A Framework for Dynamic Composition of Communication Services

CAROLINA FORTUNA, Jozef Stefan Institute
MIHAEL MOHORCIC, Jozef Stefan Institute

We propose a framework for dynamic composition of communication services which is well suited for facilitating research and prototyping on real experimental infrastructures of remotely configurable embedded devices. By using the concept of composeability, our framework supports modular component development for various networking functions, therefore promoting code re-use. The framework consists of four components: the physical testbed, the module library, the declarative language and the workbench. Its reference implementation, ProtoStack, developed using semantic web technologies, supports remote experimentation on sensor platform based infrastructure, thus being well suited also for experimenters that do not poses their own physical experimentation infrastructure. We illustrate how ProtoStack supports research in service oriented networks and a cognitive networking. 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. Compared to the most related non composable approach, the CRime library used by ProtoStack has 16 to 17% 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 pays it off.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design - Network communication, Distributed networks, Wireless communication

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: framework, composition, modular, infrastructure, experimentation, ontology, reasoning

ACM Reference Format:
Carolina Fortuna and Mihael Mohorcic, 2013. A Framework for Dynamic Composition of Communication Services. *ACM Trans. Sensor Netw.* 0, 0, Article 0 (0), 43 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Sensor networks are playing an important role in the research and development of next generation (wireless) communication technologies. A large body of research on novel algorithms and protocols for wireless sensor networks has been done without being limited by constraints present in other, more mature, computer and communication technologies. More recently, sensor networks are playing a key role in enabling cognitive radio networks being used mostly for radio spectrum sensing purposes. Many of the findings are being evaluated in sensor based testbeds that were deployed in the Future Internet Research and Experimentation [FIR 2013] and Global Environment for Network Innovations [GEN 2013] initiatives.

This work was supported by the Slovenian Research Agency (Grant no J2-4197 and P2-0016) and the European Community under CREW Cognitive Radio Experimentation World (Grant no 258301).
Author's addresses: C. Fortuna, M. Mohorcic Department of Communication Systems, Jozef Stefan Institute, Jamova 39, 1000 Ljubljana, Slovenia.

DOI: http://dx.doi.org/10.1145/0000000.0000000

An increasing research area where sensor networks are used for experimentation is cognitive (radio) networking (the wiLab.t, TWIST and LOG-a-TEC in the CREW project [CRE 2013] are sensor platform based testbeds). The main reason for using sensor platforms is their relatively low cost and wide range of functionality. For instance, a sensor node can easily perform radio spectrum sensing based on energy detection and can be (re)programmed or (re)configured to transmit on a certain channel based on the sensing result. However, cognitive networking as a concept goes beyond spectrum sensing. The idea behind it is to create dynamic and adaptable communication networks by taking advantage of artificial intelligence techniques [Thomas et al. 2006] [Fortuna and Mohorčič 2009b]. In order to be able to support experimentation with such advanced networking paradigms, a framework that supports the experimenter in designing, implementing, configuring and deploying network services in response to the operating environments and selected goals is required.

In this paper, we propose a framework for the dynamic composition of communication services that allows modular algorithm and protocol design, flexible and dynamic composition and configuration of protocol stacks and easy deployment on a testbed. In 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 prevents adaptation of networks to changing conditions.

The main novelty in this paper with respect to the state of the art in the field are:

— 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. The tool is purposely developed using semantic web technologies to support i) fully remote configuration and experimentation and ii) easy integration into federations of testbeds, thus being well suited also for experimenters that do not poses their own physical experimentation infrastructure.
— A composeable stack for sensor systems called Composeable Rime (CRime).
— The declarative language which supports configuration, validity checking, publishing and finding modules and logical reasoning.

Further novelties are: (i) practical component reuse within the CRime module library which makes programming communication functionality for sensor networks a pleasant exercise, (ii) knowledge representation of and reasoning about the CRime world model for cognitive networking and (iii) 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.

This paper is structured as follows. In Section 2 we define the framework for the dynamic composition of communication services with focus on the key functional components and requirements. Section 3 introduces the CRime module library, its abstraction and an architectural comparison with Rime. Section 4 discusses the requirements and design decisions related to the declarative language, the custom CRime ontology and aspects related to supporting experimental infrastructures. Section 5 is concerned with ProtoStack, the reference implementation of the framework and the configuration steps required for experiment configuration and instantiation. Section 6 presents the way ProtoStack can support service oriented networks by means of a use case where service composition is performed using self-descriptive network elements and logic reasoning. Section 7 presents the way ProtoStack can support experimentation with cognitive networking by showing how it automates previous manually executed tasks.

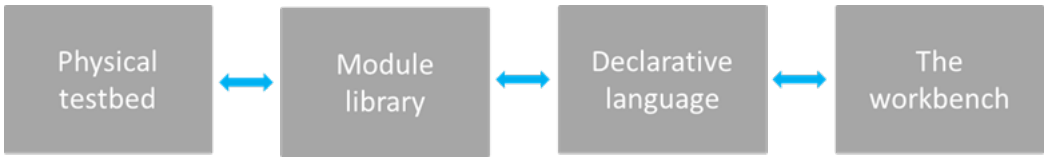


Fig. 1: The four components of the framework for the composition of communication services.

Section 8 quantitatively evaluates the cost of composeability introduced by the CRime architecture with respect to the baseline Rime architecture. This section also includes remarks related to the usability of ProtoStack based on feedback received from first time users. Section 9 summarizes related work with emphasis on the novelty brought by the proposed framework and its reference implementation, and provides evaluation consideration with respect to the related work. Finally, Section 10 concludes the paper.

2. DEFINITION OF THE FRAMEWORK

Compared to existing communication networks with predefined fixed protocol stack, the investigation of dynamically composeable services and protocol stacks that are inherently less structured requires a well defined development environment and basic building blocks to ensure successful outcomes. To this end, we propose a generic framework for the dynamic composition of communication services which can be applied to existing systems as well as future ones.

The overall framework has four functional components: the physical testbed, the module library, the declarative language, and the workbench as depicted in Figure 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 be as diverse as possible (i.e. heterogeneous). This implies that the module library should be portable. 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.

The module library. The module library consists of the source code of the basic 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. A module may correspond to a basic service such as routing (e.g. shortest path routing) or to composite services such as entire protocol (e.g. IP).

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 declarative 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.

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 functionalities that 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). These modules can then be dragged to a workspace, connected to each other and can have 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 within 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 where a summary of the completed experiment can be provided (e.g. total time per operation).

2.1. Requirements

The described framework for the dynamic composition of communication services has to support design and experimentation with new communication services and modular protocol stacks. We identified 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 the web based design of the tool, thus from the beginning considering (semantic) web technologies as most appropriate candidate to fulfill this requirement

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

3. THE COMPOSEABLE RIME MODULE LIBRARY

In this section we introduce the Composable Rime (CRime) module library[1] proposed to support modular protocol design for embedded devices and used for the reference implementation of the proposed framework. CRime is a new architecture but it is inspired by and built upon the existing Rime [Dunkels et al. 2004] architecture.

[1]The module library can be downloaded from https://github.com/sensorlab/CRime

There are two main reasons for which the Rime architecture was chosen as a starting point. First, Rime already provides modularization of communication services from a 'top down' perspective as opposed to a 'bottom up' perspective employed by [Levis et al. 2004] [Tavakoli et al. 2007b] or [Polastre et al. 2005]. By 'top down' perspective we mean that the authors designed and implemented the modules based on the most common communications services used by a network such as unicast, multicast, route discovery, etc. The development was based on the tacit knowledge acquired by the authors from practical experience or from the community and the code was written from scratch. By 'bottom up' we mean that the authors already had a large body of protocol implementations available and they investigated all of them trying to identify the core abstractions that led to the modularization. The 'top down' approach has the advantage that the design and implementation are more reduced in scope, while allowing further extensions. As such, we find this approach more flexible for future extensions while in the existing state sufficient for the proof of concept and allowing to keep focus on the overall framework and tool. The 'bottom up' approach would require more significant effort to be invested in developing and implementing the module library, thus distracting from the overall framework and enabled use cases. The 'bottom up' approach, however, has the advantage that the number of available modules is larger compared to the 'top down' approach. The second reason for choosing the Rime architecture as a foundation was that many of the abstractions already present in Rime such as the functionality of the modules, the tree-like stacks and the separation between packet creation and functionality have been shown [Braden et al. 2003][Levis et al. 2004][De Poorter et al. 2011] to be legitimate and good enablers for flexible and composable stacks.

3.1. CRime abstractions

For the composition of communication services, the CRime architecture introduces three abstractions: the *amodule*, the *pipe* and the *stack*.

CRime abstract module. The CRime amodule (short from abstract module) is a generic building block of the CRime stack. Behind each instance of an amodule there is an implementation of a communication service [Fortuna and Mohorčič 2009a] 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. Conceptually, amodules are similar to protocols in [Hutchinson and Peterson 1991] and [De Poorter et al. 2011], elements in [Kohler et al. 2000], roles in [Braden et al. 2003], components in [Taherkordi et al. 2011] and functional blocks in [Bouabene et al. 2010] in the sense that they contain well defined network functionality or communication service as we refer to it throughout this paper. Their functionality and implementation mimic in a procedural programming language the notion of a class from an object oriented programming languages.

A CRime abstract module defines a generic interface which currently contains 11 generic primitives, based on the functions used by the Rime protocols and may change as the system evolves (i.e. new modules are added, functionality of existing modules is changed). 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 with an interface that implements five primitives: c_open, c_close, c_send, c_recv and c_sent.

In addition to an interface, amodules also define triggers. This means that any amodule can be invoked when a given 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. Triggers can be invoked at any of the following three layers of abstraction: workbench, declarative language, C code. Triggers are exposed for every module available in the workbench and they can be manually configured by clicking a checkbox and providing two parameters specifying trigger time and the number of times they should be executed. Alternatively, this can be specified using the declarative language. Finally, they can be called from the C code for instance by implementing an amodule that realizes functionality such as ”retransmit if the quality of the link is lower than a threshold”.

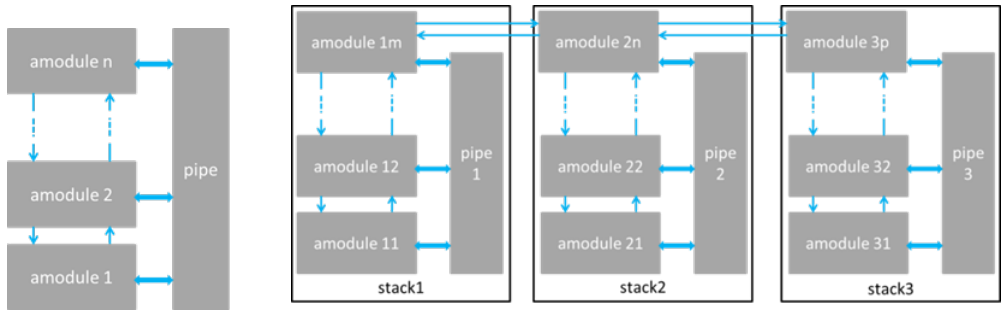
Several instances of the amodule interface are added to a one-dimensional array structure which forms a stack (note that this is a simple stack of the 1 channel —1 stack type). The generic primitives use recursion to walk through the one-dimensional array.

