Fase2ISS25-CarolloSilvia

Repository: https://github.com/SvevaNullBologna/ISS_SilviaCarollo_2025.git

Abbiamo esteso il lavoro svolto nella fase 1 sviluppando una soluzione distribuita basata sull'uso del linguaggio Qak e del modello ad attori.
Qak consente la modellazione eseguibile di sistemi distribuiti ispirati all'architettura a microservizi, grazie ad attori che si comportano come automi a stati finiti e interagiscono tramite messaggi asincroni.

Abbiamo quindi evoluto il gioco Conway Life come sistema distribuito, trasformando ogni cella in un attore indipendente.
L'interazione è diventata parte integrante del comportamento software, poiché ogni attore è naturalmente in grado di inviare e ricevere messaggi senza la necessità di un'infrastruttura di comunicazione aggiuntiva.

1. Finalità dei contenuti discussi in questa prima fase e aspettative sulle prossime fasi

Durante la secondofase del corso, la finalità principale è stata quella di **approfondire la distribuzione dei sistemi software**, approfondendo in particolare componenti che comunicavano tra loro tramite protocolli di rete come TCP, CoAP e MQTT.

Approfondendo il linguaggio **qak**, abbiamo trovato il modo di creare attori, elementi automatizzati che possono rappresentare i nostri device.

Noi ci siamo concentrati sull'uso di **MQTT**, implementando sistemi in grado di **ricevere e inviare comandi tra attori remoti**, anche in ambiente Raspberry.

Un altro tema importante è stato l'**automazione dei comportamenti distribuiti**, resa possibile attraverso l'uso del linguaggio **QAK**, che ci ha permesso di modellare attori come **automi a stati finiti**, in modo da poter modellare le celle del Game of Life in modo autonoma, ma capaci di comunicare tra loro con un attore direttore che coordina il sistema.

Mi aspetto che le prossime fasi del corso si focalizzino su una **maggiore complessità nella comunicazione**, l'evoluzione verso **sistemi distribuiti reattivi e resilienti**, e magari anche sull'**integrazione tra software e dispositivi reali più complessi**.

---

2. Sistemi realizzati e sperimentati

In questa prima fase, abbiamo realizzato diversi progetti distribuiti su **Raspberry Pi**, tra cui:

- led, per l'attivazione visiva di segnali.
- sonar, per la rilevazione della distanza.
- sonarled, che integra i due elementi precedenti.

Tutti questi sistemi utilizzano **MQTT per la comunicazione**, permettendo l'invio di comandi come sonarstart, sonarstop o l'accensione/spegnimento del LED.
Il linguaggio utilizzato per modellare il comportamento dei componenti è stato **QAK**, che ha permesso di generare automaticamente il codice Kotlin da eseguire. I sistemi sono stati testati anche in modalità **headless**, connessi in rete e controllati da remoto.

Utilizzando Mosquitto come server **broker** e gli **attori** per simulare i vari dispositivi e farli comportare in maniera autonoma e reagire per tempo agli eventi (architettura **receiver/iterator**).

Abbiamo quindi sperimentato diversi metodi di comunicazione tra i vari attori.

---

3. Competenze apprese – pratiche e concettuali

Sul piano **pratico**, ho acquisito diverse competenze nuove:

- Modellazione del comportamento con **QAK** e generazione automatica del codice Kotlin.
- Utilizzo di attori come **automi a stati finiti** .
- **Configurazione e utilizzo di un Raspberry Pi in modalità headless**
- Uso di strumenti come **FileZilla** per il trasferimento di file, **Bitvise** per connessioni SSH, ed esecuzione di **script Python** da remoto
  Dal punto di vista **concettuale**, ho interiorizzato:
- Il funzionamento della comunicazione tra attori in un **sistema distribuito**
- I principi base di un'**architettura a eventi**
- L'importanza della **modularità** e della **separazione dei ruoli** tra componenti software

---

4. Perché il Gioco della Vita come primo caso di studio

Il **Gioco della Vita di Conway** è stato scelto come caso di studio perché rappresenta un **sistema complesso emergente**, basato su regole semplici ma molto precise. Questo lo rende perfetto per essere **esteso progressivamente**, introducendo gradualmente nuove funzionalità o livelli di complessità. Inoltre, si presta bene alla **distribuzione delle celle come attori indipendenti**, simulando scenari di comunicazione tra entità autonome.

