CENTRO UNIVERSITÁRIO FEEVALE

GUSTAVO HENRIQUE CERVI

PLATAFORMA RAD PARA APLICAÇÕES LOCAIS OU DISTRIBUÍDAS

Novo Hamburgo, junho de 2005

CENTRO UNIVERSITÁRIO FEEVALE

INSTITUTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
CURSO DE CIÊNCIA DA COMPUTAÇÃO

Plataforma RAD para aplicações locais ou distribuídas

por

GUSTAVO HENRIQUE CERVI
gustavo@overstep.com.br

Trabalho de Conclusão I

Profª. Ms. Marta Rosecler Bez el Boukhari
martabez@feevale.br

Novo Hamburgo, junho de 2005

RESUMO

Atualmente existe uma forte demanda por sistemas distribuídos, o modelo de desenvolvimento WEB é utilizado em larga escala para aplicações, mesmo sendo idealizado para documentos ancorados de forma estática. O paradigma formado em volta desta rede já é amplamente trabalhado, mas possui algumas limitações quando se entra no território de sistemas integrados ou sistemas que requerem um nível de interação maior com o usuário. Alguns modelos de sistemas de várias camadas ou soluções proprietárias resolvem parte dos problemas de alta complexidade, mas acabam por complicar nas aplicações mais simples. Desta forma, este projeto tem como principal objetivo, propor uma idéia de como disponibilizar uma plataforma livre para desenvolvimento rápido de aplicações locais ou distribuídas, utilizando recursos importantes encontrados nas linguagens interpretadas e suas bibliotecas.

Palavras-chave: RAD, framework, sistemas distribuídos, linguagens interpretadas

ABSTRACT

Currently one strong demand for distributed systems exists, the model of development WEB is used in wide scale for applications, exactly being idealized for anchored documents of static form. The paradigm formed in return of this net already widely is worked, but it posses some limitations when it enters in the territory of integrated systems or systems that require a level of bigger interaction with the user. Some models of systems or proprietary solutions solves part of the problems of high complexity, but they finish for complicating in the simplest applications. Of this form, this project has as main objective, to consider an idea of to make a free platform for fast development of local or distributed applications, using important resources found in the interpreted languages and its libraries.

Keywords: RAD, framework, distributed systems, interpreted languages

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

INTRODUÇÃO

Sistemas que possuem interfaceamento com o usuário acoplam o seu ambiente "usuário" com o ambiente "processo". Normalmente, estas interfaces são fortemente ligadas aos métodos que controlam outros processos e outras funções, sendo, em alguns casos, o sistema inteiro englobado em apenas um arquivo binário executável.

O desligamento do ambiente de controle da parte oculta, onde são executados os processos, é claramente visível nos sistemas que utilizam várias camadas, sistemas WEB ou até em outros tipos de sistemas distribuídos. Isto possibilita uma série de vantagens que, através de métodos simples, podem ser utilizadas.

Algumas linguagens interpretadas possuem características interessantes com relação a este contexto, pois possibilitam que uma parte do código seja inserida em tempo de execução, sendo criada até mesmo pelo programa principal.

Este trabalho desenvolverá um estudo das tecnologias para uma plataforma RAD. Esta plataforma tem por característica o uso de uma linguagem interpretada para prototipagem de código fixo e também para geração e inserção de código em tempo de execução.

A comunicação entre as partes cliente e servidor, e ainda entre clientes, poderá ser executada com o uso desta plataforma que implementará funções de comunicação. Esta comunicação é peça-chave para que o trabalho se complete, uma vez que não será feita uma

aplicação em si, mas uma plataforma para o desenvolvimento de aplicações que utilizarão estas tecnologias.

No capítulo três deste trabalho serão vistos os conceitos de redes e formas de comunicações entre computadores e aplicações, bem como o modelo de padronização e a sua aplicação. Seguindo no capítulo quatro serão vistos alguns pontos sobre sistemas distribuídos como introdução, conceitos, arquitetura, modelos e comunicação. No quinto capítulo será visto um resumo abrangente sobre uma linguagem interpretada, com suas características e formas de utilização. Por último, será estudada uma biblioteca de componentes gráficos para formar a parte visual do trabalho.

1. OBJETIVOS

1.1. Objetivo Geral

O objetivo geral deste trabalho é, levantar dados e informações sobre modelos e estruturas de redes, sistemas distribuídos, linguagem interpretada Python e a biblioteca de componentes gráficos Tkinter.

1.2. Objetivos Específicos

Estão divididos em três etapas, a serem alcançadas, que serão vistas a seguir:

- **Plataforma RAD:** Estudar as características da linguagem Python que é considerada RAD, de simples utilização, livre e com recursos suficientes para a implementação da plataforma.
- **Redes e Sistemas Distribuídos:** Levantar informações sobre os modelos existentes de comunicação entre processos e computadores. Seus problemas e características mais relevantes a uma plataforma.
- **Ambiente gráfico:** Estudar o módulo Tkinter, suas características básicas e as formas de utilização dos objetos.

2. JUSTIFICATIVA

Este trabalho é calcado na idéia de que a utilização de algumas características das linguagens interpretadas possam ser utilizadas para a construção de uma plataforma de desenvolvimento RAD (*Rapid Application Development*). A linguagem Python possui as características necessárias para isso, sendo interpretada, dinâmica, orientada a objetos, de fácil aprendizado e considerada RAD (LUTZ, 2001; MARTELLI, 2003).

Os métodos de persistência encontrados em Python permitem que um objeto, instanciado ou não, seja salvo em uma mídia qualquer ou pode ser enviado para outro ponto de uma rede (LUTZ, 2001; MARTELLI, 2003). Isto fundamenta que parte de um código pode ser gerado e enviado, instanciado ou não para um ou vários pontos de uma rede e executado de forma distribuída.

O suporte a redes e multi-processamento fornecido pela linguagem Python possibilita que um protocolo para as trocas de informações entre os pontos seja desenvolvido (LUTZ, 2001), atingindo mais uma parte dos objetivos da plataforma.

Bibliotecas de componentes gráficos como Tkinter, tornam o trabalho de implementação de *interfaces* muito simples e podem tornar ainda mais rápido o desenvolvimento de aplicações (MARTELLI, 2003), reforçando a caracterização RAD da plataforma.

Como neste contexto de linguagem interpretada o código é carregado e interpretado em tempo de execução, outras funções e objetos podem ser criados com base em parâmetros coletados no sistema, fazendo com que partes do código sejam criados em tempo de execução. Esta característica pode não ser relevante para computação formal, mas algumas aplicações de IA, como algorítmos genéticos, (LUGER, 2002) poderão fazer uso desta metodologia que este trabalho propõe.

3. COMUNICAÇÃO E REDES

Os protocolos de comunicação são partes importantes deste trabalho, toda troca de dados e informações será feita por eles. Serão explicados a seguir o modelo OSI, sua aplicação no protocolo TCP/IP, os *sockets* e um exemplo de implementação.

3.1. O Modelo OSI

Segundo Tanenbaum (1996), o modelo OSI foi proposto pela *International Standards Organization* (ISO) como primeiro passo para a padronização dos protocolos utilizados nas várias camadas. Este modelo é chamado de *Open Systems Interconnection Reference Model*. O modelo OSI possui várias camadas que descrevem as funções isoladas dos protocolos.

Nível	***Função***
7	Aplicação
6	Apresentação
5	Sessão
4	Transporte
3	Rede
2	Enlace
1	Físico

Tabela 1 Modelo OSI

Na tabela 1, pode-se ver as várias camadas do modelo OSI que serão descritas, de forma simples, uma vez que não são peças fundamentais para esta plataforma que funcionará em uma topologia superior. A seguir, conforme Tanenbaum (1996) as camadas e suas respectivas explicações:

3.1.1. Camada Física

Esta camada é a de mais baixo nível do modelo, é onde os bits trafegam sobre o canal físico de comunicação. O projeto deve ser de modo que se for enviado o bit "1", o receptor deve receber um bit "1", não um "0". Nesta camada, são definidos valores elétricos e mecânicos como quantos volts representam um sinal "1" ou quantos pinos o conector deve ter, ainda define questões de como a comunicação deve se iniciar ou terminar, fisicamente.

3.1.2. Camada de Enlace

A tarefa principal da camada de enlace é facilitar a transmissão crua dos dados. Define bordas ou janelas para os *frames* de dados. Esta camada suporta seqüência de pacotes e determina ao emissor o reenvio de um pacote perdido, se for o caso. É nesta camada que algum controle de tráfego de baixo nível pode ser implementado.

3.1.3. Camada de Rede

Esta camada define e controla as sub-redes, determinando como os pacotes devem chegar até o destino, fornece rotas para que sejam manipulados até que sejam entregues. As rotas podem ser baseadas em tabelas estáticas ou podem ser de forma dinâmica. Esta camada pode, ainda, determinar o início de uma comunicação e redefinir os mapas das rotas para reduzir a demanda de um roteador ou servidor.

3.1.4. Camada de Transporte

A função básica desta camada é aceitar pacotes de uma sessão, dividir em unidades menores (se for preciso), passar para o próximo nível e assegurar que os pacotes foram entregues corretamente ao receptor (se for estipulado).

Em condições normais, a camada de transporte cria uma conexão distinta para cada transmissão requerida pela sessão. Se a conexão de transporte requer um grande movimento, o transporte deve criar várias conexões, dividindo os dados por elas, para aumentar o fluxo.

Esta camada também determina qual tipo de serviço prover à sessão e, depois, aos usuários da rede. O tipo de transporte mais popular é livre de erros, ponto-a-ponto e que entrega

as mensagens ou bytes na ordem que foram enviados. Outro possível tipo de transporte é de mensagens isoladas sem garantia de ordem ou entrega. Ainda existe o envio de mensagens a múltiplos pontos da rede (*broadcasting*[1]). O tipo de transporte é definido no momento da conexão.

