



Embedded OSes



Carolyn Keenan, Ian Perera, Yu Zhong

Challenges for Embedded OSeS

- ▶ Limited Resources
 - ▶ Memory
 - ▶ Computation Speed
 - ▶ Power
- ▶ Real-time, interactive processes
- ▶ Network communication



Common approaches

▶ Modularity

- ▶ Reduces size, power consumption by tailoring to the specific environment
- ▶ More flexible configurations
- ▶ Allows for computation distribution over multiple processors
- ▶ Requires coordination between modules

▶ Real-time processing

- ▶ Smartphone OS: Notification center
- ▶ RobotOS: Broadcast/Subscriber
- ▶ TinyOS: Event-driven



Specific Environments

- ▶ **Smartphone OS**

- ▶ Network communication
- ▶ Low-power consumption

- ▶ **Robot OS**

- ▶ Interoperability of multiple device drivers
- ▶ Communication between robot and offboard system

- ▶ **TinyOS**

- ▶ Low-power, restricted memory
- ▶ Real-time event handling



Smart Phone Operating Systems

Presenter: Yu Zhong

Instructor: Sandhya Dwarkadas

Outline

- ▶ **Main Focus: Android**
 - ▶ Compared with iOS
- ▶ **Aspects to analyze:**
 - ▶ Hardware specification, architecture
 - ▶ Process, Memory, Power Management
 - ▶ Network support



Introduction

- ▶ **Challenge of Embedded system:**
 - ▶ Limited resource
 - ▶ Smaller scale
 - ▶ Energy sensitive
 - ▶ Network is important



Android Hardware Specification

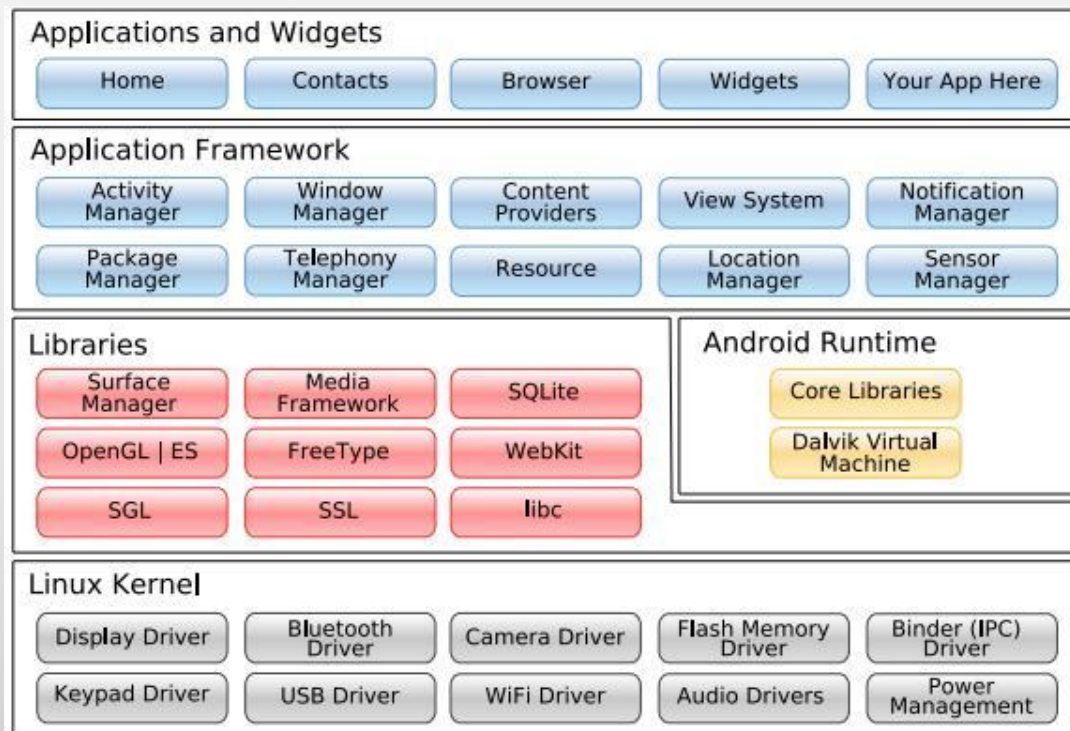
- ▶ **Minimum Requirement:**
 - ▶ 32MB RAM
 - ▶ 32MB Flash Memory
 - ▶ 200MHz Online Processor
- ▶ **Supported Processors:**
 - ▶ Mainly ARM v6 and v7, also other like x86 [1]



