Progettazione sistema IoT Semaforo Intelligente

Luca Evangelisti, Alessandro Manucci, Gianluca Milani

Esame Laboratorio di Sistemi IoT

https://github.com/lucaeva03/Smart_Traffic_Lights

Indice

1. Introduzione (G)
2. Descrizione
    a. Funzionalità implementate (A)
    b. Architettura generale del sistema (G)
    c. Architettura layer del sistema (G)
    d. Logica semaforo intelligente (A)
    e. Comunicazione dati (L)
    f. Database (L)
    g. WebApp (G)
    h. Avvio e spegnimento del sistema (L)
    i. Possibili implementazioni future (A)
3. Schema circuitale (A)
4. Flusso Logico (L)
5. Codice (L)
6. Percorso Progettuali
    a. Diario (G)
    b. Difficoltà riscontrate (A)

Introduzione

Questo documento illustra il progetto di un sistema IoT per la gestione intelligente di un incrocio semaforico. L'obiettivo principale è realizzare un prototipo funzionante che integri due schede Micro:bit programmate in MicroPython, moduli sensori/attuatori del kit, un database time-series e una moderna interfaccia web di monitoraggio e controllo.
Il sistema è articolato in tre parti:

1. Due Micro:
    - Micro:bit A gestisce sensori veicolari, logica semaforica adattiva (con modalità normale, notturna ed emergenza) e attua le luci dei semafori;
    - Micro:bit B funge da gateway wireless-seriale, inoltrando messaggi radio verso il PC e viceversa.
2. Uno script Python (microbit_to_influx.py) legge la seriale, valida i dati, li esporta in InfluxDB e inoltra i comandi di controllo.
3. Presentazione:
    - InfluxDB conserva tutte le serie temporali;
    - Grafana visualizza lo storico e i grafici dei dati;
    - WebApp Flask offre un dashboard interattivo con stato in tempo reale, grafici embedded e comandi utente.

Questa documentazione descrive in dettaglio l'architettura, la logica applicativa, la configurazione hardware/software, il funzionamento e le potenzialità di estensione futura.

Descrizione

Funzionalità implementate

1. Lettura dei Sensori di Presenza Veicolare

La funzione check_sensors() consente di acquisire i dati da sensori analogici (connessi a pin GPIO configurati come ingressi analogici) installati sulle tre vie dell'incrocio (Nord, Est, Sud). Il valore analogico acquisito viene confrontato con la soglia di attivazione definita dalla costante SOGLIA; se il valore supera tale soglia, viene rilevata la presenza di un veicolo. Il rilevamento di transizioni di stato (da assente a presente e viceversa) genera un evento trasmesso via radio.

2. Logica Semaforica Intelligente Dinamica

La funzione servi(idx) implementa la logica di assegnazione dinamica della precedenza. Ogni ciclo prevede:

- accensione verde per una via selezionata,
- mantenimento minimo GREEN_TIME_MIN,
- eventuale estensione fino a MAX_GREEN_TIME se il sensore resta attivo,
- passaggio a giallo per YELLOW_TIME,
- infine rosso e selezione della prossima via secondo politica round-robin.

La logica tiene conto della presenza simultanea di veicoli su più vie, garantendo una gestione equa e intelligente del traffico.

3. Comunicazione Radio Micro:bit A → Micro:bit B

Utilizzando radio.send() e radio.receive(), il Micro:bit A (controllore semafori) trasmette periodicamente messaggi nel formato:

- STATE;<via>;<colore>;<timestamp> per i cambi di colore,
- SENSOR;<via>;<flag>;<timestamp> per l'attivazione/disattivazione dei sensori.

Questi dati sono ricevuti dal Micro:bit B (logger), il quale li inoltra via UART verso il computer host.

4. Logging su InfluxDB 2.x via UART e Python

Lo script Python microbit_to_influx.py esegue:

- acquisizione dei messaggi via porta seriale,
- validazione tramite espressioni regolari,
- correzione automatica di errori (es. "VEDE" → "VERDE"),
- conversione nel formato InfluxDB line protocol,
- scrittura asincrona nel bucket microbit del database.

I dati vengono quindi storicizzati per consentire analisi temporali e visualizzazione.

5. Dashboard Interattive in Grafana

Le dashboard realizzate in Grafana includono:

- serie temporali dello stato dei semafori (rosso/giallo/verde),
- attivazioni binarie dei sensori per ciascuna via,
- visualizzazione in tempo reale con intervalli selezionabili (default: -5m).

6. Interfaccia Web Flask (WebApp)

L'applicazione web costruita con Flask include:

- rendering dinamico degli indicatori semaforici,
- rilevazione in tempo reale dello stato dei sensori,
- integrazione dashboard Grafana via iframe,
- pannello di controllo con pulsanti per attivare modalità speciali,
- sezione statistiche semaforiche.

L'interfaccia si aggiorna automaticamente ogni 1-5 secondi (a seconda della sezione) tramite chiamate AJAX asincrone.

7. Modalità Notturna (Night Mode)

Tramite il comando CMD;NIGHT;ON, il sistema disattiva la logica semaforica intelligente e imposta tutti i LED gialli dei semafori a lampeggio (on/off ogni 500ms), simulando la modalità notturna standard.

8. Modalità Emergenza (Uno alla Volta)

La webapp offre tre pulsanti (Emergenza Nord, Est, Sud) con comportamento mutuamente esclusivo:

- un solo pulsante attivo alla volta,
- il semaforo associato diventa verde forzatamente,
- tutti gli altri passano a rosso,
- la logica standard è sospesa finché l'emergenza è attiva.

9. Calcolo Statistiche in Tempo Reale

Ogni 5 secondi, la webapp aggiorna:

- numero totale di veicoli rilevati (conteggio degli eventi SENSOR;via;1;...),
- numero di cambi di stato per ciascun semaforo (conteggio di STATE;via;...),
- durata media delle fasi verdi (media dei tempi tra VERDE e GIALLO).

Questi dati sono estratti dinamicamente da InfluxDB tramite query Flux eseguite da Flask.

Architettura generale del sistema

Per garantire modularità, scalabilità e chiarezza nei ruoli, l'intero sistema è organizzato secondo un'architettura IoT a quattro livelli, che separa nettamente l'acquisizione dati, il loro trasporto, la memorizzazione/analisi e l'interazione con l'utente.

1. Livello dispositivo (Edge)

Componenti principali

- **Micro:bit A**
    - Sensori: tre sensori LDR collegati ai pin analogici (pin3, pin4, pin10) per rilevare il passaggio dei veicoli.
    - Attuatori: tre gruppi di LED (Rosso-Giallo-Verde) collegati ai pin digitali (pin0,1,2; pin8,12,13; pin14,15,16) che simulano i semafori.
    - Logica integrata:

        - Ciclo semaforico dinamico (rosso → verde → giallo → rosso) con tempi minimi, massimi ed estensione adattiva.
        - Monitoraggio in tempo reale dei sensori con invio di eventi SENSOR;Via;Flag;Timestamp.
        - Invio degli stati semaforici con STATE;Via;Colore;Timestamp.
        - Gestione locale di Modalità Notte e Emergenza (override prioritario via radio).

2. Livello gateway (Fog)

Componenti principali

- **Micro:bit B**
    - Funzione di **gateway radio ↔ seriale**:
        - Riceve via radio i messaggi dal Micro:bit A.
        - Li inoltra via UART (porta USB, es. COM5) al PC.
        - Riceve comandi seriali (CMD;…) dal PC e li ritrasmette via radio a Micro:bit A.
- **Script Python di ingest (microbit_to_influx.py)**
    - Aggregazione e filtraggio dei dati: parsing robusto con regex, correzione typo, smistamento tra messaggi STATE e SENSOR.
    - Validazione e sicurezza: scarta frammenti malformati, garantisce tipizzazione (line protocol) per InfluxDB.
    - Endpoint locale (Flask su porta 5001) per accettare comandi di Modalità Notte ed Emergenza e inoltrarli al Micro:bit A.

❑ Il Fog si occupa di pre-elaborare, validare e instradare i dati tra Edge e Cloud, aggiungendo un primo livello di sicurezza e resilienza.

3. Livello piattaforma (Cloud)

Componenti principali

- **InfluxDB 2.x**
    - Database time-series in esecuzione su localhost:8086.

    - Bucket microbit per dati semaforici (stato_semaforo) e sensori (sensore_attivazione).

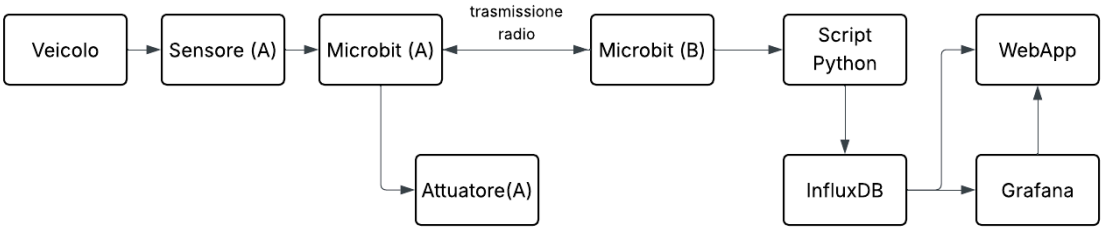
- **Grafana**
    - Dashboard per l'analisi visuale dei dati storici.
    - Pannelli embed in WebApp che interrogano InfluxDB con query Flux (now-5m → now).
    - Funzioni di alerting, reporting e gestione retention.

4. Livello applicativo

Componenti principali

- **WebApp Flask** (porta 5000)
    - API RESTful per:
        - /api/stati e /api/sensori: interrogazioni in tempo reale su InfluxDB.
        - /api/night_mode, /api/emergency: invio comandi al Fog.
        - /stats: lettura statistica aggregata (veicoli, cambi stati, durata verde).
    - **Interfaccia utente**:
        - Stato semafori/sensori live (aggiornamento ogni secondo).
        - Pulsanti Modalità Notte ed Emergenza (uno per via).
        - Sezione Statistiche (refresh ogni 5 s).
        - Grafici Grafana embedded (refresh 1 s).
- **Frontend**: HTML5, CSS3 (Grid/Flex responsive), JavaScript vanilla, icone Font-Awesome.

