Sensor Network Protocol Design and Implementation

Philip Levis
UC Berkeley
Sensor Network Constraints

- Distributed, wireless networks with limited resources
  - Energy, energy, energy.

- Communication is expensive.
  - Idle listening is the principal energy cost.
  - Radio hardware transition times can be important.
  - Low transmission rates can lower cost of idle listening.

- Nodes cannot maintain a lot of state.
  - RAM is at a premium.
Constraints, continued

• Uncontrolled environments that drive execution

• Variation over time and space
  - The uncommon is common
  - Unforeseen corner cases and aberrations
Sensor Network Behavior
Design Considerations

- Uncontrolled environment: simplicity is critical.
  - The world will find your edge conditions for you.
  - Simplicity and fault tolerance can be more important than raw performance.

- Wireless channel: cheap broadcast primitive.
  - Protocols can take advantage of spatial redundancy.

- Redundancy requires idempotency
  - But we have limited state.
A Spectrum of Protocol Classes

- Dissemination: One to N
- Collection Routing: N to One
- Landmark Routing: N to changing one
- Aggregation Routing: N to One
- Any-to-Any
A Spectrum of Protocol Classes

- Dissemination: One to N
- Aggregation Routing: N to One
A Spectrum of Protocol Classes

- Dissemination: One to N
- Aggregation Routing: N to One
Dissemination

- Fundamental networking protocol
  - Reconfiguration
  - Reprogramming
  - Management

- Dissemination: reliably deliver a datum to every node in a network.
To Every Node in a Network

- Network membership is not static
  - Loss
  - Transient disconnection
  - Repopulation

- Limited resources prevent storing complete network population information

- To ensure dissemination to every node, we must periodically maintain that every node has the data.
The Real Cost

- Propagation is costly
  - Virtual programs (Maté, TinyDB): 20-400 bytes
  - Parameters, predicates: 8-20 bytes
  - To every node in a large, multihop network...

- But maintenance is more so
  - For example, one maintenance transmission every minute
  - Maintenance for 15 minutes costs more than 400B of data
  - For 8-20B of data, two minutes are more costly!

- Maintaining that everyone has the data costs more than propagating the data itself.
Three Needed Properties

• **Low maintenance overhead**
  - Minimize communication when everyone is up to date

• **Rapid propagation**
  - When new data appears, it should propagate quickly

• **Scalability**
  - Protocol must operate in a wide range of densities
  - Cannot require *a priori* density information
Existing Algorithms Are Insufficient

- Epidemic algorithms
  - End to end, single destination communication, IP overlays

- Probabilistic broadcasts
  - Discrete effort (terminate): does not handle disconnection

- Scalable Reliable Multicast
  - Multicast over a wired network, latency-based suppression

- SPIN (Heinzelman et al.)
  - Propagation protocol, does not address maintenance cost
Solution: Trickle
Solution: Trickle

- “Every once in a while, broadcast what data you have, unless you’ve heard some other nodes broadcast the same thing recently.”
Solution: Trickle

- “Every once in a while, broadcast what data you have, unless you’ve heard some other nodes broadcast the same thing recently.”

- Behavior (simulation and deployment):
  - Maintenance: a few sends per hour
  - Propagation: less than a minute
  - Scalability: thousand-fold density changes
Solution: Trickle

• “Every once in a while, broadcast what data you have, unless you’ve heard some other nodes broadcast the same thing recently.”

• Behavior (simulation and deployment):
  - Maintenance: a few sends per hour
  - Propagation: less than a minute
  - Scalability: thousand-fold density changes

• Instead of flooding a network, establish a trickle of packets, just enough to stay up to date.
Trickle Assumptions

- Broadcast medium
- Concise, comparable metadata
  - Given A and B, know if one needs an update
- Metadata exchange (maintenance) is the significant cost
Detecting That a Node Needs an Update

- As long as each node *communicates* with others, inconsistencies will be found.
- Either reception or transmission is sufficient.
- Define a desired detection latency, $\tau$.

