

DYNAMIC COMPOSITION OF COMMUNICATION SERVICES

Carolina Fortuna

Doctoral Dissertation
Jožef Stefan International Postgraduate School
Ljubljana, Slovenia, October 2012

Evaluation Board:

Asst. Prof. Dr. Aleš Švigelj, Chairman

Jožef Stefan Institute Jamova 39, SI-1000 Ljubljana, Slovenia and
Jožef Stefan International Postgraduate School, Jamova 39, SI-1000 Ljubljana,
Slovenia.

Prof. Dr. Luiz A. DaSilva, Member

Trinity College Dublin, Dunlop Oriel House, Trinity College Dublin, Dublin 2, Ireland
and

Virginia Polytechnic Institute and State University, 4300 Wilson Blvd., Suite 750,
Arlington, VA 22203, USA.

Dr. Michael Witbrock, Member

Cycorp, Inc., 7718 Wood Hollow Drive, Suite 250, Austin, TX 78731-1601, USA.

MEDNARODNA PODIPLOMSKA ŠOLA JOŽEFA STEFANA
JOŽEF STEFAN INTERNATIONAL POSTGRADUATE SCHOOL



Carolina Fortuna

DYNAMIC COMPOSITION OF COMMUNICATION SERVICES

Doctoral Dissertation

DINAMIČNO SESTAVLJANJE KOMUNIKACIJSKIH STORITEV

Doktorska disertacija

Supervisor: Asst. Prof. Dr. Mihael Mohorčič
Co-Supervisor: Prof. Dr. Dunja Mladenić

Ljubljana, Slovenia, October 2012

Index

1 Introduction.....	1
1.1 Motivation	1
1.1.1 Why is the composition of communication services relevant?	1
1.1.2 Why is experimenting in realistic environments difficult?	2
1.1.3 What kind of research can the composition of communication services support?	3
1.2 Related work	3
1.2.1 Related experimentation tools	4
1.2.2 Related approaches to modular stacks	5
1.3 Contributions	6
1.4 Organization of the Dissertation	7
2 Layered modularity	9
2.1 Cross layer design	10
2.1.1 Implicit cross-layer design	10
2.1.2 Explicit cross-layer design	10
2.2 Service oriented networks	12
2.3 Cognitive networks	12
2.4 Summary	13
3 A framework for dynamic composition of services	15
3.1 The components of the proposed framework.....	15
3.1.1 The physical testbed	15
3.1.2 The module library	15
3.1.3 The declarative language	16
3.1.4 The workbench	16
3.2 Requirements.....	16
3.3 Reference implementation: the ProtoStack tool.....	17
3.4 Discussion	19
3.4.1 On the proposed framework	19
3.4.1 On the ProtoStack reference implementation.....	20
3.5 Summary	22
4 The Composeable Rime module library	23
4.1 CRime abstractions	23
4.1.1 CRime packets.....	24
4.1.2 Theoretical modeling of the CRime communication stack	24
4.1.3 CRime amodule	25
4.1.1 CRime pipe.....	25
4.1.2 CRime stack.....	26

4.2 Architectural comparison with Rime	27
4.2.1 Communication primitives	27
4.2.2 Support for modular stacks	30
4.2.3 Support for cross-layering	32
4.2.4 Support for cognitive networking	32
4.2.5 Portability	32
4.3 The cost of composeability	32
4.3.1 Experimental setup	33
4.3.2 Image size	33
4.3.3 Processing speed	36
4.3.4 Power consumption	37
4.4 Summary	38
5 The ProtoStack declarative language and the Workbench	39
5.1 The ProtoStack declarative language	39
5.1.1 Requirements for the declarative language	39
5.1.2 Design decisions related to the declarative language	40
5.2 The CRime ontology	41
5.3 ProtoStack configuration steps	42
5.3.1 Fact specification, translation and storage	42
5.3.2 Workbench rendering	44
5.3.1 Manual stack composition	45
5.3.1.1 Example of 1 channel – 1 stack configuration	47
5.3.1.2 Example of n channel – n stack configuration	47
5.3.2 Validity checking and code generation	48
5.4 Reusability	50
5.5 Reproducibility of the experiment	51
5.6 Support for remote experimentation	51
5.7 On the usability of ProtoStack	51
5.7.1 Methodology	52
5.7.2 The collected feedback	53
5.8 Summary	55
6 Dynamic composition of services for Service Oriented Networks	57
6.1 SOA and OSI	57
6.2 Service Oriented Networks	59
6.2.1 Service and interface descriptions	61
6.2.2 Querying for published services	61
6.2.3 Composition of services for information transport	62
6.3 Service oriented networks with ProtoStack	62
6.3.1 Describing and publishing ProtoStack services	62
6.3.1.1 Discussion	63
6.3.2 Querying for published ProtoStack services	64
6.3.2.1 Synchronizing distributed ProtoStack systems	64
6.3.2.2 Discussion	65
6.3.3 Composition of services for information transport using ProtoStack	66
6.3.3.1 Manual composition	66
6.3.3.2 Semi-automatic composition	66
6.3.3.3 Automatic composition	68

6.4 Summary	69
7 Dynamic composition of services for Cognitive Networks.....	71
7.1 Cognitive networks	71
7.1.1 Knowledge representation	72
7.1.1.1 Knowledge representation for management	73
7.1.1.2 Knowledge representation for quality of service	74
7.1.1.3 Knowledge representation for security	75
7.1.1.4 Discussion	75
7.1.2 Cognition loop	79
7.1.2.1 Loops for management	80
7.1.2.2 Loops for overlays	82
7.1.2.3 Loops for security	82
7.1.2.4 Discussion	82
7.2 Knowledge representation in ProtoStack	85
7.2.1 Smart grid use case	86
7.2.2 Cognitive ad hoc network use case	87
7.2.3 Logic based reasoning for CNs	88
7.3 Cognitive loops with ProtoStack	89
7.3.1 Network layer cognitive loop with cross-layer information	89
7.3.2 Network layer cognitive loop with cognitive radio	91
7.3.3 Network layer cognitive loop with radio environment map	93
7.3.4 Network layer cognitive loop with connectivity broker	94
7.4 Summary	95
8 Conclusions	97
8.1 Future Work	98
9 Acknowledgements	99
10 References	101
Index of Figures	109
Index of Tables	111
Index of Algorithms	113
Appendix A: Rime Communication Primitives	115
Appendix B: Code corresponding to the Rime <unicast, broadcast, abc> stack.....	123
Appendix C: Code corresponding to the CRime <c_unicast, c_broadcast, c_abc> stack.....	127
Appendix D: The CRime ontology	129
Appendix E: Turtle statements describing the c_abc module	135
Appendix F: Feedback collection: questionnaire and results.....	137
Appendix G: Publications Related to this Thesis	147
Appendix H: Biography	151

Abstract

In this thesis, we look at networking functionality as a set of services which can be composed dynamically. We argue that the dynamic composition of communication services can speed up design and experimentation with new protocol stacks. The reference Open Systems Interconnect (OSI) architecture of communication networks splits network functionality into layers, each providing a service to the layer above. The modularity provided by the OSI layered abstraction is further split in our approach by looking also at layers as a set of services.

In order to substantiate our claim, we propose a conceptual framework for quick and efficient prototyping and deployment of modular protocol stacks as a composition of communication services, and its reference implementation ProtoStack. The proposed framework consists of four components: the physical testbed, the module library, the declarative language and the workbench. The physical testbeds that ProtoStack, as the reference implementation, can support have to be based on hardware platforms which can host Contiki operating system. The module library used by ProtoStack is called Composeable Rime (CRime). The declarative language is based on the Resource Description Framework (RDF) and uses a custom built vocabulary. The workbench is a web based portal which allows easy, graphically supported configuration of a stack from the available CRime modules.

We show, through feedback collection from first time users, that the ProtoStack tool can significantly speed up prototyping and testing of new stacks and is friendly to novice and advanced users. The initial feedback shows that the tool can speed up design and prototyping of new protocol stack by at least a factor of 2. The cost of increased flexibility and prototyping speed of the protocol stack is paid in terms of increased memory footprint, processing speed and energy consumption. The CRime library used by ProtoStack has a 16% larger footprint, it takes 2.4 times longer to execute an open->send->recv->close sequence and consumes 1.6% more power in doing so. Even though with ProtoStack more resources are consumed by the node, the tradeoff in terms of prototyping speed seems to pay off.

The last part of the thesis discusses service oriented networks (SON) and cognitive networks (CN) and how the dynamic composition of communication services can be used to further research in this area. Particularly, we show on selected use cases how ProtoStack can be used to design and experiment with SONs and CN. The thesis ends with conclusions and a brief discussion identifying future work.

Povzetek

V doktorski disertaciji gledamo na funkcionalnost omrežja kot na skupek osnovnih komunikacijskih storitev, ki jih lahko dinamično sestavimo glede na podane zahteve. Z dinamičnim sestavljanjem komunikacijskih storitev lahko pospešimo načrtovanje in testiranje novih protokolnih skladov. Že referenčna arhitektura komunikacijskih omrežij OSI deli funkcionalnost omrežja v sloje, kjer nižje ležeči sloj zagotavlja storitve neposredno višje ležečemu sloju. To modularnost arhitekture OSI v našem pristopu nadalje povečamo tako, da tudi na posamezne sloje gledamo kot na skupek storitev.

Da bi dokazali svojo trditev, predlagamo konceptualni okvir za hitro in učinkovito izdelavo prototipov in uvajanje modularnih protokolnih skladov kot sestavljenih komunikacijskih storitev. V disertaciji predstavimo referenčno izvedbo konceptualnega okvirja imenovano ProtoStack. Predlagani konceptualni okvir sestavljajo fizična postavitev eksperimentalnih naprav, knjižnica modulov, deklarativni jezik in namizje. ProtoStack v okviru fizične postavitve eksperimentalnih naprav podpira strojne platforme z operacijskim sistemom Contiki. Knjižnica modulov, ki jo uporablja ProtoStack, se imenuje Composeable Rime (CRime; slovensko sestavljivi Rime) in je zasnovana po zgledu protokolnega sklada Rime, ki je implementiran v operacijskem sistemu Contiki. Deklarativni jezik temelji na podatkovnem modelu RDF (Resource Description Framework) in uporablja namensko razvit besednjak. Namizje oziroma uporabniški vmesnik orodja ProtoStack je izvedeno kot spletni portal, ki omogoča enostavno in grafično podprto konfiguracijo protokolnega sklada iz razpoložljivih modulov CRime.

Na podlagi odzivov prvih uporabnikov pokažemo, da lahko orodje ProtoStack znatno pospeši izdelavo prototipov in testiranje novih protokolnih skladov ter je prijazno za uporabo tako za začetnike kot zahtevne uporabnike. Odzivi kažejo, da lahko orodje skrajša čas zasnove in prototipiranja novih protokolnih skladov vsaj za polovico. Ceno za večjo prožnost in hitrost izdelave prototipov novih protokolnih skladov predstavljajo povečane potrebe po pomnilniškem prostoru in računskem času ter večja poraba energije. V primerjavi z osnovnim protokolnim skladom Rime ima knjižnica modulov CRime 16% večje potrebe po pomnilniku, 2,4 krat daljši čas open->send->recv->close in 1,6% večjo porabo energije.

V zadnjem delu disertacije pokažemo, da lahko s pomočjo orodja ProtoStack in dinamičnega sestavljanja komunikacijskih storitev načrtujemo in izvajamo eksperimente na raziskovalnih področjih storitveno orientiranih omrežij in kognitivnih omrežij. Doktorsko disertacijo zaključujemo z diskusijo in predlogi za nadaljnje raziskovalno delo.

Abbreviations

SCN	=	Software (re)Configurable Network
SDN	=	Software Defined Network
SDR	=	Software Defined Radio
CN	=	Cognitive Network
CRN	=	Cognitive Radio Network
CR	=	Cognitive Radio
GUI	=	Graphical User Interface
JSON	=	JavaScript Object Notation
RDF	=	Resource Description Framework
Turtle	=	Terse RDF Triple Language
OSI	=	Open Systems Interconnect
SOA	=	Service Oriented Architecture
SON	=	Service Oriented Network
SNA	=	Sensor Network Architecture

1 Introduction

1.1 Motivation

In machine communication networks, a service is a set of primitives which provide communication functionality. Services use primitives to send requests and receive responses. Protocols are implementations of services and a set of protocols can form a communication stack. Traditionally, services and protocols are fixed and defined by standards which prevent adaptation of networks to changing conditions. In this thesis, we investigate the dynamic composition of communication services using sensor networks as an example to prototype and demonstrate the proposed solutions. Nowadays, sensor nodes can be programmed to use an HTTP/TCP/IP-like communication stack, thus forming affordable yet representative communication networks for testing. We provide a framework for experimentation with dynamic composition of services, a reference implementation and a thorough evaluation.

1.1.1 Why is the composition of communication services relevant?

The composition of communication services is relevant because it leverages fast prototyping and testing of technology, therefore speeding up the development cycle. The development cycle of new protocols and algorithms typically includes the following steps: design, simulation, emulation, testing in a realistic setup, and deployment as depicted in Figure 1.

Figure 1 The development cycle of new protocols and algorithms (adapted from [1])

The design of new protocols is typically done using a whiteboard, planning tools and/or programming languages such as C. After the design, the protocol is typically evaluated through simulation and emulation. For these steps, tools such as Opnet¹, NS2², Emulab³ and virtual machines exist. For the *testing in a realistic setup* step, besides

¹ <http://www.opnet.com/>

² <http://www.isi.edu/nsnam/ns/>

³ <http://www.emulab.net/>

appropriate software tools, hardware infrastructure is also required, therefore increasing the cost of performing this step. As such, many new protocols never go beyond the simulation stage. In order to overcome this limitation, and help speed up the development of new protocols, networks that can be configured in software such as the x-kernel [2], Click [3] and IRIS [4] as well as software defined networks (SDN) such as OpenFlow [5] have been proposed.

The software (re)configurable networks (SCNs) abstract network functionality that can be found at various layers of a communication stack. Modular implementations of the identified abstractions are then provided. Using this approach, the communication stack can be easily (re)configured at compile or run time. The SCN approach based on functionality centric abstractions and implementation is thus in its essence different than the traditional approach where the abstractions as well as implementations were protocol centric.

SDNs provide a network-centric abstraction of hardware resources analogous to the way virtual machines provide an application-centric abstraction of hardware resources. SDNs are programmatically controlled networks with their state managed by logically centralized control programs with a global network view. The control programs interact with the data path implemented in hardware through a standard API [6]. A network that is now defined in software corresponds to previous physical communication network that required unstalling and configuring hardware units.

SCN and SDN implementations are often complementary, where, for instance, a router configured with Click elements can be deployed on a network defined using OpenFlow.

With the abstractions provided by SCNs and SDNs, designing, emulating and testing new protocols is significantly easier than with the traditional approach. The flexibility offered by the dynamic composition of communication services can help:

- the extension and adaptation of existing protocols and stacks
- the composition and configuration of the protocol (stack)
- the deployment of the software in a testbed

1.1.2 Why is experimenting in realistic environments difficult?

There are several possible answers to this question, depending on the desired scale of the realistic environment. Experimenting in very large networks such as the network of a service provider also which includes datacenters is very risky and may be extremely costly should anything go differently than planned. Solutions for very large scale experimentation of Internet protocols propose creating a virtual slice in the production network which is then to be used for research purposes [1][5]. Experimenting in large networks such as campuses or federations of academic networks is relatively less costly but also challenging. Solutions, such as VINI [7], and initiatives, such as FIRE [8] in Europe and GENI [9] in the US, aim to provide alternatives. Experimenting in relatively

small but real testbeds is a challenge *per se* in several communities which would need custom facilities to investigate new technologies that are too young for production or campus networks. In this case, specific testbeds are typically set up and used. Examples include ORBIT [10], w-iLab.t [11], TWIST [12] and several others in FIRE and GENI initiatives. However, building and keeping these facilities evergreen is very challenging and costly at least for the following reasons:

- need for dedicated hardware
- need for a specific management platform
- need for maintenance
- need for upgrading

Recent initiatives, such as Flexible Internetwork Stack (FINS) [13], attempt to ease experimenting in realistic environments by providing researchers with monitoring, logging, and reconfiguration utilities similar to the ones provided by simulation environments or emulation testbeds.

1.1.3 What kind of research can the composition of communication services support?

The composition of communication services in realistic environments is particularly suited for less conventional research such as clean slate design [14], cognitive radio [15] and cognitive networks [16]. With respect to the reference layered architecture of the networks, we can identify five approaches of the research community to future development [17]:

- incremental improvement of existing protocols and algorithms,
- creation of new protocols according to the conventional practice involving the rigid layered architecture,
- creation of new protocols by violating the layered architecture – also referred to as cross-layer design
- creation of new protocols and algorithms augmented by techniques from the artificial intelligence domain – also referred to as cognitive networks
- investigating fundamental architectural changes by creating new abstractions – also referred to as clean slate design

1.2 Related work

Previous work related to this thesis can be grouped in two main clusters. One cluster comprises tools which support experimentation with composable stacks. Here we look at their abstractions, the composition model, the approach to configuration and the implementation. The second cluster comprises existing modular stacks where we pay attention to their scope with respect to the layers of the OSI stack.

1.2.1 Related experimentation tools

The idea of composing communication services has already been investigated, albeit using different naming. The most representative examples are x-Kernel and Click. X-Kernel [2] is an operating system kernel architecture explicitly designed for constructing and composing network protocols. The x-Kernel defines three abstractions used to implement protocols: *protocol*, *session* and *message*. The protocol corresponds to conventional network protocol such as TCP or IP, a session is an instance of a protocol and messages are objects that move through session and protocol objects. Protocol stacks are modelled as graphs which are configured either using a textual graph description language, or a GUI. Based on the input, C code for protocol instantiation and configuration is generated. X-Kernel has been used for experimenting with the decomposition of large protocols into primitive building block pieces, as a framework for designing and evaluating new protocols, and as a platform for accessing heterogeneous collections of network services.

The Click modular router [3] is a software architecture meant for building flexible and configurable routers. Click was inspired by the x-Kernel, however, it focuses on routers and uses different abstractions. Click uses *elements* as abstractions for packet processing units. Elements are then composed in a directed graph structure using a declarative language for describing the graph or, more recently, the Clicky GUI. The resulting file is then processed by a sequence of tools performing configuration in several stages. Click is widely used for researching networking related problems including composition of functionality.

In Table 1 we provide a feature base comparison of the most relevant experimentation tools. It can be seen that both x-Kernel and Click provide abstractions that support modularity and support user friendly configuration using a GUI. They use a declarative language as an intermediate abstraction between the GUI and the implementation of the modules of the stack. They use a graph based composition model.

Table 1: Comparison of related experimentation tools.

Project	Abstraction	Composition model	Configuration			Implementation language
x-Kernel	protocol, session, message	Graph	Textual description language, editor	graph graph	Auto-generated C code	C (minimal OO style)
Click	element	Directed graph	Click for configuration (declarative), Clicky GUI	language	Auto-generated C++ code	C++

1.2.2 Related approaches to modular stacks

Modular stacks existing to date mainly differ in their scope relative to the OSI layers. Besides the x-Kernel and Click, we also refer to the sensor network architecture (SNA), an effort to modularize protocols developed for TinyOS, Rime and the recursive network architecture (RNA).

The SNA project [18], aims at developing a modular network layer (MNL) for sensor networks by considering existing network layer protocols, finding common functionality and reimplementing this functionality in a protocol independent manner. In this way, the same code can be reused by several protocols and several protocols are able to exist at runtime without prohibitive memory usage. Together with the work in [19], the aim was to build a modular architecture for sensor networks. Then, in order to further eliminate constraints imposed by layered architectures, in [20], a declarative sensor network (DSN) architecture is proposed. This architecture uses a high level declarative specification language and a compiler to permit the user to specify an application. The main problem solved by this work is the provision of an easy way to programming sensor networks.

The Rime protocol stack [21] provides a set of communication primitives for sensor networks which are arranged in a layered fashion, where the more complex communication services are implemented using the less complex ones. Even though the conceptual model behind Rime is modular, the implementation itself is not.

Other relevant approaches are provided by the RNA and ANA architectures. The RNA [22] architecture aims to develop recursive protocol stacks. Their abstraction is the *meta-protocol* and the current release uses the elements of the Click router. The ANA project [23] defined and provided a reference implementation of yet another abstraction for modular stacks. This approach is more complex and attempts to take a clean slate approach towards building protocol stacks.

Table 2 summarizes related work with respect to modular stacks. While solutions such as the x-Kernel and Click provide means to experiment with modular stacks, they target conventional networks. Furthermore, x-Kernel looks at ways to design OS kernels that support modular stacks while Click only addresses modular router reconfigurations. Both solutions provide declarative means for configuration. RNA proposes recursive network architecture using modular stacks and is based on Click. The Sensor Network Architecture project where SP, MNL and SP were developed is an effort to transform existing protocols into composition of modular functional blocks, at the same time, introduce a declarative language which would enable easier prototyping. Rime is a stack built based on modular principles with a less flexible implementation.

Table 2: Comparison of related approaches to modular stacks.

Project	Scope	Configuration
x-Kernel	All layers	Textual graph description language, graph editor
Click	Network	Click language for configuration (declarative), Clicky GUI
SNA (SP, MNL, DSN)	Network and below	Declarative
Rime	All layers	-
RNA	Meta-protocol	Click

1.3 Contributions

In this thesis, we claim the following four major contributions:

- the definition of a generic framework for dynamic composition of communication services
- a reference implementation of the framework for dynamic composition of communication services as a proof of concept. We named the reference implementation ProtoStack.
- a composeable stack for sensor systems called Composeable Rime (CRime)
- the declarative language which supports configuration, validity checking, publishing modules, finding modules and logical reasoning

The first major contribution consists of the definition of a generic framework for the dynamic composition of communication services. This framework comprises four functional components: the physical testbed, the module library, the declarative language and the workbench. The second contribution consists of a reference implementation of this framework as a proof of concept. The reference implementation is called ProtoStack and targets sensor systems. We also discuss how existing tools, namely x-Kernel and Click, fit in this framework. Two of the components of ProtoStack, the reference implementation, constitute major contributions themselves. The first component is the CRime module library which is inspired and based on the existing Rime stack, but which introduces new abstractions which allow true modularity and support composition of services. The second component is the declarative language, which is specific to the CRime module library and supports configuration, validity checking, publishing modules, finding modules and logic reasoning.

Minor contributions of this thesis are:

- practical component reuse within the CRime module library which makes programming communication functionality for sensor networks a pleasant exercise
- knowledge representation of and reasoning about the CRime world model for

cognitive networking

- repeatable and remote experimentation capability through the declarative language which allows serialization and re-loading of experiments and the web based GUI which allows remote access.

The rest of this thesis substantiates these claims.

1.4 Organization of the Dissertation

The thesis starts with an introductory chapter. In this chapter a motivation is first provided. The motivation looks at several aspects of the domain and answers questions about the relevance of the research presented herein. Then, the overview of related work is provided from two viewpoints: existing experimentation tools and existing modular stacks. A list of contributions is then discussed.

Chapter 2 provides the background necessary for understanding the remainder of the thesis. It first discusses the ISO/OSI reference architecture and the principles that guide it. Then research work which violates the OSI principles, particularly the various flavours of cross-layer and cognitive networks is discussed. Finally, service oriented networks are defined and discussed.

Chapter 3 introduces the framework for experimenting with dynamic composition of communication services and describes its four components: the workbench, the declarative language, the module library and the physical testbed. The ProtoStack reference implementation is then introduced and discussed. The three components implemented by ProtoStack are briefly described: the ProtoStack workbench and language and the CRime module library. The VESNA based testbed on which ProtoStack has been tested is also described. The chapter concludes with a qualitative comparison of ProtoStack with the most similar systems.

Chapter 4 introduces Composeable Rime (CRime), an architecture for dynamic composition of communication services based on the existing Rime architecture. The abstractions used by CRime and their implementation are first discussed. This is then followed by an architectural comparison with Rime. Finally, a quantitative comparison between the two implementations with respect to memory footprint, processing speed and energy consumption is provided. This way, the overhead introduced by modularity and composeability is assessed.

Chapter 5 introduces the ProtoStack language. The requirements and design choices for the declarative language are first identified. It then describes the syntax and introduces the Crime ontology used by the language. Next, it shows how it supports automatic workbench population and validity checking. We argue that this language is independent of the module library and the workbench and can be adapted to other systems. We also show how it supports repeatable experimentation in terms of the CREW project. Finally, we describe a feedback collection from first time users assessing the ease of use of the

ProtoStack experimentation tool.

Chapter 6 discusses service oriented networks (SONs), a type of communication networks which use principles inspired by Service Oriented Architectures. An analysis of similarities between SONs and the Open Systems Interconnect Architecture is first presented. The concept is then illustrated on a use case and three steps in realizing it are identified. A discussion on the representation languages and logic based reasoning for SONs was followed by implementations using ProtoStack. We explained the increasing complexity required from the representation language and reasoning process for achieving various degrees of automation for SONs.

Chapter 7 discusses cognitive networking experimentation. The chapter starts by providing a theoretical discussion about what cognitive networks actually are. It then continues by discussing similarities and differences between cognitive networks, cognitive radio network and cognitive radio. Then we discuss how ProtoStack can be used for knowledge representation and logic based reasoning. Next, we discuss how ProtoStack can support the cognition loop by incorporating machine learning and decision making modules. Finally, we show how ProtoStack can be used in a cognitive radio networking scenario.

Finally, Chapter 8 concludes the thesis by highlighting the main findings and pointing out directions for future work.

2 Layered modularity

Currently used communication systems are based on layered abstractions to transport information between two end points. The reference architecture of these systems has been standardized by the International Organization for Standardization and is called Open Systems Interconnect [24]. The OSI reference model has 7 layers as illustrated in Figure 2.

Figure 2 The seven layers of the Open Systems Interconnect reference model.

The layered abstraction already implies a certain modularity by facilitating the replacement of the implementation of a layer with a new implementation as long as the latter complies with the pre-defined interfaces. All the other layers of the stack can stay the same.

The OSI model is generic and is not implemented as-is, however existing implementations of protocol stacks for communication systems conform to the OSI model and its principles. Protocols enable an entity in one host to interact with a corresponding entity at the same layer in another host. Service definitions abstractly describe the functionality provided to layer N by layer $N-1$, where N was one of the seven layers of protocols operating in the local host as depicted in Figure 3. There is a strict separation between layers and implementations are typically customized and very rigid in order to improve performance.

The OSI principles have proved very useful in practice by allowing the interworking of devices from different manufacturers; however time has also shown that they have their shortcomings, particularly the fact that they slow down research and do not favour adoption of new technology. As a result, communications systems have been accused of “ossification”. Another shortcoming seems to be the increased repetition of the same functionality across several layers. Efforts to overcome these problems have taken several forms, some of which will be discussed in this chapter as they are relevant for the remainder of the thesis.

Figure 3 The communication according to the Open Systems Interconnect reference model. Two hosts, A and B communicate using protocols at different layers of the OSI model.

2.1 Cross layer design

A relatively novel and notable research direction in communication networks refers to breaking the traditional layered structure of communication architectures and is often called cross-layer design and optimization. There are two basic cross-layer design approaches as identified in [25]: implicit cross-layer design and explicit cross-layer design.

2.1.1 Implicit cross-layer design

With implicit cross-layer design there is no explicit violation of the reference layered architecture, since no layers are merged and no new interfaces are created. The violation of the reference architecture occurs implicitly during the design stage. With this approach, a protocol on a specific layer (the designed layer) is designed taking into account the processing performed at a certain fixed layer as shown in [26][25] and Figure 4a. While this approach can optimize one or perhaps more goals, it limits network flexibility and upgradeability. By employing such a design, one network layer cannot be changed without performing the corresponding changes to the second one [26]. Even more, if several layers are designed in such a way, it is challenging if not impossible to estimate whether or not the overall system will behave in the desired manner, given the large number of variables its behaviour depends on.

2.1.2 Explicit cross-layer design

Explicit cross-layer design refers to an explicit violation of the reference layered architecture in that it leads to merging and/or splitting of layers, creation of new interfaces and/or creation of new layers. Three design trends to explicit cross-layer design can be identified: merging/splitting adjacent layers, uni/bi-directional communication between (non-) adjacent layers and vertical calibration.

Figure 4 Trends in breaking the layered architecture of communication systems.

The idea of splitting a layer into sub-layers is illustrated in Figure 4b and merging adjacent layers into a super-layer is conceptually illustrated in Figure 4c. By splitting a layer into sub-layers, the outer interfaces are preserved and new interfaces are created between the new sub-layers. A super-layer that has been created from merging layers keeps the outer interfaces with the neighbouring non-merged layers and it does not create new interfaces in the stack; it uses services provided by the lower layer and provides services via primitives to the upper layers. Typically, these types of architectures optimize for a single goal and when the goal of the network changes, such architecture has difficulties adapting to the changes [27].

Interactions between layers emerged from the need to improve the performance of wireless networks and the approach is depicted in Figure 4d. Unlike splitting/merging layers, this trend proposes interactions (uni/bi-directional communication or sharing the internal information) between protocols residing on several (non-) adjacent layers of the protocol stack, thus creating new interfaces.

In order to avoid the risk of “interface creep” [16], certain cross-layer designs use a different paradigm to avoid direct bidirectional communication between (non-)adjacent layers. This trend in breaking the layering uses a common vertical plane that spans across all layers and that can be used by each layer if it chooses to. This concept, illustrated in Figure 4e, is called vertical calibration [26][16] and it does not seem to limit scalability

and flexibility as other cross-layer design proposals do.

Cross-layer designs are algorithmic approaches proposed to solve shortcomings of the traditional protocol layering. Every cross-layer design proposal serves to highlight a specific shortcoming of the traditional protocol layering [28] (p. 142). These designs are protocol stack centric in the way that they solve protocol stack specific problems, not end-to-end network goals so they do not have significant impact on solving heterogeneous network specific problems. Another shortcoming of these designs derives from the fact that they are based on algorithmic approaches. A system designed on such an approach does not learn and adapt according to an end-to-end communication goal, but preserves a rigid structure and repeats ill functioning adaptations that previously led to poor performance [16]. Finally, most cross-layer design approaches seriously limit the modularity of the system.

2.2 Service oriented networks

By Service Oriented Network (SON), we refer to a network which makes use of SOA principles to provide end-to-end transport services. SONs look at a protocol stack as a composition of services. Each service is represented by a software module and can implement a specific function within a protocol or it can implement the protocol itself. Services then have interfaces which are described in a standard way and can then be used for composition. SONs are built using modular stacks and appropriate technology for describing the modules and their interfaces in a standard way to increase interoperability. A modular stack is built from a set of available functional blocks which are able to perform the functionality of a classical protocol stack.

2.3 Cognitive networks

A recent research trend is to transform traditional networks into so called cognitive networks. To achieve this, techniques from the field of artificial intelligence are used to solve problems in communication networks. To be self-aware, cognitive networks need to have a representation of themselves [15][17] and to be able to react to changing environments, they need to implement feedback loops incorporating machine learning and decision making technologies [16][17]. Therefore, in our understanding, a cognitive network should have:

A representation of relevant knowledge about the scope (device, homogenous network, heterogeneous network, etc)

- A cognition loop which uses AI techniques inside its states (learning techniques, decision making techniques, etc)

One approach to cognitive network design is to have a knowledge plane [29] spanning vertically over layers (thus making use of cross-layer design) and/or horizontally across

technologies and nodes (thus covering a heterogeneous environment) as depicted in Figure 4f.

2.4 Summary

This chapter provided the background necessary for understanding the remainder of the thesis. It first discussed the ISO/OSI reference architecture and the principles that guide it. Then research work which violates the OSI principles, particularly the various flavours of cross-layer and cognitive networks was discussed. Finally, service oriented networks were defined. These concepts benefit from the composition of communication services as will be shown in the thesis.

The next chapter introduces a framework for the dynamic composition of services and a high level description of its reference implementation ProtoStack.

3 A framework for dynamic composition of services

In this chapter we propose a framework for dynamic composition of services. We argue that the framework is generic enough to be applied to existing systems as well as future ones. We claim that such framework is suitable for accommodating both expert and non-expert experimenters and that it is particularly well suited for remote experimentation. In this chapter, we substantiate these claims by describing the framework and its functionality, compile a set of requirements, present a reference implementation and perform a feature-wise comparison with similar existing systems.

3.1 The components of the proposed framework

The overall framework has four functional components: the physical testbed, the module library, the declarative language, and the workbench as depicted in Figure 5.

Figure 5 The four components of the framework for the composition of services.

3.1.1 The physical testbed

By physical testbed we refer to a set of machines on which the stack built by the composition of services is deployed and tested. The machines need to support the module library and any additional software that is planned to be deployed. When implementing the framework, the type of machines will determine the selection of the module library, or vice versa. To better represent the likely future deployments, it is desirable that the supported machines of the physical testbeds are as diverse as possible (i.e. heterogeneous). This implies that the module library should be as portable as possible. Further, depending on the location and configuration of the testbed, procedures for resetting the machines in case of fatal errors may be challenging, therefore it is desirable that the deployed binary image is fault proof.

3.1.2 The module library

The module library consists of the source code of the modules used for composing communication services. Besides the code for the modules it also contains additional code

necessary for compiling and linking the binary image. Depending on the programming language, the modules are implemented as classes or as a set of functions, each in its own file. The modules provide services to each other through interfaces. One module may correspond to a basic service such as routing (e.g. shortest path routing) or may correspond to composite services such as an entire protocol (e.g. IP).

3.1.3 The declarative language

The declarative language is used to instantiate and configure modules from the module library. Subsequently, tools that are able to perform validity checking, error detection, compilation of binary images and their deployment in the physical testbed can be used. The declarative language is a natural intermediate level of abstraction between a user interface such as the workbench and the program code. There is a correspondence between elements of the workbench and the elements of the language. A translation tool is employed to translate from the workbench's elements to the declarative language. In some cases, the user may want to bypass the user interface and directly use the language for describing and configuring the modules in a stack prior to the experiment. As a consequence, the language typically also needs to be human readable, possibly easy to learn and should use intuitive code words.