CRime pipe. 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 engineering perspective, nor the most resource efficient in terms of memory requirements, instantiates the concept of vertical layer and can be seen as the first building block in the implementation of the knowledge plane required for experimentation with cognitive networks [Fortuna and Mohorčič 2009b]. The approach is also a compromise between the memory footprint and the complexity required by the autogenerated C code based on user input. Pipes are designed to complement the existing shared memory for packet attributes that already exists in the Rime stack and is used by Chameleon to form protocol headers [Dunkels et al. 2007]. The IDRA system proposed in [De Poorter et al. 2011] actually generalizes the functionality of the shared packet attributes from Rime and the pipe from CRime using the shared queue concept.

CRime stack. The stack is a structure which contains a meaningful sequence of amodules and the corresponding pipe data structure. 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 being uniquely identified by the channel number. These stacks merge at the application layer or below it. For instance, the mesh routing application in Rime uses a set of communication primitives that form a logical tree and run on three different channels [Dunkels et al. 2007]. Data is transmitted on one channel and signalling for route discovery and maintenance on the other two. To enable composeability of such a complex network service as offered by the mesh routing example, CRime uses for implementation three stacks as generically illustrated in Figure 2b, each consisting of an ordered set of amodules and a corresponding pipe.

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. Figure 2 depicts the three abstractions in an example of a 1 channel —1 stack and an example of a 3 channel —3 stack communication system.

The theoretical model behind the CRime communication stack is a tree as depicted in Figure 3. Each node of the tree includes one or more amodules which are connected and



(a) 1 channel—1 stack (b) 3 channel—3 stack

Fig. 2: Example of CRime stacks.

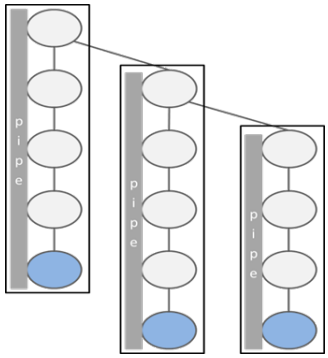


Fig. 3: Tree model with one pipe per stack which is used for the CRime implementation.

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.

In IDRA [De Poorter et al. 2011], the functionality of this abstraction is achieved by the shared queue, however, IDRA assumes a higher level of flexibility than CRime by supporting a classifier that decides what protocol (amodule in our terminology) processes each packet. In CRime this sequence is pre-specified and can change only when a stack reconfiguration action is taken and not automatically for each single incoming packet. Furthermore, through the shared queue, IDRA holds a shared list of neighbors that is accessible to all protocols (amodules). The assumption in CRime is that the set of neighbors for stack A is not the same as the list of neighbors for stack B and thus they should be stored in separate data structures offered by the pipe. However, all modules from that stack can see the list of neighbors. Due to the increased flexibility, IDRA has a larger footprint than the baseline functionality available in TinyOS where it is implemented, similar as CRime has larger footprint with respect to Rime.

3.2. Architectural comparison with Rime

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 (i.e. generates packet headers, adds the data) [Dunkels et al. 2007]. 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 does not include attributes corresponding to non-defined memory locations.

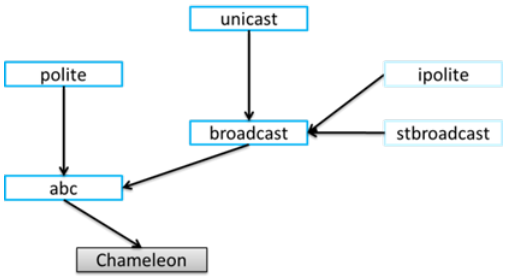
CRime communication primitives. The Rime stack originally defined 10 communication primitives [Dunkels et al. 2007]. The stack has evolved since and currently comprises around 20 primitives, depending on which modules are considered communication primitives. In CRime, the aim was to implement modularity for dynamic reconfiguration while keeping all the functionality from Rime. Since some Rime primitives can be further decomposed as we also demonstrate in this paper, the number of primitives is reduced in the CRime architecture. This means that CRime breaks modules with repeated functionality from Rime to facilitate composeability and component reuse. Furthermore, the dedicated Rime primitives that implement periodic retransmissions, also referred to as stubborn, are replaced in CRime by the amodule triggers. CRime also allows transforming any primitive into a stubborn one by activating the trigger mechanism. Rime primitives which implement very similar functionality such as polite and identified polite are reduced to a single primitive in CRime, thus increasing component reuse. This is enabled by the dynamic composition of amodules which is not supported by Rime.

To exemplify this discussion, let us consider a subset of six Rime primitives (abc, polite, broadcast, stbroadcast, ipolite and unicast) whose functionality is summarized in Appendix A. The selected primitives are by no means exhaustive but they are sufficient to explain why CRime needs fewer modules than Rime to achieve the same functionality. The CRime primitives necessary to implement the same functionality as provided by the 6 Rime primitives enumerated above are c_abc, c_polite, c_broadcast and c_unicast also summarized in Appendix B.

The stubborn unicast module (stunicast, Appendix A) does the same thing for the unicast module as the stubborn broadcast module does for the broadcast primitive. Stunicast enables the repetitive sending of the same packet using unicast while stbroadcast enables the repetitive sending of the same packet using broadcast.

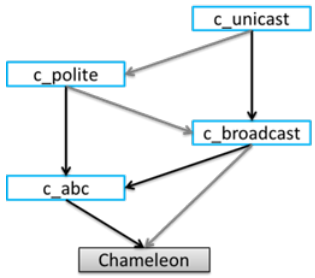
The dependency graph of the modules summarized in six Rime modules is depicted in Figure 4. It can be seen that any Rime protocol based on the 6 primitives enumerated above (and detailed in Appendix A) uses the abc module. Any path through the Rime dependency graph forms one Rime protocol stack. It can 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, CRime's c_polite, similar to Rime's polite, 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.



(a) Rime dependency graph

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



(b) CRime dependency graph

<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>

Fig. 4: Dependency graph in (a) Rime and (b) CRime. It can be seen that with the CRime approach, there are more possible combinations of modules which can form a stack (11 vs 6).

In the Rime dependency graph depicted in Figure 4a, there are a total of 6 possible paths corresponding to 6 possible stacks, while in the CRime dependency graph in Figure 4b, 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.

Support for modular stacks. Reconfiguring a stack in Rime requires good understanding of the Rime design and implementation, 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, but it involves writing only 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, the recursion is explicit and the implementation allows flat initialization using a limited set of variables, which is easy to generate automatically. In Rime, it is infeasible to automatically generate configuration code for a new stack, due to the module specific data structure and parameters and the hard wiring of modules. Additionally, the recursion is implicit through the callback mechanism and it is infeasible to devise a template which would allow automatic generation.

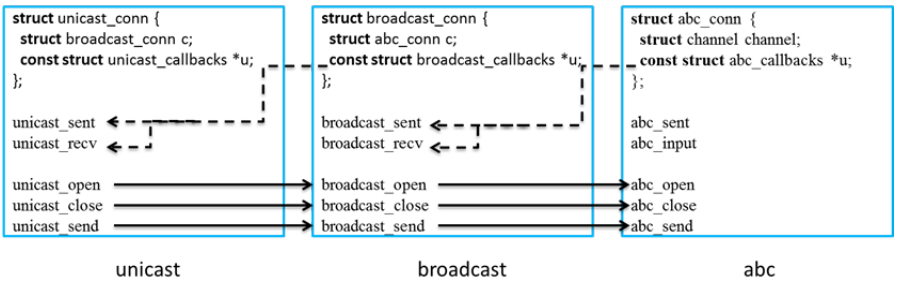


Fig. 5: Illustration of the way the Rime unicast stack is implemented.

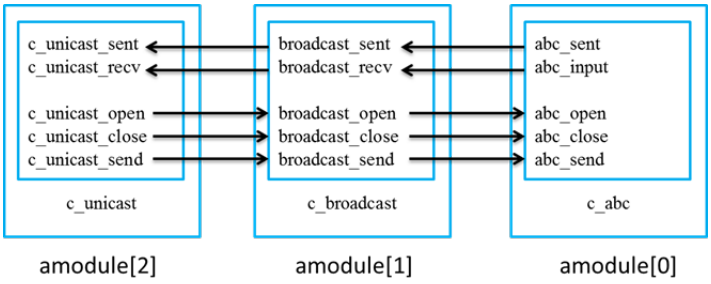


Fig. 6: Illustration of the way a possible CRime unicast stack is implemented.

If we consider the three module Rime stack <unicast, broadcast, abc> defining the unicast communication service, then Figure 5 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 dash line 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.

The equivalent CRime stack is <c_unicast, c_broadcast, c_abc> as depicted in Figure 6. 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 6, 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 as in the Rime example above, we would need to replace the c_polite module between amodule[1] and amodule[2]. The corresponding code consists of correctly initializing a one dimensional array.

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.

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.

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 [Con 2013] 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. THE DECLARATIVE LANGUAGE AND WORKBENCH

The role of the declarative language is to provide a level of abstraction between the workbench and the module library. Besides being used for configuration purposes, such a language can also be used for validity checking (Section 5.2), to federate (sensor based) research infrastructures [Ghijsen et al. 2012] and for knowledge representation and reasoning (Section 6) to support service oriented networks (SONs) [Fortuna and Mohorčič 2009a] and cognitive networks [Fortuna and Mohorčič 2009b].

4.1. Requirements for the declarative language

Upon deciding on a suitable declarative language to be used for the reference implementation of the framework for dynamic composition of communication services, the following requirements were considered:

— *Simplicity:* the proposed framework for dynamic composition of communication protocols and services is targeting researchers 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 languages (Prolog).
— *Machine interpretable:* the declarative language should be machine readable in order to facilitate easy manipulation by machines.
— *Standardized approach:* 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.

Table I: The ProtoStack language model.

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

4.2. Design decisions related to the declarative language

The requirement that 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) [RDF 2013], the Web Ontology Language (OWL) [McGuinness et al. 2004] and CycL [Lenat and Guha 1991]. RDF is the least expressive of the three; OWL is a restricted but more expressive form of RDF, while CycL is the most expressive.

