

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Pedro Henrique Bufulin de Almeida

**Sistema de vigilância *open source* e
customizável**

Uberlândia, Brasil

2021

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Pedro Henrique Bufulin de Almeida

Sistema de vigilância *open source* e customizável

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Orientador: Pedro Frosi Rosa

Universidade Federal de Uberlândia – UFU

Faculdade de Ciência da Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2021

Pedro Henrique Bufulin de Almeida

Sistema de vigilância *open source* e customizável

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Trabalho aprovado. Uberlândia, Brasil, 01 de novembro de 2016:

Pedro Frosi Rosa
Orientador

Professor

Professor

Uberlândia, Brasil
2021

Resumo

Os sistemas de vigilância carecem de soluções que sejam customizáveis. Apesar de existirem no mercado câmeras que fazem a gravação e o armazenamento de imagem, elas o fazem com baixa resolução e limitações de espaço de memória severas. Mesmo os equipamentos de monitoramento mais sofisticados, como os que possuem reconhecimento de imagem, limitam o usuário de aprimorar essas funções por ter o *software* como *closed source*. Além disso, as soluções providas pela segurança pública ou privada, quando existem, podem usar as imagens coletadas para seus próprios benefícios, o que afeta a privacidade do indivíduo. Este trabalho propõe uma solução que seja fácil de implementar, com ferramentas prontas para uso e completamente customizável por ser *open source*. Dadas essas características, surge o chamado *Self-Sovereign Camera System* (SSCS)

Palavras-chave: código aberto, visão computacional, IP câmera, streaming multimídia.

Abstract

Surveillance systems lack customizable solutions. Despite the existence of cameras on the market that handle both recording and storage of images, they do so at low resolution and with severe memory space limitations. Even the most sophisticated monitoring equipment, such as those with image recognition capabilities, restrict the user from enhancing these features due to the software being closed source. Furthermore, solutions provided by public or private security, when available, may use the collected images for their own benefit, affecting individual privacy. This work proposes a solution that is easy to implement, equipped with ready-to-use tools, and fully customizable as it is open source. Given these characteristics, the so-called Self-Sovereign Camera System (SSCS) emerges.

Keywords: open source, computer vision, IP camera, multimedia streaming.

Lista de ilustrações

Figura 1 – Representação de um arquitetura de tempo real genérica	13
Figura 2 – De <i>grayscale</i> para LBP	15
Figura 3 – Diagrama RTSP	17
Figura 4 – Componente de exemplo	18
Figura 5 – Exemplo de arquitetura de transmissão	32
Figura 6 – Diagrama isolado do gravador	34
Figura 7 – Diagrama isolado do gravador RTSP	34
Figura 8 – Diagrama isolado do indexador	36

Lista de tabelas

Tabela 1 – Tabela comparativa de resoluções e compressão versus tamanho, baseada em dados obtidos do Surveillance Capacity Calculator (DIGITAL, 2023).	27
--	----

Lista de abreviaturas e siglas

SSCS	Self-Sovereign Camera System
CAGR	Compound Annual Growth Rate
UML	Unified Modeling Language
AVC	Advanced Video Coding
HEVC	High Efficiency Video Coding
VMS	Video surveillance Management System
FPS	Frames Per Second
API	Application Programming Interface
DVR	Digital Video Recorder
NVR	Network Video Recorder
REST	Representational State Transfer
HSL	HTTP Live Streaming
RTSP	Real-time Streaming Protocol
SDP	Session Description Protocol
RTP	Real-time Transport Protocol
AU	Access Unit
NALU	Network Abstraction Layer Unit
WebRTC	Web Real-time Communication
SRT	Secure Reliable Transport
RTMP	Real Time Messaging Protocol
LBPH	Local Binary Patterns Histogram

Sumário

1	INTRODUÇÃO	10
1.1	Objetivos	11
1.1.1	Objetivos específicos	11
1.2	Justificativa	12
1.3	Resultados esperados	12
2	FUNDAMENTAÇÃO TEÓRICA	13
2.1	Elementos genéricos de um sistema de multimídia	13
2.1.1	Dados de mídia	14
2.2	Linguagem Go	14
2.3	openCV	15
2.4	Haar Cascade	15
2.5	Arquitetura de microserviços	16
2.6	Arquitetura Monolítica	16
2.7	Protocolo RTSP	16
2.8	Containerização	17
2.9	Diagrama UML de Componentes	18
2.10	Versionamento Semântico (SEMVER)	19
2.11	Daemon	20
2.12	Princípios SOLID	20
2.13	Servidor Proxy	21
2.14	Licenças de Software	21
2.15	Encodings de vídeo e som	22
2.15.1	H.264	22
2.15.2	AV1	23
2.15.3	VP9	23
3	MÉTODO DE DESENVOLVIMENTO	24
3.1	Estado da Arte de Câmeras de Vigilância	24
3.2	Tecnologias e técnicas comuns	25
3.2.1	métodos de transmissão	25
3.2.2	algoritmos de reconhecimento de imagem	26
3.2.3	armazenamento, resolução e compressão	26
3.2.4	vulnerabilidades de segurança	27
3.2.5	custos de operação	28

4	REVISÃO BIBLIOGRÁFICA	29
5	DESENVOLVIMENTO	30
5.1	Ambiente de desenvolvimento	30
5.2	Gravador	33
5.2.1	Implementação de gravador com RTSP e compressão H.264	34
5.3	Indexador	35
6	CONCLUSÃO	37
	REFERÊNCIAS	38
	APÊNDICES	40
	APÊNDICE A – QUISQUE LIBERO JUSTO	41
	APÊNDICE B – COISAS QUE FIZ E QUE ACHEI INTERESSANTE MAS NÃO TANTO PARA ENTRAR NO CORPO DO TEXTO	42
	ANEXOS	43
	ANEXO A – EU SEMPRE QUIS APRENDER LATIM	44
	ANEXO B – COISAS QUE EU NÃO FIZ MAS QUE ACHEI INTE- RESSANTE O SUFICIENTE PARA COLOCAR AQUI	45
	ANEXO C – FUSCE FACILISIS LACINIA DUI	46

1 Introdução

Em 2019 foi estimado que existem 200 milhões de câmeras de vigilância na China e na última década, avanços tecnológicos tornaram essas câmeras ainda mais eficientes em monitorar 1.4 bilhões de chineses. Ainda segundo o autor, o reconhecimento de rostos por câmeras começou a ser uma realidade em 2010 quando pesquisadores descobriram algoritmos de deep learning usados para reconhecer imagens e voz. Esses algoritmos podem também inferir em tempo real a quantidade e a densidade de pessoas numa dada imagem (QIANG, 2019).

Esta intensificação do uso da tecnologia de vigilância não se limita à China; ou está se tornando ou já é a realidade em muitos outros países. De acordo com um relatório da Market Research Future (GUPTA, 2018), a América do Norte é atualmente líder em termos de participação de mercado de câmeras de CCTV, detendo 28,5% do mercado global, seguida pela Europa com 18,3%. Este domínio é reflexo de um mercado de vigilância por vídeo que tem mostrado crescimento substancial: avaliado em 45,9 bilhões de dólares em 2021, este mercado tem uma projeção de crescimento para 110,2 bilhões de dólares até 2030, com uma taxa composta anual de crescimento (CAGR) de 11,6% para o período de 2022 a 2030. Esses dados evidenciam não apenas a escala atual da indústria de vigilância, mas também sua trajetória ascendente, realçando a importância de abordar questões de privacidade e segurança em um contexto tão amplo e em rápido crescimento.

À medida que o mercado de vigilância por vídeo se expande, os sistemas associados também se tornam progressivamente mais complexos e de maior alcance. Isso demonstra a necessidade de infraestruturas mais robustas e eficientes. Nesse contexto, é intrigante observar que as complicações técnicas enfrentadas pelos sistemas de vigilância têm muitas semelhanças com aquelas encontradas em plataformas de streaming de vídeo. Ambos os tipos de sistemas têm uma dependência substancial de infraestruturas distribuídas para assegurar a prestação de serviços em tempo real, de maneira eficaz e confiável.

Entretanto, a privacidade dos indivíduos não é uma preocupação primária dos principais provedores desse tipo de serviço. Nesse contexto, tecnologias de código aberto podem oferecer uma solução, pois permitem que qualquer pessoa verifique se o software tratando a segurança e a privacidade de maneira adequada (MARDJAN; JAHAN, 2016).

Considerando-se a necessidade de criar uma solução que traga os benefícios da vigilância com as funcionalidades modernas de reconhecimento de imagem, armazenamento e mantendo a privacidade individual, surge neste trabalho a proposta da construção de um software de vigilância.

1.1 Objetivos

O propósito deste trabalho é projetar um sistema de monitoramento integralmente sustentado por software livre, com capacidade de integração com uma ampla variedade de dispositivos de hardware. Este sistema oferecerá visualização em tempo real de imagens de câmeras, incorporando algoritmos de visão computacional para análise destas. Será possível armazenar e buscar vídeos, utilizando critérios identificados pelos algoritmos. Uma das grandes virtudes do projeto é sua adaptabilidade: ele é projetado para permitir a inclusão de novas funcionalidades, como notificações de eventos, e a modificação de características existentes. Por exemplo, o armazenamento pode ser configurado para ser local ou em nuvem, e o processamento para reconhecimento de imagens pode ocorrer tanto em transmissões ao vivo quanto em vídeos arquivados. Para facilitar a implementação e personalização do sistema, será disponibilizada uma documentação abrangente, delineando várias opções de uso para satisfazer as distintas necessidades dos usuários.

