UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

VINÍCIUS SCHEEREN

MÓDULO PARA INSPEÇÃO AUTOMÁTICA DE LINHAS DE TRANSMISSÃO POR TERMOGRAFIA

Porto Alegre
2011

VINÍCIUS SCHEEREN

MÓDULO PARA INSPEÇÃO AUTOMÁTICA DE LINHAS DE TRANSMISSÃO POR TERMOGRAFIA

Projeto de Diplomação apresentado ao Departamento de Engenharia Elétrica da Universidade Federal do Rio Grande do Sul como parte dos requisitos para a obtenção do título de Engenheiro Eletricista.

ORIENTADOR: Prof. Dr. Walter Fetter Lages

Porto Alegre
2011

VINÍCIUS SCHEEREN

MÓDULO PARA INSPEÇÃO AUTOMÁTICA DE LINHAS DE TRANSMISSÃO POR TERMOGRAFIA

Este Projeto foi julgado adequado para a obtenção dos créditos da Disciplina Projeto de Diplomação do Departamento de Engenharia Elétrica e aprovado em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: _____
Prof. Dr. Walter Fetter Lages, UFRGS
Doutor pelo Instituto tecnológico de Aeronáutica - São José dos Campos, Brasil

Chefe do DELET: _____
Prof. Dr. Altamiro Amadeu Susin

Porto Alegre, julho de 2011.

DEDICATÓRIA

Dedico este trabalho aos meus pais Ervino e Izolde e à minha irmã Patrícia.

AGRADECIMENTOS

Agradeço aos meus pais por todo o apoio dado antes e durante a minha formação acadêmica, à minha irmã pela atenção e ajuda em todos os momentos, à UFRGS e aos professores pelo ensino de qualidade e aos meus amigos e colegas pela ajuda e momentos de descontração.

RESUMO

Linhas de transmissão estão sujeitas a desgastes provocados pelos mais diversos fatores, os quais podem acarretar em grandes perdas de potência ou até mesmo faltas provocadas pelo rompimento completo dos condutores. Procedimentos emergenciais de reparo apresentam custos muito elevados, motivando o desenvolvimento de sistemas capazes de detectar preditivamente regiões da linha que apresentam falhas. Um sistema robótico para inspeção de linhas de transmissão está sendo desenvolvido no LASCAR - UFRGS, e dentro do âmbito destes projeto, se faz necessário o desenvolvimento de um módulo a ser embarcado no sistema capaz de detectar as falhas. Este projeto tratará do desenvolvimento de um módulo capaz de detectar falhas em linhas de transmissão de forma automática utilizando termografia.

Palavras-chave: Inspeção de linhas de transmissão, RTSP, Termografia.

ABSTRACT

Power lines can wear out because of several factors, which can result in high power loss or even complete power faults caused by broken lines. Emergency repair procedures are high costly, motivating the development of systems capable of detecting damaged lines in a predictive manner. A Power line inspection robot is being developed in LASCAR - UFRGS, and within the scope of this project, it is necessary the development of a module to be embedded in the system turning it capable of detecting faults. This project pursue the development of a module capable of detecting faults in transmission lines automatically using thermography.

Keywords: Power line inspection, termography, RTSP.

LISTA DE ILUSTRAÇÕES

Figura 1: Câmera FLIR A320. 14
Figura 2: BeagleBoard-xM. 16
Figura 3: Câmera Logitech C210. 17
Figura 4: Interação entre os elementos de hardware. 17
Figura 5: Hotspot e Temperatura de referência. 23
Figura 6: Modelo OSI. 28
Figura 7: Encapsulamento dos pacotes RTP. 30
Figura 8: Cabeçalho de um pacote RTP. 30
Figura 9: Máquina de estados do protocolo RTSP. 32
Figura 10: Fluxo global dos dados. 39
Figura 11: Hierarquia de classes do aplicativo *Detector*. 42
Figura 12: Ligação do cabo na fonte. 47
Figura 13: Posicionamento das câmeras e do cabo durante os testes. 47
Figura 14: BeagleBoard e o roteador utilizado. 48
Figura 15: Interface gráfica do aplicativo *Detector* durante o teste. 48
Figura 16: Visualização da stream gerada pelo módulo no software VLC. 49

LISTA DE TABELAS

Tabela 1: Condições de Medidas. 27
Tabela 2: Métodos da classe `Itada`. 44

LISTA DE ABREVIATURAS

LASCAR	*Laboratório de Sistema de Controle, Automação e Robótica*
RFC	*Request for Comments*
IETF	*Internet Engineering Task Force*
TCP	*Transfer Control Protocol*
UDP	*User Data Protocol*
IP	*Internet Protocol*
RTSP	*Real Time Streaming Protocol*
SDP	*Session Description Protocol*
RTP	*Real-time Transport Protocol*
ITADA	*Thermograghy Anomaly Detection Algorithm*
LAN	*Local Area Network*

SUMÁRIO

1 INTRODUÇÃO 12
1.1 Motivação 12
1.2 Objetivos 13
1.3 Organização 13

2 HARDWARE UTILIZADO 14
2.1 Câmera termográfica 14
2.2 Módulo para processamento da imagem 15
2.3 Câmera de imagens no espectro visível 16

3 SOFTWARE 18
3.1 Ferramentas utilizadas 20

4 PROCESSAMENTO DAS IMAGENS 22
4.1 Introdução 22
4.2 Descrição do algoritmo 22
4.3 Segmentação da imagem 22
4.4 Definição da temperatura de *threshold* T_t 23
4.5 Detecção dos Hotspots 25
4.6 Cálculo da temperatura de referência 26
4.7 Classificação dos Hotspots 27
4.8 Otimização do algoritmo 27

5 TECNOLOGIAS PARA TRANSMISSÃO MULTIMíDIA 28
5.1 Protocolos para transmissão em tempo real 29
5.1.1 RTP 29
5.1.2 RTSP 31

6 BIBLIOTECAS RTSP 33
6.1 Exemplos de uso 34
6.1.1 Servidor RTSP 34
6.1.2 Cliente RTSP 36

7 IMPLEMENTAÇÃO 38
7.1 Organização do software 38
7.2 Comunicação entre processos 38
7.3 Complementação da biblioteca liveMedia 40
7.4 Receiver 41
7.5 Detector 41

7.5.1 USBGrabber 42
7.5.2 Image 42
7.5.3 Itada 43
7.5.4 Transmitter 43
7.5.5 Receiver 43
7.6 Transmitter 45

8 RESULTADOS 46

9 CONCLUSÃO 50

REFERÊNCIAS 51

1 INTRODUÇÃO

A transmissão de energia elétrica entre as fontes geradoras e o consumidor é feita com a utilização de milhares de quilômetros de linhas de transmissão (LT), compostas por cabos condutores de energia elétrica, geralmente de alumínio. Estas linhas de transmissão geralmente são aéreas, instaladas em torres.

O fato destas linhas estarem em contato com o ambiente externo, expõe as mesmas a condições adversas, principalmente por agentes climáticos e químicos, os quais contribuem para a degradação do material, que pode acarretar no rompimento completo de um cabo.

Falhas em linhas de transmissão geram diversos problemas para as concessionárias de energia, principalmente do ponto de vista financeiro, já que reparos emergenciais são sempre mais custosos que reparos programados. Estes fatores motivam as concessionárias a investirem em métodos de inspeção das linhas de transmissão, para garantir o mínimo de reparos emergenciais.

Os métodos de inspeção convencionais são realizados por operadores humanos com o uso de aeronaves, as quais sobrevoam as linhas a distancias muito próximas das mesmas, um procedimento considerado de alto risco. Além do risco, o uso de operadores humanos inspecionando as linhas visualmente induz em erros de diagnóstico, já que se trata de uma operação monótona e repetitiva para os operadores, com uma visão limitada de toda a área superficial do cabo. Por fim, este método de inspeção apresenta altos custos, principalmente pelo uso de aeronaves.

Sistemas de inspeção robóticos para esta aplicação estão sendo estudados e desenvolvidos, tanto na parte de locomoção do sistema quanto nos métodos de diagnóstico de falhas. Um destes sistemas robóticos está sendo desenvolvido no LASCAR, laboratório da UFRGS, e é denominado de *Power Line Inspection Robot* (PLIR). Trabalhos anteriores para este projeto apresentaram a eficácia do uso de câmeras termográficas para a detecção de falhas.

Neste projeto será desenvolvido um módulo que será embarcado neste sistema robótico capaz de adquirir imagens termográficas e no espectro visível, processá-las procurando por falha no cabo e enviar os resultados a um computador cliente em tempo real. Visto que as dimensões, peso e a autonomia são quesitos importantes para um sistema robótico que ficará suspenso sobre as linhas de transmissão, o hardware a ser utilizado deve atender a estas restrições.

1.1 Motivação

O desenvolvimento do sistema robótico para inspeção de linhas de transmissão é motivado pela necessidade que as concessionárias de energia elétrica têm em fazer um diag-

nóstico preditivo de falhas nas linhas de transmissão, evitando ao máximo a ocorrência de faltas e manutenções de emergência. Dentro deste escopo, o desenvolvimento de um módulo embarcado para o processamento das imagens obtidas a partir das câmeras é de suma importância para o funcionamento do projeto.

1.2 Objetivos

O objetivo deste projeto é desenvolver um módulo embarcado capaz de receber as imagens tanto de uma câmera termográfica quanto de uma câmera de imagens no espectro visível, processar os dados adquiridos buscando por possíveis falhas na linhas de transmissão e implementar um método que possibilite a recebimento dos resultados por um computador cliente à parte do sistema robótico. O módulo por sua vez deve atender às restrições de tamanho, peso e consumo necessárias para que seja possível embarcar o mesmo em um robô de pequenas dimensões.

1.3 Organização

O primeiro capítulo apresenta uma introdução ao projeto. O segundo capítulo trata sobre detalhes técnicos sobre o *hardware* escolhido para o projeto. O terceiro capítulo faz um estudo das ferramentas de software necessárias para o desenvolvimento de um sistema embarcado. O quarto capítulo faz um estudo sobre o método de detecção utilizado. O quinto capítulo apresenta um estudo sobre os protocolos utilizados na comunicação com a câmera termográfica e com os computadores clientes. O sexto capítulo faz uma descrição das bibliotecas utilizadas. O sétimo capítulo detalha a implementação dos softwares para o projeto. O oitavo capítulo apresenta os resultados e a validação do sistema.

2 HARDWARE UTILIZADO

O desenvolvimento do sistema teve início com a escolha do *hardware* a ser utilizado. Duas câmeras serão utilizadas pelo módulo de inspeção, uma delas termográfica e uma câmera convencional de imagens no espectro visível. Além das câmeras um módulo onde as imagens são processadas também foi utilizado. Este capítulo trata sobre detalhes sobre o *hardware* utilizado neste projeto.

2.1 Câmera termográfica

A câmera termográfica utilizada no projeto é uma câmera da empresa FLIR modelo A320, a qual pode ser vista na figura 1.

Figura 1: Câmera FLIR A320.

Alguns detalhes técnicos sobre a câmera estão relacionados abaixo:

- Faixa espectral 7.5 a 13 μm
- Resolução 320$\times$240

- Faixas de temperatura -20°C a +120°C ou de 0°C a 350°C. Opcionalmente pode ir acima de +1200°C.
- Precisão ±2°C ou ±2%
- Envio de imagens com codificação MPEG4 ou RAW
- Foco motorizado automático ou manual
- Possibilidade de envio de imagens via rede Ethernet.

