P.PORTO

ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO

Licenciatura em Segurança Informática

em Redes de Computadores

Sistemas Distribuídos

Trabalho Final

Maria Clara Sameiro – 8230226

João Oliveira – 8230417

Lécio Almeida - 8210077

Índice

Introdução

O presente relatório apresenta o desenvolvimento do projeto prático da unidade curricular de **Sistemas Distribuídos**, centrado na implementação de um **simulador distribuído de tráfego urbano**. Este sistema modela o comportamento de veículos, cruzamentos e semáforos numa malha urbana 3x3, permitindo analisar o fluxo de trânsito em diferentes condições de carga, bem como avaliar políticas de temporização e estratégias de escolha de percursos.

A aplicação foi desenvolvida em **Java**, recorrendo a uma arquitetura distribuída composta por **processos e threads**, com comunicação entre componentes através de **sockets**. Cada cruzamento é executado como um processo autónomo, enquanto os semáforos e as respetivas filas de veículos são geridos por threads. A simulação segue o paradigma de **eventos discretos**, possibilitando o registo de eventos, a sincronização entre entidades e a recolha de métricas relevantes sobre o desempenho do sistema.

Este relatório descreve o modelo de simulação adotado, a arquitetura implementada, as principais classes e métodos desenvolvidos, bem como a análise dos resultados recolhidos pelo dashboard em tempo real. O objetivo final é avaliar o impacto de diferentes configurações no congestionamento do sistema e nos tempos de travessia dos veículos, contribuindo para uma melhor compreensão da aplicação de conceitos de sistemas distribuídos em cenários reais.

Diagramas

1. Arquitetura Geral do Sistema Distribuído

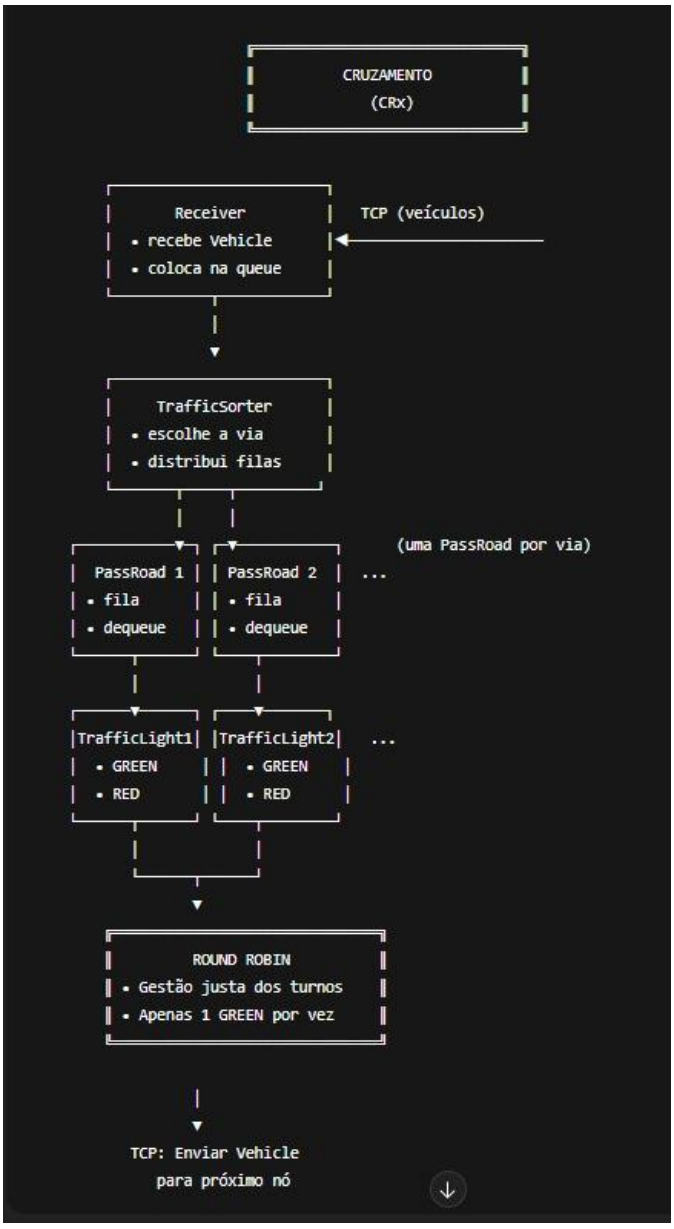

Figura 1- Diagrama da arquitetura geral do sistema distribuído

O diagrama acima representa a arquitetura distribuída do sistema, onde cada Entrada, Cruzamento e a Saída são processos independentes que comunicam via TCP. Os eventos gerados por cada nó são enviados para o EventHandler, que ordena e reencaminha esses eventos para o Dashboard, responsável pela visualização em tempo real.

2. Arquitetura Interna de um Cruzamento

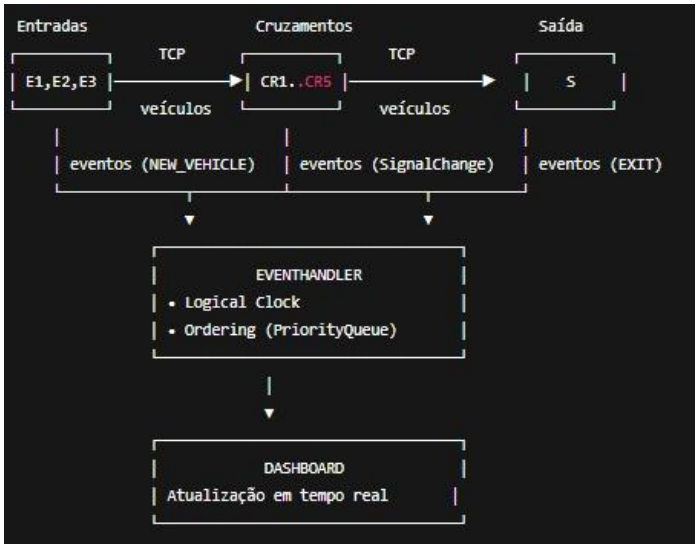

Figura 2- Diagrama da arquitetura interna de um cruzamento

Este diagrama ilustra a arquitetura interna de um cruzamento (CRx). Cada cruzamento possui threads dedicadas para receber veículos (Receiver), distribuir para as vias corretas (TrafficSorter), gerir as filas de cada via (PassRoad) e controlar os semáforos (TrafficLight). O RoundRobin assegura alternância justa entre as vias, permitindo apenas um semáforo verde de cada vez.

Modelo de Simulação

O modelo de simulação representa o fluxo de veículos numa malha urbana 3×3. Os veículos entram no sistema por três pontos de entrada (E1, E2 e E3), atravessam vários cruzamentos controlados por semáforos e terminam o percurso no ponto de saída S.

1. Estrutura da Malha

A malha é composta pelos seguintes elementos:

- **Entradas:** E1, E2, E3
- **Cruzamentos:** CR1, CR2, CR3, CR4 e CR5
- **Saída:** S
- **Ruas:** podem ser de sentido único ou duplo (dependendo do enunciado)

O **tempo de deslocamento** entre dois nós é sempre **t**, independentemente da carga da via.

2. Modelo dos Veículos

Cada veículo é representado como um objeto independente contendo:

- ID único
- Tipo: **Moto**, **Carro** ou **Camião**
- Ponto de entrada (E1, E2 ou E3)
- Tempo de chegada ao sistema
- Tempo de saída
- Percurso completo até S

2.1. Tempos de deslocamento por tipo

- Moto:

$$t_{\text{moto}} = 0.5 \times t_{\text{carro}}$$

- Carro:

$$t_{\text{carro}} = t$$

- Camião:

$$t_{\text{camião}} = 4 \times t_{\text{moto}}$$

3. Geração de Veículos (Processo de Poisson)

Cada entrada (E1, E2 e E3) possui um gerador autónomo de veículos baseado num processo estocástico de Poisson.
A taxa de chegada λ define o intervalo médio entre chegadas através de uma distribuição exponencial.

No projeto atual **não existe uma interface para seleção de carga**, pelo que o valor de λ é configurado manualmente no ficheiro de parâmetros. Alterar este valor permite simular cenários de maior ou menor carga:

- λ maior → intervalos mais curtos → mais veículos gerados
- λ menor → intervalos mais longos → menor fluxo de tráfego

Sempre que um veículo é criado:

- É atribuído aleatoriamente um tipo (moto, carro ou camião);
- É selecionado um percurso com base nas probabilidades definidas em *PathEnum*;
- O veículo é imediatamente encapsulado num **VehicleEvent** e enviado via TCP para o primeiro cruzamento do seu percurso.

4. Seleção de Caminhos

Cada entrada possui um conjunto de percursos possíveis definidos na enumeração *PathEnum*, cada um com um peso/probabilidade (*probToBeSelected*).

