Sistemi Embedded e IoT

Ingegneria e Scienze Informatiche - UNIBO
a.a 2019/2020
Docente: Prof. Alessandro Ricci

[modulo-lab-4.1]
MODELLI DI COMUNICAZIONE A SCAMBIO DI MESSAGGI

SOMMARIO

- In questo modulo si discutono i modelli di comunicazione a scambio di messaggi, considerandone l'integrazione nelle architetture/modelli di controllo visti nella programmazione di sistemi embedded delle scorse lezioni

MODELLI DI COMUNICAZIONE

- I protocolli e tecnologie viste nei moduli 4 del corso abilitano fisicamente la comunicazione in e fra sistemi embedded
- Al di là di queste tecnologie, la progettazione e programmazione di sistemi software embedded di rete e distribuiti richiede l'introduzione di opportuni **modelli e meccanismi di comunicazione di alto livello**, e i relativi supporti tecnologici a livello di librerie / piattaforme / infrastrutture
- Tra le questioni
  - quale modello di comunicazione?
    - es: a scambio di messaggi? Sincrono, asincrono?
  - quali primitive? Quale semantica relativa agli atti comunicativi?
  - quale linguaggio di comunicazione, ovvero: quali modelli e linguaggi scegliamo per rappresentare le informazioni scambiate?
    - problema dell'interoperabilità
  - In che modo integrare il modello di comunicazione con il modello usato per descrivere il comportamento del controllore?
  - Come rappresentare e implementare protocolli di comunicazione articolati?

MODELLO A SCAMBIO DI MESSAGGI

- Comunicazione mediante invio e ricezione di messaggi
  - modello alternativo alla comunicazione mediante memoria condivisa
  - modello di riferimento per i sistemi distribuiti
- Primitive
  - **send**
  - **receive**
- Varie tipologie e aspetti
  - diretta o indiretta
  - sincrona o asincrona
  - linguaggio descrizione messaggi
    - interoperabilità

DIRETTA VS. INDIRETTA

- Comunicazione **diretta**:
  - la comunicazione avviene direttamente fra i processi (thread,…), specificando il loro identificatore
    - aspetto importante: come identificare partecipanti alla comunicazione in modo univoco
  - primitive:
    - send (ProcId,Msg)
    - receive(): Msg
- Comunicazione **indiretta**
  - si utilizzano dei *canali* che fungono da mezzi di comunicazione indiretta.
  - L'invio e la ricezione avvengono specificando uno specifico canale
    - send(ChannelId,Msg)
    - receive(ChannelId):Msg

SINCRONA VS. ASINCRONA

- Comunicazione **sincrona**:
  - la send ha successo (e completa) quando il messaggio specificato è ricevuto dal destinatario mediante una receive
- Comunicazione **asincrona**
  - la send ha successo (e completa) quando il messaggio è stato inviato
    - ma non necessariamente ricevuto dal destinatario mediante una receive

BUFFERIZZAZIONE

- Un altro aspetto importante è la strategia di bufferizzazione adottata, fondamentale nel caso asincrono
  - sia nel caso indiretto, per i canali
  - sia nel caso diretto, per la coda che i processi usano per ricevere i messaggi
- Concerne la dimensione del buffer ove vengono memorizzati i messaggi ricevuti, ma ancora da ricevere con la receive

PRIMITIVE: VARI TIPI DI SEMANTICA

- **send (DestId,Msg)**
  - ha successo quando (semantiche alternative):
    - (1) il msg è stato inviato (=> caso **asincrono**)
      - in questo caso la verifica che un messaggio sia arrivato a destinazione deve essere implementata a livello di protocollo
      - errore se il destinatario non esiste
    - (2) il msg è arrivato, ma non necessariamente ricevuto
      - errore se il destinatario non esiste
    - (3) il msg è stato ricevuto
      - caso sincrono
- **receive():Msg**
  - semantica usuale: **bloccante**
    - si blocca sin quando non è disponibile almeno un messaggio

BUFFER FULL - CHE FARE?

