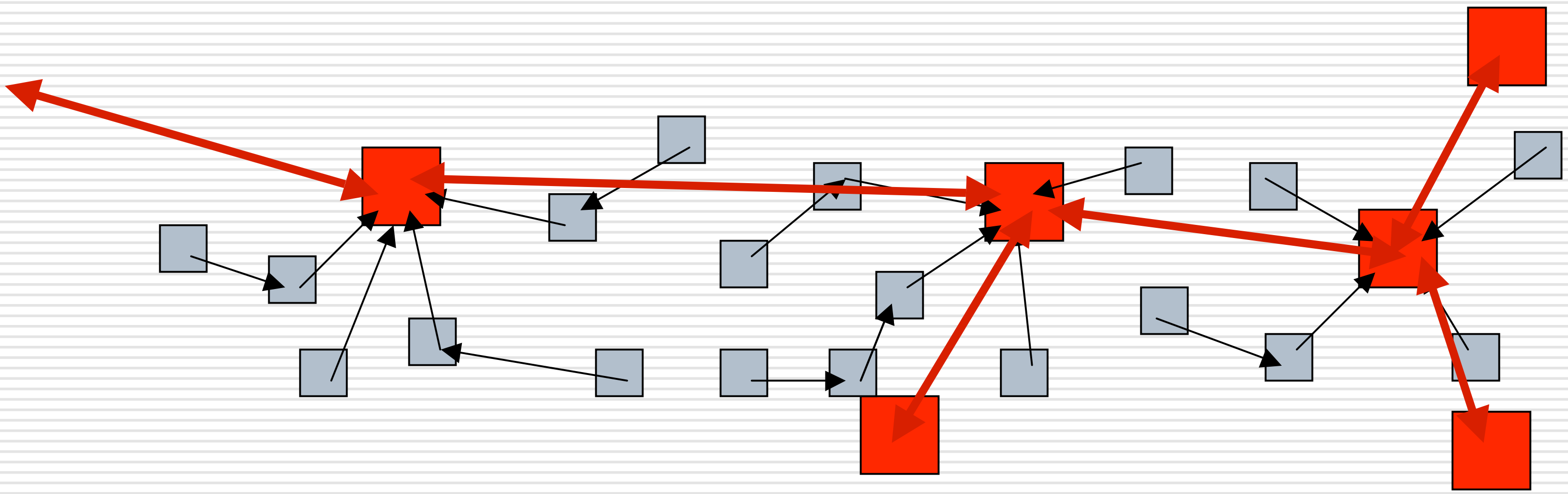
Em*: A Software Environment for Developing and Deploying Wireless Sensor Networks

© Dr. Deepak Ganesan, edited by Dr. Robert Akl

What is Em*?

- Software environment for sensor networks built from Linux-class devices (microservers)



Microservers vs. Motes

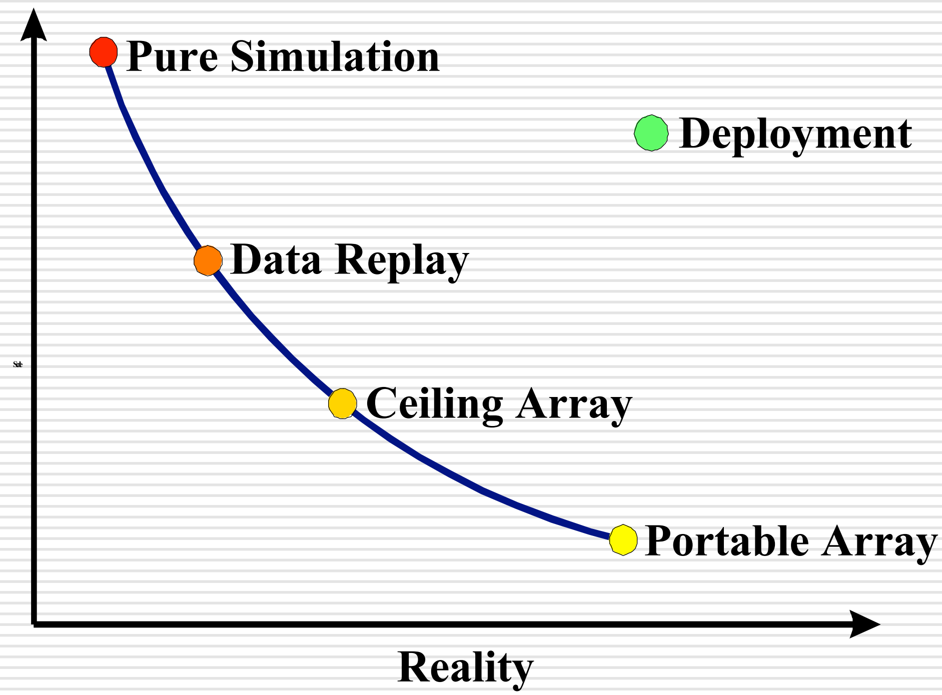
- Microservers are much less constrained
- Hence they can be much more complex
  - Image, audio processing
  - More data storage
  - Higher algorithmic complexity
  - More intelligent behavior
- Yet, still embedded and distributed
  - Autonomous – no human caretaker
  - Distributed system – complex interactions