Android

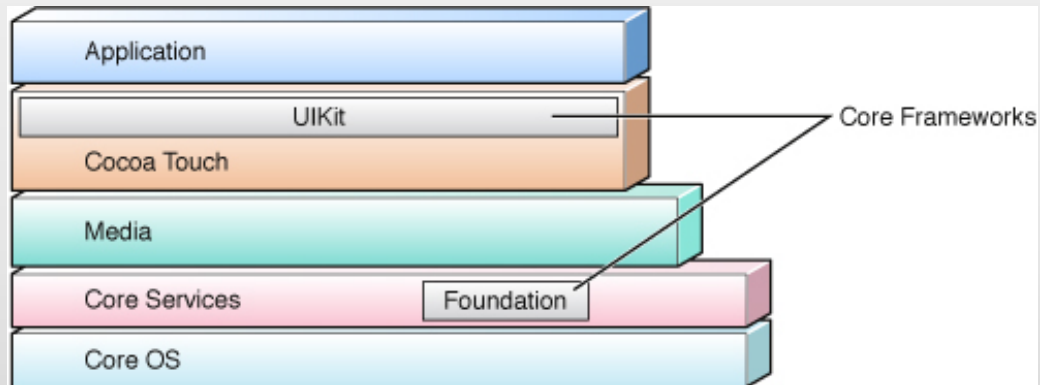
▶ Architecture

- ▶ 3 Layers: Linux Kernel, Middleware Libraries and APIs written in C, Java-compatible Virtual Machine(Dalvik)[2]



iOS

- ▶ UNIX Based, similar to Mac OS:
 - ▶ Sandbox
 - ▶ Process
 - ▶ Thread
- ▶ Runs on ARMv6 and ARMv7



iOS

- ▶ **Sandbox**
 - ▶ User mode apps runs in sandbox
 - ▶ No child process
 - ▶ No external file access
 - ▶ No interprocess communication
- ▶ **Sandbox handles main() and create main thread of applications**



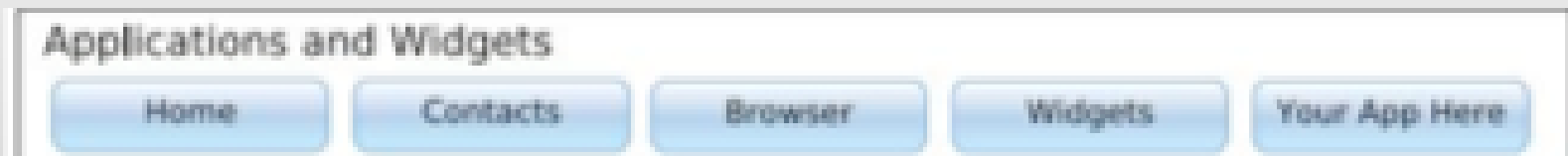
Modularity

- ▶ **Benefits:**
 - ▶ Size reduction
 - ▶ Flexible Development
 - ▶ Reliability
- ▶ **Different from desktop Linux**
 - ▶ Optimized Graphics
 - ▶ Removed heavy computational module, i.e. scripts
 - ▶ Modified file system and power management related kernel modules
 - ▶ Added remote debugging bridge into kernel



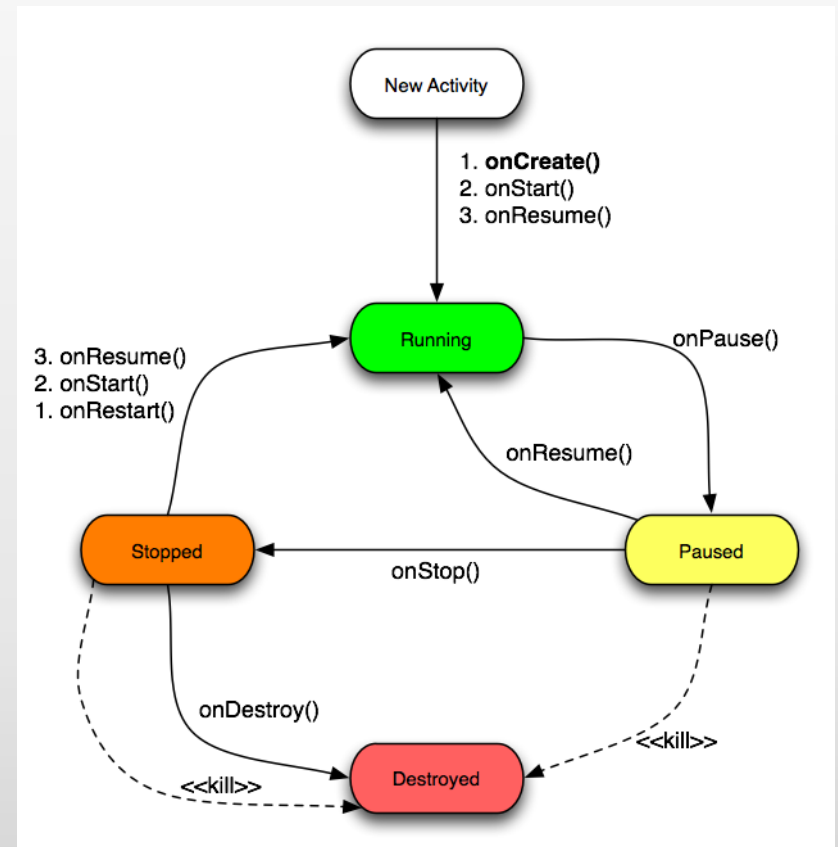
Android

- ▶ Application components:
 - ▶ Activity (GUI)
 - ▶ Service (bg or remote process, optional)
 - ▶ Broadcaster receiver (respond to system)
 - ▶ Content provider (e.g. Users' contacts)
- ▶ Main activity as entry, no main()



