OSAS
An Open Service Architecture for Sensors

Johan Lukkien
Richard Verhoeven
Remi Bosman
Framework

- A framework consists (possibly) of
  - a ‘static’ part
    - programming model, data model
      - libraries
    - life cycle model
    - methods or tooling for development
  - a ‘dynamic’ part
    - a run-time system, or platform
      - entirely separate entity or a library
    - a set of services
      - provided by the platform
      - e.g. binding, installation
    - a process model

- A framework has views, e.g.,
  - *logical view* for the framework user (application developer)
    - programming model
    - visible services
  - *development view* for the framework developer, programmer
    - the logical organization of the framework tools and platform
    - the services structure
    - the code
  - *process view* for developer and framework installer
    - the processes in the framework, the connection to the OS, the protocols
  - *deployment view*
Contents

• Introduction to the design problem
  – system operation sketch, goal
  – stakeholders
  – technical environment
  – operational environment
  – constraints

• Analysis

• Design decisions

• Reflection on concepts

• Conclusion
System outline

• The system consists of
  – clusters of wireless sensors
    • sensing, (actuating), computing, communicating
    • cheap, small, mobile, unreliable (communication), low on resources, many
    • mobile (wearing) and ambient deployment
  – infra structure, bridge-ing and backend-machines

• The goal:
  – develop a programming framework for sensor networks
    • specify sensor behavior, and computations
    • adjust sensor behavior
    • integrate in infrastructure
    • ‘convenient’ deployment (impossible to physically contact each sensor)
Elderly care: person moving between different locations

802.16e (WiMAX)
UMTS GPRS

802.11 (WLAN)
Herd control: monitoring animals in the field
General architecture

Physical organization

802.16e (WiMAX)
UMTS, GPRS

802.11 (WLAN)

IP-based (cellular) network with wide-area coverage

WSN infrastructure: static ambient nodes using the same communication technology as lowest layer

Moving clusters of nodes with rich sensing capability but with limited resources
e.g. single person moving around

MSN = mobile sensor network
ASN = ambient sensor network

Tuesday, September 28, 2010
Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Computer Science, System Architecture and Networking
## Taxonomy

<table>
<thead>
<tr>
<th>Bottom layer</th>
<th>Middle layer</th>
<th>Multi hop</th>
<th>Single hop (no hop)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Multi hop</td>
<td></td>
<td>Single hop (no hop)</td>
</tr>
<tr>
<td></td>
<td>(ambient infra structure)</td>
<td></td>
<td>(access points)</td>
</tr>
<tr>
<td>Multi hop</td>
<td>Most general case: moving clusters through ambient infra structure</td>
<td>Moving clusters connecting to access points</td>
<td></td>
</tr>
<tr>
<td>Single hop</td>
<td>Moving nodes connecting to ambient infra structure</td>
<td>Moving nodes connecting to access points</td>
<td></td>
</tr>
</tbody>
</table>
System outline

- The system consists of
  - clusters of wireless sensors
    - sensing, (actuating), computing, communicating
    - cheap, small, mobile, unreliable (communication), low on resources, many
    - mobile (wearing) and ambient deployment
  - infra structure, bridge-ing and backend-machines

- The goal:
  - develop a programming framework for sensor networks
    - specify sensor behavior, and computations
    - adjust sensor behavior
    - integrate in infrastructure
    - ‘convenient’ deployment (impossible to physically contact each sensor)
Stakeholders

- **Users** (farmers, doctors)
  - buying and deploying sensors
  - configuring sensor systems
  - **note**: the person that *wears* the sensors is not considered as stakeholder here

- **Application builders**
  - programmers, programming sensor systems
  - putting systems together

- **Integrators**
  - integration of new hardware
  - developing and integrating system software
Scenario of running system
First visit - configuration

Collect all sensors every 15 minutes
Running system
After visit - running

SpO₂
Heart rate
Motion

Sensor data collected

gateway
Running system
Next ‘visit’ - analysis

Something is wrong with the SpO₂ levels!!!

