UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

PROTÓTIPO DE UM MÓDULO DE AUTO-PROTEÇÃO

PHILIPP ALBERT SCHROEDER

BLUMENAU
2008

2008/2-18

PHILIPP ALBERT SCHROEDER

PROTÓTIPO DE UM MÓDULO DE AUTO-PROTEÇÃO

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 Ciências da Computação — Bacharelado.

Prof. Paulo Fernando da Silva - Orientador

BLUMENAU
2008

2008/2-18

MÓDULO DE AUTO-PROTEÇÃO EM UM SISTEMA PEER-TO-PEER

Por

PHILIPP ALBERT SCHROEDER

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 – Orientador, FURB
Membro:	Prof. Francisco Adell Péricas – FURB
Membro:	Prof. Sérgio Stringari – FURB

Blumenau, 10 de fevereiro de 2009

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

AGRADECIMENTOS

Primeiramente a Deus, pelo seu imenso amor e graça, e por ter me dados condições de realizar este trabalho.

Aos meus pais, pela ajuda financeira e pelo suporte dado nestes anos de labuta.

Aos colegas e professores do curso de Bacharelado em Ciências da Computação, pela troca de conhecimentos e experiências.

Aos meus amigos, por sempre me ajudarem nas horas nas quais precisei deles.

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

Se você quer ser bem sucedido, precisa ter dedicação total, buscar seu último limite e dar o melhor de si mesmo.

Ayrton Senna

RESUMO

Este trabalho apresenta alguns conceitos básicos sobre a computação autonômica, bem como o desenvolvimento de um módulo de auto-proteção para sistemas em geral, capaz de identificar portas abertas e arquivos modificados recentemente. No desenvolvimento do módulo foi utilizado um dos conceitos de computação autonômica, a auto-proteção.

Palavras-chave: Auto-proteção. Computação autonômica.

ABSTRACT

This project presents some basics concepts about autonomic computing and the development of a self-protection module for any kind of systems, able to identify open doors and recently changed archives. In the development of this project, an autonomic computing concept had been used, the self-protection concept.

Key-words: Self-protection. Autonomic computing.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

CPU – *Central Processing Unit*

DoS – *Denial of Service*

IP – *Internet Protocol*

P2P – *Peer-to-Peer*

P2PIM – *Peer-to-Peer Instant Messenger*

RF – Requisito Funcional

RNF – Requisito Não-Funcional

TI – Tecnologia da Informação

UML – *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

Para Horn (2001), com o passar das décadas, o desenvolvimento do poder computacional e a proliferação de variados tipos de aparelhos computacionais cresceu em um nível exponencial. Esse crescimento fenomenal, fomentado com o advento da Internet, trouxe uma nova era de acessibilidade: a outras pessoas, a outros sistemas, e, sobretudo, a informação.

Essa explosão simultânea de informação e integração de tecnologias no dia-a-dia trouxe novas demandas de como as pessoas mantêm e gerenciam os sistemas computacionais. Sistemas complexos e simples exigem profissionais da área da Tecnologia da Informação (TI) qualificados. A demanda por esses profissionais tende a crescer muito nos próximos anos.

Como o acesso à informação torna-se onipresente através de computadores e aparelhos sem-fio, a estabilidade da atual infra-estrutura de sistemas e dados está em um risco crescente de falhas e quebras. O aumento da complexidade dos sistemas está alcançando um nível acima das habilidades de gerenciamento e segurança humanos. Esse aumento da complexidade somado à falta de profissionais de TI qualificados aponta para uma inevitável necessidade de automatizar inúmeras das funções associados com a computação.

Estas dificuldades criaram a necessidade de investigação de um novo paradigma, a computação autonômica, para projetar, desenvolver, implantar e gerenciar sistemas. Esse novo paradigma é baseado nos sistemas biológicos e em como eles enfrentam problemas de heterogeneidade, complexidade e incertezas. Um sistema autonômico possui quatro características marcantes: auto-configuração, auto-cura, auto-otimização e auto-proteção (SALEHIE; TAHVILDARI, 2005).

As características dos sistemas de TI que levaram à criação da computação autonômica são típicas de um sistema distribuído, possuindo uma heterogeneidade de software, hardware e middleware. Um tipo de sistema distribuído muito popular é o sistema *Peer-to-Peer* (P2P), utilizado para troca de arquivos entre os usuários. Mas a falta de segurança neste tipo de sistema afasta muitos usuários e inibe sua disseminação, bem como a sua diversificação (GASPARY, 2006).

Diante do exposto, neste trabalho foi desenvolvido um módulo para incrementar a segurança em sistemas P2P de troca de arquivos, dificultando a execução de arquivos não confiáveis e a leitura de arquivos por parte de terceiros não autorizados. Estas funções serão as encarregadas de inibir os ataques maliciosos externos, bem como procurar minimizar as

falhas de segurança existentes no sistema.

Com o módulo desenvolvido, os sistemas P2P serão mais seguros, uma vez que os atacantes externos terão dificuldades maiores para ler arquivos não autorizados. Além disso, os arquivos não confiáveis, que poderiam trazer problemas ao sistema e ao usuário, não serão executados.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um módulo de auto-proteção.

Os objetivos específicos do trabalho são:

a) monitorar portas abertas configuradas como não autorizadas pelo sistema e informar ao usuário;
b) monitorar o acesso a arquivos do diretório configurado pelo sistema e informar ao usuário;
c) impedir a execução de arquivos não confiáveis, cuja procedência seja dúbia.

1.2 ESTRUTURA DO TRABALHO

A seguir está exposto como o trabalho está estruturado.

Inicialmente vem a fundamentação teórica. Na fundamentação teórica são expostos os fundamentos da computação autonômica seguida por uma explanação a respeito dos sistemas P2P e também sobre ataques remotos de terceiros.

Após vem o desenvolvimento, onde são detalhados os requisitos do módulo, bem como a especificação do mesmo. É exposto também como o módulo foi implementado.

Finalizando estão as conclusões.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados aspectos teóricos relacionados ao trabalho. Primeiro são feitas considerações a respeito da computação autonômica. A seguir vem explanações sobre sistemas *peer-to-peer* e por fim são detalhados os ataques remotos de terceiros.

2.1 COMPUTAÇÃO AUTONÔMICA

O paradigma da computação autonômica é baseado nas funções autônomas do sistema nervoso central humano. O sistema autônomo humano utiliza neurônios que mandam mensagens indiretas para os órgãos em um nível subconsciente. Essas mensagens regulam temperatura, respiração e freqüência cardíaca sem um pensamento consciente. As implicações na computação são evidentes: uma rede de componentes organizados que dão aquilo que é necessário, quando for necessário, sem esforço mental ou físico (HORN, 2001).

Um sistema autonômico deve possuir um mecanismo de sobrevivência comparável ao de um ser vivo. Esse mecanismo de sobrevivência é caracterizado pelo fato do sistema possuir variáveis essenciais ao seu bom funcionamento. Caso ocorram mudanças nessas variáveis, o sistema deve reagir mudando seu comportamento, de modo a voltar para um ponto de equilíbrio. Esse ponto de equilíbrio é a condição necessária para a sobrevivência do sistema. (PARASHAR; HARIRI, 2005).

Segundo Horn (2001), os benefícios imediatos da implantação da computação autonômica em um sistema incluem uma dependência reduzida da intervenção humana para manter um sistema complexo, acompanhada por uma forte queda nos custos, experiência necessária do usuário simplificada, estabilidade, alta disponibilidade, alta segurança e erros reduzidos.

Um sistema autonômico pode ser descrito como possuidor de oito características, descritas a seguir.

2.1.1 Auto-conhecimento

Horn (2001) afirma que, para ser autonômico, um sistema computacional precisar conhecer-se. Seus componentes devem possuir um sistema de identificação. Como um sistema pode existir em vários níveis, um sistema autonômico precisa detalhar o conhecimento de seus componentes, a condição atual e suas conexões com outros sistemas e componentes, de modo que o componente possa se governar. Ele deve saber os recursos que possui, os que ele pode compartilhar e os que ele deve manter isolados.

Esta é uma definição de sistema que aparenta ser simples, quando sistema computacional significa uma máquina que ocupa uma sala inteira, ou quando significa centenas de pequenas máquinas em uma rede de uma companhia. Mas conectando-se essas centenas de máquinas aos milhões de máquinas na Internet, fazendo-as interdependentes, e possibilitando uma audiência global para seus recursos, via uma seleção de aparelhos sem fio que se proliferam no mundo, o conceito de sistema deixa de ser claro. Permitindo todos esses aparelhos a compartilhar ciclos de processamento, armazenamento e outros recursos, chega-se a uma situação que desafia qualquer definição de sistema.

É precisamente esse reconhecimento total do sistema que a computação autonômica requer. Um sistema não pode monitorar algo que ele não sabe que existe, ou controlar pontos específicos se seu domínio de controle não está definido.

Para construir essa habilidade em sistemas computacionais, regras claramente definidas encapsuladas em softwares agentes adaptáveis devem administrar a sua definição de sistema e suas interações com sistemas ao seu redor. Estes sistemas precisam também da capacidade de fundir-se automaticamente com outros sistemas para formar novos, mesmo que temporariamente, e quebrar-se em partes se forem requeridos sistemas discretos.

2.1.2 Auto-configuração

Horn (2001) afirma que um sistema autonômico deve configurar-se e reconfigurar-se sob condições variáveis e imprevisíveis. As configurações do sistema devem ocorrer automaticamente, assim como ajustes dinâmicos para esta configuração para melhor lidar com ambientes mutáveis.

Dar possíveis mutações em sistemas complexos pode tornar a configuração difícil e

demorada. Alguns servidores sozinhos apresentam centenas de alternativas de configurações. Administradores humanos nunca serão capazes de fazer a reconfiguração dinamicamente em um período curto de tempo, uma vez que há muitas variáveis para monitorar e ajustar.

Para ativar essa habilidade de configuração automática, um sistema pode vir a criar múltiplas imagens de softwares críticos, como um sistema operacional, e realocar seus recursos à medida que for necessário. Algoritmos adaptativos executando em tais sistemas poderiam aprender a melhor configuração para alcançar determinado nível de desempenho.

2.1.3 Auto-otimização

Horn (2001) afirma que um sistema autonômico nunca está parado, ele sempre procura por meios de aperfeiçoar seu funcionamento. Ele irá monitorar suas partes constituintes e ajustar o fluxo de trabalho para alcançar os objetivos pré-determinados.

Este esforço para otimizar-se é o único meio de um sistema computacional ser capaz de alcançar as complexas, e muitas vezes conflitantes, demandas de um negócio, seus consumidores, fornecedores e empregados. E como as prioridades que dirigem estas demandas estão em constantes mudanças, somente constante auto-otimização irá satisfazê-las.

