Chiara Bruschi
Davide Ferrari
Federico Motta
Filippo Muzzini
Davide Paganelli
Davide Sapienza

****
LA GESTIONE DEGLI EVENTI IN AESOP-ACP
****

1 - DESCRIZIONE AD ALTO LIVELLO

Per entrare maggiormente nel dettaglio della dinamica relativa alla gestione degli eventi nel simulatore descritti dallo schema prodotto, bisogna innanzitutto segnalare come tutto questo flusso parta dalle operazioni eseguite dal Thread "Simulator".

Come qualsiasi Thread in Python, l'invocazione del vero e proprio lavoro che deve svolgere un generico Thread avviene per mezzo della funzione "start()" (presente nel file "startsim.py"), la quale invocherà direttamente l'opportuna funzione "run" (funzione della classe "Simulator" presente nel file acp.py) che il programmatore deve aver pianificato con il lavoro specifico di tale Thread. Quest'ultima funzione, adibita di fatto a far partire il simulatore, non fa altro che lanciare una sequenza ciclica di operazioni (ciclo while) che terminerà per una qualche condizione di uscita, ovvero la possibile assenza di eventi nella coda oppure il raggiungimento del limite temporale massimo di lavoro (parametro "until").

Oltre ad operazioni di gestione dello stato del simulatore (es. funzione "resume()" e gestione dei monitor), la funzione "goNextEvent" (anch'essa funzione della classe "Simulator") risulta essere quella maggiormente significativa.
Essa stessa è infatti in grado di far terminare il ciclo while per mezzo del suo valore di ritorno (nel caso sia True) una volta incontrato una delle condizioni di arresto indicate poc'anzi.
In suddetta funzione, inoltre, vengono effettuati diversi task: estrazione dell'evento dalla coda globale con priorità ("pq") del simulatore, gestione dei tempi associati ad ogni evento, aggiornamento della graphical user interface (gui) e soprattutto la gestione dei due principali eventi potenzialmente estraibili dalla coda globale del simulatore: ScheduleEvent e MessageEvent.

I MessageEvent sono eventi di base che possono essere sub-classati o sostituiti da qualsiasi classe in grado di assumere il loro ruolo; è necessario che siano accompagnati da una tupla del tipo <’action-name’: param> in cui il parametro può essere di ogni tipo desiderato. Questo tipo di messaggio è quello che effettivamente arriva alla coda locale di ogni agente e che descrive le operazioni che devono essere svolte.

Gli ScheduleEvent invece sono eventi particolari associati ad un agente che invocherà il metodo associato per la creazione dei diversi MessageEvent. Caso

tipico: l’invocazione dell’orderDistiller che inserisce nella coda con priorità i MessageEvent destinati agli altri agenti. Lo ScheduleEvent può essere re-inserito nella coda.

Nel caso in cui l'evento estratto dalla coda globale del simulatore sia uno ScheduleEvent, si avrà la chiamata del metodo "__call__" della classe "ScheduleEvent": invocato per mezzo della sintassi "ev()" (dove "ev" è un'istanza di suddetta classe).
All'interno del sopracitato metodo "__call__" avviene: l'invocazione dello specifico metodo dell'agente associato a tale evento, la chiamata alla funzione "nextCallAt", la rischedulazione di un nuovo ScheduleEvent identico allo stesso (qualora debba essere rischedulato in futuro) e l'aggiunta di quest'ultimo nuovamente alla coda globale del simulatore.

Questo meccanismo ciclico di rischedulazione deve avere un inizio: la prima istanza di ogni scheduleEvent avviene in fase di inizializzazione del simulatore, in particolare avviene per mezzo del parsing delle regole all'interno del file xlsx.

Dal codice si evince che gli scheduleEvent hanno la funzione di scatenare, in un determinato tempo, un'azione associata ad un agente; tale azione (rappresentata dal metodo) è dipendente dall'implementazione dell'agente; un'azione tipica è quella di generare i MessageEvent da inviare agli altri agenti.
Per fare un esempio concreto, uno ScheduleEvent con metodo "DoOrder" (ottenuto dal parsing del foglio "production" del file "schedule_sec.xlsx" dell'esempio "embroidery") chiamerà suddetto metodo (metodo di un tipo particolare di agente di produzione, es. "OrderDistiller"), quest'ultimo creerà un ordine, invocherà la funzione "dispatchOrder" per assegnare l'ordine ad uno specifico agente in grado di poterlo evadere, creerà un evento MessageEvent associato a tale ordine e all'agente di destinazione, e, per mezzo della funzione "talkTo", aggiungerà l'evento appena creato alla coda globale del simulatore.

