International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 02 Issue: 08 | Nov -2015 www.irjet.net p-ISSN: 2395-0072

SUPPORTIVE SQL INTERFACE FOR WIRELESS AD HOC NETWORKS

Pabboju Ramesh[1]

[1]Associate Professor, Department of C.S.E.
Mahaveer Institute of Science and Technology, Bandlaguda, Hyderabad – 59

Abstract-We express the database community's impression of a a SQL Interface for data aggregation, which can be applied to ad-hoc wireless sensor networks. Here, we are showing group aggregations can be effectively processed where they reduce data duplication. Network Traffic. Based on these queries, we shown SQL Interface that can execute queries within ad-hoc sensor networks.

Keywords: optimization, SQL-Interface, Ad hoc networks.

I.INTRODUCTION

At UC Berkeley,researchers have developed small sensor devices called motes, and an operating system, called Tiny OS, that is especially suited to running on them. Motes are equipped with a radio, a processor, and a suite of sensors. TinyOSmakes it possible to deploy ad-hoc networks of sensors that can locate each other and route data without any a priori knowledge of network topology.

As various groups around the country have begun to deploy large networks of sensors, a need has arisen for tools to collect and query data from these networks. Of particular interest are aggregates operations which summarize current sensor values in some or all of a sensor networks. For example given a dense network of thousands of sensors querying temperature, users want to know temperature patters in relatively large regions encompassing tens of sensors individual sensor readings are of little value.

Sensor networks are limited in external bandwidth, i.e. how much data they can deliver to an outside system. In many cases the externally available bandwidth is a small fraction of the aggregate internal bandwidth. Thus computing aggregates in network is also attractive from a network performance and longevity standpoint: extracting all data over all time from all sensors will consume large amounts of time and power as each individual sensor's data is independently routed through the network. Previous studies have shown that aggregation dramatically reduces the amount of data routed through the network, increasing throughput and extending the life of battery powered sensor networks as less load is placed on power hungry radios.

In this paper, we discuss the challenges associated with implementing the five basic database aggregated with grouping in adhoc networks of sensors. We show how our generic approach leads to a significant power savings. Further, We show that sensor network queries can be structured as time series of aggregates, and how such queries adapt to the changing network structure. We have implemented earlyversions of these techniques and are in the process of experimentally validation them.

1.1MOTES

These devices are equipped with a 4Mhz Atmel microprocessor with 512 bytes of RAM and 8KB of codespace,a 917 MHz RFM radio running at 10 KB OF CODE SPACE, A 917 mhz rfm RADIO RUNNING AT 10KB/S, AND 32KBEPROM. Current temperature options include light, temperature, magnetic field, acceleration, vibration, sound, power. The effective lifetime of the sensor is determined by its power supply. In, Motes we will use Tuny OS. TinyOS provides a number of services like simplifying the programs, process the capture data, transmitting radio messages over radio.

1.2 Ad-Hoc Sensor Networks.

Messages in the current generation of TinyOS area fixed size preprogrammed into sensors, by default, 30 byte messages are used. Each message type has a message id that distinguishes it fromother types of messages. Sensor programmers write message id specific handlers that are invoked by Tiny OS when a message of the appropriate id is heard on the radio. Each sensor has a unique sensor id that distinguishes it from other sensors. All messages specify their recipient, allowing sensors to ignore messages not intended for them, although non-broadcast messages must still be received by all sensors within range –unintended recipients simply drop messages not addressed to them.

Given this brief primer on wireless sensor communication, we now show how sensors route data. The technique we adopt is to build a routing tree. We appoint one sensor to be the root. The root is the point from which the routing tree will be built, and upon which aggregated data will converge. Thus, the root is typically the sensor that interfaces the querying user to the rest of the network. The root broadcasts a message asking sensors to organize into a routing tree; in that message it specifies its own id and its level, or distance from the root, which is zero. Any sensor that hears this messages assigns its own level to be the level in the message plus one, if its current level is not already less than or equal to the level in the message. It also chooses the sender of the

message as its parent, through which it will route messages to the root. Each of these sensors then rebroadcasts the routing message, inserting their own ids and levels. The routing, message floods down the tree in this fashion, with each node rebroadcasting the message until all nodes have been assigned a level and a parent. Nodes that hear multiple parents choose one arbitrarily, although we will discuss approaches in below where multiple parents can be used to improve the quality of aggregates. These routing messages are periodically broadcast from the root, so that the process of topology discovery goes on continuously. This constant topology maintenance makes it relatively easy to adapt to network changes caused by mobility of certain nodes, or to the addition or deletion of sensors; each sensor simply looks at the history of received routing messages, and chooses the "best" parent, while ensuring that no routing cycles are created with that decision.

This approach makes it possible to efficiently route data towards the root. When a sensor wishes to send a message to the root, it sends the message to its parents, which in turn forwards the message on to its parent, and so on, eventually reaching the root. This approach doesn't address point-to-point routing; however, for our purposes, flooding aggregation requests and routing replies up the tree to the root is sufficient., as data is routed towards the root, it can be combined with data from other sensors to efficiently combine routing and aggregation. First, however we describe how aggregates are expressed in database systems.

