Tolerância a falhas em software a partir de arquitetura de microsserviços no contexto de Internet das Coisas

Software fault tolerance using microservices architecture in the Internet of Things

Universidade Católica de Pernambuco (UNICAP) – Icam-Tech

João Vitor Coelho Tavares

Prof. Sérgio Murilo Maciel Fernandes

Resumo: *A Internet das Coisas é um dos conceitos mais discutidos no estado da arte da "Indústria 4.0". Smart objects, presentes em ambientes residenciais, industriais e de pesquisa, cada vez mais configuram ecossistemas ao invés de dispositivos isolados. Por mais que aparelhos sejam o foco das aplicações, a disponibilidade de software é um fator decisivo de sucesso ou falha. Nesse quesito, o conceito de tolerância a falhas desempenha um papel importante, permitindo que sistemas operem sob condições defeituosas, e tendo como suporte a modularização trazida pela implementação de arquiteturas de microsserviços.*

Palavras-chave: Internet of Things, Internet das Coisas, IoT, tolerância a falhas em software, arquitetura de microsserviços.

Abstract: *The Internet of Things is one of the most important concepts on the state of art of the "Industry 4.0". Smart objects, found in households, industry and research environments, are building technological ecosystems and are less seen as individual devices alone. Even though the applications are mostly device-driven, software disponibility is a crucial factor for success or failure. In this regard, software fault tolerance has a significant role, allowing that systems continue to work under faulty conditions, while the modularization brought by microservice architectures supports it.*

Keywords: Internet of Things, IoT, fault tolerance, software fault tolerance, microservice architecture.

1. Introdução

A Terceira Revolução Industrial, também chamada de "Revolução Digital", data do início da década de 1950, e foi responsável pelo constante desenvolvimento e aprimoramento

dos computadores da época. Estes avanços foram decisivos para que, cerca de 70 anos depois, a tecnologia encontre-se no estado da arte atual: a Quarta Revolução Industrial, também conhecida por "Indústria 4.0", com foco na eficiência e produtividade dos processos.

Um dos conceitos presentes nessa nova realidade é o de *Internet of Things* (Internet das Coisas, abreviado como IoT nesta pesquisa), que consiste na conexão dos chamados *smart objects* (objetos inteligentes – artefatos tecnológicos cuja aparência remete à itens comuns do dia-a-dia), *smart sensors* (sensores inteligentes, geralmente de uso específico e capazes de captar informações do meio físico [1][2] para compartilhar com algum hardware externo) ou uma combinação de ambos à servidores ou outros objetos presentes em uma rede. Como exemplo prático, há o uso de drones para mapeamento ou monitoramento de determinadas áreas em tempo real ou o uso de dispositivos vestíveis (tais como *smartwatches*) para acompanhamento de saúde de indivíduos através de sensores embutidos.

O conjunto desses dispositivos presentes na mesma aplicação (denominando-se ecossistema) pode ter diversas finalidades, tais como: envio, recebimento, troca e armazenamento de dados, ou servir como controladores dentro de um ambiente definido, tanto em escala global quanto a nível local [2]. No entanto, mesmo que seja comum dar ênfase aos dispositivos ao invés de programas em IoT, a operabilidade, confiabilidade e disponibilidade do software ainda é de extrema importância. Dependendo do contexto da aplicação, períodos de indisponibilidade, por mais breve que sejam, podem ser fatores decisivos de sucesso ou falha, especialmente em aplicações críticas. Apesar de nenhum sistema ser vulnerável à falhas (lógicas e físicas), é possível contorná-las ou minimizar seu impacto no sistema através do uso das técnicas corretas.

Weber [3] cita diversos desafios atuais para sistemas computacionais, sendo um deles evitar, detectar e contornar *bugs* em projetos de hardware e software. Nesse quesito, o conceito de tolerância a falhas desempenha um papel importante, fazendo com o que o sistema consiga operar mesmo sob condições defeituosas, através de técnicas de redundância lógica ou física. Outro conceito que é capaz de auxiliar na superação dos desafios mencionados anteriormente são os microsserviços, comumente encontrados em aplicações *web*, tais como *Netflix*, *Amazon*, *Twitter*, dentre outras presentes no cotidiano de milhões de usuários. Ao modularizar uma aplicação em diversas partes ou plataformas que comunicam-se entre si, é possível evitar, ou minimizar, os impactos de um colapso total no sistema devido a um erro ou falha isolada.

A estrutura desta pesquisa é descrita a seguir. Na seção 2, é apresentado um resumo sobre IoT. Na seção 3 é apresentado um resumo sobre tolerância a falhas em hardware e

software, bem como algumas das técnicas utilizadas. Na seção 4, é apresentado um resumo sobre microsserviços. Na seção 5, é descrita a metodologia utilizada, bem como a categorização dos artigos utilizados de referência. No item 5.1 da seção 5, são apresentadas duas provas de conceito sobre os assuntos abordados ao longo deste trabalho, que consistem na simulação de sistemas que agem como microsserviços dentro de uma aplicação maior. Na seção 6, são apresentados resultados e discussões. Na seção 7, é feito o fechamento deste trabalho com a conclusão.

Esta pesquisa tem por finalidade investigar o estado da arte e aplicações de arquitetura de microsserviços em IoT, bem como explicitar os benefícios da inclusão de conceitos de tolerância a falhas em aplicações através de uma prova de conceito implementada na plataforma virtual *Tinkercad*.

2. *Internet of Things* (Internet das Coisas)

Definido pela primeira vez em 1999 para falar sobre sensores RFID (*Radio Frequency Identification* – sensores por rádio frequência) dentro do contexto da cadeia de suprimentos [4], IoT é um amplo conceito que não possui definição formal segundo alguns autores. Atualmente, é utilizado para designar *smart objects* interconectados, através de uma rede (interna ou externa) [5], com o intuito da troca de dados para obter-se informações acerca de alguma aplicação específica. Tecnologias que podem ser encaixadas nesse conceito são, mas não se limitam a: sensores de uso específico, muitas vezes por radiofrequência, tecnologias utilizadas com outros propósitos (como câmeras de segurança, travas de portas, lâmpadas, dentre outros) e dispositivos *wearables* (vestíveis, tais como *smartwatches*, óculos inteligentes, etc).

Os componentes que permitem a comunicação entre esses aparelhos são [6]: método de comunicação local (o qual define por qual meio os aparelhos vizinhos se comunicam), protocolo de aplicação (estrutura que define como os dados devem ser transportados), *gateways* (portais que conectam, de fato, o dispositivo à internet), servidores (que atuam recebendo e gerenciando os dados, comumente formando as centrais de dados) e interface gráfica do usuário. No entanto, existe a possibilidade dos dados serem processados localmente, eliminando o papel do servidor na listagem dos componentes [7][8].

2.1 Coleta e armazenamento de dados em aplicações IoT

Os dados coletados por esses dispositivos partem do meio físico, são heterogêneos e geralmente provém de múltiplas fontes, podendo variar desde relatórios de sensores (tais como medição de temperatura ou proximidade entre objetos) até dados sensíveis de usuários (como geolocalização, identificadores de dispositivos, nomes ou outros que permitam o rastreamento de indivíduos). Podem ser armazenados localmente, em névoa ou em nuvem, também sendo possível seguir uma estratégia que englobe duas ou mais alternativas. Nesse caso, é necessário levar em consideração fatores como volume de dados a serem armazenados, disponibilidade de rede, consumo de energia [9][10] bem como a capacidade de armazenamento disponível.

Após coletados, os dados podem ser classificados em estruturados (como JSON ou XML), semi-estruturados (que possuem marcadores para separar elementos, mas que não estão associados a regras de estruturas formais de modelos de dados) ou não estruturados (que não possuem organização, sendo de difícil interpretação), não existindo, atualmente, uma forma universal de tratamento. É necessário levar em consideração a saída de cada sensor e as necessidades da aplicação em que o dispositivo se encontra.

Por se tratarem de aparelhos leves e de pequeno porte, é comum que esses tipos de dispositivos disponham de memória *flash* interna (geralmente na escala de *megabytes*) ou entradas para armazenamento físico externo, como cartões micro SD.

2.2 Processamento de dados em aplicações IoT

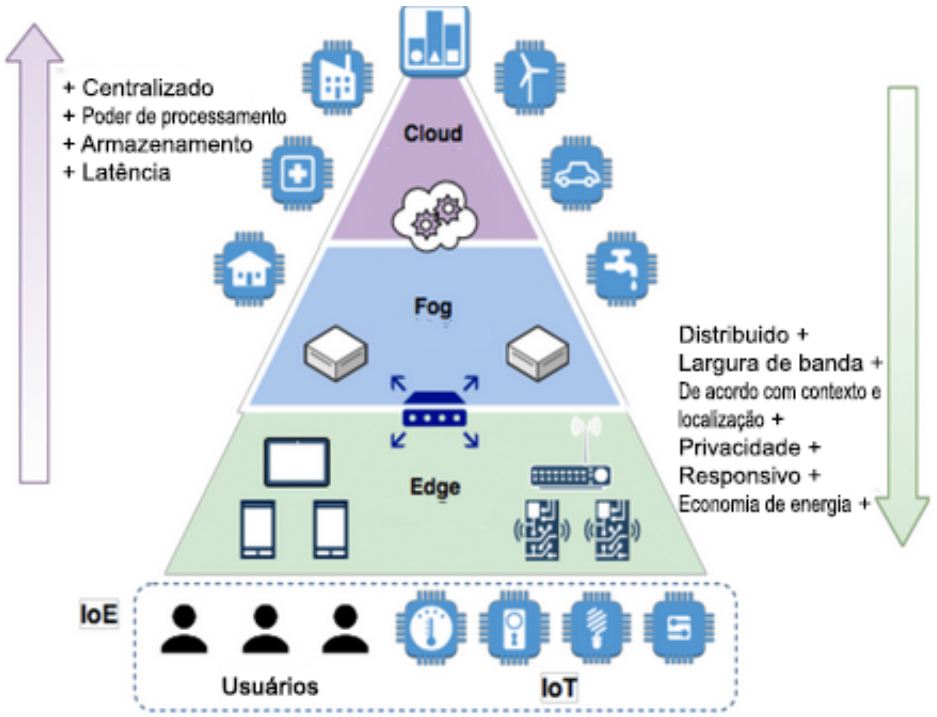
Atualmente, existem três formas de processamento de dados para aplicações em IoT [8]: *edge* (processamento local), *fog* (processamento em névoa) e *cloud* (processamento em nuvem). É possível haver mais de uma forma de processamento na mesma aplicação. Além disso, cada uma das alternativas descritas acima possui suas particularidades, definidas a seguir:

- ***Edge computing*** (computação em ponta ou em margem): também chamado de computação local. Refere-se ao processamento local dos dados coletados pelo dispositivo. Somente os resultados são transmitidos pela rede, o que diminui o consumo de banda e latência. Não é recomendado para aplicações com localização e volume de dados dinâmico. Alguns dos casos de uso para este tipo de processamento são carros autônomos, monitoramento inteligente de tráfego e *smart grids* (redes elétricas inteligentes) [11].
- ***Fog computing*** (computação em névoa): infraestrutura descentralizada, onde os recursos estão entre o dispositivo na margem e o servidor na nuvem. Pode atuar como

suporte para dispositivos da ponta, através de ações como processamento, armazenamento (mesmo que temporário) e serviços de rede com latência mais baixa do que quando comparado à nuvem [12]. Um exemplo de caso de uso é o de gerenciamento inteligente de resíduos. Através de sensores instalados em lixeiras, cidades podem monitorar o volume e a necessidade de recolher o lixo, permitindo que os caminhões de coleta somente façam a rota necessária (economizando na emissão de carbono), além de poder monitorar questões de saúde pública antes que tornem-se um problema [13].

- ***Cloud computing*** (computação em nuvem): servidores centralizados, em centrais que não necessariamente estão próximas do usuário ou dispositivo e necessitam ser acessados pela internet. Podem realizar o processamento dos dados enviados pelos dispositivos de ponta ou névoa, além de comumente possuírem opções de armazenamento. A *Amazon Alexa*, assistente virtual por voz da empresa norte-americana *Amazon*, obedece aos comandos do usuário através do NLP (*Natural Language Processing* – processamento de linguagem natural) nos servidores na nuvem da *Amazon*, devido à falta de poder computacional do aparelho [14].

Figura 1: Representação visual das vantagens e desvantagens de cada forma de processamento IoT.