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Computer Science, System Architecture and Networking

Tuesday, September 28, 2010
Running system
Next ‘visit’ – change configuration

Collect SpO₂ sensor every 2 minutes
Running system
Next ‘visit’ – set an alarm

SpO₂
Heart rate
Motion

gateway

Raise alarm when SpO₂ reading below 25
Notify within 1 minute
Running system
After ‘visit’ – running

Retrieval SpO₂ data
Stream SpO₂ sensor every minute
Building blocks & constraints

• Physically: mentioned devices
  – several kilobytes of flash, 2-10kB of RAM, slow clock
  – small batteries, small range radio
  – error-prone wireless communication

• Software (explained later):
  – on nodes:
    • Operating System, providing *system calls*
      – basic OS functionality, but little protection
      – sensor / actuator access
    • Network stack
    • to-be-developed components, using a language of choice (typically C)
  – in the infra structure:
    • standard platform (e.g. Windows)
    • to-be-developed components, using a language of choice
Extra-functional properties

- Energy constraints
  - operational for months on small batteries

- Long time, no touch

- Programmer productivity

- Portability, limit platform dependence

- (Security, privacy)
Technical environment

• For our design we used:
  – Mantis OS, with a simple link layer protocol in the sensors
  – C as programming language
  – both Linux and Windows platforms in the back-end systems
  – Python, on Linux and Windows
  – A Compiler-Compiler

• Note:
  – these are not always given constraints but choices resulting from research
  – it is debatable to what extent such choices influence the architecture. However:
    *The concepts that someone works with determine the way of thinking about a problem and the choice of solutions*
Use cases

• Use case of a doctor (see previously)

• Programmer:
  – programming model
  – workflow of writing, deploying and debugging programs for entire networks
    • specifying behavior and computations of entire system
  – special issues:
    • very limited feedback possibilities from sensors
Contents

• Introduction to the design problem

• Analysis

• Design decisions

• Reflection on concepts

• Conclusion
Assumptions about node software environment

• Common assumptions, for systems more powerful than e.g. PDAs
  – POSIX OS, supporting regular OS services
    • process, thread management
    • file, memory and other resource management
    • i/o
  – TCP/IP protocol stack

• Sensor node capacities are so small that
  – no superfluous functionality should be supported
  – in fact, the running of an application should be fully optimized
  – typically this is done by *cross-layer optimization*
    • breaking the conceptual layering, by using information at different layers to obtain global optimization
      – e.g. application-dependent use of the wireless link (app. dependent MAC)
      – e.g. integrating OS functions, applications and communication

• Hence,
  – embedded OS’s, if any, with very different programming models
  – no TCP/IP – just a simple link layer communication with neighbors
Model in development view
Program life cycle

- Adaptations at runtime range from reconfiguration to reprogramming
  - hence, program deployment upon system setup as well as during operation

- ‘Traditional’ means of deploying distributed systems:
  - compile, store and run a program for each machine
    - typically by having physical access
    - example: MPI and PVM programs started as a distributed virtual machine
  - client-server:
    - server ‘always on’
    - client code started manually, e.g. after downloading
  - configuration (parameters): encoded by the programmer, as part of the program

- We investigate:
  - how to deploy program code
  - what to deploy (partial or full binary, intermediate code)
  - when to add configuration
Traditional development applied to sensors

Prog 1 → Compiler → Binary 1 → Loader → Binary 2 → Binary 3 → SpO2

Prog 2 → Compiler → Binary 1 → Loader → Binary 2 → Heart rate

Prog 3 → Compiler → Binary 1 → Loader → Binary 3 → Motion

SpO2

Heart rate

Motion

Deployment

Tuesday, September 28, 2010

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Computer Science, System Architecture and Networking
‘Performance’ of deployment procedures

- Deployment procedures have independent choices:
  1. send code through a physical connection (A) or send code over the air (B)
  2. install specific code for each node (A) or send the same code to each node (B)

- Qualities aspects of these choices: performance & reliability
  - performance: communication volume (= energy) and time spent
    - 1A:
      - takes ~minutes per sensor (extract sensor from environment; attach to server; deploy (again) in environment)
      - obtaining the sensor physically may be prohibitive
      - may integrate with normal operation procedures: seeking the right moment for update
    - 1B, 2A (needs reliable communication)
      - the volume sent grows as the number of nodes
      - note that not all nodes are involved; however, nodes close to the source will do more work
      - may integrate with normal operation procedure
    - 1B, 2B (reliable multicast)
      - the code is sent just once (experiments: next slide)
Simulation timing of reliable multicast

100 nodes, randomly in a square (200mx200m) configuration, average delay per packet of code (128b) to be disseminated to a percentage of nodes

Injection rate is 1/s

A full binary is usually around 25kB

The time is determined by the diameter of a spanning tree built for this multicasting (here roughly 7.7). This could become much larger in certain cases.

