PROBLEMA

Obiettivo:

Realizzare un sistema multiagente per la lettura ed elaborazione di dati meteorologici.

Ci si propone di realizzare un MAS che, almeno in prima versione, soddisfi alcuni dei seguenti goal: ogni agente sia specializzato nell'attingere un tipo di dato specifico da sensori di una stazione meteorologica (in specifico termometro, anemometro, barometro e pluviometro) vi siano agenti specializzati in compiti di più alto livello e dipendenti da agenti più in basso nella gerarchia logica (es.: agenti per calcolare le previsioni, per disegnare il trend delle condizioni, per segnalare allarmi a seguito di condizioni avverse particolari, per creare reports in vari formati: HTML, XML, SMS, ecc.. su richiesta).

ANALISI

Scenario di utilizzo

Il progetto che si intende realizzare trova applicazione in una realtà in cui ancora è scarsa o quasi assente l'utilizzo di sistemi multiagente per monitorare e/o analizzare dati meteorologici. Data la loro enorme mole fino ad ora si è infatti affrontato il problema attraverso grid computing o tecniche di data mining, appoggiandosi a grandi database per depositare i dati. I report e eventuali previsioni venivano effettuate analizzando una gran mole di dati, per lo più con sistemi classici di monitoring affidati ad analisi grossolane. Scopo del progetto è quindi la realizzazione di un'infrastruttura tecnologica mista hardware e software interattiva che in real-time (o quasi) fruisca dati più o meno elaborati a utenti e/o a centri specializzati.

Tutto questo lo si riesce a realizzare grazie ad un approccio diverso al problema.
Il concetto di AGENTE è relativamente recente come applicazione a sistemi complessi reali, ma è quello che ci permette di ottenere quello che vogliamo con costi bassi e ampie prospettive di nuovi servizi "intelligenti". Il progetto dovrà inoltre essere indipendente dalla piattaforma software su cui viene eseguito (Sistema Operativo) e dovrà essere espandibile facilmente e in modo rapido per accogliere nuove funzioni e agenti.

Conoscenze tecnologiche e competenze richieste

- Lato utente

  L'utente che usufruisce dei servizi avrà a disposizione una interfaccia user-friendly che gli permetterà di avere sott'occhio tutti i dati a disposizione, eventuali allarmi e altre informazioni utili quindi, oltre alla conoscenza specifica del significato dei termini legati alla meteorologia, non si richiedono particolari conoscenze.

  Per questo il sistema deve produrre sempre un output apprezzabile e comprensibile anche da chi non conosce approfonditamente le dinamiche del sistema oggetto di studio.

- Lato sviluppatore

  Un eventuale developer che voglia prendere in mano il progetto ed espanderlo e/o modificarlo può farlo attraverso un linguaggio diffuso e intuitivo, che gli permetta di produrre un rapid-prototype per ogni esigenza.
  Da questo punto di vista l'interazione con il linguaggio Prolog ci sembra la scelta migliore. Grazie al pacchetto tuProlog riusciamo a far co-esistere ed interoperare Java e Prolog, lasciando fare computazione vera e propria a quest'ultimo, mentre lasciando a Java le sfaccettature difficili da eseguire in Prolog. Inoltre tale

linguaggio logico è facilmente assimilabile anche da chi non possiede grandi conoscenze di programmazione.

PROGETTO

L'HARDWARE USATO

Il sistema è incentrato prevalentemente nella raccolta di dati metereologici dall'alnalisi dei quali si effettueranno tutte le operazioni degli agenti.

Si è voluto tenere aperto il progetto per l'acquisizione di dati da fonti di qualsiasi natura, siano esse un file piuttosto che un database da cui attingere informazioni, piuttosto che una rete di sensori sparsi sul territorio.

Proprio quest'ultima tipologia ci sembrava più interessante oltre che legata in modo profondo alla natura della metereologia. In primis infatti in ogni sistema meteo vi è un apparato di acquisizione dati da sensori di vario tipo.

Da questo punto di vista siamo stati fortunati in quanto in possesso di una stazione metereologica semi-professionale Oregon Scientific modello WMR-918.
E' una stazione metereologica completa dotata di sensori esterni ed interni:

Dispone di un sensore termometro, barometro, igrometro interno:

un anemometro esterno con misura di velocità, direzione del vento e stima della temperatura di raffreddamento della raffica:

un pluviometro con misura del tasso di piovosità relativo e totale:

Un barometro/teromometro esterno per misurazioni termometriche, barometriche e di punto di rugiada massime e minime:

Ogni sensore è collegato WireLess all'unità centrale lavorando alla frequenza di 433 Mhz ed alimentato da un'unità a cella solare fotovoltaica per alimentare la trasmissione radio.
L'ulità centrale a sua volta è predisposta per una connessione ad elaboratore attraverso una porta seriale RS-232.

