Simplified Programming of Faulty Sensor Networks via Code Transformation and Run-Time Interval Computation

Lan S. Bai†, Robert P. Dick†, Peter D. Dinda‡, Pai H. Chou*

†University of Michigan

‡Northwestern University

*University of California, Irvine

Outline

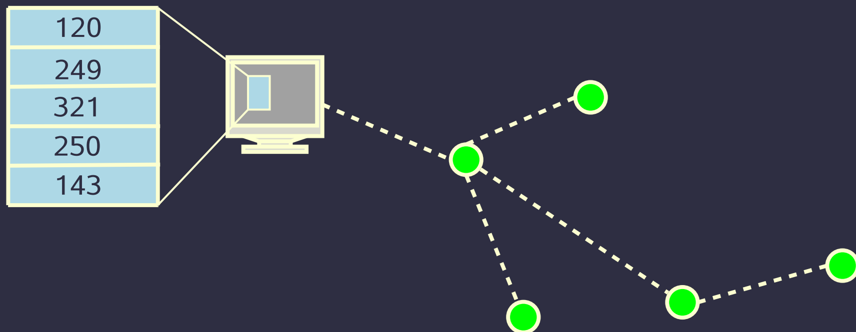
1. Introduction

2. FACTS system design

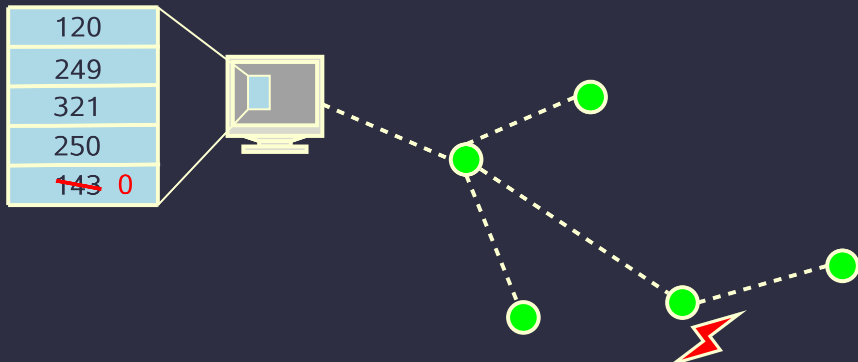
3. Evaluation

4. Conclusions

Motivation

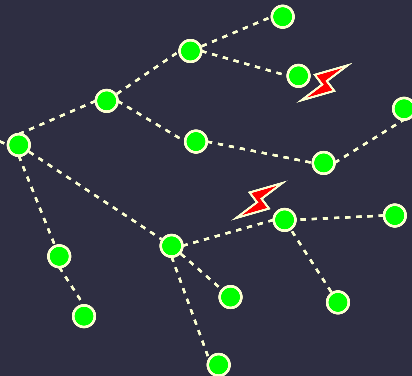


Motivation



Motivation

352	~~120~~ 30
~~230~~ 0	249
300	321
259	250
110	~~143~~ 0
~~234~~ 10	290
~~375~~ 100	333
350	290
~~420~~ 125	240



Motivation

Designing a wireless sensor network is challenging.

Faults are common in wireless sensor networks.

Manual and ad hoc fault detection or error correction is challenging or expensive.

Existing fault detection and error correction techniques require substantial efforts for novice programmers to learn and use.

High-level sensor network programming languages and fault tolerance mechanisms generally designed in isolation.

Goals

Allow novice programmers to design sensor networks that operate in harsh environments, without explicitly handling faults.

Make it easy to understand the impact of faults on system-level information.

Combine high-level programming languages and automatic fault-aware code transformation.

Leverage existing fault detection and correction techniques.

Insights and ideas

Novice programmers tend to assume a fault-free system.

Users should know impact of faults on results.

Users do not need to know low-level fault details.

Use domain knowledge (expected behaviors) from application experts for effective fault detection and correction.

Related work I

High-level programming languages for WSN

- Hide low-level implementation details (e.g., node communication) from programmers.
- TinyDB, Regiment, WASP, etc.
- Few provides support for fault detection and error correction.
- No programmer access to handle faults.

Support for failure recovery considered in language design

- Declarative failure recovery for sensor networks (Gummadi AOSD'07).
- Declarative annotations to specify checkpointing recovery strategies.

Related work II

Fault tolerance in WSN

- Classify and model faults in sensor networks.
- Minimize impact of faults on system performance and availability.
- Minimize performance and energy overhead of network diagnosis.
- Koushanfar IEEE Sensors'06, Clouqueur TC'04, Ramanathan Sensys'05, Ni TOSN'09.

Contributions

Code transformation: automatically generate code supporting reliability management, leaving little work for the programmers.

