# Lance:

## Priority-Driven Data Storage and Extraction for Sensor Networks

Geoff Werner-Allen and Matt Welsh

Harvard University

Stephen Dawson-Haggerty

UC Berkeley

Omar Marcillo and Jeff Johnson

UNH

# The Problem

Sensor networks increasingly used for *data intensive* applications:

- Seismic and acoustic monitoring of volcanoes, fault zones, bridges, and buildings
- Vibration monitoring of industrial equipment
- Acoustic monitoring of animal habitats

Data rates exceed limited bandwidth and storage resources:

- Typical sensor nodes exhibit ~ 100 Kbps radio bandwidth and ~ 1 MB flash
- Best reliable transfer protocol exhibits < 8 Kbps over multiple radio hops
- Yet, each sensor node may be sampling multiple channels at 100s to 1000s of Hz

Applications must direct limited resources to extracting the *"most interesting"* signals from the network

- Of course, "most interesting" depends on the app needs and the resources available

# The Solution: Lance

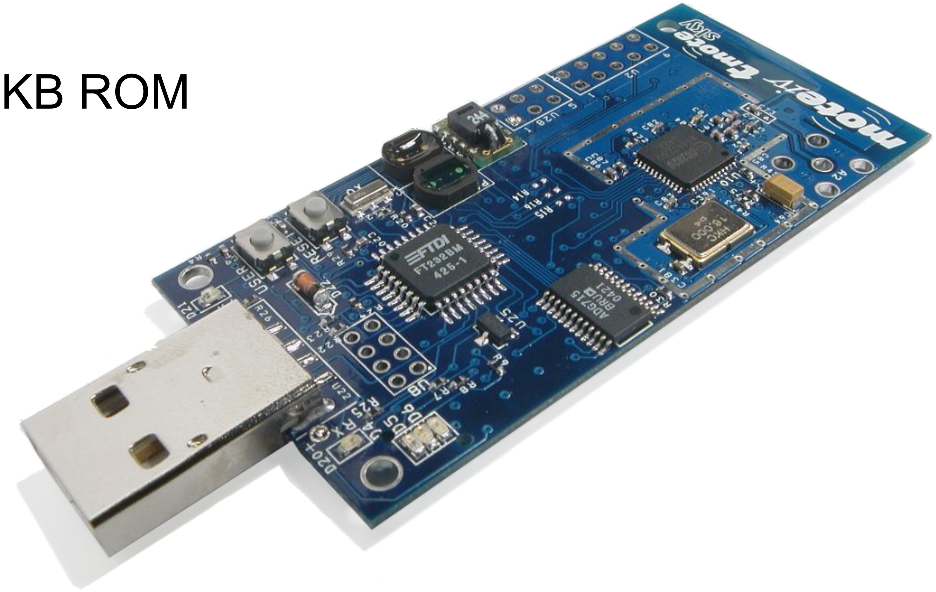
- System for priority driven allocation of bandwidth and storage resources within a sensor network
  - Sensor nodes assign **priority** to data based on application specific metric
  - Priority drives allocation of nodes' flash storage
  - Data **summaries** sent to base station, used for allocating radio bandwidth for data extraction
- Lance decouples mechanisms for resource allocation from app-specific policies
  - App-supplied **policy modules** permit a wide range of allocation policies to be implemented
  - Can target many optimization metrics: priority maximization, fairness, spatial/temporal data distribution, and more

# Talk Outline

- Motivation: Volcano monitoring
- The Lance Architecture
- Policy Modules: Application-Specific Customization in Lance
- Simulation study: Lance achieves *near-optimal* allocation efficiency
- Results from deployment at Tungurahua Volcano

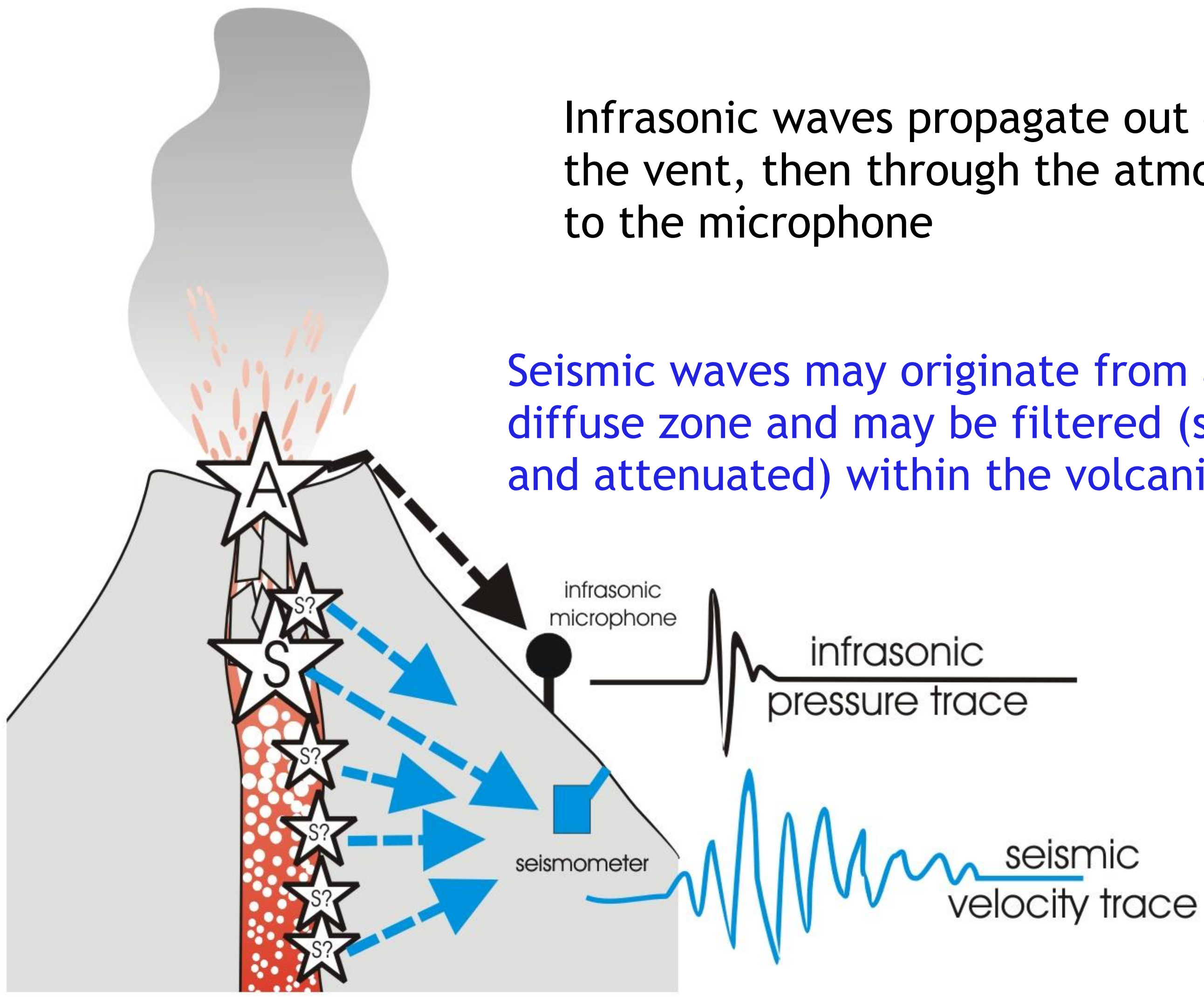
# Wireless Sensor “Motes”

- 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)

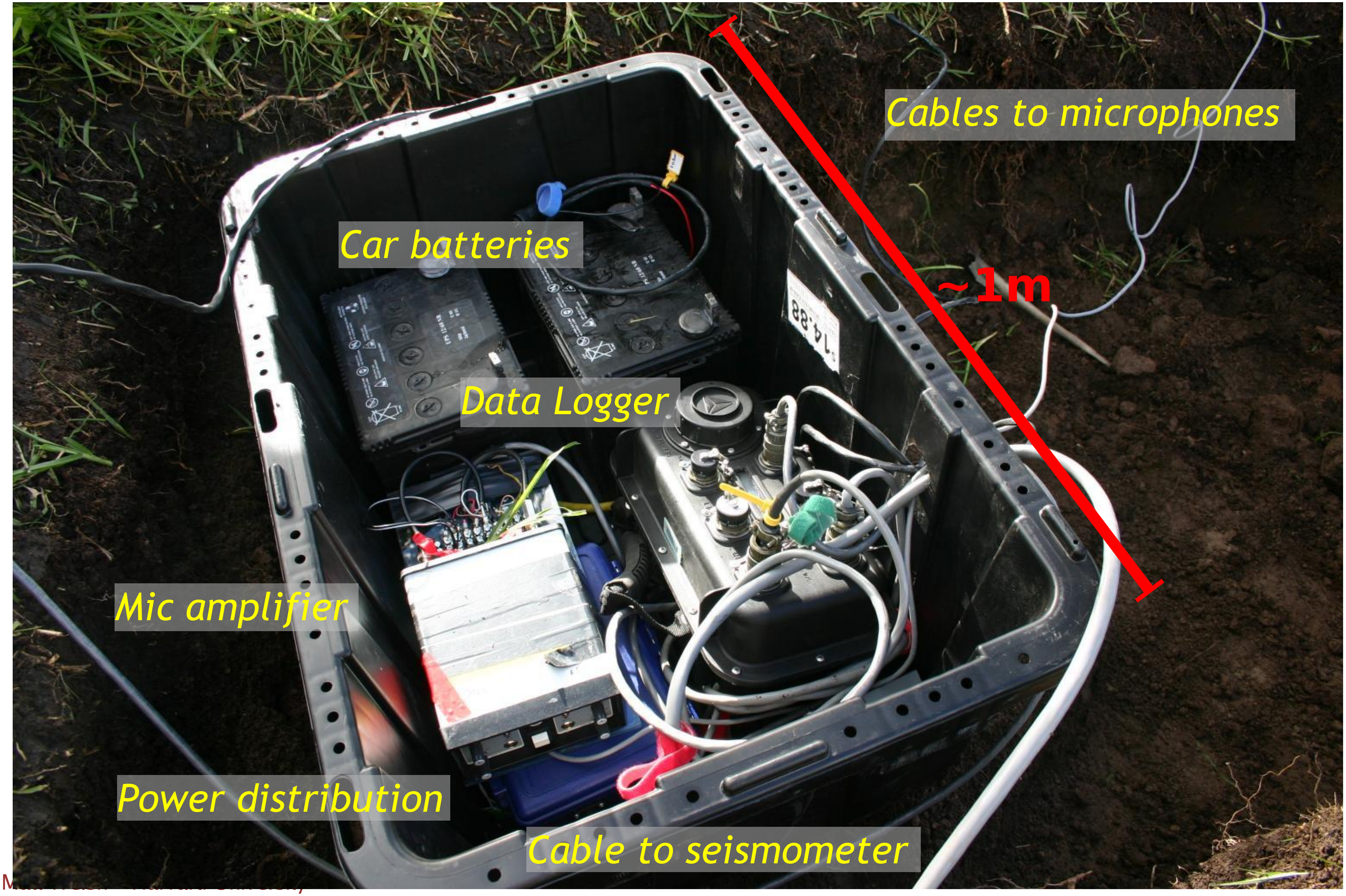
# Sensor Networks for Volcanic Monitoring



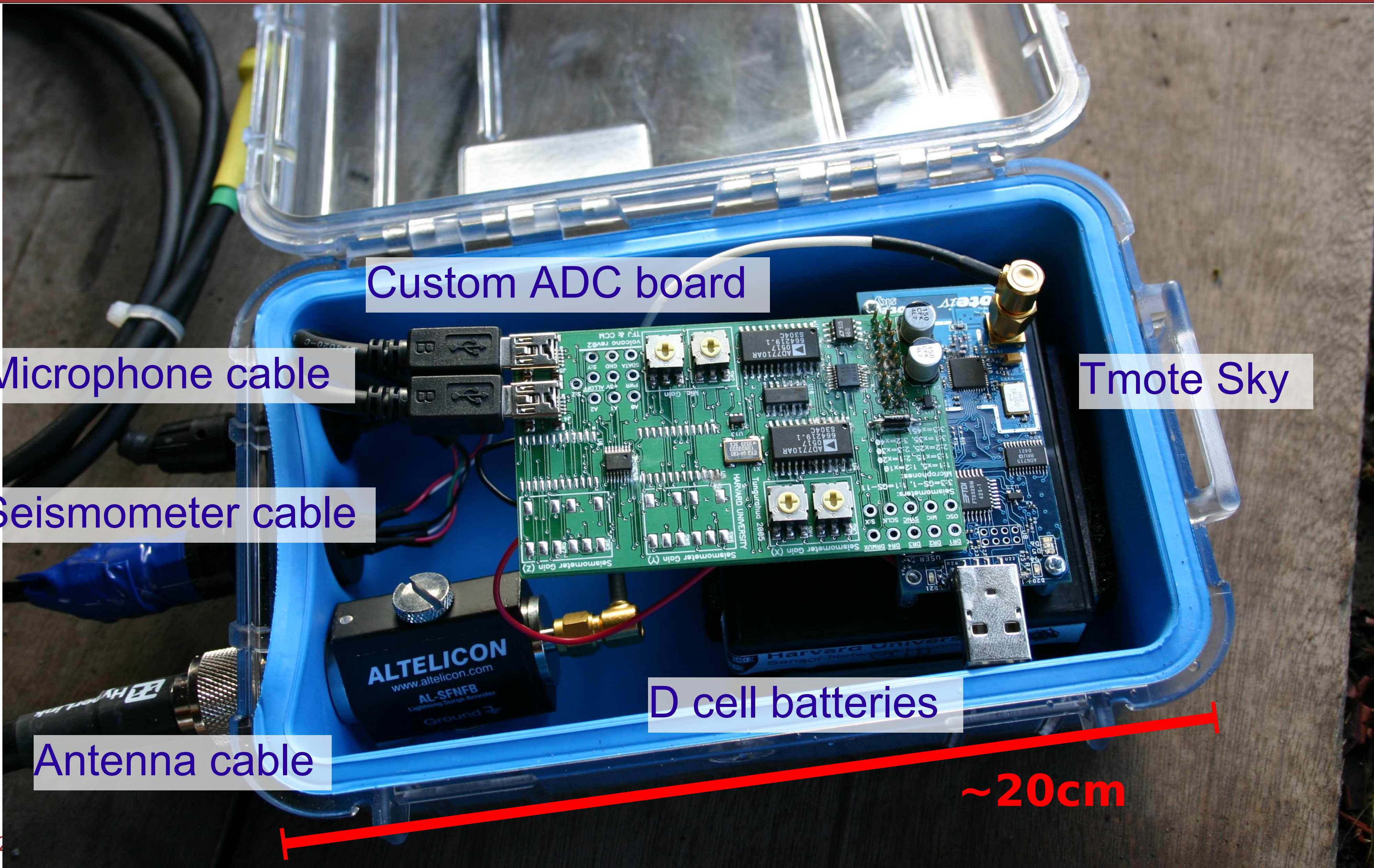

Infrasonic waves propagate out of the vent, then through the atmosphere to the microphone

Seismic waves may originate from a diffuse zone and may be filtered (scattered and attenuated) within the volcanic edifice.

