Relazione di un esercizio di profiling sulla classe TucsonNodeService.java

Sommario

Beatrice Mezzapesa

15/9/2014

1. Premessa

Obiettivo del presente rapporto è documentare l'attività di profiling effettuata sulla classe TucsonNodeService.java allo scopo di verificare:

- La memoria RAM
- Le classi che occupano più memoria
- I metodi che impiegano più tempo

La motivazione di tale verifica è capire le possibilità di riuscita di una attività di porting del sistema su un dispositivo mobile, avendo questo device meno risorse computazionali di un personal computer, in termini di: cpu, memoria e ram.

I risultati del lavoro svolto dovrebbero far emergere eventi critici, quali: intercettazione di componenti fondamentali che vincolino le prestazioni del sistema (bottleneck), utilizzo di risorse oltre il dovuto, con conseguente malfunzionamento del dispositivo mobile, bugs che impattano sulle già limitate risorse del device stesso.

2. TuCSoN

E' importante, prima di proseguire nella relazione tecnica, definire TuCSoN (Tuple Centres Spread over the Network) e il flusso delle sue operazioni.

TuCSoN è un modello per la coordinazione nei Sistemi Multi-Agente (MAS) che fornisce gli spazi di tuple come astrazione principale per progettare e sviluppare artefatti di coordinazione general purpose, utilizzando il linguaggio di specifica ReSpecT. Gli agenti interagiscono con il centro di tuple e si coordinano tramite lo scambio di tuple, attraverso un insieme di primitive Linda-like (in, rd, inp, rdp).

Flusso delle operazioni gli agenti dialogano con TuCSoN mediante gli ACC, che sono delle interfacce che permettono di eseguire le operazioni sul tuple centre. Gli ACC possono essere di vari tipi e definiscono sia le operazioni che un agente può compiere, sia le modalità di comunicazione (sincrona, asincrona, etc.) gestendone lo stato in maniera trasparente. Nell'implementazione reale l'ACC è suddiviso in due parti, un ACC lato agente e un ACC lato nodo, e i due comunicano tra di loro mediante un protocollo di alto livello chiamato Tucson Protocol. Considerando stabilita la connessione tra l'agente e il centro di tuple, e quindi tra ACCProxyAgentSide e ACCProxyNodeSide, dal punto di vista dell'agente, la richiesta di eseguire una specifica primitiva viene realizzata tramite l'invio di un messaggio, TucsonMsgRequest, direttamente al centro di tuple da contattare. Questo messaggio viene costruito partendo dall'istanza della classe TucsonOperation relativa alla primitiva richiesta. Le principali operazioni che permettono l'invio di un messaggio sono incapsulate nel metodo doOperation, definito all'interno della classe ACCProxyAgentSide; tali operazioni possono essere bloccanti o non bloccanti (metodi doBlockingOperation e doNonBlockingOperation, definiti all'interno dell'entità TupleCentreContainer*). Inviato il messaggio, l'agente attende una risposta dal centro di tuple contattato, TucsonMsgReply, grazie ad un thread di controllo. Non appena viene pervenuta una risposta, questo processo controlla se l'esecuzione è stata terminata con successo e se l'informazione ottenuta risulta coerente con la richiesta effettuata. In tal caso, lo stesso processo si occupa di segnalarlo all'ACCProxyAgentSide, che è ancora sospeso in attesa della risposta, il quale restituisce il risultato dell'operazione all'agente.

(*) TupleCenterContainer

Recupero dell'interfaccia relativa all'operazione corrispondente. A seconda che si tratti di operazioni ordinarie oppure di specifica, bloccanti (sincrone) o non bloccanti (asincrone), viene recuperata l'interfaccia che ne definisce i metodi: OrdinarySynchInterface, OrdinaryAsynchInterface, SpecificationSynchInterface, SpecificationAsynchInterface.

3. Relazione Tecnica

Nell'ambiente sopra descritto, la classe TucsonNodeService.java, meglio conosciuta come il nodo di TuCSoN cioè la parte principale del sistema, può essere divisa in varie componenti:

• L'ACCProvider;

• Il WelcomeAgent;

• Il NodeManagementAgent;

• L'ACCProxyNodeSide.

Il WelcomeAgent ha il compito di mettersi in ascolto di nuove richieste effettuate al nodo TuCSoN sulla porta da lui indicatagli e nel momento in cui riceve una richiesta entra in gioco l'ACCProvider, il quale ha il compito di creare l'ACCProxyNodeSide e di fornire l'ACC (Agent Coordination Context, interfaccia assegnata ad un agente che gli consente di effettuare operazioni su determinati centri di tuple e inoltre garantisce coordinazione fra gli agenti e il centro di tuple stesso) all'agente richiedente.

L'ACCProxyNodeSide si occuperà di tutte le comunicazioni fra l'agente e il centro di tuple. Infine il NodeManagementAgent rappresenta l'agente il quale ha il compito operativo di effettuare le operazioni richieste dagli agenti sui centri di tuple del nodo.