Fonte: Pacheco, Alberto & Flores, Alex & Trujillo, Edgar & Cano, Pablo. (2018). A Smart Classroom Based on Deep Learning and Osmotic IoT Computing. Adaptação para o português de autoria própria.

2.3 Consumo de energia em aplicações IoT

O consumo energético em dispositivos e aplicações IoT está relacionado com a finalidade da aplicação em questão, bem como os dispositivos presentes no ecossistema, e deve ser planejado de acordo. No exemplo citado anteriormente sobre a utilização de computação de ponta para automóveis autônomos, espera-se que haja uma dependência constante de energia, visto que, sem a mesma, não há o funcionamento do carro. No entanto, espera-se que dispositivos de menor porte, tais como sensores RFID, operem continuamente através de baterias de pequeno porte [15], salvo nos casos de sensores passivos.

Os dispositivos podem possuir alimentação interna, proveniente de fontes que partem desde baterias com menos de 2v até baterias de íon lítio, capazes de prover cargas que, para estes tipos de aparelhos, podem durar dias. Alguns dispositivos, especialmente no caso de assistentes por voz, operam sob uma fonte externa contínua de alimentação. Com os avanços nos estudos da projeção de cidades inteligentes, uma fonte alternativa para esses dispositivos é a energia coletada através de *energy harvesting* (colheita de energia), que consiste em capturar energia de uma ou mais fontes renováveis [15][16]. No entanto, Georgiu et al. [17] afirmam que energias renováveis não são de alta dependabilidade, devido à volatilidade de seu fornecimento e o conceito de *energy budget*, que diz respeito ao balanço da energia solar que chega à Terra, mas depois retorna ao espaço, não podendo ser aproveitada por completo.

3. Tolerância a falhas

Segundo Weber [3], falhas são inevitáveis. Estas podem partir de fatores como defeitos em componentes físicos, erros de *design* ou implementação [18], mas não somente limitando-se a estes. Ao implementar a tolerância a falhas em sistemas, é possível minimizar o impacto causado por uma interrupção, permitindo que a aplicação continue operando, mesmo que de maneira mínima, dentro de condições defeituosas.

Ainda segundo Hammed et al. [19], sistemas tolerantes a falhas dividem-se em dois: os que, mesmo com o erro no momento da execução, precisam continuar a funcionar, e aqueles que, quando encontram uma falha, podem interromper a execução sem prejudicar o usuário final. Independente da camada onde se encontram, os erros são tratados com uma das seguintes técnicas: mascaramento ou detecção, localização e reconfiguração [5]. Sistemas

tolerantes a falhas utilizam de técnicas de redundância, o que significa ter duplicatas de um determinado artefato, de forma que o mesmo esteja disponível em caso de erro. Existem quatro tipos de redundância em sistemas computacionais: redundância de hardware, de software, de informação e tempo, dos quais os dois primeiros são objetos de estudo deste trabalho.

3.1 Métricas de tolerância a falhas em sistemas

Medições tradicionais de tolerância à falha em sistema procuram avaliar dois atributos: confiabilidade e disponibilidade. Baseiam-se também, para os cálculos que serão demonstrados posteriormente, que o sistema possui somente estados de disponível e indisponível, não havendo um meio termo entre os dois. Além disso, considera-se que a confiabilidade padrão de um sistema é representada por *R(t)* (confiabilidade em relação ao tempo), onde o sistema esteve disponível entre o intervalo [0, *t*]. Para softwares, a disponibilidade do sistema significa que a aplicação está de acordo com os requisitos previamente levantados.

Koren et al. [20] afirmam que o *Mean Time to Failure* (abreviado como MTTF – Tempo Médio para Falhas) e o *Mean Time Between Failures* (abreviado como MTBF – Tempo Médio entre Falhas) são duas métricas relacionadas à confiança de um sistema. Outras métricas, apresentadas por Weber [6], são *Mean Time to Repair* (abreviado como MTTR – Tempo Médio de Reparo) e *Mean Time Between Failures* (abreviado como MTBF – Tempo Médio entre Falhas). Ainda segundo Koren et al. [20], tem-se que a diferença entre o MTTF e MTBF representa o tempo médio para consertar o sistema em seguida da falha, obtendo-se então a equação 1:

$$MTBF = MTTF + MTTR \quad (1)$$

A disponibilidade de um sistema, representada por *A(t)*, pode ser calculada a partir da equação 2:

$$A = MTTF \div MTBF \quad (2)$$

Significa que um sistema estar disponível é a razão entre o tempo médio de falha (ou seja, o tempo *t* no intervalo [0, *t*] em que está disponível) e o tempo entre duas falhas

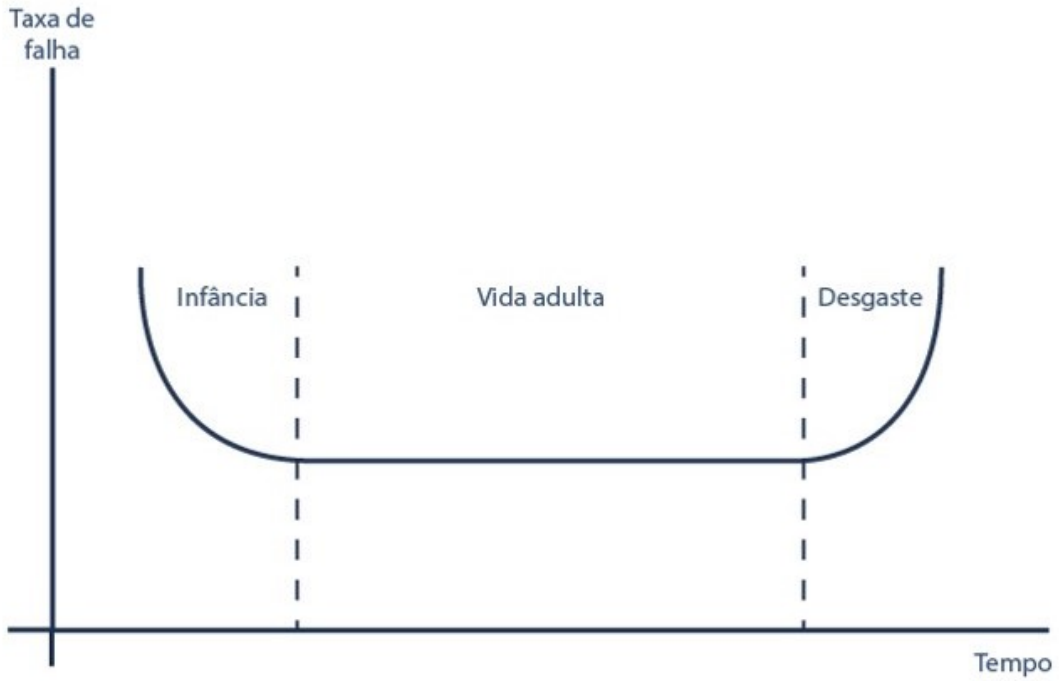
consecutivas. Como definido por Koren et al. [20], a disponibilidade de um sistema também pode ser interpretada como a chance de um sistema estar disponível, caso seja acessado em um determinado intervalo de tempo aleatório.

3.2 Tolerância a falhas em hardware

Atualmente, devido ao avanço de técnicas como controle de qualidade, melhoria na cadeia de produção, testes, *design*, etc., somente é necessário que hardwares mascarem as falhas através do uso de redundâncias [21]. A tolerância à falha em hardwares deve considerar o tempo médio de vida útil de um componente, comumente disponibilizada pelos fabricantes. Devem ser levados em consideração desgastes que podem ocorrer devido a temperatura do ambiente onde se encontra, formas de alimentação e a aplicação como um todo.

A taxa de defeitos de um componente é dada por falhas por unidade de tempo, e varia de acordo com o tempo médio de vida útil do componente [3][20]. A afirmação anterior pode ser representada através da *bathtub curve* (curva da banheira) a seguir:

Figura 2: Curva da banheira, que demonstra uma relação entre taxa de falhas de componente em relação à um intervalo de tempo



Fonte: NG Blog. Disponível em: <https://www.ngi.com.br/blog/curva-da-banheira/>. Acesso em: 23 nov. 2022.

A etapa da infância em um componente é a qual as falhas ocorrem devido a possíveis problemas de fabricação. Em seguida, na fase da vida adulta, a quantidade de falhas estabiliza em relação a anterior e passa a ser constante, dado que os componentes defeituosos não atravessaram a etapa anterior. Por fim, na fase de desgaste, a quantidade de falhas volta a crescer devido ao uso contínuo dos componentes e, consequentemente, desgaste natural causado por agentes externos. O impacto causado por outros fatores pode ser representado através da equação 3, assim como apresentado por Koren et al. [20]:

$$\lambda = \pi L \pi Q \times (C1 \pi T \pi V + C2 \pi E) \quad (3)$$

Onde:

- λ é a taxa de falha de um componente;
- πL é o fator de aprendizado, considerando se é uma tecnologia nova ou já estabelecida;
- πQ é referente à qualidade de produção do componente, presente entre o intervalo [0,25 , 25000];
- πT representa a temperatura à qual o componente é constantemente submetido (em Kelvin) e está no intervalo [0,1 , 1000];
- πV é o desgaste causado no componente pela voltagem de alimentação, e está no intervalo [1, 10];
- πE refere-se ao ambiente no qual o componente está inserido, sem intervalo mínimo precisamente definido em [21] e com intervalo máximo 13. No entanto, o exemplo apresentado, no intervalo de [0,4 (definido como muito baixo), 13], informa que o valor apresentado refere-se à uma sala com ar-condicionado;
- C1 e C2 são variáveis de complexidade, referindo-se a fatores como a quantidade de portas lógicas no *chip*. Esses valores podem ser comumente encontrados nos respectivos manuais de instrução.

Segundo Weber [3], as formas de redundância em hardware dividem-se em três:

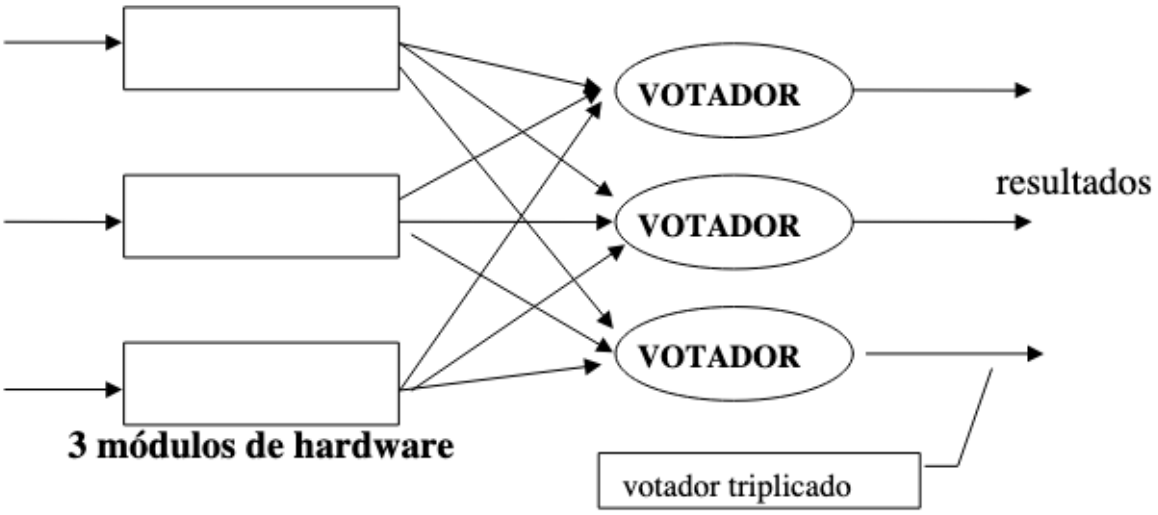
3.2.1 Redundância passiva ou estática

Consiste em mascarar as falhas. Não requer que o sistema execute ações, e também não indica ao usuário a ocorrência de erros. Todos os artefatos redundantes executam a

mesma atividade. Weber [3] ainda cita dois exemplos: *Triple Modular Redundancy* (abreviado como TMR nesta pesquisa – Tripla Redundância Modular) e *N-Modular Redundancy* (abreviado como NMR nesta pesquisa – Redundância Modular Múltipla).

A TMR é aplicada quando há três cópias do mesmo artefato, e o resultado é gerado a partir de um sistema de voto da maioria ou por uma média do resultado dos três, de forma que se um dos componentes falhar, os outros dois conseguem mascarar o erro. No entanto, uma falha no sistema de votação implica que o sistema é frágil [3], e pode sofrer uma falha completa [22]. Algumas das soluções para aumentar a confiança no sistema de votação são a implementação através de software, utilizar componentes de maior qualidade ou triplicar o votador.

