Data Management in Sensor Networks

Ellen Munthe-Kaas
Outline

• Sensor networks
  – Characteristics
  – Motes
  – Application domains
  – Data management
• TinyOS
• TinyDB
Sensor Networks

Base station (gateway)

Motes (sensors)
Sensor Network Characteristics

• Autonomous nodes
  – Small, low-cost, low-power, multifunctional
  – Sensing, data processing, and communicating components

• Sensor network is composed of large number of sensor nodes
  – Proximity to physical phenomena
    • Deployed inside the phenomenon or very close to it

• Monitoring and collecting physical data

• No human interaction for weeks or months at a time
  – Long-term, low-power nature
**Motes**

Mica2 mote with 2 AA batteries  
(provide power for one year’s use)

Mica2DOT mote.  
Powered with button battery

Spec smart dust; total size 5 mm²
Mote Hardware

• Made up of four basic components
  – sensing unit
    • usually two subunits: sensor and ADC
  – processing unit
    • makes the sensor collaborate with the other nodes to carry out the assigned sensing tasks
  – transceiver unit
    • connects the node to the network
  – power unit
    • small, standard batteries
Motes in the DMMS Laboratory

Mica2

- **Processor**: MPR400CB based on Atmel ATmega128L
- **Radio**: 900 MHz multi-channel transceiver
- **Memory**: 4 kB Configuration EEPROM
  128 kB Program Flash Memory
  512 kB Measurement (Serial) Flash
- **Power**: 2 x AA
- **OS**: TinyOS v1.0
- **Weight**: 18g (excluding batteries)
- **Sensors**:
  - Light
  - Temperature
  - Acoustic
- **Actuators**:
  - Sounder
Motes vs. Traditional Computing

- Embedded OS
- Lossy, ad hoc radio communication
- Sensing hardware
- Severe power constraints
Application Domains

- Environmental
- Health
- Military
- Commercial
Environmental Applications

- Tracking the movements of birds, animals, insects
- Monitoring environmental conditions that affect crops and livestock
- Chemical/biological detection
- Biological, earth, and environmental monitoring in marine, soil, and atmospheric contexts
- Meteorological or geophysical research
- Pollution study, precision agriculture, irrigation
- Biocomplexity mapping of environment
- Flood detection, forest fire detection
Health Applications

• Integrated patient monitoring
• Telemonitoring of human physiological data
• Tracking and monitoring doctors and patients inside a hospital
• Tracking and monitoring patients and rescue personnel during rescue operations
Military Applications

• Monitoring friendly forces, equipment and ammunition
• Battlefield surveillance
• Reconnaissance of opposing forces and terrain
• Nuclear, biological and chemical (NBC) attack detection and reconnaissance
Commercial Applications

- Monitoring product quality
- Constructing smart office spaces
- Interactive toys
- Smart structures with sensor nodes embedded inside
- Machine diagnostics
- Interactive museums
- Managing inventory control
- Environmental control in office buildings
- Detecting and monitoring car thefts
- Vehicle tracking and detection
Application Examples

**Habitat Monitoring:**
Storm petrels on Great Duck Island, microclimates on James Reserve.

**Vehicle detection:** sensors along a road, collect data about passing vehicles.

**Earthquake monitoring in shake-test sites.**

Traditional monitoring apparatus.
Managing Data

• **Purpose of sensor network:** Obtain real-world data
  – Extract and combine data from the network

• **But:** Programming sensor networks is hard!
  – Months of lifetime required from small batteries
  – Lossy, low-bandwidth, short range communication
  – Highly distributed environment
  – Application development
  – Application deployment administration
Data Management Systems for Sensor Networks
Data Management Systems for Sensor Networks

• Motivation:
  – Implement data access
    • Sensor tasking
    • Data processing
    • Possibly support for data model and query language

• Goals:
  – Adaptive
    • Network conditions
    • Varying/unplanned stimuli
  – Energy efficient
    • In-network processing
    • Flexible tasking
    • Duty cycling
Data Management System Challenges

