.class.getName() ).log( Level.SEVERE,
   null, ex );
14.         }
15.         return new Message( ARRAY ).setObjectByName( ARRAY, ja );
16.     }
17.
18.     public ArrayList<Message> getArray() {
19.         ArrayList<Message> result = new ArrayList<>();
20.         JSONObject j;
21.         Message c;
22.         try {
23.             JSONArray ja = jsonObj.getJSONArray( ARRAY );
24.             for ( int i = 0; i < ja.length(); i++ ) {
25.                 j = ja.getJSONObject( i );
26.                 c = new Message().setFromString( j.toString() );
27.                 result.add( c );
28.             }
29.         } catch ( JSONException ex ) {
30.             Logger.getLogger( Message.class.getName() ).log( Level.SEVERE,
   null, ex );
31.         }
32.         return result;
33.     }
```

Figura 4.4 – Conversão de fila para uma mensagem única
Fonte: autor

Ao desenvolvedor da aplicação é disponibilizada API que disponibiliza objetos com atributos análogos aos objetos visuais. Estes objetos abstraem a transferência das mensagens que representam componentes visuais e eventos gerados pelo usuário. Portanto a utilização dos objetos *Message* e *Communication* devem ser realizados pela camada cliente que gerenciam a interface gráfica junto ao usuário e pela camada servidora que executa a aplicação e transmite as instruções visuais.

4.3 Implementação cliente

O cliente TBAP é responsável pela interface de interação com usuário, portanto deve gerar a representação visual da aplicação e interceptar as ações iniciadas pelo usuário. Para executar estes recursos o protótipo utiliza os objetos Java Swing. Entretanto a implementação do lado cliente é flexível de modo que tenha fraco acoplamento (HORSTMANN, 2003, p. 271) sobre a tecnologia de GUI. Isto é possível através da adoção de interfaces Java. A rotina de tratamento de mensagens do cliente se relaciona com estas interfaces e não diretamente com as rotinas Swing.

A classe abstrata *Client* é responsável pelo gerenciamento do cliente. Esta classe instancia um objeto *Communication* e implementa a interface *ICommunicationListener.* Isto permite que a classe *Client* possa enviar e receber mensagens TBAP. O servidor envia ao cliente mensagens de alteração visual, por isso as mensagens são interpretadas e associadas a elementos visuais. A classe abstrata *Component* representa um componente, que pode ser uma janela ou campos. Cada componente possui um identificador numérico, nas mensagens recebidas pelo cliente este número é descrito pela propriedade denominada *COMPONENT_ID*. Isto permite determinar a qual componente a mensagem se refere. O cliente armazena em lista os componentes criados, então através do identificador é possível retornar o componente em questão.

As instruções de criação e alteração de atributo visual são geradas pelas mensagens *MESSAG_NAME_CREATE_COMPONENT* e *MSG_NAME_COMP_PROPERTY_CHANGE*. Quando estas mensagens são recebidas é recuperado o objeto componente definido pelo seu identificador e é executado o método *changeProperty* do *Component*. Este método é responsável para analisar o nome da propriedade defina na mensagem e executar o método apropriado. Um exemplo seria receber a propriedade *PROPERTY_NAME_WIDTH* e passar o seu valor para o método *setWidth*.

Para a construção do cliente Swing é declarada a classe *ClientSwing* que implementa os métodos abstratos da classe *Client*. Dentre eles estão os métodos para exibição de mensagens, confirmação e de entrada de dado (*inputbox*). A execução de mensagens recebidas também fica sobre responsabilidade da classe descendente através do método *doProcessMessageReceived*, que recebe uma interface *Runnable*. Para as classes Swing a interação com a GUI deve executa em uma *thread* sincronizada com a thread do Swing. Para isso a interface *Client* é utiliza o padrão de projeto *Factory* (FREEMAN; FREEMAN; SIERRA, 2004) - do qual delega a criação dos objetos *Component* para uma interface

chamada *IClientFactory*, que é retornada pela função abstrata *doCreateClientFactory*. Quando o cliente recebe a mensagem de criação componente é invocado o método de construção de acordo com o tipo de componente, conforme Figura 4.5. A classe *ClientSwing* implementa esta função que por usa vez retorna a classe *ClientSwingFactory*, que é a responsável por criar os outros objetos que criam componentes Java Swing: *Frame*, *Button*, *ComboBox*, *Edit* e *Label*. Estes objetos herdam de uma classe em comum, o *ComponentSwing* que por usa vez herda a classe *Component* e implementa seus métodos abstratos. Propriedades em comum são definidas nesta classe, tal como dimensão, posição, texto e ativação (*enabled*). Outros recursos em comum também são codificados nesta forma centralizada, tal como geração de mensagens de eventos.

```
1.      private void fireMessageReceived( Message _message ) {
2.          ArrayList<Message> _messages;
3.          if ( _message.isArray() )
4.              _messages = _message.getArray();
5.          else {
6.              _messages = new ArrayList<>();
7.              _messages.add( _message );
8.          }
9.          // Travel messages receiveds
10.         for ( Message m : _messages ) {
11.             String n = m.getName();
12.             ...
13.             // Create component message
14.             if ( n.equals( Protocol.MSG_NAME_CREATE_COMPONENT ) )
15.                 createComponent( m );
16.             ...
17.         }
18.     }
19.
20.     private void createComponent( Message message ) throws
    NumberFormatException {
21.         String cn = message.getValueByName( Protocol.COMPONENT_CLASS );
22.         int id = message.getIntByName( Protocol.COMPONENT_ID );
23.         // Inialine component
24.         IComponent cm = null;
25.         // Frame/window
26.         if ( cn.equals( Protocol.COMPONENT_FRAME ) )
27.             cm = clientFactory.createFrame();
28.         // Edit
29.         if ( cn.equals( Protocol.COMPONENT_EDIT ) )
30.             cm = clientFactory.createEdit();
31.         // ComboBox
32.         if ( cn.equals( Protocol.COMPONENT_COMBOBOX ) )
33.             cm = clientFactory.createComboBox();
34.         // Label
35.         if ( cn.equals( Protocol.COMPONENT_LABEL ) )
36.             cm = clientFactory.createLabel();
37.         // Button
38.         if ( cn.equals( Protocol.COMPONENT_BUTTON ) )
39.             cm = clientFactory.createButton();
40.         // If not created component
41.         if ( cm == null )
42.             return;
43.         // Define id component
44.         cm.setId( id );
45.         cm.setClient( this );
46.         // Add new component to component list
47.         components.put( new Integer( id ), cm );
48.       ...
49.    }
```

Figura 4.5 – Criação de componente pelo cliente
Fonte: Autor

As classes Swing não são *thread-safe*, ou seja, o acesso às classes não podem ser feito em qualquer *thread* (LOY; ECKSTEIN, 2002, p. 1118). Por padrão a manipulação das classes são feitas pela *thread* principal da aplicação. Porem, no caso do *framework* cliente, é utilizada uma *thread* específica para a recepção das mensagens recebidas. Os comandos de exibição dos componentes visuais são interpretados e disparados a partir desta *thread*. A camada base de implementação do cliente deve delegar a execução do comando de interface para a classe herdeira. Para isso a classe base cria uma interface *Runnable* que é passada por parâmetro para um método abstrato *doProcessMessageReceived*, conforme código fonte da Figura 4.6.

```
1.      @Override
2.      public synchronized void processMessageReceived( Message message ) {
3.          // Define runnable interface do process message received
4.          class RunThread implements Runnable {
5.
6.              public Communication communication;
7.              public Message message;
8.
9.              @Override
10.             public void run() {
11.                 fireMessageReceived( message );
12.                 communication.finishMessageReceived( message );
13.             }
14.         }
15.         RunThread rt = new RunThread();
16.         rt.message = message;
17.         rt.communication = communication;
18.         doProcessMessageReceived( rt );
19.     }
20.
21.     public abstract void doProcessMessageReceived( Runnable runnable );
```

Figura 4.6 – Execução de mensagem recebida pelo cliente
Fonte: Autor

Este método é implementado pela camada Swing, onde a execução da interface – que faz o acesso às classes do Swing – deve ser disparada pela *SwingUtilities.invokeLater* (LOY; ECKSTEIN, 2002, p. 1118). Esse comando “agenda” a instrução para ser executado de forma sincronizada com a *thread* da interface, que é visualizado pelo código fonte da Figura 4.7.

O ponto inicial de execução do cliente deve ser definido no método *main* pela classe *ClientSwing* que por sua vez deve invocar o mesmo método da classe antecessora. O mesmo

cliente permite execução de diferentes aplicações, assim como acessar servidor de diferente IP. Essa flexibilidade é definida por parâmetros definidos no método *main*.

```
@Override
public void doProcessMessageReceived( Runnable runnable ) {
    SwingUtilities.invokeLater( runnable );
}
```

Figura 4.7 – Execução de mensagem pelo cliente Swing
Fonte: Autor

O relacionamento entre as classes do cliente são demonstrada através do diagrama de classes na Figura 4.8.

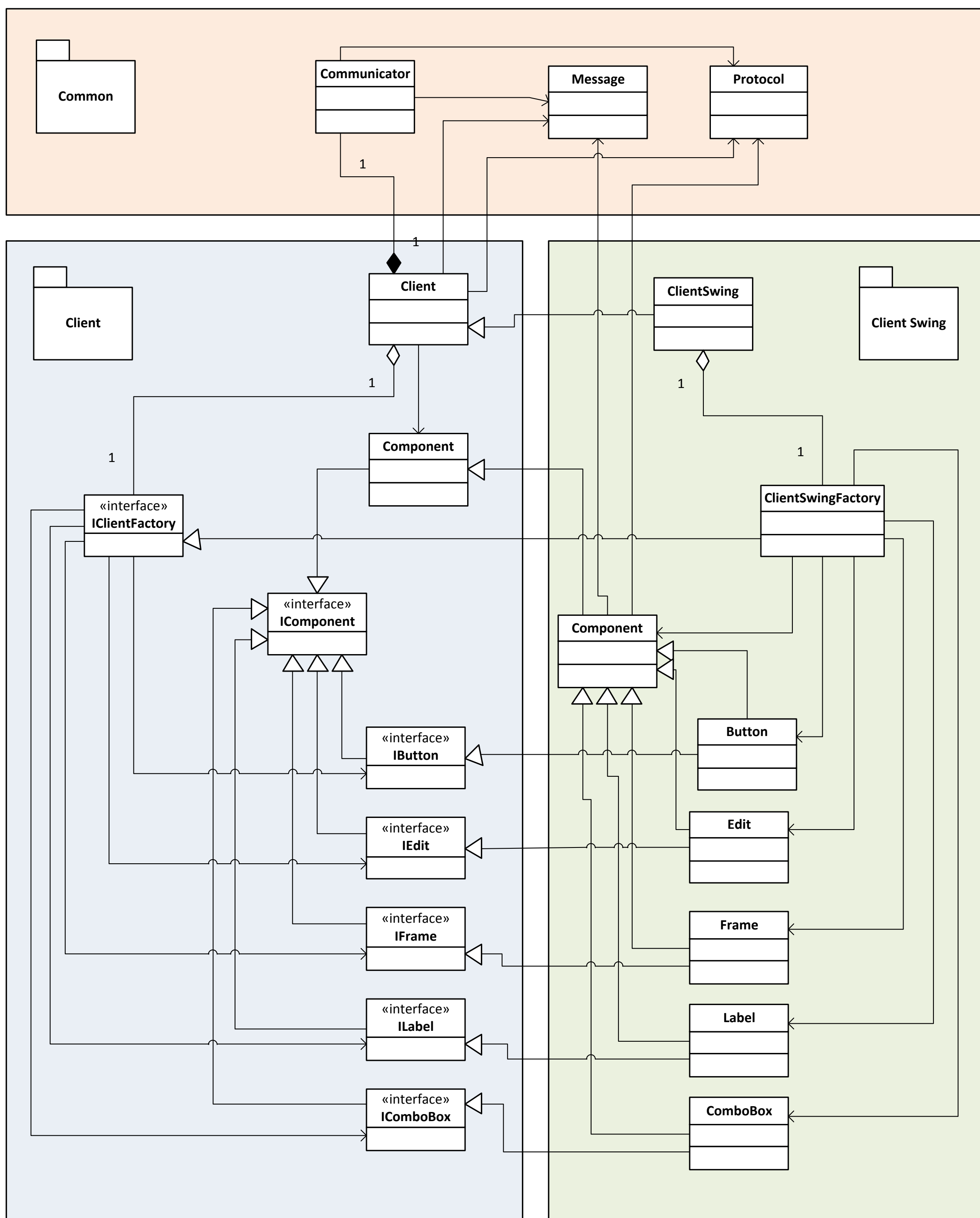

Figura 4.8 – Diagrama de classes para *framework* cliente
Fonte: Autor

4.4 Implementação servidor

O servidor é responsável por emitir a representação visual para o cliente pelo protocolo TBAP, segundo a definição realizada pela aplicação. É este *framework* que fornece ao desenvolvedor a API para construção de aplicações, isto inclui comandos para exibir

componentes visuais e interceptações de eventos do usuário. Para cada componente visual é disponibilizada uma classe equivalente. Para o servidor é indiferente a tecnologia utilizada para implementação da GUI, ou seja, o servidor não faz nenhuma referência a classes visuais tal como Swing. As classes utilizadas pelo servidor podem ser divididas em duas partes: as que interagem com a comunicação do protocolo e as que fornecem API para aplicações.

O servidor abre uma porta TCP através de um objeto *ServerSocket*, para isso é declarada a classe *Server* que opera como um *Singleton* (FREEMAN; FREEMAN; SIERRA, 2004). É definido um laço de aceitação para a porta que permanece em aguardo até ocorrer uma conexão de um cliente. Quando é estabelecida uma comunicação é instanciado um objeto da classe *Client*. Denota-se que esta classe não é a mesma da classe homônima declarada na camada cliente.

A classe *Client* simboliza o estado de um cliente que será associado a uma aplicação de regra de negócio, definida pela classe abstrata *Application*. Pelo fato do servidor poder executar diferentes aplicações o cliente deve indicar através de mensagem qual é o nome da aplicação. Para isso é utilizada a mensagem *MSG_NAME_INQUIRE_APPLICATION_NAME* que retorna o nome da aplicação na propriedade *APPLICATION_NAME*. O objeto *Server* utiliza a interface *IApplicationFactory* que possui objetivo de retornar um objeto *Application* através de um nome. É utilizado o recurso de interface, pois se tornam flexíveis as possibilidades para retornar e instanciar a aplicação desejada.

Os objetos da classe *Client* são executados em uma nova *thread* após início da conexão. Com isso o objeto *Server* pode prosseguir e aguardar conexão de um novo cliente (COULOURIS; DOLLIMORE; KINDBERG; BLAIR, 2011, p. 294). Este objeto instancia um objeto da classe *Communication* que interpreta as mensagens TBAP e permite enviar mensagens ao cliente. Também é a implementada a interface *ICommunicationListener*, tornando possível retornar mensagens geradas pela camada cliente.

A classe *Application* disponibiliza métodos para construção de diferentes classes de programas, para isso deve ser herdada e implementados seus métodos abstratos *doStart* e *doEventReceived*. São fornecidos métodos para exibição de mensagens e criação de componentes visuais. Para cada tipo de campo é declarada uma classe, que herdam da classe *Component*. Esta classe define propriedades e métodos em comum a eles. Na aplicação os objetos criados são armazenados em lista e para cada objeto criado é gerado e associado um número identificador sequencial. A criação de componente é definida pela propriedade *MSG_NAME_CREATE_COMPONENT*, o nome do tipo de componente é dito pela

propriedade *COMPONENT_CLASS* e o identificador pela *COMPONENT_ID*. A classe *Application* declara um método de criação a cada componente que por sua vez chama o método centralizado *addComponent*, conforme ilustrado na Figura 4.9.

```
1.      public Button addButton() {
2.          return addComponent( new Button() );
3.      }
4.      ...
5.      private <T extends Component> T addComponent( T component ) {
6.          component.application = this;
7.          components.put( new Integer( ++componentId ), component );
8.          component.setId( componentId );
9.          Message c = new Message( Protocol.MSG_NAME_CREATE_COMPONENT );
10.         if ( component instanceof Frame )
11.             c.setValueByName( Protocol.COMPONENT_CLASS,
    Protocol.COMPONENT_FRAME );
12.         if ( component instanceof Button )
13.             c.setValueByName( Protocol.COMPONENT_CLASS,
    Protocol.COMPONENT_BUTTON );
14.         if ( component instanceof ComboBox )
15.             c.setValueByName( Protocol.COMPONENT_CLASS,
    Protocol.COMPONENT_COMBOBOX );
16.         if ( component instanceof Edit )
17.             c.setValueByName( Protocol.COMPONENT_CLASS,
    Protocol.COMPONENT_EDIT );
18.         if ( component instanceof Label )
19.             c.setValueByName( Protocol.COMPONENT_CLASS,
    Protocol.COMPONENT_LABEL );
20.         c.setValueByName( Protocol.COMPONENT_ID, componentId );
21.         // If isn´t frame and don´t have current frame then defines frame id
22.         if ( !( component instanceof Frame ) && ( currentFrame != null ) )
23.             c.setValueByName( Protocol.PARENT_FRAME_ID, currentFrame.getId()
    );
24.         // Send message to cliente create component
25.         client.send( c );
26.         return component;
27.     }
```

Figura 4.9 – Criação de componente na aplicação
Fonte: Autor

Para facilitar a programação muitas funções da API possuem o *design pattern Builder* (FREEMAN; FREEMAN; SIERRA, 2004), que retorna a própria classe do objeto na mesma instrução. Isto permite fazer sucessivas chamadas a funções na mesma linha, conforme código fonte ilustrado pela Figura 4.10.

