UNIVERSIDADE DE CAXIAS DO SUL
CENTRO DE COMPUTAÇÃO E TECNOLOGIA DA INFORMAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUÍS FERNANDO ARCARO

Desenvolvimento de uma plataforma de comunicação para dispositivos com interface de rede *TCP/IP*

André Luis Martinotto
Orientador

Alexandre Erasmo Krohn Nascimento
Coorientador

Caxias do Sul, Outubro de 2015

Desenvolvimento de uma plataforma de comunicação para dispositivos com interface de rede *TCP/IP*

por

Luís Fernando Arcaro

Projeto de Diplomação submetido ao curso de Bacharelado em Ciência da Computação do Centro de Computação e Tecnologia da Informação da Universidade de Caxias do Sul, como requisito obrigatório para graduação.

Projeto de Diplomação

Orientador: André Luis Martinotto
Coorientador: Alexandre Erasmo Krohn Nascimento
Banca examinadora:

Ricardo Vargas Dorneles
CCTI/UCS
Alexandre Erasmo Krohn Nascimento
CCTI/UCS

Projeto de Diplomação apresentado em
9 de Novembro de 2012

Daniel Luís Notari
Coordenador

"Assiduus usus uni rei deditus et ingenium et artem saepe vincit"
"A prática constante de um assunto muitas vezes supera a inteligência e a habilidade"
Marcus Tullius Cicero

AGRADECIMENTOS

Agradeço a todas as pessoas que, direta ou indiretamente, ajudaram a tornar o curso de graduação e este trabalho possíveis e bem-sucedidos.

Inicialmente agradeço à minha amada família: aos meus pais, Izabel e Lourenço, pela minha existência e pelo ensinamento e constante reiteração sobre a importância da educação e do conhecimento, além do oferecimento de ambos em abundância. Agradeço a eles também pelo apoio que sempre forneceram e pela existência de minha irmã, Katia, e a ela pelo auxílio que me prestou desde a infância, das vogais às integrais. Agradeço também à minha namorada, Vanessa, pelo amor, compreensão e apoio durante toda a caminhada (nada teria sido tão bom sem você), e ao meu cunhado Guilherme, pelo companheirismo e pela compreensão ao abrir mão do tempo com a Katia para que pudéssemos estudar matemática.

Aos professores, agradeço pelo conhecimento e pelas experiências compartilhadas durante o curso. Se há um caminho para um melhor futuro, esse caminho passa por vocês. Em especial agradeço aos professores André e Alexandre, pela orientação, pela atenção e pela paciência durante o desenvolvimento deste trabalho, e ao professor Ricardo, pelo apoio e acompanhamento nos últimos anos do curso.

Agradeço também aos colegas e amigos, que compartilharam alegrias e dificuldades inerentes ao trabalho e ao estudo e mostraram-se sempre dispostos a prestar auxílio e a trocar valiosas experiências.

A todos vocês, minha sincera gratidão.

Luís Fernando Arcaro

SUMÁRIO

LISTA DE ACRÔNIMOS

API	*Application Programming Interface*
HTTP	*Hypertext Transfer Protocol*
JAR	*Java Archive*
JDK	*Java Development Kit*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE TRECHOS DE CÓDIGO

RESUMO

COLOCAR RESUMO

Palavras-chave: .

COLOCAR EM INGLÊS O TÍTULO DO TCC

ABSTRACT

COLOCAR RESUMO EM INGLÊS

Keywords:

.

1 INTRODUÇÃO

Primeiramente devemos entender que um Ambiente Virtual de Aprendizagem *AVA* é um local virtual que possui um conjunto de elementos tecnológicos, onde são disponibilizadas ferramentas que permitem o acesso a um ou mais cursos ou disciplinas de uma instituição de ensino. De modo geral, um AVA refere-se ao uso de recurso digitais de comunicação, principalmente, através de softwares educacionais via web que reúnem diversas ferramentas de interação (**??**). (OLIVEIRA et al., 2004; VALENTINI, SOARES, 2005)

O objetivo de um ambiente virtual de aprendizagem é de facilitar o acesso de alunos ao ensino, práticas de exercícios e livros online para consulta. Na Universidade de Caxias do Sul o AVA já é utilizado desde meados de 2005, onde é possível acessar os materiais disponibilizados pelos professores em suas respectivas disciplinas, podendo também acompanhar o cronograma, entre outras funcionalidades. (OLIVEIRA et al., 2004; VALENTINI, SOARES, 2005)

O AVA é utilizado pela UCS como uma ferramenta de gerenciamento de conteúdo, para apoio educacional. Outros softwares também oferecem apoio ao aprendizado, porém de forma mais direta, um desses softwares é o portal de algoritmos.

Em 2009, através dos professores Ricardo de Vargas Dorneles e Delcino Picinin Junior foi desenvolvido o portal de algoritmos da Universidade de Caxias do Sul, que é uma ferramenta de ensino e aprendizagem para os alunos de engenharias e computação da UCS. O aluno pode solucionar problemas algorítmicos através da linguagem português estruturado e submetê-los a avaliação do site, que oferece feedback ao aluno, de forma apoiar seu

aprendizado e desenvolvimento de sua lógica de programação. Além disso o aluno pode ver o seu desempenho em relação a outros alunos do portal através de um ranking. O professor pode cadastrar novos problemas e dados de testes, acompanhar o desempenho de cada aluno através das soluções submetidas pelos mesmos. (DORNELES et al., 2010)

Nesse trabalho serão desenvolvidas as funcionalidades do gerenciamento do portal de algoritmos, onde será possível cadastrar novos problemas algorítmicos, gerenciar usuários, gerenciar soluções submetidas pelos dos alunos, gerar relatórios. Além de documentar a sua programação e como utilizá-lo de forma correta. Visando a qualidade e usabilidade. Além do desenvolvimento da parte gerencial, existe o compilador do algoritmo que será desenvolvido por outro aluno no trabalho de conclusão de curso, sendo que o desenvolvimento será feito em paralelo e na mesma arquitetura de software.

A evolução das tecnologias de comunicação tem, constantemente, aumentado a disponibilidade e a velocidade da troca de informações entre dispositivos eletrônicos. A partir do desenvolvimento de múltiplas interfaces de rede pessoal, local, móvel e de longa distância como **IEEE!** *802.15 (Bluetooth)*, **IEEE!** *802.3 (Ethernet)* e **IEEE!** *802.11 (Wi-Fi)* entre outras, o acesso a redes e, consequentemente, à internet passou a ser facilitado. Hoje pode-se acessar a internet praticamente de qualquer local a uma velocidade satisfatória, além de trocar dados entre dispositivos instantaneamente (CARISSIMI, 2009; INTEL, 2011).

A grande diversidade de equipamentos dotados de interfaces de comunicação sugere a possibilidade do surgimento de uma nova era de conectividade, onde seria fundamental que todo equipamento fosse capaz de trocar informações com qualquer outro conectado à mesma rede, oferecendo e acessando serviços de forma simples e flexível (INTEL, 2011). Porém, atualmente toda interação entre esses dispositivos necessita ser desenvolvida utilizando uma ou mais linguagens de programação, o que não é uma tarefa simples para o profissional que pretende oferecer essa possibilidade. Além disso, as *APIs* (*Application Programming Interfaces*) de conectividade como, por exemplo, *Sockets* (para o

protocolo **TCP/IP!** - **TCP/IP!**) são complexas e de difícil utilização (DONAHOO; CALVERT, 2001).

Essa complexidade deve-se ao fato das *APIs* não oferecerem recursos simples para descoberta de dispositivos conectados à mesma rede, apresentarem diferente complexidade de uso para o dispositivo que pretende oferecer serviços e aquele que pretende fazer uso deles, além de permitirem apenas a troca de dados brutos (sequências de *bytes*) (DONAHOO; CALVERT, 2001). Sendo assim, tarefas como buscar e acessar serviços oferecidos por dispositivos conectados a uma mesma rede e trocar mensagens complexas entre eles são complicadas e trabalhosas para o desenvolvedor (CARISSIMI, 2009; DONAHOO; CALVERT, 2001).

Dessa forma, neste trabalho foi desenvolvida uma plataforma (aplicativos, protocolos e *APIs*) que possibilita aos dispositivos conectados a uma rede local através de uma interface de comunicação compatível com **TCP/IP!** (usualmente *Ethernet* ou *Wi-Fi*) a enumeração de todos os dispositivos conectados à mesma plataforma e a troca de pacotes contendo mensagens complexas. Vale ressaltar que, apesar dessa plataforma apresentar suporte apenas a dispositivos compatíveis com **TCP/IP!**, seu desenvolvimento foi feito de forma genérica, podendo ser adaptado a qualquer outra interface de conexão (*serial*, *Bluetooth*, etc.). Para isso, é necessária apenas a implementação de um novo aplicativo adaptador e/ou de uma nova biblioteca de acesso ao núcleo da plataforma.

A implementação da plataforma em questão foi realizada utilizando a linguagem *Java*. Optou-se pelo uso dessa linguagem pois, com base em sua documentação, verificou-se que essa oferece todos os recursos necessários para o alcance dos objetivos propostos, além de poder ser executada em diferentes plataformas sem alteração significativa do código-fonte (ORACLE, 2011). Vale ressaltar que, apesar de a plataforma ser desenvolvida em *Java*, ela pode ser acessada por aplicativos desenvolvidos em qualquer linguagem que possam ser executados em máquinas equipadas com interface de comunicação compatível com **TCP/IP!**. Para tanto, basta que seja desenvolvida uma biblioteca de acesso à plataforma na

linguagem escolhida.

Os protocolos utilizados pela plataforma suportam operações de aquisição e busca de identificadores de dispositivos, publicação e busca de serviços, e transporte de dados complexos. Os protocolos em questão são baseados em conteúdo binário, visando minimizar o *overhead* de transmissão causado por sua utilização, e foram desenvolvidos de forma a permitir que novos protocolos, caso tornem-se necessários, possam ser introduzidos na plataforma de maneira simples. Além disso, os pacotes de dados utilizados na comunicação suportam o armazenamento de múltiplos tipos de dados complexamente estruturados. Esses consistem em mapas de elementos nomeados, onde cada elemento pode conter tipos primitivos, *strings*, valor nulo (*null*), assim como pacotes aninhados e vetores unidimensionais de todos os tipos anteriormente citados.

A fim de evitar incompatibilidades com implementações de bibliotecas de acesso à plataforma em diferentes linguagens, esses pacotes não suportam tipos de dados cujo formato seja específico de uma determinada linguagem (por exemplo objetos serializáveis em *Java*) e apresentam tratamento especial para aqueles cuja definição seja conflitante entre duas linguagens popularmente utilizadas (por exemplo o tipo *char*, cujo tamanho em *bytes* difere nas linguagens *Java* e *C*). Acredita-se que, através da utilização dos protocolos propostos, o usuário encontre facilidade no desenvolvimento de aplicativos que façam uso da plataforma.

Recursos como a criptografia de dados, a transmissão de dados de um remetente a múltiplos destinatários e o tratamento de outros tipos de interfaces de comunicação, apesar de serem considerados importantes para a operação da plataforma (COMVERSE, 2010), não foram implementados neste trabalho. Além disso, a *API* de acesso à plataforma foi desenvolvida apenas em linguagem *Java* para execução em microcomputadores e dispositivos móveis equipados com o sistema operacional *Android*. Essas decisões foram necessárias a fim de limitar o escopo do trabalho. Todos os pontos citados representam, portanto, oportunidades de melhorias futuras no protótipo desenvolvido.

O usuário final do produto deste trabalho é o desenvolvedor de *software* que necessita de uma solução para comunicação entre dispositivos e que não deseja investir tempo no desenvolvimento da plataforma necessária. Ao usuário, esse produto apresenta-se na forma de uma *API* que fornece um conjunto de métodos para que todas as operações suportadas pela plataforma sejam executadas.

O Trecho de Código **??** demonstra a forma como a *API* desenvolvida é utilizada pelo programador. Nesse código são exemplificadas tarefas como conectar à plataforma (*cmdStart* - linha **??**) e publicar um serviço que poderá ser acessado por outros dispositivos (*cmdPublishService* - linha **??**). Além disso, demonstra as chamadas necessárias para a enumeração dos dispositivos conectados ao sistema (*cmdSearchIdentifiers* - linha **??**) e dos que oferecem um determinado serviço (*cmdSearchService* - linha **??**), para o estabelecimento de uma conexão a um serviço (*cmdConnect* - linha **??**), envio de uma requisição (*cmdSend* - linha **??**) e recebimento da resposta (*cmdReceive_DTPPacket* - linha **??**).

Para a validação da solução, foi desenvolvido um aplicativo para a exploração extensiva da plataforma, ou seja, que permite a execução de todas as funções suportadas pela *API* e exibe os dados brutos resultantes dessa interação. Além desse aplicativo, foi desenvolvido um aplicativo de bate-papo que possibilita a demonstração da comunicação entre múltiplos dispositivos.

1.1 Objetivos

1.1.1 Objetivo geral

Este trabalho tem por objetivo realizar a evolução da ferramenta de gerenciamento do portal de algoritmos, visando substituir tecnologias defasadas.

1.2 Estrutura do trabalho

COLOCAR ESTRUTURA DO TRABALHO

2 REFERENCIAL TEÓRICO

Nas últimas décadas, a evolução e a disseminação dos computadores ocorreu em dimensões mundiais, causando enormes transformações na forma de trabalho de escritórios e fábricas, além de passarem a ser utilizados em ambiente doméstico com objetivos diversos. Em escala semelhante e intimamente ligadas a essa disseminação, novas e melhores tecnologias de transmissão de dados permitiram que esses equipamentos trocassem informações a velocidades e distâncias cada vez maiores e a custos cada vez mais acessíveis (TANENBAUM, 2011; CARISSIMI, 2009).

Neste capítulo, serão apresentados conceitos relacionados à transmissão de dados e às redes de computadores, assim como uma breve introdução ao funcionamento do *software* relacionado a elas, a fim de proporcionar uma melhor compreensão do problema tratado neste trabalho. Porém, detalhes do funcionamento das redes de computadores não são o foco principal e, portanto, não serão abordados. Para maiores informações, sugere-se TANENBAUM (2011) e CARISSIMI (2009).

2.1 Protocolos de Comunicação

Redes de computadores são conjuntos de computadores autônomos interconectados (capazes de transmitir informações entre si) por uma determinada tecnologia de transmissão. Atualmente existem redes de computadores de diversos tipos, velocidades e abrangências. Como exemplo de redes de computadores podem ser citadas as tecnologias *Ethernet*

(utilizada para redes locais cabeadas) e *Wi-Fi* (utilizada para redes locais sem fio) (KUROSE; ROSS, 2006).

O serviço oferecido pelo equipamento (*hardware*) em uma rede de computadores limita-se à transmissão de informações cruas (sequências de *bits*) entre um conjunto de computadores interconectados. A fim de tornar essa funcionalidade útil a um determinado fim, é necessário que esses computadores troquem dados seguindo um formato pré-estabelecido, ou seja, de acordo com uma estrutura definida e "compreendida" por todos os computadores da rede e que permita maximizar as possibilidades de uso e minimizar a complexidade de utilização dela. Essa tarefa é desempenhada no nível de *software* de rede, e remete ao conceito de Protocolo de Comunicação (TANENBAUM, 2011).

Um Protocolo de Comunicação pode ser definido como uma convenção entre diversos computadores interconectados que define o formato no qual deverão ser transmitidos os dados durante a comunicação entre eles (KUROSE; ROSS, 2006). O *software* de rede utilizado em cada *hardware* de rede, responsável pela implementação dos protocolos, varia de acordo com a tecnologia empregada para comunicação; porém, de maneira geral, pode-se afirmar que o *software* de rede está estruturado em camadas, sendo que cada camada é responsável por abstrair uma ou mais características específicas da comunicação entre computadores, fornecendo um serviço que oferece determinadas garantias à camada superior (TANENBAUM, 2011).

O modelo de referência padrão para a implementação da estrutura de protocolos é denominado **OSI!** (**OSI!**) e seu desenvolvimento foi presidido pela **ISO!** (**ISO!**) a partir de 1978. Esse modelo sugere a utilização de sete camadas de abstração, podendo ser adicionadas ou removidas camadas de acordo com a necessidade (CARISSIMI, 2009). A Tabela 2.1 apresenta as camadas do modelo **OSI!**, assim como a descrição da função de cada uma delas. A numeração das camadas (coluna 1 da Tabela 2.1) pode ser interpretada como o nível de abstração que ela representa em relação à transmissão oferecida pelo *hardware*, valendo observar que o usuário interage sempre com a camada de maior nível de abstração (no caso, a camada de Aplicação) (TANENBAUM, 2011).

Tabela 2.1: Camadas do modelo de referência *OSI* (TANENBAUM, 2011)

Camada		Função
7	Aplicação	Camada utilizada para transporte efetivo dos dados de interesse do usuário.
6	Apresentação	Define a sintaxe e a semântica dos dados transmitidos a fim de tornar possível a comunicação entre computadores que utilizam diferentes representações internas.
5	Sessão	Permite que usuários em diferentes computadores estabeleçam sessões de comunicação, oferecendo serviços de controle do diálogo entre eles.
4	Transporte	Fragmenta os pacotes de dados para transmissão e oferece um canal fim-a-fim confiável (livre de erros).
3	Rede	Determina a forma como os pacotes de dados devem ser roteados entre a origem e o destino e gerencia a demanda de transmissão de dados a fim de evitar gargalos.
2	Enlace	Transfere unidades de dados do serviço de enlace entre diversos circuitos da camada física, além de detectar e corrigir erros dessa camada.
1	Física	Meio mecânico ou elétrico para a transmissão de *bits* entre entidades de enlace.

Tomando como exemplo prático as redes locais cabeadas (usualmente *Ethernet*), o *software* utilizado nas redes atuais permite que um usuário de um computador, conectado a uma rede devidamente configurada, seja capaz de requerer a transmissão de um pacote de dados até outro computador alcançável através da estrutura de rede, desde que seja conhecido o endereço de rede do destinatário. Na pilha de protocolos utilizada nessas redes, as camadas de sessão e apresentação sugeridas pelo modelo **OSI!** foram omitidas e, sendo assim, o serviço de transmissão de pacotes de dados fim-a-fim é oferecido pela camada de transporte, para a qual os protocolos mais comumente utilizados são o **TCP/IP!** e o **UDP/IP!** (**UDP/IP!**) (KUROSE; ROSS, 2006).

O protocolo da camada de transporte **TCP!** (**TCP!**), normalmente utilizado em conjunto com o protocolo da camada de rede **IP!** (**IP!**) – cuja associação é chamada simplesmente de protocolo **TCP/IP!** – baseia-se em conexões virtuais. Essa estratégia permite oferecer a garantia de que todos os pacotes que trafegarem em uma conexão estão íntegros, corretamente sequenciados e não duplicados, o que o torna adequado para aplicações onde é exigida garantia na transmissão de dados (por exemplo transferência de arquivos, onde a perda de pacotes não é admitida). Já o protocolo **UDP/IP!** oferece o serviço de entrega de pacotes de dados entre processos executados em computadores interconectados e o controle de integridade através de *checksum*, porém não dá garantias sobre a entrega dos pacotes, a ordem em que eles serão entregues ou a entrega desses sem duplicação. Por isso, o protocolo **UDP/IP!** é mais comumente utilizado em aplicações onde a velocidade de

transmissão é prioridade e a garantia dela é secundária, como o envio de fluxos de amostras de áudio e de quadros de vídeo, onde a perda de pacotes tem baixo impacto (CARISSIMI, 2009).

2.1.1 Protocolo TCP/IP!

Neste trabalho optou-se pela utilização do protocolo **TCP/IP!** para transporte de dados entre as partes do sistema, a fim de usufruir das garantias oferecidas por esse protocolo e a pilha associada a ele. Dessa forma, não foi necessário o desenvolvimento de estratégias de controle que garantissem a entrega, a integridade e o correto sequenciamento das informações trocadas na plataforma.

Assim como os demais protocolos de rede utilizados nas camadas inferiores da pilha, o protocolo **TCP/IP!** não permite que o usuário enumere os dispositivos conectados à rede ou seja informado sobre quais serviços estão sendo executados por eles. Ao invés disso, o usuário deve conhecer o endereço de rede e a porta (identificador de processo) que deverá atender à sua tentativa de transmissão, correndo o risco de não receber resposta caso essas informações estejam incorretas (DONAHOO; CALVERT, 2001). Além disso, o serviço oferecido pelo protocolo **TCP/IP!** é capaz de tratar apenas dados brutos (sequências de *bytes*) (COMER, 2007; DONAHOO; CALVERT, 2001), tornando necessária a definição de protocolos na camada de aplicação que convencionem o significado dos dados transmitidos e fazendo com que qualquer mensagem complexa a ser transmitida deva ser encapsulada em uma sequência de *bytes* que possa ser interpretada pelo receptor como tal (COMER, 2007; TANENBAUM, 2011). Dessa forma, neste trabalho foi desenvolvida uma plataforma capaz de contornar essas limitações, permitindo que tipos diferentes de dispositivos detectem-se e comuniquem-se de forma simplificada através de uma rede.

O desconhecimento dos dispositivos presentes na rede e dos serviços fornecidos por eles acaba por tornar a rede de computadores uma mera forma de transmitir fluxos de dados demandados de forma específica pelo usuário, não facilitando de forma alguma a interação entre os diversos equipamentos

dotados de interface de rede utilizados hoje. Existem esforços direcionados à definição de mecanismos de descoberta automática desse tipo de informação, os quais podem ser exemplificados pelos protocolos **UPnP!** (**UPnP!**) e **SLP!** (**SLP!**) (BETTSTETTER; RENNER, 2000).

O protocolo **UPnP!**, que baseia-se no protocolo *HTTP* (*Hypertext Transfer Protocol*) transportado sobre **UDP/IP!**, utiliza mensagens *multicast* (entregues a múltiplos destinatários) para notificar os demais dispositivos sobre os serviços que oferece e buscar serviços oferecidos pelos demais; essa abordagem acaba por não oferecer robustez para a notificação da desconexão de dispositivos, já que esses podem desconectar-se da rede sem antes enviar uma notificação (PSINAPTIC, 2004), além de provocar um grande volume de tráfego de rede. Já o protocolo **SLP!**, transportado sobre **TCP/IP!**, oferece maior robustez, porém exige a utilização de uma arquitetura dedicada ao seu funcionamento, o que torna sua utilização complexa (BETTSTETTER; RENNER, 2000). Devido a essas limitações, optou-se pelo desenvolvimento de protocolos específicos de enumeração dos dispositivos conectados à rede.

Neste trabalho foram desenvolvidos protocolos transportados sobre **TCP/IP!** que permitem a descoberta dos dispositivos presentes na rede e dos serviços oferecidos por eles, de forma que um dispositivo corretamente conectado e registrado na plataforma possa obter uma lista atualizada de serviços dos quais pode fazer uso, além de ser notificado sobre a desconexão de um dispositivo ao qual estava conectado. Seria possível desenvolver também, com a mesma robustez, métodos de registro para notificação sobre o ingresso de dispositivos e serviços na plataforma, permitindo que um determinado usuário estivesse sempre atualizado sobre essas informações sem necessitar a execução repetida de consultas, porém esse desenvolvimento não foi realizado devido a restrições de tempo.

2.1.2 Problemas de Representação de Dados

Durante a comunicação através de redes de computadores utilizando-se um determinado protocolo de comunicação, as

informações transmitidas devem ser representadas em forma de sequências de *bytes* que permitam ao receptor decodificá-las, possibilitando a regeneração da mensagem original. Isso é feito através da definição de tipos de dados, que convencionam a forma como uma determinada informação (textos, números ou caracteres, por exemplo) será enviada pelo transmissor e deverá ser interpretada pelo receptor. Esse tipo de convenção é necessário pois, em alguns casos, existem incompatibilidades na forma como um determinado tipo de dados é tratado em diferentes linguagens de programação e/ou arquiteturas de computadores, o que pode ocasionar problemas na troca dessas informações. Um exemplo de incompatibilidade dessa natureza é a forma de representação de caracteres nas linguagens *Java*, que representa-os em 2 bytes, e *C*, que utiliza apenas 1 byte para armazenar o mesmo tipo (ORACLE, 2011; SAVITCH, 2004).

Raramente os dados a serem transmitidos através de uma rede limitam-se a aglomerados simples de dados, que são o conteúdo mais facilmente transmitido ao fazer-se uso de *APIs* de rede comumente utilizadas como, por exemplo, *Sockets*. Ao invés disso, é comum em aplicações práticas a necessidade de transmissão de árvores de dados, ou seja, aglomerados de dados que contêm informações estruturadas em níveis de forma que cada entrada possua outras entradas aninhadas. Por exemplo, para transmissão de uma estrutura de diretórios é necessário que sejam transmitidas informações sobre o conteúdo de um diretório raiz, porém esse diretório pode conter outros diretórios cujas informações também devem ser transmitidas, e esses também podem possuir diretórios filhos, tornando esse um problema recursivo. Fazendo-se uso de *APIs* tradicionais, cabe ao programador definir uma forma de desempenhar essa tarefa utilizando os recursos disponíveis, o que pode tornar-se bastante complexo e custoso (DONAHOO; CALVERT, 2001).

Existem formatos capazes de simplificar a tarefa de transmissão de aglomerados complexos de dados, como **XML!** (**XML!**) e **CORBA!** (**CORBA!**). O formato **XML!** permite a representação de árvores de dados e é facilmente legível e editável por humanos, porém para isso utiliza formato texto, o que acaba por tornar sua interpretação custosa

computacionalmente, já que exige que as informações textuais sejam convertidas, além de não permitir a diferenciação dos tipos dos dados transmitidos e, por isso, dificultar o acesso a eles. Por sua vez, o protocolo da arquitetura **CORBA!** utiliza o formato binário para representação dos dados, o que lhe confere uma interpretação muito veloz, além de utilizar formas de representação que garantem a interpretação dos dados em diferentes arquiteturas de computador; porém, essa arquitetura exige uma infraestrutura complexa para que seja utilizada e, além disso, a forma como os pacotes de dados são definidos, transmitidos e interpretados faz com que qualquer campo adicionado a um pacote deva ser ajustado em todas as aplicações que o utilizam, tornando-a pouco flexível (ARCINIEGAS, 2002; OMG, 2011).