Em* 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

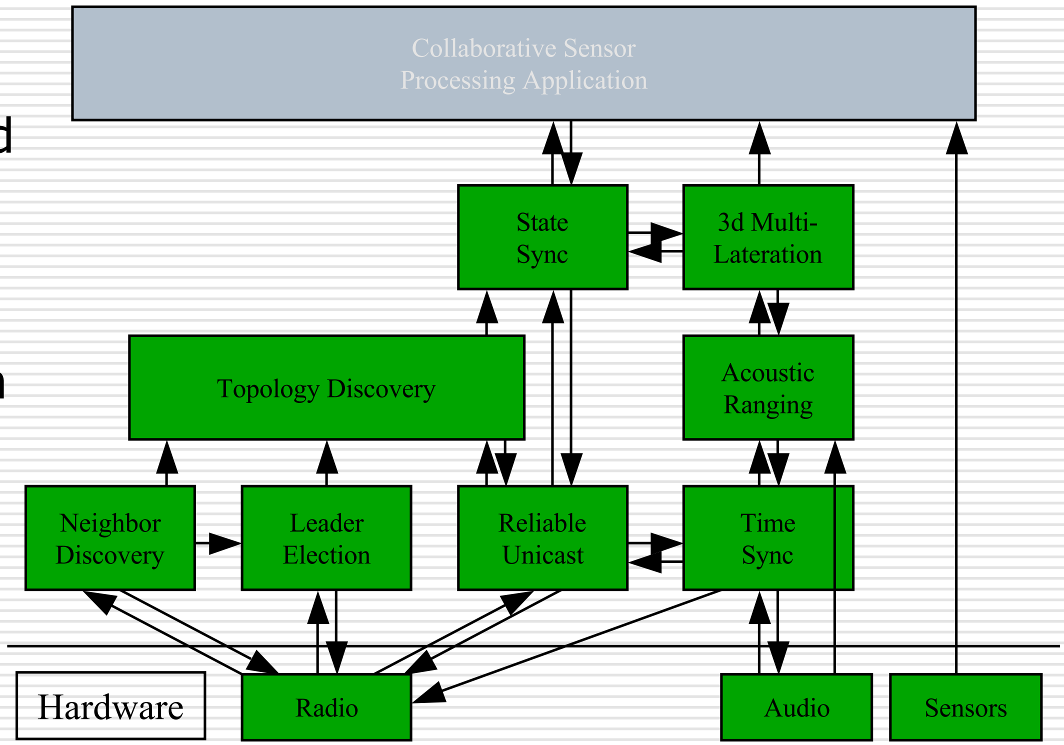
Em* Transparently Trades-off Scale vs. Reality

- Em* code runs transparently at many degrees of "reality": high visibility debugging before low-visibility deployment



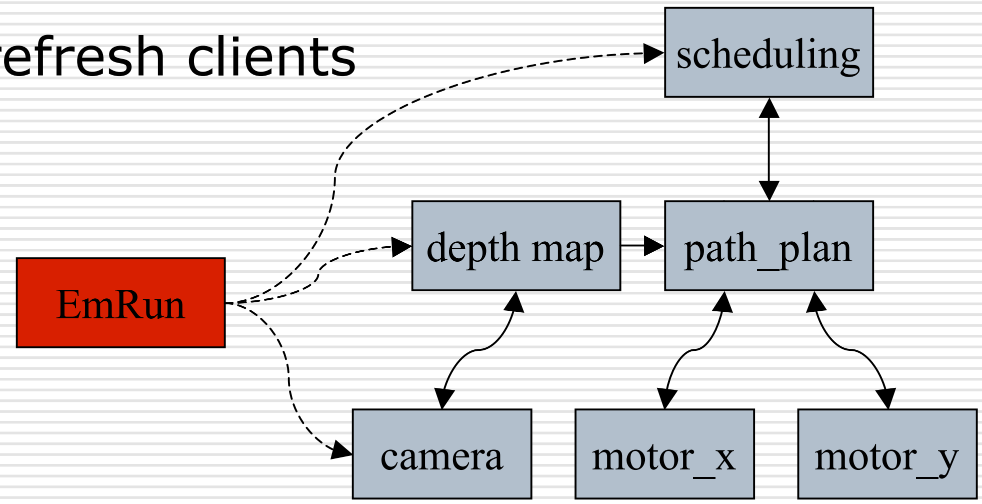
Em* Modularity

- Dependency DAG
- Each module (service)
  - Manages a resource and resolves contention
  - Well defined interface
  - Well scoped task
  - Encapsulate mechanism
  - Expose control of policy
  - Minimize work done by client library
- Application has same structure as "services"



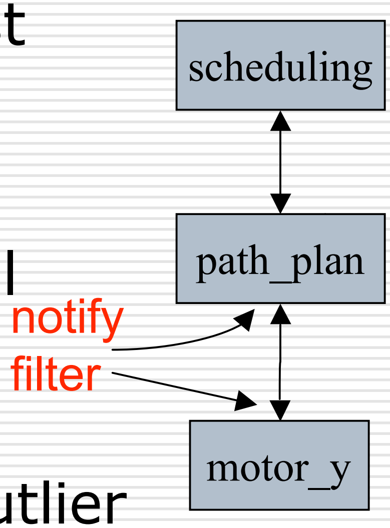
Em* Robustness

- Fault isolation via multiple processes
- Active process management (EmRun)
- Auto-reconnect built into libraries
  - "Crashproofing" prevents cascading failure
- Soft state design style
  - Services periodically refresh clients
  - Avoid "diff protocols"