- Routing
- Resource allocation
- Deployment
- Query language, query optimization
Outline

• Sensor networks
• TinyOS
• TinyDB
TinyOS

- Operating system for managing and accessing mote HW
- Characteristics:
  - Energy-efficient
  - Programming model: Components
  - Only one application running at a time
  - No process isolation or scheduling
  - No kernel
  - No protection domains
  - No memory manager
  - No multithreading
- Programming language: nesC
Outline

• Sensor networks
• TinyOS
• TinyDB
  – Overview
  – Data model
  – Query language
  – Architecture
  – Network administration
  – Aggregates
TinyDB

- High level abstraction
  - Data centric programming
  - Interact with sensor network as a whole
  - Extensible framework
- Under the hood:
  - Intelligent query processing: query optimization, power efficient execution
  - Fault mitigation: automatically introduce redundancy, avoid problem areas
Feature Overview

• Declarative SQL-like query interface
• Metadata catalog management
• Multiple concurrent queries
• Network monitoring (via queries)
• In-network, distributed query processing
• Extensible framework for attributes, commands and aggregates
• In-network, persistent storage
Query Language Essentials

• Declarative queries
  – Simple, SQL-like queries
  – Users specify the data they want and the rate at which data should be refreshed
  – Using predicates, not specific addresses

• TinyDB collects data from motes in the environment, filters it, aggregates it, and routes it out to a PC

• TinyDB does this with power-efficient in-network processing algorithms
Outline

• Sensor networks
• TinyOS
• TinyDB
  – Overview
  – Data model
  – Query language
  – Architecture
  – Network administration
  – Aggregates
Data Model

- Relational model
- Single table `sensors`
  - One column (attribute) per type of value that a device can produce (light, temperature,...)
  - One row (record) per node per instant in time
  - Physically partitioned across all nodes in the network
  - Records are materialized only at need and stored only for a short period or delivered directly to the network
  - Projections and transformations of tuples from `sensors` may be stored in materialization points
The sensors Table

sensors(epoch, nodeid, volume, light, temp, ...)

SELECT nodeid, light
FROM sensors
WHERE light > 400
SAMPLE PERIOD 1s
TinyDB Routing Tree

PC side

Mote side

Sensor network

TinyDB query processor

TinyDB GUI

TinyDB Client API

JDBC

DBMS
TinySQL Routing Example

```
SELECT nodeid, light
FROM sensors
WHERE light > 400
SAMPLE PERIOD 1s
```

- **TinySQL Routing Example**

  - **sensors(eepoch, nodeid, volume, light, temp, ...)**

<table>
<thead>
<tr>
<th>epoch</th>
<th>timeStamp</th>
<th>nodeid</th>
<th>volume</th>
<th>light</th>
<th>temp</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>...</td>
<td>1</td>
<td>null</td>
<td>410</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>2</td>
<td>...</td>
<td>2</td>
<td>null</td>
<td>460</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

- **TinySQL Routing Example**

  - **TinySQL Routing Example**

  - **TinySQL Routing Example**

<table>
<thead>
<tr>
<th>nodeid</th>
<th>light</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>410</td>
</tr>
<tr>
<td>2</td>
<td>460</td>
</tr>
</tbody>
</table>

- **TinySQL Routing Example**

  - **TinySQL Routing Example**

  - **TinySQL Routing Example**

<table>
<thead>
<tr>
<th>nodeid</th>
<th>light</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>null</td>
</tr>
</tbody>
</table>

- **TinySQL Routing Example**

  - **TinySQL Routing Example**

  - **TinySQL Routing Example**

<table>
<thead>
<tr>
<th>nodeid</th>
<th>light</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>460</td>
</tr>
</tbody>
</table>
Outline

• Sensor networks
• TinyOS
• TinyDB
  – Overview
  – Data model
  – **Query language**
  – Architecture
  – Network administration
  – Aggregates
TinySQL Example 1

- **Sample interval**: Interval during which exactly one tuple of `sensors` is produced per node for the purpose of executing the query, and during which the query is executed once.
- **Epoch**: Period of time between the start of each sample interval. Numbered consecutively.
- **Data collection period**: Period of time over which query is running.