Neste trabalho foi desenvolvido um protocolo baseado em conteúdo binário capaz de transmitir aglomerados de pares chave/valor contendo diversos tipos de dados explicitamente identificados, minimizando o custo de interpretação, garantindo que a parte receptora seja capaz de interpretá-los e que permite a adição de campos ao protocolo de forma flexível. Os tipos de dados suportados por esse protocolo são aqueles presentes nas linguagens mais comumente utilizadas (tomou-se como base as linguagens *Java* e *C*) e vetores unidimensionais deles, geralmente representados segundo os formatos adotados pela linguagem *Java*. Além disso, esse protocolo oferece suporte à transmissão de mensagens aninhadas, tornando possível ao usuário a troca de árvores de dados em um único pacote e maximizando a flexibilidade no intercâmbio de informações na plataforma.

Para tipos textuais (caracteres e cadeias de caracteres) é oferecido suporte diferenciado para caracteres de tamanho 1 *byte* (**ASCII! - ASCII!**, característicos da linguagem *C*) e de 2 *bytes* (*Unicode*, suportados pela linguagem *Java*). Optou-se por não oferecer suporte a vetores multidimensionais (matrizes), uma vez que sua forma de tratamento difere em função da linguagem. Essas restrições foram necessárias a fim de garantir que os dados transmitidos possam ser interpretados por bibliotecas desenvolvidas em qualquer linguagem e executadas em qualquer arquitetura, e podem ser observadas também na arquitetura

CORBA! e nos protocolos utilizados na implementação do **RPC!** (**RPC!**) (THURLOW, 2009; OMG, 2011).

3 ARQUITETURA DA PLATAFORMA

No desenvolvimento deste trabalho, definiu-se que o protocolo **TCP/IP!** seria utilizado na interconexão entre os componentes da plataforma, já que a garantia de entrega na transmissão de dados é imprescindível. Os recursos oferecidos por esse protocolo e a pilha associada a ele evitaram a necessidade do desenvolvimento de estratégias de controle de erros de transmissão nas comunicações fim-a-fim, fornecendo canais confiáveis de transporte de dados.

A plataforma é composta por três aplicativos interconectados através do protocolo **TCP/IP!** de acordo com a Figura 3.1. Nessa representação, as setas indicam conexões **TCP/IP!** e partem dos **Pontos de Acesso** e **Pontos-Fim** (clientes que solicitam conexão a um servidor) e alcançam os **Pontos de Acesso** e **Roteador** (servidores que aguardam conexões de um ou mais clientes), sendo possível verificar também que cada **Roteador** ou **Ponto de Acesso** é capaz de atender a múltiplos clientes simultaneamente e que os **Pontos de Acesso** atuam tanto como servidores dos **Pontos-Fim** quanto como clientes do **Roteador**. Entre cada par de aplicativos é estabelecida apenas uma conexão **TCP/IP!**, de forma a minimizar o *overhead* de gerenciamento dessas, através da qual são transmitidos todos os pacotes de dados.

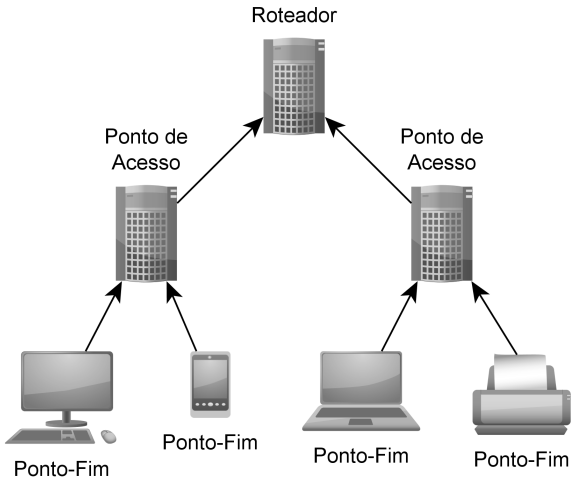


Figura 3.1: Arquitetura da plataforma

Na plataforma desenvolvida, o **Roteador** é responsável por centralizar informa-ções sobre o estado do sistema e pelo roteamento de pacotes entre os diferentes **Pontos de Acesso** conectados a ele. Os **Pontos de Acesso** são responsáveis por intermediar a interação entre o **Roteador** e os **Pontos-Fim**, repassando mensagens recebidas dos **Pontos-Fim** ao **Roteador** e encaminhando aquelas recebidas do **Roteador** aos **Pontos-Fim**. Por fim, os **Pontos-Fim** são bibliotecas utilizadas pelo usuário da plataforma a fim de acessá-la e usufruir dos recursos oferecidos, ou seja, são *softwares* de utilização da plataforma. Nas próximas seções, cada uma das partes do sistema será detalhada e será introduzida a forma como essas partes interagem para o funcionamento da plataforma.

3.1 Roteador

O sistema possui um servidor central, denominado **Roteador**, que recebe conexões **TCP/IP!** de múltiplos **Pontos de Acesso** na porta 1236. Esse aplicativo é responsável por efetuar o roteamento de pacotes de dados, além de gerenciar, armazenar e fornecer informações globais sobre os dispositivos registrados na plataforma. A opção pela utilização de um servidor central para esse fim, apesar de possivelmente diminuir o desempenho do sistema e introduzir um ponto de falha, foi feita de forma a facilitar o gerenciamento das informações de estado e torná-lo o

centro de controle, reconhecimento e notificação de eventos.

Uma importante função do **Roteador** é o gerenciamento dos identificadores de dispositivos, que são utilizados de forma análoga aos endereços de rede no protocolo **TCP/IP!**, para determinar origem e destino dos dados transmitidos. Cabe ao **Roteador** a geração de novos identificadores para os dispositivos que conectam-se à plataforma, garantindo a unicidade deles, além do armazenamento do endereço do **Ponto de Acesso** ao qual cada um dos dispositivos está conectado, permitindo o encaminhamento dos pacotes destinados a cada um deles e o fornecimento de respostas às solicitações de busca de dispositivos suportadas pelo sistema. Além disso, o **Roteador** apresenta estruturas internas para o armazenamento de informações sobre os serviços oferecidos por cada um dos dispositivos, sendo capaz de divulgá-las aos demais.

Devido às tarefas vitais que desempenha, o **Roteador** pode ser considerado o núcleo da plataforma, fazendo com que a ocorrência de falhas em sua execução comprometa o funcionamento de todo o sistema. Sendo assim, é fundamental que sua probabilidade de falha seja a mínima possível, evitando-se agregar a esse aplicativo quaisquer recursos que possam torná-lo menos estável e devendo garantir-se que não seja possível acessá-lo de forma ilícita. A fim de tornar o acesso ao **Roteador** restrito, seria adequada a utilização de um protocolo de segurança como **SSL!** (**SSL!**) ou **TLS!** (**TLS!**) – que fornecem autenticação e criptografia na camada de transporte (CARISSIMI, 2009; DIERKS; RESCORLA, 2008; FREIER; KARLTON; KOCHER, 2011) – sobre as conexões **TCP/IP!** existentes entre **Pontos de Acesso** e o **Roteador**. Porém, devido a restrições de tempo essa característica não foi desenvolvida, sendo assunto para melhoria futura.

3.2 Ponto de Acesso

Os **Pontos de Acesso** são, no contexto da plataforma, aplicativos de repasse bidirecional de informações entre o **Roteador** e um conjunto de **Pontos-Fim**. Todo **Ponto de Acesso** age como cliente **TCP/IP!** do **Roteador**,

estabelecendo uma conexão através da qual trafegam pacotes de dados em ambas as direções. Sendo assim, a função dos **Pontos de Acesso** é receber dados dos **Pontos-Fim** com os quais mantêm comunicação, executar validações sobre esses dados e encaminhá-los ao **Roteador** em forma de pacotes válidos segundo a definição de protocolos da plataforma, assim como a recepção de pacotes do **Roteador** e encaminhamento ao **Ponto-Fim** de destino.

Levando em conta unicamente o protocolo **TCP/IP!**, a utilidade dos **Pontos de Acesso** não é facilmente percebida, podendo esses ser considerados mero *overhead* de transmissão, uma vez que os pacotes **TCP/IP!** são apenas repassados sem alterações através do **Ponto de Acesso**. Porém, considerando-se que outras interfaces de comunicação (*serial* e *Bluetooth*, por exemplo) poderão futuramente ser suportadas pela plataforma, sua utilização torna-se fundamental, uma vez que esses são responsáveis por isolar a complexidade de integração dessas interfaces, por permitir a conexão de novos tipos de dispositivos à plataforma sem interromper sua execução, além de tornarem-se uma camada de "proteção" para o **Roteador**, evitando que dados inválidos sejam recebidos por ele.

A fim de tornar possível o encaminhamento de pacotes de dados, os **Pontos de Acesso** armazenam o identificador e o endereço de rede (ou outro tipo de endereço, dependendo do protocolo de que tratam) de cada um dos dispositivos a eles conectados. Para obter essas informações, o aplicativo faz uso dos pacotes trocados entre o **Roteador** e os dispositivos que carregam esses dados.

Neste trabalho foi desenvolvido o **Ponto de Acesso TCP/IP!**, que atua como servidor para os **Pontos-Fim TCP/IP!**, recebendo conexões desses na porta 1237. O **Ponto de Acesso** recebe pacotes de **Pontos-Fim**, efetua validações mínimas e os encaminha ao **Roteador** para tratamento, além de receber pacotes do **Roteador** e encaminhá-los ao **Ponto-Fim** de destino. Vale ressaltar que, apesar do **Ponto de Acesso TCP/IP!** poder receber pacotes de um **Ponto-Fim** endereçados a outro **Ponto-Fim** também conectado a ele, esses pacotes são encaminhados ao **Roteador** para tratamento, não podendo ser diretamente repassados entre os dois **Pontos-Fim**

pelo próprio **Ponto de Acesso**. Através dessa forma de comunicação, é possível manter o controle do **Roteador** sobre todo o tráfego de dados na plataforma.

3.3 Ponto-Fim

Para que um dispositivo seja capaz de utilizar a plataforma, esse deve fazer uso de uma biblioteca que forneça suporte à interação com um **Ponto de Acesso** e, indiretamente, com o **Roteador**. Nesta arquitetura, essa biblioteca é denominada **Ponto-Fim** e tem a função de fornecer ao usuário (programador) uma *API* de acesso aos recursos do sistema de fácil utilização, que exponha de forma simples as operações suportadas pela plataforma e abstraia de forma eficiente a complexidade ligada ao seu funcionamento. Dessa forma, o **Ponto-Fim** é o aplicativo responsável por permitir que o usuário conecte-se a um **Ponto de Acesso** através da infraestrutura de rede à qual se destina, registre-se na plataforma, publique e busque serviços, e envie ou receba mensagens complexas através do sistema. Para isso, o **Ponto-Fim** implementa mecanismos de construção, interpretação e controle dos pacotes utilizados pelos protocolos da plataforma.

Na arquitetura da plataforma são denominados serviços os aplicativos executados por um **Ponto-Fim** que são publicados no sistema para serem acessados por outros dispositivos. De forma semelhante às portas no protocolo **TCP/IP!**, cada serviço possui um identificador estabelecido por convenção que é utilizado para permitir o acesso dos **Pontos-Fim** a ele, devendo esse ser conhecido por todo dispositivo que desejar acessá-lo. É possível a cada **Ponto-Fim** publicar quantos serviços necessitar a fim de desempenhar suas funções, desde que cada um deles possua um identificador distinto. Diferentemente do protocolo **TCP/IP!**, no qual o usuário não é capaz de determinar quais os dispositivos conectados à rede que possuem um determinado serviço em execução, na plataforma desenvolvida é possível, de posse do identificador do serviço desejado, solicitar ao **Roteador** os identificadores dos **Pontos-Fim** que publicaram esse serviço, permitindo uma interação facilitada entre eles.

Neste trabalho foi desenvolvido o **Ponto-Fim TCP/IP!** para microcomputadores e para dispositivos móveis (celulares e *tablets*, por exemplo) equipados com o sistema operacional *Android*, ambos em linguagem *Java*. É importante ressaltar que, para cada **Ponto de Acesso** que suporte diferentes interfaces de conexão futuramente criado, deverá ser elaborado também o **Ponto-Fim** adequado, de forma a permitir que o usuário do sistema seja capaz de utilizá-lo de forma genérica através da nova interface.

3.4 Funcionamento da plataforma

Para que a plataforma seja utilizada, o **Roteador** e um ou mais **Pontos de Acesso** devem estar em execução e a conexão **TCP/IP!** entre esses aplicativos deve encontrar-se estabelecida. Todos os **Pontos de Acesso** mantêm permanentemente ativa a conexão **TCP/IP!** com o **Roteador**, ou seja, caso essa conexão seja interrompida por algum motivo, o **Ponto de Acesso** tentará periodicamente conectar-se novamente até que a conexão seja restabelecida, mantendo a máxima disponibilidade da plataforma. Os dispositivos conectados a um **Ponto de Acesso** cuja conexão com o **Roteador** for interrompida são considerados desconectados da plataforma, uma vez que não há garantia da manutenção do estado do sistema. O diagrama **UML!** (**UML!**) de sequência apresentado na Figura 3.2 mostra o processo de conexão entre **Pontos de Acesso** e o **Roteador**, exemplificando uma interrupção da conexão entre eles.

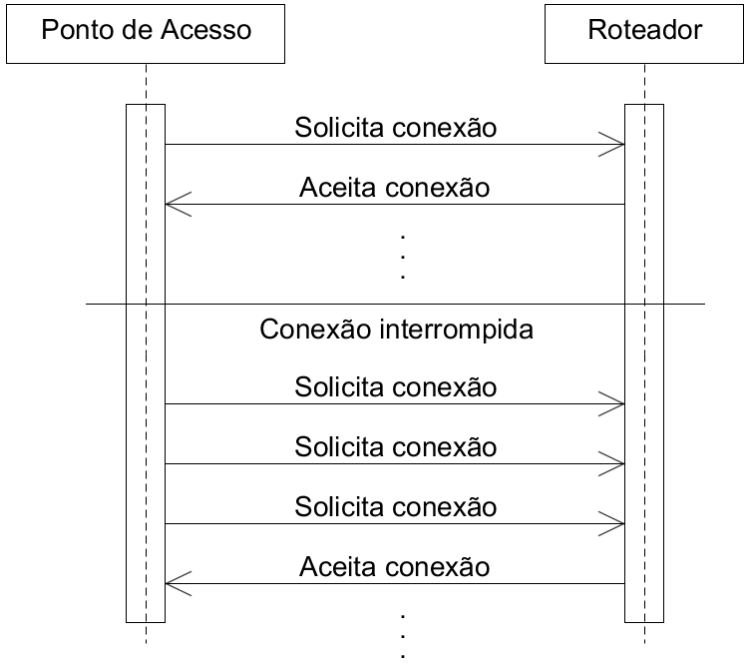


Figura 3.2: Conexão do Ponto de Acesso ao Roteador

Estando a plataforma em operação, ou seja, estando o **Roteador** e os **Pontos de Acesso** em execução e devidamente conectados, os **Pontos-Fim** podem estabelecer comunicação com os **Pontos de Acesso** através da infraestrutura de comunicação para a qual destinam-se (**TCP/IP!**, *serial* ou *Bluetooth*, por exemplo). Estabelecida a comunicação entre o **Ponto-Fim** e um **Ponto de Acesso**, o **Ponto-Fim** deve utilizar o **PGI!** (**PGI!**), descrito na Seção 4.1, para registrar-se na plataforma, obtendo assim um identificador. A Figura 3.3 apresenta o diagrama **UML!** de sequência que demonstra o processo de estabelecimento de comunicação entre o **Ponto-Fim** e um **Ponto de Acesso** e o registro do **Ponto-Fim** na plataforma através do protocolo **PGI!**.

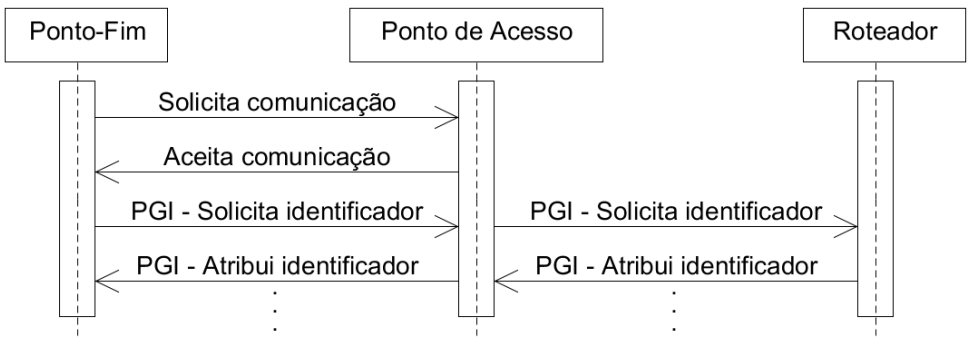


Figura 3.3: Registro de um Ponto-Fim

Uma vez registrado na plataforma, o **Ponto-Fim** está apto a interagir através dela e será visível aos demais dispositivos que efetuarem buscas através do protocolo **PGI!**. Porém, os serviços executados não se encontram acessíveis até que seja utilizado o **PPS!** (**PPS!**), descrito na Seção 4.2, para efetuar a publicação dos serviços que são efetivamente oferecidos aos demais **Pontos-Fim**. Além disso, o **Ponto-Fim** pode utilizar esse mesmo protocolo para buscar serviços oferecidos pelos outros dispositivos. A Figura 3.4 demonstra a utilização do protocolo **PPS!** para publicação de dois serviços e busca dos dispositivos que oferecem um determinado serviço.

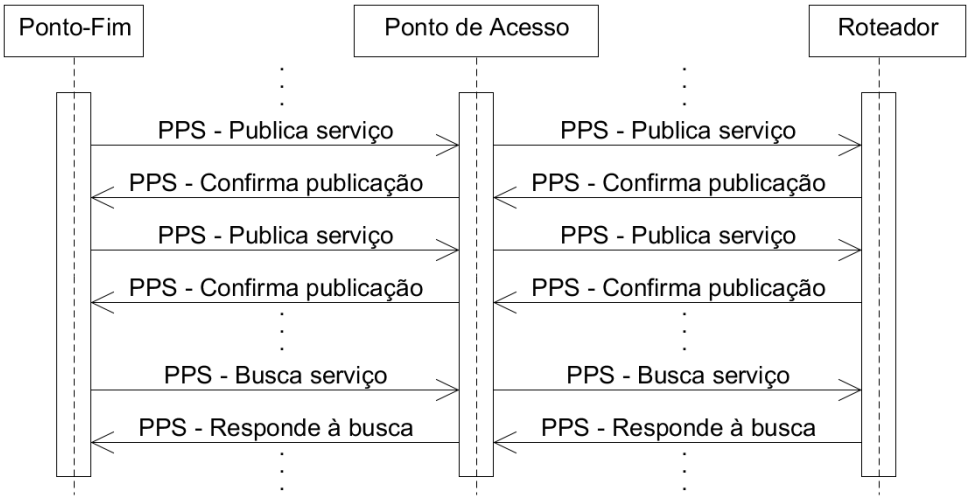


Figura 3.4: Publicação e busca de serviços

Uma vez que o **Ponto-Fim** localizar um dispositivo que possui um serviço do qual deseja usufruir, deverá ser estabelecida uma conexão de transporte com ele e, então, poderá ser utilizado o **PTD!** (**PTD!**) para trocar pacotes de dados complexos com esse serviço. Uma vez completada a negociação com o serviço em questão, a conexão de transporte deve ser finalizada. A Figura 3.5 demonstra a utilização do **PTD!** para estabelecimento de uma conexão, troca de dados e término dessa conexão.

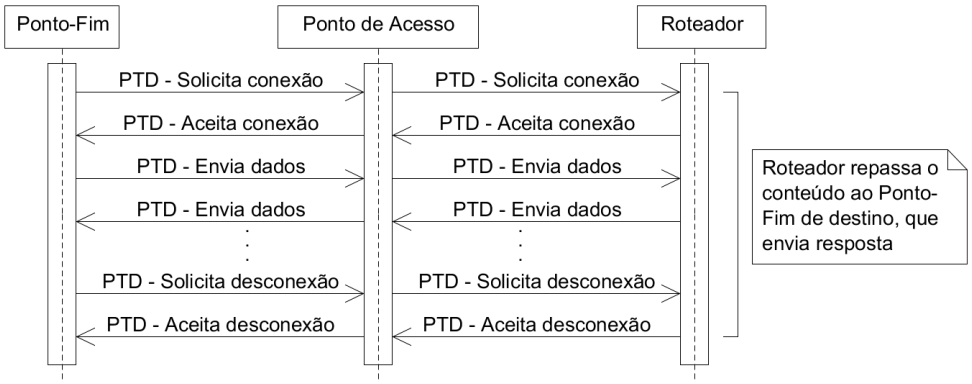


Figura 3.5: Transporte de dados

No próximo capítulo serão descritos os protocolos de aplicação desenvolvidos para utilização na plataforma a fim de alcançar os objetivos propostos. Mais especificamente, o **PGI!**, o **PPS!** e o **PTD!**, assim como detalhes de sua utilização, serão descritos nas Seções 4.1, 4.2 e 4.3, respectivamente.

4 PROTOCOLOS DE APLICAÇÃO

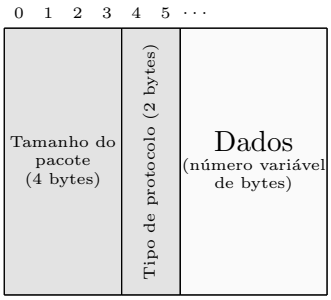
Neste trabalho foram desenvolvidos protocolos de aplicação que podem ser utilizados sobre diversas interfaces de comunicação (**TCP/IP!**, *serial* e *Bluetooth*, por exemplo) que possibilitam o gerenciamento e a busca de identificadores dos dispositivos conectados à plataforma, a publicação e busca de serviços oferecidos por eles, além de um protocolo de transmissão de mensagens complexas que facilita a comunicação entre esses equipamentos.

Os protocolos de aplicação da plataforma foram desenvolvidos utilizando o formato binário, ou seja, esses não utilizam dados em formato texto, de forma a minimizar o processamento necessário para acessar as informações contidas nos pacotes. Outras estratégias, como a utilizada pelo protocolo *HTTP* (que possui cabeçalhos baseados em texto) tornam o protocolo mais facilmente legível por humanos, porém exigem a conversão das informações para o formato binário durante a interpretação dos pacotes, o que aumenta consideravelmente o custo computacional dessa tarefa (TANENBAUM, 2011).

A fim de permitir a multiplexação de pacotes (transmissão de múltiplos tipos de pacotes através de um mesmo canal de comunicação), todos os protocolos da plataforma atendem a um formato básico (TANENBAUM, 2011). O formato básico utilizado na plataforma, formalizado na Tabela 4.1, convenciona que todo pacote contém em seu cabeçalho o tamanho do pacote e o tipo de protocolo transmitido no campo de dados, de forma a simplificar sua separação no fluxo contínuo de *bytes* fornecido pela interface de comunicação. Vale destacar que os valores numéricos são transmitidos em formato *big-endian*, ou seja, do

byte mais significativo ao menos significativo.

Tabela 4.1: Formato básico dos pacotes



Campo	Descrição
Tamanho do pacote	Tamanho do pacote excluindo-se o próprio campo.
Tipo de protocolo	Tipo do protocolo ao qual o pacote pertence, destinado à expansibilidade da gama de protocolos. Os valores possíveis são: **0x0001** Protocolo de Gerenciamento de Identidade; **0x0002** Protocolo de Publicação de Serviços; **0x0003** Protocolo de Transporte de Dados.
Dados	Dados transportados pelo pacote, especificados por cada protocolo.

Os tipos de protocolo discriminados pelo pacote básico incluem três diferentes protocolos. O **PGI!** (**PGI!**) é responsável pela solicitação e atribuição de identificadores aos dispositivos conectados, além de suportar consultas básicas a essas informações por parte de qualquer dispositivo. O **PPS!** (**PPS!**) possibilita a publicação de serviços oferecidos pelos dispositivos registrados na plataforma, além de suportar consultas a essas informações. Por fim, o **PTD!** (**PTD!**) permite aos dispositivos trocar mensagens complexas, sendo o protocolo de transporte de carga útil do sistema. Nas próximas seções serão detalhados esses protocolos e a forma como devem ser utilizados na plataforma.

4.1 PGI - Protocolo de Gerenciamento de Identidade

Para que um **Ponto-Fim** seja capaz de transmitir dados na plataforma é necessário que, primeiramente, esse estabeleça

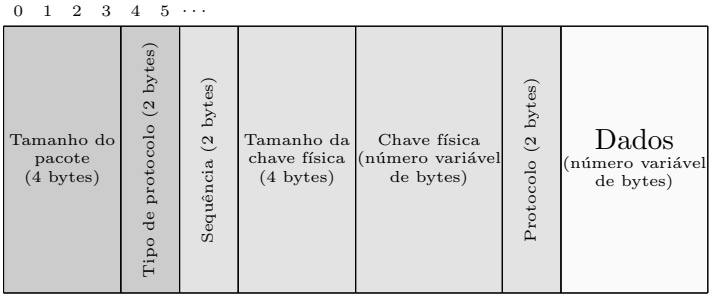
comunicação com um **Ponto de Acesso**. Em seguida, esse dispositivo deve transmitir à plataforma um pacote destinado à solicitação de identificação, que é respondido pelo **Roteador** com um pacote de atribuição de identificador. A partir do momento em que um identificador é atribuído, esse é utilizado como forma de endereçar o dispositivo durante o transporte de dados e, ao final da sessão, o **Ponto-Fim** deve transmitir à plataforma um novo pacote, destinado à liberação do identificador adquirido. Essa estratégia de gerenciamento de identificadores assemelha-se à aquisição de endereços **IP!** dinâmicos realizada através do protocolo **DHCP!** (**DHCP!**) nas redes locais (TANENBAUM, 2011), e é desempenhada no sistema pelo **PGI!** (**PGI!**). Além de possibilitar o gerenciamento (obtenção e liberação) de identificadores, o protocolo **PGI!** oferece suporte à enumeração dos identificadores dos dispositivos registrados na plataforma.

