Eliminating Implicit Dependencies in Component Models for Networked Embedded Systems

Danny Hughes
Katholieke Universiteit Leuven
danny.hughes@cs.kuleuven.be
Structure of This Talk

1. What is a component model?

2. Critique of contemporary component models

3. The Loosely-coupled Component Infrastructure

4. Conclusions
Structure of This Talk

1. What is a component model?

2. Critique of contemporary component models

3. The Loosely-coupled Component Infrastructure

4. Conclusions
Defining Components

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition.”

- Clemens Szyperski
Microsoft
Contractually Specified Interfaces

• Types of Interface:
  – *Provided* interfaces define the services that a component offers.
  – *Required* interfaces define the services that a component requires.

• Interfaces are specified in a standardized form and may be inspected.

• This enables us to easily compose custom software images optimized for a specific context.

• This is critical in NES, where we have very constrained resources.
Contractually Specified Interfaces

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition.”

- Clemens Szyperski
Contractually Specified Interfaces

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition.”

- Clemens Szyperski
Contractually Specified Interfaces

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition.”

- Clemens Szyperski
Application Compositions

- Component-based development consists of two phases:
  - Development of re-usable components.
  - Wiring of components into an ‘application composition’.

![Diagram of Fire Monitoring Composition](image)
Independent Deployable

• Components should be deployable as self-contained entities without support.

• This enables the injection of specific functionality at run-time without monolithic re-flashing.

• This enables:
  – Software evolution through component updates.
  – Adaptation through component replacement.

• This is an excellent fit with the dynamic characteristics of NES.
Characteristics of NES (1/2)

• Heterogeneity in multiple dimensions:
  – Computing, networking, sensing, software.

• Extreme resource constraints:
  – CPU, Memory and Networking.
  – Limited power supplies (battery or solar power).

• Dynamic operating environments:
  – Changing application requirements.
  – Tight coupling with environment.
Characteristics of NES (2/2)

• Unreliable:
  – Nodes use unreliable radio links.
  – Nodes power supplies may be exhausted.

• Highly distributed:
  – Typical applications use thousands or nodes.
  – Nodes have a range of may be deployed across 100s of KM of variable terrain.
3rd Party Composition

• Combination of: explicit interfaces and explicit context dependencies allows for 3rd party composition.

• As the component describes the services it provides, we can find compatible building blocks.

• As the component is independently deployable, we can treat it as a black box in our composition.

• The vision is that the developer is relieved from low-level details and becomes a ‘composer’ of components....
3rd Party Composition

- Combination of: explicit interfaces and explicit context dependencies allows for 3rd party composition.

- As the component describes the services it provides, we can find compatible building blocks.

- As the component is independently deployable, we can treat it as a black box in our composition.

- The vision is that the developer is relieved from low-level details and becomes a ‘composer’ of components.... *but there are complications.*
Explicit Context Dependencies

- Required software interfaces are one form of dependency, others include:
  - "Platform dependencies" on OS and hardware.
  - "Distribution dependencies" on IP, RMI, etc.
• Required software interfaces are one form of dependency, others include:
  – *Platform dependencies* on OS and hardware.
  – *Distribution dependencies* on IP, RMI, etc.
Explicit Context Dependencies

- Required software interfaces are one form of dependency, others include:
  - *Platform dependencies* on OS and hardware.
  - *Distribution dependencies* on IP, RMI, etc.
Introspection

• Dynamism and unreliability means that we cannot assume a fixed composition.
Introspection

- Dynamism and unreliability means that we cannot assume a fixed composition.
Introspection

• Dynamism and unreliability means that we cannot assume a fixed composition.

POWER IS INTERRUPTED AND NODE FAILS
Introspection

- Dynamism and unreliability means that we cannot assume a fixed composition.

POWER RESUMES AND NODE REACTIVATES
Introspection

- Dynamism and unreliability mean that we cannot assume a fixed composition.

BUT OUR COMPOSITION IS FAULTY DUE TO MISSING COMPONENTS.
Introspection

- Dynamism and unreliability means that we cannot assume a fixed composition.
• Dynamism and unreliability means that we cannot assume a fixed composition.
Introspection

• Dynamism and unreliability mean that we cannot assume a fixed composition.

• We need to inspect the composition at runtime and determine its state.

• This enables us to reason about composition state and sensibly reconfigure.

• The possibility of autonomic repair and reconfiguration is introduced.
Structure of This Talk

1. What is a component model?

2. Critique of contemporary component models

3. The Loosely-coupled Component Infrastructure

4. Conclusions
State of the Art (1/3)

• **NesC** is a popular WSN component model that supports TinyOS.
  – At development time, developers compose a custom runtime from generic components.
  – This application composition is optimized and compiled to a monolithic binary removing platform dependencies.
  – This binary can be deployed over the air on sensor nodes.
  – No distribution support is provided and distribution is embedded in components and implicit.

