Estratégias de Comunicação em um Sistema Distribuído: Estudo de Caso de um Sistema de Gerenciamento de Vagas

Pedro Henrique de Sousa Brito[1], **Dênis Taironi Leorne da Cunha**[1]
Emanuel Ferreira Coutinho[1]

[1] Instituto Universidade Virtual (UFC Virtual)
Universidade Federal do Ceará (UFC) – Fortaleza – CE – Brasil

p.1392@hotmail.com,denis.taironi@gmail.com,emanuel@virtual.ufc.br

Abstract. *The aim of this study was to develop a strategy for integration and communication among components of a parking space's management system. These components involve sensors present in parking spaces, servers, database and client applications. The methodology consisted of: developing a motion sensors network controlled by Arduino to represent the parking lot, create a server responsible for collecting the data obtained by the sensors, develop a database to store information, create a server application to manage customer's orders and the database information access, and develop prototype applications using this data to validate the communication strategies. The obtained results were promising. The solution has proved simple and rapid maintenance. The flow of information between the sensors and the final applications were constant and the servers played a role in keeping these data updated.*

Resumo. *O objetivo deste trabalho foi desenvolver uma estratégia de integração e comunicação entre os componentes de um sistema de gerenciamento de vagas de um estacionamento. Estes componentes envolvem sensores presentes nas vagas, servidores, banco de dados e aplicações clientes. A metodologia consistiu em: desenvolver uma rede de sensores de presença comandados por um Arduíno para representar o estacionamento, criar um servidor responsável por coletar os dados obtidos pelos sensores, desenvolver um banco de dados para guardar informações, criar uma aplicação servidora para gerenciar os pedidos dos clientes e o acesso as informações do banco e desenvolver aplicações protótipos utilizando esses dados para validar as estratégias de comunicação. Os resultados obtidos foram promissores. A solução se mostrou simples e de rápida manutenção. O fluxo de informação entre os sensores e as aplicações finais foram constantes e os servidores desempenharam seu papel em manter estes dados atualizados.*

1. Introdução

Atualmente existem milhões de dispositivos, como *smartphones*, PCs, notebooks e *smartwatches* e microcontroladores conectados à internet trocando dados a todo o momento. Essa facilidade de troca de informações atingiu diversas tarefas do cotidiano com o intuito de lhes adicionar funcionalidades, como aplicativos de corrida que enviam a distância e tempo do exercício para as redes sociais, como o RunKepper [Runkeeper 2016].

Integrar essas atividades à rede, de maneira a difundir suas informações com mais pessoas e aumentar suas funcionalidades, possibilita que elas se tornem melhores e maiores. Sistemas dessa natureza são considerados sistemas distribuídos: "sistemas no qual os componentes de hardware ou software localizados em computadores interligados em rede se comunicam e coordenam suas ações apenas enviando mensagens entre si" [Coulouris et al. 2013].

Além disso, existe a possibilidade da compra de ingressos de cinemas pela internet, com a facilidade de visualizar os assentos disponíveis na sessão e reservá-los previamente. Por exemplo o site *ingresso.com* [Ingresso.com 2016]. Existem também aplicativos de GPS que utilizam a velocidade média dos usuários para determinar a situação do tráfego de veículos, como o Waze [Waze 2016].

Sob este ponto de vista, ao se reparar em um estacionamento, serviço usado diariamente por muitos, várias oportunidades de melhoria não são tomadas. Por que não visualizar a ocupação dos setores por meio dos dispositivos do usuário? Por que não há uma opção de notificação caso ocorra algum incidente com o carro no estacionamento? Por que a máquina de tíquete na entrada do estacionamento não pode sugerir um setor mais vago para ajudar o motorista a estacionar? Por que todas as informações estão presas dentro do ambiente e são apenas transmitidas por lâmpadas verdes e vermelhas?

Registrar as informações dos sensores no estacionamento e transmiti-las para os clientes podem se tornar um problema. Os sensores, os servidores e os aplicativos finais precisam se comunicar com perfeição, independente dos sistemas operacionais e dos dispositivos em que executam.

Para resolver esse problema é necessário criar estratégias para integrar todas estas partes diferentes do sistema, desde a comunicação dos sensores com os servidores da aplicação e até comunicação do servidor com os diversos dispositivos finais dos usuários. Estas estratégias necessitam ser simples e que possam ser implementadas nos sistemas já existentes, para que ser tornem atraentes para aplicações reais.

O objetivo principal do trabalho é proporcionar uma estratégia de integração entre os sensores embarcados, os servidores e a aplicação móvel. Os objetivos secundários são desenvolver uma forma de registrar as informações dos sensores, criar um protocolo de comunicação entre as partes do sistema, criar um sistema de comunicação multiplataforma, permitindo adicionar novas partes ao sistema com maior facilidade e desenvolver um protótipo de cliente para validar a arquitetura de comunicação do sistema.

A metodologia do trabalho é constituída pelo desenvolvimento de um protocolo de comunicação, construção de uma rede de sensores de presença e um Arduíno para representar o estacionamento, desenvolver um coletor de dados para recolher estas informações, construir um banco de dados para guardar estes dados, elaborar um aplicação servidora, na qual os clientes conversariam para acessar os dados e por fim, construir um protótipo de cliente para validar a estratégia de comunicação.

A contribuição do trabalho está na demonstração de uma arquitetura simples, de fácil manutenção e que utilize características básicas da linguagem para integrar diferentes dispositivos, expondo assim, as possibilidades que os sistemas de estacionamento comuns teriam caso fossem conectados à rede e combinados com diversos outros dispositivos.

Neste trabalho será demonstrado como foi construído um sistema que atendesse aos objetivos propostos. Na segunda seção serão apresentados os componentes e a forma na qual eles foram utilizados na construção do sistema, desde sua parte física, com sensores e o Arduíno, até o servidor e aplicativos clientes. A terceira seção descreverá especificamente as estratégias tomadas para integrar as diferentes partes do sistema e garantir a comunicação e fluxo de informações. Por fim, na quarta seção serão apresentados as conclusões e trabalhos futuros.

2. Sistema de Gerenciamento de Vagas em Estacionamento

O sistema de gerenciamento de vagas de estacionamento oferece a funcionalidade de checagem do estado atual das vagas do estacionamento de um estabelecimento. Ele é dotado de sensores que monitoram constantemente as vagas e informam a um servidor o estado no qual se encontram (ocupadas ou livres). Essas informações são armazenadas em um banco de dados para que posteriormente sejam utilizadas por aplicações clientes. A Figura 1 exibe uma visão da arquitetura do sistema, ressaltando as camadas que compõem o sistema assim como algumas tecnologias e ferramentas utilizadas na construção do mesmo.

