

**UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROJETO DE GRADUAÇÃO**



Felippe Mendonça de Queiroz

**Desenvolvimento da Infraestrutura de um Espaço
Inteligente baseado em Visão Computacional e IoT**

Vitória-ES

Dezembro/2016

Felippe Mendonça de Queiroz

Desenvolvimento da Infraestrutura de um Espaço Inteligente baseado em Visão Computacional e IoT

Parte manuscrita do Projeto de Graduação do aluno Felippe Mendonça de Queiroz, apresentado ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheiro Eletricista.

Vitória-ES

Dezembro/2016

Felippe Mendonça de Queiroz

Desenvolvimento da Infraestrutura de um Espaço Inteligente baseado em Visão Computacional e IoT

Parte manuscrita do Projeto de Graduação do aluno Felippe Mendonça de Queiroz, apresentado ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheiro Eletricista.

Aprovado em 15, de Dezembro de 2016.

COMISSÃO EXAMINADORA:

Profa. Dra. Mariana Rampinelli Fernandes

Instituto Federal do Espírito Santo
Orientadora

Profa. Dra. Raquel Frizera Vassallo

Universidade Federal do Espírito Santo
Coorientadora

Profa. Dra. Roberta Lima Gomes

Universidade Federal do Espírito Santo
Examinadora

Profa. M.Sc. Cristina Klippel Dominicini

Instituto Federal do Espírito Santo
Examinadora

Vitória-ES

Dezembro/2016

RESUMO

Dispositivos eletrônicos e robóticos vem se tornando cada vez mais acessíveis a usuários residenciais, trazendo mais conforto, praticidade e segurança. Esse avanço possibilitou a criação de ambientes equipados com uma rede de sensores e capazes de tomar decisões de acordo com as necessidades do usuário. Surge então o termo ambientes inteligentes. Diversas pesquisas desenvolvidas em ambientes inteligentes utilizam informações de localização e rastreamento de objetos móveis para realizar o controle de um robô no ambiente, ou ainda, detectar pessoas e reconhecer gestos, como uma maneira de interface com o usuário. Essas e outras aplicações fazem com que as câmeras sejam sensores interessantes para equipar ambientes inteligentes. Um dos problemas encontrados na implementação de um espaço inteligente é a interconexão dos diferentes tipos de dispositivos que o compõe. Assim, a hipótese levantada por este projeto é que o uso do conceito de Internet das coisas (IoT, do inglês *Internet Of Things*) pode auxiliar no desenvolvimento de uma arquitetura para o espaço inteligente que seja flexível e escalável, melhorando, assim, o seu desempenho e tornando-o mais intuitivo para o usuário. Desta maneira, este trabalho tem como objetivo desenvolver uma infraestrutura para um espaço inteligente baseado em visão computacional e em Internet das coisas. A infraestrutura deve prover suporte para o desenvolvimento de aplicações em diversos temas, tais como visão computacional, reconhecimento de padrões e robótica móvel. Essa infraestrutura deve ser flexível de tal forma que seja possível substituir os equipamentos por outros de marcas e/ou modelos diferentes, ou ainda, incorporar novos dispositivos, sem que grandes alterações na infraestrutura sejam necessárias. Isso é possível a partir da padronização das diferentes camadas da arquitetura, de forma que alterações em uma camada não interfira em outra.

Palavras-chave: Espaços inteligentes; Internet das coisas; Visão computacional; Robótica.

LISTA DE FIGURAS

Figura 1 – Histórico e perspectiva de crescimento do número de dispositivos conectados e da população mundial.	12
Figura 2 – Esquema 3D do espaço inteligente instalado na UFES.	16
Figura 3 – Representação de uma arquitetura de Internet das coisas em uma cidade inteligente.	19
Figura 4 – Modelo funcional da arquitetura de referência IoT-A.	20
Figura 5 – Estrutura de camadas para aplicações em máquinas virtuais e <i>containers</i> . .	22
Figura 6 – Câmera Point Grey Blackfly modelo BFLY-PGE-09S2C-CS.	27
Figura 7 – <i>Switch</i> modelo SG100D-08P.	28
Figura 8 – Roteador modelo EA2700.	28
Figura 9 – Microcomputador Dell modelo OptiPlex 9020.	29
Figura 10 – Bancada montada com os equipamentos para a fase de desenvolvimento.	29
Figura 11 – Robô Móvel <i>Pioneer P3-AT</i>	30
Figura 12 – Minicomputador Raspberry Pi 2 Model B com adaptador <i>wireless</i> e conversor USB-Serial.	30
Figura 13 – Representação simplificada em camadas da arquitetura proposta.	31
Figura 14 – Representação das relações entre entidades física e virtual, com os serviços e recursos mínimos definidos.	32
Figura 15 – Representação em camadas da arquitetura proposta com detalhes da comunicação.	34
Figura 16 – Representação da comunicação entre um produtor e um consumidor através de uma fila.	34
Figura 17 – Representação da comunicação entre um produtor e dois consumidor através de uma fila.	35
Figura 18 – Representação da comunicação entre um produtor e dois consumidor, com um <i>exchange</i> e duas filas.	35
Figura 19 – Representação da comunicação entre um produtor e dois consumidor, com um <i>exchange</i> , duas filas e dois diferentes <i>bindings</i>	36
Figura 20 – Representação da comunicação cliente/servidor de uma implementação RPC.	37
Figura 21 – Representação em camadas da arquitetura proposta com detalhamento do funcionamento interno do <i>broker</i> , com exemplos do padrão <i>pub/sub</i> e implementação de serviços.	38
Figura 22 – Imagem de uma câmera com perda fragmentos devido à perda de pacotes.	39
Figura 23 – Representação das camadas de comunicação e sensoriamento, com detalhamento para entidades do tipo câmera.	41

Figura 24 – Representação das grandezas associadas ao robô <i>Pioneer 3 AT</i> : (a) odometria e (b) velocidades.	42
Figura 25 – Representação das camadas de comunicação e sensoriamento, com detalhamento para entidades do tipo robô.	43
Figura 26 – Representação gráfica da aplicação de um <i>delay</i> na amostragem do dado de uma entidade.	44
Figura 27 – Fluxograma que representa o serviço de sincronismo de entidades. . . .	46
Figura 28 – Representação gráfica do processo de leitura de <i>timestamp</i> , cálculo e aplicação de <i>delay</i> nas entidades a serem sincronizadas.	47
Figura 29 – Representação em camadas do funcionamento do serviço de sincronismo para duas entidades.	48
Figura 30 – Resultado obtido utilizando o serviço de reconstrução 3D.	48
Figura 31 – Representação em camadas do funcionamento do serviço de reconstrução 3D.	49
Figura 32 – Gráfico da latência medida para cinco resoluções diferentes ao se variar a quantidade de consumidores simultâneos de 1 até 14.	58
Figura 33 – Gráfico do <i>throughput</i> medido para cinco resoluções diferentes ao se variar a quantidade de consumidores simultâneos de 1 até 14.	59
Figura 34 – Montagem das câmeras e da placa com FPGA para o experimento de validação do serviço de sincronismo.	60
Figura 35 – Imagem das quatro câmeras mostrando o número exido pelos <i>displays</i> (a), e corte feito na imagem para facilitar a leitura dos valores (b). . . .	60
Figura 36 – Imagens dos <i>displays</i> do experimento com intervalo de amostragem igual a 1 s, em 10 instantes igualmente espaçados após o início.	61
Figura 37 – Imagens dos <i>displays</i> do experimento com intervalo de amostragem igual a 36 segundos, em 10 instantes igualmente espaçados após o início. . . .	62
Figura 38 – Ilustração da influência do tempo de exposição das câmeras no erro de sincronismo admitido.	63
Figura 39 – Informações do <i>broker</i> que mostram a existência de apenas um consumidor para o serviço de reconstrução 3D (aba <i>Consumers</i>), além de mostrar o <i>Routing key</i> do serviço “is.r3d”.	64
Figura 40 – Estatísticas disponíveis na interface <i>web</i> do <i>broker</i> durante o processo de reconstrução 3D, utilizando um consumidor.	65
Figura 41 – Estatísticas disponíveis na interface <i>web</i> do <i>broker</i> durante o processo de reconstrução 3D, utilizando quatro consumidores.	66
Figura 42 – Árvore de diretórios da implementação do projeto.	74

LISTA DE TABELAS

Tabela 1 – Erro máximo em milissegundos para cada amostra dos três experimentos com tempo de amostragem igual à 1 segundo.	61
Tabela 2 – Erro máximo em ms para cada amostra dos três experimentos com tempo de amostragem igual à 36 s.	62

LISTA DE QUADROS

Quadro 1 – Código exemplo que exibe câmeras sincronizadas.	55
Quadro 2 – Código exemplo que exibe a odometria de um robô.	56

SUMÁRIO

1	INTRODUÇÃO	10
1.1	Objetivos	12
1.1.1	Objetivos específicos	13
1.2	Estrutura do Texto	13
2	REFERENCIAL TEÓRICO	15
2.1	Espaços Inteligentes	15
2.2	Internet das coisas	17
2.2.1	Arquitetura IoT	18
2.2.1.1	Comunicação	20
2.3	Virtualização e Computação em Nuvem	21
2.4	Sincronismo	23
2.4.1	Sincronismo de Câmeras	24
3	INFRAESTRUTURA PROPOSTA	26
3.1	Infraestrutura Física	26
3.1.1	Câmeras	26
3.1.2	Equipamentos de Rede	27
3.1.3	Recursos Computacionais	28
3.1.4	Bancada	29
3.1.5	Robô Móvel	29
3.1.6	Minicomputador	30
3.2	Arquitetura IoT	31
3.2.1	Comunicação	33
3.2.1.1	Detalhamento interno do <i>broker</i>	34
3.2.2	Sensoriamento	37
3.2.2.1	Câmeras	37
3.2.2.2	Robôs	41
3.2.3	Serviços do Espaço Inteligente	43
3.2.3.1	Serviço de Sincronismo	44
3.2.3.2	Serviço de Busca de Entidades	45
3.2.3.3	Serviço de Reconstrução 3D	47
3.3	Interfaces de desenvolvimento	50
3.3.1	Cameras	50
3.3.2	Robôs	51
3.3.3	Serviços	52

4	EXPERIMENTOS E RESULTADOS	54
4.1	Exemplos de utilização das interfaces	54
4.1.1	Exibir câmeras sincronizadas	54
4.1.2	Exibir a odometria de um robô	56
4.2	Testes de latência para <i>pub/sub</i> de imagens	57
4.3	Validação do serviço de sincronismo	59
4.4	Testes com o serviço de resconstrução 3D	64
5	CONCLUSÕES E TRABALHOS FUTUROS	67
REFERÊNCIAS BIBLIOGRÁFICAS		69
APÊNDICES		73
APÊNDICE A – ESTRUTURA DO CÓDIGO		74
APÊNDICE B – BIBLIOTECA DE COMUNICAÇÃO		75

1 INTRODUÇÃO

A robótica possibilitou realizar tarefas, até então de difícil realização pelo homem. Um exemplo são tarefas repetitivas encontradas em diversos ambientes industriais. Essas tarefas são melhor executadas por robôs, pois além de serem mais precisos, também as realizam de forma mais segura, visto que o ser humano está sujeito à fadiga. Além disso, algumas tarefas só se tornaram possíveis com a evolução dos dispositivos mecânicos e eletrônicos, como tarefas que requerem manipulação de substâncias perigosas ou aquelas nas quais os objetos manipulados possuem dimensões tão pequenas que, ao serem operados pela mão humana, poderiam ser danificados.

Os dispositivos eletrônicos e robóticos também estão se tornando cada vez mais acessíveis a usuários residenciais, trazendo mais conforto, praticidade e segurança e, consequentemente, estão mais presentes no cotidiano das pessoas. Esses dispositivos são empregados, por exemplo, no monitoramento de pacientes em ambientes domiciliares e hospitalares, mudando a forma na qual os profissionais de saúde lidam com o estado de seus pacientes (PRČIĆ; ĐUREK; ŠIMUNIĆ, 2013). Essa é uma tendência, tendo em vista o aumento da expectativa de vida da população e, portanto, da quantidade de idosos.

Os equipamentos eletrônicos e robóticos vem evoluindo em diversos aspectos. Os mais notáveis são a redução de seu tamanho, ficando cada vez mais portáteis e a diminuição do seu custo, que os tornam mais acessíveis. Além disso, muitos desses têm incorporado interfaces de comunicação, sendo capazes de trocar informações entre si. O aumento da capacidade de processamento, agregada às informações sensoriais compartilhadas, possibilitam que esses dispositivos também tenham certo nível de inteligência. Todas essas características tornaram possível a concepção de ambientes que possam oferecer serviços ao seres humanos baseado em eventos e informações de sensores (HASHIMOTO, 2013).

Desse modo, surge o termo espaços inteligentes. Não existe uma definição precisa para esse termo, uma vez que diversos autores o caracterizam de forma diferente. Ahn et al. (2011) descreve que esses ambientes são dotados de uma rede de sensores capaz de detectar mudanças de estados, além de poder compartilhar suas informações com outros dispositivos, tornando possível a redução de sensores instalados em agentes como robôs de serviços. Para Morioka, Lee e Hashimoto (2002), um ambiente pode ser classificado como inteligente quando possui dispositivos dotados de inteligência distribuídos em sua infraestrutura. Esses dispositivos possuem algum tipo de sensibilidade, capacidade de processamento e funcionalidades de comunicação.

Para Province (2012), esses novos ambientes devem ser capazes de agregar informações,

inteligência, compreensão e tomada de decisões. Tal conceito possibilitou, por exemplo, que a robótica de serviço atingisse um novo patamar, no qual dispositivos robóticos tornaram-se capazes de perceber o meio em que se encontram sem a necessidade da adição de novos sensores à sua estrutura, apenas utilizando a informação que está presente no espaço inteligente.

Espaços inteligentes podem ser equipados com os mais variados tipos de sensores, atuadores e interfaces com o usuário. Câmeras, microfones, *lasers* de varredura e sensores de temperatura são exemplos de sensores que podem compor um espaço inteligente, enquanto *displays*, teclados e alto-falantes são dispositivos que realizam interface com os seres humanos. Como atuadores, um espaço inteligente pode contar com robôs de serviço, que podem ser utilizados para guiar uma pessoa em um ambiente, motores, que podem desempenhar uma variedade de funções, como abrir ou fechar uma cortina, entre outros.

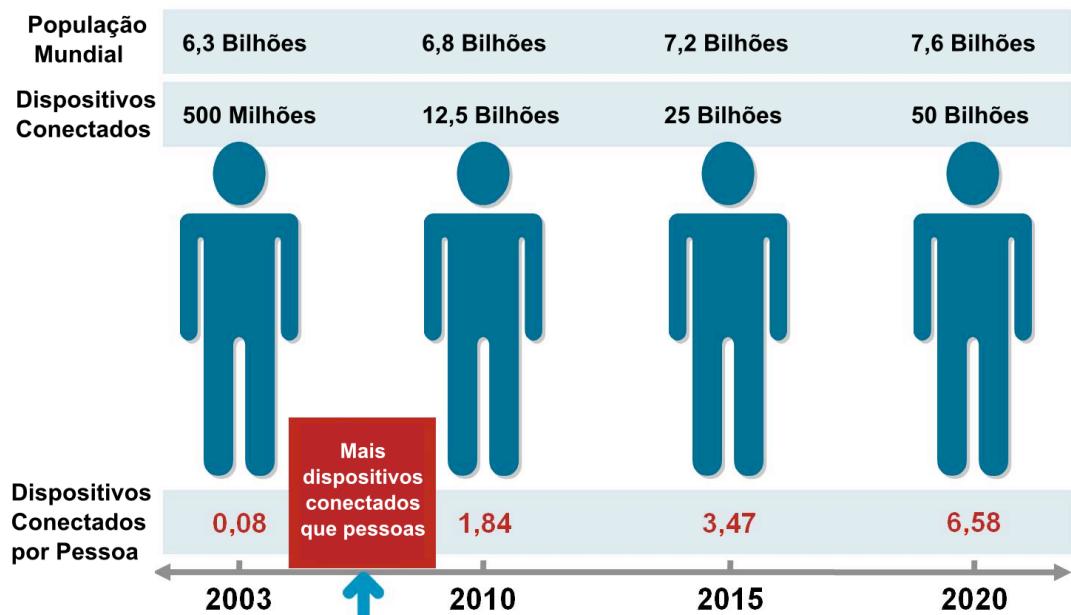
Portanto, um dos problemas encontrados na implementação do espaço inteligente é a interconexão dos diferentes tipos de dispositivos que o compõe. Assim, a hipótese levantada por este projeto é que o uso do conceito de Internet das coisas (IoT, do inglês *Internet Of Things*) pode auxiliar no desenvolvimento de uma arquitetura para o espaço inteligente que seja flexível e escalável, melhorando, assim, o seu desempenho e tornando-o mais intuitivo para o usuário.

Esse novo paradigma muda a forma na qual as coisas relacionam-se com as pessoas, interferindo diretamente no dia-a-dia e no seu comportamento (ATZORI; IERA; MORAUTO, 2010). Ainda que o conceito de Internet das coisas seja atual, muitas de suas características já eram previstas em trabalhos desenvolvidos sobre espaços inteligentes no início da década de 1990 (WEISER, 1991; WEISER, 1993). Uma dessas características é a computação ubíqua ou pervasiva, que consiste em dispositivos conectados entre si e inseridos no meio de forma que não sejam percebidos pelo usuário, tornando a interação com eles o mais natural possível. Para que isso seja viável, é necessário o desenvolvimento de uma infraestrutura flexível para o espaço inteligente, no qual seja possível a substituição ou inserção de novos dispositivos sem que sejam necessárias modificações nas aplicações ou na forma de comunicação. Este projeto apresenta uma proposta de infraestrutura com tais características.

Vale ressaltar que a quantidade de dispositivos com conectividade e capacidade de processamento é crescente e tem perspectivas gigantescas, o que torna um desafio o gerenciamento, monitoramento e endereçamento desses dispositivos. Com a popularização da Internet, os dados dos dispositivos passaram a trafegar na grande rede, sendo que tais dados são provenientes de diferentes tipos de dispositivos, com tamanhos e finalidades diferentes. De

acordo com Evans (2011), em 2003 haviam cerca de 6,3 bilhões de pessoas no planeta e em torno de 500 milhões de dispositivos conectados à Internet. Em 2015, a população cresceu para 7,2 bilhões e o número de dispositivos saltou para 25 bilhões. A perspectiva é de que em 2020 esse número dobre, conforme mostra a Figura 1.

Figura 1 – Histórico e perspectiva de crescimento do número de dispositivos conectados e da população mundial.



Fonte: Adaptado de Evans (2011).

Portanto, nota-se que há uma grande variedade de dispositivos que geram diferentes tipos informações que estão sendo transmitidas pela Internet, e consequentemente, acarretam num grande volume de dados na rede. Além disso, esses dispositivos possuem capacidades de processamento suficientes para que, a partir de informações adquiridas através de sensores e provenientes de outros dispositivos, sejam capazes de compreender e julgar uma situação a fim de tomar uma decisão, assim como em um espaço inteligente.

Vale ressaltar que, este trabalho está inserido no projeto de pesquisa financiado pela Fundação de Amparo à Pesquisa e Inovação do Espírito Santo (FAPES) sob o nº 0483/2015, cujo título é “Construção da Infraestrutura de um Espaço Inteligente para Posicionamento e Controle de Dispositivos Robóticos”, coordenado pela Profa. Dra. Mariana Rampinelli Fernandes, orientadora deste projeto de graduação.

1.1 Objetivos

Este trabalho tem como objetivo desenvolver uma infraestrutura para um espaço inteligente baseado em visão computacional e em Internet das coisas. A infraestrutura deve

prover suporte para o desenvolvimento de aplicações em diversos temas, tais como visão computacional, reconhecimento de padrões e robótica móvel. Essa infraestrutura deve ser flexível de tal forma que seja possível substituir os equipamentos por outros de marcas e/ou modelos diferentes, ou ainda, incorporar novos dispositivos, sem que grandes alterações na infraestrutura sejam necessárias. Isso é possível a partir da padronização das diferentes camadas da arquitetura, de forma que alterações em uma camada não interfira em outra.

Como o foco inicial para a utilização dessa infraestrutura será o desenvolvimento de aplicações envolvendo o controle de robôs móveis através de realimentação visual, existem alguns requisitos que a infraestrutura deverá atender. Um deles é a latência máxima na entrega de imagens para a aplicação que não deve ultrapassar 100 ms. Isso porque o robô utilizado possui odometria com taxa de amostragem máxima com período igual a 100 ms. Além disso, em aplicações multicâmeras, geralmente existe a necessidade de que a captura das imagens seja sincronizada.

1.1.1 Objetivos específicos

- Levantamento dos requisitos necessários para o espaço inteligente.
- Padronização da arquitetura baseada em IoT para a infraestrutura e especificação de seus equipamentos.
- Escolha do padrão de comunicação utilizado na infraestrutura e implementação de suas bibliotecas.
- Implementação e testes de *drivers* para os dispositivos do espaço inteligente.
- Desenvolvimento de serviços de suporte à infraestrutura e testes.
- Implementação e testes de APIs para a construção de aplicações.

1.2 Estrutura do Texto

A parte escrita deste Projeto de Graduação está dividida em cinco capítulos, descritos a seguir:

- **Introdução:** capítulo inicial que tem como objetivo contextualizar o trabalho desenvolvido, apresentando alguns trabalhos relevantes já desenvolvidos na área deste projeto. Além disso, apresenta brevemente o objetivo deste projeto de graduação.

- **Referencial Teórico:** nesse capítulo será apresentado um breve Estado da Arte sobre os temas espaços inteligentes e Internet das coisas, bem como uma visão geral dos principais conceitos abordados no desenvolvimento do trabalho.
- **Infraestrutura Proposta:** aqui será detalhada a implementação realizada, compreendendo todos os níveis de implementação.
- **Experimentos e Resultados:** após a apresentação do desenvolvimento da infraestrutura, serão apresentados alguns experimentos realizados utilizando-a, para validar o que foi proposto.
- **Conclusões e Trabalhos Futuros:** esse capítulo apresenta uma breve discussão geral do trabalho desenvolvido, bem como uma lista de trabalhos futuros que se deseja desenvolver para melhorar a infraestrutura.

2 REFERENCIAL TEÓRICO

2.1 Espaços Inteligentes

Para que um ambiente seja classificado como inteligente, deve possuir um sistema capaz de observar o meio através de uma rede de sensores e tomar decisões de acordo com as necessidades do usuário através de atuadores (RAMPINELLI, 2014).

As pesquisas nesse tema começaram a surgir no início da década de 90. Um dos primeiros trabalhos é o de Want, Falcao e Gibbons (1992), desenvolvido na Universidade de Cambridge. Este tinha como objetivo obter a localização de indivíduos em grandes instituições. Um dispositivo infravermelho emitia um sinal periodicamente, que era capturado por uma rede de sensores, para então determinar a localização das pessoas.

