Macroprogramming Heterogeneous Sensor Networks Using cosMOS

Asad Awan, Ahmed Sameh, Ananth Grama

Coordinated Systems Lab.
Purdue University
http://www.cs.purdue.edu/pdsl

Acknowledgement: ITR/DDDAS -
Wireless Sensor Networks

- Large-scale self-organized network of tiny low-cost nodes with sensors
  - Resource constrained nodes
  - Highly heterogeneous
  - Dynamic
    - Network membership
    - Data load
  - Performance and scalability requirements

- Critical applications

**Challenge:** programming the “network” to efficiently collect and process data
Structural Health Monitoring

Setting of calibration tests

- Performance
- Scale
- Accuracy
- Cost
The traditional approach to distributed programming involves writing “network-enabled” programs for each node.

- The program encodes distributed system behavior using complex messaging between nodes.
- This paradigm raises several issues and limitations:
  - Program development is time consuming.
  - Programs are error prone and difficult to debug.
  - Lack of a distributed behavior specification, which precludes verification.
  - Limitations w.r.t. scalability, heterogeneity and performance.
Macroprogramming WSNS

- Macroprogramming entails direct specification of the distributed system behavior in contrast to programming individual nodes.
- Provides:
  - Seamless support for heterogeneity
    - Uniform programming platform
    - Node capability-aware abstractions
    - Performance scaling
  - Separating the application from system-level details
  - Scalability and adaptability with network & load dynamics
  - Validation of behavioral specification
Realizable Macroprogramming

• **High-level abstractions vs. low-footprint and flexibility**
  – Low-overhead execution of macroprograms
    • `COSMOS` is specifically designed to provide a low-footprint runtime
  – Providence for domain-specific performance optimization through system-level services

• **Macroprogram composition**
  – Reusable components

• **Support for over-the-air reprogramming**
  – Ability to modify distributed system behavior
  – Reprogrammable system-level services separate from the application
Objective

To develop a second generation operating system suite that facilitates rapid macroprogramming of efficient self-organized distributed applications for WSN
Outline

• Overview and Challenges
• Related work
• Our approach
• Evaluation
• Current status
• Future directions
Related Work

- **TinyOS**
  - Low footprint: applications and OS are tightly coupled
  - Costly reprogramming: update complete node image
  - Scripting languages TinyScript*, Mottle*, SNACK
  - **Maté** – application specific virtual machine
    - Event driven bytecode modules run over an interpreter
    - Domain specific interpreter
    - Very low cost updates of modules
    - Major revision requires costly interpreter updates
    - Easy to program using simple scripting languages*
Related Work

• **SOS**
  – Interacting modules compose an application
  – OS and modules are loosely coupled
  – Modules can be individually updated: low cost
  – Larger number of runtime failure modes

• TinyOS and SOS are both node operating systems
Related Work

• **TinyDB**
  – An application on top of TinyOS
  – Macroprogramming using SQL queries
  – Limitations in behavioral specifications (due to implementation)
  – Difficult to add new features or functionality
  – Larger footprint and heavyweight
Related Work

• High level macroprogramming languages
  – Functional and intermediate programming languages
    • Region stream, abstract regions, HOOD, TML (DTM)
  – Programming interface is restrictive and system mechanisms can not be tuned
  – No mature implementations exist, no performance evaluation is available
  – Compile down to native OS: can compile down to cosmOS
Related Work

• Dynamic resource allocation
  – Impala
    • Rich routing protocols
    • Rich software adaptation subsystem
    • Aimed at resource rich nodes
  – SORA
    • Self-organized resource allocation architecture

– Complimentary to our work
• Challenges
• Related work
• **COSMOS Design Principles**
• Evaluation
• Current status
• Future directions
Design Principles

- Macroprogram centric OS design
- Network viewed as an abstract data processing machine
  - Producers, processors, and consumers
  - Dynamic self-organized system
- Macroprogram:
  - Composed of modules
  - Specification can be statically validated
- Application data and control flow
  - Transparently supported by OS
  - Asynchronous data flow semantics
- Data processing
  - Opaque to the OS
Design Principles

• Load conditioning
  • Handling network and load dynamics in a self-organized system

• Heterogeneity
  • Performance scaling
  • Node capability-aware

• Flexible reprogramming
  • Can change system services, application modules, as well as control flow
Macroprogramming Model

• Macroprogram consists of:
  • Distributed system behavioral specification
  • Constraints associated with mapping behavioral specification to physical system

