CENTRO UNIVERSITÁRIO FEEVALE

GUSTAVO HENRIQUE CERVI

PLATAFORMA RAD PARA APLICAÇÕES LOCAIS OU DISTRIBUÍDAS

Novo Hamburgo, novembro 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 II

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

Novo Hamburgo, novembro de 2005

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
Centro Universitário Feevale, RS, Brasil

Cervi, Gustavo Henrique
Plataforma RAD para aplicações locais ou distribuídas / por Gustavo Henrique Cervi – 2005.

113 f. ; 28cm.

Trabalho de conclusão II do curso de Ciência da computação do Centro Universitário Feevale.
"Orientador: Marta Rosecler Bez el Boukhari".
Inclui bibliografia.

1. RAD(Rapid application development) 2. Sistemas operacionais distribuídos(Computação) 3. Métodos orientado a objetos(Computação) 4. Prototipagem de programas(Computação) 5. Linguagem de programação(Computadores) I. Título.

CDU 004.414.32

Bibliotecária responsável: Gina Maria da Gama CRB10/1478

Agradeço à Profa. Marta Rosecler Bez el Boukhari, minha orientadora, que muito ajudou com sua imensurável paciência, sua profunda compreensão e seu insustentável otimismo que muitas vezes levantou a motivação deste trabalho.

Agradeço aos meus pais pela minha existência, pelos dias que passaram e pelos dias que virão, pela motivação, atenção, cuidado, incentivo e também pelos protestos.

Agradeço aos colegas de curso pelas ideías, críticas e sugestões para este trabalho, bem como os bons momentos de vida acadêmica e também pelos maus momentos que, de alguma forma, serviram como aprendizado.

Agradeço a toda comunidade do software livre que suporta, desenvolve, dedica e ensina ciência em troca de esperança, seja por um mundo mais justo ou seja em busca de um bem comum, independente de condições, credo ou cultura.

Agradeço também aos amigos, irmãos e especialmente à minha querida Fátima que muitas vezes abriu mão de sua própria vida para este trabalho, ajudando, motivando e principalmente acreditando.

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. Estas tecnologias já são amplamente trabalhadas, mas possuem 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 dificultando nas aplicações mais simples. Desta forma, este projeto tem como principal objetivo, propor 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 exists an strong demand for distributed systems, the model of WEB development is applied in wide form, even being created for static anchored documents. These technologies are widely worked but it have some limitations when the systems requires a bigger interaction with users. Some models of multi-tier systems or proprietary solutions solves some high-complexity problems but they turn on complex the simplest systems.. This project have as main target, 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

LISTA DE CÓDIGOS-FONTE

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 uma plataforma RAD que 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 através de chamada remota de procedimento.

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 e desenvolver uma plataforma para desenvolvimento e execução de aplicações locais ou distribuídas. Os objetivos específicos estão divididos em três etapas, a serem alcançadas, que serão apresentadas a seguir:

- **Linguagem:** 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.
- **Plataforma:** Desenvolver uma plataforma que dê suporte ao desenvolvimento e execução de aplicações locais e distribuídas de forma rápida.

No primeiro capítulo deste trabalho serão apresentados as justificativas. No segundo capítulo os conceitos de redes e formas de comunicações entre computadores e aplicações, bem como o modelo de padronização e a sua aplicação são apresentados. Seguindo, no capítulo três, serão mostrados alguns pontos sobre sistemas distribuídos como introdução, conceitos, arquitetura, modelos e comunicação. No quarto capítulo apresenta-se um resumo abrangente sobre uma linguagem interpretada, com suas características e formas de utilização. Pode ser estudado no quinto capítulo uma biblioteca de componentes gráficos para formar a parte visual do trabalho. Nos dois capítulos seguintes, demonstra-se o desenvolvimento da plataforma, com suas ferramentas e sua aplicação em, acompanhado de um exemplo. Logo são apresentadas as considerações finais e propostos trabalhos futuros para a continuidade desta pesquisa.

1. 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 possa 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 da metodologia que este trabalho propõe.

2. 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.

2.1. 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:

2.1.1. 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.

2.1.2. 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.

2.1.3. 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.

2.1.4. 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.

2.1.5. 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.

2.1.6. 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].

2.1.7. 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.

2.2. (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)

2.2.1. 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*.

2.2.2. 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.

2.2.3. 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:

4 *American Standard Code for Information Interchange*

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

6 Também conhecido como *Reply*.

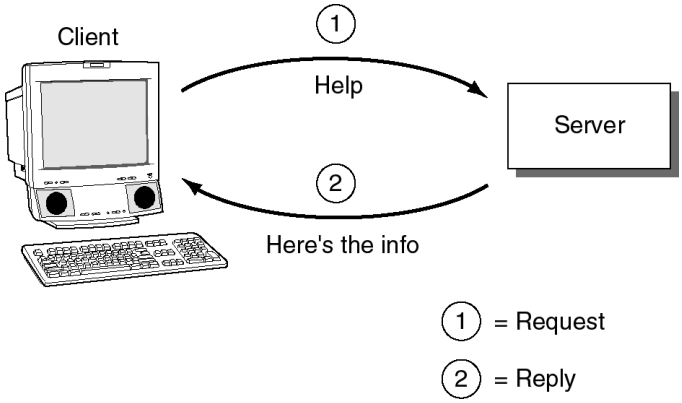

Ilustração 1: Modelo Cliente/Servidor (GALLI, 2000. p.13)

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:

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

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

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.

2.3. 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:

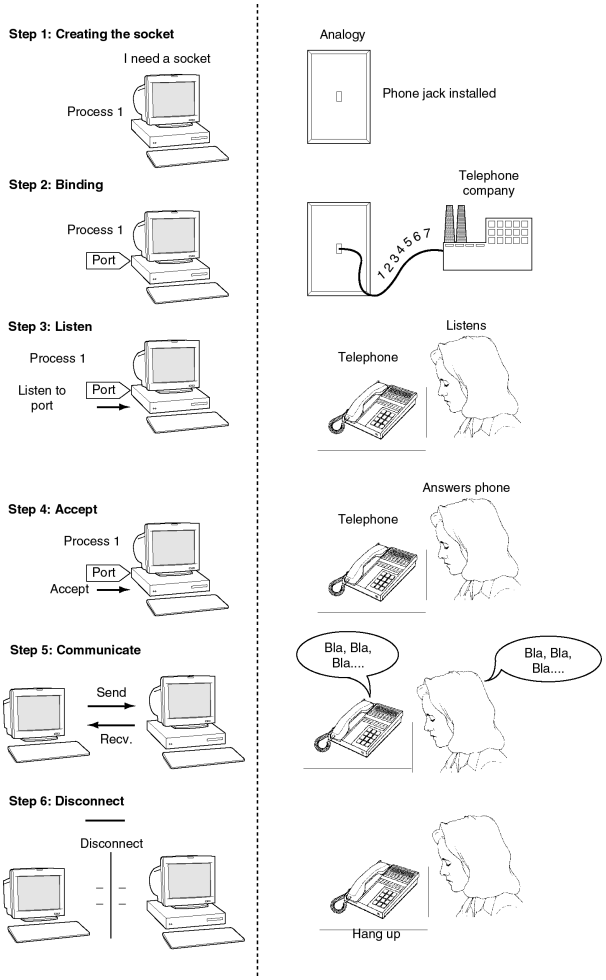

Ilustração 2: Analogia dos Sockets (GALLI, 2000. p.66)

2.3.1. 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 mesma forma que uma função OPEN executa.

2.3.2. 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.

2.3.3. *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.

2.3.4. *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*[8] 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.

8 *Fork*: significa "dividir em dois", método utilizado para dividir um procedimento em dois processos.

2.3.5. *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).

2.3.6. 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.*

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define PORTNO = 5000

int main(int argc, char *argv[])
{
     int sockfd, newsockfd, clilen, n;
     char buffer[256];
     struct sockaddr_in serv_addr, cli_addr;

     sockfd = socket(AF_INET, SOCK_STREAM, 0);
     bzero((char *) &serv_addr, sizeof(serv_addr));

     serv_addr.sin_family = AF_INET;
     serv_addr.sin_addr.s_addr = INADDR_ANY;
     serv_addr.sin_port = htons(PORTNO);

     bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
     listen(sockfd,5);

     clilen = sizeof(cli_addr);
     newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
     bzero(buffer,256);

     n = read(newsockfd,buffer,255);
     printf("Mensagem: %s\n",buffer);
     n = write(newsockfd, "Mensagem recebida!",18);

     return 0;
}
```

Código 1: Exemplo de uso de Sockets

3. SISTEMAS DISTRIBUÍDOS

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

3.1. Introdução

Sistemas ditos distribuídos são definidos de diversas formas, por diversos autores, conforme segue:

> **"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 citados 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[9]:** Uma infinidade de aplicações para internet utilizam o modelo distribuído. Como exemplo, os servidores de resolução de nome (DNS[10]) ou servidores de email (SMTP[11]). 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.

3.2. 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.

---

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

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

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

3.3. Problemática

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

3.3.1. 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 uma 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).

3.3.2. Escalabilidade

Um sistema escalável é aquele que pode ser modificado para acomodar alterações na quantidade de usuários, recursos e entidades computacionais. Pode ser medido 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[12]. (WIKIPEDIA, 2005)

3.3.3. 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 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:

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

12 Tipo de sistema distribuído, porém fortemente acoplado.

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.

3.3.4. 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).

3.4. Arquiteturas

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

3.4.1. 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[13] 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)

13 Trabalho realizado em 2005.

3.4.2. 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)

3.5. 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.
- **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.

3.6. Comunicação

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

3.6.1. 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)

```
4  send (b, msg);
5  receive (a, msg);
```

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

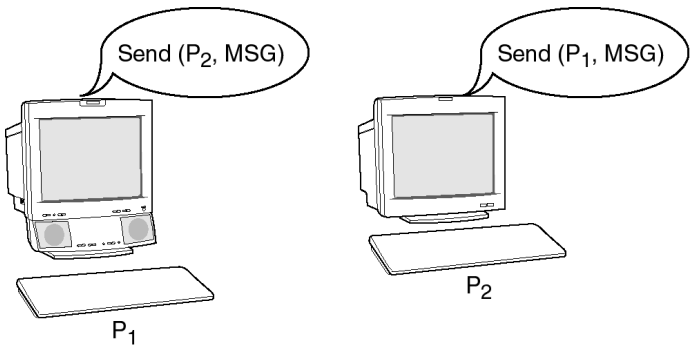

Ilustração 3: Troca de mensagens (GALLI, 2000. p.59)

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)

3.6.2. 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.

3.6.3. 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.

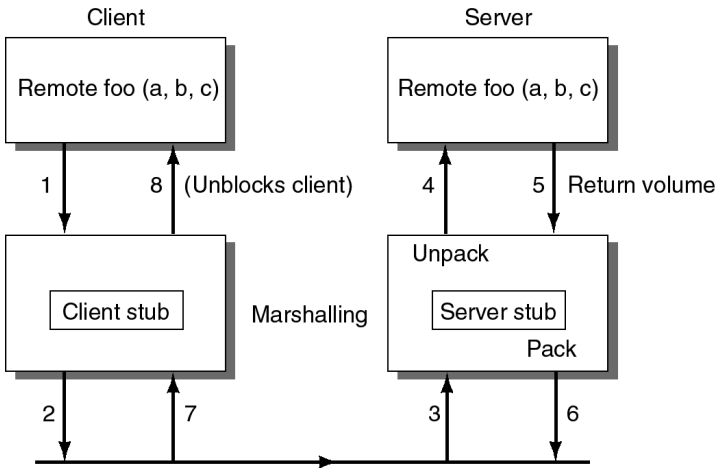

Ilustração 4: RPC - (GALLI, 2000. p.73)

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)

3.6.4. 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)

3.7. Exemplos de objetos distribuídos

Serão descritos dois exemplos de objetos distribuídos, ambos representados pela linguagem Python.

3.7.1. 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.

Dois exemplos de servidor e cliente, respectivamente, são apresentados:

Servidor:

