Universidade Federal do Rio de Janeiro

Instituto de Matemática – IM

Núcleo de Computação Eletrônica - NCE

Um Sistema para Gerenciamento de Serviços de Táxi Utilizando Sistemas Embutidos

Rodrigo Machado de Camargo Caixeta

Orientador: Prof. Manuel Lois Anido, Ph.D.

Rio de Janeiro, RJ – Brasil

fevereiro de 2006

Um Sistema para Gerenciamento de Serviços de Táxis Utilizando Sistemas Embutidos

Rodrigo Machado de Camargo Caixeta

Dissertação submetida ao corpo docente do Instituto de Matemática / Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para obtenção do grau de Mestre em Ciências da Informática.

Aprovada por:

_____

Prof. Manuel Lois Anido, Ph.D. (Orientador)

_____

Prof. Carlo Emmanuel Tolla de Oliveira, Ph.D. (Co-Orientador)

_____

Prof. Felipe Maia Galvão França, Ph.D.

_____

Prof. Adriano Joaquim de Oliveira Cruz, Ph.D.

_____

Prof. Antonio Carlos Gay Thomé, Ph.D.

Rio de Janeiro, RJ – Brasil

fevereiro de 2006

FICHA CATALOGRÁFICA

Caixeta, Rodrigo Machado de Camargo

Um Sistema para Gerenciamento de Serviços de Táxis Utilizando Sistemas Embutidos / Rodrigo Machado de Camargo Caixeta – Rio de Janeiro 2006

134 p.:il

Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro – UFRJ , Instituto de Matemática – IM/NCE

Orientador: Manuel Lois Anido

1. Gerenciamento de Táxis, Sistemas Embutidos, Comunicação com Veículos, TINI, GPRS

I. Anido, Manuel Lois Anido (Orientador). II. Universidade Federal do Rio de Janeiro. Instituto de Matemática. III. Título.

À minha esposa Edna e ao meu filho Pedro,

Pela compreensão nas horas em que estive ausente.

Aos meus Pais, José Caixeta e Maria Francisca

Pelo carinho e incentivo.

AGRADECIMENTOS

Agradeço a Deus pela possibilidade, força e saúde para realizar este trabalho.

Em especial ao professor Manuel Lois pela orientação, paciência e confiança. Agradeço pela seriedade na orientação além do empenho e dedicação.

Ao professor Carlo Emmanuel pelo apoio ao longo do período de elaboração deste trabalho.

À professora Lígia Alves Barros que muito me ensinou para realização desta dissertação.

Aos professores dos cursos de pós-graduação em Informática da Universidade Federal do Rio de Janeiro.

À Universidade Federal do Rio de Janeiro pela oportunidade de realização do curso de Mestrado em Informática.

SUMÁRIO

RESUMO **1**

ABSTRACT **2**

1 INTRODUÇÃO **3**

1.1 VISÃO GERAL 3
1.2 OBJETIVOS GERAIS 4
1.3 ABORDAGEM METODOLÓGICA 5
1.4 ORGANIZAÇÃO DO TRABALHO 5

2 REVISÃO BIBLIOGRÁFICA **7**

2.1 INTRODUÇÃO 7
2.2 ARTIGOS REVISADOS 7
2.3 CONCLUSÃO DA REVISÃO BIBLIOGRÁFICA 11

3 REVISÃO DE TECNOLOGIAS UTILIZADAS NO DESENVOLVIMENTO DO SISTEMA **13**

3.1 SISTEMAS EMBUTIDOS – VISÃO GERAL 13
3.1.1 Introdução 13
3.1.2 Microcontroladores 13
3.1.3 Sistemas Embutidos - Arquitetura 14
3.1.4 Sistemas Embutidos - CPUs 15
3.1.5 Componentes Básicos 16
3.2 LINGUAGEM JAVA 17
3.2.1 Introdução 17
3.2.2 Funcionamento 18
3.2.3 Comparação entre Java e as outras linguagens 18
3.2.4 Arquitetura da Linguagem Java 19
3.2.5 Máquina Virtual Java 19
3.2.6 Plataformas de Desenvolvimento Java 22
3.2.7 Java e Sistemas Embutidos 24
3.3 PLATAFORMA TINI (PLACA + API + SISTEMA OPERACIONAL) 27
3.3.1 Introdução 27
3.3.2 TINI - Especificações Técnicas 28
3.3.3 TINI - Software 35
3.3.4 Comunicação Serial 37
3.3.5 Rede TCP / IP 40
3.3.6 Interface PPP 42
3.4 TECNOLOGIAS DE COMUNICAÇÃO SEM FIO 44
3.4.1 Introdução 44
3.4.2 GSM 45
3.4.3 GPRS 46
3.4.4 Aplicações do GPRS 48
3.5 CONCLUSÕES DA REVISÃO DE TECNOLOGIAS 48

4 ANÁLISE, PROJETO E IMPLEMENTAÇÃO DE UM SISTEMA PARA GERENCIAMENTO DE TÁXIS UTILIZANDO SISTEMAS EMBUTIDOS 49

4.1 INTRODUÇÃO 49
4.2 ANÁLISE E PROJETO 50
4.2.1 Modelo de operação "A" - Ponto de Apoio Fixo 50
4.2.2 Modelo de operação "B" - Ponto de Apoio Flutuante 52
4.2.3 Análise dos modelos "A" e "B" 54
4.2.4 Regras de negócio 54
4.2.5 Modelagem de Dados 60
4.2.6 Implementação (Aplicação Servidora) 62
4.2.7 Implementação (Aplicação Cliente) 75
4.2.8 Configuração do Ambiente de Desenvolvimento 81
4.3 CONCLUSÕES DO CAPÍTULO 4 91

5 RESULTADOS 93

5.1 INTRODUÇÃO 93
5.2 DESCRIÇÃO DOS RESULTADOS 93
5.2.1 Sistema Cliente 93
5.2.2 Sistema Servidor 95
5.2.3 Simulador *Java Swing* para testes de operação 96
5.2.4 Análise do custo da transmissão dos dados via GPRS 97
5.2.5 Teste de Carga (Teste de *Stress*) 100
5.2.6 Cenário para 1 (um) usuário 103
5.2.7 Cenário para 10 (dez) usuários simultâneos 104
5.2.8 Cenário para 30 (trinta) usuários simultâneos 105
5.2.9 Cenário para 50 (cinqüenta) usuários simultâneos 107
5.3 CONCLUSÕES SOBRE OS RESULTADOS OBTIDOS 108

6 CONCLUSÕES 110

6.1 INTRODUÇÃO 110
6.2 PRINCIPAIS DIFICULDADES 110
6.3 CONCLUSÕES FINAIS 110
6.4 TRABALHOS FUTUROS 111

7 REFERÊNCIAS 113

APÊNDICES 116

APÊNDICE A: DETALHAMENTO DOS SINAIS DOS PINOS DE UM CONECTOR DB-9 ... 116

ANEXOS 117

ANEXO A: ESQUEMA ELÉTRICO DA PLACA BASE E10 (SOCKET BOARD) 117

LISTA DE FIGURAS

Figura 1 – Dispositivo móvel *TopAuto* para controle de frotas. 10
Figura 2 – Segmentos de mercado. 14
Figura 3 – Arquitetura de um sistema embutido. 15
Figura 4 – Carregador de classes Java (Aplicação Java + API). 20
Figura 5 – SaJe Java Controller – Placa para execução de código Java nativo. 25
Figura 6 – JStik Java Controller – Outra placa para execução de código Java nativo. 26
Figura 7 – Placa TStik da Sistronix – Execução de código Java compilado para a plataforma. 26
Figura 8 – Placa TINI da *Dallas Semiconductors* (Microcontrolador DS80C390). 27
Figura 9 – Diagrama de blocos da placa TINI390. 29
Figura 10 – Mapa de memória da plataforma TINI. 30
Figura 11 – TBM 390 *(Dallas Semiconductors)* – Visão frontal. 31
Figura 12 – TBM 390 *(Dallas Semiconductors)* – Visão traseira. 31
Figura 13 – Placa *E10 TINI Socket Board* auxiliar para conexões com interfaces. 32
Figura 14 – Placa STEP (Outra placa auxiliar para TINI desenvolvida pela *Systronix*). 33
Figura 15 – Placa de extensão SBX2 para conexão com periféricos (Ex: teclado e *display*). 34
Figura 16 – *Display* 20x4 (conectado a placa STEP via extensão SBX2). 34
Figura 17 – Teclado tipo membrana (conectado a placa STEP via extensão SBX2). 34
Figura 18 – Ambiente de execução da plataforma TINI (*Runtime*). 35
Figura 19 – Protocolos de rede implementados pela plataforma TINI. 40
Figura 20 - Diagrama de estados da interface *PPPDeamon,* incluída na API da plataforma TINI. 44
Figura 21 – Modelo de dados – Visão das principais tabelas (parte 1). 60
Figura 22 – Modelo de dados – Visão das principais tabelas (parte 2). 61
Figura 23 – Modelo de dados – Visão das principais tabelas (parte 3). 61
Figura 24 – Estrutura hierárquica do projeto *tinitaxi-ear* (Divisão lógica de camadas). 64
Figura 25 – Fluxo de talas do sistema embutido implementado (arquivos *.screens*). 71
Figura 26 – Fluxo de telas do sistema embutido implementado *(Display 20x4)* – Parte 1. 73
Figura 27 – Fluxo de telas do sistema embutido implementado *(Display 20x4)* – Parte 2. 73
Figura 28 – Distribuição das teclas no *keypad* – Sistema TiniTaxi. 77
Figura 29 – Configuração do cabo serial para conexão do celular T68i a placa. 78
Figura 30 – Aplicação *Java Swing* para simulação de uma unidade móvel (táxi). 81
Figura 31 – Dispositivo móvel TiniTaxi montado (plataforma de *hardware* TINI). 94
Figura 32 – Dispositivo móvel TiniTaxi montado (plataforma de *hardware* TINI). 94
Figura 33 – Sistema servidor (log das requisições dos clientes – *client requests*). 96

Figura 34 – Aplicação Java Swing para simulação de uma unidade móvel (táxi). 96
Figura 35 – Simulação da operação com vários táxis simultaneamente (aplicação *Java Swing*). 97
Figura 36 – Tela de configuração da execução (configuração das *threads*) – *Jmeter*. 102
Figura 37 – Gráfico de execução para o cenário com 1 (um) usuário. 103
Figura 38 – Gráfico de execução para o cenário com 10 (dez) usuários simultâneos. 104
Figura 39 – Gráfico de execução para o cenário com 30 (trinta) usuários simultâneos. 106
Figura 40 – Gráfico de execução para o cenário com 50 (cinqüenta) usuários simultâneos. 107
Figura 41 – TINI Socket: Parte 1 118
Figura 42 – TINI Socket: Parte 2 119
Figura 43 – TINI Socket: Parte 3 120
Figura 44 – TINI Socket: Parte 4 121

LISTA DE TABELAS

Tabela 1 – Representação dos dados do GPS registrados via HTML do dispositivo móvel. 8
Tabela 2 – Tarifas *TIM CONNECT FAST* – Serviço oferecido pela operadora TIM (RJ). 47
Tabela 3 – Detalhamento dos sinais dos pinos de um conector DB-9..... 116

LISTA DE ABREVIATURAS E SIGLAS

3G	Third Generation Mobile System
API	Application Programming Interface
AP	Acess Point
ATM	Asynchronous Transfer Mode
B2B	Business to Business
B2C	Business to Comerce
BIOS	Basic Input/Output System
CAN	Controller Area Network
CDMA	Code Division Multiple Access
CI	Circuito Integrado
CSD	Circuit Switch Data
CPU	Central Processing Unit
DAO	Data Access Object
DECT	Digital Enhanced Cordless Telecommunications System
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name Service
DS	Data Source
EDGE	Enhanced Data for GSM Evolution
EDGE	Enhanced Data Rates for GSM Evolution
EJB	Enterprise Java Beans
ETSI	European Telecommunications Standard Institute
FTP	File Transfer Protocol
GC	Garbage Collector
GPRS	General Packet Radio System
GPRS	General Packet Radio Service
GPS	Global Positioning System
GSM	Group Special Mobile
GSM	Global System for Mobile Communication
HSCSD	High Speed Circuit Switched Data
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IEEE	The Institute of Electrical and Electronics Engineers, Inc
IO	Input Output
IP	Internet Protocol
ISO	International Organization of Standardization
ISO	International Standard Organization
J2EE	Java 2 Enterprise Edition
J2SE	Java 2 Standard Edition
J2ME	Java 2 Micro Edition
JAVA	Linguagem de Desenvolvimento Orientada a Objetos
JCA	Java Connector Architecture
JDBC	Java Database Connectivity
JDK	Java evelopment Kit
JVM	Java Virtual Machine
JSP	Java Server Pages

JRE	Java Runtime Environment
JTA	Java Transaction Architecture
LAN	Local Area Network
MMS	Multimedia Messaging System
PDA	Personal Data Assistance
PPP	Pear to Pear Protocol
RAM	Random Access Memory
RFC	Request for Comments
ROM	Read Only Memory
RS	Result Set
SBX2	Socket Board Extension 2
SQL	Structure Query Language
TBM	Tini Board Model
TINI	Tiny Internet Interface
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol / Internet Protocol
TDMA	Time Division Multiple Access
TO	Transfer Object
UM	Unidade Móvel
UMTS	Universal Mobile Telecommunication System
VO	Value Object
WAN	Wide Area Network
WAP	Wireless Application Protocol
WAS	WebSphere Application Server
WSAD	WbSphere Studio Application Developer
WLAN	Wireless Local Area Network
WML	Wireless Markup Language
XML	Extensible Markup Language

RESUMO

Esta dissertação descreve a análise e a implementação do protótipo de um sistema para dar suporte a serviços de táxis, baseado na localização geográfica e no gerenciamento de filas de pedidos de atendimento. A proposta deste trabalho é inovadora, pois difere dos atuais sistemas de comunicação via radiofreqüência utilizada por companhias de táxi e também difere dos sistemas de localização de veículos que se baseiam em dispositivos de posicionamento global GPS (*Global Positioning System*).

O sistema proposto nesta dissertação utiliza um pequeno sistema embutido que é implantado no veículo do taxista e é capaz de dar suporte à interação entre o motorista e um servidor, dispensando a figura do operador de rádio na central de controle. As principais vantagens deste sistema, em comparação com os sistemas baseados em rádio-operador ou os sistemas que empregam uma unidade de GPS, são: a comunicação confiável, a resolução automática de conflitos, o gerenciamento automático das filas de atendimento e a possibilidade de obter custos de instalação e de operação mais baixos.

ABSTRACT

This dissertation describes the analysis and prototype implementation of a system to provide support for taxi services which are based on the geographical positioning and on the management of requesting and delivering queues. The proposal presented in this work differs from the present taxi communication systems, which are based on a radio operator, and it also differs from other vehicle tracking systems based on global positioning systems GPS (*Global Positioning System*).

The system proposed in this dissertation employs a small embedded system, which is implanted on the taxi panel, and is capable of providing the necessary support for the interaction between the taxi driver and central server, eliminating the need for operators at the command center. The main advantages of this system, in comparison with the systems based on a radio operator or with systems that employ a GPS unit, are reliable communication, automatic conflict resolution, automatic management of request queues and the possibility of achieving lower installation and operation costs.

1 INTRODUÇÃO

1.1 Visão Geral

A evolução e a convergência de duas tecnologias, a Internet e os sistemas de localização global estão criando novas oportunidades para empresas e usuários de sistemas com base em dispositivos móveis. Atualmente existe a necessidade, no setor de transportes de passageiros, de um gerenciamento automático para o posicionamento de veículos a fim de melhorar o atendimento e diminuir a mão de obra necessária para os serviços.

As soluções para gerenciamento e localização de veículos utilizam normalmente a informação da posição física do mesmo, a qual é obtida através de um dispositivo GPS (*Global Positioning System*). Com esse dispositivo é possível identificar a posição geográfica dos veículos, e assim gerenciar, por exemplo, entregas de produtos e mercadorias, prioridades de atendimento de chamadas, entre outros. Uma implementação comum utilizada nesse tipo de sistema é a integração entre os dispositivos GPS, PDA (*Personal Digital Assistants*) e aparelhos de comunicação móvel (celulares, por exemplo). Entretanto, esse tipo de implementação apresenta alguns tipos de problemas quando se leva em conta o posicionamento geográfico do veículo:

1. A complexidade na implementação de sistemas de controle com dados geo-referenciados

   Para implementação de um sistema que gerencie veículos através de posicionamento geográfico utilizando GPS, deve-se antes mapear todas coordenadas geográficas para as áreas de atendimento correspondentes (bairros, setores, ruas, etc). Algumas cidades já possuem este tipo de mapeamento (ex. Nova York, EUA), entretanto, a grande maioria das cidades não possui o mesmo.

2. O mapeamento das regiões de atendimento pode ser um mapeamento lógico, não levando em conta somente os dados geográficos.

   Determinados tipos de serviços utilizam um mapeamento lógico de áreas de atendimento, como por exemplo, companhias de táxi que atendem determinadas

empresas, sem posicionamento geográfico definido (ex: profissionais da empresa A são atendidos sempre por veículos X, independentemente da posição física da chamada).

3. Algoritmo de alocação entre o veículo, a sua localização geográfica e a localização de destino.

   De acordo com a posição geográfica do veículo não é possível identificar problemas quanto ao sentido do trânsito, conversões, condições do tráfego no momento da chamada, etc. Essas informações são essenciais para definição do algoritmo de alocação entre o veículo e o endereço de destino.

Existem alguns sistemas que possuem essas características e apresentam esses mesmos problemas, como por exemplo, a implementação de um controle de filas de atendimento para companhias de rádio táxi. Normalmente as companhias (empresas ou cooperativas) utilizam sistemas de comunicação por rádio freqüência para controle das filas de atendimento, definindo a precedência (ordem) para os táxis no atendimento de chamadas feitas por clientes, através de uma central telefônica. Na grande maioria dos sistemas em utilização no momento, esse tipo de controle não é automático, pois a comunicação entre o veículo (táxi) e a central é feita verbalmente através de um rádio (entre um operador e um taxista), gerando alguns problemas, como por exemplo, a definição de precedências e prioridades do atendimento que são feitos manualmente (dependência de pessoas). Outra desvantagem está relacionada aos problemas de transmissão no sistema de rádio. Em determinadas regiões (dependendo do tipo de relevo) o sinal apresenta problemas e distorções, dificultando a troca de informações entre o taxista e a central.

1.2 Objetivos Gerais

Considerando todos os problemas acima descritos, o presente trabalho propõe a concepção, o projeto e a implementação de um sistema embutido para gerenciamento de táxis, baseado no posicionamento das unidades em áreas ou regiões demarcadas, possibilitando troca de informações entre a central e o taxista, com o objetivo de atender chamadas ou requisições feitas por clientes de uma maneira automática. A proposta baseia-se na construção de um dispositivo móvel utilizando um sistema embutido para comunicação entre os táxis e a central de atendimento, sem a necessidade de um dispositivo GPS (a responsabilidade do

posicionamento em determinada área está associada ao taxista). Essa proposta também detalha a construção de um sistema no lado da central (servidor) para controle automático das prioridades e precedências de atendimento. O sistema proposto neste trabalho tem como objetivo substituir o modelo de operação e os sistemas atuais de algumas companhias de táxi.

Para implementação do projeto, serão utilizados alguns equipamentos e dispositivos pré-existentes que utilizam tecnologia embarcada (sistemas embutidos) com o propósito de implementar os protocolos necessários para este tipo de comunicação. Outra motivação importante nesta implementação é buscar alternativas para minimizar os custos dos dispositivos móveis e a complexidade dos sistemas de controle das filas baseados em dados georeferenciados.

1.3 Abordagem Metodológica

Por se tratar de uma proposta prática, com possíveis aplicações corporativas em empresas com necessidade de gerenciamento de posicionamento de unidades móveis (táxis), a abordagem deste trabalho consiste no desenvolvimento de um modelo de operação para uma cooperativa de táxis, que substitua o modelo de operação baseado em comunicação por rádio. Desta forma simula-se a operação de um sistema real, propiciando um ambiente para realização de testes.

Com o objetivo de oferecer fundamentação teórica e conceitual ao assunto, recorreu-se às pesquisas em s*ites*, *white papers,* livros e monografias correlatas cujas referências encontram-se no final desta dissertação.

1.4 Organização do Trabalho

Este capítulo apresentou a introdução e os objetivos desta dissertação. Fez-se uma breve apresentação do problema de gerenciar serviços de táxis de uma maneira automática e apresentou-se a hipótese para solução deste problema através de um dispositivo móvel especialista, sem a necessidade do GPS e conjunto com um sistema central para controle das chamadas e definição das prioridades de atendimento.

O capítulo 2 apresenta uma revisão bibliográfica, objetivando verificar o estado da arte de sistemas que possuem alguma relação com o tema proposto, encerrando com algumas conclusões tiradas a partir desta revisão bibliográfica.

O capítulo 3 apresenta uma breve revisão das tecnologias utilizadas na implementação do sistema proposto. Discute-se a utilização da linguagem Java em conjunto com sistemas embutidos. Também são abordados alguns aspectos importantes sobre uma plataforma de desenvolvimento para dispositivos móveis chamada TINI. Em seguida, é apresentada uma visão geral de tecnologias sem fio com possibilidade de serem utilizadas pelo sistema embutido do veículo para mantê-lo em contato com a central de gerenciamento (base). O capítulo se encerra apresentando algumas conclusões sobre esta revisão de tecnologias.

O capítulo 4 apresenta a solução proposta para o problema apresentado e descreve os componentes implementados no sistema. Este capítulo detalha a implementação do sistema embutido (veículo) e os sistemas auxiliares que ficam do lado servidor (central), bem como todos os programas e algoritmos envolvidos na solução.

O capítulo 5 aborda e analisa os resultados obtidos através de uma simulação, detalhando todos os componentes envolvidos nesta simulação e os resultados encontrados. O capítulo é concluído apresentando algumas conclusões a respeito dos resultados encontrados.

O último capítulo apresenta as principais conclusões desta dissertação, menciona as principais dificuldades encontradas ao longo do desenvolvimento e encerra a dissertação apresentando algumas sugestões de trabalhos futuros.

Os apêndices apresentam:

Apêndice A: Detalhamento dos sinais dos pinos de um conector DB-9

Os anexos apresentam:

Anexo A: Esquema Elétrico da Placa Base E10 (Socket Board)

2 REVISÃO BIBLIOGRÁFICA

2.1 Introdução

Esta revisão bibliográfica aborda artigos com alguma correlação com o tema proposto e faz uma análise dos mesmos, buscando descrever o estado da arte e apresentar as diversas soluções encontradas até o momento, discutindo suas vantagens e limitações.

Como vários artigos utilizam uma plataforma de *hardware* para desenvolvimento de sistemas embutidos chamada TINI [DS80C, 2005] [TINIGPS, 2005], apresenta-se aqui uma breve descrição desta plataforma. Um detalhamento mais aprofundado sobre esta plataforma de hardware é feito no capítulo 3.

TINI *(Tiny Internet Interface)* é uma plataforma de hardware para auxiliar a implementação de sistemas embutidos. Consiste de um conjunto de circuitos integrados e de um ambiente de programação baseado na linguagem Java. Os componentes principais desta plataforma são um microcontrolador para execução de código Java (compilado especificamente para a plataforma), memória RAM, FLASH, interface *Ethernet*, interfaces RS232 para comunicação serial e interface *1-Wire* [WIRE, 2006], tudo isto em uma placa com dimensões muito pequenas, tornando-a atrativa para utilização em sistemas embutidos. A plataforma TINI possui também um pequeno sistema operacional para auxiliar a carga e execução dos programas construídos utilizando a linguagem de programação Java.

2.2 Artigos Revisados

Em [DS80C, 2005], é apresentado um sistema de coleta de dados sobre a umidade de um ambiente baseado em um dispositivo embutido desenvolvido sobre a plataforma TINI [TINI, 2005]. Observa-se que toda a comunicação remota efetuada entre o dispositivo e o servidor (central) é feita via uma conexão discada utilizando um telefone celular. Através desta conexão, o dispositivo móvel informa à central (servidor) os dados relativos à umidade do ambiente, utilizando um sensor ligado ao dispositivo.

Esta solução é adequada para a coleta de umidade, pois as informações não necessitam serem coletadas em intervalos curtos (poderia ser utilizado intervalo de horas, por exemplo) e a comunicação discada atende plenamente.

Em [TINIGPS, 2005] é descrito um dispositivo móvel construído sobre a plataforma TINI para registro de informações relativas ao posicionamento geográfico utilizando um GPS. O sistema obtém as informações das coordenadas georeferenciadas através do GPS, registra as informações em arquivos de *log*, e disponibiliza o acesso a essas informações via WEB, através de um servidor. As informações são apresentadas através de uma página HTML *(Hypertext Markup Language)*, conforme ilustra a Tabela 1.

Tabela 1 – Representação dos dados do GPS registrados via HTML do dispositivo móvel.

Local Date	9/18/2001	Local time	1:32:4
GPS Date	9/18/2001	GPS time	1:33:31
GPS Solution	**Valid**, 3D		
Position	30.436795N 97.794306W	Altitude	300m
Heading	0.0	Speed	0.000
	Position	Vertical	Horizontal
DOP	1.57	1.33	0.83999997
SV PRN	SNR	Elevation	Azimuth
27	**38 dB**	**72**	**6**
8	**32 dB**	**48**	**327**
28	Not Tracking	46	290
13	**40 dB**	**45**	**195**
31	Not Tracking	44	49
2	Not Tracking	27	276
11	**37 dB**	**23**	**130**
1	**30 dB**	**16**	**167**
3	Not Tracking	10	41

A proposta deste trabalho é efetuar o gerenciamento da posição de veículos de acordo com a sua posição física, sem a necessidade de integração com um dispositivo GPS. Em [MTC400, 2006] é apresentado um dispositivo móvel baseado na utilização de GPS para

rastreamento de frotas de veículos chamado MTC400. Esse dispositivo é oferecido comercialmente pela empresa *M2M Solutions*, e já está sendo utilizado por quatro empresas de transporte coletivo do Rio de Janeiro. Esse sistema de rastreamento permite à empresa o acompanhamento de eventos nos veículos em intervalos de tempo relativamente curtos, a criação de mapas digitais, a geração de relatórios de viagens e a comunicação em tempo real.

O MTC400 possui um receptor GPS (*Global Positioning System*) interno, um dispositivo para conexão GSM/GPRS, entrada e saída para conexão com sensores, memória *flash* para armazenamento de posições do veículo, odômetro, medidor de temperatura interna e sensores de tensão. Esse dispositivo é capaz de receber comandos de qualquer telefone, não só através de GPRS, mas também através do canal de voz, SMS ou CSD (*Dial-Up*). Os dados captados pela rede de satélites são enviados para a central de informações da empresa e analisados por seus técnicos. Quando um problema é identificado, o profissional responsável é acionado para tomar algum tipo de decisão. Infelizmente não temos dados a respeito do *hardware* deste sistema.

Um exemplo de uso é o de um motorista de uma companhia de ônibus que corre demais e se aproxima além do desejável de um outro ônibus que partiu antes do terminal. Ao identificar o problema, a central de controle avisa a um fiscal desta linha, que adverte o motorista na sua passagem, fazendo com que o intervalo entre as viagens retorne ao planejado. Através deste dispositivo o taxista pode, sem sair da sua central de controle, obter, transmitir e armazenar dados da rotina operacional dos veículos, tais como registro do horímetro, voltímetro, odômetro, número de passageiros, dados relativos ao itinerário, consumo, entre outros.

Em [TOPAUTO, 2006] é apresentado um dispositivo similar ao descrito em [MTC400, 2006]. Este dispositivo é conhecido como *TopAuto* e é oferecido pela empresa *AboveNet*, sendo indicado para controle de frotas de veículos. É composto de um computador veicular de bordo com recursos de comunicação de dados remota, *on-line* e *real time*, voz, celular e/ou satélite, localização e rastreamento de veículos via satélite, “interfaceamento” com módulo de segurança e telemetria para transmissão de parâmetros operacionais do veículo em modo *on-line*, provendo informações relevantes para monitoração do veículo à distância.

O *TopAuto* é uma plataforma aberta, baseada em processador INTEL e sistemas operacionais *Windows* ou *Linux*. Disponível em diversas configurações, o *TopAuto* oferece até

5 interfaces seriais, 3 interfaces USB *Host*, 1 interface PCMCIA, 1 interface Ethernet, 1 interface SDIO, 1 módulo CDMA/ GSM/ GPRS/ GPS, 1 módulo *Bluetooth*, 1 interface IrDA, 1 interface de disco IDE, saída e entrada de áudio estéreo, *display* VGA LCD TFT de 6.4" até 10.4" e saída de vídeo RGB além de biometria digital. Ele Permite o "interfaceamento" com periféricos, incluindo unidades leitoras de código de barras, RFID, cartões de pagamento ou débito (magnético, *smartcard* e *contactless*), impressoras seriais ou *bluetooth*, balanças, etc.

Figura 1 – Dispositivo móvel *TopAuto* para controle de frotas.

Neste sistema não existe a necessidade de um processamento automático de tomada de decisão (os dados são coletados e enviados para análise), semelhante à proposta apresentada em [MTC400, 2006].

Em [ITJVTINI, 2005] é apresentada uma dissertação de mestrado sobre a aplicabilidade da linguagem Java para o desenvolvimento de sistemas embutidos utilizando a plataforma de *hardware* TINI. Neste trabalho, é apresentada uma introdução a respeito da linguagem de desenvolvimento Java, um histórico a respeito da evolução desta linguagem e suas principais vantagens, como por exemplo: simplicidade, portabilidade, segurança entre outras. É implementado um sistema para monitoramento da temperatura utilizando um sensor conectado a interface 1-Wire da plataforma TINI, em conjunto com uma aplicação Web para visualização dos dados.

No trabalho proposto por esta dissertação também é discutida uma implementação de um sistema para testes sobre a plataforma de hardware TINI, utilizando uma ferramenta *open-source* para testes de *stress* chamada *JMeter*, do grupo *Apache*.

Através da ferramenta *JMeter* é possível gerar gráficos a respeito do teste de *stress* executado.,

A principal contribuição do estudo apresentado em [ITJVTINI, 2005] é mostrar as principais soluções de execução de uma aplicação Java em sistemas embutidos, além de apresentar os principais fabricantes que estão investindo em soluções Java, com o intuito de diminuir custos no desenvolvimento. Este trabalho procurou mostrar através de uma implementação real utilizando a plataforma de hardware TINI as características que fazem da linguagem Java um atrativo para os fabricantes de microcontroladores, microprocessadores e sistemas embutidos.

A conclusão da dissertação [ITJVTINI, 2005] apresenta as vantagens de se aplicar Java em sistemas embutidos, especialmente na plataforma TINI, atingindo os objetivos definidos para o trabalho em questão.

2.3 Conclusão da revisão bibliográfica

Esta revisão bibliográfica abordou artigos com alguma correlação com o tema proposto e faz uma crítica dos mesmos tendo tal tema como base. Procuraram-se artigos e trabalhos sobre utilização de Java com sistemas embutidos e artigos a respeito de gerenciamento de veículos e dispositivos móveis utilizando tecnologia de comunicação sem fio.

A revisão bibliográfica deu maior enfoque às soluções utilizando placas de dimensões reduzidas que tivessem a capacidade de rodar Java e que tivessem vários tipos de interfaces, principalmente interfaces de rede e serial. Como não existe uma grande variedade de placas com essas características, a revisão bibliográfica ficou mais limitada no aspecto da base de hardware a ser utilizada.

Na referência [TINIGPS, 2005] é descrito um dispositivo móvel construído sobre a plataforma TINI para registro de informações relativas ao posicionamento geográfico utilizando um GPS. A diferença entre o projeto [TINIGPS, 2005] e esta proposta está no modelo proposto para implementação, o qual será detalhado posteriormente.

Na referência [ITJVTINI, 2005] é apresentado um estudo sobre a aplicabilidade da linguagem Java para o desenvolvimento de sistemas embutidos utilizando a plataforma de hardware TINI, que contribuiu para implementação do sistema embutido proposto por esta dissertação.

A revisão bibliográfica apresentada focou mais as características gerais das aplicações desenvolvidas do que o detalhamento técnico interno de cada sistema, tanto de *hardware* como de *software*. Isto se deve ao fato da ausência de detalhamento técnico na documentação estudada.

3 REVISÃO DE TECNOLOGIAS UTILIZADAS NO DESENVOLVIMENTO DO SISTEMA

3.1 Sistemas Embutidos – Visão Geral

3.1.1 Introdução

A evolução da microeletrônica e o barateamento das CPUs viabilizaram o emprego de sistemas computadorizados nos diversos equipamentos. Os cérebros da maioria dos equipamentos modernos são os pequenos computadores que eles trazem embutidos. É claro que esses computadores não se parecem com o que temos sobre nossas mesas. Eles são construídos para tarefas específicas e, na grande maioria das vezes, não contêm disco rígido ou flexível, não precisam de teclado e muito menos de sistema operacional.

3.1.2 Microcontroladores

O termo controlador é usado para designar o dispositivo que controla um processo ou algum parâmetro do ambiente. O controlador de temperatura do condicionador de ar, por exemplo, liga ou desliga o compressor em função da temperatura ambiente. Antigamente, os controladores usavam lógica discreta e, por isso, tinham um tamanho que dificultava seu emprego em sistemas pequenos. Hoje em dia, usam-se os circuitos integrados microprocessados e todo o controlador cabe em uma pequena placa de circuito impresso.

O microcontrolador com o avanço da microeletrônica recebeu, dentro do seu circuito integrado, uma quantidade de recursos cada vez maior. Assim tem-se a primeira definição: Microcontrolador é um circuito integrado (CI) com alta densidade de integração que inclui, dentro do CI, a maioria dos componentes necessários para o controlador. É por isso que se utiliza o apelido: "solução com único *chip*". Existe uma quantidade expressiva de microcontroladores, porém os mais conhecidos são: 8051, 8096, 68HC705, 68HC11 e os *Pics*.

A fronteira entre as definições de microcontrolador e sistema embutido não é clara e muitas das soluções poderiam ser obtidas tanto com microcontroladores quanto com sistemas embutidos. A chave para essa diferença está no tamanho e complexidade. Os

microcontroladores são simples, enquanto que os sistemas embutidos são mais complexos e usam uma grande quantidade de chips.

3.1.3 Sistemas Embutidos - Arquitetura

De forma rápida, e abusando das palavras, diz-se que o sistema embutido [AXELSON, 2003] é um "pequeno computador" que foi embutido em um sistema maior. Pode-se ainda dizer que um sistema embutido é um sistema computacional que foi incluído em um outro sistema com a finalidade outra que a de fornecer processamento genérico.

Sistema embutido geralmente é uma solução formada de microcontrolador + software (*firmware*) dedicado e específico para desempenhar as funções operacionais de um equipamento para o qual foi projetado.

Usa-se o nome “sistema embutido” ou “sistema embarcado”, pois a solução fica escondida dentro de um dispositivo eletrônico e seu objetivo é aumentar a inteligência de um equipamento para melhor funcionalidade e eficiência [AXELSON, 2003] [EMBD, 2005].

Alguns segmentos de mercado que utilizam sistemas embutidos:

Figura 2 – Segmentos de mercado.

Está cada vez mais comum a adoção de sistemas embutidos pela área industrial, para incorporar novas tecnologias e recursos aos seus produtos com o propósito de manter uma posição favorável no mercado, que atualmente exige inovações e equipamentos inteligentes. Muitos equipamentos eletromecânicos incorporam sistemas embutidos para se tornarem mais eficientes e terem sua funcionalidade aumentada, agregando valor e apresentando um diferencial em relação aos competidores.

A chave para diferenciar um sistema embutido de um microcontrolador está no tamanho da solução. Os microcontroladores usualmente são pequenos e empregam poucos CI’s, enquanto que os sistemas embutidos são maiores e mais sofisticados, com uma quantidade maior de CI’s. Existem sistemas embutidos de todos os tamanhos: desde pequenas

placas com uma CPU 8085, uma ROM e uma RAM, até sofisticados sistemas com CPUs Pentium. Os sistemas mais simples apenas trazem um pequeno programa gravado em sua ROM, enquanto que os mais sofisticados fazem uso de BIOS e trazem um sistema operacional.

É difícil especificar a anatomia de um sistema embutido, pois ela é ditada pela aplicação, mas um típico sistema embutido normalmente apresenta: CPU, ROM/RAM, memória não-volátil, relógio temporizador, entradas e saídas.