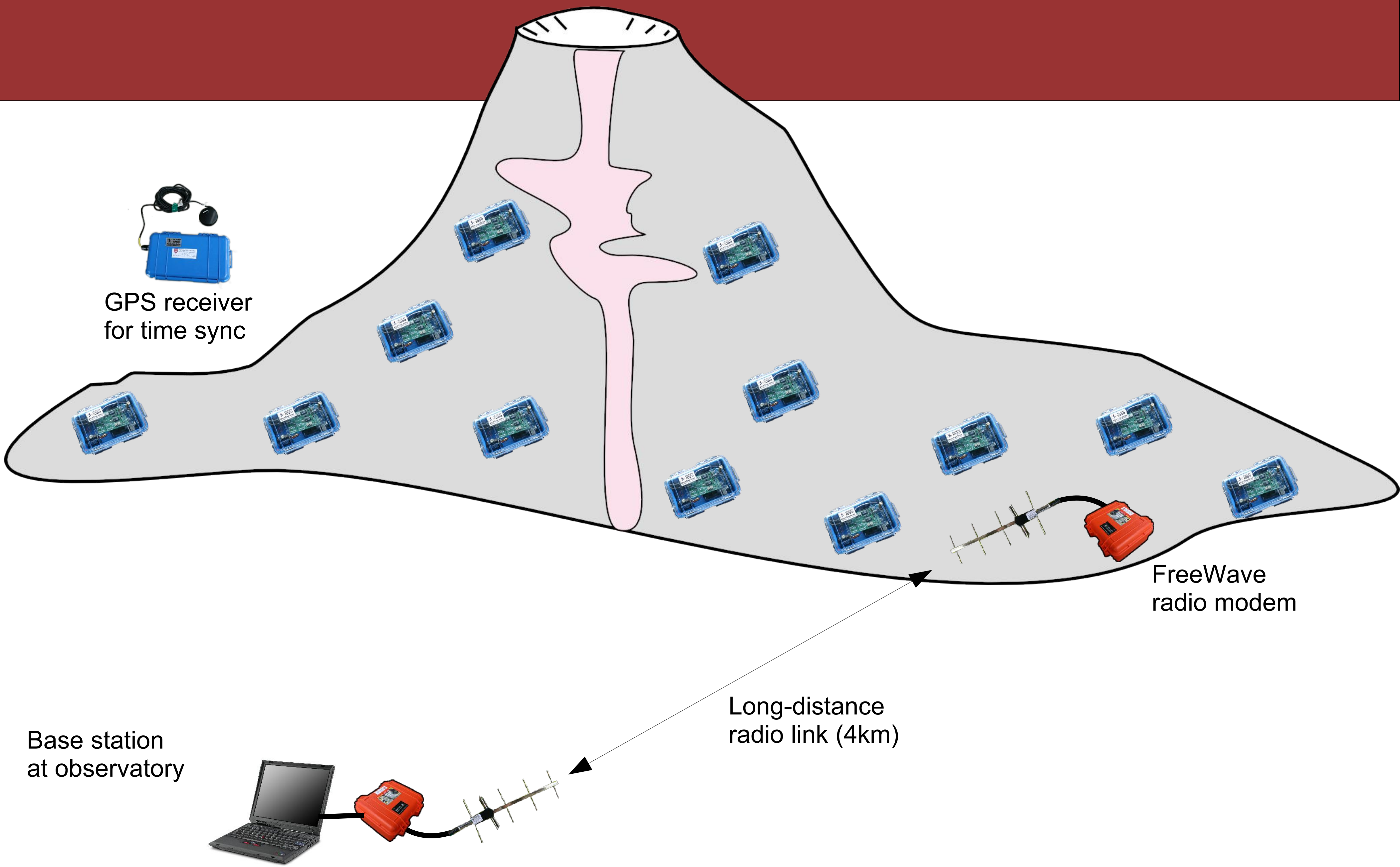
# Existing Volcanic Sensor Station



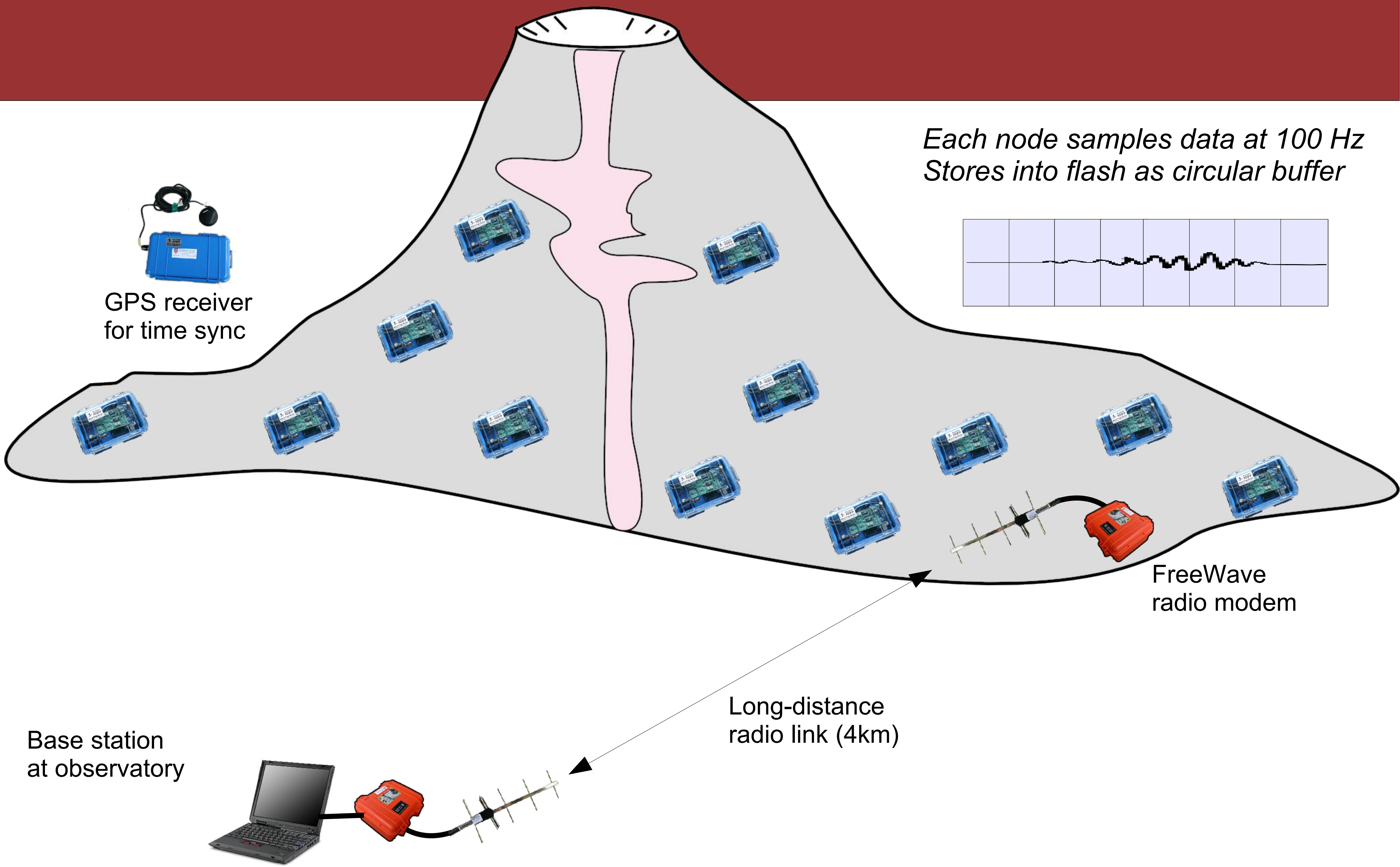
# Our Wireless Volcano Monitoring Sensor Node