2.1. Componente Embarcado

Para Barros e Cavalcante (2013), "um sistema embarcado ou embutido (*embedded system*) pode ser definido como um sistema computacional especializado que faz parte de uma máquina ou sistema maior" [Barros e Cavalcante 2016]. Sob essa luz, percebe-se recursos computacionais embarcados em diversos sistemas, como em sistemas de automóveis sensíveis à presença do motorista, em sistemas de segurança que monitoram

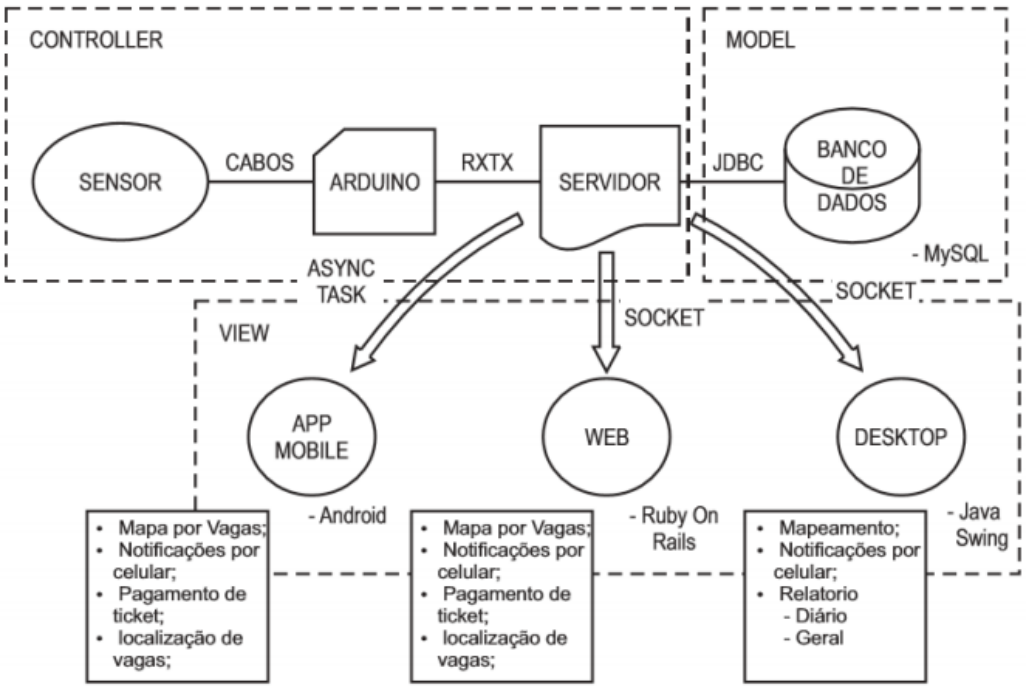

Figura 1. Arquitetura de aplicação

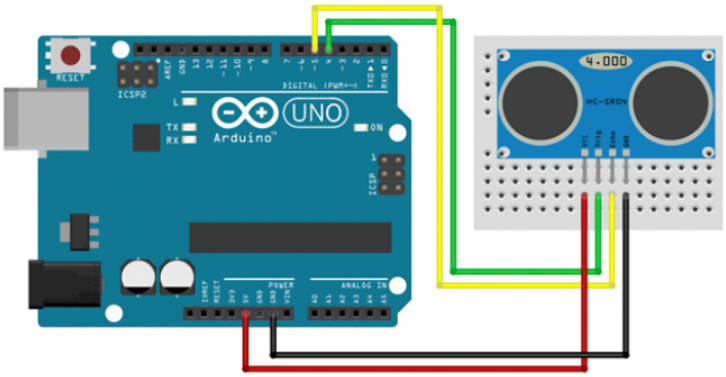

Figura 2. Esquema de montagem dos componentes embarcados

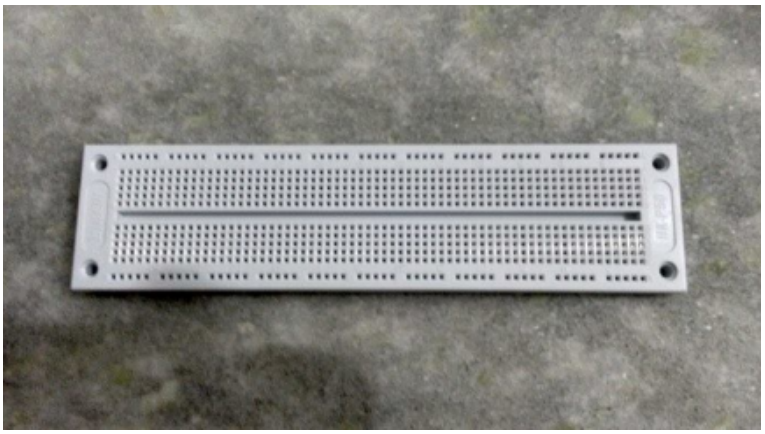

Figura 3. *Protoboard*

usuários em bancos e aeroportos, ou ainda em plataformas de videogames que contam com sensores de movimento para gerar interatividade entre o usuário e o jogo.

Com o amadurecimento de conceitos, como o de computação pervasiva, que propõem um mascaramento do sistema computacional em relação aos usuários, componentes como sensores, microcontroladores e microprocessadores são recursos indispensáveis no desenvolvimento desses tipos de sistemas. Isto se dá pelo fato de serem componentes em sua maioria pequenos e de boa adaptabilidade física, baixo custo, possuírem finalidades especificas e com grande poder de integração a outros sistemas.

Para um sistema de gerenciamento de vagas de estacionamento, cuja principal funcionalidade é o constante monitoramento do estado das vagas, contar com sensores de presença acoplados a um microprocessador se faz indispensável na realização desta tarefa. Assim, o sistema conta com sensores ultrassônicos de distância modelo HC–SR04 que por meio de uma simples *protoboard* (Figura 3), que se trata de uma placa de ensaio com furos e conexões condutoras para montagem de circuitos elétricos experimentais, e *jumpers* (Figura 4), que são pequenos condutores utilizados para conectar dois pontos de um circuito eletrônico, se conectam a um microprocessador Arduíno. A Figura 2 mostra um esquema da montagem destes componentes.

Figura 4. *Jumpers*

2.1.1. Sensores

Para a checagem das vagas é utilizado o HC–SR04, um sensor ultrassônico de distância. Um modelo simples de ultrassom, que é capaz de medir distâncias de 2cm a 4m com uma precisão de 3mm. Este sensor possui um circuito pronto com emissor e receptor acoplados e quatro pinos, o VCC, Trigger, ECHO e GND. O pino VCC é a alimentação do sensor, que deve ser 5V, o Trigger é o gatilho de emissão do pulso ultrassônico, o ECHO é o que ouve o retorno do pulso e o GND é o *ground*. Para começar a medição é necessário alimentar o sensor e colocar o pino Trigger em nível alto por mais de 10us (microsegundos). Assim o sensor emitirá uma onda sonora que ao encontrar um obstáculo rebaterá de volta em direção ao módulo, sendo que neste tempo de emissão e recebimento do sinal o pino ECHO ficará em nível alto. Logo o cálculo da distância pode ser feito de acordo com o tempo em que o pino ECHO permaneceu em nível alto após o pino Trigger ter sido colocado em nível alto.

$$\text{Distância} = \frac{\text{Tempo ECHO em nível alto} \times \text{Velocidade do Som}}{2} \quad (1)$$

A velocidade do som é considerada como sendo igual a 340 m/s, logo o resultado é obtido em metros se considerado o tempo em segundos. Na fórmula (Equação 1), a divisão por 2 deve-se ao fato que a onda é enviada e rebatida, logo ela percorre 2 vezes a distância procurada. A Figura 5 exibe o sensor utilizado na aplicação.