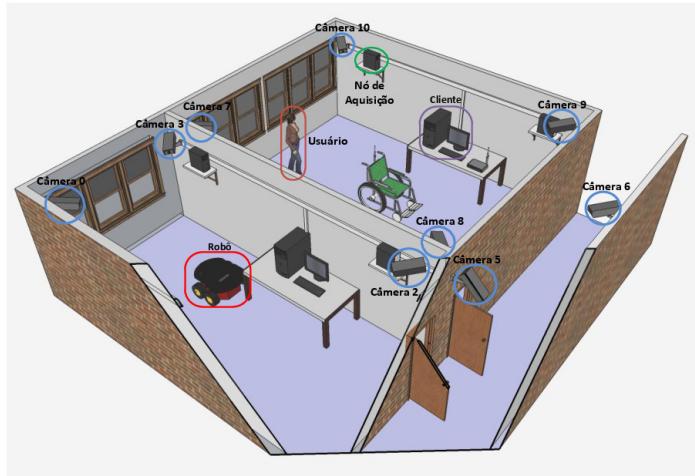
Câmeras ainda eram pouco presentes e exploradas em pesquisas nessa área devido ao elevado custo computacional que os algoritmos exigem e a capacidade de processamento disponível naquela época. Contudo, a partir da metade da década de 90, começaram a surgir trabalhos, como o desenvolvido por Pentland (1996), que utilizava câmeras para reconhecer gestos das mãos, e microfones para localizar usuários do ambiente através da voz. Outro trabalho que realizava o reconhecimento de gestos e falas é (BROOKS, 1997).

Diversos trabalhos sobre espaços inteligentes vem sendo desenvolvidos em vários lugares. Na Universidade de Alcalá (UAH), na Espanha, foi implementado um espaço inteligente onde foram desenvolvidos trabalhos como o de Pizarro et al. (2010), que tinha como objetivo encontrar a postura de um robô móvel utilizando a rede de câmeras instalada no ambiente inteligente da UAH, e o trabalho de Losada et al. (2010), que realizava a mesma tarefa, mas para múltiplos robôs móveis. Esses trabalhos serviram de inspiração para a tese desenvolvida por Rampinelli (2014) na UFES.

A infraestrutura utilizada em (RAMPINELLI, 2014) fica distribuída em dois laboratórios e no corredor que os conecta. A infraestrutura é constituída por 11 câmeras instaladas, cada uma conectada a um nó de processamento, além de um robô móvel e um microcomputador que desempenha a função de cliente do sistema. A Figura 2 mostra uma representação 3D do ambiente descrito, na qual pode-se observar o posicionamento escolhido para a instalação das câmeras.

Os trabalhos desenvolvidos nesse espaço inteligente são direcionados aos temas de visão computacional e robótica móvel. Em (RAMPINELLI et al., 2014), foi apresentado o

Figura 2 – Esquema 3D do espaço inteligente instalado na UFES.



Fonte: Rampinelli (2014).

espaço inteligente da UFES e os principais resultados obtidos com o método de calibração automática proposto (RAMPINELLI, 2014) da rede de câmeras que o compõe. Esse método consiste em utilizar um robô equipado com um padrão de calibração e, a partir das imagens do padrão capturadas pelas câmeras e da odometria do robô, realizar a otimização dos parâmetros de calibração.

A partir do resultado da calibração obtida com o método proposto em (RAMPINELLI, 2014), foram realizados experimentos de localização e guiagem de um robô móvel (QUEIROZ et al., 2015) utilizando realimentação visual para controle do robô. Nesse trabalho, a posição do robô era determinada unicamente por visão computacional. Vale ressaltar que o controle foi realizado pelo espaço inteligente, que envia informações de velocidade linear e angular para que o robô alcançasse uma posição previamente definida. O trabalho foi desenvolvido utilizando um robô móvel, mas seria possível por exemplo, controlar uma cadeira de rodas motorizada, como proposto em (RAMPINELLI et al., 2012).

Nota-se que, em um espaço inteligente que possui câmeras instaladas, é possível desenvolver diferentes pesquisas de novas tecnologias que baseiam-se em imagens, com o intuito de auxiliar atividades dos usuários, prover uma melhor qualidade de vida para pessoas de qualquer idade, principalmente idosos, maior segurança, dentre outros benefícios que podem ser incorporados à infraestrutura do espaço inteligente.

Apesar dos recentes trabalhos desenvolvidos no espaço inteligente instalado na UFES, seu modelo está obsoleto em diferentes aspectos, como por exemplo, o uso de um computador para cada câmera. Além disso, o modo como o sistema para esse espaço inteligente foi

desenvolvido dificulta a inclusão de novos dispositivos à rede de sensores e atuadores. Nesse contexto, este trabalho propõe um novo modelo de espaço inteligente. Vale ressaltar que essa infraestrutura será instalada no Ifes (Instituto Federal do Espírito Santo), para que lá sejam iniciadas pesquisas nessa área, uma vez que a orientadora deste projeto é professora do Ifes e realiza suas pesquisas em parceria com a coorientadora, Profa. Dra. Raquel Frizera Vassallo da UFES.

2.2 Internet das coisas

O que conhecemos hoje por Internet das coisas foi primitivamente introduzido por Engels et al. (2002), como uma proposta de arquitetura que visava integrar o mundo físico com o mundo virtual, entregando para cada dispositivo ou coisa uma identificação única, nomeada de *GUIDe*. A função da *GUIDe* era de identificar as informações sobre o objeto em questão, que poderia ser requisitada através de um serviço chamado de ONS (do inglês, *Object Name Service*). Dessa forma, os objetos passaram a pertencer a um mundo virtual, sendo possível ter aplicações nas quais há a troca de informações entre eles.

Contudo, Internet das coisas não se restringe a apenas utilizar etiquetas de identificação em objetos (CASAGRAS, 2009), mas também compreende a forma de capturar, transferir, e compartilhar os dados, incorporando sensores e atuadores a uma grande rede, no caso a Internet. Para isso, busca-se primariamente que os sistemas possuam interoperabilidade, ou seja, dispositivos de naturezas diferentes devem ser capazes de trocar informações e utilizar serviços entre si (KILJANDER et al., 2014).

Outro aspecto relevante é a variedade de tipos de domínios de coisas que apresentam conectividade com a Internet, o que demanda requisitos diferentes para cada domínio. Além disso, interconectar dispositivos com pouca semelhança em uma mesma infraestrutura utilizando uma mesma arquitetura é um grande desafio para o projetista. Dessa forma, diversos trabalhos na área de IoT vem sendo realizados para domínios específicos de coisas.

Um exemplo de domínio são as aplicações envolvendo dispositivos robóticos. Atualmente, grande parte dos robôs são equipados com sensores e interfaces de comunicação. Uma vez inseridos em uma rede que os permita trocar informações com outros dispositivos inteligentes, esses robôs podem executar tarefas mais complexas, melhorando a sua interação com o ambiente e com os humanos (GRIECO et al., 2014). Outro exemplo de domínio que pode ser citado é o que compreende dados multimídia. Tendo em vista a grande variedade de mídias, como vídeos, imagens, *e-books*, e a grande variedade de dispositivos que são capazes de produzir e executar esses tipos de mídias, que estão inherentemente conectados à Internet, Alvi et al. (2015) propõe que as “coisas multimídia” podem interagir e cooperar

umas com as outras, com o intuito de melhorar o acesso a esses serviços para os usuários.

O conceito de Internet das coisas também pode ser aplicado a ambientes de entretenimento como museus, onde normalmente os visitantes recebem panfletos informativos, ou ainda podem ter a possibilidade de escutar áudios sobre alguma obra de arte. Em Alletto et al. (2016) foi proposta uma arquitetura baseada em IoT com o objetivo de localizar pessoas e coisas em ambientes internos, com o intuito de melhorar a experiência de usuários em museus, entregando conteúdo multimídia baseado nas obras de arte que o visitante está observando.

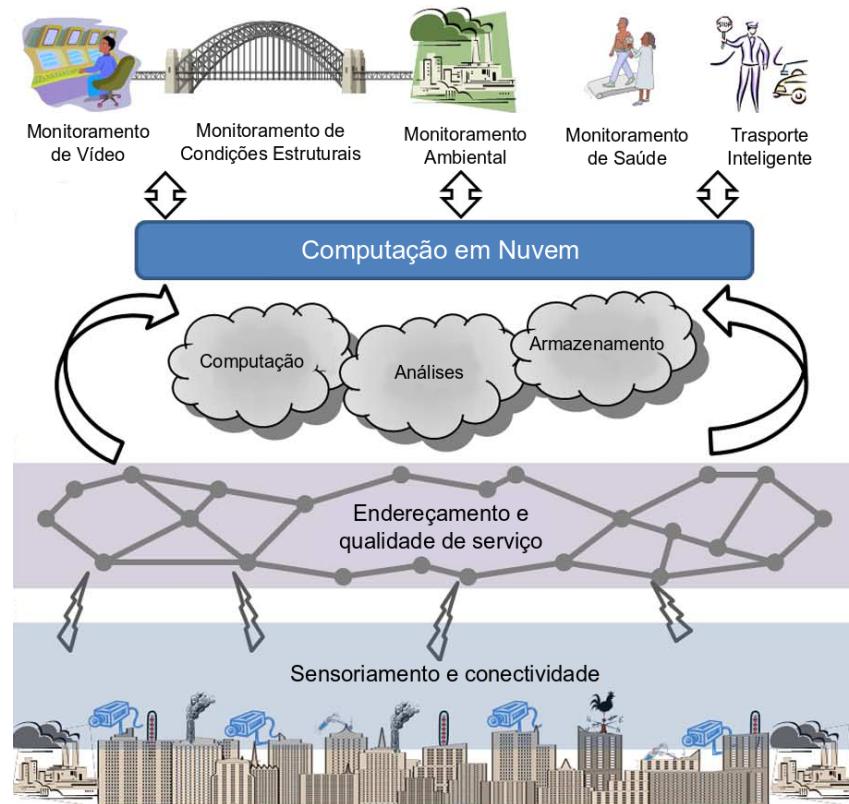
Trabalhos cujo escopo são cidades inteligentes também utilizam o conceito de IoT para modular sua infraestrutura e organização de dados. Por exemplo, em cidades onde é crescente a densidade populacional, cresce também a demanda por serviços e infraestrutura. Contudo, se cidadãos e gestores tiverem acesso à dados da cidade, esses podem ser utilizados para planejamentos futuros e tomadas de decisões. Desta maneira, Jin et al. (2014) propõe um *framework* para a realização de cidades inteligentes utilizando o conceito de IoT. Esse *framework* engloba um completo sistema de informações da cidade, desde a camada de sensoriamento até a de rede, bem como integração com serviço de armazenamento de dados na nuvem. Um diagrama que representa essa arquitetura descrita pode ser visto na Figura 3.

2.2.1 Arquitetura IoT

Arquiteturas de referências são desenvolvidas para servirem de ponto de partida para o desenvolvimento de arquiteturas para domínios específicos de uma determinada área. Talvez o mais conhecido na área de computação seja o modelo OSI (do inglês, *Open Systems Interconnection*), que serve de referência para o desenvolvimento de protocolos de comunicação para os mais diversos projetos. Modelos de referência facilitam o desenvolvimento de sistemas, reduzindo tempo e custo, além de serem guias para a evolução de sistemas já existentes. Com o crescimento da Internet das coisas, surge a necessidade de ter uma arquitetura de referência para o desenvolvimento de soluções nesse tema.

Uma arquitetura de referência para Internet das coisas foi proposta no projeto IoT-A. A especificação dessa arquitetura está descrita no livro “*Enable Things to Talk - Designing IoT solutions with the IoT Architectural Reference Model*” (BASSI et al., 2013). Esse modelo de referência é composto por um conjunto de subdivisões que visam facilitar o desenvolvimento de sistemas. Dessa forma, soluções baseadas em IoT podem ser implementadas e mantidas mais rapidamente. É a partir de uma arquitetura de referência e nos requisitos do domínio no qual os dispositivos pertencem que se idealiza uma arquitetura. Em (BELLAGENTE et al., 2015), por exemplo, é proposta uma arquitetura IoT para o gerenciamento de

Figura 3 – Representação de uma arquitetura de Internet das coisas em uma cidade inteligente.



Fonte: Adaptado de Jin et al. (2014).

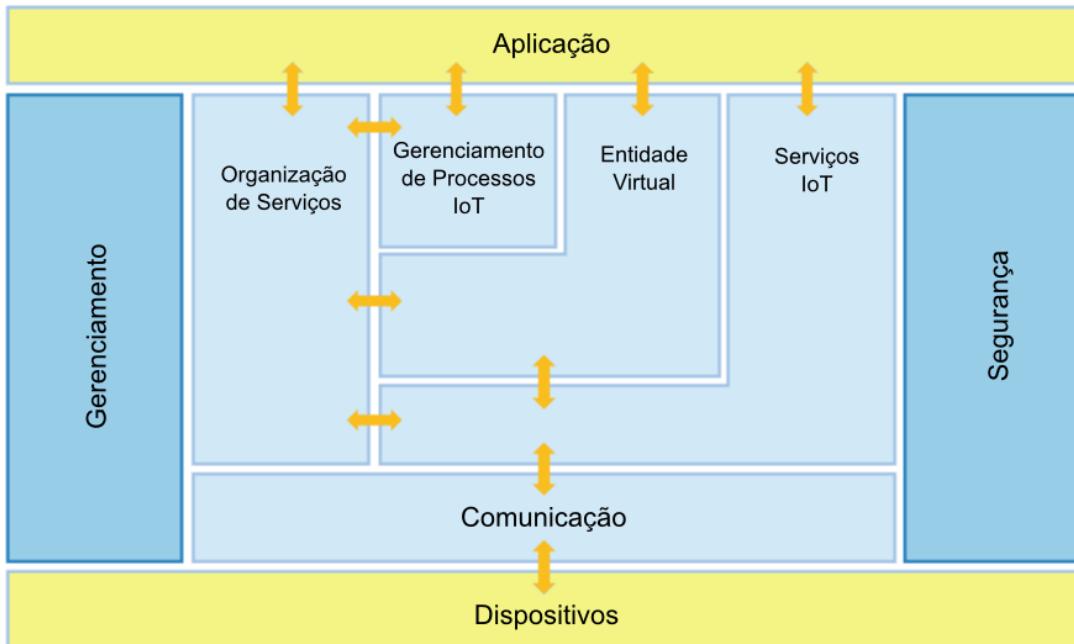
energia em construções inteligentes, que baseou-se na arquitetura de referência IoT-A. Essa arquitetura será a referência adotada para o desenvolvimento deste trabalho.

Para facilitar o desenvolvimento de sistemas baseados na arquitetura de referência IoT-A, foi proposto também em (BASSI et al., 2013) um modelo funcional, composto por grupos de funcionalidades. O modelo pode ser verificado na Figura 4. Vale ressaltar que, na Figura 4, os grupos funcionais destacados em amarelo, Aplicação e Dispositivos, está fora do escopo da arquitetura de referência, enquanto os grupos de Gerenciamento e Segurança são transversais aos demais (PIRES et al., 2015).

Nesse modelo, alguns grupos funcionais merecem destaque. O grupo de Organização de Serviços é responsável pela comunicação entre outros grupos funcionais. Além disso, como a arquitetura IoT-A é baseada em serviços, esse grupo é também responsável por compor, orquestrar e coreografar serviços. A coreografia de serviços tem o papel de mediar a comunicação entre os serviços, além de garantir que um cliente tenha sua requisição de utilização de um serviço atendida, mesmo que este não esteja prontamente disponível.

Vale mencionar também o grupo Entidade Virtual. Entidades virtuais são representações

Figura 4 – Modelo funcional da arquitetura de referência IoT-A.



Fonte: Adaptado de Bassi et al. (2013).

de entidades físicas no mundo digital. No contexto de IoT, entidades virtuais representam uma única entidade física. Contudo, pode-se ter várias entidades virtuais que representam a mesma entidade física. Cada entidade deve ser dotada de um identificador único. Ela pode ser classificada como ativa, quando refere-se a aplicações, agentes ou serviços que podem utilizar outros serviços ou recursos de outras entidades; bem como passiva, quando representa elementos de *software*. Os serviços associados a entidades virtuais permitem alterações de seu estados e também a leitura e escrita de atributos.

A camada de comunicação requer maior detalhamento pela sua importância na implementação da arquitetura do espaço inteligente desenvolvido neste projeto.

2.2.1.1 Comunicação

A camada de comunicação é responsável por prover a conectividade entre os dispositivos do espaço inteligente. Existem diversos protocolos, baseados em diferentes padrões, que são utilizados para a implementação de plataformas IoT. Um exemplo de padrão é o *request/response*, utilizado no protocolo HTTP (do inglês, *Hypertext Transfer Protocol*). Nesse padrão, existe um servidor que fica esperando uma requisição para então emitir uma resposta. Atualmente, existem microcontroladores com interfaces de comunicação com a Internet, que são capazes de suportar um servidor HTTP. Dessa forma, pode-se por exemplo enviar uma requisição para realizar a leitura dos dados de um sensor.

Existem ainda protocolos conhecidos como *m2m* (do inglês, *machine-to-machine*), que possibilitam a comunicação direta de dois dispositivos através de uma infraestrutura de rede. Um exemplo é o DDS (do inglês, *Data Distribution Service*), que vem sendo utilizado com frequência em aplicações industriais (PENIAK; FRANEKOVA, 2015). Esse protocolo atende requisitos de aplicações de tempo real, além de ser escalável e interoperável com outros sistemas.

Há também os protocolos baseados em mensagens. Esses protocolos normalmente trabalham associados à um *broker*, que tem a função de receber e encaminhar as mensagens para seus respectivos destinos. Dessa forma, para um dispositivo se comunicar com outro qualquer, basta saber o endereço do *broker* e uma identificação do destinatário da mensagem. Esse tipo de comunicação pode trabalhar com o padrão *publisher/subscriber* (*pub/sub*), no qual os *publishers*, quem publicam as mensagens, não sabem especificamente os destinatários, e os *subscribers*, os que recebem as mensagens, escolhem de quem receberão as mensagens.

O padrão *pub/sub* normalmente trabalha com o conceito de filas, que é para onde as mensagens publicadas são enviadas e então retiradas por quem desejar. Existem diversos protocolos que se baseiam nesse padrão. Nesse trabalho, serão destacados dois que estão sendo muito utilizados: AMQP (do inglês, *Advanced Message Queuing Protocol*) e o MQTT (do inglês, *Message Queuing Telemetry Transport*). Esses dois protocolos têm suas semelhanças, e sua escolha depende do dispositivo que se comunicará com o *broker*. Por exemplo, o MQTT possui implementações com o foco em microcontroladores com pouca memória, o MQTT-SN (do inglês, *MQTT for Sensor Networks*) é uma delas. Além disso, ao comparar o cabeçalho dos dois, nota-se que MQTT possui apenas 2 *bytes*, enquanto o AMQP possui no mínimo 8 *bytes*.

Existe uma grande lista de *brokers* que suportam os protocolos supracitados. Alguns só possuem suporte MQTT, outros apenas para AMQP, e existem os que suportam ambos. A escolha deve ser baseada nos dispositivos que estarão inseridos na infraestrutura implementada, bem como nos recursos computacionais disponíveis. Para situações com dispositivos muito heterogêneos, com diferentes recursos computacionais, pode ser conveniente a escolha de mais de um protocolo de mensagem e, portanto, recomendável que se utilize um *broker* compatível com os protocolos adotados.

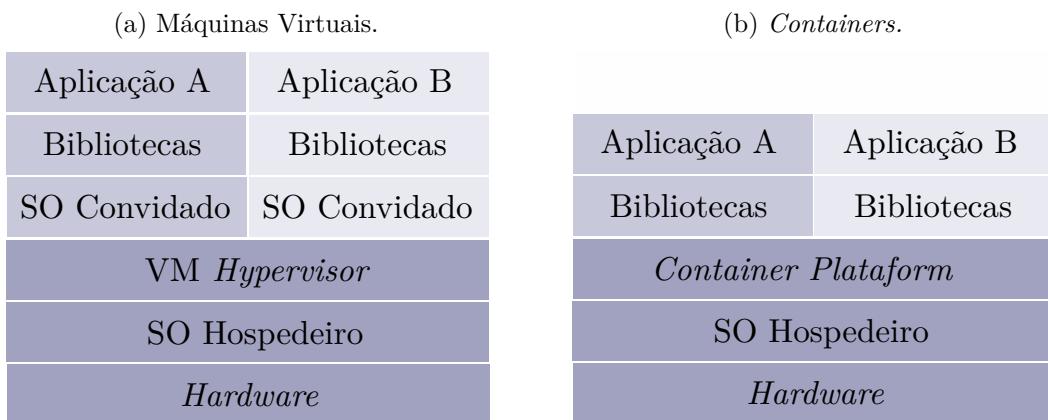
2.3 Virtualização e Computação em Nuvem

O conceito de virtualização compreende em criar uma representação de um processo físico baseado em *software* (JAIN; CHOUDHARY, 2016). Pode-se virtualizar aplicações, servidores, armazenamento e redes. Essa é uma estratégia eficaz de reduzir despesas de TI,

além de aumentar a agilidade em mudanças e expansões, uma vez que basta ter recurso computacional pra que seja possível instanciar novas virtualizações de diferentes tipos.

O uso de máquinas virtuais ou o uso de *containers* são duas maneiras de virtualização, contudo, atuam em camadas diferentes. O modelo que as máquinas virtuais utilizam tem como objetivo o compartilhamento de recursos de uma ou várias máquinas físicas, sendo que cada máquina virtual possui um sistema operacional que utiliza frações de memória e processador da máquina hospedeira, bem como os periféricos que também são virtualizados, como observado na Figura 5a. Os *containers* não possuem a visão de uma máquina completa, com um sistema operacional. Ele tem como objetivo executar apenas um processo, compartilhando o mesmo *kernel* com outros *containers*, como observado na Figura 5b. Contudo, isso é feito de uma maneira que isole a memória, processador, disco e outros recursos utilizados, fazendo com que o *container* pareça estar em um sistema dedicado. Os *containers* também podem ser executados dentro de uma máquina virtual que está hospedada em um *datacenter*.

Figura 5 – Estrutura de camadas para aplicações em máquinas virtuais e *containers*.



Fonte: Produção do próprio autor.

As tecnologias de virtualização foram um dos fatores que possibilitaram a realização do que hoje é conhecida como computação em nuvem, que tem como principal propósito oferecer serviços através da Internet. Dessa maneira, uma empresa não precisa manter um *datacenter* e, para uma expansão, basta contratar mais recursos com o provedor de serviços.

Nesse contexto, surgiram diferentes modelos de serviços oferecidos através de computação em nuvem. Os três principais modelos são:

- **SaaS** (do inglês, *Software as a Service*): nesse modelo, o usuário tem acesso a

aplicações gerenciadas por um provedor. O serviço é provisionado através da *web*, eliminando a necessidade de instalação do *software* no computador do usuário.

- **IaaS** (do inglês, *Infrastructure as a Service*): nesse modelo, o usuário pode alugar, gerenciar e monitorar recursos computacionais (máquinas virtuais, armazenamento, endereços IP, etc) presentes em um *datacenter*.
- **PaaS** (do inglês, *Platform as a Service*): nesse modelo, plataformas de computação (*frameworks*, banco de dados, etc) são oferecidas ao usuário na forma de um serviço. Esse é o modelo utilizado para o desenvolvimento deste trabalho.