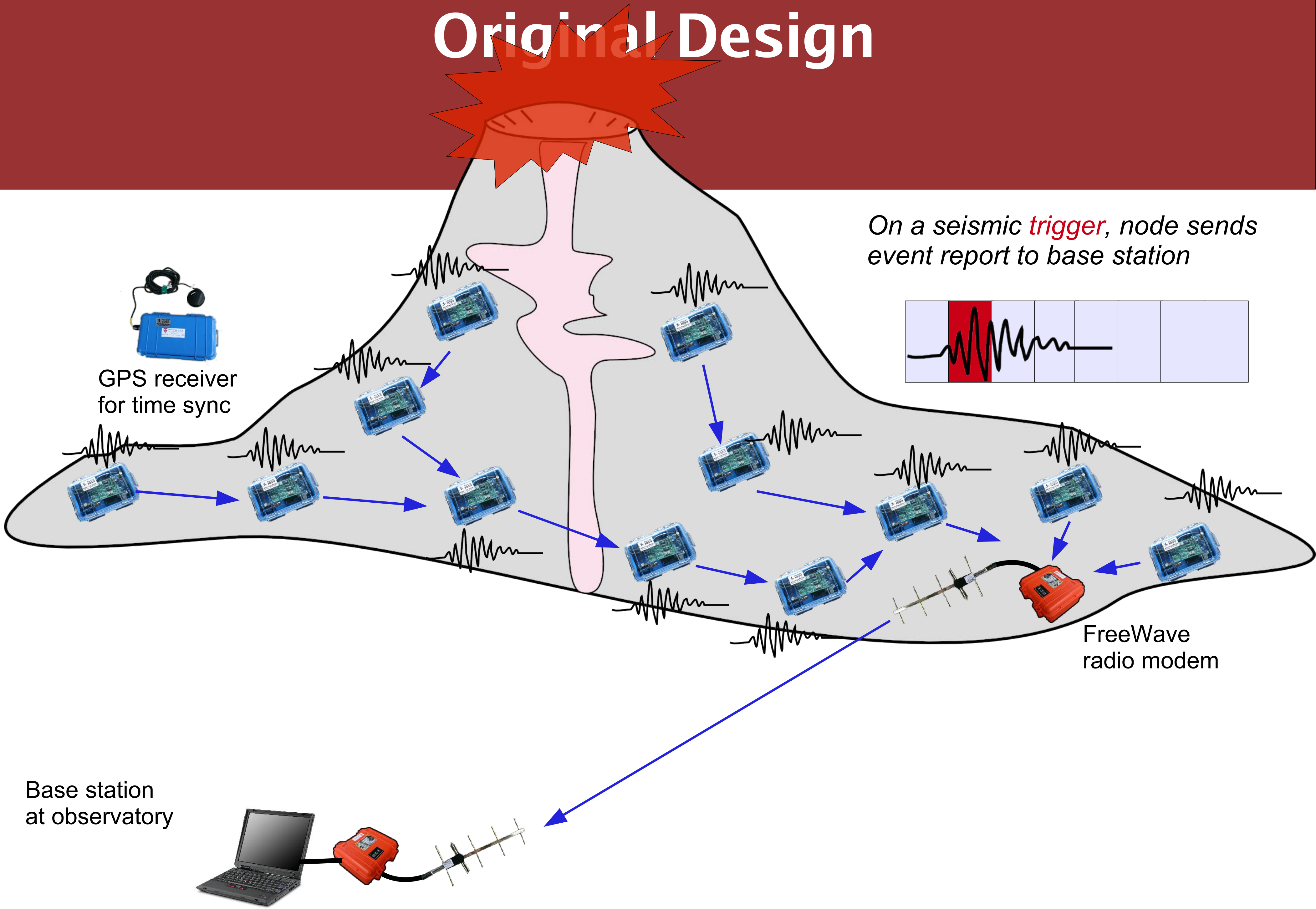
# Original Design



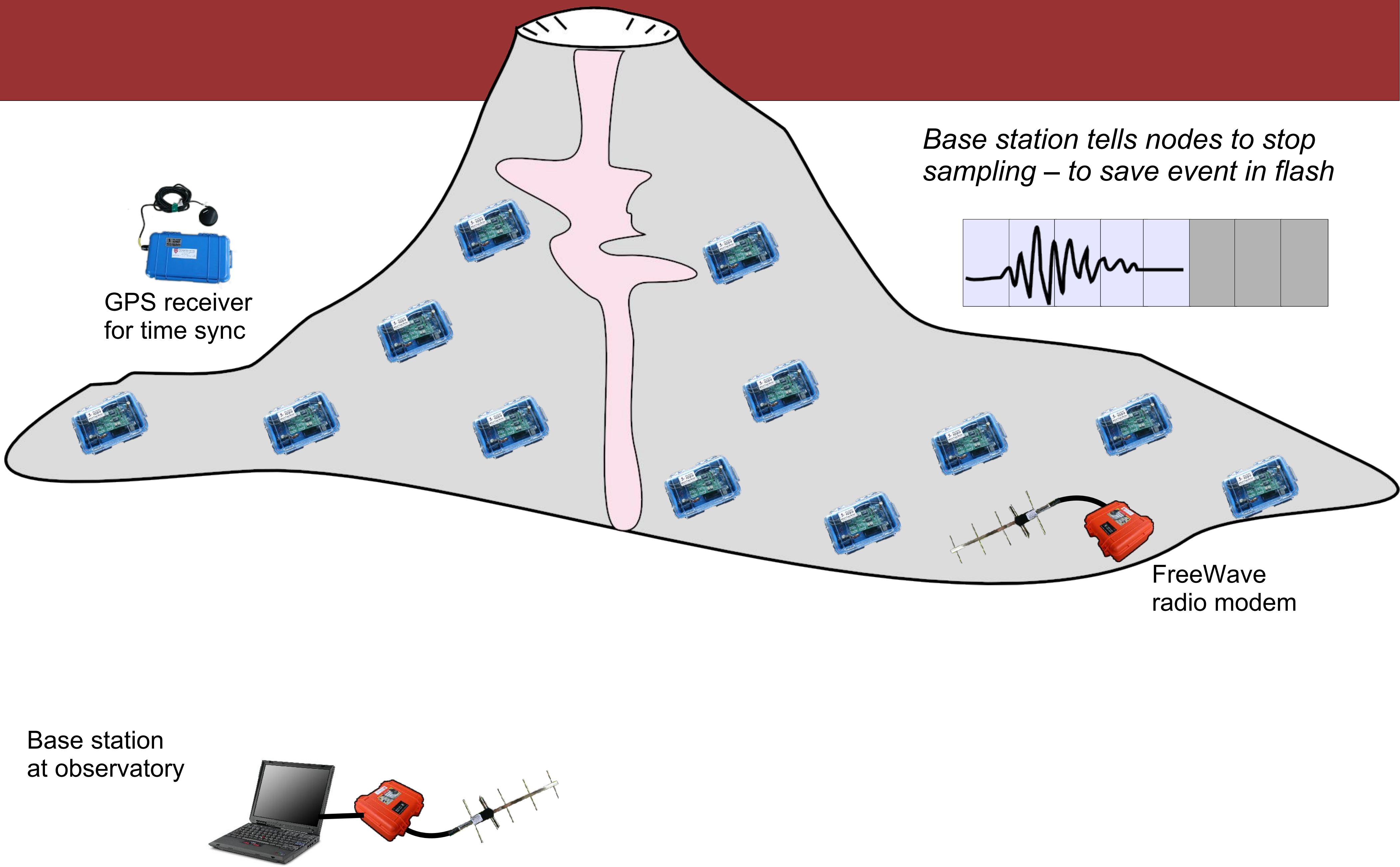
# Original Design



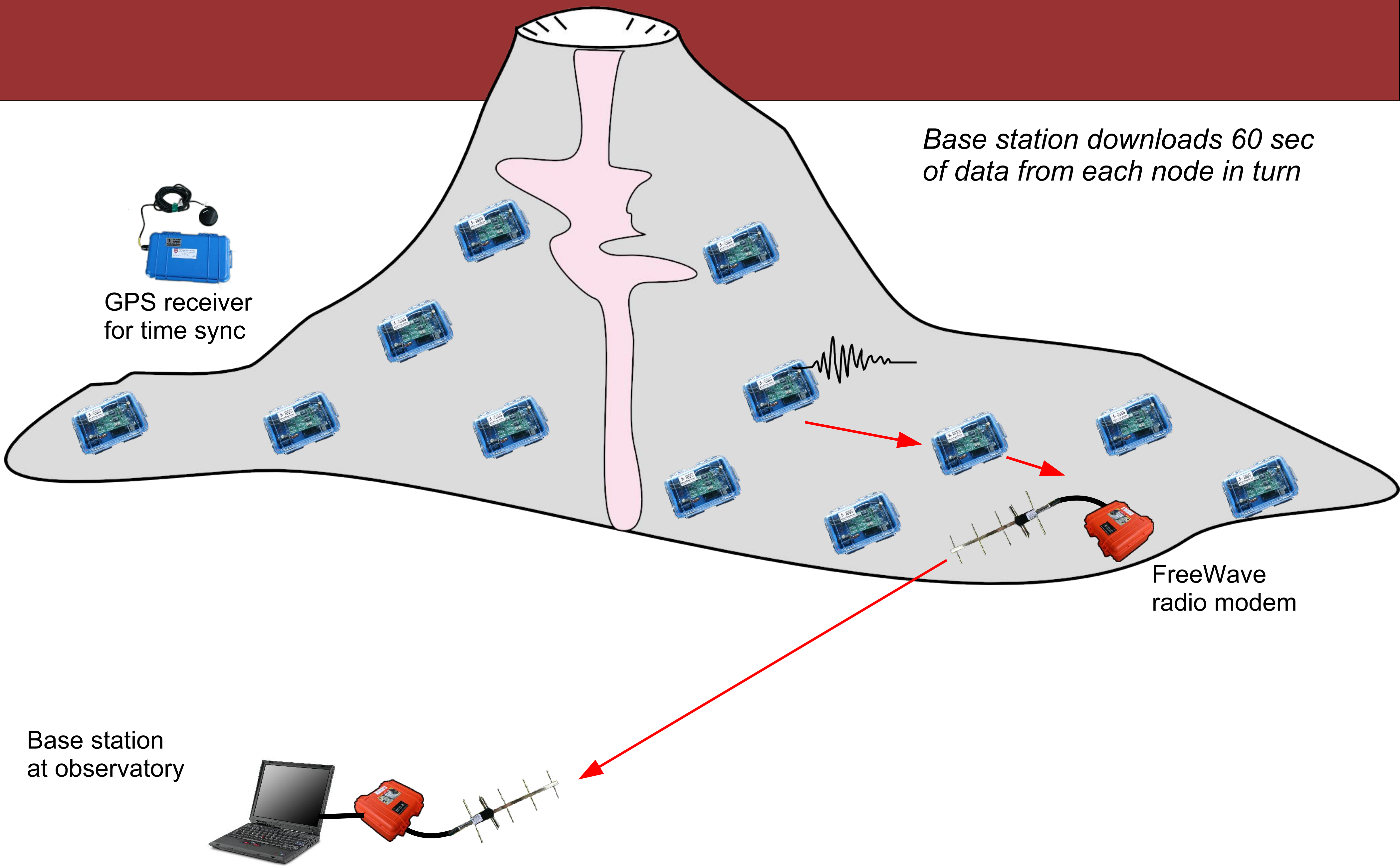
# Original Design



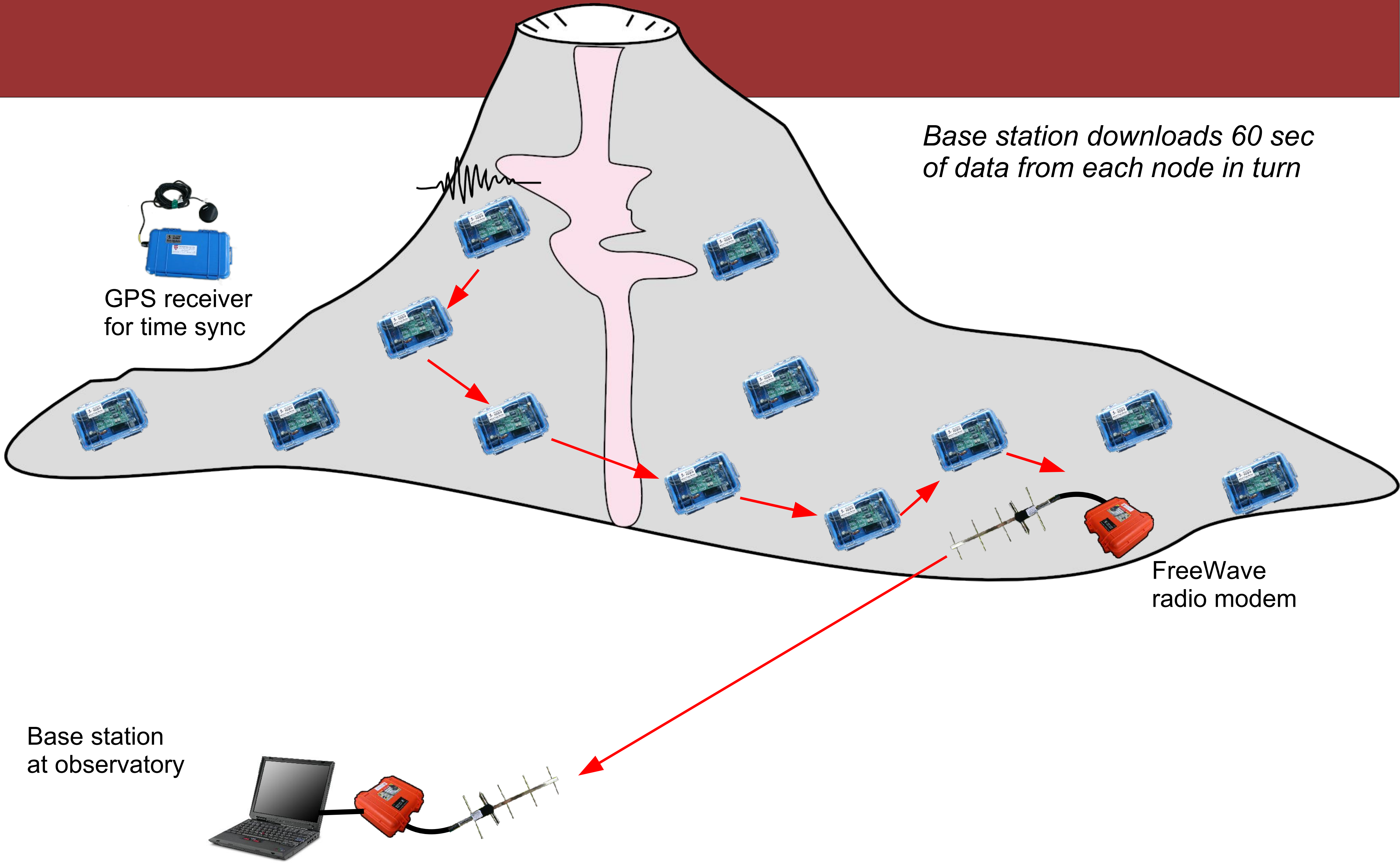
# Original Design



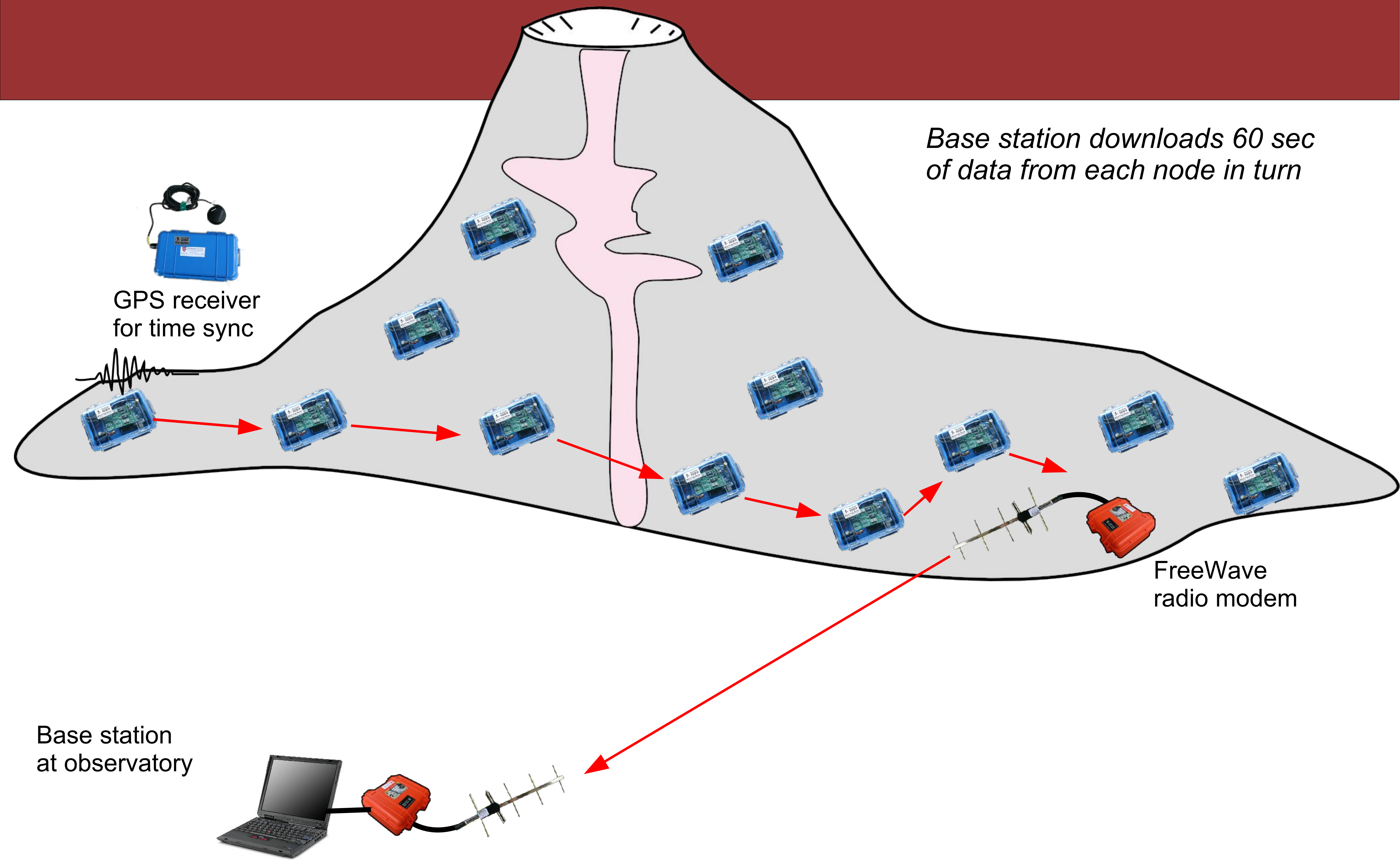
# Original Design



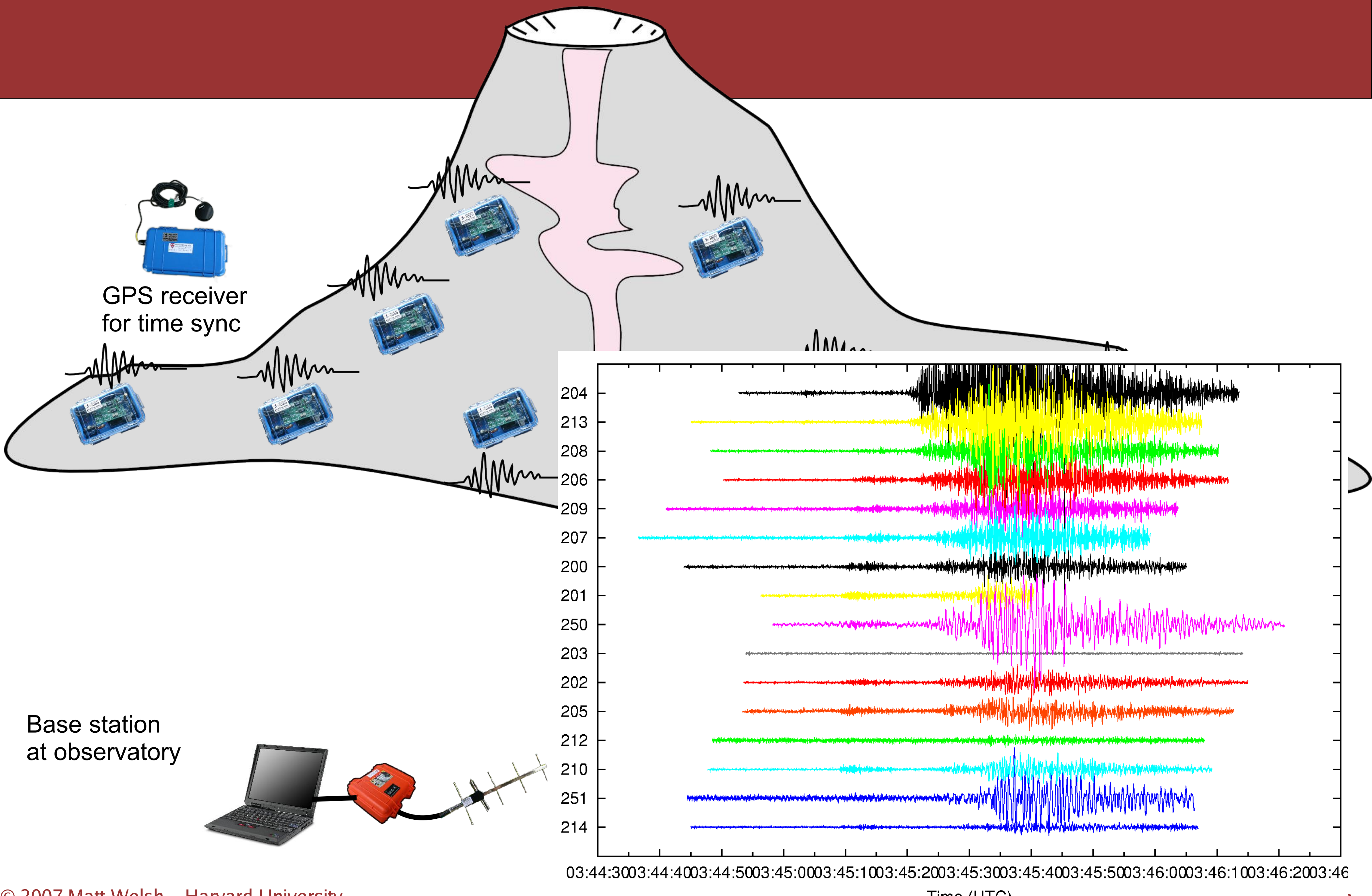
# Original Design



# Original Design

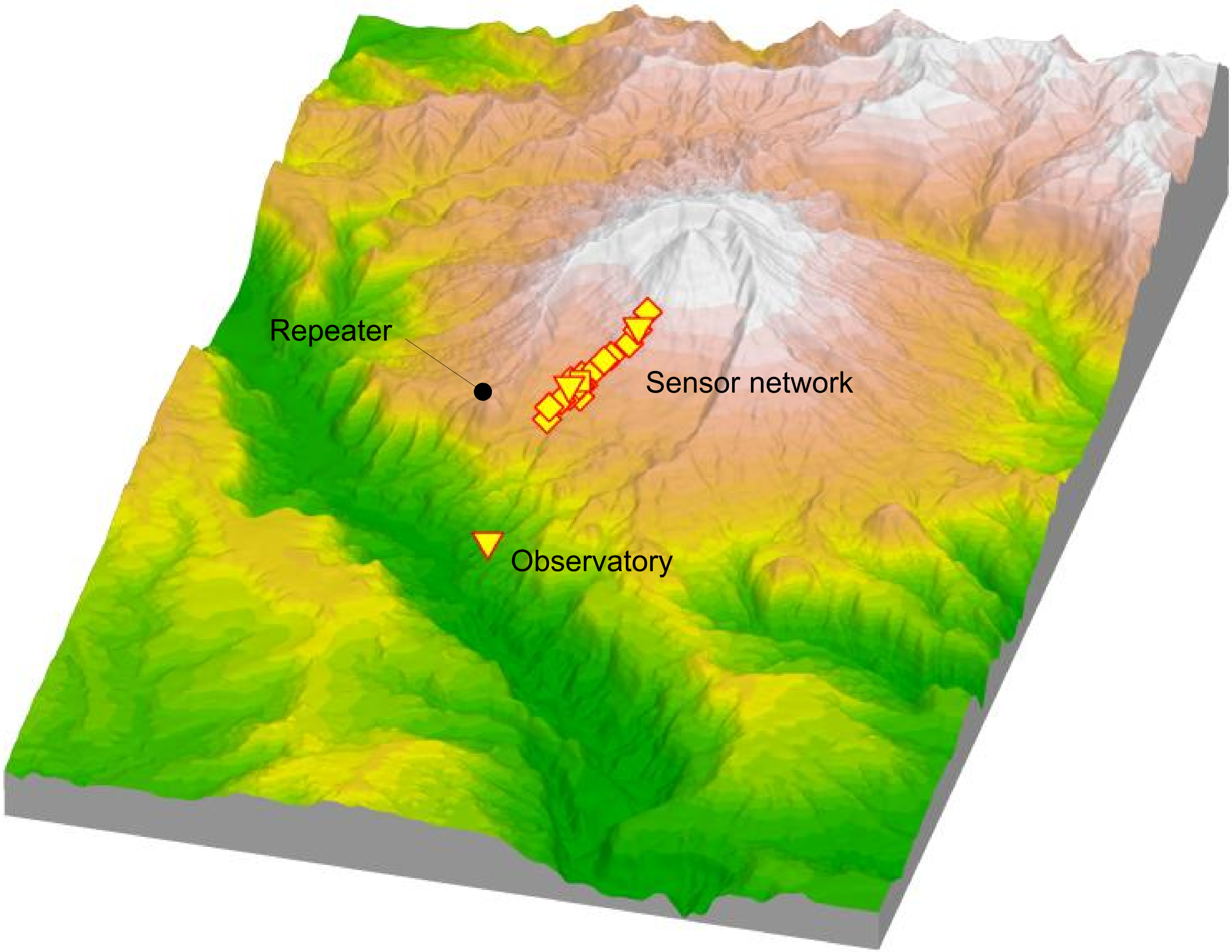


# Original Design



# Deployment at Reventador Volcano, August 2005

# Reventador Deployment Map



# Reventador Deployment - Results

- 16 nodes deployed for 3 weeks in August 2006
  - Collected data on hundreds of earthquakes, eruptions, explosions.
- Lots of lessons learned [Werner-Allen et al., OSDI 2006]
  - Reliability issues with reprogramming sensors over the air
  - Time synchronization protocol bug required extensive data post-processing
  - In-depth validation of data collected compared to traditional wired data-logger
- What (else) went wrong?
  - Nodes would trigger on a small earthquake
  - Network would stop sampling and start downloading data
  - Then “the big one” would hit... and we'd fail to capture any of it!

# How can we do better?

- Original system used FIFO storage and bandwidth management:
  - Treat flash as a circular buffer
  - Download one event at a time from all nodes following a trigger
  - Focused only on capturing discrete seismic events from all nodes in the network
- Instead: Use application-defined priority to drive resource allocation
  - Assign priority to each Application Data Unit (ADU) sampled by the network
  - Use priority to manage local node storage resources
  - Use priority to drive download process
- Main issues:
  - How do we compute and manage priorities?
  - How to tailor this approach for different applications?
  - How can we target different optimization metrics? (Data quality, fairness, etc.)?