Choose a redundancy constant $k$
- $k = (\text{receptions} + \text{transmissions})$
- In an interval of length $\tau$

- Trickle keeps the rate as close to $k/\tau$ as possible.
Trickle Algorithm

- Time interval of length $\tau$
- Redundancy constant $k$ (e.g., 1, 2)
- Maintain a counter $c$
- Pick a time $t$ from $[0, \tau]$
- At time $t$, transmit metadata if $c < k$
- Increment $c$ when you hear identical metadata to your own
- Transmit updates when you hear older metadata
- At end of $\tau$, pick a new $t$
Example Trickle Execution

\[ k=1 \]

transmission  suppressed transmission  reception

c

A

B

C

time

\( \tau \)
Example Trickle Execution

At the time $t_{A1}$, the transmission is suppressed and reception is not taking place.

$\tau$ represents the time duration.
Example Trickle Execution

$A_1$ transmission suppressed transmission reception

A \rightarrow B \rightarrow C

$C_0 \rightarrow 0 \rightarrow 1 = k = 1$

$time$

$\tau$

CS 268, Spring 2005
Example Trickle Execution

- Transmission
- Suppressed transmission
- Reception

\[ k = 1 \]

- \( t_{A1} \)
- \( t_{C1} \)
- \( \tau \)

CS 268, Spring 2005
Example Trickle Execution

\[ t_{A1} \]
\[ t_{C1} \]
\[ \tau \]

\( k=1 \)

transmission \quad suppressed \ transmission \quad reception
Example Trickle Execution

\[ \tau \]

\[ \text{transmission} \quad \text{suppressed transmission} \quad \text{reception} \]

CS 268, Spring 2005
Example Trickle Execution

\[ k=1 \]

\[ t_{A1}, t_{B1}, t_{C1} \]

Transmission suppressed transmission reception

CS 268, Spring 2005
Example Trickle Execution

\[ \begin{align*}
&\text{time} \\
&\tau \\
&k=1 \\
&A_{1} & t_{A1} & B_{1} & t_{B1} & B_{2} & t_{B2} \\
&C_{1} & t_{C1} \\
&\text{transmission} \quad \text{suppressed transmission} \quad \text{reception}
\end{align*} \]
Example Trickle Execution

- $k=1$
- $t_{A1}$
- $t_{B1}$
- $t_{B2}$
- $t_{C1}$
- $t_{C2}$

Time line with transmission, suppressed transmission, and reception marks.
Example Trickle Execution

\( k=1 \)

\[ \tau \]

\( t_{A1} \rightarrow t_{A2} \)
\( t_{B1} \rightarrow t_{B2} \)
\( t_{C1} \rightarrow t_{C2} \)

transmission
suppressed transmission
reception

c1

CS 268, Spring 2005
Experimental Methodology

- High-level, algorithmic simulator
  - Single-hop network with a uniform loss rate

- TOSSIM simulates TinyOS implementations
  - Multi-hop networks with empirically derived loss rates

- Real world deployment in an indoor setting

- In experiments (unless said otherwise), $k = 1$
Maintenance Evaluation

• Start with idealized assumptions, relax each
  - Lossless cell
  - Perfect interval synchronization
  - Single hop network

• Ideal: Lossless, synchronized single hop network
  - $k$ transmissions per interval
  - First $k$ nodes to transmit suppress all others
  - Communication rate is independent of density

• First step: introducing loss
Logarithmic Behavior of Loss

- Transmission increase is due to the probability that one node has not heard $n$ transmissions

- Example: 10% loss
  - 1 in 10 nodes will not hear one transmission
  - 1 in 100 nodes will not hear two transmissions
  - 1 in 1000 nodes will not hear three, etc.