```
addLabel().setLeft( 10 ).setTop( 10 ).setText( “Valor:” );
```

Figura 4.10 – Padrão *Builder* adotado na API
Fonte: Autor

Algumas funções são provenientes da classe superior *Component*. Para utilizar o recurso de *Builder* as funções da classe *Component* precisam retornar a classe que a implementa, ou seja, ao invés de retornar *Component* deve retornar a classe *Edit*, *Button*, etc. Para isso precisa permitir alterar a referência do retorno das funções através do uso de *generics* (FLANAGAN, 2005, p. 160), conforme exemplificado pelo código fonte da Figura 4.11.

```
1.  public abstract class Component<T extends Component> {
2.
3.          public T setWidth( int width ) {
4.             this.width = width;
5.             return (T) this;
6.          }
7.          ...
8.      }
```

Figura 4.11 – Utilização de *generics* na construção de componentes
Fonte: Autor

Na declaração das classes dos componentes finais deve ser passada a referência da classe ao antecessor *Component*, conforme código fonte da Figura 4.12.

```
public class Button extends Component<Button> ...
```

Figura 4.12 – Declaração de componente utilizando recurso de *generics*
Fonte: Autor

Através destes recursos as classes herdeiras não precisam reimplementar as funções em comum, tal como *setWidth*. Mesmo assim as funções retornam o tipo da própria classe e não o tipo ancestral, ou seja, no caso do *Button* a função *setWidth* retorna *Button*.

O funcionamento entre as classes do *framework* de servidor são demonstrados pelo diagrama da Figura 4.13.

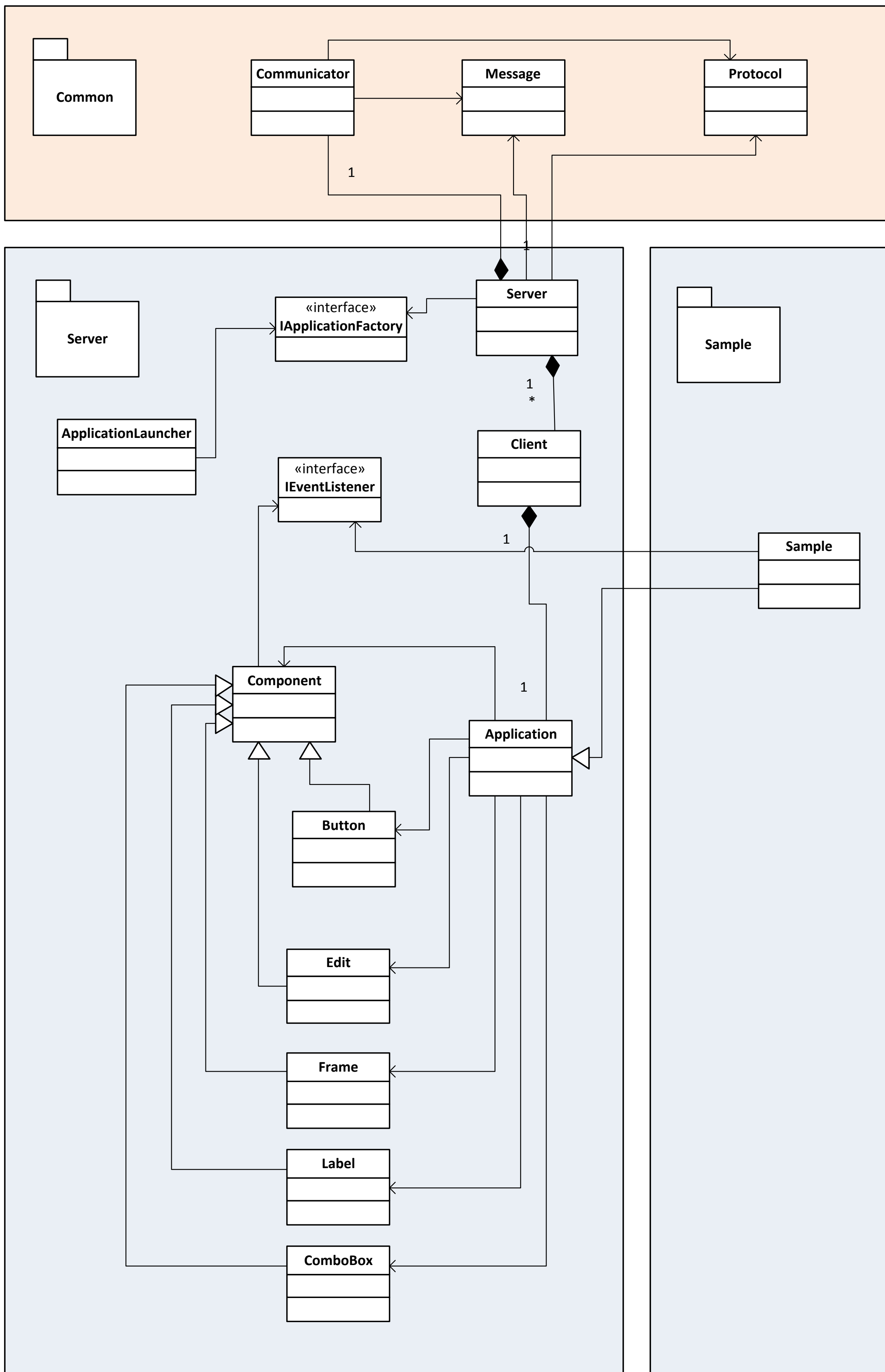

Figura 4.13 – Diagrama de classes do *framework* servidor
Fonte: Autor

4.5 Aspectos não codificados

A construção do protótipo possui um escopo restrito, tendo como foco avaliar a comunicação entre dois computadores, executando uma aplicação de forma distribuída, onde sua interface é representada num computador e seu processamento ocorre em outro. Alguns conceitos citados de sistemas distribuídos não foram inicialmente aplicados, tal como:

- Negociação de compatibilidade
- Controle de segurança, tal como criptografia e conexão TLS
- Controle de sessão para evitar invasões
- Tolerância a falhas e tratamento de reconexão

5 AVALIAÇÃO DO PROTOCOLO

Para fazer uma correta avaliação deve se levar em consideração a proposta e objetivo que o protocolo pretende alcançar. Os principais pontos são elencados e com isso é definido um cenário que explora suas características, permitindo que as situações reais de uso possam ser simuladas.

5.1 Metodologia

O protocolo proposto tem intensão de facilitar e melhorar o desenvolvimento de aplicações *thin client* em relação a alternativas atuais. Para averiguar os benefícios do protocolo é conveniente haver a comparação dos aspectos propostos do trabalho com estas atuais alternativas.

Para mensurar o desempenho de um aspecto deve ser necessário utilizar um método para registrar os índices alcançados. Para isso é necessário utilizar ferramenta apropriada para esse fim. As condições de análise devem ser equivalentes, isto inclui variação de tempo e de ambiente de execução.

Ao final da comparação é registrada a conclusão entre os comportamentos observados.

5.2 Comparação entre tecnologias atuais

Os protocolos de comunicação possuem vários atributos que podem ser mensurados e com isso estabelecer uma referência que nos indica um índice de eficiência. Uma característica conveniente de ser analisada entre os protocolo é o tráfego de dados durante um determinado período. Com isto pode-se quantificar o requisito mínimo do funcionamento ideal. A velocidade do fluxo de dados é limitada pelos recursos de rede ambiente então é possível concluir a eficiência através de comparações da necessidade de consumo de recurso de cada protocolo.

O protocolo de TBAP tem objetivo de representar aplicações gráficas em janelas, porém diferente de protocolos de mapa de *pixel* tal como RDP e VNC. Estes protocolos transmitem informações visuais de toda a tela, não diferenciam ou interpretam elementos de janelas. Apesar de possuírem diferente processamento o objetivo entre eles se assemelha, pois transportam informações visuais entre dois computadores remotos. É conveniente analisar o

funcionamento dos protocolos, com isso efetuar comparações para concluir a eficácia do modelo proposto.

5.3 Cenário

A comparação deve ser realizada através experimentações simulando um cenário prático, onde uma aplicação visual deve ser executada através de um computador remoto. As operações executadas pela aplicação devem ser as mesmas para medir o tráfego de ambos os protocolos. Para isso nos computadores envolvidos devem ser utilizados simultaneamente os serviços de cliente e servidor dos protocolos envolvidos.

Os protocolos analisados são o RDP e o VNC. Num mesmo computador remoto são executados os servidores de RDP e VNC e no computador local são executados os clientes. Para ser feita uma correta comparação ao TBAP todos os protocolos devem transferir a visualização da mesma aplicação, devido a este fato a aplicação executada é uma aplicação gerada pelo cliente do protocolo TBAP. Com isto são asseguradas que ambas as tecnologias avaliadas serão realizadas sobre as mesmas condições de tempo e de execução. Esta aplicação deve ser exibida do lado do servidor do RDP e VNC, então para a situação de comparação ocorre situação inversa para o protocolo TBAP - o cliente é executado no computador remoto. Apesar de ocorrer esta diferença não se altera a quantidade de dados transmitidos entre os computadores. A Figura 5.1 demonstra o cenário utilizado para avaliação dos protocolos sobre mesmas condições de execução e de tempo.

No computador local é executado o servidor TBAP, de onde transmite a aplicação para ser visualizada remotamente. Para operar a aplicação remotamente é utilizado cliente RDP. O cliente VNC é iniciado juntamente com o cliente RDP, o que resulta em duas visualizações simultâneas do mesmo computador remoto. Como o usuário só pode interagir com um cliente de cada vez o cliente VNC apenas recebe a visualização remota - os eventos de teclado e mouse são transmitidos pelo cliente RDP. Então nesta situação a comparação deve levar em consideração que a quantidade de dados transmitidos ao computador remoto pelo RDP seria superior o VNC.

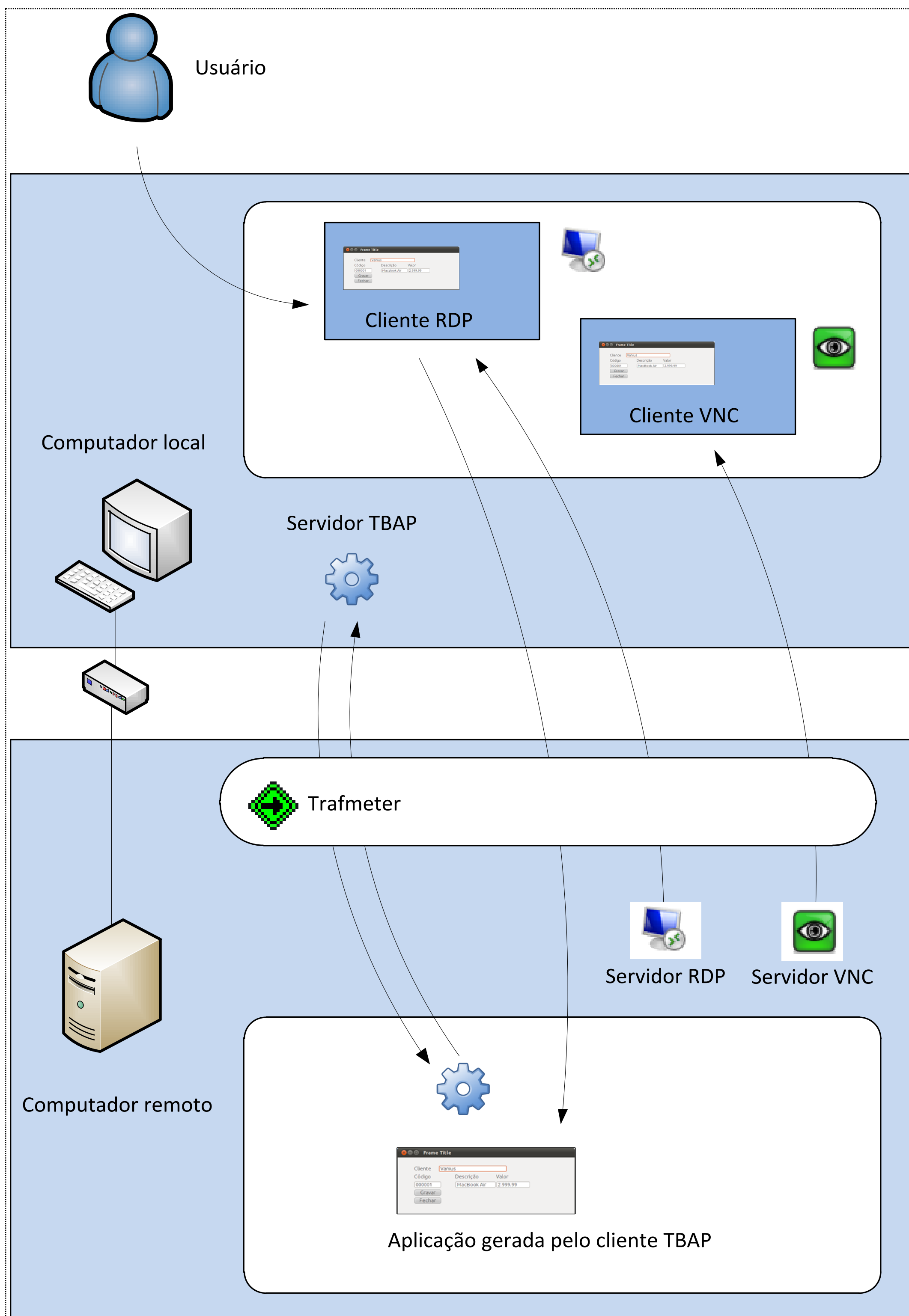

Figura 5.1 – Cenário de comparação entre protocolos
Fonte: Autor

5.4 Execução

Para ser monitorado o tráfego entre os protocolos deve ser utilizado um mecanismo que totaliza os bytes transmitidos e recebidos de forma distinta entre as tecnologias. Os protocolos avaliados possuem em comum a utilização do TCP, porém cada um utiliza uma

porta diferenciada. O RDP utiliza a porta TCP 3389, o VNC a porta 5900 e o TBAP a porta 1379. Através da análise desta característica é possível distinguir e mensurar o uso individual.

Para fazer a monitoração e totalizar o tráfego dos protocolos foi utilizado o aplicativo TrafMeter (TRAFMETER, 2012). Este aplicativo permite contar os bytes transmitidos e recebidos diferenciados por regras sobre o TCP. Existem outros aplicativos de monitoração mais difundidos, tal como Wireshark, porém o TrafMeter não tem o foco em monitorar detalhe de cada pacote TCP, mas sim gerar gráfico de bytes transferidos ao longo do tempo, a fim de analisar o total e variação do consumo de banda. Há a possibilidade de exibir gráficos distintos para cada porta, com isso é possível configurar as portas dos três protocolos e realizar a comparação.

Do lado do computador remoto foi instalado o aplicativo e configurado para totalizar a transmissão para cada porta. A monitoração por porta é feita através da configuração de regras de filtro, conforme Figura 5.2.

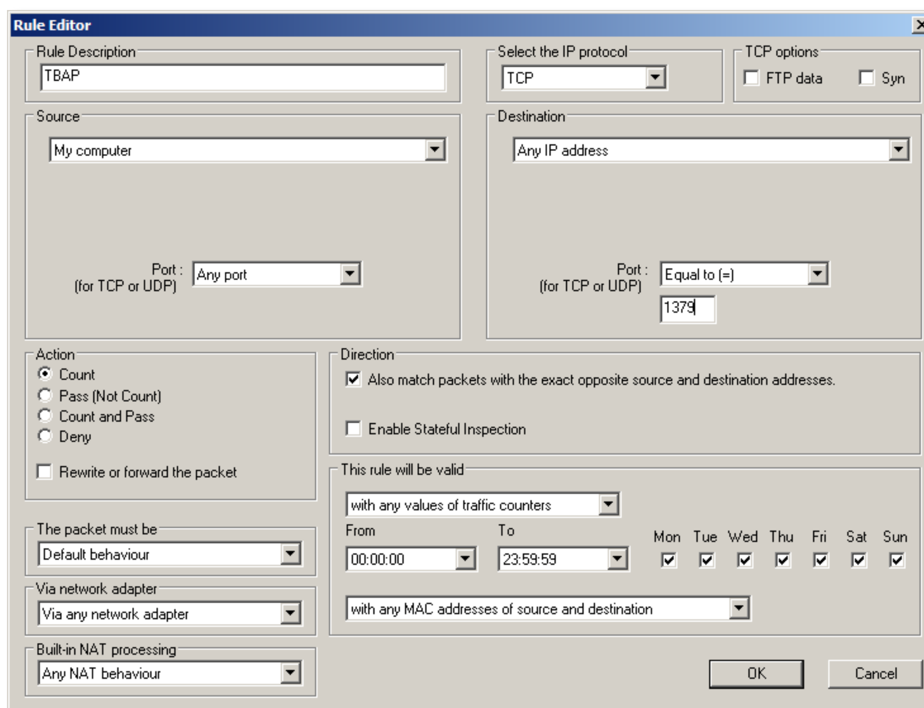

Figura 5.2 – Configuração de monitoramento do TrafMeter
Fonte: Autor

A aplicação desenvolvida possui diversos campos em que o usuário informa dado e com isso realiza cálculo exibindo o resultado em outro campo, conforme visualizado pela Figura 5.3. É exibida uma segunda janela com outros campos também aceitos pelo usuário. A

simulação ocorreu durante o tempo aproximado de dois minutos onde os campos foram preenchidos pelo usuário diversas vezes. Uma segunda janela foi exibida e fechada várias vezes, de forma que sobrepôs sobre a janela anterior. A situação de troca de janela é comum para sistemas *desktop* e é importante ocorrer durante a avaliação para forçar a retransmissão da janela.

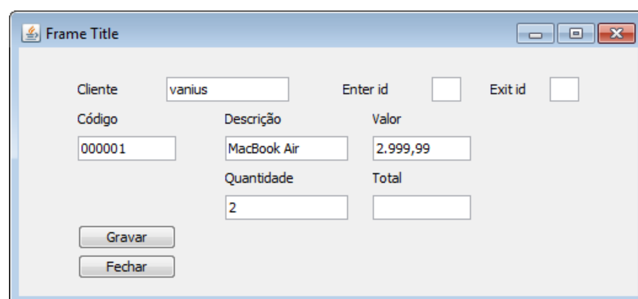

Figura 5.3 – Aplicação de teste para avaliação
Fonte: Autor

5.5 Resultado

O aplicativo TrafMeter funcionou corretamente e monitorou a movimentação de informações para cada protocolo durante o tempo que foi acionado. Os bytes enviados e recebidos foram totalizados ao longo do tempo e os resultados entre eles foram distintos, que através de análise permitiu concluir sobre quais são mais eficientes.

O gráfico do tráfego do protocolo VNC, ilustrado na Figura 5.4, confirma o alto número de bytes enviados ao cliente comparado ao recebido pelo servidor. A interação do usuário ocorreu no cliente RDP, então explica o baixo número de bytes recebidos.

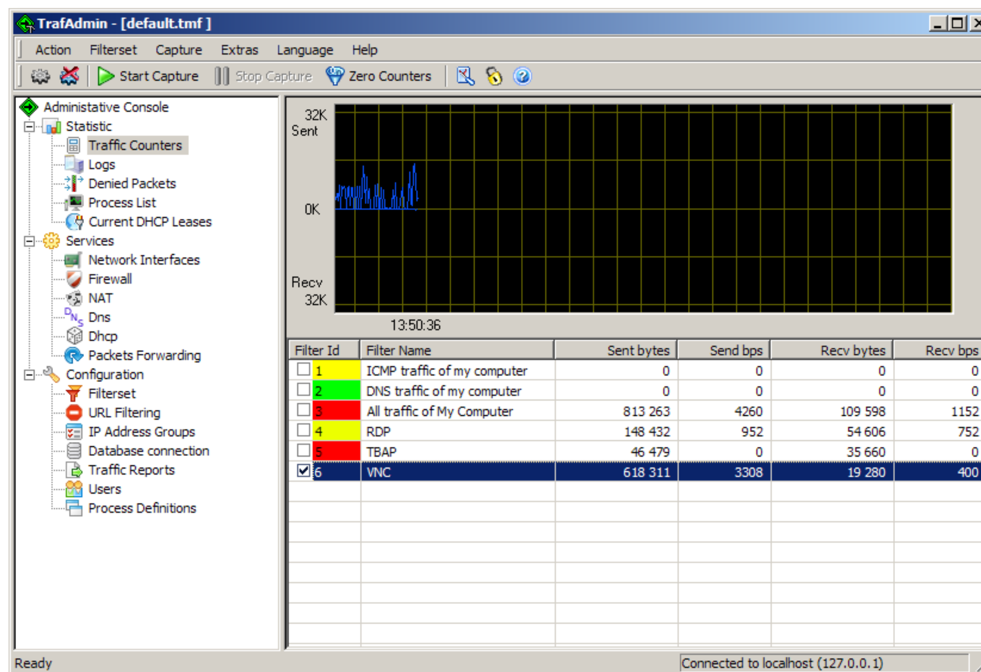

Figura 5.4 – Resultado de monitoramento com protocolo VNC
Fonte: Autor

A Figura 5.5 demonstra o fluxo do protocolo RDP. Apesar de haver um pico de transferência de bytes enviados, em geral se demonstrou mais uniforme em comparação ao VNC.

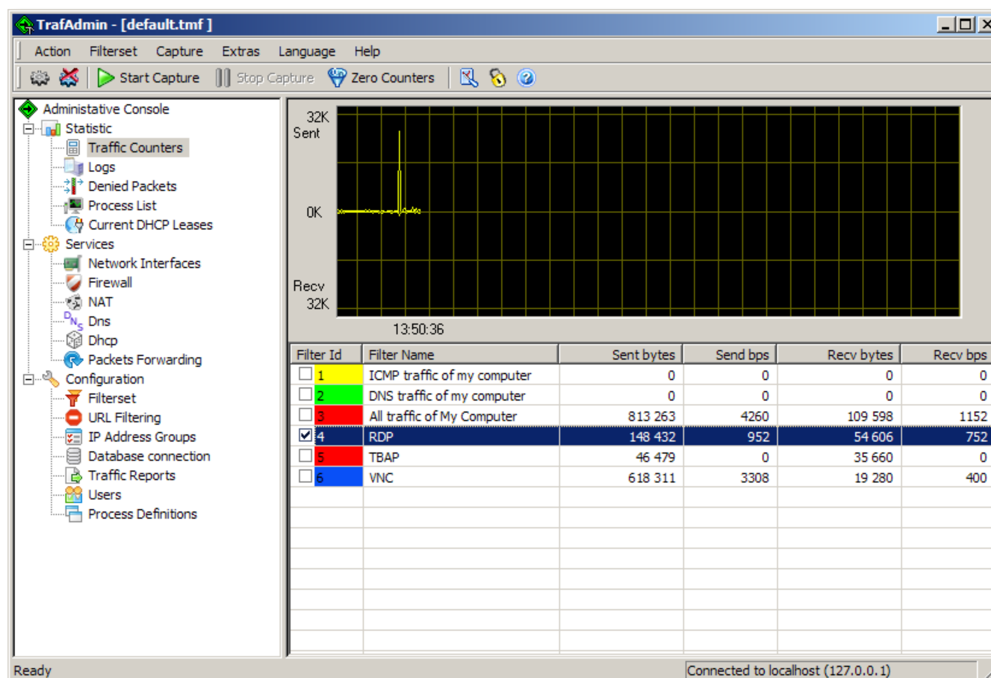

Figura 5.5 – Resultado de monitoramento com protocolo RDP
Fonte: Autor

A monitoração do protocolo TBAP é ilustrada pela Figura 5.6. O tráfego se demonstrou baixo e uniforme durante toda avaliação.

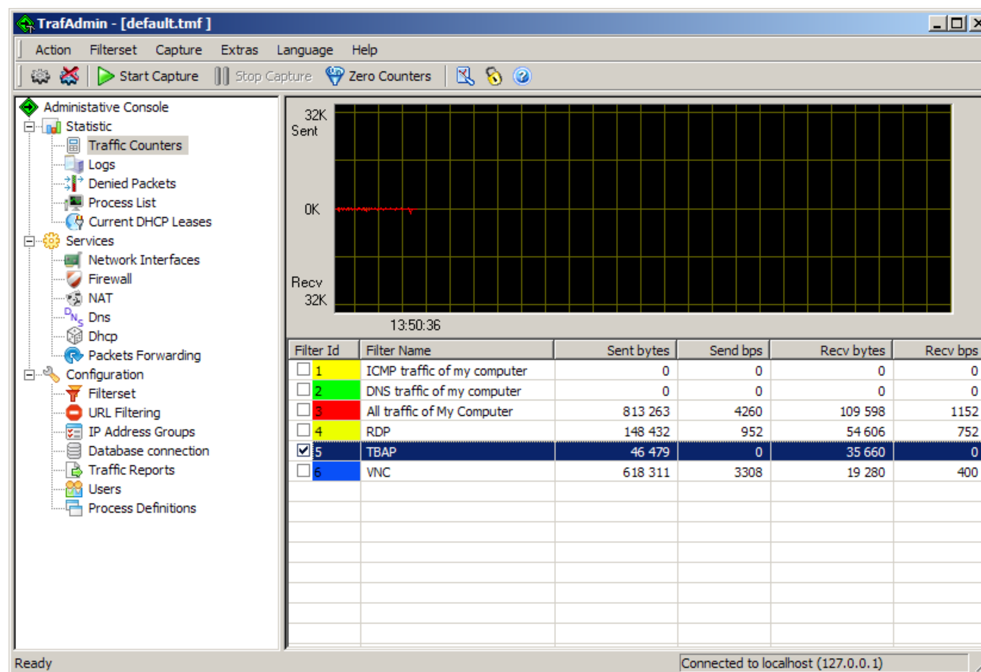

Figura 5.6 – Resultado de monitoramento com protocolo TBAP
Fonte: Autor

Conforme listado na Tabela 5.1, para todos os protocolos a quantidade dos bytes enviados foi superior a de bytes recebidos no lado do computador remoto. Nos protocolos RDP e VNC os dados enviados representam a mudança visual ocorrida pela aplicação devido à interação com o usuário. Diferentemente, neste caso de avaliação, para o TBAP os dados enviados representam os eventos gerados pelo usuário. Para o RDP, onde ocorreu à interação com o usuário, os dados recebidos representam eventos de teclado e mouse. Compreensivelmente para o RDP o fluxo de tela é bastante superior às informações geradas pelo usuário. Porém o TBAP possuiu valores de transmissão e recepção bastante próximos. Isto comprova que o TBAP efetua poucas transmissões para representação visual.

Tabela 5.1 – Resultado da comparação entre os protocolos

Protocolo	Bytes recebidos	Bytes enviados	Total
VNC	19.280	618.311	637.591
RDP	54.606	148.479	203.085
TBAP	33.660	46.479	80.139

Fonte: Autor

Em comum para todos os protocolos o número de bytes enviados representa a maior parte dos dados transmitidos, por conseguinte sua significância é proporcional à soma de bytes e recebidos. Logo não seria errôneo concluir sobre o total de bytes trafegados. Analisado neste contexto o protocolo TBAP demonstrou ser 87% mais eficiente que o VNC e 60% mais eficiente que o RDP.

No formato do protocolo TBAP as mensagens são textos contendo palavras longas, sem siglas. O número de bytes transferidos poderia ser reduzido caso fosse adotado uma representação binária ao invés de texto JSON e UTF8. Para isso as palavras reservadas das mensagens deveriam codificadas para números binários associados e teria que ser definido um formato para delimitar os pares chave e valor. Outra possibilidade seria utilizar compreensão dos bytes do formato atual. Porém para isso seria necessário um processamento computacional adicional para transmissão e recepção. Inicialmente estas alternativas não serão implementadas.

6 EXPERIMENTAÇÕES

6.1 Aplicação de negócio

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 próximo ao objetivo do artigo, conforme código fonte da Figura 6.1.

