Unified Management of Heterogeneous Sensor Networks In the Atlantis Framework

Vesselin Arnaudov
Department of Computer Science
Brown University
varnaudo@cs.brown.edu

Abstract

In recent years, the emergence of low-cost, easily maintainable embedded devices, as well as an overall increase of the proliferation and applications of the concept of ubiquitous computing, have contributed to the rise of many new sensor and actuator platforms of different design and intended purpose.
The Atlantis Framework aims to answer the need of a unified way to manage one or more arrays of sensors and actuators. It defines a model, captured in a generic schema, that governs how networks of devices are represented at various levels of granularity, their data collection, aggregation, and processing paths, and provides a simple mechanism to control this configuration. This paper introduces the problem addressed and shortcomings of current alternative solutions, and then proceeds to present the semantics and sample syntax of the Atlantis schema, an implementation of the framework along with a sample application, and concludes with suggestions for future development.

1. Introduction

The motivation behind the Atlantis Framework is the current state-of-the-art in bridging and interfacing with different sensor networks. It was conceived as a requirement-driven project, seeking to find an acceptable solution to a wide range of chores that are often introduced when creating an application that interacts with sensor data input. Consider the following scenario: an environmental scientist has the need to create an application that will visualize the energy flow in an energy-efficient building. She has access to the interface of the building's HVAC system and its related sensors (carbon dioxide, ventilation output, etc.), several wired data loggers which support attachment of devices for taking precise measurements on water flow, electrical consumption, etc., as well as a number of wireless motes running TinyOS that can be scattered around the building and provide data readings of outside temperature and light levels. The scientist is interested in being able to easily define the network, possibly creating relationship between devices (e.g., whereas the individual readings of light and temperature from each node in the mote network are not important, their local combined result would be important to visualize), and have the data pushed to a client for visualization in real-time, as well as available to be queried individually for calibration or research purposes.

Another scenario would be a university department that has a Bluetooth network deployed at its site, as well as a network of wireless cameras with a web interface. Due to instances of missing equipment, the department would like to leverage the existing infrastructure to be able to track the location of Bluetooth-enabled devices (such as a laptops), and have the cameras automatically capture and send a picture from the relevant cameras if the laptop enters a predefined area (for example, an elevator area), so as to serve as a potential theft deterrent.

Traditionally, each sensor network in both scenarios would need to be configured manually and the logic for interfacing and collecting data from each device would be embedded in the client. This, however, may not be desirable since developing such an application might be technically challenging for an environmental scientist. Furthermore, should one of the sensor networks be replaced with another, significant changes would need to be introduced in the application in order for it to interface properly with the new data provider.
It would be better for the ease of development of such applications if abstractions that define the interface and relationships between sensor devices are introduced, as well as clearer means of retrieving or processing streaming data. Although models such as the Sensor Markup Language (*SensorML*) [2] or the Couagar Framework [5] have attempted to provide

certain solutions to this research question, they do not rigorously define data processing chores, are not as flexible towards working with heterogeneous devices, and are not client oriented, making them sometimes as difficult to implement in a real scenario as would be to write a custom application that does the data handling. The TinyML schema [1], on the other hand, provided a lot of inspiration and initial start of the project. Concepts such as sensor field, platform, and virtual sensor have been adopted in this model, albeit changed. To illustrate the relationship, *AtlantisML* is the 5th transformation of the TinyML specification, but because of increasing divergence in the models due to the fact that TinyML is more concerned with instantaneous readings and does not provide a reasonable implementation unlike the Atlantis model witch is requirement-driven and adds data handling abstractions, it is reasonable to segregate them at this point and name the resulting markup specification after the *Atlantis* model.

The Atlantis model, presented in section 2 of this paper, first addresses the problem of communicating with several sensor networks based on different physical data access points. More specifically, it defines the notion of a *sensor field* as a number of sensors attached to one or more *platforms* of homogenous interface (these terms are explained in more detail in subsequent sections). Such platforms can contain both *basic* sensors (physically present sensors such as a voltmeter or a digital counter), as well as *virtual* sensors that are aggregates of one or more basic sensors connected to the same platform. In the same model, it defines a *field* sensor as a virtual sensor spanning different platforms in the sensor field. For purposes of clarity and implementation, let there be another synonym for the platform abstraction in a field – a platform being the physical device that is compatible with a field, becomes a *node* in the sensor field when it is included in the field configuration and provided with a unique identifier.

The preliminary analysis on the implementation of the model by the Atlantis Server, presented in section 3, shows that it is satisfactory in the sense that large arrays of sensor fields can be easily encoded in a human-readable form (xml document), can be stacked to form a *super sensor field,* which, when added to a service agent, I would define to constitute a *sensor network*. Whereas the original TinyML model on which *Atlantis* is based provides for the common protocol that would allow the different sensor fields to be queried by a single client, the service agent (Atlantis implementation) has the job of actually providing the interface to communicate to the whole super sensor field, manage its configuration, and status. In addition, whereas the original specification seems to be intended for synchronous polling of sensor data, it can be expanded and interpreted to be able to mandate data storage, processing, notification of subscribers or other type of handling.

2. Atlantis Architecture

This section provides an overview of the Atlantis framework architecture. The Atlantis framework provides flexible support for defining and configuring the composition and behavior of the principal elements of a sensor network. Throughout this paper, the term *sensor network* will be used to define a set of collaborating devices that can support inquiries for sensory data or requests for command actuation. Due to the distributed nature of such devices, the framework aims to define and enforce rules that govern a sensor network, so as to achieve cohesion among heterogeneous sensor devices and provide transparent access to the network for end users.

The main design goals and overall structure of the core of the framework - the Atlantis schema – are presented in section 2.1. Sections 2.2, 2.3, and 2.4 define the semantics of the principal elements of the model. Section 2.5 and 2.6 add principals to generalize data collection via data brokers and behaviors as well as a subscription mechanism to facilitate asynchronous information retrieval.