3.1.5. Camada de Sessão

A sessão permite que usuários de diferentes máquinas possam criar sessões entre eles. A sessão permite o transporte de dados e também de serviços melhorados que podem ser úteis em aplicações que as utilizem. Um dos serviços da sessão é manipular o controle de diálogo.

Um serviço relacionado à sessão é o *token management*. Para alguns protocolos, este serviço é fundamental para que múltiplos lados não iniciem a mesma operação ao mesmo tempo. Para isso, *tokens* de sessão podem ser trocados e apenas o lado que possui o *token* pode desenvolver a operação. Outro serviço de sessão é a sincronização, onde a sessão insere *checkpoints*[2] no conteúdo da mensagem para que sejam sincronizadas.

3.1.6. Camada de Apresentação

Esta camada executa certas funções que, diferentemente das outras camadas que se preocupam mais em trafegar os bits, são concentradas na semântica e sintaxe das informações transmitidas. Uma utilização típica pode ser exemplificada como uma conversão entre diferentes tipos de dados como ASCII e Unicode[3].

3.1.7. Camada de Aplicação

A camada de aplicação possui uma grande variedade de protocolos que são amplamente utilizados. Como exemplo, existem centenas de tipos de terminais incompatíveis entre si, e, considerando um editor de texto, em modo texto, que funcione via rede em diferentes tipos de terminais, haverá muitos problemas em função dos tipos de códigos de caracteres especiais (ESC, DEL, etc..). Para isso, vários protocolos são utilizados em nível de

1 Envio de dados para toda a rede.

2 Ponto de chegada, funciona como um marcador para o identificação de posição.

3 Padrão para internacionalização da tabela de caracteres, possui dois bytes para cada caractere.

aplicação, fazendo com que a compatibilidade entre aplicações seja independente do resto da rede.

3.2. (TCP|UDP)/IP

Os protocolos (TCP|UDP)/IP são responsáveis pelas comunicações em nível 4 e 3 do modelo de referência OSI, que são, respectivamente, transporte e rede (TANENBAUM, 1996). Haverá uma explicação sucinta para estas camadas. Em seguida a camada de aplicação (nível 7) será trabalhada de forma um pouco mais detalhada.

Nível	***OSI***	***TCP/IP***
7	Aplicação	FTP, HTTP
6	Apresentação	
5	Sessão	
4	Transporte	TCP, UDP
3	Rede	IP
2	Enlace	LAN, PPP
1	Físico	

Tabela 2 Modelo OSI aplicado ao protocolo TCP/IP segundo Tanenbaum (1996)

3.2.1. IP

No nível de rede, também conhecido como *Network Layer*, ocorre o endereçamento dos pontos de rede; é onde há o roteamento dos pacotes ao destino.

Uma vez que um pacote IP entra em uma rede, ele possui um destino (*unicast*), este destino pode passar por vários pontos (roteamento) até chegar no fim. O protocolo IP trabalha este roteamento de pacotes de forma que os dados atravessem os nós da rede, passando por diversos *gateways*.

3.2.2. TCP e UDP

São desenhados para que haja comunicação entre os *hosts* envolvidos. São responsáveis pelo transporte dos dados. O protocolo TCP (*Transmission Control Protocol*) é orientado a conexões, possui garantia de entrega de seus pacotes, tanto na efetividade quanto na sequência. É muito utilizado em comunicações que requerem garantia de entrega e ordem dos dados por parte do protocolo. O protocolo UDP (*User Datagram Protocol*) é utilizado para envio de dados com mais velocidade, em forma de datagramas, os dados podem não chegar ao seu destino, ou chegar fora de ordem, ficando ao cargo do software de comunicação a ordenação e verificação dos pacotes. Este protocolo é mais utilizado para *streamming* de mídia, onde em um vídeo ou áudio, se forem perdidos alguns *frames*, não faz sentido recuperar para continuar a apresentação.

3.2.3. Aplicação

É neste nível que o trabalho se encontrará, uma vez que haverá comunicação com outros pontos, e utilizará os protocolos TCP/IP. Esta camada utiliza a comunicação proposta pela aplicação, ficando assim, o protocolo final de responsabilidade de especificação da aplicação que a utilizará. A seguir uma tabela sobre alguns tipos de protocolos de aplicação:

Sigla	***RFC (IETF)***	***Função***
HTTP	2616	Comunicação na WEB
TELNET	854 / 855	Emulação de terminal
SMTP	2821	Transporte de e-mail
FTP	959	Transferência de arquivos

Tabela 3 Exemplos de protocolos de aplicação (TCP/IP)

Como exemplo de aplicação, pode-se citar o protocolo HTTP (*HyperText Transfer Protocol* - RFC 2616) que é o padrão para transferência de dados na WEB, conforme Tanenbaum (1996).

No HTTP, cada interação consiste em uma requisição ASCII[4], seguida por uma resposta do tipo MIME[5]. Neste protocolo, existe, basicamente, dois tipos de ações: *requests* e *responses*[6]. A seguir, uma ilustração demonstrando o funcionamento das requisições:

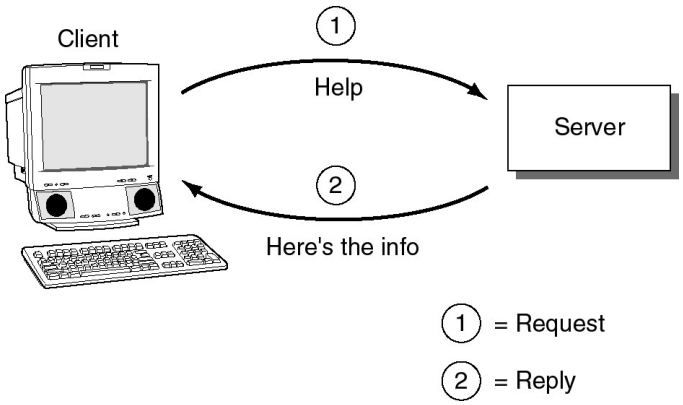

Ilustração 1 Modelo Cliente/Servidor (GALLI, 2000. p.13)

Os *requests* podem ser de dois tipos: *simple request* e *full request*. Um *simple request* se trata de uma linha única contendo um *GET*[7]. A resposta é uma página crua, sem cabeçalho nem formatação MIME. Para testar o funcionamento, basta que se execute um "telnet" no servidor www.w3c.org na porta 80 e digitar a seguinte linha:

```
1  GET /hypertext/WWW/TheProject.html
```

A página retornará sem a indicação de conteúdo (*content-type*).

Diferentemente, um *full request* é indicado pela presença da versão do protocolo. Exemplo:

```
2  GET /hypertext/WWW/TheProject.html HTTP/1.0
```

Mesmo tendo sido o HTTP desenvolvido para WEB, ele recebeu alguns recursos adicionais observando um futuro uso de aplicações e formulários. Estes recursos podem ser acessados conforme a seguinte tabela, juntamente com o *GET* visto anteriormente:

---

4 *American Standard Code for Information Interchange*

5 *Multipurpose Internet Mail Extensions* (RFC 822): padrão para formatação de emails.

6 Também conhecido como *Reply*.

7 Uma forma de requisição. Get, em inglês, significa "adquirir", "receber", "obter", etc..

Método	*Descrição*
GET	Solicita a leitura de uma página WEB
HEAD	Solicita a leitura do cabeçalho da página WEB
PUT	Solicita o envio de uma página WEB
POST	Agrega a um recurso (página WEB)
DELETE	Remove uma página WEB
LINK	Conecta dois recursos existentes
UNLINK	Elimina uma conexão entre dois recursos.

Tabela 4 Métodos HTTP-request

O protocolo que este trabalho utilizará será semelhante a este (HTTP), podendo ainda reter algum tipo de compatibilidade para o caso de algum sistema utilizar um servidor WEB para guardar linhas de código ou formas de mídias que poderão ser executadas ou anexadas a uma interface.

Adicionalmente, outros comandos para requisições de arquivos especiais ou ações de verificação de performance e escalabilidade, poderão ser implementados para uso neste trabalho.

3.3. Berkeley Sockets

Sockets podem ser definidos como adaptadores para conexões de linhas de transmissão de dados. Funcionam de forma que um *host* (servidor) abre uma porta e espera por conexões, outro *host* (cliente) conecta no primeiro e a transmissão começa. As ações dos *sockets* podem ser descritas em oito diferentes primitivas (TANENBAUM, 1996).

Primitiva	***Cliente / Servidor***	***Significado***
SOCKET	SC	Cria uma nova conexão
BIND	S	Agrega um endereço local a um socket
LISTEN	S	Abre a porta para conexões entrantes
ACCEPT	S	Bloqueia o processo enquanto a conexão não é estabelecida (modo Blocking)
CONNECT	C	Estabelece a conexão com o servidor
SEND	CS	Envia dados pela conexão
RECEIVE	CS	Recebe dados pela conexão
CLOSE	CS	Termina a conexão

Tabela 5Primitivas e significados dos Sockets

Os *sockets* serão fortemente utilizados na construção do sistema, onde serão as peças principais para as transferências de dados entre partes. Outras formas de utilização como troca de mensagens entre processos remotos também poderão fazer o uso de *sockets*, uma vez que o sistema poderá ser distribuído e, sendo assim, algumas partes executarem remotamente.