Por ser utilizado para aquisição de um identificador e, portanto, antes que esse esteja disponível, é necessário que o **PGI!** faça uso de uma estratégia de roteamento de pacotes que independa dessa informação. A abordagem utilizada foi a criação de uma chave física, ou seja, uma sequência de caracteres que deve ser gerada pelo **Ponto-Fim** e deve ser única na rede sobre a qual a plataforma é executada. Essa chave é utilizada pelo **Ponto de Acesso** a fim de identificar um dispositivo conectado a ele enquanto esse não possui um identificador. A chave física é formada pela concatenação do protocolo de comunicação utilizado com a identificação do dispositivo nesse mesmo protocolo. Por exemplo, a chave "TCPIP_192.168.0.1:1234" seria utilizada pelo dispositivo conectado através do protocolo **TCP/IP!** de endereço **IP!** 192.168.0.1 e porta **TCP!** 1234.

Todos os pacotes do **PGI!** seguem o formato básico apresentado na Tabela 4.2. Esse formato convenciona que, no protocolo **PGI!**, entre outras coisas, deve ser transmitido um cabeçalho que permita a identificação de um pacote enviado em resposta a outro (através do campo de sequência) e o roteamento sem a utilização de um identificador (através da chave física), além de possuir um campo de identificação do protocolo, garantindo a expansibilidade da gama de protocolos. Nas próximas seções,

serão detalhadas as funções do **PGI!** e os pacotes utilizados para a execução de cada uma delas no sistema.

Tabela 4.2: PGI - Pacote básico



Campo	Descrição
Tamanho do pacote	Idem Tabela 4.1.
Tipo de protocolo	Idem Tabela 4.1. Valor 0x0001.
Sequência	Sequência do pacote utilizada no controle de resposta. Inicia em 0x0000 e deve ser incrementada a cada novo pacote criado.
Tamanho da chave física	Tamanho da chave física.
Chave física	Chave que identifica o dispositivo ao qual o pacote se refere.
Protocolo	Protocolo ao qual o pacote pertence. Valor 0x0001.
Dados	Dados transportados, especificados por cada pacote.

4.1.1 Aquisição de Identidade

A *Aquisição de Identidade* é o processo através do qual um **Ponto-Fim** solicita ao **Roteador** a atribuição de um identificador, que o representará de forma única durante sua conexão à plataforma. A Figura 4.1 demonstra o envio de um *Pacote de Aquisição de Identidade* por um **Ponto-Fim** e a resposta do **Roteador** com um *Pacote de Atribuição de Identidade* que atribui o identificador 4 ao **Ponto-Fim** requisitante. As próximas seções apresentam a estrutura dos pacotes utilizados para execução dessa tarefa.

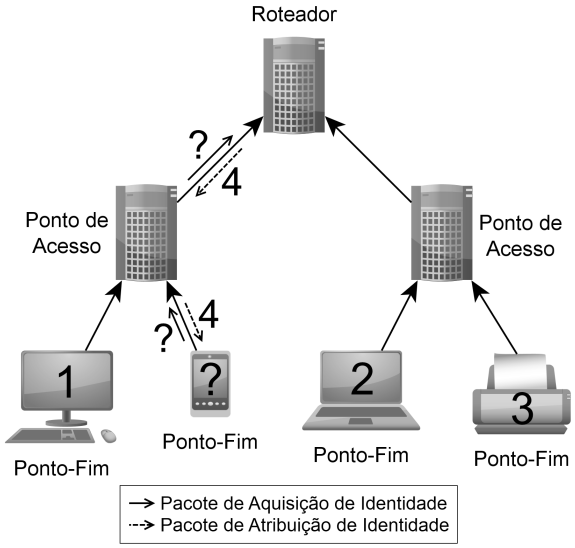
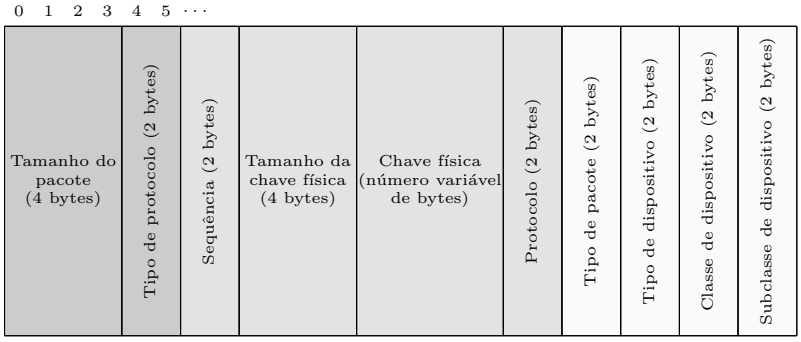


Figura 4.1: Aquisição de Identidade

4.1.1.1 Pacote de Aquisição de Identidade

A Tabela 4.3 descreve o formato do *Pacote de Aquisição de Identidade*, que deve ser enviado pelo **Ponto-Fim** a fim de solicitar a geração e atribuição de um identificador. Para isso, esse pacote apresenta campos que fornecem informações de classificação do dispositivo em tipo, classe e subclasse, permitindo a filtragem de dispositivos durante buscas. Os campos de tipo, classe e subclasse do dispositivo são convencionados para cada aplicação da plataforma, podendo ser utilizados, por exemplo, para diferenciar os dispositivos do sistema dos dispositivos de usuário ou dispositivos móveis de dispositivos estacionários. Os valores convencionados para esses campos devem ser diferentes de **0x0000**, porém não existem valores pré-definidos e, na ausência de uma convenção, o valor padrão **0x0001** deverá ser utilizado. Uma vez que o **Roteador** recebe um *Pacote de Aquisição de Identidade*, esse gera um identificador e responde com um *Pacote de Atribuição de Identidade* ao **Ponto-Fim** remetente.

Tabela 4.3: PGI - Pacote de Aquisição de Identidade

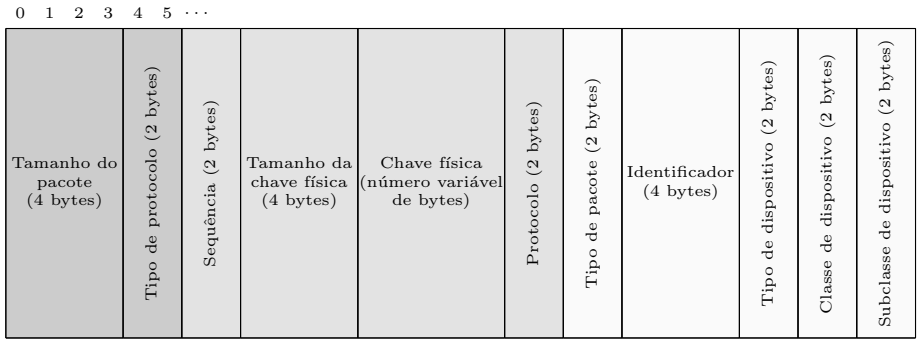


Campo	Descrição
Tamanho do pacote	Idem Tabela 4.1.
Tipo de protocolo	Idem Tabela 4.1. Valor 0x0001.
Sequência	Idem Tabela 4.2.
Tamanho da chave física	Idem Tabela 4.2.
Chave física	Idem Tabela 4.2.
Protocolo	Idem Tabela 4.2. Valor 0x0001.
Tipo de pacote	Identifica o tipo de pacote. Valor 0x0001.
Tipo de dispositivo	Identifica o tipo de dispositivo. Valor padrão 0x0001.
Classe de dispositivo	Identifica a classe de dispositivo. Valor padrão 0x0001.
Subclasse de dispositivo	Identifica a subclasse de dispositivo. Valor padrão 0x0001.

4.1.1.2 Pacote de Atribuição de Identidade

A Tabela 4.4 descreve o formato do *Pacote de Atribuição de Identidade*, que deve ser enviado pelo **Roteador** em resposta a um *Pacote de Aquisição de Identidade*. O *Pacote de Atribuição de Identidade* possui a mesma sequência do *Pacote de Aquisição de Identidade* recebido e carrega, ainda, o identificador que deve ser atribuído ao **Ponto-Fim** e uma réplica do tipo, da classe e da subclasse do dispositivo.

Tabela 4.4: PGI - Pacote de Atribuição de Identidade



Campo	Descrição
Tamanho do pacote	Idem Tabela 4.1.
Tipo de protocolo	Idem Tabela 4.1. Valor 0x0001.
Sequência	Idem Tabela 4.2.
Tamanho da chave física	Idem Tabela 4.2.
Chave física	Idem Tabela 4.2.
Protocolo	Idem Tabela 4.2. Valor 0x0001.
Tipo de pacote	Identifica o tipo de pacote. Valor 0x0002.
Identificador	Identificador atribuído.
Tipo de dispositivo	Tipo de dispositivo atribuído.
Classe de dispositivo	Classe de dispositivo atribuída.
Subclasse de dispositivo	Subclasse de dispositivo atribuída.

4.1.2 Busca de Identidade

A *Busca de Identidade* ocorre quando um **Ponto-Fim** solicita ao **Roteador** a enumeração dos identificadores dos **Pontos-Fim** que atendem a determinado critério. A Figura 4.2 demonstra o envio de um *Pacote de Busca de Identidade* por um **Ponto-Fim** para recuperação dos identificadores dos dispositivos de classe B, assim como o envio pelo **Roteador** de um *Pacote de Resultado de Busca de Identidade* que carrega o identificador 4 como resposta à requisição. As seções a seguir apresentam a estrutura dos pacotes utilizados nesse processo.

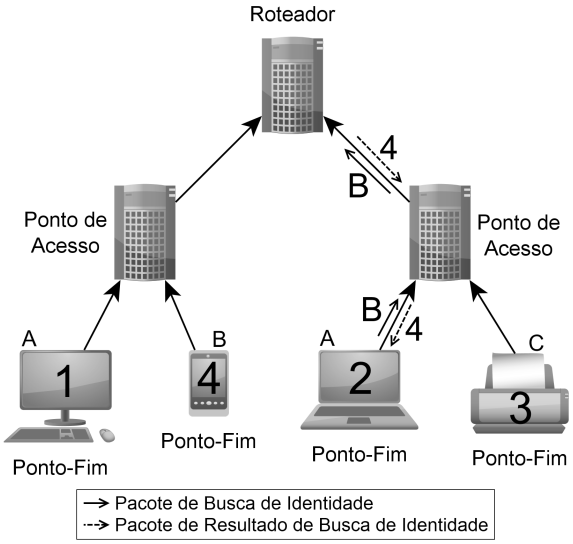
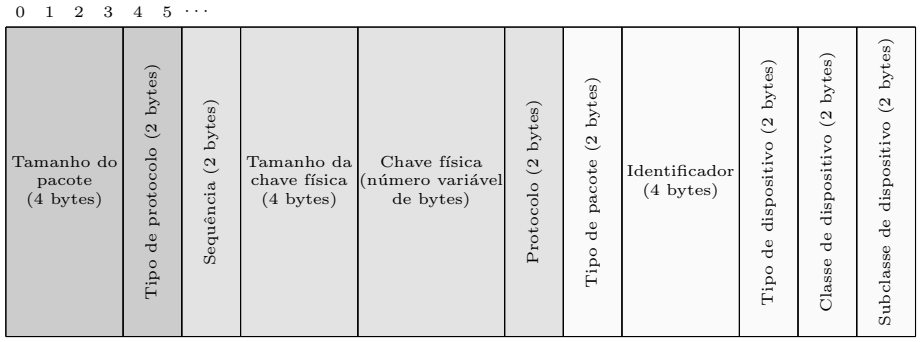


Figura 4.2: Busca de Identidade

4.1.2.1 Pacote de Busca de Identidade

A Tabela 4.5 descreve o formato do *Pacote de Busca de Identidade*, que deve ser enviado por **Pontos-Fim** que necessitam descobrir outros dispositivos na plataforma. Esse pacote contém o identificador do dispositivo requisitante (que deve ter sido previamente atribuído) e os campos de filtragem de tipo, classe e subclasse de dispositivos, que são utilizados para selecionar os identificadores de **Pontos-Fim** a serem incluídos no *Pacote de Resultado de Busca de Identidade*. Caso um dos campos de filtragem deva ser ignorado, esse deve possuir valor **0x0000**. Uma vez que o **Roteador** receber um *Pacote de Busca de Identidade*, deverá consultar sua estrutura interna de armazenamento do estado da plataforma, obtendo os identificadores dos dispositivos que atendem aos critérios de busca, e responder com um *Pacote de Resultado de Busca de Identidade*.

Tabela 4.5: PGI - Pacote de Busca de Identidade

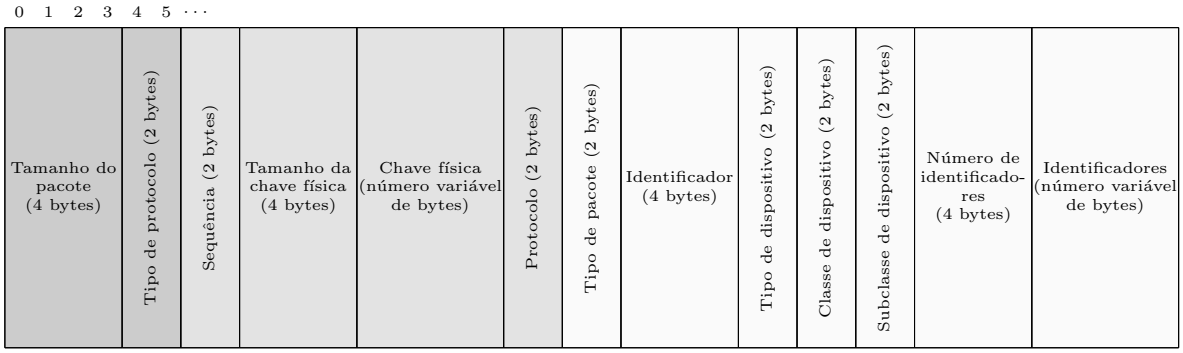


Campo	Descrição
Tamanho do pacote	Idem Tabela 4.1.
Tipo de protocolo	Idem Tabela 4.1. Valor 0x0001.
Sequência	Idem Tabela 4.2.
Tamanho da chave física	Idem Tabela 4.2.
Chave física	Idem Tabela 4.2.
Protocolo	Idem Tabela 4.2. Valor 0x0001.
Tipo de pacote	Identifica o tipo de pacote. Valor 0x0003.
Identificador	Identificador do remetente.
Tipo de dispositivo	Tipo de dispositivo buscado. Utilizar valor 0x0000 para ignorar critério.
Classe de dispositivo	Classe de dispositivo buscada. Utilizar valor 0x0000 para ignorar critério.
Subclasse de dispositivo	Subclasse de dispositivo buscada. Utilizar valor 0x0000 para ignorar critério.

4.1.2.2 Pacote de Resultado de Busca de Identidade

A Tabela 4.6 descreve o formato do *Pacote de Resultado de Busca de Identidade*, que deve ser enviado pelo **Roteador** em resposta a um *Pacote de Busca de Identidade*. O *Pacote de Resultado de Busca de Identidade* possui a mesma sequência do *Pacote de Busca de Identidade*, porém esse carrega, além do identificador do **Ponto-Fim** requisitante e de uma réplica dos filtros utilizados, os identificadores dos dispositivos presentes na plataforma que atendem aos critérios estabelecidos. Um *Pacote de Resultado de Busca de Identidade* não inclui o identificador do **Ponto-Fim** remetente do *Pacote de Busca de Identidade*.

Tabela 4.6: PGI - Pacote de Resultado de Busca de Identidade



Campo	Descrição
Tamanho do pacote	Idem Tabela 4.1.
Tipo de protocolo	Idem Tabela 4.1. Valor 0x0001.
Sequência	Idem Tabela 4.2.
Tamanho da chave física	Idem Tabela 4.2.
Chave física	Idem Tabela 4.2.
Protocolo	Idem Tabela 4.2. Valor 0x0001.
Tipo de pacote	Identifica o tipo de pacote. Valor 0x0004.
Identificador	Identificador do remetente.
Tipo de dispositivo	Tipo de dispositivo buscado.
Classe de dispositivo	Classe de dispositivo buscada.
Subclasse de dispositivo	Subclasse de dispositivo buscada.
Número de identificadores	Número de identificadores encontrados.
Identificadores	Contém os identificadores dos dispositivos encontrados, sendo que cada um ocupa 4 bytes.

4.1.3 Liberação de Identidade

O processo de *Liberação de Identidade* ocorre quando é notificada na plataforma a saída de um **Ponto-Fim**, tornando inválidas quaisquer transações pendentes com o dispositivo em questão. A Figura 4.3 demonstra o envio de um *Pacote de Liberação de Identidade* por um **Ponto-Fim** para notificar sua saída da plataforma, além de ilustrar a propagação dessa notificação a todos os dispositivos que possuem conexões ativas com o identificador liberado. Detalhes desse processo são apresentados na seção a seguir.

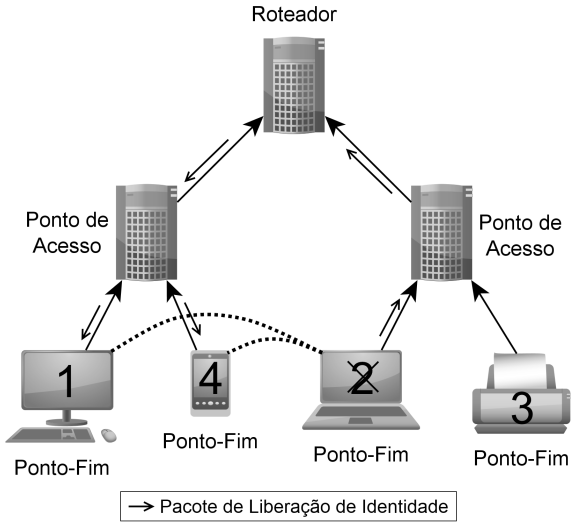


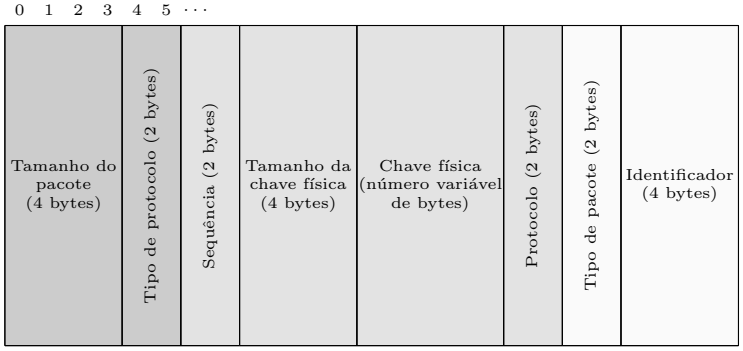
Figura 4.3: Liberação de Identidade

4.1.3.1 Pacote de Liberação de Identidade

A Tabela 4.7 descreve o formato do *Pacote de Liberação de Identidade*, que deve ser enviado na plataforma por **Pontos-Fim**, **Pontos de Acesso** e pelo **Roteador** a fim de notificar os dispositivos sobre a liberação de um identificador (saída de um **Ponto-Fim** da plataforma). Isso pode ocorrer por diferentes motivos: a desconexão voluntária do dispositivo (quando o próprio dispositivo deve enviar o *Pacote de Liberação de Identidade*), a interrupção da conexão entre um dispositivo e o **Ponto de Acesso** (caso no qual o **Ponto de Acesso** deve enviar o *Pacote de Liberação de Identidade*) e, ainda, a interrupção da conexão entre um **Ponto de Acesso** e o **Roteador** (quando o **Roteador** deve enviar o *Pacote de Liberação de Identidade*).

O **Roteador** deve propagar o *Pacote de Liberação de Identidade* a todos os **Pontos de Acesso** a ele conectados que, por sua vez, devem encaminhá-lo a todos os **Pontos-Fim** acessíveis, de forma a notificar todos os dispositivos da plataforma sobre a liberação do identificador. Ao receber um *Pacote de Liberação de Identidade*, o **Roteador**, **Ponto de Acesso** ou **Ponto-Fim** deve imediatamente considerar abortada qualquer atividade envolvendo o dispositivo em questão. Essa estratégia permite uma notificação robusta da saída de dispositivos da plataforma sem exigir que o **Roteador** mantenha registros de todas as conexões ativas no sistema.

Tabela 4.7: PGI - Pacote de Liberação de Identidade



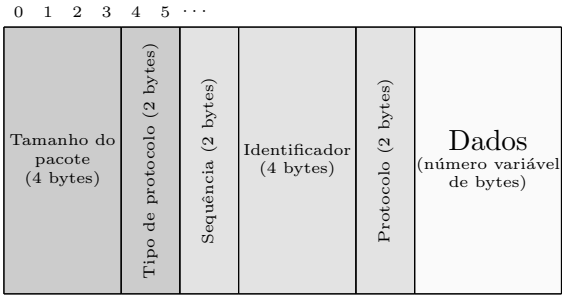
Campo	Descrição
Tamanho do pacote	Idem Tabela 4.1.
Tipo de protocolo	Idem Tabela 4.1. Valor 0x0001.
Sequência	Idem Tabela 4.2.
Tamanho da chave física	Idem Tabela 4.2.
Chave física	Idem Tabela 4.2.
Protocolo	Idem Tabela 4.2. Valor 0x0001.
Tipo de pacote	Identifica o tipo de pacote. Valor 0x0005.
Identificador	Identificador liberado.

4.2 PPS - Protocolo de Publicação de Serviços

O **PPS!** (**PPS!**) é responsável por permitir que **Pontos-Fim** publiquem e/ou consultem serviços disponibilizados por outros dispositivos. Esse protocolo permite, por exemplo, que um **Ponto-Fim** enumere todos os dispositivos que fornecem um serviço de **GPS!** (**GPS!**), a partir do qual poderá consultar sua posição geográfica aproximada sem dispor do equipamento necessário.

Para utilização do protocolo **PPS!**, o dispositivo já deve possuir um identificador, atribuído através do **PGI!**. A Tabela 4.8 descreve o formato básico dos pacotes do **PPS!**, que possuem campos para a identificação de pacotes enviados em resposta a outros (através do campo de sequência) e o identificador do **Ponto-Fim** solicitante. As seções seguintes detalharão as funções do **PPS!** e os pacotes utilizados para a execução de cada uma delas no sistema.

Tabela 4.8: PPS - Pacote básico



Campo	Descrição
Tamanho do pacote	Idem Tabela 4.1.
Tipo de protocolo	Idem Tabela 4.1. Valor 0x0002.
Sequência	Sequência do pacote utilizada no controle de resposta. Inicia em 0x0000 e deve ser incrementada a cada novo pacote criado.
Identificador	Identificador do dispositivo.
Protocolo	Protocolo ao qual o pacote pertence, destinado à expansibilidade da gama de protocolos. Valor 0x0001.
Dados	Dados transportados, especificados por cada pacote.

4.2.1 Publicação de Serviço

Chama-se *Publicação de Serviço* a requisição enviada pelo **Ponto-Fim** ao **Roteador** solicitando a associação de seu identificador a um serviço que deseja oferecer aos demais dispositivos conectados à plataforma, ou a remoção dessa associação. A Figura 4.4 demonstra o envio de dois *Pacotes de Publicação de Serviço* por **Pontos-Fim**, um deles efetuando a publicação e outro efetuando a remoção de um serviço. É ilustrado também o envio de dois *Pacotes de Publicação de Serviço* pelo **Roteador** a fim de notificar o sucesso de ambas as operações. A seção a seguir apresenta a estrutura do pacote que permite essa interação.

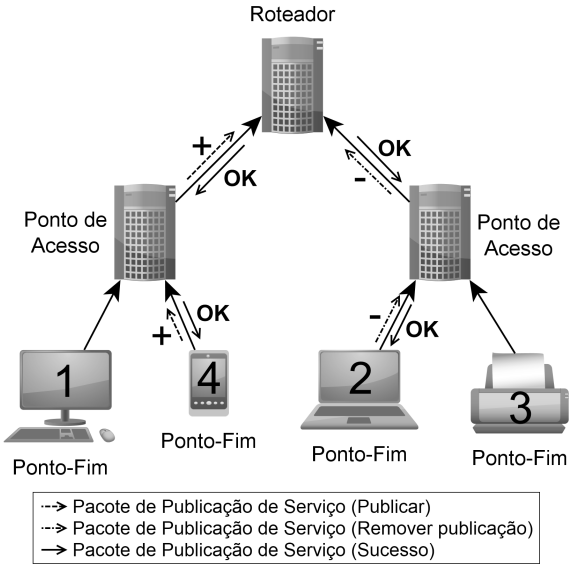
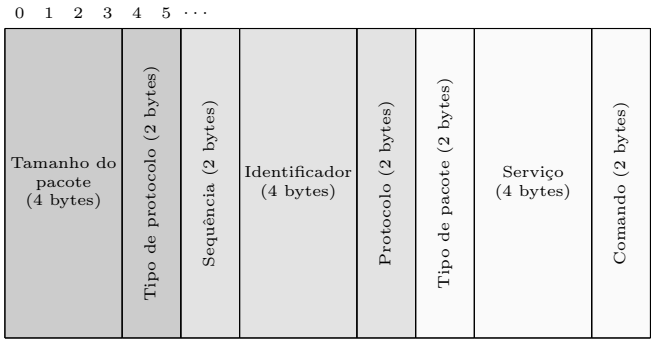


Figura 4.4: Publicação de Serviço

4.2.1.1 Pacote de Publicação de Serviço

A Tabela 4.9 descreve o formato do *Pacote de Publicação de Serviço*, que deve ser enviado por **Pontos-Fim** que desejam publicar ou remover a associação de seu identificador a um determinado serviço. Os campos do *Pacote de Publicação de Serviço* indicam o comando (publicação ou remoção) que deve ser efetuado e o identificador do serviço sobre o qual a operação deverá ser realizada. Ao receber um *Pacote de Publicação de Serviço* cujo comando indica que um serviço deve ser publicado ou removido, o **Roteador** aplica essa operação à sua estrutura interna de controle, respondendo com um *Pacote de Publicação de Serviço* de mesma sequência cujo campo de comando informa se ela foi ou não executada.

