Resource-Aware Programming in the **Pixie** Operating System

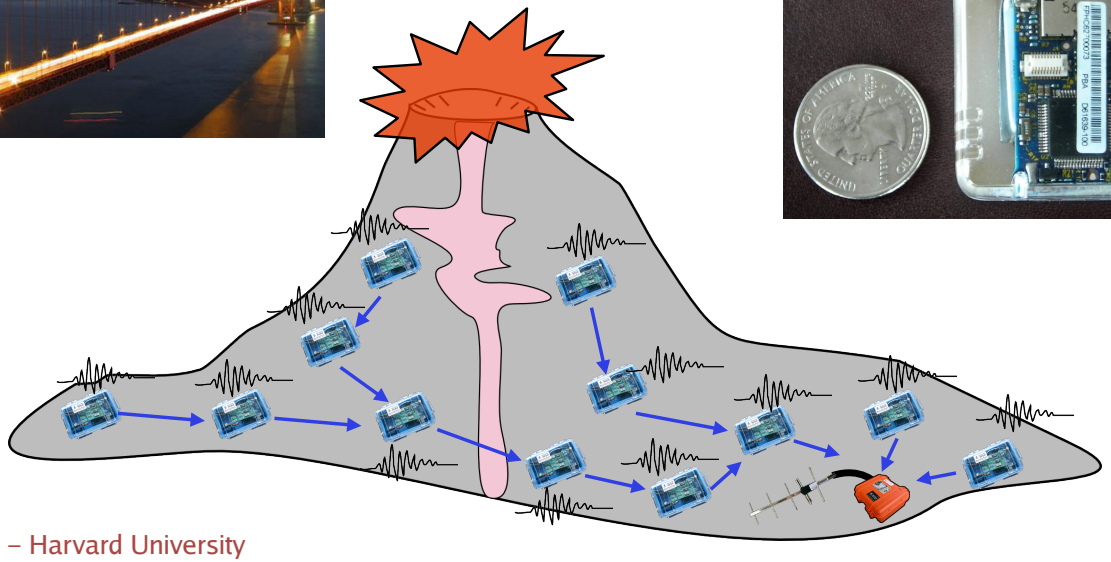
Konrad Lorincz, Bor-rong Chen, Jason Waterman,
Geoff Werner-Allen, and Matt Welsh

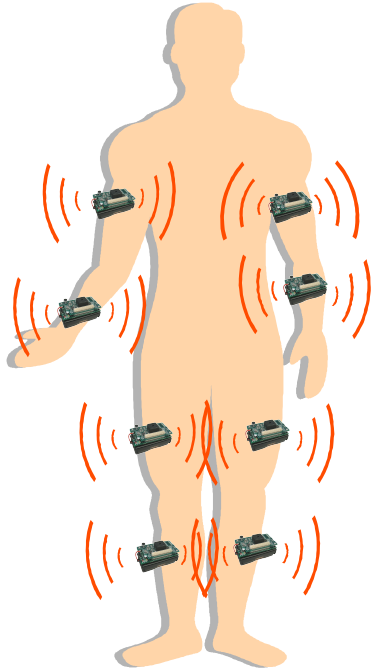
Harvard
School of Engineering
and Applied Sciences

New Challenges and Applications

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





New Challenges and Applications

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

Need to squeeze the most out of very limited resources!

- Maximize data fidelity
- Variation in load to to external stimuli
- Fluctuations in resource availability

The problem

Existing sensor node OS's don't provide much help in terms of resource management

- TinyOS provides mainly low-level access to hardware state
- Other systems mask too many details from application

TinyOS:
Too many knobs!

Can we span this divide?

Eon and Levels:
Not enough!

Our Approach: Pixie

Pixie is a new operating system for sensor networks that supports:

- Resources as a first-class programming primitive
- A **resource aware programming model** based on **resource tickets** and **brokers**

Pixie is intended to make it easy to write adaptive applications

- With such limited sensor node resources, application **must** contend with varying resource conditions!

Fundamental challenge:

- How to enable resource awareness without placing undue burden on the programmer?

Outline

Motivation and application example.

Pixie OS Architecture.

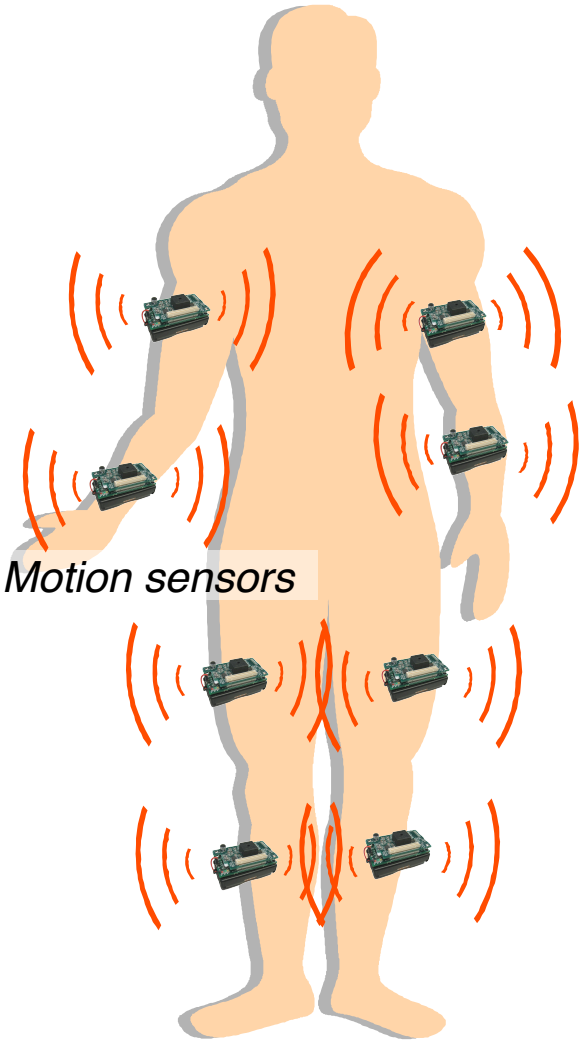
Programming model: Resource tickets and brokers.

Evaluation: Adaptation to bandwidth and energy variations.

Related work and conclusions.

Example application:
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
- Each node stores raw signal to 2GB microSD flash

Nodes perform local feature extraction

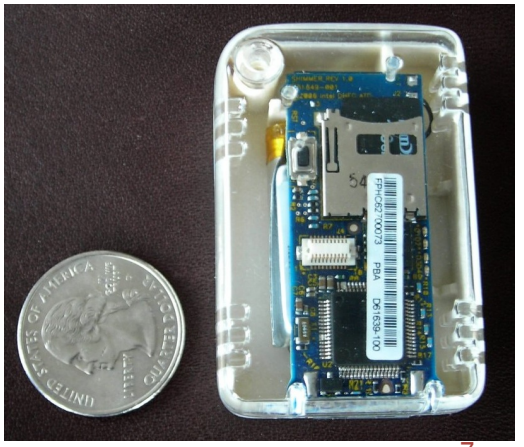
- RMS, jerk, dominant frequency, other features...
- Computationally intensive processing
- Transmit to laptop base station in home

Offline classification to map features to clinical scores

- UPDRS clinical scale

Earlier version being used to collect data on PD patients at Spaulding Rehabilitation Hospital, Boston

SHIMMER mote



The need for adaptivity

Nodes experience fluctuations in **load**

- Based on the signal collected by each sensor
- e.g., Can disable data logging when sensor not moving

Nodes experience fluctuations in **resource availability**

- Bandwidth: Radio link quality to base station varies as patient moves
- Energy: Available energy depends on node activity and communication

These variations cannot be predicted statically.