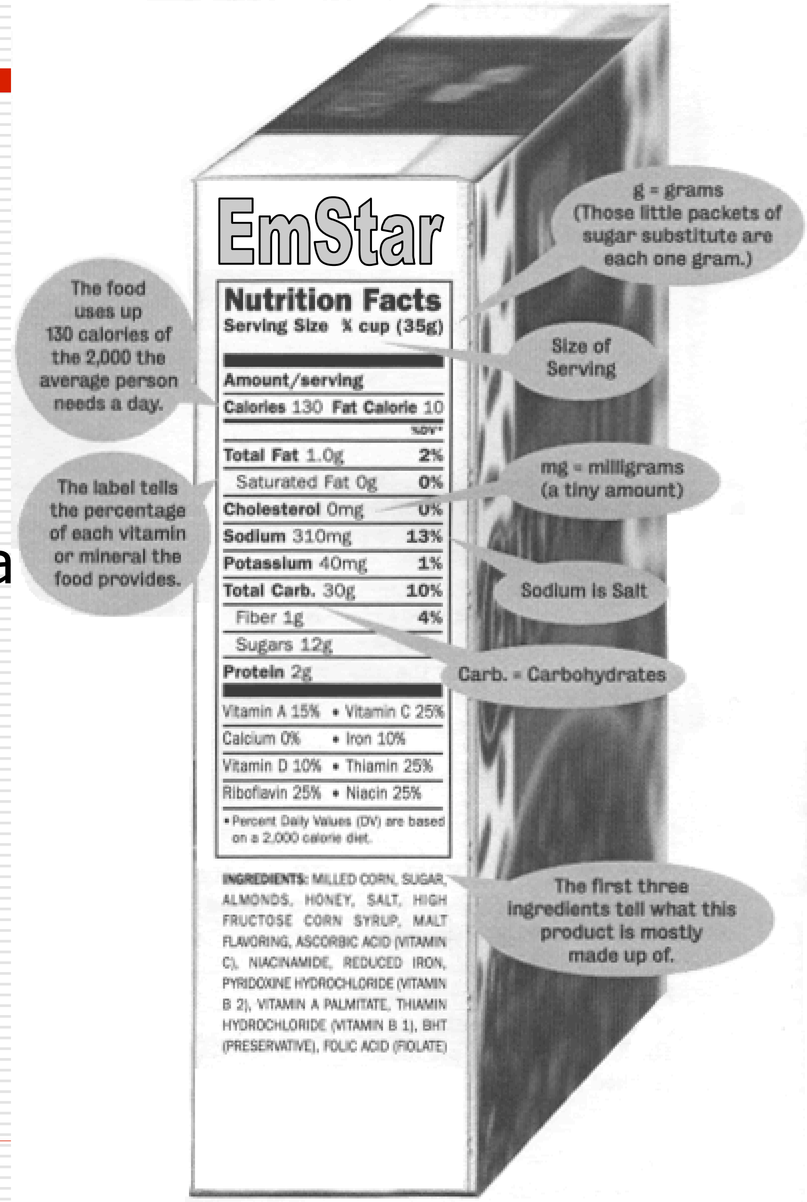
Em* Reactivity

- Event-driven software structure
  - React to asynchronous notification
  - e.g. reaction to change in neighbor list
- Notification through the layers
  - Events percolate up
  - Domain-specific filtering at every level
  - e.g.
    - neighbor list membership hysteresis
    - time synchronization linear fit and outlier rejection

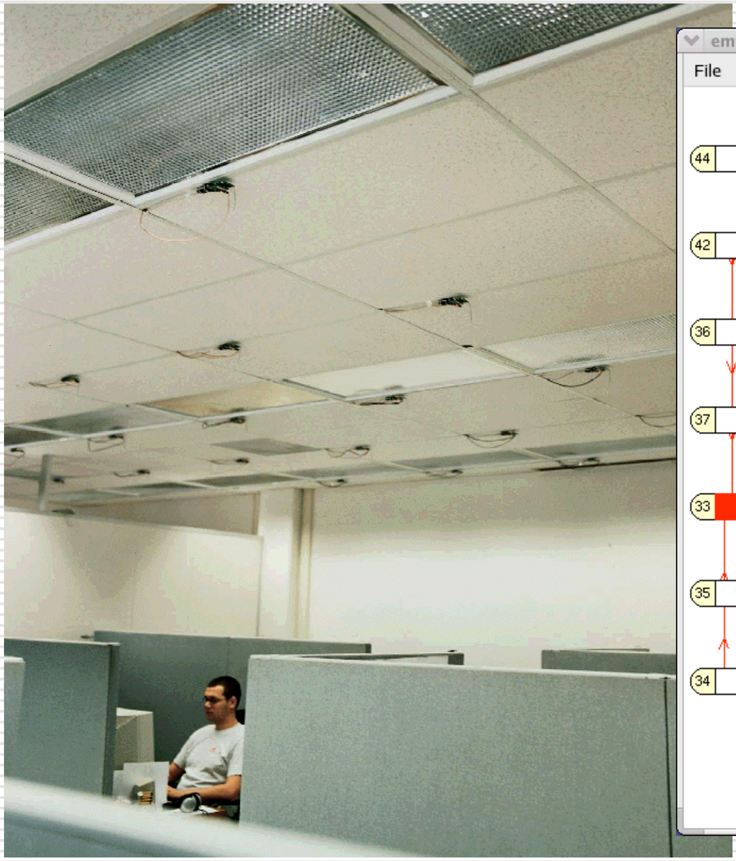


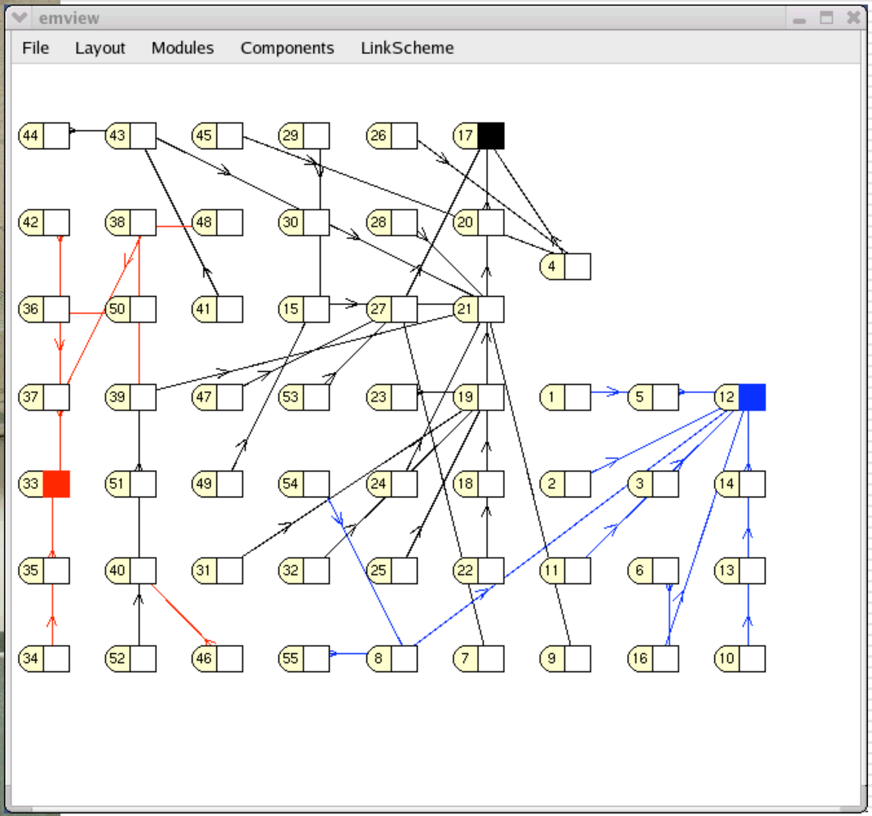
EmStar Components

- Tools
  - EmRun
  - EmProxy/EmView
  - SCALE
- Services
  - NeighborDiscovery/LinkSta
  - TimeSync/AudioServer
  - Routing
- Standard IPC
  - FUSD
  - Device Patterns



