UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

SISTEMA DISTRIBUÍDO PARA GERENCIAMENTO DE LIBERAÇÃO E DISTRIBUIÇÃO DE RELEASES DE SOFTWARE

ROGÉRIO MELLO VANTI

BLUMENAU
2013

2013/2-20

ROGÉRIO MELLO VANTI

SISTEMA DISTRIBUÍDO PARA GERENCIAMENTO DE LIBERAÇÃO E DISTRIBUÍÇÃO DE RELEASES DE SOFTWARE

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Paulo Fernando da Silva, Mestre–Orientador

BLUMENAU
2013

2013/2-20

SISTEMA DISTRIBUÍDO PARA GERENCIAMENTO DELIBERAÇÃO E DISTRIBUÍÇÃO DE RELEASES DE SOFTWARE

Por

ROGÉRIO MELLO VANTI

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Paulo Fernando da Silva, Mestre – Orientador, FURB
Membro:	Prof. Mauro Marcelo Mattos, Doutor – FURB
Membro:	Prof. Jacques Robert Heckman, Mestre – FURB

Blumenau, 05 de dezembro de 2013

Dedico este trabalho a todos os familiares e amigos, especialmente aqueles que me ajudaram diretamente na realização deste.

AGRADECIMENTOS

À minha família, que mesmo longe, sempre esteve presente.

Aos meus amigos, pelos empurrões e cobranças.

Ao professor Mauro Marcelo Mattos pelas considerações, sugestões e contribuições.

Ao professor Marcel Hugo pelas contribuições, sugestões e considerações.

Ao meu orientador, Paulo Fernando da Silva, por ter acreditado na conclusão deste trabalho.

A força não provém da capacidade corporal, mas da vontade férrea.

Gandhi

RESUMO

O presente trabalho apresenta o desenvolvimento de um protótipo de sistema distribuído para gerenciamento de liberação e distribuição de releases de software. A solução está baseada em um *middleware* de comunicação de objetos distribuídos. Durante a fase de testes foi acessado por um número razoável de clientes, demonstrando sua capacidade de escalabilidade.

Palavras-chave: RMI. Sistemas distribuídos. Gerência de configuração.

ABSTRACT

This paper presents the development of a prototype system for distributed release management and distribution of software releases. The solution is based on a communication middleware. Distributed object during the test phase was accessed by a reasonable number of clients, demonstrating its ability to scale.

Key-words: RMI. Distributed systems. Configuration manager.

LISTA DE ILUSTRAÇÕES

Figura 1- Atividade da gerência de configuração 16
Figura 2 - Gerenciamento de mudanças 18
Figura 3 - Processo de manutenção 19
Figura 4 - Funcionamento do RMI 25
Figura 5 - Criptografia simétrica 27
Figura 6 - Criptografia assimétrica 28
Figura 7 – Diagrama de casos de uso da aplicação servidora 33
Figura 8 - Diagrama de casos de uso da aplicação cliente 38
Figura 9 - Diagrama de pacotes. 43
Figura 10 - Diagrama de classes da interface remota da aplicação cliente 44
Figura 11- Diagrama de classe da interface remota da aplicação servidora 45
Figura 12 - Diagrama de classe da aplicação cliente 47
Figura 13 - Diagrama de classe da aplicação servidora 49
Figura 14 - Tela de cadastro de dados da aplicação 54
Figura 15 - Tela aplicação cliente 55
Figura 16 - Tela de login 56
Figura 17 - Tela principal do servidor 56
Figura 18 - Tela de liberação de versão 57
Figura 19 - Dados da liberação 58
Figura 20 - Operação de baixar versão 59
Figura 21 - Realização da operação de baixar versão no cliente 60
Figura 22 - Término da operação no servidor 61
Figura 23 - Enviando o comando de instalação 61
Figura 24 - Resultado da operação de instalação 62
Figura 25 - Resultado da operação de voltar versão 63

LISTA DE QUADROS

Quadro 1 - Requisitos funcionais da aplicação servidora. 31
Quadro 2 - Requisitos não funcionais da aplicação servidora. 31
Quadro 3 - Requisitos funcionais da aplicação cliente. 32
Quadro 4 - Requisitos não funcionais da aplicação cliente. 32
Quadro 5 - Caso de uso UC01S 34
Quadro 6 - Caso de uso UC02S 34
Quadro 7 - Caso de uso UC03S 35
Quadro 8 - Caso de uso UC04S 35
Quadro 9 - Caso de uso UC05S 36
Quadro 10 - Caso de uso UC06S 36
Quadro 11 – Caso de uso UC07S 37
Quadro 12 - Caso de uso UC01C 38
Quadro 13 - Caso de uso UC02C 39
Quadro 14 - Caso de uso UC03C 39
Quadro 15 - Caso de uso UC04C 40
Quadro 16 - Caso de uso UC05C 41
Quadro 17 - Caso de uso UC06C 41
Quadro 18 - Caso de uso UC07 42
Quadro 19 - Caso de uso UC08C 42
Quadro 20 - Método servidor que realiza conexão com os clientes 51
Quadro 21 - Método que escreve o conjunto de chaves criptografadas 52
Quadro 22 - Identificação de queda do servidor na classe `NovaTelaCliente` 53
Quadro 23 - Campos do arquivo `Propriedades.conf` 54
Quadro 24 - Quadro comparativo 63

LISTA DE SIGLAS

ADSL – *Asymmetric Digital Subscriber Line*

API – *Application Programming Interface*

CORBA – *Common Object Request Broker Architecture*

FTP – *File Transfer Protocol*

IDE – *Integrated Development Enviroment*

IP – *Internet Protocol*

JRE – *Java Runtime Enviroment*

JNLP – *Java Network Launch Protocol*

MAN – *Metropolitan Area Network*

RF – Requisito Funcional

RMI – *Remote MehodInvocation*

RNF – Requisito Não Funcional

RSA – *Rivest, Shamir e Adleman*

SO – Sistema Operacional

SSL – *Secure Sockets Layer*

UML – *Unified Modeling Language*

VPN – *Virtual Private Network*

SUMÁRIO

1 INTRODUÇÃO 13
1.1 OBJETIVOS DO TRABALHO 13
1.2 ESTRUTURA DO TRABALHO 14
2 FUNDAMENTAÇÃO TEÓRICA 15
2.1 GERÊNCIA DE CONFIGURAÇÃO 15
2.1.1 Manutenção 17
2.1.2 Processo de *release* 19
2.1.3 Liberação e distribuição de *release* 20
2.2 SISTEMAS DISTRIBUÍDOS 21
2.2.1 Modelo de objetos distribuídos 23
2.2.2 *Remote Method Invocation* 23
2.3 SEGURANÇA DA INFORMAÇÃO 25
2.3.1 Criptografia 26
2.4 TRABALHOS CORRELATOS 29
2.4.1 uBuild 29
2.4.2 uDeploy 29
2.4.3 Java Web Start 30
3 DESENVOLVIMENTO 31
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO 31
3.2 ESPECIFICAÇÃO 32
3.2.1 Diagrama de casos de uso da aplicação servidora 33
3.2.1.1 Liberar versão 33
3.2.1.2 Manter usuário 34
3.2.1.3 Gerar conjunto de chaves 35
3.2.1.4 Exportar chaves de segurança 35
3.2.1.5 Exportar versão para *pendrive* 35
3.2.1.6 Realizar operações 36
3.2.1.7 Identificar clientes conectados e desconectados 37
3.2.2 Diagrama de casos de uso da aplicação cliente 37
3.2.2.1 Realizar operação de baixar versão 38
3.2.2.2 Realizar operação de instalar 39

3.2.2.3 Realizar operação de reinstalar 39
3.2.2.4 Realizar operação de instalar do *pendrive* 40
3.2.2.5 Realizar operação de voltar versão 40
3.2.2.6 Solicitar chave de segurança 41
3.2.2.7 Realizar reconexão em caso de queda do servidor 41
3.2.2.8 Gerar notificações na bandeja 42
3.2.3 Diagramas de pacotes 42
3.2.4 Diagrama de classes da interface cliente 43
3.2.5 Diagrama de classes da interface servidora 44
3.2.6 Diagrama de classes da aplicação cliente 46
3.2.7 Diagrama de classes da aplicação servidora 48
3.3 IMPLEMENTAÇÃO 50
3.3.1 Técnicas e ferramentas utilizadas 51
3.3.2 Operacionalidade da implementação 53
3.3.2.1 Aplicação cliente 53
3.3.2.2 Aplicação servidora 55
3.4 RESULTADOS E DISCUSSÃO 62
4 CONCLUSÕES 64
4.1 EXTENSÕES 65
REFERÊNCIAS BIBLIOGRÁFICAS 66

1 INTRODUÇÃO

O gerenciamento de *releases* é um item da área da engenharia de *software* chamado de gerenciamento de configuração. O gerenciamento de configuração tem por objetivo manter a integridade e controle sobre os artefatos de *software* envolvidos na construção de um novo *release*. Segundo Hirama (2012, p. 123), a importância de se ter controle destes artefatos se deve à complexidade dos sistemas. Além disso, ele permite saber quais os impactos em um projeto quando uma mudança é solicitada.

Hirama (2012, p. 126) afirma que as atividades envolvidas na gerência de configuração compreendem as seguintes etapas: identificar, controlar, relatar *status*, avaliar configuração, gerenciar a liberação e distribuição. Nesta última etapa enquadram-se a forma de versionamento, a construção do *release* e a forma como o mesmo chega ao usuário final. Existem vários contratempos como *deploy*, instalação, adequação de base de dados, até mesmo a infra-estrutura. Estes fatores podem inviabilizar o processo por completo dependendo da sua complexidade e/ou qualidade.

Publicar um *software web* pode depender de um conjunto de outras ferramentas como *File Transfer Protocol* (FTP) ou até mesmo ferramentas específicas de publicação. Depende também do meio físico de comunicação. Em um ambiente adequado, operando com a aplicação executando em um servidor centralizado e com *links* de comunicação razoáveis, este processo de distribuição é relativamente trabalhoso. Em um ambiente com vários servidores de aplicação a complexidade de realizar um *deploy* aumenta consideravelmente.

Neste cenário, o presente trabalho descreve o desenvolvimento de um *software* distribuído para o gerenciamento de *releases*, para atuar com ênfase nos processos de liberação e distribuição de *releases* de *software*.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é disponibilizar um sistema distribuído para gerenciar *releases* de *software* com ênfase nos processos de liberação e distribuição em um ambiente com vários servidores de aplicação.

Os objetivos específicos do trabalho são:

a) manter o histórico de liberações;
b) ser escalável para atender uma quantidade crescente de clientes;
c) ser heterogêneo para abstrair os processos de comunicação e infra-estrutura;
d) ter requisitos de segurança para operações locais nos clientes, tais como instalar, atualização por *pendrive* e voltar para uma versão anterior;
e) ter características de tolerância a falhas para permitir que o cliente possa realizar uma reconexão em caso de queda do servidor;
f) atuar ativamente no processo de instalação possibilitando a execução de *scripts* remotamente.

1.2 ESTRUTURA DO TRABALHO

Apresentada a introdução e os objetivos, no capítulo dois será apresentado a fundamentação teórica e as tecnologias envolvidas na realização deste trabalho. Nele serão destacados tópicos relacionados à gerência de configuração, arquitetura de sistemas distribuídos e apresentados os trabalhos correlatos.

No capítulo três será apresentado o desenvolvimento do trabalho. São descritos os requisitos que devem atendidos, especificação, técnicas, implementação, além da sua operacionalidade. Ainda serão apresentados os resultados obtidos na conclusão do trabalho.

No capítulo quatro será apresentado as conclusões decorrentes da realização do trabalho, assim como as suas possíveis extensões.

2 FUNDAMENTAÇÃO TEÓRICA

A seção 2.1 abordará o contexto de gerência de configuração. Na seção 2.2 serão abordados os conceitos e técnicas sobre sistemas distribuídos. A seção 2.3 abordará alguns conceitos sobre segurança da informação. Por fim, na seção 2.4 serão vistos alguns trabalhos correlatos.

2.1 GERÊNCIA DE CONFIGURAÇÃO

Gerência de configuração é uma área de conhecimento dentro da engenharia de *software* que estuda meios de manter os artefatos de *software* que são gerados durante o processo de desenvolvimento, assim como qualquer modificação que venha a ocorrer nestes artefatos. Segundo Wazlawick (2013, p. 219), esta área é especialmente importante devido às frequentes mudanças que ocorrem em um produto de *software* durante o projeto, podendo até mesmo ocorrer mudanças após o projeto.

Segundo Sommerville (2011, p. 476), a gerência de configuração é formada por quatro etapas macros, sendo elas: gerenciamento de mudanças, gerenciamento de versões, construção do sistema e gerenciamento de *release*. Cada etapa possui seu próprio conjunto de artefatos que devem ser mantidos durante o processo.

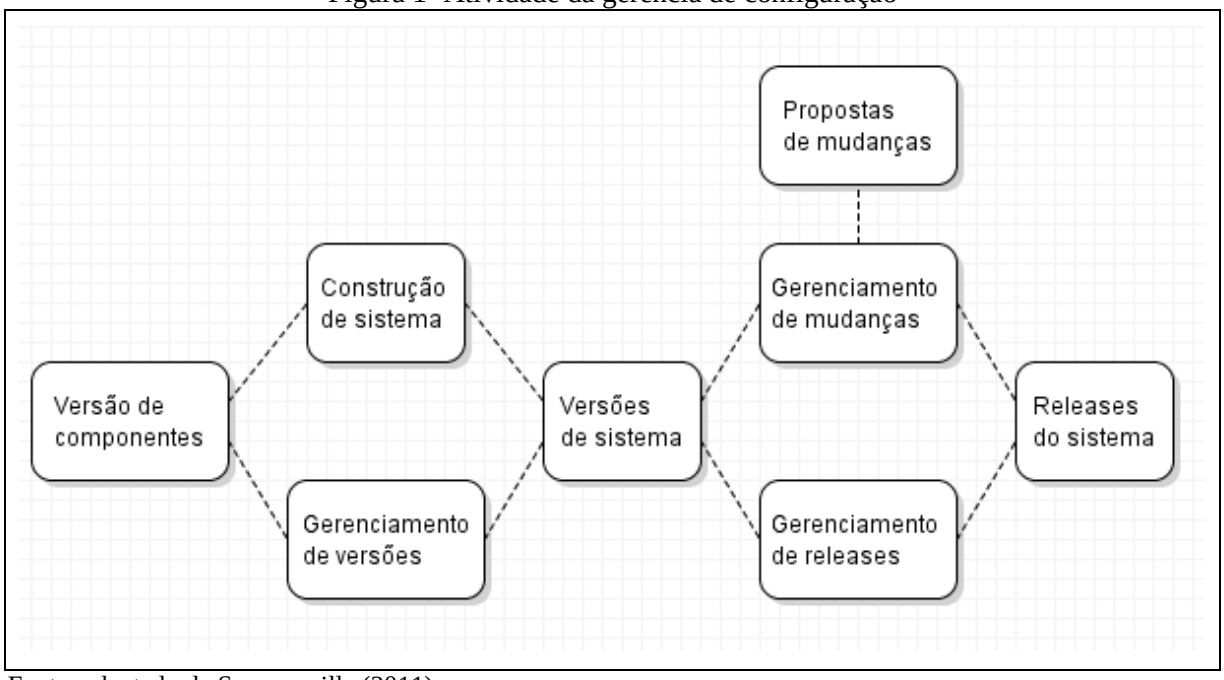
Para tanto, os artefatos são mapeados por itens de configuração, onde um item representa um artefato, ou até mesmo um conjunto de artefatos. Segundo Nunes e Falbo (2006, p. 223) este processo tem início com a confecção de um plano de gerência de configuração, identificando os itens que serão geridos. Afirma Engholm Júnior (2010, p. 237), que um item de configuração é um elemento criado ou necessário para o desenvolvimento. Wazlawick (2013, p. 220) afirma que um item de configuração é um elemento unitário para efeitos de controle de versão, ou um agregado de elementos que são tratados como uma única entidade.

Para que se possa atingir este nível de controle sobre os artefatos é recomendado que exista um grupo de controle de configuração, sendo estes responsáveis por analisar as solicitações de mudanças nos itens de uma determinada configuração. Segundo Engholm Júnior (2010, p. 237), as atividades deste grupo envolvem os seguintes processos: identificar,

revisar, controlar, recuperar e disponibilizar versões de itens de configuração, além de controlar itens de fornecedores. Hirama (2012, p. 126) afirma que o processo não limita-se às etapas de controle e versionamento, agregando uma nova etapa que inclui a liberação e distribuição de versões de *releases*, afirmando que a esta etapa deve ser devidamente controlada, assim como o restante do processo.

A figura 1 exemplifica de modo macro as atividades envolvidas na gerência de configuração.

Figura 1- Atividade da gerência de configuração



Fonte: adaptado de Sommerville (2011).

A etapa de `versões de componentes` mantem a integridade das partes menores que compõem uma versão, tais como *frameworks*, bibliotecas e outros artefatos que possam ser necessários.

A etapa de `construção de sistema` envolve o processo de compilar a aplicação para gerar uma versão estável.

A etapa de `gerenciamento de versões` envolve versionar as diferentes versões construidas permitindo que elas sejam indentificadas de forma inequívoca.

A etapa de `propostas de mudança` gera as solicitações de mudança que servirão como artefatos de entrada para a etapa de gerenciamento de mudanças.

A etapa de gerenciamento de mudanças realiza a avaliação das solicitações que são pertinentes deferindo ou indeferindo as alterações propostas.

A etapa de `gerenciamento de releases` envolve vervionar as diferente *releases* permitindo que elas sejam idenificadas de forma inequívoca.

A etapa de `releases do sistema` visa os processos de liberação e gestão das rleases.