```
1.  public class Sample extends br.com.felinelayer.server.application.Application {
2.      private Frame f;
3.      public Sample() {
4.          setApplicationName( "Sample" );
5.      }
6.
7.      @Override
8.      protected void doStart() {
9.          String nome = showInput( "Digite seu nome:" );
10.         showMessage( "Bem vindo " + nome + "!" );
11.         f = addFrame().setWidth( 520 ).setHeight( 200 );
12.         f.addLabel().setCol( 1 ).setText( "Cliente" );
13.         f.addEdit().setCol( 3 ).setText( nome ).setWidth( 200 );
14.         f.addRow();
15.         f.addLabel().setCol( 1 ).setText( "Código" );
16.         f.addLabel().setCol( 5 ).setText( "Descrição" );
17.         f.addLabel().setCol( 10 ).setText( "Valor" );
18.         f.addRow();
19.         f.addEdit().setCol( 1 ).setText( "000001" ).setWidth( 60 );
20.         f.addEdit().setCol( 5 ).setText( "MacBook Air" ).setWidth( 80 );
21.         f.addEdit().setCol( 10 ).setText( "2.999,99" ).setWidth( 60 );
22.         f.addRow();
23.         f.addButton().setCol( 1 ).setText( "Gravar" ).setWidth( 80 );
24.         f.addRow();
25.         f.addButton().setCol( 1 ).setText( "Fechar" ).setWidth( 80 );
26.     }
27.
28.     @Override
29.     protected void doMessageReceived( Message message ) {
30.         String eventName = "";
31.         if (message.isPropertyExists( Protocol.EVENT_NAME) )
32.             eventName = message.getValueByName( Protocol.EVENT_NAME );
33.         if ( eventName.equals( Protocol.EVENT_BEFORE_CLOSE ) ) {
34.             String resposta = showInput( "Pode fechar? (\"sim\")" );
35.             if ( resposta.equalsIgnoreCase( "sim" ) )
36.                 message.setValueByName( Protocol.EVENT_BEFORE_CLOSE_CANCLOSE,
    Protocol.YES );
37.             else
38.                 message.setValueByName( Protocol.EVENT_BEFORE_CLOSE_CANCLOSE,
    Protocol.NO );
39.         }
40.     }
41. }
```

Figura 6.1 – Código de aplicação de negócio
Fonte: Autor

6.2 Fluxo do protocolo

Abaixo, na, Tabela 6.1 é demonstrado 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*):

Tabela 6.1 - Exemplo de mensagens em formato JSON

Servidor e aplicação de negócio		Cliente Swing
05:57:20:045 Sent : {"MESSAGE_NAME":"ARRAY","ARRAY":[{"COMPONENT_HEIGHT":"200","ID":"1","COMPONENT_CLASS":"FRAME","COMPONENT_WIDTH":"520","MESSAGE_NAME":"CREATE_COMPONENT"},{"COMPONENT_TEXT":"Cliente","PARENT_FRAME_ID":"1","COMPONENT_TOP":"24","ID":"2","COMPONENT_CLASS":"LABEL","COMPONENT_LEFT":"48","MESSAGE_NAME":"CREATE_COMPONENT"},{"COMPONENT_TEXT":"Vanius","PARENT_FRAME_ID":"1",...]}	->	05:57:20:051 Received : {"MESSAGE_NAME":"ARRAY","ARRAY":[{"COMPONENT_HEIGHT":"200","ID":"1","COMPONENT_CLASS":"FRAME","COMPONENT_WIDTH":"520","MESSAGE_NAME":"CREATE_COMPONENT"},{"COMPONENT_TEXT":"Cliente","PARENT_FRAME_ID":"1","COMPONENT_TOP":"24","ID":"2","COMPONENT_CLASS":"LABEL","COMPONENT_LEFT":"48","MESSAGE_NAME":"CREATE_COMPONENT"},{"COMPONENT_TEXT":"Vanius","PARENT_FRAME_ID":"1",...]}
05:57:25:406 Received : {"EVENT_NAME":"EVENT_BEFORE_CLOSE","EVENT_COMPONENT_ID":"1","CAN_CLOSE":"YES","INQUIRING_ID":"1","MESSAGE_NAME":"EVENT"}	<-	05:57:25:405 Sent : {"EVENT_NAME":"EVENT_BEFORE_CLOSE","EVENT_COMPONENT_ID":"1","CAN_CLOSE":"YES","INQUIRING_ID":"1","MESSAGE_NAME":"EVENT"}
05:57:28:902 Sent : {"EVENT_NAME":"EVENT_BEFORE_CLOSE","EVENT_COMPONENT_ID":"1","CAN_CLOSE":"YES","INQUIRED_ID":"1","MESSAGE_NAME":"EVENT"}	->	05:57:28:903 Received : {"EVENT_NAME":"EVENT_BEFORE_CLOSE","EVENT_COMPONENT_ID":"1","CANCLOSE":"YES","INQUIRED_ID":"1","MESSAGE_NAME":"EVENT"}

Fonte: Autor

6.3 Execução em rede heterogênea

Um dos aspectos analisados através de experimentação é o funcionamento do protocolo através de redes com envolvendo computadores de diferentes arquiteturas. O protocolo TCP é independente de plataforma, assim como formato UTF8 e JSON. O protocolo TBAP - por funcionar sobre estas outras camadas - deve ter o mesmo comportamento de ser inerente a sistema operacional ou plataforma de execução.

Foi realizada experimentação utilizando computadores com sistema operacional Windows 7 64 bits e Linux Ubuntu 11.04 64 bits. Na Figura 6.2 é ilustrada a relação entre os

sistemas. A portabilidade da *runtime* do Java permite que a visualização gráfica da aplicação seja bastante próxima. As características das janelas foram geradas pelo servidor e exibidas pelos clientes.

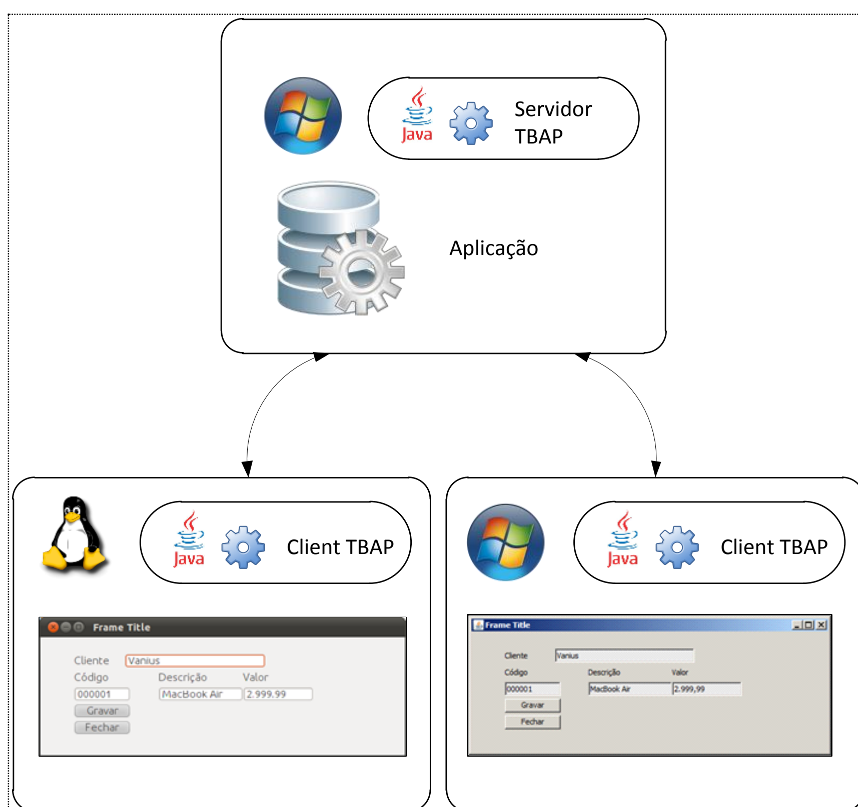

Figura 6.2 – Comportamento do cliente em diferentes sistemas operacionais
Fonte: Autor

O comportamento e a visualização da aplicação foram exatamente os mesmos para diferentes sistemas operacionais. Apesar de o protótipo utilizar Java, o protocolo TBAP não possui dependência sobre esta tecnologia. Em uma mesma rede pode haver servidor e clientes utilizando outras plataformas ou linguagens.

6.4 Teste em rede de baixa velocidade

Para avaliar a eficácia do protocolo é conveniente executar sobre uma condição crítica, para isso foi definido um cenário simulando uma conexão discada com largura de banda máxima de 56Kbps. Esta situação foi realizada através do *software* Trafmeter. É possível criar um filtro para uma dada porta TCP e a partir disto definir um limite de velocidade, conforme exemplificado na Figura 6.3. Foi inserido filtro para o protocolo TBAP através da porta 1379 e definido limite de 7KBps.

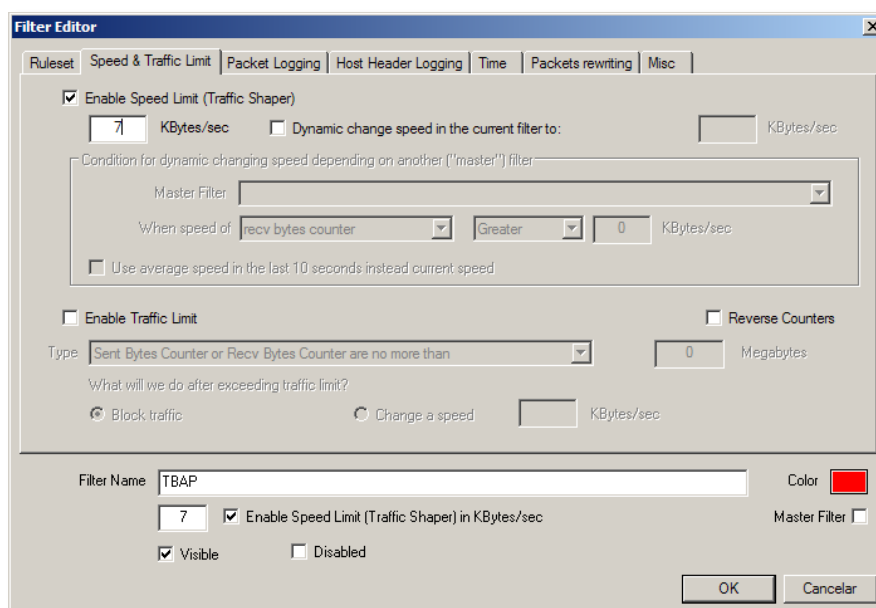

Figura 6.3 – Configuração de limite de velocidade no Trafmeter
Fonte: Autor

Após o filtro configurado é acionada a monitoração do tráfego, que exibe em gráfico a velocidade obtida durante o tempo. A aplicação protótipo foi executada acessando servidor remoto e feita interação com o usuário. Conforme visualizado na Figura 6.4, durante a simulação de uso nenhum pico de transmissão se aproximou dos 7KBps. Isto comprova que a largura de banda necessária pelo protocolo é pequena e pode funcionar adequadamente inclusive em ambiente limitados, tal como acessar um computador remotamente através de conexão discada.

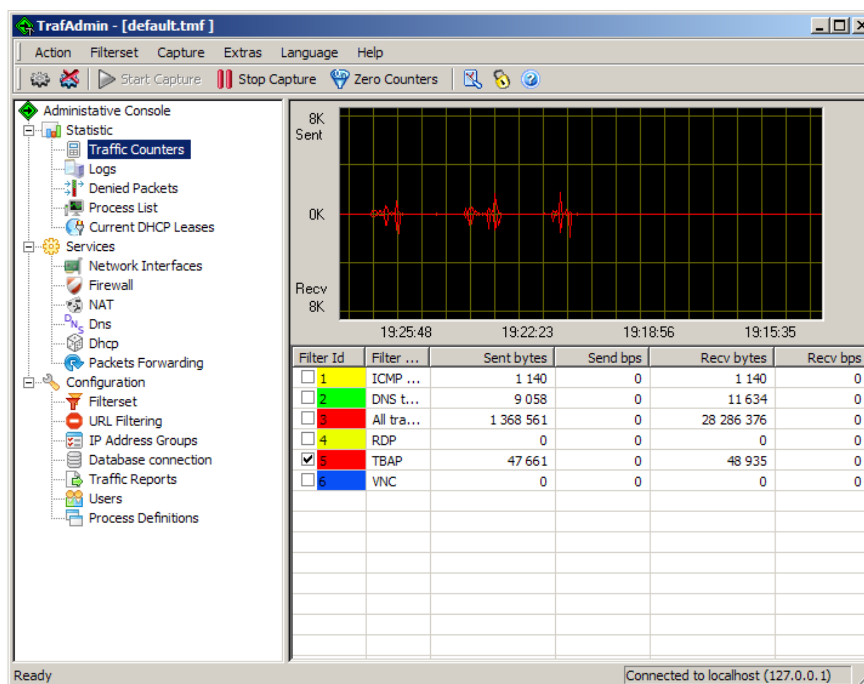

Figura 6.4 – Desempenho do protocolo sobre conexão lenta
Fonte: Autor

6.5 Resultados obtidos

Com o desenvolvimento 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.
- A baixa necessidade de largura banda de rede confirmou sua eficiência, inclusive para execução em ambientes limitados.

CONCLUSÃO

Neste trabalho se estudou o uso em conjunto do modelo cliente-servidor *thin client* com interface gráfica em janelas. O seu uso pode resultar em uma solução bastante flexível como uma 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.

É proposta a especificação de um protocolo para representação visual de aplicações em janelas, pela qual são trafegados atributos dos componentes visuais. Através da implementação do protótipo foi possível comprovar sua funcionalidade e expôr várias vantagens sobre outras tecnologias. Por consumir menos recursos de rede o protocolo demonstrou-se ser mais eficiente. Um dos motivos disto é a característica de transferir atributos dos campos visuais ao invés de detalhes da tela. Outro fator é o tratamento diferenciado de eventos que evita certos fluxos e elimana períodos de latência. As experimentações comprovaram que os requisitos de largura de banda e latência não são exigentes, podendo funcionar em situações críticas.

Pelo fato de funcionar sobre padrões abertos seu uso é facilitado e pode-se operar sobre diversos ambientes. Apesar de permitir executar aplicações cliente-servidor, a visualização da aplicação funciona igual a uma aplicação *desktop* convencional. Para o usuário é apresentada uma experiência transparente de sistema distribuído, ou seja, não é perceptiviel que a aplicação na realidade é executada remotamente.

O protocolo TBAP se diferencia entre tecnologias usuais, inclusive 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* remota 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.

Como trabalhos futuros este trabalho deixa alguns aspectos que podem ser expandidos e ter um estudo aprofundado, a fim de aperfeiçoar ou agregar novas possibilidades de aplicação.

Para aplicações comerciais não se pode deixar de citar a importância de relatórios e recursos de impressão. Poderá 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. Pode ser especificado algo semelhante ao recurso do HTML5 de *cache* local de pares chaves-valor (FREEMAN; ROBSON, 2011, p. 423).

O componente *grid* foi previsto, porém sua implementação não foi realizada no protótipo. O *grid* possui uma peculiaridade diferente entre os outros componentes, pois pode ter quantidade de dados bastante extensa. Isto implica que a carga dos dados do *grid* deve ser diferenciada. Os dados não devem ser entregues em uma única transmissão. Caso contrário poderia haver uma demora em concluir da exibição do componente. Nesta situação é conveniente haver várias transmissões, de acordo com a necessidade visual. O *grid* pode possuir ilimitadas linhas, porém as visíveis são limitadas e conhecidas - inicialmente seriam estas linhas que teriam seus dados enviados. De acordo com a interação do usuário novas linhas podem ser requisitas. Para permitir este comportamento deve-se adotar o padrão MVC (FREEMAN; FREEMAN; SIERRA, 2004, p. 532), onde os dados ficam separados da sua visualização e há um controle desacoplado para seu transporte. Mecanismo semelhante pode ser adotado a campos de caixa de texto com múltiplas linhas (campos *memo*).

Outros componentes não implementados são as imagens, incluído ícones. Além de atributos de posição e dimensão, a imagem possui sua representação visualização definida por conteúdo binário. Por padrão, formato JSON não permite a representação de conteúdo binário – apesar de que é possível representar qualquer caractere UNICODE através da notação U+0000-U+00FF (CROCKFORD, 2006). Nesta situação, em geral, para representar um byte é necessário alocar mais do que um byte. Nos casos que se deve converter conteúdo binário para textual é recomendado utilizar a conversão o formato BASE64 (JOSEFSSON, 2006). Então para a imagem binária ser transmitida deve ser serializada para o formato BASE64 – e na recepção deve ocorrer o processo inverso. Para este tipo de componente ainda devem ser definidos atributos possíveis da imagem, tal como o formato (JPEG, PNG, GIF, etc).

Por ter um formato textual, o JSON compreensivelmente possui tamanho superior em comparação a formatos binários. Isto significa que o tráfego do protocolo poderia ser reduzido se fosse utilizado dados binários. Há a possibilidade de ser criado um adendo ao

protocolo, como um formato alternativo semelhante ao JSON, tal como o formato BSON (MEMBREY; PLUGGE; THIELEN; HAWKINS, 2010). Este formato tem a proposta de ser semelhante ao JSON, porém representa as informações equivalentes em tipos binários. Existem bibliotecas para várias linguagens, permitindo inclusive fazer a conversão bilateral entre JSON e BSON. A aplicação deste modelo não causaria grande impacto nas propostas atuais, tal como regras de fluxo e palavras reservadas.

Uma possibilidade de uso do protocolo TBAP seria adoção do protocolo *WebSocket* (FETTE; MELNIKOV, 2011) como protocolo de transporte, ao invés de utilizar puramente o TCP. Este protocolo está sendo criado e definido pela W3C e IETF justamente para melhorar alguns dos problemas causados pela adoção do HTTP: permitir utilizar canais de comunicação de duas vias (*full duplex*). Este protocolo também prevê uso de modelo de segurança. Outra vantagem de adotar este protocolo é reutilizar os servidores de aplicação e clientes que adotarão este padrão. Neste cenário o servidor TBAP pode funcionar como um *middleware* no servidor de aplicação e o cliente sobre o HTML5 - executado pelo navegador *web*.

REFERÊNCIAS BIBLIOGRÁFICAS

ACHARYA, Vivek. **TCP/IP Distributed System.** New Delhi: Laxmi Publications, 2006, 474 p.

BIDGOLI, Hossein. **The Internet encyclopedia:** Volume 1. New Jersey: John Wiley And Sons, 2004, 880 p.

CAPTAINCASA GMBH. **CaptainCasa Enterprise is the Rich Internet Application** solution for Business Applications with demanding users. 2011. Disponível em: <http://www.captaincasa.com/>. Acesso em: 4 ago. 2011.

COULOURIS George; KINDBERG, Tim; DOLLIMORE, Jean; BLAIR, Gordon. **Distributed systems**: Concepts and Design. Addison-Wesley, 2011, 1008 p.

CRAFT, Melissa; BROOMES, Chris; KHNASER, Elias N.. **Configuring Citrix MetaFrame XP for Windows including future release 1**. New York: Syngress, 2002, 560 p.

CROCKFORD, Douglas. **RFC 4627– The application/json Media Type for JavaScript Object Notation (JSON)**. 2006. Disponível em: <http://tools.ietf.org/html/rfc4627/>. Acesso em: 5 mai. 2012.

DOOLEY, John. Software Development and Professional Practice. New York: Apress, 2011, 260 p.

FLANAGAN, David. **Java in a nutshell**. O'Reilly, 2005, 1224 p.

FARLEY, Jim. **Java distributed computing**. O'Reilly, 1998, 392 p.

FETTE, Ian; MELNIKOV. RFC 6455 – **The WebSocket Protocol**. 2011. Disponível em <http://tools.ietf.org/html/rfc6455>. Acesso em: 9 jun. 2012.

FREEMAN, Eric; FREEMAN, Elisabeth; SIERRA, Kathy; BATES, Bert. **Head First design patterns**. O'Reilly, 2004, 686 p.

FREEMAN, Eric; ROBSON, Elisabeth. **Head First HTML5 Programming**: Building Web Apps with JavaScript. O'Reilly, 2011, 573 p.

GENCO, A. **Mobile Agents**: Principles Of Operation Applications. WIT Press, 2008, 271 p.

GOUGH, Michael. **Skype Me**!: From Single User to Small Enterprise and Beyond. Syngress, 2005, 416 p.

HAROLD, Elliotte Rusty. **Java network programming**. O'Reilly, 3rd edition, 2004, 762 p.

HARRIS, Andy. **HTML5 for dummies quick reference**. John Wiley and Sons, 2011, 224 p.

HARWOOD, Ted. **Inside Citrix MetaFrame XP:** a system administrator's guide to Citrix MetaFrame XP/1.8 and Windows terminal serives. Indianapolis: Addison-wesley Professiona, 2002, 944 p.

HOLZNER, Steven. **Ajax: A Beginner's Guide**. McGraw-Hill Prof Med/Tech, 2008, 475 p.

HORSTMANN, Cay. **Conceitos de computação com o essencial de Java**. Bookman, 2003, 780 p.

JOSEFSSON, Simon. **RFC 4648 – The Base16, Base32, and Base64 Data Encodings**. 2006. Disponível em: <http://tools.ietf.org/html/rfc4648/>. Acesso em: 5 mai. 2012.

LOY, Marc; ECKSTEIN, Robert. Java Swing. New York: O'Reilly, 2002, 1252 p.

MACDONALD, Matthew; FREEMAN, Adam; SZPUSZTA, Mario. **Pro ASP.NET 4.0 in C# 2010.** 4. ed. New York: Apress, 2010, 1616 p.

MEERSMAN, Robert; TARI, Zahir. **On the move to meaningful Internet systems 2005:** OTM confederated international conferences CoopIS, DOA, and ODBASE 2005, Agia Napa, Cyprus, October 21-November 4, 2005 : proceedings, Parte 1. Birkhäuser: Springer. p. 781-782. 2005.

MEMBREY, Peter; PLUGGE, Eelco; THIELEN, Wouter; HAWKINS Tim. **The Definitive Guide to MongoDB**: The NoSQL Database for Cloud and Desktop Computing. New York: Apress, 2010, 328 p.

MILLER, Frederic P.; VANDOME, Agnes F.; MCBREWSTER, John. **JSON**. VDM Publishing House Ltd., 2009, 178 p.

MORIMOTO Rand; ABBATE Andrew; KOVACH Eric; ROBERTS Ed. **Microsoft Windows Server 2003 insider solutions**. Sams Publishing, 2003, 638 p.

MURUGESAN, San; DESHPANDE, Yogesh. **Web engineering:** managing diversity and complexity of Web application development. Berlin: Springer, 2001, 355 p.

OSTROVSKY, Rafail. **Security and Cryptography for Networks**: 6th International Conference. Italy: Springer, 2008, 422 p.

PACHGHARE, V.K.. **Cryptography and Information Security**. PHI Learning Pvt. Ltd, 2009, 370 p.

PARALLAX, Inc. **Programming and customizing the multicore propeller microcontroller**: the official guide. McGraw-Hil, 2010, 496 p.

PILGRIM, Mark. **HTML5**: Up and Running. O'Reilly & Google Press, 2010, 205 p.

PISA, Filippo Di. **Beginning Java and Flex**: Migrating Java, Spring, Hibernate and Maven Developers to Adobe Flex. New York: Apress, 2009, 500 p.

POWELL, Thomas A., **Ajax:** the complete reference. McGraw-Hill Prof Med/Tech, 2008, 654 p.

SCHEIFLER, Robert; GETTYS, Jim. **X Window System:** the complete reference to Xlib, X Protocol, ICCCM, XLFD. Digital Press, 1992, 1000 p.

SOSINSKY, Barrie. **Networking Bible.** Indianapolis: John Wiley And Sons, 2009, 1056 p.

STEWART, J. Michael. **Network Security, Firewalls, and VPNs**. Jones & Bartlett Publishers, 2010, 482 p.

TANENBAUM, Andrew; STEEN, Maarten Van. **Distributed Systems**: Principles and Paradigns. Upper Saddle River: Pearson Prentice Hall, 2007, 686 p.

TRAFMETER MAREL IT solutions. **Trafmeter: IP Traffic Accounting and Network Monitoring**. 2012. Disponível em: <http://www.trafmeter.com/>. Acesso em: 14 abr. 2012.

WASINGER, Rainer. **Multimodal Interaction with Mobile Devices**: Fusing a Broad Spectrum of Modality Combinations. Berlin: IOS Press, 2007, 246 p.

APÊNDICE A – ESPECIFICAÇÕES DE NOMENCLATURAS

1. Negociação

a. MESSAGE_NAME: CONFIGURATION_QUESTION

Parâmetros		Retornos	
CONFIGURATION_NAME	RESOLUTION	RESPONSE	1440X900
CONFIGURATION_NAME	COLORS_DEPTH	RESPONSE	32BITS
CONFIGURATION_NAME	OPERATIONAL_SYSTEM	RESPONSE	WINDOWS 7
CONFIGURATION_NAME	PLATAFORM	RESPONSE	JAVA 7
CONFIGURATION_NAME	GUI	RESPONSE	SWING
CONFIGURATION_NAME	PROTOCOL_VERSION	RESPONSE	1.0
CONFIGURATION_NAME	PRINTER_VERSION	RESPONSE	UNKNOWN

b. MESSAGE_NAME: END_ CONFIGURATION_QUESTION

Parâmetros e retornos: Nenhum

2. Avisos e notificações

a. MESSAGE_NAME: MESSAGE

Parâmetros	
MESSAGE	INFORMATION
MESSAGE	WARING
MESSAGE	ERROR
TEXT	<texto>

3. Atributos de componentes visuais

a. MESSAGE_NAME: CREATE_COMPONENT

Parâmetros	
COMPONENT_CLASS	COMPONENT_FRAME
COMPONENT_CLASS	COMPONENT_BUTTON
COMPONENT_CLASS	COMPONENT_COMBOBOX
COMPONENT_CLASS	COMPONENT_EDIT
COMPONENT_CLASS	COMPONENT_LABEL

PARENT_ID	<inteiro>
COMPONENT_ID	<inteiro>
LEFT	<inteiro>
TOP	<inteiro>
WIDTH	<inteiro>
HEIGHT	<inteiro>
TEXT	<texto>

b. MESSAGE_NAME: COMPONENT_PROPERTY_CHANGE

Parâmetros e retornos:

Parâmetros	
COMPONENT_ID	<inteiro>
LEFT	<inteiro>
TOP	<inteiro>
WIDTH	<inteiro>
HEIGHT	<inteiro>
TEXT	<texto>

4. Eventos

a. MESSAGE_NAME: EVENT

Parâmetros	
EVENT_NAME	ON_EXIT
EVENT_NAME	ON_ENTER
EVENT_NAME	ON_CLICK
EVENT_NAME	ON_DOUBLE_CLICK
EVENT_NAME	ON_KEY
EVENT_NAME	ON_CLOSE

APÊNDICE B - CÓDIGOS FONTES DO PROTÓTIPO

1. Pacote *Common*

br.com.felineLayer.common.Communication.java