5. Flusso generale dei Dati



- **Veicolo → Sensore (A)**
  Il passaggio del veicolo viene rilevato da un sensore di pressione o prossimità collegato a Micro:bit A.

- **Micro:bit A – Nodo Edge**
  Gestisce localmente la logica semaforica e controlla gli attuatori (LED/semafori). Trasmette lo stato e gli eventi dei sensori via radio a Micro:bit B.

- **Micro:bit B – Logger**
  Riceve i dati via radio e li inoltra al PC tramite porta seriale. Funziona anche da ponte per i comandi ricevuti dal sistema centrale (es. notte/emergenza).

- **Script Python (microbit_to_influx.py)**
  Interpreta i dati seriali ricevuti da Micro:bit B, li normalizza(standardizzati e compatibili con InfluxDB) e li scrive nel database InfluxDB.

- **InfluxDB**
  Archivia i dati storici in formato serie temporali, utili per analisi, visualizzazione e monitoraggio.

- **Grafana**
  Legge i dati da InfluxDB e genera dashboard grafiche in tempo reale, integrate nella WebApp.

- **WebApp (Flask):**
  Interfaccia utente per:
  - Visualizzare stato semafori e sensori
  - Consultare grafici storici
  - Attivare comandi come modalità notte o emergenza

Architettura layer del sistema

Nel riquadro sottostante verrà presentata una suddivisione in layer

1. Physical Layer - Livello fisico con sensori e attuatori

Questo livello comprende i dispositivi fisici distribuiti sul campo:

- Due Micro:bit:
    - Micro:bit A: controlla i semafori e legge i sensori
    - Micro:bit B: riceve e inoltra i dati via UART
- Sensori analogici collegati a pin analogici (pin3, pin4, pin10) per il rilevamento veicoli sulle vie Nord, Est, Sud.
- Attuatori (LED colorati o moduli semaforici) connessi ai pin digitali per ciascun colore semaforico (Rosso, Giallo, Verde).
- Alimentazione tramite cavo USB.
- Pulsanti hardware (simulati o reali) per attivare le modalità emergenza da web.

Il Physical Layer è responsabile del rilevamento degli eventi fisici (presenza veicoli) e dell'attuazione delle fasi semaforiche tramite logica locale.

2. Connectivity Layer - Protocolli e gateway per trasmissione dati

- Protocolli:
    - radio.send / radio.receive (protocollo RF proprietario Micro:bit, gruppo 42, indirizzo 0x12345678)
    - UART seriale (tra Micro:bit B e PC/Server)
    - HTTP/REST per la comunicazione tra webapp e backend
- Gateway:
    - Micro:bit B funge da gateway UART ⇄ RF
    - Script Python microbit_to_influx.py sul PC riceve i dati UART e li inoltra a InfluxDB

Questo livello gestisce la trasmissione affidabile e asincrona dei dati da Micro:bit al sistema informativo (via radio e UART), agendo da ponte tra mondo fisico e digitale.

3. Data Layer - Storage e processing dei dati raccolti

- Storage:
    - InfluxDB v2 locale configurato con bucket "microbit"
    - Metriche persistite:
        - Stato semafori (ROSSO/GIALLO/VERDE)
        - Attivazione sensori (1/0)
- Processing:
    - Parsing dei messaggi seriali con regex e mapping semantico
    - Calcolo su richiesta di KPI:
        - Veicoli totali rilevati
        - Cambi di stato per via
        - Tempo medio in verde
- Aggregazioni eseguite via query InfluxQL/Flux o logica Python

Il Data Layer costituisce il cuore della conoscenza del sistema: conserva l'intero storico degli eventi e consente interrogazioni temporali e analisi quantitative delle performance.

4. Twin Layer - Modello virtuale con simulazione e analytics

- Modello digitale dell'incrocio rappresentato nella webapp:
    - Stato in tempo reale di ogni semaforo
    - Stato attivazione sensori
    - Modalità attiva (normale, notte, emergenza per via)
- Visualizzazione su Dashboard Grafana (quando attiva)
- Indicatori aggiornati ogni 5 secondi:
    - Veicoli rilevati
    - Tempo medio in verde

    - Numero cambi stato

La rappresentazione nel Twin Layer funge da digital twin dell'incrocio fisico, offrendo visione immediata dello stato corrente e capacità analitica sulla storia e sui trend del traffico.

5. Service Layer - API e interfacce per interazioni e visualizzazioni

- Backend in Flask (Python):
    - Avvia thread per lettura dati reali
    - Espone endpoint interni per invio comandi via UART
- Webapp responsive:
    - Visualizza semafori, pulsanti, stato sensori
    - Pulsanti per modalità notte ed emergenza
    - Sezione statistiche auto-aggiornata
- Interazione utente → backend → hardware

Questo layer fornisce gli strumenti di interfaccia, sia grafica che di controllo, per il monitoraggio e l'interazione col sistema. Rappresenta il punto di contatto uomo-macchina.

6- Business Layer - Integrazione con processi aziendali e decision making

Attualmente limitato, ma con grande potenziale:

- Le metriche raccolte possono guidare:
    - Ottimizzazione dei tempi semaforici (data-driven tuning)
    - Studio del flusso veicolare in contesti urbani
    - Supporto a decisioni su viabilità, infrastrutture o regolazioni intelligenti
- Estendibile verso:
    - Sistemi di smart city (integrazione con altri IoT: traffico, ambiente)
    - Sistemi predittivi (machine learning su pattern di congestione)
    - Notifiche ad enti comunali o dashboard decisionali

Il Business Layer connette il sistema con un ecosistema urbano più ampio, permettendo l’impiego dei dati raccolti per strategie di governance e pianificazione urbana sostenibile.

Logica semaforo intelligente

1. Inizializzazione e Stato Base

All'avvio del sistema, tutti i semafori vengono portati nello stato ROSSO tramite la funzione tutti_rossi(). Questa funzione imposta:

- LED Rosso acceso
- LED Giallo e Verde spenti
- Invio dello stato via radio (tramite send_state_event())

Viene anche azzerato l’indice di rotazione idx, che serve a scorrere ciclicamente tra le vie in assenza di richieste.

2. Controllo continuo dei sensori

La funzione check_sensors() legge continuamente lo stato analogico dei sensori. Se un valore analogico supera la soglia di attivazione definita dalla costante SOGLIA (200), viene considerato attivo.

3. Ciclo Principale e Selezione Priorità

Nel loop principale, vengono analizzati tutti i sensori:

```
vals = [v['sens'].read_analog() for v in vie]
active = [i for i, v in enumerate(vals) if v > SOGLIA]
```

- Se ci sono vie attive (con sensore premuto), il sistema cerca ciclicamente (a partire dall’ultima via servita idx) la prima via attiva da servire. Questo meccanismo implementa una forma semplice di "fairness" per evitare starvation di una via.
- Se nessuna via è attiva, allora viene servita a rotazione la prossima via (servi(idx)), simulando un funzionamento ciclico classico.

4. Servizio di una Via: Funzione servi(v_idx)

Questa è la funzione principale che gestisce i tempi semaforici. Ha tre fasi principali:

a. Accensione Verde

```
tutti_rossi()
v['R'].write_digital(0)
v['G'].write_digital(1)
```

- Tutti gli altri semafori vengono posti a rosso.
- Il semaforo selezionato diventa verde.

b. Durata Verde Dinamica

Il verde dura un minimo di GREEN_TIME_MIN ms. Dopo questo periodo si attiva una finestra di estensione:

```
if altre and now_ext >= MAX_GREEN_TIME:
    break
if release_start is not None and (running_time() - release_start) >= RELEASE_GRACE:
    break
```

Il verde si mantiene attivo se:

- Il sensore rimane premuto
- Non ci sono richieste da altre vie
- Non è trascorso il tempo massimo (MAX_GREEN_TIME)
- Il sensore non è stato rilasciato da più di RELEASE_GRACE ms

Questa parte è estremamente intelligente: consente di prolungare il verde per una via finché necessario, ma evitando congestione e garantendo alternanza.

Il MAX_GREEN_TIME ci è utile poiché funge da valore di riferimento per il cambio di stato.

Es. se un sensore è pigiato da 20 secondi e improvvisamente un altro sensore viene premuto, avverrà il cambio di stato del semaforo corrente, da verde a giallo e poi rosso, mentre il semaforo il cui sensore è stato ora premuto passerà da rosso a verde. Tutto ciò è gestito dal MAX_GREEN_TIME a cui è stato assegnato un valore di 10 secondi. Dunque passati questi 10 secondi, anche se il sensore del semaforo corrente è ancora soggetto a pressione, se arriva una pressione da un altro sensore di un altro semaforo, avverrà il cambio di stato.

c. Giallo e Transizione

Terminata la fase verde, il semaforo passa a GIALLO per YELLOW_TIME ms, poi ROSSO.

5. Casistica: Nord premuto e altri due sensori premuti

Nel funzionamento del sistema semaforico intelligente, è stato implementato un meccanismo di priorità dinamica basato su un ciclo rotante nell'ordine fisso Nord → Est → Sud. Un caso particolare di questo comportamento si verifica quando il semaforo Nord è attivo (cioè il suo sensore è premuto) e, durante la sua fase verde, vengono premuti anche i sensori delle vie Sud ed Est. In tale circostanza, terminato il verde di Nord, il sistema controllerà i sensori partendo dalla via successiva nell'ordine ciclico, cioè Est. Di conseguenza, anche se il sensore Sud è stato premuto prima di quello Est, sarà comunque Est a essere servito per prima. Solo successivamente, se ancora attivo, sarà servita la via Sud.

Questa scelta progettuale è stata adottata deliberatamente per garantire un ordine prevedibile e regolare nella gestione dell'incrocio. Un sistema che risponde semplicemente in ordine di pressione dei sensori (FIFO) potrebbe causare confusione, specialmente in condizioni di traffico intenso o con input simultanei. L'ordine ciclico fornisce invece una logica coerente e facilmente interpretabile sia per gli sviluppatori che per gli utenti, evitando decisioni imprevedibili e potenzialmente caotiche.

Comunicazione dati

Il sistema realizzato implementa una comunicazione dati **distribuita** tra dispositivi e livelli software eterogenei. Il flusso dei dati si snoda in quattro segmenti principali:

1. Comunicazione tra Micro:bit A e Micro:bit B (via Radio)

Il Micro:bit A gestisce il controllo fisico dell'incrocio: legge i sensori analogici, regola i semafori e invia aggiornamenti di stato.
Il Micro:bit B agisce da gateway radio-seriale, inoltrando tutti i messaggi ricevuti dal Micro:bit A al PC tramite porta seriale USB.