Esta câmera suporta os protocolos RTSP e RTP para fazer o envio das imagens, transmitindo as imagens via rede Ethernet. Esta foi a forma de transmissão escolhida neste projeto.

2.2 Módulo para processamento da imagem

Tendo em vista que o módulo a ser desenvolvido deverá ser embarcado em um robô que ficará suspenso em linhas de transmissão, devem ser observados as dimensões e o consumo de energia do módulo.

A busca por um *hardware* com características compatíveis àquelas exigidas pelo projeto iniciou-se pela procura de um processador com consumo de energia adequado à necessidade. A grande maioria dos dispositivos portáteis como telefones celulares, PDAs, tablets, entre outro, utilizam processadores de arquitetura ARM, principalmente pela boa relação entre poder de processamento e consumo que processadores com esta arquitetura apresentam.

Outro ponto observado na escolha foi a possibilidade de executar um sistema operacional Linux embarcado no módulo. O projeto será desenvolvido para o sistema operacional Linux, sobre o qual irão executar os aplicativos que irão receber as imagens da câmera, realizar o processamento em cima das mesmas e enviar um fluxo de vídeo aos computadores clientes.

A placa de desenvolvimento BeagleBoard-xM se mostrou adequada para suprir as necessidades expostas acima. Ela apresenta as seguintes características:

- Dimensões: 8.25×8.25 cm
- Consumo: 1.8 a 2.2 Watts
- Processador ARM Cortex-A8
- 512 MB de RAM
- Portas USB
- Saída de vídeo DVI-D
- Conector Ethernet

A BeagleBoard-xM pode ser vista na figura 2.

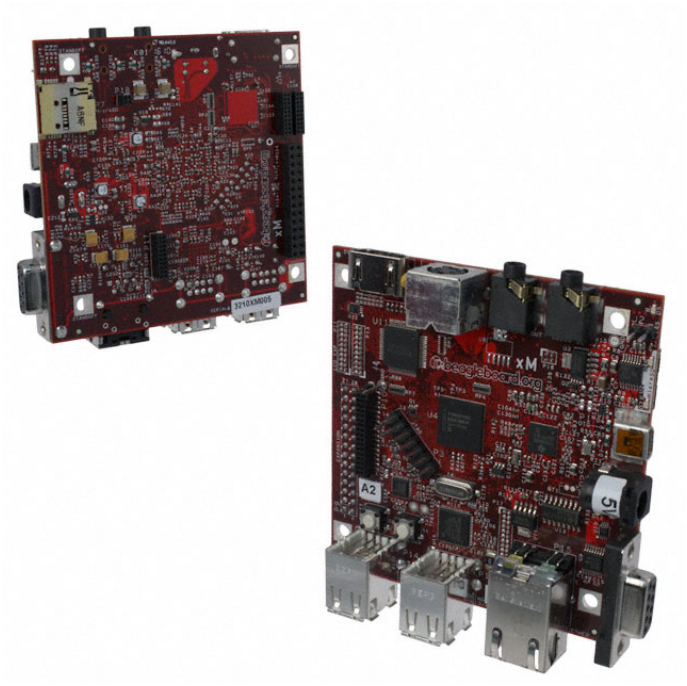

Figura 2: BeagleBoard-xM.

2.3 Câmera de imagens no espectro visível

De forma a obter uma referência visual do ambiente e da seção do cabo que está sendo analisada, uma câmera convencional será instalada no robô. As imagens obtidas por esta câmera não produzirão efeitos no processamento da imagens obtidas pela câmera termográfica, mas contribuirão com uma imagem da superfície do cabo que será analisada pelos operadores.

Para facilitar o interfaceamento com a placa que fará o processamento, decidiu-se por utilizar uma câmera com conector USB, do tipo utilizado em vídeo conferências pela internet. Os quesitos de qualidade de imagem não foram um fator determinante na escolha da câmera, já que as imagens serão utilizadas apenas para fornecer uma contextualização da falha para o operador.

A câmera utilizada é da marca Logitech, modelo C210, a qual pode ser vista na figura 3.

O diagrama de como todos elementos de hardware estão conectados e o fluxo dos dados neste sistema pode ser melhor visualizado na figura 4.

As imagens captadas pela câmera termográfica FLIR A320 são transferidas via rede Ethernet para a placa BeagleBoard, onde o processamento é realizado. Esta transferência de dados é feita utilizando o protocolo de transmissão em tempo real RTSP. Na outra ponta, a câmera Logitech C210 obtém imagens no espectro visível, as quais são transferidas via USB também para a BeagleBoard. Os dados são processados na placa e um novo *stream* RTSP contendo as imagens captadas e os quadros processados é gerado pelo sistema, o qual será enviado para um computador cliente em uma estação de monitoramento distante do robô.

Figura 3: Câmera Logitech C210.

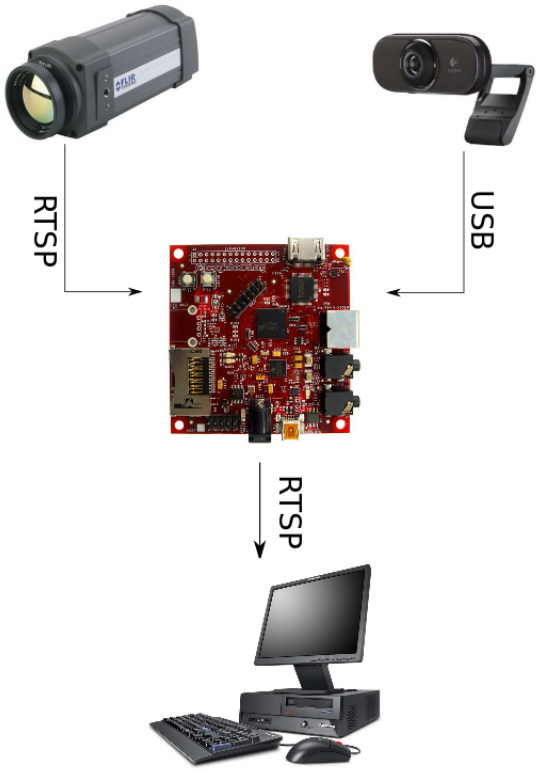

Figura 4: Interação entre os elementos de hardware.

3 SOFTWARE

Como apresentado nos capítulos anteriores, o módulo será composto de um *hardware* de pequenas dimensões com um arquitetura de processador ARM. A arquitetura ARM, embora bastante comum em dispositivos móveis, é menos utilizada do que a arquitetura x86, padrão na maioria dos computadores pessoais, o que torna mais complexa a tarefa de encontrar bibliotecas e *softwares* já compilados para estas plataformas.

Cada arquitetura precisa de um conjunto de ferramentas para gerar os arquivos binários a partir dos códigos fontes dos *softwares*, escritos nas mais diferentes linguagens de programação. Dentre esse conjunto de ferramentas, pode-se citar as principais:

- Compiladores: Conjunto de compiladores para diversas linguagens, utilizados para transformar os códigos fontes em códigos interpretáveis pelo montador;
- Utilitários: Conjunto de ferramentas incluindo o montador e o *linker*, ferramentas utilizadas para ligar os objetos compilados e transformar os códigos gerados pelo compilador em binários;
- Make: Ferramenta utilizada para automatizar a compilação e construção;
- Depurador: Ferramenta utilizada para depurar e encontrar erros no código;

Esses conjuntos de ferramentas são conhecidos como *toolchains*, os quais contém ferramentas para determinados conjuntos de arquiteturas de processadores. No caso de processadores para sistemas embarcados, como é o caso do ARM, as *toolchains* são conhecidas como *cross-toolchains*, derivado do termo em inglês *cross-compilation*, referente a compilação de *softwares* para sistemas embarcados.

É importante quando fala-se de *toolchains* distinguir os seguintes sistemas:

- Sistema construtor: sistema no qual a *toolchain* foi construída;
- Sistema hospedeiro: sistema no qual a *toolchain* é executada;
- Sistema alvo: sistema para o qual a *toolchain* gera os códigos.

Dependendo do sistema hospedeiro e alvo, pode-se classificar as *toolchains* nas seguintes categorias:

- *Toolchains* nativas: São *toolchains* encontradas normalmente em sistemas linux para computadores pessoais, geralmente compiladas em arquitetura x86, são executadas em arquitetura x86 e geram código para arquitetura x86.

- *Cross toolchains*: Geralmente usadas no desenvolvimento de sistemas embarcados, geralmente são construídas em arquitetura x86, executadas em arquitetura x86 e geram código para arquiteturas diferentes, como ARM, MIPS, PowerPC entre outras.

- *Toolchains* Cross Nativas: são *toolchains* que foram construídas em arquitetura x86, mas são executadas e geram código para determinadas arquiteturas alvos. São tipicamente utilizadas quando se deseja compilar os códigos diretamente no sistema embarcado, sem a necessidade de transferir os binários do sistema hospedeiro para o sistema alvo.

- *Canadian build* É o processo onde a *toolchain* é contruída em uma determinada arquitetura, é executada em uma segunda arquitetura e gera códigos para uma terceira arquitetura. Por exemplo, a toolchain pode ser construída em uma arquitetura x86, ser executada em uma arquitetura PowerPC e gerar códigos que sejam executados em ARM.

Essas *toolchains* podem ser dividas entre pré-construídas e contruídas pelo usuários. As *toolchains* pré-construídas, distribuídas prontas por grupos de desenvolvimento, são disponibilizadas já com os binários prontos para determinado sistema hospedeiro. As *toolchains* construídas pelo usuário, são compiladas no sistema construtor a partir de diversos fontes das ferramentas necessárias e suas dependências. Para facilitar esta tarefa, existem ferramentas que buscas os diversos fontes e dependências e realizam todas as tarefas necessárias para construir uma *toolchain* dependendo das opcões selecionadas.

Entre as *toolchains* pré-construídas pode-se citar:

- DENX ELDK

- Scratchbox

- Fedora ARM

Já para *toolchains* contruídas no sistemas hospedeiro existem as seguintes opções.

- Buildroot

- Crossdev (Gentoo)

- Crosstool

- Crosstool-NG

- Crossdev/tsrpm (Timesys)

- OSELAS

- Bitbake

3.1 Ferramentas utilizadas

Avaliando as opções de *toolchains* citadas, viu-se que a *toolchain BitBake* apresentou-se como a mais adequada para a realização deste projeto. Entre o motivos para esta escolha pode-se citar o fato de haver uma distribuição linux com suporte ao *hardware* utilizado neste projeto que foi construída com esta *toolchain*.

Para facilitar o uso da *toolchain* BitBake, foi desenvolvido o projeto OpenEmbedded, o qual consiste em um *framework* direcionado para a construção de distribuições Linux. Este projeto concentra um conjunto de instruções para construir uma distribuição utilizando o BitBake. Estas instruções são conhecidas como receitas, e guardam detalhes sobre onde encontrar os fontes para compilar, quais as dependências necessárias e executar as operações de compilação, montagem e ligação na ordem correta para todos os pacotes de softwares necessários. Sem este *framework*, a tarefa de criar uma distribuição com os pacotes de software necessários seria mais complexa.

Dentre as principais características do OpenEmbedded pode-se citar:

- Suporta tanto GlibC quanto UCLib;
- Suporte para construir binários para vários alvos diferentes com o mesmo fontes.
- Constrói automaticamente as dependências necessários para gerar e executar os binários.
- Suporte a vários tipos de pacotes
- Constrói automaticamente todas as ferramentas de compilação necessárias.