Application should adapt its behavior to changing conditions

- e.g., Scale type and quantity of data transmitted to available bandwidth
- Scale processing overhead based on battery lifetime

The Pixie Operating System

Pixie OS design goals

- Resources as first-class entity
- Direct application knowledge of available resources
- Fine-grained control over resource consumption

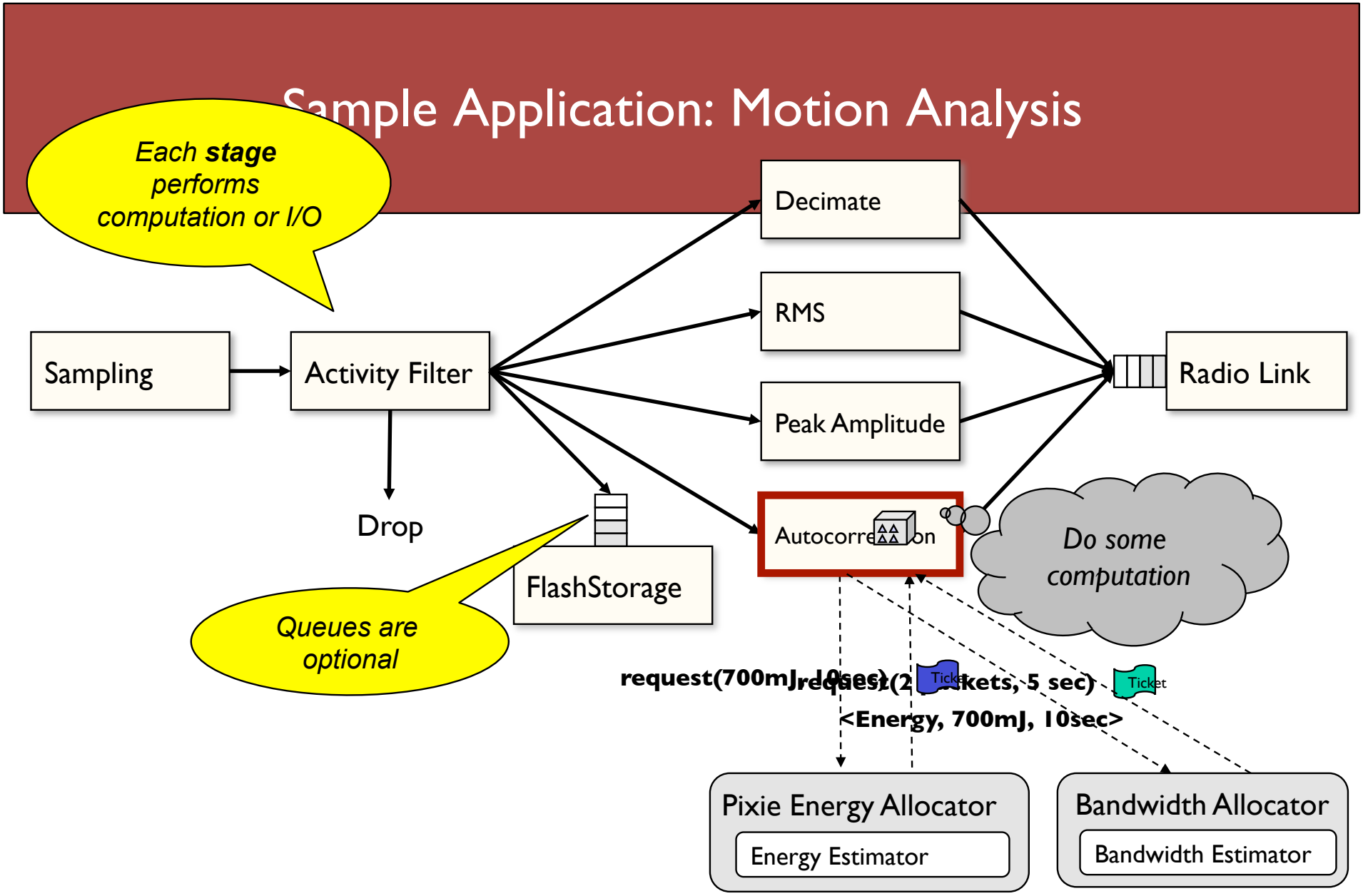
Key design features:

- **Dataflow programming model**: Application structured as a graph of **stages**
- **Resource tickets**: Primitive to represent fine-grained resource reservations
- **Resource brokers**: Implement policies to manage resources on behalf of the app

Prototype implemented in NesC

- Supports limited backwards compatibility with TinyOS

Sample Application: Motion Analysis



Resource Tickets

Core abstraction for resource management in Pixie

- Ticket **<R, c, t_e>** represents right to consume **c** units of resource **R** until the expiry time **t_e**.
- Think of as a short-term "reservation" for some resource.
- Tickets decouple resource request from usage – permits planning.

Basic resource ticket operations:

- Redeem: Performed by app when ticket is used
- Forfeit: Application gives up the ticket
- Revoke: Resource allocator reclaims resources (can happen prior to expiry time!)
- Join: Combine two tickets into one
- Split: Split a ticket into two separate tickets

Granularity depends on resource variability

- e.g., Bandwidth tickets would have a short expiry time; storage tickets need not expire.

Resource Allocators

Each physical resource has a corresponding **allocator**

- Allocators for energy, memory, flash storage, and radio
- Implicit allocator for CPU – the scheduler

Allocators estimate available resource, and allocate tickets.

- **No policy**: Always allocate ticket if the resources are available
- Policies handled by higher-level abstractions (e.g., brokers)

Allocators also enforce ticket redemption

- e.g., To transmit a packet must have corresponding bandwidth ticket

Resource Estimation

Storage and memory are straightforward

- Allocators track total flash/memory consumption

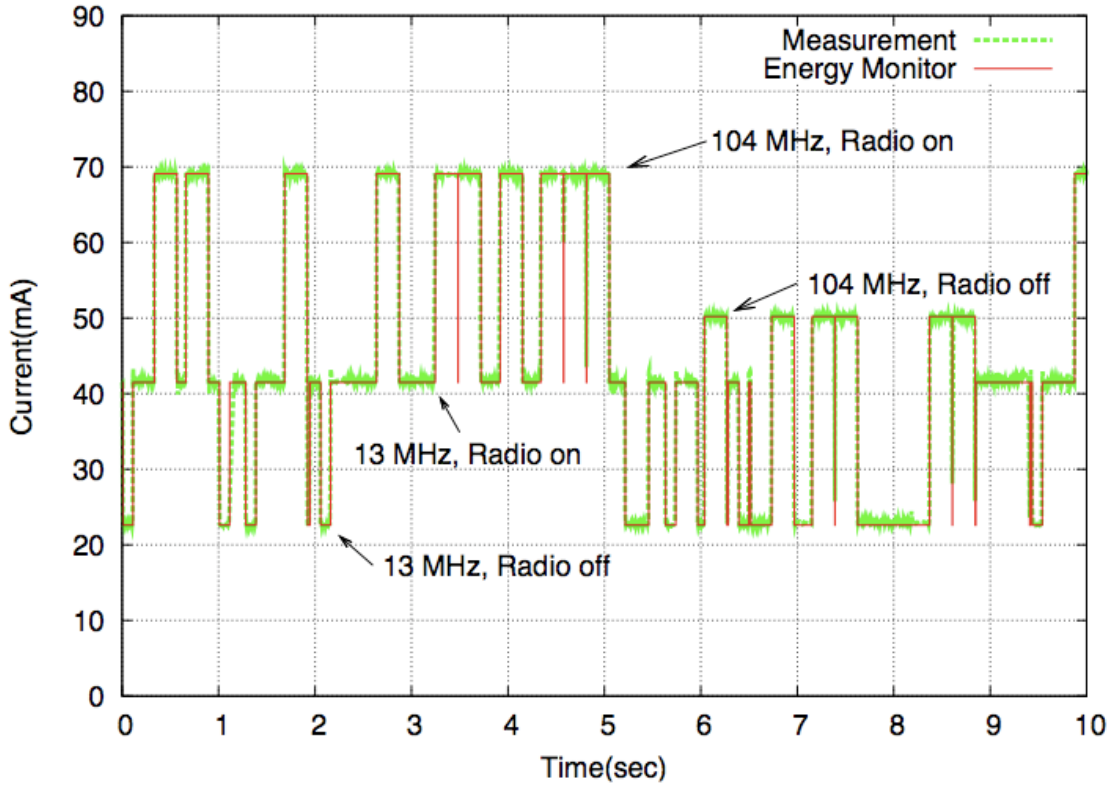
Bandwidth: Radio link estimates packet transmission delay