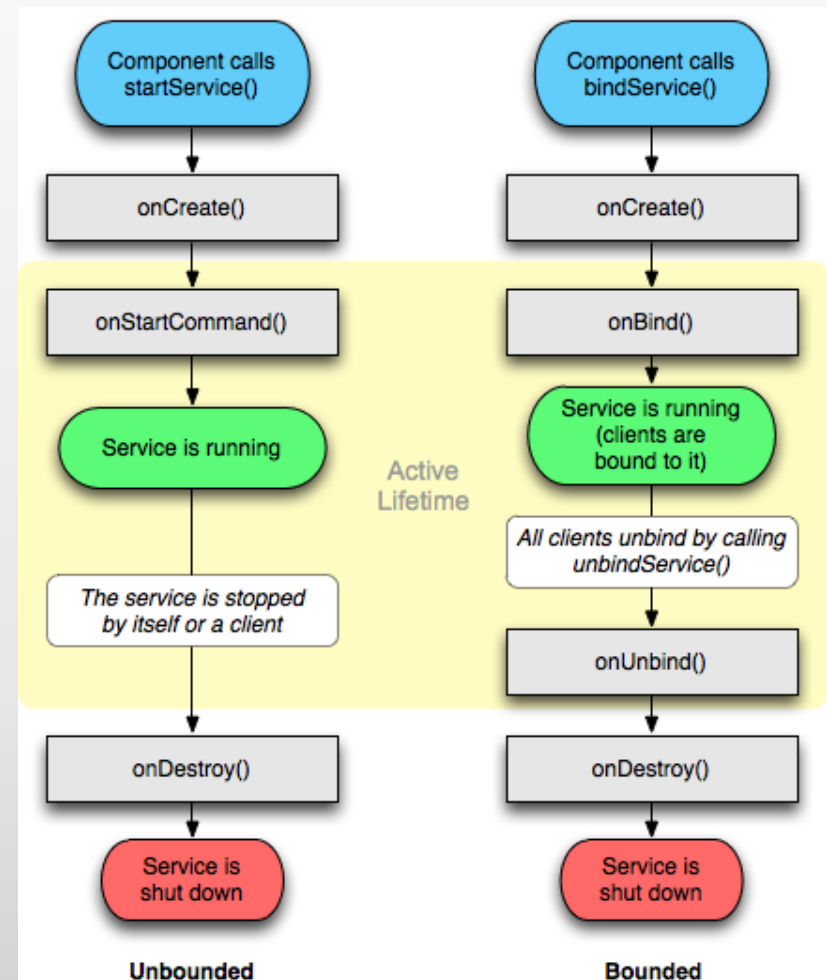
Android

- ▶ Activity Life circle:
 - ▶ Foreground
 - ▶ Visible
- ▶ State is saved when pushed into BG
 - ▶ Similar to swap



Android

- ▶ Service Life circle
 - ▶ Independent or bind to clients
- ▶ Will be killed when memory is insufficient
- ▶ Mainly used as background process



Android

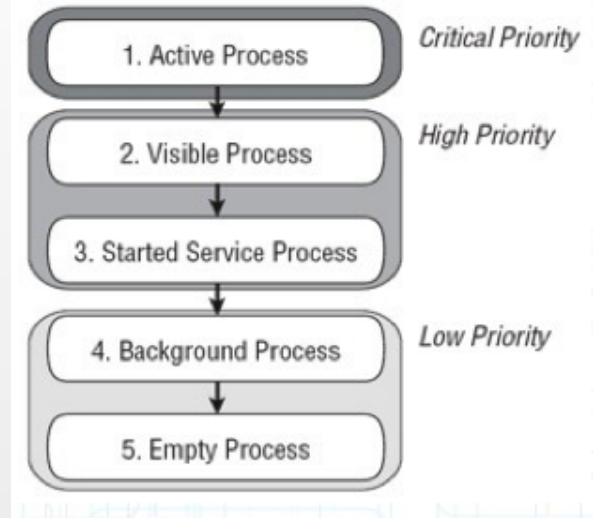
- ▶ **Broadcaster receiver**
 - ▶ Interprocess/activity communication
 - ▶ Killed after short running time
- ▶ **Content provider**
 - ▶ Provide data access to other applications