Dettagli tecnici:

- Tecnologia: Radio interna Micro:bit (2.4 GHz)
- Configurazione comune:
    - channel=3, power=7, group=42, address=0x12345678
- Formati messaggi trasmessi:
    - Stato semaforo: STATE;via;colore;timestamp
    - Stato sensore: SENSOR;via;flag;timestamp

2. Comunicazione tra Micro:bit B e PC (via Porta Seriale)

Il collegamento seriale USB tra Micro:bit B e computer (attraverso la porta seriale COM5) consente di:

- Ricevere dati telemetrici da Micro:bit A

- Inviare comandi dall'interfaccia web al sistema:
    - Comandi: CMD;NIGHT;ON, CMD;EMERGENZA;[via]

Lo script coinvolto in questa operazione è microbit_to_influx.py che legge messaggi seriali in tempo reale ed esegue parsing, validazione e salvataggio su database.
Tutti i messaggi ricevuti sono validati tramite espressioni regolari e registrati nel database InfluxDB con timestamp nativo.

3. Comunicazione tra Backend Web e Micro:bit (tramite seriale)

La web app Flask permette all'utente di inviare comandi remoti tramite i pulsanti:

- Attivazione modalità notte

- Gestione emergenze per singole vie

Il backend comunica con il Micro:bit B scrivendo sulla seriale comandi del tipo:

- CMD;NIGHT;ON

- CMD;EMERGENZA;SUD

Questi comandi sono trasmessi via radio dal Micro:bit B al Micro:bit A, che attiva la logica richiesta

4. Comunicazione tra WebApp, InfluxDB e Grafana

Dopo che i dati vengono raccolti e salvati in **InfluxDB**, essi sono resi accessibili tramite:

- Grafana: visualizzazione di grafici in tempo reale

- API Flask: endpoints RESTful per interrogare lo stato corrente dei semafori e sensori

Esempi di endpoint API:
GET /api/stati?via=Nord&range=-5m
GET /api/sensori?via=Sud&range=-10m
GET /api/statistiche

Questi endpoint vengono interrogati ciclicamente dalla web app ogni secondo per mostrare:

- Stato attuale dei semafori (colore)
- Stato attuale dei sensori (attivo/non attivo)
- Statistiche cumulative (es. veicoli rilevati, durata media verde, ecc.) (ogni 5 secondi)

5. Robustezza e Validazione

- Eventuali errori o frammenti corrotti vengono ignorati e segnalati

- Tipi di dato garantiti nel protocollo Line di InfluxDB (int, tag, timestamp)

- Lo stato del sistema può essere monitorato in tempo reale e non richiede refresh manuale

Database

L'intero sistema utilizza InfluxDB 2.x come database time-series per la raccolta, memorizzazione e analisi dei dati provenienti dai sensori e dagli attuatori del sistema semaforico intelligente. InfluxDB è stato scelto per la sua efficienza nella gestione di dati temporali in tempo reale, perfettamente in linea con i requisiti di monitoraggio e aggiornamento continuo del progetto.

1. Caratteristiche della Configurazione

- Tipo di database: Time-series database (TSDB)
- Motore utilizzato: InfluxDB 2.x
- Host: localhost:8086
- Bucket: microbit
- Organizzazione: microbit-org
- Token di accesso: (privato, configurato nello script Python)

InfluxDB è eseguito localmente su Windows tramite eseguibile influxd.exe avviato tramite PowerShell

2. Flusso di Scrittura dei Dati

La scrittura dei dati in InfluxDB avviene tramite lo script Python microbit_to_influx.py, che riceve messaggi via seriale dal Micro:bit B e li traduce nel line protocol di InfluxDB, garantendo integrità e tipizzazione corretta.

Messaggi validi:

- STATE;via;colore;timestamp
- SENSOR;via;0/1;timestamp

Ogni record contiene:

- Measurement: stato_semaforo o sensore_attivazione
- Tag: via (Nord, Est, Sud)
- Field:
    - stato (viene convertito per facilitare la rappresentazione grafica ad intero:

0=Rosso, 1=Giallo, 2=Verde)
    - active (booleano: 0/1)

- Timestamp: esplicito

3. Flusso di Lettura dei Dati

Il database è interrogato in due modi principali:

1. WebApp Flask

- Endpoint API /api/stati, /api/sensori, /api/statistiche usano InfluxDB Python SDK per eseguire query in linguaggio Flux.
- I dati sono trasformati in formato JSON e serviti al frontend per:
    - Mostrare in tempo reale lo stato di ogni semaforo e sensore
    - Calcolare statistiche (cambi stato, tempo medio verde, numero veicoli)

2. Grafana

- Due pannelli nella dashboard Grafana interrogano direttamente il bucket microbit
- Refresh: ogni 5 secondi
- Visualizzazioni:
    - Colore semafori nel tempo (line chart)
    - Attivazioni sensori nel tempo (bar chart)

4. Statistiche Calcolate

Dal database vengono calcolate automaticamente, ogni 5 secondi, le seguenti metriche per ciascuna via:

- Veicoli rilevati (conteggio di eventi SENSOR=1)
- Cambi di stato (conteggio delle transizioni stato)
- Durata media del verde (intervallo tra due eventi VERDE→GIALLO)

Questi dati vengono mostrati nell'interfaccia utente in una sezione dedicata

5. Sicurezza e integrità

- I dati sono resistenti a crash: il database mantiene integrità anche in caso di interruzioni

- Lo script di scrittura esegue validazioni sintattiche dei messaggi e scarta quelli malformati

- I campi sono correttamente tipizzati per evitare errori di analisi

WebApp

La WebApp è il cuore dell'interfaccia utente del sistema IoT semaforico: consente di monitorare in tempo reale, interagire con il backend hardware, visualizzare statistiche e integrare i grafici generati da Grafana. È stata realizzata in Python con il framework Flask, e utilizza HTML, CSS e JavaScript per il frontend.

1. Architettura

- Framework: Flask

- Frontend: HTML5 + CSS3, JavaScript

- Backend: Python (dentro ambiente virtuale venv)

- Database collegato: InfluxDB (tramite influxdb-client)

- Grafici: Grafana embedded (iframe)

- Porta di ascolto: http://127.0.0.1:5000

2. Funzionalità Implementate

Monitoraggio Real-Time

- Stato attuale dei tre semafori (colore: Rosso, Giallo, Verde)

- Stato dei sensori (pressione rilevata: SÌ/NO)

- Grafici temporali incorporati (aggiornati ogni 5s via Grafana)

- Statistiche auto-aggiornanti ogni 5s:

    - Veicoli rilevati

    - Numero cambi di stato

    - Tempo medio di verde per ogni via

Controlli interattivi

- Modalità Notte:

    - Disattiva i sensori
    - Tutti i semafori lampeggiano giallo (effetto gestito via radio)
- Emergenze (uno alla volta):
    - Pulsanti “Emergenza Nord / Est / Sud”
    - Forzano immediatamente un semaforo su verde (gli altri diventano rossi)
    - Disattiva i sensori

Grafana integrato

- Visualizzazione di:
    - Stato semafori nel tempo (line chart)
    - Attivazione sensori (bar chart)
- Aggiornamento automatico ogni 5 secondi

3. Comunicazione e API

La WebApp espone diverse API RESTful (chiamate da JavaScript):

Endpoint	Metodo	Funzione
`/api/stati?via=...`	GET	Stato semaforo per direzione
`/api/sensori?via=...`	GET	Stato sensore per direzione
`/api/statistiche`	GET	Statistiche veicoli, cambi, media verdi
`/modalita-notte`	POST	Attiva/disattiva modalità notte
`/emergenza/<via>`	POST	Attiva/disattiva emergenza via specifica

Tutte le modifiche allo stato vengono inviate dal Microbit B al Microbit A via radio, mantenendo il controllo degli attuatori sul lato trasmettitore.

4. Struttura del Layout

L’interfaccia è organizzata in tre sezioni verticali principali:

1. **Stato Attuale Semafori e Sensori**
    - Tre box colorati per ciascun semaforo (giallo, rosso, verde dinamici)

    - Stato attuale dei sensori (SÌ/NO/-)

2. **Controlli Modalità e Emergenze**
    - Pulsante “Modalità Notte” (toggle)
    - Tre pulsanti “Emergenza ” (solo uno attivo per volta)
3. **Statistiche e Grafici**
    - Due pannelli Grafana, uno per lo stato dei semafori e uno per i sensori (grafici embedded - iframe)

5. Design e Usabilità

- Stile moderno ispirato a dashboard professionali
- Utilizzo di colori coerenti per semafori e indicatori
- Animazioni fluide per cambi stato
- Icone, pulsanti e sezioni ben distanziate
- Feedback visivi immediati (es. cambio colore, lampeggio)

Avvio e spegnimento del sistema

1. Avvio

1. Assicurarsi che il servizio Grafana sia stato automaticamente avviato altrimenti avviarlo tramite PowerShell con “net start grafana”, sarà quindi in ascolto su localhost:3000
2. Avviare il database influx attraverso la PowerShell entrando nella cartella dove è presente influxd.exe e avviandolo. InfluxDB sarà quindi in ascolto su localhost:8086.
3. Collegare il Micro:bit gateway con il cavo USB e flashare di nuovo lo script tramite l’editor
4. Collegare il Micro:bit del semaforo alla batteria
5. Aprire un terminale nella cartella dove è presente il file microbit_to_influx.py e lanciarlo

6. Entrare dentro la venv contenente la webapp in Flask (se necessario disabilitare sistema di sicurezza) e lanciare app.py che sarà disponibile sul browser all'indirizzo http://127.0.0.1:5000

2. Spegnimento

1. Chiudere l'esecuzione di app.py da terminale
2. Chiudere l'esecuzione di microbit_to_influx.py da terminale
3. Scollegare Micro:bit gateway
4. Spegnere Micro:bit semaforo
5. Chiudere l'esecuzione di influxd.exe
6. Se necessario chiudere anche grafana attraverso "net stop grafana"

Possibili implementazioni future

1. Rilevamento Multisensore per Stima del Traffico

L'attuale configurazione prevede un singolo sensore per ciascuna via. Un'estensione naturale consiste nell'aggiunta di sensori in serie lungo ogni corsia d'accesso all'incrocio. Ciò consentirebbe la stima approssimativa della lunghezza della fila di veicoli, abilitando meccanismi di prioritizzazione più raffinati e proporzionali alla congestione reale. Tali dati potrebbero inoltre essere utilizzati per modellare e prevedere il flusso veicolare medio per intervallo temporale.