```
34 import Pyro.core
35 import Pyro.naming
36
37 class JokeGen(Pyro.core.ObjBase):
38     def joke(self, name):
39         return "Sorry "+name+", I don't know any jokes."
40
41 daemon=Pyro.core.Daemon()
42 ns=Pyro.naming.NameServerLocator().getNS()
43 daemon.useNameServer(ns)
44 uri=daemon.connect(JokeGen(),"jokegen")
45 daemon.requestLoop()
```

Código 2: Exemplo Servidor PYRO

Cliente:

```
46 import Pyro.core
47
48 # finds object automatically if you're running the Name Server.
49 jokes = Pyro.core.getProxyForURI("PYRONAME://jokegen")
50
51 print jokes.joke("Irmen")
```

Código 3: Exemplo Cliente PYRO

3.7.2. 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[14], 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[15]. 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.

Um exemplo de como um cliente deve formatar uma mensagem SOAP requerendo informações de um produto de uma empresa fictícia, é demonstrado:

```
52 <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
53    <soap:Body>
54      <getProductDetails xmlns="http://warehouse.example.com/ws">
55        <productID>827635</productID>
56      </getProductDetails>
57    </soap:Body>
58  </soap:Envelope>
```

Código 4: Exemplo SOAP XML Requisição

Um exemplo de como um servidor de uma empresa fictícia deve responder à mensagem de requisição de informações do produto, conforme segue:

14 Trabalho escrito em 2005.

15 *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>
```

Código 5: Exemplo SOAP XML Resposta

Um exemplo em Python de utilização de SOAP pode ser observado a seguir:

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()
```

Código 6: Exemplo SOAP Servidor

4. 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, 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.

4.1. Caracterísicas gerais

No decorrer deste capítulo, 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*.

4.1.1. 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.

4.1.2. 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++.

4.1.3. Ferramentas e extensões

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

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.

4.1.4. 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.

4.2. 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 apresentados alguns pontos semânticos e sintáticos da linguagem segundo Hoffmann (2004).

4.2.1. *Keywords* e Identificadores

Existem poucas palavras reservadas em Python, são elas:

```
87 and       del       for       is        raise
88 assert    elif      from      lambda    return
89 break     else      global    not       try
90 class     except    if        or        while
91 continue  exec      import    pass      yield
92 def       finally   in        print
```

Código 7: Palavras reservadas

Os identificadores têm a seguinte gramática:

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

Código 8: Gramática para identificadores

4.2.2. Literais

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

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

Código 9: Literais

4.2.3. 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:

```
Decimal inteiro: 1234, 1234567890546378940L (ou l)
Octal inteiro: 0177, 017777777777777777L (começa com um 0)
Hex inteiro: 0xFF, 0XFFFFFFFFFFFFFFFFL (começa com 0x ou 0X)
Long Inteiro (precisão ilimitada): 1234567890123456L
Float (precisão double): 3.14e-10, .001, 10., 1E3
Complex: 1J, 2+3J, 4+5j ('+' separa as partes real e imaginária)
```

Código 10: Tipos numéricos

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

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

Código 11: Tipos sequenciais

Seqüências podem ser quebradas utilizando a sintaxe:

```
[índice-início : índice  [ : passo]]
```

Código 12: Exemplo de quebra de seqüência

As formas de utilização são assim exemplificadas:

```
114a = (0,1,2,3,4,5,6,7)
115a[3] == 3
116a[-1] == 7
117a[2:4] == (2, 3)
118a[1:] == (1, 2, 3, 4, 5, 6, 7)
119a[:3] == (0, 1, 2)
120a[:] == (0,1,2,3,4,5,6,7) # copia a seqüência.
121a[::2] == (0, 2, 4, 6) # de dois em dois.
122a[::-1] = (7, 6, 5, 4, 3 , 2, 1, 0) # Ordem reversa.
```

Código 13: Exemplo de seqüência

Dicionários são utilizados para dar valor a índices em partes de código e aceitam qualquer tipo de formato.

```
123{} {1 : 'um'} {1 : 'um', 'próximo': 'dois'}
```

Código 14: Exemplo de dicionário

4.2.4. Operadores e ordem de resolução

Na 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

4.2.5. Controle de fluxo

As estruturas de controle de fluxo e seus resultados podem ser observadas na próxima tabela.

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

4.2.6. 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

4.3. *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.

4.3.1. 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:

```
124import threading
125
126var = 1
127
128class testeThread ( threading.Thread ):
129  def run ( self ):
130    global var
131    print 'Thread ' + str ( var ) + ' falando.'
132    print 'Olá e tchau.'
133    var = var + 1
134
135for x in xrange ( 20 ):
136  testeThread().start()
```

Código 15: Exemplo de threading

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

```
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'
```

Código 16: Exemplo de threading com lock

4.4. 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. É apresentado um exemplo de programa Python que cria novos processos enquanto a tecla “q” não é pressionada.

```
26 import os
27 def filho():
28   print 'FILHO: Olah!', os.getpid()
29   os._exit(0)   # volta para o loop do processo-pai
30
31
32 def pai():
33   while 1:
34     novoPid = os.fork()
35     if novoPid == 0:
36       filho()
37     else:
38       print 'PAI: Olah', os.getpid(), novoPid
39     if raw_input() == 'q': break
40
41 pai()
```

Código 17: Python - Fork

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)

4.5. 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)

4.5.1. Pickling

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

```
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)
```

Código 18: Pickle - Serializando

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 apresentado um exemplo de como recuperar o arquivo serializado:

```
5  import pickle
6  mydb = open('dbase', 'r')
7  table = pickle.load(mydb)
8  print table
```

Código 19: Pickle – Recuperando dados

4.5.2. 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()


9  import shelve
10 mydb = shelve.open('dbase')
11 print mydb['table1']
```

Código 20: Shelve

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

4.5.3. 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.

5. MÓDULO TKINTER

Tkinter é uma biblioteca de componentes para interação com os usuários através de interfaceamento gráfico com o usuário (GUI[16]). 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[17]. É construída através do empacotamento (*wrapping*) da biblioteca multi-plataforma Tk, que é também utilizada em conjunto com outras linguagens interpretadas como TCL[18] e PERL[19].

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.

5.1. 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:

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

17 No ano da publicação de seu livro – 2003.

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

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

```
12 import sys, Tkinter
13 Tkinter.Label(text="Olá!").pack()
14 Tkinter.Button(text="Sair", command=sys.exit).pack()
15 Tkinter.mainloop()
```

Código 21: Tkinter - Exemplo

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.

5.2. 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')*.

5.2.1. 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.*

5.2.2. 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.*

5.2.3. 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:

```
16 import Tkinter
17
18 root = Tkinter.tk()
19 tv = Tkinter.StringVar()
20
21 Tkinter.Label(textvariable=tv).pack()
22 Tkinter.Entry(textvariable=tv).pack()
23
24 tv.set('Olá!')
25
26 Tkinter.Button(text="Sair", command=root.quit).pack()
27
28 Tkinter.mainloop()
29 print tv.get()
```

Código 22: Tkinter - Objeto Variavel

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.

5.3. Widgets

O módulo Tkinter provê uma grande quantidade de *widgets* que cobrem a maioria das necessidades de uma aplicação GUI simples. Será visto, na seqüência, de maneira sucinta,

alguns objetos comuns, seus atributos e métodos. Esta referência 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.

5.3.1. 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.*

5.3.2. 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:

```
30 e.delete(0, END) # apaga o conteúdo
31 e.insert(END, algumaString) # insere algumaString no campo
32 algumaString = e.get() # atribui à algumaString o conteúdo
```

Código 23: Exemplo de métodos do Widget Entry

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

5.3.3. Label

Este objeto cria um rótulo para um texto qualquer. Não pode ser mudado pelo usuário, mas pode ser feito pelo programa. Pode 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.

6. A PLATAFORMA

A plataforma proposta neste trabalho utiliza as tecnologias estudadas anteriormente (Python, Tkinter e outras bibliotecas). Algumas funcionalidades são implementadas para facilitar o desenvolvimento e o tornar mais rápido, mas a base de tudo está na própria linguagem de programação Python e suas bibliotecas.

É importante observar que esta plataforma está sendo desenvolvida para que outras aplicações sejam produzidas e executadas nela, isto pode ser um pouco confuso nos capítulos seguintes, por este motivo é desejável que fique claro que onde se citar o "usuário" se tratará do usuário do sistema gerado pela plataforma e onde se citar "desenvolvedor" será o programador que utilizará as ferramentas e a plataforma para desenvolver seus sistemas. A seguir, uma ilustração da organização desta característica.

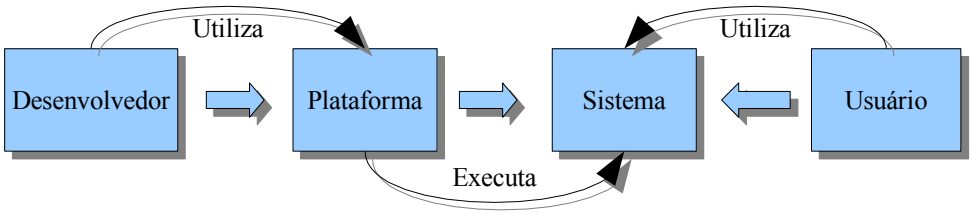

Ilustração 5: Diagrama Usuário/Desenvolvedor

6.1. Introdução à plataforma

A plataforma, objeto deste trabalho, providencia o suporte para o desenvolvimento e execução de aplicações locais ou distribuídas, sendo direcionada tanto ao desenvolvedor quanto ao usuário (de forma transparente). O desenvolvedor que optar pelo uso desta, poderá usufruir

de uma linguagem de programação, ferramentas de desenvolvimento WYSIWYG[20] e algumas bibliotecas adicionais que serão estudadas nos sub-capítulos seguintes. Algumas características da plataforma e suas ferramentas serão apresentadas a seguir:

Ferramentas: Divididas em duas: editor de código fonte e editor de interface gráfica. Através do editor de código fonte, é possível editar programas, com ajuda de facilitadores que auxiliam a escrita do código. O editor de interfaces é parte fundamental para o desenvolvimento rápido de aplicações locais "*desktop*" e utiliza a tecnologia WYSIWYG onde o desenvolvedor pode ver como ficará a janela sem precisar executar o código.

Bibliotecas: Divididas em Interface Gráfica, Cliente e Servidor. As bibliotecas para execução remota de procedimentos são peça-chave para a plataforma, suprindo as necessidades de comunicação entre as partes. O módulo de suporte a execução remota, é semelhante aos estudados no capítulo 3.7, porém uma nova alternativa, mais simples foi desenvolvido neste trabalho como opção para o desenvolvedor.

Linguagem: A base da plataforma, que é a linguagem Python, fornece o interpretador e a API-base necessária para o desenvolvimento de aplicações.

Suporte a internacionalização: As ferramentas utilizam uma função (*translate*) que retira as mensagens do código, possibilitando a tradução de suas interfaces e mensagens para outras línguas sem edição do código-fonte das ferramentas, as mensagens ficam em arquivos-texto, separados do código, e poderão futuramente ser editados através de ferramentas automatizadas. O código-fonte está escrito com comentários e identificadores na língua inglesa visando uma futura publicação em repositórios internacionais de desenvolvedores Python.

A seguir, um pequeno diagrama sobre a plataforma e seus componentes.