"Report light and temperature readings once per second for 10 seconds."

```
SELECT nodeid, light, temp
FROM sensors
SAMPLE PERIOD 1 s FOR 10 s
```
TinySQL Example 2

- **Materialization point:** Stored table in the nodes. Cf. *materialized views* in traditional RDBSs and *windows* in data stream management systems.

"Store the latest eight light readings, doing one reading every 10 seconds (forever).”

```sql
CREATE STORAGE POINT recentLight SIZE 8
AS (SELECT nodeid, light
    FROM sensors
    SAMPLE PERIOD 10 s)

... later:
DROP STORAGE POINT recentLight
```
TinySQL Example 3

- **Joins** are allowed between two materialization points or between a materialization point and the `sensors` table.
  - New `sensors` tuples are joined with tuples of the materialization point on their time of arrival.

```
"Count the number of recent light readings (from zero to eight samples in the past) that were brighter than the current reading, each current reading collected during a time span of ten seconds."

```

```
SELECT COUNT(*)
FROM sensors AS s, recentLight AS rl
WHERE rl.nodeid = s.nodeid AND s.light < rl.light
SAMPLE PERIOD 10 s
```

<table>
<thead>
<tr>
<th>nodeid</th>
<th>light</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>345</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>347</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>540</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>533</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>514</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>412</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nodeid</th>
<th>light</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>435</td>
</tr>
</tbody>
</table>
TinySQL Example 4

- **Aggregation** can be performed on grouped values as in ordinary SQL. Grouping and aggregation take place over the tuples collected during each sample interval.

"Find the rooms where the average volume is over some threshold (assuming each room can have multiple sensors). Do this every 30 seconds."

```sql
SELECT room, AVG(volume) 
FROM sensors 
WHERE floor = 6 
GROUP BY room 
HAVING AVG(volume) > threshold 
SAMPLE PERIOD 30 s 
```

... later:
```
STOP QUERY id 
```
TinySQL Example 5

- An event can be used for initiating data collection.
  - Generated by another query or by a lower-level part of the OS

"When a bird-detect event occurs, report the average light and temperature levels at sensors near the event’s location. Do this every 2 seconds for a period of 30 seconds (then go to sleep again)."

ON EVENT bird-detect(loc):
SELECT event.loc, AVG(light), AVG(temp)
FROM sensors AS s
WHERE dist(s.loc, event.loc) < 10 m
SAMPLE PERIOD 2 s for 30 s
TinySQL Example 6

• **Generating an event** from a query:

  "Signal the event **hot** whenever the temperature goes above some threshold. Read the temperature every 10 seconds."

  ```sql
  SELECT nodeid, temp
  FROM sensors
  WHERE temp > threshold
  OUTPUT ACTION SIGNAL hot(nodeid, temp)
  SAMPLE PERIOD 10 s
  ```
TinySQL Example 7

• To make sure the network runs for a guaranteed period, users may request a specific query **lifetime**.

"Get the temperature, but space out the readings to make sure that the network will survive at least 30 days."

```
SELECT nodeid, temp
FROM sensors
LIFETIME 30 days
```
TinySQL Example 8

- **Network health queries** are metaqueries over the network itself.

  "Report all sensors whose current battery voltage is less than $k$.”

  ```sql
  SELECT nodeid, voltage
  FROM sensors
  WHERE voltage < k
  SAMPLE PERIOD 10 minutes
  ```
TinySQL Example 9

• **Actuation queries** can be used to perform some physical action in response to a query.