2.1. Unified Protocol (Schema)

The Atlantis model of interaction with sensor networks is captured by an XML schema. This decision was made so that proper syntax of the protocol of communication could readily be enforced in an implementation. In addition, one goal of the framework as determined by the scenarios described in the introduction is to provide an intuitive way to control the sensor environment. Having such a schema suggests that elements of the model will be expressed using Extended Markup Language (although, of course, this is an implementation decision; XSLT transformation tools could be used to bind the schema to other languages). XML is preferred, however, because it is mostly human readable, tools in various languages exist for its parsing and construction and hence – clients to the framework could be implemented on different platforms relatively intuitively. The *AtlantisML* schema defines all

principal elements that take part in the interaction and data processing with the sensor network. A peculiar feature of the schema is that a lot of the elements are non-mandatory, i.e. they may not be present in XML document that is based on it. This provides the necessary flexibility to use *AtlantisML* as a protocol, inherently compliant with the model, to query and configure the sensor network it defines. Furthermore, *AtlantisML* compatible devices would then be able to provide services typical of distributed systems such as transparency, load distribution, fail-over, and scalability.
For the full definition of the XSD schema, please refer to the current project website (subject to change) at http://www.cs.brown.edu/~varnaudo/projects/Atlantis.

2.2. Sensor Field and Platforms

The top-level element in the schema is the *Super Sensor Field*, which is a collection of *Sensor Fields.* The sensor field*,* is the main building block in the model that captures the notion of a sensor network.

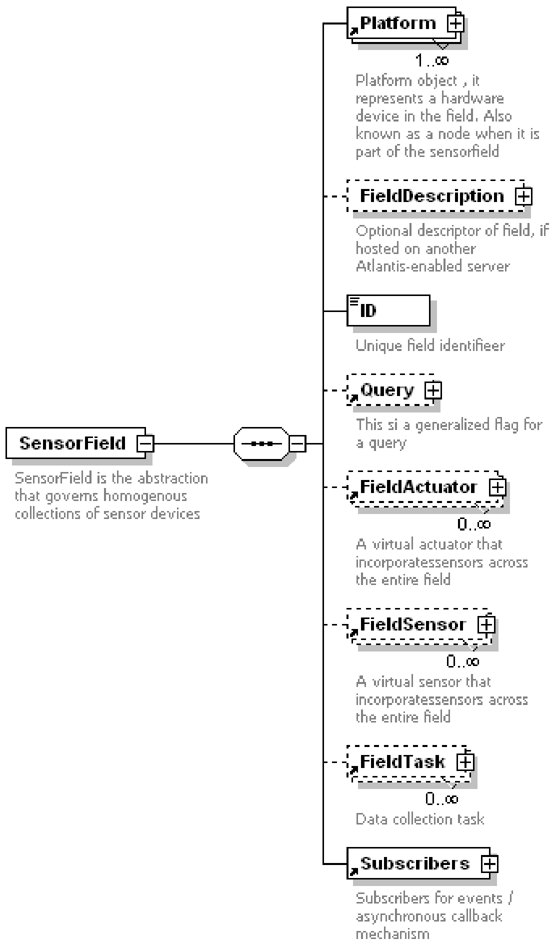
Platform
1..∞
Platform object , it represents a hardware device in the field. Also known as a node when it is part of the sensorfield
FieldDescription
Optional descriptor of field, if hosted on another Atlantis-enabled server
ID
Unique field identifieer
Query
This si a generalized flag for a query
SensorField
SensorField is the abstraction that governs homogenous collections of sensor devices
FieldActuator
0..∞
A virtual actuator that incorporatessensors across the entire field
FieldSensor
0..∞
A virtual sensor that incorporatessensors across the entire field
FieldTask
0..∞
Data collection task
Subscribers
Subscribers for events / asynchronous callback mechanism

Figure 1: Structure of Sensor Field Abstraction
Optional elements are marked by a dashed line around them and elements that are allowed repetition have the minimum and maximum occurrences as subscripts

It is an abstraction that groups together sensor devices with a common access/control interface (or a *gateway* in our model, discussed in section 2.4). Devices that have this property are said to form a homogeneous network. For example, the web-based cameras discussed in one of our scenarios will be one such group, since all of them are accessible by queries using the *http* protocol.

TinyOS motes, on the other hand, are possible reachable by TinyDB queries (assuming TinyDB is also installed on each one). Consequently, each will define their own sensor field. A sensor field must possess a unique identifier (*ID*) under the local namespace of the governing document (each *AtlantisML* schema instance is assumed to have a local and a target namespace corresponding to its site location, similar to other XML schema constructions).
The other optional elements of the sensor field (see Figure 1) are discussed in the sub-sections that follow.

Platforms

A *platform* in the model is an abstraction for a physical hardware device that provides an interface for interaction and supports one or more *sensors* and/or *actuators* attached to it (discussed in next section). Examples include the *Mica* mote, a wireless network camera, or even a workstation running Linux with a Bluetooth radio. This paper (and the model) will use also the term *node* to refer to a platform when it is a part of a sensor field. Borrowed from GML and modified to fit the needs for defining the sensor network are the optional *location* and *orientation* attributes of the platform. The *platform characteristics* include a unique identifier of the model of the platform (*ModelID*) which can be used by an implementation to organize available platforms in a catalogue and correlate them to available gateways, as well as a unique identifier (*ID*), that is assigned to a *node*.

The argument why the model organizes platforms in sensor fields (and supports communication to heterogeneous devices via the *super sensor field* abstraction), as opposed to merely having a collection of platforms is because, in real applications, homogeneous platforms are usually deployed with a common purpose and often define a site (both logically and geographically). In addition, when security structures and meta policies are added to the model (as described in section 7), better granularity over access control of particular sites, as well as the managing gateways can be achieved if such grouping already exists.

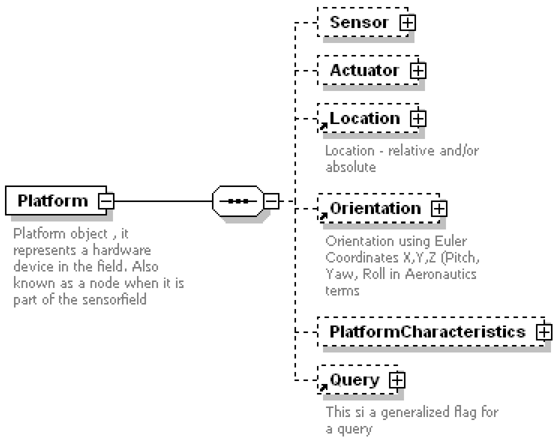
Sensor
Actuator
Location
Location - relative and/or absolute
Platform
Platform object , it represents a hardware device in the field. Also known as a node when it is part of the sensorfield
Orientation
Orientation using Euler Coordinates X,Y,Z (Pitch, Yaw, Roll in Aeronautics terms
PlatformCharacteristics
Query
This si a generalized flag for a query

Figure 2: Defining Platforms in Atlantis

2.3. Sensor/Actuator Models

Sensors, in the Atlantis framework, are split into two categories. The first one, *basic sensor*, models a physical device such as a voltmeter, or RSSI readings from a Bluetooth radio. Orientation attributes are again borrowed from GML models that suggest scientist sometimes require additional information about the sensor positioning when calibrating or analyzing its data. Additional characteristics of a basic sensor include model identifier so the sensor can be addressed, and optional vendor details and accuracy settings.