1.1.1 Objetivos específicos

Os objetivos específicos envolvem:

- Desenvolver todo o software com código aberto e mantê-lo publicamente disponível no Github.
- Usar a containerização como estratégia para conseguir portabilidade entre vários hardwares e sistemas operacionais diferentes.
- Implementar uma funcionalidade de visualização em tempo real de imagens provenientes de câmeras.
- Incorporar algoritmos de visão computacional para análise e processamento das imagens em tempo real.
- Projetar e desenvolver um módulo de armazenamento flexível, permitindo opções tanto locais quanto em nuvem.
- Habilitar a busca de vídeos com critérios baseados nas identificações realizadas pelos algoritmos de visão.
- Possibilitar a customização do sistema, incluindo a adição de novas funcionalidades como notificações de eventos.
- Adaptar o sistema para permitir variações nos métodos de reconhecimento de imagens, seja em transmissões ao vivo ou em vídeos arquivados.

- Escrever testes automatizados e *benchmarks* para manter a qualidade do software e das soluções em um nível aceitável.
- Produzir uma documentação detalhada e abrangente, facilitando a implementação, personalização e manutenção do sistema por parte dos usuários.

1.2 Justificativa

Ainda que existam várias soluções disponíveis em hardware, elas não se integram facilmente umas com as outras. Por vezes usam protocolos e formatos de arquivo diferentes, excluindo alguns softwares de se integrarem com o hardware. Outras vezes é o contrário: o software não se adapta ao hardware, e mesmo se existe um que seja possível integrar, ele não tem as funcionalidades desejadas. Portanto, faz sentido um software customizável que possa se adaptar as necessidades do usuário e que crie uma fundação para criar novas funções de vigilância.

1.3 Resultados esperados

(não sei se é necessária essa parte, já que temos todos os objetivos específicos postulados logo acima)

2 Fundamentação Teórica

Neste capítulo, são apresentadas as tecnologias e os conceitos que serão empregados ao longo da execução deste projeto. Esses conceitos estão relacionados à arquitetura do sistema que será desenvolvido, aos protocolos, técnicas de reconhecimento de imagem, compressão, técnicas de software e demais elementos necessários para a materialização da solução proposta.

2.1 Elementos genéricos de um sistema de multimídia

A figura abaixo mostra um sistema de cliente-servidor para transmissão de dados de multimídia. Na origem, tem-se a gravação comprimida, codificada e armazenada no dispositivo de armazenamento, como um disco rígido, por exemplo. Em seguida, por meio de algum software de mídia, esses arquivos são requisitados por usuários e entregues de acordo. Uma Network Interface Card (NIC na figura) então usa um protocolo de transmissão via rede para entregar os dados de multimídia para o cliente (tais como RTMP, HLS, RTSP, WebRTC), onde eles são primeiramente armazenados na memória principal ou disco e eventualmente decodificados e apresentados ao usuário.

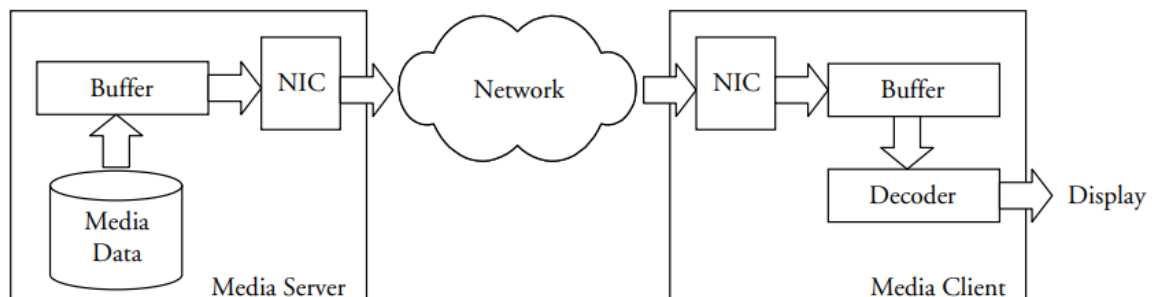


Figura 1 – Arquitetura genérica de transmissão em tempo real

Uma característica única deste modelo de sistema é que os componentes do sistema envolvidos trabalham em conjunto no processo de entrega de dados. Assim, um problema em qualquer um dos componentes do sistema pode degradar o desempenho de toda a cadeia.

2.1.1 Dados de mídia

O 'multi' em multimídia refere-se a múltiplos meios de comunicação do mesmo tipo ou de tipos diferentes que são criados, entregues e apresentados juntos. Existem muitos tipos diferentes de mídia, desde o texto simples mais básico, até texto formatado, gráficos, imagens, áudio, vídeo ou até mesmo informações táteis. Podemos classificar amplamente esses diversos tipos de mídia em duas categorias principais, especialmente no contexto da entrega de dados multimídia: mídias discreta ou contínuas.

As discretas referem-se às que não possuem nenhum requerimento explícito de tempo de apresentação. Por exemplo, buscar na rede para exibi-la no navegador. Dependendo da largura de banda disponível, o navegador pode levar um tempo largamente variável até receber a imagem antes que ela seja decodificada e em seguida apresentada. Obviamente, é preferível que esse processo seja o mais breve possível, mas desde que os dados da imagem sejam recebidos corretamente, renderizados e apresentados, a requisição é considerada bem sucedida. Em outras palavras, não há restrição inerente nos dados de mídia que exigem a apresentação em um prazo de tempo ou limite de atraso. É por isso que o tráfego de rede criado pela entrega de mídia discreta é chamado tráfego elástico. Esse nome refere-se à possibilidade de tolerar variações em tempo de atraso.

As contínuas, em contrapartida, possuem um prazo de tempo para a apresentação embarcados nos seus dados de mídia. Por exemplo, dados de vídeo estão geralmente codificados em quadros que precisam ser apresentados sequencialmente em uma frequência específica como 25 FPS. Portanto, para apresentar o vídeo corretamente é necessário não só receber os dados corretamente como também sequenciar de acordo com os tempos especificados. Falhas no sequenciamento podem causar degradações perceptíveis na qualidade do vídeo (movimentação lenta ou fatiada) mesmo se a transmissão foi bem sucedida. Nesse sentido, tráfego de rede proveniente de mídia contínua também é conhecido como tráfego não-elástico, pois precisa manter o tempo íntegro. (LEE, 2005)

2.2 Linguagem Go

O Go foi concebido em setembro de 2007 por Robert Griesemer, Rob Pike e Ken Thompson, todos do Google, e foi anunciado em novembro de 2009. Os objetivos da linguagem e suas ferramentas acompanhantes eram ser expressivos, eficientes tanto na compilação quanto na execução e eficazes na escrita de programas confiáveis e robustos.

A linguagem tem uma semelhança superficial com C e, como C, é uma ferramenta para programadores profissionais, alcançando o máximo efeito com meios mínimos. Mas é muito mais do que uma versão atualizada de C. Ele empresta e adapta boas ideias de muitas outras linguagens, enquanto evita características que levaram à complexidade e código não confiável. Suas facilidades para concorrência são novas e eficientes, e sua

abordagem para abstração de dados e programação orientada a objetos é incomumente flexível. Ele possui gerenciamento automático de memória ou coleta de lixo.

Go é especialmente adequado para construir infraestrutura, como servidores em rede, e ferramentas e sistemas para programadores, mas é verdadeiramente uma linguagem de propósito geral e encontra uso em domínios tão diversos quanto gráficos, aplicativos móveis e aprendizado de máquina. Tornou-se popular como um substituto para linguagens de script não tipadas porque equilibra expressividade com segurança: programas Go geralmente são executados mais rápido do que programas escritos em linguagens dinâmicas e sofrem muito menos falhas devido a erros de tipo inesperados.

Go é um projeto de código aberto, portanto, o código-fonte de seu compilador, bibliotecas e ferramentas está disponível gratuitamente para qualquer pessoa. Contribuições para o projeto vêm de uma comunidade mundial ativa. Go funciona em sistemas semelhantes ao Unix - Linux, FreeBSD, OpenBSD, Mac OS X - e em Plan9 e Microsoft Windows. Programas escritos em um desses ambientes geralmente funcionam sem modificações nos outros. (DONOVAN; KERNIGHAN, 2015)

2.3 openCV

(escrever sobre o openCV)

2.4 Haar Cascade

(escrever sobre o funcionamento da Haar Cascade)

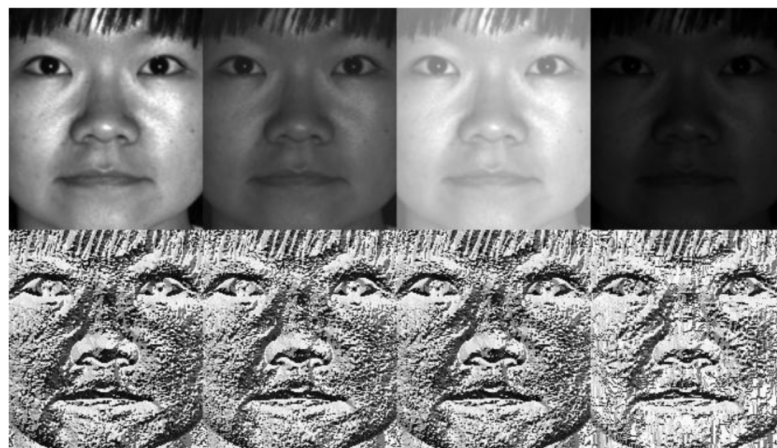


Figura 2 – Imagens resultantes do algoritmo LBP abaixo, geradas à partir da imagem em *grayscale* acima.