➲SCHEMA CONCETTUALE

Scopo di questo progetto è l'inserimento di intelligenza nella gestione dei dati meteo. Questo è stato possibile grazie all'approccio orientato agli agenti. In questo modo non c'è una acquisizione passiva dei dati con una mera visualizzazione degli stessi. Ogni agente del sistema ha un compito particolare e grazie alla cooperazione di essi si giunge a un comportamento emergente che potremmo chiamare "intelligenza".
Il sistema saprà quindi riconoscere se l'analisi delle condizioni fa presupporre l'arrivo di condizioni avverse, se il vento si sta alzando troppo, se la pressione ha un brusco calo (indice di cambiamento delle condizioni atmosferiche) e prendere provvedimenti.
Nel nostro caso ci si è fermati al lancio di un allarme o di una notifica all'utente ma la prospettiva applicativa è ampia.
Si pensi ad esempio al caso di un aeroporto che raccoglie dati da una moltitudine di sensori. L'agente del vento in cooperazione con quello della temperatura e della pressione potrebbero avvertire i cambiamenti in atto ed avvisare che il vento sta mutando di intensità e direzione spostandosi verso X gradi e che la pressione sta calando. Si lancerebbe così una notifica di allerta per eventuali decolli o atterraggi.

Una prima versione del sistema è indicata nello schema seguente. Si è provveduto ad utilizzare come modello di coordinazione ed immagazzinamento temporaneo dei dati, oltre che per lo scambio e la comunicazione fra gli agenti, una piattaforma messa a disposizione dai ricercatori del D.E.I.S. dell'università di Bologna con sede a Cesena: TuCSoN.
Esso sfrutta una nozione dello spazio locale di interazione basato su tuple, denominato centro di tuple, che è uno spazio di tuple con in più la capacità di poter specificare un comportamento. Programmando proprio quest'ultimo in risposta agli eventi, un centro di tuple può comprendere le leggi di coordinazione. Gli agenti possono allora essere progettati indipendentemente dalle differenti architetture dei nodi, secondo un protocollo diretto di interazione.

Come conseguente scelta progettuale si è provveduto ad implementare il tutto in linguaggio Java che oltretutto si integra benissimo e in modo nativo con l'architettura di TuCSoN.

- Acquisizione dei dati:

Abbiamo scelto di predisporre un centro di tuple per accogliere i dati in ingresso, dati che verranno prelevati da un apposito agente che si preoccupa di osservare la porta seriale (COM) dell'elaboratore in attesa di pacchetti.
L'agente utilizza delle librerie Java non presenti nella JavaVirtualMachine standard dette JAVACOMM. Esse ci hanno permesso di avere un driver per seguire gli eventi della porta seriale platform-indipendent (uno dei nostri obiettivi primari). Solo pochi passi sono necessari per configurare a dovere questa libreria in ogni S.O.
Quindi il nostro "AcquireAgent" si preoccuperà di aprire un listener sulla COM e, all'arrivo di un pacchetto interessato, si occuperà del parsing dei pacchetti a seconda della sorgente.
Si è provveduto ad implementare un meccanismo che lascia aperti anche ad altri modelli di stazione metereologica (o fonte in genere).
Nel nostro caso si usa la classe che ci indica le politiche usate nei pacchetti della stazione Oregon WMR-918.
Le specifiche tecniche dei pacchetti si possono trovare quì.
Le informazioni così estrapolate vanno immagazzinate in un centro di tuple come tuple logiche, fonti di conoscienza per altri agenti.

- Visualizzazione dei dati:

Per ogni tipologia di dato si è deciso di predisporre un agente che 'ascolti' sul centro di tuple per la visualizzazione e, all'arrivo della tupla interessata, la prelevi e la visualizzi in un'interfaccia grafica. Gli agenti creati sono i seguenti:

ViewWindAgent	-->	visualizza i dati relativi alle condizioni del vento
ViewStatusAgent	-->	visualizza i dati di stato e di servizio della stazione
ViewRainAgent	-->	visualizza i dati relativi alle precipitazioni
ViewOutAgent	-->	visualizza i dati di pressione, temperatura e umidità esterne
ViewinAgent	-->	visualizza i dati di pressione, temperatura e umidità interne
ViewExtraAgent	-->	visualizza i dati di eventuali sensori extra applicabili
ViewBattAgent	-->	visualizza i dati relativi allo stato delle batterie dei sensori e della stazione centrale

