UNIVERSIDADE FEDERAL DE MINAS GERAIS

Trabalho Final - Documentação
Automação em Tempo Real

Gabriel Santos Pereira - 2022061122
Natan Henrique Carvalho Rocha - 2022421064
Sofia Mara Torres de Oliveira - 2022082847

Belo Horizonte, 30 de novembro de 2025

1. INTRODUÇÃO

O presente documento tem como objetivo apresentar uma visão geral do sistema de controle e monitoramento dos veículos autônomos de mineração desenvolvido, descrevendo os requisitos atendidos e a forma como as partes críticas do código foram implementadas, em conformidade com as especificações do Trabalho Final de Automação em Tempo Real (2025/2).

A estrutura do sistema foi concebida para lidar com os desafios de um ambiente concorrente e de tempo real, no qual múltiplas tarefas executam em paralelo para gerenciar a navegação, processar dados de sensores, monitorar falhas e interagir com interfaces locais e remotas.

2. VISÃO GERAL DO SISTEMA

O sistema implementa o software embarcado de controle e supervisão de um caminhão autônomo de mineração operando em modo automático, em um ambiente simulado discreto de 100×100 unidades. A aplicação em C++ é organizada como um conjunto de tarefas concorrentes (threads) que compartilham dados por meio de buffers e mecanismos explícitos de sincronização, atendendo aos requisitos de um sistema de automação em tempo real.

O ponto de entrada (main) é responsável por inicializar os recursos compartilhados (buffers de comunicação entre módulos, objetos de sincronização e notificador de eventos de falha), configurar a comunicação via MQTT (assinatura/publicação em tópicos específicos do caminhão) e criar as threads correspondentes às tarefas principais. Cada tarefa executa em laço periódico, com tempos de amostragem definidos, caracterizando um comportamento cíclico típico de sistemas de controle em tempo real.

Do ponto de vista funcional, o fluxo de dados segue a cadeia:

- Sensores → Tratamento de Sensores: amostras brutas de posição e orientação são recebidas do simulador via MQTT e pré-processadas/tratadas, sendo então registradas em um BufferCircular de posição tratada;

- Tratamento de Sensores → Planejamento de Rota: o módulo de planejamento lê as posições tratadas e os setpoints de destino provenientes da Gestão da Mina, calculando referências de velocidade e direção (setpoints de rota) que são enviados para outro buffer;

- Planejamento Rota → Controle de Navegação: a tarefa de controle de navegação lê os setpoints de rota e as posições tratadas, estima grandezas derivadas (como velocidade) e calcula setpoints de aceleração e soma angular por meio de uma lei de controle de malha fechada;

- Controle de Navegação → Lógica de Comando → Atuadores: a lógica de comando consome os setpoints de controle, aplica a lógica de segurança (incluindo reação a falhas) e publica, via MQTT, os comandos finais de aceleração e direção para o simulador;

- Monitoramento de Falhas → Demais Tarefas: a tarefa de monitoramento assina sinais de temperatura e estados de falha, interpreta as condições de operação e utiliza o NotificadorEventos para sinalizar estados de alerta/defeito às demais tarefas (tarefa_logica_comando, tarefa_controle_navegacao e tarefa_coletor_dados) que podem, por exemplo, zerar comandos ou registrar eventos;

- Coletor de Dados: coleta amostras do buffer de posição tratada e registra em arquivo para análise posterior e rastreabilidade.

Toda a integração com o ambiente externo (simulador da mina e interface unificada) é realizada por meio de tópicos MQTT específicos por caminhão (sensores, setpoints e atuadores), o que garante baixo acoplamento entre a camada de controle em tempo real (C++) e a camada de simulação/supervisão (Python). Dentro desse escopo, o sistema implementa explicitamente: comunicação assíncrona produtor–consumidor, sincronização via mutex e variáveis de condição, buffers com capacidade finita e tarefas periódicas.