Mas para ser capaz de otimizar-se, um sistema precisará de mecanismos de controle de retorno avançados para monitorar suas dimensões e tomar a ação apropriada. Controle de retorno é uma técnica antiga, mas são necessárias novas idéias para aplicá-lo na computação. É necessário responder a questões como quão freqüente o sistema toma ações de controle, quanta espera ele pode aceitar entre uma ação e seu efeito, e como tudo isto afeta a estabilidade geral do sistema.

2.1.4 Auto-cura

Conforme Horn (2001), um sistema computacional autonômico deve ser capaz de recuperar-se de rotinas e de eventos extraordinários que podem ter causado mau funcionamento de alguma parte. Ele deve ser capaz de descobrir problemas, ou problemas em potencial, e encontrar uma maneira alternativa de utilização dos recursos ou reconfigurar o sistema para mantê-lo em funcionamento. Ao invés da criação de partes de reposição, como acontece no corpo humano, a cura em um sistema computacional significa utilizar elementos

redundantes ou com baixa utilização para funcionar como peça de reposição.

Certos tipos de cura fazem parte da computação há muito tempo. Checagem e correção de erros é uma tecnologia com mais de 50 anos, que permite transmissões de dados na Internet, de modo que os mesmos permanecem confiáveis. Sistemas de armazenamento redundante permitem que dados possam ser recuperados mesmo quando partes do sistema de armazenamento falham.

A crescente complexidade dos ambientes de TI faz com que seja cada vez mais difícil localizar a causa de uma quebra, mesmo em ambientes relativamente simples. Em sistemas mais complexos, identificar as causas de falhas pede por uma análise de causas raiz, uma tentativa de examinar sistematicamente "o que fez o que para quem", de modo a encontrar a origem do problema. Mas como o serviço de restauração para o consumidor e a minimização de interrupções são preocupações primárias, uma abordagem orientada a ações é necessária em um sistema autonômico.

Inicialmente, respostas de cura dadas por um sistema autonômico seguirão regras geradas por especialistas humanos. Mas a medida que é inserida mais inteligência nos sistemas computacionais, eles irão começar a descobrir novas regras por si próprios que ajudarão a utilizar redundância do sistema ou recursos adicionais para recuperar e alcançar o objetivo principal, alcançar os objetivos especificados pelo usuário.

2.1.5 Auto-proteção

Segundo Horn (2001), um mundo virtual não é menos perigoso que o físico, portanto um sistema autonômico computacional deve ser um especialista em auto-proteção. Ele deve detectar, identificar e proteger-se contra vários tipos de ataques para manter a segurança e a integridade geral do sistema.

Antes da Internet, os computadores operavam como ilhas. Era muito fácil proteger os sistemas computacionais dos ataques conhecidos como vírus. Como o único meio utilizado para compartilhar arquivos e programas era o disquete, levava semanas ou meses para um vírus se espalhar. A conectividade do mundo em rede mudou tudo: ataques podem vir de qualquer lugar. Os vírus são disseminados muito rapidamente, em segundos, e em larga escala, uma vez que são designados para serem mandados automaticamente para outros usuários. O dano potencial para os dados e para a imagem de uma empresa é enorme.

Mais que simplesmente responder à falhas de componentes, ou executar verificações

periódicas para encontrar sintomas, um sistema autonômico precisará permanecer alerta, antecipar ameaças e tomar as ações necessárias. Estas atitudes têm como alvo dois tipos de ataques: vírus e intrusões de sistema por *hackers*.

Por mimetizar o sistema imunológico humano, um sistema imunológico digital consegue detectar código suspeito, mandá-lo automaticamente a um centro de análise e distribuir a cura para o sistema. Todo o processo tem lugar sem que o usuário tome conhecimento que tal proteção está ocorrendo.

Para lidar com ataques maliciosos, sistemas de intrusão devem detectar automaticamente os ataques e alertar os administradores do sistema. Especialistas em segurança computacional examinam então o problema, analisando e reparando o sistema. Como a escala das redes de computadores e sistemas continua se expandindo e o número de ataques aumenta, deve-se automatizar o processo.

2.1.6 Conhecimento do ambiente

Horn (2001) afirma que um sistema autonômico computacional conhece seu ambiente e o contexto de suas atividades, e age de acordo com elas. Um sistema autonômico encontrará e gerará regras para melhor interagir com os sistemas vizinhos. Ele irá negociar recursos que não estão sendo utilizados, mudando tanto a si quanto ao ambiente no processo. Essa habilidade permitirá a um sistema autonômica manter a confiabilidade sob diversas circunstâncias e combinações de circunstâncias. Mas mais importante, ela irá permitir os sistemas a prover informações úteis ao invés de dados confusos.

Sistemas autonômicos necessitarão ser capazes de descrever-se e aos seus recursos disponíveis para outros sistemas, e também precisarão ser capazes de descobrir automaticamente outros dispositivos no ambiente. Avanços serão necessários para fazer o sistema ciente das ações do usuário, junto com algoritmos que permitem ao sistema determinar a melhor resposta com um determinado contexto.

2.1.7 Heterogeneidade

Conforme Horn (2001), um sistema computacional autonômico não pode existir em um ambiente hermético. Enquanto independente em suas habilidades de se administrar, ele

deve funcionar em um mundo heterogêneo e implementar padrões abertos. Em outras palavras, um sistema autonômico computacional não pode, por definição, ser uma solução proprietária, pertencente a uma empresa ou grupo de empresas.

Na natureza, toda espécie de organismos deve coexistir e depender um do outro para sobreviver. Essa biodiversidade ajuda a equilibrar e estabilizar o ecossistema. No altamente evolutivo mundo da computação, uma coexistência e interdependência análoga são inevitáveis. À medida que a tecnologia avança, podem-se esperar novas invenções e novos dispositivos, e uma proliferação de opções e interdependências.

Colaborações na ciência da computação para criar padrões abertos permitiram novos tipos de compartilhamento. Esforços comunitários criaram um sistema operacional, um servidor *web* e protocolos para permitir que recursos computacionais possam ser compartilhados de modo distribuído. Esses esforços aceleraram a criação de padrões abertos, como ferramentas, bibliotecas, aplicações, etc. Avanços na computação autonômica precisarão de padrões abertos. Métodos simples de identificação, comunicação e negociação de sistemas precisam ser criados.

2.1.8 Antecipação

Segundo Horn (2001), um sistema autonômico computacional precisa antecipar os recursos otimizados necessários enquanto mantém sua complexidade escondida. Este é o objetivo final da computação autonômica: a diminuição entre os objetivos do consumidor e as implementações necessárias para alcançar tais objetivos.

Os consumidores precisam se adaptar a um sistema computacional, aprendendo como usa-lo, como interagir com ele, e como coletar, comparar e interpretar os vários tipos de informações que ele retorna antes de decidir o que fazer. Mesmo soluções personalizadas raramente melhoram esse panorama.

Um sistema autonômico possuirá antecipação e suporte. Ele entregará informações essenciais com um sistema otimizado e pronto para implementar as decisões feitas pelo usuário e não emaranhá-los desnecessariamente nos resultados do sistema.

2.2 ATAQUES REMOTOS DE TERCEIROS

Segundo Segurança (2000), define-se ataque como uma ação não autorizada realizada com a intenção de danificar, impedir, incapacitar ou quebrar a segurança de um servidor ou máquina. É qualificado como ataque remoto qualquer ataque feito a partir de uma máquina remota, ou seja, a partir de uma máquina que esteja conectada com a sua, como por exemplo, através da Internet.

Existem vários tipos de ataques remotos. Os mais simples são os ataques de recusa de serviço e bombardeio de correio, cujo maior propósito é causar aborrecimentos. Outros tipos de ataques mais comuns são o *spoofing* de *Internet Protocol* (IP), *sniffers*, cavalos de tróia e *backdoors*.

2.2.1 Recusa de Serviços

Conforme Silva (2001) ataques de recusa de serviços têm como principal objetivo interromper serviços de rede, servidores ou outros recursos a usuários legítimos. Os ataques de recusa de serviço também são conhecidos como *Denial of Service* (DoS) e podem causar grandes prejuízos, pois tem a capacidade de tirar do ar redes corporativas ou provedores de Internet.

Existem três maneiras básicas de realizar um ataque de recusa de serviços. A primeira delas é o consumo da largura de banda da rede que está sendo atacada. Para tal, o atacante deve possuir uma largura de banda maior do que a que está sendo atacada, ou ser capaz de utilizar várias estações. O atacante inunda a largura de banda com consultas alteradas, de modo a dificultar a identificação da origem dos ataques.

Uma segunda maneira de realizar um ataque DoS é consumir os recursos do servidor que está sendo atacado. Nesse ataque o sistema acaba travando ou ficando inutilizável, pois o atacante procura consumir o máximo possível de ciclos do processador, memória, processos do sistema operacional e sistema de arquivos.

A terceira maneira consiste em uma alteração das informações contidas nas tabelas de roteamento da rede. Uma rota legítima pode ser transformada em uma rota inexistente, impossibilitando o acesso a determinado recurso.

2.2.2 *Spoofing*

Segundo Silva (2001), *spoofing* de IP é uma técnica para autenticar uma máquina para outra forjando pacotes provenientes de um endereço confiável. Ou seja, é a troca do IP original por outro, para se passar por outro *host*.

Um ataque de *spoofing* de IP ocorre quando dois *hosts* possuem acesso confiável entre si, sendo que a autenticação deste acesso seja feita através do endereço de IP de origem dos datagramas.

2.2.3 Sniffers

Silva (2001) afirma que *sniffers* são quaisquer procedimentos que capturem informações ao longo da rede, seja por hardware ou software. Os *sniffers* são muito utilizados para espionagem de informações. Como podem ser instalados em qualquer parte de uma rede, eles se tornam um grande problema de segurança, podendo capturar informações importantes como nome e senhas de contas.

Os alvos preferidos dos *sniffers* são redes ou servidores que recebem muitas senhas para validação.

2.2.4 Cavalos de Tróia

Segundo Silva (2001), os cavalos de Tróia, comumente conhecidos como *Trojan Horse* ou simplesmente *Trojan*, são quaisquer programas que oferecem aplicações úteis, ou mesmo apenas divertidas, mas que realizam tarefas desconhecidas pelo usuário. Estas tarefas são sempre prejudiciais ao sistema que o está executando, tais como infecção de outros arquivos com o próprio cavalo de tróia, captura de senhas, destruição de dados, entre outros.

Um cavalo de tróia pode infectar qualquer programa executável do sistema, uma vez que estes possuem comumente funções ocultas. Mas podem também estar presentes em arquivos de imagem, áudio, vídeo, documentos de texto ou quaisquer arquivos do sistema. Uma vez aberto o arquivo, são executadas as funções do cavalo de tróia. A descoberta de um cavalo de tróia torna-se uma tarefa muito difícil.

