Tema 1 - Crowdsensing

- Deadline soft: 25 martie 2019, ora 23:55
- Deadline hard: 1 aprilie 2019, ora 23:55

Obiective

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

Descriere

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 hărti 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 măsurand nivelul de poluare fonică din mai multe locuri.

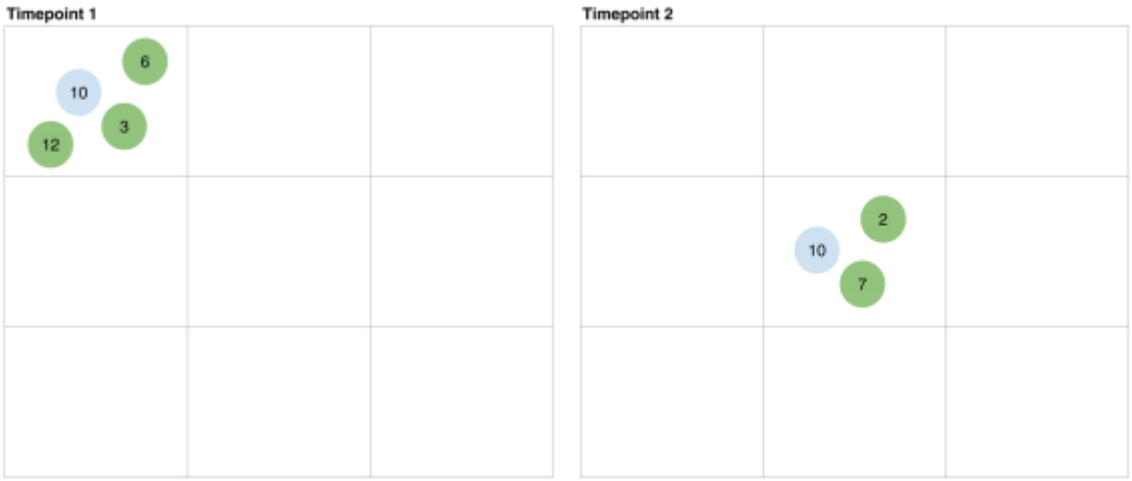
Pentru această temă va trebui să implementați, folosind limbajul Python, o *simulare a unui sistem de crowdsensing pentru noise mapping*. Infrastructura de testare pornește în cadrul fiecărui test un număr de device-uri și le atribuie scripturi (cereri despre poluarea din anumite zone). Va trebui să implementați comportamentul concurent al deviceurilor.

Sumarul acțiunilor desfășurate în timpul unei simulări (test):

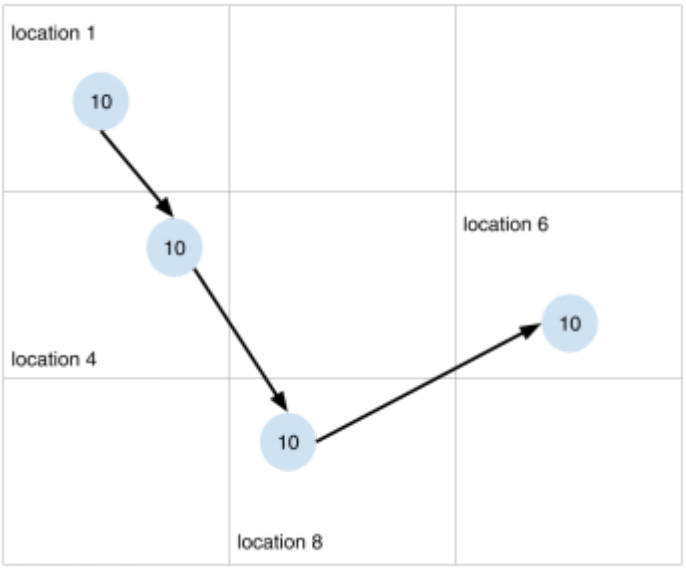
1. La începerea simulării, device-urile dețin câte o valoare a poluării din ziua anterioară pentru anumite zone
2. Sistemul de testare cu care interfatati codul device-urilor va simula cereri din partea utilizatorilor pentru a vedea nivelul de poluare din anumite zone
3. Device-urile se plimbă pe hartă, iar când întâlnesc alte device-uri care conțin date despre zonele cerute în scripturile primite, aplică algoritmul scriptului pe datele respective și datele reținute de ele, actualizând apoi datele cu valorile rezultate în urma algoritmului
4. La finalul simulării device-urile pot avea alte valori pentru același set de zone, valori modificate (imbunătătite) 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 impărtită în **timepoints** și la un anumit timepoint un device poate să se învecineze cu câteva device-uri, iar la următorul timepoint 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 folosi pentru tratarea cererilor (scripturilor) datele deja colectate.