3. ARQUITETURA DE SOFTWARE

A arquitetura de software do sistema foi projetada para separar camadas e responsabilidades, a lógica de controle em tempo real é implementada em C++ no processo do caminhão embarcado, enquanto simulação e supervisão em Python, comunicando exclusivamente via MQTT. Dentro do processo embarcado, o código é organizado em múltiplas tarefas (threads) que trocam dados por meio de buffers e notificações de eventos, de forma concorrente e cíclica.

3.1 VISÃO EM ALTO NÍVEL

Em alto nível, o sistema é composto por três blocos:

- Simulador da Mina – atualiza a dinâmica do caminhão em um mapa 100×100, publica sensores (posição com ruído, temperatura, *flags* de falha) e consome os comandos de aceleração/direção gerados pelo software embarcado.
- Gestão da Mina / Interface Unificada – exibe o mapa da mina, mostra a posição dos caminhões e permite ao operador criar/remover caminhão, injetar/limpar falha e definir destinos finais, publicados como setpoint_posicao_final para cada caminhão.

- Caminhão Embarcado – implementa tratamento de sensores, planejamento de rota, controle de navegação, lógica de comando, monitoramento de falhas e coleta de dados, concentrando a parte de tempo real.

A comunicação entre esses blocos é feita via tópicos MQTT separados por caminhão (atr/<id>/sensor/raw, atr/<id>/temperatura, atr/<id>/falha_*, atr/<id>/o_aceleracao, atr/<id>/o_direcao, atr/<id>/setpoint_posicao_final, etc.). O processo em C++ não conhece detalhes da implementação em Python, apenas os tópicos que deve assinar e publicar.

No main.cpp, são criados os buffers, mutexes, variáveis de condição e o notificador de eventos. Em seguida, é configurado o cliente MQTT para o CAMINHAO_ID correspondente, e então são iniciadas as *threads* das tarefas. Cada tarefa executa em laço periódico, com períodos definidos, caracterizando um comportamento cíclico típico de aplicações de controle em tempo real.

3.2 DESCRIÇÃO DAS TAREFAS

A seguir são descritas as principais tarefas do processo caminhao_embarcado, com suas funções, entradas e saídas, de acordo com a implementação.

- tarefa_tratamento_sensores
    - Função: receber dados brutos de posição/orientação provenientes do simulador e gerar uma representação tratada para uso pelas demais tarefas.
    - Entradas: mensagens MQTT de sensores brutos do caminhão.
    - Saídas: strings com posição tratada gravadas no BufferCircular de posição tratada e sinalização da variável de condição associada para acordar consumidores (coletor e planejamento/controle, quando aplicável).
- tarefa_planejamento_rota
    - Função: calcular referências de navegação (*setpoints* de velocidade e de ângulo) em função da posição atual e do destino final imposto pela Gestão da Mina.
    - Entradas: posição tratada lida do buffer de posição tratada e destino final recebido via MQTT (setpoint_posicao_final).
    - Saídas: *setpoints* de velocidade e de posição angular gravados em um buffer de *setpoints* de rota e publicação periódica da posição tratada em tópico MQTT para a Gestão da Mina.
- tarefa_controle_navegacao
    - Função: implementar o controle automático do caminhão em malha fechada, a partir dos *setpoints* de rota e da posição medida.
    - Entradas: posição tratada (buffer de posição tratada) e *setpoints* de rota (buffer de *setpoints* de rota).

    - Saídas: setpoint_aceleracao e setpoint_soma_angular calculados pela lei de controle, gravados em um buffer de *setpoints* de controle para a lógica de comando.

- tarefa_logica_comando
    - Função: consolidar os comandos finais de atuadores enviados ao simulador, aplicando regras de segurança em função do estado de falha.
    - Entradas: *setpoints* de controle (aceleração e soma angular) vindos do buffer de controle e estado de falha/notificações provenientes do NotificadorEventos.
    - Saídas: comandos de aceleração e direção publicados via MQTT (o_aceleracao e o_direcao) para o simulador e *log* em console do estado lógico.
    - Observação: O sistema opera apenas em modo automático na implementação atual.

