Terrasense ArborXT

A Forest Data Collection System

Summary of design fest activities, with meetings held at September 14, 22 and 23.

Table of Contents

Design team:

Nelson Lago: lago@ime.usp.br

Cristiano Breuel: cmbreuel@ime.usp.br

Lene Hallen: lene@stud.ntnu.no

Giuliano Mega: giuliano@ime.usp.br - **Moderator**

Henning: hennine@stud.ntnu.no

Pedro T: plt@ime.usp.br

Fábio Miranda: fmiranda@gmail.com – **Record keeper**

Problem:

We worked on the Forest Data Collection problem, a system to gathering and analyzing weather information and to predict forest fires and help with water table measurement. This system runs in a central computer and communicates with sensors of a variety of types (temperature, sunlight intensity, wind speed and direction, rainfall, humidity) distributed in forests. Data corresponding to the readings of the sensors is stored in the central computer and users run software that uses such data to create reports and perform analyses and predictions over the data.

In this design process we focused on the programs that maintain the configuration of functional field sensors and gather sensor data, storing it in a database.

The Process

First day:

Initially two members volunteered as moderator and record keeper, and everyone agreed. Then everybody read the problem description for the second time (the first time was prior to forming the teams).

A discussion of the system in architectural terms was held, taking into account aspects of the subsystems that would be needed, and a rough sketch of the system was made.

A *scenario playing* phase began, during which we would confront the initial sketch of the system against requisites and watch for unlooked aspects of the system, revising our design.

We had almost finished covering the “Data Analysis” scenario when, for some reason, it seemed to the members of the group that it was highly important to notify certain types of analysis plug-ins of certain abnormal events. A sudden steep raise in the temperature of a whole region is an example of the types of events we were worried about. Shortly after this concern appeared, our time for that day (2h) was over.

Second day

A lot of discussion and design was spent looking at the plug-in notification problem, and we ended up a part of the system that nobody really liked.

Since our time was getting short, we defined a list of deliverables that could probably be done within the remaining time:

- Core assumptions we have used during the discussion to shape the system
- An initial class model with the main entities in the system
- A diagram showing the class interactions
- A list of interfaces of the system to other systems
- A compilation of the lessons learned