```
1.    public final class Communication implements Runnable {
2.
3.        private DataOutputStream dataOutputStream = null;
4.
5.        protected synchronized Socket getSocket() {
6.            return communicationListener.getSocket();
7.        }
8.        private ICommunicationListener communicationListener;
9.
10.       public Communication( ICommunicationListener communicationListener ) {
11.           setCommunicationListener( communicationListener );
12.       }
13.
14.       public void disconnect() {
15.           try {
16.               getSocket().close();
17.           } catch ( IOException ex ) {
18.               Logger.getLogger( Communication.class.getName() ).log( Level.SEVERE, null, ex );
19.           }
20.       }
21.
22.       public void setCommunicationListener( ICommunicationListener communicationListener ) {
23.           try {
24.               this.communicationListener = communicationListener;
25.               this.dataOutputStream = new DataOutputStream( getSocket().getOutputStream() );
26.               System.out.println( "Communication: waiting message from other side" );
27.               Thread thread = new Thread( this );
28.               thread.start();
29.           } catch ( IOException ex ) {
30.               Logger.getLogger( Communication.class.getName() ).log( Level.SEVERE, null, ex );
31.           }
32.       }
33.       private int _inquireId = 0;
34.
35.       private synchronized int incInquireId() {
36.           int r = ++_inquireId;
37.           return r;
38.       }
39.       private Map<Integer, Object> waitingInquireLock = new HashMap<>();
40.
41.       private synchronized void putInquireLock( Integer id, Object th ) {
42.           waitingInquireLock.put( id, th );
43.       }
44.
45.       private synchronized Object getInquireLock( Integer id ) {
46.           return waitingInquireLock.get( id );
47.       }
48.       private Map<Integer, Message> waitingInquireResponse = new HashMap<>();
49.
50.       private synchronized Message getInquireResponse( Integer id ) {
51.           return waitingInquireResponse.get( id );
52.       }
53.
54.       private synchronized void putInquireResponse( Integer id, Message message ) {
55.           waitingInquireResponse.put( id, message );
56.       }
57.
58.       private synchronized void removeInquireResponse( Integer id ) {
59.           waitingInquireResponse.remove( id );
60.       }
61.       private HashMap<Thread, ArrayList<Message>> _batch = new HashMap<>();
62.
63.       private synchronized ArrayList<Message> batch() {
64.           ArrayList<Message> result = _batch.get( Thread.currentThread() );
65.           if ( result == null ) {
66.               result = new ArrayList<>();
67.               _batch.put( Thread.currentThread(), result );
68.           }
69.           return result;
70.       }
71.       private boolean inBatch = false;
72.
73.       public Communication beginBatch() {
74.           inBatch = true;
75.           return this;
76.       }
77.
78.       public boolean inBatch() {
79.           return inBatch;
80.       }
81.
82.       private Message getMessageFromBatch( String name, int componentId ) {
83.           for ( Message c : batch() ) {
84.               if ( c.getIntByName( Protocol.COMPONENT_ID ) != componentId )
```

