VICTOR DE BODT SIVIERI
VICTOR VELLOCE FERREIRA

DISPOSITIVO PARA TRANSMISSÃO DE MÚLTIPLOS VÍDEOS SINCRONIZADOS

São Paulo
2013

VICTOR DE BODT SIVIERI
VICTOR VELLOCE FERREIRA

DISPOSITIVO PARA TRANSMISSÃO DE MÚLTIPLOS VÍDEOS SINCRONIZADOS

Monografia apresentada à Escola Politécnica da Universidade de São Paulo

Área de concentração:
Engenharia Elétrica

Orientador: Prof. Dr. Celso Kurashima

São Paulo
2013

AGRADECIMENTOS

Primeiramente a Deus, por tornar possível a realização deste trabalho.

Ao Doutor Celso Kurashima, pela orientação e pelo constante estímulo transmitido durante todo o trabalho.

Aos professores Marcelo Knörich Zuffo, Sérgio Takeo Kofuji, Ramona Mercedes Straube, ao pesquisador Rodrigo Barroca Dias Ferraz, à bibliotecária Ana Maria Badiali, aos familiares, aos amigos e a todos que colaboraram direta ou indiretamente, na execução deste trabalho.

Tudo posso naquele que me fortalece

Filipenses 4:13

RESUMO

Os avanços referentes à tecnologia 3D e à projeção de imagens em diferentes tipos de *displays*, como os esféricos e cilíndricos, têm favorecido o estudo e a pesquisa na área da captura e transmissão de imagens. Dentro desse contexto, o dispositivo proposto nesse trabalho tem como objetivo a sincronização e transmissão por rede sem fio de múltiplas imagens, capturadas por mais de uma câmera, posicionadas de forma a obter imagens do objeto ou da cena de diferentes ângulos. A Raspberry Pi constitui a parte central do dispositivo de transmissão de múltiplos vídeos sincronizados e como uma de suas vantagens para o projeto, essa plataforma possui um módulo de câmera desenvolvido especialmente a ela. Três métodos de sincronização foram testados e comparados. O primeiro utiliza barreira temporal, o segundo utiliza o princípio de *timestamp multicast* e o terceiro é baseado em um protocolo conhecido como PTP (*Precision Time Protocol*), visando aumentar ao máximo a simultaneidade da captura de um quadro por cada câmera, de forma a obter uma imagem reconstruída livre de defeitos e distorções. A transmissão foi feita utilizando-se protocolo TCP/IP. Utilizando-se o método de *timestamp multicast* com roteador e com transferência de arquivos via scp, a sincronização do disparo apresentou uma média de -2,7 ms e desvio padrão de 10,071 ms. Na avaliação de imagens sincronizadas de duas câmeras após a transmissão, obteve-se uma medida qualitativa da sincronização para o processo implementado.

Palavras-chave: Sincronização. Múltiplas câmeras. Múltiplos vídeos sincronizados.

ABSTRACT

Advances related to the 3D technology and image projection at different types of display, like the spherical and cylindrical, have promoted study and research in the field of images capture and transmission. In this context, the device proposed by this work has as goal the synchronization and transmission via wireless network of multiple images, captured by more than one camera, positioned to get more images of the object from different angles of view. Raspberry Pi is the central part of the synchronized multiple videos transmission device and as one of its advantages for the project, this platform has a camera module, which was designed especially for it. Three synchronization methods were tested and compared. The first one utilizes frame lock, the second one uses the principle of timestamp multicast and the third one is based on a protocol known as PTP (Precision Time Protocol), aiming to increase as much as possible the simultaneity of frame capture by each camera, in order to obtain a reconstructed image free of defects and distortions. The transmission was made by using TCP/IP protocol. Using the timestamp multicast process with a router and file transfer via scp, the trigger synchronization presented an average number of -2,7 ms and standard deviation of 10,071 ms. In the analysis of the synchronized images of two cameras after the transmission, a qualitative measure of the synchronization for the implemented process was obtained.

Keywords: Synchronization. Multiple cameras. Synchronized multiple videos.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE GRÁFICOS

SUMÁRIO

1 Introdução

1.1 Motivação

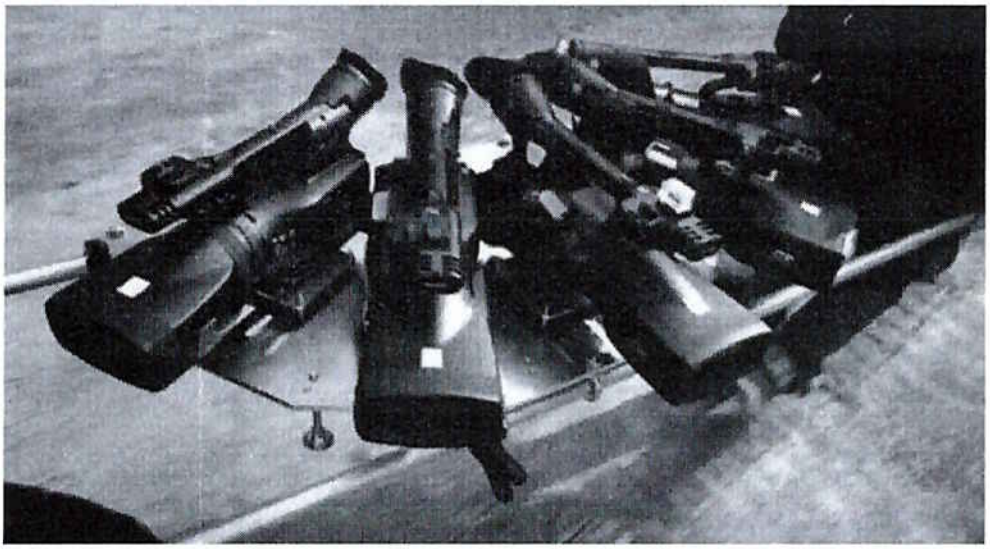
Os recentes avanços no processamento de dados e na tecnologia de fabricação de displays têm permitido a exibição de imagens de altas qualidades, resolução e nitidez. Embora haja um aumento nas dimensões das imagens exibidas e em sua qualidade, vários objetos e cenas não podem ser bem representados por telas planas, gerando a necessidade de telas curvas, como, por exemplo, esféricas e cilíndricas. Para capturar vídeos que contemplem simultaneamente vários ângulos de visão, é necessário utilizar várias câmeras.

A tecnologia 3D tem evoluído a largos passos e ganhou destaque nos últimos anos. Diferentes mercados têm também investido nessa tecnologia, como o de jogos eletrônicos, filmes e até mesmo o publicitário. A intenção é que a qualidade das imagens tridimensionais melhore cada vez mais e a área de aplicação seja expandida. A reconstrução desse tipo de imagens é algo complexo, principalmente em aplicações que exigem uma transmissão em tempo real. A fim de que isso seja possível, é necessário o uso de várias câmeras, que contemplam diferentes ângulos de visão. O surgimento de ferramentas e tecnologias mais avançadas tem contribuído bastante para o aprimoramento das projeções tridimensionais.

Para certos tipos de aplicações se faz necessário o uso de dispositivos de captura e transmissão sem fio de vídeos sincronizados. Espera-se através desse projeto fornecer uma maior liberdade às câmeras, permitindo-as se moverem ou até mesmo voarem. Em sistemas que utilizassem um grande número de picoprojetores, o uso de cabos seria praticamente inviável. Alguns outros exemplos, em que o uso desta conexão sem fio entre as câmeras e a base seria desejável, são um sistema de telepresença panorâmica e um sistema de captura esférico. No primeiro caso, imagens múltiplas capturadas por N câmeras seriam transmitidas de uma rede local e gerariam na recepção uma imagem única de alta resolução. A figura 1.1 mostra

um arranjo de câmeras criado pela Panasonic a fim de atender ao propósito de capturar, por exemplo, cenários remotos e projetá-los tridimensionalmente em uma imagem panorâmica. Outras possibilidades de aplicação do dispositivo mencionado são a reprodução de objetos tridimensionalmente em um display esférico, como o da figura 1.2 e a execução de videoconferências com telepresença, como na figura 1.3.

Figura 1.1 - Arranjo de câmeras para captura de imagens panorâmicas

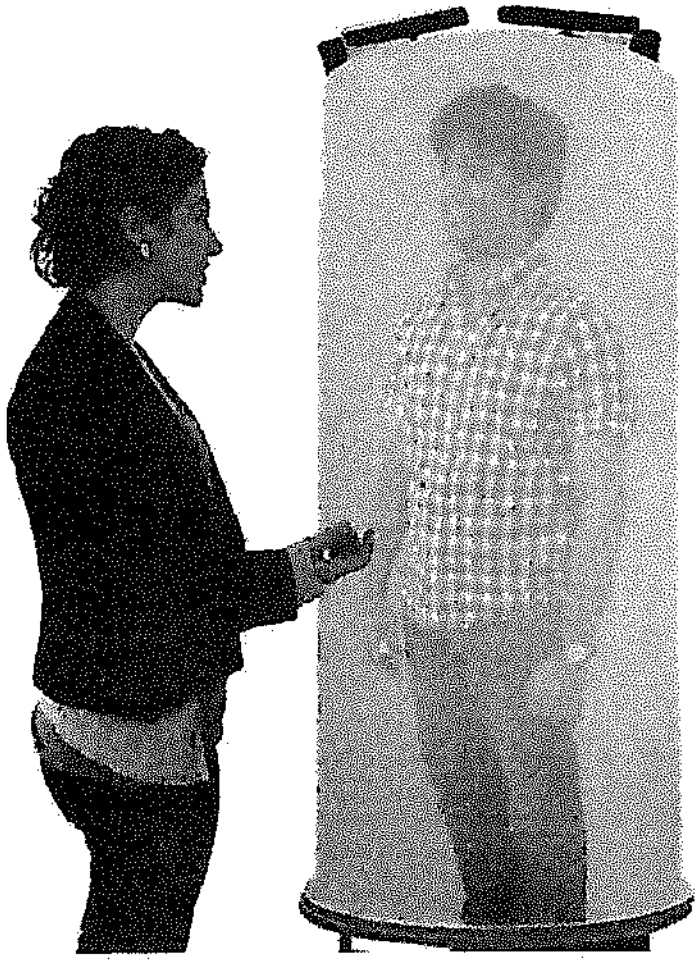


Fonte: [1]

Figura 1.2 - Display esférico exibindo imagem tridimensional

Fonte: [2]

Figura 1.3 – Sistema TeleHuman. Usuária local interagindo com locutor remoto em 3D



Fonte: [2]

Uma necessidade inerente de tais sistemas que renderizam cenas a partir de múltiplas câmeras é a captura sincronizada de todas as imagens. O instante de captura deve ser o mais próximo possível, pois caso haja atraso de uma delas, a visualização poderá ficar incorreta.

1.2 Objetivo

O objetivo do projeto é eliminar as limitações e dificuldades de posicionamento de câmeras de vídeo impostas pela necessidade de cabos de alimentação e de dados, possibilitando fixá-las facilmente em paredes, suportes de diversos formatos e até mesmo em veículos.

Para a realização do objetivo geral, será projetado e implementado um dispositivo de captura sincronizada e transmissão sem fio de baixo consumo energético. Além disso, o dispositivo deverá ser alimentado localmente por pilhas.

O dispositivo deverá aumentar a eficiência de sistemas que, para comportarem a alta taxa de transferência das câmeras, utilizam mais de um computador para a aquisição e o controle das imagens. A figura 1.4 mostra um diagrama representando uma configuração que é utilizada na prática para tentar contornar o problema. Neste exemplo, seis câmeras são conectadas por cabo a computadores de captura de imagens. Estas imagens são enviadas por rede local para o computador mestre, o qual irá realizar o processamento de todas as imagens conjuntamente. Esta captura deve ser obtida no mesmo instante, e as soluções de sincronização com câmeras cabeadas são disponobilizadas pelo fabricante [3].

A figura 1.5 mostra como a solução com os dispositivos de transmissão sem fio e sincronização é mais eficiente e elegante. O dispositivo implementa a sincronização por meio da rede sem fio e do envio de imagens para a base. Desse modo, substitui a solução cabeada, permitindo o posicionamento livre e rápido das câmeras do sistema.

Figura 1.4 - Exemplo de implementação ineficiente

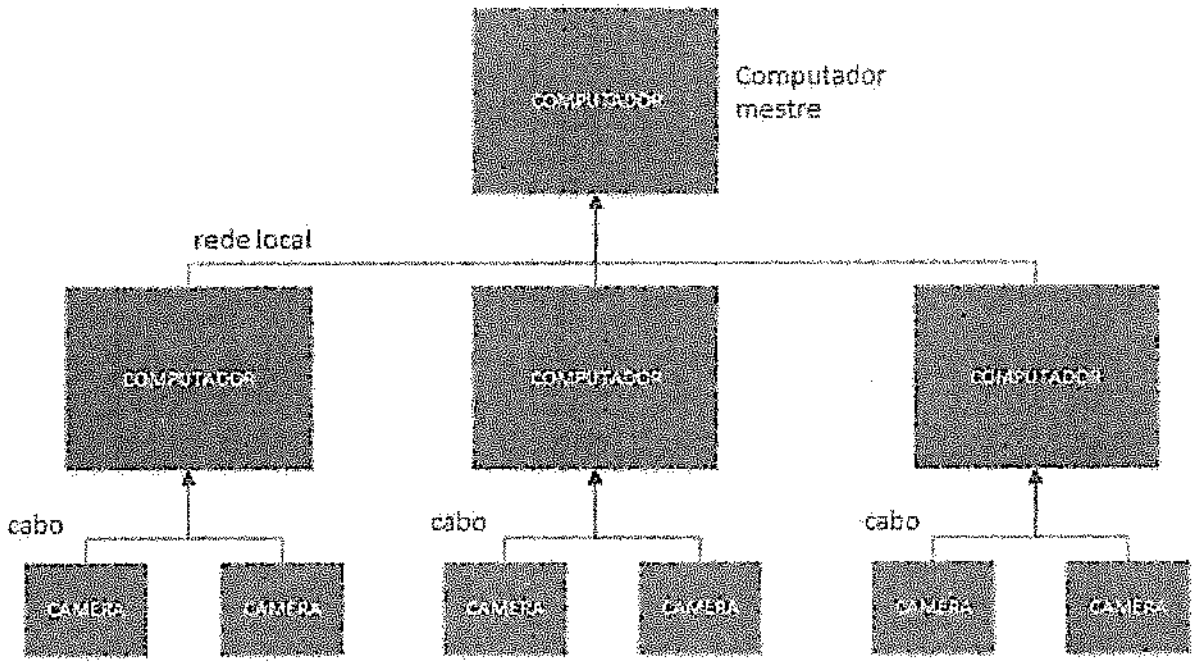
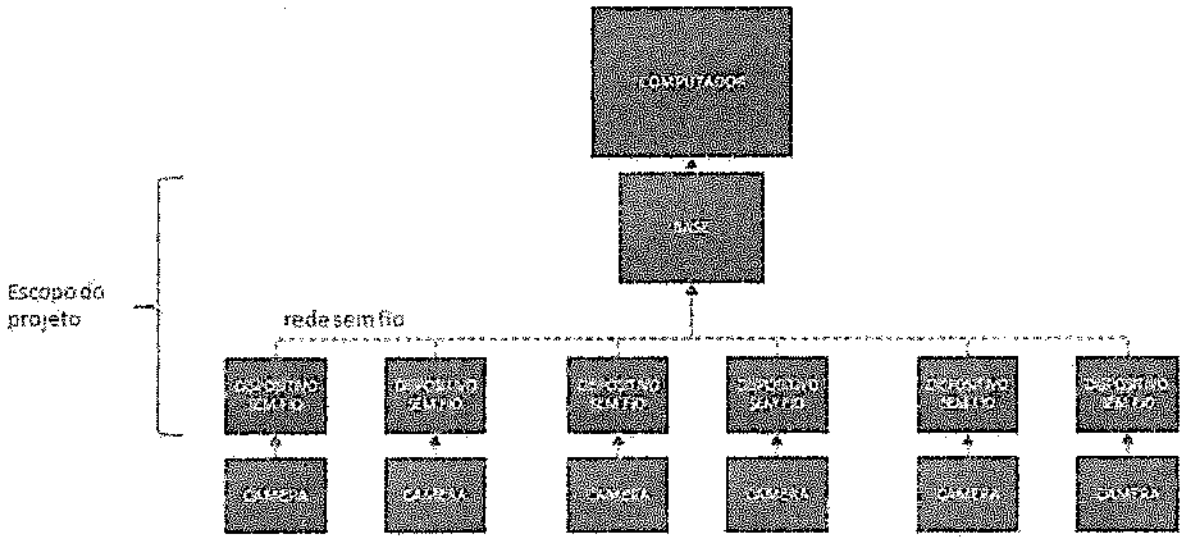


Figura 1.5 - Exemplo de implementação eficiente



2 Desenvolvimento

2.1 Revisão da literatura

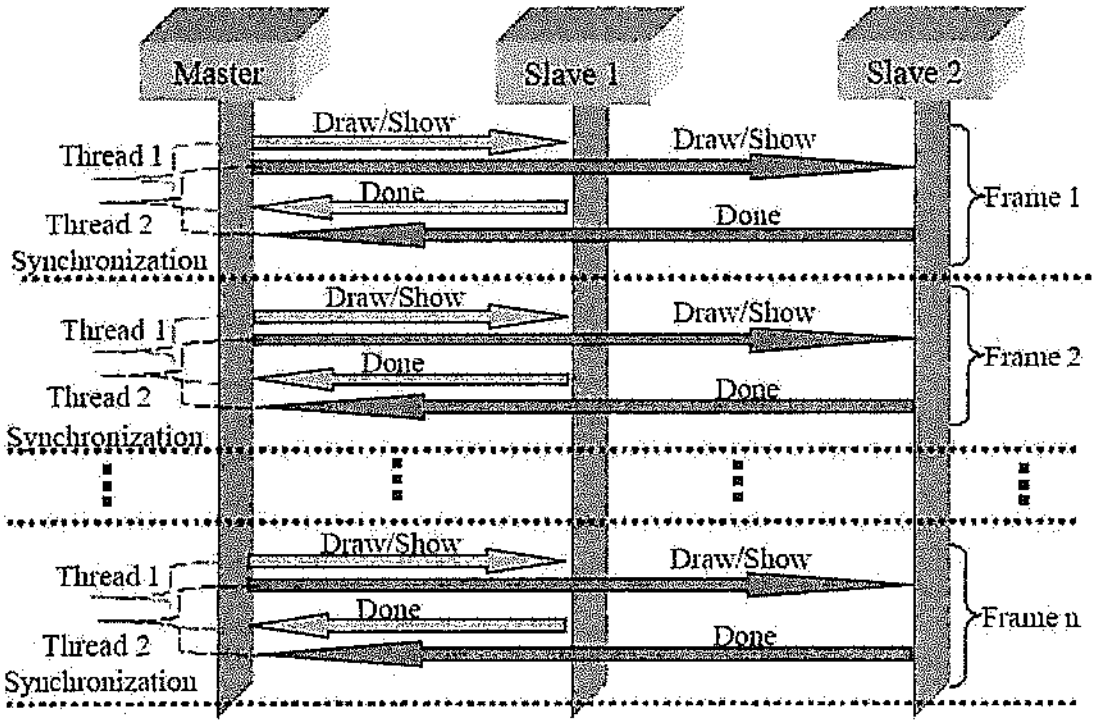
A sincronização de imagens é um tema importante para aplicações em tempo real e portanto vem sendo amplamente estudada por diversos pesquisadores com o intuito de reduzir as distorções e defeitos decorrentes da não sincronização presentes na imagem reconstruída, visto que esta não deve apresentar descontinuidades de seus componentes visuais distribuídos.

Vários métodos de sincronização já foram propostos e seu desempenho depende do contexto e aplicação em que o projeto está inserido.

Um método baseado em uma "barreira temporal", conhecido como *frame lock*, para a sincronização de imagens em ambientes imersivos de multiprojeção baseados em um cluster de computadores foi proposto por [4]. Um exemplo de ambiente imersivo é o sistema denominado Cave Automatic Virtual Environment (CAVE), que consiste de uma sala fechada, onde um ambiente virtual formado por imagens projetadas nas paredes através de projetores é criado. Esta tecnologia pode ser utilizada para aplicações em diversas áreas, como engenharia, medicina, astronomia, arquitetura, jogos e artes [5]. No caso dos clusters, o processo de *frame lock* consiste na atualização de uma imagem em um nó, imediatamente após os outros haverem terminado suas atualizações [6]. O software desenvolvido para esse fim é baseado em CORBA (Common Object Request Broker Architecture), que é a arquitetura padrão desenvolvida pelo Object Management Group (OMG) e visa facilitar a comunicação e interação, como troca de mensagens, entre sistemas distribuídos heterogêneos. Essa arquitetura simplifica o problema, pois possui características necessárias para a aplicações inseridas em um contexto de sistemas distribuídos. Em linhas gerais, o funcionamento do software consiste primeiramente na criação, no nó mestre, de uma *thread* para cada nó escravo, sendo que cada uma delas envia uma mensagem para que o respectivo nó escravo inicie o processamento da imagem. Quando estes tiverem finalizado sua tarefa, enviam uma outra mensagem

ao mestre confirmando o término do processamento. O nó mestre permanece bloqueado até que receba mensagem de finalização de todos os nós escravos. Essa última etapa descrita caracteriza o método de sincronização por *frame lock*, que conforme já foi mencionado, é uma implementação de uma barreira ou bloqueio para que todos os dispositivos ou nós iniciem suas tarefas ao mesmo tempo. Eventuais falhas de sincronização são causadas por variações no tempo de processamento de cada dispositivo, visto que em alguns casos e para determinados equipamentos é difícil controlar esse parâmetro. A figura 2.1 ilustra esse método de sincronização.

Figura 2.1 – Implementação por software da sincronização utilizando o método de frame lock



Fonte: [4]

Um exemplo de ferramenta desenvolvida que implementa a sincronização nos nós de um *cluster* de computadores foi proposta em [7]. As vantagens dessa biblioteca são alta velocidade, baixo tempo de processamento e facilidade de adaptar programas para usá-la. Para atingir esse propósito, decidiram implementar a biblioteca em TCP/IP.