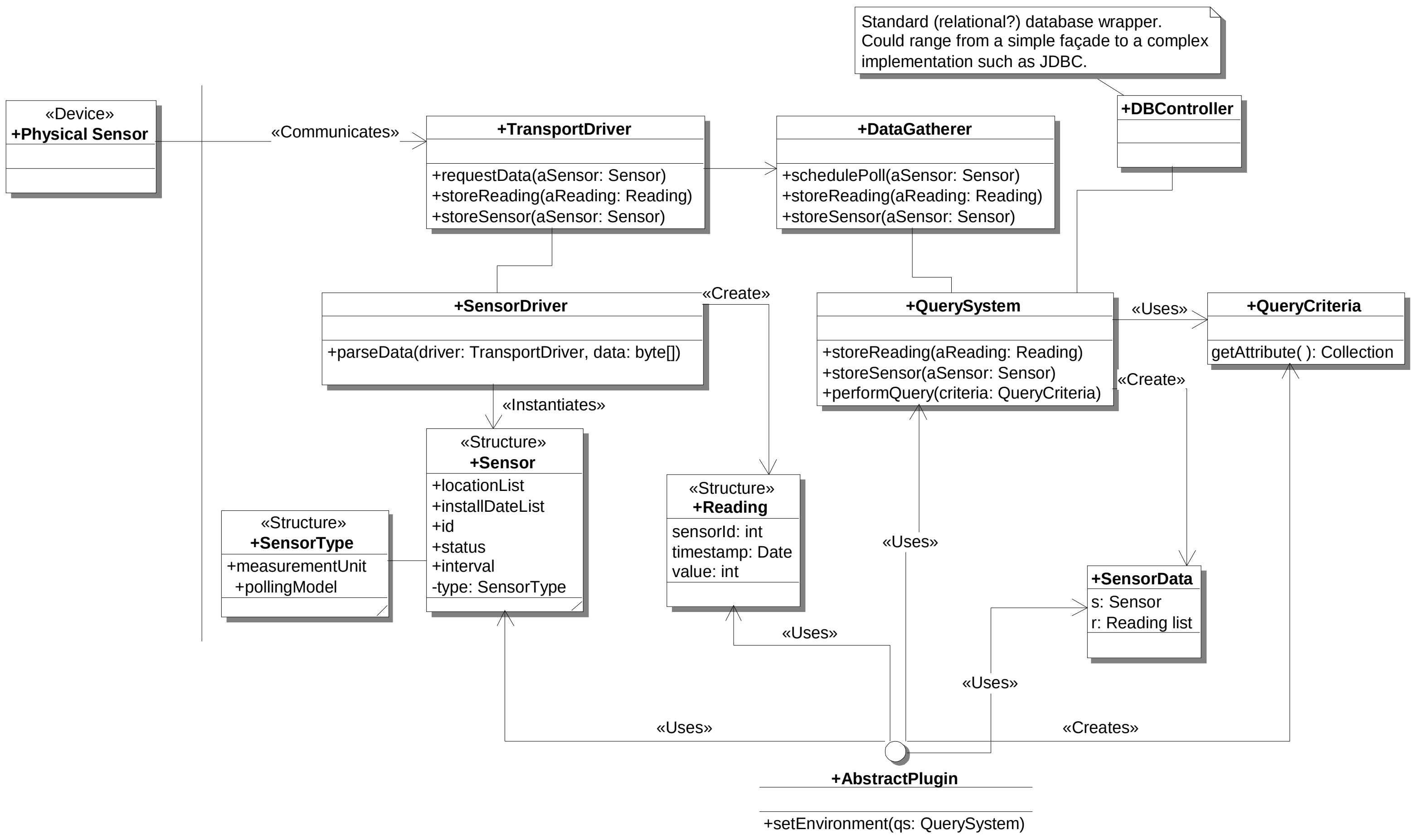
During the rest of the time we had during that day, we started working on the class model and designing the subsystem that would cope with the variety of data communication links. It became evident that we would need a little more time to finish it properly.

Third day

The group met again for a third two-hour design session, mainly to define class interactions, check the system against usage scenarios (this was left aside during the second day) and organize for the presentation (here are the slides presented). We ended up dropping the notification system that haunted us in the previous day because it wasn't in the requirements and it didn't feel right during the definition of class interactions (we felt that that part of the system was compromising the quality of the design of every class that interacted with it).

The final system ended up becoming far simpler than several previous versions of it.

Class Model


Standard (relational?) database wrapper.
Could range from a simple façade to a complex implementation such as JDBC.
«Device»
+Physical Sensor
«Communicates»
+TransportDriver
+requestData(aSensor: Sensor)
+storeReading(aReading: Reading)
+storeSensor(aSensor: Sensor)
+DataGatherer
+schedulePoll(aSensor: Sensor)
+storeReading(aReading: Reading)
+storeSensor(aSensor: Sensor)
+DBController
+SensorDriver
+parseData(driver: TransportDriver, data: byte[])
«Create»
+QuerySystem
+storeReading(aReading: Reading)
+storeSensor(aSensor: Sensor)
+performQuery(criteria: QueryCriteria)
«Uses»
+QueryCriteria
getAttribute( ): Collection
«Create»
«Instantiates»
«Structure»
+Sensor
+locationList
+installDateList
+id
+status
+interval
-type: SensorType
«Structure»
+SensorType
+measurementUnit
+pollingModel
«Structure»
+Reading
sensorId: int
timestamp: Date
value: int
«Uses»
+SensorData
s: Sensor
r: Reading list
«Uses»
«Uses»
«Uses»
«Creates»
+AbstractPlugin
+setEnvironment(qs: QuerySystem)

Sensor

This class represents sensors managed by the system, there will be a *Sensor* instance stored in the application for each sensor in the field. *Sensors* objects can be created upon reception of a sensor configuration from a sensor in the field, or by manual configuration done by a user of the system through a plug-in.