The other fundamental sensor type is a *virtual sensor,* which models an aggregate sensor on a node. Such sensors resemble the basic sensor abstraction, but have additional elements governing the basic sensors it spans and the aggregate function that is to be applied to the data.

The mechanism with which virtual sensors are handled is left to the particular implementation of the model, however, it is envisioned that a good data server will provide mechanisms for *virtualizing* a sensor (creating a new aggregate sensor) either natively on the target platform, or as a *pseudo-virtual* sensor, in which case the implementation takes care of performing the data collection and aggregation operations on the local site, so that it remains transparent to a client.
Actuators, are modeled in the exact same manner as sensors, with the addition of a *SetFlag* element with necessary attributes to hold parameters to be executed as commands.

Field sensor or *actuator* structures model virtual sensors/actuators that span sensors on multiple nodes (platforms) in a sensor field.

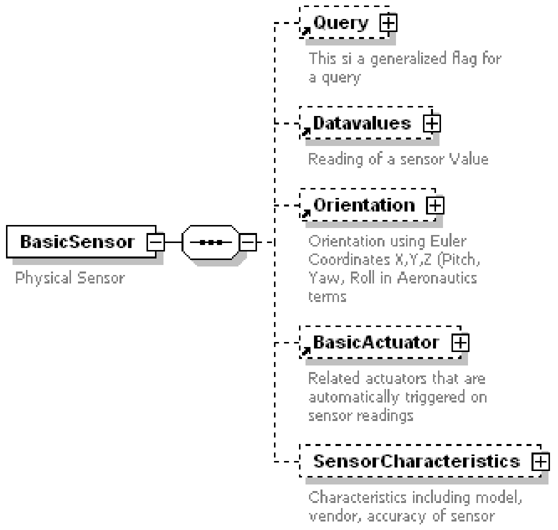
Query
This si a generalized flag for a query
Datavalues
Reading of a sensor Value
Orientation
Orientation using Euler Coordinates X,Y,Z (Pitch, Yaw, Roll in Aeronautics terms
BasicSensor
Physical Sensor
BasicActuator
Related actuators that are automatically triggered on sensor readings
SensorCharacteristics
Characteristics including model, vendor, accuracy of sensor

Figure 3: Definition of Basic Sensor

2.4. Gateways

Before the data collection and aggregation implementation can be discussed, it is important to note how *Atlantis* models access to the different sensor fields and defines the bridge to the actual hardware devices. Each sensor field has a related gateway definition element stored in a catalogue which contains specific details about the library, package, and entry point of each gateway that is to be used. Gateways are modeled as separate library packages to allow for flexibility in implementation such as dynamic installation (it is assumed that in most scenarios, there will be potentially more than one client using an *Atlantis* manager so persistence has to be accounted for in the model), as well as proprietary gateway libraries which do not fall under open source agreements. An example gateway would be the client part of TinyDB used to inject queries and retrieve data from TinyOS-mounted sensors.

Furthermore, the gateways (and corresponding sensor fields) are classified as one of the three main types: active (where the field actively pushes its data back to the gateway server as listener, such as in the case of TinyDB), passive (where the sensor field responds to a polling request by the gateway; an example in such a case would be probably building control systems with limited capabilities), and hybrid (where the sensor field will actively *save* data onto local data loggers and then respond to polling requests by accessing the stored data and providing it to the gateway). The most notable difference between the three different types is that, in the case of a power failure or crash at the gateway server, a hybrid sensor field will not suffer any loss of data point readings.

2.5. Data Collection

The Atlantis model presents two types of querying the underlying sensor networks for data readings, as required again by applications.

Synchronous querying (client sends request document to server, server identifies requested data, gathers instantaneous reading and returns from the call) is accomplished by placing a *query* element at some level of the *AtlantisML* compliant document indicating that we are interested in the values of the siblings of the current schema element. For example, a *query* element present in a *sensor field* would mandate the return of the sensor field ID, as well as the *platforms* in that sensor field, and other related structures (such as subscribers, etc.). Then, a *query* placed within a *platforms* element should reveal further details about it (sensors and orientation belonging to it, location, orientation, etc.) and so on depending on the desired level of detail.

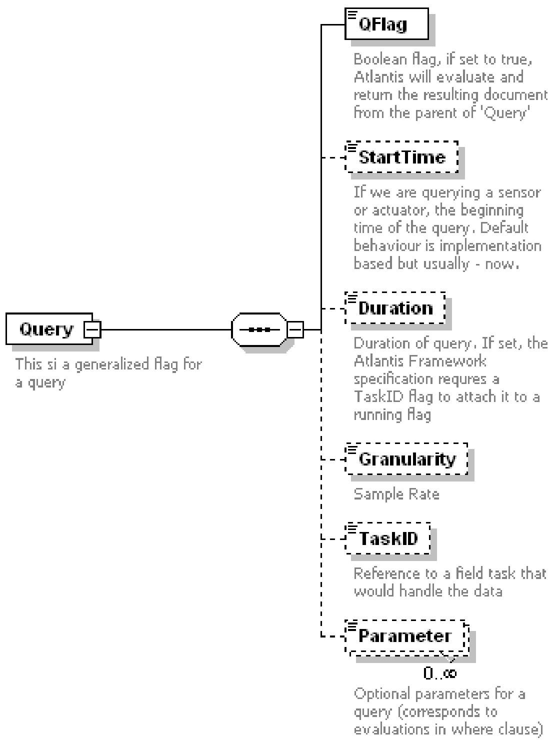
QFlag
Boolean flag, if set to true, Atlantis will evaluate and return the resulting document from the parent of 'Query'
StartTime
If we are querying a sensor or actuator, the beginning time of the query. Default behaviour is implementation based but usually - now.
Query
This si a generalized flag for a query
Duration
Duration of query. If set, the Atlantis Framework specification requres a TaskID flag to attach it to a running flag
Granularity
Sample Rate
TaskID
Reference to a field task that would handle the data
Parameter
0..∞
Optional parameters for a query (corresponds to evaluations in where clause)

Figure 4: Query Element (flag)

When a *query* element is placed under a sensor, the *datavalues* element will be populated with the instantaneous readings of the sensor. Using this model of inquiry allows for an intuitive construction of the query by examining the human-readable document produced by the server and placing the flag at the appropriate places (which could even be done by manually constructing the document, so that the client could be as simple as a text editor).