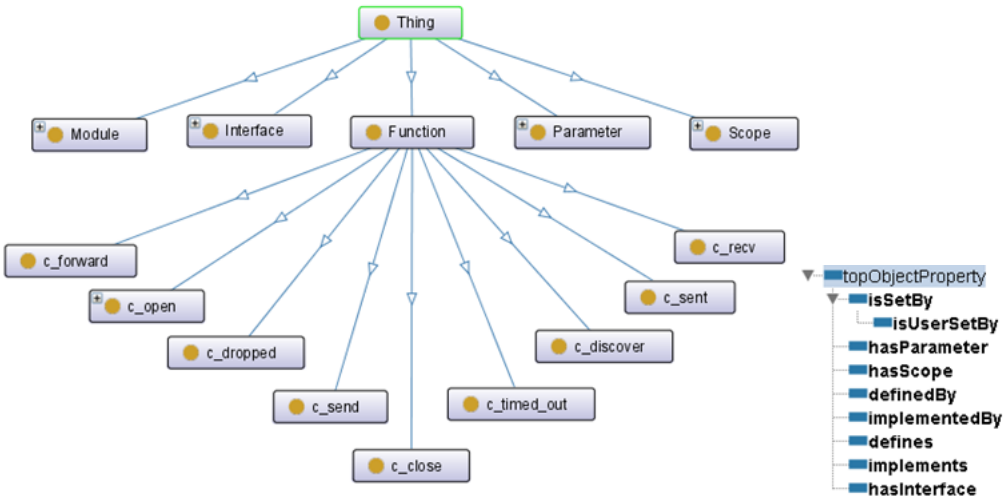
Knowledge represented using these languages can be queried and the complexity of the queries is directly related to the expressiveness of the representation language. Knowledge bases using RDF support relatively simple queries via SPARQL [Prud-Hommeaux et al. 2008]; the ones using OWL support reasoning 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 [Group et al. 2008]. 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. 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 the reference implementation of the framework.

In summary, the 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 I. 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.

4.3. 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 [Shepard et al. 2005]. The need for a generic data model for federated infrastructures has been rec-



(a) CRime classes (b) CRime predicates

Fig. 7: The CRime ontology: (a) classes and (b) predicates.

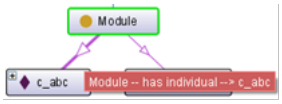
ognized and several initiatives to use ontological models are underway [Ghijsen et al. 2012]. Such data models tend to abstract the real-world network related set-up.

In the case of our work, the world model consists of CRime enabled networks, and therefore the creation of a custom ontology was necessary. In this section we discuss the CRime ontology[2] as needed by the tool while in Section 6 we look at how this ontology can be used and extended for self-descriptive network elements and logic based reasoning.

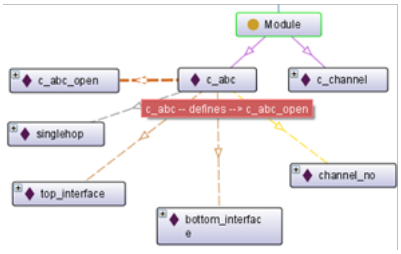
The CRime ontology (Appendix C) was built to help describe the conceptual model behind CRime. It can easily be extended for future use. The ontology consists of two levels of classes as depicted in Figure 7a. The first level contains the generic classes such as Module, Interface, Function, Parameter 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 7b).

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 Figure 8a). 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. Finally, it uses some parameters such as channel_no. All this is stated in the graph depicted in Figure 8b. By further expanding the graph, it can be seen in Figure 8c 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.

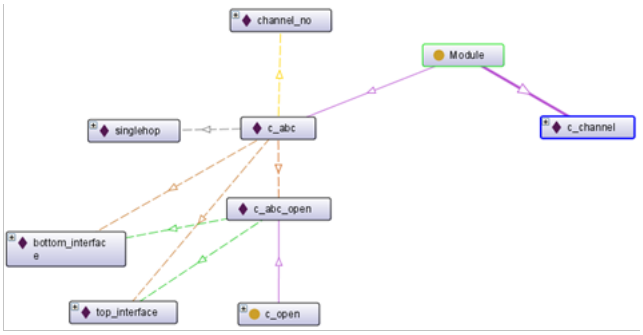
[2]The ontology can be downloaded from https://github.com/sensorlab/ProtoStack/owl



(a) c_abc is an instance of a module.



(b) c_abc defines the c_abc_open primitive.



(c) channel_no is a parameter of c_abc module.

Fig. 8: Example CRime statements describing the c_abc module.

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

4.4. Reusability

The declarative language chosen for the reference implementation of the framework uses a custom made ontology for the CRime set of modules, however. It can be re-used by any other modular stack library and any other framework for stack composition. For this purpose, 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).

4.5. Reproducibility of experiments

Reproducibility of the experiments is a key research requirement; however, it is often neglected in several research areas, including sensor networks and cognitive (radio) networks. Most of the published work include 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 [Gerwen et al. 2011] are addressed by several projects including the Cognitive Radio Experimentation World [CRE 2013].

The declarative language enables easy reproducibility of experiments and can provide support to benchmarking frameworks. Descriptions and configurations of experiments can easily be exchanged between infrastructures and interpreted as long as the data model (the CRime ontology in this case) is also shared. The configuration can then be used as an input to a benchmarking system.

4.6. Support for remote experimentation and federation

The declarative language complies with web standards and it can be used by web portals to define and configure experiments remotely. Because the declarative language uses an ontology as its vocabulary, it is possible to align or merge it with ontologies used by other testbeds [Tosic et al. 2012] to achieve federation. In a federated environment that enables remote experimentation, a user located anywhere in the world and having an internet connection is able to design and configure an experiment and run it on several testbeds with minimal effort.

5. PROTOSTACK IMPLEMENTATION

For the experimental evaluation of the proposed framework for dynamic composition of communication services we first had to develop its reference implementation, the ProtoStack tool[3]. In this section, we describe and discuss its components and the steps for experiment configuration and instantiation.

5.1. Reference implementation: the ProtoStack tool

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 frame of the CREW project [CRE 2013]. 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 an advanced user, such as the component developer, needs to focus on developing the component and make it work with Contiki [31] and a novice user needs only to focus on composing services in a stack using the workbench.

The physical testbed is based on the VESNA sensor network platform with the Contiki OS, which already includes the adaptive Rime architecture [Dunkels et al. 2004]. The constraints of this physical testbed were perfectly matched by CRime as the module library, the declarative language based on the Resource Description Framework (RDF) [RDF 2013] and the Turtle syntax together with existing standardized vocabulary and a custom ontology. The workbench is tightly integrated with the language and is implemented using WireIt, an open source javascript library which enables the creation of full web graph editors.

In Figure 9 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. At the end he/she needs to write few lines of Turtle statements (i.e. triples) which specify basic characteristics of the new module (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 them in the triple store (arrow 1 in Figure 9). 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 9).

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 9). 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 9). 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 9).

[3]The tool can be downloaded from https://github.com/sensorlab/ProtoStack

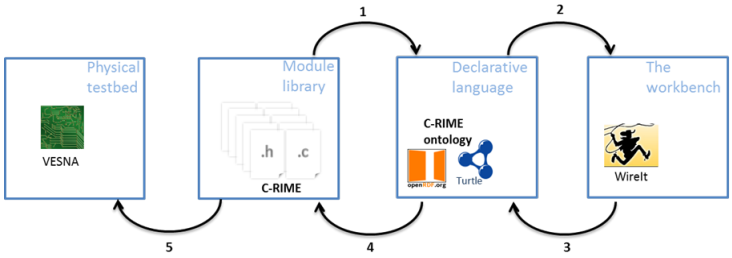


Fig. 9: ProtoStack: an implementation of the framework for composing communication services. The implementation addresses the sensor networks domain.

```
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 .
```

Fig. 10: Example Turtle statements describing the c_abc module.

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. Having the workbench implemented as a web portal and using the over the air programming supported by the VESNA platform, the described use of ProtoStack can be done remotely.

5.2. The ProtoStack configuration steps

The ProtoStack configuration consists of fact specification, translation and storage, workbench rendering, manual stack composition, validity checking and code generation.

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.

Figure 10 lists example statements describing the c_abc module (the full description has 46 statements). 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 10 states that c_abc is a module (i.e. of type module) and the second line states that c_abcs scope is singlehop (i.e. provides singlehop 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.

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

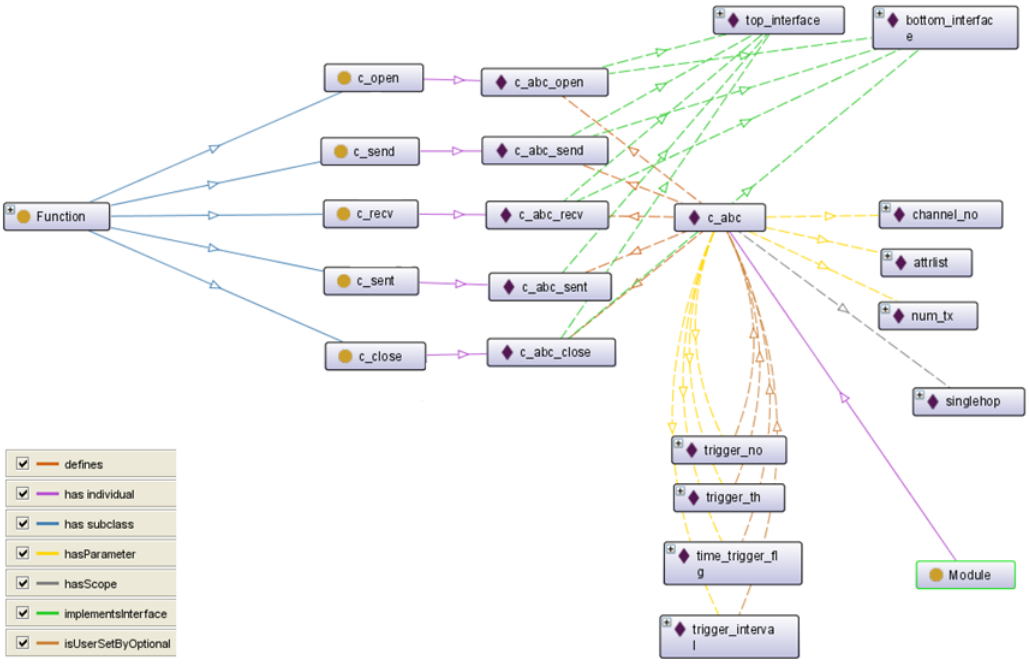


Fig. 11: Snapshot of the RDF graph describing the c_abc module.

in Figure 11 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_recv, c_sent as well as the ones connecting module, scope and interface to thing, represent the subClassOf relationship.

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 to graphical simulation environments. The workbench features a horizontal menu bar and three vertical work areas, as depicted in Figure 12. The menu bar features buttons corresponding to operations that can be performed with the stack. The left most 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 in and dropped, wired and configured in a stack. The right most area consists of the properties bar where additional description of the stack can be provided.