---

5. Scelte tecnologiche: Java e Spring Boot – obsolete o no?

L'adozione di **Java** e **Spring Boot** si è rivelata coerente con gli obiettivi didattici del corso. Si tratta di tecnologie consolidate, ben documentate e particolarmente adatte per introdurre concetti fondamentali come l'**inversion of control**, le **REST API** e l'architettura a **microservizi**.

Dal punto di vista dello sviluppo moderno, esistono alternative più leggere e recenti — come **Kotlin** (già impiegato indirettamente tramite QAK), **Micronaut** o **Quarkus** — che possono offrire maggiore rapidità e semplicità nella scrittura del codice. Tuttavia, ciò non rende Java e Spring Boot obsolete: sono scelte stabili e strutturalmente chiare, ideali per comprendere e applicare concetti architetturali con solidità.

Inoltre, dato che QAK genera codice in **Kotlin**, e considerando la piena interoperabilità tra Kotlin e Java sia a livello di bytecode che di librerie, l'utilizzo di Java resta del tutto appropriato. Si sarebbe potuto optare direttamente per Kotlin per semplificare ulteriormente lo sviluppo, ma la scelta di Java consente un maggiore controllo esplicito, utile soprattutto in un contesto formativo.

---

6. Aspetti di Ingegneria del Software richiamati

Il caso di studio ha permesso di esplorare diversi concetti fondamentali dell'ingegneria del software:

- **Progettazione modulare e a componenti**
- **Comunicazione tra attori asincrona** (eventi e messaggi)
- **Separazione tra logica applicativa e logica di comunicazione**

- **Testing incrementale e verifica dei prototipi**
- **Evoluzione graduale del sistema e gestione dei requisiti**
  Inoltre, il linguaggio QAK ha facilitato un approccio **modello-centrico** alla progettazione.

---

7. Il sistema sviluppato è un sistema distribuito a microservizi?

Il sistema sviluppato in questa fase ha **caratteristiche di distribuzione**, in quanto i componenti comunicano tramite rete e sono eseguibili su dispositivi separati (come Raspberry e PC).

- ha una gestione indipendente del ciclo di vita per ogni componente
- Ha una struttura di comunicazione con un server
- è **scalabile**

  Le celle come attori possono essere create dinamicamente, e comunicando con il broker, se ne possono creare all'infinito facilmente.

  Il problema sorge però nell'utilizzo delle risorse.

---

8. Ruolo delle librerie 'custom' e della unibo.basicomm23-1.0.jar

Le librerie personalizzate hanno un ruolo fondamentale nel **rendere riutilizzabili e manutenibili** i componenti di comunicazione.
In particolare, la libreria unibo.basicomm23-1.0.jar ha permesso di **astrarre la gestione del protocollo MQTT**, offrendo un'interfaccia uniforme per l'invio e ricezione dei messaggi. Questo approccio consente agli sviluppatori di **focalizzarsi sulla logica applicativa**, evitando la duplicazione di codice tecnico in ogni progetto.

---

9. (Opzionale) Il ruolo del concetto di linguaggio

Il docente ha spesso evocato il ruolo del **linguaggio** come evoluzione fondamentale nello sviluppo software. Più che una questione di sintassi, si tratta di **modi diversi di pensare e modellare i sistemi**.

QAK, ad esempio, è un linguaggio dedicato alla descrizione del comportamento degli attori, e consente di **elevare il livello di astrazione**, migliorando la leggibilità e la coerenza del sistema.
Permette inoltre di automatizzare la scrittura di codice in Kotlin e di Coroutines.