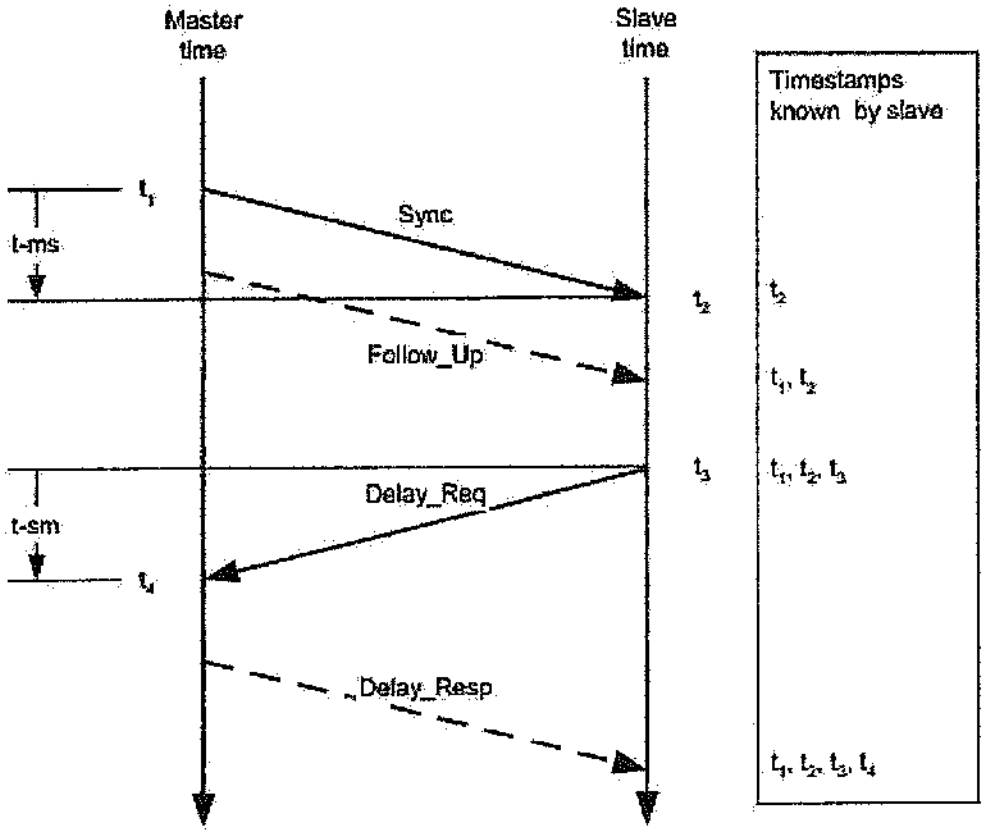
Além do *frame lock*, existem também métodos de sincronização dos relógios de cada computador, em que o mestre deve informar o momento em que a ação deverá ocorrer e consequentemente, devido à sincronização dos relógios, a ação será executada simultaneamente pelos computadores. Esse processo envolve alguns desafios devido à variação de tempo entre os relógios e a ligeiras diferenças intrínsecas e naturais de frequência entre eles. Um protocolo de sincronização de relógios existente é o protocolo PTP (IEEE1588), abreviação de Precision Time Protocol [8]. Esse protocolo foi projetado para atingir uma precisão maior do que o protocolo NTP (Network Time Protocol) e viabilizar aplicações em que não seja possível utilizar um receptor GPS em cada nó ou dispositivo, seja devido ao custo, seja devido à inacessibilidade desses sinais. Em relação a esse protocolo, os relógios são organizados em uma hierarquia de sincronização no modelo mestre-escravo, sendo o relógio de referência do sistema denominado *grandmaster clock*. A sincronização é obtida através da troca de mensagens entre os nós mestre e escravo, havendo um ajuste do relógio por parte dos escravos em relação ao relógio do mestre a partir das informações enviadas e recebidas. A escolha do relógio que servirá de referência para cada segmento de rede em um domínio é realizada através de um algoritmo conhecido como *Best Master Clock (BMC) Algorithm*. Esse algoritmo é baseado na comparação de atributos dos relógios, na seguinte ordem:

1. Prioridade 1
2. Classe
3. Precisão
4. Variância
5. Prioridade 2
6. Identidade

A variância está relacionada coma estabilidade do relógio e a sua identidade é um número único universal que o identifica.

A figura 2.2 mostra de forma esquematizada o processo de ajuste de tempo pelo nó escravo a partir de mensagens trocadas com o mestre (referência).

Figura 2.2 – Esquema da troca de mensagens entre os nós mestre e escravo



Fonte: [8]

Primeiramente, o mestre envia uma mensagem no tempo t1 para o escravo. Essa mensagem contém o *timestamp* relativo a t1. O nó escravo a recebe em t2 e marca esse tempo. Para determinar o tempo que demora para a mensagem ser transmitida do mestre para o escravo, este envia uma mensagem para o mestre em t3. Essa mensagem é recebida pelo mestre em t4. Este tempo é escrito em uma nova mensagem e enviada para o escravo. Ao final desse processo, o nó escravo tem guardado os tempos t1, t2, t3 e t4, e a partir desses valores é possível calcular o *offset* do relógio do escravo em relação ao relógio do mestre, assim como o tempo de transmissão da mensagem de um nó a outro. Considerando-se o tempo de

transmissão da mensagem do mestre para o escravo (t-ms) igual ao tempo de transmissão do escravo para o mestre (t-sm), o cálculo é feito da seguinte forma:

$$t_2 - t_1 = offset + \text{t-ms}$$

$$t_4 - t_3 = -offset + \text{t-sm}$$

$$offset = \frac{t_2 - t_1 + t_3 - t_4}{2}$$

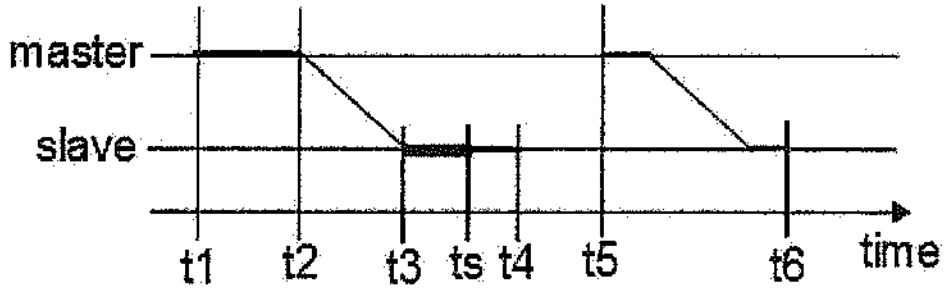
O grau de igualdade entre t-ms e t-sm é um fator que influencia a precisão da sincronização do relógio do nó escravo em relação ao relógio do nó mestre, ou seja, é desejável que o atraso de propagação seja simétrico, ou ao menos, que a assimetria seja constante.

Outro fator que está relacionado com a precisão da sincronização do relógio é a precisão do timestamp gerado pelo nó mestre e escravo e gravado nas mensagens. Valores de timestamp altamente precisos podem ser atingidos através de interfaces especializadas de hardware na camada física da rede [9]. Caso não haja esse hardware especializado, é possível fazer uma implementação em software, visando atingir a máxima precisão possível para os timestamps. Porém, como a geração do timestamp não ocorre na camada física, e sim em camadas mais altas da rede, há a introdução indesejável de jitter no processo. Um exemplo de implementação em software foi proposto em [9].

Um método de sincronização baseado no padrão IEEE 802.11 (WiFi) foi proposto em [10]. Este método consiste em utilizar o *beacon frame* enviado pelo Ponto de Acesso (AP - Access Point).

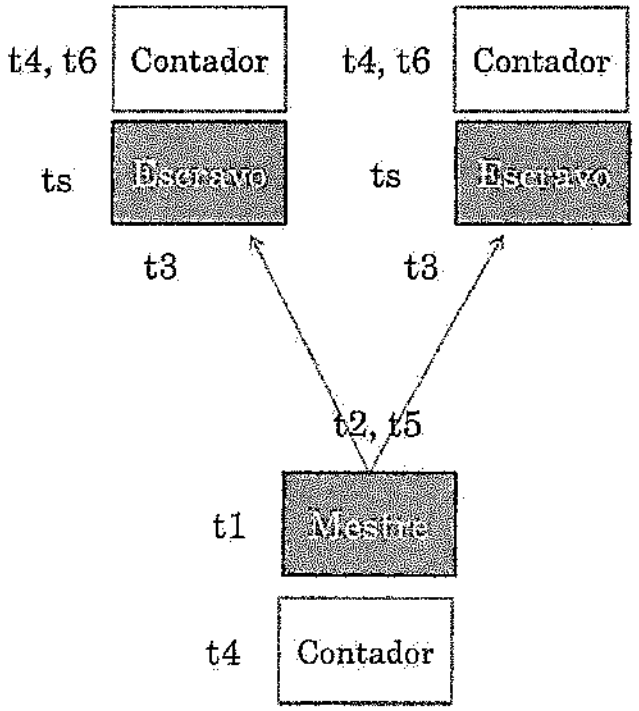
Para o protocolo proposto, é suposto que se dois receptores recebem o mesmo *frame*, eles o recebem aproximadamente ao mesmo tempo. O procedimento funciona da seguinte forma, conforme pode ser visto nas figuras 2.3 e 2.4:

Figura 2.3 – Sincronização do clock segundo o método proposto



Fonte: [10]

Figura 2.4 – Diagrama ilustrativo do processo de sincronização segundo o método proposto



O mestre prepara uma mensagem "indicativa" (*beacon frame* com *timestamp* atual) em t1 e a envia (broadcast) em t2 aos receptores (escravos), que por sua vez, ao receberem-na em t3, recebem o *timestamp* em ts (inclusive o mestre) e atualizam seus contadores internos. O tempo entre ts e t4 (cálculo e ajuste do novo valor de clock) é considerado pequeno.

Após essa etapa, o mestre estima o tempo que o *beacon frame* demora para ser processado, calcula um novo *timestamp* e prepara, em t5, outra mensagem "indicativa" contendo o *timestamp* e a envia de modo que os receptores possam, em t6, comparar o seu próprio *timestamp* com o do mestre, calcular a diferença e ajustar o valor do clock local.

O processo descrito não tolera perdas de mensagem, visto que caso isso aconteça, não será possível fazer a comparação entre os time-stamps de mensagens consecutivas. Visando aumentar a robustez contra perda de mensagens, são inclusos em cada mensagem de sincronização, os valores dos time-stamps das n (supondo que n seja a tolerância) mensagens anteriores. A máxima variação para esse método foi de 43us, utilizando hard-int do Windows NT.

Outro fator importante a ser estudado e considerado é a transmissão e compressão de vídeo em redes sem fio, na qual o canal é variante no tempo e propenso a erros. Nesse tipo de transmissão, o pacote a ser transmitido pode ser recebido corretamente, ser recebido com erros ou ser perdido [11]. Os dois últimos casos geram distorções na imagem reconstruída. Como descrito em [11], a distorção se torna cada vez maior ao longo de uma sequência de *frames*, visto que a compressão de vídeos é baseada em um processo de predição. A distorção final apresentada pela imagem reconstruída no receptor pode ser considerada como a soma de dois tipos diferentes de distorção, que são a distorção decorrente da transmissão e a decorrente dos erros de codificação. É importante analisar e modelar esses tipos de distorção, de forma a tentar minimizá-los,e assim evitar perda de qualidade da imagem.

Há necessidade de múltiplas câmeras não somente para captura de imagens 3D ou panorâmicas, mas também, por exemplo, em sistemas de monitoramento do meio ambiente em que haja risco para as câmeras, como por exemplo, risco de avalanche. Nesse caso também deve haver uma sincronização entre as câmeras, porém de uma forma diferente ao que será utilizado no nosso projeto, visto que neste tipo de sistema onde há riscos, o objetivo é que as câmeras capturem imagens de um modo intercalado. Contudo, é possível extrair a ideia principal dessa proposta de sincronização, na qual as câmeras devem ajustar o instante da próxima

captura a partir do cálculo da diferença entre o time-stamp de dois beacons adjacentes enviados por uma câmera mestre, conforme proposto em [12].

Uma rede wireless de sensores é descrita em [13], cuja execução somente se tornou possível devido ao processamento de imagens, como compressão JPEG, detecção de bordas, convolução e histogramas, na placa da própria câmera, antes de ser feita a transmissão, visto que o consumo energético deve ser baixo para que seja possível a operação através de baterias. O escopo do projeto dessa rede de sensores se assemelha ao nosso, no sentido de ser um sistema distribuído, onde a comunicação entre as câmeras é necessária, e de ser operado a bateria, necessitando um baixo consumo energético. A diferença reside no fato de que o nosso dispositivo é formado exclusivamente por câmeras (sem sensores de áudio, temperatura, luz, umidade e acelerômetros).

2.2 Raspberry Pi

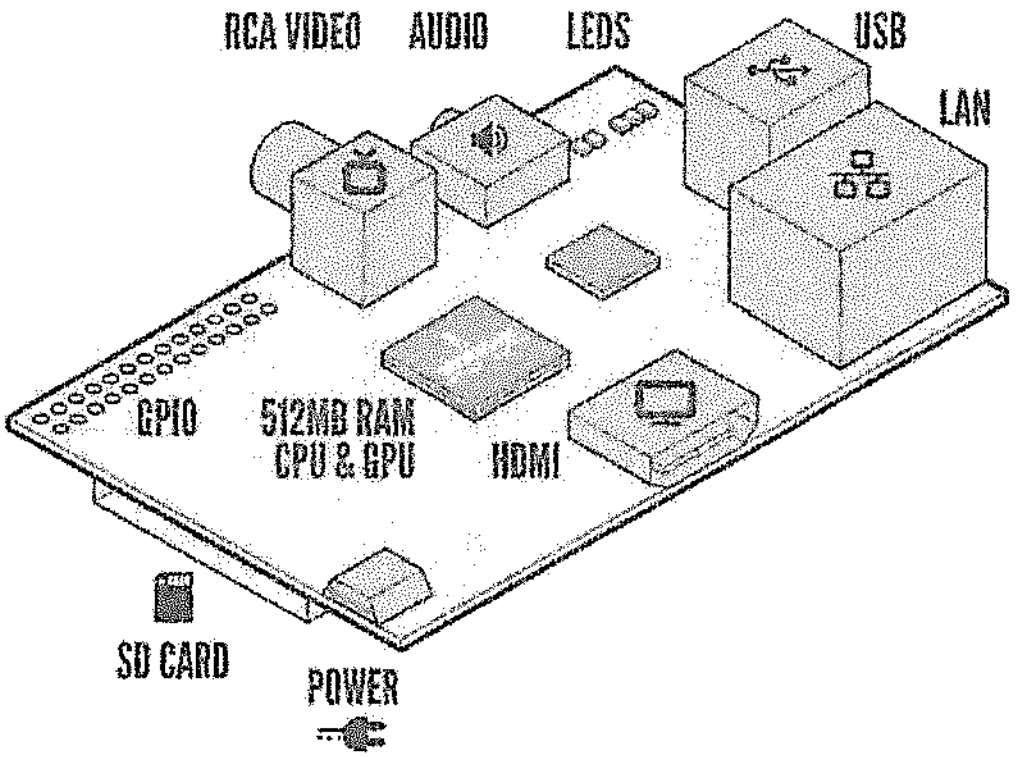
A Raspberry Pi é um computador de pequenas dimensões e constitui a parte central do dispositivo de transmissão de múltiplos vídeos sincronizados proposto. Essa plataforma foi desenvolvida pela Raspberry Pi Foundation e lançada em 29 de fevereiro de 2012.

A vantagem que o uso dessa plataforma apresenta para o projeto é a disponibilidade de um hardware já projetado que atenda às especificações desejadas e a existência de códigos e programas disponíveis publicamente para que todos possam ter acesso. A interface simples desse produto também conta como um agente facilitador.

Dois modelos de Raspberry Pi, o modelo A e o modelo B, já foram lançados, sendo o modelo B o que possui mais funcionalidades e uma maior capacidade de memória. O modelo B apresenta duas portas USB, uma porta Ethernet e uma memória RAM de 512 MB, enquanto o outro modelo apresenta somente uma porta USB, nenhuma

porta Ethernet e uma memória RAM de 256 MB. A figura 2.5 ilustra o modelo B com seus principais componentes e interfaces.

Figura 2.5 – Ilustração do modelo B da Raspberry Pi



Fonte: [14]

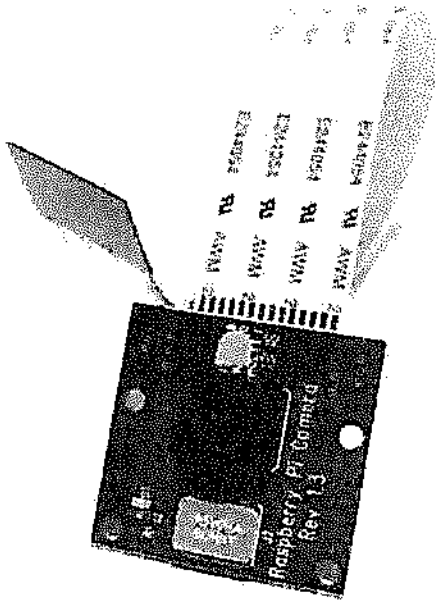
As principais especificações e características [14] deste dispositivo são listadas a seguir:

- Dimensões: 85.6mm x 56mm x 21mm
- Peso: 45 g
- Utilização de um SoC (System-on-Chip): Broadcom BCM2835, que contém um ARM1176JZFS, com ponto flutuante, operando a 700MHz e um Videocore 4 GPU. Possui codec H.264/MPEG-4 AVC a 1080p30.

Decidiu-se utilizar a Raspberry Pi para se implementar o dispositivo de transmissão de múltiplos vídeos sincronizados, pois há um módulo de câmera desenvolvido especialmente para essa plataforma e há a possibilidade de se usar um WiFi USB dongle, além de possuir um processador compatível com as necessidades, como possuir *codecs* que suportem codificar JPEG e h.264 a 1080p30.

O módulo da câmera é conectado diretamente à Raspberry Pi utilizando uma interface CSI através de um *ribbon cable*. A figura 2.6 mostra o módulo da câmera com seu cabo.

Figura 2.6 – Módulo da câmera desenvolvido e fabricado pela Raspberry Pi Foundation



Fonte: [15]

Esse módulo possui um sensor de imagem CMOS Omnivision 5647 de 5MP (2592x1944 pixels) com modo de foco fixo. A placa tem dimensões de 25mm x 20mm x 9mm e pesa aproximadamente 3g. O *ribbon cable* possui 15 pinos e conforme já mencionado, é conectado à Raspberry Pi através da interface CSI (Camera Serial Interface), cujo barramento é capaz suportar altas taxas de transferência e transporta exclusivamente pixels ao processador [15].

2.3 Materiais

Os componentes foram escolhidos com base no modelo conceitual e deecidiu-se por selecionar componentes que proporcionassem uma boa relação entre tempo de implementação do sistema, capacidade e preço. Os componentes principais são:

- Raspberry Pi + módulo de câmera
- Cartão de memória micro SD
- Adaptador WiFi USB
- Cabo Ethernet e/ou cabo HDMI
- UBEC (*Universal Battery Eliminator Circuit*)
- Interruptor
- Fonte de tensão
- Cabo micro USB

Um orçamento foi feito e pode ser conferido no apêndice A.

2.4 Características gerais do projeto

Nesta subseção serão apresentadas as características necessárias para que o dispositivo possa ser inserido no contexto de aplicações desejáveis que serão mencionadas.

O dispositivo projetado tem por objetivo viabilizar aplicações em que o uso de cabos seria um fator limitante e restritivo, como, por exemplo, câmeras móveis ou um arranjo de múltiplas câmeras a certa distância uma da outra, ou até mesmo um arranjo com um número de câmeras que impossibilitasse o cabeamento.

As principais características estão listadas a seguir:

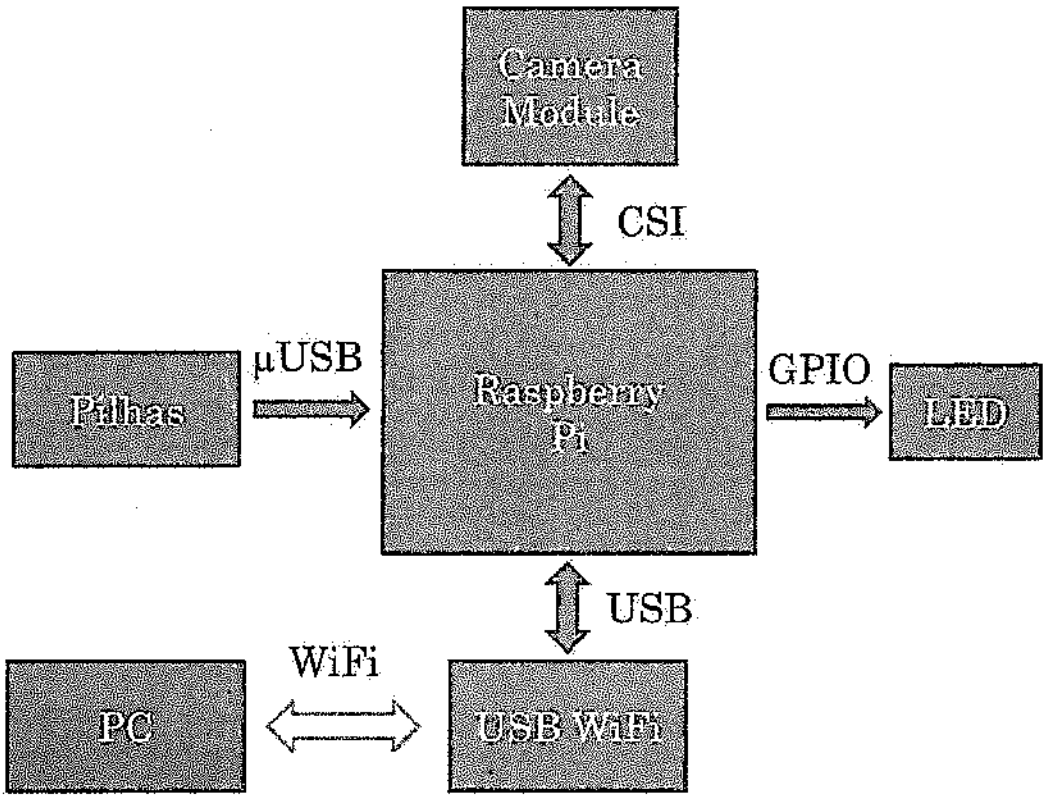
- As imagens recebidas e geradas dever apresentar alta qualidade, de tal forma que se aproximem ao máximo da realidade.
- Para promover uma boa sensação de continuidade e fluidez das cenas, faz-se necessária a redução da latência (tempo entre a captura do *frame* e sua exibição).
- As dimensões e o peso do dispositivo devem ser reduzidos, de forma que seja portátil e não acrescente muito peso ao veículo em situações em que se faça, por exemplo, as câmeras voarem.
- Operabilidade à bateria para permitir uma maior mobilidade ao dispositivo.