Em* Tools





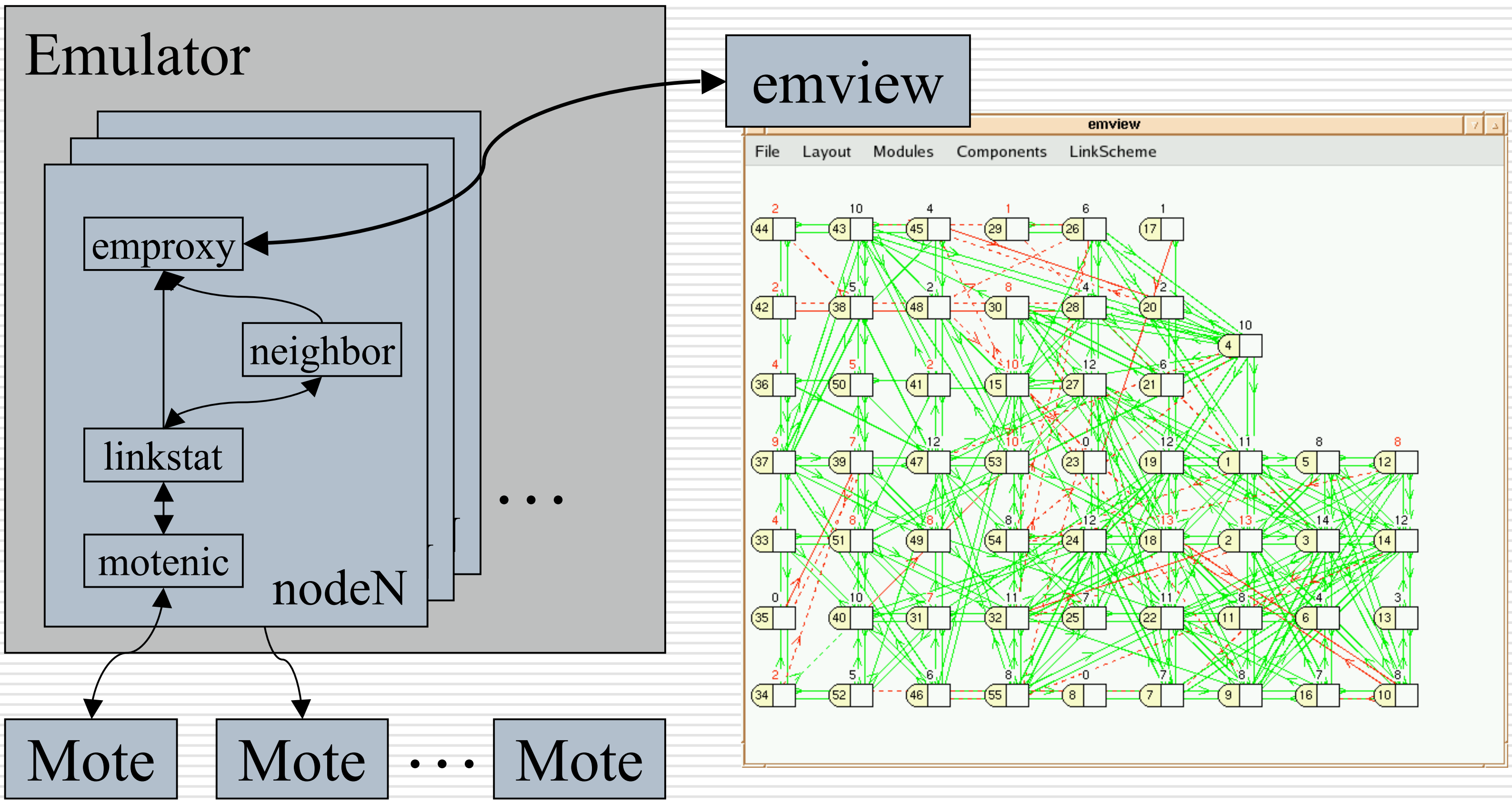
EmSim/EmCee

- Em* 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

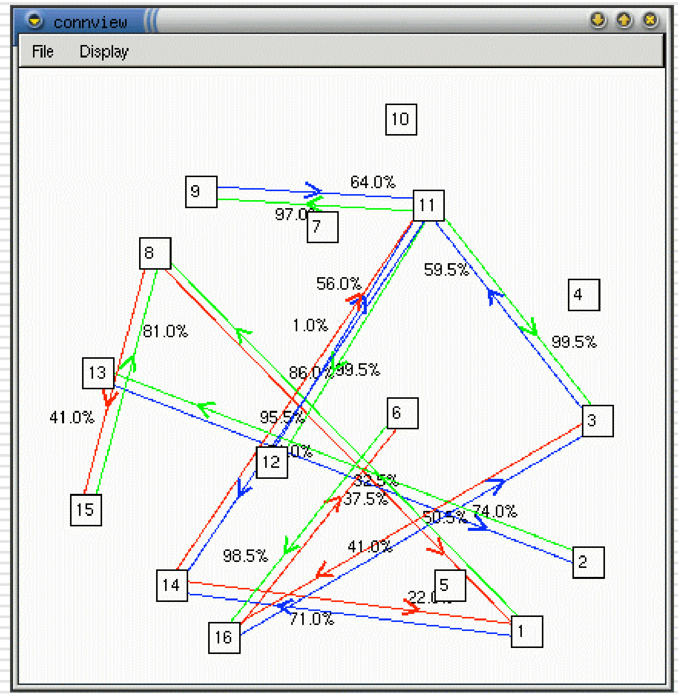
- Designed to start, stop, and monitor services
  - Increases robustness, resilience, autonomy
- EmRun config file specifies service dependencies
- Starting and stopping the system
  - Starts up services in correct order
  - Respawns services that die
  - Can detect and restart unresponsive services
  - 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