• Behavioral Specification
  – *Functional Components (FCs)*
    • Represents a specific data processing function
    • Typed input and output interface
  – *Interaction Assignment (IA)*
    • Directed graph that specifies data flow through FCs
      – Data flow through *instances* of FCs transparently handled by COSMOS asynchronous data channels
    • Data source and sinks are (logical) *device ports*
      – With typed input or output
Program Correctness

• **Statically type-checked interaction assignment**
  • The output of a component can be connected to the input of another only if their types match

• **Functional components represent a deterministic data processing function**
  • The output sequence depends only on the inputs to the FC

• **Correctness**
  • Given input at each source in the IA the outputs at sinks are deterministically known
Functional Components

- **Elementary unit of execution**
  - Isolated from the state of the system and other FCs
  - Uses only stack variables and statically assigned state memory
  - Asynchronous execution: data flow and control flow handled by cosmOS

- **Static memory**
  - Prevents non-deterministic behavior due to malloc failures
  - Leads to a lean memory management system in the OS

- **Reusable components**
  - The only interaction is via typed interfaces

- **Dynamically loadable components**
  - Runtime updates possible
Functional Components

- **Programmatically:**
  - Declaration in cosmOS language
    - GUID: globally unique identifier associated with an FC
    - Interface definition as an enumerable ordered set
  - C code that implements the functionality
    - No platform dependencies in the code
      (platform dependencies are encapsulated in device ports)
  - GUID ties the two
- **For an application developer only the declaration is important**
  - Assumes a repository of implementation
  - Binaries compiled for different platforms

```c
%fc_dplex: {
    fcid = FCID_DPELX,
    in [ raw_t, raw_t ],
    out [raw_t]
};
```
Mapping Constraints

- **COSMOS** views heterogeneous nodes as named capability-based sets of nodes

- **Application developer provides constraints on mapping**
  - For both device ports and FCs
  - A mask of required node capabilities

- **Used by the OS to provide data routing**

```plaintext
@ sensor_nodes = CAP_PHOTO_SESNOR : photo, thresh
```
COSMOS LANGUAGE

- Sections:
  - Enumerations
  - Declarations
  - Mapping constraints
  - IA Description

- Implemented using Lex & Yacc
A Simple Application

%photo : device = PHOTO_SENSOR, out [ raw_t ];
%fs : device = FILE_DUMP, in [ * ];
%avg : { fcid = FCID_AVG, in [ raw_t, avg_t ], out [ avg_t ] };
%thresh : { fcid = FCID_THRESH, in [ raw_t ], out [ raw_t ] };
@ snode = CAP_PHOTO_SENSOR : photo, thresh;
@ fast_m = CAP_FAST_CPU : avg;
@ server = CAP_FS | CAP_UNIQUE_SERVER : avg, fs;
start_ia
timer(100) → photo(1);
photo(1) → thresh(2,0,500);
thresh(2,0) → avg(3,0,10), avg(4,0,100);
avg(3,0) → fs(5) | → avg(3,1);
avg(4,0) → fs(6) | → avg(4,1);
end_ia
A Simple Application

\[ P(\cdot) \xrightarrow{\text{raw}_t} \text{Threshold (500)} \xrightarrow{\text{raw}_t} \text{Average (10)} \xrightarrow{\text{avg}_t} \text{Average (100)} \xrightarrow{\text{avg}_t} \* \xrightarrow{\text{FS}} \text{FS} \]

\[ T(t) \]

\[ \text{Threshold (500)} \]

\[ \text{Average (10)} \]

\[ \text{Average (100)} \]

\[ \text{FS} \]

\[ \text{Threshold (500)} \]

\[ \text{Average (10)} \]

\[ \text{Average (100)} \]

\[ \text{FS} \]
Runtime Model

• **Objective:** provide a low-footprint execution environment for

• **Key components**
  – Data flow and control flow
  – Locking and concurrency
  – Load conditioning
  – Routing primitives
Data Flow and Control Flow

- **Data driven model**
  - Asynchronous arrival of data triggers component execution

- **Data channels implemented as output queues:**
  A separate queue for each output
  - No single queue bottleneck $\rightarrow$ concurrency
  - Attached at application initialization
    - Avoids runtime lookups and associated failure modes
  - Minimizes memory required by queue data $\leftarrow$ common case: multi-resolution / multi-view data
  - Data object encapsulates vector of data