3.1 Approccio iniziale

Inizialmente, l'attività di profiling è stata effettuata semplicemente osservando l'operatività della classe; successivamente, invece, sono stati lanciati in esecuzione specifici esempi già forniti nell'ambito del progetto e sono stati messi a confronto gli indicatori rilevati.

L'attività progettuale è stata approcciata utilizzando il tool VisualVM, ma sono stati riscontrati una serie di problemi, risolti poi con l'adozione del tool JProfiler.

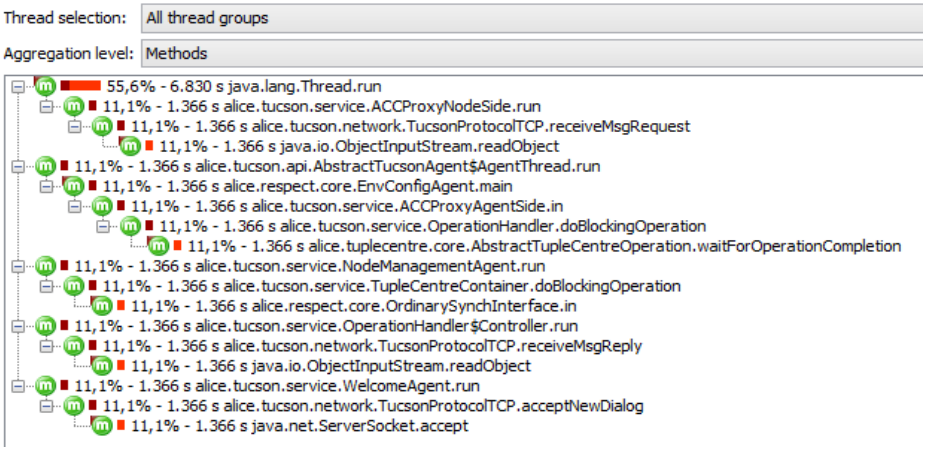
Nello specifico, il tool VisualVM nell'occupazione della memoria mostrava solo quella occupata dalle classi di Java, mentre necessitava ottenere quella occupata dalle classi di Tucson; inoltre, nella visualizzazione dei thread non veniva specificata la classe corrispondente. Per cercare di risolvere questo secondo problema, ho modificato manualmente il codice, al fine di nominare i singoli thread, ma senza ottenere alcun risultato.

3.2 Operatività della classe – dati rilevati

Nella seguente immagine vengono presentati i thread in esecuzione con i relativi tempi di permanenza nei metodi chiamati:

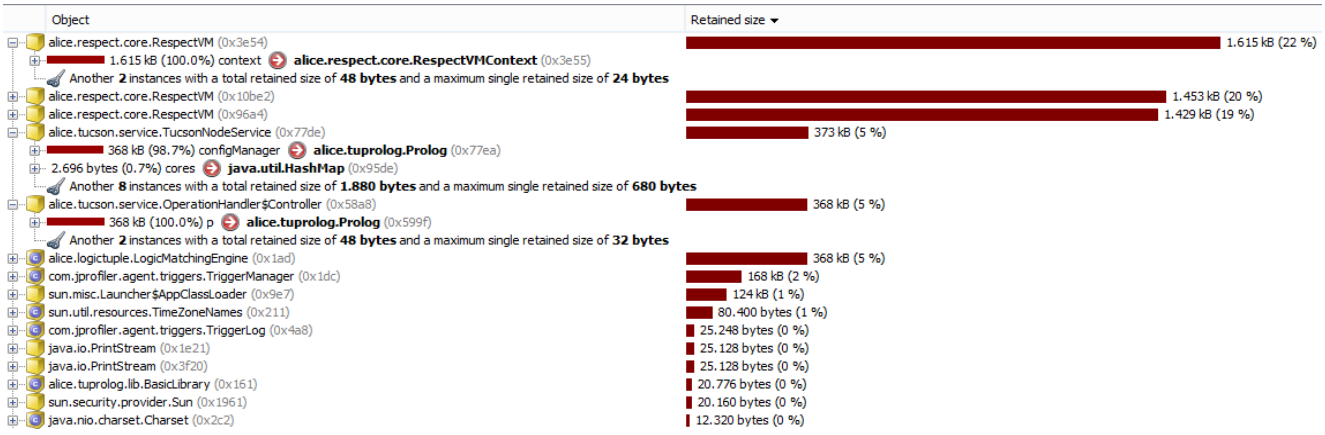
- sia quelli in stato di waiting (il thread sta dormendo e verrà svegliato o da un timer o da un altro thread)
- sia quelli in Net IO (un thread è in ascolto per le connessioni socket o è in attesa di leggere o scrivere dati su un socket);

Gli altri due possibili stati che può assumere un thread sono runnable (il thread è pronto per essere utilizzato) e blocked (il thread sta cercando di entrare in una sezione di codice o in un metodo attualmente detenuto da un altro thread), ma ovviamente fino a quando non viene lanciato un esempio specifico questi non verranno presi in considerazione.