- tarefa_monitoramento_falhas
    - Função: supervisionar continuamente sinais de temperatura e falhas específicas e gerar eventos de alerta/defeito para as demais tarefas.
    - Entradas: sinais MQTT de temperatura e *flags* de falha elétrica e falha hidráulica do caminhão.
    - Saídas: eventos disparados no NotificadorEventos (por exemplo, ALERTA_TERMICO, DEFEITO_TERMICO, FALHA_ELETRICA, FALHA_HIDRAULICA, NORMALIZACAO).

- tarefa_coletor_dados
    - Função: manter a infraestrutura para registro de dados de operação em arquivo para futura análise.
    - Entradas: notificações via variável de condição associada ao buffer de posição tratada.
    - Saídas: arquivo de *log* aberto em modo *append*, pronto para receber amostras.
    - Estado atual: A tarefa aguarda a chegada de dados, mas o trecho de código de escrita em arquivo está comentado, não estando o registro efetivo habilitado nesta versão.

Toda a supervisão é feita pela Interface Unificada em Python (Gestão da Mina + mapa da mina), que se comunica com o caminhão embarcado apenas via MQTT. Por meio dessa interface, o operador consegue acompanhar em tempo quase real a posição, a orientação e o estado de cada caminhão no mapa da mina, além de definir destinos finais individuais enviando novos setpoint_posicao_final. A mesma camada também permite acionar comandos de simulação (criação/remoção de caminhões, injeção de falhas) sem qualquer acesso direto ao código C++, o que reforça o desacoplamento entre supervisão e controle em tempo real e facilita a validação do comportamento automático do sistema.

3.3 PARTES CRÍTICAS DO CÓDIGO E SINCRONIZAÇÃO

Nesta subseção, destacam-se os trechos de código mais sensíveis para o funcionamento correto em tempo real, organizados por estruturas de comunicação entre tarefas, propagação de falhas e laços de controle. O foco está na estabilidade do controle e na gestão da concorrência.

Buffer circular e sincronização de produtores/consumidores

A classe BufferCircular é a base da comunicação produtor-consumidor entre as tarefas cíclicas. Ela é crítica por gerenciar o fluxo de dados sequenciais (posição tratada, setpoints de rota, setpoints de controle).

- Estrutura Crítica: O buffer mantém um *array* dinâmico, índices head (leitura) e tail (escrita), e a lógica de incremento circular (índice + 1) % capacidade. Um erro nessa lógica causaria a sobrescrita de dados ou a leitura de posições inválidas.
- Sincronização: Para evitar condições de corrida (*race conditions*), toda a cadeia de controle depende que o acesso ao buffer seja feito dentro de seções protegidas. Isso é garantido pelo uso de um mutex externo que assegura a exclusão mútua.
- O buffer é complementado por uma condition_variable para evitar o *busy-waiting*. Por exemplo, o Coletor de Dados só acorda quando há novas posições tratadas disponíveis. O método de escrita retorna false se estiver cheio, permitindo que a tarefa produtora (ex: Controle de Navegação) descarte a amostra e evite o bloqueio (*non-blocking*).

Notificação de falhas e uso de condition_variable

O NotificadorEventos implementa o mecanismo central de propagação de eventos de falha (assíncronos) do sistema.

- Propagação Instantânea: A tarefa de Monitoramento de Falhas converte sinais de temperatura/defeito em eventos lógicos. Ela chama disparar_evento(tipo), que atualiza o estado interno e chama notify_all().
- Sincronização: O objeto encapsula mutex e condition_variable para acordar imediatamente as *threads* que estão esperando por mudanças no estado de falha.
- As *threads* auxiliares em Controle de Navegação e Lógica de Comando chamam esperar_evento(), bloqueando em uma condition_variable. Esse mecanismo garante que qualquer falha detectada seja propagada quase instantaneamente (sem a necessidade de *polling* agressivo) para controle e atuação.

