

Esame di Sistemi Distribuiti

IMPLEMENTAZIONE PERSISTENZA TUCSON

Autori:

Lorenzo Pontellini
Vincenzo Scafuto

Coordinatore:

Stefano Mariani

Sommario

ANALISI SITUAZIONE ATTUALE	3
OBIETTIVI	4
FUNZIONAMENTO IN DETTAGLIO	6
STRUTTURA DEL FILE XML	9
TEST DI PERFORMANCE EFFETTUATI	10
MIGLIORAMENTO PERFORMANCE	11
CONCLUSIONI	11

ANALISI SITUAZIONE ATTUALE

La persistenza in TuCSoN avviene in due passaggi:

- Nel caso in cui non sia già presente un file si procede alla creazione di un file nel quale salvare lo stato del nodo;
- Viene effettuato uno snapshot TuCSoN; ossia in un primo file viene salvato lo stato attuale con le caratteristiche possedute al momento dell'abilitazione della persistenza;
- Durante tutto il periodo in cui rimane abilitata la persistenza per ogni update(inserimento, cancellazione di tuple) si controlla prima se presente il file di snapshot e si procede al suo aggiornamento andando ad indicare le tuple che hanno subito un cambiamento.

Con snapshot si indica una istantanea dello stato del nodo. Infatti nel file creato si riportano, per ogni categoria di tupla presente nel nodo, l'identificativo. Le tipologie di strutture delle quali si tiene traccia sono:

- Tuple;
- SpecTuples;
- Predicates;

Il file che contiene tutte queste informazioni ha estensione *.dat*.

Queste operazioni avvengono all'interno della classe *RespectVMContext* ed è in questa che si dovrà agire.

Nella situazione attuale, come si può vedere dal file *PersistencyTester*, l'abilitazione della persistenza avviene facendo una operazione di out con valore *"enable_persistency"*.

All'abilitazione viene richiamato il metodo *recoveryPersistent* il quale va alla ricerca di un vecchio file di persistenza se presente lo va ad aggiornare con gli update eseguiti dal dono dall'abilitazione della persistenza.

Allo stesso modo la disabilitazione avviene facendo una operazione di out con valore *"disable_persistency"*.

Entrambi sono metodi che risiedono nella classe `RespectVMContext` e sono chiamati con valore di default.

OBIETTIVI

Una volta capito e analizzato il funzionamento, si vuole modificare `TuCSoN` per fare sì che la modalità di salvataggio dei dati non avvenga più su file con estensione `.dat` ma tramite l'uso di un file XML.

I lati positivi dell'uso di questa tecnologia, che verranno nel seguito verificati, sono:

- L'XML risulta essere un formato di file standard;
- E' utilizzato come mezzo per l'esportazione di dati tra DBMS;
- La creazione e la gestione avvengono in maniera rapida;
- Data la possibilità di strutturare il file a livelli prevedendo un ordine gerarchico si adatta alle caratteristiche delle tuple presenti in `TuCSoN`.
- Possibilità di criptare il file;

A tal scopo sono state predisposte due opportune classi per la creazione e gestione di questa tipologia di file. le quali sfruttano delle librerie già presenti in java, in particolare nel pacchetto: `javax.xml`.

PersistencyXML
ROOT_NODE : String TUPLES_NODE : String TUPLE_NODE : String SPEC_TUPLES_NODE : String SPEC_NODE : String PREDICATES_NODE : String PREDICATE_NODE : String UPDATES_NODE : String UPDATE_NODE : String pDate : String pFileName : TucsonTupleCentreId pPath : String xmlFile : File
PersistencyXML(path : String,fileName : TucsonTupleCentreId) PersistencyXML(fileName : String) getNodeInfo(node : Node,childName : String) parse() : PersistencyData writeUpdate(update : LogicTuple,mode : ModType) write(pData : PersistencyData)

PersistencyData
tupleSet : ITupleSet tupleSpecSet : ITupleSet prologPredicates : TupleSet tuples : List<String> specTuples : List<String> predicates : List<String> updates : List<String>
PersistencyData(tSet : ITupleSet,tSpecSet : ITupleSet,TupleSet : pPrologPred,List<String> : tUpdates) PersistencyData(tuples : List<String>,specTuples : List<String>,pPredicates : List<String>,updates : List<String>) getTupleSet() : ITupleSet setTupleSet(tupleSet : ITupleSet) getTupleSpecSet() : ITupleSet setTupleSpecSet(tupleSpecSet : ITupleSet) getPrologPredicates() : TupleSet setPrologPredicates(prologPredicates : TupleSet) getUpdates() : List<String> setUpdates(updates : List<String>) getTuples() : List<String> setTuples(tuples : List<String>) getSpecTuples() : List<String> setSpecTuples(specTuples : List<String>()) getPredicates() : List<String> setPredicates(pPredicates : List<String>)

All'interno della classe sopra citata sono stati effettuati i seguenti cambiamenti per adattare TuCSon alla scrittura su file XML, lasciando però inalterata la struttura del file, infatti, la classe *PersistencyData* incapsula al suo interno tutte le informazioni delle varie tipologie di tuple presenti nel nodo.