3.1.4 The workbench

The workbench is thought of as a control panel which allows the experimenter to configure, start, run, retrieve and visualize the results of an experiment. Therefore the workbench should first and foremost contain functionality that would allow the experimenter to intuitively compose a stack and provide initialization parameters. This is typically achieved by having a region where available modules are listed in graphical and/or textual form (e.g. shortest path routing, transmission control protocol). The modules can then be dragged to a workspace, connected and specific parameters initialized (e.g. time to live, maximum number of retransmissions). Some error checking mechanism should be implemented to ensure that incompatible elements are not wired together and that parameters are inside the permitted ranges. Additionally, the workbench can contain an area where the experiment can be visualized while running (e.g. number of dropped packets, delay) and a summary of the completed experiment can be provided (e.g. total time per operation).

3.2 Requirements

The framework for dynamic composition of services has to support design and experimentation of new communication services and modular protocol stacks. In order to achieve this, we identify a set of requirements which help fulfill this objective:

- **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.

- **Flexibility** – the components of the workbench 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). The components should also be easy to extend and upgrade.
- **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.
- **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.
- **Remote experimentation** – remote users should be able to define and perform experiments and download the result. This can be most easily achieved through a web portal.

Reference implementations of the framework for dynamic composition of communication services should take this set of requirements as guidelines.

3.3 Reference implementation: the ProtoStack tool

In this section we briefly introduce a reference implementation of the framework for the dynamic composition of services called ProtoStack. We discuss the implementation of the framework and we identify the communities which may find such a tool useful. Two of the four components – the module library and the declarative language – represent, together with the framework, major contributions of this thesis and they will have dedicated chapters (Chapters 4 and 5). In our implementation, the workbench and the declarative language are tightly integrated and will be discussed together in Chapter 5.

The implementation of ProtoStack was triggered by a wireless sensor network testbed and is used for experimentation with cognitive radio and cognitive networking in the framework of the CREW project [30]. As such, ProtoStack is designed in a way to ease research and experimentation with communication networks, particularly with cognitive networks. The system was designed so that

- i) an advanced user such as the component developer needs to focus on developing the component and make it work with Contiki [31] and
- ii) a novice user needs only to focus on composing services in a stack using the workbench.

The physical testbed is based on VESNA sensor network platform⁴ to which the Contiki OS has been ported, partly due to of the adaptive Rime architecture which comes with it [21]. This physical testbed posed constraints that determined the selection of Composeable Rime (CRime) as the module library.

The CRime module library in the reference implementation is a purpose-built set of protocols influenced by and based on the Rime [21] architecture and is introduced in more detail and thoroughly compared against Rime in Chapter 4.

The declarative language we selected for ProtoStack is based on the Resource Description Framework (RDF) [32], a standard semantic web language. The implementation uses the Turtle⁵ syntax together with existing standardized vocabulary and a custom ontology. The language and the way it supports, together with the workbench, dynamic composition of services are discussed in detail in Chapter 5.

The workbench is tightly integrated with the language and is implemented using WireIt⁶, an open source javascript library which enables the creation of full web graph editors. This is briefly discussed in Chapter 5.

Figure 6 ProtoStack: an implementation of the framework for composing communication services. The implementation addresses the sensor networks domain.

In Figure 6 we illustrate the steps for dynamic composition of services using the ProtoStack tool. The component developer develops a module, manually tests it and makes sure everything works as intended, and, at the end he/she needs to write few lines of Turtle statements (i.e. triples) which specify basic characteristics of the new module⁷ (i.e. the name of the module, how many and what type of primitives it implements, etc.). Once this is done, ProtoStack parses the Turtle triples from the new module and stores

⁴ <http://sensorlab.ijs.si/hardware.html>

⁵ <http://www.w3.org/TeamSubmission/turtle/>

⁶ <http://nevrlic.github.com/wireit/>

⁷ The developer also needs to provide an icon that is relevant for the functionality of the new module.

them in the triple store (arrow 1 in Figure 6). When the user starts using the system, the workbench will be automatically populated with modules based on the statements stored in the triple store and rendered (arrow 2 in Figure 6).

The user will then compose the desired stack, insert the required parameters and press a button to run the stack on the physical testbed (arrow 3 in Figure 6). When such a command is received, the system first checks for consistency by making sure the composition of modules is valid and that the input parameters are in a valid range. If all is fine, some C code is automatically generated based on what the user composed (arrow 4 in Figure 6). This code configures the CRime stack. Finally, the source code is compiled into a binary form representing an image that is uploaded on VESNA (arrow 5 in Figure 6).

The experiment description resulting from the stack composed and configured by the user is saved and can be re-used at a later time for re-running the same experiment. All this can be done remotely thanks to the web based workbench.

3.4 Discussion

In this section, we first discuss how existing tools and systems which support the composition of communication services relate to the proposed framework from the component and requirements perspective. Then, we discuss how ProtoStack, the reference implementation of this framework relates to the experimentation tools and modular stacks identified as related in Section 1.2.

3.4.1 On the proposed framework

The four component framework we proposed in this Chapter is generic enough and well suited for design and experimentation with dynamic composition of communication services. This claim is supported by the fact that successful existing tools such as x-Kernel and Click implement all four components as can be seen from Table 3.

Table 3: Comparison of tools which support the composition of communication services from the perspective of the defined framework.

Name	Workbench	Declarative language	Module library	Physical testbed
x-Kernel	✓	✓	✓	✓
Click	✓	✓	✓	✓
SNA	•	✓	✓	✓
ProtoStack	✓	✓	✓	✓

Both these tools have proved to be useful for experimentation by passing the test of

time and having a wide user community. Also, within the more recent SNA project, all components but the workbench are addressed. Of the four tools, only SNA and ProtoStack address the sensor networks domain.

Table 4: Comparison of tools which support composition of communication services from the perspective of the requirements.

Name	Modularity	Flexibility	Easy programming	Reproducibility of experiments	Remote experimentation
x-Kernel	✓	✓	✓	✓	•
Click	✓	✓	✓	✓	•
SNA	✓	✓	✓	•	•
ProtoStack	✓	✓	✓	✓	✓

The comparison between the tools which support the composition of communication services from the perspective of the requirements identified in Section 3.2 is summarized in Table 4. It can be seen that all four tools address modularity, flexibility and easy programming. Reproducibility of experiments seems to be explicitly addressed only by the ProtoStack tool. However, x-Kernel and Click also provide some support in this respect through configuration scripts which can be manually saved and re-used at a later time. SNA does not currently address this requirement. With respect to remote experimentation support, only ProtoStack explicitly addresses this aspect by already incorporating the web based infrastructure to support remote design and experimentation. In order to use the other three tools in a remote manner, additional infrastructure has to be deployed.

3.4.1 On the ProtoStack reference implementation

In Table 5, a feature based comparison of ProtoStack, the reference implementation of the framework we propose, with existing experimentation tools identified in Section 1.2 is provided. It can be seen that, similar to x-Kernel and Click, ProtoStack also introduces a set of abstractions to support modularity. X-Kernel introduces three abstractions (protocol, session and message), Click introduces only one (the element) and ProtoStack introduces two (module and stack). ProtoStack's module seems closest to Click's element. Similar to the other tools, ProtoStack supports user friendly configuration using a GUI and it uses a declarative language as an intermediate abstraction between the GUI and the implementation of the modules of the stack. While the other two systems use a graph composition model, ProtoStack currently uses a tree model which seems to be sufficient for the current state of the system.

Table 5: Comparison of related experimentation tools.

Project	Abstraction	Composition model	Configuration	Implementation language
x-Kernel	protocol, session, message	Graph	Textual graph description language, graph editor + Auto-generated C code	C (minimal OO style)
Click	Element	Directed graph	Click language for configuration (declarative), Clicky GUI + Auto-generated C++ code	C++
ProtoStack	Module, stack	Tree	Graph editor + Auto-generated C code	C

In Table 6, a feature based comparison of ProtoStack, the reference implementation of the framework we propose, with existing approaches to modular stacks identified in Section 1.2.2 is provided. While projects such as the x-Kernel and Click provide means to experiment with modular stacks, they target conventional networks. Furthermore, x-kernel looks at ways to design OS kernels that support modular stacks while Click only addresses modular router reconfigurations. RNA proposes recursive network architecture using modular stacks and is based on Click. Through the CRime module library, ProtoStack introduces abstractions for experimentation with modular stacks on constrained resource devices such as sensor nodes at all layers. By addressing a domain with different requirements, these abstractions are different from the ones provided by x-Kernel, Click and RNA. Additionally, CRime includes abstractions that support experimentation with modular protocol stacks in the area of cognitive networks and ProtoStack is designed to support experimentation with service oriented networks.

Table 6: Comparison of related approaches to modular stacks.

Project	Scope	Configuration
x-Kernel	All layers	Textual graph description language, graph editor
Click	Network	Click language for configuration (declarative), Clicky GUI
SNA (SP, MNL, DSN) Rime	Network and below All layers	Declarative -
RNA	Meta-protocol	Click
ProtoStack	All layers	Graph editor (declarative language)

The Sensor Network Architecture (SNA) project where SP, MNL and DSN were developed is an effort to transform existing protocols into composition of modular

functional blocks and, at the same time, introducing a declarative language which would enable easier prototyping. While SNA targets the same domain as ProtoStack, their approach is significantly different. SNA takes a bottom-up approach to modular protocol design. They look at a large number of existing implementations or protocols for sensor networks in an attempt to find common functionality that repeats. Once identified, this functionality is implemented in a module that can be then used by the protocol. In the case of ProtoStack, functional modules have been identified and defined by Rime in a top-down approach by implementing a protocol, after defining the required functionality. Additionally, Rime provided separation between the operations performed by the protocol and the header generation. This separation facilitates cross-layer experimentation using the Rime architecture. Although the CRime module library in the ProtoStack tool is inspired and built on Rime, the resulting architecture and abstractions are significantly different from those in Rime. ProtoStack introduces abstractions necessary for modular stack composition and cognitive networking that are not present in Rime.

3.5 Summary

This chapter introduced the framework for experimenting with dynamic composition of communication services and described its four components: the workbench, the declarative language, the module library and the physical testbed. The ProtoStack reference implementation was then introduced and discussed. The three components of the framework implemented by ProtoStack were briefly described: the ProtoStack workbench and language and the CRime module library. These modules are elaborated in Chapters 4 and 5. The VESNA based testbed on which ProtoStack has been tested was also described. The chapter concluded with a qualitative comparison of ProtoStack with the most similar systems.

In the next chapter, the Composable Rime module library with its abstractions is introduced. CRime is then compared to Rime, the reference implementation of a stack for cross layer design, from an architectural point of view and the overhead introduced as a trade-off for flexibility is quantitatively assessed.

4 The Composeable Rime module library

This chapter introduces, discusses and evaluates the Composeable Rime (CRime) module library. First, the CRime abstractions are described, followed by an architectural comparison between CRime that we have proposed and the existing Rime. Then, we compare the two module libraries with respect to the resources they consume on a node. We look at the code footprint and memory consumption, at the processing speed and at the energy consumption to assess the cost of modularity.

4.1 CRime abstractions

CRime is a new architecture designed to support the composition of communication services which is inspired by and built upon the Rime [21] architecture. CRime introduces three abstractions the *amodule*, the *pipe* and the *stack*.

The *amodule* (short from abstract module) is a generic building block of the CRime stack. Behind each instance of an amodule hides a communication service [37] such as broadcast or multihop. The communication service is an implementation of a network function such as protocol or algorithm and contains only the execution logic of that function. Several amodule instances can be arranged in a pipeline to form a communication stack.

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, therefore a single channel can only have one associated pipe at a time. This implementation, while not the most efficient approach from a software engineer's perspective, nor the most resource efficient in terms of memory, instantiates the concept of vertical layer and is the first building block in the implementation of the knowledge plane required for experimentation with cognitive networks [17]. The approach is also a compromise between memory footprint and the complexity required by the autogenerated C code based on user input.

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. Using the stack abstraction, an independent communication stack can reside on each channel. These stacks merge at the application layer or below it. Figure 7 depicts the three abstractions in an example of a 1 channel - 1 stack and an example of a 3 channel – 3 stack

communication system.

(a) (b)

Figure 7 Example of CRime stacks: (a) 1 channel – 1 stack example and (b) 3 channel – 3 stack.

4.1.1 CRime packets

In CRime, the packets are formed the same way as in Rime and this is due to the fact that, like Rime, CRime is built on top of Chameleon and uses it for forming packets [21]. Chameleon can be seen as an abstraction layer which adapts packets to the actual underlying MAC protocols. Chameleon uses a block of memory where the packet header and the packet data are located. This memory is filled by the application and Rime protocols. A set of predefined packet attributes help to point to the corresponding header attribute. The form of the header attribute is influenced by the stack and is automatically handled by Chameleon, which doesn't include attributes corresponding to non-defined memory locations.

4.1.2 Theoretical modeling of the CRime communication stack

The theoretical model behind the CRime communication stack is a tree as depicted in Figure 8.

Figure 8 Tree model with one pipe per stack which is used for the CRime implementation.

Each node of the tree includes one or more abstract modules which are connected and communicate in a horizontal manner. Each leaf of the tree corresponds to a channel and

the corresponding branch forms a stack and has a pipe attached. Recursion is used to walk through the tree.

4.1.3 CRime amodule

A CRime abstract module defines a generic interface which currently contains 11 generic primitives (see Algorithm 1). Any CRime communication service has to implement some of the generic primitives defined by the amodule interface. An example is the `c_abc` module which performs **anonymous broadcasting (abc)**: it sends unsigned messages to its neighbors. `c_abc` is an instance of the amodule interface and implements five primitives: `c_open`, `c_close`, `c_send`, `c_recv` and `c_sent`. The current set of 11 generic primitive functions were determined based on the functions used by the Rime protocols (see summary in Appendix A) and may change as the system evolves (i.e. new modules are added, functionality of existing modules is changed).

Algorithm 1: The interface of the abstract module.

```

void (* c_open)(struct pipe *p, struct stackmodule_i *module);
void (* c_close)(struct pipe *p, struct stackmodule_i *module);
int (* c_send)(struct pipe *p, struct stackmodule_i *module);
void (* c_recv)(struct pipe *p, struct stackmodule_i *module);
void (* c_sent)(struct pipe *p, struct stackmodule_i *module);
void (* c_dropped)(struct pipe *p, struct stackmodule_i *module);
void (* c_timed_out)(struct pipe *p, struct stackmodule_i *module);
int (* c_discover)(struct pipe *p, struct stackmodule_i *module);
void (* c_read_chunk)(struct pipe *p, struct stackmodule_i *module);
void (* c_write_chunk)(struct pipe *p, struct stackmodule_i *module);
void (* c_new_route)(struct pipe *p, struct stackmodule_i *module);

```

Amoudles also define triggers. This means that any amodule can be invoked when a timer expires. When an amodule is invoked in such a way, the underlying modules will be invoked as well. This functionality is useful when the protocol designer wants to design a stack which is able to repeat transmissions of packets.

Several instances of the amodule interface are added to a one-dimensional array structure which forms a stack (note that this is a simple stack of the 1-channel – 1 stack type). The generic primitives use recursion to walk through the one-dimensional array as shown in the example of `c_open` primitive listed in Algorithm 2.

4.1.1 CRime pipe

The CRime pipe is a data structure which stores parameters used by the modules in the stack. It can store parameters that are of interest to several modules and also module specific parameters which are defined as specific structures and then included in the pipe.

The pipe includes two sets of addresses as can be seen in Algorithm 3. One set corresponds to the incoming packet (the `rimeaddr_t` with the `in_` prefix in Algorithm 3) while the other corresponds to the outgoing packet (the `rimeaddr_t` with the `out_` prefix in Algorithm 3). The addresses themselves have the same scope as in the original Rime architecture: the sender and receiver addresses correspond to singlehop peers (direct neighbours) while the esender and ereceiver addresses correspond to multihop peers.

Algorithm 2: The code for the `c_open` primitive.

```

void c_open(struct pipe *p, struct stackmodule_i *module, uint8_t len) {
    if (module[len].c_open == NULL) return;
    uint8_t modno = stack[module[len].stack_id].modno;
    module[len].c_open(p, &module[len]);
    len++;
    if ((len >= 0) && (len < modno)) {
        c_open(p, module, len);
    }
}

```

Algorithm 3: The pipe data structure.

```

struct pipe {
    struct channel *channel;
    uint16_t channelno;
    struct queuebuf *buf;
    struct packetbuf_attrlist *attrlist;
    rimeaddr_t in_sender, out_sender;
    rimeaddr_t in_receiver, out_receiver;
    rimeaddr_t in_esender, out_esender;
    rimeaddr_t in_ereceiver, out_ereceiver;

    int status;
    int num_tx;
    uint8_t seq_no;
    uint8_t hop_no;

    //stack specific data structure
    ....
};

```

4.1.2 CRime stack

While a set of amodules together with a pipe can form a 1-channel – 1 stack communication service, these abstractions are insufficient to support a more complex n-channel – n stack communication services. Therefore, the stack structure is designed to handle the tree specific to the n channel – n stack model. Particularly, the stack structure handles branching and merging of the communication services based on the implemented primitives. Algorithm 4 shows how the `stack_open` function invokes the `c_open`

primitive of the top module in each stack.

Algorithm 4: The stack_open function.

```

void stack_open(struct stack_i *stack){
    int p;
    for (p = 0; p < STACKNO; p++) {
        int modno = 0;
        if (stack[p].amodule[modno].c_open != NULL){
            c_open(stack[p].pip, stack[p].amodule, modno);
        }
    }
};

```

4.2 Architectural comparison with Rime

The Rime stack was designed to support modular stacks and cross-layer protocol design [21]. The CRime stack is based on the Rime design and additionally introduces abstractions which allow dynamic composition of stacks.

4.2.1 Communication primitives

The Rime stack as proposed in [21] defined 10 communication primitives. The stack has evolved since and currently comprises around 20 primitives (depending on which modules are considered communication primitives). In CRime, the aim was to implement modularity for dynamic reconfiguration while keeping all the functionality from Rime. We achieve this and we prove that CRime is configurable. The number of communication primitives is reduced in the CRime architecture due to the fact that some Rime primitives are further decomposed in CRime. CRime breaks modules with repeated functionality from Rime to facilitate composeability and component reuse.

The dedicated Rime primitives that implement periodic retransmissions, also referred to as stubborn, are replaced in CRime by the amodule triggers. Furthermore, CRime allows transforming any primitive into a stubborn one by activating the trigger mechanism. Rime primitives which implement the same functionality such as polite and identified polite are reduced to one in CRime, thus increasing component reuse. This is enabled by the dynamic composition of amodules which is not possible in Rime.

To further clarify the above claims; let us consider a subset of six Rime primitive whose functionality is summarized in Table 7. The selected primitives are by no means exhaustive but they are sufficient to explain why CRime needs fewer modules to achieve the same functionality. The CRime primitives necessary to implement the same functionality as provided by the Rime primitives from Table 7 are summarized in Table 8.

The stubborn unicast module (stunicast, not in Table 7) does the same thing for the unicast module (see Table 7) as the stubborn broadcast module (stbroadcast, see Table 7) does for the broadcast primitive (also in Table 7). Stunicast enables the repetitive sending

of the same packet using unicast while stbroadcast enables the repetitive sending of the same packet using broadcast.

Table 7: Selected Rime communication primitives (compiled based on the Contiki 2.5 Rime).

Name	Description
abc	The anonymous best-effort singlehop broadcast primitive (abc) is the most basic communication primitive in Rime. The abc primitive provides means for upper layers to send a data packet to all local neighbors that listen to the channel on which the packet is sent. No information about who sent the packet is included in the transmission, and the module adds no headers to outgoing packets. All other Rime primitives are based on the abc primitive.
polite	The polite module sends one local area broadcast packet within one time interval. If a packet with the same header is received from a neighbor within the interval, the packet is not sent. The polite broadcast module does not add any packet attributes to the header of the outgoing packets apart from those added by the upper layer.
broadcast	The broadcast module sends a packet to all local neighbors. The module adds the singlehop sender address as a packet attribute to outgoing packets. All Rime primitives that need the identity of the sender in the outgoing packets use the broadcast primitive, either directly or indirectly through any of the other communication primitives that are based on the broadcast primitive.
stbroadcast	The stbroadcast module provides stubborn best-effort local area broadcast. A message sent with the stbroadcast module is repeated until either the message is canceled or a new message is sent. The stbroadcast module does not add anything to the header of the outgoing packet; however, the packets that are sent are identified because this module calls the broadcast module.
ipolite	The ipolite module sends one local area broadcast packet within one time interval. If a packet with the same header is received from a neighbor within the interval, the packet is not sent. Ipolite always calls the broadcast module, therefore the packets that are finally sent out are identified with the sender address. This is difference from the polite module which always calls the abc module, meaning that packets that are finally sent out have no sender ID.
unicast	The unicast module sends a packet to an identified singlehop neighbor. The unicast primitive uses the broadcast primitive and adds the singlehop receiver address attribute to the outgoing packets. For incoming packets, the unicast module inspects the singlehop receiver address attribute and discards the packet if the address does not match the address of the node.

Table 8: Selected CRime communication primitives.

Name	Description
c_abc	The CRime anonymous best-effort singlehop broadcast primitive (c_abc) is the most basic communication primitive in Rime. No information about who sent the packet is included in the transmission, and the module adds no headers to outgoing packets.

c_polite	The c_polite module sends one local area broadcast packet within one time interval. If a packet with the same header is received from a neighbor within the interval, the packet is not sent. The polite broadcast module does not add any packet attributes to the header of the outgoing packets apart from those added by the upper layer.
c_broadcast	The c_broadcast module sends a packet to all local neighbors. The module adds the singlehop sender address as a packet attribute to outgoing packets. All Rime primitives that need the identity of the sender in the outgoing packets use the broadcast primitive.
c_unicast	The c_unicast module sends a packet to an identified singlehop neighbor. The unicast primitive uses the broadcast primitive and adds the singlehop receiver address attribute to the outgoing packets. For incoming packets, the unicast module inspects the singlehop receiver address attribute and discards the packet if the address does not match the address of the node.

The dependency graph of the modules summarized in Table 7 is depicted in Figure 9. It can be seen that any protocol based on the 6 primitives in Table 7 uses the abc module. Any path through the Rime dependency graph forms one Rime protocol stack. It can be seen that when an unidentified message needs to be sent, then either the abc module is invoked, or the sequence of polite-abc modules are invoked. For identified message, the broadcast-abc or ipolite- broadcast-abc modules are used, while for repetitive sending of identified messages, the stbroadcast-broadcast-abc modules are invoked.

The same functionality can be achieved in CRime using only 4 modules and dynamic composition. CRime uses the c_abc, c_broadcast, c_polite and c_unicast modules which are designed to do the same task as abc, broadcast, polite and unicast. The functionality of the Rime ipolite primitive is achieved by using c_polite over c_broadcast. In such a configuration, c_polite, similar to polite in Table 7, will make sure that one message will be broadcasted in the time interval only if a similar message has not been heard from neighbors. The broadcast module will make sure that the sender ID is inserted in the packet header. The composition of these two modules will result in identified polite (ipolite) broadcast. The Rime stbroadcast module is unnecessary in CRime as a trigger can be attached to the c_broadcast module.

In the Rime dependency graph depicted in Figure 9, there are a total of 6 possible paths corresponding to 6 possible stacks, while in the CRime dependency graph from the same figure, the number of possible paths is 11. Perhaps not all the possible paths will make sense and sometimes paths will contain redundant modules. For instance, in the CRime <c_unicast, c_broadcast, c_abc> path, the c_abc module is redundant as the same functionality can be achieved by the <c_unicast, c_broadcast> path. There will be paths, however, which permit experimenting with new configurations which may not make sense to develop otherwise and perhaps would not even be obvious otherwise.

(a)

```

<abc>
<polite, abc>
<broadcast, abc>
<unicast, broadcast, abc>
<ipolite, broadcast, abc>
<stbroadcast, broadcast, abc>

```

(b)

```

<c_abc>
<c_broadcast>
<c_broadcast, c_abc>
<c_polite, c_abc>
<c_polite, c_broadcast, c_abc>
<c_polite, c_broadcast>
<c_unicast, c_polite, c_abc >
<c_unicast, c_broadcast >
<c_unicast, c_broadcast, c_abc >
<c_unicast, c_polite,
c_broadcast, c_abc >
<c_unicast, c_polite,
c_broadcast,>

```

Figure 9 Dependency graph in (a) Rime and (b) CRime (right). It can be seen that with the CRime approach, there are more possible combinations of modules which can form a stack (11 vs 6).

4.2.2 Support for modular stacks

Reconfiguring a stack in Rime requires good understanding of the Rime design and implementations and involves manually writing a fair amount of C code, as the existing Rime tree is hard coded. Reconfiguring a stack in CRime also requires fairly good understanding of the CRime design and involves writing a few lines of C code to initialize the one-dimensional arrays and the required variables. Arguably, the CRime approach is more developer friendly and permits quicker configuration of stacks. Generating a tree out of existing modules by initializing a set of one-dimensional arrays is typically less challenging than “re-wiring” a hard coded tree.

Furthermore, in the case of CRime, the initialization code can be generated automatically. In Rime, it is impossible to automatically generate configuration code for a new stack, due to the module specific data structure and parameters and the hard wiring of modules. In CRime the recursion is explicit and the implementation allows flat initialization which is easy to generate automatically. In Rime, the recursion is implicit through the callback mechanism and it is not possible to devise a template which would allow automatic generation.

If we consider the following three module Rime stack `<unicast, broadcast, abc>`, then Figure 10 illustrates the sequence in which the corresponding functions are called. The sequence of functions which are invoked starting from the application level and going down to the `abc` module is indicated with continuous line arrows, while the sequence of functions invoked by Rime when a message is received is indicated with dashed lined

arrows. The program logic for sending a packet using the Rime unicast stack is hard coded in the open, close and send functions corresponding to each module. In order to change this sequence, the code of each function, together with the type of connection in the corresponding data structure, needs to be changed. For instance, if we wished to generate the following Rime stack <unicast, ipolite, broadcast, abc>, then the **struct** `broadcast_conn c;` from the unicast module would need to be replaced by the corresponding ipolite connection **struct** `ipolite_conn c;` and the ipolite callbacks need to be properly set to point to unicast sent and recv functions and the code of the unicast open, close and send functions would need to be changed to explicitly call the corresponding ipolite functions.

Figure 10 Illustration of the way the Rime unicast stack is implemented.

Figure 11 Illustration of the way a possible CRime unicast stack is implemented.

The equivalent CRime stack is <c_unicast, c_broadcast, c_abc> as depicted in Figure 11. The sequence of functions called when a packet needs to be sent or received is pre-determined by the corresponding `c_open`, `c_close`, `c_send`, `c_recv` and `c_sent` generic primitives from the amodules. Which functions will be called depends on what the generic primitives point to. In Figure 11, for the `c_unicast` module, the generic primitives point to the corresponding unicast implementations. If we wished to generate the <c_unicast, c_polite, c_broadcast, c_abc> stack, we would need to replace the `c_unicast` module in `amodule[2]` with the `c_polite` module and add an additional amodule containing the `c_unicast` module. The corresponding code consists of correctly initializing

a one dimensional array. For completeness we list the code corresponding to `<unicast, broadcast, abc>` in Appendix B and the code corresponding to `<c_unicast, c_broadcast, c_abc>` in Appendix C.

4.2.3 Support for cross-layering

Rime was designed with strong support for cross-layer experimentation by defining a data structure which is accessible to all the layers in a given stack and which contains a number of parameters, some of which are used for creating packets. Currently there are 27 attributes (and parameters) supported in this structure but this can be easily extended with additional ones. CRime takes full advantage of this design feature without modifying anything. Additionally, through the pipe structure, CRime permits additional parameter sharing.

4.2.4 Support for cognitive networking

CRime provides support for cognitive networking in two ways. First, due to support for cross-layer design, it facilitates information sharing through layers and with the control plane. Cognitive components can have as such access to the shared information. Second, due to the modular implementation it is easy to add cognitive components. For instance, a routing module which uses reinforcement learning can be added and wired just like any other module.

4.2.5 Portability

As an extension of Contiki OS, the CRime stack benefits from the advantages which come with the OS with respect to portability. Contiki OS has been ported to several platforms and can also be used with the Cooja simulation tool. As such, CRime can also be used on those platforms as well as with Cooja.

4.3 The cost of composeability

Rime was one of the first communication architectures for sensor networks to provide a set of abstractions in order to support adaptive communication. Compared to non-adaptive architectures, Rime incurred higher execution time [21]. CRime is, to the best of our knowledge, the first architecture that supports dynamic composition of services for sensor networks. Through dynamic composition of services, CRime helps to speed up the design, prototype and evaluation of new communication services. Compared to Rime, CRime introduces new abstractions which reflect in overhead in terms of code size, execution time and power consumption. In this section we assess the costs that CRime's overhead introduces versus Rime. In the comparison we focus on a relevant subset of modules and the example applications that include them. The selected Rime primitives

against which we compare are `abc`, `broadcast`, `polite`, `unicast` and `multihop`, while the Rime applications examples are `example_name_of_the_application`.

4.3.1 Experimental setup

In order to compare the components of the Rime stack against the components of the CRime stack in terms of the image size we use the VESNA sensor node⁸, a hardware platform developed in our lab and used in several sensor testbeds in Slovenia [38]. The VESNA sensor node is equipped with a ST ARM Cortex M3 32 bit microcontroller running up to 72 MHz, 1 MB of FLASH, 96 kB of SRAM and 128kB of fast (2,25 MB/s) non-volatile MRAM memory (NVRAM). The hardware has a fully modular design and can be programmed via RS-232 compatible interface or standard JTAG providing debug capabilities.

For the numbers provided in this chapter, we used the following experimental setup: VESNA sensor node with TI CC1101 radio module connected via serial line to a Lenovo X200 machine (Intel Core Duo CPU @2.53 GHz with 4GB or RAM) with a Contiki 2.5 OS port. The notebook runs Windows 7 Enterprise on the computer on which we installed the open source development environment consisting of Cygwin, Codesourcery tool-chain, OpenOCD, Eclipse Helios. For energy consumption measurements, we connected the hardware to a Tektronix TDS5104B oscilloscope.

4.3.2 Image size

We first compare the components of the Rime stack against the components of the CRime stack in terms of the image size. We looked at the size of the code (`.text`), the size of the initialized (`.data`) and non-initialized (`.bss`) static variables and list the results in Table 9. It can be seen that the Rime and CRime components differ just in the size of the code.

The `c_abc` code size is 36 bytes larger than the `abc` code size. This is mainly due to the fact that the `c_abc` module implements some functionality, namely the `c_abc_recv` function, the correspondent of which in the Rime version is implemented by the module above or by the application. The multi-hop and mesh communication primitives are noticeably larger in CRime compared to Rime. This is due to several calls to a function implemented by the `amodule` block which sets values in the pipe structure. The code footprint could be reduced if we optimized only for that, however, we also optimized for maintainability and easy debugging.

⁸ <http://sensorlab.ijs.si/hardware.html>

Table 9: Comparison of Rime and CRime components with respect to code size (.text), initialized static variables (.data) and uninitialized static variables (.bss). All values are in bytes.

Rime components				CRime components			
Name of component (.o)	.text [B]	.data [B]	.bss [B]	Name of component (.o)	.text [B]	.data [B]	.bss [B]
abc	192	0	0	c_abc	228	0	0
broadcast	248	0	0	c_broadcast	196	0	0
polite	604	0	0	c_polite	580	0	0
unicast	252	0	0	c_unicast	236	0	0
multihop	468	0	0	c_multihop	652	0	0
stbroadcast	348	0	0				
ipolite	712	0	0				
stunicast	512	0	0				
				amodule	2572	0	4
				stack	3760	24	0

For the broadcast, polite and unicast communication primitives, the CRime version results in smaller code size than then Rime version. This is due to the fact that in the case of CRime the functions do not directly call corresponding functions from the modules below. In the CRime case this overhead is solved by the amodule component. If we add in the c_broadcast module explicit calls to the c_abc module, we pay 8 bytes in the size of the code for each call we add.

The stbroadcast, ipolite and stunicast modules from Rime have no direct equivalent in CRime. As discussed in the architectural comparison between the modules, the functionality of these is replaced by triggers and by the modular composition of the stack. The two CRime specific modules, which enable modularity, are the amodule and the stack. The code footprint of these is relatively large if we compare them to the other modules. They also use some static variables as can be seen in the corresponding .data and .bss columns of Table 9.

Some applications using the Rime modules are already available with the Contiki code and we created a similar application (e.g. dummy sending of packets) entitled example-crime which we use with the composed stacks. The evaluation of the application memory footprint for the two stacks is listed in Table 10 and in Table 11, while the difference is listed in Table 12. It can be seen that the size of the code of the applications which use CRime stacks is about 16% larger (~13.000 bytes)

Table 10 The code size (.text), initialized static variables (.data) and uninitialized static variables (.bss) of five Rime example applications. All values are in bytes.

Application name	Bin fsize [B]	.text [B]	.data [B]	.bss [B]
example-abc	81188	79056	1636	5112
example-broadcast	81260	79128	1636	5116
example-polite	83116	80984	1636	5152
example-unicast	81500	79368	1636	5116
example-multihop	83260	81112	1652	5796

Table 11 The code size (.text), initialized static variables (.data) and uninitialized static variables (.bss) of five CRime example applications (the applications are compatible and do the same thing as the Rime applications from). All values are in bytes.

Application name	Bin fsize [B]	.text [B]	.data [B]	.bss [B]
example-crime (c_abc)	94932	92800	1636	5112
example-crime (c_broadcast)	94976	92840	1640	5108
example-crime (c_polite)	96123	94000	1636	5112
example-crime (c_unicast)	95364	93224	1642	5108
example-crime (c_multihop)	97040	94880	1664	5816

Table 12 The cost of CRime example application in terms of code size (.text), initialized static variables (.data) and uninitialized static variables (.bss). Data compiled based on the values in Table 10 and Table 11. All values are in bytes.

Difference between application using the CRime stack and the equivalent application using the Rime stack	Bin fsize		.text		.data		.bss	
	[B]	[%]	[B]	[%]	[B]	[%]	[B]	[%]
c_abc - abc	13744	16.9	13744	17.3	0	0.00	0	0.00
c_broadcast - broadcast	13716	16.8	13712	17.3	4	0.02	-8	-0.01
c_polite - polite	13016	15.6	13016	16.0	0	0.00	-40	-0.07
c_unicast - unicast	13864	17.0	13856	17.4	8	0.04	-8	-0.01
c_multihop - multihop	13780	16.5	13768	16.9	12	0.07	20	0.03

4.3.3 Processing speed

We expect that CRime is slower than Rime in terms of processing speed due to the overhead the stack and amodule abstractions add. In Table 13 we list the evaluation of the processing speed for opening and closing a connection as well as for sending and receiving packets with the two stacks. We used the `abc` and `c_abc` communication primitives for the evaluation. The results are averaged over 100 runs.

