Fiji:
A Macroprogramming Framework for Data-Intensive Sensor Network Applications

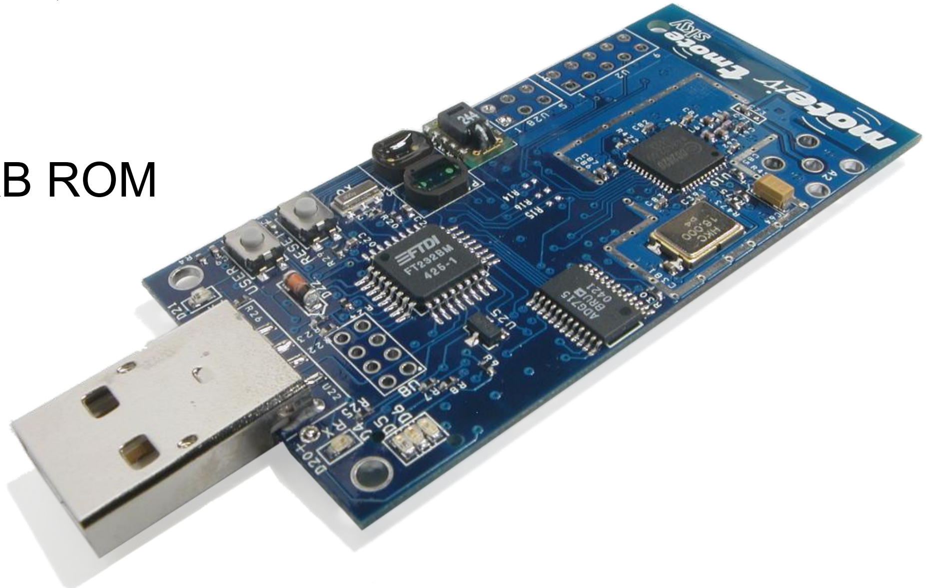
Matt Welsh
mdw@eecs.harvard.edu

Harvard
School of Engineering
and Applied Sciences

Wireless Sensor Networks: A 30-Second Crash Course

New class of computing platform:
- Low power devices with embedded CPU, radio, and sensors

Tmote Sky platform (Moteiv, Inc.)
- 8 MHz (TI MSP430) CPU, 10 KB RAM, 60 KB ROM
- 2.4 GHz IEEE 802.15.4 (“Zigbee”) radio (Chipcon CC2420)
- 1 MByte flash for data logging



Designed for low power operation
- 1.8 mA CPU active, 20 mA radio active
- 5 uA current draw in sleep state

Runs a lightweight embedded OS, called TinyOS (www.tinyos.net)

Cost: About $75 (with no sensors or packaging)

The Problem

Sensor networks increasingly used for **data intensive** applications:

- Structural monitoring: vibrations, seismic response
- Geophysical monitoring: earthquakes, fault zones, volcanoes
- Biomedical monitoring: EKG, EEG, movement, physical activity

Challenging data fidelity and processing requirements

- Not just a matter of “periodic aggregation up a spanning tree”
- Instead: sophisticated signal processing, reliable data delivery, and fine-grained time synchronization

Domain scientists want to develop sophisticated codes

- But programming sensor nodes is ***hard!***
- End users should not have to deal with the low-level details of embedded processors, sensors, energy management, flash storage, and radio communication.

The Solution: Fiji

Fiji is a **distributed operating system** and **programming framework** for sensor networks that supports:

- High-fidelity applications with precise timing and high-resolution data
- A resource aware programming model that supports adaptation to changing resource availability
- High-level programming languages at multiple levels of abstraction

Fiji is designed to support **macroprogramming**

- Program the network as a whole, not individual nodes
- Automatically compile from global description to local node program

Primary goal:

- *Make it easy for domain scientists to develop complex, distributed applications for sensor networks that adapt to resource availability.*

Outline

Application vignettes and motivation.

Overview of the Fiji system.

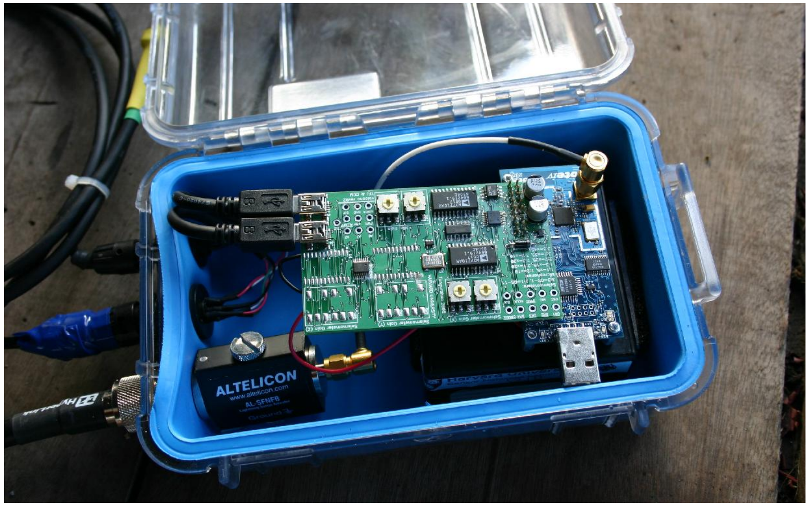
Pixie: An OS for resource-aware programming.

Flask: A dataflow-oriented intermediate language.

Regiment: A macroprogramming language for spatial computations.

Wrap-up.

Application Vignette: Wireless Sensors for Volcanic Monitoring



Nodes monitor seismic and acoustic signals

- 100 Hz data rate, custom 24 bps ADC card
- 8.5 dBi antenna to extend range

Core research challenges:

- Reliable data collection over multihop routes
- Event detection to trigger data capture and download
- Time synchronization to permit signal correlation
- Remote monitoring and administration of network in hostile environment

Reventador deployment: 16 nodes deployed for three weeks

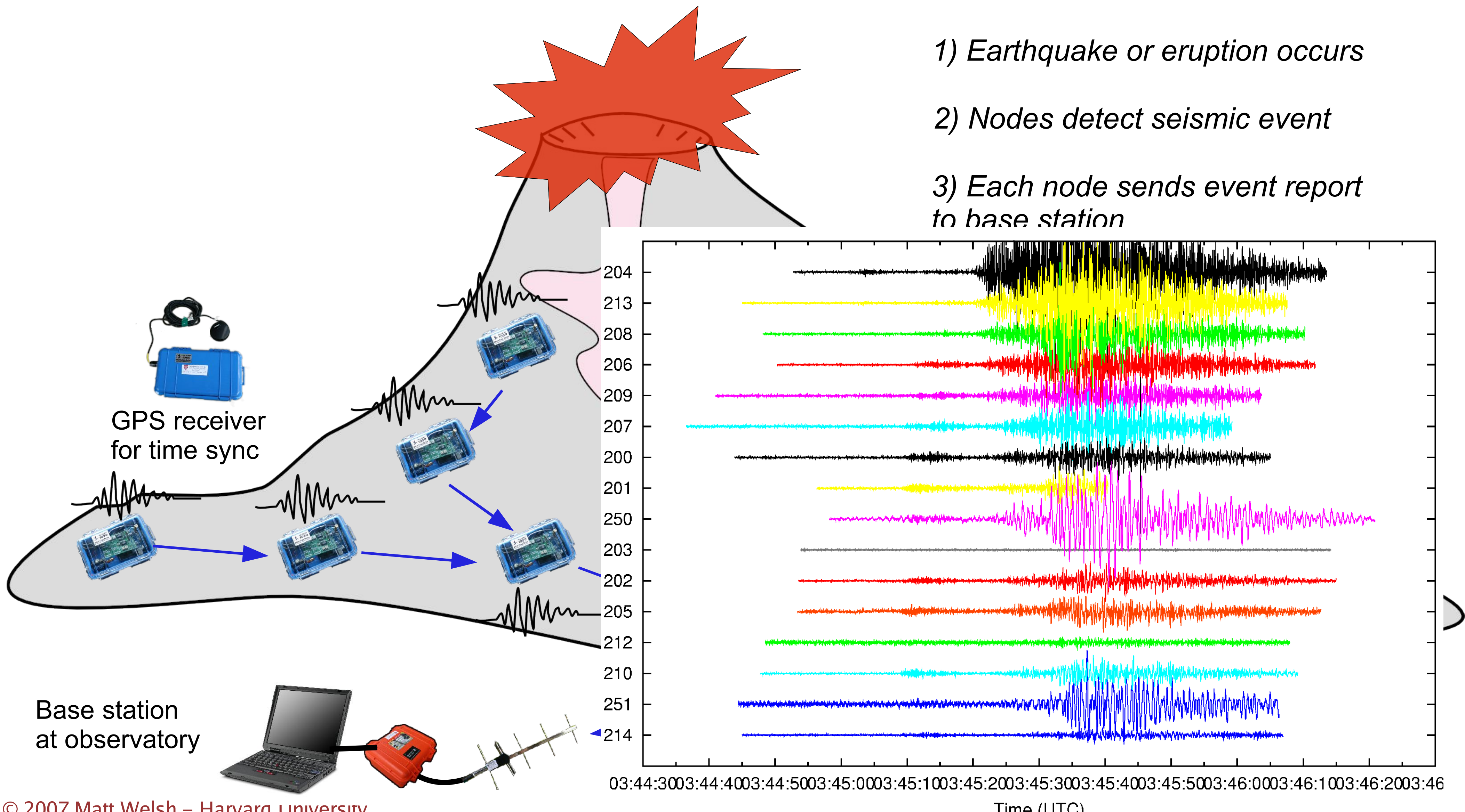
- Linear topology spanning ~3km radially from vent; base station located a further 4 km from deployment site.
- Captured data on hundreds of earthquakes and eruptions
- Extensive post-processing to correct timing errors and validate data

