# Asynchronous Agent Behaviour in TuCSoN

*pro active notifications*

Fabio Consalici - Riccardo Drudi

# Asynchronous Agent Behaviour in TuCSoN

*pro active notifications*

## Goal

Questo progetto si inserisce nel contesto dell'infrastruttura di coordinazione TuCSoN basata su centri di tuple. TuCSoN funge da medium di coordinazione per entità attive come gli agenti la cui interazione avviene attraverso operazioni sui centri di tuple.

TuCSoN fornisce agli agenti la possibilità di eseguire operazioni sia sincrone che asincrone, l'obbiettivo del nostro progetto è quello di migliorare il meccanismo di gestione delle operazioni asincrone, mettendo a disposizione del programmatore strumenti che ne semplifichino l'utilizzo e il governo.

## Il Progetto

### Prima implementazione

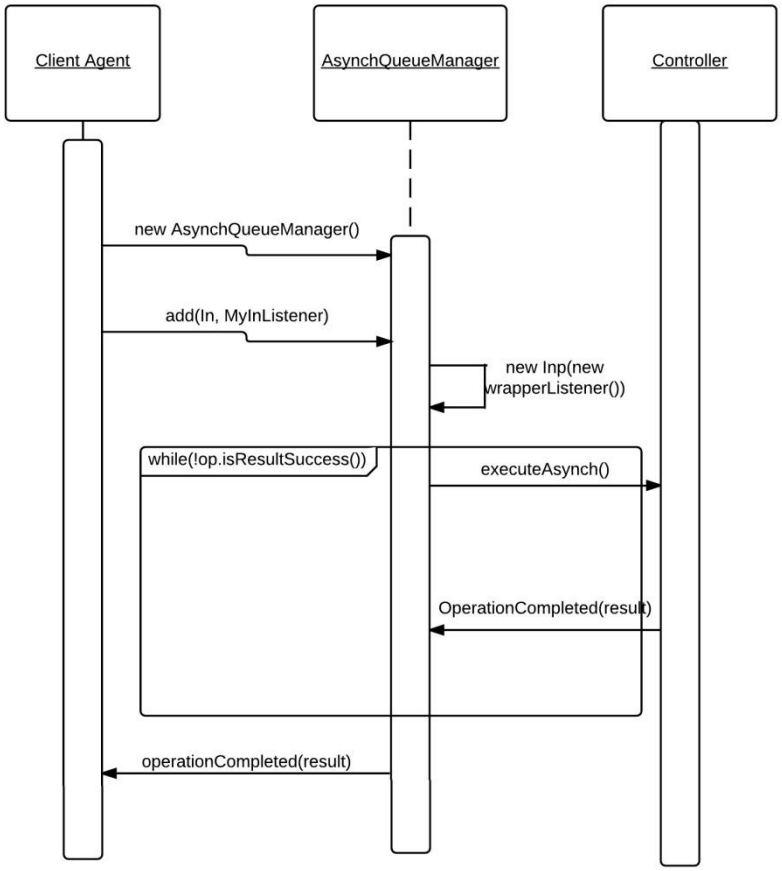
#### Idea di base

Come prima idea abbiamo deciso di realizzare un nuovo agente TuCSoN, l'**AsynchQueueManager** (aqm) il quale si occupa di eseguire ogni tipo di operazione asincrona voluta dal programmatore. Per far ciò il manager è dotato di una coda di operazioni da eseguire che vengono aggiunte man mano dal programmatore.

#### Implementazione

Il programmatore può aggiungere le operazioni da far eseguire al manager attraverso il metodo *add*. In questo metodo si deve anche specificare l'azione da compiere al termine dell'operazione attraverso un listener (*TucsonOperationCompletionListener*).
La coda viene iterata dal manager ed ogni singola operazione viene eseguita finché il client non decide di fermare l'agente.

Per quanto riguarda le operazioni non bloccanti per loro natura (out, inp, ecc ecc), il manager non fa altro che attendere la risposta ed eseguire il relativo listener specificato dal programmatore nell'*add*.
Per le operazioni bloccanti invece si era pensato di mapparle come una serie di corrispettive operazioni non bloccanti; per esempio una In si può mappare in una serie di Inp fino a che una non da esito positivo e solo allora andare ad eseguire il listener del programmatore.