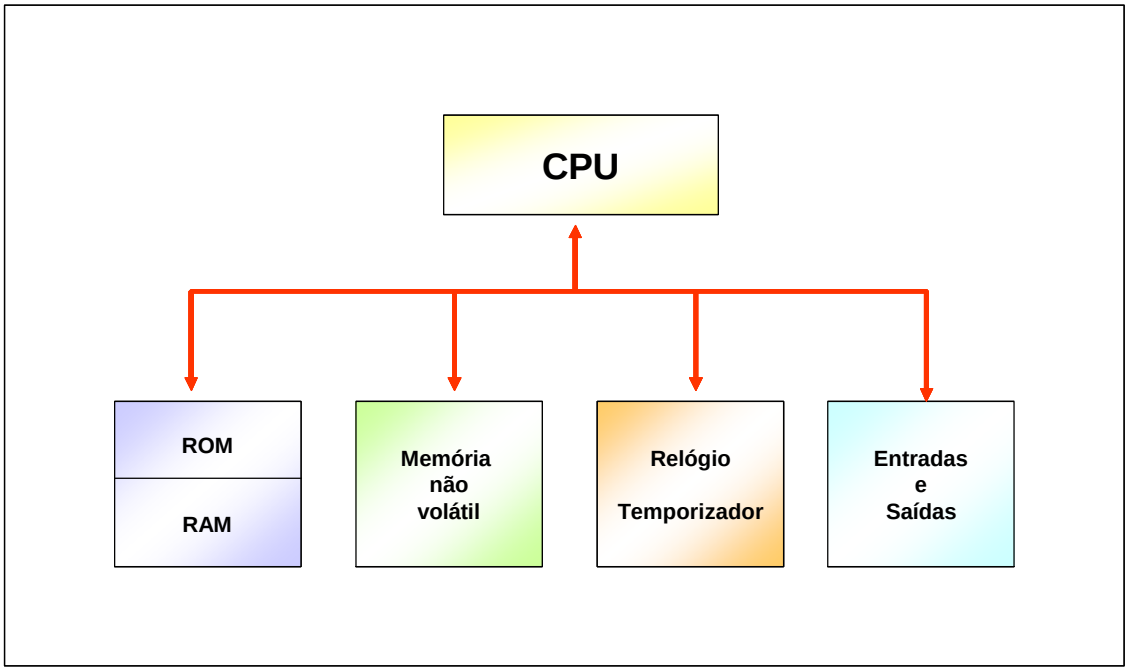


Figura 3 – Arquitetura de um sistema embutido.

3.1.4 Sistemas Embutidos - CPUs

Os sistemas embutidos classificados com "simples" lançam mão das CPUs que foram ultrapassadas pela voraz busca por desempenho. Para esse caso, citam-se as CPUs de 8 bits: 8085, Z-80 e 6.800. Existem também sistemas baseados em CPUs de 4 bits. Os sistemas classificados como "simples" também empregam as CPUs de 16 bits: 8086, 8088, 80186, 80188, 80286, 68000, 68010.

Os sistemas embutidos de "complexidade média" usam CPUs compatíveis com a família 386 e 486. Sua arquitetura é muito semelhante ou até idêntica à de um PC AT. São empregados em tarefas sofisticadas, como pontos de venda, instrumentos médicos, computadores de bordo, etc.

Os sistemas "sofisticados" lançam mão de CPUs compatíveis com a família Pentium e lembram um moderno computador, sem monitor, teclado e disco. Seu emprego está nos sistemas dedicados multimídia, na automação de centrais telefônicas, no controle de grandes plantas fabris, etc..

3.1.5 Componentes Básicos

Alguns elementos de *hardware* e *software* estão sempre presentes no desenvolvimento de sistemas embutidos. O *hardware* básico de um dispositivo embutido deve conter um processador, memória e periféricos. Dado que sistemas embutidos são geralmente microprocessados, uma prática comum entre desenvolvedores de sistemas embutidos é escolher um microprocessador específico e montar seu *hardware* em função deste. Esta escolha tem, entre outros determinantes, a disponibilidade do microprocessador no mercado e também o conjunto de bibliotecas de *software* existentes para o mesmo. A vantagem desta prática é que módulos de *software* podem ser reaproveitados entre projetos distintos.

- **Processador** – Os sistemas embutidos são geralmente microprocessados. Embora a capacidade computacional dos microprocessadores tenha aumentado bastante (a capacidade de processamento dos *microchips* dobra aproximadamente a cada 18 meses), muitos sistemas embutidos continuam sendo desenvolvidos com o uso de processadores de baixo desempenho, de 8/16 *bits*. Isto se deve ao fato de haver outros fatores determinantes na escolha do processador além do desempenho, tais como custo, consumo de potência, ferramentas de *software* disponíveis, e disponibilidade do componente no mercado, entre outras.

- **Memória** - A memória é uma das partes mais importantes do projeto de um sistema embarcado e influencia diretamente na forma como o *software* para o sistema é projetado, escrito e desenvolvido. A memória tem duas funções básicas dentro de um sistema embarcado:

  - armazenar o *software* que o sistema irá rodar – feito geralmente em memória não volátil, que mantém seu conteúdo após a ausência de uma fonte de alimentação.

- o armazenar os dados - tais como variáveis de programa e resultados intermediários, que são armazenados em memória volátil.

Muitos sistemas embutidos apresentam maior quantidade de memória não volátil em relação à volátil, dado o custo maior da memória volátil. Como resultado, o *software* para tais sistemas tem quer ser escrito de forma a minimizar os requisitos de memória volátil.

- ? **Periféricos** - um sistema embarcado comunica-se com o mundo exterior através de periféricos. Dentre os principais periféricos encontrados em um sistema embarcado, destacam-se os *displays*, *keypads*, saídas seriais, saídas binárias, temporizadores, etc.

- ? ***Software*** - O *software* de um sistema embarcado determina o que ele faz e como ele faz. Há várias formas de se desenvolver *software* para sistemas embutidos, dependendo da complexidade do mesmo, do tempo gasto e do custo da solução.

  Um desenvolvedor de *software* para sistemas embutidos geralmente tem que se preocupar não só com a funcionalidade que este sistema deve apresentar, mas também com aspectos de sincronização, escalonamento de tarefas, gerenciamento de memória, entre outros. Outra característica comum é a utilização de duas ou mais linguagens de programação, em função dos requisitos do sistema. Tais práticas aumentam o tempo de desenvolvimento do sistema, dificultam a portabilidade do código gerado, além de serem suscetíveis às falhas tanto no nível de desenvolvimento quanto em tempo de execução.

3.2 Linguagem Java

3.2.1 Introdução

A linguagem de programação Java, desenvolvida pela *Sun Microsystems*, foi projetada para ser uma linguagem de programação independente da plataforma de *hardware*, segura e poderosa o bastante para substituir o código executável nativo [HAGGAR, 2000] [JAIME, 2000].

Verificando a arquitetura da linguagem Java, nota-se que um dos aspectos mais importantes é o ambiente autônomo da máquina virtual em que as aplicações são executadas.

A linguagem foi cuidadosamente projetada de modo que a arquitetura de suporte possa ser implementada no software para as plataformas de computador existentes, ou no *hardware* personalizado para novos tipos de dispositivos. Para este trabalho, as características acima citadas contribuíram para a escolha dessa linguagem na implementação do sistema.

3.2.2 Funcionamento

Todo código gerado através da linguagem Java é compilado gerando uma linguagem intermediária denominada de *bytecode,* que é interpretado por uma máquina virtual específica do *hardware* onde será executada. Este código é compilado para um formato universal, gerando instruções para uma determinada máquina virtual - JVM *(Java Virtual Machine)* [HAGGAR, 2000].

Existem algumas plataformas de *hardware* que executam diretamente os *byte codes*. Neste caso, a linguagem Java pode ser considerada compilada.

O código de bytes (*bytecode*) gerado na compilação (também chamado *j-code) é* executado por um interpretador Java em tempo de execução (*runtime*). O sistema de *runtime* realiza todas as atividades normais de um processador real, mas faz isso em um ambiente seguro e virtual. Ele cria e manipula tipos de dados primitivos, carrega blocos de código, e transforma os trechos de códigos mais utilizados em instruções nativas da plataforma, aumentando assim o desempenho do programa executado. Ainda mais importante, ele faz tudo isso de acordo com uma especificação aberta estritamente definida, que pode ser implementada por qualquer plataforma de *hardware*. Juntas, a máquina virtual e a definição da linguagem oferecem uma especificação completa para o ambiente de execução.

O interpretador Java é relativamente leve e pequeno, podendo ser implementado no formato desejado para uma plataforma em particular.

3.2.3 Comparação entre Java e as outras linguagens

Influenciada diretamente por C++ e Eiffel, a linguagem Java segue a grande tendência das linguagens de programação nas décadas de 80 e 90 que foi de linguagens orientadas a objeto. Neste período, linguagens como Pascal, Ada, Lisp e Cobol ganharam versões orientadas a objetos. Trata-se de uma linguagem relativamente nova (apresentada pela *Sun Microsystems* em 1995), mas baseia-se em muitos anos de experiência de programação com

outras linguagens. Assim, muita coisa pode ser dita na comparação e no contraste com outras linguagens.

3.2.4 Arquitetura da Linguagem Java

Sistemas desenvolvidos em Java geralmente consistem de várias partes: um ambiente, a linguagem, interfaces de programação de aplicação (APIs) e várias bibliotecas de classes.

A Fase 1 consiste em editar um arquivo de acordo com as regras da linguagem Java e o mesmo deve ser salvo com a extensão *.java.*

A Fase 2 consiste na compilação, ou seja, o programa escrito na fase 1 é traduzido para *bytecodes*. Na mesma fase é gerado um arquivo *.class* que será interpretado durante a fase de execução.

A Fase 3 é chamada de carga. Nesta fase os arquivos *.class* são carregados para a memória.

Existem dois tipos de programas desenvolvidos em Java: os aplicativos e os *applets*. Os aplicativos são geralmente executados em um computador local. Os *applets* são pequenos programas normalmente armazenados em um computador remoto. Os *applets* são carregados por navegadores que acessam a rede utilizando protocolo HTTP.

Antes do *bytecode* de um *applet* serem executados pelo interpretador Java (embutido no navegador), eles são verificados como descrito na Fase 4 para assegurar que o *bytecode* carregado é válido e não viola as restrições de segurança da linguagem Java.

Na fase 5, o computador, sob controle da Unidade Central de Processamento (CPU), interpreta o programa, em *bytecode* por vez, realizando assim a ação especificada.

3.2.5 Máquina Virtual Java

O coração da linguagem Java é a máquina virtual (*Java Virtual Machine*) ou JVM, que faz a plataforma alcançar três características importantes: independência de plataforma, segurança e mobilidade.

A JVM é um computador abstrato. Esta especificação define todas as características que uma JVM deve possuir. A especificação é flexível o suficiente para permitir que uma JVM seja implementada completamente em *software* ou em vários níveis no *hardware*. Esta

flexibilidade da especificação permite que uma JVM seja implementada em uma grande variedade de computadores e dispositivos.

A principal função da JVM é carregar os arquivos *.class* e executar o *bytecode* contido nas classes.

A Figura 4 ilustra uma JVM contendo um carregador de classes, tanto da aplicação como das APIs. Somente as APIs utilizadas pela aplicação são carregadas pela máquina virtual.

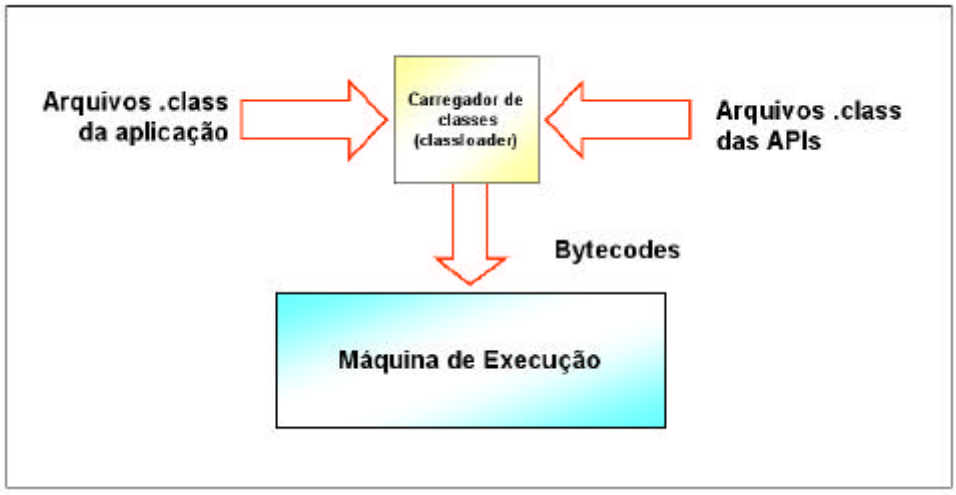


Figura 4 – Carregador de classes Java (Aplicação Java + API).

O tempo de vida de uma JVM é limitado, ou seja, nasce quando a aplicação é iniciada e morre quando a aplicação é finalizada.

Na especificação de uma JVM, o ambiente de uma máquina virtual instanciada é descrito em termos de subsistemas, áreas de memória, tipo de dados e instruções. Estes componentes descrevem uma arquitetura interna de uma JVM abstrata. O objetivo destes componentes não é impor uma arquitetura abstrata para as implementações, e sim definir o requisito para o ambiente de qualquer implementação de uma JVM em termos de componentes abstratos e suas interações.

Quando a JVM executa um programa, necessita de memória para armazenamento, principalmente para armazenar informações extraídas dos arquivos *.class* carregados, das instâncias dos objetos do programa, dos parâmetros dos métodos, dos retornos de valores, das

variáveis locais e do cálculo de resultados intermediários. A JVM organiza a memória necessária para executar um programa dentro da área de dados durante a execução.

Cada instância de uma JVM possui uma área de código (métodos) e uma área de dados que são compartilhados por todas as *threads* da JVM.

As seções a seguir detalham cada bloco da arquitetura de uma JVM.

3.2.5.1 Carregador de Classes

Conceitualmente, se a principal tarefa de uma JVM é carregar arquivos .class e executar os códigos, o trabalho do carregador de classe é localizar e carregar as classes dentro da JVM que são referenciadas em tempo de execução.

Um sistema de classe estática, onde todas as classes são carregadas em tempo de compilação, não são suportadas pela especificação da JVM. A especificação defende que a JVM deve procurar dinamicamente e carregar em tempo de execução as novas classes. O carregador de classe deve também ser capaz de manipular a possibilidade de não existência ou falta de classes que foram instanciadas ou chamadas em tempo de execução, bem como verificar a validade de classes e a conexão entre elas em tempo de execução.

3.2.5.2 Área de Código

Na instância de uma JVM as informações sobre tipos de dados carregados são mantidas em uma área lógica de memória chamada de área de métodos. Quando uma JVM carrega um tipo de dado, ela usa um carregador de classe para localizar o arquivo *.class* apropriado. O carregador faz a leitura deste arquivo e passa para a JVM, que extrai a informação sobre o tipo de dado binário e armazena a informação na área de métodos.

3.2.5.3 Área de Dados

Enquanto uma classe é instanciada ou um vetor é criado na execução de uma aplicação Java, a memória para os novos objetos é alocada na área de dados. Existe somente uma área de dados em cada instância de uma JVM, onde as *threads* da instância compartilham a mesma área de dados. Não existem maneiras de duas aplicações Java conflitarem com a mesma área de dados.

A JVM possui instruções que alocam memória na área de dados, mas não possui instruções de liberação desta memória. Como não é possível explicitar a liberação da memória

no código fonte ou no *bytecode* gerado, a JVM responsabiliza-se por decidir onde e quando liberar a memória ocupada pelos objetos que não estão sendo referenciados pela aplicação. A JVM utiliza coletor de lixo para gerenciar a área de dados.

3.2.5.4 O Contador de Programa

Cada *thread* possui seu próprio registrador do contador de programa ou somente contador de programa, que é criado quando a *thread* é iniciada. O contador mantém um ponteiro nativo e um endereço de retorno. Enquanto a *thread* executa um trecho de código, o contador mantém o endereço da instrução atual que está sendo executada pela *thread*.

3.2.5.5 A Máquina de Execução

O núcleo de qualquer JVM é a máquina de execução. Na especificação o ambiente de execução é definido em termos de conjunto de instruções. Para cada instrução, a especificação descreve em detalhes o que uma implementação deve fazer quando encontrar a instrução e como executar o *bytecode*. As implementações podem ser interpretadas, compiladas, executadas de forma nativa, usando uma combinação ou utilizando novas técnicas.

3.2.5.6 Coletor de Lixo

O coletor de lixo (*Garbage Collector* ou simplesmente GC) é um processo usado no gerenciamento de memória nos sistemas computacionais. Com este recurso é possível recuperar a zona de memória que um programa não utiliza mais. Quando isto não ocorre pode acontecer a chamada perda de memória (*memory leaks*), um erro comum que pode levar ao término não desejado do programa em execução por esgotamento da memória livre. Na linguagem Java, ao contrário do que ocorre em linguagens como C++, o coletor de lixo foi implementado de maneira que seja usado automaticamente: assim que o GC encontrar um objeto para o qual não existe referência, a área de memória onde reside tal objeto é marcada para “desalocação”.

3.2.6 Plataformas de Desenvolvimento Java

As plataformas Java desenvolvida pela *Sun Microsystems* correspondem somente ao software que é executado no nível acima das plataformas baseadas em *hardware*. A tecnologia Java tem crescido para abranger todos os campos das plataformas especializadas listadas a seguir. Cada plataforma está baseada na JVM destinada para o hardware utilizado.

- ***Java 2 Platform, Standard Edition* (J2SE)** – especificação da linguagem Java que contém APIs com as funções básicas, tais como: I/O (entrada / saída), *threads*, redes e conectividade com banco de dados.

- ***Java 2 Platform, Enterprise Edition* (J2EE)** – define o padrão para desenvolvimento de aplicações corporativas. É baseado na plataforma J2SE e oferece serviços adicionais, ferramentas e APIs. [CATTEL, 2001]

- ***Java Card*** – adapta a plataforma Java para *smart cards* e outros dispositivos inteligentes com memória e que possuem velocidade de processamento limitada.

- ***Java 2 Platform, Micro Editon* (J2ME)** – é a segunda revolução na curta história de tecnologia Java. A revolução deu-se com a inserção de tecnologia Java em dispositivos móveis, aliada a explosão de mercado desses dispositivos, principalmente telefones celulares.

3.2.6.1 J2ME – Visão Geral

A Arquitetura J2ME é dividida entre configurações, perfis (*profiles*) e APIs opcionais.

As **configurações** definem o mínimo que um desenvolvedor pode esperar do dispositivo, classificando-o por capacidade de memória e processamento. É especificada uma JVM que pode ser portada entre dispositivos e também determina um subconjunto de APIs da J2SE a serem utilizadas além de outras APIs adicionais necessárias. Existem dois tipos de configurações:

- **Configuração de Dispositivo Conectado (CDC) -** é baseada na JVM convencional, definindo um ambiente como um conjunto rico de recursos, semelhantes aos encontrados em computadores para fins gerais. Destina-se a dispositivos sem fio de alta capacidade, *top boxes*, sistemas automotivos e outras plataformas que possuam pelo menos alguns MB de memória disponível.

- **Configuração de Dispositivo Conectado Limitado (CLDC)** – consiste em uma JVM reduzida e um conjunto de classes mais apropriado para dispositivos pequenos e com limitações de desempenho e memória. Esta configuração é destinada para dispositivos sem fio menores possuindo geralmente entre 160 KB e 512 KB de memória disponíveis para aplicações desenvolvidas em Java, uma conexão de rede

limitada, intermitente e provavelmente lenta. É um ambiente de desenvolvimento para telefones celulares, *pagers*, PDAs, e outros.

Os **Perfis** são conjuntos de APIs que suplementam as configurações fornecendo funcionalidades para um determinado tipo de dispositivo ou mercado vertical.

O perfil utilizado em celulares é o *Perfil de Dispositivo com Informação Móvel (MIDP),* que exige 128 KB de memória não volátil para implementação, 32 KB de memória não volátil para a área de dados em tempo de execução e 8K para persistência de dados, uma tela de pelo menos 96 x 54 *pixels*, algum tipo de entrada de dados (teclado, teclas de telefone, ou *touch screen*, e conexões de rede, possivelmente intermitentes). As aplicações desenvolvidas com MIDP são chamadas de *Midlets*.

As APIs opcionais são funcionalidades adicionais específicas que não serão encontradas em todos os dispositivos de uma determinada configuração ou perfil, mas importantes o suficiente para serem padronizadas. AS APIs opcionais mais conhecidas são listadas abaixo:

- **WMA *(Wireless Message API),*** que permite aos aplicativos J2ME manipular mensagens SMS (*Short Message Service*).

- **MMAPI *(Mobile Media API),*** que adiciona controle de mídia aos programas J2ME nos dispositivos que permitem.

Juntos, configurações, perfis e APIs adicionais formam uma pilha que define o que está disponível para o desenvolvedor em alguns dos dispositivos ou classe de dispositivos fornecendo o escopo de recursos que podem ser acessados pelos aplicativos escritos para ele.

3.2.7 Java e Sistemas Embutidos

A quantidade e diversidade de dispositivos pessoais fizeram com que a *Sun Microsystems* entrasse no mercado de sistemas embutidos e desenvolvesse uma série de tecnologias que facilitassem o desenvolvimento de sistemas embutidos e serviços para uma ampla classe de dispositivos de consumo.

Java apresenta várias características que a tornam atrativa para o desenvolvimento de sistemas embutidos, tais como portabilidade, segurança, simplicidade, conectividade, desenvolvimento rápido de aplicações, entre outras.

A diversidade de sistemas embutidos existentes torna inviável a existência de uma única plataforma Java que atenda aos seus variados requisitos. Para garantir o ideal “escreva uma vez, execute em qualquer lugar”, a *Sun Microsystems* tem definido várias especificações para serem usadas no universo dos dispositivos embutidos, que ou são mais simples que a versão padrão de Java, como a *PersonalJava* e a *EmbeddedJava* , ou são específicas para uma categoria de dispositivos, como a *Java Card* , a *Java TV* e a *JavaPhone*.

Em 1999, a *Sun Microsystems* lançou uma versão da sua plataforma para sistemas embutidos, e que abrange várias classes de dispositivos. Esta plataforma é a *Java 2 Micro Edition (J2ME)* , desenvolvida para atender da melhor forma possível as necessidades específicas de cada tipo de dispositivo.

Existem outros dispositivos que rodam programas escritos em Java, que não implementam a especificação J2ME. Esses dispositivos têm características específicas, dependendo do fornecedor. Normalmente esses dispositivos são extremamente restritos, com limitações de memória, interfaces e APIS. Seguem abaixo alguns exemplos de dispositivos Java que não utilizam J2ME:

1. **SaJe** – Uma das primeiras placas a executar código Java nativo (*bytecode* original Java). Características: Fabricante *Systronix*, 512KB de memória RAM e 2MB de memória *Flash*, 4.00 x 6.40 polegadas, duas UARTs integradas, interface 1-Wire e interface E*thernet* RJ45. Tipicamente executa 25 milhões de *bytecodes* Java por segundo. Preço: $399,00 (Fev, 2006).

Figura 5 – SaJe Java Controller – Placa para execução de código Java nativo.

2. **JStik** – Outra placa para execução de código Java nativo (executado diretamente pelo microprocessador, sem interpretação). Características: Fabricante *Systronix*, 2MB de memória RAM e 4MB de memória *Flash*, 3.00 x 2.6 polegadas, duas UARTs integradas, interface 1-Wire e interface *ethernet* RJ45, Led de status. Tipicamente executa de 15 a 20 milhões de *bytecodes* Java por segundo, a 103 MHz. Preço $350,00 (Fev, 2006).

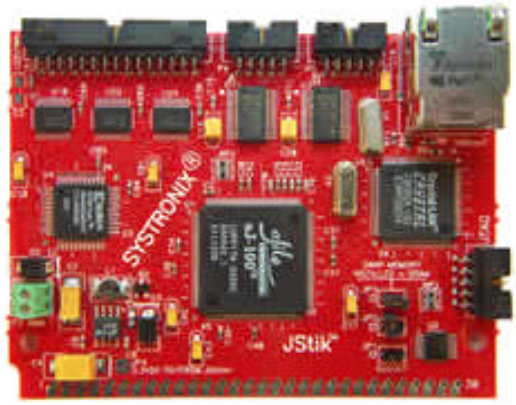


Figura 6 – JStik Java Controller – Outra placa para execução de código Java nativo.

3. **TStik** –Placa para execução de código Java específico (compilado para placa, não é o *bytecode* nativo). Características: Fabricante *Systronix*, 4.00 x 1.25 polegadas, 1MB de memória RAM e 2 MB de memória *Flash*, duas UARTs integradas, interface 1-Wire e interface E*thernet* RJ45. Preço $99,00 (Fev, 2006).

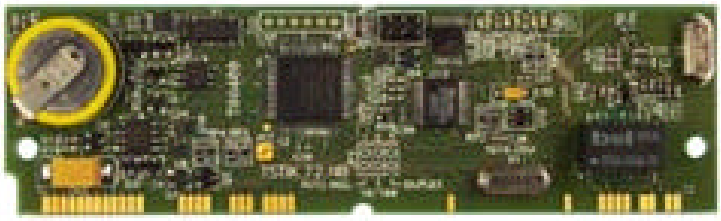


Figura 7 – Placa TStik da Sistronix – Execução de código Java compilado para a plataforma.

4. **TINI390** – Placa para execução de código Java específico (compilado para placa, não é o *bytecode* nativo). Esta placa foi uma das precursoras na execução

de código Java em circuitos integrados. Características: Fabricante *Dallas Semiconductors*, 4.00 x 1.25 polegadas, 1MB de memória RAM e 2 MB de memória *Flash*, duas UARTs integradas, interface 1-Wire e interface E*thernet* RJ45. Preço $110,00 (Maio, 2005).

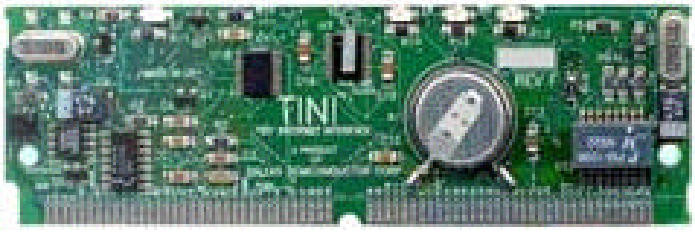


Figura 8 – Placa TINI da *Dallas Semiconductors* (Microcontrolador DS80C390).

3.3 Plataforma TINI (Placa + API + Sistema operacional)

3.3.1 Introdução

Atualmente existem várias plataformas de desenvolvimento para sistemas embutidos, utilizando processadores específicos e microcontroladores. No entanto, a oferta de sistemas de baixo custo utilizando linguagem Java ainda é bastante limitada [ITJVTINI, 2006] [LOOMIS, 2001].

Existem algumas plataformas de desenvolvimento para sistemas embutidos que facilitam a prototipação e testes utilizando a linguagem Java. Uma dessas plataformas para desenvolvimento de sistemas embutidos é a plataforma TINI *(Tiny Internet Interface)* [TINI, 2005] [LOOMIS, 2001]. Essa plataforma está centrada na utilização de um microcontrolador que executa código Java compilado especificamente para a arquitetura de *hardware* da placa. Desenvolvida pela *Dallas Semiconductor,* a plataforma TINI provê uma maneira simples e flexível para projetar uma variedade de dispositivos de *hardware* que conectam-se diretamente a redes corporativas, à Internet, a interfaces seriais, a interfaces de infravermelho e a periféricos entre outros.

A plataforma TINI é uma combinação de um conjunto de circuitos integrados e um ambiente de programação em tempo real baseado em Java. O objetivo inicial da plataforma era fazer com que todos os sensores e atuadores de um sistema de automação mantivessem uma comunicação. A combinação entre dispositivos de entrada e saída, protocolo de redes,

protocolo de controle de transmissão / protocolo de Internet (TCP/IP) e o ambiente de programação Java tornou possível não somente o controle local, mais um controle global dos dispositivos baseados na plataforma TINI. A capacidade de comunicação permite a qualquer dispositivo associado à interação com sistemas remotos e seus respectivos usuários através de aplicações de comunicação padrão, tal como navegadores.

Esta combinação de características implicou na escolha da plataforma TINI para a implementação do presente trabalho.

3.3.2 TINI - Especificações Técnicas

O elemento central da plataforma de desenvolvimento TINI é uma placa denominada TBM390 *(TINI Board Model 390)* ou simplesmente TINI390. Esta placa contém as seguintes características:

- Microcontrolador DS80C390 (JAVA Runtime);
- 512 kilobytes de memória flash (*critical system code*);
- 512 kilobytes SRAM, expansível para 1 megabyte;
- Controlador 10Base-T Ethernet;
- Realm-time clock;
- Dual 1-Wire net interface;
- Dual CAN controllers;
- Dual serial port (RS-232);
- Power jack +5V;

O diagrama em blocos com os principais componentes da placa TINI390 é apresentado na Figura 9.

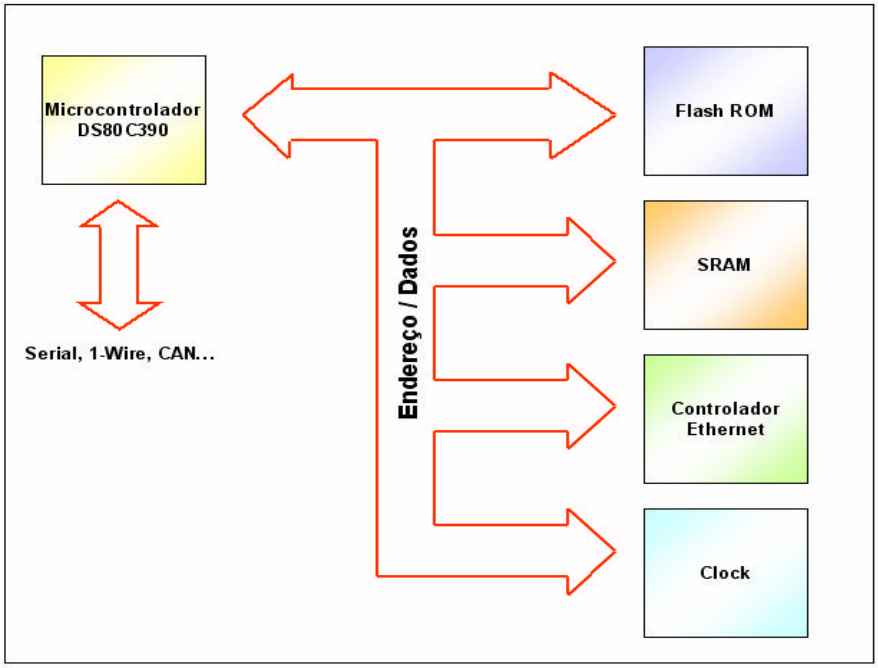


Figura 9 – Diagrama de blocos da placa TINI390.

O DS80C390 é um microcontrolador compatível com o conjunto de instruções do 8051. As instruções são executadas em quatro ciclos de máquina, ou seja, a velocidade de processamento é três vezes maior que no 8051, que utiliza doze ciclos de máquina.

Abaixo as principais características do DS80C390:

- Compatível com 82C51;
- 4KB de memória RAM interna utilizadas para dados, programas e ponteiros de memória.;
- Arquitetura de memória otimizada permitindo endereçamento de memória externa acima de 4 MB;
- 2 canais de controle CAN 2.0B;
- 2 portas seriais;
- 16 fontes de interrupção com 6 externas.

O mapa de memória da plataforma é ilustrado na figura abaixo.

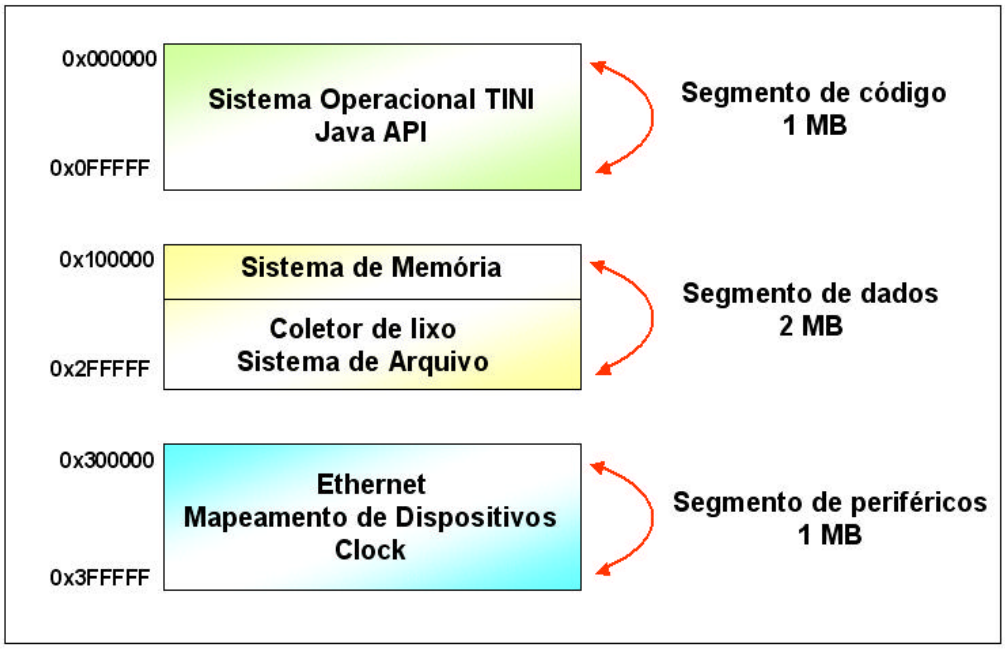


Figura 10 – Mapa de memória da plataforma TINI.

Existe também uma área denominada PCE, que pode ser utilizada para conectar memórias externas e outros dispositivos conectados diretamente no barramento de dados e endereços do microcontrolador. É recomendado que os periféricos sejam mapeados no segmento de periféricos, pois o controlador pode acessá-los de forma mais eficiente.

Todos os componentes que compõe a plataforma estão agrupados em uma placa (*TINI board*), auxiliada por uma placa base *(TINI Socket Board)*, que permite o "interfaceamento" com periféricos externos.

A placa auxiliar *TINI Socket Board* é uma implementação de referência produzida pela *Dallas Semiconductor* e serve como apoio ao ambiente de desenvolvimento de sistemas embutidos. Com ela é possível desenvolver um sistema utilizando a linguagem Java e um conjunto de APIs *(Application Programming Interface)* para execução em um dispositivo de *hardware*.

As figuras abaixo ilustram o *hardware* da plataforma TINI.

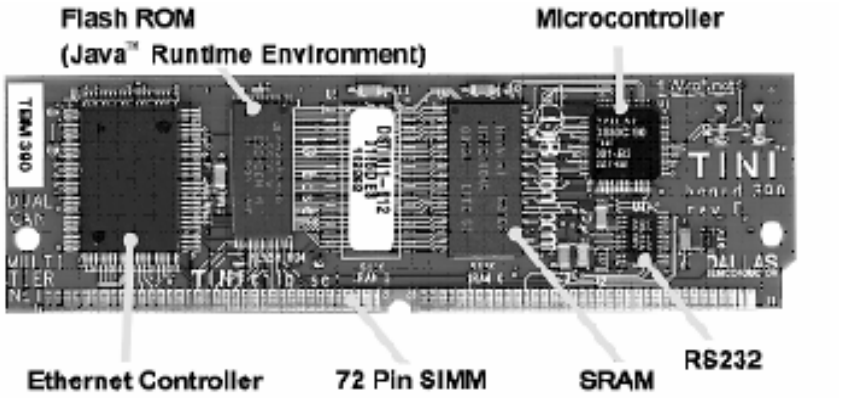


Figura 11 – TBM 390 *(Dallas Semiconductors)* – Visão frontal.

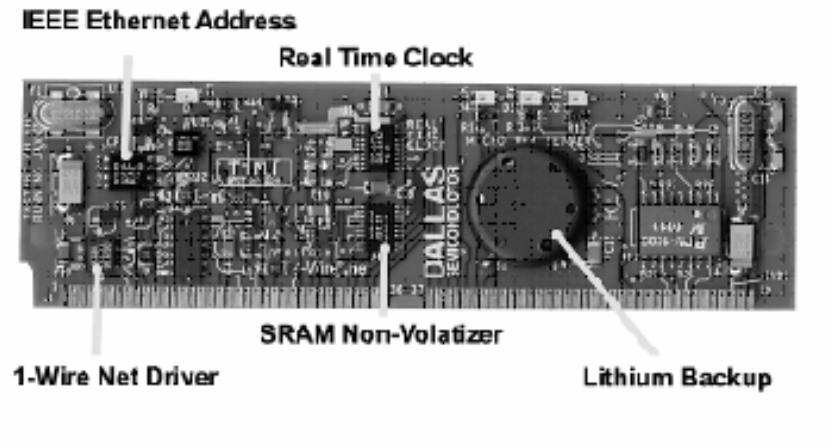


Figura 12 – TBM 390 *(Dallas Semiconductors)* – Visão traseira.

Os esquemas elétricos da placa base (*TINI* S*ocket Board*) da plataforma TINI, que são fornecidos pela *Dallas Semiconductor*, são apresentados no Anexo A: Esquema Elétrico da Placa Base E10 (Socket Board).

Como já descrito, para o desenvolvimento com a plataforma TINI são necessárias algumas placas auxiliares para acesso a determinadas interfaces: circuitos para controle das interfaces *Ethernet, Serial, 1Wire, CAN* entre outras. A placa TBM390 serve como processador central do dispositivo em questão, mas são necessários outros circuitos para controle dessas interfaces. Para isso, existem algumas placas para auxiliar a interface com outros dispositivos, conhecidas como *Socket Boards*. Segue abaixo alguns exemplos de placas *Socket Boards*:

E10 Socket Board.