# Core Resource Limitations

Sensor nodes have only 1 MByte of flash

- Enough to store ~ 20 minutes of data

Reliable transfer protocols are slow!

- Best speed we've seen is around 500 bytes per second
- Often do much worse: below 100 bytes/sec in some cases
  - *Can take several minutes to download 60 sec. of data from a single node*
- Flush [Dutta et al.] claims up to 1024 bytes/sec in multihop cases
- Also, can only reasonably perform one download at a time

So, need to be careful about how we allocate storage and bandwidth.

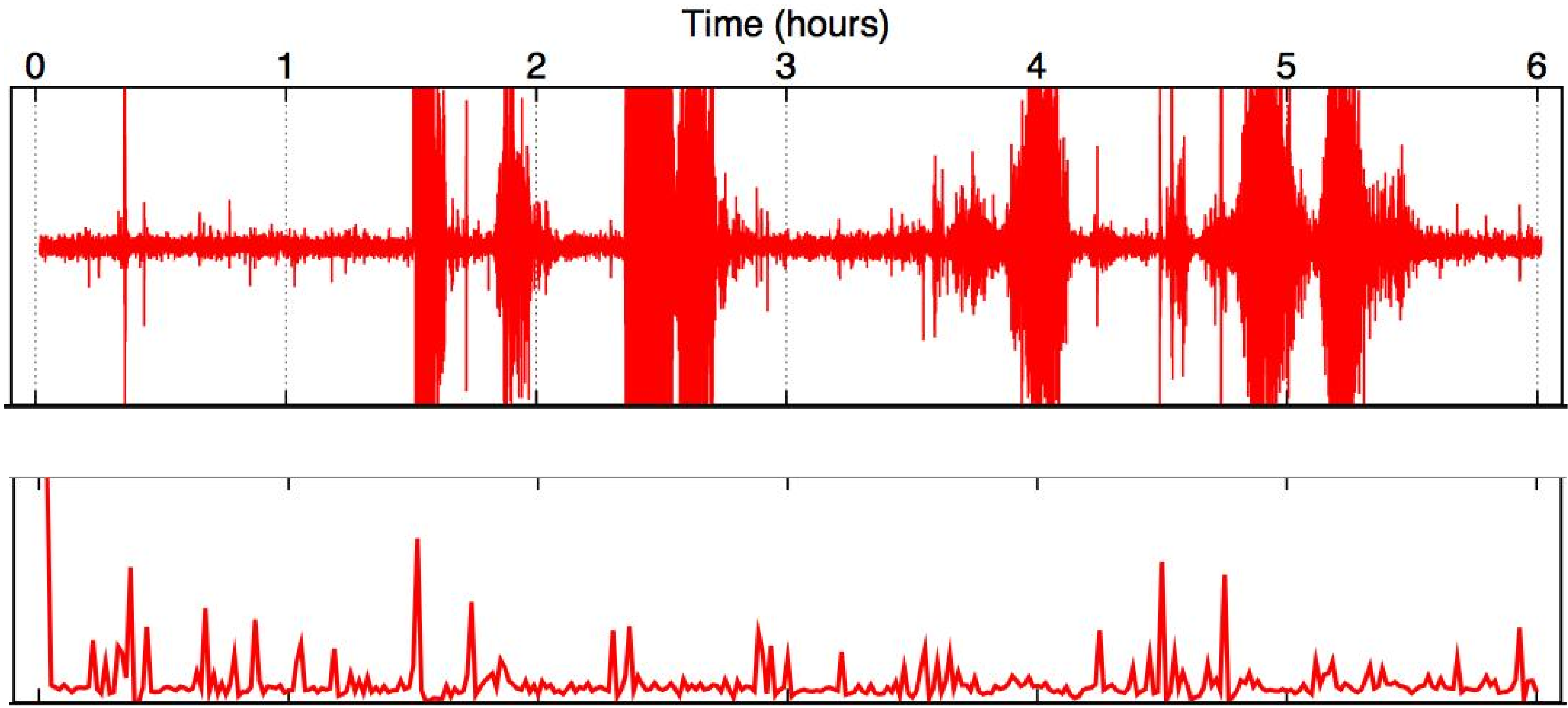
# Defining Priority

Core assumption: some data is “better” than others.

- Application's goal is to extract the “best” data given resource limitations

*Raw signal*

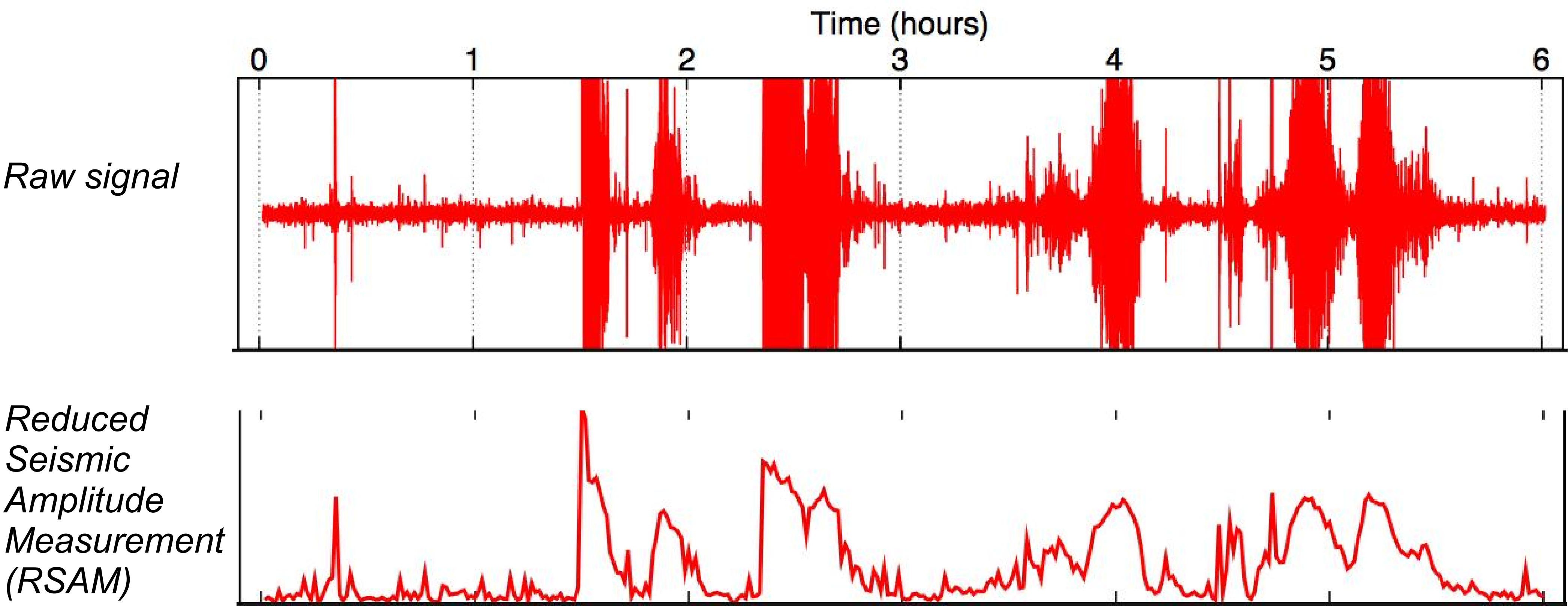
*Short-term average divided by long-term average*



# Defining Priority

Core assumption: some data is “better” than others.

- Application's goal is to extract the “best” data given resource limitations



# Problem Definition

- Each sensor node samples ADUs $A_i = \{d_i, p_i, c_i\}$ where
  - $d_i$ is the data, $p_i$ is the application assigned priority, and $c_i$ is the cost
  - Cost captures bandwidth and/or energy to reliably download ADU from the network
  - Priorities define partial order on all ADUs: $A_i \geq A_j$ iff $p_i \geq p_j$
- High-level goal: Download highest-priority ADUs such that the total cost for data retrieval is less than capacity $C$
  - Where capacity expressed in terms of bandwidth or energy availability
- We define the *optimal set* of ADUs Ω:
  - Set of ADUs, rank-ordered by decreasing priority $\{ A_1 \geq A_2 \geq \ldots \geq A_k\}$ *s.t.* total cost is less than capacity $C$

*The overall system goal is to download the ADUs in the optimal set Ω.*

# Problem Definition

Determining Ω requires complete knowledge of all ADUs over all time.

- When determining whether to download a given ADU, must know whether it is in Ω.
- No way of knowing this without knowledge of the future ADUs that will be sampled.

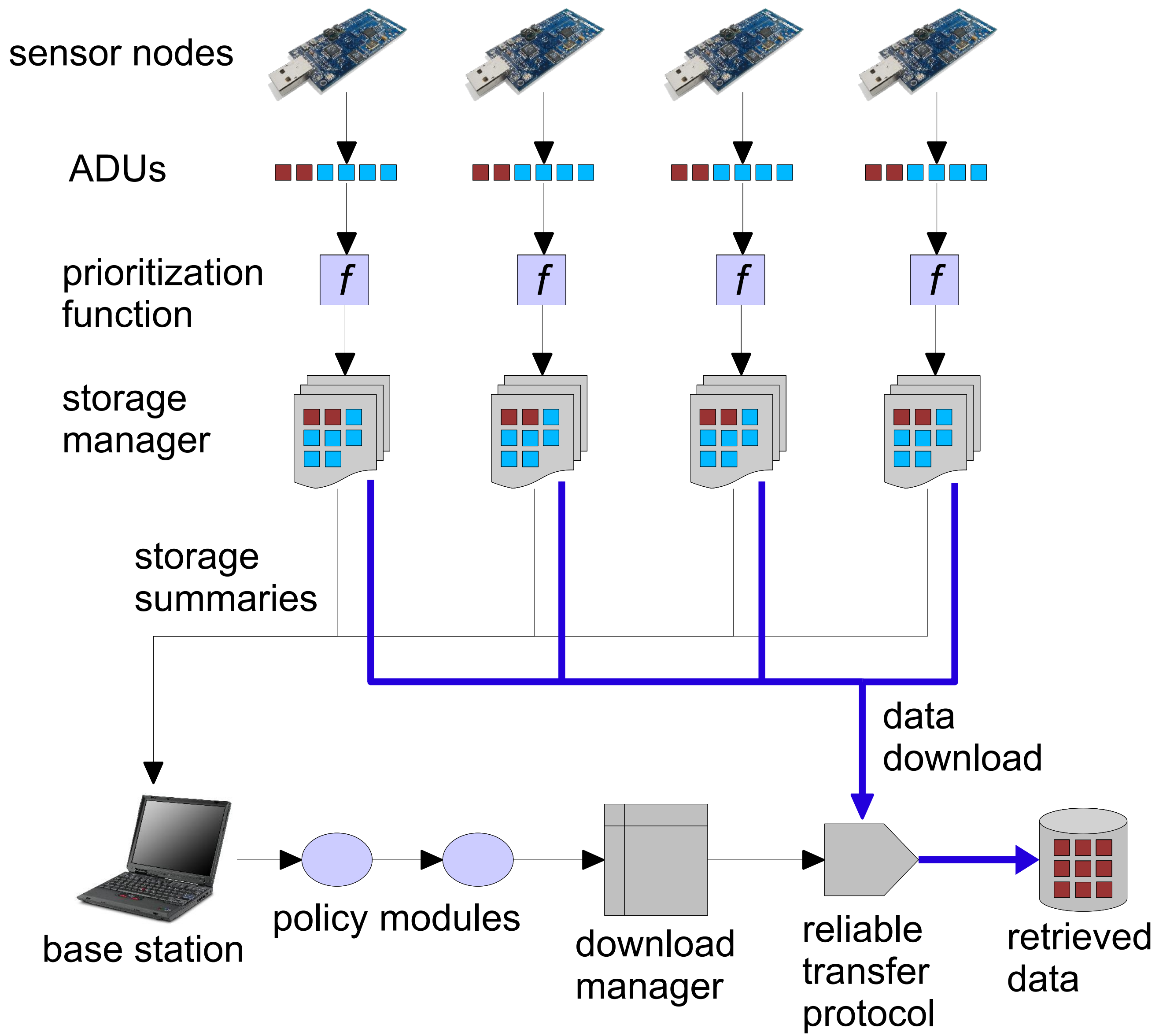
Also, retrieving Ω may require infinite per-node storage.

So, we must come up with an *online* algorithm that can operate with *limited* per-node storage capacity.

Efficiency metric: Coverage of downloaded set S with Ω

- Coverage = $| S \cap \Omega | / | \Omega |$
- Note that this metric gives no credit to ADUs downloaded *not* in the optimal set Ω
- Other metrics possible too ... more later.

# Lance Architecture



# Lance Storage Manager

- Nodes compute priority for each sampled ADU
  - Prioritization function is app-specific
  - Must not consume inordinate resources
- Nodes treat local flash as bounded-size priority queue
  - When storing new ADU, evict lowest-priority ADU first
- Flash technology imposes some limitations:
  - Must erase entire sector (e.g., 64 KB) before new data can be written
    - *We match size of ADU to one sector: About 109 sec. of data at 100 Hz*
  - Sector erase is slow – 0.6 sec typical for ST M25P80
  - Therefore, we perform sector erase concurrently with storing previous ADU
    - *Must evict a sector before we know what will be stored in it!*

# Lance Download Manager

- Design goal: Perform bandwidth allocation *centrally*, at the base station
  - Simpler, and far more robust, than a complex decentralized algorithm.
  - Allows network's behavior to be radically changed by tweaking policies at base, without reprogramming nodes.
- Nodes send periodic storage summaries to the base station
  - List of ADUs stored on each node with corresponding priority and timestamp
- Storage summaries used by download manager to assign *download priority* for each ADU
  - Recall: We perform one ADU download at a time to avoid network congestion

# Download Manager Policy

Simplest policy: Download priority == node assigned storage priority

- Simply rank-order all ADUs stored in the network by priority
- Download the highest-priority ADU

Problem: Download priority for an ADU may depend on *global* network state

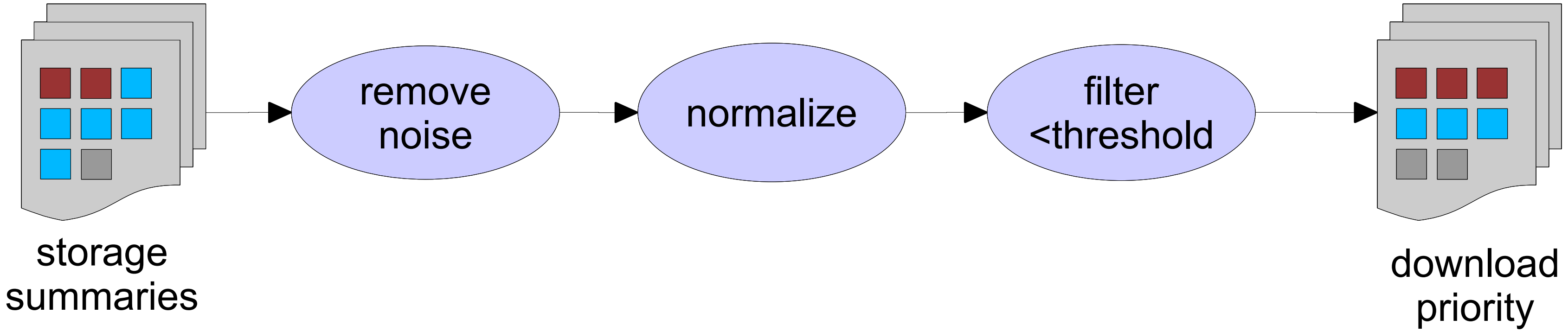
- Example: Achieving fairness across nodes, or *normalizing* ADU priorities across nodes
- Correlated event detection: Detect spike in priority across several nodes and download same time window from all nodes

Individual nodes may not be aware of the network-wide priority for a given ADU

- Want to allow application to modify the “raw” ADU priorities to implement a wide range of download policies.

# Lance Policy Modules

- User-supplied functions to inspect, filter, or modify raw ADU priorities at the base station.
- Composed into a linear chain:



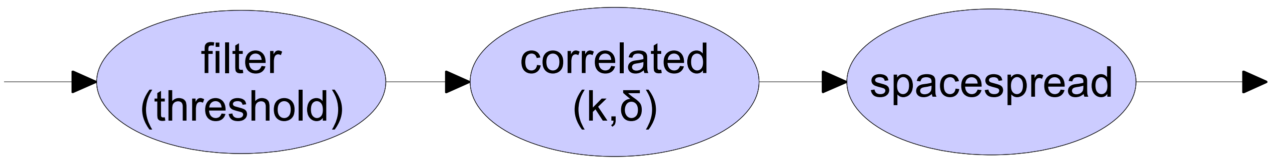

  - Take stream of ADU priorities as input, emit (possibly modified) priorities as output
  - Can maintain internal state
  - Must run efficiently (i.e., keep up with stream of ADU priorities from the network).