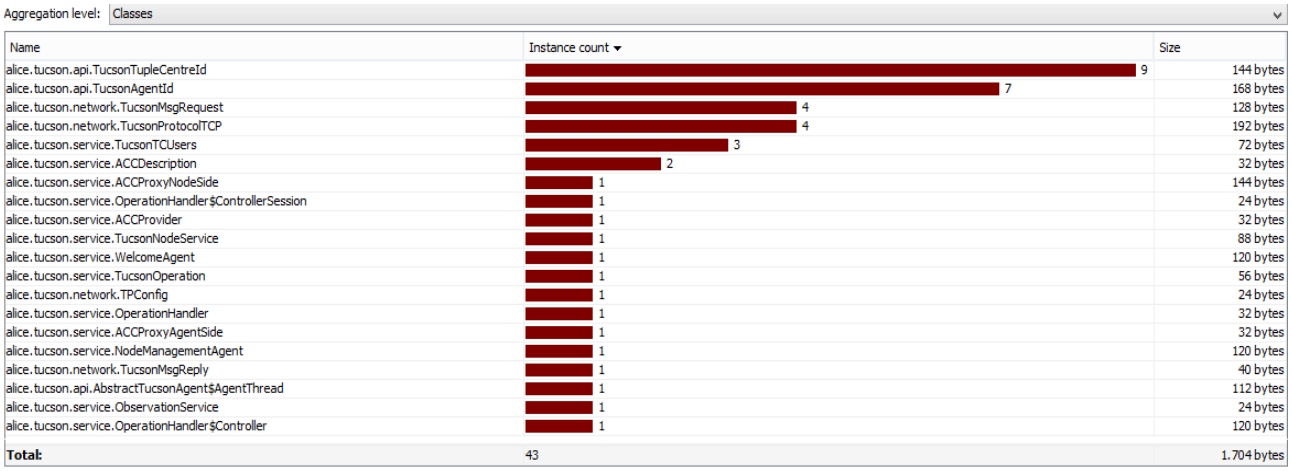


In ogni caso, la verifica dell'operatività ha l'obiettivo di fornire un riscontro preciso in merito alle classi e ai thread utilizzati; dati di riscontro che diventano poi importanti nell'ambito degli esempi chiamati.

Si evidenzia, comunque, come **respectVM** (processo incaricato di rimanere in ascolto degli eventi generati in risposta alle richieste effettuate dagli agenti; il cui ruolo principale consiste nell'assicurare la continuità del ciclo di lavoro) sia sempre in prima linea tra i biggest object; situazione che rimarrà inalterata anche quando analizzeremo gli esempi.



Per quanto riguarda le classi di TuCSoN, ci vengono mostrate le più utilizzate con il rispettivo numero di istanze che vengono allocate nell'heap, e la dimensione totale delle istanze per ogni classe.