Esta placa foi desenvolvida pela *Dallas Semiconductor* para auxiliar o desenvolvimento e a prototipação de dispositivos utilizando a plataforma de desenvolvimento TINI.

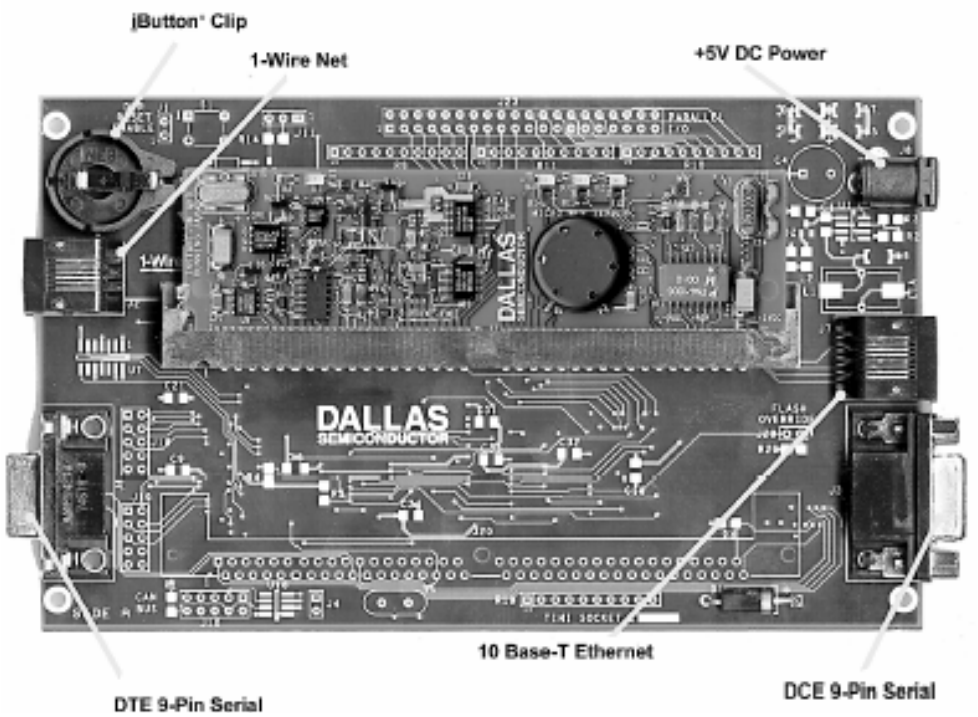


Figura 13 – Placa *E10 TINI Socket Board* auxiliar para conexões com interfaces.

Características técnicas da placa E10 *Socket Board*:

- 72-pin SIMM - Conector para TINI TBM390;
- 9-pin *female* DB9 *Connector* 9-pin *male* DB9 *Connector* - Porta serial para comunicação;
- RJ45 - Cabo 10Base-T *Ethernet* ;
- RJ11 - Comunicação 1-Wire net;
- iButton Clip;
- Power jack - +5V DC;

Placa STEP Socket Board

A empresa *Systronix* também desenvolveu uma placa auxiliar utilizando todas as interfaces disponíveis da placa TINI, chamada STEP. Esta placa apresenta uma série de interfaces, como por exemplo, possibilidade de conexão com teclado, *display* e infravermelho.

Figura 14 – Placa STEP (Outra placa auxiliar para TINI desenvolvida pela *Systronix*).

Características técnicas da placa STEP *Socket Board*:

- 72-pin SIMM - *Conector* para TINI TBM390;
- 9-pin *female* DB9 *Connector* 9-pin *male* DB9 *Connector* - Porta serial para comunicação;
- RJ45 - Cabo 10Base-T *ethernet* ;
- RJ11 - Comunicação 1-Wire net;
- *iButton Clip*;
- *Power jack* – entre +8 e +24V DC;
- SBX2 *Connector*;

A grande diferença entre a placa STEP e a E10 é a possibilidade de expansão através do conector SBX. Esta entrada SBX permite a conexão da placa STEP com outra placa auxiliar chamada *SBX Board*. Com essa placa SBX é possível conectar *Displays* e *Keypads* na placa STEP, disponibilizando esses periféricos para endereçamento via programação.

Placa SBX Socket Board Extension

A placa SBX permite a integração de periféricos à placa STEP, através de um conector de extensão, tais como teclado e *display*. Esta placa também é fabricada pela *Systronix* e serve para auxiliar a prototipação de dispositivos com essas características.

Figura 15 – Placa de extensão SBX2 para conexão com periféricos (Ex: teclado e *display*).

Principais características da placa SBX2:

- *Fast parallel* LCD interface;
- 4x5 *keypad* interface;
- 24 bits, *high-current open-drain bidirectional* I/O;
- RS232/485/IrDA serial I/O;

Exemplo abaixo um exemplo de *display* e *keypad* que podem ser conectados a placa SBX:

Figura 16 – *Display* 20x4 (conectado a placa STEP via extensão SBX2).

Teclado 4x4 tipo membrana (4 colunas por 4 linhas)

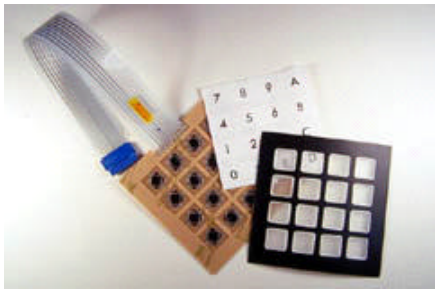


Figura 17 – Teclado tipo membrana (conectado a placa STEP via extensão SBX2).

Para implementação do trabalho proposto nesta dissertação utilizamos a interface SBX para conexão de um d*isplay* 20x4 e um *Keypad* 4x4, conforme detalhado no capítulo 4.

3.3.3 TINI - Software

O ambiente de execução das aplicações desenvolvidas pode ser dividido em duas categorias:

- *Código nativo executado diretamente pelo microcontrolador;*
- *API´s interpretadas como bytecodes pela JVM.*

A figura abaixo ilustra o ambiente de execução.

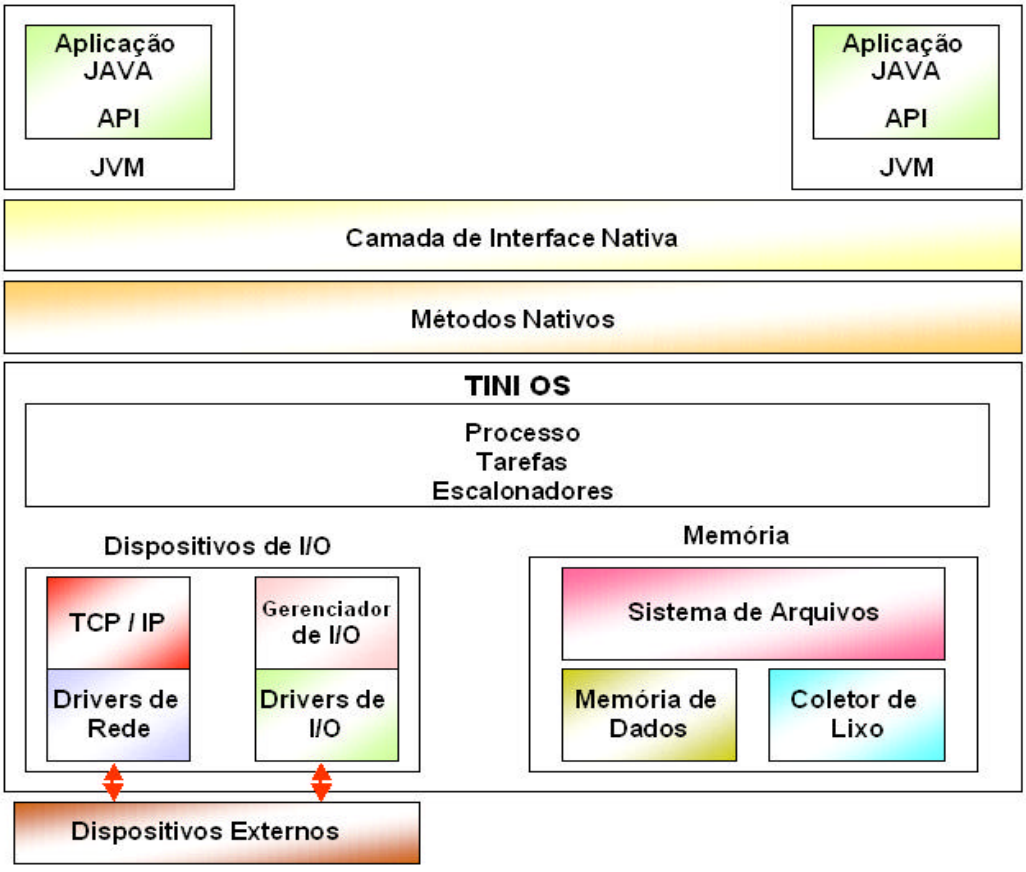


Figura 18 – Ambiente de execução da plataforma TINI (*Runtime*).

3.3.3.1 TINI - Sistema Operacional

O sistema operacional da plataforma é composto por:

1. **Escalonador de processos e *threads*** – o escalonador utilizado é do tipo *round-robin* e para cada processo são disponíveis oito milisegundos de execução. Finalizado o tempo, o mesmo é colocado no final da lista de processos. As *threads* obedecem ao mesmo sistema dos processos, mas possuem disponíveis somente dois milisegundos para execução.

2. **Gerenciador de memória** – o gerenciador é responsável pelo acesso de dados feito pelas aplicações Java ou pelo próprio sistema operacional. Além disso, executa automaticamente o coletor de lixo (*Garbage Collector* ou GC), responsável pela liberação da área de dados que deixa de ser utilizada pela aplicação. O GC normalmente pára a execução de uma aplicação enquanto ele percorre todo segmento de dados a procura de espaço não utilizado causando um atraso na execução da aplicação. Para evitar este problema, o GC da plataforma TINI é incremental e compactador, ou seja, é executado como um processo separado no sistema operacional e é disponibilizado somente em intervalos de tempo em torno de milisegundos para sua execução.

3. **Gerenciador I/O** – o gerenciador é responsável pelos dispositivos que não utilizam rede incluindo CAN e 1-Wire. As redes TCP/IP e Protocolo Ponto a Ponto (PPP) possuem controladores próprios.

3.3.3.2 TINI - JVM e APIs

A JVM TINI ocupa menos de 40 Kb, mas oferece suporte para os pacotes listados abaixo:

- ***java.lang*** – pacote com classes que são fundamentais para o desenvolvimento da linguagem de programação Java. As classes mais importantes são: *Object*, que é a principal na hierarquia, e a *Class* que é a instância que representa a classe em tempo real.

- ***java.net*** – classes para implementação de aplicações de rede. Usando a classe *socket* é possível comunicar qualquer servidor na Internet, ou implementar um servidor de Internet.

- ***java.io*** – sistemas de entrada e saída de fluxo de dados no sistema de arquivos.

- ***java.util*** – contém o *framework* de *Collections*, classes necessárias para o modelo de eventos, para manipulação de data e hora, internacionalização entre outras classes.

- ***javax.comm*** – interação com i*nterfaces* paralela e serial.

Outros pacotes que pertencem a *Dallas Semiconductor* estão agrupados no pacote *com.dalsemi* que permitem acesso à rede 1-Wire e a outros parâmetros da plataforma TINI. Alguns pacotes estão listados abaixo:

- ***com.dalsemi.comm*** – classes de baixo nível para controladores CAN e configuração das portas seriais.

- ***com.dalsemi.fs*** - extensão do padrão da classe *java.io.file.*

- ***com.dalsemi.io*** – algumas classes que auxiliam a conversão de códigos específicos entre as interfaces de I/O.

- ***com.dalsemi.protocol*** – classes usadas pelos diversos protocolos de rede.

- ***com.dalsemi.shell*** – implementação de comandos para os servidores de Protocolo de Transferência de Arquivos (FTP) e TELNET.

- ***com.dalsemi.system*** – classe de acesso ao hardware nativo do microprocessador DS80C390.

- ***com.dalsemi.tininet*** – classes de configuração e manipulação dos dispositivos de rede e suporte aos protocolos de rede.

3.3.4 Comunicação Serial

Muitos dispositivos usam a porta serial RS232 como meio de comunicação com outros dispositivos eletrônicos. Esta seção descreve aspectos de hardware e software para desenvolvimento de aplicações para plataforma TINI.

A plataforma TINI suporta cinco seriais e as portas são referenciadas como serial0 a serial4. As portas 0, 1 e 4 são integradas a UART do microcontrolador. As portas serial2 e serial3 devem ser conectadas a UARTs externas.

As portas seriais internas suportam as configurações listadas abaixo:

- ? 8 bits de dados, 1 stop bit e sem bit de paridade.
- ? 8 bits de dados, 1 stop bit e com bit de paridade (par ou ímpar).
- ? 7 bits de dados, 2 stop bits e sem paridade.
- ? 7 bits de dados, 1 stop bit com paridade (par ou ímpar).

A API de comunicação serial é definida pela *Sun Microsystems* como uma extensão da plataforma Java. A API é definida e implementada no pacote *javax.comm*.

É possível obter-se quais portas seriais estão disponíveis na plataforma TINI através do código descrito a seguir:

Programa PortLister.java

```
import java.util.Enumeration;
import javax.comm.CommPortIdentifier;

public class PortLister{

        public static void main(String[] args)){
                Enumeration ports = CommPortIdentifier.getPortIdentifiers();
                while(ports.hasMoreElements()){
                        System.out.println(((CommPortIdnetifier)
                         (ports.nextElement())).getName());
                }
        }

 }
```

A abertura da porta serial e o fechamento podem ser feitos utilizando-se os métodos *open* e *close*:

```
public synchronized CommPort open(String appname, int timeout) throws
PortInUseException;


public void close();
```

Para configurar os parâmetros de taxa de transmissão, número de *bits* de dados, número de *stop bits*, paridade e controle de fluxo podem ser configurados utilizando-se os métodos *setSerialPortParams* e *setFlowControlMode*.

```
public void setSerialPortParams(int baudrate, int dataBits,

int stopBits, int parity) throws UnsupportedCommOperationException;

public void setFlowControlMode(int flowcontrol) throws UnsupportedCommOperationException;
```

```
public int getFlowControlMode();
```

A leitura e escrita de dados da porta serial é realizada utilizando-se o pacote *java.io***.** Os métodos são *getInputStream* e *getOutputStream*.

Para controle de recebimento de dados existem os métodos *enableReceiveTimeout* e *enableReceiveThreshold* que permitem estabelecer um tempo e um *buffer* limites para leitura dos dados pela porta serial. Os métodos de controle podem ser desabilitados pelos métodos *disableReceiveTimeout* e *disableReceiveThreshold*.

```
public InputStream getInputStream() throws IOException;

public OutputStream getOutputStream() throws IOException;

public void enableReceiveTimeout(int rcvTimeout) throws UnsupportedCommOperationException;

public void disableReceiveTimeOut();

public void enableReceiveThreshold(int thresh) throws UnsupportedCommOperationException;

public void disableReceiveThreshold();
```

O tamanho do buffer para os dados pode ser definido e verificado utilizando-se os métodos *setInputBufferSize* e *getInputBufferSize*. O tamanho máximo suportado é 65535 bits.

```
public void setInputBufferSize(int size);

public int getInputBufferSize();
```

A API fornece um mecanismo de notificação assíncrono, tais como mudança de estado nas linhas de controle do *modem* (modulador/demodulador) e disponibilidade de dados. Na plataforma TINI os dados são tratados também por um processo que fica monitorando as mudanças nas portas seriais. O processo é criado quando o primeiro evento é adicionado ao objeto criado para a porta serial.

```
public void enableReceiveTimeout(int rcvTimeout) throws UnsupportedCommOperationException;

public void disableReceiveTimeout();
```

Para leitura e escrita da porta serial da plataforma TINI são apresentados os processos descritos abaixo:

3.3.5 Rede TCP / IP

Um dos principais objetivos da plataforma TINI é permitir que sistemas embutidos que não estão conectados a uma rede *Ethernet* possam ser conectados à rede podendo ser monitorados de qualquer ponto de uma rede.

A plataforma TINI implementa por completo todas as classes do pacote *java.net*, que permite desenvolver aplicações de rede. Usando a classe *socket* é possível comunicar-se com qualquer servidor na Internet ou implementar um servidor próprio.

A Figura 19 apresenta os protocolos de rede suportados pela plataforma TINI e seis camadas de aplicações implementadas pela API.

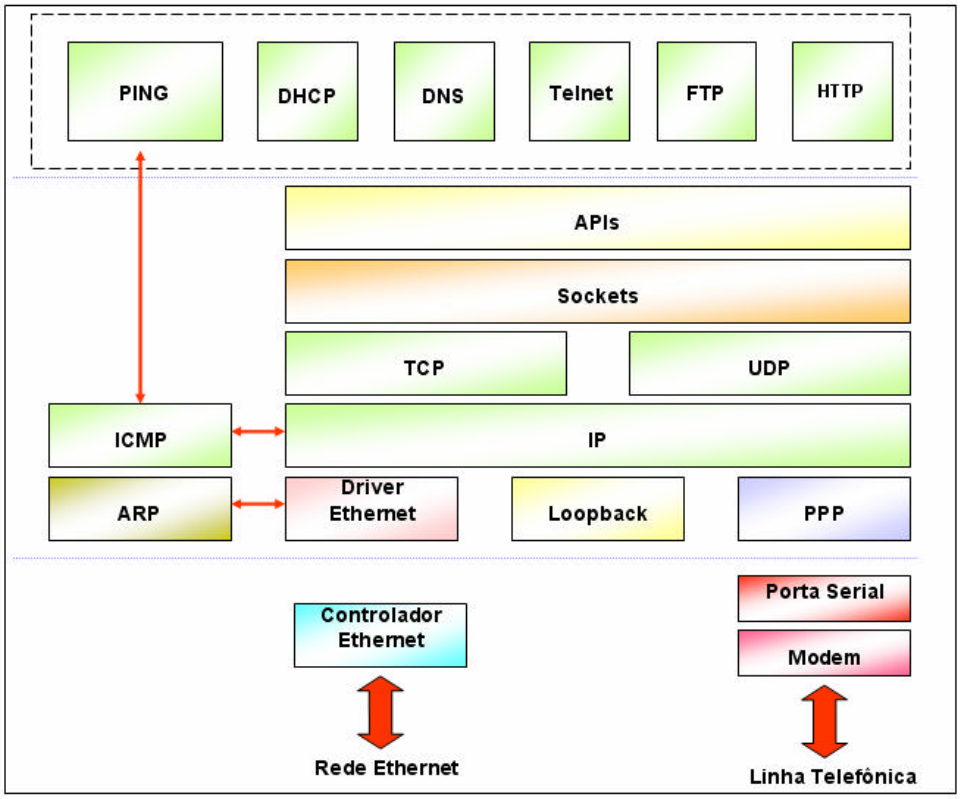


Figura 19 – Protocolos de rede implementados pela plataforma TINI.

Os protocolos de rede suportados pela plataforma TINI estão abaixo listados:

? **Transmissão de Hipertexto (HTTP)** – este protocolo é responsável pelo funcionamento da WWW. Torna possível a visualização dos documentos de hipermídias, como textos, gráficos, som, imagem e outros recursos disponíveis na WWW. Apesar da aparente complexidade das aplicações, o protocolo é

relativamente simples. As especificações do protocolo são úteis para todos que trabalham ativamente na criação de *sites* Web, manutenção de servidores Web, ou com desenvolvimento de programas clientes / servidores que venham a interagir com a WWW.

? **Serviço de Domínio de Nomes (DNS)** – este protocolo consiste em um sistema para nomear recursos de uma rede de forma que eles possam ser acessados pelo seu nome. O sistema consiste em servidores de nomes (*nameservers*) e *resolvedores* (*resolvers*).

O usuário consulta um nome no "resolvedor" e este se encarrega de conectar-se a servidores de nomes e realizar as consultas. Podem ser obtidos endereços de *hosts*, bem como informações sobre o servidor de domínio.

Um "resolvedor" deve iniciar com o endereço de pelo menos um servidor de nomes. Do ponto de vista do "resolvedor", o banco de dados de nomes no domínio está distribuído entre diferentes servidores. Quando um "resolvedor" processa uma consulta do usuário, ele faz uma consulta a um servidor de nomes conhecido. Este pode retornar a informação desejada ou o endereço de algum outro servidor de nomes.

? ``**Protocolo de Configuração Dinâmica (DHCP)** – criado para substituir o Protocolo *Bootstrap* (BOOTP) na tarefa de automatizar o fornecimento de endereços Protocolo de Internet (IP) em uma rede, o DHCP é um serviço que permite facilidades para redes que utilizam a computação móvel (rede sem fio, computadores portáteis) ou que possuem uma faixa de endereços IP limitada.

Dois fatores contribuíram para que esse novo protocolo de configuração fosse criado. O BOOTP resolveu parte do problema de subutilização do quadro no envio de um endereço IP. Com o DHCP, em uma única mensagem são enviadas para o equipamento todas as informações de inicialização necessárias. Outro fator importantíssimo é a locação rápida e dinâmica de um endereço IP para um equipamento conectado a rede.

? **Telnet** – o Telnet é um protocolo de aplicação que possibilita a interação entre dois computadores remotamente. Através deste aplicativo o computador cliente

poderá realizar uma conexão em um servidor que esteja conectado a rede ou a Internet.

- **FTP** – o protocolo FTP é um conjunto de regras que permitem que computadores realizem transferências de arquivos através da Internet ou na rede local. Devido a arquitetura do protocolo FTP, diferentes sistemas operacionais podem comunicar-se sem nenhum problema de compatibilidade. Para isso basta ter um aplicativo FTP cliente para acessar o servidor de FTP. O protocolo FTP oferece garantia de entrega de pacotes, pois utiliza o Protocolo de Controle de Transmissão (TCP) e também um controle de acesso.

- **Grupo de Pacotes de Internet (PING)** – o comando PING utiliza o Protocolo de Controle de Mensagem de Internet (ICMP) para realizar teste de conexão com a Internet.

O pacote principal que implementa os protocolos de rede é a *com.dalsemi.tininet*. Cada protocolo é implementado pelos pacotes listados a seguir:

- ***com.dalsemi.tininet.http;***
- ***com.dalsemi.tininet.icmp;***
- ***com.dalsemi.tininet.dhcp;***
- ***com.dalsemi.tininet.dns.***

3.3.6 Interface PPP

A plataforma TINI suporta também a comunicação PPP, que é atualmente um protocolo de propósito geral que suporta a transferência de dados sobre diferentes meios físicos, incluindo serial, paralela e *ethernet*. Atualmente, na plataforma TINI, o protocolo tem sido utilizado como um mecanismo de transporte de datagramas IP utilizando a serial RS232. As classes Java para manipulação do PPP estão definidas no pacote *com.dalsemi.tininet.ppp*. Depois de estabelecida a conexão, a transferência de dados é realizada utilizando-se as classes da *java.net*. [DS80C, 2005].

A vantagem de utilizar a interface PPP é que a comunicação entre os *hosts* que pode ser feita por uma linha telefônica utilizando-se *modems* permitindo assim que aplicações enviem dados mesmo em áreas que não possuem uma rede *ethernet*.

O objeto PPP é utilizado para controlar e monitorar o estado da conexão. Uma aplicação que cria um objeto PPP deve criar também um *listener*, para receber as notificações dos eventos PPP. O *listener* pode ser adicionado e removido utilizando os métodos *addEventListener* e *removeEventListener*, como ilustrado a seguir.

```
public void addEventListener(PPPEventListener listener) throws TooManyListenersException;

public void removeEventListener(PPPEventListener listener)
```

Os eventos são codificados em inteiros e assumem os estados relacionados abaixo e podem ser buscados utilizando-se o método *public int getEventType()*.

? ***STARTING*** – este evento é gerado quando o método open é chamado.

```
void open();
```

Esta transação envolve inicialização do canal de comunicação e envio de comandos para inicialização do modem.

Após a inicialização é chamado o método *up* onde é passada a referência da porta serial.

```
void up(SerialPort port) throws PPPException;
```

? ***AUTHENTICATION REQUESTED*** – este evento é gerado se o parâmetro *setAuthenticate* estiver habilitado. Se os parâmetros (identificação e senha) de conexão estiverem corretos o processo de conexão continua com a negociação. Se os parâmetros forem incorretos o evento STOPPED é gerado, rejeitando o cliente.

? ***UP*** – este evento ocorre quando a conexão é estabelecida com sucesso.

? ***STOPPED*** – este evento ocorre em resposta a algum erro gerado durante o processo de conexão ou durante a transferência de dados.

Os tipos de erros que podem ser gerados estão a seguir:

? NONE – sem ocorrência de erro

? ADDR – IP não identificado

? AUTH – autenticação inválida

? TIME – tempo de negociação da conexão esgotado

? REJECT – conexão rejeitada pelo computador remoto

? CLOSED – se o evento CLOSED for gerado devem ser removidas todas as interfaces e disponibilizadas para novas conexões.

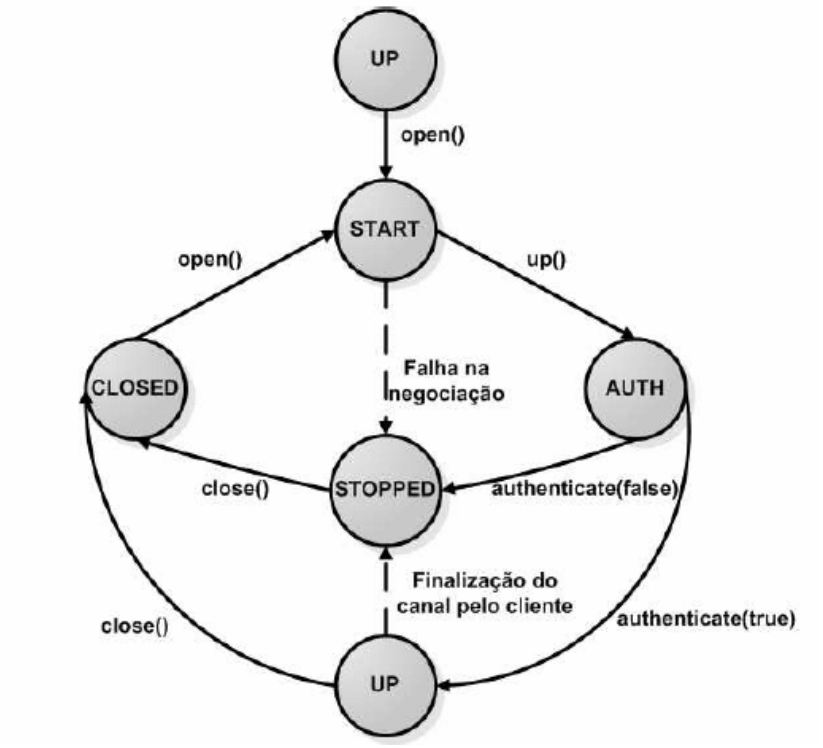


Figura 20 - Diagrama de estados da interface *PPPDeamon,* incluída na API da plataforma TINI.

3.4 Tecnologias de comunicação sem fio

3.4.1 Introdução

Tecnologias sem fio (w*ireless)* com diferentes propósitos, vantagens e desvantagens estão se tornando cada vez mais populares. De uma forma geral, as tecnologias *wireless* estão relacionadas com área de cobertura, que é a área onde a comunicação sem fio é possível, onde os usuários irão utilizar o dispositivo móvel. As tecnologias atuais proporcionam conectividade em ambientes específicos o que impõe limites com relação à banda, a área de

cobertura e aos custos. O resultado disto, é que os usuários estão limitados à tecnologia *wireless* utilizada, embora vários sistemas possam ser utilizados de forma complementar.

Classificado as tecnologias sem fio de acordo com sua área e cobertura, temos as coberturas dentro de ambientes (internas) como *Wireless* LAN (WLAN) onde as soluções estão incluídas nas especificações do IEEE 802.11 [WIFI, 2005], *High Performance Radio Local Area Network (HiperLan)*, [ETSI, 1994] e *Digital Enhanced Cordless Telecommunications* (DECT) *Systems* [DECT, 2005]. Nas tecnologias classificadas como externas ou *wide area* temos a predominância das tecnologias *Global System for Mobile Communications* (GSM) [REDL, S.M. et al, 1995] [REDL, S.M. et al, 1998], 802.16, *Code Division Multiple Access* (CDMA) e o *Time Division Multiple Access* (TDMA). Atualmente, estão sendo criados pontos de acesso públicos à Internet utilizando as tecnologias WLAN, que eram consideradas aplicações para aplicação interna. Ainda existe muito desenvolvimento e tecnologia por vir a fim de aumentar a largura de banda disponível para os usuários e, além disto, promover a integração entre as diversas tecnologias existentes.

Dentro das redes para dispositivos móveis os últimos desenvolvimentos estão centrados no aumento da largura de banda para transmissão de dados e a oferta de novos serviços (principalmente transmissão de dados) com valor agregado. Entre elas estão o *General Packet Radio Service* (GPRS) [BATES, R.J., 2001] [GHRIBI, B. et al, 2000] [ANDERSSON, C., 2001], *High Speed Circuit Switched Data (HSCSD)* [ETSI, 1999], as evoluções dessas redes como a *Enhanced Data Rates for GSM Evolution* (EDGE) [GHRIBI, B. et al, 2000] e a próxima geração também conhecida como 3G que está caminhando para a utilização do sistema *Universal Mobile Telecommunication System* (UMTS) [UMTS FORUM, 2005].

3.4.2 GSM

O GSM foi desenvolvido com tecnologia digital, diferentemente dos sistemas analógicos até então existentes. A operação comercial do GSM começou em meados de 1991 e em 1993, 36 redes já estavam operando a nova tecnologia em 22 países. Hoje, segundo o *Wireless Evolution Insider* [HOFFMAN, J., 2005], o GSM já se encontra em mais de 170 países com mais de 550 redes em operação e mais de 879 milhões de assinantes e uma média de crescimento de cerca de 14,5 milhões de assinantes por mês. Em comparação, a tecnologia CDMA possui cerca de 160 milhões de assinantes e um crescimento mensal de 2,4 milhões de novos assinantes. O GSM de fato foi o primeiro sistema que permitiu um sistema de comunicação móvel mundial como seu acrônimo o define. A capacidade de *roaming*

internacional foi habilitada, permitindo que o uso de dispositivos móveis, registrados em um país fossem utilizados em outro, desde que as operadoras destes países tivessem um acordo comercial que permitisse isto.

Para fornecer serviços de dados em redes GSM surgiu o *General Packet Radio Service (GPRS)*. Este serviço permite que as operadoras ofereçam serviços de dados apenas agregando novos equipamentos e atualizando alguns programas de controle das redes GSM já existentes. As soluções GPRS começaram a aparecer em meados de 1999/2000 utilizando a infra-estrutura das redes já em operação.

3.4.3 GPRS

GPRS é um novo serviço que provê acesso a dados através de tecnologia de comutação de pacotes em redes GSM e também em redes baseadas em *Time Division Multiple Access* (TDMA). A tecnologia GPRS foi padronizada pelo ETSI [ETSI, 2005] e é um marco para as tecnologias de dados para as redes GSM.

Um dos maiores benefícios, em comparação com *Circuit Switch Data* (CSD), é que a reserva de recursos da rede para tráfego de dados somente acontece quando o tráfego ocorre. GPRS implementa um degrau a mais antes da tecnologia de terceira geração (3G), possibilitando que as operadoras de telecomunicações transformem suas redes baseando-as em IP para tráfego de dados. O GPRS pode oferecer taxas de transferência de até 171,2 Kbps quando todas as fatias de tempo são utilizadas simultaneamente. Esta é uma taxa duas a três vezes maior que as atuais disponíveis nas conexões discadas convencionais, e até 10 vezes maior que as disponíveis nos serviços CSD. GPRS foi desenhado para coexistir com as redes públicas GSM *Public Land Mobile Networks (PLMN)* e pode ser implementado de maneira incremental sobre as redes GSM existentes em áreas específicas de maior demanda de serviços de dados. GPRS é a tecnologia que proporciona uma convergência entre redes GSM e TDMA (IS-136), pois o padrão está disponível para ambas as redes permitindo tráfego entre elas e uma transição mais transparente para as redes de terceira geração. Introduzindo a comutação de pacotes e o protocolo *Internet Protocol (IP)* nas redes móveis, GPRS possibilita o acesso aos tráfegos Internet, Intranet e redes corporativas.

O serviço GPRS é oferecido por diversas operadoras no Brasil, como por exemplo, TIM e CLARO. Segue abaixo o detalhamento deste serviço oferecido pela operadora TIM na cidade do Rio de Janeiro (pesquisa efetuada em fevereiro de 2006).

Operadora TIM (Rio de Janeiro)

Nesta operadora o serviço GPRS é oferecido através de uma oferta chamada *TIM CONNECT FAST*. Essa oferta está disponível automaticamente em todos os planos TIM GSM. A forma de faturamento deste serviço é por franquia de transmissão, conforme descrito na tabela abaixo:

Tabela 2 – Tarifas *TIM CONNECT FAST* – Serviço oferecido pela operadora TIM (RJ).

MB – Mínimo	MB - Máximo	R$ / MB
0	1	5,99
1	6	5,87
6	12	4,99
12	25	4,40
25	60	4,30
60		4.20

Observação: Os dados a respeito das tarifas foram coletados em Fevereiro de 2006.

Segue um exemplo de tarifação:

Calculo para utilização mensal de 4MB: Valor da fatura = 4 x R$ 5,87. Valor final da fatura: R$ 23,48.

Obs: Existem promoções específicas de acordo com os planos de serviços oferecidos pela operadora. Exemplo: Plano “TIM 500 Minutos”: Oferece 5MB de transmissão de dados mensais de forma gratuita.

Uma limitação do serviço oferecido pela operadora TIM é que na utilização do protocolo *Internet Protocol* (IP) para oferecer um endereço *IP* ao celular, a operadora restringe a faixa de IPs válidos somente na sua sub rede (192.168.*.*, por exemplo). Com isso não é possível iniciar uma conexão entre um computador na Internet e o celular via rede GPRS. A conexão deve sempre ser iniciada do celular para a rede externa (Internet), somente neste sentido.

3.4.4 Aplicações do GPRS

Através do GPRS, o usuário pode utilizar uma ligação contínua sem fios a redes de dados e acessar determinados serviços de informação. O GPRS é assim uma portadora de dados que possibilita o acesso por comunicação sem fio a redes de dados como a Internet,

Alguns serviços que podem ser utilizados através da tecnologia GPRS são:

- WAP: possibilita o acesso a páginas em formato WML;
- SMS: permite o envio de mensagens entre dispositivos móveis;
- MMS: permite que os usuários possam incorporar áudio, imagens e outro precioso conteúdo nas mensagens de texto tradicionais;
- Aplicações embarcadas: disponibiliza um canal de conexão a redes como a Internet para aplicações embarcadas desenvolvidas, por exemplo, com tecnologia Java;

3.5 Conclusões da revisão de tecnologias

Existem alguns sistemas embutidos de baixo custo, capazes de suportar o desenvolvimento utilizando linguagem Java, facilitando e agilizando a implementação de aplicações. No entanto, a oferta de plataformas de *hardware* com suporte a Java ainda é bastante limitada.

A arquitetura *Java 2 Micro Edition* (J2ME) oferece suporte ao desenvolvimento de aplicações Java em dispositivos de *hardware*. Contudo existem dispositivos especialistas que rodam programas Java diretamente (*bytecode* nativo), sem a necessidade de uma máquina virtual Java.

GPRS é um serviço disponibilizado em redes GSM para transmissão de dados, que oferece muitas possibilidades de aplicações em sistemas que demandam comunicação remota, e pode ser empregado no desenvolvimento de sistemas móveis.

4 ANÁLISE, PROJETO E IMPLEMENTAÇÃO DE UM SISTEMA PARA GERENCIAMENTO DE TÁXIS UTILIZANDO SISTEMAS EMBUTIDOS

4.1 Introdução

A concepção deste sistema originou-se da observação da forma de operação de várias companhias de táxi atuais. Existem algumas formas de operação destas companhias para gerenciar o atendimento de chamadas e também gerenciar as prioridades dos atendimentos. Em todos os modelos de operação levantados existem determinados requisitos funcionais comuns, os quais devem ser atendidos obrigatoriamente na nova solução, que são:

- O sistema deve possibilitar a troca de informações a respeito das chamadas entre o taxista e a central de atendimento;
- O sistema deve possibilitar o posicionamento do taxista em determinada região (regiões físicas como bairros, ruas, etc, ou posições lógicas como área 23, área 44, etc);
- O sistema deve gerenciar as filas de atendimento, verificando as prioridades para o atendimento de chamadas, ou seja, qual é o próximo taxista a atender uma chamada em determinada região;
- O sistema deve possibilitar o mapeamento geográfico dos logradouros (ruas, bairros, praças, CEPs, etc) em áreas de atendimento lógicas (ou regiões), conforme descrito acima.
- O sistema deve ser capaz de resolver conflitos entre taxistas posicionados nas áreas de atendimento, ou seja, qual taxista foi o primeiro a se posicionar em uma certa área;

A proposta desta dissertação é conceber e implementar uma solução para substituir a comunicação via rádio dos sistemas de gerenciamento de veículos, utilizando um sistema

embutido instalado no veículo e um sistema servidor instalado numa base (ou central) de operações, buscando atender os requisitos necessários para a operação diária de uma companhia de táxi. Nesta implementação não existe a necessidade de utilizar um dispositivo GPS, pois é o motorista que informa sua posição lógica à central e somente será automatizado um determinado modelo de operação utilizado por algumas companhias de táxi, o qual é baseado na troca de informações via rádio entre a central e o taxista.