- Measure total time for packet transmission plus ARQ (if used)
- Invert transmission delay to estimate instantaneous bandwidth
- Maintain separate estimate for each neighbor
- Currently assumes delay is mostly independent of packet size

Software energy metering

Energy estimation performed in software

- Tracks state of each hardware device (CPU, radio, flash, sensors) in real time
- Accurate empirical model of hardware energy consumption (about 2-3% error)
- Low overhead, no hardware support required.



Resource Brokers

Resource tickets are intended to be low-level and fine-grained

- Very flexible and powerful, but sometimes difficult to use

To simplify app design, we introduce **resource brokers**

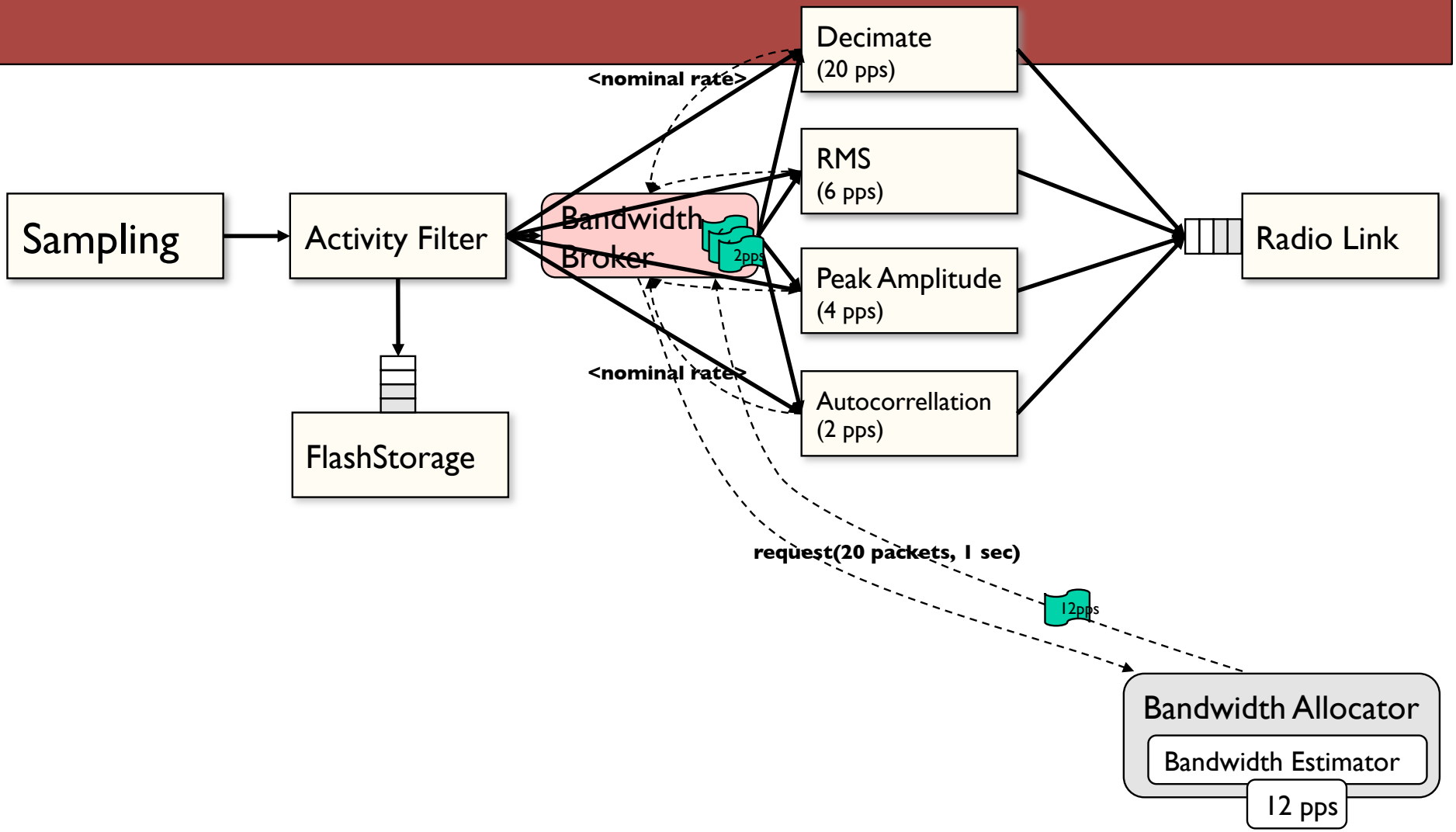
- Brokers request and manage resource tickets on behalf of the application
- Each broker implements some policy and provides a high-level API to the app

Brokers are specialized stages

- Can interpose directly on the application's dataflow path
- Can inspect, redirect, discard, or refactor data

Pixie provides a small library of standard brokers

Bandwidth Broker



The Pixie Energy Broker

Pixie's **energy broker** attempts to dole out energy tickets at a rate to achieve a given target node lifetime.

- App stages specify energy *quanta* and *priority*
- Broker delivers energy tickets to stages according to priority and energy schedule

Energy scheduling

- Depletion rate ρ calculated based on battery capacity C and lifetime target λ: $\rho = C/\lambda$
- Broker calculates *energy surplus* Δ based on available energy $e(t)$ and scheduled (target) availability $\hat{e}(t)$: $\Delta = e(t) - \hat{e}(t)$
- If $\Delta \leq 0$ the system is in energy *debt*