Error estimation: indicate to domain experts consequences of faults, they can appreciate and understand.

Outline

1. Introduction

2. FACTS system design

3. Evaluation

4. Conclusions

Background on WASP language

A compact, high-level programming language for a class of WSN applications (IPSN'09)

- Periodic sampling with data processing and aggregation in a homogeneous stationary network.
- User study shows WASP is accessible to novice programmers.

Node-level code segment specifies sampling and local data processing

Operations apply to time series data that are local to a single node.

Network-level code segment specifies data filtering, aggregation, transmission through network

Operations apply to most recent data from all the nodes in the network.

Example WASP code

Application: get average temperature from a sensor network that samples every 2 seconds.

```
local:
  sample temperature every 2 sec into mytemp
network:
  collect AVG(mytemp)
```

Example WASP code

What if a sensor node fails and produces erroneous readings?

Example WASP code

What if a sensor node fails and produces erroneous readings?

Original system: faulty data are mixed with correct data during aggregation.

Modify WASP program to ignore faulty readings

```
local:
  sample temperature every 2 sec into mytemp
network:
  collect AVG(mytemp)   where mytemp >= 0 and mytemp <= 80
```

Users will not notice anything even if only 1 sensor is functioning among 100 sensors.

Example of extended WASP code

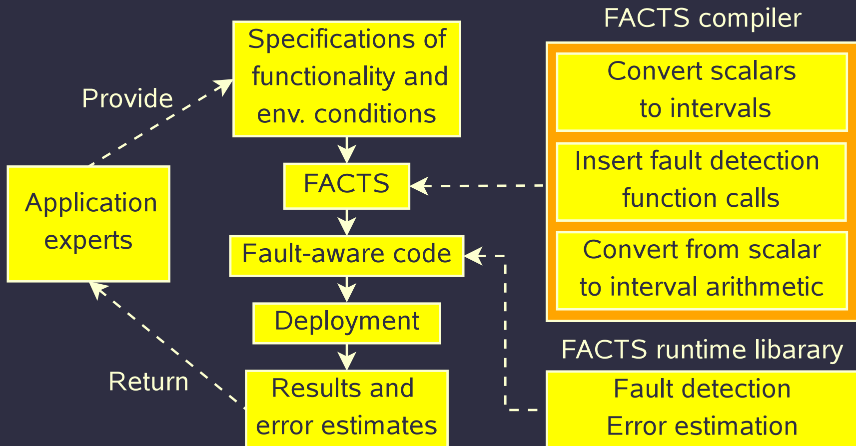
Extend WASP language to specify expected behaviors

```
local:
  sample temperature every 2 sec into mytemp
network:
  collect AVG(mytemp)
expected:
  temperature range [0, 80]
  temperature temporal gradient < 5 degrees per minute
  temperature spatial gradient < 3 degrees per meter
```

Extend results to indicate accuracy

- Users receive AVG(mytemp) as an interval indicating associated error.

FACTS design I



FACTS design II

How FACTS works

- Sensor faults detected online by range checks.
- Faulty sensor readings corrected online using bounds on sensor reading, and temporal and spatial gradients.
- Corrected faulty sensor readings represented with intervals.
- Sensor data intervals propagate to the final results with interval arithmetic.
- User receives requested data represented in intervals.

Fault detection

Sensor faults often result in out-of-range readings.

Check whether sensor readings are in expected ranges.

Check whether sensor works under expected conditions.

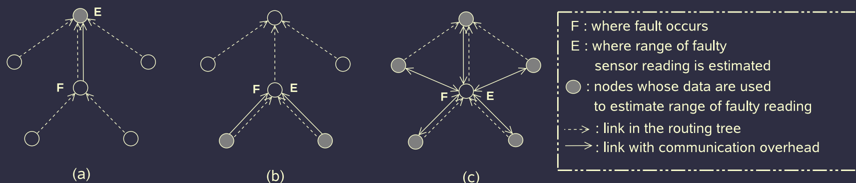
Ranges specified by application experts.

Error estimation

Use bounds on temporal gradient and spatial gradient to estimate ranges of erroneous sensor readings.

Bounds on temporal and spatial gradients specified by application experts.

Alternative approaches for error estimation with spatial data



(a): error computed at parent node using parent node's readings.

(b): error computed at faulty node using its children's readings.

(c): error computed at faulty node using its neighbors' readings.

Key considerations

Network communication overhead.

Tightness of bounds on faulty data.

Implementation complexity.

Error propagation via expression tree

Compute error in final results with interval arithmetic.

Most functions are monotonic (e.g., average, max).