2.5 Implementação

Nesta subseção serão descritas tanto a ideia e os conceitos principais relativos à implementação do projeto, como também os detalhes da execução.

O diagramas de blocos relativo à parte física do dispositivo pode ser visto na figura 2.7. Esse diagrama mostra os principais componentes e interfaces de conexão entre eles.

Figura 2.7 – Diagrama de blocos representando a implementação física do projeto

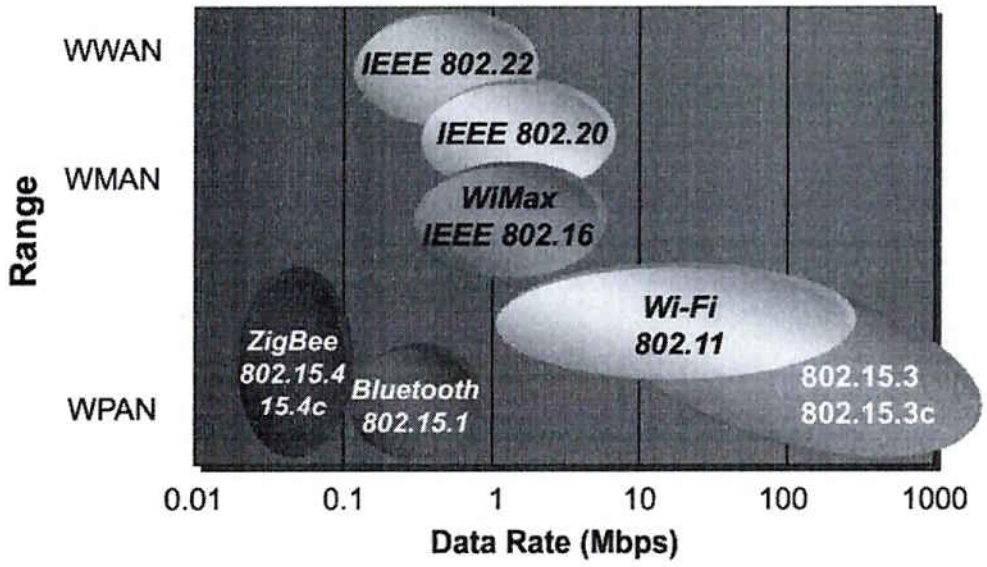


Como o projeto visa a viabilidade em aplicações que necessitam uma maior liberdade das câmeras, podendo elas ser móveis, a alimentação do dispositivo deve ser feita por meio de pilhas, de modo que não haja o incoveniente do cabo de alimentação externa. A conexão entre as pilhas e a Raspberry Pi será feita através da porta micro USB.

Conforme já foi mencionado na subseção 2.2, a câmera é conectada diretamente à Raspberry Pi através de um *ribbon cable*, sendo CSI a interface de conexão.

A transmissão será feita via WiFi, mais especificamente pelo protocolo 802.11n. Foi escolhido esse protocolo de comunicação analisando-se critérios de eficiência energética, taxa de transferência e alcance do sinal. A figura 2.8 mostra um diagrama comparativo entre características (alcance do sinal e taxa de transferência) de diferentes protocolos de comunicação.

Figura 2.8 – Análise comparativa entre diferentes protocolos de comunicação



Fonte: [16]

A tecnologia WiFi (padrão IEEE 802.11) se divide em vários padrões, sendo o IEEE 802.11n um deles. A tabela 2.1 apresenta uma análise comparativa entre alguns dos padrões referentes à essa tecnologia.

Tabela 2.1 - Análise comparativa entre características de diferentes padrões da tecnologia WiFi

	802.11a	**802.11b**	**802.11g**	**802.11n**
Data de aprovação	Julho de 1999	Julho de 2000	Junho de 2003	Outubro de 2009
Taxa máxima de transferência de dados (Mbps)	54	11	54	600
Modulação	OFDM	CCK/DSSS	CCK/DSSS ou OFDM	CCK/DSSS ou OFDM
Banda RF (GHz)	5	2,4	2,4	2,4 ou 5
Número de fluxos espaciais	1	1	1	1, 2, 3 ou 4
Largura nominal de canal (MHz)	20	20	20	20 ou 40

Fonte: [17]

Analisando-se a figura 2.8, percebe-se que a transmissão via WiFi possui um alcance maior do que certas tecnologias, como Bluetooth e o padrão 802.15.3a, e uma taxa de transferência maior em relação aos protocolos Bluetooth e ZigBee. Como já foi citado, existem subdivisões do protocolo IEEE 802.11, e a tabela 2.1 aponta que o padrão IEEE 802.11n apresenta a maior taxa de transferência dos padrões comparados, podendo atingir até 600 Mbps. Esse parâmetro foi decisivo na escolha do protocolo de transmissão, visto que a taxa de transmissão deve comportar as taxas de bits por segundo dos vídeos comprimidos e, quanto maior a taxa de transferência de dados, maior o número de câmeras que podem ser utilizadas por rede. Em relação ao projeto proposto nesse trabalho, o adaptador WiFi utilizado possui taxa de transferência igual a 150 Mbps.

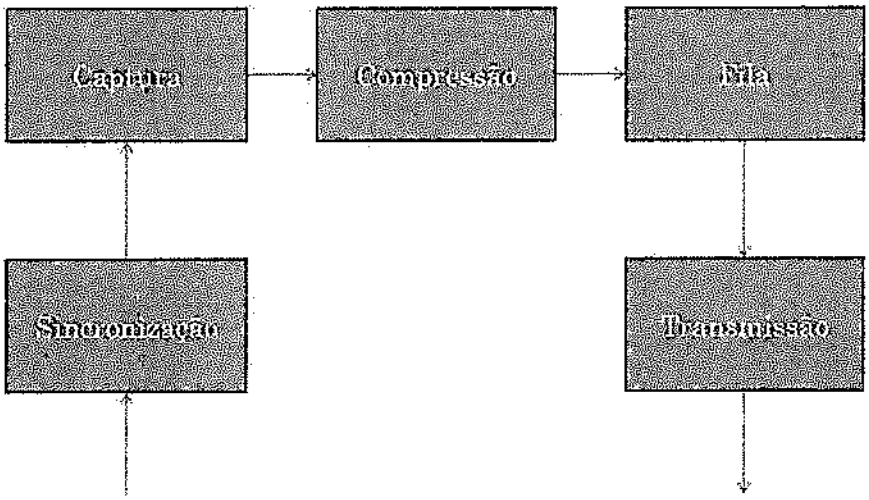
No dispositivo proposto nesse trabalho, o adaptador WiFi será conectado à Raspberry Pi através de uma das portas USB. As imagens transmitidas serão recebidas por um computador.

Para que seja possível utilizar mais câmeras por rede, as imagens serão comprimidas antes de serem transmitidas. A Raspberry Pi possui *codecs* de compressão de vídeo, configurando uma vantagem de se usar essa plataforma.

Serão utilizados alguns pinos GPIO da Raspberry Pi para a realização de alguns testes.

A figura 2.9 mostra um diagrama de blocos lógico simplificado da implementação do projeto e cada bloco será detalhado posteriormente.

Figura 2.9 – Diagrama de blocos representando a implementação lógica do projeto



Como pode ser visto no diagrama, o sistema foi dividido em cinco módulos: sincronização, captura, compressão, fila e transmissão. O processo de sincronização sincroniza os relógios dos dispositivos por meio de um protocolo de sincronização. O relógio atualizado pelo processo de sincronização é utilizado para gerar o sinal de disparo da captura. A captura e a compressão são realizadas pela GPU, que é acessada pelo programa através da API OpenMAX. A imagem comprimida é armazenada na memória principal do dispositivo e suas informações são armazenadas em uma fila para serem posteriormente acessadas e transmitidas para o mestre.

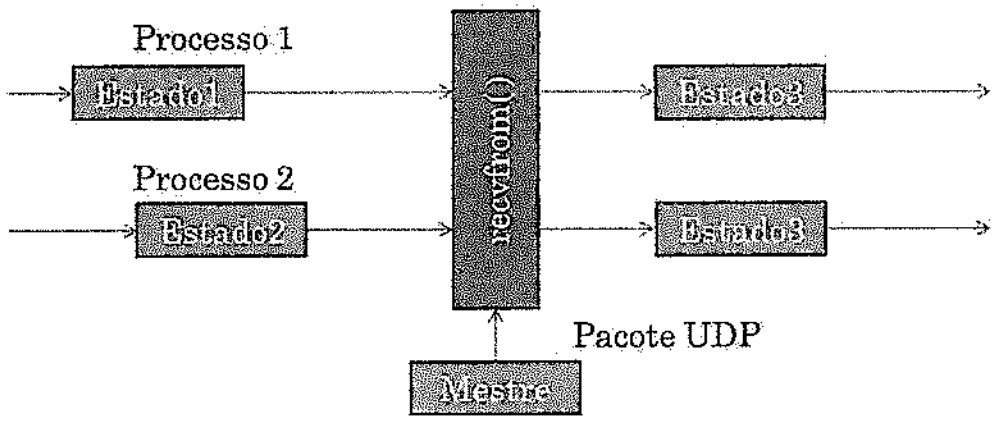
Para o escravo, decidiu-se por utilizar o sistema operacional Raspbian, que é derivado do Debian, portado para ARM e, mais especificamente, para o Raspberry Pi. Para os programas do escravo, utilizou-se a linguagem C, as bibliotecas da API OpenMAX e as bibliotecas referentes ao SoC da Broadcom utilizado no Raspberry Pi. Para os programas do mestre foi utilizado o sistema operacional Windows 7 e o Visual C++. As seções seguintes serão dedicadas a detalhar a implementação e a funcionalidade de cada um deles.

2.5.1 Sincronização

A literatura apresenta diversos métodos de sincronização baseados em trocas de pacotes. Para este projeto foram testados três métodos: o método de barreira temporal, o método de *timestamp multicast* e uma adaptação do PTP baseado em trocas de pacotes. Cada método apresentou vantagens e desvantagens que serão discutidas em detalhes na seção de resultados.

O método de barreira temporal consiste em aguardar que os dispositivos escravos cheguem a determinado ponto do programa e, através de uma instrução que bloqueia sua execução, aguardem por um sinal que os desbloqueie, no caso, o recebimento de um pacote *multicast* enviado pelo dispositivo mestre. O programa continua, então, a sua execução e efetua a captura da imagem. Neste caso, quem determina os momentos de captura de imagens e sua quantidade é o mestre. Para reduzir o tempo gasto na troca de informações, não se aguarda a confirmação de que todos chegaram à barreira, mas estima-se um tempo mínimo que o mestre deve aguardar entre duas mensagens consecutivas.

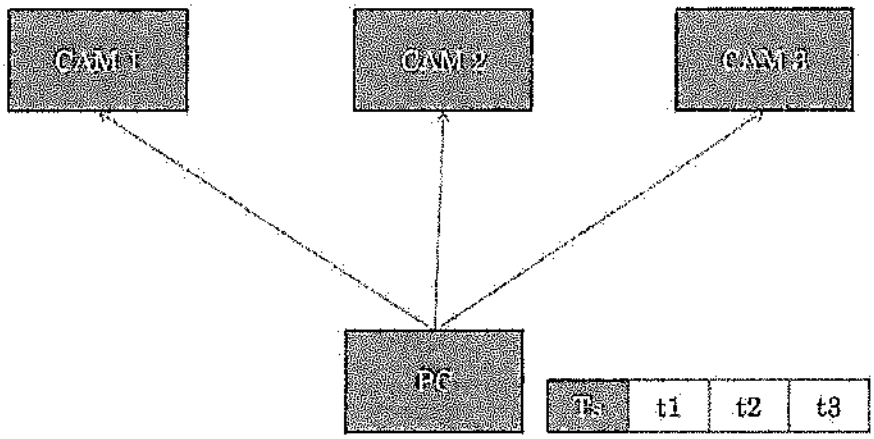
Figura 2.10 - Diagrama exemplificando o processo de barreira temporal



O método de *timestamp multicast* é uma variação do método da barreira temporal. Consiste no mestre lendo constantemente o seu relógio e enviando o valor para todos os escravos em *multicast*. Os escravos, que se encontram bloqueados pela instrução que aguarda o recebimento do pacote, são desbloqueados e atualizam seus relógios de acordo com o mestre e efetuam disparos baseados em seus

relógios e não no recebimento dos pacotes. Este método possui a vantagem de ser mais robusto à perda de pacotes que o primeiro.

Figura 2.11 – Ilustração do processo de timestamp broadcast



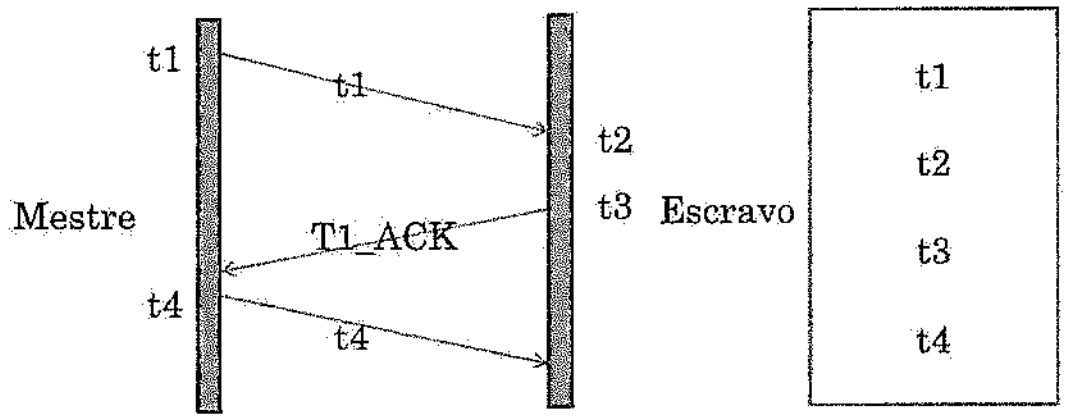
A adaptação do método do PTP consiste em uma simplificação do protocolo, uma vez que o ambiente do projeto é consideravelmente mais controlado do que os ambientes para os quais o PTP foi concebido, uma vez que o mestre é escolhido de antemão e os escravos são dispositivos da mesma natureza, executando programas semelhantes. O protocolo PTP prevê muitos pacotes de controle, enquanto que o protocolo implementado neste projeto limita-se ao processo de *handshake*:

1. O mestre envia um pacote com o valor de seu relógio imediatamente anterior ao envio (t_1).
2. O escravo armazena o valor de seu relógio no recebimento do pacote (t_2) e responde com um pacote confirmando o seu recebimento, armazenando o valor de seu relógio imediatamente anterior ao envio do pacote (t_3).
3. O mestre então envia uma resposta à confirmação do escravo, contendo o valor de seu relógio do recebimento do pacote da etapa 2 (t_4).
4. O escravo estima o *round trip*(tempo que um pacote leva para ir e voltar do mestre) e, baseado nele, estima o tempo t_d que um pacote leva para sair de um dispositivo e chegar ao outro, tempo esse que é considerado como sendo metade do *round trip*. Baseado em t_d, t_1 e t_2, o escravo estima, então, a diferença entre o seu relógio e o do mestre, ajustando o seu próprio relógio de acordo.

Para melhorar a precisão do ajuste dos relógios é interessante que o armazenamento dos valores de relógio seja realizado na passagem da camada MAC para a camada física do adaptador de rede. Deste modo, reduz-se a incerteza gerada pelas camadas mais altas da rede. Este processo, no entanto, depende de suporte dos *drivers* do adaptador e não será implementado neste projeto, sendo, portanto, elencado em trabalhos futuros.

Uma vez ajustados os relógios, como no processo de *timestamp multicast*, os disparos são baseados nos relógios dos escravos. Para reduzir o *wander*(variação a longo prazo da diferença entre os relógios de dois dispositivos, causada pela diferença de frequência dos dois dispositivos), o Linux dispõe de uma função que permite ajustar a velocidade do relógio. Pode-se, portanto, ajustar a velocidade do relógio baseado na diferença entre o mestre e o escravo. O código fonte dos programas se encontra no apêndice B.

Figura 2.12 - Diagrama representando a troca de mensagens no protocolo adaptado do PTP



2.5.2 Captura e compressão

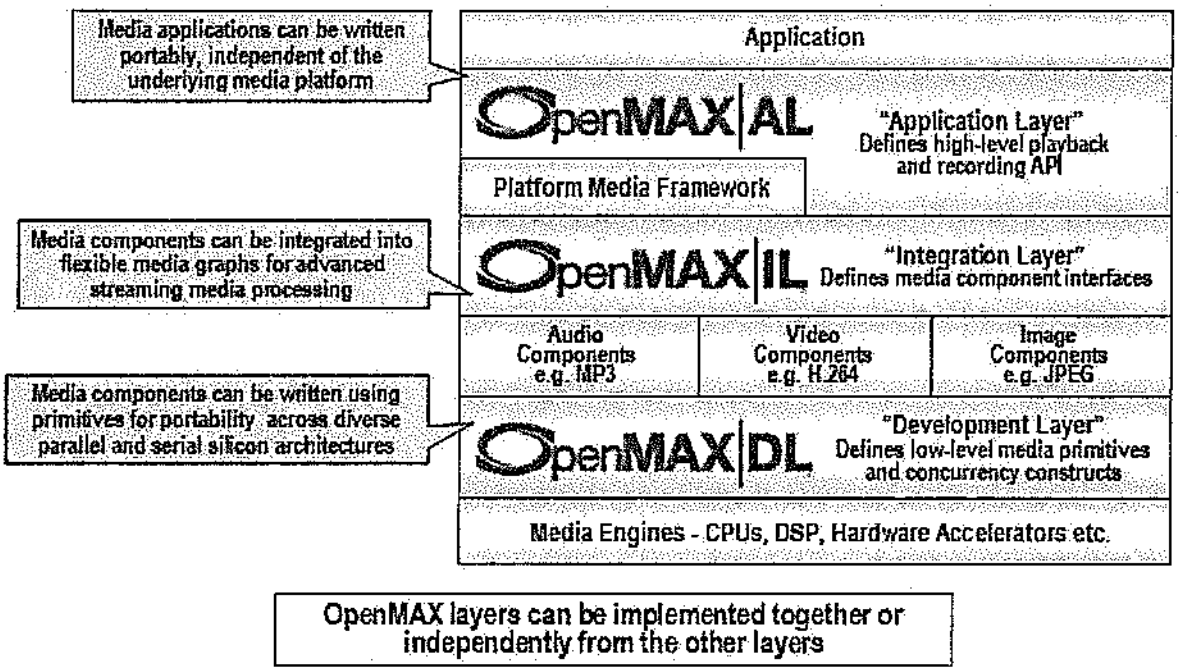
Para a captura de imagens, o Raspberry Pi modelo B disponibiliza dois tipos de conexão, possuindo duas portas USB (sendo que, neste projeto, uma já esta sendo utilizada pelo adaptador WiFi), e um conector CSI (Camera Serial Interface), cujo protocolo é estabelecido pela MIPI Alliance. O acesso a uma câmera USB é realizado pelo processador ARM do SoC da Broadcom, através de *drivers* específicos do fabricante da câmera. O acesso a câmeras CSI é realizado por meio da GPU do SoC, denominada Broadcom VideoCore IV. Para o projeto foi escolhida uma câmera CSI que utiliza o chip OV5647 da OmniVision e é vendida como módulo para a Raspbarry Pi. Para a compressão foi escolhido o formato JPEG, que é amplamente utilizado, possui taxa de compressão variável e, por simplicidade, foi escolhido um formato de compressão de imagens estáticas e não de vídeo, para facilitar o processamento de cada quadro. Porém, para a transmissão de vídeo a 30 quadros por segundo faz-se necessário lançar mão de um método de compressão que utilize a redundância temporal além da espacial, como, por exemplo, o h264/AVC.

As versões atuais do sistema operacional Raspbian contêm dois programas para capturar imagens estáticas ou vídeos da câmera, Raspistill e Raspivid respectivamente e, recentemente, os códigos fontes deles e dos drivers que são utilizados por eles para realizar a captura foram divulgados como código aberto. Apesar disso, os detalhes de operação da GPU ainda permanecem proprietários e, embora os programas tenham seus códigos abertos, ainda há pouca documentação a respeito disponível para o público. O programa Raspistill serviu de base para o desenvolvimento do programa final.

Os programas de captura utilizam a API OpenMAX, desenvolvida pelo grupo Khronos, que é uma associação de diversas empresas envolvidas em desenvolvimento de tecnologias multimídia, como ATI, Intel, NVIDIA, entre outras. A API OpenMAX foi desenvolvida para padronizar e facilitar o desenvolvimento de aplicações multimídia entre diversas plataformas, inclusive a integração entre

diferentes dispositivos. Ela é dividida em três camadas: AL(Application Layer), IL(Integration Layer) e DL(Development Layer). A figura 2.13 representa as funções de cada camada, bem como seus elementos e sua hierarquia. Para a finalidade de sincronismo da captura, é de interesse conhecer alguns detalhes da camada intermediária, a IL. Ela é responsável por realizar a interface entre a implementação de baixo nível (DL) e a de aplicação (AL).

Figura 2.13 - Diagrama exemplificando os elementos de cada camada da API OpenMAX



Fonte: [18]

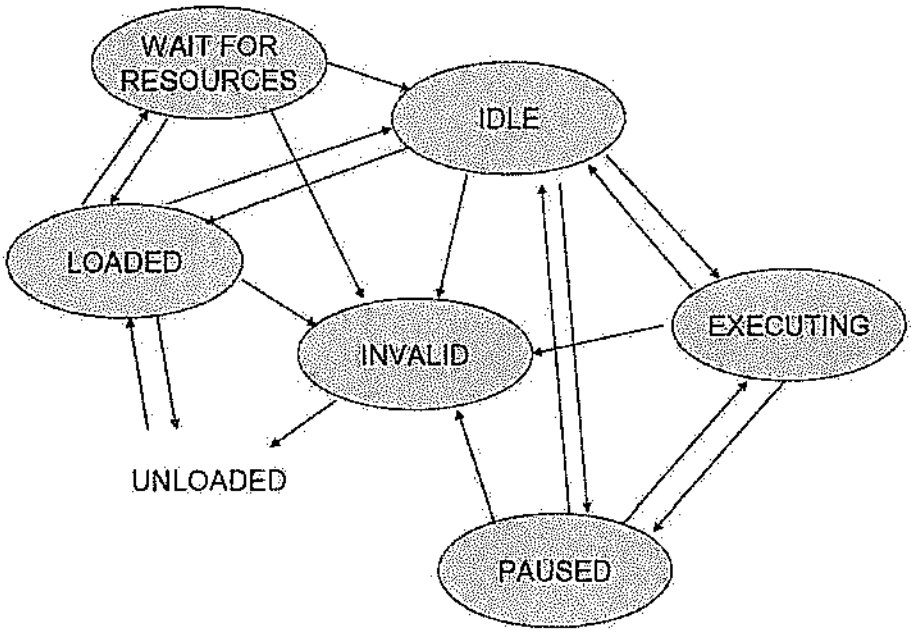
A API OpenMAX IL é constituída por componentes que encapsulam funções de processamento multimídia (em software ou hardware dedicado) que podem ser de fabricantes diferentes e possuir configurações diferentes, provendo uma interface padronizada para acessar essas funções, permitindo que os componentes se comuniquem normalmente e sejam facilmente substituíveis. Cada componente pode representar, por exemplo, um codificador de áudio ou um decodificador de vídeo, ou ainda um relógio para controlar a captura de vídeo. Os componentes possuem pelo menos uma porta pela qual se comunicam com outros componentes ou com o

programa cliente. Portas de um componente podem ser ligadas a portas de outros componentes através de tuneis, de modo que o fluxo de informação é direto e independe do programa cliente.