2.2.5 *Backdoors*

Conforme Silva (2001), *backdoors*, ou porta dos fundos, são maneiras criadas por um usuário para invadir um sistema a qualquer momento. Uma porta dos fundos pode ser criada após uma invasão, para obter um acesso rápido ao mesmo sistema outra vez. Cavalos de tróia também podem ser responsáveis pela criação de portas de fundo.

Dependendo do tipo de porta dos fundos criada, o invasor pode manipular arquivos do sistema, obter senhas, monitorar informações do sistema, entre outros.

2.3 TRABALHOS CORRELATOS

Existem ferramentas que se propõem a fazer o mesmo que o módulo proposto. Algumas utilizam apenas o paradigma da computação autonômica, mas sem utilizar a auto-proteção.

O projeto Smart (IBM, 2005) utiliza-se da computação autonômica para implementar algumas funções nos aplicativos da IBM, de modo que os mesmos possuam a característica de auto-gerenciamento, melhorando a performance dos mesmos.

O projeto AutoAdmin (MICROSOFT, 2008) tem como objetivo tornar os sistemas de banco de dados com características autonômicas, tal como a auto-gerenciamento. O AutoAdmin rastreia o uso dos bancos de dados e adapta os mesmos aos requerimentos das aplicações.

A Universidade de Bologna, na Itália, está desenvolvendo o projeto AntHill (MONTRESOR et al, 2004) em conjunto com a Universidade de Ciência e Tecnologia Norueguesa. Um sistema AntHill consiste de uma rede dinâmica de nós do tipo *peer*. Uma sociedade de agentes adaptáveis percorre esta rede e, interagindo com os nós e entre si, os agentes resolvem alguns problemas complexos, como por exemplo, problemas de resistência, adaptação e auto-organização.

3 DESENVOLVIMENTO

Nesse capítulo são abordados os requisitos do módulo, bem como sua especificação e sua implementação. Ao final são apresentados os resultados constatados.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O principal requisito do módulo é a auto-proteção para um sistema *peer-to-peer*. Essa auto-proteção é desencadeada a partir dos seguintes requisitos:

a) o sistema deve monitorar portas abertas configuradas como não autorizadas pelo sistema e avisar ao usuário (Requisito Funcional – RF);
b) o sistema deve monitorar o acesso a arquivos do diretório configurado pelo sistema e informar ao usuário (RF);
c) o sistema deve impedir a execução de arquivos não confiáveis (RF);
d) o sistema deve ser implementado em Java (Requisito Não-Funcional – RNF);
e) o sistema deve ser implementado utilizando o ambiente de desenvolvimento Eclipse 3.2 (RNF).

3.2 ESPECIFICAÇÃO

Utilizando a notação da *Unified Modeling Language* (UML), foi feita a especificação do sistema. A especificação foi dividida em três diagramas, a saber, diagrama de classes, diagrama de casos de uso e diagrama de seqüência.

3.2.1 Diagrama de Classes

O diagrama de classes da aplicação pode ser visto na Figura 1.

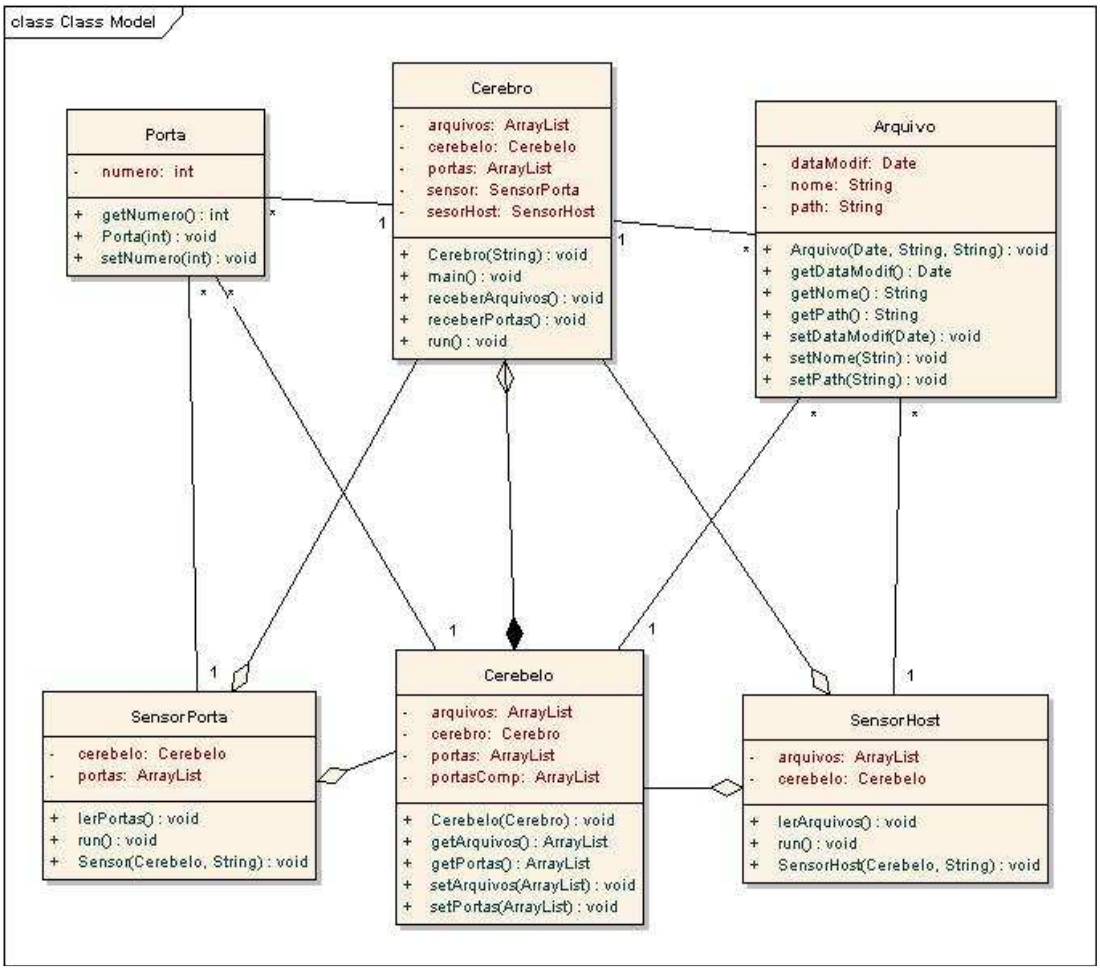


Figura 1 – Diagrama de classes

A classe `Porta` gera objetos que possuem as principais informações a respeito da porta que está sendo testada.

A classe `Arquivo` gera objetos que possuem as principais informações sobre o arquivo que está sendo testado.

A classe `SensorPorta` é responsável por varrer todas as portas e verificar quais portas estão abertas. As portas abertas são mantidas em um *array*, o qual é submetido à classe `Cerebelo`, para ser verificado se há portas abertas não-autorizadas.

A classe `SensorHost` é responsável por ler as informações dos arquivos de um diretório. Estas informações são mantidas em um *array*, que posteriormente é submetido à classe `Cerebelo`, onde será verificada a data de modificação de cada arquivo.

Na classe `Cerebelo` tem-se a verificação dos itens dos *arrays* recebidos das classes `SensorPorta` e `SensorHost`. O *array* de portas é varrido e, para cada porta, é verificado se a mesma deveria realmente estar aberta. Caso uma porta satisfaça esta condição, a mesma é inserida em outro *array* de portas. No *array* de arquivos é verificado, para cada arquivo, se o mesmo foi modificado nos últimos 10 minutos. Caso algum arquivo satisfaça esta condição, o

mesmo é inserido em outro *array* de arquivos. Os dois *arrays* criados nesta classe são submetidos à classe `Cerebro`.

A classe `Cerebro` é responsável por tomar as decisões e tomar as providências necessárias. Neste módulo, a decisão é enviar um aviso ao usuário. No caso das portas, o aviso conterá o número da porta aberta em questão. Já no caso dos arquivos, será o nome do arquivo a ser informado.

3.2.2 Diagrama de Casos de Uso

Na Figura 2 pode ser visualizado o diagrama de casos de uso.

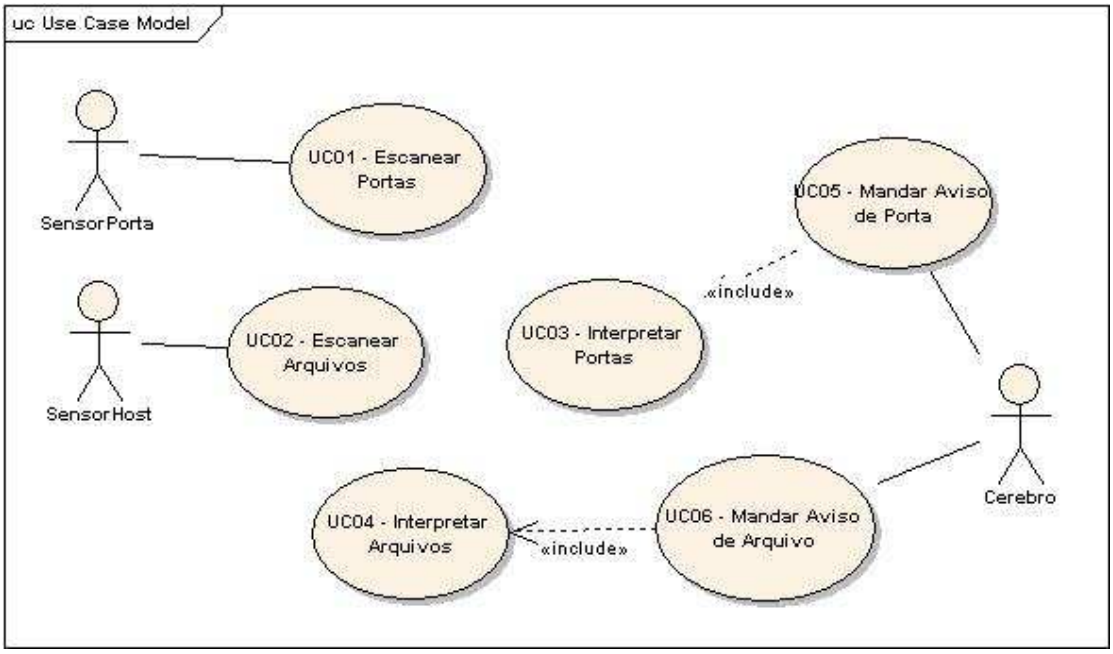


Figura 2 – Diagrama de casos de uso

Verifica-se na Figura 2 a utilização de seis casos de uso para o desenvolvimento do módulo. As classes Cérebro, SensorPorta e SensorHost são representados como um ator, pois possuem funções independentes. Para a classe `SensorPorta` foi verificada a necessidade de apenas um caso de uso, a saber, `UC01 – Escanear Portas`. Para a classe `SensorHost` foi criado também um caso de uso, `UC02 – Escanear Arquivos`.

