DYNAMIC COMPOSITION OF COMMUNICATION SERVICES

Carolina Fortuna

PhD Thesis Defense
March 29th 2013

“Jožef Stefan” International Postgraduate School
Department of Communication Systems, “Jožef Stefan” Institute
Outline

• Overview

• Contributions
  • Contribution 1: the framework
  • Contribution 2: the module library
  • Contribution 3: the declarative language
  • Contribution 4: the reference implementation

• Summary
Communication Service

• A set of primitives which provide communication functionality

• Use primitives to send requests and receive responses

• Their implementations can form protocols
Dynamic composition

- Can be done by having defined a set of standard, possibly well-described, interfaces

- Services can be composed at:
  - Compile time
  - Run-time

A set of services providing various networking functionality

Complex (composite) service
Purpose of the thesis

To propose, develop, implement and evaluate a framework for the dynamic composition of communication services that can:

• speed up design and experimentation with new protocol stacks
• support research on cognitive and service oriented networks
Major Scientific Contributions

1. The generic framework for dynamic composition of communication services
2. The module library for composeable communication services
3. The declarative language that enables service and interface description
4. The reference implementation of the framework

- Three original scientific articles:
- ~ 15 scientific conference contributions

<table>
<thead>
<tr>
<th>Publication</th>
<th>SCI IF</th>
<th>Citations in WOS/Scopus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elsevier Computer Networks</td>
<td>1.201</td>
<td>20/52</td>
</tr>
<tr>
<td>IEEE Wireless Communications Magazine</td>
<td>2.394</td>
<td>4/8</td>
</tr>
<tr>
<td>ACM Transactions on Sensor Networks</td>
<td>1.8</td>
<td>1, 2, 3, 4</td>
</tr>
</tbody>
</table>
Other Contributions

5. Logic based reasoning about communication services

6. Cognitive networking use-cases

7. Practical component re-use

8. Repeatable and remote experimentation capability
Contribution 1: the generic framework for the dynamic composition of communication services
The framework for dynamic composition of communication services

Physical testbed

is a set of machines on which the stack built by the composition of services is deployed and tested

Module library

the source code of the modules used for composing communication services

Declarative language

abstraction between the workbench and the program code

The workbench

is a panel for configuring, starting, running, retrieving and visualizing the results of an experiment.
Requirements (1/5)

• **Modularity**
  - the communication services have to have a modular design and implementation to allow composeability of more complex services which can then achieve end to end communication.
Requirements (1/5)

• **Modularity**
  
  the communication services have to have a modular design and implementation to allow composeability of more complex services which can then achieve end to end communication.

A set of services providing various networking functionality
Requirements (1/5)

- **Modularity**
  - the communication services have to have a modular design and implementation to allow composeability of more complex services which can then achieve end to end communication.
Requirements (1/5)

• **Modularity**
  - the communication services have to have a modular design and implementation to allow composeability of more complex services which can then achieve end to end communication.
Requirements (1/5)

- **Modularity**
  - the communication services have to have a modular design and implementation to allow composeability of more complex services which can then achieve end to end communication.
Requirements (1/5)

- **Modularity**
  - the communication services have to have a modular design and implementation to allow composeability of more complex services which can then achieve end to end communication.
Requirements (1/5)

- **Modularity**
  - the communication services have to have a modular design and implementation to allow composeability of more complex services which can then achieve end to end communication.
Requirements (1/5)

- **Modularity**
  - the communication services have to have a modular design and implementation to allow composeability of more complex services which can then achieve end to end communication.
Requirements (1/5)

• **Modularity**
  
  the communication services have to have a modular design and implementation to allow composeability of more complex services which can then achieve end to end communication.

