TAG: A TINY AGGREGATION SERVICE FOR AD-HOC SENSOR NETWORKS

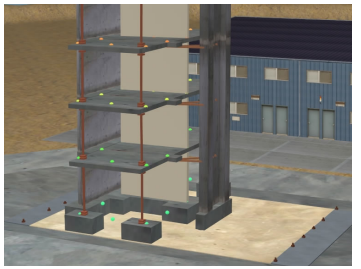
SAMUEL MADDEN, MICHAEL J. FRANKLIN, JOSEPH HELLERSTEIN, AND WEI HONG

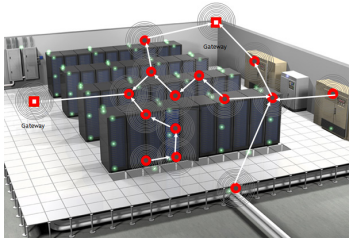
Proceedings of the Fifth Symposium on Operating Systems Design and implementation (OSDI '02), December 9 - 11, 2002, Boston, MA, USA.

Presented by Akash Kapoor

Motivation

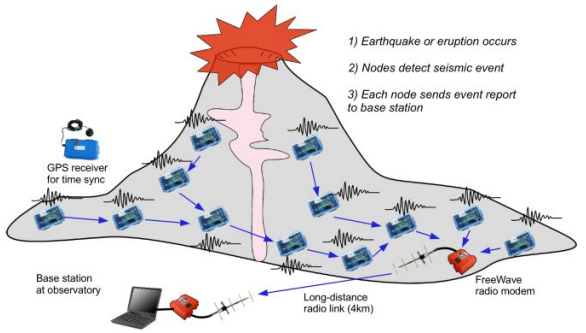
- Used to monitor and collect data about various phenomenon.
  - Wild-life, volcanoes, data centers, CPU temperatures, buildings.
- Applications require summary/aggregations rather raw sensor data.
- Users may not be well versed with the low-level optimizations.
- Important to minimize the power consumption.





How does Tiny AGgregation service help?:

- High level language similar to SQL for querying.
- Distributes the query in the network.
- Queries executed to reduce communication and power.
- Enables in-network aggregation of the results.
- Order of magnitude reduction in communication compared to centralized approach.



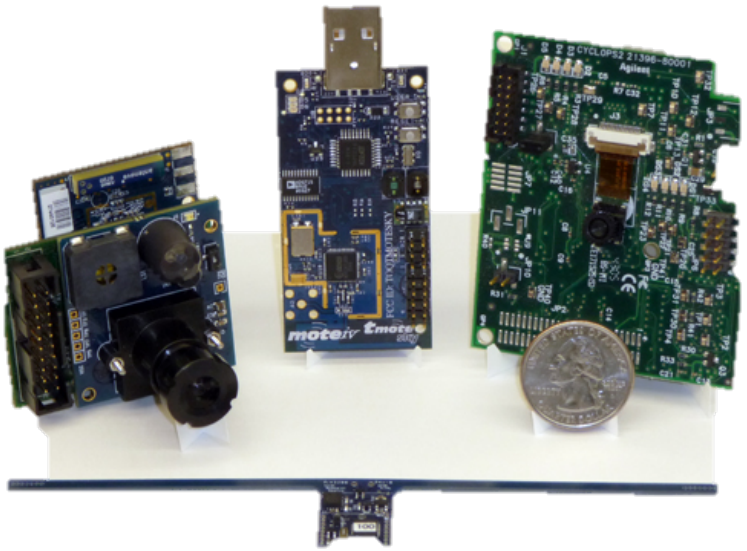
Background

Smart Sensors:

- Wireless, Battery powered.
- Really small around 2cm x 4cm x 1cm.
- Motes by UC Berkeley.
- Single-channel half-duplex radio.
- Unreliable message delivery.

Battery capabilities on motes:

- Small batteries: AA battery packs or coin cells.
- Radio communication dominates the battery consumptions on the motes.
  - Power consumption: Transmission of single bit ≈ 800 instructions.
- Calls for power conserving algorithms.