Android

▶ Process

- ▶ Priority based scheduling
- ▶ Each process holds all components of an application
- ▶ Transparent to application developer (similar to iOS, except bg running)



▶ Thread

- ▶ Main thread where most activities occurs, handles GUI rendering
- ▶ Heavy or synchronizing tasks should run in BG
- ▶ Create new thread with Java Thread objects
- ▶ iOS wraps pthread with ObjC, similar to Java Thread



Android Power Management

- ▶ **High level: Application framework Power Manager**
 - ▶ Provide access interface
 - ▶ Allows setting of “stay on”, iOS non-access
- ▶ **Low level: Linux driver**
 - ▶ Two system calls:
 - ▶ `android_register_early_suspend(android_early_suspend_t *handler)`
 - ▶ `android_register_early_resume(android_early_resume_t *handler)`



Summary

- ▶ **Common things:**
 - ▶ More or less have a sandbox
 - ▶ Priority based scheduler
 - ▶ One main GUI thread per process/application
- ▶ **Differences**
 - ▶ Abstract structure
 - ▶ Communication
 - ▶ File system
 - ▶ iOS is more closed and secured



Reference

- ▶ [1] <http://developer.android.com/reference>
- ▶ [2] [http://en.wikipedia.org/wiki/Android_\(operating_system\)](http://en.wikipedia.org/wiki/Android_(operating_system))
- ▶ [3] A Survey of Embedded Operating System, Catherine W., etc
- ▶ [4] <http://developer.apple.com/library/ios/navigation/index.html>



ROS: Robot Operating System

Ian Perera

Why do robots need an OS?

- ▶ Multiple programs running at once
 - ▶ Navigation
 - ▶ Planning
 - ▶ Networking
- ▶ Multiple sensor inputs collecting data
 - ▶ LIDAR (laser mapping)
 - ▶ Touch sensors
 - ▶ Audio sensors
 - ▶ Vision sensors
- ▶ All of these systems have to work with each other



What are the challenges faced?

- ▶ Sensors use proprietary interfaces, data formats
- ▶ Software is written by different developers
 - ▶ Must be able to send and translate data between executable components
- ▶ Most of this must take place in real-time
- ▶ Would like some processing on robot, and more intensive processing off-board
 - ▶ Data transferred over network
 - ▶ Android OS has similar requirements



ROS: Robot Operating System

▶ Goals:

- ▶ Hardware abstraction
- ▶ Interprocess communication
- ▶ Easy and efficient development iteration
- ▶ Robot-specific systems

▶ Methods:

- ▶ Modular computation system
- ▶ Broadcast-Subscriber message passing system
- ▶ Dependency graph compilation
- ▶ Exposed user functionality somewhere between smartphone OSes and TinyOS



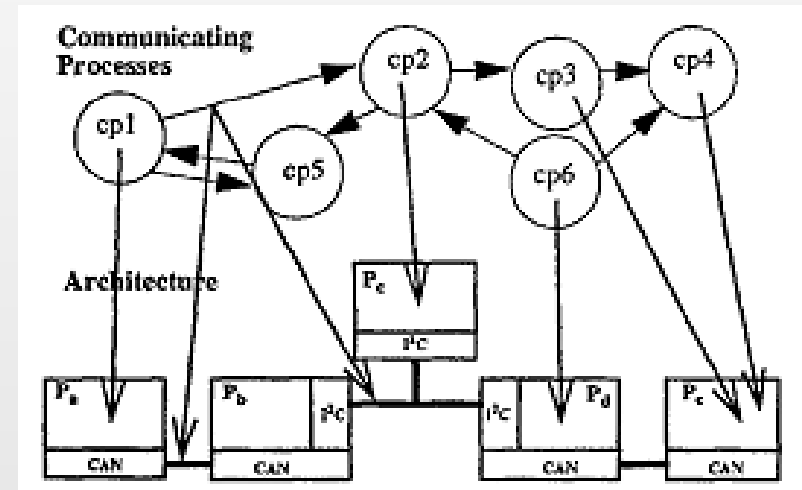
Is ROS really an OS?

- ▶ Shares many of the same principles
 - ▶ Hardware abstraction
 - ▶ Process management
 - ▶ Data sharing
 - ▶ Filesystem (package management)
- ▶ But...
 - ▶ Doesn't really "schedule" processes
 - ▶ Executing code should be data-driven and independent of the operating system
 - ▶ Dependencies on OS information are discouraged



Precursor to ROS

- ▶ Distributed embedded computing for robotics systems
- ▶ Direct message passing system between communicating processes and devices
- ▶ Required defined API and translation of data types between processes
- ▶ Messages must be timed to ensure they arrive when processes are waiting on them



Nodes

- ▶ Modularity and hardware abstraction is accomplished by separating device drivers and software into separate “nodes”
- ▶ Consists of:
 - ▶ A unique ID (graph resource name)
 - ▶ A namespace (data protection and prevention of naming conflicts)
 - ▶ Executable code
- ▶ Can:
 - ▶ Publish to topics
 - ▶ Subscribe to topics