• As components do not persist at runtime we lose introspection and reconfiguration.

• We also cannot re-use components in new distributed contexts as their distribution mechanisms are hard-coded.
State of the Art (2/3)

• **OpenCOM/RUNES:**
  – Platform independent reconfigurable component model.
  – Compositions are supported by a minimal kernel that implements introspection and reconfiguration.
  – Distribution support is embedded in components or provided transparently by the runtime.

• This allows for runtime introspection, management and reconfiguration.

• But it doesn’t fix our distribution problems and platform dependencies are implicit.
State of the Art (3/3)

• **OSGi:**
  – Platform independent reconfigurable component model.
  – Distribution support is embedded in components or provided transparently by the runtime.
  – OSGi also provides optional support for describing platform dependencies.

• **REMORA:**
  – Platform independent reconfigurable component model.
  – Uses a high-level language to specify components that is compiled down to byte-code that is executed on the REMORA VM, removing implicit dependencies.
Against Transparent Distribution

• Transparent distribution allow local communication semantics to be applied in distributed fashion.

• Classic example is RPC:
  – Under the hood interfaces and receptacles map to procedure calls.
  – RPC provides an easy way to make local component models distributed.

• But this depends on distributed registries that are outside of the model and thus invisible to the application composer.
Against Transparent Distribution

- We built a large-scale flood warning system using OpenCOM.
- We demonstrated benefits but distribution was a problem.
Against Transparent Distribution

- We built a large-scale flood warning system using OpenCOM.
- We demonstrated benefits but distribution was a problem.
Against Transparent Distribution

- We built a large-scale flood warning system using OpenCOM.
- We demonstrated benefits but distribution was a problem.

WHAT IF (C) FAILS?
Against Transparent Distribution

• We built a large-scale flood warning system using OpenCOM.
• We demonstrated benefits but distribution was a problem.

OUR COMPOSITION IS FAULTY, BUT WE CANNOT TELL!
Against Transparent Distribution

• The introspection view should match the distributed view...

• ...If not, the application developer does not see the whole story.

• Worse still, as the RPC registry is not a component, it cannot be managed.

• What we really need:
  – All distributed dependencies must be explicitly specified.
  – Distribution topology must match composition.
Against Virtual Machines

• REMORA removes implicit dependencies by compiling everything to a common VM but access to lower layers is valuable as OS and hardware developers are clever guys:
  – Abstracting features to the lowest common denominator is like building a Skoda from Ferrari parts.

• Models such as TinyOS, RUNES, OpenCOM and OSGi allow low-level access.
  – This means they can access all the OS/hardware goodies.

• I’ll conclude with a look at another potential approach.
Contemporary Component Models

N/P = Not Possible, E = Explicit, I = Implicit
No Model Meets our Definition!

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition.”

- Clemens Szyperski
  Microsoft
Structure of This Talk

1. What is a component model?
2. Critique of contemporary component models
3. The Loosely-coupled Component Infrastructure
4. Conclusions
LooCI Principles

• **Match the Abstraction with the Environment:** WSN are inherently event-based, so this lowers overhead.

• **Harmonize Composition and Distribution views:** implicit dependencies are removed.

• **Separation of Distribution Concerns:** components do not know anything about distribution.

• **Supporting Dynamic Reconfiguration:** maintaining loosely-coupled bindings:
  – Facilitates the reification of distributed application state.
  – Minimizes the need for distributed quiescence protocols.
LooCI Features

• **Low-level Access:** full access to the lower layers.

• **Distributed Event Bus:** completely decentralized publish-subscribe communication.

• **Asynchronous Interactions:** best-effort event delivery, no blocking on send/receive.

• **Multi-model Bindings:** allow for one-to-one, one-to-many, many-to-many and opportunistic bindings.
The LooCI Architecture

- Reconfiguration Engine
- Component
  - Component
  - Component
- Event Manager
- Standardized Network Framework
- Underlying Platform (SunSPOT, Contiki, OSGi)
LooCI Components

- **LooCI Component**
  - is a *coarse grained* unit of functionality
  - Encapsulates local functionality and connects it to the Reconfiguration Engine (introspection, life-cycle control) and Event Bus (distributed composition).

- **Interaction**
  - Exclusively via *events* over explicitly declared interfaces:
    - *Provided Interfaces* model the publication of events to the event bus.
    - *Required Interfaces* model the subscription to events from the event bus.
LooCl Bindings

• *Decentralized publish/subscribe* communication medium that is *distributed across all sensor nodes*

• *Consistent Views*: the distribution view matches the composition view.

