Aggregate programming

DISCLAIMER:

Questi appunti fanno cagare, ma meglio questo che niente.

Oltretutto il corso cambia praticamente ogni anno la parte di aggregate programming

Sistemi distribuiti, comunicazioni asincrone e dispositivi non affidabili (guasti, errori di comunicazione).
AC (Aggregate Computing) è un sistema decentralizzato (distribuito) e basato su un paradigma "data-based": cioè I nodi mantengono aggiornata una struttura dati distribuita, non c'è il discorso "query-risposta" ci può essere una query certo, ma la computazione non è "on demand".

Cosa ce ne facciamo? "swarm", una struttura dati sempre aggiornata, dati presi da sensori.
Ad esempio robot nelle warehouse che seguono un percoso, se incontrano un ostacolo devono deviare ma non possono farlo a meno che non siano certi che la deviazione sia "libera" da altri robot ad esempio.
Grazie ad AC ogni robot può dire agli altri "sono qui e sto seguendo questo percorso".

1° tentativo, il cloud. Ma funziona male, perché c'è un device centrale, alcuni device possono disconnettersi. Ci possono sempre essere imprevisti.
Si può pensare di spostare la maggior parte delle computazioni sull'edge.
Esempio, voglio cercare un parcheggio libero, posso chiedere al cloud per sapere dove sono parcheggiati tutti, ma posso anche chiedere a tutte le altre auto a 300 m da me se vedono posti liberi.
Questo risolve anche il problema della privacy.

AC è un modello di programmazione per le proximity based network, una rete a topologia dinamica, i nodi si spostano ed hanno un raggio di comunicazione.
Ogni dispositivo manda broadcast in un certo raggio e la topologia deriva da questi messaggio punto punto.

Come funziona?
Ogni device ha un processo che ogni tanto si attiva "fire" (fa una computazione).
Legge I sensori o i dati in memoria, legge eventuali messaggi dei vicini, computa un risultato, aggiorna I suoi dati ed eventualmente invia messaggi ai vicini (in broadcast).
Per il momento ci sono 3 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 di un programma distribuito:
1. self-stabilisation se una volta che la topologia è "ferma" (i miei vicini sono sempre gli stessi) i valori che vengono computati sono dipendenti solo dai sensori e non dalle perturbazioni della rete.
2.Eventual consistency. La computazione non dipende dal numero di device e dalla densità sul territorio. Se aumento il numero di dispositivi o la frequenza di aggiornamento l'accuratezza del risultato può solo migliorare.

3.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 massimo di errore nei dati. (qualsiasi sia la topologia è garantito che l'errore è sotto una certa soglia).

Un programma AC può essere visto come una funzione f Rete → Rete.
Da una rete di device iniziale specifica come i valori dei dati su ogni device cambiano nel tempo.
Ogni nuovo valore dipende solo da valore dei sensori, valori salvati in memoria e messaggi ricevuti da altri device (tenendo conto del tempo di trasmissione.)
L'idea è ottenere un modello formale di computazione.

Ok il modello formale. Ma quello concreto?
Assunti: ogni dispositivo esegue lo stesso identico programma (magari a seconda del dispositivo si eseguono certe parti ed altre no). Tutti i messaggi sono in broadcast (anche se magari posso inviare RISPOSTE personalizzate ai device che mi hanno contattato).
L'esecuzione è poi la solita: 1 ricevi dati da sensori, memoria e messaggi, 2 computa i dati, invia i risultati agli altri 3 eventualmente fa qualche azione locale 4torna a dormire ricevendo i messaggi che gli arrivano e salvandoseli.
Dobbiamo astrarre dall'idea di "invio e ricezione" del messaggio e dobbiamo garantire che componendo i modelli le proprietà si mantengono (se compongo il programma 1 con il programma 2 le proprietà del nuovo programma sono qeulle di 1 +2, non ne ho perse), insomma un programma funzionale. Quindi possiamo rapprensentare l'esecuzione tramite un albero (stile LFT) e possiamo indicare una precisa esecuzione usando un ramo.
Abbiamo quindi la grammatica seguente:

a value v can be:

$$\texttt{v} ::= \ell \mid \phi \mid (\overline{\texttt{x}})\texttt{=>e} \mid \texttt{b} \mid \texttt{f}$$