Nodes (continued)

- ▶ Dividing computation into fine-grained modules also abstracts away the computing hardware
- ▶ Computation can be distributed between the robot's CPU and the offboard CPU (if necessary)

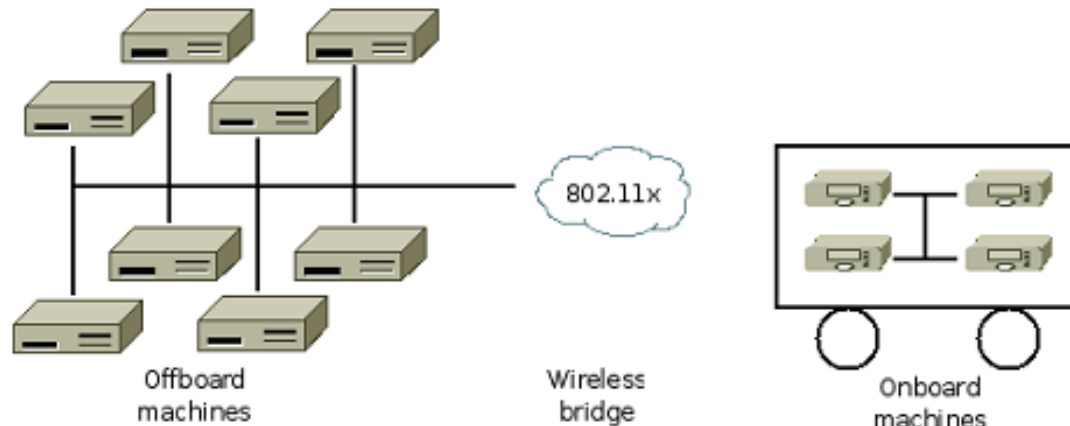


Fig. 1. A typical ROS network configuration

Nodes – not quite processes

- ▶ Nodes are higher level than processes in a typical operating system
 - ▶ Runs C++, Python, Lisp, or Octave natively in ROS
 - ▶ Not compiled into bytecode by ROS
 - ▶ Modules (roscpp, rospy) execute the code
 - ▶ ROS libraries and modules handle memory mapping for various robotics platforms
 - ▶ ROS's only specification is at the message level – communication between nodes
 - ▶ All nodes are given equal, real-time priority
 - ▶ Fixed functionality, but encouraged interdependence on anonymous nodes through messages



Message Passing

- ▶ Messages are used for communication between nodes
 - ▶ Similar to Android OS for interprocess communication
- ▶ A typed data structure
- ▶ Can be declared in either C++ or Python
 - ▶ ROS converts to a universal message type (.msg)
- ▶ Messages can be nested

```
LTLAction.msg
```

```
string verb  
string[] targets
```

```
LTLPacket.msg
```

```
LTLAction[] actions  
Scene scene
```



Message Passing Protocol

- ▶ No distinction between nodes on the same system and remote
- ▶ Network message transport can be done using TCP or UDP
- ▶ TCP
 - ▶ Reliability necessary, low-latency not
 - ▶ One-shot detector data, non-time sensitive data
- ▶ UDP
 - ▶ Low-latency necessary, reliability unnecessary
 - ▶ Redundancy
 - ▶ Video, audio streams



Topics

- ▶ Used in the passive many-to-many message passing system
- ▶ A topic is simply a unique string
- ▶ Nodes can anonymously broadcast and subscribe to topics
 - ▶ Improves modularity
- ▶ Any message broadcasted to a topic is received by all subscribers
- ▶ Topics are typeless, but nodes only receive messages that match their types