20 *What You See Is What You Get* – Sigla que significa: "O que você vê é o que você terá" (Tradução nossa)

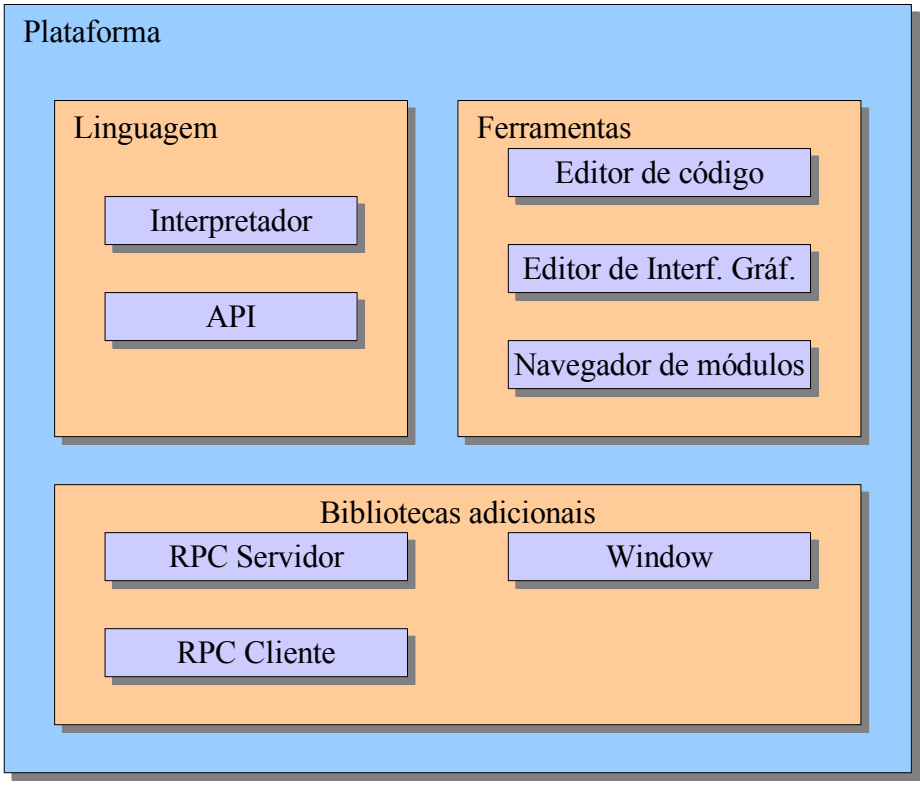

Ilustração 6: Plataforma

Como pode-se observar, estão contidas na plataforma três macro-temas:

- Linguagem e API (disponíveis na distribuição Python padrão)
- Ferramentas (desenvolvidas neste trabalho)
- Bibliotecas adicionais (desenvolvidas neste trabalho)

6.2. Aplicações

As aplicações da plataforma são as mais diversas que a linguagem Python pode suprir, como visto no capítulo 4.1.2. Respeitando o objetivo principal deste trabalho, será dado um foco para aplicações locais ou distribuídas para *desktop*. A seguir serão vistos dois diagramas de como a plataforma se aplica para estas duas modalidades. No primeiro, uma pequena aplicação de gerenciamento de usuários em um ambiente Unix, no seguinte, um exemplo genérico de aplicação distribuída.

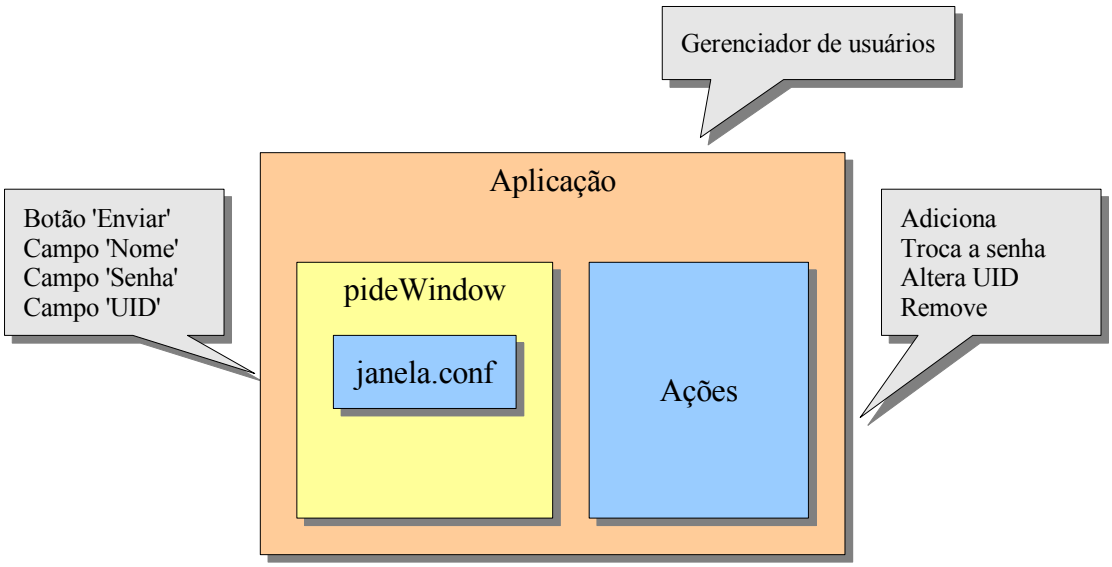

Ilustração 7: Diagrama-Exemplo de aplicação local

Neste exemplo, uma simples aplicação é demonstrada utilizando a plataforma. O exemplo é de um gerenciador de usuários que possui uma janela e uma classe contendo as ações disparadas pelo usuário na janela. As ações podem ser: adicionar, trocar senha, alterar UID ou remover usuário. A classe pideWindow é responsável por montar a interface contida em "janela.conf". O objeto "Ações" possui os métodos e atributos referentes aos eventos registrados na interface gráfica "janela.conf".

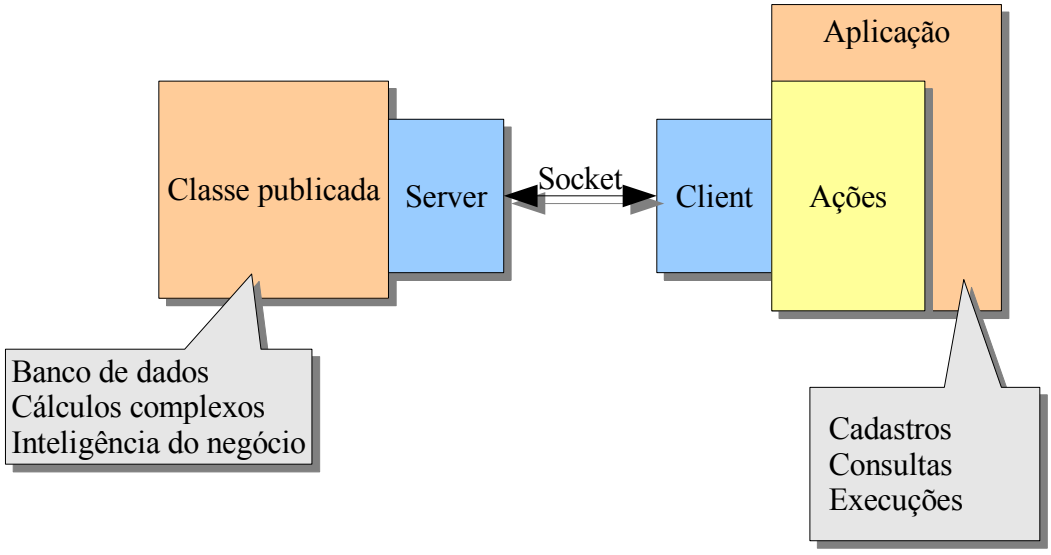

Ilustração 8: Diagrama-Exemplo de aplicação distribuída

Neste exemplo, uma aplicação é demonstrada de forma que existe uma classe publicada no servidor e uma aplicação cliente que a utiliza. A classe publicada pode conter métodos e atributos referentes a: banco de dados, cálculos complexos ou inteligência do

negócio. No lado cliente, pode-se ver uma aplicação/apresentação que pode conter cadastros, consultas e execuções diversas.

Com estes dois exemplos de aplicação, é possível ter uma idéia de como a construção da mesma é feita e como se dá a aplicação em termos de execução e desenvolvimento.

6.3. Linguagem e API

A linguagem adotada para a plataforma é a linguagem Python. Como estudado anteriormente, Python possui características que possibilitam o desenvolvimento de aplicações locais e distribuídas, aliando alguns componentes como a biblioteca Tkinter, ela se torna uma linguagem poderosa para aplicações do tipo Desktop e pode ser distribuída através de algumas bibliotecas que serão vistas mais adiante neste trabalho.

A plataforma proposta está baseada sobre os pilares da linguagem Python, sendo desenvolvida nela, para ela e sendo executada na mesma.

6.4. Persistência

A persistência dos dados referentes à interface gráfica e parâmetros do sistema é feita em arquivos-texto formatados, segundo o estilo da norma RFC-822 (FAQS.ORG, 1982), que define as características para mensagens em texto para a internet. Este padrão é utilizado a partir da biblioteca *ConfigParser* disponível na linguagem Python.

A classe *ConfigParser* funciona recebendo um arquivo como parâmetro, lendo suas configurações e retornando um objeto para a consulta e alteração. Este módulo ainda possibilita que o objeto modificado seja salvo no mesmo formato. (PYTHON.ORG, 2005)

A organização da classe *ConfigParser* se dá em três níveis: Seções, Opções e Valores. Cada arquivo (objeto) pode ter várias seções, que por sua vez podem possuir várias opções que remetem a valores. A seguir será visto um exemplo de arquivo:

```
33 [My Section]
34 foodir: %(dir)s/whatever
35 dir=frob
```

Código 24: ConfigParser

6.5. Ferramentas

Para que a plataforma seja mais produtiva, é necessária a utilização de ferramentas que auxiliam o desenvolvimento de aplicações. Este trabalho desenvolve dois tipos de ferramentas: um editor de código-fonte e um editor de telas.

O editor de código-fonte se trata de uma ferramenta que dispensa uma explicação aprofundada de sua utilização, uma vez que funciona como a maioria dos editores de código que existem no mercado. O capítulo que segue irá descrever como ela é desenvolvida.

O editor de telas é um pouco mais complexo e utiliza alguns recursos especiais da linguagem Python para o seu funcionamento. Funciona como um editor gráfico onde é possível adicionar objetos a um *frame* e configurá-los de maneira visual, sem a necessidade direta de edição do código que gera a janela.

A seguir, serão vistos em dois sub-capítulos a construção dos editores de código e de telas, respectivamente.

6.5.1. Editor de código-fonte

O editor de código-fonte utiliza um código principal e algumas classes auxiliares. Para melhor compreensão do texto a seguir, é interessante observar o layout básico da tela que pode ser acompanhado no diagrama:

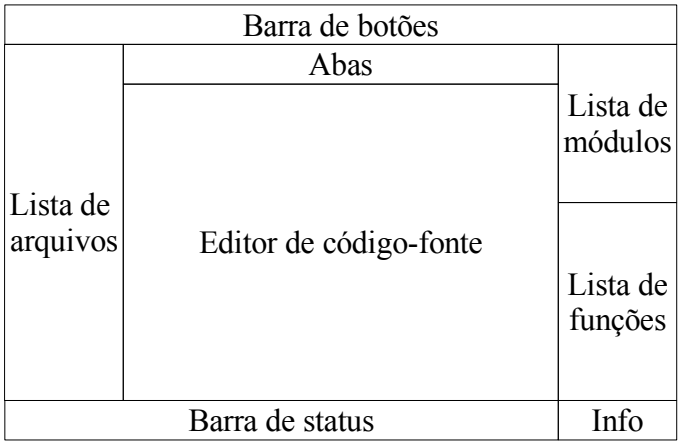

Ilustração 9: Layout do editor de código-fonte

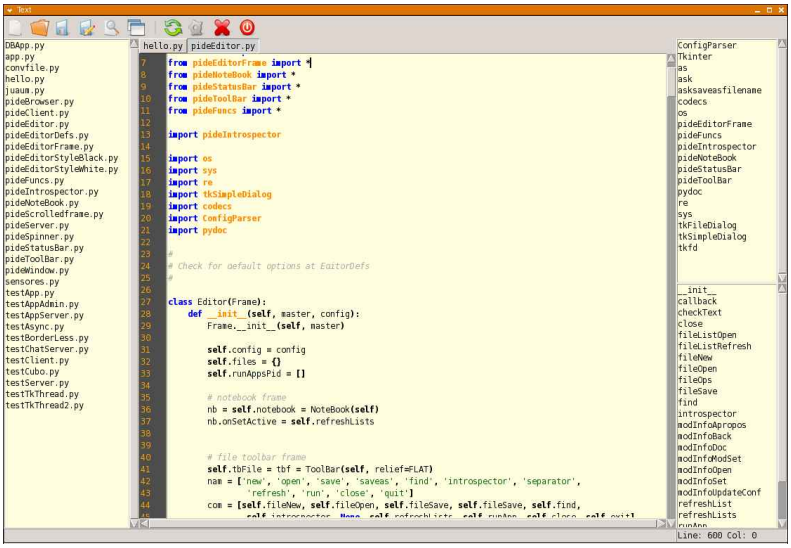

Ilustração 10: Captura de tela do editor de código-fonte

Os itens descritos no *layout*, são os seguintes:

Barra de botões: é o primeiro ítem, acima, na janela do editor, onde ficam os botões que dão acesso aos métodos para abrir, fechar, salvar, executar e etc.