Exemplu:



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.



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.

Detalii implementare

Pentru rezolvarea acestei teme va trebui să completați clasa **`Device`** cu o implementare corectă a metodelor deja definite în schelet. Pe langă acestea puteți crea bineînțeles oricâte alte metode/clase/module aveți nevoie în rezolvare. Detaliile metodelor clasei `Device` și ale celorlate clase din infrastructura de testare le găsiti în **docstring-ul temei**, în `doc/index.html`.

Clasa `Device` conține o implementare naivă și nesincronizată care functionează doar pt un moment de timp (timepoint) al operațiilor pe care un device trebuie să le efectueze.

Terminologie folosită în cod:

- *location* - loc vizitat de un device, zonă;
- *encounter* - întâlnirea unui device cu un alte device-uri;
- *script* - algoritm aplicat pe datele dintr-o zonă

Nu trebuie să modificați antetele metodelor clasei `Device` oferite în schelet. Acestea sunt apelate de către infrastructura de testare.

Comportamentul device-urilor

- 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.
- **Comunicarea cu alte device-uri** presupune apelarea metodelor clasei `Device` pentru a obține/seta o valoare (`lock_data`, `set_data`)
- 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 zonă 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

Exemplu

Setup: 3 device-uri cu id-urile 0, 1, 2 și 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 intamplă 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 device-ul 1 și cea stocată de el pentru zona 5
    3. își înlocuiește propria valoare pentru zona 5 și îi trimite și device-ului 1 noua valoare
- …
- Se încheie simularea. Tester-ul verifică valorile fiecărui device.

Multithreading

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.

Numărul de threaduri trebuie limitat, nu este eficient să se creeze câte un thread pentru fiecare script. Considerați că device-urile sunt reprezentate de telefoane mobile cu procesoare Snapdragon 625 [https://en.wikipedia.org/wiki/List_of_Qualcomm_Snapdragon_devices#Snapdragon_617_and_625], având 8 core-uri, limitați astfel la 8 numărul de threaduri al fiecărui device. Aveți grijă la oversubscription (când faceți mai multe thread-uri decât core-uri) pentru că unele teste vor verifică acest lucru.

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

Într-un timepoint, datele unui device pot fi accesate de către threadurile mai multor device-uri. Din această cauză trebui să protejați accesul concurent la aceste date.

Puteți folosi oricare dintre variabilele de sincronizare studiate la laboratoarele 2 și 3: Lock [https://docs.python.org/2/library/threading.html#lock-objects], semaphore [https://docs.python.org/2/library/threading.html#semaphore-objects], Condition [https://docs.python.org/2/library/threading.html#condition-objects], Event [https://docs.python.org/2/library/threading.html#event-objects], Queue [https://docs.python.org/2/library/queue.html]. Puteți folosi și modulul `barrier` din scheletul laboratorului 3.

Interacțiunea cu sistemul de testare

Componentele sistemului de testare:

- **`Script`** - clasa ce reprezintă algoritmul aplicat pe datele dintr-o anumită zonă

    - `run` - singura metodă pe care trebuie să o apelați, restul (cu numele începând cu __ sunt considerate private testerului)
- **`supervisor.Runtime`** - oferă device-urile vecine
    - `get_neighbours` - apelată de către un device pentru a afla cu cine se învecinează în timepoint-ul curent
- **`Supervisor`** - folosit pentru simulare, nu interacționați cu acesta!
- **`Tester`** - controlează execuția testelor, nu interacționați cu acesta!
- modulul **`test`** - folosit pentru reprezentarea testelor, nu interacționați cu acesta!

Ce sunt scripturile?

Algoritmul executat pe datele de la senzori este încapsulat de clasa `Script`, oferită impreună 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 imbunătățire a datelor despre acea zonă și trebuie partajată cu device-urile care au dat datele de intrare. Fiecare device își înlocuiește valoarea stocată cu nouă 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 rezutatul cu ei.
Device-urile vor reține și stochează scripturile pe toată durata simulării.

În ce ordine se rulează scripturile?

Prin rularea unui script vom înțelege următoarea secvență de operații:

- procurarea datelor scriptului,
- apelarea metodei `run`
- 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 știe un device în ce locație se află în timepoint-ul curent?

