1. Aggregate Programming

Cos'è

L'aggregate Programming è un nuovo **paradigma di programmazione** per sistemi distribuiti che sfruttano una rete a **topologia dinamica** (i nodi si spostano di continuo e posso sparire ed apparire), è basato su un framework **decentralizzato** (architettura logica), in cui ogni dispositivo è uguale, senza la presenza di un nodo centrale (autorità centralizzata). Questa soluzione permette una robustezza, una scalabilità ed una privacy maggiore anche se la comunicazione è più difficile e potrebbe andare in difficoltà con dispositivi eterogenei.

Cosa ce ne facciamo? **Esempio dei parcheggi**: se devo trovare parcheggio, anziché comunicare con un cloud centralizzato, io posso attenermi alle informazioni raccolte tramite sensori propagate dal mio vicinato e possiamo coordinarci insieme per, anziché lottare per un parcheggio, condividere le informazioni sullo stato dei parcheggi nella zona, potrò dire ai miei vicini "sono qui e vedo due parcheggi liberi"

Dunque, ogni dispositivo **comunicherà con i propri vicini per rimanere aggiornato**, ogni dispositivo può ricevere e inviare messaggi nel suo vicinato e si coordina con gli altri per una certa computazione S.
Per eseguire S, ogni dispositivo ogni tanto si attiva e in modo asincrono effettua un **fire** (fa una computazione**)**:

1) Colleziona dati locali (tramite sensori per esempio)
2) Computa un risultato
3) Invia messaggi in broadcast al proprio vicinato
4) riceve i messaggi in una fase di sleep

L'idea è di avere ogni punto dello spazio occupato da un dispositivo con una struttura dati che evolve attraverso le interazioni locali, i dispositivi ogni tanto si attivano e mandano messaggi ai vicini che aggiornano i valori.

Implementazioni: una per Java (Protelis un DSL, un linguaggio Esterno a java. Però gira su JVM) , una libreria su Scala (Scafi) ed una libreria in C++ (FCPP).
Le librerie "parlano" direttamente con l'hardware, al di sopra delle librerie ci sono delle API che permettono allo sviluppatore di scrivere il programma distribuito.

Proprietà

a) **Self-stabilisation:** La computazione è resiliente ai cambiamenti sia dello stato dei dispositivi che ai cambiamenti della topologia di rete

b) **Eventual consistency**: La computazione non soffre del numero di device e della densità sul territorio. Se aumento il numero di dispositivi o la frequenza di aggiornamento l'accuratezza del risultato può solo migliorare.

c) **Certified error bounds**: posti un numero di parametri (un numero mimino di device, una minima densità, un limite di messaggi perduti...) c'è un bound di errore massimo nei dati. (qualsiasi sia la topologia è garantito che l'errore è sotto una certa soglia).

Concetti fondamentali

Neighbouring value: sono delle coppie [vicino, valore]. In soldoni una tabella che per ogni vicino si segna che valore ha assegnato ad una espressione (tipo nelle righe tengo tutti i vicini e nelle colonne temperatura, umidità, etc)

L'unione dei neighbouring value è il **campo computazionale**. Una struttura che mappa [device, valore] su tutta la rete. Questa struttura è ovviamente continua, non discreta, perché è una media di più valori. (Esempio. Mappa delle temperature, tra due sensori avrò una "sfumatura", una approssimazione come in una partizione di Voronoi).
Esempio temperatura: immaginiamo uno spazio che ha temperatura diversa di spazio in spazio, allora userò la temperatura percepita da ogni dispositivo per identificare se un punto ha più o meno di 20 gradi, attraverso questo creo una approssimazione in cui ogni dispositivo ha un'area approssimata di competenza (poligoni voronoi)

Runtime Verification usando aggregate programming

Abbiamo la necessità di monitorare il sistema in esecuzione, si fa sfruttando due logiche: temporale (LTL e CTL) e logica spaziale

Logica temporale: gli stessi costrutti di Agenti Intelligenti (futuro: X next time, U until, G sempre) (passato: lo scorso istante di tempo ecc…)

Logia spaziale: esprimono proprietà riguardo a valori di variabili in un punto ed ai valori nei punti vicini. Quello che posssiamo fare è identificare delle proprietà che possono valere in certe zone della nostra topologia.

