ProtoStack – a tool for remote experimentation with composable stacks

Carolina Fortuna

Jozef Stefan Institute,

Ljubljana, Slovenia

CREW training days,

Brussels, Feb 19-20 2013

Outline

- Motivation
  - Why is the composition of communication services relevant?
  - Why is experimenting in realistic environments difficult?
  - What kind of research can the composition of communication services support?
- The ProtoStack tool and its components
- The CRime module library
- The declarative language and the workbench
- Service oriented networks with ProtoStack
- Cognitive networks with ProtoStack

Motivation

Why is the composition of communication services relevant? (1/3)

- Speeds up the development cycle for communication technology

Design

Simulate

Emulate

Test

Deploy

Why is the composition of communication services relevant? (1/3)

- Speeds up the development cycle for communication technology

Design

Simulate

Emulate

Test

Deploy

- ***How come?***

Why is the composition of communication services relevant? (1/3)

- Speeds up the development cycle for communication technology

Design

Simulate

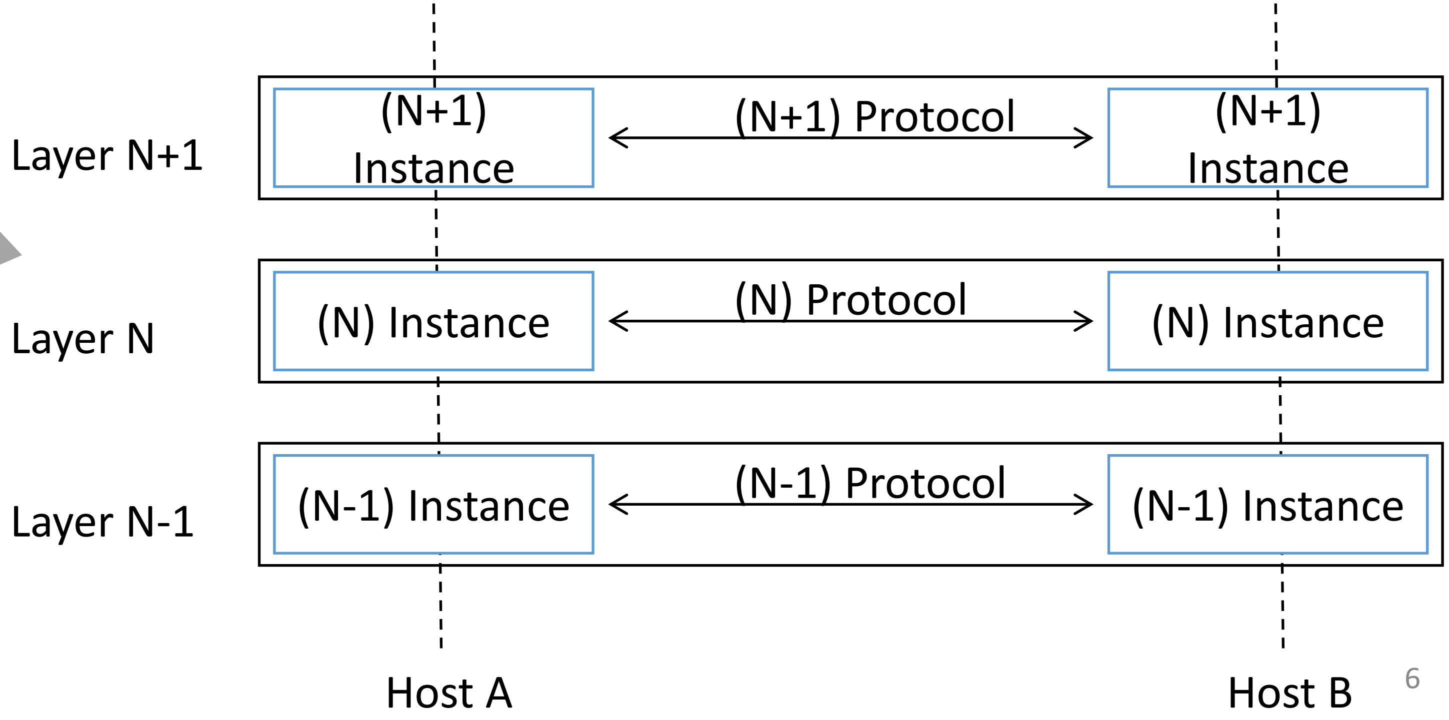
Emulate

Test

Deploy

- ***How come?*** Let’s see how communication networks function…

Hosts implement communication functionality across several layers of abstraction: link layer (physical connection between machines), network layer (logical connection between machines), transport layer (logical connection between processes)



Layer N+1

(N+1) Instance

(N+1) Protocol

(N+1) Instance

Layer N

(N) Instance

(N) Protocol

(N) Instance

Layer N-1

(N-1) Instance

(N-1) Protocol

(N-1) Instance

Host A

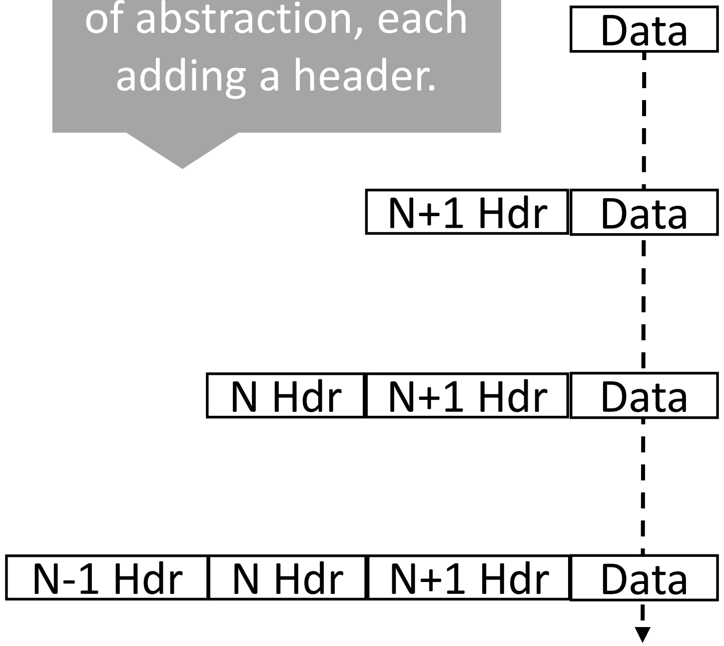
Host B

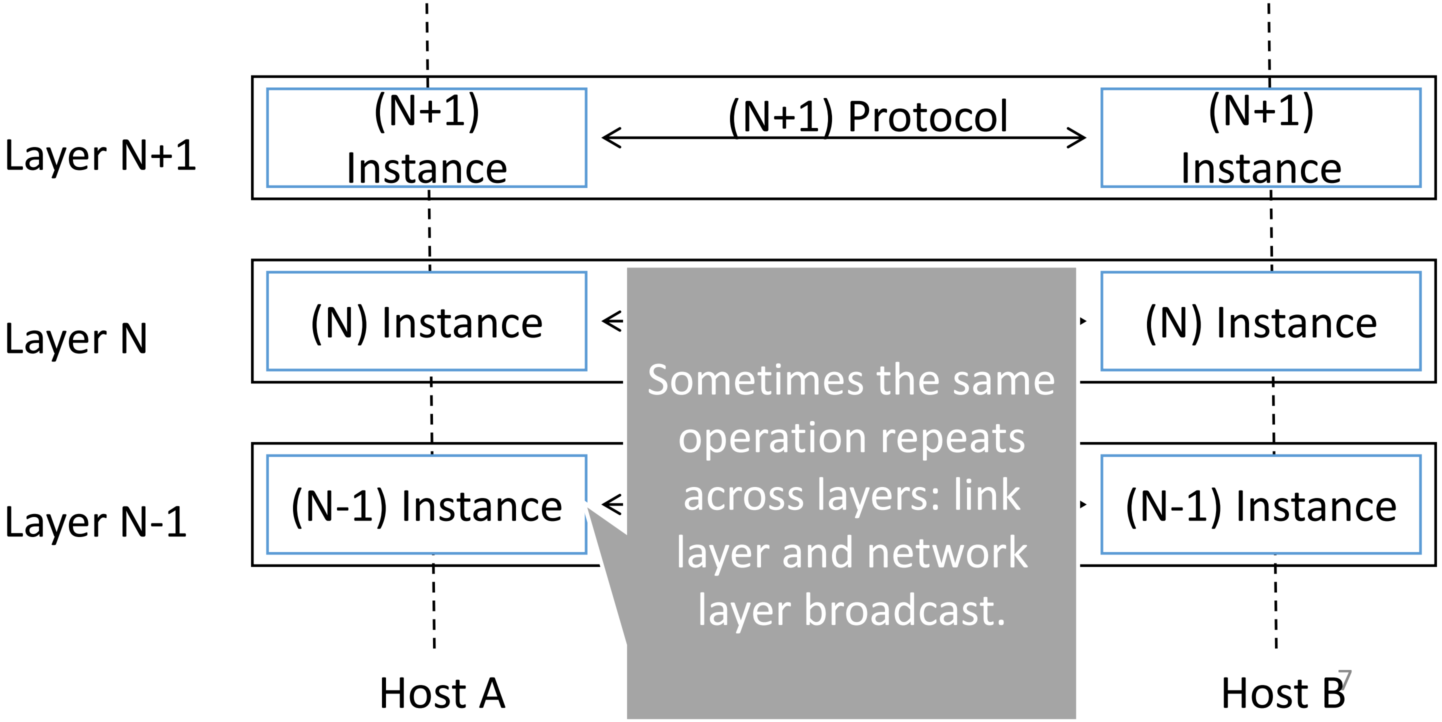
Why is the composition of communication services relevant? (1/3)

- Speeds up the development cycle for communication technology

Data are processed across several layers of abstraction, each adding a header.

Let's see how communication networks function…





Sometimes the same operation repeats across layers: link layer and network layer broadcast.

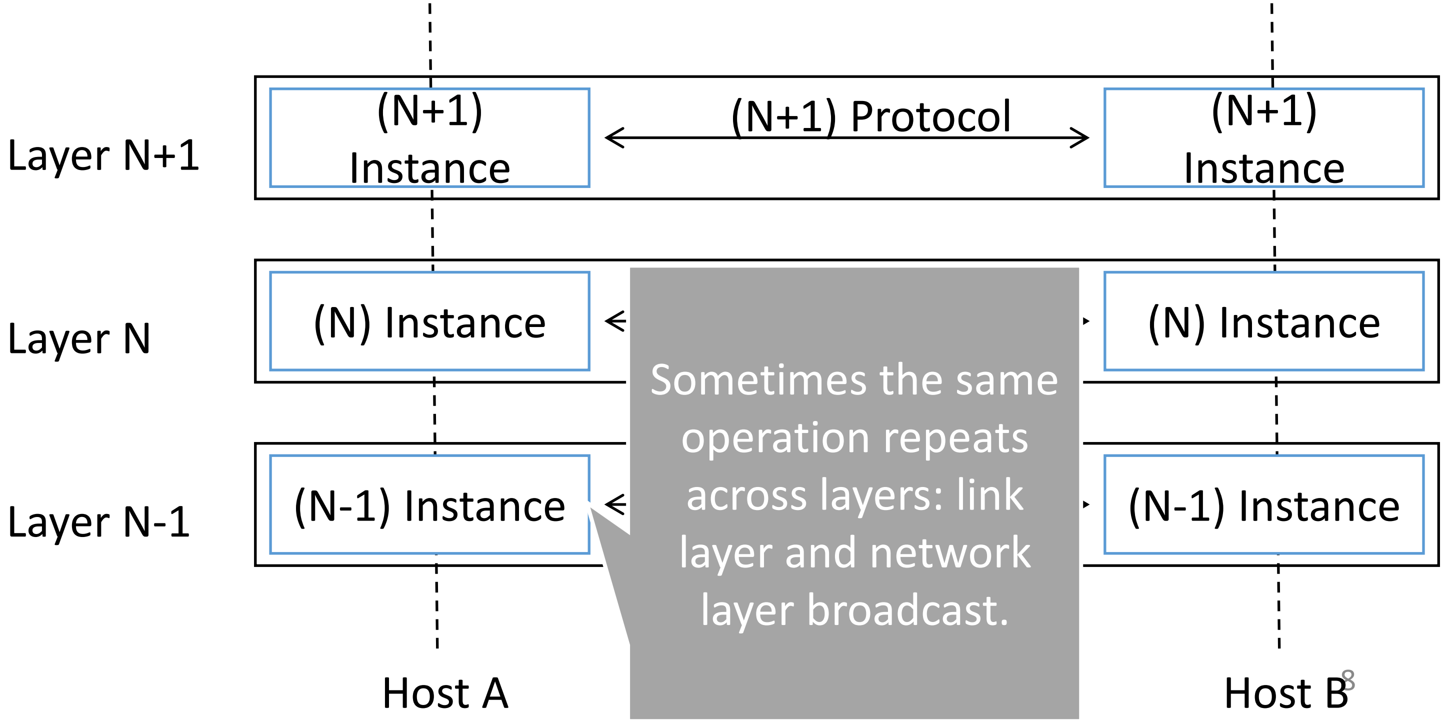
Why is the composition of communication services relevant? (1/3)

- Speeds up the development cycle for communication technology

- ***How come?*** Let's see how communication networks function…

What if we found an abstraction that:
- looks at operations as services
- allows composition of these services
- facilitates it in a dynamic way

With such abstraction, operations can be:
- implemented once and used many times
- can be layered in different ways
- the layering is semi-automatic



Sometimes the same operation repeats across layers: link layer and network layer broadcast.

Why is the composition of communication services relevant? (2/3)

- Speeds up the development cycle for communication technology

Design

Simulate

Emulate

Test

Deploy

- ***How come?*** Let's see how communication networks function…

- ***So, why hasn't this been done before?***
  - because part of the communication network processing has been implemented in hardware
  - because communication network design precedes significant breakthroughs in software engineering

Why is the composition of communication services relevant? (3/3)

- Speeds up the development cycle for communication technology

Design

Simulate

Emulate

Test

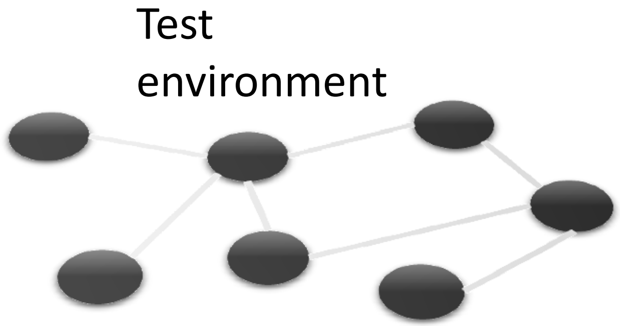
Deploy

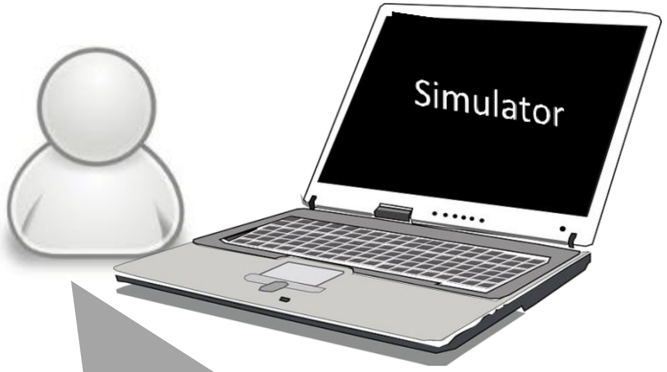
- ***How come?*** Let’s see how communication networks function…

- ***So, why hasn’t this been done before?***

- ***And why can it be done now?***
    - because software configurable networks and software defined networks allow it
    - because service oriented design has been proposed

Why is experimenting in realistic environments difficult?

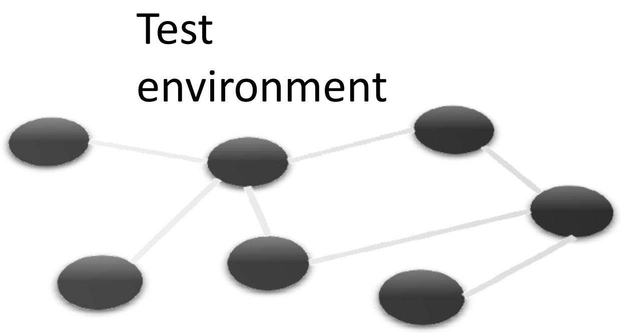




- What simulator to use?
- Is there a model I can adapt?
- What are the scenarios to investigate?
- What are the parameters and variables?