Tabela 4.9: PPS - Pacote de Publicação de Serviço



Campo	Descrição
Tamanho do pacote	Idem Tabela 4.1.
Tipo de protocolo	Idem Tabela 4.1. Valor 0x0001.
Sequência	Idem Tabela 4.8.
Identificador	Idem Tabela 4.8.
Protocolo	Idem Tabela 4.8. Valor 0x0001.
Tipo de pacote	Identifica o tipo de pacote. Valor 0x0001.
Serviço	Identifica o serviço.
Comando	Identifica o comando. Os valores possíveis são: **0x0001** Publicar: utilizado para solicitar a publicação de um serviço; **0x00FE** Publicação realizada: utilizado em resposta a uma solicitação de publicação; **0x00FD** Publicação não realizada: utilizado em resposta a uma solicitação de publicação; **0x0002** Remover publicação: utilizado para solicitar a remoção da publicação de um serviço; **0x00EE** Publicação removida: utilizado em resposta a uma solicitação de remoção de publicação; **0x00ED** Remoção da publicação não realizada: utilizado em resposta a uma solicitação de remoção de publicação.

4.2.2 Busca de Serviço

A *Busca de Serviço* ocorre quando um **Ponto-Fim** solicita ao **Roteador** a enumeração dos identificadores dos **Pontos-Fim** que publicaram um determinado serviço. A Figura 4.5 demonstra o envio de um *Pacote de Busca de Serviço* por um **Ponto-Fim** para recuperação dos identificadores dos dispositivos que possuem o serviço C e a resposta do **Roteador** com um *Pacote de Resultado de Busca de Serviço* que carrega o identificador 3 como resposta à requisição. As seções a seguir

apresentam a estrutura dos pacotes utilizados nessa tarefa.

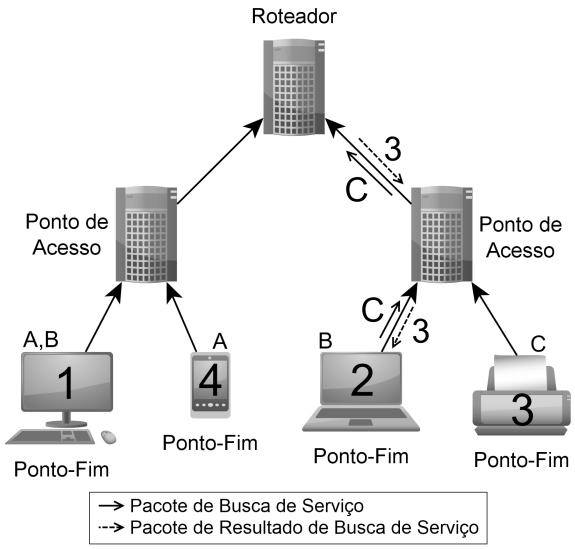
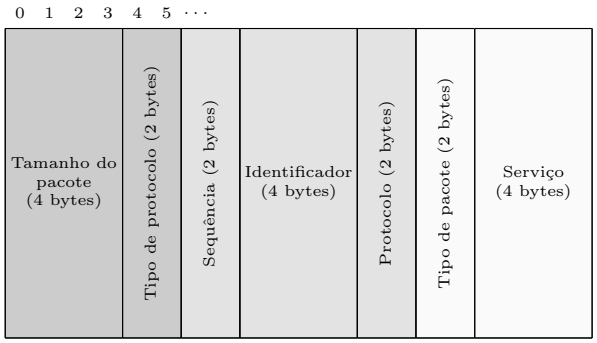


Figura 4.5: Busca de Serviço

4.2.2.1 Pacote de Busca de Serviço

A Tabela 4.10 descreve o formato do *Pacote de Busca de Serviço*, que deve ser enviado por **Pontos-Fim** que necessitam descobrir os dispositivos que publicaram um determinado serviço na plataforma. Ao receber um *Pacote de Busca de Serviço*, o **Roteador** consulta sua estrutura interna de controle, respondendo com um *Pacote de Resultado de Busca de Serviço* que contém os identificadores dos **Pontos-Fim** que publicaram o serviço solicitado.

Tabela 4.10: PPS - Pacote de Busca de Serviço

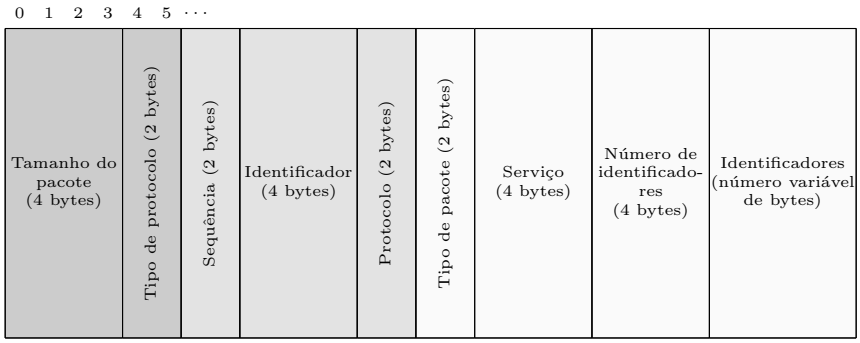


Campo	Descrição
Tamanho do pacote	Idem Tabela 4.1.
Tipo de protocolo	Idem Tabela 4.1. Valor 0x0001.
Sequência	Idem Tabela 4.8.
Identificador	Idem Tabela 4.8.
Protocolo	Idem Tabela 4.8. Valor 0x0001.
Tipo de pacote	Identifica o tipo de pacote. Valor 0x0002.
Serviço	Identifica o serviço a ser buscado.

4.2.2.2 Pacote de Resultado de Busca de Serviço

A Tabela 4.11 descreve o formato do *Pacote de Resultado de Busca de Serviço*, que deve ser enviado pelo **Roteador** em resposta a um *Pacote de Busca de Serviço*. Esse pacote possui a mesma sequência do *Pacote de Busca de Serviço* e carrega, além disso, o identificador do serviço buscado e os identificadores dos **Pontos-Fim** que o publicaram. Destaca-se que o *Pacote de Resultado de Busca de Serviço* não inclui o identificador do dispositivo que enviou o *Pacote de Busca de Serviço*.

Tabela 4.11: PPS - Pacote de Resultado de Busca de Serviço



Campo	Descrição
Tamanho do pacote	Idem Tabela 4.1.
Tipo de protocolo	Idem Tabela 4.1. Valor 0x0001.
Sequência	Idem Tabela 4.8.
Identificador	Idem Tabela 4.8.
Protocolo	Idem Tabela 4.8. Valor 0x0001.
Tipo de pacote	Identifica o tipo de pacote. Valor 0x0003.
Serviço	Identifica o serviço buscado.
Número de identificadores	Número de identificadores encontrados.
Identificadores	Contém os identificadores dos dispositivos encontrados, sendo que cada um ocupa 4 bytes.

4.3 PTD - Protocolo de Transporte de Dados

Estando registrado na plataforma e tendo localizado um dispositivo que possui um serviço ao qual deseja acesso, um **Ponto-Fim** é capaz de estabelecer uma conexão virtual a esse serviço e de trocar pacotes de dados através dela, usufruindo dessa forma dos recursos publicados por esse dispositivo. Na plataforma, o protocolo responsável pelo controle dessas conexões virtuais e do tráfego de pacotes de dados através delas é o **PTD!** (**PTD!**).

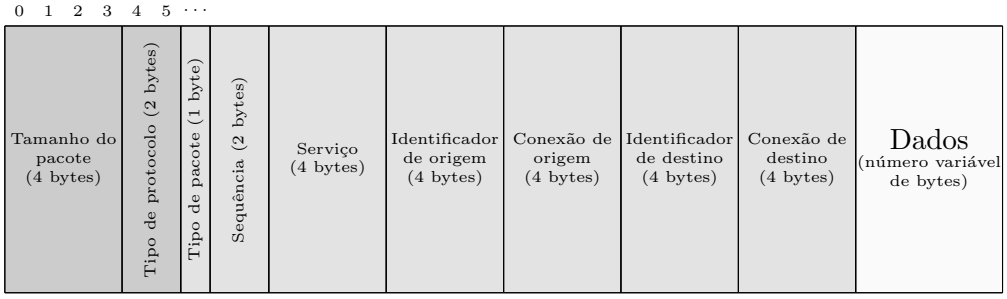
Os pacotes do protocolo **PTD!** estão associados a um serviço ao qual está sendo feito acesso, e sua transmissão ocorre entre dois **Pontos-Fim**: aquele que publicou o serviço e aquele que solicitou acesso a ele. Por essa razão, esses pacotes possuem campos que indicam qual o serviço acessado e quais os identificadores dos **Pontos-Fim** de origem e de destino entre os quais devem ser roteados.

Devido ao fato de que cada serviço publicado na plataforma é capaz de gerenciar diversas conexões virtuais, e que cada

Ponto-Fim é capaz de iniciar múltiplas conexões virtuais a esses serviços, não é suficiente para o roteamento a identificação dos dispositivos de origem e de destino, tornando necessário que a cada conexão solicitada por um **Ponto-Fim** ou recebida por um serviço seja atribuído um identificador. Esses identificadores são gerados pelos **Pontos-Fim** e transmitidos nos pacotes do protocolo **PTD!** associados aos identificadores dos dispositivos.

Todos os pacotes do **PTD!** devem ser utilizados entre dois **Pontos-Fim** cujo identificador já foi atribuído através do **PGI!**, e seguem o formato básico apresentado na Tabela 4.12. As seções a seguir detalham as funções do **PTD!** e os pacotes utilizados para a execução de cada uma delas no sistema.

Tabela 4.12: PTD - Pacote básico

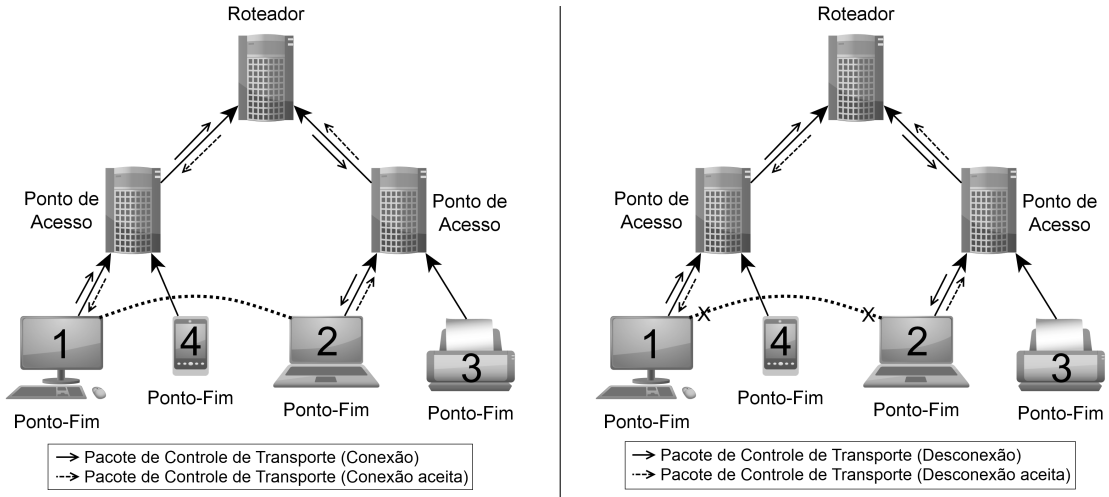


Campo	Descrição
Tamanho do pacote	Idem Tabela 4.1.
Tipo de protocolo	Idem Tabela 4.1. Valor 0x0003.
Tipo de pacote	Especificado por cada pacote.
Sequência	Sequência do pacote utilizada no controle de resposta. Inicia em 0x0000 e deve ser incrementada a cada novo pacote criado.
Serviço	Identifica o serviço em utilização.
Identificador de origem	Identificador do dispositivo que enviou o pacote.
Conexão de origem	Identificador da conexão que enviou o pacote.
Identificador de destino	Identificador do dispositivo ao qual o pacote deve ser entregue.
Conexão de destino	Identificador da conexão à qual o pacote deve ser entregue.
Dados	Dados transportados, especificados por cada pacote.

4.3.1 Controle de Conexão

O *Controle de Conexão* ocorre quando um **Ponto-Fim** requer conexão a um serviço ou responde a uma requisição recebida.

Esse mesmo processo engloba a finalização de conexões e a notificação de resultado para todas as operações citadas. A Figura 4.6 demonstra a utilização do *Pacote de Controle de Transporte* para o estabelecimento e a finalização de uma conexão virtual entre dois **Pontos-Fim**, representada pela linha tracejada. É ilustrada também a utilização do mesmo pacote como notificação de sucesso para ambas as operações. A seção a seguir apresenta a estrutura do pacote utilizado nessas tarefas.



(a) Estabelecimento de conexão

(b) Finalização de conexão

Figura 4.6: Controle de Conexão

4.3.1.1 Pacote de Controle de Transporte

A Tabela 4.13 descreve o formato do *Pacote de Controle de Transporte*, que deve ser enviado por **Pontos-Fim** a fim de executar operações de controle sobre uma determinada conexão virtual e notificar a contraparte sobre o resultado da execução de operações por ela solicitadas. As operações suportadas pelo *Pacote de Controle de Transporte* são o estabelecimento de conexões, o término de conexões e a notificação sobre a entrega de *Pacotes de Transporte de Dados* (descritos na próxima seção).

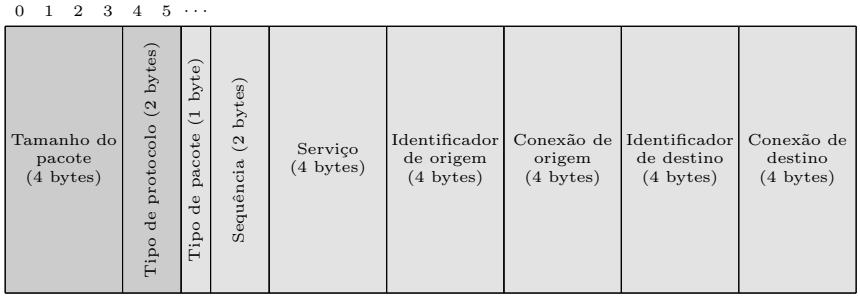
Um *Pacote de Solicitação de Conexão* deve ser enviado pelos **Pontos-Fim** que desejam estabelecer uma conexão a um serviço publicado por um dispositivo na plataforma. Para isso, o **Ponto-Fim** solicitante gera um identificador local para a conexão, que é enviado ao dispositivo de destino nesse pacote. Ao receber um *Pacote de Solicitação de Conexão*, um **Ponto-Fim** delega seu tratamento ao serviço ao qual essa foi

enviada; nesse momento, o segundo identificador de conexão é gerado pelo serviço, e é enviado ao dispositivo que solicitou a conexão um *Pacote de Notificação de Conexão Aceita*, de forma que ambos os **Pontos-Fim** passam a conhecer o identificador de conexão de sua contraparte. Caso a solicitação de conexão não possa ser tratada, um *Pacote de Notificação de Conexão Recusada* é enviado.

Um *Pacote de Notificação de Entrega* deve ser enviado pelos **Pontos-Fim** sempre que esses receberem um pacote de dados com notificação, ou seja, um *Pacote de Transporte de Dados* que exige o envio de uma notificação de entrega. Caso um pacote de dados com notificação não possa ser entregue, um *Pacote de Notificação de Falha de Entrega* é enviado ao remetente. Não é necessário o envio de resposta a pacotes de dados sem notificação de entrega.

Uma vez finalizada a utilização de uma conexão, o **Ponto-Fim** envia um *Pacote de Solicitação de Desconexão* à contraparte. Ao receber esse tipo de pacote, o **Ponto-Fim** responde com um *Pacote de Notificação de Desconexão Aceita*.

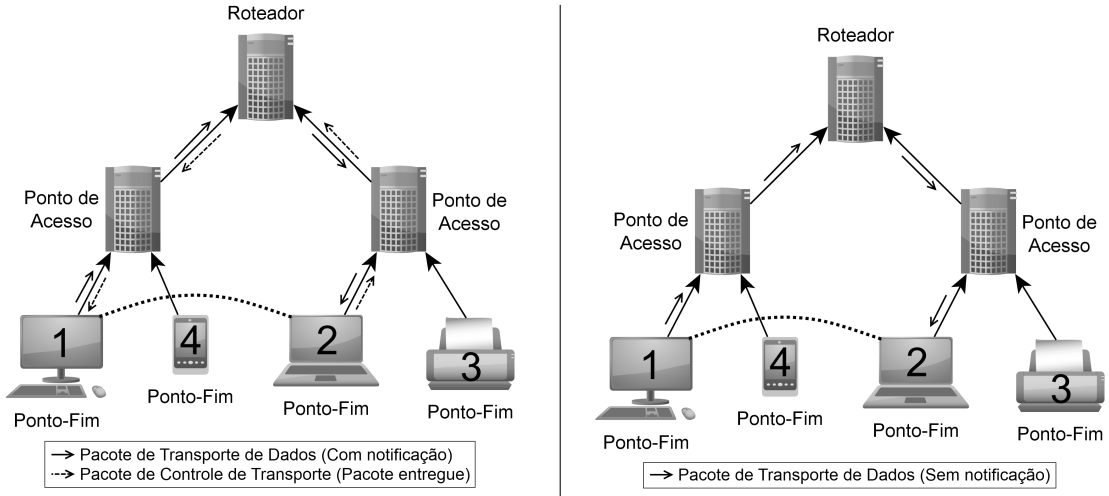
Tabela 4.13: PTD - Pacote de Controle de Transporte



Campo	Descrição
Tamanho do pacote	Idem Tabela 4.1.
Tipo de protocolo	Idem Tabela 4.1. Valor 0x0003.
Tipo de pacote	Tipo de pacote. Os valores possíveis são: **0x00** Solicitação de conexão; **0x01** Notificação de conexão aceita; **0x02** Notificação de conexão recusada; **0x82** Notificação de *Pacote de Transporte de Dados* entregue; **0x83** Notificação de *Pacote de Transporte de Dados* não entregue; **0xFE** Solicitação de desconexão; **0xFF** Notificação de desconexão aceita.
Sequência	Idem Tabela 4.12.
Serviço	Idem Tabela 4.12.
Identificador de origem	Idem Tabela 4.12.
Conexão de origem	Idem Tabela 4.12.
Identificador de destino	Idem Tabela 4.12.
Conexão de destino	Idem Tabela 4.12.

4.3.2 Transporte de Dados

O *Transporte de Dados* é o processo através do qual um **Ponto-Fim** requer o envio de um pacote de dados (sequência de *bytes*) para outro **Ponto-Fim** virtualmente conectado a ele, exigindo ou não uma confirmação de entrega. A Figura 4.7 demonstra a utilização do *Pacote de Transporte de Dados* para o envio de dados com e sem notificação de entrega, respectivamente, e a utilização do *Pacote de Controle de Transporte* como notificação de sucesso de entrega para pacotes com notificação. Na seção a seguir será detalhada a estrutura do *Pacote de Transporte de Dados*.



(a) Com notificação

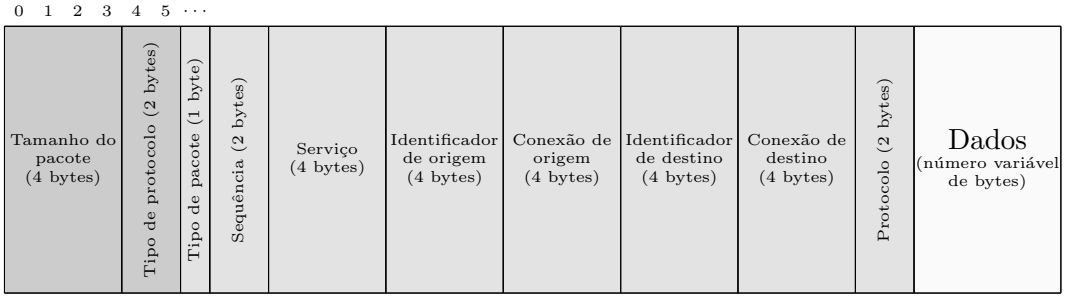
(b) Sem notificação

Figura 4.7: Transporte de Dados

4.3.2.1 Pacote de Transporte de Dados

A Tabela 4.14 descreve o formato do *Pacote de Transporte de Dados*, que é utilizado para transmissão de mensagens através de uma conexão estabelecida entre dois **Pontos-Fim**. Um *Pacote de Transporte de Dados* pode ou não exigir confirmação de entrega; caso essa confirmação seja exigida, o **Ponto-Fim** de destino, ao receber o pacote, responde com um *Pacote de Notificação de Entrega*. Os campos do *Pacote de Transporte de Dados*, além das informações de roteamento, carregam o identificador do protocolo que define o formato dos dados transmitidos, a fim de garantir a expansibilidade dos protocolos da plataforma, e os dados propriamente ditos.

Tabela 4.14: PTD - Pacote de Transporte de Dados



Campo	Descrição
Tamanho do pacote	Idem Tabela 4.1.
Tipo de protocolo	Idem Tabela 4.1. Valor 0x0003.
Tipo de pacote	Tipo de pacote. Os valores possíveis são: **0x80** *Pacote de Transporte de Dados* com notificação; **0x81** *Pacote de Transporte de Dados* sem notificação.
Sequência	Idem Tabela 4.12.
Serviço	Idem Tabela 4.12.
Identificador de origem	Idem Tabela 4.12.
Conexão de origem	Idem Tabela 4.12.
Identificador de destino	Idem Tabela 4.12.
Conexão de destino	Idem Tabela 4.12.
Protocolo	Protocolo ao qual o pacote pertence, destinado à expansibilidade da gama de protocolos. Valor 0x0001.
Dados	Dados transportados pelo pacote. Especificado no próximo capítulo.

Os dados transportados pelo *Pacote de Transporte de Dados* devem seguir um formato que permita a troca de mensagens complexas de forma simplificada, evitando que o usuário precise tratar a codificação e a interpretação de dados brutos (sequências de *bytes*) para que possa trocar informações na plataforma. No próximo capítulo será apresentado o formato utilizado no campo de dados desses pacotes.

5 TRANSPORTE DE MENSAGENS

Os protocolos de aplicação apresentados no Capítulo 4 tratam o registro de dispositivos, a publicação e busca de serviços, e o transporte de pacotes de dados através de conexões virtuais, de forma a possibilitar a transmissão de sequências de *bytes* entre **Pontos-Fim**. A fim de facilitar o intercâmbio de dados na plataforma, as mensagens transmitidas devem seguir um formato que permita o armazenamento de informações complexas aninhadas, ou seja, que delimite múltiplos campos capazes de armazenar diferentes tipos de informação na mesma mensagem, inclusive estruturas aninhadas que permitam a transmissão de árvores de dados, além de garantir que os tipos transmitidos possam ser interpretados em qualquer linguagem e/ou arquitetura de computador. Formatos como **XML!** e a arquitetura **CORBA!** foram criados para atender à demanda por soluções para a troca de dados que possuem essas características (ARCINIEGAS, 2002; OMG, 2011).

O formato **XML!** consiste em uma linguagem textual de representação de dados demarcada por caracteres especiais através da qual é possível a descrição de árvores de registros que podem conter qualquer tipo de dado, o que acaba por torná-la uma boa opção para arquivos que devem ser editados por humanos. Porém, por utilizar formato texto, toda informação precisa ser convertida nesse formato para que possa ser representada, tornando computacionalmente custoso o acesso aos dados armazenados (ARCINIEGAS, 2002).

A arquitetura **CORBA!**, por sua vez, utiliza protocolos que permitem o transporte de árvores de dados em formato binário, o que torna sua interpretação computacional muito mais veloz

em relação ao formato **XML!**, já que as informações são transmitidas em formato binário, semelhante àquele em que serão utilizadas pelo receptor. Em contrapartida, essa arquitetura utiliza pacotes de dados nos quais os campos são necessariamente transmitidos e interpretados na mesma ordem em que são utilizados na aplicação, de forma que uma alteração na estrutura dos pacotes deve ser imediatamente propagada a todos os clientes, causando a falha do serviço caso essa sincronização não seja realizada (ARCINIEGAS, 2002; OMG, 2011). No formato **XML!** esse problema não ocorre, uma vez que os campos são interpretados pela aplicação de forma independente da ordem em que são transmitidos (ARCINIEGAS, 2002).

Devido às limitações ligadas aos formatos citados, optou-se neste trabalho pelo desenvolvimento de um formato que permita o transporte de dados com flexibilidade semelhante àquela oferecida pelo **XML!**, mas que, de forma semelhante à arquitetura **CORBA!**, utilize formato binário para codificação dos dados, tornando sua interpretação computacionalmente veloz.

Para a formalização das mensagens transmitidas na plataforma, optou-se pela utilização da linguagem **XDR!** (**XDR!**), que é um padrão de descrição e codificação de dados mantido pelo **IETF!** (**IETF!**) (EISLER, 2006). A fim de representar nessa linguagem o formato em questão, foi necessário adicionar à **XDR!** a possibilidade de representação de números inteiros de tamanho menor que 4 *bytes*. Essa possibilidade foi adicionada de forma a permitir o armazenamento dos tipos especiais de dados formalizados no Trecho de Código 5.1, que representam números inteiros e caracteres de 8 e 16 *bits*.

Trecho de Código 5.1: Definição de tipos de dados especiais

```
 1 /* Inteiro de 8 bits com sinal */
 2 typedef byte int(8);
 3 /* Inteiro de 8 bits sem sinal */
 4 typedef unsigned byte unsigned int(8);
 5 /* Inteiro de 16 bits com sinal */
 6 typedef short int(16);
 7 /* Inteiro de 16 bits sem sinal */
 8 typedef unsigned short unsigned int(16);
 9 /* Caractere de 8 bits */
10 typedef character8 unsigned int(8);
```

```
11 /* Caractere de 16 bits */
12 typedef character16 unsigned int(16);
```

O padrão **XDR!** também define que todo campo deve ter tamanho múltiplo de 4 *bytes*, sendo necessário concatenar de 0 a 3 *bytes* a cada campo para que esse alinhamento seja alcançado (EISLER, 2006). No formato definido neste trabalho, esse alinhamento não é realizado. Na próxima seção será formalizado o formato do conteúdo das mensagens transmitidas na plataforma através da linguagem **XDR!**.

5.1 Mensagens

As mensagens transmitidas na plataforma seguem um formato binário que representa um conjunto não ordenado de campos. O Trecho de Código 5.2 apresenta o formato dessas mensagens, que consiste em uma estrutura denominada *Conteúdo* que armazena um vetor de tamanho variável de estruturas denominadas *Campo*, cuja estrutura será apresentada na Seção 5.1.1.