À classe `Cerebro`, por ser mais complexa e responder por mais ações, foram definidos dois casos de uso: `UC05 – Mandar Aviso de Porta` e `UC06 – Mandar Aviso de Arquivo`. O caso de uso UC05 inclui ainda o caso de uso `UC03 – Interpretar Portas`, realizado pelo `Cerebelo`. Do mesmo modo, o caso de uso UC06 inclui o caso de uso `UC04 – Interpretar Arquivos`, também realizado pelo

Cerebelo.

No Quadro 1 tem-se descrito os cenários do Caso de Uso UC01 – Escanear Portas.

Principal: 1. O sensor de portas cria um socket em cada porta. 2. O sensor de portas manda o array de portas para o cerebelo. Porta aberta {Exceção}. No passo 1: 1.1.caso não for possível criar um socket em uma porta, o sensor acrescenta o objeto correspondente à porta em questão no array de portas.

Quadro 1 – Cenários do Caso de Uso UC01 – Escanear Portas

A descrição do cenário do Caso de Uso UC02 – Escanear Arquivos pode ser vista no Quadro 2.

Principal: 1. O sensor de host varre o diretório configurado. 2. Para cada arquivo do diretório, o sensor de host verifica a data de modificação. 3. O sensor de host manda o array de portas para o cerebelo.

Quadro 2 – Cenários do Caso de Uso UC02 – Escanear Arquivos

No Quadro 3 tem-se descrito os cenários do Caso de Uso UC03 – Interpretar Portas.

Principal: 1. O cerebelo recebe o array de portas do sensor de portas. 2. Para cada porta no array, o cerebelo verifica se ela deveria estar aberta. 3. O cerebelo interrompe o cerebro. 4. O cerebelo envia o um aviso ao cerebro. Porta aberta {Exceção}. No passo 2: 2.1. Caso a porta aberta deveria estar fechada, o cerebelo adiciona o objeto correspondente no array portas não-autorizadas.

Quadro 3 – Cenários do Caso de Uso UC03 – Interpretar Portas

A descrição dos os cenários do Caso de Uso UC04 – Interpretar Arquivos pode ser encontrada no Quadro 4.

Principal:
1. O cerebelo recebe o array de arquivos do sensor de host.
2. Para cada arquivo no array, o cerebelo verifica a data de modificação.
3. O cerebelo interrompe o cerebro.
4. O cerebelo envia um aviso ao cerebro.

Arquivo modificado {Exceção}.
No passo 2:
2.1. Caso o arquivo foi modificado nos últimos 10 minutos, o cerebelo adiciona o objeto correspondente no array de arquivos modificados.

Quadro 4 – Cenários do Caso de Uso UC04 – `Interpretar Arquivos`

No Quadro 5 tem-se descrito os cenários do Caso de Uso UC05 – `Mandar Aviso de Porta`.

Principal:
1. O cerebro recebe o aviso do cerebelo.
2. O cerebro pede ao cerebelo que mande o array de portas não-autorizadas.
3. O cerebro recebe o array de portas não-autorizadas.
4. Para cada porta no array de portas não-autorizadas, o cerebro manda um aviso ao usuário informando que a porta não está autorizada a estar aberta.

Quadro 5 – Cenários do Caso de Uso UC05 – `Mandar Aviso de Porta`

No Quadro 6 tem-se descrito os cenários do Caso de Uso UC06 – `Mandar Aviso de Arquivo`.

Principal:
1. O cerebro recebe o aviso do cerebelo.
2. O cerebro pede ao cerebelo que mande o array de arquivos modificados.
3. O cerebro recebe o array de arquivos modificados.
4. Para cada porta no array de arquivos modificados, o cerebro manda um aviso ao usuário informando que o arquivo foi modificado recentemente.

Quadro 6 – Cenários do Caso de Uso UC06 – `Mandar Aviso de Arquivo`

3.2.3 Diagramas de Seqüência

Na Figura 5 pode ser observado o diagrama de seqüência utilizado para os casos de uso UC01 – `Escanear Portas`, UC03 – `Interpretar Portas` e UC05 – `Mandar Aviso de Porta`.

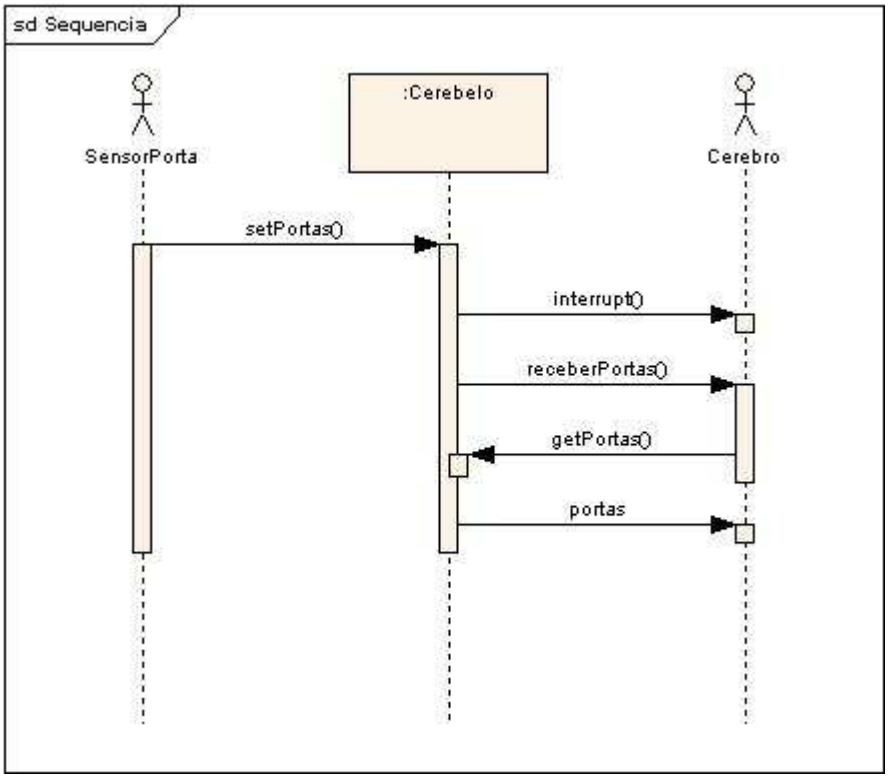


Figura 3 – Diagrama de Seqüência para os casos de uso UC01, UC03 e UC05

No diagrama da Figura 3, pode-se observar que, primeiramente, o `SensorPorta` envia uma mensagem ao `Cerebelo`, com as portas escaneadas. Após o processamento destas informações por parte do `Cerebelo`, o mesmo interrompe a execução do `Cerebro` e avisa-o que as informações a respeito das portas foram processadas e estão prontas para serem avaliadas.

Na Figura 4 pode ser observado o diagrama de seqüência utilizado para os casos de uso UC02 – `Escanear Arquivos`, UC04 – `Interpretar Arquivos` e UC06 – `Mandar Aviso de Arquivo`.

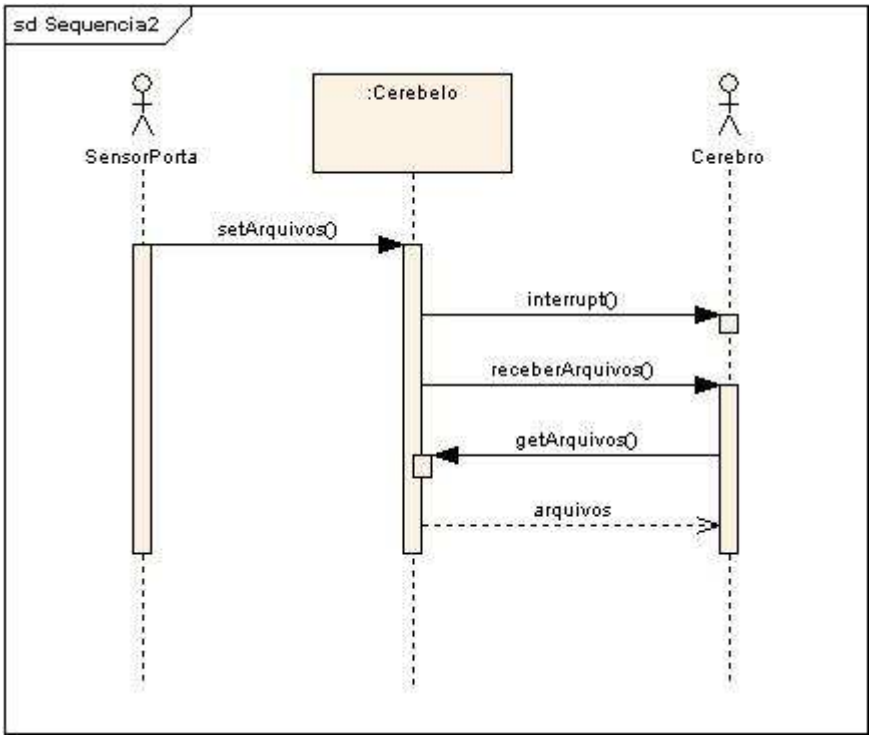


Figura 4 – Diagrama de Seqüência para os casos de uso UC02, UC04 e UC06

No diagrama da Figura 4, pode-se observar que, primeiramente, o `SensorHost` envia uma mensagem ao `Cerebelo`, com os arquivos escaneados. Após o processamento destas informações por parte do `Cerebelo`, o mesmo interrompe a execução do `Cerebro` e avisa-o que as informações a respeito dos arquivos foram processadas e estão prontas para serem avaliadas.

3.3 IMPLEMENTAÇÃO

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

3.3.1 Técnicas e ferramentas utilizadas

Para a implementação deste módulo foi utilizado a linguagem Java, bem como o ambiente de desenvolvimento Eclipse 3.2.

Apesar de modelado e especificado utilizando notações da orientação a objetos, este

módulo procura seguir o paradigma da computação autonômica, visando a auto-proteção.

Foram implementadas 7 classes neste módulo.