A proposta deste trabalho é efetuar o gerenciamento da posição de veículos de acordo com a sua posição física, sem a necessidade de integração com um dispositivo GPS. A diferença entre o projeto [TINIGPS, 2005] e esta proposta está na estratégia de implementação. No modelo proposto por este trabalho, o dispositivo móvel exige uma interação externa para a confirmação do posicionamento físico, que não é necessariamente um posicionamento global. Um sistema embutido [EMBD, 2005] irá controlar o posicionamento da unidade móvel através de entrada de dados externa (feita por um usuário do sistema), que pode ser categorizada em regiões ou áreas lógicas.

O Principal objetivo é proporcionar um serviço equivalente e mais eficiente que o atual, automatizando o modelo de operação existente, que é baseado em troca de mensagens via rádio.

4.2 Análise e Projeto

Para analisar o problema foram levantados alguns modelos de operação em várias cooperativas de táxis (nas cidades de Brasília, São Paulo e Rio de Janeiro). Todos os modelos são baseados na comunicação via rádio entre o taxista e a central (operador). Nesta análise foi possível identificar dois tipos de modelos de operação que são: o Modelo “A” - Ponto de Apoio Fixo e o Modelo “B” - Ponto de Apoio Flutuante. Segue abaixo o detalhamento destes dois modelos.

4.2.1 Modelo de operação “A” - Ponto de Apoio Fixo

Neste modelo, toda a comunicação entre o taxista e a central é feita via rádio, pois todos os veículos possuem um rádio instalado. A cooperativa transmite as informações através de antenas de rádio espalhadas pela cidade, que podem ser proprietárias ou alugadas. Normalmente, os canais de comunicação são divididos em um canal para receber chamadas e outro canal para falar diretamente com a central.

Uma outra característica deste modelo é o conceito de ponto de apoio fixo (PA Fixo). Neste modelo os pontos de apoio são associados a determinadas localidades físicas. Exemplo: PA Zona Sul, PA Rio Comprido, etc. Esses pontos de apoio são filas **reais** de veículos, onde o primeiro taxista a entrar na fila é o primeiro a ser chamado para atendimento naquela região. O taxista pode entrar em qualquer ponto de apoio (entrar na fila), desde que esteja fisicamente no local.

Logística de atendimento do modelo "A"

Segue abaixo um cenário utilizando o modelo "Ponto de Apoio Fixo":

Passo 1) O atendente (*call center*) recebe um pedido de um cliente via telefone e o repassa para o operador de rádio;

Passo 2) O operador de rádio seleciona o ponto de apoio (fixo) mais próximo da chamada e avisa o primeiro taxista da fila (primeiro da fila naquele ponto de apoio fixo);

Passo 3) O taxista atende a chamada, dando lugar para o próximo veículo dessa fila (este passa a ser o primeiro da fila);

Passo 4) O taxista termina o atendimento e retorna para um ponto de apoio fixo a sua escolha (chegando fisicamente no local, o taxista entra na fila daquele ponto de apoio);

Vantagens e desvantagens do modelo "A"

A grande vantagem deste modelo é o fato dele evitar os conflitos entre os taxistas (o primeiro taxista da fila física sempre atende a chamada). O modelo não é tendencioso e não privilegia um determinado veículo ou motorista, ou seja, a decisão de qual é o próximo táxi a atender uma determinada chamada não depende dos operadores de rádio.

A principal desvantagem deste modelo é o retorno do taxista para um determinado ponto de apoio fixo. Neste modelo o taxista é obrigado a estar fisicamente no local para poder entrar em uma nova fila de atendimento, não aproveitando o tempo de deslocamento até o ponto de apoio desejado para receber chamadas via rádio. Outra desvantagem é o tempo perdido na própria fila do ponto de apoio, não sendo possível receber chamadas ou efetuar um atendimento enquanto estiver aguardando sua posição na fila (se o taxista se ausentar do ponto de apoio fixo ele perde sua posição na fila).

4.2.2 Modelo de operação “B” - Ponto de Apoio Flutuante

Neste modelo de operação, toda a comunicação entre o taxista e a central também é feita via rádio. Os canais de comunicação são divididos por freqüência, sendo um canal para recepção de chamadas e outro canal para falar diretamente com a central.

A grande diferença entre este modelo e o modelo “A” é que os pontos de apoio (PA) não são fixos, ou seja, não existe uma fila física de veículos em um determinado local. Neste modelo os pontos de apoio são associados a determinadas localidades lógicas. Exemplo: PA Barrashopping, PA Zona Sul e Centro, PA Rio Comprido, etc. Esses pontos de apoio são filas **virtuais** de veículos, onde o primeiro taxista a entrar na fila virtual é o primeiro a ser chamado para atendimento naquela região ou área lógica. O taxista pode entrar em qualquer ponto de apoio a qualquer momento, bastando ele se comunicar com a central e se reposicionar em outro PA.

Neste modelo, os taxistas podem se posicionar em determinada área se estiverem a *N* minutos da região demarcada. Caso ele se posicione em uma determinada região e não consiga atender a chamada em *N* minutos ele pode ser penalizado, dependendo das regras de negócio da companhia de táxi em que ele opera.

Existem vários tipos de penalização para o taxista, e cada companhia implementa a sua política de penalização de maneira diferente. Seguem abaixo alguns tipos de penalização aplicados em algumas companhias:

- Penalização para atrasos de 20 (vinte) minutos no atendimento: 30 (trinta) minutos sem comunicação via rádio (ou seja, o taxista não recebe nenhuma chamada através do operador);
- Reincidência no atraso: 60 (sessenta) minutos sem comunicação via rádio;
- Terceiro atraso: Um dia sem comunicação via rádio;

Logística de atendimento do modelo “B”

Segue abaixo um cenário utilizando o modelo “Ponto de Apoio Flutuante”:

Passo 1) O taxista se posiciona em uma determinada área lógica através do rádio. Ele informa ao operador de rádio qual a região desejada para posicionamento. Exemplo: “Estou me posicionando na área 42, centro”;

Passo 2) O operador de rádio retorna para o taxista a confirmação do pedido e a posição do taxista nesta fila virtual. Exemplo: “Posicionamento efetuado, você é o terceiro táxi desta fila”;

Passo 3) O atendente (call center) recebe um pedido de um cliente via telefone e o repassa para o operador de rádio;

Passo 4) O operador de rádio seleciona o ponto de apoio virtual que está associado ao endereço da chamada, e avisa o próximo taxista desse ponto de apoio virtual, ou seja, o primeiro táxi da fila para o determinado ponto de apoio;

Passo 5) O taxista atende a chamada, dando lugar para o próximo veículo dessa fila. Passo 6) O taxista termina o atendimento e se posiciona novamente em algum ponto de apoio virtual, sem a necessidade de chegar em uma fila física de veículos, pois esta não existe;

Vantagens e desvantagens do modelo “B”

Segundo a maioria dos taxistas entrevistados, o modelo de ponto de apoio flutuante é a alternativa mais interessante no sentido de aproveitamento de tempo, pois ele não perde tempo voltando para um ponto de apóio fixo. Ele pode ser cadastrar a qualquer momento, em qualquer ponto de apoio virtual.

Contudo, este tipo de solução gera um grande congestionamento na comunicação via rádio, pois todos os taxistas têm que se cadastrar na região lógica para receberem as chamadas daquela região. Eles também têm a necessidade de consultar a quantidade de veículos em determinada área para saber se já há muitos táxis na área, aumentando ainda mais a troca de informações entre a central e o táxi, gerando mais congestionamento na freqüência de rádio e dificultando o trabalho do operador. Outra desvantagem a ser considerada é a possibilidade do operador ser tendencioso e simplesmente ignorar o pedido de posicionamento de um determinado taxista, ou até mesmo de privilegiar outro, já que a gerencia da fila virtual é feita por um operador.

4.2.3 Análise dos modelos “A” e “B”

Alguns problemas são comuns às duas soluções, como por exemplo a qualidade da transmissão via rádio. Em determinadas regiões a qualidade da transmissão via rádio é muito ruim, seja devido ao nível de ruído ou às interferências externas, tornando impraticável a troca de informações entre a central e o taxista. A comunicação via telefone GPRS também está sujeita a zonas de silêncio (sem cobertura), porém evita erros de interpretação de mensagens. Outro problema inerente à comunicação por rádio é o fato de serem necessários vários operadores de rádio trabalhando 24 horas por dia. O modelo “A” (Ponto de Apoio Fixo) apresenta uma desvantagem na logística de atendimento: a necessidade da presença física do taxista na fila. Já o modelo “B” (Ponto de Apoio Flutuante), apresenta algumas desvantagens tais como o congestionamento do rádio e a possibilidade do operador privilegiar ou prejudicar determinado taxista.

Após a análise das vantagens e desvantagens dos dois modelos, foi escolhido o modelo “B” (Ponto de Apoio Flutuante) como o mais apropriado para a implementação do sistema em questão.

4.2.4 Regras de negócio

O objetivo desta seção é prover uma visão completa dos requisitos levantados para implementação do sistema. Para levantamento das regras de negócio foram utilizadas as informações coletadas em campo, de acordo com o modelo de operação B descrito na seção 4.2.2. Estas regras serão descritas através de funcionalidades do sistema, as quais foram chamadas de casos de uso e suas atividades estão detalhadas a seguir.

4.2.4.1 Macro Funcionalidades

Foram levantadas as seguintes macro-funcionalidades para o sistema proposto:

- **FUNC-01 – Autenticação Usuário**
- **FUNC-02 – Seleção Ponto Apoio**
- **FUNC-03 – Consulta Ponto Apoio**
- **FUNC-04 – Recebimento Chamada Atendimento**
- **FUNC-05 – Finalização Chamada Atendimento**

4.2.4.2 Requisitos Funcionais

Segue o levantamento dos requisitos descritos através de casos de uso do sistema,

Caso de Uso FUNC-01 "Autenticação Usuário"

Descrição do Caso de Uso

O objetivo deste caso de uso é descrever o processo de autenticação de usuários no sistema

Atores do Caso de Uso

Taxistas

Pré-Condições

Dispositivo móvel instalado no táxi; Sistema servidor *on-line*; Sistema de Banco de Dados *on-line*;

Pós-Condições

Usuário autenticado no sistema.

Fluxo Básico

Passo 1) O caso de uso tem início quando o taxista acessa o sistema a partir do dispositivo móvel instalado no táxi.

Passo 2) O taxista informa o código do usuário e a senha.

Passo 3) O sistema recupera os dados do usuário, verifica as informações na base de dados e autentica o usuário. Neste processo são armazenadas as seguintes informações do usuário na sessão do cliente: código do usuário autenticado; número da unidade móvel do usuário (táxi), nome do usuário (nome do taxista) e data e hora do login.

Fluxo Alternativo (Código de Usuário Inválido)

Passo 1) O usuário (taxista) informa um código de usuário incorreto

Passo 2) O sistema exibe mensagem "Usuário Inválido"

Passo 3) O sistema retorna para o passo 2 do fluxo básico

Fluxo Alternativo (Senha Inválida)

Passo 1) O usuário (taxista) informa a senha incorreta

Passo 2) O sistema exibe mensagem "Usuário Inválido"

Passo 3) O sistema retorna para o passo 2 do fluxo básico

Caso de Uso FUNC-02 - “Selecionar Ponto de Apoio”

Descrição do Caso de Uso

O objetivo deste caso de uso é descrever o processo de seleção de ponto de apoio a um taxista. Através da seleção do ponto de apoio o taxista pode se posicionar na fila virtual daquele ponto. Com isso ele se cadastra na fila e aguarda o recebimento de uma chamada de atendimento.

Atores do Caso de Uso

Taxistas

Pré-Condições

Dispositivo móvel instalado no táxi; Sistema servidor *on-line*; Sistema de Banco de Dados *on-line*; O Usuário deve estar autenticado no sistema.

Pós-Condições

Ponto de apoio selecionado (posicionamento na fila de atendimento deste ponto)

Fluxo Básico

Passo 1) O caso de uso tem início quando o taxista acessa a o sistema a partir do dispositivo móvel instalado no táxi.

Passo 2) O usuário se autentica no sistema (FUNC-01)

Passo 3) O usuário informa o código do ponto de apoio para seleção

Passo 4) O sistema recebe o código do ponto de apoio e cadastra o usuário na fila de atendimento para este ponto de apoio.

Passo 5) O sistema retorna ao taxista a sua posição da fila para o ponto de apoio selecionado

Fluxo Alternativo (Ponto de Apoio Inválido)

Passo 1) O usuário (taxista) informa um código de ponto de apoio incorreto

Passo 2) O sistema exibe mensagem “Ponto de Apoio Inválido”

Passo 3) O sistema retorna para o passo 3 do fluxo básico

Caso de Uso FUNC-03 - “Consultar Ponto de Apoio”

Descrição do Caso de Uso

O objetivo desse caso de uso é descrever o processo de consulta do *status* da fila de um determinado ponto de apoio. Através desta consulta o taxista pode visualizar a quantidade de táxis em uma determinada fila virtual, referente ao ponto de apoio

informado. Este processo ajuda o taxista a decidir em qual ponto de apoio ele tem mais chances de receber uma chamada, ou seja, possivelmente a fila com menor quantidade de táxis.

Atores do Caso de Uso

Taxistas

Pré-Condições

Dispositivo móvel instalado no táxi; Sistema servidor *on-line*; Sistema de Banco de Dados *on-line*; O Usuário deve estar autenticado no sistema; O usuário deve ter efetuado a seleção de um ponto de apoio;

Pós-Condições

Não há pós-condições.

Fluxo Básico

Passo 1) O caso de uso tem início quando o taxista acessa a o sistema a partir do dispositivo móvel instalado no táxi.

Passo 2) O usuário se autentica no sistema (FUNC-01)

Passo 3) O usuário se posiciona em um determinado ponto de apoio (FUNC-02)

Passo 4) O usuário informa o código de um ponto de apoio para (quantidade de táxis na fila de atendimento)

Passo 5) O sistema recebe o código de um ponto de apoio e verifica na base de dados a quantidade de táxis na fila de atendimento para este ponto

Passo 6) O sistema retorna ao taxista o numero máximo de táxis na fila do ponto de apoio informado

Fluxo Alternativo (Ponto de Apoio Inválido)

Passo 1) O usuário (taxista) informa um código de ponto de apoio incorreto

Passo 2) O sistema exibe mensagem “Ponto de Apoio Inválido”

Passo 3) O sistema retorna para o passo 4 do fluxo básico

Caso de Uso FUNC-04 - “Recebimento Chamada Atendimento”

Descrição do Caso de Uso

O objetivo deste caso de uso é descrever o processo de como o taxista recebe um aviso para atender uma determinada chamada.

Atores do Caso de Uso

Taxistas

Pré-Condições

Dispositivo móvel instalado no táxi; Sistema servidor *on-line*; Sistema de Banco de Dados *on-line*; O Usuário deve estar autenticado no sistema; O usuário deve ter efetuado a seleção de um ponto de apoio;

Pós-Condições

Usuário retirado da fila de atendimento (necessita se posicionar novamente em algum ponto de apoio);

Fluxo Básico

Passo 1) O caso de uso tem início quando o taxista acessa o sistema a partir do dispositivo móvel instalado no táxi.

Passo 2) O usuário se autentica no sistema (FUNC-01)

Passo 3) O usuário se posiciona em um determinado ponto de apoio (FUNC-02)

Passo 4) O usuário aguarda na fila de atendimento até que ele seja o primeiro da dessa fila.

Passo 5) O sistema notifica o usuário a respeito de uma chamada para aquela área de atendimento.

Passo 6) O usuário confirma o recebimento e inicia o atendimento da chamada

Passo 7) O sistema receba a confirmação e retira o usuário daquela fila de atendimento, passando a posição para o próximo táxi da fila.

Passo 7) O sistema marca essa chamada como uma chamada em andamento

Fluxo Alternativo (Usuário Rejeita a Chamada)

Passo 1) O usuário aguarda na fila de atendimento até que ele seja o primeiro da dessa fila.

Passo 2) O sistema notifica o usuário a respeito de uma chamada para aquela área de atendimento.

Passo 3) O usuário rejeita por algum motivo essa chamada.

Passo 4) O sistema retira o usuário da fila (perde sua posição) e passa a chamada para o próximo táxi da mesma fila.

Passo 3) O sistema retorna para o passo 3 do fluxo básico

Caso de Uso FUNC-05 - “Finalização Chamada Atendimento”

Descrição do Caso de Uso

O objetivo deste caso de uso é descrever o processo de como o taxista finaliza uma chamada em andamento.

Atores do Caso de Uso

Taxistas; Operador da Central de atendimento

Pré-Condições

Dispositivo móvel instalado no táxi; Sistema servidor *on-line*; Sistema de Banco de Dados *on-line*; O Usuário deve estar autenticado no sistema; O usuário deve ter efetuado a seleção de um ponto de apoio; Deve haver uma chamada em andamento;

Pós-Condições

Chamada finalizada

Fluxo Básico

Passo 1) O caso de uso tem início quando o taxista acessa o sistema a partir do dispositivo móvel instalado no táxi.

Passo 2) O usuário se autentica no sistema (FUNC-01)

Passo 3) O usuário se posiciona em um determinado ponto de apoio (FUNC-02)

Passo 4) O usuário aguarda na fila de atendimento até que ele seja o primeiro da dessa fila.

Passo 5) O sistema notifica o usuário a respeito de uma chamada para aquela área de atendimento. O usuário aceita a chamada (FUNC-03)

Passo 6) O usuário informa ao sistema que a chamada foi finalizada

Passo 7) O sistema marca a chamada como finalizada

Fluxo Alternativo (Usuário Cancela a Chamada em Andamento)

Passo 1) O usuário aguarda na fila de atendimento até que ele seja o primeiro da dessa fila.

Passo 2) O sistema notifica o usuário a respeito de uma chamada para aquela área de atendimento. O usuário aceita a chamada (FUNC-03)

Passo 3) O usuário cancela a chamada em andamento.

Passo 4) O sistema marca a chamada como cancelada.

Passo 5) A chamada permanece cancelada até que o operador da central reative a chamada

4.2.5 Modelagem de Dados

De acordo com as regras de negócio definidas no levantamento de requisitos, foram modeladas as tabelas do sistema para a base de dados da solução. O servidor de banco de dados escolhido para o desenvolvimento foi o IBM DB2, versão 7.0. Para a modelagem foi utilizada a ferramenta PLATINUM ERwin ERX versão 3.5.2.

Segue abaixo o diagrama de Entidade / Relacionamento das principais tabelas do sistema.

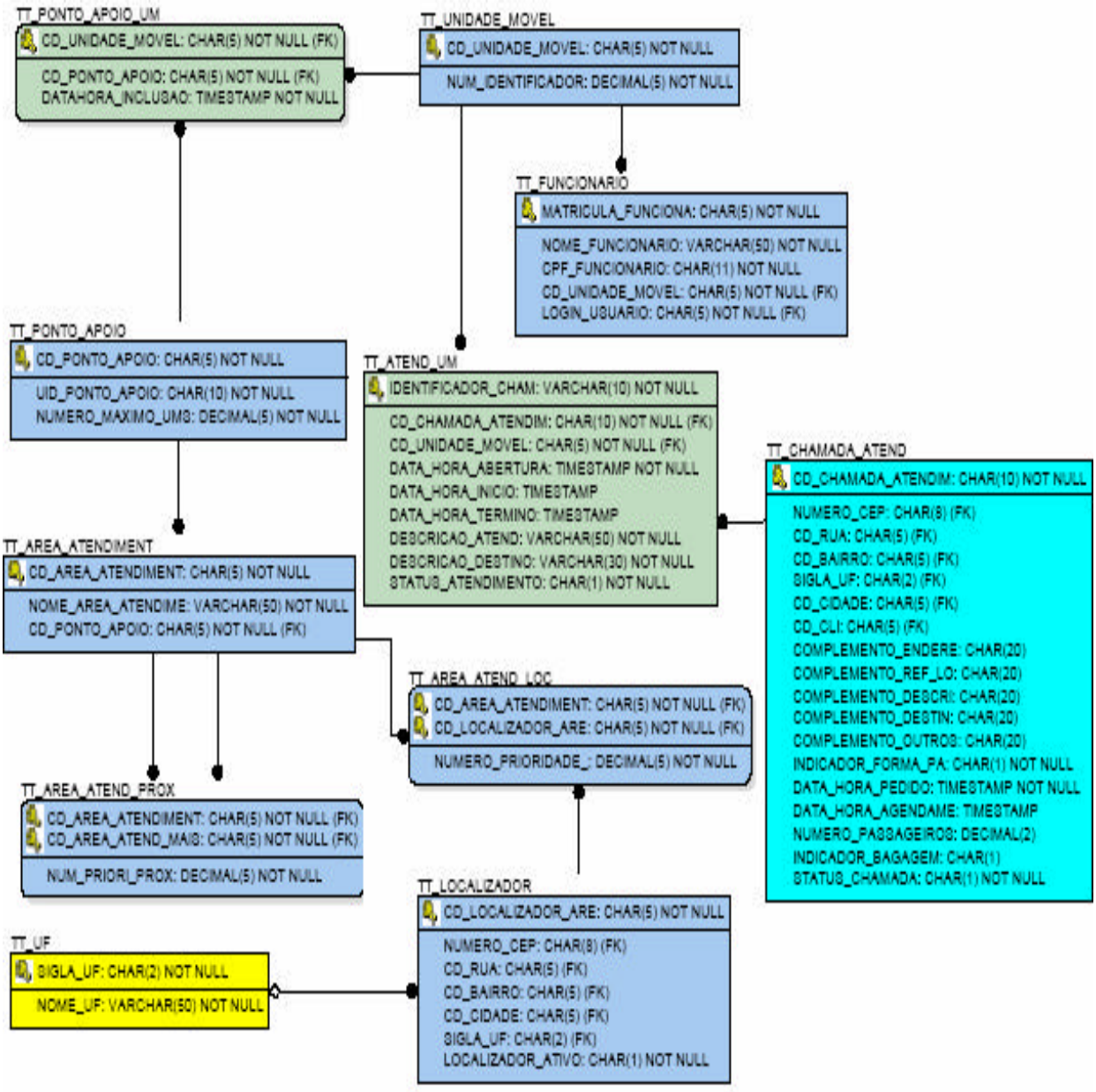


Figura 21 – Modelo de dados – Visão das principais tabelas (parte 1).

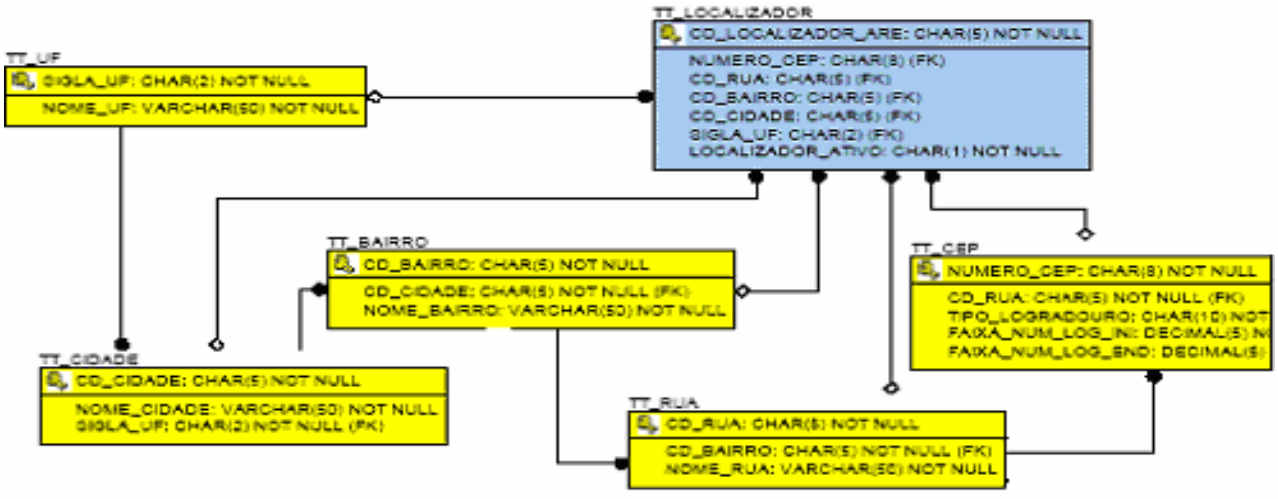


Figura 22 – Modelo de dados – Visão das principais tabelas (parte 2).

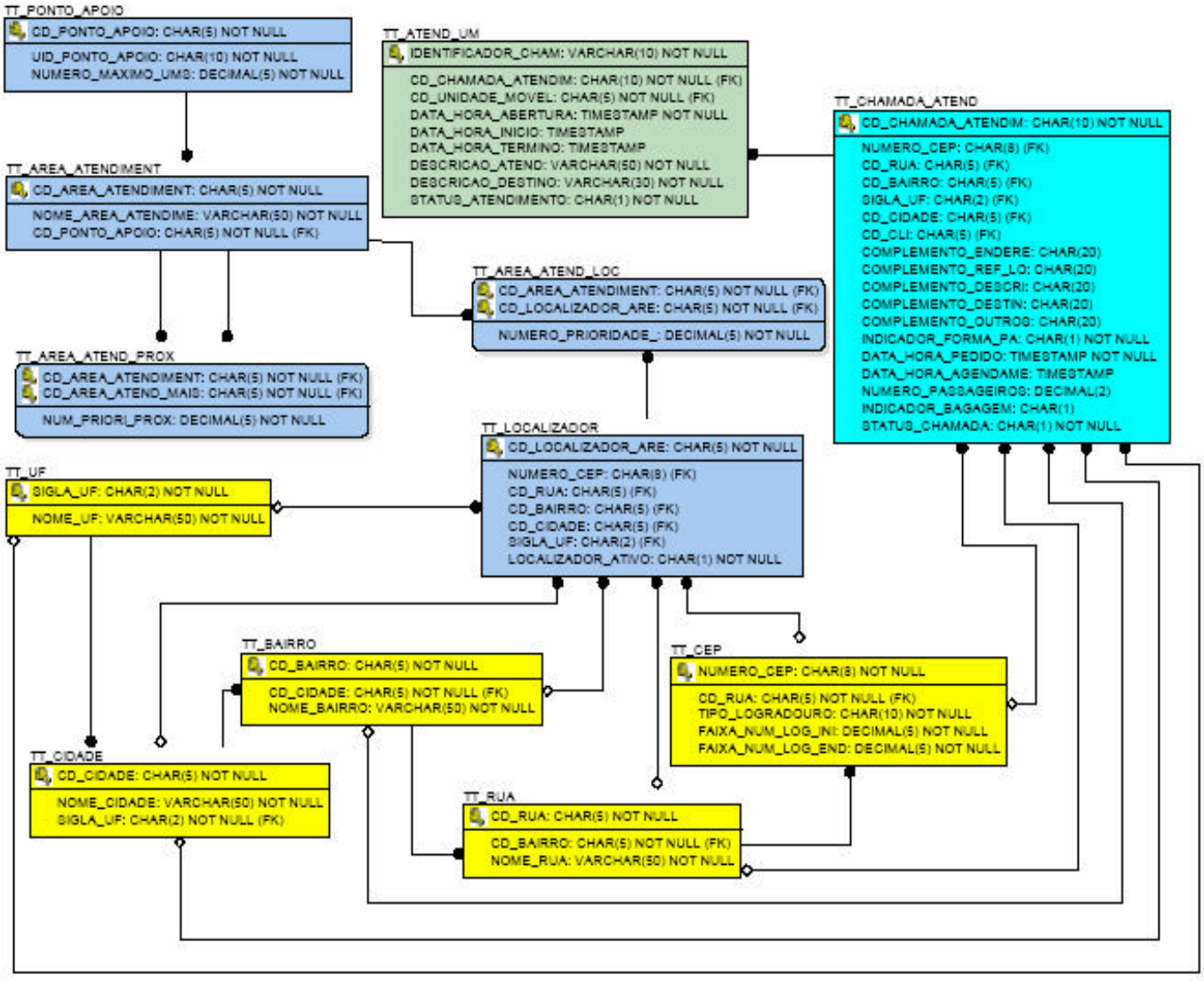


Figura 23 – Modelo de dados – Visão das principais tabelas (parte 3).

Para implementação, foi criado um identificador chamado TiniTaxi para facilitar a nomenclatura utilizada no desenvolvimento do sistema. Nesta implementação estão descritos os projetos presentes no lado servidor (central de atendimento) e os projetos presentes no lado cliente (táxi).

Todos os projetos implementados (lado cliente e lado servidor) foram desenvolvidos utilizando linguagem Java. Cada camada do sistema em especial utiliza determinada especificação Java (lado cliente - J2SE; lado servidor - J2EE [CATTEL, 2001]). Todos os programas Java implementados seguem o padrão proposto pela convenção de código conhecida por *Sun Code Conventions.* As classes implementadas ficam distribuídas em pacotes dentro do sistema. Como padrão, todos os pacotes criados seguem a seguinte definição:

domínio da instituição/empresa + subdivisões internas/áreas/departamentos + nome do projeto + componente.

Segue um exemplo de nome de pacote utilizado na implementação:

br.ufrj.nce.se.tinitaxi.server.io, sendo que **br.ufrj.nce** é o domínio da instituição, **se** é uma subdivisão (Sistemas Embutidos), **tinitaxi** é o nome do projeto e **server.io** é o nome do componente. Cada pacote possui um conjunto de classes, representando a implementação real do componente.

4.2.6 Implementação (Aplicação Servidora)

De acordo com as regras de negócio levantadas e a modelagem de dados realizada, foi desenvolvida uma aplicação para gerenciar o lado servidor de todo o sistema. Entende-se como lado servidor a parte do sistema que acessa a base de dados e centraliza as regras de negócio. Também está incluída nesta aplicação servidora a implementação de um programa do tipo *socket server* para comunicação com os dispositivos instalados nas unidades móveis (táxis).

A aplicação servidora foi construída utilizando a plataforma *Java 2 Enterprise Edition* (J2EE [CATTEL, 2001]). Essa aplicação é composta por sete projetos, de acordo com a divisão lógica de camadas do sistema. Segue abaixo a lista de projetos implementados do lado servidor:

- ***tinitaxi-ear*** **– Projeto principal para empacotamento do sistema servidor**
- ***tinitaxi-ejb*** **– Projeto com a lógica de negócios do sistema servidor**
- ***tinitaxi-web*** **– Projeto para apresentação de informações na WEB**
- ***tinitaxi-server*** **– Projeto servidor (*socket server*) para gerenciar as conexões com os dispositivos móveis**
- ***tinitaxi-batch*** **– Projeto para processamento *batch* das filas de atendimento**
- ***tinitaxi-framework*** **– Projeto com código comum (API) para todo o sistema servidor**
- ***tinitaxi-container*** **– Projeto com as configurações do servidor de aplicação**

Cada subprojeto possui um conjunto de pacotes e classes específicas para implementação das funcionalidades do sistema. Segue abaixo o detalhamento de todos os subprojetos do lado servidor.

4.2.6.1 Projeto tinitaxi-ear

O projeto *tinitaxi-ear* representa a aplicação servidora como um todo. O sufixo *.ear* significa *(Enterprise Application Archive).* Na verdade, todos os outros projetos do lado servidor estão empacotados dentro do projeto *tinitaxi-ear*. Essa estrutura é padrão e está definida na especificação *Java 2 Enterprise Edition* (J2EE [CATTEL, 2001]), sendo necessária para instalação do Servidor de Aplicações (*Application Server)* – Empacotamento da aplicação.

O servidor de aplicações é um produto à parte do sistema, e também é descrito na especificação J2EE [CATTEL, 2001]. Ele é responsável por executar a aplicação e disponibilizar seus componentes para acesso, de acordo com o ambiente de execução específico, por exemplo: Ambiente WEB, ambiente cliente-servidor, etc.

Para realizar a instalação da aplicação no lado servidor, é necessária a geração de um arquivo com extensão *.ear*. Esse arquivo é gerado através do empacotamento do projeto *tinitax-ear*, de acordo com a especificação J2EE [CATTEL, 2001].

O projeto *tinitaxi-ear* está dividido em três divisões lógicas: *Modules*, *Project Utility JARs* e *Utility JARs*.

Segue abaixo a visão hierárquica do projeto *tinitaxi-ear*:

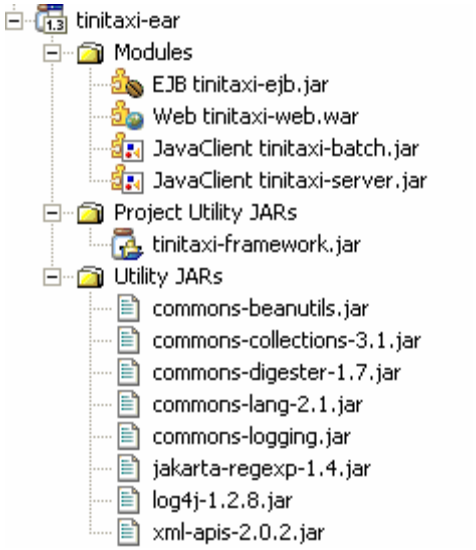


Figura 24 – Estrutura hierárquica do projeto *tinitaxi-ear* (Divisão lógica de camadas).

Modules - Contém todos os módulos do sistema. Estes módulos podem ser categorizados em três tipos:

- Módulos EJB (extensão *.jar*) – Módulos para implementação de componentes de negócio. No sistema existe um módulo EJB, gerado a partir do empacotamento do projeto *tinitaxi-ejb* [FLOYD, 2001] [MANSON-HALFEL, 2001, 200, 1999] [ROMAN, 1999].

- Módulos WEB (extensão *.war*) – Módulos para implementação de aplicações WEB. No sistema existe um módulo WEB, gerado a partir do empacotamento do projeto *tinitaxi-web*.

- Módulos JavaClient (extensão *.jar*) – Módulos para implementação de aplicações clientes, que normalmente acessam os componentes de negócio implementados nos módulos EJB [FLOYD, 2001] [MANSON-HALFEL, 2001, 200, 1999] [ROMAN, 1999]. No sistema existem dois módulos JavaClient, gerados a partir do empacotamento dos projetos *tinitaxi-batch* e *tinitaxi-server*.

Project Utility JARs - Contém todos os arquivos JARs comuns a todo o projeto EAR. Nesta implementação existe apenas um JAR comum a todo EAR, que é gerado a partir do empacotamento do projeto *tinitaxi-framework*.

Utility JARs – Contém o conjunto de bibliotecas de API comuns a todo o projeto EAR. Foram utilizadas as seguintes bibliotecas de API:

- *commons-beanutils.jar* – API para controle e manipulação de objetos *JavaBeans*.
- *commons-collections-3.1.jar* – API utilitária para controle e manipulação de listas e coleções Java.
- *commons-digester-1.7.jar* – API para leitura e *parser* de documentos XML [JAMES, 2001]
- *commons-lang-2.1.jar* – API de utilidades gerais (controle de *Strings*, *Arrays*, etc)
- *commons-logging.jar* – API para gerenciamento de *logs* dentro da aplicação
- *jakarta-regexp-1.4.jar* – API para interpretação de expressões regulares
- *log4j-1.2.8.jar* – API para gerenciamento de *logs* dentro da aplicação (utilizada pela *commons-logging*)
- *xml-apis-2.0.2.jar* – API para leitura e *parser* de documentos XML (utilizada pela *commons-digester*) [JAMES, 2001]

4.2.6.2 Projeto tinitaxi-ejb

O projeto *tinitaxi-ejb* centraliza toda a lógica de negócios do sistema servidor. Esse projeto representa a implementação de um módulo EJB *(Enterprise Java Bean Module)* [BROWN, 2001] [FLOYD, 2001] [MANSON-HALFEL, 2001, 200, 1999] definido na especificação J2EE [CATTEL, 2001]. Nesta implementação, o módulo possui a implementação de um componente EJB chamado *TiniTaxiSessionFacade*. O prefixo *SessionFacade* define o padrão de projeto *(design pattern)* utilizado para implementação do componente. *Session* – Implementação utilizando um EJB *Stateless Session Bean; Facade* – Fachada de acesso único às regras de negócio.