- What test environment to use?
- How can I access the environment?
- How can I configure the environment?
- Is there any code I can adapt?
- What are the scenarios to investigate?
- What are the parameters and variables?
- Is there anything wrong with my model?
- Is the environment malfunctioning?

Why is experimenting in realistic environments difficult?



More distractions, more worries, more questions to answer.

- What simulator to use?
- Is there a model I can adapt?
- What are the scenarios to investigate?
- What are the parameters and variables?

- What test environment to use?
- How can I access the environment?
- How can I configure the environment?
- Is there any code I can adapt?
- What are the scenarios to investigate?
- What are the parameters and variables?
- Is there anything wrong with my model?
- Is the environment malfunctioning?

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

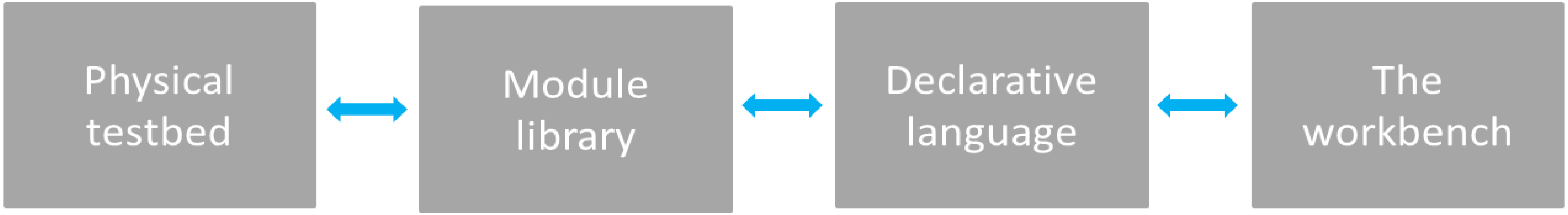
- The research that monolithic approaches supported thus far …

+

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

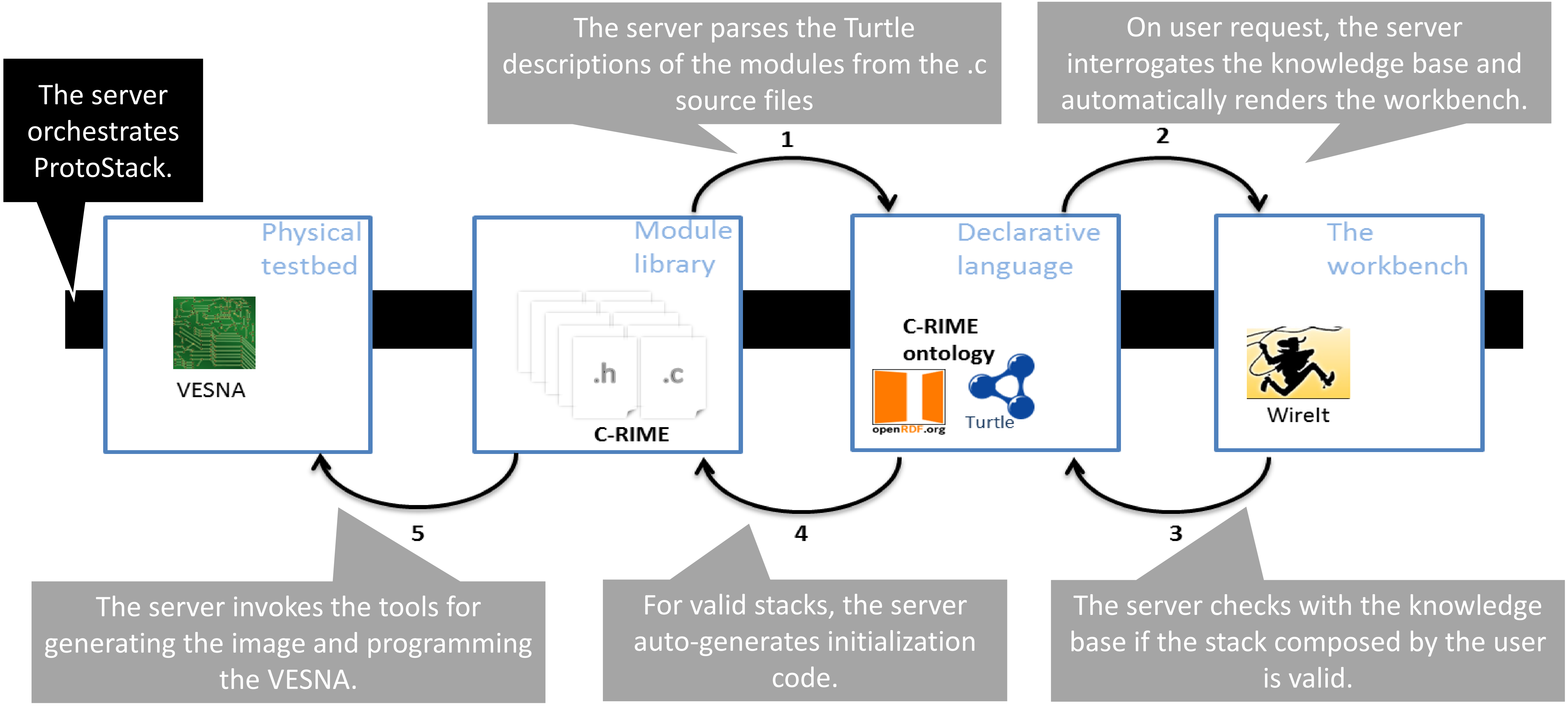
The ProtoStack tool and its components

The components of the ProtoStack



Physical testbed	Module library	Declarative language	The workbench
is a set of machines on which the stack built by the composition of services is deployed and tested	the source code of the modules used for composing communication services. Besides the code for the modules it also contains additional code necessary for compiling and linking the binary image	is an intermediate level of abstraction between a user interface such as the workbench and the program code. It is used to instantiate and configure modules from the module library. Can accommodate validity checking, error detection, etc.	is thought of as a control panel which allows the experimenter to configure, start, run, retrieve and visualize the results of an experiment.

On how ProtoStack operates



The server orchestrates ProtoStack.

Physical testbed

VESNA

Module library

.h

.c

C-RIME

Declarative language

C-RIME ontology

openRDF.org

Turtle

The workbench

WireIt

1

The server parses the Turtle descriptions of the modules from the .c source files

2

On user request, the server interrogates the knowledge base and automatically renders the workbench.

3

The server checks with the knowledge base if the stack composed by the user is valid.

4

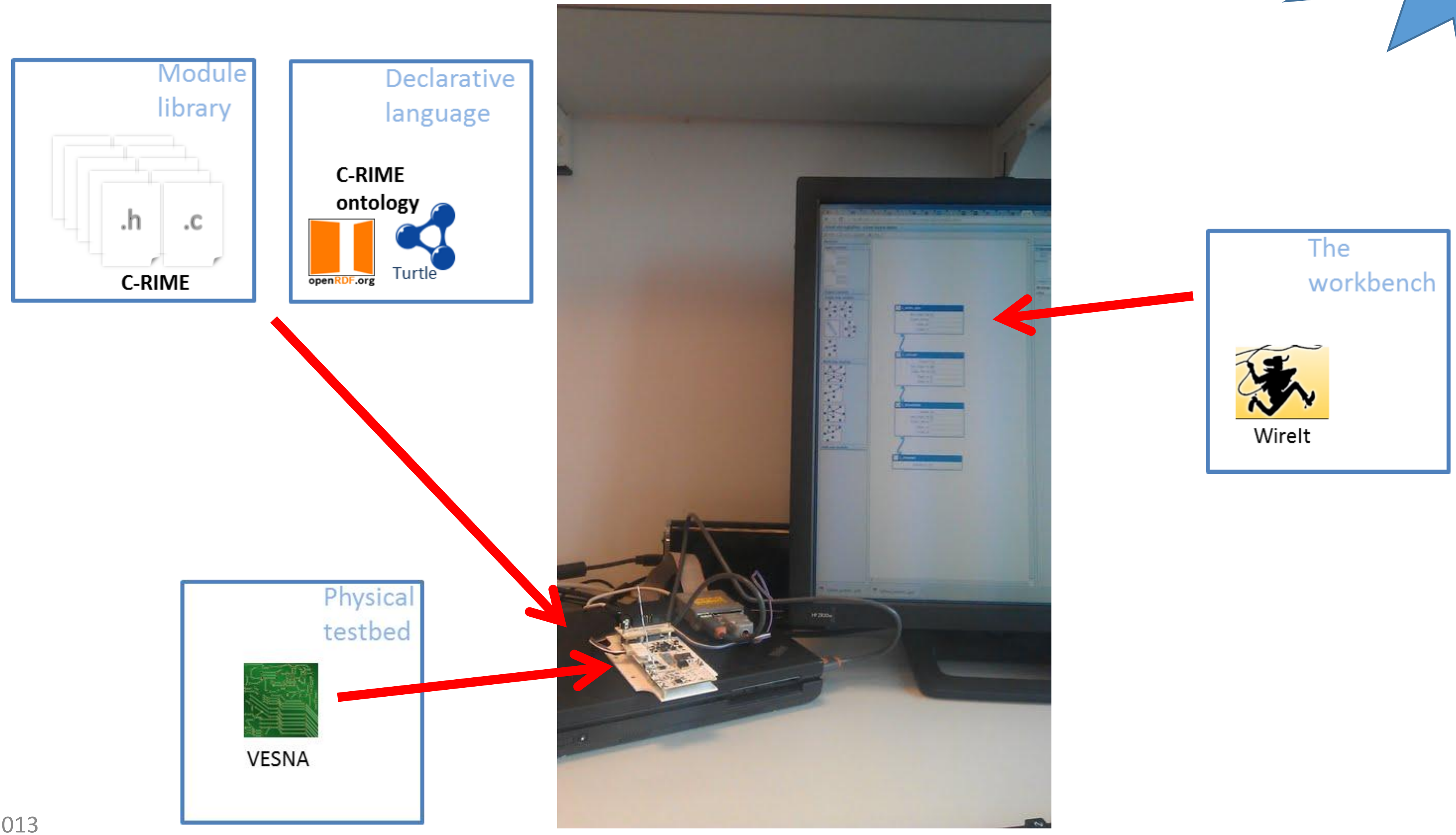
For valid stacks, the server auto-generates initialization code.

5

The server invokes the tools for generating the image and programming the VESNA.

Example of operational set-up

Broadcast demo



Module library

.h .c

C-RIME

Declarative language

C-RIME ontology

openRDF.org

Turtle

The workbench

WireIt

Physical testbed

VESNA

ProtoStack's features

- **Modularity**
  - the communication services have to have a modular design and implementation to allow composability of more complex services which can then achieve end to end communication.
- **Flexibility**
  - the components of the workbench should be designed and implemented in a way that allows interacting with the resulting tool at different levels of abstractions (e.g. at the module library level, at the workbench level). The components should also be easy to extend and upgrade.
- **Easy programming**
  - users with various levels of programming skills should find it easy to use the tools appropriate to their level of experience resulting from the implementation of the framework.
- **Reproducibility of experiments**
  - the framework should support re-running and reproducing experiments in an easy way for instance by saving and reloading an experiment description.
- **Remote experimentation**
  - remote users should be able to define and perform experiments and download the result. This can be most easily achieved through a web portal.

Comparison of ProtoStack with other tools

- The four component approach is generic enough and well suited for design and experimentation with dynamic composition of communication services.

- All tools address most of the requirements
- Only ProtoStack explicitly addresses the requirement related to remote experimentation

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

Name	Modularity	Flexibility	Easy programming	Reproducibility of experiments	Remote experimentation
x-Kernel	✓	✓	✓	✓	●
Click	✓	✓	✓	✓	●
SNA	✓	✓	✓	●	●
Proto Stack	✓	✓	✓	✓	✓

The Composeable Rime module library

Composeable Rime (CRime)

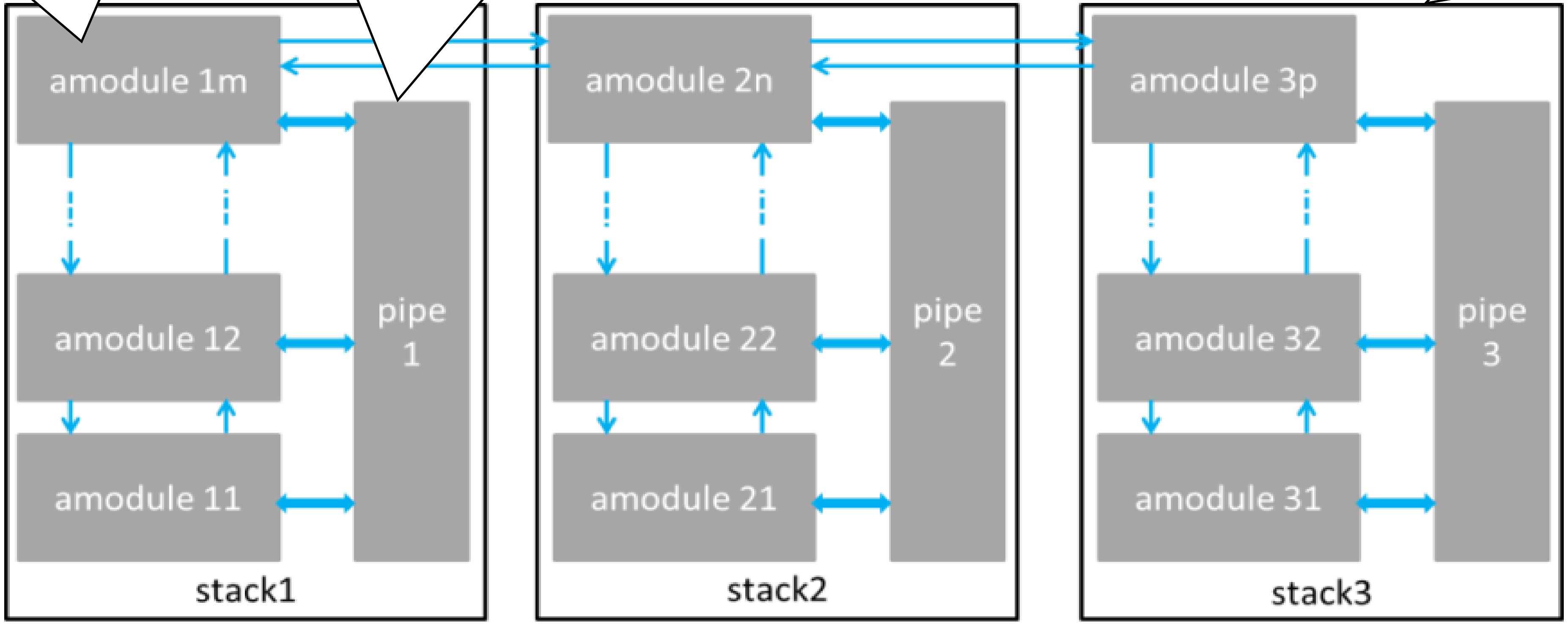
- Is a module library
  - Written in C
  - Inspired by and based on Contiki's Rime stack
  - Works on any platform that can run Contiki OS
- Each module implements a communication service (communication functionality)
- Modules can be composed to offer complex communication services

CRime abstractions

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

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.