Ilustração 11: Barra de botões

Lista de arquivos: é exibida a lista de arquivos editáveis do diretório corrente, pode ser ocultada para aumentar a área de edição.

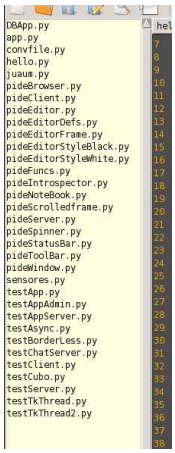

Ilustração 12: Lista de arquivos

Abas e Editor de código-fonte: fazem parte da classe mais extensa do editor que é onde o código é exibido e editado.

Ilustração 13: Abas

Lista de módulos: contém a lista dos módulos inseridos no arquivo que está sendo editado. Quando algum ítem da lista recebe duplo-clique, uma janela é aberta com o "Navegador de Módulos", onde são mostrados os componentes que integram o módulo, servindo de "ajuda" na hora de editar o código. A janela de informações sobre o módulo utiliza a introspecção (reflexão computacional) embarcada no módulo "pydoc" (padrão na distribuição dos módulos). Através do método "pydoc.locate(modulo)", uma instância do módulo é atribuída a uma variável que é inspecionada pelo método interno "dir()" . Este método retorna uma lista contendo os objetos que constituem o módulo. Por padrão, as classes Python possuem um atributo especial chamado "__doc__" que contém uma pequena documentação sobre o objeto. Estes dados são mostrados em um campo de texto que apresenta as informações pertinentes ao objeto. Pode-se visualizar a lista dos módulos e a janela com os dados mostrados:

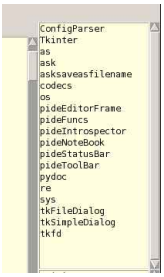

Ilustração 14: Lista de módulos

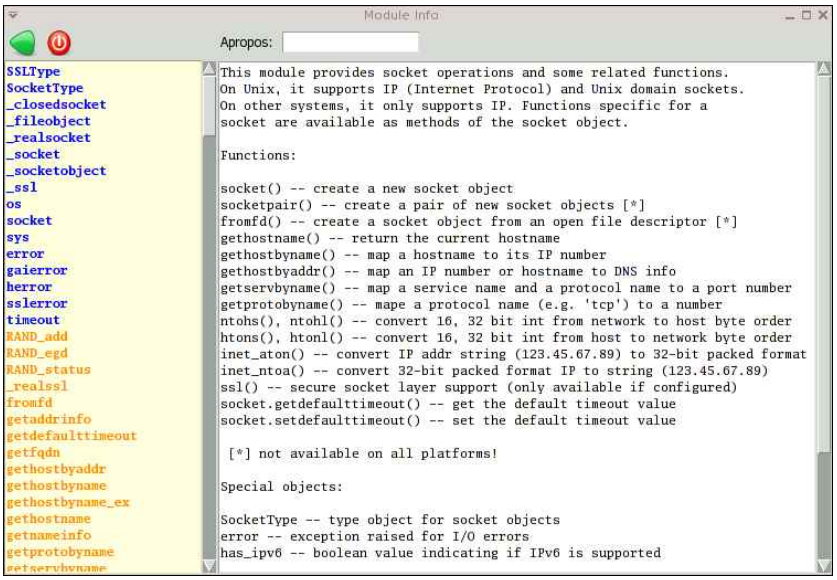

Ilustração 15: Informações sobre o módulo (introspecção)

Lista de funções: lista das funções/métodos no arquivo que está sendo editado. Quando algum ítem da lista recebe duplo-clique, o editor rola a janela do código em edição (ativa) para o ponto onde a função ou método é definido. Funciona como um navegador interno do código, tornando a busca mais rápida.

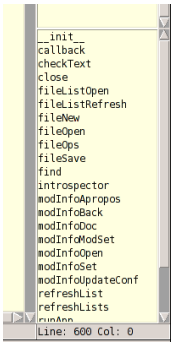

Ilustração 16: Lista de funções

Editor de código-fonte: se trata de uma classe com métodos e atributos separados da janela principal, é lá que se dão os processos referentes à edição do código. Esta classe, além de editar o código, ainda possibilita alguns facilitadores:

- **Realce de sintaxe**: realça as palavras reservadas e outros identificadores com cores diferentes para facilitar a leitura e compreensão do código.
- **Complemento de código**: quando alguma palavra está sendo digitada e o editor já possui uma ou mais palavras semelhantes a ela em sua memória, ele abre uma pequena lista para que o desenvolvedor possa escolher a palavra ao invés de digitá-la até o final.
- **Suporte a tema**: o desenvolvedor pode definir cores, fontes e atributos de fontes a seu gosto e guardar em arquivos, os arquivos contendo as definições são em formato texto e seguem a RFC-822.

6.5.1.1. Estrutura do editor

A sua estrutura interna de classes pode ser vista no seguinte diagrama:

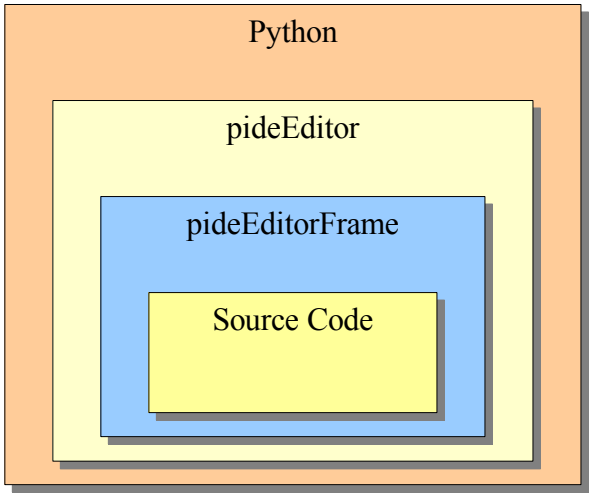

Ilustração 17: Estrutura interna do editor

A classe *pideEditor* é a classe principal, suas funções são: criar a janela do editor, disponibilizar os métodos para gerenciar os arquivos e as ações nas listas de funções e de módulos. A classe *pideEditorFrame* contém o editor de textos que deriva do componente *Tkinter.Text* e implementa alguns métodos que auxiliam a edição do código. O "*source code*", é o código fonte que está sendo editado, se trata de um arquivo texto.

A classe *pideEditorFrame*, implementa o componente *Tkinter.Frame* e, dentro dele, o componente *Tkinter.Text*. Este componente (*Text*) fornece um simples editor de textos com poucas funcionalidades mas com alguns recursos que facilitam a expansão dele. Um destes recursos é muito utilizado na classe e serve para realçar o texto com "*tags*" pré-definidas.

6.5.1.2. Realce de sintaxe

Para que a edição do texto se torne mais ágil e compreensiva, o editor utiliza um recurso chamado "Realce de sintaxe". Este recurso possibilita que diferentes tipos de identificadores no texto do código-fonte sejam destacados de forma diferente, fazendo com que o código se torne mais "legível" ao desenvolvedor. O processo funciona da seguinte forma: utilizando um simples *parser* que funciona como uma máquina de Turing, lendo caractere por caractere, o *token* é montado e reconhecido seguindo a gramática de identificadores da linguagem Python, que foi vista no capítulo correspondente, e a sua pequena lista de palavras reservadas. Existem quatro listas definidas para a identificação do *token*: palavras reservadas, operadores, funções e módulos. Uma vez que o *token* foi reconhecido, a parte do código onde ele se encontra é realçada com a *tag* correspondente (*operator*, *module*, *method*, etc..). Cada *tag*

possui uma configuração que define cor, tipo de letra, tamanho e etc.. Este procedimento é executado somente na linha que está sendo editada, e ocorre a cada toque ou clique de mouse. Na abertura do arquivo em disco, o procedimento é executado em todo o conteúdo para gerar a primeira leitura. Esta varredura global só é executada novamente se o método "*refresh*" for acionado, seja internamente, pelo conjunto de teclas "*Control*+r" ou pelo botão "*Refresh*" na barra de botões, no topo da janela principal.

6.5.1.3. Complemento de código

Se trata de um recurso que elimina boa parte da digitação de palavras longas no código-fonte e acaba por facilitar a edição. Este processo se dá da seguinte forma: uma vez que a palavra, que está sendo digitada, atinge o tamanho estipulado (padrão de 3 letras), o processo principal procura no *buffer* do arquivo aberto pelas palavras que iniciam com as mesmas letras da palavra digitada, se encontradas sentenças compatíveis, uma pequena lista é inserida na janela, na posição do cursor, disponibilizando as opções para serem escolhidas em vez de digitadas até o fim. A ilustração a seguir demonstra o seu funcionamento.

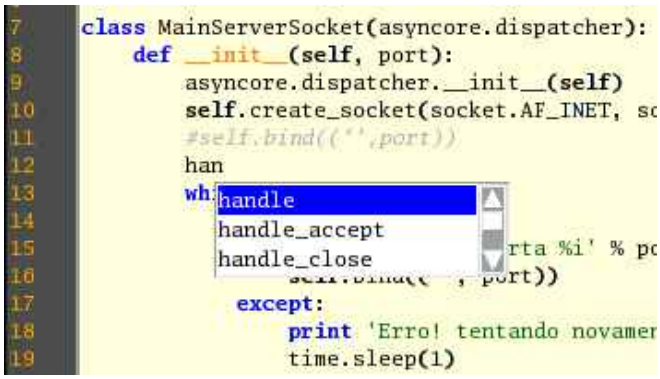

Ilustração 18: Exemplo de complemento de código

6.5.2. Editor de interface gráfica

O editor de interface gráfica auxilia o desenho das telas, disponibilizando uma ferramenta visual que permite que o desenvolvedor monte a janela visualizando como ela ficará quando executada. O *layout* do mesmo é demonstrada:

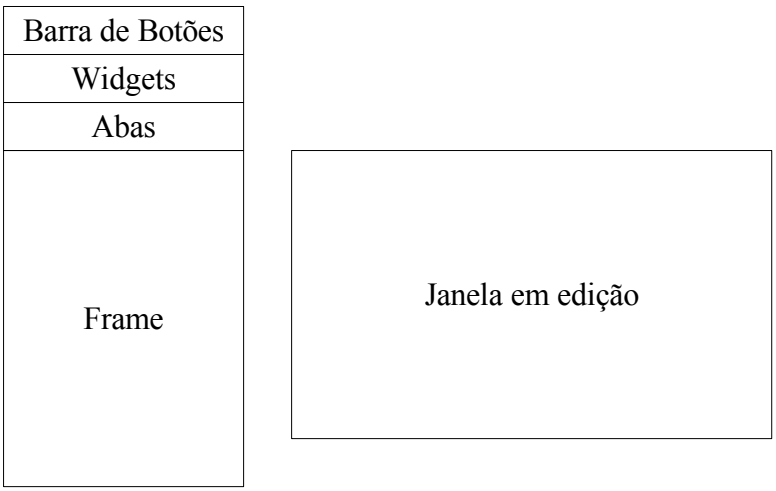

Ilustração 19: Layout do editor de janelas

O editor está escrito em duas classes predominantes: *pideIntrospector* e *pideWindow*. A primeira é a responsável por desenhar a tela do editor e disponibilizar os métodos necessários para a edição da janela bem como gerenciamento do arquivo (abrir, salvar, fechar). A classe *pideWindow* é responsável por "montar" a janela em função dos comandos recebidos pela editor, é instanciada cada vez que um arquivo é aberto e possui um dicionário que contém todos os componentes da janela. Os dados de cada componente são obtidos diretamente do componente, utilizando uma característica da biblioteca Tkinter que implementa um método chamado "config" que retorna todos os parâmetros de um *widget*.