O custo é um fator que influencia na contratação de serviços de computação em nuvem, visto que não há o investimento inicial em *hardware* e *software*, além da instalação, configuração e manutenção de *datacenters*. Outro aspecto relevante é a segurança oferecida por esse tipo de serviço. A computação em nuvem torna mais fácil e reduz os custos em operações de *backup* de dados. Além disso, os *datacenters* em nuvem estão em constante atualização de sua infraestrutura, dispensando ao cliente final essa tarefa.

Os diversos benefícios, como flexibilidade e escalabilidade, que a utilização de máquinas virtuais e *containers* associados à computação em nuvem trazem são interessantes para uma infraestrutura de um espaço inteligente baseado em IoT (CELESTI et al., 2016). É possível, por exemplo, serviços que precisam ser instanciados de acordo com a demanda. Ou ainda, serviços que estejam ociosos podem ser removidos temporariamente. Essa forma possibilita um melhor gerenciamento de recursos da infraestrutura. Outra vantagem é que utilizar virtualização torna desnecessária a configuração de um novo *hardware*, bem como a instalação de bibliotecas no sistema operacional, pois uma vez criada a imagem do *container* ou da máquina virtual, essa imagem só precisa ser instanciada.

2.4 Sincronismo

O sincronismo na captura de dados em sensores é um problema crítico em infraestruturas distribuídas. Cada aplicação possui um erro máximo de sincronismo aceitável, e esse erro está associado a um tempo de persistência (ELSON; ESTRIN, 2001; ROMER, 2005), ou seja, o tempo no qual o sincronismo permanece com o erro dentro do limite tolerável. Alguns exemplos nos quais encontramos esses requisitos são: estimativa de velocidades a partir da integração de séries temporais; realização da medida do tempo de propagação de uma onda sonora para determinar a localização de sua fonte; ou para rejeição de dados, ao reconhecer que descrevem detecções duplicadas do mesmo evento por diferentes sensores.

A maneira que o sincronismo é feita pode influenciar no consumo energético do dispositivo. Além do mais, cada método utilizado para o sincronismo possui um tempo de resposta diferente para que se atinja os parâmetros desejáveis. Portanto, existirão situações em que se deseja sincronizar dispositivos que permitem diferentes meios para isso, tornando essa tarefa não trivial e ocasionalmente com restrições.

2.4.1 Sincronismo de Câmeras

O problema de sincronismo de câmeras é mais antigo que as recentes aplicações de visão computacional com múltiplas câmeras. Na época do cinema mudo, esse era um problema na gravação de cenas com vários pontos de vista. De acordo com Richardson (1923), em uma captura sequencial para a reprodução de filme, qualquer dessincronização na aquisição das imagens, alteração na taxa de captura das câmeras ou na velocidade da reprodução, faria com que a reprodução do filme ficasse diferente comparada com a cena que realmente foi capturada, desde que tais diferenças sejam perceptíveis para o espectador.

Atualmente, o problema de sincronismo também é enfrentado em aplicações de visão computacional com duas ou mais câmeras, no qual são exigidas imagens sincronizadas em tempo real com erro menor que um milissegundo (LITOS; ZABULIS; TRIANTAFYLLOIDIS, 2006). Tal requisito é importante para não haver inconsistência entre os dados obtidos a partir das imagens, uma vez que, sem sincronismo, não há garantias de que as imagens corresponderão à mesma cena.

Vários métodos de sincronismo de câmeras podem ser encontrados na literatura. Tais métodos podem ser divididos em quatro categorias: sincronismo por *hardware*; sincronismo por pós-processamento; sincronismo por rede; e sincronismo por *software*.

O mais comum e com melhor resultado é o sincronismo por *hardware*. Este utiliza uma unidade controladora dedicada que emite sinais de disparo para a captura dos *frames* das câmeras conectadas ao sistema, desde que as câmeras possuam um entrada para esse propósito. Algumas câmeras com interface *FireWire*, quando conectadas ao mesmo barramento, oferecem recursos de sincronismo. Mesmo sendo o método com melhor desempenho, oferece pouca flexibilidade na disposição das câmeras, pois estas devem ter uma conexão física entre elas, ou por exemplo, estarem conectadas ao mesmo barramento para que seja possível o sincronismo. Em (LIU et al., 1997), é proposto um método de sincronismo para câmeras rotativas, no qual existe um microcomputador que realiza o controle do motor e gera os sinais de disparo para as câmeras.

Os métodos de sincronismo por pós-processamento utilizam algoritmos que estimam a diferença de tempo entre cada imagem capturada, para que então seja feita o sincronismo

da rede de câmeras. Yan et al. (2004) propõem um método para sincronizar vídeos que foram gravados em diferente pontos de vista, utilizando pontos de interesse associados a instantes de tempos. Após a extração de inúmeros pontos, é realizada a correlação entre eles para se encontrar a diferença de tempo entre cada vídeo, e então realizar a sincronia. Mesmo apresentando bons resultados, métodos que utilizam esse tipo de processamento dificilmente conseguem ser aplicados em tempo real, além de apresentarem problemas com vídeos que não possuam sobreposição de imagens.

Se as câmeras utilizadas estão conectadas a computadores que, por sua vez, estão conectados a uma rede local, é possível realizar o sincronismo utilizando o protocolo de rede NTP (do inglês, *Network Time Protocol*). A realização dessa técnica é simples, mas experimentos mostram que esse método não apresenta melhores resultados para o sincronismo de câmeras quando comparado com métodos baseados em *hardware* (LITOS; ZABULIS; TRIANTAFYLLODIS, 2006).

Por fim, o sincronismo feito por *software* pode ser implementado de diversas maneiras. Em (SVOBODA; HUG; GOOL, 2002), é utilizado um sistema capaz de inicializar todas as câmeras praticamente no mesmo instante. Entretanto, é desconsiderada a latência existente entre o envio do comando pelo computador até o momento em que ele é interpretado pela câmera. Essa latência está associada à rede *Ethernet* utilizada, ao sistema operacional e ao *driver* da câmera.

Independente do método de sincronismo adotado para um rede de câmeras em um espaço inteligente, é necessário conhecer as aplicações que se deseja desenvolver, para então se definir o erro máximo aceitável entre a captura das imagens. O método de sincronismo deve ser levado em consideração na especificação da câmera, pois assim, pode-se verificar se essa possui algum meio de disparo por *hardware* ou *software*, por exemplo. Neste trabalho optou-se por realizar um sincronismo por *software*.

3 INFRAESTRUTURA PROPOSTA

Neste capítulo, serão detalhados os equipamentos da infraestrutura física utilizados, a arquitetura IoT adotada, além da estrutura de comunicação escolhida. Também será descrito o funcionamento dos serviços implementados para o espaço inteligente, e apresentadas as interfaces de desenvolvimento implementadas.

3.1 Infraestrutura Física

O passo inicial foi a especificação das câmeras, computadores e infraestrutura de rede, visando futuras aplicações e expansão do espaço inteligente. Posteriormente esses equipamentos foram comprados. O laboratório no qual foi desenvolvido este projeto possui o robô móvel utilizado, dispensando a compra de um novo.

3.1.1 Câmeras

A conexão de uma câmera com um computador pode ser feita através de vários tipos de interfaces. Por exemplo, uma câmera com a saída analógica, pelo qual a imagem é transmitida continuamente através de um sinal elétrico, interpretado por uma placa de captura de imagem conectada ao computador. A qualidade do sinal da câmera e da placa de captura interferem diretamente na imagem resultante. Essas câmeras costumam ser mais simples e baratas, mas possuem limitações em resolução e taxa de captura, além de serem mais susceptíveis a ruídos elétricos. Atualmente, câmeras com interface analógica são comuns em videomonitoramento devido ao seu custo mais baixo.

Outra alternativa são as câmeras com interface de comunicação digital. Nesse caso, a câmera possui uma unidade de processamento embarcada para poder armazenar e transmitir a imagem. Com a interface digital, tornou-se possível transmitir imagens com resoluções e taxas maiores. Duas interfaces que merecem destaque são a *FireWire* e a *USB* (do inglês, *Universal Serial Bus*). Essas interfaces foram desenvolvidas com intuios diferentes: a *FireWire* é mais voltada para aplicações de alto desempenho e que necessitam de transmissão em tempo real, como por exemplo, áudio e imagem; já a interface *USB*, surgiu com o intuito de ser simples, barata, e voltada para múltiplas aplicações. Portanto, como as taxas atuais de transmissão atingidas pela interface *USB* ultrapassaram as da interface *FireWire*, esta última ficou obsoleta.

Existem ainda, câmeras com interface *Ethernet*, que podem ser conectadas diretamente a uma rede e acessadas por diferentes computadores, tornando-as mais flexíveis que as câmeras *USB*. Além disso, essas câmeras apresentam velocidades que se equiparam às da

USB. Por tais motivos, optou-se por câmeras *Ethernet* para esse projeto. Outro aspecto que tornam essas câmeras mais flexíveis é a possibilidade de alimentá-las através do próprio cabo UTP (do inglês, *Unshielded Twisted Pair*). Esse recurso é conhecido como PoE (do inglês, *Power Over Ethernet*).

Baseado nesses requisitos, a câmera escolhida foi a BFLY-PGE-09S2C-CS, mostrada na Figura 6, produzida pela empresa *Point Grey* (RESEARCH, 2015). Essa câmera possui interface de comunicação Ethernet Gigabit, resolução de 1288×728 pixels, podendo operar com até 30 FPS nessa resolução.

Figura 6 – Câmera Point Grey Blackfly modelo BFLY-PGE-09S2C-CS.



Fonte: Research (2015).

O fabricante da câmera fornece a SDK (do inglês, *Software Development Kit*) *FlyCapture2*, disponível para os sistemas operacionais *Windows* e *Linux*. Essa SDK consiste em um conjunto de bibliotecas implementadas em C/C++, que possibilitam a configuração dos parâmetros das câmeras, além de uma aplicação gráfica, que permite ajustar manualmente suas configurações, bem como ver os *frames* capturados. Escolheu-se por adquirir um conjunto de 4 câmeras, pois os trabalhos que deseja-se desenvolver com a infraestrutura têm o requisito de múltiplas vistas em um mesmo ambiente. Além disso, essa quantidade foi limitada pelo orçamento do projeto ao qual este trabalho está vinculado.

3.1.2 Equipamentos de Rede

Tendo como requisito as 4 câmeras serem Gigabit, além de serem PoE, foi necessário especificar um *switch* que possuísse no mínimo 4 portas com esses recursos. Foi então escolhido o *switch* SG100D-08P, mostrado na Figura 7, fabricado pela Cisco (Cisco Systems, 2015). Esse equipamento possui 8 portas Gigabit, sendo 4 delas PoE, que podem fornecer até 33,6 W para cada dispositivo conectado. Essa potência é mais que o suficiente para as câmeras que possuem um consumo máximo de 2,5 W. Esse *switch* possui capacidade de chaveamento de 16 Gbps, o que é suficiente para o funcionamento das 4 câmeras simultaneamente.

Figura 7 – *Switch* modelo SG100D-08P.



Fonte: Cisco Systems (2015).

Além de um *switch*, foi especificado também um roteador com interface *wireless* para que dispositivos sem fio, como por exemplo o robô, possam ser conectados ao espaço inteligente. O modelo escolhido foi o EA2700, mostrado na Figura 8, produzido pela Linksys (LINKSYS, 2013). Ele possui 4 portas Gigabit na interface LAN e uma porta Gigabit na interface WAN.

Figura 8 – Roteador modelo EA2700.



Fonte: Linksys (2013).

3.1.3 Recursos Computacionais

Para o desenvolvimento da infraestrutura foi adquirido um microcomputador fabricado pela Dell, modelo OptiPlex 9020 (Dell Computadores do Brasil Ltda., 2015), equipado com processador Intel Core i5-4590 de 3,30 GHz e 8 GB de memória RAM, mostrado na Figura 9. Esse computador já possui por padrão uma interface de rede Gigabit. Contudo, foi adquirida mais uma placa de rede também Gigabit, para que tivesse uma rede dedicada para os equipamentos da infraestrutura, evitando assim que tráfegos provenientes de outras fontes interferissem no funcionamento do espaço inteligente.

Além disso, já existia no laboratório um servidor também da Dell, modelo PowerEdge T410, equipado com dois processador Xeon E5504 de 2,00 Ghz e 6 GB de memória RAM,

Figura 9 – Microcomputador Dell modelo OptiPlex 9020.



Fonte: Dell Computadores do Brasil Ltda. (2015).

além de duas interfaces de rede Gigabit. Esse servidor foi utilizado para realizar diversos testes da infraestrutura.

3.1.4 Bancada

A fim de se facilitar a fase de desenvolvimento, os equipamentos foram instalados em uma bancada que pode ser vista na Figura 10. Nessa bancada, foram instaladas as 4 câmeras, o *switch* e o roteador.

Figura 10 – Bancada montada com os equipamentos para a fase de desenvolvimento.



Fonte: Produção do próprio autor.

3.1.5 Robô Móvel

O robô móvel utilizado foi o *Pioneer 3 AT* (P3-AT), apresentado na Figura 11. Esse robô é produzido pela empresa Adept MobileRobots (MOBILEROBOTS, 2011), que fornece o SDK ARIA (do inglês, *Advanced Robot Interface for Applications*). Tal SDK implementa uma interface que permite enviar comandos para o robô e ler informações de seus sensores.

Figura 11 – Robô Móvel *Pioneer P3-AT*.

Fonte: Adaptado de MobileRobots (2011).

3.1.6 Minicomputador

O robô móvel *Pioneer 3 AT* não possui computador de bordo, nem interface de comunicação sem fio. Para que o robô comunique-se com a infraestrutura do espaço inteligente, foi adicionado um minicomputador modelo *Raspberry Pi 2 Model B*, que possui processador ARM Cortex-A7 quad-core de 900 MHz e 1 GB de memória RAM. Além disso, ele possui 4 portas USB 2.0, onde foram conectados um adaptador *wireless* da TP-Link modelo TL-WN823N, bem como um conversor USB-Serial para estabelecer comunicação com o robô. A *Raspberry* com o adaptador e o conversor podem ser vistos na Figura 12.

Figura 12 – Minicomputador Raspberry Pi 2 Model B com adaptador *wireless* e conversor USB-Serial.

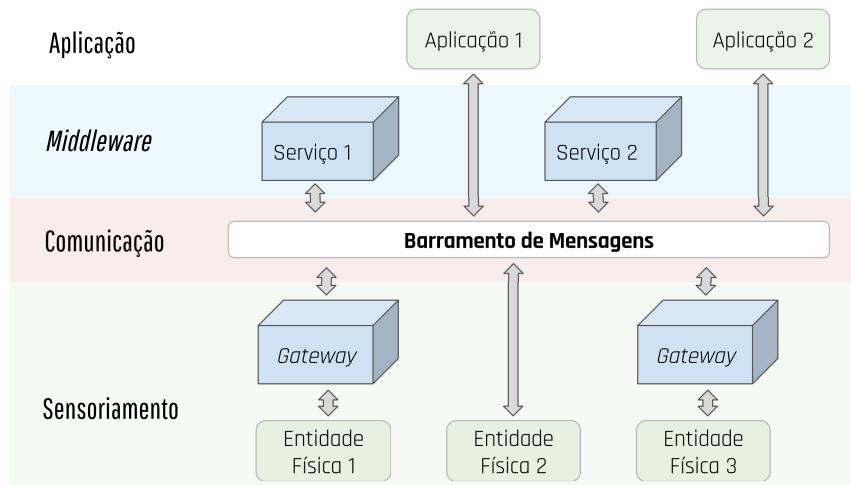
Fonte: Produção do próprio autor.

3.2 Arquitetura IoT

De acordo com o que foi estudado sobre arquiteturas para IoT, e baseado nos dispositivos presentes no espaço inteligente para o qual se propõe essa arquitetura, chegou-se no modelo de camadas simplificado apresentado na Figura 13, que será utilizado para subdividir as etapas de desenvolvimento dessa infraestrutura.

Na Figura 13, a camada de sensoriamento é responsável por expor os recursos do mundo físico para o mundo virtual. Além disso, essa camada deve simplificar a comunicação entre equipamentos heterogêneos através de uma padronização. A camada de comunicação é responsável por transportar os dados entre a camada de sensoriamento e as camadas superiores. Na camada de *middleware*, estão os diversos serviços oferecidos às aplicações, podendo ser do tipo que oferecem suporte da própria infraestrutura, ou ainda serviços específicos de processamento. A camada de aplicação provê uma interface para o desenvolvimento das aplicações do espaço inteligente.

Figura 13 – Representação simplificada em camadas da arquitetura proposta.



Fonte: Produção do próprio autor.

A separação por camadas é fundamental na fase de implementação e manutenção, pois, dessa maneira, definindo-se os padrões de como cada camada se relaciona com a outra, uma alteração em alguma delas, seja por uma atualização ou correção de problemas, não afetará o funcionamento global da arquitetura. A padronização também é fundamental para garantir um dos requisitos de arquiteturas IoT, que é prover suporte à dispositivos heterogêneos.

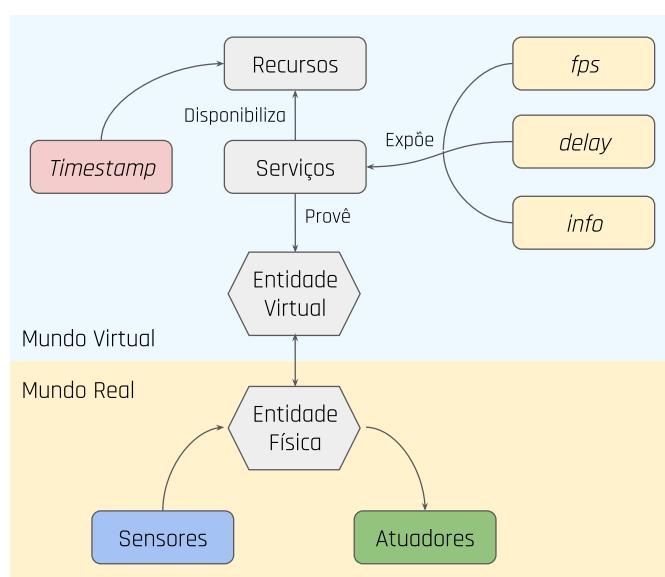
Em (CARMO et al., 2016) foi apresentado um modelo ontológico conceitual para esse espaço inteligente, onde é mostrada a relação entre cada objeto básico. Com base nesse modelo, e visando os requisitos que o ambiente em questão deveria ter, para a fase

de implementação foram definidos vários aspectos da arquitetura. Por exemplo, alguns detalhes sobre as entidades virtuais foram fixados. Portanto, toda entidade virtual presente nessa arquitetura deve possuir, quando possível:

- o recurso de *timestamp*, que indica o instante no qual algum dado do dispositivo foi capturado, ou ainda, o instante em que um dado foi inserido em um banco de dados;
- o serviço de *fps*, para que possa alterar a taxa de aquisição de dados;
- o serviço de *delay*, que deverá aplicar um atraso no período de amostragem de no máximo um período, logo que o serviço for requisitado;
- o serviço de *info*, que deverá fornecer uma estrutura de dados com a identificação da entidade virtual na infraestrutura, bem como os seus recursos e serviços oferecidos.

Com base nos requisitos mínimos que uma entidade deve ter, foi construído o modelo apresentado na Figura 14. Pode-se verificar que no mundo real, existem sensores e atuadores pertencentes à entidade física. Por outro lado, no mundo virtual existe a representação do mundo físico por meio da entidade virtual. Essa entidade deve prover serviços, e através desses, disponibilizar recursos. Além disso, deve expor serviços que sejam capazes de alterar parâmetros da entidade física. Vale ressaltar que, para essa implementação, foi feita uma definição de recursos diferente do proposto no modelo de referência IoT-A (BASSI et al., 2013). Aqui, recursos são dados provenientes dos sensores da entidade física, bem como o *timestamp*. Além disso, cada entidade possuirá um nome único, que é constituído pela classe à qual ela pertence, seguida de um ID único, da seguinte maneira: “classe.ID”.

Figura 14 – Representação das relações entre entidades física e virtual, com os serviços e recursos mínimos definidos.



Fonte: Produção do próprio autor.

Para este trabalho, não foi implementado um mecanismo de gerência e controle de acesso aos recursos das entidades, ou seja, pode haver concorrência na utilização de serviços que alterem parâmetros de alguma entidade que já está sendo utilizada por alguma aplicação. Isso afeta na utilização de recursos que podem ser alterados por diferentes aplicações, por exemplo, se uma aplicação utiliza as imagens de uma câmera com uma determinada taxa de aquisição e resolução, se uma segunda aplicação só poderá utilizar as imagens dessa mesma câmera com os mesmos parâmetros que a primeira aplicação.

3.2.1 Comunicação

Como protocolo de comunicação para a infraestrutura foi escolhido o AMQP (do inglês, *Advanced Message Queuing Protocol*). Esse é um protocolo de mensagens assíncrono, ou seja, depois que uma mensagem é enviada, não se espera uma resposta imediata. Uma vantagem de se utilizar um protocolo assíncrono é que o destinatário da mensagem não precisa estar disponível. Para utilizar esse protocolo, deve-se escolher também um *broker*.

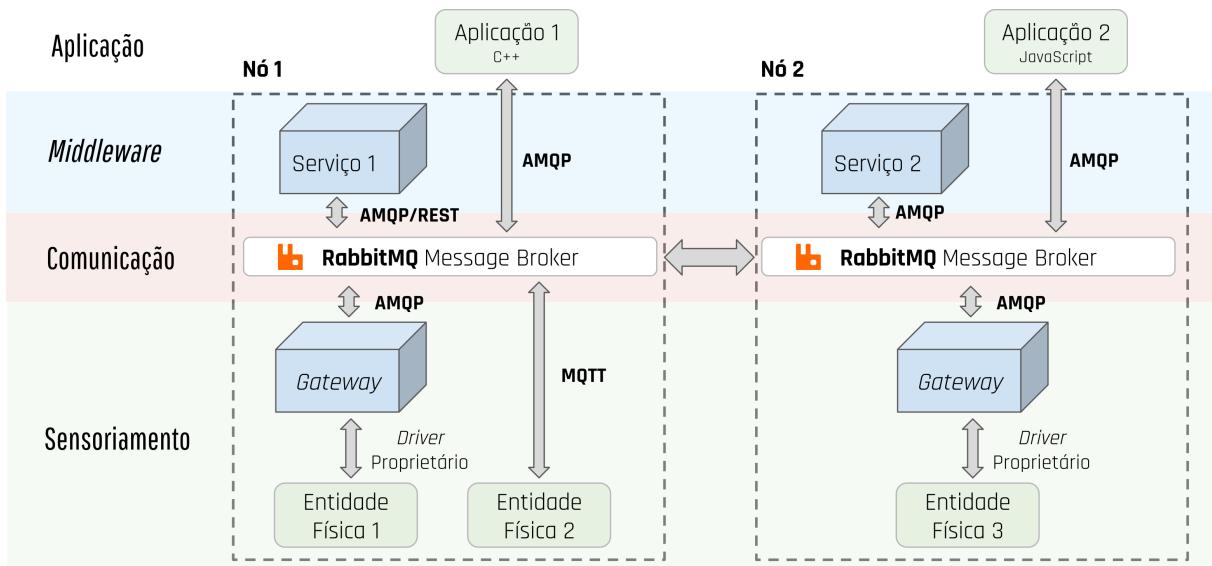
O *broker* é um elemento intermediário na comunicação que tem a função de receber e encaminhar mensagens. Essas mensagens podem ser modificadas antes de serem encaminhadas e, através do *broker*, é possível monitorar o fluxo de mensagens. Uma vantagem de utilizar uma infraestrutura com um *broker* é que as aplicações só precisam saber o seu endereço para se comunicar com outras aplicações. Além disso, ele oferece a possibilidade de persistir mensagens, ou seja, se o destinatário não estiver disponível, a mensagem fica armazenada até que este fique disponível. Uma desvantagem de se ter um elemento centralizador é a convergência do fluxo de mensagens para um único ponto. Contudo, é possível possuir diversos *brokers* em pontos diferentes da rede, o que ajuda a minimizar esse problema.