Objects of the sensor class are stored in the database and can be retrieved by the plugins through the query interface.

SensorType

Each *Sensor* object has an attribute called *type* that specifies what kind of sensor it is. This attribute is an instance of class *SensorType*, and has information about the measurement unit and the polling model of the sensor (whether it sends its data or has to be queried by the system).

TransportDriver

Sensors communicate with the central system through a variety of communication links (packet radio, dial-up phone, dedicated line). Each type of communication link needs a specific piece of software capable of receiving a stream of data from link and extracting,with the help of a *SensorDriver,* packets that contain sensor readings from it, such piece of software is the *TransportDriver.*

Instances of *TransportDriver* are responsible for receiving data from the sensors in the field, and also enclose mechanisms that enable them to query the sensors that are not capable of automatically sending their data.

SensorDriver

A *SensorDriver* is capable of parsing a stream of data received from the communication link through a TransportDriver and extract either sensor readings or sensor configurations from it. A given *SensorDriver* builds two types of objects based on the data received:

- *Sensor* objects – When a sensor in the field sends its configuration through a communication link, a *TransportDriver* asks a *SensorDriver* to build a *Sensor* object to represent that sensor.
- *Reading* objects – A *Reading* instance corresponds to a sensor reading, a measurement made by a given sensor.

DataGatherer

The *DataGatherer* is responsible for:

- Gathering data from sensors in the field – It is capable of receiving data from sensors that send it automatically (“push” sensors) as well as scheduling polls of sensors that have to be asked for their data.
- Updating the database – it asks the *QuerySystem* to store new information as it arrives.

DBController

DBController stores in permanent storage information about the sensors in the field and all sensor readings. It can be a wrapper for a standard relational database or some other persistance mechanism.

QuerySystem

This class has the responsibility of storing sensor readings and keeping reference to sensors in the system. It can be queried by the plug-ins and returns a collection of objects representing sensors (instances of the *Sensor* class) and sensor readings (instances of the *SensorData* class) based on some selection criteria.

AbstractPlugin

The system should provide connection points to allow the use of plug-ins. The problem specification mentions analysis plug-ins as a requisite. Such plug-ins analyze the result of queries and do tasks like forest fire risk computation, short-time weather prediction, etc.

Each plug-in must implement a setEnvironment method that allows it to connect to the QuerySystem and start issuing queries in the form o QueryCriteria. The plug-ins will receive data in the form of a collection of SensorData objects.

SensorData

SensorData models the result of queries made to the *QuerySystem*. Each *SensorData* packs a *Sensor* and a list of some of its *Readings*. This class is necessary to put constraints in the number of Readings associated with a given Sensor that comprise the result of a query to the *QuerySystem*, since a *Sensor* can potentially be associated with several years of *Readings*.

QueryCriteria

QueryCriteria represents selection criteria passed by the plug-ins to the query system to perform queries. This part of the design of the system could be better refined in future iterations of the design process.

Assumptions

During design sessions a number of assumptions were made to provide orientation to the project. These assumptions are summarized here:

- A group of sensors is the result of a query to the set of sensors. It is a responsibility of the plug-ins to store this grouping information (what it does is storing the query that generates que group).

- There is a standardized message format and data sent by sensors complies with such format. Packets sent by sensors contain a sensor id, a value of the sensor reading and a timestamp.

- All sensor readings are stored in a database, in the format described in the assumption above.

- Plug-ins embed functionality that allows them to build query criteria and pass them to the query system

- The plug-ins are able to run as many threads on their on as they need. In case of plug-ins that analyze data that has been collected after the program has started, it's the plug-in's problem to issue new queries and make sure it always has recent data to analyze.