Background

Ad-Hoc Networks:

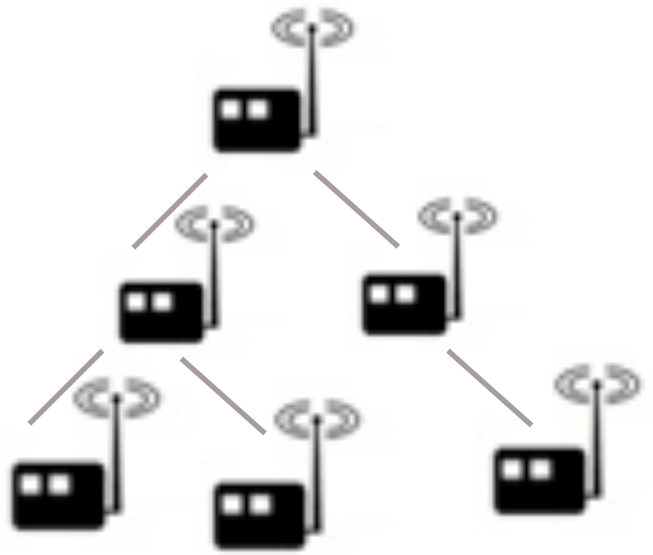
- Dynamic in nature as topology can change.
- Identify and route data between devices without prior knowledge.

Tree based routing in Ad-Hoc networks:

- Motes arrange themselves as a tree
- Node interfacing with the user act as the root.
- Root sends this message periodically to adapt to new network changes.

TAG requirements:

- Root should be able to propagate a message in the whole network.
- A route from a node to the root.
- At-most once semantics for message delivery.



Query Model

- SQL-style query syntax.
  - Over a single table, *sensors*.
  - It's schema is known to the base station.
  - Stream of values.
  - Each mote has a small catalog of attributes.
  - Each sensor is a row in the *sensors* table.
- Example:
  - Monitor occupancy of conference rooms of a floor.
  - Use the microphones.

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

Query Model

- *WHERE* clause:
  - Filters out individual sensor readings, locally at mote, before aggregation.
- *GROUP BY* clause:
  - Attribute based partitioning of sensor readings.
- *HAVING* clause:
  - Suppress groups that do not satisfy the predicates.
- *EPOCH DURATION:*
  - Specifies how often the updates must be delivered.
- Records:
  - Consists of one <*group id*, *aggregate value*> pair per group.
  - Readings used to compute an aggregate record all belong to the same time interval, or epoch.

```
SELECT {agg(expr), attrs}
FROM sensors
WHERE {selPreds}
GROUP BY {attr}
HAVING {havingPreds}
EPOCH DURATION i
```

Aggregates

Aggregates supported by TAG:

- SQL supports: MIN, MAX, SUM, AVERAGE, and COUNT
- TAG can support a broader set.

Aggregates Implemented using 3 functions:

- Merging function *f:* Merges partial state record.
- Initializer *i:* Instantiates state record for single sensor value.
- Evaluator *e:* Computes actual value from the partial state record.

Aggregates

AVERAGE aggregate:

- Each partial state record is of the form <*SUM*, *COUNT*>
- Merging function *f:*
  - *f* (<*SUM1*, *COUNT1*>, <*SUM2*, *COUNT2*>) = <*SUM1 + SUM2*, *COUNT1 + COUNT2*>
- Initializer *i:*
  - *i* (sensor_value) = <*sensor_value*, 1>
- Evaluator *e:*
  - *e* (<*SUM*, *COUNT*>) = SUM/COUNT

Aggregates

Aggregates Taxonomy:

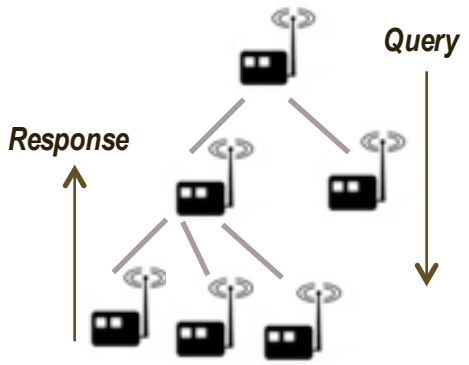
- Define different dimensions for a general classification of the aggregates functions
  1. *DUPLICATE SENSITIVITY*
  2. EXEMPLARY or SUMMARY
  3. MONOTONIC
  4. PARTIAL STATE requirements

	MAX, MIN	**COUNT, SUM**	**AVERAGE**	**MEDIAN**	**COUNT DISTINCT**	**HISTOGRAM**
Duplicate Sensitive	NO	YES	YES	YES	NO	YES
Exemplary, Summary	E	S	S	E	S	S
Monotonic	YES	YES	NO	NO	YES	NO
Partial State	Distributive	Distributive	Algebraic	Holistic	Unique	Content-Sensitive

In-Network Aggregation

Aggregation Consists of two phases:

- Distribution phase: Queries are pushed down into the network.
- Collection Phase: Aggregate values and route them up from children to parent.



Centralized Approach:

- Each node sends back the reply to the central node.
- The central node processes the data to compute the aggregate.

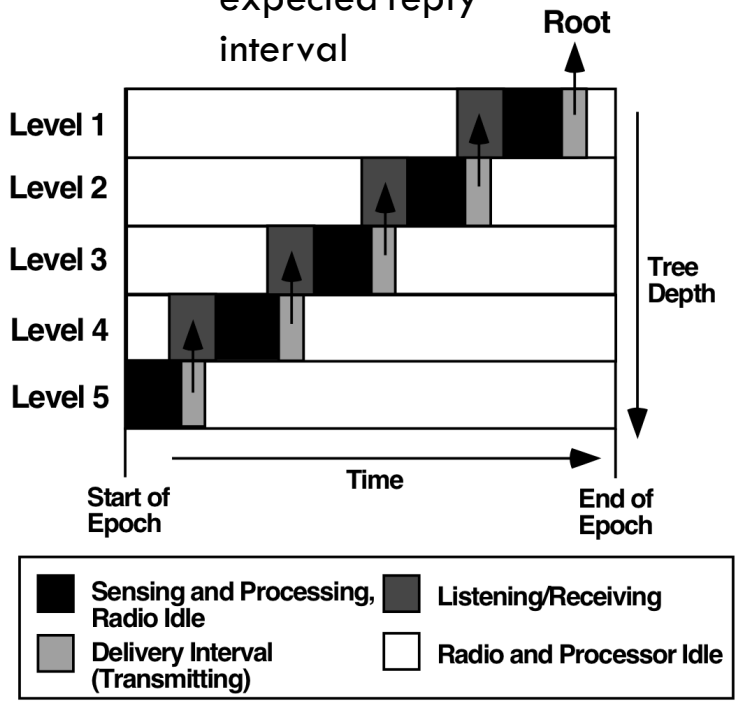
In-Network Aggregation Approach:

- Aggregates are computed within the network.
- Only necessary information sent to the parent nodes.

In-Network Aggregation

Simple Tiny aggregation:

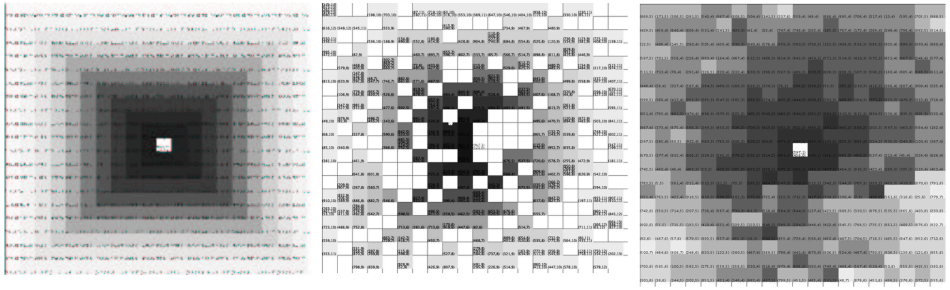
Mote p receives request to aggregate with expect reply interval → Synchronizes the clock and choose the sender of message as the parent. → Forwards the request with its own information and modified expected reply interval → Process own reading with child's partial state record → Reply to parent with the updated partial state record