- Fundamental bound to maintaining a per-node communication rate
Synchronization
(algorithmic simulator)
Short Listen Effect

- Lack of synchronization leads to the "short listen effect"

- For example, B transmits three times:

\[ \tau \]

Time

A
B
C
D

CS 268, Spring 2005
Short Listen Effect Prevention

- Add a listening period: $t$ from $[0.5\tau, \tau]$
Effect of Listen Period
(algorithmic simulator)

![Graph showing the effect of listen period on transmissions per interval for different numbers of motes, with lines indicating not synchronized, synchronized, and listening states.](image_url)
Multihop Network (TOSSIM)

- Redundancy: \( \frac{(\text{transmissions} + \text{receptions})}{\text{intervals}} - k \)
- Nodes uniformly distributed in 50’x50’ area
- Logarithmic scaling holds

![Redundancy over Density in TOSSIM](image)
Empirical Validation (TOSSIM and deployment)

- 1-64 motes on a table, low transmit power
Maintenance Overview

- Trickle maintains a per-node communication rate
- Scales logarithmically with density, to meet the per-node rate for the worst case node
- Communication rate is really a number of transmissions over space
Interval Size Tradeoff

- Large interval $\tau$
  - Lower transmission rate (lower maintenance cost)
  - Higher latency to discovery (slower propagation)

- Small interval $\tau$
  - Higher transmission rate (higher maintenance cost)
  - Lower latency to discovery (faster propagation)

- Examples ($k=1$)
  - At $\tau = 10$ seconds: 6 transmits/min, discovery of 5 sec/hop
  - At $\tau = 1$ hour: 1 transmit/hour, discovery of 30 min/hop
Speeding Propagation

- Adjust $\tau$: $\tau_l, \tau_h$
- When $\tau$ expires, double $\tau$ up to $\tau_h$
- When you hear newer metadata, set $\tau$ to $\tau_l$
- When you hear newer data, set $\tau$ to $\tau_l$
- When you hear older metadata, send data
Simulated Propagation

- New data (20 bytes) at lower left corner
- 16 hop network
- Time to reception in seconds
- Set $\tau_1 = 1$ sec
- Set $\tau_h = 1$ min
- 20s for 16 hops
- Wave of activity
Empirical Propagation

- Deployed 19 nodes in office setting
- Instrumented nodes for accurate installation times
- 40 test runs
Empirical Propagation

- Deployed 19 nodes in office setting
- Instrumented nodes for accurate installation times
- 40 test runs
Network Layout
(about 4 hops)
Empirical Results

$k=1, \tau_l=1 \text{ second}, \tau_h=20 \text{ minutes}$

Mote Distribution, $\tau_h=20\text{m}, k=1$
Network Layout
(about 4 hops)
Empirical Results

$k=1, \tau_l=1 \text{ second}, \tau_h=20 \text{ minutes}$

- Sparse networks can have a few stragglers.
Dissemination

- Trickle scales logarithmically with density
- Can obtain rapid propagation with low maintenance
  - In example deployment, maintenance of a few sends/hour, propagation of 30 seconds
- Controls a transmission rate over space
  - Coupling between network and the physical world
- Trickle is a nameless protocol
  - Uses wireless connectivity as an implicit naming scheme
  - No name management, neighbor lists...
  - Stateless operation (well, eleven bytes)
A Spectrum of Protocol Classes

- Dissemination: One to N

- Aggregation Routing: N to One
Aggregation Routing

- Collect data *aggregates* from a network
  - “How many nodes are there?”
  - “What is the mean temperature?”
  - “What is the median temperature?”