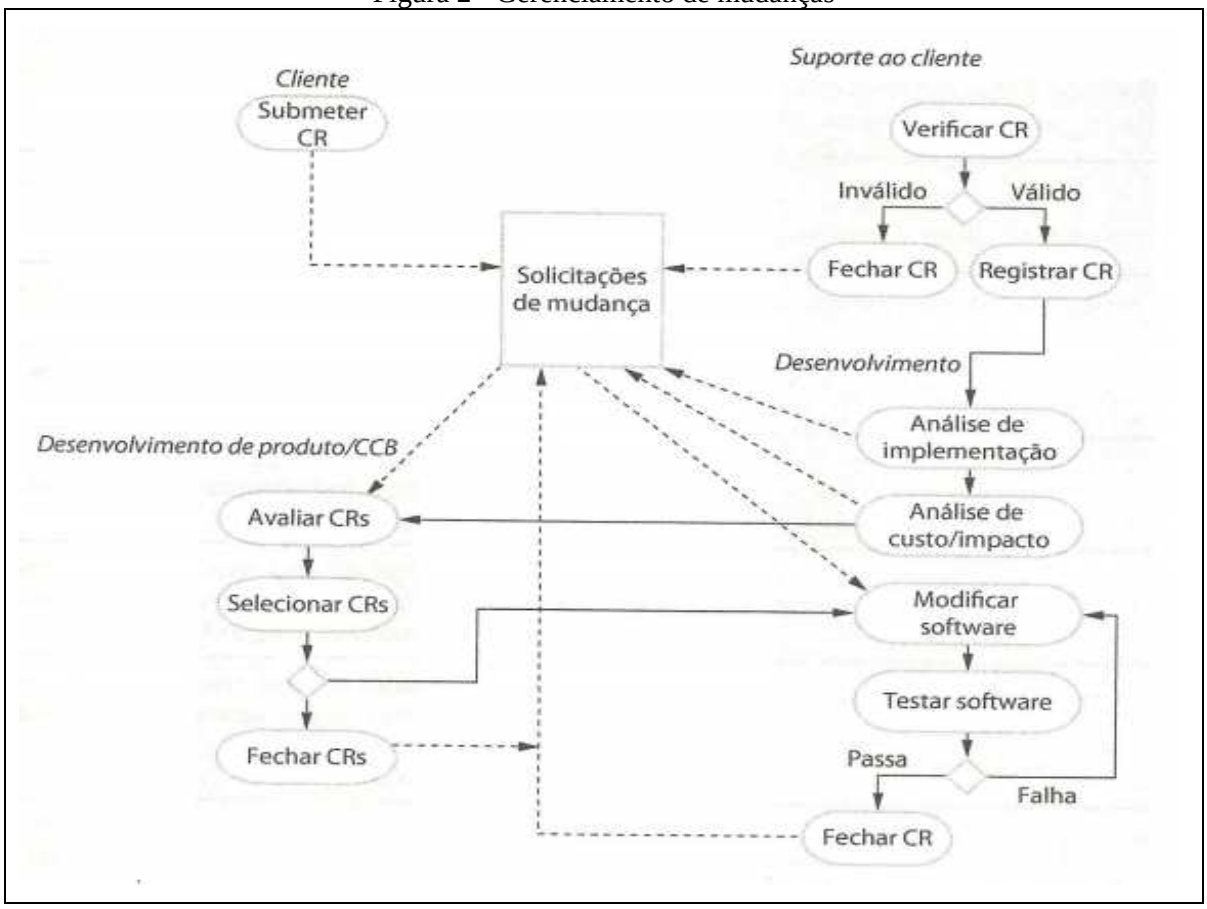
2.1.1 Manutenção

Este processo é uma etapa do ciclo de vida do produto de *software* que muitas vezes possui como entradas as novas necessidades dos clientes e demandas de mercado. Esta etapa do ciclo de vida é gerida pela gerência de configuração, mais especificamente no processo de gerenciamento de mudanças. Segundo Sommerville (2011, p. 476), gerir as mudanças envolve o acompanhamento das solicitações realizadas por clientes e desenvolvedores, definindo os seus impactos, assim como o momento em que será implementado.

A fase de manutenção do *software* ocorre a partir do momento em que o mesmo está implantado no ambiente de produção, ou seja, a partir do momento em que se tem um *deploy* bem sucedido. Afirma Sommerville (2011, p. 478) que este processo tem início quando o cliente solicita uma mudança, sendo que esta pode ser decorrente de um *bug* ou da agregação de uma nova funcionalidade a versão já existente.

Como pode ser visto na figura 2, uma solicitação de mudança não precisa vir apenas do cliente. O setor de suporte e o próprio desenvolvimento podem solicitar mudanças no produto.

Figura 2 - Gerenciamento de mudanças



Fonte: adaptado de Sommerville (2011).

Assim a manutenção pode ter diferentes motivos para ser solicitada e/ou realizada, existindo desta forma uma categorização dos tipos de manutenção. Podendo estas serem atribuídas a uma das seguintes categorias:

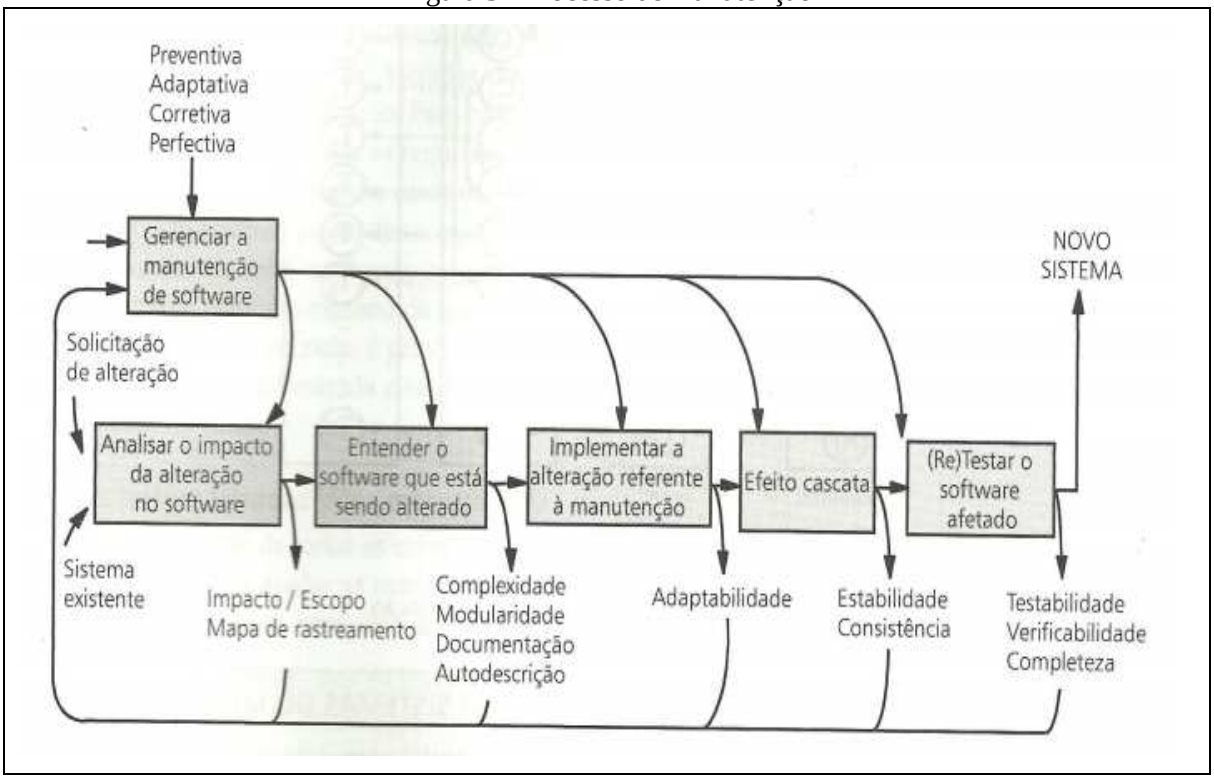
a) manutenção corretiva: visa corrigir defeitos e/ou inconsistência que o produto de *software* venha a apresentar. Segundo Pfleeger (2004, p. 386) este tipo de manutenção trata os problemas ocorridos em função de falhas, gerando solicitações de mudanças que devem ser mantidas, tendo como respostas providências rápidas a fim de garantir o funcionamento do produto;
b) manutenção adaptativa: visa implementar algum tipo de reformulação em alguma funcionalidade do produto. Segundo Pfleeger (2004, p. 386), este tipo de manutenção é necessária quando é preciso realizar correções secundárias devido à alguma alteração realizada no produto ou mudanças de *hardware*;
c) manutenção perfectiva: visa melhorar o modo como alguma funcionalidade opera, mesmo que a forma atual não gere inconsistências. Segundo Pfleeger (2004, p.

386), este tipo de manutenção visa realizar mudanças para melhorar alguns aspectos do produto, mesmo que não seja em função da ocorrência de uma falha;

d) manutenção preventiva: semelhante à manutenção perfectiva, porém visa prevenir a ocorrência de falhas. Segundo Pfleeger (2004, p. 386), este tipo de manutenção visa realizar melhoramentos a fim de impedir que falhas ocorram.

A figura 3 exemplifica o modo como são organizadas as etapas envolvidas no processo de manutenção. Como pode ser visto, o resultado de uma manutenção é uma nova versão do produto de *software*.

Figura 3 - Processo de manutenção



Fonte: adaptado de Pfleeger (2006)

Como pode ser visto na figura 3, o processo de manutenção inicia com a análise de impacto das alterações solicitadas que compreende as tarefas de entender o que será alterado e onde que estas alterações irão ter impacto. Após a análise segue-se para a implementação das solicitações que foram analisadas. Por fim o teste do *software* que visa averiguar a estabilidade, consistência e completude do *software*.

2.1.2 Processo de *release*

Release é o produto final entregue pelo processo de desenvolvimento criado a partir de uma versão do produto de *software*. Sempre será a aplicação em um estado consistente e

estável. Segundo Sommerville (2003, p. 557), *release* é uma versão do sistema que é distribuída para os clientes, onde esta deve possuir correções e/ou disponibilizar novas funcionalidades, ou ainda ser uma *release* idêntica, porém destinada a uma nova plataforma de *hardware*.

Da mesma forma que os itens de configuração, as *releases* são versionadas para identificar qual versão deu origem a ela, e também identificar de forma não ambígua as várias *releases* de um mesmo produto de *software*. Segundo Wazlawick (2013, p. 222), o controle de versão é necessário para que seja viável rastrear as alterações realizadas de uma *release* para outra, além de facilitar o controle quando for necessário ter duas versões diferentes em execução devido a limitações de *hardware* ou outras características.

Existem algumas maneiras de realizar este versionamento, variando de uma simples numeração a um agregado de características próprias de cada *release* que formam a sua identificação. Segundo Tsui e Karam (2013, p. 172), este controle é necessário para viabilizar um controle apropriado sobre os *releases*. Para tanto, é preciso ser capaz de identificar de forma inequívoca cada item e associá-los a itens correlatos. Segundo Sommerville (2003, p. 557), existem três formas possíveis de realizar este versionamento, sendo eles: usando uma numeração que o identifique de forma única; utilizando atributos onde o identificador será uma combinação de um conjunto de atributos associados ou usar a identificação orientada por mudanças que é semelhante à identificação por atributos, contudo está associada a uma ou mais solicitações de mudança.

2.1.3 Liberação e distribuição de *release*

Gerenciar liberações de software, assim como sua distribuição são processos delicados, pois lidam diretamente com o cliente. Ele pode apresentar resistência à mudança de um *release* para outro se o processo não for consistente o suficiente ou confiável. Segundo Sommerville (2003, p. 560), a preparação e distribuição de *releases* é um processo dispendioso, onde a frequência de liberações influi diretamente na satisfação do cliente e na qualidade do produto, assim como uma má gerência dos *releases* ou mesmo a falta de gerência pode influenciar no grau de satisfação, sendo que no pior dos casos, vir a liberar *releases* falhos ou inconsistentes pode afetar diretamente a qualidade do produto.

Segundo Magalhães (2007, p. 70), o gerenciamento de liberação é responsável por implementar as mudanças no ambiente de produção, ou seja, colocar em produção novos itens de configuração que sofreram alterações e gerenciar as atividades relacionadas a ela. Este tipo

de prática remove parte da complexidade de se efetuar alterações no ambiente de produção, além de diminuir a resistência à mudança por parte do cliente.

Este processo ainda sofre influência de fatores externos ao processo de desenvolvimento, como *links* de comunicações ou outros meios de distribuição do *release*. O processo de distribuição de *releases* de *software* ainda é difícil, principalmente em ambientes com vários servidores. Quando se consegue atuar em um ambiente favorável e próximo do ideal (*links* de comunicação apropriados) para a implantação de um sistema *web*, a atualização de *software* é um processo manual e demorado. Se considerado um ambiente de *Metropolitan Area Network* (MAN), esta atividade pode tornar-se extremamente complexa e eventualmente inviabilizar o processo.

2.2 SISTEMAS DISTRIBUÍDOS

Um sistema distribuído é um componente de *software* presente em um ou mais elementos interligados por meio de uma rede ou meio de comunicação que se apresenta como um único componente para o usuário. Coulouris e Kindberg (2005, p. 1) afirmam que a principal motivação para o desenvolvimento de sistemas distribuídos é o compartilhamento de recursos. Estes recursos podem ser gerenciados por servidores e acessados por clientes, ou ainda, eles podem ser encapsulados em objetos que podem ser acessados por objetos clientes.

O desenvolvimento de sistemas distribuídos torna-se complexo por envolver particularidades próprias, que muitas vezes são chamadas de desafios, sendo elas:

a) heterogeneidade: pode ser referente a diversos elementos como rede, *hardware*, sistemas operacionais e linguagens de programação. Este desafio geralmente é contornado com a utilização de *middlewares*. Segundo Dantas (2005, p. 127), “[...] um *middleware* é um pacote de *software* que possui suas próprias interfaces de programação que auxiliam desenvolvedores de aplicação a não se preocuparem com características específicas do sistema operacional”;
b) abertura: determina a capacidade do sistema de ser estendido ou ampliado, agregando novas funções e serviços aos já existentes. Segundo Coulouris e Kindberg (2005, p. 17), “A abertura de sistemas distribuídos é determinada principalmente pelo grau em que novos serviços de compartilhamento de recursos podem ser adicionados e ser disponibilizados para uso por uma variedade de

programas cliente” (tradução nossa). Esta característica é muito importante para realização de integrações, visto que os serviços estão disponíveis. Basta para a outra aplicação conhecer a interface de comunicação para poder consumir os serviços disponibilizados;

c) transparência: se refere ao modo como o conjunto de aplicações deve se apresentar para o usuário; ele deve mostrar-se como uma única aplicação, operando de forma transparente para o usuário. O usuário não conhece e nem precisa conhecer como o conjunto de aplicações se comunica para realizar uma determinada tarefa. Segundo Coulouris e Kindberg (2005, p. 23), “transparência é definida como a ocultação da separação dos componentes de um sistema distribuído [...], de modo que o sistema é visto como um todo, em vez de ser visto como um conjunto de componentes independentes.” (Tradução nossa);
d) tolerância a falhas: se refere à capacidade do sistema de superar condições de quebra, falhas internas e/ou falha de algum recurso compartilhado. Segundo Coulouris e Kindberg (2005, p. 21), “falhas em um sistema distribuído são parciais, ou seja, se algum componente falhar, os outros continuam funcionando, por consequência, o tratamento de falhas é particularmente difícil” (tradução nossa);
e) escalabilidade: referente à capacidade do sistema de atender as demandas dos clientes conectados, gerenciando os recursos disponibilizados. Coulouris e Kindberg (2005, p. 19) afirmam que a escalabilidade é a capacidade do sistema de permanecer eficaz quando existe um aumento significativo do número de usuários ou no número de recursos compartilhados;
f) segurança: muitos sistemas operam com informações associadas ao usuário e somente a ele, por isso a grande relevância em garantir a segurança destas informações. Coulouris e Kindberg (2005, p. 18) afirmam que a segurança é muito importante em sistemas de informação pois os mesmos trabalham com informações intrínsecas ao usuário e que estas devem estar em conformidade com os três elementos da segurança, sendo estes a confidencialidade, integridade e disponibilidade.

2.2.1 Modelo de objetos distribuídos

O modelo de objetos distribuídos consiste no uso dos conceitos de orientação objetos ao projeto de sistemas distribuídos. Segundo Guedes (2007, p. 2), esta tecnologia vem a agregar aos sistemas distribuídos as vantagens advindas da programação orientada a objetos.

Este modelo atua sobre o modelo tradicional de cliente/servidor diferenciando apenas no modo como trocam informações, estes geralmente são estruturados na forma de objetos, permitindo que aplicações diferentes troquem objetos entre elas. Este modelo pode ser consolidado através da utilização de *middlewares*, que atuam como intermediários no processo de comunicação. Estes podem centralizar os serviços ou ainda podem descentralizá-los. Neste último caso, tornando o modelo semelhante ao *Peer-to-Peer* (P2P), modelo no qual se admite a troca dos papéis em momentos variados.

Existem alguns *middlewares* que trabalham com este paradigma de objetos distribuídos, sendo os mais conhecidos: *Common Object Request Broker Architecture* (CORBA) e *Remote Method Invocation* (RMI). Segundo Prata (2008, p. 4), o CORBA tem por objetivo ser uma arquitetura independente de linguagem, plataforma e *hardware*.

Segundo Castro, Raeder e Nunes (2007, p. 3), “RMI trata-se de uma solução Java para implementar comunicação entre aplicações distribuídas orientadas a objetos.”

2.2.2 *Remote Method Invocation*

Remote Method Invocation (RMI) é um *middleware* de comunicação de objetos distribuídos, desenvolvido e mantido pela Sun, empresa do grupo Oracle. Segundo Coulouris e Kindberg (2005, p. 17), o termo *middleware* aplica-se a uma camada de *software* que fornece uma abstração, sendo esta capaz de mascarar a heterogeneidade de infra-estrutura. Além disso, fornece um modelo computacional uniforme para ser utilizada por programadores de sistemas distribuídos.

A utilização de *middlewares* garante um bom nível de abstração para o desenvolvedor. Desenvolver utilizando RMI é muito semelhante ao modo como se desenvolve para ambiente local. Segundo Carvalhal (2001, p. 4), o RMI é um ambiente de programação distribuído homogêneo e orientado a objetos, que permite que desenvolvedores façam uso da sintaxe e semântica já conhecida do Java para desenvolver aplicações distribuídas.

Segundo Castro, Raeder e Nunes (2007, p. 3), “o servidor disponibiliza os serviços para serem acessados remotamente. O cliente por sua vez utiliza esses serviços de acordo com

sua necessidade. Desta forma é possível que objetos em diferentes máquinas virtuais interajam entre si independentemente da localização física dessas máquinas.”

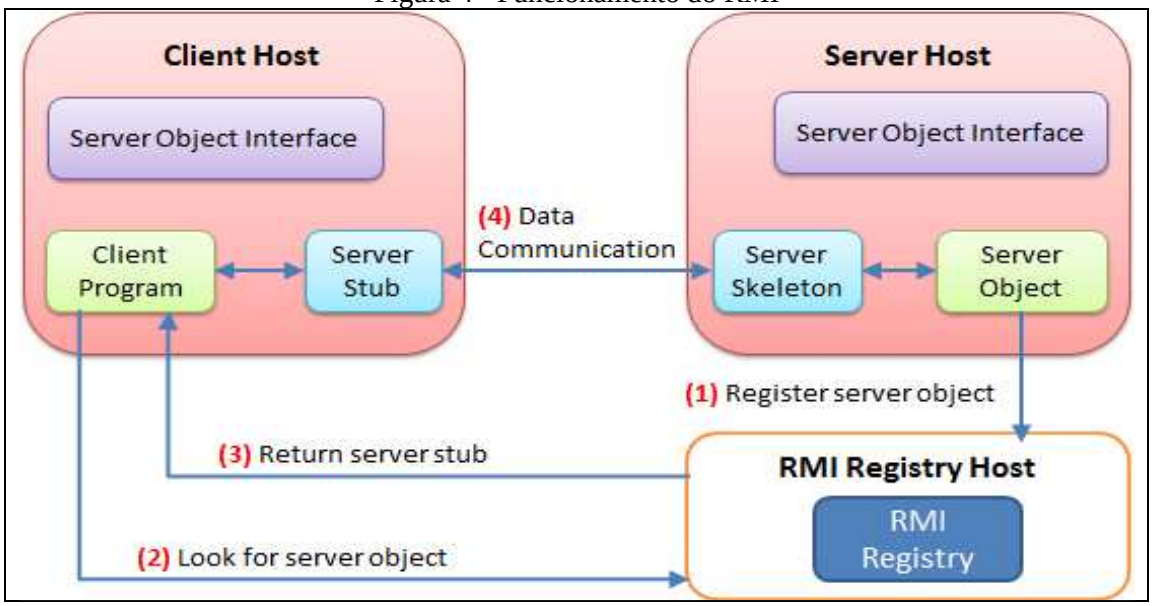
Para que isto seja viável é necessário que exista um meio de descrever os serviços ou funcionalidades que serão disponibilizadas pelo servidor. Este meio é denominado interface remota e serve como ponto de abertura, ou seja, é um meio de outras aplicações se comunicarem com a aplicação servidora. Segundo Burke e Monson-Haefel (2007, p. 11), a interface remota é usada para definir métodos de negócio que podem ser acessados a partir de outros aplicativos. Estas serão as funcionalidades disponíveis para o mundo exterior da aplicação servidora.