Simulation-Based Evaluation

Simulator set-up:

- Java based simulation with time divided into units of epochs.
- Doesn't account for byte-level radio contention or time to send and decode messages.
- Models a parallel execution and the costs of topology maintenance
- Three communication models



(a) Simple (b) Random (c) Realistic

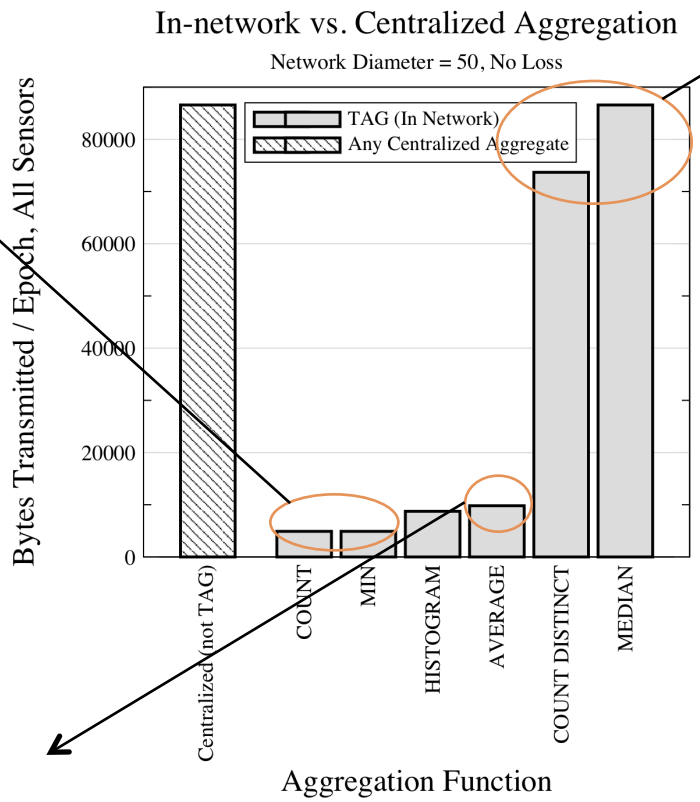
Darker nodes are closer to the roots

Simulation-Based Evaluation

- Cost to flood a request down the tree in not considered.

Extremely low. Only an integer required for the partial state record.

Don't benefit. Require more state to compute the aggregation



Double as compared to COUNT/MIN. Two integers required for the partial state record.

Optimizations

Using the shared channel:

- Snoop on the network to look for missed aggregation request messages.
- Can snoop at specified time interval to save power.