Da notare come lo specifico metodo degli eventi ScheduleEvent (eventi presi direttamente dal file "xlsx") sia anche rappresentante del comportamento dell'agente stesso: un metodo generico (es "doOrder") rappresenta un agente passivo (es. PRagents), in grado solamente di effettuare una risposta dopo la sua avvenuta sollecitazione da un agente esterno, mentre il metodo specifico "handleBehavior" rappresenta un agente attivo (es. SCagents), in grado di agire per mezzo di un comportamento autonomo.
All'interno della funzione "handleBehavior", infatti, avverrà l'invocazione del metodo "nextWakeup", la quale si occupa di decidere il tempo in cui l'agente possa attivamente rischedulare se stesso.
Tale valore temporale è dipendente dal valore del tempo corrente del simulatore (variabile "whereNow"), dal tempo di inizio del lavoro ("fromTime") e dal tempo necessario per il lavoro ("every").
In particolare se si è al tempo 0 e tocca a tale agente, allora si darà un valore pari ad "every" (tempo in cui finirà il lavoro che doveva fare), se al tempo corrente non tocca a lui, allora si darà un valore pari al suo tempo richiesto di inizio ("fromTime"), altrimenti si setta tale valore attraverso una distribuzione gaussiana centrata sul campo "every" e con una particolare varianza.

Dopo di ciò, sempre all'interno della funzione "handleBehavior", avverrà la

consueta creazione di un evento ScheduleEvent associato all'agente stesso e
con metodo "handleBehavior" in modo da rischedulare la stessa azione;
successivamente si avrà anche la sua aggiunta alla coda globale del simulatore.
Da notare è il fatto che in base al valore temporale, sopracitato, che si
otterrà, dipenderà la successiva estrazione del relativo evento; ciò è
dovuto dal fatto che la priorità associata alla coda globale del simulatore
detterà, proprio in base a tali valori temporali, l'ordinamento degli eventi che contiene.

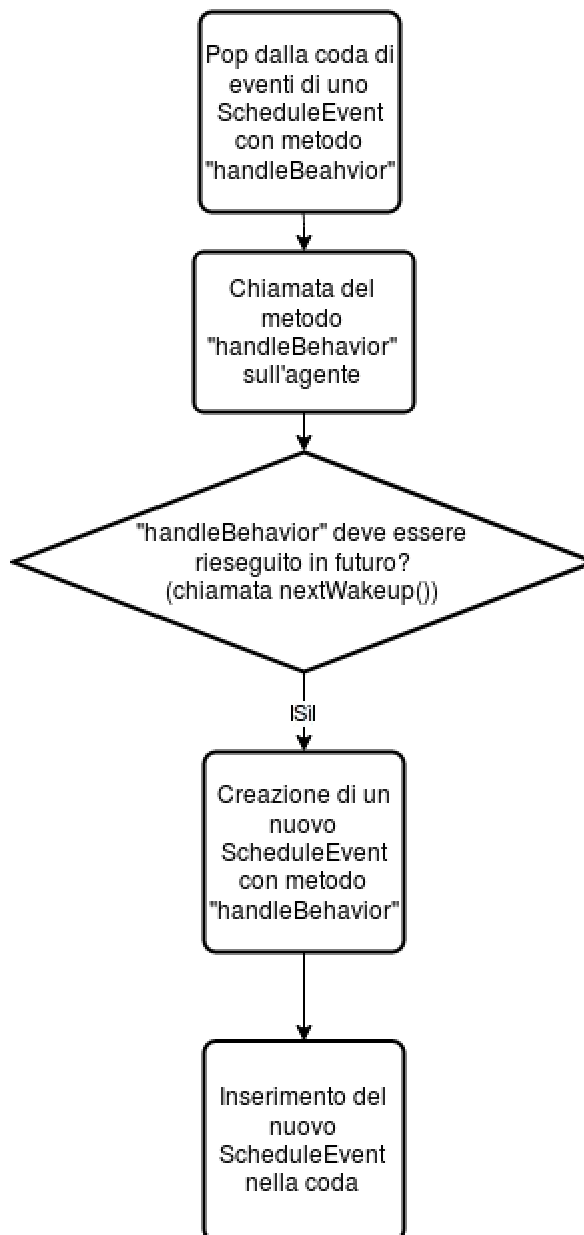
Ritornando alla funzione "__call__" e in particolare a "nextCallAt" si segnala
come tale metodo serva solamente per decidere il tempo in cui dovrà essere
estratto dalla coda globale del simulatore l'evento ScheduleEvent in esame.
In particolare tale valore temporale assumerà il numero indicato nel campo
"every" del file "xlsx" qualora sia entro il limite massimo (parametro
"until"), oppure -1.