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



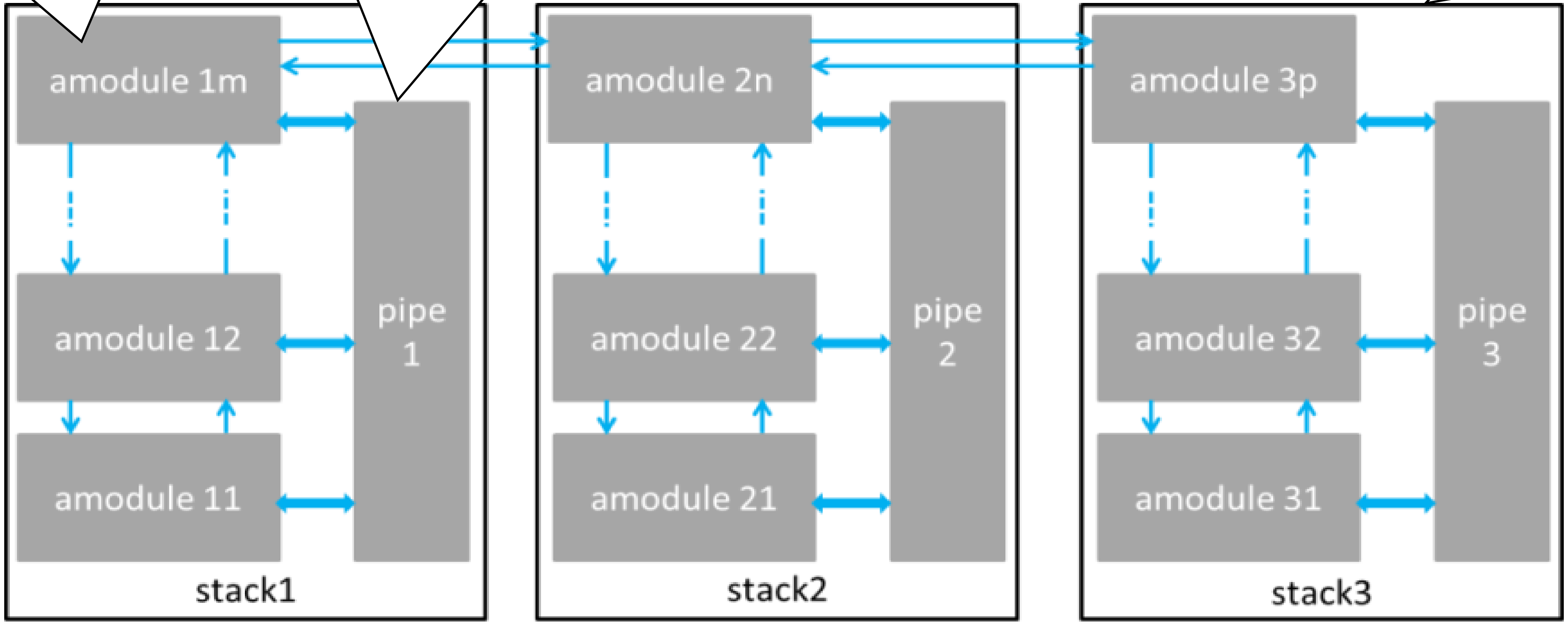
CRime abstractions

These abstractions are introduced by CRime and they cannot be found in Rime.

The abstract module is a generic building block of the CRime stack. Behind each instance of an amodule hides a communication service

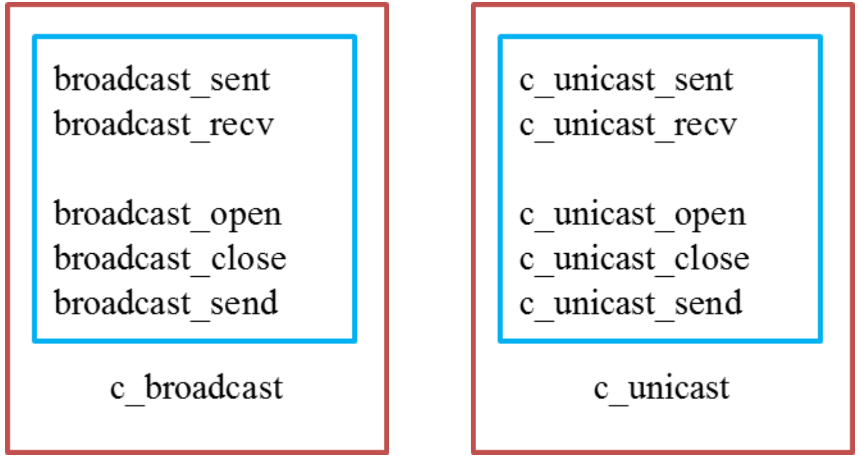
The pipe is a vertical structure which ca be accessed by any of the modules in a composed stack. The pipe contains only data structures corresponding to parameters that are used by the stack.

which tains a ce of amodules and a pipe. It haves 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.



The abstract module (amodule)

- The amodule is an abstraction of communication modules and behaves as a wrapper around those modules (red box)



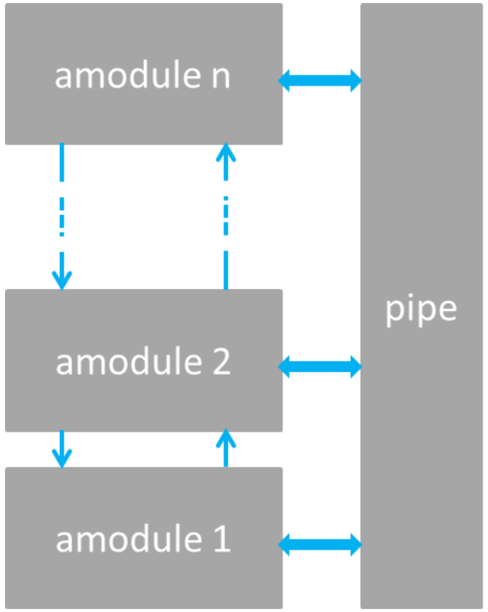
- It defines a set of generic functions (the interface)

- ***The interface of the abstract module.***

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

The pipe

- The pipe is a data structure corresponding to the concept of vertical layer from cross-layer design terminology.
- It stores cross layer information that is beyond the scope of chameleon's data structures.
- Facilitates modular protocol stack and cognitive networking experimentation.

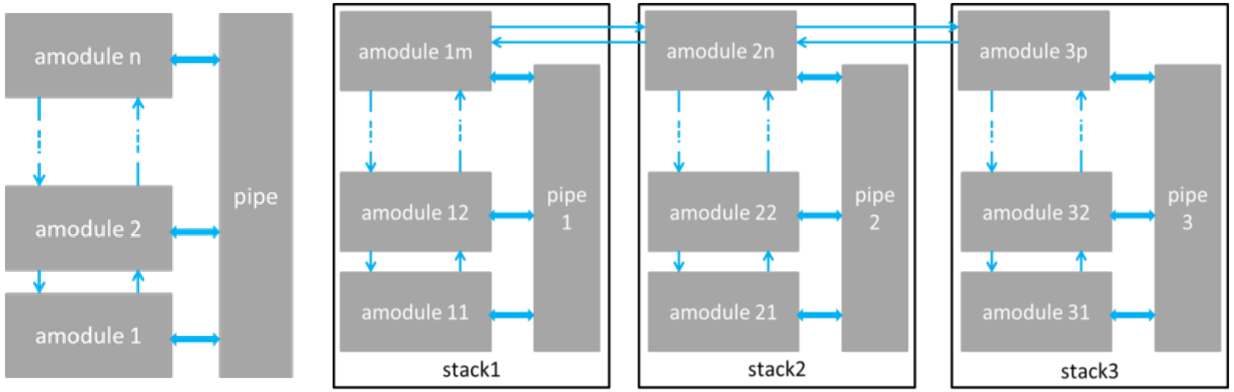


The pipe data structure.

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

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

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

The stack

- Is an abstraction that refers to a complex communication service.
- It consists of a set of amodules and one or more pipes.

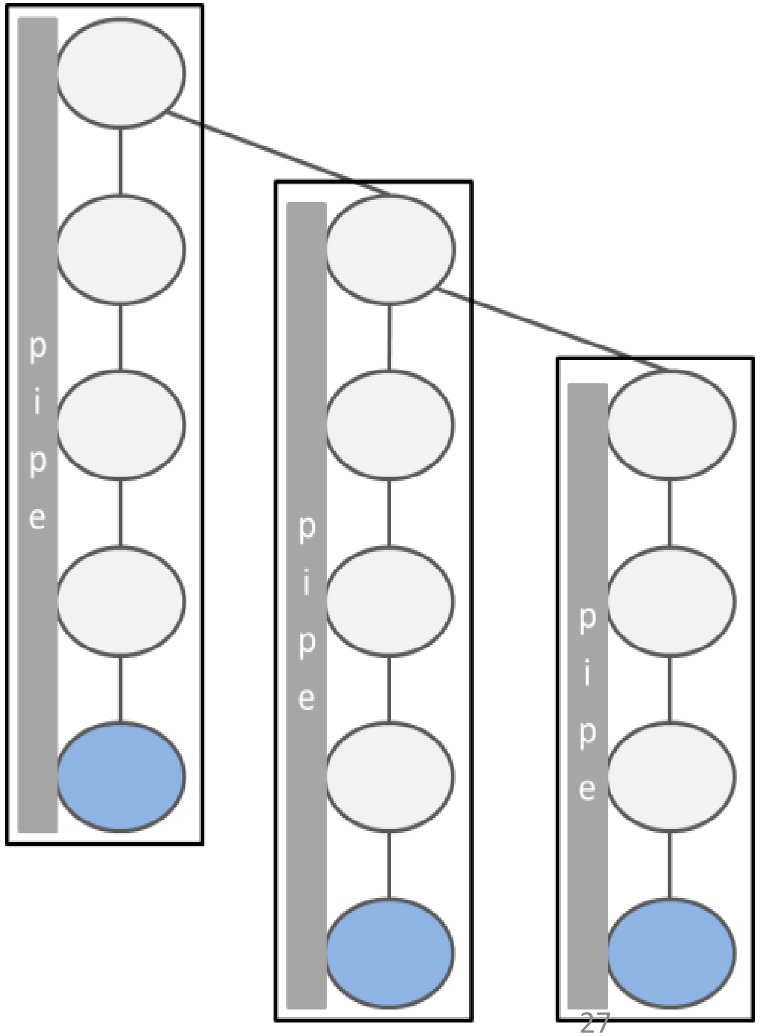


The stack_open function.

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

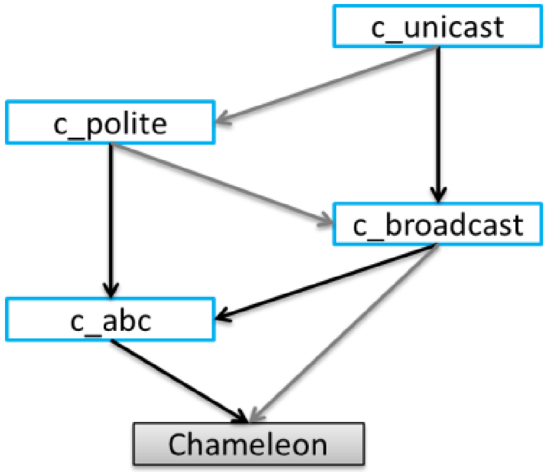
Theoretical modeling of the CRime communication stack

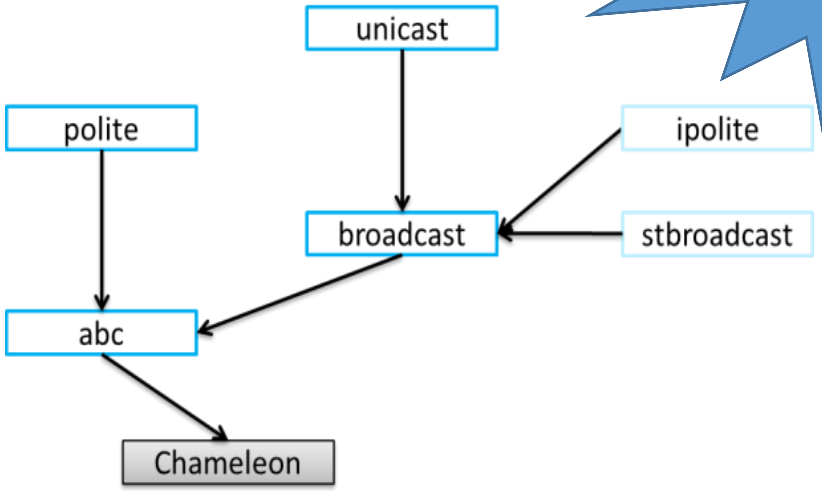
- The theoretical model behind the CRime communication stack is a tree
- Each node of the tree includes one or more abstract modules which are connected and communicate in a horizontal manner.
- Each leaf of the tree corresponds to a channel and the corresponding branch forms a stack and has a pipe attached.
- Recursion is used to walk through the tree.



Architectural comparison with Rime

Triggered broadcast demo

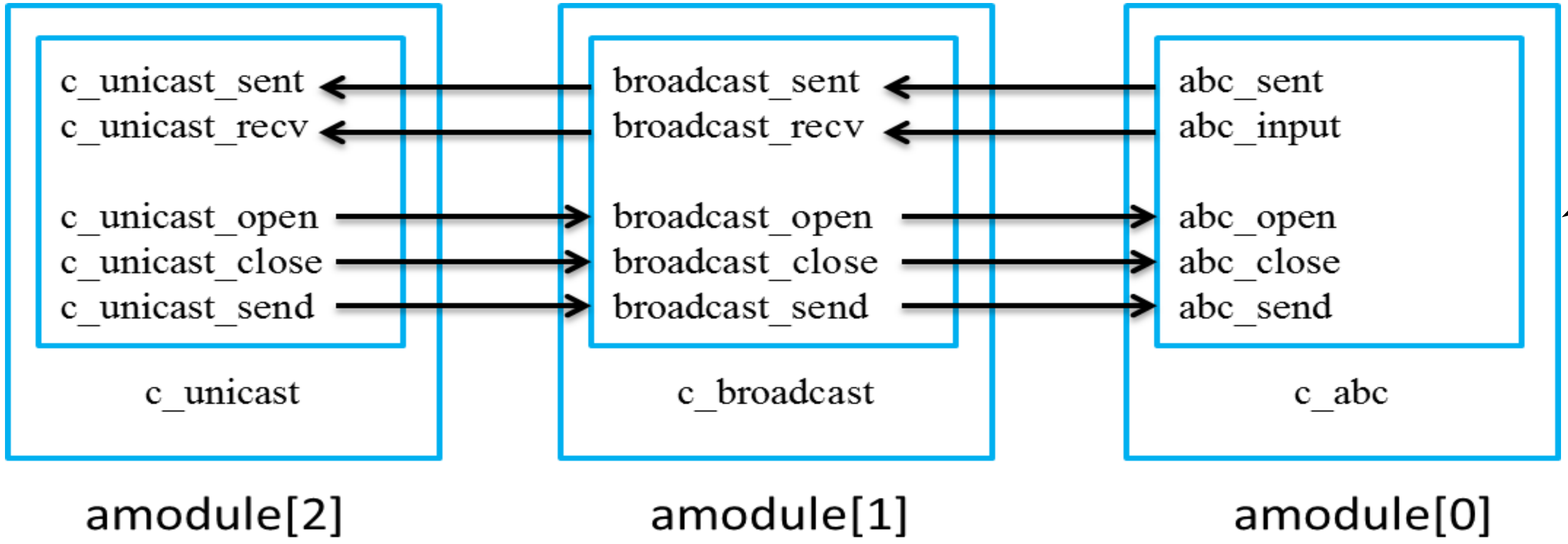

c_unicast
c_polite
c_broadcast
c_abc
Chameleon


unicast
polite
ipolite
broadcast
stbroadcast
abc
Chameleon

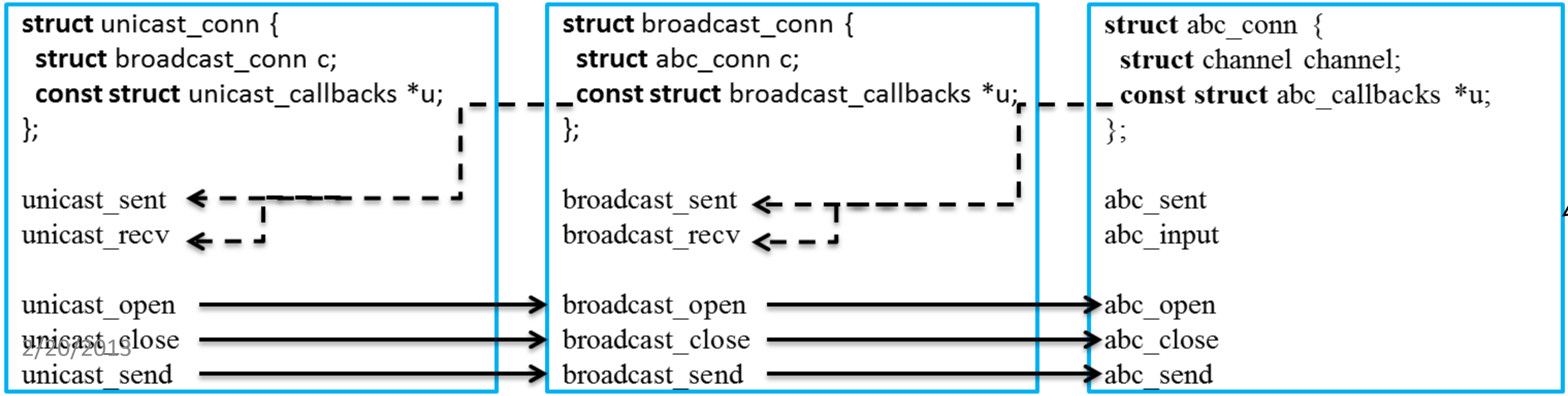
- The CRime dependency graph is composed by the user un a dynamic way
- Some paths through the CRime dependency graph may not form a valid stack
  - *functionality across modules needn't repeat*

