A Software Architecture for Accessing Data in Sensor Networks

Arijit Ghosh and Tony Givargis
Center for Embedded Computing Systems
School of Information and Computer Science
University of California
Irvine, CA 92697
Email: {Arijit.Ghosh,givargis}@uci.edu

***Abstract*—In this paper, we present a software architecture for accessing data in sensor networks. Designing a generic data access system for sensor networks is difficult. This is because the underlying physical architecture of sensor networks cannot be generalized which in turn affects the efficiency of the protocol. Further, applications have unique and often different data requirements. Thus the data service layer should be configurable to satisfy the needs of the application. Recognizing this, we focus on defining a software architecture that specifies the interface but leaves out the implementation. Any protocol can be used to realize this as long as it provides the services that matches the specifications. This is conceptually similar to template based programming where presentation is separated from implementation. We provide the details of our architecture and evaluate both its expressiveness to the application programmer and flexibility of implementation through a few example scenarios.**

I. Introduction

Accessing data is at the heart of sensor network systems. Applications specify what is required and underlying systems figure out a mechanism to serve the request most efficiently. This is typical of most computing systems. However, the fact that sensor nodes potentially generate huge amounts of data in the form of infinite streams coupled with their inherent resource-impoverishness adds a degree of difficulty to data access in sensornets. Typically there has been two approaches to this problem. The first is the more cumbersome method of writing programs in low-level embedded languages while explicitly addressing issues related to wireless communication, resource management and asynchronous event processing. While this might result in more efficient systems, it makes them non-portable and hence the program has to be rewritten for a different sensornet. The second more intuitive, and hence more popular approach, is the declarative paradigm where a user specifies what is required leaving it to the system to decide how to deliver it. Clearly this is very portable in addition to making it extremely easy for aplication programmers. But it makes the task of system programmers exceedingly difficult. This is because sensornet applications have different requirements for data quality, for example accuracy and latency. What is missing is a more *translucent* architecture where an application programmer specifies what is required along with some additional attributes. The system designer provides the implementation but uses the attributes as hints for optimizing system resources. To address this, we present a *Data Access layer in SEnsor Networks*, DASEN (pronounced like the word "dozen"). DASEN is a software architecture that consists of a set of specifications that represent the data subsystem in terms of services. A uniform interface makes applications portable across a wide range of sensornet platforms. The implementation is **not** part of the specification and is provided by the system designers. The specification allows applications to specify data quality attributes which are used by system designers to provide the most efficient implementation depending upon the platform, the resources and the usage pattern in case of multiple applications. Thus DASEN in a wired camera sensor network with multiple applications, for example, can be implemented very differently from a wireless RFID network with a single user. This is very similar to the Standard Template Library in the C++ programming language. Our design goal is to provide a STL-like specification for the DASEN.

The paper is organized as follows. In Section II, we discuss our design goals. In Section III, we present our software architecture. Section IV presents an example of how the data access layer might be used by an application. Section V discusses a couple of possible implementation of the specification, highlighting the fact that implementation is completely decoupled from the presentation and can be chosen based on the underlying system. We present related work in Section VI and conclude with future directions in Section VII.

II. DASEN: Software Architecture Specification

DASEN provides a stream-centric view of the system. Just as a filesystem is a collection of files, DASEN is a collection of streams. Each stream is identified by a unique name and has several attributes. The attributes include geographic location, the type of data in the stream and a timestamp for each data item. The data type can be primitive, for example `int` and `float`, or user-defined complex types. The system provides a type registration facility for this purpose. Data items need not be generated at a fixed rate. By default, streams are non-persistent and read-only.

In order to use a stream, a user requests a *view* of a stream from DASEN. A view is like a container of data items from a stream. It is a snapshot, containing a finite

number of elements, and is the mapping between the infinite nature of the stream and the finite nature of the application. A view can be composed from multiple streams. The user specifies how a composite view is generated by defining a `generate` function. This particular aspect makes DASEN extremely powerful by allowing complex types being defined by the user. A view can have qualities attached to it. These need to be specified by the user. Finally, if a user specifies the lifetime of a view to be infinite and registers the view with the system, then the view *becomes* a stream that is available for use by other applications. This promotes reuse, a key property which allows evolution of existing systems and creation of newer systems.