Scalability of deployment procedures

• Usage parameters:
  – number of nodes
  – code size

• Metrics:
  – delay, energy

• Scalability criterion:
  – better than linear dependence
  – small constants (e.g. per node handling penalty)

• In order to keep delay and energy small, our architecture
  – must support loading over the air
  – have a single, small code for all nodes
    • or more precisely: a small number of different code classes

• Note that network diameter is not under our control
Analysis & design choices

Code specialization per node
- before deployment
  - node-specific code
  - simpler, for a node
- at or after deployment
  - all nodes same code
  - requires selection mechanism on node

Deployment options
- Compact code
- Machine & OS independence
- Interpreter on nodes
- Good for coordination code
- Compact code (but less)
- Machine & OS dependence
- Linkloader on nodes
- Good for computational code
- Large code
- Machine & OS dependence
- Gives most efficient run-time

Program for network
- Compiler
- Intermediate code (e.g. bytecode)
- Jit / assembler
- Machine code
- Linkloader (add libs+OS)
- Runnable

Interpreter
- static configuration (config info given once)
  - useful mainly for static deployment (utmost performance)
- dynamic configuration (config info input to running program)
Contents

• Introduction to the design problem

• Analysis

• Design decisions
  – decisions
  – results

• Reflection on concepts

• Conclusion
Applied styles

• Architecture style
  – publish & subscribe
  – service oriented
  – virtual machine

• Interaction style
  – event-based
  – asynchronous RPC
  – active messages
Event-based interaction style

- **Event:** change in state of an entity
  - typically related to observations, or time evolution
  - entity: e.g. software component, object, machine

- **Event notification:**
  - asynchronous message, without connection or acknowledge
  - asynchronous invocation, without result (cf. asynchronous RPC)
  - connector: *event bus, event dispatcher*

- **Event-based interaction style:**
  - interaction between entities is through event notifications only

- **Motivation:**
  - decoupling in *synchronization*, delay *binding*

- **Extended in an architectural style by adding architectural elements**
  - eventing subsystem, “brokers”
Design decision: services

- (after further literature search): Service Oriented Architecture

- **Services** are delivered by nodes through exposed interfaces on the network, comprising
  - **events**: sampled conditions
    - with a given frequency the condition is checked; when *true*, the event 'occurs' (an event body is executed, resulting in notifications)
  - **actions**: procedure call (without result)

- **Subscriptions**: fill in event *parameters*; connect events and actions
  - event generates a notification: an asynchronous remote procedure call
  - *publisher*: event generator service
  - *subscriber*: service of which an action is (possibly) triggered
  - the sampling rate, and other parameterization *is determined by the subscriber*

- A **system service** on each node comprises
  - installing/managing new services
  - establishing subscriptions
Process / development model

![Diagram showing a process development model with nodes, publishers, subscribers, and runtime systems.]

- **Publisher** (e.g., Temp service)
- **Subscriber** (e.g., a temp value accumulator)
- **Runtime system**

```
on event Temp when true do
    SendToSubscribers (Accumulate, Temp())
end

on event Tempalarm when Temp>40 do
    SendToSubscribers (AlarmCall, Temp())
end
```

Need interface here to inform OS about optimization possibilities

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Computer Science, System Architecture and Networking
28-Sep-10

OS

- byte code interpreter
- system calls
- service creation
- message handling

Network interface

Hardware

- OS API e.g., sensor inspection
- send (dest, ...) receive (...)

