Wireless Sensor Networks
Concepts, Protocols and Applications

Middleware Approaches

Prof. Dr. rer. nat. Peter Langendörfer
fon: 0335 5625 350
fax: 0335 5625 671
e-mail: langendoerfer [ at ] ihp-microelectronics.com
web: www.tu-cottbus.de/systeme
Middleware

**Host 1**

- **Distributed Application**
  - Middleware API
    - Middleware
      - Operating System API
        - Operating System
          - Comm
          - Processing
          - Storage

**Host 2**

- **Distributed Application**
  - Middleware API
    - Middleware
      - Operating System API
        - Operating System
          - Comm
          - Processing
          - Storage

Network
Middleware

- Hides (abstracts) the details of the underlying distributed system:
  - System heterogeneity,
  - Data exchanges,
  - etc.
- Simplifies programming of distributed applications
- Provides value-added services:
  - Naming,
  - Transactions,
  - etc.
- Provides Transparency
  - Location transparency
  - Concurrency transparency
  - Replication transparency
  - Failure transparency
• Sensor Network
  – A distributed sensing network with a large number of devices
    • Constraint resources
  – A great amount of data

• Middleware for Sensor Networks
  – Scalable
  – Self-organizing
  – Energy efficient

• Middleware approaches
  – Data centric
  – Programming abstractions
Data handling

- **Measurement**
  - Input from sensor, e.g., temperature

- **Computation**
  - Result of an internal operation, e.g., average value of several last temperature measurements used to eliminate errors

- **Event**
  - Predefined or dynamically defined exceptional situation
  - described based on available measurements or computations, e.g., temperature (or average) higher than 200 °C

- **Query**
  - Asynchronous access to data not available locally
Data Storage schemes

- External Storage
- Local Storage
- Data-Centric Storage
Data items are named with keys

- DCS supports two operations:
  - $PUT(k, v)$ – stores the value $v$ according to the key $k$ – its name
  - $GET(k)$ – retrieves a value associated with the key $k$

- Hash function
  - Hashes a key $k$ into geographic coordinates
  - $PUT()$ and $GET()$ use the same hash function, i.e., the same key $k$ results in the same location
Data-Centric Storage – example

Put("elephant", data)

(11, 28) = Hash("elephant")
Data-Centric Storage – example

Get(“elephant”) = Hash(“elephant”)
The primary goal of TinyDB is to allow data-driven applications to be developed and deployed much more quickly.

TinyDB frees you from the burden of writing low-level code for sensor devices, including the very tricky sensor network interfaces.

Acquire and deliver desired data while conserving as much power as possible.
• *TinyDB: A declarative Database for Sensor Networks (TinyDB)*
• Relational data model
• SQL like query language
• In network aggregation supported
• Uses Meta data
  – Semantic Routing Tree (SRT)
  – Stored along the routing tree
  – Limits somewhat data to be gathered to nodes which are part of the routing tree
• Central station processes query optimization
• Network wide time synchronization assumed
• Queries submitted in PC
• Parsed, optimized in PC
• Disseminated and processed in network
• Results flow back through the routing tree
Power-aware Optimization

• Cost-based optimizer → lowest overall power consumption

• The cost is dominated by sampling the physical sensors and transmitting query results rather than applying individual operators.

• Focus on ordering joins, selections, and sampling operations.
Power-aware Optimization Metadata Management:

- Each node in TinyDB maintains a catalog of metadata that describes its local attributes, events, and user-defined functions.

- Periodically copied to the root of the network for use by the optimizer.

<table>
<thead>
<tr>
<th>Metadata</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>Cost to sample this attribute (in J)</td>
</tr>
<tr>
<td>Sample Time</td>
<td>Time to sample this attribute (in s)</td>
</tr>
<tr>
<td>Constant?</td>
<td>Is this attribute constant-valued (e.g. id)?</td>
</tr>
<tr>
<td>Rate of Change</td>
<td>How fast the attribute changes (units/s)</td>
</tr>
<tr>
<td>Range</td>
<td>Dynamic range of attribute values (pair of units)</td>
</tr>
</tbody>
</table>

Metadata fields kept with each attribute
• Ordering of Sampling and Predicates:
  – Sampling is often an expensive operation in terms of power.
  – The metadata information is used in query optimization to order the sampling and predicates.

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Power mW</th>
<th>Sample time ms</th>
<th>Sample Energy (VI * t), uJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light, Temp</td>
<td>.9</td>
<td>.1 [5]</td>
<td>90</td>
</tr>
<tr>
<td>Organic Byproducts $^5$</td>
<td>15</td>
<td>&gt; 1000</td>
<td>&gt; $1.5 \times 10^7$</td>
</tr>
</tbody>
</table>

Energy costs of accessing various common sensors
Consider the query below:

```
SELECT accel, mag
FROM sensors
WHERE accel > c1 AND mag > c2
SAMPLE INTERVAL 1s
```

Compare the following options of ordering:

1. sample accelerometer and magnetometer, then apply the selection
2. sample magnetometer, apply selection over its reading first; then sample accelerometer
3. sample accelerometer, apply selection over its reading first; then sample magnetometer
• When each sensor hears a query, it must decide if the query applies locally or needs to be broadcast to its children in the routing tree.

