Minitask Architecture and MagTracking Demo

NEST Retreat Jan 2003

Presented by
Cory Sharp
Kamin Whitehouse
Rob Szewczyk

Minitask Group Assignment

- Estimation
  - Ohio State (spanning tree)
  - UVA
- Localization
  - Lockheed ATC
  - Berkeley
- Power Management
  - CMU
- Routing
  - Ohio State
  - UVA
  - PARC
  - UMich
- Service Coordination
  - UC Irvine
  - Lockheed ATC
  - Berkeley (oversight)
- Team Formation
  - Ohio State
  - Lockheed ATC
  - UVA
- TimeSync
  - Ohio State (fault tolerance)
  - UCLA
  - Notre Dame

Minitask Goals

- “Composable middleware”
- Services
  - Metric: usefulness (refactorization)
- Components
  - Metric: composability, modularity
- Assist collaboration between groups in the short term
  - Provide an initial, working testbed
  - Groups will enhance and replace services
- Toward code generation

Composability Gives:

- Maintainability
- Collaboration
- Extensibility
- Consistency
- Predictability of interface
- Verifiability

Architecture Overview

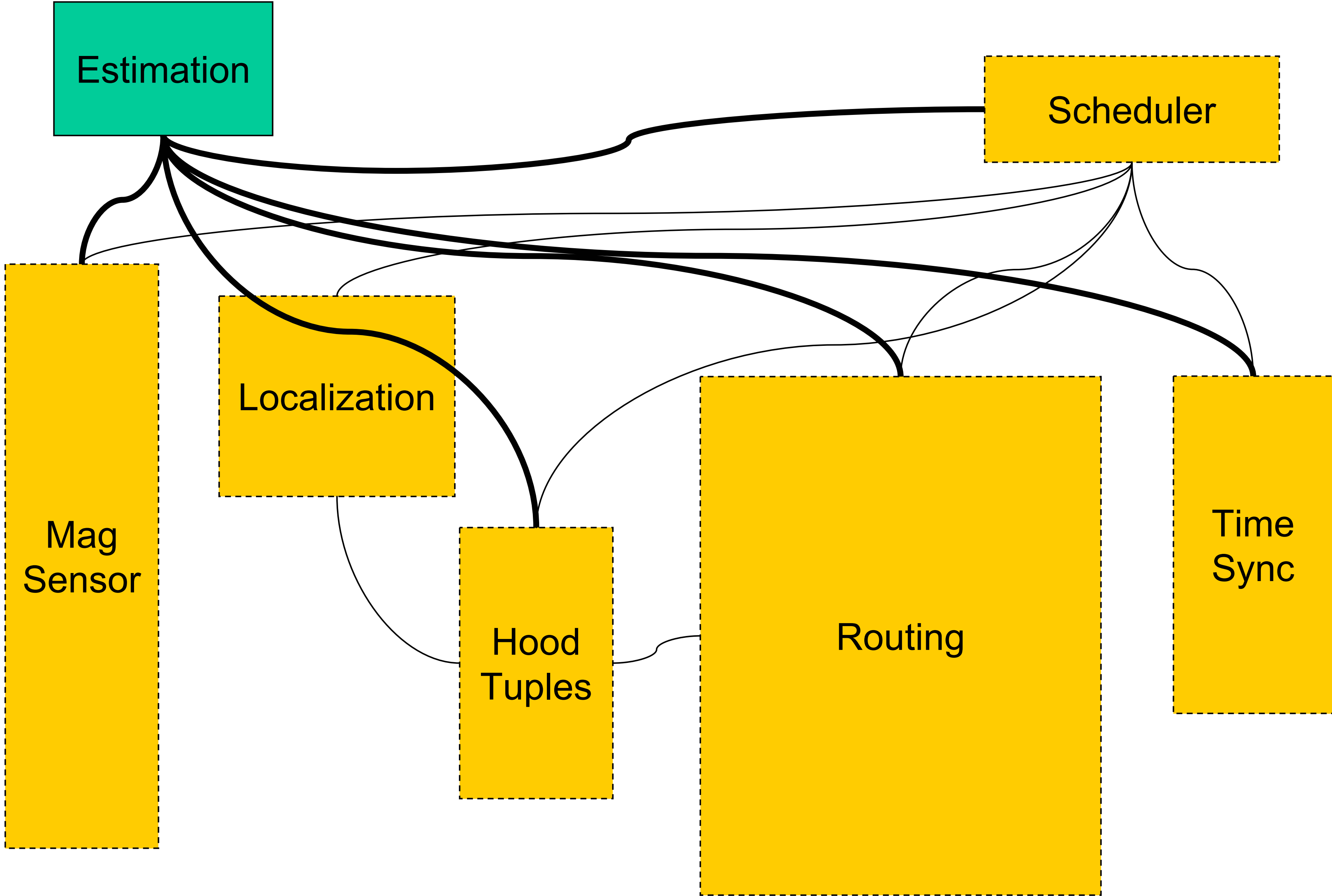
MagTracking Demo Logic

- Each mote knows only its own location.
- Neighborhood discovery
  - Learn of neighbors and their location
- Magnetometer readings are acquired, filtered, and placed on the neighborhood.
- If the local mote has the highest reading, it calculates and sends an estimate…
- ... via geographic location-based routing to (0,0).
  - Neighborhood membership restricted to force reliable multi-hop.
- The mote at (0,0) sends the estimate to the camera.

