Accessing and Processing Sensing Data

Gilberto Zonta Pastorello Jr and Claudia Bauzer Medeiros
IC–UNICAMP — Av Albert Einstein, 1251 – Campinas-SP – Brasil
{gilberto,cmbm}@ic.unicamp.br

André Santanchè
DCEC–UNIFACS — Av Cardeal da Silva, 747 – Salvador-BA – Brazil
santanche@unifacs.br

Abstract

Scientific models are increasingly dependent on processing large volumes of streamed sensing data from a wide range of sensors, from ground based to satellite embarked infrareds. The proliferation, variety and ubiquity of those devices have added new dimensions to the problem of data handling in computational models. This raises several issues, one of which – providing means to access and process these data – is tackled by this paper. Our solution involves the design and implementation of a framework for sensor data management, which relies on a specific component technology – Digital Content Component (DCC). DCCs homogeneously encapsulate individual sensors, sensor networks and sensor data archival files. They also implement facilities for controlling data production, integration and publication. As a result, developers need not concern themselves with sensor particularities, dealing instead with uniform interfaces to access data, regardless of the nature of the data providers.

1 Introduction

Advances in sensor networks have leveraged research in computational models, which can now rely on more kinds of data on real world phenomena. This, however, brings new challenges to computational science. From the data perspective, challenges include dealing with integration of heterogeneous sources, sampling rates, data redundancy, sensor data querying, fusion and summarization, processing of stream real-time data, all subject to node, sensor and communication failures. From the network point-of-view, challenges include power management, communication protocols, physical device management, or dynamic reconfiguration of nodes. This paper is concerned with the data perspective - i.e., how to offer applications that implement the models transparently access sensor data, regardless of sensor physical characteristics, specific middleware programming environments, and data publication formats.

Middleware-like solutions, e.g. [8, 11, 17], are frequently proposed to enable applications to access the sensing data sources – usually through the offer of APIs. If, however, the models require new data processing functions, it is necessary to adapt or extend the middleware. Moreover, if extra data sources are required by the models, the applications must cope with the problem of complying with additional middleware or data formats.

To solve these problems, our approach supports uniform encapsulation of sensors and sensing data. It is based on a special kind of component paradigm – *Digital Content Components* (DCC) [15] – which provides mechanisms for uniformly encapsulating both data and/or data processing units, and for composing them into more complex elements, thereby helping solve interoperability issues. Applications that encode the models thus have homogeneous access to heterogeneous sensing devices, eliminating the need for application developers to concern themselves with whether data comes from static files or dynamic sources, as well as device-level implementation issues.

To achieve these goals, we had to create a new family of DCCs, geared towards stream and sensing data sources. As will be seen, this involved dealing with several challenges, such as how to encapsulate data sources with dynamic and/or context-sensitive behavior. Yet another obstacle concerned the need for implementing device-dependent drivers, to offer application developers uniform access to sensor generated data.

We illustrate our solution with a real case study of modeling environmental conditions for agricultural (i) planning and (ii) monitoring [3]. Planning (i) involves developing sophisticated models which run on heterogeneous files containing sensor-produced data to simulate crop growth in a given region. The main data sources are satellite-based sensors and spatially distributed networks of ground-based sensors. Once the crop is planted, monitoring (ii) concerns re-running and calibrating the models, for continuous use of the same kinds of sensing data sources, now at real time, at different sampling time frames, to capture and control crop response to changes. While data sets in phase (i) are static, phase (ii) adds the issues of real-time data capture and management. Our framework, as will be seen, provides a uniform solution to both phases.

The paper is organized as follows. Section 2 presents basic concepts and an overview of our proposal. Section 3 shows our solution to encapsulate data sources, software and sensor data within DCCs. Section 4 concerns preliminary implementation results. Section 5 discusses related work. Section 6 presents conclusions and ongoing work.