# Example Policy Modules

- Priority thresholding
  - **filter**: Set priority to 0 if input priority below threshold T
    - *Download manager will not download an ADU with zero priority value*
- Noise removal and calibration
  - **adjust**: adjust raw priority by adding or subtracting fixed offset
  - **debias**: normalize priority values across nodes
- Priority dilation
  - **timespread**: assign high ADU priority values to ADUs adjacent in time
  - **spacespread**: assign high ADU priority values to ADUs sampled by different nodes
- Cost weighting
  - **costweight**: scale ADU priority by cost to download
    - *e.g., based on number of radio hops from the base station*

# Example: Correlated Event Detection

- **correlated** policy module W(k,δ)
  - counts number of ADUs within a time window δ with a nonzero priority value
  - if at least k ADUs match, retain input priorities for ADUs in the window
  - otherwise, set ADU priorities in window to 0
- Implementing our original monitoring system in Lance:




  - Policy now implemented at the base station, rather than on motes.
  - Easy to modify behavior of network just by changing policy modules.

# Evaluation Metrics

## Recall: Definition of optimal set $\Omega$

- Set of ADUs that a perfect system with full knowledge of future data arrival would have downloaded, given the same constraints on bandwidth and storage.

## Coverage metric: $| S \cap \Omega | / | \Omega |$

- Given downloaded set S and optimal set $\Omega$
- Problem: doesn't matter *which* ADUs in $\Omega$ we manage to download.

## We define *weighted coverage $K(S,\Omega)$* as follows:

- Assign a score $\sigma$ to each ADU as: $| \Omega |$ - rank of ADU in $\Omega$

  (Top ranked ADU gets score N, second-ranked N-1, etc.)

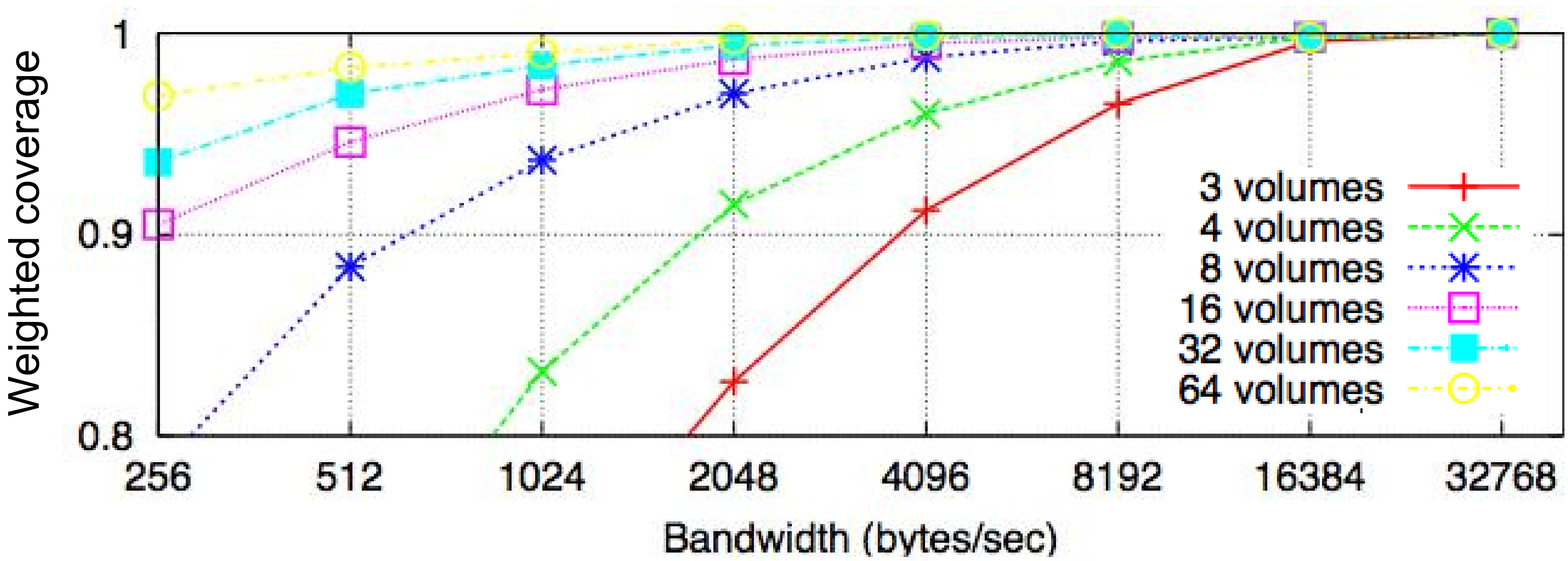
$$K(S,\Omega) = \frac{\sum_{x \in S \cap \Omega} \sigma(x)}{\sum_{y \in \Omega} \sigma(y)}$$

# Methodology

- Simulated 16 sensor nodes in a radial or linear topology
  - ADU size of 64 KB and data generation rate of one ADU per node per minute
- ADU priorities drawn from three distributions:
  - Uniformly random
  - Zipf ($\alpha = 1$)
  - "Bursty Zipf": With probability P, select a new value from a Zipf distribution; with probability 1-P use previous value.
    - *P controls "burstiness": P=1 is equivalent to Zipf. P=0 all values are the same.*
- Also make use of real data sets from Reventador and Mt. St. Helens seismic sensors.

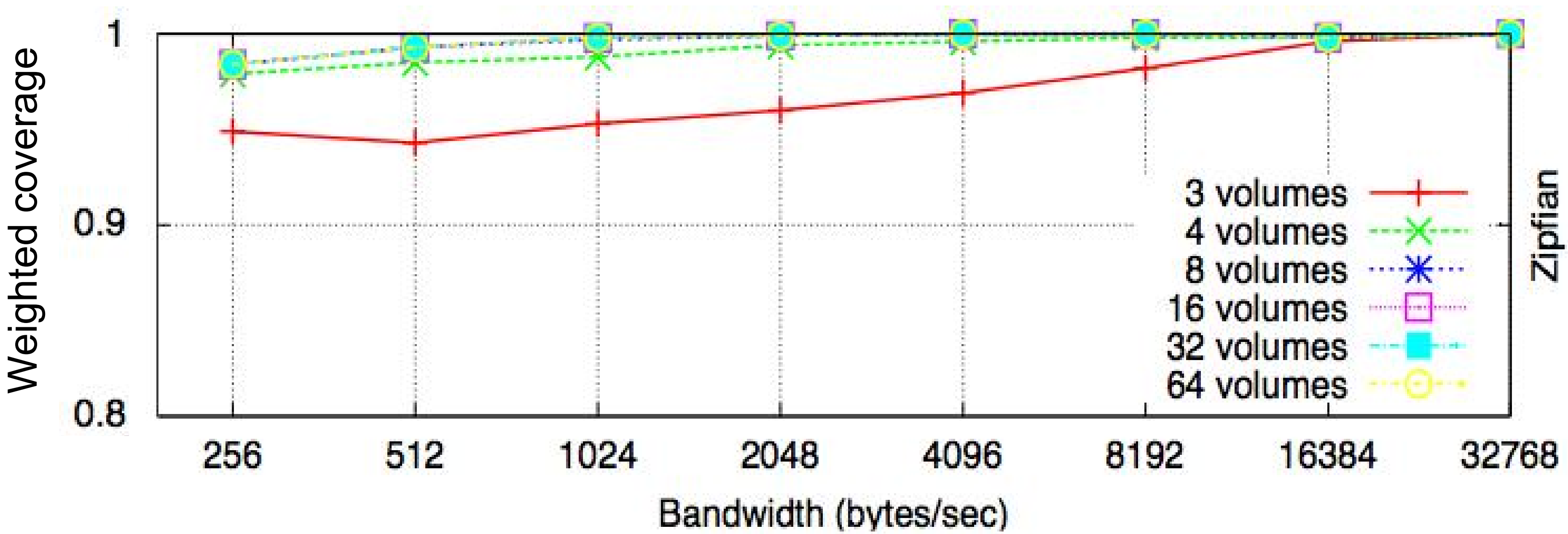
# FIFO storage management

- Poor coverage under constrained bandwidth and storage capacity

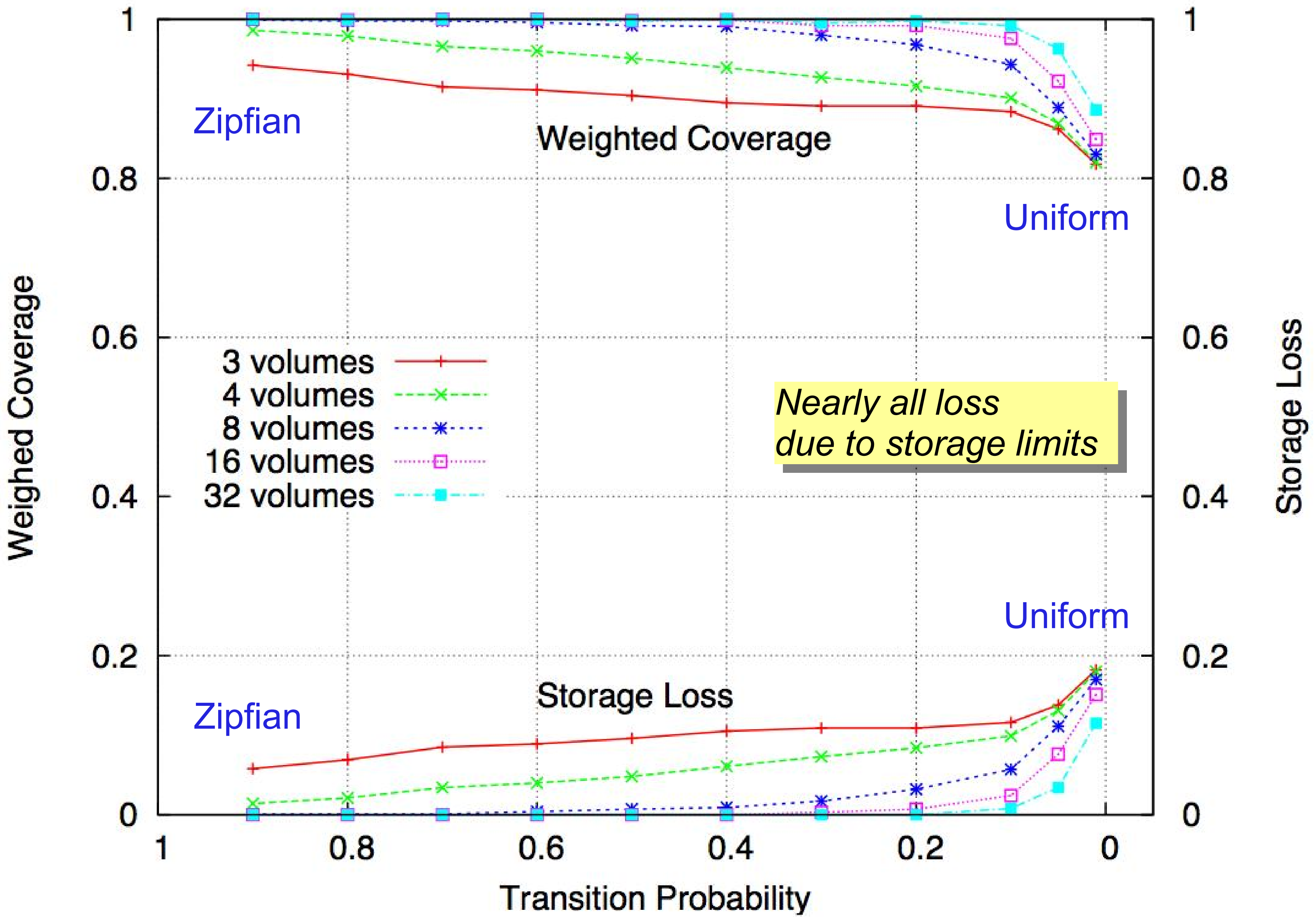


# Performance using Lance

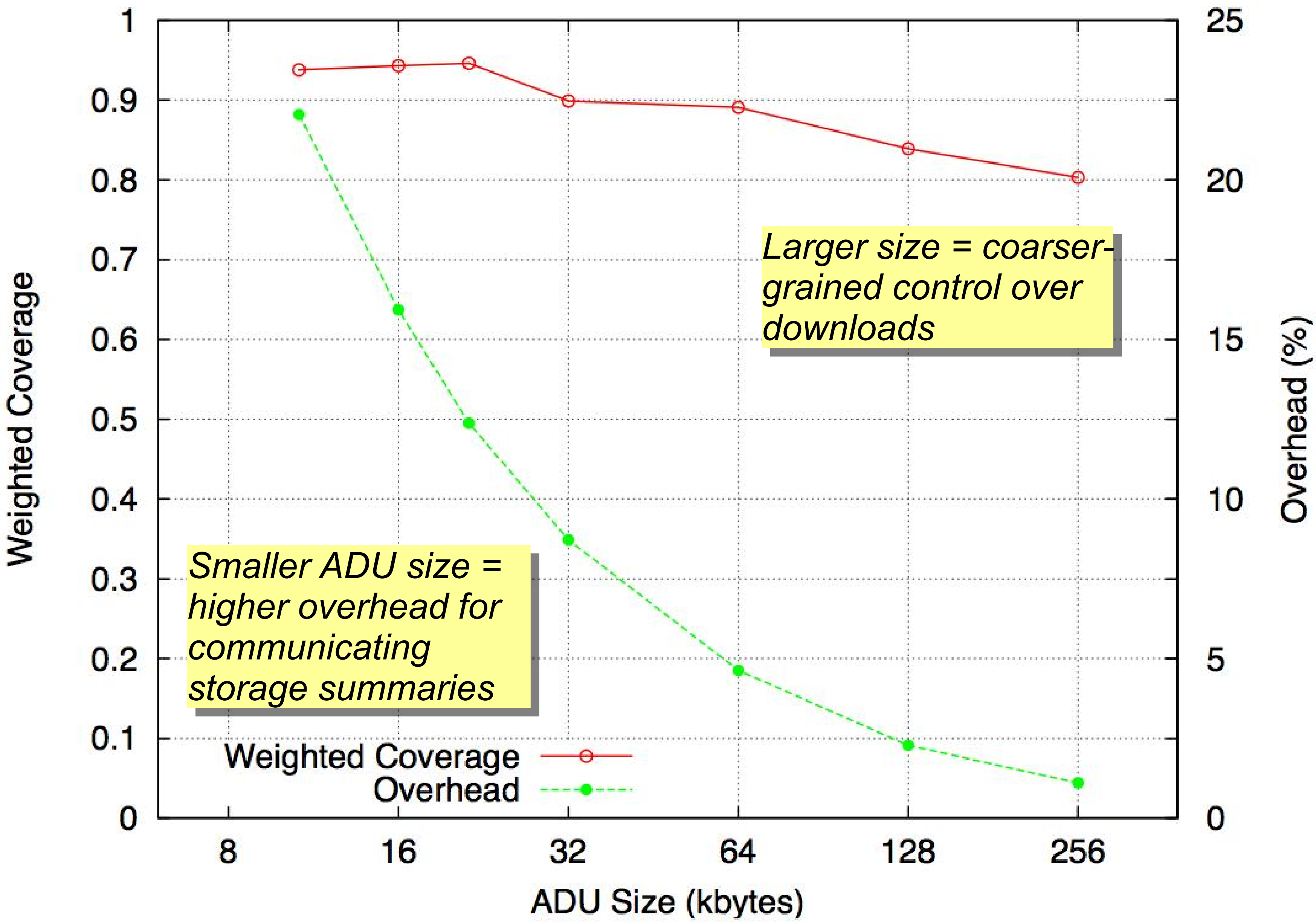
- *Near-optimal* coverage under range of resource constraints
  - Exceeds 95% in all cases