- The sensors that are capable of announcing their presence do that announcement through a protocol that is understood by a specific sensor driver, already present in the system

- Our system doesn't have specific functionality to assemble random sets of sensors of a given area (see “Data Analysis” scenario in the problem description). In that scenario, we assume the ranger should have selected all the sensors of the area and assembled the groups previously by hand.

- The system doesn't have to provide a separate model for sensor arrays, they're just a group of sensors that happens to be in the same GPS coordinates.

- A sensor stores a list of locations and timestamps of when it was put in that locations for the first time. This information is useful important when relocating sensors, to avoid associating past readings of sensors to their new location.

- Data packets arriving from the sensors are small. When a *TransportDriver* calls a *SensorDriver*, it always has complete chunks of data to ask for it to parse. There's not a case where a *SensorDriver* is requested to parse part of a message.

Class interactions through usage scenarios

Data analysis

In this scenario a user of the system performs a query to groups of sensors that are in a place called Rumbling Range National forest.

One of our assumptions state that groups of sensors are built by users of the system, and such functionality is provided by plug-ins, so it's not a responsibility of ArborXT to support such functionality directly, only provide and interface through which plug-ins can make queries to que central sustem.

This scenario states that the frequency at which a sensor reports is part of a sensor's configuration and is transmitted to the central site upon deployment. Although this does not happen during the course of action of this scenario, it is a functionality that the system is required to support as a precondition of this scenario and is shown in Figure 1.

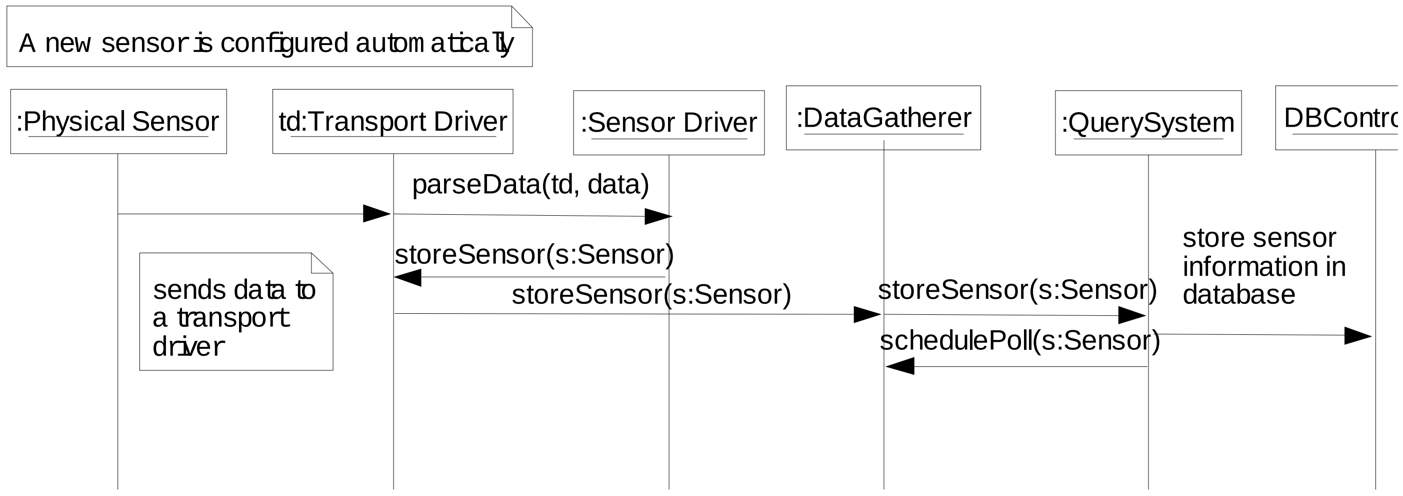
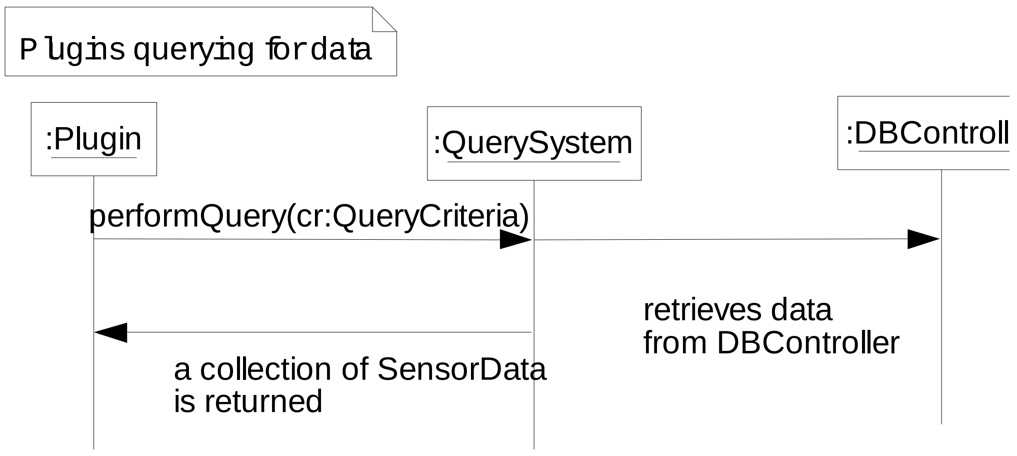