2 Solution Basics

2.1 Digital Content Components

A *Digital Content Component* (DCC) is a unit of content and/or process reuse, which can be employed to design complex digital artifacts [15]. From a high level point of view, a DCC can be seen as digital content (data or software) encapsulated into a semantic description structure. As shown in the example in Figure 1, it is comprised of four sections:
(i) the content itself (data or code, or another DCC), in its original format. In the example, a communication driver for a MICAz[1] sensor;
(ii) the declaration, in XML, of an organization structure that defines how DCC internal elements relate to each other (here, delimitating driver software);
(iii) specification of an interface, using adapted versions of WSDL and OWL-S – in the example, the `getTemp` and `subscribeGetTemp` operations;
(iv) metadata to describe functionality, applicability, etc., using OWL (in the example, the DCC is declared as belonging to the `temperatureSensorDCC` class).
Interface and metadata are linked to ontology terms – e.g., the `getTemp` operation has an input parameter that is a timestamp, as defined by the "Time" concept of NASA's SWEET [13] ontology.

There are two kinds of DCC – process and passive. A *ProcessDCC* encapsulates any kind of process description that can be executed by a computer (e.g., software, sequences of instructions or plans). Their interfaces declare operations they can execute. Non-process DCCs, named *PassiveDCCs*, consist of any other kind of content (e.g., a text or video file). We refer the reader to [14, 15] for details on DCCs.

[1] `www.xbow.com/Products/productsdetails.aspx?sid=101`

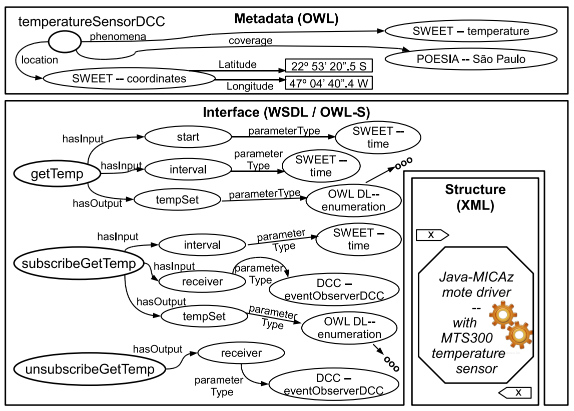

Figure 1. Structure of a SingleSensorDCC

2.2 Overview of Our Solution

The usual approach to access sensor generated data is either to communicate with the sensor directly in its specific protocols or to use a wrapper implemented for each type of sensor. We propose instead to encapsulate all the data production particularities behind new kinds of DCCs. Besides providing data access, such DCCs can aggregate several functionalities – e.g., data delivery rate, stream data control, data annotation.

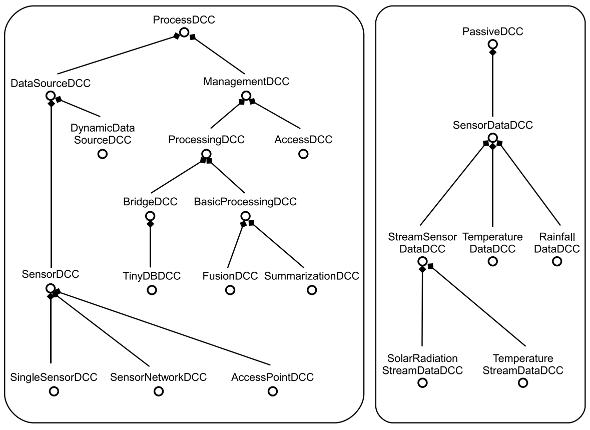

Figure 2. DCC taxonomy for sensor sources

Figure 2, gives a functional overview, summarizing our new DCC types. On the left, there are Process-

DCCs proposed to encapsulate data sources and data manipulation software; at the right side are the PassiveDCCs proposed to encapsulate data itself.

The diamond ended lines represent the subclass relationship, i.e., a SensorDataDCC *is a* PassiveDCC. From the process point of view, there are two main branches: *DataSourceDCC* and *ManagementDCC*. The former is used for data source encapsulation – see Section 3.1; and the latter is the basis for encapsulating data manipulation functions, discussed in Section 3.2. As will be seen, data sources can be dynamic or stable. A *SensorDataDCC* (Section 3.3) encapsulates sensor generated data. Data encapsulation consists of wrapping data with accessibility rules, descriptive metadata and structure. Data, devices and software are accessed through the same interface scheme, the major advantage of adopting a DCC framework.