As ações executadas pelo OpenEmbedded são listadas e explicadas abaixo:

- Download dos códigos fontes: Nesta etapa são baixados todos os códigos fontes necessários para a construção do projeto. Os códigos podem ser adquiridos por download convencional ou de algum repositório git ou svn.
- Extração: A tarefa de extração consiste em extrair os códigos fontes de arquivos .tar.gz ou .zip por exemplo.
- *Patch*: Nesta tarefa são aplicados as correções e modificações nos códigos fontes originais.
- Configuração: Nesta tarefa são configuradas as opções de compilação dos pacotes, geralmente executando o comando ".configure <opções>"e gerando uma Makefile para o projeto.
- Compilação: Nesta etapa é executada a compilação do projeto previamente configurado. Do ponto de vista do OpenEmbedded, geralmente é simplesmente executado um comando "Make".
- Gravação na árvore de diretórios: Neste estágio, os cabeçalhos e bibliotecas gerados e que serão necessários por outros aplicativos são armazenados na arvore de diretórios do sistema que esta sendo construídos. Esta tarefa é diferente da tarefa de instalação pelo fato que a tarefa de instalação é direcionada a gravar os arquivos em locais específicos somente para a construção do pacote em questão.

- Instalação: Durante a instalação, os binários compilados são posicionados nos diretórios corretos dentro da arvore completa de diretórios. Por exemplo, se um binários deve ficar na pasta "/bin"do sistema alvo, este binário é copiado para a pasta "D/bin"do sistema construtor, onde "D"é chamada de pasta de destino no sistema construtor, a qual será a raiz do sistema alvo.

- Empacotamento: A tarefa de empacotamento utiliza os arquivos instalados e separa-os por pacote, gerando uma árvore de diretórios do sistema para cada um deles.

- Gravação do pacote: Na gravação do pacote, a árvore de diretórios gerada para cada pacote é utilizada para criar um arquivo de intalaçõa do tipo .rpm, .deb ou .ipk.

O OpenEmbedded é utilizado para a construção de uma distribuição Linux voltada para sistemas embarcados chamada de Angstrom. Todas as receitas para os pacotes utilizados por essa distribuição estão disponíveis no projeto OpenEmbedded, e a criação de uma versão customizada do Angstrom pode ser facilmente obtida.

A biblioteca Live555, utilizada para na recepção e envio de pacotes RTSP também foi compilada com esta ferramenta, o que simplificou razoavelmente o processo de compilação cruzada dos fontes para a arquitetura utilizada neste projeto.

4 PROCESSAMENTO DAS IMAGENS

4.1 Introdução

As imagens obtidas pela câmera termográfica devem ser processadas *frame* a *frame* para avaliar se a seção da linha de transmissão enquadrada pela imagem apresenta ou não alguma falha, classificando a mesma em casos positivos.

O processamento utilizado neste trabalho é uma implementação de um método conhecido por *Termography Anomaly Detection Algorithm* (ITADA) proposto por (CHOU; YAO, 2009). Este método faz a análise para detectar regiões quentes, e utiliza alguns critérios para classificar a criticidade da falha.

4.2 Descrição do algoritmo

O algoritmo ITADA utiliza algumas técnicas de processamento para analisar os dados termográficos de forma a identificar pontos quentes. Métodos estatísticos e morfológicos são utilizados na análise.

O primeiro passo executado pelo algoritmo é eliminar as baixas temperaturas, conhecidas como temperatura de *background*, mantendo apenas os pontos considerados quentes, ou *foreground*. O *foreground* é então avaliado para que sejam identificados os *hotspots*, ou áreas com as maiores temperaturas.

A seguir, é definida uma temperatura de referência, a partir de um ponto da imagem que represente uma área da mesma estrutura que está sendo avaliada. A figura 5 apresenta uma imagem com um *Hotspot* e uma Temperatura de referência.

A partir dos dados de temperaturas nestes pontos, são executados métodos estatísticos para avaliar se os pontos com maior temperatura estão em estado de operação normal ou apresentam algum defeito.

4.3 Segmentação da imagem

A primeira tarefa a ser executada com a imagem termográfica obtida é separar a imagem em componentes de alta temperatura e componentes de baixa temperatura. Visto que o meio ambiente sempre apresenta temperaturas mais baixas que os cabos a serem avaliados, é possível remover as partes que não representam a estrutura do cabo da imagem. Para realizar tal operação, alguns conjuntos de valores são definidos:

- Temperatura de *Foreground*: Componentes quentes da imagem
- Temperatura de *Background*: Componentes frios que não devem ser analisados

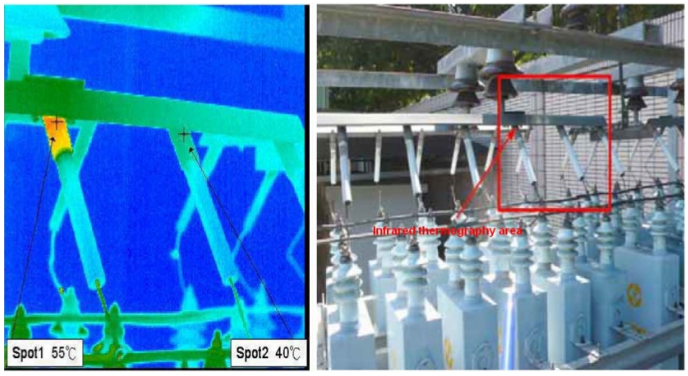

Figura 5: Hotspot e Temperatura de referência.

- Temperatura de threshold(Tt): Valor limiar de temperatura utilizado para segmentar a imagem.

Dada uma certa temperatura de *threshold* Tt, a separação entre o *Foreground* e o *Background* pode ser feita aplicando uma máscara à imagem, os componentes com valor abaixo do limiar são zerados e os acima do limiar assumem valor 1.

$$\beta(x,y) = \begin{cases} 1 & \text{se } \alpha(x,y) \geq T_t \\ 0 & \text{se } \alpha(x,y) < T_t \end{cases} \quad \forall 0 \leq x < W, 0 \leq y < H \tag{1}$$

onde:

- α é a imagem original;
- T_t é a temperatura de *threshold*;
- W é a largura da imagem em número de píxeis;
- H é a altura da imagem em número de píxeis;
- x e y são valores inteiros que indicam a posição de determinado píxel.

Após obtida a máscara de segmentação, a mesma é aplicada sobre a imagem original, produzindo o *foreground* da imagem:

$$\gamma(x,y) = \begin{cases} \alpha(x,y) & \text{se } \beta(x,y) = 1 \\ 0 & \text{se } \beta(x,y) = 0 \end{cases} \quad \forall 0 \leq x < W, 0 \leq y < H \tag{2}$$

4.4 Definição da temperatura de *threshold* T_t

De forma a se ter uma máscara que filtre apenas as partes significativas da imagem, é necessário definir uma temperatura limiar adequada, a qual deve ser calculada dinamicamente com base em algum algoritmo.

O método proposto por (OTSU, 1979) para a seleção de nível de *threshold* adequado em imagens em escala de cinza se mostra bastante adequado para selecionar a temperatura de limiar neste projeto.

Segundo (OTSU, 1979), idealmente há dois picos de níveis de cinza em um histograma de uma imagem qualquer, e o nível limiar que se procura está localizado no

vale entre estes dois picos. O algoritmo assume que a imagem contém duas classes de pixels. Os píxels com valor maior que T_t compõem a classe de *foreground* C_f, enquanto os outros formam a classe de *background* C_b.

A temperatura limiar procurada, é aquela que maximiza a variância entre as classes C_b e C_f:

$$\sigma_W^2 = \omega_b(\mu_b - \mu_t)^2 + \omega_f(\mu_f - \mu_t)^2 \tag{3}$$

$$T_t^* = \arg\max_{T_t} \sigma_W^2 \tag{4}$$

onde:

- σ_W^2 é a variância calculada;
- ω_b é a probabilidade de um píxel pertencer ao *background*;
- ω_f é a probabilidade de um píxel pertencer ao *foreground*;
- μ_b é a média da classe de *background*;
- μ_f é a média da classe de *foreground*;
- μ_t é a média total ;

Para uma imagem formada em escala de cinza, com intensidade que varia de [0,L], o número de píxeis com determinado valor i é denotado por n_i. O histograma desta imagem pode ser então representado por:

$$n_i = n_0, n_1, n_2, ...n_L \tag{5}$$

Somando todos estes níveis, tem se o total de píxeis, dado por:

$$M = n_0 + n_1 + n_2 + ...n_L \tag{6}$$

A probabilidade de um píxel possuir determinado nível de cinza i é calculado por:

$$p_i = \frac{n_i}{M},\ p_i \leq 1,\ \sum_{i=1}^{L} p_i = 1 \tag{7}$$

Sendo ω_b a probabilidade de um píxel pertences ao *background* e ω_f a probabilidade de um píxel pertencer ao *foreground*, pode-se calcular a probabilidade de cada classe por:

$$\omega_f = \sum_{i=T_t+1}^{L} p_i = \omega(T_t) \tag{8}$$

$$\omega_b = \sum_{i=1}^{T_t} p_i = 1 - \omega(T_t) \tag{9}$$

a média μ de cada classe:

$$\mu_f = \sum_{i=T_t+1}^{L} \frac{ip_i}{\omega_f} = \omega(T_t) \tag{10}$$

$$\mu_b = \sum_{i=1}^{T_t} \frac{ip_i}{\omega_b} = 1 - \omega(T_t) \tag{11}$$

e a média total μ_t

$$\mu_t = \sum_{i=1}^{T_t} ip_i \tag{12}$$

As variâncias de cada classe são calculadas por:

$$\sigma_f^2 = \sum_{i=T_t+1}^{L} (i - \mu_f)^2 \frac{p_i}{\omega_f} \tag{13}$$

$$\sigma_b^2 = \sum_{i=1}^{T_t} (i - \mu_b)^2 \frac{p_i}{\omega_b} \tag{14}$$

4.5 Detecção dos Hotspots

Os *hotspots* são os pontos mais quentes da imagem, nos quais é analisada se a temperatura corresponde a uma situação de falha ou não. Para encontrar onde ficam esses pontos quentes da imagem termográfica, primeiro são selecionados os píxels mais quentes da imagem, os quais serão dilatados por um método recursivo até atingirem uma situação de equilíbrio segundo um critério.

A dilatação é realizada por dois motivos. Primeiramente para evitar que ruído no valor de alguns píxels possam ser considerados hotspots. A dilatação só acontecerá se a região vizinha ao píxel com valor alto também possuir valores altos, garantindo que toda a região esteja quente para ser considerada um hotspot. Em segundo lugar, a dilatação garante que um *hotspot* não seja segmentado em dois ou mais por uma distância de poucos píxels, já que se a distância é muito pequena pode-se desconsiderar estes píxels que estão separando as regiões.

Primeiramente é definida a maior temperatura:

$$T_{max} = \max_{0 \leq x < W, 0 \leq y < H} \gamma(x, y) \tag{15}$$

E os píxeis com a maior temperatura são dados por:

$$\Omega_0(x, y) = \begin{cases} 1 & \text{se } \alpha(x, y) = T_{max} \\ 0 & \text{caso contrário} \end{cases} \forall 0 \leq x < W, 0 \leq y < H \tag{16}$$

sendo:
W é a largura da imagem.
H é a altura da image.
α é a imagem original.
x e y são valores que indicam a posição do píxel.