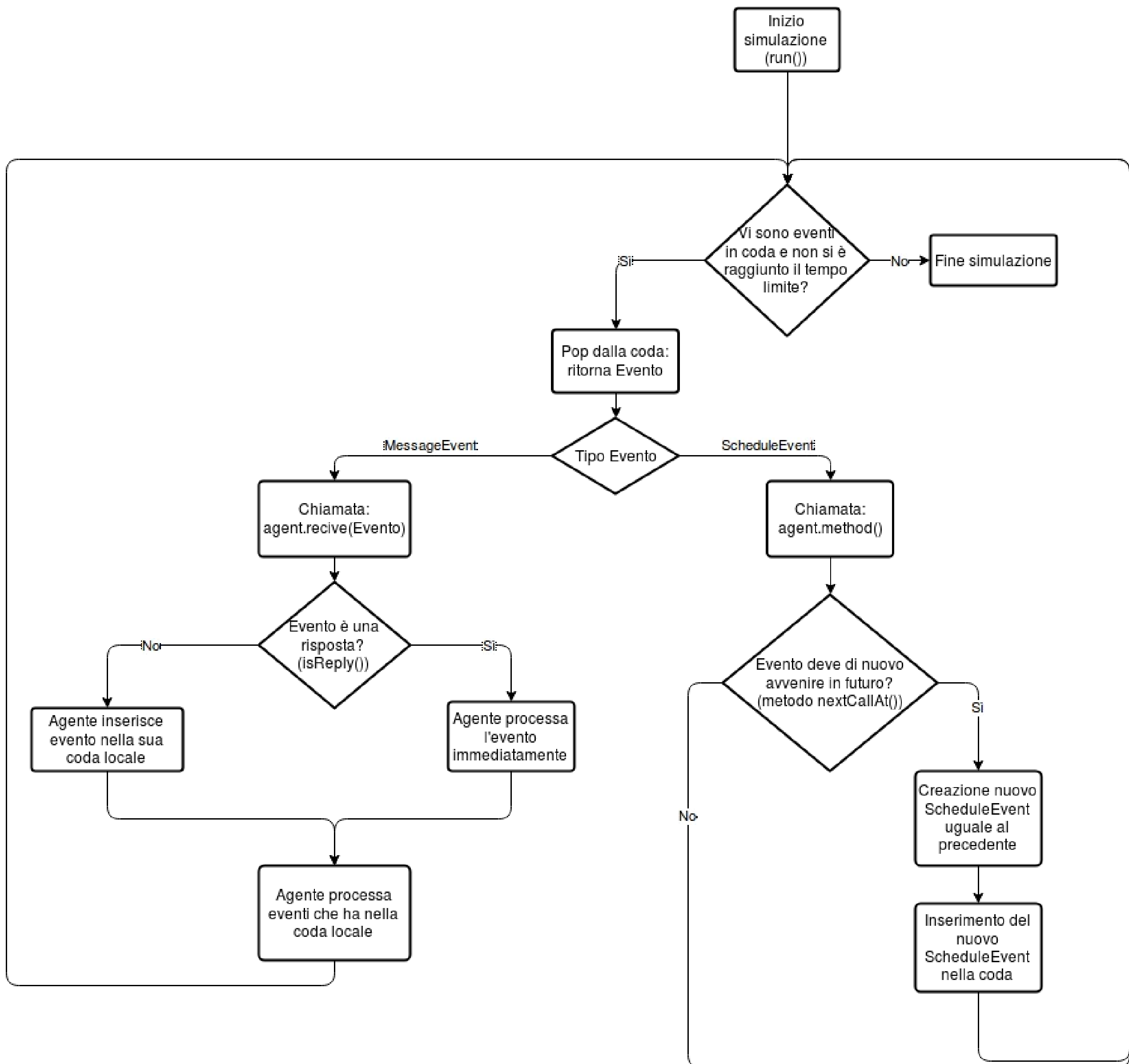
Qualora invece l'evento estratto dalla coda globale del simulatore sia un
MessageEvent (evento generabile solamente da altri ScheduleEvent e non in modo
esterno dal file "xlsx" come per gli ScheduleEvent, e quindi rappresentante di
fatto un ordine di esecuzione di un agente ad un altro) avverrà l'invocazione
del metodo "receive" relativo ad un agente.
Nel caso in cui il parametro "isReplay", sia posto a "True" l'evento verrà
direttamente processato, altrimenti (parametro a False) si passerà allo step
preliminare di aggiunta di suddetto evento alla coda locale dell'agente e poi
alla successiva esecuzione di tutti gli eventi in essa presenti per mezzo della
stessa funzione "process" valida anche per la condizione precedente.
Un utilizzo tipico del parametro "isReply" è l'auto-notifica di un agente per
segnalare a se stesso la fine di un compito.

Ancora una volta tale funzione sarà specifica per ogni agente, ma per fare un
esempio concreto, la classe "Unit" del file "PRagents" sfrutta il parametro
isReply dei MessageEvent per auto-notificarsi i vari step (es.
"procurement_step") associati a tale evento e, nel caso tale unità di
produzione non sia eccessivamente occupata in base alla sua specifica
disponibilità, ad eseguirli.

Riprendendo la struttura finora descritta, si nota come la dinamica degli
eventi, in particolare di quelli ScheduleEvent, venga gestita da un codice
altamente trasparente e flessibile, in quanto in esso si noterà solamente la
seguente riga all'interno della funzione "__call__" della classe ScheduleEvent:
"getattr(self.instance, self.methodName)(*self.args, **self.kwargs)".
Tale istruzione serve per dire che si invocherà, su una determinata istanza di
un evento ScheduleEvent, un generico "methodName" che a runtime si tradurrà in
un metodo specifico (es. "DoOrder", "handelBehavior") di un particolare agente.
A titolo conclusivo si nota anche come il clock che scandisce il tempo di
estrazione degli eventi dalla coda globale del simulatore non necessariamente
rispetti l'ordine temporale scandito da istanti successivi di tempo, ma possa
fare salti temporali atti a raggiungere il tempo di schedulazione del prossimo
evento disponibile in coda.
Tale accorgimento risulta particolarmente efficiente rispetto agli strumenti
sincroni tradizionali che sprecano potenzialmente tempo nel scandire istanti
temporali in cui nessun evento verrà di fatto schedulato.