Trecho de Código 5.2: Conteúdo das mensagens

```
1 /* Conteúdo */
2 struct conteudo {
3     campo cmCampo<>;
4 }
```

5.1.1 Conteúdo

Cada *Campo* de um *Conteúdo* possui um *Nome*, um *Tipo de Campo* e um *Valor de Campo*. O *Nome* do *Campo* consiste em um identificador textual de tamanho variável utilizado para acessar um *Campo* do *Conteúdo*. Esse identificador deve ser preenchido de forma que não existam dois *Campos* de mesmo *Nome* em um mesmo *Conteúdo*; o *Tipo de Campo* define a forma como devem ser interpretados os dados carregados no *Valor do Campo*. Com a utilização desse formato, torna-se possível a transmissão de árvores de dados complexas, às quais podem ser adicionados ou removidos *Campos* sem que seja necessária a alteração do protocolo.

O Trecho de Código 5.3 apresenta o formato de cada *Campo* presente no *Conteúdo* das mensagens, que contém um vetor de

caracteres de 1 *byte* para armazenamento do *Nome* do *Campo*, além de uma enumeração que define o *Tipo do Campo* e de uma estrutura que armazena o *Valor do Campo*. As próximas seções apresentarão, respectivamente, a enumeração dos *Tipos de Campo* e a estrutura do *Valor de Campo*.

Trecho de Código 5.3: Campo de um Conteúdo

```
1 /* Campo do conteúdo da mensagem */
2 struct campo {
3     unsigned short shTamanhoNome;
4     character8 stNome[shTamanhoNome];
5     tipo_campo tcTipoCampo;
6     valor_campo vcValorCampo;
7 }
```

5.1.2 Tipo de Campo

O formato utilizado apresenta restrições na troca de tipos de dados cujo suporte pelas linguagens de programação mais comumente utilizadas não existe ou apresenta incompatibilidades geralmente conhecidas. Exemplos das limitações cujo suporte é restrito são objetos serializáveis e caracteres.

A troca de objetos serializáveis, suportada por exemplo pelas linguagens orientadas a objetos *Java* e *C#*, não é viável na plataforma uma vez que o sistema poderá ser acessado por aplicativos escritos em linguagens que não suportam esse tipo de estrutura. A serialização de objetos é geralmente implementada através da conversão dos atributos de objetos em um formato binário ou textual que possa ser posteriormente interpretado, permitindo assim a restauração do objeto pela parte receptora (ORACLE, 2011; MSDN, 2012). Porém, linguagens como *C* não oferecem suporte a objetos e, portanto, não são capazes de tratar esse tipo de estrutura (DEITEL; DEITEL; CAETANO, 2011). Devido ao fato da serialização de objetos não ser suportada por diversas linguagens, a transmissão de objetos serializáveis não é tratada pela plataforma.

Os tipos de dados *char* (caractere) e *string* (cadeia de caracteres) são armazenados de forma diferente por diferentes linguagens. Por exemplo, as linguagens *Java* e *C#* armazenam caracteres no formato *Unicode* utilizando 2 *bytes* (16 *bits*) (ORACLE, 2011; MSDN, 2012); já a linguagem *C* armazena

caracteres no formato **ASCII!**, utilizando apenas 1 *byte* (8 *bits*) (DEITEL; DEITEL; CAETANO, 2011). A plataforma oferece suporte explícito a ambas as formas de armazenamento citadas, cabendo ao usuário utilizar a forma adequada para a aplicação à qual o sistema servirá. O formato **XML!** suporta ambos os tipos, devendo todo o documento ser convertido para um ou outro formato. Já a arquitetura **CORBA!** oferece tratamento especial a esses tipos, cabendo ao usuário definir o mais adequado (ARCINIEGAS, 2002; OMG, 2011).

O Trecho de Código 5.4 descreve formalmente os *Tipos de Campo* que são suportados pelas mensagens da plataforma. É possível verificar-se que são permitidos *Valores de Campo* nulos (*null*); caso a linguagem na qual a mensagem está sendo lida não ofereça suporte nativo a esse valor, um valor padrão deverá ser assumido para o *Campo* nulo (de acordo com seu tipo) e deverá ser possível ao usuário a verificação da validade do *Valor do Campo*, de forma a permitir um tratamento correto. Caberá, portanto, ao usuário a utilização apenas de valores válidos e tratamento de valores nulos.

Trecho de Código 5.4: Tipos de Campo suportados

```
1 /* Enumeração de Tipo de Campo, de tamanho 1 byte */
2 enum tipo_campo {
3     NULL = 0x00,
4     BOOLEAN = 0x01,
5     BOOLEANARRAY = 0x02,
6     BYTE = 0x03,
7     BYTEARRAY = 0x04,
8     CHARACTER = 0x05,
9     CHARACTERARRAY = 0x06,
10    CHARACTER8 = 0x07,
11    CHARACTER8ARRAY = 0x08,
12    SHORT = 0x09,
13    SHORTARRAY = 0x0A,
14    INTEGER = 0x0B,
15    INTEGERARRAY = 0x0C,
16    LONG = 0x0D,
17    LONGARRAY = 0x0E,
18    FLOAT = 0x0F,
19    FLOATARRAY = 0x10,
20    DOUBLE = 0x11,
21    DOUBLEARRAY = 0x12,
22    STRING = 0x13,
23    STRINGARRAY = 0x14,
24    STRING8 = 0x15,
25    STRING8ARRAY = 0x16,
26    CONTEUDO = 0x17,
27    CONTEUDOARRAY = 0x18
28 };
```

5.1.3 Valor de Campo

Os dados presentes no *Valor do Campo* variam de acordo com o *Tipo do Campo*. O Trecho de Código 5.5 apresenta as informações que são armazenadas pelo *Valor do Campo* para cada *Tipo de Campo* suportado. Vale destacar a utilização recursiva da estrutura de *Conteúdo*, que torna possível a transmissão de árvores de dados, e da representação de vetores de todos os tipos de dados suportados, que são representados por estruturas dispostas sequencialmente e precedidas por um número inteiro de 4 *bytes* que indica o número de elementos do vetor. As estruturas utilizadas na representação de vetores serão apresentadas na próxima seção.

Trecho de Código 5.5: Valor de Campo

```
1 /* Valor de Campo */
2 union valor_campo switch (tipo_campo tcTipoCampo) {
3 case NULL:
4     void;
5 case BOOLEAN:
6     bool blValor;
7 case BOOLEANARRAY:
8     bool_arrayentry blValor<>;
9 case BYTE:
10    byte btValor;
11 case BYTEARRAY:
12    byte_arrayentry btValor<>;
13 case CHARACTER:
14    character16 chValor;
15 case CHARACTERARRAY:
16    character16_arrayentry chValor<>;
17 case CHARACTER8:
18    character8 chValor;
19 case CHARACTER8ARRAY:
20    character8_arrayentry chValor<>;
21 case SHORT:
22    short shValor;
23 case SHORTARRAY:
24    short_arrayentry shValor<>;
25 case INTEGER:
26    int inValor;
27 case INTEGERARRAY:
28    integer_arrayentry inValor<>;
29 case LONG:
30    hyper lnValor;
31 case LONGARRAY:
32    long_arrayentry lnValor<>;
33 case FLOAT:
34    float flValor;
```

```
35 case FLOATARRAY:
36     float_arrayentry flValor<>;
37 case DOUBLE:
38     double dbValor;
39 case DOUBLEARRAY:
40     double_arrayentry dbValor<>;
41 case STRING:
42     character16 stValor<>;
43 case STRINGARRAY:
44     string16_arrayentry stValor<>;
45 case STRING8:
46     character8 stValor<>;
47 case STRING8ARRAY:
48     string8_arrayentry stValor<>;
49 case CONTEUDO:
50     conteudo cnValor;
51 case CONTEUDOARRAY:
52     conteudo_arrayentry cnValor<>;
53 };
```

5.1.4 Vetores

A fim de permitir a transmissão de vetores, são utilizadas estruturas no formato apresentado no Trecho de Código 5.6. Essas estruturas contêm um *Tipo de Campo*, utilizado para indicar se a entrada contém um valor válido ou um valor nulo. O tratamento de valores nulos para cada entrada de vetor equivale àquele adotado para *Campos* de valor nulo, ou seja, caberá ao usuário utilizar a *API* de forma a tratar esse tipo de valor de forma correta.

Trecho de Código 5.6: Entradas de vetor

```
1 /* Entradas de vetor */
2 struct bool_arrayentry {
3     tipo_campo tcTipoCampo; /* Deve ser NULL ou BOOLEAN */
4     bool *blValor;
5 }
6 struct byte_arrayentry {
7     tipo_campo tcTipoCampo; /* Deve ser NULL ou BYTE */
8     byte *btValor;
9 }
10 struct character16_arrayentry {
11     tipo_campo tcTipoCampo; /* Deve ser NULL ou CHARACTER */
12     character16 *chValor;
13 }
14 struct character8_arrayentry {
15     tipo_campo tcTipoCampo; /* Deve ser NULL ou CHARACTER8 */
16     character8 *chValor;
17 }
18 struct short_arrayentry {
19     tipo_campo tcTipoCampo; /* Deve ser NULL ou SHORT */
20     short *shValor;
```

```
21 }
22 struct integer_arrayentry {
23     tipo_campo tcTipoCampo; /* Deve ser NULL ou INTEGER */
24     int *inValor;
25 }
26 struct long_arrayentry {
27     tipo_campo tcTipoCampo; /* Deve ser NULL ou LONG */
28     hyper *inValor;
29 }
30 struct float_arrayentry {
31     tipo_campo tcTipoCampo; /* Deve ser NULL ou FLOAT */
32     float *flValor;
33 }
34 struct double_arrayentry {
35     tipo_campo tcTipoCampo; /* Deve ser NULL ou DOUBLE */
36     double *dbValor;
37 }
38 struct string16_arrayentry {
39     tipo_campo tcTipoCampo; /* Deve ser NULL ou STRING */
40     character16 *stValor<>;
41 }
42 struct string8_arrayentry {
43     tipo_campo tcTipoCampo; /* Deve ser NULL ou STRING8 */
44     character8 *stValor<>;
45 }
46 struct conteudo_arrayentry {
47     tipo_campo tcTipoCampo; /* Deve ser NULL ou CONTEUDO */
48     conteudo *cnValor;
49 }
```

Através do formato descrito, torna-se possível ao usuário da plataforma a transmissão de mensagens contendo múltiplos *Campos*, sendo cada um deles identificado por um *Nome* e capaz de armazenar um tipo de dado distinto, simplificando a tarefa de troca de informações entre dispositivos na plataforma.

6 IMPLEMENTAÇÃO

O sistema foi desenvolvido em *Java SE* (*Standard Edition*) utilizando a versão 1.6.0_35 do *JDK* (*Java Development Kit*) mantido pela *ORACLE*. Para maiores informações, recomenda-se a página de apresentação (ORACLE, 2012) e a documentação da versão 6 da plataforma (ORACLE, 2011).

A fim de facilitar a execução dos aplicativos que compõem o sistema, foram gerados arquivos executáveis para o sistema operacional *Microsoft Windows* (MICROSOFT, 2012) utilizando o *software Launch4j* (KOWAL, 2012) e *scripts* para o sistema operacional *Linux* que invocam automaticamente a máquina virtual *Java*, além de um *JAR* (*Java Archive*) que pode ser executado manualmente em qualquer plataforma com suporte a *Java SE*. O núcleo da plataforma e o **Ponto-Fim** para o protocolo **TCP/IP!** (TANENBAUM, 2011), além de programas de validação, foram também desenvolvidos para a plataforma *Android* (GOOGLE, 2012), podendo ser executados em dispositivos como celulares e *tablets* equipados com esse sistema operacional.

Durante o desenvolvimento deste trabalho, buscou-se implementar o sistema de forma genérica e expansível, procurando minimizar o esforço necessário para criar novos protocolos, novos pacotes para os protocolos ou suportar diferentes interfaces de comunicação no futuro. Dessa forma, diversas classes do sistema são representações abstratas de elementos para os quais foi feita uma única implementação concreta, devendo esses ser reconhecidos como pontos de expansibilidade. Um exemplo dessa característica são as classes que representam **Pontos de Acesso** e **Pontos-Fim**, já que

para ambas foram desenvolvidas implementações apenas para o protocolo **TCP/IP!**, mas essas simplificam o tratamento de novas interfaces de comunicação por carregarem a porção genérica (reaproveitável) desses aplicativos.

Quatro projetos integram o *software* desenvolvido neste trabalho, cada um representado por um pacote cujo nome busca indicar sua função na arquitetura. O projeto **Core** (*platform.core*) é a biblioteca do núcleo do sistema, possuindo recursos que são compartilhados pelos projetos **Router** (*platform.router*), **TCPIPAccesspoint** (*platform.accesspoint.tcpip*) e **TCPIPEndpoint** (*platform.endpoint.tcpip*). Esses projetos implementam, respectivamente, o **Roteador**, o **Ponto de Acesso** para o protocolo **TCP/IP!** e o **Ponto-Fim** para o protocolo **TCP/IP!**. A Figura 6.1 apresenta o diagrama **UML!** de pacotes dos quatro projetos que compõem o sistema, indicando também as relações de utilização existentes entre eles.

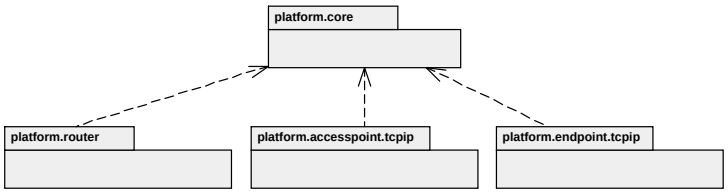


Figura 6.1: Pacotes de topo

As próximas seções descrevem e ilustram através de diagramas **UML!** os pacotes que integram o sistema, fornecendo uma apresentação sucinta das classes que os compõem. O código-fonte completo do sistema pode ser encontrado no *CD* anexo a este trabalho, no qual também está contida sua documentação no padrão *Javadoc* (ORACLE, 2011).

6.1 platform.core

A biblioteca denominada **Core** (*platform.core*) une todos os recursos que são ou podem vir a ser utilizados pelas demais partes do sistema, visando maximizar o reaproveitamento de código, porém limitando-se a recursos genéricos (relacionados à plataforma como um todo, e não a uma parte específica dela). O diagrama de pacotes da Figura 6.2 apresenta os subpacotes da

biblioteca em questão, destacando também as relações existentes entre eles. Nas próximas seções esses subpacotes serão apresentados, destacando as funcionalidades implementadas pelas classes neles contidas.

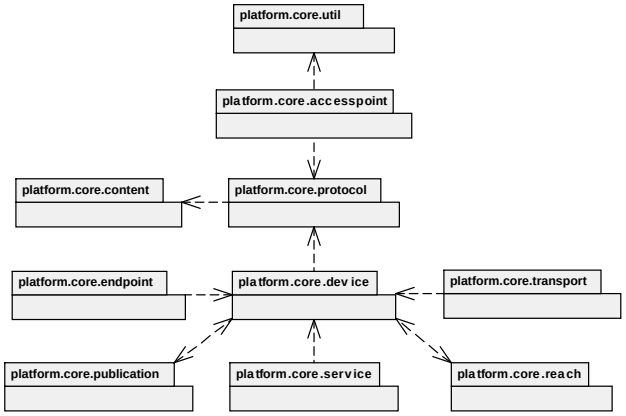


Figura 6.2: platform.core

6.1.1 platform.core.util

O pacote *platform.core.util* contém classes de propósito geral utilizadas por diversas outras classes do sistema. No diagrama de classes apresentado na Figura 6.3 pode-se observar a classe **ArrayUtil**, que contém métodos estáticos para a manipulação de vetores, e a classe **PacketStream**, que implementa o controle de envio e recebimento de pacotes (sequências de *bytes* delimitadas por tamanho) sobre o fluxo bidirecional de dados crus fornecido pela interface de comunicação.

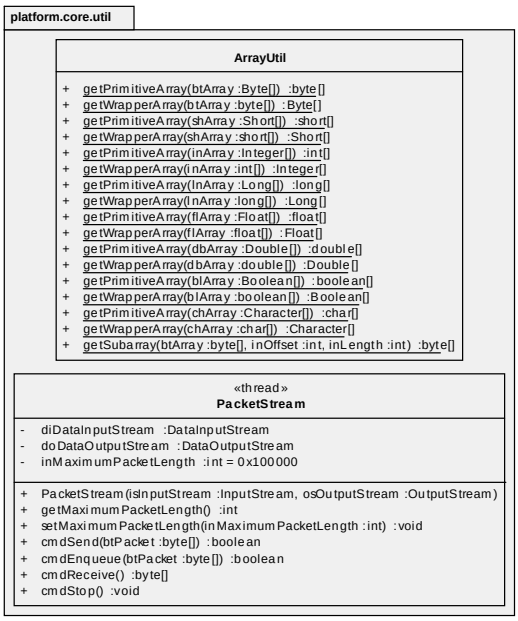


Figura 6.3: platform.core.util

6.1.2 platform.core.device

O pacote *platform.core.device* contém classes que representam dispositivos e as conexões entre dois dispositivos. A Figura 6.4 apresenta a classe **Device**, que representa genericamente dispositivos (concretizados no sistema como **Pontos-Fim**), a classe **Connection**, que gerencia uma conexão entre dois dispositivos na
plataforma, oferecendo métodos para troca de pacotes de dados, e a interface
Connection::Listener, que é utilizada por classes usuárias para recebimento de eventos relacionados a uma conexão.

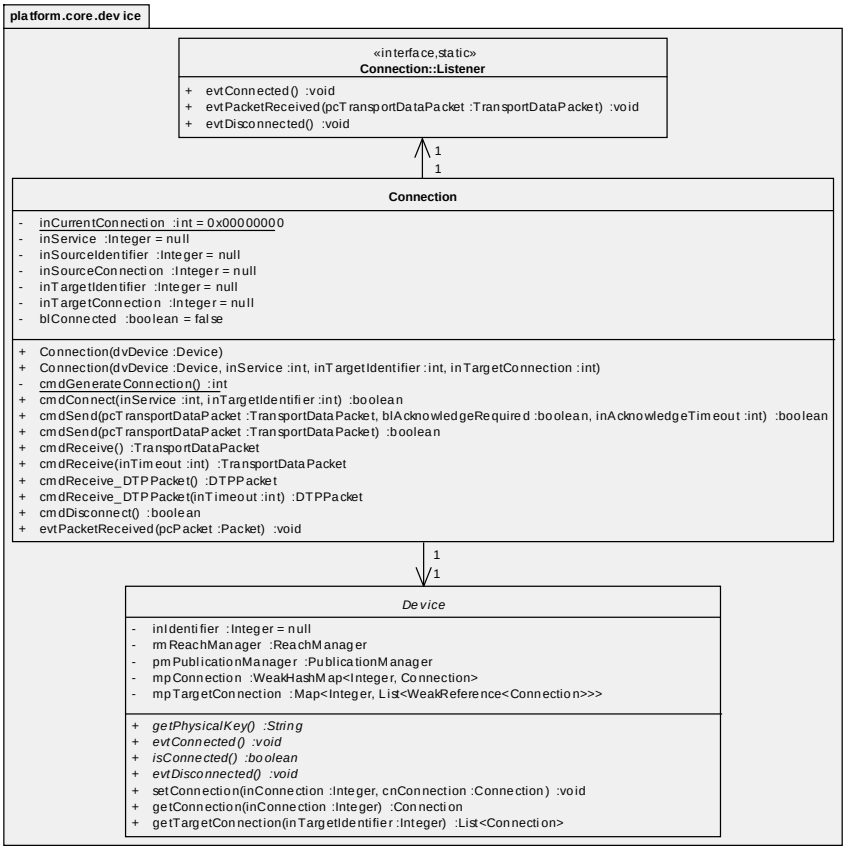


Figura 6.4: platform.core.device

6.1.3 platform.core.service

Na Figura 6.5 observa-se o diagrama de classes do pacote *platform.core.service*. Nela podem ser observadas as classes **Service** e **ServiceClient** que representam, respectivamente, serviços e clientes de serviços, que não possuem implementação concreta no núcleo do sistema, e devem ser utilizadas com herança para o desenvolvimento de aplicativos que fazem uso da plataforma.

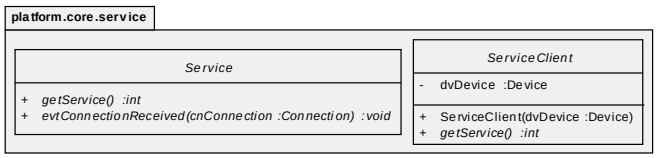


Figura 6.5: platform.core.service

6.1.4 platform.core.reach

O pacote *platform.core.reach* contém classes que facilitam a utilização de protocolos de alcance. Esses protocolos são

caracterizados pela possibilidade de roteamento sem a utilização de identificadores de dispositivo (seus pacotes são utilizados enquanto um identificador não está disponível) uma vez que a identificação de **Pontos-Fim** é feita através desse tipo de protocolo. O único protocolo de alcance implementado na plataforma é o *Protocolo de Gerenciamento de Identidade*, porém pode vir a ser necessária a implementação de novos protocolos desse tipo no futuro, dependendo de características específicas das interfaces de comunicação que vierem a ser tratadas. No diagrama de classes apresentado na Figura 6.6 podem ser observadas as classes **ReachHandler**, utilizada pelo **Roteador** para armazenar e gerenciar informações relacionadas aos protocolos de alcance, e a classe **ReachManager**, utilizada pelo **Ponto-Fim** para elaborar e tratar pacotes desses protocolos, oferecendo uma interface simplificada para o desempenho de tarefas a eles relacionadas.

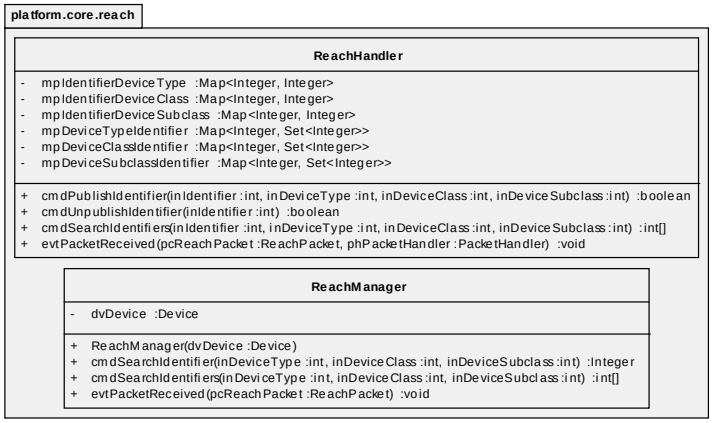


Figura 6.6: platform.core.reach

6.1.5 platform.core.publication

O pacote *platform.core.publication* contém classes facilitadoras para protocolos de publicação. Esses protocolos são caracterizados por permitirem a publicação ou remoção de recursos na plataforma, sendo tratados pelo **Roteador**. O único protocolo de publicação atualmente implementado é o *Protocolo de Publicação de Serviços*, porém futuramente pode ser necessário o desenvolvimento de novos protocolos com essa mesma característica, caso para o qual a plataforma já oferece a base necessária para o tratamento. A Figura 6.7 apresenta o

diagrama de classes do pacote em questão, no qual podem ser observadas as classes **PublicationHandler**, utilizada pelo **Roteador** para armazenar informações relacionadas aos protocolos de publicação e **PublicationManager**, utilizada pelo **Ponto-Fim** para gerar e tratar pacotes desse tipo de protocolo, oferecendo a interface de execução de tarefas de publicação.

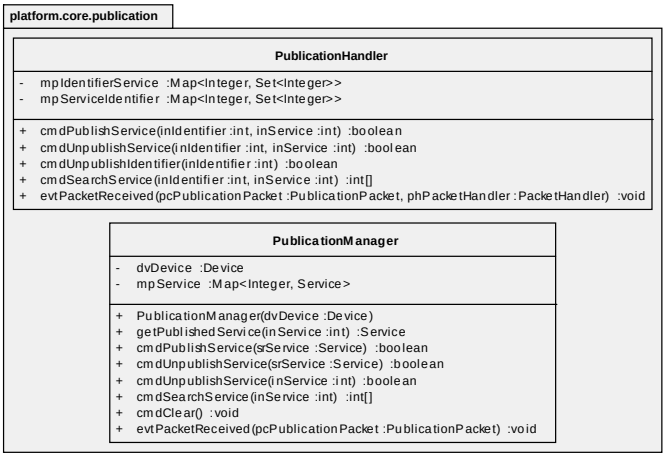


Figura 6.7: platform.core.publication

6.1.6 platform.core.transport

É comum na comunicação entre dispositivos que a interação ocorra em um regime requisição/resposta, de forma que um dos dispositivos envia uma requisição à contraparte, que é tratada e cujo resultado é enviado no sentido contrário em resposta à requisição recebida. Protocolos como o *HTTP* atuam em um regime requisição/resposta unidirecional (TANENBAUM, 2011), onde o cliente (*browser*) efetua requisições e o servidor as trata e envia respostas. Buscando maximizar as possibilidades e facilitar a utilização da plataforma, foi desenvolvida uma estrutura capaz de permitir a interação em regime requisição/resposta bidirecional, de forma que um **Ponto-Fim** seja capaz de enviar requisições à contraparte, assim como receber requisições enviadas por ela.

Conforme apresentado na Figura 6.8, o pacote *platform.core.transport* contém classes que facilitam a utilização de protocolos de transporte, que são caracterizados por permitir o estabelecimento de conexões virtuais entre dispositivos e oferecer a troca de pacotes de dados fim-a-fim na plataforma. O

único protocolo de transporte implementado é o *Protocolo de Transporte de Dados*, desenvolvido para transporte de mensagens complexas entre dispositivos, porém no futuro pode tornar-se necessário o transporte de outros tipos de conteúdo, como por exemplo **XML!**, caso para o qual a plataforma já oferece facilidades para o desenvolvimento. A classe **DTPTerminal** do pacote em questão faz uso do *Protocolo de Transporte de Dados* para permitir que dois dispositivos virtualmente conectados sejam capazes de enviar e processar requisições (cuja resposta carrega conteúdo) e comandos (cuja resposta carrega apenas uma notificação de sucesso de execução) em ambas as direções. Essa classe possibilita também o transporte de exceções ocorridas na execução da solicitação pela contraparte, permitindo maior facilidade na depuração de erros durante o desenvolvimento. A interface **DTPTerminal::Listener** deve ser utilizada pela classe usuária para recebimento de eventos e de requisições enviadas pela contraparte.