É definida uma matriz de restrição C, utilizada para restringir o processo de dilatação:

$$C(x, y) = \begin{cases} 1 & \text{se } \gamma(x, y) \geq T_{hotspot} \\ 0 & \text{caso contrário} \end{cases} \forall 0 \leq x < W, 0 \leq y < H \tag{17}$$

A equação 18 representa o processo de dilatação, onde um ω_{k-1} é aumentado 1 píxel em todas as direções a cada iteração. O processo só é finalizado quando $\Omega_k(x,y) = \Omega_{k-1}(x,y)$, definido como $\Omega^*(x,y)$. Isto só ocorre quando a o incremento não faz intersecção com os valores de C.

$$\Omega_k(x,y) = (\Omega_{k-1}(x,y) \oplus B) \cap C,\ k = 1,2,3,... \tag{18}$$

sendo: B é uma máscara com os 8 píxels vizinhos iguais a 1.

A temperatura $T_{hotspot}$ é o valor de temperatura que delimita os *hotspots*. Definiu-se empiricamente que um píxel pertence a um *hotspot* caso seu valor esteja abaixo da temperatura máxima por até 60% da diferença de temperaturas do *foreground*.

$$T_{hotspot} = T_{max} - 0.6(T_{max} - T_{min}) \tag{19}$$

Cada conjunto de píxeis interligados formando um *hotspot* é denominado de A_i, onde i = 1...N_h, sendo N_h o número de *hotspots*.

Com o conjunto de *hotspots*, é calculada a área de cada um deles com a seguinte expressão:

$$D_i = \sum_{x=0}^{W} \sum_{y=0}^{H} A_i(x,y) \tag{20}$$

Neste ponto, são avaliadas as áreas de cada *hotspot* e descartados os que apresentam área abaixo de certo limiar, 5 píxels por exemplo, já que *hotspots* com área muito pequena geralmente são ruídos na imagem.

Fazendo-se a média da temperatura de cada píxel, encontramos o valor de temperatura de determinado *hotspot*. Este procedimento deve ser realiza para cada *hotspot* a ser analisado.

$$T_{hot} = \frac{1}{D_i}(\sum_{x=0}^{W-1} \sum_{y=0}^{H-1} \gamma(x,y), \forall(x,y) \in A_i) \tag{21}$$

4.6 Cálculo da temperatura de referência

Visto que a temperatura geral do cabo pode alterar de acordo com as condições climáticas, não é possível utilizar uma temperatura fixa como referência, já que isto poderia acarretar em falhas no diagnóstico.

Desta forma, define-se a temperatura de referência pela temperatura restante do material energizado desconsiderando os *hotspots*, ou seja, é o conteúdo do *foreground* sem os *hotspots*.

$$\rho(x,y) = \overline{\beta(x,y) \cap \Omega^*(x,y)} \tag{22}$$

O somatório dos valores de temperatura da imagem γ desconsiderando os hotspot da imagem Ω^* é dado por:

$$Mref = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} \gamma(x,y),\ se\ \rho(x,y) = 1 \tag{23}$$

Nref é a quantidade de píxels do foreground desconsiderando o *hotspot*.

$$Nref = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} (\rho(x, y)) \tag{24}$$

Finalmente, a temperatura de referência é dada pela média:

$$T_{ref} = \frac{Mref}{Nref} \tag{25}$$

4.7 Classificação dos Hotspots

Para classificar a situação do *hotspot*, é feita uma análise quantitativa dos valores obtidos.

Para simplificar o processo de detecção, neste trabalho decidiu-se avaliar apenas o hotspot de maior área.

Primeiramente é calculado o percentual da diferença de temperatura entre o *hotspot* selecionado e a referência.

$$\Delta T = \frac{T_{hot} - T_{ref}}{T_{ref}} \tag{26}$$

Com este valor, o hotspot pode ser classicada de acordo com a tabela 1 (CHOU; YAO, 2009).

Tabela 1: Condições de Medidas.

Condição	Variação
Normal	$\Delta T_r < 9\%$
Atenção	$9\% \leq \Delta T_r < 90\%$
Anormal	$90\% \leq \Delta T_r$

4.8 Otimização do algoritmo

Durante a execução dos testes na placa, percebeu-se que o sistema estava levando um tempo demasiadamente grande para processar cada frame. O processo de busca pelo maior valor é realizado de forma exaustiva pelo algoritmo, processando a variância para cada valor do histograma e verificando qual o maior. Como isto deve ser feito pra 65535 valores correspondentes aos 16 bits de dados que a câmera termográfica envia, e a para cada um destes valores devem ser calculadas as médias e variâncias de cada elemento este processo pode levar certo tempo.

Para reduzir este tempo, usou-se uma matriz auxiliar onde os dados da câmera foram divididos por 255. O histograma desta forma foi reduzido 255 vezes. O cálculo foi então realizado para descobrir o valor que maximiza a variância para este conjunto de dados truncados, em torno do qual se encontra o valor exato.

A partir do valor obtido, utilizou-se a matriz original, percorrendo 128 pontos para cada lado da temperatura descoberta anteriormente, para calcular em qual valor a variância é maximizada.

5 TECNOLOGIAS PARA TRANSMISSÃO MULTIMÍDIA

De forma a tornar possível a transmissão dos dados recebidos e tratados pelo módulo embarcado, torna-se necessário o uso de algum padrão de comunicação que define os protocolos a serem utilizados.

O modelo OSI (Open system Interconnection) é um modelo utilizado para definir como uma pilha de protocolos deve ser implementada (FOROUZAN, 2006). As camadas nas quais este modelo divide a comunicação podem ser vistas na figura 6.

Figura 6: Modelo OSI.

A partir do conhecimento destas camadas, pode-se fazer um estudo das possíveis combinações de protocolos e tecnologias que podem ser usadas em cada camada de forma a possibilitar a comunicação.

As duas camadas mais inferiores, física e enlace, são definidas por padrões como o Ethernet IEEE 802.3 para conexões elétricas ou padrões sem fio como o IEEE 802.11 (IEEE, 1997). O uso de uma tecnologia sem fio seria mais adequada para a transmissão dos dados, já que o robô que fará a inspeção estará isolado em campo.

Para a camada de rede, existe o protocolo Internet Protocol (IP). O IP define uma forma de transmitir blocos de dados chamados de datagramas de uma fonte a um destino, onde fonte e destino são dispositivos identificados por um endereço de tamanho fixo POSTEL. O protocolo IP é um protocolo bastante difundido, e há inúmeros dispositivos que

dão suporte a esta tecnologia hoje em dia, o que garante que a comunicação desenvolvida pode ser facilmente adaptada para diversos canais de transmissão cabeados ou sem fio.

Acima da camada de rede no modelo OSI, encontra-se a camada de transporte. Nesta camada pode-se citar dois protocolos importantes: TCP e UDP.

O User Datagram Protocol (UDP), fornece um meio de comunicação com um mecanismo mínimo implementado pelo protocolo. Este protocolo é orientado por transação e não garante a entrega nem a não duplicidade dos pacotes (POSTEL, 1980).

O UDP possui as seguintes características principais:

- Não garante a entrega dos datagramas
- Não garante a ordem em que os datagramas são entregues
- Não possui nenhuma proteção contra datagramas duplicados
- Não garante a integridade do conteúdo transmitido

O Transmission Control Protocol (TCP) é um protocolo destinado a comunicação com alto grau de confiabilidade entre dispositivos em uma rede (POSTEL, 1981b). O TCP implementa um controle por número de sequência dos pacotes e um sistema de retransmissão de pacotes perdidos, garantindo a entrega segura dos dados.

Pode-se citar as seguintes características para o protocolo TCP:

- Retransmissão de dados perdidos
- Garantia da ordem correta dos dados
- Controle de fluxo
- Controle cotra pacotes duplicados

O TCP, diferentemente do UDP, necessita que seja estabelecida uma conexão entre os dispositivos antes de iniciar a comunicação. Todos estes controles exigem um tempo maior para a transmissão dos dados. Desta forma, troca-se desempenho por uma maior confiabilidade.

5.1 Protocolos para transmissão em tempo real

Como visto anteriormente, o UDP é o protocolo mais rápido e adequado para transmissões em tempo real, porém o mesmo não possui nenhum controle de tempo de envio e de entrega. Transmissões de vídeo e de áudio exigem que seja feita a reconstrução temporal dos dados recebidos, e isso só é possível se for enviada alguma informação de tempo com os dados.

5.1.1 RTP

O Real Time Protocol (RTP), é um protocolo de transmissão de dados que funciona acima do protocolo UDP, adicionando informações temporais e número de sequencia para que seja possível fazer a reconstrução dos dados na ordem e tempo corretos. A figura 7 demonstra o encapsulamento completo dos dados com todos os protocolos envolvidos.

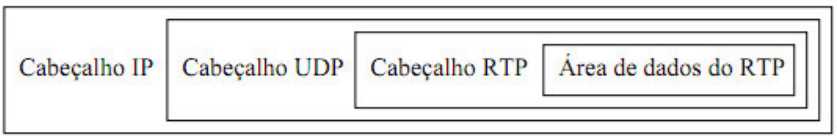

Figura 7: Encapsulamento dos pacotes RTP.

O fato do RTP utilizar o procolo UDP na camada de transporte torna possível a transmissão dos dados de uma fonte para mais de um destinatário ao mesmo tempo, conhecida como *multicast*.

O cabeçalho de um pocote RTP pode ser visto na figura 8 (GROUP et al., 1996), a partir do qual será feita a análise das informações contidas.

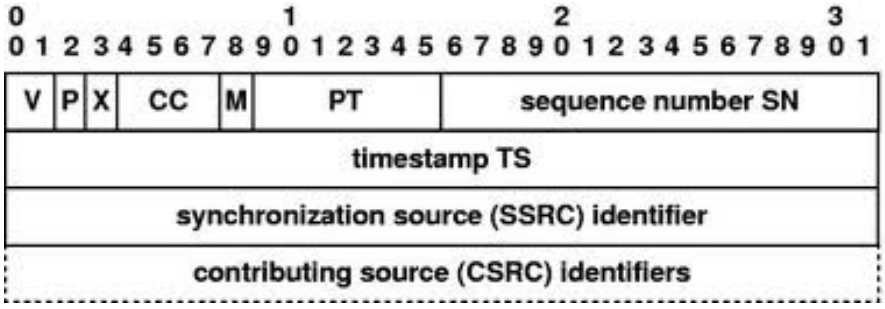

Figura 8: Cabeçalho de um pacote RTP.

Pode-se destacar algumas informações contidas no cabeçalho dos pacotes RTP

- Synchronization source (SSRC) identifier

  O protocolo RTP pode ser usado para transmitir múltiplos fluxos ao mesmo tempo, de áudio e vídeo por exemplo. A função deste campo é de identificar cada fluxo com um número aleatório, comum a todos os pacotes do mesmo fluxo, tornando possível que o destinatário possa distinguir os pacotes entre um fluxo e outro.

- Timestamp

  O selo de tempo oferece informações para que seja feita a correta sincronização dos dados pelo destinatário. Visto que não há garantia do tempo de entrega em uma rede TCP/IP, é necessário que sejam enviadas informações do ritmo que estes dados devem ser usados pelo receptor. Quando há mais de um fluxo de dados, áudio e vídeo por exemplo, este *timestamp* também é usado para sincronizar os dois.