Figura 3: esquema do TMR com votador triplicado.



Fonte: WEBER, Taisy Silva. Tolerância a falhas: conceitos e exemplos

Em relação ao sistema inicialmente proposto por Von Neumann, o sistema com triplo votador apresenta uma maior confiabilidade [23]. A NMR é a generalização do conceito de TMR, de forma que a TRM é um caso especial em NMR [3].

3.2.2 Redundância ativa ou dinâmica

Consiste na detecção, localização e recuperação da falha, empregando a substituição de módulos na aplicação. Costuma ser utilizada em sistemas que podem operar em estado de erro durante um curto período de tempo. Ao contrário da redundância passiva, esta requer ação do sistema. Além disso, consome menos recursos, visto que os mesmos só são requisitados por demanda, e oferece maior confiabilidade à longo prazo. Fu et al. [24] trazem

como objeto de estudo a implementação de redundância dinâmica para sistemas de direção ativa.

3.2.3 Redundância híbrida

Consiste na combinação da redundância ativa e passiva, e possui alto custo computacional. O IEEE [25] cita como exemplo de redundância dinâmica um sistema que pode mascarar falhas até um determinado limite, e em seguida parte para técnicas de redundância ativa.

3.3 Tolerância a falhas em software

Considerando os dois tipos de sistemas tolerantes a falhas descritos no item 4, os softwares podem se dividir em quatro subcategorias: (1) falho operacional, (2) falho-ativo, (3) falho-seguro e (4) alta disponibilidade, com suas definições a seguir:

1. ***Fail-Operational*** **(FO, falho-operacional)**: o sistema continua operando normalmente mesmo em situação de falha, de forma que seja imperceptível para o usuário. Sistemas críticos costumam pertencer a esta categoria.
2. ***Fail-Active*** **(FA, falho-ativo)**: a aplicação continua operando, mas com performance reduzida à um limiar aceitável.
3. ***Fail-Safe*** **(FS, falho-seguro)**: o sistema não é interrompido após a falha, mas precisa entrar em modo restrito (também chamado de modo seguro em [3]) até que seja reiniciado.
4. ***High-Availability*** **(HA, alta disponibilidade)**: o sistema precisa retornar ao estado normal sempre que houver um erro de degradação de performance.

Técnicas de tolerância a falhas em software dividem-se em duas: *Single-Version* (versão única) e *Multple-Versions* (versão múltipla). Diferente da redundância em hardwares, a replicação de componentes não é efetiva [3]. Mais importante do que considerar quais técnicas devem ser aplicadas é desenvolver um bom planejamento da arquitetura do software em questão [26]. Boas práticas de programação também auxiliam na construção de um software tolerante a falhas [26]. Além disso, a tolerância a falhas em software contribui para a qualidade do sistema, e não para o seu funcionamento em si, e pode diminuir a capacidade do sistema em troca de maior confiança.

A modularização do sistema, que consiste em dividi-lo em pequenas partes independentes, mas que podem se comunicar entre si para troca de dados e informações, é

uma das técnicas mais utilizadas. Separando o sistema em atividades específicas, mas sem interdependência obrigatória de funcionamento, é possível evitar que erros propaguem para outros módulos e, consequentemente, para a aplicação como um todo.

Outra técnica que pode ser aplicada é a de particionamento, que consiste em isolar funcionalidades individuais de módulos independentes, facilitando testes e manutenção e evitando a propagação de erro entre módulos distintos.

A *system closure technique* (técnica de fechamento de sistema) baseia-se em embutir um mecanismo que simule autoridade dentro da aplicação, de forma que ações só possam ser executadas pelo sistema caso sejam devidamente autorizadas. Um exemplo prático são as permissões de aplicativos em sistemas operacionais, responsáveis por limitar quais são os recursos do sistemas que podem, de fato, serem acessados, mesmo que não haja explicitamente influência do usuário sob tal decisão.

Por fim, a *temporal structuring* (estruturação temporal) baseia-se no fato de que um sistema é composto por ações indivisíveis (logo, atômicas). Desta forma, cada interação do sistema é composta por somente dois componentes, de forma que agentes externos aos dois não consigam receber e nem enviar qualquer tipo de informação que possa influenciar na interação. Além disso, quando uma falha for identificada, somente os componentes envolvidos serão afetados.

3.3.1 Tolerância a falhas em software de versão única

Baseia-se na redundância de software aplicada a um módulo individual da aplicação. São necessárias duas propriedades básicas: autoproteção (detecção de falhas ou erros em informações externas recebidas, de forma que o módulo não seja comprometido) e auto-verificação (capacidade de identificar e recuperar erros internos) [27].

Wrappers, entidades que encapsulam uma parte da aplicação ou código, servem como uma forma de interface para filtrar as entradas de usuário para um software, bem como filtram as mensagens e informações mostradas para os usuários, e que são saídas do mesmo software [21]. O tratamento de exceções, apesar de estarem mais atreladas à decisões de design, são um mecanismo de detecção de falha, podendo interromper o sistema ou forçar o usuário a tomar uma determinada ação enquanto a aplicação continua a executar. Randell [28] descreve três classes de exceções:

- **de Interface**: mecanismo de autoproteção, tratado e declarado pelo próprio módulo da aplicação que está executando a ação em questão. Diz respeito à uma ação inválida que um usuário do sistema tentou executar.

- **Locais**: referente à erros internos, não necessariamente resultados por uma ação do usuário. São tratadas pela capacidade de tolerância a falhas do módulo em si [29].
- **de Falha**: ocorre somente após um erro detectado não ser resolvido pelas capacidades de tolerância a falhas do módulo.

No entanto, apesar do tratamento de exceções ser considerado uma forma de tolerância a falhas em software, existem vertentes de pensamento que afirmam que são dois tópicos distintos [29]. Inacio [30] afirma que a diferença entre os dois é que o tratamento de exceções auxilia no desenvolvimento de um software robusto, enquanto o conceito de tolerância a falhas ajudaria no desenvolvimento de um software confiável. Uma aplicação robusta, ainda segundo Inacio [30], consegue operar mesmo em situações adversas, enquanto um software confiável é capaz de identificar as falhas, preferencialmente sem que o sistema fique indisponível.

Existem também mecanismos de recuperação nesse tipo de tolerância a falhas, tais como *checkpoint and restart mechanism* (mecanismo de checkpoint e reinício), que age após erros que não deixam rastros (Hameed et al. [29] afirmam que eles vêm, fazem algum dano e logo em seguida somem). Seu conceito principal é baseado em resetar, podendo retornar o módulo para um estado antes do erro (denominado mecanismo de checkpoint e reinício estático) ou em algum momento após o erro (denominado mecanismo de checkpoint e reinício dinâmico).

3.3.2 Tolerância a falhas em software de versões múltiplas

Baseia-se no uso de *n* versões (também chamadas por variantes) do código de um módulo. A execução dessas versões pode ser em sequência ou em paralelo, podendo ser em pares ou grupos. Apesar de que espera-se que os módulos realizem a mesma atividade, é preferível que não sejam idênticos, podendo ter variações de *design*, implementação, dentre outros fatores que podem impactar em resultados ou falhas diferentes. Existem alguns padrões que podem ser seguidos, dos quais dois são descritos a seguir:

- ***Recovery Blocks* (blocos de recuperação)**: combina o conceito dos mecanismos de checkpoint e reinício juntamente com múltiplas versões do mesmo software, de forma que, quando seja detectado um erro em uma variante, o software seja resetado para algum intervalo de tempo anterior, mas com uma versão diferente da que falhou. Nesse padrão, os checkpoints são criados antes de uma nova versão inicializar.

- ***N-Version programming*** **(programação de múltiplas versões)**: nesse padrão, todas as versões são desenvolvidas para satisfazer os mesmos requisitos básicos. Comparam-se todas as saídas para decidir qual está correta, geralmente através de um sistema de votação. Ao contrário dos blocos de recuperação, a complexidade desse padrão não está no desenvolvimento de novas versões, visto que elas precisam ser similares, mas pode estar no algoritmo de voto.

3.3.3 Detecção de erros

Existem diversas checagens que podem ser realizadas para as saídas de dados ou informações de um módulo, sendo elas [29]:

- ***Replication checks*** (verificações de réplica): utilizada em software de versões múltiplas apresentados na seção 3, item 3.2.6. Corresponde à utilização de componentes idênticos, e a comparação da saída de ambos os módulos.
- ***Timing checks*** (verificações de tempo): aplicável a módulos que trabalham com restrições de tempo. Permitem que sejam verificados comportamentos de desvio de tempo durante a execução.
- ***Reversal checks*** (verificações inversas): utilizada para calcular as entradas de um módulo com base em suas saídas. Erros são detectados somente se o conjunto de entradas calculado não é igual ao inicial. Utilizado em sistemas onde a computação inversa não depende de variáveis complexas, e pode ser calculado facilmente.
- ***Coding checks*** (verificações de código): de acordo com [26] e [29] essas checagens utilizam "redundância da informação com relacionamentos fixos entre a informação e si e a redundante". A relação é verificada antes e depois das operações. Um exemplo é o *checksum*.
- ***Reasonableness checks*** (verificações de razoabilidade): utiliza-se de propriedades, que podem ser baseadas nos requerimentos do módulo onde se encontram, presentes nos dados para detecção de erros.
- ***Structural checks*** (verificações de estrutura): baseiam-se em estruturas de dados (filas, listas, árvores, etc). A verificação ocorre de acordo com particularidades da estrutura em questão (por exemplo, o número de elementos na estrutura, seus ponteiros, dentre outros [29]).
- ***Run-Time checks*** (verificações em tempo de execução): geralmente implementado em hardwares. Em sua maioria, são mecanismos já presentes (como verificação de divisão por zero, *overflow*, etc). Ainda segundo Hameed et al. [29], não precisam estar

atrelados a nenhuma aplicação específica, mas conseguem, efetivamente, detectar erros.

4. Microsserviços

Em contrapartida às arquiteturas monolíticas, arquiteturas de microsserviços utilizam o conceito de modularização, dividindo um programa em diversas partes menores (denominadas por serviço) com funções específicas e que se comunicam entre-si, mas sem que haja uma dependência mútua para funcionamento da aplicação como um todo. Cada módulo pode ser desenvolvido, implementado e testado independente de outros. Graças a isso, aplicações projetadas para esse tipo de arquitetura possuem facilidade de manutenção, baixo acoplamento de código e são mais escaláveis [31]. Adicionalmente, cada serviço pode ser desenvolvido utilizando tecnologias distintas da aplicação. Porém, apesar das vantagens providas, a complexidade desse tipo de arquitetura tende a aumentar com o tempo, conforme novos serviços são implementados no sistema principal [32].

Apesar de não haver definição formal, existem características comuns entre implementações, de forma que se tornaram os padrões atuais. Os serviços devem ser pequenos, implementando somente uma regra de negócio, autônomos, onde um único módulo deve possuir os dados necessários para operação [33] e ser capaz de comunicar-se com outro através de protocolos de comunicação leves (como o HTTP, através de chamadas de API) [34]. No entanto, em aplicações práticas, é comum que serviços distintos troquem dados de entrada e saída.

Assim como outros sistemas distribuídos, segurança é um dos fatores de discussão. Como microsserviços são uma arquitetura orientada a serviços, é comum que dificuldades similares sejam enfrentadas [35]. Por ser comum o envio dos dados em formatos estruturados (devido ao uso de REST APIs) como JSON e XML, é necessário o uso de cifras de criptografia adicionais, ou o uso de APIs de autenticação que possam garantir a segurança dos dados.

5. Metodologia

Foi realizada uma revisão sistemática acerca dos temas IoT, destrinchando-se em *IoT state of art*, *IoT concepts*, *IoT challenges*, *microservice architecture* e *fault tolerant systems*. As pesquisas foram realizadas através das seguintes strings de busca: *IoT storage*, *IoT power consumption*, *IoT processing*, *IoT data storage*, *IoT energy management*, *fault tolerant systems*, *software fault tolerance*, *hardware fault tolerance*, *software redundancy*, *hardware redundancy*, *microservice architectures, msa*, utilizando websites como IEEE, Google Scholar, ResearchGate e ScienceDirect para obtenção dos artigos. Ademais, foram desenvolvidas duas provas de conceito, a fim de explicitar e demonstrar a aplicabilidade dos objetos de estudo desta pesquisa.