Segundo Horstman e Cornell (2001, p. 227), quando o objeto cliente quer chamar um método remoto, na verdade ele faz uma chamada a um método normal que está encapsulada em um objeto substituto chamado *stub*. Este objeto realiza a comunicação com o *skeleton* da interface remota que ele referencia, sendo também responsável por realizar a reunião dos parâmetros necessários para a invocação do método remoto. Segundo Oracle (2001), o *stub* pode realizar as chamadas aos métodos da interface porque ele próprio implementa o conjunto de interfaces remotas do objeto que representa, contudo apenas os métodos definidos nas interfaces remotas poderão ser chamados.

Afirma Burke e Monson-Haefel (2007, p. 15) que o *stub* é responsável por enviar o método de invocação pela rede até o objeto real no servidor. O processo de codificação dos parâmetros é chamado de reunião de parâmetros. O objetivo da reunião é converter os parâmetros para um formato apropriado para o transporte de uma máquina virtual para outra.

A figura 4 ilustra uma aplicação cliente/servidor usando RMI. Inicialmente o servidor deve se registrar no serviço de resolução de nomes conforme `passo 1`. Neste caso o `rmiregistry`. Uma vez registrado, o servidor poderá ser localizado pelos clientes. Assim eles iniciam uma conexão procurando pelos serviços registrados em um endereço *Internet Protocol* (IP) como pode ser visto no `passo 2,` se o serviço procurado estiver registrado no `rmiregistry` naquele IP, uma referência para o servidor é retornado para o cliente conforme `passo 3`. A partir deste momento o cliente já poderá executar chamadas remotas ao serviço do servidor conforme `passo 4`.

Figura 4 - Funcionamento do RMI



Fonte: Taing (2011).

2.3 SEGURANÇA DA INFORMAÇÃO

Segundo Araujo (2008, p. 2), segurança da informação é a intenção de proteger sistemas e seus dados, assim como impedir acesso não autorizado ao mesmo com propósito de impedir, detectar e deter qualquer ameaça que o sistema venha a sofrer.

A necessidade de possuir um sistema seguro deve-se à garantia de trabalhar com dados reais e consistentes. Assim quando evita-se o acesso não autorizado, evita-se que os dados sejam alterados por pessoas que não estariam aptas a realizar tais alterações, aumentando a confiabilidade do sistema. Esses acontecimentos nem sempre são intencionais e/ou maliciosos, porém geram uma exposição desnecessária a situações que talvez não estivessem previstas.

Segundo Araujo (2008, p. 3), evitar este tipo de exposição acaba por reduzir os riscos de fraude, sabotagem, roubo e outras atitudes maliciosas. A segurança não traz apenas recursos para impedir atos de má fé. Ela também agrega um bom grau de resguardo com as informações as quais o sistema faz uso. No intuito de proteger os dados, existem técnicas que podem ser aplicadas para garantir a segurança dos mesmos, com a finalidade de obter características de sigilo, autenticidade e integridade.

2.3.1 Criptografia

Afirma Lemos (2010, p. 71) que a criptografia é uma técnica de ocultar de terceiros dados e/ou informações que devem ser compartilhadas. Segundo Schmitt (2001, p. 1) criptografia é o processo de transformação de um texto qualquer em um texto ininteligível, com propósito de garantir a confidencialidade e/ou autenticidade do mesmo.

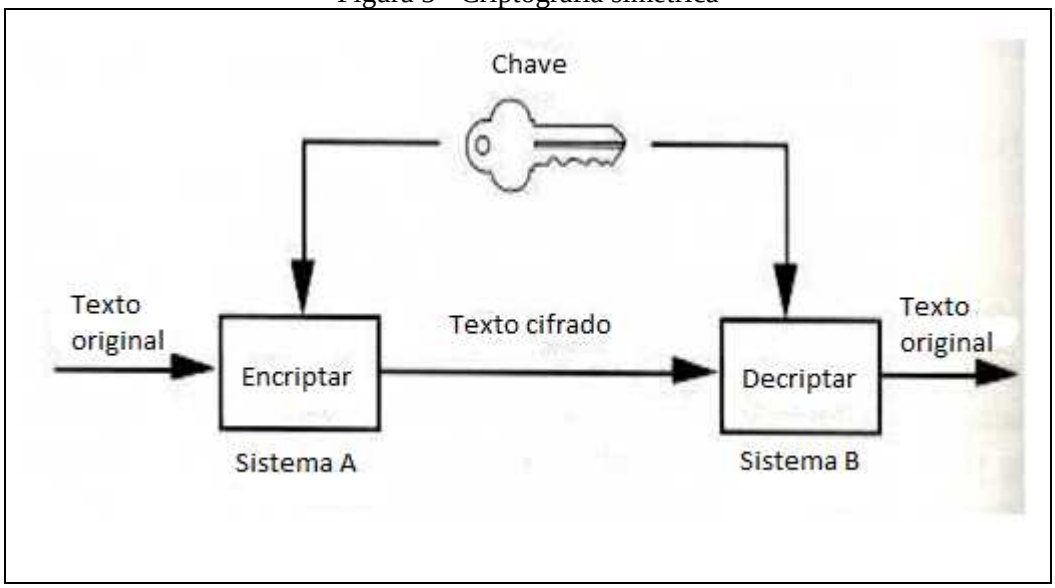
Trata-se de uma técnica de segurança muito utilizada para preservar informações importantes ou sensíveis que trafegam por um meio de um canal de comunicação inseguro. López (2012, p. 5) afirma que a criptografia é largamente usada para obter as seguintes características nos sistemas desenvolvidos: sigilo, autenticidade, integridade. Elas são melhor descritas a seguir:

a) sigilo: visa impedir o acesso indesejado aos dados e/ou informação. Segundo López (2012, p. 9) o sigilo visa proteger os dados de divulgação não autorizada;
b) autenticidade: visa garantir a procedência dos dados. Segundo López (2012, p. 9), garantir que os elementos envolvidos em uma comunicação são realmente quem afirmam ser;
c) integridade: visa impedir que os dados e/ou informação seja adulterada. Segundo López (2012, p. 9), esta característica se refere à garantia que a informação recebida é exatamente a mesma que foi enviada.

Para tanto existem dois tipos de técnicas criptográficas, sendo elas:

a) simétrica: neste tipo de criptografia faz-se uso de uma única chave, usada tanto para cifrar como para decifrar a informação. Segundo Oliveira (2012, p. 2), esta técnica é a mais antiga, na qual a chave usada para trocar informações entre as partes envolvidas é a mesma, devendo esta permanecer em segredo. A figura 5 demonstra o processo de cifragem e decifragem utilizando esta técnica, onde pode ser visto que durante os processos de cifrar e decifrar uma mensagem uma única chave é utilizada. A mensagem é cifrada no `Sistema A` utilizando uma chave. O texto cifrado é enviado para `Sistema B` que decifra a mensagem utilizando a mesma chave com que a mensagem foi cifrada;

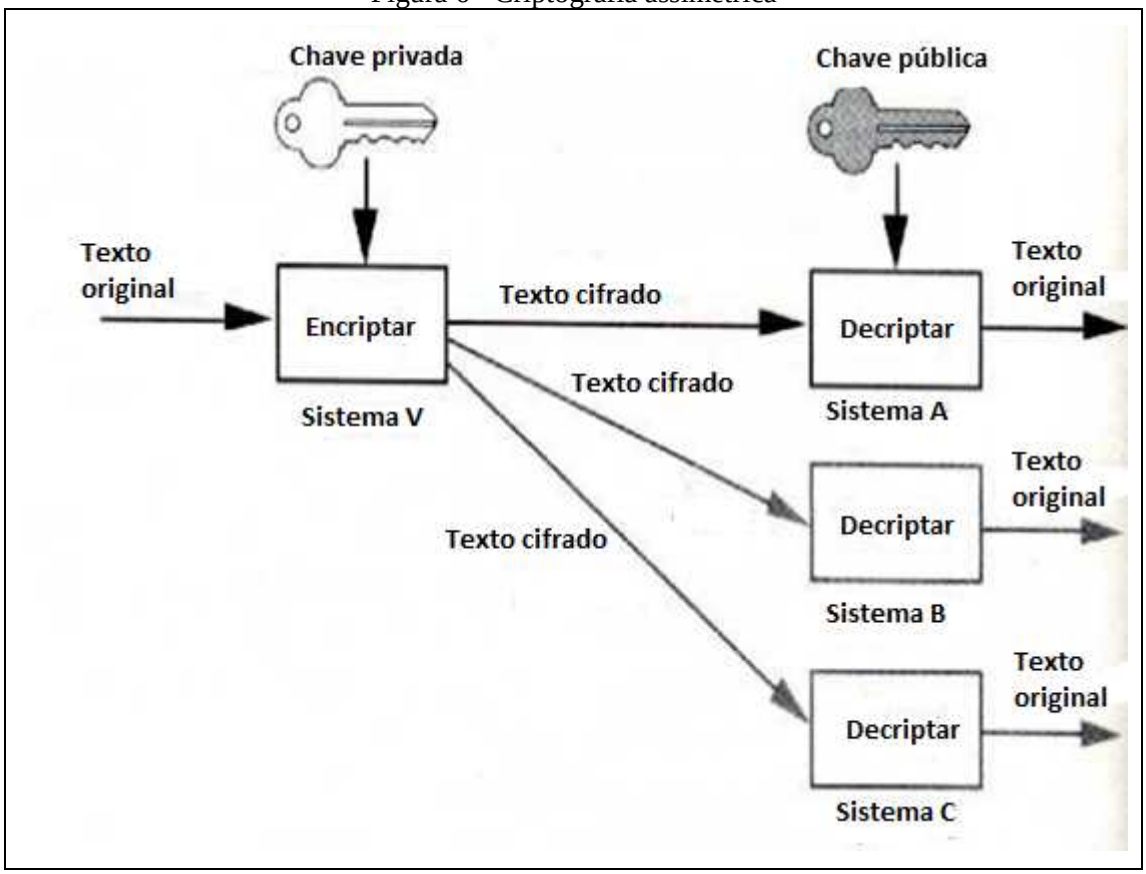
Figura 5 - Criptografia simétrica



Fonte: adaptado de Schmitt (2001).

b) assimétrica: a criptografia assimétrica basea-se na proposta de chaves públicas. Segundo Lemos (2010, p. 70), esta técnica foi proposta em 1976 por Diffie e Hellman. Nela propõem-se que cada usuário conheça um par de chaves criptográficas sendo uma delas usada para cifrar e a outra para decifrar uma informação. Neste caso a chave para decifragem é tornada pública. A figura 6 demonstra o processo de cifragem e decifragem utilizando criptografia assimétrica. Como pode ser visto, estes processos utilizam chaves diferentes, sendo um par de chaves no qual uma destas é pública e outra é privada. A chave privada é usada para cifrar o texto original, já a chave pública é utilizada para decifrar o texto cifrado recebido.

Figura 6 - Criptografia assimétrica



Fonte: adaptado de Schmitt (2001).

Segundo Lemos (2010, p. 72) em 1978 foi desenvolvido o primeiro algoritmo de criptografia de chave pública por Rivest, Shamir e Adleman. Sendo este conhecido como RSA devido aos nomes de seus criadores.

Este algoritmo é bastante utilizado e possui um elevado nível de segurança. Lemos (2010, p. 86) afirma que este algoritmo é muito utilizado para garantir a segurança em transações eletrônicas. Segundo Schmitt (2001, p. 6), ele é baseado em cálculos matemáticos que possuem como entrada dois números primos grandes. Afirma Lemos (2010, p. 76) que o custo de decifragem deste algoritmo é elevado, quanto maiores forem os números primos de entrada, maior será o custo para decifrar uma mensagem.

2.4 TRABALHOS CORRELATOS

Existem disponíveis no mercado aplicações que se propõem a efetuar a distribuição de software, atuando desde o momento do empacotamento do *release* até a sua implantação. Foram selecionadas as ferramentas da UrbanCode , sendo elas a *uBuild, uDeploy*e o *Java Web Start* da Oracle.

2.4.1 uBuild

Esta aplicação é semelhante a um servidor de integração contínua como outros disponíveis no mercado, ou seja, ela é responsável por localizar os recursos necessários para efetuar a construção de uma determinada aplicação e gerenciar os dados da construção. Segundo Urbancode (2013), a *uBuild* é uma solução de construção de *software* baseada em uma plataforma própria, sendo um servidor de integração contínua com uma arquitetura escalável capaz de realizar *builds* complexos.

O processo de construção da ferramenta inicia com os desenvolvedores realizando o *commit* dos seus fontes na aplicação *uBuild*. Ela gerencia estes recursos, cria os artefatos pertinentes e gerencia um repositório de artefatos que servem como recursos de entrada para outras ferramentas da empresa.

A ferramenta efetua a construção de aplicações desenvolvidas em várias linguagens, sendo algumas delas C#, Java e C/C++.

2.4.2 uDeploy

Esta ferramenta realiza a entrega (*deploy*) das *releases* construídas pela aplicação uBuild. Ela dá continuidade ao processo de liberação. A partir de uma construção gera-se um pacote para *deploy*, o qual vai para uma fila onde aguarda aprovação. Quando aprovada a entrega, esta é realizada deixando o pacote em um novo estado, ou seja, pronto para implantação no cliente. Segundo a Urbancode (2013), “implantação de aplicativos é o processo de entrega de versões específicas de vários componentes de um aplicativo em um ambiente de destino.”

O processo de *deploy* está sujeito a vários contratempos, porém a utilização de métodos bem definidos e estáveis podem auxiliar a superar as barreiras encontradas neste processo. Segundo Urbancode (2013), “*uDeploy* resolve os problemas de implantações

desafiadoras, tais como conteúdo de implementações incrementais, código de *middleware* e implantação de configuração, implantação de banco de dados e reversões.”

Segundo UrbanCode (2013), “uDeploy fornece uma estrutura de implantação automatizada que reduz os erros de implementação e melhora a eficiência, exatidão e rastreabilidade”

2.4.3 Java Web Start

Esta tecnologia está disponível na plataforma Java e atua na distribuição de aplicativos escritos nesta linguagem. Segundo Oracle (2005, p. 2), é uma tecnologia para *deploy* de aplicações java, que permite ao usuário baixar e executar aplicações sem passar por processos de instalação complicados.

Ela possui várias características importantes como: versionamento e atualização incremental, assinatura de código, instalação automática de dependências, independente de plataforma, entre outras. Estas características são todas pertinentes, porém as mais notáveis são: atualização incremental e instalação automática de dependências. A atualização incremental dispensa a necessidade de distribuir uma *release* para cada cliente, sendo que a aplicação que faz uso desta tecnologia verifica se houve alterações no repositório. Caso tenha ocorrido alguma alteração, a atualização é iniciada. Segundo Haghighat (2007, p. 2), a atualização da aplicação ocorre durante a inicialização da mesma, onde uma etapa de verificação é executada.

A instalação automática de dependências reduz as possibilidades de ocorrerem erros durante a instalação da aplicação. Segundo Marcelino (2004, p. 1), o J*ava Web Start* verifica se todos os recursos necessários estão disponíveis na máquina do cliente, verificando até mesmo a instalação da *Java Runtime Enviromment* (JRE), caso algum dos recursos não esteja disponível ele será baixado antes da aplicação ser instalada.

A tecnologia consiste em agregar a uma aplicação Java um protocolo de *deploy*, denominado *Java Network Launch Protocol* (JNLP). Um arquivo JNLP é um conjunto de definições, onde são configuradas as dependências da aplicação, local do repositório, definições de segurança, modo de execução, entre outras propriedades.

3 DESENVOLVIMENTO

Neste capítulo são abordadas as etapas envolvidas no desenvolvimento do sistema proposto. Sendo apresentados os principais requisitos, especificações, implementações, mencionando as tecnologias aplicadas. Por fim será apresentada a operacionalidade do sistema.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Abaixo apresentam-se os Requisitos Funcionais (RF) da aplicação servidora no quadro 1 e seus Requisitos Não Funcionais (RNF) no quadro 2.

Quadro 1 - Requisitos funcionais da aplicação servidora.

Requisito Funcional
RF01S – O servidor deverá liberar uma nova versão.
RF02S – O servidor deverá manter usuários.
RF03S – O servidor deverá manter o histórico de liberação.
RF04S – O servidor deverá gerar um conjunto de chaves de segurança.
RF05S – O servidor deverá exportar chaves de segurança.
RF06S – O servidor deverá exportar versão para *pendrive*.
RF07S – O servidor deverá realizar operações de instalação, reinstalação e voltar versão.
RF08S – O servidor deverá identificar os clientes conectados e desconectados.

Quadro 2 - Requisitos não funcionais da aplicação servidora.

Requisito Não Funcional
RNF01S – O servidor deverá ser implementado em Java.

Abaixo apresentam-se os Requisitos Funcionais (RF) da aplicação cliente no quadro 3 e seus Requisitos Não Funcionais (RNF) no quadro 4.

Quadro 3 - Requisitos funcionais da aplicação cliente.

Requisito Funcional
RF01C – O cliente deverá realizar a operação de baixar versão.
RF02C – O cliente deverá realizar a operação de voltar versão.
RF03C – O cliente deverá realizar a operação de instalar versão.
RF04C – O cliente deverá realizar a operação de instalar versão a partir de *pendrive*.
RF05C – O cliente deverá manter o diretório de aplicação.
RF06C – O cliente deverá solicitar chave de segurança.
RF07C – O cliente deverá realizar reconexao em caso de queda do servidor.
RF08C – O cliente deverá gerar notificações na bandeja.

Quadro 4 - Requisitos não funcionais da aplicação cliente.

Requisito Não Funcional
RNF01C – O cliente deverá ser implementado em Java.
RNF02C – O cliente deverá ser capaz de executar em segundo plano.