• If a node knows none of its children will ever satisfy the value of some selection predicate, it need not forward the query down the routing tree, which can save the costs of disseminating, executing, and forwarding results for the query.
• Semantic Routing Tree (SRT):
  – allow each node to efficiently determine if any of the nodes
    below it will need to participate in a given query over some
    constant attributes.
  – An SRT is an index over constant attribute that can be used
    to locate nodes that have data relevant to the query.
  – Provide an efficient mechanism for disseminating queries
    and collecting query results for queries over constant
    attributes.
  – Reduce the number of nodes that must disseminate queries
    and forward the continuous stream of results from children
    by nearly an order of magnitude.
How to use SRT:

- When a query $q$ with a predicate over $A$ arrives at node $n$, $n$ checks whether any child’s value of $A$ overlaps the query range of $A$ in $q$:
  - If yes, prepare to receive results and forward the query
  - If no, do not forward $q$

- Is query $q$ applied locally:
  - If yes, execute the query
  - If no, simply ignore it
SRT example

```
QUERY
SELECT light
WHERE x > 3
AND x < 7
```

Location: (4,12)

SRT(x)
1: [1,1]
3: [5,10]

Location: (8,7)

SRT(x)
4: [5,5]
5: [10,10]

Location: (1,7)
SRT(x)

Location: (5,3)
SRT(x)

Location: (10,3)
SRT(x)
tinyDSM

Application Logic

Event & Replication Logic

Memory Manager

Communication Interface

Operating System adaptation layer

Operating System

Protocols

Hardware

Donnerstag, 2. Februar 2012

www.tu-cottbus.de/systeme
• **Application logic** defines the sources of data and actions in case of events.
• **Event & Replication Logic** is responsible for detecting the events for the incoming data. It also takes the decisions on the replication and storage of new data and controls data locating on reading.
• **Query Logic** is responsible for interpreting incoming request messages (queries) and building results into answer messages.
• **Memory Manager** controls the physical data storage on the node. Provides the logical data system in the physical data storage and operations on it.
• **Communication Interface** controls the communication with other nodes. It hides the mapping between different kinds of messages and protocols.
• **Policies** are a virtual module that controls the behaviour of the system. Exchanging the policy file at compile time allows to create several versions of the application that fulfil different requirements using the same source code.
• **OS Adaptation Layer** is a layer that allows running the same base skeleton implementation on different operating systems. It provides the drivers for the services provided by the OS, as well as, means to translate the application requests into tinyDSM native ones.
tinyDSM Policy Parameters, examples

Parameters that control the replication quantities

- **RANGE** = integer – specifies the replication range in the allowed number of hops for the given variable.
  - RANGE = 3 – the variable is replicated on nodes within the 3 hops
- **DENSITY** = byte[] – this array of values between 0 and 100 specifies the density of replication for each hop within the replication range
  - DENSITY = 80,50,10 – causes the number of replicating nodes be 80% within the first hop, 50% within the second and 10% within the third
- **HISTORY** = integer[] – this array specifies the desired amount of historical values stored on each replicating node depending on the hop distance
  - HISTORY = 5,2,1 – the nodes within the first hop that replicate the variable keep five recent values, the nodes within the second and third hop two and one, respectively
- **TIMESTAMP** or **VERSION** – enables versions or timestamp as chronological identification of instances of the variable
tinyDSM Policy Parameters, examples

- parameters that control the quality of the replication, e.g., switching on the replication acknowledgements

- parameters control the security, like local and external access rights to the variable and message authentication or encryption
tinyDSM Variables

- The tinyDSM provides variable-based granularity
- The tinyDSM variables are pieces of data handled by the middleware
  - a distributed variable has a defined type and handling
  - an instance of a variable is identified by the ID in the source node
  - they are like normal local variables, but may be accessed remotely between nodes
  - to reduce the read access time and increase the robustness a variable may be replicated among nodes in the surrounding of the source node.
  - the variables may be of diverse character, they may represent sensed data, as well as, parameters controlling the node’s behaviour

- The definition syntax (simplified)
  
  DISTRIBUTED type VariableName [handling policy params];
• The tinyDSM events are a special kind of variables
  - each event variable is of the Boolean type
  - the state is changed according to a defined logic equation that
    - includes other distributed variables
    - is evaluated each time one of its terms changes
    - triggers an action on positive evaluation
    - is specified at compile time (up to date)
  - the state cannot be directly changed by the application logic

• The definition syntax (simplified)

  EVENT EventName IF condition [actions] [policy params];