MagTracking Services

- MagSensor
  - Accumulates, filters magnetometer readings.
- Routing
  - Supports a number of routing methodologies.
- Neighborhood
  - Facilitates local discovery and data sharing
- Localization
  - Discovers geographic location of the mote
- TimeSync
  - Synchronizes time between motes
- Service Coordination
  - Controls behavior of an aggregation of services

Service Wiring



MagTracking Services

- MagSensor
  - Accumulates, filters magnetometer readings
- Routing
  - Supports a number of routing methodologies
- Neighborhood
  - Facilitates local discovery and data sharing
- Localization
  - Discovers geographic location of the mote
- TimeSync
  - Synchronizes time between motes
- Service Coordination
  - Controls behavior of an aggregation of services

Magnetometer

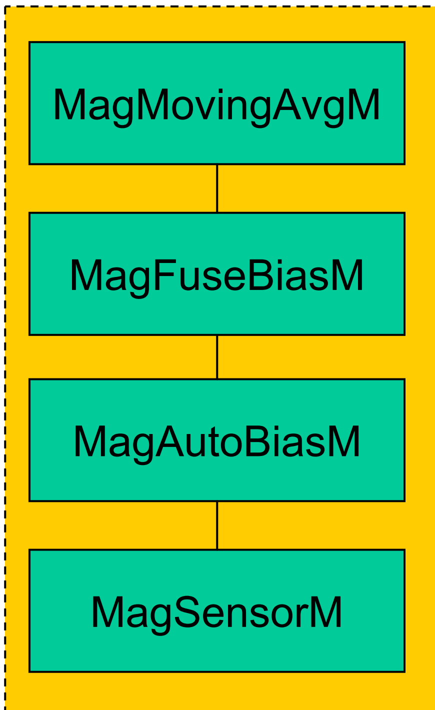
Magnetometer Philosophy

- Seeking composability
- Break functionality into Sensor, Actuator, and Control
  - Sensors: get() and getDone(value)
  - Actuators: set(value) and setDone()
  - Control: domain-specific manipulations that gain nothing from generalization
- Separate module behavior from composite behavior
  - Filter modules live in a vacuum
- Maximize opportunity for composability

Magnetometer Interfaces

- MagSensor interface
  - Abstract sensor interface
    - `command result_t read();`
    - `event result_t readDone( Mag_t* mag );`
- MagBiasActuator interface
  - Abstract actuator interface
    - `command result_t set( MagBias_t* bias );`
    - `event result_t setDone( result_t success );`
- MagAxesSpecific interface
  - “Control” functionality that doesn’t fit the model of an actuator or sensor
    - `command void enableAxes( MagAxes_t* axes );`
    - `command MagAxes_t* isAxesEnabled();`

Magnetometer Modules



- MagMovingAvgM module
  - Filter from MagSensor to MagSensor
  - Performs a moving average across magnetometer readings
- MagFuseBiasM module
  - Filter from MagSensor to MagSensor
  - "Fuses" bias with reading to give one "absolute" reading value
- MagAutoBiasM module
  - Filter from MagSensor to MagSensor
  - Magnetometer readings vary between 200 and 800 but have an absolute range of about 9000.
  - Readings often rail at 200 or 800, continually adjust bias to drive readings to 500.
- MagSensorM module
  - Translation layer between TinyOS and MagSensor

Magnetometer Conclusions

- Filtering
  - Transparency in composition

Routing

Routing Overview

- Application-level features
  - Broadcast example
- Developer features
  - General routing structure
  - Anatomy of a SendByLocation
  - Routing configuration file
  - Routing extensions

Application-Level Routing Features

- Send and receive interfaces are nearly-identical to TinyOS counterparts.
  - Destination differs per semantic
    - Broadcast: max hops
    - Location: position and radius in R3
    - Etc.
- Unification of routing methods.
  - Receive is independent of the routing module and method.
  - Interact with the routing stack as a single, multi-method component.
- Message body packing is independent of the routing method.