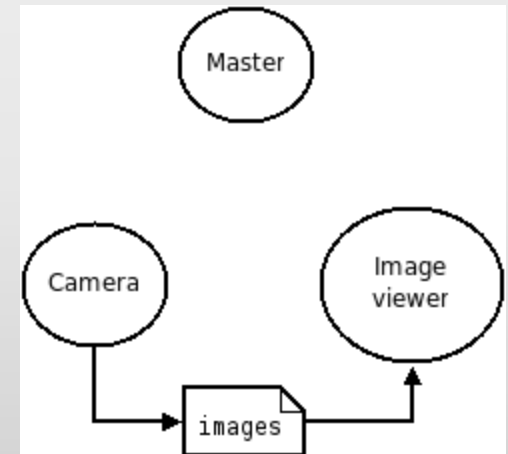
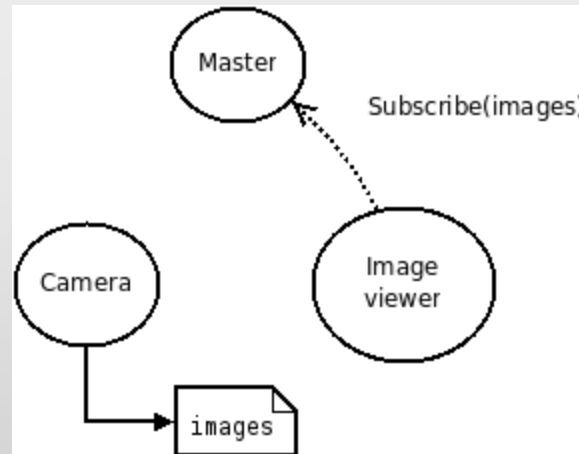
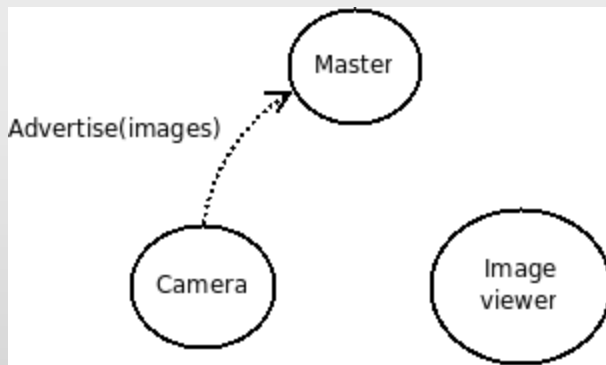
Services

- ▶ Alternative to broadcast system
- ▶ Active request-response system
- ▶ A Service is a named interface to a particular node (the server)
- ▶ Client node makes a request of a server node by querying a service associated with the server node



ROS Master

- ▶ A singleton that coordinates message passing
- ▶ Acts similarly to a DNS server
- ▶ Keeps track of all messages, topics, and services
 - ▶ Nodes may subscribe after a message has been broadcasted
 - ▶ Removes dependency on timing of messages



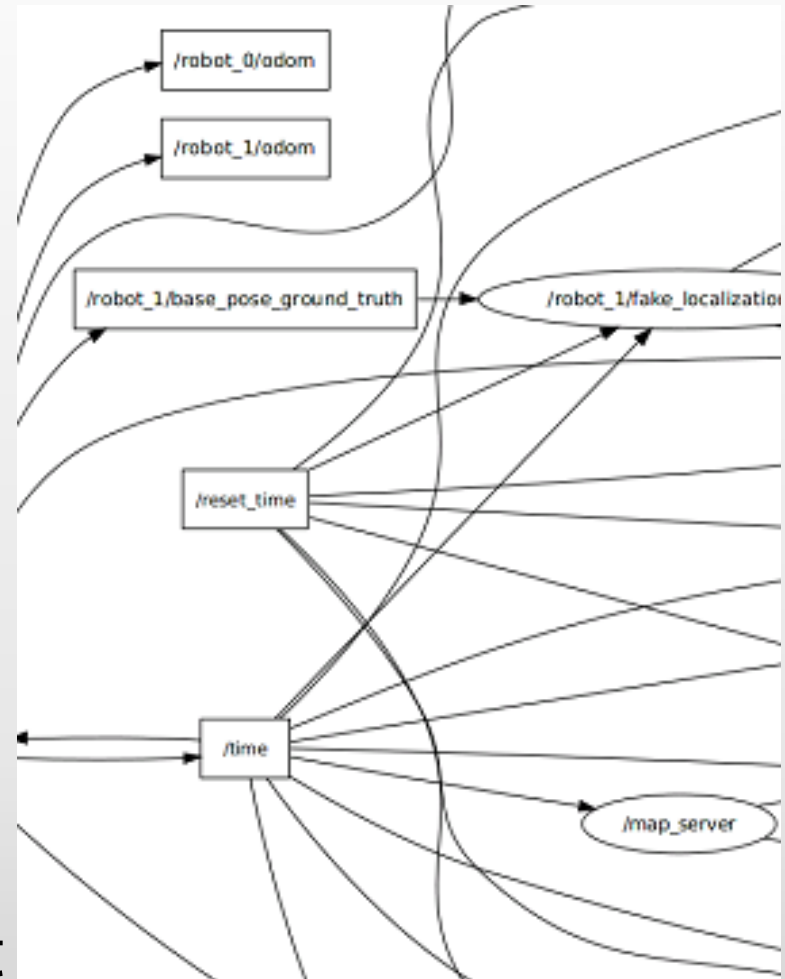
Coordinate Space Abstraction

- ▶ Using data from sensors that return spatial information requires a consistent coordinate space
 - ▶ Is the Z-direction in the direction of the robot, or some axis in the environment?
 - ▶ Different sensors might have different spaces
- ▶ Transformations between coordinate spaces handled automatically through the tf system
 - ▶ ROS stores a transformation tree to automatically perform the required transformations for spatial data
- ▶ Devices do not need to consider the coordinate space of any devices other than themselves