- Sequence number

  Visto que o protocolo UDP não garante que os datagramas cheguem ao destinatário na ordem correta, o campo de número de sequencia é utilizado para transmitir a informação da ordem em que os dados devem ser utilizados. Com esta informação também é possível saber a quantidade de datagramas perdidos na transmissão.

5.1.2 RTSP

O Real Time Streaming Protocol (RTSP), é um protocolo do nível de aplicação responsável por controlar a entrega de dados com propriedades de tempo real (SCHULZRINNE; A.RAO; R.LANPHIER, 1998).

O motivação para a criação deste protocolo é que o usuário precisa ter certo controle sobre o vídeo que está assistindo (FOROUZAN, 2006). Desta forma, este protocolo implementa controles para parar e recomeçar a transmissão dos dados multimídia.

Todo servidor RTSP deve suportar obrigatoriamente os seguintes métodos:

- `OPTIONS`

  Quando o servidor recebe uma solicitação com este método, deve responder com a lista de todos os métodos implementados no servidor.

- `SETUP`

  É um método utilizado para definir os parâmetros da transmissão, como o protocolo a ser utilizado, as portas, se será *unicast* ou *multicast*, entre outros.

- `PLAY`

  Método utilizado para sinalizar ao servidor que o mesmo deve iniciar o transmissão do fluxo.

- `TEARDOWN`

  Uma requisição de `TEARDOWN` faz com que o servidor para a transmissão do fluxo. É equivalente ao parar de um controle remoto.

Os seguintes métodos também são recomendados:

- `DESCRIBE`

  Solicita a descrição de todos as sessões, comumente retornadas pelo protocolo SDP.

- `PAUSE`

  Solicita a pausa na transmissão do fluxo de dados.

Nos pacotes RTSP os dados são todos enviados em caracteres ASCII, e o servidor basicamente responde com códigos pré-definidos às solicitação enviadas pelo cliente. Uma diferença importante entretanto, é que o servidor guarda informação quanto ao estado em que o cliente utilizando tal sessão se encontra, por exemplo, o servidor sabe quando o cliente está em um estado de inicialização, de apresentação ou de pausa KUROSE.

A figura 9 ilustra a máquina de estados do protocolo RTSP.

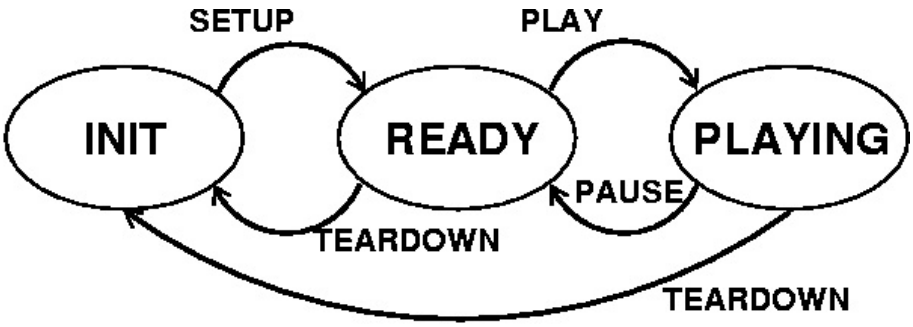

Figura 9: Máquina de estados do protocolo RTSP.

6 BIBLIOTECAS RTSP

De forma a fazer a recepção, controle e envio dos pacotes RTP e RTSP, optou-se por utilizar uma biblioteca que implementa classes e métodos para o correto tratamento destes protocolos.

Alguns exemplos de bibliotecas para protocolo RTSP são:

- libnemesi
- libcurl
- librtsp
- gst-rtsp
- live555

Algumas das bibliotecas acima, como a `libnemesi` e a `libcurl` implementam apenas as funções de cliente RTSP. Neste projeto porém, deseja-se desenvolver um servidor RTSP para o envio das streams. O projeto `librtsp` é um projeto mais novo, que ainda está em fase de testes. Não há referências de nenhum software conhecido que utilize esta biblioteca. Assim, a biblioteca escolhida foi a Live555, a qual é utilizada por software de vídeo como VLC e Mplayer, o que indica que a mesma está estável.

O projeto Live 555 é um conjunto de bibliotecas para tratar *streams* multimídia usando protocolos abertos como RTP, RTSP, SIP. As bibliotecas tem suporte aos mais diversos sistemas operacionais (LIVE555, 2009) como Linux, Mac OS, Windows.

As seguintes bibliotecas fazem parte do projeto:

- `UsageEnvironment`

  Esta biblioteca implementa duas classes principais: `UsageEnviroment` e `TaskScheduler`. Estas classes são usadas para programar eventos, relacionar funções de tratamentos de sinais e fazer a interface de saída para as mensagens de erro.

- `Groupsock`

  A biblioteca `GroupSock` implementa classes que fazem o encapsulamento dos datagramas.

- `LiveMedia`

  A biblioteca `LiveMedia` é a mais extensa de todas e é onde estão definidas todas as classes que fazem o tratamento dos mais diversos formatos de mídia suportados.

Existem duas classes nesta biblioteca responsáveis pelo tratamento dos pacotes RTSP, uma para aplicação em servidores e outra para clientes. A classe `RTSPclient` envia os comandos de `SETUP`, `PLAY`, `TEARDOWN` entre outros suportados pelo protocolo para fazer o controle da sessão. No caso da classe `RTSPserver`, uma máquina de estados do protocolo RTSP é implementada e são tratadas todas as solicitação requisitadas pelos clientes.

As classe `MediaSession` fornece os métodos necessários para iniciar uma nova sessão, de acordo com os dados vindos nos pacotes SDP. Uma `MediaSession` deve ser então relacionada a uma `MediaSubSession`, de acordo com o tipo de mídia que a sessão irá tratar.

Existem ainda uma série de classes classificadas como *sources* e *sinks*, responsáveis por tratar os dados de mídia vindos nos pacotes RTP. As classes do tipo *source* são classificadas assim pois implementam funções que provem dados de alguma fonte. Já as classes do tipo *sink* são responsáveis por despachar estes dados para um certo destino. No caso de uma aplicação cliente, pode-se usar por exemplo a classe `RTPSource` como fonte dos dados oriundos dos pacotes RTP e usar a classe `FileSink` para enviar estes dados a um arquivo para salvar a mídia.

- `BasicUsageEnvironment`

  Esta biblioteca define uma implementação concreta da classe `UsageEnviroment` para aplicações que são executadas em console.

6.1 Exemplos de uso

A complexidade de implementação de um cliente ou servidor RTSP é reduzida quando se utiliza uma biblioteca que dê suporte a este protocolo, porém, ainda assim, é necessário entender a estrutura da biblioteca para desenvolver um aplicativo utilizando a mesma.

O uso de exemplos ajuda no entendimento do uso das funções, desta forma, serão apresentados um exemplo de cliente RTSP e um exemplo de servidor RTSP:

6.1.1 Servidor RTSP

Será mostrado a seguir um exemplo de servidor RTSP para transmitir streams de vídeo no formato MPEG4.

1. Criar um ambiente a partir da classe `Environment`.
2. Criar `GroupSock` para RTP e RTCP.
3. Instanciar um `VideoRTPSink`.
4. Instanciar a classe `RTSPServer`.
5. Criar uma sessão e uma sub-sessão do formato de mídia a ser enviado.
6. Gerar um `Source` utilizando alguma fonte, um arquivo por exemplo.
7. Iniar a reprodução do `VideoRTPSink`

```
1
2  #include "liveMedia.hh"
3  #include "BasicUsageEnvironment.hh"
4  #include "GroupsockHelper.hh"
5
6  UsageEnvironment* env;
7  char const* inputFileName = "test.m4e";
8  MPEG4VideoStreamFramer* videoSource;
9  RTPSink* videoSink;
10
11 void play(); // forward
12
13 int main(int argc, char** argv) {
14
15   //Configura um environment
16   TaskScheduler* scheduler = BasicTaskScheduler::createNew();
17   env = BasicUsageEnvironment::createNew(*scheduler);
18
19   // Cria 'groupsocks' para RTP e RTCP:
20   struct in_addr destinationAddress;
21   destinationAddress.s_addr = chooseRandomIPv4SSMAddress(*env);
22
23   //Configura as portas RTP e RTCP
24   const unsigned short rtpPortNum = 18888;
25   const unsigned short rtcpPortNum = rtpPortNum+1;
26   const unsigned char ttl = 255;
27
28   const Port rtpPort(rtpPortNum);
29   const Port rtcpPort(rtcpPortNum);
30
31   Groupsock rtpGroupsock(*env, destinationAddress, rtpPort, ttl);
32   rtpGroupsock.multicastSendOnly();
33   Groupsock rtcpGroupsock(*env, destinationAddress, rtcpPort, ttl);
34   rtcpGroupsock.multicastSendOnly();
35
36   // Cria um sink para MPEG-4
37   videoSink = MPEG4ESVideoRTPSink::createNew(*env, &rtpGroupsock, 96);
38
39   // Cria e inicia o sink RTCP for this RTP sink:
40   const unsigned estimatedSessionBandwidth = 500;
41   const unsigned maxCNAMElen = 100;
42   unsigned char CNAME[] = "PLIR_RTSP_SERVER";
43   RTCPInstance* rtcp
44   = RTCPInstance::createNew(*env, &rtcpGroupsock,
45                             estimatedSessionBandwidth, CNAME,
46                             videoSink, NULL,
47                             True);
48
49   RTSPServer* rtspServer = RTSPServer::createNew(*env, 8554);
50
51   ServerMediaSession* sms
52     = ServerMediaSession::createNew(*env, "testStream", inputFileName,
53                     "Sessao RTSP", True);
54   sms->addSubsession(PassiveServerMediaSubsession::createNew(*videoSink
       , rtcp));
55   rtspServer->addServerMediaSession(sms);
56
```

```
57    char* url = rtspServer->rtspURL(sms);
58    delete[] url;
59
60    // Inicia o envio:
61    *env << "Enviando...\n";
62    play();
63
64    //Recomeça outra vez
65    env->taskScheduler().doEventLoop();
66
67    return 0;
68  }
69
70  void afterPlaying(void* /*clientData*/) {
71    *env << "...fim do arquivo\n";
72
73    Medium::close(videoSource);
74
75    // Recomeça o envio
76    play();
77  }
78
79  void play() {
80    // Abre o arquivo como um stream de bytes
81    ByteStreamFileSource* fileSource = ByteStreamFileSource::createNew(*
        env, inputFileName);
82
83          FramedSource* videoES = fileSource;
84
85          // Cria um framer para MPEG4
86          videoSource = MPEG4VideoStreamFramer::createNew(*env, videoES);
87
88          // Inicia o vídeo
89          *env << "Lendo arquivo...\n";
90
91          videoSink->startPlaying(*videoSource, afterPlaying, videoSink);
92  }
```

6.1.2 Cliente RTSP

A seguir será mostrado o exemplo de um cliente RTSP para MPEG2.