![Diagram of communication services]

Complex (composite) service
Requirements (1/5)

- **Modularity**
  - the communication services have to have a modular design and implementation to allow composeability of more complex services which can then achieve end to end communication.
Requirements (1/5)

- **Modularity**
  - the communication services have to have a modular design and implementation to allow composeability of more complex services which can then achieve end to end communication.
Requirements (1/5)

• **Modularity**
  • the communication services have to have a modular design and implementation to allow composeability of more complex services which can then achieve end to end communication.
Requirements (1/5)

- **Modularity**
  - the communication services have to have a modular design and implementation to allow composeability of more complex services which can then achieve end to end communication.

---

Complex (composite) service
Requirements (1/5)

- **Modularity**
  - the communication services have to have a modular design and implementation to allow composeability of more complex services which can then achieve end to end communication.
Requirements (1/5)

• **Modularity**
  • the communication services have to have a modular design and implementation to allow composeability of more complex services which can then achieve end to end communication.
Requirements (1/5)

- **Modularity**

  - the communication services have to have a modular design and implementation to allow composeability of more complex services which can then achieve end to end communication.
Requirements (2/5)

- **Flexibility**
  - the components of the framework should be
    - designed and implemented in a way that allows interacting with the resulting tool at different levels of abstractions (e.g. at the module library level, at the workbench level).
  - easy to extend and upgrade.
Requirements (2/5)

- **Flexibility**
  - the components of the framework should be
    - designed and implemented in a way that allows interacting with the resulting tool at different levels of abstractions (e.g. at the module library level, at the workbench level).
  - easy to extend and upgrade.
Requirements (2/5)

- **Flexibility**
  - the components of the framework should be
    - designed and implemented in a way that allows interacting with the resulting tool at different levels of abstractions (e.g. at the module library level, at the workbench level).
  - easy to extend and upgrade.
Requirements (3/5)

- **Easy programming**
  - users with various levels of programming skills should find it easy to use the tools appropriate to their level of experience resulting from the implementation of the framework.

```
Researcher
Skills: embedded software

Researcher
Skills: scripting languages

Researcher
Skills: GUI based configuration
```

```
Physical testbed  Module library  Declarative language  The workbench
```
Requirements (4/5)

• **Reproducibility of experiments**
  • the framework should support re-running and reproducing experiments in an easy way for instance by saving and reloading an experiment description.
Requirements (4/5)

- Reproducibility of experiments
  - the framework should support re-running and reproducing experiments in an easy way for instance by saving and reloading an experiment description.
Requirements (4/5)

• **Reproducibility of experiments**
  • the framework should support re-running and reproducing experiments in an easy way for instance by saving and reloading an experiment description.

Design and configure experiment
Requirements (4/5)

• **Reproducibility of experiments**
  - the framework should support re-running and reproducing experiments in an easy way for instance by saving and reloading an experiment description.
Requirements (4/5)

- **Reproducibility of experiments**
  - the framework should support re-running and reproducing experiments in an easy way for instance by saving and reloading an experiment description.
Requirements (4/5)

- **Reproducibility of experiments**
  - the framework should support re-running and reproducing experiments in an easy way for instance by saving and reloading an experiment description.
Requirements (4/5)

- **Reproducibility of experiments**
  - The framework should support re-running and reproducing experiments in an easy way for instance by saving and reloading an experiment description.
Requirements (5/5)

• Remote experimentation
  • remote users should be able to define and perform experiments and download the result (can be most easily achieved through a web portal).
Requirements (5/5)

• **Remote experimentation**
  • remote users should be able to define and perform experiments and download the result (can be most easily achieved through a web portal).
Requirements (5/5)

- **Remote experimentation**
  - remote users should be able to define and perform experiments and download the result (can be most easily achieved through a web portal).
Requirements (5/5)

- Remote experimentation
  - remote users should be able to define and perform experiments and download the result (can be most easily achieved through a web portal).
Contribution 2: the module library for composeable communication services
The Composable Rime module library

• Inspired by the Rime communication architecture available in the Contiki operating system