Link layer

Need interface here to inform network stack about optimization possibilities (e.g., sleep modes)
Application builders view on WSN (deployment view)

**Program:** event based; specification of node services and interconnections

(logical) connections pass events (asynchronous RPCs without result) packaged as active messages

Gateway
A (logical) model in the development view

- The mobile node clusters at the bottom
- Gateway bridges node domain with UDP
  - can connect multiple WSNs in this way
- Simulated nodes interpret the exact same messages
- Simulated nodes
  - have access to arbitrary PC and Internet services
  - may provide services to access the WSN
- Broadcast traffic injected in UDP appears in the node domain as well
  - in this way a loader can upload code
- Rather flat: two layers
Design decision: addressing

• A message address is a boolean function that evaluates to true on a destination
  – parameters to this function may be provided
    • within the message
    • within the node

• Special cases are dealt with in the system service:
  – a (regular) destination address
  – a broadcast address

• Example:
  – ‘node contains a temperature sensor’
  – ‘local temperature is larger than 25’
    • 25 as parameter in message

• We call this content based addressing
Design: Language & semantics

Node 1 & 2

Address table:
0 : true // HasSysCall(Temp)
1 : false // NodeID==3
2 : true // HasService(TempService)

Sys Calls
HasSysCall
Temp
NodeID

TempService

service TempService($Handler)
on event read_temperature when True do
SendToSubscribers($Handler, Temp())

AvgTempService

define
sum := 0
count := 0

on event flush_avg when 2 <= count do
SendToSubscribers($Handler, sum / count);
count := 0; sum := 0

action ReportTemp(temp) do
sum := sum + temp;
count := count + 1

Node 3

Address table:
0 : false // HasSysCall(Temp)
1 : true // NodeID==3
2 : false // HasService(TempService)

Sys Calls
HasSysCall
NodeID

AvgTempService

event
flush_avg

State
sum
count

service AvgTempService($Handler)

don AverageTemp to AvgTempService($Handler=ReportTemp)

for [Network|*|HasSysCall(Temp)]
install TempService

for [Network|*|NodeID()=3]
install AvgTempService
install AverageTemp on
[Network|*|HasService(TempService)]

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Computer Science, System Architecture and Networking
Design decision: messages

- A message fits in a single packet; the format corresponds closely to the asynchronous procedure call (aka *active messages*)
  - the payload is a *handler* and parameters to this handler
    - *handler*: a user-defined service action or system service action

<table>
<thead>
<tr>
<th>Application ID</th>
<th>QoS</th>
<th>Deadline</th>
<th>Hop</th>
<th>Payload</th>
<th>CRC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Handler</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Upon receipt of a message, a node executes the handler if it understands it

- The following are examples of messages that encode a call to a specific handler
  - subscriptions
  - flooding
  - code upload

<table>
<thead>
<tr>
<th>Sub</th>
<th>Event</th>
<th>Period</th>
<th>NodeID</th>
<th>Callback</th>
<th>PrioID</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Composition

- Parameters to a handler can be:
  - (remainder of) message
  - (handler, parameter) pairs

- This allows
  - conditional execution
  - forwarding
  - having more than one handler
  - adapting a message while handling and forwarding
Design decision: virtual machine (byte code)

- The compiler translates a program into byte code to be executed by a virtual machine on nodes

- Fairly standard stack machine with respect to computations
  - Focus on relating system calls to messaging

- Special: memory organization with respect to context
  - the code of an event is executed in the context of each subscription
    - parameters are found with the subscription
  - the code of a (message) handler is executed in the context of the received message
    - that can provide parameters to this handler
  - global variables of the service are shared by handlers and actions
  - global variables in the node are shared by the services
**Contexts**

- **Ncontext**
  - Standard handlers
  - System calls
- **Gcontext**
  - Shared state
- **Scontext**
  - Timing, params
  - Subscribers
- **Mcontext**
  - Handler args

- The VM has instructions to refer to each context
  - e.g. `PUSHG 1`
Example: code load

- Configuration handler
  - Built-in handler
  - Install content-based addresses
  - Install new services
    - (initialized) variables
    - events
    - actions (handlers)
- Bytes
  - VM code
  - Native code
  - ....