Cada componente possui estados de execução, que podem ser: UNLOADED, LOADED, WAIT FOR RESOURCES, IDLE, EXECUTING, PAUSED ou INVALID. A figura 2.14 mostra as possíveis transições entre os diferentes estados. Um provável ponto de interesse para a sincronização é a transição entre o estado IDLE do componente de captura e o estado EXECUTING. Outro potencial interesse para a sincronização é a manipulação de componentes de relógio. Esta possibilidade, entretanto, não será abordada neste projeto.

Para a captura e compressão são utilizados três componentes: Camera Component, Preview Component e Encoder Component. O Camera Component acessa a câmera CSI, configura-a e captura as imagens por meio dela, tendo como saída em sua porta as imagens no formato RAW. O Preview Component, além de possibilitar a visualização em um display das imagens sendo capturadas, é utilizado para possibilitar correções de luminosidade e de cores nas imagens da câmera. O Encoder Component recebe as imagens adquiridas e as comprime para um formato suportado escolhido. Como previamente observado, a transição do Camera Component do estado IDLE para o estado EXECUTING é utilizado para sincronizar a captura. A instrução anterior à da transição dos estados é bloqueada e aguarda um sinal de disparo, que pode ser dado por um sinal do sistema operacional ou pelo expirar de um *timer*.

Figura 2.14 - Estados de um componente da API OpenMAX



Fonte: [18]

O código fonte do programa de captura não foi adicionado ao apêndice por ser muito extenso. O programa Raspistill foi modificado para levar em conta o bloqueio anterior à transição de estado do Camera Component. Além disso, muito tempo era gasto em inicialização e finalização dos componentes, pois o programa foi desenvolvido para permitir uma única captura a cada chamada. Para reduzir o tempo gasto entre uma captura e outra, modificou-se o programa para que, uma vez inicializados os componentes, ele executasse um laço que aguarda o sinal de disparo e realize a captura, modificando o nome do arquivo de saída. Uma vez capturada a imagem, comprimida e gravada em um arquivo, ela é enviada para uma fila para ser transmitida ao computador mestre.

2.5.3 Transmissão e fila

A transmissão das imagens capturadas é realizada utilizando o protocolo TCP/IP. Uma vez aberta a imagem para o envio, o seu tamanho é determinado e enviado em uma *string* específica, e este valor será utilizado para controlar o envio do arquivo. O servidor contabiliza o número de bytes a cada recebimento de pacote e, uma vez que a quantidade de bytes se iguala à quantidade enviada no início, o servidor dá por encerrada essa transmissão, fecha o arquivo e se prepara para receber outro arquivo.

Por conta do seu controle de acesso ao meio, o WiFi não divide igualmente o tempo de transmissão entre os dispositivos, de modo que dispositivos mais distantes do AP podem acabar tendo menores taxas de transmissão. Além disso, existe uma variação estatística do tempo de entrega de um pacote e das taxas de transmissão, devido a colisões de pacotes, filas, etc. Para tentar reduzir os efeitos citados, pode-se implementar um sistema de *token ring*, uma vez que os dispositivos do sistema são previamente conhecidos e fazem usos similares do meio de transmissão. No sistema de *token ring*, os dispositivos recebem um marcador que circula entre eles, de modo que somente um deles possui o marcador em um instante de tempo e, somente quem possui-lo poderá transmitir. Este método reduz a quantidade de colisões, porém à custa do tráfego de controle, que é ainda mais agravado pelo fato de ser implementado na camada de aplicação. Para garantir um maior controle do sistema, o mestre coordena os dispositivos enviando um pacote com o identificador do dispositivo que receberá o marcador, além de um valor de *timeout*. Se o dispositivo não receber nenhum outro pacote referente ao marcador, ao se passar o tempo referente ao *timeout*, o dispositivo automaticamente liberará o marcador, parando de transmitir as imagens até que receba o marcador novamente. O dispositivo pode, ainda, antes de passado o tempo referente ao *timeout*, receber outro pacote exigindo que ele libere o marcador antecipadamente ou que aumente o valor do *timeout*, para poder continuar transmitindo por mais tempo.

Além do sistema de *token ring*, pode-se aproveitar o sincronismo dos dispositivos para implementar uma espécie de TDMA(*Time Division Multiple Access*), de modo que cada dispositivo receba uma janela de tempo na qual pode transmitir, não podendo transmitir no restante do tempo dentro de um período específico. Uma desvantagem desse método é que se devem realocar as janelas de tempo toda vez que um dispositivo entrar ou sair da rede. Uma vantagem é que, uma vez mantidos constantes os dispositivos conectados à rede, o custo de tráfego de controle é minimizado em comparação ao caso do *token ring*. Deve-se, também, considerar que a eficiência desse sistema depende diretamente da qualidade da sincronização dos dispositivos.

Para contornar os problemas gerados pela natureza estatística da taxa de transmissão da rede, pode-se armazenar as imagens capturadas até que elas sejam enviadas. Para melhor organizar os arquivos a serem transmitidos, programou-se uma fila que contém os descritores dos arquivos que contém as imagens. Cada vez que uma nova imagem é capturada e codificada, ela é inserida na fila e, se a fila estiver cheia, ela substitui a imagem mais antiga da fila. O compromisso entre latência e quantidade de quadros perdidos pode ser ajustado através do tamanho da fila.

2.5.4 Alimentação

O Raspberry Pi é normalmente alimentado por uma porta Micro USB, com uma tensão de 5V, que é diretamente alimentada aos periféricos conectados à porta USB A, exigindo, portanto, que a tensão de entrada da porta Micro USB seja bem regulada. Além disso, a fonte deve suportar pelo menos 700mA para o modelo B com periféricos USB. A maioria das fontes de celular atuais adotam estes padrões fornecendo corrente suficiente.

A fim de dar mais liberdade para o posicionamento das câmeras, decidiu-se utilizar pilhas para alimentá-las. No entanto, não é possível combinar pilhas para atingir as

tensões necessárias e um mero divisor de tensão não proveria a estabilidade necessária na entrada. Por isso, optou-se por utilizar um regulador chaveado de tensão que provesse 5V na saída com estabilidade suficiente para garantir o funcionamento correto do circuito. Foi escolhido um regulador disponível comercialmente muito utilizado em aeromodelismo para a alimentação de servo-motores, um UBEC (*Universal Battery Eliminator Circuit*), que provê até 3A a 5V.

Figura 2.15 - Foto do regulador UBEC

Fonte: [19]

2.6 *Testes e Resultados*

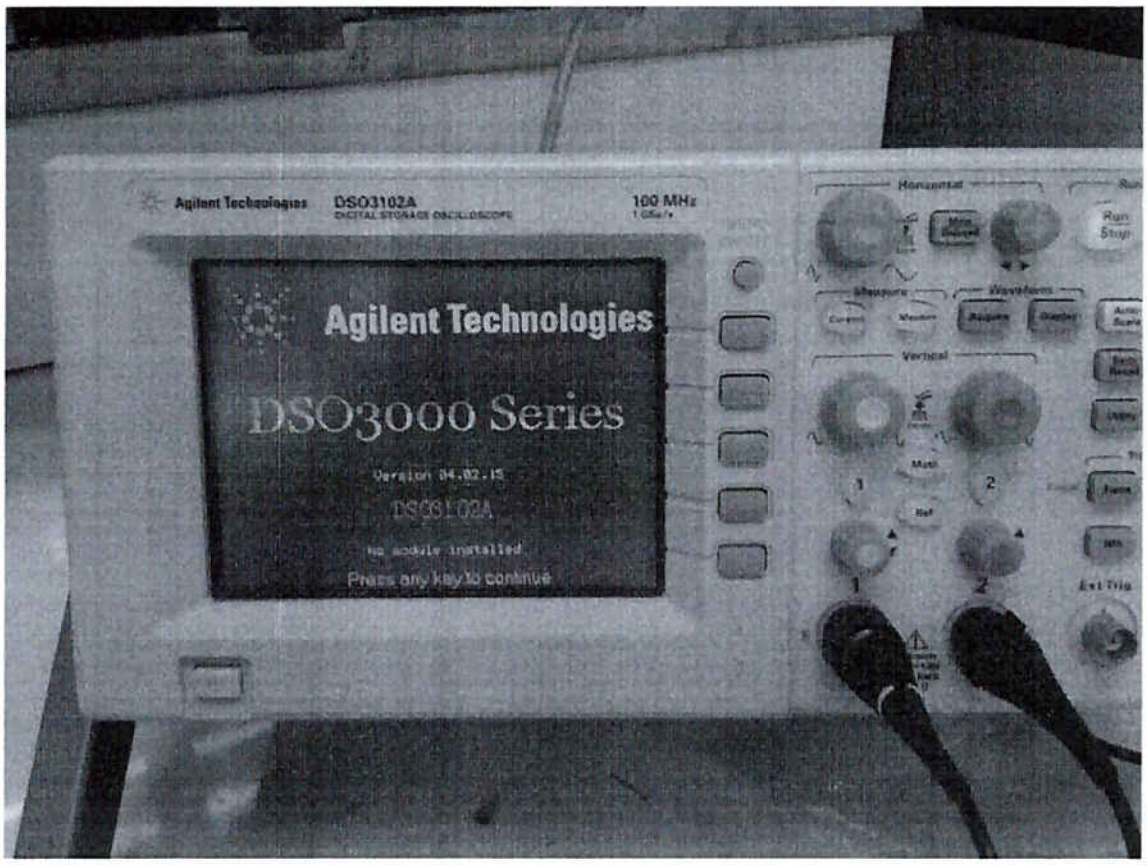
O funcionamento de cada módulo foi testado separadamente para facilitar a depuração do sistema e para identificar possíveis gargalos. Depois, o sistema foi testado como um todo. As seções seguintes detalham os procedimentos dos testes, bem como seus resultados.

2.6.1 Sincronização

A fim de testar a precisão e a exatidão da sincronização dos sinais de disparo das câmeras, pinos GPIO dos dispositivos foram utilizados como saída dos sinais de disparo, sendo alternados entre os níveis lógicos baixo (0V) e alto(3.3V) a cada momento que a função de captura seria ativada. Os pinos foram conectados a um

osciloscópio DSO3102A, da Agilent Technologies, de modo que a diferença de tempo entre os sinais dos dois dispositivos pôde ser determinada com suficiente precisão. A borda de subida do canal 1 foi utilizada como *trigger*. As medidas constituíram na variação da diferença de tempo entre os sinais de disparo dos dois dispositivos ao longo do tempo, para intervalos de 1s dos procedimentos de sincronização e 1s de intervalo entre disparos, bem como para nenhum procedimento de sincronização e 1s entre disparos.

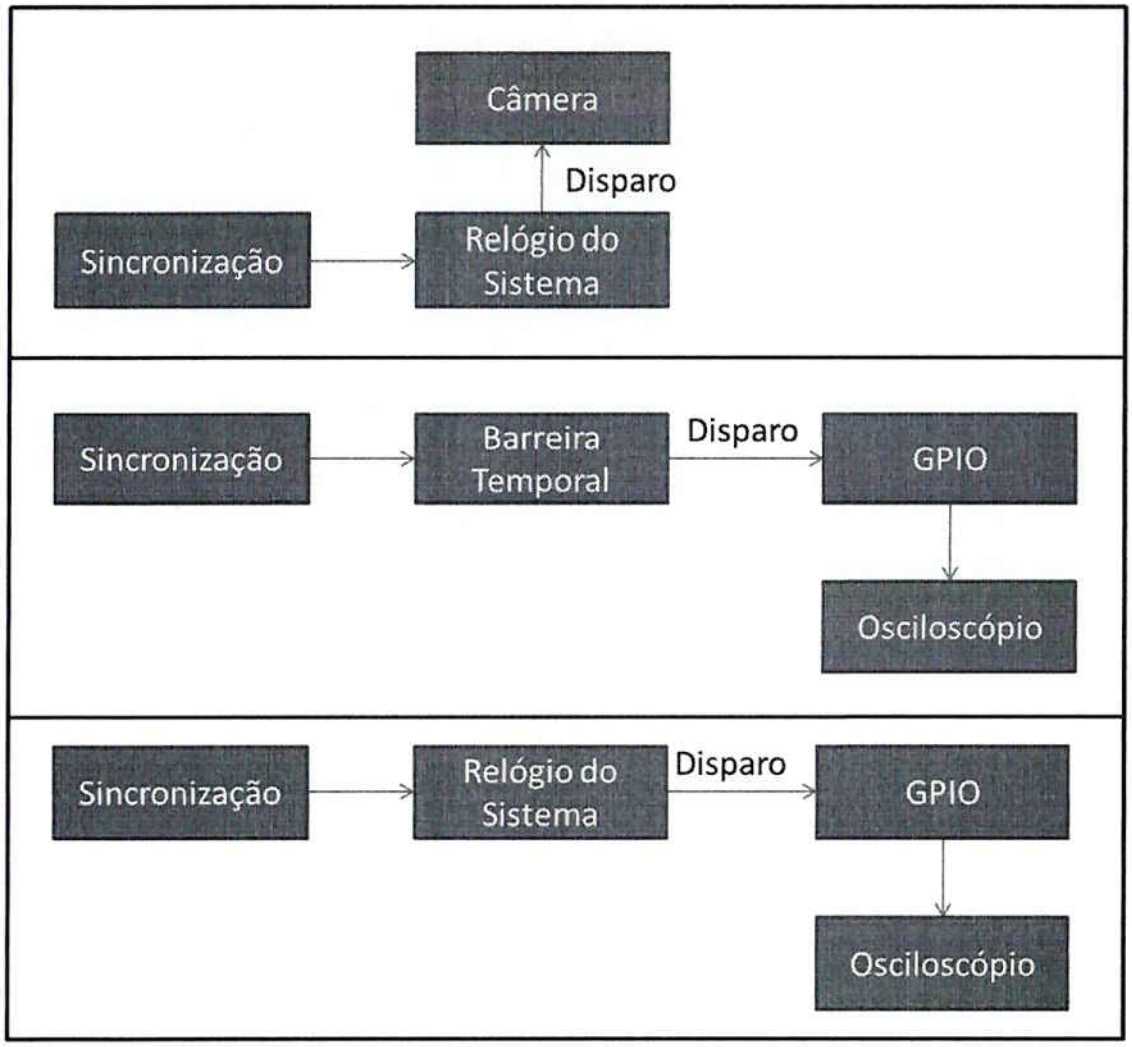
Figura 2.16 - Osciloscópio utilizado no teste de sincronização



Os três métodos de sincronização discutidos na seção de implementação foram testados. O método de barreira temporal foi testado sem uso de roteador e sem transferência de arquivos. Os outros dois métodos foram testados em situações tanto com o uso de roteador como sem, bem como com transferência de arquivo pelo comando scp como sem.A figura 2.17 exemplifica os processos de disparo utilizados nos testes. No primeiro quadro, o relógio do sistema é atualzado pelo processo de sincronização e um processo verifica o relógio do sistema até que ele

atinja certo valor realizando, então, o disparo. O disparo corresponde ao momento em que a imagem deve ser capturada. Para testar o sincronismo dos disparos, optou-se por utilizar o GPIO da placa, de modo que o momento do disparo passa a equivaler aos dois quadros de baixo da figura, nos quais o sinal de disparo não é enviado às câmeras mas sim aos GPIOs para comparação em um osciloscópio.

Figura 2.17 - Diagrama exemplificando os processos de disparo utilizados nos testes.



As figuras 2.18 a 2.22 exemplificam com mais detalhes dos dispositivos e métodos envolvidos os diferentes tipos de sincronização e disparo.

Figura 2.18 - Diagrama exemplificando o processo de sincronização e disparo pelo método da baerreira temporal, sem uso de roteador.

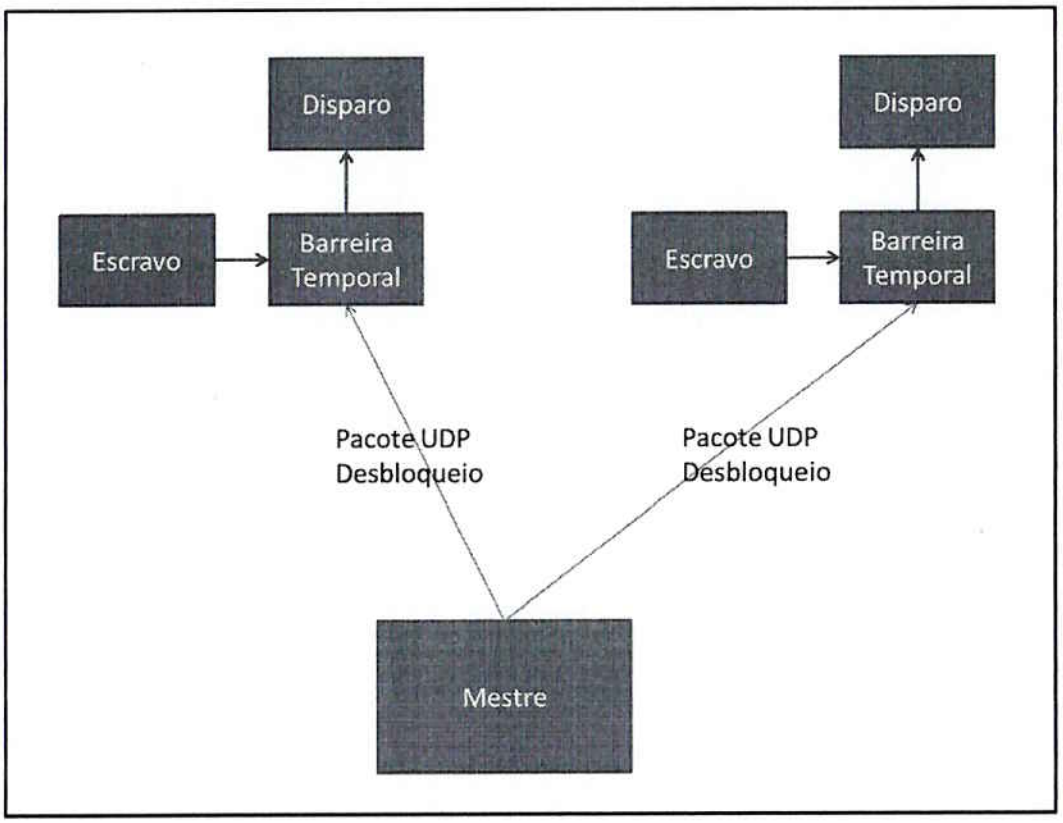


Figura 2.19 - Diagrama exemplificando o processo de sincronização e disparo pelo método timestamp multicast sem o uso de roteador

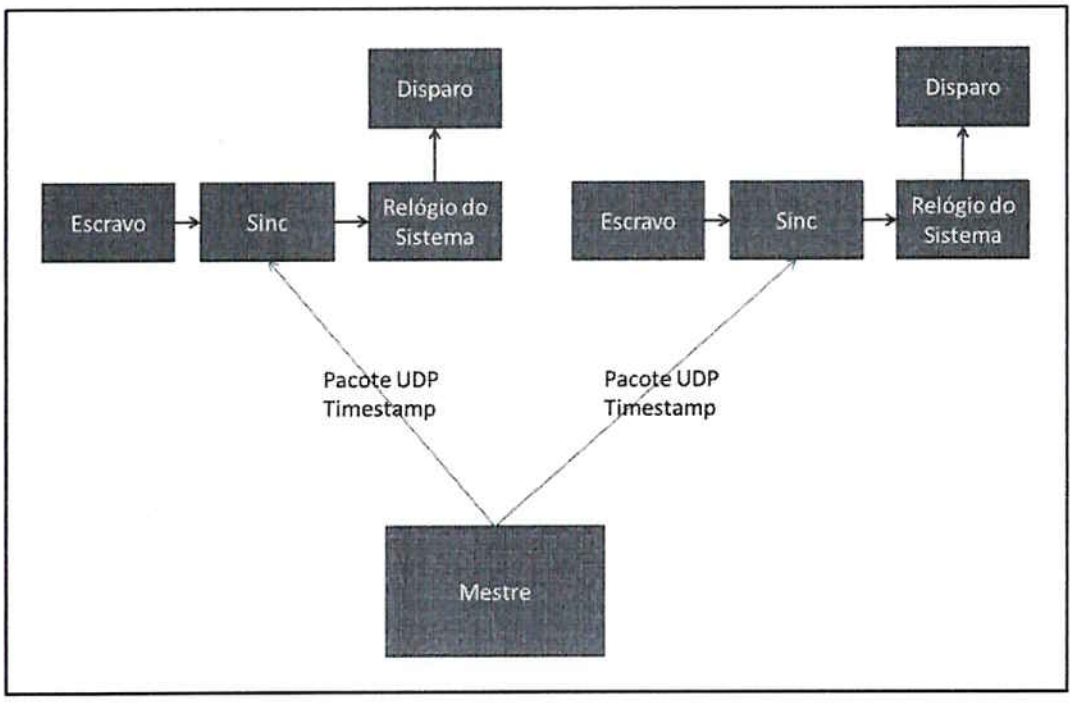


Figura 2.20 - Diagrama exemplificando o processo de sincronização e disparo pelo método adaptado do PTP, sem o uso de roteador.