A view definition contains at least the following:

- *Generate function:* This is the composition function specified by the user. It identifies the source streams and instructs DASEN on how to generate data items of the view by combining the individual data samples from the component streams. The source can be one or more streams that are currently available in the system. A naming and directory service of DASEN provides the user with the details of all currently available streams.
- *Data attributes:* The data items in a view have some attributes. Since this is a user defined type with user-defined qualities, the system needs to know the attributes for making decisions related to routing, scheduling etc. The only required attribute is the timestamp. Optional attributes include the accuracy of the data, timeliness (for example in real-time systems), and tolerance to loss.
- *View properties:* Each view has properties whose specification is mandatory. This is required for life cycle management of the views to optimize resource usage. This is necessary because DASEN is a generic specification for many different kinds of underlying systems. Default behavior can be assumed where the user doesn't specify the properties. The essential properties are the persistence span, the location where the view has to be made available and the mobility specification of the view. The persistence span is the time window with respect to the current time for which data in the view will be available.
- *Accessor methods:* DASEN provides data access function in terms of accessor methods. Data items in the view is accessible by the following calls: `getNext()`, `getAt(time t)`, `getAll()` and `getRange(time start, time end)`. The decision to make the call blocking or not depends on the underlying system.
- *Error handling:* The user needs to specify how DASEN should behave when the system encounters an error in view handling. By default, the system ignores the error.

As can be seen, DASEN allows the user to define a container whose data is composed from available streams in an application specific way. The user can specify where the data is to be made available, optionally with certain accuracy and timeliness properties. The underlying system decides the best way of providing the requested services.

III. Implementing DASEN

As mentioned above, implementation is not part of the specification. In this section, we provide a small example of how different implementations can provide the same set of services and the one chosen will depend upon other factors like the underlying platform.

Let us consider a concurrent multi-user system, for example a camera network in an international airport. The sensornet uses wired networking, and is made up of resource-rich nodes similar to Intel iMote running embedded Linux. Let M views be requested where each view is composed of one or more streams. All the views are static. Considering the fact that we are dealing with video data, there are soft real-time requirements on the timeliness and accuracy of the data. As such, we use TCP/IP as the underlying networking protocol. We assume the existence of a stream lookup service. The problem can now be formulated as below:

Given N streams and $M \subset N$ views, where each view is composed of S streams ($1 \leq S \leq N$), find a data distribution plan that minimizes the latency and jitter of each view.

We assume that all streams generate data at a fixed rate. For simplicity, we assume that the generating function is associative. We refer to nodes that produce the streams as producers and the nodes that request the view as consumers.

IP Multicast: An efficient way to solve this problem is to use multicasting. A shortest path tree rooted at each of the producers with the consumers as the leaves will ensure the fastest delivery time. Let us take the example of a network with 9 nodes (Figure 1). Nodes 1, 2 and 8 are the producers. Nodes 5 and 9 are the consumers and each request all the three producers. Node 8 will generate 2 packets which will follow the paths 8-7-9 and 8-4-5. Nodes 1 and 2 will generate a single packet which will traverse through node 3 and reach 4. At 4, 2 copies are generated. One flows along 4-5 while the other flows along 4-6-7-9. Nodes 5 and 9 can generate the data elements in their view by combining the data from the producers.

MergeCast: While the above approach is very efficient, we can improve it further by introducing in-network functionality. Observe that the link 3-4 transmits data from 1 and 2 which are eventually merged. If the merging function could be performed at 3 instead of at the consumers, then we could reduce the traffic. We call this technique MergeCast. As before, producers still send data through multicasting. However, intermediate nodes now merge data opportunistically to reduce traffic. In our example, 3 will merge data from 1 and 2 before sending it to 4. Node 4 will merge this data with that from 8 before sending it to 5. In addition to merging data, MergeCast also allows multicasting of merged data where possible. For example, merged data from 1 and 2 can be multicast to 5 and 9 along the shortest path tree rooted at 3.

A. Comparison:

To compare the two approaches, we evaluated the protocols using a packet-level simulator written in C++. We generated