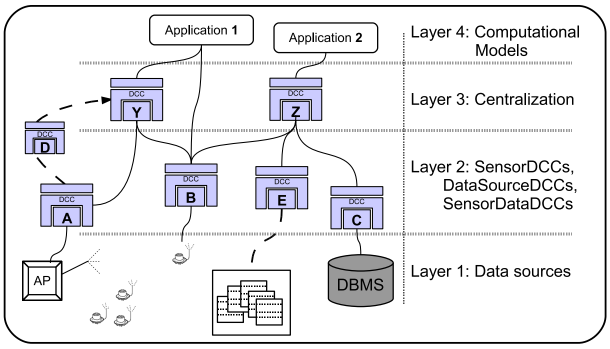

Figure 3. Management Layers

Figure 3 gives an architectural overview of our solution, using the DCC types from Figure 2. It shows the encapsulation of data in PassiveDCCs and of data sources and data management functions in ProcessDCCs. Continuous lines indicate data flow between the elements (e.g., from **B** to **Y**), whereas dotted lines indicate reference to data sources (e.g., from **A** to **D** to **Y**). The bottom Layer contains the data sources: sensors (and their auxiliary devices and communication features), DBMS and other kinds of data sources (e.g., repositories of text, satellite images, historical time series). The second Layer contains the DCCs that provide access to data (e.g., SensorDCCs **A** and **B**, DynamicDataSourceDCC **C**, and SensorDataDCCs **D** and **E**), which play a role comparable to that of a mediator to access the data. The third Layer has data organization and centralization features (pre-processing, summarization, fusion). Applications, which implement the computational models, are in Layer four and access raw data from Layer two or pre-processed data from Layer three.

3 Encapsulation of Resources

This section explains how we met the challenges involved in the encapsulation of three kinds of resources: data sources (Section 3.1), manipulation software (Section 3.2), and sensor data (Section 3.3).

3.1 Data Sources

We consider two kinds of data sources: stable and dynamic. Stable data sources are characterized by eventual updates, being used here to supply computational models with context data, e.g., the spatial location of sensors' readings (including geographic coordinates or region names), data quality parameters, etc. These sources can also supply metadata, depending on the application. There are many implementations of these sources, including DBMS, XML files, or even web services.

Encapsulation of stable sensor-related data sources into ProcessDCCs is a straightforward application of DCC techniques – see [14, 15]. Sensor-related challenges appear when dynamic access is considered.

Dynamic data sources go through systematic updates. Two kinds of dynamic data sources are considered here: (i) sensing devices and (ii) subscribable services. Sensing devices are encapsulated within a *SensorDCC*, a specialization of a ProcessDCC. Sensor encapsulation is further explored in below.

Subscribable services provide data under some sort of agreement (e.g., upon request). An example is a weather forecasting service which produces updates on the rainfall forecast every hour, or when some threshold is crossed. These services are encapsulated into *DynamicDataSourceDCCs*, which offer access to the same functions of the service plus simulation of stable data source functions. Following the previous example, a DynamicDataSourceDCC encapsulating a rainfall forecast service can generate a notification adapted to each forecast event received. It can also offer, for instance, a summary of the forecasts for a period.

When a sensor is encapsulated within a DCC, its features are exposed through the uniform interface provided by the DCC. The major advantage of this approach is the separation of concerns it provides. On the one hand, there is the problem of developing sensor device drivers, including here other sources of sensing data, such as other middlewares (see Section 3.2.1). On the other hand, using DCCs, application changes do not require driver modification, and sensor (or middleware) changes do not affect applications.

Since each kind of sensing device has a specialized format for outputting data, different drivers must be implemented for distinct platforms, resulting in different SensorDCCs. As an example, a specific SensorDCC implementation had to be built to access TelosB[2] devices (sensor network enabled device). SensorDCCs are similar to proxies to access the data. They allow creating a network of heterogeneous sensors as if they were homogeneous – e.g., in a crop monitoring model, a network with both rainfall and temperature sensors.