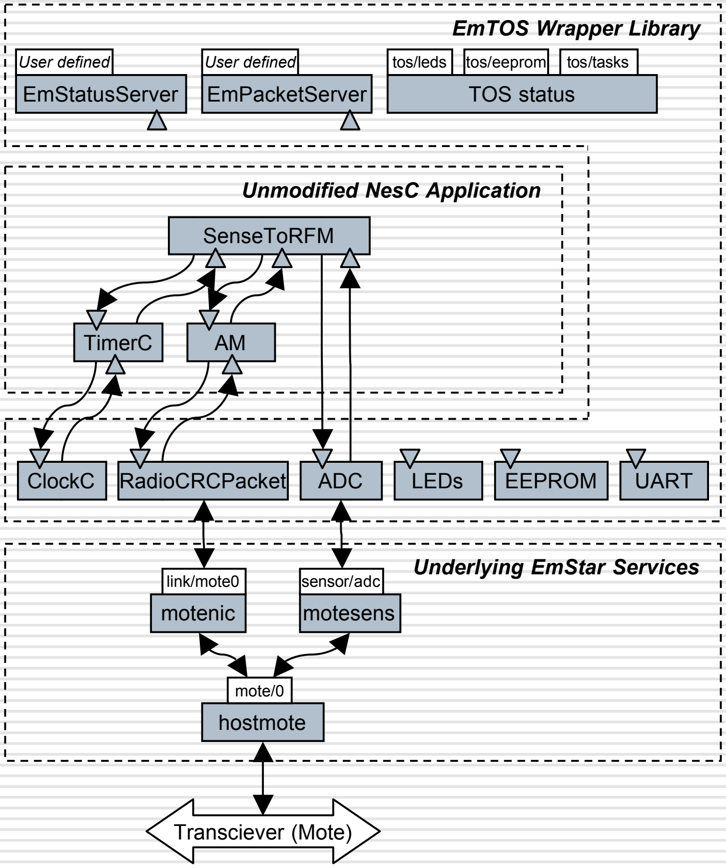
SCALE: Deployment Assessment

- SCALE is a tool for assessing connectivity across a deployed network
- Estimates link quality by repeated experiments
- Integrates to EmView visualizer
- Enables deployment to be tuned in the field



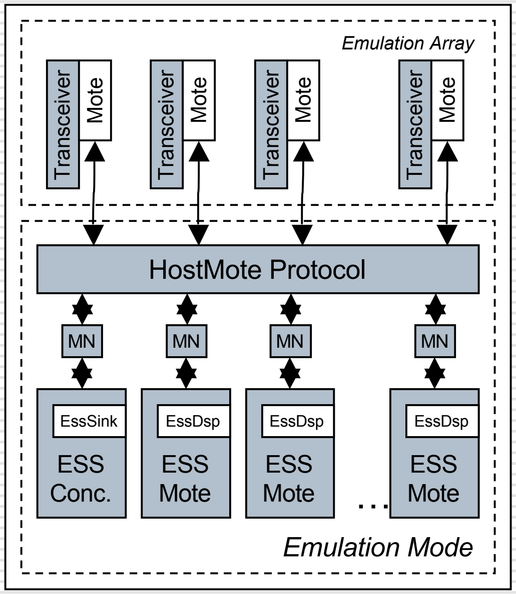
EmTOS: Support for Heterogeneous Systems

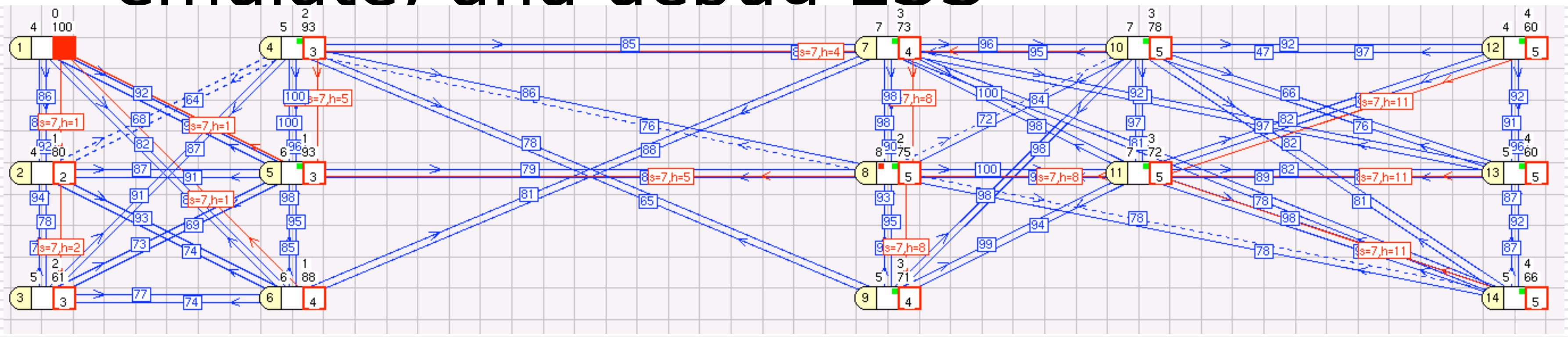
- Compile NesC Application
  - Platform "emstar"
  - Builds single EmStar module
- Wrapper Library
  - Provides TinyOS services
  - Enables NesC to provide new EmStar services
- Useful for deployment (ESS)
- Useful for simulation
  - Heterogeneous systems