Table 13: CRime vs Rime processing speed.

Rime operations			CRime operations		
Name	Duration [μ s]	Duration [%]	Name	Duration [μ s]	Duration [%]
open	59.0	23.0	open	107.0	17.7
send	104.0	40.5	send	380.0	61.0
recv	71.5	27.8	recv	96.5	15.5
close	22.0	8.50	close	67.0	5.70
Total	256.5	99.8		622	99.9

It can be seen that in both stacks, the most time consuming operation is sending a packet. Rime needs on average 104 μ s to send one packet while CRime needs 380 μ s, an increase by a factor of 3.5. If we consider the sequence of operations `open`→`send`→`recv`→`close`, it can be seen that Rime spends 40% of the time sending the packet while CRime uses 61% of the time for the same task. In Rime, the second most time consuming operation is processing the received packet throughout the stack. The CRime `open`, `close` and `recv` operations are relatively simple; the overhead comes mostly from the code necessary to propagate through the tree. The CRime `send` operation is more complex and incurs higher processing time mostly for the following two reasons. First, the operation is typically sent over one stack. This means that some checking needs to be done so that the operation does propagate on the path of the tree corresponding to that stack and not over the entire tree. Second, this operation needs to handle triggers, for which additional instructions need to be executed.

The total amount of time needed by Rime to execute the sequence of operations `open`→`send`→`recv`→`close` is $\sim 256 \mu$ s. CRime needs 622 μ s for the same sequence; this is an execution time which is a factor of ~ 2.4 higher.

Next, we take a closer look at CRime and the overheads introduced by the amodule and stack abstractions in terms of processing speed. Table 14 lists the name of the four basic operations in the first column and the overall duration in the second column. In the third column, the names of the CRime functions called in order to execute the functionality of the `c_abc` stack are listed. For instance, for the `open` operation, the sequence of functions `c_abc_open` and `c_channel_open` are called and their cumulative duration is 73 μ s, as listed in the fourth column. The last column lists the overhead introduced by the amodule and stack abstractions (i.e. walking through the tree,

performing checks, etc.). This last column is the difference between the second and the fourth, namely between the duration of the overall operation and the duration of the executed functions.

Table 14: The cost of stack and module abstractions in terms of processing speed.

CRime operations					
Name of operation	Duration of the overall operation [μs]	Name of the executed functions	Duration of the executed functions [μs]	A module and Stack overhead [μs]	[%]
open	107.0	c_abc_open, c_channel_open	73.0	34.0	31
send	380.0	c_abc_send, c_rime_output	76.0	304.0	80
recv	96.5	c_abc_recv, c_abc_input	12.0	84.5	87
close	67.0	c_abc_close, c_channel_close	26.0	41.0	61

It can be clearly seen in this breakdown in Table 14 that the overhead for the send operation where checks for walking through the tree and trigger handling are quite high, representing 80 % of the total duration of the operation (304 μs out of 380 μs). The overheads for receiving a packet and closing a connection are also high in relative terms representing 87 % and 61 % of the total duration of the respective operations. The smallest relative overhead occurs for the open operation, representing 31 % of the total duration. The differences between the overheads of the four operations are justified by the different implementations of the tree walking algorithms (for the send operation the trigger handling also has to be accounted for).

4.3.4 Power consumption

In order to evaluate the cost of CRime in terms of energy consumption, we performed measurements using a Tektronix TDS5104B oscilloscope. We measured the power consumption of representative Rime and equivalent CRime applications. Each application was run 10 times for 100 ms and the results are summarized in Table 15.

It can be seen that on average, CRime consumes 1.6 % more energy than Rime. In other words, if a battery powered node could run for 365 days sending messages using the Rime stack, the same node could only run for about 360 days sending messages with the CRime stack.

Table 15: CRime vs Rime energy consumption.

Rime power consumption		CRime power consumption		$\Delta P/(P_R)$
Application name:	Consumed power (P_R) [mW]	Application name:	Consumed power (P_{CR}) [mW]	
example_abc	89.96	(c_abc)	91.84	0.020
example_broadcast	88.96	(c_broadcast)	90.21	0.014
example_polite	88.70	(c_polite)	89.58	0.010
example_unicast	90.09	(c_unicast)	91.84	0.019
example_multihop	91.84	(c_multihop)	93.59	0.019

4.4 Summary

This chapter introduced Composeable Rime (CRime), an architecture for dynamic composition of communication services based on the existing Rime architecture. The abstractions used by CRime and their implementation were first discussed. This was then followed by an architectural comparison with Rime. Finally, a quantitative comparison between the two implementations with respect to memory footprint, processing speed and energy consumption was provided. This way, the overhead introduced by modularity and composeability was assessed.

The next chapter will focus on the other key components of the framework: the declarative language and the workbench. The requirements and design choices for the declarative language will be first introduced. Then, a domain ontology for the CRime module library will be proposed. The workbench that is tightly coupled with the declarative language will be then described. Finally, the potential of the ProtoStack tool is by collecting will be assessed feedback from a group of first time users.

5 The ProtoStack declarative language and the Workbench

ProtoStack requires a level of abstraction between the workbench and the module library and, similar to other systems [2][3], it uses a declarative language for this purpose. Typically, such a language is used for configuration purposes. ProtoStack, besides using the declarative language for configuration purposes, also uses it for additional two tasks. First, the language is used for validity checking as discussed in this chapter. Second, the language is used for knowledge representation and reasoning to support service oriented networks (SONs) [37] as defined in Section 2.2 and further discussed in Chapter 6, and cognitive networks [17] as discussed in Chapter 7.

5.1 The ProtoStack declarative language

The ProtoStack declarative language represents an abstraction layer between the workbench and the CRime library. Besides being used for the validity checking of a manually composed stack of communication services and interoperability between testbeds, it is intended to enable and support emergent research on logic based composition and configuration of networks particularly for research in the areas of service oriented networks and cognitive networks. In order to achieve these goals, a set of requirements for the declarative language is presented in this section, followed by the design decisions including a discussion of trade-offs.

5.1.1 Requirements for the declarative language

Upon deciding on a suitable declarative language to be used for ProtoStack, as the reference implementation of the proposed framework, the following requirements were considered:

- **Simplicity:** the proposed framework for dynamic composition of communication protocols and services is targeting in communication networks and technologies. This target community is typically accustomed to imperative programming (C, C++, Java, Python, Matlab) and it can be more difficult to start using complicated declarative language (Prolog).
- **Machine interpretable:** the declarative language should be machine readable in order to facilitate easy manipulation by machines.
- **Standardized:** the purpose of the language within the proposed framework is to provide a necessary layer of abstraction and to support domain specific research

challenges; therefore a relatively widely adopted, open and stable standardized approach is preferred to a less stable and potentially proprietary approach.

- **Interoperability:** the language should be designed to facilitate the interoperability of systems so that potential reference implementations of the framework can be easily integrated at this level of abstraction. Such approach can lead to a large scale formalized representation of communication systems.
- **Support for knowledge representation and logic reasoning:** besides its role as an abstraction layer in the proposed framework, the declarative language should also support emerging logical reasoning for self-configuration of communication networks. Two relevant research areas where this applies are service oriented networks and cognitive networks as further discussed in Chapters 6 and 7.

5.1.2 Design decisions related to the declarative language

The requirement that probably restricts most of the search space for a suitable declarative language is the last one referring to support for knowledge representation and logic reasoning. Knowledge can be represented in various forms, using very expressive languages such as the natural language or less expressive, artificial languages such as algebraic representations. Recent efforts in creating expressive artificial languages for machine interpretability resulted in languages such as the Resource Description Framework (RDF) [32], the Web Ontology Language (OWL) [33] and CycL [33]. RDF is the least expressive of the three; OWL is a restricted but more expressive form of RDF, while CycL is the most expressive.

Knowledge represented using these languages can be queried and the complexity of the queries is directly related to the expressiveness of the representation language. Knowledge bases using RDF support relatively simple queries via SPARQL; the ones using OWL support reasoning with first order logic, while the ones using CycL support second and higher order reasoning. The trade-off for expressiveness is the complexity of the reasoning engine and the time needed to deduce the result.

RDF and OWL are more appropriate for encoding knowledge which has to be transported between systems as can be seen from most application areas, including the IEEE 802.21 specification. In IEEE 802.21, for instance RDF is used for encoding location based knowledge that is transported between the Media Independent Information Service residing on the mobile terminal and its peer in the service provider network [35]. Both RDF and OWL are standard languages used in the semantic web. RDF is the simpler of the two while also supporting a SPARQL based inference mechanism and a simplified syntax called Turtle, which prevailed in the selection procedure of the declarative language to be used for ProtoStack.

In summary, the ProtoStack declarative language uses the RDF data model; the custom vocabulary built by creating the CRime ontology and the Turtle syntax which is human

readable and can easily be transformed in XML if needed. The RDF data model consists of a triple subject-predicate-object model as shown in Table 16. Some keywords are part of pre-existing standard vocabulary such as `subClassOf` which is a property defined by the W3C RDF Schema. Two example statements in standardized form with the preceding namespace using Turtle representation are provided in the table, the first one says that “`c_abc` is of type module” and the second says that “`c_open` is a sub class of the class function”.

Table 16: The ProtoStack language model.

Subject (Resource)	Predicate (Property)	Object (Statement)
<code>crime:c_abc</code>	<code>rdf:type</code>	<code>cpan:Module</code>
<code>crime:c_open</code>	<code>rdfs:subClassOf</code>	<code>crime:Function</code>

5.2 The CRime ontology

Standardized knowledge representation languages typically use a common vocabulary. Often this vocabulary takes the form of an ontology which in addition to common terms also provides relationships between these terms. Many ontologies have been developed that are modelling different aspects of the world, while specific and detailed ontologies modelling aspects of communication networks are rare and typically very narrow for instance covering a very specific aspect of the network such as security [36]. Since ProtoStack itself requires a very specific world model related to CRime enabled networks, the creation of a custom ontology was necessary. In this section we discuss the CRime ontology as needed by the tool while in Chapters 6 and 7 we look at how this ontology is used and what extensions might be needed for some use cases.

The CRime ontology (see Appendix D) is built to help describe the conceptual model behind CRime. It can easily be extended for future use. The ontology consists of two levels of classes as depicted in Figure 12 a. The first level contains the generic classes such as Module, Interface, Function, Parameter and Scope. The second level consists of subclasses of the Function concept, also referred to as primitives (i.e. from primitive functions) in this work. These correspond to the 11 generic primitives defined by CRime amodules. We also defined 9 properties to form statements about CRime (see Figure 12 b).

In the following we illustrate on the `c_abc` example how the ontology can be used to describe the CRime world model. The `c_abc` instance is an implementation of an amodule in the CRime parlance and it represents an individual (or instance) of the module class in the CRime ontology (see **Error! Reference source not found.** a). `C_abc` defines 5 of the 11 primitives from the amodule. It has a singlehop scope as it sends packets only to direct neighbors and it implements top and bottom interfaces through which it interacts with

adjacent modules above and below.

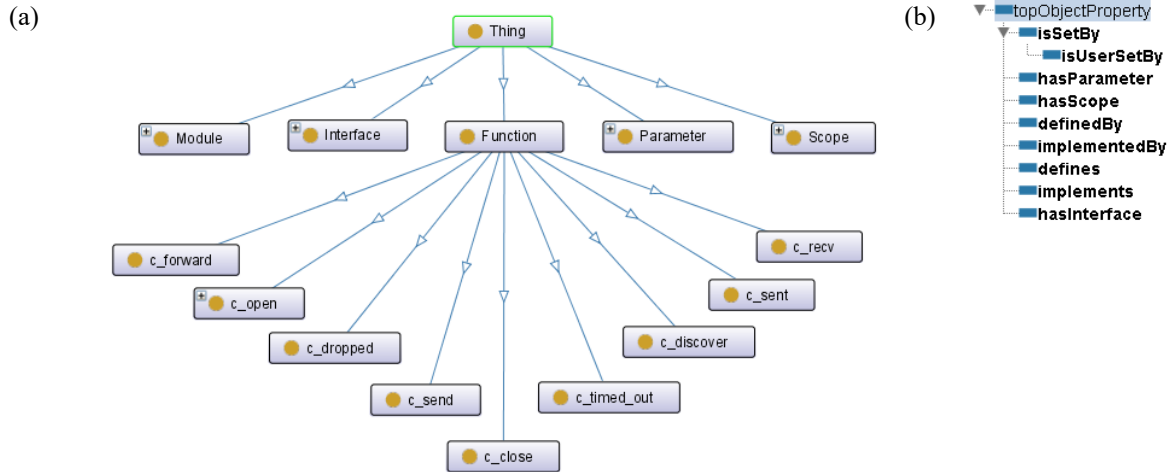


Figure 12 CRime ontology (a) classes, (b) properties.

Finally, it uses some parameters such as `channel_no`. All this is stated in the graph depicted by Figure 13b. By further expanding the graph, it can be seen in Figure 13c that the `c_abc_open` primitive, which is defined by the `c_abc` module, is an instance of the `c_open` class of the ontology and that it is implemented in both bottom and top interfaces.

We used the Protege ontology editing tool⁹ for creating the CRime ontology and for the visualizations presented in this section.

5.3 ProtoStack configuration steps

The ProtoStack configuration consists of four steps further explained in the following subsections:

- fact specification, translation and storage,
- workbench rendering,
- manual stack composition,
- validity checking and code generation.

5.3.1 Fact specification, translation and storage

ProtoStack assumes that the module developer, besides writing the C code necessary for the module, also provides the description of the module in the ProtoStack language (i.e. annotates the module). This description consists of a sequence of Turtle sentences using the CRime ontology as vocabulary. The sentences are assumed to be written as comments in the header file corresponding to the module.

⁹ <http://protege.stanford.edu/>

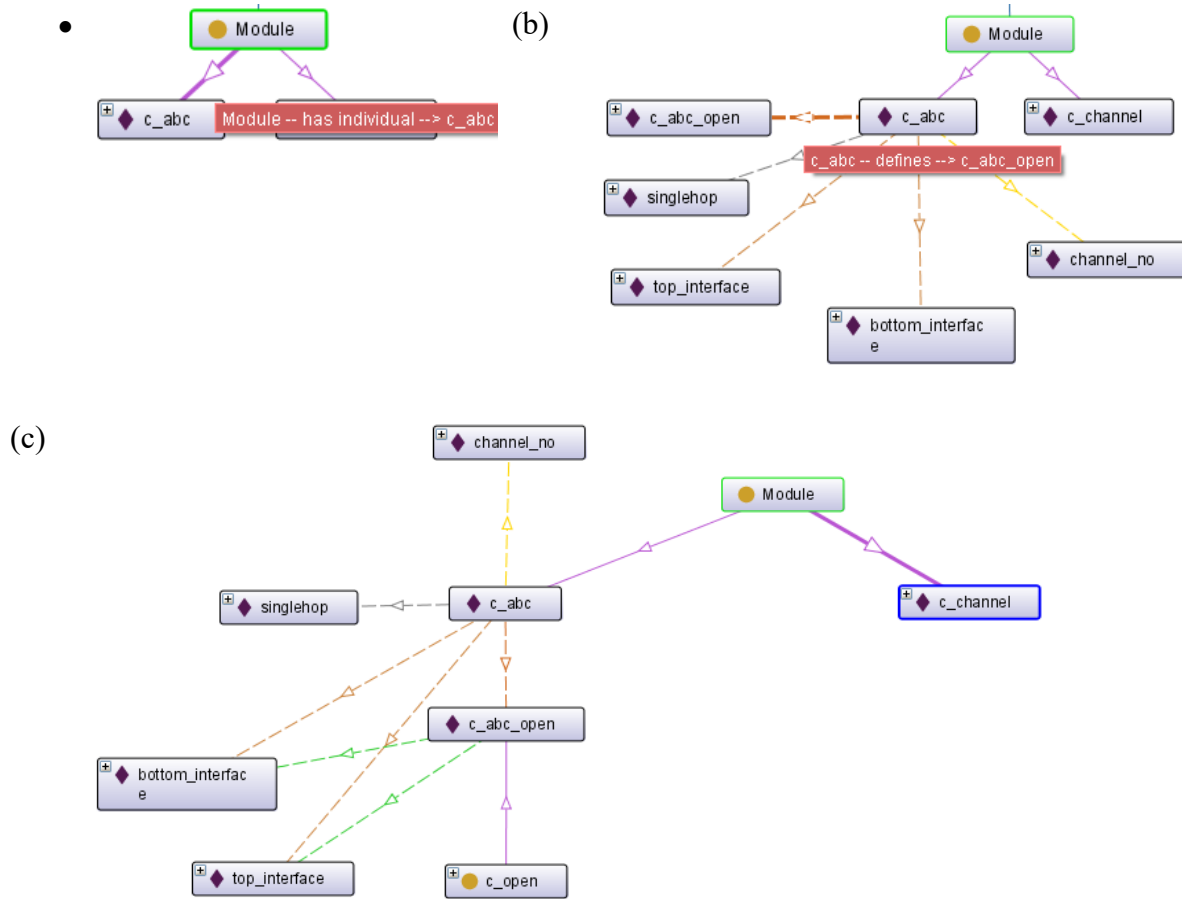


Figure 13 Example CRime statements describing the *c_abc* module.

Figure 14 lists example statements describing the *c_abc* module (the full description has 46 statements and can be seen in Appendix E). The server parses the statements in each CRime header file and inserts them in a specialized storage called triple store. We use the Sesame triple store and a custom Java implementation based on the Jetty server for this.

The first line in Figure 14 states that *c_abc* is a module (i.e. of type module) and the second line states that *c_abc*'s scope is single-hop (i.e. provides single-hop communication functionality). Subsequent lines state that *c_abc_open* is of type open, that *c_abc_open* implements bottom and top interfaces and that *c_abc* defines *c_abc_open*.

```

1. //turtle crime:c_abc rdf:type cpan:Module .
2. //turtle crime:c_abc crime:hasScope crime:singlehop .
3. //turtle crime:c_abc_open rdf:type crime:c_open .
4. //turtle crime:c_abc_open crime:implements crime:top_interface .
5. //turtle crime:c_abc_open crime:implements crime:bottom_interface .
6. //turtle crime:c_abc crime:defines crime:c_abc_open .

```

Figure 14 Example Turtle statements describing the *c_abc* module.

The system knows that `c_open` is a primitive function and that functions implement interfaces. It also knows that modules have two interfaces: top and bottom. All these relationships between the concepts are modelled in the CRime ontology. The Turtle notation in the header files specifies instances of the concepts present in the ontology and relationships between these instances. All the Turtle statements from the CRime modules are stored in a triple store and internally represented as a graph as shown in Figure 15 for the `c_abc` module. The nodes of the graph are either concepts (classes) or instances (individuals). Concepts such as module and function are depicted with yellow in the graph, while instances such as `c_abc` and `c_abc_open` are depicted with violet. The edges of the graphs represent the relationships between the nodes, and each colour corresponds to one relationship as can be seen in the legend of the figure. For instance, the blue edges connecting function with `c_open`, `c_close`, `c_send`, `c_rcv`, `c_sent` as well as the ones connecting module, scope and interface to thing represent the `subClassOf` relationship.

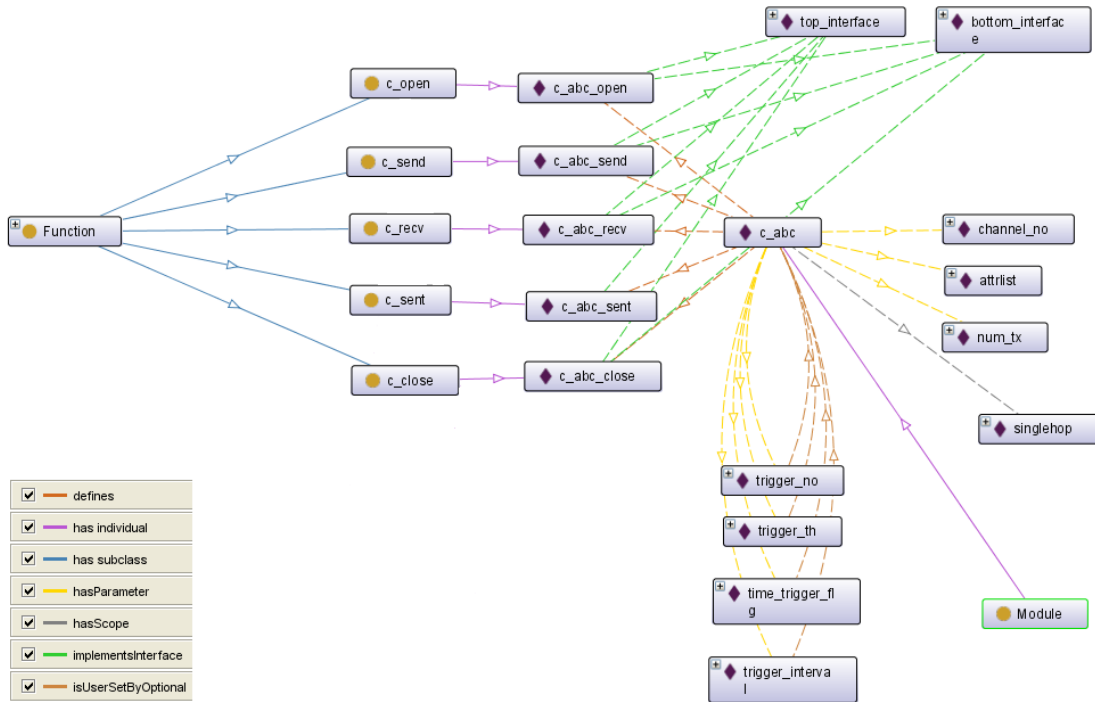


Figure 15 Snapshot of the RDF graph describing the `c_abc` module.

5.3.2 Workbench rendering

The workbench is a graphical user interface which facilitates drag and drop style composition of a protocol stack. Such component is useful for users, particularly to those accustomed with simulation environments, most of which feature such a component. The workbench features a horizontal menu bar and three vertical work areas, as depicted in Figure 16. The menu bar features buttons corresponding to operations that can be performed with the stack. The first area consists of the toolbox where the CRime modules are listed. The middle area is occupied by a panel where the modules from the toolbox

can be dragged and dropped, wired and configured in a stack. The third area consists of the properties bar where additional description of the stack can be provided.

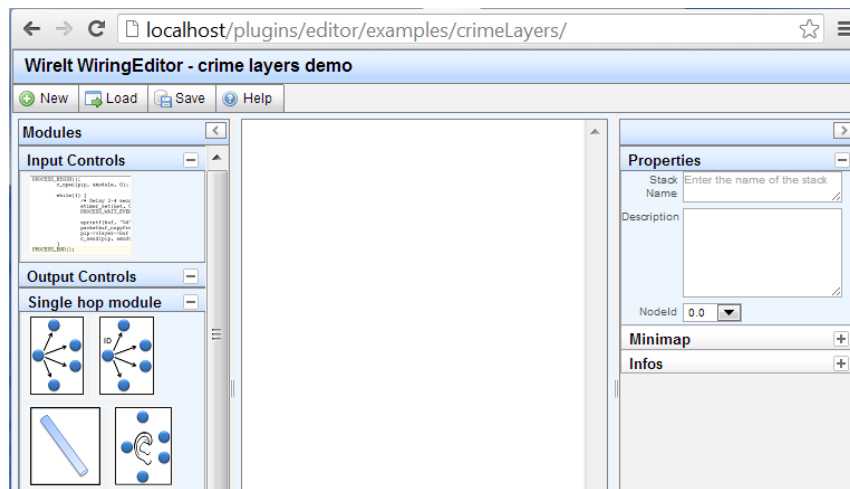


Figure 16 Snapshot of the ProtoStack workbench.

When the user opens the browser, a request is sent to the ProtoStack server for the list of available modules and their corresponding parameters. The server receives the request and interrogates the triple store (described in Section 5.3.1) for the information needed to populate the toolbox in order to render the workbench. ProtoStack uses the RDF graph from the triple store to automatically populate the toolbox situated on the left side of the workbench. This is achieved by querying the store using the SPARQL query language which performs template matching against the RDF graph. The system performs incremental querying. First it asks for all the available modules and their descriptions (see Figure 17 line 7). Then, it asks for the scope and the parameters corresponding to each module that must be inserted by the user (Figure 17 line 8).

The response is generated based on the responses to the SPARQL queries and is returned to the client in the form of a JSON message. All the available modules and the corresponding parameters are then rendered in the toolbox. These can be dragged to the panel and configured in a stack as illustrated in Figure 18. Once the stack is built and configured, the corresponding data is sent to the server on user command.

5.3.1 Manual stack composition

In order to compose a simple stack, the user first loads the workbench in the browser and gets the available modules and their descriptions presented in the toolbox. Then the user selects the desired modules and drags them one by one to the panel, wires them and then fills in the values of the parameters. The name of the stack needs to be provided in the “Stack name” box on the right hand side of the workbench. An optional description of the stack can also be provided. When the stack configuration process is completed, the user submits the new stack to the server.

```

1. PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2. PREFIX cpan: <http://downlode.org/rdf/cpan/0.1/cpan.rdf#>
3. PREFIX crime: <http://sensors.ijs.si/crime#>
4. PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
5.   SELECT ?name ?category ?description
6.   WHERE {
7.       ?name rdf:type cpan:Module .
8.       ?name crime:hasScope ?category .
9.       ?name rdfs:comment ?description .
10.  }

```

Figure 17 SPARQL query which retrieves the modules and their descriptions for populating the toolbox of the workbench.

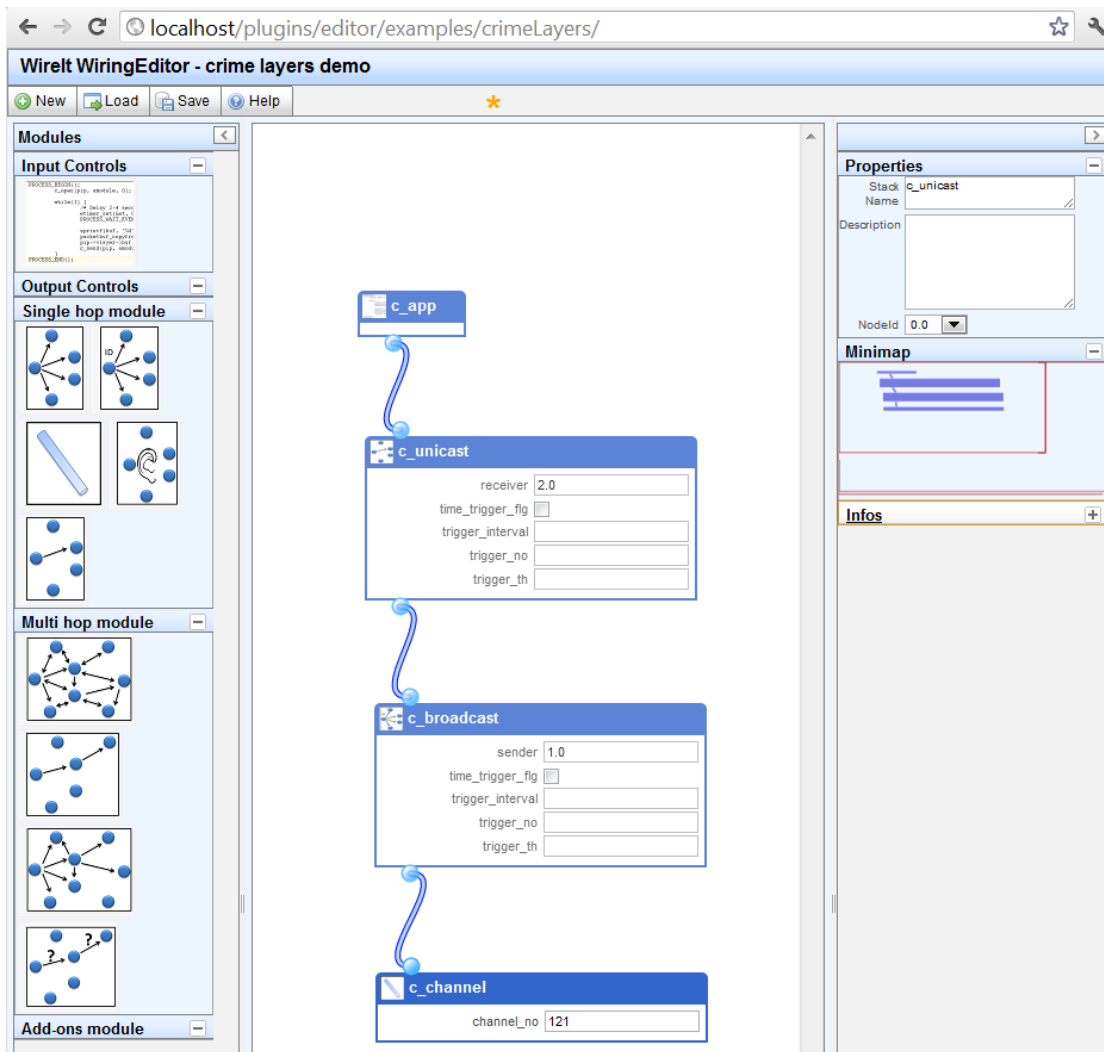


Figure 18 Example of a 1 channel – 1 stack configuration using the ProtoStack workbench.

When the server receives a new stack, it parses the JSON configuration and performs consistency checking. In case of any inconsistency, the user is prompted, using meaningful error messages which help with convergence towards a valid configuration. Upon successful validation, the corresponding C code is generated, the binary image compiled and then loaded to the node.

5.3.1.1 Example of 1 channel – 1 stack configuration

Figure 18 depicts an example of a correct 1 channel – 1 stack configuration. The stack is a unicast stack formed of a beginning module – symbolically named `c_app` – and an ending module symbolically named as `c_channel`. In between the `c_broadcast` and `c_unicast` modules are placed. The composition is not complete before the modules are properly wired.

The depicted stack initializes the sensor node with Rime address 1.0 (see the sender field in the `c_broadcast` module) and identifies the node with Rime address 2.0 as the recipient of the messages (see the receiver field in the `c_unicast` module). The communication will be carried out on channel 121 as configured by the `channel_no` parameter of the `c_channel` module. Triggers are not activated for either of the modules in Figure 18 (`time_trigger_flg` is not selected), therefore the trigger parameters are not required. The same stack with the triggering mechanism activated is depicted in Figure 19. In the example, the unicast module's sending function is invoked each 60 seconds, therefore the stack behaves as the stunicast stack from Rime.

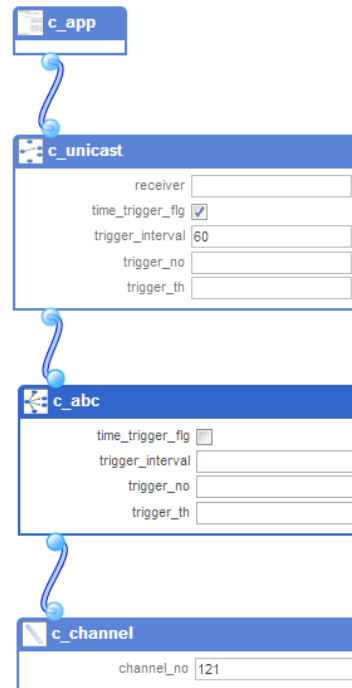


Figure 19 Example of a stack which uses the triggering mechanism.

5.3.1.2 Example of n channel – n stack configuration

Complex stacks are CRime n channel – n stack models which use n channels for communication. The steps for composing and configuring it are the same as mentioned in the previous section. Figure 20 depicts a correct 3 channel – 3 stack communication built for a mesh application. When composing complex stacks, the system assumes that the main stack – the communication or data stack – is situated at the leftmost, while the management stacks follow towards the right. If this assumption is violated, the stack is

not going to work as expected. In the `c_mesh` example, the data stack operated on channel 132, while the route request and route reply functions are implemented by the next two stacks on channels 133 and 134, respectively. When such a stack configuration is submitted, the server parses the tree from left to right and from bottom upwards. The order in which the modules and the wires are added is irrelevant as long as the left to right condition is fulfilled. When parsing the tree, the server determines the split nodes (`c_mesh` and `c_route_discovery` in our examples) and generates the appropriate C configuration code.

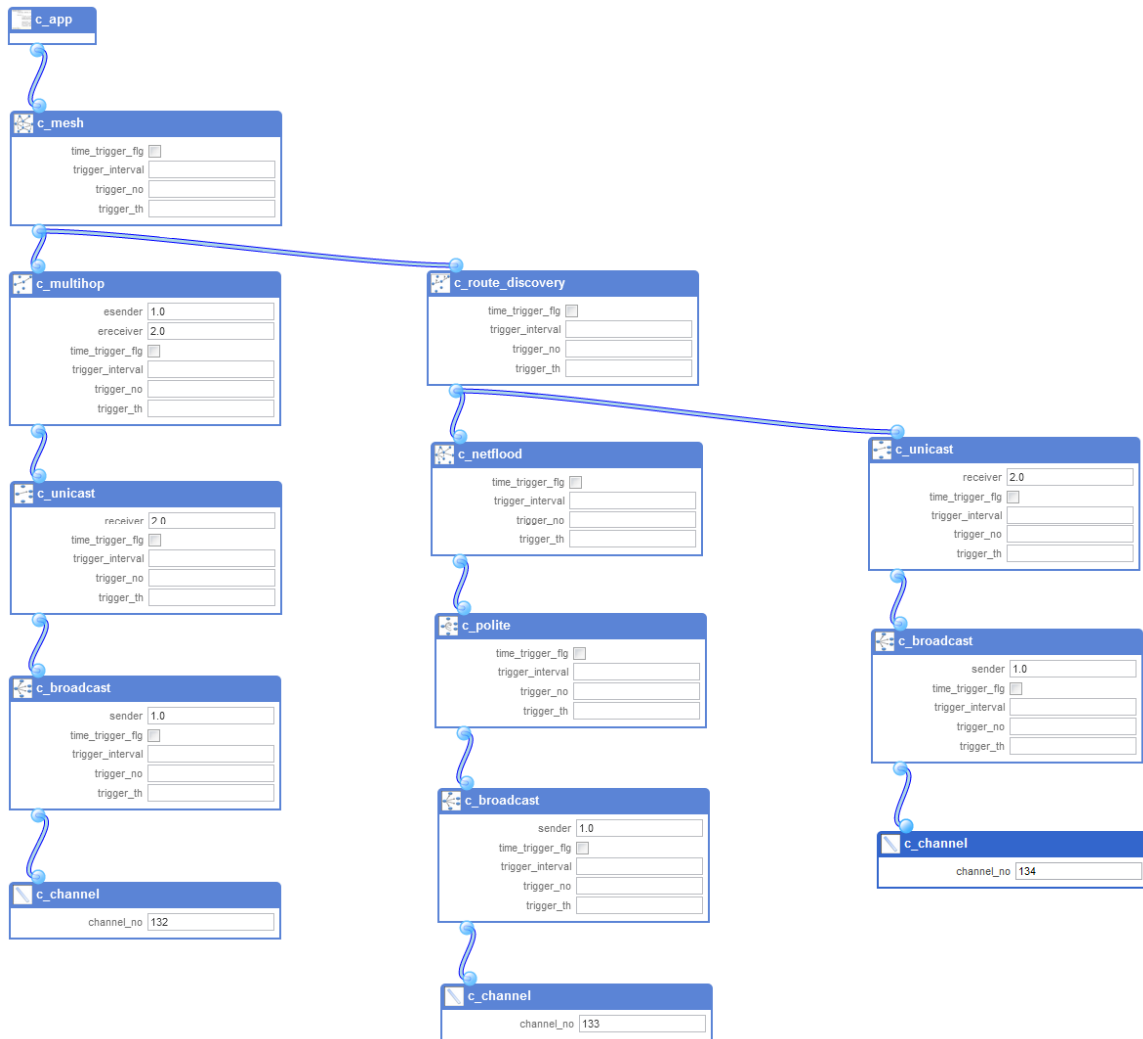


Figure 20 Example of 3 channel – 3 stack configuration using the ProtoStack workbench.