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 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 asks for all the

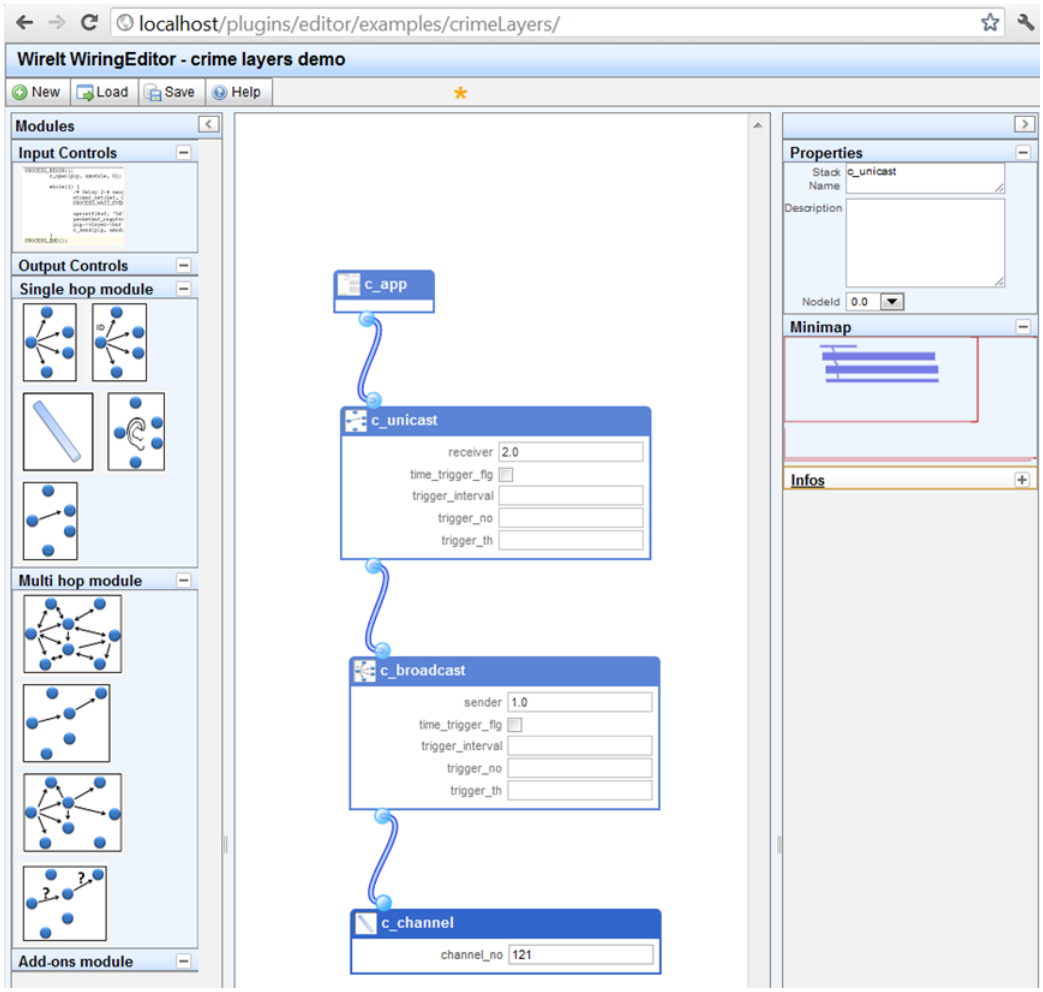


Fig. 12: Snapshot of the ProtoStack workbench with a a 1 channel —1 stack configuration.

available modules and their descriptions (see Figure 13 line 7). Then, it asks for the scope and the parameters corresponding to each module that must be inserted by the user (Figure 13 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 12. Once the stack is built and configured, the corresponding data is sent to the server on user command.

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.

```
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. }
```

Fig. 13: SPARQL query which retrieves the modules and their descriptions for populating the toolbox of the workbench.

When the stack configuration process is completed, the user submits the new stack to the server.

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.

Figure 12 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 12 (time_trigger_flg is not selected), therefore the trigger parameters are not required. The same stack with the triggering mechanism activated would have the time_trigger_flg activated and the trigger_interval parameter value set to the desired triggering frequency (for instance 10 ms) and the trigger_no parameter value set to the number of times the trigger should be activated, thus defining the number of retransmissions and their frequency

Validity checking and code generation. By validity checking we refer to the validity of the composed stack. As already discussed, 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 terminal points. 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. 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 the interface level. For each pair of connected interfaces <N-1 top, N bottom>, <N top, N+1 bottom>, 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.

Experiment deployment. The experiment deployment step depends on the target platform and the management network supported by the testbed. First, ProtoStack using CRime as a module library can be used only on testbeds with sensors nodes that support Contiki OS (over 15 hardware platforms are supported). In order to support other module libraries, the CRime ontology needs to be changed and the corresponding code for automatic code generation has to be added to the server. Second, most sensor network testbeds use the available wireless interfaces for performing the experiment and are controlled through a wired management network such as USB or Ethernet (MoteLab, wiLab.t, TWIST, etc.). In such cases, the most straightforward way to deploy the experiment is to prepare the binary image for the target hardware platform, distribute it over the wired management channel to the nodes, then flash and restart the nodes. Alternatively, the new experiment can be prepared as an ELF file that is dynamically loaded as a new module to the already existing OS running on the node.

Testbeds in a real world operating environment (e.g. LOG-a-TEC, SmartSantander, etc.) however, are managed through a wireless management network running on a different radio interface than the experimental interface. For supporting ProtoStack on such configuration, we have modified the Contiki OS to be able to run 2 network stacks in parallel on two different radio interfaces. Thus in the LOG-a-TEC testbed, it currently runs the 6LoWPAN/CoAP [Kovatsch et al. 2011] stack as a management stack on the Atmel AT86RF212 radio and the CRime experimental stack on the TI CC radio. The reprogramming speed in the case of using wireless management network is significantly lower than in the case with the wired management alternative which can be improved by using dynamic reconfiguration through CoAP service.

6. SUPPORT FOR SERVICE ORIENTED NETWORKS

In [Fortuna and Mohorčič 2009a] we proposed service oriented networks as a network that makes use of principles defined by service oriented architectures (SOA) and performed a comparison of the logical components defined by SOA and the Open Systems Interconnect (OSI). The most important differences consist in the fact that in SOA services have a well described interface using a standard definition language and these interfaces are published, therefore they can be searched for and found. ProtoStack provides an instantiation of this concept for sensor networks: amodules are the communication services that have well described and published interfaces. This section explains how ProtoStack services are described and published, how they can be found

and synchronized across different systems and then illustrates how their composition can be automatised using logic reasoning.

The design and implementation choices employed for ProtoStack allow it being easily co-opted into federations because it uses semantic web technologies and has an associated domain ontology. In recent efforts to federate experimental infrastructures such as the ones in GENI [GEN 2013] and FIRE [FIR 2013], the importance of a common information model to provide unified configuration and control has been recognized [Ren and Jiang 2011][Tosic et al. 2012]. In order to achieve such goal, a common and infrastructure independent way of expressing experimental settings such as channel width, sweeping step, time duration of the experiment should be defined. This is best done using a high level representation language, often by XML syntax that is machine interpretable, standardized and designed to support interoperability. Efforts in creating domain ontologies and then mapping these to upper level, common ontologies are in progress for specific types of experimental infrastructures [Tosic et al. 2012].

The ProtoStack declarative language and the CRime ontology were designed having in mind machine interpretability, standards and interoperability. The language uses the standard RDF data model and the statements describing any module can be readily mapped to XML, JSON and other syntaxes. The CRime ontology, while specific for the CRime modules, can be easily mapped to an upper layer ontology, thus ready to be integrated with other infrastructures. Additionally, the ProtoStack language supports knowledge representation and logic reasoning that can help to: (1) detect inconsistent configurations and (2) increase the level of automation in protocol stack construction to further the state of the art towards logic-based self-configurable networks.

Describing and publishing ProtoStack 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 and published on the web. It can then be further sent to a broker or crawled by semantic web tools [Dodds 2006] which will make it easier to identify and find new modules that are relevant. In other words, ProtoStack services and as a result testbeds using ProtoStack are self-described and discoverable.

Synchronizing distributed ProtoStack systems. The web based implementation of the workbench readily supports distributed use and extension of the ProtoStack tool but also requires an appropriate synchronization mechanism to be put in place. In order to 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 $M_i, i = 1, N$. 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 $M_i \cup M_j$. Furthermore, 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 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

```
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. }
```

Fig. 14: SPARQL formulation of a query to retrieve all instances of type Function.

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

Fig. 15: OWLIM compatible rule that defines transitivity relationship.

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) [Codd 1970]. 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 14.

The query from Figure 14 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 14 can be seen in Figure 15.

Composition of services for information transport using ProtoStack. With respect to the composition of services for information transport using ProtoStack we distinguish among manual, semi-automatic and automatic composition.

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 5.2.

Semi-automatic composition. By choosing a declarative language able to support logic reasoning, ProtoStack can be used for semi-automatic service composition. For instance, assume that the human user selects a module that is suitable for the desired composed service. The tool can then suggest other compatible modules that can

```
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)
```

Fig. 16: SPARQL query that retrieves all multihop modules thus narrowing down the search space for the user.

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

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

be used to build the composed service. 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 ones. 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 Figure 13 is sufficient. In the case the user selected a multihop module, the system should narrow the selection options only to multihop modules. To achieve this, the additional line numbered 11 shown in Figure 16 has to be added to the query in Figure 13.

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 [Matuszek et al. 2006] reasoning engine together with its tightly coupled knowledge base (KB) is used. The CRime ontology was connected to Cyc's knowledge base by mapping the top five concepts to appropriate Cyc knowledge base concepts as can be seen in Table II. 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 shown in Figure 17 were then added to Cyc KB specifying module compatibility based on the scope. Given the current setup consisting of Cyc's KB without application specific extensions, the following query is sufficient to retrieve all combinations of two modules that can be stacked given the constraints we imposed: (#$onAbstract ?X ?Y).

```
(#$implies
     (#$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)
)
```

Fig. 17: Cyc KB rules for scope based stack composition.

Automatic composition. ProtoStack can be further extended for automatic communication service composition 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 the required quality of service (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 [Fortuna et al. 2008].

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 as discussed in [Fortuna and Mohorcic 2010][Fortuna and Mohorcic 2008]. 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 establish a communication because each uses a different protocol stack. Both nodes can communicate with a coordinator (gateway) or an access point and can 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 the protocol stack reconfiguration, the nodes will

be able to communicate directly thus making use of radio resources and reduce overall energy consumption in the network.

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 communication 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).