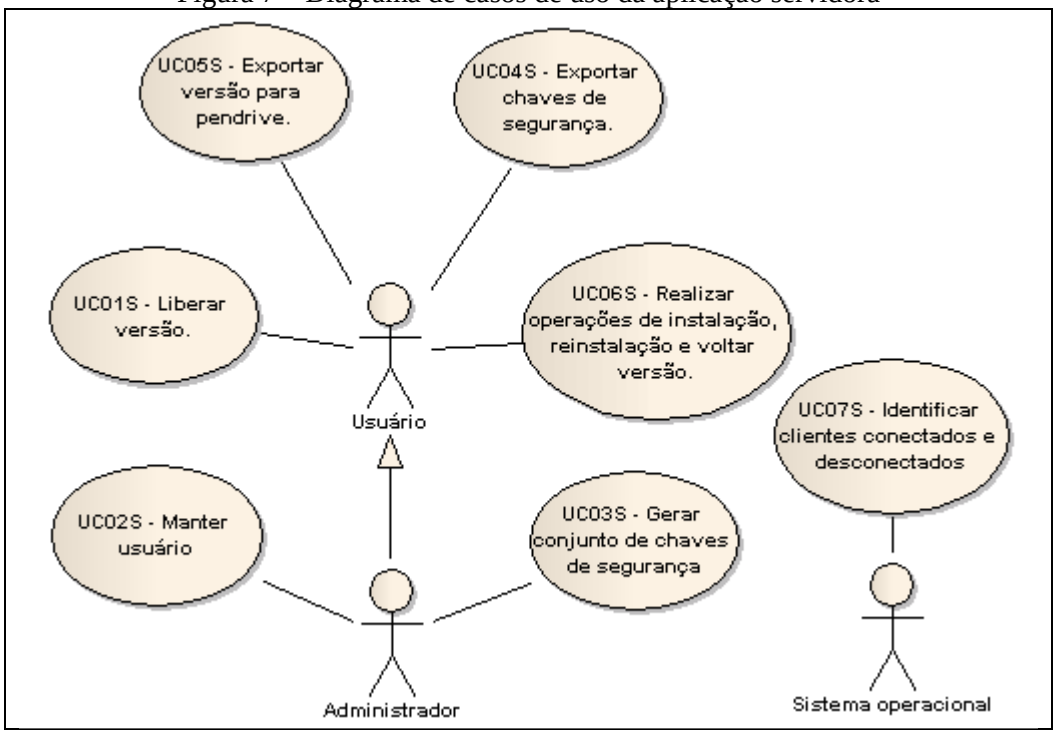
3.2 ESPECIFICAÇÃO

A especificação do sistema proposto foi realizada utilizando os diagramas da *Unified Modeling Language* (UML) em conjunto com a ferramenta *Enterprise Architect* 7.5. Foram utilizados os diagramas de caso de uso, diagrama de classes, diagrama de sequência que serão apresentados nas seções seguintes.

3.2.1 Diagrama de casos de uso da aplicação servidora

Nesta seção será apresentado o diagrama de casos de uso da aplicação servidora, conforme apresentado na figura 7, seguido do detalhamento dos mesmos.

Figura 7 – Diagrama de casos de uso da aplicação servidora



3.2.1.1 Liberar versão

No quadro 5 é descrito o caso de uso que permite a liberação de novas *releases*. Este caso de uso implementa o requisito funcional RF01S e RF03S.

Quadro 5 - Caso de uso UC01S

Número	01S
Caso de uso	Liberar versão.
Descrição	Este caso de uso tem por objetivo gerar um novo *release* da aplicação alvo no diretório de histórico para que possa ser posteriormente distribuído.
Ator	`Usuário`
Pre-condições	O `usuário` deve estar logado.
Pós-condições	Gera um release no diretório de histórico da aplicação.
Cenário principal	1. Selecionar o menu“Ações” na barra de tarefas da aplicação; 2. Em seguida selecionar a opção “Liberar nova versão”; 3. Informar o caminho do diretório da aplicação que será liberada; 4. Informar o número da versão do sistema; 5. Informar o número da versão do banco de dados; 6. Informar o número da construção; 7. Informar qual a liberação do dia; 8. Clicar em no botão “Liberar”.
Alternativo	1. Após o passo 7 podem ser adicionados arquivos de execução em lote (arquivos de extensão .bat) ao release, bastando clicar no botão “Adicionar”; 2. Em seguida selecione o caminho do arquivo; 3. Informe qual o tipo de cliente que deve ser executado; 4. Clicar no botão “Adicionar”.

3.2.1.2 Manter usuário

No quadro 6 é descrito o caso de uso que permite manter o controle do cadastro dos usuários da aplicação servidora. Este caso de uso implementa o requisito funcional RF02S.

Quadro 6 - Caso de uso UC02S

Número	02S
Caso de uso	Manter usuário.
Descrição	Este caso de uso tem por objetivo cadastrar, alterar e excluir um usuário da aplicação.
Ator	`Administrador`
Pre-condições	O `administrador` deve estar logado.
Pós-condições	Cadastra, altera ou exclui um usuário da aplicação.
Cenário principal	1. O `administrador` deve selecionar o menu “Cadastros” na barra de tarefas; 2. Selecionar a opção “Cadastro usuário”; 3. Informar o nome de usuário; 4. Informar a senha de usuário; 5. Informar o nome do usuário do sistema; 6. Informar o identificador do usuário; 7. O administrador deve clicar no botão “Cadastrar”.
Alternativo 1	1. Após o passo 2 clique no botão “Visualizar”; 2. Informe o nome do usuário que deseja-se visualizar; 3. Clique no botão “Excluir”.
Alternativo 2	1. Após o passo 2 clique no botão “Visualizar”; 2. Informe o nome do usuário que se quer visualizar; 3. Altere a informação desejada; 4. Clique no botão “Editar”.
Exceção	1. No passo 7, caso já exista um usuário com os dados informados cadastrados, será exibido uma mensagem de erro informando esta situação.

3.2.1.3 Gerar conjunto de chaves

No quadro 7 é descrito o caso de uso que permite gerar um conjunto de chaves de segurança. Esta operação somente pode ser executada por um usuário administrador sendo que estas servem para homologar e/ou autorizar a execução de uma operação executada localmente em um dos clientes. Este caso de uso implementa o requisito funcional RF04S.

Quadro 7 - Caso de uso UC03S

Número	03S
Caso de uso	Gerar conjunto de chaves de segurança.
Descrição	Gera um conjunto de chaves de segurança criptografadas, para serem usadas na validação das operações locais dos clientes.
Ator	`Administrador`
Pre-condições	O `administrador` deve estar logado.
Cenário principal	1. O `administrador` deve selecionar o menu “Ações” na barra de tarefas; 2. O `administrador` deve selecionar o menu “Gerar chaves de emergência”; 3. Surgirá uma mensagem informando a execução da operação. 4. O número da versão do arquivo de chaves será incrementado na interface gráfica.

3.2.1.4 Exportar chaves de segurança

No quadro 8 é descrito o caso de uso que permite exportar o conjunto de chaves de segurança para um disco removível, com a finalidade de executar manualmente a instalação das chaves em um cliente que por ventura não consiga realizar uma conexão com o servidor ou que possua problemas em relação ao link de comunicação. Este caso de uso implementa o requisito funcional RF05S.

Quadro 8 - Caso de uso UC04S

Número	04S
Caso de uso	Exportar chaves de segurança.
Descrição	Essa funcionalidade exporta para um arquivo em uma unidade de armazenamento selecionada o arquivo de chaves e a chave pública da aplicação.
Ator	`Usuário`
Pre-condições	O `usuário` deve estar logado.
Cenário principal	1. Informar a unidade de armazenamento onde se deseja exportar as chaves. 2. Surgirá uma caixa de diálogo informando a realização da operação.

3.2.1.5 Exportar versão para *pendrive*

No quadro 9 é descrito o caso de uso que permite um usuário exportar uma *deploy* do último *release* liberado com a finalidade de executar a instalação manualmente em um cliente

que não consiga estabelecer uma conexão com o servidor ou que possua problemas em relação ao link de comunicação. Este caso de uso implementa o requisito funcional RF06S.

Quadro 9 - Caso de uso UC05S

Número	05S
Caso de uso	Exportar versão para *pendrive*.
Descrição	Este caso de uso tem por objetivo gerar um arquivo de deploy para que possa ser efetuado de forma manual pelo usuário responsável.
Ator	`Usuário`
Pre-condições	O `usuário` deve estar logado.
Cenário principal	1. O `usuário` deverá selecionar uma unidade de disco removível onde tenha permissão para gravar arquivos; 2. A aplicação gerará o deploy no disco selecionado; 3. A aplicação apresentará uma mensagem indicando a realização da operação e o caminho do deploy que foi gerado.
Exceção	1. Após o passo 2, caso o servidor não tenha nenhuma versão liberada, será apresentada uma mensagem de erro.

3.2.1.6 Realizar operações

No quadro 10 é descrito o caso de uso que permite um usuário exportar uma *deploy* do último *release* liberado com a finalidade de executar a instalação manualmente em um cliente que não consiga estabelecer uma conexão com o servidor ou que possua problemas em relação ao link de comunicação. Este caso de uso implementa o requisito funcional RF07S.

Quadro 10 - Caso de uso UC06S

Número	06S
Caso de uso	Realizar operações de baixar versão, instalação, reinstalação e voltar versão.
Descrição	Este caso de uso tem por objetivo enviar um comando para um conjunto de clientes conectados, sendo que este pode ser um comando de baixar versão, voltar versão, instalar, reinstalar.
Ator	`Usuário`
Pre-condições	O `usuário` deve estar logado.
Cenário principal	1. A aplicação apresentará uma caixa de diálogo com os comandos possíveis; 2. Estes serão listados em uma caixa de combinação; 3. O `usuário` seleciona o comando desejado na caixa de combinação; 4. A aplicação atualiza o status de notificação dos clientes selecionados na tabela de identificação dos clientes.
Exceção	1. Após o passo 4, havendo alguma inconsistência na operação, o cliente responderá com uma notificação de erro para o servidor.

3.2.1.7 Identificar clientes conectados e desconectados

No quadro 11 é descrito o caso de uso que permite a aplicação servidora detectar os clientes conectados e desconectados. Este caso de uso implementa o requisito funcional RF08S.

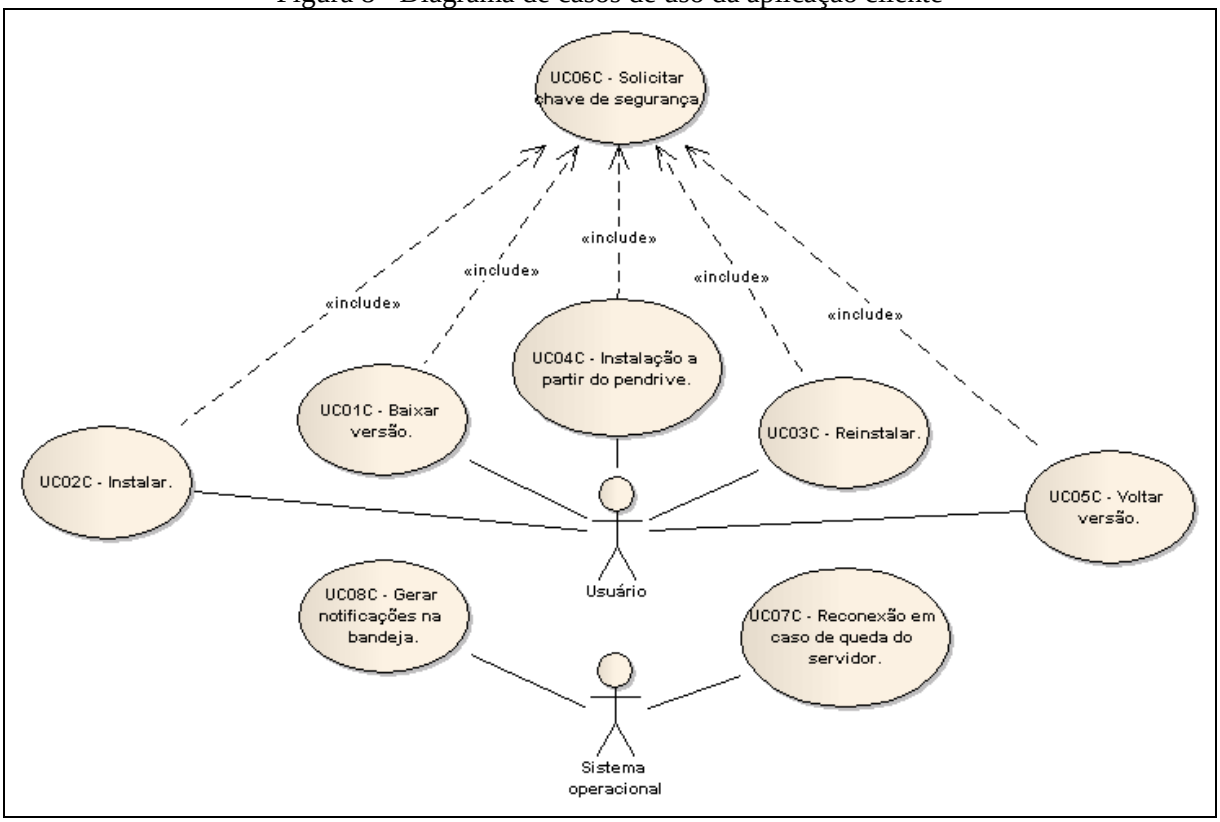
Quadro 11 – Caso de uso UC07S

Número	07S
Caso de uso	Identificar clientes conectados e desconectados
Descrição	Este caso de uso tem por objetivo verificar quais clientes estão conectados e quais estão desconectados.
Ator	`Sistema operacional`
Cenário principal	1. Iniciar o contador de *timeout*; 2. Receber as notificações de conexão dos clientes; 3. Alterar o *status* dos clientes que notificaram a conexão; 4. Verificar os clientes que enviaram a notificação e desconectar os clientes que não enviaram a notificação.

3.2.2 Diagrama de casos de uso da aplicação cliente

Nesta seção será apresentado o diagrama de caso de uso da aplicação cliente conforme a figura 8, seguido do seu respectivo detalhamento. Esta aplicação deve, a princípio, ser executada em segundo plano, sendo possível acessá-la a partir da bandeja do sistema. Ela serve como um agente que executa as ações enviadas remotamente pela aplicação servidora.

Figura 8 - Diagrama de casos de uso da aplicação cliente



3.2.2.1 Realizar operação de baixar versão

No quadro 12 é descrito o caso de uso que permite um usuário baixar o último *release* liberado na aplicação servidora. Para tanto deverá possuir uma chave de segurança que autoriza a execução da operação. Esta chave é necessária para garantir que a operação esta sendo realizada de forma adequada e com o conhecimento dos responsáveis pelo servidor de aplicação. Este caso de uso implementa o requisito funcional RF01C.

Quadro 12 - Caso de uso UC01C

Número	01C
Caso de uso	Realizar operação de baixar versão.
Descrição	Este caso de uso tem por objetivo baixar o último *release* liberado para *deploy* na aplicação servidora.
Ator	`Usuário`
Cenário principal	1. A aplicação irá solicitar a chave de segurança que possua uma determinada numeração; 2. O `usuário` deve informar a chave solicitada pela aplicação, executa o UC06; 3. A aplicação baixa a última liberação existente no servidor.

3.2.2.2 Realizar operação de instalar

No quadro 13 é descrito o caso de uso que permite a um usuário executar a instalação de uma *release* baixada anteriormente da aplicação servidora. Para tanto deverá possuir uma chave de segurança que autoriza a execução da operação. Esta chave é necessária para garantir que a operação está sendo realizada de forma adequada e com o conhecimento dos responsáveis pelo servidor de aplicação. Este caso de uso implementa o requisito funcional RF02C.

Quadro 13 - Caso de uso UC02C

Número	02C
Caso de uso	Realizar operação de instalar.
Descrição	Este caso de uso tem por objetivo realizar a instalação de uma *release* que tenha sido anteriormente baixada.
Ator	`Usuário`
Pre-condições	O `usuário` deve ter realizado um comando de baixar versão previamente.
Cenário principal	1. A aplicação irá solicitar a chave de segurança que possua uma determinada numeração; 2. O `usuário` deve informar a chave solicitada pela aplicação, executa o caso de uso UC06; 3. A aplicação realiza a instalação do último release baixado.

3.2.2.3 Realizar operação de reinstalar

No quadro 14 é descrito o caso de uso que permite um usuário executar a reinstalação de uma *release* anteriormente instalada no cliente. Para tanto deverá possuir uma chave de segurança que autoriza a execução da operação. Esta chave é necessária para garantir que a operação está sendo realizada de forma adequada e com o conhecimento dos responsáveis pelo servidor de aplicação. Este caso de uso implementa o requisito funcional RF03C.

Quadro 14 - Caso de uso UC03C

Número	03C
Caso de uso	Realizar operação de reinstalar.
Descrição	Este caso de uso tem por objetivo realizar a reinstalação da *release* atualmente instalada.
Ator	`Usuário`
Cenário principal	1. A aplicação irá solicitar a chave de segurança que possua uma determinada numeração; 2. O `usuário` deve informar a chave solicitada pela aplicação, executa o caso de uso UC06; 3. A aplicação reinstala a versão do *release* atualmente instalado.

3.2.2.4 Realizar operação de instalar do *pendrive*

No quadro 15 é descrito o caso de uso que permite um usuário executar a instalação de uma *release* liberada na aplicação servidora exportada em um disco removível. Este processo deve ser executado quando houver problemas com o *link* de comunicação e/ou outro contratempo que impossibilite a execução deste processo remotamente pela aplicação servidora. Para tanto deverá possuir uma chave de segurança que autoriza a execução da operação, sendo que esta chave é necessária para garantir que a operação esta sendo realizada de forma adequada e com o conhecimento dos responsáveis pelo servidor de aplicação. Este caso de uso implementa o requisito funcional RF04C.

Quadro 15 - Caso de uso UC04C

Número	04C
Caso de uso	Realizar operação de instalação a partir do *pendrive*.
Descrição	Este caso de uso tem por objetivo realizar a instalação de um *release* gerado para instalação manual.
Ator	`Usuário`
Cenário principal	1. O `usuário` deverá selecionar o menu “Ações” na barra de tarefas; 2. O `usuário` deve selecionar o menu “Atualizar do pendrive”; 3. A aplicação irá solicitar a chave de segurança que possua uma determinada numeração; 4. O `usuário` deve informar a chave solicitada pela aplicação, ver caso de uso UC06; 5. A aplicação realiza a instalação do *release* gerado para instalação manual.

3.2.2.5 Realizar operação de voltar versão

No quadro 16 é descrito o caso de uso que permite ao usuário voltar à uma *release* (versão) anteriormente instalada no cliente. Para tanto deverá possuir uma chave de segurança que autoriza a execução da operação, sendo que esta chave é necessária para garantir que a operação esta sendo realizada de forma adequada e com o conhecimento dos responsáveis pelo servidor de aplicação. Este caso de uso implementa o requisito funcional RF05C.

Quadro 16 - Caso de uso UC05C

Número	05C
Caso de uso	Realizar operação de voltar versão.
Descrição	Este caso de uso tem por objetivo realizar a reinstalação de um release anteriormente instalado.
Ator	`Usuário`
Pre-condições	Deve-se ter uma instalação de um release anterior ao instalado atualmente.
Cenário principal	1. A aplicação irá solicitar a chave de segurança que possua uma determinada numeração; 2. O `usuário` deve informar a chave solicitada pela aplicação; 3. Executa o caso de uso UC06; 4. A aplicação realiza a instalação da versão anteriormente instalada no cliente.
Exceção	Após o passo 2, no caso de não haver uma instalação anterior será apresentada uma mensagem informando que não existe uma versão anterior para se voltar.