Laço de controle de navegação

O laço principal de tarefa_controle_navegacao_run() também é uma parte crítica: é ele que fecha a malha entre posição medida, setpoints de rota e comandos gerados. Qualquer erro

nesse laço impacta diretamente a estabilidade e a segurança do movimento do caminhão. A cada período, a thread:

- A thread realiza a leitura da posição tratada e a atualização dos setpoints de rota, sendo que ambas as leituras são feitas sob a proteção de mutex. Essa proteção é fundamental para que o controle não tome decisões com dados "misturados" de instantes diferentes (evitando *race conditions* lógicas).
- O laço calcula a velocidade como variação de posição dividida pelo intervalo de tempo medido com steady_clock, com checagem para evitar divisão por zero.
- lê os setpoints de rota de outro buffer compartilhado, novamente dentro de uma região crítica;
- calcula erros de velocidade e ângulo, normaliza o erro angular para o intervalo [-180°, 180°] e aplica ganhos e saturações.
- Ao final de cada iteração, o laço dorme por um período fixo, implementando uma tarefa periódica de controle

Se essa sequência fosse feita sem travas adequadas, seria possível ler uma posição antiga com um setpoint novo (ou vice-versa), caracterizando uma race condition lógica: o controle estaria tomando decisão com dados “misturados” de instantes diferentes. A proteção sistemática com mutex ao acessar buffers compartilhados reduz esse risco e mantém a coerência temporal das informações.

Lógica de segurança na atuação

Por fim, tarefa_logica_comando_run() concentra a decisão final sobre o que é enviado aos atuadores do simulador. Antes de publicar o_aceleracao e o_direcao, a tarefa verifica o estado de defeito (atualizado pelas threads de eventos) e, se houver falha ativa, força os comandos a zero.

Esse trecho implementa o comportamento fail-safe do sistema: mesmo que o controle de navegação continue calculando setpoints diferentes de zero, a atuação é bloqueada enquanto houver defeito. Também foi importante evitar *race conditions*: o acesso à flag de defeito é feito por meio de variáveis atômicas ou dentro de seções protegidas, garantindo que a lógica de comando leia o estado de falha de forma consistente e confiável..

Em resumo, as partes mais críticas do código combinam:

- estruturas de dados centrais (buffer circular e notificador de eventos);
- laços periódicos de controle;
- e sincronização cuidadosa entre threads com mutex, variáveis de condição e tipos atômicos, justamente para evitar concorrencias que poderiam comprometer a estabilidade ou a segurança do caminhão em operação.

4. AMBIENTE DE EXECUÇÃO

O sistema foi projetado para operar em um ambiente distribuído e conteinerizado, garantindo isolamento de processos, reprodutibilidade de dependências e simulação de uma rede real via Docker, onde a latência de rede, a concorrência de processos e a comunicação via mensagens são fatores determinantes para o desempenho. A execução é orquestrada da seguinte forma:

Todo o ambiente de execução é encapsulado em uma imagem Docker baseada no Ubuntu 22.04. O Dockerfile define a construção deste ambiente em camadas:

- Dependências de Sistema: Instalação de ferramentas de compilação (build-essential, cmake) e bibliotecas gráficas (SDL2, X11) para suporte à interface visual do simulador.
- Middleware de Comunicação: Compilação e instalação manual das bibliotecas Eclipse Paho MQTT (versões C e C++), essenciais para a conectividade do sistema embarcado.
- Build do Sistema Embarcado: O código fonte C++ é copiado e compilado dentro do contêiner, gerando o binário caminhao_embarcado localizado em /app/caminhao_cpp/build.
- Interface Python: Instalação das dependências Python (pygame, paho-mqtt) para execução das tarefas de Simulação e Gestão da Mina.

A topologia da rede é definida pelo docker-compose.yml, que cria uma rede virtual (net_mina) permitindo a resolução de nomes via DNS interno. O ambiente é dividido em serviços distintos:

- infra_mina: Atua como servidor central. Ele hospeda o Broker MQTT (Mosquitto) na porta 1883 e executa a simulação física (simulator_view.py). Este contêiner é configurado com acesso ao socket X11 do host (/tmp/.X11-unix), permitindo a renderização gráfica das janelas da simulação e gestão.
- caminhao_01 a caminhao_05: Instâncias isoladas que simulam os veículos. Cada contêiner executa uma cópia idêntica do binário C++, diferenciada apenas pela variável de ambiente CAMINHAO_ID, que altera o comportamento lógico e os tópicos MQTT utilizados.