A classe `Configuração` serve para o administrador configurar o diretório a ser monitorado, bem como configurar quais as portas estão autorizadas a estarem abertas. O código-fonte desta classe pode ser observado no Quadro.

```
public class Configuracao {

	private static Properties props = new Properties();
	private static String PROPS_FILE_PATH = "C:/temp/props";

	public static void main (String args[]){
		File propsFile = new File(PROPS_FILE_PATH);
		ArrayList<Porta> portas = new ArrayList<Porta>();
		ArrayList<String> hashes = new ArrayList<String>();
		InputStreamReader isr = new InputStreamReader(System.in);
	    BufferedReader br = new BufferedReader(isr);
	    int nPorta;

		try {
			props.load(new FileInputStream(propsFile));
			for(int i=0; i<props.size(); i++)
			{
				if(props.getProperty("porta"+i)!=null){
					Porta po = new
Porta(Integer.parseInt(props.getProperty("porta"+i)));
					portas.add(po);
				}
			}
			String diretorio = props.getProperty("diretorio");
			File diretorio2 = new File(diretorio);
			File listaArquivos[] = diretorio2.listFiles();
			for(int i = 0; i < listaArquivos.length; i++){

	hashes.add(props.getProperty(listaArquivos[i].getName()));
			}

			String opcao = "6";
			while(!opcao.equals("0")){
				System.out.println("Escolha uma opção:");
				System.out.println("1 - Listar diretorio");
				System.out.println("2 - Mudar diretorio");
				System.out.println("3 - Listar portas");
				System.out.println("4 - Excluir porta");
				System.out.println("5 - Adicionar porta");
				System.out.println("0 - Sair");

				opcao = br.readLine();
				if (opcao.equals("1")){
					System.out.println("Diretorio: "+diretorio);
				}
				if (opcao.equals("2")){
					System.out.println("Informe o novo
diretorio:");
					diretorio = br.readLine();
				}
				if (opcao.equals("3")){
					System.out.println("Portas:");
```

```
                                    for (int i = 0; i < portas.size(); i++){
                                          Porta po = portas.get(i);
                                          System.out.println(po.getNumero());
                                    }
                              }
                              if (opcao.equals("4")){
                                    System.out.println("Informe a porta a ser
excluída:");
                                    nPorta = Integer.parseInt(br.readLine());
                                    for (int i = 0; i < portas.size(); i++){
                                          Porta po = portas.get(i);
                                          if (po.getNumero()==nPorta){
                                                portas.remove(i);
                                                break;
                                          }
                                    }
                              }
                              if (opcao.equals("5")){
                                    System.out.println("Informe a nova porta:");
                                    nPorta = Integer.parseInt(br.readLine());
                                    Porta po = new Porta(nPorta);
                                    portas.add(po);
                              }
                              if (opcao.equals("0")){
                                    props.clear();
                                    props.setProperty("diretorio",diretorio);
                                    for(int i=0; i<portas.size(); i++){
                                          Porta po = portas.get(i);
                                          if(po!=null){
                                                props.setProperty("porta"+i,
""+po.getNumero());
                                          }
                                    }

                                    for(int i = 0; i < listaArquivos.length;
i++){
                                          String hash = hashes.get(i);

props.setProperty(listaArquivos[i].getName(),hash);
                                    }
                                    propsFile.createNewFile();
                                    props.store(new FileOutputStream(propsFile),
"PropertiesDemo");
                                    break;
                              }

                        }

                  } catch (FileNotFoundException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                  } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                  }
            }
}
```

Quadro 7 – Código da classe `Configuracao`

Na classe `Configuracao`, ocorre primeiramente a leitura de todos os dados que

estão no arquivo de configurações. O usuário pode então alterar a lista de portas autorizadas, bem como definir um novo diretório onde se encontram os arquivos a serem monitorados.

As classes `Porta` e `Arquivo` tem por objetivo guardar as informações referentes à uma porta e a um arquivo, respectivamente. Para tanto foram utilizados os códigos-fonte que podem ser observados nos Quadros 8 e 9.

```
public class Porta {

	private int numero;

	public Porta(int numero) {
		this.numero = numero;
	}

	public int getNumero() {
		return numero;
	}

	public void setNumero(int numero) {
		this.numero = numero;
	}
}
```

Quadro 8 – Código da classe `Porta`

A classe `Porta` possui apenas uma variável, que é o seu número. Esta informação é suficiente para referenciá-la sempre que necessário, visto que não há portas com números iguais.

```
public class Arquivo {

	private String nome;
	private String path;
	private Date dataModif;
	private byte[] hash;

	public Arquivo(String nome, String path, Date dataModif, byte[]
hash){
		this.nome = nome;
		this.path = path;
		this.dataModif = dataModif;
		this.hash = hash;
	}

	public Date getDataModif() {
		return dataModif;
	}

	public void setDataModif(Date dataModif) {
		this.dataModif = dataModif;
	}

	public String getNome() {
		return nome;
	}

	public void setNome(String nome){
		this.nome = nome;
	}

	public String getPath() {
		return path;
	}

	public void setPath(String path){
		this.path = path;
	}

	public byte[] getHash(){
		return this.hash;
	}
}
```

Quadro 9 – Código da classe `Arquivo`

A classe `Arquivo` tem por função o armazenamento dos dados de um arquivo, sendo estes dados o nome do arquivo, seu caminho de localização e sua data de última modificação. Como não é possível ter-se dois arquivos de mesmo nome dentro de uma mesma pasta, estes dados são suficientes para referenciá-los e para fazer os testes necessários.

No Quadro 10 tem-se o código-fonte da classe `SensorPorta`. A classe `SensorPorta` realiza o escaneamento das portas, observando se as mesmas estão abertas ou fechadas. A listagem das portas abertas é mandada para um objeto de `Cerebelo`.

```
public class SensorPorta extends Thread{

	private ArrayList<Porta> portas;
	private Cerebelo cerebelo;
	private Properties props = new Properties();

	public SensorPorta(String str, Cerebelo cerebelo){
		super(str);
		this.cerebelo = cerebelo;
	}

	public void run(){
		while(true){
			//le as portas abertas
			lerPortas();
			//passa os dados ao cerebelo
			cerebelo.setPortas(this.portas);
			try {
				sleep(100);
			} catch (InterruptedException e) {
			}
		}
	}

	private void lerPortas(){
		portas = new ArrayList<Porta>();
		props.clear();
		//para cada porta existente, tenta-se criar um socket
		//onde o socket não pode ser criado, a porta está aberta
		//e é inserida no array
		for(int i = 0; i <= 6553; i++)
		{
			try{
			   ServerSocket port = new ServerSocket(i);
			   port.close();
			 }catch(java.io.IOException e){
				Porta p = new Porta(i);
				portas.add(p);
			 }
		}
	}

}
```

Quadro 10 – Código da classe `SensorPorta`

A classe `SensorHost`, por sua vez, realiza o escaneamento dos arquivos do diretório especificado em um arquivo de configurações. Essas informações enviadas para um objeto de `Cerebelo`. O código-fonte da classe `SensorHost` pode ser visto no Quadro 11.

```
public class SensorHost extends Thread{

	private ArrayList<Arquivo> arquivos;
	private Cerebelo cerebelo;
	private Properties props = new Properties();
	private static String PROPS_FILE_PATH = "c:/temp/props";

	public SensorHost(String str, Cerebelo cerebelo){
```

```
		super(str);
		this.cerebelo = cerebelo;
	}

	public void run(){
		while(true){
			//le os arquivos
			lerArquivos();
			//passa os dados ao cerebelo
			cerebelo.setArquivos(this.arquivos);
			try {
				sleep(25000);
			} catch (InterruptedException e) {
			}
		}
	}

	private void lerArquivos(){
		arquivos = new ArrayList<Arquivo>();
		//cria um array com os arquivos do diretorio
		File propsFile = new File(PROPS_FILE_PATH);
		try {
			props.load(new FileInputStream(propsFile));
		} catch (FileNotFoundException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		String dir = props.getProperty("diretorio");
		File diretorio = new File(dir);
		File listaArquivos[] = diretorio.listFiles();

		//define a variável do hash
	  MessageDigest md = null;
	  try {
		//cria uma instância do gerador de hash
		md = MessageDigest.getInstance("SHA-1");
	  }
	  catch(NoSuchAlgorithmException e) {
		e.printStackTrace();
	  }

		for(int i = 0; i < listaArquivos.length; i++){

			FileInputStream fis = null;
			String caminho = listaArquivos[i].getPath();
			File arquivo = new File(caminho);
			try {
				//System.out.println(arquivo);
				fis = new FileInputStream(arquivo);
			} catch (FileNotFoundException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
			byte fileContent[] = new byte[(int)arquivo.length()];
			try {
				fis.read(fileContent);
			} catch (IOException e1) {
				// TODO Auto-generated catch block
```

```
                        e1.printStackTrace();
                    }

                //coloca a mensagem no gerador de hash
                md.update(fileContent);

                //gera o hash
                byte [] hash = md.digest();
                String hash2 = hash.toString();
                    //para cada arquivo no array
                    //é criado um objeto que será passado ao cerebelo
                    Arquivo arq = new Arquivo(listaArquivos[i].getName(),
listaArquivos[i].getPath(), new Date(listaArquivos[i].lastModified()),
hash);
                    //(2) Lista de arquivos
                    //System.out.println("Arquivo: "+arq.getNome());
                    arquivos.add(arq);
            }
        }
}
```

Quadro 11 – Código da classe `SensorHost`

No Quadro 12 pode-se ver o código-fonte da classe `Cerebelo`. Ela é responsável por receber os dados enviados pelas classes `Sensor` e `SensorHost`. Com os dados recebidos da classe `SensorPorta`, que são todos objetos do tipo `Porta`, é feita uma comparação com uma coleção de `Portas` lidas de um arquivo de configurações, de modo a averiguar se a `Porta` em questão deveria estar aberta. As `Portas` que estão abertas e que não o deveriam são enviadas a um objeto do tipo `Cerebro`. Recebendo os dados da classe `SensorHost`, que são objetos de `Arquivo`, é feita uma comparação do *hash* deste arquivo com o *hash* correspondente armazenado no arquivo de configurações. Os arquivos que sofreram alterações são enviados a um objeto de `Cerebro`.

```
public Cerebelo(Cerebro cerebro){
            this.cerebro = cerebro;
            File propsFile = new File(PROPS_FILE_PATH);
            try {
                  props.load(new FileInputStream(propsFile));
            } catch (FileNotFoundException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            } catch (IOException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
            for(int i=0; i<props.size(); i++)
            {
                  if(props.getProperty("porta"+i)!=null){
                        Porta po = new
Porta(Integer.parseInt(props.getProperty("porta"+i)));
                        //(3) Lista de portas autorizadas
                        //System.out.println("Porta autorizada:
"+po.getNumero());
                        portasComp.add(po);
                  }
```

```
		}
	}

	public ArrayList<Porta> getPortas() {
		return portas;
	}

	public void setPortas(ArrayList<Porta> port) {
		//para cada porta recebida
		//verifica se ela se encontra tambem no
		//array de comparação
		//caso não se encontre, significa que não deveria estar aberta
		boolean tem = false;
		while(port.size()>0){
			tem = false;
			Porta po = port.get(0);
			for(int i=0; i<portasComp.size(); i++){
				Porta q = portasComp.get(i);
				if (po.getNumero()==q.getNumero()){
					tem = true;
				}
			}
			if (!tem){
				portas.add(po);
				//(4) Lista de portas abertas não-autorizadas
				System.out.println("Porta aberta:
"+po.getNumero());
			}
			port.remove(0);
		}
		//interrompe o cerebro
		cerebro.interrupt();
		//manda o cerebro receber os dados
		cerebro.receberPortas();
	}

	public ArrayList<Arquivo> getArquivos(){
		return arquivos;
	}

	public void setArquivos(ArrayList<Arquivo> arq){
		while(arq.size()>0){
			Arquivo ar = arq.get(0);
			if(props.getProperty(ar.getNome())!=null){
				String hash = new
String(props.getProperty(ar.getNome()));
				String hash2 = ar.getHash().toString();
				if (!hash.equals(hash2)){
					arquivos.add(ar);
					//(5) Lista de arquivos modificados
					System.out.println("Arquivo modificado:
"+ar.getNome());
				}
				arq.remove(0);
			}
			else {
				arquivos.add(ar);
			}
		}
		//interrompe o cerebro
		cerebro.interrupt();
```

```
            //manda o cerebro receber os dados
            cerebro.receberArquivos();
      }
}
```