• There are three major differences with respect to Rime:
  • CRime introduces three abstractions that enable modularity
  • CRime further generalizes existing Rime communication functionality
  • CRime is designed to encourage code re-use
CRime abstraction 1

- The *abstract module*
  - is a generic building block of the CRime stack
  - each instance of an amodule hides a communication service
CRime abstraction 1

- The *abstract module*
  - is a generic building block of the CRime stack
  - each instance of an amodule hides a communication service
CRime abstraction 1

- The *abstract module*
  - is a generic building block of the CRime stack
  - each instance of an amodule hides a communication service
CRime abstraction 1

- The *abstract module*
  - is a generic building block of the CRime stack
  - each instance of an amodule hides a communication service
CRime abstraction 2

• The *pipe*
  • is a vertical structure which can be accessed by any of the modules in a composed stack.
  • the pipe contains only data structures corresponding to parameters that are used by the stack.
  • pipes are uniquely identified by the channel number they are assigned to
The pipe

- is a vertical structure which can be accessed by any of the modules in a composed stack.
- the pipe contains only data structures corresponding to parameters that are used by the stack.
- pipes are uniquely identified by the channel number they are assigned to
CRime abstraction 2

- The *pipe*
  - is a vertical structure which can be accessed by any of the modules in a composed stack.
  - the pipe contains only data structures corresponding to parameters that are used by the stack.
  - pipes are uniquely identified by the channel number they are assigned to
CRime abstraction 3

• The *stack*
  • is a structure which contains a meaningful sequence of modules and a pipe.
  • it behaves as a container for these elements and enables the composition of more complex communication services which use more than a single channel at a time.
CRime abstraction 3

• The *stack*
  
  • is a structure which contains a meaningful sequence of modules and a pipe.
  
  • it behaves as a container for these elements and enables the composition of more complex communication services which use more than a single channel at a time.
CRime abstraction 3

- The *stack*
  - is a structure which contains a meaningful sequence of amodules and a pipe.
  - it behaves as a container for these elements and enables the composition of more complex communication services which use more than a single channel at a time.

![Diagram of CRime abstraction 3]
Rime vs. CRime

Diagram showing the dynamic composition of communication services with nodes like 'unicast', 'broadcast', 'c_polite', 'c_unicast', 'c_broadcast', 'c_abc', 'ipolite', and 'abc' connected by arrows to represent different service compositions.
Rime vs. CRime

- To achieve the same basic functionality, CRime needs 4 service implementations while Rime needs 6.
Rime vs. CRime

- To achieve the same basic functionality, CRime needs 4 service implementations while Rime needs 6.
- With CRime, 11 complex services are possible, while Rime enables only 6.
The cost of composeability

- Composeability introduces additional overhead (the implementation of the abstractions)
  - the Rime and CRime components differ just in the size of the code with no clear advantage on one or the other side

- the size of the code of the applications which use CRime stacks is about 16% larger (~13,000 bytes)

- execution time for the sequence of operations open→send→recv→close is ~256 ms in Rime and 622 ms in CRime (a factor of ~2.4 higher)

- CRime consumes 1.6% more energy than Rime (can run 360 days vs 365 with Rime)
The cost of composeability

- Composeability introduces additional overhead (the implementation of the abstractions)
  - the code with the Rime and CRime components differ just in the size of the code blocks is about 16% larger (~13,000 bytes)
  - the execution time for the sequence of operations open→send→recv→close is ~256 ms in Rime and 622 ms in CRime (a factor of ~2.4 higher)
  - CRime consumes 1.6% more energy than Rime (can run 360 days vs 365 with Rime)

... but it brings freedom to:

- easily compose and test new service combinations
- instantiates through the pipe abstraction the concept of vertical layer required for experimentation with cognitive networks
Contribution 3: the declarative language that enables service and interface description
The declarative language

- is a natural intermediate level of abstraction between a user (interface) and the program code