A utilização do editor é simples, uma vez que o desenvolvedor criar um arquivo novo, ele pode inserir *widgets* e configura-los utilizando a aba "Config" que disponibiliza os parâmetros de cada *widget* como cor, tamanho, modo de exibição, ação e etc.

Existe cinco abas no editor: *Widgets, Info, Config, Binds* e *Extras*. A seguir a função de cada aba.

- **Widgets**: possui a lista dos *widgets* da janela, com a opção de alterar o nome de cada um.
- **Info**: mostra uma pequena informação sobre o *widget* (tamanho, posição, classe e etc..)
- **Config**: disponibiliza uma lista de parâmetros para que o usuário altere as características e o comportamento do *widget*.
- **Binds**: são os eventos (teclado, mouse) que *widget* "escutará"; que é a ação (função, método) que ele executará cada vez que o mesmo foi acionado.

- **Extras**: aba reservada para uso futuro.

6.6. Módulo de interface gráfica

Uma das características mais expressivas deste trabalho é a possibilidade de se "desenhar" janelas utilizando o editor de janelas e executar a aplicação sem precisar escrever o código referente ao *layout* da janela. Para isso, o Editor de Janelas foi desenvolvido, mas no momento da execução da aplicação, é necessário, de alguma forma, ler o conteúdo do arquivo gravado anteriormente e montar a tela de acordo com o que foi desenhado. Esta é a função da biblioteca montadora de janelas.

A função desta biblioteca é, basicamente, ler o arquivo gerado pelo Editor de Janelas e montar a janela conforme foi desenhada, observando as chamadas de eventos e outras características.

Para que a janela desenhada no editor de janelas seja disponibilizada na aplicação, a biblioteca montadora de janelas deve ser instanciada e o arquivo contendo a configuração desta janela deve alimentar o seu construtor, que apresentará a janela pronta para interagir com usuário.

Para ilustrar a utilização da biblioteca, segue uma pequena demonstração de como ela pode ser usada:

```
1  from pideWindow import *
2  from Tkinter import *
3  import sys
4
5  class Funcs:
6      def teste(self):
7          print 'Teste!'
8
9      def quit(self):
10         print 'Tchau!!'
11         sys.exit()
12
13
14 f = Funcs()
15 w = Wind('tela.conf', funcs=f)
16
17 mainloop()
```

Código 25: Exemplo de utilização da biblioteca montadora de janelas

- Linha 1: o módulo *pideWindow* é inserido no código

- Linhas 5 até 11: uma classe é desenvolvida contendo os métodos e atributos que serão utilizados na aplicação. No exemplo, existem dois métodos: “teste” e “quit” que podem ser executados a partir de um evento gerado pelo usuário na interface gráfica, seja por meio de um botão ou por meio de uma combinação de teclas, ambos definidos no editor de janelas.
- Linha 14: declara a instância da classe com os métodos e a linha 15 declara a instância da janela. Esta classe (*pideWindow*) recebe como parâmetros, em seu construtor, o arquivo contendo as configurações da janela, que pode ser uma *string* com o nome do arquivo ou um objeto tipo File, e a instância da classe que será utilizada na execução da aplicação.
- Linha 17: é executado o *loop* principal da biblioteca Tkinter que é responsável pelo gerenciamento dos componentes gráficos e seus eventos.

Como visto, em quatro passos uma aplicação pode ser desenvolvida e executada. A classe *pideWindow* ainda possui outros métodos que servem para a interação com os componentes da janela, uma vez que o programa pode alterar o comportamento dos componentes quando necessário. Para isso, pode-se utilizar o método “get()” que recebe como parâmetro o nome do objeto (*widget*) e retorna a instância do mesmo, que pode ser alterado do jeito que for necessário. Outros métodos estão disponíveis na classe e não serão documentados neste trabalho por motivos de espaço, mas podem ser verificados junto ao código-fonte da biblioteca que possui a devida documentação.

6.6.1. Construção

A Construção do editor é dividida em duas classes: *pideWindow* e *pideIntrospector*. A primeira é responsável pela criação da janela e composição dos itens com seus devidos parâmetros. A segunda interage com o usuário, criando e manipulando a janela em edição e também recebendo e executando os comandos.

6.6.1.1. Classe *pideWindow*

A classe *pideWindow* possui os métodos e os atributos necessários para o desenho da janela com seus componentes. O seu funcionamento se dá da seguinte forma: uma vez que o

objeto é instanciado, ele recebe por parâmetro um objeto *file* ou *ConfigParser*. Se for do primeiro tipo, o construtor da classe abrirá o arquivo com os dados da janela, senão utilizará o objeto *ConfigParser* que receber. No momento da inicialização, é executada uma varredura pelo objeto com os parâmetros e para cada ítem, o método de criação (*add*) da classe *Window* é executado, instanciando a classe correspondente ao objeto (*widget*) e alimentando seus atributos com os parâmetros salvos no arquivo.

6.6.1.2. Classe *pideIntrospector*

Esta classe é responsável pelo suporte ao desenvolvimento da interface, gerando uma janela que contém os componentes, botões de acesso e as abas onde são configurados os *widgets* adicionados na janela em edição.

Como a classe *pideWindow* é genérica, ela é utilizada por esta classe para montar a janela no momento em que é editada. Esta classe ainda possui os métodos de persistência para a leitura e gravação dos arquivos contendo as definições dos componentes da janela.

O seu funcionamento é de forma que, uma vez instanciada, cria uma janela contendo vários itens. O conteúdo da classe é, na sua grande maioria, de uso interno, sendo apenas utilizado por ela mesma, em resposta aos eventos gerados pelo usuário.

6.7. Módulo de comunicação e execução remota

Para que os sistemas desenvolvidos e suportados pela plataforma possam ter características distribuídas, o desenvolvedor pode optar por utilizar os recursos da própria linguagem Python, vistos no capítulo 3.7 ou então desenvolver sua aplicação utilizando o método que será visto na continuação.

O desenvolvimento de aplicações distribuídas, utilizando esta plataforma, pode ser feito com o auxílio de dois módulos: Cliente e Servidor. O primeiro provê o suporte à conexão com o Servidor e a comunicação com ele, o segundo se encarrega de servir uma classe para ser executada.

Os módulos aqui desenvolvidos levarão em conta apenas o fundamental para a execução de aplicações distribuídas, sendo desenvolvida a execução remota de procedimentos e a troca de objetos através da serialização, outros pontos estudados no capítulo 3.3 não serão abordados, mas em alguns casos podem ser importantes, ficando o estudo anterior como referencial teórico para o desenvolvedor que precisar algum destes recursos.

A aplicação destes módulos pode ser feita utilizando os recursos da linguagem Python com, virtualmente, todos os seus recursos que cabem à classe exportada. Para ilustrar, um exemplo de aplicação será visto.

Exemplo de aplicação servidor:

```
18 from pideServer import *
19
20 class Teste:
21     def mais(self, v, vv):
22         return v+vv
23
24 h = Teste()
25 s = pideServer(h)
```

Código 26: Exemplo de aplicação Servidor

- Linha 1: o módulo *pideServer* é importado
- Linhas 3 a 5: a classe que será exportada é declarada
- Linha 7: a classe que será exportada é instanciada
- Linha 8: a classe *pideServer* é instanciada e o servidor é executado

É interessante observar que, neste exemplo, uma classe "Teste" é exportada para uso externo, contendo um método chamado "mais" que não faz nada além de uma simples soma de dois números.

A seguir, um exemplo de aplicação Cliente:

```
26 from pideClient import *
27 c = pideClient()
28
29 print c.send('mais', (11, 22))
30 print c.send('mais', (1, 5))
31
32 c.close()
```

Código 27: Exemplo aplicação Cliente

- Linha 1: o módulo *pideClient* é importado
- Linha 2: a classe *pideClient* é instanciada

- Linha 4: é executado o método “mais” com os argumentos 11 e 22 e o resultado é impresso no terminal
- Linha 5: é executado o método “mais” com os argumentos 1 e 5 e o resultado é impresso no terminal
- Linha 7: o cliente é fechado

O funcionamento dos módulos, como visto anteriormente, se dá de forma que o servidor, quando instanciado, recebe um objeto contendo os métodos e atributos que serão utilizados pelo cliente. O cliente, por sua vez, quando instanciado, conecta no servidor e envia as chamadas, que são executadas no servidor. Logo após a execução, o servidor envia o retorno ao cliente (quando tiver). A execução é transparente para o desenvolvedor, isento da necessidade de trabalhar com componentes de acesso à rede como *sockets* ou suporte a multiprogramação como *threads*.

A troca de informações é feita por meio de envio de objetos instanciados, no caso, uma tupla contendo dados em forma de tipos nativos. O processo se dá conforme pode ser observado no diagrama a seguir:

Ilustração 20: Diagrama de troca de objetos

Um objeto é serializado e depois enviado via *socket*. No outro lado, é recebido e desserializado, retonando a forma original do objeto.

Utilizando este método, a classe do lado servidor é instanciada no servidor e não no cliente, que apenas executa procedimentos remotos e obtém resultados (quando existir). É desejável que fique claro que o servidor não exporta a classe ou sua instância para ser utilizada diretamente no cliente, mas funciona recebendo um comando e retornando um resultado. Quando for citado o termo “exportado” entenda-se como explicado neste parágrafo.

Os dados referentes à porta que será utilizada internamente pelo *socket* ou o *host* que será utilizado pelo cliente para conectar estão definidos nos construtores das classes e utilizam por padrão a porta 32786 e o *host* “localhost”, omitidos nos exemplos anteriores por utilizarem os valores padrão.

Na seqüência, serão vistos os dois módulos que dão suporte a este método.

6.7.1. Servidor

O Servidor é responsável pelo suporte à execução de procedimentos na máquina ou processo que contém a classe que se deseja acessar. Ele está escrito em forma de módulo contendo uma classe. Para utilizá-lo, o desenvolvedor deve criar uma classe, contendo os métodos e atributos que desejar, e em seguida instanciar ela em um identificador. O procedimento seguinte é instanciar o servidor e disparar o loop principal, então o serviço estará disponível para chamadas remotas.

As características do módulo servidor podem ser descritas como um simples servidor para chamadas remotas de procedimentos, onde o mesmo aguarda por conexões e depois por comandos requeridos pelo cliente (já conectado). Pode servir desde simples aplicações como demonstrada no capítulo anterior até execuções mais complexas que utilizem outros tipos de recursos (desde que suportados pela linguagem Python).

A sua construção é feita de forma que vários clientes possam se conectar e fazer chamadas simultâneas e para isso utiliza *threads*, mas a instância da classe é a mesma para todos, isto possibilita que os dados possam ser compartilhados entre os cliente conectados no mesmo servidor, mas pode ser um problema se o desenvolvedor optar por processos separados, com um área de memória separada, para cada cliente. Desta forma é aconselhável que se utilize outro método para o desenvolvimento.

Este módulo utiliza um recurso da linguagem Python que permite que os componentes de uma classe possam ser acessados por meio de uma função interna que recebe como parâmetros o objeto e o nome do atributo (ou método) que se deseja acessar. Esta função interna é a *getattr* e recebe como primeiro parâmetro o objeto "exportado" e segundo parâmetro o método que se deseja acessar. (LUTZ, 2001)

Para ilustrar melhor a sua utilização, será explicado como o servidor é construído. Primeiramente será apresentado o código da classe Servidor, que é dividido em duas partes:

6.7.1.1. Construtor da classe

O construtor, método que é executado na instância da classe, cria o *socket* para que seja possível o acesso ao servidor pela rede. Em seguida, entra na rotina (*loop*) principal. Este *loop* é constituído de duas partes: uma que espera por novas conexões e outra que dispara uma

nova *thread* quando houver uma conexão. O código do construtor é, basicamente, o descrito no capítulo 2.3, onde constam os procedimentos relativos a criação do *socket*, *bind*, *listen*, *accept* e *close*.

Para que o servidor suporte múltiplos clientes, é necessário que o método que gerencia a conexão seja executado de maneira paralela, este procedimento é executado através de um *thread* conforme visto no capítulo 4.3.1.

Uma vez que o servidor é executado, ele entra em um *loop* infinito, pondendo ser parado somente por interrupção do sistema operacional, em função de que o mesmo é planejado para ser executado como serviço (*daemon*).