Um dos motivos para a escolha do AMQP foi a sua interoperabilidade. Ele possui implementações em diversas linguagens como Java, Phyton, PHP, JavaScript, C/C++, entre outras. Além disso, possui toda sua especificação aberta, o que facilita a solução de eventuais problemas na fase de desenvolvimento. Para utilizar o AMQP, deve-se escolher uma de suas inúmeras implementações. Usualmente utiliza-se implementações do protocolo associadas a um *broker*. Para o AMQP, existem várias possibilidades de *broker*: Apache ActiveMQ, FFMQ, HornetQ, RabbitMQ entre outros. Basicamente, todos eles possuem as características que se espera de um *broker*, como escalabilidade e robustez. Para essa implementação foi escolhido o RabbitMQ. Um dos motivos dessa escolha é a presença da implementação do cliente AMQP em diversas linguagens, como a escolhida para o desenvolvimento da maior parte deste trabalho, C++. Além disso, o RabbitMQ possui extensões para operar com outros protocolos como o MQTT.

A Figura 15 mostra a representação em camadas com detalhamento da comunicação.

Nessa representação, há dois *brokers* que se comunicam entre si, no entanto pode-se estender para um número maior de *brokers*. Dessa maneira, aplicações conectadas a um *broker* podem acessar entidades que estão conectadas a outro *broker*. Também estão representadas entidades físicas que conversam diretamente com o *broker* através do protocolo MQTT, como a entidade 2, e as entidades 1 e 3, que precisam de um *gateway* para falar AMQP. Também estão representadas duas aplicações, implementadas em linguagens de programação diferentes (C++ e JavaScript), que se comunicam cada uma com um *broker* utilizando AMQP.

Figura 15 – Representação em camadas da arquitetura proposta com detalhes da comunicação.

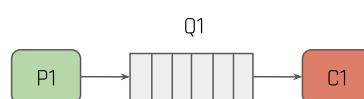


Fonte: Produção do próprio autor.

3.2.1.1 Detalhamento interno do *broker*

Como comentado anteriormente, a função do *broker* é receber e encaminhar as mensagens. Para isso, o RabbitMQ implementa filas internamente, onde essas mensagens ficam armazenadas para que sejam então encaminhadas ao destinatário. A Figura 16, mostra uma representação com um produtor, P1, uma fila, Q1 e um consumidor, C1. Essa fila pode ser configurada, por exemplo, para ter um tamanho fixo, ser persistente, dentre outros parâmetros. Se uma mensagem for entregue à fila e ninguém consumi-la, ela ficará lá até que alguém a consuma.

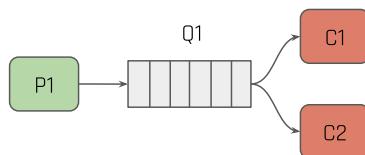
Figura 16 – Representação da comunicação entre um produtor e um consumidor através de uma fila.



Fonte: Produção do próprio autor.

Em certas situações, podem haver mais de um consumidor para uma mesma fila, como mostra a Figura 17. Nesse caso, quando uma mensagem chegar à fila, ela será entregue a um dos consumidores com o critério de alternância semelhante ao utilizado *round-robin*, i.e. cada consumidor recebe uma mensagem por vez em sequência e de forma cíclica. Esse modelo pode ser interessante para serviços de processamento, onde várias requisições são enviadas, e então o *broker* distribui as tarefas para cada consumidor disponível. Se houver mais tarefas que consumidores, essas ficam aguardando na fila.

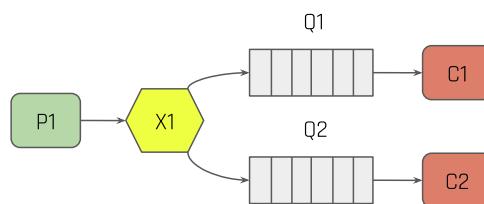
Figura 17 – Representação da comunicação entre um produtor e dois consumidor através de uma fila.



Fonte: Produção do próprio autor.

Contudo, pode-se desejar implementar um padrão *publisher/subscriber*, no qual existem vários consumidores recebendo simultaneamente o mesmo dado de um produtor, como representado na Figura 18. Para que isso funcione, o *broker* precisa de algo a mais do que filas. Surgem então os *exchanges*, que são posicionados antes das filas, e têm a função de encaminhar as mesmas mensagens para uma ou várias filas. A Figura 18 ilustra um exemplo do produtor P1 enviando uma mensagem que chega no *exchange* X1, que, por sua vez, decide para qual fila irá enviá-la, podendo assim, a mensagem chegar para C1, para C2, ou para ambos.

Figura 18 – Representação da comunicação entre um produtor e dois consumidor, com um *exchange* e duas filas.

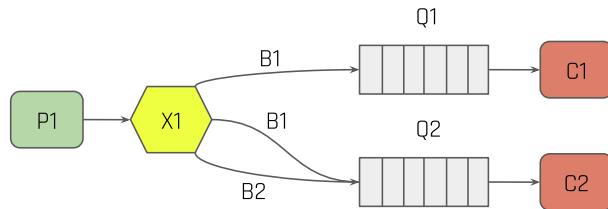


Fonte: Produção do próprio autor.

No entanto, é necessário que o *exchange* saiba para qual fila enviar. A relação entre a fila e o *exchange* é chamada de *binding*. Os *bindings* podem ser entendidos como regras de roteamento. Uma mesma fila pode ter mais de uma regra, ou ainda, diferentes filas podem possuir a mesma regra de roteamento, fazendo com que a mensagem seja replicada. Portanto, uma mensagem quando enviada deve possuir um *exchange* de destino, bem como sua identificação de roteamento, *routing_key*. Assim, o *broker* saberá para onde enviá-la. Por exemplo, na Figura 19, a fila Q2 possui dois *bindings*, B1 e B2 com o *exchange* X1,

bem como a fila Q1 possui um *binding* com X1. Dessa maneira, uma mensagem enviada ao *broker*, com o *exchange* X1 de destino e o *routing_key* B1, será enviada para as filas Q1 e Q2, e portanto, chegarão aos consumidores C1 e C2.

Figura 19 – Representação da comunicação entre um produtor e dois consumidor, com um *exchange*, duas filas e dois diferentes *bindings*.

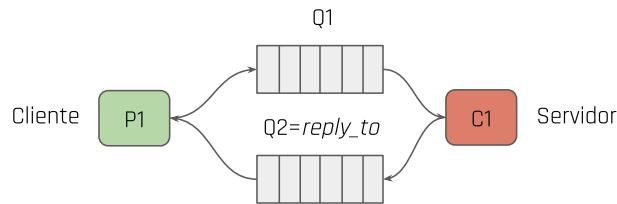


Fonte: Produção do próprio autor.

Existem vários tipos de *exchanges*. Até então, nos exemplos mencionados, eles eram do tipo *direct*, ou seja, a mensagem era encaminhada para uma fila apenas de acordo com o *binding*. Entretanto, se for necessário mais critérios para encaminhar as mensagens, um *exchange* do tipo *topic* deverá ser utilizado, por exemplo. As mensagens entregues para um *exchange* desse tipo devem possuir *routing_keys* construídos por uma lista de palavras separadas por pontos, por exemplo “operacoes.soma” e “operacoes.subtracao”. Normalmente, utiliza-se essa lógica para separar filas por tipos e subtipos. Além disso, pode-se utilizar os caracteres especiais “*” e “#” para realizar um grupo de *bindings*, por exemplo, se fizermos o *binding* “operacoes.#”, as mensagens com “operacoes.soma” e “operacoes.subtracao” serão enviadas para a mesma fila.

Além das configurações acima apresentadas, pode ser implementado um modelo baseado em RPC (do inglês, *Remote Procedure Call*), que é útil para a criação de serviços, onde envia-se uma mensagem com algum conteúdo que será processado, ou utilizado para alterar algum parâmetro da entidade, e uma resposta será enviada de volta para o remetente da mensagem. A Figura 20 mostra uma representação desse modelo, no qual foi omitido o *exchange* para simplificar a explicação. Nesse exemplo, o cliente C1 deve enviar o seu pedido para a fila que o servidor está escutando, Q1. Além disso, deve enviar o nome da fila para a qual a resposta será enviada, Q2. Essa fila usualmente não possui um nome conhecido, ficando a cargo do *broker* gerar um nome único para ela. O cliente fica então escutando essa fila, e quando a mensagem de resposta chega, é feita a correlação com a identificação colocada na mensagem de pedido.

Figura 20 – Representação da comunicação cliente/servidor de uma implementação RPC.



Fonte: Produção do próprio autor.

Como o protocolo de mensagens utilizado é assíncrono, se forem feitos vários pedidos pela mesma fonte, as respostas podem não chegar na ordem que os pedidos foram feitos. Para isso, junto da mensagem é enviado um ID (*correlation_ID*), para que seja feita a correlação com a resposta recebida. Tanto o *correlation_ID* quanto a fila na qual será recebida a resposta (*reply_to*) são inseridos em campos do cabeçalho do protocolo AMQP.

A Figura 21 ilustra um exemplo com duas entidades virtuais e duas aplicações. A entidade E1 possui uma fila Qe1 por onde recebe os pedidos para atendimento de serviços. Essa fila está conectada ao *exchange* X1 através do *binding* E1.serv1. O mesmo vale para a entidade E2, mas com fila de nome Qe2 e *binding* E2.serv1. A aplicação 1 envia ao *broker* pedidos de serviço para as entidades 1 e 2, que chegam ao *exchange* X1 e são então encaminhados para as filas de acordo com o *routing_key*. Além disso, nesse exemplo, as aplicações 1 e 2 estão consumindo um recurso da entidade 2. Para isso, cada aplicação cria no *broker* uma fila que se conecta ao *exchange* X2 através do *binding* referente ao recurso E2.res1. Vale ressaltar que, para o recurso chegar às aplicações, ele deve ser encaminhado ao *exchange* X2 com o *routing_key* E2.res1.

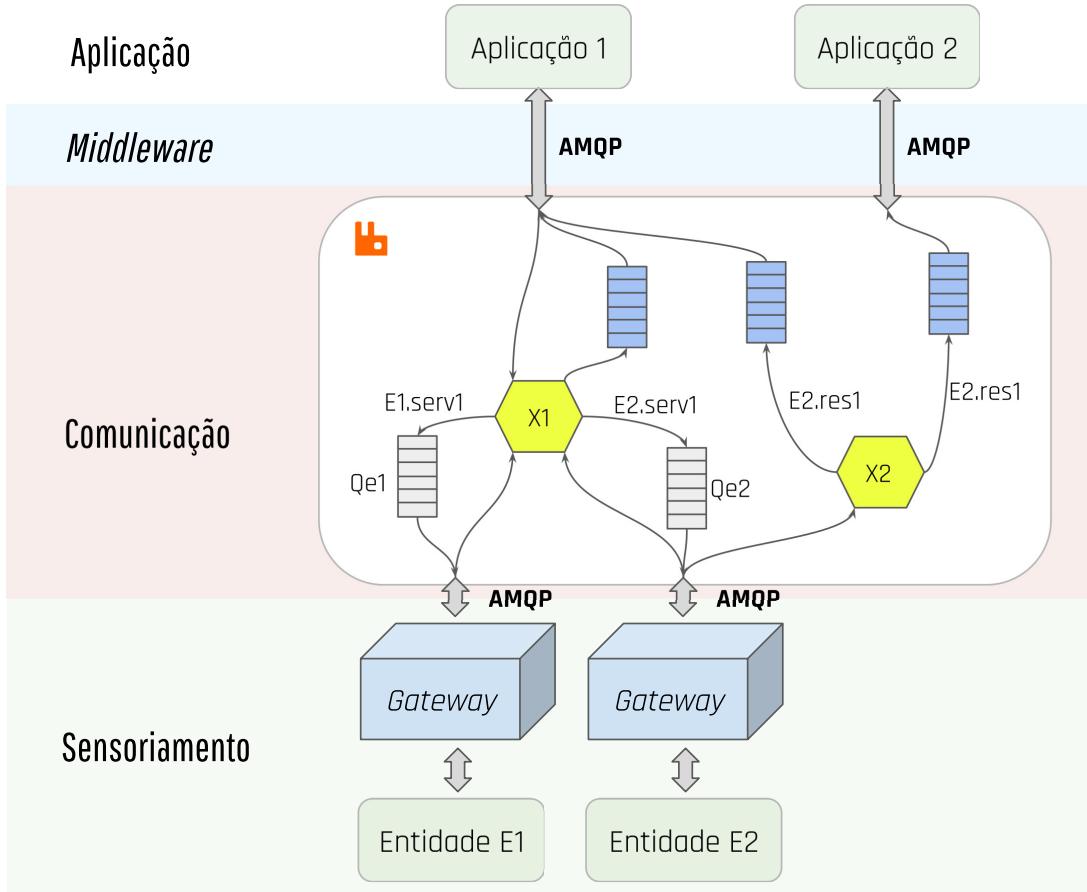
3.2.2 Sensoriamento

Nesta seção será discutida a implementação dos *drivers* para as câmeras e robô utilizados na infraestrutura. O objetivo é tornar o acesso aos recursos das entidades físicas mais fáceis, além de padronizá-los, para que, ao adicionar câmeras de outro fabricante, por exemplo, não haja mudanças nas outras fases de implementação.

3.2.2.1 Câmeras

Como apresentado na Seção 3.1.1, as câmeras especificadas e adquiridas para o projeto possuem uma SDK a FlyCapture2, que fornece um conjunto de funções para acesso às funcionalidades das câmeras. Essas câmeras possuem resolução máxima de 1288×728 pixels, que pode ser ajustada para qualquer valor múltiplo de 4 na horizontal e de 2 na vertical. Como esse ajuste é feito internamente na câmera, ao reduzir a resolução, diminui-se

Figura 21 – Representação em camadas da arquitetura proposta com detalhamento do funcionamento interno do *broker*, com exemplos do padrão *pub/sub* e implementação de serviços.



Fonte: Produção do próprio autor.

também a banda ocupada. Além disso, é possível definir o número de canais da imagem entregue, podendo ter 3 canais para o RGB ou 1 canal para a escala de cinza.

Existem várias formas de disparar a captura da imagem nessas câmeras. Pode-se enviar, por exemplo, um comando pela interface de rede para que seja feita a captura e então envia-se a imagem para o requisitante. É possível, ainda, realizar o disparo através de uma interface física proprietária, que possui pinos de entrada e saída, onde pode-se configurar um deles para receber um sinal de disparo. O terceiro método, e o mais usual, é ajustar uma taxa de captura fixa, e a cada um período, a aquisição de uma imagem será realizada e transmitida.

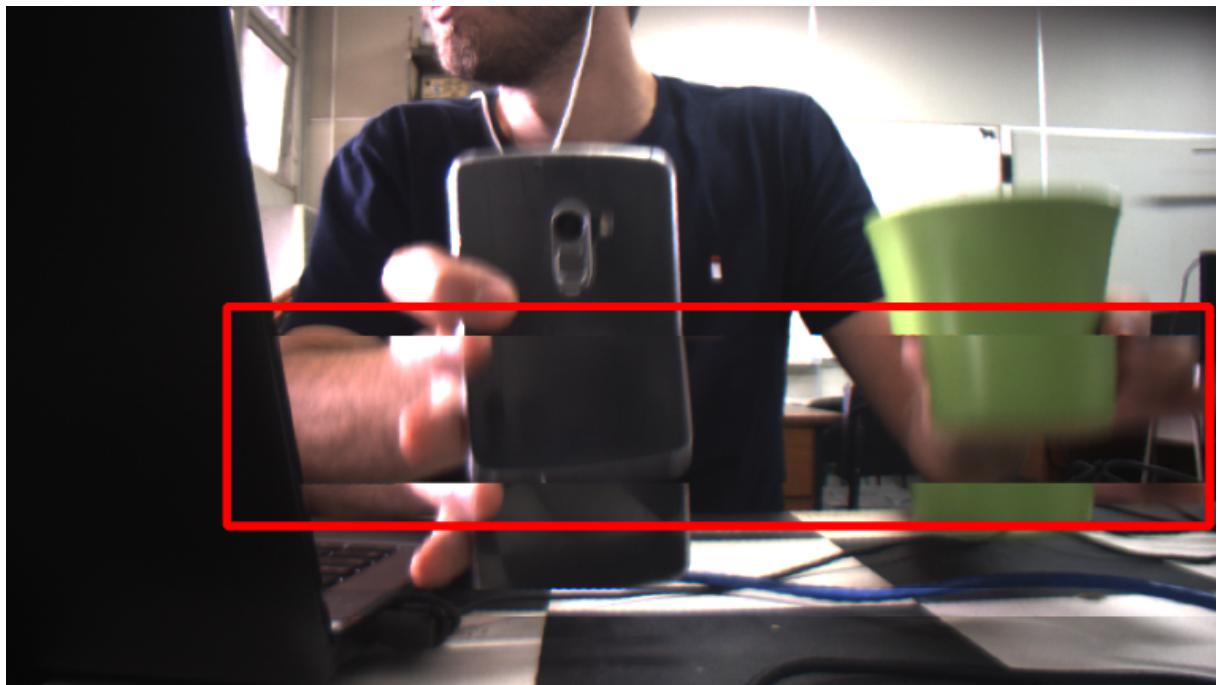
Por questões de flexibilidade, descartou-se a utilização de uma interface física para o disparo. A opção de enviar um comando através da rede também foi descartada, pois, se for necessária uma taxa de amostragem constante, esta precisaria ser gerada pelo sistema operacional, que não sendo de tempo real, não garantiria uma precisão. Portanto, optou-se por ajustar uma taxa de captura. Além da possibilidade do ajuste de uma taxa

de captura, é possível aplicar um atraso com duração de até um período. Esse recurso será posteriormente utilizado pelo serviço de *delay* exposto pela câmera, que é requisito para o serviço de sincronismo, descrito em uma seção posterior.

O recurso *timestamp* também é indispensável para os requisitos do serviço. Ao optar-se por uma captura de imagens com taxa constante, pode-se utilizar uma estrutura de *callback* para que, sempre que uma imagem tiver sua transmissão concluída, a função do *callback* é chamada. A primeira operação a se realizar nessa função é a captura do tempo do sistema operacional, esse será utilizado como o *timestamp* daquela imagem.

Como já mencionado, essas câmeras possuem interface Gigabit e todas foram conectadas a um *switch*, que por sua vez se conectam ao computador através de uma interface de rede Gigabit. Portanto, as 4 câmeras compartilham o mesmo meio físico para transmitir os dados até o computador. Se Cada câmera operar em sua configuração máxima, i.e. resolução de 1288×728 e 30 FPS, gera um tráfego de aproximadamente 27 MB/s (desconsiderando os cabeçalhos dos pacotes), totalizando cerca de 108 MB/s, o que está dentro do limite teórico de um canal Gigabit, 1000 Mb/s (125 MB/s). Entretanto, durante a realização de testes com as 4 câmeras em operação simultânea, foi detectada a aparição de fragmentos nas imagens, ocasionadas pela perda de pacotes, como pode ser visto na Figura 22. Nessa situação, a região na qual os pacotes foram perdidos é preenchida com a imagem anterior.

Figura 22 – Imagem de uma câmera com perda fragmentos devido à perda de pacotes.



Fonte: Produção do próprio autor.

O protocolo de transporte utilizado por essas câmeras é o UDP (do inglês, *User Datagram*

Protocol), sendo possível ajustar o tamanho dos pacotes e intervalo de tempo entre cada um deles. Os pacotes podem ter de 576 à 9000 *bytes*, e o intervalo pode variar de 0 à 6250 ciclos de *clock* da câmera. Por padrão de fábrica, o tamanho do pacote é configurado em 1500 *bytes*. Contudo, esse é o tamanho apenas do campo de dados do pacote que, com mais 86 *bytes* de cabeçalho, ultrapassa o tamanho do MTU (do inglês, *Maximum Transmission Unit*) configurado na interface de rede. Essa configuração padrão causava a fragmentação de pacotes, praticamente dobrando o número de pacotes na rede. Preferiu-se, então, reduzir o tamanho do campo de dados dos pacotes para 1400 *bytes*. Além disso, por padrão, o intervalo entre pacotes era configurado inicialmente com 400 ciclos. Em testes com as quatro câmeras em operação, observou-se que era possível aumentar o intervalo de pacotes para 6000 ciclos. Com esse ajuste, 1400 *bytes* e 6000 ciclos, foi possível operar com as 4 câmeras na máxima resolução, com a taxa de captura de 10 FPS. Uma possível solução para esse problema seria utilizar uma placa de rede, por exemplo, para cada duas câmeras, ou tentar utilizar um *switch* com interface 10 Gigabit, conectado ao computador, também com interface dessa velocidade.

Cada parâmetro da câmera descrito anteriormente, com exceção do tamanho do pacote e do intervalo entre pacotes utiliza a estrutura de comunicação explicada na Seção 3.2.1 para expor serviços, que possibilitarão as aplicações alterar esses parâmetros. Vale ressaltar, que o serviço de *info* também será exposto para as entidades do tipo câmera. Além disso, os recursos oferecidos pelas câmera, a imagem capturada por elas e o *timestamp* são publicados em seus respectivos tópicos.

A Figura 23 mostra a representação detalhada das camadas de comunicação e sensoriamento. O *gateway* da câmera está subdividido em três partes: interface proprietária, que corresponde às bibliotecas fornecidas pelo fabricante; interface padrão, que são as padronizações feitas que devem ser seguidas por qualquer câmera inserida à infraestrutura; e o cliente AMQP, responsável por fazer a interface com a camada de comunicação.