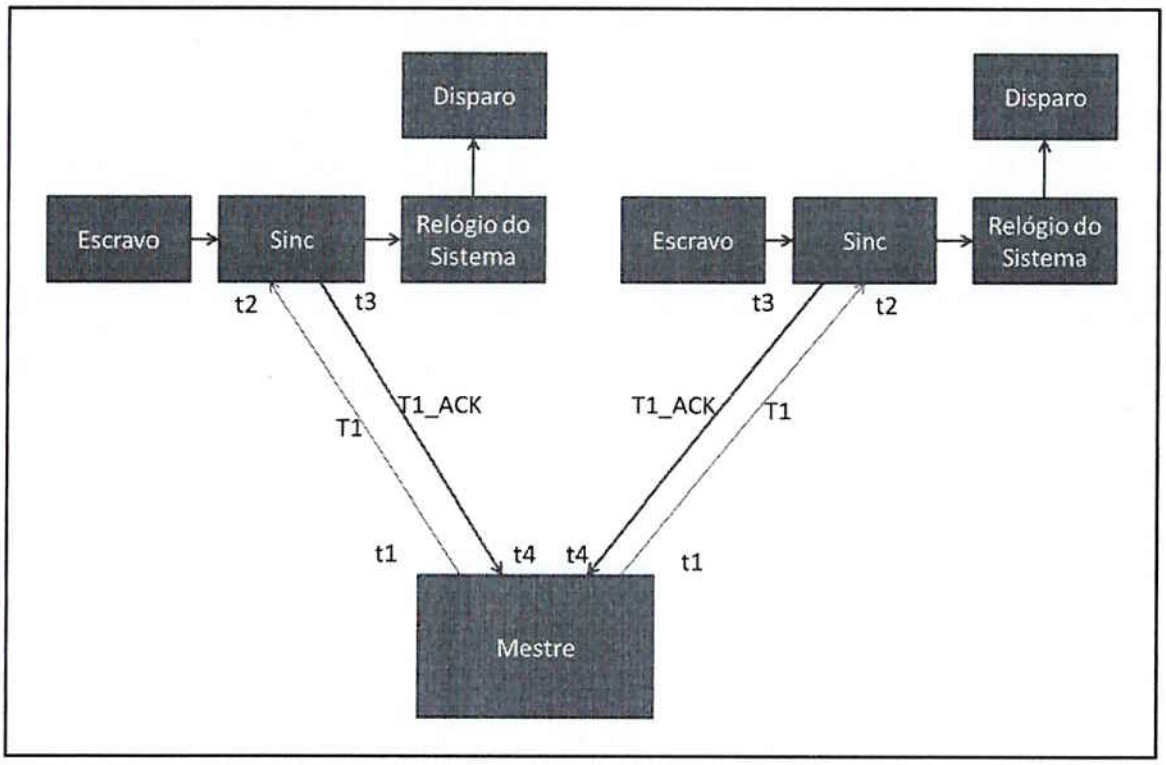


Figura 2.21 - Diagrama exemplificando o processo de sincronização e disparo pelo método timestamp multicast com o uso de roteador.

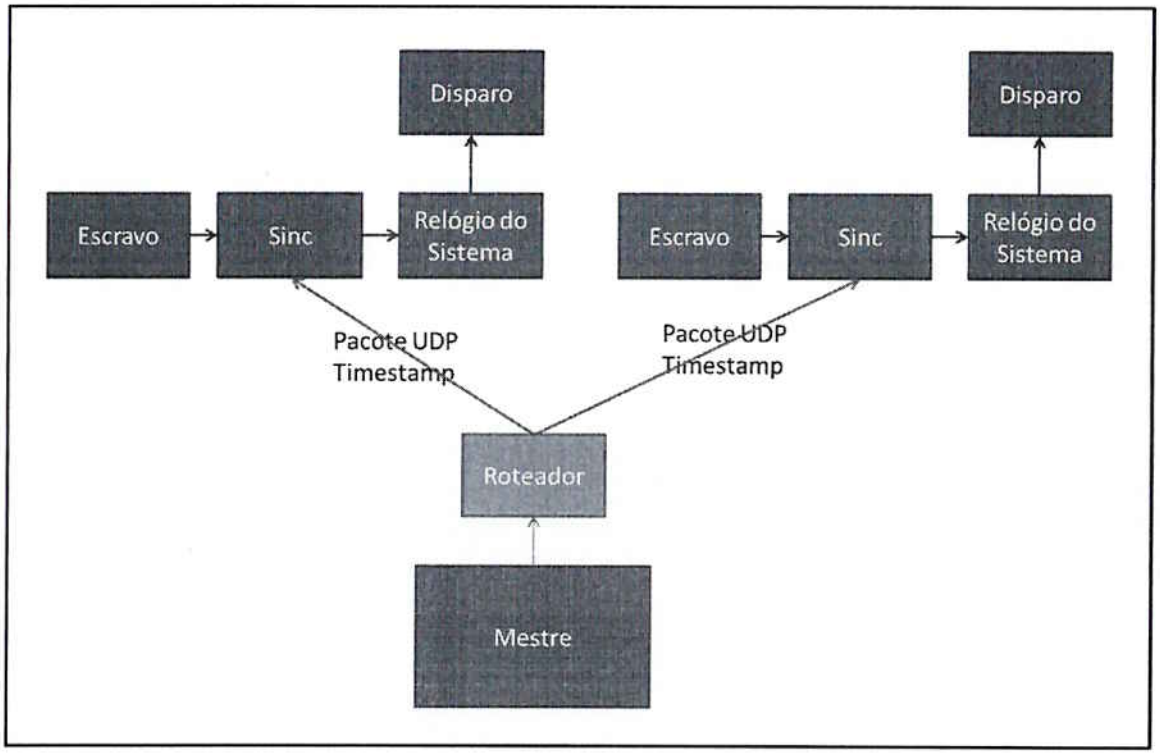


Figura 2.22 - Diagrama exemplificando o processo de sincronização e disparo pelo método adaptado do PTP, com uso de roteador

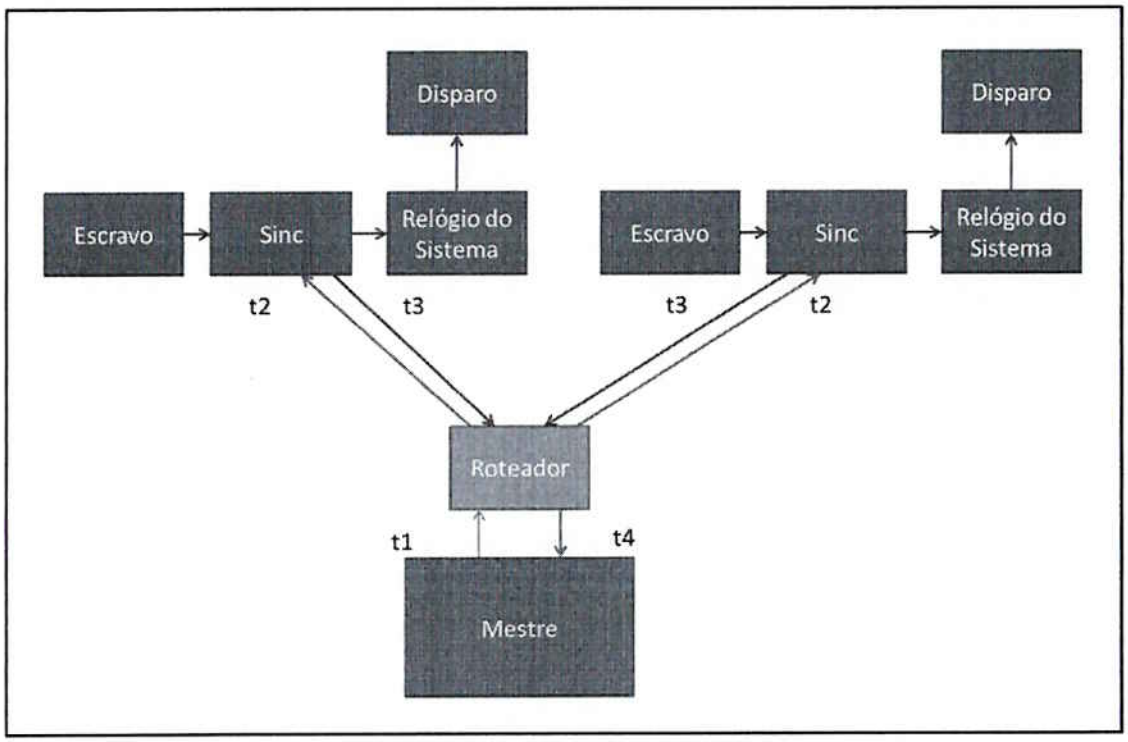
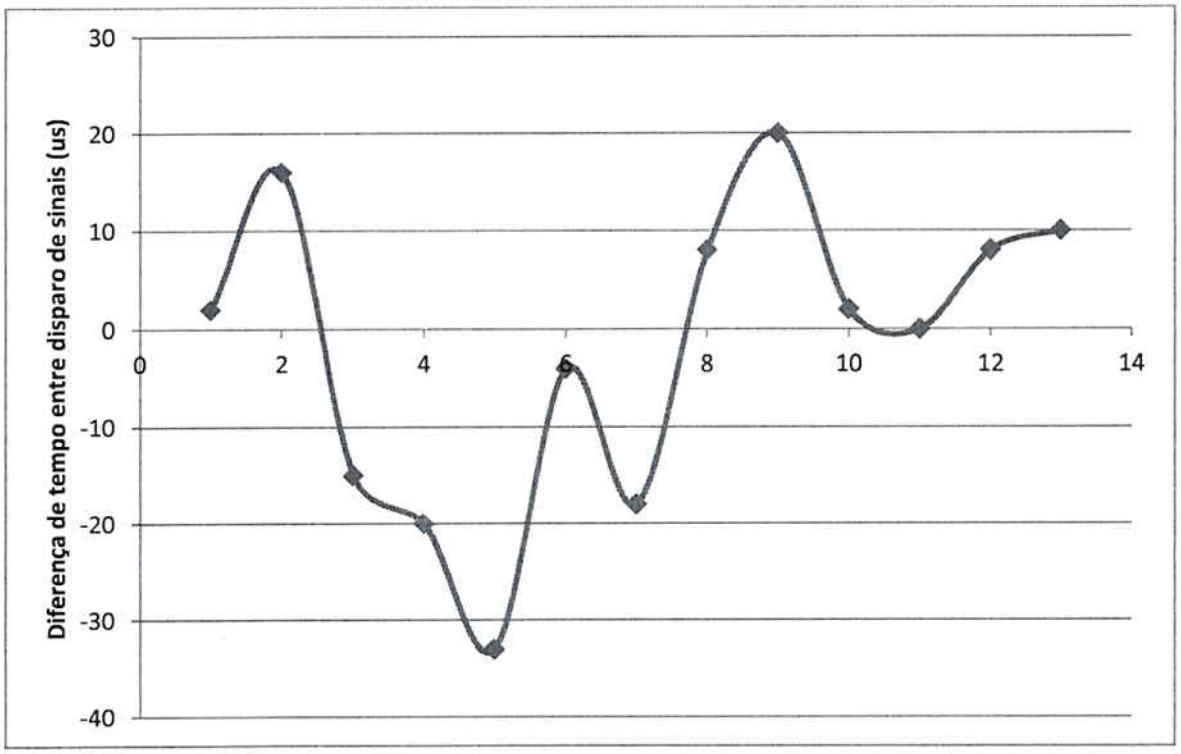
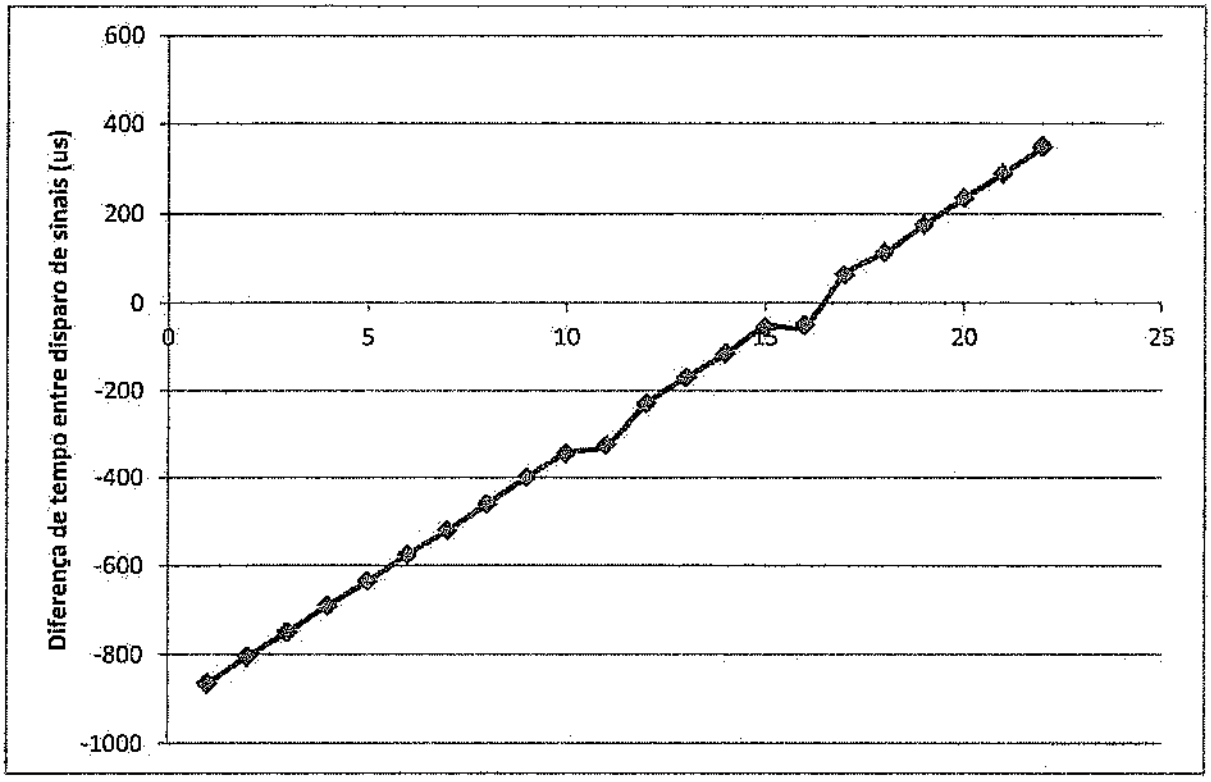


Gráfico 1 - Resultado do teste de sincronização utilizando barreira temporal



O gráfico 1 mostra os resultados do teste de sincronização utilizando o método de barreira temporal sem o envio de *timestamp* pelo mestre. Para este teste, o computador (mestre) foi utilizado como *Access Point* (AP) e para isso utilizou-se um programa do Windows 7, chamado netsh, que emula um AP. Neste caso, os sinais de disparo eram gerados a cada vez que um pacote recebido atingia a barreira temporal, e a diferença de tempo entre os sinais de disparo de dois dispositivos foi medida e plotada no gráfico 1.

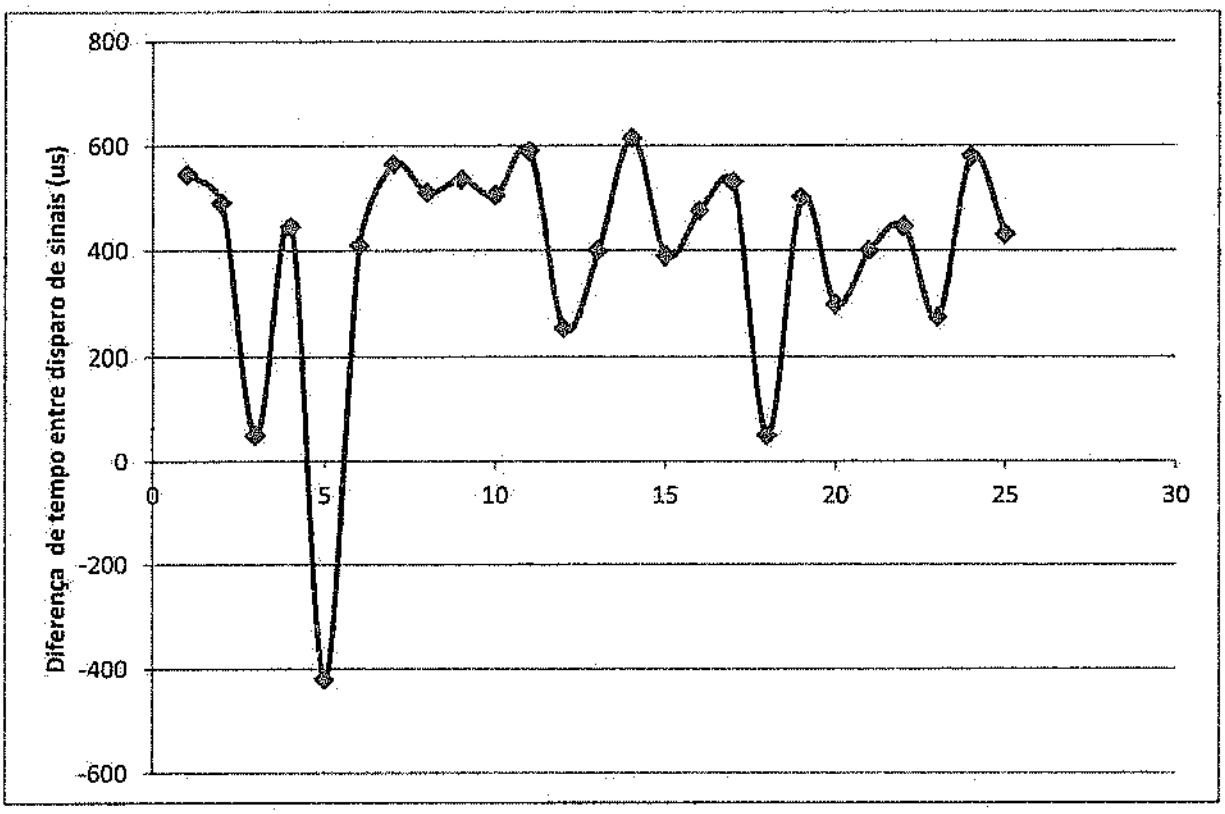
Gráfico 2 - Resultado do teste sem sincronização



O gráfico 2 mostra o resultado dos testes de sincronização utilizando o relógio. Primeiramente, o mestre enviou um único *timestamp* para os dispositivos com as câmeras para que elas ajustassem seu relógio e, a partir daí, a cada segundo geravam um sinal de disparo. Devido a problemas de conexão que surgiram no teste da barreira temporal mencionado, utilizou-se neste segundo teste, um roteador. A diferença de tempo entre os sinais de disparo de dois dispositivos foi medida e plotada no gráfico 2. Através dessas medidas pode-se estimar o *drift* entre os

relógios dos dois dispositivos. Fica bem claro que existe uma relação linear entre as variações de tempo dos dois dispositivos, pois a curva da variação das diferenças de tempo entre as duas câmeras é uma reta inclinada. Pode-se considerar que, em curto prazo, as frequências dos relógios dos dois dispositivos são constantes, levando à uma variação linear da diferença de tempo entre os dois disparos.

Gráfico 3 - Resultado do teste de sincronização utilizando *timestamp multicast* com roteador



O gráfico 3 mostra o resultado dos testes de sincronização utilizando o processo de *timestamp multicast*, no qual o mestre enviava seu *timestamp* periodicamente (neste teste, a cada segundo) para os dispositivos com as câmeras, de forma que elas ajustassem seu relógio também periodicamente. A cada segundo, sinais de disparo eram gerados por cada dispositivo. A diferença de tempo entre os sinais de disparo de dois dispositivos com a utilização de um roteador foi medida e plotada no gráfico 3 e as medidas sem o roteador estão apresentadas no gráfico 4.

Gráfico 4 - Resultado do teste de sincronização utilizando *timestamp multicast* sem roteador

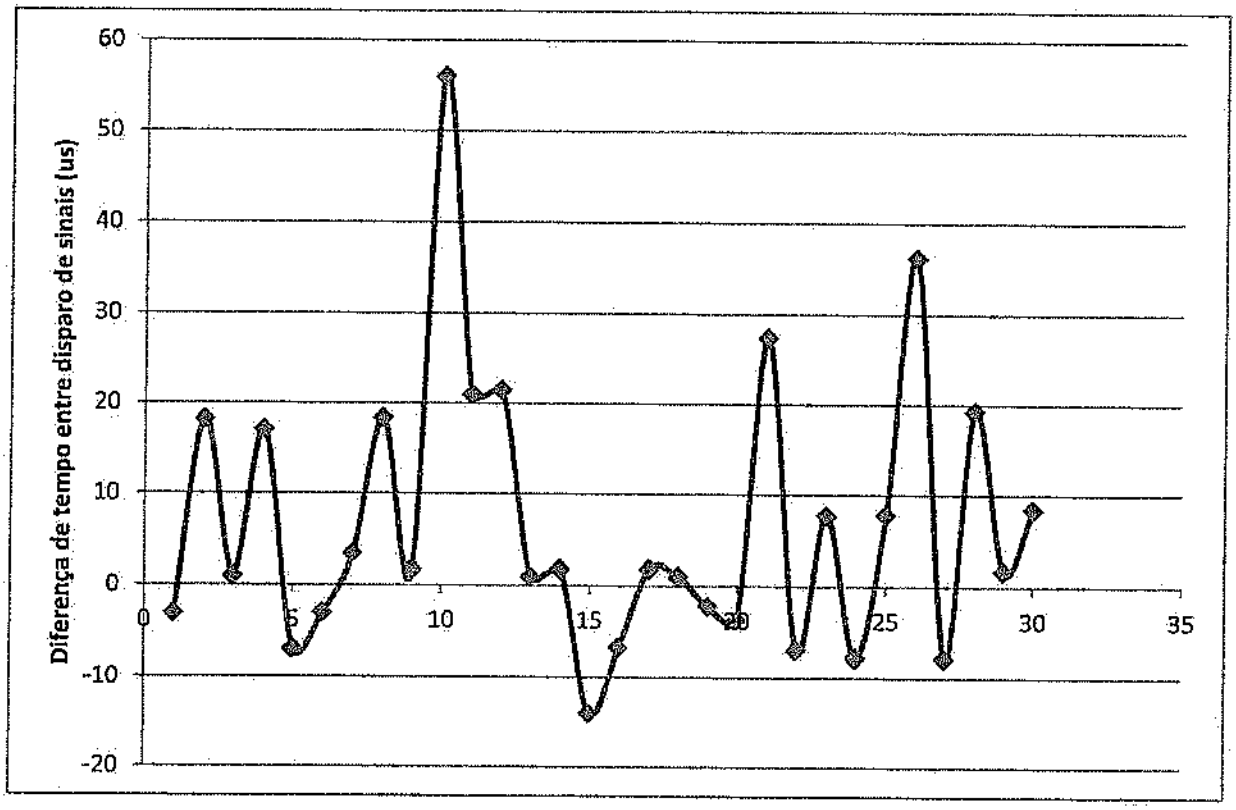


Gráfico 5 - Resultado do teste de sincronização utilizando adaptação do PTP com roteador