Come prima cosa nel metodo *enablePersistency* si crea un oggetto di classe *PersistencyData*.

Questo oggetto costruito ad hoc viene passato alla classe *PersistenceXML* che tramite il metodo *write()* leggerà il valore dei campi interni all'oggetto e aggiornerà il file XML.

Non sono cambiate le modalità di denominazione del file, infatti, viene salvata la data e l'ora di attivazione della persistenza che daranno il nome al file.

Questo servirà per distinguere quelli più recenti.

FUNZIONAMENTO IN DETTAGLIO

Quando è abilitata la persistenza, le operazioni svolte sono qui brevemente descritte:

- Si crea un oggetto di classe *PersistenceData*.
- Il costruttore assegna ai campi interni dell'oggetto le specifiche del nodo TucSoN le quali sono passate come parametri e salvate come liste di stringhe:
 - o *Tuples*;
 - o *SpecTuples*;
 - o *Predicates*;
 - o *Updates*;

Fino a qui il funzionamento è simile alla precedente versione. Una volta creato il *PersistenceData*, si crea un oggetto di classe *PersistenceXML* con parametri:

- nome del file;
- path di dove effettuare il salvataggio;

La classe mette a disposizione un metodo che preso in ingresso l'oggetto *PersistenceData* scrive il file XML.

Il metodo *write()* lanciato sull'oggetto di classe *PersistenceXML* prende come argomento un *PersistenceData* e crea un file di estensione XML nel cui nome è presente la stampa della data e dell'ora per tenere traccia il momento in cui viene fatto l'aggiornamento e l'update.

Per la creazione del file si sfrutta un oggetto di classe *DocumentBuilderFactory* usato per la creazione dell'oggetto *DocumentBuilder*.

Si passa ora alla creazione dei livelli o nodi di cui è costituito il file XML. Il primo è il ROOT_NODE che, come nel precedente file *.dat*, corrisponde all'abilitazione dello snapshot. Si settano poi gli attributi del nodo root quali tc e time.

Una volta completato il nodo radice, gli si aggiungono tutti i nodi figli che possono essere presenti nel nodo TuCSoN che sono rappresentati da

- *Tuples;*
- *SpecTuples;*
- *Predicates.*

Tramite metodi *get* presenti nella classe *PersistenceData* si ritorna in possesso delle informazioni, precedentemente salvate all'interno dell'oggetto, sotto-forma di liste di stringhe.

Per popolare il file XML si fa uso di oggetti appartenenti alla classe *Element*. I quali fungono sia da figli della classe root sia da elemento appartenente ad un figlio.

Per riempire ognuno dei quattro oggetti creati, si sfrutta un iteratore applicato alla lista della stessa tipologia ottenuta dall'oggetto *PersistenceData*.

Tramite questo si aggiungono tutti i figli ad una sottocategoria e come ultimo passaggio la macro categoria al root node.

Una volta portate a termine queste operazioni per ogni categoria di dato si conclude la prima creazione del file XML.

Per quanto riguarda la seconda fase, quella cioè che serve per tenere traccia degli update fatti sulle tuple del nodo, si predispone un nuovo oggetto *PersistenceXML* sul quale viene chiamato il metodo *parse()*.

Questo metodo itera su tutti i nodi dell'oggetto e per ognuno riconosce il tipo di figlio e passando come argomento la lista di stringhe di tutte le tuple di quel tipo crea un nuovo oggetto di classe *PersistenceData* che contiene tutti gli update divisi per tipologia di tupla.

Questo nuovo oggetto creato viene passato all'interno del metodo *writePersistenceUpdate(...)* presente nella classe *RespectVMContext* il quale, nel caso di file *.dat* andava a scrivere gli update, in questo caso va ad aggiornare il relativo file XML.

L'update funziona con la stessa struttura ma con un unico metodo che crea un figlio chiamato appunto update. Esso viene scritto mediante la stessa semantica precedentemente usata ovvero:

(<inserimento/rimozione><tipologia di tupla>)<update fatto>

STRUTTURA DEL FILE XML

La struttura del file xml viene di seguito descritta:

```
<persistence>
  <snapshot tc="'@(default,':'(localhost,20504))'" time="2014-10-13_21.06.53">
    <tuples>
      <tuple> ... </tuple>
    </tuples>
    <specTuples>
      <spec> ... </spec>
    </specTuples>
    <predicates>
      <predicate> ... </predicate>
    </predicates>
  </snapshot>
<updates time="2014-10-13_21.07.00">
  <update action="addition" subject="tuple"> ... </update>
</updates>
</persistence>
```