Este componente EJB [FLOYD, 2001] centraliza todo o código de negócio do sistema servidor. Todas as regras de negócio estão implementadas através de métodos da classe

principal deste componente, chamada *TiniTaxiSessionFacadeBean.* Segue abaixo a definição dos métodos implementados por esta classe:

```
public class TiniTaxiSessionFacadeBean extends BaseSessionFacade implements
javax.ejb.SessionBean
```

```
public LoginVo obterLogin(String usuario, String senha) throws IntegrationException,
LoginException

public PontoApoioVo obterPontoApoio(String cdPontoApoio) throws IntegrationException,
PontoApoioNaoEncontradoException

public Integer selecionarPontoApoio(String cdUnidadeMovel, String cdPontoApoio) throws
IntegrationException, PontoApoioNaoEncontradoException

public Integer obterPosicaoFilaUnidadeMovel(String cdUnidadeMovel) throws IntegrationException

public Integer obterPosicaoFilaPontoApoio(String cdPontoApoio) throws IntegrationException

public Collection obterListaChamadasEmAberto() throws IntegrationException

public AreaAtendimentoVo obterAreaAtendimentoChamada(ChamadaVo chamadaVo) throws
IntegrationException

public UnidadeMovelVo obterUnidadeMovelParaAtendimento(AreaAtendimentoVo areaAtendimentoVo)
throws IntegrationException

public void selecionarUnidadeMovelParaAtendimento(ChamadaVo chamadaVo, UnidadeMovelVo
unidadeMovelVo) throws IntegrationException

public void executarProcessamentoBatch() throws IntegrationException

public void aceitarChamadaUnidadeMovel(String cdChamadaAtendimento, String
idUnicoChamadaAtendimento, String cdUnidadeMovel) throws IntegrationException

public void rejeitarChamadaUnidadeMovel(String cdChamadaAtendimento, String
idUnicoChamadaAtendimento, String cdUnidadeMovel) throws IntegrationException

public void finalizarChamadaUnidadeMovel(String cdChamadaAtendimento, String
idUnicoChamadaAtendimento, String cdUnidadeMovel) throws IntegrationException

public AtendimentoVo obterAtendimentoChamadaUnidadeMovel(String cdUnidadeMovel) throws
IntegrationException

public void cancelarChamadaUnidadeMovel(String cdChamadaAtendimento, String
idUnicoChamadaAtendimento, String cdUnidadeMovel) throws IntegrationException
```

Cada método é responsável por implementar determinada operação de negócio dentro do componente. Os dados trafegados são representados através de objetos que implementam um outro padrão de projeto chamado *ValueObjet* (objetos que contêm somente dados). Exemplo: *UsuárioVO* – Classe que contém somente os atributos, e nenhuma operação.

O projeto *tinitaxi-ejb* também possui classes de acesso aos dados. Estas classes foram implementadas seguindo o padrão de projeto *(design pattern)* conhecido como DAO *(Data Access Object)*. A classe *TiniTaxiDAODB2* é a classe principal para o acesso aos dados

A classe de negócio *TiniTaxiSessionfacade* utiliza a classe de *TiniTaxiDAODB2* (implementação da interface *TiniTaxiDAO*) para acesso a base de dados. Todo o acesso à base é feito a partir desta classe.

4.2.6.3 Projeto tinitaxi-web

Este projeto tem como responsabilidade disponibilizar algumas informação do sistema via interface WEB *(Browser)*. No presente trabalho este módulo disponibiliza uma página WEB para consultas sobre as filas de atendimento e permite a abertura de um chamado, simulando o processamento feito por um profissional da companhia de táxi.

4.2.6.4 Projeto tinitaxi-server

O principal objetivo deste projeto é a criação de um servidor *socket* para comunicação do sistema embutido instalado no táxi. Este módulo foi desenvolvido como um *Java Client Module*, conforme descrito anteriormente. Ele se comunica com o sistema embutido fazendo uma ponte entre as regras de negócio e as informações apresentadas no dispositivo móvel. Essas informações serão apresentadas em um *display* conectado ao dispositivo móvel, conforme detalhado na seção 4.2.7. Os dados são enviados do dispositivo móvel para o servidor (*socket server)* que interpreta a requisição e envia a resposta. A resposta é composta de informações de controle e da tela propriamente dita (texto). Dessa forma, o sistema embutido implementado se comparto como um "terminal burro", simplesmente desviando requisições e recebendo dados para a composição da tela a ser apresentada.

Para facilitar o desenvolvimento das telas do dispositivo móvel foi criada uma *framework* para comunicação com o sistema embutido. Essa *framework* define um protocolo de comunicação entre o dispositivo e o servidor *socket*, interpretando os dados da requisição e compondo automaticamente a tela de resposta, baseado em alguns arquivos de configuração. Com isso torna-se mais simples o desenvolvimento de vários serviços (telas), bastando criar alterar alguns arquivos de configuração para criação de novas funcionalidades.

Framework para comunicação (_Socket Server_)

Existe um padrão de projeto conhecido como MVC (*Model View Controller*). Esse padrão de projeto é caracterizado pela separação de três camadas lógicas da aplicação conhecidas como *Model* (camada de negócios), *View* (camada de visualização) e *Controller* (camada de controle). A camada de negócios contém todos os componentes de negócio da

aplicação (normalmente implementados em Java utilizando EJB). A camada de visualização é responsável pela lógica de apresentação do sistema (telas, formatação de campos para a apresentação, internacionalização de *Strings*, etc). Já a camada de controle é responsável pela ponte entre a camada de visualização e a camada de negócios. Nela é implementada toda a lógica de controle do sistema, sendo responsável pela manipulação dos parâmetros de entrada do sistema, repassando-os para a camada de negócio, diminuindo assim o acoplamento entre as camadas de negócio e visualização. Na implementação do sistema servidor o projeto *tinitaxi-ejb* representa a implementação da camada de negócios. A lógica de controle e de visualização ficam implementadas no projeto *tinitaxi-server* através de uma *framework* MVC, construída para facilitar a manipulação das requisições e o envio de dados entre os sistemas cliente/servidor.

Toda a comunicação entre o dispositivo móvel (aplicação cliente) e o servidor é feita via conexão *socket*. Para isso foi implementado um componente para gerenciamento das conexões *sockets* do lado servidor utilizando a API *java.net*, disponível na plataforma J2SE. Foi criado também um protocolo de comunicação entre o cliente e o servidor para facilitar a troca de informações entre o dispositivo móvel e o servidor. Neste protocolo, as mensagens enviadas do cliente (dispositivo móvel) para o servidor seguem o seguinte formato:

SERVIÇO?**PARAM1**=**VALOR**&**PARAM2**=**VALOR&PARAMN=VALOR/n**

Sendo que:

SERVIÇO: Nome do serviço a ser executado do lado servidor (Exemplo: LOGIN)

PARAM1/VALOR : Nome do primeiro parâmetro a ser enviado para o serviço (Exemplo: usuário=22331);

PARAM2/VALOR : Nome do segundo parâmetro a ser enviado para o serviço (Exemplo: senha=12345);

PARAMN/VALOR : Nome do enésimo parâmetro a ser enviado para o serviço;

/n: Caractere para indicar fim de comando.

Segue dois exemplos de comandos enviados do cliente para o servidor:

LOGIN?USR=99912&PASS=12345/n

SELPA?PA=23/n

Do lado servidor, o sistema recebe as requisições enviadas pelo dispositivo móvel, decide qual classe de controle irá responder pelo serviço e repassa a requisição para esta classe. Para isso foi implementado um componente utilizando um outro padrão de projeto conhecido como *Front Controller*. Todas as requisições feitas para o servidor passam por um componente controlador (*controller*) que repassa a responsabilidade da execução do serviço para uma determinada classe, chamada *Action Class*, onde ficam implementadas nas classes de ações (*Actions*) específicas para cada serviço [DAVID, 2001].

O componente utiliza um arquivo de configuração chamado para auxiliar o mapeamento de serviços e ações. Este arquivo é um documento XML [JAMES, 2001] onde se encontram as seguintes informações (*tags*): *action-mappings*, *global-exceptions*, *global-forwards* e *filter-mappings.*

Tag action-mapping: Descreve a configuração das classes de controle (*actions*) do sistema. Dentro desta *tag* se encontra uma lista de *actions*, sendo que cada *action* é composta pelas seguintes informações: *Action Path*, *Action Type*, *Forwards.*

```
<action path="LOGINF" type="br.ufrj.nce.se.tinitaxi.server.action.LOGINFTiniAction">
       <forward name="fwd-sucesso" screen="login-form.screen"/>
</action>
```

Action Path: Representa o nome do serviço a ser mapeado por esta *action* (*LOGINF* no exemplo acima). Através desta informação o componente de controle consegue repassar a requisição para uma classe de controle.

Action Type: Representa a classe Java que implementa a lógica de controle para esta *action* (*br.ufrj.nce.se.tinitaxi.server.action.LOGINFTiniAction* no exemplo acima). O componente de controle identifica através do parâmetro *action path* qual *action* será executada para o serviço solicitado, em seguida executa o código de controle implementado pela classe Java informada.

Forwards: Representam as telas a serem chamadas a partir da *action* implementada. Esse *forward* configura (através de uma referencia) a tela a ser chamada, de acordo com a lógica implementada na classe *action.* Por exemplo, a *action* de login possui um *forward* chamado *fwd-sucesso*. Esse *forward* aponta para uma tela (*screen*) chamada *login-*

form.screen. Essa tela (*screen*) descreve as informações a serem apresentadas na tela do usuário. Segue abaixo um exemplo de uma *screen*:

```
.:TINI TAXI - V1.0:.
LOGIN: [_    ]
SENHA: [     ]
A)OK         B)APAGAR
```

Para cada *screen* existe um mapeamento descrito em um outro arquivo de configuração chamado *screens.xml*. Neste arquivo se encontram as informações relacionadas à tela a ser enviada para o cliente em conjunto com o *metadado* desta tela .

Segue abaixo um exemplo com as informações relacionadas à tela *login-form.screen*:

```
<screen name="login-form.screen">
        <sources>
                <path columns="20" rows="4" path="screens/LCD20x4/login-form.screen"/>
        </sources>

        <mapping>
                <server-fields>
                </server-fields>

                <form-fields>
                        <field name="USR" size="5" />
                        <field name="PASS" size="5" />
                </form-fields>

                <buttons>
                        <button name="OK" key="A" path="LOGIN"></button>
                        <button name="APAGAR" key="B" path="@APG"></button>
                </buttons>
        </mapping>
</screen>
```

A configuração da *screen* está dividida em duas partes: *sources* e *mapping*. Em *source* estão contidas as informações relacionadas à tela propriamente dita (caracteres a serem apresentados). Essas informações são referenciadas através do atributo *path* (caminho *XPath*: screen/sources/path). Este *path* contém a referencia para um arquivo com as definições da tela.

Exemplo: Arquivo *screens/LCD20x4/login-form.screen*

```
.:TINI TAXI - V1.0:.
LOGIN: [$    ]
SENHA: [$    ]
A)OK         B)APAGAR
```

Para implementação do lado cliente (dispositivo móvel) foi utilizado um *display* de 20 (vinte) colunas por 4 (quatro) linhas, conforme descrito na seção 4.2.7. As definições da tela

contidas no arquivo *login-form.screen* obedecem às restrições de tamanho da tela, sendo que o total de caracteres enviados ao cliente para apresentação nunca ultrapassa 80 (oitenta) caracteres.

Segue abaixo o fluxo de telas (*screens*) e seus correspondentes arquivos de configuração:

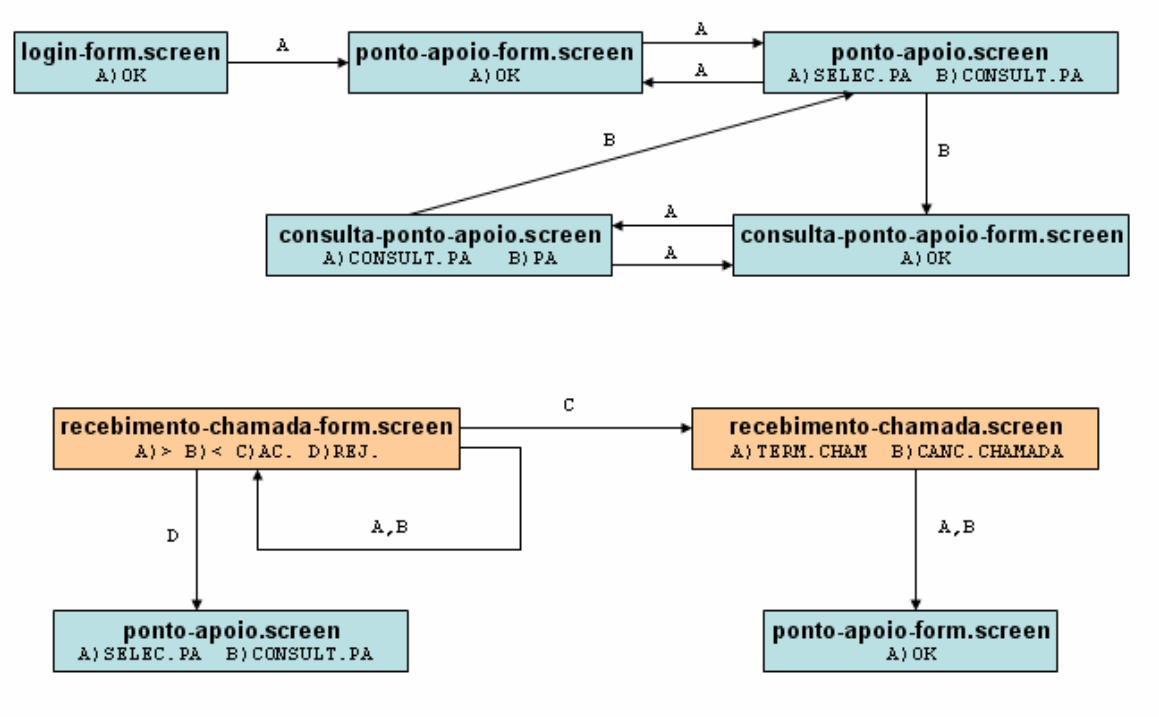


Figura 25 – Fluxo de talas do sistema embutido implementado (arquivos *.screens*).

Na configuração da *screen* (*login-form.screen*) existe uma outra *tag* chamada *mapping*. Nesta *tag* estão contidas as informações relacionadas ao *metadado* da tela (códigos de controle). A *tag* está subdividida em três outras *tags*: *server-fields*, *form-fields* e *buttons*.

? *server-fields*: Informações relacionadas aos campos da tela enviados do cliente para o servidor:

Por exemplo, na requisição **LOGIN?USR=99912&PASS=12345/n**, os campos mapeados em *server-fields* são: USR e PASS.

? *form-fields*: Informações relacionadas aos campos de entrada de dados do usuário, que retornam para o cliente através de *input fields*. Estes campos são endereçados através do caractere $, contido na definição da *screen*.

Por exemplo, na resposta enviada ao cliente para a requisição **LOGIN/n,** o sistema devolve uma *screen* com dois campos do tipo *form-fields*: USR, PASS.

```
LOGIN: [$    ]
SENHA: [$    ]
```

Neste caso, o cliente que recebe esta tela e já sabe que deve controlar dois campos para *input* de dados (campos indicados com $). O tamanho destes campos também se conontra na *tag form-fields:*

```
<form-fields>
        <field name="USR" size="5" />
        <field name="PASS" size="5" />
 </form-fields>
```

? *buttons*: Representam as ações que o cliente pode executar após receber a tela de resposta após uma requisição. No caso da *screen login-form.screen*, os *buttons* associados são: "A" para enviar os dados e "B" para apagar os campos. Os *buttons* são associados a 4 (quatro) teclas de função definidas no dispositivo móvel (A, B, C, D), descritas na seção 4.2.7. O *button* "A" representa o comando a ser enviado para o servidor se a tecla de função "A" for pressionada.

De acordo com a configuração abaixo para a *screen login-form.screen*, quando pressionada a tecla "A" será enviado o comando LOGIN para o servidor, em conjunto com os parâmetros definidos na *tag form-fields.*

```
<form-fields>
        <field name="USR" size="5" />
        <field name="PASS" size="5" />
</form-fields>

<buttons>
        <button name="OK" key="A" path="LOGIN"></button>
        <button name="APAGAR" key="B" path="@APG"></button>
</buttons>
```

No exemplo citado, caso seja pressionada a tecla de função "B" será executado um comando específico chamado *@APG*. Este comando foi criado para indicar ao cliente (dispositivo) que deve ser feita a limpeza dos campos de entrada (inputs). Este comando especificamente não envia nenhuma informação ao servidor (apenas limpa os valores dos campos na tela do dispositivo cliente e posiciona o *cursor* para digitação no primeiro campo).

Para um entendimento melhor do fluxo de informações do sistema, segue abaixo um diagrama de estados representando as telas do sistema embutido:

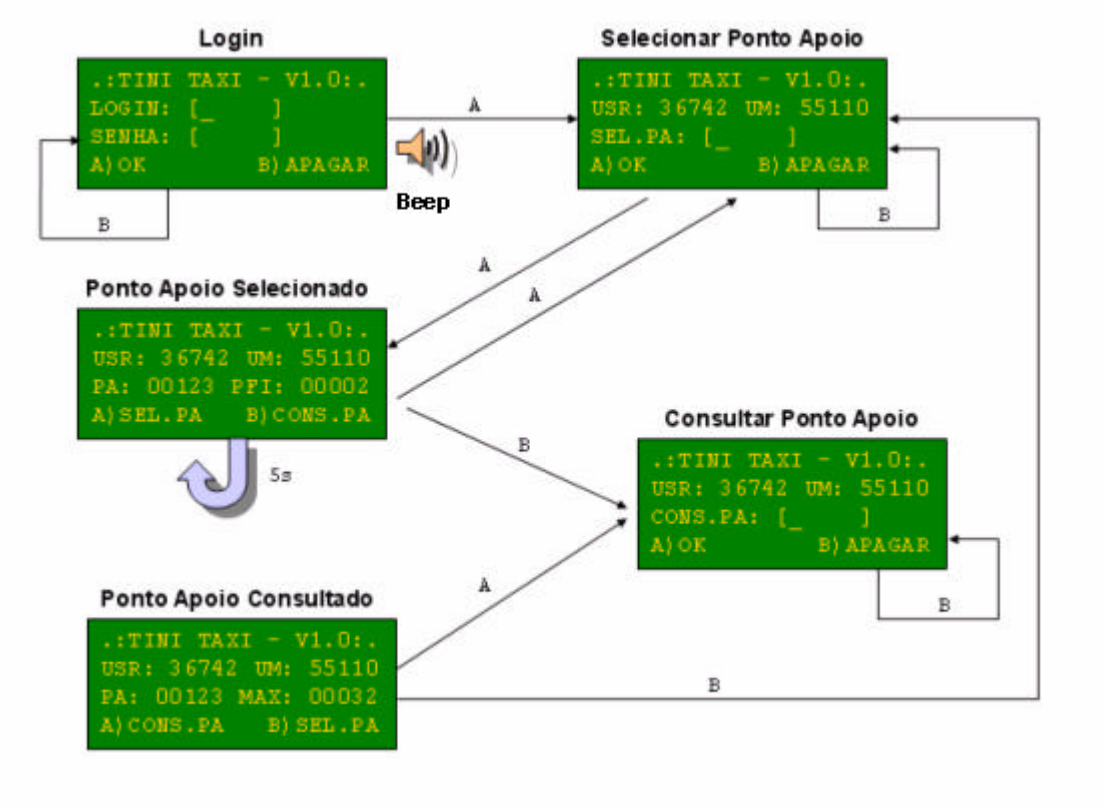


Figura 26 – Fluxo de telas do sistema embutido implementado *(Display 20x4)* – Parte 1.

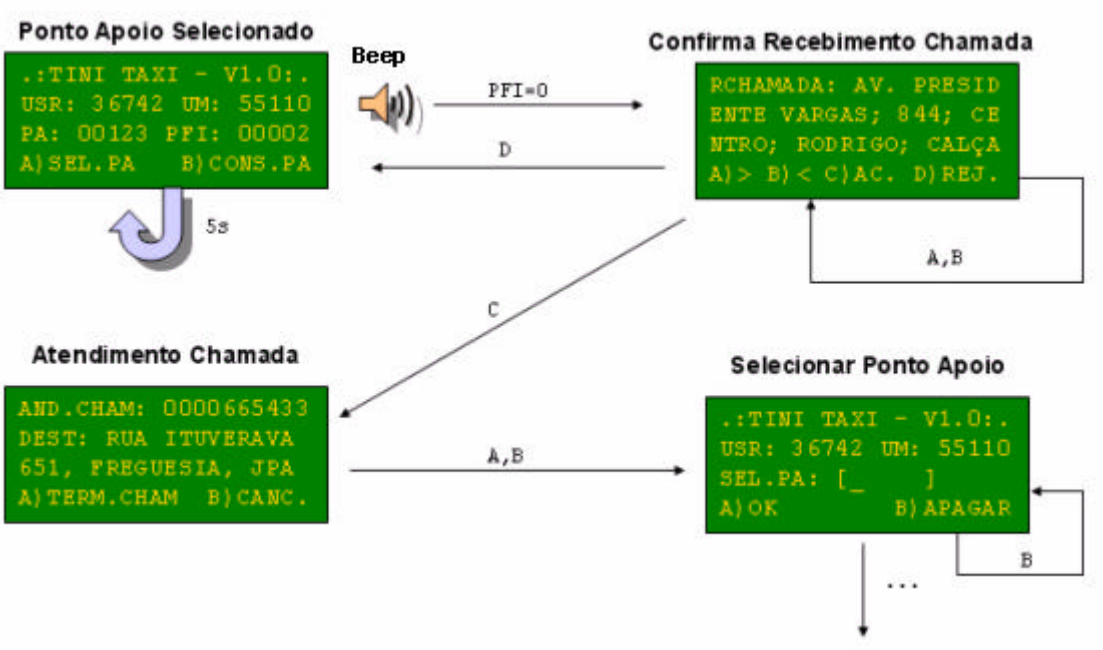


Figura 27 – Fluxo de telas do sistema embutido implementado *(Display 20x4)* – Parte 2.

Do lado servidor também foi implementado um protocolo para envio de mensagens de resposta ao cliente (unidade móvel), que segue o seguinte formato:

<S>INFO_ TELA</S> <F>INFO_CAMPOS</F> <B>INFO_BOTOES</B> /n

Sendo que:

INFO_TELA: São as informações referentes aos dados a serem apresentados na tela (*screen*). Exemplo:

```
.:TINI TAXI - V1.0:.LOGIN: [_    ]         SENHA: [     ]
A)OK        B)APAGAR
```

INFO_CAMPOS: São as informações referentes aos campos lógicos a serem controlados pelo cliente (*matadados*). Estas informações seguem o seguinte formato:

NOME_CAMPO=TAMANHO_CAMPO;

Exemplo: USR=5;PASS=5;

INFO_BOTOES: São as informações referentes às teclas de função a serem manipuladas pelo cliente e suas respectivas ações (*action paths*). Estas informações seguem o seguinte formato:

NOME_BOTAO=ACAO;

Exemplo: A=SELPA;B=CONPA;C=LOGIN;

Através destas informações o cliente sabe manipular a entrada de dados nos campos (como se fossem formulários *Web*), controlar o fluxo do cursor nos campos (através da ordem definida na *tag* INFO_CAMPOS) e controlar as ações referentes às teclas de função do cliente (A, B, C e D), conforme descrito na seção 4.2.7.

4.2.6.5 Projeto tinitaxi-batch

Este projeto tem como objetivo implementar o processamento *batch* para gerenciamento das filas de atendimento dos pontos de apoio. Segue abaixo um trecho de código da principal classe deste projeto:

GerenciadorChamadasBatch.java

```
public class GerenciadorChamadasBatch
```

```
        public static void main(String[] args) throws IntegrationException, RemoteException {
                GerenciadorChamadasBatch gerenciadorChamadasBatch =
```

```
                                                new GerenciadorChamadasBatch();
                gerenciadorChamadasBatch.run();
        }
```

```
        private void run() throws IntegrationException, RemoteException {
                TiniTaxiSessionFacade tiniTaxiSessionFacade = this.getTiniTaxiFacade();

                while (true) {
                        this.sleep();

                        // executa o processamento batch de seleção de unidades móveis para
                           atendimento (ponto de apoio + area de atendimento)
                        tiniTaxiSessionFacade.executarProcessamentoBatch();
                }
        }
```

4.2.6.6 Projeto tinitaxi-framework

Este projeto contém as classes comuns a toda aplicação servidora. Todas as superclasses (pai de outras classes) estão implementadas através de classes comuns dentro deste projeto. As principais classes deste projeto são: *BaseAction, BaseActionForm, BaseBusinessDelegate, BaseDataAccessObject, BaseSessionFacade, BusinessException e ServiceLocator*.

4.2.6.7 Projeto tinitaxi-containeir

O objetivo deste projeto é armazenar todas as informações relacionadas à configuração do servidor de aplicações, neste caso *WebSphere Application Server*, versão 5.1. Através deste projeto é possível instalar e configurar um novo servidor de aplicações mantendo as mesmas configurações para execução da aplicação implementada neste trabalho. Os principais arquivos de configuração contidos neste projeto são: *log4j.properties, memento.xml, cell.xml, resources.xml, variables.xml, virtualhosts.xml* e *admin-console.ear*.

4.2.7 Implementação (Aplicação Cliente)

A proposta deste trabalho se resume na implementação de um dispositivo móvel (*hardware*) em conjunto com um sistema automático (*software*) para automatização do modelo de operação presente em algumas companhias de táxi. Nesta seção apresenta-se o detalhamento do dispositivo móvel como canal de comunicação entre o taxista e o servidor e a implementação dos componentes de *software* utilizados pela camada cliente.

A aplicação cliente foi construída utilizando a plataforma *Java 2 Standard Edition* (J2SE). Essa aplicação é composta por dois subprojetos: ***tinitaxi-client*** e ***tinitaxi-ppp***. Para a implementação da aplicação cliente também foi construído um dispositivo móvel baseado na plataforma de *hardware* TINI, que representa o computador de bordo do taxista. Este dispositivo se comunica com a central (sistema servidor) através de um telefone celular, utilizando uma rede GPRS para comunicação de dados.

4.2.7.1 Dispositivo Móvel (TiniTaxi)

Para implementação do dispositivo móvel foi utilizada a placa TINI *Board Model* TBM390, oferecida pela plataforma de *hardware* TINI. Esta placa atua como processador central do dispositivo móvel, sendo responsável pelo processamento da comunicação entre o cliente e o servidor, através de um celular. Ela também é responsável pelo gerenciamento das interfaces principais com o usuário (teclado + *display*) e pela conexão GPRS entre o dispositivo e a central.

Para implementação do dispositivo em questão foram utilizados os seguintes componentes de *hardware*:

1. TINI TBM390 - *Board Model* 390 (Microcontrolador DS80C390, 1MB de memória RAM, 2MB de memória *Flash*, duas UARTs integradas e interface E*thernet* RJ45), fabricante *Dallas Smiconductors;*

2. STEP *Socket Board* (72-pin SIMM - Conector para TINI TBM390), fabricante *Systronix;*

3. Placa SBX *Socket Board Extension* (para conexão com o teclado e *display*), fabricante *Systronix;*

4. Teclado (*Keypad*, 4x4 tipo membrana), fabricante *Systronix* (conexão na placa STEP através da placa SBX);

5. *Display* (20x4 LCD caractere, tamanho: 3.88" x 2.37" x 0.47". área visível: 3" x 1". Tamanho do caractere: 4.7mm x 2.90mm. LED *backlight*), fabricante *Samtron* (conexão na placa STEP através da placa SBX);

6. Celular GSM (modelo T68i, operadora TIM, plano *TIM CONNECT FAST* que inclui pacote de conexão GPRS), fabricante Sony Ericsson;

7. Cabo serial RS232 (DTE/DTE) para conexão entre a placa STEP (UART) e o celular;

Para implementação da comunicação entre a placa TBM390 e o teclado (*keypad*) foi utilizado uma API fornecida pela própria fabricante (*Systronix*). Segue abaixo um trecho de código utilizando esta API:

```
int oKey = 0;
int state = 0;

while (true) {
	int key = ~cSBX.getInput(SBX2.KEYPAD);

	if (key == oKey) {
		sleep(30);
		continue;
	}

	if (state == 0) {
		if ((key & 0x80) != 0)
			state = 1;
	} else if (state == 1) {
		if ((key & 0x80) == 0)
			return key;
	}
	oKey = key;
}
```

O teclado (*keypad*) 4x4 foi dividido da seguinte forma: teclas numéricas de 0 a 9, uma tecla de função para auxiliar a entrada de dados (P), facilitando a troca de um campo para outro (formulário de entrada de dados) e mais quatro teclas de função (A, B, C e D) para execução de funções específicas (dependendo da tela apresentada). A figura abaixo ilustra a distribuição das teclas:

1	2	3	4
5	6	7	8
9	0		P
A	B	C	D

Figura 28 – Distribuição das teclas no *keypad* – Sistema TiniTaxi.

Para implementação da comunicação entre a placa TBM390 e o *display* (20x4 LCD caractere) foi utilizado também uma API fornecida pela *Systronix*. Segue um trecho de código utilizando esta API:

```
Misc.setBacklight(Misc.DIM);

Misc.setContrast(Misc.DARK, (byte)100);
cLCD = new LCD((byte)20, LCD.LINES4, LCD.FONT5X7);
cLCD.initLCD(LCD.cursorOFF, LCD.displayON);

cLCD.setLine(LCD.LINE1);
cLCD.write(data.substring(0, 20));
```

Na comunicação entre a placa TBM390 e o celular GSM foi utilizado a interface serial disponível na placa STEP. Nesta implementação, o celular se comporta como a ponte de

comunicação entre o dispositivo e o servidor através da rede GPRS. Toda comunicação é feita vai conexão serial entre a placa STEP e o celular T68i através de comando AT, utilizando um cabo serial personalizado.

A figura abaixo mostra a pinagem do cabo serial utilizado para conexão.

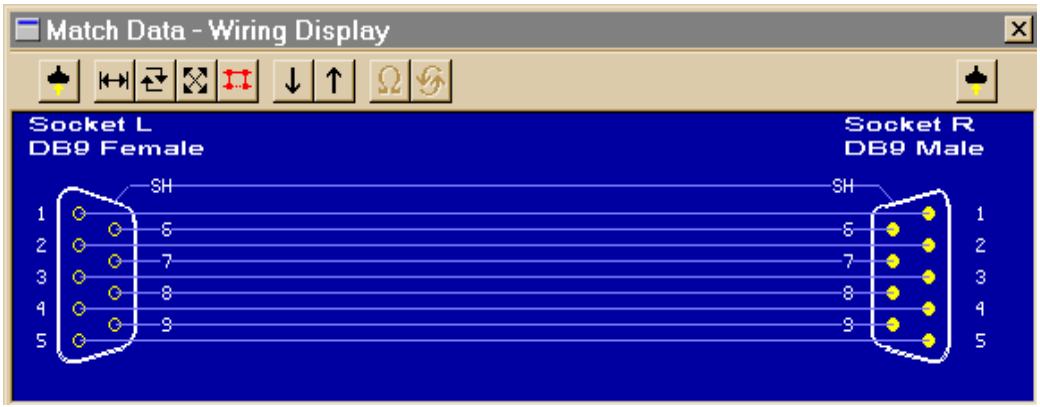


Figura 29 – Configuração do cabo serial para conexão do celular T68i a placa.

Obs: Lado esquerdo DTE (TINI); Lado direito DCE – (celular T68i).

Para comunicação GPRS o celular precisa ser configurado com determinados comandos AT específicos para este tipo de conexão. Estes comandos habilitam o celular para se comunicar em uma rede privada da operadora, com acesso publico a Internet (baseado no protocolo IP). No entanto, como a comunicação entre o celular e a placa STEP é feita utilizando interface serial RS232 (e não interface de rede e*thernet*), se faz necessária a utilização do protocolo PPP (*Point to Point Protocol*) para implementação do protocolo IP sobre interface serial. O protocolo PPP originalmente surgiu como um "encapsulamento" para transportar pacotes de dado utilizando protocolo IP sobre *links* ponto a ponto.

A implementação da comunicação GPRS entre o dispositivo e o servidor sobre protocolo PPP se encontra no projeto *tinitaxi-ppp*.

4.2.7.2 tinitaxi-ppp

A implementação do código para configuração e comunicação GPRS está contida neste projeto. Ele contém apenas uma classe chamada *TiniTaxiPPP*, responsável pela lógica de conexão PPP e pela comunicação GPRS entre cliente e servidor.

Segue abaixo os comandos AT para inicialização da conexão GPRS (celular):

- **gprsInitMsgOne**: "AT\r"; Resposta esperada: "OK"
- **gprsInitMsgTwo**: = "AT+CGATT?\r"; Resposta esperada: "OK"
- **gprsInitMsgThree** ="ATE0\r""; Resposta esperada: "OK"
- **gprsInitMsgFour** = "AT+CGDCONT=1,"IP","tim.br","0.0.0.0"\r"; Resposta esperada: "OK"
- **gprsInitMsgFive** = "AT+CGQREQ=1,0,0,3,0,0\r"; Resposta esperada: "OK"
- **gprsInitMsgSix** = "AT+CGQMIN=1,0,0,3,0,0\r"; Resposta esperada: "OK"
- **gprsInitMsgSeven** ="AT+IPR=57600\r"; Resposta esperada: "OK"
- **gprsInitMsgEight** ="ATZ\r"; Resposta esperada: "OK"
- **gprsInitMsgNine** = "ATE1V1\r"; Resposta esperada: "OK"
- **gprsInitMsgTen** = "AT+IFC=2,2\r"; Resposta esperada: "OK"
- **gprsInitMsgEleven** = "ATS0=0\r"; Resposta esperada: "OK"
- **gprsInitMsgTwelve** = "AT+CGQREQ=1,0,0,3,0,0\r"; Resposta esperada: "OK"

Seqüência utilizada para inicialização do celular

```
private ModemCommand[] dialSequence = {
        gprsInitMsgOne,
        gprsInitMsgTwo,
        gprsInitMsgFour,
        gprsInitMsgFive,
        gprsInitMsgSix,
};
```

A compilação deste projeto é feita através de um *script* (arquivo *.bat*). Este *script* é responsável pela compilação e geração de um binário que é instalado na placa TINI TB390. Este arquivo possui uma extensão *.tini*, sendo executado dentro do sistema operacional oferecido pela plataforma TINI no dispositivo móvel (conforme descrito no Capítulo 3). Na execução do código binário, a classe principal *TiniTaxiPPP* envia os comandos AT para o celular via conexão serial para inicialização do *modem* (conexão GPRS) e disponibiliza uma conexão PPP através de uma nova interface de rede criada pela própria classe chamada *pppClient*.

Para funcionamento desta interface criada (*pppClient*) são necessárias algumas modificações nas configurações de rede da placa TINI TBM390. Essas modificações permitem o “roteamento” dos pacotes de dados endereçados a placa TINI para a nova interface de rede PPP. Segue abaixo essas configurações:

1. Configurar a interface padrão *Ethernet* para um endereço não “roteavel” –Ação: Ajustar a mascara de rede desta interface para 255.255.255.255 (Comando TINI: *ipconfig* –m 255.255.255.255)

2. Configurar a interface de rede PPP como interface padrão – Ação: Ajustar no código o *flag defaultInterface* para *true – setDefaultInterface(true)*.

4.2.7.3 tinitaxi-client

Neste projeto estão as classes de controle para comunicação do cliente com o servidor. Nele também estão contidos os componentes necessários para interpretação dos parâmetros de retorno do servidor, bem como a lógica de manipulação dos campos de entrada de dados via teclado e apresentação de dados via *display*. Toda a lógica de controle está centralizada em uma classe principal chamada *ClientApplication*. Foi criada uma interface *Java* chamada *ClientContyroller* para deixar transparente para lógica de controle qual o dispositivo utilizado na implementação. Com isso, foi possível reusar a mesma lógica de controle para duas implementações distintas, uma para a placa TINI (dispositivo de *hardware*) e outra para um simulador utilizando uma API para criação de aplicativos visuais em Java chamada *Java Swing*.

Segue abaixo o código da interface comum para ambas às implementações (*ClientContyroller*):

```
public interface ClientController {
        public void initialize() throws ClientControllerException;
        public void writeToDisplay(String data) throws ClientControllerException;
        public int readFromKeyboard() throws ClientControllerException;
        public void beep() throws ClientControllerException;
}
```

A implementação desta interface para a plataforma de *hardware* utiliza a API fornecida pela *Systronix* para comunicação com o teclado e o *display* via placa SBX (conectada a placa STEP). É importante frisar que o código de controle foi desenvolvido de uma maneira independente da plataforma de *hardware* utilizada, sendo possível que seja criada facilmente uma nova implementação para qualquer outra plataforma (*hardware* ou outro simulador via *software*).

Dentro deste projeto a classe responsável pela implementação da interface *ClientContyroller* para a plataforma de *hardware* é a classe *SBXBoardClientController*. Já a

implementação desta interface para o simulador fica associada à classe *SwingClientController.* Segue abaixo o *layout* do simulador utilizando a API *Java Swing*:

Figura 30 – Aplicação *Java Swing* para simulação de uma unidade móvel (táxi).

4.2.8 Configuração do Ambiente de Desenvolvimento

Foi adotada como ferramenta de desenvolvimento para este trabalho o *Websphere Studio Application Development* (*WSAD*) da IBM, na versão 5.1. Esta seção descreve passo a passo como criar uma nova *workspace* dentro do WSAD para importação dos projetos desenvolvidos neste trabalho, facilitando futuros trabalhos.

