

Tema 1 - Crowdsensing

Adriana Drăghici, Dan Dragomir

Eduard Stăniloiu, Giorgiana Vlăsceanu

adriana.draghici at cs pub ro, dan.dragomir at cs pub ro

eduard.c.staniloiu at gmail com, giorgiana.vlasceanu at gmail com

12.03.2017

1 Obiective

- implementarea unei aplicații concurente folosind thread-uri
- folosirea eficientă a elementelor de sincronizare
- interfațarea cu un alt cod Python (tester-ul)

2 Descriere și cerințe

Un sistem pentru *environmental crowdsensing* folosește dispozitivele utilizatorilor pentru a afla diverși parametri despre mediu. Acesta este implementat folosind o aplicație instalată pe telefoanele utilizatorilor. Aplicația primește de la un server sau chiar de la utilizator cereri de "sensing" de date și salvează sau trimite ca răspuns valorile măsurate de către senzorii implicați. Un posibil scenariu de crowdsensing este cel de *noise mapping* - realizarea de hărți cu poluarea fonică din mediul urban. Device-urile implicate folosesc microfonul pentru a înregistra nivelul de zgomot din jurul lor. Device-urile sunt mobile și își schimbă poziția, astfel măsurând nivelul de poluare fonică din mai multe locuri.

Pentru această temă va trebui să implementați, folosind limbajul Python, o simulare a unui sistem de crowdsensing pentru noise mapping. Sistemul de testare cu care interfațați tema va simula cereri din partea utilizatorilor pentru a vedea nivelul de poluare din anumite zone. Considerăm că la începerea simulării, device-urile dețin câte o valoare a poluării din ziua anterioară pentru anumite zone. La finalul simulării ele pot avea alte valori pentru același set de zone, valori modificate (îmbunătățite) în urma întâlnirilor cu alte device-uri, schimbării de date și execuției de scripturi. Considerăm că prin aplicarea unui algoritm (reprezentat de execuția scriptului) pe datele colectate de mai multe device-uri pentru o anumită zonă, valoarea poluării din acea zonă va fi mai precisă.

Pentru simplificarea simulării durata este împărțită în *time-points* și la un anumit time-point un device poate să se învecineze cu câteva device-uri, iar la următorul time-point cu altele (s-a deplasat în altă zonă, sau a stat pe loc iar vecinii lui s-au mișcat). Tot pentru simplificare device-urile nu vor mai colecta date noi de mediu, ci vor rula algoritmi primiti prin intermediul script-urilor pe datele deja colectate.

3 Detalii pentru implementare

Pentru rezolvarea acestei teme va trebui să completați clasa **Device** cu o implementare corectă a metodelor deja definite în schelet. Pe lângă acestea puteți crea bineînțeles oricâte alte metode/clase/module aveți nevoie în rezolvare. Nu trebuie să modificați clasele tester-ului, trebuie doar să interfațați codul vostru cu acestea. Detaliile metodelor clasei **Device** și ale celorlate clase din infrastructura de testare le găsiți în *docstring*-ul temei, în `doc/index.html`.

Clasa **Device** conține o implementare naivă și nesincronizată care funcționează doar pt un moment de timp (time-point) al operațiilor pe care un device trebuie să le efectueze.

La pornire, device-urile cunosc ce zone (*locations*) au vizitat și ce nivel de poluare era în acestea. Pe parcursul simulării ele se vor plimba și vor comunica cu alte device-uri întâlnite, dacă este cazul. La sfârșitul simulării testerul apelează `get_data` cu fiecare zonă pentru care device-ul a primit date inițial pentru a verifica rezultatele.

