Caso di Studio: un robot autonomo per competizioni studentesche

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it

Programmazione Sistemi Robotici

Caratteristiche (1)

- **Velocità/Accelerazione**
  - Solo 100 secondi di durata della gara
  - Necessità di effettuare velocemente i percorsi
  - Tuttavia i percorsi sono spesso brevi $\Rightarrow$ buona accelerazione
- **Precisione**
  - Capacità (dei meccanismi) di raggiungere **sempre** la posizione richiesta (con tolleranza **nota**)
- **Adattabilità alle tolleranze**
  - Il mondo reale non è “preciso”, occorre essere adattabili a eventuali imprecisioni (entro margini adeguati)

Caratteristiche (2)

- **Ripetibilità/Affidabilità**
  - Bug Free
  - Capacità di “ripetere sempre” le stesse cose, con le stesse performance
- **Flessibilità**
  - Capacità di adeguarsi a situazioni impreviste (problemi posizionamento, oggetti di gara in posizioni non previste, presenza dell’avversario, etc.)

Piattaforma Hardware e Executive

- Scheda embedded equipaggiata con MCU STM32F446RE
- Executive basato sul sistema operativo real-time NuttX (http://www.nuttx.org)
- Software di controllo e di strategia (multi-task) interamente scritto in C++
- Circa 30000 righe di codice (escluso la parte del kernel)

Task di Controllo

- **Motion Control**
- **Telemetry**
- **GPIO**
- **Strategy**

Strategy (st)

- Implementazione del comportamento del robot
- Pianificazione degli obiettivi
- Pianificazione dei percorsi
- Gestione degli ostacoli

Entità **Goal**

- Classe astratta **`Goal`**
- Metodo **`int feasible()`**: include il codice per per verificare se il goal è fattibile e con quale priorità.
  Valori di ritorno:
  - < 0 goal non fattibile
  - > 0 goal fattibile, valore numerico più basso $\Rightarrow$ priorità maggiore

```
class Goal {
  public:
    bool isAchieved();
    void setAchieved(bool ach);

  protected:
    virtual int feasible() = 0;
    virtual success_t execute() = 0;
};
```

Entità **Goal**

- Classe astratta **`Goal`**
- Metodo **`success_t execute()`**: include il codice da eseguire per raggiungere il goal.
  Valori di ritorno:
  - *FAIL* il goal non è stato raggiunto
  - *SUCCESS* il goal è stato raggiunto con successo

```
class Goal {
  public:
    bool isAchieved();
    void setAchieved(bool ach);

  protected:
    virtual int feasible() = 0;
    virtual success_t execute() = 0;
};
```