Developing an Heterogeneous System

- Extensible Sensing System
  - Mote sources, Microserver sink
  - How to simulate in lab?
- Used EmTOS to simulate, emulate, and debug ESS

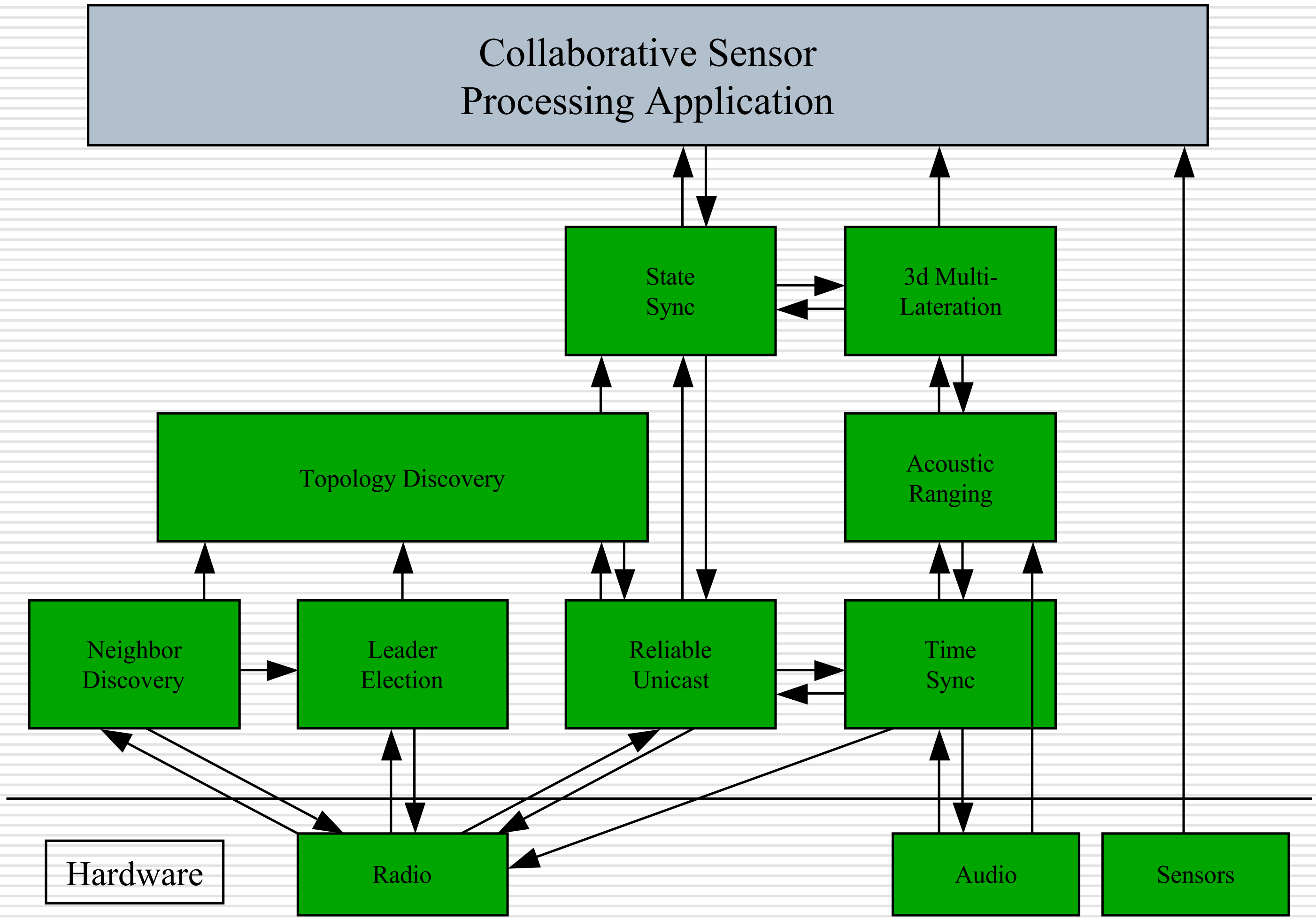




A Range of Simulation Modes

- ***Pure Simulation Mode:*** Microserver and Mote code is run centrally in the EmStar environment, and all nodes communicate through a simulated RF channel.
- ***Emulation mode:*** Microserver and Mote code is run centrally in the EmStar environment, but nodes communicate using a real RF channel
- ***Real Mode:*** Microserver code runs centrally, while Mote code runs natively on real Motes, with a serial backchannel for debugging. Emulated Microservers communicate with real Motes and other Microservers through the real RF channel.
- ***Hybrid Mode:*** A mixture of Real and Emulated modes, where some Motes are emulated and some run natively. All nodes communicate through the real RF channel.

Em* Services



Collaborative Sensor Processing Application
State Sync
3d Multi-Lateration
Topology Discovery
Acoustic Ranging
Neighbor Discovery
Leader Election
Reliable Unicast
Time Sync
Hardware
Radio
Audio
Sensors