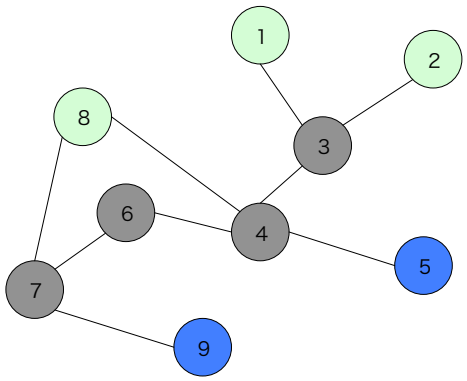

Fig. 1. Example Graph

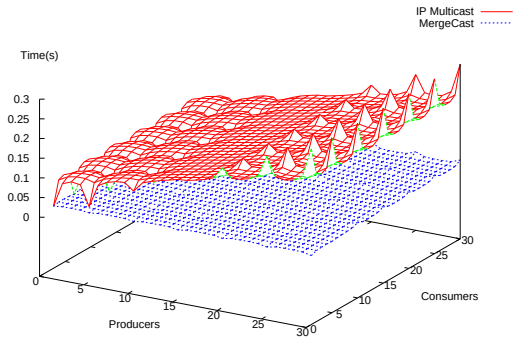

Fig. 4. Average Jitter

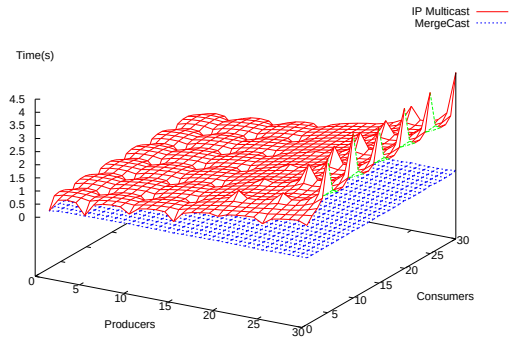

Fig. 2. Average Startup Delay

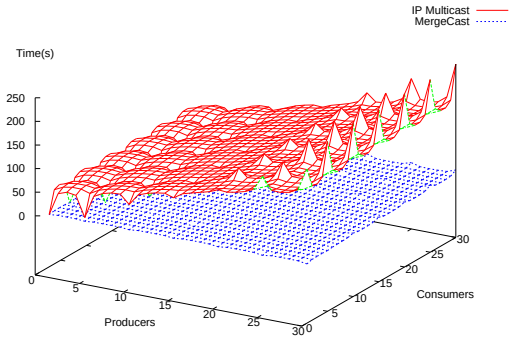

Fig. 3. Average Latency

a random graph of 200 nodes using Georgia Tech's GT-ITM topology generator [11]. We randomly chose from one upto thirty producers. For each producer, we varied the number of clients from one to thirty. Each client chose an arbitrary subset of producers. In all simulations, 10,000 packets, each of 1500 bytes, are generated at 500 Kbps. We assume the existence of a network wide time synchronization protocol. Our simulations are run on a `sparcv9` 64-bit processor operating at 1503 MHz, with 2048 MB of memory and running `SunOS`. We measure the following: *latency* (Figure 3) is the average difference between the time at which the data item at the stream was produced and the time at which the corresponding item in the view was generated. *startup delay* (Figure 2) which is the latency of just the first item; *jitter* (Figure 4) is the time difference between successive data items in a **view** and the *total messages* (Figure 5) that traverse the network. The common trend across all graphs is that IP Multicast and MergeCast are comparable when the numbers of both consumers and producers are few but diverges as the numbers increase.

There are two key takeaways from the above experiments. One is obviously the trend of MergeCast being more efficient than IP Multicast across various configurations. As the number of producers and consumers increase, the efficiency gain is even more noticeable. The second observation is that the performance of MergeCast shows a much gentler degradation than that of IP Multicast. The MergeCast surfaces look practically smooth in all the plots, although the same cannot be said about multicast.

While MergeCast is definitely the more efficient of the two, that still doesn't make it an automatic choice. The reason being that MergeCast requires changes to the networking protocol to enable in-network processing. While considered an obvious choice by researchers in wireless sensor networks deployed in remote monitoring, it might prove to be an entry barrier where sensor networks are planned to be deployed as an extension to existing IP networks. In our example from the graph above, if we imagine nodes 3 and 4 to be part of the Internet, then MergeCast cannot be deployed, at least not at

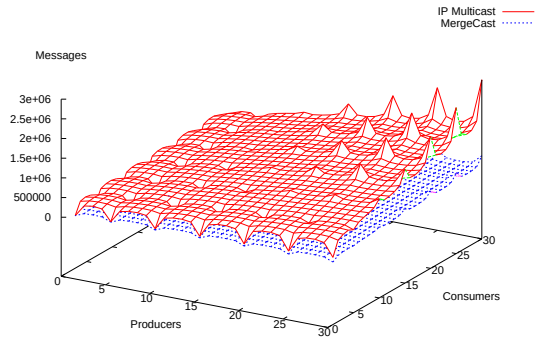

Fig. 5. Total Messages

the network layer, and IP Multicast remains the most viable implementation.