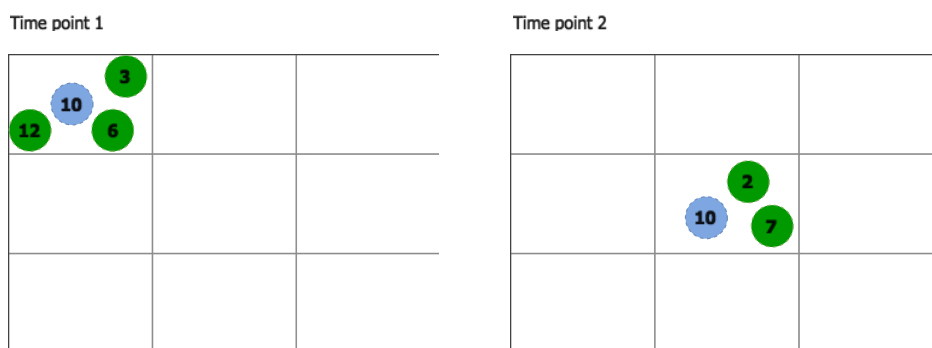


Figure 1: La un moment de timp 1 device-ul 10 a detectat 3 device-uri iar apoi s-a deplasat și la momentul timp 2 a detectat alte device-uri.

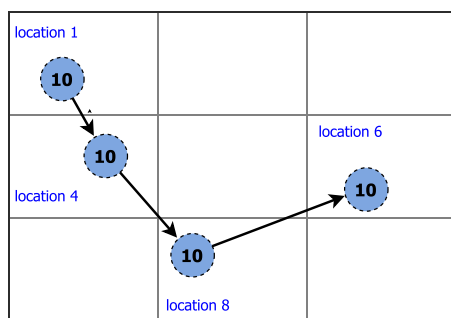


Figure 2: După ce și-a terminat deplasarea, device-ul 10 va stoca valori pentru fiecare zonă vizitată, de exemplu: location1: 30.5 dB; location4: 60 dB; location8: 40 dB; location 6: 45 dB.

Ce sunt scripturile?

Algoritmul executat pe datele de la senzori este încapsulat de clasa **Script**, oferită împreună cu testerul, și care nu trebuie modificată. Rularea scriptului se face prin apelarea metodei `run`. Fiecare script e interesat să prelucreze datele dintr-o zonă și el trebuie rulat la fiecare întâlnire (*encounter*) cu device-uri care dețin date pentru zona de care este interesat scriptul. Metoda `run` primește o listă cu datele device-urilor (inclusiv valoarea device-ului care îl rulează, dacă este cazul), aplică un algoritm și returnează o valoare. Valoarea aceasta reprezintă o îmbunătățire a datelor despre acea zonă și trebuie partajată cu device-urile care au dat

datele de intrare. Fiecare device își înlocuiește valoarea stocată cu noua valoare, care reprezintă o informație mai bună despre zona repectivă.

Cum primesc scripturi?

Pe durata simulării fiecare device poate primi mai multe scripturi pe care să le execute, prin apelarea de către tester a metodei `assign_script`. Device-ul va trebui să ruleze aceste scripturi pe unul sau mai multe threaduri create de el.

Ce face un device când a primit un script?

Un device reține că va trebui să colecteze datele din ziua anterioară pentru zona precizată de către script. La timpii următori, atunci când se întâlnește cu alte device-uri, iar acestea au date pentru zona repectivă, va rula scriptul cu date colectate de la acești vecini și va partaja apoi rezultatul cu ei.

În ce ordine se rulează scripturile?

Prin rularea unui script vom înțelege următoarea secvență de operații: procurarea datelor scriptului, apelarea metodei `run` și salvarea rezultatului. Ordinea de rulare într-un timepoint, de către toate device-urile, nu este relevantă pentru rezultatul final, dar rularea acestora trebuie făcută respectând *consistența secvențială*. Consistența secvențială înseamnă că se poate găsi o ordine în care script-urile au fost rulate. Cu alte cuvinte, sistemul oferă iluzia unei rulări secvențiale a operațiilor, însă în realitate ele sunt făcute în paralel. Pentru a păstra această iluzie trebuie să aveți grijă la atomicitatea unei rulări: citirea, prelucrarea și scrierea să fie făcute atomic pentru un anumit script, limitând însă paralelismul doar acolo unde este necesar.

Cum ia testerul rezultatele de la device-uri?