2.1.2. Arduíno

Como microcontrolador da parte embarcada do sistema, utilizou-se a placa Arduíno (Figura 6), uma plataforma de código aberto desenvolvida em 2005 pelo italiano Massimo Banzi e colaboradores, para o auxílio nos estudos de eletrônica. As principais vantagens no uso desta plataforma são seu baixo custo e por ser concebido de forma *open-source*, possibilitando uma grande contribuição da comunidade no desenvolvimento da ferramenta.

Basicamente o Arduíno se divide em componentes de hardware e software. O

Figura 5. Sensor de distância ultrassônico HC–SR04

Figura 6. Placa Arduíno UNO

hardware é composto por uma placa de prototipagem na qual são construídos os projetos. Enquanto o software é uma IDE (*Integrated Development Environment* ou Ambiente de Desenvolvimento Integrado), que é executada em um computador onde é realizada a programação, conhecida como *sketch*, na qual será feita o *upload* para a placa de prototipagem Arduíno através de uma comunicação serial. O *sketch* feito pelo projetista dirá à placa o que deve ser executado durante o seu funcionamento.

Conforme visto na Figura 6, a placa Arduíno UNO possui diversos conectores que servem para interface com o mundo externo, não sendo o foco deste trabalho o aprofundamento desses componentes. Porém, é importante destacar alguns pontos cruciais para o entendimento do projeto. A placa conta com 14 pinos de entrada e saída digital (pinos postos de 0 a 13), esses pinos podem ser utilizados como entradas ou saídas digitais de acordo com a necessidade do projeto e conforme foi definido no *sketch* criado na IDE. Possui também 6 pinos de entradas analógicas (pinos postos de A0 a A5), dedicados a receber valores analógicos, por exemplo, a tensão do sensor. O valor a ser lido deve estar na faixa de 0 a 5V onde serão convertidos para valores entre 0 e 1023 bits. Por fim, tem-se

Figura 7. Entradas e saídas da placa Arduíno UNO

os pinos de 5V que fornece tensão de 5V para alimentação de *shields* e circuitos externos e o pino GND referência de terra. A alimentação da placa pode ser feita a partir da porta USB do computador ou através de um adaptador AC. Para o adaptador AC recomenda-se uma tensão de 9V a 12V. A Figura 7 exibe as entradas e saídas presentes na placa.

O software para programação do Arduíno é uma IDE multiplataforma que permite a criação de *sketchs* para a placa. A linguagem de programação da ferramenta é modelada a partir da linguagem Wiring e Processing. Quando pressionado o botão *upload* da IDE, o código escrito é traduzido para a linguagem C e é transmitido para o compilador avr-gcc, que realiza a tradução dos comandos para uma linguagem que pode ser compreendida pelo microcontrolador. A IDE do Arduíno é uma interface de programação simples com poucos recursos, mas que oferece um bom poder de abstração aos desenvolvedores do ambiente. A ferramenta possui uma linguagem própria baseada na linguagem C e C++. Importante destacar que após a conexão da placa com um computador, deve-se atribuir o modelo da placa que está sendo utilizada assim como uma porta serial COM. O ciclo de programação do Arduíno pode ser visto na Figura 8.

Para o sistema de gerenciamento de vagas de estacionamento foram utilizados dois sensores ultrassônicos HC–SR04 para simularem duas vagas de um setor de estacionamento. Esses sensores são conectados a um circuito como mostra a Figura 9 e alimentados pelo Arduíno através do pino de alimentação externa 5V. Para a utilização desses sensores nos *sketchs* construídos na IDE, é necessário a importação de uma biblioteca referente aos sensores, que no caso trata-se da **Ultrasonic.h**. Esta biblioteca fornece acesso aos recursos de ECHO e Trigger dos sensores, responsáveis pelo sensoriamento do ambiente, assim como o acesso à funções como o **Ranging**, responsável pela conversão do tempo de resposta do ECHO em centímetros.

No código da *sketch* existem duas funções essenciais: a função **setup** e a função **loop**. A função **setup**, executada na inicialização do programa, é responsável pelas

Figura 8. Ciclo de programação no Arduíno

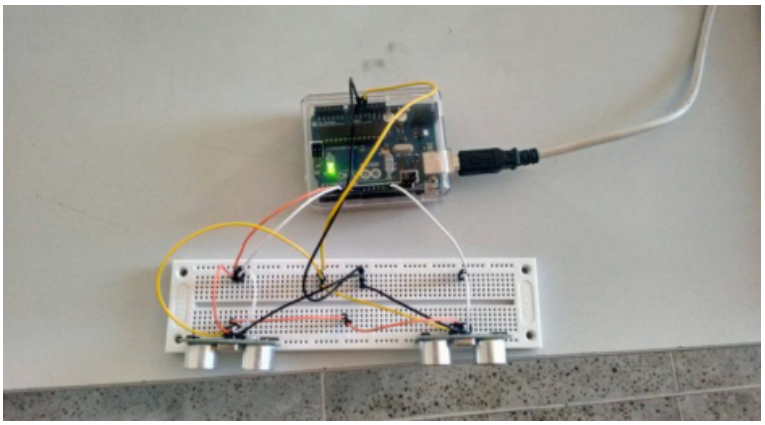

Figura 9. Componentes embarcados conectados em circuito

configurações iniciais do microcontrolador, tal como definição dos pinos de entrada e saída, inicialização da comunicação serial, entre outras. A função **loop** será onde ocorrerá o laço infinito da programação, ou seja, onde será inserido o código que será executado continuamente pelo microcontrolador. No projeto desenvolvido neste trabalho, os sensores foram conectados à placa nos pinos de entrada 4 e 13 e no pino de saída 3, simulando duas vagas de estacionamento de um setor com 2 vagas.

2.2. Servidor

Após o recolhimento dos dados do estacionamento, é necessário que este material seja coletado e salvo para uso futuro. O responsável por estas ações no sistema é um servidor, chamado aqui de Coletor de Dados. Ele terá todos os métodos necessários para realizar estas atividades. Também é necessário um servidor que gerencie o acesso aos dados do sistema, servindo de intermédio entre os clientes e a base de dados. Este será chamado de aplicação servidora.

2.2.1. Coletor de Dados

O coletor de dados é o primeiro servidor a entrar em ação no sistema. A primeira função do coletor é receber do Arduíno os dados do estado das vagas do estacionamento, enviado pela conexão serial do computador. Nos experimentos a serem descritos nas próximas seções utilizou-se também a linguagem de programação Java, e algumas de suas APIs (*Application Programming Interface* ou Interface de Programação de Aplicação). O Java não pode manipular o serial nativamente, por esta razão foi utilizado a **API RXTX**. Por meio dela é possível ter o controle dos dados na serial e apontá-los a uma variável no sistema.

As informações sobre o estacionamento são salvas pelo Arduíno em uma *string*, na qual o caractere 0 (zero) representa uma vaga vazia e o caractere 1 (um) representa uma vaga ocupada, para depois serem enviadas.

De posse destas informações novas sobre o estacionamento, é necessário atualizar vários dados presentes no banco de dados para que as aplicações finais tenham sempre os dados certos. Como o estado do estacionamento pode mudar constantemente com a entrada e saída de veículos, essa atualização deve ocorrer de forma sistemática.