```
85.                 continue;
86.             String n = c.getName();
87.             if ( n.equals( name ) )
88.                 return c;
89.         }
90.         return null;
91.     }
92.
93.     private Message getMessageFromBatch( String[] names, int componentId ) {
94.         for ( Message c : batch() ) {
95.             if ( c.getIntByName( Protocol.COMPONENT_ID ) != componentId )
96.                 continue;
97.             String[] ns = c.getNames();
98.             if ( Arrays.equals( ns, names ) )
99.                 return c;
100.        }
101.        return null;
102.    }
103.
104.    private synchronized Communication addBatch( Message message ) {
105.        ArrayList<Message> batch = batch();
106.        if ( message.getName().equals( Protocol.MSG_NAME_COMPONENT_PROPERTY_CHANGE ) ) {
107.            int componentId = message.getIntByName( Protocol.COMPONENT_ID );
108.            // Check if exist message in que to some component and property then rewrite
109.            Message m = getMessageFromBatch( message.getNames(), componentId );
110.            if ( m != null )
111.                batch.remove( m );
112.            else {
113.                // Check if exists message in queue to some component
114.                m = getMessageFromBatch( Protocol.MSG_NAME_CREATE_COMPONENT, componentId );
115.                if ( m == null )
116.                    m = getMessageFromBatch( Protocol.MSG_NAME_COMPONENT_PROPERTY_CHANGE,
117.                            componentId );
118.                // If found message with property of some component then acumulates
119.                if ( m != null ) {
120.                    m.insertValuesFrom( message );
121.                    return this;
122.                }
123.            }
124.        }
125.        batch.add( message );
126.        return this;
127.    }
128.
129.    public synchronized Communication sendMessage( Message message ) {
130.        if ( inBatch ) {
131.            addBatch( message );
132.            return this;
133.        }
134.        message = markTimeSent( message );
135.        sendMessage( message.toString() );
136.        return this;
137.    }
138.
139.    private Message markTimeSent( Message message ) {
140.        if ( message.isPropertyExists( Protocol.MSG_TIME_SENT ) )
141.            message.setValueByName( Protocol.MSG_TIME_SOURCE_SENT, message.getLongByName(
142.                    Protocol.MSG_TIME_SENT ) );
143.        message.setValueByName( Protocol.MSG_TIME_SENT, System.currentTimeMillis() );
144.        return message;
145.    }
146.
147.    public synchronized Communication endBatch() {
148.        ArrayList<Message> batch = batch();
149.        inBatch = false;
150.        if ( batch.size() > 0 ) {
151.            markTimeSent( batch.get( 0 ) );
152.            String fullString;
153.            if ( batch().size() == 1 )
154.                fullString = batch.get( 0 ).toString();
155.            else
156.                fullString = Message.arrayToMessage( batch ).toString();
157.            sendMessage( fullString );
158.            batch.clear();
159.        }
160.        return this;
161.    }
162.    private static final SimpleDateFormat formatter = new SimpleDateFormat( "hh:mm:ss:SSS" );
163.    private long last = 0;
164.
165.    public String now() {
166.        Date n = new Date();
167.        long t = n.getTime();
168.        long diff = 0;
169.        if ( last != 0 )
170.            diff = t - last;
171.        last = t;
172.        return formatter.format( n ) + " " + diff;
173.    }
174.
175.    public synchronized void sendMessage( String message ) {
176.        try {
177.            // Create output data stream
178.            if ( communicationListener.getSocket().isClosed() ) {
179.                System.out.println( "PROTOCOL: impossÃ-vel enviar comando, cliente fechado" );
180.                return;
181.            }
182.            // Print a line to output data stream
183.            FelLogger.log( "PROTOCOL " + now() + " Seed : " + message );
```

```
184.             writeText( dataOutputStream, message );
185.         } catch ( IOException ex ) {
186.             Logger.getLogger( Communication.class.getName() ).log( Level.SEVERE, null, ex );
187.         }
188.     }
189.
190.     public void finishMessageReceived( Message message ) {
191.         // If need send inquire reply (confirmation)
192.         if ( message.isPropertyExists( Protocol.INQUIRING_ID ) ) {
193.             int id = message.getIntByName( Protocol.INQUIRING_ID );
194.             message.deleteValue( Protocol.INQUIRING_ID );
195.             message.setValueByName( Protocol.INQUIRED_ID, id );
196.             sendMessage( message );
197.         }
198.     }
199.
200.     private String readText( DataInputStream dataInputStream ) throws IOException {
201.         return dataInputStream.readUTF();
202.     }
203.
204.     private void writeText( DataOutputStream dataOutputStream, String message ) throws IOException {
205.         dataOutputStream.writeUTF( message );
206.         dataOutputStream.flush();
207.     }
208.
209.     public Message inquire( Message message ) {
210.         if ( inBatch ) {
211.             addBatch( message );
212.             return message;
213.         }
214.         try {
215.             Inquire inq = createInquire();
216.             message.setValueByName( Protocol.INQUIRING_ID, inq.id.toString() );
217.             sendMessage( message );
218.             synchronized ( inq.getObject() ) {
219.                 inq.getObject().wait();
220.             }
221.             message = getInquireResponse( inq.getId() );
222.             removeInquireResponse( inq.getId() );
223.             return message;
224.         } catch ( InterruptedException ex ) {
225.             Logger.getLogger( Communication.class.getName() ).log( Level.SEVERE, null, ex );
226.             return null;
227.         }
228.     }
229.
230.     private Inquire createInquire() {
231.         Inquire inq = new Inquire();
232.         int inquireId = incInquireId();
233.         Object thread = new Object(); //Thread
234.         inq.setObject( thread ).setId( new Integer( inquireId ) );
235.         putInquireLock( inquireId, thread );
236.         return inq;
237.     }
238.
239.     class Inquire {
240.
241.         private Object thread;
242.         private Integer id;
243.
244.         public Inquire setId( Integer id ) {
245.             this.id = id;
246.             return this;
247.         }
248.
249.         public Integer getId() {
250.             return this.id;
251.         }
252.
253.         public Inquire setObject( Object object ) {
254.             this.thread = object;
255.             return this;
256.         }
257.
258.         public Object getObject() {
259.             return thread;
260.         }
261.     }
262.
263.     @Override
264.     public void run() {
265.         try {
266.             DataInputStream input = new DataInputStream( getSocket().getInputStream() );
267.             // Loop to listen from other side
268.             while ( true ) {
269.                 if ( ( input == null ) || ( getSocket() == null ) || ( getSocket().isClosed() ) ) {
270.                     System.out.println( "PROTOCOL: peer desconnected!" );
271.                     break;
272.                 }
273.                 String text = readText( input );
274.                 if ( ( text == null ) || text.isEmpty() )
275.                     continue;
276.                 // Convert text message do protocol object
277.                 Message message = new Message().setFromString( text );
278.
279.                 FelLogger.log( "PROTOCOL latency " + getLantency( message ) + " echo "
280.                         + getEchoLantency( message ) + " Received : " + text );
281.
282.                 // Start thread to fire event from message received
```

```
283.                new FireMessageReceived().fireMessageReceived(this, communicationListener, message);
284.            }
285.            if ( input != null )
286.                input.close();
287.            dataOutputStream.close();
288.        } catch ( Exception ex ) {
289.            String m = ex.getMessage();
290.            if ( ( ( m != null ) && ( m.indexOf( "socket closed" ) >= 0 ) ) || ( ( m != null )
291.                    && ( m.indexOf( "Connection reset" ) >= 0 ) )
292.                    || ( ex instanceof java.io.EOFException ) )
293.                System.out.println( "PROTOCOL: peer desconectou!" );
294.            else
295.                Logger.getLogger( Communication.class.getName() ).log( Level.SEVERE, null, ex );
296.        }
297.    }
298.
299.    private long getLantency( Message message ) {
300.        long diff = -1;
301.        if ( message.isPropertyExists( Protocol.MSG_TIME_SENT ) )
302.            diff = System.currentTimeMillis() - message.getLongByName(
303.                    Protocol.MSG_TIME_SENT );
304.        return diff;
305.    }
306.
307.    private long getEchoLantency( Message message ) {
308.        long diff = -1;
309.        if ( message.isPropertyExists( Protocol.MSG_TIME_SOURCE_SENT ) )
310.            diff = System.currentTimeMillis() - message.getLongByName(
311.                    Protocol.MSG_TIME_SOURCE_SENT );
312.        return diff;
313.    }
314.
315.    // run -> new FireMessageReceived
316.    class FireMessageReceived extends Thread {
317.
318.        private Communication communication;
319.        private ICommunicationListener communicationListener;
320.        private Message message;
321.
322.        public void fireMessageReceived( Communication communication,
323.                ICommunicationListener communicationListener, Message message ) {
324.            this.communication = communication;
325.            this.communicationListener = communicationListener;
326.            this.message = message;
327.            start();
328.        }
329.
330.        @Override
331.        public void run() {
332.            communicationListener.processMessageReceived( message );
333.            // If must wait inquire response
334.            communication.wakeUpInquiringThread( message );
335.        }
336.    }
337.
338.    /**
339.     * responseInquire
340.     *
341.     * @param message
342.     */
343.    private void wakeUpInquiringThread( Message message ) {
344.        if ( !message.isPropertyExists( Protocol.INQUIRED_ID ) )
345.            return;
346.        //
347.        String sId = message.getValueByName( Protocol.INQUIRED_ID );
348.        if ( sId.isEmpty() )
349.            return;
350.        Integer id = new Integer( sId );
351.        Object o = getInquireLock( id );
352.        if ( ( o != null ) && ( o instanceof Object ) ) {
353.            Object th = (Object) o; // Thread
354.            putInquireResponse( id, message );
355.            synchronized ( th ) {
356.                th.notify();
357.            }
358.        } else
359.            System.out.println( "thread " + id + " nÃ£o encontrada!" );
360.    }
361. }
```

br.com.felineLayer.common.ICommunicationListetner.java

```
1.   public interface ICommunicationListener {
2.
3.       Socket getSocket();
4.
5.       void processMessageReceived( Message message );
6.   }
```

br.com.felineLayer.common.Message.java