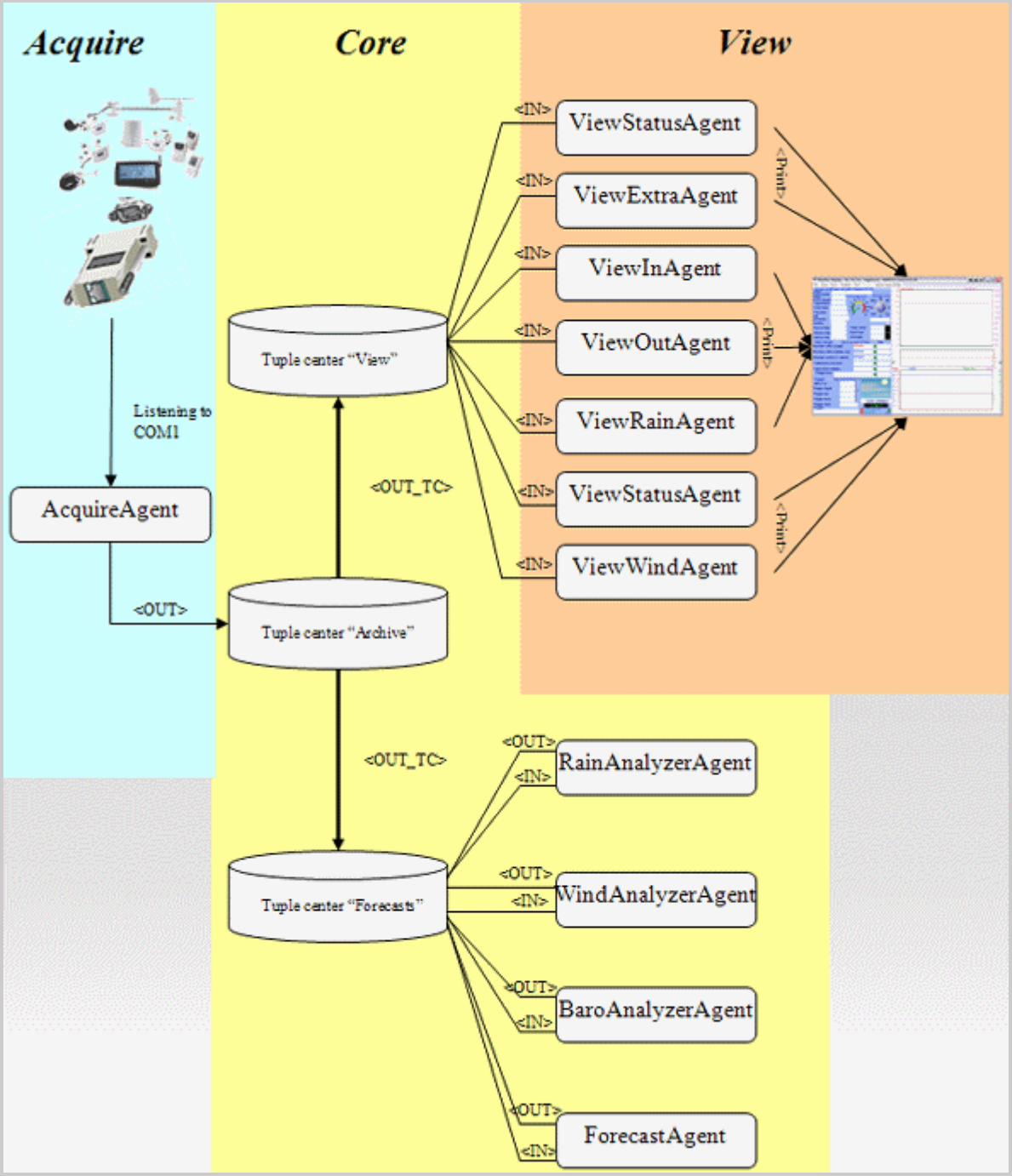
- Agenti di alto livello:

Vi sono poi una serie di agenti che riescono a fornire servizi di più alto livello di una pura ripresa di dati. Essi sfruttano il tuple center come centro di scambio di conoscienza ed elaborazione e si organizzano per prevedere condizioni avverse, per far scattare allarmi o per stimare la tendenza meteo.

RainForecastAgent	-->	analizza i dati relativi alle precipitazioni e compie un elaborazione su 5 misurazioni consecutive calcolando media e varianza.
WindForecastAgent	-->	analizza i dati relativi alle condizioni del vento e compie un elaborazione su 5 misurazioni consecutive calcolando media e inserendo una tupla con essa. Quindi si pone in ascolto su 5 ulteriori campioni di essa. Quindi ci da informazioni sulla media della velocità e sul trend calcolato sulle 5 medie consecutive, con relativa varianza.
BaroForecastAgent	-->	analizza i dati relativi alla pressione e compie un elaborazione ogni 10 misurazioni consecutive calcolando

		il calo di pressione come variazione di essa.
ForecastAgent	-->	ascolta cosa hanno da dire i precedenti agenti e analizza la situazione in tempo reale, accorgendosi se si sta preconfigurando un cambiamento delle condizioni.
AlarmAgent	-->	ascolta le informazioni di ForecastAgent e in caso lancia allarmi a seguito di condizioni avverse confrontando i valori con dei valori threshold (di soglia) che è possibile settare.