Nu este relevantă locația curentă, ci device-urile cu care se învecinează în timepoint-ul curent. Superviser-ul oferă prin intermediul metodei `get_neighbours` această listă de vecini.

Cum știe un device că timepoint-ul curent este finalizat?

La sfârșitului fiecărui timepoint supervisor-ul trimite tuturor device-urilor câte un script cu valoarea `None` prin intermediul metodei `assign_script`. După acest moment se garantează că supervisor-ul nu va trimite alte script-uri până la timepoint-ul următor.

Toate device-urile trebuie să termine de executat script-urile din timepoint-ul curent pentru a putea trece la următorul timepoint.

Cum este semnalizat sfârşitul simulării?

Apelul `get_neighbours` va întoarce `None` când device-ul a ajuns la sfârșitul simulării. Această funcționalitate este prezentă și în scheletul temei.

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.

Nu trebuie să modificați clasele tester-ului, trebuie doar să interfațați codul vostru cu acestea. Scheletul de cod vă oferă documentația tester-ului, este necesară citirea acesteia pentru a afla detalii despre metodele cu care trebuie să interactionati. Tratați modulele tester-ului ca un pe black-box, nu trebuie să analizați codul lor.

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 execuţia fiecărei iterații a testului respectiv. Scopul timeout-ului nu este să verifice performanța, ci corectitudinea (testele sunt gândite să treacă și pe sisteme mai puțin performante). Cu toate acestea, 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.

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

- fişierul `device.py` cu implementarea clasei `Device`
- alte surse `.py` folosite de soluţia voastră (nu includeţi fişierele infrastructurii de testare)
- fişierul README cu detaliile implementării temei (poate fi în engleză)

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

- folosirea busy-waiting sau a “sincronizării” prin sleep (între -10pct și $-50 pct 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) (-2 pct)
- prezența print-urilor de debug (maxim -10 pct în funcție de gravitate)
- folosirea lock-urilor globale (-10 pct)
- folosirea variabilelor globale/statice (-5 pct)
    - Variabilele statice pot fi folosite doar pentru constante
- închiderea și recreerea thread-urilor pe parcursul algoritmului (-10 pct)
- crearea unui număr mare de thread-uri per device (-10 pct)
- folosirea inutilă a variabilelor de sincronizare (ex: se protejează operații care sunt deja thread-safe) (-5 pct)
- alte ineficiențe (ex: creare obiecte inutile, alocare obiecte mai mari decât e necesar, căutări liniare în locul indexării etc.) (-5 pct)
- lipsa organizării codului, implementare încâlcită și nemodulară, cod duplicat, funcții foarte lungi (între -1pct și -5 pct în funcție de gravitate)
- cod înghesuit/ilizibil, inconsistenţa stilului - vedeți secțiunea Pylint
    - pentru code-style recomandăm ghidul oficial PEP-8 [https://www.python.org/dev/peps/pep-0008/]
- cod comentat/nefolosit (-1 pct)
- lipsa comentariilor utile din cod (-5 pct)
- fişier README sumar (până la -5 pct)
- alte situaţii nespecificate, dar considerate inadecvate având în vedere obiectivele temei; in 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 [https://www.pylint.org/] configurat conform fișierului **`pylintrc`** din arhiva temei. Atenție, rulăm 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, **1.8.3**. 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.

Observații

- Pot exista depunctări mai mari decât este specificat în secţiunea Notare pentru implementări care nu respectă obiectivele temei și pentru situatii care nu sunt acoperite în mod automat de către sistemul de testare
- Implementarea şi folosirea API-ul oferit este obligatorie
- 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.)
- Bug-urile de sincronizare, prin natura lor sunt nedeterministe; o temă care conţine 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

Resurse

- PDF
- tema1-skel.zip - schelet de cod, tester, documentație cod
- Autori temă: Adriana Drăghici [mailto:adriana.draghici@cs.pub.ro], Dan Dragomir [mailto:dan.dragomir@cs.pub.ro]
- Responsabili: Mihai Cristian Volmer [mailto:mihaivolmer@gmail.com], Giorgiana Vlăsceanu [mailto:giorgiana.vlasceanu@cs.pub.ro], Silvia Stegaru [mailto:silvia.stegaru@cs.pub.ro], Eduard Stăniloiu [mailto:eduard.staniloiu@cs.pub.ro]

tema1.txt · Last modified: 2019/03/23 15:42 by mihai.volmer