### Problematiche

La soluzione appena proposta si è rivelata inefficace. Uno dei primi problemi sorti è relativo alla necessità da parte del centro di tuple di essere osservabile. Per quanto dal punto di vista dell'agente, e quindi del programmatore, il risultato finale di un mapping di una In in più Inp è ininfluente (non se ne accorge) deve però essere possibile capire dalle richieste pervenute al centro di tuple il meccanismo di base dell'interazione tra agenti. Quindi l'esecuzione di una In da parte dell'agente deve necessariamente essere vista tale anche nel centro di tuple.
Un'altra problematica è relativa all'utilizzo delle reazioni (ReSpecT), quindi se ad una operazione è associata una certa reazione, mappando tale operazione con un'altra/altre si perde la reazione.

## Seconda Implementazione

### Idea di base

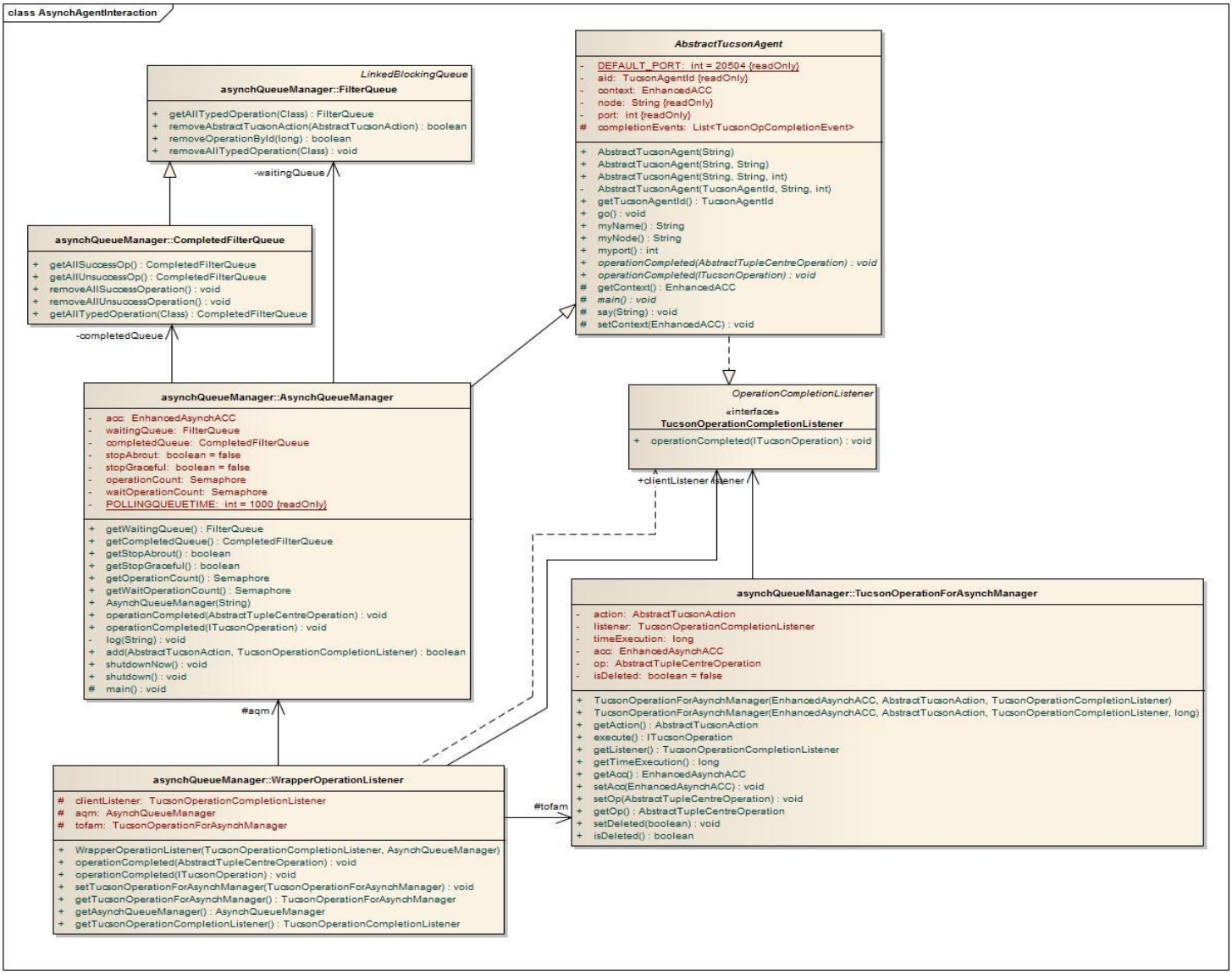
L'idea di base è rimasta in qualche modo la stessa della prima implementazione, si è sempre pensato di creare un nuovo agente TuCSoN chiamato **AsynchQueueManager** (aqm) al quale il programmatore può delegare determinate operazioni che intende eseguire in modo asincrono. Il programmatore oltre a passare l'operazione da eseguire può decidere di passare al manager anche un listener (*TucsonOperationCompletionListener*).