Quadro 12 – Código da classe `Cerebelo`

O `Cerebro` é a classe principal do módulo. É nele que são iniciadas as *threads* que fazem com que os sensores começam a trabalhar. As informações recebidas de um `Cerebelo` são enviadas ao usuário através de alertas que informam sobre a situação, ale´m de serem gravadas em um *log*. O código-fonte da classe `Cerebro` pode ser vista no Quadro 13.

```
      public Cerebro(String str){
            super(str);
            //cria as threads dos sensores
            sensorPorta.start();
            sensorHost.start();
      }

      public static void main(String [] args){
            //cria a thread principal, ou seja, a si próprio
            new Cerebro("cerebro").start();
      }

      public void run() {
            while(true){
                  //se houver recebido as portas
                  if(portas.size()>0){
                        //para cada porta recebida
                        //imprime a porta aberta
                        while(portas.size()>0){
                              Porta po = portas.get(0);
                              java.util.Date agora = new java.util.Date();
                              gravarLog(""+agora+" Porta aberta:
"+po.getNumero());
                              boolean inter =
this.currentThread().interrupted();
                              //OptionPane.showMessageDialog(null,"Atenção!
\nA porta "+po.getNumero()+" não deveria estar aberta!");
                              portas.remove(0);
                        }
                  }
                  //se recebeu o array de arquivos
                  if(arquivos.size()>0){
                        //para cada arquivo
                        //imprime o arquivo modificado
                        while(arquivos.size()>0){
                              Arquivo ar = arquivos.get(0);
                              java.util.Date agora = new java.util.Date();
                              gravarLog(""+agora+" Arquivo modificado:
"+ar.getNome());
                              boolean inter =
this.currentThread().interrupted();

      //JOptionPane.showMessageDialog(null,"Atenção! \nO arquivo
"+ar.getNome()+" foi modificado!");
                              arquivos.remove(0);
                        }
```

```
                    }
                    try {
                          sleep(500);
                    } catch (InterruptedException e){
                    }
              }
        }

        public void receberPortas(){
              portas = cerebelo.getPortas();
        }

        public void receberArquivos(){
              arquivos = cerebelo.getArquivos();
        }

        public static void gravarLog(String texto) {
              try {
                 // Gravando no arquivo
                 File arquivo;
                 arquivo = new File("arquivo.txt");
                 FileOutputStream fos = new FileOutputStream(arquivo);
                 fos.write(texto.getBytes());
                 fos.close();
              }
              catch (Exception ee) {
                 ee.printStackTrace();
              }
        }
}
```

Quadro 13 – Código da classe `Cerebro`

3.3.2 Operacionalidade da implementação

Nesta seção são apresentadas as operacionalidades da implementação, demonstrando as etapas para verificar as portas abertas não-autorizadas e os arquivos modificados recentemente.

O primeiro passo é executar a aplicação. Foi utilizado o programa *Peer-to-Peer Instant Messenger* (P2PIM) (DICKINSON, HOLLENBECK, SINGH, 2002) para a demonstração das funcionalidades do módulo. Na Figura 5 pode ser observado uma visão geral quando da execução do aplicativo.

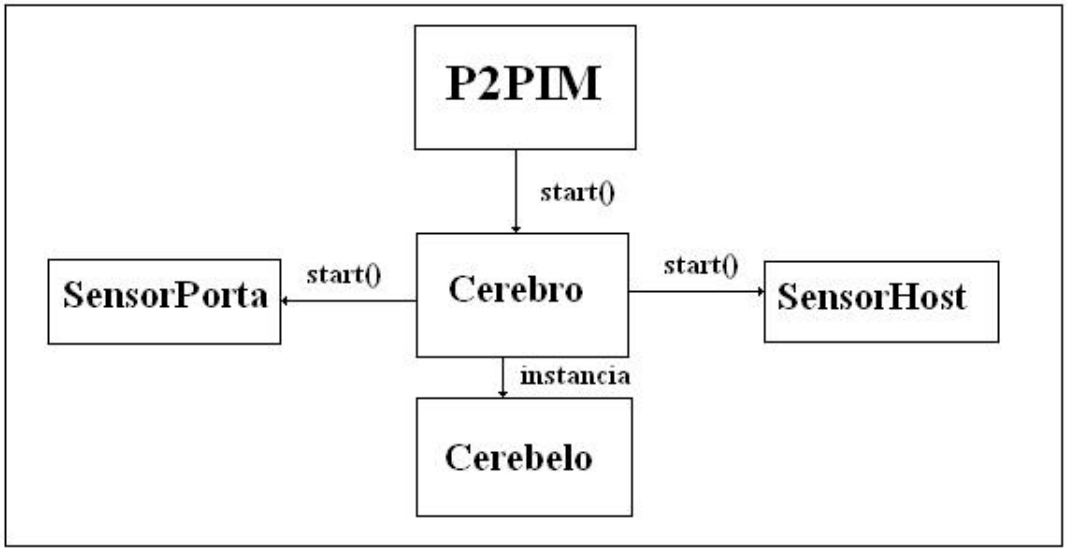


Figura 5 – Visão geral da inicialização do sistema

Observa-se na Figura 5 que o P2PIM, que é o aplicativo em si, executa a *thread* principal do módulo, o `Cerebro`. O `Cerebro`, por sua vez, instancia um objeto de `Cerebelo` e inicia a execução das *threads* `SensorPorta` e `SensorHost`.

Na Figura 6 pode ser observada a troca de mensagens que ocorrem durante a execução do módulo.

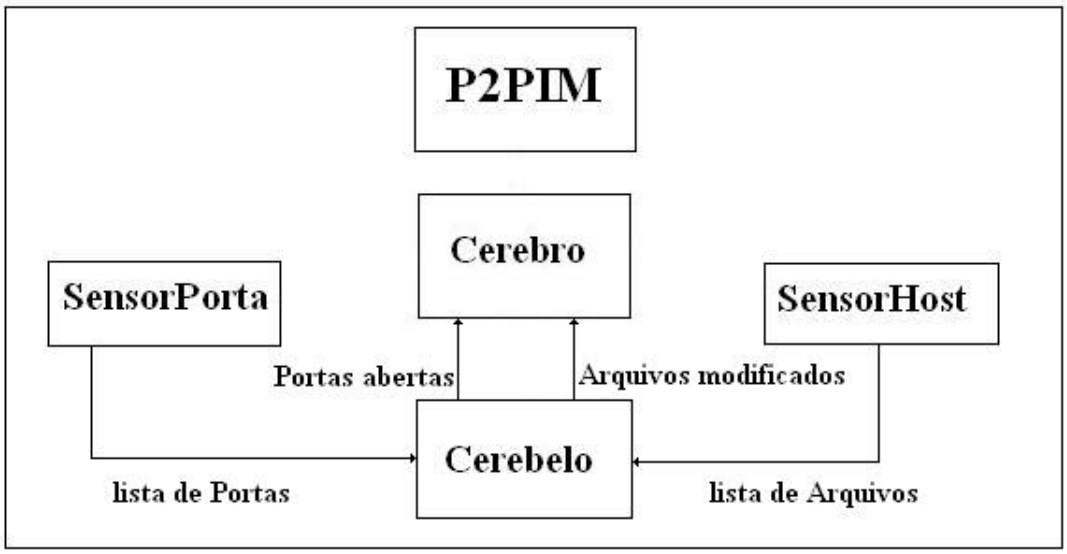


Figura 6 – Visão geral das trocas de mensagens no sistema

Observa-se na Figura 6 a troca de mensagens entre os componentes do sistema. O `SensorPorta` manda ao `Cerebelo` a lista de portas abertas. O `SensorHost` manda ao `Cerebelo` a lista de arquivos do diretório. O `Cerebelo`, por sua vez, envia a lista de portas abertas e a lista de arquivos modificados ao `Cerebro`. Nota-se que, em nenhum momento, ocorre a interação entre algum componente do módulo e a aplicação, de modo que o módulo

não interfere na execução da mesma.

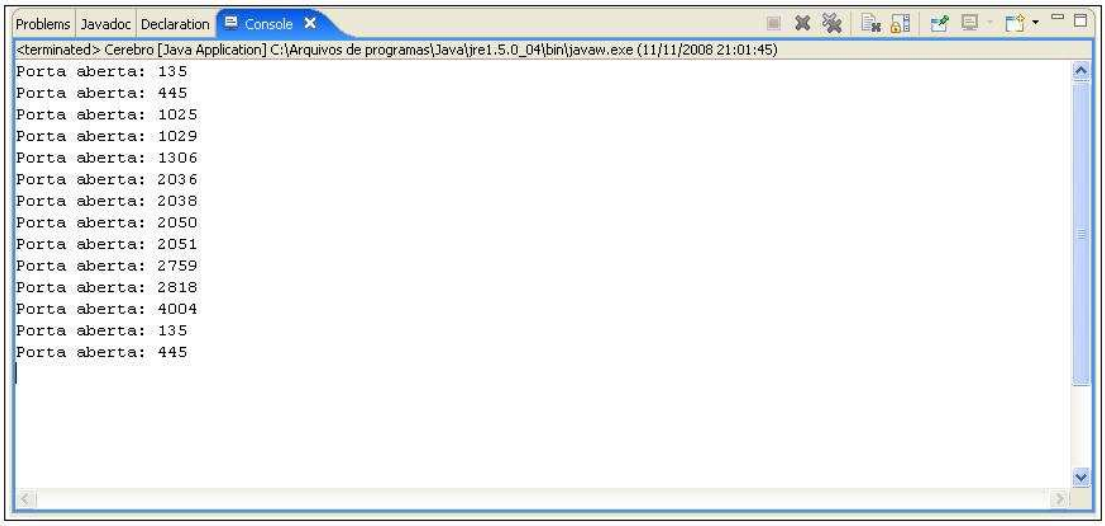
No Quadro 14 pode ser observada a tela inicial da aplicação. No momento em que a aplicação é iniciada, o módulo também o é, executando independentemente da aplicação até a mesma ser finalizada.