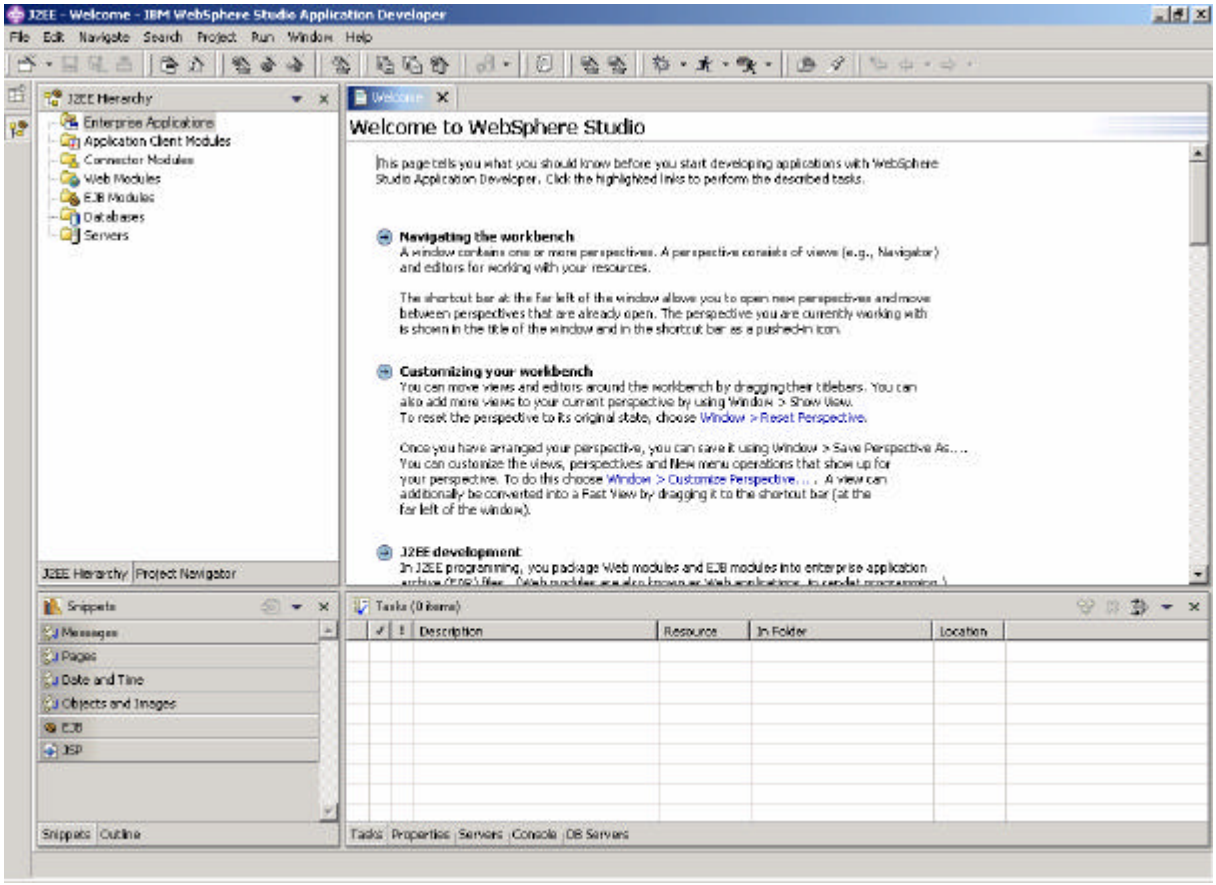
4.2.8.1 Criação da nova workspace (WSAD 5.1)

Passo 01) Criar um atalho na área de trabalho da máquina para o WSAD, apontando para a seguinte URL: *<WAD_HOME>\wsappdev.exe -data "<TINITAXI_HOME>\workspace"*, sendo que:

- WSAD_HOME = Caminho da instalação do *Websphere Studio*

- TINITAXI_HOME = Caminho principal do projeto TINITAXI na máquina de desenvolvimento. Obs: Copiar a pasta do seguinte caminho: *CD DISSERTACAO\Fontes do Sistema\Fontes Java (Workspace WSAD)*

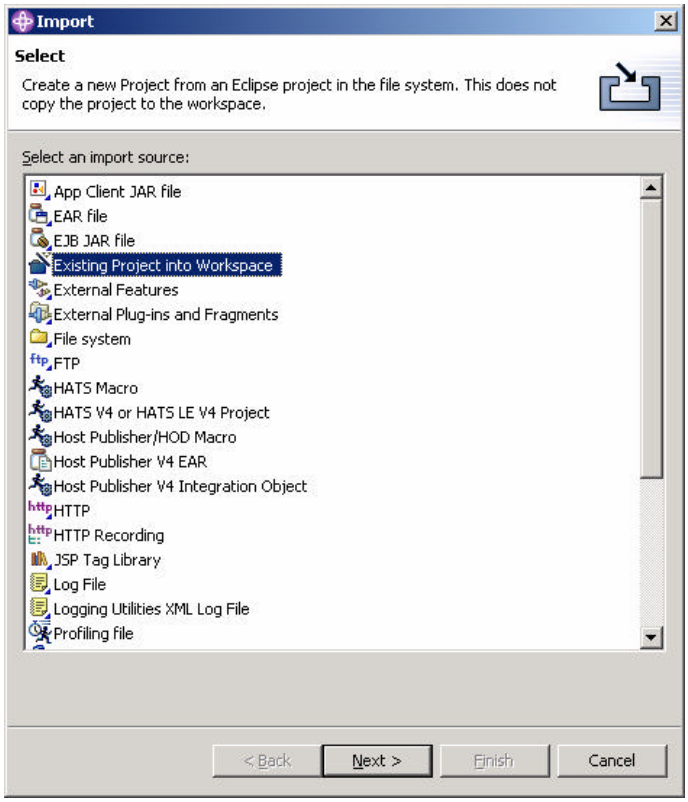
Passo 2) Abrir a nova *workspace* através do atalho criado.



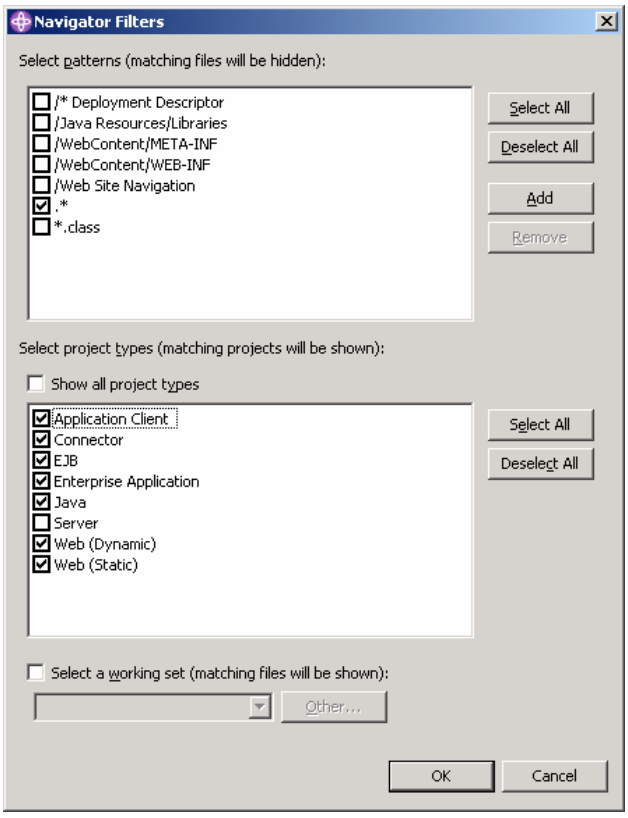
Passo 3) Importar os projetos dentro da *workspace* criada através da opção no *Websphere Studio. Obs:* Os seguintes projetos devem ser importados dentro do WSAD, nesta seqüência:

1. *<TINITAXI_HOME>\workspace\tinitaxi-framework*
2. *<TINITAXI_HOME>\workspace\tinitaxi-ear*
3. *<TINITAXI_HOME>\workspace\tinitaxi-ejb*
4. *<TINITAXI_HOME>\workspace\tinitaxi-web*
5. *<TINITAXI_HOME>\workspace\tinitaxi-batch*
6. *<TINITAXI_HOME>\workspace\tinitaxi-server*
7. *<TINITAXI_HOME>\workspace\tinitaxi-containeir*

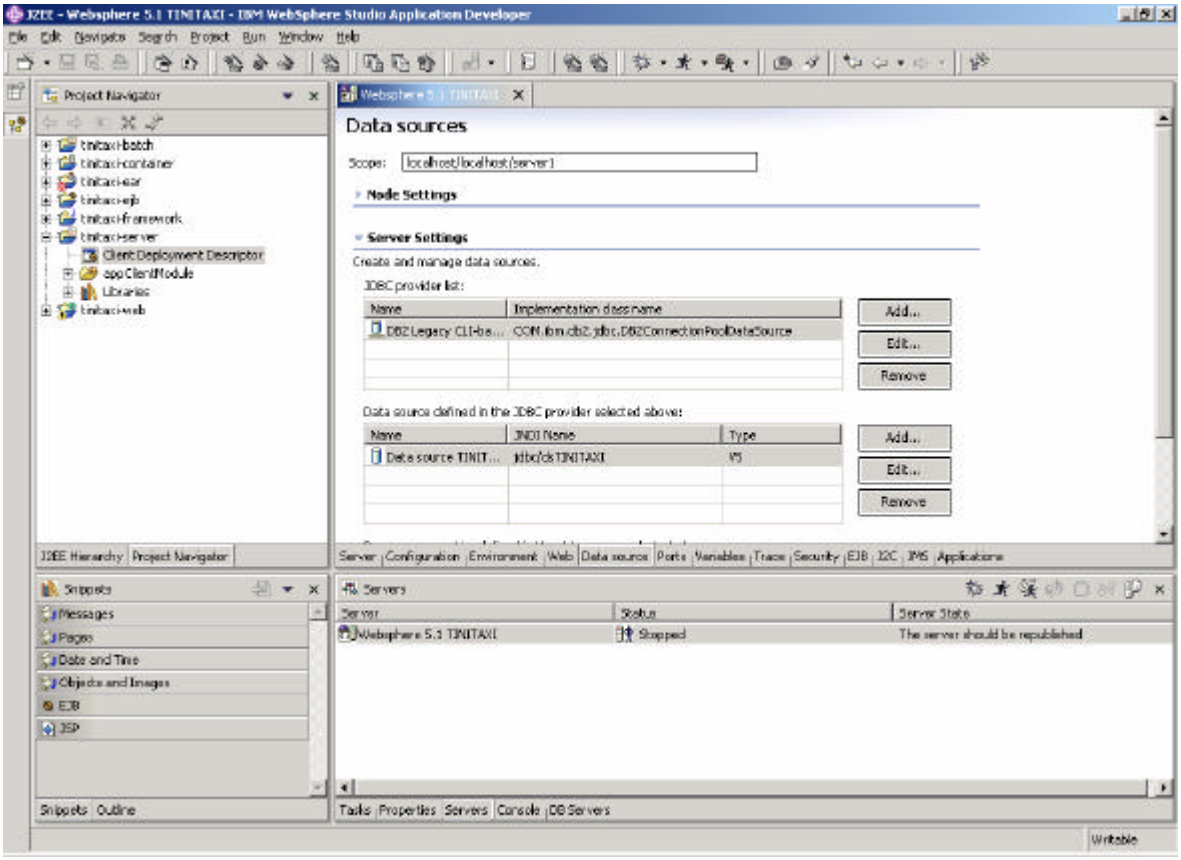
Para importação deve ser utilizada a seguinte opção no menu do *Websphere Studio*: *File > Import > Existent Project into Workspace*, conforme ilustrado na figura abaixo:



Passo 4) Após a importação dos projetos, deve ser habilitada no WSAD a opção para mostrar o projeto *Server* dentro da *workspace* utilizada: *Project Navigator* > *Filters* > *Select Project Types* > *Server.*



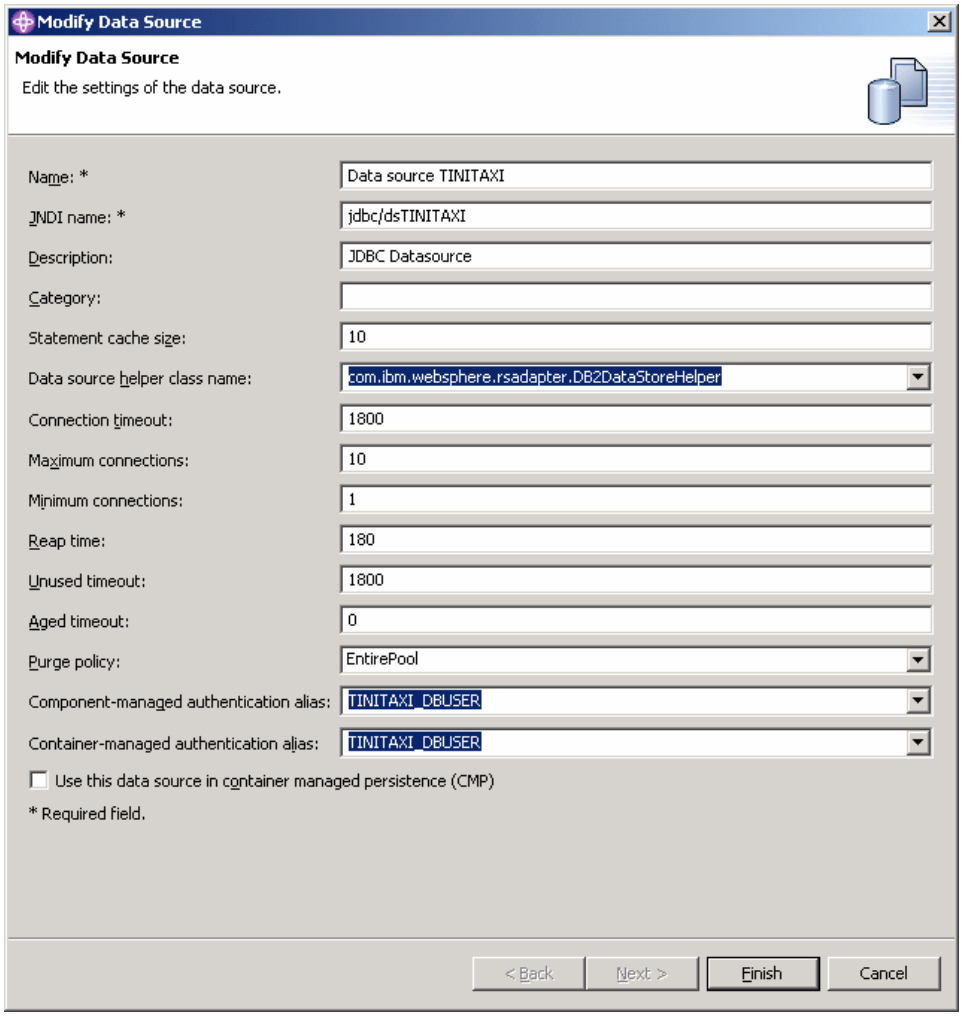
Passo 5) Configuração do acesso ao Banco de dados TINITAXI. Para isso, deve ser criado um *datasource* dentro do *Websphere Application Server* instalado na workspace criada (através do projeto *tinitaxi-containeir*).



Criar um novo *datasource* no menu WSAD: *Servers > Websphere 5.1 TINITAXI > Data source > JDBC Provider List DB2 Legacy CLI-based Type 2 > Add*

As seguintes opções devem ser preenchidas na tela de criação no *datasource*:

- *Name:* ***Data source TINITAXI***
- *Data source helper class name:* ***com.ibm.websphere.rsadapter.DB2DataStoreHelper***
- *Component-managed authentication alias:* ***TINITAXI_DBUSER***
- *Container-managed authentication alias:* ***TINITAXI_DBUSER***



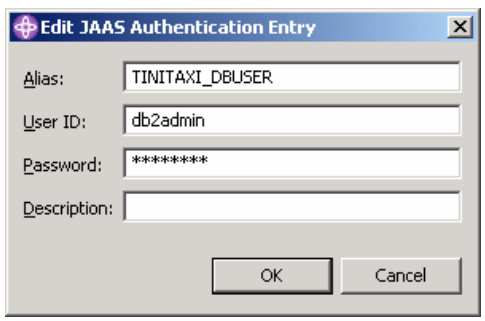
Passo 6) Configuração do usuário para acesso a base de dados no menu WSAD: *Servers > Websphere 5.1 TINITAXI >Security > JAAS Authentication Entry > Add*

As seguintes propriedades devem ser preenchidas na tela de criação:

? *Alias:* ***TINITAXI_DBUSER***

? *User ID:* ***db2admin***

? *Password:* ***db2admin***



Passo 7) Configuração do nome do banco de dados - menu WSAD: *Servers > Websphere 5.1 TINITAXI > Data source > TINITAXI data source > resource properties defined in datasource > databaseName > Edit*

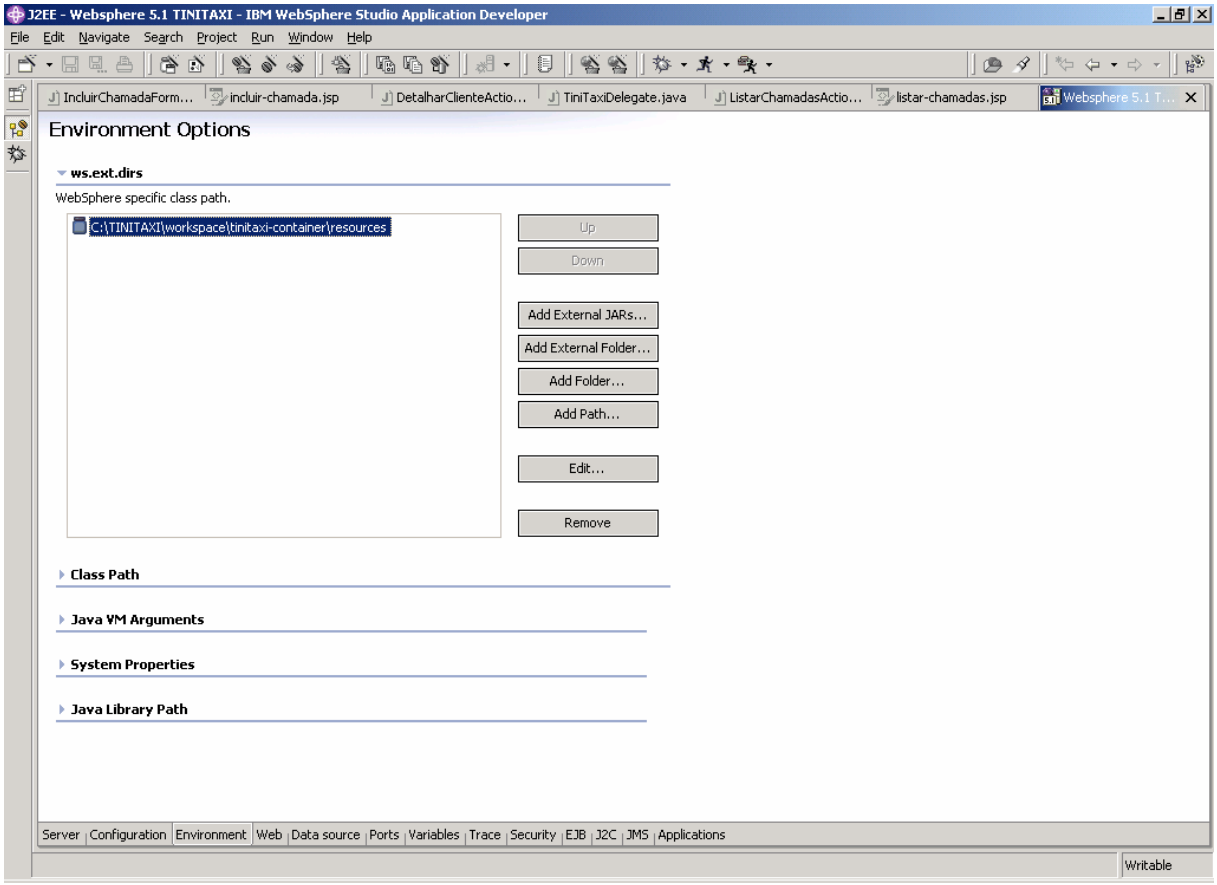
As seguintes propriedades devem ser preenchidas:

- *Name:* ***databaseName***
- *Type:* ***java.lang.String***
- *Value:* ***TINITAXI***
- *Required:* ***checked***

Passo 8) Configuração do caminho das classes (*ClassPath*) dentro do *Websphere Application Server* instalado no WSAD: *Servers > Websphere 5.1 TINITAXI > Enviroment > ws.ext.dirs > Add Path*

Devem ser preenchidas as seguintes propriedades:

- *Path: <TINITAXI_HOME>\workspace\tinitaxi-containeir\resources*



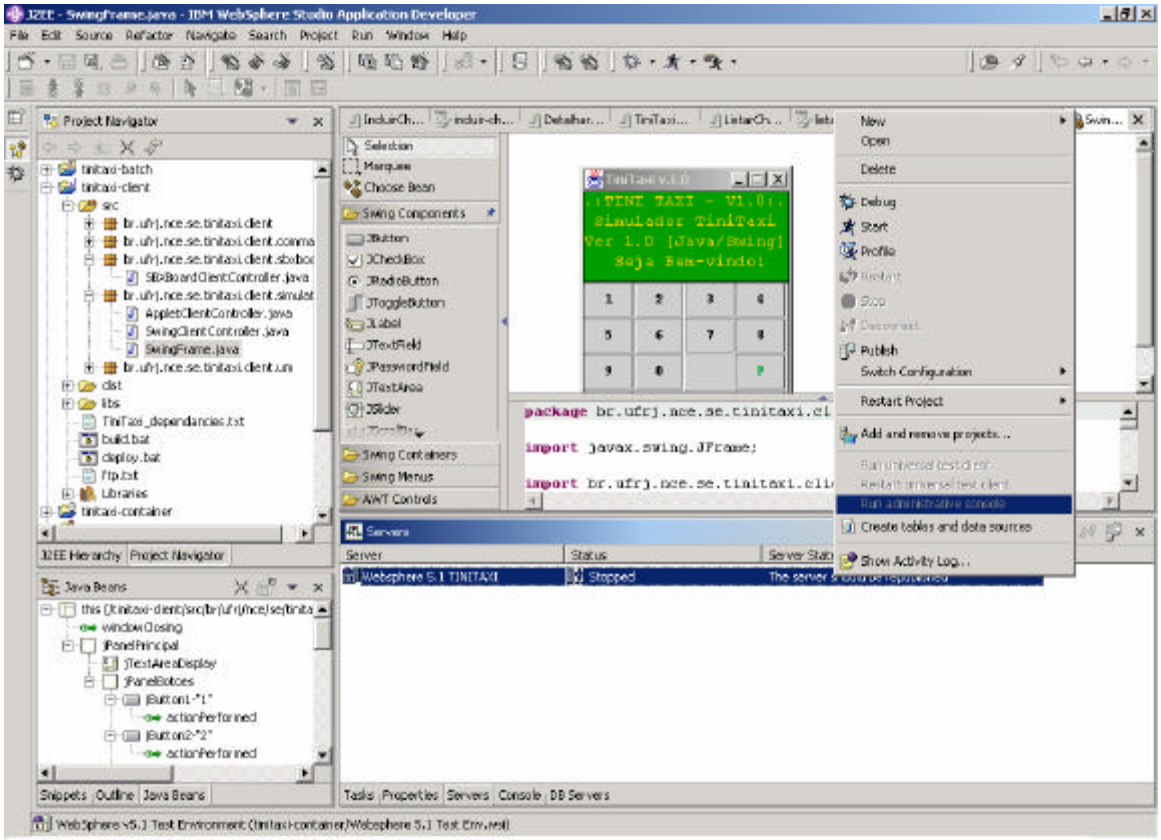
Passo 9) Importar os seguintes projetos dentro da *workspace* criada através da opção no *Websphere Studio:*

1. *<TINITAXI_HOME>\workspace\tinitaxi-client*
2. *<TINITAXI_HOME>\workspace\tinitaxi-ppp*

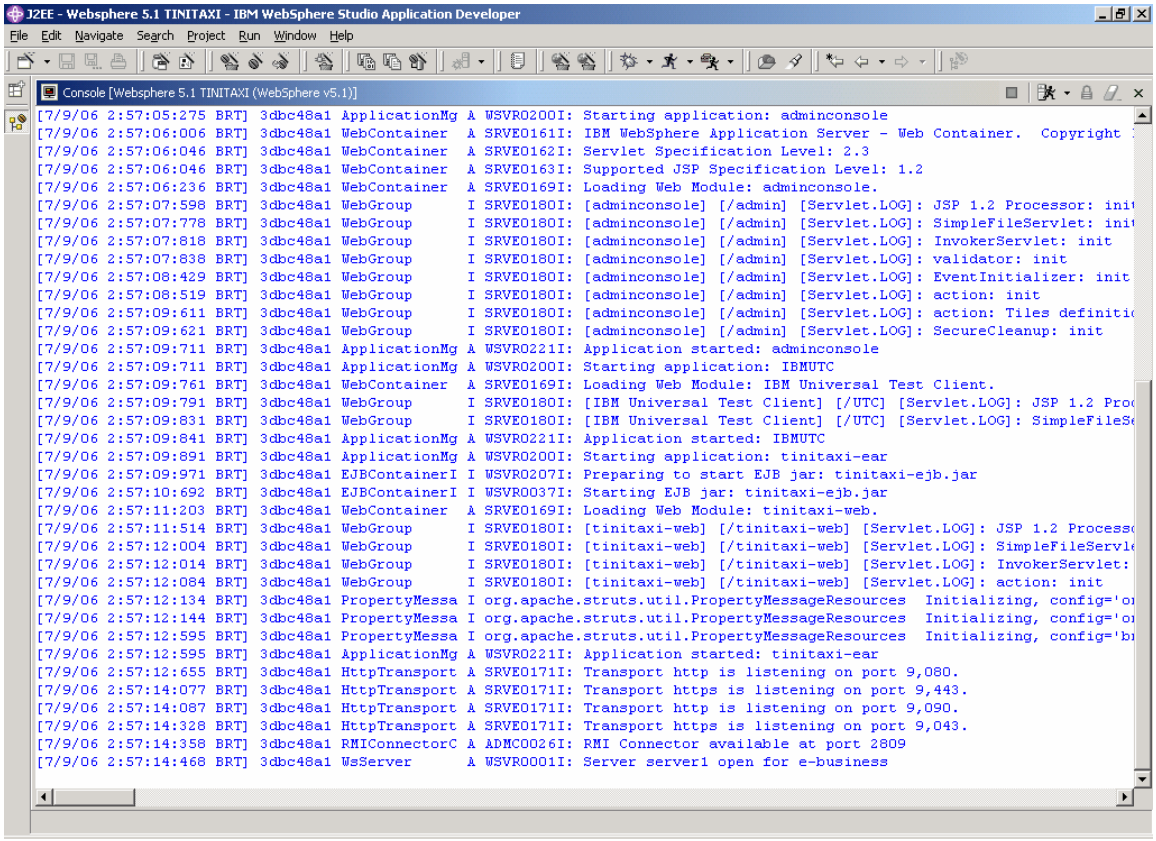
Para importação deve ser utilizada a seguinte opção no menu do *Websphere Studio*: *File > Import > Existent Project into Workspace.*

4.2.8.2 Início dos servidores (Websphere Application Server TINITAXI)

Passo 1) Iniciar o servidor *Websphere Application Server* dentro do WSAD: *Servers > Websphere 5.1 TINITAXI > Start Server*



Após a inicio do servidor, observar o *log* do servidor na janela *console* do WSAD. Deve ser observada a ultima linha do *log*, conforme figura abaixo:



Obs: O seguinte texto deve aparecer na *console*: ***Server server1 open for e-business.***

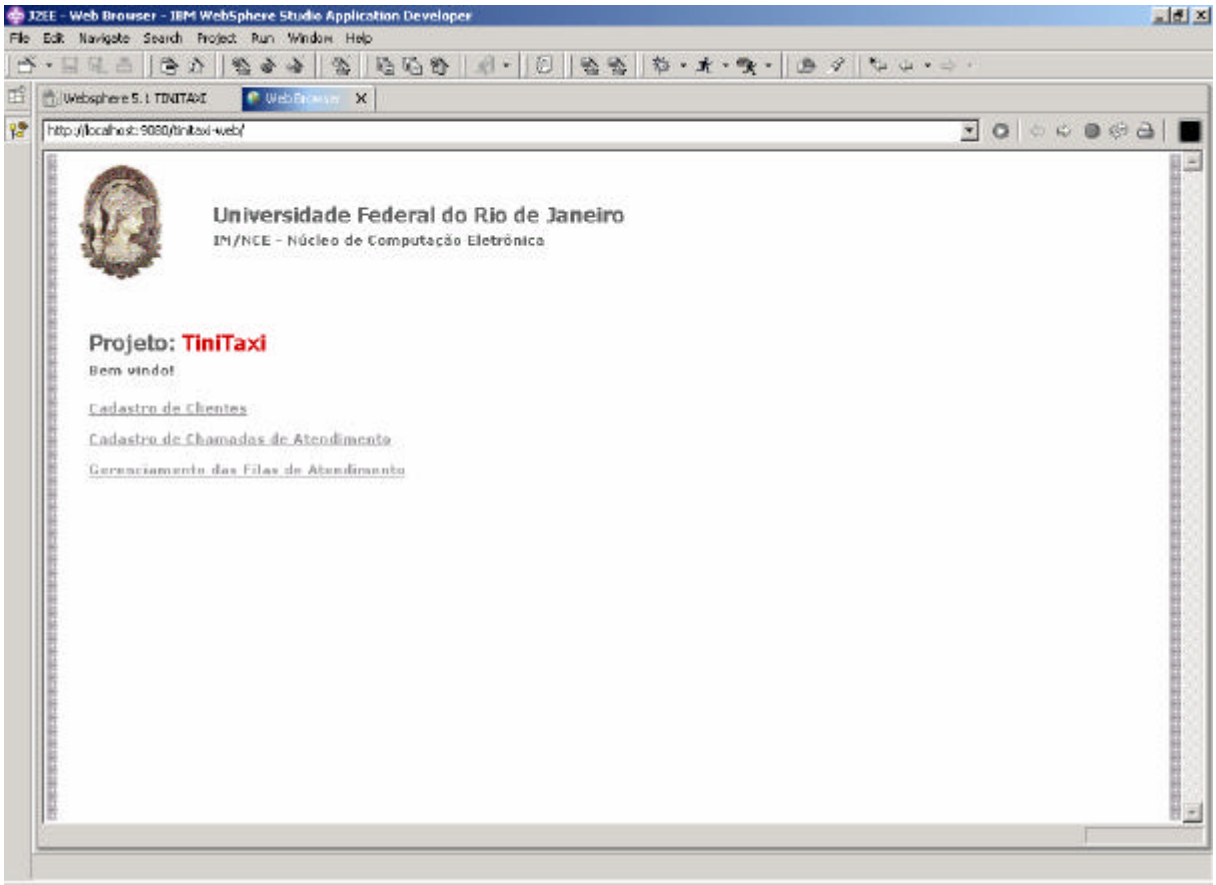
Neste ponto a aplicação WEB *(tinitaxi-web)* deve estar *online*, e pode ser testada através da seguinte URL: *http://localhost:9080/tinitaxi-web/*

As seguintes opções devem aparecer na tela principal do projeto (Web):

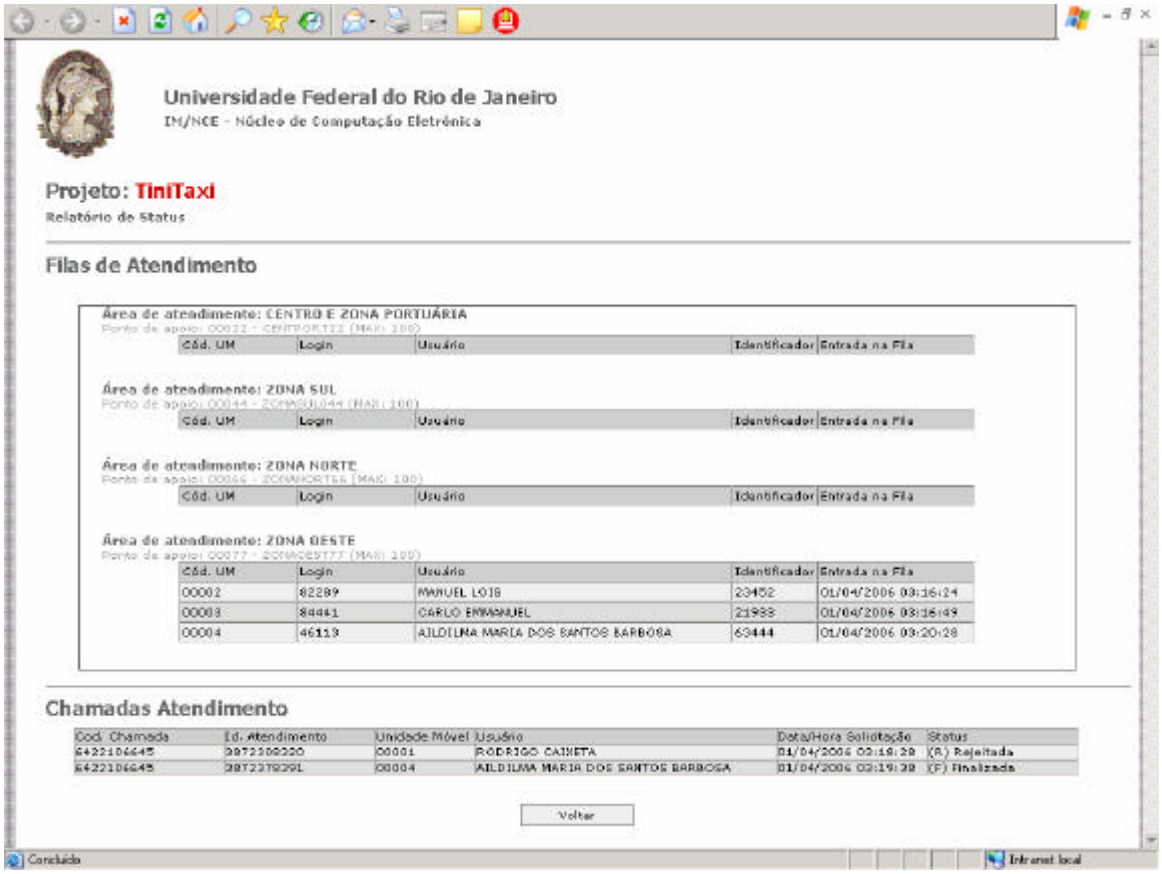
? Cadastro de Clientes

? Cadastro de Chamadas de Atendimento

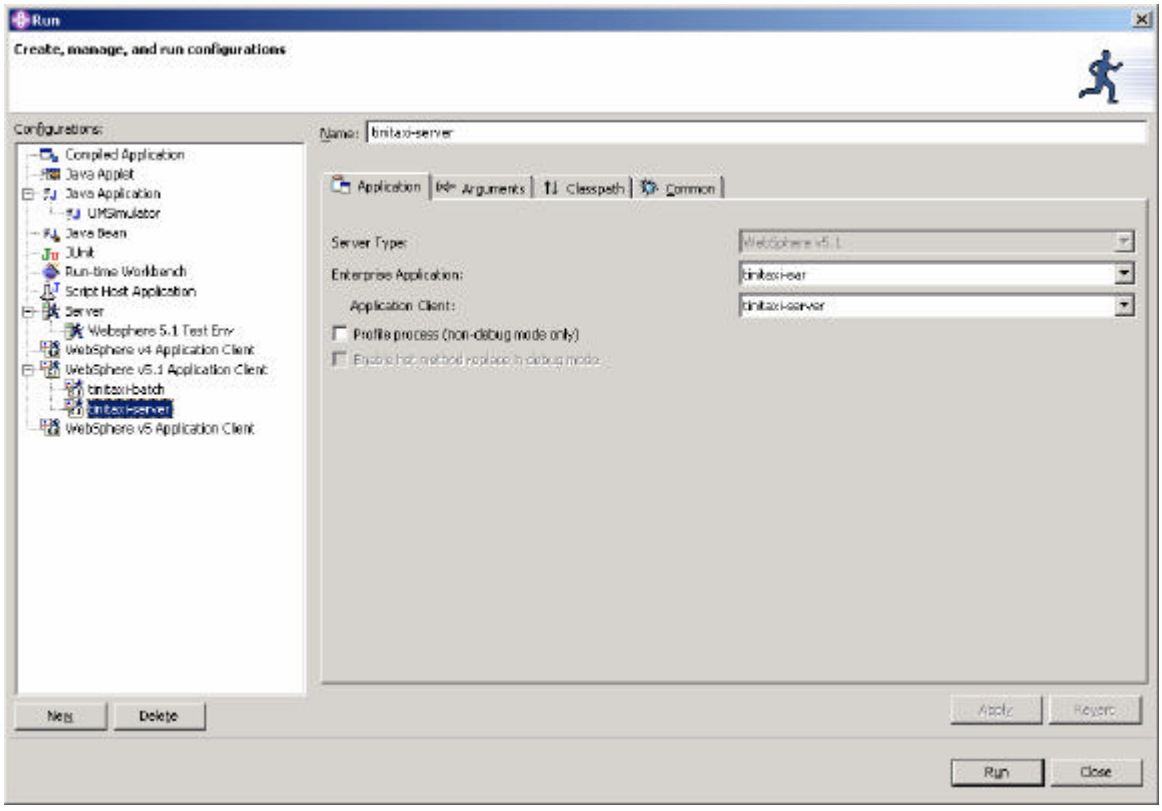
? Gerenciamento das Filas de Atendimento



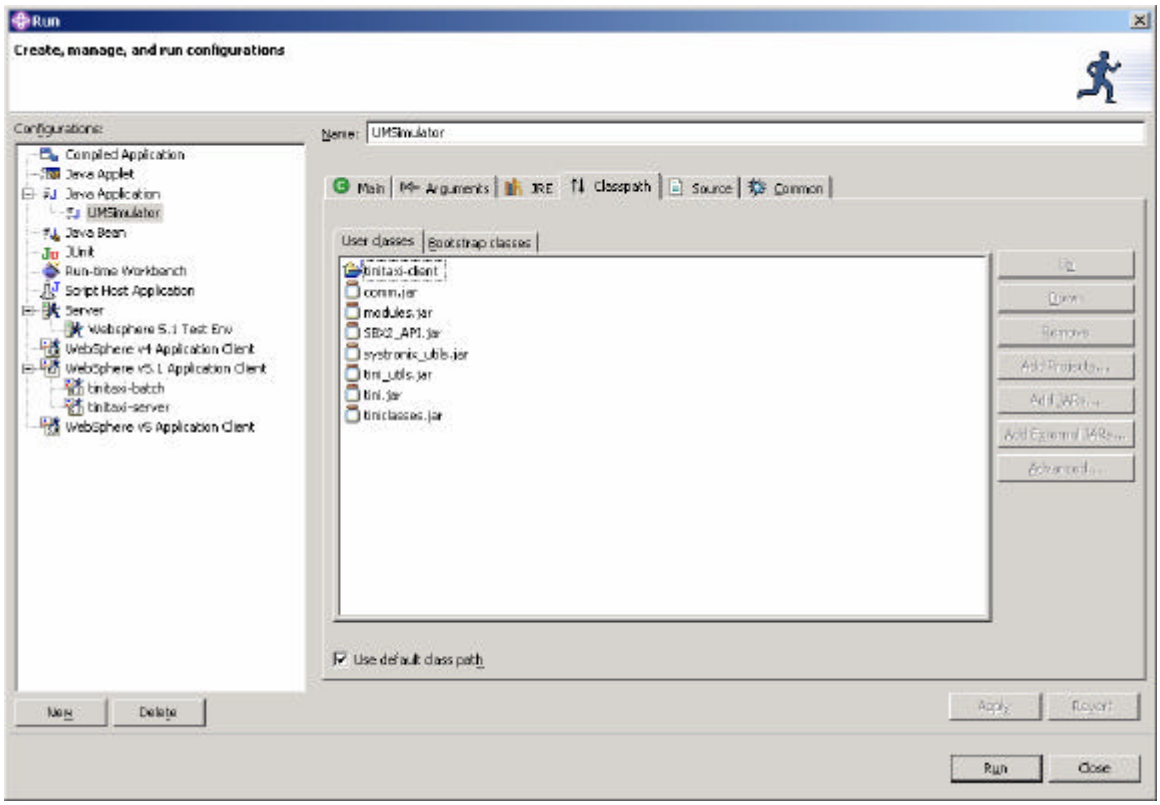
Outra URL que pode ser utilizada para o teste do servidor: *http://localhost:9080/tinitaxi-web/relatorioStatus.do*



Passo 2) Execução da classe principal *tinitaxi-server*, através do menu WSAD: *Run > tinitaxi-server*



Passo 3) Execução do projeto *tinitaxi-client*, através do menu WSAD: *Run > UMSimulator*



4.3 Conclusões do Capítulo 4

Foram encontradas dificuldades iniciais para implementação da comunicação GPRS através da placa TINI. Estas dificuldades foram causadas principalmente pela falta de documentação adequada a respeito do assunto. Poucos trabalhos foram desenvolvidos integrando a placa TINI com um aparelho celular para comunicação GPRS.