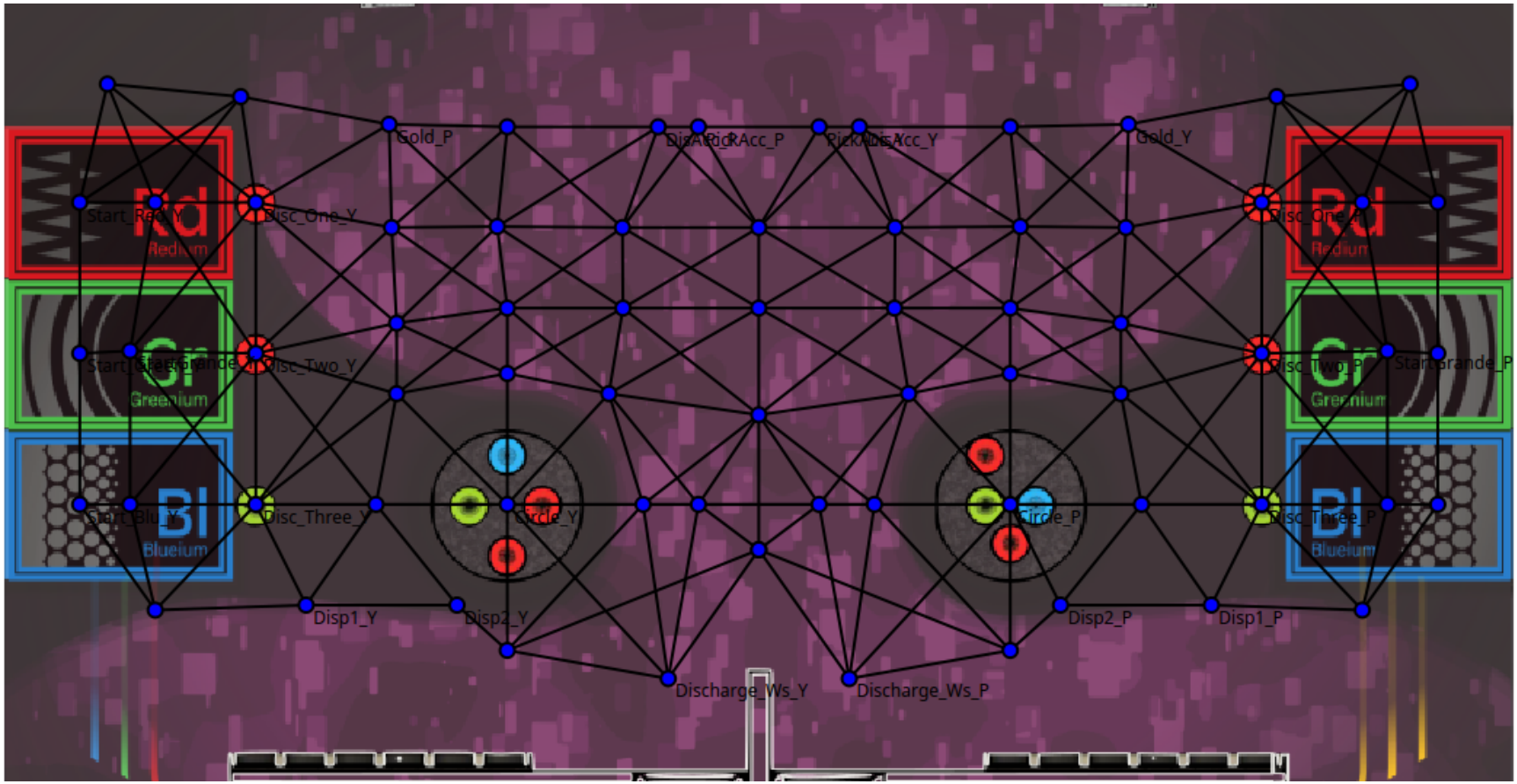
Algoritmo di scheduling

Scheduling

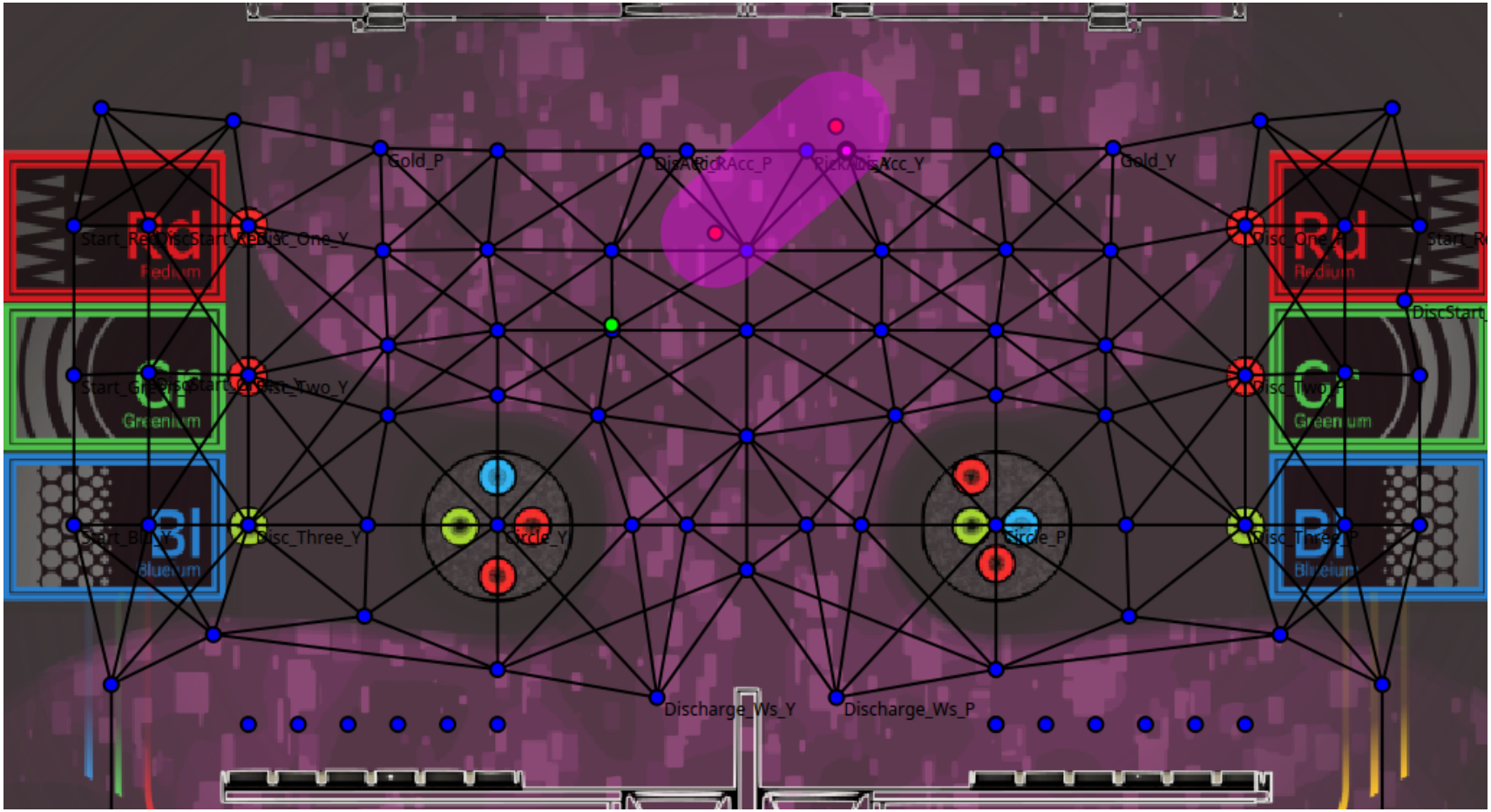
```
while True do
    selected_goal ← {g ∈ GOAL_SET | g.is_feasible() is min and > 0};
    if selected_goal is nil then
        continue;
    end
    if selected_goal.execute() = SUCCESS then
        // Remove goal from goalset
        GOAL_SET ← GOAL_SET − selected_goal;
    end
end
```