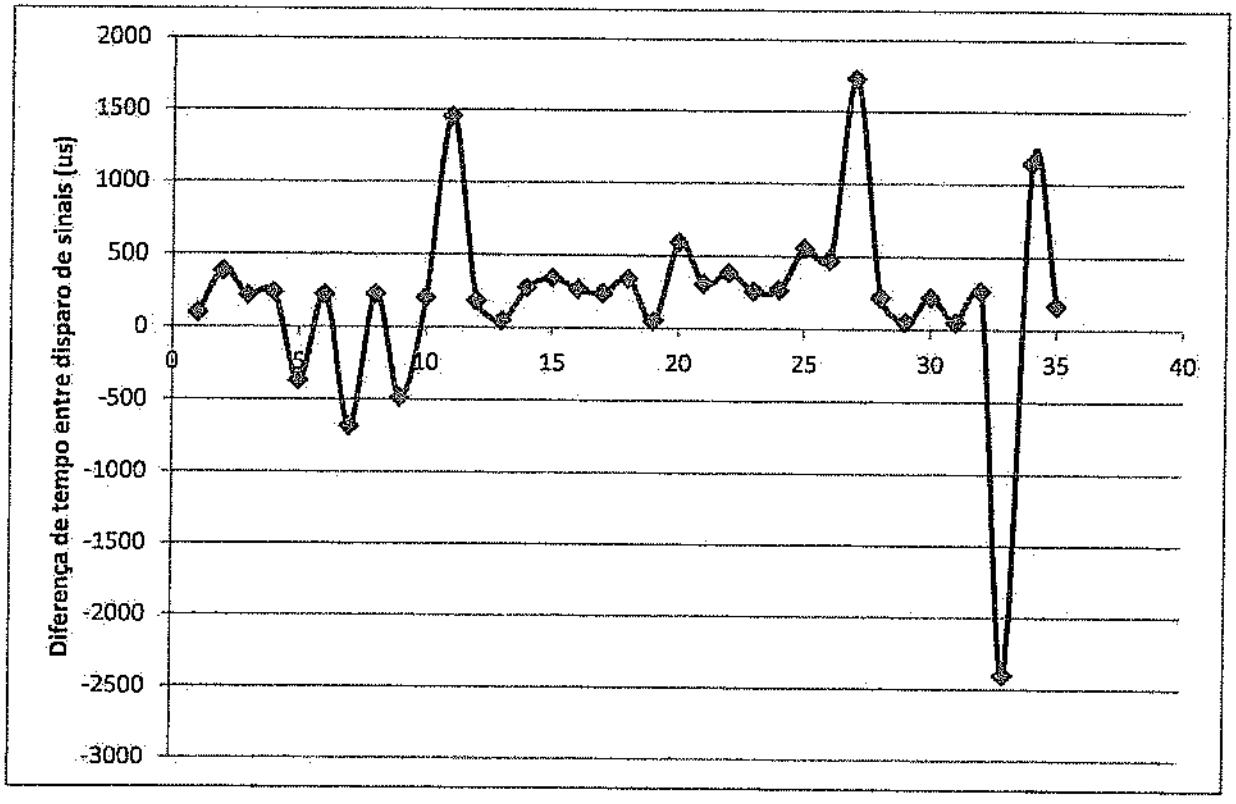


Gráfico 6 - Resultado do teste de sincronização utilizando adaptação do PTP sem roteador

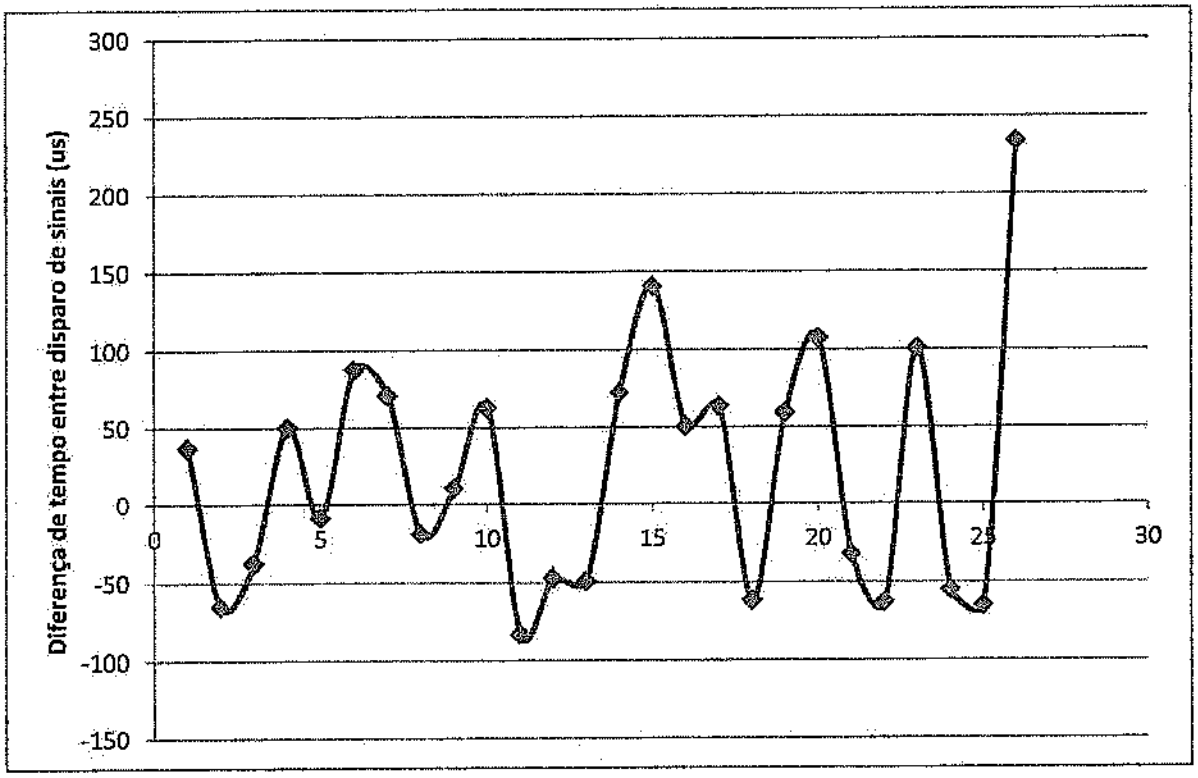


Gráfico 7 - Resultado do teste de sincronização utilizando adaptação do PTP com roteador e com transferência de arquivos via scp

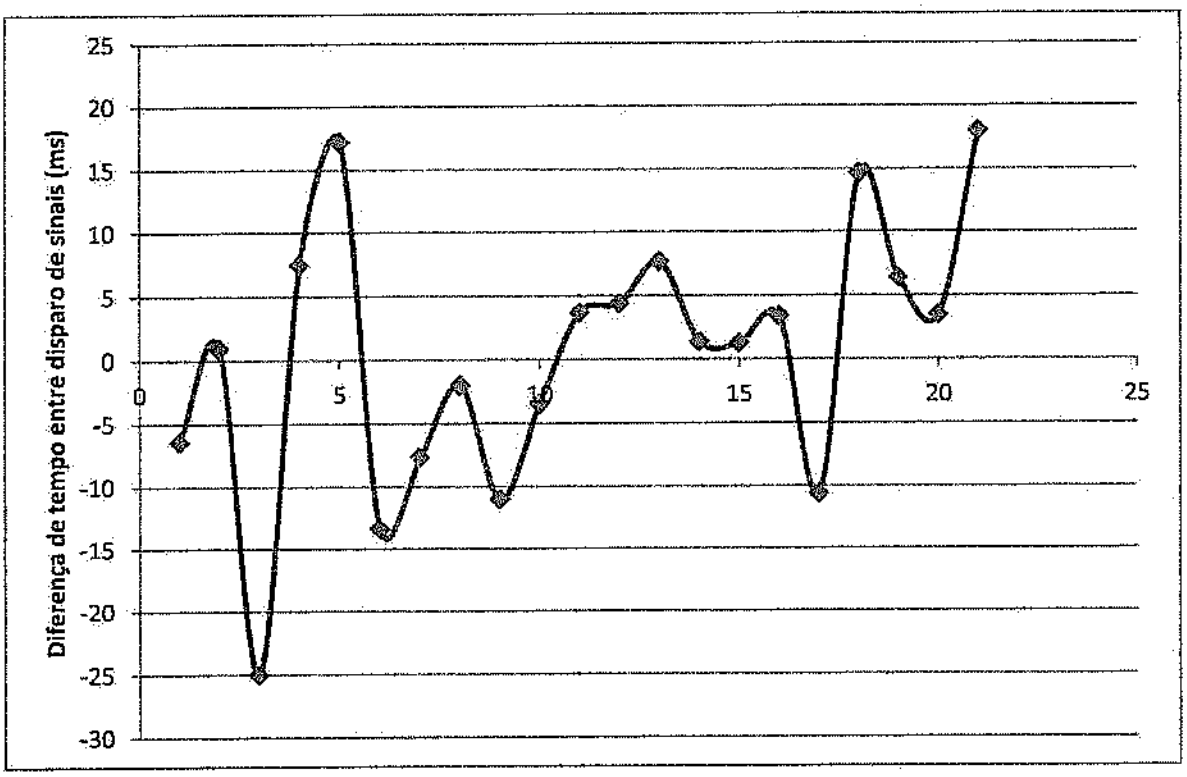


Gráfico 8 - Resultado do teste de sincronização utilizando adaptação do PTP sem roteador e com transferência de arquivos via scp

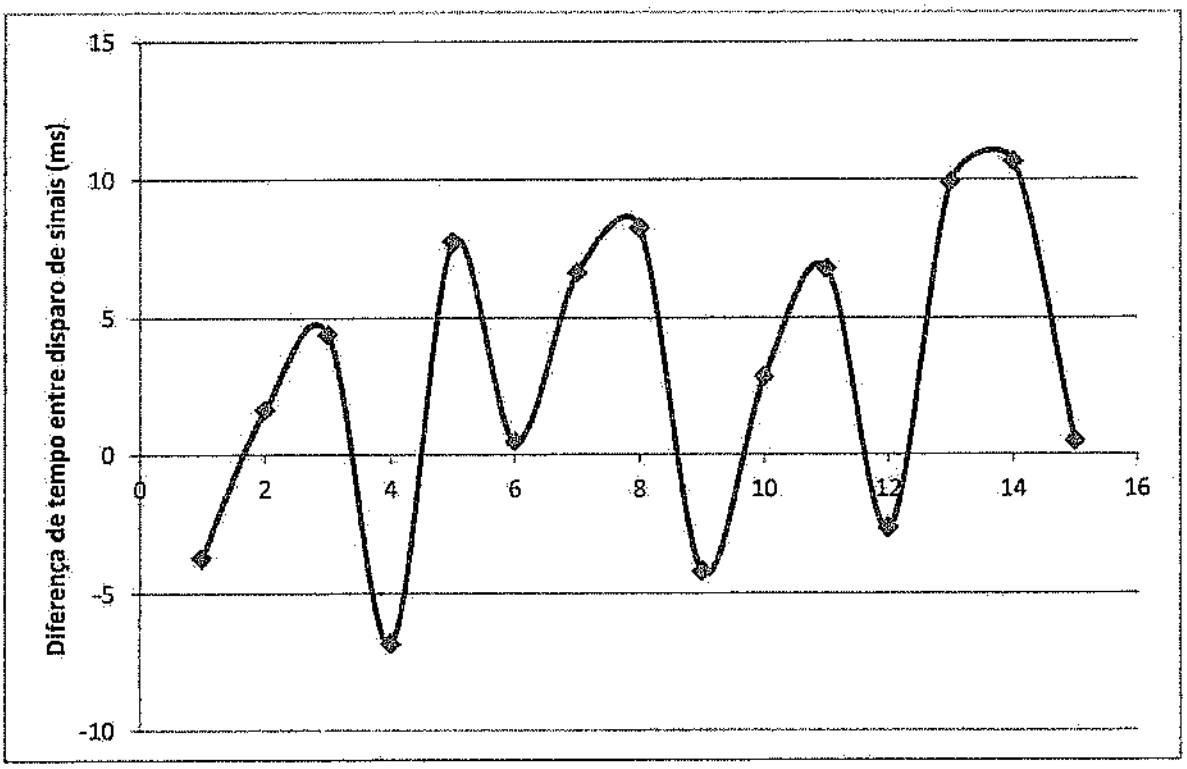


Gráfico 9 - Resultado do teste de sincronização utilizando timestamp multicast com roteador e com transferência de arquivos via scp

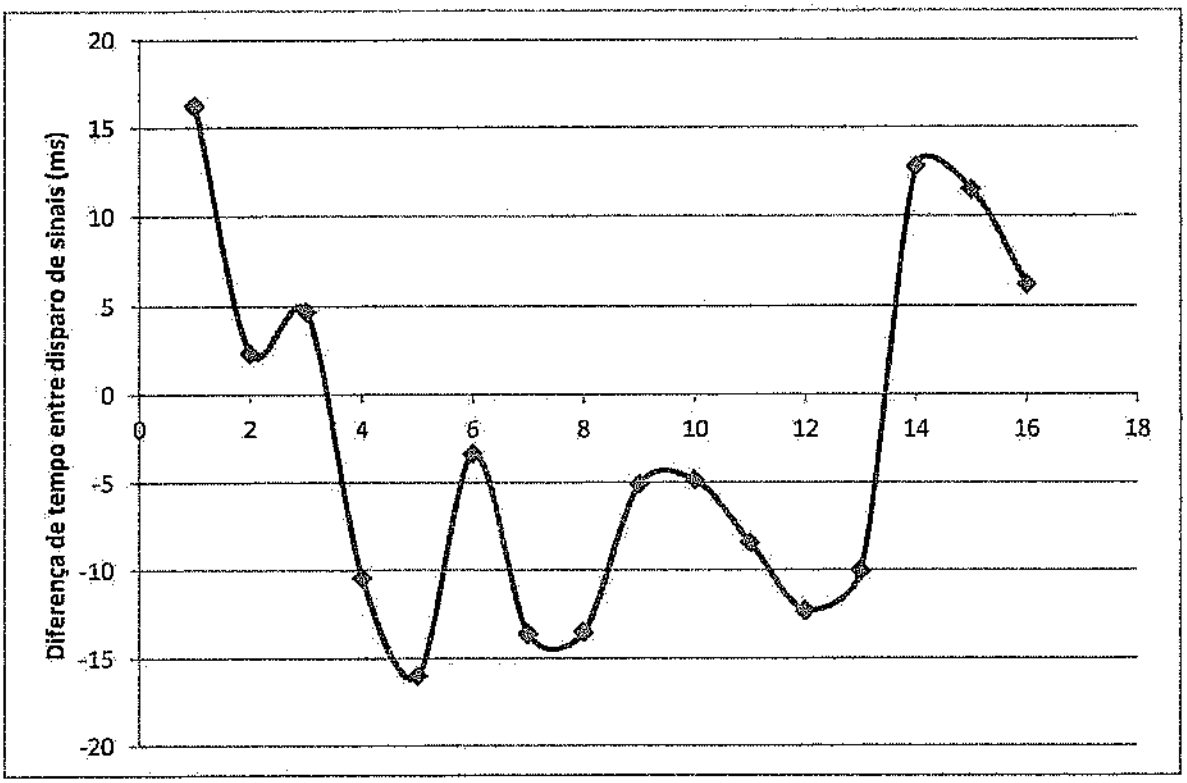


Gráfico 10 - Resultado do teste de sincronização utilizando timestamp multicast sem roteador e com transferência de arquivos via scp

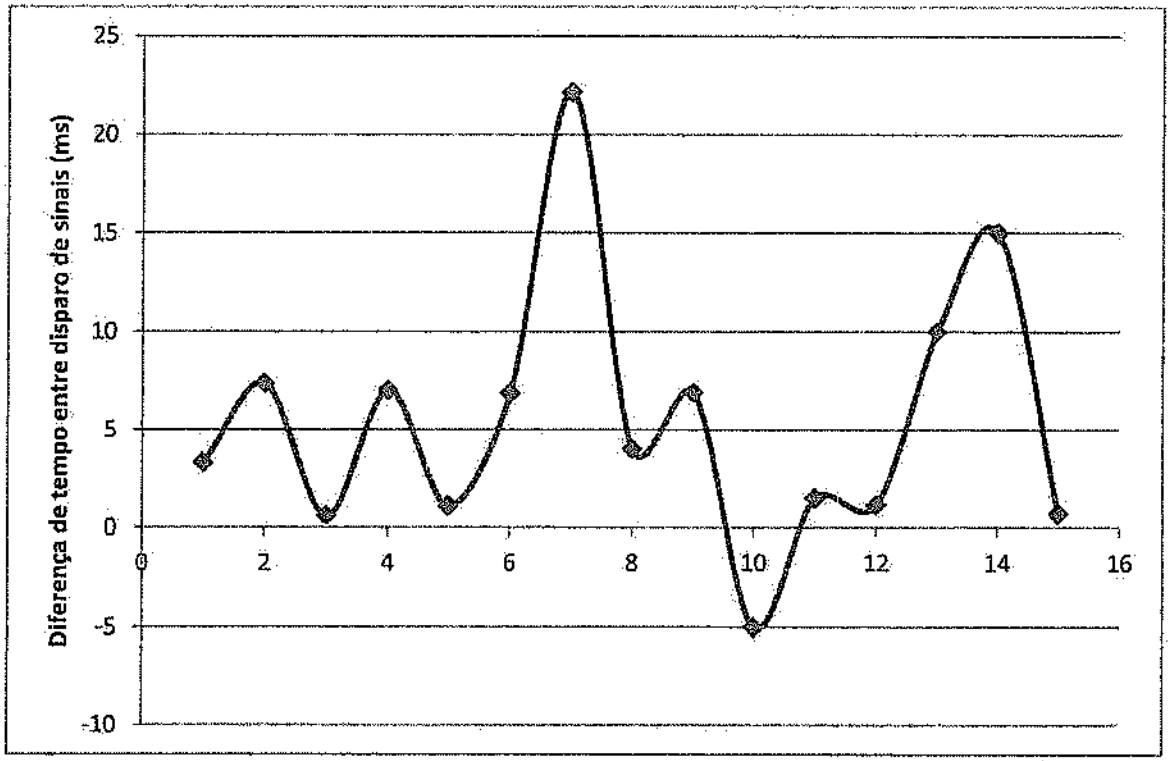


Tabela 2.2- Tabela comparativa dos resultados dos testes de sincronização

Processo de sincronização	Roteador	Transferência de arquivos via scp	Média	Desvio padrão
Timestamp multicast	Sem	Sem	6,9 us	14,927 us
		Com	5,5 ms	6,408 ms
	Com	Sem	394,8 us	220,315 us
		Com	-2,7 ms	10,071 ms
Adaptação do PTP	Sem	Sem	21,6 us	76,959 us
		Com	2,8 ms	5,321 ms
	Com	Sem	216,1 us	626,545 us
		Com	0,5 ms	10,236 ms
Barreira temporal	Sem	Sem	-1,8 us	14,940 us

Para analisar as medidas obtidas, optou-se por calcular a média e o desvio padrão de cada conjunto. A tabela 2.1 exibe esses valores.

Das medidas obtidas e dos valores calculados pode-se chegar a algumas conclusões. O fato dos valores medidos apresentarem valores negativos e positivos indica que um fator de variação são as trocas de contexto devido ao ambiente multitarefa. Isto é ainda mais evidenciado na diferença de desvios padrões em situações com ou sem envio de arquivo. Além disso, existe uma variação estatística gerada pelo acumulo de pacotes em filas no roteador e nas camadas de rede dos dispositivos, assim como a retransmissão de pacotes perdidos.

Um fator a ser considerado nos processos de sincronização é o fato dos *timestamps* terem sido tirados na camada de aplicação, gerando várias chamadas ao sistema, possibilitando troca de contexto antes do real envio do pacote. Mais correto seria tirar *timestamps* quando o pacote passar da camada MAC para a camada física. Isto, porém, depende de suporte dos *drivers* e dos dispositivos.

Pode-se observar que os desvios padrões e as médias obtidas nos processos sem transmissão de arquivo são bastante satisfatórios. Já com a transferência, os valores são mais altos do que o que se consideraria satisfatório, tendo em vista o frame lock de 125μs disponível em câmeras sincronizadas em FireWire e que a diferença entre quadros de um vídeo de 30 quadros por segundo é de 33ms.

Para tentar reduzir a variação gerada pela transferência de arquivos, pode-se implementar uma janela de tempo na qual nenhum arquivo é enviado e somente a sincronização e a captura ocorrem. Além disso, podem-se empregar métodos para corrigir o *drift* entre os relógios, de modo a reduzir a frequência com a qual se faz necessário novas sincronizações.

Se comparados os valores calculados com e sem roteador, pode-se concluir, também, que o roteador gera mais incerteza no tempo de transmissão de cada pacote, de modo que a sincronização de vários dispositivos provavelmente se daria melhor utilizando o processo adaptado do PTP, uma vez que ele leva em consideração uma estimativa do tempo de transmissão do pacote, o que não é considerado nos outros métodos de sincronização.

2.6.2 Captura e compressão

Os testes de captura e compressão envolvem determinar a funcionalidade correta do sistema de captura e do sistema de compressão. Para este efeito foram tiradas fotos com diferentes taxas de compressão. Determinou-se o correto funcionamento da câmera, com qualidade compatível para o tipo de câmera utilizado. A tabela apresenta os tamanhos dos arquivos gerados pelo codificador JPEG para valores de qualidade diferentes.

Figura 2.23 - Resultado do teste de captura e compressão, com diferentes taxas de compressão

Tabela 2.3 - Tamanho do arquivo para diferentes taxas de compressão

Qualidade	Tamanho
100	3289 KB
75	3282 KB
50	2918 KB
20	1908 KB

Foi atingida uma taxa de captura de um quadro por segundo para as configurações previamente descritas, pois, por simplicidade, optou-se por capturar as imagens como imagens estáticas e não como quadros de um vídeo.