Network Architecture

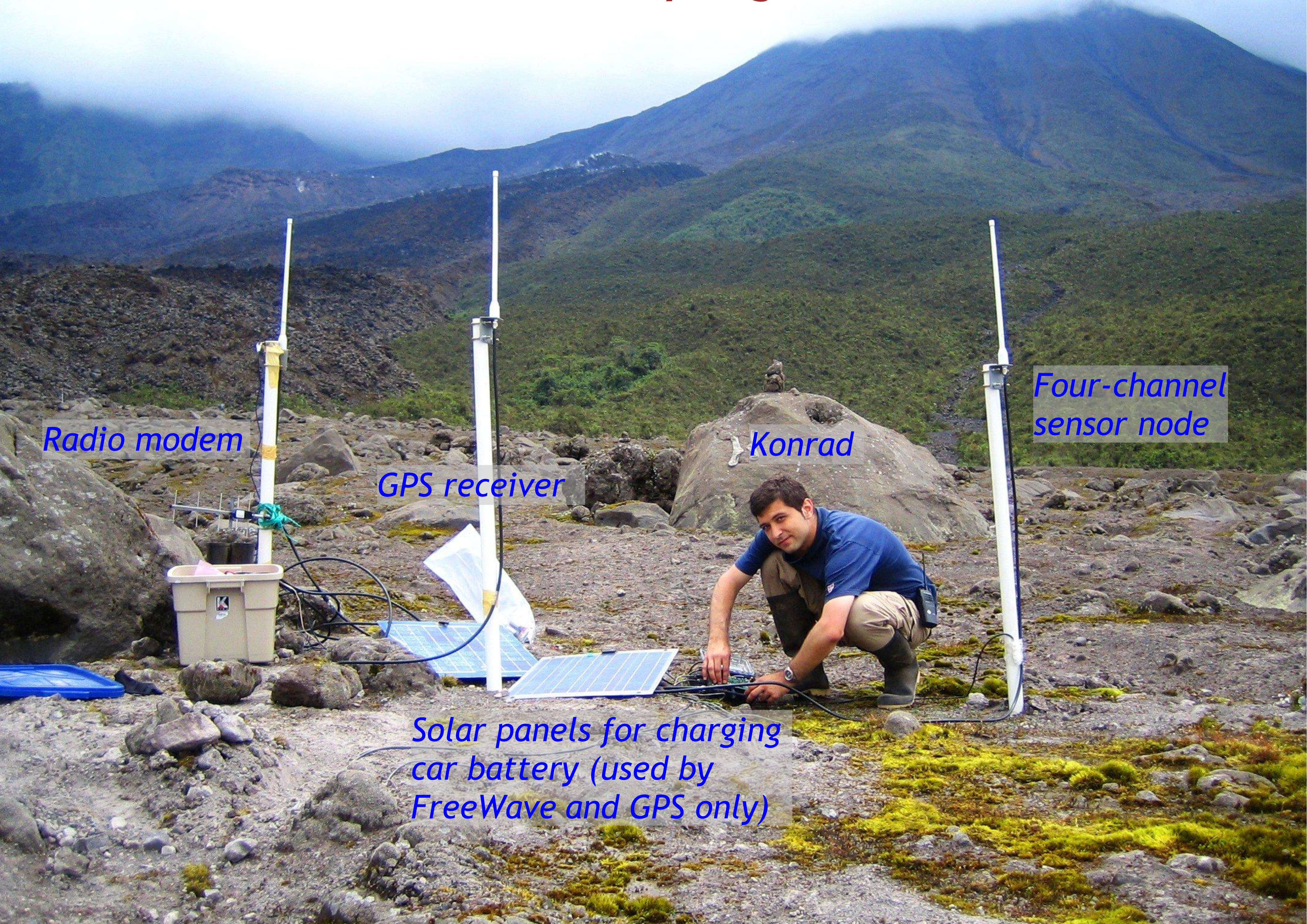
1) Earthquake or eruption occurs

2) Nodes detect seismic event

3) Each node sends event report to base station

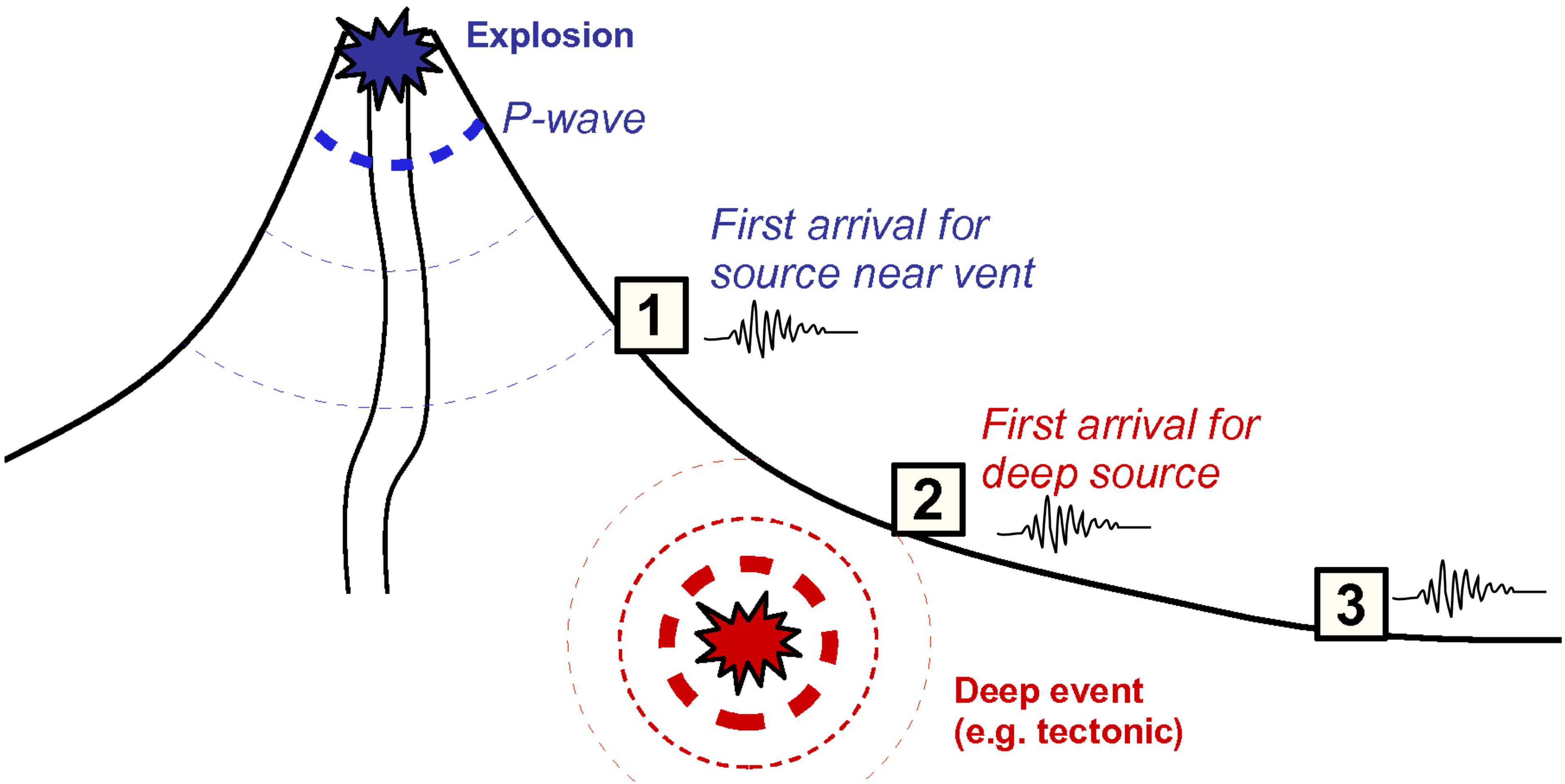


Reventador Volcano, Ecuador (July-August 2005)