```
1.   public final class Message {
2.   
3.       private JSONObject jsonObj;
4.   
5.       public Message() {
6.           setFromString( "" );
7.       }
8.   
9.       public Message( String name ) {
10.          setFromString( "" ).setName( name );
11.      }
12.  
13.      public Message setFromString( String fullString ) {
14.          try {
15.              if ( fullString.isEmpty() )
16.                  jsonObj = new JSONObject();
17.              else
18.                  jsonObj = new JSONObject( fullString );
19.          } catch ( JSONException ex ) {
20.              Logger.getLogger( Message.class.getName() ).log( Level.SEVERE, null, ex );
21.          }
22.          return this;
23.      }
24.  
25.      public String getName() {
26.          return getValueByName( Protocol.MSG_NAME );
27.      }
28.  
29.      public Message setName( String name ) {
30.          setValueByName( Protocol.MSG_NAME, name );
31.          return this;
32.      }
33.  
34.      @Override
35.      public String toString() {
36.          return jsonObj.toString();
37.      }
38.  
39.      public long getLongByName( String name ) {
40.          return Long.parseLong( getValueByName( name ) );
41.      }
42.  
43.      public int getIntByName( String name ) {
44.          return Integer.parseInt( getValueByName( name ) );
45.      }
46.  
47.      public String getValueByName( String name ) {
48.          try {
49.              return jsonObj.getString( name );
50.          } catch ( JSONException ex ) {
51.              Logger.getLogger( Message.class.getName() ).log( Level.SEVERE, null, ex );
52.          }
53.          return "";
54.      }
55.  
56.      public Message setValueByName( String name, long value ) {
57.          return setValueByName( name, value + "" );
58.      }
59.  
60.      public Message setValueByName( String name, int value ) {
61.          return setValueByName( name, value + "" );
62.      }
63.  
64.      public Message setObjectByName( String name, Object value ) {
65.          try {
66.              jsonObj.put( name, value );
67.          } catch ( JSONException ex ) {
68.              Logger.getLogger( Message.class.getName() ).log( Level.SEVERE, null, ex );
69.          }
70.          return this;
71.      }
72.  
73.      public Message setValueByName( String name, String value ) {
74.          return setObjectByName( name, value );
75.      }
76.  
77.      public boolean isPropertyExists( String name ) {
78.          return !jsonObj.isNull( name );
79.      }
80.      private static String ARRAY = "array";
81.  
82.      public boolean isArray() {
83.          return isPropertyExists( ARRAY );
84.      }
85.  
86.      public Message deleteValue( String name ) {
87.          jsonObj.remove( name );
88.          return this;
89.      }
90.  
91.      public ArrayList<Message> getArray() {
92.          ArrayList<Message> result = new ArrayList<>();
93.          JSONObject j;
94.          Message c;
95.          try {
96.              JSONArray ja = jsonObj.getJSONArray( ARRAY );
97.              for ( int i = 0; i < ja.length(); i++ ) {
98.                  j = ja.getJSONObject( i );
99.                  c = new Message().setFromString( j.toString() );
```

```
100.                 result.add( c );
101.             }
102.         } catch ( JSONException ex ) {
103.             Logger.getLogger( Message.class.getName() ).log( Level.SEVERE, null, ex );
104.         }
105.         return result;
106.     }
107.
108.     public static Message arrayToMessage( ArrayList<Message> messages ) {
109.         JSONArray ja = new JSONArray();
110.         try {
111.             for ( Message c : messages )
112.                 ja.put( new JSONObject( c.toString() ) );
113.         } catch ( JSONException ex ) {
114.             Logger.getLogger( Message.class.getName() ).log( Level.SEVERE, null, ex );
115.         }
116.         return new Message( ARRAY ).setObjectByName( ARRAY, ja );
117.     }
118.
119.     public String[] getNames() {
120.         return JSONObject.getNames( jsonObj );
121.     }
122.
123.     public Message insertValuesFrom( Message message ) {
124.         for ( String n : message.getNames() ) {
125.             if ( n.equals( Protocol.MSG_NAME ) )
126.                 continue;
127.             setValueByName( n, message.getValueByName( n ) );
128.         }
129.         return this;
130.     }
131. }
```

br.com.felineLayer.common.Protocol.java

```
1.  public class Protocol {
2.      public static final String YES = "YES";
3.      public static final String NO = "NO";
4.      public static final String MSG_NAME = "MESSAGE_NAME";
5.      public static final String MSG_NAME_INQUIRE_APPLICATION_NAME = "INQUIRE_APPLICATION_NAME";
6.      public static final String APPLICATION_NAME = "APPLICATION_NAME";
7.      public static final String MSG_NAME_CREATE_COMPONENT = "CREATE_COMPONENT";
8.      public static final String MSG_NAME_COMPONENT_PROPERTY_CHANGE = "COMPONENT_PROPERTY_CHANGE";
9.      public static final String MSG_NAME_METHOD = "METHOD";
10.     public static final String MSG_NAME_DESTROY_COMPONENT = "DESTROY_COMPONENT";
11.     public static final String MSG_TIME_SENT = "MSG_TIME_SENT";
12.     public static final String MSG_TIME_SOURCE_SENT = "MSG_TIME_SOURCE_SENT";
13.     public static final String METHOD_NAME = "METHOD_NAME";
14.     public static final String METHOD_FRAME_SHOW = "FRAME_SHOW";
15.     public static final String METHOD_FRAME_CLOSE = "FRAME_CLOSE";
16.     public static final String PROPERTY_NAME_WIDTH = "NAME_WIDTH";
17.     public static final String PROPERTY_NAME_LEFT = "NAME_LEFT";
18.     public static final String PROPERTY_NAME_TOP = "NAME_TOP";
19.     public static final String PROPERTY_NAME_HEIGHT = "NAME_HEIGHT";
20.     public static final String PROPERTY_NAME_TEXT = "NAME_TEXT";
21.     public static final String COMPONENT_CLASS = "COMPONENT_CLASS";
22.     public static final String COMPONENT_FRAME = "FRAME";
23.     public static final String COMPONENT_BUTTON = "BUTTON";
24.     public static final String COMPONENT_COMBOBOX = "COMBOBOX";
25.     public static final String COMPONENT_EDIT = "EDIT";
26.     public static final String COMPONENT_LABEL = "LABEL";
27.     public static final String COMPONENT_ID = "ID";
28.     public static final String PARENT_FRAME_ID = "PARENT_FRAME_ID";
29.     public static final String INQUIRING_ID = "INQUIRING_ID";
30.     public static final String INQUIRED_ID = "INQUIRED_ID";
31.     public static final String MSG_NAME_SHOW_MESSAGE = "NAME_SHOW_MESSAGE";
32.     public static final String TEXT_SHOW_MESSAGE = "TEXT_SHOW_MESSAGE";
33.     public static final String MSG_NAME_SHOW_INPUT = "SHOW_INPUT";
34.     public static final String SHOW_INPUT_RESPONSE = "SHOW_INPUT_RESPONSE";
35.     public static final String MSG_NAME_EVENT = "EVENT";
36.     public static final String EVENT_NAME = "EVENT_NAME";
37.     public static final String EVENT_COMPONENT_ID = "EVENT_COMPONENT_ID";
38.     public static final String EVENT_FOCUS_EXIT = "EVENT_FOCUS_EXIT";
39.     public static final String EVENT_FOCUS_ENTER = "EVENT_FOCUS_ENTER";
40.     public static final String EVENT_CLICK = "EVENT_CLICK ";
41.     public static final String EVENT_DBL_CLICK = "EVENT_DBL_CLICK";
42.     public static final String EVENT_BEFORE_CLOSE = "EVENT_BEFORE_CLOSE";
43.     public static final String EVENT_BEFORE_CLOSE_CANCLOSE = "EVENT_BEFORE_CLOSE_CANCLOSE";
44.     public static final String EVENT_ADD_LISTENER = "EVENT_ADD_LISTENER";
45. }
```

2. Pacote *Client*

br.com.felineLayer.client.Client.java

```
1.  public abstract class Client implements ICommunicationListener {
```

```
2.
3.      private Socket socket;
4.      protected Communication communication = null;
5.      private String applicationName;
6.
7.      public Client() {
8.          String[] params = { "default" };
9.          initialize( params );
10.     }
11.
12.     public Client( String[] args ) {
13.         initialize( args );
14.     }
15.
16.     protected abstract IClientFactory doCreateClientFactory();
17.
18.     private void initialize( String[] args ) {
19.         clientFactory = doCreateClientFactory();
20.         String parameters = "";
21.         String ip = "localhost:7000";
22.         for ( int i = 0; i < args.length; i++ )
23.             if ( i == 0 )
24.                 ip = args[0];
25.             else {
26.                 if ( !parameters.isEmpty() )
27.                     parameters = parameters + "";
28.                 parameters = parameters + args[i];
29.             }
30.         if ( parameters.isEmpty() )
31.             parameters = "sample";
32.         applicationName = parameters;
33.         int port = 7000;
34.         String[] ips = ip.split( ":" );
35.         ip = ips[0];
36.         if ( ips.length > 1 )
37.             port = Integer.parseInt( ips[1] );
38.         try {
39.             // cria o socket com o recurso desejado na porta especificada
40.             socket = new Socket( ip, port );
41.         } catch ( Exception e ) {
42.             System.out.println( "Algum problema ocorreu ao criar ou enviar dados pelo socket.\n"
43.                     + e.getMessage() );
44.             return;
45.         }
46.         communication = new Communication( this );
47.     }
48.
49.     @Override
50.     public Socket getSocket() {
51.         return socket;
52.     }
53.
54.     public Message createEvent( IComponent component, String eventName ) {
55.         Message m = new Message( Protocol.MSG_NAME_EVENT );
56.         m.setValueByName( Protocol.EVENT_NAME, eventName );
57.         m.setValueByName( Protocol.EVENT_COMPONENT_ID, getIdByComponent( component ) );
58.         return m;
59.     }
60.
61.     public void sendEvent( IComponent component, String eventName ) {
62.         Message m = createEvent( component, eventName );
63.         sendMessage( m );
64.     }
65.
66.     public void sendMessage( String message ) {
67.         communication.sendMessage( message );
68.     }
69.
70.     public void sendMessage( Message message ) {
71.         communication.sendMessage( message );
72.     }
73.
74.     public Message inquireMessage( Message message ) {
75.         return communication.inquire( message );
76.     }
77.     private HashMap<Integer, IComponent> components = new HashMap<>();
78.
79.     public IComponent getComponentById( int id ) {
80.         return components.get( new Integer( id ) );
81.     }
82.
83.     private static Integer getKeyByValue( Map<Integer, IComponent> map, IComponent value ) {
84.         for ( Entry<Integer, IComponent> entry : map.entrySet() )
85.             if ( value.equals( entry.getValue() ) )
86.                 return entry.getKey();
87.         return null;
88.     }
89.
90.     public int getIdByComponent( IComponent component ) {
91.         Integer i = getKeyByValue( components, component );
92.         if ( i == null )
93.             return 0;
94.         else
95.             return i.intValue();
96.     }
97.
98.     @Override
99.     public synchronized void processMessageReceived( Message message ) {
100.        // Define runnable interface do process message received
```

```
101.        class RunThread implements Runnable {
102.
103.            public Communication communication;
104.            public Message message;
105.
106.            @Override
107.            public void run() {
108.                fireMessageReceived( message );
109.                communication.finishMessageReceived( message );
110.            }
111.        }
112.        RunThread rt = new RunThread();
113.        rt.message = message;
114.        rt.communication = communication;
115.        doProcessMessageReceived( rt );
116.    }
117.
118.    private IClientFactory clientFactory;
119.
120.    public void setClientFactory( IClientFactory clientFactory ) {
121.        this.clientFactory = clientFactory;
122.    }
123.
124.    public IClientFactory getClientFactory() {
125.        return clientFactory;
126.    }
127.
128.    protected abstract void doShowMessage( String value );
129.
130.    protected abstract String doShowInput( String value );
131.
132.    protected void doFireMessageReceived( Message message ) {
133.    }
134.
135.    public abstract void doProcessMessageReceived( Runnable runnable );
136.
137.    @SuppressWarnings( "element-type-mismatch" )
138.    private void removeComponent( IComponent cm ) {
139.        components.remove( cm );
140.    }
141.
142.    private void fireMessageReceived( Message _message ) {
143.        ArrayList<Message> _messages;
144.        if ( _message.isArray() )
145.            _messages = _message.getArray();
146.        else {
147.            _messages = new ArrayList<>();
148.            _messages.add( _message );
149.        }
150.        // Travel messages receiveds
151.        for ( Message m : _messages ) {
152.            String n = m.getName();
153.            // If question application name
154.            if ( n.equals( Protocol.MSG_NAME_INQUIRE_APPLICATION_NAME ) )
155.                m.setValueByName( Protocol.APPLICATION_NAME, applicationName );
156.            //
157.            if ( n.equals( Protocol.MSG_NAME_SHOW_MESSAGE ) )
158.                doShowMessage( m.getValueByName( Protocol.TEXT_SHOW_MESSAGE ) );
159.            //
160.            if ( n.equals( Protocol.MSG_NAME_SHOW_INPUT ) ) {
161.                // Get user input and put response at some message
162.                String text = doShowInput( m.getValueByName( Protocol.TEXT_SHOW_MESSAGE ) );
163.                m.setValueByName( Protocol.SHOW_INPUT_RESPONSE, text );
164.            }
165.            //
166.            if ( n.equals( Protocol.MSG_NAME_CREATE_COMPONENT ) )
167.                createComponent( m );
168.            // Get component object from id
169.            IComponent cm = null;
170.            if ( m.isPropertyExists( Protocol.COMPONENT_ID ) ) {
171.                cm = getComponentById( m.getIntByName( Protocol.COMPONENT_ID ) );
172.                if ( cm == null )
173.                    System.err.println( "ID NOT FOUNT Protocol.COMPONENT_ID " + m.getIntByName(
174.                            Protocol.COMPONENT_ID ) );
175.            }
176.            // If message has component
177.            if ( cm != null ) {
178.                // If is change property message (create or change)
179.                if ( n.equals( Protocol.MSG_NAME_CREATE_COMPONENT ) || n.equals(
180.                        Protocol.MSG_NAME_COMPONENT_PROPERTY_CHANGE ) )
181.                    // If found component then call changeProperty/method
182.                    cm.changeProperty( m );
183.                //
184.                cm.messageReceived( m );
185.                // If must destroy component
186.                if ( n.equals( Protocol.MSG_NAME_DESTROY_COMPONENT ) )
187.                    cm.destroy();
188.            }
189.            //
190.            doFireMessageReceived( m );
191.        }
192.    }
193.
194.    private void createComponent( Message message ) throws NumberFormatException {
195.        String cn = message.getValueByName( Protocol.COMPONENT_CLASS );
196.        int id = message.getIntByName( Protocol.COMPONENT_ID );
197.        // Inialine component
198.        IComponent cm = null;
199.        // Frame/window
```

```
200.         if ( cn.equals( Protocol.COMPONENT_FRAME ) )
201.             cm = clientFactory.createFrame();
202.         // Edit
203.         if ( cn.equals( Protocol.COMPONENT_EDIT ) )
204.             cm = clientFactory.createEdit();
205.         // ComboBox
206.         if ( cn.equals( Protocol.COMPONENT_COMBOBOX ) )
207.             cm = clientFactory.createComboBox();
208.         // Label
209.         if ( cn.equals( Protocol.COMPONENT_LABEL ) )
210.             cm = clientFactory.createLabel();
211.         // Button
212.         if ( cn.equals( Protocol.COMPONENT_BUTTON ) )
213.             cm = clientFactory.createButton();
214.         // If not created component
215.         if ( cm == null )
216.             return;
217.         // Define id component
218.         cm.setId( id );
219.         cm.setClient( this );
220.         // Add new component to component list
221.         components.put( new Integer( id ), cm );
222.         // If exists parent frame id add component to the this frame
223.         if ( message.isPropertyExists( Protocol.PARENT_FRAME_ID ) ) {
224.             String sFrameId = message.getValueByName( Protocol.PARENT_FRAME_ID );
225.             if ( !sFrameId.isEmpty() ) {
226.                 IComponent c = getComponentById( Integer.parseInt( sFrameId ) );
227.                 if ( c instanceof IFrame )
228.                     ( (IFrame) c ).addComponent( cm );
229.             }
230.         }
231. 
232.     }
233. 
234.     protected void destroyCallBack( IComponent component ) {
235.         removeComponent( component );
236.     }
```

br.com.felineLayer.client.Component.java

```
1.   Component<T extends IComponent> implements IComponent {
2.   
3.       @Override
4.       public final void changeProperty( Message message ) {
5.           if ( message.isPropertyExists( Protocol.PROPERTY_NAME_WIDTH ) )
6.               setWidht( message.getIntByName( Protocol.PROPERTY_NAME_WIDTH ) );
7.           if ( message.isPropertyExists( Protocol.PROPERTY_NAME_LEFT ) )
8.               setLeft( message.getIntByName( Protocol.PROPERTY_NAME_LEFT ) );
9.           if ( message.isPropertyExists( Protocol.PROPERTY_NAME_TOP ) )
10.              setTop( message.getIntByName( Protocol.PROPERTY_NAME_TOP ) );
11.          if ( message.isPropertyExists( Protocol.PROPERTY_NAME_HEIGHT ) )
12.              setHeight( message.getIntByName( Protocol.PROPERTY_NAME_HEIGHT ) );
13.          if ( message.isPropertyExists( Protocol.PROPERTY_NAME_TEXT ) )
14.              setText( message.getValueByName( Protocol.PROPERTY_NAME_TEXT ) );
15.          doChangeProperty( message );
16.      }
17.  
18.      protected void doChangeProperty( Message message ) {
19.  
20.      }
21.  
22.  
23.      private Client client;
24.  
25.      @Override
26.      public final void setClient( Client client ) {
27.          this.client = client;
28.      }
29.      public final Client getClient() {
30.          return client;
31.      }
32.  
33.      protected void doDestroy() {
34.  
35.      }
36.  
37.      @Override
38.      public final void destroy() {
39.          setVisible( false );
40.          doDestroy();
41.          client.destroyCallBack( this );
42.      }
43.  }
```

br.com.felineLayer.client.IComponent.java

```
1.   public interface IComponent<T extends IComponent> {
2.   
3.       public T setLeft( int left );
4.   
5.       public T setTop( int top );
```