A seguir uma ilustração contendo uma analogia, segundo Galli (2000) de como funcionam as primitivas vistas anteriormente:

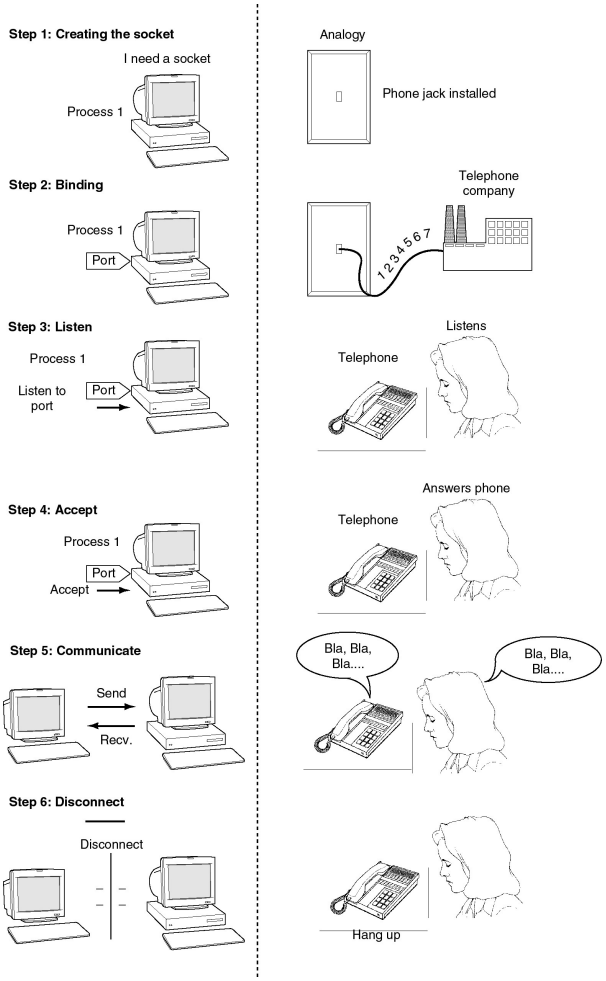

Ilustração 2 Analogia dos Sockets (GALLI, 2000. p.66)

3.3.1. Primitiva *Socket*

Esta primitiva cria uma nova instância de socket e aloca espaço na tabela de transporte. Os parâmetros especificam os formatos utilizados e o protocolo. Um socket criado com sucesso, retorna um *file descriptor* que pode ser utilizado nos eventos seguintes, do mesmo jeito que uma função OPEN executa.

3.3.2. Bind

Sockets recém criados não possuem endereço estipulado. Estes são definidos utilizando a primitiva BIND. Uma vez que o servidor delegou um endereço ao socket, os clientes podem conectar.

3.3.3. *Listen* e *Accept*

A primitiva LISTEN aloca espaço na lista de conexões entrantes para o caso de vários clientes tentarem uma conexão ao mesmo tempo. O LISTEN não bloqueia o processo servidor.

Para que o servidor bloqueie o processo até que receba a conexão do cliente, a primitiva ACCEPT é utilizada. No momento em que a conexão é estabelecida, ela delega um *handler* para que gerencie a transmissão dos dados, isso é feito, normalmente, em forma de objeto ou função.

3.3.4. *Connect* e *Close*

A primitiva CONNECT bloqueia o cliente e executa a conexão com o servidor; quando a conexão é estabelecida, o processo cliente é continuado e as trocas de mensagens podem ser efetuadas. Nesta parte, normalmente é feito um *fork* da aplicação, se for necessário o uso de múltiplos clientes. Cada novo processo ou *thread* "filho" trata de uma conexão.

Close funciona encerrando a conexão. É trabalhada de forma simétrica, quando ambos os lados executarem a função, a conexão é encerrada.

3.3.5. *Send* e *Receive*

Estas duas ações permitem as trocas de informações entre cliente e servidor. Estas informações podem ser em forma de mensagem ou *streamming* tornando os sockets extremamente flexíveis para comunicação entre processos ou hosts. O envio e recebimento de dados, normalmente é executado em uma *thread* separada, quando o RECEIVE é definido para bloquear o processo até que se recebam dados (simétrico). Em situações em que os dados podem trafegar de forma assimétrica, o RECEIVE não bloqueia o processo, fazendo apenas uma verificação na entrada.

É interessante observar que muitos prejuízos na internet foram causados pelo mal uso destas duas funções, uma vez que se o socket não define quantos bytes pode receber, fica a cargo do processo receptor, a manipulação dos dados. Normalmente, quando um buffer é definido de tamanho fixo (*array*) e a leitura se dá por movimentação de blocos de memória, ataques de *buffer overflow* podem ser executados. A função GETS() da linguagem C faz esse tipo de leitura de bloco e jamais deve ser utilizada para uso com sockets. (McGRAW, 2000).

3.3.6. Exemplo de uso de *socket* em C (Posix)

A seguir, um exemplo de uso de *sockets* em C (Posix). É interessante observar as funções (primitivas) citadas anteriormente: *socket*, *bind*, *listen*, *accept*, *read* e *write*, possuem o mesmo nome nas funções e são relativamente simples de se usar. Código de um *socket-server.*

```
3  #include <stdio.h>
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7
8  #define PORTNO = 5000
9
10 int main(int argc, char *argv[])
11 {
12      int sockfd, newsockfd, clilen, n;
13      char buffer[256];
14      struct sockaddr_in serv_addr, cli_addr;
15
16      sockfd = socket(AF_INET, SOCK_STREAM, 0);
17      bzero((char *) &serv_addr, sizeof(serv_addr));
18
19      serv_addr.sin_family = AF_INET;
20      serv_addr.sin_addr.s_addr = INADDR_ANY;
21      serv_addr.sin_port = htons(PORTNO);
22
23      bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
24      listen(sockfd,5);
25
26      clilen = sizeof(cli_addr);
27      newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
28      bzero(buffer,256);
29
30      n = read(newsockfd,buffer,255);
31      printf("Mensagem: %s\n",buffer);
32      n = write(newsockfd, "Mensagem recebida!",18);
33
34      return 0;
35 }
```

4. SISTEMAS DISTRIBUÍDOS

Neste capítulo serão vistos alguns conceitos, problemáticas, arquiteturas, modelos e exemplos de aplicações de sistemas distribuídos. A seguir uma breve introdução sobre o assunto.

4.1. Introdução

Sistemas ditos distribuídos são definidos de diversas formas, por diversos autores, como pode ser visto a seguir:

> **"Coulouris**: Um sistema distribuído é um conjunto de componentes hardware (nós, hosts, máquinas ou computadores) e software interligados através de uma infra-estrutura de comunicações (geralmente uma rede de computadores ou um bus especial, ...), que cooperam e se coordenam entre si através de troca de mensagens.
> **Tanenbaum:** Um sistema distribuído é um conjunto de computadores independentes, interligados através de uma infra-estrutura de comunicações que se apresentam aos seus usuários como um conjunto único, integrado e coerente.
> **Lamport:** Sistema distribuído é aquele em que um defeito em um computador que você nem sabia que existia pode impedir o uso de seu computador.
> **Silberschatz:** Sistema distribuído é um conjunto de processadores fracamente acoplados, interconectados por uma rede de comunicação. Os processadores não compartilham memória ou relógio.
> **Galli:** Conjunto de computadores e processadores heterogêneos conectados por uma rede. Este conjunto trabalha em cooperação para a realização de uma tarefa. O objetivo de um sistema distribuído é prover uma cisão comum e global do sistema de arquivos, *name space*, tempo, segurança e acesso a recursos". (NOVAES, 2005. p.1)

Segundo Coulouris (1994), sistemas distribuídos tiveram seu início com o desenvolvimento de computadores multi-usuários e redes de computadores na década de 1960 e foi estimulado pelo desenvolvimento de computadores pessoais de baixo custo.

A importância do conceito de sistemas distribuídos pode ser verificada pela diversidade de aplicações que o utilizam. A seguir serão vistos alguns tipos, segundo Coulouris (1994):

- **Sistemas operacionais distribuídos:** É construído de forma modular que permite que novos componentes sejam aplicados em resposta as novas necessidades das aplicações. A modularidade se baseia na forma em que se dá a comunicação entre os módulos.
- **Sistemas comerciais:** Vários sistemas comerciais como os bancários ou de companhias aéreas utilizam modelos distribuídos, seja para emissão de *tickets* ou para movimentação de contas bancárias através de caixas eletrônicos.
- **Aplicações WAN[8]:** Uma infinidade de aplicações para internet utilizam o modelo distribuído. Como exemplo, os servidores de resolução de nome (DNS[9]) ou servidores de email (SMTP[10]). Páginas WEB podem possuir informações de vários lugares e processar dados de forma distribuída.
- **Informações multimídia ou de conferência:** Serviços de *brodcasting* de mídia podem ser distribuídos em vários servidores afim de diminuir o tráfego de dados em uma única rede.

4.2. Comparação com sistemas centralizados

Em comparação, segundo Galli (2000), sistemas centralizados possuem todas decisões e dados centralizados em uma localidade. Este tipo de solução é mais simples mas possui uma série de desvantagens que podem comprometer o sistema. Existe uma característica crítica no centro do sistema. Ocorrendo uma falha, o sistema inteiro é passivo de falhar. O tráfego de rede no centro é muito grande, uma vez que os participantes do sistema precisam se comunicar com ele. Por outro lado, a manutenção do sistema é favorecida por ter apenas uma arquitetura a ser mantida.

Sistemas distribuídos também possuem as suas fraquezas. O tráfego na rede inteira é aumentado pelo fato de muitas soluções possuírem informações de *broadcasting*. É também, muitas vezes, difícil de manter o sistema todo atualizado com informações consistentes. Muitas localizações podem receber informações de atualizações antes que as outras.

---

8 *Wide Area Network:* redes de larga escala. Ex: internet.

9 *Domain Name Service:* serviço utilizado para resolver nomes em endereços IP

10 *Simple Mail Transfer Protocol:* protocolo utilizado para transporte de email

4.3. Problemática

Neste capítulo serão vistos algumas problemáticas que atingem os sistemas distribuídos e algumas formas de tratamento.

4.3.1. Concorrência

Quando vários processos coexistem em uma única máquina é dito que estão sendo executados de forma concorrente. Se o computador possui apenas uma CPU, esta executará, intercaladamente, porções de cada processo separadamente. Se o computador possui *N* CPUs, então até *N* processos podem ser executados de forma paralela.