- Tutti i nodi sono figli di **persistence**;
- Figlio **Snapshot**: come attributo vi sarà un campo nome e parametro il nome del nodo dove si è deciso di abilitare la persistenza; un campo time: identificativo de tempo nel momento in cui l'abilitazione è attiva;
 - Figlio **tuples**: all'interno di questo vi sarà un figlio tuple per ogni tupla presente nel nodo al momento dello snapshot;
 - Figlio **specsTuples**: all'interno di questo vi sarà un figlio spec per ogni tupla presente nel nodo al momento dello snapshot;
 - Figlio **predicates**: all'interno di questo vi sarà un figlio predicate per ogni tupla presente nel nodo al momento dello snapshot;
- Figlio **updates**: con campo time sta ad indicare tutti i cambiamenti che sono accorsi nel nodo;
 - Figlio **update**: come campo **action** avrà l'azione che è stata svolta (addition, deletion, clean) e come **subject** chi ha subito l'azione prima specificata (predicate, specification tuple, tuple);

TEST DI PERFORMANCE EFFETTUATI

Per quanto riguarda l'esecuzione di test riguardanti le performance è stata predisposta una opportuna classe chiamata **PersistencyPerformance** la quale non fa altro che scrivere su un file di log una stringa, questa viene utilizzata all'interno della classe **RespectVMContext** per valutare le tempistiche relative alla creazione di un file **.dat** comparata a quella di un file **XML**. Oltre a questo anche all'update del file. Di seguito viene riportato un esempio:

Time elapsed for create file dat: 13	Time elapsed for update file XML: 16
Time elapsed for create file xml: 108	Time elapsed for update file DAT: 0
Time elapsed for update file XML: 58	Time elapsed for update file XML: 15
Time elapsed for update file DAT: 0	Time elapsed for update file DAT: 0
Time elapsed for update file XML: 57	Time elapsed for update file XML: 14
Time elapsed for update file DAT: 0	Time elapsed for update file DAT: 0
Time elapsed for update file XML: 41	Time elapsed for update file XML: 15
Time elapsed for update file DAT: 0	Time elapsed for update file DAT: 0
Time elapsed for update file XML: 48	Time elapsed for update file XML: 13
Time elapsed for update file DAT: 0	...
Time elapsed for update file XML: 31	
Time elapsed for update file DAT: 0	
Time elapsed for update file XML: 26	
Time elapsed for update file DAT: 0	
Time elapsed for update file XML: 28	
Time elapsed for update file DAT: 0	
Time elapsed for update file XML: 16	
Time elapsed for update file DAT: 0	

Come si può vedere dai risultati ottenuti si nota come in generale i tempi siano a favore dell'utilizzo del file con estensione **.dat** dato che già nella fase di creazione il tempo che intercorre è 1/6 di quello per la creazione di un file **XML**, si pensa che questo sia imputabile al fatto che il file **XML** sia strutturato.

Si nota inoltre che mentre le tempistiche degli update (misurate in millisecondi) siano pari a 0 per ogni tipo di update nell'utilizzo della prima tipologia di file, non lo sono per quanto riguarda l'uso di **XML** dove si aggirano come media sui 20 millisecondi.

Se si vuole avere più informazioni si rimanda nella cartella dove vengono salvati i file e all'interno si troverà il file completo con estensione **.log**.

Per quanto riguarda un'estensione delle features di XML aggiungendo una cifratura ai dati per renderlo anche più sicuro non sono state effettuate prove di performance in quanto si ipotizza che questa sia una operazione onerosa in termini di tempo e risorse da fare solo nei contesti dove viene richiesto una forte sicurezza anche a discapito delle prestazioni.

MIGLIORAMENTO PERFORMANCE

Dati i risultati ottenuti sulle performance della creazione e gestione del file xml si è proceduto ad effettuare delle modifiche per migliorare le performance in fase di update:

1. Si svolge il parsing degli updates;
2. In base al subject (tuple/spec/predicate) e l'action (addition/deletion/clean) si creano stringhe identificative presenti nella vecchia versione con il file .dat (esempio un `<update subject="tuple" action="addition">5</update>` diventerà (+t) 5;
3. a questo punto, una volta creata la lista di stringhe di update si va a ricercare (et) (es) (ep);
4. Se trovo (et) cancello dalla lista tutte le tuple (quindi tutti i +t e -t) dall'inizio fino alla posizione (et) (e cancello anche (et));
5. Allo stesso modo faccio per (ep) ed (es);

Si è verificato che questo accorgimento tende ad aumentare le performance di scrittura degli update nel file.

CONCLUSIONI

Dai risultati ottenuti si vede come l'utilizzo di un file strutturato tipo XML da come garanzie la possibilità di essere compreso e gestito in maniera automatica, essendo XML lo standard per lo scambio di informazioni sulla rete, dando la possibilità di essere esteso tramite crittografia per gestire il fattore sicurezza ma tutto questo a discapito di prestazioni in termini di risorse utilizzate e tempiste di esecuzione.

Si aspetta una conferma su quale strada intraprendere ora.