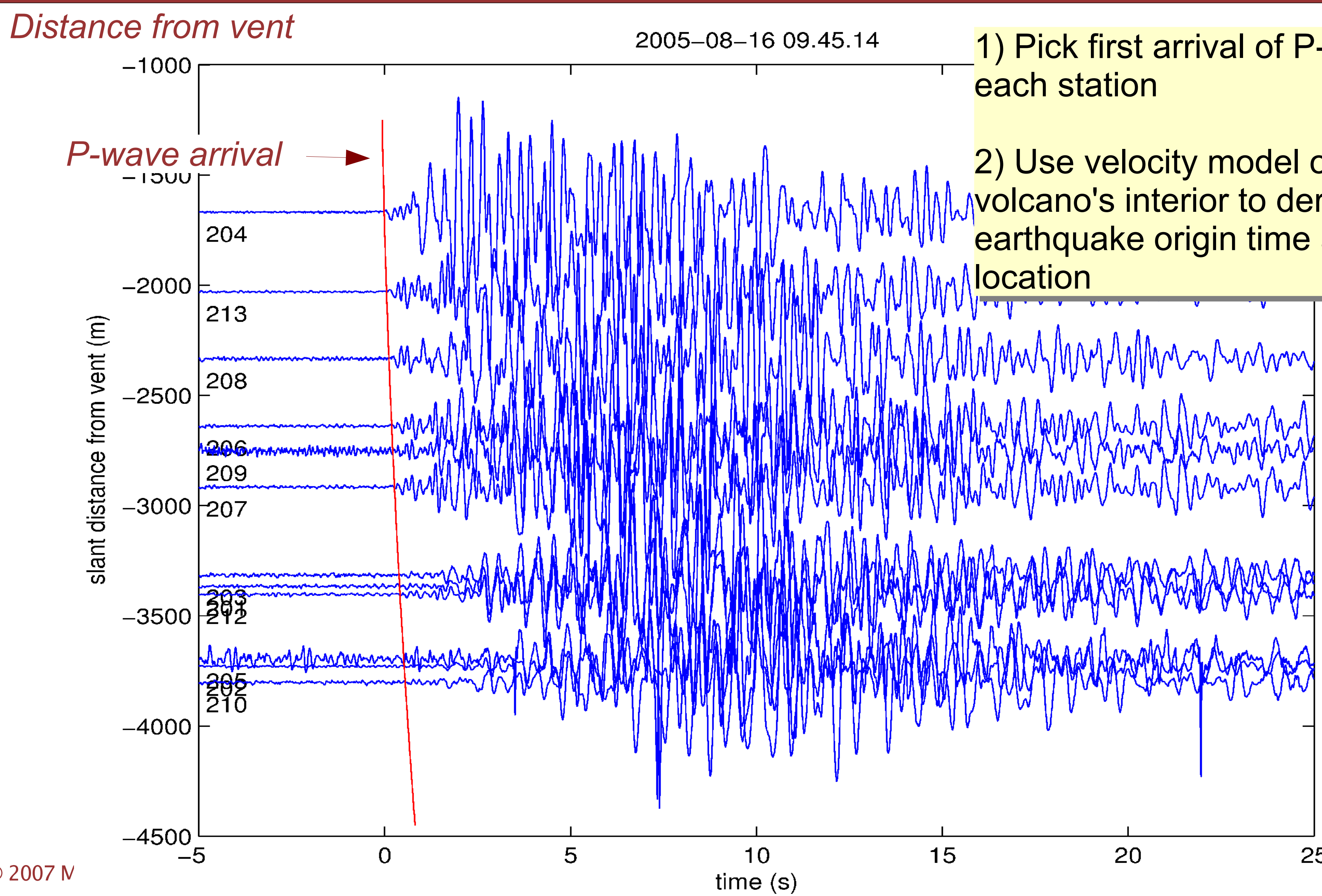


In-network Earthquake Localization

Earthquake location can be derived from P-wave arrival times at each station



In-network Earthquake Localization

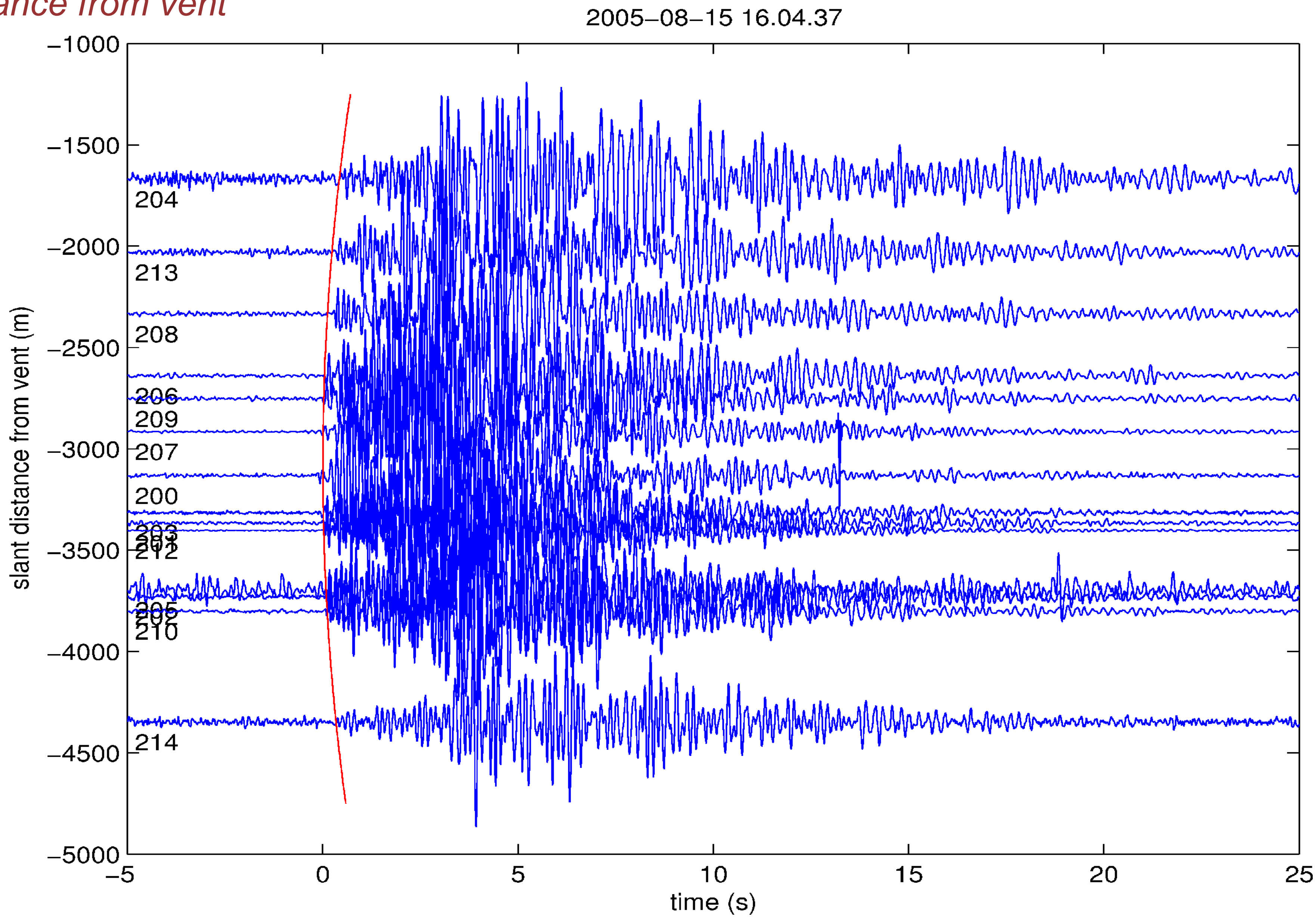


1) Pick first arrival of P-wave at each station

2) Use velocity model of volcano's interior to derive earthquake origin time and location

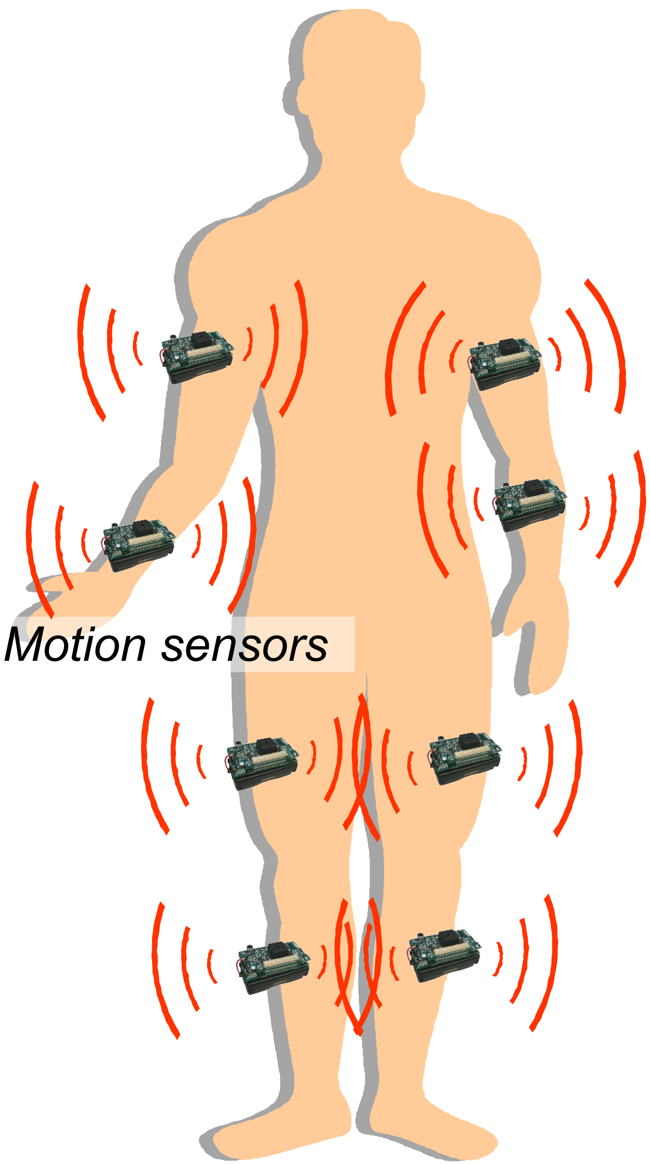
In-network Earthquake Localization

Distance from vent



Parkinson's Disease and Stroke Rehab Monitoring

with P. Bonato, Spaulding Rehabilitation Hospital



High-fidelity monitoring of limb motion

- Triaxial accelerometer, triaxial gyroscope
- 6 channels per node, 100 Hz per channel
- Full resolution signal exceeds radio bandwidth
- Store raw data to flash (2 GB MicroSD)

Nodes perform local feature extraction

- RMS, jerk, dominant frequency, other features...
- Computationally intensive processing
- Requires communication (e.g., time sync, and signal correlation across nodes)

Offline classification to map features to clinical scores

- UPDRS scale

Intel SHIMMER mote

Things to notice about these applications...

High data rates with fine-grained time synchronization requirements

- 100 Hz sampling rate, multiple channels per node
- Timing accuracy is paramount to support signal processing!

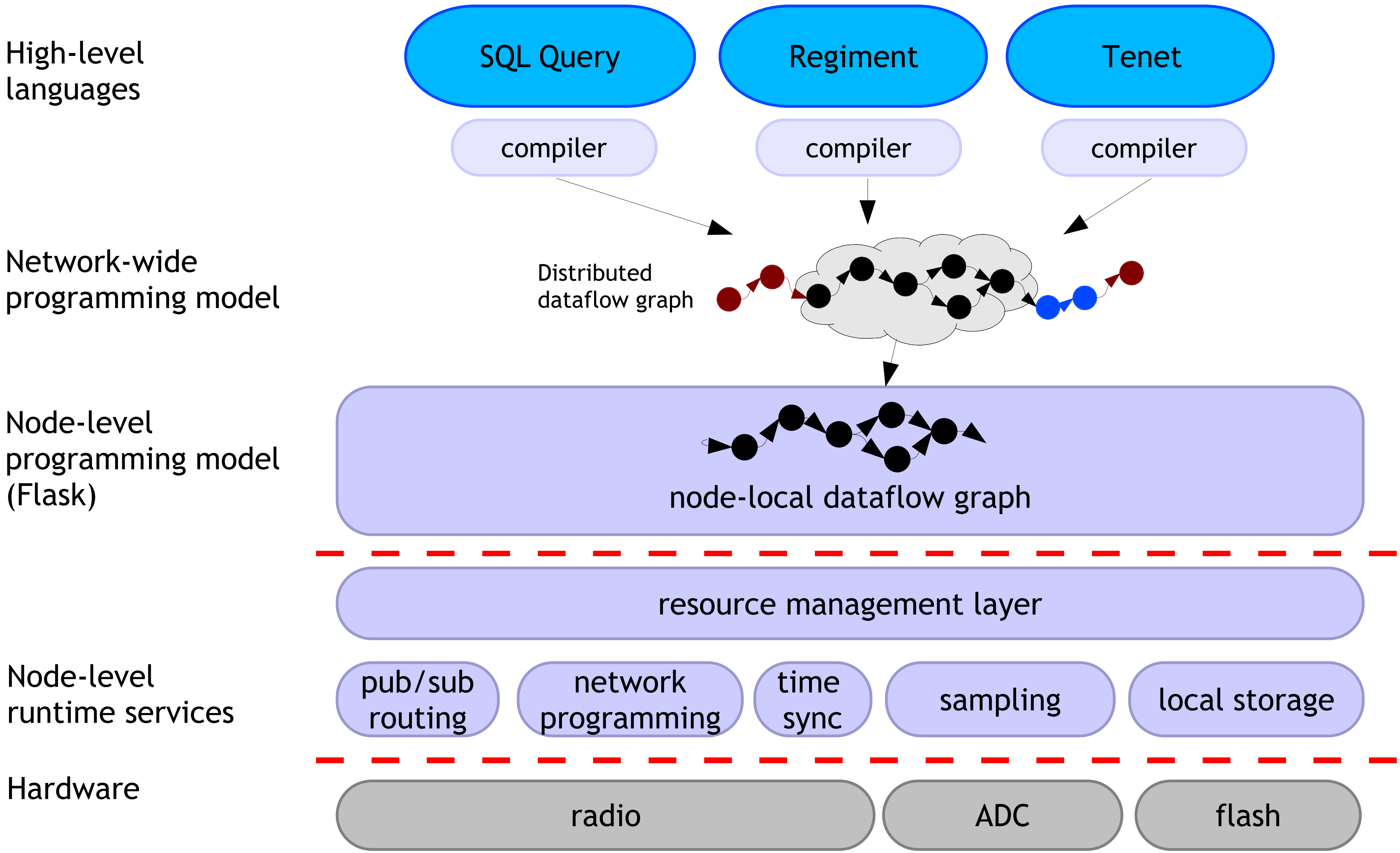
Fairly complex **domain-specific** processing:

- P-wave arrival computation, velocity model to extract earthquake locations
- Domain-specific feature extraction and classification of motion sensor data

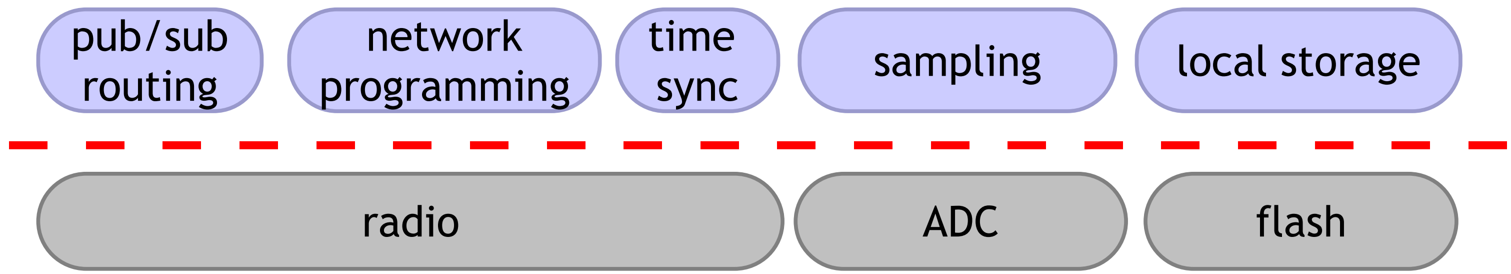
Applications should adapt to changing resource availability

- e.g., Variations in sensor data, changing energy reserves, or fluctuations in radio bandwidth
- Adaptation is also highly application-specific

Fiji design overview



Node-level runtime



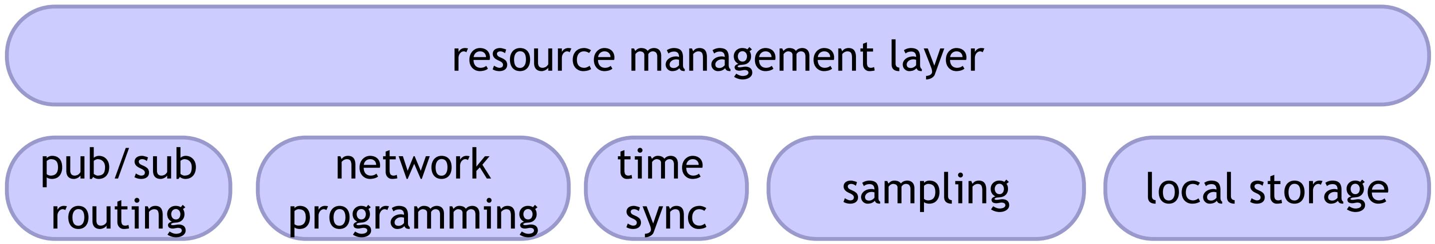
Not the focus of this project! Let's build on the best stuff out there:

- Pub/sub routing protocol: **Flows** (Harvard)
  - *Fairly general API, can be specialized for specific environments*
- Network reprogramming (e.g., Deluge)
- Time sync (e.g., FTSP)
- Sampling and flash storage layers (built at Harvard)

Need an appropriate runtime for each supported hardware platform

- TinyOS/NesC for motes and iMote nodes
- Linux for base station and more powerful gateways (requires subset of functionality, i.e., no sensors)

Resource management layer



Arbitrates access to sensor node resources:

- Manages radio bandwidth in congested/bursty/shared environments
- Manages node energy reserves (battery, solar power, etc.)
- Synchronizes actions across sensor nodes as necessary (e.g., scheduled duty cycling)

Fundamental shift in the **node-level OS design**

- All low-level resources provide ***feedback*** on availability and congestion
- Applications must be designed to respond to feedback and issue resource requests

Current prototype: **Pixie**

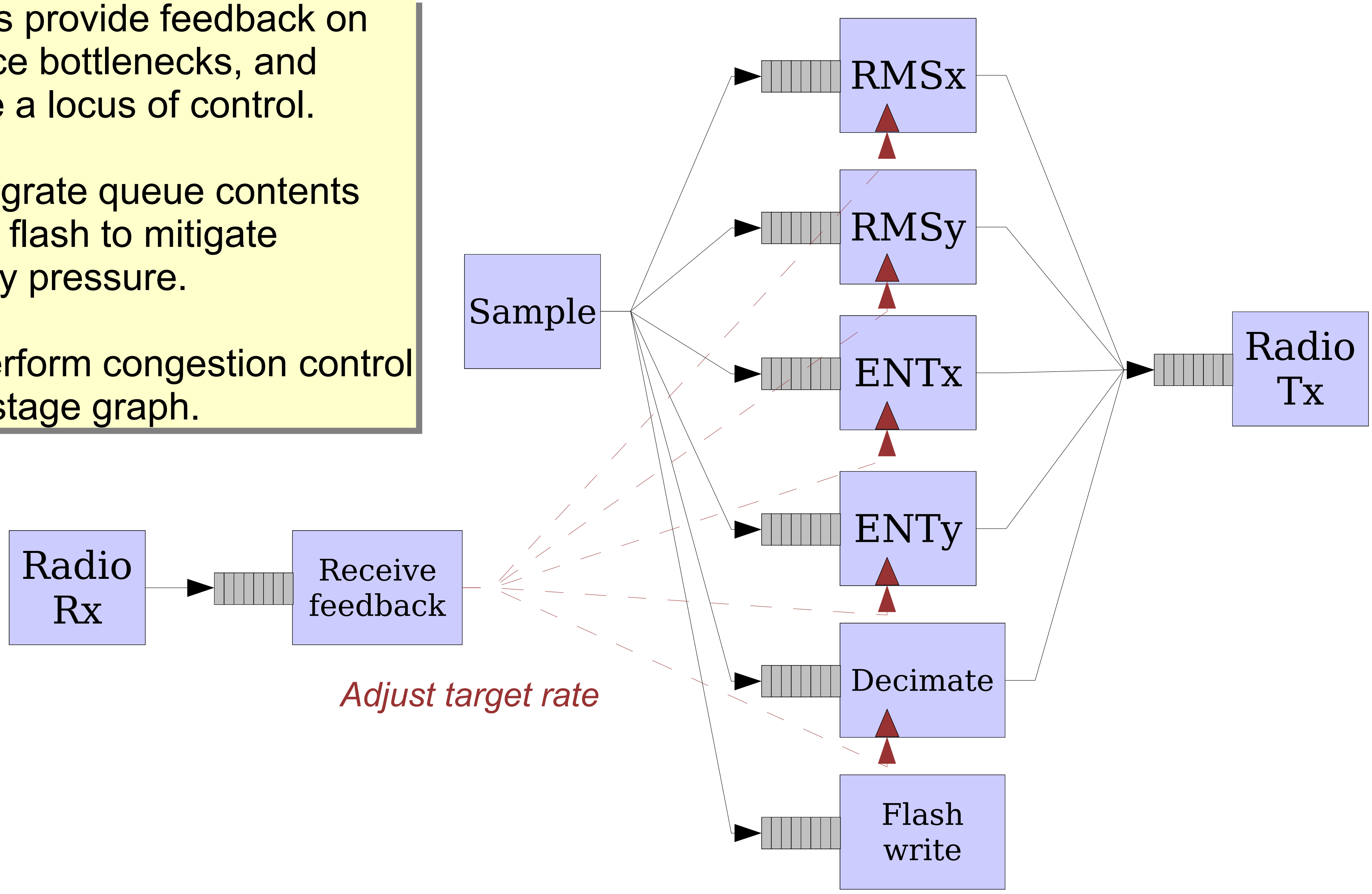
- New node OS (implemented in NesC) based on staged concurrency model
- Allows fine-grained control over CPU, storage, memory, and bandwidth usage

Pixie example

Queues provide feedback on resource bottlenecks, and provide a locus of control.

Can migrate queue contents to/from flash to mitigate memory pressure.

Can perform congestion control within stage graph.



Resource management in Pixie

Radio bandwidth management

- Provide feedback to application stages on available bandwidth (varying due to routing path, node mobility, interference, etc.)
- Application-specific policy adjusts target Tx rate for each stage to avoid congestion

Memory management

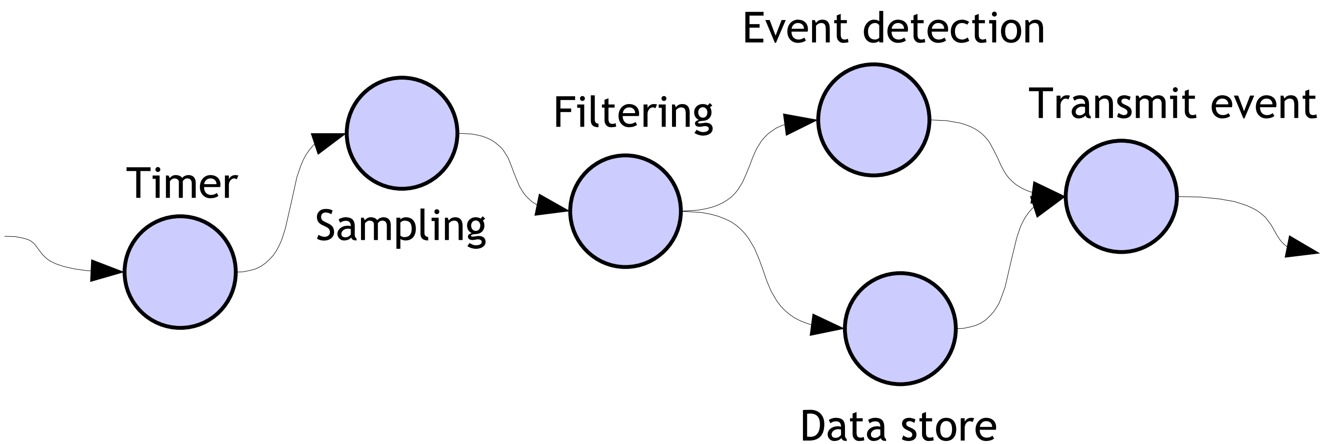
- Large flash memories (2GB+) becoming commonplace; let's take advantage of this.
- Idea: Swap queue contents to/from flash when memory pressure is high
- Assumes some stages can tolerate (potentially high) delays in processing