6.7.1.2. Método manipulador de conexões (*handler*)

Para que a conexão proveniente de um cliente seja corretamente manipulada, um método é disponibilizado contendo alguns procedimentos. Este método é executado dentro de um *thread* que pode estar sendo executado em "paralelo" com vários outros. Ele recebe como parâmetro o *file handler* gerado pelo *socket* que recebeu a conexão.

Uma vez que é executado, o método entra em um *loop* infinito que possui a rotina de receber as chamadas, executá-las e retornar ao cliente. Quando o cliente desconecta do servidor, o *loop* é finalizado e a *thread* é terminada.

O funcionamento desta rotina se dá em função do método interno da linguagem Python chamado *getattr* que possibilita que um método ou atributo de um objeto possa ser executado ou referenciado através do seu nome.

A seguir, o código deste método que ilustra como funciona esta parte.

```
33 while 1:
34   data = conn.recv(1024)
35   if not data: break
36
37   data = pickle.loads(data)
38
39   method, args = data[0], data[1]
40
41   try:
42     res = getattr(self.handlerClass, method)(*args)
43     res = pickle.dumps((method, res))
44   except:
45     res = pickle.dumps((method, None))
46
47   conn.send(res)
```

Código 28: Trecho do código do manipulador de conexões

- Linhas 2 e 3: recebe dados, se não houver retorno, termina o *loop.*
- Linha 5: desserializa a tupla recebida do cliente.
- Linha 7: separa a tupla em duas partes: método a executar e argumentos a passar para este método.
- Linha 10: executa o método com os argumentos; *self.handlerClass* é o atributo que contém a instância da classe com os métodos "exportados".
- Linha 11: serializa o resultado da execução. O resultado é enviado em forma de tupla (serializada), contendo dois elementos: método executado e resultado.
- Linhas 12 e 13: se acontecer algum erro, o servidor envia para o cliente uma tupla contendo o método que ele executou e *None* no lugar do retorno.
- Linha 15: envia ao cliente o resultado.

6.7.2. Cliente

Para que seja possível a conexão com o servidor e o envio de chamadas remotas, este trabalho implementa um módulo de acesso ao servidor chamado "Cliente" que deve ser utilizado em conjunto com o Servidor visto no capítulo anterior.

O módulo Cliente possui uma classe chamada "pideClient" que disponibiliza três métodos: *send, recv* e *close*.

- ***Send***: serve para enviar uma tupla contendo o comando e os argumentos que serão executados no servidor, recebe como parâmetros o nome do método a ser executado e seus argumentos. Retorna o resultado da execução.

- ***Recv***: utilizado pelo método *send* mas pode ser utilizado separadamente, serve para receber o retorno do comando executado no servidor.
- ***Close***: termina a conexão com o servidor, não recebe parâmetros.

O funcionamento deste módulo e sua construção são extremamente resumidos, contendo apenas poucos comandos, dado o alto nível da linguagem base. O processo pode ser descrito em quatro partes: serializar, enviar, receber, desserializar, como visto na Ilustração 20.

Quando uma chamada remota é executada, o Cliente possui a característica de bloquear o processo enquanto não receber a resposta do servidor, isto pode ser um problema se o processamento da chamada no servidor for muito demorado, fazendo com que a aplicação congele até que a resposta retorne. Para isso é recomendada a utilização de uma *thread* na aplicação para procedimentos muito demorados. Como trabalho futuro, um modelo de comunicação assíncrona pode ser desenvolvido utilizando o módulo *asyncore* que dá suporte ao registro de eventos para o recebimento de dados em um *socket*.

7. EXEMPLO DE APLICAÇÃO

Neste capítulo, será visto um exemplo de aplicação desenvolvida utilizando a plataforma e sendo aplicada a ela. Esta serve para demonstrar como a plataforma funciona e como pode ser feito um programa nela.

O sistema que é desenvolvido neste capítulo, se trata de um pequeno simulador de ambiente remoto postulado[21], onde constam variáveis referentes a temperatura, luminosidade e umidade que podem sofrer variações em função de objetos ligados no ambiente como ar-condicionado, lâmpadas ou aquecedor.

A aplicação é dividida em duas partes: Cliente e Servidor. A parte cliente trata uma interface gráfica para informar os dados do servidor ou enviar alterações ao mesmo. O servidor guarda os dados referentes ao ambiente e, quando solicitado, altera o estado dos componentes registrados.

7.1. Descrição do ambiente

O ambiente remoto é uma simulação de uma sala, quarto ou outra parte de uma casa. Possui as seguintes variáveis ambientais:

- Temperatura
- Luminosidade
- Umidade

Estas podem ser influenciadas, para mais ou para menos, em função de objetos que interferem ou não o ambiente. Estes podem ser:

- Ar condicionado – esfria e seca o ambiente

21 As afirmações físicas sobre o ambiente não serão demonstradas, sendo assim, as afirmações referentes ao comportamento do ambiente serão tomadas por verdade sem a sua demonstração ou comprovação.

- Aquecedor – aquece e umedece o ambiente
- Lâmpada – aquece e ilumina o ambiente
- Microcomputador – aquece o ambiente

Cada ítem listado possui seus coeficientes que são calculados em função do tempo em que está ligado. Como exemplo, é considerado que um ar-condicionado pode esfriar 0,01 grau célsius a cada segundo ligado. A física térmica, dinâmica dos fluidos e os valores aplicados se tratam de postulações, ou seja, sem demonstração de sua verdade, uma vez que o programa se trata de um exemplo com foco no desenvolvimento e não nas questões físicas do ambiente.

7.2. Mecânica dos processos

Para que o ambiente possa ser simulado, algumas regras devem ser definidas e aplicadas. Toma-se por verdade que um aparelho ligado é uma fonte de calor constante e linear, aquecendo o ambiente, segundo um fator, até um limite máximo aplicado. As lâmpadas, além de aquecer o ambiente, fornecem luminosidade fixa e acumulativa, em função de quantidade, independente de posição. O ar-condicionado, como exceção à regra, refrigera o ambiente até um limite, também de forma linear e constante. Ainda são tomados por verdade que o ar-condicionado seca o ambiente, de forma linear e constante, até um limite, sendo que o aquecedor, ao seu contrário, umedece o local.

Segundo esta mecânica, as fórmulas de simulação se tornam bastante práticas, sendo possível de se implementar tudo, em apenas um laço (*loop*) principal.

7.3. Variáveis utilizadas

Para as variáveis que guardam os valores do ambiente, um tipo 'dicionário' é utilizado contendo as chaves: *temperatura*, *luminosidade* e *umidade*. Os valores das chaves são do tipo ponto-flutuante e guardam os valores correspondentes às suas chaves.

Os objetos lotados na sala são armazenados em outro dicionário que contém como chave o nome (ou identificador) do objeto e como valor uma lista contendo os seguintes itens:

- **Estado** – pode ser ligado ou desligado, se estiver desligado não influencia no ambiente.

- **Fator de calor:** Neste campo um fator é definido para o cálculo da temperatura do ambiente, se o fator é positivo, o objeto fornece calor, se for negativo se trata de um objeto que absorve o calor.
- **Limite de temperatura:** Este valor é o máximo ou o mínimo que o objeto pode alcançar de temperatura. Como exemplo, é dito que um ar-condicionado não refrigera menos que 10 graus célsius, ou um aquecedor não aquece mais que 40 graus célsius.
- **Intensidade luminosa:** É quanto um objeto pode iluminar um ambiente, por exemplo: é dito que uma lâmpada de 40w produz a intensidade de 600 candela.
- **Fator de umidade:** Fator que serve para calcular a influência do objeto na umidade do ambiente.

7.4. Construção do servidor

O servidor é construído utilizando o módulo Servidor descrito no capítulo 6.7.1 e possui uma *Classe de Execução* que contém os métodos e atributos utilizados nos processos referentes ao servidor da aplicação.

7.4.1. Classe de execução

A classe de execução é padrão, sem herança, que depois de instanciada e atribuída ao Servidor, processa e guarda informações referente à mecânica aplicada ao simulador.

7.4.1.1. Atributos

Os atributos estão divididos em dois dicionários: *status* e *tomadas*. O primeiro guarda o estado do ambiente, em forma de dicionário com valores referentes às chaves que podem ser: *temperatura, umidade e luminosidade*. O dicionário *tomadas* guarda os dados referentes às tomadas elétricas, as chaves são os nomes dos aparelhos ligados nelas e os valores são listas contendo os dados: *estado, coeficiente de temperatura, limite de temperatura, luminosidade* e *coeficiente de umidade.*

7.4.1.2. Métodos

Os métodos contidos na classe podem ser descritos conforme a tabela a seguir:

Método	*Função*
__init__ (construtor da classe)	É executado na inicialização do objeto, define as variáveis e atributos que serão utilizadas nos outros métodos.
getStatus	Atualiza as variáveis e retorna um dicionário contendo o estado do ambiente (temperatura, umidade, luminosidade).
getTomadasAtivas	Retorna a lista de tomadas (elétricas) que estão ligadas.
getTomadasInativas	Retorna a lista de tomadas (elétricas) que estão desligadas.
setAtiva, setInativa	Define como ativa ou inativa uma tomada.
setStatus	Define o estado do dicionário que contém os dados do ambiente.

Tabela 11: Métodos da classe de execução da aplicação exemplo

Estes métodos serão acessados pelo programa cliente que irá interagir com o usuário.

7.4.2. Código fonte

A seguir, o código fonte dos métodos será explicado para ilustrar o desenvolvimento.

```
0  def __init__(self):
1    self.status = {'ambiente':'Quarto',
2                   'temperatura':25,
3                   'umidade':40,
4                   'luminosidade':80 }
5
6    self.tomadas = {
7      'Ar Condicionado':[1, -0.01, 10, 0, -0.01],
8      'Aquecedor':[0, 0.03, 40, 0, 0.02],
9      'Micro':[1, 0.002, 34, 0, 0],
10     'Lampada 40w':[1, 0.002, 34, 45, 0.002],
11     'Lampada 25w':[1, 0.002, 34, 28, 0.001] }
12
13   self.tempo = time.time()
```

Código 29: Construtor classe servidor - exemplo

- Linhas 1 a 4: definição do dicionário de estado do ambiente.
- Linhas 6 a 11: definição do dicionário de estado das tomadas.
- Linha 13: variável que guarda a hora da última consulta.

O código do restante da classe pode ser observada no anexo contendo a listagem do código comentado.

7.5. Construção do cliente

O cliente é construído de forma que o usuário possa interagir com o Servidor utilizando uma interface gráfica contendo componentes como botões, campos de entrada, rótulos e listas.

Os eventos gerados pelo usuário podem ser:

- **Alterar parâmetros**: o usuário pode alterar os parâmetros do simulador como temperatura atual, umidade atual e luminosidade atual.
- **Enviar dados ao servidor**: se os parâmetros forem alterados pelo usuário, ele poderá enviar os mesmos ao servidor para que este atualize as suas variáveis.
- **Receber status do servidor**: recebe os dados do servidor e atualiza a interface.
- **Ligar ou desligar objetos**: o usuário pode ligar ou desligar objetos a distância, para isso a interface possui duas listas, uma de objetos ligado e outra de objetos desligados. Com um duplo-clique do mouse em cima de um objeto, ele troca de estado, e, por conseqüência, de lista.

7.5.1. Interface gráfica

A partir do editor de interface gráfica, o desenvolvedor pode “desenhar” as telas do sistema, então, a seguir, serão mostradas várias imagens ilustrando como a interface é criada.

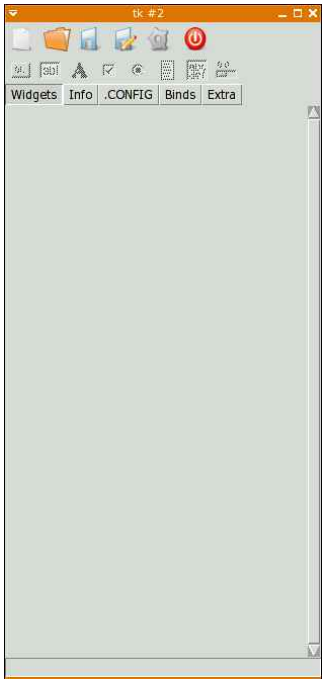

Ilustração 21: Editor de interfaces

Nesta janela, clicando no primeiro botão (folha em branco), é possível iniciar uma nova janela. No momento que se clica, uma janela de "salvar como" é aberta para a escolha do arquivo que será gravado com os dados da janela.