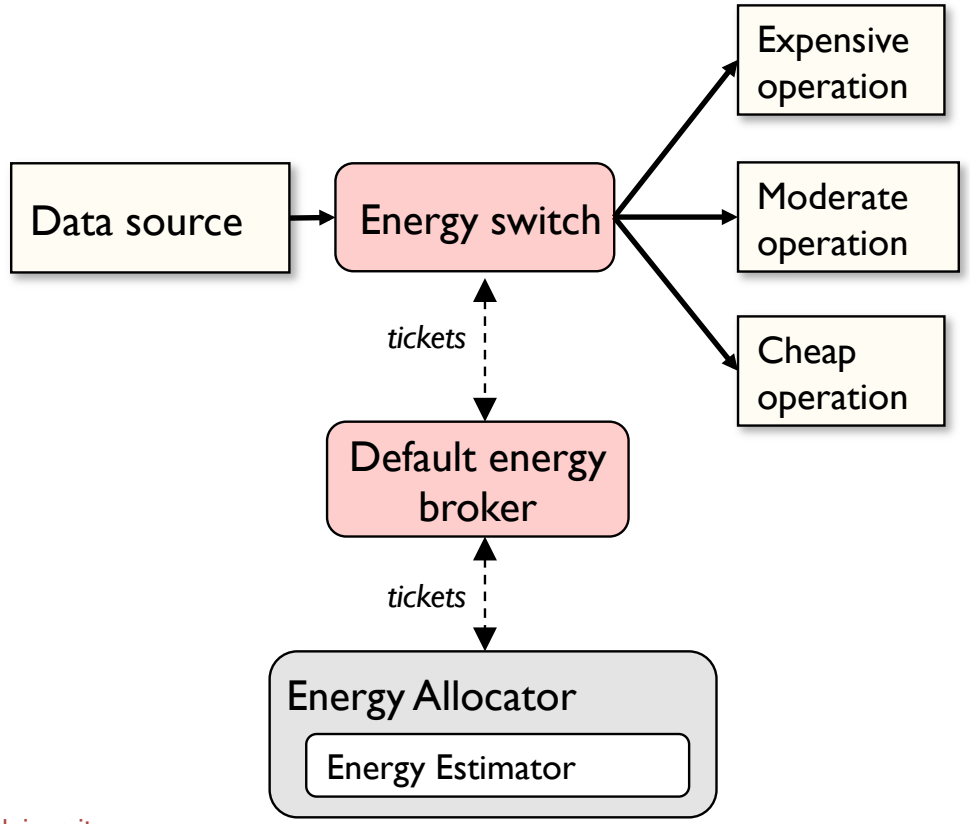
Broker policies

- *Conservative*: Only allocate tickets if $\Delta > 0$ – strictly adheres to energy schedule
- *Credit-based*: Allow accumulation of debt up to $\Delta = -\alpha$, then require payback until $\Delta > 0$

Other Energy Brokers

Energy-aware switch

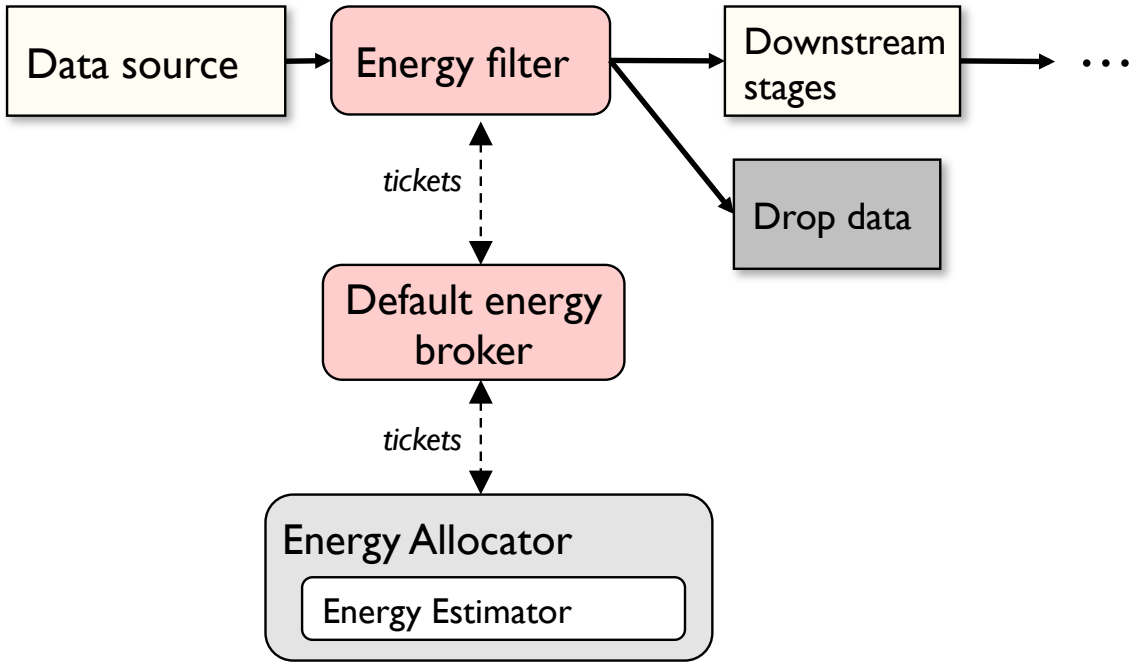
- Sends data down one or more downstream paths based on energy availability
- Mimics policy used by Eon *[Sorber et al., Sensys 2007]*



Other Energy Brokers

Energy-aware filter

- Selectively drops data to meet energy target
- Effectively controls scheduling and duty-cycling of downstream stages



Programming Benefits

Dataflow programming model maps well onto application structure

- Similar to Eon, Tenet, WaveScope, and Flask

Tickets offer a great deal of flexibility and power

- Fine-grained feedback and control of resource usage at all levels

Brokers decouple resource management policies from mechanisms

- Enables composable resource adaptations

Resource-awareness is pervasive in the programming model

- Not “off to the side”
- Annoying at first, but makes sense when you get used to programming in this way
- Exposes resource dependencies, rather than relying on hidden magic knobs

Evaluation Summary

Microbenchmarks

- Pixie's overhead is low compared to TinyOS

Accuracy of resource estimation

- Pixie accurately estimates available resources

Bandwidth adaptivity

- Pixie enables rapid adaptation to changing radio link characteristics

Energy adaptivity

- Pixie applications degrade gracefully as lifetime target is increased

Combined bandwidth and energy adaptivity

- Can jointly optimize app behavior based on multiple resource constraints

Evaluation Summary

Microbenchmarks

- Pixie's overhead is low compared to TinyOS

Accuracy of resource estimation

- Pixie accurately estimates available resources

Bandwidth adaptivity

- Pixie enables rapid adaptation to changing radio link characteristics

Energy adaptivity

- Pixie applications degrade gracefully as lifetime target is increased

Combined bandwidth and energy adaptivity

- Can jointly optimize app behavior based on multiple resource constraints

Bandwidth Adaptation in the Motion Analysis Application

Goal: Adapt type and quantity of data transmitted by each node as link quality varies