O script de entrada (entrypoint) gerencia o ciclo de vida dos processos dentro dos contêineres, utilizando variáveis de ambiente para decidir qual papel o contêiner desempenhará:

- Sincronização de Inicialização: Os contêineres dos caminhões possuem uma rotina de espera ativa (wait loop), aguardando que o serviço do Broker no infra_mina esteja totalmente operacional antes de iniciar o processo C++. Isso previne falhas de conexão na partida ("CrashLoopBackOff").
- Execução Condicional: O script verifica flags como START_SIMULATOR e START_CAMINHAO. Se configurado como caminhão, ele executa o binário

/app/caminhao_cpp/build/caminhao_embarcado; se configurado como infraestrutura, inicia o Mosquitto e os scripts Python.

Dentro de cada container de caminhão, o binário C++ (main.cpp) instancia a arquitetura de software embarcado. O ambiente de execução interno utiliza std::thread para paralelismo e primitivas de sincronização para gerenciar o fluxo de dados em tempo real:

- Estrutura Compartilhada entre threads: Buffers circulares protegem a troca de dados entre as threads produtoras (ex: Tratamento de Sensores) e consumidoras (ex: Controle de Navegação), garantindo acesso seguro via std::mutex, para evitar race conditions em leituras e escritas concorrentes.
- Notificação de Eventos: O uso de std::condition_variable permite que tarefas críticas, como o controle de navegação, permaneçam em estado de espera (sleep) até que novos dados estejam disponíveis ou eventos de falha sejam disparados pelo NotificadorEventos, otimizando o uso de CPU.

5. INSTRUÇÕES DE COMPILAÇÃO E EXECUÇÃO

Caso execute os códigos no subsistema WSL, a interface gráfica com o Pygame só irá funcionar se estiver com o programa XLaunch baixado e em execução no computador, pois não existe suporte nativo para janelas GUI no WSL, portanto, interfaces gráficas só conseguem aparecer na tela com essa especificação. Após executar o XLaunch é necessário colocar os seguintes comandos no final do seu ~/.bashrc:
export DISPLAY=169.254.171.157:0.0
export LIBGL_ALWAYS_INDIRECT=1

IMPORTANTE: O IP destacado em cinza é referente ao IPV4 da sua máquina, portanto, deve ser alterado de acordo com sua especificação.

Feito isso deve ser executado *source ~/.bashrc* no terminal para que as alterações sejam confirmadas.

Novamente, todas as etapas acima devem ser seguidas caso a execução do projeto for feita no WSL.

Inicialmente, é necessário compilar os containers determinados para executá-los. No primeiro terminal execute:
sudo docker compose up --build

Em outro terminal é necessário iniciar a interface com o usuário executando:
sudo docker exec -it infra_mina python3 /app/interface_unificada/cli_gestao.py

Dessa forma, o usuário pode escolher quais ações ele quer executar apertando os números indicados no terminal.