CPU scheduling

- Stage queues provide direct feedback to application and runtime system on resource bottlenecks
- Borrow ideas from SEDA [1] for adjusting CPU priority and queue admission rate within stage graph

[1] Welsh et al., SOSP 2001

Flask: A dataflow programming toolkit



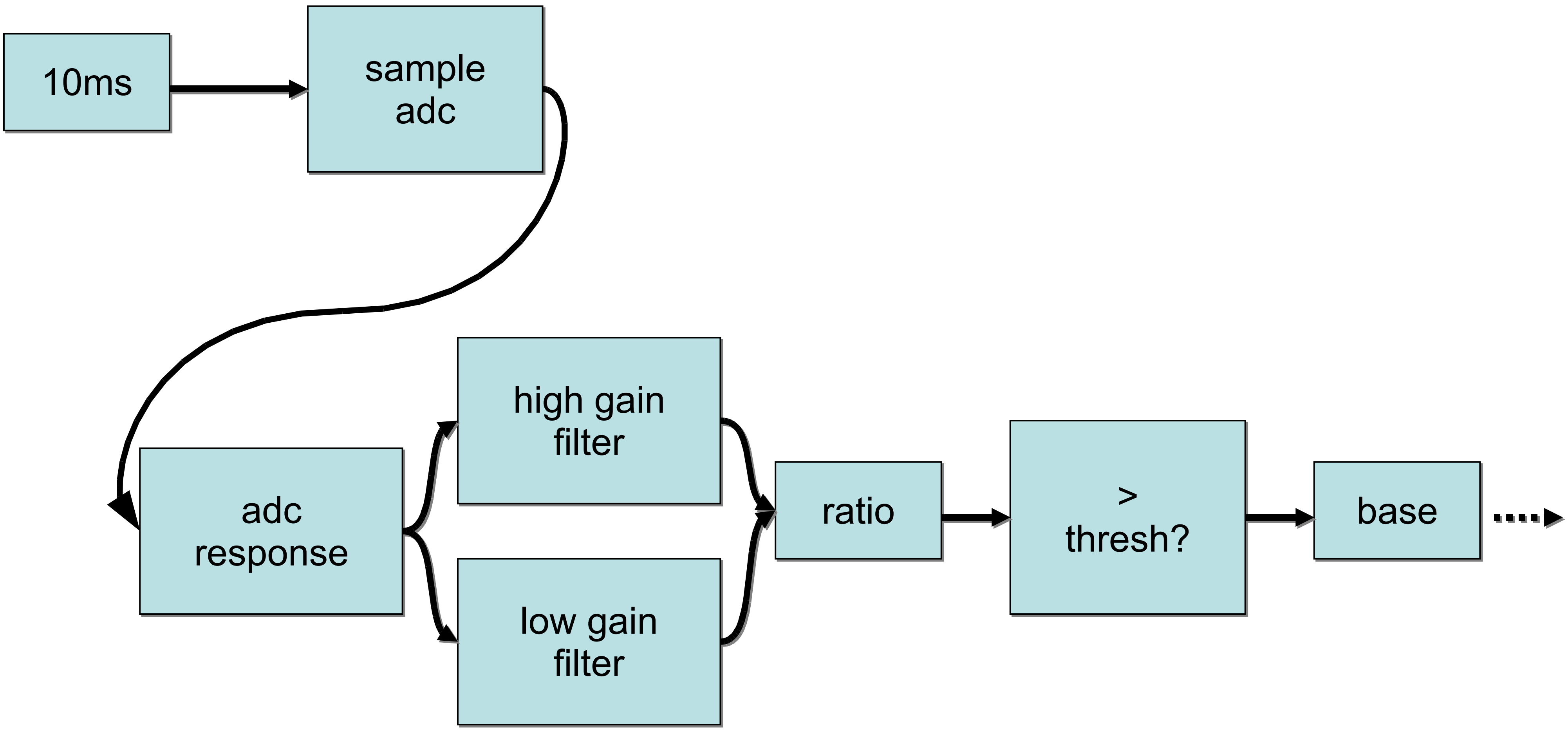
Unified programming abstraction for the sensor node level.

- More structured (and limited) than NesC/TinyOS
- Easier to synthesize from high-level descriptions; possible to weave multiple graphs (from different apps!) together on same node

Flask: A dataflow programming framework for sensor networks

- Flask defines a dataflow intermediate language
- Provides a compiler from dataflow to NesC and Pixie/TinyOS
- Higher-level language compilers can then be implemented using Flask

A Simple Earthquake Detector



Flask Code for Earthquake Detection App

```
let quake_detector (high,low,thresh) =
  let c : unit stream = Flask.clock 10
      s : float stream = Flask.seismometer c
      h : float stream = ewma_filter high s
      l : float stream = ewma_filter low s
      r : float stream = ratio h l
      t : float stream = filter_thresh thresh r
  in
     Flask.send(t,Flask.base_id)
```

Compare with ~200 lines of NesC code.

Basic Dataflow Abstraction: Streams

Key Type: α `stream` ($\approx$ `time` $\rightarrow$ α)

`clock : int` $\rightarrow$ `unit stream`

`seismometer : unit stream` $\rightarrow$ `float stream`

`zip :` α `stream` $\rightarrow$ β `stream` $\rightarrow$ $(\alpha * \beta)$ `stream`

Similar to Yale work on *functional reactive programming.*

- c.f., Eliot, Hudak, Peterson, et al.
- Closer to hardware specification languages (e.g., Arvind's Bluespec), but embedded within a functional language.

Typical Dataflow Embedding in Flask

Other stream combinators:

$\texttt{map} : (\alpha \rightarrow \beta) \rightarrow \alpha\ \texttt{stream} \rightarrow \beta\ \texttt{stream}$

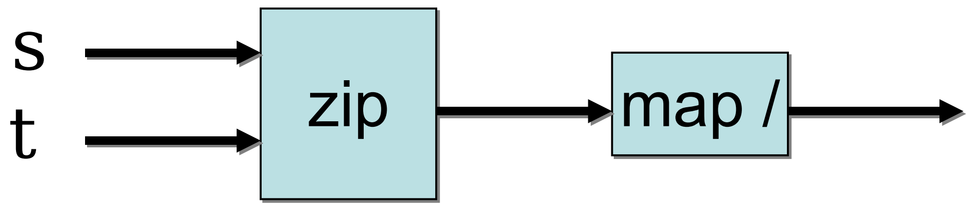
$\texttt{filter} : (\alpha \rightarrow \texttt{bool}) \rightarrow \alpha\ \texttt{stream} \rightarrow \alpha\ \texttt{stream}$

$\texttt{integrate} :$

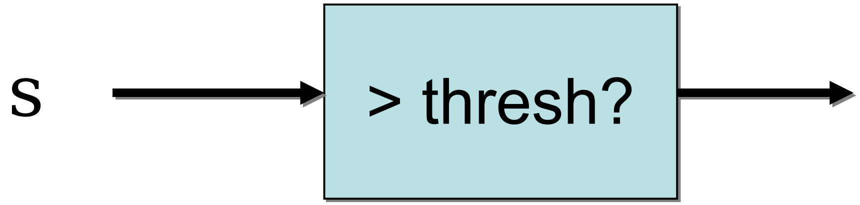
$(\alpha * \beta \rightarrow \beta * \gamma) \rightarrow \beta \rightarrow \alpha\ \texttt{stream} \rightarrow \gamma\ \texttt{stream}$

Examples: zip, map, filter

```
let ratio (s:float stream) (t:float stream) =
    let p : (float * float) stream = Flask.zip s t
    in
       Flask.map (λ (x,y) => x / y) p
```




```
let filter_thresh (thresh:float) (s:float stream) =
    Flask.filter (λ x => x > thresh) s
```



Flask Implementation

Flask is implemented as a *metaprogramming toolkit*

- Implementations in OCaml and Haskell
- Dataflow graphs described as a **wiring program** using the Flask toolkit
- Programmer can leverage all of the power of functional programming to compose dataflow graphs – contrast to NesC's limited “boxes and arrows” wiring language

Body of each dataflow operator implemented in one of two *object languages*:

- NesC (directly inlined into OCaml code)

```
let ratio s = Flask.map (<:cfunc
      float ratio(x,y) { return x / y; }
    >>) s
```

- Hump (subset of Core ML, better integration with surrounding OCaml)

```
let inc s = Flask.map (.< λx => x + 1 >.) s
```

Flask Overhead

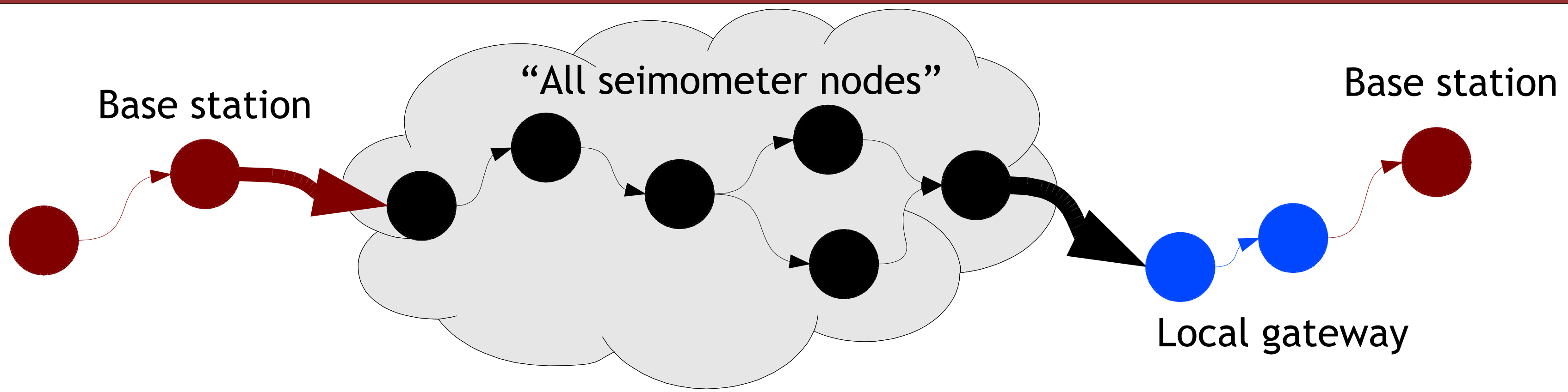
CPU microbenchmarks compared to native NesC code:

	Flask	NesC
Simple filter	4	4
Zip	71	41
EWMA filter	707 ± 132	441 ± 83
Windowed average	1700 ± 78	585 ± 59
Chained calls	42	49
Earthquake detection	2194 ± 127	1580 ± 63

Memory footprint for volcano monitoring application:

	Flask		NesC	
	ROM	RAM	ROM	RAM
Base TinyOS	8308	926	8332	1070
Communications	7416	2657	2840	1638
Common	10734	916	10360	940
Application specific	12096	5138	16380	5540
Total	**38554**	**9637**	**37912**	**9637**

Network level programming model: distributed dataflow graphs



Describes dataflow graph distributed throughout network

- Computation is decomposed across different node types
- Replicated subgraphs across "regions" of nodes
  - *e.g., All seismometer nodes form a region and run a subgraph*
  - *Regions may be determined statically or dynamically*

Edges represent inter-node or intra-node communication

  - *Fat arrows represent scatter/gather to or from a group*
- Can migrate operators in the network at runtime

High level languages and macroprogramming

SQL Query | Regiment | Tenet

A core goal of Fiji is to support multiple high-level languages for application development

- Choice of language is highly domain-dependent
- Some languages we can support already:
  - *SQL query (e.g., TinyDB) for periodic data collection and processing*
  - *Regiment for more sophisticated distributed computing*
  - *Tenet for simple multi-tier applications*

Compile each language down to uniform distributed data flow graph specification, using Flask

- Clean interface between language compiler and distributed runtime system
- Use Flask to generate optimized node-level code

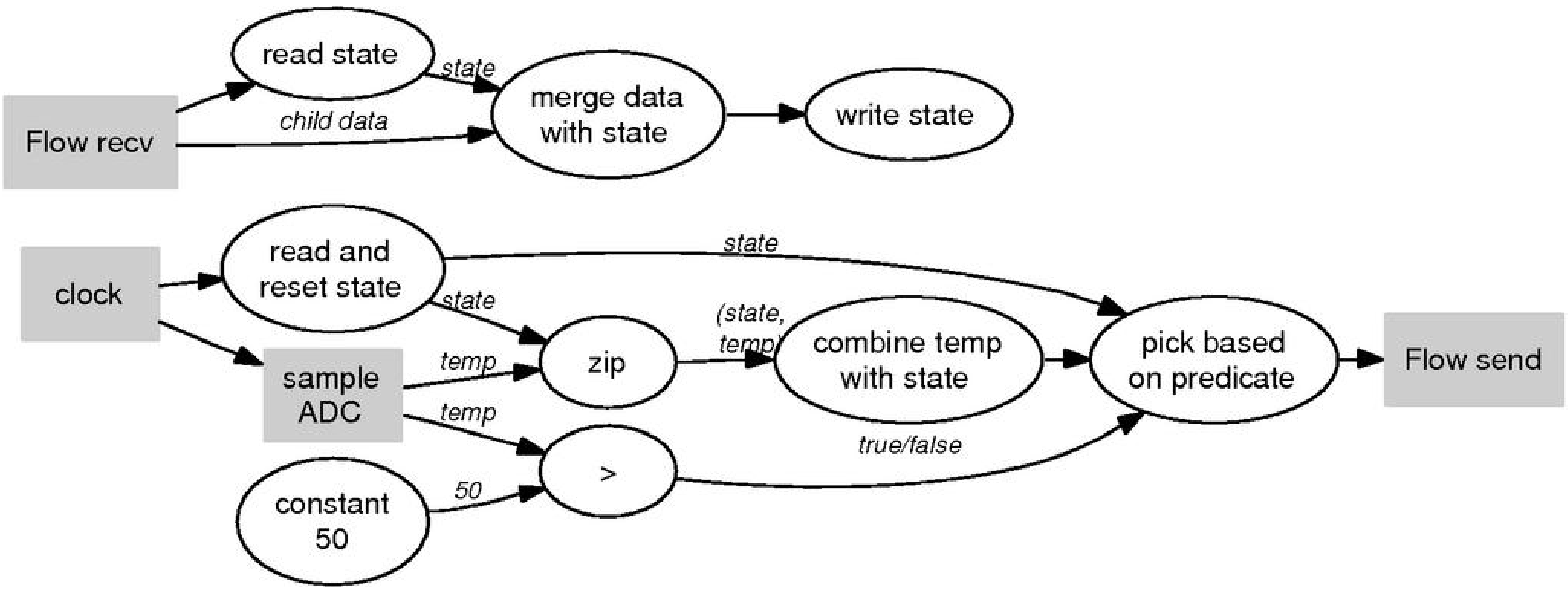
FlaskDB: SQL-to-NesC compiler in Flask

SQL query compiled into lean NesC binary

- Eliminates need for general query interpreter
- FlaskDB translates SQL statement into DFG which is then compiled to NesC

```
SELECT AVERAGE(temp) WHERE temp > 50 PERIOD 10 s
```

Resulting dataflow graph:



Regiment: A Stream-Oriented Macroprogramming Language

Abstract the sensor network as a collection of time-varying **streams**

- Streams represent node state, sensor values, timers, etc.
- Can map a function on a stream or filter out values of a stream

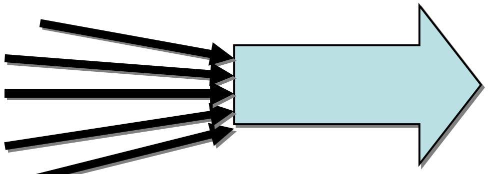
Collections of streams represented as **regions**

- A region represents a group of nodes – e.g., “all temperature sensors”, “all nodes within 100 m of point (x,y)”
- Can filter out members of a region, apply function to each stream in a region, or **fold** a region, aggregating it into a single stream.
- Regions can also be **nested** – e.g., “all one-hop neighbors of nodes in region R”

A few simple primitives allow us to write very sophisticated distributed programs!

- Complex spatial programming represented in a few lines of code

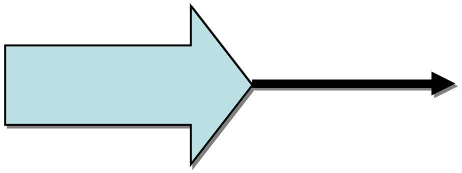
Some Region Primitives



```
everywhere : α stream → α region
gossip     : int → α stream → α region

map : (α → β) exp → α region → β region
filter : (α → bool) exp → α region → α region

fold   : (α *β → β) exp * β exp →
              α region → β stream
funnel : α region → α stream
```