2.5 Arquitetura de microsserviços

Explicar aqui o que são microsserviços e como eles funcionam. Fazer talvez uma comparação em relação à uma monolito

2.6 Arquitetura Monolítica

Explicar aqui como funciona uma arquitetura monolítica e quais as desvantagens e vantagens em relação ao microsserviço.

2.7 Protocolo RTSP

O Real-time Streaming Protocol (RTSP) é um protocolo da camada de aplicação projetado para controlar a entrega de dados de mídia (por exemplo, reproduzir, pausar e buscar) com informações de tempo embutidas, como áudio e vídeo. O protocolo é independente do protocolo de camada inferior. Assim, o RTSP pode ser transmitido por meio de TCP, UDP ou outros protocolos de transporte. A sintaxe do RTSP compartilha muitas semelhanças com o HTTP/1.1, simplificando assim a implementação e a implantação. No entanto, além das semelhanças de sintaxe, o RTSP difere do HTTP em muitos aspectos importantes.

Primeiro, ao contrário do HTTP, o RTSP é um protocolo com estado, exigindo que o host mantenha informações de estado de uma sessão de streaming em várias requisições RTSP. Segundo, tanto o servidor quanto o cliente RTSP podem emitir requisições RTSP. Finalmente, os dados de mídia devem ser entregues fora do canal principal, ou seja, usando um protocolo separado, como, mas não limitado a, o Protocolo de Transporte em Tempo Real (RTP).

Em uma aplicação de streaming típica (veja a figura abaixo), o cliente primeiro obterá um arquivo de descrição de apresentação usando métodos fora do canal principal (por exemplo, através da web usando HTTP). O arquivo de descrição de apresentação descreve uma ou mais apresentações, cada uma composta por uma ou mais transmissões de mídia sincronizadas. O arquivo de descrição de apresentação também contém propriedades das transmissões de mídia, como o formato de codificação, para permitir que o cliente selecione e se prepare para a reprodução da mídia. Cada transmissão de mídia controlável é identificada por uma URL RTSP separada, que é semelhante à URL HTTP no sentido de que identifica o servidor que hospeda a transmissão de mídia e o caminho lógico que identifica a transmissão de mídia. Observe que as transmissões de mídia em uma apresentação podem vir de vários servidores e cada transmissão é controlada por meio de uma sessão RTSP separada. (RAO; LANPHIER; SCHULZRINNE, 1998)

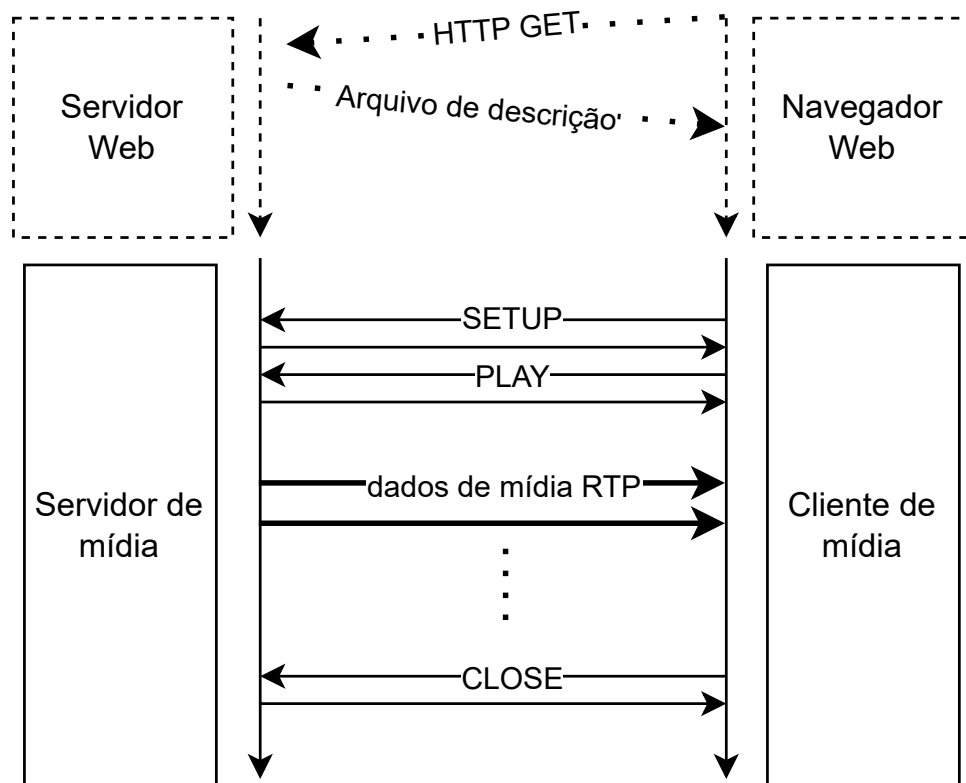


Figura 3 – Transmissão de mídia usando o RTSP, início ao fim.

2.8 Containerização

A containerização emergiu como uma solução inovadora para os desafios associados ao desenvolvimento, implantação e gestão de aplicações de maneira uniforme em diversos ambientes. Diferentemente das máquinas virtuais, que necessitam de um sistema operacional dedicado por instância, os contêineres funcionam de forma isolada, aproveitando o sistema operacional do host, o que impulsiona uma significativa eficiência e minimiza o consumo excessivo de recursos (BURNS, 2018a).

Em essência, o propósito de um contêiner é criar limites para recursos específicos (como uma aplicação que requer dois núcleos e 8 GB de memória), definir a propriedade dentro das equipes (identificando qual equipe é responsável por qual imagem) e promover a separação de preocupações (estipulando que cada imagem cumpre uma função específica). Estes fatores motivam a segmentação de uma aplicação em múltiplos contêineres dentro de uma única máquina (BURNS, 2018b).

Com o advento da tecnologia de contêineres, o Docker se destacou como uma ferramenta primordial desde sua criação como projeto open-source no início de 2013 (MERKEL, 2014). Ele permite que os desenvolvedores encapsulem aplicações e suas respectivas dependências em contêineres autossuficientes, aptos a operar em qualquer servidor, in-

dependentemente das configurações do ambiente host. Esta abordagem tem implicações substanciais para o desenvolvimento ágil, integrando-se perfeitamente com as práticas de Integração Contínua e Entrega Contínua (CI/CD). Em suma, a containerização facilita a replicação, portabilidade e escalabilidade, mitigando as inconsistências entre os ambientes de desenvolvimento, teste e produção, o que resulta em um ciclo de vida do desenvolvimento de software mais integrado e gerenciável.

2.9 Diagrama UML de Componentes

O diagrama de componentes é uma ferramenta da UML (Unified Modeling Language). Ele tem como objetivo visualizar a organização e as relações entre diferentes componentes em um sistema de software. Enquanto outros diagramas focam na funcionalidade do sistema, o diagrama de componentes se concentra no aspecto estrutural.

Um diagrama de componentes representa partes modulares de um sistema, chamadas de componentes. Estes componentes podem ser físicos (como bibliotecas, módulos, arquivos) ou conceituais e estão conectados através de interfaces. Estas interfaces são os pontos de interação entre os componentes.

Características dos diagramas de componentes:

- Mostram as dependências entre os componentes.
- Identificam as interfaces que conectam os componentes.
- Auxiliam na modularização e reutilização de software.
- Oferecem uma visão da arquitetura do sistema.

Ao trabalhar com diagramas de componentes, as equipes podem compreender melhor a estrutura do sistema e identificar áreas para atenção ou desenvolvimento.

Abaixo, um exemplo de um componente e suas partes:



Figura 4 – Componente, provendo uma interface e com requerimento de outra

O componente é representado pela estrutura retangular. O ícone no topo direito do retângulo é opcional, sendo uma especificação do UML 1.4. Na Figura, a bolinha à

esquerda com a legenda `PublishStream` indica uma interface provida, e o semi círculo indica uma interface requerida.

Uma interface fornecida é aquela que é realizada diretamente pelo próprio componente, ou realizada por um dos classificadores que implementam o componente, ou ainda fornecida por uma porta pública do componente. Na notação de pirulito, temos a interface fornecida do componente. No exemplo, o componente de `MediaServer` fornece (implementa) a interface de `PublishStream`.

Uma interface necessária é aquela designada por uma dependência de uso do próprio componente, ou designada por uma dependência de uso de um dos classificadores que implementam o componente, ou ainda é exigida por uma porta pública do componente. No exemplo, o componente `MediaServer` requer a interface `ReadStream`.

2.10 Versionamento Semântico (SEMVER)

O *SEMVER*, abreviação de Versionamento Semântico, é uma convenção de nomenclatura usada para gerenciar e comunicar eficazmente as mudanças em versões de software. Ele cria uma maneira compreensível de representar as mudanças em software através do número de sua versão.

A estrutura básica de um número de versão SEMVER é *MAJOR.MINOR.PATCH*, onde:

- *MAJOR* é incrementado para mudanças incompatíveis que requerem que o usuário faça algo diferente.
- *MINOR* é incrementado quando são feitas adições em um modo compatível, ou seja, quando funcionalidades são adicionadas sem alterar o comportamento das funcionalidades existentes.
- *PATCH* é incrementado após correções de bugs que são compatíveis e não afetam o funcionamento global do software de maneira significativa.

Além destes três números principais, o SEMVER também permite a adição de etiquetas pré-lançamento e de build, que podem ser usadas para representar versões alfa, beta, candidatas a lançamento, entre outros.