O sistema servidor não foi totalmente desenvolvido, pois existem diversas variáveis envolvidas no modelo de operação real das companhias de táxi, como por exemplo um modelo de penalizações do taxista. Contudo os testes iniciais, mesmo com um modelo simplificado de operação, mostraram-se satisfatórios, principalmente em relação à implementação do dispositivo móvel e sua comunicação sem fio com um servidor de aplicações.

Em relação ao sistema servidor, novas funcionalidades podem ser agregadas facilmente. A modelagem de dados e a implementação das regras de negócio da companhia de táxi podem ser alteradas para refletir qualquer modelo de operação. O dispositivo móvel pode ser

incrementado para conexão com outros periféricos, como por exemplo: Leitoras de cartão de credito, taxímetro, impressora entre outros.

Conforme previsto, o desenvolvimento em Java utilizando as mais modernas técnicas de desenvolvimento de sistemas facilitou o desenvolvimento da aplicação. A utilização de padrões de projetos facilitou a modelagem das classes, garantindo a compatibilidade com qualquer servidor de aplicações J2EE [CATTEL, 2001] e diminuindo o acoplamento entre os componentes do sistema.

Para que este sistema possa se tornar um sistema comercial, é necessário investigar mais a fundo os problemas existentes nos sistemas atuais, buscando acrescentar funcionalidades ao modelo proposto.

5 RESULTADOS

5.1 Introdução

Este capítulo apresenta e descreve os principais resultados obtidos neste trabalho, os quais consistem da implementação do sistema cliente, construído com base num *hardware* a ser embutido no táxi, de um sistema servidor para controle de chamadas e filas de atendimento, e de um simulador para testes de operação.

5.2 Descrição dos resultados

Segue abaixo a descrição dos resultados obtidos neste trabalho.

5.2.1 Sistema Cliente

A implementação do dispositivo móvel está centrada na comunicação entre as unidades móveis e a central utilizando um dispositivo de hardware com um sistema embutido. A proposta é que esse dispositivo estabeleça uma comunicação remota entre a central (servidor ou conjunto de servidores) e a unidade móvel (cliente) através de uma rede TCP/IP, utilizando a placa TINI como interface entre a unidade móvel e o servidor. Para comunicação remota, foi utilizado um aparelho de celular como uma unidade de comunicação com suporte.

Para a implementação deste dispositivo, foram necessários alguns dispositivos auxiliares integrados à placa TINI. Esses dispositivos possibilitaram a integração da placa com outros periféricos, tais como teclado e display. Esses periféricos proporcionaram ao usuário um canal de entrada e saída de dados através do dispositivo móvel, possibilitando uma interface homem/máquina com o sistema embutido desenvolvido. Apresar das restrições quanto ao tamanho da tela utilizada (20x4), foi possível desenvolver uma seqüência lógica de telas para a execução das funcionalidades do sistema, de acordo com o modelo de operação da companhia de táxi.

Montagem do Dispositivo Móvel

A Figura 31 ilustra o dispositivo móvel TiniTaxi utilizando os componentes acima citados.

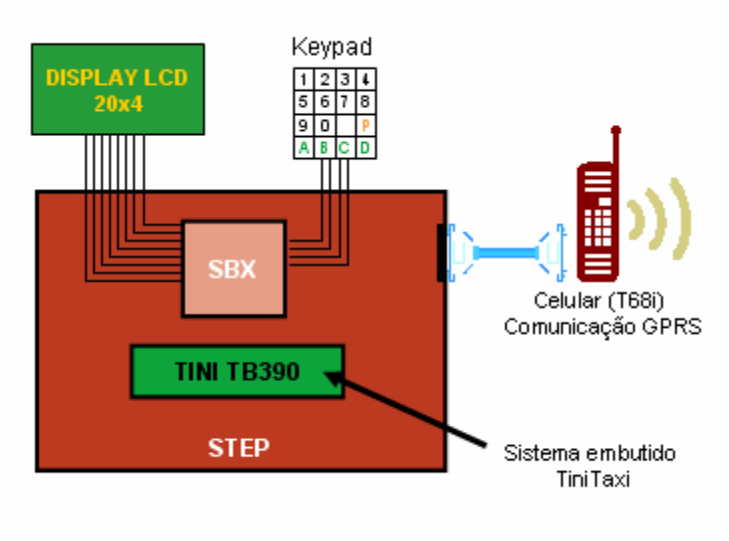


Figura 31 – Dispositivo móvel TiniTaxi montado (plataforma de *hardware* TINI).

Para montagem do dispositivo móvel foi utilizada a placa STEP como *socket board* para a placa TINI TBM390. Para interface com o teclado e o *display* foi utilizada a placa SBX conectada a placa STEP (conforme especificação fabricante). Segue abaixo um foto do dispositivo montado com todos os componentes da solução:

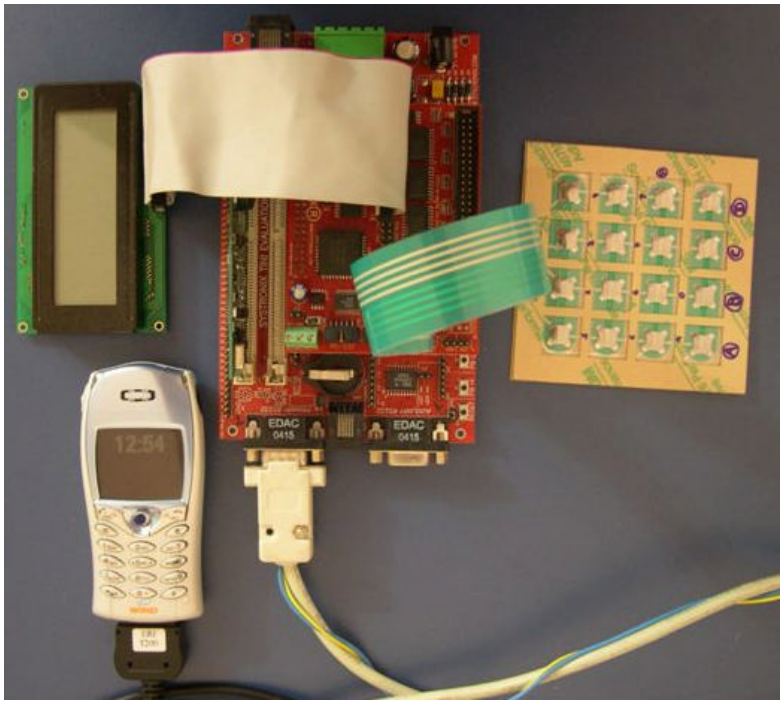


Figura 32 – Dispositivo móvel TiniTaxi montado (plataforma de *hardware* TINI).

5.2.2 Sistema Servidor

A aplicação servidora foi construída utilizando a plataforma Java 2 Enterprise Edition (J2EE [CATTEL, 2001]), sendo divida em sete subprojetos, de acordo com a divisão lógica de camadas do sistema.

A estrutura da aplicação foi dividida em projetos lógicos:

? *tinitaxi-ear* – Projeto principal para empacotamento do sistema servidor

? *tinitaxi-ejb* – Projeto com a lógica de negócios do sistema servidor

? *tinitaxi-web* – Projeto para apresentação de informações na WEB

? *tinitaxi-server* – Projeto servidor (*socket server*) para gerenciar as conexões com os dispositivos móveis

? *tinitaxi-batch* – Projeto para processamento *batch* das filas de atendimento

? *tinitaxi-framework* – Projeto com código comum (API) para todo o sistema servidor

? *tinitaxi-container* – Projeto com as configurações do servidor de aplicação

Esse sistema foi executado com sucesso em um servidor de aplicações compatível com a especificação *Java 2 Enterprise Edition*. A comunicação entre as aplicações clientes (*batch* e *socket server*) e o modulo centralizador da regra de negocio (EJB) foi estabelecida com sucesso conforme o planejado.

A figura abaixo ilustra a saída de *log* do servidor para as requisições (transações) feitas pelos clientes (taxistas):

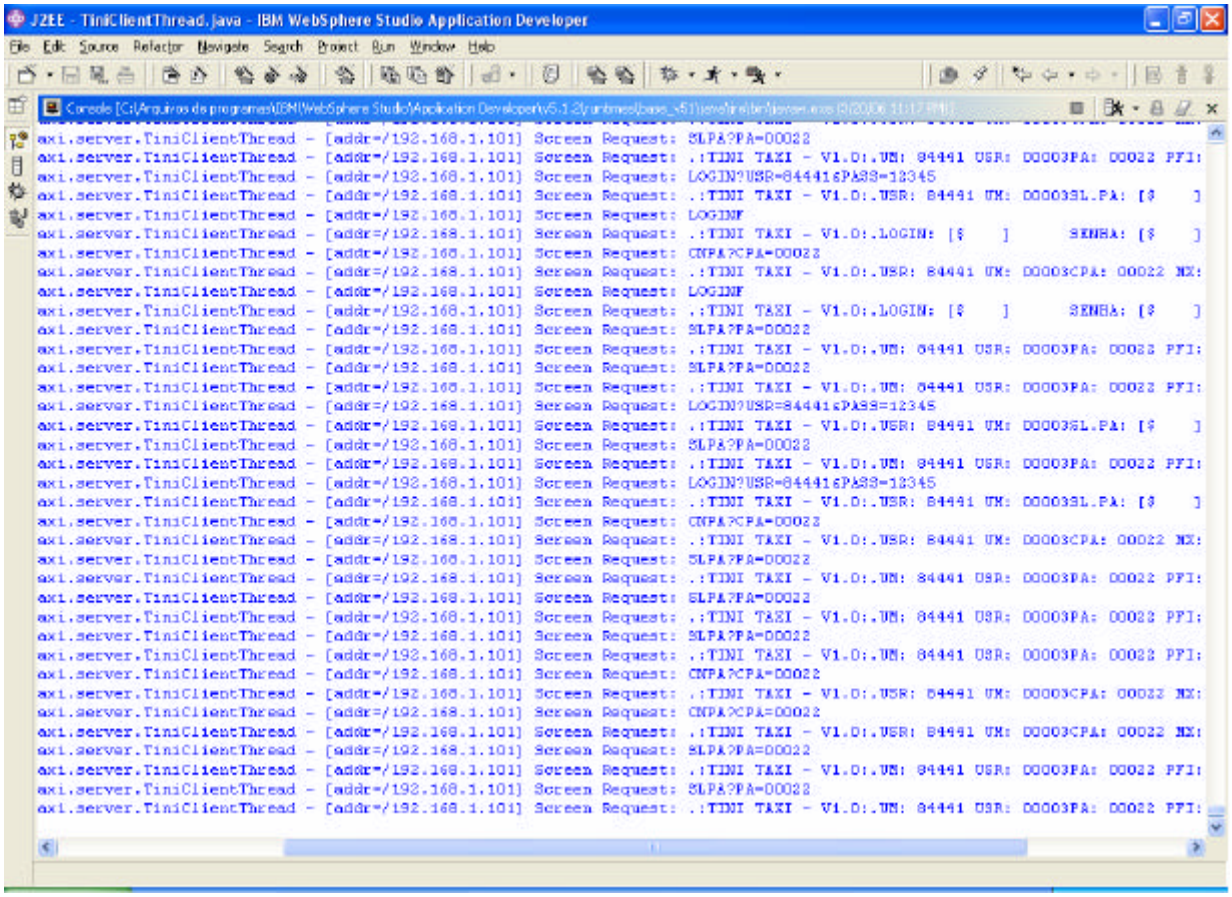


Figura 33 – Sistema servidor (log das requisições dos clientes – *client requests*).

5.2.3 Simulador *Java Swing* para testes de operação

Para viabilizar o teste da aplicação com várias unidades moveis (táxis) foi desenvolvido um simulador utilizando a plataforma *Java Swing*. Através deste componente foi possível simular a operação do sistema com vários táxis ao mesmo tempo. Segue abaixo a interface utilizada neste simulador.

Figura 34 – Aplicação Java Swing para simulação de uma unidade móvel (táxi).

A simulação foi realizada levando em conta os dados inseridos na base de dados do sistema. Os dados para simulação das chamadas de atendimento foram inseridos manualmente na base de dados (chamadas de atendimento).

Para execução dos testes, foram criadas várias *threads* executando simultaneamente a aplicação *Java Swing* (simulador), conforme ilustra a figura abaixo:

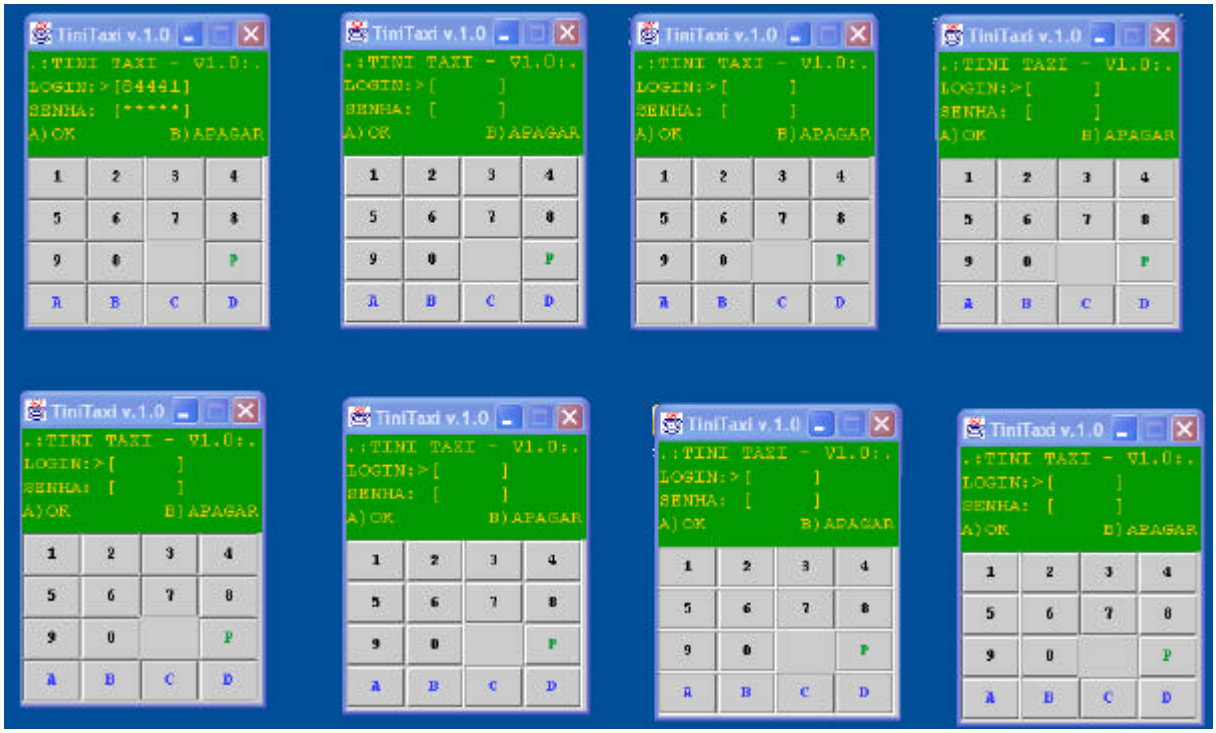


Figura 35 – Simulação da operação com vários táxis simultaneamente (aplicação *Java Swing*).

A análise da transmissão de dados entre o dispositivo móvel e o simulador apresentou os seguintes resultados:

5.2.4 Análise do custo da transmissão dos dados via GPRS

Com o objetivo de estimar o custo da transmissão dos dados via GPRS, foi analisada a quantidade de bytes transmitida na comunicação entre o cliente e o servidor durante uma sessão de atendimento (login, posicionamento no ponto de apoio, recebimento da chamada e confirmação da chamada). Em cada requisição, a quantidade de *bytes* transmitida é:

1. Tela Login - 164 bytes
2. Formulário Seleção Ponto Apoio - 144 bytes

3. Resultado Seleção Ponto Apoio - 130 bytes
4. Formulário Consulta Ponto Apoio - 143 bytes
5. Resultado Consulta Ponto Apoio - 130 bytes
6. Tela Recebimento de Chamada - 129 bytes
7. Tela Confirmação de chamada - 130 bytes

Média de quantidade de bytes transmitidos: 139 bytes

Foi também analisado um cenário usando compactação de dados através de uma API fornecida pela plataforma J2SE (classe Java *GZIPInputStream*). Obs: O algoritmo utilizado para compactação é o mesmo utilizado no formato GZIP

Foram encontrados os seguintes resultados na transmissão dos dados utilizando compactação GZIP:

1. Tela Login
    a. Quantidade de bytes enviados (com compactação): 135 bytes (Taxa de compactação: 17.68%)
2. Formulário para Seleção Ponto Apoio
    a. Quantidade de bytes enviados (com compactação): 127 bytes (Taxa de compactação: 11.81%)
3. Resultado Seleção Ponto Apoio
    a. Quantidade de bytes enviados (com compactação): 110 bytes (Taxa de compactação: 15.38%)
4. Formulário Consulta Ponto Apoio
    a. Quantidade de bytes enviados (com compactação): 126 bytes (Taxa de compactação: 11.89%)
5. Resultado Consulta Ponto Apoio
    a. Quantidade de bytes enviados (com compactação): 113 bytes (Taxa de compactação: 13.08%)
6. Tela Recebimento de Chamada
    a. Quantidade de bytes enviados (com compactação): 89 bytes (Taxa de compactação: 31.01%)
7. Tela Confirmação de chamada

a. Quantidade de bytes enviados (com compactação): 100 bytes (Taxa de compactação: 23.08%)

Resultados:

Média de quantidade de bytes enviados (sem compactação): 139 bytes

Média de quantidade de bytes enviados (com compactação): 114 bytes

Média de taxa de compactação: 17.7%

Apesar da avaliação do cenário utilizando compactação de dados, o algoritmo não foi implementado na comunicação entre o cliente e o servidor, ficando assim uma sugestão para trabalhos futuros.

Estimativa de bytes transferidos em 1 (um) mês:

1. Estimativa do número de transações por dia: 10 horas de trabalho x 5 transações por hora = 50 transações por dia

2. Estimativa de bytes transmitidos por dia (usando compactação): 50 x 114 = 5.700

3. Estimativa de bytes transmitidos por mês: 26 x 5.700 = 148.200 bytes por mês (0.1413 MB)

4. Tabela de tarifas GPRS aplicada pela TIM (plano **TIM CONNECT FAST**)

MB – Mínimo	**MB - Máximo**	**R$ / MB**
0	1	5,99

5. Custo mensal para o taxista (serviços): R$ 5,99 + custo fixo

Para comparar o custo de operação utilizando GPRS com o modelo atual (baseado em rádio-operador), foi realizada uma pesquisa de campo envolvendo taxistas de diversas cooperativas para levantamento das despesas mensais com a administração do serviço de rádio. Normalmente a mensalidade do taxista não inclui somente as despesas relacionadas à comunicação via rádio, mas incorpora vários outros gastos referentes a salários de funcionários, cobertura de seguro, impostos, rateios, etc.

Neste levantamento foi identificado que o valor pago pelos taxistas fica entre 200 e 400 reais por mês. De acordo com informações coletadas diretamente com os taxistas, estima-se que 30% das despesas são relativas ao modelo de operação baseado em transmissão via radio. Conclui-se que o valor mensal gasto com o modelo de operações baseado em rádio é de 60 (sessenta) reais no melhor caso e de 120 (cento e vinte) reais no pior caso.

5.2.5 Teste de Carga (Teste de *Stress*)

O principal objetivo de se efetuar um teste de carga em um sistema é assegurar que a arquitetura desenvolvida realmente consegue responder a uma quantidade prevista de usuários que irão acessar o aplicativo. Este tipo de teste não elimina a possibilidade do sistema conter algum tipo de erro.

Existem algumas ferramentas que auxiliam na criação de *scripts* automáticos para execução de testes de *stress*, como por exemplo a ferramenta *JMeter*. O *JMeter* é um *software* de código aberto mantido pelo grupo *Apache Jakarta*, e tem como objetivo executar planos de teste de forma automática, simulando várias requisições simultâneas no servidor. Seguem abaixo algumas características que tornaram o *JMeter* uma ferramenta gratuita de alto valor agregado:

- Permite a execução de testes através de amostras simulando HTTP, FTP, protocolo simples de acesso a objetos (SOA), conexão com banco de dados (JDBC), protocolo de acesso a diretórios (LDAP), entre outros;
- Portabilidade de plataforma (escrito em linguagem Java);
- Interface gráfica elaborada (utiliza API *Java Swing*);
- Ferramenta *multithreading* de teste permitindo que uma só máquina simule muitas requisições de forma simultânea;
- Log de resultados para análise off-line;

5.2.5.1 Configuração do plano de testes

Para configuração do plano de testes utilizando o *JMeter* foram realizados os seguintes passos:

1. Configuração do servidor: *start* no servidor de aplicações (*WebSphere App Server*) e no *Client App tinitaxi-server* (*Socket Server*)

2. Configuração das requisições: configuração do número de *threads* que devem ser disparadas para acesso ao servidor (*Number of Threads*), intervalo entre as requisições (*Ramp-Up Period*) e a quantidade de vezes que as *threads* irão ser executadas (*Loop Count*). As configurações para o caso de teste criado podem ser visualizadas na Figura 36.

3. Configuração das amostras (Samplers): Tipo de amostra para requisições. No caso deste plano de testes foi utilizado somente *Samplers* do tipo TCP, conforme Figura 36.

4. Execução: Configuração do tempo de execução de todas as *threads* de forma simultânea

O sistema servidor foi implantado em uma máquina com a seguinte configuração: Pentium® IV 2.4 GHz, 1.0 GB de memória RAM. Nesta mesma máquina também foram instalados o servidor de banco de dados (DB2) e o servidor de aplicações (*WebSphere Application Server*).

O teste foi configurado para efetuar várias transações simultâneas no servidor. Cada transação representa um conjunto de requisições necessárias para o posicionamento do taxista em um determinado ponto de apoio.

Seqüência das requisições: Login no sistema (*LOGIN Action*), Consulta de Ponto de Apoio (*CONPA Action*) e seleção de ponto de apoio (*SELPA Action*).

Segue abaixo uma das telas da configuração feita no *JMeter* para a execução do plano de teste:

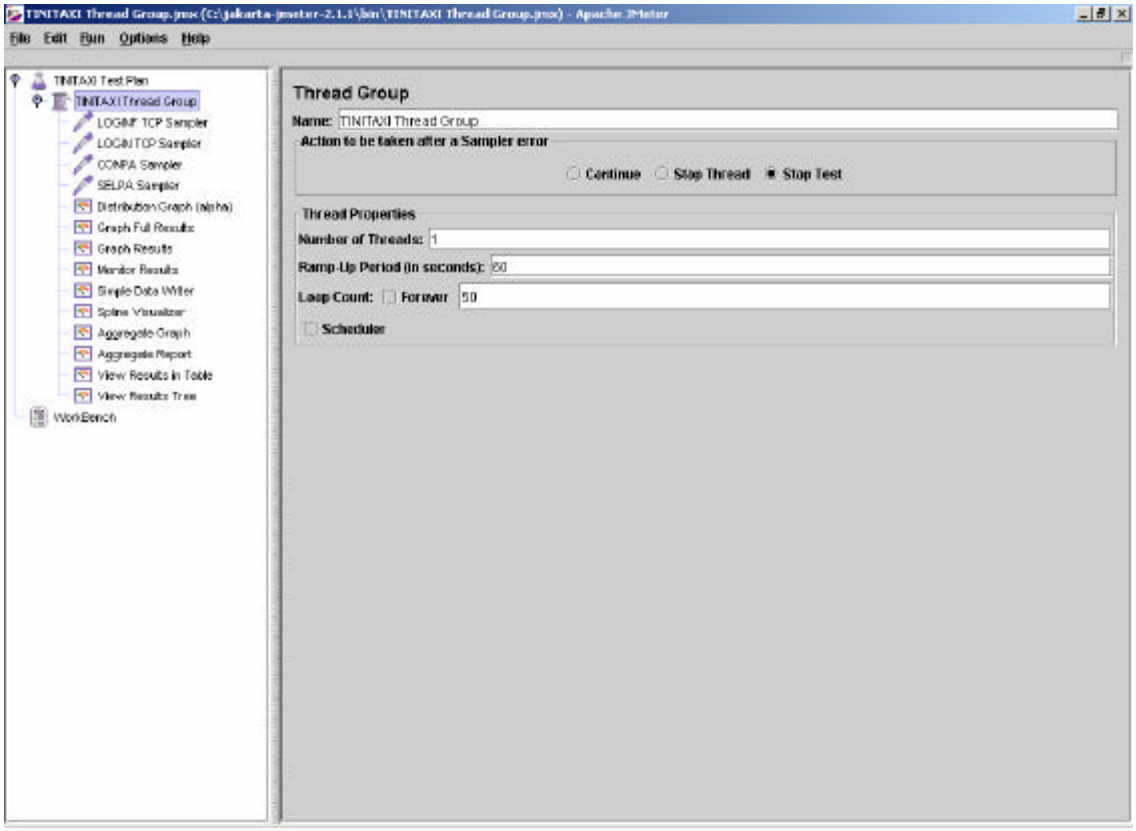


Figura 36 – Tela de configuração da execução (configuração das *threads*) – *Jmeter.*

5.2.5.2 ***Resultados do teste de* stress**

Os resultados do teste de *stress* foram gerados a partir da configuração do plano de teste, conforme descrito nas seções anteriores. Foram criados vários cenários para execução dos testes, aumentando gradativamente o número de usuários simultâneos e verificando o comportamento do servidor (tempo de resposta).

Os seguintes cenários foram criados para execução do teste de *stress*:

- ? 1 usuário, efetuando 50 transações com intervalos de 5 segundos entre as requisições;
- ? 10 usuários simultâneos, efetuando 10 transações cada com intervalos de 5 segundos entre as requisições;
- ? 30 usuários simultâneos, efetuando 10 transações cada com intervalos de 5 segundos entre as requisições;
- ? 50 usuários simultâneos, efetuando 10 transações cada com intervalos de 5 segundos entre as requisições;

Seguem abaixo os gráficos gerados para cada cenário de teste:

5.2.6 Cenário para 1 (um) usuário

Segue o gráfico dos tempos de resposta para o cenário:

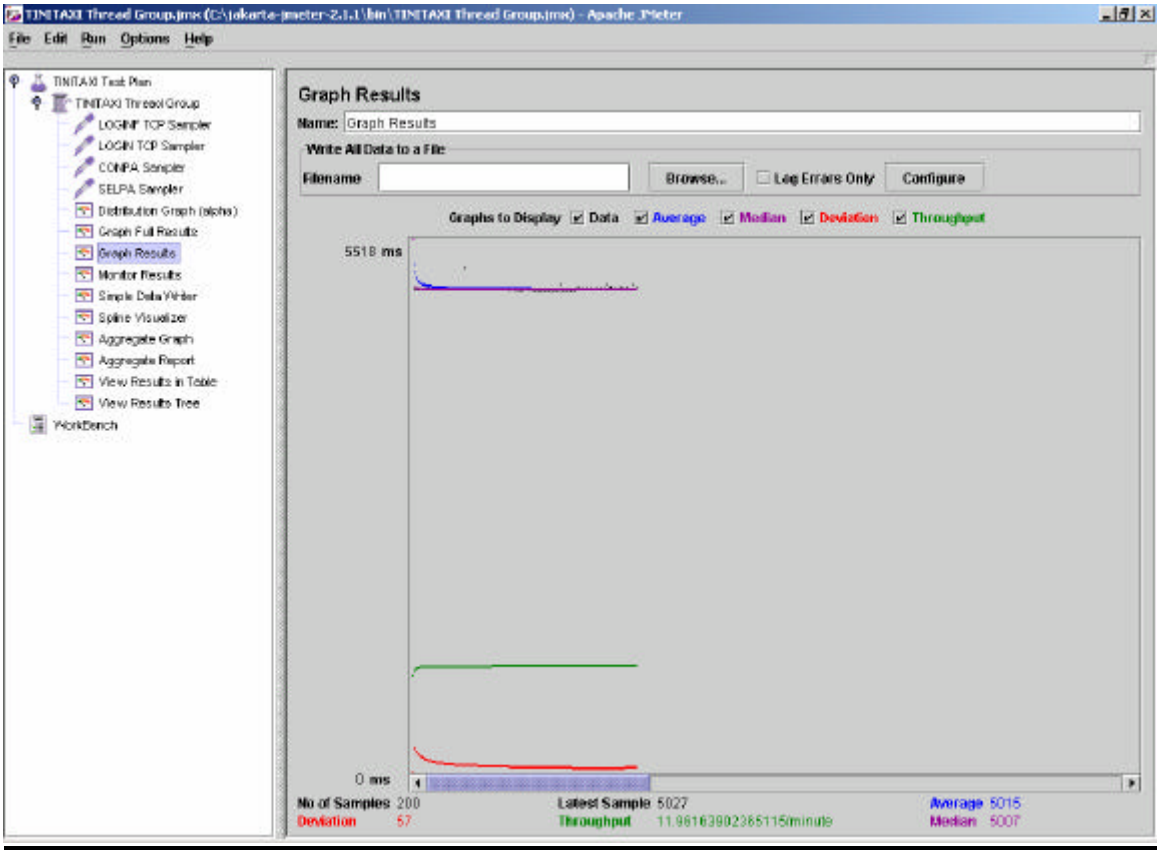


Figura 37 – Gráfico de execução para o cenário com 1 (um) usuário.

? Número de *threads* simultâneas: 1 *thread*;

? Intervalo entre as requisições: 5 segundos;

? Tempo total de testes (tempo de execução): 15 minutos;

? Número de transações executadas no servidor: 50 transações;

? Número de requisições executadas no servidor: 200 requisições.

Resultados:

? Vazão aferida (*throughput*): 12 requisições por minuto;

? Tempo de resposta (médio): 0.15 segundos;

? Tempo de resposta no pior caso: 0.59 segundos;

? Tempo de resposta no melhor caso: 0.07 segundos.

Interpretação do gráfico:

Este gráfico ilustra o tempo de *request/response* entre um número estimado de clientes (virtuais) e o servidor de negócio do projeto TINITAXI. Para este teste, foi utilizado um componente da *framework JMeter* chamado *Graph Listener*, que coleta os tempos de *request/response* entre os clientes e o servidor. Cada *thread* representa um cliente virtual que dispara uma seqüência de comandos ao servidor, simulando um atendimento de chamada de acordo com o modelo de negócio do sistema. As curvas geradas foram: Dados coletados - *data* (preto), média calculada - *media*n (azul), desvio padrão – *deviation* (vermelho) e vazão aferida - *throughput* (verde), todas em milisegundos. No caso do teste com 1 (um) usuário simultâneo (uma *thread* simultânea) a vazão aferida (*throughput*) do servidor se manteve constante, demonstrando a estabilidade do servidor para a carga submetida.

5.2.7 Cenário para 10 (dez) usuários simultâneos

Segue o gráfico dos tempos de resposta para o cenário:

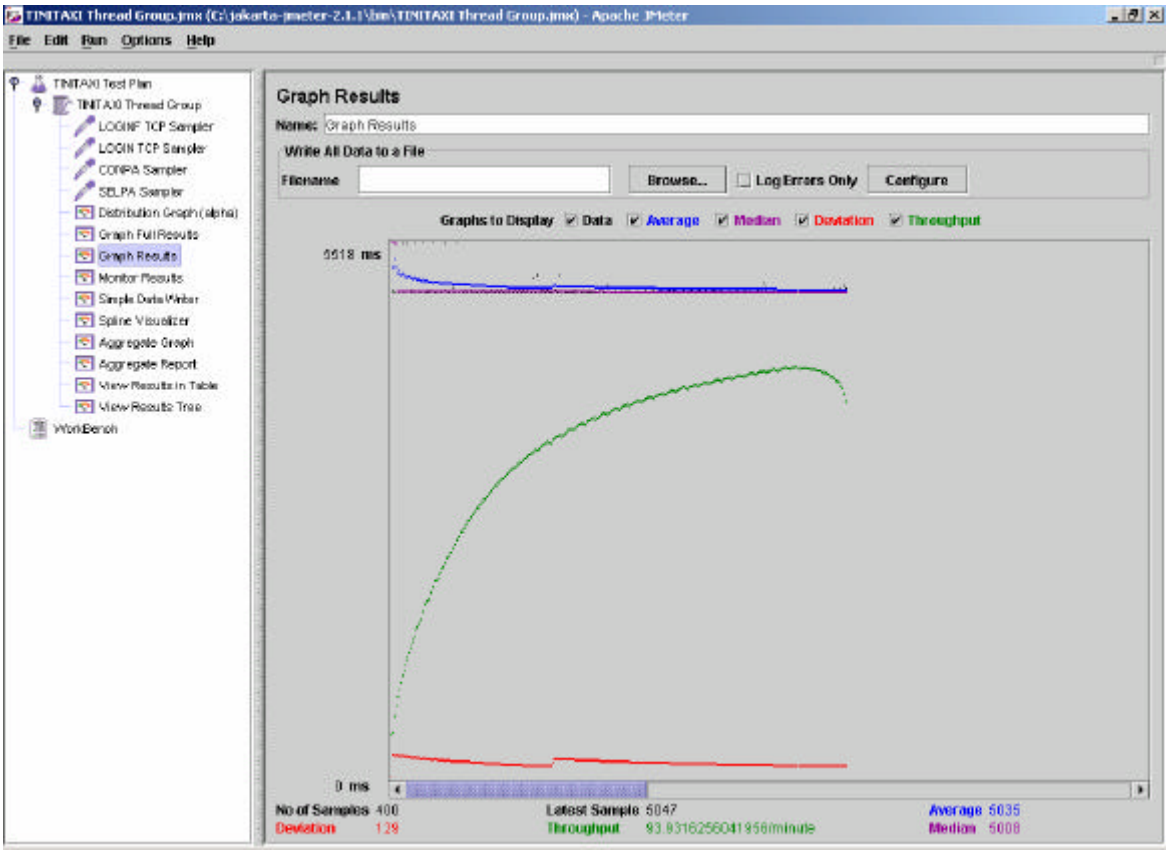


Figura 38 – Gráfico de execução para o cenário com 10 (dez) usuários simultâneos.