• The event manager stores *Binding* information used to dispatch events:
  - From *local* components to *local* or *remote* components.
  - From *remote* components to *local* components
## Runtime Memory Evaluation

### STATIC RUNTIME MEMORY CONSUMPTION

<table>
<thead>
<tr>
<th></th>
<th>Contiki</th>
<th>SQUAWK</th>
<th>OSGi</th>
</tr>
</thead>
<tbody>
<tr>
<td>LooCI</td>
<td>24,398 bytes</td>
<td>45,363 bytes</td>
<td>49,987 bytes</td>
</tr>
<tr>
<td>Underlying Platform</td>
<td>40,614 bytes</td>
<td>616,048 bytes</td>
<td>87,044 bytes</td>
</tr>
<tr>
<td>Total Flash Required</td>
<td>65,012 bytes</td>
<td>661,411 bytes</td>
<td>137,031 bytes</td>
</tr>
</tbody>
</table>

### DYNAMIC RUNTIME MEMORY CONSUMPTION

<table>
<thead>
<tr>
<th></th>
<th>Contiki</th>
<th>SQUAWK</th>
<th>OSGi</th>
</tr>
</thead>
<tbody>
<tr>
<td>LooCI</td>
<td>2,167 bytes</td>
<td>7,168 bytes</td>
<td>84,992 bytes</td>
</tr>
<tr>
<td>Underlying Platform</td>
<td>8,885 bytes</td>
<td>78,848 bytes</td>
<td>445,440 bytes</td>
</tr>
<tr>
<td>Total RAM Required</td>
<td>11,052 bytes</td>
<td>86,016 bytes</td>
<td>530,432 bytes</td>
</tr>
</tbody>
</table>

LooCI is small.
## Component Memory Evaluation

### STATIC COMPONENT MEMORY CONSUMPTION

<table>
<thead>
<tr>
<th>TempMonitor</th>
<th>Contiki</th>
<th>SQUAWK</th>
<th>OSGi</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Native Application</td>
<td>281 bytes</td>
<td>1,740 bytes</td>
<td>1,511 bytes</td>
</tr>
<tr>
<td>- LooCI Component</td>
<td>220 bytes</td>
<td>1,843 bytes</td>
<td>1,750 bytes</td>
</tr>
<tr>
<td>- Change using LooCI</td>
<td>- 61 bytes</td>
<td>+103 bytes</td>
<td>+ 239 bytes</td>
</tr>
</tbody>
</table>

### DYNAMIC COMPONENT MEMORY CONSUMPTION

<table>
<thead>
<tr>
<th>TempMonitor</th>
<th>Contiki</th>
<th>SQUAWK</th>
<th>OSGi</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Native Application</td>
<td>63</td>
<td>21,504 bytes</td>
<td>4,588 bytes</td>
</tr>
<tr>
<td>- LooCI Component</td>
<td>59</td>
<td>26,624 bytes</td>
<td>2,304 bytes</td>
</tr>
<tr>
<td>- Change using LooCI</td>
<td>- 4 bytes</td>
<td>+5,120 bytes</td>
<td>+ 2,284 bytes</td>
</tr>
</tbody>
</table>

Components are small.
LooCI Performance Evaluation

RUNTIME PERFORMANCE

<table>
<thead>
<tr>
<th></th>
<th>Contiki</th>
<th>SQUAWK</th>
<th>OSGi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime Initialization:</td>
<td>14 ms</td>
<td>498 ms</td>
<td>19 ms</td>
</tr>
<tr>
<td>Component Initialization:</td>
<td>0.26 ms</td>
<td>35 ms</td>
<td>1050 ms</td>
</tr>
<tr>
<td>Component Wiring:</td>
<td>0.15 ms</td>
<td>12 ms</td>
<td>0.12 ms</td>
</tr>
<tr>
<td>Component Unwiring:</td>
<td>0.15 ms</td>
<td>12 ms</td>
<td>0.12 ms</td>
</tr>
</tbody>
</table>

The event bus is fast.
LooCI Developer Effort

<table>
<thead>
<tr>
<th>DEVELOPER EFFORT</th>
<th>Contiki</th>
<th>SQUAWK</th>
<th>OSGi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface/receptacle declarations:</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Other component declaration code:</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Event publication:</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Total LooCI LoC:</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Total non-LooCI LoC:</td>
<td>14</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

Development is easy.
Future Work

• You will notice that we have not tackled implicit distribution dependencies.

• We have advocated against VMs (REMORA) and static composition (NesC) is not an option.

• We are working on embedding explicit platform dependencies:
  – Embed explicit descriptions of platform requirements in components.
  – Embed explicit descriptions of platform features in node run-tim.
Structure of This Talk

1. What is a component model?

2. Critique of contemporary component models

3. The Loosely-coupled Component Infrastructure

4. Conclusions
Conclusions

• We have seen that no component model fulfils the complete vision of CBSE.

• In NES problems that were glossed over by the DS community come to the fore:
  – Implicit distribution dependencies.
  – Implicit platform dependencies.

• We have demonstrated that implicit distribution dependencies can be removed and advocate for the removal of implicit platform dependencies.
Thanks To...