5.3.2 Validity checking and code generation

By validity checking we refer to the validity of the composed stack. As discussed in this chapter, each module defines a set of functions and uses a set of parameters. However, the user is not able to see the functions and can see only the parameters that need to be provided. This is a trade-off between simplicity and completeness. If we render all functions for a module on top and bottom interfaces (i.e. minimum 4 functions per

interface), this would require the user to drag in excess of 4 wires between each two modules and also make sure the wires are connected to the correct terminals. We consider this to be less user friendly, therefore we hide it and we perform instead validity checking on the server side.

To prevent invalid sequences of modules being stacked and wired together, ProtoStack performs validity checking in two steps: pre-rendering checks and post-rendering checks. Before rendering the tools in the toolbox, the workbench requests from the server configuration parameters for each module. The server generates these based on the knowledge inserted in the triple store. For validity checking, two pieces of information are important: the connectors for each module and the required user defined parameters.

To simplify the module composition, we defined a top and a bottom interface for each module. Intuitively, the top interface of module N can only be connected to the bottom interface of module $N+1$ in the sequence (see Figure 21). Similarly, the bottom interface of module N can only be connected to the top interface of module $N-1$. The module developer inserts Turtle statements specifying which primitive function is defined by which interface. The developer also defines the parameters which are used by a module and the subset which needs to be provided by the user. This information is used by the workbench to generate appropriate boxes for each module and prevents connecting (wiring) incompatible interfaces (i.e. top to top and bottom to bottom) or submitting empty user defined parameters.

When the user submits a composition of modules to the server, a more fine grained validity check is performed at interface level. For each pair of connected interfaces $\langle N-1 \text{ top}, N \text{ bottom} \rangle$, $\langle N \text{ top}, N+1 \text{ bottom} \rangle$, a check for the overlap in implemented primitives is performed. For instance, if $N+1$ bottom uses the `c_open` primitive to request the opening of a connection and N top does not support that primitive, then the composition is invalid and the appropriate message is conveyed to the user. If we consider the example in Figure 21, then if the broadcast module's bottom interface would not define the `c_broadcast_open` function, then there would be no way to process `c_abc` module's sent notification. In such cases, the stack is declared invalid and the user is prompted accordingly.

A stack is valid if the vector of functions defined by the N -th module's top interface is included in the vector of functions defined by the $N+1$ -th module's bottom interface as follows:

$$\begin{aligned} I_{N,t} &= \{f_1, f_2, \dots, f_n\}, f_i - \text{Function} \\ I_{N+1,b} &= \{f_1, f_2, \dots, f_m\}, f_i - \text{Function} \\ I_{N,t} &\subseteq I_{N+1,b} \end{aligned}$$

Figure 21 Validity checking example on a three module stack: making sure that the top interfaces of lower modules need to be compatible with the bottom interfaces of higher modules.

5.4 Reusability

The ProtoStack language is custom made for the CRime set of modules, however, it is not limited to it. It can be re-used by any other modular stack library and any other framework for stack composition. For this, the explained ontology would need to be extended or a project specific ontology would need to be created. Then, the server side logic would need to be customized for that particular stack (i.e. we assume that the

configuration code and the abstractions in other stacks are significantly different than the ones in CRime).

5.5 Reproducibility of the experiment

Reproducibility of the experiments is a key research requirement; however, it is currently ignored in several research areas, including sensor networks and cognitive (radio) networks. Much of the published work includes insufficient information to reproduce the simulations and their findings. Furthermore, a relatively small number of authors verify their findings in a realistic scenario, such as a physical testbed. The issues of reproducibility and benchmarking [39] are addressed by several projects including the Cognitive Radio Experimentation World [30].

The ProtoStack system enables easy reproducibility of experiments and can provide support to benchmarking frameworks. The system is able to save all the composed stacks and their corresponding configurations. These can be used at a later time to exactly reproduce the experiments on the same or on another testbed supporting CRime. The saved configuration can then be used as an input to a benchmarking system.

5.6 Support for remote experimentation

The ProtoStack system uses a web based client-server architecture. As such, it can be used for remote experimentation once it is integrated with the components of the experimental infrastructure provider. The experimenter logs in to the portal provided by the experimental infrastructure provider, sets up the experiment and deploys it. Reports on the progress of the experiment, as well as the results need to be provided by other components of the portal, as ProtoStack currently does not support such functionality.

5.7 On the usability of ProtoStack

The workbench and the CRime module library are part of the ProtoStack tool. The first is meant to make its use more user friendly by providing a composition and configuration environment similar to those provided by simulation tools. Further, the workbench allows module developers and experimenters to focus on stack composition rather than on coding and debugging. The second is developed to support dynamic composition and configuration of protocol stacks and the degree to which this has been achieved directly reflects in the workbench. For instance, the toolbar is a direct reflection of the CRime modules. To assess the degree to which ProtoStack supports stack composition and collect feedback on how to improve the overall tool, we performed a study by collecting feedback from a group of first time users.

5.7.1 Methodology

In order to collect feedback about the usability of ProtoStack we decided to interview potential users of the tool. In doing so, we were faced with two challenges:

- **How to determine the target users?** ProtoStack is primarily targeted at the research community, particularly at researchers interested in one or more of the following areas: sensor network protocols, modular protocol stacks, service oriented networks, cognitive networks. Among researchers working in these or related areas those that were already using the Contiki OS in their experiments can be considered as advanced users.
- **How to perform a statistically significant study?** The research community we are targeting is relatively small with probably no more than 5000 people¹⁰. To perform a statistically significant study, we would need to identify interested researchers from the research areas mentioned above and incentivize them to use and provide feedback for the tool, at the same time taking care for statistical significance also in terms of their actual experience level, background knowledge, and other parameters that might influence the opinion. Doing this would prove tedious task with questionable rate and quality of feedbacks, especially if carried out remotely by some automated means, so we intentionally opted for what we refer to as representative and not statistically significant study.

To address these challenges, collect feedback and also disseminate the tool in relevant communities, we decided on the following strategy:

- **Collect explicit and implicit feedback.**
By collecting explicit feedback we refer to having the users of the tool fill in a relevant questionnaire. By relevant, we refer to relevancy with respect to the research interests of the target groups. Therefore, the content of the questionnaire may change from group to group and will also be influenced by the stage of development of the tool. Explicit feedback may not be enough in some cases. For instance, it is hard for a person to monitor the time she spent on a task and all the issues that she may encounter. Such data is implicitly collected by the examiner in the case of interview-like feedback collection or automatically in the case of group hand-on like feedback collection.
- **Keep the questionnaires to a minimum.** The questionnaires should be kept short to minimize the chance that people get annoyed and refuse to fill them.
- **Perform progressive feedback collection.** Performing large one-time feedback

¹⁰ A typical high profile sensor networks conference attracts between 100 and 200 people. According to Google Scholar, the number of citations attracted by the paper introducing the Click modular router is ~2000. According to the same source, the number of citations attracted by the paper introducing Contiki OS since 2004 is ~1000. The seminal work on cognitive networks has attracted ~500 citations since 2005. The available download figures for the Contiki OS are ~ 300 per week, including commercial users.

collection in the target community is infeasible for obvious reasons. Therefore our approach is to perform progressive feedback collection. The first step is to collect feedback from the members of our lab. The next step is to collect feedback from project partners through dedicated events such as training days or summer schools. Further steps are planned to involve hands-on tutorials at relevant events.

- **Persist with dissemination.** Disseminate the framework and the tool through available channels such as papers, tutorials, videos and open code¹¹.

5.7.2 The collected feedback

The first step in our progressive feedback collection involved preparing relevant questionnaires and performing interviews with the members of our lab¹². Based on the experience from this step, the next step, involving project partners and interested external users is under preparation¹³.

The questionnaire in the first step consisted of two parts. The first part included a set of 12 questions meant to profile the user and his/her background experience. In the second part of the questionnaire, an introduction of the tool is provided, a training example, two tasks and 10 questions meant to help evaluating and improving the ProtoStack tool. The training example was performed by the developer of the tool for each user in particular. The developer also had to answer a set of three questions about the way the user was using the tool. The filled questionnaires are provided in Appendix F.

The study group consisted of six users aged 23-30. The users completed BSc or MSc in a technical domain ranging from computer science to electronics. All users were working in research, one is just starting, most have 2 years' experience and one has 5 years' experience. Their research interest varied from cognitive radio to semantic sensor web and most of them were familiar with basic communication primitives as can be seen from Table 17. Most of the users had some previous experience with the Contiki OS and with the Rime and TCP/IP protocol stack. Most of them were not familiar with the concept of “modular stacks” and only half of them are familiar with the concept of “service composition” as shown in the results summarized in Table 18.

¹¹ <https://github.com/sensorlab>

¹² Note that the subjects were not under pressure to favorably rate the tool by means of upcoming evaluations that could affect marks or pay.

¹³ The second step is planned for February 20th, 2013 at CREW training days in Brussels (<http://www.crew-project.eu/trainingdays>)

Table 17: The study group profile: in terms of education, research interest and familiarity with basic communication primitives.

User ID	Age	Education	Research interest	Familiarity with: broadcast, unicast, multihop, mesh
1	25	MSc Computer Sci.	Semantic web, semantic sensor web	yes, yes, yes, maybe
2	24	BSc Telecomm.	Cognitive radio, spectrum sensing	yes, yes, yes, yes
3	29	BSc Electronics	Cognitive radio	yes, yes, yes, yes
4	28	BSc Telecomm.	Web technologies	yes, yes, yes, yes
5	23	BSc. Information Tech.	-	yes, yes, yes, yes
6	30	BSc Telecomm	Cognitive networks	yes, yes, yes, yes

Table 18: The study group profile: in terms of experience with Contiki, Rime, TCP/IP, modular stacks and composition of services.

UserID	Contiki exp.	Rime exp.	TCP/IP	Modular stack	Composition of services
1	no	no	yes	no	no
2	yes	yes	yes	no	yes
3	yes	no	yes	no	no
4	yes	yes	yes	yes	yes
5	yes	yes	yes	no	no
6	yes	yes	yes	yes	yes

All users found the GUI sufficiently intuitive, mostly rating it with 4 points (out of 5). Suggestions for improvement mostly referred to adding extra features such as the possibility to download the configuration code and more pop-ups following interactions with the GUI. The toolbar was rated with 5 points (out of 5) for intuitiveness by all the users. The description provided for the modules seems to be sufficient for some users and requires more details for others. It seems that users which have more experience with Contiki feel that there should be available also an extended description on the right hand side of the workbench where space is currently used in a less efficient way. The tool was confirmed to save time. According to the study it seems that users that have previously used Contiki in their research work estimate that ProtoStack can speed up the design of and experimentation with protocol stack by at least a factor of two. It also looks like the tool increases flexibility with respect to the baseline Rime. All these results are summarized in Table 19.

Table 19: User rating of the ProtoStack tool in terms of intuitiveness, completeness and usefulness.

User ID	Intuitiveness of the GUI (1-5)	Intuitiveness of the toolbar (1-5)	Description of modules	Estimated percentage of saved time	Flexibility
1	4	5	5	-	-
2	4	5	3	50-70%	yes
3	4	-	-	-	yes
4	5	5	5	50%-90%	yes
5	4	5	4	50%	yes
6	4	5	4	50%-90%	yes

The users were observed performing the two tasks from the questionnaire. The time to completion as well as the number of failed saving attempts of a new stack were collected and are summarized in Table 20. It seems that both stacks were fast to complete. The average completion time for the first task was 2.66 minutes with less than 1 error message. The average completion time for the second task was slightly lower at 2.1 minutes with a slightly higher number of error messages. The complexity of the second task was slightly higher than of the first one, with more degrees of freedom. While some users were inspired by this to create novel stacks that were valid, other users had difficulty creating valid stacks. These users, for instance, found it unclear why in the case of a unicast stack, a broadcast module identifying the sender is also needed. This problem seems to be correlated with the experience one has with communication stacks – the more experience, the less problems in composing the stack.

Table 20: Time spent on tasks and numbers of failed attempts to save the stack by the study group.

UserID	Task 1		Task 2	
	Completion time (min)	Error no	Completion time (min)	Error no
1	2	1	5	2
2	3	1	4	1
3	2	0	2	1
4	3	2	1	1
5	5	1	2	2
6	1	0	1	0

5.8 Summary

This chapter was concerned with the declarative language and the workbench components of the framework for dynamic composition of communication services. The requirements

and design choices for the declarative language were first identified. Then, the syntax of the language was described and a domain ontology for the CRime module library was introduced. We then showed how the language supports automatic workbench population and validity checking. We argued that this language is independent of the module library and that the workbench can be adapted to other systems. We also showed how it supports repeatable experimentation in terms of the CREW project. We concluded the chapter with a feedback collection from first time users assessing the ease of use of the ProtoStack experimentation tool.

While Chapters 4 and 5 focused on the components of the framework for the composition of communication services, Chapters 6 and 7 will show how promising research areas can benefit from the framework. The next chapter introduces and discusses Service Oriented Networks and their similarities with the Open Systems Interconnect Architecture. The concept is then illustrated on a use case and three steps in realizing it are identified. A discussion on the representation languages and logic based reasoning for SONS is then followed by implementations using ProtoStack. We explain the increasing complexity required from the representation language and reasoning process for achieving various degrees of automation for SONS.

6 Dynamic composition of services for Service Oriented Networks

Similar principles as addressed in protocol stack composition can also be extended to larger scale problems such as the provision and maintenance of end-to-end Quality of Service (QoS). In this chapter, we discuss end-to-end service composition for information transport based on principles from Service Oriented Architecture (SOA), i.e. dynamically composing communication services with characteristics matching the requirements of a given application. Then we show how services can be composed using the ProtoStack tool using knowledge representation and logic based configuration of networks by discussing and illustrating the concept on service discovery and composition for CRime services using the ProtoStack declarative language. We also identify and discuss related open research directions.

6.1 SOA and OSI

SOA principles are similar to those behind the division of labour, and therefore should incentivize providers to adopt them to make better use of resources and increase revenue. As we show, some SOA concepts are similar to concepts in the Open Systems Interconnection (OSI) reference model, with the key advantages of dynamism and flexibility. In this section, we compare the SOA and the OSI architectures and discuss similarities and differences between the two, synthesizing the logical components of both architectures in Table 21. With this comparison we want to show that SOA principles have been present in communication networks for a while and that their implementation becomes more advanced as technology improves, evolving from communication services over service independent platforms to web service technologies [41].

In SOA, services are software resources, each service having its distinct functional context [42]. A combination of these services can be dynamically composed in a composite service resulting in added value functionality. In OSI, services also have distinct functional roles, each service being a set of primitives that provides communication functionality in order to interconnect open (standards compliant) systems. These primitives can make use of both software and hardware resources, making the definition of a service in OSI broader. On the other hand, the purpose of a service in SOA is to provide business functionality [42][43], and therefore is broader than in OSI. Furthermore, in SOA, services have three essential properties: they are self-contained, platform independent and can be dynamically located, invoked and recombined. In OSI,

however, services are statically defined and invoked and cannot be recombined. Recent research in dynamically composing protocol stacks and layers can be seen as an attempt to apply SOA principles to communication networks, therefore breaking or enhancing OSI principles such as communication restricted to adjacent layers, by cross-layer interactions.

Table 21: Comparison of the logical components of SOA and OSI models.

SOA		OSI	
Logical component	Description	Logical component	Description
Service	is a software resource provides business functionality communicates with other services has a published interface is described in a standard definition language	Service	is a set of primitives (operations) provides communication functionality communicates with services on adjacent layers has an interface -
Interface	defines the identity of a service and its invocation logistics	Interface	specifies what the parameters are and what results to expect
Message	is a formatted service request or service response	Primitive	is a formatted service request or service response
Functionality	can be anything useful to a business is used by entities to implement their service definitions is implemented in software	Protocol	is an agreement between the communicating parties on how communication is to proceed is a set of rules governing the format and meaning of the packets that are exchanged by the peer entities within a layer is used by entities to implement their service definitions may be implemented in hardware, software, or a combination of the two

In SOA, interfaces to services define the identity of the service and its invocation logistics through a standard language which can be understood by a machine (e.g. Web Service Description Language). Then, this information is published to facilitate dynamic discovery [42]. None of these apply to OSI interfaces, which are statically defined as a result of long standardization processes.

In OSI, layer N can only request a service using service primitives (i.e. means for inter-layer communication between service requestor and service provider) from layer N-1 as discussed in Chapter 2. Services in OSI refer to communication services and use protocols to achieve their functionality. Entities on the same layer use these protocols to move the information from one machine to another. However, services are decoupled from protocols so they can use any protocol as long as they provide the same functionality and use the same interfaces.

In SOA, services of different types can communicate with other services via messages, which can be seen as equivalent to primitives in the OSI model. Making use of this communication, the composition of services can be performed based on the needed functionality, not in a static, pre-defined manner. The service functionality and the manner in which they discover other useful services is transparent to third parties, similar to the way protocols used by the underlying layer are transparent to the client layer in OSI architecture. Therefore the functionality of a service in SOA is logically equivalent to the protocol for providing communication functionality in OSI.

Both SOA and OSI are theoretical models that assume a modular structure and follow a loose coupling principle that calls for a clear separation of services from their implementation. While OSI has a clear vertical structure, SOA can be both horizontal and vertical. The main difference between the two models (see Table 22) is that SOA refers to dynamic composition of services while OSI defines only two types of services (i.e. connection oriented and connectionless) and seven functional layers. SOA is also an interconnection architecture, as it refers to interconnecting software systems into business applications. It is important to note that neither of the two models has been implemented as defined. The dominant implementation of a communication architecture is the TCP/IP protocol stack, while thus far the only implementation of an architecture for business functionality seem to be web services (e.g. Elastic Compute Cloud – EC2, Simple Storage Service – S3).

Table 22: Key similarities and differences between SOA and OSI models.

Subject (Resource)	SOA	OSI
Model	Yes	Yes
Modularity	Yes	Yes
Decoupling	Yes	Yes
Dynamic composition	Yes	No
Implemented as is	No	No
Implementation	Web services (e.g. EC2, S3)	ISDN, GSM, TCP/IP

6.2 Service Oriented Networks

By Service Oriented Network (SON), we refer to a network which makes use of SOA principles to provide end-to-end transport services. The structure of SONs is vertical

through network layers (OSI) and horizontal through network segments (autonomous systems) and devices as depicted in Figure 22.

Figure 22 End-to-end service composition in a SON.

The parties involved in service composition in SON can be categorized into two groups as shown in Figure 22: application level service providers/requestors and transport level service providers/requestors. For delivering an online gaming service, depicted as an example in Figure 22, an integrator has to determine a suitable composition of services on both levels. On the application level, this includes player services (e.g. managing account information), game publisher services (e.g. updating game software) and system services (e.g. directory services). More details on how these types of services collaborate for providing online gaming and the kind of computing and storage infrastructure needed can be found in [44].

On the transport level, transport services are needed for the access and core segments of the connection. The transport services can be provided over different media (i.e. wired and wireless) using several technologies and providing various QoS levels. The roles of transport service providers and integrators could be assumed by telecom operators since they already have network infrastructure and a large customer base. Telecom operators would need to provide interfaces for network related aspects to the services on top. The user (application) could compose a query and the integrator could integrate the required set of services into a composite service that satisfies the user (application) needs [42][45]. All this could be done automatically: finding the suitable services, and reserving and configuring the required resources (i.e. network and hardware resources using a virtualization approach).

The way this scenario could work is the following. The user (application) issues a request for a service (a query). This query can be high level, relatively abstract. The query is translated into lower level, less abstract conjunctions. Parts of these refer to application

level services and part to transport level services. The transport level query mainly refers (but is not restricted) to the quality and price of transport service. First, it refers to the access part of the network looking for published access services, and once this access part is established, the core part needs to be discovered and reconfigured. Also application level services need to be discovered and composed using a similar publish-query approach. The described scenario enables querying for composite services directly from the network edge.

6.2.1 Service and interface descriptions

According to the SOA paradigm, the description of a service and its interface must be published in order to become discoverable. The information regarding the service and its interface is logically structured in an information model. This information model is encoded using a standard description language. The descriptions can range from simple attribute-value pairs, as in the case of Service Location Protocol (SLP), to more complicated and more powerful languages such as Web Ontology Description Language Services (OWL-S), as in the case of web services. Simple description languages typically introduce less overhead but restrict interoperability, since machines cannot interpret them. Conversely, more powerful languages, which are designed to encode semantics, are not meant to be read and interpreted by humans, but by machines. Therefore semantic dissonances could be handled automatically rather than by human intervention.

The description of a service and its interface is published by the provider in a repository which can take the form of a simple database [45] or, if richer in semantics, the form of knowledge base (KB) [46]. A KB is composed of ontology (i.e. an information model and relations between its elements) and instances of service offers. The published descriptions (i.e. service offers) can be complete, partial or very high level [47]. The higher the level of the description, the more ambiguous its semantics and the harder its interpretation.

6.2.2 Querying for published services

Depending on the implementation of the repository, the service requestors send appropriate queries to discover offers with a suitable profile (i.e. a profile that matches the requestor's needs). Query is thus the mechanism by which a service requestor discovers published services and takes the form of messages, describing the requestor's needs. Queries can be performed through simple keyword search or through ontology-structured descriptions. The latter enable agents acting on behalf of requestors to automatically discover, invoke, compose, and monitor resources offering services under specified constraints. The use of ontology and logic-based reasoning also helps in solving any semantic inconsistencies and ambiguous descriptions. For the composition of transport services, we distinguish two segments: the access segment and the core segment

composed of the home autonomous system (AS) and transit ASs. For the access segment, querying is needed so that the mobile can identify the most appropriate network for connection. The more access interfaces are available and the faster they evolve, the higher the need to have a flexible and scalable query system that would support technology independent queries and would automatically interpret semantics and solve inconsistencies. For instance, the IEEE 802.21 standard is moving towards this direction by allowing RDF (Resource Description Framework) based query in addition to the binary one [45].

6.2.3 Composition of services for information transport

As already discussed, SOA principles enable flexibility and dynamic composition of discoverable services. In communications the need for flexibility and dynamic service composition for information transport is increasing. In the access segment, there is a need for choosing the appropriate access technology that allows high Quality of Experience (QoE), calling for vertical service composition [47]. In the subsequent IP domains, there is a need to establish service agreements across several autonomous systems, where each autonomous system is under different management authority, thus requiring horizontal service composition. Protocols for dynamic service composition [48] negotiate contracts, so-called service level agreements (SLAs) for end-to-end service composition. The part of the SLA specifying technical terms is called service level specification (SLS).

6.3 Service oriented networks with ProtoStack

ProtoStack is an implementation of a framework for experimentation with composition of communication stacks or, in other words, modular protocol stacks. However, the choices that we made in implementing ProtoStack also make the system suitable for experimentation with Service Oriented Networks for embedded and sensor networks. As we will show in the following sections, the key enabler of experimentation with SONs is the ProtoStack language. Another enabler is the modular design which supports composeability.

A SON is a type of network where the communication functionality is achieved by a set of communication services invoked in the correct order to perform the desired functionality. In ProtoStack, such services are wrapped by amodules and a composition of amodules form a stack which provides communication functionality. In other words, ProtoStack is a tool which allows designing and prototyping of SONs for embedded devices.

6.3.1 Describing and publishing ProtoStack services

As discussed in Chapters 3- 5, ProtoStack enables users to compose a stack out of several modules. In SON parlance, ProtoStack enables users to compose a service (or super

service) out of existing services (or atomic services). To speed up the composition and deployment process, users are spared from the implementation, configuration and deployment details. All these are hidden behind a web based workbench which is automatically populated with services available in a module library. The ProtoStack language is used as an intermediary abstraction between the C implementation of the modules and the Javascript based workbench. The knowledge provided as statement through the ProtoStack language is stored in a triple store.

According to the SON paradigm, services and interfaces need to have a description which is published and which can then be queried. In Section 5.3.1 and Appendix D we show the Turtle description of the `c_abc` information transport service. All other information transport services in the CRime stack are similarly described and all the statements are stored in a triple store and form a graph which can then be queried using SPARQL (see Sections 5.3.2 and 5.3.1). All the knowledge from the triple store can be easily published on the web and browsed by human user and/or queried using SPARQL. Figure 23 presents all the primitive functions of type `c_open` from the triple store.

Workbench

Sesame server

Repositories

New repository

Delete repository

Explore

Summary

Namespaces

Contexts

Types

Explore

Query

Export

Modify

SPARQL Update

Add

Remove

Clear

System

Information

Current Selections:

Sesame server: <http://localhost:8081/sesame> [change]

Repository: Composable Rime (c-rime) [change]

Explore (<http://sensors.ijs.si/crime#c_open>)

Subject	Predicate	Object	Context
http://sensors.ijs.si/crime#c_app_open	rdfs:type	http://sensors.ijs.si/crime#c_open	
http://sensors.ijs.si/crime#c_abc_open	rdfs:type	http://sensors.ijs.si/crime#c_open	
http://sensors.ijs.si/crime#c_broadcast_open	rdfs:type	http://sensors.ijs.si/crime#c_open	
http://sensors.ijs.si/crime#c_channel_open	rdfs:type	http://sensors.ijs.si/crime#c_open	
http://sensors.ijs.si/crime#c_mesh_open	rdfs:type	http://sensors.ijs.si/crime#c_open	
http://sensors.ijs.si/crime#c_multihop_open	rdfs:type	http://sensors.ijs.si/crime#c_open	
http://sensors.ijs.si/crime#c_netflood_open	rdfs:type	http://sensors.ijs.si/crime#c_open	
http://sensors.ijs.si/crime#c_polite_open	rdfs:type	http://sensors.ijs.si/crime#c_open	
http://sensors.ijs.si/crime#c_route_discovery_open	rdfs:type	http://sensors.ijs.si/crime#c_open	
http://sensors.ijs.si/crime#c_unicast_open	rdfs:type	http://sensors.ijs.si/crime#c_open	

Resource:

Limit results:

Figure 23 Published CRime service descriptions.

6.3.1.1 Discussion

For describing and publishing services, ProtoStack uses the standard RDF language with the custom vocabulary selected according to the requirements listed in Section 5.1.1. This approach is relatively simple, standards compliant and based on open tools. Given the described set-up, SPARQL can be used to query the repository. In the next section we discuss querying available services more in detail. We also show the role of logical reasoners for selected use cases.

6.3.2 Querying for published ProtoStack services

In this section we discuss how published ProtoStack services can be queried for. We do this by first describing a use case assuming distributed ProtoStack systems which require synchronization, and then discussing the advantage of the ProtoStack implementation versus other possible approaches.

6.3.2.1 Synchronizing distributed ProtoStack systems

In this section we show how the synchronization of distributed ProtoStack systems can be achieved. Let us assume N distributed installations of the ProtoStack system at several users across the world, each of the systems featuring the implementation of the base CRime modules M_{base} . Additionally, users of each system further develop modules of their own $M_i, i = 1 \dots N$. Any ProtoStack system S_i can check the services (module descriptions) published by any other ProtoStack system $S_j, j \neq i$ and determine the set of new modules $M_j, j \neq i$. Once this set is determined, the ProtoStack server will ask its peer for the source files (.h and .c) of the corresponding modules $M_j, j \neq i$. The Turtle statements from the files will be parsed and inserted into the local triple store which will contain a larger set of modules $M_i \cup M_j$. Further, the new modules will be integrated with the local module library and can be used by S_i 's users for experimentation with service composition.

The synchronization can be performed manually, semi-automatically or automatically. By manual synchronization we mean that the user of the S_i system inserts the addresses of target ProtoStack systems $S_j, j \neq i$ that he or she needs to synchronize with and initializes synchronization. By semi-automatic synchronization we mean that the published services of ProtoStack systems $S_j, j \neq i$ are automatically detected and the user is presented a list of systems and perhaps a list of services to select from. The synchronization is then done based on user selection. The $S_j, j \neq i$ systems can be automatically detected by using a crawler, dedicated search engine or service broker. By automatic synchronization we mean that the ProtoStack systems $S_j, j \neq i$ are automatically detected and the list of services is synchronized without any user involvement. In such case, the user may pre-set a list of preferences (i.e. synch all services that have a multi-hop scope) or can allow synchronization of everything.

One challenge that may occur with such synchronization is that the same name for a module can be used for different implementations. For instance, a routing module using reinforcement learning may be named `c_rl_routing` in two different systems, while the actual implementation may be different (i.e. using only reinforcement in one implementation while using also penalization on the other). If the system is widely adopted this may become an issue that needs to be appropriately addressed, but for now, it is beyond the scope of this work. The current ProtoStack system implementation supports

manual synchronization.

6.3.2.2 Discussion

The main advantage of the choices we made in our reference implementation of the framework with respect to the use case described in the previous section is that RDF/SPARQL can integrate data from distributed systems. This integration is trivial due to the schema re-use, one of the main advantages of SPARQL versus the widely used Structured Query Language (SQL) [49]. In the case of ProtoStack, the use of RDF with the CRime ontology facilitates the retrieval of all instances of type Module with the same query. Similarly, the retrieval of all modules implementing `c_discover` is trivial since all the systems will understand what `c_discover` is because they use the common vocabulary.

Looking beyond CRime, by extending the ontology with appropriate sub-concepts for Module, Interface, Function, Parameter and/or Scope, any other tool implementing the framework and keeping the same design decisions with respect to the declarative language can benefit from such an easy synchronization. In this context, the main disadvantage becomes obvious when a more generic query that is agnostic to various module libraries has to be fired. One such example is a query to retrieve all functions across all different module libraries. The SPARQL formulation of such a query is presented in Figure 24.

```

1. PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2. PREFIX cpan: <http://downlode.org/rdf/cpan/0.1/cpan.rdf#>
3. PREFIX crime: <http://sensors.ijs.si/crime#>
4. PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
5.   SELECT ?name ?category ?description
6.   WHERE {
7.       ?name rdf:type crime:Function .
10.  }
```

Figure 24 SPARQL formulation of a query to retrieve all instances of type Function.

The query from the figure above returns no results, even though the ProtoStack knowledge stored in a Sesame triple store contains several functions of various types (`c_open`, `c_close`, etc.). The explanation is that the reasoning capabilities in Sesame are limited. For such a query to return results, the reasoner should be able to support rules and reason on transitive relationships. In order to support such queries, ProtoStack has to be extended with a reasoner that supports rules such as Jena or OWLIM. The OWLIM compatible rule that returns valid results for the query in Figure 24 can be seen in Figure 25.

```
[rule1: (?a rdf:type ?b) (?b rdf:type ?c) -> (?a rdf:type ?c)]
```

Figure 25 OWLIM compatible rule that defines transitivity relationship.

Other advantages of using reasoners will be highlighted in the following section.

6.3.3 Composition of services for information transport using ProtoStack

In this section, we illustrate the concept of composition of services for information transport using ProtoStack. In this respect, we identify three scenarios:

- Manual composition of services, where all the necessary reasoning is performed by the human.
- Semi-automatic composition of services where the human is guided in the decision making process by machine reasoning.
- Automatic composition of services where all the reasoning is performed by machines.

6.3.3.1 Manual composition

By default, the ProtoStack system supports manual composition of communication services where all the necessary reasoning is performed by the human. Given the CRime modules, the user composes a complex communication service by selecting a subset of modules, and connecting them in a meaningful way. The responsibility for creating a correct and meaningful service lies with the user while the machine in this case performs a post-composition validity checking as discussed in Section 5.3.2.

6.3.3.2 Semi-automatic composition

By choosing a declarative language able to support logic reasoning, ProtoStack can be used for semi-automatic service composition. This way, the system will guide the human in the decision making process by using machine reasoning. One scenario in which this can be done is as follows. Assume that the human user identifies a module that is suitable for the desired complex service. The tool can then suggest other modules that can be used to build the complex service (other compatible modules). Such modules can be identified based on a set of rules used by the logic based machine reasoning. These rules progressively narrow down the search space, thus guiding the user in the decision making process.

In order to illustrate the concept of semi-automatic composition of services using ProtoStack, we assume that any CRime complex service can be composed by stacking singlehop modules on top of singlehop modules, multihop modules on top of multihop modules and multihop modules on top of singlehop one¹⁴. Stacking singlehop modules on top of multihop modules should be forbidden. By default, using ProtoStack, such constraints can be expressed by formulating two different SPARQL queries. In the case when the user selected a singlehop module, the SPARQL query presented in is sufficient. In the case the user selected a multihop module, the system should narrow the selection

¹⁴ The assumption is that the stack is built bottom-up as and not vice versa. This approach is typical for communication networking.

options only to multihop modules. To achieve this, the additional line numbered 11 shown in Figure 26 has to be added to the query in :

```

1. PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2. PREFIX cpan: <http://downlode.org/rdf/cpan/0.1/cpan.rdf#>
3. PREFIX crime: <http://sensors.ijs.si/crime#>
4. PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
5.   SELECT ?name ?category ?description
6.   WHERE {
7.       ?name rdf:type cpan:Module .
8.       ?name crime:hasScope ?category .
9.       ?name rdfs:comment ?description .
10.  }
11. FILTER (?category = crime:multihop)

```

Figure 26 SPARQL query that retrieves all multihop modules thus narrowing down the search space for the user.