7. SUPPORT FOR COGNITIVE NETWORKING

In this section, we refer to cognitive networks as sensor networks that implement the cognitive cycle consisting of the *sense*, *plan*, *learn*, *decide* and *act* states described for instance in [Fortuna and Mohorčič 2009b]. We define the use case as follows. The sensors forming the network are *sensing* the link by observing the instant LQI (link quality indicator) and RSSI (received signal strength indication) values provided by the radio transceivers and/or snooping into all the messages received from the neighbors and monitoring sequence numbers (see Figure 18). In a more advanced scenario where a cognitive radio performing more advanced energy detection or channel occupancy detection can also be used to sense. The *planing* and the *learning* states can materialize in models, estimators or predictors of the link quality. For instance, a moving average of the sensed properties can be maintained to be able to tell more about the link than provided by an instant measurement. Additionally, simple prediction models such as linear regression can be learnt over time from the sensed parameters. The *decide* state can materialize in the routing algorithm that selects the best route. With traditional wired networks, a decision can be made based on a shortest path (SP) policy which is performed by the SP routing algorithm. However, in wireless networks, the decision can be made by the SP routing algorithm taking into account sensed (i.e. instant RSSI) or learned (i.e. prediction provided by the linear regression) information or by other algorithms such as minimum transmission (MT) that may give less weight to the hop count and more to the scores provided by the link quality estimators. The *act* state consists of sending the packet on the route selected by the decider.

The cognitive networking use case, on which we show the advantage of using tools such as ProtoStack, considers running extensive experiments in which the options for the *plan*, *learn* and *decide* phases of the cognitive cycle are extensively explored to learn and understand more about the multihop performance of changing network topologies determined by varying link quality. As shown in the literature [Woo et al. 2003], [Tavakoli and Culler 2009], [Kim et al. 2011], [Srinivasan et al. 2010], [Carles et al. 2010] and [Baccour et al. 2012], these kind of explorations need to be performed frequently as new transceivers and more powerful microprocessors supporting more advanced features are used by sensor nodes. Often, the resulting implementations reflect in standardization efforts such as IETF drafts [Tavakoli and Culler 2009] [Kim et al. 2011].

The design space for understanding and optimizing advanced routing in wireless networks along the stages of the cognitive cycle is large and labor intensive. Figure 18 illustrates this space and possible combinations for a selected set of instantiations of the cognitive cycle. Sensing of a choice of link properties such as RSSI, LQI or received frames can be coupled with any planning and learning techniques that estimate the link quality and with any of the decision techniques. More choices that further increase the design space can be found in [Baccour et al. 2012], [Woo et al. 2003], etc. ProtoStack decreases the overhead and required effort for exploring this space as fol-

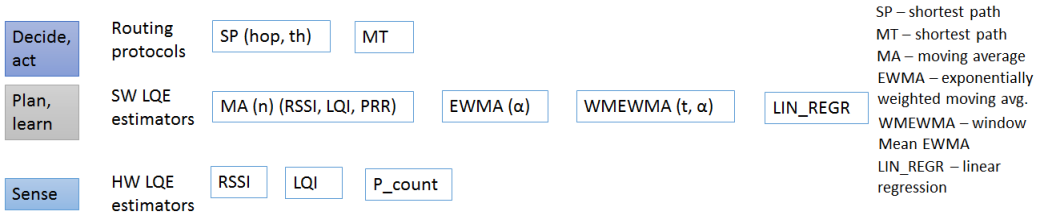
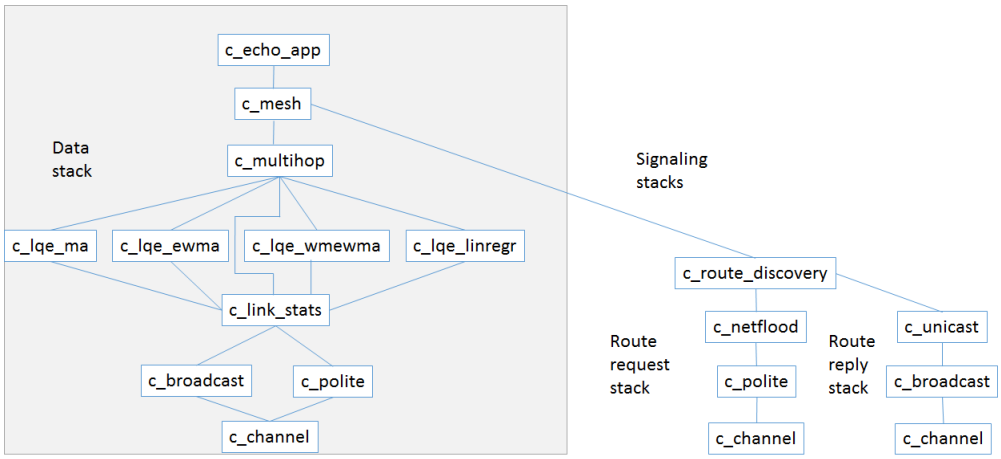


Fig. 18: Example LQE and routing protocols relevant for a cognitive networking scenario.

lows. First, it implements 5 link quality estimation (LQE) modules (see Figure 19a): c_link_stats, c_lqe_ma, c_lqe_ewma, c_lqe_wmewma and c_lqe_linregr which correspond to link statistics of average and standard deviation of RSSI and LQI, link quality estimation using moving average (MA), estimation using exponentially weighted moving average (EWMA), link quality estimation using window mean with EWMA and link estimation based on linear regression, respectively. These modules include adjustable parameters such as time window t size for the moving average estimator and α for the EWMA estimator. Each of these modules fills in a data structure named *neighbor_list* located in the common pipe structure described in Section 3.1. The neighbor list contains a list of all neighbors from which the current node N heard messages. Besides the neighbor's Rime address, the data structure holds time averaged RSSI and LQI and a cost. This cost is computed differently, depending on which software LQE is being used. CRime's c_multihop service extends the routing table with a cost additionally to the already existing hop count. This routing table is also contained in the pipe structure. Besides supporting the shortest path routing, CRime's c_multihop can thus be configured to use shortest path based on threshholding where instant or time averaged LQI and RSSI values are used from the pipe structure's neighbor_list that is filled in by the LQE modules and shared with all the other modules. Additionally, the cost from the route_table can be set based on values provided by the LQE estimators according to the strategy of the decider. Figure 19b lists all the possible combinations of a data stack using the elements that instantiate the cognitive cycle as discussed so far in this section.

As a reference scenario to explain the benefits of using the ProtoStack tool, we consider [Woo et al. 2003] in which the authors were looking for a simple and reliable link estimator (or predictor in machine learning terminology) that would be used by the routing protocol in deciding which route to select. Since the search space for finding and evaluating the estimator and the performance of the routing algorithm is very large, the authors take a four stage approach: 1) determine empirical link characteristics, 2) investigate link estimation techniques, 3) design a neighbor management policy and 4) design and implement the routing framework. This approach can be carried out using the ProtoStack tool as follows.

Empirical link characteristics determination. Empirical link characterization stands at the basis of any work related to multihop wireless network design and is performed in several works such as [Woo et al. 2003], [Srinivasan et al. 2010], [Carles et al. 2010] to name a few. This stage is carried out using real hardware and the results vary across transceivers and deployment environments, so it has to be redone frequently. The result of this step is a statistical model of the channel on which the design and evaluation of LQE estimators is based on. ProtoStack can be used in this stage of work in two ways. First, it can be used to ease the configuration and deployment of



(a) Dependency graph of a stack achieving multihop communication using ProtoStack with focus on the cognitive networking options for the data transmission functionality represented by the Data stack.

<c_channel, c_broadcast, c_link_stats, c_multihop, c_mesh, c_echo_app >
<c_channel, c_broadcast, c_link_stats, c_lqe_ma, c_multihop, c_mesh, c_echo_app >
<c_channel, c_broadcast, c_link_stats, c_lqe_ewma, c_multihop, c_mesh, c_echo_app>
<c_channel, c_broadcast, c_link_stats, c_lqe_wmewma, c_multihop, c_mesh, c_echo_app>
<c_channel, c_broadcast, c_link_stats, c_lqe_linregr, c_multihop, c_mesh, c_echo_app>
<c_channel, c_polite, c_link_stats, c_multihop, c_mesh, c_echo_app>
<c_channel, c_polite, c_link_stats, c_lqe_ma, c_multihop, c_mesh, c_echo_app>
<c_channel, c_polite, c_link_stats, c_lqe_ewma, c_multihop, c_mesh, c_echo_app>
<c_channel, c_politet, c_link_stats, c_lqe_wmewma, c_multihop, c_mesh, c_echo_app>
<c_channel, c_polite, c_link_stats, c_lqe_linregr, c_multihop, c_mesh, c_echo_app>

(b) List of possible composition options for the data stack using selected services offered by the ProtoStack tool.

Fig. 19: Dependency graph of a stack achieving multihop communication using ProtoStack with focus on the cognitive networking options for the data transmission functionality represented by the Data stack.

the experiment where a <c_channel, c_broadcast, c_link_stats, c_echo_app>stack would achieve the required function. Second, and most relevant for research with cognitive networks, the same stack can be used to enable each node to build automatically a statistical model for the link quality. Rather than manually computing the link model and subsequently use the same model on all nodes, each node can build its own model.

Link estimation techniques. Link estimation techniques are typically designed using a statistical model of the link that is often based on purely theoretical model and sometimes in empirical observation such as in the works considered in this paper. Since the number of link estimation techniques tends to be large and each has at least one tuning parameter, their evaluation is performed in a simulator using a statistical model. With ProtoStack, the cost for empirically evaluating these techniques is lowered because 1) a new link estimator is not significantly harder to implement then it would be in a simulator and 2) once implemented, this estimator can be even remotely added to a stack and configured via a graphical user interface. With the increasing number of available sensor testbeds and their increasing size [Sanchez et al. 2013], and with a tool such as ProtoStack that adds a simulation-like interface to them, empirical eval-

uations should become more prevalent also yielding more realistic results compared to those obtained by simulations. Since link estimation techniques embody stages of the cognitive cycle, and ProtoStack already includes advanced link quality estimators such a the one based on linear regression (c_lqe_linregr), also empirical experimentation with self-adaptive and self-learning strategies should become easier.

Neighbor management policy. Neighbor management policies aim at finding the best way for a node to determine, over time, how large a neighbor table should be and which nodes should be added to this table. The size of the table as well as the periodic refresh time are parameters in ProtoStack and can be varied across experiments to have an empirical validation of its effect on routing. The currently available LQE estimation modules are in charge of filling the neighbor list of the pipe structure, each of them using a certain strategy. These modules would need to be extended or new ones added with various approaches to neighbor addition and eviction to support experiments concerning neighbor management policies.

Design and implement the routing framework. The design and implementation of the routing framework has to take into account the previous three stages and also to deal with routing specific aspects such as routing strategy and algorithm, table management, cycle detection, etc. A small set of routing protocols using the most promising link estimation techniques is typically evaluated empirically. Using any of the stacks listed in Figure 19, ProtoStack can support easy experiment configuration, deployment, execution and monitoring of shortest path, shortest path considering link quality and minimum transmission protocols.

8. EXPERIMENTAL EVALUATION WITH RESPECT TO THE BASELINE

In this section, we quantitatively evaluate the cost of composeability introduced by the CRime architecture to the baseline which is Rime. To complement the quantitative evaluation of the ProtoStack tool we conclude with remarks related to the usability of ProtoStack based on feedback received from first time users. Additional evaluation, comparison and discussion to the related work that puts our work into context is provided in Section 9.