2. Integrazione con Sistemi Smart City

L'intero sistema potrebbe essere trasformato in nodo di una rete urbana interconnessa. Attraverso protocolli MQTT o LoRaWAN, il semaforo smart invierebbe periodicamente i dati a una piattaforma centrale di gestione urbana, facilitando il controllo aggregato e la pianificazione del traffico cittadino.

3. Dashboard Avanzata con Storico Interattivo

Estensione della webapp per includere una sezione storica consultabile, in cui siano visibili in forma tabellare o grafica i dati raccolti: numero di veicoli per fascia oraria, durata media delle fasi semaforiche, frequenza delle emergenze, ecc.

4. Notifiche Automatiche di Anomalia

L'integrazione con servizi di messaggistica (es. Telegram Bot, Email SMTP) abiliterebbe l'invio di notifiche automatiche in caso di eventi critici: sensori offline, guasti radio,

congestioni anomale, durata eccessiva in emergenza.

Schema circuitale

1. Schema circuitale

Direzione	Componente	Signal (S)	VCC (V)	GND
Nord	Sensore Press.	P3	P5	distribuzione non rilevante
	LED Rosso	P0		
	LED Giallo	P1		
	LED Verde	P2		
Est	Sensore Press.	P4	P2	
	LED Rosso	P8		
	LED Giallo	P12		
	LED Verde	P13		
Sud	Sensore Press.	P10	P11	
	LED Rosso	P14		
	LED Giallo	P15		
	LED Verde	P16		

2. Motivazioni Tecniche della Scelta dei Pin Signal

La selezione dei pin è il risultato di una progettazione attenta, vincolata dalle seguenti considerazioni hardware:

a. I sensori di pressione richiedono pin analogici

I sensori del kit restituiscono valori analogici che devono essere letti tramite l'ADC (Analog-to-Digital Converter) del micro:bit.

Sul micro:bit v2, solo i pin P0, P1, P2, P3, P4 e P10 sono abilitati alla lettura analogica.

Per questo motivo, abbiamo assegnato:

P3 → Sensore Nord

P4 → Sensore Est

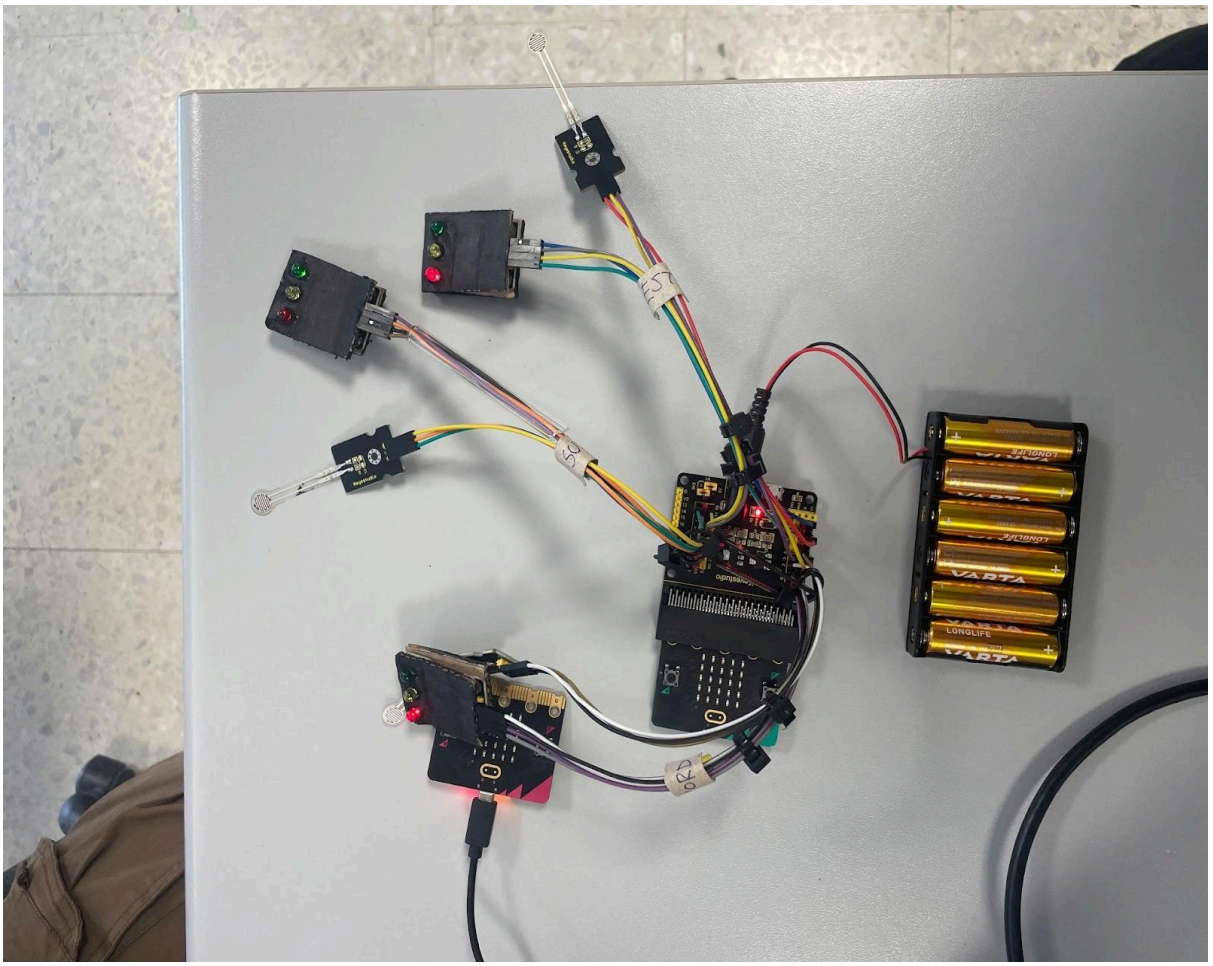
P10 → Sensore Sud
Questa scelta evita errori critici (come assegnare sensori analogici a pin digital-only come P8 o P13) e garantisce una lettura precisa del valore.

b. I LED possono essere collegati a qualunque pin digitale

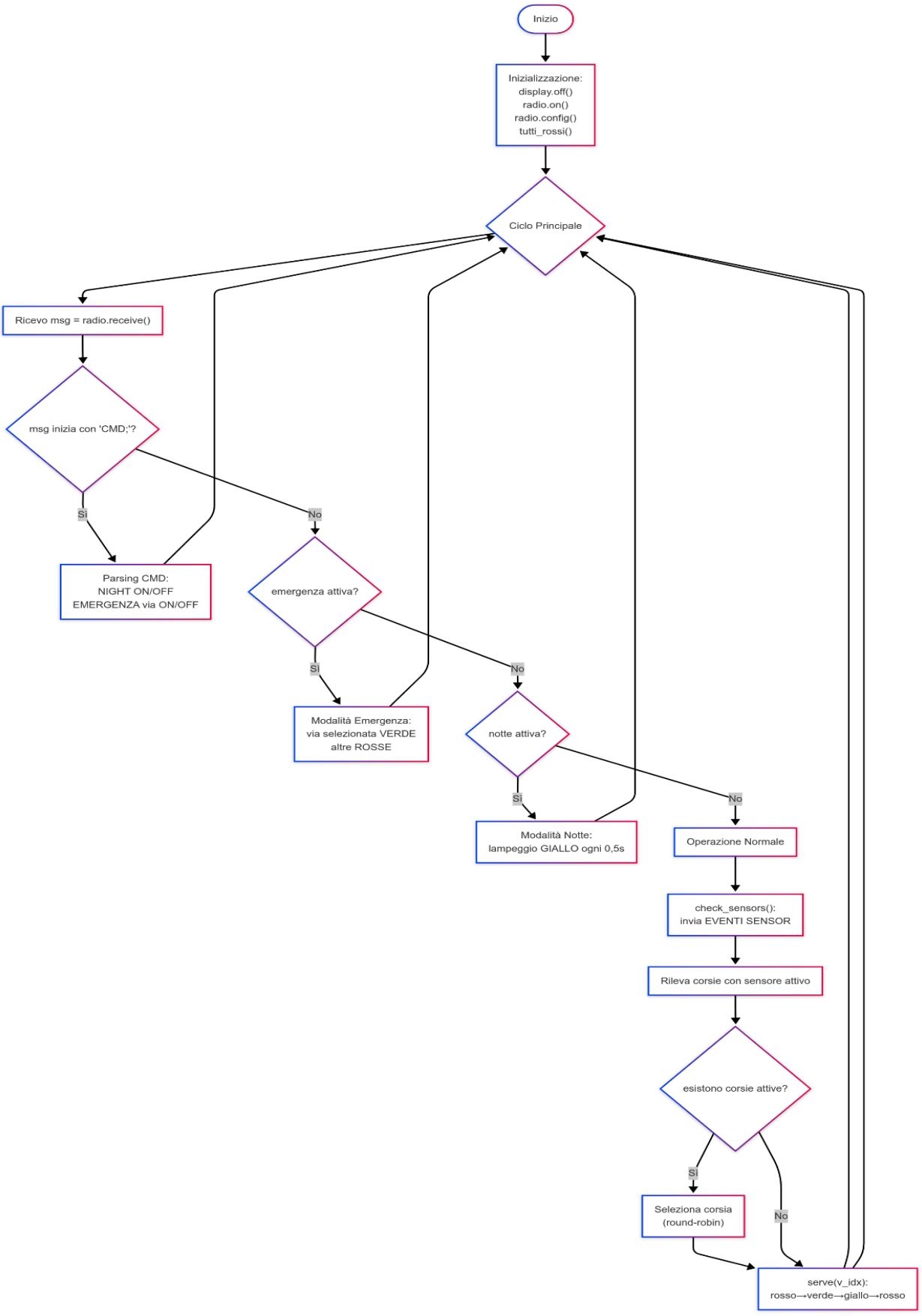
I LED non richiedono lettura analogica, quindi possono essere collegati a pin digitali qualsiasi, purché non condivisi con i sensori.