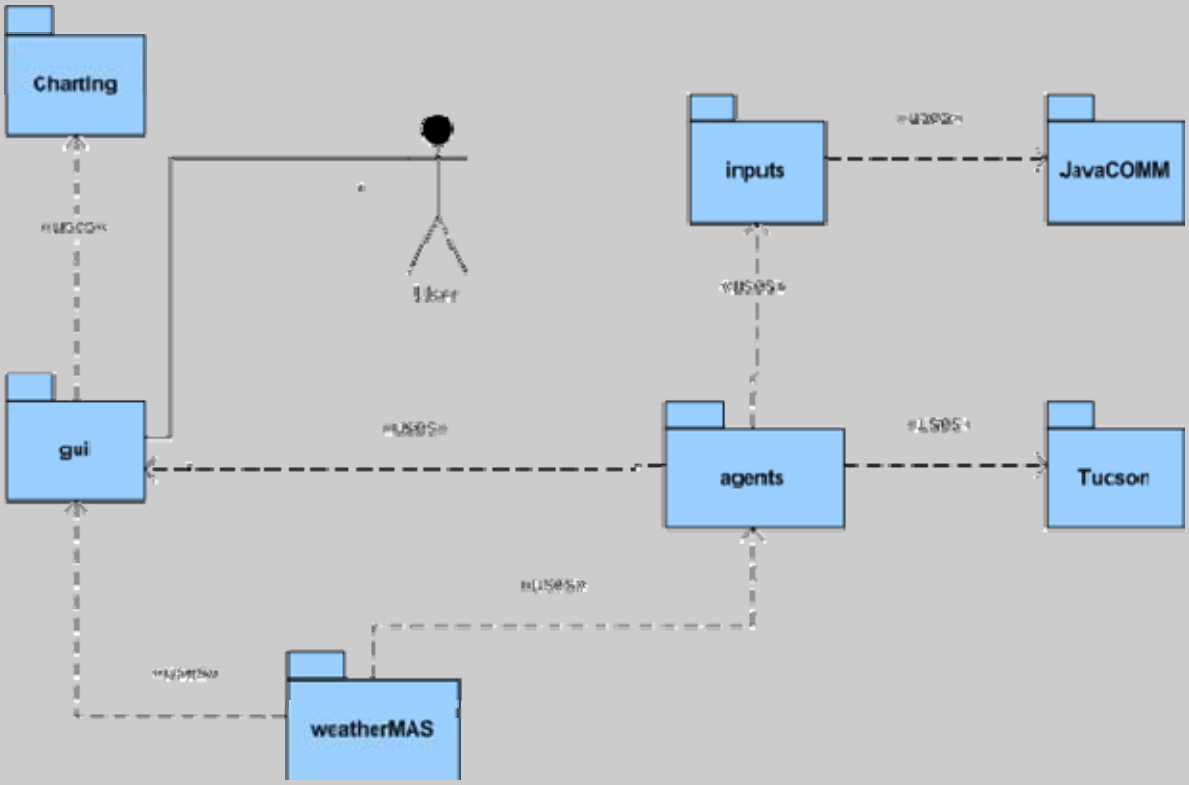
Quello che segue è lo schema concettuale di comunicazione ed interazione fra le varie parti del sistema:



IMPLEMENTAZIONE

DIAGRAMMA DEL SISTEMA

Quello che segue è il diagramma delle classi usato per costruire e codificare il sistema:



IL CORE E L'INIZIALIZZAZIONE

Il cuore del sistema, che gli da forma e consente l'interazione fra le varie parti è l'infrastruttura di TuCSoN.
Si è scelto di attivare tre centri di tuple differenti, scelta puramente progettuale per accentuare la chiarezza. Si sarebbe potuti infatti utilizzare un solo centro di tuple in cui scambiare la conoscienza, ma in questo modo si evidenziano maggiormente le varie tipologie di dati producendo una prima divisione.

In questo modo gli agenti visualizzativi, per esempio, "sporcano" solamente un ambiente dedicato a loro qual'è il tuple centre "VIEW".
Essi infatti hanno un compito unidirezionale e non necessitano di interazioni particolari. Quindi si è attivato un tuple center dedicato solo al loro lavoro.