O processo de seleção é feito por **amostragem ponderada**:

1. Cálculo do total de probabilidade

- Recolhem-se todos os percursos que começam na entrada atual (ex.: todos os caminhos de E1)
- Soma-se o valor de *probToBeSelected* desses percursos, obtendo-se um total **T**

2. Escolha aleatória ponderada

- Gera-se um número aleatório inteiro **r** no intervalo [1, T]
- Percorrem-se os percursos acumulando os pesos
- O primeiro percurso onde a soma acumulada ≥ r é o percurso selecionado

Este mecanismo garante seleção aleatória mas proporcional às probabilidades definidas.

5. Cruzamentos (Crossroads)

Cada cruzamento funciona como um processo concorrente com múltiplas threads internas.
No cruzamento **CR4**, existe ainda um semáforo de peões gerido por uma thread própria (*PedestrianLight*), que participa no algoritmo de RoundRobin juntamente com as vias automóveis.

5.1. Threads Internas

TrafficLight

- Controla o estado do semáforo (GREEN/RED) de cada via
- Implementa RoundRobin para garantir justiça entre vias
- Gere a fila de veículos prontos a avançar
- Envia veículos para o próximo nó
- Emite SignalChangeEvent e VEHICLE_DEPARTURE

PassRoad

- Simula o tempo de deslocação no troço antes do semáforo
- Calcula o tempo com base no tipo de veículo
- Quando termina, coloca o veículo na fila do TrafficLight

TrafficSorter

- Recebe veículos do Receiver
- Determina a estrada de entrada correta com base no percurso
- Coloca o veículo na respetiva fila de PassRoad

PedestrianLight (apenas em CR4)

- Simula o semáforo de peões
- Participa no RoundRobin do cruzamento
- Mantém verde durante um intervalo fixo antes de regressar a vermelho

5.2. Semáforos

O controlo dos semáforos é garantido pelo mecanismo **RoundRobin**, que:

- Seleciona qual a via que recebe o **próximo semáforo verde**
- Garante alternância justa entre as vias
- Evita starvation
-

6. Eventos da Simulação

A simulação gera continuamente os seguintes eventos:

- **TRAFFIC_LIGHT_CHANGE**
- **NEW_VEHICLE**
- **VEHICLE_SIGNAL_ARRIVAL**
- **VEHICLE_ROAD_ARRIVAL**
- **VEHICLE_DEPARTURE**
- **VEHICLE_EXIT**

Estes eventos são usados para representação gráfica e cálculo das métricas.

7. Medidas de Desempenho

As métricas recolhidas incluem:

Contagens

- Número de veículos por tipo
- Veículos criados por tipo
- Veículos ativos por tipo

Filas

- Número atual de veículos num semáforo
- Máximo registado
- Média ao longo da simulação

Tempos

- Tempo médio de espera por tipo
- Tempo médio de passagem por tipo
- Tempo de travessia mínimo, médio e máximo por tipo de veículo

8. Critérios de Finalização

A simulação termina em dois cenários:

1. **Quando o utilizador pressiona "Stop"**
   → o sistema continua até **todos os veículos saírem** da malha
2. **Quando passam 60 segundos de tempo de simulação**
   → a simulação finaliza automaticamente, mesmo que existam veículos ativos

Arquitetura do Sistema

A arquitetura do sistema é distribuída e composta por múltiplos processos independentes que comunicam entre si através de sockets TCP. Cada Entrada, cada Cruzamento e o Nó de Saída executam como processos autónomos, garantindo paralelismo real, isolamento de falhas e elevada escalabilidade.

O sistema encontra-se estruturado em três grandes camadas:

1. **Nós Funcionais (Entradas, Cruzamentos e Saída)**
2. **Comunicação Distribuída (troca de veículos e eventos via TCP)**
3. **Camada de Monitorização e Visualização (EventHandler + Dashboard)**

1. Nós Funcionais

O sistema é constituído por três tipos principais de nós, cada um executado como um processo independente no ambiente distribuído.

Entradas (E1, E2, E3)

As Entradas são responsáveis por introduzir veículos no sistema.
Cada entrada:

- Gera veículos de forma estocástica, seguindo um Processo de Poisson;
- Atribui a cada veículo um tipo (moto, carro ou camião);
- Define um percurso probabilístico (PathEnum);
- Envia o veículo, via TCP, para o primeiro cruzamento do seu trajeto.

Cruzamentos (CR1- CR5)

Os cruzamentos gerem o fluxo automóvel da malha urbana, coordenando várias vias de entrada. Cada cruzamento contém várias threads concorrentes:

- **Receiver**: recebe veículos via TCP
- **TrafficSorter**: determina a via correta e envia para PassRoad
- **PassRoad**: simula a travessia até ao semáforo
- **TrafficLight**: controla o semáforo, gere filas e envia veículos para o próximo nó

No cruzamento **CR4**, existe adicionalmente a thread **PedestrianLight**, que integra o mesmo esquema de RoundRobin.

Saída (S)

O nó de Saída representa o fim da simulação.

Funções principais:

- Receber veículos provenientes dos cruzamentos finais
- Enviar o evento **VEHICLE_EXIT** para o EventHandler
- Finalizar logicamente o percurso do veículo

O instante exato de saída é registado e utilizado pelo Dashboard/Statistics para calcular métricas como o tempo total de viagem.

Processos Independentes

Cada nó (Entrada, Cruzamento ou Saída) funciona como um processo completamente isolado, contendo:

- O seu próprio espaço de memória;
- Conjunto de threads internas para processamento concorrente;
- Um relógio lógico (LogicalClock) que assegura a causalidade distribuída e permite ordenar eventos.

Esta independência garante um paralelismo real e evita que falhas locais interrompam toda a simulação.

2. Comunicação Distribuída

A comunicação entre os processos é realizada exclusivamente através de **sockets TCP**, garantindo:

- Fiabilidade
- Ordem nas mensagens
- Conexões ponto-a-ponto (unicast)
- Comunicação assíncrona (não bloqueante para o fluxo do sistema)

2.1. Envio de Veículos

Todos os veículos são enviados entre nós através de VehicleEvent (eventos Java serializados) transportados por sockets TCP:

- Entrada → Cruzamento
- Cruzamento → Cruzamento
- Cruzamento → Saída

Cada evento inclui:

- O objeto Vehicle
- Tipo de evento
- Identificação do nó
- Timestamp lógico

Isto permite ao Dashboard reconstruir o estado e o percurso de cada veículo**.**

2.2. Envio de Eventos

Todos os nós enviam eventos para um único processo central:

- **EventHandler (porta 8000)**

Estes eventos permitem:

- Visualização do estado
- Atualização das estatísticas
- Registo temporal e causal

A receção é central, mas o sistema continua funcional mesmo que o Dashboard deixe de responder.

3. Camada de Monitorização e Visualização

3.1. EventHandler

É o processo responsável por:

- Receber eventos de todos os nós (TCP)
- Atribuir timestamps usando logical clocks
- Ordenar eventos pela sua causalidade
- Colocar eventos numa **PriorityBlockingQueue** para o Dashboard

O EventHandler funciona como **agregador**, mas **não coordena comportamentos dos nós**, garantindo independência total dos cruzamentos.

3.2. Dashboard

O Dashboard consome os eventos ordenados e apresenta:

- Estado dos semáforos
- Localização dos veículos
- Filas nos cruzamentos
- Estatísticas globais e por tipo de veículo
- Mapa animado da cidade

Internamente, o Dashboard possui:

- Thread de consumo de eventos
- Thread gráfica (UI)
- Sistema de atualização periódica (timer)

Este componente **não interfere** na simulação — apenas observa.

4. Concorrência e Sincronização

Cada cruzamento utiliza múltiplas threads que acedem a recursos partilhados (filas, estado do sinal, RoundRobin).
Para garantir consistência e segurança, utiliza-se:

- Estruturas thread-safe (BlockingQueue)
- Monitores Java (synchronized, wait, notify)
- Algoritmo **RoundRobin** para gerir turnos das vias
- Timers individuais para cada semáforo

A sincronização evita:

- Corridas de dados
- Colisão de veículos
- Mais do que uma via simultaneamente ativa
- Deadlocks

5. Independência e Escalabilidade

A arquitetura é altamente escalável:

- Cada cruzamento é um processo totalmente independente
- Falhas locais não bloqueiam o sistema inteiro
- É possível adicionar mais cruzamentos ou entradas sem alterações profundas
- As métricas e eventos continuam a ser produzidos da mesma forma

A independência dos nós permite simulação distribuída realista e robusta.

Arquitetura de Classes e Métodos do Sistema de Simulação Distribuída

A arquitetura do sistema é modular e está organizada em vários *packages*, cada um responsável por uma parte específica da simulação.
De seguida apresenta-se uma descrição clara e estruturada das classes principais e dos métodos mais relevantes, permitindo compreender a função de cada componente no funcionamento global do simulador.

