Relazione Fase2-ISS25

Sebastiano Giannitti
Repository GitHub: https://github.com/sebastianogiannitti/iss

Introduzione

Durante la fase iniziale del progetto ConwayGUI, abbiamo analizzato e definito una prima soluzione sfruttando linguaggi noti come JavaScript e Java e implementando vari protocolli di comunicazione per realizzare il gioco *Conway Life*.

Per superare i limiti di uno sviluppo bottom-up, abbiamo adottato un modello di sviluppo top-down. Questo approccio si fonda su un'analisi approfondita del problema e dei requisiti, con l'obiettivo di colmare l'abstraction gap. Il concetto di *abstraction gap* rappresenta la distanza tra le operazioni elementari offerte da un automa o da un linguaggio di programmazione general-purpose e le operazioni astratte di più alto livello necessarie per affrontare efficacemente il problema in esame.

Approccio Top-Down

Attraverso un approccio top-down è possibile:

- Analizzare in modo sistematico il dominio applicativo;
- Progettare architetture e componenti software a partire da una visione d'insieme;
- Selezionare le tecnologie più appropriate o sviluppare soluzioni ad hoc quando necessario.

Tale approccio consente di strutturare lo sviluppo in maniera più razionale ed efficiente, aumentando la qualità complessiva del software prodotto. Tuttavia, è fondamentale concentrarsi sulle fasi iniziali di analisi e progettazione, evitando di procedere direttamente alla scrittura del codice senza una visione chiara e coerente del sistema da realizzare.

In sintesi, il passaggio a un paradigma top-down rappresenta una scelta metodologica che consente di affrontare problemi complessi con maggiore efficacia, riducendo il rischio di soluzioni parziali, non scalabili o difficili da mantenere nel tempo.

QaK

QaK è un linguaggio di modellazione eseguibile progettato per supportare l'analisi e la progettazione di sistemi software distribuiti, in particolare quelli basati sull'architettura ad attori e sul paradigma dei microservizi.

Il termine QaK deriva dalla combinazione di:

- **Q/q**, che sta per "quasi", a indicare che non si tratta di un linguaggio di programmazione general-purpose, ma di un linguaggio pensato specificamente per la modellazione eseguibile dei comportamenti di sistema;

- **aK**, dove la *K* finale fa riferimento a Kotlin, il linguaggio scelto per l'implementazione, in alternativa all'uso della libreria Akka.

Il linguaggio QaK consente la modellazione di sistemi come insiemi di attori, in linea con i seguenti principi:

- Il sistema è composto da attori, ciascuno dei quali è un'entità autonoma che riceve ed elabora messaggi;
- Gli attori comunicano tramite messaggi strutturati (nel nostro caso, di tipo IApplMessage);
- Ogni attore è progettato come un Automa a Stati Finiti (FSM), favorendo una modellazione più intuitiva e rigorosa rispetto a una macchina di Turing generica;
- Gli attori sono organizzati in contesti, che abilitano la comunicazione via rete;
- I contesti possono essere distribuiti su diversi nodi computazionali, sia fisici che virtuali.

Un attore definito in QaK è una specializzazione della classe astratta `ActorBasicFsm.kt`, che a sua volta estende `ActorBasic.kt`, entrambe fornite all'interno dell'infrastruttura QaK (`it.unibo.kactor`). Questa gerarchia astratta semplifica lo sviluppo, poiché fornisce un'architettura coerente per la definizione del comportamento degli attori.

QaK è sviluppato nell'ambito della QaK Software Factory con l'obiettivo di:

- Favorire la progettazione top-down di sistemi distribuiti;
- Facilitare la prototipazione rapida mediante la definizione di modelli eseguibili;
- Guidare il ragionamento sul sistema attraverso strutture comportamentali ben definite e verificabili, come gli automi a stati finiti.

ConwayLife

Durante le lezioni, sono state analizzate diverse versioni del gioco ConwayLife implementate utilizzando il paradigma degli attori fornito da QaK. Inizialmente, il sistema prevedeva un singolo attore con il ruolo di controller, ma successivamente l'architettura è stata evoluta fino a modellare ogni cella come un attore indipendente.