"Turn on the fan if the temperature is rising above a certain level."

```sql
SELECT nodeid, temp
FROM sensors
WHERE temp > threshold
OUTPUT ACTION power-on(nodeid)
SAMPLE PERIOD 30 s
```
Outline

• Sensor networks
• TinyOS
• TinyDB
  – Overview
  – Data model
  – Query language
  – Architecture
  – Network administration
  – Aggregates
Inside TinyDB

SELECT AVG(temp) WHERE light > 400

TinyDB

TinyOS

~10,000 Lines Embedded C Code
~5,000 Lines (PC-Side) Java
~3200 Bytes RAM (w/ 768 byte heap)
~58 kB compiled code
(3x larger than 2nd largest TinyOS Program)

Name: temp
Time to sample: 50 uS
Cost to sample: 90 uJ
Calibration Table: 3
Units: Deg. F
Error: ± 5 Deg F
Get f: getTempFunc() ...
Metadata Management

- Each node maintains a **metadata catalog** containing
  - local attributes
    - name
    - cost: power, sample time
  - events
    - name
    - signature
    - cost: frequency estimate
  - user-defined functions and predicates
- Periodically copied to root for use by query optimizer
- Registered via static linking at compile time using nesC
Outline

• Sensor networks
• TinyOS
• **TinyDB**
  – Overview
  – Data model
  – Query language
  – Architecture
  – **Network administration**
  – Aggregates
TinyDB Routing Tree

PC side

TinyDB GUI
TinyDB Client API

JDBC

DBMS

Mote side

TinyDB query processor

Sensor network
Routing

- A routing tree is established
  - Root is a gateway
  - Spanning tree
  - No multiple paths
  - Tree construction and maintenance
- Query fragments are disseminated down the routing tree
  - Query optimization performed centrally, outside the sensor network, using metadata obtained from the nodes
  - Semantic routing tree to avoid flooding
Routing Tree Creation

1. One mote is appointed the root (usually used as the interface/gateway of the network)
2. The root broadcasts a message 
   \(<\text{ID}, \text{distanceFromRoot}>\)
   asking motes to organize into a routing tree
3. Any mote without an assigned level that hears this message assign its level as the \text{received+1}
   and chooses the sender as its parent through which it will route messages to the root
4. Motes re-broadcast the routing message, inserting their own IDs and levels... and so on
Communication Scheduling

- Data processed up the routing tree
- Sample periods: Sleep; data sampling; receiving; processing; transmitting
  - Sleep period defined based on number of children
    - Awakening just in time to receive results
  - Sampling
    - Expensive!
  - Receiving
  - Processing:
    - Filtering, partial aggregate
  - Network transmission
    - Adaptation to network contention and power consumption
Communication Scheduling

• A mote upon receiving a request to perform a query:
  – awakens
  – synchronizes its clock
  – chooses the sender of the msg as its parent
  – forwards the query, setting the delivery interval for children to be slightly before the time its parent expects to see the partial state record
Sample Period

- Long enough to allow all nodes to report
- Delivery interval length = \( \text{SamplePeriod/TreeDepth} \)
  - Sets a lower bound to sample period
  - Limits the maximum sample rate of the network
- The sample rate can be increased by pipelining the communication schedule
Outline

• Sensor networks
• TinyOS
• TinyDB
  – Overview
  – Data model
  – Query language
  – Architecture
  – Network administration
  – Aggregates
Aggregates: Centralized Approach

- Server-based approach: All sensor readings are sent to the base station, which then computes the aggregates

- Example:
  
  \[ \text{SELECT COUNT(*) FROM sensors} \]

  How many transmissions?
Aggregates: Distributed Approach

• In TinyDB aggregates are computed in-network whenever possible.
  – Lower number of transmissions
  – Lower latency
  – Lower power consumption

• Example:
  \[
  \text{SELECT COUNT(*) FROM sensors}
  \]

How may transmissions?
Implementation of $agg$

- Each aggregation $agg$ is implemented via a partial state record and three functions:
  - the partial state record at each node holds an intermediate aggregate value
  - $i$ – initializer: Specifies how to instantiate a partial state record for a single sensor value
  - $f$ – merging function: Specifies how to compute a combined intermediate aggregate from two intermediate aggregates
  - $e$ – evaluator: Takes a partial state record and computes the actual value of the aggregate