O Arduíno pergunta o estado dos sensores a cada 15 segundos e envia estas informações ao servidor, que depois as guarda no banco de dados. Este intervalo de tempo foi escolhido para dar ao cliente que está estacionando o veículo tempo suficiente para manobrá-lo, sem gerar assim informações de vaga ocupada caso ele entre por alguns instantes na área de dos sensores.

Para atualizar o banco de dados utilizou-se uma API chamada **JDBC** (*Java Database Connectivity*), responsável por executar comandos SQL no banco. O método do sistema responsável por atualizar o banco é o **Atualizar()**, que tem como parâmetro de entrada duas *strings*. A primeira representa o estado mais atual das vagas, o que acabou de ser recebido do Arduíno, e a segunda o estado passado. No método, é comparada a *string* anterior com a nova para checar o que é necessário ser executado. Caso a vaga tenha se mantido vazia, nada é feito, se a vaga foi ocupada é necessário atualizar a informação do estado da vaga no banco, aumentar em 1 (um) o número de vezes que ela foi ocupada e aumentar o tempo que ocupação por 15 segundos, já que esse é o intervalo de tempo de checagem dos sensores. É realizado também o incremento de 1 (um) no número de vagas ocupadas no setor em que a vaga se encontra. Caso a vaga tenha se mantido ocupada, é feito apenas o acréscimo de tempo ocupação da vaga no banco de dados. Se a vaga foi desocupada, atualiza-se o estado da vaga no banco de dados para vazio e diminui-se o número de vagas ocupadas no seu setor.

2.2.2. Aplicação Servidora

Com todas as informações presentes no banco de dados e sendo atualizadas constantemente, é necessário haver um método para que os clientes possam usar estes dados. O responsável por isso é a aplicação servidora. As funções da aplicação servidora são gerenciar os pedidos dos clientes, requisitar do banco os dados necessários e enviar-lhes a resposta. Este servidor possui o maior número de métodos do sistema.

O primeiro passo é fazer que esse servidor possa ser acessado. Para isso foi construído um **ServerSocket** na porta 6667, criando assim um endereço constante em que ele possa ser alcançado pelos clientes.

Para poder entender de forma correta os pedidos, foi criado um protocolo que deve ser seguido pelo servidor e pelo cliente. Neste protocolo o cliente deve enviar o pedido em uma *string*. Nela estarão as informações necessárias em uma espécie de código. O primeiro caractere da *string* se refere ao método que o cliente deseja invocar no servidor para receber os dados necessários, e os outros caracteres, separados por vírgula, se referem aos parâmetros deste método, caso eles existam. Por exemplo, caso o cliente queira invocar o método que devolve o mapa do estado de vagas no estacionamento, **MapaEst()**, ele deve enviar na string 0 (zero), que é o código referente ao método e também o mais um número, que é um parâmetro do método que se refere a identificação do estacionamento do qual ele deseja o mapa. No caso do sistema o estacionamento exemplo tem como código o 1 (um). Ou seja, o cliente tem que enviar uma string "0,1".

O servidor deve, ao receber um pedido, tratá-lo para entender o que o cliente deseja. É utilizado um *split*, que quebra os caracteres da *string* entre as vírgulas. Um *split* é uma função em Java que quebra uma *string* em *strings* menores a partir de um caractere definido, nesse caso uma vírgula.

Para direcionar a execução da aplicação para o método correto utilizou-se *switch* e *case*. Nele, apenas certas partes do código são chamadas de acordo com o valor de um parâmetro. Nesse caso o código referente ao método, ou seja, o cliente envia "0,1", o *case* 0 (zero) é ativado e é chamado o método 0 (zero) que é o **MapaEst()**. Funciona da mesma forma para todos os outros métodos, mudando apenas o valor do código.

O método que podem ser chamadas além do **MapaEst()** são o **VagaEst()**, que devolve o estado de uma vaga apenas, o **Pagar()**, que paga um tíquete, necessitando do seu código de barras. Pode ser chamado o **OcupSetor()**, que devolve informações de ocupação dos setores, necessitando da identificação do setor. Por fim existe o **Rel()**, que devolve um relatório que conta com estatísticas sobre o uso do estacionamento e tem como parâmetro a identificação do estacionamento. As informações devolvidas são quantas vezes uma vaga foi ocupada, por quanto tempo e uma média de tempo por ocupação.

Cada método requer uma consulta ao banco de dados, que devolverá vários dados. A partir das informações recebidas, o servidor cria um objeto referente ao pedido, como uma vaga ou um setor, e por *socket*, as retorna ao cliente. Para enviar um objeto por *socket* é necessário chamar um **ObjectOutputStream()**, que quebra esse objeto em uma cadeia de bits para que se possam ser transmitidos pela rede.É preciso que a classe deste objeto permita essa quebra, ela sinaliza a permissão implementando a interface **Serializable**.

Podendo agora enviar a resposta, o servidor cria um *socket*, endereçado para o ip de origem de onde ele recebeu o pedido. A porta usada pelo cliente para receber as respostas é a 6668. Para que o cliente consiga traduzir de volta essa cadeia de string em objeto, ele precisa das mesmas classes presentes no servidor, que servirão com um modelo para a reconstrução.

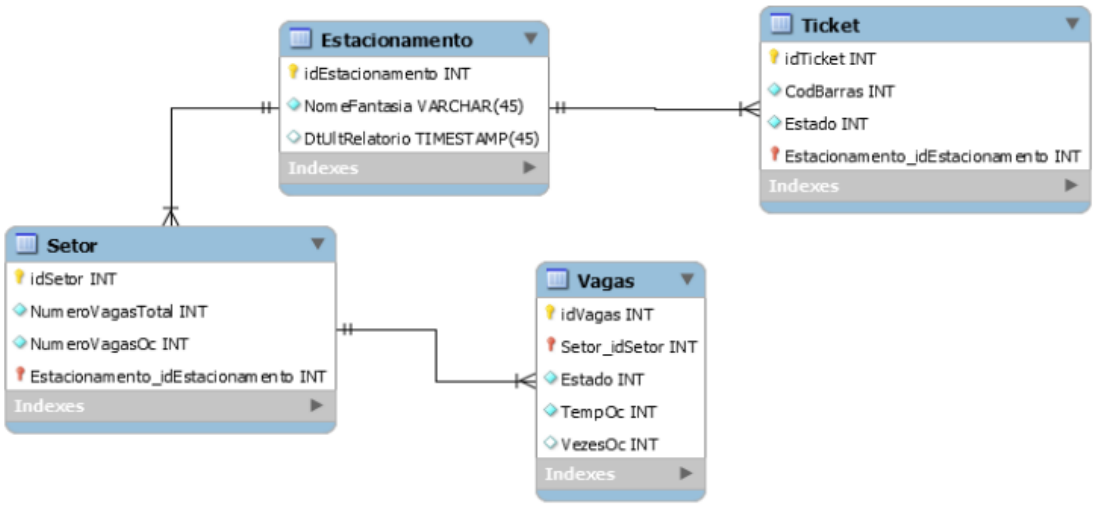

Figura 10. Modelo relacional do banco de dados

2.3. Banco de Dados

O sistema de gerenciamento de bancos de dados utilizado foi o MySQL Server 5.6. Para manipulação e construção do modelo relacional foi empregado o MySQL Workbench 5.6. A Figura 10 exibe o modelo relacional do banco de dados:

A tabela **Estacionamento** possui com parâmetros **idEstacionamento**, que é a chave primária. Uma chave primária é única e ela é responsável pela identificação de cada linha da tabela. A tabela também conta com **NomeFantasia**, que é um nome pelo qual o estacionamento é conhecido e também o **DtUltRelatorio**, que guarda a data do último relatório de estatísticas de ocupação emitido.