  Where actions is:
  TRIGGER functionOn() [RELEASE functionOff()]

• for each function a delay (histeresis) can be defined to avoid too frequent function calls.
// The special keywords in the application code
// The definition of distributed variables
DISTRIBUTED int Temp, Hum;

// The definition of simple events
// The events here monitor the variables only
EVENT tooHot IF Temp > 150;
EVENT tooDry IF Hum < 10;

// The definition of a more complex event
// That monitors the two other events and triggers
// the function alarmOn() on positive evaluation
EVENT itMayBurn IF tooHot && tooDry TRIGGER alarmOn();
tinyDSM Queries

- Each instance of a tinyDSM variable is a tuple consisting of four elements that allow to distinguish any two of them (like a row in a database table)
  - variableID specifying the concrete variable (Temp, Hum, etc.)
  - nodeID identifying the source (owner) node
  - timestamp or version number to order the instances chronologically
  - value

- It is possible to query the system for instances that fulfil a specified query
- It is possible to insert new instances into the system

- The query definition syntax (simplified)
  
  GET varbleList [FOR tupleConstraints][WHERE condition]

- The insert definition syntax (simplified)
  
  SET setList FOR sourceConstraints
tinyDSM Queries, examples

• The compile time definitions of variables and events
  - DISTRIBUTED int Temp, Hum, SamplingFreq;
  - EVENT tooHot IF Temp > 150;
  - EVENT tooDry IF Hum < 10;
  - EVENT itMayBurn IF tooHot && tooDry TRIGGER alarmOn();

• The runtime queries and inserts
  - GET nodeID WHERE tooHot && Hum < 25
  - GET Temp, Hum, nodeID FOR nodeID BETWEEN 3 AND 8
  - SET SamplingFreq = 50 FOR nodeID = 12
  - SET SamplingFreq = 35 FOR nodeID BETWEEN 1 AND 12
• OS specific functionality realisations
  – Timers
  – Input/Output (flash, radio, etc.)
  – Task scheduling

• The tinyDSM middleware core integration
  – The OS adaptation layer allows easy integration

• tinyDSM is implemented in C
  – Uses a specified internal interface for the used OS functions
  – For each OS specific interface a wrapper is needed
  – The OS adaptation layer consists of a complete set of wrappers
tinyDSM: Query language

• SQL like constructs

• NO meta data

• QEP optimization in the WSN

• Provides slightly better energy efficiency than tinyDB and Cougar iff majority of queries is low complexity
tinyDSM running on IHP’s FeuerWhere Node successfully tested
### Comparison of data storage approaches

<table>
<thead>
<tr>
<th>Parameter</th>
<th>tinyPeds</th>
<th>tinyDSM</th>
<th>tinyDB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Storage</strong></td>
<td>In-network</td>
<td>In-network</td>
<td>outside WSN, storage points</td>
</tr>
<tr>
<td><strong>Data types</strong></td>
<td>aggregated</td>
<td>Raw, events; aggregated,</td>
<td>Raw, aggregated, events, alarms</td>
</tr>
<tr>
<td><strong>Type of storage</strong></td>
<td>DB like</td>
<td>Main memory like</td>
<td>DB like</td>
</tr>
<tr>
<td><strong># of replicas</strong></td>
<td>Small (&gt;=2)</td>
<td>Medium (&gt;&gt;10 depending on network density)</td>
<td>none</td>
</tr>
<tr>
<td><strong>Network structure</strong></td>
<td>Hierarchical</td>
<td>Flat</td>
<td>Flat</td>
</tr>
<tr>
<td><strong>Information retrieval</strong></td>
<td>Pull</td>
<td>Pull; compile time publish/subscribe</td>
<td>Pull; event triggered</td>
</tr>
<tr>
<td><strong>Query language</strong></td>
<td>SQL like</td>
<td>SQL like</td>
<td>SQL like</td>
</tr>
</tbody>
</table>
## Comparison of data storage approaches

<table>
<thead>
<tr>
<th>Parameter</th>
<th>tinyPeds</th>
<th>tinyDSM</th>
<th>tinyDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query dissemination</td>
<td>Controlled flooding</td>
<td>Controlled flooding / geographic</td>
<td>SRT overlay</td>
</tr>
<tr>
<td>Support for mobile readers</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Robustness</td>
<td>To local disaster (replication of aggregates on a fixed neighbour cluster head)</td>
<td>To local disaster (replication in a close area)</td>
<td>-</td>
</tr>
<tr>
<td>Load balancing</td>
<td>Periodic cluster head election</td>
<td>Replication in a close area</td>
<td>No</td>
</tr>
<tr>
<td>Security</td>
<td>Confidentiality, Integrity and access control</td>
<td>Confidentiality Integrity and access control</td>
<td>No</td>
</tr>
</tbody>
</table>
Geschafft!!
Uff...
Thanks for your attention!