- The Rime dependency graph is hard coded
- Any path through the Rime dependency graph forms a valid stack
  - *because of this, functionality can repeat across modules*

Architectural comparison with Rime



Flexible tree implementation example using CRime's amodule abstraction



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

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

```
struct abc_conn  {
  struct channel channel;
  const struct abc_callbacks  *u;
};
```

Hard coded tree implementation example in Rime

The cost of composeability

- Composeability introduces additional overhead (the implementation of the abstractions)
- the Rime and CRime components differ just in the size of the code with no clear advantage on one or the other side
    - the cost of the abstraction does not appear at this level yet
- the size of the code of the applications which use CRime stacks is about 16% larger (~13.000 bytes)
- execution time for the sequence of operations open→send→recv→close is ~256 ms in Rime and 622 ms in CRime (a factor of ~2.4 higher)
- CRime consumes 1.6 % more energy than Rime

The ProtoStack declarative language and the Workbench

The ProtoStack declarative language - requirements

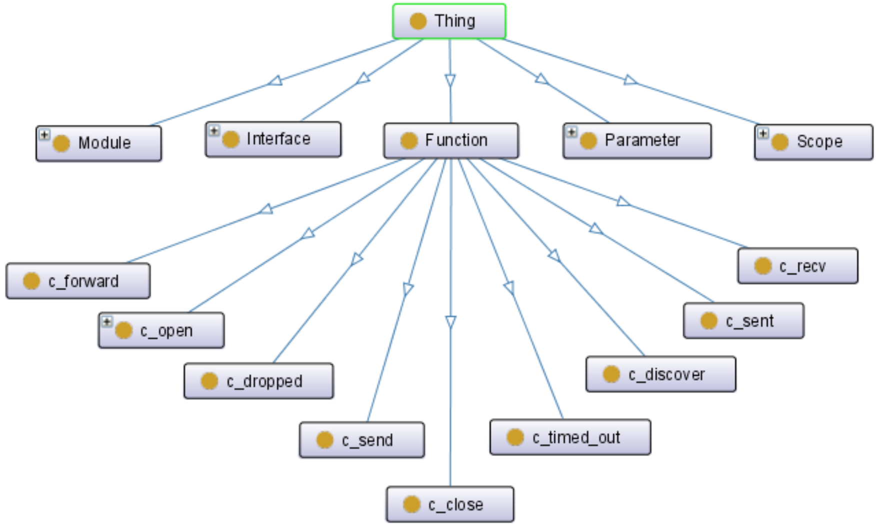
- **Simplicity**
  - as friendly as possible to the target user group
- **Machine readable**
  - to facilitate easy manipulation by machines
- **Standardized**
  - a relatively widely adopted, open and stable standardized approach is preferred to a less stable and potentially proprietary approach
- **Interoperability**
  - to facilitate the interoperability of systems so that potential reference implementations of the framework can be easily connected at this level of abstraction
- **Support for knowledge representation and logic reasoning**
  - should also support emerging logical reasoning for self-configuration of communication networks

The ProtoStack declarative language

- uses the RDF data model

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

- the custom vocabulary built by creating the CRime ontology



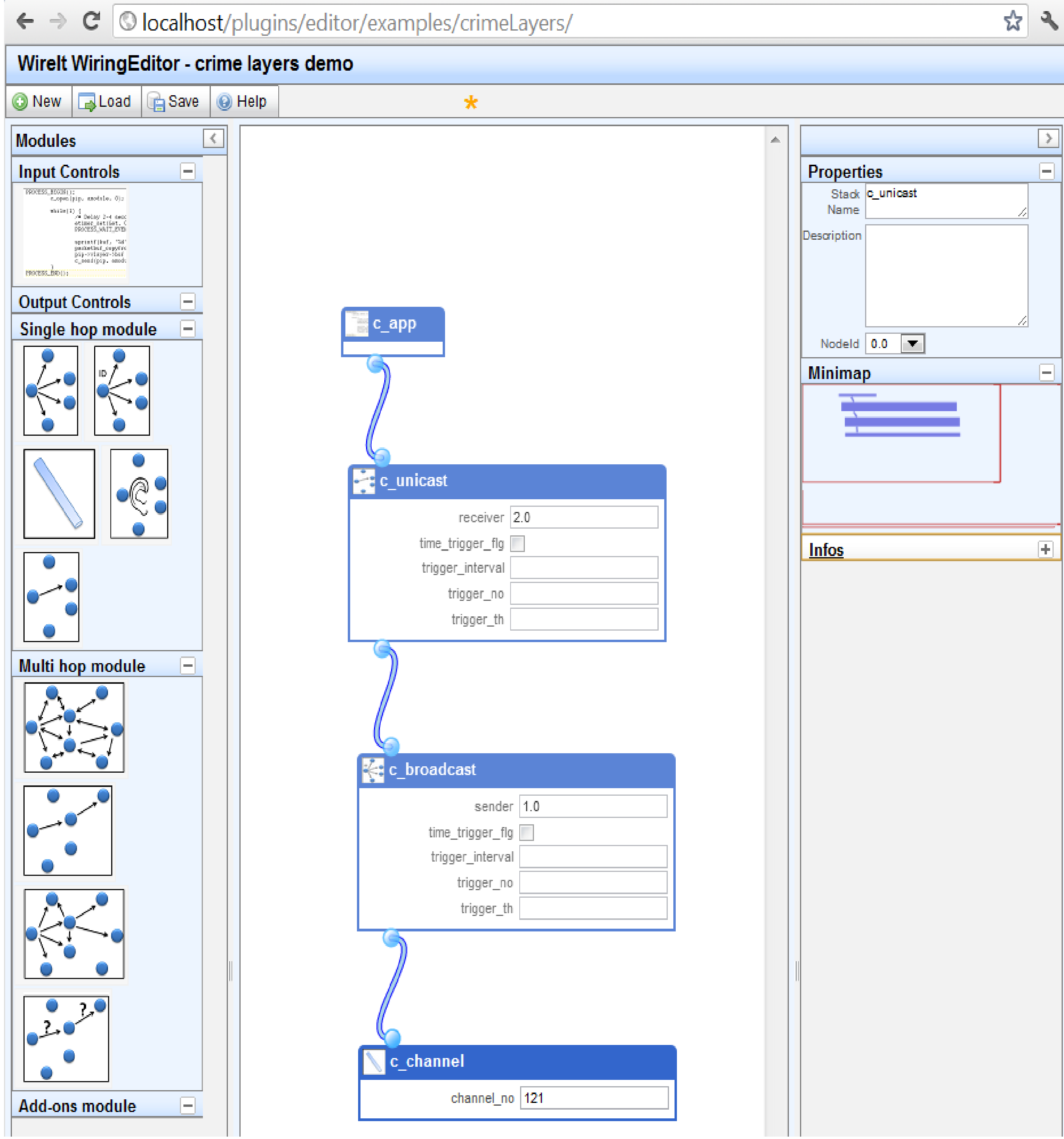
- the Turtle syntax which is human readable and can easily be transformed in XML if needed.

```
//turtle crime:c_abc rdf:type cpan:Module .
```

ProtoStack configuration steps

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

2/20/2013



c_unicast

receiver 2.0

time_trigger_flg ☐

trigger_interval

trigger_no

trigger_th

c_broadcast

sender 1.0

time_trigger_flg ☐

trigger_interval

trigger_no

trigger_th

c_channel

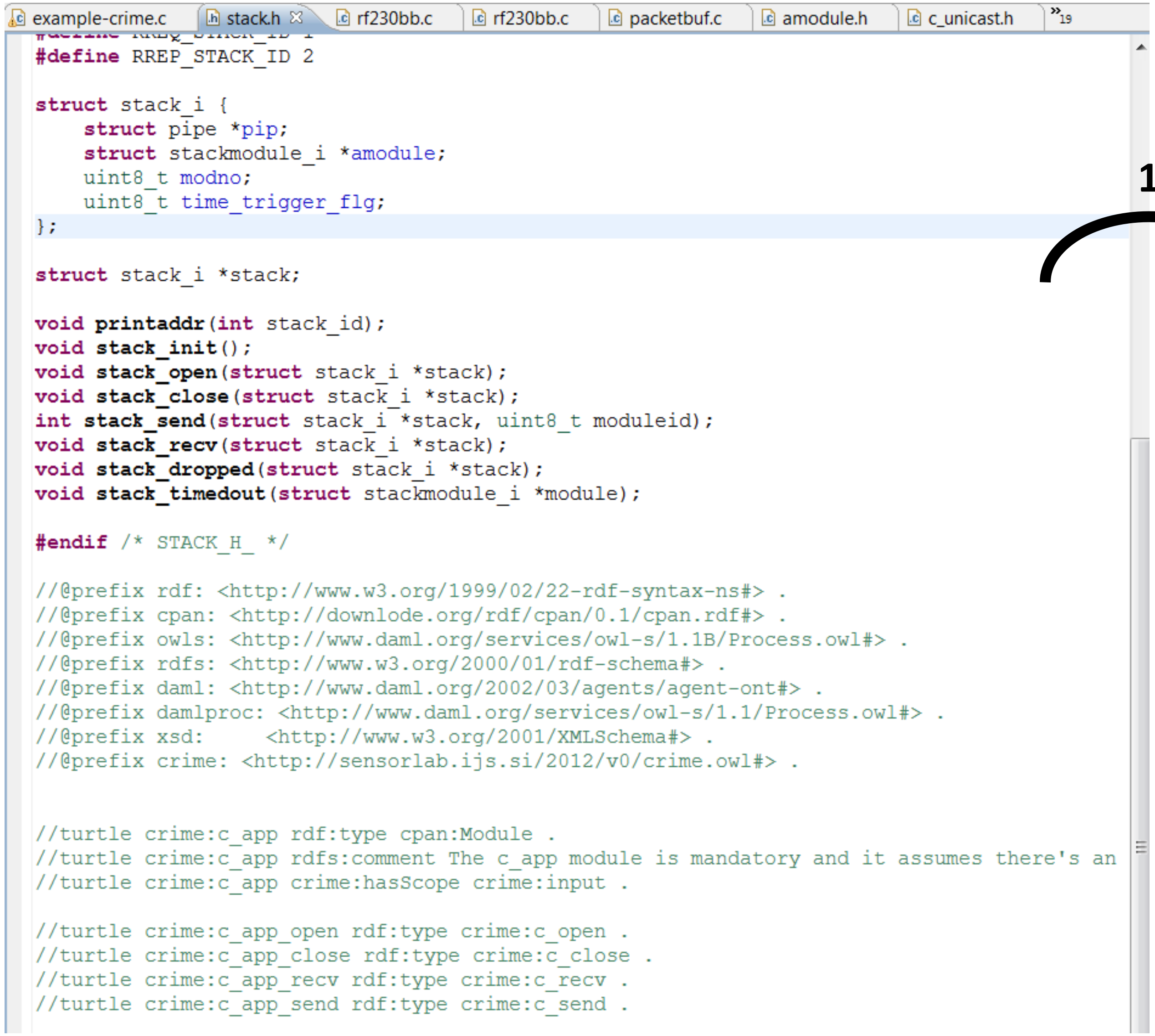
channel_no 121

Stack Name c_unicast

Description

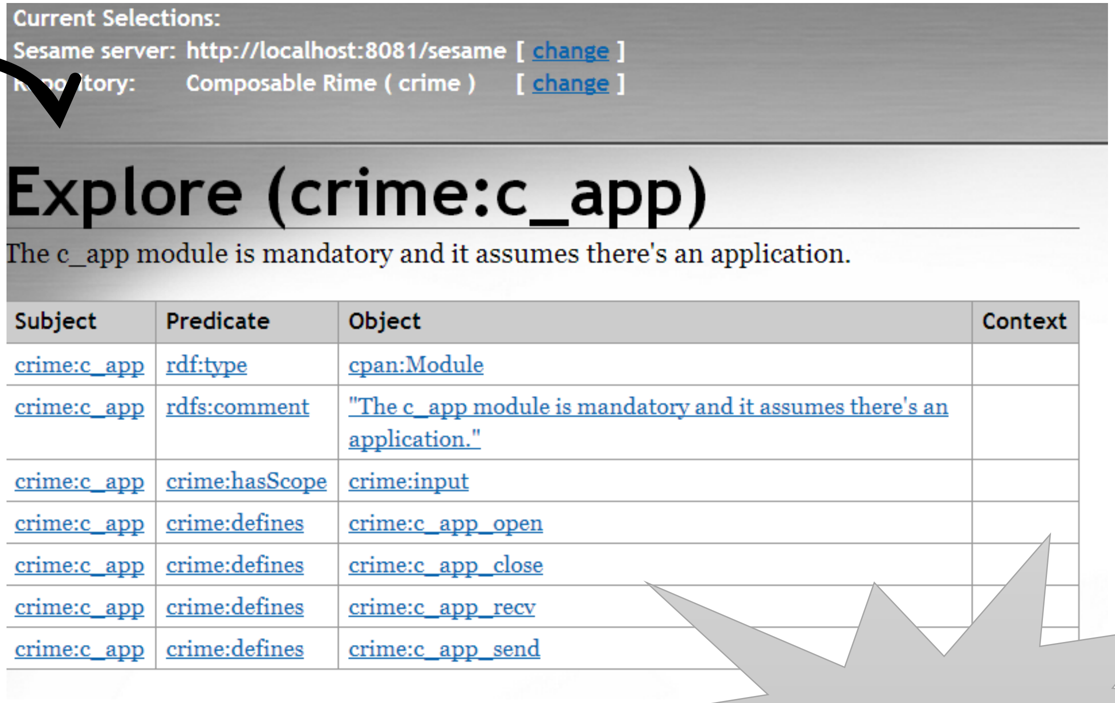
NodeId 0.0

Fact specification, translation and storage



```
#define RREP_STACK_ID 2

struct stack_i {
    struct pipe *pip;
    struct stackmodule_i *amodule;
    uint8_t modno;
    uint8_t time_trigger_flg;
};

struct stack_i *stack;

void printaddr(int stack_id);
void stack_init();
void stack_open(struct stack_i *stack);
void stack_close(struct stack_i *stack);
int stack_send(struct stack_i *stack, uint8_t moduleid);
void stack_recv(struct stack_i *stack);
void stack_dropped(struct stack_i *stack);
void stack_timedout(struct stackmodule_i *module);

#endif /* STACK_H_ */

//@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
//@prefix cpan: <http://downlode.org/rdf/cpan/0.1/cpan.rdf#> .
//@prefix owls: <http://www.daml.org/services/owl-s/1.1B/Process.owl#> .
//@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
//@prefix daml: <http://www.daml.org/2002/03/agents/agent-ont#> .
//@prefix damlproc: <http://www.daml.org/services/owl-s/1.1/Process.owl#> .
//@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .
//@prefix crime: <http://sensorlab.ijs.si/2012/v0/crime.owl#> .


//turtle crime:c_app rdf:type cpan:Module .
//turtle crime:c_app rdfs:comment The c_app module is mandatory and it assumes there's an
//turtle crime:c_app crime:hasScope crime:input .

//turtle crime:c_app_open rdf:type crime:c_open .
//turtle crime:c_app_close rdf:type crime:c_close .
//turtle crime:c_app_recv rdf:type crime:c_recv .
//turtle crime:c_app_send rdf:type crime:c_send .
```