8.1. Quantitative assessment of the cost of composeability

A fair quantitative assessment of the cost of composeability brought in ProtoStack by the CRime module library can only be based on a direct comparison between the CRime and Rime architectures because Rime is the closest non-composeable reference implementation. Additional comparison to modular and flexible architectures and the reference implementations of the provided abstractions is provided in Section 9. 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 [Dunkels et al. 2007]. 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 the following, we quantify this overhead focusing 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.

Table III: 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				
				c_link_stats	1117	0	0
				c_lqe_ma	1717	0	4
				c_lqe_ewma	1689	0	4
				c_lqe_wmewma	1641	0	4
				c_lqe_linregr	1233	0	0
				c_echo_app	173	0	0
				amodule	2572	0	4
				stack	3760	24	0

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 physical testbed based on the VESNA sensor platform which is used in several wireless sensor network testbeds in Slovenia [Smolnikar et al. 2011]. The VESNA sensor node is equipped with a ST ARM Cortex M3 32 bit microcontroller running at 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 and a Contiki 2.5 OS port connected via serial line to a Lenovo X200 machine (Intel Core Duo CPU @2.53 GHz with 4GB or RAM). 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 and Eclipse Helios.

Image size. We first compare the components of the Rime stack against the components of the CRime stack in terms of the image size. Table III summarizes the size of the code (.text), the size of the initialized (.data) and non-initialized (.bss) static variables. It can be seen that the Rime and CRime components differ only in the size of the code.

The c_abc code size is 36 bytes larger than the code size of the corresponding abc module. 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 multihop communication primitive is notably 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.

For the broadcast, polite and unicast communication primitives, the CRime version results in smaller code size than the Rime version. This is because in CRime the functions do not directly call corresponding functions from the modules below. In CRime

Table IV: 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 V: 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 Table IV). 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

this overhead is solved by the amodule component. For instance, if we add in the c_broadcast module explicit calls to the c_abc module, each such call increases the size of the code by 8 bytes.

The stbroadcast, ipolite and stunicast modules from Rime have no direct equivalent in CRime. As discussed in the architectural comparison between the modules (see Section 3.2), the functionality of these is replaced by triggers and by the modular composition of the stack. The two CRime only specific modules, which enable modularity, are the amodule and the stack. The code footprint of these is relatively large compared to the other modules. They also use some static variables as can be seen in the corresponding .data and .bss columns of Table III. The set of 5 link estimation modules present in CRime and described in Section 7 have relatively larger sizes of 1117 to 1689 bytes mostly due to the more complex application logic and the additional data structures required to store the statistics and learned models. The c_echo_app module implements a very simple functionality of printing the received packets, therefore it has the smallest size of 173 bytes.

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 IV and in Table V, while the difference is listed in Table VI. It can be seen that the size of the code of the applications which use CRime stacks is about 16 to 17% larger (13.000 bytes) while the absolute size difference of the initialized and uninitialized data sections is below 0.1%.

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 VII 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.

It can be seen that in both stacks, the most time consuming operation is sending a packet. Rime needs on average 104 µs to send one packet while CRime needs 380 µs, 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 con-

Table VI: 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 IV and Table V.

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

Table VII: CRime vs Rime processing speed (results averaged over 100 runs).

Rime operations			CRime operations		
Name	Duration[µs]	Duration[%]	Name	Duration[µs]	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	8.5	close	67.0	5.7
Total	256.5	99.8		622	99.9

suming 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 propagates only 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 256 µs. CRime needs 622 µs 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 VIII 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 µs, 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.

It can be clearly seen in this breakdown in Table VIII that the overhead for the send operation where checks for walking through the tree and trigger handling is quite high, represent 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

Table VIII: The cost of stack and amodule abstractions in terms of processing speed (results averaged over 100 runs).

CRime operations					
Name of operation	Duration of the overall operation [µs]	Name of the executed functions	Duration of the executed functions [µs]	amodule & 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

Table IX: CRime vs Rime energy consumption.

Rime power consumption		CRime power consumption		$\Delta P / P_R$
Application name	Consumed power (P_R) [mW]	Application name example_crime	Consumed power (P_R) [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

the different implementations of the tree walking algorithms (for the send operation the trigger handling also has to be accounted for).

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 IX.

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.

8.2. Qualitative Assessment 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 graphical simulation tools. Furthermore, the workbench allows module developers and experimenters to focus on stack composition rather than on coding and debugging. The CRime module library 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. In order to evaluate the usability and user friendliness of ProtoStack and to collect some feedback on how to improve the overall tool, we carried out a small qualitative assessment involving a group of first time users which were not involved in any aspect of the development of the framework of ProtoStack. Still, they are familiar with the basic goals and principles that are driving the development of such tool as well as with the domain specific background knowledge, so we could refer to them as representative end users/experimenters. This qualitative assessment is thus by no means a thorough and statistically representative study but it rather aims to help the readers to get a feeling on the friendliness and the learning curve of the tool.

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[4].
To perform a statistically significant study, we would need to identify interested researchers from the research areas mentioned above and incentivize them to use the tool and provide feedback, 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 number 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.

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

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 separately. The developer also had to answer a set of three questions about the way the user was using the tool.

The study groups profiles are summarized in Table X and Table XI. The 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 was just starting, most had 2 years experience and one had 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 X. 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 were familiar with the concept of 'service composition'as shown in the results summarized in Table XI.

The feedback received from the user group can be summarized as follows. 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-up messages following interactions with the GUI. The toolbar was rated with 5 points (out of 5) for intuitiveness by all

[4]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 over 13 years. 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.

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

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

Table XI: 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

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.

The users were observed performing the two tasks from the questionnaire. The first task required them to set up a unicast communication stack while the second task required the set up of a multihop stack. The time to completion as well as the number of failed saving attempts of a new stack were collected. It seems that both stacks were fast to complete. The average completion time for the first task was 160 seconds with less than 1 (0.83) error message on average. The average completion time for the second task was slightly lower at 126 seconds with the average number of error messages slightly above 1 (1.16). The complexity of the second task was slightly higher compared to 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.

9. COMPARISON TO AND EVALUATION WITH RESPECT TO RELATED WORK

The idea of composing communication services has already been investigated, albeit using different naming, in several papers such as the X-Kernel [Hutchinson and Peterson 1991], the Click modular router [Kohler et al. 2000], the role based architecture (RBA) [Braden et al. 2003], the sensor network architecture (SNA) [Tavakoli et al. 2007b], the autonomic network architecture (ANA) [Bouabene et al. 2010], the infor-

mation driven architecture IDRA [De Poorter et al. 2011], REMORA [Taherkordi et al. 2011] and Rime.

The most relevant previous work related to this paper can be grouped in two main clusters. One cluster comprises frameworks and tools which offer complete support for experimentation with composeable stacks. Besides looking at the abstractions they introduce to allow composeability, we also look at the composition model, the approach to configuration and the implementation. This cluster includes the related work that can be compared to the overall framework introduced by this paper. The second cluster comprises existing work related to modular stacks where we pay attention to their scope with respect to the layers of the OSI stack and other design aspects such as support for cross-layer design, header and payload manipulation, etc. This cluster includes work that can be compared to the module library introduced by the paper. Detailed analyses on how the proposed framework and the ProtoStack reference implementation relates to existing work is provided in Section 8.

9.1. Related experimentation tools

The most representative examples for this cluster are x-Kernel, Click, SNA, and REMORA.

X-Kernel [Hutchinson and Peterson 1991] 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 a 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 [Kohler et al. 2000] 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.

The NEST project [NES 2014] focused on providing an open experimentation environment that would accelerate the development of algorithms, services, and their composition into applications in the area of sensor networks. The experiences acquired during the project led to the development of abstractions suitable for modularizing sensor network stacks to enable easy protocol development and experimentation as well as code re-use. For instance, in [Ee et al. 2006][Tavakoli et al. 2007b] the authors aim 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 [Polastre et al. 2005], the aim was to build a modular architecture for sensor networks (SNA). In [Tavakoli et al. 2007b] they argumented that a declarative programming model represents a good mechanism for decoupling the application logic from the actual implementation [Tavakoli et al. 2007b] thus helping the user and lowering the entry barrier in experi-

Table XII: 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	✓	✓	✓	✓
NEST/SNA	•	✓	✓	✓
REMORA	•	✓	✓	✓
ProtoStack	✓	✓	✓	✓

mentation and development in the area of sensor networks. As a result, a declarative sensor network (DSN) architecture that uses a high level declarative specification language and a compiler to permit the user to specify an application was proposed in [Tavakoli et al. 2007a]. The main problem solved by this work is the provision of an easy way to programming sensor networks.

REMORA [Taherkordi et al. 2011] is a component based framework that aims to provide a well structured programming paradigm to ease high-level software development in wireless sensor networks. It provides a set of components implementing different functionality and an XML syntax for describing these components. The REMORA engine running on a development machine converts the component descriptions generated via the development box into a REMORA application that is then compiled. The resulting application image is then distributed to sensor nodes which are locally running the REMORA runtime, which is supporting the applications and provides mechanisms for event management.

9.1.1. On the proposed framework. The contributions of our paper with respect to the state of the art in this field include the formalization of the four component framework that is generic enough and well suited for the design and experimentation with dynamic composition of communication services. This claim is supported by the fact that existing experimentation tools that have proved to be useful for stack composition by passing the test of time and having wide user communities evolved to fit into such framework, although never formalized thus far. For instance, x-Kernel introduced modular design principles for operating systems to support high speed networking. These principles enabled the modularization of protocols and correspond to the module library described by our framework. The initially supporting Sun3 workstations and other, more recent, physical devices that can be used for experimentation correspond to the physical testbed as described by the framework. The declarative language corresponds to textual graph description language that allows description of x-kernel protocols while the workbench corresponds to x-Kernel's graph editor. Click also evolved over time to support all four components as depicted in Table XII. With respect to the area of sensor networks, the concepts and tools developed within or as a result of the NEST project, including SNA, include all components but the workbench. The more recent REMORA framework also excludes the workbench while ProtoStack includes all four.

9.1.2. On the ProtoStack reference implementation. The comparison between the tools which support the composition of communication services from the perspective of the requirements identified in Section 2 is summarized in Table XIII. It can be seen that all five 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. To the best of our knowledge, SNA does not currently address this requirement. With respect to remote

Table XIII: 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	✓	✓	✓	✓	•
NEST/SNA	✓	✓	✓	•	•
REMORA	✓	✓	✓	✓	✓
ProtoStack	✓	✓	✓	✓	✓

experimentation support, only REMORA and ProtoStack explicitly address this aspect, however there is a major difference between the approaches. REMORA assumes that the application is composed from the available components on a development machine and then distributed to the network of sensors. The REMORA development machine, on which the user is working, has to be somehow connected to the network or sensors most likely through a gateway. REMORA focuses on component assembly and distribution and provides a code distribution use case. ProtoStack on the other hand is fully web based and designed to support remote experimentation. With ProtoStack, the sensor network can be located at one physical location, the server that renders the workbench, has the module library, performs the consistency checking, etc can be located at another physical location, while the user can be located at the third physical location. This is a major difference with respect to other tools that all assume that the user is physically located or connected to the actual development machine while ProtoStack, besides enabling that, also enables the user to be completely ignorant of any development aspect and just focus on the composition and configuration.