If one single sensor is encapsulated, we call it a *SingleSensorDCC*; if a sensor network is encapsulated, we have a *SensorNetworkDCC*. Both are SensorDCCs. A SensorNetworkDCC is responsible for all the sensors it encapsulates. Any message sent to this DCC is relayed to the encapsulated sensors, and it controls the forwarding of the data generated by its sensors. The sensors are not aware of the existence of the DCCs, thus suffering no interference in their functioning.

Figure 1 shows an example of a SingleSensorDCC we implemented, with details omitted. The structure section describes the organization of a software module that communicates with the sensor to access its data, i.e., it is the driver that establishes the communication between the SensorDCC and the sensor. Here, the driver implements a Java communication interface with a MICAz mote coupled with a temperature sensor. The `getTemp` operation receives a time interval in which the readings (floating-point numbers representing Celsius temperatures) will be returned to the caller. The second operation (`subscribeGetTemp`) receives the frequency in which it should pack and send the polled sensor data, until the (`unsubscribeGetTemp`) operation is invoked. The metadata section describes the SensorDCC: `sensorType` indicates the type of sensing device that is encapsulated within the DCC, in this case a TemperatureSensorDCC; `phenomena` indicates which kind of measure the produced data represents; `coverage` shows in which region the sensor is acting; finally, `location` specifies where the sensor is located.

SensorNetworkDCC allow representing an entire network within one DCC. The schematics are similar to the SingleSensorDCC, adapting the structure part to take care of multiple sensors. An external request sent to the DCC (e.g., `getTemp`) is translated into a request that is retransmitted to the sensors (through the wireless network) by the drivers. The query can be answered by every sensor individually or with data condensed within the network. The results make the inverse flow path. A SensorNetworkDCC is particularly valuable in networks that do not univocally identify each sensor, or in situations where it is not interesting or feasible to control each sensor node individually.

A sensor network can actually be encapsulated through its access point or base station. A DCC that encapsulates an access point (*AccessPointDCC*) creates an interface to the entire network. Through these interfaces, the applications can query the sensor network.

[2]`www.xbow.com/Products/productdetails.aspx?sid=252`

3.2 Encapsulation of Data Manipulation Functions

Each SensorDCC can offer individual management methods in its interface – e.g., setting and reconfiguring data generation parameters. However, controlling each SensorDCC on an individual basis may not be feasible. Higher level management layers can be created in order to further facilitate the management of data production and annotation. We thus introduce a specialized DCC, called *ManagementDCC*, which aggregates operations whose implementation is based on the individual management operations of each SensorDCC.

3.2.1 Processing Data

The processing of sensor data requires several specialized functions – e.g., summarization. We encapsulate such functionalities within *ProcessingDCCs*, a specialization of ManagementDCC – located in Layer 3 (centralization) of Figure 3. There are two kinds of ProcessingDCCs: *BasicProcessingDCCs*, which are implemented to directly compute functions on sensor-produced data, and *BridgeDCCs*, which serve as a connection point to sensor middlewares.

The former include functionalities such as data filtering, clustering and classification, application of association rules, multi-source data fusion, data summarization, among others. They, can be combined to obtain more complex processes.

BridgeDCC exist in order to take advantage of the many solutions already implemented in other frameworks. Consider, for instance, TinyDB [11], a popular middleware solution for accessing sensor data. Its SQL-like queries can be offered by operations on a BridgeDCC interface. Instead of having to become familiar with TinyDB, the application sends a request to the corresponding BridgeDCC, which translates it into the appropriate syntax, forwards the request and returns the result. The query proxy system from Cougar [17], or application adaptation update mechanisms from Impala [10], can also be made available through a BridgeDCC's high level interface. The same interface specification can thus serve to access these distinct systems.

3.2.2 Accessing Data

The *AccessDCC* is a ManagementDCC that offers operations for higher level data access. These operations reflect all the features offered by DCCs in lower layers. Consider, the following three examples. A BridgeDCC that implements access to the TinyDB middleware will transform requests from an AccessDCC into TinyDB's SQL-like queries. A SensorDCC that offers access to reprogramming sensor nodes may receive a parameter that contains the compiled software to reprogram the node. A ProcessingDCC that classifies data may receive as a parameter the maximum number of categories desired.