Si è potuta così realizzare anche una caratteristica che rende TuCSoN unico, ovvero la possibilità di rendere attivo un tuple center facendogli compiere azioni scritte in linguaggio ReSpecT.

Il tuple center ha una specifica in ReSpecT che gli permette di essere reattivo tutte le volte che vede inserita una tupla logica. In particolare ogni qualvolta nota una OUT compie una IN negli altri due tuple centers "VIEW" e "FORECASTS".
La specifica ReSpecT introdotta è la seguente ed è estremamente corta ma efficace:

```
setSpec(reaction(out(T), out_tc(forecasts, T))).

setSpec(reaction(out(T), out_tc(view, T))).
```

di creare ed attivare i tuple centers che gli vengono passati come parametro (in specifico un array di stringhe), di inizializzare l'interfaccia grafica e di istruire i tuole centers con azioni ReSpecT. Inoltre crea e da il via a tutti gli altri agenti menttendoli in esecuzione con la direttiva SPAWN().

```
public class Initializer extends Agent {

...

public Initializer(AgentId arg0, String[] TCtoStart, GuiSwing gui)
throws TucsonException {
super(arg0);
...
}

public void body(){

try {
TucsonContext cn1 = Tucson.enterDefaultContext();

//facciamo partire tutti i tuple center passati
for (int tcCount=0; tcCount<=tCtoStart.length-1; tcCount++){
Tc[tcCount] = new TupleCentreId(tCtoStart[tcCount]);
}

////////////////////////////////////////////////////////////////////////////////
////Il tuple center Archive copia una tupla in ingresso su Forecasts
setSpec(Tc[0], "reaction(out(T),( out_tc(forecasts, T))).");

////Il tuple center Forecasts copia una tupla in ingresso su View
setSpec(Tc[2], "reaction(out(T),( out_tc(view, T))).");

// //Devo farlo in 2 passi a catena sennò non funziona sullo stesso centro
////////////////////////////////////////////////////////////////////////////////

AgentId AcquireAgent=new AgentId("acquireAgent");
new AcquireAgent(AcquireAgent, Tc[0], "wmr918", "COM1").spawn();

AgentId GuiWindAgent=new AgentId("guiWindAgent");
new ViewWindAgent(GuiWindAgent, myGui, Tc[1]).spawn();

AgentId GuiStatusAgent=new AgentId("guiStatusAgent");
new ViewStatusAgent(GuiStatusAgent, myGui, Tc[1]).spawn();

AgentId GuiRainAgent=new AgentId("guiRainAgent");
new ViewRainAgent(GuiRainAgent, myGui, Tc[1]).spawn();

AgentId GuiOutAgent=new AgentId("guiOutAgent");
new ViewOutAgent(GuiOutAgent, myGui, Tc[1]).spawn();

AgentId GuiInAgent=new AgentId("guiInAgent");
new ViewInAgent(GuiInAgent, myGui, Tc[1]).spawn();

// AgentId GuiExtraAgent=new AgentId("guiExtraAgent");
// new ViewExtraAgent(GuiExtraAgent, myGui).spawn();
```

```
AgentId GuiBattAgent=new AgentId("guiBattAgent");
new ViewBattAgent(GuiBattAgent, myGui, Tc[1]).spawn();

}
catch(Exception ex){
System.out.println("Error Initializer :"+ex);
}

}

}
```

GLI AGENTI

AcquireAgent

L'agente che è a monte della conoscienza è AcquireAgent che si preoccupa dell'acquisizione dei dati preleva pacchetti dalla porta seriale, effettua un parsing di essi e li trasforma in tuple logiche da inserire nel tuple center "ARCHIVE".
Nei seguenti pezzi di codice illustreremo la procedura di lettura dalla porta seriale attraverso le librerie JavaCOMM, il settaggio dei parametri della porta e di un listener che resta in scolto per notificare l'arrivo di un pacchetto:

```
public static void ReadCom(String WeatherStationModel) {
try {
inputStream = PortaSeriale.getInputStream();
out = PortaSeriale.getOutputStream();
PortaSeriale.addEventListener(new commListener());
}
catch (Exception e) {
System.out.println("Errore ReadCom: "+e);
}

PortaSeriale.notifyOnDataAvailable(true);

try {
PortaSeriale.setSerialPortParams(9600,
SerialPort.DATABITS_8,
SerialPort.STOPBITS_1 ,
SerialPort.PARITY_NONE);
}

catch (UnsupportedCommOperationException e) {
}
}
```