A tabela **Setor** possui **idSetor** como chave primária, **NumeroVagasTotal**, que guarda o total de vagas do setor e **NumeroVagasOc**, que guarda o número de vagas atualmente ocupadas. Na tabela também existe uma chave estrangeira **Estacionamento_idEstacionamento**, uma chave deste tipo é apresentada quando há um relacionamento entre tabelas, neste caso, um setor está relacionado a um estacionamento.

A tabela **Vagas** possui **idVaga** como chave primária, **Setor_idSsetor** como chave estrangeira que a relaciona com setor, **Estado**, que guarda o estado atual da vaga, com 0 (zero) para vazia e 1 (um) para ocupada, **TempoOc**, que guarda o tempo que aquela vaga ficou ocupada e **VezesOc**, que guarda quantas vezes aquela vaga foi ocupada.

A última tabela é a **Ticket**, que possui **idTicket** representando a chave primária, o **CodBarras**, que guarda o código de barras do tíquete, tem o Estado, que diz se o tíquete está pago (um) ou não (zero) e uma chave estrangeira, a **Estacionamento_idEstacionamento**, que o relaciona com Estacionamento.

2.4. Componente *Desktop*

Para a validação da arquitetura, uma aplicação em Java Swing que utiliza as funcionalidades presentes no sistema foi desenvolvida [1]. Swing é uma API Java para construção

[1] A aplicação em ação pode ser visualizada no link: https://www.youtube.com/watch?v=DyGl0jF5x8I

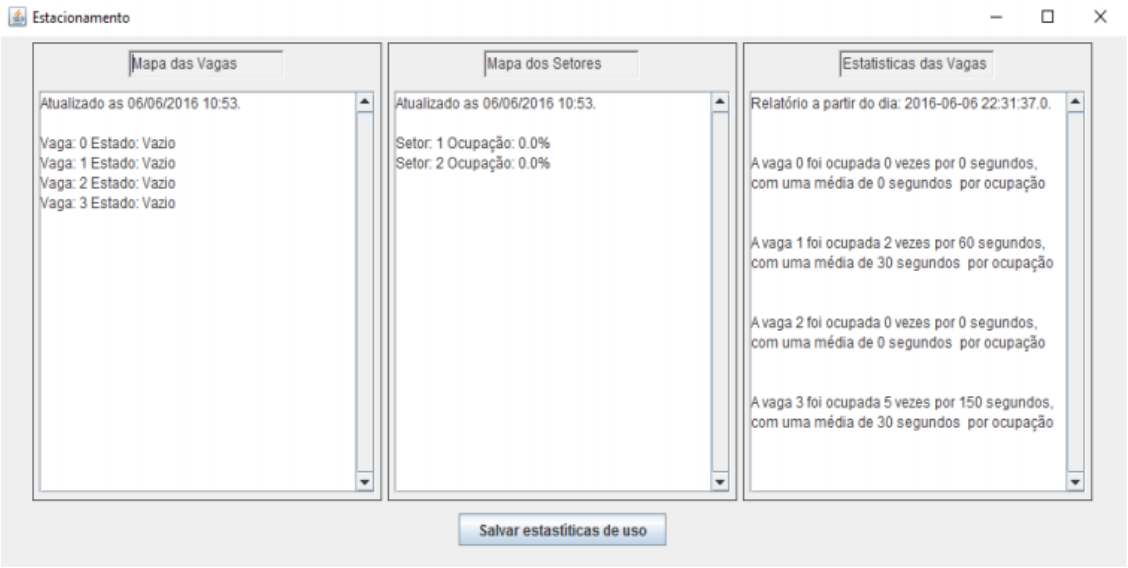

Figura 11. Tela da aplicação *desktop*

de interfaces gráficas. Conta com diversos componentes que o desenvolvedor pode optar por usar, como botões, menus e barras de rolagem. Além disso, seus componentes são multiplataforma, ou seja, a interface só precisa ser desenvolvida uma vez e se comportará da mesma maneira e terá uma aparência semelhante em qualquer sistema operacional. A interface é bem simples. Conta com três painéis, um para o mapa do estacionamento, outro para a ocupação dos setores e um para as estatísticas do estacionamento. Abaixo dos painéis há um botão para salvar em arquivo estas estatísticas. A Figura 11 mostra a tela da aplicação.

As funcionalidades presentes são a de visualização do estado das vagas do estacionamento, da ocupação dos setores e das estatísticas de ocupação das vagas, que são quantas vezes a vaga foi ocupada, por quanto tempo e uma média de tempo por ocupação, além de gerar um documento com as estatísticas de ocupação do estacionamento.

Estas informações são automaticamente atualizadas a cada quinze segundos. O método **main()** é responsável por inicializar a janela conta também com os métodos responsáveis pela atualização.

Após receber da aplicação servidora as tabelas e *strings* de respostas, ela são escritas linha por linha em um componente **JTextArea**. O método responsável por gerar o documento é chamado por um **ActionListener** presente no botão. Ele cria um arquivo texto na mesma pasta da aplicação Java e escreve as estatísticas atuais neste arquivo.

2.5. Possíveis Novos Componentes

O sistema de gerenciamento de vagas de estacionamento exposto neste trabalho (Figura 1) é um sistema que tem como uma de suas premissas a separação de seus componentes, conceito importante para superar desafios como a heterogeneidade das aplicações finais. Desta maneira, diferentes aplicações finais podem ser integradas ao sistema, além de propiciar mais recursos e funcionalidades.

São inúmeras as funcionalidades que podem ser implementadas em um serviço

de estacionamento. Baseado no modelo de negócios conhecido de gerências de estacionamentos, o sistema poderia, por exemplo, contar com uma página *web* ou ser posto como serviço de modo integrado aos próprios *websites* dos estabelecimentos já em funcionamento, onde seriam ofertadas funcionalidades como exibição do mapa de vagas, pagamento de tíquete, informações sobre preços e promoções, publicidades relacionadas ao estabelecimento, entre outras. Esta ferramenta poderia ser integrada ao sistema na forma de *web services*.

Outro componente que poderia ser integrado ao sistema seria um totem provido de uma tela *touchscreen* localizada idealmente nas cancelas de entradas e saídas dos estacionamentos, onde os usuários poderiam rapidamente usufruir do recurso de visualização do mapa das vagas, colhendo informações como nível de ocupação por setor, número de vagas livres e ocupadas por setor, localização de vagas cobertas, entre outros.

3. Estratégias de Integração dos Componente Embarcados, Servidores e Clientes

O produto descrito na Seção 2 tem seu funcionamento atrelado ao trabalho em conjunto de diversos componentes que são desenvolvidos em diferentes de programação e podem ser executados em sistemas operacionais distintos. Para construir um sistema que integre estas diferentes partes é necessário se pensar em estratégias para que a comunicação entre elas seja correta.