A feature based comparison of ProtoStack, the reference implementation of the framework we propose, with existing experimentation tools identified in Section 9.1, is provided in Table XIV. It can be seen that, similar to the other tools, ProtoStack also introduces a set of abstractions to support modularity. X-Kernel introduces three abstractions, Click and REMORA introduce only one while ProtoStack also introduces three. ProtoStack's amodule seems closest to Click's element, both implementing well contained functionality. Specific to CRime is the pipe abstraction allowing powerful cross-layer optimizations and cognitive networking. Additionally, ProtoStack's stack abstraction allows running concurrent protocol stacks. Similar to the first two 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 three systems use a graph composition model, ProtoStack uses a tree model which seems to be sufficient for the current state of the tool's development.

To complement the first time user study provided in Section 8.2, we provide a feature-based comparison of the GUIs offered by the existing tools in Table XV. In order to visually compose and configure a protocol stack using the x-Kernel and Click GUIs, the user must use a graph editor installed on the workstation while ProtoStack allows this to be done in a browser irrespective of where the server that generates the web page is located (i.e. on the workstation or somewhere in the cloud). This makes the ProtoStack GUI independent of any operating system. Additionally, once a new amodule is developed, this will be automatically added to the GUI by parsing the Turtle comments written in the .c source file. No additional configuration or other operations need to be performed by the user to see the upgrade. In summary, the ProtoStack GUI fully benefits not only from the advantages of web technologies and, as a consequence, of remote configuration, but also from the advantages of semantic web technologies thus of increased interoperability and integration with federations.

Table XIV: Comparison of related experimentation tools.

Project	Abstraction	Composition model	Configuration	Implementation
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++
REMORA	components,	graph	declarative language + Auto-generated C code	C-like
ProtoStack	amodule, pipe stack	tree	Graph editor + declarative language + Auto-generated C code	C

Table XV: Comparison of the GUIs offered by the existing tools.

Name	Rendering	Local	OS dependent	Auto-configurable
X-Kernel	graph editor	yes	yes	no
Click	graph editor	yes	yes	no
ProtoStack	browser	yes and no	no	yes

9.2. Related approaches to modular stacks

There are many papers about different ways to generalize protocol design and processing so in this subsection we aim at providing a brief overview of those approaches that are most related to our work as well as qualitative comparison. Thus, in addition to the examples mentioned in the previous subsection (i.e. x-Kernel, Click, NEST/SNA (including SP, MNL and DSN) and REMORA) which all include, besides other contributions, also abstractions for modular stacks, we refer in the following also to the dynamic network architecture (DNA) [O'Malley and Peterson 1992], role based architecture (RBA) [Braden et al. 2003], Rime [Dunkels et al. 2007], autonomic network architecture (ANA) [Bouabene et al. 2010] and the information driven architecture (IDRA) [De Poorter et al. 2011].

DNA [O'Malley and Peterson 1992] is an architecture that builds on the x-Kernel that provides a fine grained modularization of protocol functionalities, enables the construction of complex protocols by connecting the modules in complex graphs and enables the application to select the topology of the protocol graph. ProtoStack is quite similar in the way that it also enables the composition of CRime modules into more complex functionality, however, this composition is performed by the user in the current implementation. Additionally, ProtoStack provides architectural support for automating this composition as discussed in the use case presented in Section 6.

The authors of RBS [Braden et al. 2003] noticed that with traditional layered architectures, new services fit poorly into the existing layered structure. They provide examples of inter-layer protocols such as multi-protocol label switching (MPLS) at layer 2.5, IPsec at layer 3.5 and transport layer security at layer 4.5. In order to provide architectural support for adding new services, the authors introduce the *role* and the *role-specific header* (RBH) concepts. The *role* abstraction corresponds to a modular protocol unit that allows splitting network functionality in smaller units than traditional layered architecture allow. A set of *role-specific headers* form a data structure that corresponds to a sequence of traditional packet headers with the exception that RBHs are not removed once they have been inspected by a role. The authors estimate that the exact functional breakdown in roles is hard to pre-determine but experience will help with the modularization. Their estimate was that a real network using RBA would need a relatively few well known roles that are standardized.

The Rime protocol stack [Dunkels et al. 2007] 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. The conceptual model behind Rime is modular and assumes static configuration of the sets of modules. Rime services are somewhat similar to the roles in RBH and, same as RBH, decouples functionality from header generation. While RBH is very generic and provides no reference implementation, Rime incorporates concepts from RBH, adapts them to the domain of sensor networks and provides a reference implementation. Being inspired by Rime, CRime also incorporates some of the concepts from RBH and additionally introduces three concepts that enable dynamic composition and configuration of communication services, which is not supported by Rime.

The authors of ANA [Bouabene et al. 2010], introduce a set of generic abstractions that allow for the coexistance of multiple and diverse networking styles and protocols from clean slate design approaches, including modular network stacks, to legacy internet stacks. The aim is to support functional scaling which means that a network is able to extend both horizontally (adding more functionality) as well as vertically (different ways of integrating abundant functionality). In order to achieve this, four abstractions have been introduced: *network compartment, information channel, functional block* and *information dispatch point*. A *network compartment* is in a way a slice or a part of a network consisting of several nodes that are using any combination of addressing, naming, routing, networking mechanisms, protocols, packet formats, etc. However, these compartments must support a generic API that wraps their internal operation with generic constructs. For instance, a set of sensor nodes running 6LoWPAN could form a network compartment while a set of machines running an IPv4 stack could form a second compartment. These two compartments would interact through a generic API. The *information channel* represents an abstraction for the communication channel provided inside a network compartment such as unicast, multicast, reliable stream, etc. The *functional block* corresponds to a protocol entity that generates, consumes, processes and forwards information. The *functional blocks* can take the form of an algorithm or an entire protocol, i.e. their granularity is not specified in advance by the architecture. Finally, the *information dispatch point* that provides a mechanism to access the *functional blocks* to which they are attached. The binding of an *information dispatch point* is dynamic and can change over time as the network stack is reconfigured.

CRime and ANA share the concepts of modularization and composition of services, however the mechanisms for achieving these are different. While ANA's functional blocks are similar to CRime's amodules, the other abstractions introduced by ANA are not present in CRime as such. The ANA framework essentially organizes network functionality in modules and in slices. Besides supporting modularity through the functional blocks it also supports virtualization through the network compartments. CRime on the other hand can run multiple network stacks in parallel on the same radio interface, it supports parallelization rather than virtualization. Additionally, ANA also supports heterogeneity of networks though the API exposed by the network compartments. ANA targets traditional networks, its aim is to support autonomous networks that can self-assemble and the reference implementation is meant for Linux workstations. CRime, on the other hand, targets constrained devices and the aim is to provide modularity. In order to enable autonomy, ANA provides a mechanism to search for functional blocks (i.e. functional blocks are labeled, thus can be looked for based on labels). CRime amodules are on the other hand annotated using semantic web language. As a result, the CRime modules are not only searchable on the server, but they can also be reasoned about to eventually increase the autonomy of the network as discussed in the use case presented in Section 6.

In IDRA [De Poorter et al. 2011], the authors propose a networking architecture that that aims to reduce the complexity of developing new protocols for wireless sensor networks, support advanced network requirements such as QoS and support heterogeneous networks. The main abstractions introduced by IDRA are the *information exchange, packet facade, shared queue, information waiting space* and *classifier*. Through the *information exchange* abstraction, the creation of packets is delegated to the system instead of being handled by each protocol as is the case with the layered model. The *packet facade* is the means through which protocols interact with packets by setting or reading packet attributes such as destination, time to live, etc. These packet attributes are stored in packet part descriptors that describe how and where attributes are stored in a header. The *shared queue* is a system wide shared packet buffer that can be accessed by any protocol, thus providing architectural support for cross-layer information exchange and minimizing resource utilization on already constrained devices. The *information waiting space* is an abstraction that enables the system to aggregate data that are not time sensitive, from multiple protocols, to achieve energy efficiency and support increased throughput. IDRA is able to support multiple protocol stacks running at the same time and enables run-time reconfiguration of these stacks. In IDRA, the protocols that process the packets are selected by a *classifier* based on pre-specified filters.

IDRA takes a more generic approach to modularity compared to the CRime was designed to do. As a result, IDRA provides full flexibility with respect to packet processing, making it possible to use different protocols to process two different packets coming from the same protocol from the sending device, and it can use different networks stacks for sending and receiving packets. IDRA packet processing modules and their sequence is determined locally on the node by the classifier after they have registered their filters, however IDRA does not explicitly address the labelling or descriptions of the packet processors in the sense ANA and ProtoStack do. From the perspective of discoverability of components and the use of reasoning techniques for stack composition, ProtoStack, REMORA and ANA exhibit a higher degree of automation. Figure 20 provides a relative plot of the related work discussed so far in terms of flexibility and automation. While most of the related work provide conceptualizations that enable a higher degree of flexibility in terms of networking (run time) recomposition and reconfiguration, ProtoStack is superior in terms of supporting automation in the sense of providing architectural support for reaching the goal of machine reasoning for protocol stack composition as described in Section 6.

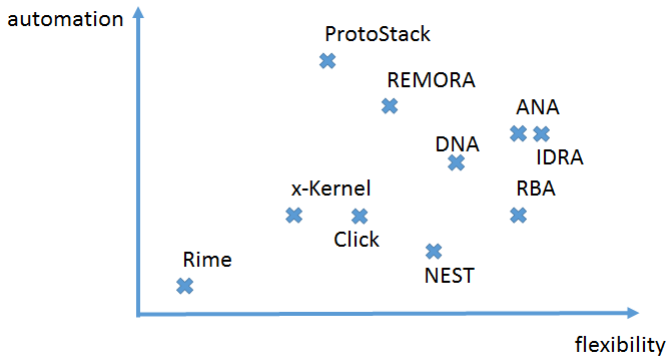


Fig. 20: Related work according to the supported flexibility and automation.

A direct quantitative comparison of ProtoStack with most of the described approaches related to modular composition of protocol stacks is not possible to do either because they are not targeted to sensor networks, or because they are conceptually different. IDRA, however, provides a reference implementation for wireless sensor systems that enables a rough comparison to CRime in terms of the overhead the IDRA system introduces on the sensor node and thus indicates the operation costs for higher degree of flexibility. IDRA's code size amounts to 27kB while CRime's amounts to almost 7kB (note that IDRA is implemented in TinyOS while CRime is implemented in Contiki OS). Obviously, the increased flexibility of the IDRA system has to incur a cost and increased code size is part of it. The combined IDRA and radio overhead for sending a packet on a TinyOS node with a CC2420 transceiver is 12 ms while the CRime and radio overhead on a VESNA node with a CC1101 transceiver is 380 μs.