- Escolhendo a opção 6, abrirá o mapa da mina com as instruções na tela e o que pode ser feito. Clique no mapa e tecle ‘C’, depois no terminal digite o ID do caminhão que deseja criar de 1 a 5, digite a posição inicial com os pontos que vão de 0 a 100 no eixo X e Y, e o ângulo em que o caminhão deve iniciar. Após criar o caminhão clique de volta no mapa de interface da mina e tecle ‘T’ para determinar a posição de destino do caminhão específico. A única forma de determinar a posição de destino de cada caminhão é dessa forma teclando ‘T’ no mapa.
- Para escolher a opção 1 no terminal o mapa deve estar fechado. Nessa opção o usuário pode escolher dentre 5 caminhões que deseja controlar, cada um referente a um container. Após, ele deve determinar qual o ponto de partida do caminhão no mapa de simulação. O usuário verá exibido no terminal uma tabela com a posição inicial com ruído, essa posição é atualizada a cada movimento do caminhão.
- A opção 2 no terminal remove os caminhões criados, porém, ela não remove o setpoint de destino, por exemplo, o caminhão 1 foi criado com uma posição inicial, foi dada uma posição de destino para ele, caso ele seja removido, quando ele for criado de novo, o setpoint de destino criado antes dele ser removido permanece, quando o mesmo caminhão for criado novamente ele continuará seguindo o setpoint de destino até alcançá-lo.
- A opção 3 é para injetar falha elétrica no caminhão. Após determinar um setpoint de destino usando a tecla ‘T’ no mapa, o usuário deve fechar o mapa e escolher essa opção 3. Após abrir o mapa novamente verá que o caminhão parou no ponto em que estava devido à falha elétrica. É possível observar isso no terminal de logs que está com o comando ‘sudo docker compose up --build’, o caminhão recebe o comando de falha elétrica nesses logs. Ele só volta a andar após receber um comando de limpar falha.
- A opção 4 permite ao usuário limpar a falha elétrica, e deve ser executada com o mapa fechado, basta digitar o ID do caminhão que deseja, assim que a falha sair o caminhão voltará a andar e a tentar ir para o setpoint de destino.
- A opção 5 atualiza a tabela de posições com ruídos que é exibida ao criar um caminhão e também deve ser executada com o mapa fechado. Atua como um F5 da tabela, a cada atualização mostra a posição atual com ruído.

6. DIFICULDADES E DECISÕES DO PROJETO

Ao longo do desenvolvimento foram feitas escolhas de escopo e simplificações para garantir uma entrega funcional dentro do tempo da disciplina. Abaixo estão as principais decisões, seus impactos e caminhos claros de evolução.

1. Ausência de modo manual e interface local

- Decisão: Não implementar modo manual nem interface local dedicada a um caminhão.

- Pró: Reduziu a complexidade de interação (apenas modo automático), evitando concorrência entre comandos manuais e automáticos nos mesmos atuadores e permitindo focar toda a energia em planejamento, controle e lógica de segurança.
- Contra: O sistema não demonstra a troca de controle entre modo manual e modo automático, que é um aspecto relevante em aplicações reais.
- Possível evolução: Introduzir uma interface local simples (TUI ou GUI) que publique comandos em tópicos separados e uma lógica de arbitragem na lógica de comando (ex.: prioridade de emergência manual, ou estados "AUTO/MANUAL" com transição segura).

2. Limitação a 5 caminhões em contêineres separados

- Decisão: Limitar a frota a 5 caminhões, cada um executando em seu próprio contêiner com um núcleo C++ independente.
- Pró: Deixou o isolamento de estado e a depuração mais simples; cada caminhão tem seu ID, seu conjunto de tópicos MQTT e sua própria "caixa-preta" de log, o que ajuda muito na demonstração.
- Contra: Não explora totalmente a escalabilidade para dezenas de caminhões num único processo, nem a complexidade de compartilhar recursos de CPU entre muitos veículos.
- Possível evolução: Implementar um processo único capaz de gerenciar múltiplos caminhões em threads internas, com buffers indexados por ID, ou então automatizar a criação/remoção dinâmica de contêineres a partir da interface de gestão.

3. Controle com valores inteiros e faixas simplificadas

- Decisão: Trabalhar com posição, ângulo e comandos em valores inteiros, com arredondamento e limiares simples para definir "parado/andando".
- Pró: Tornou o comportamento mais previsível e os logs mais fáceis de ler, o que é útil em um contexto.
- Contra: Perde-se suavidade de controle e precisão, e alguns efeitos de discretização aparecem no movimento (oscilações, pequenos overshoots).
- Possível evolução: Adotar representação em double em toda a cadeia, mantendo apenas quantização na interface com o simulador; ajustar ganhos do controle e faixas de saturação para obter um movimento mais suave e próximo de um sistema contínuo.