- **Transparent network data flow**

- **Abstractions to allow synchronization of inputs**
Concurrency and Locking

- **COSMOS** supports both multi-threaded and non-preemptive environments.
- **Motes** have non-preemptive scheduling:
  - Locking through disabling interrupts
    - **COSMOS** design eliminates locking in data path.
- **On resource rich nodes**:
  - Multi-threading: concurrent FC execution
  - Scope of locks: interacting FCs
  - Locks are not held while processing
    - Input and output commit primitives
Load Conditioning

- Dynamic peering implies variable load
- Bounded memory on nodes
  \[ \rightarrow \text{need for load conditioning} \]
- Load conditioning
  - Reactive
    - Notification of queue sizes to FCs
  - Pro-active
    - Count virtual flows and offload excess to the network
    - \textbf{thresh}(2,0) \rightarrow [5] \rightarrow \text{avg}(3,0,10);
    - Load control implemented using an FC that can be reprogrammed by users
Routing Primitives

• Instead of providing a fixed routing scheme we abstract routing primitives required by the OS
  – `send_data(cap_mask, logical_fcid, appid, data)`
  – `group_send(cap_mask, pkt)`
    • Reliable multicast
    • Only used for application or system update

• Default implementation provided
  – Hierarchical tree routing
  – System service → can be reprogrammed
OS Design

- Each node has a static OS kernel
  - Consists of platform dependent and platform independent layers
- Each node runs service modules
- Each node runs a subset of the components that compose a macro-application

```
Updateable
User space

Static OS
Kernel

Platform Independent Kernel

Hardware Abstraction Layer

Services

App FC

App FC

App FC

Services

HW Drivers

HW Drivers

HW Drivers
```
Implementation

• Functional implementations for Mica2 and POSIX (on Linux)

• Mica2:
  • Non-preemptive function pointer scheduler
  • Dynamic memory management

• POSIX:
  • Multi-threading using POSIX threads and underlying scheduler
  • The OS exists as library calls and a single management thread
Implementation

• **Services:**
  • Have access to system-level management functions (system calls)
  • Can be run independently of application and manage single node performance

• **Extensibility:**
  • Core of the OS is platform independent
  • New devices can be easily added by implementing simple device port interface functions
  • Communication with external applications by writing a virtual device driver
  • Complex devices can use an additional service to perform low-level interactions
Outline

• Challenges
• Related work
• Our approach
• Evaluation
• Current status
• Future directions
Evaluation

- Remote sensing application
  - No processing $\rightarrow$ stresses OS
  - Operational range

Performance limited by hardware
Evaluation

• Remote sensing application
Evaluation

- Micro evaluation for Mica2 using AVRORA

- Comparison with SOS – a dynamic OS for motes

Slightly better than SOS, while providing a comprehensive macroprogramming environment
Evaluation

• Load conditioning
  – A 3pps FC on a processing node
  – 1pps per node received at processing node

Load conditioning enables efficient adaptation and self-organization in dynamic environments
Evaluation

- Effect of adding *NULL* FCs

Supports highly modular applications through low-footprint management of FCs
Outline

• Challenges
• Related work
• Our approach
• Evaluation
• Current status
• Future directions
WSN @ BOWEN

Pilot deployment at BOWEN labs

MICA2 motes with ADXL 202

Laser attached via serial port to Stargate computers

FM 433MHz

ECN Net

802.11b Peer-to-Peer

Internet

Currently laser readings can be viewed for from anywhere over the Internet (conditioned on firewall settings)
Current Status: OS

- We have completed an initial prototype of our operating system for AVR μc (Mica2) and POSIX (over Linux)

- Current activities
  - Exhaustive testing
  - Release: www.cs.purude.edu/~awan/cosmos/
Outline

• Challenges
• Related work
• Our approach
• Evaluation
• Current status
• Ongoing work
Ongoing Work

- Implement common data processing modules that can be reused
  - E.g., aggregation, filtering, FFT
- Further evaluation of deployment on a real-world large-scale heterogeneous test bed: BOWEN labs
  - Iteratively develop a DDDAS system for structural health monitoring
- High level functional programming abstractions, visual (WYSIWYG) application design utilities
Ongoing Work

• **Data processing challenges**
  • Spatial and temporal correlation of data from several independent sources
  • Processing of disparate measurement information to estimate/analyze the “actual” physical phenomenon

• **Exploring distributed algorithms:**
  • FC allocation
  • Routing strategies
  • Aggregation strategies

• **Exploring other application domains**

• **Formal characterization of COSMOS programming model**
Questions?

Thank you!

http://www.cs.purude.edu/~awan/cosmos/