3.3 Encapsulation of Data

This section discusses the *SensorDataDCC*, a specialization of a PassiveDCC that encapsulates sensor generated data (Section 3.3.1), and its specialization, *StreamDataDCC* (Section 3.3.2), which encapsulates streamed data from sensor data sources.

3.3.1 Sensor Data

An infrared satellite image, or a file containing a temporal series of rain data, are typical examples of environmental data to be encapsulated into a SensorDataDCC. Like any DCC, a SensorDataDCC is annotated using ontology terms, e.g., data type, the physical phenomenon being measured (temperature, level of moisture, etc), the geographical location of the reading.

Any sensor generated data can be encapsulated in a SensorDataDCC. One option is to simply encapsulate existing files. Another is to apply filters to these files. Here we consider three factors that limits sensor data encapsulation: size, granularity and time.

From the granularity (number of readings) viewpoint, the options considered by us for creating a SensorDataDCC are: ignore readings (store only metadata from the sensor, for verification of the sensor's capabilities); one reading (a single reading is stored, for instance, the temperature at a given timestamp); a pre-defined number of readings (e.g., the last five readings of the sensor); and an unbounded number of readings, which requires a signal to stop transmission. From the size viewpoint (storage occupied), the options are: a pre-defined limit on the size of the DCC (e.g., using a memory window); and an unbounded size, also requiring a stop signal. From the time viewpoint, the options are: a pre-defined start/stop time limit, time interval or no time limit (any reading can be considered);

We can also combine the dependency among pairs of factors, such as (i) size dependent granularity (e.g., a pre-defined number of messages limited by their total size); (ii) size dependent timing (e.g., a start time for a pre-defined size); or (iii) time dependent granularity (e.g., all the messages within a time frame). Combinations of the three factors are also possible (e.g., all the messages within a pre-defined time and memory windows). Dealing with a stream source includes a fourth dimension to the problem, discussed next.

3.3.2 Streamed Sensor Data

DCCs have, by nature, a closed scope specification, i.e., they are clearly defined and can only be changed by an authorized user. Thus, a *StreamSensorDataDCC* does not directly support data stream encapsulation, rather it uses different strategies to mimic the behavior of stream data. This is a problem for a computational model that needs to handle stream data – e.g., for crop monitoring. Frequently, not all data can be stored, which either requires immediate consumption by applications, or some form of caching, or selective storage. Each of these approaches are implemented with DCCs for dealing with this incompatibility.

Since StreamSensorDataDCCs are passive, streams require a ProcessDCC – say, `P` – to pre-process the data to be encapsulated. In the first approach (forwarding the data for immediate consumption), `P` accesses the stream source only when needed, i.e., when an operation is posted to `P`. In this case, `P` ignores the data stream production until data is requested.

In the caching approach, `P` keeps the data available (e.g., in main memory) using a window of limited size combined with a discarding policy, e.g., first-in-first-out, least-used, or least-recently-used.

In selective storage, `P` polls the stream source in order to acquire the data at some constant rate, and stores it according to one (or more) of these three strategies: (i) directly into a database management system; (ii) in files, using some kind of structure for the data; (iii) `P` sends on the data to one or more additional ProcessDCCs that will take care of the data from that point on. In strategies (i) and (ii) the implementation of the storage must use a selection policy on the datasets to be stored. Examples of such policies include those for caching plus summarization (e.g., only means are stored), sampling methods (only a few values are stored), outlier detection methods (only out-of-range values are stored), and so on. In strategies (ii) and (iii), files can be encapsulated by SensorDataDCCs.

4 Implementation Issues

This section gives an overview of DCC implementation aspects. The presentation concentrates on specific components, thereby exemplifying some of the problems encountered. Code was implemented in Java. Annotations follow the OWL vocabulary and refer mainly to three ontologies: NASA's SWEET [13], POESIA [3] and the DCC ontology [14]. Performance issues are yet to be evaluated with a larger number of sensing devices, data sources and applications.