Application Interface – Broadcast Example

- Choose a **protocol number**, say 99, and wire to it:

```
AppC -> RoutingC.SendByBroadcast[99];
AppC -> RoutingC.Receive[99];
```

- Initialize and send your message:

```
mydata_t* msgbody =
  (mydata_t*)initRoutingMsg( &msg, sizeof(mydata_t) );
// ...
call SendByBroadcast.send( 2 , &msg );
```

- Send done event:

```
SendByBroadcast.sendDone(
  TOS_MsgPtr msg, result_t success );
```

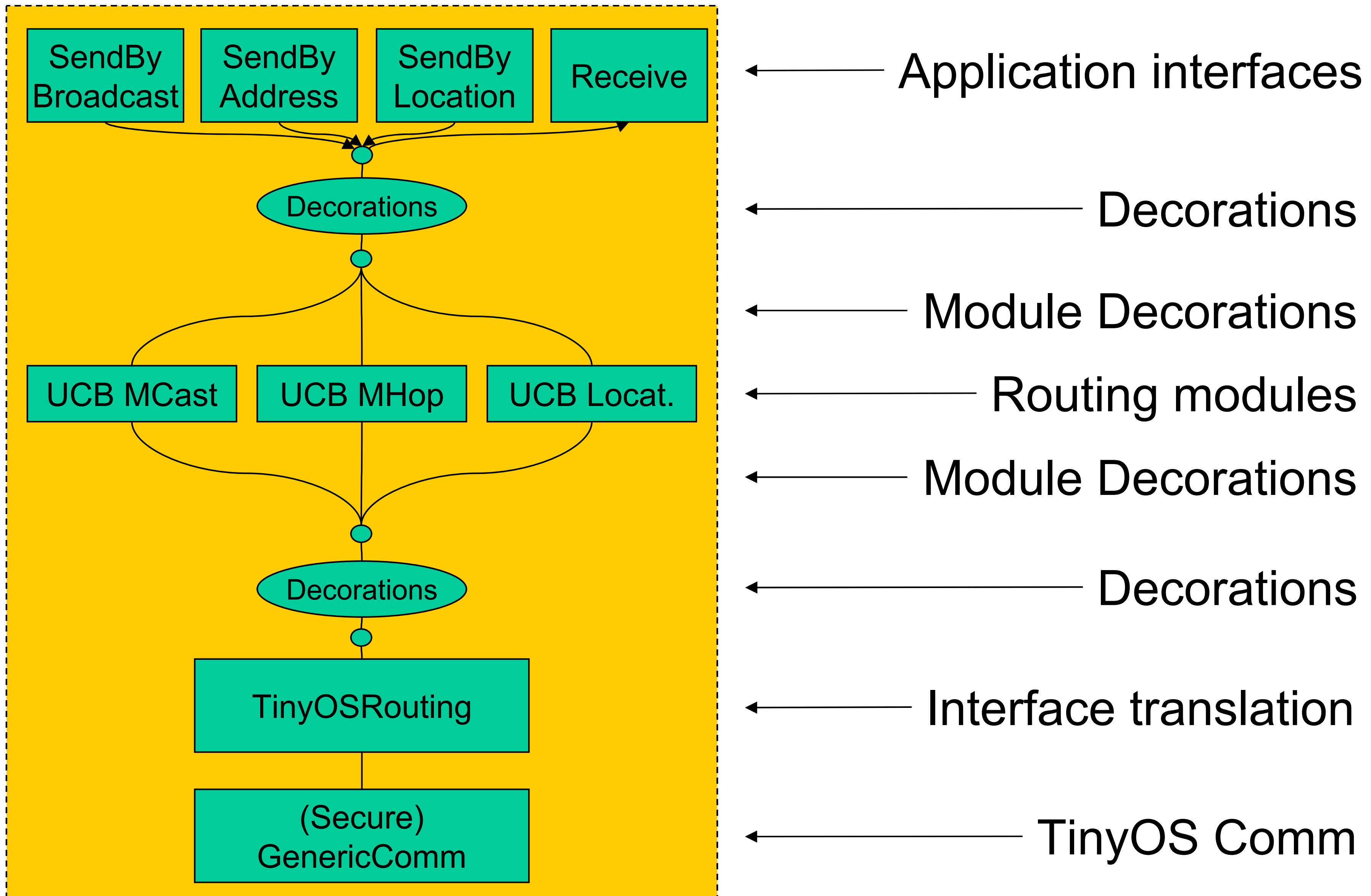
- Receive time sync messages:

```
TOS_MsgPtr RoutingReceive.receive(TOS_MsgPtr msg);
```

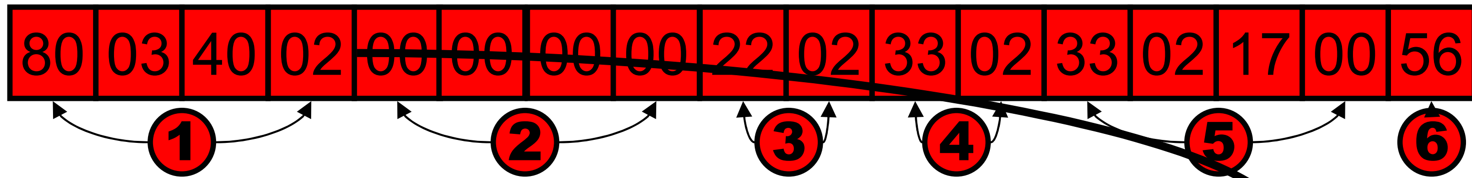
Developer Routing Features

- Internal routing components are modular and composable.
  - Routing **modules** *only* responsible for decisions of destination.
  - Routing **decorations** augment the stack independent of routing modules.
  - Modules and decorations always provide and use RoutingSend and RoutingReceive.
- Routing **extensions** enabled by per-message key-value pairs.