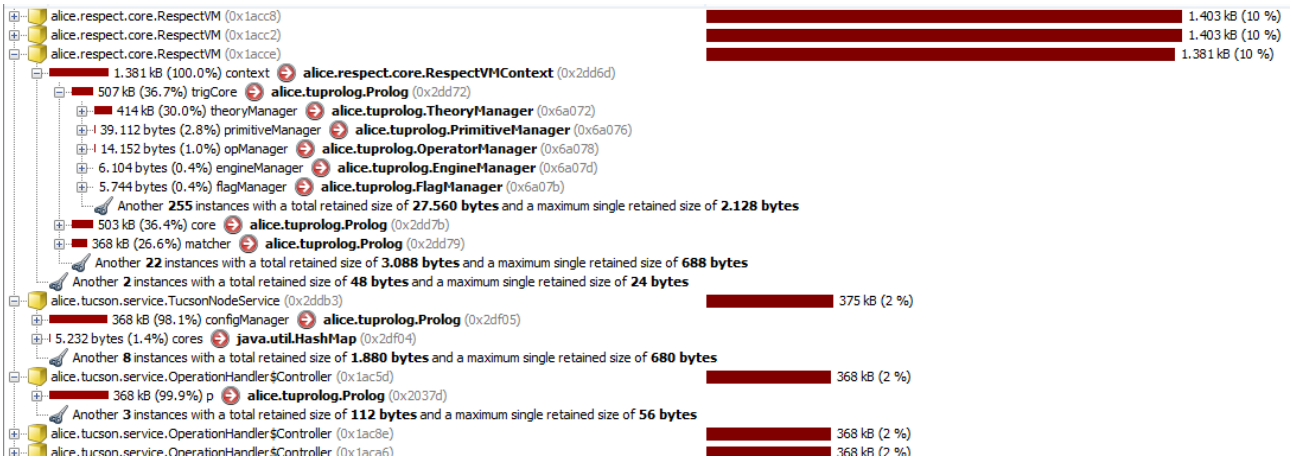


Aggregation level: Classes

Name	Instance count	Size
alice.tucson.api.TucsonTupleCentreId	9	144 bytes
alice.tucson.api.TucsonAgentId	7	168 bytes
alice.tucson.network.TucsonMsgRequest	4	128 bytes
alice.tucson.network.TucsonProtocolTCP	4	192 bytes
alice.tucson.service.TucsonTCUsers	3	72 bytes
alice.tucson.service.ACCDescription	2	32 bytes
alice.tucson.service.ACCProxyNodeSide	1	144 bytes
alice.tucson.service.OperationHandler$ControllerSession	1	24 bytes
alice.tucson.service.ACCProvider	1	32 bytes
alice.tucson.service.TucsonNodeService	1	88 bytes
alice.tucson.service.WelcomeAgent	1	120 bytes
alice.tucson.service.TucsonOperation	1	56 bytes
alice.tucson.network.TPConfig	1	24 bytes
alice.tucson.service.OperationHandler	1	32 bytes
alice.tucson.service.ACCProxyAgentSide	1	32 bytes
alice.tucson.service.NodeManagementAgent	1	120 bytes
alice.tucson.network.TucsonMsgReply	1	40 bytes
alice.tucson.api.AbstractTucsonAgent$AgentThread	1	112 bytes
alice.tucson.service.ObservationService	1	24 bytes
alice.tucson.service.OperationHandler$Controller	1	120 bytes
Total:	43	1.704 bytes

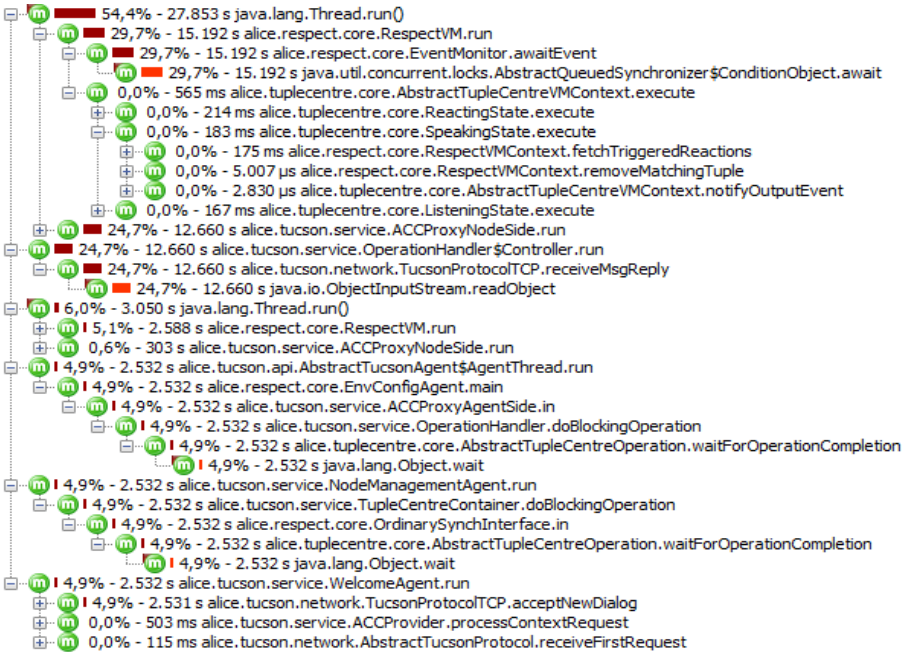
3.3 Esecuzione degli esempi – dati rilevati

Una volta fatta la prova iniziale, possiamo passare a quella più complessa dove, oltre a far girare il TucsonNodeService in background, si lanciano in esecuzione esempi specifici. Questa analisi è stata svolta per evidenziare possibili differenze dal caso iniziale, nonché per evidenziare se le classi che occupano maggiormente CPU cambiano a seconda degli esempi o meno.



In tutti gli esempi è stato riscontrato che i dati che occupano più memoria, sono sempre gli stessi, in particolare **respectVM**, come già avevamo detto sopra, con il rispettivo **tuprolog**.