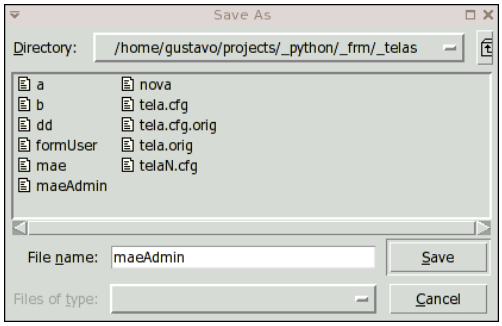

Ilustração 22: Janela de "salvar como"

Escolhido o arquivo, uma janela vazia é aberta para o início do trabalho de inserção de componentes.

Ilustração 23: Janela vazia, aguardando para receber componentes

Na seqüência, os componentes são inseridos na janela. A seguir é apresentada a janela com alguns *Labels*. Para isso, é necessário clicar sobre o ícone do componente que se deseja incluir na janela, atribuir-lhe um nome e arrastar até a posição desejada.

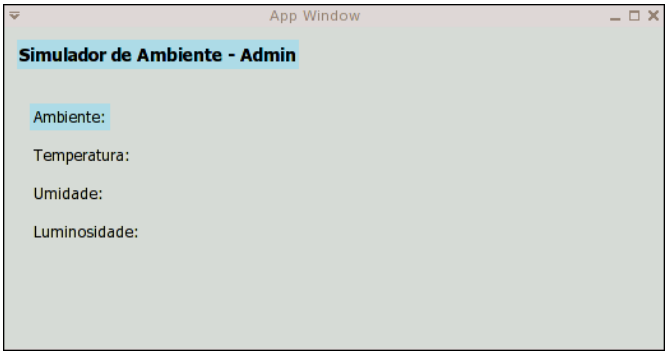

Ilustração 24: Janela da aplicação em desenvolvimento

Após este passo, são inseridos os componentes para a interação com o usuário: campos de entrada de dados, botões e listas.

Ilustração 25: Janela construída

O próximo passo é atribuir ações aos componentes. Os botões devem responder aos cliques recebidos e as listas devem possuir interação, ativando ou desativando os objetos ao receber duplo-clique.

Para isto, na janela do Editor, existe uma aba chamada ".CONFIG" e outra "Binds". Na primeira, em um dos campos, pode-se incluir o nome de um comando a ser executado (apenas para os botões) quando houver este evento, conforme a ilustração a seguir.

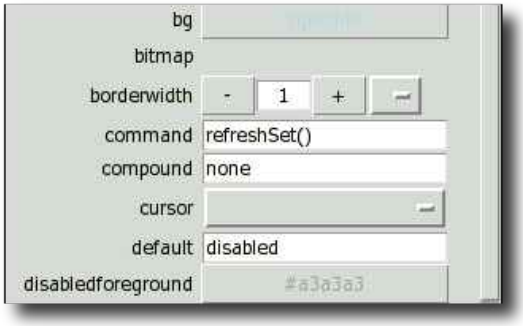

Ilustração 26: Configurando o comando a executar

Na figura, o comando a ser executado é o método *refreshSet()* que será apresentado no próximo sub-capítulo. Para as listas, o evento desejado é o duplo-clique do mouse, atribuído através de um *Bind*, que é acessado na aba "Binds" do editor. A configuração é mostrada na figura a seguir:

Ilustração 27: Atribuindo um Bind a uma lista

Nesta ilustração, é atribuído um evento '*<Double-1>*' ao método *setInativa(e)*, o parâmetro 'e', é passado pela biblioteca Tkinter e contém os dados do evento como posição do mouse e componente que gerou o evento, entre outros.

Após este procedimento a interface está pronta e pode ser salva, o Editor de interfaces pode ser fechado e o trabalho continua na *classe de execução* que será mostrada na seqüência.

7.5.2. Classe de execução

A classe de execução da interface gráfica do cliente possui os métodos necessários para a interação com o usuário e a troca de dados com o servidor.

A interação com o usuário é feita por meio de eventos disparados pela interface gráfica que executam os métodos da classe escrita. A biblioteca utilizada é a *pideWindow*, vista no *Módulo de interface gráfica* (capítulo 6.6).

A execução remota de procedimentos e a troca de dados é feita utilizando os métodos da classe *pideClient* mostrada no capítulo 6.7.2 e possui os atributos e métodos listados a seguir.

Atributo	***Descrição***
ascFields	Campos da interface que são do tipo texto
floatFields	Campos da interface que são do tipo *float*
fields	A soma da lista ascFields e floatFields
w	Instância da classe *pideWindow* que será utilizada para gerenciar a interface gráfica
c	Instância da classe *pideClient* que será utilizada para gerenciar a comunicação com o servidor

Tabela 12: Atributos da classe de execução da aplicação exemplo

Os atributos "**fields"*, são listas constantes que guardam os nomes (identificadores) dos componentes da interface gráfica que serão lidos (campo de entrada de texto) através do método *get* da classe *pideWindow*.

Método	***Descrição***
start	Inicia os atributos, zerando os campos da interface gráfica e instanciado a classe *pideClient*
getParms	Lê os valores dos campos da interface e retorna um dicionário
setParms	Atribui os valores recebidos aos campos da interface gráfica
setAtiva	Ativa um objeto no servidor (ex. liga uma lâmpada)
setDesativa	Desativa um objeto no servidor (ex. desliga uma lâmpada)
refreshGet	Lê o estado do servidor
refreshSet	Salva o estado no servidor

Tabela 13: Métodos da classe de execução da aplicação exemplo

Os códigos referentes ao cliente e o servidor podem ser consultados nos anexos deste trabalho.

7.6. Execução

Após a execução do programa, o cliente está pronto para interagir com o usuário. A primeira coisa a se fazer é clicar no botão *refrehGet* que atualiza os campos com os dados que receber do servidor. Para ativar um objeto (ex.: acender uma lâmpada), basta um duplo-clique no objeto apresentado na lista *Tomadas inativas*. Para desativar um objeto, basta executar o mesmo procedimento, porém na lista *Tomadas ativas*.

Para que a interface seja atualizada com os valores atuais, basta pressionar o botão *refreshGet*. Se for necessário modificar o ambiente com um valor estabelecido na interface, o botão *refreshSet* deverá ser acionado.

Neste capítulo foi desenvolvido uma pequena demonstração para ilustrar como se pode aplicar de maneira concreta a plataforma em uma situação real. Nesta aplicação ficou claro que com poucos passos e sem muita referência a se decorar, é possível construir uma pequena aplicação distribuída que seja executada remotamente, de forma transparente e consistente.

CONSIDERAÇÕES FINAIS

Neste trabalho, foram vistos conceitos de redes de computadores, sistemas distribuídos e linguagens interpretadas, com ênfase na linguagem Python, que está aplicada a este. Os três assuntos são muito importantes a este projeto que aborda 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, tornando a compreensão e o desenvolvimento mais organizado.

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 que serão objetos de interesse do desenvolvedor e não da plataforma, isto é, para que a plataforma não fique "engessada" no compromisso de se possuir um ambiente puramente distribuído. Em contrapartida, ficou claro a importância dos mesmos para a evolução futura da plataforma em termos de opções e parametrização da mesma.

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.

O desenvolvimento da plataforma se deu de maneira resumida, sem se aprofundar muito em todas as partes, mas sim no fundamental para compreensão das características mais relevantes. Alguns itens foram suprimidos para que o assunto não fosse tangenciado e este não se tornasse um documento que ensina a programar.

Para ilustrar o seu funcionamento, pode-se verificar uma pequena aplicação-exemplo que demonstrou a sua utilização em uma sistema real, mesmo não utilizando todos os recursos que a linguagem e a plataforma oferecem, foi possível ver como se dá a construção e execução da mesma.

Com este trabalho, uma forma alternativa de se produzir software foi apresentado e é claro, e de conhecimento comum, que uma plataforma não se cria de forma definitiva em um passo único, então pode-se esperar avanços da mesma com o decorrer do tempo e do uso.

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 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.

Para que a plataforma seja ainda mais produtiva, o suporte a comunicação assíncrona pode ser feito para que os dados sejam produzidos no servidor e enviados ao cliente sem a necessidade de haver consulta disparada do lado cliente.

Melhorias em geral nas ferramentas, onde o conjunto de componentes gráficos possa ser abordado em sua totalidade e outras características como depuração e um complemento de código mais apurado também podem ser desenvolvidos futuramente.

REFERÊNCIAS BIBLIOGRÁFICAS

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

FAQS.ORG. **RFC 822** - Standard for the format of ARPA Internet text messages. 1982. Disponível em: http://www.faqs.org/rfcs/rfc822.html Acesso em setembro de 2005.

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.

ANEXO I – testAppServer.py

Este anexo se refere ao programa exemplo citado no capítulo 7, e se trata do código fonte do servidor.

```
from pideServer import *
import time

UMIDADE_MAXIMA = 94
UMIDADE_MINIMA = 23

# class to export
class Teste:
    def __init__(self):
        self.status = {'ambiente':'Quarto', 'temperatura':25, 'umidade':40,
'luminosidade':80}
        # nome, [status, incremento, maximo/minimo, luminosidade, umidade
        self.tomadas = {
            'Ar Condicionado':[1, -0.01, 10, 0, -0.01],
            'Aquecedor':[0, 0.03, 40, 0, 0.02],
            'Micro':[1, 0.002, 34, 0, 0],
            'Lampada 40w':[1, 0.002, 34, 45, 0.002],
            'Lampada 25w':[1, 0.002, 34, 28, 0.001]
            }
        self.tempo = time.time()

    def getStatus(self):
        # tempo
        tn = time.time()            # hora (em s) local
        t = tn - self.tempo         # tempo q passou desde o ultimo get
        self.tempo = tn             # hora do ultimo get

        # aliases
        to = self.tomadas                       # tomadas
        te = self.status['temperatura']         # temperatura
```

```
    lu = self.status['luminosidade'] = 0  # luminosidade
    um = self.status['umidade']           # umidade

    for i in to:
        # aliases
        ts = to[i][0]                     # status (0 ou 1)
        ti = to[i][1]                     # temperatura incremento
        tm = to[i][2]                     # temperatura maxima / minima
        li = to[i][3]                     # luminosidade
        ui = to[i][4]                     # umidade incremento

        if ts==1:                         # tomada ligada?

            # temperatura
            if ti > 0:                    # aquecendo? (ti > 0)
                if te < tm:               # menor q o maximo?
                    te += t * ti          # incrementa
                    if te > tm: te = tm   # estourou? vai pro maximo

            else:
                if te > tm:               # maior que o minimo?
                    te += t * ti          # decrementa
                    if te < tm: te = tm   # estourou? vai pro minimo

            # luminosidade
            lu += li
            if lu > 100: lu = 100

            # umidade
            um += ui * t
            if um > UMIDADE_MAXIMA: um = UMIDADE_MAXIMA
            if um < UMIDADE_MINIMA: um = UMIDADE_MINIMA

            # set vars
            self.status['temperatura'] = te
            self.status['luminosidade'] = lu
            self.status['umidade'] = um


    return self.status

def getTomadasAtivas(self):
    r = []
    to = self.tomadas
    for i in to:
        if to[i][0] == 1:
            r.append(i)

    return r

def getTomadasInativas(self):
    r = []
    to = self.tomadas
    for i in to:
        if to[i][0] == 0:
            r.append(i)

    return r
```

```
    def setAtiva(self, tomada):
        self.getStatus()
        self.tomadas[tomada][0] = 1

    def setInativa(self, tomada):
        self.getStatus()
        self.tomadas[tomada][0] = 0

    def setStatus(self, d):
        self.status = d
        print 'setStatus'

#class instance
h = Teste()

# start server
s = pideServer(h)
```