The objective of the above discussion is to emphasize the fact that our software architecture provides the flexibility to choose an implementation that suits the network. The physical properties and practical considerations would dictate the choice of IP Multicast, or MergeCast, or some other protocol. However, that does not affect the users view of the underlying system and a system once composed can be reused in any kind of sensornet.

IV. Related Work

There has been a substantial amount of research in data management systems in sensor networks. These can be broadly divided into two categories: querying live data and querying historical data. TinyDB[7], Cougar[8] and Directed Diffusion[9] provide the feature to push down continuous queries into the network such that data processing is performed close to where the data is sensed. This provides a savings of energy, since only the end results are communicated, and thereby increases the life time of sensor networks. A related and interesting technique is called Acquisitional Query processing. In these systems, typically data models are maintained at the base station. All queries are first answered using this model. If error and confidence measures are violated, then the data is intelligently gathered from only those nodes that will increase the confidence value. BBQ[10] is a query system that is acquisitional in nature. Historical query processing could be performed by blasting all the data from all nodes to a base station where they are stored and indexed. In contrast, a more intelligent approach is to build a sensor database inside the network. The StonesDB[1] project does this by exploiting the increasing capacity of NAND flashes. It employs a two level structure where the lower tier of sensor nodes that provides the storage substrate and an upper tier of proxy nodes that deal with query planning and optimizations.

A different approach to data management is to provide a file system view. In [2], the authors present a file system abstraction based on the Plan 9 design principles. In this, multiple logical and application-specific views are maintained through the filesystem namespace. In [3], the authors provide a distributed data storage abstraction using geographic hash tables. GHT hashes keys into geographic coordinates and stores a key-value pair at a sensor node closest to this coordinate. Capsule[4] and MicroHash[5] offers a way of accesinf flash filesystems. DALi[6] is a data abstraction layer that is inserted between the application layer and the file system. It is a two-tier data hierarchy where data is organized to optimize communication cost for search and retrieval.

V. Conclusion

In this paper, we present a software architecture for accessing data in sensor networks. The architecture provides a STL-like template for data access in sensor networks. It provides a stream-centric view of the system. Programmers generate application specific data types by defining views that are composed of one or more streams. The system allows user specification of quality of data and properties of the view. Implementation details are hidden from the application thereby promoting portability. Generated views can in turn be made available to other systems which promotes reuse. As part of our future work, we intend to explore the design space of protocols and investigate the possibility of automatic protocol selection.

References

[1] Yanlei Diao, Deepak Ganesan , Gaurav Mathur and Prashant Shenoy, Re-thinking Data Management for Storage-centric Sensor Networks, Proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, January 2007.

[2] S. Tilak, B. Pisupati, K. Chiu, et al. A File System Abstraction for Sense and Respond Systems. In Workshop on End-to-End, Sense-and-Respond Systems, Applications, and Services, 2005.

[3] S. Ratnasamy, B. Karp, S. Shenker, et al. Data-Centric Storage in Sensornets with GHT, a Geographic Hash Table. In Mobile Networks and Applications (MONET), Journal of Special Issues on Mobility of Systems, Users, Data, and Computing: Special Issue on Algorithmic Solutions for Wireless, Mobile, Ad Hoc and Sensor Networks, 2003.

[4] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Capsule: An Energy-Optimized Object Storage System for Memory-Constrained Sensor Devices. In Proc. of the ACM Conf. on Embedded Networked Sensor Systems (SenSys), Nov. 2006.

[5] S. Lin, D. Zeinalipour-Yazti, V. Kalogeraki, et al. Efficient Indexing Data Structures for Flash-Based Sensor Devices. In ACM Transactions on Storage, 2006.

[6] C. Sadler and M. Martonosi, DALi: A Communication-Centric Data Abstraction Layer for Energy-Constrained Devices in Mobile Sensor Networks, Proceedings of the ACM Conference on Mobile Systems, Applications, and Services (MobiSys) 2007, June 2007.

[7] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tinydb: An acqusitional query processing system for sensor networks. ACM TODS, 2005.

[8] Y. Yao and J. E. Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. In SIGMOD Record, Vol 31 Number 3, Sept 2002.

[9] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In ACM/IEEE Mobicom, pages 5667, Boston, MA. Aug, 2000.

[10] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In VLDB, 2004.

[11] www.cc.gatech.edu/projects/gtitm/