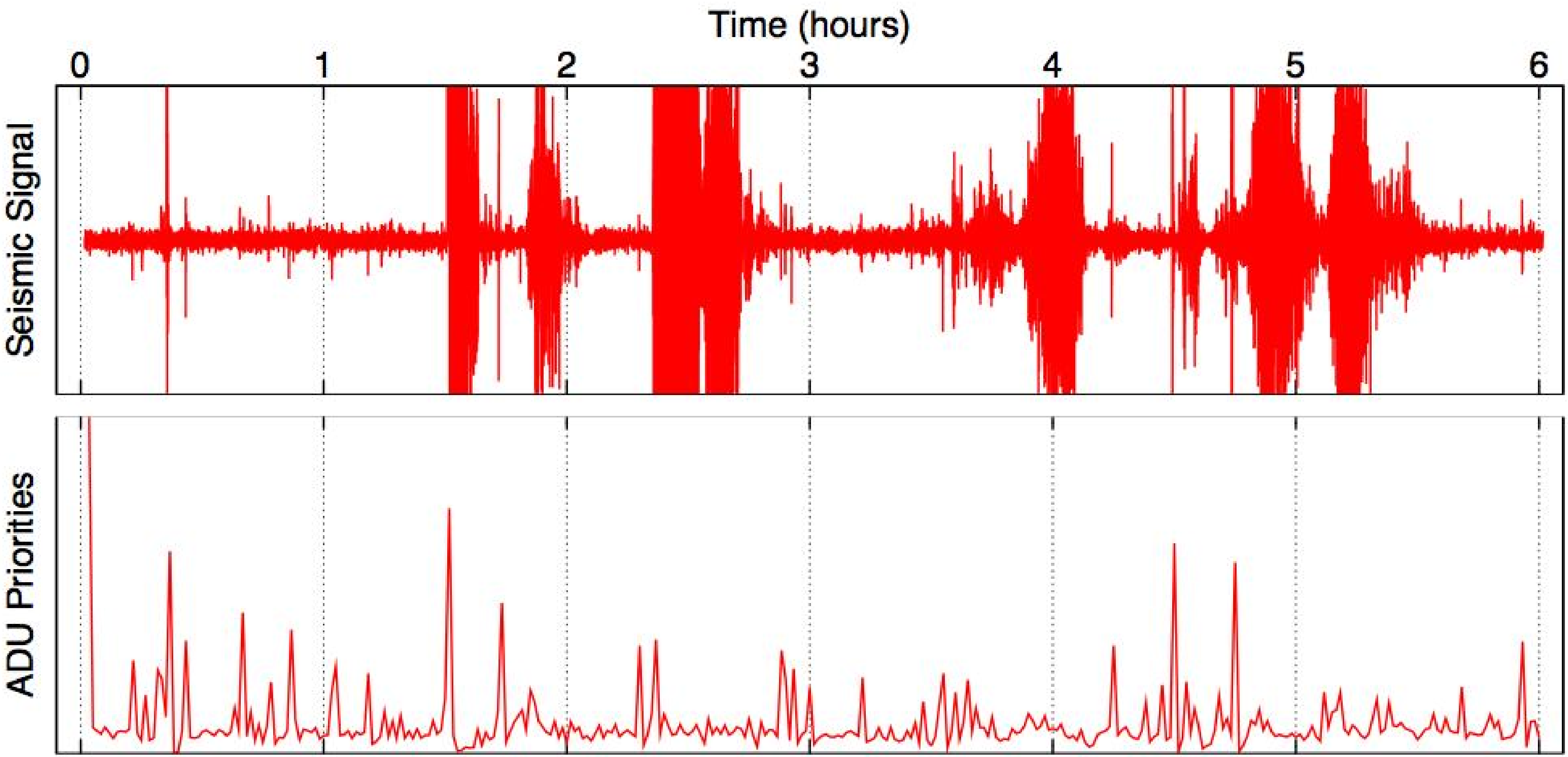
# Effect of priority burstiness



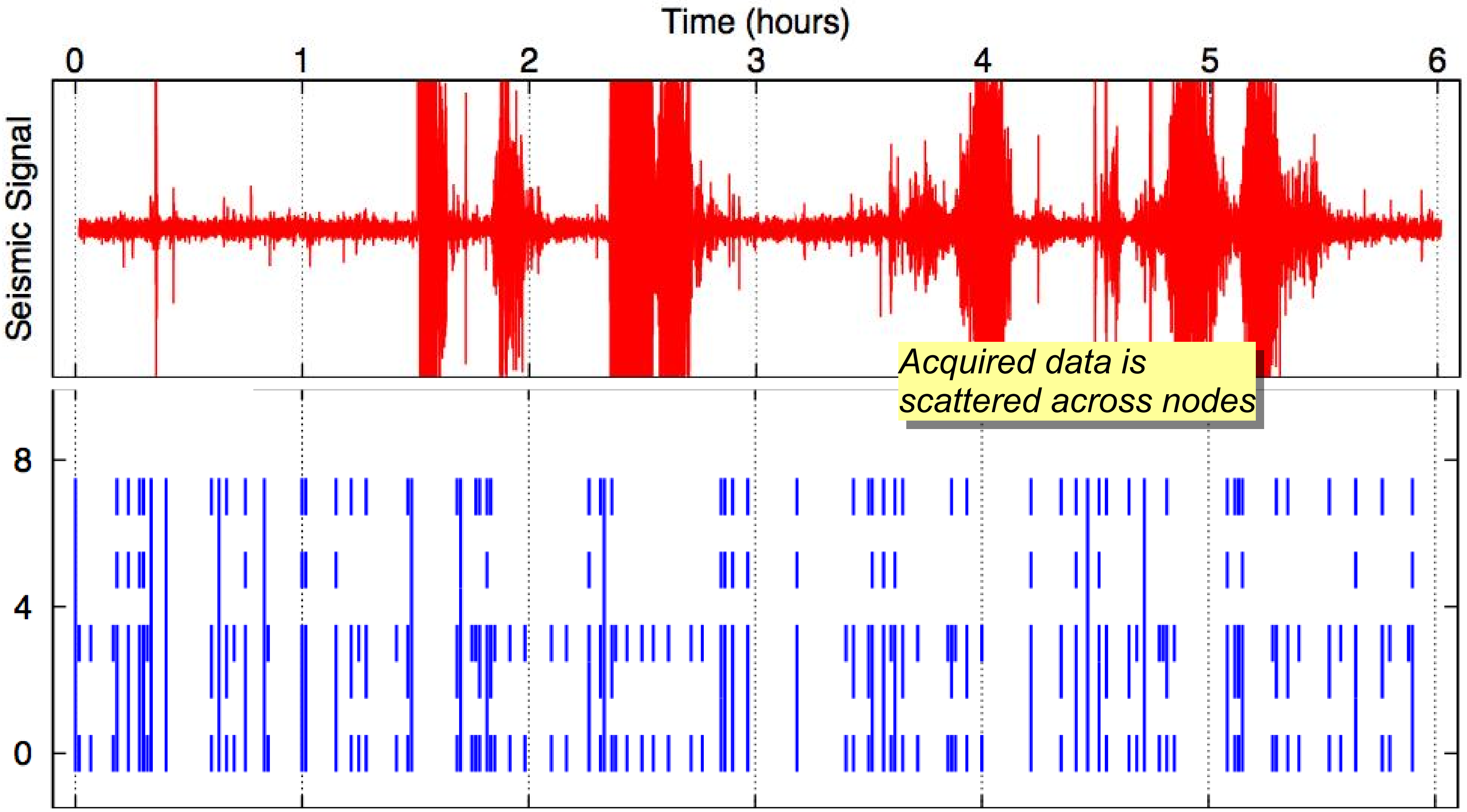
# Effect of varying ADU size



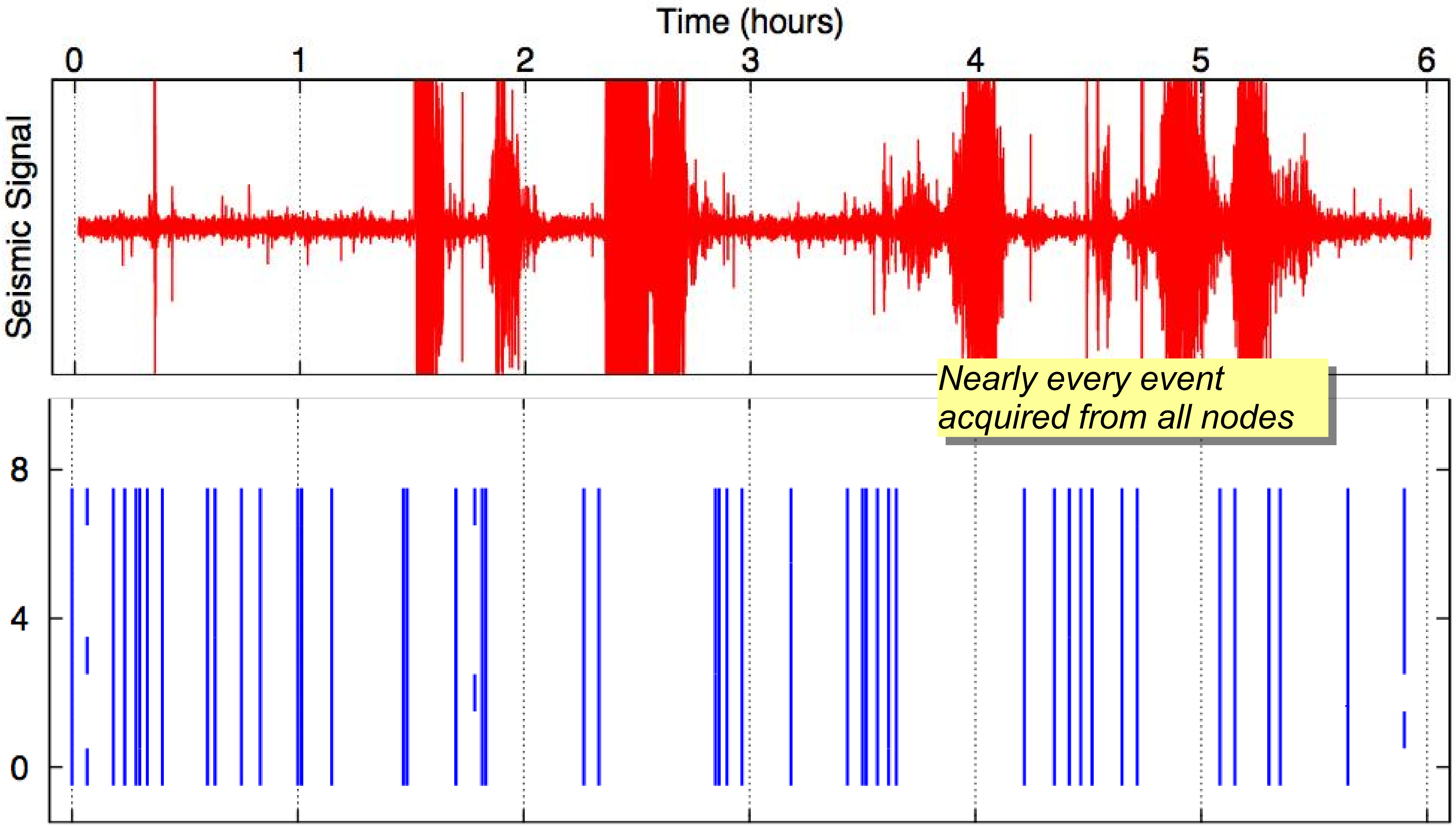
# Correlated event detection



# Default download policy



# Correlated event detection policy

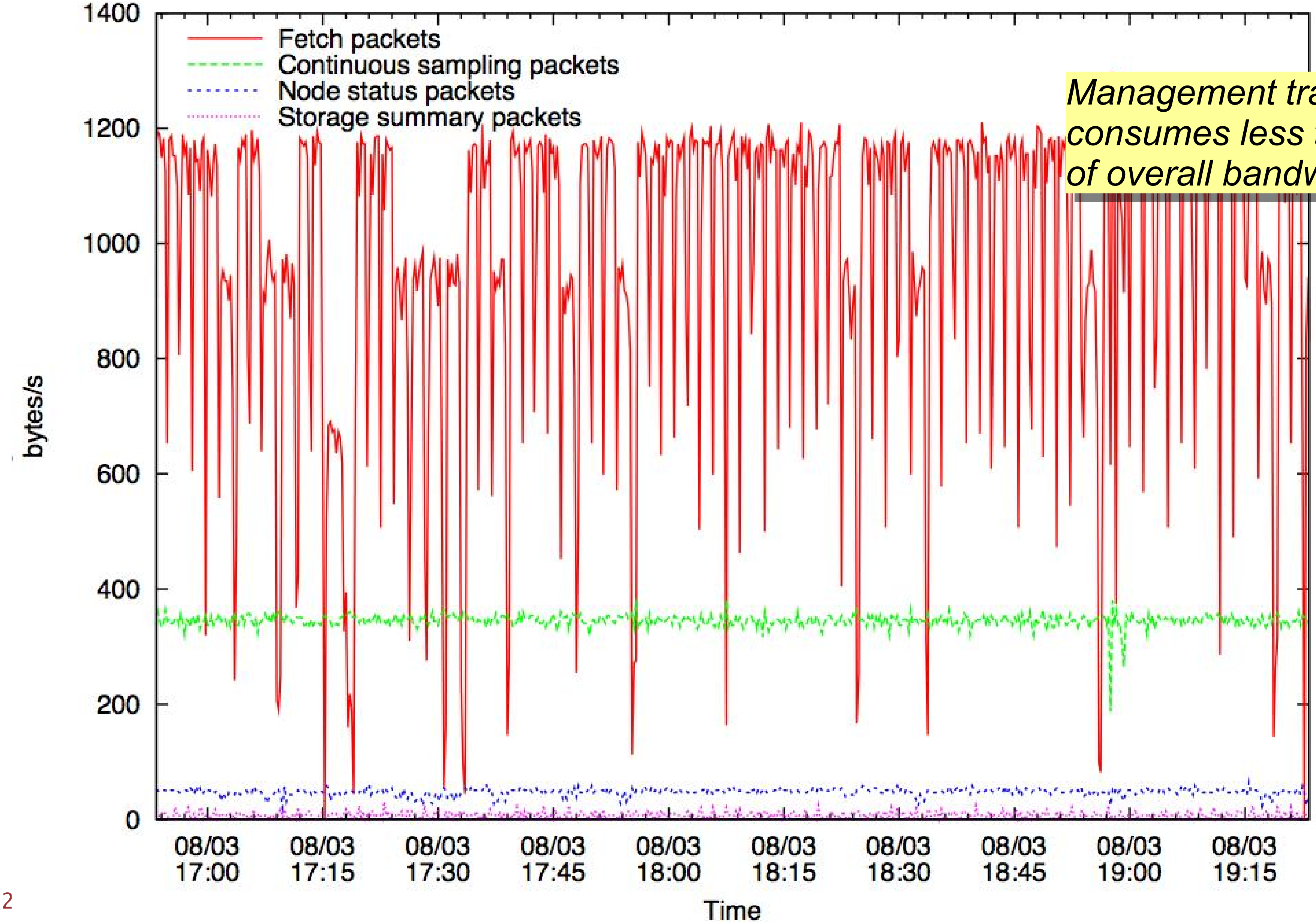


# Tungurahua field deployment, August 2007

# Deployment Statistics

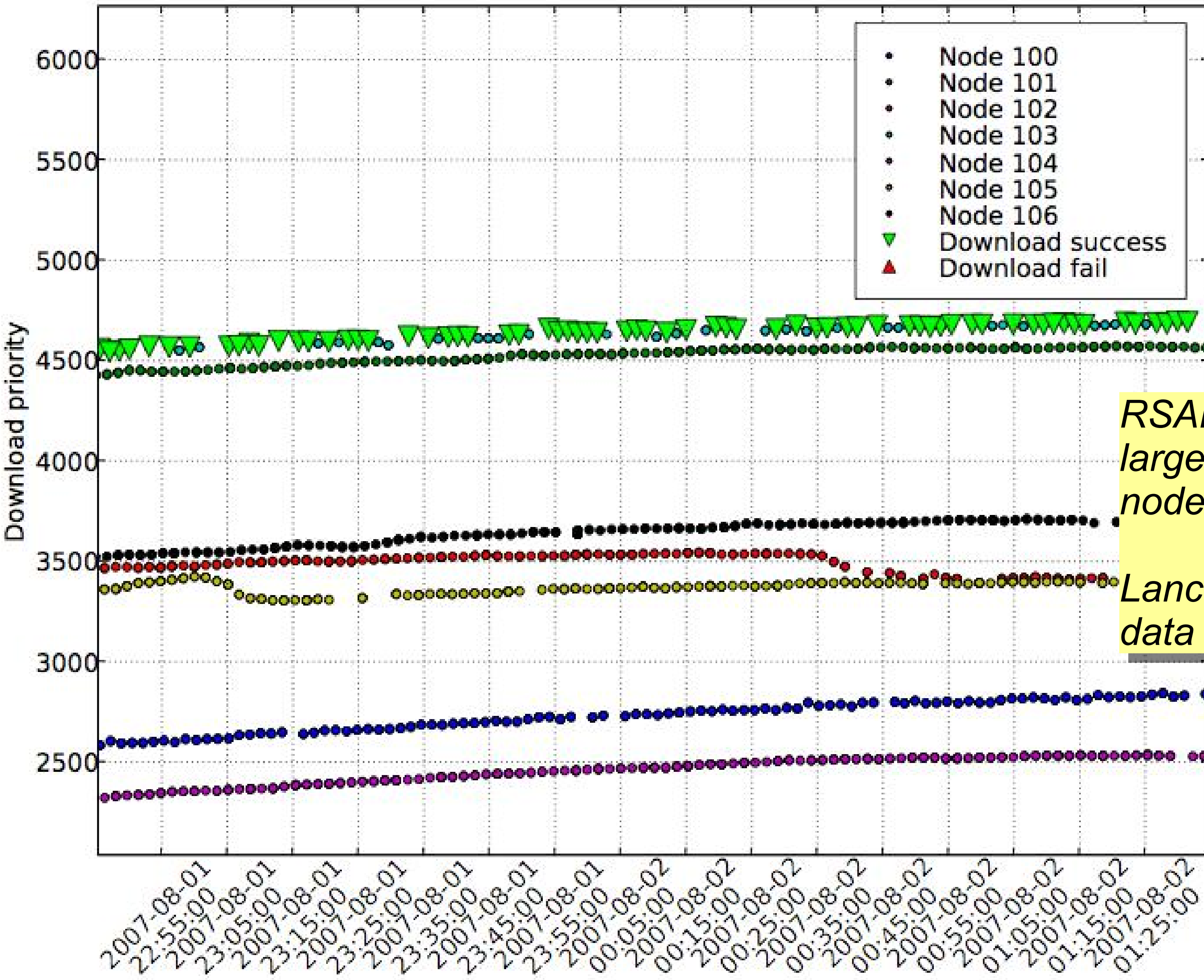
- Ran 8 sensor nodes for a total of 71 hours
  - Lance used to manage storage and bandwidth
  - Experimented with different prioritization functions and policy modules
- Successfully downloaded 1232 ADUs (77 MB of data)
  - 308 downloads failed due to timeouts: success rate 80%
- Total storage summaries span 11012 ADUs (688 MB of data)
  - Lance downloaded 11% of the data acquired by the network
- No significant node outages observed