The nature of sensor data acquisition, however, requires the *Atlantis* model to support asynchronous means of data retrieval, where a query is submitted to the framework with requirements set for the start time, sample rate (granularity), and duration. Then, the *query* element will need to set an appropriate identifier for the *field task* (see section below) that will take control over the acquisition process.

2.5.1. Field Tasks

A *field task* is a structure in the model that governs how asynchronous data retrieval and processing is done. It has a unique identifier, set by the requesting client (or automatically depending on the implementation of the system), a main data processing agent (called a *data broker* in this model), field for whether the task is to remain persistent (in case of needs to pause it and restart or in the situation of a server shutdown), a priority element that would dictate how tasks should be scheduled, and a collection of *broker behaviors* (explained in section 2.5.3) that allow the modeling of a more advanced data flow path.

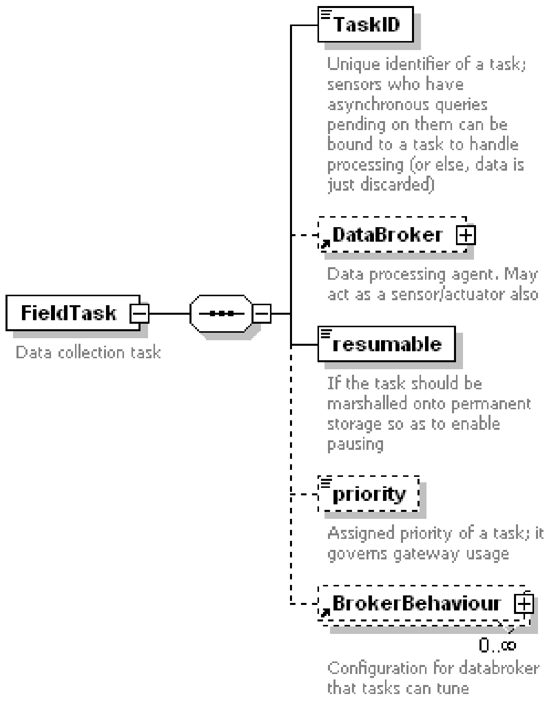
TaskID
Unique identifier of a task; sensors who have asynchronous queries pending on them can be bound to a task to handle processing (or else, data is just discarded)
DataBroker
Data processing agent. May act as a sensor/actuator also
FieldTask
Data collection task
resumable
If the task should be marshalled onto permanent storage so as to enable pausing
priority
Assigned priority of a task; it governs gateway usage
BrokerBehaviour
0..∞
Configuration for databroker that tasks can tune

Figure 5: Field Task Structure

In a simple scenario, where the client is only interested in data collected and occasionally spooled for its retrieval), the main data broker would be defined to be an event spooler that will take advantage of the *subscribers* element in a sensor field and log appropriate data values there.

2.5.2. Data Brokers

Data brokers define processing agents in the framework. Similar to a *gateway*, *Atlantis* models the data broker, so that it can be an atomic object in the implementation. The reason for this is again to ensure that new computational logic can be dynamically added to the sensor network server, and that they can be developed by third parties without caring about the specific implementation of the framework, as long as it conforms to the model. Furthermore, it would be desirable if data brokers can be potentially outsourced to other compliant implementation so that load balancing and fail-over capabilities can also be implemented. For the purpose, a unique identifier of the broker is required in its definition, as well as the corresponding package, endpoint, and library where it can be accessed. An additional *location* element is in place to facilitate exchange of brokers among different servers, and two Boolean flags that allow discovery of the broker as a *field sensor/actuator*.

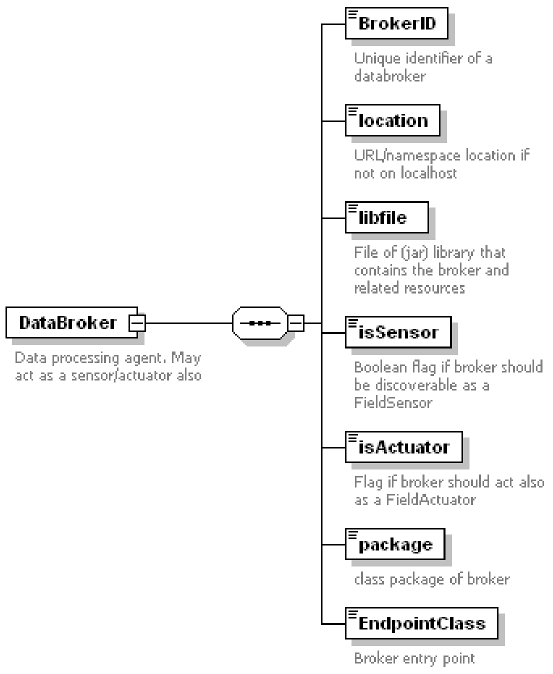
DataBroker
Data processing agent. May act as a sensor/actuator also
BrokerID
Unique identifier of a databroker
location
URL/namespace location if not on localhost
libfile
File of (jar) library that contains the broker and related resources
isSensor
Boolean flag if broker should be discoverable as a FieldSensor
isActuator
Flag if broker should act also as a FieldActuator
package
class package of broker
EndpointClass
Broker entry point

Figure 6: Data Broker Definition

Due to the fact that data brokers can be linked together via *behaviors*, an additional structure that specifies the syntax of the input and output of these agents is required. The *broker data exchange* object