4. Implementação completa apenas da falha elétrica

- Decisão: Apesar da arquitetura suportar falhas térmica, elétrica e hidráulica, apenas a falha elétrica foi integrada até o fim (detecção, notificação e reação do controle/lógica).
- Pró: Permitiu ter pelo menos um cenário de falha completo, com parada segura, e validar bem a cadeia de NotificadorEventos + controle + lógica de comando.
- Contra: O sistema não explora outros modos de falha previstos na especificação, reduzindo a cobertura de casos de teste.
- Possível evolução: Reaproveitar o mesmo padrão já usado na falha elétrica para implementar as demais: definir limiares térmicos mais elaborados, mapear falha

hidráulica para restrições sobre direção ou movimento, e adaptar a lógica de comando para respostas diferenciadas por tipo de falha.

5. Uso extensivo de buffers e pouca limpeza/consumo

- Decisão: Criar vários BufferCircular específicos (posição tratada, setpoints de rota, setpoints de controle, estado lógico) e, em muitos casos, priorizar operações de leitura (ler) em vez de consumo (retirar), sem rotinas de limpeza global.
- Pró: Separar os canais por responsabilidade facilitou o raciocínio sobre concorrência, e o uso de ler reduziu o risco de "apagar" dados que outra tarefa ainda poderia precisar. A combinação com mutex e condition_variable já garantiu boa proteção contra race conditions.
- Contra: Alguns buffers acabam funcionando como "memória de último valor com histórico limitado"; se a taxa de produção for alta e ninguém consumir com retirar, o buffer pode ficar cheio e novas amostras podem ser descartadas (overflow silencioso).
- Possível evolução: Revisar, tarefa a tarefa, quais buffers devem ser verdadeiras filas (sempre com retirar) e quais podem ser substituídos por estruturas de "último valor" (por exemplo, uma struct protegida por mutex). Também é possível implementar políticas de limpeza controlada (ex.: descartar tudo em caso de reset ou quando a interface mandar reiniciar o caminhão).

6. Registro de dados e "caixa-preta" simplificada

- Decisão: Implementar um logger central por caminhão que escreve em um arquivo output/cam_<id>.log, mas registrar apenas eventos agregados (tratamento, planejamento, falhas), sem logar todas as amostras brutas ou o estado interno completo.
- Pró: Gera um histórico legível e curto para storytelling, mostrando as principais decisões e eventos de falha, sem gerar arquivos gigantes nem impactar o desempenho.
- Contra: Não captura cada detalhe interno do controle, o que limita um pouco a análise fina em caso de comportamento inesperado.
- Possível evolução: Aumentar a granularidade do log para incluir mais variáveis (por exemplo, erro de posição, velocidade estimada, estado das flags de falha), possivelmente com um modo "debug" configurável para não pesar na operação normal.

7. CONCLUSÃO

O trabalho desenvolvido atinge o objetivo proposto de implementar, em ambiente simulado, um sistema de controle e monitoramento para caminhões autônomos de mineração com foco em automação em tempo real. A solução integra, de forma consistente, um núcleo embarcado em C++ com múltiplas tarefas concorrentes, uma infraestrutura de comunicação via MQTT e uma camada de simulação e supervisão em Python, executando em um ambiente conteinerizado com Docker.

Do ponto de vista técnico, foram aplicados de forma explícita os principais conceitos da disciplina: uso de threads para tarefas periódicas, comunicação produtor–consumidor via buffers circulares, sincronização com mutex e condition_variable para evitar race conditions, e propagação assíncrona de falhas por meio do NotificadorEventos. O modo automático de navegação foi implementado de ponta a ponta — do tratamento de sensores ao envio dos comandos de aceleração e direção — e validado por meio da Interface Unificada, que permite visualizar o comportamento do caminhão em tempo quase real e injetar falhas elétricas no sistema.

Alguns aspectos previstos na especificação original foram simplificados ou deixados como evolução futura, como o modo manual, a interface local dedicada e a integração completa de todas as categorias de falha. Essas decisões permitiram concentrar o esforço nas partes mais críticas para o entendimento de sistemas concorrentes e de tempo real, garantindo uma entrega funcional, coerente com os requisitos centrais da disciplina e com caminhos claros de extensão para trabalhos posteriores ou versões futuras do sistema.