10. CONCLUSIONS

In this paper we proposed a framework for dynamic composition of communication services which is well suited to facilitate research and prototyping on real embedded experimental infrastructures. By using the concept of composeability, the framework encourages modular component development for various networking functions, therefore promoting code re-use.

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 identified requirements.

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 for the introduced modularity and support for experimentation in the areas of service oriented networking and cognitive networking 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 to 17% larger footprint, it takes 2.4 more time to execute open→ send→ recv→ close sequence of operatios 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 and support for the implementation of the knowledge plane seems to pay off.

REFERENCES

2013. Cognitive Radio Experimentation World project. (March 2013). http://www.crew-project.eu/

2013. Contiki Hardware. (May 2013). http://www.contiki-os.org/hardware.html

2013. Future Internet Research and Experimentation. (March 2013). http://cordis.europa.eu/fp7/ict/fire/

2013. GENI.net Global Environment for Network Innovations. (March 2013). http://www.geni.net

2013. Resource Description Framework (RDF) Model and Syntax Specification. (March 2013). http://www.w3.org/TR/PR-rdf-syntax/

2014. NEST Project at Berkley. (2014). http://nest.cs.berkeley.edu/nest-index.html

Nouha Baccour, Anis Koubaa, Luca Mottola, Marco Antonio Zuniga, Habib Youssef, Carlo Alberto Boano, and Mario Alves. 2012. Radio link quality estimation in wireless sensor networks: a survey. *ACM Transactions on Sensor Networks (TOSN)* 8, 4 (2012), 34.

Ghazi Bouabene, Christophe Jelger, Christian Tschudin, Stefan Schmid, Ariane Keller, and Martin May. 2010. The autonomic network architecture (ANA). *Selected Areas in Communications, IEEE Journal on* 28, 1 (2010), 4–14.

Robert Braden, Ted Faber, and Mark Handley. 2003. From protocol stack to protocol heap: role-based architecture. *ACM SIGCOMM Computer Communication Review* 33, 1 (2003), 17–22.

Gomez Carles, Boix Antoni, and Paradells Josep. 2010. Impact of LQI-based routing metrics on the performance of a one-to-one routing protocol for IEEE 802.15. 4 multihop networks. *EURASIP Journal on Wireless Communications and Networking* 2010 (2010).

Edgar F Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377–387.

Eli De Poorter, Evy Troubleyn, Ingrid Moerman, and Piet Demeester. 2011. IDRA: A flexible system architecture for next generation wireless sensor networks. *Wireless Networks* 17, 6 (2011), 1423–1440.

Leigh Dodds. 2006. Slug: A semantic web crawler. In *Proceedings of Jena User Conference*, Vol. 2006.

Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. 2004. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. IEEE, 455–462.

Adam Dunkels, Fredrik Österlind, and Zhitao He. 2007. An adaptive communication architecture for wireless sensor networks. In *Proceedings of the 5th international conference on Embedded networked sensor systems*. ACM, 335–349.

Cheng Tien Ee, Rodrigo Fonseca, Sukun Kim, Daekyeong Moon, Arsalan Tavakoli, David Culler, Scott Shenker, and Ion Stoica. 2006. A modular network layer for sensornets. In *USENIX OSDI*, Vol. 6.

Carolina Fortuna and Mihael Mohorcic. 2008. Advanced access architecture for efficient service delivery in heterogeneous wireless networks. In *Communications and Networking in China, 2008. ChinaCom 2008. Third International Conference on*. IEEE, 1173–1177.

Carolina Fortuna and Mihael Mohorčič. 2009a. Dynamic composition of services for end-to-end information transport. *Wireless Communications, IEEE* 16, 4 (2009), 56–62.

Carolina Fortuna and Mihael Mohorčič. 2009b. Trends in the development of communication networks: Cognitive networks. *Computer Networks* 53 (2009), 1354–1376.

Carolina Fortuna and Mihael Mohorcic. 2010. A local knowledge base for the media independent information system. *Future Internet-FIS 2009* (2010), 15–24.

Carolina Fortuna, M Mohorčič, and B Filipič. 2008. Multiobjective optimization of service delivery over a heterogeneous wireless access system. In *Wireless Communication Systems. 2008. ISWCS'08. IEEE International Symposium on*. IEEE, 133–137.

Vanhie-Van Gerwen, Stefan Bouckaert, Ingrid Moerman, Piet Demeester, and others. 2011. Benchmarking for wireless sensor networks. In *SENSORCOMM 2011, The Fifth International Conference on Sensor Technologies and Applications*. 134–139.

Mattijs Ghijsen, Jeroen van der Ham, Paola Grosso, and Cees de Laat. 2012. Towards an Infrastructure Description Language for Modeling Computing Infrastructures. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*. IEEE, 207–214.

IEEE 802.21 Working Group and others. 2008. IEEE P802.21/D11.0 Draft IEEE standard for local and metropolitan area networks: Media independent handover services. *IEEE p802* (2008), D00.

Norman C. Hutchinson and Larry L. Peterson. 1991. The x –Kernel: an architecture for implementing network protocols. *Software Engineering, IEEE Transactions on* 17, 1 (1991), 64–76.

E Kim, D Kaspar, C Gomez, and C Bormann. 2011. Problem Statement and Requirements for 6LoWPAN Routing. (2011). http://tools.ietf.org/html/draft-tavakoli-hydro-01

Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.

Matthias Kovatsch, Simon Duquennoy, and Adam Dunkels. 2011. A low-power CoAP for Contiki. In *IETF Internet Draft*. IEEE, 855–860.

Douglas B Lenat and Ramanathan V. Guha. 1991. The evolution of CycL, the Cyc representation language. *ACM SIGART Bulletin* 2, 3 (1991), 84–87.

Philip Levis, Samuel Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric A Brewer, and David E Culler. 2004. The Emergence of Networking Abstractions and Techniques in TinyOS.. In *NSDI*, Vol. 4. 1–1.

Cynthia Matuszek, John Cabral, Michael Witbrock, and John DeOliveira. 2006. An introduction to the syntax and content of Cyc. In *Proceedings of the 2006 AAAI spring symposium on formalizing and compiling background knowledge and its applications to knowledge representation and question answering*, Vol. 3864.

Deborah L McGuinness, Frank Van Harmelen, and others. 2004. OWL web ontology language overview. *W3C recommendation* 10, 2004-03 (2004), 10.

Sean W O'Malley and Larry L Peterson. 1992. A dynamic network architecture. *ACM Transactions on Computer Systems (TOCS)* 10, 2 (1992), 110–143.

Joseph Polastre, Jonathan Hui, Philip Levis, Jerry Zhao, David Culler, Scott Shenker, and Ion Stoica. 2005. A unifying link abstraction for wireless sensor networks. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*. ACM, 76–89.

Eric PrudHommeaux, Andy Seaborne, and others. 2008. SPARQL query language for RDF. *W3C recommendation* 15 (2008).

Chunmei Ren and Daihong Jiang. 2011. A New Ontology of Resource Specification for Wireless Sensor Networks. In *Information Technology, Computer Engineering and Management Sciences (ICM), 2011 International Conference on*, Vol. 2. IEEE, 138–140.

Luis Sanchez, Luis Muñoz, Jose Antonio Galache, Pablo Sotres, Juan R Santana, Veronica Gutierrez, Rajiv Ramdhany, Alex Gluhak, Srdjan Krco, Evangelos Theodoridis, and others. 2013. SmartSantander: IoT Experimentation over a Smart City Testbed. *Computer Networks* (2013).

Blake Shepard, Cynthia Matuszek, C Bruce Fraser, William Wechtenhiser, David Crabbe, Z Gungordu, John Jantos, Todd Hughes, Larry Lefkowitz, Michael Witbrock, and others. 2005. A Knowledge-based approach to network security: applying Cyc in the domain of network risk assessment. In *Proceedings of the National Conference on Artificial Intelligence*, Vol. 20. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 1563.

Miha Smolnikar, Carolina Fortuna, Matevž Vučnik, Marko Mihelin, and Mihael Mohorčič. 2011. Wireless sensor network testbed on public lighting infrastructure. In *EcoSense 2011 The Second International Workshop on Sensing Technologies in Agriculture, Forestry and Environment*. 6–7.

Kannan Srinivasan, Prabal Dutta, Arsalan Tavakoli, and Philip Levis. 2010. An empirical study of low-power wireless. *ACM Transactions on Sensor Networks (TOSN)* 6, 2 (2010), 16.

Amirhosein Taherkordi, Frédéric Loiret, Romain Rouvoy, and Frank Eliassen. 2011. A generic component-based approach for programming, composing and tuning sensor software. *Comput. J.* 54, 8 (2011), 1248–1266.

Arsalan Tavakoli, David Chu, Joseph M Hellerstein, Phillip Levis, and Scott Shenker. 2007a. A declarative sensornet architecture. *SIGBED Rev* 4, 3 (2007), 55–60.

Arsalan Tavakoli and David Culler. 2009. Hydro: A hybrid routing protocol for low-power and lossy networks. (2009). http://tools.ietf.org/html/draft-tavakoli-hydro-01

Arsalan Tavakoli, Prabal Dutta, Jaein Jeong, Sukun Kim, Jorge Ortiz, David E Culler, Philip Levis, and Scott Shenker. 2007b. A modular sensornet architecture: past, present, and future directions. *SIGBED Review* 4, 3 (2007), 49–54.

Ryan W Thomas, Daniel H Friend, Luiz A DaSilva, and Allen B MacKenzie. 2006. Cognitive networks: adaptation and learning to achieve end-to-end performance objectives. *Communications Magazine, IEEE* 44, 12 (2006), 51–57.

Milorad Tosic, Ivan Seskar, and Filip Jelenkovic. 2012. TaaSOR–Testbed-as-a-Service Ontology Repository. In *Testbeds and Research Infrastructure. Development of Networks and Communities*. Springer, 419–420.

Alec Woo, Terence Tong, and David Culler. 2003. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems*. ACM, 14–27.

Received May 2013; revised February 2014; accepted ????

Online Appendix to:
A Framework for Dynamic Composition of Communication Services

CAROLINA FORTUNA, Jozef Stefan Institute
MIHAEL MOHORCIC, Jozef Stefan Institute

A. 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.

B. SELECTED CRIME COMMUNICATION PRIMITIVES (COMPILED BASED ON THE CONTIKI 2.5 RIME

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.

C. THE CRIME ONTOLOGY

```
<?xml version="1.0"?>
```


DOI:http://dx.doi.org/10.1145/0000000.0000000

```
<!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="&cpan;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="&cpan;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: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
        <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 initializatio
        <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 -->
```