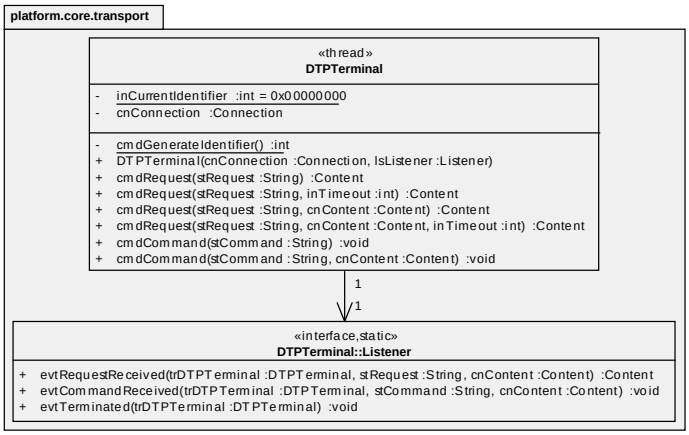


Figura 6.8: platform.core.transport

6.1.7 platform.core.protocol

O pacote *platform.core.protocol* contém uma estrutura de classes que representa e facilita a utilização dos protocolos do sistema. Uma das principais tarefas realizadas pelas classes nele contidas é a elaboração e leitura de vetores de *bytes* que contêm pacotes desses protocolos, sendo essas utilizadas por todos os aplicativos da plataforma. O pacote de topo representado no diagrama de classes da Figura 6.9 contém as classes **Protocol**, **Packet** e

PacketHandler, que representam genericamente protocolos, pacotes e manipuladores de pacotes (classes que enviam e recebem pacotes), respectivamente.

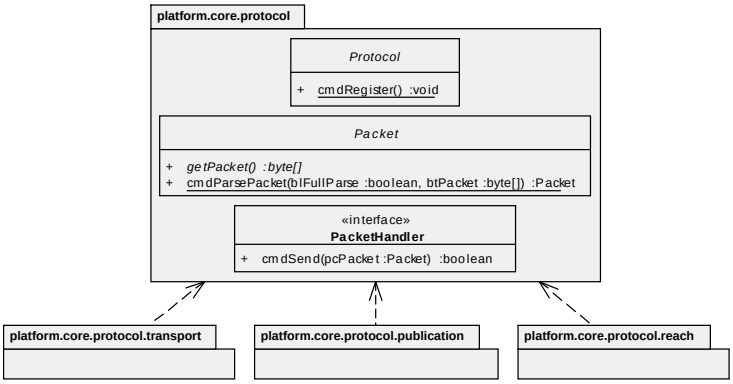


Figura 6.9: platform.core.protocol

Como pode ser observado na Figura 6.9, fazem parte do pacote *platform.core.pro-tocol* os subpacotes *platform.core.protocol.reach*, *platform.core.protocol.publication* e *platform.core.protocol.transport* que contêm, respectivamente, classes de representação e tratamento para protocolos de alcance, publicação e transporte. Nessas classes encontram-se recursos comuns a todos os protocolos suportados, além de uma estrutura de registro e acesso a esses recursos. Atrelados aos pacotes anteriormente citados, encontram-se os pacotes *platform.core.protocol.reach.imp*, *platform.core.protocol.publication.spp* e *platform.core.protocol.transport.dtp*. Esses pacotes contêm, respectivamente, classes de tratamento do *Protocolo de Gerenciamento de Identidade*, do *Protocolo de Publicação de Serviços* e do *Protocolo de Transporte de Dados*, assim como as classes de tratamento para cada tipo de pacote desses protocolos. Através dessa estrutura de classes, é possível o acoplamento de novos protocolos e de novos pacotes à plataforma, bastando para isso que seja efetuado o registro de novas classes de representação durante a inicialização do sistema.

6.1.8 platform.core.accesspoint

O pacote *platform.core.accesspoint* contém classes que representam e são a base para o desenvolvimento de **Pontos de Acesso**. Conforme pode ser observado no diagrama de classes

apresentado pela Figura 6.10, a classe **Accesspoint** apresenta a implementação abstrata básica de um **Ponto de Acesso**, oferecendo uma conexão com o **Roteador** através da classe **AccesspointRouter**, que a estabelece e, em caso de perda, restabelece a conexão assim que possível. Para implementação de um novo **Ponto de Acesso**, a classe **Accesspoint** deve ser herdada.

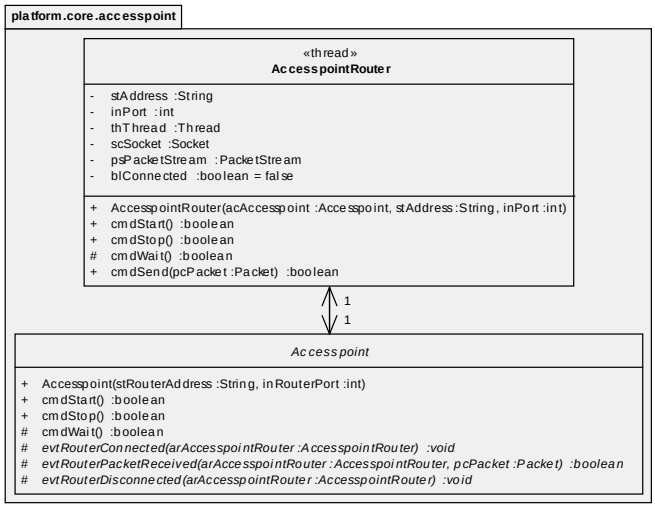


Figura 6.10: platform.core.accesspoint

6.1.9 platform.core.endpoint

As classes contidas no pacote *platform.core.endpoint* representam e podem ser utilizadas como base para o desenvolvimento de **Pontos-Fim** que tratem novas interfaces de comunicação. A Figura 6.11 apresenta o diagrama de classes desse pacote, que contém a classe **Endpoint**, que representa genericamente um **Ponto-Fim** e deve ser herdada para a implementação desse tipo de aplicativo, e a interface **Endpoint::Listener**, que deve ser utilizada por classes usuárias do **Ponto-Fim** para recebimento de eventos disparados por esse, como por exemplo a conexão e desconexão com um **Ponto de Acesso**.

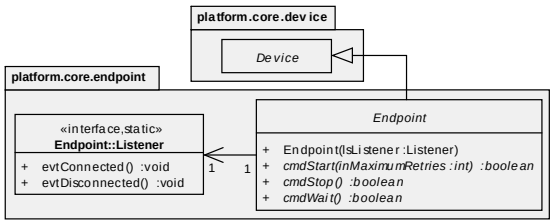


Figura 6.11: platform.core.endpoint

6.1.10 platform.core.content

A classe **Content**, contida no pacote *platform.core.content*, implementa a estrutura de armazenamento de informações utilizada para troca de mensagens complexas na plataforma. Essa estrutura consiste em um mapa chave/valor onde a chave é uma cadeia de caracteres única, mapeada a um valor cujo tipo pode variar de acordo com a necessidade do usuário. No diagrama de classes apresentado pela Figura 6.12, é possível observar a estrutura dessa classe, assim como os métodos que fornecem as funcionalidades necessárias à sua utilização. Vale ressaltar que muitos dos métodos de obtenção e atribuição de valores de campos de diferentes tipos foram suprimidos a fim de tornar o diagrama mais sucinto. A documentação completa da classe em questão pode ser observada no *CD* anexo a este trabalho.

Figura 6.12: platform.core.content

6.2 Router

O aplicativo denominado **Router**, contido no pacote *platform.router*, consiste na implementação do **Roteador** da plataforma. Como pode ser observado na Figura 6.13, que apresenta o diagrama de classes do pacote, esse aplicativo é integrado pela classe principal da implementação, denominada **Router**, pela classe **RouterAccesspoints**, responsável por gerenciar o recebimento de conexões de **Pontos de Acesso** e delegar o tratamento de cada conexão à terceira classe, **RouterAccesspoint**, que gerencia a troca de dados com o **Ponto de Acesso** conectado.

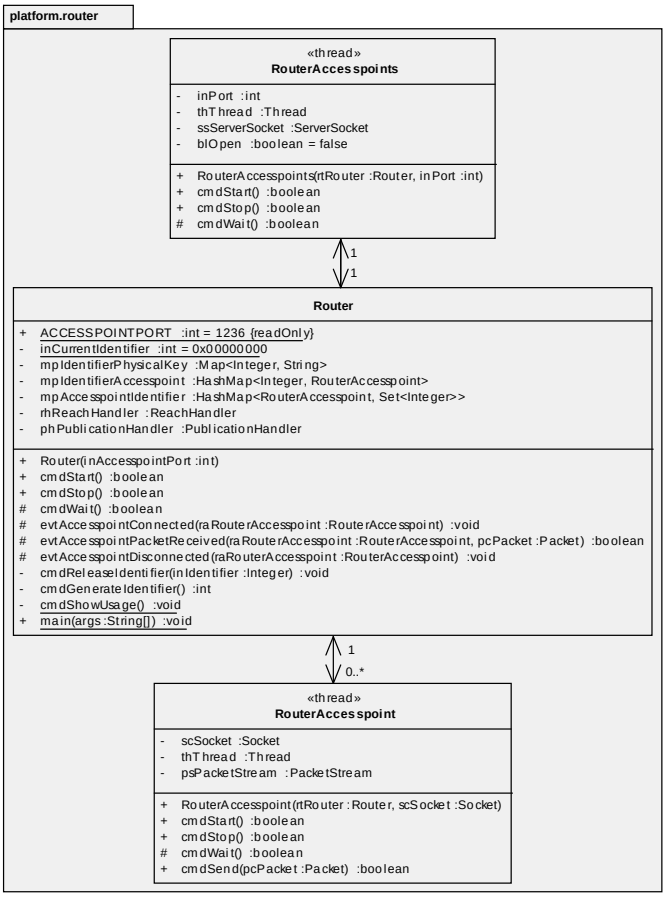


Figura 6.13: platform.router

6.3 TCPIPAccesspoint

O aplicativo denominado **TCPIPAccesspoint** está contido no pacote *platform.accesspoint.tcpip* e consiste na implementação do **Ponto de Acesso** para o protocolo **TCP/IP!**. Na Figura 6.14 é apresentado o diagrama de classes do pacote em questão, e nela pode-se observar a classe principal da implementação, denominada **TCPIPAccesspoint**, a classe **TCPIPAccesspointEndpoints**, que é responsável por receber conexões **TCP/IP!** de múltiplos **Pontos-Fim** e delegar o tratamento de cada uma delas à classe **TCPIPAccesspointEndpoint**, que gerencia a interface de comunicação com um **Ponto-Fim**.

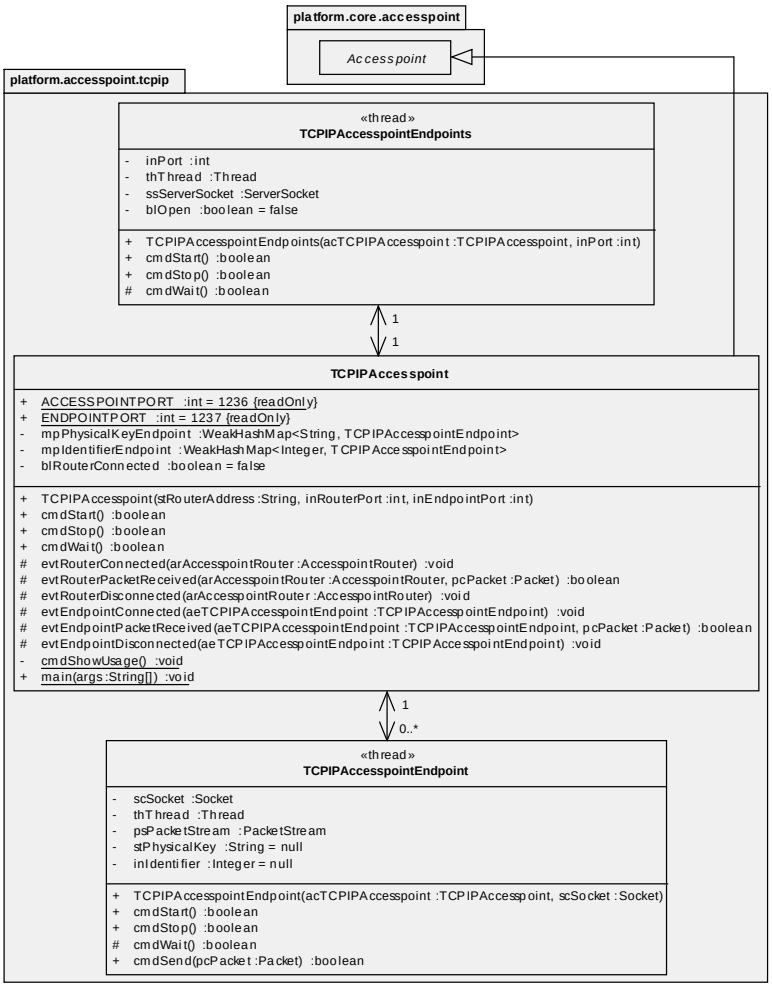


Figura 6.14: platform.accesspoint.tcpip

6.4 TCPIPEndpoint

O pacote chamado *platform.endpoint.tcpip* contém a biblioteca denominada
TCPIPEndpoint, que consiste na implementação do **Ponto-Fim** para o protocolo **TCP/IP!**. Essa, em conjunto com a biblioteca **Core**, é responsável por acessar a plataforma através de uma instância do **Ponto de Acesso** para o protocolo **TCP/IP!** e fornecer a *API* necessária para a execução de todas as tarefas suportadas pelo sistema. A Figura 6.15 apresenta o diagrama de classes dessa biblioteca, e nela pode-se observar a classe principal da implementação, denominada **TCPIPEndpoint**, e a classe **TCPIPEndpointAccesspoint**, responsável por gerenciar a interface de comunicação com o **Ponto de Acesso**.

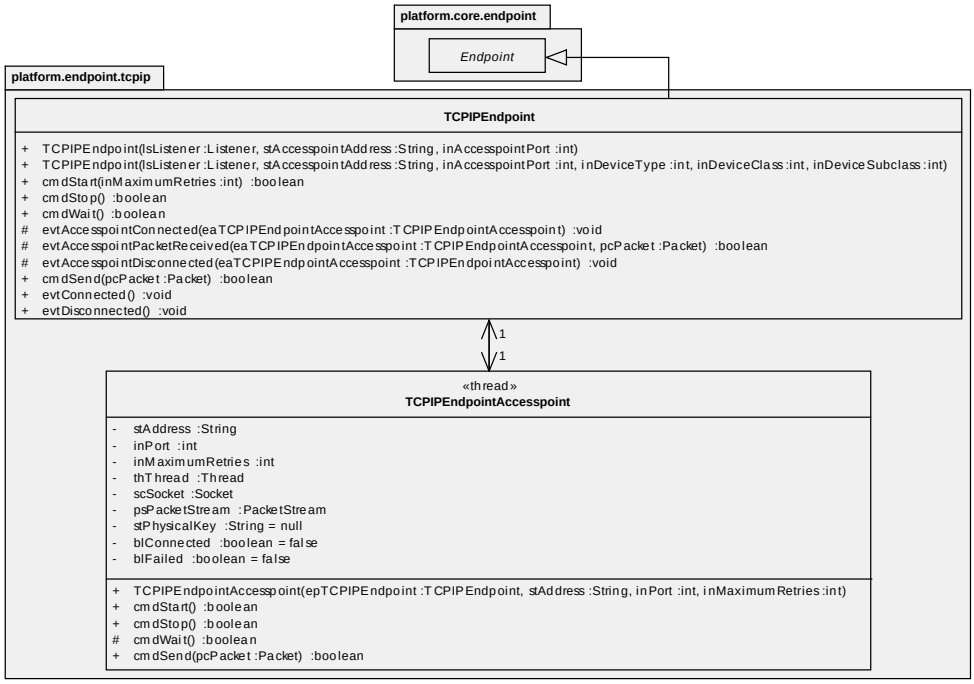


Figura 6.15: platform.endpoint.tcpip

6.5 Descrição geral do sistema

O diagrama de classes apresentado na Figura 6.16 sintetiza todos os pacotes e classes da plataforma e suas relações, fornecendo uma visão geral sobre o funcionamento do sistema e a forma como seus componentes interagem. A fim de tornar o diagrama mais sucinto, foram suprimidos os detalhes de implementação. Para maiores informações, sugere-se o *CD* anexo a este trabalho, que contém o código-fonte completo e a documentação do sistema. Pode-se destacar na Figura 6.16 que:

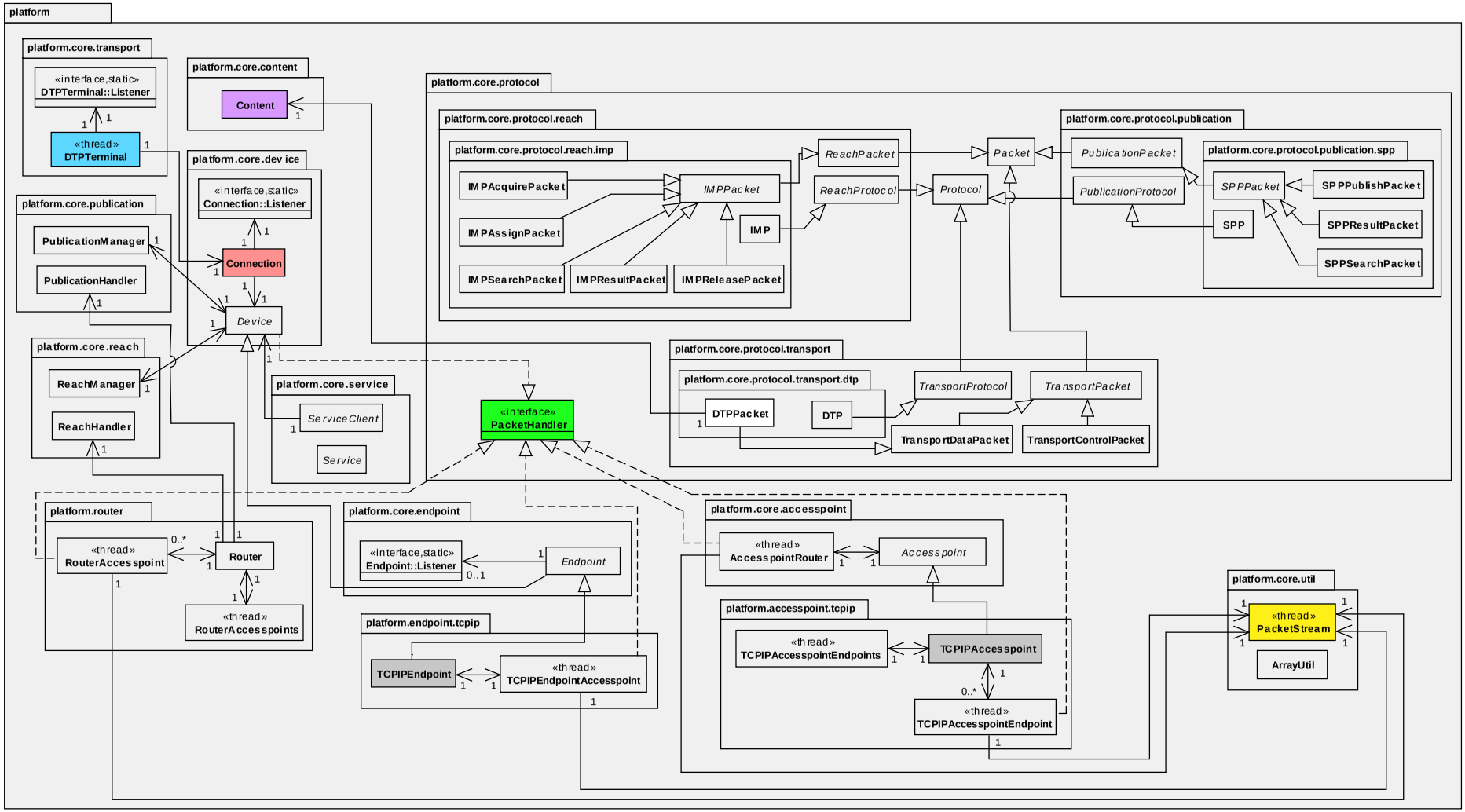


Figura 6.16: Descrição geral do sistema

- A interface **PacketHandler** (destacada no diagrama na cor verde) é implementada por todas as classes que manipulam a interface de comunicação com outras partes do sistema, no caso **RouterAccesspoint**, **AccesspointRouter**, **TCPIPAccesspointEndpoint** e **TCPIPEndpointAccesspoint**, e que portanto são capazes de enviar e receber pacotes de dados. Da mesma forma, a classe **PacketStream** (destacada em amarelo), responsável por transmitir sequências delimitadas de *bytes* através de uma conexão, é utilizada sobre a interface de comunicação para generalizar a troca de pacotes e, portanto, é utilizada em conjunto com a interface **PacketHandler**;
- Para construção de uma instância da classe **DTPTerminal** (em azul no diagrama) deve ser fornecida uma instância da classe **Connection** (em vermelho), a fim de obter-se suporte à execução bidirecional de operações em regime requisição/resposta. Vale ressaltar que, para o correto funcionamento desse mecanismo, ambos os dispositivos conectados devem fazer uso da classe **DTPTerminal**;
- As classes do pacote *platform.core.protocol* são acessadas estaticamente para leitura de pacotes recebidos através da interface de comunicação e instanciadas localmente para elaboração de pacotes a enviar, de forma que não existem referências explícitas a elas no diagrama apresentado. Além disso, pode-se observar que esse pacote oferece um conjunto expansível de classes para tratamento de tipos de protocolo, protocolos e pacotes, o que torna simples adicionar a essa estrutura novos elementos em qualquer nível;
- A classe **DTPPacket** (destacada em cor branca no diagrama), que representa um pacote de dados do *Protocolo de Transporte de Dados*, é a única que referencia a classe **Content** (destacada em roxo), o que evidencia sua função na plataforma;
- Observando as classes **TCPIPAccesspoint** e **TCPIPEndpoint**, destacadas em cinza, pode-se verificar a forma como deve ser utilizado o núcleo do sistema para o desenvolvimento de **Pontos de Acesso** e **Pontos-Fim**, através da herança das classes **Accesspoint** e **Endpoint**,

respectivamente.

Neste capítulo foram apresentados os aspectos técnicos da implementação do sistema. Já no próximo capítulo são apresentadas instruções e exemplos para a utilização da plataforma desenvolvida.

7 UTILIZAÇÃO DA PLATAFORMA

Para o desenvolvimento de um aplicativo que faz uso da plataforma através do protocolo **TCP/IP!**, devem ser executados os seguintes passos:

- Importar a biblioteca **Core** (arquivo "Core.jar"), obtendo acesso às classes básicas do sistema;
- Importar a biblioteca **TCPIPEndpoint** (arquivo "TCPIPEndpoint.jar"), obtendo acesso à implementação do **Ponto-Fim** para o protocolo **TCP/IP!**;
- Desenvolver a classe principal do projeto, que será responsável pela preparação das bibliotecas da plataforma e pela inicialização do aplicativo;
- Assegurar que, durante a inicialização do aplicativo e antes de qualquer utilização da *API* da plataforma, o método estático *cmdRegister* da classe
  Protocol seja invocado, já que esse método é responsável pelo registro de todos os protocolos do sistema;
- Criar uma instância da classe **TCPIPEndpoint** (**Ponto-Fim** para o protocolo **TCP/IP!**), cujo construtor recebe como parâmetro o endereço e a porta (o valor padrão é 1237) do **Ponto de Acesso** a ser utilizado, além de uma referência ao objeto que tratará eventos relacionados à interação com a plataforma;
- Efetuar o estabelecimento da comunicação com o **Ponto de Acesso** e a aquisição de um identificador, através da chamada ao método *cmdStart* do **Ponto-Fim**, que recebe como parâmetro o número máximo de tentativas para o estabelecimento de comunicação;

- Preferencialmente o aplicativo deve garantir que, antes de sua finalização, o método *cmdStop* do **Ponto-Fim** seja invocado, finalizando a comunicação com a plataforma e liberando o identificador.

De modo a demonstrar a utilização da plataforma, foram desenvolvidos trechos de código e aplicativos na linguagem *Java* que fornecem exemplos simples de comunicação explorando os recursos do sistema. Os trechos de código apresentados nas próximas seções exemplificam a utilização da plataforma para o fornecimento de um serviço, para o consumo de um serviço e para a elaboração de mensagens complexas, respectivamente. Na seção final serão apresentados os aplicativos desenvolvidos e a forma como esses devem ser utilizados.

7.1 Fornecimento de serviço

O Trecho de Código 7.1 apresenta um exemplo de utilização da *API* da plataforma para publicação de um serviço de *echo*, ou seja, que responde a todo pacote recebido com o mesmo pacote. No exemplo, as seguintes operações são executadas:

1. Linha 8: Registra os protocolos da plataforma;
2. Linha 10: Cria instância do **Ponto-Fim** para o protocolo **TCP/IP!**;
3. Linha 12: Inicializa o **Ponto-Fim**, conectando-o ao **Ponto de Acesso** indicado;
4. Linha 17: Publica o serviço de *echo*, permitindo que outros **Pontos-Fim** sejam capazes de localizá-lo;
5. Linha 19: Aguarda que a execução do **Ponto-Fim** seja concluída, ou seja, a *thread* principal do aplicativo aguardará a finalização do **Ponto-Fim** (através da invocação do método *cmdStop*) por um agente externo, caso esse exista, ou a ocorrência de um erro que termine sua execução. No exemplo apresentado não existe um agente externo que solicite a finalização do **Ponto-Fim**, portanto a invocação do método *cmdWait* só será concluída (retornará) em caso de erro;

6. Linha 22: Finaliza a execução do **Ponto-Fim**, desconectando-o do **Ponto de Acesso** e liberando o identificador. Neste exemplo, esse método não será invocado a menos que algum erro ocorra;
7. Linha 26: O evento *evtConnectionReceived* é chamado pela plataforma para cada conexão recebida por um serviço. Neste exemplo, esse evento é responsável por iniciar uma *thread* (linha 28) que atende a um cliente do serviço publicado;
8. Linha 34: Aguarda o recebimento de uma mensagem enviada pelo **Ponto-Fim** cliente por tempo indeterminado;
9. Linha 40: Envia a mensagem recebida sem alterações de volta ao **Ponto-Fim** cliente.