Somente foram incluídas fontes de literatura cinzenta que partissem de empresas estabelecidas há mais de 5 anos no mercado, e que, preferencialmente, tivessem vertentes de pesquisa ou atuação no mercado em relação ao assunto pesquisado. No entanto, o website de Martin Fowler foi uma das exceções, visto que ele foi um dos primeiros a discutir abertamente sobre o assunto antes que o mesmo fosse repercutido como é atualmente.

Quaisquer materiais anteriores a 2017 foram descartados, tendo em vista um embasamento mais condizente com o estado da arte, especialmente sobre IoT, salvos estudos de casos ou bases teóricas, visto que são o ponto de partida para esses assuntos. Os materiais incluídos foram divididos nas seguintes categorias, mas não limitando-se somente a uma:

Quadro 1: Categorização dos artigos utilizados como referência para o desenvolvimento desta pesquisa.

Categoria	**Referências**
Armazenamento IoT	[5] A Storage Solution for Massive IoT Data Based on NoSQL [7] Resource management in pervasive Internet of Things: A survey [9] Making sense of IoT data [13] Fog Computing for Sustainable Smart Cities: A Survey. ACM Computing Surveys
Processamento de dados IoT	[1] Cloud computing and Internet of Things fusion: Cost issues. [8] SAT-IoT: An Architectural Model for a High-Performance Fog/Edge/Cloud IoT Platform [9] Making sense of IoT data [10] 10 Edge computing use case examples [12] DIFFERENCE Between Cloud Computing and Fog Computing
Consumo de energia em IoT	[15] Energy Harvesting towards Self-Powered IoT Devices [16] Energy harvesting in IoT devices: A survey [17] The IoT energy challenge: A software perspective

Embasamento IoT	[1] Cloud computing and Internet of Things fusion: Cost issues. [2] Resource management in pervasive Internet of Things: A survey. [4] A systematic review of IoT in healthcare: Applications, techniques, and trends [5] A Storage Solution for Massive IoT Data Based on NoSQL [6] HOW Do IoT Devices Communicate? [7] Resource management in pervasive Internet of Things: A survey [8] SAT-IoT: An Architectural Model for a High-Performance Fog/Edge/Cloud IoT Platform [9] Making sense of IoT data [10] 10 Edge computing use case examples [11] IoT Architecture for Smart Grids [12] DIFFERENCE Between Cloud Computing and Fog Computing [13] Fog Computing for Sustainable Smart Cities: A Survey. ACM Computing Surveys [14] How Amazon Alexa works? Your guide to Natural Language Processing (AI) [15] Energy Harvesting towards Self-Powered IoT Devices [16] Energy harvesting in IoT devices: A survey [17] The IoT energy challenge: A software perspective [37] Survey on IoT solutions applied to Healthcare [38] Digital Image Processing and IoT in Smart Health Care - A review
Tolerância a falhas	[3] Tolerância a falhas: conceitos e exemplos [18] Sistema de Tolerância a falhas através de redundância de software e hardware [19] Software Fault Tolerance: A Theoretical Overview [20] Fault-Tolerant Systems [21] Hardware Redundancy. In: Fault-Tolerant Design [22] Triple modular redundancy [23] The Use of Triple-Modular Redundancy to Improve Computer Reliability [24] Fault-tolerant design and evaluation for a railway bogie active steering system [25] Terminology Proposal: Redundancy for Fault Tolerance [26] Software Fault Tolerance: A Tutorial [27] Resourceful Systems for Fault Tolerance, Reliability, and Safety [28] The Evolution of the Recovery Block Concept, in Software Fault Tolerance [29] Software Fault Tolerance: A Theoretical Overview [30] Software Fault Tolerance [36] Standby redundancy for reliability improvement of wireless sensor network
Microsserviços	[31] Microservices: yesterday, today, and tomorrow [32] The pains and gains of microservices: A

	Systematic grey literature review [33] Design, monitoring, and testing of microservices systems: The practitioners' perspective [34] Microservices a definition of this new architectural term [35] Quality attributes and service-oriented architectures

Fonte: autoria própria

5.1 Prova de conceito

Como prova de conceito, foi realizada a prototipação de uma aplicação que simula uma central de dispositivos IoT dentro do contexto de casas inteligentes. Todo o desenvolvimento foi realizado através da plataforma web *Tinkercad*.

Cada dispositivo possui alguma forma de redundância de hardware, software ou a combinação de ambos, e simula um microsserviço. Não foi possível implementar um dispositivo central por completo, devido às dificuldades descritas no item 6.1.5 desta mesma seção. O dispositivo central seria responsável por simular uma aplicação onde o usuário poderia interagir através de comandos físicos, aplicações de celular ou para computadores, bem como receber informações (tais como alertas) dos outros módulos à que estivesse conectado. Como forma de tolerância a falhas em software, a aplicação é modularizada em diversos serviços diferentes, de forma que o dispositivo central poderia gerenciar os demais, mas cada módulo seria independente nos quesitos de funcionalidade e controle.

Ao todo, foram desenvolvidos dois sistemas que simulam serviços: sistema medidor de temperatura e um sistema de detecção de gás. Levando em consideração os conceitos de tolerância a falhas em software apresentadas na seção 3, item 3.3 desta pesquisa, observa-se que:

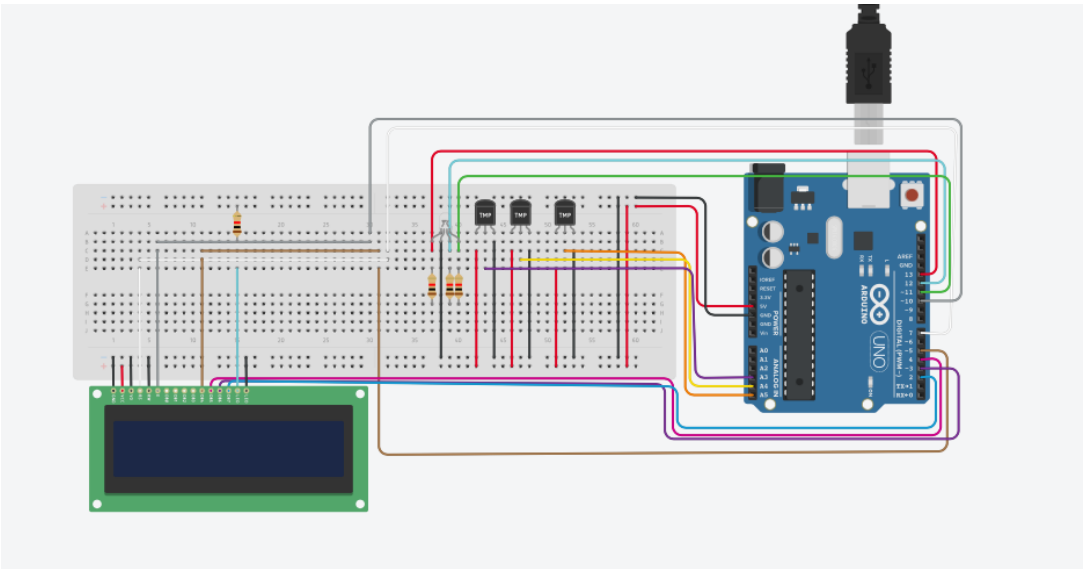
- A implementação em 5.1.1 representa um sistema falho-ativo. É possível que o sistema continue operando após a falha em um ou mais dos sensores, mas será de forma que a capacidade estará reduzida;
- A implementação em 5.1.2 representa um sistema falho-operacional, visto que é imperceptível para o usuário o erro ocorrido, cabendo ao próprio software fazer as mudanças necessárias;

6.1.1 Sistema medidor de temperatura

O conceito dessa simulação é montar, à nível de hardware e software, um sistema medidor de temperatura que é responsável por avisar a um usuário (através da mudança de cor de um LED) caso a temperatura de um determinado sistema exceda um limite pré-estabelecido por software. Uma das aplicabilidades desse serviço pode ser, por exemplo, o gerenciamento da temperatura de um gabinete de servidores. Do modo que o software foi desenvolvido, é possível substituir o LED por outra forma de sinalização (como, por exemplo, sensor sonoro). A complexidade de implementação em software é mínima, e a física depende do sensor em questão. O papel do dispositivo central nessa aplicação seria receber informações de temperatura e alertar ao usuário caso a mesma esteja acima do valor máximo estabelecido.

Foram utilizados três sensores de temperatura a fim de simular uma TMR, com o sistema de votação baseado na média da saída e com implementação em software. Há um LCD 16x2 que exibe a temperatura atual do sistema, calculada a partir do sistema de votação previamente descrito. O LED presente no sistema possui quatro estados, listados a seguir: (1) caso a temperatura seja maior ou igual à temperatura alta (nesse caso, considerada com 65º C), o LED irá piscar vermelho, (2) se a temperatura estiver entre a média (estabelecida como 55º C) e a alta, o LED pisca amarelo, (3) em qualquer temperatura abaixo da média, o LED permanece verde, (4) caso o sistema esteja indisponível, o LED ficará apagado (por sistema indisponível, nesta aplicação, entende-se que não há corrente elétrica para alimentação). As temperaturas foram escolhidas de forma arbitrária, baseado em dados de temperatura média de computadores em funcionamento.

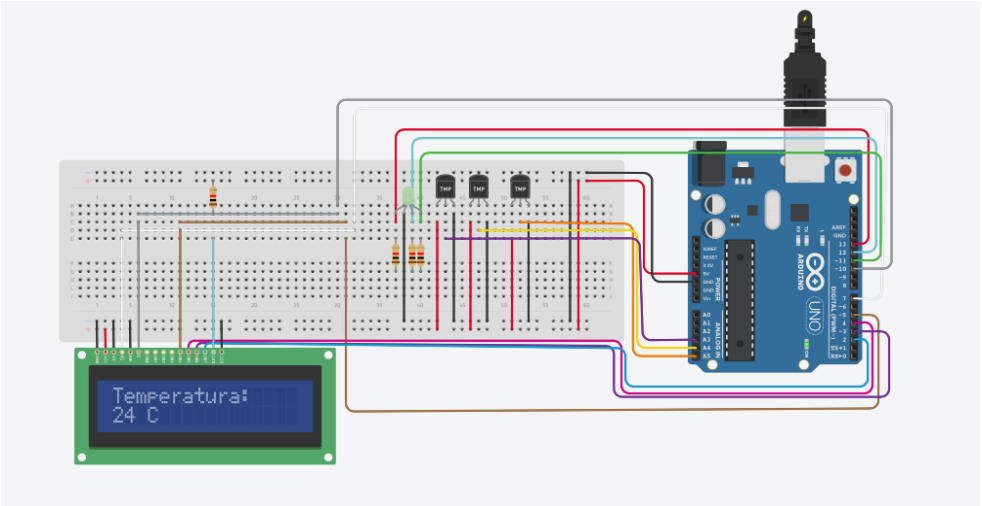
Figura 4: Circuito do sistema medidor de temperatura.



Fonte: *Tinkercad*

Nas figuras 4, 5 e 6, é possível ver retratos do sistema funcionando em cada um dos estados mencionados anteriormente: em temperatura abaixo da média, entre a média e a alta, e acima ou igual a alta.

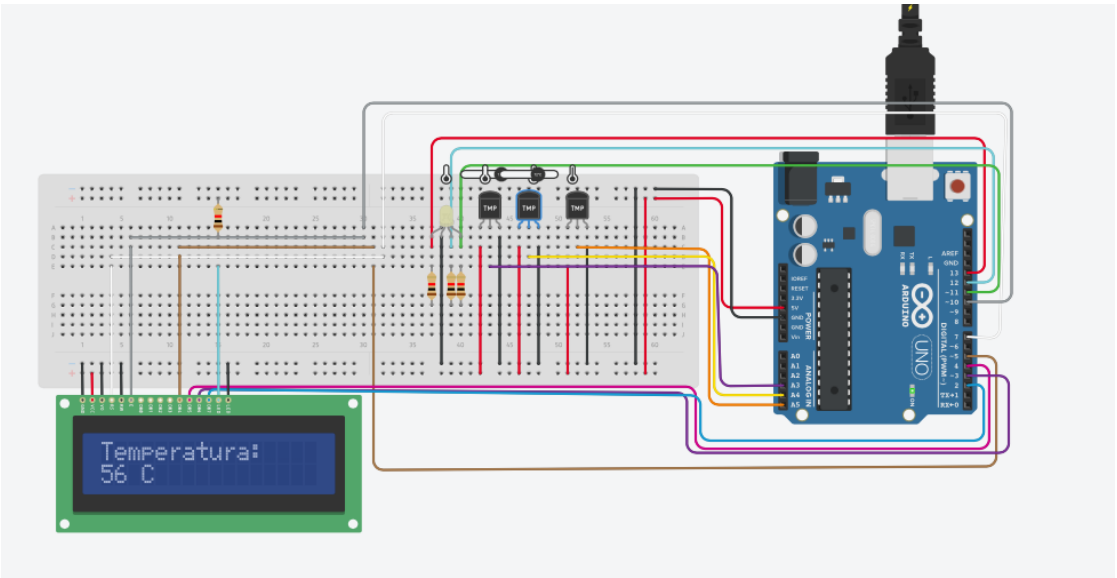
Figura 5: Sistema em funcionamento, com temperatura estabilizada abaixo da média.