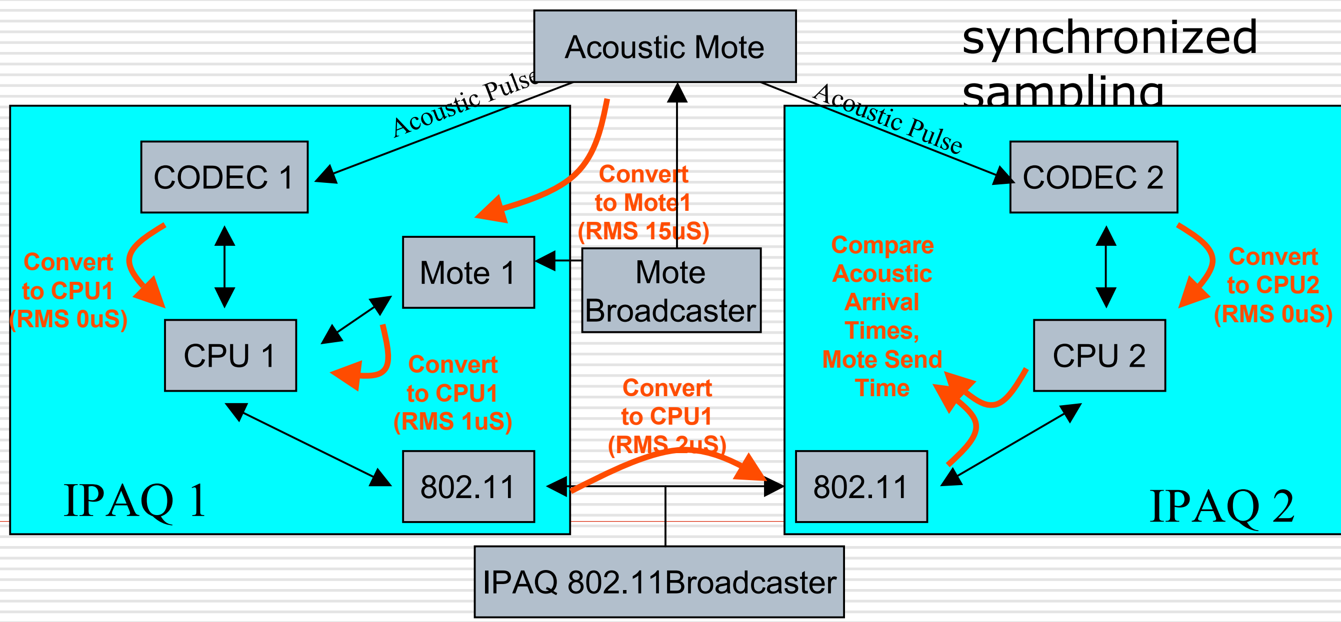
Neighbor Discovery / LinkStats

- 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

TimeSync and Audio Server

- Time sync estimates conversions
  - To remote nodes' CPU clocks
  - Among local clocks
    - Audio codec clock
    - Radio processor clock

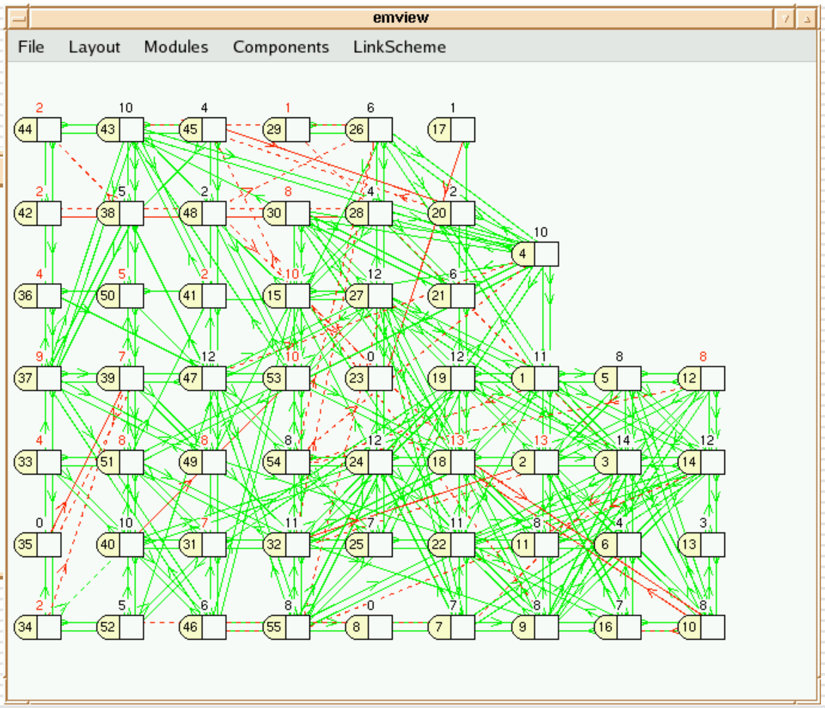
- Audio Server buffers audio data
  - Post-facto triggering
  - High accuracy time synch
    - Averaging many time stamps
    - Enables continuous synchronized sampling



Em* Service Lifecycle

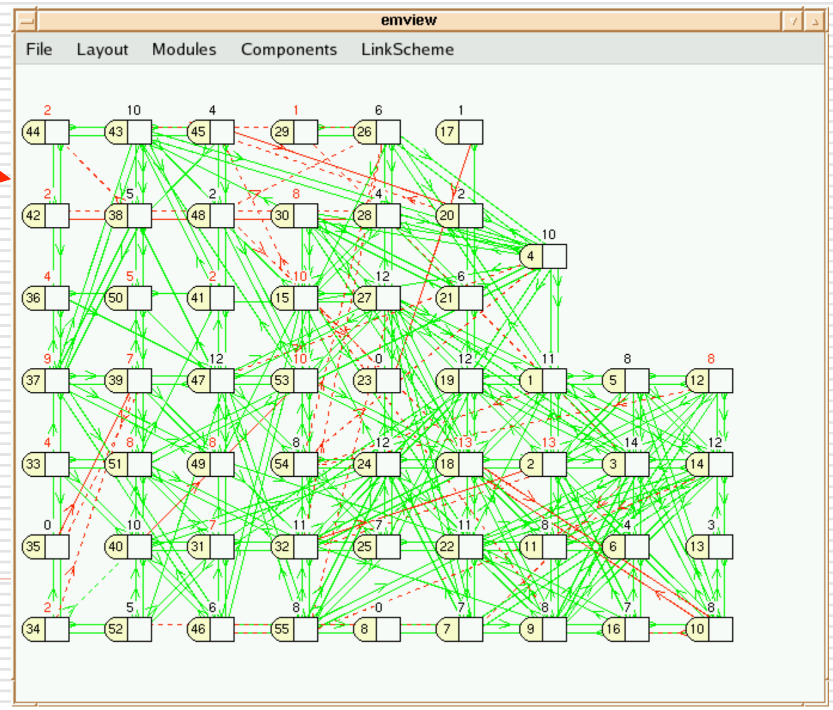
- Interface design:
  - Encapsulate some useful mechanism
  - Expose the application-specific policy decisions
- Choosing modularity:
  - Don’t bite off too much at once
  - Something that at first looks simple can grow more complex
  - Don’t worry about efficiency of more modules.. Optimize later
  - **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