- Example: $avg$
  - partial state record $<s,c>$ where $s$ is sum and $c$ count
  - $i(x) = <x,1>$ for a sensor value $x$
  - $f(<s_1,c_1>, <s_2,c_2>) = <s_1+s_2, c_1+c_2>$
  - $e(<s,c>) = s/c$
Taxonomy of Aggregates

We classify aggregates according to four properties:

• Duplicate sensitivity
• Tolerance to losses
• Monotonicity
• State requirements

Yield a general set of optimizations that can be automatically applied
Duplicate Sensitivity

- *duplicate insensitive* aggregates are unaffected by duplicate readings from a single device
  - max, min, count, distinct
- *duplicate sensitive* aggregates will change when a duplicate reading is reported
  - count, sum, average, median,...
Loss-Tolerance

- **exemplary** aggregates return one or more representative values from the set of all values
  - behave unpredictably in the face of loss
  - max, min, median,…
- **summary** aggregates compute some property over all values
  - the aggregate applied to a subset can be treated as a robust approximation of the true aggregate value
  - count, sum, average, count distinct,…
Monotonicity

- **Monotonic** aggregates: For two partial state records $s_1$ and $s_2$,
  \[ \forall s_1, s_2, \ e(f(s_1, s_2)) \geq \max(e(s_1), e(s_2)) \]
  or
  \[ \forall s_1, s_2, \ e(f(s_1, s_2)) \leq \min(e(s_1), e(s_2)) \]

- Important when determining whether some predicates (such as `HAVING`) can be applied in-network, before the final value of the aggregate is known
  - Early predicate evaluation saves messages by reducing the distance that partial state records must flow up the aggregation tree.
State Requirements

- *Distributive* aggregates
- *Algebraic* aggregates
- *Holistic* aggregates
- *Unique* aggregates
- *Content-sensitive* aggregates
State Requirements: Distributive aggregates

• The partial state is the aggregate for the partition of data over which data are computed

• The size of the partial state records is the same as the size of the final aggregate
  – max, min, count, sum
State Requirements: Algebraic aggregates

- The partial state records are not themselves aggregates for the partitions, but are of constant size
  - average
State Requirements: Holistic aggregates

- The partial state records are proportional in size to the set of data in the partition.
- No useful partial aggregation can be done, and all the data must be brought together to be aggregated by the evaluator.
  - median
State Requirements: Unique aggregates

• Similar to holistic aggregates, except that the amount of state that must be propagated is proportional to the number of distinct values in the partition
  – count distinct
State Requirements: Content-sensitive aggregates

- The partial state records are proportional in size to some (perhaps statistical) property of the data values in the partition
## Properties and Optimizations

<table>
<thead>
<tr>
<th>Property</th>
<th>Examples</th>
<th>Affects</th>
</tr>
</thead>
</table>
| Duplicate sensitivity | Min: dup. insensitive  
                      | Avg: dup. sensitive         | Routing redundancy           |
| Loss-tolerance    | Max: exemplary  
                      | Count: summary               | Effect of loss               |
| Monotonicity      | Count: monotonic  
                      | Avg: non-monotonic           | Snooping, hypothesis testing |
| Partial state     | Median: unbounded  
                      | Max: 1 record                | Effectiveness of TAG         |
Beyond TinyDB: LifeUnderYourFeet

- 10 motes in the soil of an urban forest environment
  - MicaZ motes
  - Slanted grid, approx. 2m apart
  - A small stream runs through the middle of the grid; depth depends on recent rain events
  - Collecting air and soil temperature and soil moisture data

- Sampling and data collection
  - **NOT TinyDB!** "Sample-and-collect schemes can loose up to 50% of collected measurements"
  - Sample every minute
  - Store on on-board flash
    - 23 kB/day
    - 512 kB flash means flash is overwritten after 22 days
  - Gather results once a week or fortnight
    - Wireless basestation connected to PC is travelled to the perimeter of the deployment site to collect measures
    - Simple sliding window ARQ protocol
Literature


(Available through the ACM Digital Library; cf. [http://x-port.uio.no](http://x-port.uio.no))