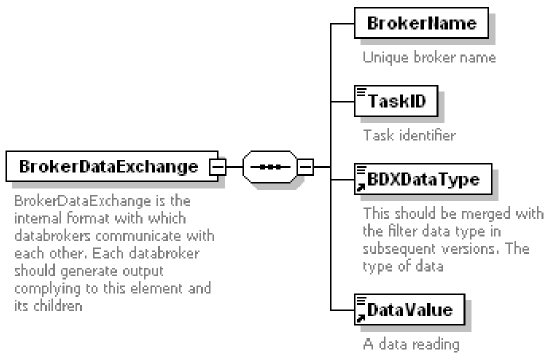
BrokerDataExchange
BrokerDataExchange is the internal format with which databrokers communicate with each other. Each databroker should generate output complying to this element and its children
BrokerName
Unique broker name
TaskID
Task identifier
BDXDataType
This should be merged with the filter data type in subsequent versions. The type of data
DataValue
A data reading

Figure 7: Broker input/output standard

encapsulates the broker identifier (*broker name*), the task that originated it, type definitions of the data their value. It is important to introduce some notion of types in the model, as data could all be potentially encoded as a string. Types will further be useful in section 2.5.4 where data filters are defined.

2.5.3. Broker Behaviors

A broker behavior structure is a part of a field task defined in a sensor field and provides mechanisms for processing of data in addition to that done by the original task data broker.

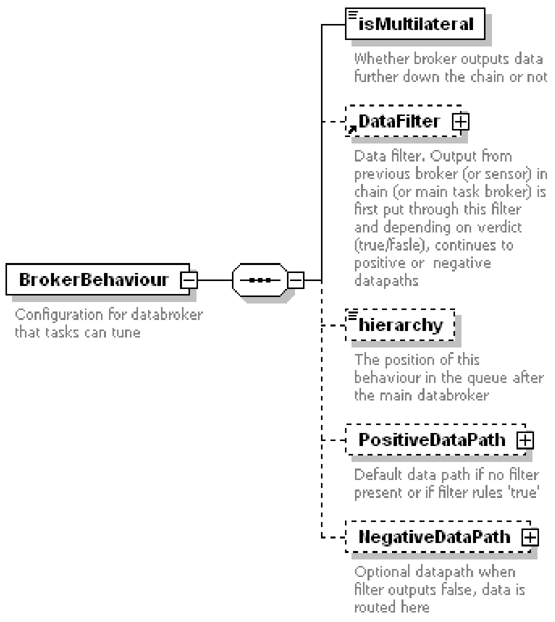
BrokerBehaviour
Configuration for databroker that tasks can tune
isMultilateral
Whether broker outputs data further down the chain or not
DataFilter
Data filter. Output from previous broker (or sensor) in chain (or main task broker) is first put through this filter and depending on verdict (true/fasle), continues to positive or negative datapaths
hierarchy
The position of this behaviour in the queue after the main databroker
PositiveDataPath
Default data path if no filter present or if filter rules 'true'
NegativeDataPath
Optional datapath when filter outputs false, data is routed here

Figure 8: Data Broker Behavior

The broker behavior first specifies whether there will be data output from this structure. It proceeds to establish which position of the chain of behaviors it occupies by the *hierarchy* element (it is implementation-specific how the hierarchy will be handled but the recommendation is to use 1 for the first behavior, and subsequent integers for the rest). A mandatory *positive data path* references a data broker structure which will be normally responsible for the processing of the incoming data broker exchange object. Optionally, a *data filter* could be present in this abstraction to provide a Boolean ruling of whether the data should be processed by the positive broker ('yes' ruling) or the negative data path broker. If the negative data path is not define, and the filter return false, then the data object is discarded and does not proceed further down the chain. Otherwise, the data is processed by one of the two brokers defined in the data paths and their output data (possibly different from the incoming data) is then propagated down the chain. Thus, the broker behavior object provides for an if/else conditional control of the data that is flexible enough to create data paths scenarios fulfilling the needs of most applications (examples of which will be discussed in the sample application section).

2.5.4. Data Filters

The *data filter* that was mentioned in the previous section is a simple evaluator that checks data encapsulated in incoming objects of the broker data exchange type against some criteria and renders a true/false ruling.

The data filter, due to its computational nature, is also meant to be an atomic structure that can be moved to a different *Atlantis* compliant server. Therefore, it defines the elements that have been so far required for portability (*libfile,*etc.), as well as a *check type* element governing the type of check to be performed of the *data values* vector (for example, some permissible values in the current implementation are *individual* where each element of the vector is checked against the *check expression*, *average* where the average value of the vector is checked, and others). The *check operator* defines the type of check to be performed (*greater-or-equal-to, not-equal-to,* etc.) and the *check expression* is what the data vector is compared against. Since the filter is an atomic object in the model, the precise syntax of the filter and especially the *check expression* can be different depending on its implementation. However, it is recommended that the authors of filters implement a wildcard that allows pattern matching among the other checks.

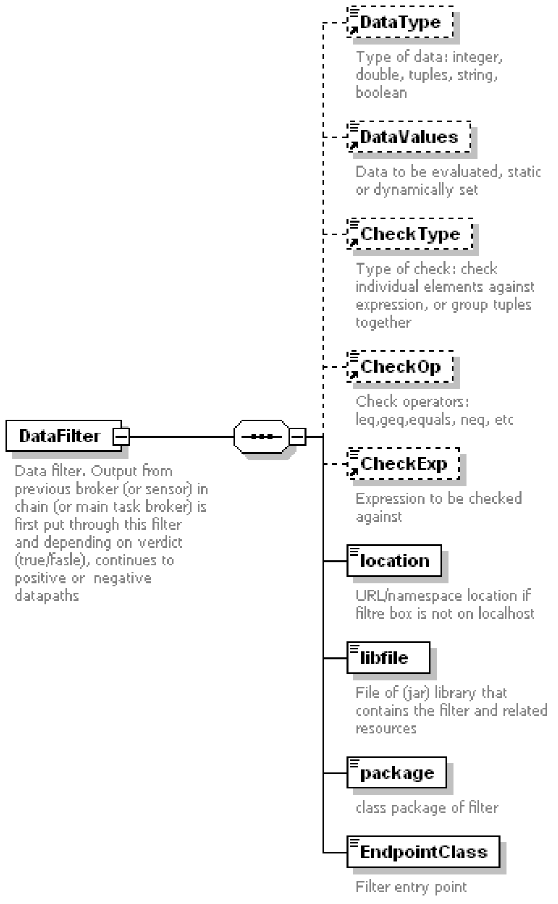
DataType
Type of data: integer, double, tuples, string, boolean
DataValues
Data to be evaluated, static or dynamically set
CheckType
Type of check: check individual elements against expression, or group tuples together
CheckOp
Check operators: leq,geq,equals, neq, etc
DataFilter
Data filter. Output from previous broker (or sensor) in chain (or main task broker) is first put through this filter and depending on verdict (true/fasle), continues to positive or negative datapaths
CheckExp
Expression to be checked against
location
URL/namespace location if filtre box is not on localhost
libfile
File of (jar) library that contains the filter and related resources
package
class package of filter
EndpointClass
Filter entry point