Dependency Graph

- ▶ Nodes are connected via dependency links in a graph
- ▶ Can make changes to source code without stopping the execution independent nodes
 - ▶ Real-time development, unlike kernel (or even most user-level) development
- ▶ Certain devices may have a long startup time or may require constant data input



Comparison to TinyOS

▶ Differences:

- ▶ Power consumption and memory footprint not as crucial (same with smartphone OSes)
- ▶ Dynamic modularity
 - ▶ TinyOS is hard-wired once compiled
 - ▶ ROS can switch out modules while code is running
- ▶ Interface
 - ▶ TinyOS provides a system for building a custom OS
 - ▶ ROS provides a system for building custom robot system
- ▶ TinyOS is event-driven rather than message-driven



References

- ▶ M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, “ROS: an open-source robot operating system,” in Open-source software workshop of the Int. Conf. on Robotics and Automation, Kobe, Japan, 2009.
- ▶ R.B. Ortega and G. Borriello, “Communication Synthesis for Distributed Embedded Systems,” Proc. Int’l Conf. Computer-Aided Design (ICCAD 98), IEEE CS Press, Los Alamitos, Calif., 1998, pp. 437-44.
- ▶ ROS Wiki : <http://www.ros.org/wiki/>





TinyOS



Presentation by Carolyn Keenan

Overview

- Description of TinyOS
 - Why is it necessary?
 - How is it different from previous embedded OSes?
- Basic structure of TinyOS – event-based
 - Sample application
- Specific features of TinyOS :
 - Concurrency
 - Address binding
- Implementation examples



What is TinyOS?

- Very small, flexible “programming framework”
 - Less than 400 bytes!
- Written in NesC (a dialect of C)
- Used in embedded systems, specifically :
 - Sensor networks
 - Personal Area Networks
 - Smart buildings
 - Smart meters



Relation to other Embedded Systems

- **Comparison to Android :**
 - Modularity still essential
 - Resource usage is biggest issue
 - Little to no user interaction
- **Comparison to RobotOS :**
 - No subscription-based message passing
 - Both link network of sensors
 - TinyOS : event-based



An OS for sensor networks

- Why not use Windows Vista?
 - Motes (sensor nodes) are intentionally tiny!
 - 1-MIPS processor, 10KB of storage
 - Low power
- Special needs for sensor networks
 - Real-time, reactive concurrency
 - Flexibility

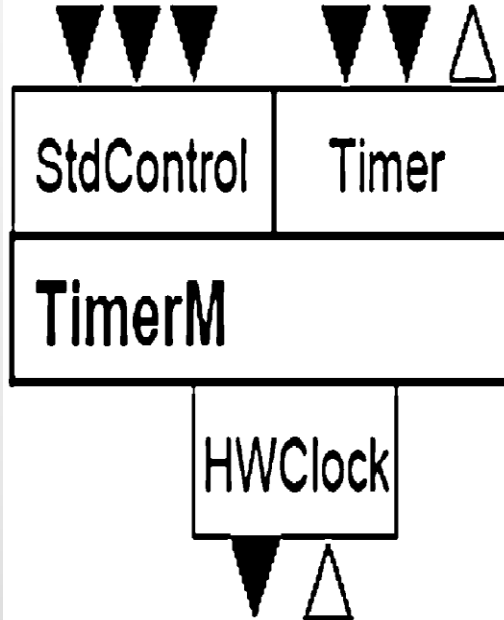


Basic Idea

- What does TinyOS itself contain?
 - Reusable system components
 - Scheduler
- What does the programmer write?
 - *Components* -
 - Modules : specify *interfaces* (services) which the component provides and uses
 - Configurations : wires interfaces together (more on this)
 - *Wiring specification*



A sample module...



```
module TimerM {
    provides {
        interface StdControl;
        interface Timer[uint8_t id];
    }
    uses interface Clock;
}
implementation {
    ... a dialect of C ...
}
```