Parsing file xlsx
Creazione ScheduleEvent relativi al foglio schedule
Aggiunta ScheduleEvent alla coda


Pop dalla coda di eventi di uno ScheduleEvent con metodo "handleBeahvior"
Chiamata del metodo "handleBehavior" sull'agente
"handleBehavior" deve essere rieseguito in futuro? (chiamata nextWakeup())
lSil
Creazione di un nuovo ScheduleEvent con metodo "handleBehavior"
Inserimento del nuovo ScheduleEvent nella coda



1 - DESCRIZIONE A PIU' BASSO LIVELLO

Dopo l’inizializzazione del simulatore (“sim.initialize()” nel file “startsim.py”) viene chiamato sulla rispettiva istanza il metodo “start()”. Tale invocazione scatena indirettamente l’esecuzione del metodo “run()”

di “Simulator”; seguono i dettagli tecnici che spiegano tale avvenimento.

La classe “Simulator” deriva dal tipo di dato “Thread”, messo a disposizione dal linguaggio Python, e ne eredita i metodi. Tra questi ci sono “start()” e “run()”, i quali possono essere invocati su qualsiasi istanza di “Simulator” (come ad esempio “sim”).

Il metodo “start()” ha il compito di fare richiesta al sistema operativo della creazione di un nuovo flusso di esecuzione (detto thread) che possa essere schedulato indipendentemente dal flusso di esecuzione corrente (il thread su cui sta eseguendo la funzione “main()”) o, ad esempio, su un altro processore se disponiamo di una CPU multi-core.

Dopo aver creato un nuovo thread, il metodo “start()” gli assegna come lavoro da svolgere il codice contenuto nel metodo “run()”. Quest’ultimo non possiede, di default, compiti specifici nell’implementazione della classe “Thread”: occorre quindi farne l’override (cioè ridefinirlo) per assegnargli il lavoro che intendiamo svolgere in parallelo al thread della funzione “main()”.
Per questo motivo nella classe “Simulator” (file “acp.py”) è presente un’implementazione del metodo “run()” (o meglio, una ridefinizione). In questo modo, il thread che viene creato chiamando “sim.start()” potrà eseguire le istruzioni definite in “run()”.

Nota 1: da questo punto in poi con “thread principale” si intenderà il thread del “Simulator” e non quello del “main()”

Nota 2: con “sezione critica” si intende un blocco di codice condiviso tra tutti i thread che deve essere eseguito da solo uno di essi alla volta; se vi accedessero tutti senza policy di accesso, altererebbero lo stato della memoria (ad esempio scrivendo contemporaneamente nella stessa area e producendo quindi uno stato inconsistente per il programma).
Metaforicamente si può pensare al blocco di codice come una stanza dove siano presenti un solo foglio ed una sola penna e dove si possa entrare solamente uno per volta, essendo disponibile un’unica chiave per poter aprire la porta. Se le chiavi fossero più di una o la porta fosse aperta a tutti, più persone potrebbero entrare contemporaneamente e scriverebbero in modo confuso e disordinato sull’unico foglio disponibile, utilizzando una sola penna passata di mano in mano senza regole: il risultato sarebbe uno scritto indecifrabile.

Come prima cosa nel metodo costruttore (“__init__()”) della classe “Simulator” vengono inizializzati i seguenti attributi:
* “paused”, uguale a True (pone lo stato del thread principale come “in pausa”)
* “state”, che rappresenta un oggetto di tipo “Lock” (ovvero un “lucchetto” di sicurezza per accedere ad una sezione critica e garantire che non avvengano accessi concorrenti da parte di altri thread)
* “pq”, come coda con priorità (“priority queue”)

Quindi all’invocazione del metodo “run()” sull’istanza del “Simulator”:
* si invoca il metodo “resume()” che:

* acquisisce il lock su “state” (inizio sezione critica)
  * imposta lo stato del thread principale a “non in pausa” (assegnando False a “paused”)
  * invoca il metodo “notify()”, che risveglia esattamente uno dei thread in attesa di ottenere il lock “state” (e poter quindi entrare nella sezione critica)
  * rilascia il lock su “state” (fine sezione critica)
* si controlla che esista l’oggetto globale “Simulator” (“cmstate.sim”) e, nel caso non sia stato inizializzato, si esce stampando un messaggio errore
* il flag di controllo “exit_flag” viene inizializzato a False
* si esegue un ciclo finchè il flag di uscita “exit_flag” non diventa True, dove:
  * si acquisisce il lock su “state” (inizio sezione critica)
  * si controlla se lo stato è in pausa (“paused”) e, in tal caso, si chiama il metodo “wait()” sullo stato (“state”): questo addormenta il thread corrente finché qualcun altro non lo risveglierà
  * si rilascia il lock su “state” (fine sezione critica)
  * in seguito viene invocato il metodo “goNextEvent()”, il quale ritorna un valore booleano che viene poi assegnato al flag di uscita “exit_flag” (la spiegazione del metodo è riportata di seguito per maggiore chiarezza)
  * se il flag di uscita “exit_flag” è True, per ogni agente si controlla se il campo “enable_monitor_output” è True (di default è False, altrimenti è inizializzato diversamente al momento dell’esecuzione da terminale); in tal caso si chiama il metodo “save()” sull’istanza dell’agente di modo da salvare i dati dei “Monitor” su file:
    * per ogni “Monitor” associato all’agente, il salvataggio avviene su un file del tipo “nome_monitor.txt”
    * i dati sono salvati in una serie di “x” (tempi memorizzati) ed una di “y” (lunghezza della coda al tempo corrispondente)

Metodo “goNextEvent()”:
* se la coda con priorità “pq” è vuota, si esce subito dal metodo ritornando True
* altrimenti, dato che la coda “pq” contiene sicuramente dei “MessageEvent”, se ne estrae uno (i “MessageEvent” sono tuple costituite da: timestamp, evento, agente)
* se il timestamp è oltre il limite globale “until”, si ritorna True e si esce (perché è stato superato il numero di iterazioni massimo consentito)
* se il timestamp supera “nextlog”, si stampa a terminale il timestamp (con la stringa “Current time: t”); quindi si imposta il prossimo “nextlog” ad un timestamp futuro aggiungendo al timestamp corrente un periodo pari alla variabile globale “logtime” (da codice, “Simulator.nextlog+=cmstate.logtime”)
* se il tempo globale (“time”) è diverso dall’attuale timestamp, viene aggiornato (perché ciò significa che nel periodo compreso tra il timestamp presente e il timestamp del prossimo evento non c’è niente da eseguire; quindi non conviene aspettare tale tempo tenendo impegnato il processore senza ragione: si salta direttamente al timestamp del prossimo evento)
* se il file “orders_traverse_time.txt” è impostato (come avviene di default nell’inizializzazione del “Simulator”), si importa la lista di ordini completati dal file “agents/PRagents.py”; quindi per ognuno di questi ordini si calcola la quantità di tempo trascorso dalla creazione al completamento; tale periodo viene pesato sul numero totale di ordini

completati e il risultato (detto accumulo) viene aggiunto al file
  “order_traverse_time.txt” (si veda la “Nota 3” per un approfondimento
  su questo tipo di “statistiche”)
* se l’oggetto “guiObj” è stato inizializzato, si imposta il flag “update_gui”
  a True
* se l’evento estratto dalla coda con priorità “pq” è di tipo “ScheduleEvent”,
  questo viene invocato (in appendice la descrizione dettagliata degli
  “ScheduleEvent”), altrimenti l’evento è ricevuto dall’agente estratto dalla
  coda con priorità “pq”
* se esiste un oggetto “guiObj” ed il flag “update_gui” è True, viene
  invocato su tale oggetto il metodo “update()” in modo che l’interfaccia
  grafica venga aggiornata
* infine, si esce dal metodo ritornando False (per segnalare che non ci sono
  stati problemi durante l’esecuzione)

Nota 3: come si vedrà in seguito con la spiegazione dei “Monitor” (in
appendice), è complicato ottenere statistiche o dati aggregati sulla totalità
degli agenti perché, per come è stato progettato il software, ogni agente ha
propri “Monitor” che si occupano di fornire statistiche sul singolo agente.
In genere conviene operare così perché, altrimenti, gestire a livello globale
(o di observer) ogni singolo dato, le proprietà e le statistiche di ogni
agente come un’unica massa aggregata risulta molto più dispendioso in
termini di tempo computazionale e comporta una complessità del codice che ne
peggiora inevitabilmente la qualità e la leggibilità, così come la facilità
di comprensione. Conviene quindi gestire separatamente la parte statistica o di
aggregazione dei dati (leggendo i file generati dai “Monitor”), eventualmente
tramite codice scritto in un linguaggio come “R” (https://www.r-project.org),
con cui è più semplice ed efficace eseguire considerazioni statistiche e
“plottare” relative rappresentazioni in forma tabellare o di grafico.

Nota 4: una classe A figlia di una classe B eredita da quest’ultima tutti i
metodi e gli attributi e ne può ridefinire i metodi qualora ne dovesse
estendere o cambiare il comportamento. Si può anche dire che la classe A
“estende” la classe B perché, di fatto, ne specifica più nel dettaglio il
comportamento.

****

Appendice

****

-- Classe “Monitor”

La classe “Monitor” è definita in “acp.py” ed è figlia dalla classe “list”. Essa raccoglie, in due serie di valori “x” ed “y”, le “osservazioni” memorizzate in vari momenti dell’esecuzione. Tali “osservazioni” riguardano la lunghezza delle code proprie degli agenti e su di esse vengono calcolate statistiche come sum, arithmetic mean, variance, time average, time variance.