- it can be used for
  - describing what a program should do,
  - how it should do it,
  - to perform validity checking,
  - to provide interoperability with other systems
  - to support particular types of research
Requirements

- **Simplicity**
  - as friendly as possible to the target user group

- **Machine readable**
  - to facilitate easy manipulation by machines

- **Standardized**
  - a relatively widely adopted, open and stable standardized approach is preferred to a less stable and potentially proprietary approach

- **Interoperability**
  - to facilitate the interoperability of systems so that potential reference implementations of the framework can be easily connected at this level of abstraction

- **Support for knowledge representation and logic reasoning**
  - should also support emerging logical reasoning for self-configuration of communication networks
The ProtoStack declarative language

- uses a Subject-Predicate-Object model widely popular in the semantic web area
- uses the Resource Description Framework (RDF) standard

<table>
<thead>
<tr>
<th>Subject (Resource)</th>
<th>Predicate (Property)</th>
<th>Object (Statement)</th>
</tr>
</thead>
<tbody>
<tr>
<td>crime:c_abc</td>
<td>rdf:type</td>
<td>cpan:Module</td>
</tr>
<tr>
<td>crime:c_open</td>
<td>rdfs:subClassOf</td>
<td>crime:Function</td>
</tr>
</tbody>
</table>
The ProtoStack declarative language

- uses a Subject-Predicate-Object model widely popular in the semantic web area
- uses the Resource Description Framework (RDF) standard

<table>
<thead>
<tr>
<th>Subject (Resource)</th>
<th>Predicate (Property)</th>
<th>Object (Statement)</th>
</tr>
</thead>
<tbody>
<tr>
<td>crime:c_abc</td>
<td>rdf:type</td>
<td>cpan:Module</td>
</tr>
<tr>
<td>crime:c_open</td>
<td>rdfs:subClassOf</td>
<td>crime:Function</td>
</tr>
</tbody>
</table>

- created the CRime ontology reflecting the CRime “world” model
The ProtoStack declarative language

- uses a Subject-Predicate-Object model widely popular in the semantic web area
- uses the Resource Description Framework (RDF) standard

<table>
<thead>
<tr>
<th>Subject (Resource)</th>
<th>Predicate (Property)</th>
<th>Object (Statement)</th>
</tr>
</thead>
<tbody>
<tr>
<td>crime:c_abc</td>
<td>rdf:type</td>
<td>cpan:Module</td>
</tr>
<tr>
<td>crime:c_open</td>
<td>rdfs:subClassOf</td>
<td>crime:Function</td>
</tr>
</tbody>
</table>

- created the CRime ontology reflecting the CRime “world” model

Satisfies requirements 1-4:
- simplicity
- machine readability
- standards based
- interoperability
Support for Reasoning

- The resulting declarative language can be used for logic based reasoning by:
  - Selecting a logic reasoner that supports rules (there are several such tools available in the semantic web community: Jena, OWLIM)

\[
\text{rule1: } (?a \text{ rdf:type } ?b) (?b \text{ rdf:type } ?c) \rightarrow (?a \text{ rdf:type } ?c)
\]

- The CRime ontology can be mapped to more complex ontologies for which powerful reasoners are available

<table>
<thead>
<tr>
<th>CRime Concepts</th>
<th>Cyc Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>ComputerProgramModule-CW</td>
</tr>
<tr>
<td>Interface</td>
<td>InterfaceProgram</td>
</tr>
<tr>
<td>Function</td>
<td>CodingFunction</td>
</tr>
<tr>
<td>Parameter</td>
<td>CodingFunctionParameter</td>
</tr>
<tr>
<td>Scope</td>
<td>/ (created Scope-Module)</td>
</tr>
</tbody>
</table>
Support for Reasoning

• the resulting declarative language can be used for logic based reasoning by:
  • Selecting a logic reasoner that supports rules (there are several such tools available in the semantic web community: Jena, OWLIM)

\[\text{[rule1: } (?a \text{ rdf:type } ?b) (\text{?b rdf:type } ?c) \rightarrow (?a \text{ rdf:type } ?c)\]