General Routing Structure

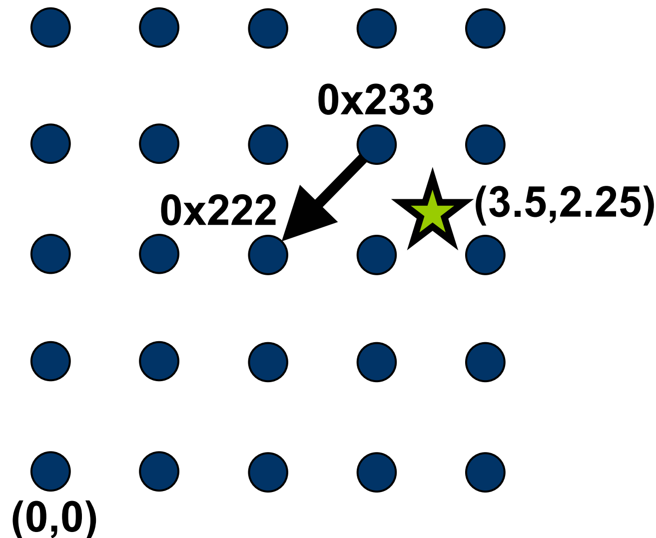


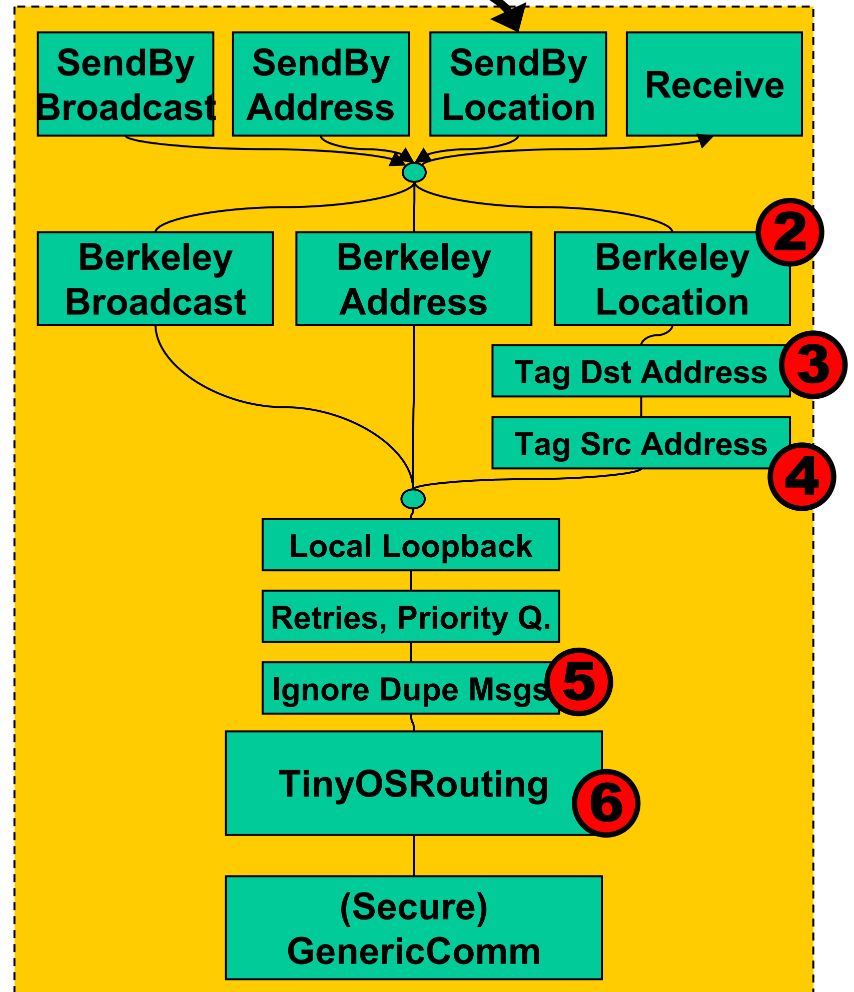
Anatomy of a SendByLocation



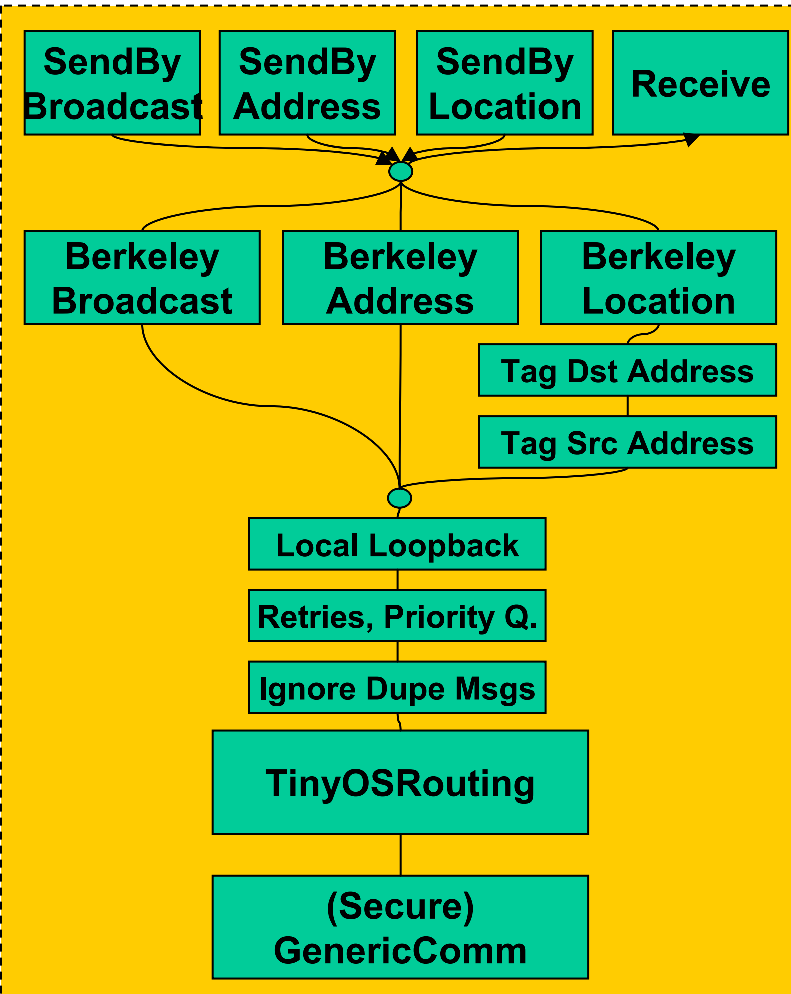
1

Mote 0x233 sends an estimate (3.5,2.25) to location (0,0) using “protocol” 86.





Routing Configuration File




```
/*<routing>
Top:
TOSAM 100:
  provides interface RoutingSendByAddress;
  BerkeleyAddressRoutingM;
TOSAM 101:
  provides interface RoutingSendByBroadcast;
  BerkeleyBroadcastRoutingM;
TOSAM 102:
  provides interface RoutingSendByLocation;
  BerkeleyLocationRouting2M;
  TagDestinationAddressRoutingM;
  TagSourceAddressRoutingM;
Bottom:
  LocalLoopbackRoutingM;
  ReliablePriorityRoutingSendM;
  IgnoreDuplicateRoutingM;
  IgnoreNonlocalRoutingM;
</routing>*/
includes Routing;
configuration RoutingC {
}
implementation {
  // ...
}
```

Routing Extensions

- Composability is at odds with customization
- Extensions use key-value pairs per message
- Modules/decorations that provide an extension have some extra markup:
  - `//!! RoutingMsgExt { uint8_t priority = 0; }`
  - `//!! RoutingMsgExt { uint8_t retries = 2; }`
- Preprocessed into a substructure of TOS_Msg.
- Applications assume extensions exist:
  - `mymsg.ext.priority = 1;`
  - `mymsg.ext.retries = 4;`
- Unsupported extensions produce compile-time errors.