Cada entidade virtual do tipo câmera possui uma fila no *broker* com o seu nome, seguindo o padrão já apresentado: “camera.ID”. Como pode ser visto na camada de comunicação da Figura 23, a fila de nome “camera.0” possui os seguintes *bindings* ligados ao *exchange* de serviços:

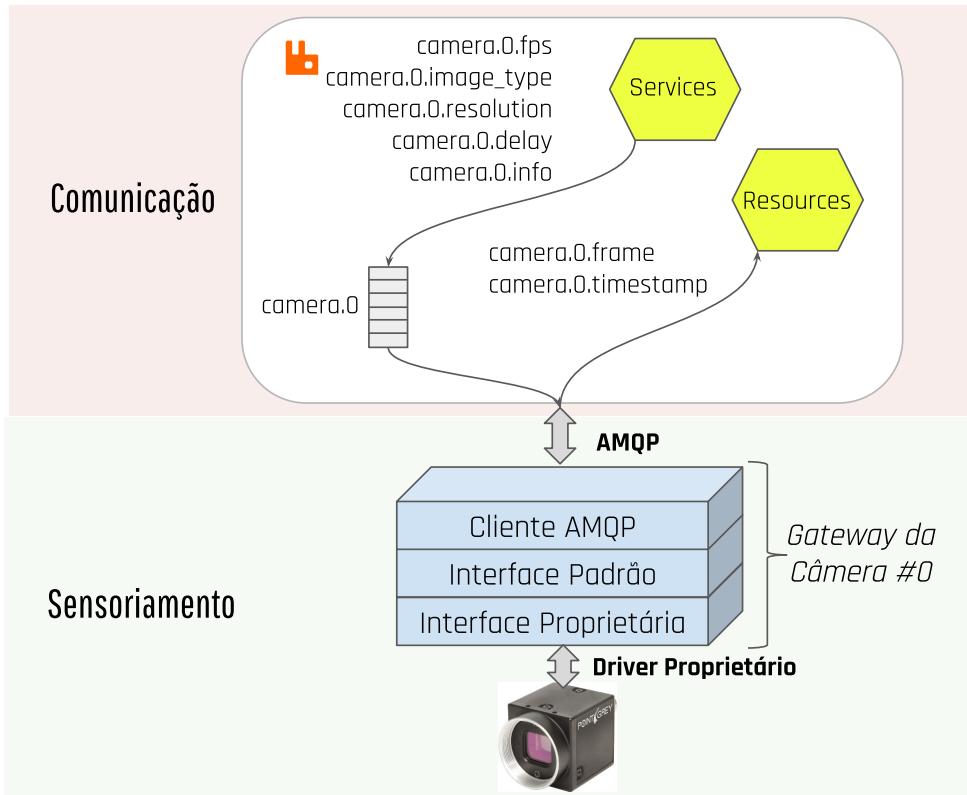
- FPS: “camera.0.fps”;
- Tipo de imagem: “camera.0.image_type”;
- Resolução: “camera.0.resolution”;
- *Delay*: “camera.0.delay”;

- Informação: “camera.0.info”.

Além disso, os recursos oferecidos por cada câmera são enviados ao *exchange* de recursos com os seguintes *routing_keys*.

- Image: “camera.0.frame”;
- Timestamp: “camera.0.timestamp”.

Figura 23 – Representação das camadas de comunicação e sensoriamento, com detalhamento para entidades do tipo câmera.



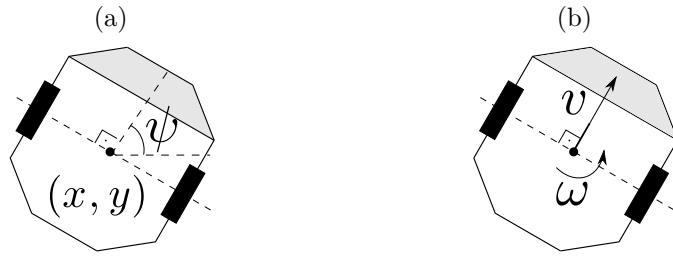
Fonte: Produção do próprio autor.

3.2.2.2 Robôs

O robô *Pioneer 3-AT* possui tração diferencial, e pode ser controlado por comandos de velocidades linear e angular dadas em mm/s e °/s, respectivamente, representadas na Figura 24b. Além disso, sua odometria pode ser lida em mm e orientação em *graus*, como apresentado na Figura 24a. Além de poder enviar comandos de velocidade para o robô, o espaço inteligente deve ser capaz de alterar sua posição atual. Por exemplo, em uma aplicação com dois ou mais robôs, um deles pode ser adotado como a referência do sistema de coordenadas, mas os outros devem ter sua pose definida em relação à referência.

Assim, como pode ser feito para as câmeras, para os robôs também é necessário mudar a taxa na qual os dados são publicados, no caso, odometria e *timestamp*. Para o robô em

Figura 24 – Representação das grandezas associadas ao robô *Pioneer 3 AT*: (a) odometria e (b) velocidades.



Fonte: Produção do próprio autor.

questão, o intervalo de amostragem não pode ser menor que 100 ms, pois corresponde à sua taxa de atualização interna. Como a SDK fornecida pelo fabricante não oferece a funcionalidade de receber os dados em um *callback* a uma taxa constante, deve-se fazer uma requisição a cada período de amostragem. A temporização foi feita utilizando o tempo do sistema operacional, bem como o efeito da aplicação de um atraso no período de amostragem.

Todos parâmetros do robô descritos anteriormente (velocidades, pose, intervalo de amostragem e atraso), são expostos como serviços da entidade virtual que representa o robô, utilizando a estrutura de comunicação apresentada na Seção 3.2.1. Os recursos oferecidos, odometria e *timestamp*, são publicados em seus respectivos tópicos.

Na Figura 25 está apresentado o detalhamento das camadas de comunicação e sensoriamento para a entidade do tipo robô. No *gateway* do robô, a camada de interface proprietária corresponde às bibliotecas fornecidas pelo fabricante. A interface padrão representa o conjunto de funções padronizadas para qualquer robô desse tipo inserido à infraestrutura. A camada referente ao cliente AMQP tem a função de fazer a interface com a camada comunicação.

Cada entidade virtual do tipo câmera possui uma fila no *broker* com o seu nome, seguindo o padrão já apresentado: “robot.ID”. Na camada de comunicação podemos verificar os *bindings* referentes a cada serviço exposto, que conectam o *exchange* de serviços à fila da entidade. Eles são listados a seguir:

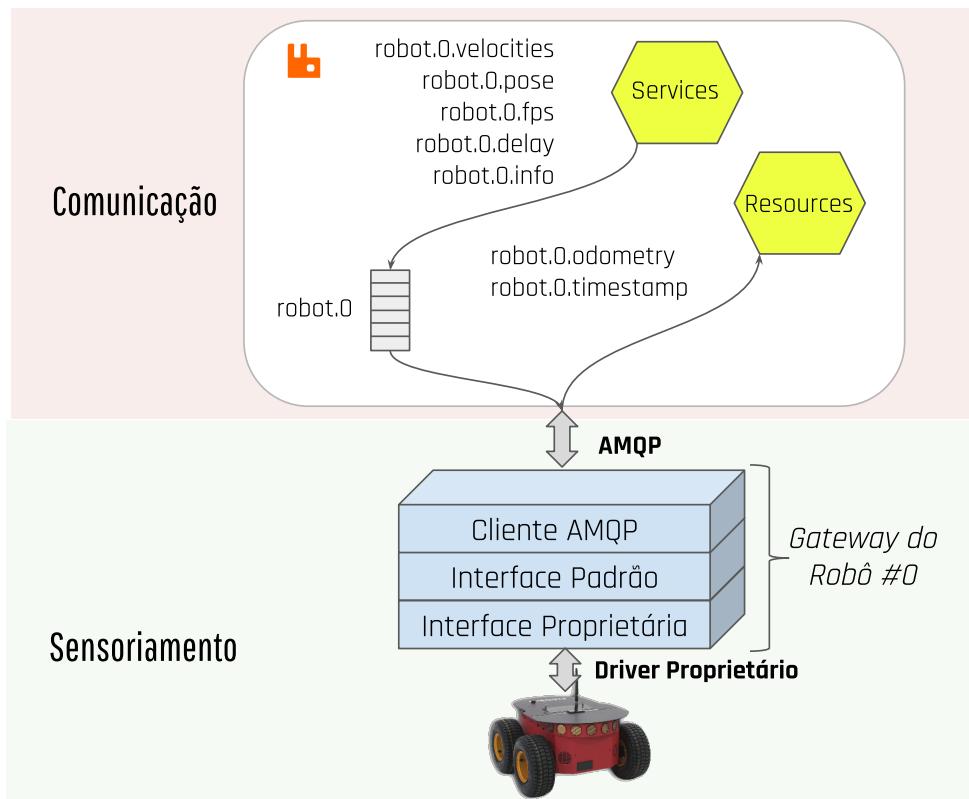
- Velocidades: “robot.0.velocities”;
- Pose: “robot.0.pose”;
- Taxa de amostragem: “robot.0.fps”;
- *Delay*: “robot.0.delay”;

- Informação: “robot.0.info”.

Além disso, os recursos oferecidos por cada robô são enviados ao *exchange* de recursos com os seguintes *routing_keys*.

- Odometria: “robot.0.odometry”;
- Timestamp: “robot.0.timestamp”.

Figura 25 – Representação das camadas de comunicação e sensoriamento, com detalhamento para entidades do tipo robô.



Fonte: Produção do próprio autor.

3.2.3 Serviços do Espaço Inteligente

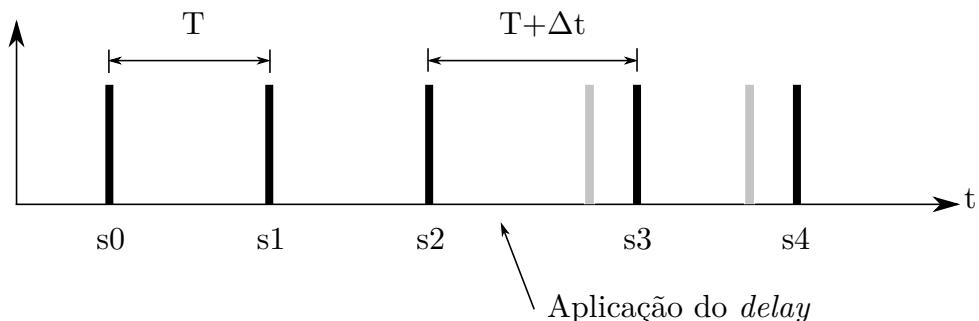
Como discutido anteriormente, na arquitetura proposta existem serviços que estão relacionados a entidades virtuais, e os que não estão associados. Esses serviços podem ser para dar suporte a aplicações, ou ainda, serviços de processamento utilizados por alguma aplicação. Nesse trabalho, foram desenvolvidos dois serviços de suporte: o serviço de sincronismo e o de busca de entidades. Além disso, parte de um trabalho que está sendo desenvolvido por um aluno de mestrado no qual este trabalho foi desenvolvido, foi transformado em um serviço do espaço inteligente.

3.2.3.1 Serviço de Sincronismo

Com já discutido na Seção 2.4.1, diversas aplicações envolvendo várias câmeras exigem o sincronismo na captura das imagens. Se a aplicação utiliza dados visuais junto da informação de outros sensores, pode ser que seja necessária a sincronia desses dados também. Dessa maneira, como deseja-se utilizar essa infraestrutura para desenvolver aplicações, torna-se necessário que haja um serviço que promova o sincronismo de entidades.

Como já mencionado, foi padronizado que, quando possível, toda entidade virtual deva oferecer o recurso de *timestamp* além do serviço de *delay*. Esses são necessários para que seja realizado o serviço de sincronismo. O serviço de *delay* consiste em aplicar um atraso de até 1 período de amostragem, em apenas um instante. Na Figura 26, isso pode ser melhor entendido.

Figura 26 – Representação gráfica da aplicação de um *delay* na amostragem do dado de uma entidade.



Fonte: Produção do próprio autor.

Além disso, vale frisar que os *timestamps* são gerados a partir do tempo do sistema de onde está sendo executado o *gateway* de cada entidade, quando houver. Portanto, para que seja possível realizar o sincronismo, os relógios dos computadores devem estar sincronizados, utilizando por exemplo um servidor NTP (do inglês, *Network Time Protocol*).

Na Figura 27, encontra-se um fluxograma que descreve o funcionamento desse serviço. Após inicializado, o serviço aguarda o recebimento de uma requisição com uma lista de entidades. Após o recebimento, verifica-se se todas as entidades possuem a mesma taxa de amostragem. Se sim, aplica-se um *delay* igual a zero em todas entidades; caso contrário, retorna-se um código de erro indicando diferentes taxas. Feito isso, consome-se 10 amostras de *timestamp* de cada entidade. Em seguida, verifica-se qual entidade está mais atrasada em relação às outras, para que assim, seja possível determinar qual o *delay* que deve-se aplicar a cada entidade para que elas se sincronizem. Envia-se então para cada uma o *delay* calculado e então são consumidos 5 *timestamp* para verificar se o sincronismo foi realizado com sucesso. Se foi, retorna-se um código que indica o sucesso, caso contrário,

repete-se a operação mais uma única vez, e, se mesmo assim não se atingir o sincronismo, é retornado um código de erro.

Para ficar mais claro, na Figura 28 encontra-se um exemplo de sincronismo de 3 entidades. Após a captura de 10 amostras, (s0 à s9), verifica-se qual é a mais atrasada. Neste caso, a entidade mais atrasada foi a 1. Calcula-se, então, o *delay* entre essa entidade e as outras para cada instante, tira-se a média dos *delays* referentes a cada entidade, para que sejam então enviados às entidades.

Para compreender melhor como funciona o serviço de sincronismo, na Figura 29 temos a representação em camadas do serviço de sincronismo para duas entidades, com detalhamento da estrutura interna criada pelo *broker*. Neste exemplo, temos uma aplicação que faz a requisição do sincronismo das entidades E1 e E2. A representação do pedido foi omitida para simplificar o diagrama. Como mencionado anteriormente, o serviço de sincronismo utiliza o recurso *timestamp* das entidades, recebendo-os através das filas Q2 e Q3, que possuem os *bindings* “E1.timestamp” e “E2.timestamp”. Após receber a quantidade devida de *timestamps*, o serviço de sincronismo calcula os *delays* que devem ser aplicados em cada entidade e os envia para o *broker*. O *delay* de cada entidade é enviado com seu respectivo *routing_key*, no caso, “E1.delay” e “E2.delay”, sendo então encaminhados para a fila de serviços das entidades, Qe1 e Qe2.

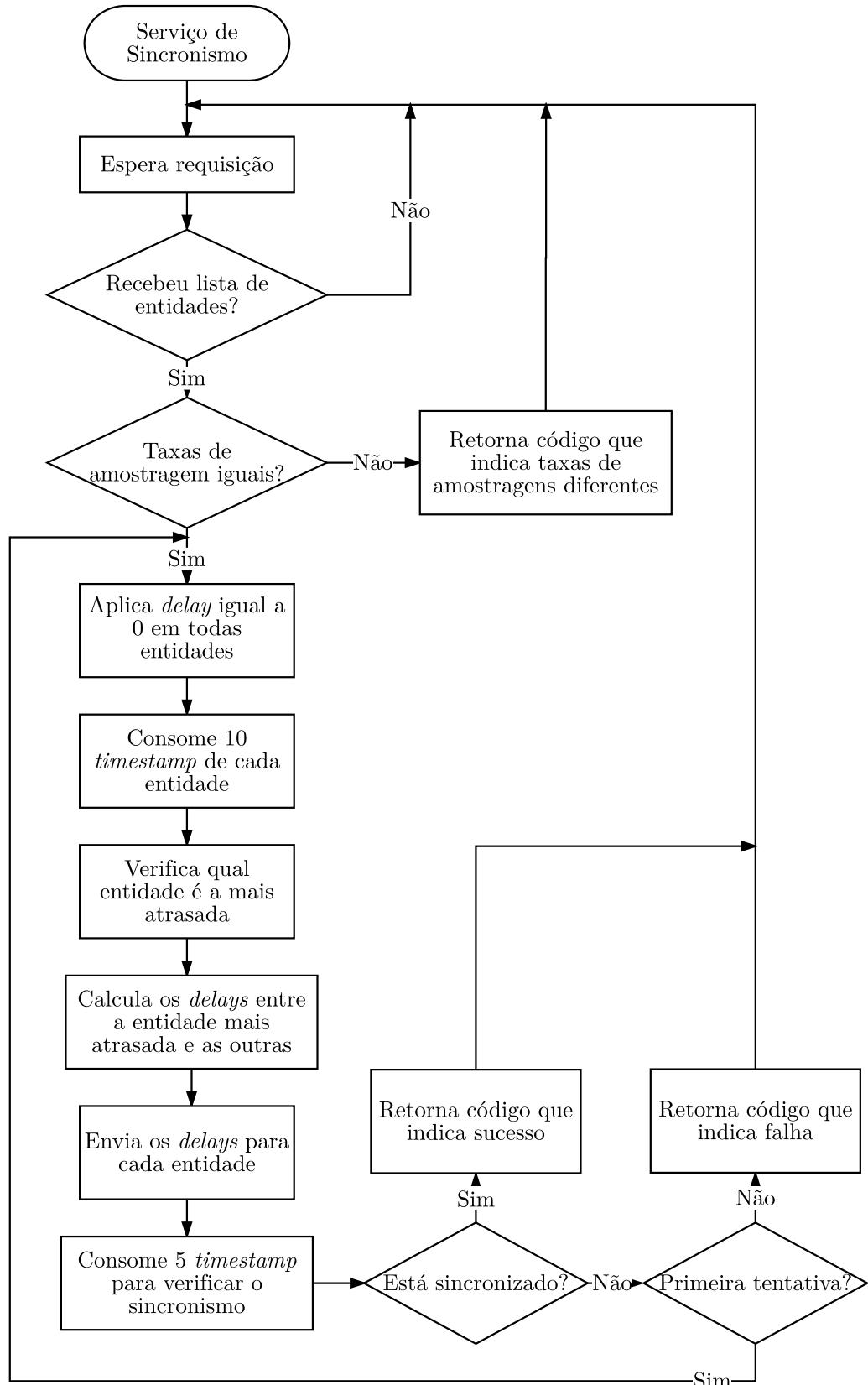
Esse serviço pode ser escalado para que várias aplicações o utilizem simultaneamente, visto que o tempo para o atendimento do pedido ser concluído vai depender do tempo de amostragem das entidades, uma vez que o serviço consome uma quantidade fixa de *timestamps* para poder calcular os *delays*.

3.2.3.2 Serviço de Busca de Entidades

Esse serviço tem o objetivo de localizar as entidades de uma determinada classe presentes na infraestrutura do espaço inteligente. Para que esse serviço funcione, como já foi mencionado, toda entidade deve possuir um serviço de *info* que retorne uma estrutura de dados de nome *entity*, contendo o tipo de entidade, seu *id*, e os recursos e serviços disponíveis.

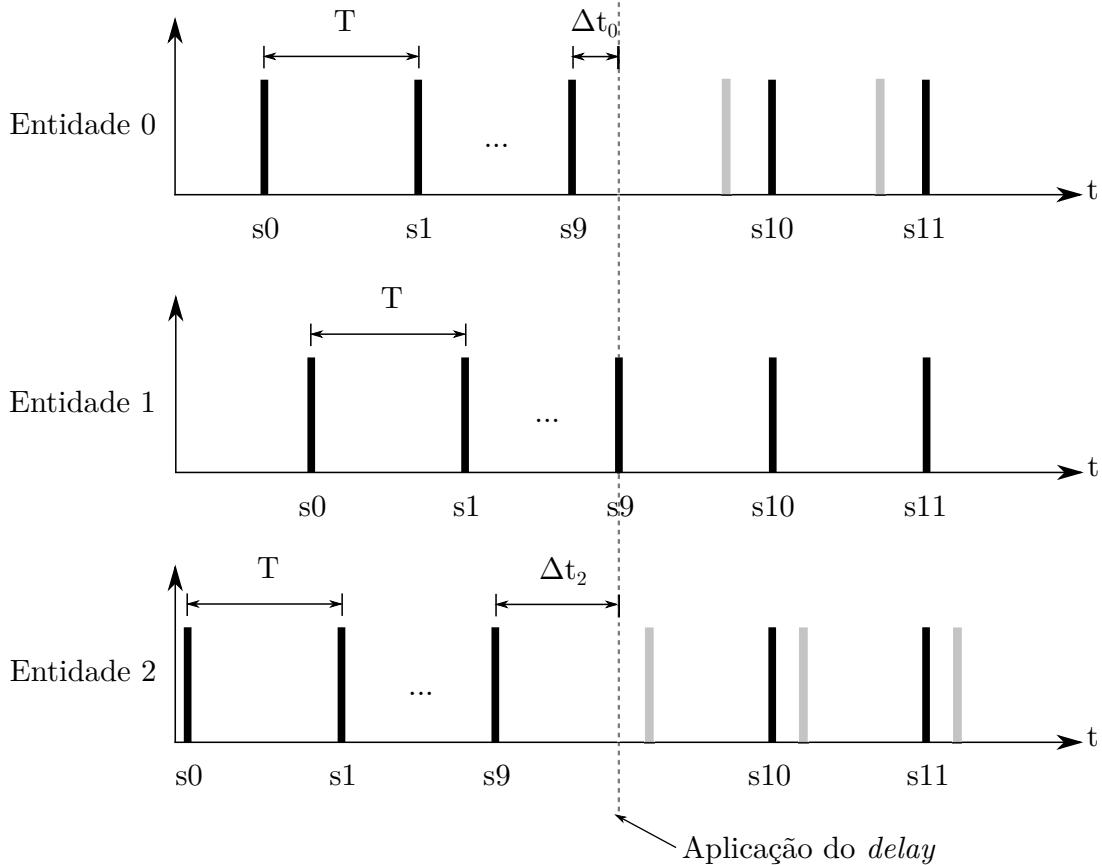
A busca por entidades é inicialmente feita realizando uma consulta ao *broker* através de uma API REST (do inglês, *Representational State Transfer*). Essa consulta busca os *bindings* associados ao *exchange* de serviços. Após isso, filtra-se apenas os serviços de *info* que pertencem ao tipo de entidade que está se buscando. Cada serviço de *info* é então requisitado, recebendo como resposta a estrutura de dados com as informações fornecidas por esse serviço. Agrupa-se todas as informações em um *array* e envia-se a resposta da requisição do serviço de busca de entidades.

Figura 27 – Fluxograma que representa o serviço de sincronismo de entidades.



Fonte: Produção do próprio autor.

Figura 28 – Representação gráfica do processo de leitura de *timestamp*, cálculo e aplicação de *delay* nas entidades a serem sincronizadas.



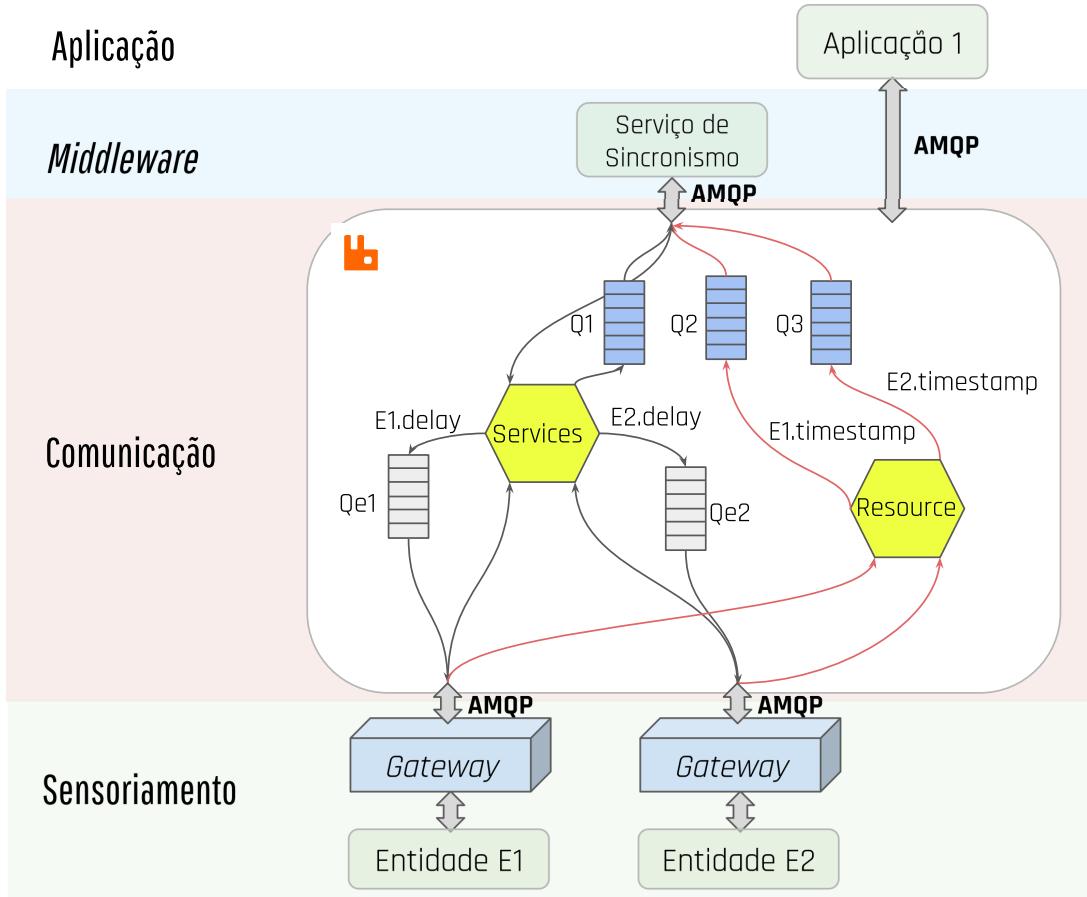
Fonte: Produção do próprio autor.