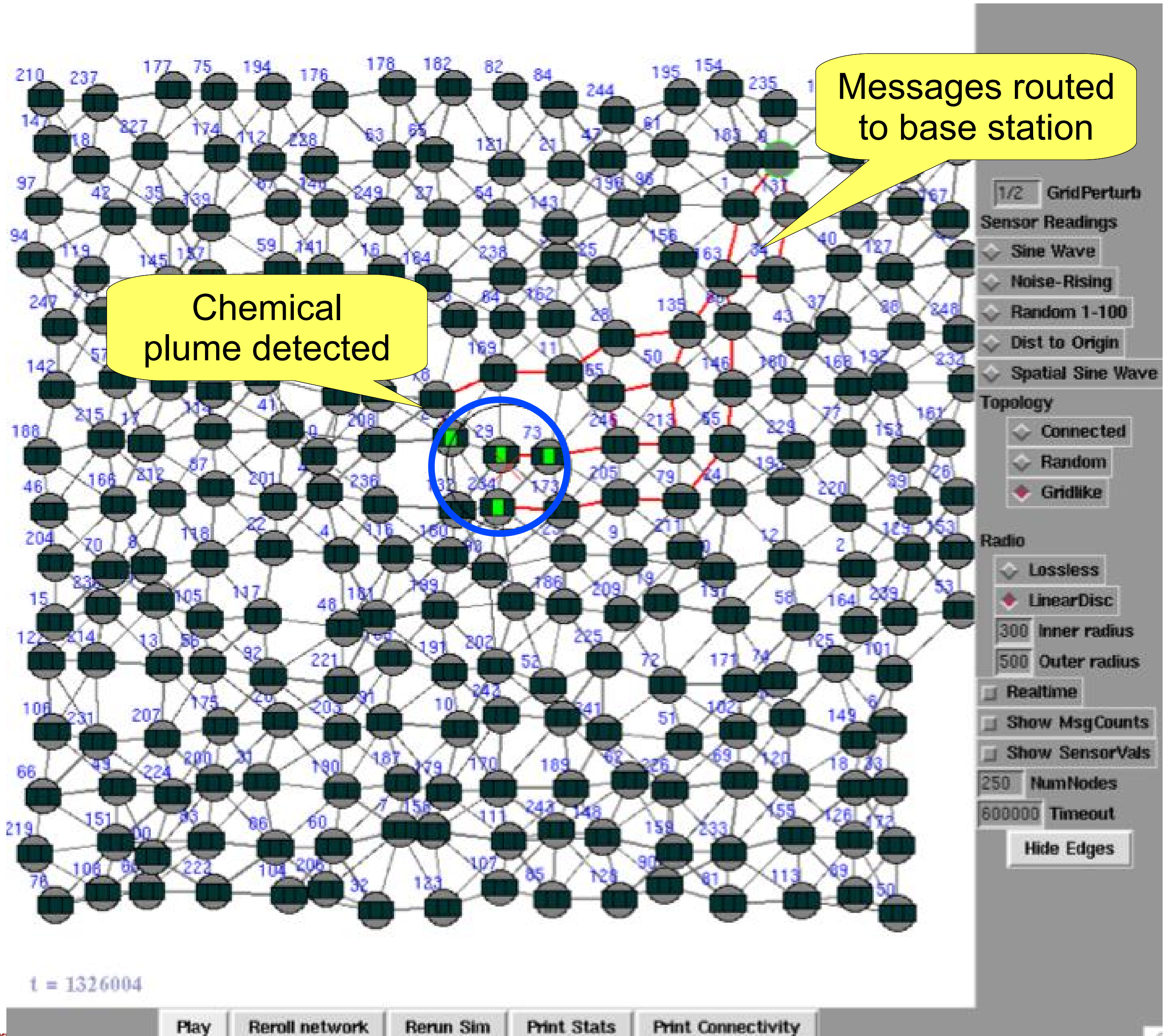


Regiment Example: Chemical plume "Hotspot" Detection

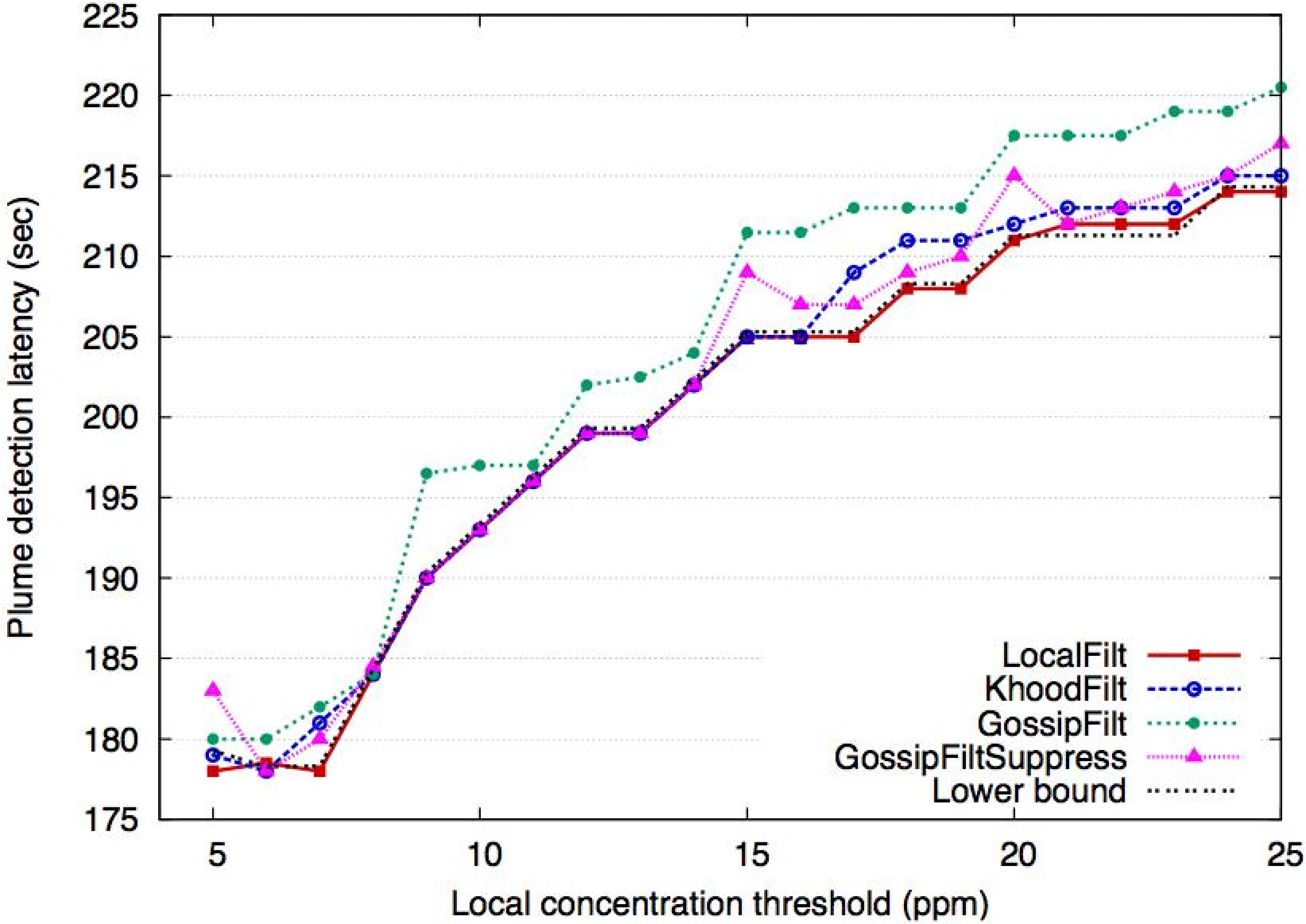
```
let hot_spot : concentration position =
  let cs = filter (λ c . c < THRESH) concentration
      vote = map (λ _ . 1) cs
      neighbor_votes = gossip 1 vote
      tally = fold (+) 0 neighbor_votes
      pt = zip position tally
      above = filter (λ (p,c) . c > COUNT) pt
  in
    map (λ (p,c) . p) above

let conc_stream = map (λ n . get_conc(n)) world in
let pos_stream = map (λ n . get_pos(n)) world in
  hot_spot (conc_stream pos_stream)
```

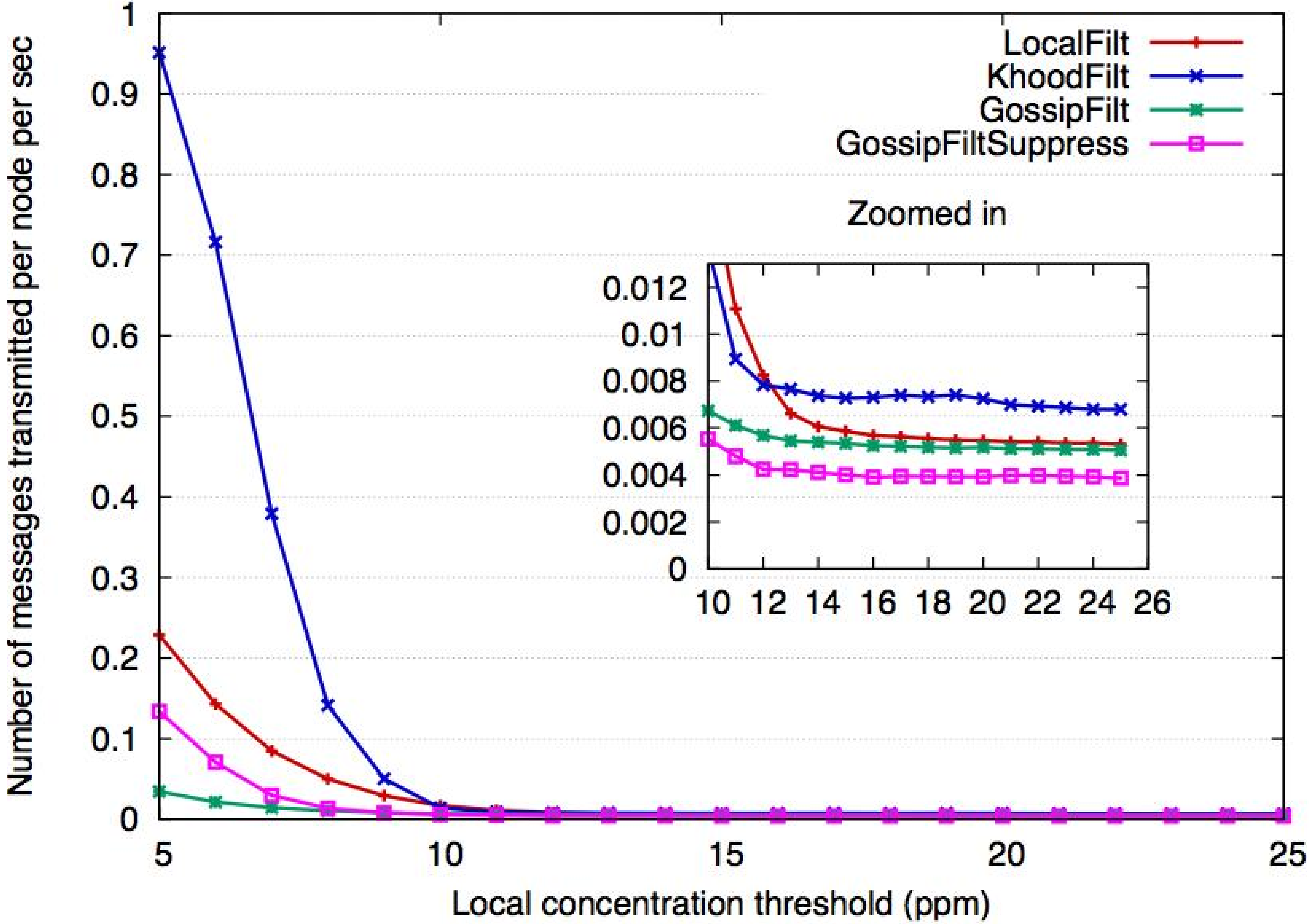
Regiment Simulation Environment



Plume detection latency



Communication overhead



Open research problems

What language designs are appropriate for each app domain?

How to efficiently compile into distributed dataflow representation?

How does high-level language design impact low-level system interfaces?

How much detail do we provide the programmer about the network's operation?

When abstracting away details, how do we avoid imposing high overhead or losing opportunities for optimization?

Conclusions and Future Directions

Sensor networks have tremendous potential for data-intensive science, but need far better programming models to be effective.

Fiji is one step in this direction:

- Node-level OS for resource adaptation
- Network-wide dataflow graph programming model as intermediate abstraction
- Support for multiple high-level languages for application development
- Flask metaprogramming toolkit for composing and compiling dataflow graphs

Next steps...

- More work on OS and runtime to manage resources: current focus on bandwidth management for body sensor nets
- Linux-based runtime and Flask target compiler for non-mote platforms

Thank you!