- plain old value ℓ (local values)
- neighbouring field value ϕ
- anonymous function $(\overline{\texttt{x}})$=>e
- function name (built-in b or defined f): def f$(\overline{\texttt{x}})${e}

an expression e can be:

$$\texttt{e} ::= \texttt{x} \mid \texttt{v} \mid \texttt{e}(\overline{\texttt{e}}) \mid \texttt{let x = e in e} \mid \texttt{if(e)\{e\} else \{e\}} \mid \texttt{rep(e)\{e\}} \mid \texttt{nbr\{e\}} \mid \texttt{share(e)\{e\}}$$

- variable x (should be bounded in programs)
- value v (no neighbouring ϕ in programs)
- function application e$(\overline{\texttt{e}})$
- let expression let x = e in e
- branching if and other coordination constructs rep, nbr, share (we'll see)

l sono i valori (bool, int, etc).

I 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)
funzioni di qualsiasi tipo (builtin, definite dall'utente o lambda)

l'unione dei neighbouring value è il computational field (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 "voronoi partitioning").
Questo risultato varia in continuazione (device si disconnettono, il valore dei sensori cambia, etc..)

Verifica
Runtime Verification (RV). Monitoriamo il sistema in esecuzione.
Si fa in due modi: logica temporale (LTL e CTL) e logica spaziale.
Logica temporale la sai. Logica spaziale: 15:19
Esprimono proprietà riguardo a valori di variabili in un punto ed ai valori nei punti vicini.

$$\phi ::= \bot \mid \top \mid q \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \Rightarrow \phi) \mid (\phi \Leftrightarrow \phi) \quad \text{logical op.}$$
$$\mid (\Box\,\phi) \mid (\Diamond\,\phi) \mid (\partial\,\phi) \mid (\eth\,\phi) \mid (\partial^{+}\,\phi) \quad \text{local mod.}$$
$$\mid (\phi\,\mathcal{R}\,\phi) \mid (\phi\,\mathcal{T}\,\phi) \mid (\phi\,\mathcal{U}\,\phi) \mid (\mathcal{G}\,\phi) \mid (\mathcal{F}\,\phi) \quad \text{global mod.}$$

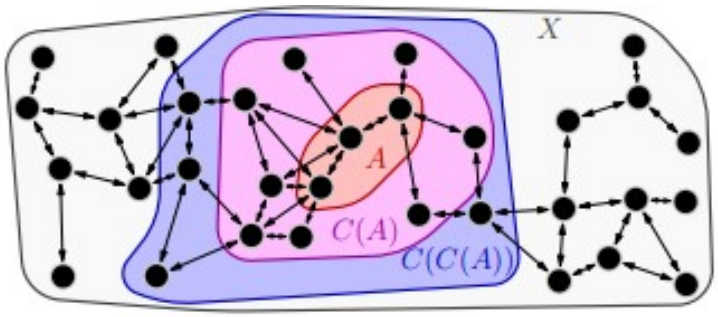
Local modalities

- $\Diamond\,\phi$ (closure) holds at points with some neighbour satisfying ϕ
- $\Box\,\phi$ (interior) holds at points with all neighbours satisfying ϕ
- $\partial\,\phi$ (boundary) holds at points with some (not all) neighbours satisfying ϕ
- $\eth\,\phi$ (interior boundary) holds where ϕ and some neighbour $\neg\phi$
- $\partial^{+}\,\phi$ (closure boundary) holds where $\neg\phi$ and some neighbour ϕ

Global modalities

- $\phi\,\mathcal{R}\,\psi$ (reaches) holds at the start of paths satisfying ϕ ending in ψ
- $\phi\,\mathcal{T}\,\psi$ (touches) like reachability but ϕ may not hold in the ending point
- $\phi\,\mathcal{U}\,\psi$ (surrounded by) holds in areas satisfying ϕ whose neighbours satisfy ψ
- $\mathcal{G}\,\phi$ (everywhere) holds if ϕ is true everywhere
- $\mathcal{F}\,\phi$ (somewhere) holds if ϕ is true somewhere

Nello spazio possiamo identificare dei Closure Space ed interpretare le formule SLCS (Spatial logic of closure spaces) sulla nostra tipologia.
In pratica identifichiamo le "zone" della nostra topologia che rispettano le formule che stiamo valutando.



Ogni costrutto in SLCS può essere tradotto in una espressione Field Calculus