Per la risoluzione delle problematiche riscontrate nella prima implementazione, si è deciso di abbandonare l'esecuzione di operazioni bloccanti in modo sincrono. Ogni operazione bloccante (es: una In) viene eseguita in modo asincrono.

## Implementazione

Per facilitare al programmatore la gestione dei risultati delle operazioni sono state messe a disposizione delle code. Quando il programmatore decide di aggiungere un operazione al manager, questa viene inserita in una coda (*waitingQueue*), una volta che l'operazione viene mandata in esecuzione questa viene eliminata dalla queue, per essere poi inserita in una coda di operazioni completate (*completedQueue*) una volta che viene ricevuta una risposta. Su queste code sono state implementate diverse operazioni per la loro gestione e filtraggio.

L'implementazione dell'AsyncheQueueManager ha prodotto 5 classi principali:




### *AsynchQueueManager*

```
public class AsynchQueueManager extends AbstractTucsonAgent
```

Rappresenta l'agente TuCSoN incaricato di eseguire le operazioni in modo asincrono, mette a disposizione dell'utente diverse operazioni:

`public boolean add(AbstractTucsonAction action,TucsonOperationCompletionListener clientListener)`
Serve a poter inserire un'operzione e il relativo listener nella coda delle operazioni da eseguire

`public void shutdownNow()`
Interrompe l'esecuzione del manager, attraverso questo metodo non vengono eseguite le operazioni in coda, ma il manager rimane comunque attivo fino a che tutte le operazioni eseguite non ricevono una risposta che comunque non verrà eseguita lato client (il listener del client non viene messo in esecuzione).

`public void shutdown()`
Interrompe l'esecuzione del manager, in questo caso però, il manager continua a funzionare finché la coda delle operazioni da eseguire non è vuota. Eseguendo, di conseguenza, i listener delle operazioni.

In entrambi i casi, una volta chiamata la funzione di shutdown non sarà più possibile aggiungere operazioni alla coda dell'Aqm.

### *FilterQueue*

`public class FilterQueue extends LinkedBlockingQueue<TucsonOperationForAsynchManager>`
Rappresenta la coda delle operazioni da eseguire, sono state messe a disposizione dell'utente diverse operazioni:

`public FilterQueue getAllTypedOperation(Class optype)`
restituisce una `FilterQueue` con tutte le operazioni che fanno parte della classe passata come argomento (ad esempio io posso passare come argomento `In.class`)

`public boolean removeAbstractTucsonAction(AbstractTucsonAction action)`
Rimuove dalla coda la specifica azione passata come argomento (se non c'è l'operazione restituisce false)

`public boolean removeOperationById(long id)`
Rimuove dalla coda la specifica azione identificata attraverso il campo ID (se non c'è l'operazione restituisce false)

`public void removeAllTypedOperation(Class optype)`
Rimuove dalla coda tutte le operazioni che fanno parte della classe passata come argomento (ad esempio io posso passare come argomento `In.class`)

### *CompletedFilterQueue*

`public class CompletedFilterQueue extends FilterQueue`
Rappresenta la coda delle operazioni completate, sono state messe a disposizione dell'utente diverse operazioni. Eredita inoltre tutte le operazioni di `FilterQueue`.

`public CompletedFilterQueue getAllSuccessOp()`
Restituisce una lista con tutte le operazioni che hanno avuto successo.

`public CompletedFilterQueue getAllUnsuccessOp()`
Restituisce una lista con tutte le operazioni che non hanno avuto successo.

`public void removeAllSuccessOperation()`

Rimuove dalla lista tutte le operazioni che hanno avuto successo.

```
public void removeAllUnsuccessOperation()
```
Rimuove dalla lista tutte le operazioni che non hanno avuto successo.

### *TucsonOperationForAsynchManager*

```
public class TucsonOperationForAsynchManager
```

Questa entità ci serve a rappresentare un operazione TuCSoN.