Em sistemas distribuídos existem vários computadores. Cada um com um ou mais CPUs. Se há *M* computadores em um sistema distribuído com uma CPU cada, então será possível executar até *M* processos em paralelo (COULOURIS, 1994).

4.3.2. Escalabilidade

Um sistema escalável é aquele que pode ser modificado para acomodar alterações na quantidade de usuários, recursos e entidades computacionais. Pode ser medida em três diferentes dimensões:

- **Escalabilidade de carga** – Um sistema distribuído deve escalonar a carga do sistema, expandindo ou contraindo o *pool* de recursos para acomodar aplicações "leves" ou "pesadas".
- **Escalabilidade geográfica** – Um sistema geograficamente escalável é aquele que mantém a utilidade e usabilidade sem levar em consideração o quão longe os usuários estão dos recursos.
- **Escalabilidade administrativa** – Não importa quantas organizações diferentes precisam compartilhar um único sistema distribuído, ele deve ser fácil de usar e administrar.

Um protocolo de roteamento é considerado escalável, quando se trata do tamanho da rede, se o tamanho da tabela de roteamento necessária em cada nó aumenta em O(log *N*) onde *N* é o número de nós na rede.

Escalonar verticalmente significa adicionar recursos a um único nó no sistema, como adicionar memória ou um disco rígido mais rápido. Escalonar horizontalmente significa adicionar mais nós ao sistema, como adicionar um novo computador a uma aplicação em um cluster[11]. (WIKIPEDIA, 2005)

4.3.3. Tolerância a falhas

Sistemas computacionais, as vezes, falham. Quando as falhas são produzidas por software ou hardware, os sistemas podem produzir resultados incorretos ou podem travar antes de concluir a tarefa.

Para produzir um sistema tolerante a falhas, existem duas abordagens que devem ser observadas:

- **Hardware redundante**: utilização de componentes redundantes.
- **Software recuperável**: utilização de software desenhado para se recuperar de falhas.

O desenho de software que seja tolerante a falhas envolve o que os dados persistentes devam poder ser retomados (*rollback*) para um estado consistente quando uma falha for detectada. (COULOURIS, 1994)

Segundo Coulouris (1994), a caracterização das falhas que podem ocorrer em um servidor seguem conforme a tabela:

11 Tipo de sistema distribuído, porém fortemente acoplado.

Classe da falha	*Subclasse*	*Descrição*
Falha por omissão		O servidor omite a resposta
Falha na resposta		O servidor responde incorretamente uma requisição
	Falha de valor	Retorna um valor errado
	Falha de transição de estado	Causa um efeito errado nos recursos (ex. Atribui valores errados em ítens de dados)

Tabela 6 Caracterização de falhas

Falhas temporais e bizantinas também fazem parte das classificações de sistemas tolerante a falhas (COULOURIS, 1994), mas não serão implementadas neste trabalho, ficando como ítem-sugestão para trabalhos futuros. Em contrapartida, este trabalho implemenatará o controle de falhas pelos métodos de controle de tempo de espera (*timeout*), por validação de protocolo e tratamento de exceções como, por exemplo, servidor inexistente ou perda de conexão.

4.3.4. Transparência

Transparência é concebida pela supressão das características dos sistemas distribuídos ao usuário. O modelo de referência para sistemas distribuídos da ISO (RM-ODP) identificam oito formas de transparências (COULOURIS, 1994):

- **Transparência de acesso:** objetos locais e remotos podem ser acessados de forma idêntica.
- **Transparência de local:** objetos podem ser acessados sem o conhecimento de sua real localização.
- **Concorrência transparente:** vários processos podem ser executados utilizando recursos compartilhados sem interferirem uns nos outros.
- **Replicação transparente:** várias instâncias do mesmo objeto podem ser replicadas para aumentar a confiabilidade e a segurança sem que o usuário tome conhecimento.

- **Transparência de falhas:** falhas são suprimidas de forma que o usuário ou o sistema termine seu trabalho sem que a mesma tenha sido notada.
- **Migração transparente:** objetos são migrados para partes diferentes do sistema sem que isto fique aparente.
- **Performance transparente:** o sistema pode ser configurado para suportar novos níveis de carga.
- **Escalonagem transparente:** permite que os sistemas sejam escalonados e as aplicações expandidas em escala sem modificar a estrutura ou algorítmos do sistema.

A transparência esconde e torna anônimos os recursos que não são relevantes às tarefas dos usuários e sistemas. Isto nem sempre pode ser requerido, uma vez que a topologia do sistema pode utilizar algumas informações como localização ou dados dos equipamentos (COULOURIS, 1994).

4.4. Arquiteturas

A seguir serão vistas algumas arquiteturas existentes nos sistemas distribuídos de forma simples e concisa com o trabalho.

4.4.1. Sistemas multi-processados

Um sistema multiprocessado é simplesmente um computador que possui mais de um processador na sua placa-mãe ou dentro de seu invólucro. Se o sistema operacional é desenhado para tirar vantagem disso, então é possível executar diferentes processos em diferentes CPUs, ou diferentes *threads* pertencentes ao mesmo processo.

Através dos anos, muitas opções diferentes de multiprocessamento foram exploradas para uso em computação distribuída. CPUs podem ser conectadas por barramento ou por redes, usar memória compartilhada ou ter sua própria memória RAM, ou até mesmo uma abordagem híbrida pode ser utilizada.

Sistemas multiprocessados são disponibilizados atualmente[12] de forma comercial para usuários finais e usuários corporativos. Sistemas operacionais como Mac OS X, Microsoft Windows e Linux já possuem suporte para isto. Recentemente os processadores Intel começaram a utilizar uma tecnologia chamada *Hyperthreading* que permite mais de uma *thread* ser executada na mesma CPU. (WIKIPEDIA, 2005)

4.4.2. Sistemas multi-computadores

Um sistema multi-computador é um sistema feito com vários computadores independentes conectados por redes. Pode ser homogêneo ou heterogêneo: Um sistema homogêneo é onde todas as CPUs são similares e interconectadas pelo mesmo tipo de rede. Cada computador trabalha em uma parte de um único problema. Um sistema heterogêneo é aquele em que cada computador pode ser de um tipo diferente com diferentes características como quantidade de memória ou velocidade de processamento. (WIKIPEDIA, 2005)

4.5. Modelos de interação

Várias arquiteturas de hardware ou software existentes são utilizadas para computação distribuída. Em um nível mais baixo, é necessário interconectar múltiplas CPUs com algum tipo de rede, sem levar em consideração se a rede é feita em uma placa de circuito impresso ou deriva de vários dispositivos fracamente acoplados e cabos. Em um nível mais alto, é necessário conectar processos executados nas CPUs com algum tipo de sistema de conexão. (WIKIPEDIA, 2005)

- **Cliente-servidor:** Um código cliente inteligente contacta o servidor, recebe dados, os formata e mostra em algum tipo de interface ao usuário. As alterações do cliente são enviadas ao servidor assim que ele solicitar uma mudança permanente dos dados.
- **Três camadas:** Este modelo move a inteligência do cliente a uma camada intermediária, sendo então possível a utilização de clientes sem estado. Isto simplifica o desenvolvimento. A maioria das aplicações WEB são de três camadas.

12 Trabalho realizado em 2005.

- **Várias camadas:** Aplicações de várias camadas referem tipicamente a aplicações WEB onde uma requisição pode ser encaminhada a outros serviços. Este tipo de aplicação é a maior responsável pelo sucesso dos servidores de aplicação.
- **Fortemente acopladas (*clustered*):** Se referem a máquinas integradas que executam o mesmo processo em paralelo, subdividindo as tarefas em partes que são feitas individualmente, uma a uma, e depois agrupadas para fornecer um resultado final.
- **Ponto-a-ponto:** Nesta arquitetura não existe uma máquina especial que controla os trabalhos. Todas as responsabilidades são uniformemente divididas pelas máquinas.

4.6. Comunicação

A seguir serão vistas alguns modelos de comunicações entre sistemas e alguns exemplos de aplicações.

4.6.1. Troca de mensagens

A troca de mensagens permite que dois processos se comuniquem copiando os dados compartilhados. Este procedimento se dá no envio da mensagem contendo os dados. Esta forma de comunicação é muito comum quando dois processos não compartilham o mesmo espaço de memória ou se encontram em sistemas diferentes. Comunicação por troca de mensagens envolve duas primitivas *(send* e *receive*) similares as seguintes: (GALLI, 2000)

```
1  send (b, msg);
2  receive (a, msg);
```

Uma ilustração conforme Galli (2000) pode ser verificada a seguir:

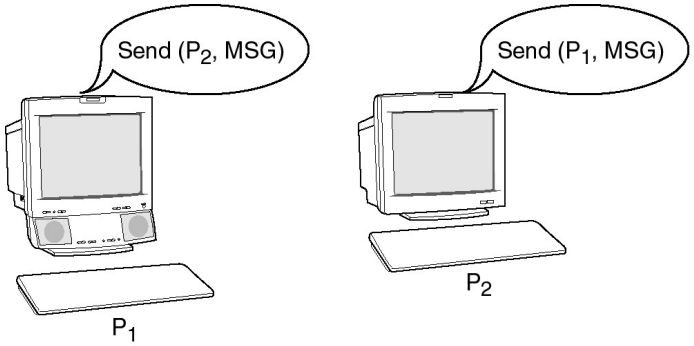

Ilustração 3 Troca de mensagens (GALLI, 2000. p.59)

Estas primitivas podem ser *blocking* ou *nonblocking*. A primeira bloqueia o envio até que receba uma confirmação do receptor e a recepção até que a mensagem seja completamente recebida. Este tipo é de operação (*blocking*) é utilizado onde as trocas de mensagens são síncronas. O segundo tipo (*nonblocking*) é utilizado em conjunto com um *buffer* que libera a execução do programa enquanto as mensagens são trocadas. Este método é utilizado para comunicação assíncrona. (GALLI, 2000)

4.6.2. Memória compartilhada

Em sistemas distribuídos pequenos, com múltiplos processadores, é possível a utilização de comunicação por memória compartilhada. Este modelo permite que vários processos utilizem a mesma área de memória, sendo mais eficiente que troca de mensagens em sistemas que exigem um volume muito grande de dados. Isto se dá porque os dados não são copiados, mas sim acessados do mesmo local pelos vários processos. (GALLI, 2000)

O modelo de memória compartilhada distribuída (DSM) foi introduzida em 1989 por Li Hudak e é amplamente discutida na literatura desde então. Um sistema que emprega este modelo consiste de vários computadores com suas próprias memórias, conectados via rede. Em um DSM, o sistema provê a manutenção do espaço de memória de cada participante que é mapeada para um endereço global. Para gerenciar um DSM, questões como "como a memória será compartilhada?" e "quantos *readers* e quantos *writers* terão acesso a estes dados?" são muito importantes e devem ser analisadas para a escolha do melhor modelo de eficiência.