2.6.3 Transmissão

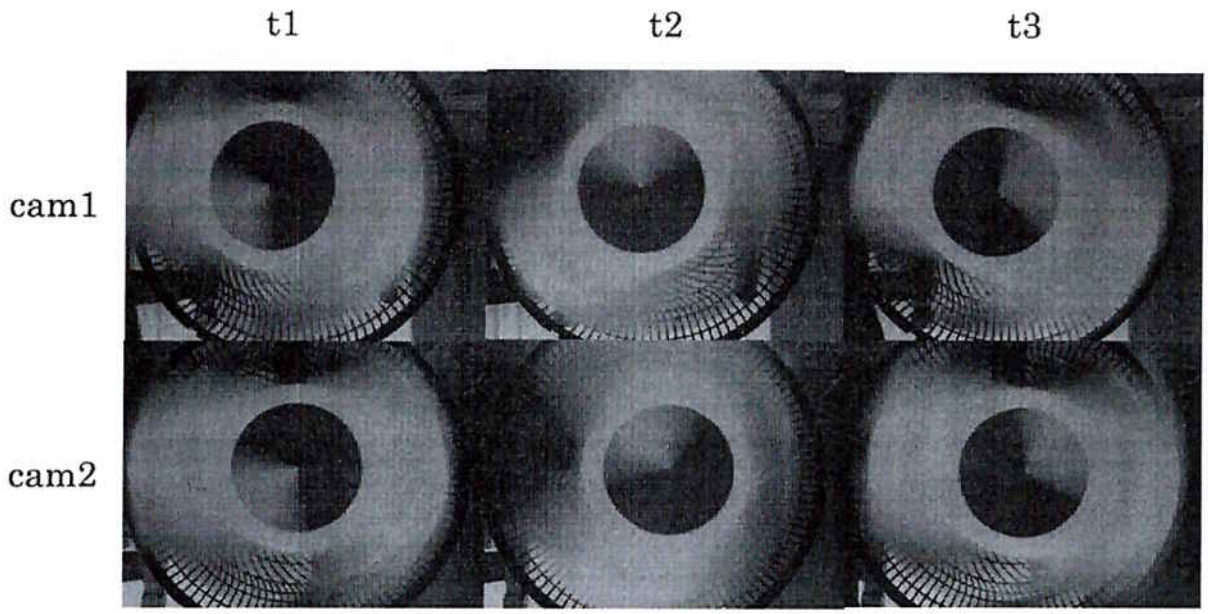
A transmissão foi realizada com sucesso e a taxa de envio das fotos foi suficiente para o envio dos quadros capturados. Alguns erros de estouro de fila foram detectados durante os testes, indicando que um controle maior da transmissão deve ser implementado. Como não foram implementados sistemas de controle de acesso ao meio na camada de usuário, as transmissões dos dispositivos foram feitas de modo concorrente. Para um maior número de câmeras, provavelmente se faz necessário o uso de métodos de controle de acesso ao meio que dividam o tempo entre os dispositivos.

2.6.4 Sistema integrado

O teste do sistema integrado consistiu em testar o funcionamento do dispositivo como um todo, capturando imagens de modo sincronizado, comprimindo-as e transmitindo-as para o PC, onde serão exibidas. Foi verificado o funcionamento contínuo do sistema e a sincronização das imagens foi verificada utilizando-se um padrão circular dividido em quadro cores que foi fixado a um ventilador. Pode-se

observar na figura 2.24 que, para quadros correspondentes, as posições das cores no círculo são indicativo de que as imagens estão sincronizadas dentro de uma tolerância, que poderá ser definida a partir dos ângulos das divisões entre as cores. Esta tolerância não foi determinada neste trabalho.

Figura 2.24 - Fotos em diferentes momentos de duas câmeras sincronizadas



3 Conclusão e Trabalhos futuros

Tendo em vista os processos implementados, concluímos que os resultados a partir do teste de sincronização ainda não são totalmente satisfatórios a ponto de poderem gerar imagens reconstruídas completamente livre de distorções, porém possuem significado relevante do ponto de vista da pesquisa. Três métodos de sincronização foram testados. A sincronização dos dispositivos sem transferência concomitante de arquivos apresentou resultados muito bons, da ordem de poucas centenas de microssegundos. A situação se altera razoavelmente quando são transferidos arquivos ao mesmo tempo em que se sincroniza, resultando em variações da ordem de dezenas de milissegundos. Para minimizar o problema, uma possível solução é reservar uma janela de tempo na qual nenhum dispositivo pode transmitir arquivos, ficando reservada para a sincronização. Além disto, se faria necessário um método para compensar o *drift* entre os relógios dos dispositivos, para reduzir a frequência de janelas de tempo necessárias. Com os testes, concluiu-se que o método com maior potencial de ser utilizado para sincronizar vários dispositivos é o PTP. Para refinar a precisão do sincronismo, entretanto, é necessário que a aquisição de *timestamps* seja realizada a nível de *driver* ou de *hardware.* Um possível próximo passo seria implementar as aquisições de *timestamps* no *driver*.

As capturas foram feitas em modo de imagens estáticas. Embora isto aumente a qualidade das imagens, limita a taxa de aquisição a poucos quadros por segundo. Além disto, o método de compressão empregado, o JPEG, não é muito eficiente para vídeos, pois não aproveita as redundâncias temporais das imagens. Para atingir a taxa desejável de trinta quadros por segundo, com uma resolução de 1920x1080, deve-se alterar as configurações de captura para modo vídeo e utilizar compressão h264 pois, de outro modo, as taxas de captura e de transmissão podem não ser satisfatórias.

Uma possível fonte de incerteza no momento de captura das imagens é o tempo entre a ativação do componente de câmera da API OpenMAX e a aquisição efetiva das imagens. Como a API encapsula os métodos com os quais a captura é

realmente efetuada, os detalhes do processo se tornam transparentes ao desenvolvedor. Ademais, a documentação aberta sobre os detalhes de operação da GPU VideoCore IV é escassa. Um estudo mais detalhado é necessário para uma melhor avaliação do impacto da API na variação do instante efetivo de captura. Para vídeos, sugere-se testar o uso de componentes relógio da API para a sincronização.

Outro fator que interfere na captura é o sensor da câmera. O método de obturação eletrônica utilizado no módulo escolhido é o *Rolling Shutter,* no qual a exposição de pontos diferentes é realizada em tempos diferentes, de modo que o último pixel da imagem corresponde a um momento de aquisição diferente do primeiro. Este método é bastante utilizado em câmeras de baixo custo, como as presentes em celulares. Isto cria possíveis distorções na imagem, limitando seu uso em cenas de alta velocidade. Por conta disso, para imagens de melhor qualidade, recomenda-se o uso de câmeras com *Global Shutter.*

Por limitações de prazo, não se implementou métodos de controle de acesso ao meio que permitissem dividir o tempo de transmissão igualmente entre as câmeras. Para aplicações com muitas câmeras, isto provavelmente será necessário, para evitar que haja situações nas quais câmeras recebam muito pouco do tempo de transmissão. Para aproveitar o sincronismo já implementado, sugere-se aplicar um método de TDMA.

Finalmente, o método proposto para verificar a sincronização mostrou-se satisfatório, porém pode-se aprimorá-lo para estimar a diferença de tempo entre as capturas. O funcionamento exitoso do programa integrado possibilita futuros desenvolvimentos no tópico estudado.

Referências

[1] Panasonic creates a panoramic array for ultra wide 3D imagery. Disponível em: http://petapixel.com/2013/01/16/panasonic-creates-a-panoramic-camera-array-for-ultra-wide-3d-imagery/. Acesso em: 10/04/2013.

[2] BOLTON, J. A., Exploring the use of 360 degree curvilinear displays for the presentation of 3d information. MSc Thesis - Queen's University Kingston, Ontario, Canada - January 2013.

[3] Point Grey Research Inc. <http://www.ptgrey.com>. Acesso em: 17/11/2013.

[4] Guimarães, M. P.; Bressan, P. A.; Zuffo, M. K., Frame Lock Synchronization For Multiprojection Immersive Environments based on PC Graphics Clusters. 5th SBC Symposium on Virtual Reality. In: **Proceedings, pp.150-157, Out 2002.**

[5] Realidade virtual. Disponível em:http://www.lsi.usp.br/interativos/nrv/caverna.html. Acesso em: 13/11/2013.

[6] Raffin, B.; Soares, L. PC Clusters for Virtual Reality.In: **Proceedings of the IEEE Virtual Reality Conference**. 2006.

[7] Gnecco, B. B.; Bressan, P. A.; Lopes, R. D.; Zuffo, M. K. DICElib: A Real Time Synchronization Library for Multi-Projection Virtual Reality Distributed Environments. In: **4th SBC Symposium on Virtual Reality**, SRV 2001, p.338-343, October 2001.

[8] **IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems.**

[9] Correll, K.; Barendt, N.; Branicky, M. **Design Considerations for Software Only Implementations of the IEEE 1588 Precision Time Protocol.**

[10] Mock, M. et.al. Clock Synchronization for Wireless Local Area Networks. In: **Proceedings 12th Euromicro Conference on Real-Time Systems**, 2000.

[11] He, Z. Transmission Distortion Analysis for Real-Time Video Encoding and Streaming Over Wireless Networks. In: IEEE **Transactions on Circuits and Systems for Video Technology**, 2006.

[12] Chen, Z. et. al. Share Risk and Energy: Sampling and Communication Strategies for Multi-Câmera Wireless Monitoring Networks.In: **31st Annual IEEE International Conference on Computer Communications**, 2012.

[13] Rowe, A. et.al. FireFly Mosaic: A Vision-Enabled Wireless Sensor Networking System. In: **Real-Time Systems Symposium**, 2007.

[14] http://www.raspberrypi.org/faqs. Acesso em: 15/11/2013.

[15] Raspberry Pi Camera Board (5MP, 1080p, v1.3). Disponível em: https://www.modmypi.com/raspberry-pi-camera-board. Acesso em: 15/11/2013.

[16] ZigBee overview. Disponível em: http://docs.zigbee.org/zigbee-docs/dcn/09-4825.pdf. Acesso em: 08/05/2013.

[17] Redes Wi-Fi: Breve histórico. Disponível em: http://www.teleco.com.br/tutoriais/tutorialwifiiee/pagina_3.asp. Acesso em: 18/11/2013.

[18] OpenMAX™ Integration Layer Application Programming Interface Specification, Version 1.0 ,The Khronos Group Inc, 2005.

[19] Power supply confirmed as 5V micro USB. Disponível em: http://www.raspberrypi.org/archives/260. Acesso em: 18/11/2013.

Apêndice A – Orçamento

Orçamento de unidade sensitiva			
Componente	Quantidade	Preço unitário	Preço total
Raspberry Pi modelo B 512mb + Modulo de câmera 5MP	1	$89,90	$89,90
Cartão de memoria micro SDHC Classe 4	1	R$25,00	R$25,00
Adaptador WiFi USB	1	R$29,99	R$29,99
Cabo Ethernet	1	R$5,00	R$5,00
Fonte de tensão 5V 700mA	1	R$39,00	R$39,00
Cabo micro USB	1	R$9,99	R$9,99
UBEC 3 Ampéres	1	R$34,99	R$34,99
			R$143,97 + $89,90

Apêndice B – Código fonte dos programas implementados

net.h - Contém protótipos de funções relativas a acesso a rede. Referente ao arquivo net.c

net.c - Contém funções relativas a acesso a rede.

time.h - Contém protótipos de funções relativas a manipulacao de valores de tempo. Referente ao arquivo time.c

time.c - Contém funções relativas a manipulacao de valores de tempo.

sync.c - Programa de sincronizacao.

fila.h - Contém protótipos de funções e estruturas relativas ao gerenciamento de filas. Referente ao arquivo fila.c

fila.c - Contém funções e estruturas ao gerenciamento de filas.