3.2.2.6 Solicitar chave de segurança

No quadro 17 é descrito o caso de uso que permite realizar a conferência e validação de uma chave de segurança. Este processo é responsavel por descriptografar a chave da posição solicitada do conjunto de chaves e validá-la com a chave informada. Este caso de uso implementa o requisito funcional RF06C.

Quadro 17 - Caso de uso UC06C

Número	06C
Caso de uso	Solicitar chave de segurança.
Descrição	Este caso de uso tem por objetivo realizar solicitar ao usuário uma chave para permitir a realização de uma operação.
Ator	`Usuário`
Cenário principal	1. A aplicação apresentará uma caixa de diálogo solicitando a chave referênte a uma determinada numeração; 2. O `usuário` irá informar a chave; 3. A aplicação valida a chave; 4. A aplicação permite a execução de uma determinada operação.
Exceção	1. Após o passo 3, no caso da chave ser inválida ou incorreta será registrado um registro de *log* informando a tentativa de realizar uma operação local.

3.2.2.7 Realizar reconexão em caso de queda do servidor

No quadro 18 é descrito o caso de uso que permite detectar a queda do servidor e realizar uma reconexão quando ele estiver executando novamente. Este caso de uso implementa o requisito funcional RF07C.

Quadro 18 - Caso de uso UC07

Número	07C
Caso de uso	Realizar reconexão em caso de queda do servidor
Descrição	Este caso de uso tem por objetivo fazer com que o cliente detecte a queda do servidor e realize a reconexão.
Ator	`Sistema operacional`
Cenário principal	1. Iniciar o contador; 2. Enviar o sinal de *heartbeat*; 3. Reiniciar o contador.
Exceção	1. No passo 3, caso não exista conexão com o servidor será realizada uma nova tentiva de conexão; 2. Reiniciar o contador.

3.2.2.8 Gerar notificações na bandeja

No quadro 19 é descrito o caso de uso que permite gerar as notificações na bandeja do sistema operacional. Este caso de uso implementa o requisito funcional RF08C.

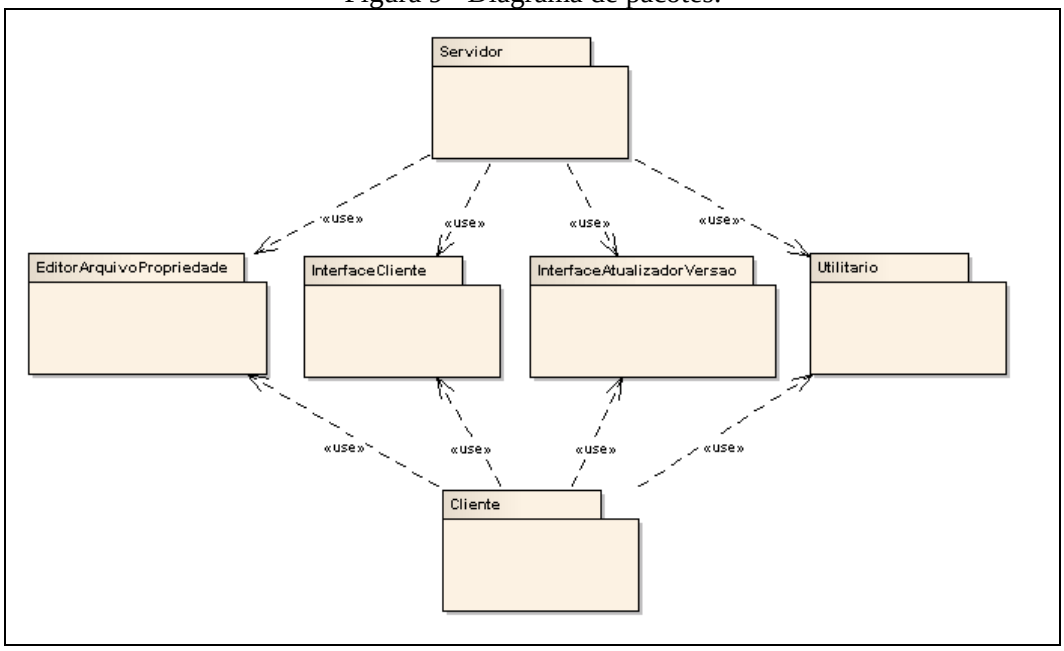
Quadro 19 - Caso de uso UC08C

Número	08C
Caso de uso	Gerar notificações na bandeja.
Descrição	Este caso de uso tem por objetivo permitir ao cliente gerar notificações na bandeja do sistema operacional.
Ator	`Sistema operacional`
Pre-condições	Estar executando em segundo plano.
Cenário principal	1. O sistema envia uma mensagem para a bandeja do sistema quando uma operação é realizada.

3.2.3 Diagramas de pacotes

O sistema foi desenvolvido de modo que fosse possível reaproveitar código. Para tanto foi necessário modularizar a solução proposta em pacotes compilados que integram partes maiores, como pode ser visto na figura 9.

Figura 9 - Diagrama de pacotes.



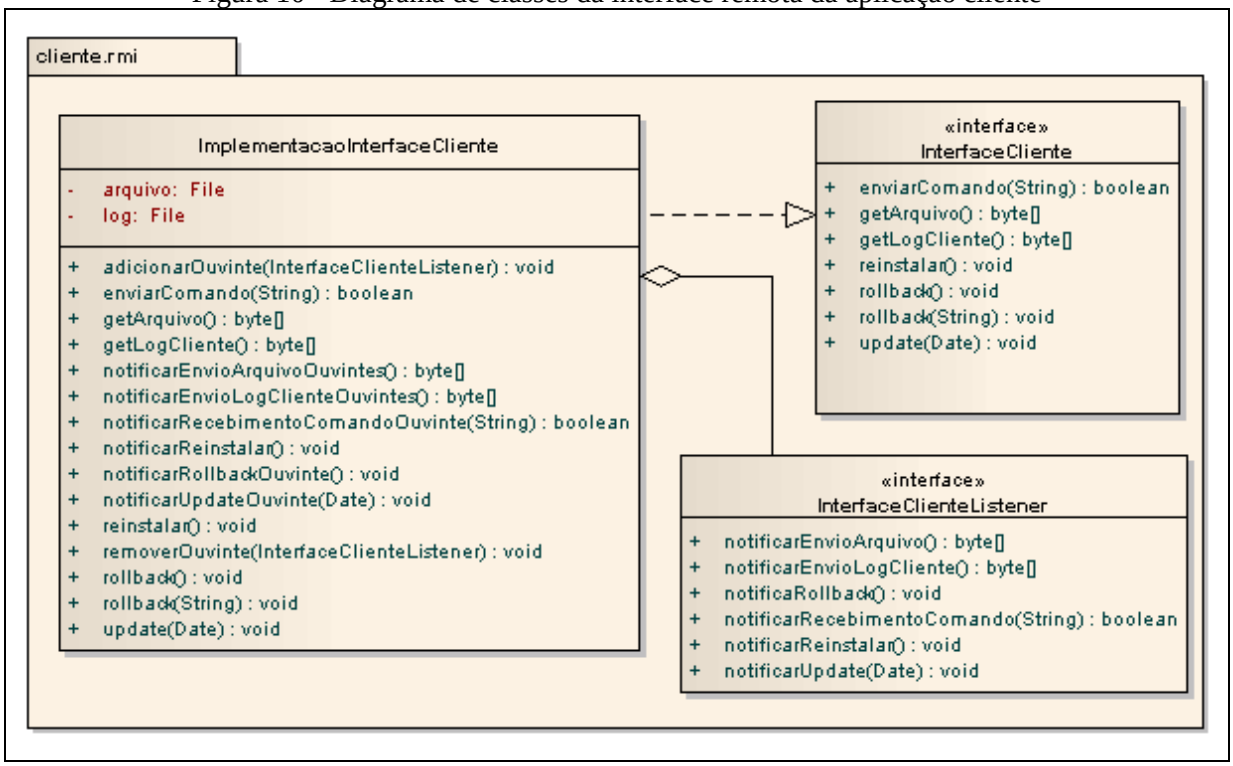
A solução está dividida em 6 pacotes, sendo estes:

a) `Servidor`: este pacote contém a aplicação servidora, assim como sua lógica de negócio;
b) `EditorArquivoPropriedade`: neste pacote se encontra o conjunto de classes que gera e lê o arquivo de configuração da aplicação cliente;
c) `InterfaceCliente`: este pacote contém a lógica de negócio da interface remota do cliente;
d) `InterfaceAtualizadorVersao`: este pacote contém a lógica de negócio da interface remota do servidor;
e) `Utilitario`: neste pacote se encontram um conjunto de classes que oferecem serviços e/ou funcionalidades genéricas úteis a ambas as aplicações (cliente e servidor), tais como geração e validação de *hash*, criptografia, gravação de arquivos e outros;
f) `Cliente`: este pacote contêm a aplicação cliente, assim como sua lógica de negócio.

3.2.4 Diagrama de classes da interface cliente

Nesta seção será visto o pacote `InterfaceCliente`, onde é definida a interface remota da aplicação cliente. O pacote é composto por uma classe e duas interfaces, como pode ser visto na figura 10. Nelas são definidos os métodos que podem ser invocados remotamentamente, assim como os eventos gerados quando uma operação é realizada.

Figura 10 - Diagrama de classes da interface remota da aplicação cliente



A interface `InterfaceCliente` é a interface remota responsável por definir os métodos que poderão ser invocados remotamente pelo servidor.

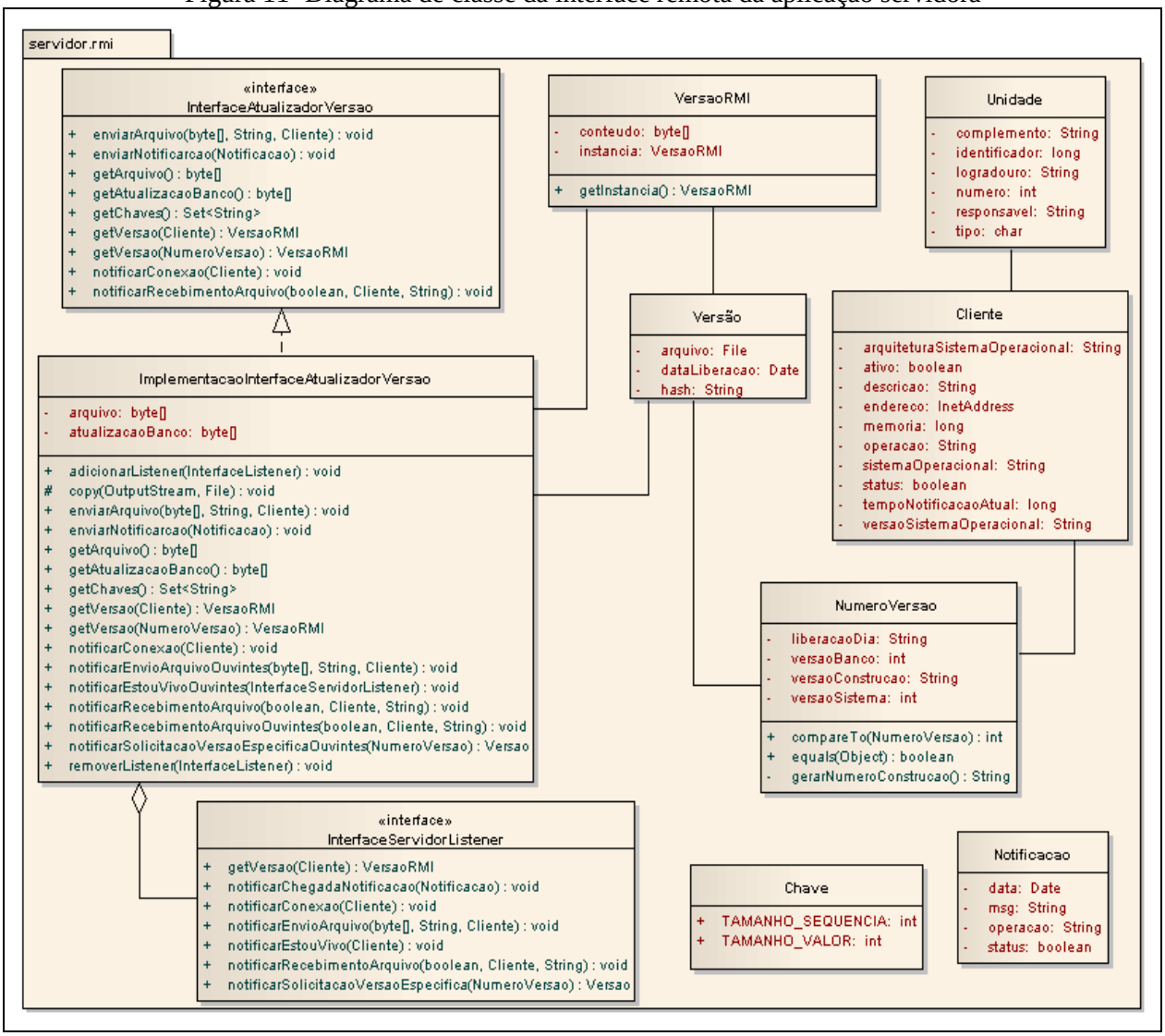
A interface `InterfaceClienteListener` implementa o padrão *observer* sobre a classe `ImplementacaoInterfaceCliente`, gerando eventos nas classes que a realizam.

A classe `ImplementacaoInterfaceCliente` é responsavel por gerar os respectivos eventos para cada invocação remota aos seus ouvintes registrados.

3.2.5 Diagrama de classes da interface servidora

Nesta seção será visto o pacote `InterfaceAtualizadorVersao`, onde é definida a interface remota da aplicação servidora. O pacote é composto por duas interfaces e oito classes, como pode ser visto na figura 11. Nelas são definidos os métodos que podem ser invocados remotamentamente, assim como os eventos gerados quando uma operação é realizada, além de objetos utilizados para a lógica de negócio do servidor. As classes deste pacote implementam a interface *Serializable* do Java, pois estes objetos trafegam entre as aplicações.

Figura 11- Diagrama de classe da interface remota da aplicação servidora



A interface `InterfaceAtualizadorVersao` é a interface remota responsável por definir os métodos que poderão ser invocados remotamente pelo cliente.

A interface `InterfaceServidorListener` implementa o padrão *observer* sobre a classe `ImplementacaoInterfaceAtualizadorVersao`, gerando eventos nas classes que a realizam.

A classe `ImplementacaoInterfaceAtualizadorVersao` é responsável por gerar os respectivos eventos para cada invocação remota aos seus ouvintes registrados.

A classe `Versao` possui os dados de uma *release* liberada, sendo estes o diretório da *release*, a data de liberação, o *hash* e o número da versão.

A classe `VersaoRMI` implementa o padrão *singleton*. Ela é a responsável por trafegar a *release* entre as aplicações.

A classe `Unidade` é responsável por guardar as informações de questões operacionais referentes a um cliente.

A classe `Cliente` é responsável por guardar as informações sistemicas de um cliente, que serão utilizadas pela aplicação servidora para obter as referências remotas de cada cliente.

A classe `Notificacao` é responsável por guardar as informações referentes ao *status* de uma operação, sendo esta enviada pelo cliente para o servidor como parâmetro na chamada remota do respectivo método.

A classe `Chave` possui as definições e regras de formação das chaves de segurança.

A classe `NumeroVersao` é responsável por armazenar as informações referentes à numeração da versão, além de possuir os métodos responsáveis para realizar as devidas comparações. Esta classe implementa a interface *Comparable*.

3.2.6 Diagrama de classes da aplicação cliente

Nesta seção será visto o pacote `Cliente`, no qual é definida a estrutura e as regras de negócio da aplicação cliente. O pacote é composto por seis interfaces e dezesseis classes, como pode ser visto na figura 12. Este pacote está sub-dividido em quatro pacotes, sendo: `visao`, `modelo`, `controle` e `acao` (sub-pacote de controle).

No sub-pacote `acao` estão definidas as classes que definem o processo e regras de negócios das operações correspondentes aos comandos que podem ser enviados de modo remoto pela aplicação servidora.

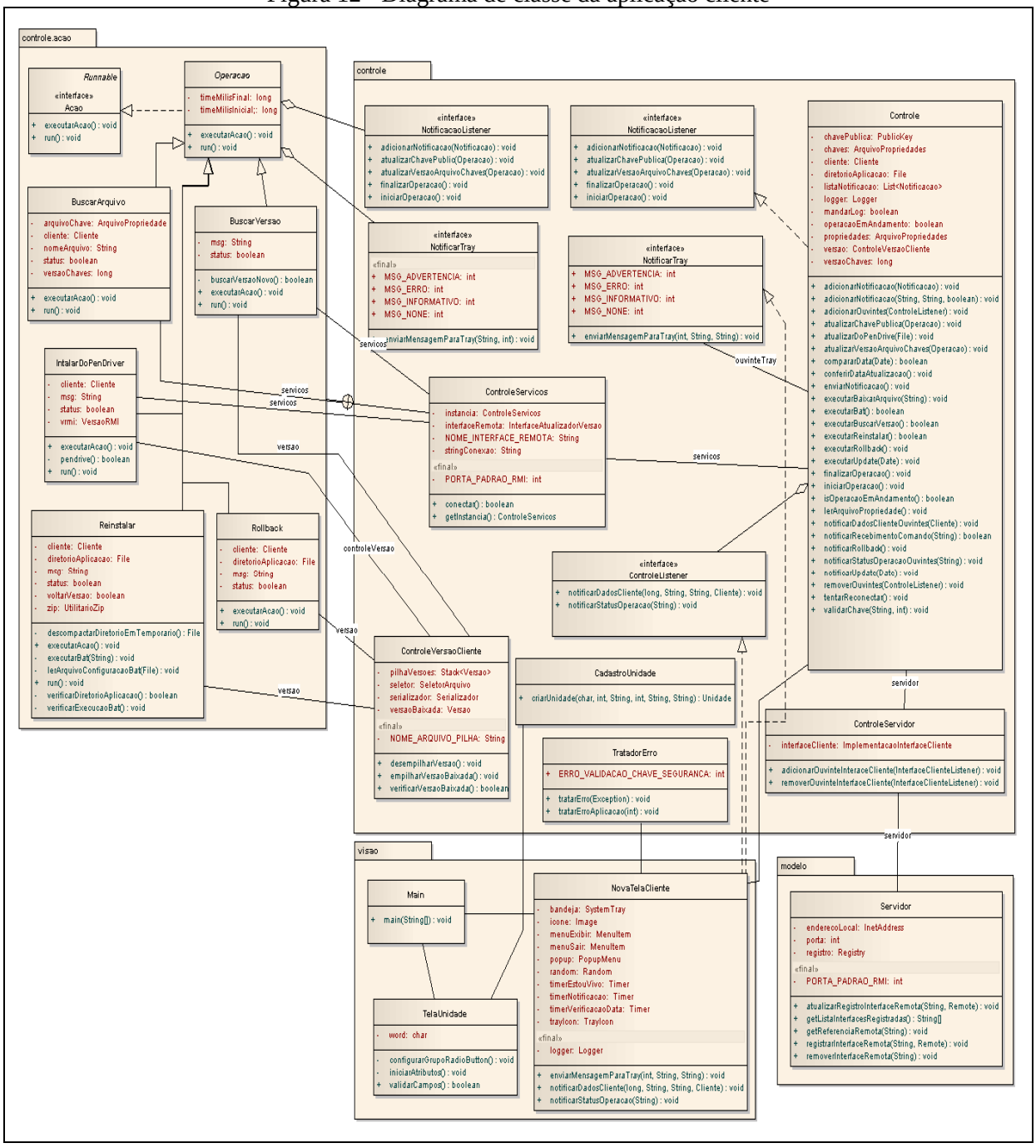
A interface `Acao` define como o método que irá conter a lógica de négocio da respectiva operação deverá ser realizada. Esta interface herda de *java.lang.Runnable*, pois todas as classes que pertencerem a sua linha hereditária serão responsáveis por iniciar uma nova linha de execução.