- Deliver aggregate to a central point (root)

- Two examples: TinyDB and Synopsis Diffusion
TinyDB: Tiny Aggregation (TAG)

- Proposed by Sam Madden et al.
- Compute aggregates over a collection tree
Query Propagation Via Tree-Based Routing

- Tree-based routing
  - Used in:
    - Query delivery
    - Data collection
  - Topology selection is important; e.g.
    - LEACH/SPIN. Heinzelman et al MOBICOM 99
    - SIGMOD 2003
  - Continuous process
    - Mitigates failures
Basic Aggregation

- In each epoch:
  - Each node samples local sensors once
  - Generates **partial state record (PSR)**
    - local readings
    - readings from children
  - Outputs PSR during assigned **comm. interval**

- At end of epoch, PSR for whole network output at root

- New result on each successive epoch

- Extras:
  - Predicate-based partitioning via GROUP BY
SELECT COUNT(*)
FROM sensors

Illustration: Aggregation

Sensor #

Interval #

Interval 4

Epoch

CS 268, Spring 2005
Illustration: Aggregation

SELECT COUNT(*)
FROM sensors

Sensor #

Interval #

Interval 3

CS 268, Spring 2005
Illustration: Aggregation

```
SELECT COUNT(*)
FROM sensors
```

Sensor #

Interval #

CS 268, Spring 2005
Illustration: Aggregation

```
SELECT COUNT(*)
FROM sensors
```

![Diagram showing sensor aggregation and interval 1 with sensor # and interval #]
Illustration: Aggregation

SELECT COUNT(*)
FROM sensors

Sensor #

Interval #

CS 268, Spring 2005
Interval Assignment: An Approach

4 intervals / epoch

Interval # = Level

Level = 1

- CSMA for collision avoidance
- Time intervals for power conservation
- Many variations
- Time Sync

SELECT COUNT(*)...

Pipelining: Increase throughput by delaying result arrival until a later epoch

CS 268, Spring 2005
Aggregation Framework

- Support any aggregation function conforming to:

  \[ \text{Agg}_n = \{ f_{\text{init}}, f_{\text{merge}}, f_{\text{evaluate}} \} \]

  \[ F_{\text{init}} \{a_0\} \rightarrow <a_0> \]

  \[ F_{\text{merge}} \{<a_1>,<a_2>\} \rightarrow <a_{12}> \]

  \[ F_{\text{evaluate}} \{<a_1>\} \rightarrow \text{aggregate value} \]

Example: Average

  \[ \text{AVG}_{\text{init}} \{v\} \rightarrow <v,1> \]

  \[ \text{AVG}_{\text{merge}} \{<S_1, C_1>, <S_2, C_2>\} \rightarrow <S_1 + S_2, C_1 + C_2> \]

  \[ \text{AVG}_{\text{evaluate}} \{<S, C>\} \rightarrow S/C \]
Types of Aggregates

- SQL supports MIN, MAX, SUM, COUNT, AVERAGE
- Any function over a set can be computed via TAG
- In network benefit for many operations
  - E.g. Standard deviation, top/bottom N, spatial union/intersection, histograms, etc.
  - Compactness of PSR
TAG/TinyDB Observations

- Complex: requires a collection tree as well as pretty good time synchronization
- Fragile: single lost result can greatly perturb result
- In practice, really hard to get working.
  - Sonoma data yield < 50%
  - Intel TASK project (based on TinyDB) has had many deployment troubles/setbacks (GDI 2004)
Another Take on Aggregation

• “Sketches” proposed by two groups
  - BU: Jeffrey Considine, Feifei Li, George Kollios and John Byers (ICDE 2004)

• Based on work probabilistic counting algorithm work by Flajolet and Martin (1985)
Basic Observation

- Fragility comes from duplicate and order sensitivity
  - A PSR included twice will perturb the result
  - Computational model bound to communication model

- Decoupling routing from computation is desirable
  - Compute aggregates over arbitrary network layers, can take advantage of multipath propagation, etc.
  - Decoupling requires aggregates to be resistant to a single result being included multiple times, ordering
Synopsis Diffusion

- Order and duplicate insensitive (ODI) aggregates
- Every node generates a “sketch” of its value
- Aggregation combines sketches in an ODI way
  - E.g., take a boolean OR
- Compute aggregate result from combined sketch
More Specifically

- Three functions:
  - Generate initial sketch (produce a bit field)
  - Merge sketches (ODI)
  - Compute aggregate from complete sketch
Example