• the CRime ontology can be mapped to more complex ontologies for which powerful reasoners are available

<table>
<thead>
<tr>
<th>CRime Concepts</th>
<th>Cyc Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>ComputerProgramModule-CW</td>
</tr>
<tr>
<td>Interface</td>
<td>InterfaceProgram</td>
</tr>
<tr>
<td>Function</td>
<td>CodingFunction</td>
</tr>
<tr>
<td>Parameter</td>
<td>CodingFunctionParameter</td>
</tr>
<tr>
<td>Scope</td>
<td>/ (created Scope-Module)</td>
</tr>
</tbody>
</table>
Contribution 4: the reference implementation of the framework
ProtoStack – implementation of the framework

VESNA sensor network platform

The CRime module library

The ProtoStack declarative language

The workbench is tightly integrated with the language and it is based on standard web technologies.
ProtoStack’s workflow

1. Module library
   - C-RIME
2. Declarative language
   - C-RIME ontology
   - Turtle
3. The workbench
   - WireIt
4. 4
5. 5
ProtoStack’s workflow

1. Parse the Turtle statements describing CRime module and populate the knowledge base.
2. The workbench
3. WireIt
4. Declarative language
5. Module library
   C-RIME
   .h, .c
6. Physical testbed
   VESNA
ProtoStack’s workflow

1. Parse the Turtle statements describing CRime module and populate the knowledge base.
2. Automatically populate the workbench based on the existing knowledge.

Physical testbed
- VESNA

Module library
- C-RIME

Declarative language
- C-RIME ontology
- Turtle

The workbench
- WireIt
ProtoStack's workflow:
Dynamic Composition of Communication Services

Parse the Turtle statements describing the CRime module and populate the knowledge base.

Automatically populate the workbench based on the existing knowledge.
ProtoStack’s workflow

1. Parse the Turtle statements describing CRime module and populate the knowledge base.
2. Automatically populate the workbench based on the existing knowledge.
3. The workbench
4. Declarative language
   - C-RIME ontology
   - Turtle
5. Module library
   - .h, .c
   - C-RIME
6. Physical testbed
   - VESNA

Perform validity checking on the sequence of composed services.
ProtoStack’s workflow

1. Parse the Turtle statements describing CRime module and populate the knowledge base.
2. Automatically populate the workbench based on the existing knowledge.
3. C-RIME ontology
4. Turtle
5. VESNA

Auto-generate the C code.

Perform validity checking on the sequence of composed services.
ProtoStack’s workflow

1. Parse the Turtle statements describing CRime module and populate the knowledge base.
2. Automatically populate the workbench based on the existing knowledge.
3. Perform validity checking on the sequence of composed services.
4. Auto-generate the C code.
5. Deploy on the testbed.

Physical testbed
VESNA

Module library
C-RIME

Declarative language
C-RIME ontology
Turtle

The workbench
WireIt
Use Cases enabled by ProtoStack

- Enable the description, publishing and querying of services for service oriented networks
- Guide the user in composing complex services
- Enable cognitive networking prototypes of various complexities:
  - cognitive loop with cross-layer information
  - cognitive loop with cognitive radio
  - cognitive loop with radio environmental maps
  - cognitive loop with connectivity broker
Summary

• Four major contributions
  1. the generic framework for dynamic composition of communication services
  2. the module library for composeable communication services
  3. the declarative language that enables service and interface description
  4. the reference implementation of the framework

• Resulting in 3 journal and several conference papers
Future Work

• Extend the CRime module library with learning modules

• Add over the air programming functionality and deploy on testbeds such as LOG-a-TEC

• Investigating the trade-offs in offering distributed versus centralized brokerage functionality in cognitive networks

• Increase the degree of automation for complex service composition using logic reasoning
Thank you

• Questions?