Interfaces for Clock

```
interface StdControl {
    command result_t init();
    command result_t start();
    command result_t stop();
}

interface Timer {
    command result_t start(char type, uint32_t interval);
    command result_t stop();
    event result_t fired();
}

interface Clock {
    command result_t setRate(char interval, char scale);
    event result_t fire();
}

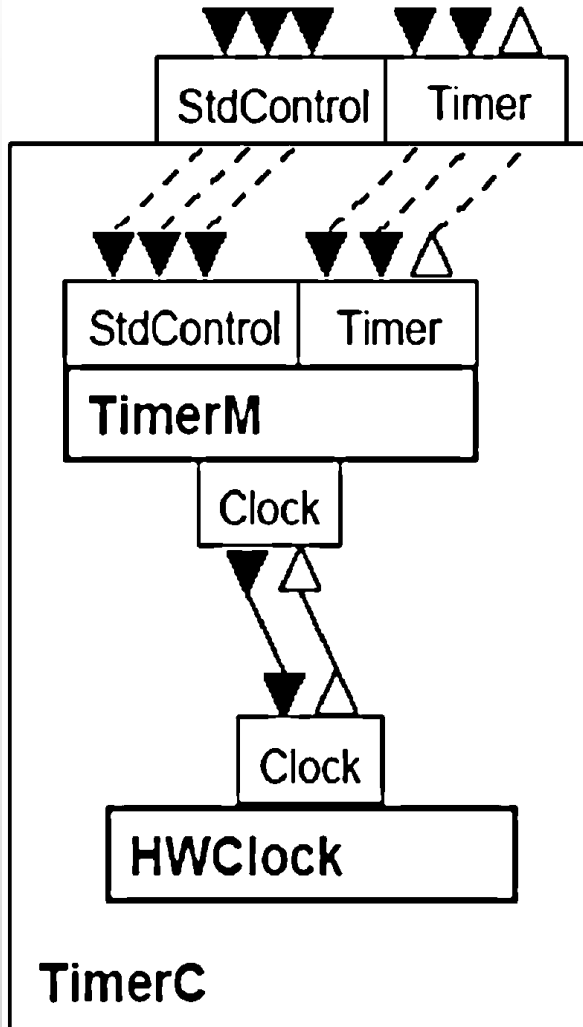
interface SendMsg {
    command result_t send(uint16_t address,
                          uint8_t length,
                          TOS_MsgPtr msg);
    event result_t sendDone(TOS_MsgPtr msg,
                            result_t success);
}
```

Commands and Events

- *Command* – “downcall” in traditional OS
 - Sort of like a system call
- *Event* – “upcall”
 - signal that the command has been completed
- These are *split phase* :
 - Command returns immediately
 - Event signals completion later



Timer Configuration



```
configuration TimerC {  
    provides {  
        interface StdControl;  
        interface Timer[uint8_t id];  
    }  
}  
implementation {  
    components TimerM, HWClock;  
  
    StdControl = TimerM.StdControl;  
    Timer = TimerM.Timer;  
  
    TimerM.Clk -> HWClock.Clock;  
}
```

Execution and Concurrency

- Everything is event-centric
- Computation is done by *tasks*
 - *Run-to-completion vs. run indefinitely (threads)*
 - Cannot be preempted
 - Problem : race conditions within interrupt handlers or commands



Concurrency, continued

- Synchronous code (SC) : only reachable from tasks
- Asynchronous code (AC) : reachable from at least one interrupt handler.
- Traditional OS approach : minimize AC, and keep it at the kernel level.
 - Too strict for real-time sensor networks.
 - NesC compiler detects nearly all race conditions



Typical race condition and fix

```
/* Contains a race: */  
if (state == IDLE) {  
    state = SENDING;  
    count++;  
    // send a packet  
}
```

```
/* Fixed version: */  
uint8_t oldState;  
atomic {  
    oldState = state;  
    if (state == IDLE) {  
        state = SENDING;  
    }  
}  
if (oldState == IDLE) {  
    count++;  
    // send a packet  
}
```



Address binding

- NesC compiler can generate
 - Absolute code
 - Relocatable code
- Result : no virtual memory
 - Not like you'd want it
- Problem : need to “burn” the code into the mote every time you change it.
 - Recent solution : “incremental reprogramming” (W. Dong et al, 2011)



Absent features

- In contrast to Linux and other OSes, TinyOS lacks
 - Virtual memory management
 - File system
 - Protection (users, groups, etc.)
 - Complex message-passing
 - Done through events



Implementations

- Habitat monitoring system on Great Duck Island
 - Nodes left unattended for 4 months
 - Sampled environmental data
- Object-tracking
 - Track remote-controlled car moving through sensor field
- TinyDB
 - SQL-like queries retrieve data from sensors



Sources

P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *TinyOS: An operating system for wireless sensor networks*. In *Ambient Intelligence*. Springer-Verlag, 2004.

Wei Dong; Yunhao Liu; Chun Chen; Jiajun Bu; Chao Huang; , " $\mathcal{R}2$: Incremental \mathcal{R} eprogramming using \mathcal{R} elocatable code in networked embedded systems," INFOCOM, 2011 Proceedings IEEE , vol., no., pp.376-380, 10-15 April 2011

TinyOS programming guide :

<http://csl.stanford.edu/~pal/pubs/tos-programming-web.pdf>

TinyOS wiki :

http://docs.tinyos.net/tinywiki/index.php/Main_Page



Summary

- ▶ **Modularity**

- ▶ Custom tailoring features reduces space, provides flexible interface for programming
- ▶ Processing power not wasted on unneeded features

- ▶ **Communication between processing units (processes, nodes, sensor chips, mobile devices over a network) essential**

- ▶ **Must run on a wide range of different devices**
 - ▶ **Must run in real-time**
-