1



Current Selections:
Sesame server: http://localhost:8081/sesame [change]
Repository: Composable Rime (crime) [change]

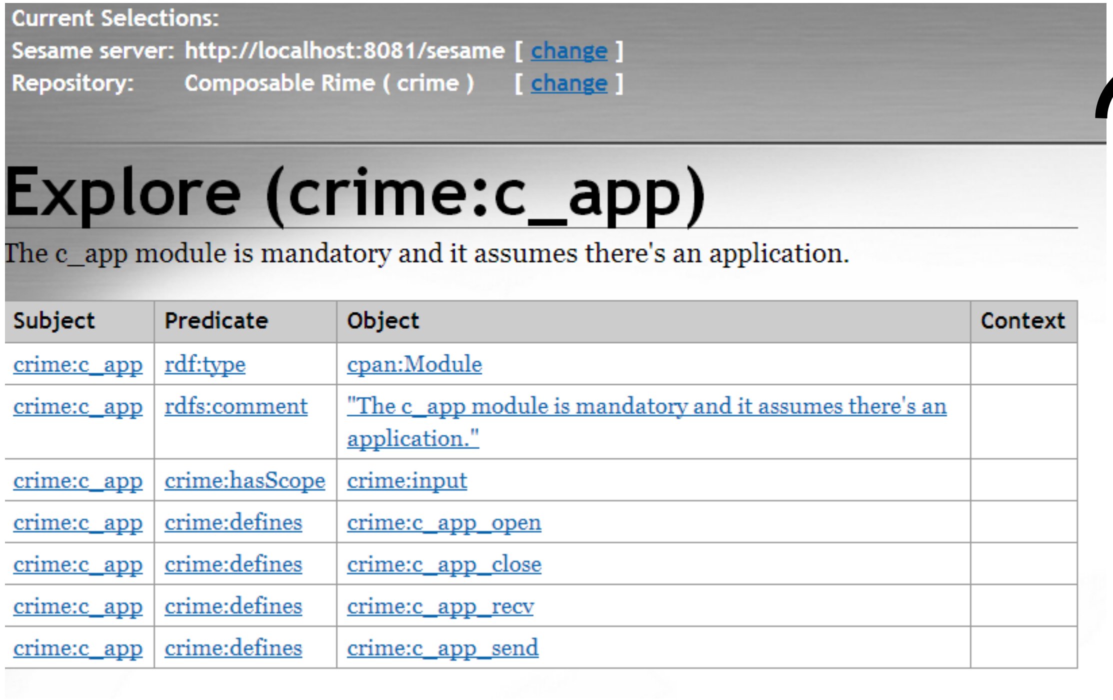
Explore (crime:c_app)

The c_app module is mandatory and it assumes there's an application.

Subject	Predicate	Object	Context
crime:c_app	rdf:type	cpan:Module	
crime:c_app	rdfs:comment	"The c_app module is mandatory and it assumes there's an application."	
crime:c_app	crime:hasScope	crime:input	
crime:c_app	crime:defines	crime:c_app_open	
crime:c_app	crime:defines	crime:c_app_close	
crime:c_app	crime:defines	crime:c_app_recv	
crime:c_app	crime:defines	crime:c_app_send	

c_app KB

Workbench rendering



Current Selections:

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

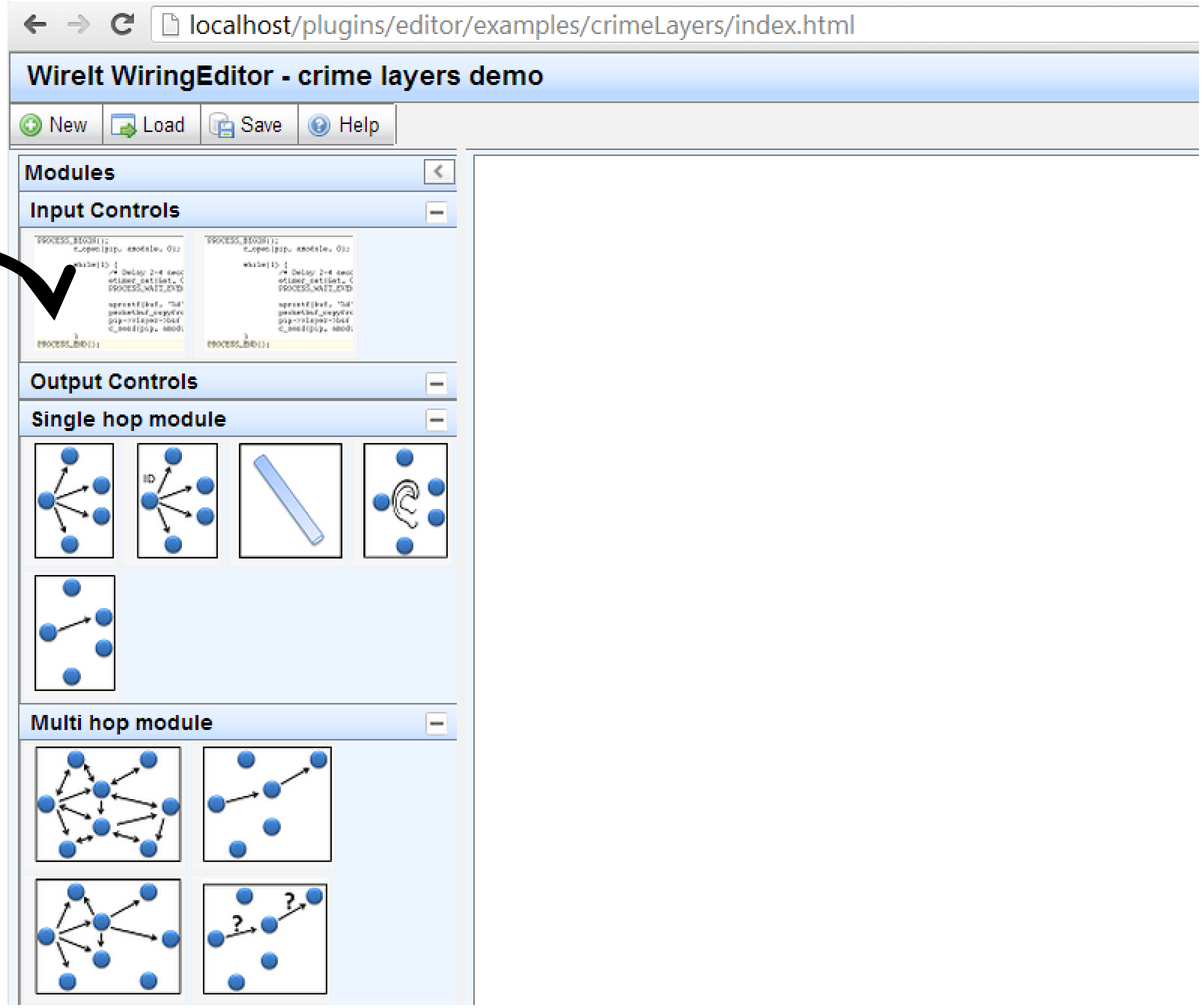
Repository: Composable Rime (crime) [change]

Explore (crime:c_app)

The c_app module is mandatory and it assumes there's an application.

Subject	Predicate	Object	Context
crime:c_app	rdf:type	cpan:Module	
crime:c_app	rdfs:comment	"The c_app module is mandatory and it assumes there's an application."	
crime:c_app	crime:hasScope	crime:input	
crime:c_app	crime:defines	crime:c_app_open	
crime:c_app	crime:defines	crime:c_app_close	
crime:c_app	crime:defines	crime:c_app_recv	
crime:c_app	crime:defines	crime:c_app_send	