Fonte: *Tinkercad*.

Na figura 5, é possível observar a temperatura, impressa no LCD, abaixo da média especificada, e o LED na cor verde.

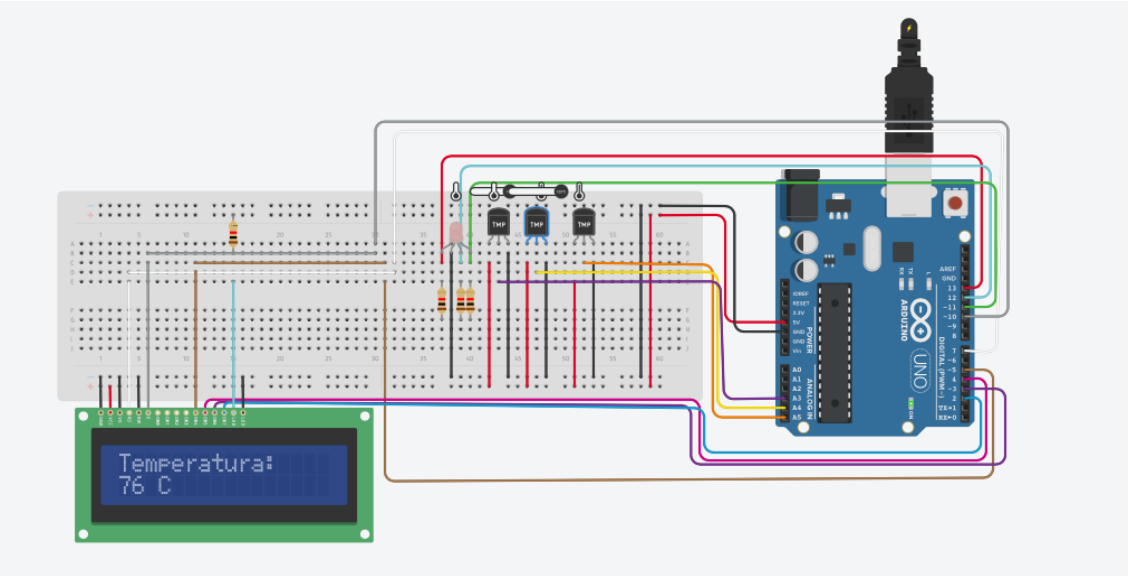
Figura 6: Sistema medidor de temperatura em funcionamento, com temperatura estabilizada abaixo da alta e acima da média.



Fonte: *Tinkercad.*

Na figura 6, percebe-se que a temperatura impressa no visor está acima da média estabelecida de 55º C, mudando a coloração do LED para amarelo.

Figura 7: Sistema em funcionamento, com temperatura acima da alta.

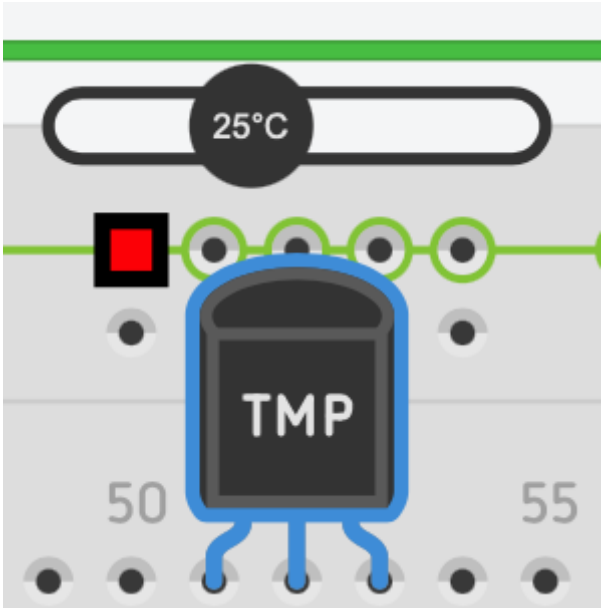


Fonte: *Tinkercad*.

Por fim, na figura 7, observa-se a temperatura de 76º C, acima da máxima definida, e o LED na coloração vermelha.

O controle de temperatura de cada sensor é individual, e realizado através de interruptores deslizantes que ficam disponíveis no momento em que a aplicação está sendo executada. O intervalo de possíveis valores é de -40º C até 125º C.

Figura 8: Interruptor deslizante que controla a temperatura do sensor.



Fonte: *Tinkercad*.

Para esta simulação, foram utilizados os seguintes componentes:

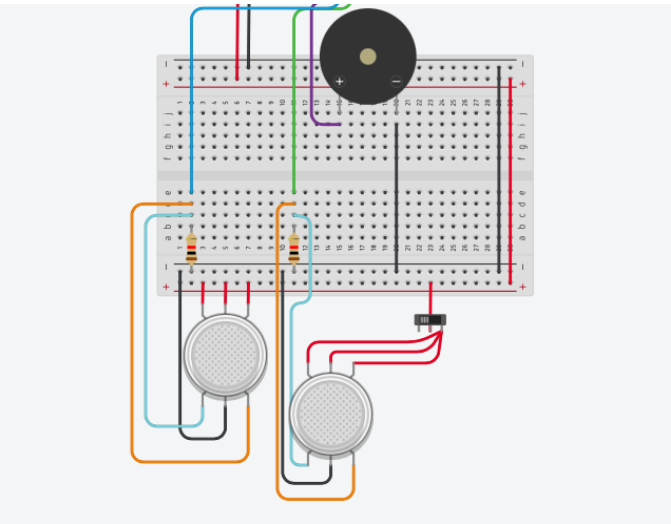
- 1 Arduino Uno R3;
- 3 sensores de temperatura TMP36;
- 1 placa de ensaio (sem especificação no *Tinkercad*);
- 3 resistores com 1kΩ de resistência;
- 1 LED RGB;
- 1 LCD 16x2;

5.1.2 Sistema de detecção de gás

Nesta simulação, foi montado um circuito, à nível de hardware e software, responsável por fazer a medição do vazamento de gás de um determinado ambiente, e alertar ao usuário (através de um aviso sonoro) caso seja detectado um vazamento. Uma das aplicabilidades desse serviço pode ser, por exemplo, o uso em indústrias ou ambientes que lidam com gases inflamáveis, ou até mesmo em cozinhas residenciais. Um dos pontos chaves dessa aplicação é que os sensores devem estar próximos um do outro.

Foram implementados dois detectores de gás, sendo um deles constantemente ativo e o outro na funcionalidade de *warm standby*, onde só deverá ser ativado caso haja falha do primeiro [36]. O acompanhamento do sensor principal é feito através da medição de corrente pelo software. Mesmo que não haja vazamentos, o sensor ainda emite corrente elétrica suficiente para ser detectada. No momento que não é mais identificada a passagem de corrente, o software altera o sensor responsável pela leitura para o de suporte, servindo como forma de verificação de razoabilidade, assim como apresentado na seção 4. A interrupção de alimentação do detector principal ocorre através de um interruptor deslizante no qual o mesmo está conectado.

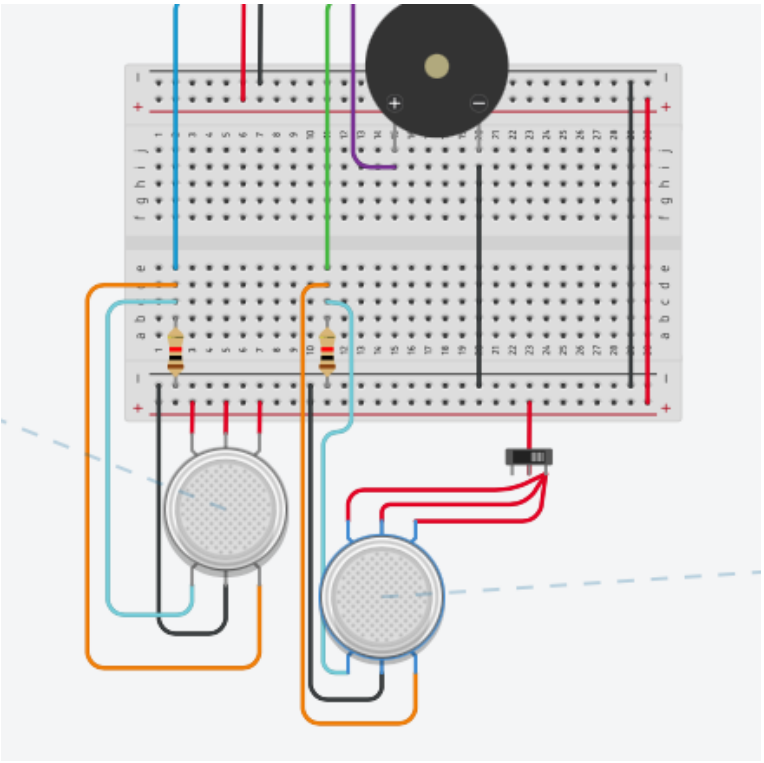
Figura 9: Circuito dos detectores de gás.



Fonte: *Tinkercad*.

Nas figuras 10, 11, 12, 13 e 14, é possível ver retratos do sistema funcionando nos seguintes casos: sem vazamento de gás (ou com vazamento distante o suficiente para não ser detectado pelos sensores), com vazamento de gás somente sob o sensor de suporte enquanto o principal funciona (sem disparo do alarme sonoro), com vazamento de gás sob o sensor principal enquanto o mesmo funciona (com disparo do alarme sonoro), com vazamento de gás sob o sensor principal enquanto o mesmo não está funcionando (sem disparo do alarme sonoro) e com vazamento de gás sob o sensor de suporte, enquanto o principal não está funcionando (com disparo do alarme sonoro).

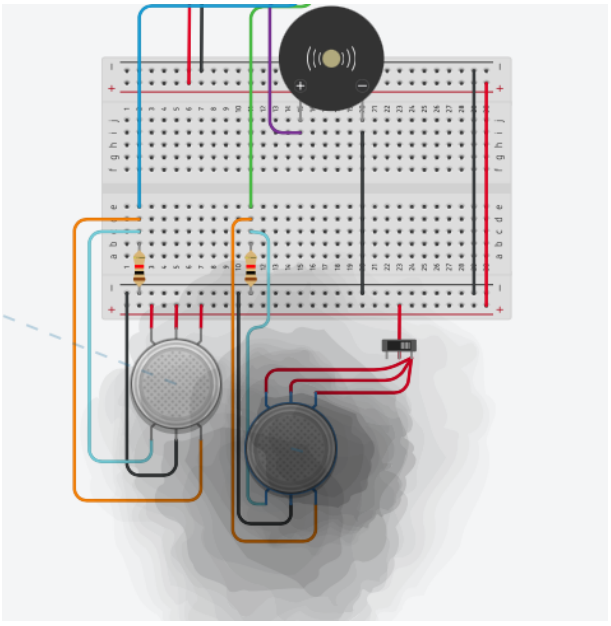
Figura 10: Sistema funcionando sem vazamento de gás.



Fonte: *Tinkercad*.