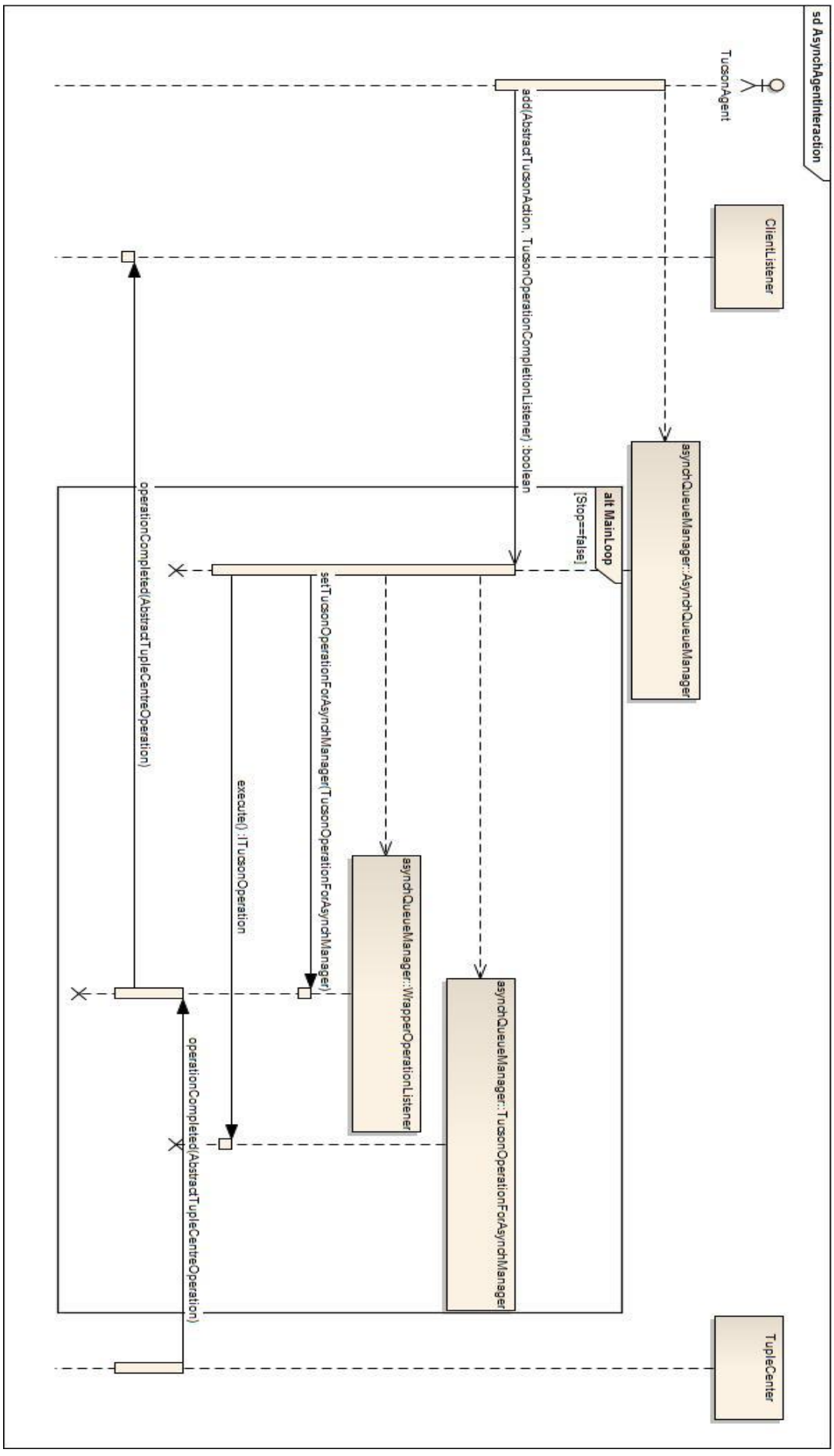
Attraverso un'unica classe intendiamo incapsulare sia una *AbstractTucsonAction* sia una *AbstractTupleCentreOperation*. Ci serve una nozione di *AbstractTucsonAction* per facilitare operazioni di filtraggio sulle code in base alla classe delle operazioni. Inoltre l'utilizzo dell'*AbstractTupleCentreOperation* ci serve per valutare il risultato dell'operazione sempre a livello di filtraggio.