- Three functions:
  - Generate initial sketch (produce a bit field)
  - Merge sketches (ODI)
  - Compute aggregate from complete sketch
Example

- Three functions:
  - Generate initial sketch (produce a bit field)
  - Merge sketches (ODI)
  - Compute aggregate from complete sketch
Example

• Three functions:
  - Generate initial sketch (produce a bit field)
  - Merge sketches (ODI)
  - Compute aggregate from complete sketch
Example

- Three functions:
  - Generate initial sketch (produce a bit field)
  - Merge sketches (ODI)
  - Compute aggregate from complete sketch
Example

- Three functions:
  - Generate initial sketch (produce a bit field)
  - Merge sketches (ODI)
  - Compute aggregate from complete sketch
Example ODI: Count

- How many nodes are there in the network?

- Generating initial sketch
  - Generate a random number
  - Sketch is the least significant 0 bit in the number
  - E.g.: 0110101011101011 → 0000000000000100

- Merging sketches: binary OR of two sketches

- Resolving aggregate:
  - Final value is the least significant 0 in the sketch/1.546
  - 0000000101111111 → 0000000010000000/1.546
Count Example

- Three functions:
  - Generate initial sketch
  - Merge sketches (ODI)
  - Compute aggregate from complete sketch
• Three functions:
  - Generate initial sketch
  - Merge sketches (ODI)
  - Compute aggregate from complete sketch
Count Example

- Three functions:
  - Generate initial sketch
  - Merge sketches (ODI)
  - Compute aggregate from complete sketch
Count Example

- Three functions:
  - Generate initial sketch
  - Merge sketches (ODI)
  - Compute aggregate from complete sketch
Count Example

- Three functions:
  - Generate initial sketch
  - Merge sketches (ODI)
  - Compute aggregate from complete sketch
Count Example

- Three functions:
  - Generate initial sketch
  - Merge sketches (ODI)
  - Compute aggregate from complete sketch
  - \( \frac{8}{1.556} = 5.14 \)
ODI Issues

- Sketches are robust, but are inaccurate estimates.
  - Standard deviation of error is 1.13 bits

- Many sketches are needed for an accurate result
  - Compute many (e.g., 12) in parallel
  - Generate a stream of sketches
A Stream of Sketches

- ODI rings
- Only merge in lower rings
A Stream of Sketches

- ODI “rings”
- Only merge in lower rings
- Example: hop count
Arbitrary Network Layers

• ODI aggregates can be computed with Trickle
  - Two trickles: ODI sequence number, ODI sketches
  - When you hear a new sequence number, generate a new local sketch, reset both trickles
  - Delivers aggregate to every node in the network

• ODI aggregates require a gradient
  - Gradient determines what values can be merged
  - Prevents a sketch persisting forever
  - Gradient through time (seq no.), or through space (hops)
TAG vs. ODI

• TAG: computes exact value, bound to a specific routing layer that is vulnerable to loss and requires complex synchronization
  - If it works right once, you get the precise answer.
  - Really hard to get to work right.

• ODI: computes estimate, decoupled from network layer, multipath makes it more resistant to loss, requires simple synchronization
  - Simple implementations can accurately compute estimate, many estimates needed for a precise answer.
Implementation Experience

• TAG: implemented in TinyDB system
  - Two months of work to get TinyDB to work in deployment.
  - Very low data yield, no-one has been able to get it to work again (TASK project).

• ODI: a count query is 30 lines of code
  - A few tricks: coarse time synchronization needed.
  - Hasn’t been tested to the same degree as TAG.
Design Considerations

- **Uncontrolled environment:** simplicity is critical.
  - The world will find your edge conditions for you.
  - Simplicity and fault tolerance can be more important than raw performance.

- **Wireless channel:** cheap broadcast primitive.
  - Protocols can take advantage of spatial redundancy.

- **Redundancy requires idempotency**
  - But we have limited state.
Questions