O principal benefício de adotar o *SEMVER* é a clareza. Ao olhar para o número de versão de um software, os desenvolvedores podem rapidamente entender a natureza e a magnitude das mudanças, facilitando decisões sobre a atualização ou integração de dependências. (PRESTON-WERNER, 2013)

2.11 Daemon

Um daemon é um processo em segundo plano que normalmente é executado indefinidamente. O sistema operacional UNIX depende de muitos processos daemon para realizar tarefas rotineiras (e não tão rotineiras). No ambiente operacional Solaris, por exemplo, o daemon `pageout` gerencia a paginação para o gerenciamento de memória. O `in.rlogind` lida com solicitações de login remoto. Outros daemons lidam com correio, transferência de arquivos, estatísticas e solicitações de impressora, para citar alguns. (KAY, 2003)

2.12 Princípios SOLID

Os princípios SOLID, nos dizem como organizar nossas funções e estruturas de dados em classes e como essas classes devem estar interconectadas. O uso da palavra "classe" não implica que esses princípios se aplicam apenas ao software orientado a objetos. Uma classe é simplesmente um agrupamento acoplado de funções e dados. Todo sistema de software tem esses agrupamentos, quer sejam chamados de classes ou não. Os princípios SOLID se aplicam a esses agrupamentos. O objetivo dos princípios é a criação de estruturas de software de nível médio que são fáceis de entender, formam a base de componentes que podem ser usados em muitos sistemas software.

A história dos princípios SOLID é longa. Comecei a montá-los no final dos anos 1980, enquanto debatia princípios de design de software com outras pessoas no USENET (um tipo primitivo de Facebook). Ao longo dos anos, os princípios mudaram e evoluíram. Alguns foram excluídos. Outros foram mesclados. Ainda outros foram adicionados. O agrupamento final estabilizou-se no início dos anos 2000, embora eu os tenha apresentado em uma ordem diferente.

- **SRP: Princípio da Responsabilidade Única**

Um corolário ativo da lei de Conway: A melhor estrutura para um sistema de software é fortemente influenciada pela estrutura social da organização que o utiliza, de forma que cada módulo de software tenha uma, e apenas uma, razão para mudar.

- **OCP: Princípio Aberto-Fechado**

Bertrand Meyer tornou este princípio famoso na década de 1980. Em essência, para que os sistemas de software sejam fáceis de alterar, eles devem ser projetados para permitir que o comportamento desses sistemas seja alterado pela adição de novo código, em vez de alterar o código existente.

- **LSP: Princípio da Substituição de Liskov**

A famosa definição de subtipos de Barbara Liskov, de 1988. Em resumo, este prin-

cípio diz que para construir sistemas de software a partir de peças intercambiáveis, essas peças devem aderir a um contrato que permita que sejam substituídas umas pelas outras.

- **ISP: Princípio da Segregação da Interface**

Este princípio aconselha os designers de software a evitar depender de coisas que eles não usam.

- **DIP: Princípio da Inversão de Dependência**

O código que implementa políticas de alto nível não deve depender do código que implementa detalhes de baixo nível. Pelo contrário, os detalhes devem depender das políticas.

Esses princípios, se aplicados em conjunto e habilmente, tornam o código mais limpo, legível, organizado e de fácil manutenção. ([MARTIN, 2018a](#))

2.13 Servidor Proxy

Explicar o porquê de às vezes usar um proxy. Às vezes queremos dar um feed de imagem, mas não tem como fazer uma conexão direta ao sistema que está provendo diretamente a imagem com RTSP. Tanto porque o sistema não consegue lidar com muitas requisições ao mesmo tempo, ou porque ele está numa intranet que precisa de um acesso seguro para prover ao resto da rede.

2.14 Licenças de Software

Ao desenvolver software, a escolha de uma licença apropriada é crucial, pois ela dita como o software pode ser distribuído, utilizado, modificado e compartilhado. Licenças de software são instrumentos legais que protegem tanto o criador quanto os usuários finais, estabelecendo regras claras sobre o que pode e não pode ser feito com o software.

A escolha da licença impacta diretamente a maneira como um sistema é construído e mantido. Por exemplo, licenças permissivas como a MIT ([INITIATIVE, 2023](#)) ou a Licença Apache ([FOUNDATION, 2004](#)) permitem que o código seja utilizado em software proprietário, enquanto licenças copyleft, como a GNU GPL (General Public License) ([FOUNDATION, 2007](#)), exigem que modificações ou trabalhos derivados também sejam de código aberto, mantendo o software livre e protegendo futuras modificações e redistribuições sob os mesmos termos.

No entanto, licenças copyleft são por vezes rotuladas como "licenças virais", um termo pejorativo que surgiu em meados de 1990, um ano depois do lançamento da GPLv1.

Devido à maneira como suas obrigações legais podem se estender aos projetos derivados. Essas licenças estipulam condições específicas, sob as quais os direitos de uso, cópia, modificação e distribuição são concedidos. O não cumprimento dessas condições pode revogar esses direitos, resultando em implicações legais para os usuários. Isso tem o potencial de criar um efeito cascata, onde as obrigações da licença se estendem não apenas ao projeto original, mas a todos os projetos derivados, exigindo que mantenham as mesmas liberdades para os usuários finais, sob pena de terem que interromper o uso ou distribuição do software. ([GENERAL...](#),)

2.15 Encodings de vídeo e som

Encodings de vídeo, também conhecidos simplesmente como codecs de vídeo, são algoritmos e padrões que comprimem e codificam dados de vídeo, removendo redundâncias e detalhes visuais não perceptíveis, para reduzir o tamanho dos arquivos e permitir sua transmissão, armazenamento e reprodução eficientes. Eles desempenham um papel fundamental na compressão de vídeo e na manutenção da qualidade do conteúdo visual.

Existem vários padrões de codecs de vídeo, como H.264, H.265 (HEVC), VP9, AV1, entre outros. A escolha do codec pode afetar a compatibilidade entre dispositivos e aplicativos, bem como a eficiência da compressão. No caso, dada a natureza desse projeto, veremos aqui apenas aqueles que fazem sentido em um ambiente de transmissão em tempo real.

2.15.1 H.264

O H.264, também conhecido como MPEG-4 Part 10 ou Advanced Video Coding (AVC), é um dos padrões de compressão de vídeo mais amplamente utilizados e eficientes em termos de taxa de bits disponíveis atualmente. Foi desenvolvido pelo ITU-T Video Coding Experts Group (VCEG) e pelo Moving Picture Experts Group (MPEG).

Um dos pontos fundamentais do seu funcionamento, é a divisão de quadros. Isto é, O vídeo é dividido em quadros individuais, que são imagens estáticas em uma sequência de vídeo. Existem três tipos de quadros no H.264:

- Quadros I (Intra): São quadros-chave que não dependem de nenhum quadro anterior. Eles são codificados de forma independente e contêm informações completas da imagem.
- Quadros P (Predictive): São quadros que dependem do quadro I anterior e/ou de outros quadros P. Eles contêm apenas as diferenças entre o quadro atual e os quadros de referência.

- Quadros B (Bi-predictive): Dependem tanto dos quadros anteriores quanto dos futuros. Eles também contêm apenas as diferenças em relação aos quadros de referência.

Para manter a qualidade da imagem, o H.264 opera em um "ciclo de quadros", onde os quadros de referência são periodicamente atualizados. Isso assegura que quadros-chave (I-frames) sejam inseridos regularmente para preservar a qualidade visual durante a transmissão de vídeo. Também utilizam-se várias técnicas avançadas de compressão de vídeo para eficientemente codificar quadros P e B. Uma dessas técnicas é a "predição de movimento", que identifica padrões de movimento nos quadros de referência e registra apenas as diferenças entre o quadro atual e as previsões de movimento dos quadros de referência. Isso economiza espaço de armazenamento e largura de banda, eliminando informações redundantes.

Outra técnica crucial é a "transformada de blocos", onde os blocos de pixels em um quadro são convertidos em coeficientes de frequência por meio da Discrete Cosine Transform (DCT). Essa conversão permite a compactação de informações, e os coeficientes de alta frequência, que contêm menos detalhes visíveis, podem ser quantizados ou eliminados com perda mínima de qualidade. (WANG; KWONG; KOK, 2006)

Em resumo, esse codec é amplamente utilizado em aplicações em tempo real, como videoconferência, streaming ao vivo e videochamadas, devido à sua eficiência de compressão e baixa latência. Ele oferece uma boa qualidade de vídeo com taxas de bits relativamente baixas e é compatível com uma ampla variedade de dispositivos e aplicativos.

2.15.2 AV1

2.15.3 VP9

3 Método de Desenvolvimento

A metodologia selecionada para o avanço deste projeto se baseia, inicialmente, em um exame das soluções de hardware e software atualmente presentes no mercado. O objetivo é compreender suas características, funcionalidades e deficiências. Esse estudo, de caráter essencialmente qualitativo, procura identificar tendências, padrões e lacunas nas soluções existentes.

Em seguida, serão exploradas as tecnologias e técnicas comumente adotadas na indústria. Essa investigação abrange desde protocolos de transporte até algoritmos sofisticados de compressão e reconhecimento, assim como casos de vulnerabilidades de segurança. O intuito é discernir os padrões de uso e as preferências do setor.

Dentro desse contexto, uma análise focada no SSCS será realizada. Investigaremos se ele propõe uma reinvenção dessas soluções padrão ou se opta por uma integração harmônica com elas. Esta etapa é crucial para entender a proposta de valor e o diferencial que o sistema proposto oferece em relação às alternativas tradicionais.