Original: $z = x + y$

FACTS: $z.min = x.min + y.min$

$z.max = x.max + y.max$

Error propagation via control flow

Faulty variable appears in conditional expressions.

Example

collect AVG(y) where x $>$ 100

Node	1	2	3	4	5
x	[120]	[90,110]	[80,120]	[83,102]	[130]
y	2	4	6	3	8

What is the range of AVG(y)?

Could consider all combinations (2^n).

Error propagation via control flow

Compute MIN(AVG(y))

- Sort undecided variables by y.
- Compute average of decided y's.
- Incrementally include remaining y's.
- This guarantees MIN(AVG(y)) will be encountered.
- Stop at local minimum.

Runtime is $\mathcal{O}(n \log n)$.

Converse is true for MAX(AVG(y)).

y's	2	8	3	4	6
AVG(y)		5	4.33	4.25	4.6

Error propagation via control flow

Compute MIN(AVG(y))

- Sort undecided variables by y.
- Compute average of decided y's.
- Incrementally include remaining y's.
- This guarantees MIN(AVG(y)) will be encountered.
- Stop at local minimum.

Runtime is $\mathcal{O}(n \log n)$.

Converse is true for MAX(AVG(y)).

y's	2	8	3	4	6
AVG(y)		5	4.33	4.25	4.6

Error propagation via control flow

Compute MIN(AVG(y))

- Sort undecided variables by y.
- Compute average of decided y's.
- Incrementally include remaining y's.
- This guarantees MIN(AVG(y)) will be encountered.
- Stop at local minimum.

Runtime is $\mathcal{O}(n \log n)$.

Converse is true for MAX(AVG(y)).

y's	2	8	3	4	6
AVG(y)		5	4.33	4.25	4.6

Error propagation via control flow

Compute MIN(AVG(y))

- Sort undecided variables by y.
- Compute average of decided y's.
- Incrementally include remaining y's.
- This guarantees MIN(AVG(y)) will be encountered.
- Stop at local minimum.

Runtime is $\mathcal{O}(n \log n)$.

Converse is true for MAX(AVG(y)).

y's	2	8	3	4	6
AVG(y)		5	4.33	4.25	4.6

Error propagation via control flow

Compute MIN(AVG(y))

- Sort undecided variables by y.
- Compute average of decided y's.
- Incrementally include remaining y's.
- This guarantees MIN(AVG(y)) will be encountered.
- Stop at local minimum.

Runtime is $\mathcal{O}(n \log n)$.

Converse is true for MAX(AVG(y)).

y's	2	8	3	4	6
AVG(y)		5	4.33	4.25	4.6

Error propagation via control flow

Compute MIN(AVG(y))

- Sort undecided variables by y.
- Compute average of decided y's.
- Incrementally include remaining y's.
- This guarantees MIN(AVG(y)) will be encountered.
- Stop at local minimum.

Runtime is $\mathcal{O}(n \log n)$.

Converse is true for MAX(AVG(y)).

y's	2	8	3	4	6
AVG(y)		5	4.33	4.25	4.6

Error propagation via control flow

Compute MIN(AVG(y))

- Sort undecided variables by y.
- Compute average of decided y's.
- Incrementally include remaining y's.
- This guarantees MIN(AVG(y)) will be encountered.
- Stop at local minimum.

Runtime is $\mathcal{O}(n \log n)$.

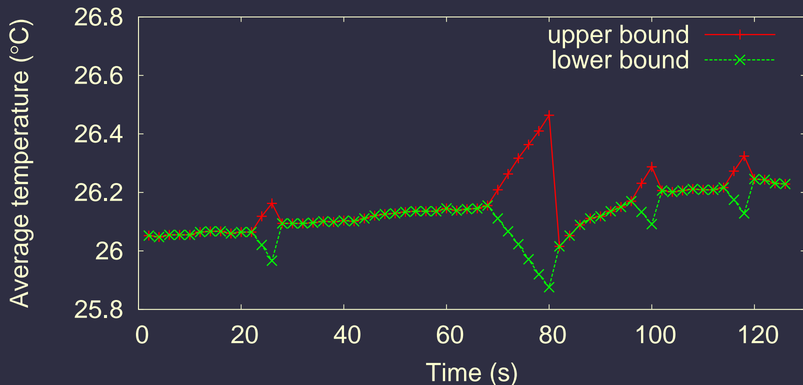
Converse is true for MAX(AVG(y)).

y's	2	8	3	4	6
AVG(y)		5	4.33	4.25	4.6

Outline

1. Introduction
2. FACTS system design
3. Evaluation
4. Conclusions

Prototype evaluation



Tested in a small-scale sensor network composed of four TelosB nodes.

Injected intermittent sensor faults by shorting the terminals of the thermal sensor.