A classe abstrata `Operacao` realiza a interface `Acao`, sendo esta responsável por realizar as respectivas notificações aos ouvintes registrados. Esta classe define a estrutura de uma operação.

A classe `Rollback` é sub-classe de `Operacao`, sendo responsável por definir a regra de negócio da operação de voltar versão e gerar as notificações pertinentes aos ouvintes registrados.

A classe `BuscarVersao` é sub-classe de `Operacao`, sendo responsável por definir a regra de negócio da operação de baixar versão. Isto envolve realizar a conferência e validação do hash (MD5) do arquivo e gerar as notificações pertinentes aos ouvintes registrados.

Figura 12 - Diagrama de classe da aplicação cliente



A classe `Instalar` é sub-classe de `Operacao`, sendo esta responsável por definir as regras de negócio da operação de instalação de uma versão baixada, assim como gerar as notificações pertinentes aos ouvintes registrados.

A classe `Reinstalar` é sub-classe de `Operacao`, sendo esta responsável por definir as regras de negócio da operação de reinstalação da versão atualmente instalada e gerar as notificações pertinentes aos ouvintes registrados.

A classe `InstalarDoPenDrive` é sub-classe de `Operacao`, sendo esta responsável por definir as regras de negócio da operação de instalação de uma versão exportada pela aplicação servidora e gerar as notificações pertinentes aos ouvintes registrados.

A classe `BuscarArquivo` é sub-classe de `Operacao`, sendo esta responsável por definir as regras de negócio da operação de buscar um arquivo, utilizada para realizar o *download* da chave pública da aplicação servidora e o conjunto de chaves de segurança.

O pacote `visao` contém as classes responsáveis pela interface gráfica da aplicação, sendo estas: `Main`, `TelaUnidade`, `NovaTelaCliente`.

O pacote `modelo` contém as classes de modelo da aplicação: `Servidor` e `Chave`.

A classe `Chave` define as regras de formação, geração e validação de uma chave de segurança.

A classe `Servidor` define as regras de negócio do servidor RMI da aplicação cliente, assim como meios de registrar uma interface remota.

O pacote `controle` contém as classes e interfaces responsáveis pela gerência e controle dos recursos utilizados pela aplicação servidor, sendo elas: `Controle`, `ControleServidor`, `ControleVersaoCliente`, `ControleServicos`, `NotificarTray` e `NotificacaoListener`.

A interface `NotificacaoListener` é responsável por definir os eventos que serão gerados quando uma notificação tiver que ser enviada para o servidor.

A interface `NotificacaoTray` é responsável por definir os eventos que geram notificações na bandeja do sistema.

A classe `ControleVersaoCliente` é responsável por controlar e persistir as informações referentes à pilha de versões do cliente, assim como os dados da *release* atual.

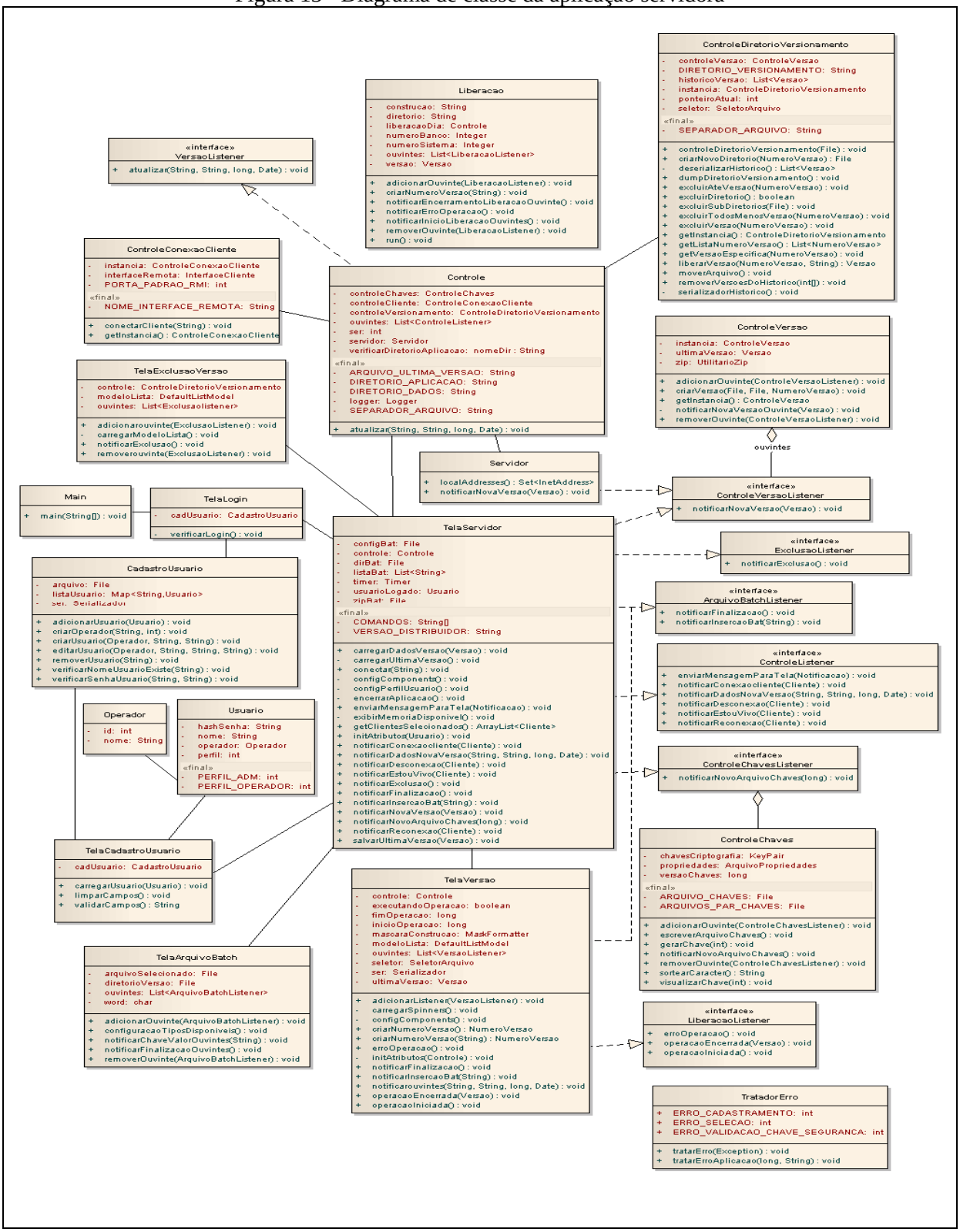
A classe `ControleServidor` é responsável por controlar e gerenciar as atividades do servidor cliente, tais como registrar o servidor no serviço de nomes (`rmiregistry`) e registrar as interfaces remotas.

A classe `ControleServico` é responsável por controlar, estabelecer e gerenciar a conexão com a aplicação servidora.

3.2.7 Diagrama de classes da aplicação servidora

Nesta seção será visto o pacote `Servidor`, onde é definida a aplicação servidora. O pacote é composto por sete interfaces e dezessete classes, como pode ser visto na figura 13. Nelas estão a estrutura, regras de negócio e modo de operação da aplicação.

Figura 13 - Diagrama de classe da aplicação servidora



A interface `ControleVersaoListener` define os métodos para gerar eventos quando uma nova *release* for liberada.

A interface `ExclusaoListener` define os eventos que devem ser gerados quando uma *release* for excluída do diretório de versionamento.

A interface `ArquivoBatchListener` define os eventos necessários para agregar um arquivo tipo `.bat` na *release* durante uma operação de liberação.

A interface `ControleListener` define os eventos que devem ser gerados para a atualização das interfaces da aplicação servidora.

A interface `ControleChavesListener` define os eventos que devem ser lançados quando um conjunto de chaves for gerado.

A interface `VersaoListener` define os eventos que notificam dados de uma nova notificação, tais como *hash*, data, tamanho e nome.

A classe `ControleVersao` é responsável por gerenciar e controlar o processo de liberação.

A classe `ControleChaves` é responsavel por gerenciar e controlar a geração da chaves de segurança.

A classe `ControleDiretorioVersionamento` é responsável por gerenciar e controlar o diretório onde as *releases* são geradas e mantidas.

A classe `ControleConexaoCliente` é responsavel por estabelecer as conexões com os clientes da aplicação servidora.

A classe `Controle` é responsável por controlar e manter as outras entidades de controle, além de gerenciar de forma macro as regras de negócio as entidades menores que compõem a aplicação.

A classe `Liberacao` é responsável por efetuar o processo de empacotamento, numeração e geração do *hash* MD5 de uma *release* liberada.

A classe `Servidor` é responsável por manter um servidor RMI, que atende às invocações remotas realizadas pelas aplicações clientes.

3.3 IMPLEMENTAÇÃO

A seguir são apresentadas as técnicas, ferramentas utilizadas e a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

Para o desenvolvimento da solução foi utilizada a linguagem de programação Java em conjunto com o ambiente integrado de desenvolvimento Netbeans na sua versão 7.2 e o JDK 1.6.

Foi utilizado o *middleware* RMI para implementar a arquitetura de cliente/servidor com um bom grau de abstração para o desenvolvimento. Utilizá-lo permitiu agregar algumas características como transparência e escalabilidade. O grau de abstração pode ser observado no quadro 20, onde apresenta-se um trecho de código pertencente à classe `ControleConexaoCliente` da aplicação servidora, onde realiza-se uma conexão com uma aplicação cliente.

Quadro 20 - Método servidor que realiza conexão com os clientes

```
public void conectarCliente(String ip) throws NotBoundException,
MalformedURLException, RemoteException {
      String enderecoCliente = "//"+ip+":"+1098+"/"+NOME_INTERFACE_REMOTA;
      System.out.println("Solicitando objeto remoto no endereço:
"+enderecoCliente);
      this.interfaceRemota = (InterfaceCliente) Naming.lookup(enderecoCliente);
      System.out.println("Interface remota: " + this.interfaceRemota.toString());
      System.out.println("Cliente conectado!");
}
```

Neste trecho pode-se observar o momento em que realiza-se a montagem da *string* de conexão, assim como a solicitação à referência remota do cliente.

A modularização das aplicações permitiu que fossem definidas classes utilitárias que oferecem serviços comuns a ambas as aplicações (cliente e servidor). Uma destas classes é a `UtilRSA`, utilizada na aplicação servidora quando gera-se um conjunto de chaves de segurança. Estas chaves são criptagrafadas antes de serem persisitidas em arquivo. A sua utilização pode ser vista no quadro 21, onde é apresentado o trecho de código responsável por escrever o arquivo.

Quadro 21 - Método que escreve o conjunto de chaves criptografadas

```
      public     void     escreverArquivoChaves()     throws     IOException,
NoSuchAlgorithmException,    NoSuchPaddingException,    InvalidKeyException,
IllegalBlockSizeException, BadPaddingException{
      this.propriedades = new ArquivoPropriedades();
              String sPosicao = null;
      String chave = null;
              this.versaoChaves++;
      this.propriedades.adicionarPropriedade("Versao",
this.versaoChaves+"");
      for (int i = 0; i < 100; i++) {
                  sPosicao = i+"";
                  if(sPosicao.length() < 2){
                      sPosicao = "0" + sPosicao;
                  }//fim if
                  System.out.println("sPosicao: "+sPosicao);
      chave = gerarChave(i);
                  System.out.println("chave: "+ chave+"\n");
                  this.propriedades.adicionarPropriedade(sPosicao,
UtilRSA.cifrarMensagem(chave,this.chavesCriptografia.getPrivate()));
              }//fim for
              this.propriedades.persistirPropriedades("chaves.dat", "");
              notificarNovoArquivoChaves();
          }
```

Outra característica interessante agregada à solução foi a capacidade da aplicação cliente identificar quando a aplicação servidora está fora do ar. Utiliza-se de chamadas remotas de controle para implementar a técnica de *heartbeat*, para detectar quedas do servidor e forçar a reconexão assim que possível. Como pode ser visto no trecho de código da classe `NovaTelaCliente` apresentada no quadro 22. Este trecho pode ser encontrado no método `actionListener` registrado para os *timers* de verificação.

Quadro 22 - Identificação de queda do servidor na classe `NovaTelaCliente`

```
if (e.getSource() == this.timerEstouVivo) {
    try {
        System.out.println("Notificando estou vivo!");
this.controle.getServicos().getInterfaceRemota().notificarEstouVivo(this.controle.g
etCliente());
        this.timerEstouVivo.restart();
    } catch (Exception erro) {
         System.out.println("Servidor esta fora do ar");
         try {
             this.controle.tentarReconectar();
         } catch (NotBoundException ex1) {
         } catch (RemoteException ex1) {
         } catch (MalformedURLException ex1) {
         } catch (UnknownHostException ex1) {
         }//fim catch
    }//fim catch
}
```

3.3.2 Operacionalidade da implementação

A operacionalidade da aplicação é apresentada com base nos casos de uso descritos, sendo apresentadas *screenshots* da aplicação em execução e descrição dos passos para melhor entendimento.

3.3.2.1 Aplicação cliente

Nesta seção será apresentada a operacionalidade da aplicação cliente, assim como sua configuração e instalação. Esta aplicação é distribuída em uma pasta contendo o executável da aplicação, um diretório de dependências e um arquivo de configuração. Deve-se realizar a cópia da aplicação para um diretório, que será a sua pasta de instalação.

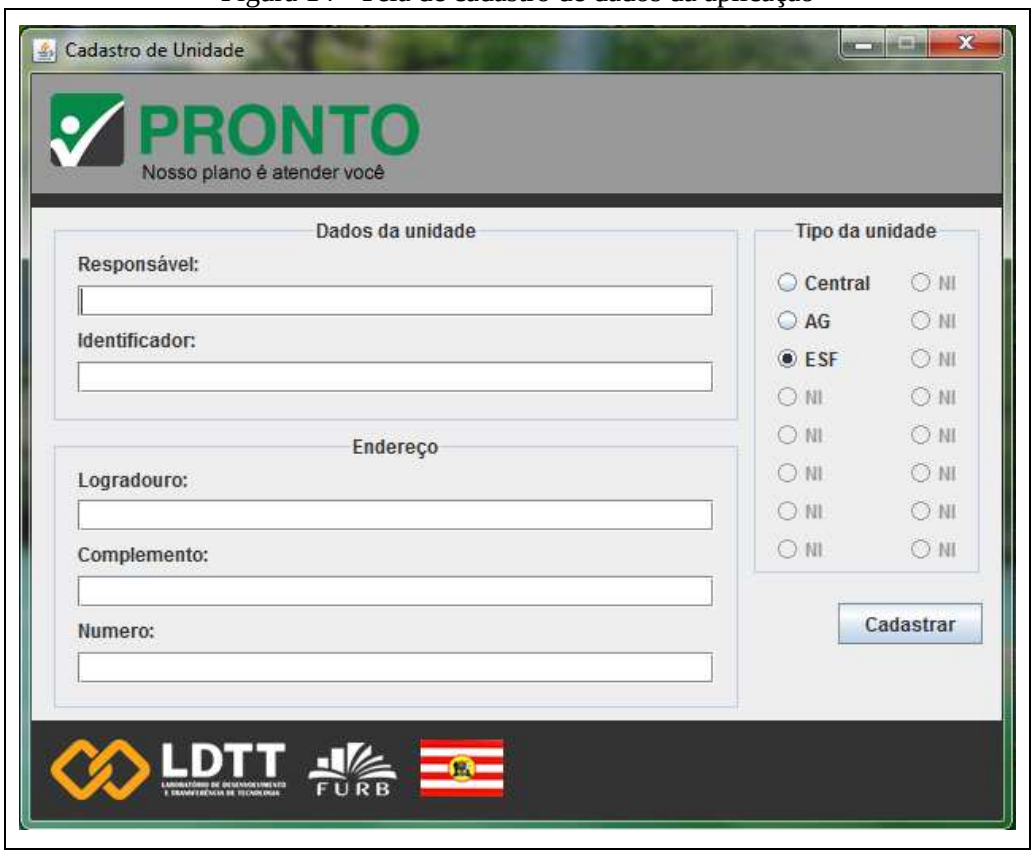
Após ser copiada faz-se necessário configurar as propriedades da aplicação. Para tanto basta editar o arquivo `Propriedades.conf`. Neste arquivo serão editados dois campos: `dirApp` e `ipServidor`. O campo `dirApp` indica o diretório da aplicação alvo ou que será gerenciado pela aplicação. No campo `ipServidor` deve-se informar o endereço IP da aplicação servidora. Estes dois campos deverão parecer com o apresentado no quadro 23.

Quadro 23 - Campos do arquivo `Propriedades.conf`

```
dirApp=C\:\\temp\\tcc
ipServidor=201.54.201.34
```

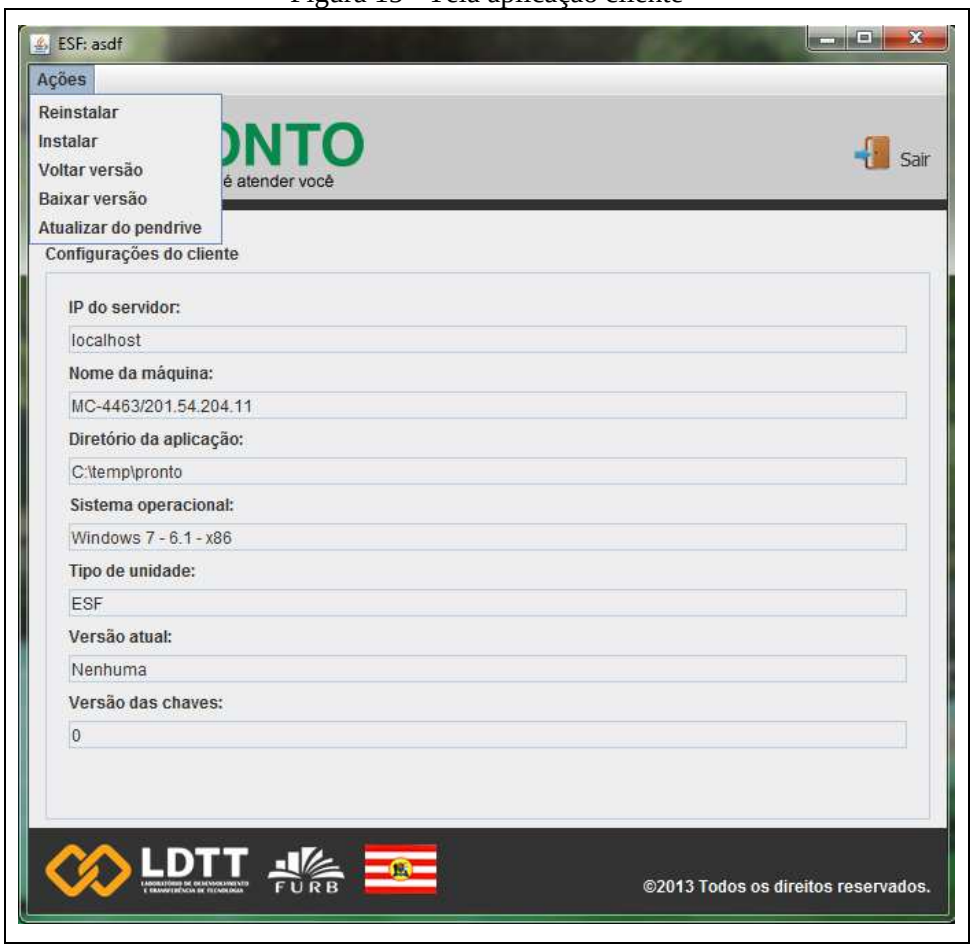
Efetuados estes passos, a aplicação está pronta para ser executada. Inicialmente ela irá solicitar um breve cadastro com dados referentes à própria aplicação, como pode ser vsito na figura 14.