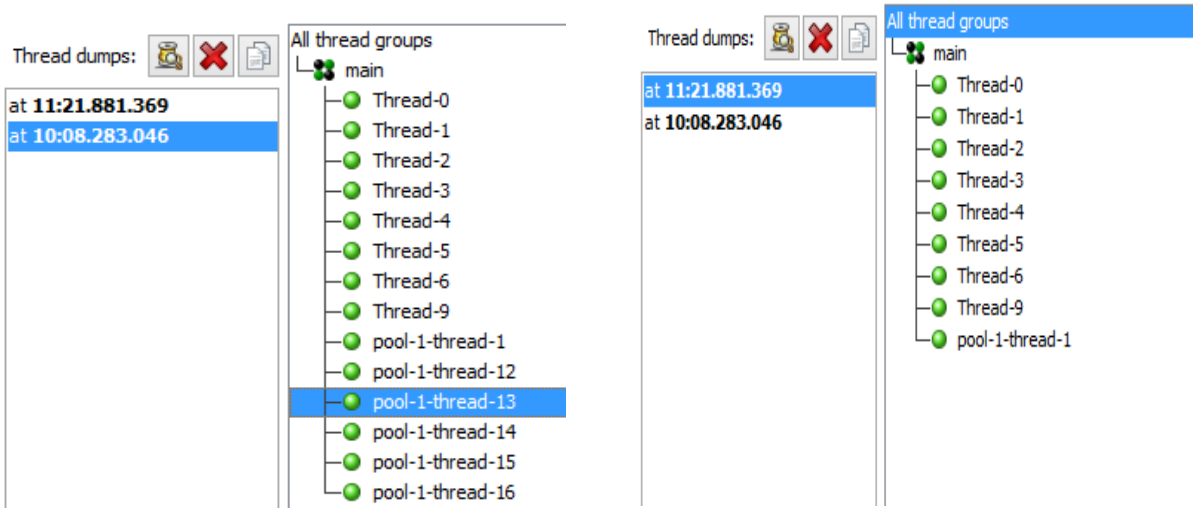
Per quanto riguarda invece i thread in funzione, ritroviamo per la maggior parte ciò che era già stato descritto nel capitolo 2 con il flusso delle operazioni; nello specifico:



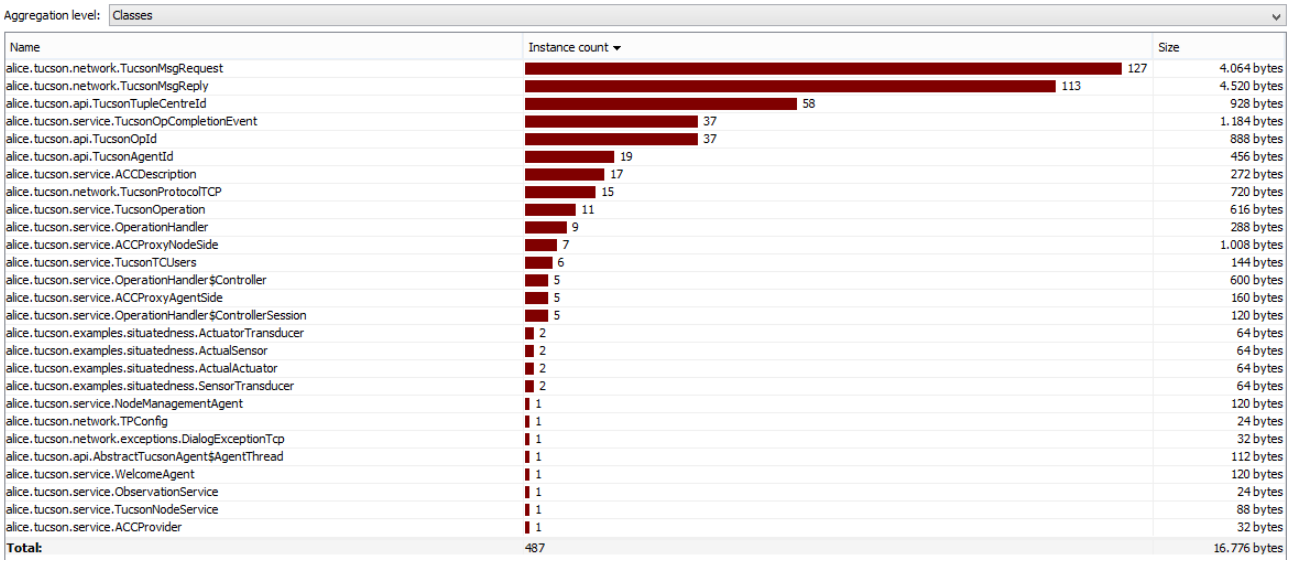
- ACCProxyAgentSide
- ACCProxyNodeSide
- doOperation,
- TucsonMsgReply
- TucsonMsgRequest

Inoltre, in figura vengono anche mostrati i tempi di esecuzione.

In particolare si è dimostrato che alla terminazione del main del programma client, in poco più di un minuto, i nuovi thread creati non rimangono più attivi in memoria:



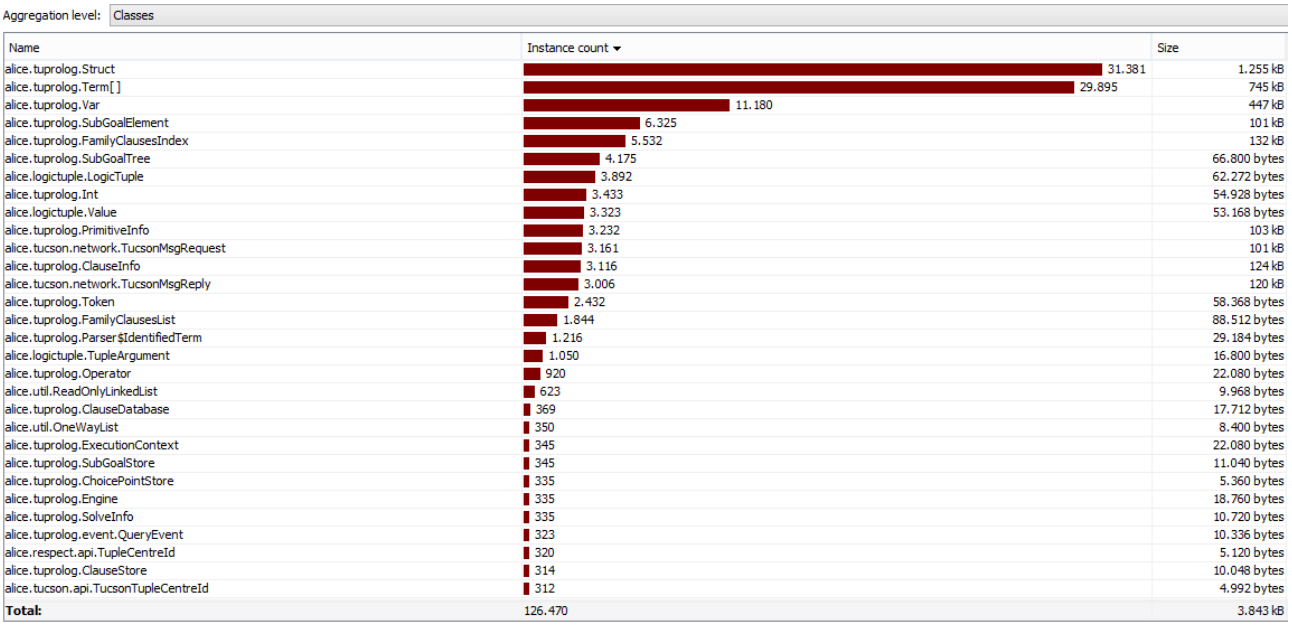
A proposito delle classi di TuCSoN e le rispettive istanze possiamo notare che le prime due classi più utilizzate sono giustamente quelle del messaggio di richiesta e di risposta, cosa che ovviamente non è presente nell'esecuzione della sola classe TucsonNodeService perché non c'è alcun agente attivo che instaura la comunicazione.



Aggregation level: Classes

Name	Instance count	Size
alice.tucson.network.TucsonMsgRequest	127	4.064 bytes
alice.tucson.network.TucsonMsgReply	113	4.520 bytes
alice.tucson.api.TucsonTupleCentreId	58	928 bytes
alice.tucson.service.TucsonOpCompletionEvent	37	1.184 bytes
alice.tucson.api.TucsonOpId	37	888 bytes
alice.tucson.api.TucsonAgentId	19	456 bytes
alice.tucson.service.ACCDescription	17	272 bytes
alice.tucson.network.TucsonProtocolTCP	15	720 bytes
alice.tucson.service.TucsonOperation	11	616 bytes
alice.tucson.service.OperationHandler	9	288 bytes
alice.tucson.service.ACCProxyNodeSide	7	1.008 bytes
alice.tucson.service.TucsonTCUsers	6	144 bytes
alice.tucson.service.OperationHandler$Controller	5	600 bytes
alice.tucson.service.ACCProxyAgentSide	5	160 bytes
alice.tucson.service.OperationHandler$ControllerSession	5	120 bytes
alice.tucson.examples.situatedness.ActuatorTransducer	2	64 bytes
alice.tucson.examples.situatedness.ActualSensor	2	64 bytes
alice.tucson.examples.situatedness.ActualActuator	2	64 bytes
alice.tucson.examples.situatedness.SensorTransducer	2	64 bytes
alice.tucson.service.NodeManagementAgent	1	120 bytes
alice.tucson.network.TPConfig	1	24 bytes
alice.tucson.network.exceptions.DialogExceptionTcp	1	32 bytes
alice.tucson.api.AbstractTucsonAgent$AgentThread	1	112 bytes
alice.tucson.service.WelcomeAgent	1	120 bytes
alice.tucson.service.ObservationService	1	24 bytes
alice.tucson.service.TucsonNodeService	1	88 bytes
alice.tucson.service.ACCProvider	1	32 bytes
Total:	487	16.776 bytes

Infine si è voluto dimostrare che la classe Parser è una di quelle che occupa più memoria, infatti modificando il metodo di aggiunta delle tuple da New LogicTuple ("t", new Value(i)) a LogicTuple.parse("t"+i) si può notare che tra le classi che occupano più memoria ci sono le classi Token e Parser (2° immagine):