# Bandwidth Breakdown




*Management traffic consumes less than 5% of overall bandwidth*

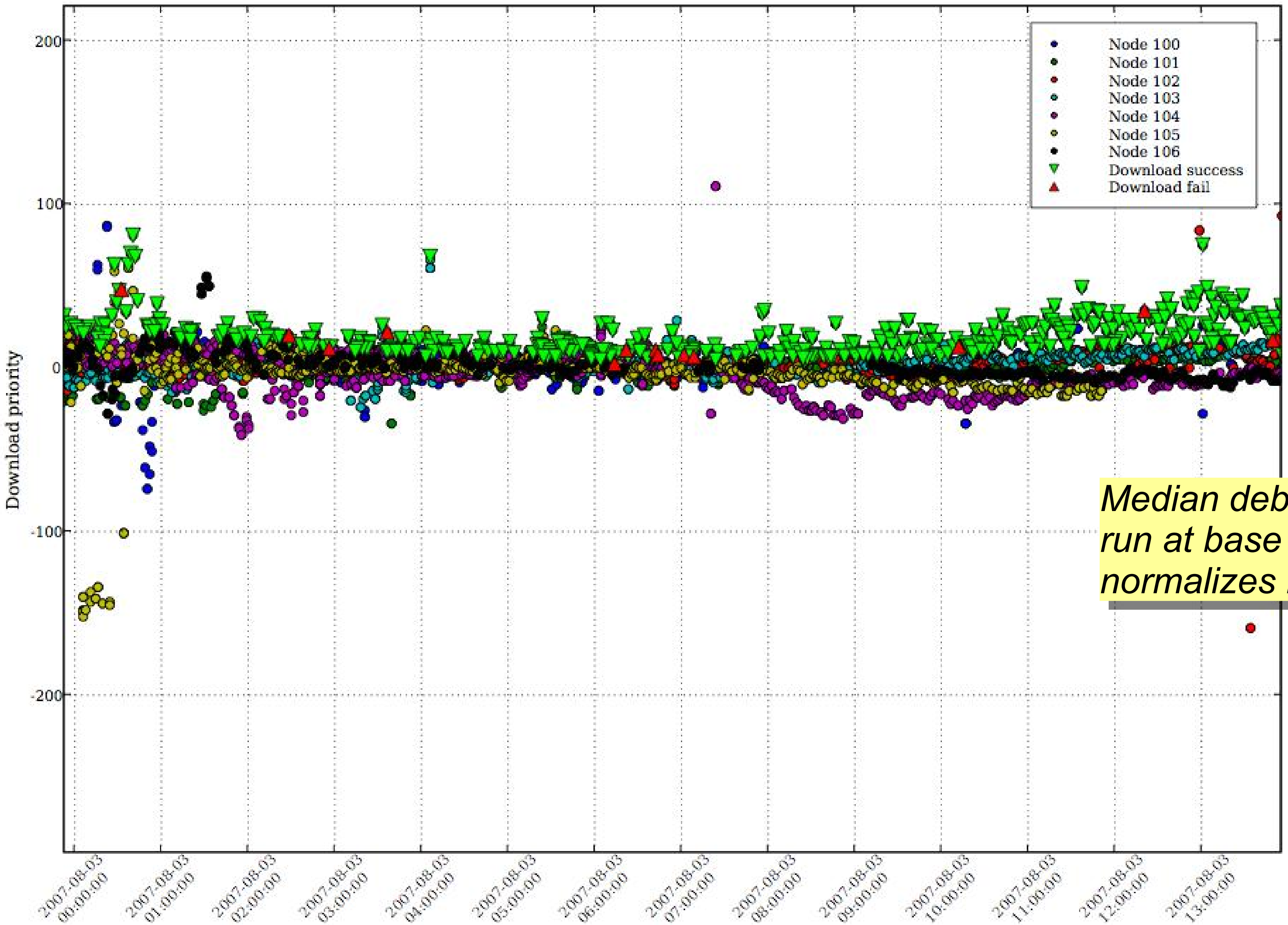
# RSAM prioritization: DC bias




*RSAM values exhibited large DC bias across nodes.*

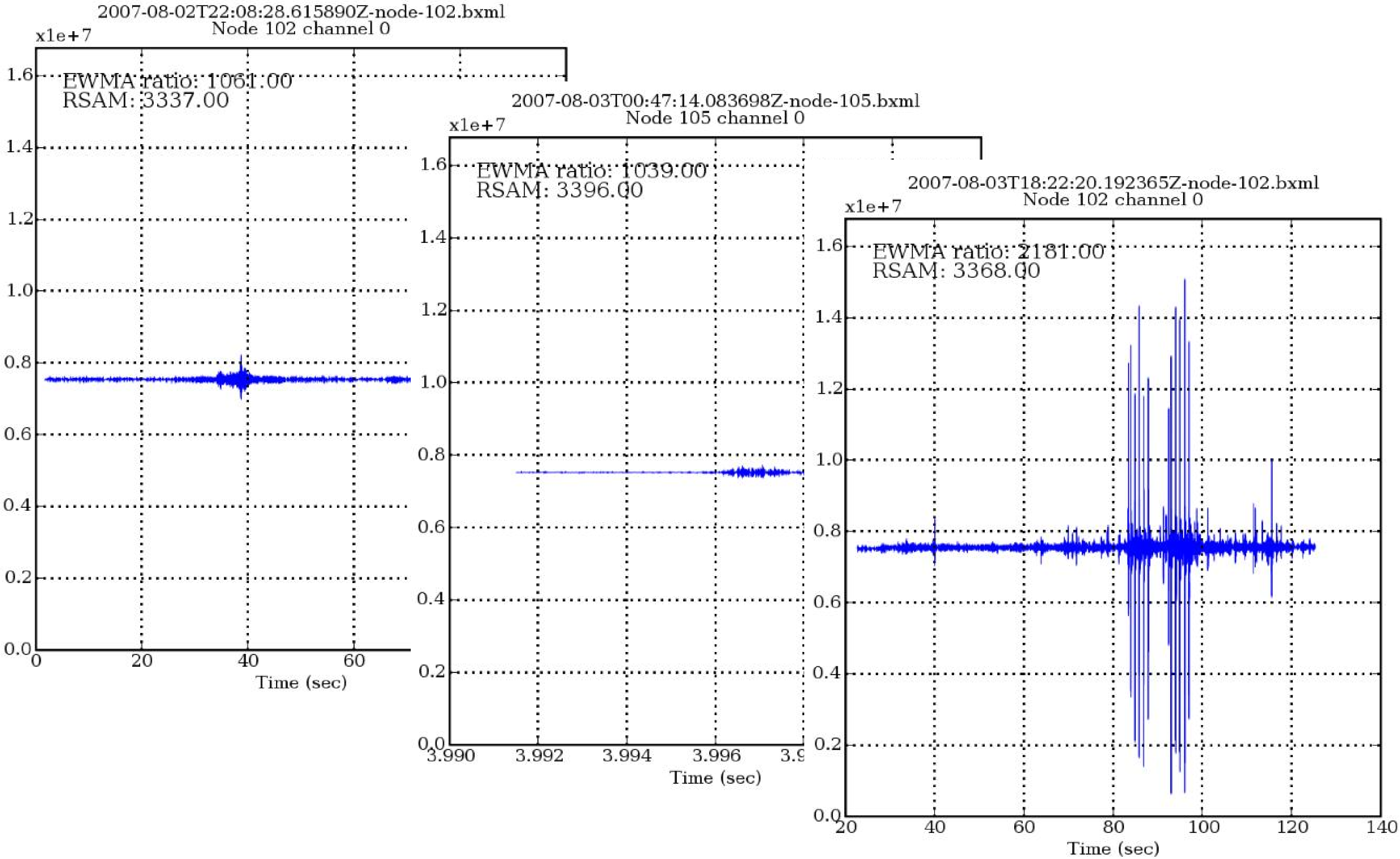
*Lance only downloaded data from 1 or 2 nodes!*

# The fix: Debiasing policy module




*Median debiasing filter run at base station normalizes RSAM values*

# We fixed the volcano...



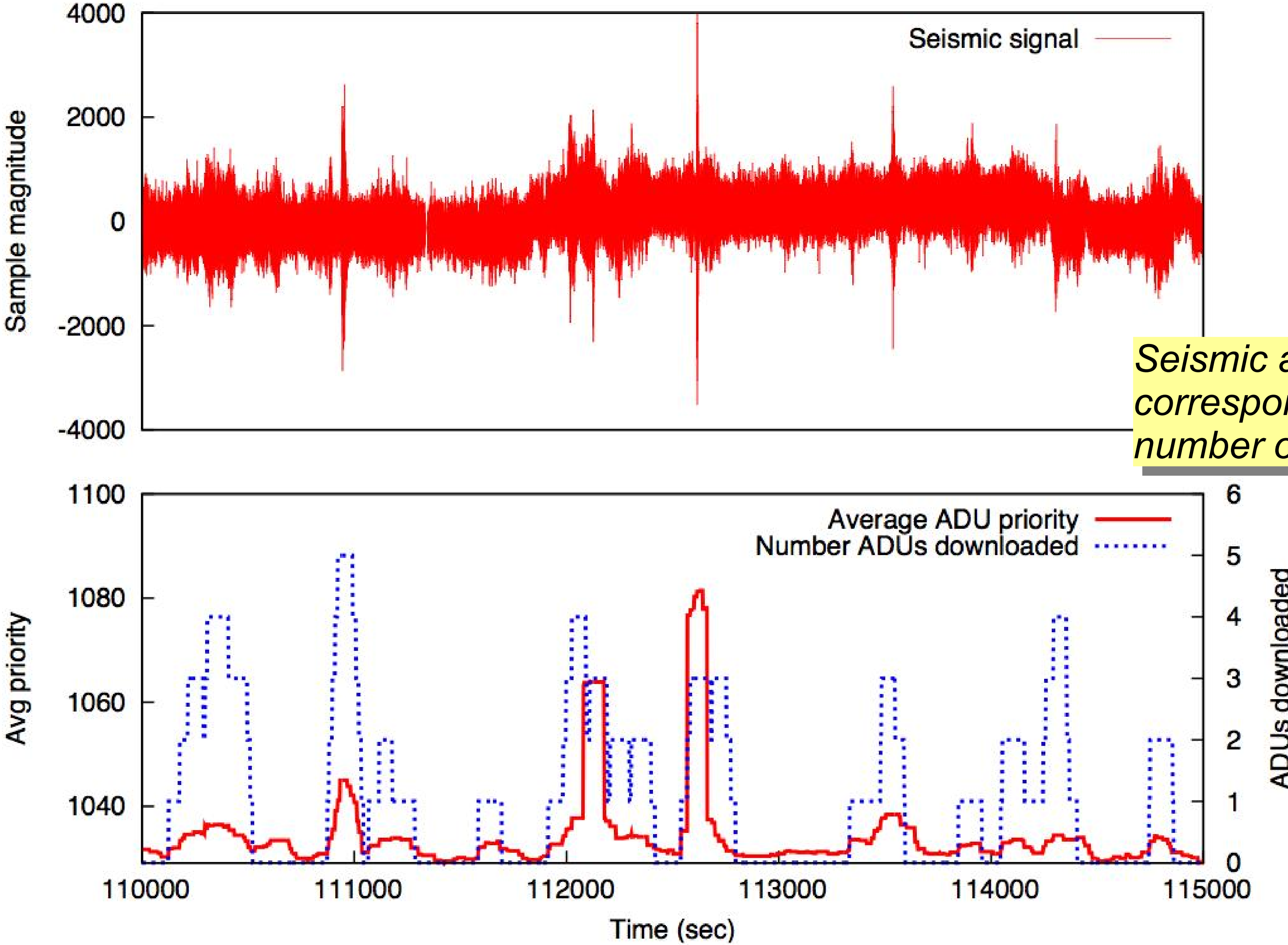
# How did RSAM perform?

- Volcanic activity was unusually low during the deployment
  - Only about 20 small earthquakes, no explosions
  - Week before, activity was much higher, with numerous explosions and dozens of earthquakes a day
- Took 8483 ADU summaries received after applying RSAM filter
  - Covering about 16 hours of the deployment
  - Computed optimal set $\Omega$ and weighted coverage $K(S,\Omega)$ for the downloaded data
- Results: Optimal set $\Omega$ included 393 ADUs. Lance downloaded 418.
  - Why more? Lance may have downloaded from nodes with faster transfer bitrates than optimal set may have chosen
  - Weighted coverage of **73%**
    - *(FIFO would have achieved only 51%)*
  - Lower than simulations, probably due to lack of variation in ADU priorities

# EWMA prioritization function

- Reprogrammed nodes after 25 hours to try to trigger on earthquakes
  - Still, we observed only 9 discernible events after the reprogram.
- 11012 ADU summaries received during this time
  - Optimal system would have downloaded 554 of them.
  - Lance downloaded 518. Weighted coverage of **80%.**
    - *Fifo system would have achieved 50%*

# EWMA prioritization behavior




*Seismic activity corresponds to increased number of downloads*

# Future Directions

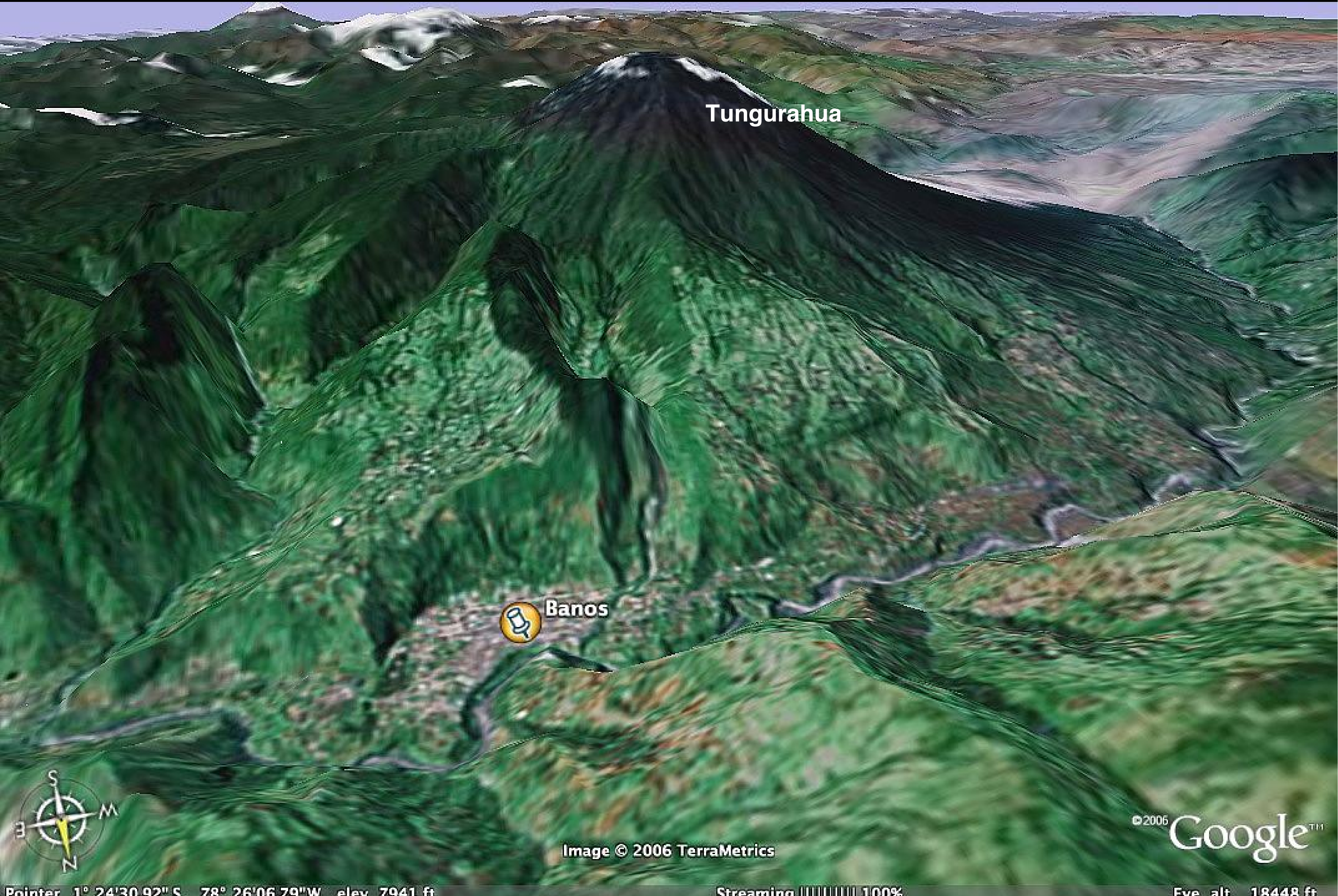
- Exploring more powerful sensor node platforms
  - Xscale based platforms (e.g., iMote2) offer significant horsepower for modest energy cost
- Extend to multitier networks
  - Microservers in the field for local data collection and processing
- Move beyond data collection to in-network computation
  - Explore cost/fidelity tradeoff between raw data and extracting higher-level features
- New application domains
  - e.g., Biomedical monitoring, structural/bridge monitoring, acoustic applications
  - Limb motion analysis of patients with Parkinson's Disease (with Spaulding Rehabilitation Hospital, Boston)