Posteriormente, adentraremos na esfera econômica, conduzindo uma pesquisa sobre os custos envolvidos. Essa avaliação não se restringirá apenas aos valores de aquisição de software ou hardware, mas também contemplará os gastos operacionais, incluindo a execução de softwares e os custos de aquisição, manutenção e atualização de hardwares.

3.1 Estado da Arte de Câmeras de Vigilância

Diversas soluções de vigilância estão atualmente disponíveis no mercado, cada uma adotando um conjunto único de padrões e tecnologias. Entre as várias opções, as câmeras CCTV (*Closed-circuit television*) emergem como um sistema tradicional e amplamente empregado. Curiosamente, o conceito de CCTV remonta a 1927, quando o físico russo Léon Theremin desenvolveu o primeiro dispositivo desse tipo, dando origem à primeira geração de sistemas de vigilância (GLINSKY, 2000). Esse sistema consiste em um número de câmeras localizadas em múltiplas posições remotas e conectadas a um conjunto de monitores, geralmente em uma única sala de controle, através de switches (uma matriz de vídeo). Ao contrário de soluções mais modernas que utilizam transmissão via IP, as câmeras CCTV são frequentemente associadas a sistemas de transmissão analógica, embora modelos mais recentes possam incorporar capacidades digitais e transmissão pela web. Atualmente, os modelos mais comuns de câmeras de vigilância para uso comercial já possuem integração direta à internet usando o protocolo IP.

O avanço tecnológico nas câmeras de vigilância levou ao desenvolvimento de sis-

temas semi-automáticos, conhecidos como sistemas de vigilância de segunda geração. A maior parte da pesquisa nesses sistemas é baseada na criação de algoritmos para detecção automática em tempo real de eventos, auxiliando o usuário a reconhecer os eventos.

Seguindo essa mesma tendência tecnológica, chegamos então à terceira geração, que se refere aos sistemas de vigilância concebidos para lidar com um grande número de câmeras, uma dispersão geográfica de recursos, muitos pontos de monitoramento, e para espelhar a natureza hierárquica e distribuída do processo humano de vigilância. Os principais objetivos que se espera de uma aplicação genérica de vigilância visual de terceira geração, baseada nas exigências do usuário final, são fornecer uma boa compreensão da cena, orientada para atrair a atenção do operador humano em tempo real, possivelmente em um ambiente multi-sensorial, informações de vigilância e usando componentes padrões de baixo custo.

3.2 Tecnologias e técnicas comuns

3.2.1 métodos de transmissão

No quesito dos métodos de transmissão multimídia por essas câmeras, temos três métodos principais: *streaming* tradicional, *download* progressivo, e *streaming* adaptativo. Cada um deles tem suas vantagens e desvantagens quanto aos critérios de latência, qualidade, requerimentos de processamento e compatibilidade com outros tipos de software. O *Streaming* tradicional requer um protocolo *stateful* que estabelece uma sessão entre o servidor e o cliente. Nesse último método, a mídia é transmitida como uma corrente constante de pacotes por UDP ou TCP. Como exemplo o Real Time Streaming Protocol (RTSP), baseado no Real-time Transport Protocol (RTP), que é o mais comum para a transmissão de vídeo nas câmeras por IP: a vasta maioria delas o integram como padrão. Já no método do streaming por *download* progressivo, usa-se um servidor HTTP para transferir o vídeo entre o cliente e o servidor, de maneira *stateless*. Os clientes fazem requisições de multimídia, que é enviado progressivamente para um buffer local. Assim que ele tem informação o suficiente, o vídeo começa a tocar. Se a taxa de *playback* excede a taxa de transmissão de dados, então o vídeo é interrompido até que o buffer seja preenchido adequadamente. O *streaming* adaptativo por sua vez, consiste em detectar a largura de banda de rede e capacidade de CPU do cliente para ajustar a qualidade do vídeo transmitido buscando a melhor opção para as condições dadas. Isso requer um *encoder* para prover vídeo em múltiplos *bit rates* diferentes, ou múltiplos *encoders* e pode ser usada uma Content Delivery Network (CDN) para aumentar a escalabilidade. É comum em streaming adaptativo utilizar a técnica de *stream switching*. Esse é um método híbrido que usa HTTP como o protocolo de entrega no lugar de definir um novo protocolo. Os dados de multimídia são segmentados em pequenas partes de mesmo tamanho, codifica-

dos no formato desejado e armazenadas em um servidor. Os clientes então requisitam os segmentos sequencialmente por download progressivo e eles são tocados em ordem já que são contíguos. O resultado é uma experiência de usuário praticamente livre de gargalos de buffering em condições normais de rede e CPU.

3.2.2 algoritmos de reconhecimento de imagem

Existem duas abordagens convencionais principais para detecção de objetos: "diferença temporal" e "subtração de fundo". A primeira abordagem consiste na subtração de dois quadros consecutivos seguida por uma limiarização. A segunda técnica é baseada na subtração de um modelo de fundo ou referência e a imagem atual seguida por um processo de rotulagem. Além destas, existem abordagens baseadas em aprendizado de máquina, como o Haar Cascade, onde classificadores são treinados usando imagens positivas (com objetos) e imagens negativas (sem objetos) para criar uma função que pode ser usada para detectar algo específico em outras imagens. O Haar Cascade é especialmente eficaz para detecção de faces e oferece processamento rápido, sendo uma opção viável para detecção em tempo real. No entanto, pode apresentar desafios em cenas complexas ou com objetos ocluídos. Além disso, há técnicas mais avançadas como Deep Learning, que, através de Redes Neurais Convolucionais (CNNs) e variações como R-CNN, Faster R-CNN, YOLO, e SSD, conseguem alta precisão e robustez na detecção de diversos tipos de objetos em cenários variados, embora exijam um grande volume de dados para treinamento e recursos computacionais consideráveis. Diferentemente das técnicas de subtração de fundo e diferença temporal, que são menos intensivas computacionalmente, as técnicas de aprendizado de máquina e aprendizado profundo podem requerer mais recursos e um conjunto de treinamento de dados, mas oferecem maior precisão e capacidade de generalização, tornando-se preferíveis em aplicações mais complexas ou quando a alta precisão é crucial.

3.2.3 armazenamento, resolução e compressão

Uma das funções fundamentais presente em quase todos os sistemas de vigilância é o armazenamento do vídeo e sua leitura posterior. A maneira de como é feita esse armazenamento depende de vários fatores: volume de dados, período de retenção, velocidade de acesso, largura de banda (se houver interação com sistemas pela rede), escalabilidade, segurança, conformidade legal e outros. Em especial, a resolução da câmera tem uma forte relação com o volume de dados que por sua vez afeta muitos dos outros fatores também. A tabela a seguir detalha o espaço de armazenamento necessário, em Terabytes, para arquivar 30 dias de filmagem, capturada em diferentes resoluções de 12 câmeras, operando continuamente durante 24 horas por dia.

Resolução	MJPEG	H.264	H.265
1MP (1280x720)	27,5 TB	3,97 TB	1,34 TB
2MP (1920x1080)	61 TB	8,9 TB	3,0 TB
3MP (2048x1536)	93 TB	13 TB	4 TB
5MP (2592x1904)	150 TB	21,74 TB	7,37 TB
8MP (4K)	464,38 TB	67,10 TB	22,76 TB

Tabela 1 – Tabela comparativa de resoluções e compressão versus tamanho, baseada em dados obtidos do Surveillance Capacity Calculator ([DIGITAL, 2023](#)).

Essas resoluções na tabela são as mais comuns encontradas nas câmeras disponíveis no mercado para uso domiciliar e empresarial. As resoluções são medidas em megapixels (MP) e indicam o total de pixels existentes na imagem. Quanto mais pixels, maior a resolução e melhor a qualidade, porém, como é possível perceber pela tabela, mais espaço é necessário para armazenar o vídeo.

A resolução adequada depende de qual vai ser o uso dessas câmeras. A resolução de 1 MP é uma boa escolha para domicílios, pequenas lojas e outros locais onde só é preciso detectar movimento de pessoas sem muito detalhes. Full HD (2 MP) já é capaz de fazer reconhecimento facial ou de texto, tornando-se uma opção adequada para locais que requerem um nível de detalhe intermediário. Para a monitoração de espaços amplos, resoluções de 4 MP e 5 MP são recomendadas, sendo que a última oferece uma qualidade superior na ampliação de imagens, com uma menor perda de detalhes. 4K é uma resolução comum em ambientes que exigem identificação precisa, como instituições financeiras no geral, cassinos e áreas de infraestrutura crítica como usinas de energia e estações de tratamento.

Repare que além da resolução o tipo de compressão pode afetar drasticamente o tamanho. MJPEG, por ser um formato antigo usado pela primeira em meados de 1990, é bem menos eficiente mas amplamente adotado. H.264, também conhecido como AVC, é bem mais eficiente e não deixa de ser comum para compressão de vídeos em HD. Por último, o H.265, ou HEVC, é um dos métodos mais eficientes mas nem sempre suportado pelos equipamentos de NVR e DVR. Até mesmo alguns protocolos de transmissão de vídeo em rede como o WebRTC e RTMP não possuem suporte esse padrão.