O processo para criar um cliente RTSP é bastante semelhante a criação do servidor, porém, os *sources* e *sinks* são trocados.

```
1   #include "liveMedia.hh"
2   #include "GroupsockHelper.hh"
3   #include "BasicUsageEnvironment.hh"
4
5   void afterPlaying(void* clientData); // forward
6
7   struct sessionState_t {
8     RTPSource* source;
9     MediaSink* sink;
10    RTCPInstance* rtcpInstance;
11  } sessionState;
12
13  UsageEnvironment* env;
14
```

```
15 int main(int argc, char** argv) {
16   // Criando um environment:
17   TaskScheduler* scheduler = BasicTaskScheduler::createNew();
18   env = BasicUsageEnvironment::createNew(*scheduler);
19
20   // O saída será a saída padrão 'stdout':
21   sessionState.sink = FileSink::createNew(*env, "stdout");
22
23   // Criar um groupsock para RTP and RTCP:
24   char const* sessionAddressStr = 0.0.0.0
25   const unsigned short rtpPortNum = 8888;
26   const unsigned short rtcpPortNum = rtpPortNum+1;
27
28   struct in_addr sessionAddress;
29   sessionAddress.s_addr = our_inet_addr(sessionAddressStr);
30   const Port rtpPort(rtpPortNum);
31   const Port rtcpPort(rtcpPortNum);
32
33   Groupsock rtpGroupsock(*env, sessionAddress, rtpPort, ttl);
34   Groupsock rtcpGroupsock(*env, sessionAddress, rtcpPort, ttl);
35
36   // Cria uma fonte de dados RTP para MPEG2
37   sessionState.source = MPEG1or2VideoRTPSource::createNew(*env, &
        rtpGroupsock);
38
39   const unsigned estimatedSessionBandwidth = 160; // in kbps; for RTCP
        b/w share
40   const unsigned maxCNAMElen = 100;
41   unsigned char CNAME[maxCNAMElen+1];
42   gethostname((char*)CNAME, maxCNAMElen);
43   CNAME[maxCNAMElen] = '\0'; // just in case
44   sessionState.rtcpInstance
45     = RTCPInstance::createNew(*env, &rtcpGroupsock,
46                               estimatedSessionBandwidth, CNAME,
47                               NULL, sessionState.source);
48
49   // Inicia o recebimento:
50   *env << "Recebendo stream...\n";
51   sessionState.sink->startPlaying(*sessionState.source, afterPlaying,
        NULL);
52
53   //Fica em loop
54   env->taskScheduler().doEventLoop();
55
56   return 0;
57 }
58
59 void afterPlaying(void* /*clientData*/) {
60   *env << "...fim\n";
61
62   Medium::close(sessionState.rtcpInstance);
63   Medium::close(sessionState.sink);
64   Medium::close(sessionState.source);
65 }
```

7 IMPLEMENTAÇÃO

Todo o estudo dos protocolos e das bibliotecas feitos neste trabalho culminaram com o desenvolvimeto de um cojunto de aplicativos que juntos realizam as tarefas de receber os dados das câmeras, fazer as devidas análises, além de gerar e enviar os resultados para um computador cliente. Este capítulo detalhará os aplicativos desenvolvidos que rodam na placa BeagleBoard.

7.1 Organização do software

O *software* desenvolvido foi dividido em três aplicativos inter-dependentes que são executados em *background*. Os três aplicativos foram nomeados de:

- *Receiver*: Responsável pela recepção dos dados da câmera FLIR A320.
- *Detector*: Responsável pela aquisição dos frames provenientes do *Receiver*, análise dos dados e geração dos frames a serem enviados.
- *Transmitter*: Responsável pelo envio dos frames gerados pelo *Detector*.

Um outro componente a ser citado é o USB Video Class Driver, um driver compatível com uma grande variedade de câmeras USB que foi utilizado para fazer a ponte entre o hardware da câmera USB e o software *Detector*, o que utiliza os frames da câmera USB na formação das imagens a serem enviadas aos computadores clientes.

O sistema foi dividido em três aplicativos para funcionar de forma modular, sendo possível substituir os aplicativos `Receiver` e `Detector` por outros aplicativos semelhantes que recebam ou transmitam de outra forma, sem precisar para isso alterar o aplicativo principal de detecção.

O fluxo dos dados entre estes aplicativos pode ser melhor visualizado na figura 10.

7.2 Comunicação entre processos

Como foi visto na seção anterior, o fluxo de dados passa entre 3 processos diferentes responsáveis por funções específicas. A comunicação e transferência de dados entre estes processos não é tão simples como a comunicação interna de um processo apenas, já que neste caso, os dados de cada processo não são diretamente acessíveis pelos demais processos, e um mecanismo de comunicação entre processos deve ser utilizado.

Pode-se citar 4 principais métodos de comunicação entre processos, os quais foram avaliados durante o desenvolvimento deste projeto:

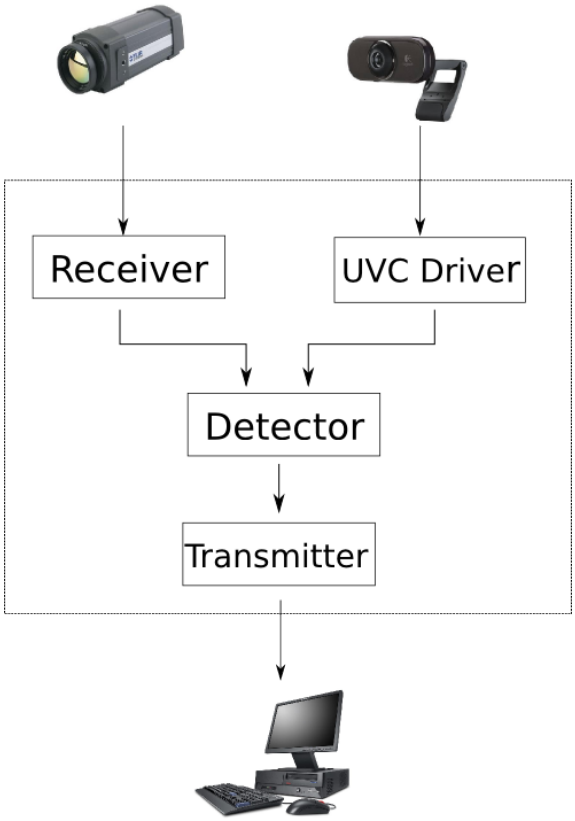

Figura 10: Fluxo global dos dados.

- Socket
- Sinais
- Fila de mensagem
- Memória compartilhada

O uso de sinais foi descartado visto que com este método não é possível transferir dados em si, apenas sinalizadores entre os processos, e na aplicação em questão, deve haver a transferência de frames de dados, inviabilizando o uso deste método.

O método por fila de mensagens é geralmente utilizado para comunicação assíncrona entre processos, onde um processo deve enviar um conjunto de dados para outro sem que haja uma dependência entre o ritmo que os dados são processados pelo processo que enviou e o processo que está recebendo as mensagens. Neste projeto, deve haver uma sincronia entre o recebimento dos frames, o processamento dos mesmos e o envio para o processo de transmissão.

Os métodos ainda disponíveis, via socket e memória compartilhada poderiam ser usados para realizar a comunicação desejada. Porém, avaliando do ponto de vista de desempenho, sabe-se que sockets demandam um processamento muito maior do sistema operacional, visto que a comunicação é feita através de protocolos TCP e UDP, os quais requerem que os dados sejam encapsulados e desencapsulados durante a comunicação. Memória compartilhada por sua vez, não requer maiores processamentos por parte do

sistema operacional durante a comunicação, apenas durante a inicialização da região de memória a ser compartilhada.

Tendo em vista os pontos apresentados acima, o método de comunicação escolhido foi o de utilizar memória compartilhada.

De forma a possibilitar a transferência dos dados mantendo uma sincronia entre os estados dos diferentes aplicativos, uma estrutura de variáveis a serem compartilhadas foi definida. A estrutura contém os seguintes campos:

- SemId: Identificador do semáforo utilizados para sincronizar a leitura e escrita por diferentes processos.

- Data: Dados a serem transferidos.

7.3 Complementação da biblioteca liveMedia

A biblioteca liveMedia é uma das bibliotecas que formam o projeto live555, e é a biblioteca que contém todas as classes que tratam do envio e recebimento de streams utilizando o protocolo RTSP. Como explicado anteriormente, as classes podem ser classificadas em *sources* ou *sinks*, de acordo com o papel da classe no fluxo dos dados.

A implementação original desta biblioteca dispõe somente de classes que utilizam arquivos como fonte ou destino final dos streams, sendo uma opção bastante limitada para o uso em paralelo com outros aplicativos. Como comentado na seção anterior, a forma escolhida para a comunicação entre processos foi utilizando memória compartilhada, o que demandou a criação de novas classes na biblioteca *liveMedia* para suportar este modelo de comunicação.

Há uma cadeia de classes que tem seus métodos chamados umas pelas outras na biblioteca *liveMedia*, e as classes responsáveis pela leitura ou gravação dos arquivos de origem ou destino das streams sempre ficam na ponta desta cadeia. Esta cadeia formada com as classes gerou um padrão quanto a nomenclatura dos métodos que cada classe implementa, e este padrão deve ser respeitado na criação de novas classes para que as mesmas sejam compatíveis com as outras classes da biblioteca original.

Uma das novas classes criadas é a classe *ShmSink*, nome derivado de *Shared Memory Sink*. Esta nova classe é equivalente a classe *FileSink* presenta na biblioteca original, e possui os mesmos métodos desta.

A classe *ShmSink* é utilizada pelo aplicativo de recebimento da stream proveniente da câmera termográfica. Ao invés de salvar os dados em um aquivo como é feito utilizando a classe *FileSink*, os dados são transferidos ao aplicativo de detecção via memória compartilhada.

O aplicativo de transmissão também deve utilizar memória compartilhada, porém para receber os dados, ou seja, um classe do tipo *source* deve ser desenvolvida com suporte a este tipo de comunicação entre processos. Para tal, foi criada a classe nomeada de *ByteStreamShmSource*, a qual é equivalente a sua versão para leitura direta de arquivos chamada de *ByteStreamFileSource*. A classe *ByteStreamShmSource* é utilizada pelo aplicativo de transmissão.

Estas novas classes são compiladas juntamente com os aplicativos, não sendo necessário recompilar toda a biblioteca *liveMedia*. A biblioteca permanece inalterada desta forma, garantindo a compatibilidade do aplicativo com a biblioteca original.

7.4 Receiver

O aplicativo *Receiver* tem a função de receber a stream de dados provenientes da câmera termográfica e enviar os dados para o aplicativo de detecção utilizando a classe desenvolvida *ShmSink*.

Este aplicativo é responsável por iniciar a sessão RTSP com a câmera, selecionando o tipo de mídia adequado, neste caso, o envio de frames de dados não comprimidos, e enviando todos os pacotes RTP para o aplicativo de detecção. Os dados passados desta forma ainda possuem todo o cabeçalho de um pacote RTP com dados não comprimidos, o qual é definido pela RFC4175, seguido dos dados da imagem em si. Não existe a correspondência de um pacote RTP para cada frame, e o aplicativo de detecção deve concatenar os dados de diversos pacotes até formar um frame completo a ser analisado.

7.5 Detector

O aplicativo *Detector* é o principal aplicativo desenvolvido, no qual é realizada a análise das imagens provenientes da câmera termográfica. Este aplicativo ainda implementa um *frame grabber* de imagens da câmera USB, utilizadas para obter uma imagem no espectro visível da falha. O conjunto de resultado obtidos é então utilizado para formar uma imagem com os principais dados da análise, a qual será transmitida a um computador cliente.