Routing Conclusions

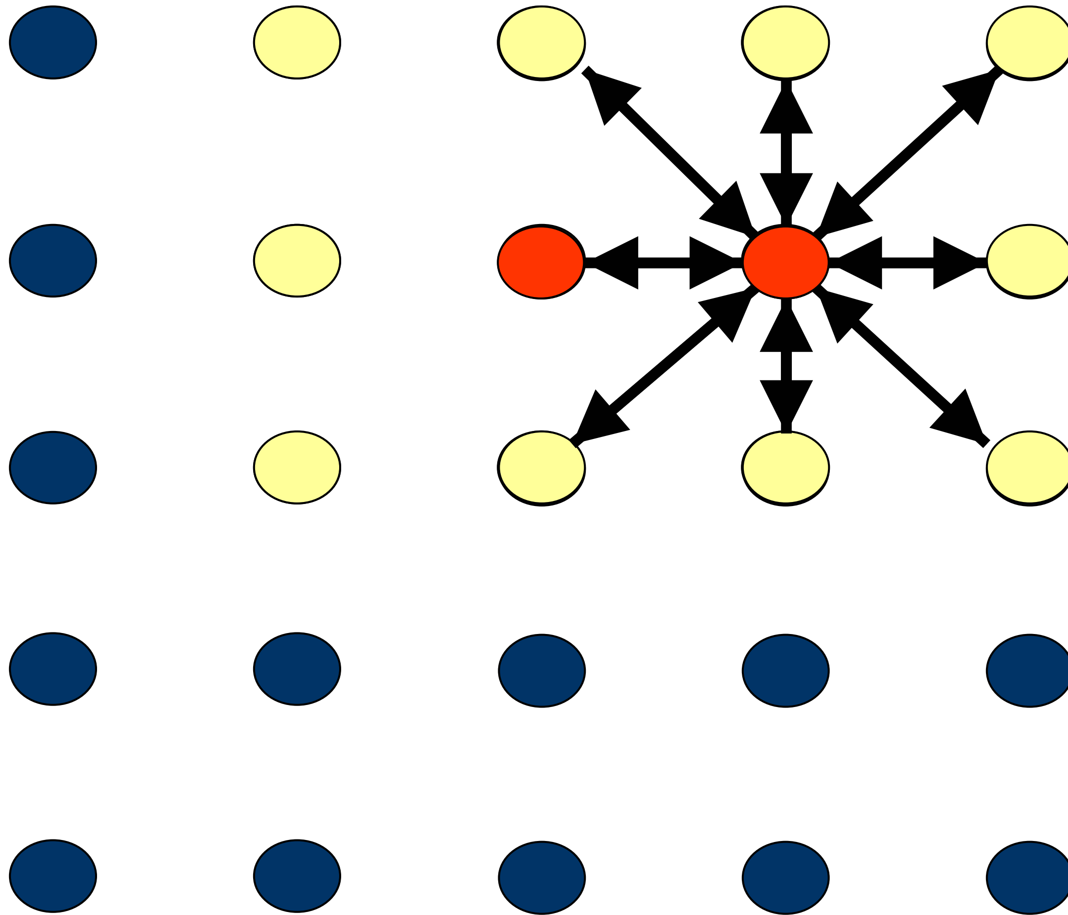
- Application level is nearly identical to TOS
- Unification of routing methods
- Internally modular and composable
- Extensible

Neighborhood

Outline

- The Neighborhood Service
- Our Implementation
- Alternatives

Data Sharing



Data Sharing

Data Sharing Today

- radio protocol
- Implement comm functionality
- Choose neighbors
- Data management