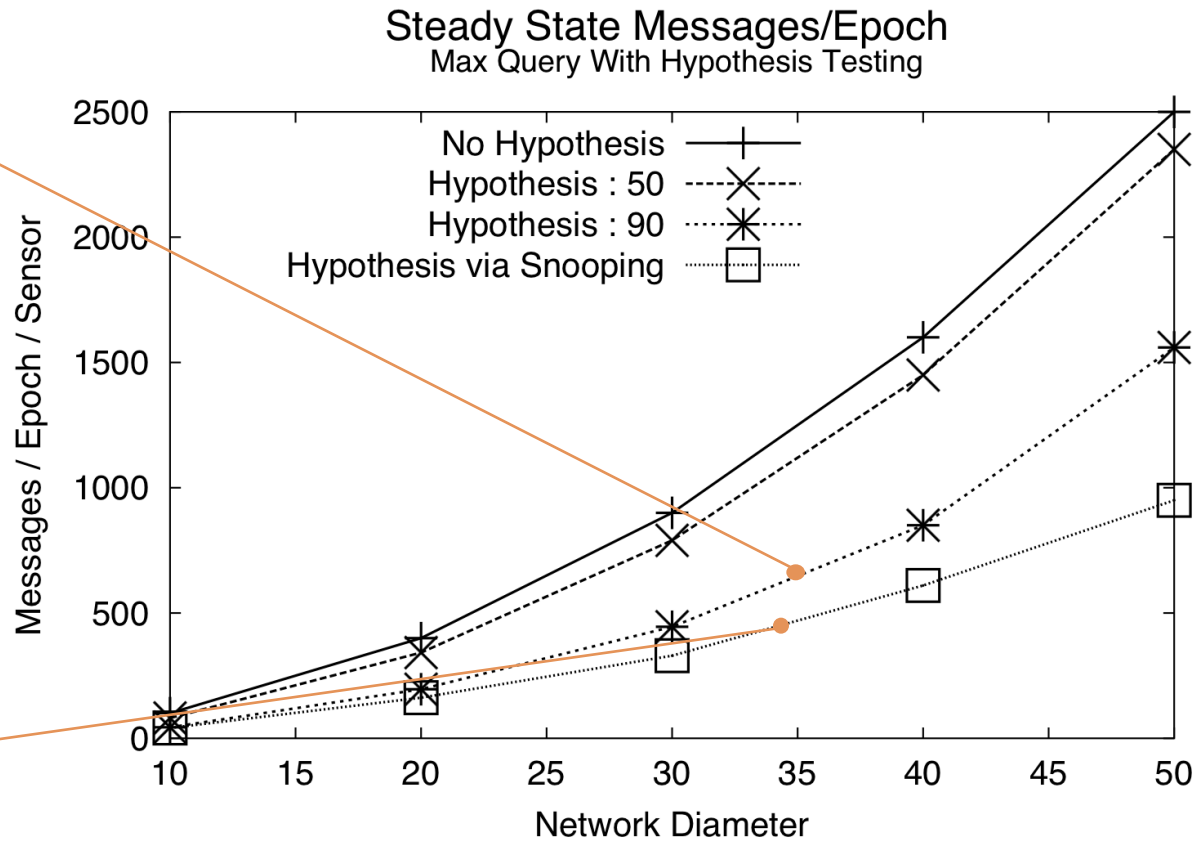
Hypothesis testing:

- The aggregation requests are sent with a guess value.
- Motes only respond if their values affect the end value.
- Motes can also snoop on the network to check if its local value can affect the end value.

Optimizations

Informed guesses can lead to significant performance gain.

Snooping can further improve the perform especially in packed networks.



Tolerating loss

Link Quality monitoring:

- Each node monitors the quality of the link to a subset of its neighbors by tracking the proportion of packets received from each neighbor.
- If a node *n* observes weak link to its parent *p* and better link to *p'*, chooses *p'* as its new parent.

Child caching:

- Improve quality of aggregates by remember child's partial state record for some number of states.

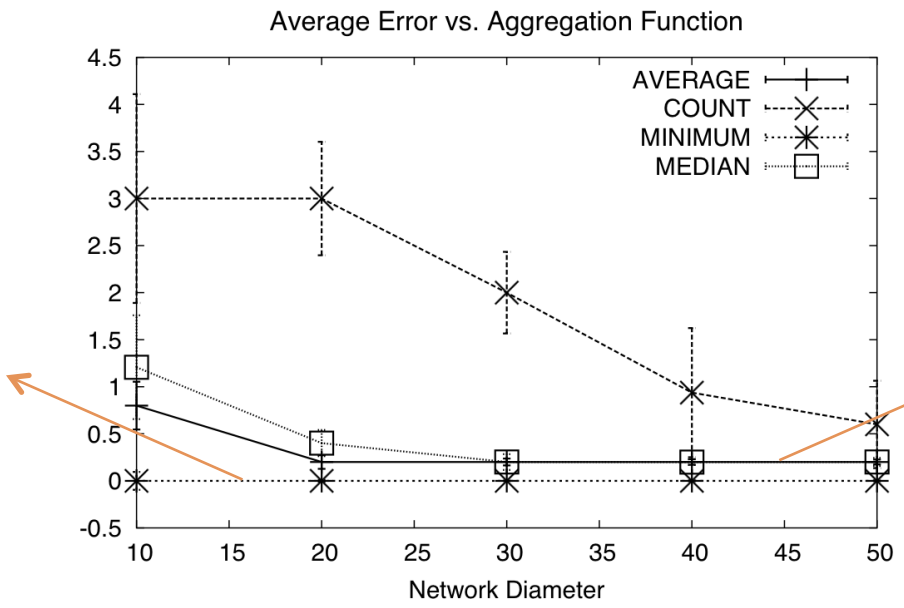
Using available redundancy:

- Uses the network topology to Improve quality of aggregates by maintaining multiple parents and sending part of or whole partial state records to each of them

Tolerating loss

Effect of single loss on various aggregate functions

MIN insensitive to loss as several nodes are at the true minimum.



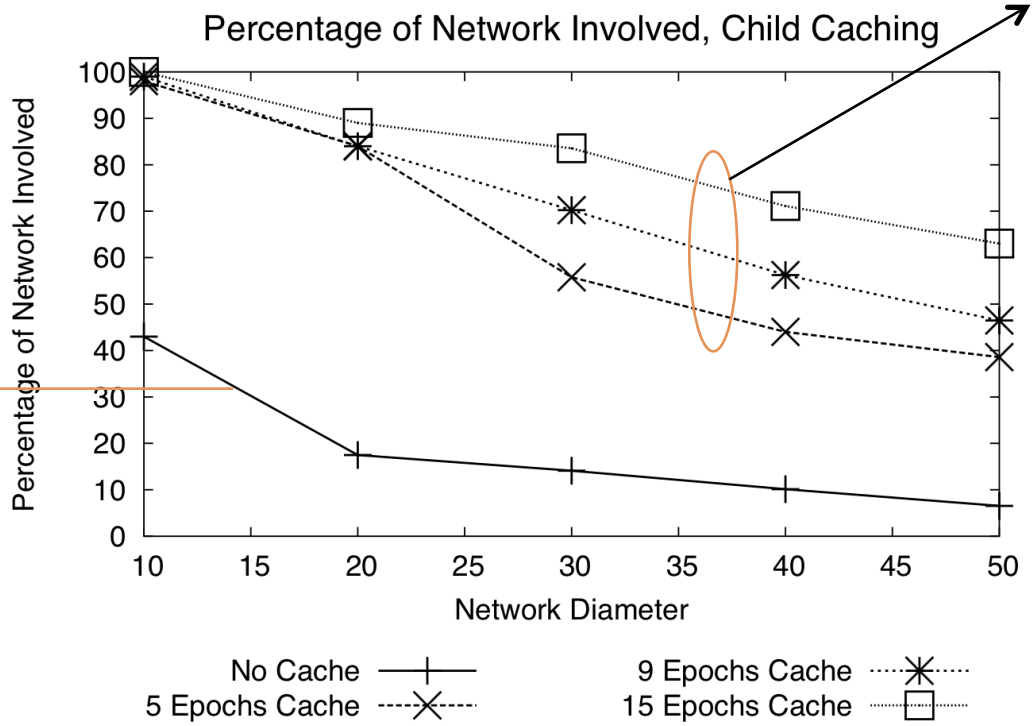
MEDIAN and AVERAGE are somewhat sensitive to variation in number of nodes.

Average Error

Tolerating loss

Realistic communication

Significant in number of participating nodes.