Em seu início do desenvolvimento, o projeto apresentou um viés mais comercial, em virtude da disciplina em que ele foi iniciado, Gestão de Projetos Multimídia, disciplina do curso de graduação Sistemas e Mídias digitais, que apresentava um aspecto de visão de mercado mais acentuada, além também de oportunidades comerciais que surgiram fora do âmbito acadêmico. Por essa razão, o desenvolvimento se baseou em integrar bases de dados existentes das administradoras dos estacionamentos a aplicações e soluções propostas. A solução teria que ser simples, para que a sua implementação fosse rápida e também possibilitasse que novos dispositivos fossem adicionados ao sistema sem grandes mudanças na arquitetura.

Esta seção tratará como estas estratégias de integração funcionam, como foram construídas, quais os componentes foram incorporados e como foram usados.

3.1. Estratégia de Integração e Comunicação – Arquitetura e Visão geral

O fluxo de informação do sistema se inicia no Arduíno. A partir dos sensores conectados a suas portas digitais, ele coleta os estados das vagas do estacionamento. Posteriormente, concatena essas informações em uma *string*, escrevendo 0 (zero) para uma vaga vazia e 1 (um) para uma ocupada, formando por exemplo a frase 010010 e a envia pelo serial, no qual está conectado a um computador com função de coletor de dados.

Com o auxílio da biblioteca RXTX, uma aplicação Java presente no computador coletor lê as informações do serial e as envia usando o JDBC para um banco de dados.

Para o acesso a esses dados, existe uma aplicação servidora que está conectado ao banco de dados. Os aplicativos finais entram em contato com esse servidor que os devolve informações como o mapa de vagas do estacionamento, ocupação dos setores, pagamento de tíquete e estatísticas de ocupação das vagas. A Figura 12 mostra o fluxo de informação do sistema e as ferramentas usadas para a comunicação.

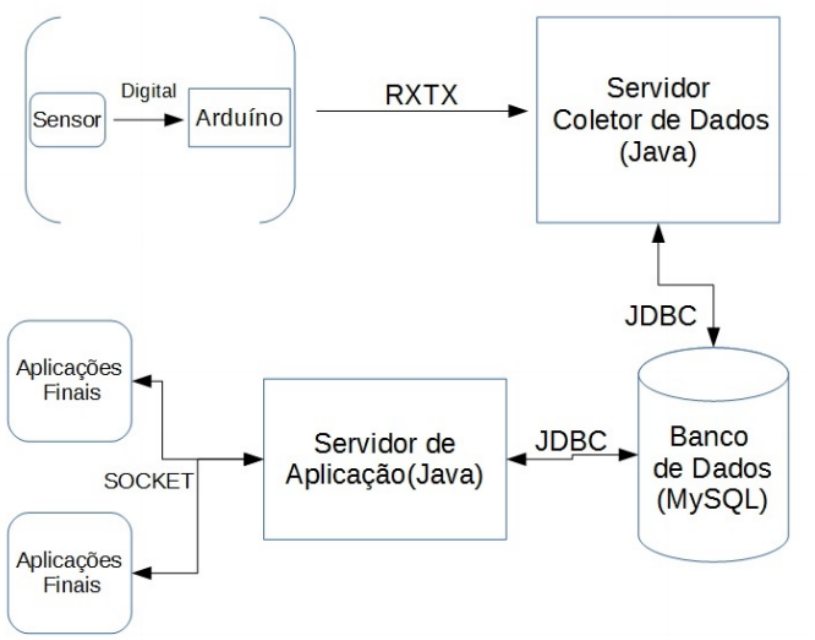

Figura 12. Fluxo de informação do sistema e ferramentas de comunicação

3.2. Arduíno

O primeiro componente a ser trabalhado foi o Arduíno. Ele é uma parte muito importante do sistema já que é responsável por requisitar as informações sobre as vagas do estacionamento dos sensores.

Os sensores HC-SR04 se comunicam com o Arduíno através dos pinos digitais. Com as informações obtidas a placa calcula se as vagas referentes estão ocupadas ou não.

A partir deste cálculo, o Arduíno forma uma *string* que representa a ocupação do estacionamento. Essa frase é formada pela concatenação dos caracteres 0 e 1, 0 representando uma vaga vazia e 1 representando uma em uso, formando por exemplo 01011, que representa um estacionamento onde a vaga 1 está vaga, a dois está em uso e assim por diante. Com a *string* formada, a placa então a envia para o computador coletor através do cabo serial. A Figura 13 mostra um exemplo de um estacionamento e a *string* resultante do seu estado atual.

3.3. Java

Um dos grandes desafios presentes no planejamento e consequente construção do sistema, foi o fato de ele apresentar diferentes partes que necessitavam trabalhar juntas para que o objetivo final fosse alcançado e as informações chegassem aos usuários.

Seria preciso criar um ambiente entre a placa Arduíno, o banco de dados, o servidor e as aplicações que iriam se utilizar dos dados. Sendo assim, necessita-se de uma linguagem que fosse flexível o suficiente para trabalhar em cima de diferentes dispositivos e que apresentasse meios para agregar funcionalidades diferentes, como um banco de dados e bibliotecas. Nesse contexto, o Java, onde as características da linguagem se encaixaram perfeitamente com as necessidades.

O primeiro problema que o Java resolveu foi agregar as diferentes tecnologias que

Figura 13. Exemplo de um estacionamento e a *string* que representa o seu estado

foram usadas na construção do sistema, embora a linguagem não as suportasse nativamente, existem diversas bibliotecas que ampliem as suas capacidades. No sistema foram utilizadas duas bibliotecas, a JDBC e a RXTX, ambas desempenhado papéis importantes no fluxo de informação do sistema.

Uma das características desejadas no sistema era que o cliente *desktop* pudesse ser executado em qualquer computador, não importando o seu sistema operacional. Para isso seria necessário reescrever o código do cliente para cada sistema, necessitando assim mais tempo de desenvolvimento. O Java resolve essa situação. Ele é uma linguagem projetada para a portabilidade, que independente de plataforma, onde o desenvolvedor escreve o código uma vez e executa em qualquer lugar, "*write once, run everywhere*" ou "WORA".

Os códigos executam em uma máquina virtual, a JVM (*Java Virtual Machine*). Ela carrega e executa as aplicações em java, convertendo-as em código executável de máquina. Assim basta apenas que o computador possua uma versão da JVM instalada para que ele possa executar a aplicação. Uma outra facilidade proporcionada pelo Java foi a de distribuição da aplicação *desktop*, bastando apenas copiar o arquivo ".jar", o executável do Java, para o computador do usuário.

3.4. Aplicação Coletora de Dados

O sistema utilizou sistematicamente o Arduíno. Periodicamente ele reúne as informações das vagas e as envia pelo serial para o computador. Porém, como descrito previamente, o Java não consegue nativamente ler e manipular o serial. Então, como solução, utilizou-se o RXTX. O RXTX é uma biblioteca Java, que provê comunicação serial e paralela para o Kit de Desenvolvimento Java (JDK - (*Java Development Kit*).

Uma aplicação coletora de dados, instalada no computador no qual o Arduíno

está conectado, se utilizando RXTX, lê constantemente as informações sobre o estacionamento enviadas pela placa.