Package: Comunication

Classe: Receiver

A classe Receiver é uma *thread* responsável por receber eventos de veículos enviados por outros nós através de sockets TCP. Sempre que um evento é recebido, o recetor coloca o veículo na fila local para processamento e envia ao EventHandler central um evento de chegada, atualizando o relógio lógico com base no timestamp recebido.

Métodos

Método	Descrição
Receiver(SynchronizedQueue<Vehicle> queue, NodeEnum node, LogicalClock clock)	Construtor da classe Receiver. Recebe a fila local onde serão colocados os veículos recebidos, o nó lógico associado a este recetor e o relógio lógico usado para sincronizar os timestamps dos eventos. A porta é inferida a partir do nó.
stopReceiver()	Fecha o *server socket* do recetor, altera o estado interno e interrompe a *thread*, terminando o ciclo de receção.
run()	Ciclo principal: aceita ligações na porta configurada, lê eventos de veículos recebidos, reencaminha um evento de chegada correspondente para o EventHandler (atualizando o relógio lógico) e coloca o veículo recebido na fila local.

Classe: Sender

A classe Sender é responsável por enviar eventos (associados a veículos ou semáforos) via TCP para outros componentes do simulador, incluindo o EventHandler central e os vários nós (entradas, cruzamentos e saída).

Métodos

Método	Descrição
sendToEventHandler(Event event)	Envia um evento diretamente para o EventHandler central (porta 8000). Recebe um objeto Event serializável, cria uma ligação TCP e escreve o evento no *output stream*.
sendVehicle(Event event, int destPort)	Envia um evento para um nó específico, abrindo uma ligação TCP para a porta de destino indicada. Usado internamente para encaminhar eventos de veículos entre nós.
sendVehicleDeparture(Vehicle v, int destPort, NodeEnum node, LogicalClock clock)	Envia um evento de saída de veículo: cria um VehicleEvent do tipo VEHICLE_DEPARTURE, notifica o EventHandler central e envia o mesmo evento para o nó de destino através da porta indicada.

Package: Event

Classe: Event

Classe base para o sistema de eventos do simulador.
Cada evento possui um tipo, um nó associado e um timestamp do relógio lógico.

Métodos

Método	Descrição
Event(EventType type, NodeEnum node, long logicalClock)	Construtor base de um evento. Recebe o tipo do evento, o nó associado e o timestamp do relógio lógico.
getType()	Devolve o tipo do evento (EventType).

Método	Descrição
getNode()	Devolve o nó (NodeEnum) associado ao evento.
getLogicalClock()	Devolve o timestamp do relógio lógico do evento.
toString()	Devolve uma representação textual do evento, incluindo tipo, nó e relógio lógico.

Classe: EventHandler

Servidor TCP central que recebe objetos Event serializados enviados pelos componentes do simulador e os insere numa fila de prioridade.

Os eventos recebidos na porta PORT (8000) são colocados numa PriorityBlockingQueue<Event> para consumo pela interface gráfica ou outros componentes. Esta *thread* deve ser executada como um serviço central único dentro do *host* da simulação.

Métodos

Método	Descrição
EventHandler(PriorityBlockingQueue<Event> p, boolean running)	Cria um *event handler* que escuta na porta definida e insere os eventos recebidos na fila de prioridade. Recebe a fila de eventos e o estado inicial de execução.
stopHandler()	Para o *handler*, alterando a flag running e permitindo que o ciclo principal termine.
run()	Ciclo principal da *thread*: cria o *server socket*, aceita ligações na porta PORT, lê um objeto Event do *socket* e insere-o na fila de eventos. O método bloqueia em serverSocket.accept() até stopHandler() ser chamado.

Enum: EventType

Tipos de eventos suportados pelo simulador**.**

Tipo	Descrição
TRAFFIC_LIGHT_CHANGE	Mudança do estado de um semáforo.
NEW_VEHICLE	Criação de um novo veículo no sistema.
VEHICLE_SIGNAL_ARRIVAL	Chegada de um veículo a um semáforo.
VEHICLE_ROAD_ARRIVAL	Chegada de um veículo a um troço de estrada/interseção.
VEHICLE_DEPARTURE	Saída de um veículo de um nó em direção ao nó seguinte.
VEHICLE_EXIT	Saída definitiva do veículo do sistema (nó de saída).

Classe: SignalChangeEvent

Representa um evento em que um semáforo muda de estado para uma estrada específica.
Este evento transporta a estrada afetada, a nova cor do semáforo e o instante lógico em que ocorreu a mudança. Estende a classe Event e utiliza o tipo de evento TRAFFIC_LIGHT_CHANGE.

Métodos

Método	Descrição
SignalChangeEvent(RoadEnum road, long time, String signalColor)	Constrói um novo SignalChangeEvent com a estrada associada, o instante lógico e a nova cor (por exemplo "GREEN" ou "RED").
getSignalColor()	Devolve a nova cor do semáforo associada a este evento.
getRoad()	Devolve a estrada (RoadEnum) afetada pela mudança de sinal.
toString()	Devolve uma representação textual do evento de mudança de semáforo.

Classe: VehicleEvent

Evento relacionado com um Vehicle na simulação.
Representa ações ou alterações de estado associadas a um veículo (criação, chegada a estrada ou semáforo, partida ou saída do sistema). Regista o tipo de evento, o nó onde ocorre e o timestamp lógico herdado de Event.

Métodos

Método	**Descrição**
VehicleEvent(EventType type, NodeEnum node, long time, Vehicle vehicle)	Cria um evento associado a um veículo (por exemplo criação, chegada a estrada/sinal, saída ou saída do sistema).
getVehicle()	Devolve o veículo (Vehicle) associado ao evento.
toString()	Devolve uma representação textual do evento de veículo.

Package: Launcher

Classe: Dashboard

O Dashboard é o componente que integra toda a informação relevante do simulador num único ambiente visual, permitindo ao utilizador monitorizar, analisar e interagir com a simulação de forma intuitiva e organizada**.**

Métodos

Método	**Descrição**
Dashboard()	Constrói e inicializa toda a interface do *Dashboard*, incluindo layout, painéis gráficos, secções de estatísticas, controlador e temporizador responsável pela animação dos veículos.
createTopPanel()	Cria o painel superior da interface, que contém os botões Start e Stop e o indicador visual do estado da simulação (RUNNING/STOPPED).
createCenterContainer()	Cria a área central da interface, composta pelo renderizador gráfico (onde os veículos são

Método	Descrição
	desenhados) e pelo painel de *logs* da simulação.
createStatsContainer()	Constrói os painéis inferiores de estatísticas globais, por tipo de veículo e por cruzamento, organizados em secções independentes e roláveis.
createController()	Instancia o DashboardController, responsável por comunicar com a lógica da simulação, atualizar o ecrã, recolher estatísticas e gerir o estado visual da interface.
attachControlListeners()	Liga os botões Start e Stop às respetivas ações, permitindo ao utilizador iniciar ou parar a simulação através da interface.
startSpriteTimer()	Inicia um temporizador Swing que atualiza periodicamente as posições dos veículos (sprites), remove os que já terminaram o percurso e aciona o redesenho do mapa.
createStatSection(String title)	Cria uma secção gráfica para apresentação de estatísticas, com título e painel formatado para os dados.
makeButton(String text)	Cria e estiliza um botão com uma aparência consistente para o painel de controlo.
makeTextArea(int rows, int cols, Font font, boolean lineWrap, boolean editable)	Cria e configura uma área de texto estilizada utilizada para *logs* e visualização de informação textual.
wrapInScroll(Component comp, int vPolicy, int hPolicy)	Encapsula um componente num painel com *scroll*, definindo as políticas de rolagem vertical e horizontal.
addLabelWithGap(JPanel parent, JLabel label, int gap)	Adiciona um *label* a um painel seguido de um pequeno espaço vertical, melhorando a legibilidade.

Método	Descrição
updateStatsLabels()	Atualiza todos os valores estatísticos apresentados na interface (veículos criados, ativos, saídos, tempos médios, estatísticas por tipo e por cruzamento).
log(String s)	Adiciona uma nova entrada de *log* com timestamp na área de texto, atualizando a interface em segurança (via SwingUtilities).
main(String[] args)	Ponto de entrada da aplicação gráfica: cria uma instância de Dashboard e torna a janela visível.

Classe: DashboardController

A classe DashboardController é o componente responsável por gerir toda a lógica que liga a interface gráfica (Dashboard) ao simulador. É o "cérebro" entre a visualização e o comportamento interno do sistema distribuído.

A sua função principal é coordenar a execução da simulação, recolher os eventos produzidos pelos nós e semáforos do sistema e convertê-los em animações e estatísticas que são apresentadas ao utilizador.