Figure 1 – Sequence diagram showing a sensor that sends its configuration to the central site being configured automatically

During the rest of this scenario, the park ranger (the user) uses information retrieved from queries to the central site to either evaluate the risk of fires inside the region of interest or find out sensors that have not reported for a given period of time to send a repair crew to their sites. Both functionalities use the central system respond to queries that lists of *SensorData* return to the plugins so that the required analyses can be performed on the data.



Plugins querying for data
:Plugin
:QuerySystem
:DBControll
performQuery(cr:QueryCriteria)
retrieves data
from DBController
a collection of SensorData
is returned

Figure 2 - Sequence diagram showing a plugin performing queries to the query system

Installation of new sensors

In this scenario two arrays of sensors are installed by a crew in the forest, and these sensors keep sending data prior to being configured the system users. Arrays of sensors are defined as a group of colocated sensors, so the system does not need to make any distinction whether sensors are part of an array or not.

Though the sensors have not been configured, there must exist some data communication link between the deployed sensor and the central site. Consequently, there already is a *TransportDriver* receiving data through such link and a *SensorDriver* capable of building *Reading* data from objects in that stream. This can be seen in Figure 3.

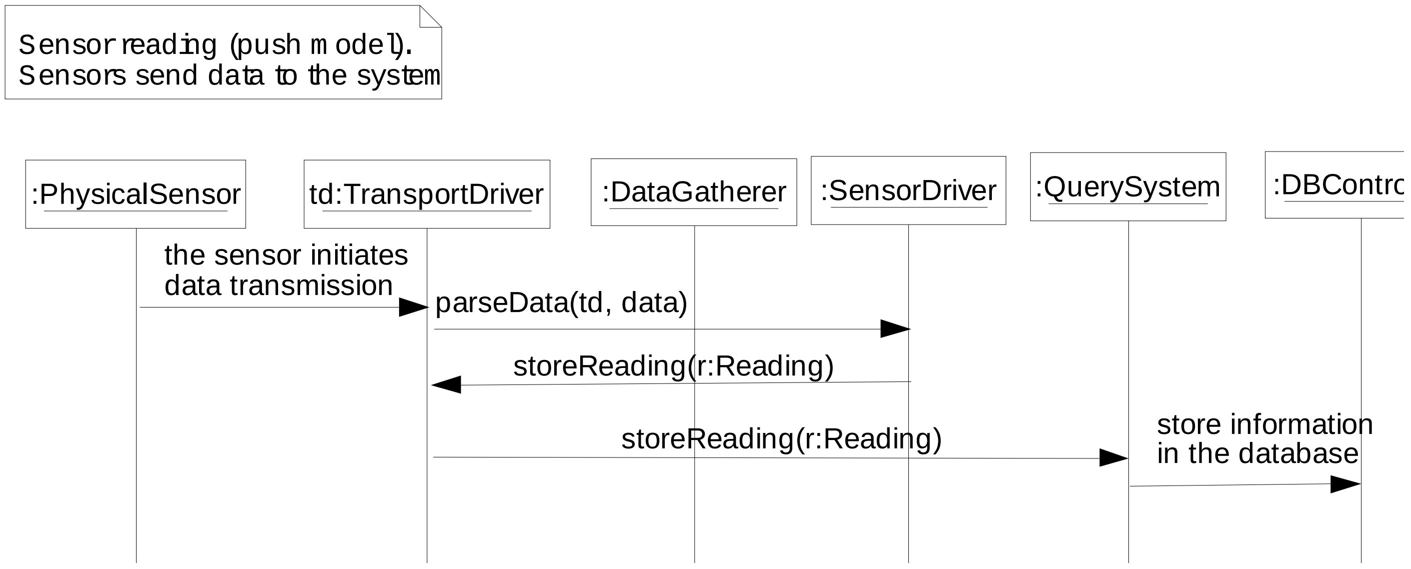


Figure 3 - Sensors actively send their data to the central system

It is worth noticing that the format of *Reading* objects are the same regardless of the sensor that sends them, this makes is possible to store this information before knowing which specific sensor is responsible for the *Reading*. Each *Reading* object keeps the id of the *Sensor* that made the *Reading*. Later, when a sensor is configured, all *Reading* objects that have the same id can be related to it.

In this usage scenario the configuration of the sensor is made by hand. The system provides these functionality through the use of plugins that create Sensor objects and send them to the *QuerySystem* (by passing them as an argument to the *storeSensor* method) to make them available to the system. The *QuerySystem* in turn registers the newly created sensor with the *DataGatherer* so that polls to the newly configured sensor can be scheduled should it be necessary. This interaction can be seen in Figure 4.

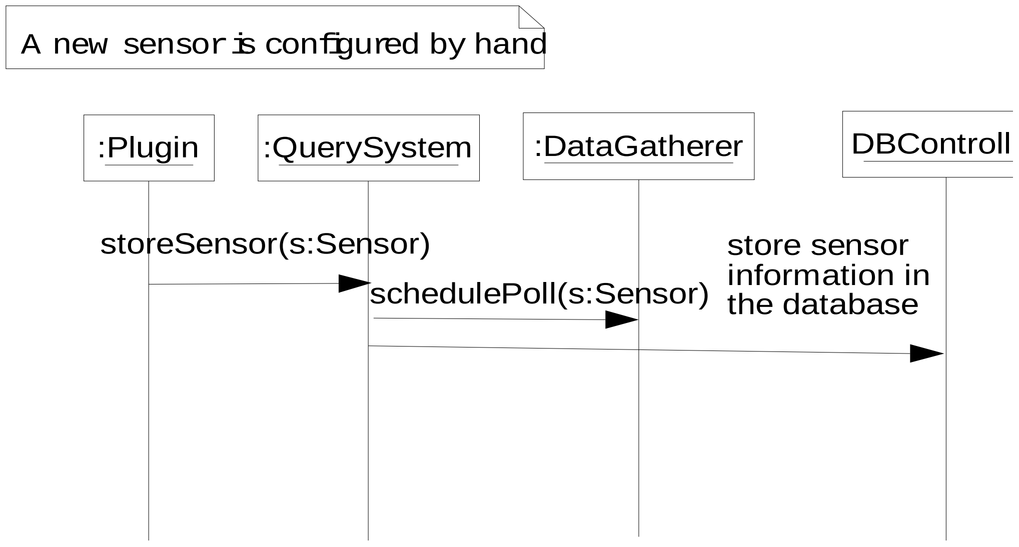


Figure 4 - A new sensor is configured by hand

Sensor reliability example

This usage scenario starts with common queries to the central system, as seen on Figure 2. Sunlight sensors are queried to detect certain trends in their readings that show possibility of premature failure. All the system has to do is provide data regaring sensors and their readings to a plug-in that will embed specialized knowledge that will be used to perform the required analisys.