Sistemul de testare, prin intermediul clasei `Supervisor`, preia datele de la device-uri la finalul simulării prin apelul metodei `get_data` a clasei `Device`. El apelează metoda aceasta pentru fiecare zonă pentru care device-ul a primit date la pornire.

Exemplu

Setup: 3 device-uri cu id-uri 0,1,2. 6 zone posibile.

- Device 0 a măsurat date pentru zonele 1,3,5 și are date pentru acestea la începutul simulării.
- Device 1 a măsurat date pentru zonele 2,4,5 și are date pentru acestea la începutul simulării.
- Device 2 a măsurat date pentru zonele 0,1,3,4 și are date pentru acestea la începutul simulării.
- La momentul t device-ul 0 primește script pentru zona 5.
- La momentul $t+1$ device-ul 0 întâlnește device-ul 2 și nu se întâmplă nimic deoarece device-ul 2 nu deține date interesante pentru script.
- La momentul $t+2$ device-ul 0 întâlnește device-ul 1. Device-ul 0 are următorul comportament:
 1. cere valoarea pentru zona 5 de la device-ul 1
 2. aplică scriptul pentru valoarea primită de la 1 și cea stocată de el pentru zona 5
 3. își înlocuiește propria valoare pentru zona 5 și îi trimite și lui 1 noua valoare
- ...
- Se încheie simularea. Tester-ul verifică valorile fiecărui device.

Alte detalii:

- terminologie folosită în cod: `location` - loc vizitat de un device, zonă; `encounter` - întâlnirea unui device cu un alt device-uri; `script` - algoritm aplicat pe datele dintr-o zonă
- un device se poate întâlni cu un alt device în mai multe momente de timp

- zona în care se întâlnesc două device-uri în timpul simulării nu este relevantă pentru script, ci faptul ca device-ul întâlnit are date pentru zona cerută.
- atenție, învecinarea nu este neapărat simetrică: dacă un device $D1$ detectează un alt device $D2$ la momentul t în locul l nu este garantat că și $D2$ îl detectează pe $D1$ - o constrângere a tester-ului.

4 Notare

Tema va fi verificată automat, folosind infrastructura de testare, pe baza unor teste definite în directorul `tests`. Testele forțează diferite intercalări ale execuției și conțin mai multe rulări pentru a maximiza șansele de a detecta eventuale bug-uri de sincronizare. Există de asemenea un timeout, specific fiecărui test, în care trebuie să se încadreze execuția fiecărei iterații a testului respectiv. Cu toate acestea, la fel ca la orice test, rularea fără erori nu garantează lipsa bug-urilor. În schimb, eșecul unui test indică existența a cel puțin unui bug în implementare.

Punctajul dat unui test depinde de toate iterațiile acestuia, deci pot exista punctaje parțiale. De exemplu, dacă testul 1 pică o dată din 4 iterații, atunci se primește 75% din punctajul aferente testului.

Infrastructura de testare, scheletul clasei `Device`, precum și documentația API-ului poate fi descărcată de pe site-ul cursului ¹.

Tema se va implementa în **Python 2.7**. Arhiva temei (fișier .zip) va fi uploadată pe site-ul cursului și trebuie să conțină:

- fișierul `device.py` cu implementarea clasei `Device`
- alte surse .py folosite de soluția voastră (nu includeți fișierele infrastructurii de testare)
- fișierul README cu detaliile implementării temei (poate fi în engleză)

Notarea va consta în 100pct acordate egale între teste (cu excepția testului 0 care nu valorează nimic). Depuneri posibile sunt:

- folosirea busy-waiting sau a “sincronizării” prin sleep (între $-10pct$ și $-50pct$ în funcție de gravitate)
- folosirea incorectă a variabilelor de sincronizare (ex: lock care nu protejează toate accesele la o variabilă partajată, notificări care se pot pierde) ($-2pct$)
- prezența print-urilor de debug (maxim $-10pct$ în funcție de gravitate)
- folosirea lock-urilor globale ($-10pct$)
- folosirea variabilelor globale ($-5pct$)
- închiderea și recrearea thread-urilor pe parcursul algoritmului ($-10pct$)
- crearea unui număr mare de thread-uri per device ($-10pct$)
- folosirea inutilă a variabilelor de sincronizare (ex: se protejează operații care sunt deja thread-safe) ($-5pct$)
- alte ineficiențe (ex: creare obiecte inutile, alocare obiecte mai mari decât e necesar, căutări liniare în locul indexării etc.) ($-5pct$)
- lipsa organizării codului, implementare încălțită și nemodulară, cod duplicat, funcții foarte lungi (între $-1pct$ și $-5pct$ în funcție de gravitate)
- cod înghesuit/ilizibil, inconsistența stilului - vedeți secțiunea *Pylint*
 - pentru code-style recomandăm ghidul oficial <https://www.python.org/dev/peps/pep-0008/>

¹http://cs.curs.pub.ro/2016/draftfile.php/3021/user/draft/371682520/tema1_skel.zip

- cod comentat/nefolosit ($-1pct$)
- lipsa comentariilor utile din cod ($-5pct$)
- fișier README sumar (până la $-10pct$)
- alte situații nespecificate, dar considerate inadecvate având în vedere obiectivele temei; în special situațiile de modificare a interfeței oferite, accesare a variabilelor interne ale testerului sau folosirea informațiilor despre implementarea metodei `run` a script-urilor

Temele vor fi testate împotriva plagiatului. Orice tentativă de copiere va fi depunctată conform [regulamentului](#).

Pylint

Vom testa sursele voastre cu [pylint](#) configurat conform fișierului *pylintrc* din arhiva temei. Atenție, *run* *pylint* doar pe modulele scrise de voi, nu și pe cele ale testerului.

Deoarece apar diferențe de scor între versiuni diferite de pylint, vom testa temele doar cu ultima versiune, **1.6.5**. Vă recomandăm să o folosiți și voi tot pe aceasta.

Acordăm un **bonus** de până la 10% din punctajul temei dacă verificarea făcută cu pylint vă dă un scor mai mare sau egal de 9. Bonusul va fi proporțional cu scorul oferit de pylint.

Vom face depunctări de până la $-5pct$ dacă verificarea făcută cu pylint vă dă un scor mai mic de 8.

5 Precizări

- pot exista depunctări mai mari decât este specificat în secțiunea *Notare* pentru implementări care nu respectă obiectivele temei și pentru situații care nu sunt acoperite în mod automat de către sistemul de testare
- implementarea și folosirea API-ului oferit este obligatorie
- toate operațiile făcute de către un device (execuția scriptului, comunicare cu alte device-uri) trebuie făcute din thread-uri proprii device-ului; atenție, metoda `assign_script` este apelată de tester de pe thread-ul său
- avansarea timepoint-ului trebuie să fie făcută în lock-step de către toate device-urile; în orice moment de timp nu trebuie să existe o diferență mai mare de 1 între timepoint-urile la care se află oricare 2 device-uri
- pentru încadrare în timp, tema nu necesită optimizarea algoritmului, ci folosirea corectă a operațiilor blocante ale elementelor de sincronizare, paralelizarea pe mai multe thread-uri etc; cu toate acestea, folosirea unor operații ineficiente când alternativele eficiente sunt ușor disponibile va fi penalizată (ex: liste în loc de dicționare, parcurgere liniară în loc de indexare etc.)
- considerați că device-uri sunt reprezentate de telefoane mobile cu procesoare [Snapdragon 625](#) (facem simularea pt viitoarele telefoane Samsung Galaxy S8 :-). Aveți grijă la oversubscription (când faceți mai multe thread-uri decât core-uri) pentru că unele teste vor verifica acest lucru.
- bug-urile de sincronizare, prin natura lor sunt nedeterministe; o temă care conține astfel de bug-uri poate obține punctaje diferite la rulări succesive; în acest caz punctajul temei va fi cel dat de tester în momentul corectării
- recomandăm testarea temei în cât mai multe situații de load al sistemului și pe cât mai multe sisteme pentru a descoperi bug-urile de sincronizare; cluster-ul oferă câteva tipuri de sisteme pe care puteți face testarea