Métodos

Método	Descrição
DashboardController(DashboardModel model, Map<String, VehicleSprite> sprites, Map<NodeEnum, Point> nodePositions, DashboardRenderer renderer, Consumer<String> logCb, Runnable updateStatsCb, Consumer<String> statusTextCb, Consumer<Color> statusColorCb)	Construtor que inicializa o controlador com o modelo de dados, sprites, posições dos nós, renderizador e *callbacks* para *logs*, atualização de estatísticas e estado visual. Também inicializa estruturas internas, como o agendamento de passagem por estrada (passingSchedule).

Método	Descrição
getStatistics()	Devolve o objeto Statistics que armazena todas as métricas da simulação.
startSimulation()	Inicia uma nova simulação: arranca o Simulator, prepara a fila de eventos e define o estado visual como RUNNING.
stopSimulation()	Pára a simulação imediatamente, terminando processos, consumidor de eventos, temporizadores e limpando sprites e filas.
requestGracefulStop()	Efetua uma paragem suave (*graceful stop*): termina apenas as entradas, deixa os veículos em circulação completarem o percurso e só encerra quando já não existirem veículos ativos nem eventos pendentes.
startEventConsumer()	Cria e inicia uma *thread* que consome eventos da fila produzida pelo simulador e os encaminha para handleEvent().
stopEventConsumer()	Cancela a *thread* consumidora de eventos, caso esteja ativa.
scheduleAutoStopTimer()	Configura um temporizador que, após um intervalo pré-definido, solicita uma paragem suave da simulação. Útil para execuções automáticas.

Método	Descrição
clearEventQueue()	Esvazia a fila de eventos, removendo todos os eventos pendentes.
handleEvent(Event ev)	Processa um evento recebido, delegando para os métodos específicos consoante o tipo (SignalChangeEvent, VehicleEvent, etc.).
handleSignalChange(SignalChangeEvent s)	Atualiza o estado dos semáforos no modelo e desencadeia o redesenho do dashboard.
handleNewVehicle(VehicleEvent ve, Vehicle v)	Cria um novo sprite na posição do nó de entrada, regista o veículo como criado nas estatísticas e atualiza o dashboard.
handleVehicleDeparture(VehicleEvent ve, Vehicle v)	Regista a saída de um veículo de um nó, atualiza tempos de espera, remove o veículo da fila e anima a sua deslocação para o próximo nó.
handleVehicleSignalArrival(VehicleEvent ve, Vehicle v)	Regista a chegada do veículo ao semáforo, atualiza tempos de viagem, coloca o sprite na fila de sinal correspondente e atualiza estatísticas.
handleVehicleExit(VehicleEvent ve, Vehicle v)	Indica que o veículo saiu do sistema: marca o sprite para remoção, regista o veículo como concluído e atualiza tempos finais e estatísticas globais.

Método	Descrição
handlePassRoad(VehicleEvent ve, Vehicle v)	Gere a animação da passagem de um veículo entre dois nós, calculando o tempo de deslocamento com base no tipo de veículo e evitando sobreposição visual.
roadFromPrevToNode(Vehicle v, NodeEnum node)	Determina automaticamente qual a estrada (RoadEnum) percorrida pelo veículo entre o nó anterior e o nó atual.
shutdown()	Encerra completamente o controlador, pedindo a paragem do simulador e encerrando o *executor* de *threads* de forma segura.

Classe: DashboardModel

A classe DashboardModel é o modelo de dados do dashboard e armazena toda a informação necessária para representar graficamente a simulação: sprites dos veículos, posições dos nós, estados dos semáforos, filas em cada estrada e estatísticas associadas às filas.

É a camada que liga os dados produzidos pela simulação ao que é mostrado no ecrã.

Métodos

Método	Descrição
DashboardModel()	Construtor que inicializa as estruturas: sprites, nodePositions, trafficLights, signalQueues e queueStats. Configura todas as estradas que chegam a cruzamentos com semáforo vermelho e filas vazias.

Método	Descrição
getSprites()	Devolve o mapa de todos os sprites, onde cada entrada representa um veículo animado no dashboard.
getNodePositions()	Devolve as coordenadas gráficas de cada nó, usadas para desenhar nós e veículos.
getTrafficLights()	Devolve o estado atual dos semáforos por estrada ("RED", "GREEN", etc.).
getSignalQueues()	Devolve as filas de veículos que aguardam por estrada (RoadEnum) à entrada de cruzamentos.
getQueueStats()	Devolve as estatísticas associadas às filas (tamanho máximo, médio, número de amostras, etc.).
enqueueToSignal(RoadEnum road, VehicleSprite s)	Coloca um veículo na fila do semáforo da estrada indicada, calcula a posição gráfica correta, anima a aproximação ao semáforo e atualiza as estatísticas da fila.
removeSpriteFromAllQueues(String id)	Remove um veículo de todas as filas onde se encontre. Usado quando o veículo sai do sistema ou deixa de aguardar em semáforos.
compactQueue(RoadEnum road)	Reorganiza graficamente a fila de uma estrada após a saída de um veículo, recalculando as posições de todos os elementos para evitar espaços vazios.
computeTrafficPoint(Point origin, Point dest, int index)	Função auxiliar que calcula a posição exata de um veículo na fila, com base na direção da estrada e na sua posição lógica na fila.
toString()	Devolve uma representação resumida do estado atual do modelo (por exemplo, número de sprites e número de filas).

Classe: DashboardRenderer

A classe DashboardRenderer é responsável por desenhar toda a componente gráfica do simulador: nós, estradas, sentidos de circulação, semáforos e veículos.

Funciona como o motor gráfico do dashboard, convertendo a informação do modelo em elementos visuais apresentados em tempo real.

Métodos

Método	**Descrição**
DashboardRenderer(DashboardModel model)	Inicializa o renderizador, carregando do modelo as posições dos nós, os sprites dos veículos e o estado dos semáforos. Também configura o painel (fundo, *double buffering*, *listeners* de redimensionamento e rato).
markMapDirty()	Marca o mapa estático como inválido, forçando a sua reconstrução no próximo *repaint* (útil após redimensionamento).
paintComponent(Graphics g)	Método principal de desenho: aplica *anti-aliasing*, redesenha o mapa estático se necessário, desenha semáforos e todos os veículos na posição atual.
buildStaticMap(int w, int h)	Cria uma imagem estática do mapa (estradas, setas, nós) e guarda em cache para melhorar o desempenho.
recomputeNodePositions(int panelW, int panelH)	Recalcula automaticamente as posições dos nós no ecrã com base no tamanho do painel, garantindo um layout proporcional e centrado.
computeRoadGeometries()	Calcula a geometria de todas as estradas, ajustando pontos de origem e destino para coincidir com os limites dos nós.

Método	Descrição
projectToNodeBorder(Point center, Point toward, int width, int height)	Determina o ponto exato onde a estrada toca no nó, ajustando o fim da linha ao retângulo do nó.
RoadGeom(Point from, Point to)	Classe auxiliar interna que representa o segmento gráfico de uma estrada, com pontos ajustados de início e fim.
drawRoadsStatic(Graphics2D g2)	Desenha todas as estradas no mapa, com espessura uniforme, e chama drawArrow() para indicar o sentido.
drawArrow(Graphics2D g2, RoadGeom rg)	Desenha uma seta na extremidade da estrada, representando o sentido de circulação.
drawNodes(Graphics2D g2)	Desenha todos os nós (entradas, cruzamentos, saída) com estilos de cor distintos e rótulos legíveis.
drawTrafficLightsOverlay(Graphics2D g2)	Desenha os semáforos sobre cada estrada, incluindo luz vermelha e verde, e regista áreas clicáveis para consulta de estatísticas de filas.

Classe: QueueStats

A classe QueueStats é responsável por armazenar estatísticas relacionadas ao tamanho das filas de veículos que aguardam num semáforo ou estrada. Permite calcular o tamanho máximo observado, a média de tamanho ao longo do tempo e o número de amostras registadas.

Métodos

Método	Descrição
recordSample(int size)	Regista uma nova amostra com o tamanho atual da fila. Atualiza soma, número de amostras e máximo.
getMax()	Devolve o maior tamanho de fila observado desde o início da simulação.

Método	Descrição
getAverage()	Calcula a média de tamanho da fila (soma das amostras / número de amostras). Se não houver amostras, devolve 0.0.
getSamples()	Devolve o número total de amostras registadas.

Classe: Simulator

A classe Simulator é responsável por iniciar, coordenar e terminar todas as entidades que compõem a simulação distribuída: Entradas, Cruzamentos, Saída e o EventHandler.
Funciona como o mecanismo de execução global, controlando o ciclo de vida da simulação e a comunicação entre processos.