Ogni agente mantiene un dictionary (“monitors”) in cui vengono memorizzati tutti gli oggetti di tipo “Monitor” a lui associati (le keys sono i nomi dei “Monitor” ed i values sono gli oggetti stessi).
Nella classe “Agent” (file “core_agents.py”), oltre al dictionary, è presente anche la variabile booleana “monitored” (default=True) che definisce se i “Monitor” devono essere gestiti o no. Se richiesto, viene riempito il dictionary dei “Monitor” per ogni agente. Questa operazione viene eseguita tramite il metodo “addMonitor()”, che viene chiamato sull’istanza di un agente al momento della definizione ed inizializzazione del tipo d’agente stesso (si ricorda che tale tipo viene letto dalle proprietà “classType” impostate nel file .dot “agents_graph”). In particolare il metodo “addMonitor()” viene invocato solo durante la creazione di agenti "Unit" o “Warehouse” (classi definite nel file “PRagents.py”). Entrambe queste classi sono figlie della classe “ProductionAgent”, la quale a sua volta è figlia della classe di base “Agent”, definita in “core_agents.py”.

Ogni agente raccoglie nel suo dictionary “monitors” tre diversi “Monitor”, ognuno legato ad una coda specifica:
* “waitMon”, associato alla coda d’attesa “incomingQ”
* “activeMon”, associato alla coda degli eventi in processing “activeQ”
* “storageMon”, associato alla coda “local_storage” dell’agente (la coda è un dictionary con la funzione di magazzino per i prodotti che non è possibile consegnare)

Per eseguire “osservazioni” sulle code, si chiama il metodo “observe()” sui rispettivi oggetti “Monitor”, i quali memorizzano al tempo “t” (definito come “t=now()”) la situazione della coda che viene passata in input al metodo. Tale situazione è memorizzata in una lista di due elementi: il tempo “t” e la lunghezza della coda passata in input.
Per ogni chiamata del metodo “observe()”, viene aggiunta tale lista di due elementi alla lista globale rappresentata dall’oggetto “Monitor” (si ricorda che eredita dal tipo “list”). Il risultato finale sarà una lista che comprende a sua volta liste di due elementi ciascuna.

-- Classe “ScheduleEvent”

La classe “ScheduleEvent” è utilizzata per creare oggetti che in fase di inizializzazione ricevono:
* un’istanza di oggetto
* un nome di metodo (che deve essere un metodo presente tra quelli dell’istanza

indicata in precedenza)
* dei parametri da passare al metodo
Il metodo “__call__()” fa in modo che gli oggetti possano essere trattati come funzioni, quindi che possano essere invocati piuttosto che essere solamente passati come parametri al metodo “receive()” di un agente (come invece avviene per gli oggetti di tipo ”MessageEvent”).
Durante l’invocazione (nel caso l’oggetto sia di tipo “ScheduleEvent”) viene chiamato il metodo indicato in fase di inizializzazione sull’istanza indicata e coi parametri indicati; quindi si calcola con il metodo “nextCallAt()” se questo evento necessita di una rischedulazione e, in tal caso, se ne aggiunge una nuova istanza al “Simulator” indicando anche a quale timestamp dovrà essere eseguita.

-- Classe Simulator

Run()
- Invoca il metodo resume() che risveglia il thread self. Toglie dallo stato di pausa il simulatore.
- Se l’istanza del simulatore non è inizializzata, allora stampa un messaggio di errore e termina il programma.
- Altrimenti continua l’esecuzione di run(); in questo caso quindi imposta il flag “exit_flag” a False e fintanto che rimarrà così:
  - Se il simulatore è in pausa allora rimane in attesa sullo stato fino all’arrivo di una notifica di sblocco; lo stato è un oggetto di tipo Condition().
  - Condition() è importata da threading; rappresenta una Condition Variable: uno strumento per gestire la sincronizzazione dei thread (sempre associato ad un lock per una risorsa).
  - Dopodichè si aggiorna “exit_flag” al valore di ritorno del metodo goNextEvent(). Tale metodo determina il passo successivo da fare. In questo modo se il tempo è scaduto e/o la coda degli eventi è vuota (considerando solo i MessageEvent) allora termina l’esecuzione del simulatore (descrizione più approfondita in seguito).
  - Se il valore del flag dovesse essere True, che quindi indicherebbe la fine della simulazione, allora prima di terminare, per ogni oggetto agente, se il campo enable_monitor_output è definito, salva l’agente.
  - il campo enable_monitor_output è un campo che può essere fornito in input al programma tramite l’opzione -M. Tale campo viene recuperato nel main tramite args[“monitor_output”].

goNextEvent()
goNextEvent ritorna True se l’esecuzione deve essere fermata (per es. se la coda è vuota di MessageEvent o il tempo è scaduto).