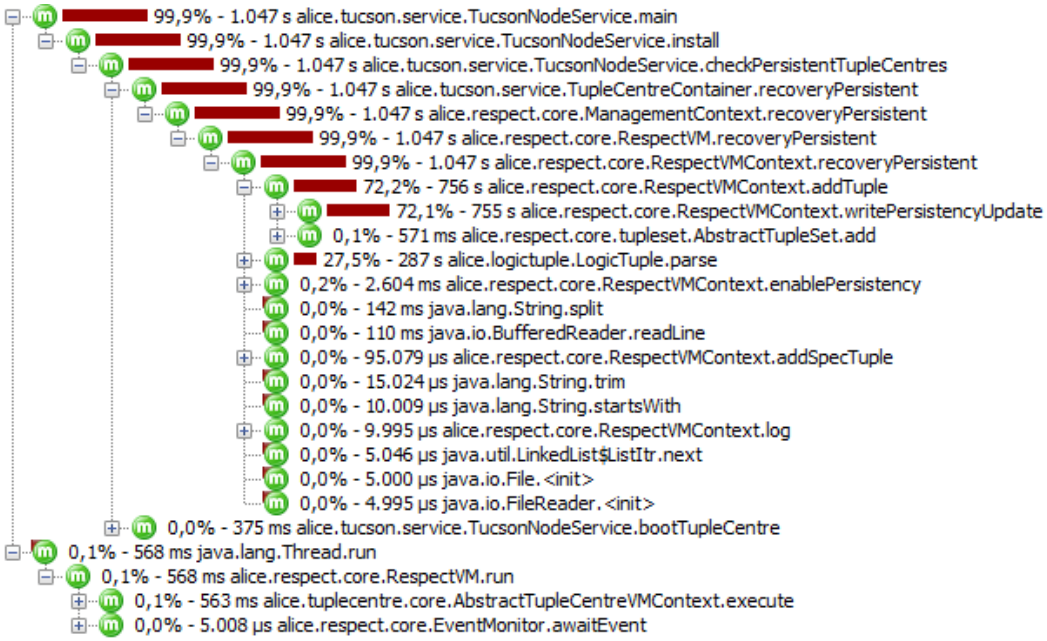
Aggregation level: Classes

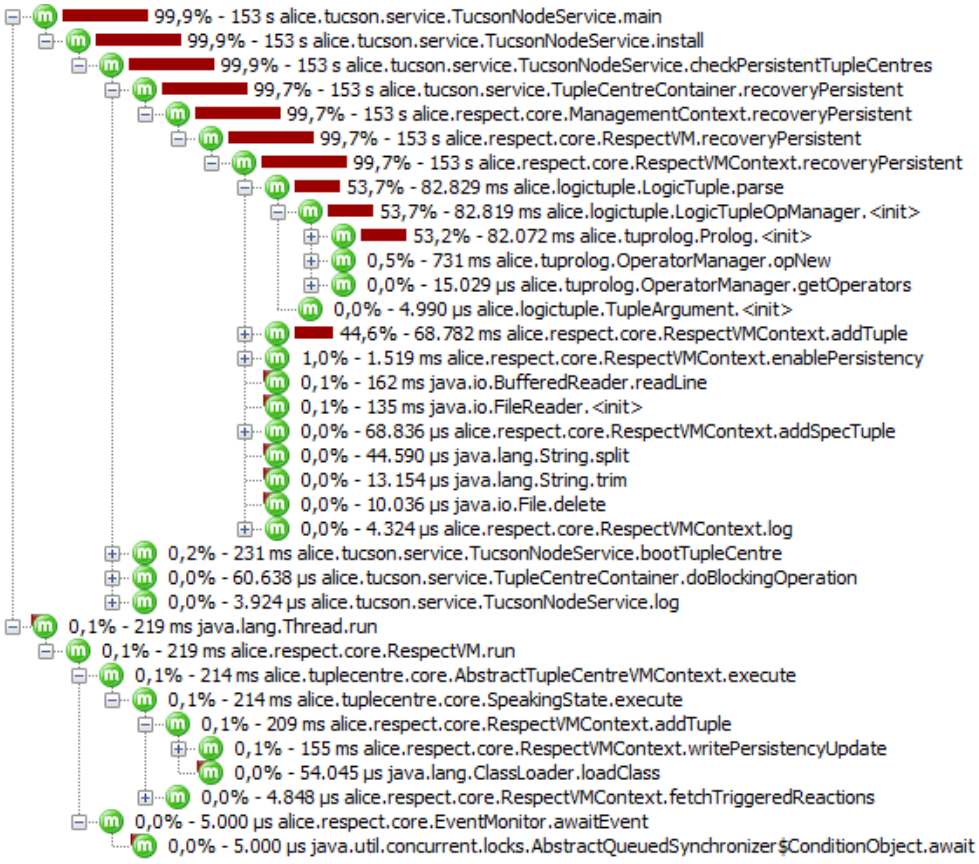
Name	Instance count	Size
alice.tuprolog.Struct	31.381	1.255 kB
alice.tuprolog.Term[]	29.895	745 kB
alice.tuprolog.Var	11.180	447 kB
alice.tuprolog.SubGoalElement	6.325	101 kB
alice.tuprolog.FamilyClausesIndex	5.532	132 kB
alice.tuprolog.SubGoalTree	4.175	66.800 bytes
alice.logictuple.LogicTuple	3.892	62.272 bytes
alice.tuprolog.Int	3.433	54.928 bytes
alice.logictuple.Value	3.323	53.168 bytes
alice.tuprolog.PrimitiveInfo	3.232	103 kB
alice.tucson.network.TucsonMsgRequest	3.161	101 kB
alice.tuprolog.ClauseInfo	3.116	124 kB
alice.tucson.network.TucsonMsgReply	3.006	120 kB
alice.tuprolog.Token	2.432	58.368 bytes
alice.tuprolog.FamilyClausesList	1.844	88.512 bytes
alice.tuprolog.Parser$IdentifiedTerm	1.216	29.184 bytes
alice.logictuple.TupleArgument	1.050	16.800 bytes
alice.tuprolog.Operator	920	22.080 bytes
alice.util.ReadOnlyLinkedList	623	9.968 bytes
alice.tuprolog.ClauseDatabase	369	17.712 bytes
alice.util.OneWayList	350	8.400 bytes
alice.tuprolog.ExecutionContext	345	22.080 bytes
alice.tuprolog.SubGoalStore	345	11.040 bytes
alice.tuprolog.ChoicePointStore	335	5.360 bytes
alice.tuprolog.Engine	335	18.760 bytes
alice.tuprolog.SolveInfo	335	10.720 bytes
alice.tuprolog.event.QueryEvent	323	10.336 bytes
alice.respect.api.TupleCentreId	320	5.120 bytes
alice.tuprolog.ClauseStore	314	10.048 bytes
alice.tucson.api.TucsonTupleCentreId	312	4.992 bytes
Total:	126.470	3.843 kB