# Conclusions

- Wireless sensor networks can be used for data-intensive applications
  - But, radio bandwidth and storage are precious and must be managed carefully!
- Lance provides a flexible framework for maximizing network efficiency
  - Driven by application-defined *prioritization* of data
  - Node-local prioritized storage management
  - Network-wide download management
  - Policy modules enable
- Lance achieves highly efficient management of limited resources
  - Simulations: > 95% efficiency for wide range of storage capacities, bandwidths, and data distributions
  - Real deployment: efficiency of 73-80%, possibly hampered by low level of volcanic activity

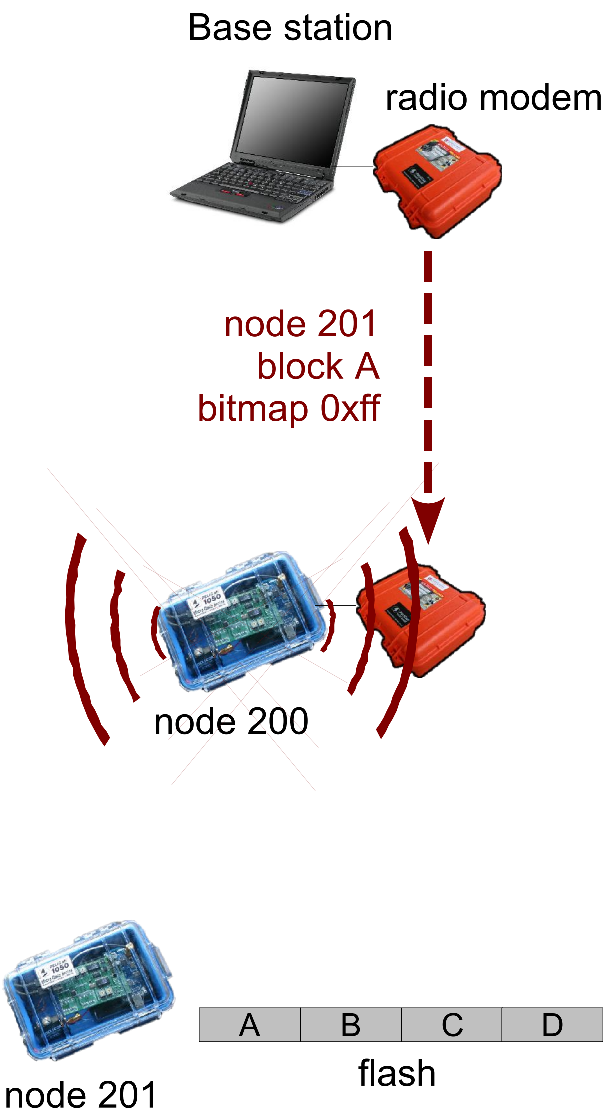
# Thank you!

Source code, hardware designs, and data sets available:
**http://www.eecs.harvard.edu/~mdw/proj/volcano**


Tungurahua
Banos
Image © 2006 TerraMetrics
©2006 Google™
Pointer 1° 24'30.92" S 78° 26'06.79"W elev 7941 ft
Streaming |||||||||| 100%
Eye alt 18448 ft

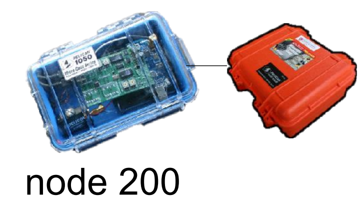
# Fetch Reliable Transfer Protocol

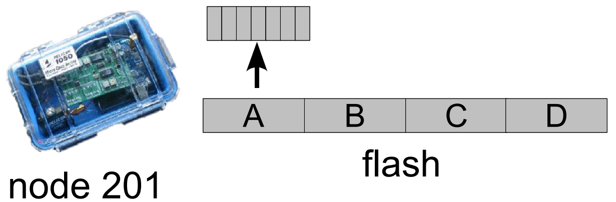
- Base station generates request containing:
  - node ID, block ID, bitmap of needed chunks
- Intermediate nodes flood request to network
  - Eliminates need for forward routing path from base
- 



# Fetch Reliable Transfer Protocol

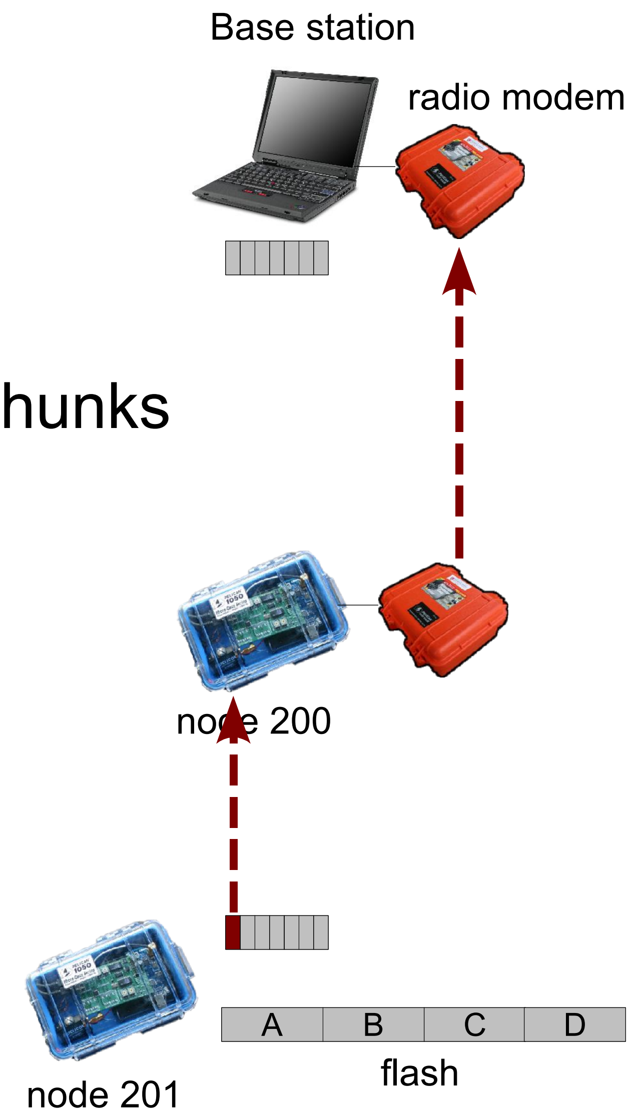
- Base station generates request containing:
  - node ID, block ID, bitmap of needed chunks
- Intermediate nodes flood request to network
  - Eliminates need for forward routing path from base
- Target node reads data from flash, breaks into chunks
  - One chunk per radio message (32 bytes of payload)
- 

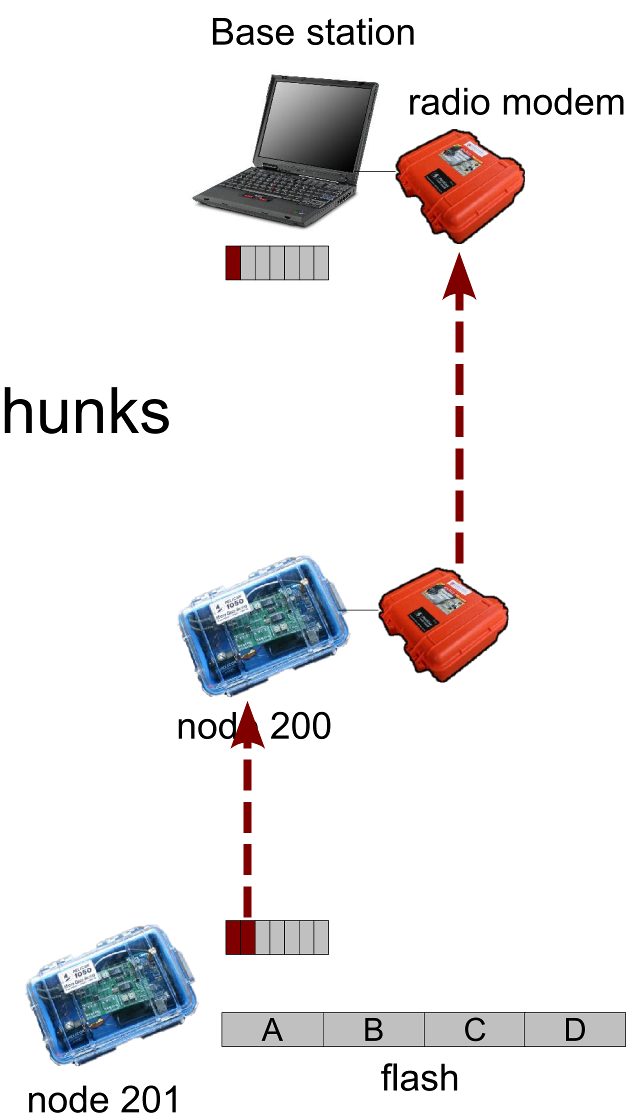
# Fetch Reliable Transfer Protocol

- Base station generates request containing:
  - node ID, block ID, bitmap of needed chunks
- Intermediate nodes flood request to network
  - Eliminates need for forward routing path from base
- Target node reads data from flash, breaks into chunks
  - One chunk per radio message (32 bytes of payload)
  - Route each chunk to base over multihop path
-



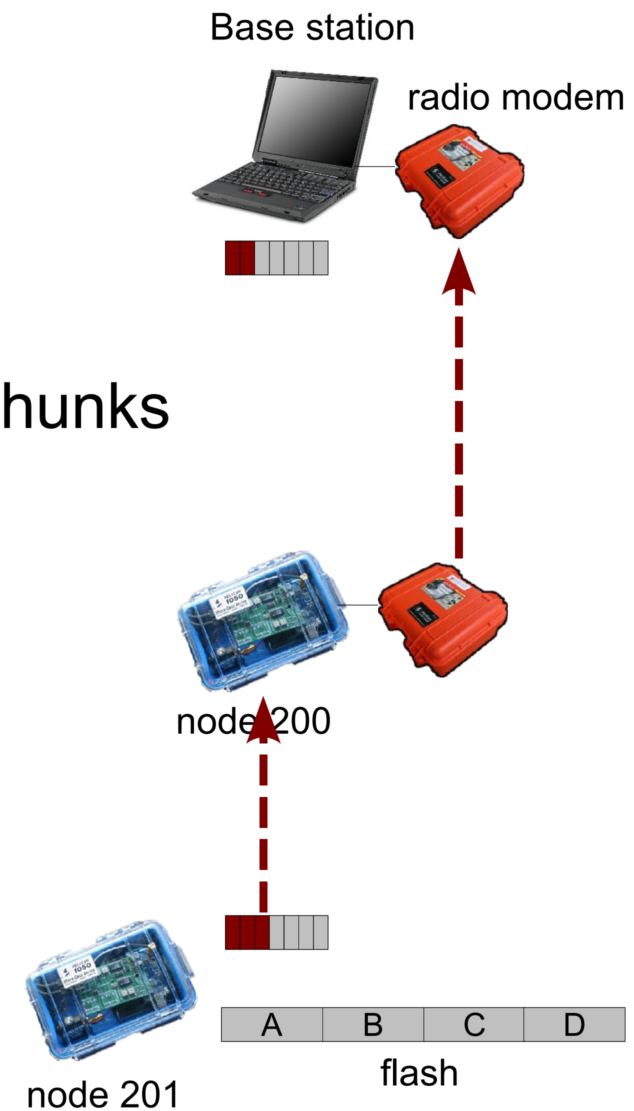
# Fetch Reliable Transfer Protocol

- Base station generates request containing:
  - node ID, block ID, bitmap of needed chunks
- Intermediate nodes flood request to network
  - Eliminates need for forward routing path from base
- Target node reads data from flash, breaks into chunks
  - One chunk per radio message (32 bytes of payload)
  - Route each chunk to base over multihop path
-



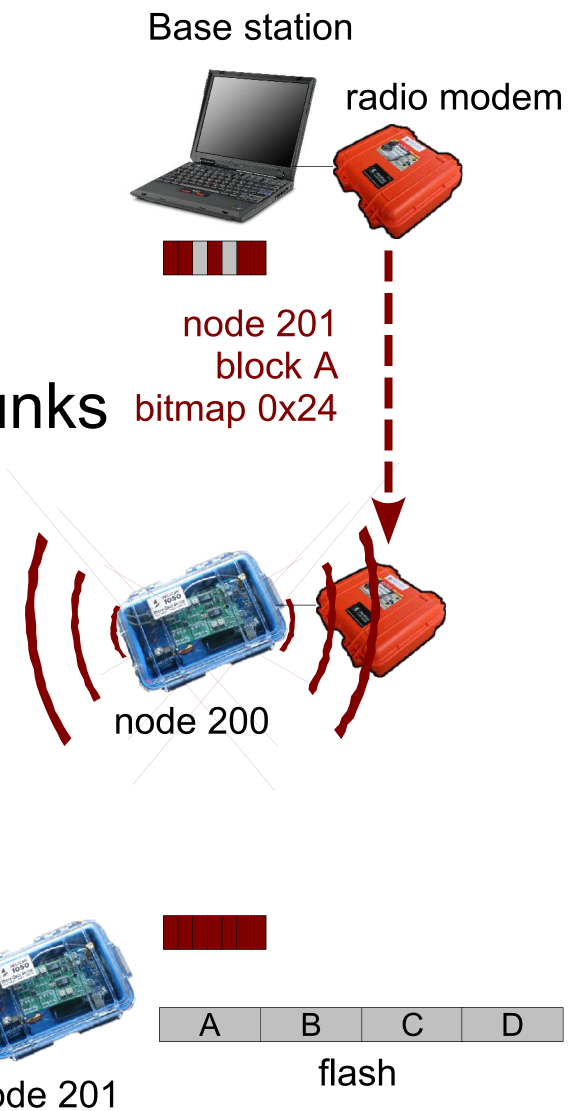
# Fetch Reliable Transfer Protocol

- Base station generates request containing:
  - node ID, block ID, bitmap of needed chunks
- Intermediate nodes flood request to network
  - Eliminates need for forward routing path from base
- Target node reads data from flash, breaks into chunks
  - One chunk per radio message (32 bytes of payload)
  - Route each chunk to base over multihop path
- 



# Fetch Reliable Transfer Protocol

- Base station generates request containing:
  - node ID, block ID, bitmap of needed chunks
- Intermediate nodes flood request to network
  - Eliminates need for forward routing path from base
- Target node reads data from flash, breaks into chunks
  - One chunk per radio message (32 bytes of payload)
  - Route each chunk to base over multihop path
- Base requests missing chunks after timeout
-



# Fetch Reliable Transfer Protocol

- Base station generates request containing:
  - node ID, block ID, bitmap of needed chunks
- Intermediate nodes flood request to network
  - Eliminates need for forward routing path from base
- Target node reads data from flash, breaks into chunks
  - One chunk per radio message (32 bytes of payload)
  - Route each chunk to base over multihop path
- Base requests missing chunks after timeout
- Node responds with missing data
-