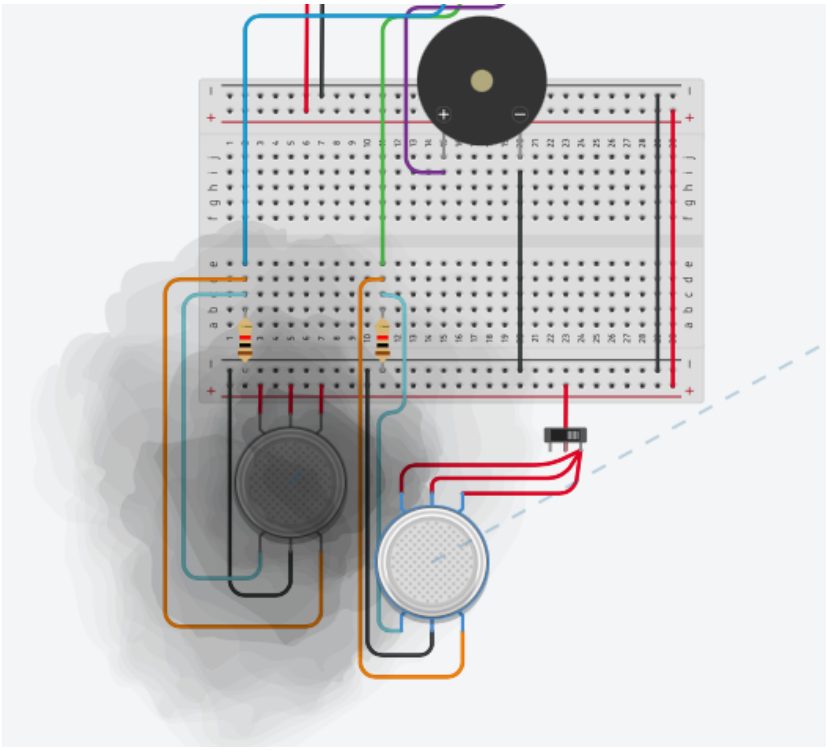
A linha azul que parte do sensor à direita na figura 10 representa a distância do gás em relação ao mesmo, e somente fica disponível durante a execução da aplicação.

Figura 11: Sistema funcionando com vazamento de gás, detectado pelo sensor principal, juntamente com o alarme sonoro.



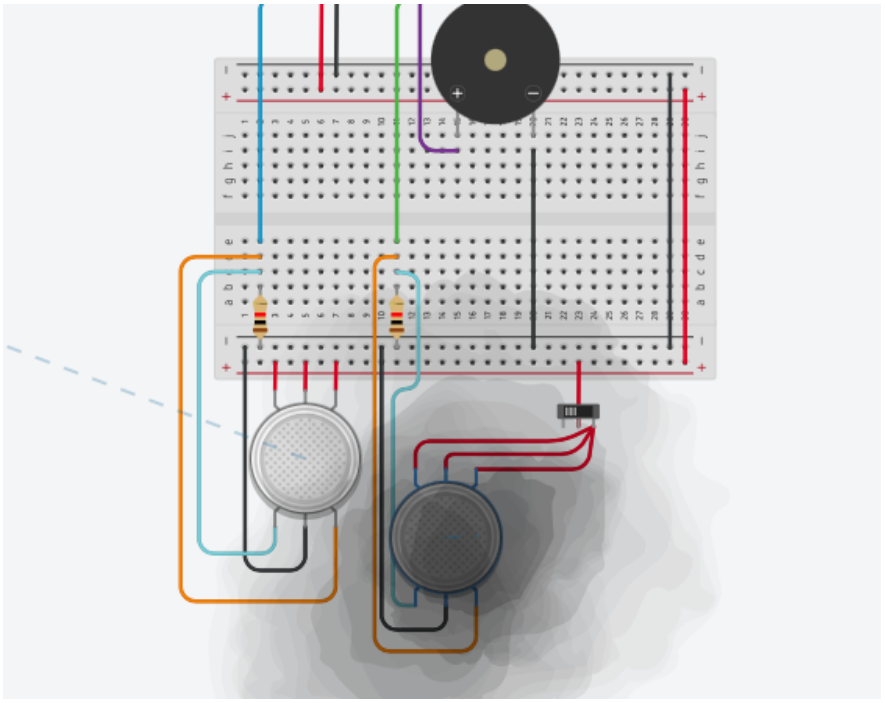
Fonte: *Tinkercad*.

Figura 12: Sistema funcionando com vazamento de gás sob o sensor secundário, sem que haja disparo do alarme.



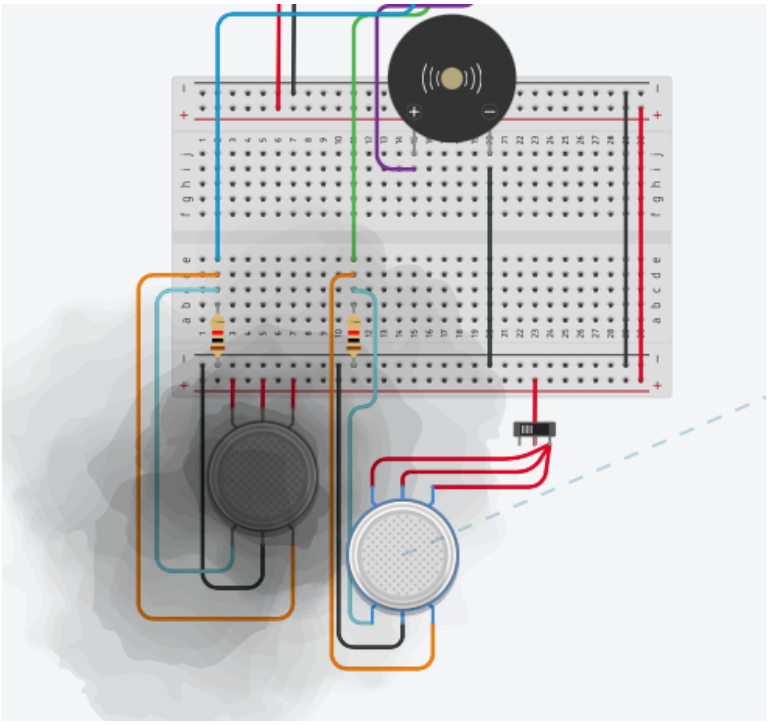
Fonte: *Tinkercad*.

Figura 13: Sistema funcionando, com vazamento de gás sob o sensor principal, enquanto o mesmo não funciona.



Fonte: *Tinkercad.*

Figura 14: Sistema funcionando com vazamento de gás sob o sensor de suporte, enquanto o principal não funciona.



Fonte: *Tinkercad*.

Devido à limitações da plataforma, é preferível simular o vazamento de gás separadamente, visto que cada sensor possui uma simulação própria. Em algumas iterações de execução, a plataforma não reconhecia o vazamento de gás do sensor distinto.

O funcionamento do detector principal é interrompido através de um interruptor deslizante durante o funcionamento do programa. Caso seja detectada falha no sensor principal, o sistema realiza uma contagem de 100 segundos (valor escolhido arbitrariamente, somente para fins de prototipagem), e, se permanecer a falta de corrente elétrica, uma mensagem é emitida, e a troca do sensor principal para o *warm standby* é realizada.

Para esta implementação, foram utilizados os seguintes componentes:

- 1 Arduino Uno R3;
- 2 sensores de gás (sem especificação na plataforma);
- 2 resistores com 1kΩ de resistência;
- 1 piezo;
- 1 interruptor deslizante (sem especificação na plataforma);

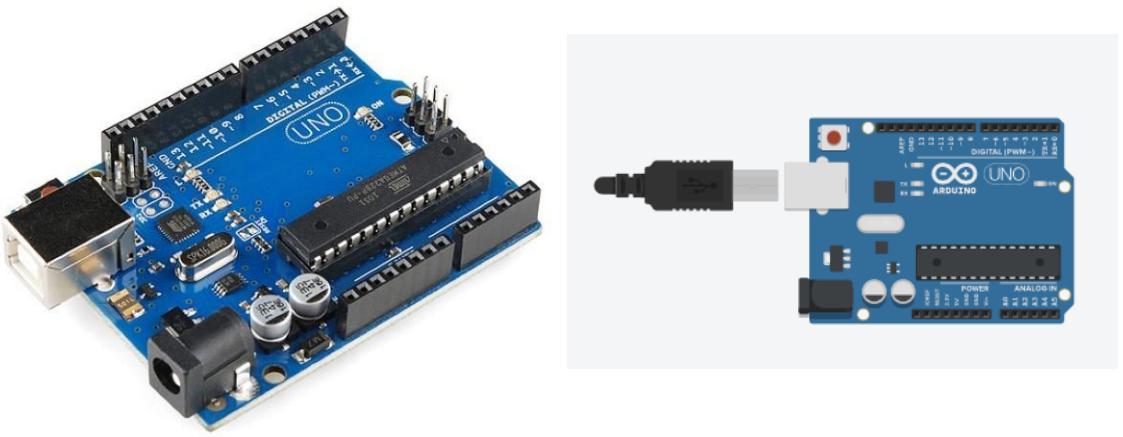
5.1.4 Especificação dos sensores utilizados

A seguir são apresentados os sensores utilizados na aplicação como um todo. É possível que algumas das especificações variem devido ao fato do *Tinkercad* não descrever as características dos componentes presentes nela. No entanto, como é possível perceber através das imagens que seguem as especificações, procurou-se o sensor físico cuja aparência mais remetia à versão virtual, baseada no nome disponível na plataforma e no fato de que alguns dos componentes presente nela estão presentes em *kits* básicos de Arduino.

5.1.4.1 Arduino Uno R3

- Microcontrolador: ATmega328
- Velocidade do Clock: 16 MHz
- Pinos I/O Digitais: 20(6 podem ser usadas como PWM)
- Portas Analógicas: 6
- Tensão de Operação: 5 V
- Tensão de Alimentação: 7-12 V
- Corrente Máxima Pinos I/O: 40 mA
- Memória Flash: 32 KB(0,5 KB usado no bootloader)
- SRAM: 2 KB
- EEPROM: 1 KB
- Dimensões: 53,4 x 86,6 mm

Figura 15: Arduino Uno R3 físico e virtual (visto de cima).

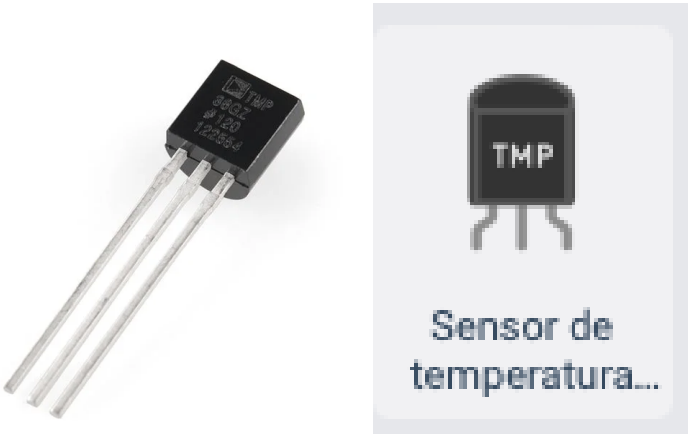


Fonte: montagem a partir de imagens coletadas no site Filipeflop e *Tinkercad*. Disponível em: <https://www.filipeflop.com/produto/placa-uno-r3-cabo-usb-para-arduino/>. Acesso em: 23 nov. 2022.

5.1.4.2 Sensor de temperatura TMP

- CI: TMP36GZ;
- Tensão de Operação: 2.7 a 5.5VDC;
- Faixa de medição: -40° a 125°C;
- Precisão: ±2°C;
- Linearidade: ±0,5°C;
- Sensibilidade: 10mV/ºC;
- Tipo de saída: analógica.

Figura 16: sensor de temperatura TMP físico e virtual (visto de frente).

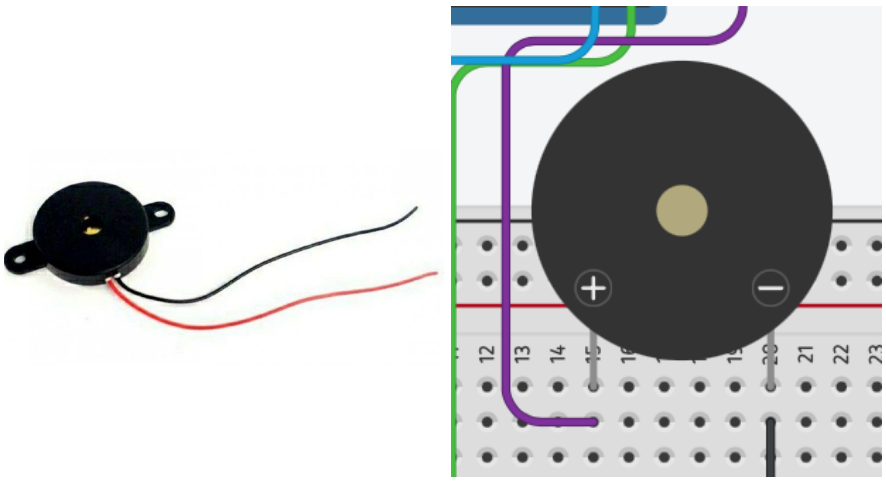


Fonte: montagem a partir de imagens coletadas nos sites AutoCore Robótica e *Tinkercad*. Disponível em: <https://www.autocorerobotica.com.br/sensor-de-temperatura-tmp36>. Acesso em: 23 nov. 2022.