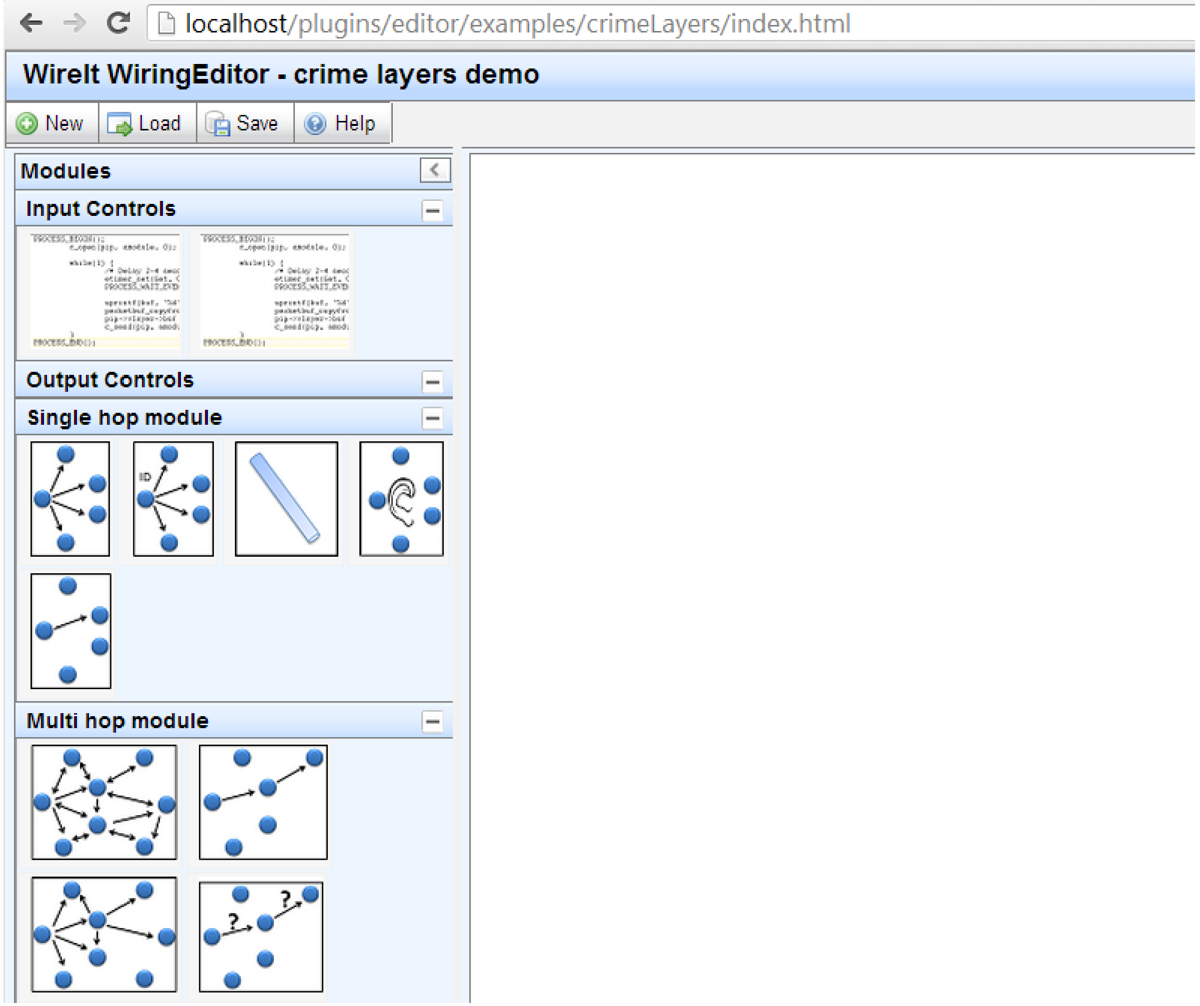
2

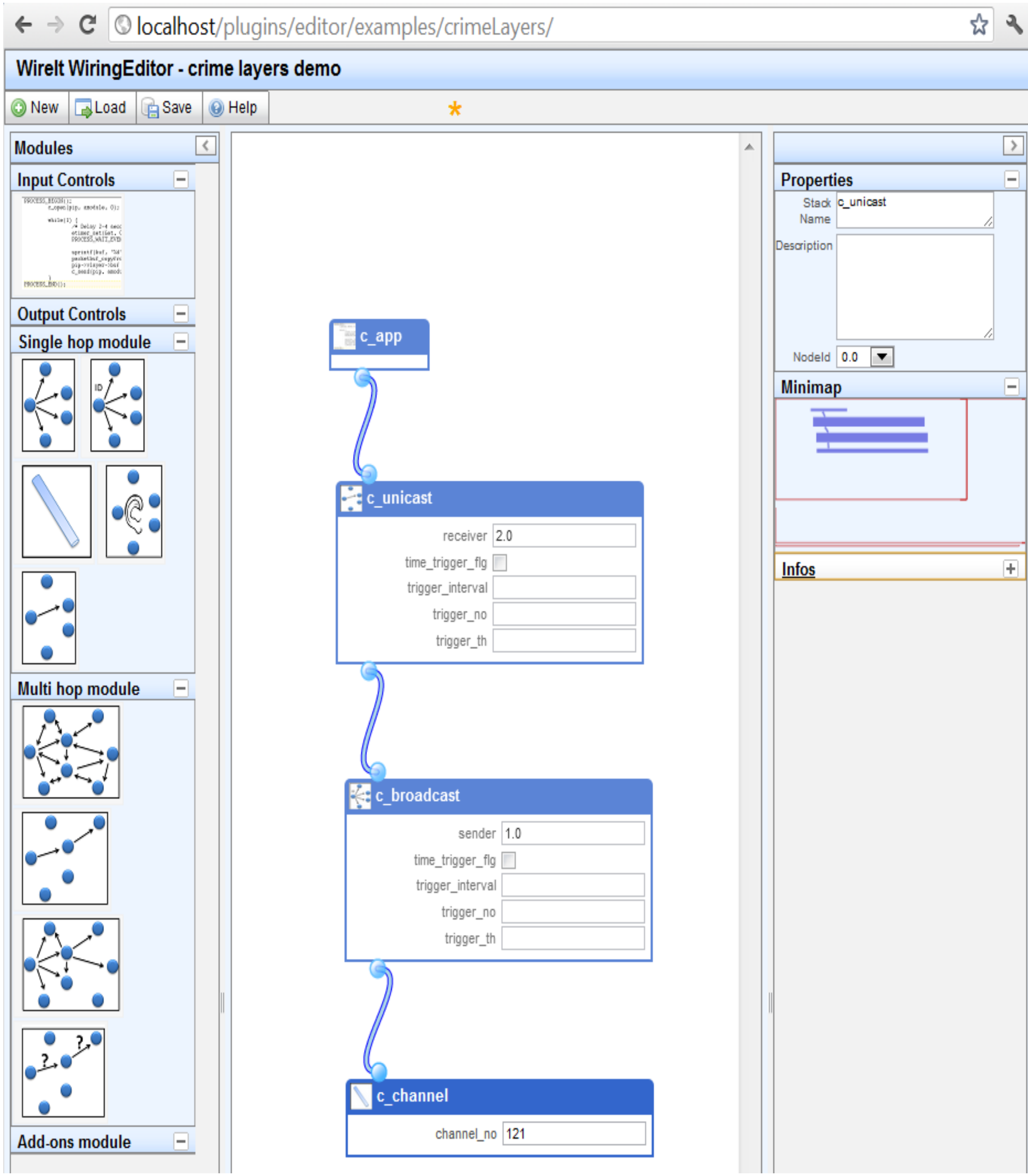


localhost/plugins/editor/examples/crimeLayers/index.html

WireIt WiringEditor - crime layers demo

Manual stack composition





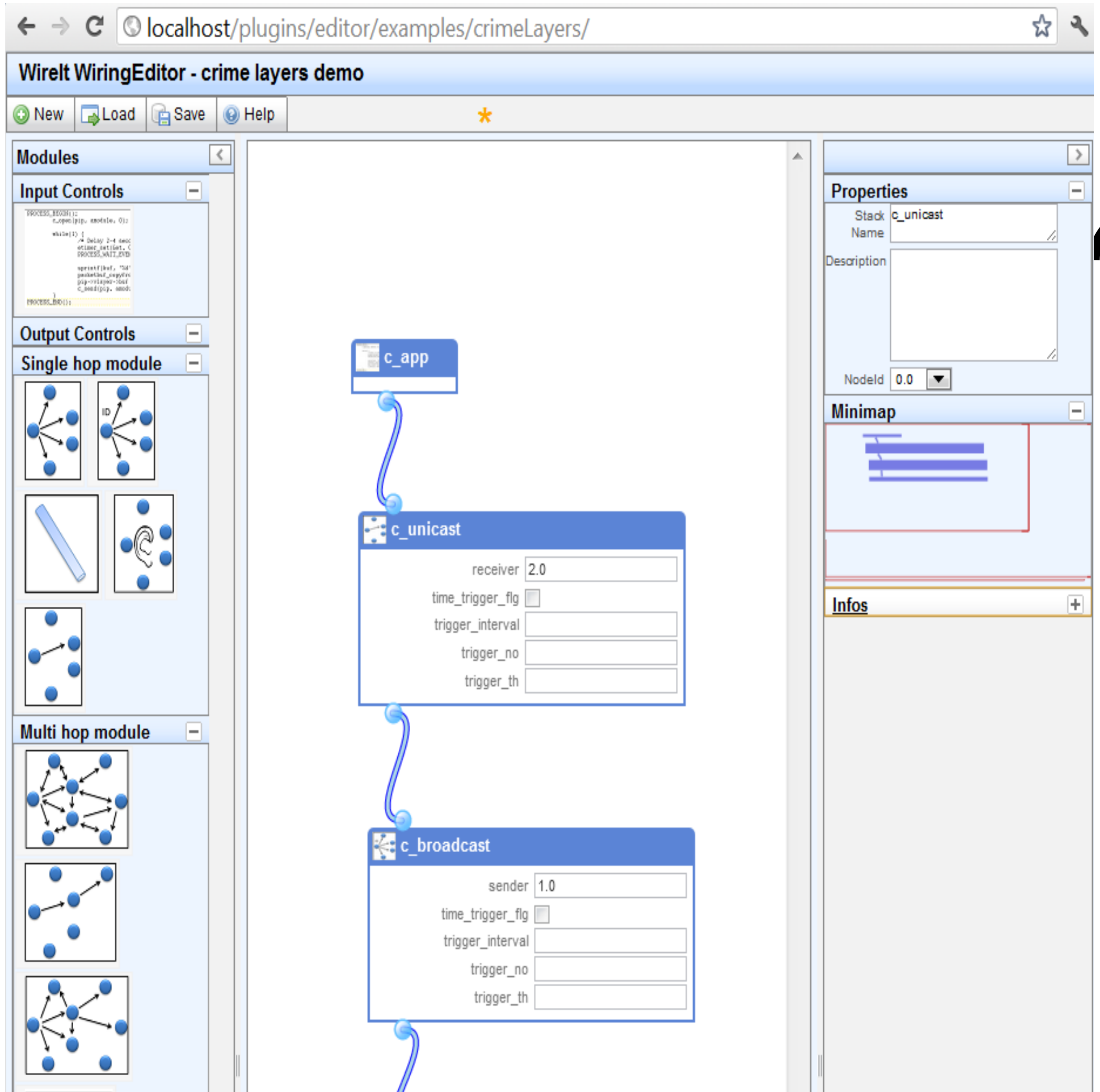
Stack Name: c_unicast
Description:
NodeId: 0.0

c_unicast
receiver: 2.0
time_trigger_flg: ☐
trigger_interval:
trigger_no:
trigger_th:

c_broadcast
sender: 1.0
time_trigger_flg: ☐
trigger_interval:
trigger_no:
trigger_th:

c_channel
channel_no: 121

Validity checking



Stack Name: c_unicast

Description:

NodeId: 0.0

c_unicast

receiver: 2.0

time_trigger_flg: ☐

trigger_interval:

trigger_no:

trigger_th:

c_broadcast

sender: 1.0

time_trigger_flg: ☐

trigger_interval:

trigger_no:

trigger_th:

3

Current Selections:

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

Repository: Composable Rime (crime) [change]

Explore (crime:c_app)

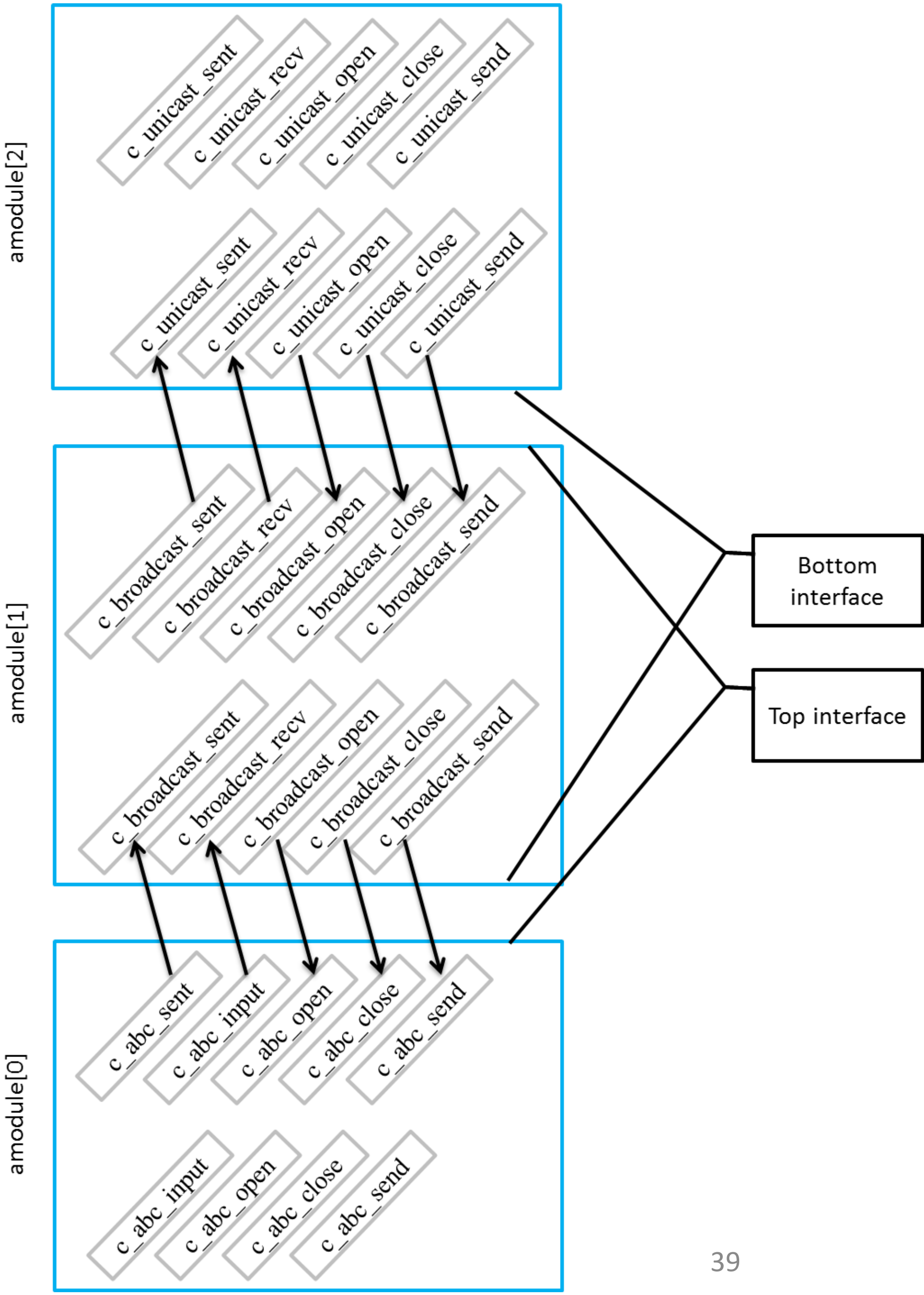
The c_app module is mandatory and it assumes there's an application.

Subject	Predicate	Object	Context
crime:c_app	rdf:type	cpan:Module	
crime:c_app	rdfs:comment	"The c_app module is mandatory and it assumes there's an application."	
crime:c_app	crime:hasScope	crime:input	
crime:c_app	crime:defines	crime:c_app_open	
crime:c_app	crime:defines	crime:c_app_close	
crime:c_app	crime:defines	crime:c_app_recv	
crime:c_app	crime:defines	crime:c_app_send	

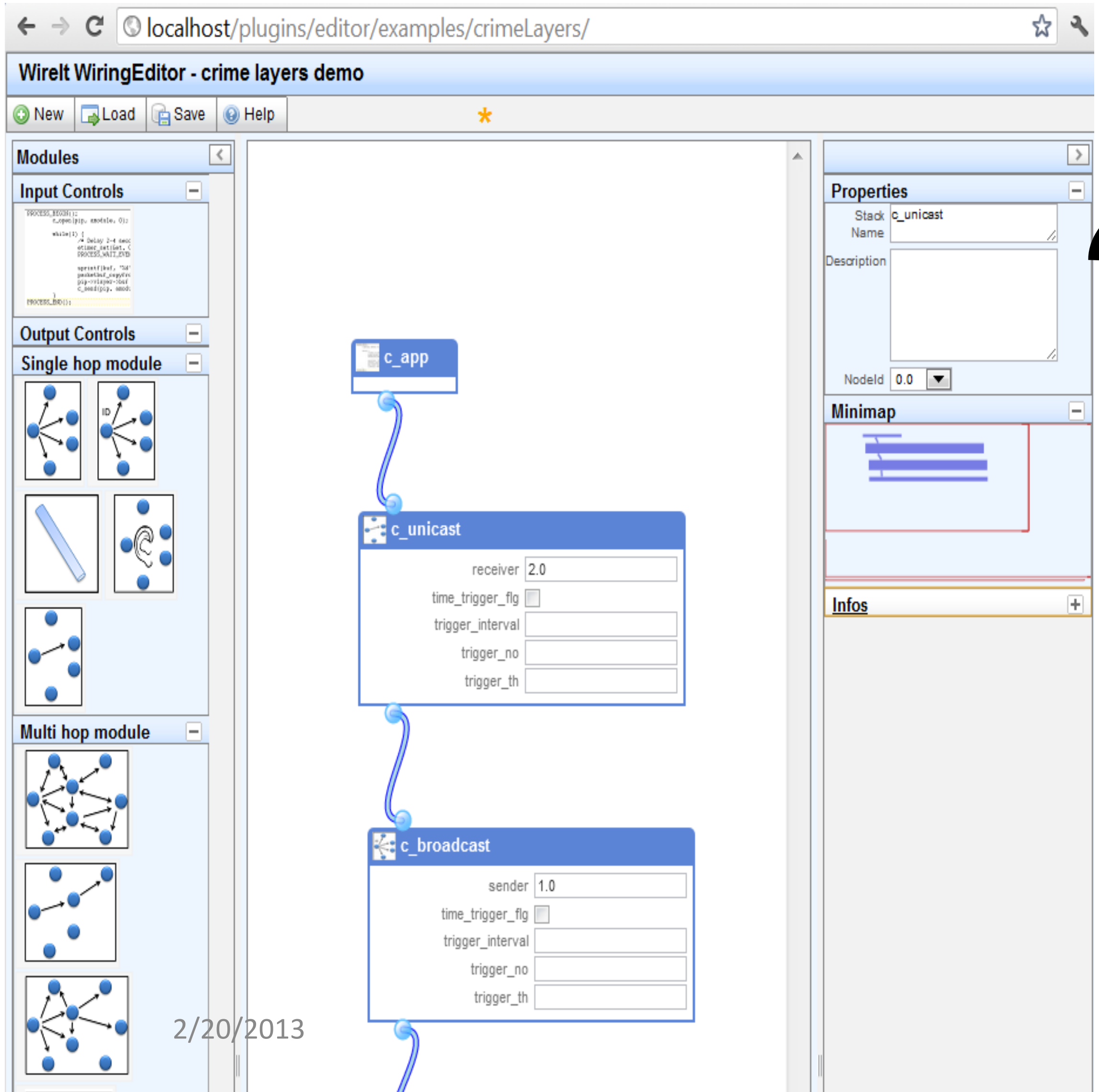
Rules for validity checking

- Rules are currently hard coded in the server
- For experimentation using Service Oriented Networks, a reasoner supporting rules has to be integrated in the system

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



Code generation

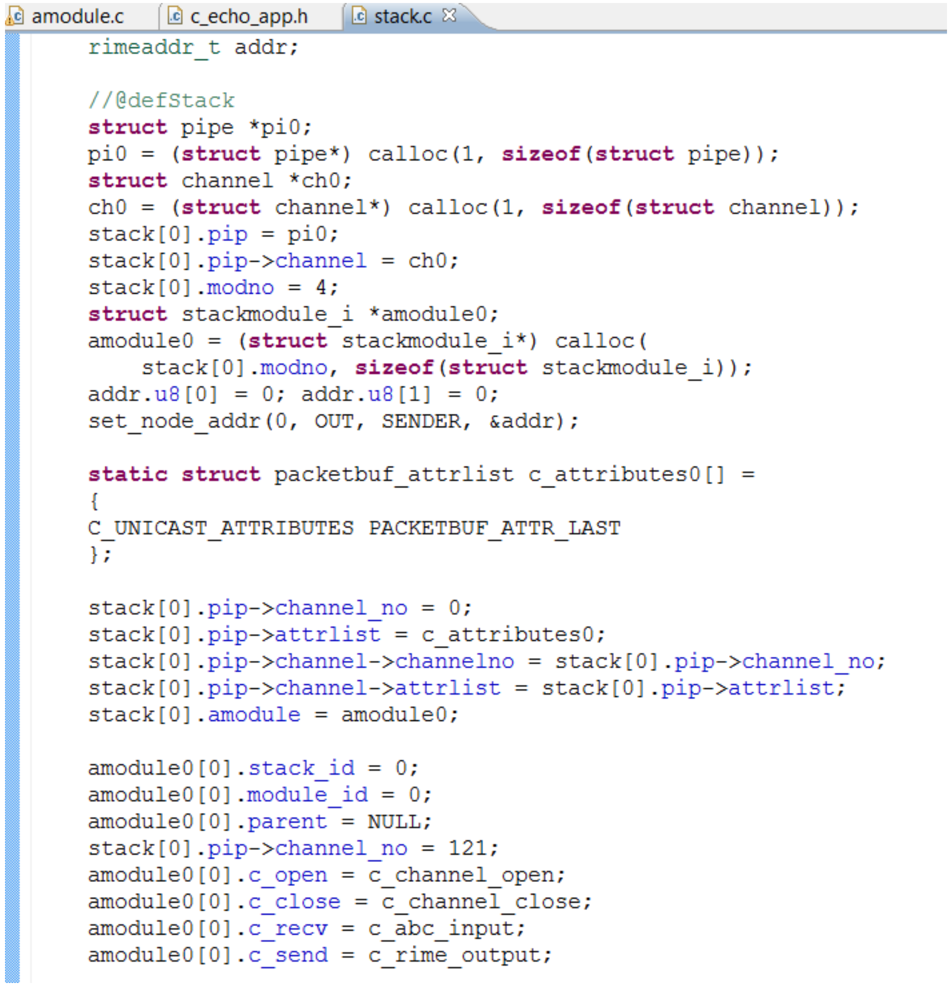


Stack Name: c_unicast
Description:
NodeId: 0.0

c_unicast
receiver: 2.0
time_trigger_flg: ☐
trigger_interval:
trigger_no:
trigger_th:

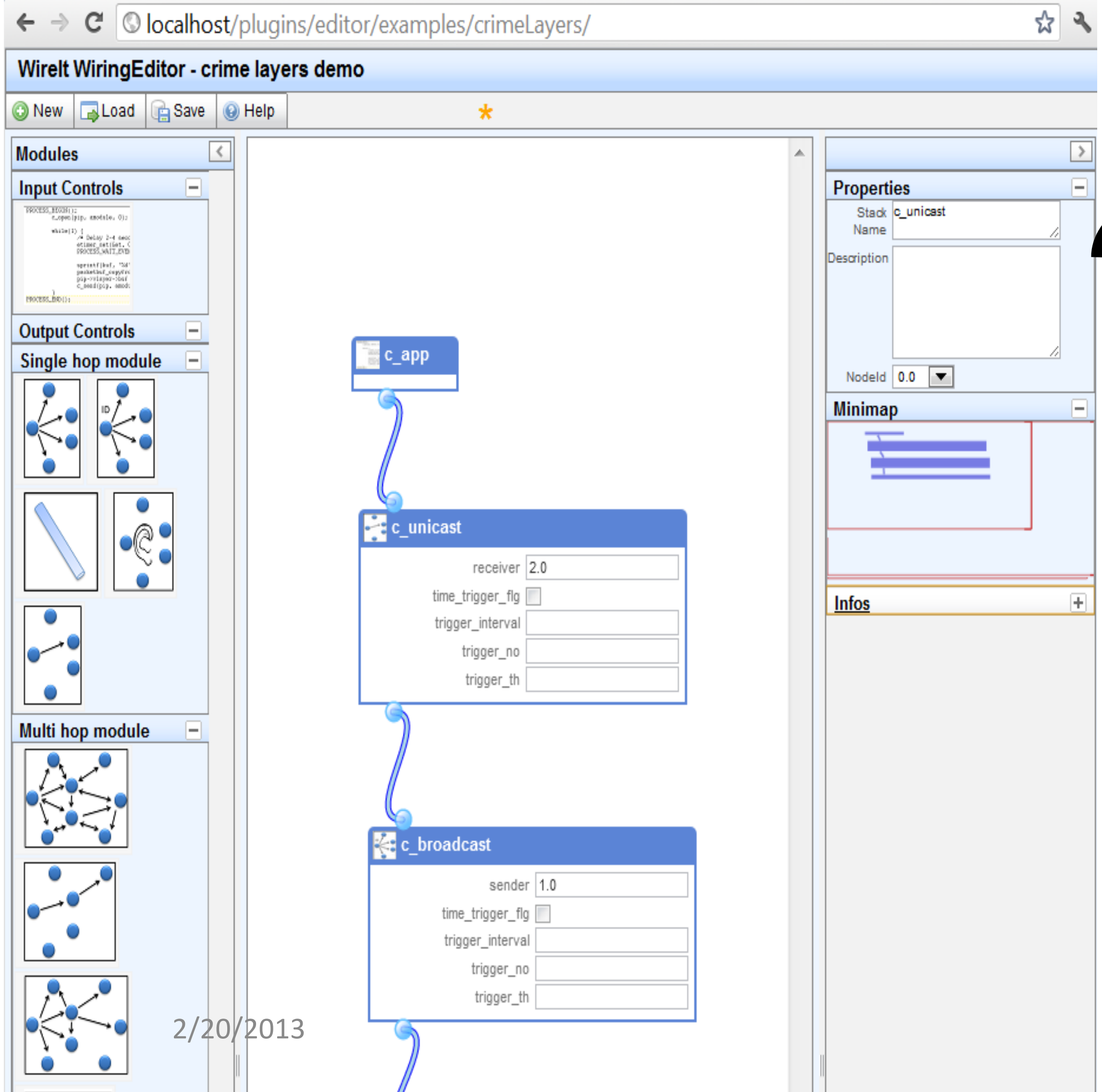
c_broadcast
sender: 1.0
time_trigger_flg: ☐
trigger_interval:
trigger_no:
trigger_th:

4



```c
    rimeaddr_t addr;

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

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

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

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

Reprogram the testbed



Stack Name: c_unicast

Description:

NodeId: 0.0

c_unicast

receiver: 2.0

time_trigger_flg: ☐

trigger_interval:

trigger_no:

trigger_th:

c_broadcast

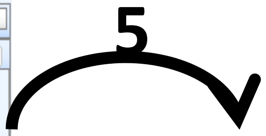
sender: 1.0

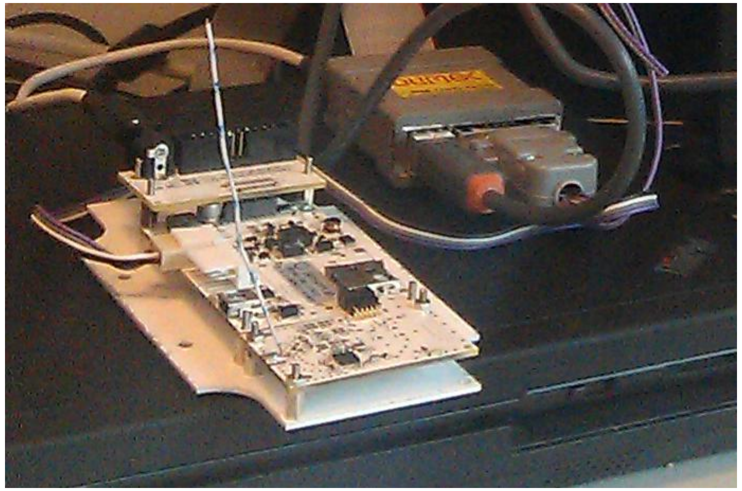
time_trigger_flg: ☐

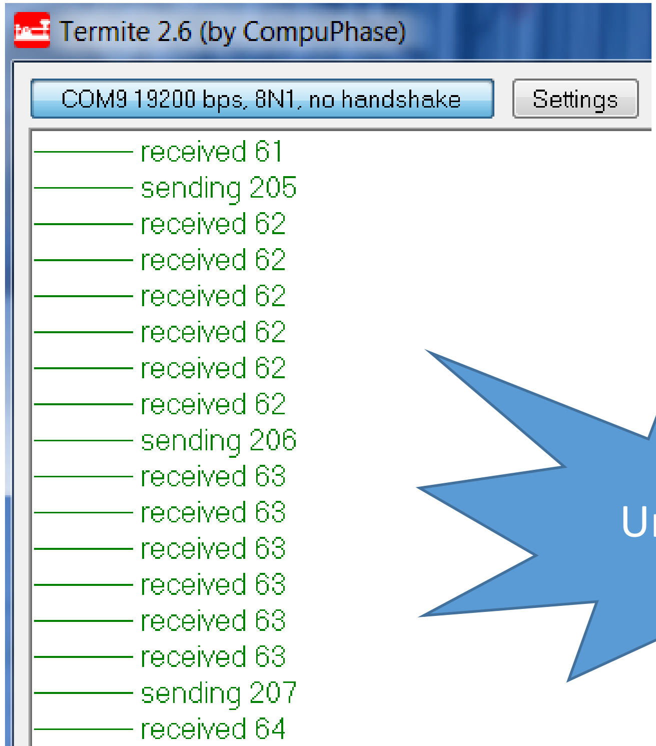
trigger_interval:

trigger_no:

trigger_th:







```
received 61
sending 205
received 62
received 62
received 62
received 62
received 62
received 62
sending 206
received 63
received 63
received 63
received 63
received 63
received 63
sending 207
received 64
```

Unicast demo

2/20/2013

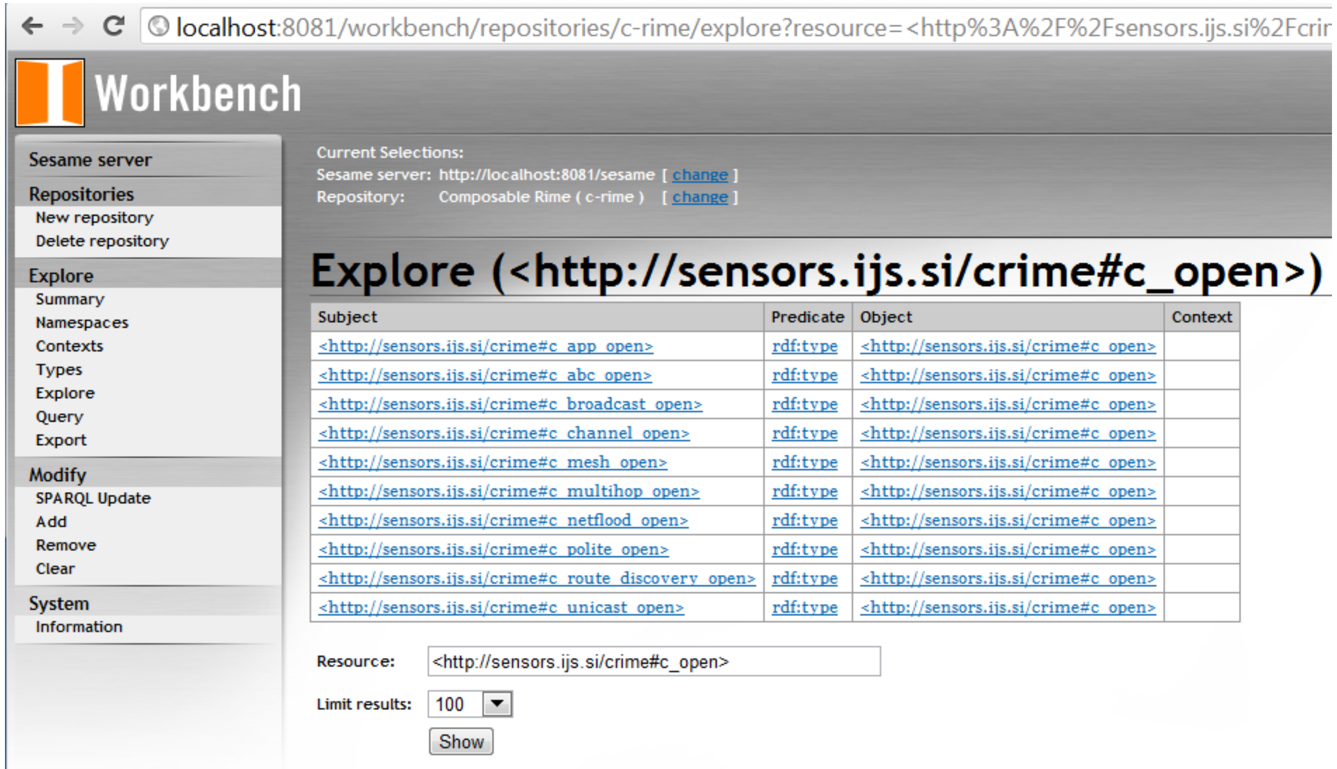
Service Oriented Networks with ProtoStack

Service Oriented Networks

- a network which makes use of Service Oriented Architecture principles to provide end-to-end transport services
  - Services are described, published and can be discovered
  - Queried related to aspects of services are supported
  - Services can be composed

Describing and publishing services with ProtoStack

- Description is done in the .h files using RDF language, the CRime ontology and other relevant vocabularies
- Publishing is done using the Sesame tool (triple store + web server)



localhost:8081/workbench/repositories/c-rime/explore?resource=<http%3A%2F%2Fsensors.ijs.si%2Fcrir

Current Selections:
Sesame server: http://localhost:8081/sesame [change]
Repository: Composable Rime (c-rime) [change]

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

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

Resource: <http://sensors.ijs.si/crime#c_open>

Limit results: 100

Querying for published ProtoStack service

- Can be manually done through the knowledge base’s workbench
- Can be done by using SPARQL tools.
- Can be automatic between ProtoStack instances, typically used for synchronization

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

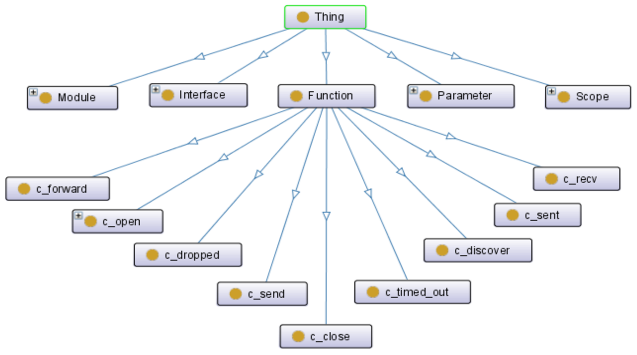
Composition of services using ProtoStack

- Composition of services for information transport using ProtoStack
  - **Manual composition of services**
    - where all the necessary reasoning is performed by the human.
  - **Semi-automatic composition of services**
    - where the human is guided in the decision making process by machine reasoning.
  - **Automatic composition of services**
    - where all the reasoning is performed by machines.

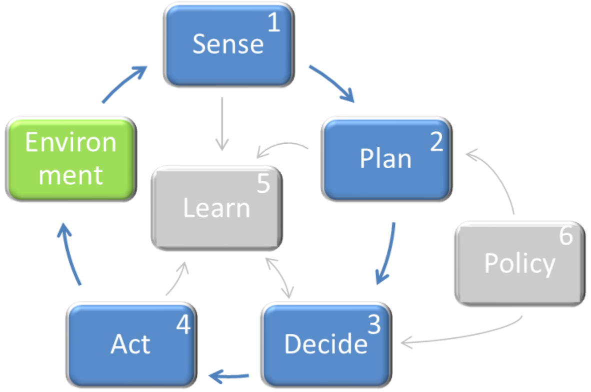
Cognitive networks with ProtoStack

Cognitive Networks

- Networks augmented by a knowledge plane that contains two key elements
  - A representation of relevant knowledge about the scope (device, homogenous network, heterogeneous network, etc.)

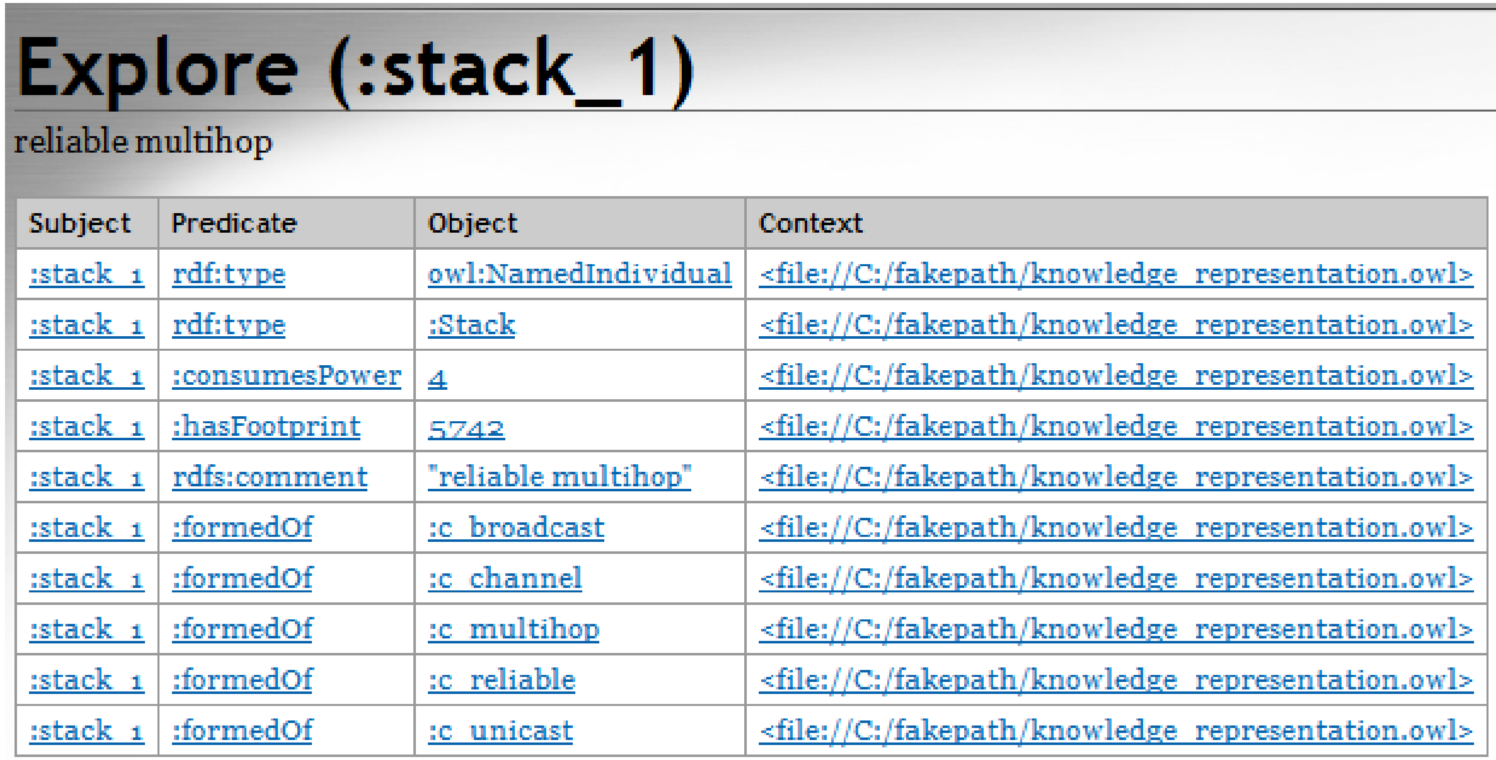


  - A cognition loop which uses AI techniques inside its states (machine learning techniques, decision making techniques, etc.)



Knowledge representation in ProtoStack