Neighborhood Service

- "Get" interface to remote data
- Use Neighborhood Infce

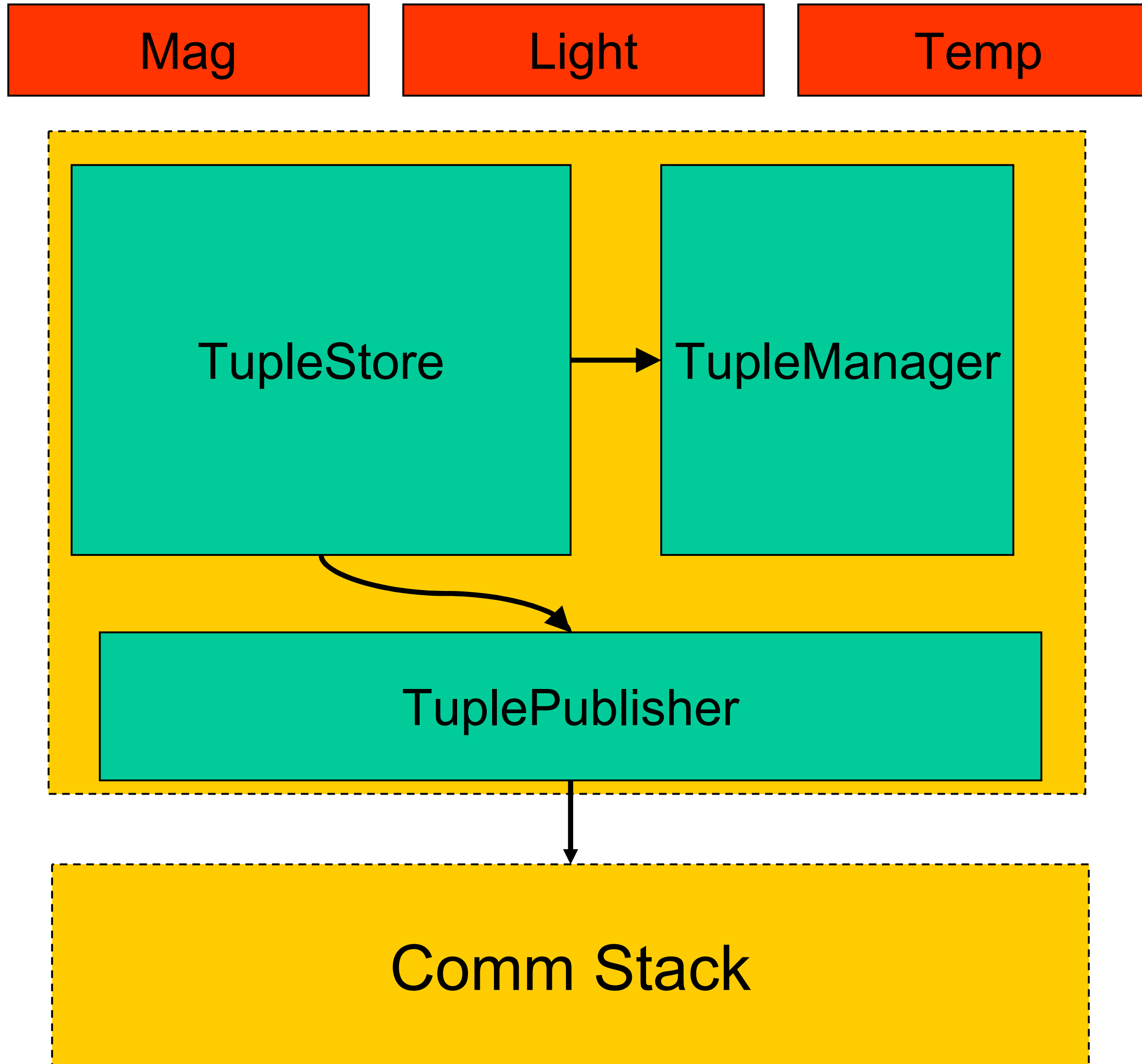
Standardized API

- Declare data to be shared
- Get/set data
- Choose neighbors
- Choose sync method

Benefits

- Refactorization
- Simplifies interface to data exchange
- Clear sharing semantic
- Sharing of data between components
- Optimization

Our Implementation



Our API

- Declare data to be shared
- Get/set data
- Choose neighbors
- Choose sync method

Our API

- Declare data to be shared

```
//!! Neighbor {mag = 0;}
```

- Get/set data
- Choose neighbors
- Choose sync method

Our API

- Declare data to be shared
- Get/set data
- Choose neighbors
- Choose sync method

Our API

- Declare data to be shared
- Get/set data
  - Neighborhood “interface”
- Choose neighbors
- Choose sync method

Neighborhood Interface

- `command struct* get(nodeID)`
- `command struct* getFirst()`
  `command struct* getNext(struct*)`
- `event updated(nodeID)`
- `command set(nodeID, struct*)`
- `ommand requestRemoteTuples();`

Our API

- Declare data to be shared
- Get/set data
- Choose neighbors
- Choose sync method

Our API

- Declare data to be shared
- Get/set data
- Choose neighbors
  - Choose tupleManager component
- Choose sync method

Our API

- Declare data to be shared
- Get/set data
- Choose neighbors
- Choose sync method

Our API

- Declare data to be shared
- Get/set data
- Choose neighbors
- Choose sync method
  - Choose tuplePublisher component

Limitations

- Each component might want to have different
  - neighbors
  - sharing semantics
- Might want to share data with non-local nodes
- Space efficiency

Alternatives

- multiple instantiations
- multi-hop hoods
- groups
- distributed shared memory
- SQL interface
- lazy/eager

Conclusion

- Provides simpler interface to remote data
- Simplifies application logic

- Evaluate usefulness:
- Used by
  - magTracking
  - Localization
  - Service Coordination
  - Routing
  - Etc.

