EmStar: A Software Environment for Developing and Deploying Wireless Sensor Networks

Presented by Roy Shea
6 October 2004

Em* is joint work by a number of contributors including Alberto Cerpa, Jeremy Elson, Lewis Girod, Nithya Ramanthan, Thanos Stathoupoulos, and many others.
What is EmStar?

- Software environment for sensor networks built from Linux-class devices (microservers)
Microservers vs. Motes

- Microservers are much less constrained
- Hence they can be much more complex
  - Image, audio processing
  - More data storage
  - Higher algorithmic complexity
  - More intelligent behavior
- Yet, still embedded and distributed
  - Autonomous – no human caretaker
  - Distributed system – complex interactions
EmStar is Designed for WSNs

- Simulation and emulation tools
- Modular, but not strictly layered architecture
- Robust, autonomous, remote operation
- Fault tolerance within node and between nodes
- Reactivity to dynamics in environment and task
- High visibility into system: interactive access to all services
EmStar Transparently Trades-off Scale vs. Reality

- Em* code runs transparently at many degrees of “reality”: high visibility debugging before low-visibility deployment
EmStar Components

- Tools
  - EmRun
  - EmProxy/EmView
  - EmTOS

- Services
  - NeighborDiscovery/LinkStats
  - TimeSync/AudioServer
  - Routing

- Standard IPC
  - FUSD
  - Device Patterns
EmSim/EmCee

- EmStar supports a variety of types of simulation and emulation, from simulated radio channel and sensors to emulated radio and sensor channels (ceiling array).
- In all cases, the code is identical (sometimes even identical binaries).
- Multiple emulated nodes run in their own spaces, on the same physical machine.
- Nodes in sim/emulation do NOT know anything about other nodes in the system, except what they receive via sensors, radio, etc… just like in real life.
EmRun: Manages Services

- Designed to start, stop, and monitor services
  - Increases robustness, resilience, autonomy
- EmRun config file specifies service dependencies
- Starting and stopping the system
  - Starts up services in correct order
  - Respawn services that die
  - Can detect and restart unresponsive services
  - Notifies services before shutdown, enabling graceful shutdown and persistent state
- Error/Debug Logging
  - Per-process logging to in-memory ring buffers
  - Configurable log levels
EmView/EmProxy: Visualization
EmTOS: Support for Heterogeneous Systems

- Compile NesC Application
  - Platform “emstar”
  - Builds single EmStar module
- Wrapper Library
  - Provides TinyOS services
  - Enables NesC to provide new EmStar services
- Useful for deployment (ESS)
- Useful for simulation
  - Heterogeneous systems
EmStar Services

Collaborative Sensor Processing Application

State Sync

3d Multi-Lateralization

Topology Discovery

Acoustic Ranging

Neighbor Discovery

Leader Election

Reliable Unicast

Time Sync

Hardware

Radio

Audio

Sensors
Neighbor Discovery / LinkStats

- Neighbor Discovery Service
  - Maintains list of active neighbors
  - Hysteresis prevents neighbor flapping

- Link Statistics Estimation
  - Passively monitors traffic over radio
  - Adds sequence number to each packet
  - Detects gaps in sequence number

- Blacklisting (F. Silva)
  - Filters packets based on link quality
EmStar Service Lifecycle

- **Interface design:**
  - Encapsulate some useful mechanism
  - Expose the application-specific policy decisions

- **Choosing modularity:**
  - Don’t bite off too much at once
  - Something that at first looks simple can grow more complex
  - Don’t worry about efficiency of more modules.. Optimize later
  - **BUT**.. avoid “blue sky” modularity designs.. Instead, factor

- **Factoring:**
  - If a module is too complex, look for ways to break it down
  - New problems sometimes suggest new patterns
    - Factor new pattern libraries out of existing code
EmStar IPC Standards

```
[girod@kingfisher girod]$ cat /dev/sim/group7/node1/link/0/status
Link type: MoteNIC on /dev/tty00
Interface Addr: 0,0,0,16
MTU: 200
Stats:
   packets_rx: 6625
   packets_tx: 345
   bytes_rx: 1171883
   bytes_tx: 67415

Active: 1
Promisc: 0
PDT: 73
[girod@kingfisher girod]$ 
```
Interacting With EmStar

- Text/Binary on same device file
  - Text mode enables interaction from shell and scripts
  - Binary mode enables easy programmatic access to data as C structures, etc.

- EmStar device patterns support multiple concurrent clients
  - IPC channels used internally can be viewed concurrently for debugging
  - “Live” state can be viewed in the shell (“echocat –w”) or using emview
Inter-module IPC: FUSD

- Creates device file interfaces
- Text/Binary on same file
- Standard interface
  - Language independent
  - No client library required
- Uses DevFS if available
Status Device

- Designed to report current state
  - No queuing: clients not guaranteed to see every intermediate state
- Supports multiple clients
- Interactive and programmatic interface
  - ASCII output via “cat”
  - Binary output to programs
- Supports client notification
  - Notification via select()
- Client configurable
  - Client can write command string
  - Server parses it to enable per-client behavior
Packet Device

- Designed for message streams
- Supports multiple clients
- Supports queuing
  - Round-robin service of output queues
  - Delivery of messages to all, or specific clients
- Client-configurable:
  - Input and output queue lengths
  - Input filters
  - Optional loopback of outputs to other clients (for snooping)
Device Files vs. Regular Files

- **Regular files:**
  - Require locking semantics to prevent race conditions between readers and writers
  - Support “status” semantics but not queuing
  - No support for notification, polling only

- **Device files:**
  - Leverage kernel for serialization: no locking needed
  - Arbitrary control of semantics:
    - Queuing, text/binary, per client configuration
  - Immediate action, like an function call:
    - System call on device triggers immediate response from service, rather than setting a request and waiting for service to poll.
Conclusions: Why EmStar?

- Robustness / Fault Tolerance
  - Clients can’t crash Servers
  - Process Spaces
  - Soft State
- Need for Framework to build Microserver SW
  - IP stack / Traditional systems not a good fit
    - “NFS Server Timed Out”
  - Debug dynamic, distributed embedded systems?
    - More complex than many virtual / network systems
The End!
Thank you..