Motivation

- Programming sensor nets is difficult
- Worst of distributed and embedded worlds
- Dynamic, unreliable networks
  - Links come and go
  - Nodes come and go
  - Mobility
- Constrained resources
  - Simple OS
  - System-centric programming
  - Many competing optimization goals
  - Limited visibility

ON World Study 7/2005
50% of interviewed OEM’s and platform providers believe that limited ease of use inhibits adoption of WSN
**Simplifying Programming**

- Identify common subtasks
  - Self configuration
  - Distributed event patterns

- Programming abstractions for subtasks
  - Domain-specific programming language
  - Compiler
  - Node runtime

- Complete application
  - Code for one or more programming abstractions
  - Application-specific code
Desirable Languages

- **Declarative**
  - Specify desired application behavior
  - Not: how to achieve this behavior

- **Node ensembles**
  - Specify behavior of a group of nodes or whole network
  - Not: individual nodes

1. Send request containing …
2. Wait 10 seconds
3. Perhaps retransmit
4. Count distinct replies

„# neighbors with a temperature sensor“
Benefits and Limitations

- High-level programming
  - Raise level of abstraction from system-centric to application-centric
  - Hide complexity

- Efficiency
  - Very domain-specific
  - Compilation
  - Use hints from network (e.g., avg node density) to optimize
Example I: Generic Role Assignment
Self-Configuration as Role Assignment

- Self-configuration
  - Initially, all nodes are (more or less) equal
  - Nodes take on specific functions

- Examples
  - Clustering: HEAD, SLAVE, GATEWAY
  - Coverage: ON, OFF
  - Aggregation: SOURCE, AGGREGATOR
Generic Role Assignment

- Supports automatic assignment of roles to sensor nodes
  - Maintain valid assignment as network changes
- Declarative role specifications
  - Definition of roles
  - Definition of rules (constraints) for assignment
  - Rules refer to node properties
Architecture

Role Specifications

Sink (Compiler)

Sensor Node

RA Algorithm

Property Directory

App.

battery = 80%
pos = (12.3, 3.4)
role = ON
...
Coverage [cf. PEAS]

\[
\text{ON} :: \{ \\
\quad \text{battery} >= \text{threshold} \&\& \\
\quad \text{count(1 hop)} \{ \\
\quad\quad \text{role} == \text{ON} \&\& \\
\quad\quad \text{dist(pos, super.pos) < R} \\
\quad \} == 0 \\
\}\ \\
\text{OFF} :: \text{else}
\]

- \text{count(scope) \{ pred \}}
  - Counts nodes matching \text{pred} within \text{scope}
  - \text{super.x} equals property \text{x} of referring node
Clustering [cf. Passive Clustering]

**CLUSTERHEAD** :: {
    count(1 hop) {
        role == CLUSTERHEAD
    } == 0 }

**GATEWAY** :: {
    cheads == retrieve(1 hop, 2) {
        role == CLUSTERHEAD
    } &&
    count(2 hops) {
        role == GATEWAY &&
        cheads == super.cheads
    } == 0 }

**SLAVE** :: else

- **retrieve(scope, num) { pred } == cheads**
  - At least *num* nodes in *scope* must fulfil *pred*
  - Bind the 2 nodes to *cheads*
Distributed Algorithm

- Property propagation
  - Derive scope
  - Scoped broadcast
- Rule evaluation
  - Evaluate all rules locally
  - Assign first matching role
  - Re-propagate changed properties
- Scheduling
  - Random delays to break synchronization
- Notification
  - Notify application of „stable roles“
- Distributed fix-point iteration
  - Convergence...

```plaintext
ON :: { count(1 hop) { role == ON } == 0 } 
OFF :: else
```
Role Initialization

- **Base algorithm**
  - All nodes start with role UNDEFINED

- **Probabilistic role initialization**
  - „Guess“ initial roles for each node
  - Repair wrong guesses with base algorithm
  - Hope: faster convergence

- **Two variants**
  - Use only static information
  - Use runtime information
Static Initialization

- **Basic approach**
  - Given: role specification, network density
  - Compute: $P[r] = P[\text{node assumes role } r]$
  - Role init.: according to probabilities

- **Translate spec. to equation system**
  - $P[\text{ON}] = P[\text{no neighbors are ON}]$
    - $= (1 - P[\text{ON}])^N$
  - $P[\text{OFF}] = 1 - P[\text{ON}]$
  - Solve for $P[\text{ON}]$, $P[\text{OFF}]$

```plaintext
ON :: { count(1 hop) { role == ON } == 0 }
OFF :: else
```
Dynamic Initialization

- Static doesn’t work well in many situations
  - Property values hard to predict offline
  - Retrieve introduces dependence on node identity

- Combine specification flood and initialization
  - Adjust probabilities
  - Use actual roles where known

\[
\text{ON} :: \{ \begin{align*}
\text{count}(1 \text{ hop}) \{ \\
&\text{role} == \text{ON} \\
\} == 0 \\
\text{OFF} :: \text{else}
\end{align*}
\]
Dynamic Initialization

- Adjust probabilities
  - Similar equation system as for static
  - Replace probabilities with known info
  - \( P'[\text{ON}] = (1 - P[\text{ON}])^U \prod_k (1 - \text{ON}(k)) \)
    
    unknown \hspace{2cm} known

  - Where \( \text{ON}(k) = 1 \) iff node \( k \) is ON

- Example
  - If one of the known neighbors is ON, then \( P'[\text{ON}] = 0 \)
Implementation

- Comprehensive simulation tool
  - Discrete event model
  - Visualization / specification frontend
  - Specification compiler
- Prototype implementation on BTnodes
Message Overhead

- How many messages does a node send?