Métodos

Método	Descrição
Simulator()	Inicializa o simulador, criando a fila de eventos, o mapa de processos e determinando o comando Java a utilizar para lançar os nós externos.
startSimulation()	Inicia toda a simulação: arranca o EventHandler, cria os processos para Entradas, Cruzamentos e Saída e aguarda a inicialização dos componentes.
startEntranceProcess(NodeEnum entrance, String classpath, File workDir)	Inicia um processo externo que executa o nó de entrada dado (E1, E2, E3) e regista o processo internamente.
startExitProcess(NodeEnum exit, String classpath, File workDir)	Inicia o processo correspondente ao nó de saída (S).
startCrossroadProcess(NodeEnum crossroad, String classpath, File workDir)	Inicia um processo para cada cruzamento (CR1–CR5).
stopAllProcesses()	Encerra todos os processos ativos, tentando primeiro uma paragem normal e recorrendo a destroyForcibly() caso necessário.

Método	Descrição
stopSimulation()	Termina completamente a simulação, parando todos os processos dos nós e o EventHandler e atualizando o estado running.
stopEntranceProcesses()	Pede uma paragem suave das entradas, encerrando apenas os processos dos nós de entrada para impedir novos veículos de entrarem.
isRunning()	Indica se a simulação está atualmente ativa.
getEventQueue()	Devolve a fila de eventos (PriorityBlockingQueue<Event>) usada pelo dashboard para processar a simulação.

Classe: Statistics

A classe Statistics é responsável por reunir e calcular todas as métricas relacionadas ao desempenho da simulação: veículos criados, veículos concluídos, tempos de espera nos semáforos, tempos de deslocamento entre nós, tempos totais de viagem, estatísticas por tipo de veículo e por nó.

Métodos

Método	Descrição
recordCreatedVehicle(Vehicle v)	Regista a criação de um novo veículo e incrementa a contagem total e por tipo.
recordExitedVehicle(Vehicle v)	Regista que um veículo saiu do sistema, atualizando a contagem total e a contagem por tipo.
recordPassedAtNode(NodeEnum node, Vehicle v)	Regista que um veículo passou por um determinado nó, contabilizando por tipo de veículo.
recordDepartureTimestamp(String id)	Guarda o timestamp em que um veículo saiu de um nó (início de deslocamento entre nós).

Método	Descrição
removeDepartureTimestamp(String id)	Remove e devolve o timestamp de saída armazenado para um veículo.
recordEntranceTimestamp(String id, long timestamp)	Regista o instante em que o veículo entrou no sistema.
recordSignalArrival(String id)	Regista o timestamp de chegada de um veículo ao semáforo.
removeSignalArrival(String id)	Remove e devolve o timestamp de chegada ao semáforo para um veículo.
recordTravelTime(Vehicle v, long ms)	Regista o tempo de deslocamento entre nós (tempo de estrada), atualizando estatísticas por tipo de veículo.
recordTripTimeByType(Vehicle v)	Calcula e regista o tempo total de travessia do sistema (entrada → saída), atualizando min, max e média por tipo.
recordWaitForType(VehicleType vt, long waitMs)	Regista o tempo de espera de um veículo num semáforo, agrupando por tipo.
getTotalCreated()	Devolve o total de veículos criados.
getTotalExited()	Devolve o total de veículos que concluíram o percurso.
getTotalTravelTimeMs()	Devolve o tempo total acumulado de deslocamentos entre nós.
getCompletedTrips()	Devolve o número de viagens completas registadas.
getCreatedByType()	Devolve um mapa com o número de veículos criados por tipo.
getExitedByType()	Devolve um mapa com o número de veículos terminados por tipo.
getAvgWaitByType()	Calcula a média do tempo de espera nos semáforos por tipo de veículo.

Método	Descrição
getPassedByNodeByType()	Devolve o número de veículos de cada tipo que passou por cada nó.
getAvgRoadByTypeSeconds()	Calcula o tempo médio de deslocamento entre nós (em segundos) por tipo de veículo.
getTripStatsMillis()	Devolve, por tipo de veículo, os tempos mínimo, médio e máximo de viagem (em milissegundos).

Classe: VehicleSprite

A classe VehicleSprite representa graficamente cada veículo no dashboard. Armazena posição, direção, estado de animação e propriedades visuais, permitindo mostrar o movimento dos veículos em tempo real.

Métodos

Método	Descrição
VehicleSprite(String id, Vehicle vehicle, double x, double y)	Constrói o sprite de um veículo, definindo o ID, o veículo associado e a posição inicial.
setTarget(double tx, double ty, long durationMs)	Define o destino do veículo e inicia uma animação suave até à nova posição ao longo do tempo dado.
setFaceTarget(double fx, double fy)	Define um alvo para o qual o veículo deve "olhar", ajustando o ângulo do sprite.
clearFaceTarget()	Remove o alvo de orientação, permitindo que o ângulo seja calculado automaticamente pelo movimento.
markForRemoval()	Marca o sprite para remoção quando terminar a animação atual.
shouldRemoveNow()	Indica se o sprite já deve ser removido do dashboard.

Método	Descrição
updatePosition()	Atualiza a posição e orientação do veículo ao longo da animação; também corrige rotação quando parado mas com alvo de orientação.
draw(Graphics2D g2)	Desenha o sprite do veículo no ecrã, aplicando transformações, cor, forma e *label* com o ID.
clamp(double v, double a, double b)	Função auxiliar que limita um valor a um intervalo [a, b], usada nos cálculos de interpolação.

Package: Node

Classe: Crossroad

A classe Crossroad representa um nó de cruzamento do sistema. É responsável por inicializar semáforos, filas sincronizadas, *threads* PassRoad, coordenadores RoundRobin e o TrafficSorter.
Pode operar em modo multi-sinal ou de sinal único, consoante o número de estradas de entrada.

Métodos

Método	Descrição
Crossroad(NodeEnum crossroad)	Constrói e inicia o nó de cruzamento correspondente ao identificador fornecido.
start()	Inicializa e arranca os componentes do cruzamento, escolhendo entre modo multi-sinal (várias entradas) ou modo de sinal único (uma entrada).
startMultipleSignals()	Configura o cruzamento com múltiplos semáforos: cria RoundRobin, filas por estrada, *threads* PassRoad e TrafficLight, e arranca um Receiver + TrafficSorter.
startSingleSignal()	Configuração simplificada para cruzamentos com uma única estrada de entrada: cria um PassRoad, um TrafficLight, um PedestrianLight e um Receiver.
main(String[] args)	Ponto de entrada que arranca uma instância de cruzamento com base no identificador ("CR1", etc.).

Classe: Entrance

A classe Entrance representa um nó de entrada. É responsável por gerar veículos periodicamente, usando uma distribuição exponencial para os tempos de chegada, atribuir tipo e percurso a cada veículo e enviar eventos de criação e partida.

Métodos

Método	Descrição
Entrance(NodeEnum entrance)	Constrói e inicia o nó de entrada com base no identificador fornecido.
start()	Ciclo principal: gera veículos, regista o evento NEW_VEHICLE e envia o veículo para o primeiro cruzamento do percurso.
getProbabilitySum()	Calcula a soma dos pesos/probabilidades dos percursos possíveis a partir desta entrada.
selectPath()	Seleciona aleatoriamente um percurso (PathEnum) com base nas probabilidades configuradas.
generateVehicle()	Cria um novo veículo com tipo aleatório e o percurso selecionado.
getExponentialInterval()	Calcula o intervalo (em ms) para a próxima geração de veículo, usando uma distribuição exponencial.
main(String[] args)	Arranca o processo da entrada com o identificador fornecido (por exemplo "E1").

Classe: Exit

A classe Exit representa o nó de saída do sistema. Recebe veículos encaminhados pelos cruzamentos, processa a sua chegada final e envia um evento VEHICLE_EXIT para o EventHandler, registando a conclusão da viagem.

Métodos

Método	Descrição
Exit(NodeEnum exit)	Constrói e inicia o nó de saída correspondente ao identificador fornecido.

Método	Descrição
start()	Inicializa filas por estrada, cria *threads* PassRoad, Receiver e TrafficSorter, e para cada veículo concluído envia um evento VEHICLE_EXIT ao EventHandler.
main(String[] args)	Arranca o processo de saída com o identificador fornecido (por exemplo "S").

Enum: NodeEnum

A enumeração NodeEnum representa todos os nós do sistema (entradas, cruzamentos, saída), cada um associado a um tipo (NodeType) e a um porto TCP.

Métodos

Método	Descrição
NodeEnum(NodeType type, int port)	Construtor interno que define o tipo de nó e o porto correspondente.
toNodeEnum(String nodeString)	Converte uma string (por exemplo "E1") no valor correspondente do enum.
getPort()	Devolve o porto TCP utilizado pelo nó.
getType()	Devolve o tipo do nó (ENTRANCE, EXIT, CROSSROAD).
getEntrances()	Devolve uma lista com todos os nós de entrada.
toString()	Devolve o nome textual do nó (igual ao nome do enum).

Enum: NodeType

Enumeração que define os tipos de nós existentes no sistema.

Tipo	Descrição
ENTRANCE	Nó de entrada, responsável por gerar veículos para o sistema.
EXIT	Nó de saída, responsável por recolher e terminar veículos.