3.2.4 vulnerabilidades de segurança

Em relação aos riscos de segurança, as câmeras IP apresentam várias vulnerabilidades que podem expô-las a ameaças. Uma das falhas mais comuns é a utilização de credenciais de login padrão, que são facilmente acessíveis para qualquer pessoa com conhecimento básico em segurança de rede. Além disso, políticas de autenticação fracas frequentemente permitem a seleção de senhas simples e fáceis de adivinhar. A criptografia inadequada de informações confidenciais é outro ponto de preocupação, pois muitas vezes

os dados são transmitidos sem qualquer processo de criptografia, tornando-os suscetíveis a interceptações e ataques. Como exemplo dessas falhas, em (ABDALLA; VAROL, 2020) uma câmera smart identifica como Onvif YY HD passou por um teste de penetração. Nesse teste, um ataque do tipo man in the middle (MITM) foi capaz de visualizar em texto plano todos os dados transmitidos pela câmera, inclusive credenciais.

O local onde o vídeo é armazenado e seu acesso também podem ser vetores para ataques. Se o VMS utilizar armazenamento em nuvem, por exemplo, como a conexão em rede é necessária praticamente o tempo todo existe a chance de ocorrer um Denial Of Service (DoS). Por outro lado, sistemas que operam isoladamente da internet, utilizando DVR ou NVR em redes locais, embora menos suscetíveis a DoS, ainda requerem uma gestão do acesso por parte de usuários e do controle dos recursos. Portanto, mecanismos de controle de acesso técnicos, físicos e administrativos precisam ser implementados e documentados independente da escolha armazenamento, como forma de mitigar vulnerabilidades.

3.2.5 custos de operação

4 Revisão Bibliográfica

Um ou mais capítulos (por exemplo, se há duas linhas de trabalhos relacionados).

5 Desenvolvimento

Nesta seção, apresentaremos uma descrição detalhada do processo de construção do SSCS. O sistema foi projetado com uma arquitetura modular, composta por componentes independentes, o que justifica a escolha de representá-los por meio de diagramas de componentes do UML. Os componentes são definidos por um conjunto de métodos essenciais que admitem várias implementações, adaptáveis às demandas do usuário. Contudo, é importante que essas implementações obedeçam a certos padrões para garantir que o sistema opere de forma coesa e integrada.

5.1 Ambiente de desenvolvimento

Para possibilitar testes e criar um ambiente de desenvolvimento que reproduza fielmente as condições em que o sistema operará quando implantado em um ambiente de produção, faz sentido criar uma forma de simular um fluxo de vídeo. É importante esclarecer que, ao mencionar o "sistema em produção", está sendo feita uma referência à aplicação prática do software em contextos comerciais, residenciais ou outras situações reais onde a vigilância é utilizada para fins de segurança e monitoramento.

Para criar esse ambiente de simulação, utilizamos o MediaMTX ([ROS, 2023](#)), um servidor de mídia em tempo real de código aberto. O MediaMTX facilita a publicação, leitura, gravação e roteamento de fluxos de vídeo. Embora tenha sido originalmente projetado como um roteador de mídia de ponta a ponta, ele oferece uma plataforma robusta que permite ao SSCS acessar e consumir fluxos de vídeo. No entanto, para tornar o servidor operacional, é necessário publicar a mídia nele.

De acordo com a documentação do MediaMTX, uma das abordagens recomendadas para isso é o uso do ffmpeg, um framework de multimídia que oferece funcionalidades como codificação, decodificação, transcodificação, muxing, demuxing, streaming e reprodução de dados de mídia. Utilizando o ffmpeg ([BELLARD, 2023](#)), é possível enviar fluxos de vídeo para o servidor MediaMTX, criando assim um ambiente adequado para a integração e operação eficiente com o SSCS. Observe também que essas duas ferramentas são plenamente capazes de lidar não apenas com ambientes de simulação, mas também com ambientes de produção. A intensificação aqui é minimizar a diferença entre esses dois cenários, facilitando a transição futura de uma simulação para a prática.

A implementação da tecnologia de contêineres, especialmente por meio do Docker, surge como uma estratégia crucial para reforçar a eficiência e a confiabilidade do ambiente de desenvolvimento e testes. A "dockerização" do ambiente, que inclui tanto o

MediaMTX quanto o ffmpeg, contribui significativamente para a criação de um sistema mais robusto e controlado. Isso ocorre porque, com o Docker, é possível encapsular o software e suas dependências em um contêiner isolado, o que evita os tradicionais problemas de incompatibilidade ou conflitos com os sistemas onde eles são implantados. Esse nível de isolamento promove a consistência entre os ambientes de desenvolvimento, teste e produção, minimizando assim as discrepâncias que geralmente surgem quando se transita entre diferentes plataformas e infraestruturas. Além disso, o Docker facilita a automação e a replicação de contêineres, o que acelera significativamente o ciclo de desenvolvimento, permitindo que os desenvolvedores se concentrem mais na lógica de aplicação do que em questões operacionais. Em um cenário onde a agilidade, a escalabilidade e a estabilidade são indispensáveis, a utilização do Docker não apenas otimiza o uso dos recursos e melhora o desempenho, mas também simplifica o processo de integração e entrega contínua (CI/CD), fundamentais para a evolução consistente do SSCS. A seguir, está um dos métodos, utilizando-se essa ferramenta com o terminal, para replicar localmente os sistemas externos básicos dos quais o SSCS depende.

```
1  docker build -t sscs-mtx ./dev
2
3  docker run -d --rm --network=host -p 8554:8554 -p 8889:8889 \
4  -v ./dev/mediamtx.yml --name sscs-mtx sscs-mtx
5
6  docker run -d --rm --network=host --name sscs-postgres \
7  -e POSTGRES_PASSWORD=gorm -e POSTGRES_USER=gorm -e \
8  POSTGRES_DB=gorm -p 5432:5432 postgres
```

Esse script realiza as seguintes ações: primeiro constrói uma imagem Docker chamada **sscs-mtx** a partir de um Dockerfile no diretório `./dev`. Esse arquivo tem como base uma imagem do **mediamtx** e responsável por encapsular as amostras de mídia. Depois, ele cria e inicializa um contêiner a partir dessa imagem, mapeia portas de rede e monta o arquivo de configuração **mediamtx.yml** dentro do contêiner. A intenção nesse caso é recriar um servidor de mídia localmente que contenha alguns vídeos para testes que podem ser consumidos pelo SSCS.

Em seguida, ele cria e inicia outro contêiner usando a imagem oficial do PostgreSQL, configurando-o com variáveis de ambiente e mapeando a porta 5432. Ambos os contêineres usam a rede do host para comunicação.

No diretório `/sscs/dev`, presente no repositório de código do projeto, encontram-se todos os arquivos requeridos para a construção do contêiner que compõe o servidor intermediário, configurando, assim, um ambiente de desenvolvimento plenamente operacional.

Adicionalmente, um arquivo **Makefile**, localizado em **/sscs/**, facilita este processo, por meio de um comando que automatiza a montagem desse ambiente. Vale ressaltar que os testes automatizados utilizam um contêiner similar, porém este inclui um executável do SSCS, garantindo que as funcionalidades sejam testadas num contexto que replica fielmente o ambiente de produção.

Por alto nível, o sistema assume uma configuração que envolve a comunicação entre dois componentes: um encarregado da transferência das imagens da câmera para um servidor intermediário e outro, o servidor intermediário, que cria uma corrente de multimídia destinada ao SSCS.

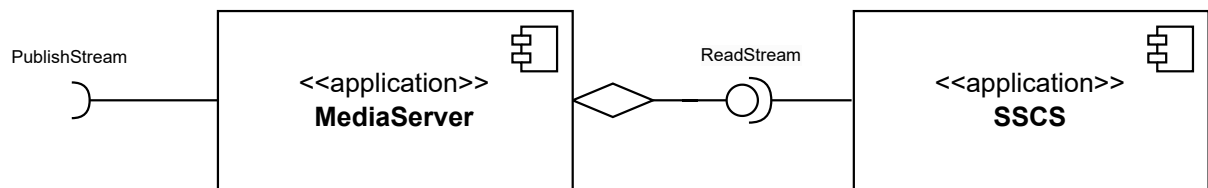


Figura 5 – Arquitetura de transmissão, múltiplas fontes para um servidor de mídia

A imagem mostra a configuração básica do SSCS, onde câmeras ou outras fontes de transmissão de multimídia podem encaminhar fluxos para um servidor intermediário. Esse servidor de mídia intermediário disponibiliza uma interface chamada **PublishStream**, que possibilita a recepção de fluxos de multimídia de diversas origens distintas. O símbolo em forma de losango indica que o **MediaServer** age como um agregador, o que significa que ele pode interagir simultaneamente com várias aplicações do SSCS. Contudo, é importante observar que essa conexão é de natureza flexível: o **MediaServer** pode operar de forma independente, sem depender do SSCS.

Os servidores intermediários desempenham o papel de proxies para o servidor que opera o SSCS. A utilização de um servidor de mídia como proxy apresenta diversas vantagens. Por exemplo, pode ser necessário converter o protocolo de transmissão de RTSP para WebRTC ou HLS, realizar balanceamento de carga, ou até mesmo implementar autenticação externa.

À seguir, será apresentada uma visão geral de cada componente do sistema, sua função, como ele interage com as outras partes do sistema e detalhes de implementação. Note que cada componente é abstrato. Ou seja, é uma base que deve ser adaptada de acordo com os protocolos, formato de imagem, banco de dados e outros detalhes específicos de implementação. Em cada seção, há também exemplos de implementações acrescentando os detalhes.