3.2.3.3 Serviço de Reconstrução 3D

Esse é um serviço de processamento que dá suporte a uma aplicação de reconstrução 3D, que vem sendo desenvolvida em um trabalho de mestrado no mesmo laboratório (SILVA; LUBE; VASSALLO, 2016). O processo de reconstrução em questão consiste em, a partir de uma amostra de 10 imagens e uma imagem de referência, são realizadas uma série de operações para determinar a profundidade correspondente a cada *pixel* da imagem. Como resultado, obtém-se uma nuvem de pontos para cada par de imagem. O processo é finalizado fundindo-se a informação das 10 nuvens de pontos obtidas. Um resultado desse processo pode ser visto na Figura 30.

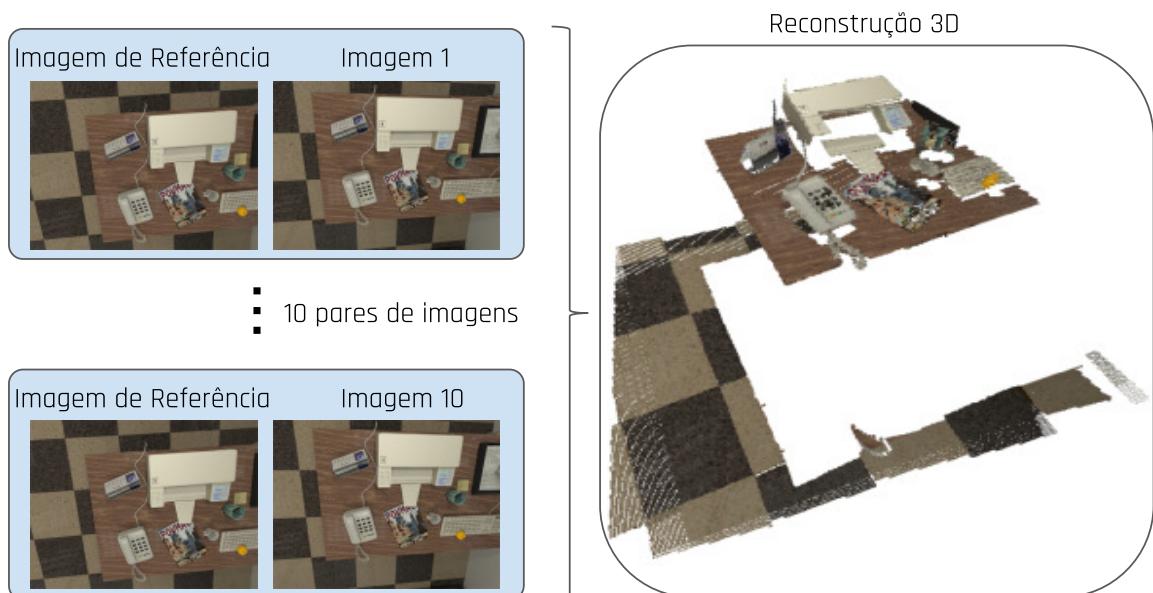
Como o processo realizado em cada par de imagens é independente, ele pode ser realizado em paralelo. Ou seja, quanto mais pares de imagens puderem ser processados ao mesmo tempo, mais rápido se obtém o resultado final. A partir do momento que esse serviço é incorporado à infraestrutura, um pedido enviado ao serviço de reconstrução vai para uma fila, e, quando há alguma instância desocupada, o pedido é encaminhado. Dessa forma, pode-se ter instâncias sendo executadas em computadores diferentes, sendo necessário

Figura 29 – Representação em camadas do funcionamento do serviço de sincronismo para duas entidades.



Fonte: Produção do próprio autor.

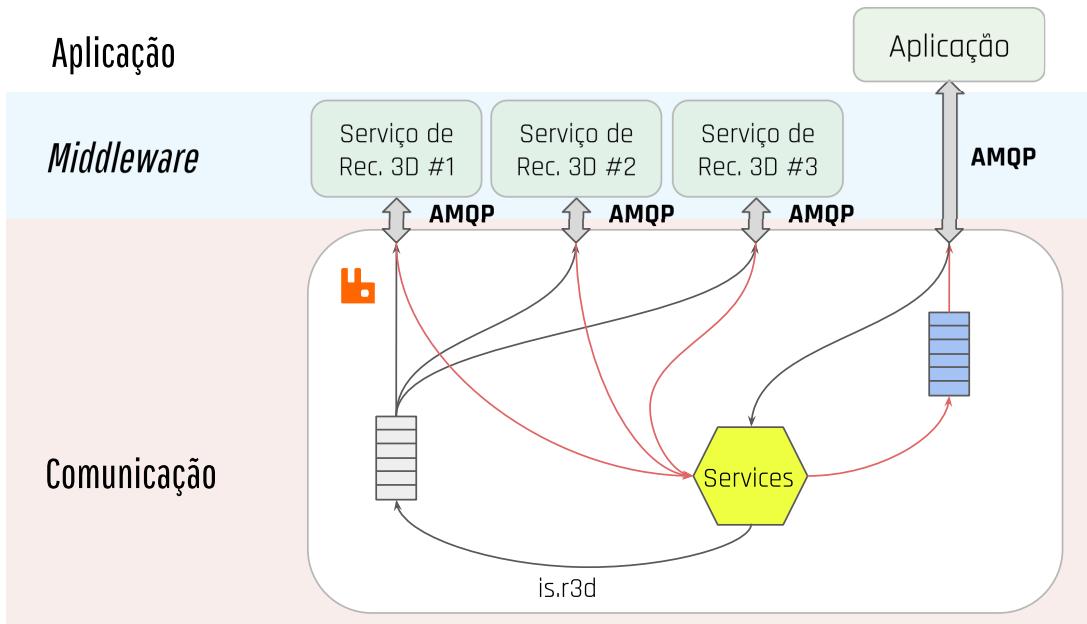
Figura 30 – Resultado obtido utilizando o serviço de reconstrução 3D.



Fonte: Produção do próprio autor.

apenas que estejam conectados ao *broker*. Na Figura 31 temos um exemplo com três instâncias do serviço de reconstrução 3D. Essas três recebem requisições enviadas por uma aplicação, e são encaminhadas a uma fila que possui um *binding* com o *exchange services* igual a “is.r3d”. As requisições são distribuídas para cada instância do serviço de acordo com sua disponibilidade, seguindo uma sequência. A resposta de cada processamento realizado é enviada ao *broker* e é encaminhada a uma fila criada pela aplicação para receber as respostas.

Figura 31 – Representação em camadas do funcionamento do serviço de reconstrução 3D.



Fonte: Produção do próprio autor.

Utilizando a ferramenta de *containers Docker*, construiu-se um *container* contendo o serviço de reconstrução 3D, com o intuito de mostrar a flexibilidade que se ganha ao utilizar a virtualização para disponibilizar serviços desse tipo. Pode-se utilizar ainda a ferramenta *Docker Swarm* para a criação de *clusters* de *Docker*. Dessa maneira, podemos ter vários nós que ofereçam recursos para um *container* ser implantado. Assim, ao criar-se um novo *container* em um *cluster* de *Docker*, a ferramenta que definirá em qual nó ele será implantado. Além disso, a ferramenta *Docker Swarm* possibilita escolher o número de instâncias de um *container* que serão implantadas no *cluster*. Essa quantidade pode ser alterada a qualquer momento, ficando a cargo da ferramenta realizar a distribuição entre os nós. Ademais, escolhida a quantidade, caso algum *container* falhe, a ferramenta se encarrega de reimplantá-lo no *cluster*. Desta maneira, um serviço de processamento como o descrito nessa seção pode ser escalado de acordo com a demanda, alocando os recursos computacionais apenas quando necessário.

3.3 Interfaces de desenvolvimento

Para o desenvolvedor de aplicações, foram definidas interfaces para cada tipo de entidade virtual, câmeras e robôs, além dos serviços de suporte, sincronismo e busca de entidades. Essas interfaces visam facilitar o processo de desenvolvimento, deixando transparente os conceitos acerca da camada de comunicação, como *broker*, tópicos, filas, etc. A única informação que o usuário deverá saber sobre o *broker* é o seu endereço IP ou *hostname*. Exemplos de utilização dessas interfaces apresentadas a seguir serão mostrados no Capítulo 4.

3.3.1 Cameras

A abstração criada para as câmeras possibilita apenas o acesso aos serviços de interesse ao desenvolvedor de aplicação. Por exemplo, não espera-se que o serviço de *delay* seja utilizado por uma aplicação, mas sim por outro serviço, o de sincronismo por exemplo. Já serviços que afetam diretamente um recurso da câmera, como o de resolução da imagem ou o seu tipo, devem ter interfaces de acesso para o desenvolvedor. A utilização dessa classe é feita através do header “cameras/cameras.hpp”, como explicado no Apêndice A.

O construtor para a classe *Camera* é definido por

```
1 Camera(const string& broker, const string& id);
2 Camera(const string& broker, const is::entity& entity);
```

onde o primeiro parâmetro corresponde ao endereço de IP ou *hostname* do *broker*, e o segundo parâmetros pode ser o ID da câmera que deseja-se utilizar, ou ainda a descrição da entidade câmera presente no tipo de dado *entity*. Foram criadas essas duas opções pois o desenvolvedor pode saber o ID da câmera através, por exemplo de um portal *web*, ou ele pode utilizar o serviço de busca de entidades que retornará uma lista de *entity*.

Depois de instanciado o objeto *Camera*, o usuário pode utilizar os métodos para acessar os serviços e alterar os seguintes parâmetros da câmera: FPS, tipo da imagem e resolução. Os métodos estão definidos à seguir. Todos eles retornam um *bool* indicando o sucesso ou não da requisição.

```
1 bool set_fps(float fps);
2 bool set_type(ImageType type);
3 bool set_resolution(int width, int height);
```

Para que a aplicação comece a consumir o recurso da câmera, o usuário deve utilizar o método *start_consume*. Feito isso, o desenvolvedor está habilitado a utilizar o seguinte método

```
1 cv::Mat get_frame();
```

Internamente, esse método fica aguardando chegar um *frame*, e então o converte para `cv::Mat`. Optou-se por retornar o frame no formato utilizado pelo OpenCV (BRADSKI, 2000) por ser a biblioteca mais difundida para aplicações de processamento de imagens e visão computacional.

Visando as aplicações que utilizam mais de uma câmera, criou-se também a classe *GroupCamera*. Com essa, pode-se instanciar um objeto que agrupa várias câmeras, sendo possível acessar os serviços de todas com apenas uma chamada de função. Seu construtor está definido como

```
1 GroupCamera(const string& broker, const vector<string>& ids);
2 GroupCamera(const string& broker, const vector<entity>& entities);
```

Aqui também é necessário informar o IP ou *hostname* do *broker*, e uma lista de IDs ou entidades. Os métodos para acesso aos serviços são os mesmos que para a classe *Camera*, contudo, eles retornam um vetor de *bool*, que indica o sucesso ou não da requisição para cada câmera do grupo.

O método *start_consume* tem a mesma funcionalidade que o da classe *Camera*. Os *frames* das câmeras pertencentes ao grupo são adquiridos através do método *get_frame*, que retorna um vetor com todas as imagens do grupo de câmeras, também utilizando o formato `cv::Mat`.

3.3.2 Robôs

Para os robôs, o acesso aos recursos e serviços pode ser feito pelo desenvolvedor de aplicações através da classe *Robot*, definida no *header* “*robots/robots.hpp*”, como explicado no Apêndice A. O construtor dessa classe é definido, como mostrado a seguir de modo semelhante ao que foi feito para as câmeras: primeiro parâmetro refere-se ao *broker* e o segundo à entidade.

```
1 Robot(const string& broker, const string& id);
2 Robot(const string& broker, const is::entity& entity);
```

Os serviços disponíveis para os robôs são de velocidade, pose e taxa de amostragem. Os protótipos das funções estão definidos à seguir.

```
1 bool set_velocity(tuple<double, double> velocites);
2 bool set_pose(tuple<double, double, double> pose);
3 bool set_sample_rate(double sample_rate);
```

A função *set_velocity* recebe o par de velocidades linear e angular, em mm/s e °/s respectivamente. A pose pode ser definida através da função *set_pose*, passando como parâmetros as coordenadas *x* e *y* em mm e a orientação Φ em graus. Por fim, a taxa de amostragem deve ser enviada em Hertz (amostras por segundo).

O recurso disponível para o desenvolvedor de aplicação é apenas a odometria. Essa pode ser consumida através do método *get_odometry*, definido a seguir, mas somente após o desenvolvedor chamar a função *start_consume*. O método de odometria retorna o tipo de dado *Odometry*, já previamente definido, que contém a pose do robô naquele instante, bem como o *timestamp* correspondente.

```
1 Odometry get_odometry();
```

Assim como foi feito para as câmeras, também é possível definir um grupo de robôs através da classe *GroupRobot*, disponível no mesmo *header* da classe *Robot*. Seu construtor segue o mesmo padrão: recebe como parâmetro informações do *broker* e uma lista de IDs ou entidades.

Em relação aos métodos para utilização dos serviços, há uma diferença em relação ao das câmeras. Para configurar a pose e a velocidade, envia-se um valor diferente para cada robô pertencente ao grupo. Essa escolha foi baseada em trabalhos já realizados na área utilizando mais de um robô, nos quais se definem velocidades diferentes para cada um deles. Já para as câmeras, normalmente se ajusta os parâmetros de todas as câmeras com valores iguais em uma mesma aplicação. Caso o desenvolvedor deseje parâmetros diferentes para as câmeras, ele pode criar objetos *Camera* diferentes. O único parâmetro do grupo de robôs cujo valor enviado é o mesmo para todos é a taxa de amostragem, através do método *set_sample_rate*.

O método *get_odometry* para a classe *GroupRobot* retorna um vetor com a odometria referente a cada robô do grupo. Mais uma vez, esse método só pode ser utilizado após a chamada do método *start_consume*.

3.3.3 Serviços

Para os serviços de sincronismo e busca de entidades foram implementadas interfaces de utilização. Para o serviço de sincronismo, deve-se incluir o *header* “services/sync/sync.hpp”, como explicado no Apêndice A. Neste, encontram-se duas funções que para a utilização do serviço, são elas:

```
1 // List of is :: entities
2 Sync sync(string broker, vector<entity> entities);
```

```
3 // List of entities names i.e. { "camera.0", "robot.1" }
4 Sync sync(string broker, vector<string> entities);
```

Para ambas interfaces, deve-se informar o endereço IP ou *hostname* do *broker*. O segundo parâmetro pode ser uma lista do tipo *entity*, ou uma lista de nomes das entidades, composta pelo padrão “classe.ID”. O retorno é uma enumeração que indica se o sincronismo obteve sucesso, ou, caso contrário, indica qual erro. A definição de Sync é dada por

```
1 enum class Sync{
2     success = 0,
3     fail = 1,
4     different_fps = 2,
5     timeout = 3
6 };
```

O serviço de busca de entidades está disponível no *header* “services/find-entities/find-entities.hpp”, como explicado no Apêndice A. Sua interface de utilização é dada por

```
1 vector<entity> find_entities(string broker, string filter);
```

onde o primeiro parâmetro corresponde à informação da localização do *broker* e a segunda é um filtro para a busca, que nesse caso corresponde à classe de entidades que deseja-se listar. O retorno é um vetor do tipo *entity*, ou seja, pode ser utilizado nas interfaces de *Camera*, *GroupCamera*, *Robot*, *GroupRobot*, além do serviço de sincronismo.

4 EXPERIMENTOS E RESULTADOS

Nesse capítulo serão apresentados códigos de exemplo utilizando as interfaces descritas na Seção 3.3. Além disso, serão apresentados resultados de experimentos feitos para validar o serviço de sincronismo. Medições de latência para *pub/sub* de imagens foram feitas para mostrar que os valores obtidos estão dentro do aceitável para as aplicações que se pretende. Para o serviço de processamento distribuído discutido na Seção 3.2.3.3, foram feitos testes também para mostrar o ganho de desempenho com diferentes números de instâncias do serviço.

4.1 Exemplos de utilização das interfaces

Os exemplos apresentados a seguir encontram-se dentro da pasta “examples/”, como mostrado no Apêndice A.

4.1.1 Exibir câmeras sincronizadas

Esse exemplo consiste em exibir todas as câmeras conectadas à infraestrutura. Nele, foram utilizados os serviços de busca de entidades e de sincronismo. O código desse exemplo está apresentado no Quadro 1.

Vale destacar algumas linhas desse código.

- linhas 4 a 6: são *headers* das interfaces implementadas para utilizar a infraestrutura;
- linha 12: define o *hostname* do *broker*;
- linha 14: utiliza o serviço de busca de entidades com o filtro “cameras”;
- linha 16: define o objeto *GroupCamera*, passando como parâmetro a lista de entidades retornada na linha 12;
- linha 18: configura o FPS das câmeras para 1,0;
- linha 19: configura o tipo da imagem das câmeras para *grayscale*;
- linha 20: configura a resolução das câmeras para 640×480 *pixels*;
- linha 24: utiliza o serviço de sincronismo passando como parâmetro a lista de entidades retornada na linha 12;
- linha 26: verifica o resultado retornado pelo serviço de sincronismo;
- linha 31: inicia o consumo das imagens das câmeras;

Quadro 1 – Código exemplo que exibe câmeras sincronizadas.

```

1 #include <iostream>
2 #include "opencv2/imgproc.hpp"
3
4 #include "services/sync/sync.hpp"
5 #include "services/find-entities/find-entities.hpp"
6 #include "cameras/cameras.hpp"
7
8 using namespace is;
9 using namespace is::camera;
10
11 int main (int argc, char *argv[]) {
12     std::string broker = "localhost";
13
14     auto entities = find_entities(broker, "camera");
15
16     GroupCamera cameras(broker, entities);
17
18     cameras.set_fps(1.0);
19     cameras.set_type(ImageType::rgb);
20     cameras.set_resolution(640, 480);
21
22     std::cout << "Waiting for sync.." << std::endl;
23
24     auto sync_code = sync(broker, entities);
25
26     if (sync_code != Sync::success) {
27         std::cout << "Sync failed, exiting.." << std::endl;
28         exit(-1);
29     }
30
31     cameras.start_consume();
32
33     while (1) {
34         auto frames = cameras.get_frame();
35
36         for (auto& f : frames) {
37             cv::resize(f, f, cv::Size(f.cols/2, f.rows/2));
38         }
39         cv::Mat image;
40         cv::hconcat(frames, image);
41
42         cv::imshow("cameras", image);
43         cv::waitKey(1);
44     }
45
46     return 0;
47 }
```

- linha 34: recebe o vetor com as imagens de cada câmera;
- linhas 36 à 43: redimensiona cada imagem, as concatena horizontalmente e as exibe.

4.1.2 Exibir a odometria de um robô

Esse exemplo serve para mostrar a utilização da interface feita para utilizar robôs conectados à infraestrutura. O exemplo apresentado no Quadro 2 consiste em exibir a odometria de um robô, após a configuração da taxa de amostragem, sua posição inicial e suas velocidades.

Quadro 2 – Código exemplo que exibe a odometria de um robô.

```

1 #include <iostream>
2 #include "robots/robots.hpp"
3
4 using namespace is;
5 using namespace is::robot;
6
7 int main(int argc, char const *argv[]) {
8
9     std::string broker = "localhost";
10    std::string id = "0";
11
12    Robot robot(broker, id);
13
14    robot.set_sample_rate(10.0);
15    robot.set_pose(make_tuple(0, 0, 0));
16    robot.set_velocity(make_tuple(100.0, 0.0));
17
18    robot.start_consume();
19    while (1) {
20        auto odometry = robot.get_odometry();
21        std::cout << "(" << odometry.x
22                      << "," << odometry.y
23                      << "," << odometry.th << ")\\n" << std::flush;
24    }
25
26    return 0;
27 }
```

Para melhor entendimento, algumas linhas do código serão detalhadas.

- linha 2: *header* da interface feita para utilizar os robôs da infraestrutura;
- linha 9: define o *hostname* do *broker*;
- linha 10: define o ID do robô;
- linha 12: define o objeto *Robot*, tendo como parâmetro de entrada o ID definido na linha 10;
- linha 14: configura a taxa de amostragem para 10 Hertz;
- linha 15: configura a pose inicial do robô para [0 0 0];
- linha 16: configura a velocidade linear do robô para 100 mm/s e angular para 0 °/s;

- linha 18: inicia o consumo da odometria do robô;
- linha 20: recebe a odometria do robô;
- linhas 21 a 23: exibe a odometria do robô.

4.2 Testes de latência para *pub/sub* de imagens

Para utilizar as câmeras escolhidas para a infraestrutura do espaço inteligente, as aplicações poderiam ser desenvolvidas diretamente com a SDK fornecida pelo fabricante. No entanto, a proposta deste projeto de graduação é desenvolver uma arquitetura baseada em IoT para o espaço inteligente. Nessa arquitetura, existe um elemento concentrador de mensagens chamado *broker*. Com o uso do *broker*, os pacotes que compõe uma imagem tem que trafegar pela rede para sair de seu *gateway*, ir até o *broker*, para então serem encaminhados para a aplicação. Aumentando o caminho percorrido pelos pacotes e o número de saltos, aumenta-se a latência para que o pacote seja entregue.

Além disso, é possível ter várias aplicações requisitando imagens do *broker*, fazendo com que o canal de comunicação fique sobrecarregado. Outro fator importante, é que o sistema é baseado em imagens que, dependendo de sua resolução e taxa de captura, podem gerar tráfegos na rede na ordem de dezenas de *megabytes* por segundo.