De posse dos dados do estado do estacionamento, o próximo passo é persisti-los no banco para o uso por outras aplicações. A biblioteca usada para isso é a JDBC. O JDBC é uma biblioteca Java que reúne classes e interfaces que possibilita se conectar, por meio de um *driver* especifico, ao banco de dados desejado. Com este *driver* é possível executar instruções SQL no banco de dados relacional. Como na aplicação utilizamos o banco MySQL, utilizamos o seu *driver* para possibilitar a conexão.

A aplicação coletora exerce um papel de suma importância na integração das partes do sistema. Ela é responsável por constantemente receber e tratar as informações oriundas do Arduíno e posteriormente guardá-las para que possam ser utilizadas pelos outros componentes da aplicação, realizando esses trabalhos durante toda a execução. Ela é a parte que une os elementos embarcados e o servidor da aplicação de uma forma que os dois se entendam.

3.5. Aplicação Servidora

Uma vez devidamente estabelecidas a integração e comunicação entre os sensores até o banco de dados, todas as informações do estacionamento estão sendo salvas constantemente a cada atualização do estado das vagas. Mas ainda existe trabalho a ser feito para que estas informações estejam à disposição dos aplicativos finais e dos usuários.

Para resolver este problema a aplicação servidora foi criada. Ela é um servidor escrito em java, tendo como funções receber os pedidos dos aplicativos finais, resgatar do banco de dados as informações para responder tal pedido e enviar as respostas de uma forma que os aplicativos possam entender.

Entretanto, para que os pedidos sejam recebidos com êxito, respondidos de forma correta, e a integração do servidor e das aplicações finais fossem corretas, fez-se necessário criar protocolos para a comunicação, padrões que guiassem as trocas de informação e ajudassem os desenvolvedores dos aplicativos finais no momento de escrever os códigos responsáveis pela conversa.

A primeira medida a ser tomada foi criar uma forma para que as aplicações finais entrassem em contato com a aplicação servidora. Para isso foi usado a API Sockets do pacote **java.net**. Esta API, que é nativa do Java, ou seja, não é necessário a instalação de nenhuma outra biblioteca de terceiros para seu funcionamento, permite ao desenvolvedor utilizar várias ferramentas que auxiliam na construção de um programa para comunicação entre dois computadores. Para construir o servidor, utilizou-se três recursos presentes na API: protocolo TCP, porta e *socket*.

O TCP (*Transmission Control Protocol*) é um protocolo que gerencia as trocas de mensagens entre os dois computadores comunicantes, facilitando assim o trabalho do desenvolvedor, já que ele não terá que programar algo tão baixo nível. O TCP, se utilizando de diversas técnicas, garante que as mensagens enviadas serão corretamente entregues, ou seja, completas e ordenadas.

A porta é uma solução encontrada para endereçar diversas aplicações dentro de uma mesma máquina, podendo variar de 0 a 65535.

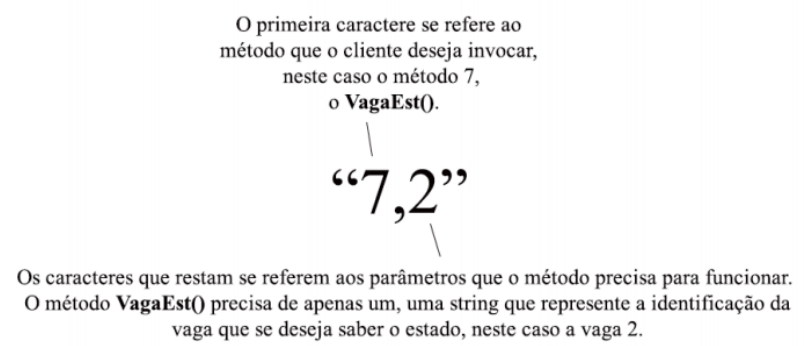

Figura 14. Descrição do significado dos caracteres da *string* código

Por fim, o *socket*, que é um ponto de comunicação entre duas máquinas diferentes, ou seja, mensagens podem ser trocadas entres duas máquinas utilizando uma conexão estabelecida pelo *socket*.

No servidor, a primeira ação a ser feita é criar um **ServerSocket**, passando como parâmetro a porta que será utilizada pela aplicação. Um **ServerSocket** é responsável por esperar as conexões do cliente. A aplicação utilizou a porta 6667.

Dado que as aplicações finais podem contatar a aplicação servidora, foi necessário estabelecer a maneira na qual elas comunicarão ao servidor o que desejam. Para isso é neecssário criar um padrão de requisição (*request*), ou pedido, para as aplicações.

A solução encontrada foi estipular uma *string* código a ser passada junto ao *socket* inicial da conexão. O código seria formado de uma concatenação de *strings* separadas por “,”.

A primeira parte da *string* seria referente a um número previamente estabelecido que seria traduzido na aplicação servidora a um método, ou seja, o cliente passaria como primeiro valor o número referente ao método que ele quer invocar. No código da aplicação, o método **MapaEst()**, que devolve uma lista com a última atualização das vagas do estacionamento e seu estado, tem como código o 0, então o cliente teria como primeiro dado o 0.

Os valores seguintes presentes na *string* seriam referentes aos parâmetros necessários para o funcionamento do método que o cliente deseja invocar. Continuando com o exemplo, o método **MapaEst()** precisa da identificação correspondente ao estacionamento que o cliente quer o mapa atualizado das vagas. A Figura 14 mostra uma descrição do significado de cada caractere da *string*. Assim, o cliente que deseja o mapa atualizado do estacionamento de identificação 1 enviaria a seguinte *string*: “0,1”.

Cada método da aplicação terá seu próprio código para que possa ser chamado pelo cliente, também como seus parâmetros. A Tabela 1 exibe o nome do método, uma descrição do que ele faz seguido do seu código e parâmetros necessários.

Como a solução empregada utiliza uma combinação de *strings* e *sockets*, que são funcionalidades nativas da plataforma, não existe a necessidade da instalação de qualquer outra API, facilitando a criação e distribuição das aplicações clientes.

Depois de receber dos aplicativos finais os *sockets*, o próximo passo para o ser-

vidor da aplicação é tratar as *strings* para separar os parâmetros passados e organizar o pedido do cliente. A solução utilizada é separar a *string* recebida pelo "," utilizando o *split* e jogando as partes resultantes em um vetor de *string*.

Com os parâmetros separados, o servidor lê o vetor seguindo as regras do protocolo apresentados. O primeiro dado, que representa o método a ser chamado é passado por um **switch()**, que serve para controlar várias ações diferentes que podem ser tomadas de acordo com uma variável, nesse caso o código do método. De acordo com o código, um *case* é invocado, que são as possibilidades de resultados obtidas pelo *switch*, e nele o método é executado se utilizando dos restantes dos dados encontrados no vetor de *string*.

Durante esta seção foi demostrado como a aplicação servidora seria achada pelos aplicativos finais, como eles devem se comunicar e como o servidor trata os pedidos, mas ainda é necessário explicar como as respostas são entregues aos clientes.

Cada método da aplicação devolve algum resultado, seja uma simples *string* ou uma tabela, estão é preciso criar uma maneira de enviar estas informações e que elas possam ser corretamente lidas do lado dos clientes.

Para enviar as informações, o recurso empregado foi a criação de um **ServerSocket()** no cliente após o envio do *socket* de pedido. Este servidor temporário no cliente funciona na porta 6668 e fica na espera pela resposta do servidor com os dados. Na aplicação servidora é criado um *socket*, endereçado ao IP (*Internet Protocol*) do qual foi recebido o pedido e nele são guardados os dados resposta a serem enviados.