II. AGGREGATION IN DATABASE SYSTEMS

Aggregation in SQL based database systems is definedby an aggregate function and a grouping predicate. The aggregate function specifies how a set of values should be combined tocompute an aggregate; the standard set of SQL aggregate functions is COUNT, MIN, MAX, AVERAGE, and SUM. These compute the obvious functions; for example, the SQ statements.

SELECT AVERAGE FROM ALL_sensors

Computes the average temperature from some table sensors, which represents a set of sensor readings that have been read into the system. Similarly, the COUNT functions compute minimal and maximal values and SUM calculates the total of all values. Additionally, most database systems allow user-defined functions that specify more complex aggregates than the five listed above.

Grouping is also a standard feature of database systems. Rater than merely computing a single aggregate value over the entire set of data values, a grouping predicate partitions the values into groups based on some attribute. For example, the query;

```
SELECT TRUNC, DATA_AVERAGE
FROM ALL sensors
GROUP BY TRUNC
HAVING DATA_AVERAGE> 60
```

Partition sensor readings into groups according to their temperature reading and computes the average light reading within each group. The HAVING clause excludes groups whose average light reading are less than or equal to 60.

In the rest of this paper, we discuss the challenges associated with implementing the five basic aggregates with grouping in ad-hoc networks of Tiny OS sensors. We start by considering a single aggregate being computed at a time, and then argue that often users are interested in viewing aggregates as sequences of changing values over time. Throughout this work, we will assume the user is stationed at a desktop- class PC with ample memory. Despite the simple appearances of this architecture, there are a number of difficulties presented.

III. GENERIC AGGREGATION TECHNIQUES

A native implementation of sensor network aggregation would be to use a centralized, server-based approach where all sensor readings are sent to the host PC, which then computes the aggregates. However, as was shown in a distributed, in wireless network approach where aggregates are partially or fully computed by the sensors themselves as readings are routed through the network towards the host-PC can be considerably more efficient. In this section, we focus on the in network approach, because, if properly implemented, it has the potential to be both lower latency and lower power than the server based approach.

To illustrate the potential advantages of the in network approach; consider the simple example of computing an aggregate over a group of sensors arranged as shown in figure1. Dotted lines represent connection between sensors; solid lines represent the routing tree imposed on top of this graph to allow sensors to propagate data to the root along a single path. In the centralized approach, each sensor value mist be routed to the root of the network; for a node at depth m, this requires n-1 messages to be transmitted per sensor. The sensors in figure 2 have been labeled with their distance from the root; summing these numbers gives a total sixteen messages required routing all aggregation information to the root. Combine their own readings with the readings of their reading to their parents. Intermediate nodes combines their own reading with the reading of their children via the aggregation function f and propagate the partial aggregate, along with any extra data required to update the aggregate, up the tree.

The amount of data transmitted in this solution depends on the aggregate. Consider the AVERAGE function. at each intermediate node n, the sum and count of all children's

sensor readings are needed to compute the assume that, in the case of AVERAGE, both pieces of information will easily fit into a single 30 byte message. Thus a total of 5 messages needed to be be sent for the average function. IN the case of the other standard SQL aggregates, no additional state is required: COUNT, MIN, MAX and SUM can be computed by a parent node given sensor or partial aggregate values at all of the child nodes.

Aggregates can be expressed as an aggragte function f over the sets a & b such that

$$f(a \cup b) = g(f(a), f(b))$$

IV. INJECTING A QUERY

Computing Aggregation consist of two phases: a propagation phase in which aggregations are pushed down into sensor networks, and an aggregation phase, where the aggregate values prop aged from the children to parents. The basic approach to propagation works just like the network discovery algorithm, except that leaf nodes that propagates to their parents. Thus, when a sensor p receives an aggregate a, either from another sensor or from the user, it transmit a and begins listening. If p has any children, it will hear those children re-transit a to their children, and will know it is not a leaf. If, after some time interval t, p has heard no children, it concludes is is a leaf and transmit its current sensor value up the routing tree. If p has children, it assumes they will al report within time t, and so after time t it computes the value of applied to its own value and the values of its children and forwards this partial aggregate to its parent.

IV. STREAMING AGGREGATES

Sensor networks are inherently unreliable: individual radio transmission can fail, nodes can move, and so on. Thus, it is very hard to guarantee that a significant portion of a sensor network was not detached during a particular aggregate computation. Consider, for example, what happens when a sensor, p, broadcast a and its only child c, somehow misses a message P will never hear c rebroadcast, and will assume that it has no children and that it should forward only its own sensor value. The entire network below p is thus excluded from the aggregation computation, and the end result is probably is in correct. Indeed, when any sub tree of the graph can full in this way, it is impossible to give any guarantees about the accuracy of the result.

One solution to this problem is to double-check aggregates by computing them by multiple times. The simplest way to do this would be request the aggregate be computed multiple times at the root of the network; by observing the common-case value of the aggregate, the client could make a reasonable guess as to its true value. The problem with this technique is that it requires retransmitting the aggregate request down the network multiple times, at a significant message overhead, and the user must wait for the entire aggregation interval for each additional result.