Figure 9: Data Filter

2.6. Events and Subscribers

The *subscriber* and *events* structures model the last

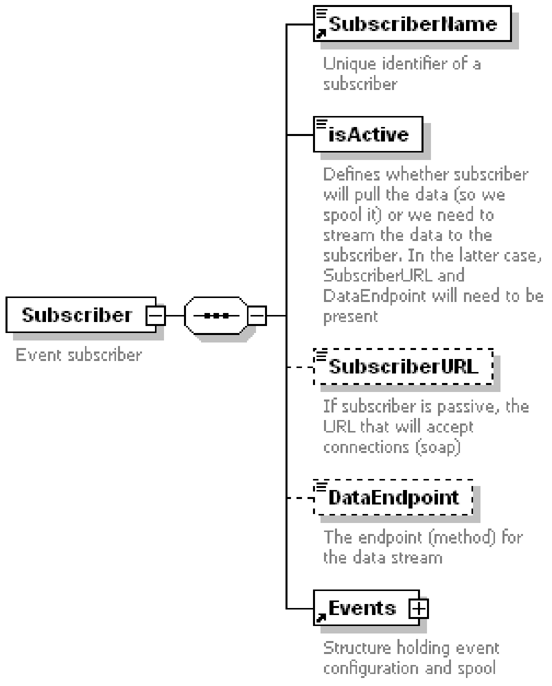
SubscriberName
Unique identifier of a subscriber
isActive
Defines whether subscriber will pull the data (so we spool it) or we need to stream the data to the subscriber. In the latter case, SubscriberURL and DataEndpoint will need to be present
Subscriber
Event subscriber
SubscriberURL
If subscriber is passive, the URL that will accept connections (soap)
DataEndpoint
The endpoint (method) for the data stream
Events
Structure holding event configuration and spool

Figure 10: Subscriber Description

portions of the data flow path in the *Atlantis* framework needed for asynchronous data processing and retrieval. A *subscriber* is an element added to the sensor field when a client desires to be notified (or to pull) data that has been prepared for it.

It features the unique name of the client (*subscriber name*), and whether it is going to be an active (pull) relationship or the client will declare a URL and endpoint (*is Active, subscriber URL,* and *data endpoint* respectively) so that the server can push data when it is ready directly to it.

In all cases, the client will specify one or more *event configurations* (belonging to *events*) that it is interested in receiving, each containing enough data about the originating task that triggered the event, the broker, and if applicable (when using the passive approach, the position of the spooling broker in the chain of behaviors). The *event spool* sequence (see figure on next page) of the event configuration then holds the actual data pertinent for the client that has not yet been picked up.

Section 4 of this paper, where a sample application based on an implementation of the platform is described, will provide a more detailed discussion on the data flow and how each element from the model is used.

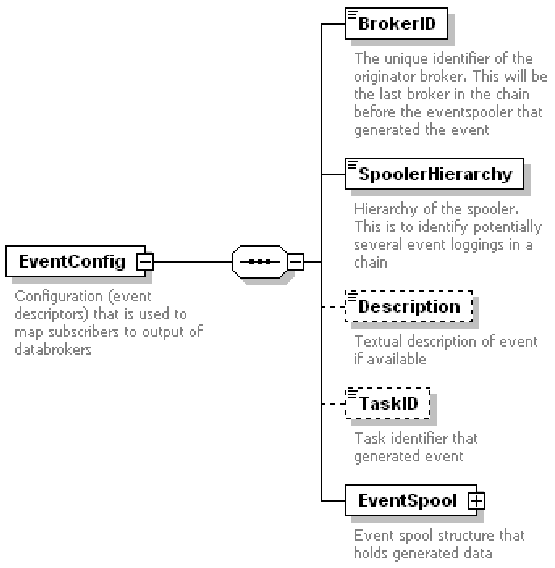
BrokerID
The unique identifier of the originator broker. This will be the last broker in the chain before the eventspooler that generated the event
SpoolerHierarchy
Hierarchy of the spooler. This is to identify potentially several event loggings in a chain
EventConfig
Configuration (event descriptors) that is used to map subscribers to output of databrokers
Description
Textual description of event if available
TaskID
Task identifier that generated event
EventSpool
Event spool structure that holds generated data

Figure 11: Event Configuration and Spool

3. Implementation

This paper offers a sample implementation that follows the model and guidelines of Section 2. The *Atlantis Server* is presently a Java-based daemon exposing a web service accessible through Simple Object Access Protocol, in order to fulfill the goal of being accessible from a broad range of clients.

The service then exposes a single entry-point method to the *AtlantisML* interpreter, as well as a number of methods that can be invoked through the service endpoint and can be used to administer the server: install new nodes in a field, remove sensors or nodes in a field, commandeer data acquisition and processing chores, report errors and notifications, and in general, maintain the proper consistency in the super sensor field configuration. Since it is a SOAP service, it is accessible through any client that is capable of receiving and transmitting properly encoded SOAP packages; the current proof-of-concept application discussed in the next section uses a Java client but previous versions of the server have had Macromedia Flash clients for a Pocket PC that demonstrate the ease of use and interoperability of the service.

Clients are envisioned to communicate with the server via the endpoint of the *AtlantisML* interpreter, by providing an *AtlantisML* conforming XML document. One further advantage of the single entry point decision is that, as a per-client service that deals only with AtlantisML compliant documents, it can in future use caching, implemented by a simple round-robin queue, to store hashes of AtlantisML requests and answer, if possible, by a cached AtlantisML response document. By lessening the burden in this manner when numerous clients are often inquiring for the sensor field configuration for example, one could expect less congestion within each sensor field and gateway, and enhance performance and power consumption especially when it is comprised of low-power, limited-resource embedded devices (such as the Mica family of motes).