Flask Implementation

Flask wiring program is first converted into an internal dataflow graph representation

- Inlined NesC operators preprocessed into AST (to support typechecking)
- Inlined Hump code statically monomorphized at each call site, then translated to NesC
- Inputs and outputs of all operators are typechecked
- Asynchronous operations (e.g., ADC sampling) split into two operators: one to initate request, second for continuation

Dataflow graph then rendered as a single NesC component

- Operators mapped to individual NesC functions
- Operator composition implemented using direct function calls (which gcc will inline)
- All operators executed within TinyOS tasks to ensure atomicity
- Extremely low overhead compared to hand-coded NesC

MoteLab testbed experiments

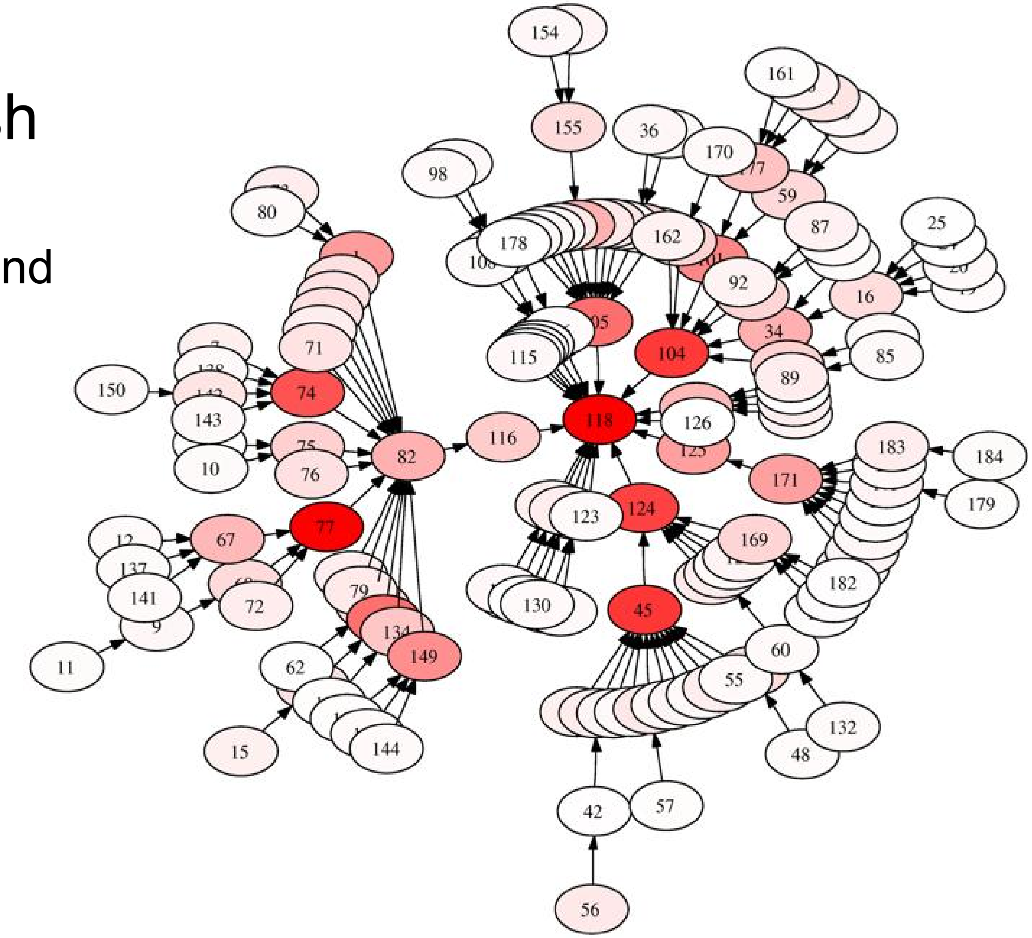
MoteLab: 190 node sensor network testbed at Harvard

- Deployed over three floors of the EECS building
- Tmote Sky nodes attached to Ethernet bridge; backchannel used for programming and data collection

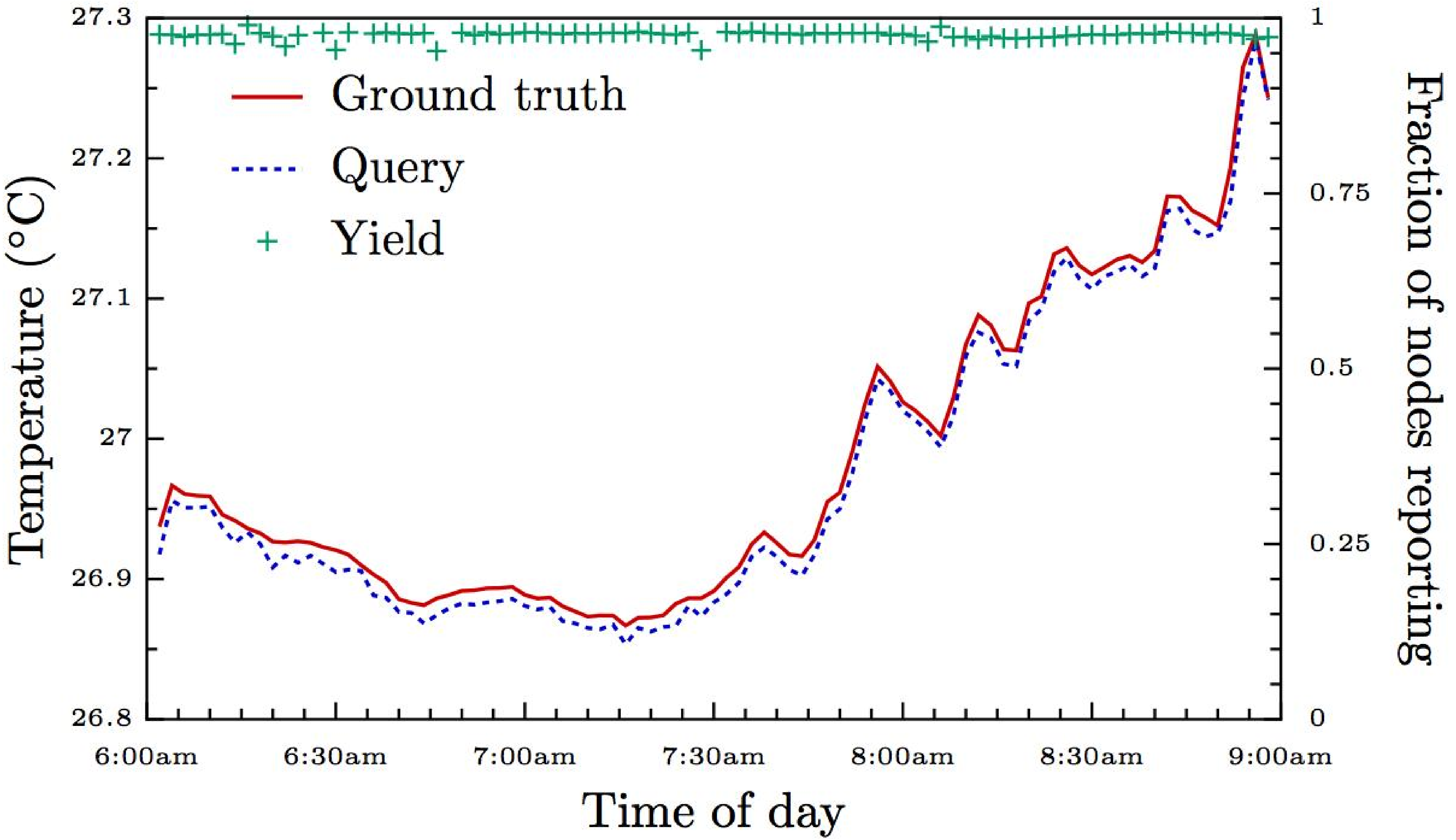
Can use backchannel to establish ground truth

- Complete sensor data logged to backend database, compare to data collected via wireless

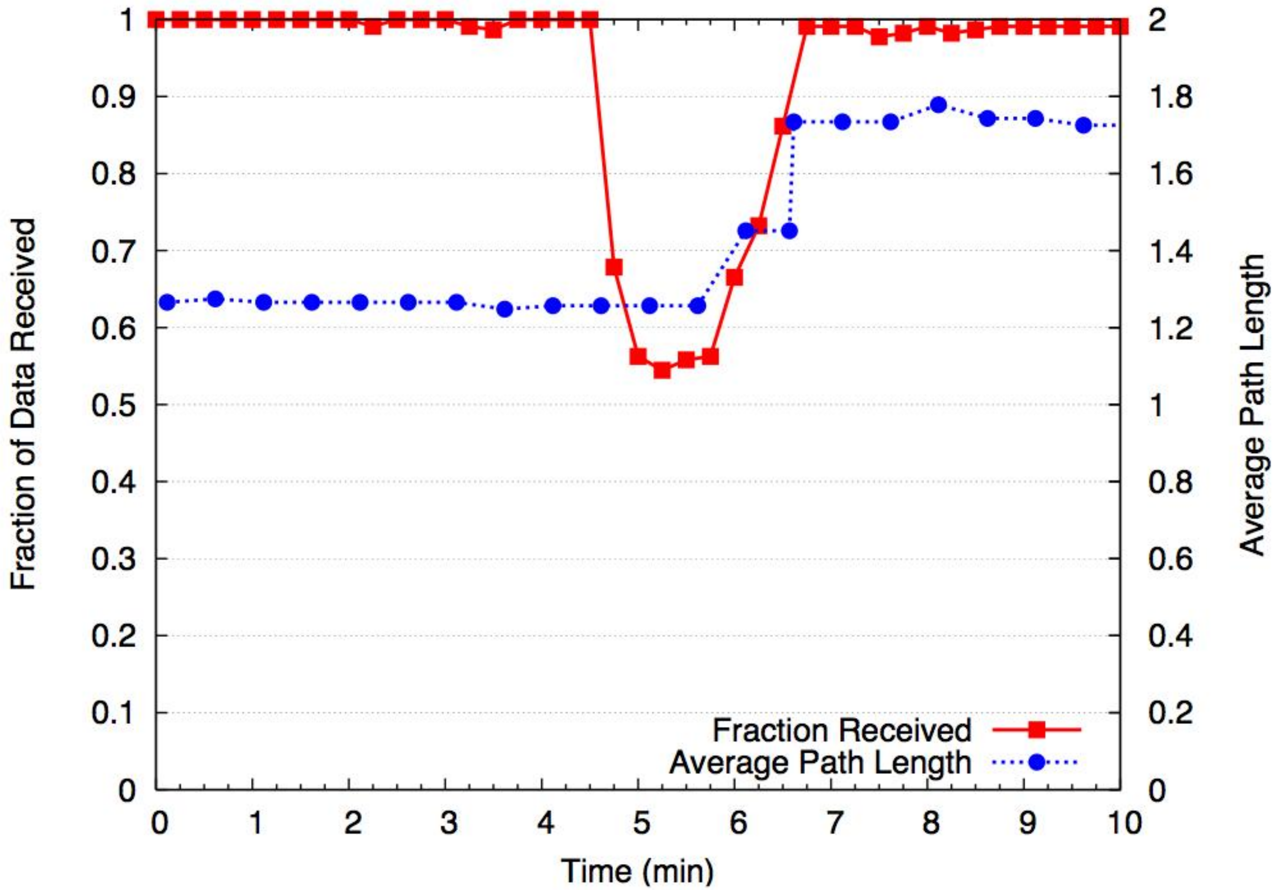
Measure robustness and scalability of Fiji communication layer (Flows protocol)



SQL query of average building temperature



Network robustness experiment



115 nodes publishing data to 10 anycast sinks

- Kill 5 sinks at t = 5 min
- Takes ~2 minutes for network to recover