Pianificazione dei Percorsi

Fixed Graph


Gold_P
Gold_Y
Disc_One_Y
Disc_Two_Y
Disc_Three_Y
Disp1_Y
Disp2_Y
Disp2_P
Disp1_P
Discharge_Ws_Y
Discharge_Ws_P

Fixed Graph e Obstacle Avoidance



Path Planning e Goal PlanningTask

Path e Goal

- Il sistema dei goal e la pianificazione dei percorsi sono strettamente legati
- La funzione **`measurePath(dest)`** restituisce:
  - la lunghezza del path verso una destinazione, oppure
  - -1 se la destinazione non è raggiungibile
- Può essere inserita nella **`feasible()`** di un goal e il valore di ritorno utilizzato per stabilire la priorità del goal stesso

Path e Goal

- Ogni goal è infatti caratterizzato da un **goal point** ossia la posizione da raggiungere per poi effettuare le azioni (presa o deposizione dischi)
- La funzione `doPath(dest)` calcola e avvia un percorso verso una destinazione
- Essa è la prima funzione invocata da un goal

Eventi

- Il codice di un goal necessita di parecchie condizioni di attesa (percorso completato, ostacolo, sensore attivato, etc.)
- La gesione di tali condizioni è modellata attraverso un sistema di attesa e notifica di **eventi**
- L'utente può creare un'istanza di `EventChannel` per ottenere un canale di comunicazione tra task basato su notifica di eventi
- L'evento è una sottoclasse della classe astratta `Event` ed e' caratterizzato da:
  - un tag che identifica la tipologia di evento
  - uno o più parametri associati all'evento

Goal ed Eventi

EventChannel: Metodi

- `wait(...)` sospende il task in attesa che si verifichi di uno degli "n" eventi che vengono forniti come parametro
- `wait(int millis, ...)` effettua l'attesa, consierando un possibile timeout specificato in millisecondi
- `post_event(...)` invia uno specifico evento sul canale; se un altro task era in attesa di tale evento esso viene risvegliato
- Il meccanismo di attesa e notifica si basa su una coppia *mutex/condition variable*

Motion Control (mc)

- Controllo velocità ruote
- Odometria
- Controllo movimenti vari (rotazioni, controllo polare, etc.)
- Gestione di una coda di comandi
- Processo periodico $\Delta T = 5\ ms$
- Priorità massima, scheduling FIFO

Periodicità

- Utilizzo di un timer, virtualizzato su un device driver (**/dev/timer0**) e associato (nel driver) ad un timer hardware
- Programmato per generare un segnale (*signal*) ad un period prefissato

```
// Timer that fires SIGALRM
int timer_fd = open("/dev/timer0", O_RDONLY);
if (timer_fd < 0)
    perror("Failed to open timer");

struct timer_notify_s notify;
notify.arg = NULL;
notify.pid = getpid();
notify.signo = SIGALRM;
ioctl(timer_fd, TCIOC_NOTIFICATION, (unsigned long)((uintptr_t)&notify));

uint32_t timeout = 1000000 / 200; // 200 Hz
ioctl(timer_fd, TCIOC_SETTIMEOUT, timeout);
ioctl(timer_fd, TCIOC_START, 0);
```

Periodicità

- Il main loop attende l'evento di timer ...
- acquisisce il tempo corrente ...
- avvia un'iterazione per tutti i task di controllo

```
while (true) {
    if (sigwaitinfo(&mask, nullptr) != SIGALRM)
        continue;

    float t = TICK2SEC( (float)clock_systimer() );

    mc_scheduler.run_all_tasks(t);
}
```

Task di Controllo

1. **Odometria** classe `Odometry`
2. **Controllo velocità ruote** classe `SpeedControlTask`
3. Controlli in posizione (mutuamente esclusivi)
   - `AbsoluteRotation`
   - `RelativeRotation`
   - `CircularRotation`
   - `GoTo_Point`
4. **Controllo blocco ruote**, classe `LockChecker`
5. **Coda comandi** classi `CommandQueue` e `CommandPool`

Task

Telemetry

- Campionamento dei sensori
- Trasmissione dati sensori (su richiesta) via wireless (per monitoraggio remoto/notifica remota)
- Gestione degli ostacoli tramite arresto istantaneo del robot e notifica dell'evento al processo di strategia

Task

GPIO

- Campionamento delle linee di GPIO
- Identificazione dei fronti (rising o falling)
- Generazione di eventi relativamente ai fronti rilevati

Caso di Studio: un robot autonomo per competizioni studentesche

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it

Programmazione Sistemi Robotici