5.1.4.3 Piezo

- Buzzer do tipo Ativo;
- Tensão de operação: 3 a 24 VDC
- Corrente de operação: 40 mA (Som intermitente), 60 mA (Som de alarme)
- Saída de som mínima (a 10cm): 105 dB (Som intermitente), 110 dB (Som de alarme)
- Frequência de ressonância: 2800±300 Hz
- Temperatura de operação: -20 a +80 °C
- Material: ABS
- Cor: Preto
- Dimensões: 29 mm (diâmetro) x 25 mm

Figura 17: Piezo físico e virtual (visto de cima).



Fonte: montagem a partir de imagens coletadas nos sites *Cyton Marketplace* e *Tinkercad*. Disponível em: <https://www.cytron.io/p-3-5v-22x4.5-piezo-buzzer-with-wires> . Acesso em: 23 nov. 2022.

5.1.4.4 Sensor de gás

Por existirem diversos modelos no mercado e não haver especificações no *Tinkercad*, existe a possibilidade de que algumas das informações abaixo sejam diferentes em relação ao sensor que foi utilizado no projeto.

- Modelo: MQ-4
- Detecção de gases: Metano, Propano e Butano.
- Concentração de detecção: 300-10.000ppm
- Tensão de operação: 5V
- Sensibilidade ajustável via potenciômetro
- Saída Digital e Analógica
- Fácil instalação
- Comparador LM393
- Led indicador para tensão
- Led indicador para saída digital
- Dimensões: 32 x 20 x 15mm
- VCC: 5V
- GND: GND
- D0: Saída Digital
- A0: Saída Analógica

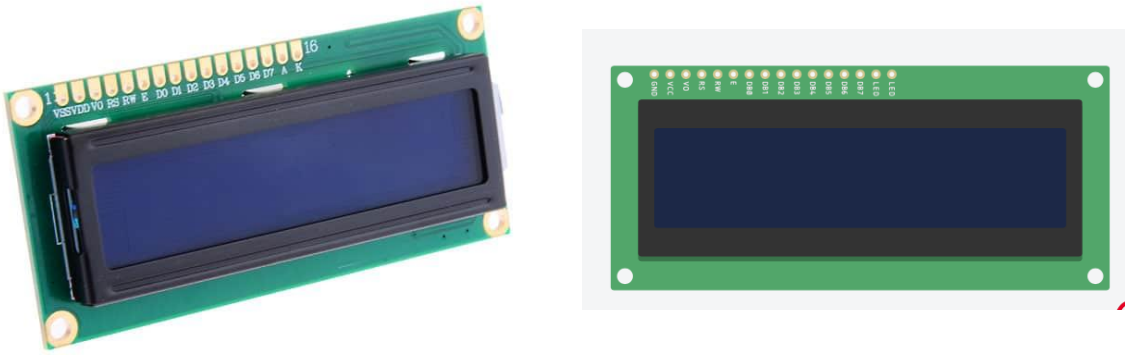
Figura 18: Sensor de gás MQ-4 físico e virtual (visto de cima).

Fonte: montagem a partir de imagens coletadas nos sites FilipeFlop e *Tinkercad*. Disponível em: <https://www.sparkfun.com/products/9404>. Acesso em: 25 nov. 2022.

5.1.4.5 LCD 16x2

- Cor *backlight*: Azul
- Cor escrita: Branca
- Dimensão Total: 80mm X 36mm X 12mm
- Dimensão Área visível: 64,5mm X 14mm
- Dimensão Caracter: 3mm X 5,02mm
- Dimensão Ponto: 0,52mm X 0,54mm

Figura 19: LCD 16 x 6 físico e virtual.



Fonte: montagem a partir de imagens coletadas nos sites FilipeFlop e *Tinkercad*. Disponível em: <https://www.filipeflop.com/produto/display-lcd-16x2-backlight-azul/>. Acesso em: 25 nov. 2022

5.1.5 Limitações da plataforma

Apesar das vantagens que proporciona, o *Tinkercad* também apresenta diversos fatores que dificultam o desenvolvimento de uma prototipagem mais complexa.

Um dos desafios que reduziu o escopo do projeto foi a falta de comunicação com entidades externas. O módulo WiFi ESP8266 é um dos sensores presentes na plataforma, mas com funcionalidades descontinuadas, impossibilitando a comunicação com outros dispositivos através da internet. Também não existe a possibilidade de realizar a conexão por Bluetooth, de forma que a única maneira de haver a comunicação entre dois Arduinos é através de conexão serial. No entanto, esta, além de limitante devido a dificuldade de conectar vários aparelhos ao mesmo tempo, não consegue ler e imprimir as mensagens (trocadas através do comando Serial.println) corretamente na plataforma.

A lentidão em determinados navegadores web (tais como *Mozilla Firefox* e *Safari*, que foram os principais utilizados durante o desenvolvimento desta pesquisa) dificulta a depuração dos sistemas, em especial o de detecção de gás, onde houveram vezes que o alarme demorava mais do que o usual para soar ou continuava soando mesmo depois da simulação de gás ter sido removida das proximidades do sensor. Não foi possível determinar ao certo se foi latência entre o sensor detectar o gás ou tempo de resposta do alarme, mas foi preciso reiniciar a aplicação diversas vezes até que fosse percebido que o erro não era relativo à montagem ou código. Além disso, utilizar a mesma nuvem de gás em dois sensores não funcionou em todas as iterações, novamente levantando o questionamento se era latência ou comportamento esperado da plataforma.

Por fim, houve uma etapa de desenvolvimento onde a página web precisou ser reiniciada, mas as alterações não haviam sido salvas (apesar da mensagem no canto superior direito na tela indicar o contrário). O projeto foi resetado para um estado de dois dias antes, e um dos módulos precisou ser refeito.

5.1.6 Informações do projeto de acordo com a plataforma

O *Tinkercard* oferece algumas visualizações em relação ao projeto, as quais estão presentes nos itens 6.6.1 e 6.6.2 a seguir:

5.1.6.1 Listagem dos componentes

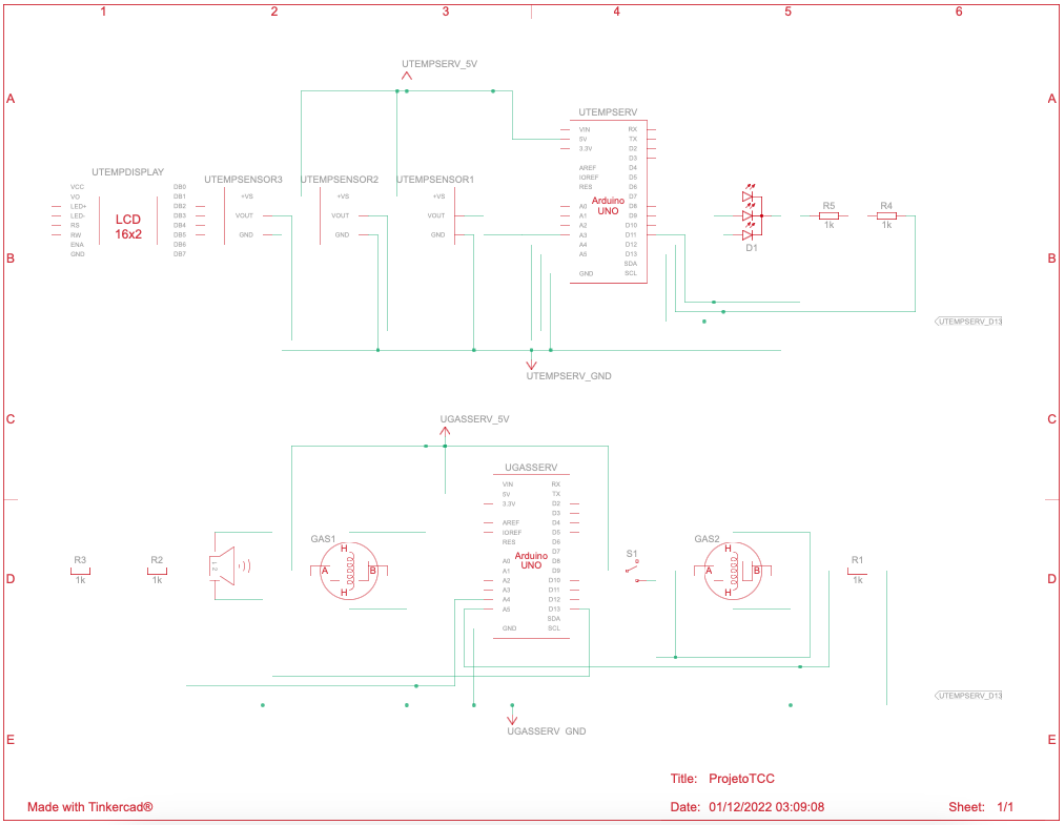
Figura 20: tabela com listagem dos componentes.

Nome	Quantidade	Componente
UGasServ UTempServ	2	Arduino Uno R3
UTempSensor1 UTempSensor2 UTempSensor3	3	Sensor de temperatura [TMP36]
R3 R4 R5 R1 R2	5	1 kΩ Resistor
D1	1	LED RGB
UTempDisplay	1	LCD 16 x 2
GAS1 GAS2	2	Sensor de gás
PIEZO1	1	Piezo
S1	1	Interruptor deslizante

Fonte: *Tinkercad.*

5.1.6.2 Esquema do projeto

Figura 21: esquema do projeto auto-gerado pela ferramenta.



Fonte: *Tinkercad.*

6. Resultados e discussões

As vantagens trazidas pela aplicação de tolerância a falhas dentro do contexto de software e hardware são evidentes. A possibilidade de arquitetar sistemas que sabem como se comportar em situações de falha não só aumenta a dependabilidade de aplicações presente no cotidiano, mas também faz com que sistemas críticos não fiquem dependentes de fluxos perfeitos. Juntamente com IoT e microsserviços, esses três conceitos distintos possuem diversos fatores em comum, o que faz com que acabem convergindo para um mesmo ponto no estado da arte tecnológica.

6.1 IoT

É evidente que IoT foi um conceito que veio modificando-se diversas vezes ao longo dos últimos anos. E, ao analisar tendências futuras, fica claro que continuará sendo modificado. Aplicações como automação residencial e cidades inteligentes fazem cada vez mais parte do cotidiano, já sendo realidade em diversos países.

6.2 Armazenamento em IoT

Com a miniaturização tecnológica que vem ocorrendo ao longo dos anos, não é natural considerar que um dispositivo, por menor que seja, não possua capacidade razoável de armazenamento. Dispositivos de memória flash medem pouco mais do que alguns centímetros, e podem possuir capacidade de armazenamento na faixa de gigabytes. No entanto, é necessário considerar que, dependendo do dispositivo em questão, não há a necessidade de armazenamento local. Sensores inteligentes, por exemplo, não necessitam de espaço para guardar os dados coletados, pois é comum que haja uma entidade externa (como microcontroladores, comumente utilizados em ambientes de prototipação) responsáveis por participar por mais uma etapa do fluxo que resultará na entrega da informação.

Para dispositivos que operam em *edge*, e que precisam realizar processamento local, existem diversas alternativas de armazenamento que consistem em guardar somente o que for necessário para o funcionamento da aplicação.

6.3 Processamento de dados em IoT

Uma dificuldade geralmente presente em dispositivos IoT é a falta de poder computacional. Esta restrição, no entanto, não se limita somente a dificuldades físicas, podendo também ser uma decisão de *design*, com o intuito de minimizar o dispositivo ou baratear um determinado produto para o usuário final. A depender do ecossistema, não é necessário maior poder computacional no *edge*, pois as atividades na ponta são suficientes para serem reproduzidas por sensores inteligentes já disponíveis no mercado. É esperado, por padrão, que dispositivos *fog* e *cloud* possuam uma maior capacidade de processamento.

Outro ponto, que estende-se para a área de segurança, é que dispositivos IoT, independente de porte, geralmente não podem depender de algoritmos de criptografia já estabelecidos por não conseguirem processar em paralelo ou por serem atividades custosas computacionalmente.

6.4 Consumo de energia em aplicações IoT

Ao longo dos anos, vêm sendo apresentadas diversas soluções visando que dispositivos IoT deixem de depender de baterias ou fontes de alimentação através de cabos de energia. Apesar de soluções como energias renováveis funcionarem para diversas aplicações, atualmente, como apresentado por Georgiu et al. [17] na seção 2 deste trabalho, existem fatores que podem fazer com que essa não seja a melhor opção.