Sono stati scelti pin come P0, P1, P2 (Nord), P8, P12, P13 (Est), e P14, P15, P16 (Sud) che sono perfettamente adatti a controllare LED.

P0–P2 sono anche analogici, ma non è un problema usarli per i LED, poiché non avevamo bisogno di usarli per ingressi analogici.



Flusso Logico



Codice

1. Trasmettitore: Semaforo, Micro:bit A

```
from microbit import *
import radio

# ── SEZIONE 1: CONFIGURAZIONE INIZIALE
──────────────────────────────────────
# Accende il modulo radio e ne imposta i parametri (canale,
potenza, lunghezza pacchetto, gruppo, ecc.)
radio.on()
radio.config(
    channel=3,
    power=7,
    length=32,
    queue=3,
    address=0x12345678,
    group=42
)

# ── SEZIONE 2: COSTANTI DI FUNZIONAMENTO
──────────────────────────────────
# Soglia di lettura del sensore (analogico)
SOGLIA          = 200
# Tempi in millisecondi: verde minimo, verde massimo, giallo,
grace period
GREEN_TIME_MIN  = 5000
MAX_GREEN_TIME  = 10000
YELLOW_TIME     = 2000
RELEASE_GRACE   = 3000   # intervallo di “grace” dopo il rilascio
del sensore

# Definizione delle tre corsie (Nord, Est, Sud) con sensore + pin
LED R/Y/G
vie = [
    { 'nome':'Nord', 'sens': pin3,  'R': pin0,  'Y': pin1,  'G':
pin2  },
    { 'nome':'Est',  'sens': pin4,  'R': pin8,  'Y': pin12, 'G':
pin13 },
    { 'nome':'Sud',  'sens': pin10, 'R': pin14, 'Y': pin15, 'G':
pin16 }
]

# Stato corrente dei semafori: 0=verde, 1=giallo, 2=rosso
stati_correnti   = [2, 2, 2]
```

```
# Stato precedente dei sensori (per inviare eventi solo al
cambiamento)
prev_sens_active = [False] * 3

# ── SEZIONE 3: VARIABILI MODALITÀ SPECIALI
──────────────────────────────────
# Night mode
night_mode     = False
blink_state    = False
last_blink     = running_time()
BLINK_INTERVAL = 500   # ms per il lampeggio giallo

# Emergency mode
emergency_via = None   # None o stringa 'Nord'/'Est'/'Sud'

# ── SEZIONE 4: FUNZIONI DI INVIO EVENTI
──────────────────────────────────────
def send(msg):
    """Invia un messaggio via radio e fa un print di debug."""
    radio.send(msg)
    print(">>TX>>", msg)
    sleep(50)  # piccolo ritardo per evitare collisioni RF

def send_state_event(i, stato):
    """Invia un evento STATE per la corsia i con timestamp."""
    ts  = running_time()
    msg = "STATE;%s;%s;%d" % (vie[i]['nome'], stato, ts)
    send(msg)

def send_sensor_event(i, active):
    """Invia un evento SENSOR per la corsia i (1=attivo, 0=non
attivo)."""
    ts   = running_time()
    flag = 1 if active else 0
    msg  = "SENSOR;%s;%d;%d" % (vie[i]['nome'], flag, ts)
    send(msg)

# ── SEZIONE 5: LETTURA E GESTIONE SENSORI
─────────────────────────────────────
def check_sensors():
    """
    Legge tutti i sensori analogici.
    Se lo stato cambia rispetto al precedente, invia un evento
SENSOR.
    """
    for i, v in enumerate(vie):
        active = v['sens'].read_analog() > SOGLIA
```

```
        if active != prev_sens_active[i]:
            send_sensor_event(i, active)
            prev_sens_active[i] = active

# ── SEZIONE 6: METODO DI SICUREZZA (TUTTI ROSSI) ──────────────────────────
def tutti_rossi():
    """
    Imposta tutti i semafori su ROSSO e invia un evento STATE;...;ROSSO
    solo se il loro stato non era già rosso.
    """
    for i, v in enumerate(vie):
        v['R'].write_digital(1)
        v['Y'].write_digital(0)
        v['G'].write_digital(0)
        if stati_correnti[i] != 2:
            stati_correnti[i] = 2
            send_state_event(i, "ROSSO")

# ── SEZIONE 7: SUBROUTINE PRINCIPALE (SERVI UNA CORSIA) ──────────────────
def servi(idx):
    """
    Gestisce il ciclo di una singola corsia:
    1) tutti rossi → verde
    2) verde MIN con check sensori
    3) estensione con grace period o max time
    4) giallo
    5) rosso
    Invia gli eventi STATE ad ogni transizione.
    """
    v = vie[idx]

    # → 1) Passa a VERDE
    tutti_rossi()
    v['R'].write_digital(0)
    v['G'].write_digital(1)
    if stati_correnti[idx] != 0:
        stati_correnti[idx] = 0
        send_state_event(idx, "VERDE")

    # → 2) Verde minimo, leggendo i sensori ogni 100 ms
    start = running_time()
    while running_time() - start < GREEN_TIME_MIN:
        check_sensors()
        sleep(100)
```

```
    # → 3) Estensione verde con grace period
    ext_start     = running_time()
    release_start = None
    while True:
        check_sensors()
        now_ext = running_time() - ext_start

        attivo = v['sens'].read_analog() > SOGLIA
        if attivo:
            release_start = None
        else:
            if release_start is None:
                release_start = running_time()

        # verifica se c'è richiesta su altre vie
        altre = any(w['sens'].read_analog() > SOGLIA
                    for j, w in enumerate(vie) if j != idx)

        # Uscita: superato MAX_GREEN_TIME + altre richieste
        if altre and now_ext >= MAX_GREEN_TIME:
            break
        # Uscita: superato grace period dopo rilascio
        if release_start is not None and (running_time() -
release_start) >= RELEASE_GRACE:
            break

        sleep(100)

    # → 4) Passa a GIALLO
    v['G'].write_digital(0)
    v['Y'].write_digital(1)
    if stati_correnti[idx] != 1:
        stati_correnti[idx] = 1
        send_state_event(idx, "GIALLO")
    ystart = running_time()
    while running_time() - ystart < YELLOW_TIME:
        check_sensors()
        sleep(100)

    # → 5) Torna a ROSSO
    v['Y'].write_digital(0)
    v['R'].write_digital(1)
    if stati_correnti[idx] != 2:
        stati_correnti[idx] = 2
        send_state_event(idx, "ROSSO")
```

```
# ── SEZIONE 8: LOOP PRINCIPALE ──────────────────────────────
# Inizializzazione display e stato iniziale
display.off()
tutti_rossi()
idx = 0
n   = len(vie)

while True:
    # 1) Ricezione di eventuali comandi radio (NIGHT / EMERGENCY)
    cmd = radio.receive()
    if cmd:
        parts = cmd.split(';')
        if parts[0] == "CMD":
            # --- Modalità Notte
            if parts[1] == "NIGHT" and len(parts) == 3:
                night_mode = (parts[2] == "ON")
                if night_mode:
                    emergency_via = None  # disattiva emergenza
            # --- Modalità Emergenza
            elif parts[1] == "EMERGENCY" and len(parts) == 4:
                via, m = parts[2], parts[3]
                if m == "ON":
                    emergency_via = via
                    night_mode = False   # disattiva notte
                elif emergency_via == via:
                    emergency_via = None

    # 2) Emergency Mode (override)
    if emergency_via:
        for v in vie:
            if v['nome'] == emergency_via:
                # corsia in emergenza: solo verde
                v['R'].write_digital(0)
                v['Y'].write_digital(0)
                v['G'].write_digital(1)
                if stati_correnti[vie.index(v)] != 0:
                    stati_correnti[vie.index(v)] = 0
                    send_state_event(vie.index(v), "VERDE")
            else:
                # tutte le altre corsie: rosso
                v['R'].write_digital(1)
                v['Y'].write_digital(0)
                v['G'].write_digital(0)
                if stati_correnti[vie.index(v)] != 2:
                    stati_correnti[vie.index(v)] = 2
                    send_state_event(vie.index(v), "ROSSO")
```

```
        continue

    # 3) Night Mode (lampeggio giallo)
    if night_mode:
        if running_time() - last_blink >= BLINK_INTERVAL:
            blink_state = not blink_state
            for v in vie:
                v['R'].write_digital(0)
                v['G'].write_digital(0)
                v['Y'].write_digital(1 if blink_state else 0)
            last_blink = running_time()
        continue

    # 4) Logica normale: sensori + round-robin
    check_sensors()
    vals   = [v['sens'].read_analog() for v in vie]
    active = [i for i, val in enumerate(vals) if val > SOGLIA]

    if active:
        # se ci sono corsie attive, scegli la prima disponibile
        for off in range(n):
            i = (idx + off) % n
            if i in active:
                servi(i)
                idx = (i + 1) % n
                break
    else:
        # nessuna corsia attiva: serve quella corrente
        servi(idx)
        idx = (idx + 1) % n

    sleep(100)
```

2. Ricevitore: Logger, Micro:bit B

```
from microbit import *
import radio
from microbit import uart

# ——— Configurazione radio ———
radio.on()
radio.config(
    channel=3,
    power=7,
```

```
    length=32,
    queue=3,
    address=0x12345678,
    group=42
)

# ——— Configurazione UART ———
uart.init(baudrate=115200)

# svuoto eventuali pacchetti
while radio.receive() is not None:
    pass

print("Gateway UART⇄RF in ascolto...")

while True:
    # comandi PC → UART → RF
    if uart.any():
        line = uart.readline().strip()
        if line and line.startswith("CMD;"):
            radio.send(line)
            print(">>CMD TX>>", line)
    # dati RF → UART → PC
    msg = radio.receive()
    if msg:
        uart.write(msg + "\n")
    sleep(50)
```