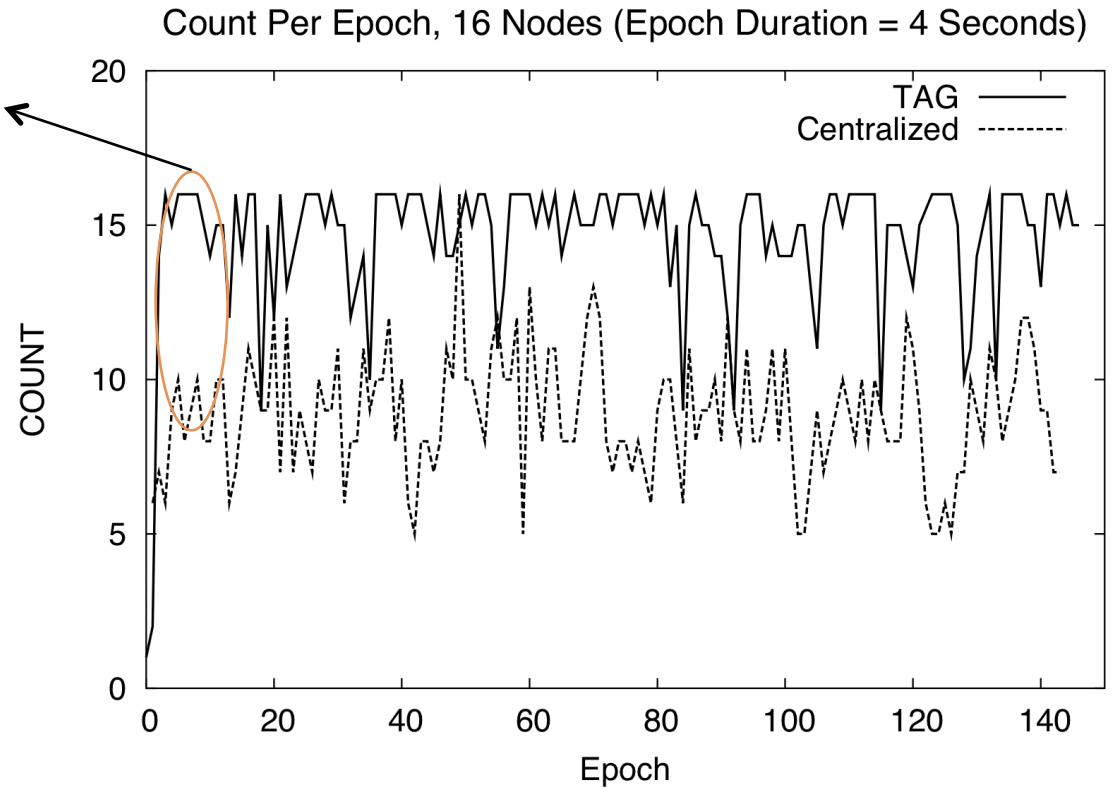
Without caching TAG or even centralized aggregation not highly tolerant to loss.

Prototype implementation

Set-up:

- Does not include optimizations.
- Uses 16 TinyOS Mica motes arranged in a depth four tree.

Better Quality of aggregation



50% reduction communication

Conclusions

- Offers significant reduction in network communication. A node needs to transmit only once per epoch.
- Improves the ability to tolerate losses.
- Epochs provide a convenient mechanism to put CPU to sleep.
- Offers up to an order of magnitude reduction in bandwidth consumption.
- Simple declarative query interface

Pros & Cons

Pros:

- Offers an order of magnitude reduction in network communication.
- Provides mechanisms to tolerate losses and improve quality of aggregates.
- SQL-like syntax for query makes it easy to use.

Cons:

- The simulation is too simplified as it doesn't account for power, CPU and other important aspects.
- The prototype implementation consisted of merely 16 motes.
- It is unclear on what the interval value should be when forwarding the query.
- Malicious nodes can corrupt/falsify the aggregates.

Discussion

- Evaluation:
  - Not a through evaluation.
  - Doesn't consider power consumption, CPU utilization and radio contention in simulated evaluation.
- Malicious Nodes can corrupt aggregation values.
- Affects of stale caches are not considered.
- Tree based topology:
  - Require extra communication to form the tree.
  - Failure prone.