6.5 Segurança em dispositivos IoT

Atualmente existem diversos estudos para a implementação dos chamados *lightweight algorithms* (algoritmos leves) como forma de aumentar a segurança dos dados que transitam entre dispositivos IoT, assim como apresentado na seção 2 deste trabalho. No entanto, como é comum que a capacidade de processamento desses dispositivos também vá aumentando com o passar do tempo, existe a possibilidade que, futuramente, os algoritmos atuais consigam executar nesses dispositivos. Adicionalmente, existe também a possibilidade de que esses dispositivos possam executar algoritmos futuros sem que haja falta de poder computacional.

6.6 IoT e microsserviços

Ao analisar a definição de ambos os conceitos, percebe-se que, mesmo que não intencionalmente, diversas aplicações de IoT aplicam os conceitos dos microsserviços: módulos independentes, dentro de um ecossistema, capaz de se auto gerenciar e comunicar-se com outros para obtenção de dados. As provas de conceito apresentadas na seção 6 deste

trabalho levaram esse fator em consideração. As formas de comunicação em microsserviços também vão de acordo com as de IoT.

Na literatura atual, existem diversos autores que reconhecem as similaridades entre esses dois conceitos, propondo, inclusive, formas de como os micro serviços podem auxiliar não só na segurança dos dados em IoT, mas também na tolerância à falha das aplicações.

6.7 Tendências futuras em IoT

IoT vem desempenhando um papel revolucionário na área de saúde. O monitoramento de pacientes através de dispositivos vestíveis vem sendo cada vez mais comum, através de aplicações como: monitoramento de sono, monitoramento de batimentos cardíacos (identificando picos ou casos de arritmia, por exemplo), acompanhamento de atividades físicas, dentre outros. Atualmente, existe uma área denominada HIoT (*Healthcare IoT*), com foco na aplicação desse tipo de tecnologia através da criação de ecossistemas para a área de saúde.

O estudo desenvolvido por Castro et al. [37] que considera prontuários eletrônicos é factível, e pode implementar outras tecnologias em alta (como *blockchain* e contratos inteligentes) para aumentar a segurança da aplicação. A proposta de monitoramento de pacientes com demência em [38] demonstra que a área consegue expandir-se não somente para saúde física, mas também mental.

6.8 Tolerância a falhas em IoT

Apesar de ser um conceito datado desde por volta da década de 1950, como pode ser visto em [23], muitas das técnicas propostas ainda são plausíveis de serem aplicadas devido à sua robustez. Devido a popularização da internet e dos dispositivos que dependem dela, vem-se percebendo um aumento na preocupação na falha de sistemas como um todo, não apenas dos críticos. No entanto, muito do esforço atual é direcionado para aplicado para web (especialmente em áreas como *cloud computing* e *serverless architecture* – alocação de recursos da nuvem sob demanda).

As provas de conceito descritas na seção 6 reforçam que é preciso olhar para a aplicação como um todo antes de escolher um modelo de tolerância a falhas para ser aplicado. No caso de IoT, por mais que os dados enviados sejam o foco das aplicações atuais (temas como *big data* reforçam este pensamento), podem existir momentos onde é necessário focar no funcionamento do hardware tanto quanto nos dados que ele transmite.

Garantir a integridade do hardware é também uma forma de garantir a integridade dos dados que trafegam. Além disso, o devido planejamento do ecossistema onde os dispositivos se encontram permite visualizar quais as técnicas de tolerância a falhas devem ser aplicadas nos softwares. Diferente de técnicas de criptografia e processamento de dados que muitas vezes vão além das capacidades desses dispositivos, muitas técnicas de tolerância a falhas podem ser aplicadas em IoT sem que haja mudanças significativas na aplicação.

7. Conclusão

Assim como descrito por diversos autores, falhas são inevitáveis em sistemas. No entanto, os possíveis impactos podem ser mitigados e minimizados, se as devidas técnicas forem aplicadas de antemão. Em uma sociedade onde a tecnologia vem se integrando cada vez mais em aspectos críticos da rotina, e onde velocidade e disponibilidade são fatores de sucesso ou falha, a capacidade de um sistema operar sob condições defeituosas é de extrema importância.

Com IoT tornando-se cada vez mais parte do cotidiano, bem como de tendências futuras, é crucial o desenvolvimento de sistemas e plataformas que considerem não somente questões de rede e segurança física dos dispositivos, mas também de componentes que fazem parte do fluxo de coleta, transmissão e processamento de dados, incluindo o software responsável, muitas vezes, por transformá-los em informação. São inegáveis os avanços e facilidades proporcionadas pelas tecnologias de objeto de estudo desta pesquisa, e também conceitos, que hoje são vistos como tendências futuras, teriam um progresso mais lento na sua ausência.

Referências

[1] DHIRANI, Lubna L. NEWE, Thomas. LEWIS, Elfed. NIZAMANI, Shahzad. Cloud computing and Internet of Things fusion: Cost issues. Eleventh International Conference on Sensing Technology.Eleventh International Conference on Sensing Technology (ICST). 2017.

[2] ZAHOOR, Saniya. MIR, Roohie Naaz. Resource management in pervasive Internet of Things: Asurvey. Journal of King Saud University - Computer and Information Sciences. Volume 33.

[3] WEBER, Taisy Silva. **Tolerância a falhas: conceitos e exemplos**. Programa de Pós-Graduação em Computação - Instituto de Informática - UFRGS.

[4] KASHANI, Mostafa Haghi *et al*. **A systematic review of IoT in healthcare: Applications, techniques, and trends**. Journal of Network and Computer Applications, Journal of Network and Computer Applications 192 (2021), jul. 2021.

[5] MAO, W. et al. **A Storage Solution for Massive IoT Data Based on NoSQL**. IEEE International Conference on Green Computing and Communications, p. 50-57, nov 2012.

[6] **HOW Do IoT Devices Communicate?** Digi. Disponível em: https://www.digi.com/blog/post/how-do-IoT-devices-communicate. Acesso em: 13 set. 2022.

[7] ZAHOOR, Saniya. MIR, Roohie Naaz. **Resource management in pervasive Internet of Things: A survey**. Journal of King Saud University - Computer and Information Sciences. Volume 33.

[8] PENA, Lopez. ANGEL, Miguel. MUNOZ, Isabel Fernandez. **SAT-IoT: An Architectural Model for a High-Performance Fog/Edge/Cloud IoT Platform**. IEEE 2019 IEEE 5th World Forum on Internet of Things (WF-IoT'19). abr. 2019.

[9] GERBER, Anna. KANSAL, Satwik. **Making sense of IoT data**. IBM. 26 mar. de 2020. Disponível em: https://developer.ibm.com/tutorials/IoT-lp301-IoT-manage-data/. Acesso em: 20 set. de 2022.

[10] **10 Edge computing use case examples**. Partners. Disponível em: https://stlpartners.com/articles/edge-computing/10-edge-computing-use-case-examples/. Acesso em: 26 set. 2022.

[11] SHAHINZADEH, Hossein. MORADI, Jalal. B. Gharehpetian, Gevork. NAFISI, Hamed. ABEDI, Mehrdad. **IoT Architecture for Smart Grids**. (2019).

[12] **DIFFERENCE Between Cloud Computing and Fog Computing**. GeeksforGeeks. 06 mai. de 2020. Disponível em: https://www.geeksforgeeks.org/difference-between-cloud-computing-and-fog-computing/. Acesso em: 03 out. 2022.

[13] PERERA, Charith. QIN, Yongrui. ESTRELLA, Juilio. REIFF-MARGANIEC, Stephan. VASILAKOS, Athanasios. **Fog Computing for Sustainable Smart Cities: A Survey. ACM Computing Surveys**. 2017.

[14] GONFALONIERI, Alexandre. **How Amazon Alexa works? Your guide to Natural Language Processing (AI)**. Towards Data Science. 21 nov. de 2018. Disponível em: https://towardsdatascience.com/how-amazon-alexa-works-your-guide-to-natural-language-processing-ai-7506004709d3. Acesso em: 12 out. 2022.

[15] ELAHI, Hassan. KHUSHBOO, Munir. EUGENI, Marco. ATEK, Sofiane. GAUDENZI, Paolo. **Energy Harvesting towards Self-Powered IoT Devices**. Energies. 2020.

[16] GARG, N. GARG, R. **Energy harvesting in IoT devices: A survey**. 2017 International Conference on Intelligent Sustainable Systems (ICISS). p. 127-131. 2017.

[17] GEORGIOU, Kyriakos. XAVIER-DE-SOUZA, Samuel. EDER, Kerstin. **The IoT energy challenge: A software perspective**. Universidade Federal do Rio Grande do Norte. 27 jun. 2017.

[18] ROCHA, Aline Souza. RAIMANN, Eliane. MACÊDO, Heverton Barros de. **Sistema de Tolerância a falhas através de redundância de software e hardware**. Relatório final do PIBIC/CNPq/IFG, jul. 2011.

[19] HAMEED, Omar *et al*. **Software Fault Tolerance: A Theoretical Overview**. International Journal of Simulation: Systems, Science & Technology. jun. 2019.

[20] KOREN, Israel; KRISHNA, C. Mani. **Fault-Tolerant Systems**. Elsevier.

[21] Dubrova, E. **Hardware Redundancy. In: Fault-Tolerant Design**. Springer, New York, NY. 2013.

[22] **Triple modular redundancy**. ACM SIGDA Newsletter. 2007. 37. 1-1. 10.1145/1859863.1859864.

[23] LYONS, R. E. VANDERKULK, W. **The Use of Triple-Modular Redundancy to Improve Computer Reliability.** IBM Journal. abr. 1962.

[24] FU, Bin. BRUNI, Stefano. **Fault-tolerant design and evaluation for a railway bogie active steering system**. Vehicle System Dynamics. 2020.

[25] SPECHT, Johannes. **Terminology Proposal: Redundancy for Fault Tolerance**. University of Duisburg-Essen. 2013.

[26] TORRES-POLMARES, Wilfredo. **Software Fault Tolerance: A Tutorial**. Langley Research Center, NASA. out. 2000.

[27] ABBOTT, Russel J.. **Resourceful Systems for Fault Tolerance, Reliability, and Safety**. ACM Computing Surveys, Vol. 22, No. 1, March 1990, pp. 35 - 68.

[28] RANDELL, Brian. XU, Jie. **The Evolution of the Recovery Block Concept, in Software Fault Tolerance**. Wiley. 1995.

[29] HAMEED, Omar Abdul. RESEN, Israa. HUSSAIN, Saif. **Software Fault Tolerance: A Theoretical Overview.** International Journal of Simulation: Systems, Science & Technology. 2019.

[30] INACIO, Chris. **Software Fault Tolerance**. Dependable Embedded Systems. 1998.

[31] DRAGONI, Nicola. GIALLORENZO, Saverio. LAFUENTE, Alberto Lluch. MAZZARA, Manuel. MONTESI, Fabrizio. MUSTAFIN, Ruslan. SAFINA, Larisa. **Microservices: yesterday, today, and tomorrow**. Cornell University. 2017.

[32] SOLDANI, Jacopo. TAMBURRI, Damian Andrew. HEUVEL, Willem-Jan Van Den. **The pains and gains of microservices: A Systematic grey literature review**. Journal of Systems and Software. 2018,

[33] WASEEM, M. LIANG, P. SHAHIN, M. DI SALLE, A. MÁRQUEZ, G. **Design, monitoring, and testing of microservices systems: The practitioners' perspective**. Journal of Systems and Software. 2021.

[34] LEWIS, James. FOWLER, Martin. **Microservices a definition of this new architectural term**. MartinFowler. Disponível em: https://martinfowler.com/articles/microservices.html. Acesso em: 16 out. 2022.

[35] BASS Len. MERSON Paulo. O'BRIEN Liam. **Quality attributes and service-oriented architectures**. Department of Defense. Technical Report September, 2005.

[36] CATELANI, Marcantonio. CIANI, Lorenzo. BARTOLINI, Alessandro. GUIDI, Giulia. PATRIZI, Gabriele. **Standby redundancy for reliability improvement of wireless sensor network**. 2019.

[37] CASTRO, Diego. CORAL, William. CABRA, José, et al. **Survey on IoT solutions applied to Healthcare**. Universidad Nacional de Colombia. 12 out. 2017.

[38] KANSAL, Isha. POPLI, Renu. VERMA, Jyoti, et al. **Digital Image Processing and IoT in Smart Health Care - A review**. 2022 International Conference on Emerging Smart Computing and Informatics (ESCI). 22 abr. 2022.