3. Script Python: microbit_to_influx.py

```
#!/usr/bin/env python3
import threading, time, re, serial
from flask import Flask, request, jsonify
from influxdb_client import InfluxDBClient, Point

# — Config InfluxDB —
INFLUX_URL = "http://localhost:8086"
TOKEN      =
"s5SFeety1Oicla2NKF_bjgUHIybWAkUNKJiNw6SS8-qbgtJo5kWgfBJ7K26X1eQnf
1smfcLMqnHT5ZlwxXVXiA=="
ORG        = "microbit-org"
BUCKET     = "microbit"

# — Config Seriale —
```

```
COM_PORT  = "COM5"
BAUD_RATE = 115200

# — Pattern & Mapping —
MSG_PATTERN   =
re.compile(r"(STATE;[^;]+;[^;]+;\d+|SENSOR;[^;]+;[01];\d+)")
STATE_MAPPING = {'VERDE':2,'GIALLO':1,'ROSSO':0}

# — Statistiche —
vias = ["Nord","Est","Sud"]
stats = {
    v: {
      'sensor_hits':0,
      'state_changes':0,
      'green_sum':0,
      'green_count':0,
      'green_start':None
    } for v in vias
}

# Influx client
influx = InfluxDBClient(url=INFLUX_URL, token=TOKEN, org=ORG)
write_api = influx.write_api()

# Apri seriale
ser = serial.Serial(COM_PORT, BAUD_RATE, timeout=0)
print(f"[ingest] Serial open {COM_PORT}")

def ingest_loop():
    buffer = ""
    while True:
        to_read = ser.in_waiting or 1
        data    =
ser.read(to_read).decode('utf-8',errors='ignore')
        buffer += data

        while '\n' in buffer:
            line, buffer = buffer.split('\n',1)
            raw = line.strip()
            if not raw: continue

            msgs = MSG_PATTERN.findall(raw)
            for m in msgs:
                _, via, val, ts_dev = m.split(';')
                ts_dev = int(ts_dev)
                # STATISTICS
                if m.startswith("SENSOR;"):
```

```
                    if val=="1":
                        stats[via]['sensor_hits'] += 1
                else:  # STATE
                    stats[via]['state_changes'] += 1
                    # green timing
                    prev_start = stats[via]['green_start']
                    if val=="VERDE":
                        stats[via]['green_start'] = ts_dev
                    else:
                        if prev_start is not None:
                            duration = ts_dev - prev_start
                            stats[via]['green_sum']   += duration
                            stats[via]['green_count'] += 1
                            stats[via]['green_start'] = None

                # Write to InfluxDB as before
                if m.startswith("STATE;"):
                    num = STATE_MAPPING.get(val.upper())
                    if num is not None:
                        lp = f"stato_semaforo,via={via}
stato={num}i"

write_api.write(bucket=BUCKET,org=ORG,record=lp)
                else:
                    flag = int(val)
                    p =
Point("sensore_attivazione").tag("via",via).field("active",flag)

write_api.write(bucket=BUCKET,org=ORG,record=p)

            time.sleep(0.01)
        time.sleep(0.1)

# Flask per API night, emergency e stats
app = Flask(__name__)

def send_cmd(cmd):
    ser.write((cmd+"\n").encode())
    print(f"[cmd] {cmd}")

@app.route("/night_mode", methods=["POST"])
def night_mode():
    mode = request.json.get("mode","OFF").upper()
    if mode not in ("ON","OFF"):
        return jsonify(error="mode must be ON or OFF"),400
    send_cmd(f"CMD;NIGHT;{mode}")
    return jsonify(status="ok", mode=mode)
```

```
@app.route("/emergency", methods=["POST"])
def emergency():
    via  = request.json.get("via")
    mode = request.json.get("mode","OFF").upper()
    if via not in vias or mode not in ("ON","OFF"):
        return jsonify(error="via must be Nord/Est/Sud and mode
ON/OFF"),400
    send_cmd(f"CMD;EMERGENCY;{via};{mode}")
    return jsonify(status="ok", via=via, mode=mode)

@app.route("/stats")
def get_stats():
    out = {}
    for v in vias:
        s = stats[v]
        avg = (s['green_sum']/s['green_count']) if
s['green_count']>0 else 0
        out[v] = {
          'vehicles':       s['sensor_hits'],
          'changes':        s['state_changes'],
          'avg_green_ms':   int(avg)
        }
    return jsonify(out)

if __name__=="__main__":
    threading.Thread(target=ingest_loop, daemon=True).start()
    app.run(port=5001)
```

4. Back-End WebApp: app.py

```
from flask import Flask, render_template, request, jsonify
import requests
from influxdb_client import InfluxDBClient

# ── SEZIONE 1: CONFIGURAZIONE GENERALE
──────────────────────────────────
# URL e token per connettersi a InfluxDB
INFLUX_URL       = "http://localhost:8086"
TOKEN            = "s5SF…XiA=="               # token di
scrittura/query
ORG              = "microbit-org"
BUCKET           = "microbit"
```

```
# URL e identificatori per Grafana
GRAFANA_URL      = "http://localhost:3000"
DASHBOARD_UID    = "075b9381-2df1-4d19-8b33-ad212d2f2531"
DASHBOARD_SLUG   = "semaforo-intelligente"
PANEL_ID_STATO   = 1
PANEL_ID_SENSORE = 2

# ── SEZIONE 2: CLIENT INFLUXDB ──────────────────────────────
# Inizializza il client InfluxDB e l'API per le query
influx_client = InfluxDBClient(url=INFLUX_URL, token=TOKEN,
org=ORG)
query_api     = influx_client.query_api()

# ── SEZIONE 3: SETUP FLASK APP ──────────────────────────────
app = Flask(__name__)

# ── SEZIONE 4: HOME ROUTE ───────────────────────────────────
@app.route("/")
def index():
    # Ritorna la pagina HTML principale, passando le variabili per
l'embed di Grafana
    return render_template("index.html",
        grafana_url      = GRAFANA_URL,
        dashboard_uid    = DASHBOARD_UID,
        dashboard_slug   = DASHBOARD_SLUG,
        panel_id_stato   = PANEL_ID_STATO,
        panel_id_sensore = PANEL_ID_SENSORE
    )

# ── SEZIONE 5: API PER STATI SEMAFORI ──────────────────────
@app.route("/api/stati")
def api_stati():
    # Parametri query: via (Nord/Est/Sud) e range temporale (es.
-5m)
    via      = request.args.get("via", "Nord")
    duration = request.args.get("range", "-5m")
    # Costruzione della query Flux per InfluxDB
    flux = f"""
    from(bucket:"{BUCKET}")
      |> range(start:{duration})
      |> filter(fn:(r)=> r._measurement=="stato_semaforo")
      |> filter(fn:(r)=> r.via=="{via}")
    """
```

```
    # Esecuzione della query e raccolta dei risultati
    tables = query_api.query(flux)
    data = []
    for table in tables:
        for rec in table.records:
            data.append({
                "time":  rec.get_time().isoformat(),
                "value": rec.get_value()
            })
    return jsonify(data)

# ── SEZIONE 6: API PER STATI SENSORI
──────────────────────────────────────
@app.route("/api/sensori")
def api_sensori():
    via      = request.args.get("via", "Nord")
    duration = request.args.get("range", "-5m")
    flux = f"""
    from(bucket:"{BUCKET}")
      |> range(start:{duration})
      |> filter(fn:(r)=> r._measurement=="sensore_attivazione")
      |> filter(fn:(r)=> r.via=="{via}")
    """
    tables = query_api.query(flux)
    data = []
    for table in tables:
        for rec in table.records:
            data.append({
                "time":  rec.get_time().isoformat(),
                "value": rec.get_value()
            })
    return jsonify(data)

# ── SEZIONE 7: API PER MODALITÀ NOTTE
──────────────────────────────────────
@app.route("/api/night_mode", methods=["POST"])
def api_night_mode():
    # Riceve JSON {"mode":"ON"|"OFF"} e lo inoltra al servizio di
ingest (porta 5001)
    data = request.json or {}
    mode = data.get("mode", "OFF").upper()
    resp = requests.post("http://localhost:5001/night_mode",
                         json={"mode": mode})
    return jsonify(resp.json()), resp.status_code

# ── SEZIONE 8: API PER MODALITÀ EMERGENZA
──────────────────────────────────
```

```
@app.route("/api/emergency", methods=["POST"])
def api_emergency():
    # Riceve JSON {"via":"Nord/Est/Sud", "mode":"ON"|"OFF"}
    data = request.json or {}
    via  = data.get("via")
    mode = data.get("mode", "OFF").upper()
    resp = requests.post("http://localhost:5001/emergency",
                         json={"via": via, "mode": mode})
    return jsonify(resp.json()), resp.status_code

# ── SEZIONE 9: API PER STATISTICHE ──────────────────────────
@app.route("/stats")
def api_stats():
    # Proxy della chiamata al servizio ingest che espone /stats su
porta 5001
    resp = requests.get("http://localhost:5001/stats")
    return (resp.text, resp.status_code, {'Content-Type':
'application/json'})

# ── SEZIONE 10: ENTRY POINT ─────────────────────────────────
if __name__ == "__main__":
    # Avvia il server Flask in modalità di debug (riavvi
automatico su modifica)
    app.run(debug=True)
```