Service Coordinator Design

Outline

- Motivation
- Use Scenarios
- High-level Overview
- Interfaces

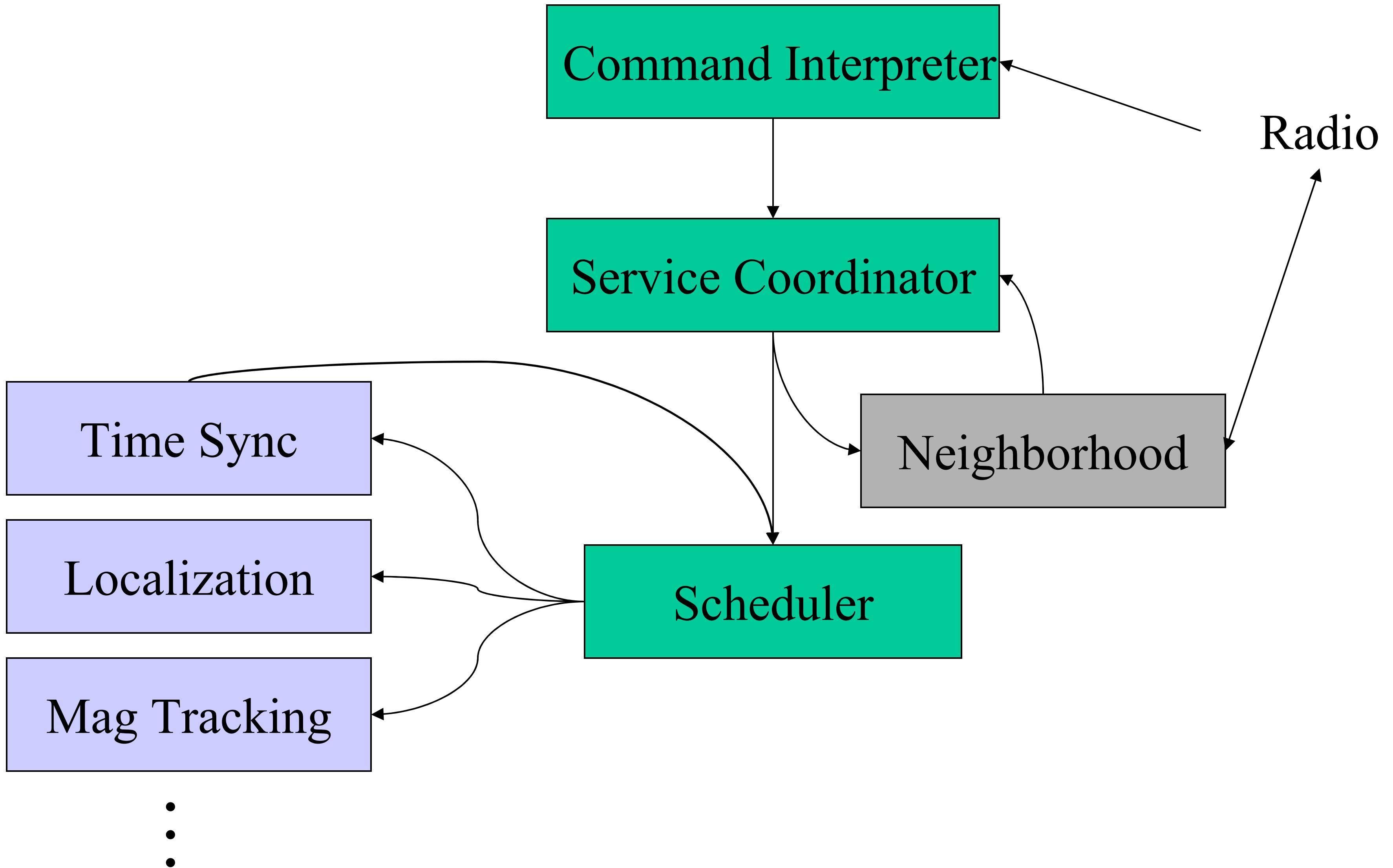
Motivation

- Services need not to run all the time
  - Resource management – power, available bandwidth or buffer space
  - Service conditioning – minimize the interference between services
  - Run time of different services needs to be coordinated across the network
- Generic coordination of services
  - Separation of service functionality and scheduling (application-driven requirement)
  - Scheduling information accessible through a single consistent interface across many applications and services
  - Only a few coordination possibilities
- Types of coordination
  - Synchronized
  - Colored
  - Independent

Use Scenarios

- Midterm demo
  - Localization, time sync
  - Scheduling of active nodes based to preserve sensor coverage
- Other apps
  - Control the duty cycle based on relevance of data
  - Sensing perimeter maintenance
  - Generic Sensor Kit

High-level Overview



Interfaces

- Service Control

```
interface ServiceCtl {
        command result_t ServiceInit();
        command result_t ServiceStart();
        command result_t ServiceStop();
        event  void  ServiceInitialized(result_t status);
}
```

- Service Scheduler Interface

```
typedef struct {
        StartTime;
        Duration;
        Repeat;
        CoordinationType;
} ServiceScheduleType;
interface ServiceSchedule {
        command result_t ScheduleSvc(SvcID, ServiceScheduleType *);
        command ServiceScheduleType *getSchedule(SvcID);
        event ScheduleChanged(SvcID);
}
```

End

Secret Bonus Slide

Routing - Key Definitions

- AM number
  - TinyOS Active Messaging
- "Protocol" number
  - Defined by and dispatched into the application
  - Irrelevant to the routing stack
- "Method" number
  - Determines which **routing module** handles a message
- Routing modules
  - Expressly limited responsibility
    - Translate from a semantic destination (hops, location, etc) to network address
    - Choose between forwarding or local delivery of incoming messages
  - Examples: BerkeleyBroadcastRouting, PARCBroadcastRouting, BerkeleyLocationRouting, etc
- Decorations
  - Augment ("decorate") a routing stack with behaviors beyond the scope of routing modules
  - Examples: duplicate message rejection, outgoing message queuing, debugging headers, etc