Nel codice seguente illustriamo l'azione del listener della porta seriale che preleva il pacchetto in entrata e lo immagazzina in un buffer; quindi per tutta la sua lunghezza converte il contenuto da binario a stringa, per poi passarlo al parser insieme alle informazioni sul tipo di stazione meteo in uso e sul tuple center usato:

```
public static class commListener implements SerialPortEventListener{
```

```
public void serialEvent(SerialPortEvent event) {
if(event.getEventType()==SerialPortEvent.DATA_AVAILABLE){
try
{
int numBytes = 0;
while (inputStream.available() > 0)
{
strReadBuffer="";
numBytes = inputStream.read(binReadBuffer);
for (int i=0; i<20; i++)
{
strReadBuffer+=byteToStr(binReadBuffer[i])+" ";
}

}
count++;
try {
parseStream(strReadBuffer, WeatherStationModel, Tc);
}
catch (Exception e){System.out.println("Errore del parseStream: "+e);}
}
catch (IOException e){System.out.println("Errore ComListener");}
}
}
}
```

Di seguito la funzione ParseStream prende il buffer letto e a seconda del modello usato per l'acquisizione passa il tutto al parser specializzato. Prima però effettua un controllo sul pacchetto.
Ogni pacchetto inizia obbligatoriamente con un pacchetto di 2 byte con il contenuto "ff". Se questo non è il nostro caso si sincronizza la lettura nello stream semplicemente ignorando il pacchetto.

```
public static void parseStream(String strReadBuffer, String
WeatherStationModel, TupleCentreId tc){

StringTokenizer buff = new StringTokenizer(strReadBuffer);

//choose the Weatherstation Type
if (WeatherStationModel.compareTo("wmr918")==0){
String packetBody="";
String curToken = buff.nextToken();
if (curToken.compareTo("ff")!=0){
}
else{
curToken = buff.nextToken();
curToken = buff.nextToken();
int packetType = takePacketTypeNumber(curToken);
while(buff.hasMoreTokens()){
packetBody += buff.nextToken()+" ";
}
try{
WeatherStations.oregonScientificWmr918(packetBody, packetType, tc);
} catch (Exception e){System.out.println("\nParsing errato: "+e);}
```

```
}
}
else {
System.out.println("\nThis Weather Station is not actually supported");
}

}
```

ViewBattAgent
ViewExtraAgent
ViewInAgent
ViewOutAgent
ViewRainAgent
ViewStatusAgent

Tutti questi agenti hanno una struttira simile e sono gli agenti "visualizzatori".
Essi si pongono in ascolto sul tuple center "VIEW" ognuno specializzandosi sul proprio tipo di tupla richiedendo una IN.
Quando una tupla "fa match" con la richiesta posta, l'infrastruttura di TuCSoN glie la cede ed essi si preoccupano di settare sulla GUI i campi di loro competenza attraverso una funzione setField() messa a disposizione proprio dalla classe GuiSwing:

```
public class ViewXXXXXXAgent extends Agent {

...

public ViewXXXXXXAgent(AgentId arg0, GuiSwing gui,
TupleCentreId tid) throws TucsonException {
super(arg0);
....
}

public void body(){
....
while (true){
try {
LogicTuple Battery = in(Tid, new LogicTuple("battery", new
Var("Data"),
new Var("BatteryStatus")));

....
myGui.setField(XX, fieldToSend);
myGui.updatePacketCount();
myGui.updateCurrentPacketTime(XXXXX.getArg(0).toString());
....

}
} catch (Exception ex){System.out.println(ex);}
}
}
}
```