- Número de *threads* simultâneas: 10 *threads*;
- Intervalo entre as requisições: 5 segundos;
- Tempo total de testes (tempo de execução): 15 minutos;
- Número de transações executadas no servidor: 100 transações;
- Número de requisições executadas no servidor: 400 requisições.

Resultados:

- Vazão aferida (*throughput*): 96 requisições por minuto;
- Tempo de resposta (médio): 0.35 segundos;
- Tempo de resposta no pior caso: 1.58 segundos;
- Tempo de resposta no melhor caso: 0.07 segundos.

Interpretação do gráfico:

Para este teste cada *thread* representa um cliente virtual que dispara uma seqüência de comandos ao servidor, simulando um atendimento de chamada de acordo com o modelo de negócio do sistema. As curvas geradas foram: Dados coletados - *data* (preto), média calculada - *media*n (azul), desvio padrão – *deviation* (vermelho) e vazão aferida - *throughput* (verde), todas em milisegundos.

No caso do teste com 10 (dez) usuários simultâneos (*threads* simultâneas) a vazão aferida (*throughput*) do servidor aumentou proporcionalmente com a entrada dos usuários (de um a dez), mas o servidor se manteve estável conforme o esperado. Uma observação importante neste teste é a entrada gradativa dos usuários no servidor (As *threads* são disparadas uma a uma até o número máximo estabelecido no teste, neste caso 10 (dez) *threads*).

5.2.8 Cenário para 30 (trinta) usuários simultâneos

Segue o gráfico dos tempos de resposta para o cenário:

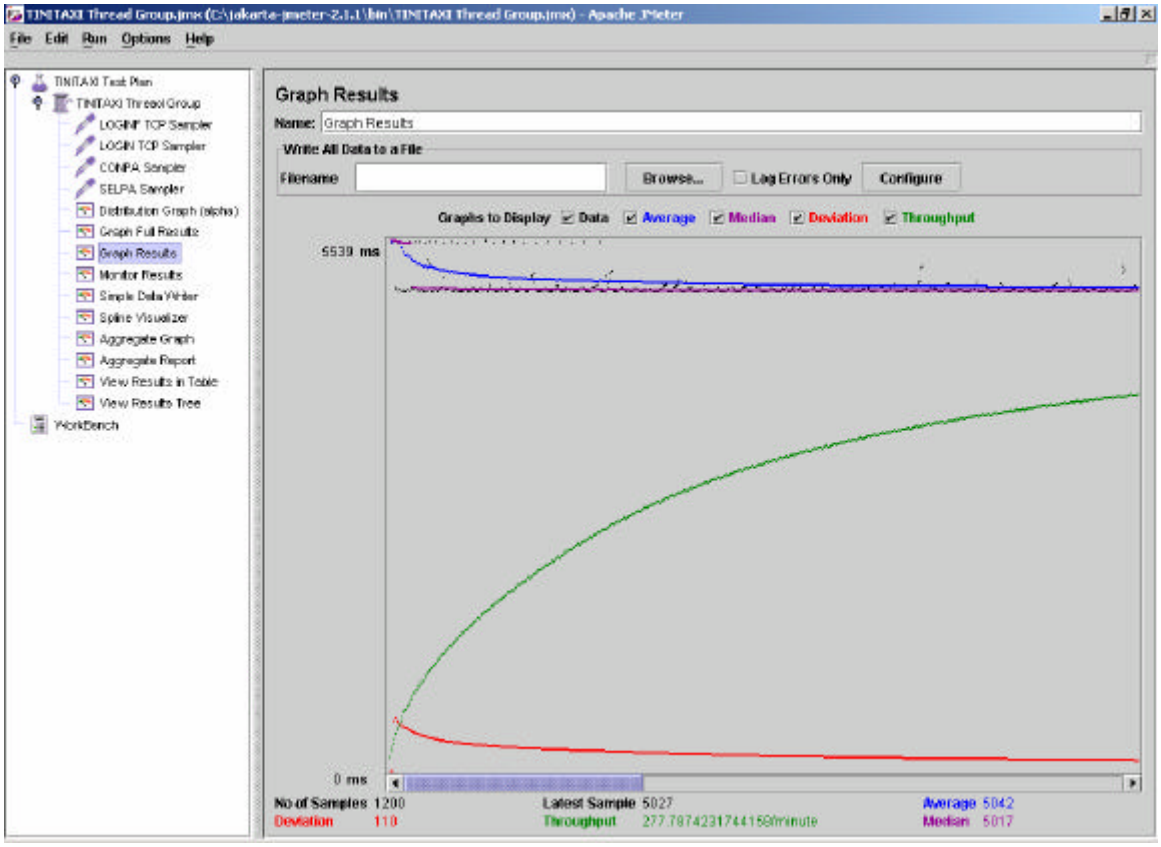


Figura 39 – Gráfico de execução para o cenário com 30 (trinta) usuários simultâneos.

? Número de *threads* simultâneas: 30 *threads*;

? Intervalo entre as requisições: 5 segundos;

? Tempo total de testes (tempo de execução): 15 minutos;

? Número de transações executadas no servidor: 300 transações;

? Número de requisições executadas no servidor: 1200 requisições.

Resultados:

? Vazão aferida (*throughput*): 276 requisições por minuto;

? Tempo de resposta (médio): 0.42 segundos;

? Tempo de resposta no pior caso: 1. 87 segundos;

? Tempo de resposta no melhor caso: 0.07 segundos.

Interpretação do gráfico:

No caso do teste com 30 (trinta) usuários simultâneos (*threads* simultâneas) a vazão aferida (*throughput*) do servidor aumentou proporcionalmente com a entrada dos usuários (de um a trinta), mas o servidor se manteve estável conforme o esperado. Uma observação importante neste teste é a entrada gradativa dos usuários no servidor (As *threads* são disparadas uma a uma até o número máximo estabelecido no teste, neste caso 30 (trinta) *threads*).

5.2.9 Cenário para 50 (cinqüenta) usuários simultâneos

Segue o gráfico dos tempos de resposta para o cenário:

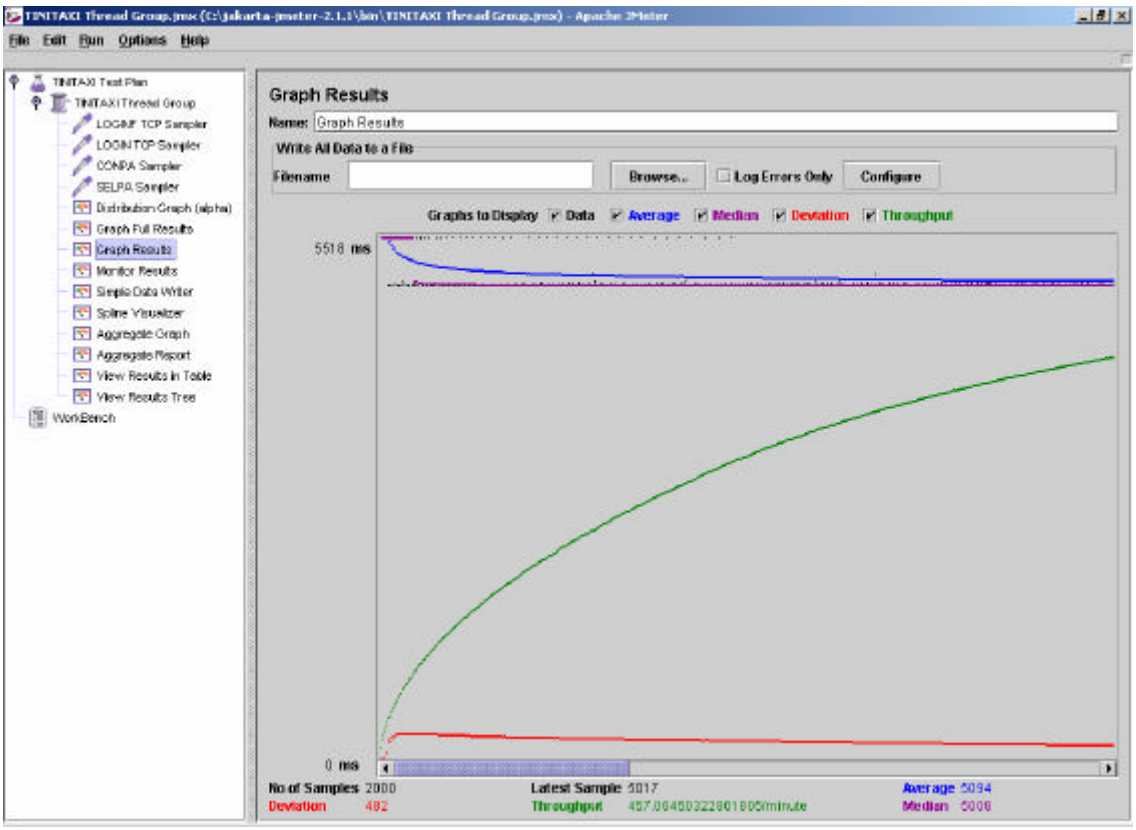


Figura 40 – Gráfico de execução para o cenário com 50 (cinqüenta) usuários simultâneos.

- Número de *threads* simultâneas: 50 *threads*;
- Intervalo entre as requisições: 5 segundos;
- Tempo total de testes (tempo de execução): 15 minutos;
- Número de transações executadas no servidor: 500 transações;
- Número de requisições executadas no servidor: 2000 requisições;

Resultados:

- Vazão aferida (*throughput*): 456 requisições por minuto;
- Tempo de resposta (médio): 0.94 segundos;
- Tempo de resposta no pior caso: 5.40 segundos;
- Tempo de resposta no melhor caso: 0.07 segundos;

Interpretação do gráfico:

No caso do teste com 50 (cinqüenta) usuários simultâneos (*threads* simultâneas) a vazão aferida (*throughput*) do servidor aumentou proporcionalmente com a entrada dos usuários (de um a trinta), mas o servidor se manteve estável conforme o esperado.

5.3 Conclusões sobre os resultados obtidos

O resultado principal deste trabalho é a concepção e implementação do dispositivo móvel (a ser implantado no táxi) e do sistema servidor (controle das filas). Levando em conta todos os resultados encontrados neste trabalho, podemos concluir que:

1. O custo da utilização de um dispositivo móvel utilizando um telefone com GPRS é baixo, em comparação com o custo atual que um taxista tem com o uso de um rádio;

2. O *hardware* utilizado se mostrou adequado para execução das operações diárias de um taxista (modelo de operação). Apesar do *display* ser limitado a apenas 80 (oitenta) caracteres, foi possível implementar as funcionalidades mais importantes para realização de uma transação de atendimento de chamada, de acordo com o modelo de negócio escolhido na implementação deste trabalho;

3. O sistema servidor se comportou da forma esperada para até 30 (trinta) requisições simultâneas, mantendo um desempenho de menos de 1 (um) segundo de tempo de resposta. Considerou-se que trinta requisições simultâneas representam o universo de uma empresa com 300 (trezentos) veículos.

4. Ainda é necessário realizar testes de campo com o sistema para determinar sua operação correta e em condições de difícil comunicação e também sua viabilidade com respeito à área de cobertura das antenas da operadora. Dado

que a operadora de telefonia móvel oferece telefones móveis a uma grande quantidade de assinantes, assume-se que os táxis também terão uma cobertura adequada.

6 CONCLUSÕES

6.1 Introdução

Esta dissertação descreveu a análise e a implementação do protótipo de um sistema para dar suporte a serviços de táxis, baseados na localização geográfica e no gerenciamento de filas de pedidos e de atendimento. O principal objetivo foi automatizar o modelo de operação existente em empresas de táxi que utilizam comunicação via rádio como forma de prover suporte ao gerenciamento das filas de atendimento.

A solução apresentada é inovadora pois difere das soluções existentes, que são baseadas em serviços de rádio-operador ou utilizam dispositivos de posicionamento com um GPS.

6.2 Principais dificuldades

Nesta seção são apresentadas as principais dificuldades que enfrentadas durante o desenvolvimento deste trabalho para compartilhar a experiência obtida durante este processo.

Foi realizado um estudo de cada equipamento utilizado para a construção do dispositivo móvel, principalmente nas placas de extensão para conexão dos periféricos. Alguns problemas foram encontrados na conexão da placa TINI com o *display* através da placa de extensão SBX2, devido a vários padrões de *display* diferentes. Outros problemas foram encontrados na utilização do celular como ponte GPRS para conexão com o servidor, principalmente pela falta de documentação disponível e pela dificuldade de aquisição do adaptador necessário para conexão. Outro problema enfrentado foi a apresentação das informações no sistema embutido em um *display* com apenas 20 (vinte) caracteres por linha, com quatro linhas, limitando a apresentação dos dados para o usuário (taxista).

6.3 Conclusões Finais

Esta seção apresenta as principais conclusões desta dissertação, em relação aos objetivos estabelecidos no capítulo 1, as quais são:

1. O principal objetivo estabelecido no início deste trabalho foi atingido, ou seja, projetar um sistema capaz de gerenciar táxis, controlando prioridades de filas de atendimento através de um dispositivo baseado em sistemas embutidos com comunicação sem fio;

2. O emprego de Java para desenvolver sistemas embutidos diminui consideravelmente o tempo de desenvolvimento, facilitando a implementação através da utilização de APIs pré-definidas na própria linguagem, como por exemplo acesso a redes *Ethernet*, comunicação serial, entre outros;

3. O projeto de sistemas embutidos completos, que incluam *hardware* e *software* com suporte à linguagem Java são pouco divulgados e possuem documentação restrita. Isto aumenta a dificuldade do desenvolvimento de aplicações de sistemas com essas características;

4. O custo da utilização de um dispositivo móvel utilizando um telefone com GPRS é baixo em comparação com o custo atual que um taxista tem com o uso de um rádio;

5. Ainda é necessário realizar testes de campo com o sistema para determinar sua operação correta e condições de difícil comunicação e também sua viabilidade com respeito à área de cobertura das antenas da operadora.

6.4 Trabalhos futuros

Para trabalhos futuros, sugere-se que os requisitos que ainda não foram atendidos sejam implementados. Além disso, funções serão adicionadas permitindo assim uma evolução da solução proposta.

Entre novas funcionalidades, estudam-se as seguintes implementações e usos:

1. Otimizar o protocolo de comunicação entre o dispositivo móvel e a central de controle, a fim de diminuir o numero de *bytes* trafegados e assim minimizar os custos da solução, pois os serviços de comunicação via GPRS normalmente são cobrados por número de *bytes* transmitidos;

2. Adicionar um controle de segurança nos serviços de negócio disponibilizados pela central (servidor);

3. Complementar a lógica de negócios do sistema servidor para contemplar o modelo de “penalização” para os taxistas que não atenderem às chamadas no tempo adequado (dependendo das regras de negócio da companhia);

4. Implementar algoritmos para compactação de dados com o objetivo de diminuir a quantidade de bytes enviados, e conseqüentemente diminuir o custo da transmissão GPRS.

7 REFERÊNCIAS

[ANDERSSON, C., 2001] ANDERSON, C., ***GPRS and 3G Wireless Applications*** – John Wiley & Sons, Inc, 2001

[AXELSON, 2003] AXELSON JAN, ***Embedded Ethernet and Internet Complete***, Lakeview Research, 2003

[BATES, R.J., 2001] BATES, R.J., ***GPRS General Packet Radio Service*** – McGraw Hill, 2001.

[BROWN, 2001] BROWN, KYLE, ***Enterprise Java Programming with IBM® WebSphere®***, Addison Wesley, 2001.

[CATTEL, 2001] CATTEL, RICH AND JNSCORE JIM, ***J2EE Technology in practice: Building Business Applications with the Java 2 Platform, Enterprise Edition***, Sun Microsystem Inc., 2001.

[DAVID, 2001] GEARY, DAVID M., ***Java Server Pages Specification***, Sun Microsystems Inc., 2001.

[DECT, 2005] ***DECT Forum, Digital Enhanced Cordless Telecommunications,*** disponível em: http://www.dect.ch, consultado em novembro de 2005.

[DS80C, 2005] MAXIM, *Dallas Semiconductor*, ***Dial-Up Networking with the DS80C400 Microcontroller***, disponível em: http://pdfserv.maxim-ic.com, consultado em: novembro de 2005.

[EMBD, 2005] ***Embedded Devices***, disponível em: http://www.embedded.com, consultado em: novembro de 2005.

[ETSI, 2005] ***ETSI – European Telecommunications Standards Institute,*** disponível em http://www.etsi.org, consultado em: novembro 2005.

[FLOYD, 2001] MARINESAU, FLOYD, ***Ejb ® Design Patterns***, Middleware Company, 2002.

[GHRIBI, B. et al, 2000] GHRIBI, B. e LOGRIPPO, L., ***Understanding GPRS: The GSM Packet Radio Service,*** Computer Networks, vol. 34, pp.763-779,2000.

[HAGGAR, 2000] HAGGAR, PETER, 1965, ***Practical Java Programming Language Guide***, Addison Wesley, 2000.

[HOFFMAN, J., 2005] HOFFMAN, J., ***Wireless Evolution Insider – The Wireless Intelligence from the GSM Association,*** disponível em: http://www.gsmworld.com, consultado em: novembro de 2005.

[ITJVTINI, 2006] ***Ita Java Tini, UFLA 2001*** disponível em: http://www.comp.ufla.br/monografias/ano2001/ mavel.pdf, consultado em: fevereiro de 2006.

[JAIME, 2000] JAWORSKI, JAIME, ***Java 2 Certification Guide***, New Riders, 1999-2000.

[JAMES, 2001] JAMES, HART, ANDREI CIOROIANU, ***Java XML Professional***, Wrox Press LTD, 2001

[LOOMIS, 2001] DON, LOOMIS, ***The Tini Specification***, Dallas Semiconductors Corporation, 2001

[MANSON-HALFEL, 2001, 200, 1999] MANSON-HALFEL, RICHARD, ***Enterprise Java Beans***, O'Reilly, 2001, 200, 1999

[MTC400, 2006] ***M2M Solutions,*** *MTC400 Auto System****,*** disponível em: http://www.transurbana.com/novas_tecnologias.htm, consultado em: fevereiro de 2006.

[REDL, S.M. et al, 1998] REDL, S. M., ***GSM Personal Communications Handbook – Mobile Communications Series,*** Artech House Inc, 1998.

[ROMAN, 1999] ROMAN, ED, ***Enterprise Java Beans***, John Wiley & Sons, Inc., 1999

[SESHADRI, 1999] SESHADRI, GOVIND, ***Enterprise Java Computing***, Cambrige University Press, 1999

[SHANNOM et al, 2000] SHANNOM, BILL., ***Java 2 Platform, Enterprise Edition: Platform and component specifications***, Sun Microsystems Inc., 1999-2000.

[TINI, 2005] MAXIM, Dallas Semiconductor, ***TINI -Tiny InterNet Interfaces***, disponível em: http://www.maxim-ic.com/TINIplatform.cfm, consultado em: novembro de 2005.

[TINIGPS, 2005] MAXIM, Dallas Semiconductor, ***TINI – GPS Project***, disponível em: http://web.hazmat.com/~mjb/projects/tinigps/ consultado em: novembro de 2005.

[TOPAUTO, 2006] ***TopAuto System,*** disponível em: http://www.abovenet.com.br/topauto.asp, consultado em: fevereiro de 2006.

[UMTS FORUM, 2005] ***UMTS Forum***, disponível em: http://www.umts-forum.org/, consultado em: novembro de 2005.

[WIFI, 2005] ***Wi-Fi Alliance,*** *Open Section* disponível em: http://www.weca.net/OpenSection/index.asp, consultado em: novembro de 2005.

[WIRE, 2006] ***Dallas Maxim,*** *1-Wire and iButton*, disponível em: http://www.maxim-ic.com/1-Wire.cfm

APÊNDICES

Apêndice A: Detalhamento dos sinais dos pinos de um conector DB-9

As portas seriais seguem as configurações de “Equipamento de Comunicação de Dados (DCE)” e “Equipamento Terminal de Dados (DTE)”, que determinam características do canal de comunicação. A principal diferença encontra-se na pinagem do conector.

Pela tabela abaixo é possível verificar os sinais dos pinos de um conector DB-9 DTE e os sinais correspondentes em outro conector DB-9 DTE (null-modem).

Tabela 3 – Detalhamento dos sinais dos pinos de um conector DB-9.

DTE	Sinal	null-modem
1	CD (Carrier Detect)	4 (DTR)
2	RD (Receive Data)	3 (TD)
3	TD (Transmit Data)	2 (RD)
4	DTR (Data Terminal Ready)	2 (RD)
5	Common (Signal Ground)	5 (Common)
6	DSR (Data Set Ready)	4 (DTR)
7	RTS (Request to Send)	8 (CTS)
8	CTS (Clear to Send)	7 (RTS)
9	RI (Ring Indicator)	N/C

ANEXOS

Anexo A: Esquema Elétrico da Placa Base E10 (Socket Board)

Este Anexo apresenta os esquemas elétricos da placa base E10 (TINI socket) da plataforma TINI, fabricada pela Dallas Semiconductor.

Figuras:

- Figura 41 – TINI Socket: Parte 1
- Figura 42 – TINI Socket: Parte 2
- Figura 43 – TINI Socket: Parte 3
- Figura 44 – TINI Socket: Parte 4

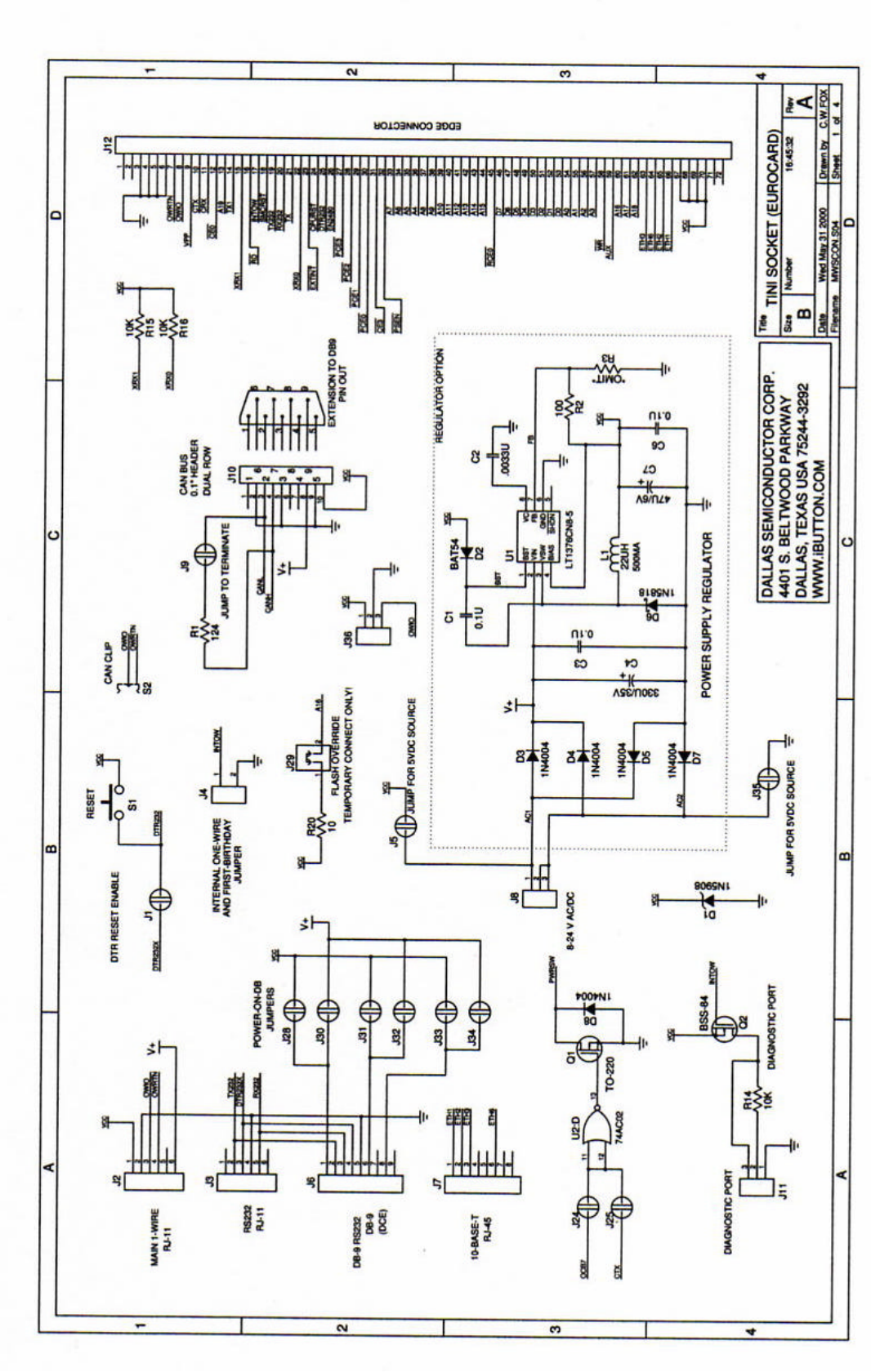


Figura 41 – TINI Socket: Parte 1

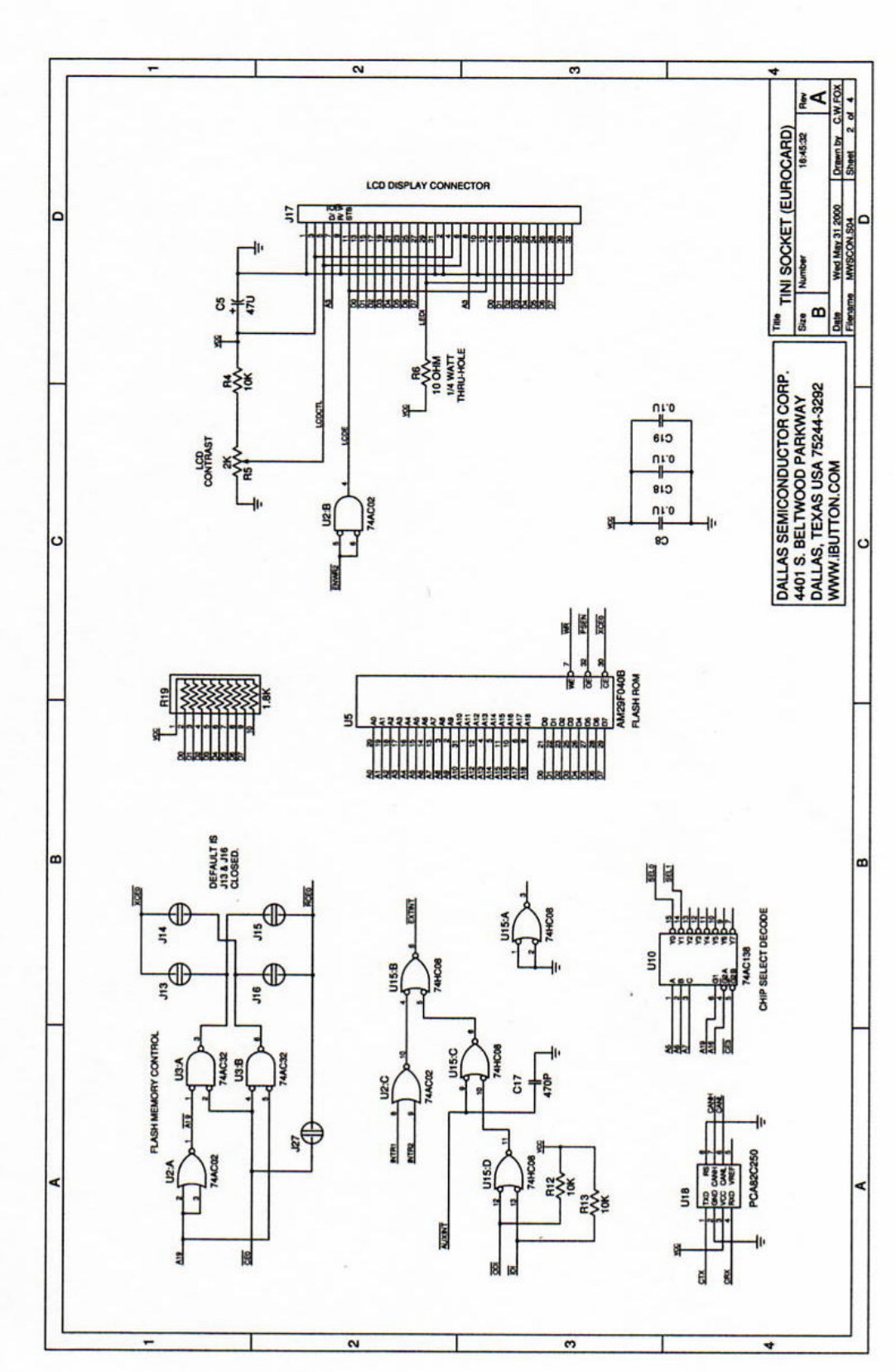


Figura 42 – TINI Socket: Parte 2

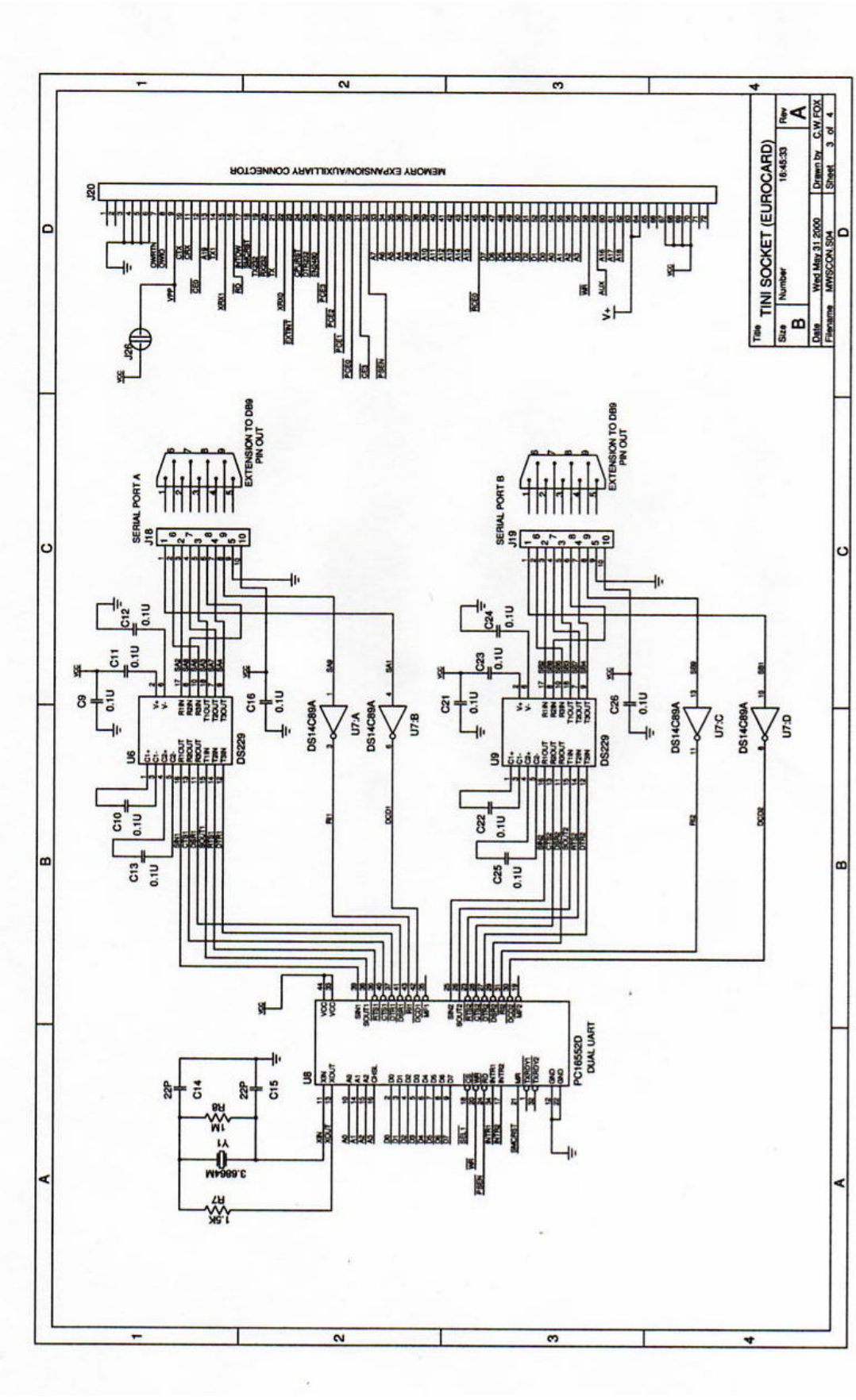


Figura 43 – TINI Socket: Parte 3

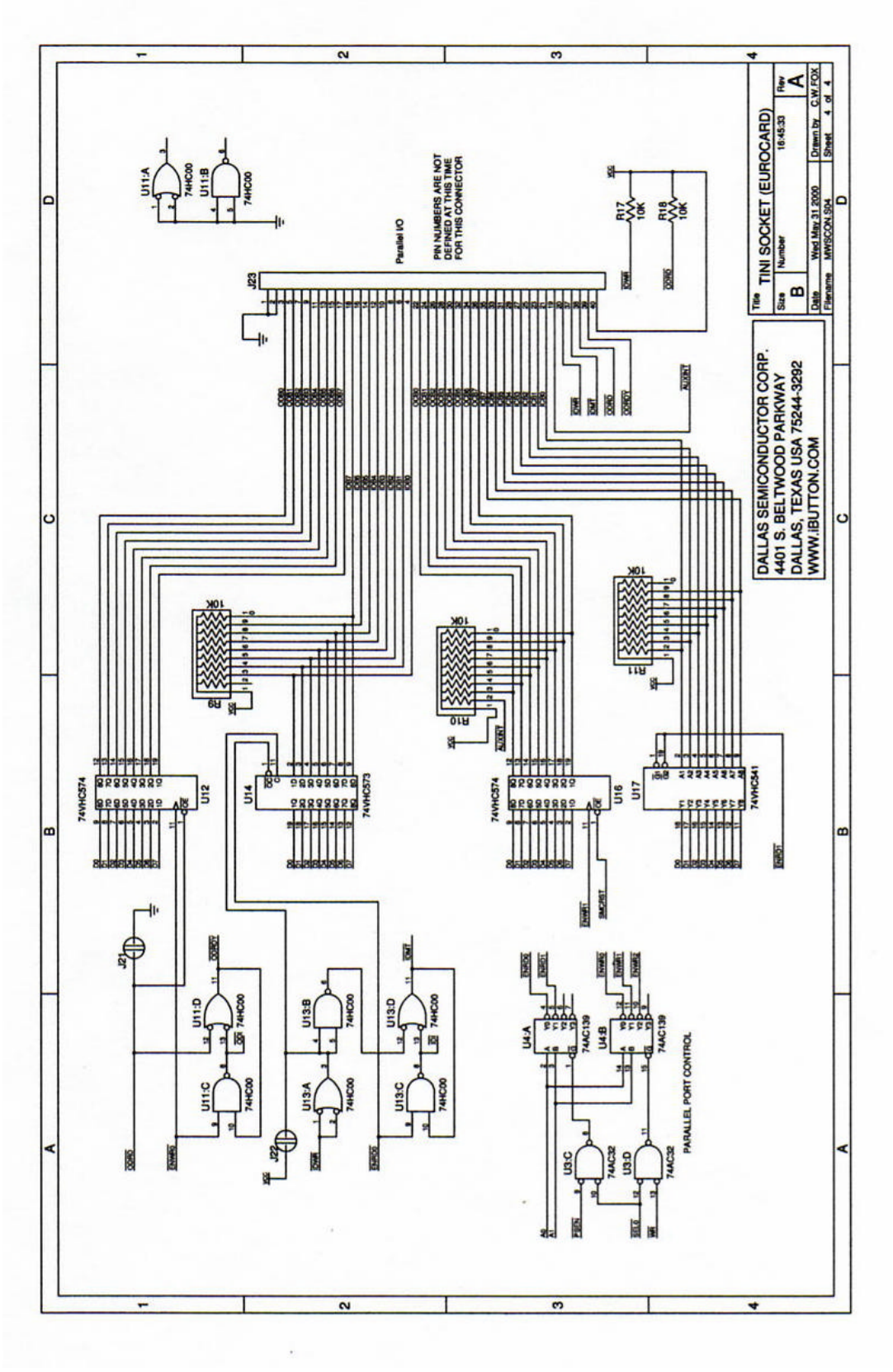


Figura 44 – TINI Socket: Parte 4