For more complex scenarios, having specific queries for each case can be prohibitive. One solution can be to implement more intelligence in the system, therefore using a more powerful reasoning engine. Such approach would permit running a single uniform query for all cases, only once by inserting the rules to the engine according to which the query would be answered. For the proof of concept, the powerful Cyc reasoning engine together with its tightly coupled knowledge base is used. The CRime ontology was connected to Cyc's knowledge base by mapping the top five concepts to appropriate Cyc knowledge base concept as can be seen in Table 23. Cyc knowledge base lacked a concept for Scope as used in CRime, therefore the equivalent Scope-Module was created and inserted in the KB. The properties from CRime were analogously mapped to Cyc predicates or created.

Three rules were then added to CycKB specifying module compatibility based on the scope (see Figure 27). Given the current setup, the following query is sufficient to retrieve all combinations of two modules that can be stacked given the constraints we imposed: (#\$on-Abstract ?X ?Y).

Table 23 The mapping of Crime concepts in the Cyc knowledge base.

CRime Concepts	Cyc Concepts
Module	ComputerProgramModule-CW
Interface	InterfaceProgram
Function	CodingFunction
Parameter	CodingFunctionParameter
Scope	/ (created Scope-Module)

```

($implies
  ($and
    ($isa ?X $ComputerProgramModule-CW)
    ($isa ?Y $ComputerProgramModule-CW)
    ($hasScope ?Y $multihop)
    ($hasScope ?x $multihop)
  )
  ($on-Abstract ?X ?Y)
)

($implies
  ($and
    ($isa ?X $ComputerProgramModule-CW)
    ($isa ?Y $ComputerProgramModule-CW)
    ($hasScope ?Y $singlehop)
    ($hasScope ?x $multihop)
  )
  ($on-Abstract ?X ?Y)
)

($implies
  ($and
    ($isa ?X $ComputerProgramModule-CW)
    ($isa ?Y $ComputerProgramModule-CW)
    ($hasScope ?Y $singlehop)
    ($hasScope ?x $singlehop)
  )
  ($on-Abstract ?X ?Y)
)

```

Figure 27 CycKB rules for scope based stack composition.

6.3.3.3 Automatic composition

ProtoStack can be further extended to support research and experimentation with automatic service composition. This means that stacks are automatically composed without user intervention. A set of services can be composed automatically by providing some specifications of the desired outcome. The simplest specification is to provide the name of the uppermost module and then allow the system to figure out the required underlying modules. More complex specifications involve specifying functionality and/or QoS.

In order to support automatic composition, several components of the ProtoStack system need to be extended. First and foremost, the descriptions of the services need to be richer and more emphasis needs to be put on the input parameter description and their allowed ranges. This requires ontology extension, additional Turtle statements in the source files and the extension of the validity checking module. Furthermore, potentially some heuristic for determining values for parameters will need to be employed [50].

The complexity of the resulting system will increase but will open the possibility to further explore aspects of self-assembling protocol stacks and information transport services [51][52][53][54][55]. For instance, consider the following use case: two nodes (if

we refer to embedded or sensor networks) or two terminals (if we refer to mobile communication services) are in the range of each other, can detect each other but cannot “talk” to each other because each uses a different protocol stack. Both nodes could communicate with a coordinator (gateway) or an access point and could provide a description of their current software and possibly hardware configuration. The coordinator/access point could then automatically find the services needed to compose the required protocol stack and send it to one or both nodes. Finally, after reconfiguration, the nodes will then be able to communicate directly.

To implement the described use case, additional description of the modules that compose a stack is needed. The knowledge base should contain information about the modules that are appropriate for wireless communication, that offer functionality for reliable communication, etc. Then, appropriate rules stating what reliable communications means need to be inserted (i.e. if at least one module in the stack offers such functionality, then the stack is reliable). Given a query asking for wireless, reliable, multihop stack, a set of candidate stacks should be assembled and presented to the user (human or machine). Extending the CRime ontology and/or mapping it to an external ontology to cover such complex scenarios is left for future work. Such extensions are necessary to create a specialized logic-based self configuration system for SONS but the resulting vocabulary might be too complex for users with other research interest.

6.4 Summary

This chapter introduced and discussed Service Oriented Networks and their similarities with the Open Systems Interconnect Architecture. The concept was then illustrated on a use case and three steps in realizing it were identified. A discussion on the representation languages and logic based reasoning for SONS was followed by implementations using ProtoStack. We explained the increasing complexity required from the representation language and reasoning process for achieving various degrees of automation for SONS.

The next chapter discusses and surveys the state of the art in cognitive networks focusing on two aspects: knowledge representation and cognition loops. This is then followed by illustrations of how ProtoStack is used for knowledge representation and cognition loops.

7 Dynamic composition of services for Cognitive Networks

In this chapter we first look at theoretical aspects of cognitive networks by describing the two key components of a cognitive network: the knowledge representation and the cognition loop. We show on two use cases how the CRime declarative language and the ProtoStack tool can be used for knowledge representation and reasoning. Then we show how modularity can ease the design and implementation of cognitive loops by discussing a routing strategy which uses reinforcement learning implemented as a CRime module.

7.1 Cognitive networks

The notion of Cognitive Network (CN) was defined in [56] and it has presumably been inspired by the Knowledge Plane (KP) described in [29] as a “distributed cognitive system that permeates the network”. The KP makes use of cross layer information and extends beyond this in a cross-device manner by making use of cross-device information models. In [56], the authors define the CN as a network with a cognitive process that can perceive current network conditions, plan, decide, act on those conditions, learn from the consequences of its actions, all while following end-to-end goals. This loop, the cognition loop, senses the environment, plans actions according to input from sensors and network policies, decides which scenario fits best its end-to-end purpose using a reasoning engine, and finally acts on the chosen scenario as discussed in the previous section

This definition of CN does not explicitly mention the knowledge of the network; it only describes the cognitive loop and adds end-to-end goals that would distinguish it from cognitive radio (CR) or so-called cognitive layers [56]. We consider this definition of CN incomplete, since it lacks knowledge which is an important component of a cognitive system [29][28][15][57][17].

In particular, [57] gives an interesting view of the role of ontological knowledge representation: “The persistent nature of this ontology enables proactiveness and robustness to ‘ignorable events’ while the unitary nature enables end-to-end adaptations.” We consider this statement essential for CNs. The authors of [57] also state that the “core persistent and unitary ontology is what will distinguish cognitive systems from merely adaptive systems which may also need ontology to function but will not have such a notion of ‘self’.”

In this work, we consider that a CN is a communication network augmented by a knowledge plane (KP) [29]. This KP can span vertically over layers (thus making use of cross-layer design, as discussed in Section 2.1) and/or horizontally across technologies

and nodes (thus covering a heterogeneous environment), as depicted in Figure 4f. The KP needs at least two elements:

- A representation of relevant knowledge about the scope (device, homogenous network, heterogeneous network, etc.)
- A cognition loop which uses AI techniques inside its states (machine learning techniques, decision making techniques, etc.)

7.1.1 Knowledge representation

In [15], Mitola claims that “to be termed ‘cognitive’, a radio must be self-aware”. By self-aware he refers to a computationally accessible description of its own structure. Extending this beyond the radio domain, it can be said that for a network to be cognitive, it must be self-aware [28] (p. xix), that means it should have knowledge about itself, its building blocks and their interconnection, it should be able to share this knowledge and reason based on this. This aspect is analogous with the automatic composition of services in SONs as discussed in Section 6.3.3.3.

In [57], context awareness is seen as prerequisite for achieving cognition. By context awareness, the authors refer to awareness about the “self” and about the “world” and define it as the “information that surrounds and gives semantic meaning to an entity”. To this end, ProtoStack is a possible implementation of a knowledge representation system for devices which feature a stack composed with CRime modules. ProtoStack saves descriptions of the modules, their interconnection and configuration parameters and is able to share these. In other words, ProtoStack stores a description of the CRime world model and of instances of composed CRime services.

Alternative views and definitions of context and context-awareness can be found in [58]. However, defining what exactly is “self”, “world” and “context” depends on the domain of application, the design choices, etc. and is beyond the scope of this thesis.

In summary, knowledge empowered networks seem to be the only way towards reaching the point where a network can configure itself, explain itself and repair itself [29]. But this raises a fundamental question: “How to represent this knowledge and how to reason about it?” In [59], the authors argue that a knowledge representation is most fundamentally “a surrogate, a substitute for the thing itself”, thus a model [60] (p. 48). This model, the reasoning based on it and updating it can be implemented in several ways, depending on the focus (i.e. on what is wanted to be captured). Thus tables, Boolean circuits, neural networks, artificial languages, etc. can all be viewed as forms of knowledge representation. According to [59], intelligent reasoning and corresponding knowledge representation can be classified into five categories according to their origins: mathematical logic, psychology, biology, statistics and economics.

In the following sub-sections we briefly overview existing work with respect to knowledge representation for communication networks and conclude with a discussion.

7.1.1.1 Knowledge representation for management

The work in [61] is an illustrative example for two types of structured knowledge representations for telecommunication network management: ontological and Bayesian network representations. They use ontology to represent domain specific knowledge and knowledge about Bayesian networks. This conceptual knowledge enables interoperability and data integration but also allows machine based reasoning through inference. The domain ontology is then used to construct Bayesian networks in an automatic fashion. The obtained Bayesian network is a probabilistic model of the telecommunications network and is used for fault management.

In [28](p. 23) it is argued that self-aware entities need ontological models that encode domain specific knowledge. In building their system they use information models and ontologies for modeling the managed elements of a network. The information models are high level, generic models designed to be able to model all managed elements. Vendor specific facts, encoded as Object Management Group (OMG) compliant data models, are derived from these information models to describe the managed devices produced by a specific vendor. Each data model has one or more ontologies associated with it. These ontologies encode definitions, relationships and semantics, things that cannot be encoded by information or data modeling languages [62]. With this approach, the FOCAL system is able to reason about the facts represented in the data models by using Web Ontology Language (OWL) ontologies of the metamodels associated with the respective data models. The system supports description logic (DL) reasoners and querying with SPARQL and other query languages for OWL (see [62]). This complex approach involving both semantic and non-semantic technologies for modeling a system described as self-governing seems to have been taken for two reasons. The first is to ease sharing of information residing in private knowledge bases and the second is to keep backward compatibility with existing standards.

An approach for modeling infrastructure wireless networks with the scope of resource management can be found in [67]. They use Bayesian networks to model the capability of a node according to a corresponding configuration. The model encodes information about the coverage area and average bit rate that can be achieved in that area. The approach in [68] is similar as both of them are coming out of E2R and its follow-up E3 [69] project.

The case study in [70] demonstrates the use of learning automata for wireless resource management. The learning automaton holds a model of the environment. This model is created using delayed rewards based on the reaction of the environment to previous actions [71](p. 8). The environment is unknown and random (e.g. wireless path) and learning automata seem suitable for modeling such unreliable environment.

The next example of utilization of ontological knowledge representation are the Knowledge Based Networks (KBNs) [72][73]. KBNs are semantically enhanced content based networks in which content publishers use ontological representations of their

information while subscribers use semantic queries to find relevant information. In [73], heterogeneous ontologies from different publishers are semantically inter-mapped to enhance querying. This technology is applied to the problem of exchanging policies related to dynamic spectrum access in a self-managed network. In this context, publishers can be seen as entities that are self-aware of their capabilities and advertise them. On the other hand, subscribers are aware of their needs and query for resources able to satisfy them.

In [66], the authors consider domain specific ontologies that model dependencies between state variables of a device. The proposed graphical model incorporates the dependency information between state variables as the knowledge base for effectively guiding the device centric optimization process. This work is a follow-up of the work in [57].

Finally, in [74] the author identifies requirements for a Cognitive Radio Policy Language and performs a cognitive linguistics analysis of the cognitive radio domain. The analysis shows that the language needs five constructs under which most of the necessary knowledge can be encoded. The five constructs are things, places, paths, actions and causes.

7.1.1.2 Knowledge representation for quality of service

From several papers on modeling QoS related aspects, we were able to identify two areas of application for semantic QoS modeling. The first one is horizontal QoS provisioning which is concerned with finding, matching, providing and monitoring QoS based services between two parties [75][76]. The second one is vertical QoS provisioning in which application QoS requirements are semantically translated into lower level requirements (platform dependent or platform independent) [77]. However, in [78] and [79] a system that encompasses both approaches is described.

The work in [78] uses multilayer neural networks (MNN) to model and estimate user preferences with respect to wireless services. The MNN are trained with user feedback data to build a model of the user's perception of past provision of wireless services. Based on this model, an agent is able to estimate the satisfaction of the user with respect to a new, previously unseen service. The goal is to choose a wireless service that would satisfy the user preferences without requiring extensive interaction [79].

In [75], a QoS ontology (DAML-QoS) modeling QoS profiles, properties and metrics is defined. A matchmaking algorithm determines the published services that semantically match the ones inquired. A measurement system makes sure that the service agreement is respected. The approach in [77] seems suitable for achieving semantic translation between platform independent and platform dependent QoS requirements. However, with the translation the authors do not go that low in the OSI architecture claiming that "low level metrics are not easily understandable and applicable for application developers".

In [80], self-awareness is achieved through on-line self monitoring and measurements with the final scope of offering QoS to users. The essence of the work is to validate a test bed for QoS routing. At each node (router), the network (all possible outgoing routes) is modeled using random neural networks (RNN). Smart packets (SPs) are sent to discover new routes in the network and acknowledgement packets (ACK) are used to bring back information discovered by the SPs. The network model (i.e. the weights of the RNN) is updated with the new information.

7.1.1.3 Knowledge representation for security

The work in [81] focuses on security semantics and tests the prioritization of types of content and flow over an enterprise network. It uses semantic technology to enable fine grained, highly specialized services for policy based networking. A packet sent through the network has an OWL semantic tag which is used by the policy decision points encountered on the way for reasoning and inferring the set of operations to perform.

In [36], security domain knowledge was manually encoded in a special purpose unit of the Cyc knowledge base (KB). The KB is periodically updated with the information gathered by sentinels placed on client machines. In essence, the CycSecure application, takes the information pulled from the sentinels, transforms it into CycL (Cyc internal representation language), and builds and maintains a model of the network based on the pre-existing ontologies of networking, security and computing concepts. This system is able to explain in natural language what the vulnerabilities of the network are and allows high level input from the user.

An ontology for modeling network attacks has been created in [82] using Resource Description Framework Schema (RDFS) formalism and Prolog programming language for reasoning. The authors further illustrate the benefits of using ontology in this particular context. Based on the ontology, they update the KB to always have an accurate model of the state of the network with respect to attacks. The work in [83] focuses on a specific security problem, namely distributed denial of service attacks (DDoS). As an intermediate step towards modeling DDoS attacks, the authors built an ontology of such attacks.

7.1.1.4 Discussion

In this section we showed that networks of the future will have to be based on knowledge, we briefly addressed the problem of knowledge representation and summarized relevant work related to knowledge representation and reasoning for networks. Table 24 synthesizes this research work and its main features. The first four rows of the table present research for knowledge representation and reasoning for management purposes, the next four rows for QoS while the last three for security.

The columns of the table list key-features of research on knowledge representation and reasoning for networks. The first column contains the name of the project, of the

prototype or just an intuitive name for the research work. The second column lists the scope for which knowledge representation and reasoning are used. The third column refers to the type of model used for representing the thing under investigation. The fourth column lists the representation language, where the case, and the fifth column lists the type of the representation (symbolic, where an artificial language is used, and numeric otherwise). The sixth column notes whether the model is updated or not. Finally, the last column lists the reasoning engine used in the experiments.

As can be seen from the third column, ontology is the most used form of knowledge representation in the surveyed papers. Ontological representation is a structured knowledge representation using concepts such as actions, time, physical objects, etc [60]. Domain specific (management, QoS and security) ontologies were built and in some studies they were extended [36] or were joined [61][73] with other domain ontologies as the nature of the problem being solved required knowledge about other areas as well.

An OWL [39] approach was mainly used as the language for ontology representation. RDF (Resource Description Framework) [40], RDFS (RDF Schema), DAML (DARPA Agent Markup Language) [41] and CycL are other representation languages that were used. OWL, RDF and RDFS are all standardized by the W3C (World Wide Web Consortium), DAML extends RDF and RDFS while CycL is a proprietary language. All these languages allow representing semantic knowledge and are used in the semantic web [40]. UML (Unified Modeling Language), MOF (Managed Object Format), OCL (Object Constraint Language), and QVT (Query/View/Transformation) are all Object Management Group (OMG) [85] specifications used for network management but unable to provide semantics [28] (p. 23). The lack of semantics in those representations is the motivation for building OWL meta-models that empower semantic translations for autonomic management [62].

Ontological models make use of reasoning engines to infer new knowledge from the one encoded in the model. Jena [44], Pellet [45], FaCT [46], RacerPro [47] (see Table 24) are all description logic reasoners based on tableau calculus. Jess [48] is a rule based engine, Cyc [34] has various modules for different types of reasoning while Prolog is a well known logic programming language.

Table 24 Summary of surveyed approaches to knowledge representation and reasoning.

Research	Scope	Model Type	Repres. language	Repres. type	Model update	Reasoning engine
Node management [61]	Network management	Ontology + Bayesian network	OWL	Symbolic & numeric	Yes	Jena
FOCALE [28](p. 23), [62]	Backbone network management	Information and data models + ontologies	UML, MOF, OCL, QVT + OWL	Symbolic	?	RacerPro, Pellet, FaCT
Node management [67][69][68]	Wireless infrastructure management	Bayesian network	-	Numeric	Yes	Bayesian network
Cognitive network [16]	Wireless resource management	Learning automaton	-	Numeric	Yes	Learning automaton
Knowledge based networks (KBN) [72][73]	Semantic publish-subscribe networks	Ontologies	RDF, OWL	Symbolic	Yes	RacerPro, Jena, Pellet
Cognitive agent for the personal router [78]	Wireless service selection	Multilayer neural network	-	Numeric	Yes	Multilayer neural network
DAML-QoS [75]	Semantic QoS matchmaking and measure	Ontologies	DAML+OIL, OWL	Symbolic	No	Jena
Cognitive packet networks (CPN) [80]	QoS aware routing	Random neural network	-	Numeric	Yes	Random neural network
Semantic tags [81]	Semantics for secure enterprise networking	Ontology	OWL	Symbolic	No	Jess
CycSecure [36]	Network risk assesment	Ontology	CycL	Symbolic	Yes	Cyc
Ontology for intrusion detection [82]	Modeling computer attacks	Ontology	RDFS	Symbolic	No	Prolog
Self-configuring and self-optimizing devices [66]	Link adaptation	Influence graphs	-	Symbolic	Yes	Influence graph

It can be seen in Table 24, that besides ontological models, also neural networks, Bayesian networks and learning automata are used for modeling management and QoS related aspects. These techniques are best suited to model uncertain knowledge and perform reasoning under incomplete information [60](p. 462). Even though several papers that address cognition related issues for communication networks [29][15][57] strongly argue for the use of conceptual knowledge encoding (i.e. ontology), most likely that will not suffice, if for no other reason then for the complexity and high resource consumption of this technology. Reasoning for this kind of knowledge representation is based on logic, but logic cannot always solve the problems. When only partial knowledge or unreliable knowledge is available, then technologies modeling uncertainty are required [60](p. 462). In this sense symbolic and numeric knowledge representations should both be used to complement each other.

Typically, with a good prior model, less training data is required to obtain the posteriori model than with a less good prior model. An illustrative example in [61] uses Bayesian networks (numeric) for modeling the node (probabilistic model with incomplete knowledge), but the construction of this model makes use of prior domain expert knowledge encoded in an ontology (logic model with domain knowledge). There is also a way to combine probability theory, which is able to handle uncertainty, with deductive logic, which is able to exploit structure, resulting in probabilistic logic [95].

In summary, the form (or combination of forms) of representation used is not important provided it is adequate for the task. Suggestions for approaches to knowledge representations for computer networks (e.g. [57]) and sources of knowledge representation techniques in AI (e.g. [60]) are widely available. Also, as discussed in this section, there exist several approaches towards knowledge based networks. However, most seem to be describing architectures and frameworks, presenting initial approaches to build knowledge representations for computer networks. Very few present actual simulation results and even fewer show performance comparisons; in fact, we were not able to find any representative for this last category.

Another issue with modeling is whether the models are updated or not. It can be seen in Table 24 that ontology based models are typically not updated or if they are, this happens in a manual or semi-automatic way. As opposed to this, the models which use numeric representation are all updated automatically. Evaluations are required on how reliable the ontology based models are in the context of communication networks. Are they actually fit for the use inside the cognition loop, or are they more useful in interpreting and preparing relevant data to be presented to the user as done in [34]. Proving the utility of these models for critical 99.999% systems is still an open challenge.

7.1.2 Cognition loop

All systems that are able to adjust their functioning according to changes in their environment are based on feedback information. Cognitive networks are no exception in this respect, so they will also use a control loop, also called cognition cycle [15] (p. 47), feedback loop [56], context based adaptation loop [57], etc. According to [56], the loop employed by a cognitive network should be based on the concept of the Observe-Orient-Decide-Act loop originally used in the military, augmented by machine learning and following end-to-end goals to achieve cognition. In [57], the loop also has a communicating capability for communicating with other loops in a distributed environment.

The cognition cycle as described in [15] (p. 48) features the following states: observe, orient, plan, decide, act and learn. It uses the orient module for classifying stimuli and does not explicitly encompass policies. In [57], a taxonomy of cognitive capabilities can be found. They introduce a context-based adaptation loop having five high level states.

- The first state is Sense which is presented as having the following components: context gathering, measure, monitor, pre-process and approximate.
- The second state is Analyze and is responsible for interpretation and abstraction, consistency check, prediction, data fusion, reasoning, model management and update, and learning.
- The third state is the Decide state that is formed of the following components: select action, prune solution space, apply policies and constraints, evaluate alternatives, optimize and make the decision.
- The Reconfigure state is the fourth and it implements decisions, selects components, downloads components and sends alarms and warnings.
- The last state is the Communicate state and this is responsible for sending and receiving commands and data between cognitive engines distributed around the network.

Partially inspired by the approaches mentioned above, we define a reference cognition loop as depicted in Figure 28, consisting of six states: Sense, Plan, Decide, Act, Learn, Policy. In our view, the self-aware network will employ sensors to sense the operating environment (Sense). The observations captured by the sensors will be further used for planning (Plan), but they will also be fed to a machine learning module able to learn and remember (build a model from the) useful observations (Learn), which can aid the decision making module in the future (Decide). The planning module determines potential actions, i.e. strategies to be followed based on observations and policies (e.g. externally defined restrictions) stored in the policy module (Policy). The decision module decides on the actions to be taken based on possible moves (actions) and (learnt) experience. Finally, the actuators (Act) are responsible for executing the adequate

changes (reconfigurations). The machine learning module is the one best connected in the sense that it can learn from several sources: from sensing data, from strategies, from decisions and from actuators, and can correlate and infer from this knowledge.

Figure 28 The cognition loop

According to [15] (p. 48), the described cognition cycle would occur under normal operating conditions. In “under stress” operating conditions, such as the ones where a power failure occurs, some modules can be bypassed. This is done by assigning priorities to the sensor output such that a stimulus classified as “Urgent” would trigger actions immediately.

In the following sub-sections we briefly overview existing work with respect to cognition loops for communication networks and conclude with a discussion.

7.1.2.1 Loops for management

In [78], the authors investigate a cognitive agent for wireless network selection which is designed to hide the complexity of the wireless environment from the user. The selection problem is decomposed into four elements that enhance the agent to select the network which is most suitable to user preferences. First, user’s feedback that the decision making process will be used is captured. Second, the available services are evaluated against learned user preferences. Third, the agent decides when to change services and which new service to select based on user’s preferences, context and goals. Fourth, the value of previously unseen services is predicted. Using this approach, the agent continuously monitors the wireless environment and selects the best service according to the current model of user preferences. However, when the user is not satisfied (or changes preferences), the model is updated and a new selection is made to satisfy the preferences.

In [70], a case study of a cognitive network is considered. The work addresses distributed wireless resource management issues with the scope of maximizing connection lifetime. Each node of the network employs its own cognition loop. The cognition loop gets sensor information from a directional reception antenna and controls an omnidirectional transmission antenna that has the role of the actuator. The plan, decide and learn modules of the cognition loop are implemented by a learning automaton [71](p.

8). With learning automaton, the plan module of the loop allows different available alternatives (e.g. received power levels). The decision module selects one of the alternatives based on the reaction of the environment to one of the previous actions. These reactions are then learnt.

The node management in wireless infrastructure [67] has two end-to-end goals: maximize the QoS levels allocated to users, and minimize the number of needed reconfigurations. They also use a feedback loop for solving the problem. In the monitoring phase the cognitive node gathers information about the number of users, their mobility and traffic characteristics. The planning and decision phases use Bayesian networks and a four phase strategy for choosing the best reconfiguration. They make use of element profiles and network operator policies to determine possible reconfigurations. In case of a well performing reconfiguration, the probability of choosing it in the future is augmented by reinforcement learning.

In [63], the authors employ a cognition loop to improve the performance of delay-sensitive applications such as streaming services over multihop cognitive networks. The cognition loop in this work makes use of cross layer information by learning channel conditions and source characteristics as well as the action of competing nodes. The decisions for the channel and network layer route are based on the learnt models. Simulations have shown superior results comparing to existing non-learning based alternatives.

The authors of [64] propose a secondary network topology control that learns the behaviour of primary users and maintains this knowledge in the form of an outage potential map. A heuristics approach is then used to search this map at the time of decision. The evaluations show that the heuristics generally outperforms the reference topology control algorithm with up to 77% over 90% of the time. When the heuristics performs worse than the reference topology control algorithm, it is usually worse by less than 10%.

In [65], the authors propose using a node level factor graph representation of the network and Bayesian inference on the resulting graph in order to learn channel occupation probabilities. The learnt model could then be used for network level decision making in a multihop setup.

In [57], the architecture for a cognitive node is introduced. This node uses a cognitive process, namely a Sense-Analyze-Decide-Reconfigure loop, specified using process algebra notations. In the follow-up work [66], the cognition loops is instantiated using influence graphs that are used to solve an optimization problem formulated for a link adaptation use case.

A Cognitive Resource Manager (CRM) and its conceptual architecture are introduced in [93]. The CRM's functioning is based on a cognition cycle adapted from [15] and aims at enabling autonomic optimization of the communication stack as a whole, thus acting as

an intelligent vertical calibration (cf. Section 2.1). The instantiation of the concept has been subsequently reported in [94], where optimization and learning methods are used for cross-layer optimization in wireless networks. Simulated annealing is used for parameter optimization, and graphical models of relations between parameters are used to decrease the convergence time. The evaluations show the potential of the CRM over existing technology and highlight open issues mostly related to the cold start problem and to the speed of convergence.

Other related cognition loops can be found in [28](p. 23), [95], [96] and [97]. These are mainly high-level views of cognition loops used for various types of management tasks.

7.1.2.2 Loops for overlays

The scope of the CogNet project [98] was to build an experimental cognitive radio network for testing cognitive techniques. It targeted the first three layers of OSI. In [99] they evaluated a multi-overlay network layer, based on the Host Identity Protocol (HIP) operating on top of IP, with respect to the delay introduced by the addition of HIP and security. However, they envisaged that the multi-overlay network layer can adapt based on sensing and learning and that the usage of a particular overlay can be decided based on reasoning and learning.

7.1.2.3 Loops for security

The CycSecure application [36] makes use of an incomplete cognitive loop. It uses daemons installed on machines in the network that collect local information and send it to the server when polled. A human operator can examine and modify the network model, query and view network statistics. The system is able to generate possible attack plans based on the information gathered from the system and the internal knowledge base. Based on these attack plans, the human operator can decide for remedy measures to increase the security of the system.

7.1.2.4 Discussion

In this section we have showed that a loop or cycle having suitable states will empower the cognitive engine. Such systems will be able to behave in a reactive way but, even more important, they are also expected to be able to behave pro-actively through learning and planning.

It can be seen from the overview that the number of instantiations, trials and validations of cognition loops has increased in recent years. Our previous overview of the domain [17], mostly found high level definitions of cognitive engines. Most of the work focuses on management issues (i.e. resource and network as foreseen in [15] (p. 57)), one focuses on overlays and one on security. Further results are expected to be shown by mainstream projects [100] employing cognitive loops in their prototype systems.

Table 25 synthesizes relevant work implementing cognition loops. All but the last row of the table focus on management aspects of networks, while the last one focuses on security aspects. The columns of the table list the technologies used for instantiating the states of the loop. Compared with Table 24, this table has fewer entries. This is also due to the fact that the symbolic models based on ontologies from Table 24 are hard to map to the stages of the cognition loop based on the information available in the papers. Exceptions are [57][66] and [36]. One remark is that the surveyed work seems to address only some of the stages of the cognition loop such as Sense-Learn-Decide-Act or Sense-Plan. This is a natural evolution in research when after the definition of concepts more focused lines of research address specific aspects in specific scenarios.

While in the previous overview [17] the dominant approach for learning seemed to be the reinforcement learning technique, the current state of the art uses a diversity of probabilistic methods for the learning and decision states as can be seen in Table 25. One remark is that most of the work with respect to cognitive networks focuses on PHY and MAC aspects and it is hard to identify progress on cognitive networks in general and beyond IP in particular. For instance, from a recent survey on routing in cognitive radio networks [101] only one of the considered solutions involves learning. All the other solutions assume that the cognition happens at PHY/MAC level and mostly existing routing strategies are redesigned for a cognitive radio set-up. Probably the so-called ossification of existing TCP/IP protocol stack hinders such development while network and higher level protocols for cognitive radio networks are still in very early stage. The less rigid protocol development environment from the sensor networks community might prove more open to investigating cognitive approaches at network and higher levels. The work reported in this thesis is a small step in this direction.

Table 25 Summary of surveyed approaches to cognition loops.

Research	Sense	Plan	Decide	Learn	Policy	Act
Personal router [78][79]	Personal router (wireless services, user feedback)	Multi-layer neural network in [78], Markov decision process in [79]		Reinforcement learning	User preferences	Personal router (wireless service)
Cognitive network [70][16]	Directional reception antenna (power)	Learning reward-interaction automaton			-	Omnidirectional transmission antenna (power)
Node management [67]	Reconfigurable node (users, applications, mobility)	Bayesian network (bit rate, coverage)		Reinforcement learning	Element profiles (transceiver, spectrum, etc.), network operator policies	Reconfigurable node (change RAT, spectrum)
Resource Management in multihop CRNs [63]	Channel occupancy, source characteristic, transmissions of competing nodes	-	Model based learning (fictitious play)		-	Transmission power and strategy
Secondary Network Topology Control[64]	Path loss	-	Maximal independent set based heuristic	Outage potential map	-	
Channel Occupation Probabilities [65]	Spectrum sensing (power), packet	-	Bayesian networks	Improved Iterative Scaling learning	-	Transmission power
Cognitive node [57][66]	Throughput, channel	Influence graph		-	-	?
CRM/CE [93][94]	Phy/MAC/IP parameters (packet size, rety limit, sensing interval, etc.)	-	Simulated annealing, graphical models, Bayesian networks		User defined boudry values for parameters.	Values of Phy/MAC/IP parameters
CycSecure [36]	Daemon	Simple Hierarchical Ordered Planner	Cyc	-	-	User

7.2 Knowledge representation in ProtoStack

As shown in Chapter 5, the ProtoStack declarative language is used to describe a stack and its configurations. Chapter 6 discussed the use of the ProtoStack declarative language to describe and publish communication modules as services. In this section we argue and demonstrate that the same language can be used for representing knowledge and reasoning about elements of the network, again taking CRime as the reference module library.

In CRime, each module is described using the ProtoStack declarative language. The description is done using the CRime ontology and currently covers only the needs of the ProtoStack experimentation system. However, ontologies are recognized as an efficient means to support extensions of systems. If we consider the case of CRime, we can see that for each module we describe the primitive functions it defines, the interfaces these functions implement, a set of parameters that the module uses and the scope of the module. This model is called the CRime world and is forms a graph structure as presented in Figure 29.

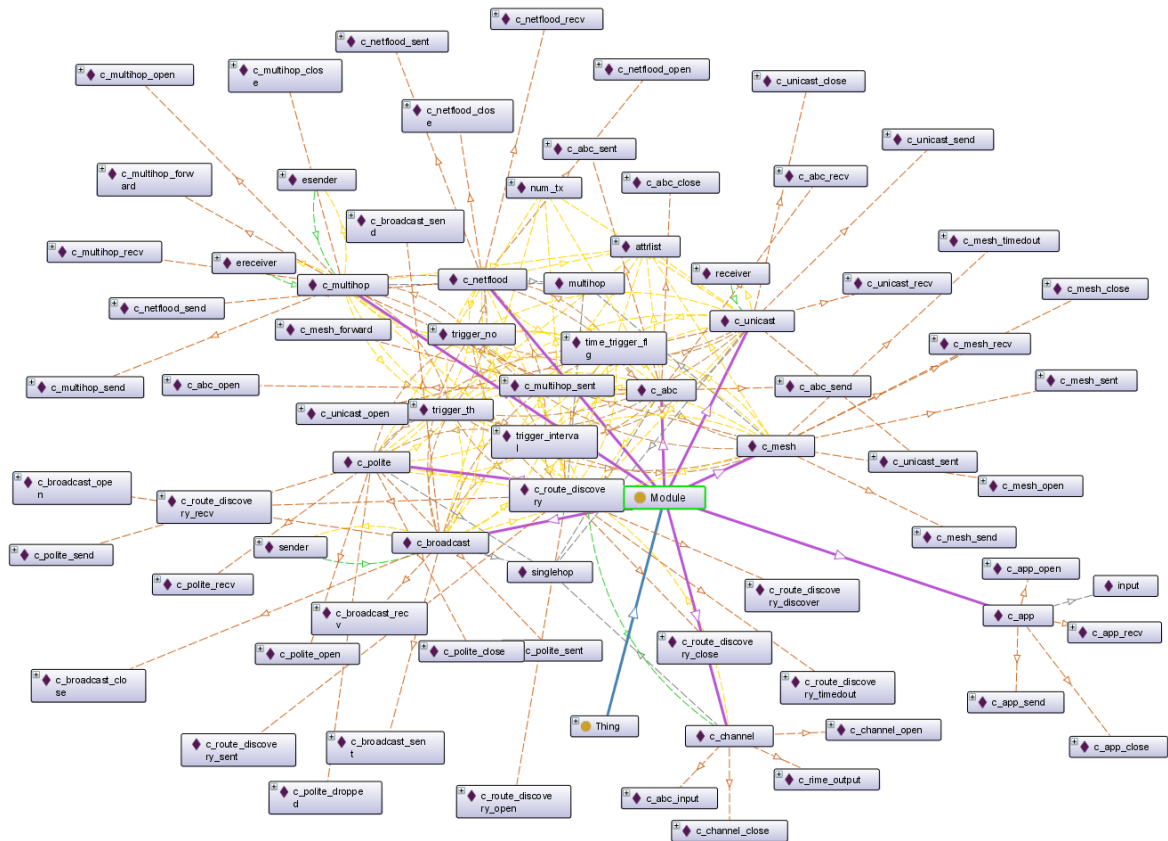


Figure 29 Snapshot of the CRime RDF graph.