As noted, the service contains a gateway catalogue file residing amongst its resources with specific details about the location and library of each gateway implementation that is being used in the current configuration of the super sensor field. The library consists of an archive *.jar* file, in addition to a pointer to an entry class that implements the *Gateways* interface: the server dynamically attempts to create a new instance of the gateway at initialization, and can have a gateway updated to a newer version without a recompile/reinstall of the server base. On the other hand, this also allows 3rd parties to develop their own, potentially proprietary gateways without any other knowledge or tweaking of the main service code.

Proof-of-concept gateways for interacting with a camera network, the TinyDB application running on top of TinyOS, and a Linux-based network of Bluetooth devices are provided. A gateway, in addition to providing basic functionality to assess the status of and poll the sensor field, has the extended ability to report whether the field supports native virtualization of sensors or not: as discussed in the AtlantisML model, a native virtualization would mean that the virtual sensor operation is defined in and processed at each *platform* in the field, and then data could be queried just like a *basic sensor* reading. If the sensor field does not support such a function, the server creates a *pseudo-virtual* sensor: one that is defined at the server level and not at the platform level. This, however, is transparent to any of the clients of the service and only matters how data collection is performed.

Data Collection and Aggregation

The basic data collection and aggregation performed by the *Atlantis* service is done in the following fashion. Whenever a '*query'* flag is detected in the incoming *AtlantisML* document by the interpreter, it exposes all other elements at the current level for that particular sensor field and returns the document back to the reader (for exapamle, a *query* flag set inside the `<SensorField>` parent would indicate that the client wants to learn about all nodes within the sensor field, with *at least*, their identifiers and characteristics, as well as the characteristics of the sensor field itself).
In this implementation, the data about the configuration is actually stored in a AtlantisML compliant document residing on the file system and is bound via a casting engine back to Java classes that enforce its marshalling to and from the disk.

When a *query* flag is detected inside a <BasicSensor> or <VirtualSensor> tag, the start time, end time, granularity (sample rate), and optional parameter of the query would also need to be indicated. The service then proceeds to evaluate the time span of the request. If it finds that the time span includes future periods of time, it returns an error response that the data is not yet available, and it requires a new *field task* (implemented in the *Task* object) to perform the data collection.

A new *task* or data collection chore is dynamically instantiated upon the particular type of the gateway that needs to communicate with the sensor field: if it is an active gateway, then the corresponding task is a *PassiveTask* object that registers itself as a listener to the gateway and takes in whatever results arrive from the gateway (of course, it initially injects the required query inside the target network). On the other hand, if it is a hybrid or passive gateway, the task that is run is an *ActiveTask* which, as a separate thread that is awake only at the specified sample rate period, actively polls the gateway for the required information and then propagates the result.

All tasks use a *MetaQuery* object that identifies the data that needs to be collected as well as how precisely it needs to be reconstructed: the structure is quite complicated since I allow virtual sensors to be aggregated as well (building a virtual sensor from one or more other virtual sensors). The MetaQuery constructs a balanced tree of the objects with the real physical (or natively virtual) sensors as the leaves. The gateway only uses the identifier of the physical (or natively virtual) sensors and when it returns data, the MetaQuery is responsible to propagate the collected results to each object further up in the tree, gradually fulfilling its requirements until all sensor aggregates have been filled. Using such a structure provides us also with the flexibility of optimizing the injections in the sensor fields (when multiple tasks are running at the same time, sensors do not need to be polled again if they are already asked for by more than one task that runs at the same sampling rate with the same parameters).
The data handling, upon a received *MetaQueryResult* object from the gateway is performed again in a customizable way by the *Task* object (implementation of the field task and related behaviors) and is best illustrated by the discussion of the sample application implemented in the *Atlantis* framework that follows in the coming section.

4. Sample Application

One of the easiest way to illustrate the versatility of the *Atlantis* model, implementation and data flow would be to trace the execution of a sample application that uses the framework. For this purpose, this section will briefly discuss the interaction between the Bluetooth (BT) Localization application [4] and the *Atlantis* server. The goal of the client application is to visually track the location of a BT device using several Linux machines equipped with BT radios and capture an image obtained from a wireless camera if the device is found in a user-defined range of coordinates. The setup is as follows.

There are two sensor fields that will need to be supported and configured in the *Atlantis* server: one that defines the BT network, with platforms being the BT-enabled workstations. *Basic* sensors supported on each base station are readings of the remote signal strength indicator (RSSI), power transmit level, and link quality. Actuators are the ability of a base station to connect and disconnect to a remote device.

The camera network is defined by a sensor field in which platforms are the actual web-based cameras and the sensors – the image sensor of the camera.

The process begins by an *AtlantisML* request by the client to return all platforms, their respective platform characteristics, and location elements from the BT sensor field so that they can be displayed in the GUI.
It then proceeds to ask the server to add a new resumable field task to the field with the proper identifier and priority and asks that the custom *BTLocalizer* data broker (which produces a tuple of *x,y* coordinate values of the device or a special string if not enough data could be gathered to find them) be assigned as the primary processing agent.

Afterwards, the client requests that a new broker behavior be added to the field task at the first available hierarchy (in this case – 1) and defines a positive data path (a data broker) to be the *Event Spooler* (a special broker that interacts with the *subscribers* structure and saves or pushes data to requesting clients when it is received) and negative data path to be empty (data is discarded). A data filter is then associated with this behavior and defines that verdict should be true if the data vector coming from the *BTLocalizer* broker is not the string "INVALID" (known to the client from the documentation of the data broker to mean that not enough data was available to produce localization coordinates).

Next, the client requests another broker behavior to be attached to the field task, and specifies that the *CameraBroker* be the one in the mandatory positive data path. This processing agent grabs an image from the camera network defined by the second sensor field. A filter is attached to this behavior that has a tuple of coordinates in the *check expression* element (ex..: (<15,30>, <12,25>)), a check operator of "between", and a check and data type of "tuple". The result is that if the data type entering this behavior is of the type "tuple", each element of it ((*x,y*) coordinates) will be checked to be in between the values of the corresponding tuple in the check expression.

A third behavior is finally added that has the *Event Spooler* data broker again set as the positive data path and no additional filter is configured (i.e. the data will always flow to the spooler).