WindForecastAgent

BaroForecastAgent

Tutti questi agenti hanno una struttira simile e sono gli agenti addetti alle previsioni. Essi si pongono in ascolto sul tuple center "VIEW" ed analizzano i dati costruendo medie e varianze sui dati.
A loro volta depositano tuple con informazioni sull'andamento del vento, delle raffiche, della pioggia, del tasso di piovosità, della pressione e del tasso di variazione di essa:

```
public class WindAnalyzerAgent extends Agent {

private TupleCentreId Tid;

/**
* Default constructor of the class
* @param arg0 AgentId - the id of the Agent
* @param gui the gui used
* @param tid the tuple center used
* @throws TucsonException
*/
public WindAnalyzerAgent(AgentId arg0, TupleCentreId tid) throws TucsonException
{
super(arg0);
Tid = tid;
}

public void body(){
while (true){
try {
startAnalisis();
} catch (Exception ex){System.out.println(ex);}
}
}

/**
* This void start the analisis of the wind analyzer agent
* and interact with the tuple center forecasts
*/
public void startAnalisis(){

...

for (timesToReadWindMedia=0; timesToReadWindMedia<var;
timesToReadWindMedia++ ){
try {
TucsonContext context = Tucson.enterDefaultContext();
for (timesToReadWind=0; timesToReadWind<var; timesToReadWind++ ){
LogicTuple Vento = in(Tid, new LogicTuple("vento",
new Var("Data"),
new Var("GustDirection"),
new Var("GustSpeed"),
new Var("AverageSpeed"),
new Var("WindChill")));
```

```
if (Vento==null){} else {
AvgSpeed = new Float(Vento.getArg(3).toString());
AvgWindSpeed+=AvgSpeed.floatValue();
}
}
AvgWindSpeed = AvgWindSpeed/var;

context.out(Tid, new LogicTuple("ventoMedia",new Value(AvgWindSpeed)));

} catch (Exception ex){System.out.println(ex);}


try {
TucsonContext context = Tucson.enterDefaultContext();

LogicTuple VentoMedia = in(Tid, new LogicTuple("ventoMedia",
new Var("Media")));

if (VentoMedia==null){} else {
currValueFloat = new Float(VentoMedia.getArg(0).toString());
currValue[timesToReadWindMedia]=currValueFloat.floatValue();
//tieni 5 rilevamenti

}
} catch (Exception ex){System.out.println(ex);}

}
// //se il trend è superiore a X lancia 1 avviso altrimenti tutto ok
//se l'avviso è superiore alla soglia di allarme lancia un allarme altrimenti notifica
for (timesToReadWindMedia=0; timesToReadWindMedia<var;
timesToReadWindMedia++ ){
avgWind+=currValue[timesToReadWindMedia];
}
avgWind = avgWind/var;

for (timesToReadWindMedia=0; timesToReadWindMedia<var;
timesToReadWindMedia++ ){
avgWindVarianza+=(currValue[timesToReadWindMedia]-avgWind);
}
avgWindVarianza = avgWindVarianza/var;
try {
TucsonContext context = Tucson.enterDefaultContext();
context.out(Tid, new LogicTuple("windTrend",new Value(avgWind),new
Value(avgWindVarianza)));

System.out.println("\n"+currValue.toString()+"\n"+avgWind+"\n"+avgWindVarianza);
} catch (Exception ex){System.out.println(ex);}
}



}
```

AlarmAgent

Questo agente legge le informazioni proposte dagli agenti di previsione e si preoccupa di lanciare allarmi a seguito di lettura di dati che oltrepassano soglie di guardia prefissate. E' prevista anche la visualizzazione in un pannello della GUI dei messaggi di avviso sugli allarmi lanciati:

```
public class AlarmAgent extends Agent {

...


/**
* Default constructor of the class
* @param arg0 AgentId - the id of the Agent
* @param gui the gui used
* @param tid the tuple center used
* @throws TucsonException
*/
public AlarmAgent(AgentId arg0, GuiSwing gui, TupleCentreId tid)
throws TucsonException {
super(arg0);
myGui = gui;
Tid = tid;
}

public void body(){
while (true){
try {

startAnalisis();

} catch (Exception ex){System.out.println(ex);}
}
}

/**
* This void start the analisis of the alarm agent
* and interact with the tuple center forecasts
*/
public void startAnalisis(){
String[] fields = new String[10];

try {
TucsonContext context = Tucson.enterDefaultContext();

LogicTuple WindAlarm = inp(Tid, new LogicTuple("windAlarm",
new Var("Vento")));

if (WindAlarm==null){
fields[0]="0";
} else {
fields[0]=WindAlarm.getArg(0).toString();
}

LogicTuple WindRafficaAlarm = inp(Tid, new
LogicTuple("windRafficaAlarm",
```

```
new Var("Raffica")));

if (WindRafficaAlarm==null){
fields[1]="0";
} else {
fields[1]=WindRafficaAlarm.getArg(0).toString();
}

LogicTuple RainAlarm = inp(Tid, new LogicTuple("rainAlarm",
new Var("Pioggia")));

if (RainAlarm==null){
fields[2]="0";
} else {
fields[2]=RainAlarm.getArg(0).toString();
}

LogicTuple RainAlluvioneAlarm = inp(Tid, new
LogicTuple("rainAlluvioneAlarm",
new Var("Alluvione")));

if (RainAlluvioneAlarm==null){
fields[3]="0";
} else {
fields[3]=RainAlluvioneAlarm.getArg(0).toString();
}

LogicTuple BaroAlarm = inp(Tid, new LogicTuple("baroAlarm",
new Var("Pressione")));

if (BaroAlarm==null){
fields[4]="0";
} else {
fields[4]=BaroAlarm.getArg(0).toString();
}

LogicTuple BaroSbalzoAlarm = inp(Tid, new
LogicTuple("baroSbalzoAlarm",
new Var("SbalzoPressione")));

if (BaroSbalzoAlarm==null){
fields[5]="0";
} else {
fields[5]=BaroSbalzoAlarm.getArg(0).toString();
}
myGui.setField(16, fields);
} catch (Exception ex){System.out.println(ex);}

}

}
```

➲LA CLASSE INPUTS

Nel nostro progetto abbiamo usato una stazione metereologica Oregon Scientific WMR-918 ma ve ne sono in commercio moltissime altre. Per questo abbiamo lasciato il progetto aperto

anche ad altri modelli.
La classe inputs si preoccupa di implementare la fase di parsing dei pacchetti di ogni stazione in quanto ognuna ha il proprio formato, quindi risultava inutile una analisi per riassumerne le caratteristiche a fattor comune.