5. Front-End WebApp: index.html

```
<!DOCTYPE html>
<html lang="it">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1">
  <title>Gestione Incrocio Semaforico</title>

  <!-- ============================== -->
  <!-- 1) RISORSE ESTERNE           -->
  <!-- ============================== -->
  <!-- Google Font “Inter” per il testo -->
  <link
href="https://fonts.googleapis.com/css2?family=Inter:wght@400;600;
700&display=swap" rel="stylesheet">
```

```
  <!-- Font Awesome per le icone -->
  <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.4.0/cs
s/all.min.css"/>

  <!-- ============================== -->
  <!-- 2) STILI CSS                   -->
  <!-- ============================== -->
  <style>
    /* Variabili CSS globali */
    :root {
      --primary:   #1F2A44;
      --secondary: #3E9F9F;
      --accent:    #E94F37;
      --bg-light:  #F5F7FA;
      --bg-dark:   #FFFFFF;
      --text-dark: #2A303C;
      --text-light:#FFFFFF;
      --card-shadow: rgba(0, 0, 0, 0.1);
      --speed:     0.25s;
      --radius:    12px;
    }

    /* Reset e impostazioni di base */
    * {
      box-sizing: border-box;
      margin: 0;
      padding: 0;
    }
    body {
      font-family: 'Inter', sans-serif;
      background: var(--bg-light);
      color: var(--text-dark);
      line-height: 1.5;
    }

    /* ============================== */
    /* 2.1) HEADER                    */
    /* ============================== */
    header {
      background: var(--primary);
      color: var(--text-light);
      padding: 1rem 2rem;
      display: flex;
      align-items: center;
      justify-content: space-between;
      box-shadow: 0 2px 6px var(--card-shadow);
```

```
}
header h1 {
  font-size: 1.5rem;
  font-weight: 600;
}
/* Spazio tra icona e testo del titolo */
header h1 i {
  margin-right: 1rem;
}

/* ============================== */
/* 2.2) PULSANTI CONTROLLO       */
/* ============================== */
.controls {
  display: flex;
  flex-wrap: wrap;
  gap: .75rem;
}
.btn {
  display: flex;
  align-items: center;
  gap: .5rem;
  padding: .5rem 1rem;
  border: none;
  border-radius: var(--radius);
  font-weight: 600;
  cursor: pointer;
  transition: background var(--speed), transform var(--speed);
  box-shadow: 0 2px 4px var(--card-shadow);
}
.btn:hover {
  transform: translateY(-2px);
}
/* Stile pulsante notte */
.btn-night {
  background: var(--secondary);
  color: var(--text-light);
}
.btn-night.active {
  background: var(--accent);
}
/* Stile pulsanti emergenza */
.btn-emg {
  background: var(--accent);
  color: var(--text-light);
}
.btn-emg.active {
```

```
  background: #c0392b;
}

/* ============================== */
/* 2.3) LAYOUT PRINCIPALE       */
/* ============================== */
main {
  max-width: 1200px;
  margin: 2rem auto;
  padding: 0 1rem;
  display: grid;
  grid-template-rows: auto auto 1fr;
  row-gap: 2rem;
}

/* ============================== */
/* 2.4) SEZIONE STATISTICHE      */
/* ============================== */
.stats {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
  gap: 1rem;
}
.stat-box {
  background: var(--bg-dark);
  border-radius: var(--radius);
  padding: 1rem;
  text-align: center;
  box-shadow: 0 4px 8px var(--card-shadow);
  transition: transform var(--speed);
}
.stat-box:hover {
  transform: translateY(-4px);
}
.stat-box h3 {
  font-size: 1.25rem;
  margin-bottom: .75rem;
  color: var(--primary);
}
.stat-box p {
  font-size: .95rem;
  margin: .25rem 0;
}

/* ============================== */
/* 2.5) SEZIONE STATI SEMAFORI   */
/* _____________________________ */
```

```
    .status-group {
      display: grid;
      grid-template-columns: repeat(auto-fit, minmax(150px, 1fr));
      gap: 1rem;
    }
    .status-box {
      background: var(--bg-dark);
      border-radius: var(--radius);
      padding: 1rem;
      text-align: center;
      color: var(--text-dark);
      box-shadow: 0 4px 8px var(--card-shadow);
      transition: transform var(--speed);
    }
    .status-box:hover {
      transform: translateY(-4px);
    }
    .status-box h2 {
      font-size: 1rem;
      margin-bottom: .5rem;
    }
    .light {
      width: 60px;
      height: 60px;
      border-radius: 50%;
      background: grey;
      margin: .5rem auto;
      transition: background var(--speed);
      box-shadow: inset 0 0 8px rgba(0,0,0,0.2);
    }
    .sensor-indicator {
      font-size: 1.3rem;
      font-weight: 700;
      margin-top: .5rem;
      transition: color var(--speed);
    }

    /* ============================= */
    /* 2.6) SEZIONE GRAFICI          */
    /* ============================= */
    .panels {
      display: grid;
      grid-template-columns: 1fr 1fr;
      gap: 1rem;
      margin-bottom: 2rem;
    }
    .panel {
```

```
      background: var(--bg-dark);
      border-radius: var(--radius);
      overflow: hidden;
      box-shadow: 0 4px 8px var(--card-shadow);
      transition: transform var(--speed);
    }
    .panel:hover {
      transform: translateY(-4px);
    }
    .panel iframe {
      width: 100%;
      height: 300px;
      border: 0;
    }

    /* Responsive per schermi piccoli */
    @media (max-width: 768px) {
      .panels {
        grid-template-columns: 1fr;
      }
    }
  </style>
</head>
<body>

  <!-- ============================== -->
  <!-- 3) HEADER CON CONTROLLI      -->
  <!-- ============================== -->
  <header>
    <h1>
      <i class="fa-solid fa-traffic-light"></i>
      Incrocio Semaforico
    </h1>
    <div class="controls">
      <button id="btn-night" class="btn btn-night">
        <i class="fa-solid fa-moon"></i>
        Modalità Notte
      </button>
      <button data-via="Nord" class="btn btn-emg">
        <i class="fa-solid fa-triangle-exclamation"></i>
        Emergenza Nord
      </button>
      <button data-via="Est" class="btn btn-emg">
        <i class="fa-solid fa-triangle-exclamation"></i>
        Emergenza Est
      </button>
      <button data-via="Sud" class="btn btn-emg">
```

```
      <i class="fa-solid fa-triangle-exclamation"></i>
      Emergenza Sud
    </button>
  </div>
</header>

<!-- ============================== -->
<!-- 4) CONTENUTI PRINCIPALI       -->
<!-- ============================== -->
<main>
  <!-- 4.1) Statistiche -->
  <section class="stats" id="stats-container">
    <!-- Popolato dinamicamente da JS -->
  </section>

  <!-- 4.2) Stato Semafori -->
  <section class="status-group">
    <div class="status-box">
      <h2>Semaforo Nord</h2>
      <div class="light" id="light-Nord"></div>
    </div>
    <div class="status-box">
      <h2>Semaforo Est</h2>
      <div class="light" id="light-Est"></div>
    </div>
    <div class="status-box">
      <h2>Semaforo Sud</h2>
      <div class="light" id="light-Sud"></div>
    </div>
  </section>

  <!-- 4.3) Stato Sensori -->
  <section class="status-group">
    <div class="status-box">
      <h2>Sensore Nord</h2>
      <div class="sensor-indicator" id="sensor-Nord">NO</div>
    </div>
    <div class="status-box">
      <h2>Sensore Est</h2>
      <div class="sensor-indicator" id="sensor-Est">NO</div>
    </div>
    <div class="status-box">
      <h2>Sensore Sud</h2>
      <div class="sensor-indicator" id="sensor-Sud">NO</div>
    </div>
  </section>
```

```
    <!-- 4.4) Grafici Grafana -->
    <section class="panels">
      <div class="panel">
        <iframe id="iframe-stato"></iframe>
      </div>
      <div class="panel">
        <iframe id="iframe-sensore"></iframe>
      </div>
    </section>
  </main>

  <!-- ============================== -->
  <!-- 5) SCRIPT JAVASCRIPT           -->
  <!-- ============================== -->
  <script>
    // 5.1) Variabili globali (passate da Flask)
    const grafanaUrl   = "{{ grafana_url }}",
          dashboardUid = "{{ dashboard_uid }}",
          dashboardSlug= "{{ dashboard_slug }}",
          panelStato   = "{{ panel_id_stato }}",
          panelSensore = "{{ panel_id_sensore }}";

    // Elenco delle direzioni
    const dirs   = ['Nord','Est','Sud'];

    // Mappa di elementi DOM per semafori e sensori
    const lights  = Object.fromEntries(dirs.map(v=>[v,
document.getElementById('light-'+v)]));
    const sensors = Object.fromEntries(dirs.map(v=>[v,
document.getElementById('sensor-'+v)]));
    const ifSt    = document.getElementById('iframe-stato');
    const ifSe    = document.getElementById('iframe-sensore');
    const btnNight= document.getElementById('btn-night');
    const btnsEmg = document.querySelectorAll('.btn-emg');
    const statsC  = document.getElementById('stats-container');

    // Stati interni
    let nightOn=false, emgVia=null, blinkState=false,
blinkHandle=null;

    // 5.2) Costruzione URL Grafana embed
    function buildIframe(id) {
      const base =
`${grafanaUrl}/d-solo/${dashboardUid}/${dashboardSlug}`;
      const params = new URLSearchParams({
        orgId: 1,
        from: 'now-5m',
```

```
        to: 'now',
        panelId: id,
        refresh: '1s'
      });
      return `${base}?${params.toString()}`;
    }

    // Inizializza gli iframe dei grafici
    function initIframes() {
      ifSt.src = buildIframe(panelStato);
      ifSe.src = buildIframe(panelSensore);
    }

    // 5.3) Funzioni per lampeggio in modalità Notte
    function startBlink() {
      if (blinkHandle) return;
      blinkHandle = setInterval(() => {
        blinkState = !blinkState;
        const col = blinkState ? '#F1C40F' : 'transparent';
        dirs.forEach(v => lights[v].style.backgroundColor = col);
      }, 500);
    }
    function stopBlink() {
      if (blinkHandle) {
        clearInterval(blinkHandle);
        blinkHandle = null;
      }
    }

    // 5.4) Aggiorna le statistiche ogni 20s
    async function updateStats() {
      try {
        const res = await fetch('/stats'),
              data= await res.json();
        statsC.innerHTML = '';
        dirs.forEach(v => {
          const s = data[v] || { vehicles:0, changes:0,
avg_green_ms:0 };
          const box = document.createElement('div');
          box.className = 'stat-box';
          box.innerHTML = `
            <h3>${v}</h3>
            <p><i class="fa-solid fa-car"></i> Veicoli:
${s.vehicles}</p>
            <p><i class="fa-solid fa-exchange-alt"></i> Cambi
stati: ${s.changes}</p>
            <p><i class="fa-solid fa-stopwatch"></i> Verde medio:
```

```
${(s.avg_green_ms/1000).toFixed(2)}s</p>
          `;
          statsC.appendChild(box);
        });
      } catch(e) {
        console.error('Errore stats', e);
      }
    }

    // 5.5) Aggiorna lo stato di semafori e sensori (ogni 1s)
    async function updateStatus() {
      // Modalità Emergenza
      if (emgVia) {
        stopBlink();
        dirs.forEach(v => {
          lights[v].style.backgroundColor = (v===emgVia ?
'#27AE60' : '#E74C3C');
          sensors[v].textContent = '–';
          sensors[v].style.color = '#E74C3C';
        });
        return;
      }
      // Modalità Notte
      if (nightOn) {
        startBlink();
        return;
      }
      // Funzionamento normale
      stopBlink();
      for (let v of dirs) {
        // Stato semaforo
        const r1 = await fetch(`/api/stati?via=${v}&range=-5m`),
              d1 = await r1.json();
        if (d1.length) {
          const last = d1[d1.length-1].value;
          let col = '#BDC3C7';  // default grigio
          if (last===2) col='#27AE60';
          if (last===1) col='#F1C40F';
          if (last===0) col='#E74C3C';
          lights[v].style.backgroundColor = col;
        }
        // Stato sensore
        const r2 = await fetch(`/api/sensori?via=${v}&range=-5m`),
              d2 = await r2.json();
        if (d2.length) {
          const on = d2[d2.length-1].value===1;
          sensors[v].textContent = on?'SÌ':'NO';
```

```
        sensors[v].style.color   = on?'#27AE60':'#E74C3C';
      } else {
        sensors[v].textContent = 'NO';
        sensors[v].style.color = '#E74C3C';
      }
    }
  }

  // 5.6) Gestione pulsante “Modalità Notte”
  btnNight.addEventListener('click', async () => {
    nightOn = !nightOn;
    emgVia = null;
    btnsEmg.forEach(b => b.classList.remove('active'));
    btnNight.classList.toggle('active', nightOn);
    await fetch('/api/night_mode', {
      method: 'POST',
      headers: {'Content-Type':'application/json'},
      body: JSON.stringify({mode: nightOn ? 'ON' : 'OFF'})
    });
    updateStatus();
  });

  // 5.7) Gestione pulsanti “Emergenza”
  btnsEmg.forEach(btn => {
    btn.addEventListener('click', async () => {
      const via = btn.dataset.via;
      const activate = (emgVia !== via);
      if (activate && nightOn) {
        nightOn = false;
        btnNight.classList.remove('active');
      }
      // Disattiva eventuale emergenza precedente
      if (emgVia) {
        await fetch('/api/emergency', {
          method: 'POST',
          headers: {'Content-Type':'application/json'},
          body: JSON.stringify({via: emgVia, mode: 'OFF'})
        });
      }
      // Attiva/disattiva la nuova emergenza
      if (activate) {
        await fetch('/api/emergency', {
          method: 'POST',
          headers: {'Content-Type':'application/json'},
          body: JSON.stringify({via, mode: 'ON'})
        });
        emgVia = via;
```

```
        } else {
          emgVia = null;
        }
        btnsEmg.forEach(b => b.classList.toggle('active',
b.dataset.via===emgVia));
        updateStatus();
      });
    });

    // 5.8) INIZIALIZZAZIONE ALL'AVVIO DEL DOM
    document.addEventListener('DOMContentLoaded', () => {
      initIframes();           // Carica i grafici Grafana
      updateStatus();          // Mostra subito stato corrente
      setInterval(updateStatus, 1000);  // Refresh stato ogni
secondo
      updateStats();           // Mostra subito statistiche
      setInterval(updateStats, 20000);  // Refresh statistiche
ogni 20s
    });
  </script>
</body>
</html>
```