<table>
<thead>
<tr>
<th>Configuration handler</th>
<th>Config instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CON</td>
<td>Config instructions</td>
</tr>
<tr>
<td>DEFA</td>
<td>AddrID total length offset bytes</td>
</tr>
<tr>
<td>INIT</td>
<td>length bytes</td>
</tr>
<tr>
<td>SERV</td>
<td>serviceID CRC</td>
</tr>
<tr>
<td>STATE</td>
<td>total size</td>
</tr>
<tr>
<td>DEFG</td>
<td>varID size</td>
</tr>
<tr>
<td>DEFGI</td>
<td>varID total length offset bytes</td>
</tr>
<tr>
<td>DEFE</td>
<td>eventID total length offset bytes</td>
</tr>
<tr>
<td>DEFH</td>
<td>handlerID total length offset bytes</td>
</tr>
<tr>
<td>DEFF</td>
<td>params functionID total length offset bytes</td>
</tr>
</tbody>
</table>
Configuration example

- **Two** handler, calls two other handlers
  - Install address (CON)
  - Evaluate Content-Based Address
    - with CON as parameter

- When address holds:
  - install State
  - install Event:
    - Flush
    - install Handler:
      - StoreValue0

- Notice: all in a single (~80 byte) message
Contents

• Introduction to the design problem

• Analysis

• Design decisions
  – decisions
  – results

• Reflection on concepts

• Conclusion
What is OSAS?

• A programming system for networked devices
  – with special emphasis on low-resource devices

• Definition of
  – language
    • with the service and subscription concept
    • and content-based addressing
      – use a predicate to refer to a (set of) nodes
  – virtual machine (byte code)
    • has access to a set of system calls (Library and OS-provided functions)
  – message format

• Four components
  – Compiler
  – Loader
  – Runtime system
  – Simulator
    • transparent; interprets the exact same message format and can be part of the network
Development cycle & toolchain (logical view for programmer)

1. wsp
2. Compiler
3. Loader
4. Simulator

1. wnc
2. wbc
3. wnl
4. wsl

SpO₂
Heart rate
Motion
Gateway

Tuesday, September 28, 2010
Virtual or real

Simulated nodes act as if they are part of a network

- Test a single node
- Connect multiple sites
- Test a cluster
- Test routing protocol
- Simulate a scenario
Contents

• Introduction to the design problem
• Analysis
• Design decisions
• Reflection on concepts
• Conclusion
Which role play the concepts?

- **Processes**
  - client-server relations based on *third party* binding
  - each sensor acts as separate process

- **Communication**
  - event: asynchronous remote procedure call without result
  - link (neighborhood communication) as basis
    - layer 2, or overlay

- **Naming and addressing**
  - content-based addressing
    - implemented on top of flooding
  - neighborhood addressing, can be used to build routing

- **Reliability**
Conclusion

• The goal:
  – develop a programming framework for sensor networks
    • specify sensor behavior, and computations
      – This is derived from a single program for the entire network
      – The compiler uses global knowledge for optimization
        » no discovery or interpretation of functionality needed
      – *Computations*: delegated to pre-installed libraries (weak point)
      – *Coordination*: events, subscriptions
    • adjust sensor behavior
      – by changing subscriptions
    • integrate in infrastructure
      – simulated nodes are part of the network but support more powerful OS calls
    • ‘convenient’ deployment (impossible to physically contact each sensor)
      – deployment through the air using content-based addressing
Memory resources

**Memory Footprint**

TinyOS version for ICL nodes (values in bytes)

**Static**

- OS image: 24084 ROM, 1191 RAM
- interpreter: 1664 ROM, 20 RAM
- system calls: 1530 ROM, 146 RAM

**Dynamic**

- bytecode evaluation stack: 160
- content based address: 4 + \(\text{length(bytecode)}\)
- service: 14 + \(#\text{global variables} \times 2\)
- event generator: 4 + \(\text{length(bytecode)}\)
- action: 6 + \(\text{length(bytecode)}\)
- subscription: 16 + \(#\text{parameters} \times 2\)
- message: 12 + \(\text{length(payload)}\)
Conclusion

- I did not address the development view in detail
  - modules:
    - run-time system with node OS
    - simulator, loader, compiler organization
      - perhaps sharing of data structures
  - files:
    - directory organization

- The current system has been built according to this architecture and operates satisfactorily
  - needs improvements in reducing energy

- The architecture as well as the design were driven mainly by extra-functional properties, and directed by careful analysis

- Ideas can carry over to ‘regular’ systems; however, their value is not obvious then
  - current trends are towards independent services that determine their mode of cooperation by extensive processing (e.g. web services with ontologies)
Some studied approaches

• Virtual machines

• Special-purpose engine

• Macro-programming
Approaches (cnt’d)

• Active messages

• IP to the sensors

• Content-based addressing