4.1 Data Encapsulation

This section describes three of the implemented SensorDataDCCs (Section 3.3), namely, *TemperatureSensorDataDCC*, *RainfallMapSensorDataDCC* and *RainfallMapSetSensorDataDCC*.

The TemperatureSensorDataDCC encapsulates a temperature time series stored in a plain text file, whose records are pairs <temperature, timestamp> – both measures as defined by SWEET – captured by a sensor in a given geographic location. There is one TemperatureSensorDataDCC per sensor/location. Operations available include retrieval of one pair, or a sequence thereof, and some summarization computations (e.g., average). For instance, `getTemp(Date begin, Date end)` returns the set of all temperature readings within the time frame from *begin* to *end*. DCC metadata contain location coordinates, the measurement unit (e.g., Celsius degrees), and information on originating sensor.

The RainfallMapSensorDataDCC encapsulates one GeoTIFF file (standard where each pixel corresponds to a given geographical coordinate and contains one rainfall measure for a particular month of a particular year). Its operations concern: getting the entire GeoTIFF file, getting values from individual pixels (given either geographical coordinates or pixel relative position in the image). Metadata are of a similar nature to those provided for the temperature sensor.

We recall that SensorDataDCCs are passive components; thus, the implementation of these DCCs had to be followed by the implementation of ProcessDCCs, which contains the code of all interface operations. The main challenges faced here were determining the appropriate operations and their parameters, and the actual implementation of the operations.

4.2 Data Sources Encapsulation

Here we discuss issues of implementing DCCs described in Section 3.1. Implementation was divided in two parts – creating communication drivers to access data through sensor-specific protocols, and creating the DCCs themselves. The first part required familiarity with the characteristics of each sensor, e.g., to interpret the data packets emitted. This kind of effort is needed for every new kind of sensor device encountered – not only in our approach, but for any environment that wishes to support access to sensor data. The difference to other solutions appears in the second part. Whereas they require extensive communication-oriented coding to forward data delivered by the drivers, DCCs directly map driver operations to interface operations. Thus, coding effort is much smaller – the DCC approach not only supports homogeneous access from external applications, but also simplifies a programmer's work. Challenges in developing these DCCs were mostly associated with semantic annotations (e.g., finding appropriate ontology terms).

We implemented the communication driver using TinyOS [9] interfaces. Sensors were programmed with TinyOS and software developed in the NesC language, a C-derived component programming language.

We implemented SensorDCCs for the MICAz mote (MicazSensorDCC) and for the TelosB mote (TelosbSensorDCC). These SensorDCCs have similar interfaces and were tested with temperature and light readings. The MICAz mote requires an auxiliary MTS300 sensorboard, whereas TelosB came with integrated sensors. The MicazSensorDCC and the TelosbSensorDCC are generating real-time data and making them available in four ways (operations implemented): (i) on-demand access (application receives the data as needed in real time); (ii) generating text file data outputs; (iii) generating new SensorDataDCCs (TemperatureSensorDataDCCs); and (iv) updating existing SensorDataDCCs.

4.3 Examples with sensors

Models in agricultural planning require analysis of several kinds of sensing data, and their comparison along time, to detect trends in climatological values and their relationship with productivity of a given crop (e.g., [12]).

Consider a model that needs to compare rainfall data from two different periods, for São Paulo State, producing a map that shows the result of this comparison. Figure 4 shows a screen capture of this application. It invokes `getMap` operations on two RainFallMapSetSensorDataDCC containing data on São Paulo State. One of these DCC encapsulates one GeoTIFF file, created from historical rainfall data, whereas the other encapsulates a service that provides rainfall

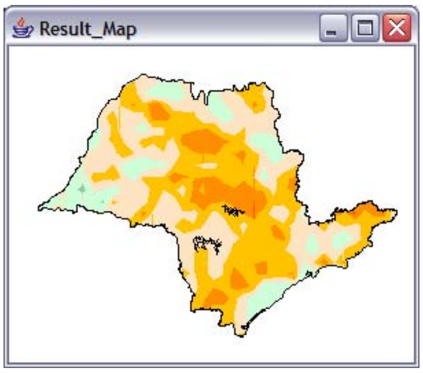

Figure 4. Screenshot comparing rainfall data.

data expressed in the GeoTIFF standard. Once these two datasets are obtained, the model calculates their "difference"; this is obtained by comparing the values of pixels that correspond to the same geographical positions. This may also requires pre-processing the data, to ensure that both datasets are scale and coordinate compatible (information provided by associated DCC metadata).

