Programming and Architecting Embedded Networked Systems

Om Gnawali, Ben Greenstein, Ramakrishna Gummadi, Ki-Young Jang, August Joki, Nupur Kothari, Jeongyeup Paek, Marcos Vieira,
Deborah Estrin, Ramesh Govindan, Eddie Kohler, Todd Millstein

Tenet: An Architecture for Tiered Embedded Systems

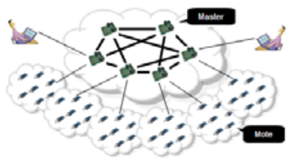
Problem Statement

- **Large-scale sensor network deployment will be tiered**

  Motes enable flexible deployment and dense instrumentation, while 32-bit ***Masters*** have greater network bandwidth and computing resources.
- **Multi-mote data fusion leads to fragile and unmanageable systems**

  Applications are hard to develop and debug, and reduces the re-usability of the mote tier.

  Can we take advantage of tiered systems to improve manageability and robustness of the overall system?



Approach

- **The Tenet Principle:**

Multi-node data fusion functionality and application logic should be implemented only in the master tier. The cost and complexity of implementing this functionality in a fully distributed fashion on motes outweighs the performance benefits of doing so.

- **Tenet applications run on Masters**

  Masters ***task*** motes. Motes sense and ***locally process*** generated sensor data. Results are delivered to the application program which can then fuse the results, and re-task the motes or trigger other sensing modalities.

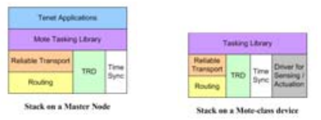
```
Sample(1000ms, 20, REPEAT, 1, ADC0, LIGHT)
        -> ClassifyAmplitude(99, 1, LIGHT, ABOVE)
        -> StampTime(TIME, LOCAL)
        -> LinkAttributes(LIGHT, TIME, AND)
        -> SendPtr()
```

Benefits

- **Simplifies application development**

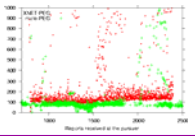
  Task library's tasklets can be flexibly composed for different apps
- **Re-usable generic mote tier**

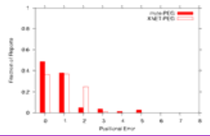
  Multiple applications/tasks can run concurrently
- **Robust and scalable network sub-system**



- **Case study: Pursuit–evasion application**

  Comparable performance with only 12% additional overhead





Vango: Systems Support for High Data-Rate Sensing

Motivation

- **Data-processing requirements are dynamic**

  Researchers don't know the best way to filter data; haven't seen such spatially dense data before.

  Ambient noise changes with ***environmental conditions***.

  Placement of a sensor affects its response to stimulation.
- **The physical environment is unpredictable**

  Climatic and physical variation affects ***RF availability***

  Deployment density affects link availability, contention, and network congestion.
- **Capturing high-rate phenomena requires calibrated node-local in-network filtering**

  Each node produces lots of data and best effort collection leads to collisions and significant data loss.

Approach

- **Application life-cycle oscillates between efficiency and experimentation phases**

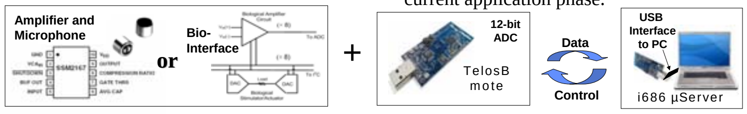
  For calibration, hypothesis tests, and pattern searches, it's best to collect representative waveforms to masters.

  Given bandwidth limitations, best to transfer data processing to sensor nodes to return as much interesting information as possible.
- **Vango components**

  Filters to measure, transform, and interpret data.

  Simple way to connect filters (linearly) across platforms to serve an application task.

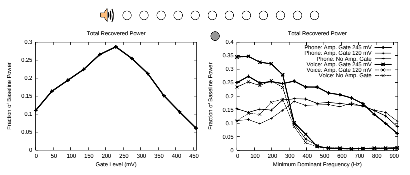
  Control mechanism to activate and configure processing on the mote or master tier, depending on current application phase.



Applications

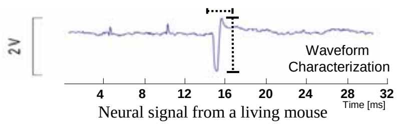
- **Auricle (acoustics)**

  Improving application performance requires balancing user-specific data filtering with bandwidth availability.



- **Neuromote (neural signals)**

  Runs with ***same*** Vango software as acoustics.



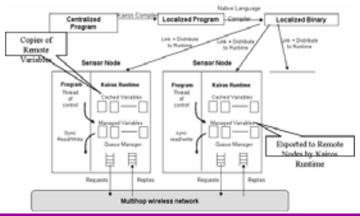
Neural signal from a living mouse

Kairos: Macroprogramming Wireless Sensor Networks

Motivation

- Macroprogramming: Allow all nodes to be programmed as a single unit
- Global program behavior captured as a single sequential task on a centralized memory model
- No need for explicit parallelization or synchronization code
- Challenge is designing the compiler and runtime components that generate and implement an equivalent concurrent distributed version …

Kairos Architecture

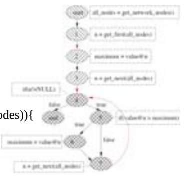


Approach

- Main programming primitives in Kairos: `get_neighbors(node)` to obtain current one-hop neighbors of a node, `var@node` to access node-local data, a concurrent version of the `for` statement
- Kairos compiler partitions the central program into *nodecuts*, each of which is executed entirely at a single node
- Kairos runtime orchestrates the execution of each successive nodecut, fetching the remote variables needed by them, and migrating the computation from node to node as necessary

Example

```
#include "kairos.h"
int maximum;
node-local int value;
void module_max(){
    nodelist all_nodes=get_nextwork_nodes();
   for(n=get_first(all_nodes);n!=NULL;n=get_next(all_nodes)){
        if(value@n > maximum)
            maximum = value@n;
}   }
```



Benefits

- Ease of programming (Kairos extends C)
- Kairos code almost a third the size of nesC code
- No user level synchronization code needed

Ongoing and future work

- Mote implementation: A Kairos compiler which can output nesC + Kairos runtime for motes
- Generic Failure Recovery: Automated recovery mechanisms in presence of various classes of failures
- Various levels of performance optimizations
- Exploiting Heterogeneity, Hierarchy, and User-level Energy/Resource Management