Tipo	Descrição
CROSSROAD	Nó de cruzamento onde se dá a gestão de semáforos e fluxos de tráfego.

Package: Traffic

Classe: PassRoad

A classe PassRoad representa a *thread* responsável por gerir a passagem de veículos num troço de estrada. Lê veículos da fila de chegada, calcula o tempo de travessia com base no tipo de veículo e no tempo base da estrada, e quando o tempo simulado termina envia um evento de chegada ao semáforo seguinte (VEHICLE_SIGNAL_ARRIVAL).

Métodos

Método	Descrição
PassRoad(SynchronizedQueue<Vehicle> arrivingQueue, SynchronizedQueue<Vehicle> passedQueue, RoadEnum road, LogicalClock clock)	Cria uma nova *thread* PassRoad associada a uma estrada, definindo filas de chegada e saída e o relógio lógico.
run()	Ciclo principal: lê veículos que chegam, agenda o tempo de passagem e processa os veículos cuja travessia terminou.
processNewArrival(Vehicle vehicle)	Agenda a passagem de um veículo recém-chegado, calculando o instante em que termina a travessia e garantindo a ordem de passagem.
processPassedRoad()	Processa o próximo veículo cuja travessia terminou: envia evento VEHICLE_SIGNAL_ARRIVAL para o EventHandler e adiciona o veículo à fila de veículos já passados.

Classe: PedestrianLight

A classe PedestrianLight implementa um controlador de semáforo pedonal simples. Coopera com um coordenador RoundRobin para obter o seu turno de verde, manter um intervalo de passagem de peões e depois voltar a vermelho, repetindo o ciclo.

Métodos

Método	Descrição
PedestrianLight(RoundRobin roundRobin, int id)	Cria um controlador de semáforo pedonal associado a um identificador no esquema RoundRobin.
run()	Ciclo principal: espera pelo turno no RoundRobin, ativa o verde durante um período fixo, imprime mensagens no *console* e depois passa a vermelho, libertando o turno.

Enum: RoadEnum

A enumeração RoadEnum representa todas as estradas do sistema. Cada estrada liga um nó de origem a um nó de destino, tem um tempo base de deslocação e pode ter uma duração de verde associada.

Métodos

Método	Descrição
RoadEnum(NodeEnum origin, NodeEnum destination, int timeToTravel)	Construtor interno que cria uma estrada com tempo base de viagem e sem duração de verde explícita.
RoadEnum(NodeEnum origin, NodeEnum destination, int timeToTravel, int greenLightDuration)	Construtor interno que cria uma estrada com tempo base de viagem e duração de verde associada.
toRoadEnum(String roadStr)	Converte uma string (ex.: "E1_CR1") no valor correspondente de RoadEnum, ou devolve null se não existir.

Método	Descrição
getOrigin()	Devolve o nó de origem da estrada.
getDestination()	Devolve o nó de destino da estrada.
getTime()	Devolve o tempo base (ms) para atravessar a estrada.
getGreenLightDuration()	Devolve a duração do verde (ms) configurada para essa estrada.
getRoadsToCrossroad(NodeEnum node)	Devolve a lista de estradas que terminam no nó indicado.
getRoadsFromCrossroad(NodeEnum node)	Devolve a lista de estradas que têm origem no nó indicado.
toString()	Devolve o nome textual da estrada (igual ao nome do enum).

Classe: TrafficLight

A classe TrafficLight representa o controlador de semáforo para uma estrada de entrada num nó (tipicamente um cruzamento). Utiliza um coordenador RoundRobin para obter o seu tempo de verde, deixa passar veículos enquanto houver tempo disponível e envia eventos de mudança de sinal e de partida de veículos para o nó seguinte.

Métodos

Método	Descrição
TrafficLight(SynchronizedQueue<Vehicle> vehicleQueue, RoadEnum road, LogicalClock clock, RoundRobin roundRobin)	Cria um controlador de semáforo para uma estrada específica, associando a fila de veículos, o relógio lógico e o coordenador RoundRobin.

Método	Descrição
run()	Ciclo principal: espera pelo turno no RoundRobin, ativa o verde, envia eventos de SignalChangeEvent (GREEN/RED), permite a passagem de veículos durante o intervalo verde e liberta o turno.
handleGreenLight(long greenEndTime)	Durante o período verde, permite a passagem de veículos desde que o tempo de travessia caiba na janela de verde restante. Caso contrário, termina o verde.
handleDeparture()	Processa a partida de um veículo: remove da fila, determina o próximo nó com base no percurso e envia a mensagem de partida (sendVehicleDeparture) para esse nó.

Classe: TrafficSorter

A classe TrafficSorter é responsável por distribuir veículos recém-chegados para as filas corretas por estrada num determinado nó. A partir do percurso do veículo, identifica o nó anterior, constrói a estrada correspondente e encaminha o veículo para a fila que será processada por PassRoad/TrafficLight.

Métodos

Método	Descrição
TrafficSorter(Map<RoadEnum, SynchronizedQueue<Vehicle>> trafficQueues, SynchronizedQueue<Vehicle> vehiclesToSort, NodeEnum node)	Cria um TrafficSorter associando-o a um nó, a uma fila de chegada partilhada e a um conjunto de filas por estrada.
run()	Ciclo principal: retira veículos da fila de chegada, determina a estrada de entrada com base no nó anterior do percurso e adiciona o veículo à fila correspondente.

Package: Utils

Classe: LogicalClock

A classe LogicalClock implementa um relógio lógico simples (Lamport), usado para garantir ordenação causal entre eventos distribuídos.

Métodos

Método	Descrição
tick()	Incrementa o relógio lógico e devolve o novo valor.
update(long received)	Atualiza o relógio garantindo ordem causal: time = max(time, received) + 1.
get()	Devolve o valor atual do relógio lógico.

Classe: RoundRobin

A classe RoundRobin funciona como um coordenador de turnos para *threads* cooperantes. É usada em cruzamentos para gerir semáforos, garantindo que cada estrada recebe o "seu turno" de verde de forma sequencial.

Métodos

Método	Descrição
RoundRobin(int totalThreads)	Cria um coordenador round-robin com um número fixo de participantes.
esperarTurno(int id)	Bloqueia a *thread* até ser o seu turno (id).
terminarTurno()	Avança para o próximo turno e acorda todas as *threads* em espera.

Classe: SynchronizedQueue<E>

A classe SynchronizedQueue é uma fila sincronizada usada para comunicação segura entre *threads*.
Fornece operações bloqueantes e não bloqueantes, adequadas ao processamento concorrente de veículos, eventos e mensagens entre módulos.

Métodos

Método	Descrição
add(E element)	Adiciona um elemento à fila e notifica as *threads* em espera.
remove()	Remove e devolve o primeiro elemento; bloqueia se a fila estiver vazia.
poll()	Remove e devolve o primeiro elemento sem bloquear (pode devolver null).
peek()	Devolve o primeiro elemento sem o remover (ou null se estiver vazia).
peekLast()	Devolve o último elemento da fila (ou null se estiver vazia).

Package: Vehicle

Enum: PathEnum

A enumeração PathEnum define todos os percursos possíveis que um veículo pode seguir no sistema.
Cada percurso contém uma sequência fixa de nós (entradas, cruzamentos e saída) e uma probabilidade associada, usada pelas entradas para seleção aleatória ponderada.

Métodos

Método	Descrição
PathEnum(int probToBeSelected)	Construtor interno que define o peso/probabilidade com que o percurso pode ser escolhido.
getPath()	Devolve a lista ordenada de nós que compõem o percurso.

Método	Descrição
getPathsFromEntrance(NodeEnum entrance)	Devolve todos os percursos que se iniciam no nó de entrada indicado.
toString()	Devolve uma representação legível do percurso (ex.: E1 → CR1 → CR4 → CR5 → S).
getProbToBeSelected()	Devolve o peso/probabilidade associada ao percurso para seleção aleatória.

Classe: Vehicle

A classe Vehicle representa um veículo no sistema.
Guarda o identificador, o tipo de veículo, o percurso a seguir e os timestamps de entrada e saída do sistema, além de métodos auxiliares relacionados com o percurso.

Métodos

Método	Descrição
Vehicle(String id, VehicleType type, PathEnum path)	Cria um veículo com ID único, tipo e percurso associado.
getId()	Devolve o identificador do veículo.
getType()	Devolve o tipo do veículo (VehicleType).
getEntranceTime() / setEntranceTime(long)	Obtém/define o timestamp de entrada do veículo no sistema.
getExitTime() / setExitTime(long)	Obtém/define o timestamp de saída do veículo do sistema.
getPath()	Devolve o percurso (PathEnum) associado ao veículo.
findNextNode(NodeEnum current)	Devolve o próximo nó no percurso, dado o nó atual, ou null se não existir.
findPreviousNode(NodeEnum current)	Devolve o nó anterior no percurso, dado o nó atual, ou null se não existir.