Trecho de Código 7.1: Fornecimento de serviço

```
1 // Exemplo de utilização para fornecimento de serviço
2 public class ServiceExample extends Service {
3     // Identificador do serviço
4     private static final int SERVICE = 1;
5     // Método de execução
6     public void cmdRun() throws Exception {
7         // Registra protocolos
8         Protocol.cmdRegister();
9         // Cria ponto-fim
10        Endpoint epEndpoint = new TCPIPEndpoint(null, "
              localhost", 1237);
11        // Inicia
12        if (!epEndpoint.cmdStart(5)) {
13            throw new Exception("Erro ao conectar à
                  plataforma");
14        }
15        try {
16            // Publica serviço
17            epEndpoint.getPublicationManager().
                  cmdPublishService(this);
18            // Aguarda
19            epEndpoint.cmdWait();
20        } finally {
21            // Termina
22            epEndpoint.cmdStop();
23        }
24    }
25    @Override
26    public void evtConnectionReceived(final Connection
          cnConnection) {
27        // Trata conexão
28        new Thread(new Runnable() {
29            @Override
30            public void run() {
31                // Laço de recebimento
32                while (true) {
33                    // Recebe pacote
34                    TransportDataPacket pcTransportDataPacket
                          = cnConnection.cmdReceive();
35                    // Verifica pacote
36                    if (pcTransportDataPacket == null) {
37                        break;
38                    }
39                    // Envia pacote
40                    if (!cnConnection.cmdSend(
                          pcTransportDataPacket)) {
41                        System.err.println("Erro ao enviar
                              pacote");
42                    }
43                }
44            }
45        }).start();
46    }
47    @Override
48    public int getService() {
49        return SERVICE;
50    }
51 }
```

7.2 Consumo de serviço

O Trecho de Código 7.2 apresenta um exemplo de utilização da *API* da plataforma para acesso a um serviço de *echo* publicado por outro **Ponto-Fim**. No exemplo, as seguintes operações são executadas:

1. Linha 8: Registra os protocolos da plataforma;
2. Linha 10: Cria instância do **Ponto-Fim** para o protocolo **TCP/IP!**;
3. Linha 12: Inicializa o **Ponto-Fim**, conectando-o ao **Ponto de Acesso** indicado;
4. Linha 17: Busca identificadores de dispositivos que publicaram o serviço de *echo*;
5. Linha 27: Cria uma conexão com o serviço no primeiro dispositivo encontrado;
6. Linha 29: Estabelece a conexão com o serviço;
7. Linha 34: Cria e preenche uma mensagem;
8. Linha 38: Envia a mensagem ao serviço;
9. Linha 42: Recebe a resposta do serviço;
10. Linha 47: Finaliza a conexão com o serviço;
11. Linha 51: Finaliza a execução do **Ponto-Fim**, desconectando-o do **Ponto de Acesso** e liberando o identificador.

Trecho de Código 7.2: Consumo de serviço

```
1 // Exemplo de utilização para consumo de serviço
2 public class ClientExample {
3     // Identificador do serviço
4     private static final int SERVICE = 1;
5     // Método de execução
6     public void cmdRun() throws Exception {
7         // Registra protocolos
8         Protocol.cmdRegister();
9         // Cria ponto-fim
10        Endpoint epEndpoint = new TCPIPEndpoint(null, "
              localhost", 1237);
11        // Inicia
12        if (!epEndpoint.cmdStart(5)) {
13            throw new Exception("Erro ao conectar à
                  plataforma");
14        }
15        try {
16            // Busca serviço
17            int[] inIdentifiers = epEndpoint.
                  getPublicationManager().cmdSearchService(
                  SERVICE);
18            // Verifica identificadores
19            if (inIdentifiers == null) {
20                throw new Exception("Erro ao buscar serviço")
                      ;
21            }
22            // Verifica identificadores
23            if (inIdentifiers.length < 1) {
24                throw new Exception("Serviço não encontrado")
                      ;
25            }
26            // Cria conexão
27            Connection cnConnection = new Connection(
                  epEndpoint);
28            // Conecta ao primeiro dispositivo
29            if (!cnConnection.cmdConnect(SERVICE,
                  inIdentifiers[0])) {
30                throw new Exception("Erro ao conectar ao
                      serviço");
31            }
32            try {
33                // Cria conteúdo
34                Content cnContent = new Content();
35                cnContent.setString("stUserName", "username")
                      ;
36                cnContent.setString("stPassWord", "secret");
37                // Envia pacote
38                if (!cnConnection.cmdSend(new DTPPacket(
                      cnContent))) {
39                    throw new Exception("Erro ao enviar");
40                }
41                // Recebe
42                DTPPacket pcReceived = cnConnection.
                      cmdReceive_DTPPacket();
43                // Exibe conteúdo recebido
44                System.out.println("Resposta: " + pcReceived.
                      getContent());
45            } finally {
```

```
46                  // Desconecta
47                  cnConnection.cmdDisconnect();
48              }
49          } finally {
50              // Termina
51              epEndpoint.cmdStop();
52          }
53      }
54 }
```

7.3 Mensagens complexas

O Trecho de Código 7.3 apresenta um exemplo de elaboração de uma mensagem complexa. Esse exemplo permite extrair informações sobre um diretório do sistema de arquivos e armazená-las no conteúdo de uma mensagem, além de suportar um limite de profundidade a ser explorado nesse processo. No exemplo, as seguintes operações são executadas:

1. Linha 4: Cria um conteúdo para armazenamento das informações do diretório;
2. Linhas 6 e 8: Cria listas que conterão as informações dos arquivos e dos subdiretórios contidos no diretório, respectivamente;
3. Linha 10: Enumera os arquivos e subdiretórios existentes no diretório;
4. Linhas 16 e 23: Diferencia arquivos de subdiretórios;
5. Linhas 18 e 25: Cria e preenche um conteúdo com as informações de cada um dos arquivos e subdiretórios, respectivamente;
6. Linhas 22 e 35: Adiciona à lista correspondente as informações de cada um dos arquivos e subdiretórios, respectivamente;
7. Linha 28: Verifica se, de acordo com a profundidade da exploração, as informações dos subdiretórios devem ser incluídas na mensagem;
8. Linha 30: Chama a função de forma recursiva para exploração do subdiretório, decrementando a profundidade de exploração, e inclui na mensagem as informações resultantes (linha 32).

O exemplo apresentado retorna, portanto, uma mensagem composta de forma recursiva que contém informações sobre um determinado diretório do sistema de arquivos, assim como dos arquivos e subdiretórios nesse contidos, com possibilidade de limitação da profundidade de exploração.

Trecho de Código 7.3: Elaboração de mensagens complexas

```
1 // Exemplo de geração de mensagem complexa
2 public static Content getDirectoryContent(File flDirectory,
      int inDepth) {
3     // Cria conteúdo
4     Content cnContent = new Content();
5     // Arquivos
6     List<Content> lsFile = new LinkedList<Content>();
7     // Diretórios
8     List<Content> lsDirectory = new LinkedList<Content>();
9     // Lista arquivos
10    File[] flFiles = flDirectory.listFiles();
11    // Itera arquivos
12    for (int i = 0; i < flFiles.length; i++) {
13        // Obtém arquivo
14        File flFile = flFiles[i];
15        // Verifica arquivo
16        if (flFile.isFile()) {
17            // Cria conteúdo do arquivo
18            Content cnFile = new Content();
19            cnFile.setString("stName", flFile.getName());
20            cnFile.setLong("lnSize", flFile.length());
21            // Adiciona arquivo
22            lsFile.add(cnFile);
23        } else if (flFile.isDirectory()) {
24            // Cria conteúdo do diretório
25            Content cnDirectory = new Content();
26            cnDirectory.setString("stName", flFile.getName())
                  ;
27            // Verifica profundidade
28            if ((inDepth > 1) || (inDepth < 0)) {
29                // Obtém conteúdo do diretório
30                Content cnDirectoryContent =
                      getDirectoryContent(flFile, inDepth-1);
31                // Atribui conteúdo do diretório
32                cnDirectory.setContent("cnContent",
                      cnDirectoryContent);
33            }
34            // Adiciona diretório
35            lsDirectory.add(cnDirectory);
36        }
37    }
38    // Atribui vetores de arquivos e de diretórios
39    cnContent.setContentArray("cnFile", lsFile.toArray(new
          Content[] {}));
40    cnContent.setContentArray("cnDirectory", lsDirectory.
          toArray(new Content[] {}));
41    return cnContent;
42 }
```

7.4 Aplicativos

Para exemplificação da utilização da plataforma, foram desenvolvidos aplicativos que demonstram o funcionamento do sistema em aplicações práticas. Vale ressaltar que, antes de utilizar quaisquer desses projetos, devem ser iniciadas uma instância do **Roteador**, através do executável de nome *Router.exe* para *Microsoft Windows*, *Router.sh* para *Linux* ou *Router.jar* para outras plataformas, e uma instância do **Ponto de Acesso** para o protocolo **TCP/IP!**, através do executável *TCPIPAccesspoint.exe* para *Microsoft Windows*, *TCPIPAccesspoint.sh* para *Linux* ou *TCPIPAccesspoint.jar* para outras plataformas.

Para a exemplificação foram desenvolvidos os aplicativos **Explorer**, que permite a execução de todas as operações suportadas pelo sistema, e **Chat**, que implementa um bate-papo simples. Ambos os projetos utilizam a biblioteca denominada **Example**, que contém a implementação dos serviços e clientes de serviços utilizados nesses aplicativos. As próximas seções apresentarão esses projetos e um breve manual de utilização deles. O código-fonte completo e os executáveis desses aplicativos encontram-se no *CD* anexo a este trabalho.

7.4.1 Aplicativo de exploração

O projeto denominado **Explorer** consiste em um aplicativo de exploração extensiva da plataforma, que permite a execução de qualquer tarefa suportada pelo sistema através de uma interface gráfica. Esse aplicativo fornece também a publicação do serviço de bate-papo que será apresentado na Seção 7.4.2.

Para utilizar esse aplicativo, deve-se iniciá-lo através do executável de nome *Explorer.exe* para *Microsoft Windows*, *Explorer.sh* para *Linux* ou *Explorer.jar* para outras plataformas. Assim que o aplicativo for iniciado, a tela da Figura 7.1 será apresentada. Nessa tela deverá ser informado o endereço e a porta do **Ponto de Acesso** ao qual deverá ser feita a conexão, e deverá ser utilizado o botão *Connect* para estabelecimento da comunicação com o **Ponto de Acesso** e aquisição de um identificador. É possível informar também um tipo, classe e

subclasse (estabelecidos por convenção) para o dispositivo na plataforma; caso esses dados não sejam informados, serão admitidos os valores padrão.

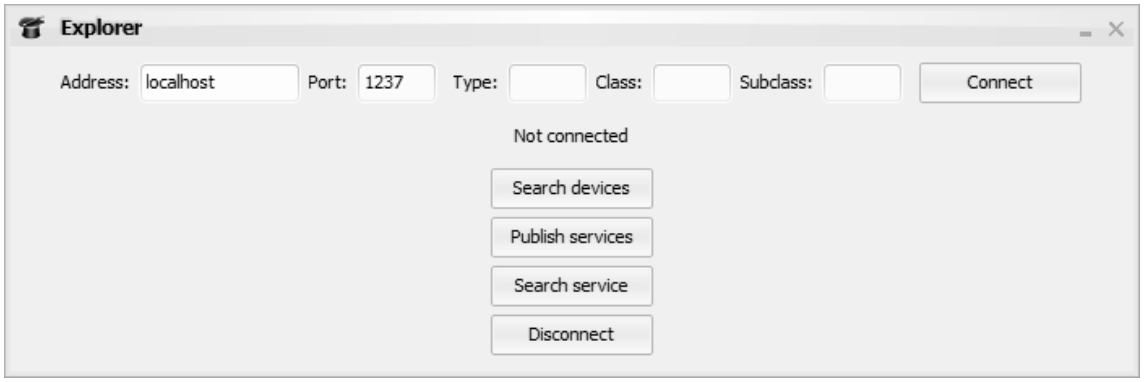


Figura 7.1: Explorer - Tela inicial

Uma vez que a conexão ao **Ponto de Acesso** estiver estabelecida e o **Ponto-Fim** possua um identificador, esse será exibido na tela conforme a Figura 7.2.

Figura 7.2: Explorer - Conectado

A função de busca de dispositivos, apresentada na Figura 7.3, pode ser acessada através do botão *Search devices* da tela principal do aplicativo **Explorer**. Essa função permite a enumeração dos identificadores dos dispositivos conectados à plataforma, possibilitando a filtragem (opcional) por tipo, classe e subclasse de dispositivo. Uma vez que os valores para esses filtros devem ser estabelecidos por convenção, neste aplicativo os valores informados são livres.

Figura 7.3: Explorer - Buscar dispositivos

A função de publicação de serviços, apresentada na Figura 7.4, pode ser acessada através do botão *Publish services* da tela principal do aplicativo **Explorer** e possibilita realizar a publicação e a remoção de serviços. Estão disponíveis os serviços de *echo*, de informações e de bate-papo. O serviço de *echo* responde a qualquer pacote com uma cópia do mesmo pacote. Já o serviço de informações envia um pacote contendo as propriedades do sistema no qual o serviço é executado, finalizando a conexão em seguida. Por sua vez, o serviço de bate-papo é utilizado pelo aplicativo de exemplo apresentado na Seção 7.4.2.

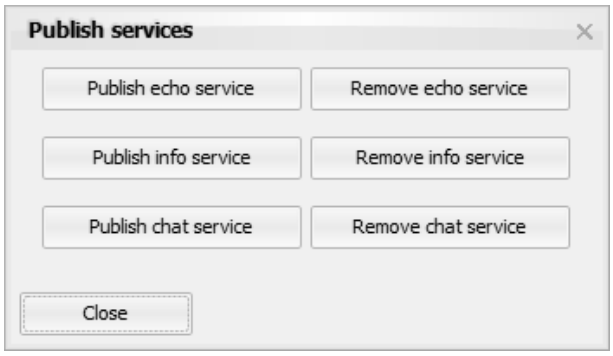


Figura 7.4: Explorer - Publicar serviços

A função de busca de serviço, apresentada na Figura 7.5, pode ser acessada através do botão *Search service* da tela principal do aplicativo. Através dela é possível a entrada de um identificador de serviço e a busca dos identificadores dos dispositivos que o

publicaram na plataforma. Uma vez que os identificadores de serviço são estabelecidos por convenção, o campo para essa informação é livre. Para os serviços publicados através da função de publicação de serviços, os identificadores para os serviços de *echo*, informações e bate-papo são, respectivamente, 1, 2 e 1001.

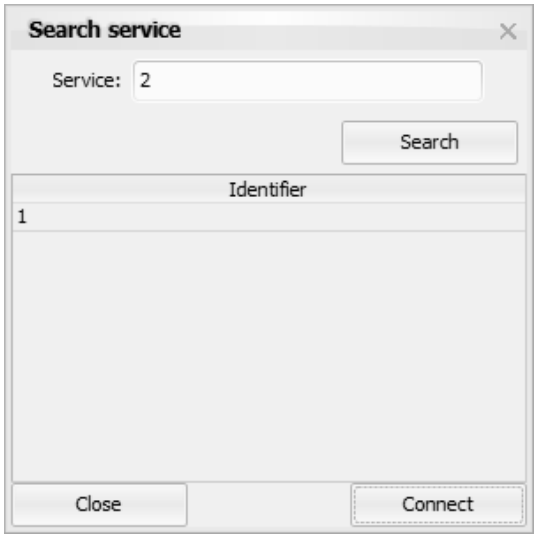


Figura 7.5: Explorer - Buscar serviço

Uma vez localizado o identificador de um dispositivo que publicou um determinado serviço, é possível estabelecer conexão a ele selecionando-se o identificador na lista e clicando em *Connect*, conforme ilustrado na Figura 7.6.

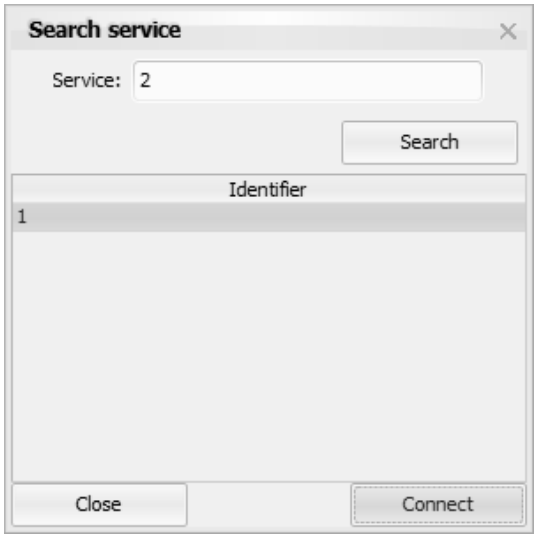


Figura 7.6: Explorer - Conectar a um serviço

Uma vez efetuada a conexão a um serviço, é apresentada a tela de acesso da Figura 7.7. Essa tela consiste em um histórico de

troca de mensagens, onde consta o horário, a direção (enviada ou recebida) e o conteúdo da mensagem, que pode ser observado através de um editor de conteúdo. É possível também a elaboração e envio de mensagens, utilizando o botão *Send*.

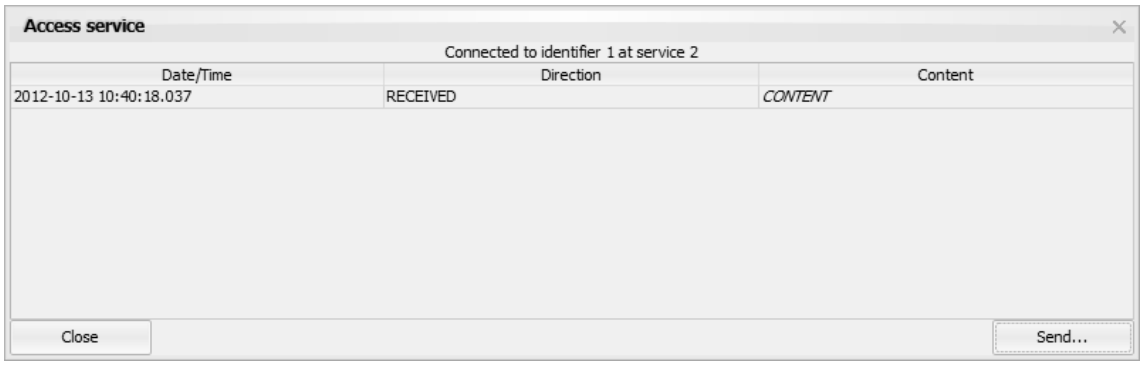


Figura 7.7: Explorer - Acessar serviço

Para envio de uma mensagem, a tela da Figura 7.8 é apresentada. Essa tela consiste em um editor completo de conteúdos de mensagens capaz de criar e remover campos (utilizando a barra de ferramentas no topo da janela), além de editar seus valores com a utilização dos editores acessíveis na listagem de campos. Esse editor suporta a criação e a edição de conteúdos recorrentes, através dos tipos de campo *CONTENT* e *CONTENT[]*, que indicam que o campo apresenta um conteúdo recorrente ou um vetor de conteúdos recorrentes, respectivamente.

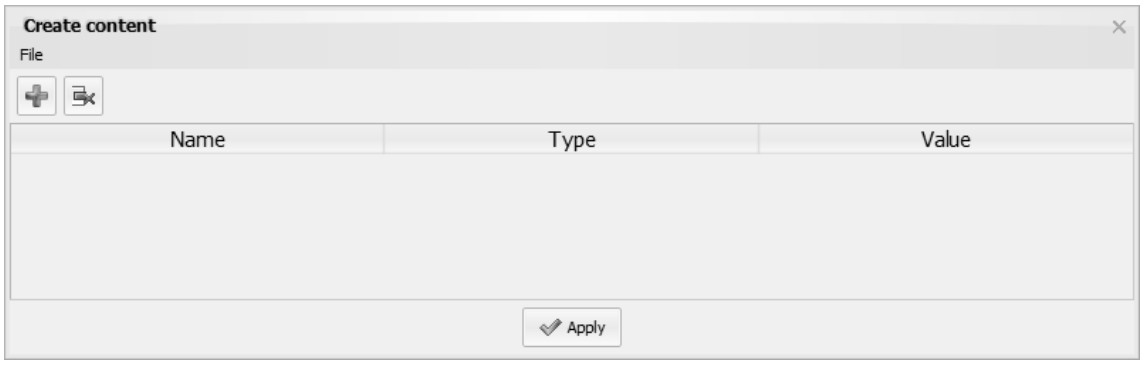


Figura 7.8: Explorer - Enviar mensagem

7.4.2 Aplicativo de bate-papo

O aplicativo denominado **Chat** implementa um bate-papo que demonstra a comunicação entre múltiplos dispositivos centralizada por um serviço. Para utilizar esse aplicativo, deve-se primeiramente utilizar uma instância do aplicativo de exploração que foi descrito na Seção 7.4.1 e efetuar a publicação do serviço de bate-papo através da função de publicação de serviços. Em seguida, o aplicativo **Chat** deve ser iniciado através do executável de nome *Chat.exe* para *Microsoft Windows*, *Chat.sh* para *Linux* ou *Chat.jar* para outras plataformas.

Uma vez iniciado o aplicativo, a tela da Figura 7.9 será apresentada. Nessa tela, deve ser informado o endereço e a porta do **Ponto de Acesso** ao qual o dispositivo deverá ser conectado, clicando-se posteriormente em *Connect*.

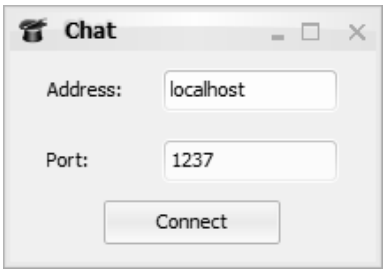


Figura 7.9: Chat - Tela inicial

Assim que a conexão ao **Ponto de Acesso** for estabelecida e um identificador for adquirido pelo **Ponto-Fim**, o aplicativo buscará por uma instância já publicada do serviço de bate-papo. Ao localizar essa instância, será iniciada uma conexão ao serviço e, então, será exibida a tela apresentada na Figura 7.10 para que o usuário informe seu apelido no bate-papo. Assim que o apelido for confirmado, é enviada uma requisição de início de acesso ao serviço, na qual é informado o apelido do usuário.

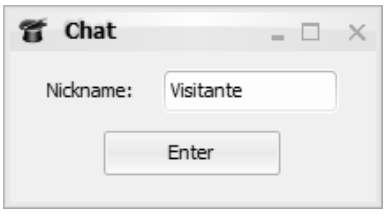
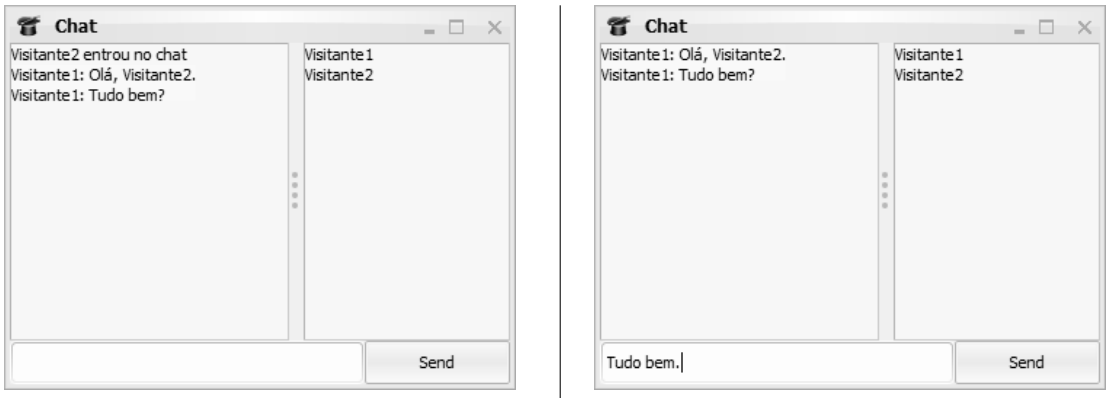


Figura 7.10: Chat - Entrar

Uma vez iniciado o acesso, o serviço executa no dispositivo um

comando de preenchimento da lista de contatos e, então, o dispositivo pode executar no serviço comandos de envio de mensagem, que serão replicados a todos os demais dispositivos conectados. Pode-se observar na Figura 7.11 a troca de mensagens ocorrida entre duas instâncias do aplicativo em questão.



(a) Instância 1 (b) Instância 2

Figura 7.11: Chat - Troca de mensagens

Neste capítulo foram apresentados exemplos de utilização da plataforma que foram criados com o objetivo de demonstrar o desenvolvimento de aplicativos que fazem uso do sistema para fornecimento e consumo de serviços. No próximo capítulo serão descritos os testes realizados a fim de avaliar a estabilidade e o desempenho da plataforma em condições possíveis durante sua utilização.

8 AVALIAÇÃO DE DESEMPENHO

De forma a avaliar a plataforma desenvolvida, foram executados testes com o sistema que tiveram como objetivo garantir sua estabilidade e verificar seu desempenho em situações de grande demanda. Buscou-se alcançar esses objetivos através da simulação de cenários possíveis durante sua utilização, desenvolvendo aplicativos capazes de efetuar múltiplas requisições "simultaneamente" e calcular o tempo decorrido entre a solicitação e a obtenção de resposta à operação. O processo principal dos aplicativos desenvolvidos para teste executa as seguintes operações:

1. Cria e inicia um determinado número de *threads* (definido através de argumentos de execução);
2. Aguarda preparação de todas as *threads*;
3. Dispara a execução de todas as *threads*;
4. Aguarda a conclusão da execução de todas as *threads*;
5. Armazena o tempo de execução em milissegundos de todas as *threads* em uma linha do arquivo de resultados;
6. Finaliza todas as *threads*, terminando os **Pontos-Fim**.