Figura 14 - Tela de cadastro de dados da aplicação



Após o preenchimento do cadastro, a aplicação será lançada para o *system tray* (bandeja do sistema), a qual pode ser acessada na lateral direita da barra de tarefas do windows. Clicando no seu ícone e selecionando a menu “Exibir” aparecerá a tela apresentada na figura 15, contendo os dados do cliente.

Figura 15 - Tela aplicação cliente



No parte inferior da tela da aplicação podem ser vistos as mensagens referentes às operações que a aplicação está executando. Estas mesmas mensagens podem ser visualizadas em notificações na barra de tarefas do Windows quando a aplicação está executando em segundo plano.

As operações que podem ser realizadas encontram-se disponíveis na barra de tarefas da aplicação por meio do menu “Ações”. Estas são as mesmas ações que podem ser executadas remotamente pela aplicação servidora, exceto que para serem executadas localmente elas devem ser validadas por meio de uma chave de segurança, solicitada em uma caixa de diálogo logo após a seleção do menu da respectiva ação.

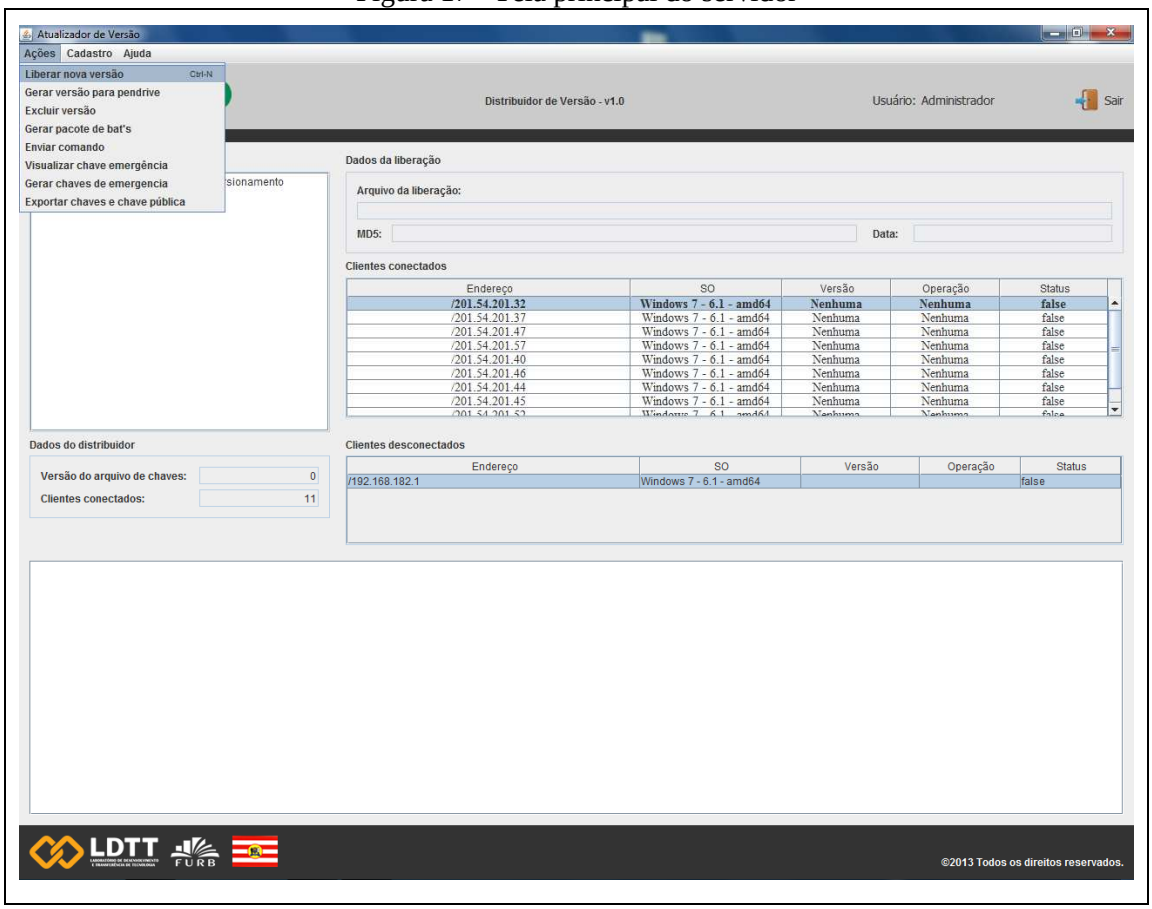
3.3.2.2 Aplicação servidora

Nesta seção será apresentada a operacionalidade da aplicação servidora. Esta aplicação é distribuída em uma pasta contendo o executável da aplicação e um diretório de dependências. Deve-se realizar a cópia da aplicação para um diretório, que será a sua pasta de

instalação. Ao ser executada, a aplicação apresentará a tela de login conforme pode ser visto na figura 16 e após aparecerá a tela apresentada na figura 17.

Figura 16 - Tela de login

Figura 17 - Tela principal do servidor

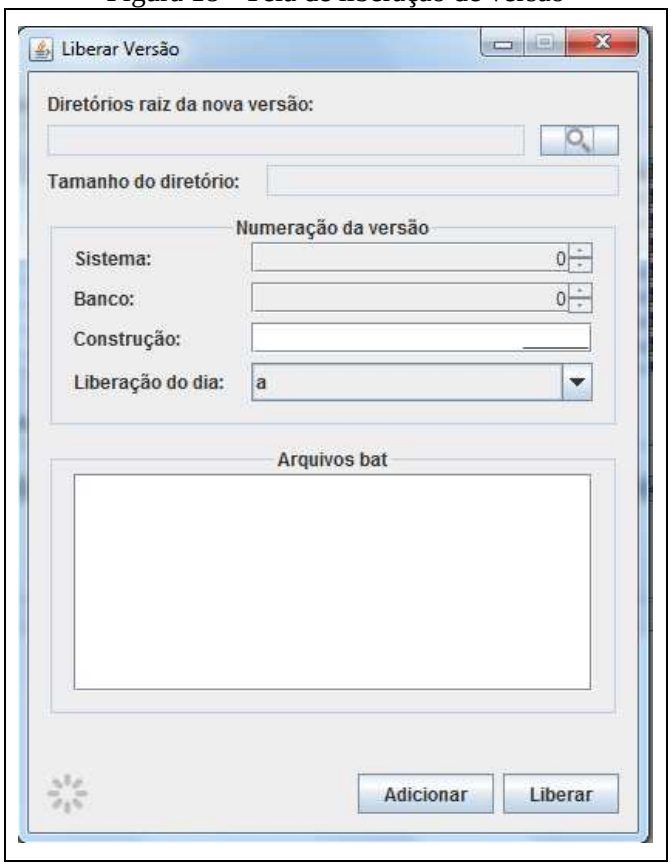


Esta é a tela principal da aplicação servidora. Nela estão dispostas na região central da tela as informações sobre os clientes que estão conectados e os que perderam conexão. Nesta execução pode-se perceber que onze clientes estão conectados ao servidor, sendo estes listados na tabela de clientes conectados. Todos estes estão aptos a receber comandos do

servidor. Na parte inferior está disposta a área de notificação dos clientes. Já do lado esquerdo da tela estão dispostas as informações referentes ao histórico de releases liberados.

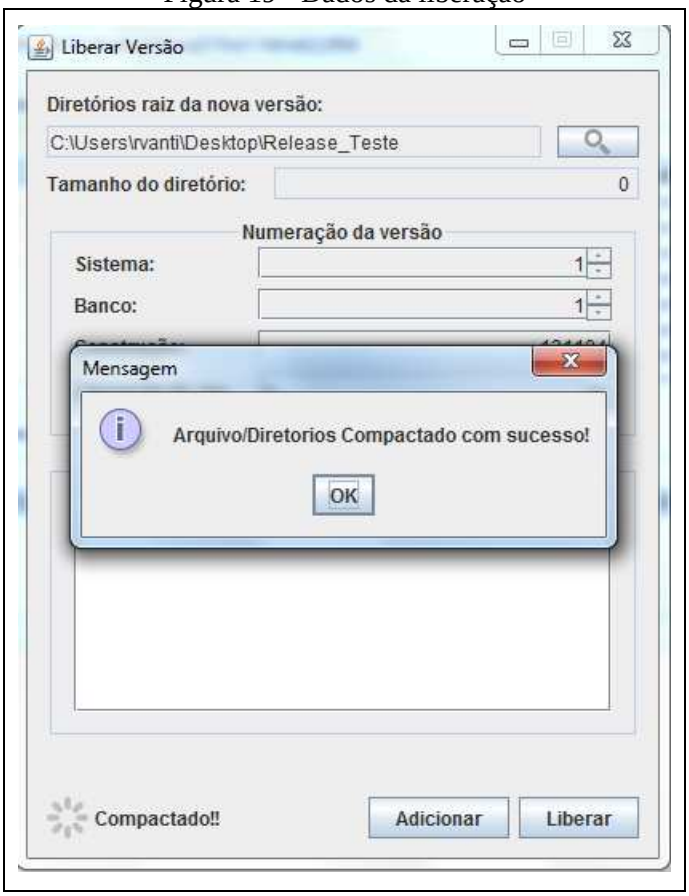
As operações que o servidor pode realizar estão expostas no menu “Ações” na barra de tarefas da aplicação. Inicialmente uma release deve ser liberada para que seja possível realizar as outras principais operações. Na figura 18 pode ser vista a tela de liberação de versão.

Figura 18 - Tela de liberação de versão



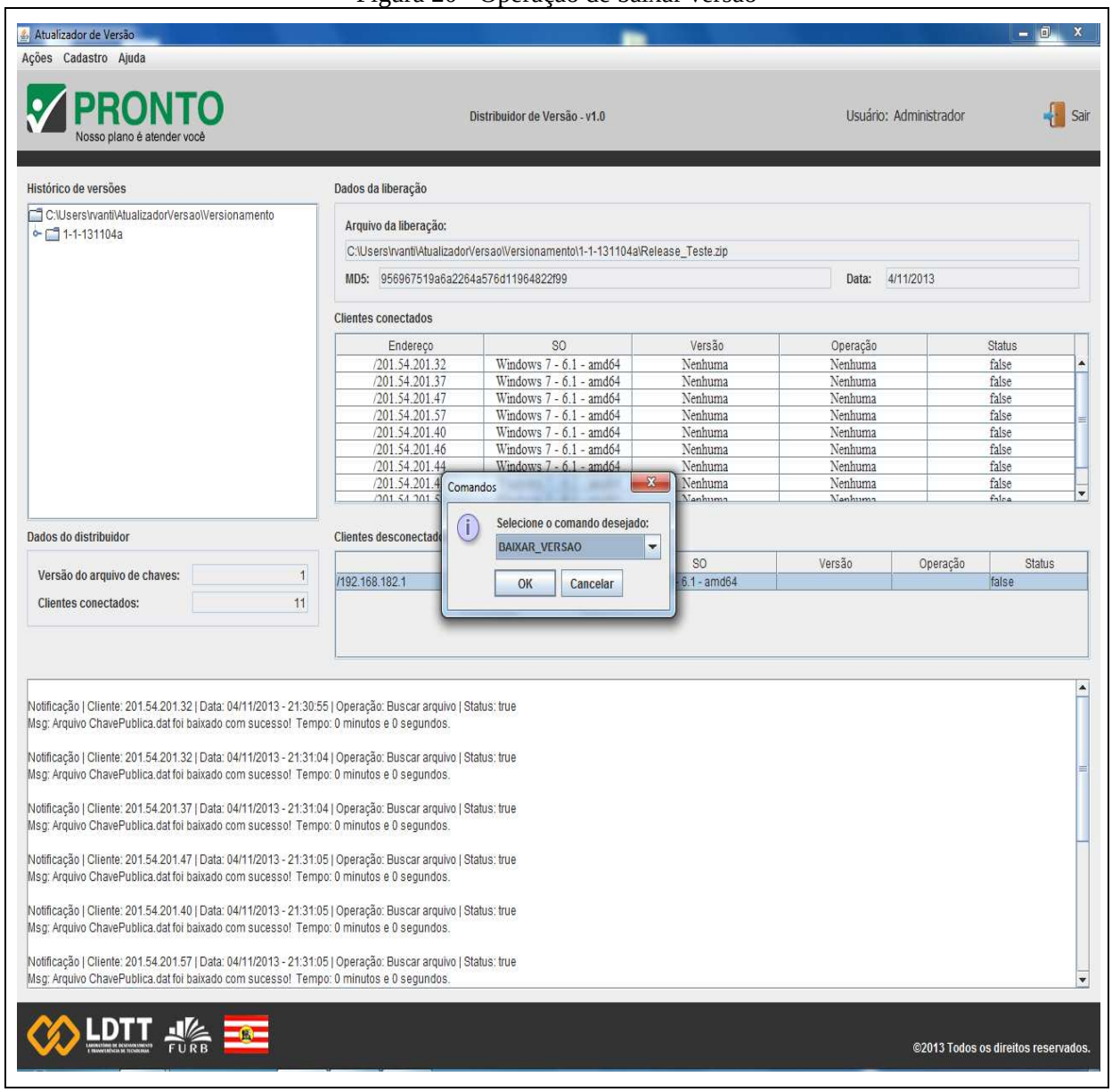
Nesta tela devem ser inseridas as informações referentes ao versionamento da *release* que será liberada, assim como o diretório da aplicação alvo que deve ser empacotado para gerar uma liberação. Este processo pode demorar alguns minutos dependendo do tamanho da aplicação. Na figura 19 podem ser vistos os dados da versão liberada nesta execução.

Figura 19 - Dados da liberação



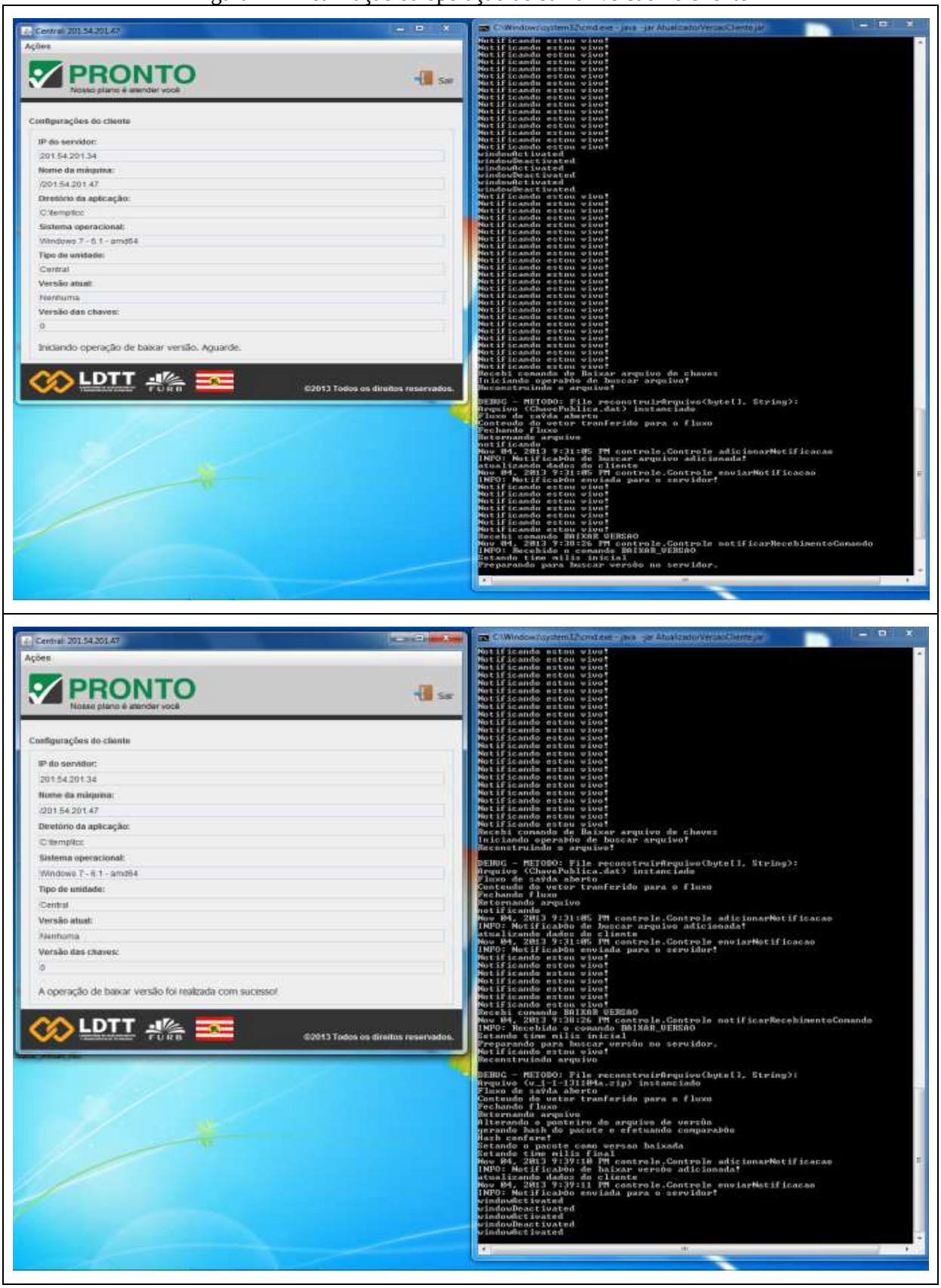
Nesta execução foi liberada uma *release* versionada com a numeração `1-1-131104a` conforme pode ser visualizado na árvore que controla o histórico de versões. Após realizar a liberação de uma *release*, pode-se realizar a operação de baixar versão, o que pode ser visto na figura 20.