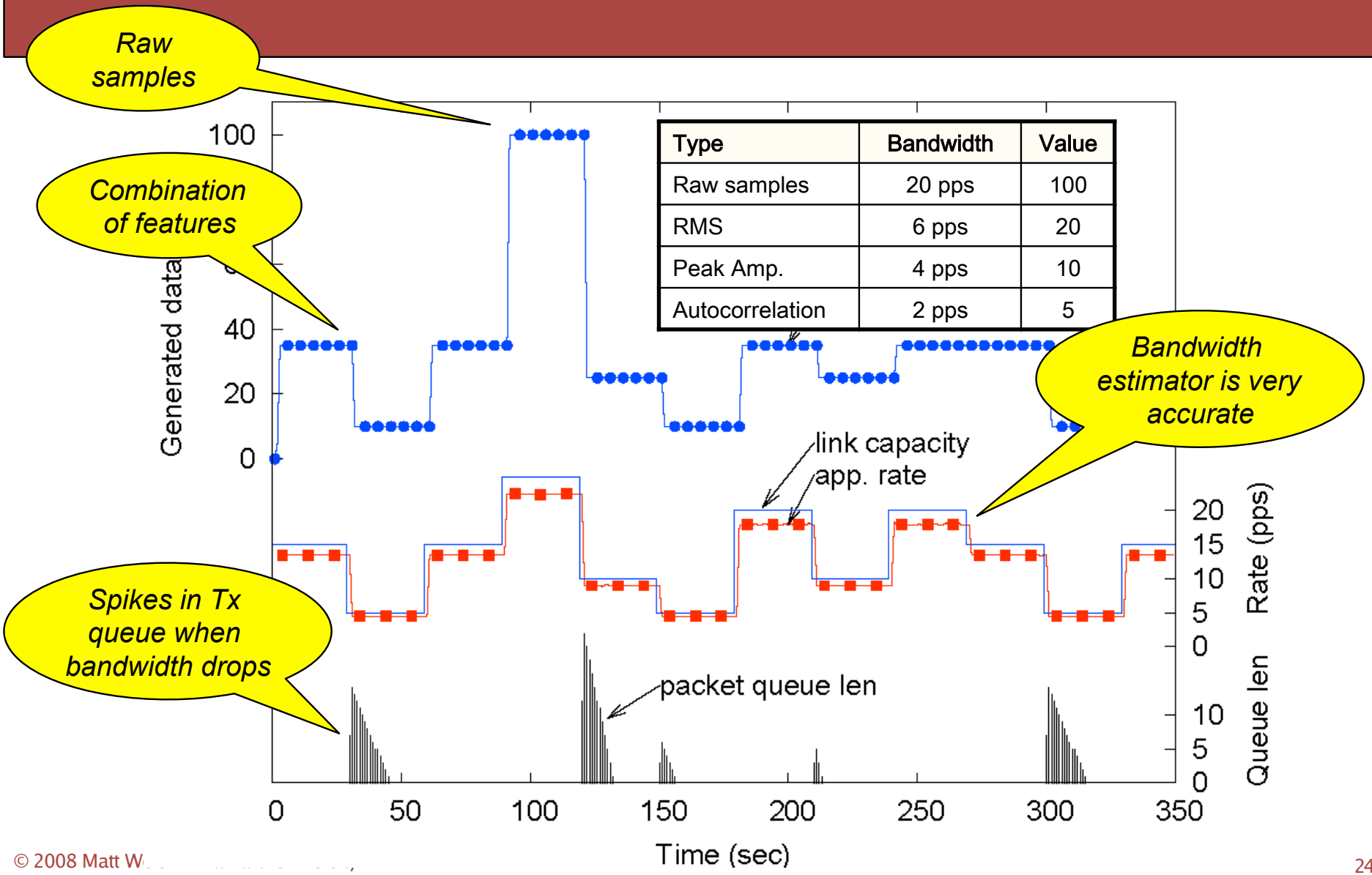
- Don't want to transmit data that won't get to the base station.

Uses Pixie's bandwidth broker

- Multiple data types with associated **cost** and app-assigned **value**
- Estimate available link bandwidth
- Assign bandwidth tickets to maximize value subject to bandwidth limit

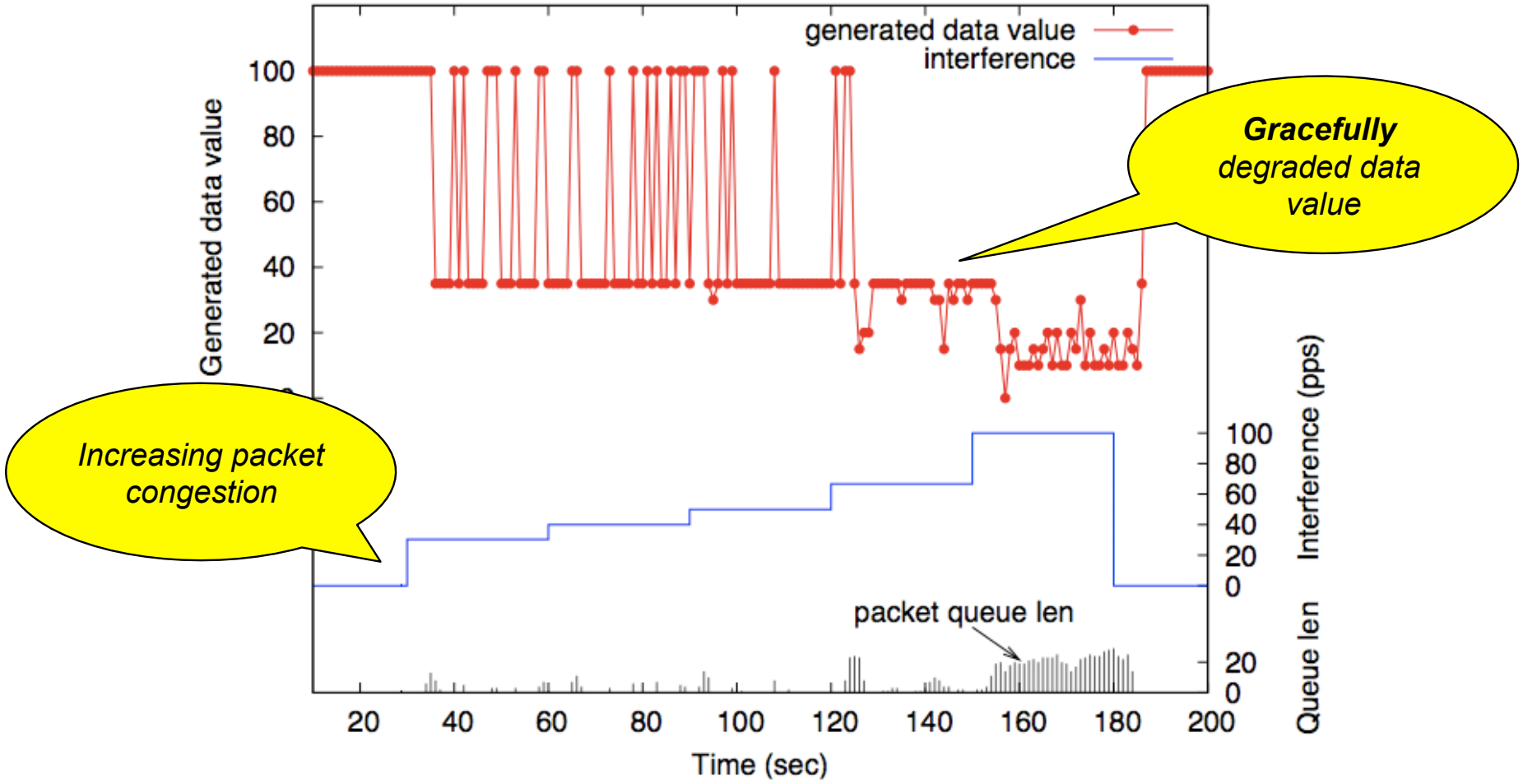
Type	Bandwidth	Value
Raw samples	20 pps	100
RMS	6 pps	20
Peak Amp.	4 pps	10
Autocorrelation	2 pps	5

Experiment: Bandwidth Adaptation



Type	Bandwidth	Value
Raw samples	20 pps	100
RMS	6 pps	20
Peak Amp.	4 pps	10
Autocorrelation	2 pps	5