![Graph showing the average number of messages sent vs. number of nodes, with lines for Base, Static, and Dynamic caching methods.](image)
Correctness

- Percentage of nodes with incorrect roles

![Graph showing the percentage of nodes with incorrect roles over the number of nodes, with lines for Base, Static, and Dynamic schemes.](image)
Convergence

- How many role changes until stable configuration?

![Graph showing the number of role changes per node against the number of nodes, comparing Base, Static, and Dynamic configurations. The graph indicates that Dynamic has the least number of role changes, followed by Static, and then Base.](image-url)
Example II: Frequent Event Patterns
Distributed Event Patterns

- Correlations among events observed by different sensor nodes
- Example: Great Duck ++
Event Pattern Abstraction

- Discovery of frequent event patterns
  - Distributed in the sensor network
  - Transmit compact patterns rather than raw data streams

- Declarative „query“ language
  - Events of interest
  - Constraints on sought patterns
  - Compiled and executed on sensor nodes
Event Patterns

- Correlation among events on a node and recent events in its neighborhood

- Example
  - IF bird leaves
  - THEN at least 5 trample events have been observed during last 10 min no more then 20 m away
  - IN 80% of all cases

\[
(\text{trample}, <20\text{m}, <10\text{min}, \geq 5) \Rightarrow \text{leave [80\%]}
\]

\[(E_1, D_1, T_1, N_1) \land (E_2, D_2, T_2, N_2) \ldots \Rightarrow E [S]\]
Declarative Query

- Epoch length
- Event definitions
- Causal / consequential events
- Minimum support
- Scope
  - Neighborhood size
  - History length
- Resolution
  - Distance
  - Time
  - Frequency
**Mining Algorithm 1**

- Event collection and distance quantization

```
SELECT SUM(T), <distance>
from sensors WHERE T == 1
GROUP BY <distance>

SAMPLE PERIOD 30 sec
SCOPE 1 hop
```

**Epoch** | **T.near** | **T.far** | **L**
---|---|---|---
1 | 2 | 0 | 1
2 | 0 | 0 | 0
3 | 1 | 1 | 1
4 | 0 | 0 | 0
5 | 3 | 1 | 1
6 | 1 | 0 | 1

ePOCH = 30 sec
levents = {T}
revents = {L}
neighborhood = 1 hop
history = 6 epochs
distance = {
    near=(0,10], far=(10,20]
}
## Mining Algorithm 2

### Time quantization

<table>
<thead>
<tr>
<th>Epoch</th>
<th>T.near</th>
<th>T.far</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Epoch</th>
<th>T.near</th>
<th>T.far</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>old</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>recent</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>now</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\text{time} = \{
\text{now}=0, \\
\text{recent}=[1,3], \\
\text{old}=[4,6]
\}\]
Mining Algorithm 3

- Event-frequency quantization

<table>
<thead>
<tr>
<th>Epoch</th>
<th>T.near</th>
<th>T.far</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>old</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>recent</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>now</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Epoch</th>
<th>T.near</th>
<th>T.far</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>old</td>
<td>some</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>recent</td>
<td>some</td>
<td>some</td>
<td>2</td>
</tr>
<tr>
<td>now</td>
<td>some</td>
<td>none</td>
<td>1</td>
</tr>
</tbody>
</table>

frequency = {
  none=0,
  some=[1,\infty]
}
## Mining Algorithm 4

- **Set creation**

<table>
<thead>
<tr>
<th>Epoch</th>
<th>T.near</th>
<th>T.far</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>old</td>
<td>some</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>recent</td>
<td>some</td>
<td>some</td>
<td>2</td>
</tr>
<tr>
<td>now</td>
<td>some</td>
<td>none</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>T.near.old.some</td>
</tr>
<tr>
<td>T.near.recent.some</td>
</tr>
<tr>
<td>T.near.now.some</td>
</tr>
<tr>
<td>T.far.old.none</td>
</tr>
<tr>
<td>T.far.recent.some</td>
</tr>
<tr>
<td>T.far.now.none</td>
</tr>
<tr>
<td>L</td>
</tr>
</tbody>
</table>
Mining Algorithm 5

- Repeat steps 1-4 every epoch
  - One set for each epoch
  - Stream of sets
- Find frequent sets in stream
  - Set is frequent if subset of at least minsupport percent of the stream elements
- Example
  - minsupport = 50%
  - Frequent sets: \{A\}, \{B\}, \{A,B\}
- Std. data mining task
  - Challenge: constrained resources!
  - Implementation on BTnodes with 64k RAM
  - Ask me for details!
Mining Algorithm 6

- Transform frequent sets to event patterns

<table>
<thead>
<tr>
<th>Frequent Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>T.near.old.some</td>
</tr>
<tr>
<td>T.far.recent.none</td>
</tr>
<tr>
<td>L</td>
</tr>
</tbody>
</table>

\[(T, \text{near, old, some}) \land (T, \text{far, recent, none}) \Rightarrow L\]
Mining Algorithm 7

- Typically, only some nodes execute mining algorithm
  - Use role assignment
- Report patterns to sink
- Pattern aggregation
  - Cluster nodes with similar patterns
Putting it all together

- Interplay of programming abstractions
- Application may use multiple abstractions
- One abstraction may use other abstractions
  - Mining uses Role Assignment and TinyDB
- A toolbox of programming abstractions
  - Openness
  - Composition
  - Interference at the networking level
  - …
Summary

- Simplify application development
- Introduce programming abstractions
  - Language, compiler, runtime for specific application tasks
  - Declarative, node ensembles
- Two examples
  - Role assignment for self-configuration
  - Frequent patterns for event correlation
- Toolbox [future work]
  - Composable declarative programming abstractions
Thanks!

roemer@inf.ethz.ch