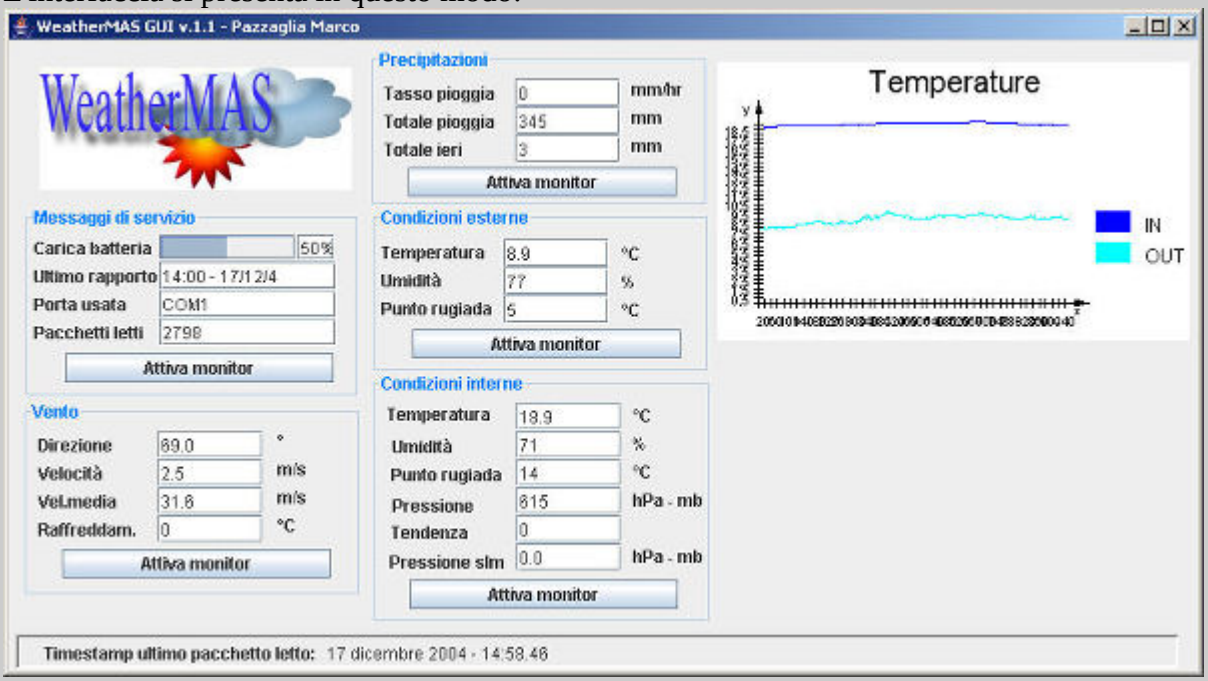
Ad ogni pacchetto inoltre viene associato un TimeStamp univoco per riconoscerlo e per ricostruire una cronologia.

Il pacchetto quindi viene classificato per tipologia di appartenenza e quindi viene creata una tupla logica per l'inserimento sul tuple center:

Packet Type 00:	Misurazioni sul vento
Packet Type 01:	Misurazioni sulle precipitazioni
Packet Type 02:	Condizioni esterne (sensori extra)
Packet Type 03:	Condizioni esterne
Packet Type 06:	Condizioni interne
Packet Type 0E:	Stato, messaggi di servizio vari
Packet Type 0F:	Condizioni carica batteria e bollettino orario

➲L'INTERFACCIA GRAFICA

La GUI è implementata nella classe GuiSwing e, come è intuibile, utilizza le librerie javax.swing per essere creata.
Sono inoltre presenti funzioni per il settaggio dei campi, utilizzate dagli agenti visualizzatori (setField), per contare i pacchetti in entrata, per aggiornare la situazione del/dei grafico/i.
L'interfaccia si presenta in questo modo:



Nella parte sinistra vi è la situazione dei sensori e i dati acquisiti, nella parte destra vi è un grafico che si aggiorna in real-time con l'assunzione degli stessi.

Vi sono pulsanti sotto ogni settore per attivare l'agente responsabile per la visualizzazione di quel tipo di dato.