Adaptation to radio channel interference



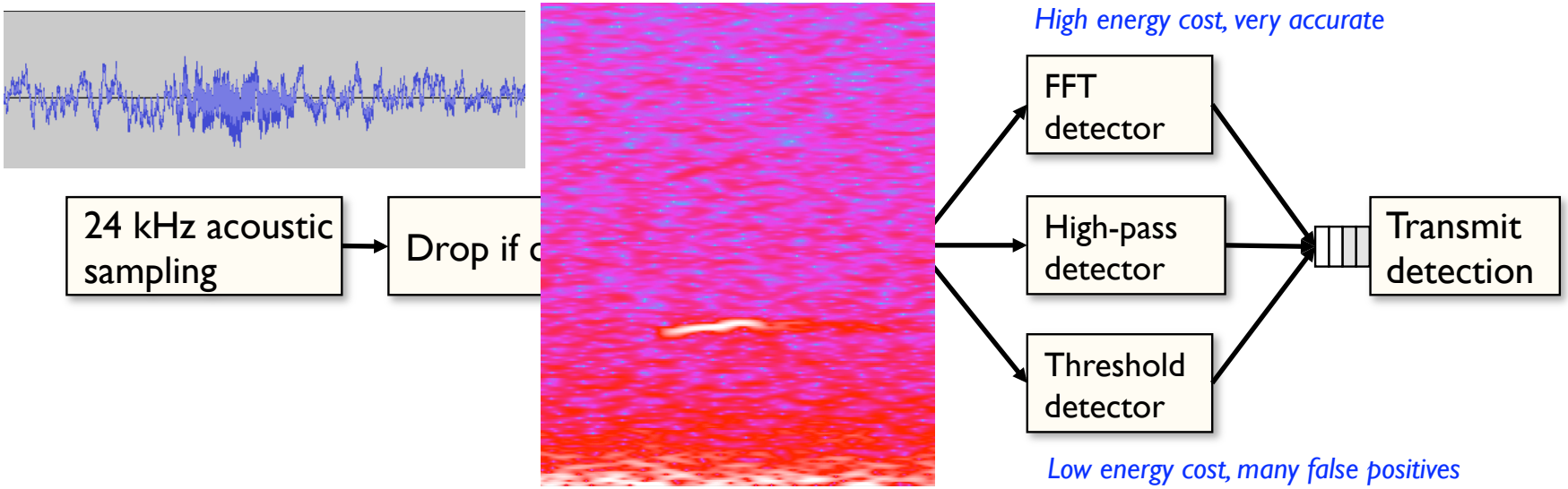
Application #2: Acoustic Target Detection

Detect acoustic signature of target (marmot call) in acoustic signal, subject to variable noise.

- Similar to ENSBox *[Girod et al. Sensys'06]* – but no ranging

Three detection algorithms with increasing energy cost and accuracy

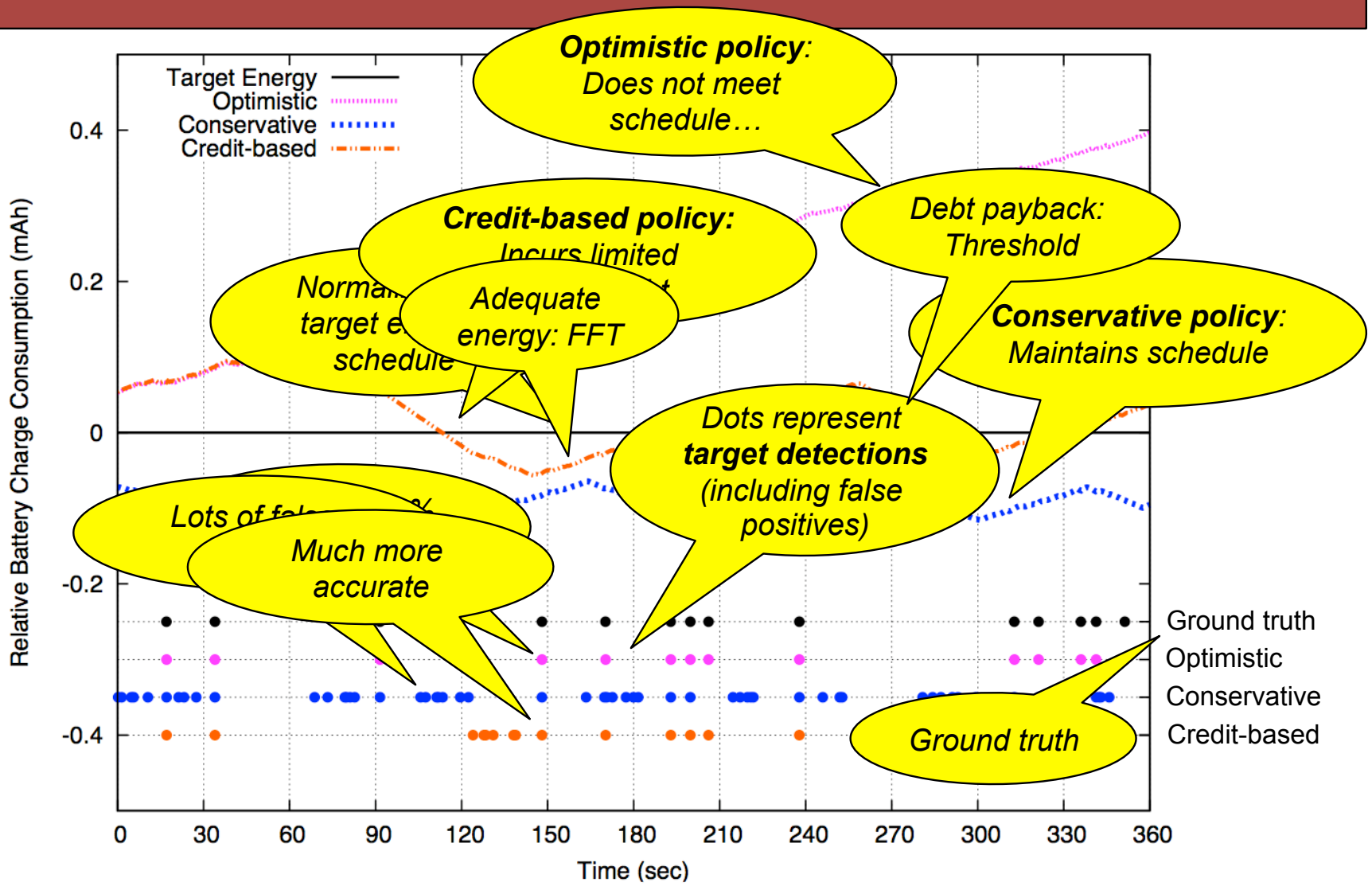
Goal: Maximize accuracy subject to battery lifetime target



* Thanks to Lew Girod for providing acoustic signal samples used in this study.

Energy adaptivity: Varying policies

40-day lifetime target



Related Work

Eon *[Sorber 2007]* and Levels *[Lachenmann 2007]*

- Focus on energy management
- Application not directly involved in adaptivity
- Pixie decouples resource management mechanisms (tickets) from policies (brokers)

Nano-RK *[Eswaran 2005]*

- Real-time OS for sensor nets, focuses on static resource reservations
- Does not address dynamic changes in load or resource availability at runtime

Odyssey *[Noble 1997]*, ECOsystem *[Zeng 2002]*, Puppeteer *[Lara 2001]*

- Designed for mobile systems with a focus on legacy OS and application support

Conclusions and Future Directions

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

Pixie is a new OS to facilitate **resource aware programming**:

- Staged dataflow programming model to simplify application design
- Resource tickets expose resource requests and grants from the system
- Resource brokers mediate between application code and the underlying OS
- Runtime estimation of CPU, radio bandwidth, energy, and storage availability

Next steps: Coordinated resource management across a network

- Extend Pixie ticket model across multiple nodes

For more information, papers, and code:
`http://fiji.eecs.harvard.edu/Pixie`

Thanks!

Backup slides follow

Estimating Resource Demands

To use Pixie, apps need to estimate their demand.