Enum: VehicleType

A enumeração VehicleType define os tipos de veículos suportados pelo simulador. Cada tipo contém um multiplicador que ajusta o tempo base necessário para atravessar uma estrada.

Métodos

Método	Descrição
VehicleType(double multiplier)	Construtor interno que define o multiplicador de tempo para o tipo de veículo.
getMultiplier()	Devolve o multiplicador aplicado ao tempo base da estrada.
getTimeToPass(long baseTimeMs)	Calcula o tempo necessário para o veículo atravessar uma estrada, ajustando o tempo base pela sua característica.
getTypeToString()	Devolve o tipo em texto ("Car", "Truck", "Motorcycle"), usado em estatísticas e UI.

Dashboard

O Dashboard é o componente responsável por apresentar, em tempo real, o estado da simulação distribuída de tráfego. Ele funciona como um cliente visual que recebe eventos enviados pelos vários processos (Entradas, Cruzamentos, Semáforos e Saída), transformando-os em animações, métricas e indicadores gráficos.

1. Como o Dashboard Funciona

O Dashboard opera como um **cliente passivo**, recebendo e processando eventos enviados por todos os processos da simulação. Para garantir ordem temporal e consistência num ambiente distribuído, os eventos são tratados de forma sequencial.

Fluxo operacional:

1. **Os processos da simulação** enviam eventos TCP para o *EventHandler*.
2. O **EventHandler** recebe e guarda todos os eventos numa *PriorityBlockingQueue* ordenada pelo **timestamp lógico**.
3. O **DashboardController** executa uma thread consumidora que lê continuamente eventos desta fila.
4. Cada evento é tratado pelo método **handleEvent()**, que atualiza:
    - a interface gráfica (DashboardRenderer),
    - o modelo de dados (DashboardModel),
    - as estatísticas (Statistics),
    - e os logs apresentados ao utilizador.

Este mecanismo garante que a visualização está sempre sincronizada com o estado real da simulação.

2. Métricas Apresentadas pelo Dashboard

As métricas são organizadas em três grandes grupos: gerais, por tipo de veículo e por cruzamento.

2.1. Estatísticas Gerais

Métrica	Significado
Created	Total de veículos criados desde o início. Representa a carga de tráfego inserida no sistema.
Active	Número de veículos atualmente dentro da rede. Indica nível de congestionamento em tempo real.
Exited	Veículos que concluíram o percurso e saíram do sistema.
Avg Trip (s)	Tempo médio de viagem completa (entrada → saída). Avalia a eficiência global da rede.

2.2. Estatísticas por Tipo de Veículo

O Dashboard apresenta estatísticas separadas para:

- **CAR**
- **TRUCK**
- **MOTORCYCLE**

Métrica	Descrição
Created by type	Quantos veículos desse tipo foram gerados.
Active by type	Quantos veículos desse tipo estão na rede.
Exited by type	Quantos concluíram o percurso.
Avg Wait (s)	Tempo médio de espera em sinais de trânsito.
Avg Road (s)	Tempo médio de travessia das estradas.
Trip (min/avg/max)	Estatísticas completas da viagem: mais curta, média e mais longa.

Estas métricas permitem estudar diferenças de desempenho entre tipos de veículos.

2.3. Estatísticas por Cruzamento

Para cada cruzamento (CR1 a CR5) o Dashboard mostra:

- Número total de veículos que passou por esse cruzamento.
- Distribuição por tipo de veículo.

Permite identificar:

- gargalos,
- rotas mais utilizadas,
- cruzamentos mais congestionados.

3. Como o Dashboard Recebe Dados dos Processos

Todos os dados chegam ao Dashboard sob a forma de **eventos TCP**, enviados pelos nós da simulação.

Eventos associados a veículos (VehicleEvent)

- **NEW_VEHICLE** – quando um veículo é criado numa entrada.
- **VEHICLE_DEPARTURE** – quando sai de um nó para o próximo.
- **VEHICLE_SIGNAL_ARRIVAL** – quando chega a um semáforo.
- **VEHICLE_ROAD_ARRIVAL** – quando o veículo chega ao início de um novo troço de estrada, imediatamente antes de iniciar a sua travessia;
- **VEHICLE_EXIT** – quando sai definitivamente do sistema.

Eventos associados a sinais (SignalChangeEvent)

- Mudanças do estado do semáforo (VERDE / VERMELHO).

Cada evento desencadeia uma alteração visual imediata no Dashboard.

4. Gráficos e Indicadores Usados

4.1. Visualização do Mapa

- Nós da rede representados graficamente:
    - **Verde** – Entradas
    - **Amarelo** – Cruzamentos
    - **Vermelho** – Saída
- Estradas desenhadas com setas indicando direção.

4.2. Animação dos Veículos

- Cada veículo é um retângulo colorido:
    - Cor depende do tipo (car, truck, motorcycle).
- A posição é atualizada continuamente.
- Filas de espera são representadas junto aos semáforos.

4.3. Indicadores de Semáforos

Os semáforos aparecem sobre cada estrada no mapa e são interativos: ao clicar num semáforo, abre-se um painel com:

- tamanho atual da fila
- tamanho máximo observado
- média da fila
- número de amostras recolhidas

Permite identificar facilmente estradas congestionadas

4.4. Área de Logs

- Lista cronológica de todos os eventos recebidos.
- Ferramenta essencial para analisar comportamento e depurar o sistema.

5. Interpretação das Métricas

As métricas permitem analisar:

Métrica	Interpretação
Avg Trip	Avalia a eficiência global do sistema. Valores altos → congestionamento.
Avg Wait	Mede o impacto dos semáforos e gestão de cruzamentos.
Active	Observação em tempo real do congestionamento.
Fluxo por cruzamento	Identificação de gargalos e sobrecarga.
Comparação por tipo de veículo	Estuda prioridades, velocidades e impacto de veículos lentos.

Estas métricas permitem tomar decisões sobre otimização, detetar problemas e avaliar políticas alternativas.

6. Comparação de Políticas

O Dashboard permite comparar execuções variando **apenas a duração do verde dos semáforos**, conforme previsto no enunciado.

No projeto atual **não existe mecanismo automático** para alterar políticas — **os tempos de verde são modificados manualmente nos ficheiros de configuração**.

A alteração da duração do verde impacta métricas como:

- tempo médio de viagem (Avg Trip)
- tempo médio de espera nos sinais (Avg Wait)
- tamanho das filas
- tempos médios em estrada por tipo de veículo

Outras políticas (prioridades, adaptação dinâmica, etc.) **não estão implementadas**.

Screenshots do Dashboard

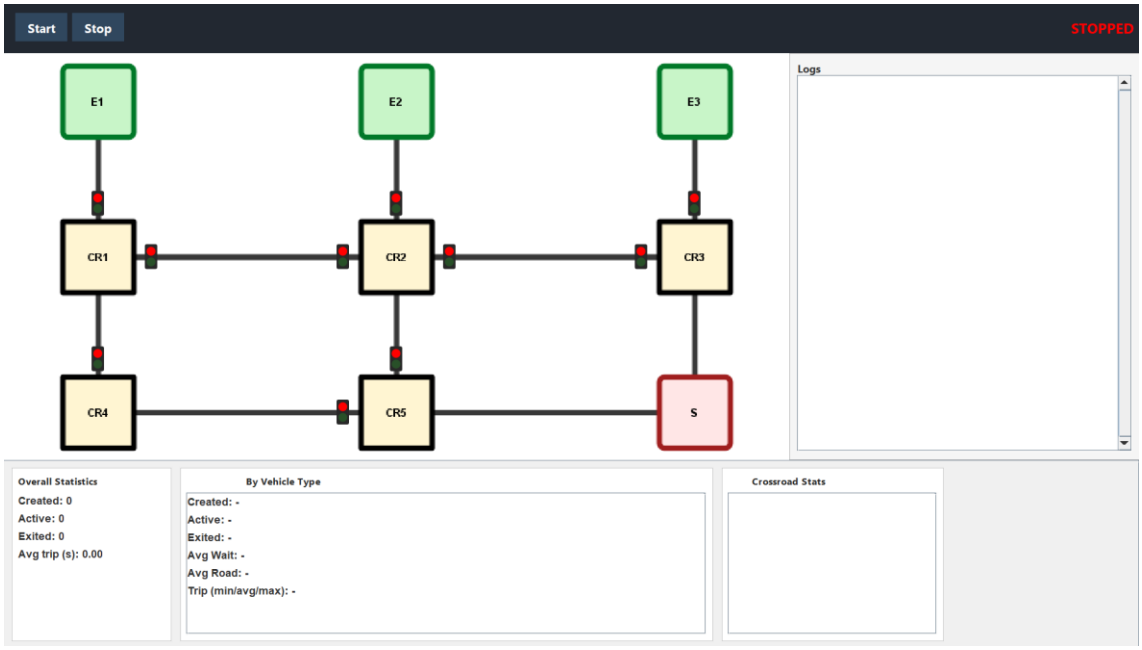

Figura 3- Dashboard antes da simulação

Nesta fase observa-se:

- Layout da malha de tráfego
- Semáforos representados graficamente
- Painéis de métricas e logs ainda vazios
- Botões *Start* e *Stop*

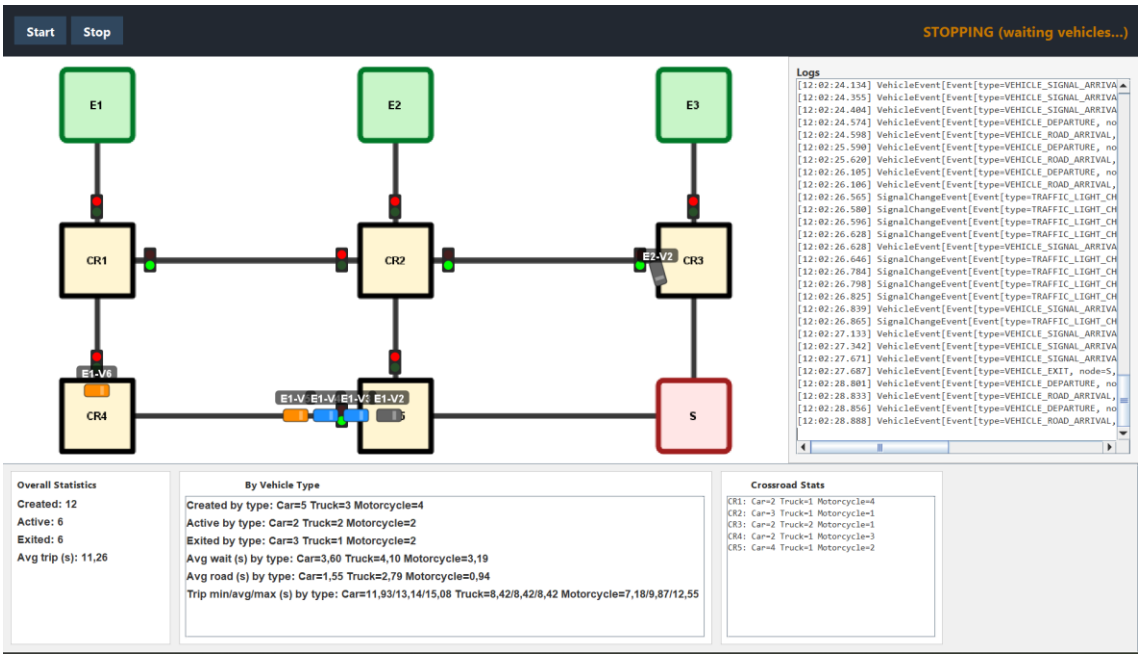

Figura 4-Dashboard depois de começar a simulação

Evidencia os seguintes comportamentos:

- Veículos animados (sprites) em movimento
- Atualização contínua do estado dos semáforos
- Logs de eventos (chegadas, partidas, mudanças de sinal)
- Atualização das estatísticas gerais, por tipo de veículo e por cruzamento

Análise dos Resultados das Simulações

Foram realizados testes variando o nível de carga (baixa, média e alta) e o tempo de semáforo (3 s, 6 s e 9 s). Para cada combinação, analisaram-se o **tempo médio de viagem**, o **tempo médio de espera** e o comportamento por tipo de veículo.

Resultados por Carga

1. Baixa Carga

Tempo Semáforo	Tempo Médio Viagem (s)	Observações
3 s	**21,67**	Melhor desempenho geral; tempos de espera mais baixos para todos os veículos.
6 s	41,35	Desempenho intermédio.
9 s	42,18	Pior desempenho para baixa carga.

2. Média Carga

Tempo Semáforo	Tempo Médio Viagem (s)	Observações
3 s	**49,29**	Melhor desempenho; tempos de espera equilibrados.
6 s	50,73	Desempenho intermédio.
9 s	52,00	Pior desempenho.

3. Alta Carga

Tempo Semáforo	Tempo Médio Viagem (s)	Observações
3 s	49,48	Tempo de viagem mais alto; camiões com espera elevada.
6 s	48,89	Desempenho intermédio; equilíbrio entre viagem e espera.

Tempo Semáforo	Tempo Médio Viagem (s)	Observações
9 s	**47,83**	Menor tempo médio de viagem, mas camiões têm maior tempo de espera (16,22 s).

Análise Detalhada

Baixa Carga

O tempo de semáforo de **3 segundos** é claramente o mais eficiente.
Apresenta:

- Menor tempo médio de viagem (21,67 s)
- Tempos de espera reduzidos
- Maior fluidez devido aos ciclos curtos

Ciclos longos (6 s e 9 s) tornam o sistema menos responsivo quando existe pouco tráfego.

Média Carga

Novamente, **3 segundos** fornece:

- O **melhor tempo médio de viagem** (49,29 s)
- Tempos de espera equilibrados entre os veículos
- Melhor adaptação ao fluxo moderado

Os tempos de 6 s e 9 s aumentam o tempo total de viagem sem benefícios adicionais.

Alta Carga

Com carga elevada, o comportamento muda:

- **9 segundos** apresenta o menor tempo médio de viagem (47,83 s)
- Porém, camiões sofrem **tempos de espera muito elevados** (16,22 s)
- O tempo de **6 segundos** é o mais equilibrado, mesmo que o tempo médio seja ligeiramente superior

Isto evidencia que, em alta carga, ciclos mais longos mantêm as vias abertas tempo suficiente para escoar filas maiores, mas penalizam veículos mais lentos (como camiões).

Recomendação Final

A política ideal de temporização dos semáforos deve adaptar-se ao nível de carga do sistema:

1. Baixa Carga → 3 segundos

- Maior fluidez
- Menor tempo médio de viagem
- Menor espera para todos os tipos de veículo

2. Média Carga → 3 segundos

- Melhor desempenho global
- Evita atrasos desnecessários
- Mantém o sistema responsivo e eficiente

3. Alta Carga → 6 segundos

- Melhor equilíbrio entre eficiência e justiça
- Evita tempos de espera excessivos para camiões
- Mantém boa fluidez sem desigualdades severas

Se a prioridade absoluta for apenas minimizar o tempo médio de viagem, mesmo sacrificando camiões → **9 segundos**.

Recomenda-se a implementação de **semáforos adaptativos**, capazes de ajustar automaticamente o tempo de ciclo consoante a carga detetada em tempo real. Esta abordagem maximiza a eficiência e reduz desigualdades entre tipos de veículos.

Conclusão

O desenvolvimento deste simulador distribuído permitiu aplicar, de forma prática, conceitos fundamentais de Sistemas Distribuídos, como comunicação entre processos, sincronização através de relógios lógicos, concorrência e processamento paralelo. A arquitetura criada demonstrou ser capaz de representar uma malha urbana dinâmica e de suportar diferentes cenários de carga, possibilitando a análise de políticas de temporização dos semáforos e do seu impacto no fluxo de tráfego.

Durante o desenvolvimento, **as maiores dificuldades surgiram na implementação do Dashboard e na gestão da lógica de eventos**. O Dashboard exigiu a coordenação de múltiplas threads, animação em tempo real, atualização contínua de métricas e sincronização rigorosa com o estado lógico da simulação. Por outro lado, a lógica de eventos distribuídos implicou garantir a ordem causal, evitar conflitos na receção simultânea de mensagens e assegurar que todos os nós operavam de forma consistente. Apesar destes desafios, o resultado final revelou-se estável, funcional e essencial para a compreensão visual do comportamento da simulação.

A análise dos testes realizados mostrou que a temporização dos semáforos tem impacto direto na eficiência do sistema. Verificou-se que **ciclos curtos (3 s)** são mais adequados para cargas baixa e média, promovendo fluidez e reduzindo tempos de espera, enquanto **ciclos intermédios (6 s)** proporcionam melhor equilíbrio em cenários de alta carga, evitando penalizações excessivas para veículos mais lentos, como camiões. Estes resultados evidenciam a importância de adaptar a política de semáforos ao volume de tráfego existente.

Trabalho Futuro

O projeto abre espaço para várias melhorias que poderiam tornar o simulador mais flexível e poderoso, nomeadamente:

- **Dashboard mais completo**, permitindo ao utilizador selecionar o nível de carga, alterar tempos de semáforo e ajustar a simulação diretamente pela interface.
- **Simulação mais dinâmica**, com possibilidade de escolher ou construir novos mapas, adicionar cruzamentos, redefinir estradas e criar topologias personalizadas.
- **Políticas adaptativas**, onde os semáforos ajustam automaticamente a duração do verde com base na carga observada em tempo real.
- **Expansão da análise**, integrando novas métricas ou mecanismos de previsão de congestionamento.