ANEXO II – testAppAdmin.py

Este anexo se refere ao programa exemplo citado no capítulo 7, e se trata do código fonte do cliente.

```
from pideWindow import *
from pideClient import *
from Tkinter import *
import sys
import socket

WINDOW = '_telas/testAppGUI'

class Funcs:
    def start(self):
        self.ascFields = ['ambiente']
        self.floatFields = ['temperatura', 'umidade', 'luminosidade']
        self.fields = self.ascFields + self.floatFields
        for i in self.fields:
            w.get(i).config(text='')

        self.c = pideClient()

    def getParms(self):
        d = {}
        for i in self.floatFields: d[i] = float(w.get(i).get())
        for i in self.ascFields: d[i] = w.get(i).get()
        return d

    def setParms(self, parms):
        for i in self.fields:
            w.get(i).delete(0, END)
            if i in self.ascFields:
                v = parms[i]
            if i in self.floatFields:
```

```
                v = '%2.2f' % (float(parms[i]))
            w.get(i).insert(0, v)

    def setAtiva(self, e):
        t = e.widget.get(ACTIVE)
        self.c.send('setAtiva', t)
        self.refreshGet()

    def setInativa(self, e):
        t = e.widget.get(ACTIVE)
        self.c.send('setInativa', t)
        self.refreshGet()

    def refreshGet(self):
        d = self.c.send('getStatus')
        self.setParms(d[1])

        d = self.c.send('getTomadasAtivas')
        ta = w.get('tomadasAtivas')
        ta.delete(0, END)
        for i in d[1]:
            ta.insert(0, i)

        d = self.c.send('getTomadasInativas')
        ta = w.get('tomadasInativas')
        ta.delete(0, END)
        for i in d[1]:
            ta.insert(0, i)


    def refreshSet(self):
        d = self.getParms()
        d = self.c.send('setStatus', (d))

f = Funcs()
w = Wind(WINDOW, funcs=f)

f.start()

tk.mainloop()
```

ANEXO III – testAppGUI

Este anexo se refere ao programa exemplo citado no capítulo 7, e se trata do arquivo contendo as definições da interface gráfica.

```
[lAmbiente]
highlightthickness = 0
text = Ambiente:
image =
compound = none
height = 0
borderwidth = 2
pady = 1
padx = 1
font = Tahoma -12
activeforeground = Black
activebackground = #ececec
underline = -1
width = 0
state = normal
highlightcolor = Black
textvariable =
type = Label
takefocus = 0
bd = 2
foreground = Black
bg = #d9d9d9
background = #d9d9d9
fg = Black
bitmap =
highlightbackground = #d9d9d9
disabledforeground = #a3a3a3
wraplength = 0
cursor =
place = 20x60
```

```
relief = flat
anchor = center
justify = center

[temperatura]
readonlybackground = #d9d9d9
highlightthickness = 1
show =
xscrollcommand =
insertwidth = 2
exportselection = 1
borderwidth = 1
font = Tahoma -12
insertontime = 600
insertbackground = Black
width = 20
state = normal
highlightcolor = Black
selectbackground = blue
textvariable =
disabledbackground = #d9d9d9
type = Entry
takefocus =
bd = 1
foreground = Black
bg = white
selectborderwidth = 0
validatecommand =
background = white
fg = Black
validate = none
highlightbackground = #d9d9d9
disabledforeground = #a3a3a3
insertborderwidth = 0
selectforeground = white
insertofftime = 300
invalidcommand =
vcmd =
invcmd =
cursor = xterm
place = 110x90
relief = sunken
justify = left

[lTemperatura]
highlightthickness = 0
text = Temperatura:
image =
compound = none
height = 0
borderwidth = 2
pady = 1
padx = 1
font = Tahoma -12
activeforeground = Black
activebackground = #ececec
underline = -1
width = 0
state = normal
```

```
highlightcolor = Black
textvariable =
type = Label
takefocus = 0
bd = 2
foreground = Black
bg = #d9d9d9
background = #d9d9d9
fg = Black
bitmap =
highlightbackground = #d9d9d9
disabledforeground = #a3a3a3
wraplength = 0
cursor =
place = 20x90
relief = flat
anchor = center
justify = center

[tomadasInativas]
selectmode = browse
highlightthickness = 1
setgrid = 0
xscrollcommand =
height = 13
borderwidth = 1
font = Tahoma -12
activestyle = underline
_bindaction1 = setAtiva(e)
width = 20
state = normal
highlightcolor = Black
selectbackground = blue
_bindevent1 = <Double-1>
listvariable =
type = Listbox
takefocus =
bd = 1
foreground = Black
bg = white
selectborderwidth = 0
background = white
fg = Black
exportselection = 0
highlightbackground = #d9d9d9
disabledforeground = #a3a3a3
selectforeground = white
yscrollcommand =
cursor =
place = 380x30
relief = sunken

[luminosidade]
readonlybackground = #d9d9d9
highlightthickness = 1
show =
xscrollcommand =
insertwidth = 2
exportselection = 1
```

```
borderwidth = 1
font = Tahoma -12
insertontime = 600
insertbackground = Black
width = 20
state = normal
highlightcolor = Black
selectbackground = blue
textvariable =
disabledbackground = #d9d9d9
type = Entry
takefocus =
bd = 1
foreground = Black
bg = white
selectborderwidth = 0
validatecommand =
background = white
fg = Black
validate = none
highlightbackground = #d9d9d9
disabledforeground = #a3a3a3
insertborderwidth = 0
selectforeground = white
insertofftime = 300
invalidcommand =
vcmd =
invcmd =
cursor = xterm
place = 110x150
relief = sunken
justify = left

[refreshSet]
highlightthickness = 0
text = refreshSet
image =
compound = none
height = 3
borderwidth = 1
pady = 1m
padx = 3m
font = Tahoma -12
activeforeground = Black
activebackground = #ececec
underline = -1
width = 10
state = normal
highlightcolor = Black
textvariable =
command = refreshSet()
overrelief = raised
type = Button
takefocus =
bd = 1
foreground = Black
bg = lightblue
repeatinterval = 0
repeatdelay = 0
```

```
background = lightblue
fg = Black
bitmap =
highlightbackground = #d9d9d9
disabledforeground = #a3a3a3
wraplength = 0
default = disabled
cursor =
place = 20x180
relief = raised
anchor = center
justify = center

[umidade]
readonlybackground = #d9d9d9
highlightthickness = 1
show =
xscrollcommand =
insertwidth = 2
exportselection = 1
borderwidth = 1
font = Tahoma -12
insertontime = 600
insertbackground = Black
width = 20
state = normal
highlightcolor = Black
selectbackground = blue
textvariable =
disabledbackground = #d9d9d9
type = Entry
takefocus =
bd = 1
foreground = Black
bg = white
selectborderwidth = 0
validatecommand =
background = white
fg = Black
validate = none
highlightbackground = #d9d9d9
disabledforeground = #a3a3a3
insertborderwidth = 0
selectforeground = white
insertofftime = 300
invalidcommand =
vcmd =
invcmd =
cursor = xterm
place = 110x120
relief = sunken
justify = left

[refreshGet]
highlightthickness = 0
text = refreshGet
image =
compound = none
height = 3
```

```
borderwidth = 1
pady = 1m
padx = 3m
font = Tahoma -12
activeforeground = Black
activebackground = #ececec
underline = -1
width = 10
state = normal
highlightcolor = Black
textvariable =
command = refreshGet()
overrelief = raised
type = Button
takefocus =
bd = 1
foreground = Black
bg = #d9d9d9
repeatinterval = 0
repeatdelay = 0
background = #d9d9d9
fg = Black
bitmap =
highlightbackground = #d9d9d9
disabledforeground = #a3a3a3
wraplength = 0
default = disabled
cursor =
place = 150x180
relief = raised
anchor = center
justify = center

[lTomadasAtivas]
highlightthickness = 0
text = Tomadas Ativas
image =
compound = none
height = 0
borderwidth = 2
pady = 1
padx = 1
font = Tahoma -12
activeforeground = Black
activebackground = #ececec
underline = -1
width = 0
state = normal
highlightcolor = Black
textvariable =
type = Label
takefocus = 0
bd = 2
foreground = Black
bg = #d9d9d9
background = #d9d9d9
fg = Black
bitmap =
highlightbackground = #d9d9d9
```

```
disabledforeground = #a3a3a3
wraplength = 0
cursor =
place = 250x10
relief = flat
anchor = center
justify = center

[__root]
type = Tk
size = 517x252

[tomadasAtivas]
selectmode = browse
highlightthickness = 1
setgrid = 0
xscrollcommand =
height = 13
borderwidth = 1
font = Tahoma -12
activestyle = underline
_bindaction1 = setInativa(e)
width = 20
state = normal
highlightcolor = Black
selectbackground = blue
_bindevent1 = <Double-1>
listvariable =
type = Listbox
takefocus =
bd = 1
foreground = Black
bg = white
selectborderwidth = 0
background = white
fg = Black
exportselection = 0
highlightbackground = #d9d9d9
disabledforeground = #a3a3a3
selectforeground = white
yscrollcommand =
cursor =
place = 250x30
relief = sunken

[lTomadasInativas]
highlightthickness = 0
text = Tomadas Inativas
image =
compound = none
height = 0
borderwidth = 2
pady = 1
padx = 1
font = Tahoma -12
activeforeground = Black
activebackground = #ececec
underline = -1
width = 0
```

```
state = normal
highlightcolor = Black
textvariable =
type = Label
takefocus = 0
bd = 2
foreground = Black
bg = #d9d9d9
background = #d9d9d9
fg = Black
bitmap =
highlightbackground = #d9d9d9
disabledforeground = #a3a3a3
wraplength = 0
cursor =
place = 380x10
relief = flat
anchor = center
justify = center

[lTitulo]
highlightthickness = 0
text = Simulador de Ambiente - Admin
image =
compound = none
height = 0
borderwidth = 2
pady = 1
padx = 0
font = Tahoma -14 bold
activeforeground = Black
activebackground = #ececec
underline = -1
width = 0
state = normal
highlightcolor = Black
textvariable =
type = Label
takefocus = 0
bd = 2
foreground = Black
bg = lightblue
background = lightblue
fg = Black
bitmap =
highlightbackground = #d9d9d9
disabledforeground = #a3a3a3
wraplength = 0
cursor =
place = 10x10
relief = flat
anchor = center
justify = center

[lLuminosidade]
highlightthickness = 0
text = Luminosidade:
image =
compound = none
```

```
height = 0
borderwidth = 2
pady = 1
padx = 1
font = Tahoma -12
activeforeground = Black
activebackground = #ececec
underline = -1
width = 0
state = normal
highlightcolor = Black
textvariable =
type = Label
takefocus = 0
bd = 2
foreground = Black
bg = #d9d9d9
background = #d9d9d9
fg = Black
bitmap =
highlightbackground = #d9d9d9
disabledforeground = #a3a3a3
wraplength = 0
cursor =
place = 20x150
relief = flat
anchor = center
justify = center

[ambiente]
readonlybackground = #d9d9d9
highlightthickness = 1
show =
xscrollcommand =
insertwidth = 2
exportselection = 1
borderwidth = 1
font = Tahoma -12
insertontime = 600
insertbackground = Black
width = 20
state = normal
highlightcolor = Black
selectbackground = blue
textvariable =
disabledbackground = #d9d9d9
type = Entry
takefocus =
bd = 1
foreground = Black
bg = white
selectborderwidth = 0
validatecommand =
background = white
fg = Black
validate = none
highlightbackground = #d9d9d9
disabledforeground = #a3a3a3
insertborderwidth = 0
```

```
selectforeground = white
insertofftime = 300
invalidcommand =
vcmd =
invcmd =
cursor = xterm
place = 110x60
relief = sunken
justify = left

[lUmidade]
highlightthickness = 0
text = Umidade:
image =
compound = none
height = 0
borderwidth = 2
pady = 1
padx = 1
font = Tahoma -12
activeforeground = Black
activebackground = #ececec
underline = -1
width = 0
state = normal
highlightcolor = Black
textvariable =
type = Label
takefocus = 0
bd = 2
foreground = Black
bg = #d9d9d9
background = #d9d9d9
fg = Black
bitmap =
highlightbackground = #d9d9d9
disabledforeground = #a3a3a3
wraplength = 0
cursor =
place = 20x120
relief = flat
anchor = center
justify = center
```