Better approach is pipelined aggregate, in the pipe lined approach, time is divided into intervals of duration I, during each interval, every sensor that has heard the request to aggregate transmit a partial aggregate by applying a to its local reading and the values its children reported during the previous interval. Thus, after the first interval, the root hears from the sensors one and two missed the request to begin aggregation, a sensor that hears another sensor reporting the its aggregate values can assume it too should begin reporting its aggregate value.

VI. GROUPING

The basic technique for grouping is to push down a set of predicates that specify group membership, ask sensors to choose the group they belong to, and then, as answers flow back, update the aggregate values in the appropriate groups. Group predicates are appended to requests to begin aggregation. If sending all predicates requires more storage than will fit into a single message, multiple messages are sent. Each group predicate specifies a group id, a sensor attribute (e.g. light, temperature), and a range of sensor values that define membership in the group. Groups are assumed to be disjoint and defined over the same attribute, which is typically not the attribute being aggregated. Because the number of groups can be large enough such that information about all groups does not fit into the RAM of any one sensor, sensors pick the group they belong to as messages defining group predicates flow past and discard information about other groups. Messages containing sensed values are propagated just as in the pipelined approach described above. When a sensor is a leaf, it simply tags the sensor value with its group number. When a sensor receives a message from a child, it checks the group number. If the child is in the same group as the sensor, it combines the two values just as above. If it is in a different group, it stores the value of the child's group along with its own value for forwarding in the next interval. If another child message arrives with a value in either group, the sensor updates the appropriate aggregate. During the next interval, the sensor will send out the value of all groups it collected information about during the previous interval, combining information about multiple groups into a single message as long as the message size permits. Figure shows an example of computing a query grouped by temperature that selects average light readings. In this snapshot, data is assumed to have filled the pipeline, such that results from the bottom of the tree have reached the root. Recall that SQL queries also contain a HAVING clause that constrains the set of groups in the final query result by applying a filtration predicate to each group's aggregate value. We sometimes pass this predicate into the network along with partitions. The predicate is only sent into the network if it can potentially be used to reduce the number of messages that must be sent:

for, example, if the predicate is of the form MAX(attr)<x, then information about groups with MAX(attr)>= x need not be transmitted up the tree, and so the predicate is sent down into the network. However, other HAVING predicates, such as those filtering AVERAGE aggregates, or of the form MAX(attr)>x, cannot be applied in the network because they can only be evaluated when the final group-aggregate value is known. Because the number of groups can exceed available storage on any one sensor, a way to evict groups is needed. Once an eviction victim is selected, it is forwarded to the sensor's parent, which may choose to hold on to the group or continue to forward it up the tree. Because groups can be evicted, the user workstation at the top of the network may be called upon to combine partial groups to form an accurate aggregate value. Evicting partially computed groups is known as partial pre-aggregation, as described in the database literature. There are a number of possible policies for choosing which group to evict. We believe that policies which incur a significant storage overhead (more than a few bits per group) are undesirable because they will reduce the number of groups that can be stored and increase the number of messages that must be sent. Evicting groups with low membership is likely a good policy, as those are the groups that are least likely to be combined with other sensor readings and so are the groups that benefit the least from in-network aggregation. Evicting groups forces information about the current time interval into higher level nodes in the tree. Since in the standard pipelined scheme presented above, aggregates are computed over values from the previous time interval, this presents an inconsistency. We believe, however, that this will not dramatically effect aggregates; verifying this remains an area of future work. Thus, we have shown how to partition sensor readings into a number of groups and properly compute aggregates over those groups, even when the amount of group information exceeds available storage in any one sensor.

VII. CONCLUSION

We have explained techniques for applying database style aggregates with groups to sensor readings flowing through ad hoc sensor networks. By applying generic aggregation operations in the tradition of database systems, our approach offer the ability to query arbitrary data in a sensor network without custom building applications by pipelining the flow of data through the sensor network, we are able to robustly compute aggregates while providing raid and continuous updates of their value to the user.

Finally, by snooping on messages in the shared channel and applying techniques for hypothesis testing, we are able to substantially improve the performance of our basic approach. we have

This work mars a first step towards a generic, in network approach for collecting and computing over sensor data. SQL, as it has developed over many years, has proven to work well in the context of database systems. When properly applied to sensor networks, will offer similar benefits as SQL: ease of use, expressiveness, and a standard on which research and industry can build.

REFERENCES:

[1]P.-A. Larson. Data reduction by partial preaggregation. In *ICDE*, 2002. (to appear).

[2]S. Madden and M. J. Franklin. Fjording the stream: An architechture for queries over streaming sensor data. In *ICDE*, 2002. (to appear.

[3]A. Shatdal and J. Naughton. Adaptive parallel aggregation algorithms. In *ACM SIGMOD*, 1995.

[4]D. Tennenhouse. Active networks. In *OSDI*, October 1996.

[5]B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris. Span: An energy-efficient coordination algorithm for topology maintenance in ad-hoc wireless networks. In *ACM MobiCom*, July 2001.