The client asks the server to enter it in the list of subscribers for the sensor field with a unique identifier as an active subscriber (it will poll for new events when it decides to) and sets up event configurations to listen for the specified task and the event spooler at hierarchy one and for events generated by a spooler at hierarchy three.

The last step that the client undertakes is to set the query flag of the appropriate sensors on the base stations of the BT field, the parameter of the query (the address of the BT device to be tracked), and the identifier of the field task that has been set up to take care of the data.

The server evaluates and complies with each of the requests in the previous steps and sets up the necessary structures in the current sensor field configuration. When it processes the last step, it creates a *MetaQuery* object as explained in the previous section when the implementation of the server was discussed. The *MetaQuery* is provided to the *Task Manager* which spawns passive or active tasks, depending on the gateway. In this particular applications, the gateway is passive so the server needs an *active* task that re-injects the query at the specified sample rate.

Once data is available from the sensors, the *MetaQueryResult* object (containing a vector of data from each platform) is modified to comply with the Broker Data Exchange (data type and values are set accordingly in a schema-compliant document) and the data is provided to the main broker (the *BTLocalizer*). This is done by the *Task* object that has access to the *field task* settings in the sensor network configuration and uses it to guide the data flow.

Next, the task analyzes whether there is a broker behavior chain that needs to do something with the data. The new Broker Data Exchange (BDX) object generated by the main broker is then handled by the broker behavior construct that is first in hierarchy. If there is a filter (as in this case, results of "INVALID" need to be discarded), the BDX object is forwarded for further processing by the broker identified in the positive data path of the behavior in case of a positive ruling from the filter or it is discarded since no negative data path has been defined at this level.

Therefore, a valid coordinate tuple will pass the filter and continue to the first Event spooler. It will look into the *subscribers* and take the appropriate action (spool as in this case, or push the data to the client). The event spooler echoes the incoming data as its output (performs no modifications to it).

The BDX object output by the event spooler now must enter the next behavior, in this case – the one defining whether or not we need to take a picture. A filter analyzes again the data and if the coordinates are within the specified region, the BDX data is sent to the positive broker, which is the *Camera Broker* (or discarded since we do not care to continue otherwise and have not defined a negative data path).

The Camera Broker takes the BDX object and using the coordinates determines the camera that needs to be used for capturing. It outputs a *new* BDX object that contains the image data.

The new BDX data travels to the next behavior in the chain. It contains no filter so it automatically goes to become the input of the broker defined in the *positive* data path. In this case, this is another event spooler that saves the image data to be used by the client.

5. Conclusions

It can be argued that the more abstractions and structures a model specifies for such a framework, the more bulky and complicated its implementation becomes. The *Atlantis* model does introduce a number of abstractions, however, its requirements to an implementing service are flexible and allow for optimization on many levels (for example, when load distribution, fail-over and security policies are introduced in future versions). The current model achieves it goals by allowing clients as simple as a text editor (with network capabilities) to manipulate, configure, and extract data from a variety of sensor fields. Development of plug-ins (gateways and data brokers) is made straightforward and does not require any other knowledge about the framework implementation except the necessary interfaces for data access and exchange. Moreover, the flexible structures governing data flow allow for the construction of many usable applications or tools to augment existing applications (for example, providing sensor data and management of client subscriptions to the *Borealis Streaming Database* [7]). All in all, the *Atlantis* framework should be a useful tool and model whenever rapid deployment of new sensor networks and research applications that bridge them is required.

6. Future Directions

Due to the sheer size of the project, a number of important further improvements have so far been left out at this stage. They include:

- Access control policy to govern relationships between principals (including clients) of the framework. Since the model is represented by an XML schema, this would be an easy but necessary addition to facilitate real deployment of the system and data integrity
- Peer-to-peer or directory-based discoverability of *Atlantis* servers and principals (sensor fields, gateways, brokers) and mechanisms for server-to-server communication that is transparent to the client.
- Load distribution and fail-over mechanisms of data brokers and filters
- Complex data aggregation functions (virtualization): currently, only Avg, Min, Max, Log functions are supported
- Pseudo-native calibration in the same manner as the pseudo-native virtualization
- Porting catalogue and configuration files under database management – using JDBC to automatically create database schemas in accordance with XML schema of sensor data being obtained or sensor field configuration
- Preventing the tampering with the integrity of the collected data through watermarking techniques

7. Acknowledgments

My gratitude goes to Professor Ugur Cetintemel and Professor John Jannotti for their support, guidance, and encouragement in realizing this project. Many thanks also to Professor Roberto Tamassia for the use of the Algorithmics laboratory and the patience to let me become distracted with this project for awhile. Credit is due to Jason Ye for providing and adapting his great Bluetooth Localizer to become a proof-of-concept application for this framework, and of course, the invaluable contribution of the Department TStaff for their support and patience in defeating the foils of deploying Bluetooth and sensor networks within the department.

8. Bibliography

[1] Ota, Nathan, Kramer, William, T.C. *TinyML: Meta-data for Wireless Networks*. <http://kingkong.*me.berkeley.edu/~nota/research/ TinyML/project-paper-1.pdf*>

[2] Sensor Model Language (SensorML) – <http://vast.uah.edu/SensorML>

[3] Cox, Simon, Daisey, Paul, Lake, Ron, Clemens, Portele. OpenGIS Geopgraphy Markup Language Implementation Specification (GML 3). Version 3.0 for the Open GIS Geography Markup Language (GML) Implementation Specification. <http://xml.coverpages.org/geographyML.html>

[4] Ye, Jason. <u>Atlantis: Location Based Services with Bluetooth</u>

[5] Philippe Bonnet, J. E. Gehrke, and Praveen Seshadri. Towards Sensor Database Systems. Proceedings of the Second International Conference on Mobile Data Management. Hong Kong, January 2001.

[6] XML Schema Working Group, W3C XML Schema <http://www.w3.org/XML/Schema>

[7] Daniel Abadi, Yanif Ahmad, Hari Balakrishnan, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, John Jannotti, Wolfgang Lindner, Samuel Madden, Alexander Rasin, Michael Stonebraker, Nesime Tatbul, Ying Xing, Stan Zdonik: The Design of the Borealis Stream Processing Engine, Technical Report (Brandeis University/Brown University/MIT), March, 2004.