Em* IPC Standards



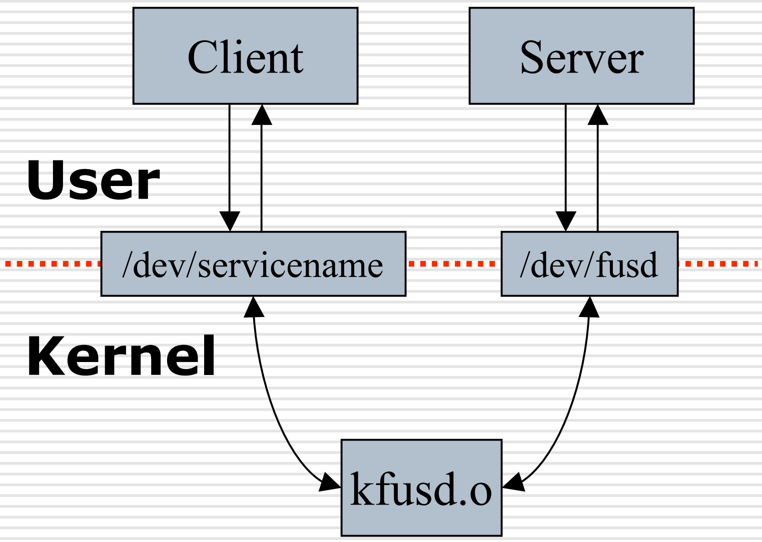
Interacting With em*

- 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



FUSD IPC

- Inter-module IPC: FUSD
  - Creates device file interfaces
  - Text/Binary on same file
  - Standard interface
    - Language independent
    - No client library required
  - Requires Linux "devfs"
    - (Until kernel 2.6)

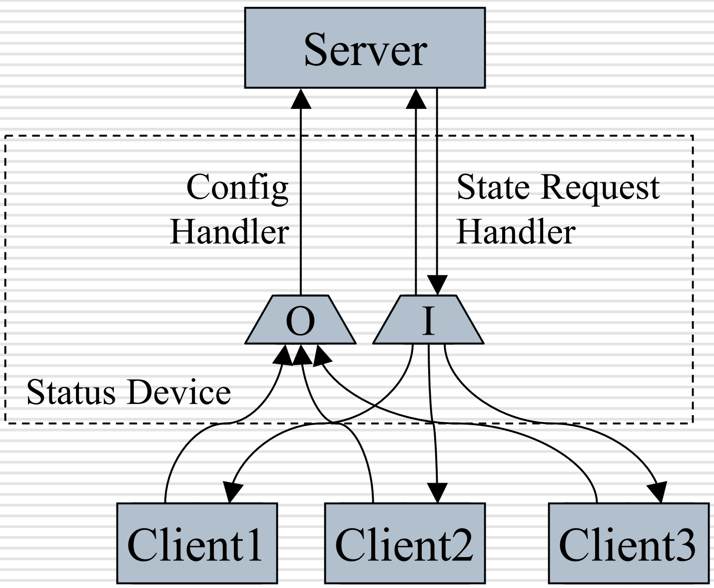


Device Patterns

- FUSD can support virtually any semantics
  - What happens when client calls read()? etc..
- But many interfaces fall into certain patterns
- Device Patterns
  - Encapsulate specific semantics
  - Take the form of a library:
    - Objects, with method calls and callback functions
    - #1 priority: ease of use
  - Integrates with the GLib event loop

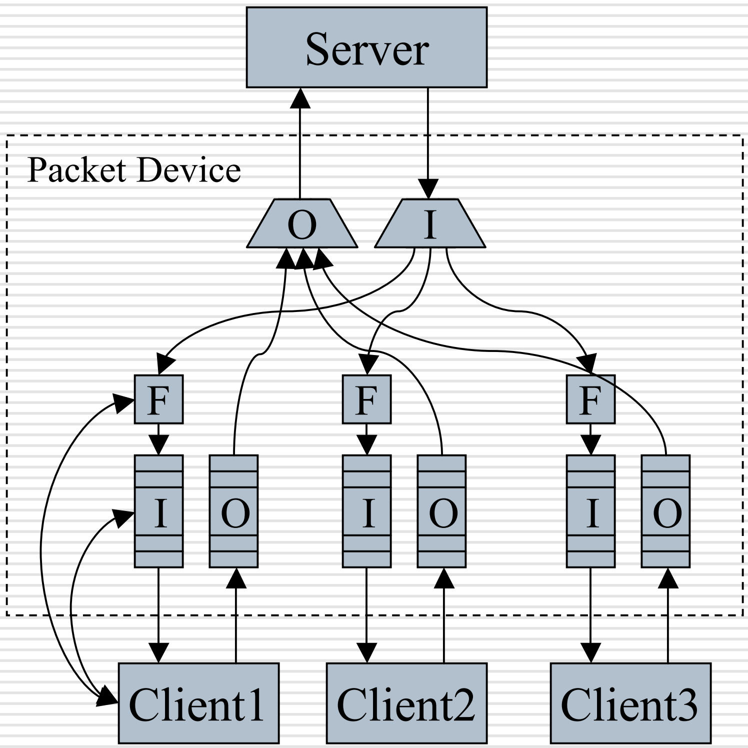
Status Device

- 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)



Programming tips

- Write robust code
  - Always check for sequence number (to avoid duplicate transmissions/infinite loops)
  - Always check for the 'status' flag. If queues on another component is full, it may return FAIL. If you do not deal with it, component state machine can be stuck.
- Think about randomization
  - Important to avoid network collisions.
  - Should be neither too short nor too long.

The End