In ultima analisi in questa classe è stato introdotto un campo che ha lo scopo di tenere traccia del tempo di esecuzione dell'operazione. Si può quindi prevedere uno sviluppo futuro del manager con la possibilità di eseguire operazioni asincrone ma con timeout.

### *WrapperOperationListener*

```
public class WrapperOperationListener
        implements TucsonOperationCompletionListener
```

Questa classe aggiunge un layer attorno al listener definito dall'utente. Questo layer è necessario per una corretta coordinazione con l'AsynchQueueManager.
In fase di Shut Down immediato ha il compito di comunicare all'AsynchQueueManager la terminazione di tutte le operazioni pendenti (operazioni mandate in esecuzione di cui attendo la risposta) e di evitare l'esecuzione del listener definito dall'utente.
Inoltre al completamento di un'operazione TuCSoN il wrapper si occupa sia dell'inserimento dell'operazione nella completedQueue sia della sua eliminazione dalla waitingQueue.


sd AsynchAgentInteraction
TucsonAgent
ClientListener
add(AbstractTucsonAction, TucsonOperationCompletionListener) :boolean
asynchQueueManager::AsynchQueueManager
alt MainLoop
[Stop==false]
operationCompleted(AbstractTupleCentreOperation)
setTucsonOperationForAsynchManager(TucsonOperationForAsynchManager)
execute() :ITucsonOperation
asynchQueueManager::WrapperOperationListener
asynchQueueManager::TucsonOperationForAsynchManager
operationCompleted(AbstractTupleCentreOperation)
TupleCenter

## Esempio Guida: PrimeCalcSystem

In questo esempio vogliamo far capire l'utilizzo dell'AsynchQueueManager e delle sue funzionalità. Lo scopo è quello di calcolare il valore della funzione enumerativa dei primi per molti valori. Questo tipo di calcolo è definito computazionalmente pesante, l'idea di base è quindi quella di demandare a più calcolatori il calcolo di questo valore attendendo in maniera asincrona il risultato.

L'esempio si compone di due entità principali: il **MasterAgent** e il **PrimeCalculator**.

Il MasterAgent ha il compito di scegliere l'argomento della funzione enumerativa, una volta fatto questo, utilizza l'aqm per inserire la richiesta di calcolo nel centro di tuple.
Subito dopo richiede i risultati (attraverso operazioni di In utilizzando l'aqm) in maniera asincrona e, di conseguenza, non bloccante. In queste operazioni, viene specificato un listener da eseguire al loro completamento, che non farà altro che stampare il risultato ottenuto.

Attraverso le funzionalità della filterQueue il MasterAgent può controllare l'andamento delle operazioni demandate all'aqm, può quindi controllare quelle eseguite, quelle che hanno avuto successo e quelle che non lo hanno avuto.

Una volta che tutte le risposte sono state ricevute il MasterAgent attraverso l'inserimento di una determinata tupla decide di fermare il funzionamento di tutti i PrimeCalculator.

Infine fa terminare l'aqm attraverso la funzione di shutdown.

I PrimeCalculator (nell'esempio ne vengono fatti partire 3) utilizzano l'aqm per eseguire una catena asincrona di Inp nella quale controllano che nel centro di tuple sia presente una richiesta di calcolo. Alla risposta calcola il risultato della funzione di enumerazione, e ne inserisce il valore nel centro di tuple, successivamente esegue di nuovo un'altra operazione di Inp. Il tutto utilizzando l'aqm.

Eseguono però, senza l'ausilio dell'aqm, e quindi in maniera sincrona (bloccante) la richiesta di una tupla di stop con la quale interrompe la catena di Inp, termina l'aqm e conclude il suo ciclo.

## Conclusioni

Attraverso questo progetto abbiamo compreso sotto il lato pratico il funzionamento di TuCSoN e abbiamo realizzato un'infrastruttura che aiuti il programmatore nell'esecuzione e nella gestione di operazioni puramente asincrone. Attraverso questa funzionalità la gestione delle operazioni asincrone da parte del programmatore risulta molto più semplice e intuitiva grazie alla delegazione dell'esecuzione delle operazioni al manager e alla possibilità di filtrare le diverse code messe a disposizione.

Un possibile sviluppo futuro potrebbe essere l'aggiunta di una nuova funzionalità al manager, dando l'opportunità di eseguire diverse operazioni con un timeOut.