Figura 20 - Operação de baixar versão



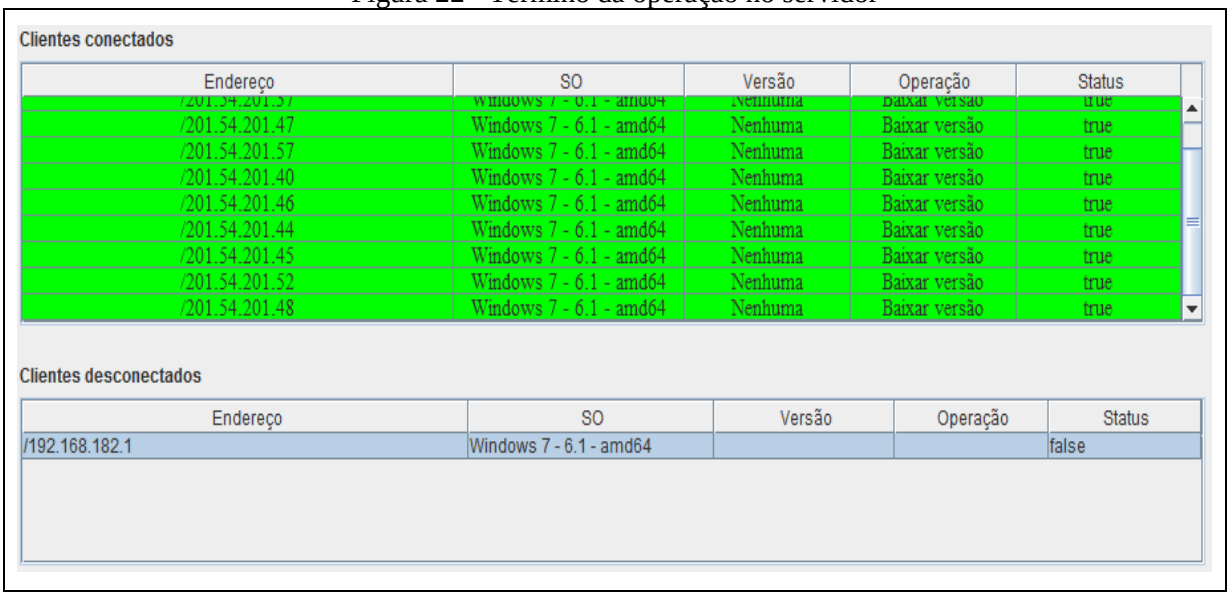
As operações são realizadas por padrão para todos os clientes. Portanto, se não houver pelo menos um cliente selecionado na lista de clientes conectados, a operação selecionada será realizada para todos os clientes listados. O início e o término da operação no cliente podem ser visualizados na figura 21.

Figura 21 - Realização da operação de baixar versão no cliente



O cliente gera as notificações das operações que realiza sendo que elas podem ser visualizadas na área de notificação da aplicação servidora conforme figura 22.

Figura 22 - Término da operação no servidor



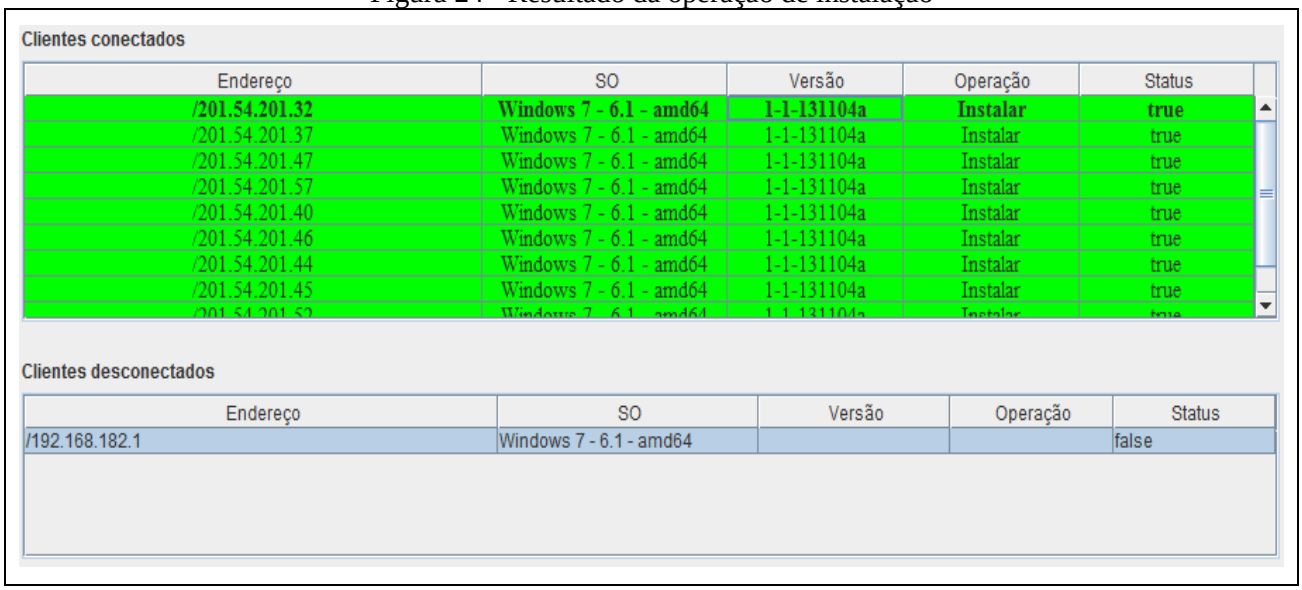
Quando uma operação é realizada com sucesso, o fundo da linha do cliente assume a cor verde informando que a operação foi bem sucedida.

A operação de instalação pode ser executada em seguida, conforme pode ser visualizada na figura 23.

Figura 23 - Enviando o comando de instalação

O resultado desta operação pode ser visualizada na figura 24, onde a coluna operação assumiu o valor de “Instalar” e a coluna versão assumiu o valor de `1-1-131104a`. Isto indica que a operação de instalação foi relizada para a release `1-1-131104a` para o determinado cliente.

Figura 24 - Resultado da operação de instalação



3.4 RESULTADOS E DISCUSSÃO

O presente trabalho apresentou o desenvolvimento de um sistema distribuído para distribuição de *releases* de *software*.

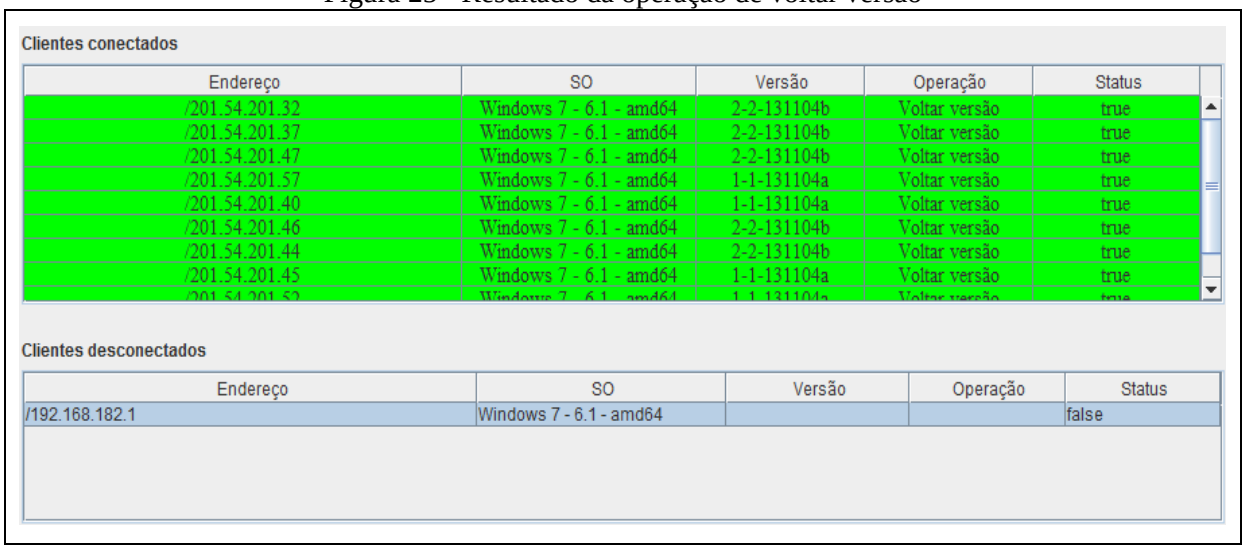
Na solução proposta foi utilizado um *middleware* de comunicação de objetos distribuídos para realizar a implementação da arquitetura cliente/servidor. O mesmo permitiu atingir objetivos como escalabilidade, transparência e aplicação de princípios de tolerância a falhas, por meio da aplicação da técnica de *heartbeat*.

A solução desenvolvida é composta de duas aplicações *desktop*, onde a aplicação servidora assume um perfil mais ativo, sendo esta responsável por efetuar envios de comandos para clientes que estejam conectados a ela. A aplicação cliente possui um perfil passivo, sendo esta executada preferencialmente em segundo plano. Porém também é possível efetuar operações a partir dela, sempre solicitando-se uma chave de segurança para homologar a operação.

Em um dos testes realizados utilizou-se onze computadores com a aplicação cliente instalada e um computador executando a aplicação servidora. Este teste foram realizados em ambiente de rede local. Os resultados obtidos com a escalabilidade mostraram-se satisfatórios para o problema inicialmente proposto, além de atestar a transparência do *middleware*. O teste teve início após a preparação do ambiente (instalação das aplicações). Iniciou-se liberando uma *release* seguida das operações de baixar versão, instalação, reinstalação. Posteriormente

liberou-se uma nova versão seguida de uma nova instalação, para que fosse possível executar os testes da operação de voltar versão. O resultado da operação de voltar versão pode ser visto na figura 25.

Figura 25 - Resultado da operação de voltar versão



Além deste teste a aplicação foi testada em ambiente piloto da aplicação PRONTO, projeto de gestão de saúde pública que esta sendo desenvolvido pelo Laboratório de Desenvolvimento e Tranferência de Tecnologia (LDTT) em conjunto com prefeitura municipal de Blumenau. O presente trabalho vem sendo testado ao longo dos últimos seis meses, em um ambiente composto por oito unidades de saúde com *links* de comunicações diversos, tendo durante este período sido liberadas aproximadamente oito *releases* por meio da aplicação.

No quadro 24 são apresentadas as principais características da aplicação em relação aos trabalhos correlatos.

Quadro 24 - Quadro comparativo

Funcionalidade	Distribuidor de versão	uBuild	uDeploy	Java Web Start
Independência da plataforma de desenvolvimento	X	-	-	-
Construção da versão da aplicação alvo	-	X	-	-
Mais de um ambiente de *deploy*	-	-	X	-
Atualização automática	-	-	-	X
Realiza o *deploy* da *release*	X	-	X	X
Realiza a instalação da *release*	X	-	-	X
Realiza a *rollback* (voltar versão)	X	-	X	-

4 CONCLUSÕES

Atualmente faz-se necessário agilizar a forma como os produtos de *software* chegam aos seus usuários finais, seja por necessidades de correções, lançamentos de novas funcionalidades ou mesmo por vantagem comercial.

Embora esse processo ainda seja manual, ele pode ser mapeado de modo que seja possível automatizar partes significativas do mesmo. O presente trabalho vem a atuar neste sentido, de automatizar o versionamento, entrega e instalação de *software*, de modo a ser escalável, para conseguir atender um grande número de usuários finais.

Os objetivos foram atingidos e os resultados obtidos na fase de testes se enquadraram dentro do esperado para o ambiente de testes ao qual a solução foi aplicada, sendo esta capaz de atender á demanda existente. Neste sentido, as tecnologias e conceitos de sistemas distribuídos estudados auxiliaram na definição da arquitetura da aplicação, assim como nas características que foram agregadas a ela.

Utilizar um *middleware* de comunicação de objetos distribuídos ofereceu uma plataforma eficiente para a comunicação das aplicações cliente e servidor, com um bom grau de abstração para o desenvolvimento. A solução também mostrou-se escalável, pois conseguiu atender uma demanda inicial de oito clientes, que posteriormente aumentou para onze clientes. Este aumento da quantidade de clientes ocasionou um aumento da demanda de recursos da aplicação servidora, pois conforme a quantidade de clientes aumenta a demanda por *deploy* aumenta na mesma proporção.

A utilização do *middleware* também permitiu atingir o objetivo de heterogeneidade, pois a solução operou durante a fase de testes em um cenário onde havia diferentes tipo de *links* de comunicação. O cenário completo de distribuíção do sistema PRONTO conta com onze servidores de aplicação executando a aplicação cliente, sendo que três destes possuem *link's* de comunicação de *Asymmetric Digital Subscriber Line* (ADSL), quatro com *link's* de fibra óptica e quatro com *Virtual Private Network* (VPN). Foram liberados oito *releases* ao longo do período de testes. Desta forma, foi possível operar em um ambiente com uma infra-estrutura heterogênea.

Os *releases* são gerados e mantidos em um diretório gerenciado pela própria aplicação servidora. Assim evita-se que o usuário tenha que executar alguma operação manualmente, o que poderia ocasionar a perda de informações e/ou perdas de *releases*. Isto garante a integridade das informações com que a aplicação trabalha.

A implementação da técnica do *heartbeat* possibilitou atingir o objetivo de tolerância a falhas, permitindo aos próprios clientes identificar a queda do servidor e assim reestabelecer a conexão assim que possível. Desta forma, não é necessário nenhum processamento adicional por parte da aplicação servidora para recuperar as conexões com os clientes anteriormente conectados.

4.1 EXTENSÕES

Como sugestões de extensões para trabalhos futuros indica-se:

a) Permitir manter mais de uma aplicação ao mesmo tempo;
b) selecionar um *release* diferente da última liberação para ser distribuída;
c) implementar atualizações automáticas;
d) permitir gerenciar outros itens de configuração;
e) implementar comunicação SSL usando RMI.

REFERÊNCIAS BIBLIOGRÁFICAS

ARAUJO, Everson S. **Introdução a segurança da informação**. [S.l.], 2008. Disponível em: <http://everson.com.br/files/Introdu%C3%A7%C3%A3o%20%C3%A0%20Seguran%C3%A7a%20da%20Informa%C3%A7%C3%A3o.pdf>. Acesso em: 9 out. 2013.

BURKE, Bill; MONSON-HAEFEL, Richard. **Enterprise JavaBeans 3.0**. 5. ed. São Paulo: Pearson Prentice Hall, 2007. 538 p.

CARVALHAL, Carlos R. G.; PINTO, Arménio A. A. P.; ROMEIRO, Manuel A. de O. **Sistema de atendimento telefónico automatizado dos SDUA (SATA-SDUA).** [S.l.], 2001. Disponível em: <http://revistas.ua.pt/index.php/revdeti/article/view/1558/1440>. Acesso em: 2 ago. 2013.

CASTRO, Márcio; RAEDER, Mateus; NUNES, Thiago. **RMI:** uma visão conceitual. Disponível em: <http://www.inf.pucrs.br/~gustavo/disciplinas/sd/material/Artigo_RMI_Conceitual.pdf>. Acesso em: 9 out. 2013

COULOURIS, George F.; DOLLIMORE, Jean; KINDBERG, Tim. **Distributed systems**: concepts and design. 4th ed. USA: Addison Wesley, 2005.

DANTAS, Mário. **Computação distribuída de alto desempenho:** redes, clusters e grid computacionais. Rio de Janeiro: Axcel, 2005.

ENGHOLM JÚNIOR, Hélio. **Engenharia de software na prática**. São Paulo: Novatec, 2010. 438 p.

GUEDES, Luis A. **Objetos distribuídos:** programação distribuída orientada a objetos. Disponível em: <http://www.dca.ufrn.br/~affonso/DCA2401/2004_1/aulas/objetos_distribuidos.pdf>. Acesso em: 9 out. 2013.

HAGHIGHAT, Kevin. **Java web start &JNLP**. [S.l.], 2007. Disponível em: <kevinhaghighat.com/webtutorials/jnlp/jws_jnlp.pdf>. Acesso em: 30 ago. 2013.

HIRAMA, Kechi. **Engenharia de software**: qualidade e produtividade com tecnologia. Rio de Janeiro: Elsevier: Campus, 2012. 210 p.

HORSTMANN, Cay S.; CORNELL, Gary. **Core Java 2** – recursos avançados. São Paulo: Makron Books, 2001. 823 p.

LEMOS, Manoel. **Criptografia, números primos e algoritmos**. [S.l.], 2010. Disponível em: <http://www.impa.br/opencms/pt/biblioteca/pm/PM_04.pdf>. Acesso em: 9 out. 2013.

LÓPEZ, Julio**. Introdução à criptografia**. [S.l.], 2012. Disponível em: <http://www.lca.ic.unicamp.br/~jlopez/intro889.pdf>. Acesso em: 9 out. 2013.

MAGALHÃES, Ivan L.; PINHEIRO, Walfrido B. **Gerenciamento de serviços de TI na prática**: uma abordagem com base na ITIL. São Paulo: Novatec, 2007. 667 p.

MARCELINO, Leonardo. **Java web start**. [S.l.], 2004. Disponível em: <www.guj.com.br/content/articles/jws/jws.pdf>. Acesso em: 30 ago. 2013.

NUNES, Vanessa B.; FALBO, Ricardo de A. **Uma ferramenta de gerência de configuração integrada a um ambiente de desenvolvimento de software.** [S.l.], 2006. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/sbqs/2006/016.pdf>. Acesso em: 9 out. 2013

OLIVEIRA, Ronielton R. **Criptografia simétrica e assimétrica:** os principais algoritmos de cifragem. [S.l.], 2012. Disponível em: <http://www.ronielton.eti.br/publicacoes/artigorevistasegurancadigital2012.pdf>. Acesso em: 9 out. 2013.

ORACLE. **Overview of RMI applications**. [S.l.], 2001. Disponível em: <http://docs.oracle.com/javase/tutorial/rmi/overview.html>. Acesso em: 29 maio 2013.

_____. **Java web start overview**. [S.l.], 2005. Disponível em: <www.oracle.com/technetwork/java/javase/jws-white-papper-150004.pdf>. Acesso em: 30 ago. 2013.

PFLEEGER, Sharin L. **Engenharia de software:** teoria e prática. 2. ed. São Paulo: Pearson Prentice Hall, 2004. 535 p.

PRATA, Paula. **Programação de sistemas distribuídos em Java**. Covilhã, 2008. Disponível em: <http://www.di.ubi.pt/~pprata/spd/SD_08_09_T09.pdf >. Acesso em: 4 ago. 2013.

SCHMITT, Marcelo A. R. **Criptografia de chaves públicas**. [S.l.], 2001. Disponível em: <http://www.rnp.br/wrnp2/2001/palestras_middleware/pal_middl_02.pdf>. Acesso em: 9 out. 2013.

SOMMERVILLE, Ian. **Engenharia de software**. 6. ed. São Paulo: Addison Wesley, 2003. 592 p.

_____. **Engenharia de software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011. 529 p.

TAING, Nguonly. **Java RMI overwiew**. [S.l.], 2011. Disponível em: <http://lycog.com/distributed-systems/java-rmi-overview>. Acesso em: 3 set. 2013.

TSUI, Frank; KARAM, Orlando. **Fundamentos de engenharia de software**. 2. ed. Rio de Janeiro: LTC, 2013. 221 p.

URBANCODE. **Build, deploy, release automation**. [S.l.], 2013. Disponível em: <http://www.urbancode.com>. Acesso em: 29 maio 2013.

WAZLAWICK, Raul S. **Engenharia de software**: conceitos e práticas. Rio de Janeiro: Elsevier: Campus, 2013. 343 p.