Alcuni costrutti:

- $\Box\ \phi$ (tutti i vicini soddisfano phi)
- $\Diamond\ \phi$(qualche vicino soddifa phi)
- $\alpha\ R\ \beta$ (esiste un percorso in cui vale sempre alpha fino a un punto in cui vale beta)

Esempio:

D = true se dispositivo elettronico
O = true se dispostivio accesso
P = persona

$D \Rightarrow (O \Leftrightarrow \Diamond\ P)$ se sono un dispositivo elettronico sarò acceso se qualche vicino ha percepito delle persone passare

Questa logica è poi traducibile in **field calculus** (linguaggio dell'aggregate programming)

2. SWIFT VS KOTLIN

Somiglianze:

- Entrambi hanno **l'inferenza automatica del tipo** (può anche essere messa manualmente)
- Supporto per evitare il **NullPointerException** (attraverso ? e !)

Differenze:
Strutture: Esattamente come una classe di Kotlin questa può tenere proprietà, metodi, inizializzatori. La vera differenza sta nell'uso di strutture, che al contrario delle classi, permettono di copiare l'intera struttura quando assegnata ad una variabile e non solo il riferimento all'oggetto come in Java, per esempio, e questo evita il **memory leak** (potevo ad esempio modificare variabili private di una classe se ne ho la reference).

Garbage Collector: È un meccanismo che in linguaggi come Java o Kotlin permettono di recuperare memoria dall'Heap che è inutilizzata. Come funziona: si parte dallo stack (dove ci sono i record in esecuzione) si guardano i riferimenti all'Heap e si seguono come fosse un grafo in cui si ricerca il percorso a partire da un nodo, quando un nodo non è raggiunto in alcun modo **si libera in automatico lo spazio di memoria di quel nodo**. Questo permette di liberare il programmatore dal dover deallocare le risorse. **Svantaggio**: processo pesante in termini di risorse di calcolo, memory leak (oggetti che non servono ma che rimangono nello heap. Potrebbero essere deallocato ma il GC non lo sa)

Automatic Reference Counting: Ogni oggetto tiene un intero che segna il numero di variabili che si riferiscono ad esso (**strong reference**), quando il contatore è a 0, lo spazio di memoria è liberato. Questo meccanismo può creare problemi di **cicli di strong reference** (esempio persona e appartamento che se tolto il riferimento rimane comunque la memoria occupata a causa degli strong reference). Per questo è possibile anche fare **weak reference** che non incrementano il counter e vengono rese nil se il counter è 0. (persona –strong-> appartamento)

Inizializzazione: In java i costruttori sono ereditati, ma questo è fonte di errore quando a volte capita che ci si riferisca a proprietà non ancor inizializzate, in swift le sottoclassi non ereditano di default gli inizializzatori.

È sempre presente un **inizializzatore di default** che inizializza con valori di default altrimenti si definisce un **inizializzatore designato** ma se fai questo allora dovrai obbligatoriamente inizializzare tutte quante le proprietà.
Vi sono inoltre gli **inizializzatori convenience**, ovvero degli shortcut per pattern di inizializzazione (obbligatoriamente dovranno chiamare un inizializzatore della stessa classe)
Se si è in presenza di **sottoclasse** questa dovrà per forza chiamare un inizializzatore della sua superclasse immediata.
È possibile implementare un metodo per **deinizializzare**. Per esempio per liberare risorse

Il compilatore effettua 3 controlli:

1. Un inizializzatore designato deve richiamare un inizializzatore della sua superclasse immediata.
2. Un inizializzatore convenience deve chiamare un altro inizializzatore della stessa classe
3. Alla fine di una catena di chiamate ad inizializzatori convinience ci deve essere una chiamata a un inizializzatore designato

3. KOTLIN VS JAVA

Differenze: Singleton, Data Classes, coroutines (per gestire programmi asincroni in modo fluente) e il meccanismo di Null safety (? !)

4. REACT NATIVE

Framework che permette lo sviluppo per Android e IOS, il **codice sorgente è uno solo** e successivamente vengono **mappati i componenti JS con i componenti nativi**, così si usa React per scrivere l'interfaccia grafica.

Basato sull'idea di **componente**: un oggetto con uno stato e delle proprietà. A differenza di altri framework non si basa su una web-view (che introduce latenza), ma chiede direttamente al sistema operativo di **disegnare il Virtual-DOM** (document object model), che genera un albero dei contenuti da renderizzare. Quando aggiungo un componente lo aggiunge al DOM e lo renderizza.

Quando si compila il codice vengono creati 2 pacchetti, uno android uno IOS (come flutter). Il codice è scritto in JS + CSS

PRO : comunità molto attiva, molti componenti open source