Looking at the knowledge representation from the cognitive network perspective, it would be useful to see how the CN world model would look like and which aspects of a protocol stack are relevant for such a network. First, and foremost, the CN needs the

concept of Stack. Second, it needs to know that stacks can be modular and that CRime provides modules. It would also need to know that stacks can be part of an operating system or they can be separate (i.e. we refer to the sensor network domain; the terminology differs in the communication networks domain in general). In both cases the CN needs a notion of memory footprint of the image. Then, there should be a notion of Network Element (or Embedded Device, Sensor Platform) and its capabilities (i.e. model of the components of such platform, their technical specifications, etc.).

It can be seen that we are looking to represent the domain specific knowledge that humans possess as a result of the education, training and experimenting. So we seek to transfer the knowledge of domain experts in networking technologies to a system which will be able to perform some tasks on our behalf. Such knowledge can be encoded in ontologies and, using rules and reasoning, a cognitive network can be able to determine which stack would be suitable for which device – at least from a footprint point of view.

In order to refine the selection criteria (i.e. which stack should be sent to which device), additional knowledge can be used. For example, does the stack need to provide physical single or multihop communication? In order to reason about this, we need additional rules describing which combination of logical singlehop and/or multihop modules can provide physical singlehop and multihop functionality. The knowledge and scope of the system can be further extended by considering application requirements, QoS, context, etc.

7.2.1 Smart grid use case

The first use case we consider for demonstrating the use of ProtoStack for reasoning about the network is as follows. Consider a smart grid scenario where sensor nodes are mounted on light poles and measure power consumption and luminance. The sensors themselves should consume as little power as possible, and the footprint of the OS should be as small as possible to allow saving measurements on the remaining memory space. Also, the communication stack should be updated as soon as better performing modules become available.

The application requirements are as follows. The sensor nodes may use a less reliable stack during the day when light poles are switched off and should use a reliable stack during the night when no data should be lost. Typically more reliable and better performing stacks also come with a higher footprint. For the proof of concept, we extended the CRime ontology with few concepts (i.e. Stack, consumesPower, hasFootprint) and we show queries which can be used to implement such use case. Figure 30 presents the query for a stack to be used at night. The stack needs to use the `c_reliable` module and the results should be ordered by power. Stack one is the top result which is displayed. The corresponding query for a less reliable stack would skip the third line of the query “`?stack :formedOf :c_reliable`” and in this case the result would be Stack 2

which has lower power consumption and does not include a `c_reliable` module. Figure 31 illustrates the query for selecting a stack with footprint less than 5232 bytes.

```

1. PREFIX :<http://sensorlab.ijs.si/2012/v0/crime.owl#>
2. PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
3. PREFIX Process:<http://www.daml.org/services/owl-s/1.1B/Process.owl#>
4. PREFIX owl:<http://www.w3.org/2002/07/owl#>
5. PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
6. PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
7. PREFIX cpan:<http://download.org/rdf/cpan/0.1/cpan.rdf#>
8.
9. SELECT ?stack WHERE {
10.   ?stack rdf:type :Stack .
11.   ?stack :formedOf :c_reliable .
12.   ?stack :consumesPower ?power .
13. }
14. ORDER BY ?power
15. LIMIT 1

```

Explore (:stack_1)			
reliable multihop			
Subject	Predicate	Object	Context
:stack_1	rdf:type	owl:NamedIndividual	<file://C:/fakepath/knowledge representation.owl>
:stack_1	rdf:type	:Stack	<file://C:/fakepath/knowledge representation.owl>
:stack_1	:consumesPower	4	<file://C:/fakepath/knowledge representation.owl>
:stack_1	:hasFootprint	5742	<file://C:/fakepath/knowledge representation.owl>
:stack_1	rdfs:comment	"reliable multihop"	<file://C:/fakepath/knowledge representation.owl>
:stack_1	:formedOf	:c_broadcast	<file://C:/fakepath/knowledge representation.owl>
:stack_1	:formedOf	:c_channel	<file://C:/fakepath/knowledge representation.owl>
:stack_1	:formedOf	:c_multihop	<file://C:/fakepath/knowledge representation.owl>
:stack_1	:formedOf	:c_reliable	<file://C:/fakepath/knowledge representation.owl>
:stack_1	:formedOf	:c_unicast	<file://C:/fakepath/knowledge representation.owl>

Figure 30 Querying for a reliable stack which consumes little power.

7.2.2 Cognitive ad hoc network use case

The second use case considers a cognitive radio network scenario as follows. A set of sensor nodes covering a given region (i.e. a university campus) form a network by communicating on channel 132 in the 868 MHz ISM band using a multihop stack. A mobile node having a tunable radio module comes in the vicinity of the cognitive ad-hoc network, scans the spectrum and detects there is communication on channel 132. The mobile node cannot join the network because it does not know what communication protocols the network is using. The mobile node can send a query to a repository (possibly co-located with a spectrum usage database) specifying its location and the sensed parameters. The repository is aware of the cognitive ad-hoc network and its technical configuration (i.e. if not, it queries peer repositories to find the necessary information). This configuration may have been saved using the ProtoStack system and is specified using the ProtoStack language. The specification of the stack used by the

cognitive ad-hoc networks is retrieved, the corresponding modules are gathered and compiled and the resulting image is then sent to the mobile node. From this point on, the node can join the cognitive ad-hoc network.

```

1. PREFIX :<http://sensorlab.ijs.si/2012/v0/crime.owl#>
2. PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
3. PREFIX Process:<http://www.daml.org/services/owl-s/1.1B/Process.owl#>
4. PREFIX owl:<http://www.w3.org/2002/07/owl#>
5. PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
6. PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
7. PREFIX cpan:<http://download.org/rdf/cpan/0.1/cpan.rdf#>
8.
9. SELECT ?stack ?footprint WHERE {
10.   ?stack rdf:type :Stack .
11.   ?stack :hasFootprint ?footprint .
12.   FILTER (?footprint < 5232)
13. }
14. ORDER BY ?footprint
15. LIMIT 1

```

Explore (:stack_2)			
unreliable multihop			
Subject	Predicate	Object	Context
:stack_2	rdf:type	owl:NamedIndividual	<file:///C:/fakepath/knowledge representation.owl>
:stack_2	rdf:type	:Stack	<file:///C:/fakepath/knowledge representation.owl>
:stack_2	:consumesPower	2	<file:///C:/fakepath/knowledge representation.owl>
:stack_2	:hasFootprint	4257	<file:///C:/fakepath/knowledge representation.owl>
:stack_2	rdfs:comment	"unreliable multihop"	<file:///C:/fakepath/knowledge representation.owl>
:stack_2	:formedOf	:c broadcast	<file:///C:/fakepath/knowledge representation.owl>
:stack_2	:formedOf	:c channel	<file:///C:/fakepath/knowledge representation.owl>
:stack_2	:formedOf	:c multihop	<file:///C:/fakepath/knowledge representation.owl>
:stack_2	:formedOf	:c unicast	<file:///C:/fakepath/knowledge representation.owl>

Figure 31 Querying for a stack which has footprint below a threshold.

7.2.3 Logic based reasoning for CNs

Once there is a formalized world model of the cognitive network, reasoning using this model can be performed. The CRime world model currently supports only simple reasoning. By extending the ontology and increasing the degree of automation for composing services (discussed in Section 6.3.3), more complex reasoning will also be supported.

We provide a simple example of reasoning performed by the current implementation of the ProtoStack tool. When describing CRime modules in the corresponding header files using Turtle statements, one type of statement that we provide is that “a module *defines* a function”. Defines is a property which connects the domain of modules to the range of functions. The inverse statement that “a function *isDefinedBy* a module” is also true but there is no need to provide it. This statement can be inferred by a reasoner based on the CRime ontology. In the ontology, the *isDefinedBy* property has been created as the

inverse of the *defines* property. Such simple reasoning also helps limit the amount of statements the developer needs to write to describe the module.

7.3 Cognitive loops with ProtoStack

The ProtoStack tool can also be used to support the implementation of cognitive loops provided that the necessary planning, learning, deciding, etc. modules are implemented in the CRime library. In order to illustrate a node centric cognition loop, we focused on the routing functionality of the protocol stack. The default Rime and CRime routing uses a shortest path strategy based on the least number of hops to the destination. Each sensor node holds a table which is periodically updated by removing expired routes and adding new ones. We say that a node has a model of the logical network topology and this model is represented in the routing table. As an example, Figure 32 depicts a topology in which A wants to send a message to B. According to A's model, there are two paths to B (drawn with red), each having a cost of 3 (i.e. three hops distance).

Figure 32 A's model of logical connectivity to B (there are two paths from A to B, each travelling through two intermediate hops).

7.3.1 Network layer cognitive loop with cross-layer information

Determining the cost of a path only according to the number of hops may be simplistic in many practical situations, especially when network conditions are subject to change: varying traffic load, varying link quality, etc. One additional parameter that can be considered is the received signal strength indicator (RSSI). When a node selects a path, it can select one with good RSSI. Even more, the node can learn over time which links consistently have good RSSI and prefer them over others. This means that the cost has to have two components: the old estimate and the current finding. This is expressed in the formula below: the most recent belief B_{t+1} consists of the old belief B_t and the weighted finding $kf(hops, RSSI)$. We modelled the current finding as a function of number of hops and RSSI in such way as the higher the RSSI the lower the cost and vice-versa. So in the case of two paths with equal number of hops, the path with better RSSI and better

previous performance will be preferred. Using such a model for the cost, the node learns over time and adapts to changes in the network.

$$B_{t+1} = B_t + kf(hops, RSSI)$$

This example can be readily mapped to the cognition loop as depicted in Figure 33. The environment step corresponds to the wireless channel, the sensing step corresponds to RSSI sensing, the learning step corresponds to the model of the node, the decision step corresponds to the route selection (e.g. lowest cost route selection) and the act step corresponds to the sending of a packet over the wireless channel. The example can be further developed to include the planning step of the loop, however this is beyond the scope of this work.

a)

b)

Figure 33 Network layer cognitive loop with cross-layer information (a) and the composed communication service (b).

In order to permit the implementation of and experimentation with the cognition loop in Figure 33, we complemented CRime with `c_model` source files where all the model specific logic is stored. Let us further consider the example of a stack corresponding to a mesh network as depicted in Figure 20. The `<c_app, c_mesh, c_multihop, c_unicast, c_broadcast, c_channel>` composite service corresponds to the data communication service while the `<c_route_discovery, c_netflood, c_polite, c_broadcast, c_channel>` and `<c_unicast, c_broadcast, c_channel>` composite services correspond to the management service in Figure 33 b. The model creation and update functions are called from the `c_route_discovery` module. Access to the RSSI is straightforward using the Rime packet attributes which are visible to all the modules of the stack. The costs of the routes in the routing table are set based on the estimates in the model. The `c_multihop` module then uses the resulting routing table when sending packets.

Due to the abstractions we introduced, CRime is ready to be extended with additional modules offering additional networking functionality to enable use cases as further discussed in this chapter. For instance, a set of routing modules which use machine learning methods such as reinforcement learning, Bayesian networks, artificial neural

networks, etc. can be added. These methods would be then used to build a model of the network based on which routing decisions can be taken.

7.3.2 Network layer cognitive loop with cognitive radio

Assuming a cognitive radio device able to sense the spectrum and dynamically select available channels, the simple routing problem becomes more complex. First, the routing table has to contain entries for each potentially available channel. This routing table can be filled in using the approach considered in Section 7.3.1 using cross-layer information. The node would also need to have a model for channel availability that is as accurate as possible. A very simple channel estimation model can be built based on energy detection and it can be maintained as a binary vector. At packet transmission time, the routing module needs to consult the channel availability and decide which channel is suitable for transmission.

The corresponding mapping to the cognition loop is depicted in Figure 34. It can be seen that this use case actually involves two cognition loops that interact. The first loop corresponds to the cognitive radio (CR loop in Figure 34). Within this loop, the device scans the available channels, performs energy detection based on RSSI values and decides whether a channel is free or busy. The second loop (CN loop in Figure 34) corresponds to the cognitive radio network and is the same as the one presented in Section 7.3.1 except for the decision step. At decision time, the CN loop retrieves additional information with respect to the channel availability as determined by the CR.

a)

b)

Figure 34 Network layer cognitive loop with cognitive radio (a) and the composed communication service (b).

In order to permit the implementation of and experimentation with the cognition loops

in Figure 34, two approaches can be taken. We use the same example as depicted in Figure 20 and Section 7.3.1.

In the first approach, the assumption is that the CR loop is implemented independently of CRime. In this case the information from the CN loop can be transferred by the CR loop in an easy way, such as the CR loop periodically updating the communication service's pipe structure. In this case, the `c_multihop` module has to take into account the channel availability information as provided by the CR when sending a packet. The `c_model` module may need to be generalized or replaced by an implementation that takes into account the more complex communication scenarios introduced by the CR. For instance, if the radio module can also switch between channels, then this has to be reflected in the network model and routing table.

This approach can be demonstrated by using extended CRime along with the LOG-a-TEC¹⁵ VESNA based CR testbed [102], once the the Contiki support for the testbed¹⁶ is completed. The LOG-a-TEC testbed currently supports simple energy detection based on RSSI samples and can be reprogrammed remotely.

In the second approach, the assumption is that the CR loop is implemented using CRime modules. In such case, additional CRime modules implementing the CR loop need to be implemented. For instance, `c_channel_scan` and `c_channel_availability_model` modules may need to be inserted between the `c_broadcast` and `c_channel` modules of the communication service.

The CN loop can continue using the RSSI to determine the costs of the routes in the routing table, this being independent of the channel availability determined by the CR in the CR loop. Thus the network layer can continue discriminating between routes based on history. The CR example can be relatively complex requiring the node to build routing tables for each possible channel. The node can discriminate between routes on the same channel by using previous experience (i.e. RSSI information). For instance, if a node previously communicating with direct neighbors A and B on channel `c` and experienced better link quality to A, this will reflect in the routing table corresponding to channel `c` and is realized by the CN loop.

Furthermore, with ProtoStack, more complex experiments can be carried out such as validating the delay-sensitive resource management algorithms proposed in [63] and summarized in Sections 7.1.2.1 and 7.1.2.4. The sensing and dynamic access of radio spectrum are already available in the LOG-a-TEC testbed while corresponding CRime modules for the information exchange protocol as well as for the learning strategies need to be added.

¹⁵ <https://crn.log-a-tec.eu/>

¹⁶ Currently the outdoor testbed work for cognitive radio scenarios and uses custom firmware as partly described in [102].

7.3.3 Network layer cognitive loop with radio environment map

The third example showing how ProtoStack can be used in implementation of cognitive loops refers to a scenario where devices that are not capable of spectrum sensing or the local spectrum sensing is not sufficient and needs to be complemented by information available from radio environment mapping services [103]. In such case, the cognition loops presented in Sections 7.3.1 and 7.3.2 should base their decisions also on this complementary information.

Information related to the occupancy of the channels is most easily acquired from radio environment map services through a control channel on which nodes can request information from the remote database, or such information is periodically broadcast. Either way, the device needs an additional management service dedicated to the communication with the REM. The network level cognitive loop needs to take into account spectrum occupancy information at the time of making a decision. This information is obtained from the radio environment map through the management channel and can be complemented by information from the cognitive radio as depicted in Figure 35 a.

a)

b)

Figure 35 Network layer cognitive loop with radio environment map and/or cognitive radio (a) and the corresponding composed communication and management services (b).

In order to permit the implementation of and experimentation with the cognition loop depicted in Figure 35 the discussion in Section 7.3.2 still applies. Additionally, the dedicated management service communicating with the REM has to be implemented. This can be done using existing CRime modules or developing additional modules that are necessary.

7.3.4 Network layer cognitive loop with connectivity broker

The connectivity broker is a concept introduced in [104] that provides abstractions necessary for developing large scale cognitive wireless network environments by enabling joint optimization of spectrum resources. The connectivity broker operates in the control and management planes of the networks and the core concept behind it is the connectivity agent. The connectivity agent represents a particular interest in the brokerage arena such as the interests of a terminal (or a user), a wireless network, or a cluster of collaborating networks. The connectivity agent can support several functionalities such as discovery, optimization, policies, execution, access and repository. These functionalities can also be seen as corresponding to particular states of the cognitive loop. For instance, the discovery functionality is related to the learning state of the loop where the CN/CRN/CR learns about the environment. The optimization functionality can be a component of the learning and/or decision making state. The execution function corresponds to the act state, while the policy support corresponds to the policy state. The repository is a form of knowledge representation while the access control can be seen as rules for restricting access to the knowledge.

Figure 36 illustrates the network layer cognitive loop in the case where the connectivity broker is used. The most important difference from the use case where REMs are used is that some of the functionality of the connectivity agent is implemented by the management stack on the node. As a result, the node is able to sustain bi-directional communication with higher management entities (such as the connectivity brokerage's Unified Network Agent or Composite Network Agent) for dynamic and efficient utilization of resources. For instance, the connectivity agent can use standard connectivity brokerage functionality to send and acquire channel state information from REM services.

Similar to the use case described in Section 7.3.3, ProtoStack can be used to design and assemble the node centric management and communication services. There are two options for the implementation of the management service corresponding to the inter-connectivity agent communication. The implementation can be independent of ProtoStack by using for instance the standard compliant CoAP/UDP/6LowPAN stack available in Contiki. In case the implementation is done using ProtoStack, then the CRime library needs to be extended with modules required by this task. The available implementations of the connectivity broker's inter-agent communications currently use TCP/IP protocol stack and "raw" TCP sockets or ZeroMQ¹⁷ sockets [105]. This approach may prove impractical in some testbeds where TCP/IP connections to all sensor nodes are not

¹⁷ <http://www.zeromq.org/>

available. In such cases, using CRime modules might speed up the implementation.

a)

b)

Figure 36 Network layer cognitive loop with connectivity broker (a) and the corresponding composed communication and management services (b)

Additionally the ProtoStack tool can be used to implement node-level functionality corresponding to the connectivity agent. For instance, the optimization and/or discovery functionality of the connectivity agent can be implemented in a distributed fashion by adding corresponding CRime modules. The experiment descriptions regarding the communication services available on the node that are used by the connectivity brokerage are readily provided by ProtoStack using the declarative language, and can be easily transformed in alternative representations if needed by using a parser.

ProtoStack further extends the scope of the connectivity brokerage beyond cognitive radio level, which currently represents the state of the art [105], onto cognitive (radio) networking level. In this respect, ProtoStack can be seen as a Unified Network Agent or a Composite Network Agent implementing repository, discovery and execution functionalities through the declarative language and the ontology. The repository and discovery functions have been discussed in Section 6.3 and the execution function has been discussed in Section 5.3. While the state of the art addresses cognitive radio problems, ProtoStack addresses problems related to communications services (or protocols) on various layers of the stack. The discovery and optimization components can be designed using additional CRime modules or by plugging additional components to existing feature-less server that connects the components of ProtoStack.

7.4 Summary

This chapter discussed the role of the ProtoStack tool in cognitive networking experimentation. The chapter started by providing a theoretical discussion about what cognitive networks actually are. It then continued by discussing similarities and

differences between cognitive networks, cognitive radio network and cognitive radio. Then we showed how ProtoStack can be used for knowledge representation and logic based reasoning. Next, we showed how ProtoStack can support the cognition loop by incorporating machine learning and decision making modules. Together with Chapter 6 this chapter showed how promising research areas of service oriented networks and cognitive networks can benefit from the proposed framework for dynamic composition of communication services.

8 Conclusions

In this thesis we investigated the dynamic composition of communication services as an alternative to more rigid approaches such as the reference layered Open Systems Interconnect (OSI) architecture of communication networks. Then, we proposed a framework and its reference implementation, ProtoStack, to support dynamic composition of communication services. Finally, we show how two emerging research directions, service oriented networks and cognitive networking can benefit from the dynamic composition of communication services.

We argued that the framework we propose consisting of four components is generic enough to be applied to existing systems as well as future ones. Furthermore, such a framework is suitable to accommodate both expert and non-expert experimenters and that is particularly well suited for remote experimentation. We showed that existing tools supporting the composition of communication services in various segments of communication networks are covered by this framework and demonstrated that ProtoStack, our reference implementation, complies with it and meets all the requirements.

In the investigation of dynamic composition of communication services we used sensor networks as a reference target to prototype and demonstrate the proposed solutions. As a result, the ProtoStack tool we developed supports dynamic composition of services for sensor networks. We showed by means of feedback collection from first time users that ProtoStack increases flexibility and saves time for designing, prototyping and testing of new protocols in realistic scenarios. The tradeoff comes in terms of suboptimal performance of the composed protocol stack. While the studies showed that the tool can speed up design and prototyping of new protocol stack by at least a factor of 2, the CRime library used by ProtoStack tool has 16% larger footprint, it takes 3.5 more time to send a packet and consumes 1.6% more power for doing so. Even though with ProtoStack more resources are consumed by the node, the tradeoff in terms of prototyping speed seems to pay off.

The ProtoStack tool implements the four components of the proposed framework. Two of these components, the CRime module library and the CRime declarative language constitute major contributions of this thesis and were thoroughly described and evaluated.

The last part of the thesis discussed service oriented networks (SON) and cognitive networks (CN) and how the dynamic composition of communication services can be used to further research in this area. Particularly, we showed that ProtoStack can be used to

design and experiment with SONs and CN.

The services in SONs can be described, published and discovered using ProtoStack. Furthermore, the CRime ontology provides the base for semi-automatic and automatic service composition. The knowledge representation and reasoning in CNs is supported by ProtoStack and we demonstrate this using a smart grid and a cognitive ad-hoc network use cases.

ProtoStack also supports the implementation of cognition loops and we discussed selected use cases showing how the CRime library has to be extended to implement them. We also showed how ProtoStack goes beyond the current state of the art with respect to the scope of the connectivity brokerage.

8.1 Future Work

The first important line of future work is focused around the CRime module library. With respect to this, the implementation of a set of learning modules corresponding to widely used machine learning approaches is planned. Additionally, the range of multihop communication services has to be augmented and additional application level services need to be added. These building blocks will increase the power of the module library with respect to experimentally verifying prospective communication services.

The second line of future work is concerned with the tool itself. ProtoStack has to be integrated with remote reprogramming¹⁸ functionality in order to facilitate operation on a large testbed such as LOG-a-TEC testbed deployed in the frame of the CREW project [30]. Then, the experiment configuration has to be exported according to the CREW common data format and the cOntrol and Management Framework¹⁹ should be supported in order to offer Testbed as a Service-like functionality.

The third line of future work refers to investigating the trade-offs in offering distributed versus centralized brokerage functionality in cognitive networks. As we showed in Section 7.3.4, ProtoStack already addresses components of the Connectivity Brokerage framework and furthers the state of the art by extending the scope to other network layers. Investigating how the trade-offs affect the cognitive radio network ecosystem is a promising way ahead.

The fourth line of future work involves logic based configuration of communication services with the goal of achieving as high degree of automation as possible as discussed in Section 6.3.3. This line requires the extension of the CRime ontology and more advanced reasoning among others.

¹⁸ <http://www.crew-project.eu/portal/log-tec-documentation>

¹⁹ <http://mytestbed.net/>

9 Acknowledgements

First, I wish to thank my advisor Asst. Prof. Dr. Mihael Mohorčič, for his patience, support and guidance during my doctoral studies. Then, I would like to thank Prof. Dr. Dunja Mladenić for being the best co-advisor and one of the most goal oriented persons I know. Special thanks go also to Marko Grobelnik and Prof. Dr. Gorazd Kandus for the many nice discussions we had.

I would also like to thank my group of co-authors and collaborators, and my colleagues from the Jožef Stefan Institute, particularly from Sensorlab, for many valuable discussions and contributions. Zoltan Padrah and Marko Mihelin also helped in my efforts to add some finishing touches to the evaluations.

Next, I would like to thank the members of the Evaluation Board of my doctoral committee, Asst. Prof. Dr. Aleš Švigelj, Prof. Dr. Luiz DaSilva and Dr. Michael Witbrock for their valuable comments and remarks.

Special thanks go to Blaž and Davor for all the nice time we spent together.

Parintii mei, Geta si Mihai, au avut un rol deosebit in viata si cariera mea. Lucrarea aceasta nu ar fi fost finalizata fara ajutorul lor.

Hvala tudi Anici in Franciju.

This work was supported by the Slovenian Research Agency and the European Community under CREW Cognitive Radio Experimentation World (Grant no 258301).

10 References

- [1] Sherwood, R.; Gibb, G.; Yap, K. K.; Appenzeller, G.; Casado, M.; McKeown, N.; Parulkary, G. Can the Production Network Be the Testbed? In: *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. 1–6 (USENIX Association, Berkeley, CA, USA, 2010).
- [2] Hutchinson, N. C.; Peterson, L. L. The x-Kernel: An Architecture for Implementing Network Protocols, *IEEE Transactions on Software Engineering* **17**, 64–76 (1991).
- [3] Kohler, E.; Morris, E.; Chen, B.; Jannotti, J; and Kaashoek, M. F. The Click Modular Router. *ACM Transactions on Computer Systems* **18**, 263–297 (2000).
- [4] MacKenzie, P. Software and reconfigurability for software radio systems, Ph.D dissertation (University of Dublin, Trinity College, Ireland, 2004).
- [5] McKeown, N.; Anderson, T.; Balakrishnan, H.; Parulkar, G.; Peterson, L.; Rexford, J.; Shenker, S; and Turner, J. OpenFlow: enabling innovation in campus networks. *SIGCOMM Computer Communications Review* **38**, 69–74 (2008).
- [6] Handigol, N.; Heller, B.; Jeyakumar, V., Mazieres, D., McKeown, N. Where is the Debugger for my Software-Defined Network? In: *Proceedings of Hot Topics in Software Defined Networking*. 55–60 (ACM, Helsinki, Finland, 2012).
- [7] Bavier, A.; Feamster, N.; Huang, M.; Peterson, L.; Rexford, J. In VINI Veritas: Realistic and Controlled Network Experimentation. In: *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications*. 3–14 (ACM, Pisa, Italy, 2006).
- [8] FIRE - Future Internet Research & Experimentation. <http://cordis.europa.eu/fp7/ict/fire/> (accessed: October 2012).
- [9] GENI.net Global Environment for Network Innovations. <http://www.geni.net> (accessed: October 2012).
- [10] Raychaudhuri, D.; Seskar, I.; Ott, M.; Ganu, S.; Ramachandran, K.; Kremo, H.; Siracusa, R.; Liu, H.; and Singh, M. Overview of the ORBIT Radio Grid Testbed for Evaluation of Next-Generation Wireless Network Protocols. In: *Proceedings of IEEE Wireless Communications and Networking Conference*. 1664–1669 (IEEE, New Orleans, LA, USA, 2005).
- [11] Bouckaert, S.; Vandenberghe, W.; Jooris, B.; Moerman, I.; Demeester, P. The w-iLab.t testbed *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering* **46**, 145– 154 (2011).
- [12] Handziski, V.; Köpke, A.; Willig, A; and Wolisz, A. TWIST: A Scalable and Reconfigurable Testbed for Wireless Indoor Experiments with Sensor Network. In:

- Proceedings of the 2nd International Workshop on Multihop Ad Hoc Networks: from Theory to Reality*. 63–70 (ACM, Florence, Italy, 2006).
- [13] Abdallah, A. S.; MacKenzie, A. B.; DaSilva, L. A.; Thompson, M. S. On Software Tools and Stack Architectures for Wireless Network Experiments. In: *Proceedings of the IEEE Wireless Communications and Networking Conference*. 2131–2136 (IEEE, Cancun, Mexico, 2011).
 - [14] Feldmann, A.; Internet clean-slate design: what and why?, *ACM SIGCOMM Computer Communication Review* **37**, 59–64 (2007).
 - [15] Mitola, J. Cognitive Radio – An Integrated Agent Architecture for Software Defined Radio, PhD Dissertation (Royal Institute of Technology, Kista, Sweden, 2000).
 - [16] Thomas, R. W. Cognitive Networks, PhD Dissertation (Virginia Polytechnic and State University, Blacksburg, Virginia, 2007).
 - [17] Fortuna, C.; Mohorcic, M.; Trends in the development of communication networks: Cognitive networks. *Computer Networks* **53**, 1354–1376 (2009).
 - [18] Polastre, J.; Hui, J.; Levis, P.; Zhao, J.; Culler, D.; Shenker, S.; and Stoica, I. A unifying link abstraction for wireless sensor networks. In: *Proceedings of the Third ACM Conference on Embedded Networked Sensory Systems*. 76–89 (ACM, San Diego, CA, USA, 2005).
 - [19] Ee, C. T.; Fonseca, R.; Kim, S.; Moon, D.; Tavakoli, A.; Culler, D.; Shenker, S.; Stoica, I. A Modular Network Layer for Sensornets. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. 249–262 (ACM, Seattle, Washington, USA, 2006).
 - [20] Tavakoli, A.; Chu, D.; Hellerstein, J. M.; Levis, P.; Shenker, S. A Declarative Sensornet Architecture, *ACM SIGBED Review - Special issue on the workshop on wireless sensor network architecture* **4**, 55–60 (2007).
 - [21] Dunkels, A.; Osterlind, F.; He, Z., An Adaptive Communication Architecture for Wireless Sensor Networks. In: *Proceedings of the ACM Conference on Embedded Networked Sensor Systems*, 335–349 (Sydney, Australia, 2007).
 - [22] Touch, J. D.; Pingali, V. K. The RNA Metaprotocol. In: *Proceeding of the International Conference on Computer Communication Networks*. 1–6 (IEEE, St. Thomas, US Virgin Islands, 2008).
 - [23] Bouabene, G.; Jelger, C.; Tschudin, C.; Schmid, C.; Keller, A.; May, M. The Autonomic Network Architecture, *IEEE Journal of Selected Areas in Communications* **28**, 4–14 (2010).
 - [24] Zimmermann, H. OSI Reference Model--The ISO Model of Architecture for Open Systems Interconnection, *IEEE Transactions on Communications* **28**, 425–432 (1980).
 - [25] Giambene, G.; and Kota, S. Cross-Layer Protocol Optimization for Satellite Communication Networks: A Survey, *International Journal of Satellite Communications and Networking* **24**, 323–341 (2006).

- [26] Srivastava, V.; Motani, M. Cross-Layer Design: A Survey and the Road Ahead, *IEEE Communications Magazine* **43**, 112–119 (2005).
- [27] Carneiro, G.; Ruela, J.; and Ricardo, M. Cross-Layer Design in 4G Wireless Terminals, *IEEE Wireless Communications* **11**, 7–13 (2004).
- [28] Mahmoud, Q. Cognitive Networks – Towards Self-Aware Networks, (John Wiley & Sons, 2007. ISBN: 9780470061961).
- [29] Clark, D. D.; Partridge, C.; Ramming, J. C.; and Wroclawski, J.T. A Knowledge Plane for the Internet. In: *Proceedings of the ACM Special Interest Group on Data Communication*, 3–10 (ACM, Karlsruhe, Germany, 2003).
- [30] CREW project, <http://www.crew-project.eu/> (accessed: August 8, 2012).
- [31] Dunkels, A.; Gronvall, B.; Voigt, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In: *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*. 455–462 (IEEE, Tampa, Florida, USA, 2004).
- [32] Resource Description Framework (RDF) Model and Syntax Specification, <http://www.w3.org/TR/PR-rdf-syntax/> (accessed: October 2012).
- [33] Web Ontology Language (OWL), <http://www.w3.org/TR/owl-features/> (accessed October 2012).
- [34] Matuszek, C.; Cabral, J.; Witbrock, M.; and DeOliveira, J. An introduction to the syntax and content of Cyc. In: *Proceedings of the Twenty-first National Conference on Artificial Intelligence*, 44–49 (AAAI Press, Menlo Park, CA, USA, 2006).
- [35] IEEE P802.21/D11.0 Draft Standard for Local and Metropolitan Area Networks: Media Independent Handover Services (2008).
- [36] Shepard, B.; Matuszek, C.; Fraser, C.B.; Wechtenhiser, W.; Crabbe, D.; Gungordu, Z.; Jantos, J.; Hughes, T.; Lefkowitz, L.; Witbrock, M.; Lenat, D.; and Larson, E. A knowledge-based approach to network security: applying Cyc in the domain of network risk assessment. In: *Proceedings of the 17th conference on Innovative applications of artificial intelligence*. 1563–1568 (AAAI Press, Pittsburgh, Pennsylvania, USA, 2005).
- [37] Fortuna, C.; Mohorcic, M.; Dynamic composition of services for end-to-end information transport. *IEEE Wireless Communications Magazine* **16**, 56–62 (2009).
- [38] Smolnikar, M.; Fortuna, C.; Vučnik, M.; Mihelin, M.; Mohorčič, M. Wireless Sensor Network Testbed on Public Lighting Infrastructure. In: *Proceedings of the Second International Workshop on Sensing Technologies in Agriculture, Forestry and Environment* 6–7 (Workshop papers, Belgrade, Serbia, 2011).
- [39] Vanhie-Van Gerwen, J.; Bouckaert, S.; Moerman, I.; Demeester, P. Benchmarking for Wireless Sensor Networks. In: *Proceedings of the Fifth International Conference on Sensor Technologies and Applications*, 134–139 (French Riviera, Nice/Saint Laurent du Var, France, 2011).
- [40] Berners-Lee, T.; Hendler J.; and Lassila, O. The Semantic Web. *Scientific American*

- Magazine* **284**, 28–37 (2001).
- [41] Magedanz, T.; Blum, N.; Dutkowski, S. Evolution of SOA Concepts in Telecommunications, *IEEE Computer* **40**, 46–50 (2007).
 - [42] Papazoglou, M. P.; van den Heuvel, W.-J. Service oriented architectures: approaches, technologies and research issues, *Very Large Databases Journal* **16**, 389–415 (2007).
 - [43] Siegele, L. A Special Report on Corporate IT, *The Economist* **389** (2008).
 - [44] Saha, D.; Sahu, S.; and Shaikh, A. A Service Platform for Online Games. In: *Proceedings of Network and Systems Support for Games*, 180–184 (ACM, Redwood City, CA, USA, 2003).
 - [45] Ben Yahia, I. G.; Bertin, E.; and Crespi, N. Ontology based Management Systems for the Next Generation Services: State-of-the-Art. In: *Proceedings of the third International Conference Network and Services*. 40–46 (IEEE, Athens, Greece, 2007).
 - [46] Trang Nguyen, T. M.; Boukhatem, N.; and Puille, G. COPS-SLS Usage for Dynamic Policy-based QoS Management over Heterogeneous IP Networks, *IEEE Network* **17**. 44–50 (2003).
 - [47] Mandic, P.; and Jaehnert, J. Service Oriented Network Selection. In: *Proceedings of the International Symposium on Wireless Communication Systems*, 138–143 (IEEE, Reykjavik, Iceland, 2008).
 - [48] Dutta, A.; Madhani, S.; Zhang, T.; Ohba, Y.; Taniuchi, K.; Schulzrinne, H. Network Discovery Mechanisms for Fast-handoff, In: *Proceedings of the International Conference on Broadband Communications, Networks, and Systems*. 1–11 (IEEE, San Jose, CA, USA, 2006).
 - [49] Codd, E. F. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM* **13** (1970).
 - [50] Fortuna, C.; Mohorcic, M.; Filipic, B. Multiobjective optimization of service delivery over a heterogeneous wireless access system. In: *Proceedings of the IEEE International Symposium on Wireless Communication Systems*. 133–137 (IEEE, Reykjavik, Iceland, 2008).
 - [51] Fortuna, C.; Mohorcic, M. A local knowledge base for the media independent information system. In: *Proceedings of the second future internet symposium* 15–24 (Springer-Verlag, Berlin, Germany, 2009).
 - [52] Fortuna, C.; Ivan, B.; Padrah, Z.; Bradesko, L.; Fortuna, B.; Mohorcic, M. Demonstration: Wireless access network selection enabled by semantic technologies. *International Semantic Web Conference*, (Chantilly, Va, USA, 2009).
 - [53] Moraru, A.; Fortuna, C.; Fortuna, B.; Slavescu, R. R. A hybrid approach to QoS-aware web service classification and recommendation. In: *Proceedings of the IEEE 5th International Conference on Intelligent Computer Communications and Processing*. 343–346 (IEEE, Cluj-Napoca, Romania, 2009).