Quadro 14 – Tela inicial

Ao iniciar a aplicação, a mesma inicia a *thread* principal, o `Cerebro`, e continua sua execução. O `Cerebro`, por sua vez, instancia um objeto de `Cerebelo` e inicia as *threads* do `SensorPorta` e do `SensorHost`.

A *thread* do `SensorPorta`, iniciada pelo `Cerebro`, varre as portas do sistema, tentando criar *sockets* de comunicação em cada uma delas. Se não for possível criar o *socket*, é sinal de que ela está aberta.

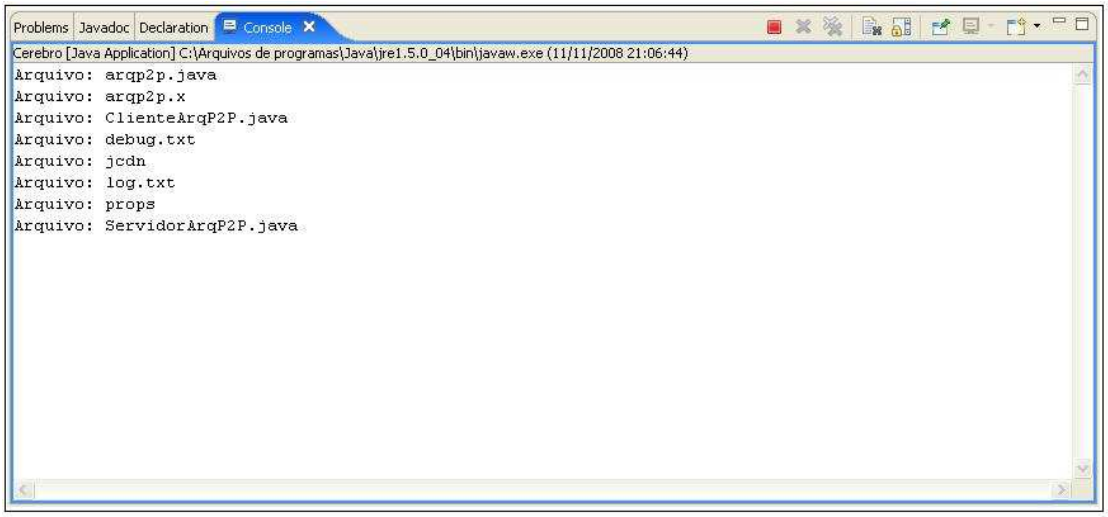
No Quadro 15 pode-se observar um exemplo desta lista de portas abertas encontradas pelo sensor de portas.



Quadro 15 – Lista de Portas Abertas

A lista de portas apresentada no Quadro 15 representa as portas abertas encontradas pela *thread* `SensorPorta`. Esta lista é repassada à instância do `Cerebelo` para que o mesmo efetue as comparações necessárias. Após passar a lista ao `Cerebelo`, o `SensorPorta` continua escaneando as portas, sem se importar com as ações tomadas pelo `Cerebelo`.

A *thread* do `SensorHost`, iniciada pelo `Cerebro`, verifica todos os arquivos de um diretório. No Quadro 16 pode-se observar um exemplo dos arquivos encontrados pelo sensor no diretório configurado.



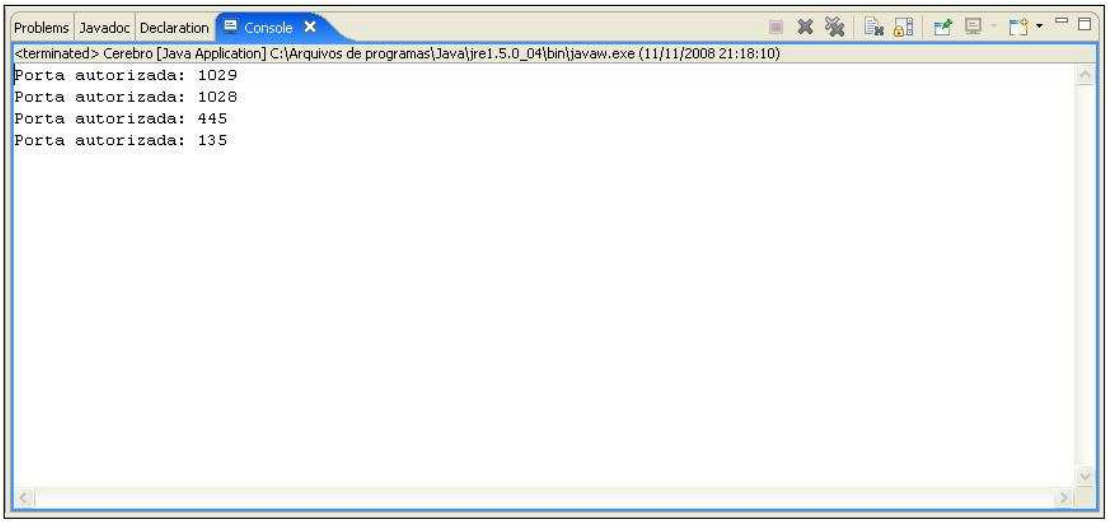
Quadro 16 – Lista de Arquivos

Observa-se no Quadro 16 a lista de arquivos encontrados pelo `SensorHost` no diretório estipulado. Nota-se que não apenas o nome do arquivo é fornecido, mas também sua extensão.

Esta lista de arquivos é repassada à instância do `Cerebelo`, para que o mesmo faça as

verificações necessárias. Nota-se que mesmo após passar a lista de arquivos ao `Cerebelo`, o `SensorHost` continua procurando os arquivos do diretório, sem se importar com as ações tomadas pelo `Cerebelo`.

O `Cerebelo` lê a lista de portas autorizadas presentes em um arquivo de propriedades. No Quadro 17 observa-se um exemplo desta lista de portas. Qualquer porta não relacionada nesta lista não deveria estar aberta e pode vir a ser a causa ou o meio de uma invasão.

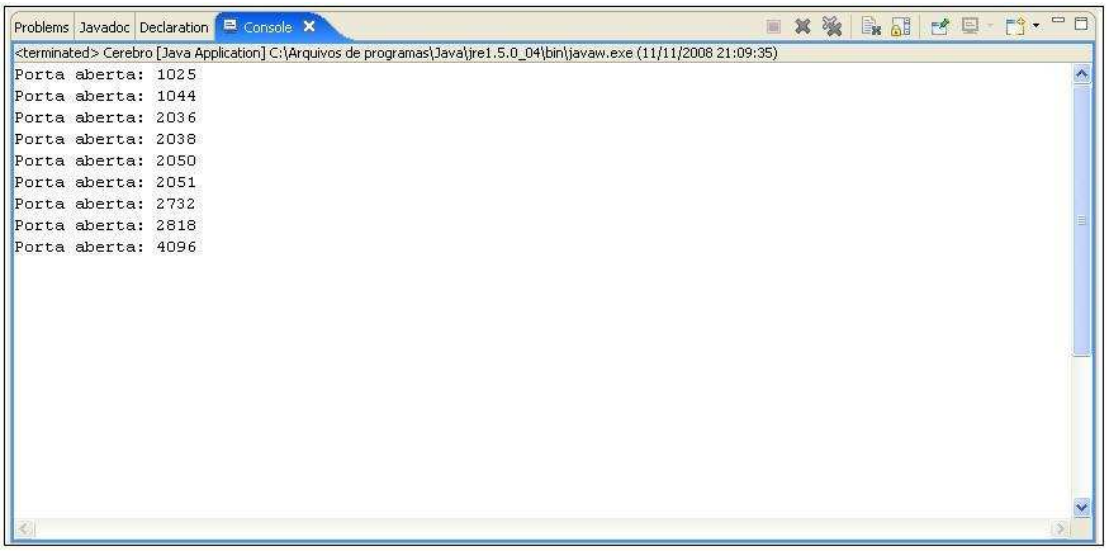


Quadro 17 – Lista de portas autorizadas

Na Figura 16 observa-se a lista de portas autorizadas encontradas em um arquivo de configuração. Esta lista apresenta apenas as portas autorizadas a estarem abertas e qualquer porta aberta não constante desta lista pode significar um ataque remoto.

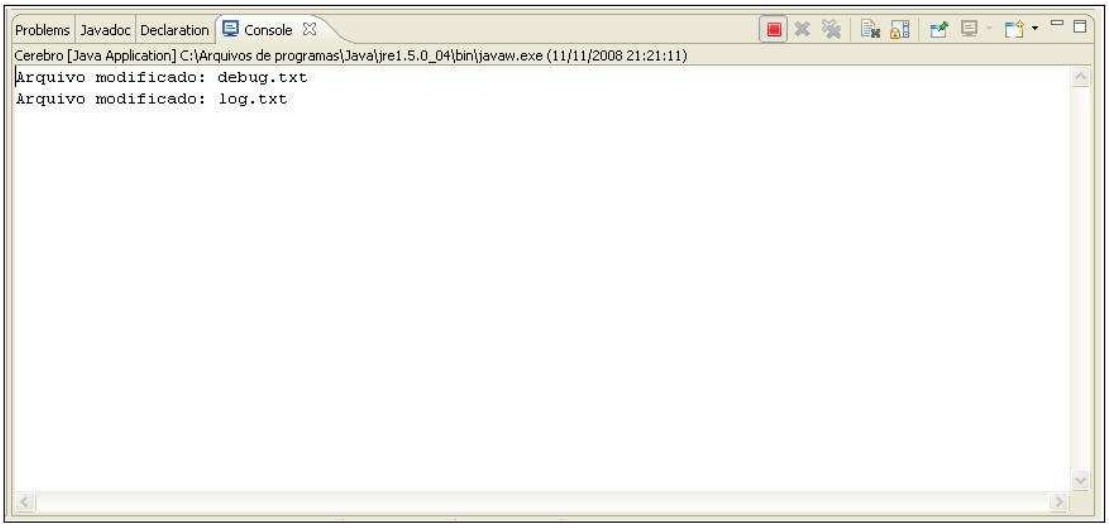
O `Cerebelo`, a partir da lista de portas autorizadas e da lista de portas abertas recebidas do `SensorPorta`, faz uma comparação entre as duas. As portas abertas que não estão na lista de portas autorizadas são repassadas ao `Cerebro`.