Aggregation level: Classes

Name	Instance count	Size
alice.tuprolog.Token	469.928	11.278 kB
alice.tuprolog.Parser$IdentifiedTerm	221.576	5.317 kB
alice.tuprolog.Term[]	149.407	3.741 kB
alice.tuprolog.Struct	141.622	5.664 kB
alice.tuprolog.Var	105.020	4.200 kB
alice.tuprolog.SubGoalElement	22.560	360 kB
alice.tuprolog.FamilyClausesIndex	19.758	474 kB
alice.tuprolog.PrimitiveInfo	11.647	372 kB
alice.tuprolog.SubGoalTree	11.219	179 kB
alice.tuprolog.ClauseInfo	10.921	436 kB
alice.tuprolog.Parser	8.995	287 kB
alice.tuprolog.Tokenizer	8.995	791 kB
alice.tuprolog.FamilyClausesList	6.586	316 kB
alice.tuprolog.Operator	6.283	150 kB
alice.tuprolog.Int	6.205	99.280 bytes
alice.logictuple.TupleArgument	1.717	27.472 bytes
alice.logictuple.LogicTuple	1.615	25.840 bytes
alice.tuprolog.Node	1.160	46.400 bytes
alice.tuprolog.Flag	603	24.120 bytes
alice.tuprolog.Long	403	9.672 bytes
alice.tuprolog.Theory	306	7.344 bytes
alice.tuprolog.StateEnd	268	8.576 bytes
alice.tuprolog.ClauseDatabase	247	11.856 bytes
alice.tuprolog.ClauseDatabase$CompleteIterator	234	5.616 bytes
alice.tuprolog.TermIterator	228	5.472 bytes
alice.tuprolog.event.LibraryEvent	222	7.104 bytes
alice.tuprolog.Tokenizer$PushBack	165	3.960 bytes
alice.util.OneWayList	146	3.504 bytes
alice.tuprolog.OperatorManager$OperatorRegister	121	2.904 bytes
alice.tuprolog.ExecutionContext	89	5.696 bytes
Total:	1.210.665	33.953 kB

e che l'esempio modificato impiega più tempo nel terminare, come possiamo notare dalla differenza dei thread con i rispettivi tempi di esecuzione, nelle successive due immagini (la seconda immagine è quella relativa all'esempio modificato con l'inserimento delle tuple attraverso il metodo parse):





4. Conclusioni

Sicuramente tra le cose positive abbiamo che i thread si comportano in modo corretto quando il main della classe client termina, perché diventano dead thread.
Inoltre gli ACC vengono creati in modo corretto, in particolare ne abbiamo uno lato agente e il corrispettivo lato nodo.

D'altra parte la RespectVM, principale biggest object del sistema, necessiterebbe una profonda rivisitazione da eseguire come un lavoro mirato e attento data la complessità di tale componente.
In aggiunta abbiamo riscontrato un bottleneck nel parsing delle tuple, derivato dalle classi Token e Parser.

Per quanto riguarda i malfunzionamenti, fortunatamente non sono stati constatati strani errori da parte del sistema.