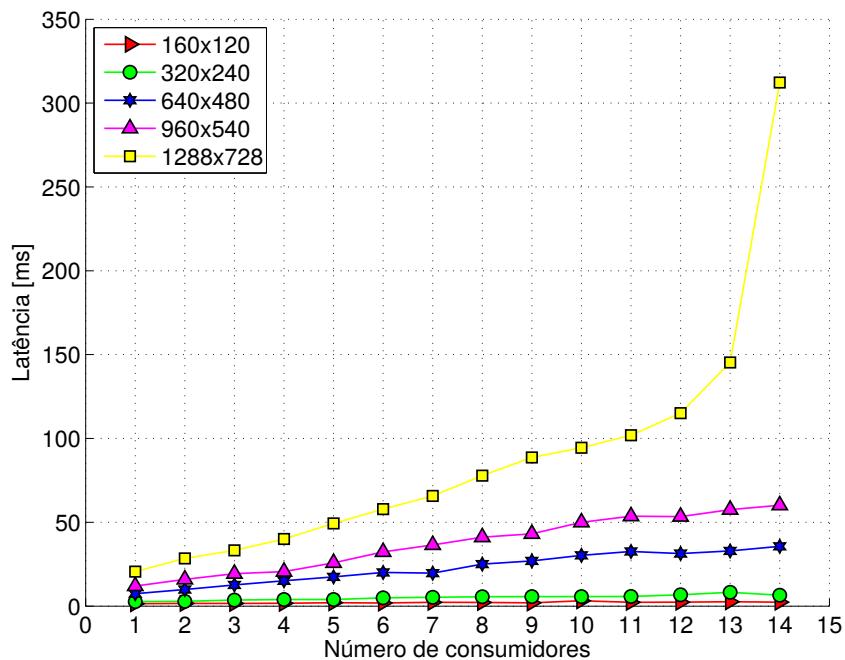
Essa infraestrutura deve ser capaz de suportar aplicações de controle de robôs baseados em imagens. Uma vez que o robô utilizado possui sua taxa de atualização interna mínima de 100 ms, deseja-se que a latência para que a imagem da câmera utilizada em uma aplicação dessas seja menor que esse valor. Desse modo, considerando-se que após receber a imagem ainda há um tempo para processá-la e enviar as informações de controle para o robô, o tempo total deve ser no máximo 100 ms.

O experimento realizado para verificar quantitativamente a latência foi iniciar o *gateway* de uma câmera em um computador, e conectá-lo ao *broker* em outro computador. Ambos computadores foram conectados ao mesmo *switch* através de interfaces Gigabit. No computador do *gateway* da câmera, iniciou-se uma aplicação que configura os parâmetros da câmera para 10 FPS, e o tipo de imagem para escala de cinza. A resolução foi configurada em cinco valores diferentes: 160×120 , 320×240 , 640×480 , 960×540 e, o máximo valor suportado, 1288×728 . Para cada resolução, foram consumidas 100 imagens, para que então fosse medida a latência. Além da latência, foi medido o *throughput*, a fim de verificar se a vazão estava próxima do limite teórico.

Dessa forma, a latência medida corresponde ao tempo da imagem sair do *gateway*, ir até ao

broker e ser enviado a uma aplicação. Para a atual infraestrutura, com apenas um *switch*, esse seria o pior caso. As configurações de resolução escolhidas correspondem a valores típicos e foram variados para mudar a ocupação do canal entre o *broker* e o *switch*. Os gráficos apresentados nas Figuras 32 e 33 mostram, respectivamente, os valores de latências e *throughput* para cada resolução, variando o número de consumidores simultâneos de 1 até 14.

Figura 32 – Gráfico da latência medida para cinco resoluções diferentes ao se variar a quantidade de consumidores simultâneos de 1 até 14.

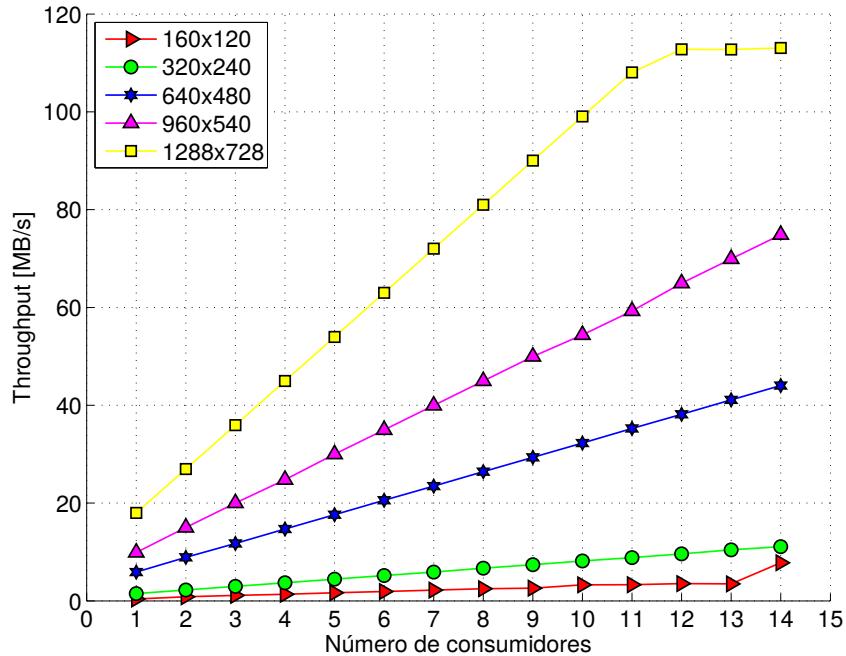


Fonte: Produção do próprio autor.

No gráfico de latência, mostrado na Figura 32, nota-se que para cada resolução, ela é crescente à medida que se aumenta o número de consumidores simultâneos, e será maior para resoluções maiores. Com a resolução 1288×728 e 14 consumidores simultâneos, nota-se um aumento fora do padrão observado até 13 consumidores, com um aumento de 145,3 ms para 312,3 ms. Um dos motivos para isso ocorrer, é o valor do *throughput* nesse ponto de operação que é de 113,1 MB/s, valor próximo ao máximo teórico para uma rede Gigabit, que é de 125 MB/s. Além disso, como pode ser observado na Figura 33, para a maior resolução, a partir de 12 consumidores, o valor do *throughput* praticamente não sofre variações, confirmando que está no limite da capacidade do canal.

Apesar dos valores de latência obtidos, os experimentos mostraram que é possível que aplicações executadas recebam imagens com uma latência bem menor que 100 ms, mesmo com execução de aplicações simultâneas. A limitação existente está associada com a capacidade física do canal de comunicação entre o *broker* e o elemento de rede ao qual ele

Figura 33 – Gráfico do *throughput* medido para cinco resoluções diferentes ao se variar a quantidade de consumidores simultâneos de 1 até 14.



Fonte: Produção do próprio autor.

está conectado. Ademais, para o que se pretende da infraestrutura proposta, os valores obtidos são satisfatórios.

4.3 Validação do serviço de sincronismo

Para validar o serviço de sincronismo e estimar o erro entre cada entidade sincronizada, foram utilizadas as quatro câmeras montadas lado a lado, como mostrado na Figura 34a. De frente para as câmeras, foi posicionada uma placa de desenvolvimento com uma FPGA que possui quatro *displays* de sete segmentos, como mostrado na Figura 34b. Foi então programado para que a FPGA exibisse um número de 0000 a 9999, incrementando-o a cada 1 ms, e reiniciando a contagem ao atingir o valor máximo. A base de tempo utilizada para a contagem foi o *clock* interno da FPGA.

Como o valor do número mostrado nos *displays* é atualizado a cada 1 ms, para que seja possível capturar o valor de um número com nitidez, deve-se garantir que o tempo de exposição da câmera seja bem menor que 1 ms. Foi, então, ajustado o tempo de abertura para 0,082 ms, que corresponde ao menor tempo de abertura possível para essas câmeras. Esse tempo, mesmo sendo muito pequeno, é suficiente para sensibilizar o sensor da câmera e poder capturar os números dos *displays*, como pode ser visto na Figura 35a.

Figura 34 – Montagem das câmeras e da placa com FPGA para o experimento de validação do serviço de sincronismo.



Fonte: Produção do próprio autor.

O experimento proposto consistiu em realizar a solicitação de sincronismo para as quatro entidades virtuais que representam as câmeras e, ao receber uma resposta de que está sincronizado, capturar e salvar 100 imagens com um tempo de amostragem fixo. Para melhorar a visualização, as imagens das quatro câmeras foram concatenadas horizontalmente, como mostrado na Figura 35a, onde também pode-se notar a nitidez dos números com o ajuste no tempo de abertura. Para facilitar a leitura dos valores, as imagens foram recortadas como mostrado na Figura 35b também foi adicionado ao lado esquerdo da figura o número que indica a amostra que aquela imagem representa.

Figura 35 – Imagem das quatro câmeras mostrando o número exido pelos *displays* (a), e corte feito na imagem para facilitar a leitura dos valores (b).



Fonte: Produção do próprio autor.

Foram escolhidos dois intervalos de amostragem: 1 segundo, totalizando 100 segundos capturando imagens, e 36 segundos, totalizando 1 hora. Para cada intervalo de amostragem, repetiu-se o experimento três vezes. Na Figura 36, pode-se visualizar as imagens capturadas dos *displays* em 11 instantes, incluindo o inicial e o final, de um dos 3 experimentos. Pode-se notar também que há uma imprecisão no *clock* da FPGA. Por exemplo, o valor exibido no *display* mais à esquerda na amostra 009 é 9049. Já na amostra 099 o valor registrado foi 9045, mas deveria ser 9049. Esse erro de 4 ms em 90 s foi considerado desprezível, visto que deseja-se aferir o erro relativo entre cada câmera, e, consequentemente, entre cada amostra.

Figura 36 – Imagens dos *displays* do experimento com intervalo de amostragem igual a 1 s, em 10 instantes igualmente espaçados após o início.



Fonte: Produção do próprio autor.

Na Tabela 1, estão apresentados os erros máximos em ms para cada amostra, ou seja, a diferença entre a câmera mais adiantada e a mais atrasada, além do erro médio de cada amostra. Nota-se que passados 100 s, não há alteração significativa no erro de sincronismo, atingindo um erro máximo de 2 ms.

Tabela 1 – Erro máximo em milissegundos para cada amostra dos três experimentos com tempo de amostragem igual à 1 segundo.

Experimento	Amostra										
	000	009	019	029	039	049	059	069	079	089	099
1	1	1	0	1	1	1	0	1	1	1	1
2	1	1	2	1	2	1	2	1	2	2	2
3	1	0	1	1	1	1	2	2	2	1	1
Média	1	0,67	1	1	1,33	1	1,33	1,33	1,67	1,33	1,33

Fonte: Produção do próprio autor.

O experimento com tempo de amostragem de 36 segundos, com duração total de 1 hora, foi feito com intuito de tornar o erro de sincronismo mais evidente. A Figura 37 mostra as imagens capturadas de alguns instantes em um dos três experimentos feitos.

Figura 37 – Imagens dos *displays* do experimento com intervalo de amostragem igual a 36 segundos, em 10 instantes igualmente espaçados após o início.



Fonte: Produção do próprio autor.

Na Tabela 2, estão apresentados os erros máximos em ms para cada amostra. Pode-se notar que, entre cada amostra apresentada, que tem um intervalo de 360 segundos, há um crescimento do erro de cerca de 3 ms. Após a captura de 100 imagens, o erro médio apresentado foi de 33 ms.

Tabela 2 – Erro máximo em ms para cada amostra dos três experimentos com tempo de amostragem igual à 36 s.

Experimento	Amostra										
	000	009	019	029	039	049	059	069	079	089	099
1	0	3	6	9	12	15	18	21	24	27	32
2	1	3	7	9	13	17	20	23	26	30	34
3	0	4	6	9	13	15	18	22	25	28	32
Média	0,33	3,33	6,33	9	12,67	15,67	18,67	22	25	28,33	33

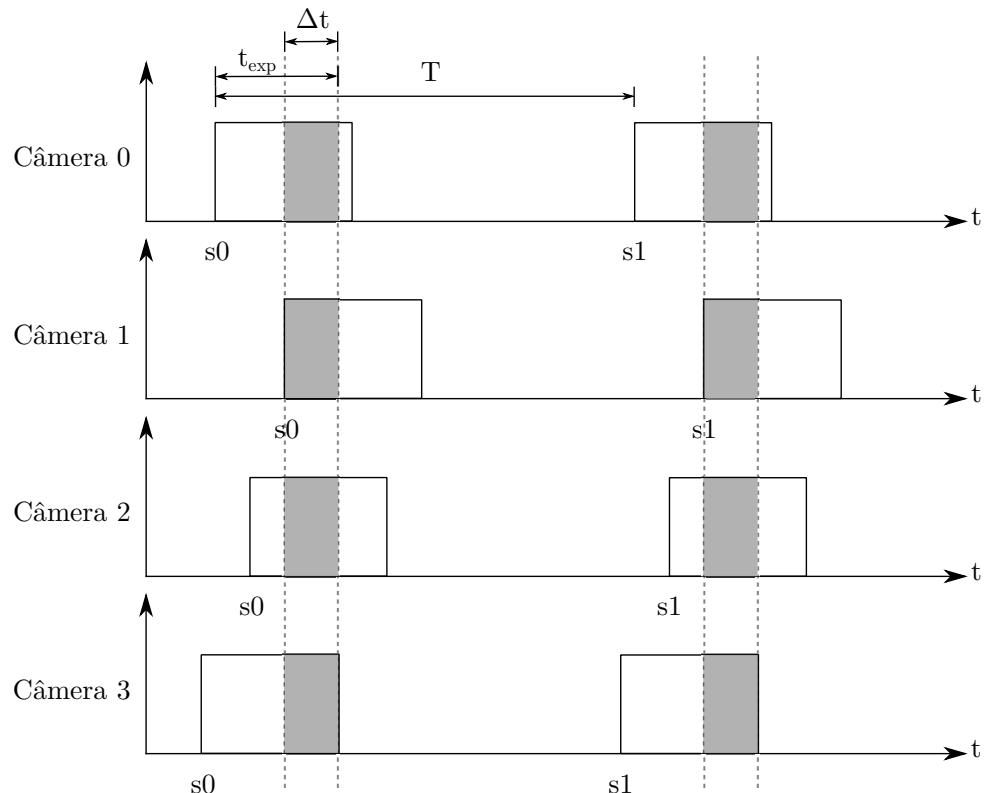
Fonte: Produção do próprio autor.

O erro máximo admitido por uma aplicação vai depender de diversos fatores, como por exemplo os requisitos de seu uso. Contudo, no trabalho com câmeras, existe outro fator que influência no erro máximo admitido: o tempo de exposição. Como mencionado, para esse experimento, esse tempo foi reduzido para 0,082 ms. Entretanto, em operações normais, o tempo de exposição fica em torno de dezenas de milissegundos, variando de acordo com a iluminação e o tempo de amostragem. Para uma operação a 1 FPS, esse tempo pode chegar à 100 ms. Ou seja, o sensor da câmera recebe luz do ambiente por todo esse tempo.

Dessa forma, se estivermos operando com quatro câmeras, e o erro máximo for de 33 ms, no pior caso, teremos uma sobreposição de exposição entre as câmeras de 67 ms. Isso faz com que haja pouca influência do erro de sincronismo no resultado visual da captura das imagens.

A partir da Figura 38 é possível compreender melhor a relação entre o tempo de exposição das câmeras e o erro de sincronismo. No exemplo, há quatro câmeras que foram inicialmente sincronizadas. Após um tempo, o erro entre o instante de amostragem das câmeras começou a aumentar. A parte destacada em cinza dentro de cada retângulo de tempo de amostra indica a janela de sobreposição, onde todas câmeras estão com o obturador aberto. Dessa forma, mesmo as câmeras não disparando a captura da imagem ao mesmo tempo, há um intervalo de tempo Δt onde todas estão com seus sensores sendo sensibilizados.

Figura 38 – Ilustração da influência do tempo de exposição das câmeras no erro de sincronismo admitido.



Fonte: Produção do próprio autor.

Como mostrado com os experimentos, existe um tempo no qual o sincronismo entre as entidades permanece com um erro aceitável. Se uma aplicação que utiliza o serviço de sincronismo tiver um tempo de duração maior que esse tempo de persistência, uma alternativa é acionar automaticamente o serviço de sincronismo mais uma vez. Outra alternativa é que o serviço de sincronismo seja adaptado para monitorar o erro e tentar restabelecer o sincronismo assim que se identificar um erro acima do tolerável.

4.4 Testes com o serviço de resconstrução 3D

Esse experimento teve como objetivo mostrar o funcionamento de um serviço de processamento, que utiliza a infraestrutura proposta. Esse serviço foi o de reconstrução 3D, descrito na Seção 3.2.3.3. Para os testes realizados, foram utilizados 10 pares de imagens, ou seja, um total de 10 requisições ao serviço. Duas situações foram testadas: na primeira, havia apenas um consumidor com o *binding* na fila do serviço; e na segunda, haviam quatro.

Na Figura 39, é possível verificar nas informações do *broker* que há apenas um consumidor para esse serviço conectado. Além disso, observa-se na aba *Bindings* o *Routing key* do serviço de reconstrução 3D, conforme padrão descrito na Seção 3.2.1.1.

Figura 39 – Informações do *broker* que mostram a existência de apenas um consumidor para o serviço de reconstrução 3D (aba *Consumers*), além de mostrar o *Routing key* do serviço “is.r3d”.

The screenshot shows two sections of a broker configuration interface:

- Consumers** section:

Channel	Consumer tag	Ack required	Exclusive	Prefetch count	Arguments
10.255.0.3:60810 (1)	amq.ctag-HBAuMwTEIWKz7R5IiuwS_g	•	○	0	
- Bindings** section:

From	Routing key	Arguments	
(Default exchange binding)	services	is.r3d	Unbind

 A blue arrow points downwards from the "Unbind" button to a box labeled "This queue".

Fonte: Produção do próprio autor.

Ao iniciar a aplicação que realiza a reconstrução 3D, foram enviadas as 10 requisições ao *broker*, cada uma com seu respectivo par de imagens. Essas requisições esperaram na fila para serem consumidas. A Figura 40a mostra as estatísticas referentes à fila do serviço, no qual observa-se, pela coluna *Total*, que 10 mensagens foram enviadas ao *broker*. Além disso, na coluna *Ready*, há a indicação de 9 mensagens, o que quer dizer que existem 9 mensagens esperando para serem consumidas. A coluna *Unacked* exibe a quantidade de requisições que estão sendo processadas. A coluna *In memory* indica o número de pedidos que ainda está armazenado no *broker*, e a coluna *Persistent* indica a quantidade de mensagens que ainda não foram respondidas. Para este caso, *In memory* e *Persistent* indicarão a mesma quantidade.

A Figura 40b, mostra 5 mensagens na coluna *Total*, 4 em *Ready* e 1 em *Unacked*, ou seja, 5 requisições já foram respondidas, e existem 5 pendentes, com uma sendo processada. Bem no fim do processo, é possível verificar na Figura 40c que há apenas uma requisição pendente, e essa está sendo processada. Para essa configuração com apenas um consumidor, o processo de reconstrução 3D levou 229,7 s para ser concluído.

Figura 40 – Estatísticas disponíveis na interface *web* do *broker* durante o processo de reconstrução 3D, utilizando um consumidor.

		(a)							
		State	idle	Messages (?)	Total	Ready	Unacked	In memory	Persistent
Consumers	1			Messages (?)	10	9	1	10	10
Consumer utilisation (?)	0%			Message body bytes (?)	18MB	16MB	1.8MB	18MB	18MB
				Process memory (?)	87kB				

		(b)							
		State	idle	Messages (?)	Total	Ready	Unacked	In memory	Persistent
Consumers	1			Messages (?)	5	4	1	5	5
Consumer utilisation (?)	0%			Message body bytes (?)	8.8MB	7.0MB	1.8MB	8.8MB	8.8MB
				Process memory (?)	23kB				

		(c)							
		State	idle	Messages (?)	Total	Ready	Unacked	In memory	Persistent
Consumers	1			Messages (?)	1	0	1	1	1
Consumer utilisation (?)	0%			Message body bytes (?)	1.8MB	0B	1.8MB	1.8MB	1.8MB
				Process memory (?)	17kB				

Fonte: Produção do próprio autor.

O segundo teste foi feito com 4 consumidores. Essa quantidade pode ser verificada na Figura 41. Nesse teste também foram enviadas as 10 requisições no início do processo, valor esse que pode ser verificado na coluna *Total* da Figura 41a, que corresponde ao estado da fila no início do processo. Verifica-se também que há 4 requisições na coluna *Unacked*, e 6 na coluna *Ready*. Após algum tempo, já com 8 requisições respondidas, a seguinte situação é mostrada na Figura 41b: 2 mensagens na coluna de *Unacked* e nenhuma na coluna *Ready*, ou seja, restam 2 requisições, que estão sendo processadas. Como há 4 consumidores, 2 deles estão ociosos.

Para o segundo teste, o processo foi concluído com 73,6 s, o que corresponde a uma redução para menos de 1/3 do tempo gasto com apenas 1 consumidor. Esse tempo pode ser reduzido ainda mais conforme o número de consumidores escutando a fila do serviço de reconstrução aumenta. Vale ressaltar que para isso, considera-se que todos consumidores executariam a tarefa em um tempo muito próximo.

Figura 41 – Estatísticas disponíveis na interface *web* do *broker* durante o processo de reconstrução 3D, utilizando quatro consumidores.

The figure shows two tables, (a) and (b), representing statistics from a broker's web interface during a 3D reconstruction process using four consumers.

	State	Total	Ready	Unacked	In memory	Persistent	
Consumers	4	Messages (?)	10	6	4	10	10
Consumer utilisation (?)	0%	Message body bytes (?)	18MB	11MB	7.0MB	18MB	18MB
		Process memory (?)	34kB				

	State	Total	Ready	Unacked	In memory	Persistent	
Consumers	4	Messages (?)	2	0	2	2	2
Consumer utilisation (?)	0%	Message body bytes (?)	3.5MB	0B	3.5MB	3.5MB	3.5MB
		Process memory (?)	20kB				

Fonte: Produção do próprio autor.

A vantagem de se utilizar essa infraestrutura para serviços de processamento é a gerência feita com as requisições enviadas. A mensagem com a requisição sai do *broker* apenas quando recebe a resposta, e o consumidor responde que finalizou aquela requisição. Dessa forma, se o *broker* detectar que o consumidor perdeu conexão, a mensagem volta para a fila e fica esperando para ser enviada para outro consumidor. Isso garante que os pedidos não vão se perder caso algum computador que possua instâncias do serviço desligue ou perca conexão com a rede. Além disso, o número de consumidores pode ser escalado de acordo com a demanda requisitada. Esse processo pode ser feito facilmente quando há um serviço sendo executado através de um *container* e utiliza-se alguma ferramenta de orquestração.

5 CONCLUSÕES E TRABALHOS FUTUROS

A proposta deste trabalho foi utilizar o conceito de Internet das coisas para desenvolver a arquitetura de um espaço inteligente baseado em visão computacional. Em termos de flexibilidade, é possível afirmar que o sistema implementado neste projeto traz uma melhoria significativa quando comparado ao desenvolvido em (RAMPINELLI, 2014). Com o sistema atual, é possível inserir um novo dispositivo à infraestrutura sem a necessidade de grandes alterações na infraestrutura. Ao adicionar uma nova câmera no espaço inteligente, por exemplo, seu uso ocorre de modo transparente: basta que o usuário tenha conectividade com o *broker* e conheça o ID da nova câmera para usá-la de modo semelhantes ao usado pelas câmeras antigas.