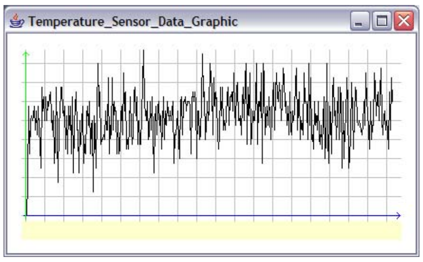

Figure 5. Polling a TemperatureDCC

Monitoring models, on the other hand, process sensing data at real time. Figure 5 shows the result of a loop that polls a MicazSensorDCC, and plots a real time temperature curve, to be interpreted by experts. Polled data is also fed to another set of modules. In agriculture, experts typically compute aggregate values (e.g., average or maximum for a given time window) that are then used in a broader context – e.g., to create a dynamic average temperature map of an area, or to detect deviations from the norm. Such real time data is also stored in temporary files, to re-run the original planning models (e.g., as in the previous example).

We point out that, from an application point of view, all was achieved by invoking DCC operations, regardless of the characteristics of the underlying sensing sources. A planning model will process a sequence of invocations, whereas monitoring will perform a loop that will invoke a given sequence over and over, at specific time intervals.

5 Alternative Solutions

We have commented on related work throughout the paper. There remains to compare our proposal with alternative approaches to data management solutions, in particular to homogeneous access to resources, which we achieve by encapsulating them within DCCs.

The solutions considered are: (i) Specialized (and language specific) implementations; (ii) Standards for data access such as Service Data Objects (SDO)[3] (iii) Software components and communication middlewares, such as CORBA, COM (DCOM and COM+) and .NET , EJB, and others; and, (iv) WSN middleware as defined by [5]. The first approach has the classic overhead of unnecessary repetition of work, hard maintenance, lack of standardization and interoperability. SDO or similar initiatives only take into account data representation; furthermore a DCC can offer access to data using SDO. Components and general middleware lack stream manipulation flexibility, semantic descriptions, and, more importantly, homogeneous treatment of data, devices and software.

Exploring further the WSN middlewares, four approaches are close to ours. *Global Sensor Network* (GSN) [1, 7], has similar goals. However, data management is restricted to homogeneously accessing the network using a declarative language, while our proposal considers including pieces of software in new DCCs. The *Sensor Network Services Platform* (SNSP) [16] proposal considers the possibility of including processing software through the concept of "auxiliary service". However, it is centered around a formal specification of levels and services, leaving aside implementation issues, both for data publishers and data consumers. *Hourglass* [6] concisely considers processing solutions, but requires the use of a specific definition language and uses low-level TCP socket communication schemes, while DCCs use Semantic Web standards for both specification and communication. *IrisNet* [4] limits the use of sensor data to a hierarchical XML database, using XPath queries. DCC interfaces, on the other hand, can support a wide range of access mechanisms.

Other proposals act in more specialized branches. *TinyDB* [11] provides access to an entire network in a single entry point, which uses an SQL-like query system. It also aggregates data readings, decreasing the transmission costs, but at the cost of maintaining information on the network structure, compromising scalability. *Cougar* [17] works well on large sensor sets and

[3]`www.jcp.org/en/jsr/detail?id=235`

makes data access easy with its query system; however, its focus is centered in efficient sensor programming, while our proposal is concerned with the management of data from the sensor outwards. *Impala* [10] is limited to a specific handheld hardware, but supports protocol and operation mode updates. *Maté* [8] addresses issues such as protocol updates and node heterogeneity (using a virtual machine approach), but lacks effective and easy communication with applications. *Magnet* [2] delivers a Java virtual machine on top of the network, facilitating the development of Java applications, but is unfit for nodes with limited capacity. These solutions can be encapsulated in our BridgeDCC.