4.6.3. Chamada remota de procedimento (RPC)

Remote Procedure Calls (RPC) ou chamada remota de procedimento consiste, basicamente, em executar um código em outro *host* sem o envio do mesmo.

RPC é um paradigma simples e popular para implementação de modelos cliente-servidor de computação distribuída. Um RPC é iniciado pelo cliente, enviando uma mensagem de request para um sistema remoto (servidor) para executar um certo procedimento usando os argumentos fornecidos. Uma mensagem de resultado é retornada ao cliente assim que o procedimento do servidor a disponibilizar. (GALLI, 2000)

A seguir poderá ser vista uma ilustração demonstrando como os procedimentos são executados remotamente.

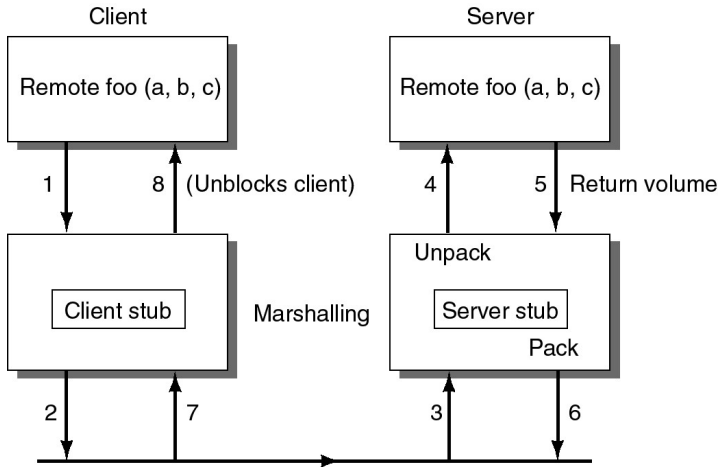

Ilustração 4 RPC - (GALLI, 2000. p.73)

O cliente é bloqueado quando invoca o procedimento remoto, enviando a um *middleware* que prepara o pacote e envia pela rede ao lado servidor. Então o pacote é aberto, executado e retornado ao cliente, desbloqueando o processo. (GALLI, 2000)

4.6.4. Objetos distribuídos

Como exemplo de objetos distribuídos, CORBA (*Common Object Request Broker Architecture*) é um padrão para composição de software. CORBA é mantido pela OMG (*Object Management group*) que define as APIs, protocolos de comunicações e os modelos de

objetos/serviços que possibilitam aplicações heterogêneas escritas em várias linguagens e que em várias plataformas interagirem. (GALLI, 2000)

Em geral, CORBA empacota um código escrito em alguma linguagem em um bloco contendo informações adicionais que descrevem as capacidades do código contido e como executar o mesmo. O pacote resultante pode ser invocado por outros programas pela rede.

CORBA utiliza IDLs (*Interface Definition Language*) para especificar as interfaces que os objetos irão apresentar ao mundo, especifica ainda um "mapeamento" do IDL para uma específica linguagem como C++ ou Java. Este mapeamento define precisamente como os tipos CORBA deverão ser utilizados em ambos, cliente e servidor. Mapeamentos padrão existem para Ada, C, C++, Lisp, Smalltalk, Java e Python. (GALLI, 2000)

4.7. Exemplos de objetos distribuídos

A seguir serão descritos dois exemplos de objetos distribuídos, ambos representados pela linguagem Python.

4.7.1. PYRO

Pyro ou *Python Remote Objects,* segundo Pyro (2005), é uma tecnologia avançada de objetos distribuídos escrita inteiramente em Python que é desenhada para ser muito fácil de se utilizar. Funciona exportando objetos de um servidor para outros (clientes), permitindo que objetos sejam instanciados como se estivessem locais.

A seguir serão apresentados dois exemplos de servidor e cliente, respectivamente:

Servidor:

```
1  import Pyro.core
2  import Pyro.naming
3
4  class JokeGen(Pyro.core.ObjBase):
5      def joke(self, name):
6          return "Sorry "+name+", I don't know any jokes."
7
8  daemon=Pyro.core.Daemon()
9  ns=Pyro.naming.NameServerLocator().getNS()
10 daemon.useNameServer(ns)
11 uri=daemon.connect(JokeGen(),"jokegen")
12 daemon.requestLoop()
```

Cliente:

```
1  import Pyro.core
2
3  # finds object automatically if you're running the Name Server.
4  jokes = Pyro.core.getProxyForURI("PYRONAME://jokegen")
5
6  print jokes.joke("Irmen")
```

4.7.2. SOAP

SOAP originalmente era um acrônimo para *Simple Object Access Protocol*, mas o acrônimo foi eliminado na versão 1.2 do SOAP, originalmente desenvolvido por Dave Winer, Don Box, Bob Atkinson e Mohsen Al-Ghosein em 1998 com o apoio da Microsoft, onde Atkinson e Al-Ghosein trabalhavam naquele tempo. SOAP é, atualmente[13], mantido pela XML *Protocol Working Group* que é parte da *World Wide Web Consortium.* (WIKIPEDIA, 2005)

SOAP é um padrão para a troca de mensagens XML através de uma rede, utilizando, normalmente, o protocolo HTTP e formando as bases dos *Web Services*.

Existem vários modelos de mensagens em SOAP, mas o uso mais comum é o modelo RPC já estudado anteriormente.

Uma mensagem SOAP é contida em um "envelope", dentro deste há duas seções adicionais: o cabeçalho e o conteúdo da mensagem. As mensagens utilizam *namespaces* XML[14]. O *header* contém informações relevantes sobre a mensagem. Como exemplo, o *header* pode conter a data que a mensagem foi enviada ou informações sobre a autenticação.

A seguir é apresentado um exemplo de como um cliente deve formatar uma mensagem SOAP requerendo informações de um produto de uma empresa fictícia.

```
1   <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
2     <soap:Body>
3       <getProductDetails xmlns="http://warehouse.example.com/ws">
4         <productID>827635</productID>
5       </getProductDetails>
6     </soap:Body>
7   </soap:Envelope>
```

A seguir é apresentado um exemplo de como um servidor de uma empresa fictícia deve responder à mensagem de requisição de informações do produto.

13 Trabalho escrito em 2005.

14 *Extensive Markup Language*

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetailsResponse xmlns="http://warehouse.example.com/ws">
      <getProductDetailsResult>
        <productName>Toptimate 3-Piece Set</productName>
        <productID>827635</productID>
        <description>3-Piece luggage set.  Black
Polyester.</description>
        <price>96.50</price>
        <inStock>true</inStock>
      </getProductDetailsResult>
    </getProductDetailsResponse>
  </soap:Body>
</soap:Envelope>
```

A seguir pode ser visto um exemplo em Python de utilização de SOAP.

Servidor:

```
#!/usr/bin/python
from SOAPpy import SOAPServer

def calcula(op1,op2,operacao):
        if operacao == '+':
                return op1 + op2
        if operacao == '-':
                return op1 - op2
        if operacao == '*':
                return op1 * op2
        if operacao == '/':
                return op1 / op2
server = SOAPServer(('localhost',8081))
server.registerFunction(calcula)
server.serve_forever()
```

Cliente:

```
#!/usr/bin/python
from SOAPpy import SOAPProxy

server = SOAPProxy('http://localhost:8081/')
print '2 + 2 = ' + str(server.calcula(2,2,'+'))
print '5 - 2 = ' + str(server.calcula(5,2,'-'))
print '2 * 2 = ' + str(server.calcula(2,2,'*'))
print '6 / 2 = ' + str(server.calcula(6,2,'/'))
```

5. A LINGUAGEM PYTHON

A linguagem Python teve origem em 1990 quando Guido van Rossum a desenvolveu no Stichting Mathematisch Centrum (CWI), na Noruega, para ser sucessora da linguagem "ABC". Existem muitos colaboradores ao redor do mundo trabalhando para crescer e melhorar a linguagem, sendo que algumas empresas dão suporte ao software para fins comerciais. (PYTHON.ORG, 2005; LUTZ, 2001).

Python é uma linguagem dinâmica, objeto-orientada, fracamente tipada, modulada, multi-plataforma, multi-programada, com uma vasta API disponível e crescendo (LUTZ, 2001). Possui uma grande integração com outras linguagens como C/C++ e Java, fazendo com que seja simples desenvolver *wrappers* para bibliotecas desenvolvidas em outras linguagens. Isso é o caso de boa parte de sua API, que é, na sua maioria, desenvolvida em Python, e mais adiante, sendo codificada em C/C++ para executar nativamente na plataforma que se escolher. É interessante lembrar que C/C++ possui diferenças entre plataformas, mas com a evolução dos compiladores, se tornou simples tratar estas diferenças de forma que um código pode ser facilmente adaptado para outras plataformas, desde que não seja fortemente ligado aos recursos proprietários como a API do Windows ou as funções internas do Kernel do Linux.

5.1. Caracterísicas gerais

A seguir, serão descritas algumas características da linguagem Python, bem como uma pequena comparação com Java. Estas características são apenas uma pequena parte da linguagem, mas já servem para se observar algumas semelhanças e diferenças entre soluções equivalentes. Este trabalho utilizará alguns módulos como *socket e threading*.

5.1.1. Modelo de funcionamento

A linguagem Python funciona de forma interpretada e modular. O seu *Framework* é formado pelo seu interpretador e suas bibliotecas padrão. As bibliotecas são agrupadas de forma hierárquico-funcional em módulos. Os módulos podem ser escritos em Python, ou em C/C++. Também existem formas de implementar módulos em outras linguagens através de *wrappers* em C (LUTZ, 2001).

O seu conjunto padrão de módulos, segundo Lutz (2001), é definido pelo mantenedor da linguagem. Para que uma biblioteca nova faça parte deste conjunto, o mantenedor oficial abre uma votação e os membros mais ativos da comunidade decidem se o código tem qualidade e relevância para integrar a distribuição oficial da linguagem.

Existem, também, várias implementações de Python em outras plataformas como Java (Jython), ou .NET (Python.NET) (MARTELLI, 2003), mas que não são objeto deste trabalho.

5.1.2. Aplicação

Segundo Lutz (2001), Python é recomendada para o desenvolvimento dos seguintes tipos de aplicações:

- Utilitários do sistema
  Ferramentas de linha de comando portáveis, teste de sistemas.
- Aplicações para internet
  CGI, Applets, XML, email
- Interfaceamento gráfico com o usuário
  Com APIs como Tk, MFC, Gnome (GTK), KDE (QT)
- Integração com componentes
  C/C++ *library front-ends*, adequações de sistemas
- Banco de dados
  Persistência de objetos, SQL
- Programação distribuída
  Com APIs cliente-servidor como CORBA, COM, SOAP, PYRO
- RAD
  Prototipagem e demonstração

- Módulos baseados em linguagens
  Substitui linguagens especificas por Python em aplicações
- Outros
  Processamento de imagens, numéricos, AI, etc..

Em contrapartida, por não ser uma linguagem compilada e, desta forma, sofrendo perdas de performance, Python não é recomendado para aplicações de tempo real ou sistemas onde o componente crítico é a performance. Não obstante, módulos compilados em C/C++ podem implementar bibliotecas contendo funções ou objetos em código binário nativo da plataforma, tornando o processo atômico crítico, tão eficiente quanto o similar em C/C++.

5.1.3. Ferramentas e extensões

Algumas ferramentas e extensões disponíveis para Python, segundo Lutz (2001), podem ser verificadas na tabela a seguir:

Domínio	***Extensões***
Programação do sistema	Sockets, threads, signals, pipes, RPC, Posix bindings
Interface gráfica	Tk, PMW, MFC, X11, wxPython, KDE, GNOME
Interface com Banco de Dados	Orcale, Sybase, PostGreSQL, mSQL, Firebird, MySQL, persistence, dbm
Ferramentas para Microsoft Windows	MFC, COM, ActiveX, ASP, ODBC, .NET
Ferramentas para internet	Jython, CGI, HTML/XML parsers, email, Zope
Objetos Distribuídos	DCOM, CORBA, ILU, Fnorb, SOAP

Tabela 7 Ferramentas e extensões para Python

Python ainda conta com uma ferramenta chamada Psyco, que otimiza o código tornando-o, em alguns casos, tão veloz quanto C/C++. Esta ferramenta funciona como um pré-processador, executando loops e chamadas recursivas de uma só vez, atalhando o processamento de métodos.

5.1.4. Comparação básica com Java

Python, comparada com Java, possui as mesmas características relevantes a este trabalho. Ambas compartilham aspectos como multi-programação, vasta API, multi-plataforma, consideradas "maduras" (estáveis) com um suporte suficientemente bom às redes e componentes gráficos.

Segundo Lutz (2001), houve uma grande discussão na comunidade científica envolvida sobre o tema Python x Java. A discussão tornou-se mais amena quando chegou-se a conclusão que Java, por ser considerada uma linguagem de complexidade semelhante ao C/C++ (de onde derivou), foi desaconselhada para prototipagem ou desenvolvimento RAD de aplicações simples (*scripting*). Ainda segundo Lutz (2001), Python é mais simples, mais aberta e mais fácil de se aprender e trabalhar, podendo ser, inclusive, embarcada em Java ou em C/C++ de forma a complementá-las.

Estes argumentos traduzem a escolha da linguagem Python para o desenvolvimento deste trabalho.

5.2. Características sintáticas e semânticas

Python não é muito diferente das outras linguagens interpretadas de alto nível. Possui estruturas de controle, decisão e etc. Nos sub-tópicos seguintes, serão vistos alguns pontos semânticos e sintáticos da linguagem segundo Hoffmann (2004).

5.2.1. *Keywords* e Identificadores

Existem poucas palavras reservadas em Python, são elas:

```
1 and       del       for       is        raise
2 assert    elif      from      lambda    return
3 break     else      global    not       try
```

```
4 class     except    if        or        while
5 continue  exec      import    pass      yield
6 def       finally   in        print
```

Os identificadores devem seguir a seguinte gramática:

```
7 (letra | "_") (letra | número | "_")*
```

5.2.2. Literais

Literais podem ser delimitados e referenciados de várias formas, a seguir os modos de utilização dos literais:

```
1 "Uma string entre duas aspas"
2 'Uma string delimitada por apóstrofes com “ASPAS” dentro'
3 '''Uma string contendo “aspas” e 'apóstrofes' e ainda \n (newline) '''
4 """ uma string delimitada por três aspas """
5 u'uma string unicode'
6 U"outra string unicode"
7 r'uma string raw onde os escapes '\' são mantidos'
8 R"outra raw string" -- raw strings não podem terminar com um '\'
9 ur'uma string unicode e raw'
10 UR"outra string unicode e raw"
```

5.2.3. Tipos

Três tipos básicos podem ser descritos: Numérico, Seqüencial e Dicionário. O tipo numérico, utilizado para cálculos e indexação, pode ser referenciado de tais formas:

```
1 Decimal inteiro: 1234, 1234567890546378940L (ou l)
2 Octal inteiro: 0177, 01777777777777777777L (começa com um 0)
3 Hex inteiro: 0xFF, 0XFFFFFFFFFFFFFFFFL (começa com 0x ou 0X)
4 Long Inteiro (precisão ilimitada): 1234567890123456L (termina com L ou l)
  ou long(1234)
5 Float (precisão double): 3.14e-10, .001, 10., 1E3
6 Complex: 1J, 2+3J, 4+5j (termina com J ou j, o '+' separa as partes real
  e imaginária)
```

O tipo seqüencial é muito utilizado para substituição e processamento de dados, sendo muito versátil para várias soluções como processamento de textos.

```
1 Strings: '', '1', "12", 'olá\n'
2 Tuplas: () (1,) (1,2)
3 Listas: [] [1] [1,2]
```

Seqüências podem ser quebradas utilizando a seguinte sintaxe:

```
1 [índice-início : índice  [ : passo]]. Índice-início é 0 como padrão,
  índice tem como padrão len(seqüência) e passo tem como padrão 1.
```

As formas de utilização são exemplificadas a seguir:

```
1  a = (0,1,2,3,4,5,6,7)
2  a[3] == 3
3  a[-1] == 7
4  a[2:4] == (2, 3)
5  a[1:] == (1, 2, 3, 4, 5, 6, 7)
6  a[:3] == (0, 1, 2)
7  a[:] == (0,1,2,3,4,5,6,7) # copia a seqüência.
8  a[::2] == (0, 2, 4, 6) # de dois em dois.
9  a[::-1] = (7, 6, 5, 4, 3 , 2, 1, 0) # Ordem reversa.
```

Dicionários são utilizados para dar valor a índices em partes de código e aceitam qualquer tipo de formato. São utilizadas conforme segue:

```
1  Dicionários: {} {1 : 'um'} {1 : 'um', 'próximo': 'dois'}
```

5.2.4. Operadores e ordem de resolução

A seguir, a lista de operadores e sua ordem de resolução, serão respeitados os valores de *eval,* conforme a tabela a seguir.

Operador	*Comentário*
, [...] {...} `...`	Tupla, lista e dict. criação; string conv.
s[i]; s[i:j]; s.*attr* f(...)	Indexação e corte; atributos, cham. de func.
+x, -x, ~x	Operadores unários
x**y	Potenciação
x*y; x/y; x%y	Mult., divisão, módulo
x+y; x-y	Adição, subtração
x<<y; x>>y	Bit shifting
x&y	Bitwise E
x^y	Bitwise OU exclusivo
x|y	Bitwise OU
x<y; x<=y; x>y; x>=y; x==y; x**!**=y; x**<>**y x **is** y; x **is not** y x **in** s; x **not in** s	Comparação, identificação, participação
not x	Negação booleana
x **and** y	E booleano
x **or** y	OU booleano
lambda *args*: *expr*	Função anônima

Tabela 8 Operadores e ordem de resolução

5.2.5. Controle de fluxo

A tabela a seguir descreve as estruturas de controle de fluxo e seus resultados.

Statement	*Resultado*
if *condition*: *suite* [**elif** *condition*: *suite*]* [**else:** *suite*]	se/então usuais
while *condition*: *suite* [**else:** *suite*]	O `else` é executado assim que o *loop* termina, a não ser que o *loop* receber um `break`.
For *element* **in** *sequence*: *suite* [**else:** *suite*]	Itera pela *seqüência*, atribuindo cada elemento para *element*. O `else` é executado assim que o *loop* termina, a não ser que o *loop* receber um `break`.
break	Aborta imediatamente o `for` ou `while`.
continue	Executa a próxima iteração de um `for` ou `while`.
return *[result]*	Termina a função ou método e utiliza uma tupla para retornar mais de um parâmetro. Se nada for especificado em *result,* 'None′ será enviado.

Tabela 9 Controle de fluxo e seus resultados

5.2.6. Controle de exceções

Os controles de exceções seguem conforme a tabela descreve:

Statement	*Resultado*
assert *expr*[, *message*]	*expr* is resolvida. Se falso, executa a exceção.
try: *suite1* [**except** *exception* [, *value*]**:** *suite2*]+ [**else:** *suite3*]	Os statements na *suite1* são executados. Se uma exceção ocorrer, procura na cláusula `except` por uma exceção. Se encontrar, executa a *suite* desta cláusula. Senão executa a *suite* na cláusula `else`.
try: *suite1* **finally:** *suite2*	Os statements na *suite1* são executados. Se nenhuma exceção ocorrer, executa a *suite2*.
raise *exceptionInstance*	Instancia uma classe derivada de `Exception`
raise *exceptionClass* [,*value* [, *traceback*]]	Instancia uma classe derivada de `Exception` com os opcionais.
raise	Se aplicada sem argumentos, acaba por executar a exceção anterior.

Tabela 10 Tabela de controle de exceções

5.3. *Threading* em Python

Um *Thread* é um fluxo de controle que compartilha o estado global com as outras *threads;* todas as *threads* aparentam executar simultaneamente. Um processo é uma instância do programa que está sendo executado e será visto mais adiante. Algumas vezes, se pode obter um melhor resultado utilizando vários processos do que utilizando *threads,* uma vez que o sistema operacional protege os processos um do outro. Processos que desejam trocar mensagens entre si devem utilizar IPCs *(Inter-Process Communications)*, arquivos, *sockets* ou banco de dados (MARTELLI, 2003).

Python oferece *multithreading* em plataformas que suportam este recurso, como Win32, Linux e a maioria das variantes do Unix. Python não chaveia aleatóriamente os *threads*,

mas utiliza um *Global Interpreter Lock* (GIL) para assegurar que o chaveamento só se dará onde o *bytecode* ou o código C liberar o GIL (o código C do Python libera o GIL quando faz I/O ou em *Sleep*)

Python disponibiliza *multithreading* em dois módulos diferentes, um em baixo nível chamado *thread*, que não é recomendado a utilização direta no código, e outro chamado *threading* que é feita sobre a anterior porém mais desenvolvida.

5.3.1. O módulo *threading*

Este módulo é feito sobre o módulo *thread* que é desenvolvido em mais baixo nível e fornece funcionalidades de *multithreading* de uma maneira mais utilizável. *Threads* em Python é muito semelhante ao modelo utilizado em Java. A seguir, um exemplo de utilização é apresentado:

```
1  import threading
2
3  var = 1
4
5  class testeThread ( threading.Thread ):
6    def run ( self ):
7      global var
8      print 'Thread ' + str ( var ) + ' falando.'
9      print 'Olá e tchau.'
10     var = var + 1
11
12 for x in xrange ( 20 ):
13   testeThread().start()
```

Para o sincronismo na utilização de *streams*, o módulo *threading* provê um recurso de bloqueio chamado *threading.lock*. A seguir um exemplo de utilização é apresentado:

```
1  import threading
2
3  class mythread(threading.Thread):              # instância thread
4    def __init__(self, myId, count):
5      self.myId = myId
6      self.count = count
7      threading.Thread.__init__(self)
8    def run(self):                               # lógica run
9      for i in range(self.count):                # sincronismo no acesso
10       stdoutmutex.acquire()
11       print '[%s] =>' % (self.myId, i)
12       stdoutmutex.release()
13
14 stdoumutex = threading.Lock()
15 threads = []
16
17 for i in range(10):                           # monta 10 threads
```

```
18   thread = mythread(i, 100)
19   thread.start()
20   threads.append(thread)
21
22 for thread in threads:                          # espera até a finalização
23   thread.join()                                 # das threads
24
25 print 'Saindo da thread principal'
```

5.4. Processos em Python

Outra forma de implementar código multi-programado é utilizando o método de bifurcação de processos (*forking*). Utilizando *fork*, o sistema operacional cria uma cópia do programa na memória e executa a nova cópia em paralelo ao já existente. Alguns sistemas não copiam realmente o programa original, por ser uma operação muito custosa para o sistema, mas a nova cópia funciona como se fosse uma cópia literal. (Lutz, 2001)

Após esta operação de cópia, o processo original passa a ser chamado de *processo-pai* e os processos criados pelo *fork* são chamados de *processos-filho*. Em geral, *processos-pai* podem criar qualquer quantidade de *processos-filho* que podem, por sua vez, criar outros *processos-filho* de forma independente. (Lutz, 2001)

É provável que seja mais simples de se entender esta parte utilizando exemplos, então a seguir é apresentado um exemplo de programa Python que cria novos processos enquanto a tecla “q” não é pressionada.

```
1  import os
2
3  def filho():
4    print 'FILHO: Olah!', os.getpid()
5    os._exit(0)   # volta para o loop do processo-pai
6
7
8  def pai():
9    while 1:
10     novoPid = os.fork()
11     if novoPid == 0:
12       filho()
13     else:
14       print 'PAI: Olah', os.getpid(), novoPid
15     if raw_input() == 'q': break
16
17 pai()
```

O procedimento *fork* encontrado no Python é um simples *wrapper* para o *fork* padrão da biblioteca C.

É importante observar que este modelo de multi-programação não funciona no Windows porque o mesmo não suporta este tipo de função, mas funciona de forma concisa no Linux e no Unix. (Lutz, 2001)

5.5. Serialização

Serialização de objetos é um método utilizado para converter instâncias de classes, partes de código ou variáveis em uma seqüência de bytes que podem ser guardados em um arquivo ou enviado pela rede via *sockets.* Python suporta serialização de objetos em três formas: *Pickling, Shelving e Marshaling*. (LUTZ, 2001)

5.5.1. Pickling

O método *Pickling,* que significa "pôr em conserva", faz com que os objetos sejam despachados (*dump*) para um arquivo (persistência) ou socket (envio). A seguir será visto um exemplo de código que serializa um dicionário em um arquivo:

```
1 import pickle
2 table = {'a': [1,2,3], 'b':['foo', 'bar'], 'c': {'nome':'bob'}}
3 mydb = open('dbase', 'w')
4 pickle.dump(table, mydb)
```

É interessante observar na linha 3, onde o *handler* "mydb" é do tipo arquivo, mas poderia ser do tipo *socket*, uma vez que o Python trata *sockets* como se fossem arquivos.

A seguir será visto um exemplo de como recuperar o arquivo serializado:

```
1 import pickle
2 mydb = open('dbase', 'r')
3 table = picle.load(mydb)
4 print table
```

5.5.2. Shelving

Este objeto utiliza *Pickling* internamente para serializar os objetos e traz uma facilidade a mais em relação ao anterior: permite que os objetos sejam tratados como um dicionário, fazendo com que vários objetos possam ser serializados de uma forma única e melhorando a organização. Esta facilidade conta com mais uma adição, o módulo não lê para a memória todo o arquivo, mas apenas os objetos utilizados, otimizando a carga da memória. (LUTZ, 2001)

A seguir será visto um exemplo de gravação e outro de leitura de objeto utilizando *shelve*.

```
1 import shelve
2 mydb = shelve.open('dbase')
3 table1 = {'a': [1,2,3]}
4 table2 = {'b':['foo', 'bar']}
5 mydb = shelve.open('dbase')
6 mydb['table1'] = table1
7 mydb['table2'] = table2
8 mydb.close
```

```
1 import shelve
2 mydb = shelve.open('dbase')
3 print mydb['table1']
```

No primeiro código, duas tabelas são inseridas em um arquivo para depois ser utilizada no segundo código.

5.5.3. Marshalling

Este método permite que objetos sejam serializados em arquivos da mesma forma que o módulo *Pickle*, porém os dados gravados são em formato binário compilado (*bytecode*) e podem ser incompatíveis entre versões diferentes de Python. (PYTHONLR, 2005)

Este módulo será evitado ao máximo por não garantir compatibilidade nem segurança para o sistema.

6. MÓDULO TKINTER

Tkinter é uma biblioteca de componentes para interação com os usuários através de interfaceamento gráfico com o usuário (GUI[15]). Programada através de um conjunto de ferramentas (*toolkit*) que contém uma série de componentes gráficos (*widgets*) (MARTELLI, 2003).

Segundo Martelli (2003), Tkinter é a biblioteca gráfica (GUI) para Python mais difundida[16]. É construída através do empacotamento (*wrapping*) da biblioteca multi-plataforma Tk, que é também utilizada em conjunto com outras linguagens interpretadas como TCL[17] e PERL[18].

Tkinter, assim como TCL/TK, executa em Windows, Macintosh e sistemas derivados de Unix. Para Windows, Tkinter já faz parte da instalação do Python como padrão, bem como as partes da linguagem TCL/TK necessárias para a execução. No Unix, é necessária a instalação separada do TCL/TK.

6.1. Aspectos fundamentais do Tkinter

O módulo Tkinter facilita o desenvolvimento de aplicações GUI. Para começar, é necessário apenas a importação do módulo, a criação, configuração e posicionamento dos objetos e a execução do loop de execução.

A seguir, é apresentado um exemplo de aplicação GUI com Tkinter:

---

15 *Graphical User Interface:* Interface Gráfica com o Usuário.

16 No ano da publicação de seu livro – 2003.

17 *Tool Command Language*: linguagem interpretada com sintaxe semelhante a execução de aplicações por linha de comandos.

18 *Practical Extraction and Report Language*: outra linguagem interpretada.