- Se la coda pq (coda di priorità) non è vuota:
  - imposta di non aggiornare la GUI.
  - recupera la tripla costituita da tempo, evento e agente dalla coda pq tramite heappop(). Da questa tripla cerca di determinare il prossimo evento e/o se deve invece terminare l’esecuzione.
  - Quindi verifica se il tempo è maggiore di until, il quale determina il tempo

massimo della simulazione, ed in tal caso ritorna True (indica che l'esecuzione deve essere terminata).
- Dopodichè determina il tempo in cui effettuare la prossima stampa. quindi se il tempo è minore di until, ma maggiore di nextlog: cioè se il tempo associato all'evento corrente che deve essere schedulato è maggiore di nextlog, allora:
  - incremento nextlog di un logtime (che indica ogni quanto tempo fare le stampe).
  - Controllo se la differenza tra il tempo massimo di esecuzione (until) e il tempo del simulatore (nextlog) è maggiore di un logtime. in questo caso allora incremento ancora noxtlog di logtime.
  - Altrimenti imposto il nextlog ad until
- Se il cmstate.time, che è un contatore globale del tempo, è diverso da t, allora aggiorno il global time a t. Questo si fa per portare avanti il tempo del simulatore: so per certo che non ci sono altri eventi prima (pq è una coda con priorità che viene aggiornata ogni volta che si inserisce un evento, riordinandosi).
  - Se order_traverse_time_file è definito, cioè non è uguale ad una stringa nulla, allora deve aprire il file e scriverci dentro.
    - importa dal file PRagents la lista accomplished_orders, che mantiene la lista degli ordini completati. Ogni volta che un ordine viene completato viene appeso a questa lista tramite il metodo accomplishOrder().
    - Per tutti gli ordini della lista, somma tra loro la seguente espressione: (tempo di completamento dell'ordine - tempo attuale attuale) / il numero di ordini completati. Il tempo di completamento dell'ordine viene impostato in accomplishOrder() una volta completato. Il numero degli ordini completati lo ricava ottenendo la lunghezza della lista accomplished_orders. In questo modo calcola l'accumulo.
    - Appende l'accumulo sul file order_traverse_time_file. L'operazione di appendere qualcosa al file permette di non sovrascrivere il file. Nel caso in cui il file non esista, lo crea come nuovo.
  - Se è definita una GUI, allora reimposta update_gui a true. (per bilanciare la prima operazione svolta).
- Se l'evento ottenuto dalla coda pq è un'istanza di ScheduleEvent (cioè non è di tipo MessageEvent), allora, se è impostata cmstate.verbose (cioè se siamo in debug mode) stampa l'evento. In più invoca l'evento stesso: chiama lo scheduler decorator agganciato al metodo dentro l'evento.
- Viceversa, se l'evento è di tipo MessageEvent, allora mette l'evento direttamente nella coda locale dell'agente tramite la receive() (in questo caso si tratta di un'operazione che l'agente può svolgere direttamente).

addEvent()

- Aggiunge un evento alla coda con priorità del simulatore.

- Verifica che l'evento non sia nullo,
  - inserisce nella coda con priorità pq, la tripla: tempo di schedulazione (dato dal tempo attuale più il tempo di delay, cioè fra quanto schedulare l'evento), l'evento stesso e l'agente che dovrà ricevere l'evento.
  - Dopodichè incrementa il contatore corrispondente al tipo di evento che è stato appena aggiunto, cioè se è ScheduleEvent o MessageEvent. Questi due contatori però non vengono mai usati. L'operazione si potrebbe considerare ininfluente per il resto del programma.

-- Classe Order

Class Order
La classe Order rappresenta l'oggetto ordine all'interno della simulazione.

- __id_counter: è un identificativo univoco dell'ordine.
- accomplished_at: tempo di completamento. Impostato in accomplishOrder() (di ./agents/core_agents.py)
- ts: tempo di creazione
- Altri campi che non sono stati approfonditi (non compaiono nei metodi studiati).

-- Metodi della classe Agent

-- ClassE Agent

La classe Agent è definita in "./agents/core_agents.py". Questa classe implementa le funzionalità di base di un nodo agente e richiede di essere estesa per completarne le caratteristiche.
NodeAgent è una classe figlia di Agent e la specializza per implementare le funzionalità di Network Simulation in stile P2P. Questa classe fornisce funzionalità di dispatch per i MessageEvents, eventi da eseguire da parte dell'agente.

BackgroundAgent è una classe figlia di Agent i cui oggetti però non partecipano alla simulazione. Si tratta di agenti accessori che servono per realizzare l'ambiente di lavoro per i NodeAgent (per esempio la realizzazione della GUI). L'implementazione garantisce che questi agenti non siano mai raggiungibili dal sistema di dispatch degli eventi.

Quando AESOP è utilizzato come un simulatore P2P, la classe NodeAgent si comporta come un contenitore di protocolli che effettivamente realizzano la computazione. I protocolli non sono propriamente degli agenti, ma contengono quello che può essere considerato come il loro comportamento.