Uma das premissas durante o desenvolvimento do projeto foi de não precisar instalar nenhum software no computador cliente para obter as detalhes do módulo de inspeção. Desta forma, decidiu-se por enviar os dados no formato de vídeo, o qual pode ser facilmente visualizado no computador cliente utilizado um *player* de vídeo com suporte a RTSP, recurso oferecido pela maioria dos *players*.

Durante o desenvolvimento do projeto, foi de suma importância a visualização das imagens processadas, antes mesmo da transmissão, como forma de depurar problemas. Para tal, foi utilizada a biblioteca GTK para criar uma interface gráfica ao aplicativo, onde são mostradas informações durante o processamento. A biblioteca GTK foi escolhida por ser a biblioteca utilizada no desenvolvimento do ambiente gráfico XFCE, compilado junto no sistema instalado na BeagleBoard para este projeto. O XFCE por sua vez, foi escolhido devido a sua simplicidade e desempenho, característica importantes visto que o sistema em questão têm limitações de processamento.

O aplicativo Detector é o que apresenta maior complexidade devido ao número de diferentes tarefas que devem ser realizadas por ele. Para facilitar o desenvolvimento, foram desenvolvidas diversas classes organizadas em uma hierarquia as quais são utilizadas por este aplicativo. O diagrama apresentado na figura 11 representa esta hierarquia de classes.

É importante salientar que os blocos no diagrama de classes nomeados de *Receiver* e *Transmitter* correspondem a classes, e não devem ser confundidos com os aplicativos de mesmo nome. Estas classes recebem o mesmo nome dos aplicativos justamente porque estão diretamente ligadas com os mesmos, fornecendo uma interface entre os 3 aplicativos.

Como mostra o diagrama de classes, todas as classes secundárias são instanciadas pela classe principal *Detect*. Esta classe é o núcleo do aplicativo de detecção, onde são concentrados todos os dados recebidos ou gerados pelas outras classes. Pode-se citar as principais funções da classe *Detect*.

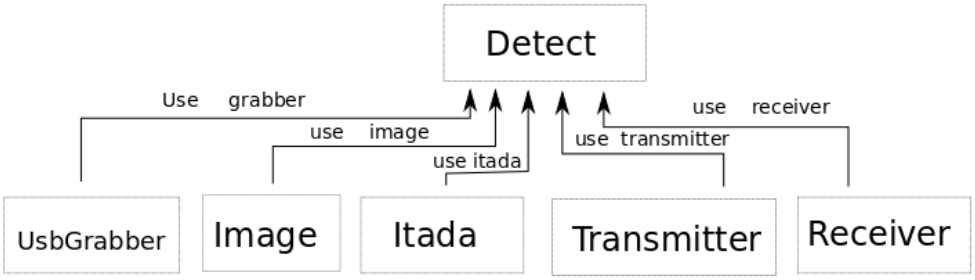

Figura 11: Hierarquia de classes do aplicativo *Detector*.

- Gerar instâncias de todas as classes secundárias.
- Armazenar os frames provenientes das classe Receiver, a qual está conectada via memória compartilhada com o software de recebimento.
- Chamar os métodos das classes de processamento dos frames na ordem correta.
- Sincronizar o recebimento, processamento e envio dos frames.

Cada uma das classes secundárias exerce uma função distinta e serão melhor descritas nas seções abaixo:

7.5.1 USBGrabber

A classe `USBGrabber` é responsável pela aquisição de frames da câmera USB, utilizando o driver UVC como interface com o hardware. No construtor da classe são configurados os parâmetros de inicialização da câmera, como a saturação, brilho, ganho e contrante da mesma, além de abrir um descritor para as leituras dos frames que são feitas por um método específico, denominado `GrabFrame`. Toda vez que o método `GrabFrame` é chamado, uma nova leitura é requisitada para o driver, e os dados do frame são armazenados em uma variável pública desta classe, disponível para outras classes acessarem.

A classe `USBGrabber` utiliza uma API proveniente do projeto UVCcapture (UVCCAPTURE, 2007) para abstrair as diversas configurações que devem ser feitas no driver relacionados com o tipo de câmera utilizado.

A câmera USB utilizada neste projeto suporta o envio de *frames* compactados em formato MPEG ou sem compactação, onde é envia o valor de luminância e crominância de cada píxel. Como será visto posteriormente com maiores detalhes, os objetos da biblioteca GTK utilizados recebem os dados em formato RGB, exigindo que algumas conversões sejam feitas.

7.5.2 Image

A classe *Image* reúne um conjunto de métodos utilizados na conversão entre os diferentes formatos de representação de imagens, como YUV, RGB e a representação em temperatura proveniente da câmera termográfica.

Algumas funções disponíveis nesta classe fazer a transformação entre os modelos de representação de cores. A representação pelo modelo de luminâncina e crominância, conhecido como YUV pode ser convertida para o modelo de representação nas 3 cores primárias, ou RGB de acordo com as expressões 27 a 29:

$$B = 1.164(Y - 16) + 2.018(U - 128) \tag{27}$$

$$G = 1.164(Y - 16) - 0.813(V - 128) - 0.391(U - 128) \tag{28}$$

$$R = 1.164(Y - 16) + 1.596(V - 128) \tag{29}$$

Esta classe também implementa métodos para a conversão dos valores de temperatura enviados pela câmera termográfica em valores que representam cores. Para isto, é utilizado um método de pseudo-colorização, utilizando uma palheta de cores, onde um determinado valor de temperatura será convertido para uma cor correspondente. Foi utilizada uma palheta com 256 cores para fazer a conversão neste projeto. A câmera termográfica envia valores de 16 bits para cada píxel, sendo possível representar.

7.5.3 Itada

A classe `Itada` é responsável por implementar o algoritmo de reconhecimento de falhas descrito anteriormente neste trabalho. O algoritmo foi implmentado passo a passo em diferentes métodos desta classe, os quais tem as suas funções descritas na tabela 2.

7.5.4 Transmitter

A classe transmitter é uma classe especial do ponto de vista de seu funcionamento. Em seu construtor, esta classe gera uma *thread*, a qual é responsável por manter a sincronia com o aplicativo de transmissão do frame gerado, que será enviado via RTSP ao computador cliente.

Visto que os dados serão transmitidos via rede Ethernet, utilizando uma camada física sem fio possivelmente, é de interesse que a taxa de transmissão do vídeo não ocupe grande parte da banda disponível, principalmente no casos em que a banda disponível é baixa. Por este motivo, resolveu-se codificar os frames em um formato de vídeo com compressão dos dados. O formato utilizado foi o MPEG4.

Para efetuar a conversão do vídeo para MPEG4 foi utilizada uma biblioteca, conhecido por `libavcodec` (LIBAV, 2009). A bibliteca `libavcodec` é utilizada por projetos bastante robustos como o *ffmpeg*, que por sua vez é utilizado por *players* de vídeo como o VLC.

Todo o processamento para converter os frames sem compactação para MPEG4 é feito por esta classe, antes da transmissão, posicionando os frames compactados em um *buffer* a ser utilizado pela *thread* de transmissão.

A *thread* de transmissão mantém uma sincronia com o aplicativo de transmissão, já que ambos utilizam a mesma memória compartilhada para se comunicar.

7.5.5 Receiver

A classe `receiver` possui um papel semelhante a classe `transmitter`, já que ambas provêem o interfaceamento entre os aplicativos Receiver com Detector e Transmitter com Detector respectivamente.

Esta classe tem também gera uma *thread* para o aplicativo, responsável pela tarefa de recebimento dos frames da câmera termográfica. A comunicação com a aplicativo Receiver é feito via memória compartilhada, como explicado anteriormente, e está *thread* é responsável por manter a sincronia com o outro aplicativo para uma comunicação efetiva.

Tabela 2: Métodos da classe `Itada`.

`Itada`	O construtor realiza a alocação dinamica de memória para armazenar os frames e as memórias temporárias utilizadas durante o processamento.
`Itada`	O destrutor da classe é responsável apenas pela liberação da memória alocada pelo construtor.
`LoadImage`	É o primeiro método que deve ser chamada a partir da inicialização da classe. É responsável por carregar o frame que se deseja processar para dentro da classe.
`GetGrayTable`	Gera o histograma com a relevancia de cada temperatura.
`GetPiTable`	Calcula a probabilidade de cada temperatura a partir do histograma.
`GetUk`	Calcula as variâncias de cada temperatura.
`CalcBestThreshold`	Este método utiliza os métodos anteriores que geram o histograma e calculam as variâncias para calcular a temperatura de *threshold*.
`ProcessFG`	Gera uma imagem com o *foreground* do frame, o qual é armazenado em uma variável para uso futuro.
`CalcTmax`	Identifica a máxima temperatura do *foreground* e armazena em uma variável global para uso futuro.
`CalcTmin`	Identifica a mínima temperatura do *foreground* e armazena em uma variável global para uso futuro.
`CalcDilate`	Processa uma iteração de dilatação dos *hotspots*.
`ProcessHotspots`	Processa os *hotspots* por completo, realizando quantas iterações forem necessárias até gerar o hostspot por completo.
`MakeSeed`	Gera as sementes dos *hotspots* para o processo de dilatação.
`MakeConstrain`	Processa a matriz de restrição que é usada pelo método que gera os *hotspots*.
`CalcArea`	Calcula a área do *hotspot*.
`CalcThot`	Calcula a temperatura média do *hotspot*.
`CalcTref`	Calcula a temperatura de referência, subtraindo o *hotspot* do *foreground* e calculando a média.
`ProcessState`	Processa o estado final do *frame*, identificando como situção normal ou de falha.

Como os dados enviados pelo aplicativo Receiver ao aplicativo Detector ainda possuem o cabeçalho do pacote RTP, está classe também é responsável por desencapsular os dados do pacote, extraindo somente os dados do frame.

7.6 Transmitter

O aplicativo *transmitter* é responsável por receber os frames a serem enviados a um computador cliente com o resultado da análise das falhas. Este frame possui as informações mais importantes obtidas durante o processo de detecção, e possibilitará que as imagens sejam recebidas em qualquer software que tenha suporte a recebimento de vídeos por RTSP.

A função do *transmitter* é receber os dados por memória compartilhada utilizando a classe *ByteStreamShmSource*, encapsular em pacotes RTP e gerenciar as sessões RTSP que serão abertas com os computadores clientes. Durante o processo de encapsular os dados em pacotes RTP, informações de tempo atual são necessárias para gerar um *timestamp*, assim como um fluxo relativamente constante de frames a serem enviados, já que a falta de frames disponíveis na fila de encapsulamento RTP ira gerar problemas na transmissão.

Grande parte das classes da biblioteca *liveMedia* utilizadas na transmissão de streams são específicas para certo tipo de mídia a ser enviada. Neste caso, foram utilizadas classes compatíveis com MPEG4, formato dos frames gerados pelo aplicativo de detecção.

8 RESULTADOS

Para validar o funcionamento do módulo, foi realizado um experimento com um cabo de linha transmissão real, no qual uma falha foi provocada propositalmente de forma mecânica, desgastando uma seção do cabo.

Uma fonte capaz de fornecer alta corrente foi utilizada para simular a operação do cabo, e um aquecimento maior foi verificado na região com falha. O cabo desgastado foi então conectado aos terminais da fonte, como mostra a figura 12.

Primeiramente, fixou-se a câmera USB acima da câmera termográfica para que as duas estivesse direcionadas para o mesmo ponto. A câmera termográfica foi posicionada de forma a visualizar o cabo. A figura 13 mostra o posicionamento da câmeras e do cabo a frente.