```
1  import sys, Tkinter
2  Tkinter.Label(text="Olá!").pack()
3  Tkinter.Button(text="Sair", command=sys.exit).pack()
4  Tkinter.mainloop()
```

As chamadas para os objetos *Label* e *Button*, criam os respectivos *widgets* e retornam eles como resultado. Uma vez que não foi definida uma janela para os objetos, Tkinter colocará os objetos na janela principal da aplicação. É possível ainda atribuir um gerenciador de *layout* para a janela, que fará a adequação das posições em relação ao tamanho e orientação das janelas.

Os eventos são atribuídos em forma de comandos, como pode ser visto na linha 3 do exemplo anterior. Um evento é dado ao botão, informando que quando pressionado, deve ser executado a função exit() do modulo sys, importado na linha 1.

Todas as *strings* utilizadas no módulo Tkinter são unicode.

6.2. Aspectos fundamentais dos *widgets*

O módulo Tkinter disponibiliza vários tipos de objetos (*widgets*) e a maioria deles possuem aspectos e características em comum. Todas os *widgets* são instâncias de classes que derivam da classe Widget, que é uma classe abstrata.

Para instanciar qualquer tipo de *widget*, é necessário 'chamar' a classe do *widget*. O primeiro argumento é a janela onde ficará o objeto. Se for omisso, o *widget* ficará na janela principal da aplicação. Todos os outros argumentos de um *widget* "w" podem ser alterados pelo comando w.*config(option=value)*, e consultadas pela função *w.cget('option')*.

6.2.1. Atributos comuns aos *widgets*

Muitos *widgets* aceitam algumas opções comuns. Algumas opções afetam características como cor, tamanhos e posições. Ainda existem várias outras opções que podem ser comuns entre alguns tipos de *widgets* como: *anchor, command, font, image, justify, relief, state, takefocus, text* e *textvariable.*

6.2.2. Métodos comuns aos *widgets*

Assim como os atributos, os métodos também são comuns entre vários tipos de *widgets*. Dentre eles estão: *cget, config, focus_set, greb_set, grab_release, mainloop, quit, update, update_idletasks, wait_variable, wait_visibility, wait_window, winfo_height* e *winfo_width.*

6.2.3. Objeto "variável"

Tkinter possui uma característica interessante, que é o uso de uma variável externa para definição de um atributo de um ou vários objetos. Isto se dá criando uma variável e atribuindo-a nos *widgets*. Quando esta variável muda, os atributos que referenciam a mesma, também mudam. A seguir é apresentado um exemplo de utilização deste método:

```
1  import Tkinter
2
3  root = Tkinter.tk()
4  tv = Tkinter.StringVar()
5
6  Tkinter.Label(textvariable=tv).pack()
7  Tkinter.Entry(textvariable=tv).pack()
8
9  tv.set('Olá!')
10
11 Tkinter.Button(text="Sair", command=root.quit).pack()
12
13 Tkinter.mainloop()
14 print tv.get()
```

Quando o programa chegar na linha 9, o texto da *Label* e o conteúdo do *Entry* será "Olá!", uma vez que a variável "tv" receberá este valor. Esta variável poderá, ainda, ser modificada no campo *Entry*. Depois que o programa terminar, o *print* da linha 14 mostrará o seu valor final.

6.3. Widgets

O módulo Tkinter provê uma grande quantidade de *widgets* que cobrem a maioria das necessidades de uma aplicação GUI simples. A seguir será visto, de maneira sucinta, alguns objetos comuns, seus atributos e métodos. Esta referência a seguir serve apenas para ilustrar o uso e a forma como os *widgets* são configurados. Por motivo de relevância e espaço, a referência completa não será vista.

6.3.1. Button

Este objeto é responsável pelo *widget Button*, que cria um botão na janela. Este botão pode conter um texto *(text='texto'),* uma imagem *(image=objetoImagem)* e um evento agregado *(command=comando)*. Ainda possui os métodos e atributos comuns vistos anteriormente e mais dois métodos: *flash* e *invoke.*

6.3.2. Entry

O objeto *Entry* cria um campo de entrada de texto na janela. Este campo é fundamental para qualquer tipo de cadastro onde haja entrada de informações. A instância do *Entry* provê vários métodos que podem ser utilizados para diversas refinações, mas a maioria da aplicações utilizam apenas três métodos:

```
1  e.delete(0, END)                 # apaga o conteúdo
2  e.insert(END, algumaString)      # insere algumaString no campo
3  algumaString = e.get()           # atribui à algumaString o conteúdo
```

Este objeto ainda suporta a atribuição do estado que pode ser desabilitado *(state=DISABLED)* e normal *(state=NORMAL).*

6.3.3. Label

Este objeto cria um rótulo para um texto qualquer. Não pode ser mudado pelo usuário, mas pode ser feito pelo programa. Podem ainda ser do tipo *image* e conter uma imagem no seu interior.

Uma instância da classe *Label* não permite que o usuário copie o conteúdo ao *clipboard*. Para isso, é recomendado que se use um campo *Entry* com a opção *state=DISABLED* para que o usuário não altere o conteúdo.

CONSIDERAÇÕES FINAIS

Este trabalho é requisito-parte como aprovação da disciplina de Trabalho de Conclusão I, e será continuado na disciplina de Trabalho de Conclusão II. Em consideração a isto, os objetivos descritos no início deste ainda não estão alcançados, ficando a cargo desta parte do trabalho o levantamento teórico e a fundamentação para a continuidade do mesmo.

Foram vistos conceitos de redes de computadores, sistemas distribuídos e linguagens interpretadas, com ênfase na linguagem Python, que será aplicada a este trabalho. Estes três assuntos são muito importantes a este projeto que abordará diretamente implementação, redes e sistemas distribuídos.

Em redes, os conceitos básicos como modelo OSI e como este se aplica ao protocolo TCP/IP foram muito importantes para que o projeto produza a devida comunicação entre as partes. Ficou claro que os conceitos de camadas especificam os limites de cada componente do modelo.

Sistemas distribuídos foram abordados de forma simples, uma vez que este trabalho implementará aplicações distribuídas desta forma, ou seja, sem o compromisso formal com os conceitos mais avançados de tolerância a falhas, escalabilidade e transparência. Em contrapartida, ficou claro a importância dos mesmos para a evolução futura da plataforma.

Para que a plataforma se concretize como ambiente de execução e desenvolvimento, o estudo de uma linguagem que dê o devido suporte, tanto ao desenvolvimento quanto a sua aplicação, foi essencial para a construção deste trabalho. Foi visto que Python é adequada ao

desenvolvimento e aplicação do trabalho, contendo todos os requisitos necessários para a implementação de um sistema RAD com suporte a execução distribuída.

Em alguns pontos, como o capítulo da linguagem Python, que está descrita de forma resumida, não se levou em consideração aspectos internos da linguagem, uma vez que este trabalhado executará um nível acima deles e não no mesmo ou em níveis inferiores. Ainda neste capítulo, aspectos mais comuns da linguagem foram suprimidos para que este não se torne extenso demais, uma vez que não é objetivo do trabalho criar uma referência da linguagem.

Para que este texto não seja inflado com informações irrelevantes, aspectos que não serão utilizados não foram abordados, mas que se, por ventura, se tornarem necessários, então serão devidamente levantados e fundamentados no Trabalho de Conclusão II. Isto poderá ser necessário em caso de desenvolvimento de algum módulo adicional para Python utilizando a linguagem C ou outra linguagem que permita a execução, ficando desta forma, o levantamento teórico deste procedimento como objeto a ser trabalhado no momento da utilização.

TRABALHOS FUTUROS

Como sugestão para trabalhos futuros, a abordagem de um sistema tolerante a falhas mais aprofundada e que contemple falhas bizantinas e temporais, bem como ocultações, falhas de espera, amnésia e travamento de servidores pode ser implementado. Isto tornará a plataforma mais concreta em função dos paradigmas da computação distribuída.

O levantamento de outros *toolkits* semelhantes como GTK ou wxWidgets poderão ser feitos para usuário que sejam mais íntimos a eles. Esta adaptação poderá ser feita com o mínimo de alteração das características da especificação do trabalho e poderão ser muito úteis para dispositivos móveis que suportem apenas outros *toolkits*.

O estudo da tradução de alguns módulos da plataforma para linguagens compiladas poderá ser uma grande ajuda para o desempenho da mesma. Alguns pontos poderão ser desenvolvidos em C/C++ ou alguma linguagem funcional como Haskel ou OCAML, tornando a mesma mais eficiente.

REFERÊNCIAS BIBLIOGRÁFICAS

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. **Distributed Systems:** Concepts and Designs. 2ª Ed. Inglaterra: Addison-Wesley, 1994. 644p.

GALLI, Doreen L. **Distributed Operational Systems:** Concepts & Practice. EUA: Prentice Hall, 2000. 463p.

HOFFMANN, Chris et al. ***Python Quick Reference.*** s/l: 2004.

LUGER, George F. **Inteligência Artificial**: Estruturas e estratégias para solução de problemas complexos. 4ª Edição. Porto Alegre: Bookman, 2002. 774p.

LUTZ, Mark. **Programming Python, Second Edition**. EUA: O'Reilly, 2001. 1255p.

MARTELLI, Alex. **Python in a Nutshell**. EUA: O'Reilly, 2003. 636p.

McGRAW, Gary; VIEGA, John. **Learning the basics of buffer overflows.** IBM DevelopperWorks, 2000. Disponível em: <http://www-106.ibm.com/developerworks/security/library/s-overflows/>. Acesso em: maio 2005.

NOVAES, Reynaldo Cardoso. **Apostila de aula.** Novo Hamburgo: FEEVALE, 2005. 17p.

PYRO. **Python Remote Objects**. Disponível em: http://pyro.sourceforge.net/. Acesso em: maio de 2005.

PYTHON.ORG. **Página oficial da comunidade Python**. Disponível em: <http://www.python.org>. Acesso em: maio de 2005.

PYTHONLR. ***Python Library Reference.*** Disponível em <http://www.python.org/doc>. Acesso em: maio de 2005.

TANENBAUM, Andrew S. **Computer Networks**. EUA: PTR, 1996. 813p.

WIKIPEDIA. **Enciclopédia virtual livre**. Disponível em: <http://en.wikipedia.org>. Acesso em: maio de 2005.