5.2 Gravador

Esse é o componente principal do sistema. Sua função básica consiste em receber pacotes da corrente de vídeo por meio de algum protocolo de rede, concatená-los e codificá-los em um arquivo único, com subsequente armazenamento no disco ou nuvem em segmentos de arquivos. Estes arquivos são registrados com uma timestamp, refletindo o instante de criação do arquivo de gravação.

Assim como todos os outros componentes, o gravador precisa implementar todas as funções de uma interface padrão, à seguir. Os métodos `Start()`, `Stop()` e `setupLogger()` fazem parte de qualquer componente. Os dois primeiros são evidentemente usados para iniciar e parar o serviço. O método `setupLoogger()` por sua vez cria um padrão de logs customizado para aquele componente. Isso facilita a depuração de problemas e performance. Os últimos dois ficam á critério da implementação.

```
1 // Recorder is an interface for recording streams.
2 type Recorder interface {
3     Start() error // Starts the recording
4     Stop() error  // Stops the recording
5     setupLogger()
6     record() error
7     sendFrame(image.Image) error
8 }
```

No contexto do desenvolvimento em Go, o método `record` implementado pelo componente `Recorder` é privado, o que significa que sua acessibilidade se restringe ao pacote onde foi declarado. Esse método realiza o consumo de uma corrente de dados exposta por um servidor de mídia. Na prática, uma entidade do tipo `MediaServer` fornece uma interface para que o `Recorder` desempenhe suas operações. A gravação também não é exclusiva a um protocolo específico: qualquer sistema que tenha a capacidade de transmitir mídia utilizando os protocolos RTSP, WebRTC ou SRT pode ser integrado também, assumindo que o componente de gravador dê suporte.

Para realizar a comunicação com o indexador de maneira eficiente, utiliza-se a interface provida `RecordedEvent`, que encapsula informações como o tempo de início e término da gravação. Esta estrutura compartilhada é transmitida através de um canal com buffer, permitindo assim que o gravador continue seu processo sem aguardar a finalização do indexador para cada evento de gravação. Em contraste, se um canal sem buffer fosse utilizado, o gravador necessitaria esperar pela conclusão da indexação de cada evento antes de iniciar a gravação subsequente, resultando em uma sincronização mais rígida e

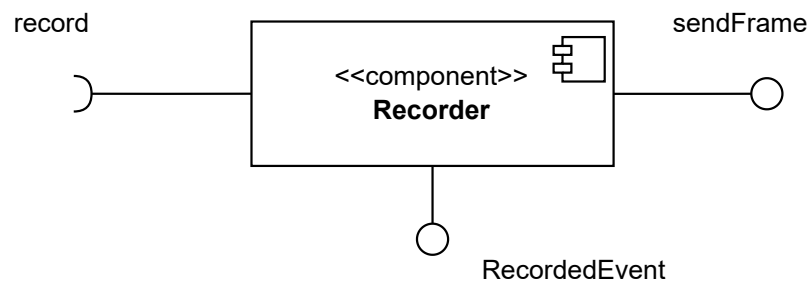


Figura 6 – Diagrama isolado do gravador

potencialmente uma latência maior no sistema, o que seria menos vantajoso em questões de performance.

O método **sendFrame** aloca quadros de vídeo em um canal. Em relação a comunicação com outros componentes, seu funcionamento é semelhante ao da interface anterior, utilizando também um canal com buffer. A diferença está apenas no tamanho do buffer do canal. A responsabilidade dessa função é pegar quadros recebidos pelo gravador e enviá-los para a leitura por detectores de imagem.

5.2.1 Implementação de gravador com RTSP e compressão H.264

Considerando a prevalência do protocolo RTSP e do codec de compressão H.264 no universo das câmeras IP, é fundamental que o SSCS empregue essas tecnologias em seu gravador. Essa implementação utiliza um cliente RTSP, um multiplexador para converter quadros H.264 em um arquivo MPEG-TS e um decodificador para extrair quadros de vídeo.

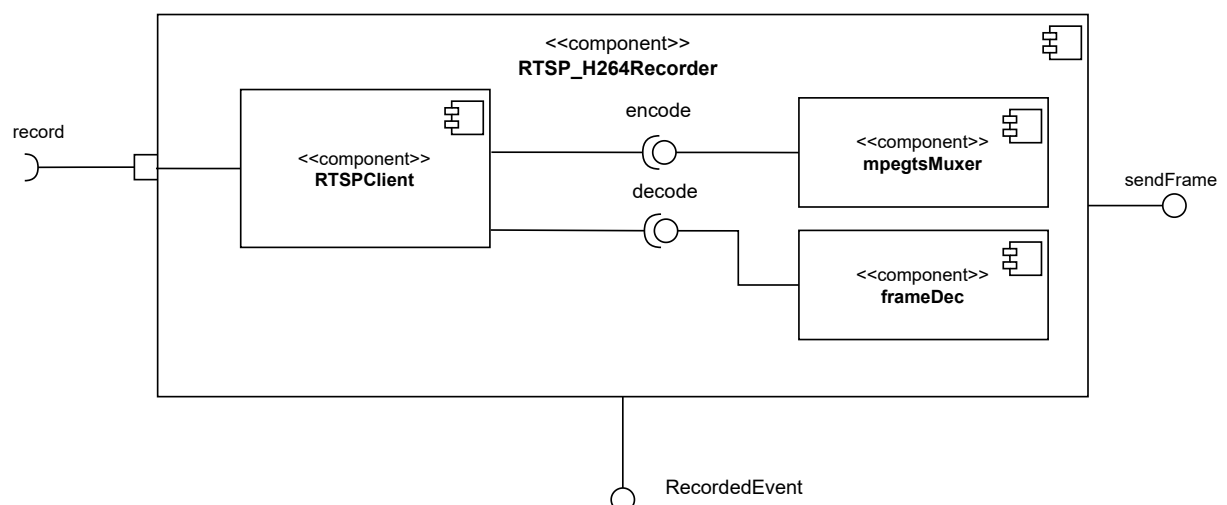


Figura 7 – Diagrama isolado do gravador RTSP

O cliente RTSP primeiro emite um requisição do tipo DESCRIBE ao servidor de mídia, que por sua vez responde com a descrição dos dados da sessão, de acordo com o protocolo SDP. Essa descrição contém as informações necessárias para iniciar o streaming.

Em seguida, o cliente processa cada pacote RTP recebido e extrai as Access Units (AUs). Essas, são formadas por um conjunto de unidades de Network Abstraction Layer (NAL) sempre contendo uma imagem codificada primária, tipicamente um quadro do tipo I. Além da imagem codificada primária, uma AU também pode conter uma ou mais imagens codificadas redundantes ou outras unidades NAL que não contenham fatias ou partições de dados de fatias de uma imagem codificada. A decodificação de uma AU sempre resulta em uma imagem decodificada. Por isso, as AUs são passadas para o multiplexador que codifica elas e organiza as unidades NAL para formar um único segmento de vídeo e salvá-lo no disco.

O decodificador que extrai quadros de vídeo faz isso iterando as unidades NAL até encontrar um Quadro I, que contém uma imagem completa. Quando encontrado, ele usa o método `sendFrame()` para comunicar a imagem à um canal com buffer. A intenção nisso é facilitar a comunicação com a parte do software que realiza o reconhecimento de imagem em tempo real.

Por último, uma canal que recebe exclusivamente interface `RecordedEvent` será alimentado toda vez que uma gravação for concluída, contendo o nome do vídeo e uma timestamp do fim da gravação. Isso pode ser usado para comunicar o registro de cada arquivo à outros componente do sistema.

5.3 Indexador

O indexador é responsável por fazer referências aos vídeos em um banco de dados e salvar eventos de reconhecimento de imagem. Posteriormente, exploraremos como o indexador interage com uma API que simplifica e expande o processo de busca e recuperação de gravações. É importante salientar também que a implementação do indexador não se torna mandatória, a menos que corresponda a uma necessidade específica do usuário.

A interface do indexador é bem mais simplificada. Ele requer apenas um método privado, chamado `listen()`. A única função desse método é esperar por eventos de indexação enviados pelos canais de outros componentes e encaminhá-los para as funções que lidam o registro da informação no banco de dados.

```
1 // an interface for indexing data into a database
2 type Indexer interface {
3     Start() error
4     Stop() error
5     setupLogger()
6     listen() error
7 }
8
```

No diagrama à seguir, o indexador não provê nenhuma interface, ele apenas requer algumas. Isso porque ele lida exclusivamente com a escrita das informações que recebe por meio dos eventos recebidos por meio dos canais. Um detalhe pertinente é o fato do indexador ser agnóstico quanto ao banco de dados usado. Do ponto de vista da arquitetura, o banco de dados é uma não-entidade. É um detalhe que não se eleva ao nível de um elemento arquitetônico. Ele é uma utilidade que provê acesso aos dados. No mesmo ponto de vista, essa utilidade é irrelevante pois é um detalhe de baixo nível, ou seja, um mecanismo. (MARTIN, 2018b)

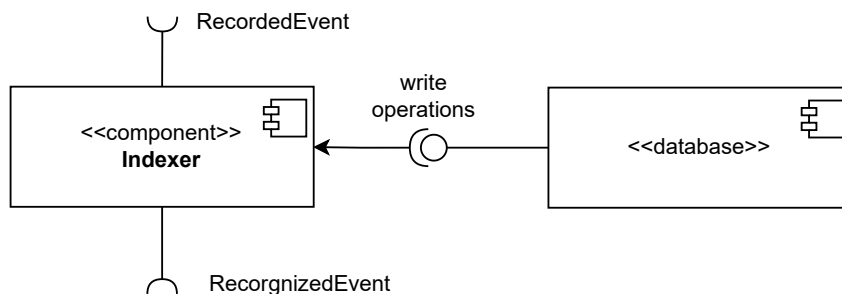


Figura 8 – Diagrama isolado do indexador

6 Conclusão

E daí?