- [54] Fortuna, C.; Mohorcic, M. A local knowledge base for service oriented access network selection. In: *Proceedings of the SIGCOMM Conference and co-located Workshops MobiHel*. 65–66 (ACM, Barcelona, Spain, 2009).
- [55] Fortuna, C.; Mohorcic, M. Advanced access architecture for efficient service delivery in heterogeneous wireless networks. In: *Proceedings of the international Workshop on Cognitive Networks and Communications Communications*. 1173–1177 (IEEE, Hangzhou, China, 2008).
- [56] Thomas, R. W.; DaSilva, L. A.; MacKenzie, A. B. Cognitive Networks. In: *Proceedings of the first IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks*. 352–360 (IEEE, Baltimore, MD, USA, 2005).
- [57] Balamuralidhar, P.; Prasad, R. A context driven architecture for cognitive nodes, *Wireless Personal Communications* **45**, 423–434 (2008).
- [58] Jamalipour, A. Cognitive Heterogeneous Mobile Networks, *IEEE Wireless Communications*, **15** (2008).
- [59] Davis, R.; Shrobe, H.; and Szolovits, P. What is a Knowledge Representation?, *AI Magazine* **14**, 17–33 (1993).
- [60] Russell, S.; Norvig, P. Artificial Intelligence: A Modern Approach 2nd Edition, (Prentice Hall, 2002, ISBN 0137903952).
- [61] Devitt, A.; Danev, B.; and Matusikova, K. Contructing Bayesian networks automatically using ontologies. In: *Proceedings of Formal Ontologies Meets Industry* (IOS Press, Trento, Italy, 2006).
- [62] Lehtihet, E.; Strassner, J.; Agoulmine, N.; and O’Foghlu, M. Ontology-Based Knowledge Representation for Self-governing Systems, *Large Scale Management of Distributed Systems* **4269**, 74–85 (Springer Berlin/Heidelberg, 2006, ISBN 9783540476597).
- [63] Shiang, H.-P.; van der Saar, M. Delay-Sensitive Resource Management in Multihop Cognitive Radio Networks. In: *Proceedings of the Third IEEE Symposium on New Frontiers in Dynamic Spectrum Access Networks*. 1–12 (IEEE, Chicago, Illinois, 2008).
- [64] Friend, D. H.; MacKenzie, A. B. Environmentally-Friendly Secondary Network Topology Control for Minimizing Outage Potential. In: *Proceedings of the Third IEEE Symposium on New Frontiers in Dynamic Spectrum Access Networks*. 1–8 (IEEE, Chicago, Illinois, 2008).
- [65] Penna, F.; Garelo, R.; Spirito, M. A. Distributed Inference of Channel Occupation Probabilities in Cognitive Networks via Message Passing. In: *Proceedings of the Third IEEE Symposium on New Frontiers in Dynamic Spectrum Access Networks*. 1–11 (IEEE, Singapore, 2010).
- [66] Balamuralidhar, P.; Prasad, R. Self-Configuration and Optimization for Cognitive Networked Devices, *Wireless Personal Communications* **59**, 471–486 (2011).
- [67] Dimitrakopoulos, G.; Tsagkaris, K.; Demestichas, P.; Adamopoulou, E.;

- Demestichas, P. A Management Scheme for Distributed Cross-Layer Reconfigurations in the Context of Cognitive B3G Infrastructures, *Computer Communications* **30**, 3807–3822 (Elsevier, 2007).
- [68] Demestichas, P.; Stavroulaki, V.; Boskovic, D.; Lee, A.; and Strassner, J. m@ANGEL: Autonomic Management Platform for Seamless Cognitive Connectivity to the Mobile Internet, *IEEE Communications Magazine* **6**, 118–127 (2006).
- [69] End-To-End Efficiency. <https://www.ict-e3.eu/project/overview/overview.html> (accessed: December 2012).
- [70] Thomas, R. W.; DaSilva, L. A.; MacKenzie, A. B. Cognitive Networks. In: *Proceedings of the First IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks*. 352–360 (IEEE, Baltimore, MD, USA, 2005).
- [71] Thathachar, M. A. L.; Sastry, P. S. Networks of Learning Automata: Techniques for Online Stochastic Optimization, 1st Ed, (Springer, 2003, ISBN: 1402076916).
- [72] Lewis, D.; Keeney, J.; O'Sullivan, D.; Guo, S. Towards a Managed Extensible Control Plane for Knowledge-Based Networking. In: *17th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*. 98–111 (IFIP/IEEE, Dublin, Ireland, 2006).
- [73] Guo, S.; Keeney, J.; O'Sullivan, D.; Lewis, D. Adaptive Semantic Interoperability Strategies for Knowledge Based Networking. In: *Proceedings of the 3rd International Workshop on Scalable Semantic Web Knowledge Base Systems*. 1187–1199 (Springer-Verlag Berlin, Heidelberg, Vilamoura, Algarve, Portugal, 2007).
- [74] Mitola, J. Cognitive Radio Policy Languages. In: *Proceedings of the IEEE International Conference on Communications*. 1–4 (IEEE, Dresden, Germany, 2009).
- [75] Zhou, C.; Chia, L.-T.; and Lee, B.-S. Semantics in Service Discovery and QoS Measurement, *IEEE IT Professional Magazine* **7**, 29–34 (2005).
- [76] Dobson, G.; Sanchez-Macian, A. Towards Unified QoS/SLA Ontologies. In: *Proceedings of the IEEE Services Computing Workshops*, 169–174 (IEEE, Chicago, USA, 2006).
- [77] Zhou, H. K. Pung, L. H. Ngoh, Towards Semantic Modeling for QoS Specification. In: *Proceedings of the IEEE Conference on Local Computer Networks*, 361–368 (IEEE, Tampa, Florida, USA, 2006).
- [78] Lee, G.; Faratin, P.; Bauer, S.; Wroclawski, J. A user-guided cognitive agent for network selection in pervasive computing environments, In: *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications*. 219–228 (IEEE, Orlando, Florida, USA, 2004).
- [79] Lee, G.; Faratin, P.; Bauer, S.; Wroclawski, J. Learning user preferences for wireless service provisioning. In: *Proceedings of the third International Joint Conference on Autonomous agents and Multiagent Systems*. 480–487 (NY, USA,

- 2004).
- [80] Gelenbe, E.; Lent, R.; Nunez, A. Self-aware networks and QoS, *Proceedings of the IEEE* **92**, 1478–1489 (2004).
 - [81] Kodeswaran, S. B.; Ratsimor, O.; Joshi, A.; Perich, F. Utilizing Semantic Tags for Policy Based Networking. In: *Proceedings of the IEEE Global Telecommunications Conference*, 1954–1958 (IEEE, Washington DC, USA, 2007).
 - [82] Undercoffer, J.; and Pinkston, J. Modeling Computer Attacks: A Target-Centric Ontology for Intrusion Detection, *CADIP Research Symposium*, (Baltimore, USA, 2002). <http://www.csee.umbc.edu/cadip/2002Symposium/Ont-for-IDS.pdf> (accessed: December 2012).
 - [83] Gorodetsky, V. I.; Kotenko, I. V.; Michael, J. B. Multi-agent Modeling and Simulation of Distributed Denial of Service Attacks on Computer Networks. In: *Proceedings of the Third International Conference on Navy and Shipbuilding Nowadays* (Krylov Shipbuilding Research Institute, St. Petersburg, Russia, June 2003).
 - [84] The DARPA Agent Markup Language Homepage, <http://www.daml.org/> (accessed: December 2012).
 - [85] OMG Object Management Group, <http://www.omg.org/> (accessed: December 2012).
 - [86] Jena – A Semantic Web Framework for Java, <http://jena.sourceforge.net/> (accessed: December 2012).
 - [87] Pellet, <http://pellet.owldl.com/> (accessed: December 2012).
 - [88] FaCT, <http://owl.man.ac.uk/factplusplus/> (accessed: December 2012).
 - [89] RacerPro, <http://www.racer-systems.com/> (accessed: December 2012).
 - [90] Jess, the Rule Engine for Java Platform, <http://herzberg.ca.sandia.gov/> (accessed: December 2012).
 - [91] Cyc, <http://www.cyc.com/> (accessed: December 2012).
 - [92] Nilsson, N. J. Probabilistic Logic, *Artificial Intelligence* **28**, 71–87 (1986).
 - [93] Mahonen, P.; Petrova, M.; Riihijarvi, J.; and Wellens, M.; Cognitive Wireless Networks: Your Network Just Became a Teenager. In: *Proceedings of the 25th Conference on Computer Communications* (IEEE, Barcelona, Spain, 2006).
 - [94] Meshkova, E.; Riihijarvi, J.; Mahonen, P. Learning in Cross-layer Wireless Network Optimization. In: *Proceedings of the Learning for Networking Workshop, in conjunction with SIGMETRICS/Performance* (Seattle, USA, 2009).
 - [95] Sutton, P.; Doyle, L. E.; Nolan, K. E. A Reconfigurable Platform for Cognitive Networks. In: *Proceedings of the first International Conference on Cognitive Radio Oriented Wireless Networks and Communications* 1–5 (IEEE, Mykonos Island, Greece, 2006).
 - [96] Xie, J.; Howitt, I.; Raja, A. Cognitive Resource Management using multi-agent systems. In: *Proceedings of the 4th IEEE Consumer Communications and*

- Networking Conference*. 1123–1127 (IEEE, Las Vegas, Nevada, 2007).
- [97] Marojevic, V.; Reves, X.; Gelonch, A. Cooperative Resource Management in Cognitive Radio. In: *Proceedings of IEEE International Conference on Communications*. 5939–5944 (IEEE, Glasgow, Scotland, 2007).
 - [98] Raychaudhuri, D.; Mandayam, N. B.; Evans, J. B.; Ewy, B. J.; Seshan, S.; Steenkiste, P. CogNet – an architectural foundation for experimental cognitive radio networks within the future internet. In: *Proceedings of the 1st ACM/IEEE International Workshop on Mobility in the Evolving Internet Architecture*. 11–16 (ACM/IEEE, San Francisco, California, 2006).
 - [99] Pitchaimani, M.; Ewy, B. J.; Evans, J. B. Evaluating Techniques for Network Layer Independence in Cognitive Networks, In: *Proceedings of IEEE International Conference on Communications* 6527–6531 (IEEE, Glasgow, Scotland, 2007).
 - [100] The Future of the Internet: A Compendium of European Projects on ICT Research Supported by the EU 7th Framework Programme for RTD, European Communities, 2008
ftp://ftp.cordis.europa.eu/pub/fp7/ict/docs/ch1-g848-280-future-internet_en.pdf (accessed: December 2012).
 - [101] Cesana, M.; Cuomo, F.; Ekici, E. Routing in cognitive radio networks: Challenges and solutions. *Ad Hoc Networks* **9**. 228–248 (2011).
 - [102] Padrah, Z. Distributed Spectrum Sensing in Unlicensed Bands Using the VESNA Platform, *MSc thesis* (Jožef Stefan International Postgraduate School, Ljubljana, Slovenia, 2012).
 - [103] Zhao, Y. Enabling Cognitive Radios through Radio Environment Maps, *PhD thesis* (Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 2007).
 - [104] Rabaey, J.; Wolisz, A.; Ercan, A. O.; Araujo, A.; Burghardt, F.; Mustafa, S.; Parsa, A.; Pollin, S.; Wang, I-H.; Malagon, P. Connectivity Brokerage – Enabling Seamless Cooperation in Wireless Networks, *White Paper* (Berkley Wireless Research Center, Berkley, California, 2010).
 - [105] Finn, D.; Tallon, J.; DaSilva, L.; Mehari, M.; Liu, W.; Bouckaert, S.; Moerman, I.; Heller, C.; Thao, S.; Sanchez, A.; Depierre, D.; van Wesemael, P.; Desmet, M.; Hauer, J.; Chwalisz, M.; Michailow, N.; Padrah, Z. Initial report on demand driven federation functionality, *CREW project deliverable 5.2* (2012).

Index of Figures

Figure 1 The development cycle of new protocols and algorithms (adapted from [1])	1
Figure 2 The seven layers of the Open Systems Interconnect reference model.	9
Figure 3 The communication according to the Open Systems Interconnect reference model. Two hosts, A and B communicate using protocols at different layers of the OSI model.	10
Figure 4 Trends in breaking the layered architecture of communication systems.	11
Figure 5 The four components of the framework for the composition of services.	15
Figure 6 ProtoStack: an implementation of the framework for composing communication services. The implementation addresses the sensor networks domain.	18
Figure 7 Example of CRime stacks: (a) 1 channel – 1 stack example and (b) 3 channel – 3 stack.	24
Figure 8 Tree model with one pipe per stack which is used for the CRime implementation.	24
Figure 9 Dependency graph in (a) Rime and (b) CRime (right). It can be seen that with the CRime approach, there are more possible combinations of modules which can form a stack (11 vs 6).	30
Figure 10 Illustration of the way the Rime unicast stack is implemented.	31
Figure 11 Illustration of the way a possible CRime unicast stack is implemented.	31
Figure 12 CRime ontology (a) classes, (b) properties.	42
Figure 13 Example CRime statements describing the c_abc module.	43
Figure 14 Example Turtle statements describing the c_abc module.	43
Figure 15 Snapshot of the RDF graph describing the c_abc module.	44
Figure 16 Snapshot of the ProtoStack workbench.	45
Figure 17 SPARQL query which retrieves the modules and their descriptions for populating the toolbox of the workbench.	46
Figure 18 Example of a 1 channel – 1 stack configuration using the ProtoStack workbench.	46
Figure 19 Example of a stack which uses the triggering mechanism.	47
Figure 20 Example of 3 channel – 3 stack configuration using the ProtoStack workbench.	48
Figure 21 Validity checking example on a three module stack: making sure that the top interfaces of lower modules need to be compatible with the bottom interfaces of higher modules.	50
Figure 22 End-to-end service composition in a SON.	60
Figure 23 Published CRime service descriptions.	63
Figure 24 SPARQL formulation of a query to retrieve all instances of type Function.	65

Figure 25 OWLIM compatible rule that defines transitivity relationship.....	65
Figure 26 SPARQL query that retrieves all multihop modules thus narrowing down the search space for the user.	67
Figure 27 CycKB rules for scope based stack composition.....	68
Figure 28 The cognition loop.....	80
Figure 29 Snapshot of the CRime RDF graph.	85
Figure 30 Querying for a reliable stack which consumes little power.....	87
Figure 31 Querying for a stack which has footprint below a threshold.	88
Figure 32 A's model of logical connectivity to B (there are two paths from A to B, each travelling through two intermediate hops).	89
Figure 33 Network layer cognitive loop with cross-layer information (a) and the composed communication service (b).	90
Figure 34 Network layer cognitive loop with cognitive radio (a) and the composed communication service (b).....	91
Figure 35 Network layer cognitive loop with radio environmetal map and/or cognitive radio (a) and the corresponding composed communication and management services (b).	93
Figure 36 Network layer cognitive loop with connectivity broker (a) and the corresponding composed communication and management services (b).....	95

Index of Tables

Table 1: Comparison of related experimentation tools.	4
Table 2: Comparison of related approaches to modular stacks.	6
Table 3: Comparison of tools which support the composition of communication services from the perspective of the defined framework.	19
Table 4: Comparison of tools which support composition of communication services from the perspective of the requirements.	20
Table 5: Comparison of related experimentation tools.	21
Table 6: Comparison of related approaches to modular stacks.	21
Table 7: Selected Rime communication primitives (compiled based on the Contiki 2.5 Rime).	28
Table 8: Selected CRime communication primitives.	28
Table 9: Comparison of Rime and CRime components with respect to code size (.text), initialized static variables (.data) and uninitialized static variables (.bss). All values are in bytes.	34
Table 10 The code size (.text), initialized static variables (.data) and uninitialized static variables (.bss) of five Rime example applications. All values are in bytes.	35
Table 11 The code size (.text), initialized static variables (.data) and uninitialized static variables (.bss) of five CRime example applications (the applications are compatible and do the same thing as the Rime applications from). All values are in bytes.	35
Table 12 The cost of CRime example application in terms of code size (.text), initialized static variables (.data) and uninitialized static variables (.bss). Data compiled based on the values in Table 10 and Table 11. All values are in bytes.	35
Table 13:CRime vs Rime processing speed.	36
Table 14:The cost of stack and amodule abstractions in terms of processing speed.	37
Table 15:CRime vs Rime energy consumption.	38
Table 16: The ProtoStack language model.	41
Table 17: The study group profile: in terms of education, research interest and familiarity with basic communication primitives.	54
Table 18:The study group profile: in terms of experience with Contiki, Rime, TCP/IP, modular stacks and composition of services.	54
Table 19:User rating of the ProtoStack tool in terms of intuitiveness, completeness and usefulness.	55
Table 20:Time spent on tasks and numbers of failed attempts to save the stack by the study group.	55
Table 21: Comparison of the logical components of SOA and OSI models.	58
Table 22: Key similarities and differences between SOA and OSI models.	59

Table 23 The mapping of Crime concepts in the Cyc knowledge base.	67
Table 24 Summary of surveyed approaches to knowledge representation and reasoning.	77
Table 25 Summary of surveyed approaches to cognition loops.....	84

Index of Algorithms

Algorithm 1: The interface of the abstract module.....	25
Algorithm 2: The code for the <code>c_open</code> primitive.....	26
Algorithm 3: The pipe data structure.....	26
Algorithm 4: The <code>stack_open</code> function.	27

Appendix A: Rime Communication Primitives

No.	Name	Data structure	Function	Input	Packet attributes
1	abc	<pre> struct abc_conn { struct channel channel; const struct abc_callbacks *u; }; </pre>	<pre> abc_open abc_close abc_send abc_input abc_sent </pre>	<pre> (struct abc_conn *c, uint16_t channel, const struct abc_callbacks *u) (struct abc_conn *c) (struct abc_conn *c) (struct channel *channel) (struct channel *channel, int status, int num_tx) </pre>	ABC_ATTRIBUTES
2	broadcast²⁰	<pre> struct broadcast_conn { struct abc_conn c; const struct broadcast_callbacks *u; }; </pre>	<pre> broadcast_open broadcast_close broadcast_send </pre>	<pre> (struct broadcast_conn *c, uint16_t channel, const struct broadcast_callbacks *u) (struct broadcast_conn *c) (struct broadcast_conn *c) </pre>	BROADCAST_ATTRIBUTES { PACKETBUF_ADDR_SENDER, PACKETBUF_ADDRSIZE }, ABC_ATTRIBUTES
3	stbroadcast	<pre> struct stbroadcast_conn { struct broadcast_conn c; </pre>	<pre> stbroadcast_open </pre>	<pre> (struct stbroadcast_conn *c, uint16_t channel, </pre>	

²⁰ Also sometimes referred as ibc (Identified Broadcast) in the literature

	struct ctimer t; struct queuebuf *buf; const struct stbroadcast_callbacks *u; };		const struct stbroadcast_callbacks *u)	
		stbroadcast_close	(struct stbroadcast_conn *c)	
		stbroadcast_send_stubborn	(struct stbroadcast_conn *c, clock_time_t t)	
		stbroadcast_cancel	(struct stbroadcast_conn *c)	
		stbroadcast_set_timer	(struct stbroadcast_conn *c, clock_time_t t)	
4	struct trickle_conn { struct broadcast_conn c; const struct trickle_callbacks *cb; struct ctimer t, interval_timer, first_transmission_timer; struct pt pt; struct queuebuf *q; clock_time_t interval; uint8_t seqno; uint8_t interval_scaling; uint8_t duplicates; };	trickle_open	(struct trickle_conn *c, clock_time_t interval, uint16_t channel, const struct trickle_callbacks *cb);	TRICKLE_ATTRIBUTES { PACKETBUF_ATTR_EPACKET_ID, PACKETBUF_ATTR_BIT * 8 }, BROADCAST_ATTRIBUTES
		trickle_close	(struct trickle_conn *c)	
		trickle_send	(struct trickle_conn *c)	
5	struct polite_conn { struct abc_conn c; const struct polite_callbacks *cb;	polite_open	(struct polite_conn *c, uint16_t channel, const struct polite_callbacks *cb)	POLITE_ATTRIBUTES ABC_ATTRIBUTES

²¹ Reliable single-source multihopflooding

	<pre> struct ctimer t; struct queuebuf *q; uint8_t hdrsize; }; </pre>	<pre> polite_close polite_send polite_cancel </pre>	<pre> (struct polite_conn *c) (struct polite_conn *c, clock_time_t interval, uint8_t hdrsize) (struct polite_conn *c) </pre>	
6	<pre> struct ipolite_conn { struct broadcast_conn c; const struct ipolite_callbacks *cb; struct ctimer t; struct queuebuf *q; uint8_t hdrsize; uint8_t maxdups; uint8_t dups; }; </pre>	<pre> ipolite_open ipolite_close ipolite_send ipolite_cancel </pre>	<pre> (struct ipolite_conn *c, uint16_t channel, uint8_t maxdups, const struct ipolite_callbacks *cb) (struct ipolite_conn *c) (struct ipolite_conn *c, clock_time_t interval, uint8_t hdrsize) (struct ipolite_conn *c) </pre>	<pre> IPOLITE_ATTRIBUTES IBC_ATTRIBUTES </pre>
7	<pre> struct netflood_conn { struct ipolite_conn c; const struct netflood_callbacks *u; clock_time_t queue_time; rimeaddr_t last_originator; uint8_t last_originator_seqno; }; </pre>	<pre> netflood_open netflood_close netflood_send </pre>	<pre> (struct netflood_conn *c, clock_time_t queue_time, uint16_t channel, const struct netflood_callbacks *u) (struct netflood_conn *c) (struct netflood_conn *c, uint8_t seqno) </pre>	<pre> NETFLOOD_ATTRIBUTES { PACKETBUF_ADDR_ESENDER, PACKETBUF_ADDRSIZE }, \ { PACKETBUF_ATTR_HOPS, PACKETBUF_ATTR_BIT * 5 }, \ { PACKETBUF_ATTR_EPACKET_ID, PACKETBUF_ATTR_BIT * 4 }, IPOLITE_ATTRIBUTES </pre>

8	unicast ²²	<pre> struct unicast_conn { struct broadcast_conn c; const struct unicast_callbacks *u; }; </pre>	unicast_open	(struct unicast_conn *c, uint16_t channel, const struct unicast_callbacks *u)	UNICAST_ATTRIBUTES { PACKETBUF_ADDR_RECEIVER, PACKETBUF_ADDRSIZE }, BROADCAST_ATTRIBUTES
			unicast_close	(struct unicast_conn *c)	
			unicast_send	(struct unicast_conn *c, const rimeaddr_t *receiver)	
9	stunicast	<pre> struct stunicast_conn { struct unicast_conn c; struct ctimer t; struct queuebuf *buf; const struct stunicast_callbacks *u; rimeaddr_t receiver; }; </pre>	stunicast_open	(struct stunicast_conn *c, uint16_t channel, const struct stunicast_callbacks *u)	STUNICAST_ATTRIBUTES UNICAST_ATTRIBUTES
			stunicast_close	(struct stunicast_conn *c)	
			stunicast_send_stubborn	(struct stunicast_conn *c, const rimeaddr_t *receiver, clock_time_t rxmittime)	
			stunicast_cancel	(struct stunicast_conn *c)	
			stunicast_send	(struct stunicast_conn *c, const rimeaddr_t *receiver);	
			stunicast_set_timer	(struct stunicast_conn *c, clock_time_t t);	
			*stunicast_receiver	(struct stunicast_conn *c)	
10	runicast	<pre> struct runicast_conn { struct stunicast_conn c; const struct runicast_callbacks *u; uint8_t sndnxt; }; </pre>	runicast_open	(struct runicast_conn *c, uint16_t channel, const struct runicast_callbacks *u)	RUNICAST_ATTRIBUTES { PACKETBUF_ATTR_PACKET_TYPE, PACKETBUF_ATTR_BIT }, { PACKETBUF_ATTR_PACKET_ID, PACKETBUF_ATTR_BIT *
			runicast_close	(struct runicast_conn *c)	

²² Also sometimes referred as uc (Best-Effort Singlehop Unicast) in the literature

	uint8_t is_tx; uint8_t rxmit; uint8_t max_rxmit; };	runicast_send runicast_is_transmitting	(struct runicast_conn *c, const rimeaddr_t *receiver, uint8_t max_retransmissions) (struct runicast_conn *c)	RUNICAST_PACKET_ID_BITS }, STUNICAST_ATTRIBUTES
11	struct rucb_conn { struct runicast_conn c; const struct rucb_callbacks *u; rimeaddr_t receiver, sender; uint16_t chunk; uint8_t last_seqno; };	rucb_open rucb_close rucb_send	(struct rucb_conn *c, uint16_t channel, const struct rucb_callbacks *u) (struct rucb_conn *c) (struct rucb_conn *c, const rimeaddr_t *receiver)	
12	rudolph0-2?????23 Collect?			
13	struct multihop_conn { struct unicast_conn c; const struct multihop_callbacks *cb; };	multihop_open multihop_close multihop_send	(struct multihop_conn *c, uint16_t channel, const struct multihop_callbacks *u) (struct multihop_conn *c) (struct multihop_conn *c, const rimeaddr_t *to)	MULTIHOP_ATTRIBUTES { PACKETBUF_ADDR_ESENDER, PACKETBUF_ADDRSIZE }, { PACKETBUF_ADDR_ERECEIVER, PACKETBUF_ADDRSIZE }, { PACKETBUF_ATTR_HOPS, PACKETBUF_ATTR_BIT * 5 }, UNICAST_ATTRIBUTES
14	struct rmh_conn { struct runicast_conn c; const struct rmh_callbacks	rmh_open	(struct rmh_conn *c, uint16_t channel, const struct rmh_callbacks	RMH_ATTRIBUTES { PACKET_ADDR_ESENDER, PACKET_ADDRSIZE },

²³ Singlehop reliable bulk data transfer

	*cb; uint8_t num_rexmit; };	rmh_close	*u) (struct rmh_conn *c)	{ PACKET_ADDR_ERECEIVER, PACKET_ADDRSIZE }, { PACKET_ATTR_TTL, PACKET_ATTR_BIT * 5 }, { PACKET_ATTR_MAX_REXMIT, PACKET_ATTR_BIT * 5 }, RUC_ATTRIBUTES
		rmh_send	(struct rmh_conn *c, rimeaddr_t *to, uint8_t num_rexmit, uint8_t max_hops)	
15	struct mesh_conn { struct multihop_conn multihop; struct route_discovery_conn route_discovery_conn; struct queuebuf *queued_data; rimeaddr_t queued_data_dest; const struct mesh_callbacks *cb; };	mesh_open	(struct mesh_conn *c, uint16_t channels, const struct mesh_callbacks *callbacks)	
		mesh_close	(struct mesh_conn *c)	
mesh		mesh_send	(struct mesh_conn *c, const rimeaddr_t *dest)	
16	struct route_discovery_conn { struct netflood_conn rreqconn; struct unicast_conn rrepconn; struct ctimer t; rimeaddr_t last_rreq_originator; uint16_t last_rreq_id; uint16_t rreq_id;	route_discovery_open	(struct route_discovery_conn *c, clock_time_t time, uint16_t channels, const struct route_discovery_callbacks *callbacks);	
		route_discovery_discover	(struct route_discovery_conn *c, const rimeaddr_t *dest, clock_time_t timeout);	
route-discovery				

	const struct route_discovery_callbacks *cb; };	route_discovery_close	(struct route_discovery_conn *c)
17			
	struct neighbor_discovery_conn { struct broadcast_conn c; const struct neighbor_discovery_callbacks *u; struct ctimer send_timer, interval_timer; clock_time_t initial_interval, min_interval, max_interval; clock_time_t current_interval; uint16_t val; };	neighbor_discovery_open	(struct neighbor_discovery_conn *c, uint16_t channel, clock_time_t initial, clock_time_t min, clock_time_t max, const struct neighbor_discovery_callbacks *u);
		neighbor_discovery_close	(struct neighbor_discovery_conn *c)
		neighbor_discovery_set_val	(struct neighbor_discovery_conn *c, uint16_t val)
		neighbor_discovery_start	(struct neighbor_discovery_conn *c, uint16_t val)

Appendix B: Code corresponding to the Rime <unicast, broadcast, abc> stack

Algorithm 5: The Rime code corresponding to composing the unicast stack.

```

struct unicast_callbacks {
    void (* recv)(struct unicast_conn *c, const rimeaddr_t *from);
    void (* sent)(struct unicast_conn *ptr, int status, int num_tx);
};

struct unicast_conn {
    struct broadcast_conn c;
    const struct unicast_callbacks *u;
};

/*-----*/
static void
recv_from_broadcast(struct broadcast_conn *broadcast, const rimeaddr_t
*from)
{
    struct unicast_conn *c = (struct unicast_conn *)broadcast;

    PRINTF("%d.%d: uc: recv_from_broadcast, receiver %d.%d\n",
           rimeaddr_node_addr.u8[0], rimeaddr_node_addr.u8[1],
           packetbuf_addr(PACKETBUF_ADDR_RECEIVER)->u8[0],
           packetbuf_addr(PACKETBUF_ADDR_RECEIVER)->u8[1]);
    if(rimeaddr_cmp(packetbuf_addr(PACKETBUF_ADDR_RECEIVER),
&rimeaddr_node_addr)) {
        c->u->recv(c, from);
    }
}
/*-----*/
static void
sent_by_broadcast(struct broadcast_conn *broadcast, int status, int
num_tx)
{
    struct unicast_conn *c = (struct unicast_conn *)broadcast;

    PRINTF("%d.%d: uc: recv_from_broadcast, receiver %d.%d\n",
           rimeaddr_node_addr.u8[0], rimeaddr_node_addr.u8[1],
           packetbuf_addr(PACKETBUF_ADDR_RECEIVER)->u8[0],
           packetbuf_addr(PACKETBUF_ADDR_RECEIVER)->u8[1]);

    if(c->u->sent) {
        c->u->sent(c, status, num_tx);
    }
}
/*-----*/
static const struct broadcast_callbacks uc = {recv_from_broadcast,
                                           sent_by_broadcast};
/*-----*/
-----*/

```

```

void
unicast_open(struct unicast_conn *c, uint16_t channel,
             const struct unicast_callbacks *u)
{
    broadcast_open(&c->c, channel, &uc);
    c->u = u;
    channel_set_attributes(channel, attributes);
}
/*-----*/
void
unicast_close(struct unicast_conn *c)
{
    broadcast_close(&c->c);
}
/*-----*/
int
unicast_send(struct unicast_conn *c, const rimeaddr_t *receiver)
{
    PRINTF("%d.%d: unicast_send to %d.%d\n",
           rimeaddr_node_addr.u8[0], rimeaddr_node_addr.u8[1],
           receiver->u8[0], receiver->u8[1]);
    packetbuf_set_addr(PACKETBUF_ADDR_RECEIVER, receiver);
    return broadcast_send(&c->c);
}

struct broadcast_callbacks {
    /* Called when a packet has been received by the broadcast module. */
    void (* recv)(struct broadcast_conn *ptr, const rimeaddr_t *sender);
    void (* sent)(struct broadcast_conn *ptr, int status, int num_tx);
};

struct broadcast_conn {
    struct abc_conn c;
    const struct broadcast_callbacks *u;
};

/*-----*/
static void
recv_from_abc(struct abc_conn *bc)
{
    rimeaddr_t sender;
    struct broadcast_conn *c = (struct broadcast_conn *)bc;

    rimeaddr_copy(&sender, packetbuf_addr(PACKETBUF_ADDR_SENDER));

    PRINTF("%d.%d: broadcast: from %d.%d\n",
           rimeaddr_node_addr.u8[0], rimeaddr_node_addr.u8[1],
           sender.u8[0], sender.u8[1]);
    c->u->recv(c, &sender);
}
/*-----*/
static void
sent_by_abc(struct abc_conn *bc, int status, int num_tx)
{
    struct broadcast_conn *c = (struct broadcast_conn *)bc;

    PRINTF("%d.%d: sent to %d.%d status %d num_tx %d\n",
           rimeaddr_node_addr.u8[0], rimeaddr_node_addr.u8[1],
           packetbuf_addr(PACKETBUF_ADDR_SENDER)->u8[0],
           packetbuf_addr(PACKETBUF_ADDR_SENDER)->u8[1],
           status, num_tx);
}