In many cases this is straightforward:

- Easy to estimate transmission, memory, and storage needs statically

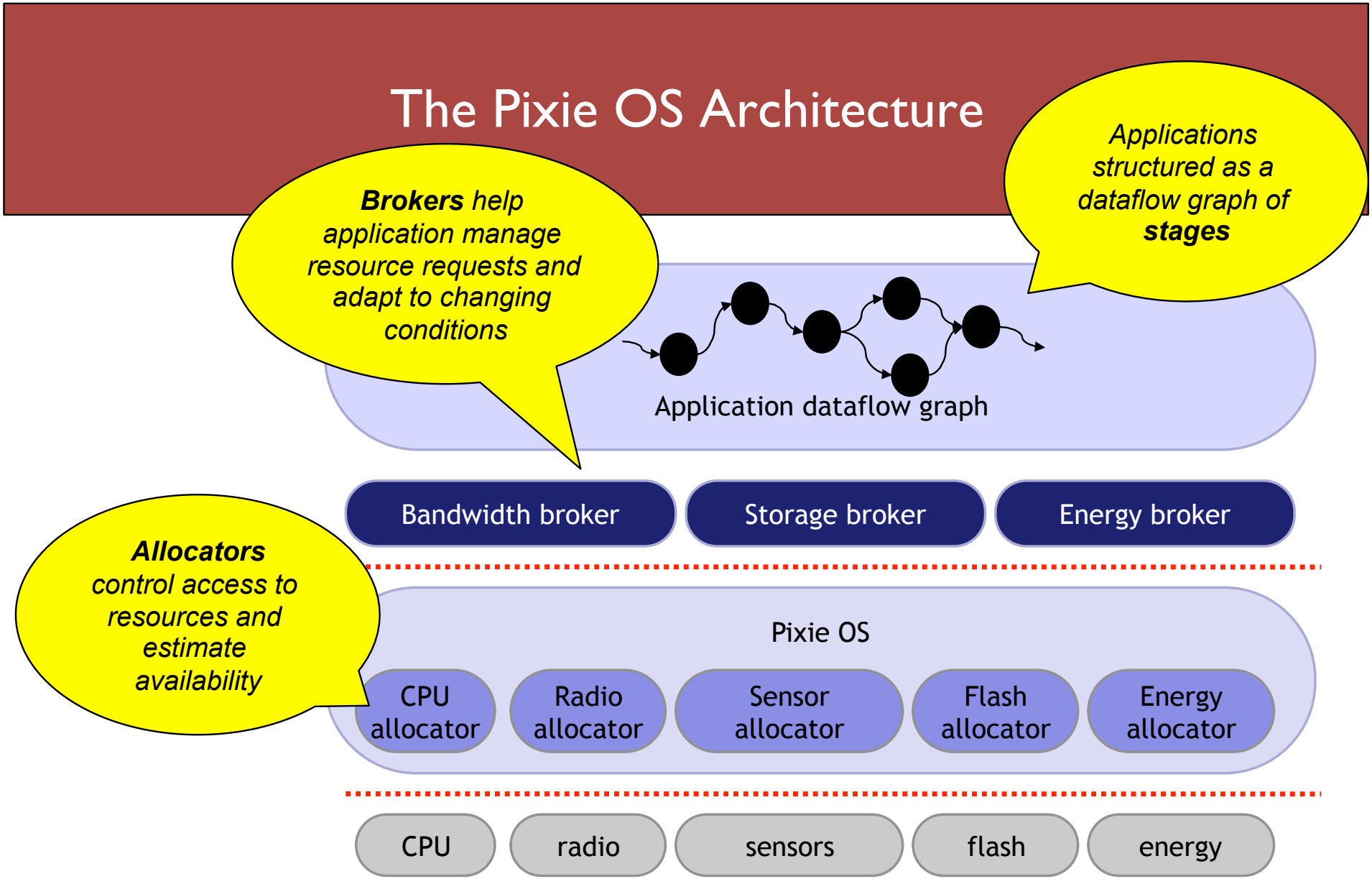
Energy estimation can be performed using a number of tools

- PowerTOSSIM, Quanto, iCount, etc.

Stages can also measure resource demands on the fly

- Software energy meter can be used to provide dynamic estimates

The Pixie OS Architecture



Pixie Design Philosophy

Existing sensor node OS's don't provide much help in terms of resource management

- TinyOS provides mainly low-level access to hardware state
- Other systems mask too many details from application

Pixie: A new OS to support **resource-aware programming**

- Key idea: Provide extensive feedback and control over resources to the app
- Permit system to *adapt* to resource fluctuations
- Fundamental shift in OS design!

We argue that sensor network applications *must* be adaptive in the face of extremely limited and variable resources.

Dataflow Programming Model

Pixie applications structured as a dataflow graph of **stages**

- Similar to operators in Click *[Kohler et al., ACM TOCS'00]*
- Stages have one or more input ports, zero or more output ports
- Edges between stages perform no queueing; can insert special queue stage if desired.

Stage scheduling:

- Depth-first traversal of all downstream stages from a given source stage
- Direct function call: Low overhead
- Stages can block, which stalls all upstream stages until reaching a source

Benefits over TinyOS component or process model:

- Maps well onto application structure (also used by Eon, Tenet, Flask, etc.)
- Dataflow explicitly represents resource and data dependencies within the app
- Naturally supports interpositioning

Implementation

Pixie is implemented in NesC

- Supports limited compatibility with legacy TinyOS code
- Must wrap TinyOS components as stages, and make them resource-aware
- Mostly we ~~*steal*~~ import low-level drivers from TinyOS

Core OS is 8755 lines of NesC code, including comments

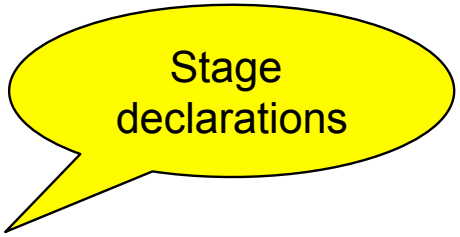
Download from `http://fiji.eecs.harvard.edu/Pixie`

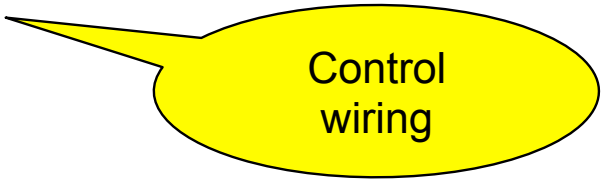
Example Pixie Application Wiring

```
configuration MyApp {
} implementation {
  components
    PixieCore,
    new PixieSamplingStage(RATE, CHANNELS),
    new SampleFilterStage() as Filter;
    new PixieStorageStage() as Storage;
    new PixieEnergySwitch() as Broker,
    new ProcessingStageA() as ProcA,
    new ProcessingStageB() as ProcB,
    new PixieSendStage(BASE_STATION_ID);

  /* Dataflow graph wiring */
  PixieSamplingStage.Output -> Filter.Input;
  Filter.Output -> ProcA.Input;
  Filter.Output -> ProcB.Input;
  Filter.Output -> Storage.Input;
  ProcA.Output -> SendStage.Input;
  ProcB.Output -> SendStage.Input;

  /* Control wiring between broker and stages */
  Broker.EnergyControl -> ProcA.EnergyControl;
  Broker.EnergyControl -> ProcB.EnergyControl;
  /* Interface to energy allocator */
  Broker.EnergyAlloc -> PixieCore.EnergyAlloc;
}
```