As aplicações que se deseja realizar neste espaço inteligente compreendem áreas como visão computacional e robótica móvel. Tais aplicações têm por requisito a utilização de múltiplas câmeras e robôs, por exemplo. Vale ressaltar que, para utilizar os imagens capturadas pelas câmeras do espaço inteligente em aplicações de visão computacional, deve haver um sincronismo entre elas. Para realizar o sincronismos das câmeras, foi desenvolvido um serviço, cujos resultados dos testes foram apresentados na Seção 4.3. Os resultados desses testes mostraram que o serviço de sincronismo foi capaz de manter as câmeras sincronizadas pelo tempo de até uma hora, sendo o erro de sincronismo acumulado de aproximadamente 33 ms. Desse modo, é possível concluir que esse serviço é capaz de suprir os requisitos de aplicações desse tipo, dadas as limitações de tempo de exposição discutidas na Seção 4.3.

Para aplicações de controle utilizando realimentação visual no espaço inteligente, o tempo máximo de envio de informações para o robô deve ser de 100 ms, uma vez que esse é o tempo de atualização dos sinais de controles do robô móvel utilizado. Desse modo, a latência de recebimento das imagens pela aplicação que as usa deve ser menor que esse tempo de atualização. Os testes realizados mostraram que a infraestrutura desenvolvida tem latência abaixo de 100 ms, mesmo quando 10 aplicações são executadas simultaneamente utilizando imagens capturadas em resolução máxima. Assim, é possível verificar que a infraestrutura desenvolvida atende satisfatoriamente os requisitos de latência para essas aplicações.

A infraestrutura também mostrou-se capaz de prover serviços de processamento distribuído como o apresentado na Seção 3.2.3.3. O modelo utilizado oferece vantagens para esse tipo de serviço, uma vez que as requisições ficam armazenadas no *broker* e, caso algum provedor de serviço falhe, essa requisição é enviada para outro que esteja disponível. Tal característica fornece confiabilidade para o usuário. Além disso, o uso de *containers* para prover esses serviços torna o sistema mais flexível, sendo possível alocar recursos de maneira fácil e

rápida.

Apesar dos ganhos relevantes que essa nova infraestrutura trouxe ao espaço inteligente, é importante ressaltar que algumas melhorias podem ser realizadas. Vale destacar as seguintes propostas de trabalhos futuros:

- Incorporar à infraestrutura dispositivos que utilizam MQTT, ampliando assim a quantidade de tipos de dispositivos que podem ser incorporados. Para isso, deve-se buscar uma alternativa para implementar serviços utilizando esse protocolo, uma vez que o MQTT não possui em seu cabeçalho um campo para inserir o nome da fila para receber a resposta da requisição. Dessa maneira, será possível que serviços e aplicações que se comunicam com AMQP possam utilizar serviços que usam MQTT.
- Melhorar o serviço de sincronismo, fazendo com que haja uma checagem periódica do erro, corrigindo-o quando possível, e informando para a aplicação o estado em que se encontra o sincronismo.
- Incorporar novas opções de filtragem para o serviço de busca de entidades, para que seja possível por exemplo, buscar por câmeras instaladas em um determinado local, ou ainda, por robôs que possuam um sensor específico.
- Implementar uma interface *web* para o desenvolvedor, na qual ele possa listar as entidades instaladas na infraestrutura, selecionar o que deseja utilizar em sua aplicação, e então fazer o *download* dos arquivos necessários para o desenvolvimento.
- Realizar o cadastro de entidades, descrevendo os recursos e serviços providos por ela, e entregando-a um ID único, de tal forma que se essa entidade for removida da infraestrutura, uma nova não pode receber esse ID.
- Adicionar uma linguagem de descrição para tornar o processo de inserção de novas entidades e serviços ao espaço inteligente mais fácil.
- Implementar o mecanismo de gerência de acesso aos recursos e serviços das entidades, para evitar problemas de concorrências.

REFERÊNCIAS BIBLIOGRÁFICAS

- AHN, J. H. et al. Human position estimation in Intelligent Space for an active information display. In: *Control, Automation and Systems (ICCAS), 2011 11th International Conference on*. [S.l.: s.n.], 2011. p. 1497–1500. ISSN 2093-7121.
- ALLETTO, S. et al. An indoor location-aware system for an iot-based smart museum. *IEEE Internet of Things Journal*, v. 3, n. 2, p. 244–253, April 2016. ISSN 2327-4662.
- ALVI, S. A. et al. Internet of multimedia things: Vision and challenges. *Ad Hoc Networks*, v. 33, p. 87 – 111, 2015. ISSN 1570-8705. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S1570870515000876>>.
- ATZORI, L.; IERA, A.; MORABITO, G. The internet of things: A survey. *Comput. Netw.*, Elsevier North-Holland, Inc., New York, NY, USA, v. 54, n. 15, p. 2787–2805, out. 2010. ISSN 1389-1286. Disponível em: <<http://dx.doi.org/10.1016/j.comnet.2010.05.010>>.
- BASSI, A. et al. *Enabling Things to Talk - Designing IoT solutions with the IoT Architectural Reference Model*. New York: Springer, 2013.
- BELLAGENTE, P. et al. Adopting iot framework for energy management of smart building: A real test-case. In: *Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI), 2015 IEEE 1st International Forum on*. [S.l.: s.n.], 2015. p. 138–143.
- BRADSKI, G. *Dr. Dobb's Journal of Software Tools*, 2000.
- BROOKS, R. A. The Intelligent Room project. In: *Proceedings of the 2nd International Conference on Cognitive Technology (CT '97)*. Washington, DC, USA: IEEE Computer Society, 1997. (CT '97), p. 271–276.
- CARMO, A. P. do et al. Ontologia das coisas para espaços inteligentes baseado em visão computacional. In: *Seminário de Pesquisa em Ontologias do Brasil*. [S.l.: s.n.], 2016.
- CASAGRAS. *RFID and the Inclusive Model for Internet of Things*. 2009.
- CELESTI, A. et al. Exploring container virtualization in iot clouds. In: *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*. [S.l.: s.n.], 2016. p. 1–6.
- Cisco Systems. *Cisco 100 Series Unmanaged Switches - Datasheet*. 2015. Disponível em: <http://www.cisco.com/c/en/us/products/collateral/switches/small-business-100-series-unmanaged-switches/datasheet_C78-582017.pdf>.
- Dell Computadores do Brasil Ltda. *OptiPlex 9020 - Technical Specification*. 2015. Disponível em: <<https://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/optiplex-9020-micro-technical-spec-sheet.pdf>>.
- ELSON, J.; ESTRIN, D. Time synchronization for wireless sensor networks. In: *In Proceedings of the 2001 International Parallel and Distributed Processing Symposium (IPDPS)*. [S.l.: s.n.], 2001.

- ENGELS, D. W. et al. The networked physical world: An automated identification architecture. In: *In Proceedings of the Second IEEE Workshop on Internet Applications*. [S.l.: s.n.], 2002.
- EVANS. *The Internet of Things - How the Next Evolution of the Internet Is Changing Everything*. 2011.
- GRIECO, L. et al. IoT-aided robotics applications. *Comput. Commun.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 54, n. C, p. 32–47, dez. 2014. ISSN 0140-3664. Disponível em: <<http://dx.doi.org/10.1016/j.comcom.2014.07.013>>.
- HASHIMOTO, H. Intelligent space: Interaction and intelligence. *Artificial Life and Robotics*, v. 7, n. 3, p. 79–85, 2013. ISSN 1614-7456.
- JAIN, N.; CHOUDHARY, S. Overview of virtualization in cloud computing. In: *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*. [S.l.: s.n.], 2016. p. 1–4.
- JIN, J. et al. An information framework for creating a smart city through internet of things. *IEEE Internet of Things Journal*, v. 1, n. 2, p. 112–121, April 2014. ISSN 2327-4662.
- KILJANDER, J. et al. Semantic interoperability architecture for pervasive computing and internet of things. *IEEE Access*, v. 2, p. 856–873, 2014. ISSN 2169-3536.
- LINKSYS. *Linksys EA-Series - Userr Guide*. 2013. Disponível em: <http://downloads.linksys.com/downloads/userguide/EA-Series_UG_Full_3425-00125D_EN_FR-CA_Web.pdf>.
- LITOS, G.; ZABULIS, X.; TRIANTAFYLLODIS, G. Synchronous image acquisition based on network synchronization. In: *2006 Conference on Computer Vision and Pattern Recognition Workshop (CVPRW'06)*. [S.l.: s.n.], 2006. p. 167–167. ISSN 2160-7508.
- LIU, N. et al. *Control system for several rotating mirror camera synchronization operation*. 1997. 695-699 p.
- LOSADA, C. et al. Multi-camera sensor system for 3d segmentation and localization of multiple mobile robots. *Sensors*, v. 10, n. 4, p. 3261–3279, 2010.
- MOBILEROBOTS, A. *Pioneer 3 - AT - Manual*. 2011. Disponível em: <<http://www.mobilerobots.com/Libraries/Downloads/Pioneer3AT-P3AT-RevA.sflb.ashx>>.
- MORIOKA, K.; LEE, J.-H. L. J.-H.; HASHIMOTO, H. Intelligent Space for Human Centered Robotics. *Advances in Service Robotics, Ho Seok Ahn (Ed.)*, InTech, 2002.
- PENIAK, P.; FRANEKOVA, M. Open communication protocols for integration of embedded systems within industry 4. In: *Applied Electronics (AE), 2015 International Conference on*. [S.l.: s.n.], 2015. p. 181–184. ISSN 1803-7232.
- PENTLAND, A. P. Smart rooms. *Scientific American*, v. 274, n. 4, p. 54–62, 1996.
- PIRES, P. F. et al. *Minicursos - XXXIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. [S.l.: s.n.], 2015.

- PIZARRO, D. et al. Localization of mobile robots using odometry and an external vision sensor. *Sensors*, v. 10, n. 4, p. 3655–3680, 2010.
- PRČIĆ, V. L.; ĐUREK, M.; ŠIMUNIĆ, D. Efficient wireless signal emitting sensor distribution within an environment modified for the needs of the aging population. In: *Information Communication Technology Electronics Microelectronics (MIPRO), 2013 36th International Convention on*. [S.l.: s.n.], 2013. p. 484–489.
- PROVINCE, S. Intelligent Space System Oriented to the Home Service Robot. p. 50–54, 2012.
- QUEIROZ, F. M. de et al. Localização e Guiagem de um Robô Móvel em um Espaço Inteligente. In: *Anais do XII Simpósio Brasileiro de Automação Inteligente (SBAI'2015)*. [S.l.: s.n.], 2015.
- RAMPINELLI, M. *Calibração de uma Rede de Câmeras e Controle de um Robô Móvel em um Espaço Inteligente*. Tese (Doutorado) — Universidade Federal do Espírito Santo, Vitória - ES, 2014.
- RAMPINELLI, M. et al. Implementation of an intelligent space for localizing and controlling a robotic wheelchair. In: *Biosignals and Biorobotics Conference (BRC), 2012 ISSNIP*. [S.l.: s.n.], 2012. p. 54–58.
- RAMPINELLI, M. et al. An Intelligent Space for Mobile Robot Localization Using a Multi-Camera System. *Sensors*, v. 14, n. 8, p. 15039–15064, 2014.
- RESEARCH, P. G. *BlackFly - Datasheet*. 2015. Disponível em: <<https://www.ptgrey.com/support/downloads/10131>>.
- RICHARDSON, F. H. Importance of Synchronizing Taking and Camera Speeds. *Transactions of the Society of Motion Picture Engineers*, v. 7, n. 17, p. 117–123, 1923. ISSN 0096-6460.
- ROMER, K. *Time Synchronization and Localization in Sensor Networks*. Tese (Doutorado) — Swiss Federal Institute of Technology Zurich (ETH Zurich), Zurich, 2005.
- SILVA, L. d. A.; LUBE, J. G. P.; VASSALLO, R. F. Aproximação Planar por Partes para Reconstrução 3D Densa. In: *Anais do XXI Congresso Brasileiro de Automática (CBA'2016)*. [S.l.: s.n.], 2016.
- SVOBODA, T.; HUG, H.; GOOL, L. V. Pattern recognition: 24th dagm symposium zurich, switzerland, september 16–18, 2002 proceedings. In: _____. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. cap. ViRoom — Low Cost Synchronized Multicamera System and Its Self-calibration, p. 515–522.
- WANT, R.; FALCAO, V.; GIBBONS, J. The Active Badge Location System. *ACM Transactions on Information Systems*, v. 10, p. 91–102, 1992.
- WEISER, M. The computer for the 21st century. *Scientific American*, v. 265, n. 3, p. 66–75, set. 1991.
- WEISER, M. Some computer science issues in ubiquitous computing. *Communications of ACM*, v. 36, n. 7, p. 74–83. In Special Issue, Computer-Augmented Environments, Jul 1993.

YAN, J. et al. Video synchronization via space-time interest point distribution. In: *In Proceedings of Advanced Concepts for Intelligent Vision Systems*. [S.l.: s.n.], 2004.

Apêndices

APÊNDICE A – ESTRUTURA DO CÓDIGO

O desenvolvimento deste trabalho foi feito no sistema operacional *Ubuntu 14.04*. Boa parte da implementação foi feita em C++, com exceção de um serviço que foi desenvolvido em *JavaScript* para simplificar sua implementação, e para também mostrar que, seguindo os padrões adotados, pode-se utilizar diferentes linguagens de programação no espaço inteligente.

A estrutura de pastas mostrada na Figura 42 foi utilizada para organizar as implementações. Dentro da pasta “*cpp*” encontra-se quase toda implementação feita e, na pasta “*json*”, encontra-se o serviço que foi desenvolvido em *JavaScript*. Na pasta “*ispace*” temos a biblioteca de comunicação que está detalhada no Apêndice B. Em “*utils*” estão os *headers* com as estruturas de dados utilizadas durante a implementação. Em “*services*” temos a implementação de alguns serviços de suporte à infraestrutura. Nas pastas “*cameras*” e “*robots*” estão os *drivers* das câmeras e dos robôs respectivamente, bem como seus *gateways* e as interfaces para os desenvolvedores de aplicações. Por fim, na pasta “*examples*” encontram-se exemplos de utilização das bibliotecas desenvolvidas.

Figura 42 – Árvore de diretórios da implementação do projeto.

```

| -- cpp
|   | -- cameras
|   | -- examples
|   | -- ispace
|   | -- robots
|   | -- services
|   | -- utils
|   `-- json
|     `-- services
  
```

Fonte: Produção do próprio autor.

APÊNDICE B – BIBLIOTECA DE COMUNICAÇÃO

A partir de uma implementação do AMQP em C++ disponível em <<https://github.com/alanxz/SimpleAmqpClient>>, foram feitas abstrações para tornar o uso mais amigável. Para publicar e consumir os recursos das entidades virtuais, foram criadas as classes **Publisher** e **Subscriber**. Já para expor e utilizar os serviços, foram criadas **ServiceProvider** e **ServiceClient**. Além disso, foi definido o tipo de dado **Channel**, que representa uma conexão com o *broker*, e também funções de serialização e deserialização de dados.

O processo de serialização é feito com o objetivo de converter as estrutura de dados para um formato no qual pode ser transmitido, e que, ao passar pelo processo de deserialização, a estrutura de dados é reconstruída. Utilizar essas ferramentas são fundamentais quando deseja-se conversar com aplicações desenvolvidas em diferentes linguagens de programação. Nessa implementação, foi utilizada a biblioteca *MessagePack*, que está disponível em uma grande quantidade de linguagens. Mais detalhes em <<http://msgpack.org/>>.

O tipo de dado **Channel** é retornado pela função *connect*. Sua implementação é mostrada à seguir. Deve receber como parâmetro o endereço IP do *broker*, sua porta, que por padrão é a 5672, além de uma estrutura de dados *Credentials*, que é composta pelo usuário e senha, além da rede que será utilizada no **broker**.

```
1 Channel connect(const string& ip ,
2                     const uint16_t& port ,
3                     const Credentials& credentials );
```

A classe **Publisher** é responsável por publicar um dado em um determinado *exchange*, com um determinado *binding*. Seu construtor está definido à seguir. Esse deve receber um **Channel**, o nome da entidade, bem como o *exchange* que a mensagem será enviada, que por padrão é o *data*. Como mostrado na sessão 3.2, definiu-se que o nome da entidade é dado por “*classe.ID*”.

```
1 Publisher(Channel channel ,
2             const string& entity ,
3             const string& exchange = "data" );
```

Essa classe possui a função *publish*, que é responsável por publicar um dado em um tópico específico. Seu protótipo está definido à seguir. Essa função é um *template*, podendo receber como parâmetro *data* qualquer estrutura de dados que seja serializável. Além disso, deve-se informar o tópico que será publicado aquele dado. Vale ressaltar que no *broker*, o nome do tópico que será criado será composto por “*classe.ID.topic*”. Isso será importante na hora de utilizar a classe **Subscriber**.

```

1 template <typename DataType>
2 void publish(const DataType& data, const string& topic);

```

A classe **Subscriber** complementa o par *pub/sub* junto com a classe **Publisher**. Seu construtor está definido à seguir. Esse também deve receber um **Channel**. O parâmetro *key* corresponde ao tópico que deseja-se consumir o dado. No caso, deve-se informar o nome completo do tópico: “*classe.ID.topic*”. Pode-se também informar o *exchange*, que por padrão é o *data*.

```

1 Subscriber(Channel channel,
2           const string& key,
3           const string& exchange = "data");

```

Essa classe possui a função *consume* mostrado à seguir, que é definida como *template*, e recebe como parâmetro o tipo de dado a ser recebido. Além disso, pode-se definir um *timeout* em milissegundos. Caso não seja definido, a função ficará bloqueada até que algum dado chegue no consumidor.

```

1 template <typename DataType>
2 bool consume(DataType& data, int timeout = -1);

```

Um exemplo onde deseja-se publicar a imagem de uma câmera pode ser visto à seguir. Inclusão de cabeçalhos e outros detalhes do código foram omitidos. Nesse exemplo, a imagem da câmera da entidade virtual de nome “camera.0” é publicada no tópico “camera.0.frame”.

```

1 // Create a channel
2 Channel channel = is::connect("localhost", 5672, {"admin", "admin", "/"});
3 // Create a publisher
4 Publisher publisher(channel, "camera.0");
5 /*
6  * User code to grab a frame from the camera here!
7  */
8 // Publish a frame on topic "camera.0.frame"
9 publisher.publish(frame, "frame");

```

Para consumir as imagens publicadas por essa câmera, segue o seguinte exemplo:

```

1 // Create a channel
2 Channel channel = is::connect("localhost", 5672, {"admin", "admin", "/"});
3 // Create a subscriber
4 Subscriber subscriber(channel, "camera.0.frame");
5 // Create an image
6 Image image;
7 // Blocking call consume
8 subscriber.consume(image)

```

```

9 /*
10  * Do anything with the image here!
11 */

```

A classe **ServiceProvider** é responsável por expor serviços que podem estar associados à uma entidade ou não. Seu construtor está apresentado à seguir, ele deve receber como parâmetro um **Channel**, um identificador do serviço (*service*), que quando for associado à uma entidade será o nome dela, e quando não for, pode ser a um grupo à qual aquele serviço pertence. Serão mostrados exemplos que deixarão isso mais claro. O *exchange* também deve ser informado, e como padrão utiliza-se *services*.

```

1 ServiceProvider(Channel channel,
2           const string& service,
3           const string& exchange = "services");

```

Essa classe possui a função *expose*, que tem o papel de expor o serviço desejado. Seu protótipo é apresentado em seguida. Como parâmetros, recebe o nome do serviço (*key*), que será concatenado com o nome da entidade virtual, criando o padrão “*service.key*”, além de um *handle* para a função que será chamada quando o serviço for requisitado.

```
1 void expose(const string& key, handle_t handle);
```

Além disso, a classe **ServiceProvider** possui as funções *listen* e *listen_sync*. Na primeira, cria-se uma nova *thread* onde o serviço ficará esperando por uma requisição, já na segunda, a própria *thread* onde o serviço foi criado será utilizada, bloqueando-a naquele ponto.

Para complementar, temos a classe **ServiceCliente**, necessária para utilizar os serviços expostos. Seu construtor está apresentado à seguir, exigindo apenas um **Channel** como parâmetro, e, se necessário, um *exchange* diferente do padrão.

```

1 ServiceClient(Channel channel,
2                 const string& exchange = "services");

```

O primeiro passo para se utilizar um serviço é enviar uma requisição à ele. Para isso, a classe **ServiceCliente** possui a função *request*, apresentada à seguir. Essa é um *template*, e recebe como parâmetros o nome do serviço e o dado a ser enviado para ele. Essa então retorna um ID, para que seja feita a correlação com a resposta recebida.

```

1 template <typename DataType>
2 uint request(const string& service, const DataType& data);

```

Após o pedido ser feito, deve-se esperar por uma resposta. Para isso, foi implementada da função *receive*, mostrada à seguir. Pode-se passar um único ID ou uma lista. Deve-se

também definir um *timeout* em milissegundos, esse é importante pois pode ser que a requisição chegue na fila do provedor de serviços no *broker* e não haja nenhuma instância do provedor de serviços disponível no momento, fazendo com que a resposta demore, ou até mesmo não seja respondida, dessa forma, a função ficaria bloqueada.

```
1 set<uint> receive(const uint& id, const uint& timeout);
2 set<uint> receive(const set<uint>& ids, const uint& timeout);
```

Caso a função *receive* retorne algum ID, realiza-se então a correspondência com o ID do pedido enviado, e então, consome-se a resposta utilizando a função *consume*, apresentada à seguir. Essa é um *template*, e recebe como parâmetro o ID que deseja-se consumir, além do tipo de dado que o serviço envia como resposta. Seu retorno é um booleano que indica se o ID desejado recebeu resposta.

```
1 template <typename DataType>
2 bool consume(const uint& id, DataType& data)
```

A seguir, um exemplo de utilização, onde é exposto um serviço de uma entidade. No caso, consideramos uma câmera que possui entidade virtual de nome “camera.0”, e expõe um serviço de mudança taxa de captura de nome “fps”.

```
1 // Create a channel
2 Channel channel = is::connect("localhost", 5672, {"admin", "admin", "/"});
3 // Create a service provider
4 ServiceProvider server(channel, "camera.0");
5 // Expose FPS service
6 server.expose("fps", [] (auto& service) {
7     float fps;
8     // Get client request
9     service.request(fps);
10    /*
11     * Set fps here!
12     */
13    // Send reply
14    service.reply(reply);
15});
```

O serviço de mudança de taxa de captura fica então exposto no tópico “camera.0.fps”, e pode ser utilizado como mostrado no exemplo à seguir.

```
1 // Create a channel
2 Channel channel = is::connect("localhost", 5672, {"admin", "admin", "/"});
3 // Create a service client
4 ServiceClient client(channel);
5 // Request fps = 10.0
```

```
6 auto req_id = client.request("camera.0.fps", 10.0);
7 // Waiting response with 1000ms timeout
8 auto rec_id = client.receive(req_id, 1000);
9 // Get response
10 int response;
11 if (req_id == rec_id) {
12     client.consume(req_id, response);
13 }
```