As *threads* iniciadas pelo processo principal são responsáveis pela execução e cálculo da duração de uma solicitação na plataforma. Essas *threads* executam as seguintes operações:

1. Prepara a execução (inicializa o **Ponto-Fim** e, se for o caso, conecta à plataforma e adquire um identificador);
2. Aguarda sincronização com as demais *threads*, garantindo que todos os
   Pontos-Fim estejam prontos para iniciar a execução com

concorrência máxima;
3. Armazena o tempo atual do sistema para posterior cálculo da duração;
4. Executa a operação à qual se destina;
5. Calcula o tempo decorrido desde o início da execução da operação até o momento atual.

Conhecidamente, a simulação utilizando um processo e múltiplas *threads* não é fiel a um cenário real, uma vez que todo o tráfego é direcionado a uma única interface de rede por um único microcomputador, podendo portanto ocorrer falhas que não ocorreriam em outro cenário, e sendo evitados outros tipos de falha que ocorreriam em uma rede mais ampla. Porém, devido à limitação de recursos para a simulação e buscando um maior controle sobre o cenário de testes optou-se pela utilização dessa estratégia.

A fim de evitar que eventos esporádicos gerassem resultados inconsistentes, cada teste foi executado múltiplas vezes e todos os resultados obtidos foram utilizados na geração das estatísticas apresentadas. Os dados foram estatisticamente analisados e apresentados utilizando gráficos que permitem a observação da tendência do tempo de execução em função do número de **Pontos-Fim** concorrentes, além de permitirem a percepção da variação existente no tempo de atendimento a solicitações conforme essa quantidade aumenta.

O ambiente no qual os testes foram executados (Figura 8.1) é composto por dois microcomputadores interconectados através de uma rede *Ethernet* 100Mbps. O primeiro, denominado **PC1**, foi utilizado para execução do **Roteador** e do **Ponto de Acesso** para o protocolo **TCP/IP!**, enquanto o segundo, denominado **PC2**, foi responsável pela execução dos aplicativos de teste e, portanto, dos **Pontos-Fim** para o protocolo **TCP/IP!**.

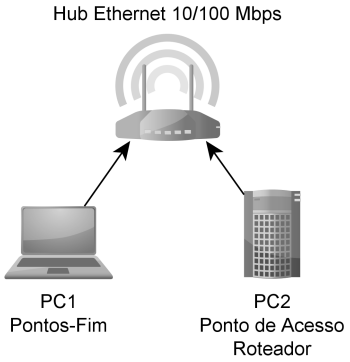


Figura 8.1: Ambiente de teste

Na Tabela 8.1 são apresentadas as características técnicas dos microcomputadores utilizados para teste. Observa-se nessa tabela que o sistema *multi-core* com maior capacidade de memória **RAM!** (**RAM!**) foi utilizado para execução dos **Pontos-Fim**, buscando uma maior precisão nos cálculos de tempo, graças ao maior paralelismo alcançado nesse sistema, e uma maior velocidade nas solicitações à plataforma. Além disso, a utilização do sistema *single-core* de menor memória **RAM!** para execução do **Roteador** e do **Ponto de Acesso** torna o cenário pessimista, de forma que os resultados obtidos representam o desempenho da plataforma em um ambiente não otimizado, ou seja, que pode ser melhorado com a utilização de equipamentos superiores. As seções a seguir apresentarão detalhes sobre cada teste executado, bem como os resultados obtidos.

Tabela 8.1: Equipamento de teste

	PC1	**PC2**
Sistema operacional	Microsoft Windows Server 2008 6.0.6001 Service Pack 1 Build 6001	Microsoft Windows 7 6.1.7601 Service Pack 1 Build 7601
Arquitetura	x86	x64
Processador	Intel Pentium 4 @ 2666Mhz 1 núcleo, 1 processador lógico	Intel Core i7-3612QM @ 2101Mhz 4 núcleos, 8 processadores lógicos
Memória instalada	1,13GB	8,00GB
Memória disponível	551MB	6,30GB
Java	Java SE 6 1.6.0_29-b11 32 bits	Java SE 6 1.6.0_35-b10 32 bits

8.1 Teste de conexão e registro

O teste de conexão e registro tem como objetivo avaliar o desempenho do sistema durante uma grande demanda por conexão e registro à plataforma, ou seja, o teste em questão simula um alto número de solicitações concorrentes de conexão a

um **Ponto de Acesso** para o protocolo **TCP/IP!**, assim como a atribuição de um identificador aos **Pontos-Fim**. Os resultados deste teste podem ser utilizados para prever dificuldades de acesso e definir boas práticas na utilização da plataforma, podendo por exemplo levar à conclusão de que é preferível manter uma conexão persistente entre **Pontos-Fim** e **Pontos de Acesso** a estabelecer novas conexões para acessos esporádicos.

O teste em questão foi executado 150 vezes para cada um dos conjuntos (100 a 1000 **Pontos-Fim**), sendo que para cada execução foi armazenado o tempo decorrido entre a solicitação de conexão **TCP/IP!** e a atribuição de um identificador para cada **Ponto-Fim**. O gráfico apresentado na Figura 8.2 mostra o tempo médio de conexão e registro em função do número de **Pontos-Fim** utilizados, assim como o desvio padrão da amostragem efetuada. Através da análise do gráfico, é possível verificar que o tempo de conexão e registro possui um crescimento linear em relação ao aumento do número de **Pontos-Fim** que concorrem por uma conexão à plataforma. Da mesma forma, o desvio padrão do tempo de atendimento a esse tipo de solicitação apresenta um crescimento linear, ou seja, com o aumento do número de **Pontos-Fim**, o tempo de resposta e a variação desse tempo crescem proporcionalmente. Sendo assim, pode-se concluir que o aumento do número de solicitações de conexão concorrentes não tende a tornar a plataforma inoperante ou causar a recusa por novas conexões. A tendência da plataforma nessas situações é tornar o atendimento a todas as solicitações mais lento, mantendo porém a estabilidade do sistema.

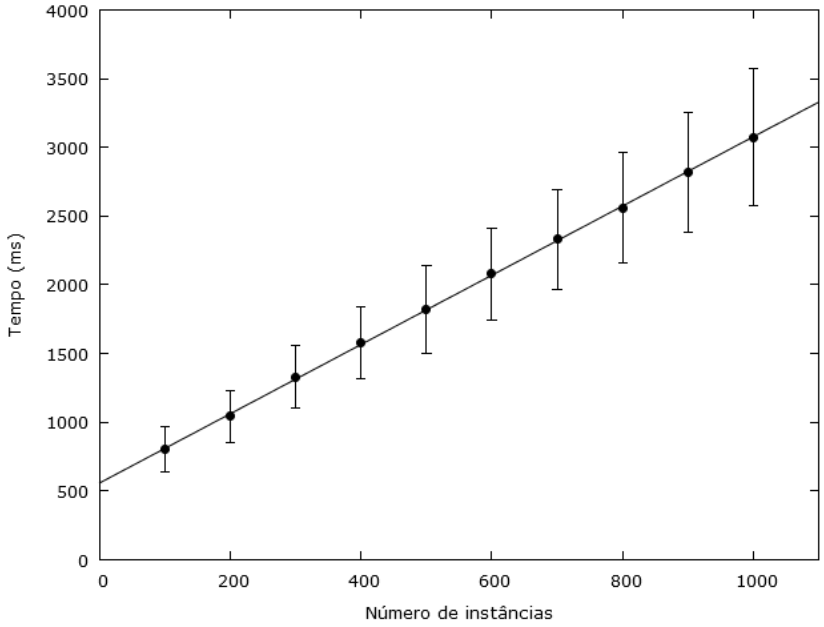


Figura 8.2: Teste de conexão e registro

8.2 Testes de publicação e busca de serviço

Com o objetivo de analisar o desempenho da plataforma durante uma grande demanda pela publicação e busca de serviços, foram desenvolvidos testes que simulam a solicitação concorrente dessas operações. Esses testes são necessários pois as informações de publicação de serviços são, na implementação atual do sistema, armazenadas e manipuladas exclusivamente pelo **Roteador**, sendo que cada operação executada sobre elas é sincronizada, ou seja, apenas uma operação de consulta ou alteração pode ser executada em um determinado momento, o que acaba por tornar esse um possível ponto de "gargalo" do sistema. Os resultados obtidos serão utilizados para avaliar a necessidade de melhorias no mecanismo de tratamento de solicitações de publicação e busca de serviços, por exemplo através do desenvolvimento de estratégias que permitam a realização de múltiplas buscas simultâneas, sendo essas postergadas apenas quando uma publicação estiver em progresso.

Novamente os testes foram executados 150 vezes para cada um dos conjuntos (100 a 1000 **Pontos-Fim**), sendo armazenado o tempo decorrido entre o início e o fim da execução de cada operação. Os gráficos das Figuras 8.3 e 8.4 apresentam o tempo médio de resposta em função do número de **Pontos-Fim** para a

publicação de um serviço e a busca de um serviço, respectivamente. Analisando os gráficos é possível verificar que o tempo de resposta, assim como o desvio padrão, possuem um crescimento linear em relação ao aumento do número de **Pontos-Fim** que realizam publicações e buscas, o que indica que a forma atualmente utilizada de controle de acesso às informações em questão, apesar de ser passível de otimização através da estratégia anteriormente proposta, não representa ameaça à estabilidade ou ao desempenho do sistema.

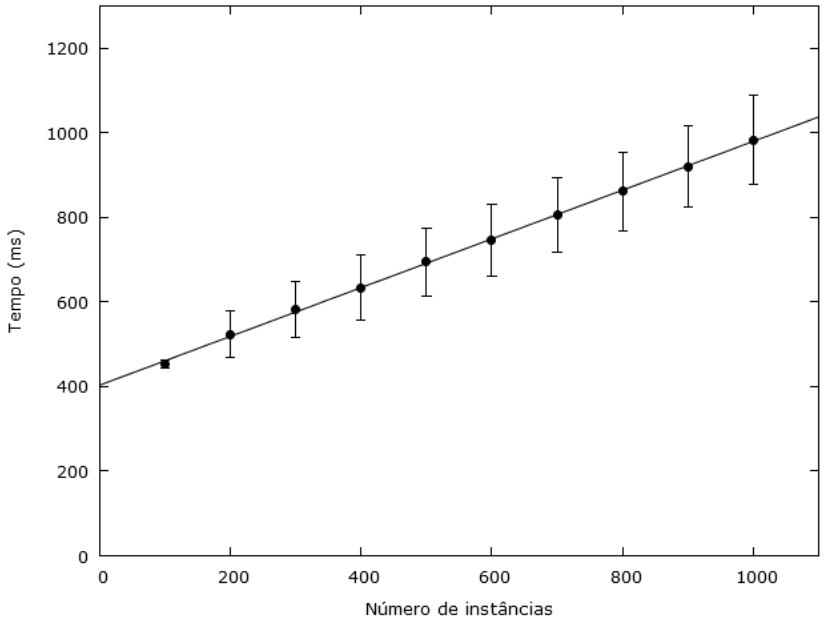


Figura 8.3: Teste de publicação de serviço

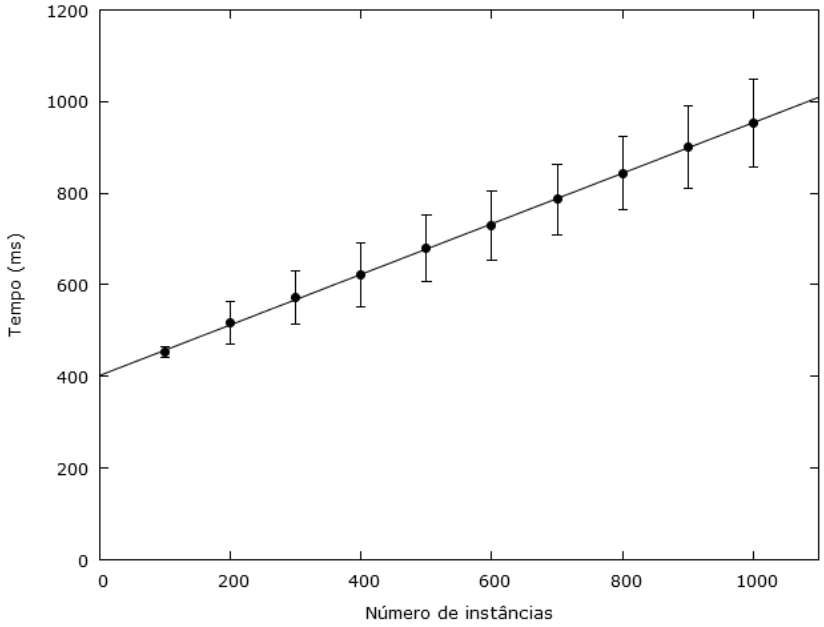


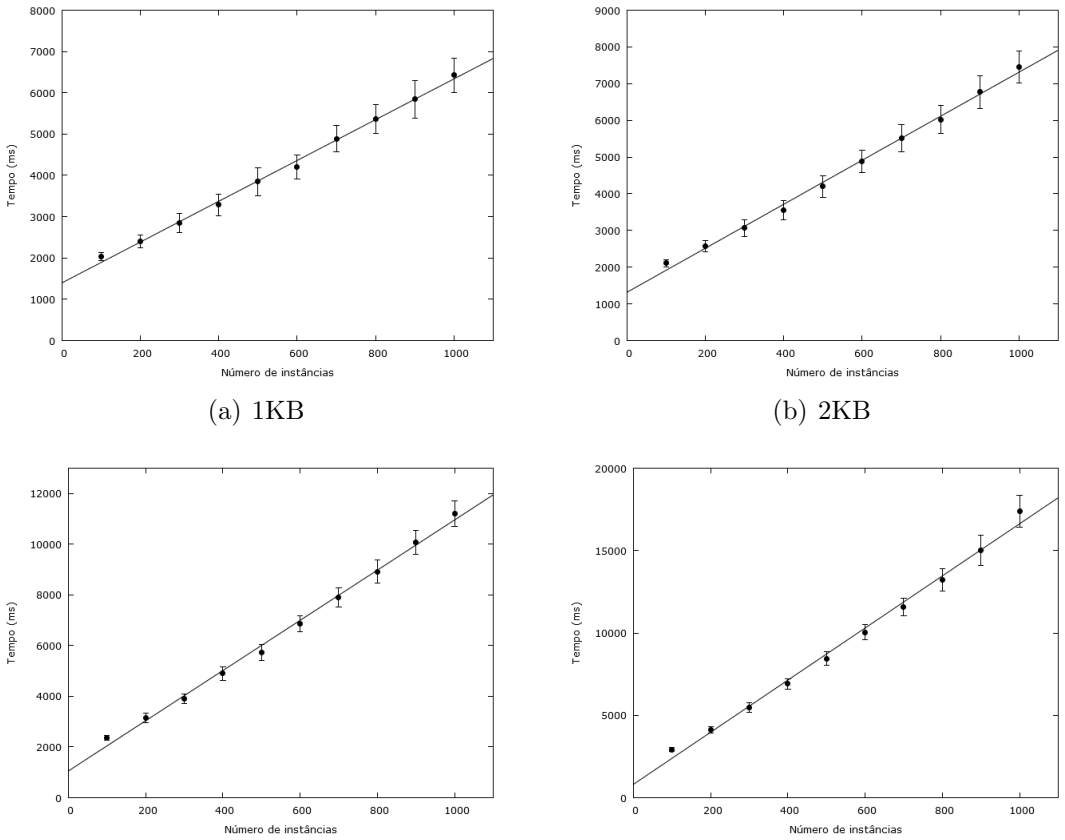
Figura 8.4: Teste de busca de serviço

8.3 Teste de acesso a serviço

A transmissão de pacotes de dados pode ser considerada a mais importante operação da plataforma, uma vez que todas as demais atuam como ferramentas que facilitam a sua realização. Com o objetivo de analisar o comportamento do sistema quando esse estiver sob grande demanda pela transmissão de dados, foi realizado um teste onde um **Ponto-Fim** efetua o estabelecimento de uma conexão e a troca de um pacote de dados com o serviço de *echo* previamente publicado por ele mesmo. Esse teste funciona de forma semelhante ao chamado *loopback* realizado através do protocolo **TCP/IP!**, no qual um aplicativo conecta-se ao próprio endereço **IP!**, ou seja, a um serviço local. Vale ressaltar que, apesar de o serviço ser publicado pelo mesmo **Ponto-Fim** que o acessa, os dados enviados trafegam através do **Ponto de Acesso** e do **Roteador** (conforme descrito no Capítulo 3). A análise dos resultados obtidos permitirá avaliar a necessidade de melhorias no mecanismo de transmissão de dados, podendo por exemplo indicar que pacotes trocados entre **Pontos-Fim** conectados ao mesmo **Ponto de Acesso** devem ser encaminhados pelo próprio **Ponto de Acesso** ao invés de serem enviados ao **Roteador** para posterior encaminhamento.

Os gráficos apresentados na Figura 8.5 foram gerados por 150 execuções do teste em questão para pacotes de tamanho 1KB, 2KB, 5KB e 10KB respectivamente, sendo que em cada execução foram coletados dados de 10 cenários diferentes (100 a 1000 **Pontos-Fim**). Para cada **Ponto-Fim** foi calculado o tempo decorrido entre a solicitação de conexão ao serviço e a confirmação de finalização dessa conexão, durante a qual foi enviado um pacote contendo um vetor de *bytes* do tamanho determinado e esse foi respondido com o mesmo pacote pelo serviço de *echo*, totalizando um tráfego superior ao dobro do tamanho desse vetor, levando em conta ambas as direções e a existência de dados de controle no pacote transmitido. Destaca-se que os processos de conexão, envio de dados e de desconexão, conforme descrito no Capítulo 4, consistem na troca de dois pacotes: um de solicitação e um de confirmação, o que leva a concluir que no teste em questão o tempo calculado se refere à troca de oito pacotes, que são:

1. Pacote de solicitação de conexão;
2. Pacote de confirmação de conexão;
3. Pacote de dados, enviado ao serviço;
4. Pacote de confirmação de entrega do pacote, enviado ao cliente;
5. Pacote de dados, enviado ao cliente;
6. Pacote de confirmação de entrega do pacote, enviado ao serviço;
7. Pacote de solicitação de desconexão;
8. Pacote de confirmação de desconexão.



(a) 1KB

(b) 2KB

(c) 5KB

(d) 10KB

Figura 8.5: Teste de acesso a serviço

Analisando os gráficos da Figura 8.5, percebe-se que o crescimento do tempo de resposta da plataforma em função do aumento do número de **Pontos-Fim**, mesmo com o crescimento do tamanho dos pacotes transmitidos, apresenta comportamento linear.

Levando em conta que a transmissão de pacotes de dados relativamente grandes faz parte da proposta da plataforma, torna-se necessária a análise do comportamento do sistema em função do crescimento desses pacotes. Com esse objetivo, foi desenvolvido o gráfico apresentado na Figura 8.6, que apresenta o tempo médio de transmissão de um pacote em função do tamanho dos dados nele contidos, assim como seu desvio padrão. Para coleta dessas informações, fixou-se o número de **Pontos-Fim** concorrentes em 100 e variou-se o tamanho do pacote de dados por eles transmitidos de 1KB a 20KB, executando-se esse procedimento 150 vezes. A análise da figura em questão aponta, novamente, um crescimento linear do tempo necessário para transmissão de pacotes de dados em função do

aumento do tamanho desses pacotes.

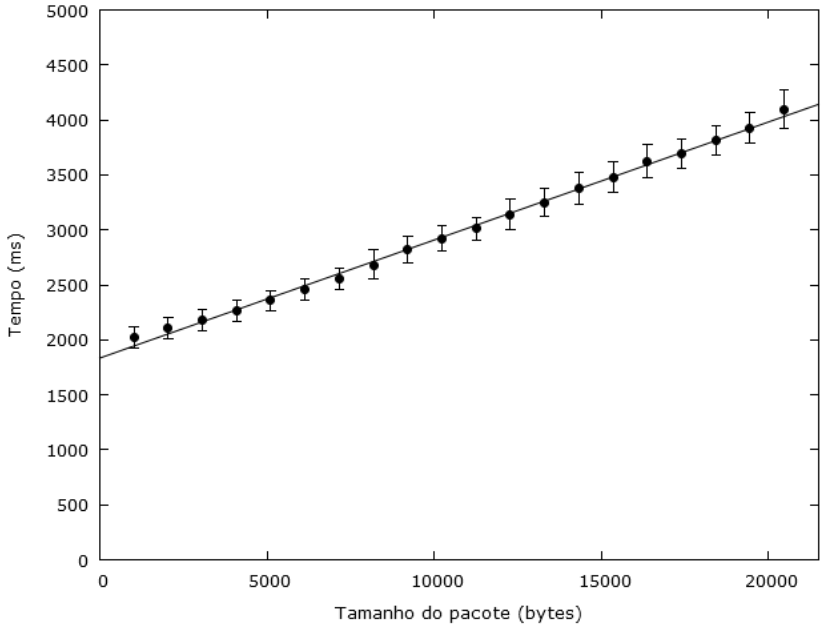


Figura 8.6: Teste de acesso variando tamanho do pacote

Buscou-se com os testes realizados verificar que o tempo de resposta da plataforma, em função do número de **Pontos-Fim** a ela conectados e do volume de dados transmitidos, não assume um crescimento elevado (quadrático ou exponencial, por exemplo), o que poderia comprometer sua utilização sob grande demanda. Com base nos gráficos apresentados neste capítulo e observando as retas ajustadas sobre seus pontos, é possível verificar que esse comportamento apresenta-se linear em todas as análises efetuadas, o que indica que o tempo de resposta do sistema é aceitável, já que cresce proporcionalmente em relação ao número de **Pontos-Fim** conectados e do tamanho dos pacotes transmitidos.

9 CONSIDERAÇÕES PARCIAIS

COLOCAR CONSIDERAÇÕES PARCIAIS

REFERÊNCIAS

ARCINIEGAS, F. **C++ XML**. São Paulo: Pearson Education do Brasil Ltda., 2002. Tradução Flavia Barktevicius Cruz. ISBN 8534614180.

BETTSTETTER, C.; RENNER, C. **A Comparison of Service Discovery Protocols and Implementation of the Service Location Protocol**. Twente, Netherlands: [s.n.], 2000.

CARISSIMI, A. S. **Redes de computadores**. Porto Alegre: Bookman, 2009. n.20. (Séries Livros Didáticos). 391 p. ISBN 9788577804962.

COMER, D. E. **Redes de computadores e internet**: abrange transmissão de dados, ligação inter-redes, web e aplicações. 4.ed. Porto Alegre: Bookman, 2007. ISBN 9788560031368.

COMVERSE. **Where is the M2M Revolution Taking Us?**: bss requirements for coping with m2m growth. [S.l.]: Comverse Incorporation, 2010. <Disponível em: `http://www.comverse.com/landing/comtalk01.2011/files/bss_requirements_for_m2m.pdf`>. Acesso em: 6 de Março de 2012.

DEITEL, P. J.; DEITEL, H. M.; CAETANO, C. A. C. **C: como programar**. 6.ed. atual. São Paulo: Pearson, 2011. xxvii, 818 p. ISBN 9788576059349.

DIERKS, T.; RESCORLA, E. **The Transport Layer Security (TLS) Protocol Version 1.2**. [S.l.]: IETF, 2008. n.5246. (Request for Comments). Updated by RFCs 5746, 5878, 6176.

DONAHOO, M. J.; CALVERT, K. L. **TCP/IP sockets in C**: practical guide for programmers. San Francisco: Morgan

Kaufmann Publishers, 2001. (The Morgan Kaufmann practical guides series). ISBN 9781558608269.

EISLER, M. **XDR: external data representation standard**. [S.l.]: IETF, 2006. n.4506. (Request for Comments).

FREIER, A.; KARLTON, P.; KOCHER, P. **The Secure Sockets Layer (SSL) Protocol Version 3.0**. [S.l.]: IETF, 2011. n.6101. (Request for Comments).

GOOGLE. **Android**. [S.l.: s.n.], 2012. <Disponível em: `http://www.android.com/`>. Acesso em: 8 de Outubro de 2012.

INTEL. **Connecting the Dots in M2M Connectivity**. [S.l.]: Intel Corporation, 2011. <Disponível em: `http://www.edn.com/file/26127-Connecting_the_Dots_in_M2M_Connectivity_PDF.pdf`>. Acesso em: 6 de Março de 2012.

KOWAL, G. **Launch4j**. [S.l.: s.n.], 2012. <Disponível em: `http://launch4j.sourceforge.net/`>. Acesso em: 8 de Outubro de 2012.

KUROSE, J. F.; ROSS, K. W. **Redes de computadores e a Internet**: uma abordagem top-down. 3.ed. São Paulo: Pearson Addison Wesley, 2006. ISBN 8588639181.

MICROSOFT. **Microsoft Windows**. [S.l.: s.n.], 2012. <Disponível em: `http://windows.microsoft.com/`>. Acesso em: 8 de Outubro de 2012.

MSDN. **MSDN Library**. [S.l.: s.n.], 2012. <Disponível em: `http://msdn.microsoft.com/library`>. Acesso em: 11 de Junho de 2012.

OMG. **Common Object Request Broker Architecture (CORBA) Specification, Version 3.2**. [S.l.: s.n.], 2011. <Disponível em: `http://www.omg.org/spec/index.htm`>. Acesso em: 1 de Junho de 2012.

ORACLE. **Java SE 6 Documentation**. [S.l.: s.n.], 2011. <Disponível em: `http://docs.oracle.com/javase/6/docs/`>. Acesso em: 7 de Março de 2012.

ORACLE. **Java SE Overview**. [S.l.: s.n.], 2012. <Disponível em: `http://www.oracle.com/javase/`>. Acesso em: 8 de Outubro de 2012.

PSINAPTIC. **Jini and Universal Plug and Play (UPnP) Notes**. [S.l.: s.n.], 2004. <Disponível em: `http://www.psinaptic.com/link_files/jini_and_plugandplay.pdf`>. Acesso em: 5 de Junho de 2012.

SAVITCH, W. J. **C++ absoluto**. São Paulo: Addison Wesley, 2004. Tradução Claudia Martins. ISBN 8588639092.

TANENBAUM, A. S. **Redes de computadores**. São Paulo: Pearson Prentice Hall, 2011. ISBN 9788576059240.

THURLOW, R. **RPC: remote procedure call protocol specification version 2**. [S.l.]: IETF, 2009. n.5531. (Request for Comments).