- Nel caso asincrono, deve essere considerato anche il caso in cui il buffer di ricezione dei messaggi si riempia
- In quel caso occorre scegliere il tipo di semantica per la send
  - fallisce
    - msg scartato
  - si blocca in attesa che il buffer non sia pieno
    - stile produttori/consumatori
  - ha successo
    - riscritto messaggio del buffer
      - più vecchio, più nuovo,…

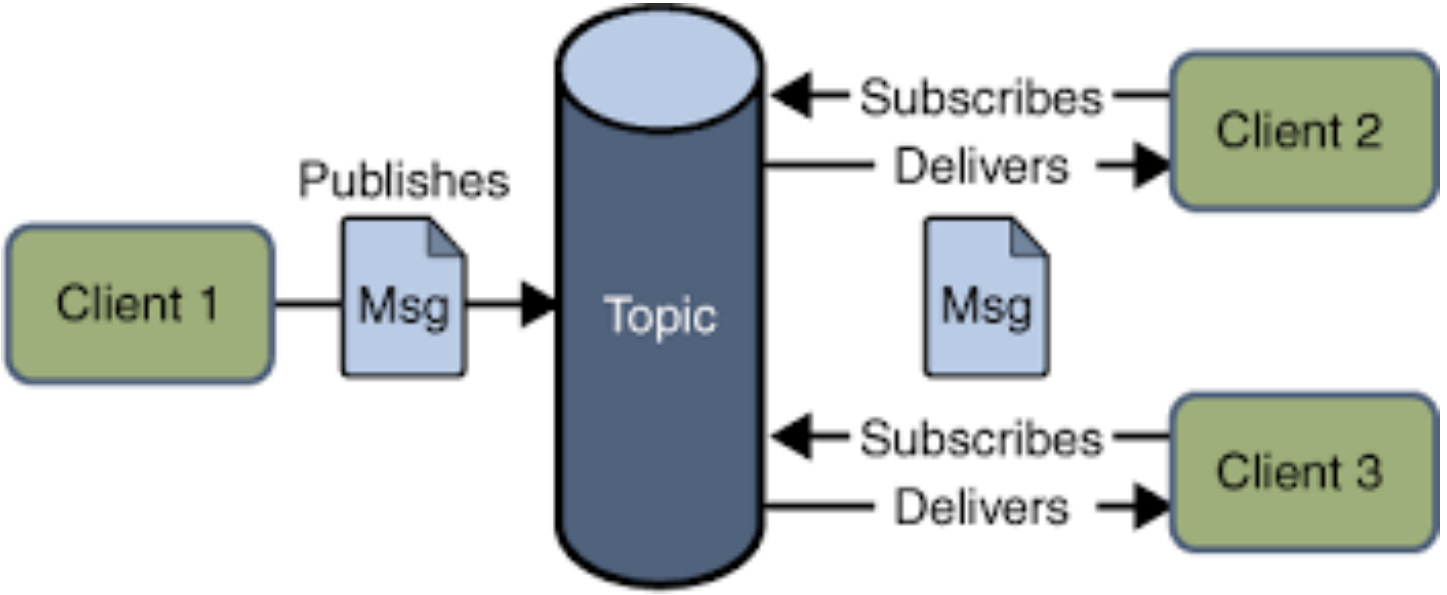
RAPPRESENTAZIONE MESSAGGI

- Aspetto fondamentale della comunicazione è dato dal "linguaggio" con cui si rappresentano i messaggi
  - deve essere il medesimo utilizzato da tutti i partecipanti alla comunicazione, come primo requisito per avere ***interoperabilità***
- Esempi di linguaggi usati per la rappresentazione di messaggi per avere interoperabilità:
  - **JSON,** XML
- L'utilizzo del medesimo linguaggio non è sufficiente per avere interoperabilità
- Occorre mettersi d'accordo su
  - formato messaggi / vocabolario utilizzato
    - *ontologie*
  - in caso di protocolli => semantica dei protocolli

MODELLO PUBLISH/SUBSCRIBE

- Modello a scambio di messaggi tipicamente usato per realizzare architetture ad eventi ad alto livello in sistemi distribuiti
  - simile concettualmente al *pattern observer*
  - utile per la comunicazione in sistemi *open, dynamic, loosely-coupled*
  - molto usato in ambito IoT
    - prossimo modulo
- Meta-modello
  - **channels/topics**
    - canali/argomenti sui quali è possibile inviare messaggi e che è possibile “osservare” mediante sottoscrizione
  - **publishers**
    - inviano messaggi su canali
  - **subscribers**
    - si registrano su canale/topic per ricevere tutti messaggi pubblicati su quel canale/topic
- Primitive (astratte)
  - publish, subscribe