```
6.
7.      public T setWidht( int width );
8.
9.      public T setHeight( int height );
10.
11.     public T setText( String text );
12.
13.     public T setVisible( boolean visible );
14.
15.     public String getText();
16.
17.     public T setId( int id );
18.
19.     public Object getObject();
20.
21.     public void changeProperty( Message message );
22.
23.     public void setClient( Client client );
24.
25.     public void messageReceived( Message message );
26.
27.     public void destroy();
28. }
```

br.com.felineLayer.client.IClientFactory.java

```
1.  public interface IClientFactory {
2.
3.      public IFrame createFrame();
4.
5.      public IButton createButton();
6.
7.      public IComboBox createComboBox();
8.
9.      public IEdit createEdit();
10.
11.     public ILabel createLabel();
12. }
13.
```

br.com.felineLayer.cliente.Component.java

```
1.  public abstract class Component<T extends IComponent> implements IComponent {
2.
3.      @Override
4.      public final void changeProperty( Message message ) {
5.          if ( message.isPropertyExists( Protocol.PROPERTY_NAME_WIDTH ) )
6.              setWidht( message.getIntByName( Protocol.PROPERTY_NAME_WIDTH ) );
7.          if ( message.isPropertyExists( Protocol.PROPERTY_NAME_LEFT ) )
8.              setLeft( message.getIntByName( Protocol.PROPERTY_NAME_LEFT ) );
9.          if ( message.isPropertyExists( Protocol.PROPERTY_NAME_TOP ) )
10.             setTop( message.getIntByName( Protocol.PROPERTY_NAME_TOP ) );
11.         if ( message.isPropertyExists( Protocol.PROPERTY_NAME_HEIGHT ) )
12.             setHeight( message.getIntByName( Protocol.PROPERTY_NAME_HEIGHT ) );
13.         if ( message.isPropertyExists( Protocol.PROPERTY_NAME_TEXT ) )
14.             setText( message.getValueByName( Protocol.PROPERTY_NAME_TEXT ) );
15.         doChangeProperty( message );
16.     }
17.
18.     protected void doChangeProperty( Message message ) {
19.
20.     }
21.
22.     private Client client;
23.
24.     @Override
25.     public final void setClient( Client client ) {
26.         this.client = client;
27.     }
28.     public final Client getClient() {
29.         return client;
30.     }
31.
32.     protected void doDestroy() {
33.
34.     }
35.
36.     @Override
37.     public final void destroy() {
38.         setVisible( false );
39.         doDestroy();
40.         client.destroyCallBack( this );
41.     }
42. }
```

br.com.felinelayer.client.IComponent.java

```
1.   public interface IComponent<T extends IComponent> {
2.
3.       public T setLeft( int left );
4.
5.       public T setTop( int top );
6.
7.       public T setWidht( int width );
8.
9.       public T setHeight( int height );
10.
11.      public T setText( String text );
12.
13.      public T setVisible( boolean visible );
14.
15.      public String getText();
16.
17.      public T setId( int id );
18.
19.      public Object getObject();
20.
21.      public void changeProperty( Message message );
22.
23.      public void setClient( Client client );
24.
25.      public void messageReceived( Message message );
26.
27.      public void destroy();
28.  }
```

br.com.felinelayer.client.clientSwing.ClientSwingFactor.java

```
1.   public class ClientSwingFactory implements IClientFactory {
2.
3.       @Override
4.       public br.com.felinelayer.client.IFrame createFrame() {
5.           return new br.com.felinelayer.client.clientSwing.Frame();
6.       }
7.
8.       @Override
9.       public Button createButton() {
10.          return new br.com.felinelayer.client.clientSwing.Button();
11.      }
12.
13.      @Override
14.      public ComboBox createComboBox() {
15.          return new br.com.felinelayer.client.clientSwing.ComboBox();
16.      }
17.
18.      @Override
19.      public Edit createEdit() {
20.          return new br.com.felinelayer.client.clientSwing.Edit();
21.      }
22.
23.      @Override
24.      public Label createLabel() {
25.          return new br.com.felinelayer.client.clientSwing.Label();
26.      }
27.  }
```

br.com.felinelayer.client.clientSwing.ClientSwing.java

```
1.   public class ClientSwing extends br.com.felinelayer.client.Client {
2.
3.       @SuppressWarnings( "ResultOfObjectAllocationIgnored" )
4.       public static void main( String[] args ) {
5.           new ClientSwing( args );
6.       }
7.
8.       private ClientSwing( String[] args ) {
9.           super( args );
10.          // Get the currently installed look and feel
11.          LookAndFeel lf = UIManager.getLookAndFeel();
12.          // Install a different look and feel; specifically, the Windows look and feel
13.          try {
14.              UIManager.setLookAndFeel( UIManager.getSystemLookAndFeelClassName() );
15.          } catch ( ClassNotFoundException | InstantiationException | IllegalAccessException |
16.                  UnsupportedLookAndFeelException e ) {
17.          }
18.      }
19.
20.      @Override
21.      public void doProcessMessageReceived( Runnable runnable ) {
22.          SwingUtilities.invokeLater( runnable );
23.      }
24.
```

```
25.     @Override
26.     protected void doShowMessage( String value ) {
27.         JOptionPane.showMessageDialog( null, value, "InformaÃ§Ã£o", JOptionPane.INFORMATION_MESSAGE );
28.     }
29.
30.     @Override
31.     protected String doShowInput( String value ) {
32.         return JOptionPane.showInputDialog( null, value, "Titulo", 1 );
33.     }
34.
35.     @Override
36.     protected IClientFactory doCreateClientFactory() {
37.         return new ClientSwingFactory();
38.     }
39. }
```

br.com.felinelayer.client.clientSwing.Component.java

```
1.  public abstract class Component<T extends br.com.felinelayer.client.IComponent>
2.          extends br.com.felinelayer.client.Component<T> implements FocusListener {
3.
4.      public static final int HEIGHT_DEFAULT = 20;
5.      public int id = 0;
6.      private Object object;
7.      private String previosText = "";
8.
9.      @Override
10.     public T setId( int id ) {
11.         this.id = id;
12.         return (T) this;
13.     }
14.
15.     @Override
16.     public T setVisible( boolean visible ) {
17.         if ( getObject() instanceof JComponent )
18.             ( (JComponent) getObject() ).setVisible( visible );
19.         if ( getObject() instanceof JFrame )
20.             ( (JFrame) getObject() ).setVisible( visible );
21.         return (T) this;
22.     }
23.
24.     @Override
25.     public T setWidht( int width ) {
26.         Dimension d = getDimension();
27.         d.width = width;
28.         setDimension( d );
29.         return (T) this;
30.     }
31.
32.     @Override
33.     public T setHeight( int height ) {
34.         Dimension d = getDimension();
35.         d.height = height;
36.         setDimension( d );
37.         return (T) this;
38.     }
39.
40.     @Override
41.     public abstract T setText( String text );
42.
43.     @Override
44.     public abstract String getText();
45.
46.     @Override
47.     public Object getObject() {
48.         return object;
49.     }
50.
51.     public T setObject( Object object ) {
52.         this.object = object;
53.         if ( object instanceof JComponent )
54.             ( (JComponent) getObject() ).addFocusListener( this );
55.         return (T) this;
56.     }
57.
58.     public Rectangle getBounds() {
59.         if ( getObject() instanceof JComponent )
60.             return ( (JComponent) getObject() ).getBounds();
61.         if ( getObject() instanceof JFrame )
62.             return ( (JFrame) getObject() ).getBounds();
63.         return null;
64.     }
65.
66.     public T setBounds( Rectangle rectangle ) {
67.         if ( getObject() instanceof JComponent )
68.             ( (JComponent) getObject() ).setBounds( rectangle );
69.         if ( getObject() instanceof JFrame )
70.             ( (JFrame) getObject() ).setBounds( rectangle );
71.         return (T) this;
72.     }
73.
74.     public Dimension getDimension() {
75.         if ( getObject() instanceof JComponent )
76.             return ( (JComponent) getObject() ).getSize();
```

```
77.         if ( getObject() instanceof JFrame )
78.             return ( (JFrame) getObject() ).getSize();
79.         return null;
80.     }
81.
82.     public T setDimension( Dimension rectangle ) {
83.         if ( getObject() instanceof JComponent )
84.             ( (JComponent) getObject() ).setSize( rectangle );
85.         if ( getObject() instanceof JFrame )
86.             ( (JFrame) getObject() ).setSize( rectangle );
87.         return (T) this;
88.     }
89.
90.     public int getHeight() {
91.         return (int) getBounds().getHeight();
92.     }
93.
94.     @Override
95.     public T setTop( int top ) {
96.         Rectangle b = getBounds();
97.         b.y = top;
98.         setBounds( b );
99.         return (T) this;
100.    }
101.
102.    @Override
103.    public T setLeft( int left ) {
104.        Rectangle b = getBounds();
105.        b.x = left;
106.        setBounds( b );
107.        return (T) this;
108.    }
109.
110.    protected Message createEvent( String name ) {
111.        return getClient().createEvent( this, name );
112.    }
113.
114.    protected void fireEvent( String name ) {
115.        getClient().sendEvent( this, name );
116.    }
117.
118.    protected T fireEvent( Message message ) {
119.        getClient().sendMessage( message );
120.        return (T) this;
121.    }
122.
123.    protected Message fireInquireEvent( Message message ) {
124.        return getClient().inquireMessage( message );
125.    }
126.
127.    @Override
128.    protected void doChangeProperty( Message message ) {
129.    }
130.
131.    @Override
132.    public void messageReceived( Message message ) {
133.    }
134.
135.    @Override
136.    public void focusGained( FocusEvent e ) {
137.        previosText = getText();
138.        fireEvent( Protocol.EVENT_FOCUS_ENTER );
139.    }
140.
141.    @Override
142.    public void focusLost( FocusEvent e ) {
143.        Message m = createEvent( Protocol.EVENT_FOCUS_EXIT );
144.        String currentText = getText();
145.        if ( !previosText.equalsIgnoreCase( getText() ) )
146.            m.setValueByName( Protocol.PROPERTY_NAME_TEXT, currentText );
147.        fireEvent( m );
148.    }
149.
150.    @Override
151.    protected void doDestroy() {
152.        if ( object instanceof JComponent )
153.            ( (JComponent) getObject() ).getParent().remove( (JComponent) getObject() );
154.    }
155. }
```

br.com.felinelayer.client.clientSwing.Frame.java

```
1.  public class Frame extends ComponentSwing implements br.com.felinelayer.client.IFrame, WindowListener {
2.
3.      private JFrame frame;
4.      private final Container contentPane;
5.
6.      @SuppressWarnings( "LeakingThisInConstructor" )
7.      public Frame() {
8.          // Create the frame
9.          String title = "Frame Title";
10.         frame = new JFrame( title );
11.         // Show the frame
12.         int width = 600;
```

```
13.         int height = 300;
14.         frame.addWindowListener( this );
15.         contentPane = frame.getContentPane();
16.         contentPane.setLayout( null );
17.         frame.setDefaultCloseOperation( JFrame.DO_NOTHING_ON_CLOSE );
18.         frame.setSize( width, height );
19.         frame.setVisible( false );
20.     }
21.
22.     @Override
23.     public void addComponent( br.com.felinelayer.client.IComponent component ) {
24.         if ( component.getObject() instanceof java.awt.Component ) {
25.             java.awt.Component o = (java.awt.Component) component.getObject();
26.             contentPane.add( o );
27.         }
28.     }
29.
30.     @Override
31.     public Object getObject() {
32.         return frame;
33.     }
34.
35.     @Override
36.     public Frame setText( String text ) {
37.         frame.setTitle( text );
38.         return this;
39.     }
40.
41.     @Override
42.     public void close() {
43.         frame.setVisible( false );
44.     }
45.
46.     @Override
47.     public void windowClosing( WindowEvent e ) {
48.         Runnable runner = new Runnable() {
49.
50.             @Override
51.             public void run() {
52.                 Message message = createEvent( Protocol.EVENT_BEFORE_CLOSE );
53.                 message.setValueByName( Protocol.EVENT_BEFORE_CLOSE_CANCLOSE, Protocol.YES );
54.                 message = fireInquireEvent( message );
55.                 if ( message.getValueByName( Protocol.EVENT_BEFORE_CLOSE_CANCLOSE ).equals(
56.                         Protocol.YES ) ) {
57.                     frame.setVisible( false );
58.                     System.exit( 0 );
59.                 }
60.             }
61.         };
62.         new Thread( runner ).start();
63.     }
64.
65.     @Override
66.     public void windowOpened( WindowEvent e ) {
67.         //throw new UnsupportedOperationException( "Not supported yet." );
68.     }
69.
70.     @Override
71.     public void windowClosed( WindowEvent e ) {
72.         //throw new UnsupportedOperationException( "Not supported yet." );
73.     }
74.
75.     @Override
76.     public void windowIconified( WindowEvent e ) {
77.         //throw new UnsupportedOperationException( "Not supported yet." );
78.     }
79.
80.     @Override
81.     public void windowDeiconified( WindowEvent e ) {
82.         //throw new UnsupportedOperationException( "Not supported yet." );
83.     }
84.
85.     @Override
86.     public void windowActivated( WindowEvent e ) {
87.         //throw new UnsupportedOperationException( "Not supported yet." );
88.     }
89.
90.     @Override
91.     public void windowDeactivated( WindowEvent e ) {
92.         //throw new UnsupportedOperationException( "Not supported yet." );
93.     }
94.
95.     @Override
96.     public String getText() {
97.         return frame.getTitle();
98.     }
99.
100.    @Override
101.    public void messageReceived( Message message ) {
102.        if (message.getName().equalsIgnoreCase( Protocol.MSG_NAME_METHOD) ) {
103.            String method = message.getValueByName( Protocol.METHOD_NAME);
104.            if (method.equalsIgnoreCase( Protocol.METHOD_FRAME_SHOW ) ) {
105.                frame.setVisible( true );
106.            }
107.            if (method.equalsIgnoreCase( Protocol.METHOD_FRAME_CLOSE ) ) {
108.                frame.setVisible( false );
109.            }
110.        }
111.    }
```

```
112.
113. }
```

br.com.felinelayer.client.clientSwing.Edit.java

```
1.    public class Edit extends ComponentSwing implements br.com.felinelayer.client.IEdit {
2.
3.        public Edit() {
4.            jTextField = new JTextField();
5.            Dimension s = jTextField.getSize();
6.            s.height = HEIGHT_DEFAULT;
7.            s.width = 24;
8.            jTextField.setSize( s );
9.            setObject( jTextField );
10.       }
11.       private JTextField jTextField;
12.
13.       @Override
14.       public Edit setText( String text ) {
15.           jTextField.setText( text );
16.           return this;
17.       }
18.
19.       @Override
20.       public String getText() {
21.           return jTextField.getText();
22.       }
23.   }
```

br.com.felinelayer.client.clientSwing.Edit.java

```
1.    public class Button extends ComponentSwing implements ActionListener, br.com.felinelayer.client.IButton {
2.
3.        @SuppressWarnings( "LeakingThisInConstructor" )
4.        public Button() {
5.            jButton = new JButton();
6.            Dimension s = jButton.getSize();
7.            s.height = HEIGHT_DEFAULT;
8.            s.width = 80;
9.            jButton.setSize( s );
10.           jButton.addActionListener( this );
11.           setObject( jButton );
12.       }
13.       private JButton jButton;
14.
15.
16.       @Override
17.       public Button setText( String text ) {
18.           jButton.setText( text );
19.           return this;
20.       }
21.
22.       @Override
23.       public void actionPerformed( ActionEvent e ) {
24.           fireEvent(Protocol.EVENT_CLICK);
25.       }
26.
27.       @Override
28.       public String getText() {
29.           return jButton.getText();
30.       }
31.   }
```

br.com.felinelayer.client.clientSwing.Label.java

```
1.    public class Label extends ComponentSwing implements br.com.felinelayer.client.ILabel {
2.
3.        public Label() {
4.            jLabel = new JLabel();
5.            Dimension s = jLabel.getSize();
6.            s.height = HEIGHT_DEFAULT;
7.            s.width = 100;
8.            jLabel.setSize( s );
9.            setObject( jLabel );
10.       }
11.       private JLabel jLabel;
12.
13.       @Override
14.       public Label setText( String text ) {
15.           jLabel.setText( text );
16.           return this;
17.       }
```

```
18.
19.     @Override
20.     public String getText() {
21.         return jLabel.getText();
22.     }
23. }
```

3. Pacote *Server*

br.com.felinelayer.server.IApplicationFactory.java

```
1.  interface IApplicationFactory {
2.      public Application createApplicationByName(String applicationName);
3.  }
```

br.com.felinelayer.server.ApplicationLauncher.java

```
1.  public class ApplicationLauncher extends Thread {
2.  
3.      private final Client client;
4.      private final IApplicationFactory aplicationFactory;
5.  
6.      public ApplicationLauncher( IApplicationFactory aplicationFactory, Client client ) {
7.          this.aplicationFactory = aplicationFactory;
8.          this.client = client;
9.      }
10. 
11.     @Override
12.     public void run() {
13.         // Assumes that not found application
14.         Application application = null;
15.         // Inquire application name
16.         Message message = new Message( Protocol.MSG_NAME_INQUIRE_APPLICATION_NAME );
17.         Message response = client.getCommunication().inquire( message );
18.         String applicationName = response.getValueByName( Protocol.APPLICATION_NAME );
19.         // Create application from informed name
20.         application = aplicationFactory.createApplicationByName( applicationName );
21.         // If creates application then defines client object
22.         if ( application != null ) {
23.             application.setClient( client );
24.             client.setApplication( application );
25.             // Start application
26.             application.run();
27.         }
28.     }
29. }
```

br.com.felinelayer.server.Server.java