Um exemplo de portas abertas não autorizadas pode ser vista no Quadro 18. Observa-se que estas portas relacionadas estão abertas, como observado no Quadro 15 e não estão autorizadas a estarem abertas, como observado no Quadro 17.



Quadro 18 – Lista de portas abertas não autorizadas

A partir da lista de arquivos repassada pelo `SensorHost`, que pode ser observada no Quadro 16, o `Cerebelo` verifica, um a um, se sua última modificação foi realizada em um intervalo de tempo configurado. A lista dos arquivos modificados neste intervalo de tempo é repassada ao `Cerebro`. Um exemplo desta lista de arquivos modificados pode ser observado no Quadro 19.



Quadro 19 – Lista de arquivos modificados

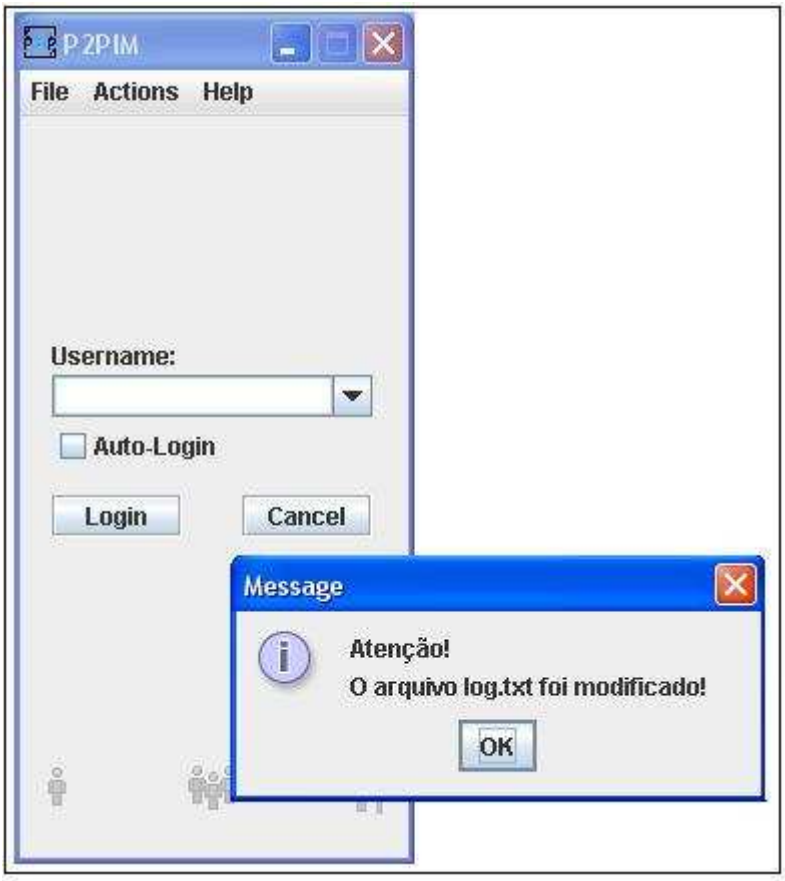
O `Cerebro` recebe as duas listas do `Cerebelo` e apresenta ao usuário uma mensagem de erro avisando se uma porta não autorizada está aberta, ou se um arquivo do diretório configurado foi modificado no intervalo de tempo definido.

Um exemplo da mensagem de erro quando uma porta não-autorizada está aberta pode ser observado no Quadro 20.

Quadro 20 – Mensagem de erro de porta aberta não autorizada

Observa-se pelo Quadro 20 que, mesmo com a mensagem de que uma porta não autorizada está aberta, o sistema continua sua execução normalmente.

No Quadro 21 pode-se observar um exemplo de uma mensagem de erro quando um arquivo for modificado recentemente.



Quadro 21 – Mensagem de erro de arquivo modificado

Observa-se pelo Quadro 21 que, mesmo com a mensagem de que um arquivo foi modificado recentemente, o sistema continua em execução normalmente.

Verificou-se que o módulo encontra as portas abertas e avisa ao usuário se alguma não está autorizada a estar aberta. Também avisa ao usuário se algum arquivo do diretório especificado foi alterado recentemente. Notou-se que a execução do módulo não interfere na execução da aplicação.

3.4 RESULTADOS E DISCUSSÃO

Neste trabalho foram citados três objetivos para melhorar a segurança em um sistema P2P. Para alcançar estes objetivos foi escolhida a computação autonômica, em especial a auto-proteção, para implementar estes objetivos.

O usuário é avisado por meio de uma mensagem quando uma porta não autorizada está aberta e quando um arquivo do diretório especificado foi modificado recentemente.

Desse modo, caso haja alguma invasão, o usuário pode precaver-se contra possíveis

malefícios.

Uma porta aberta pode significar uma *backdoor*, por onde algum invasor pode entrar no sistema e roubar informações, ou até causar danos irreparáveis ao funcionamento do sistema.

Quando algum arquivo é modificado, pode-se depreender do fato que alguém o fez. Se não foi o usuário nem o sistema, foi algum invasor. É importante saber se algum arquivo foi modificado, pois o mesmo pode ter sido alterado de modo a causar danos ao sistema, ou de modo a passar informações ao invasor.

O módulo é independente da aplicação, executando de modo a não interferir na mesma. A aplicação necessita somente iniciar a execução do módulo. Assim, o módulo não interfere na execução da aplicação e vice-versa. Essa vantagem permite ao módulo uma maior liberdade de ações, pois não precisa da autorização da aplicação para executar e não precisa dar um retorno de suas ações para ela.

4 CONCLUSÕES

Este trabalho apresentou alguns conceitos de computação autonômica, de sistemas *peer-to-peer* e de ataques remotos.

As técnicas/ferramentas utilizadas foram adequadas ao desenvolvimento do trabalho, desde o paradigma da computação autonômica à linguagem de programação Java.

O protótipo verifica corretamente se alguma porta configurada como não-autorizada está aberta e se algum arquivo foi modificado recentemente, como foi objetivado. Porém, ele não impede a execução de arquivos não-confiáveis, devido às dificuldades encontradas em determinar a confiabilidade de um arquivo.

Outros sistemas utilizam a computação autonômica para a resolução de alguns problemas. Não foi possível encontrar nenhum que utilize a auto-proteção, porém utilizam outros aspectos da computação autonômica, tais como o auto-gerenciamento.

O projeto AntHill (MONTRESOR et al, 2004) assemelha-se a este trabalho no que se refere a utilizar um sistema P2P para o desenvolvimento de aspectos da computação autonômica. Porém, o AntHill utiliza a tecnologia de agentes, diferentemente deste trabalho. O AntHill procura resolver aspectos de auto-organização e de adaptação, sem possuir, no entanto, aspectos da auto-proteção.

De maneira geral, conclui-se que o trabalho contribui para o desenvolvimento de aplicações de segurança, visto que usa técnicas de segurança. O uso da computação autonômica torna-o diferente das demais aplicações e contribui para o desenvolvimento da mesma.

4.1 EXTENSÕES

A complexidade e a abrangência da segurança de sistemas possibilitam a extensão deste trabalho de várias maneiras.

Podem-se verificar outros aspectos do sistema, tais como o uso da memória por parte dos processos em execução. Outra opção é transformar o módulo de modo a utilizar agentes e não apenas estático em uma máquina.

Pode-se ainda verificar a procedência de arquivos, para assim determinar se o mesmo é

de fonte segura antes de executá-lo.

4.2 DIFICULDADES

Foram encontradas várias dificuldades na realização deste trabalho.

A utilização da linguagem Java traz vários benefícios e facilidades, mas traz também algumas dificuldades, tais como métodos referentes à *threads* e exceções ocorridas não esperadas.

A principal dificuldade encontrada foi o pouco conteúdo sobre computação autonômica disponível, com poucas informações aprofundadas sobre o assunto e poucos exemplos disponíveis, sendo em sua maioria apenas teoria.

REFERÊNCIAS BIBLIOGRÁFICAS

DICKINSON, R.; HOLLENBECK, J; SINGH, S. **P2P Instant Messenger**. [S.l.], 2002. Disponível em: <http://sourceforge.net/projects/p2pim/>. Acesso em: 16 nov 2008.

ENGLE, M.; KHAN, J. I. **Vulnerabilities of P2P systems and a critical look at their solutions**. [S.l.], 2006. Disponível em: <http://www.medianet.kent.edu/techreports/TR2006-11-01-p2pvuln-EK.pdf>. Acesso em: 31 mar. 2008.

GASPARY, L. P. **Incorporação de segurança em aplicações peer-to-peer**: habilitando novas oportunidades além de compartilhamento de arquivos. [Porto Alegre], 2006. Disponível em: <si3.inf.ufrgs.br/HomePage/seminariosII/arquivos/seminarioII-2006-LUCIANO_PASCHOAL_GASPARY.pdf>. Acesso em: 6 abr. 2008.

HORN, P. **Autonomic computing**: IBM´s perspective on the state of information technology. EUA, 2001. Disponível em: <http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf>. Acesso em: 30 set 2008.

IBM. **Computer science**: autonomic databases. [S.l.], [2005?]. Disponível em: <http://www.almaden.ibm.com/cs/projects/autonomic/>. Acesso em: 6 abr. 2008.

MICROSOFT. **AutoAdmin**: self-tuning and self-administering databases. [S.l.], 2008. Disponível em: <http://research.microsoft.com/dmx/autoadmin/>. Acesso em: 6 abr. 2008.

MONTRESOR, A. et al. **The Anthill project**. [S.l], 2004. Disponível em: <http://www.cs.unibo.it/projects/anthill/>. Acesso em: 6 abr. 2008.

PARASHAR, M.; HARIRI, S. **Autonomic computing**: an overview. [S.l.], 2005. Disponível em: <www.caip.rutgers.edu/TASSL/Papers/automate-upp-overview-05.pdf >. Acesso em: 6 abr. 2008.

SALEHIE, M.; TAHVILDARI, L. **Autonomic computing**: emerging trends and open problems. Waterloo, 2005. Disponível em: <http://www.simondobson.org/files/teaching/content/distributed-systems/papers/salahie.pdf>. Acesso em: 6 abr. 2008.

SEGURANÇA máxima: o guia de um hacker para proteger seu site na Internet e sua rede. 3. ed. Tradução Edson Furmankiewicz, Joana Figueiredo. Rio de Janeiro: Campus, 2000.

SILVA, P. F. **Protótipo de software de segurança em redes para a monitoração de pacotes em uma conexão TCP/IP**. 2001. 112 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

ZAIANTCHICK, J. B. **Análise sobre o impacto da tecnologia peer-to-peer**. São Paulo, 2001. Disponível em: <http://www.ime.usp.br/~is/ddt/mac339/projetos/2001/demais/bello/>. Acesso em: 2 out. 2009.