Questa evoluzione ha messo in evidenza la necessità di un'ulteriore analisi del problema, sollevando domande cruciali in merito alla progettazione dell'architettura distribuita del sistema.

Architettura del Sistema

Sarebbe possibile realizzare il sistema senza un orchestratore centrale, demandando tutta la logica di coordinamento alle singole celle. Tuttavia, questa scelta complicherebbe notevolmente lo sviluppo, riducendo la tolleranza ai guasti e rendendo più difficile l'avvio, la sincronizzazione e il controllo del ciclo di vita della simulazione.

L'introduzione di un orchestratore permette invece di:

- Coordinare l’avvio della simulazione (es. attendere la connessione di tutte le celle);
- Gestire esplicitamente il ciclo delle epoche;
- Centralizzare il monitoraggio del sistema;
- Interfacciarsi con un controller esterno (es. tramite GUI) per ricevere comandi come start, stop, clear.

Ogni cella, al momento della connessione, può essere registrata con un nome temporaneo. È l’orchestratore (o il configuratore di sistema) ad assegnare a ogni cella un nome definitivo che rifletta la sua posizione all’interno della griglia logica (es. `cell_2_3`).

Questo approccio consente di:

- Associare direttamente il nome della cella alla topic MQTT;
- Permettere alla cella, conoscendo il proprio nome e la dimensione della griglia, di dedurre autonomamente i vicini (es. da `cell_1_1` si deduce `cell_0_1`, `cell_1_0`, ecc.);
- Eliminare la necessità di un “deus ex machina” che distribuisca configurazioni statiche.

Comportamento della Cella

Una volta assegnato il nome e nota la propria posizione, ogni cella segue questo comportamento durante un'epoca:

- Pubblica il proprio stato (ON/OFF) sulle topic corrispondenti ai propri vicini;
- Riceve i messaggi dai vicini;
- Elabora il nuovo stato sulla base delle regole di Conway;
- Aggiorna la propria visualizzazione (es. LED);
- Notifica all’orchestratore di aver completato il calcolo dell’epoca.

Il comportamento è completamente autonomo e ripetibile ad ogni ciclo, secondo un approccio reattivo e concorrente.

Strategie di Sincronizzazione

Due strategie sono state considerate per la sincronizzazione:

1. **Coreografia con DT (delta time)**: tutte le celle attendono un tempo prefissato (es. 5 secondi) prima di passare all’epoca successiva. Più semplice da realizzare ma sensibile a disallineamenti temporali.
2. **Orchestrazione con segnale di Sync**: l’orchestratore invia un messaggio di sincronizzazione quando tutte le celle hanno completato l’epoca. Più robusta, ma richiede un attore supervisore centrale.

Raspberry Pi

L’analisi del problema finora condotta è finalizzata alla realizzazione di una versione distribuita del gioco ConwayLife, in cui ogni cella viene eseguita su un nodo indipendente — ad esempio un Raspberry Pi — dotato di un LED per indicare visivamente lo stato della cella (accesa/spenta).

In questo contesto, ogni dispositivo fisico rappresenta un attore autonomo in grado di partecipare alla simulazione in modo coordinato, ma indipendente. Oltre a ConwayLife, sono stati analizzati anche altri scenari applicativi, come il Sonar, in cui il nodo Raspberry integra sensori ambientali per acquisire dati dal contesto fisico circostante.

Questi dispositivi sono definiti come **Situated Actors** (Agenti Software Situati):

- Entità software che operano in un ambiente dinamico, percependolo attraverso sensori, reagendo e agendo sulla base del contesto locale.

Questa prospettiva consente di estendere il paradigma degli attori tradizionali verso sistemi più reattivi, autonomi e adattivi, aprendo la strada a una progettazione che tenga conto non solo del flusso di messaggi tra attori, ma anche delle loro interazioni con il mondo fisico.