6 Concluding Remarks

This paper presented a framework to support flexible management and publication of sensor-produced data, an ever-growing need of computational models. Part of this framework has been implemented and validated. It is based on two main aspects: the use of DCCs to provide uniform access to sensor data, sensing devices and software to process the data; and the construction of ManagementDCCs to coordinate sensor data integration and publication.

Through this solution, applications that implement scientific computational models do not need to concern themselves with device dependent issues or with whether they are handling historical data files, or real time data streams. They just need to invoke the appropriate sensor and management DCCs to access the desired data, simplifying the problem of model construction and tuning.

Ongoing work involves several issues. As mentioned in Section 4, we are constructing more SensorDCCs, and extending our implementation to process data from large sensor networks. We are also working on the actual construction of BridgeDCCs to expand the compatibility base for experiments.

Acknowledgements. This work was supported by FAPESP (grant 04/14052-3) and partially financed by CNPq, CAPES and an HP Digital Publishing grant.

References

[1] K. Aberer, M. Hauswirth, and A. Salehi. A Middleware for Fast and Flexible Sensor Network Deployment. In *Proceedings of the 32nd VLDB Conference (Demo Session)*, 2006.

[2] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. G. Sirer. On the need for system-level support for ad hoc and sensor networks. *ACM SIGOPS Operating Systems Review*, 36(2):1–5, 2002.

[3] R. Fileto, L. Liu, C. Pu, E. D. Assad, and C. B. Medeiros. POESIA: An Ontological Workflow Approach for Composing Web Services in Agriculture. *The VLDB Journal*, 12(4):352–367, 2003.

[4] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: An Architecture for a World-Wide Sensor Web. *IEEE Pervasive Computing*, 2(4), 2003.

[5] S. Hadim and N. Mohamed. Middleware Challenges and Approaches for Wireless Sensor Networks. *IEEE Distributed Systems Online*, 7(3), 2006.

[6] J. Shneidman and P. Pietzuch and J. Ledlie and M. Roussopoulos and M. Seltzer and M. Welsh. Hourglass: An Infrastructure for Connecting Sensor Networks and Applications. Technical report, Harvard University, 2004. TR-21-04.

[7] K. Aberer and M. Hauswirth and A. Salehi. The Global Sensor Networks middleware for efficient and flexible deployment and interconnection of sensor networks. Technical report, Ecole Polytechnique Fédérale de Lausanne, 2006. LSIR-REPORT-2006-006.

[8] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Proc. 10th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pages 85–95, 2002.

[9] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *Ambient Intelligence*, chapter TinyOS: An Operating System for Wireless Sensor Networks. Springer Verlag, 2004.

[10] T. Liu and M. Martonosi. Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. In *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP 03)*, 2003.

[11] S. R. Madden, M. J. Franklin, and J. M. Hellerstein. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Transactions on Database Systems*, 30(1):122–173, 2005.

[12] S. Park, J. Feddema, and S. Egbert. Hydroclimatological parameters and their relationship with land surface temperatures (LST) and NDVI anomalies. In *Proc. 2002 ASPRS-ACSM Annual Conference*, 2002.

[13] R. Raskin and M. Pan. Semantic Web for Earth and Environmental Terminology (SWEET). In *Proc. of the Workshop on Semantic Web Technologies for Searching and Retrieving Scientific Data*, 2003.

[14] A. Santanchè and C. B. Medeiros. A Component Model and an Infrastructure for the Fluid Web. *IEEE Transactions on Knowledge and Data Engineering*, 19(2):324–341, 2007.

[15] A. Santanchè, C. B. Medeiros, and G. Z. Pastorello Jr. User-centered Multimedia Building Blocks. *Multimedia Systems Journal*, 12(4):403–421, 2007.

[16] M. Sgroi, A. Wolisz, A. Sangiovanni-Vincentelli, and J. M. Rabaey. *Ambient Intelligence*, chapter A service-based universal application interface for ad hoc wireless sensor and actuator networks. Springer, 2005.

[17] Y. Yao and J. Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. *SIGMOD Record*, 31(3), 2002.