MODELLO PUBLIC-SUBSCRIBE



MODELLI DI COMUNICAZIONE E ARCHITETTURE DI CONTROLLO

- Aspetto importante
  - integrazione modelli di comunicazione con i modelli e architetture software di controllo visti per i sistemi embedded
  - ovvero:
    - loop di controllo semplici
    - macchine a stati finiti
    - task
    - event-loop

INTEGRAZIONE NEI SISTEMI EMBEDDED ORGANIZZATI A LOOP

Importante

- Problema: receive *bloccante*
  - comporta il blocco del loop
  - non è usabile nell'implementazione di macchine a stati finiti (sincrone o asincrone)
- Esempio semplice:
  - sistema blinking che è possibile controllare dall'esterno con invio di messaggio stop
    - seiot.modulo_lab_3_2.blinker_with_msg
  - altri esempi nel materiale
    - seiot.modulo_lab_3_2.pingpong
    - seiot.modulo_lab_2_2.msg (PingPong via seriale)

ESTENSIONE DEL MODELLO

Importante

- **Receive non bloccante:**
  - receiveMsg(): { Msg, errore/null }
  - nel caso in cui non ci siano messaggi, la receive non si blocca
    - restituisce errore o genera eccezione o restituisce un riferimento null
- Aggiunta di una primitiva per testare presenza messaggi disponibili:
  - **isMsgAvailable**(): boolean

INTEGRAZIONE NEL MODELLO FSM

Importante

- Sfruttando l'estensione, l'integrazione nel modello a stati può avvenire considerando che
  - una transizione può essere scatenata dalla presenza di un messaggio => uso della primitiva di test messaggi nelle guardie
  - nell'insieme delle azioni associate alla transizione o allo stato c'è anche la ricezione (non bloccante) e l'invio di messaggi
- NOTA:
  - nelle guardie non può essere modificato lo stato: sono solo predicati..
    - quindi si può usare isMsgAvailable, non receiveMsg…

PROBLEMA SELEZIONE MESSAGGI

- Supponiamo che nel medesimo stato possano essere ricevuti più messaggi, di tipo diverso, e che a seconda del messaggio ricevuto la macchina a stati debba transitare in uno stato diverso.
- In questo caso le primitive fornite non sono sufficientemente espressive per implementare questo comportamento
- Un modo per risolvere il problema è introdurre primitive che permettano di esplicitare in modo più specifico *quali messaggi* ricevere.

MSG PATTERN

- Un modo per implementare la soluzione è mediante l'uso di **pattern**:
  - isMsgAvailable(Pattern pattern): boolean
    - è true se è presente un messaggio che corrisponde al pattern specificato
  - receiveMsg(Pattern pattern): Msg
    - recupera un qualsiasi messaggio che corrisponde al pattern specificato
- Un pattern fornisce un modo per *denotare un insieme dei messaggi*
  - in questo modo possiamo indicare se sono presenti messaggi che appartengono allo specifico insieme o di ricevere messaggi che appartengono allo specifico insieme

IMPLEMENTAZIONE DI PATTERN

- Un pattern può essere rappresentato o da un predicato, o analogamente da una interfaccia con metodo match implementata da classi concrete

```
interface Pattern {
  boolean match(Msg msg);
}
```

- Nelle primitive:
  - isMsgAvailable(Pattern pattern): boolean
    - è true se è presente almeno un messaggio che soddisfa la funzione match del pattern
  - receiveMsg(Pattern pattern): Msg
    - seleziona un messaggio (se presente) che soddisfi la funzione match del pattern

INTEGRAZIONE NEL MODELLO A TASK

- Nel modello a task lo scambio di messaggi può essere usato come modello di comunicazione fra task stessi, in alternativa o a complemento del modello basato su memoria condivisa

INTEGRAZIONE NEL MODELLO AD EVENT LOOP

Importante

- Anche nel caso di architetture di event-loop, il modello di comunicazione da considerare è necessariamente asincrono con receive *implicita* oppure non bloccante
  - questo poiché negli event handler non è possibile eseguire primitive bloccanti
- Integrazione
  - l'arrivo di un messaggio è modellato come evento
    - la coda di eventi è usata come coda di messaggi
  - l'invio di un messaggio tramite send ha come effetto accodare un evento nella coda degli eventi del destinatario

ESEMPIO

- Nel materiale è fornito un esempio in cui un agente con architettura di controllo event-loop scambia messaggi via Seriale / Bluetooth mediante un componente MsgService implementato come risorsa osservabile
  - `package seiot.modulo_lab_4_3`
    - `MsgService, MsgEvent, MyAgent`

Importante