Referências

ABDALLA, P. A.; VAROL, C. Testing iot security: The case study of an ip camera. In: *2020 8th International Symposium on Digital Forensics and Security (ISDFS)*. [S.l.: s.n.], 2020. p. 1–5. Citado na página 28.

BELLARD, F. *FFmpeg*. 2023. GitHub repository. Disponível em: <<https://github.com/FFmpeg/FFmpeg>>. Citado na página 30.

BURNS, B. *Designing distributed systems: patterns and paradigms for scalable, reliable services*. [S.l.]: "O'Reilly Media, Inc.", 2018. Citado na página 17.

BURNS, B. *Designing distributed systems: patterns and paradigms for scalable, reliable services*. In: _____. [S.l.]: "O'Reilly Media, Inc.", 2018. cap. Part I. Single-node Patterns. Citado na página 17.

DIGITAL, W. *Surveillance Capacity Calculator*. 2023. Accessed on 2023-11-11. Disponível em: <<https://www.westerndigital.com/tools/surveillance-capacity-calculator>>. Citado 2 vezes nas páginas 6 e 27.

DONOVAN, A. A. A.; KERNIGHAN, B. W. The go programming language. In: _____. [S.l.]: Publisher Name, 2015. cap. Preface, p. 20. ISBN 13: 978-0-13-419044-0. Citado na página 15.

FOUNDATION, A. S. *Apache License, Version 2.0*. 2004. <<https://www.apache.org/licenses/LICENSE-2.0>>. Acesso em: 2023. Citado na página 21.

FOUNDATION, F. S. *GNU General Public License*. 2007. <<https://www.gnu.org/licenses/gpl-3.0.html>>. Acessado em: 2023. Citado na página 21.

GENERAL Public Virus. The Jargon File. Disponível em: <<http://catb.org/jargon/html/G/General-Public-Virus.html>>. Citado na página 22.

GLINSKY, A. *Theremin: ether music and espionage*. Urbana: University of Illinois Press, 2000. 46–47 p. ISBN 0252025822. Citado na página 24.

GUPTA, A. *Video Surveillance Market Research Report Information By Component (Hardware, Software, Services), By Hardware (Camera, Storage System And Others), By Application (Residential, Commercial, Defense And Infrastructure), And By Region (North America, Europe, Asia-Pacific, And Rest Of The World) – Market Forecast Till 2030*. 2018. Disponível em: <<https://www.marketresearchfuture.com/reports/video-surveillance-market-95q7>>. Citado na página 10.

INITIATIVE, O. S. *The MIT License*. 2023. Disponível em: <<https://opensource.org/licenses/MI>>. Citado na página 21.

KAY, R. *Unix Systems Programming: Communication, Concurrency And Threads, 2/E*. [S.l.]: Pearson Education, 2003. ISBN 0-13-042411-0. Citado na página 20.

LEE, J. *Scalable continuous media streaming systems: Architecture, design, analysis and implementation*. [S.l.]: John Wiley & Sons, 2005. Citado na página 14.

MARDJAN, M. J.; JAHAN, A. Open reference architecture for security and privacy. CreateSpace Independent Publishing Platform, 2016. Citado na página 10.

MARTIN, R. Clean architecture: A craftsman's guide to software structure and design. In: _____. [S.l.]: Prentice Hall, 2018. cap. Part III: Design Principles. ISBN 9780134494166. Citado na página 21.

MARTIN, R. Clean architecture: A craftsman's guide to software structure and design. In: _____. [S.l.]: Prentice Hall, 2018. cap. 30. ISBN 9780134494166. Citado na página 36.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, v. 2014, n. 239, p. 2, 2014. Disponível em: <https://dl.acm.org/doi/fullHtml/10.5555/2600239.2600241>. Citado na página 17.

PRESTON-WERNER, T. *Semantic Versioning 2.0.0*. 2013. Disponível em: <https://semver.org/>. Citado na página 19.

QIANG, X. The road to digital unfreedom: President xi's surveillance state. *Journal of Democracy*, Johns Hopkins University Press, v. 30, n. 1, p. 53–67, 2019. Citado na página 10.

RAO, A.; LANPHIER, R.; SCHULZRINNE, H. *Real Time Streaming Protocol (RTSP)*. RFC Editor, 1998. RFC 2326. (Request for Comments, 2326). Disponível em: <https://www.rfc-editor.org/info/rfc2326>. Citado na página 16.

ROS, A. *MediaMTX*. 2023. GitHub repository. Disponível em: <https://github.com/bluenviro/mediamtx>. Citado na página 30.

WANG, H.; KWONG, S.; KOK, C.-W. Efficient prediction algorithm of integer dct coefficients for h.264/avc optimization. *IEEE Transactions on Circuits and Systems for Video Technology*, v. 16, n. 4, p. 547–552, 2006. Citado na página 23.

Apêndices

APÊNDICE A – Quisque libero justo

Quisque facilisis auctor sapien. Pellentesque gravida hendrerit lectus. Mauris rutrum sodales sapien. Fusce hendrerit sem vel lorem. Integer pellentesque massa vel augue. Integer elit tortor, feugiat quis, sagittis et, ornare non, lacus. Vestibulum posuere pellentesque eros. Quisque venenatis ipsum dictum nulla. Aliquam quis quam non metus eleifend interdum. Nam eget sapien ac mauris malesuada adipiscing. Etiam eleifend neque sed quam. Nulla facilisi. Proin a ligula. Sed id dui eu nibh egestas tincidunt. Suspendisse arcu.

APÊNDICE B – Coisas que fiz e que achei interessante mas não tanto para entrar no corpo do texto

Nunc velit. Nullam elit sapien, eleifend eu, commodo nec, semper sit amet, elit. Nulla lectus risus, condimentum ut, laoreet eget, viverra nec, odio. Proin lobortis. Curabitur dictum arcu vel wisi. Cras id nulla venenatis tortor congue ultrices. Pellentesque eget pede. Sed eleifend sagittis elit. Nam sed tellus sit amet lectus ullamcorper tristique. Mauris enim sem, tristique eu, accumsan at, scelerisque vulputate, neque. Quisque lacus. Donec et ipsum sit amet elit nonummy aliquet. Sed viverra nisl at sem. Nam diam. Mauris ut dolor. Curabitur ornare tortor cursus velit.

Morbi tincidunt posuere arcu. Cras venenatis est vitae dolor. Vivamus scelerisque semper mi. Donec ipsum arcu, consequat scelerisque, viverra id, dictum at, metus. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut pede sem, tempus ut, porttitor bibendum, molestie eu, elit. Suspendisse potenti. Sed id lectus sit amet purus faucibus vehicula. Praesent sed sem non dui pharetra interdum. Nam viverra ultrices magna.

Aenean laoreet aliquam orci. Nunc interdum elementum urna. Quisque erat. Nullam tempor neque. Maecenas velit nibh, scelerisque a, consequat ut, viverra in, enim. Duis magna. Donec odio neque, tristique et, tincidunt eu, rhoncus ac, nunc. Mauris malesuada malesuada elit. Etiam lacus mauris, pretium vel, blandit in, ultricies id, libero. Phasellus bibendum erat ut diam. In congue imperdiet lectus.

Anexos

ANEXO A – Eu sempre quis aprender latim

Sed mattis, erat sit amet gravida malesuada, elit augue egestas diam, tempus scelerisque nunc nisl vitae libero. Sed consequat feugiat massa. Nunc porta, eros in eleifend varius, erat leo rutrum dui, non convallis lectus orci ut nibh. Sed lorem massa, nonummy quis, egestas id, condimentum at, nisl. Maecenas at nibh. Aliquam et augue at nunc pellentesque ullamcorper. Duis nisl nibh, laoreet suscipit, convallis ut, rutrum id, enim. Phasellus odio. Nulla nulla elit, molestie non, scelerisque at, vestibulum eu, nulla. Ut odio nisl, facilisis id, mollis et, scelerisque nec, enim. Aenean sem leo, pellentesque sit amet, scelerisque sit amet, vehicula pellentesque, sapien.

ANEXO B – Coisas que eu não fiz mas que achei interessante o suficiente para colocar aqui

Sed consequat tellus et tortor. Ut tempor laoreet quam. Nullam id wisi a libero tristique semper. Nullam nisl massa, rutrum ut, egestas semper, mollis id, leo. Nulla ac massa eu risus blandit mattis. Mauris ut nunc. In hac habitasse platea dictumst. Aliquam eget tortor. Quisque dapibus pede in erat. Nunc enim. In dui nulla, commodo at, consectetur nec, malesuada nec, elit. Aliquam ornare tellus eu urna. Sed nec metus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

ANEXO C – Fusce facilisis lacinia dui

Phasellus id magna. Duis malesuada interdum arcu. Integer metus. Morbi pulvinar pellentesque mi. Suspendisse sed est eu magna molestie egestas. Quisque mi lorem, pulvinar eget, egestas quis, luctus at, ante. Proin auctor vehicula purus. Fusce ac nisl aliquam ante hendrerit pellentesque. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi wisi. Etiam arcu mauris, facilisis sed, eleifend non, nonummy ut, pede. Cras ut lacus tempor metus mollis placerat. Vivamus eu tortor vel metus interdum malesuada.