After the analysis is performed, some sensors are verified in their location, and it is found that the is a small pine tree growing in front of a sunlight sensor, that had to repositioned and have its position updated in the database. This can be seen in Figure 5.

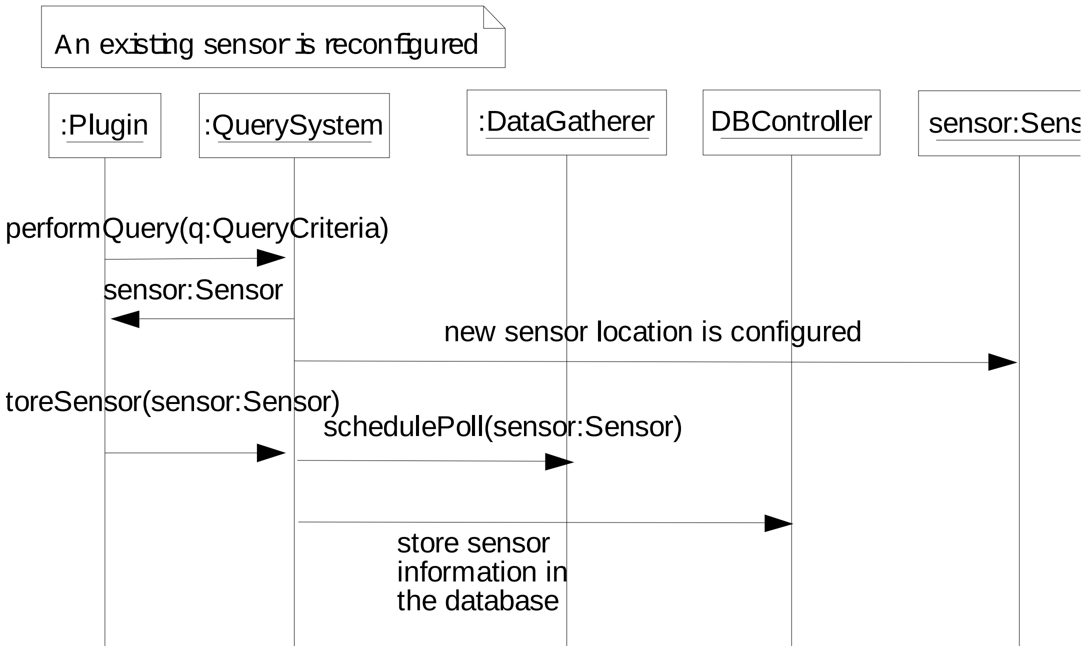

An existing sensor is reconfigured
:Plugin
:QuerySystem
:DataGatherer
DBController
sensor:Sens
performQuery(q:QueryCriteria)
sensor:Sensor
new sensor location is configured
toreSensor(sensor:Sensor)
schedulePoll(sensor:Sensor)
store sensor information in the database

Figure 5 - A sensor configuration is updated

In this scenario, a plugin asks a sensor object to the central system, edits its configuration and sends it back to the central system.

Sensors that have to be polled

This interaction was not a explicit part of any of the scenarios, but is necessary to be understood since only one type of sensor is capable of actively sending its data to the central site. This interaction is very similar to that one shown in Figure 3, for sensors that actively send their data.