La struttura a grafo degli agenti permette di definire i concetti di neighboor (vicino) e neighborhood (vicinato). Il vicino di un nodo è qualsiasi nodo ad esso direttamente collegato mentre il vicinato è l'insieme di tutti i noti ad esso direttamente collegati. Questa distinzione è utile perché è possibile sfruttare i metodi getNeighbor() e getNeighborhood() per ottenere un vicino casuale o tutti i vicini rispettivamente.

incomingQ = []
incomingQ è una coda che ogni Agente memorizza internamente. L'obiettivo è quello di accodare gli eventi che sono mandati all'Agente per ordinarne l'esecuzione. Un evento è rappresentato da un oggetto MessageEvent.

Il metodo utilizzato per accodare un nuovo evento è receive() e accetta come unico parametro l'evento da inserire nella coda.

receive()
- Questo è un metodo implementato sia dalla classe Agent che da quella NodeAgent. La receive() viene chiamata solamente in goNextEvent() e solo nel caso in cui l'evento sia di tipo MessageEvent. Tuttavia, dentro al metodo viene ricontrollato che l'evento sia di tipo MessageEvent.
  - Se l'evento è replicato viene consumato direttamente; non viene inserito nella coda locale.
  - altrimenti viene inserito nella coda incomingQ: coda locale in ingresso dell'agente.
  - In entrambi i due sottocasi appena descritti, viene scorsa tutta lista locale e ad ogni evento viene invocato il metodo process().

process()
- Questo metodo implementa il comportamento passivo dell'agente (reazione ad ogni messaggio ricevuto). Semplicemente stampa l'evento.

handleBehavior()
- Questo metodo implementa il comportamento attivo dell'agente. Gestisce gli ScheduleEvent stabilendo quando un'evento di questo tipo deve essere schedulato inserendolo nella coda di priorità. Tramite quindi next_t, inteso come ritardo, imposta il tempo di schedulazione del prossimo evento (ScheduleEvent).
- next_t viene impostato tramite nextWakeup()

nextWakeup()
- Calcola il tempo, inteso come tempo di attesa, per la prossima attivazione del comportamento attivo dell'agente cioè il tempo in cui verrà schedulato il prossimo ScheduleEvent. Metodo invocato solo da handleBehavior().
  - Imposta quindi il tempo a -1, cioè come se non dovesse essere più schedulato nessun evento.
  - Recupera il valore del contatore globale del tempo (cmstate.time) e lo assegna a whereNow.
  - Se contemporaneamente fromTime ed evrey sono impostate a zero, allora termina e come tempo restituisce -1 alla handleBehavior(). fromTime e every sono due variabili dell'agente che di default sono impostate a zero ma che eventualmente possono essere settate quando viene creato l'agente.
  - Se invece la variabile every non è nulla, ma whereNow e fromTime invece lo sono, allora si imposta il tempo ad every. whereNow a zero indica che siamo all'inizio della simulazione.
  - Se invece whereNow è zero ma fromTime non è nullo allora imposta next_time a fromTime.
  - Altrimenti, cioè se non sono all'inizio della simulazione (whereNow diverso da zero):
    - se la variabile stdev è uguale a 0.0 allora imposta next_time a every. viceversa viene utilizzato un generatore gaussiano per assegnare un valore a tale variabile.
    - Infine restituisce il tempo di ritardo in cui verrà rischedulato lo ScheduleEvent.

talkTo()

Il metodo talkTo() invia un MessageEvent ad un agente di destinazione tramite il suo nome. Come comportamento di default, effettua questo invio istantaneamente, lasciando comunque la possibilità di inserire un ritardo arbitrario.

- Nel caso un destination_agent, descritto nel grafo degli agenti (agentsGraph), abbia l’attributo ‘delay’ diverso da zero allora calcola il tempo di ritardo per l’evento da aggiungere alla coda del simulatore. Questo valore prevale su quello passato come parametro a talkTo():
  - Viene quindi settato graph_delay al valore dell’attributo ‘delay’ dell’arco del agentsGraph.
  - Se graph_delay è un oggetto con il metodo __call__, cioè che la classe può essere chiamata come una funzione, allora imposta delay al valore ritornato dalla classe stessa.
  - Altrimenti, recupera dall’attributo delay_stdev il valore da assegnare a stdev. Se stdev è uguale a zero allora setta delay al valore che era presente sull’arco; altrimenti calcola il ritardo tramite una distribuzione gaussiana avente il suddetto valore come valore medio.
- Altrimenti utilizza direttamente i parametri passati a talkTo().
- Infine aggiunge l’evento (addEvent) alla coda del simulatore con i parametri appena definiti.

trigger()
Questo metodo automatizza la chiamata del default method handleBehavior() al prossimo step. Questo metodo deve essere chiamato quando un nuovo agente è inserito nel sistema, invece, durante l’inizializzazione, il simulatore gestisce questa funzionalità da solo.
In realtà questo metodo non viene mai chiamato all’interno dell’esecuzione del simulatore.

Pubblicato con licenza Creative Commons Attribution-ShareAlike 4.0 (CC BY-SA 4.0).