Percorso progettuale

1. Diario

Fase 1: Dai requisiti al concept

All'inizio ci siamo riuniti per tradurre i requisiti della consegna in una bozza di architettura: due Micro:bit, un database time-series e una webapp interattiva. Abbiamo definito i casi d'uso principali (traffico normale, notte e emergenza) e scelto InfluxDB + Grafana per il monitoraggio storico, Flask per il backend e HTML/CSS/JS per il frontend.

Fase 2: Allestimento hardware e ambiente di sviluppo

Abbiamo montato il prototipo fisico: un Micro:bit A alimentato a batterie collegato a tre semafori (rosso, giallo, verde) e a tre rispettivi sensori di pressione; un Micro:bit B come gateway radio↔seriale collegato al computer. Nel frattempo abbiamo installato InfluxDB e Grafana sul portatile Windows e creato il virtual environment Python con tutte le librerie (flask, pyserial). Abbiamo poi verificato che InfluxDB fosse raggiungibile su localhost:8086 e che Grafana rispondesse su localhost:3000.

Fase 3: Firmware semaforico (Micro:bit A)

Con l’hardware pronto, siamo passati alla stesura dello script del Micro:bit A. Abbiamo implementato la funzione tutti_rossi() e la subroutine per il ciclo rosso–verde–giallo, calibrando i tempi MIN e MAX e gestendo il “grace period” al rilascio del sensore. I primi test hanno rivelato qualche sfarfallio nel lampeggio, risolto aggiungendo un piccolo sleep(50) dopo ogni radio.send(). Al termine di questa fase, il Master era capace di inviare correttamente via radio messaggi STATE;… e SENSOR;….

Fase 4: Gateway e forwarding (Micro:bit B)

Successivamente, abbiamo scritto lo script per Micro:bit B: un loop che legge da uart.readline() e da radio.receive(), inoltra i comandi CMD dal PC a Micro:bit A e passa tutti i messaggi di stato sensori al PC in seriale. I primi tentativi mostravano pacchetti troncati a causa del buffer, così abbiamo introdotto delle correzioni come un piccolo timeout e un flush iniziale in fase di avvio. Alla fine il Gateway si è rivelato stabile e trasparente.

Fase 5: Ingestione dati su InfluxDB

La parte principale del backend, microbit_to_influx.py, è stato costruito basandoci su quello di esempio ed è servito trasformare in modo robusto messaggi arbitrari in line protocol InfluxDB. Abbiamo scritto un parser basato su regex, gestito typo (“VEDE” → “VERDE”), frammenti malformati e mapping numerico degli stati. Dopo aver avviato lo script, i punti e i bucket su InfluxDB si sono popolati in tempo reale, e abbiamo subito creato i primi pannelli in Grafana per verificare che i dati fossero coerenti.

Fase 6: Estensioni – Emergenza e Statistiche

In questa fase abbiamo integrato la modalità emergenza, che permette di forzare un’unica corsia su verde e la raccolta di statistiche (conteggio veicoli, cambi di stato, tempo medio verde). Il calcolo lato ingest ha richiesto di memorizzare timestamp di inizio/fine verde in variabili globali e di esporre un endpoint /stats. Il frontend aggiorna automaticamente questi dati ogni 5 s, mostrando un quadro completo dell’efficienza dell’incrocio.

Fase 7: WebApp – Backend Flask e API

Con il database operativo, abbiamo creato app.py in Flask: endpoint /api/stati, /api/sensori per interrogare InfluxDB, e proxy /api/night_mode, /api/emergency, /stats verso lo script di ingest. Le difficoltà principali sono state legate alla sincronizzazione tra porte seriali e ai permessi di accesso su COM5, risolte chiudendo altri processi che occupavano la porta e isolando il servizio di ingest su un terminale dedicato.

Fase 8: WebApp – Frontend e grafica

Per il frontend abbiamo scelto un layout a grid CSS responsive, pulsanti “Modalità Notte” e “Emergenza” con feedback visivi immediati, e un restyling con palette coordinate e icone Font-Awesome. Il polling delle API (1 s per stati, 5 s per statistiche) è stato ottimizzato per evitare sovraccarichi.

Fase 9: Testing e bug-fixing

È stata seguita una lunga fase di collaudo: pressione ripetuta dei pulsanti sensori, accensione/spegnimento modalitá, prove di stress sui cicli notturni e di emergenza.

Fase 10: Documentazione e consegna

Infine abbiamo redatto la documentazione tecnica completa: architettura, comunicazione dati, database, webapp, flusso logico con diagrammi Mermaid, avvio/spegnimento e diario progettuale. Abbiamo poi preparato uno schema circuitale, un flowchart e un diario in forma discorsiva.

2. Difficoltà riscontrate

Durante la realizzazione del sistema, sono emerse diverse criticità di natura sia hardware che software, in parte dovute alla complessità del progetto e in parte alla nostra iniziale inesperienza pratica.

Una delle principali difficoltà ha riguardato il cablaggio del sistema. Il numero elevato di collegamenti tra i sensori di pressione, i LED dei semafori (rosso, giallo, verde) e le uscite della scheda Micro\:bit ha richiesto una notevole attenzione. L'utilizzo dello shield Sensor Shield V2.0 del kit Keyestudio ha semplificato alcuni aspetti, ma la mancanza di familiarità iniziale con la disposizione fisica dei pin (soprattutto nel distinguere pin analogici, digitali e i relativi VCC e GND) ha reso complessa l'attività di collegamento. Inoltre, l'impiego di cavetteria fitta e ravvicinata ha aumentato il rischio di errori di connessione, cortocircuiti accidentali e scarsa leggibilità visiva dell'impianto, costringendoci a frequenti revisioni del wiring e a un'attenta pianificazione della disposizione fisica dei moduli.

Un'altra criticitàè stata quella legata all'accesso concorrente alla porta seriale COM5, utilizzata per la comunicazione UART tra il Micro:bit ricevitore e l'applicazione Python. L'esecuzione simultanea dello script di acquisizione (microbit_to_influx.py) e dell'applicazione web (app.py), entrambi configurati per aprire la stessa porta seriale, ha generato un errore di tipo PermissionError (codice 13: "Accesso negato"), impedendo l'avvio corretto del server Flask.

Per risolvere il problema è stato adottato un modello di separazione dei ruoli: la lettura dalla porta seriale è stata delegata esclusivamente allo script microbit_to_influx.py, che si occupa di acquisire, validare e inviare i dati a InfluxDB. L'applicazione web accede ai dati solo attraverso query al database, senza interagire direttamente con la seriale. Questo approccio ha eliminato i conflitti di accesso, aumentato la stabilità del sistema e migliorato la scalabilità.

Al di là di queste criticità, il resto del progetto si è sviluppato in maniera relativamente fluida. Le funzionalità di base, tra cui la gestione semaforica intelligente, la comunicazione via radio tra le due Micro\:bit, la scrittura su InfluxDB e la realizzazione della web app interattiva, hanno richiesto un buon livello di ragionamento logico, ma sono state affrontate in modo efficace e coordinato. Il lavoro è stato distribuito equamente all'interno del gruppo, consentendo una gestione sinergica delle competenze e una progressiva risoluzione dei problemi riscontrati durante le varie fasi di sviluppo.