Desta forma é criada uma desacoplação entre as partes do sistema, em que só se comunicam durante os pedidos e respostas. Fora desse período não há conexão nenhuma entre eles, deixando mais fácil o acréscimo de novas aplicações finais.

Depois da criação do canal de entrega, é necessário então empacotar a resposta de uma forma em que ela possa ser enviada pela rede e que após sua chegada possa ser lida corretamente pelo cliente. A resposta foi o **Serializable**, recurso nativo do Java.

O **Serializable** em Java permite transformar um objeto em um fluxo de bits para que possar ser enviado pela rede e depois no destino possa ser transformado novamente em objeto. O desenvolvedor implementa essa função através da interface **Serializable** nas

Tabela 1. Relação entre os métodos e os códigos de chamada pelos clientes

Método	Descrição	Código	Parâmetro de entrada
MapaEst()	Devolve mapa atualizado do estacionamento	0	Identificação do estacionamento, inteiro
Rel()	Devolve relatório do uso das vagas	1	Identificação do estacionamento, inteiro
PagarTicket()	Confirma o pagamento do tíquete	2	Código de barras do tíquete, inteiro
VagaEst()	Devolve o estado de apenas uma vaga	7	Identificação da vaga, inteiro
OcupSetor()	Devolve mapa de ocupação por setor	5	Identificação do estacionamento, inteiro

classes da aplicação.

No entanto, para que o cliente possa traduzir o fluxo de bits para objeto, ele precisa ter uma cópia da classe em seu código fonte, para servir como modelo de interpretação dos bits. Para qualquer outra funcionalidade implementada em novas aplicações clientes, um novo método deve ser construído no servidor de aplicações.

3.6. Aplicações Finais

As aplicações finais, independente de suas especificidades, como sistema operacional e dispositivos nos quais executam, têm que seguir o protocolo estabelecido para garantir a comunicação.

Um *socket* deverá ser criado com o código e enviá-lo para o servidor, como estabelecido previamente, então deve-secriar um **ServerSocket** na porta 6668 e esperar pela resposta do servidor.

No código fonte da aplicação será necessário ter uma cópia das classes presentes no servidor e elas terão também que implementar a interface **Serializable**, para que seja garantido a tradução correta.

Duas aplicações foram desenvolvidas, uma móvel em Android, não apresentada neste trabalho, e uma *desktop*, em Java Swing, para validar as estratégias tomadas.

Para atualizar os dados apresentados na tela, três métodos que se comunicam com a aplicação servidora, o **AtualizarVaga()**, que atualiza o quadro correspondente as vagas, o **AtualizarSetor()**, que atualiza as informações de ocupação dos setores e o **AtualizarEst()**, que atualiza as estatísticas sobre o uso do estacionamento. Eles são invocados de 15 em 15 segundos, o mesmo tempo que o Arduíno envia as informações dos sensores ao servidor coletor.

Os três métodos funcionam de maneira semelhante, criando um *socket* e enviando para a aplicação servidora o pedido com as informações que necessitam. Em seguida criam um **ServerSocket** para receber a resposta. Para exibir as informações na interface da aplicação, a resposta é lida linha por linha e é escrita em uma **JTextArea** utilizando um comando *append*.

4. Conclusão

Este trabalho consistiu no desenvolvimento de uma estratégia para integração e comunicação entre diferentes dispositivos, sendo eles sensores, Arduíno, servidores, banco de dados e aplicações *mobile* e *desktop*, para que eles pudessem entregar informações do estado de um estacionamento para o cliente através da internet. Para isso, construiu-se um estacionamento (modelo/protótipo de estacionamento) com uma rede de sensores para determinar o estado das vagas, comandadas por uma placa Arduíno, um servidor para coletar estas informações e as salvar em um banco de dados, um outro servidor para gerenciar o acesso a estes dados e, por fim, clientes, para utilizar as informações e validar as estratégias de comunicação.

Ao fim do desenvolvimento do sistema, os resultados obtidos foram bastante positivos. A criação de um sistema que integra componentes tão diferentes não é trivial, mas as ferramentas utilizadas ofereceram muita ajuda.

O uso da biblioteca RXTX possibilitou a comunicação dos sensores e do Arduíno com o computador, comunicação essa que era crucial para o sistema.

Os recursos utilizados pela aplicação servidora e os aplicativos finais são nativos da plataforma Java, assim não há a necessidade de instalações de APIs de terceiros deixando o sistema com uma maior facilidade de distribuição e instalação, aumentando a compatibilidade. Além disso, a linguagem proporciona uma manutenção mais fácil e rápida do sistema.

Ainda é importante notar um possível paralelo dos conceitos mostrados no sistema com a Internet das Coisas (*Internet of Things* - IoT). Nela, objetos do dia a dia, como eletrodomésticos e roupas, são conectados à internet. No trabalho apresentando, um estacionamento foi conectado a grande rede com o intuito de ampliar suas possibilidades de uso e facilitar os acessos as informações.

No âmbito acadêmico do curso de Sistema e Mídias Digitais, da Universidade Federal do Ceará, consideramos também a aplicação como um sucesso, pois apresenta conceitos estudados durante todo o período do curso, apresentado em diversas disciplinas. Disciplinas como Programação Orientada a Objetos e Estruturas de Dados foram responsáveis pelo conhecimento sobre o Java, Redes de Computadores e Sistemas Distribuídos ensinaram conceitos sobre redes que foram utilizados na construção do sistema, Bancos de Dados Aplicados a Multimídia ajudou na construção do banco de dados da aplicação, Programação *Web* demonstrou as ferramentas utilizadas no sistema para comunicação em rede e Sistemas Embarcados auxiliaram no desenvolvimento das redes de sensores presentes no estacionamento.

Os problemas encontrados durante o desenvolvimento foram o acesso aos materiais utilizados na construção do estacionamento. Os sensores e a placa Arduíno são difíceis de encontrar no mercado local, portanto tendo que ser importados, o que gera um alto tempo de espera e custo, já que seriam pagos em dólar.

Para trabalhos futuros seria interessante a ampliação do estacionamento, com um aumento no número de sensores e placas, além do desenvolvimento de novas funcionalidades com um servidor para aplicações *web*. Adicionalmente pretende-se analisar o desempenho da aplicação e avaliar a segurança na troca de informações no sistema.

Referências

Barros, E. e Cavalcante, S. (2016). Introdução aos sistemas embarcados. http://www.cin.ufpe.br/∼vba/periodos/8th/s.e/aulas/STP%20-%20Intro%20Sist%20Embarcados.pdf. Online; acessado em maio-2016.

Coulouris, G., Dollimore, J., e Kindberg, T. (2013). *Sistemas Distribuídos - Conceitos e Projeto*. Bookman, 5 edition.

Ingresso.com (2016). Home - fortaleza - ingresso.com. http://www.ingresso.com/fortaleza/home/. Online; acessado em maio-2016.

Runkeeper (2016). Runkeeper - track your runs, walks and more with your iphone or android phone. https://runkeeper.com/. Online; acessado em maio-2016.

Waze (2016). Aplicativo gratuito de trânsito e navegação baseado em mapas da comunidade. https://www.waze.com/pt-BR. Online; acessado em maio-2016.