```
1.  public class Server implements Runnable, IApplicationFactory {
2.  
3.      private static ServerSocket serv;
4.  
5.      @SuppressWarnings( "ResultOfObjectAllocationIgnored" )
6.      public static void main( String[] args ) {
7.          new Server();
8.      }
9.      private int port;
10. 
11.     @SuppressWarnings( "CallToThreadStartDuringObjectConstruction" )
12.     public Server() {
13.         try {
14.             // Create ServerSocket at port 7000
15.             port = 7000;
16.             new Thread( this ).start();
17.         } catch ( Exception ex ) {
18.             Logger.getLogger( Server.class.getName() ).log( Level.SEVERE, null, ex );
19.         }
20.     }
21. 
22.     @Override
23.     public void run() {
24.         if (serv == null)
25.             try {
26.             serv = new ServerSocket( port );
27.         } catch ( IOException ex ) {
28.             Logger.getLogger( Server.class.getName() ).log( Level.SEVERE, null, ex );
29.             return;
30.         }
31.         //
32.         System.out.println( "SERVER: waiting clients connection at port " + port );
33.         try {
```

```
34.             // Loop to wait new connections
35.             while ( true ) {
36.                 // Wait new connections
37.                 Socket socket = serv.accept();
38.                 // When client conects then runs statement
39.                 System.out.println( "SERVER: client connected!" );
40.                 // Create client
41.                 Client client = new Client( socket );
42.                 // Execute application thread
43.                 new ApplicationLauncher( this, client ).start();
44.             }
45.         } catch ( IOException ex ) {
46.             Logger.getLogger( Server.class.getName() ).log( Level.SEVERE, null, ex );
47.         }
48.     }
49.
50.     @Override
51.     public synchronized Application createApplicationByName( String applicationName ) {
52.         try {
53.             Object o = getClass().getClassLoader().loadClass( applicationName ).newInstance();
54.             if ( o instanceof Application )
55.                 return (Application) o;
56.         } catch ( InstantiationException | IllegalAccessException | ClassNotFoundException ex ) {
57.             ex.printStackTrace();
58.         }
59.         return null;
60.     }
61. }
```

br.com.felinelayer.server.Client.java

```
1.  public final class Client implements ICommunicationListener {
2.
3.      public Client( Socket socket ) {
4.          setSocket(socket);
5.      }
6.
7.      private Socket socket;
8.      private Communication communication;
9.
10.     public void setSocket( Socket socket ) {
11.         this.socket = socket;
12.         if ( this.communication == null )
13.             setCommunication( new Communication( this ) );
14.     }
15.
16.     public Communication getCommunication() {
17.         return communication;
18.     }
19.
20.     public void setCommunication( Communication communication ) {
21.         this.communication = communication;
22.     }
23.     private Application application;
24.
25.     public void setApplication( Application application ) {
26.         this.application = application;
27.         application.setClient( this );
28.         Message c = new Message( "ConnectingApplicationName" );
29.         c.setValueByName( "ApplicationName", application.getApplicationName() );
30.         processMessageReceived( c );
31.     }
32.
33.     @Override
34.     public Socket getSocket() {
35.         return socket;
36.     }
37.
38.     public void disconnect() {
39.         communication.disconnect();
40.     }
41.
42.     public void sendMessage( Message message ) {
43.         communication.sendMessage( message );
44.     }
45.
46.     public Client beginBatch() {
47.         communication.beginBatch();
48.         return this;
49.     }
50.
51.     public boolean inBacth() {
52.         return communication.inBatch();
53.     }
54.
55.     public Client endBatch() {
56.         communication.endBatch();
57.         return this;
58.     }
59.
60.     public Message inquire( Message message ) {
61.         return communication.inquire( message );
62.     }
63.
```

```
64.      @Override
65.      public void processMessageReceived( Message message ) {
66.          if ( application != null )
67.              application.messageReceived( message );
68.          communication.finishMessageReceived( message );
69.      }
70.  }
```

br.com.felinelayer.server.application.Application.java

```
1.   public abstract class Application<T extends Application> implements Runnable {
2.   
3.       private Client client;
4.       private Frame currentFrame;
5.       private String applicationName = "";
6.   
7.       public T setApplicationName( String applicationName ) {
8.           this.applicationName = applicationName;
9.           return (T) this;
10.      }
11.  
12.      public String getApplicationName() {
13.          return applicationName;
14.      }
15.  
16.      public T setClient( Client client ) {
17.          this.client = client;
18.          return (T) this;
19.      }
20.  
21.      protected T showMessage( String message ) {
22.          boolean wasInBatch = unloadBatch();
23.          Message c = new Message( Protocol.MSG_NAME_SHOW_MESSAGE );
24.          c.setValueByName( Protocol.TEXT_SHOW_MESSAGE, message );
25.          inquire( c );
26.          if (wasInBatch)
27.              beginBatch();
28.          return (T) this;
29.      }
30.  
31.      protected String showInput( String message ) {
32.          boolean wasInBatch = unloadBatch();
33.          Message c = new Message( Protocol.MSG_NAME_SHOW_INPUT );
34.          c.setValueByName( Protocol.TEXT_SHOW_MESSAGE, message );
35.          c = inquire( c );
36.          String response = "";
37.          if ( c.isPropertyExists( Protocol.SHOW_INPUT_RESPONSE ) )
38.              response = c.getValueByName( Protocol.SHOW_INPUT_RESPONSE );
39.          if (wasInBatch)
40.              beginBatch();
41.          return response;
42.      }
43.  
44.      public T beginBatch() {
45.          client.beginBatch();
46.          return (T) this;
47.      }
48.  
49.      public T endBatch() {
50.          client.endBatch();
51.          return (T) this;
52.      }
53.  
54.      protected T sendMessage( Message message ) {
55.          client.sendMessage( message );
56.          return (T) this;
57.      }
58.  
59.      protected Message inquire( Message message ) {
60.          return client.inquire( message );
61.      }
62.  
63.      protected boolean inBacth() {
64.          return client.inBacth();
65.      }
66.  
67.      public boolean unloadBatch() {
68.          boolean wasBatching = inBacth();
69.          if (wasBatching)
70.              endBatch();
71.          return wasBatching;
72.      }
73.  
74.      public void messageReceived( Message message ) {
75.          beginBatch();
76.          doMessageReceived( message );
77.          FelLogger.Log( message.toString() );
78.          // If is event command
79.          if ( message.getName().equalsIgnoreCase( Protocol.MSG_NAME_EVENT ) ) {
80.              String event = message.getValueByName( Protocol.EVENT_NAME );
81.              int id = message.getIntByName( Protocol.EVENT_COMPONENT_ID );
82.              Component c = getComponentById( id );
83.              //
```

```
84.             if ( message.isPropertyExists( Protocol.PROPERTY_NAME_TEXT ) ) {
85.                 c.setUpdateFromOtherSide( true );
86.                 c.setText( message.getValueByName( Protocol.PROPERTY_NAME_TEXT ) );
87.                 c.setUpdateFromOtherSide( false );
88.             }
89.             // Fire event managment
90.             doEventReceived( c, event );
91.             c.messageReceived(message);
92.         }
93.         endBatch();
94.     }
95.     private int componentId = 0;
96.     private HashMap<Integer, Component> components = new HashMap<>();
97.
98.     public Component getComponentById( int id ) {
99.         return components.get( new Integer( id ) );
100.    }
101.
102.    @SuppressWarnings( "element-type-mismatch" )
103.    public Application<T> removeComponent(Component component) {
104.        components.remove( component );
105.        return this;
106.    }
107.
108.    private <T extends Component> T addComponent( T component ) {
109.        component.application = this;
110.        components.put( new Integer( ++componentId ), component );
111.        component.setId( componentId );
112.        Message c = new Message( Protocol.MSG_NAME_CREATE_COMPONENT );
113.        if ( component instanceof Frame )
114.            c.setValueByName( Protocol.COMPONENT_CLASS, Protocol.COMPONENT_FRAME );
115.        if ( component instanceof Button )
116.            c.setValueByName( Protocol.COMPONENT_CLASS, Protocol.COMPONENT_BUTTON );
117.        if ( component instanceof ComboBox )
118.            c.setValueByName( Protocol.COMPONENT_CLASS, Protocol.COMPONENT_COMBOBOX );
119.        if ( component instanceof Edit )
120.            c.setValueByName( Protocol.COMPONENT_CLASS, Protocol.COMPONENT_EDIT );
121.        if ( component instanceof Label )
122.            c.setValueByName( Protocol.COMPONENT_CLASS, Protocol.COMPONENT_LABEL );
123.        c.setValueByName( Protocol.COMPONENT_ID, componentId );
124.        if ( !( component instanceof Frame ) && ( currentFrame != null ) )
125.            c.setValueByName( Protocol.PARENT_FRAME_ID, currentFrame.getId() );
126.        // Send message and wait response
127.        client.inquire( c );
128.        return component;
129.    }
130.
131.    public Frame addFrame() {
132.        Frame c = new Frame();
133.        currentFrame = c;
134.        return addComponent( c );
135.    }
136.
137.    public Button addButton() {
138.        return addComponent( new Button() );
139.    }
140.
141.    public ComboBox addComboBox() {
142.        return addComponent( new ComboBox() );
143.    }
144.
145.    public Edit addEdit() {
146.        return addComponent( new Edit() );
147.    }
148.
149.    public Label addLabel() {
150.        return addComponent( new Label() );
151.    }
152.    protected boolean stopAuto = true;
153.
154.    @Override
155.    public void run() {
156.        beginBatch();
157.        doStart();
158.        endBatch();
159.        if ( stopAuto )
160.            stop();
161.    }
162.
163.    protected abstract void doStart();
164.
165.    protected abstract void doEventReceived( Component sender, String event );
166.
167.    protected abstract void doMessageReceived( Message message );
168.
169.    protected void stop() {
170.        client.disconnect();
171.    }
172. }
```

br.com.felinelayer.server.application.IEventListener.java

```
1.    public interface IEventListener<T extends Component> {
2.
3.        void onEvent( Component sender, String eventName, Message message );
4.    }
```

br.com.felinelayer.server.application.Component.java

```
1.    public abstract class Component<T extends Component> {
2.
3.        public Application application;
4.        private int left = 0;
5.        private int top = 0;
6.        private int width = 0;
7.        private int height = 0;
8.        private int id = 0;
9.        private String text = "";
10.       private boolean updateFromOtherSide = false;
11.
12.       public boolean isUpdateFromOtherSide() {
13.           return updateFromOtherSide;
14.       }
15.
16.       public T setUpdateFromOtherSide( boolean updateFromOtherSide ) {
17.           this.updateFromOtherSide = updateFromOtherSide;
18.           return (T) this;
19.       }
20.
21.       public T setText( String text ) {
22.           this.text = text;
23.           if ( !updateFromOtherSide )
24.               firePropertyChange( Protocol.PROPERTY_NAME_TEXT, text );
25.           return (T) this;
26.       }
27.
28.       public String getText() {
29.           return text;
30.       }
31.       private Frame frame;
32.
33.       public void setFrame( Frame frame ) {
34.           this.frame = frame;
35.       }
36.       private static final int PIXELS_PER_ROW = 24;
37.       private static final int PIXELS_PER_COL = 24;
38.
39.       public T setRow( int row ) {
40.           setTop( ( row + 1 ) * PIXELS_PER_ROW );
41.           return (T) this;
42.       }
43.
44.       public T setCol( int col ) {
45.           setLeft( ( col + 1 ) * PIXELS_PER_COL );
46.           return (T) this;
47.       }
48.
49.       public int getLeft() {
50.           return left;
51.       }
52.
53.       public T setLeft( int left ) {
54.           this.left = left;
55.           firePropertyChange( Protocol.PROPERTY_NAME_LEFT, left + "" );
56.           return (T) this;
57.       }
58.
59.       public int getTop() {
60.           return top;
61.       }
62.
63.       public T setTop( int top ) {
64.           this.top = top;
65.           firePropertyChange( Protocol.PROPERTY_NAME_TOP, top + "" );
66.           return (T) this;
67.       }
68.
69.       public int getWidth() {
70.           return width;
71.       }
72.
73.       public T setWidth( int width ) {
74.           this.width = width;
75.           firePropertyChange( Protocol.PROPERTY_NAME_WIDTH, width + "" );
76.           return (T) this;
77.       }
78.
79.       public int getHeight() {
80.           return height;
81.       }
82.
83.       public T setHeight( int height ) {
84.           this.height = height;
85.           firePropertyChange( Protocol.PROPERTY_NAME_HEIGHT, height + "" );
86.           return (T) this;
87.       }
88.
```

```
89.     public int getId() {
90.         return id;
91.     }
92.
93.     protected Message inquireMessage( Message message ) {
94.         message.setValueByName( Protocol.COMPONENT_ID, getId() );
95.         return application.inquire( message );
96.     }
97.
98.     protected T sendMessage( Message message ) {
99.         message.setValueByName( Protocol.COMPONENT_ID, getId() );
100.        application.sendMessage( message );
101.        return (T) this;
102.    }
103.
104.    protected void firePropertyChange( String propertyName, int value ) {
105.        firePropertyChange( propertyName, value + "" ) ;
106.    }
107.
108.    protected void firePropertyChange( String propertyName, String value ) {
109.        Message message = new Message( Protocol.MSG_NAME_COMPONENT_PROPERTY_CHANGE );
110.        message.setValueByName( Protocol.COMPONENT_ID, getId() );
111.        message.setValueByName( propertyName, value );
112.        inquireMessage( message );
113.    }
114.
115.    public T setId( int id ) {
116.        this.id = id;
117.        return (T) this;
118.    }
119.    private HashMap<String, ArrayList<IEventListener>> eventListeners = new HashMap<>();
120.
121.    public T addEventListener( String eventName, IEventListener eventListener ) {
122.        ArrayList<IEventListener> l = eventListeners.get( eventName );
123.        if ( l == null )
124.            l = new ArrayList<>();
125.        if ( l.indexOf( l ) < 0 )
126.            l.add( eventListener );
127.        eventListeners.put( eventName, l );
128.        Message message = new Message( Protocol.EVENT_ADD_LISTENER );
129.        message.setValueByName( Protocol.EVENT_NAME, Protocol.EVENT_CLICK );
130.        sendMessage( message );
131.        return (T) this;
132.    }
133.
134.    public void messageReceived( Message message ) {
135.        if ( !message.isPropertyExists( Protocol.EVENT_NAME ) )
136.            return;
137.        String eventName = message.getValueByName( Protocol.EVENT_NAME );
138.        ArrayList<IEventListener> l = eventListeners.get( eventName );
139.        if ( l == null )
140.            return;
141.        for ( IEventListener e : l )
142.            e.onEvent( this, eventName, message );
143.    }
144.
145.    public void clearListeners() {
146.        eventListeners.clear();
147.    }
148.
149.    public void destroy() {
150.        Message m = new Message().setName( Protocol.MSG_NAME_DESTROY_COMPONENT );
151.        sendMessage( m );
152.    }
153. }
```

br.com.felinelayer.server.application.Frame.java

```
1.   public class Frame extends Component<Frame> {
2.
3.       public Frame setCurrentRow( int currentRow ) {
4.           this.currentRow = currentRow;
5.           return this;
6.       }
7.       private int currentRow = 0;
8.       private int currentCol = 0;
9.
10.      public int getCurrentRow() {
11.          return currentRow;
12.      }
13.
14.      public Frame setCurrentCol( int currentCol ) {
15.          this.currentCol = currentCol;
16.          return this;
17.      }
18.
19.      public int getCurrentCol() {
20.          return currentCol;
21.      }
22.
23.      public Frame addRow() {
24.          ++currentRow;
25.          currentCol = 0;
26.          return this;
```

```
27.         }
28.
29.         private ArrayList<Component> components = new ArrayList<>();
30.
31.         public <T extends Component> T addComponent( T component ) {
32.             components.add( component );
33.             component.setFrame( this );
34.             component.setRow( currentRow );
35.             component.setCol( currentCol );
36.             return component;
37.         }
38.
39.         public Button addButton() {
40.             return addComponent( application.addButton() );
41.         }
42.
43.         public ComboBox addComboBox() {
44.             return addComponent( application.addComboBox() );
45.         }
46.
47.         public Edit addEdit() {
48.             return addComponent( application.addEdit() );
49.         }
50.
51.         public Label addLabel() {
52.             return addComponent( application.addLabel() );
53.         }
54.
55.         public void show() {
56.             Message m = new Message( Protocol.MSG_NAME_METHOD );
57.             m.setValueByName( Protocol.METHOD_NAME, Protocol.METHOD_FRAME_SHOW );
58.             inquireMessage( m );
59.         }
60.
61.         public void close() {
62.             for (Component c : components) {
63.                 c.clearListeners();
64.             }
65.             Message m = new Message( Protocol.MSG_NAME_METHOD );
66.             m.setValueByName( Protocol.METHOD_NAME, Protocol.METHOD_FRAME_CLOSE );
67.             inquireMessage( m );
68.         }
69.     }
```