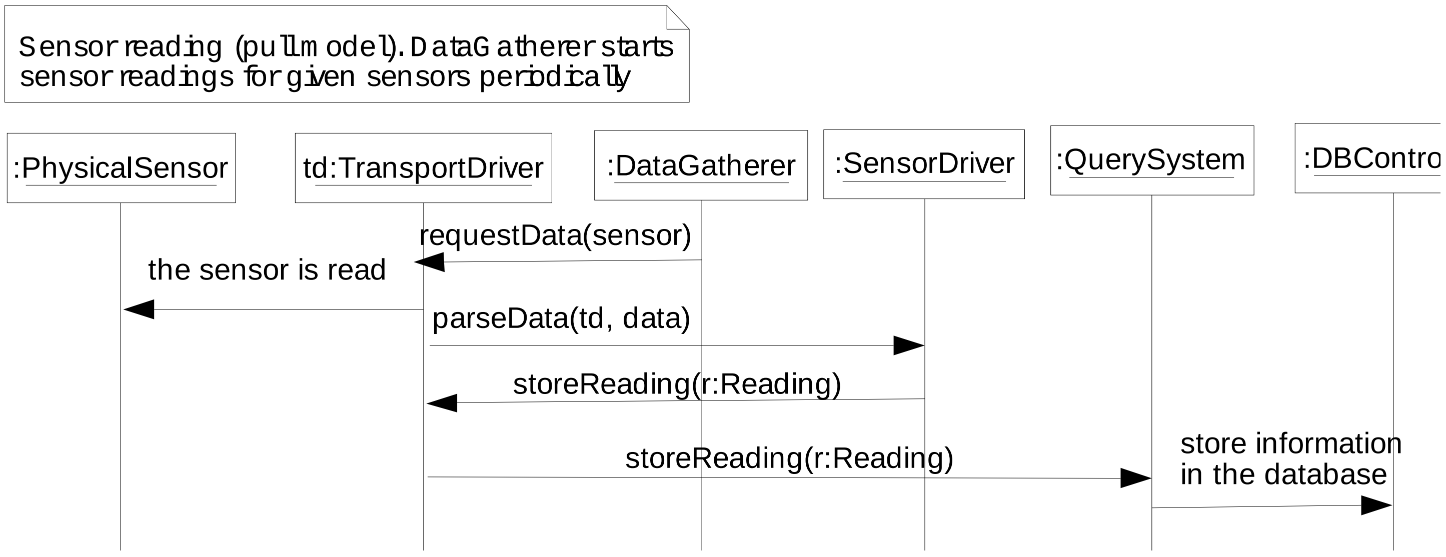


Figure 6 - Reading data from sensors that have to be polled

The differences between the two types of sensors (pull or push) are hidden behind the specific *TransportDrivers*

System Interfaces

The ArborXT system has two points of connection to external systems:

- The *TransportDriver* layer – the set of *TransportDrivers* connect the system to the physical sensors in the field through a variety of types of communication links.
- Plug-ins – Software will that interacts with the system play the role of plug-ins, and through the *setEnvironment* method gain access to the *QuerySystem*, which in turn gives them access to *SensorData*, *Sensor* and *Reading* objects.

Lessons learned

At the end of the process, we summarized in group the most important lessons learned during the *design fest*.

•Process planning – it is very important to plan activities during the design fest to prevent runaway topics of discussion from wasting all available time and have an overall feeling that progress is being made. During day 2 we forgot to plan at times and ended up spending more time than needed on not so relevant subjects.

•Read and discuss the case thoroughly – one of the features that we spent a considerable time designing wasn't explicitly (or maybe even implicitly) listed as a requirement. That time could be better used improving the overall design of the system.

•Keep it simple - at several moments complicated decisions made before hampered the design process to the point that we ended up coming back and canceling such decisions. It helps to focus on simplicity the first time to avoid rework.

•Patterns and communication – the use of patterns greatly improve the speed of design by shifting the degree of abstraction of discussion to a higher level. At several times members proposed patterns-inspired subsystems and everyone understood what that meant and went on to discuss the advantages and disadvantages of that.

•Patterns and design – by adopting solutions based on design patterns, we benefit from tested solutions with known advantages and disadvantages, making decisions more consciously.

•Naming of classes – The naming of classes is very important, sometimes members of the group that had absolutely the same point of view engaged on arguments because the name of a class was doing a poor job at communicating what that class did, giving the impression that it didn't conform to common design decisions.

To sum it up, participating in a *design fest* is very fun, everyone in the group got very involved in the discussions and ended up feeling responsible for the system as a whole.