Evaluation of code size and memory use

App. 1: periodically gathers temperature and light data.

App. 2: periodically samples light and averages data among nodes at similar heights.

App. 3: periodically samples temperature and sends data only when the increase in temperature exceeds a threshold.

	Code size (Byte)			Memory usage (Byte)		
	App. 1	App. 2	App. 3	App. 1	App. 2	App. 3
Fault-unaware	32,556	33,060	27,722	2,130	2,134	2,038
Fault-aware	37,358	37,740	32,088	2,212	2,224	2,096
Overhead (%)	14.7	14.2	15.7	3.8	4.2	2.7

Evaluation of programming overhead

	WASP LoC			NesC LoC		
	App. 1	App. 2	App. 3	App. 1	App. 2	App. 3
Fault-unaware	6	7	7	489	495	484
Fault-aware	12	10	10	621	585	545

Evaluation with simulation

Weather data from LUCE deployment at EPFL.

Generate fault-free data traces

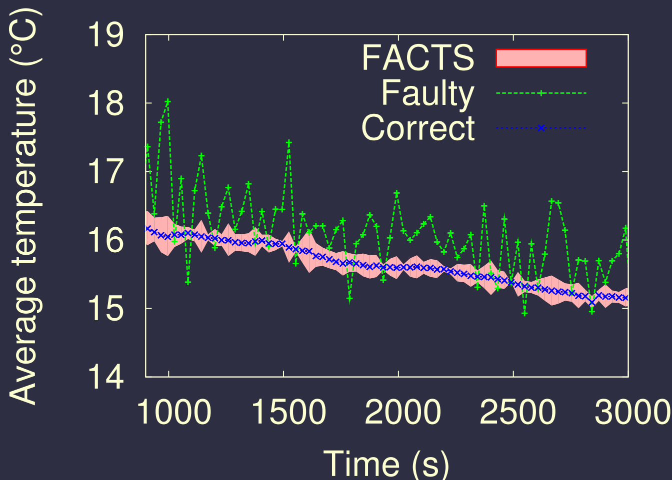
- Identify a time interval and a set of nodes with low data drop rate. One hour data traces (9,028 data samples) from 74 nodes.
- Parse data to produce synchronized periodic time series.
- Determine lower and upper bounds based on histogram. 99.4% data in 5–30 °C range.
- Compute bounds on temporal and spatial correlation. 3 °C per 30 s and 5 °C per 50 m.
- Replace faulty and missing data (3.7%) with values generated based on spatial and temporal correlation.

Evaluation with simulation

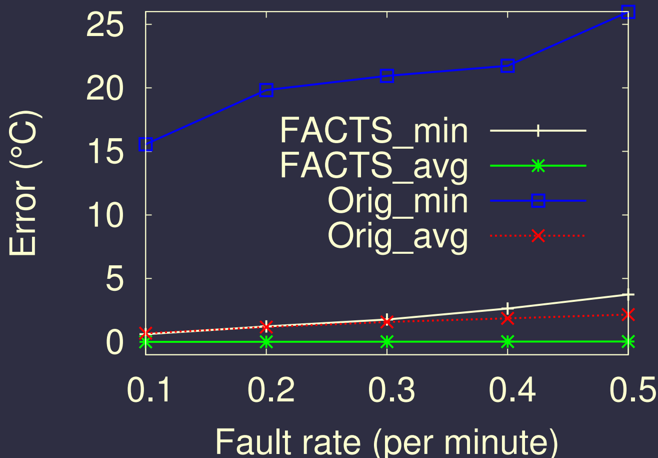
Fault injection during evaluation

- Model sensor transient faults using Poisson processes.
- Independent but equal-rate (0.1 to 0.5 per minute) fault processes for sensor nodes.
- Sensor faults last one sampling period (29.3 s), same as original data set.
- Generate faulty sensor readings by sampling from set of outliers extracted from original data set.

Corrected and uncorrected time series



Error magnitude distribution



Outline

1. Introduction
2. FACTS system design
3. Evaluation
4. Conclusions

Conclusions

Extended a high-level sensor network programming language, its compiler, and runtime system to incorporate reliability management techniques.

Implemented code transformation to generate fault detection and error estimation code.

FACTS uses easily specified domain-specific expert knowledge to support on-line detection of sensor data faults.

FACTS computes accuracy intervals of data analysis expressions to make the system-level impact of faults clear to users.

Fault-aware program has small code and memory overhead.

Acknowledgment

Prof. Lawrence Henschen, Northwestern University.

The SensorScope team, EPFL.

Supported in part by National Science Foundation awards CNS-0721978, CNS-0910816, and CNS-0347941.

Q&A

Thank you