```

```

    if(c->u->sent) {
        c->u->sent(c, status, num_tx);
    }
}
/*-----*/
static const struct abc_callbacks broadcast = {recv_from_abc,
sent_by_abc};
/*-----*/
void
broadcast_open(struct broadcast_conn *c, uint16_t channel,
               const struct broadcast_callbacks *u)
{
    abc_open(&c->c, channel, &broadcast);
    c->u = u;
    channel_set_attributes(channel, attributes);
}
/*-----*/
void
broadcast_close(struct broadcast_conn *c)
{
    abc_close(&c->c);
}
/*-----*/
int
broadcast_send(struct broadcast_conn *c)
{
    PRINTF("%d.%d: broadcast_send\n",
           rimeaddr_node_addr.u8[0], rimeaddr_node_addr.u8[1]);
    packetbuf_set_addr(PACKETBUF_ADDR_SENDER, &rimeaddr_node_addr);
    return abc_send(&c->c);
}

```

Appendix C: Code corresponding to the CRime <c_unicast, c_broadcast, c_abc> stack

Algorithm 6: The CRime code corresponding to composing the c_unicast stack.

```

struct pipe *pi0;
pi0 = (struct pipe*) calloc(1, sizeof(struct pipe));
struct channel *ch0;
ch0 = (struct channel*) calloc(1, sizeof(struct channel));
stack[0].pip = pi0;
stack[0].pip->channel = ch0;
stack[0].modno = 4;
struct stackmodule_i *amodule0;
amodule0 = (struct stackmodule_i*) calloc(
    stack[0].modno, sizeof(struct stackmodule_i));
addr.u8[0] = 0; addr.u8[1] = 0;
set_node_addr(0, OUT, SENDER, &addr);

static struct packetbuf_attrlist c_attributes0[] =
{
    C_UNICAST_ATTRIBUTES PACKETBUF_ATTR_LAST
};

stack[0].pip->channel_no = 0;
stack[0].pip->attrlist = c_attributes0;
stack[0].pip->channel->channelno = stack[0].pip->channel_no;
stack[0].pip->channel->attrlist = stack[0].pip->attrlist;
stack[0].amodule = amodule0;

amodule0[0].stack_id = 0;
amodule0[0].module_id = 0;
stack[0].pip->channel_no = 121;
amodule0[0].c_open = c_channel_open;
amodule0[0].c_close = c_channel_close;
amodule0[0].c_recv = c_abc_input;
amodule0[0].c_send = c_rime_output;

amodule0[1].stack_id = 0;
amodule0[1].module_id = 1;
amodule0[1].c_open = c_abc_open;
amodule0[1].c_close = c_abc_close;
amodule0[1].c_send = c_abc_send;
amodule0[1].c_sent = c_abc_sent;
amodule0[1].c_recv = c_abc_recv;

amodule0[2].stack_id = 0;
amodule0[2].module_id = 2;
addr.u8[0] = 2; addr.u8[1] = 2;
set_node_addr(0, OUT, SENDER, &addr);
amodule0[2].time_trigger_flg = 0;
amodule0[2].trigger_init_flg = 0;
set_amodule_trigger(0, 2);
amodule0[2].c_open = c_broadcast_open;

```

```
amodule0[2].c_close = c_broadcast_close;
amodule0[2].c_send = c_broadcast_send;
amodule0[2].c_sent = c_broadcast_sent;
amodule0[2].c_recv = c_broadcast_recv;

amodule0[3].stack_id = 0;
amodule0[3].module_id = 3;
addr.u8[0] = 1; addr.u8[1] = 1;
set_node_addr(0, OUT, RECEIVER, &addr);
amodule0[3].trigger_init_flg = 0;
set_amodule_trigger(0, 3);
amodule0[3].c_open = c_unicast_open;
amodule0[3].c_close = c_unicast_close;
amodule0[3].c_send = c_unicast_send;
amodule0[3].c_sent = c_unicast_sent;
amodule0[3].c_recv = c_unicast_recv;
```

Appendix D: The CRime ontology

```

<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
  <!ENTITY cpan "http://downlode.org/rdf/cpan/0.1/cpan.rdf#" >
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
  <!ENTITY      Process      "http://www.daml.org/services/owl-
s/1.1B/Process.owl#" >
]>
<rdf:RDF xmlns="http://sensorlab.ijs.si/2012/v0/crime.owl#"
  xml:base="http://sensorlab.ijs.si/2012/v0/crime.owl"
  xmlns:Process="http://www.daml.org/services/owl-
s/1.1B/Process.owl#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:cpan="http://downlode.org/rdf/cpan/0.1/cpan.rdf#"
  xmlns:owl="http://www.w3.org/2002/07/owl#">
  <owl:Ontology
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl"/>
    <!--
    //
    // Annotation properties
    //
    -->
    <!--
    //
    // Datatypes
    //
    -->
    <!--
    //
    // Object Properties
    //
    -->
    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#definedBy -->

    <owl:ObjectProperty
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#definedBy">
      <rdf:type rdf:resource="&owl;FunctionalProperty"/>

```

```

        <rdfs:comment>Functions are defined by Modules.</rdfs:comment>
        <owl:inverseOf
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#defines"/>
        </owl:ObjectProperty>
        <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#defines -->
        <owl:ObjectProperty
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#defines">
        <rdfs:domain>
            <owl:Restriction>
                <owl:onProperty
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#defines"/>
                <owl:someValuesFrom rdf:resource="&#x2013;Module"/>
            </owl:Restriction>
        </rdfs:domain>
        <rdfs:range>
            <owl:Restriction>
                <owl:onProperty
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#defines"/>
                <owl:someValuesFrom
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#Function"/>
            </owl:Restriction>
        </rdfs:range>
        </owl:ObjectProperty>
        <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#hasInterface -->
        <owl:ObjectProperty
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#hasInterface">
        <rdfs:domain>
            <owl:Restriction>
                <owl:onProperty
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#hasInterface"/>
                <owl:someValuesFrom rdf:resource="&#x2013;Module"/>
            </owl:Restriction>
        </rdfs:domain>
        <rdfs:range>
            <owl:Restriction>
                <owl:onProperty
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#hasInterface"/>
                <owl:someValuesFrom
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#Interface"/>
            </owl:Restriction>
        </rdfs:range>
        </owl:ObjectProperty>
        <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#hasParameter -->
        <owl:ObjectProperty
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#hasParameter">
        <rdfs:comment>A            Module            can            have            several
parameters.</rdfs:comment>
        <rdfs:domain>

```

```

        <owl:Restriction>
            <owl:onProperty
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#hasParameter"/>
                <owl:someValuesFrom rdf:resource="&cpan;Module"/>
            </owl:Restriction>
        </rdfs:domain>
    <rdfs:range>
        <owl:Restriction>
            <owl:onProperty
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#hasParameter"/>
                <owl:someValuesFrom rdf:resource="&Process;Parameter"/>
            </owl:Restriction>
        </rdfs:range>
    </owl:ObjectProperty>
    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#hasScope -->
    <owl:ObjectProperty
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#hasScope">
        <rdfs:comment>From the communication point of view, a module can
have several scopes: singlehop, multihop, etc.</rdfs:comment>
        <rdfs:range>
            <owl:Restriction>
                <owl:onProperty
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#hasScope"/>
                    <owl:someValuesFrom
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#Scope"/>
                        </owl:Restriction>
            </rdfs:range>
        <rdfs:domain>
            <owl:Restriction>
                <owl:onProperty
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#hasScope"/>
                    <owl:someValuesFrom rdf:resource="&cpan;Module"/>
                </owl:Restriction>
            </rdfs:domain>
        </owl:ObjectProperty>
    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#implementedBy -->
    <owl:ObjectProperty
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#implementedBy">
        <rdfs:comment>Interfaces          are          implementedBy
functions.</rdfs:comment>
        <owl:inverseOf
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#implements"/>
            </owl:ObjectProperty>
    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#implements -->
    <owl:ObjectProperty
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#implements">
        <rdfs:domain>
            <owl:Restriction>

```

```

        <owl:onProperty
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#implements"/>
        <owl:someValuesFrom
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#Function"/>
        </owl:Restriction>
    </rdfs:domain>
    <rdfs:range>
        <owl:Restriction>
            <owl:onProperty
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#implements"/>
            <owl:someValuesFrom
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#Interface"/>
            </owl:Restriction>
        </rdfs:range>
    </owl:ObjectProperty>
    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#isSetBy -->
    <owl:ObjectProperty
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#isSetBy">
        <rdfs:comment>The value of the parameter is set by a Module and
is either determined automatically or requested from the end user as
initialization parameter.</rdfs:comment>
        <rdfs:subPropertyOf rdf:resource="&owl;topObjectProperty"/>
        <rdfs:domain>
            <owl:Restriction>
                <owl:onProperty
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#isSetBy"/>
                <owl:someValuesFrom rdf:resource="&Process;Parameter"/>
            </owl:Restriction>
        </rdfs:domain>
        <rdfs:range>
            <owl:Restriction>
                <owl:onProperty
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#isSetBy"/>
                <owl:someValuesFrom rdf:resource="&cpan;Module"/>
            </owl:Restriction>
        </rdfs:range>
    </owl:ObjectProperty>
    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#isUserSetBy -->
    <owl:ObjectProperty
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#isUserSetBy">
        <rdfs:comment>The value of the parameter is set by a Module and
is requested from the end user as initialization
parameter.</rdfs:comment>
        <rdfs:subPropertyOf
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#isSetBy"/>
        </owl:ObjectProperty>
    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#isUserSetByOptional -
->

```

```

    <owl:ObjectProperty
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#isUserSetByOptional
">
    <rdfs:subPropertyOf
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#isSetBy"/>
    </owl:ObjectProperty>
    <!--
    //
    // Classes
    //
    -->
    <!-- http://downlode.org/rdf/cpan/0.1/cpan.rdf#Module -->
    <owl:Class rdf:about="&cpan;Module"/>

    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#Function -->
    <owl:Class
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#Function"/>
    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#Interface -->
    <owl:Class
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#Interface"/>
    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#Scope -->
    <owl:Class
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#Scope"/>
    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#c_close -->
    <owl:Class
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#c_close">
        <rdfs:subClassOf
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#Function"/>
        </owl:Class>
    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#c_discover -->
    <owl:Class
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#c_discover">
        <rdfs:subClassOf
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#Function"/>
        </owl:Class>
    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#c_dropped -->
    <owl:Class
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#c_dropped">
        <rdfs:subClassOf
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#Function"/>
        </owl:Class>
    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#c_forward -->
    <owl:Class
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#c_forward">
        <rdfs:subClassOf
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#Function"/>
        </owl:Class>
    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#c_open -->

```

```

    <owl:Class
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#c_open">
    <rdfs:subClassOf
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#Function"/>
    </owl:Class>
    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#c_recv -->
    <owl:Class
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#c_recv">
    <rdfs:subClassOf
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#Function"/>
    </owl:Class>
    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#c_send -->
    <owl:Class
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#c_send">
    <rdfs:subClassOf
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#Function"/>
    </owl:Class>
    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#c_sent -->
    <owl:Class
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#c_sent">
    <rdfs:subClassOf
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#Function"/>
    </owl:Class>
    <!-- http://sensorlab.ijs.si/2012/v0/crime.owl#c_timed_out -->
    <owl:Class
rdf:about="http://sensorlab.ijs.si/2012/v0/crime.owl#c_timed_out">
    <rdfs:subClassOf
rdf:resource="http://sensorlab.ijs.si/2012/v0/crime.owl#Function"/>
    </owl:Class>
    <!-- http://www.daml.org/services/owl-s/1.1B/Process.owl#Parameter -
->
    <owl:Class rdf:about="&Process;Parameter"/>
</rdf:RDF>
<!-- Generated by the OWL API (version 3.2.3.1824)
http://owlapi.sourceforge.net -->

```

Appendix E: Turtle statements describing the c_abc module

```
//turtle crime:c_abc rdf:type cpan:Module .
//turtle crime:c_abc rdfs:comment The abc module is designed to send
anonymous broadcast messages on the selected channel. It uses three
parameters chammelno, attrlist and num_tx. Channel_no configures the
channel, attrlist determines the header of the packet while num_tx
stores the number of attempted transmissions. All parameters are set in
other modules. .

//turtle crime:c_abc_open rdf:type crime:c_open .
//turtle crime:c_abc_close rdf:type crime:c_close .
//turtle crime:c_abc_send rdf:type crime:c_send .
//turtle crime:c_abc_sent rdf:type crime:c_sent .
//turtle crime:c_abc_recv rdf:type crime:c_recv .

//turtle crime:c_abc_open crime:implements crime:top_interface .
//turtle crime:c_abc_close crime:implements crime:top_interface .
//turtle crime:c_abc_send crime:implements crime:top_interface .
//turtle crime:c_abc_sent crime:implements crime:top_interface .
//turtle crime:c_abc_recv crime:implements crime:top_interface .

//turtle crime:c_abc_open crime:implements crime:bottom_interface .
//turtle crime:c_abc_close crime:implements crime:bottom_interface .
//turtle crime:c_abc_send crime:implements crime:bottom_interface .
//turtle crime:c_abc_recv crime:implements crime:bottom_interface .

//turtle crime:c_abc crime:hasScope crime:singlehop .

//turtle crime:c_abc crime:defines crime:c_abc_open .
//turtle crime:c_abc crime:defines crime:c_abc_close .
//turtle crime:c_abc crime:defines crime:c_abc_send .
//turtle crime:c_abc crime:defines crime:c_abc_sent .
//turtle crime:c_abc crime:defines crime:c_abc_recv .

//turtle crime:channel_no rdf:type owls:Parameter .
//turtle crime:c_abc crime:hasParameter crime:channel_no .
//turtle crime:attrlist rdf:type owls:Parameter .
//turtle crime:c_abc crime:hasParameter crime:attrlist .
//turtle crime:num_tx rdf:type owls:Parameter .
//turtle crime:c_abc crime:hasParameter crime:num_tx .

//turtle crime:time_trigger_flg rdf:type owls:Parameter .
```

```
//turtle crime:c_abc crime:hasParameter crime:time_trigger_flg .
//turtle crime:time_trigger_flg crime:isUserSetByOptional crime:c_abc .
//turtle crime:trigger_interval rdf:type owls:Parameter .
//turtle crime:c_abc crime:hasParameter crime:trigger_interval .
//turtle crime:trigger_interval crime:isUserSetByOptional crime:c_abc .
//turtle crime:trigger_no rdf:type owls:Parameter .
//turtle crime:c_abc crime:hasParameter crime:trigger_no .
//turtle crime:trigger_no crime:isUserSetByOptional crime:c_abc .
//turtle crime:trigger_th rdf:type owls:Parameter .
//turtle crime:c_abc crime:hasParameter crime:trigger_th .
//turtle crime:trigger_th crime:isUserSetByOptional crime:c_abc .
```

Appendix F: Feedback collection: questionnaire and results

User1:

User profile

1. Age 25
2. Education: MSc. Computer Science
3. Research interest: semantic web, semantic sensor web
4. Previous experience with Contiki (yes/no)? No
5. Previous experience with Rime (yes/no)? No
6. Are you familiar with TCP/IP or other protocol stack? Yes
7. Are you familiar with the concept of “modular stack”? No
8. Are you familiar with the concept of “composition of services”? No
9. Are you familiar with broadcast communication? Yes
10. Are you familiar with unicast communication? Yes
11. Are you familiar with multihop communication? Yes
12. Are you familiar with mesh communication? Maybe

Introduction

You are presented with the prototype of a tool designed to support experimentation with dynamic composition of communication services (or, in other words modular protocol stacks). The idea is to use the workbench and the components available within to construct a protocol stack with a given functionality. To construct a stack you need to drag and drop modules (boxes) from the left hand side of the UI to the working area. Then you have to wire these components in a meaningful way. Finally, you have to provide some configuration parameters as required by the module. In the end you need to save the stack.

!!! When building a stack you need to add an application module (c_app) and at least one channel module (c_channel).

Training

Netflood stack (channel, broadcast, polite, netflood).

Task description

Task 1: Construct a simple stack which allows VESNA to send messages to all neighbors that can hear it.

Task 2: Construct a simple stack which allows VESNA to send messages to a particular neighbor.

User feedback

13. Is the GUI intuitive (rate from 1 to 5)? 4
14. How to improve the GUI? -

15. Is the toolbar intuitive (rate from 1 to 5)? 5
16. How to improve the toolbar?
 Right now, when you hover over a module, a short description appears. Make it in such way that when selected, more details about the module are available on the right hand side of the workbench where space is not currently used.
17. Are the modules sufficiently described (rate from 1 to 5)? 5
18. Can this system save you time (yes/no)? Probably
19. How much time do you estimate it can save? -

20. Does it give you more flexibility vs Rime(yes/no)? -
21. What would you like to do with ProtoStack? Play with sensors.

22. Please give comments to help improve the tool:
 Put a dropbox for selecting the name of the stack. Prompt with a message when the stack is successfully saved.

Observer feedback

Total time spent on Task 1, 2.

2 min, 5 min (it is no clear why unicast module needs the broadcast module)

Trial-error number for Task 1, 2 (how many save failed messages).

1, 2

Questions asked by users.

User2:

User profile

1. Age 24
2. Education: BSc in Telecommunications
3. Research interest: Cognitive radio, spectrum sensing
4. Previous experience with Contiki (yes/no)? yes
5. Previous experience with Rime (yes/no)? yes
6. Are you familiar with TCP/IP or other protocol stack? yes
7. Are you familiar with the concept of “modular stack”? no

8. Are you familiar with the concept of “composition of services”? yes
9. Are you familiar with broadcast communication? yes
10. Are you familiar with unicast communication? yes
11. Are you familiar with multihop communication? yes
12. Are you familiar with mesh communication? yes

Introduction

You are presented with the prototype of a tool designed to support experimentation with dynamic composition of communication services (or, in other words modular protocol stacks). The idea is to use the workbench and the components available within to construct a protocol stack with a given functionality. To construct a stack you need to drag and drop modules (boxes) from the left hand side of the UI to the working area. Then you have to wire these components in a meaningful way. Finally, you have to provide some configuration parameters as required by the module. In the end you need to save the stack.

!!! When building a stack you need to add an application module (c_app) and at least one channel module (c_channel).

Training

Netflood stack (channel, broadcast, polite, netflood).

Task description

Task 1: Construct a simple stack which allows VESNA to send messages to all neighbors that can hear it.

Task 2: Construct a simple stack which allows VESNA to send messages to a particular neighbor.

User feedback

13. Is the GUI intuitive (rate from 1 to 5)? 4
14. How to improve the GUI?
Disable controls that set parameters not relevant to the current settings (disable/hide trigger_no is trigger_flg is not selected). Add explanations to parameters (sort of a help).
15. Is the toolbar intuitive (rate from 1 to 5)? 5
16. How to improve the toolbar? -
17. Are the modules sufficiently described (rate from 1 to 5)? 3
18. Can this system save you time (yes/no)? CRime yes, GUI not
19. How much time do you estimate it can save? The composition of services can save a lot of time (50%-70%).
20. Does it give you more flexibility vs Rime(yes/no)? yes
21. What would you like to do with ProtoStack? Quickly configure and perform experiments.
22. Please give comments to help improve the tool:

Modules which can use more than one communication interface could have multiple connection points (mesh should have two points at the bottom interface). Stack name should be automatically generated.

Observer feedback

Total time spent on Task 1, 2.

3, 4

Trial-error number for Task 1, 2 (how many save failed messages).

1, 1

Questions asked by users.

User3:

User profile

1. Age 29
2. Education BSc. Electronics
3. Research interest Cognitive radio
4. Previous experience with Contiki (yes/no)? yes
5. Previous experience with Rime (yes/no)? no
6. Are you familiar with TCP/IP or other protocol stack? yes
7. Are you familiar with the concept of “modular stack”? no
8. Are you familiar with the concept of “composition of services”? no
9. Are you familiar with broadcast communication? yes
10. Are you familiar with unicast communication? yes
11. Are you familiar with multihop communication? yes
12. Are you familiar with mesh communication? yes

Introduction

You are presented with the prototype of a tool designed to support experimentation with dynamic composition of communication services (or, in other words modular protocol stacks). The idea is to use the workbench and the components available within to construct a protocol stack with a given functionality. To construct a stack you need to drag and drop modules (boxes) from the left hand side of the UI to the working area. Then you have to wire these components in a meaningful way. Finally, you have to provide some configuration parameters as required by the module. In the end you need to save the stack.

!!! When building a stack you need to add an application module (c_app) and at least one channel module (c_channel).

Training

Netflood stack (channel, broadcast, polite, netflood).

Task description

Task 1: Construct a simple stack which allows VESNA to send messages to all neighbors that can hear it.

Task 2: Construct a simple stack which allows VESNA to send messages to a particular

neighbor.

User feedback

13. Is the GUI intuitive (rate from 1 to 5)? 4
14. How to improve the GUI? Feedback on save, the “not saved” warning doesn’t disappear when saved.
15. Is the toolbar intuitive (rate from 1 to 5)?
16. How to improve the toolbar? Update descriptions for the modules.
17. Are the modules sufficiently described (rate from 1 to 5)?
18. Can this system save you time (yes/no)?
19. How much time do you estimate it can save?
20. Does it give you more flexibility vs Rime(yes/no)? yes
21. What would you like to do with ProtoStack?
22. Please give comments to help improve the tool:

Observer feedback

Total time spent on Task 1, 2.

2, 1

Trial-error number for Task 1, 2 (how many save failed messages). 0, 1

Questions asked by users. Are we evaluating the UI or the entire system. What is the difference between CRime and Rime? Does the stack <c_channel, c_unicast, c_broadcast> work?

User4:

User profile

1. Age 28
2. Education BSc Electrical Engineering
3. Research interest web technologies
4. Previous experience with Contiki (yes/no)? yes
5. Previous experience with Rime (yes/no)? yes
6. Are you familiar with TCP/IP or other protocol stack? yes
7. Are you familiar with the concept of “modular stack”? yes
8. Are you familiar with the concept of “composition of services”? yes
9. Are you familiar with broadcast communication? yes
10. Are you familiar with unicast communication? yes
11. Are you familiar with multihop communication? yes

12. Are you familiar with mesh communication? yes

Introduction

You are presented with the prototype of a tool designed to support experimentation with dynamic composition of communication services (or, in other words modular protocol stacks). The idea is to use the workbench and the components available within to construct a protocol stack with a given functionality. To construct a stack you need to drag and drop modules (boxes) from the left hand side of the UI to the working area. Then you have to wire these components in a meaningful way. Finally, you have to provide some configuration parameters as required by the module. In the end you need to save the stack.

!!! When building a stack you need to add an application module (c_app) and at least one channel module (c_channel).

Training

Netflood stack (channel, broadcast, polite, netflood).

Task description

Task 1: Construct a simple stack which allows VESNA to send messages to all neighbors that can hear it.

Task 2: Construct a simple stack which allows VESNA to send messages to a particular neighbor.

User feedback

13. Is the GUI intuitive (rate from 1 to 5)? 5

14. How to improve the GUI? More messages to suggest which modules are missing.

15. Is the toolbar intuitive (rate from 1 to 5)? 5

16. How to improve the toolbar? Give a correspondence between modules and layers of ISO/OSI.

17. Are the modules sufficiently described (rate from 1 to 5)? 5

18. Can this system save you time (yes/no)? yes

19. How much time do you estimate it can save? 50%-90%

20. Does it give you more flexibility vs Rime(yes/no)? yes

21. What would you like to do with ProtoStack? Build applications for VESNA

22. Please give comments to help improve the tool:

Observer feedback

Total time spent on Task 1, 2.

3, 1

Trial-error number for Task 1, 2 (how many save failed messages).

2, 1

Questions asked by users.

User5:

User profile

1. Age 23
2. Education BSc IT
3. Research interest -
4. Previous experience with Contiki (yes/no)? yes
5. Previous experience with Rime (yes/no)? yes- basic
6. Are you familiar with TCP/IP or other protocol stack? yes
7. Are you familiar with the concept of “modular stack”? no
8. Are you familiar with the concept of “composition of services”? no
9. Are you familiar with broadcast communication? yes
10. Are you familiar with unicast communication? yes
11. Are you familiar with multihop communication? yes
12. Are you familiar with mesh communication? yes

Introduction

You are presented with the prototype of a tool designed to support experimentation with dynamic composition of communication services (or, in other words modular protocol stacks). The idea is to use the workbench and the components available within to construct a protocol stack with a given functionality. To construct a stack you need to drag and drop modules (boxes) from the left hand side of the UI to the working area. Then you have to wire these components in a meaningful way. Finally, you have to provide some configuration parameters as required by the module. In the end you need to save the stack.

!!! When building a stack you need to add an application module (c_app) and at least one channel module (c_channel).

Training

Netflood stack (channel, broadcast, polite, netflood).

Task description

Task 1: Construct a simple stack which allows VESNA to send messages to all neighbors that can hear it.

Task 2: Construct a simple stack which allows VESNA to send messages to a particular neighbor.

User feedback

13. Is the GUI intuitive (rate from 1 to 5)? 4

14. How to improve the GUI? Add feature to the GUI that would allow the user to download the generated C code used for defining a stack.
15. Is the toolbar intuitive (rate from 1 to 5)? 5
16. How to improve the toolbar? -

17. Are the modules sufficiently described (rate from 1 to 5)? 4
18. Can this system save you time (yes/no)? yes
19. How much time do you estimate it can save? 50%

20. Does it give you more flexibility vs Rime(yes/no)? yes
21. What would you like to do with ProtoStack?

22. Please give comments to help improve the tool:
 Add more info about the modules on the right side of the workbench.
 Insert in the help a quick start guide with few examples.
 Give more explanations in the error messages.

Observer feedback

Total time spent on Task 1, 2.

5, 2

Trial-error number for Task 1, 2 (how many save failed messages).

1, 2

Questions asked by users.

User6:

User profile

1. Age 30
2. Education BSc in Telecommunications
3. Research interest cognitive networks
4. Previous experience with Contiki (yes/no)? yes
5. Previous experience with Rime (yes/no)? yes
6. Are you familiar with TCP/IP or other protocol stack? yes
7. Are you familiar with the concept of “modular stack”? yes
8. Are you familiar with the concept of “composition of services”? yes
9. Are you familiar with broadcast communication? yes
10. Are you familiar with unicast communication? yes
11. Are you familiar with multihop communication? yes
12. Are you familiar with mesh communication? yes

Introduction

You are presented with the prototype of a tool designed to support experimentation with

dynamic composition of communication services (or, in other words modular protocol stacks). The idea is to use the workbench and the components available within to construct a protocol stack with a given functionality. To construct a stack you need to drag and drop modules (boxes) from the left hand side of the UI to the working area. Then you have to wire these components in a meaningful way. Finally, you have to provide some configuration parameters as required by the module. In the end you need to save the stack.

!!! When building a stack you need to add an application module (c_app) and at least one channel module (c_channel).

Training

Netflood stack (channel, broadcast, polite, netflood).

Task description

Task 1: Construct a simple stack which allows VESNA to send messages to all neighbors that can hear it.

Task 2: Construct a simple stack which allows VESNA to send messages to a particular neighbor.

User feedback

13. Is the GUI intuitive (rate from 1 to 5)? 4
14. How to improve the GUI? Add more description for modules on the right side of the workbench.
15. Is the toolbar intuitive (rate from 1 to 5)? 4
16. How to improve the toolbar? -
17. Are the modules sufficiently described (rate from 1 to 5)? 4
18. Can this system save you time (yes/no)? yes
19. How much time do you estimate it can save? 50-70%
20. Does it give you more flexibility vs Rime(yes/no)? yes
21. What would you like to do with ProtoStack? Prototype and validate protocols.
22. Please give comments to help improve the tool:

Observer feedback

Total time spent on Task 1, 2.

1, 1

Trial-error number for Task 1, 2 (how many save failed messages).

0, 0

Questions asked by users.

Appendix G: Publications Related to this Thesis

Original scientific article

Fortuna, C.; Mohorcic, M.; Trends in the development of communication networks: Cognitive networks. *Computer Networks* **53**, 1354–1376 (2009).

Fortuna, C.; Mohorcic, M.; Dynamic composition of services for end-to-end information transport. *IEEE Wireless Communications Magazine* **16**, 56–62 (2009).

Published scientific conference contribution

Fortuna, C.; Oniga, P.; Padrah, Z.; Mohorcic, M.; Moraru, A. Metadata management for the web of things: a practical perspective. In: *Proceedings of the Third International Workshop on the Web of Things*. 1–4 (ACM, Newcastle, UK, 2012).

Fortuna, C.; Vucnik, M.; Fortuna, B.; Kenda, K.; Moraru, A.; Mladenec, D. Towards building a global oracle : a physical mashup using artificial intelligence technology. In: *Proceedings of the Third International Workshop on the Web of Things*. 1–4 (ACM, Newcastle, UK, 2012).

Vucnik, M.; Padrah, Z.; Fortuna, C.; Mohorcic, M. Development of discovery and identification protocol for sensor networks. In: *Proceedings of the 4th Jožef Stefan International Postgraduate School Students Conference*, 188–194 (Ljubljana, Slovenija, 2012).

Moraru, A.; Mladenec, D.; Vucnik, M.; Porcius, M.; Fortuna, C.; Mohorcic, M. Exposing real world information for the web of things. In: *the Proceedings of the Eighth International Workshop on Information Integration on the Web*. 1–6 (ACM, Hyderabad, India, 2011).

Smolnikar, M.; Fortuna, C.; Vučnik, M.; Mihelin, M.; Mohorčič, M. Wireless Sensor Network Testbed on Public Lighting Infrastructure, In: *Proceedings of the Second International Workshop on Sensing Technologies in Agriculture, Forestry and Environment*. (Belgrade, Serbia, 2011).

Fortuna, C.; Grobelnik, M. The web of things: tutorial description. In: *Proceedings of the 20th International World Wide Web Conference*. 1–6 (ACM, Hyderabad, India, 2011).

Vucnik, M.; Fortuna, C.; Porcius, M.; Mohorcic, M. WSN testbed for lighting control and environmental monitoring. In: *Proceedings of the 3th Jožef Stefan International Postgraduate School Students Conference*. 1–6 (Ljubljana, Slovenija, 2012).

Padrah, Z.; Fortuna, C.; Mohorcic, M. Connecting Contiki enabled versatile sensor nodes via CC1101 radio. In: *Proceedings of the 3th Jožef Stefan International Postgraduate School Students Conference*, 110–115 (Ljubljana, Slovenija, 2012).

Fortuna, C.; Mohorcic, M. A local knowledge base for the media independent information system. In: *Proceedings of the second future internet symposium*. 15–24 (Springer-Verlang, Berlin, Germany, 2009).

Porcius, M.; Fortuna, C.; Kandus, G.; Mohorcic, M. Integrating custom hardware into sensor web. In: *Proceedings of the 18th International Conference on Software Telecommunications and Computer Networks*, 1–5 (Split, Croatia, 2012).

Fortuna, C.; Ivan, B.; Padrah, Z.; Bradesko, L.; Fortuna, B.; Mohorcic, M. Demonstration: Wireless access network selection enabled by semantic technologies. *International Semantic Web Conference*, (Chantilly, Va, USA, 2009).

Moraru, A.; Fortuna, C.; Fortuna, B.; Slavescu, R.R. A hybrid approach to QoS-aware web service classification and recommendation. In: *Proceedings of the IEEE 5th International Conference on Intelligent Computer Communications and Processing*. 343–346 (IEEE, Cluj-Napoca, Romania, 2009).

Fortuna, C.; Mohorcic, M. A local knowledge base for service oriented access network selection. In: *Proceedings of the SIGCOMM Conference and co-located Workshops MobiHel*. 65–66 (ACM, Barcelona, Spain, 2009).

Fortuna, C.; Mohorcic, M.; Filipic, B. Multiobjective optimization of service delivery over a heterogeneous wireless access system. In: *Proceedings of the IEEE International Symposium on Wireless Communication Systems*. 133–137 (IEEE, Reykjavik, Iceland, 2008).

Fortuna, C.; Mohorcic, M.; Kandus, G.; Filipic, B. SmartA: developing an architecture for cognitive wireless networks. In: *Proceedings of the 16th IEEE Workshop on Local and Metropolitan Area Networks*. 1–2 (Cluj-Napoca, Transylvania, Romania, 2008).

Fortuna, C.; Mohorcic, M. Advanced access architecture for efficient service delivery in heterogeneous wireless networks. In: *Proceedings of the international Workshop on Cognitive Networks and Communications Communications*. 1173–1177 (IEEE, Hangzhou, China, 2008).

Moraru, A.; Fortuna, B.; Fortuna, C. Semantic modeling, translation and matching of QoS. In: *Proceedings of the 11th International Multiconference of Information Society*. 182–185 (IS, Ljubljana, Slovenia, 2008).

Appendix H: Biography

Carolina Fortuna is a research assistant at the Department of Communication Systems at the Jozef Stefan Institute since 2006. During this time she pursued her doctoral studies, published several journal and conference papers, helped starting the SensorLab group and contributed to several successful project proposals.

Prior to this, Carolina completed her Bachelor of Science degree in telecommunications engineering at the Technical University of Cluj-Napoca, Romania. She also graduated from the “Tiberiu Popoviciu” Informatics College in Cluj-Napoca, Romania.

Carolina was an intern at Siemens PSE, Cluj and Bloomberg Inc, New York, USA. She collaborated on EU funded projects CAPANINA (Communications from Aerial Platform Networks delivering 'Broadband for All'), ACTIVE (ACTIVE – Enabling the Knowledge Powered Enterprise), CREW (Cognitive Radio Experimentation World) and PlanetData.