Finalmente, foi utilizado um roteador wireless, no qual a câmera termográfica e a BeagleBoard foram conectados. Com esta configuração, foi possível obter os dados em um computador via rede wireless. O roteador e a BeagleBoard podem ser vistos na figura 14.

Após todo o sistema ter sido posicionado, a fonte foi ligada e os aplicativos foram executados. A figura 15 apresenta o resultado visto na interface gráfica do aplicativo Detector, produzida em GTK.

O quadro superior da esquerda da figura 15 apresenta a imagem termográfica convertida utilizando uma palheta de cores. O quadro inferior da esquerda apresenta apenas o *foreground*, obtido após a primeira parte do algoritmo de detecção ser executada. O quadro superior da esquerda apresenta o maior *hotspot* da imagem, produzido após as dilatações dos pontos com maior temperatura. O quadro inferior da direito apresenta uma imagem obtido a partir da câmera USB, no espectro visível. O objeto de texto no canto inferior da interface apresenta uma mensagem do estado do sistema, que neste caso está classificado como situação anormal.

Após a verificação do correto processamento e validação do funcionamento do algoritmo, foi testado o servidor RTSP implementado pelo aplicativo `Transmitter`. Um computador na mesma rede fez a conexão utilizando o software `VLC`, um *player* de vídeo capaz de receber streams RTSP.

A visualização do stream na tela do computador cliente pode ser vista na figura 16.

Não é possível perceber na imagem mostrada, mas durante a visualização da stream, a área com problemas é marcada por um retângulo, sinalizando que uma falha foi encontrada naquela região. Esta marcação foi configurada para aparecer de forma intermitente com o objetivo de chamar a atenção do operador neste momento.

Figura 12: Ligação do cabo na fonte.

Figura 13: Posicionamento das câmeras e do cabo durante os testes.

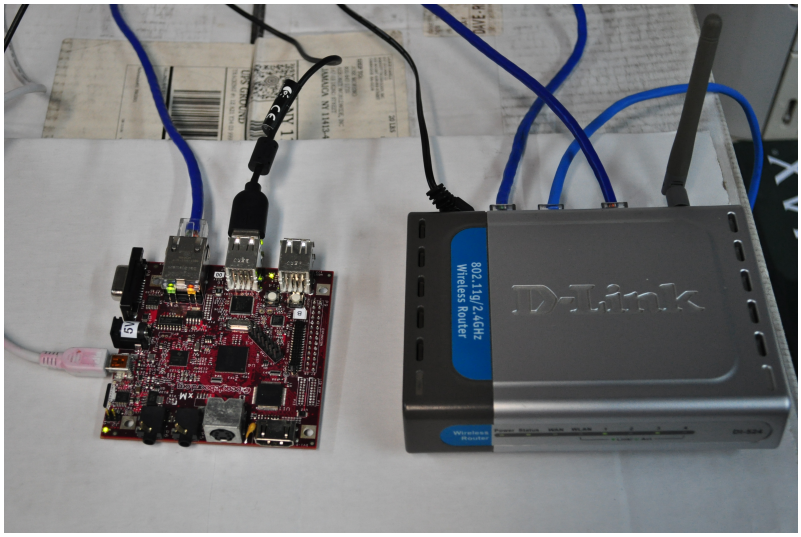

Figura 14: BeagleBoard e o roteador utilizado.

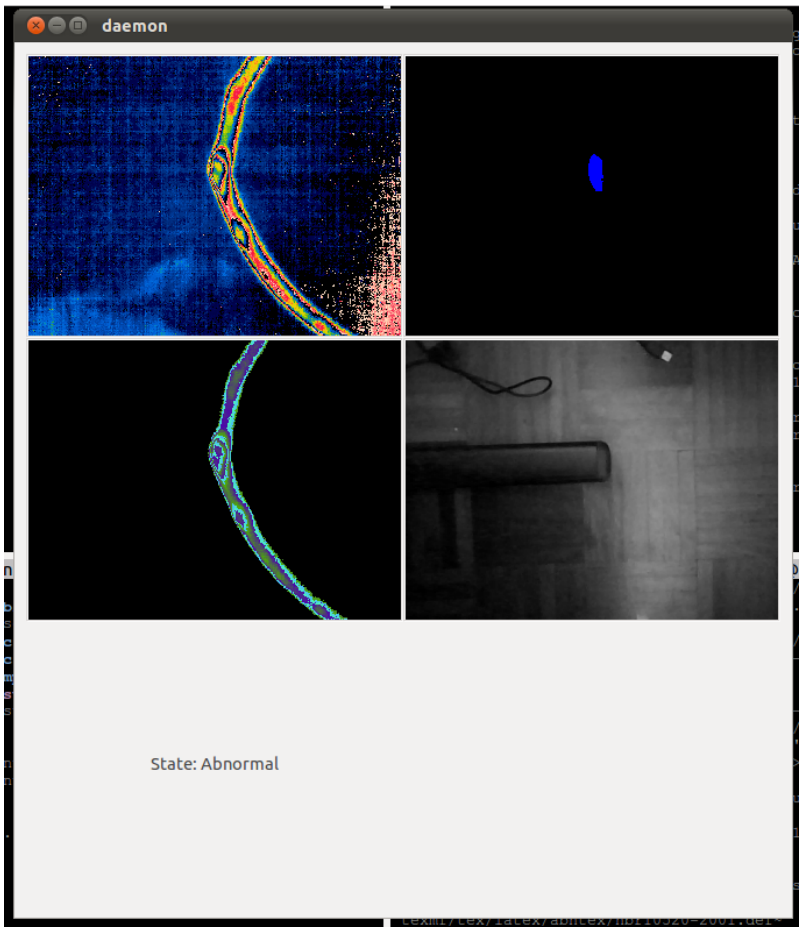

Figura 15: Interface gráfica do aplicativo *Detector* durante o teste.

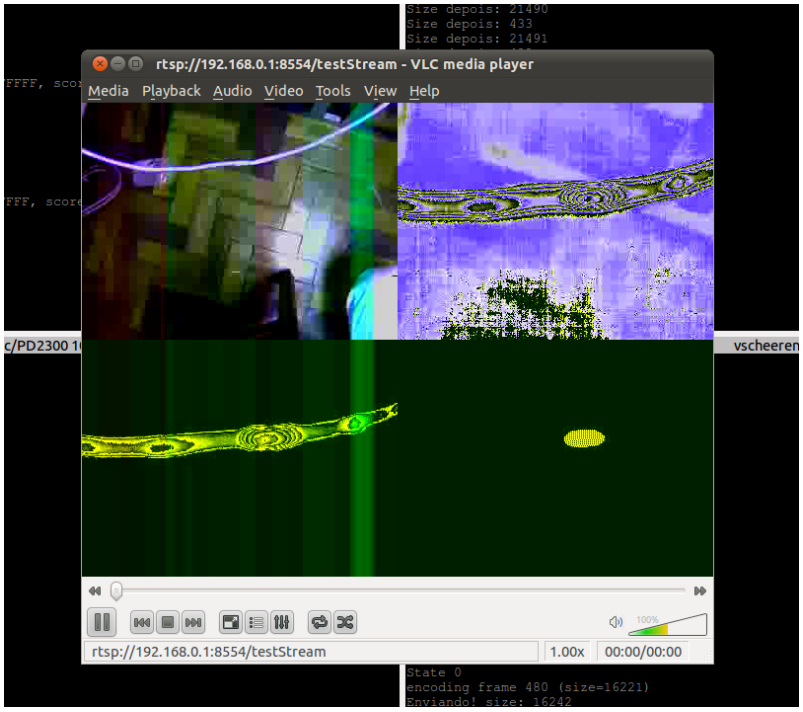

Figura 16: Visualização da stream gerada pelo módulo no software VLC.

9 CONCLUSÃO

Foi implementado neste projeto um módulo utilizando uma placa com processador de arquitetura ARM, uma câmera termográfica e uma câmera USB capaz de detectar falhas em linhas de transmissão em tempo real.

A forma de transmissão pela rede ethernet utilizada tanto pela câmera termográfica quanto pelo módulo até o computador cliente exigiu um estudo sobre os diversos protocolos de rede envolvidos. Este estudo foi documentado e pode servir de ponto de partida para futuros trabalhos nesta linha.

O uso de bibliotecas para tratamento dos protocolos RTSP e RTP foi importante pois abstraiu o tratamento de baixo nível com estes protocolos, apresentando uma solução robusta sobre a qual o projeto pôde ser desenvolvido. Isto diminuiu o tempo de implementação, porem houve um período de aprendizado da biblioteca, até chegar a um nível de conhecimento suficiente para complementar a mesma com novas classes necessárias para a aplicação desenvolvida. O resultado deste desenvolvimento pode agora ser aproveitado por outro usuários desta biblioteca em projeto semelhantes.

O sistema se mostrou eficaz na detecção *online* de falhas, gerando um resumo em tempo real no formato de vídeo para uso de operadores em uma central distante do sistema robótico.

Os requisitos de baixo consumo e pequenas dimensões foram atendidos utilizando uma placa de desenvolvimento com estas características. Já que a arquitetura de processador utilizado neste módulo é diferente do convencional x86, foi necessário utilizar ferramentas de cross-compilação, além do trabalho extra em portar as bibliotecas necessárias para o sistema alvo.

REFERÊNCIAS

CHOU, Y. C.; YAO, L. Automatic diagnostic system of electrical equipment using infrared thermography. In: INTERNATIONAL CONFERENCE OF SOFT COMPUTING AND PATTERN RECOGNITION, Malaca. **Proceedings...** Piscataway: IEEE, 2009. p.155–160.

FOROUZAN, G. **Data Communications and Networking**. [S.l.: s.n.], 2006.

GROUP, A.-V. T. W. et al. **RTP**: a transport protocol for real-time applications. [S.l.]: Internet Engineering Task Force, 1996. n.1889. (Request for Comments).

IEEE. **Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification**. Piscataway, NJ: IEEE, 1997. n.802.11.

KUROSE, J. F. **Computer Networking**: a top-down approach featuring the internet. [S.l.: s.n.], 2009.

LIBAV. **LibAv codec library**. [S.l.: s.n.], 2009. Disponivel em: http://www.libav.org. Acesso em: 2011.

LIVE555. **Live555 RTSP library**. [S.l.: s.n.], 2009. Disponivel em: http://www.live.com. Acesso em: 2011.

OTSU, N. A threshold selection method from gray-level histograms. In: IEEE TRANS. SYSTEMS, MAN, AND CYBERNETICS. **Anais...** Piscataway: IEEE, 1979. p.62–66.

POSTEL, J. **User Datagram Protocol(UDP) RFC 768**. [S.l.]: Internet Engineering Task Force, Network Working Group, 1980.

POSTEL, J. **Internet Protocol(IP) RFC 791**. [S.l.]: Internet Engineering Task Force, Network Working Group, 1981.

POSTEL, J. **Transmission Control Protocol (TCP) RFC 793**. [S.l.]: Internet Engineering Task Force, Network Working Group, 1981.

SCHULZRINNE, H.; A.RAO; R.LANPHIER. **Real time streaming protocol(RTSP) RFC 2326**. [S.l.]: Internet Engineering Task Force, Network Working Group, 1998.

UVCCAPTURE. **Command-line tool to capture webcam images**. [S.l.: s.n.], 2007. Disponivel em: http://www.quickcamteam.net/software/linux/v4l2-software/uvccapture. Acesso em: 2011.