Microbenchmark Results

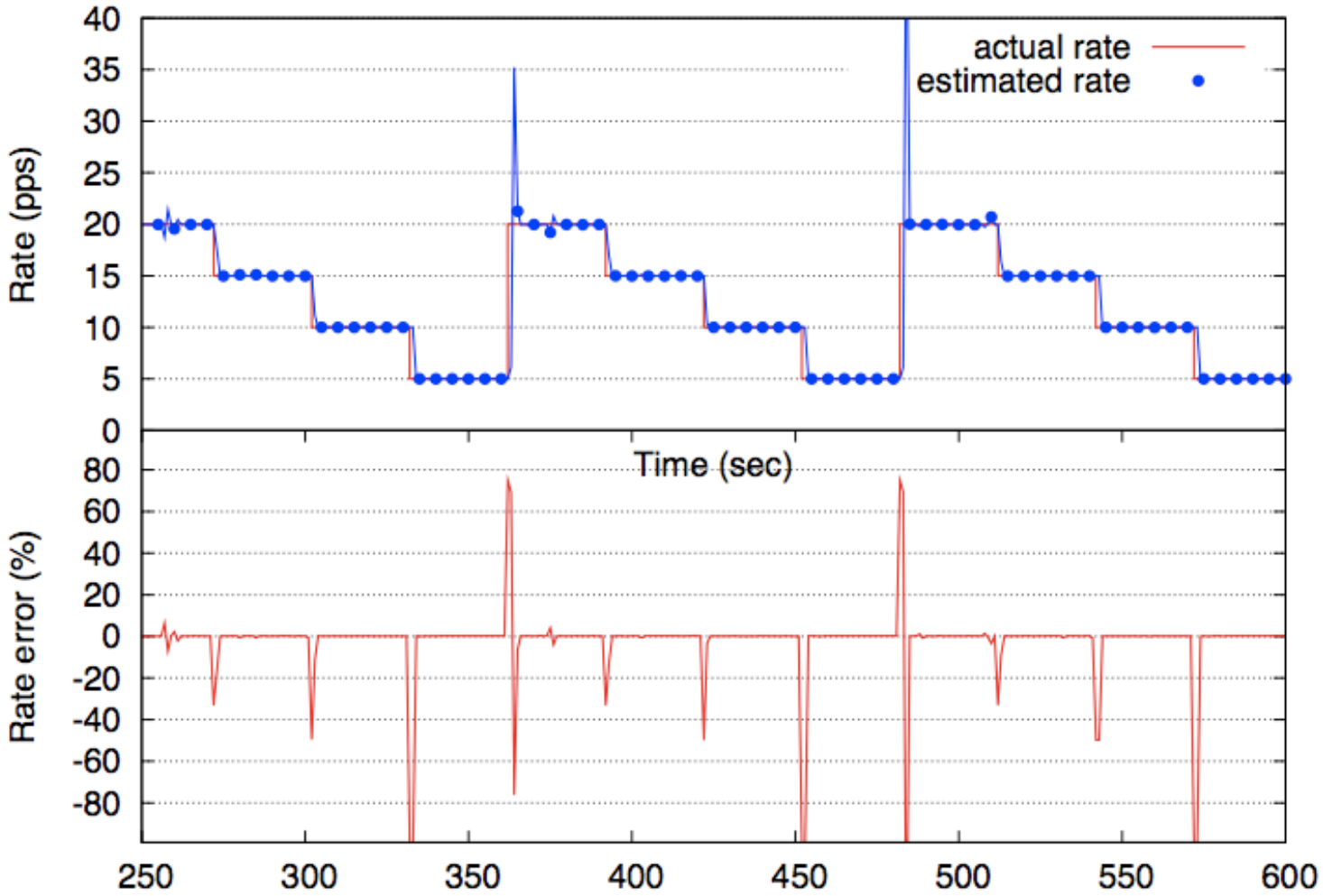
Overhead of Pixie stage traversal is low

- Direction function call between stages
- Takes advantage of NesC inlining
- 9 usec in Pixie, versus 3 usec in TinyOS
- Difference mostly due to memref reference counting

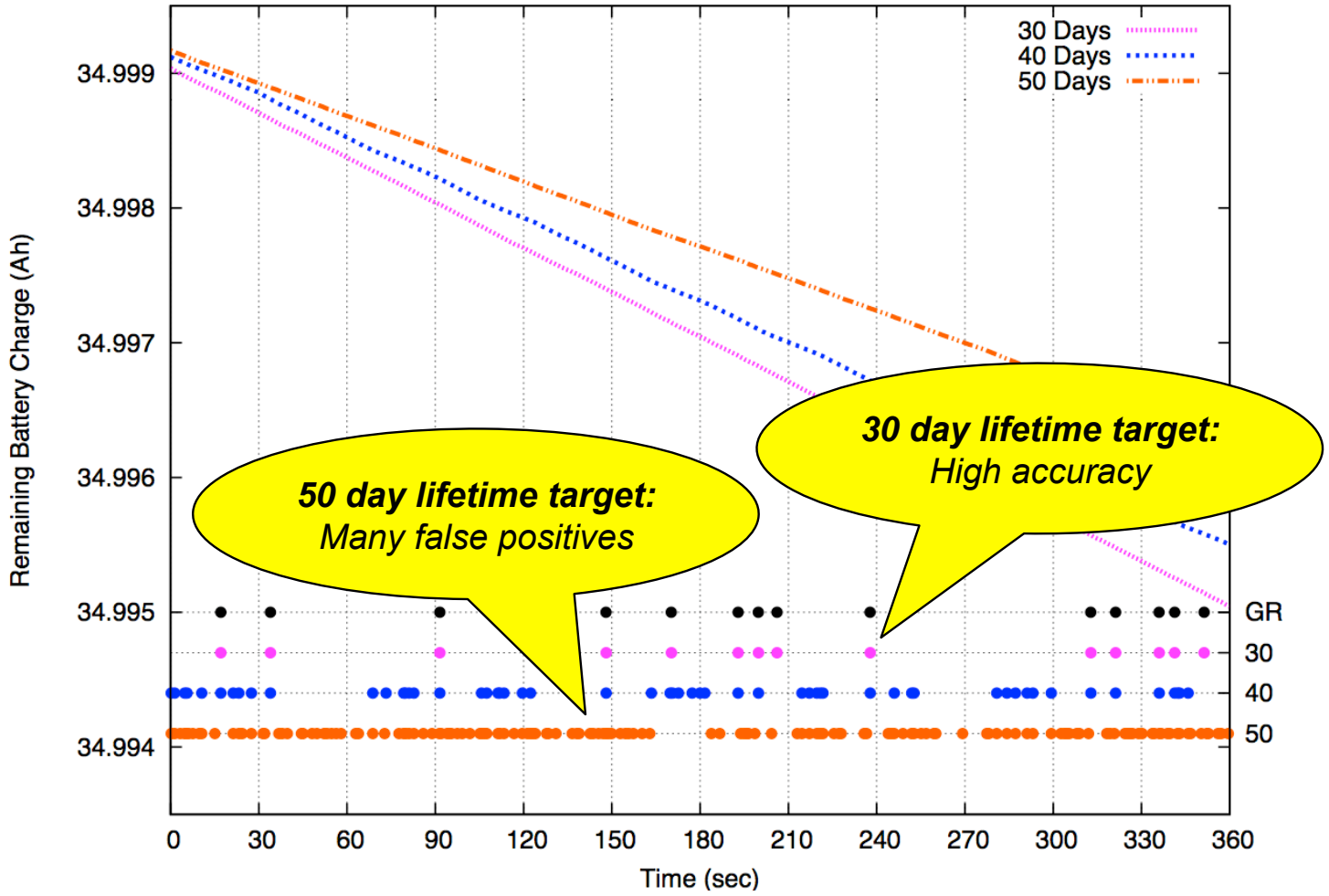
Memory footprint is modest

- Pixie scheduler and memory manager: 958 bytes ROM, 5825 bytes RAM
- Most of this is the statically allocated heap

Bandwidth Estimation Accuracy



Energy adaptivity: Varying lifetime target



Combined bandwidth and energy adaptation