master.cpp - Programa mestre do projeto TRANSINC.

```
/*

	Arquivo net.h
	Contém protótipos de funções relativas a acesso a rede.
	Referente ao arquivo net.c

	É parte do projeto TRANSINC de transimissão de videos sincronizados

	Autor: Victor Velloce Ferreira

*/

// Funcao send_file(int sd, char* filename)
// Abre o arquivo dado por filename e o envia pelo socket descrito por sd.
//
// int sd - Socket descriptor referente a coneccao pela qual se deseja enviar o
arquivo
//
// char* filename - nome do arquivo a ser enviado

int send_file(int sd, char* filename);



// Funcao send_buffer (int sd, char* src_buffer, int buffer_size, int chunk_size);
//
// Envia um buffer src_buffer do tamanho buffer_size em pacotes de tamanho chunk_suze
// pelo socket descrito por sd
//
// int sd - Socket descriptor referente a conecçcao pela qual se deseja enviar o buffer
//
// char* src_buffer - buffer a ser enviado
//
// int buffer_size - tamanho do buffer a ser enviado
//
// int chunk_size - tamanho de cada pacote a ser enviad

int send_buffer(int sd, char* src_buffer, int buffer_size, int chunk_size);



// int tcp_connect(char *hostnamep, char *servicesnamep)
//
// Conecta a um servidor, cujo endereco e hostenamep, e na porta servicesnamep
// utilizando o protocolo tcp/ip e retorna o socket descriptor
//
// Retirado das notas de aula de Meios Eletrônicos Interativos do primeiro semestre de
2013,
// da Escola Politecnica da Universidade de Sao Paulo, ministrada pelo prof. Volnys
Borges Bernal
//
// char *hostnamep - nome do host a ser conectado
// char* servicesnamep - nome do servico ou da porta a ser conectada

int tcp_connect(char *hostnamep, char *servicenamep);


// int join_mcast_group(int sd, char* groupaddress)
//
```

```
// Faz com que o socket descrito por sd pertença ao grupo de multicast groupaddress
//
// int sd - Socket descriptor
//
// char* groupaddress - string com o endereco do grupo no formato "X.X.X.X"

int join_mcast_group(int sd, char* groupaddress);
```

```
/*

	Arquivo net.c
	Contém funções relativas a acesso a rede.

	É parte do projeto TRANSINC de transimissão de vídeos sincronizados

	Autor: Victor Velloce Ferreira

*/

// Funcao send_file(int sd, char* filename)
// Abre o arquivo dado por filename e o envia pelo socket descrito por sd.
//
// int sd - Socket descriptor referente a coneccao pela qual se deseja enviar o
arquivo
//
// char* filename - nome do arquivo a ser enviado

int send_file(int sd, char* filename)
{
	char* buffer;			// buffer para os dados que serao enviados
	struct timespec timenow1;  // tempo do sistema antes do envio do arquivo
	struct timespec timenow2;  // tempo do sistema depois do envio do arquivo
	FILE* fp;				// ponteiro para o arquivo a ser lido

	printf("Abrindo arquivo %s\n", filename);

	buffer = (char*) malloc(SEND_BUFFER_SIZE+1);

	fp = fopen(filename, "r");

	if(fp==NULL)
	{
		printf("Erro ao abrir o arquivo.\n");
		return -1;
	}

	fseek(fp, 0, SEEK_END); // vai ao final do arquivo
	int size = ftell(fp);	// le posicao do final do arquivo
	fseek(fp, 0, SEEK_SET); // volta ao inicio do arquivo

	clock_gettime(CLOCK_REALTIME, &timenow1);

	sprintf(buffer, "SIZE=%d:TS=%ld:TN=%ld\n", size, timenow1.tv_sec,
timenow1.tv_nsec);
	write(sd, buffer, strlen(buffer));

	do
	{
		int nitems = fread(buffer, 1, SEND_BUFFER_SIZE, fp);
		write(sd, buffer, nitems);

	} while(!(feof(fp)));

	clock_gettime(CLOCK_REALTIME, &timenow2);

	timenow2.tv_sec = timenow2.tv_sec - timenow1.tv_sec;
	 timenow2.tv_nsec = timenow2.tv_nsec - timenow1.tv_nsec;
```

```
        if (timenow2.tv_nsec < 0){
                timenow2.tv_nsec = 1000000000-timenow2.tv_nsec;
                timenow2.tv_sec = timenow2.tv_sec - 1;}

        double seg = (1.0*timenow2.tv_sec + 1.0*timenow2.tv_nsec/1000000000.0);

        double trate=1.0*size/seg;
        printf("Time to transfer: %lfs\n", seg);
        printf("Average rate: %lfBps\n", trate);
        fclose(fp);

        return 0;
}

int send_buffer(int sd, char* src_buffer, int buffer_size, int chunk_size)
{
        char* buffer;                       // buffer para os dados que serao enviados
        struct timespec timenow1;  // tempo do sistema antes do envio do arquivo
        struct timespec timenow2;  // tempo do sistema depois do envio do arquivo

        if (buffer_size>SEND_BUFFER_SIZE)
        {
                printf("ERRO: tamanho de buffer maior que o limite\n");
                return -1;
        }

        buffer = (char*) malloc(SEND_BUFFER_SIZE+1);

        clock_gettime(CLOCK_REALTIME, &timenow1);

        sprintf(buffer, "SIZE=%d:TS=%ld:TN=%ld\n", buffer_size, timenow1.tv_sec,
timenow1.tv_nsec);
        write(sd, buffer, strlen(buffer));

        int size = buffer_size;
        int n = 0;

        while(size>0)
        {
                if(size > chunk_size)
                        n = chunck_size;
                else
                        n = size;

                memcpy(buffer, src_buffer, n);
                write(sd, buffer, n);

                size -= n;
        }

        clock_gettime(CLOCK_REALTIME, &timenow2);

        timenow2 = timespec_sub(timenow2, timenow1);

        double seg = timespec_toseg(timenow2);

        double trate=1.0*size/seg;
        printf("Time to transfer: %lfs\n", seg);
```

```
        printf("Average rate: %lfKB/s\n", trate/1000);

        return 0;
}

// int tcp_connect(char *hostnamep, char *servicesnamep)
//
// Conecta a um servidor, cujo endereco e hostenamep, e na porta servicesnamep
// utilizando o protocolo tcp/ip e retorna o socket descriptor
//
// Retirado das notas de aula de Meios Eletrônicos Interativos do primeiro semestre de
2013,
// da Escola Politecnica da Universidade de Sao Paulo, ministrada pelo prof. Volnys
Borges Bernal
//
// char *hostnamep - nome do host a ser conectado
// char* servicesnamep - nome do servico ou da porta a ser conectada

int tcp_connect(char *hostnamep, char *servicenamep)
{
       int                 serverport;
    unsigned short int  netbyteordered_serverport;
    unsigned long  int  ipaddr;
    int                 socketdescriptor;
    int                 status;
    char                ipstring[IPMAXSTRSIZE];
    struct sockaddr_in  socketaddress;
    struct servent    * serviceentryp;
    struct hostent    * hostentryp;

    /************************************/
    /* Obtem o numero da porta do servidor */
    /************************************/

    serviceentryp = getservbyname(servicenamep,"tcp");
    if (serviceentryp == NULL)
        {
        // Nao conseguiu converter o nome do servico para numero de porta.
       // Verifica se foi informado numero da porta */
        serverport = atoi(servicenamep);
        if (serverport <= 0)
            {
            printf("Nome do servico (ou porta) invalido. \n");
            return(-1);
            }
        netbyteordered_serverport = htons(serverport);
        }
    else
        {
        serverport = ntohs(serviceentryp->s_port);
        if (serverport <= 0)
            {
            printf("Numero de porta invalida. \n");
            return(-1);
            }
        netbyteordered_serverport = htons(serverport);
        }
    printf("Porta do servidor    = %hu \n",serverport);
```

```
    /*********************************/
    /* Converte endereco IP          */
    /*********************************/

    hostentryp = gethostbyname(hostnamep);
    if (hostentryp == NULL)
        {
       // Nao conseguiu converter nome para endereco IP
        // Verifica se hostname informado é um endereço IP
        status = inet_pton(AF_INET,hostnamep,&socketaddress.sin_addr);
        if (status <=0)
            {
            perror("ERRO: intet_pton()");
            return(-1);
            }
        }
    else
        {
        bcopy(hostentryp->h_addr, (char *)&socketaddress.sin_addr, hostentryp-
>h_length);
        }
     printf("Endereco IP do servidor = %s
\n",inet_ntop(AF_INET,&socketaddress.sin_addr,ipstring,IPMAXSTRSIZE));

    /*****************************************************************/
    /* Socket(): Criacao do socket                                   */
    /*****************************************************************/
    socketdescriptor = socket(PF_INET,SOCK_STREAM,6); // 6="tcp"
    if (socketdescriptor < 0)
        {
        printf("Erro na criação do socket. \n");
        perror("Descricao do erro");
        return (-1);
        }

    /*****************************************************************/
    /* connect(): */
    /*****************************************************************/
    socketaddress.sin_family = AF_INET;  /* address family = internet */
    socketaddress.sin_port   = netbyteordered_serverport;   /* porta */

    status = connect(socketdescriptor,(struct
sockaddr*)&socketaddress,sizeof(socketaddress));
    if (status != 0)
        {
        printf("Erro na chamada Connect().\n");
        perror("Descricao do erro");
        return(-1);
        }
    return(socketdescriptor);
}

int join_mcast_group(int sd, char* groupaddress)
{
       struct ip_mreq mreq;

      mreq.imr_multiaddr.s_addr = inet_addr(groupaddress);
    mreq.imr_interface.s_addr = htonl(INADDR_ANY);
```

```
        if (setsockopt(sd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                    &mreq, sizeof(mreq)) < 0) {
                perror("setsockopt mreq");
                return -1;
        }

}
```

```
/*

	Arquivo time.h
	Contém protótipos de funções relativas a manipulacao de valores de tempo.
	Referente ao arquivo time.c

	É parte do projeto TRANSINC de transmissão de vídeos sincronizados

	Autor: Victor Velloce Ferreira

*/

// void timespec_normalize(struct timespec* ts)
//
// verifica se o valor de nanosegundos em ts eh negativo e modifica
// ts para que nao seja mais negativo
//
// struct timespec* ts - estrutura timespec a ser corrigida

void timespec_normalize(struct timespec* ts);



// struct timespec timespec_add(struct timespec ta, struct timespec tb)
//
// soma os valores de duas estruturas timespec
//
// struct timespec ta - primeiro operando
//
// struct timespec tb - segundo operando

struct timespec timespec_add(struct timespec ta, struct timespec tb);


// struct timespec timespec_sub(struct timespec ta, struct timespec tb);
//
// subtrai os valores da estrutura timespec tb de ta
//
// struct timespec ta - minuendo
//
// struct timespec tb - subtraendo

struct timespec timespec_sub(struct timespec ta, struct timespec tb);


// double timespec_tosec(struct timespec ts)
//
// retorna o valor de uma estrutura timespec ts em segundos
//
// struct timespec ts - estrutura timespec a ser transformada em segundos

double timespec_tosec(struct timespec ts);
```

```
/*

	Arquivo time.c
	Contém funções relativas a manipulacao de valores de tempo.

	É parte do projeto TRANSINC de transimissão de vídeos sincronizados

	Autor: Victor Velloce Ferreira

*/

void timespec_normalize(struct timespec* ts)
{
	if (ts->tv_nsec < 0){
		ts->tv_nsec = 1000000000-ts->tv_nsec;
		ts->tv_sec = ts->tv_sec - 1;}
}

struct timespec timespec_add(struct timespec ta, struct timespec tb)
{
	struct timespec tadd;

	timespec_normalize(&ta);
	timespec_normalize(&tb);

	tadd.tv_nsec = ta.tv_nsec + tb.tv_nsec;
	tadd.tv_sec = ta.tv_sec + tb.tv_sec;

	timespec_normalize(&tadd);

	return tadd;
}

struct timespec timespec_sub(struct timespec ta, struct timespec tb)
{
	struct timespec tsub;

	timespec_normalize(&ta);
	timespec_normalize(&tb);

	tsub.tv_nsec = ta.tv_nsec - tb.tv_nsec;
	tsub.tv_sec = ta.tv_sec - tb.tv_sec;

	timespec_normalize(&tsub);

	return tsub;
}

double timespec_tosec(struct timespec ts)
{
	return (1.0*ts.tv_sec + 1.0*ts.tv_nsec/1000000000.0);
}
```

```
/*

	Arquivo sync.c
	Programa de sincronizacao.

	É parte do projeto TRANSINC de transimissão de vídeos sincronizados

	Autor: Victor Velloce Ferreira

*/

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <bcm2835.h>

#include "net.h"
#include "time.h"

struct sync_status_s
{
	// bool is_clock_master;                                    // Usado para mestre
variavel
	int sd;
	struct sockaddr_in clock_master_addr;
	struct timespec last_ts;
	struct timespec ts[4];
};

int handle_msg(char* msg, int msgsize , struct sync_status_s sync_status)
{
	struct timespec ts;
	struct timespec td;
	struct timespec toff;


	if(strcmp( msg, "TS=", 3)==0)
	{
		sscanf(msg, "TS=%ld:TN=%ld\n", &ts.tv_sec, &ts.tv_nsec);
		adjust_clock(ts);
	}
	else if(strcmp( msg, "TS1=", 4)==0)
	{
		sscanf(msg, "TS1=%ld:TN1=%ld\n", &ts.tv_sec, &ts.tv_nsec);
		sync_status.ts[1] = ts;
		sync_status.ts[2] = sync_status.last_ts;
		clock_gettime(CLOCK_REALTIME, &sync_status.ts[3]);
		sendto(sync_status.sd, "TS1_ACK\n", 9, 0,
		  &sync_status.addr, sizeof(&sync_status.addr));

	}
	else if(strcmp( msg, "TS4=", 4)==0)
```

```
	{
		sscanf(msg, "TS4=%ld:TN4=%ld\n", &ts.tv_sec, &ts.tv_nsec);
		sync_status.ts[4] = ts;
		td = timespec_div(timespec_sub(
			timespec_sub(sync_status.ts[4], sync_status.ts[1])	,
			timespec_sub(sync_status.ts[3], sync_status.ts[2])	),
			2);

		toff = timespec_sub(timespec_add(sync_status.ts[1],
td),sync_status.ts[2]);

		clock_gettime(CLOCK_REALTIME, &ts);

		adjust_clock(timespec_add(ts, toff));

	}
	else
	{
		printf("Comando nao reconhecido\n");
		return -1;
	}

	return 0;
}

int adjust_clock(struct timespec ts_new)
{
	clock_settime(CLOCK_REALTIME, ts_new);
}

int main(int argc, char** argv)
{
	struct sync_status_s sync_status;

	int sd;
	struct sockaddr_in addr;

	sock = socket(AF_INET, SOCK_DGRAM, 0);

	if (sock < 0) {
		perror("socket");
		exit(1);
	}

	bzero((char *)&addr, sizeof(addr));
	addr.sin_family = AF_INET;
	addr.sin_addr.s_addr = htonl(INADDR_ANY);
	addr.sin_port = htons(SYNC_PORT);
	addrlen = sizeof(addr);

	if(join_mcast_group(sd, SYNC_GROUP)!=0)
	{
		printf("Erro: nao foi possivel se unir ao grupo de multicast\n");
		exit(1);
	}

	while(1)
	{
		cnt = recvfrom(sd, message, sizeof(message), 0,
```

```
                        (struct sockaddr *) &addr, &addrlen);

                clock_gettime(CLOCK_REALTIME, &sync_status.last_ts);

                sync_status.clock_master_addr=addr;

                if(cnt>0)
                        handle_msg(char* message, int cnt, struct sync_status_s
sync_status);
                else if(cnt==0)
                        printf("Servidor fechou conexão\n");
                        exit(0);
                else
                        printf("Erro na chamada recvfrom\n");
                        exit(1);
        }
        return 0;
}
```

```
/*

	Arquivo fila.h
	Contém protótipos de funções e estruturas relativas ao gerenciamento de filas.
	Referente ao arquivo fila.c

	É parte do projeto TRANSINC de transimissão de vídeos sincronizados

	Autor: Victor Velloce Ferreira

*/

struct frame_header_s
{
	FILE* fp;			// Ponteiro para arquivo que contem o quadro
	char* filename[100]; // Nome do arquivo que contém o quadro
	char* buffer;		// Ponteiro para o buffer que contem o quadro
	int buff_size;		// Tamanho do buffer que contem o quadro
	struct timespec ts; // Timestamp do quadro
	int framen;			// Numero do frame
	int camera_id;		// Id da camera que gerou o quadro
};

struct fila_s
{
	struct frame_header_s* items;
	int nitems;
	int max_items;
	int fila_inicio;
	int fila_fim;
};

int fila_init(struct fila_s fila);

int fila_insere(struct fila_s fila, struct frame_header_s item);

int fila_nprox(struct fila_s fila);

int fila_tira(struct fila_s fila, struct frame_header_s* item);

int fila_le(struct fila_s fila, struct frame_header_s* item);

int fila_destroy(struct fila_s fila);
```

```
/*

	Arquivo fila.c
	Contém funções e relativas ao gerenciamento de filas.

	É parte do projeto TRANSINC de transimissão de vídeos sincronizados

	Autor: Victor Velloce Ferreira

*/

#include "fila.h"


int fila_init(struct fila_s* fila, int n)
{
	fila->items = (struct frame_header_s*)malloc(sizeof(struct
frame_header_s)*nitems);
	fila->max_items = n;
	fila->nitems = 0;
	fila->fila_inicio=0;
	fila->fila_fim=0;
}

int fila_insere(struct fila_s* fila, struct frame_header_s item)
{
	int ret = 0;

	fila->nitems++;
	if(fila->nitems > fila->max_items)
	{
		fila->nitems = fila->max_items;
		ret = -1;
	}
	fila->items[fila->fila_fim]=item;

	fila->fila_fim++;
	if(fila->fila_fim>=fila->max_items)
		fila->fila_fim=0;

	return ret;
}

int fila_nprox(struct fila_s* fila)
{
	int ret=fila->nitems+1;

	if(ret >= fila->max_items)
		ret = 0;
}

int fila_tira(struct fila_s* fila, struct frame_header_s* item)
{
	if(fila->nitems>0)
	{
		fila->nitems--;
		(*item)=fila->items[fila->fila_inicio];
		fila->fila_inicio++;
```

```
		if(fila->fila_inicio>=fila->max_items)
			fila->fila_inicio=0;

		return 0;
	}

	return -1;
}

int fila_le(struct fila_s* fila, struct frame_header_s* item)
{
	if(fila->nitems>0)
	{
		(*item)=fila->items[fila->fila_inicio];

		return 0;
	}

	return -1;
}

int fila_destroy(struct fila_s* fila)
{
	free(fila->items);
}
```

```
/*

	Arquivo master.cpp
	Programa mestre do projeto TRANSINC.

	É parte do projeto TRANSINC de transimissão de videos sincronizados

	Autor: Victor Velloce Ferreira

*/

#include "stdafx.h"

#undef UNICODE

#define WIN32_LEAN_AND_MEAN

//#include "cv.h"	// include it to used Main OpenCV functions.
//#include "highgui.h" //include it to use GUI functions.
#include <opencv2/core/core.hpp>		// Basic OpenCV structures (cv::Mat, Scalar)
#include <opencv2/highgui/highgui.hpp>

#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>
#include <mswsock.h>
#include <time.h>

#define u_int32 UINT32  // Unix uses u_int32

// Need to link with Ws2_32.lib
#pragma comment (lib, "Ws2_32.lib")
// #pragma comment (lib, "Mswsock.lib")

#define DEFAULT_BUFLEN 15000
#define DEFAULT_PORT "27015"

#define MULTICAST_PORT 6000
#define SEND_FILE_PORT 6001
#define MULTICAST_GROUP "239.0.0.1"
#define MULTICAST_SOURCE "192.168.1.14"
#define MULTICAST_INTERFACE "192.168.1.14"


double PCFreq = 0.0;
__int64 CounterStart = 0;
LARGE_INTEGER li;


// Funcao int join_source_group(int sd, u_int32 grpaddr, u_int32 srcaddr, u_int32
iaddr)
// Retirada do exemplo de multicast da MSDN
//
// Une o socket descrito pro sd ao grupo de multicast cujo endereco e grpaddr.
//
// int sd - Socket descriptor
//
```

```
// u_int32 grpaddr - Endereco do grupo de multicast
//
// u_int32 srcaddr - Endereco da fonte de multicast
//
// u_int32 iaddr - Endereco da interface utilizada

int
join_source_group(int sd, u_int32 grpaddr,
   u_int32 srcaddr, u_int32 iaddr)
{
   struct ip_mreq_source imr;

   imr.imr_multiaddr.s_addr  = grpaddr;
   imr.imr_sourceaddr.s_addr = srcaddr;
   imr.imr_interface.s_addr  = iaddr;
   return setsockopt(sd, IPPROTO_IP, IP_ADD_SOURCE_MEMBERSHIP, (char *) &imr,
sizeof(imr));
}

// Funcao int join_source_group(int sd, u_int32 grpaddr, u_int32 srcaddr, u_int32
iaddr)
// Retirada do exemplo de multicast da MSDN
//
// Desliga o socket descrito pro sd do grupo de multicast cujo endereco e grpaddr.
//
// int sd - Socket descriptor
//
// u_int32 grpaddr - Endereco do grupo de multicast
//
// u_int32 srcaddr - Endereco da fonte de multicast
//
// u_int32 iaddr - Endereco da interface utilizada

int
leave_source_group(int sd, u_int32 grpaddr,
   u_int32 srcaddr, u_int32 iaddr)
{
   struct ip_mreq_source imr;

   imr.imr_multiaddr.s_addr  = grpaddr;
   imr.imr_sourceaddr.s_addr = srcaddr;
   imr.imr_interface.s_addr  = iaddr;
   return setsockopt(sd, IPPROTO_IP, IP_DROP_SOURCE_MEMBERSHIP, (char *) &imr,
sizeof(imr));
}

// Funcao void StartCounter()
//
// Determina a frequencia do relogio para fins de calculo de tempo

void StartCounter()
{
    LARGE_INTEGER li;
    if(!QueryPerformanceFrequency(&li))
       printf( "QueryPerformanceFrequency failed!\n" );

    PCFreq = double(li.QuadPart)/1000000000.0;

    QueryPerformanceCounter(&li);
```

```
    CounterStart = li.QuadPart;
}

// Funcao double GetCounter()
//
// Obtem o valor atual do relogio de alta precisao

double GetCounter()
{
    QueryPerformanceCounter(&li);
    return double((li.QuadPart)-CounterStart)/PCFreq;
}

// Funcao DWORD WINAPI thread_udp_emissor( void * lpParameter)
//
// Thread que emite pacotes a todos os dispositivos a fim de sincroniza-los
//
// void * lpParameter - Ponteiro para o socket descriptor do socket dedicado aos
pacotes UDP

DWORD WINAPI thread_udp_emissor( void * lpParameter)
{
            SOCKET udpSocket = *(SOCKET *) lpParameter;

            struct sockaddr_in addr1;
            struct sockaddr_in addr2;

            struct sockaddr_in sender_addr;

            int addrlen = sizeof(addr1);            // Evita recalcular o tamanho do
endereco a cada vez que sendto() eh chamada

            char message[100];

            int iResult=0;

            double ts, ts_s, ts_n;
            long lts_s, lts_n;
            unsigned long long lts;

            addr1.sin_family = AF_INET;
            addr1.sin_port = htons(MULTICAST_PORT);
//          addr1.sin_addr.s_addr = inet_addr("192.168.0.100");
            addr1.sin_addr.s_addr = inet_addr(MULTICAST_GROUP);


            addr2.sin_family = AF_INET;
            addr2.sin_port = htons(MULTICAST_PORT);
            addr2.sin_addr.s_addr = inet_addr("192.168.0.102");
//          addr2.sin_addr.s_addr = inet_addr(MULTICAST_GROUP);


            printf("Started thread_udp_emissor\n");

            while(1)
            {
                  Sleep(500);
                  ts = GetCounter();
                  lts = ts;
```

```
                        lts_n = lts%1000000000;
                        lts_s = lts/1000000000;

                        sprintf_s(message, "TS1=%ld:TN1=%ld\n", lts_s, lts_n);
//                      sprintf(message, "TS=%ld:TN=%ld\n", lts_s, lts_n);

                        //printf("%s", message);

                        sendto(udpSocket, message, sizeof(message), 0,
                       (struct sockaddr *) &addr1, addrlen);


                }
}

// Funcao DWORD WINAPI thread_udp_emissor( void * lpParameter)
//
// Thread que recebe pacotes dos dispositivos e os responde com seu valor atual do
relogio
//
// void * lpParameter - Ponteiro para o socket descriptor do socket dedicado aos
pacotes UDP

DWORD WINAPI thread_udp_receptor( void * lpParameter)
{
                SOCKET udpSocket = *(SOCKET *) lpParameter;

                struct sockaddr_in addr;
                struct sockaddr_in sender_addr;

                int sender_addr_size = sizeof(sender_addr);
                char message[100];

                int addrlen = sizeof(addr);

                int iResult=0;

                double ts, ts_s, ts_n;
                long lts_s, lts_n;
                unsigned long long lts;

                addr.sin_family = AF_INET;
                addr.sin_addr.s_addr = htonl(INADDR_ANY);
                addr.sin_port = htons(MULTICAST_PORT);

                printf("Started thread_udp_receptor\n");

                while(1)
                {
                        iResult = recvfrom(udpSocket, message, 100, 0, (SOCKADDR *) &
sender_addr, &sender_addr_size);
                        if (iResult == SOCKET_ERROR) {
                                wprintf(L"recvfrom failed with error %d\n",
WSAGetLastError());
                        }

                        //printf("Received: %s", message);

                        if(strncmp(message, "TS1_ACK", 7)==0)
```

```
                    {
                    ts = GetCounter();
                    lts = ts;
                    lts_n = lts%1000000000;
                    lts_s = lts/1000000000;

                          sprintf_s(message, "TS4=%ld:TN4=%ld\n", lts_s, lts_n);
                          //printf("%s", message);
                          sendto(udpSocket, message, sizeof(message), 0,
                                      (struct sockaddr *) &sender_addr,
sender_addr_size);
                    }
                    else
                          ;
             }
}

// Funcao DWORD WINAPI thread_receptor( void * lpParameter)
//
// Thread responsavel por receber os arquivos dos dispositivos
//
// void * lpParameter - Ponteiro para o socket descriptor do socket dedicado aos
pacotes UDP

DWORD WINAPI thread_receptor( void * lpParameter)
{
       SOCKET ClientSocket = *(SOCKET *) lpParameter;
    FILE* fp;

int iResult;
    bool receiving_file = false;
    int size;
       double count;
       char nome_janela[100];

       sprintf_s(nome_janela, "Janela_%d", ClientSocket);

        //cvNamedWindow(nome_janela, CV_WINDOW_NORMAL );
        //cvResizeWindow(nome_janela, 640, 480);

       char filename[100];

    struct addrinfo *result = NULL;
    struct addrinfo hints;

       int i=0;
    int iSendResult;
       int totalsize = 0;
       int writtensize = 0;
    char recvbuf[DEFAULT_BUFLEN];
       char sendbuf[DEFAULT_BUFLEN];
    int recvbuflen = DEFAULT_BUFLEN;

       printf("Accepted client connection and started thread_recpton()");

             do {
                    Sleep(30);
                    iResult = recv(ClientSocket, recvbuf, recvbuflen, 0);
```

```
                if (iResult > 0) {
                    if (receiving_file)
            {
                            size = size-iResult;

                            if(size <= 0)
                            {
                        printf("Reached EOF\n");
                                writtensize = fwrite(recvbuf,1,
size+iResult,fp);
                                totalsize = totalsize+writtensize;
                        printf("Bytes Gravados ateh EOF: %d\n", writtensize);
                               receiving_file=false;
                        size=0;
                        fclose(fp);
                                printf("Carregando imagem %s\n", filename);
                                //plImage* img = cvLoadImage(filename);
                                //cvShowImage(nome_janela, img);
                            }
                            else{
                                    writtensize=fwrite(recvbuf,1, iResult,
fp);
                                    totalsize = totalsize+writtensize;
                            }


                        }
                if(!receiving_file)
                        {
                    recvbuf[iResult]='\0';

                            sscanf_s(recvbuf, "SIZE=%d\n", &size);
                    if(size>0)
                                printf("%d bytes to receive\n", size);

                            if(size>0)
                            {
                                count=GetCounter();
                                i=i+1;
                                sprintf_s(filename, "Frame_%d_%d_%lf.jpg",
ClientSocket, i,count);
                                fopen_s(&fp, filename,"wb");
                                if(fp==NULL)
                        printf("erro ao abrir o arquivo\n");
                                else
                                    receiving_file=true;
                            }
                        }

                }
            else if (iResult == 0)
                printf("Connection closing...\n");
            else  {
                printf("recv failed with error: %d\n", WSAGetLastError());
                closesocket(ClientSocket);
                WSACleanup();
                return 1;
             }
```

```
    } while (iResult > 0);

            //cvDestroyWindow( nome_janela);
}

// Funcao principal. Entrada do programa. Baseada em [20] e [21]
int __cdecl main(void)
{
    WSADATA wsaData;
    int iResult;

    SOCKET ListenSocket = INVALID_SOCKET;       // Socket para aguardar conexoes
    SOCKET ClientSocket = INVALID_SOCKET;       // Socket para redirecionar conexoes
      SOCKET udpSocket = INVALID_SOCKET;              // Socket para troca de
datagramas UDP

    struct addrinfo *result = NULL;
    struct addrinfo hints;
      struct sockaddr_in udp_addr;

    int iSendResult;
    char recvbuf[DEFAULT_BUFLEN];
      char sendbuf[DEFAULT_BUFLEN];
    int recvbuflen = DEFAULT_BUFLEN;

      cvNamedWindow("My_Window");

      // Calcula a frequencia do relogio para calculo de tempo
      StartCounter();

      // Inicializa a Winsock API(WSA)
    iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup erro: %d\n", iResult);
        return 1;
    }

      // Cria o socket para troca de datagramas UDP
      udpSocket = socket(AF_INET, SOCK_DGRAM, 0);
    if (udpSocket == INVALID_SOCKET) {
            printf("Erro ao conectar udpSocket\n");
            return 1;
      }

      // Se une ao grupo multicast
      if(join_source_group(udpSocket, inet_addr(MULTICAST_GROUP),
inet_addr(MULTICAST_SOURCE), inet_addr(MULTICAST_INTERFACE))!=0)
         printf("Erro: %d\n",WSAGetLastError());

      udp_addr.sin_family = AF_INET;
    udp_addr.sin_port = htons(MULTICAST_PORT);
    udp_addr.sin_addr.s_addr = htonl(INADDR_ANY);

      // Se liga a porta
      iResult = bind(udpSocket, (SOCKADDR *) & udp_addr, sizeof (udp_addr));
    if (iResult != 0) {
        wprintf(L"bind failed with error %d\n", WSAGetLastError());
        return 1;
    }
```

```
    ZeroMemory(&hints, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = AI_PASSIVE;

    iResult = getaddrinfo(NULL, DEFAULT_PORT, &hints, &result);
    if ( iResult != 0 ) {
        printf("getaddrinfo failed with error: %d\n", iResult);
        WSACleanup();
        return 1;
    }

       // Cria o socket para aguardar conexoes
    ListenSocket = socket(result->ai_family, result->ai_socktype, result-
>ai_protocol);
    if (ListenSocket == INVALID_SOCKET) {
        printf("socket failed with error: %ld\n", WSAGetLastError());
        freeaddrinfo(result);
        WSACleanup();
        return 1;
    }

       // Liga o socket a porta
    iResult = bind( ListenSocket, result->ai_addr, (int)result->ai_addrlen);
    if (iResult == SOCKET_ERROR) {
        printf("bind failed with error: %d\n", WSAGetLastError());
        freeaddrinfo(result);
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    freeaddrinfo(result);

       cvWaitKey(0);

       // Dispara as thread de comunicacao UDP
       if(CreateThread(NULL, 0, thread_udp_receptor, &udpSocket , 0, 0)==NULL)
       {
                    printf("CreateThread error: %d\n", WSAGetLastError());
                    closesocket(ListenSocket);
                    WSACleanup();
                    return 1;
       }
       if(CreateThread(NULL, 0, thread_udp_emissor, &udpSocket , 0, 0)==NULL)
       {
                    printf("CreateThread error: %d\n", WSAGetLastError());
                    closesocket(ListenSocket);
                    WSACleanup();
                    return 1;
       }

       while(1){
              printf("Aguardando conexao\n");
              iResult = listen(ListenSocket, SOMAXCONN);
              if (iResult == SOCKET_ERROR) {
                    printf("listen failed with error: %d\n", WSAGetLastError());
```

```
                closesocket(ListenSocket);
                WSACleanup();
                return 1;
            }
            printf("Aceitando conexao\n");
            ClientSocket = accept(ListenSocket, NULL, NULL);
            if (ClientSocket == INVALID_SOCKET) {
                printf("accept failed with error: %d\n", WSAGetLastError());
                closesocket(ListenSocket);
                WSACleanup();
                return 1;
            }

            char nome_janela[100];
            printf("Criando thread_receptor\n");

            sprintf_s(nome_janela, "Janela_%d", ClientSocket);

            //cvNamedWindow(nome_janela);
            //cvResizeWindow(nome_janela, 640, 480);

            if(CreateThread(NULL, 0, thread_receptor, &ClientSocket , 0, 0)==NULL)
            {
                printf("CreateThread error: %d\n", WSAGetLastError());
                closesocket(ListenSocket);
                WSACleanup();
                return 1;
            }


        }
    closesocket(ListenSocket);

    iResult = shutdown(ClientSocket, SD_SEND);
    if (iResult == SOCKET_ERROR) {
        printf("shutdown failed with error: %d\n", WSAGetLastError());
        closesocket(ClientSocket);
        WSACleanup();
        return 1;
    }

       if(leave_source_group(udpSocket, inet_addr(MULTICAST_GROUP),
inet_addr(MULTICAST_SOURCE), inet_addr(MULTICAST_INTERFACE))!=0)
          printf("Erro: %d\n",WSAGetLastError());

    closesocket(ClientSocket);
       closesocket(udpSocket);
    WSACleanup();

    return 0;
}
```