```
PREFIX :<http://sensorlab.ijs.si/2012/v0/crime.owl#>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX Process:<http://www.daml.org/services/owl-s/1.1B/Process.owl#>
PREFIX owl:<http://www.w3.org/2002/07/owl#>
PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX cpan:<http://downlode.org/rdf/cpan/0.1/cpan.rdf#>

SELECT ?stack WHERE {
   ?stack rdf:type :Stack .
   ?stack :formedOf :c_reliable .
   ?stack :consumesPower ?power .
}
ORDER BY ?power
LIMIT 1
```



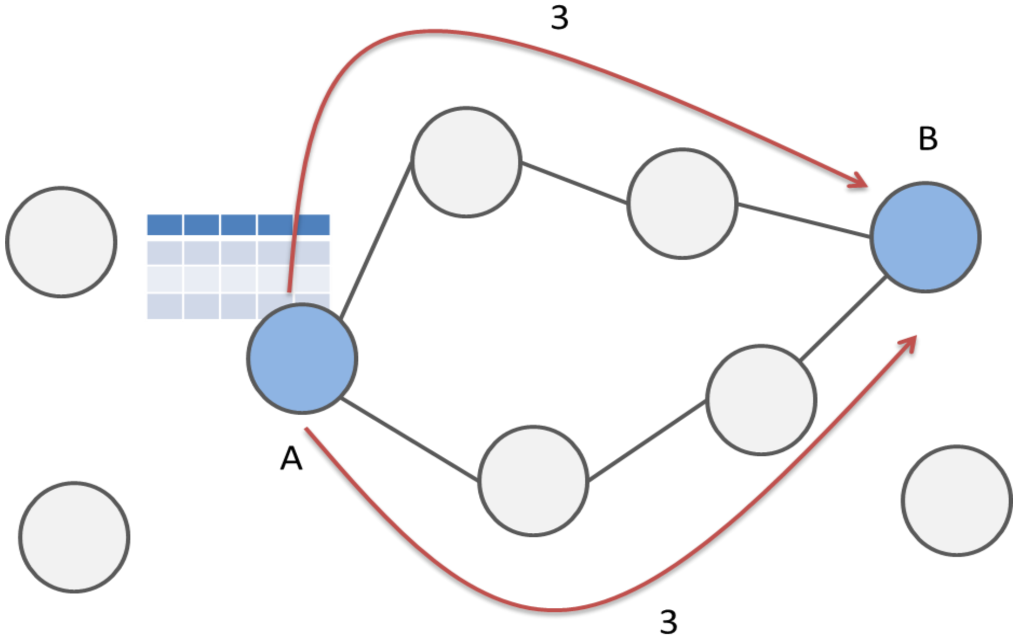
Explore (:stack_1)

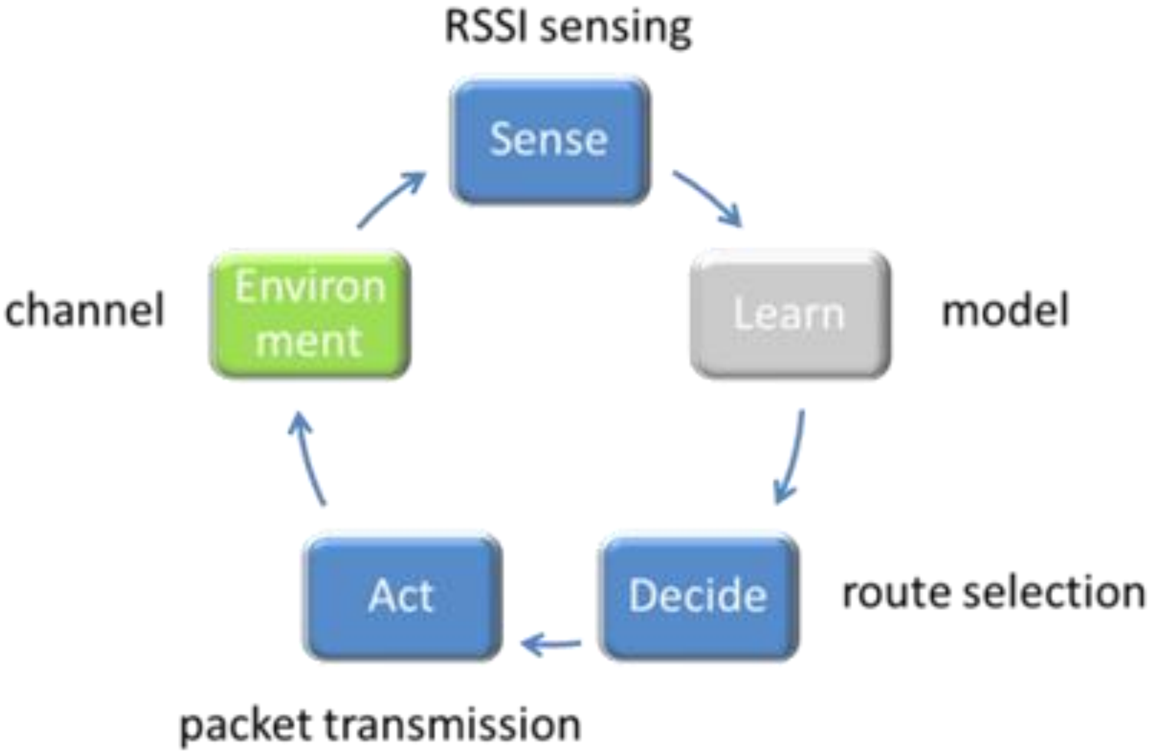
reliable multihop

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

Network layer cognitive loop with cross-layer information

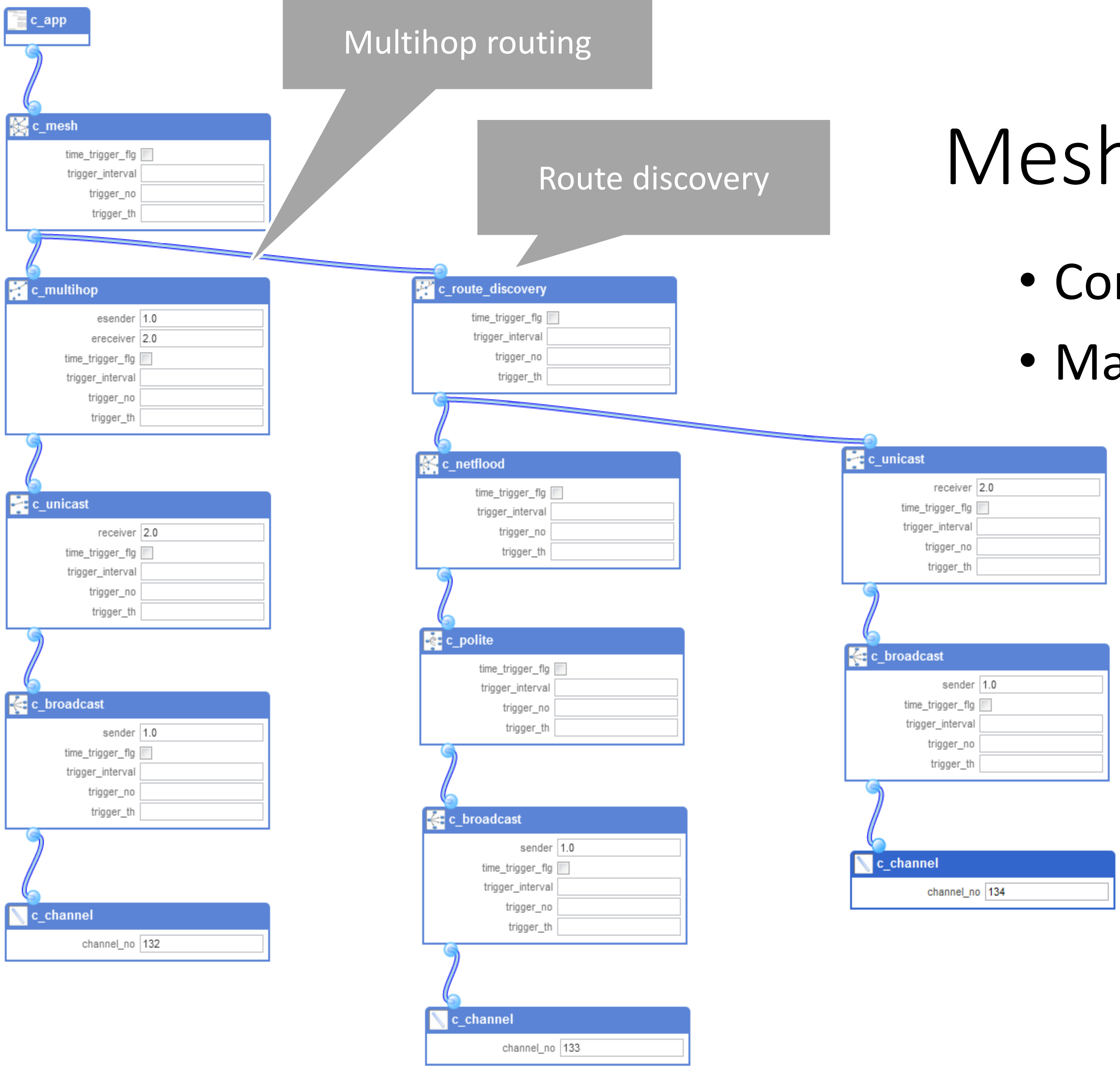
- Path cost in terms of hop no – could be too simplistic at times
- Additional parameter to be considered is RSSI





Mesh stack with CRime

- Communication service
- Management service



Multihop routing

Route discovery

c_app

c_mesh
time_trigger_flg ☐
trigger_interval
trigger_no
trigger_th

c_multihop
esender 1.0
ereceiver 2.0
time_trigger_flg ☐
trigger_interval
trigger_no
trigger_th

c_unicast
receiver 2.0
time_trigger_flg ☐
trigger_interval
trigger_no
trigger_th

c_broadcast
sender 1.0
time_trigger_flg ☐
trigger_interval
trigger_no
trigger_th

c_channel
channel_no 132

c_route_discovery
time_trigger_flg ☐
trigger_interval
trigger_no
trigger_th

c_netflood
time_trigger_flg ☐
trigger_interval
trigger_no
trigger_th

c_polite
time_trigger_flg ☐
trigger_interval
trigger_no
trigger_th

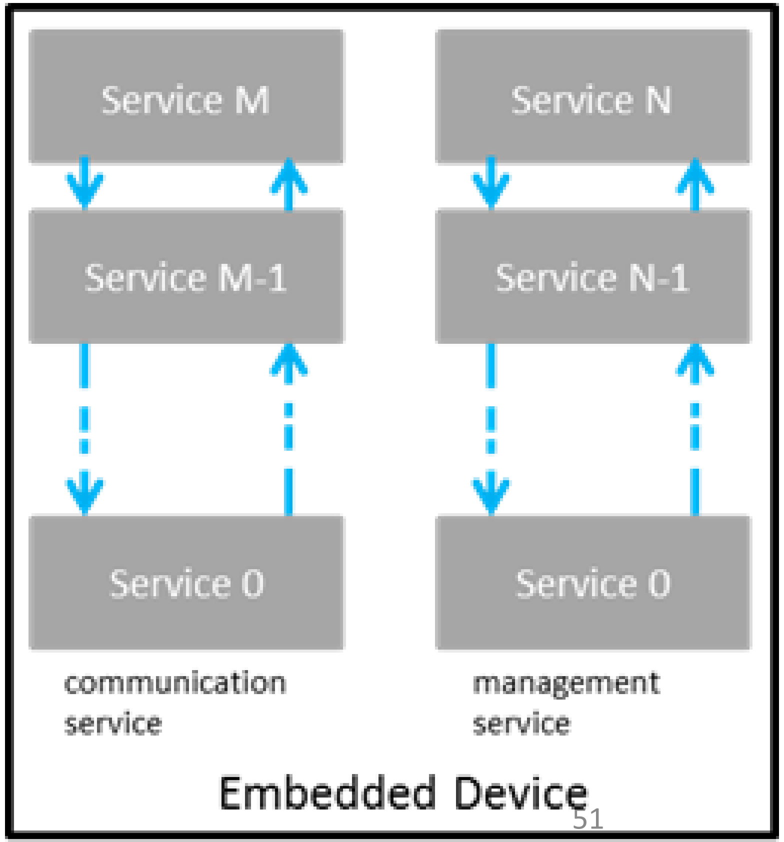
c_broadcast
sender 1.0
time_trigger_flg ☐
trigger_interval
trigger_no
trigger_th

c_channel
channel_no 133

c_unicast
receiver 2.0
time_trigger_flg ☐
trigger_interval
trigger_no
trigger_th

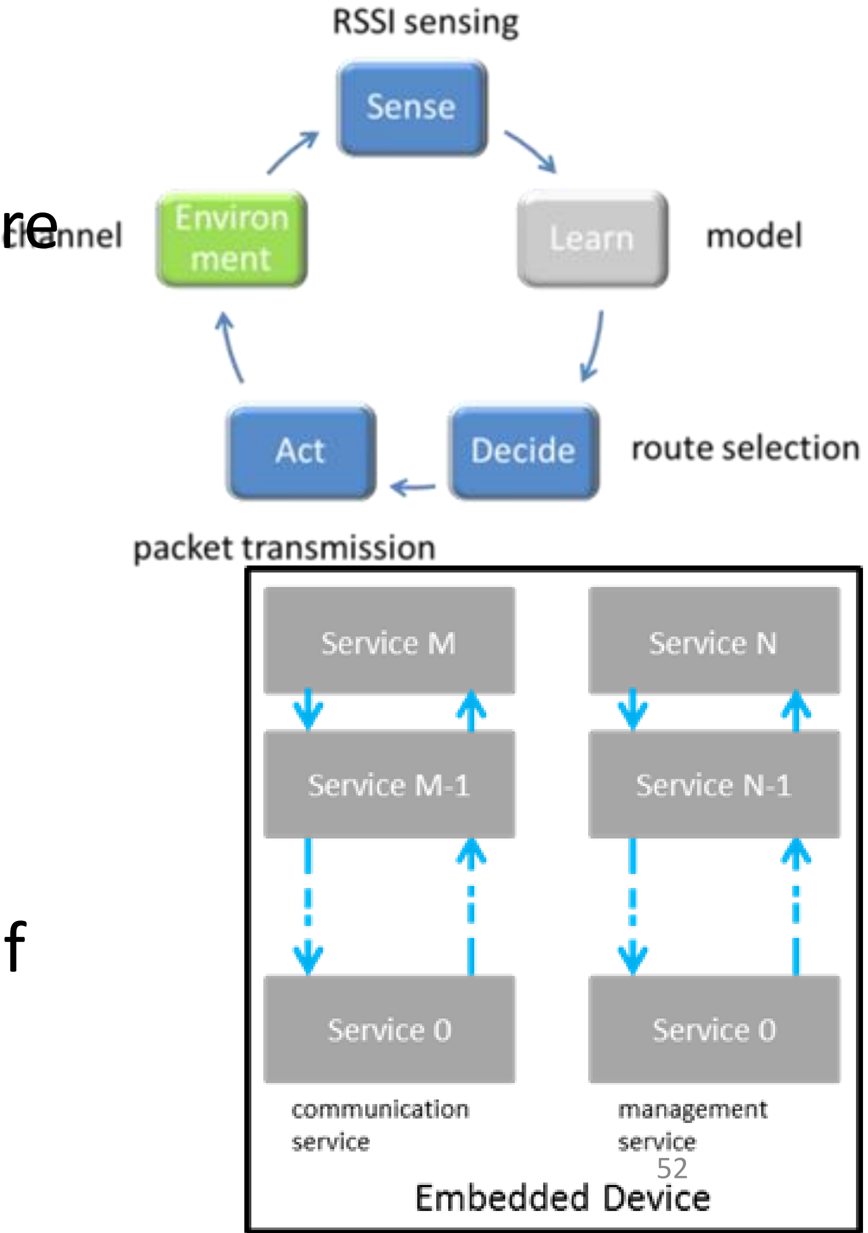
c_broadcast
sender 1.0
time_trigger_flg ☐
trigger_interval
trigger_no
trigger_th

c_channel
channel_no 134



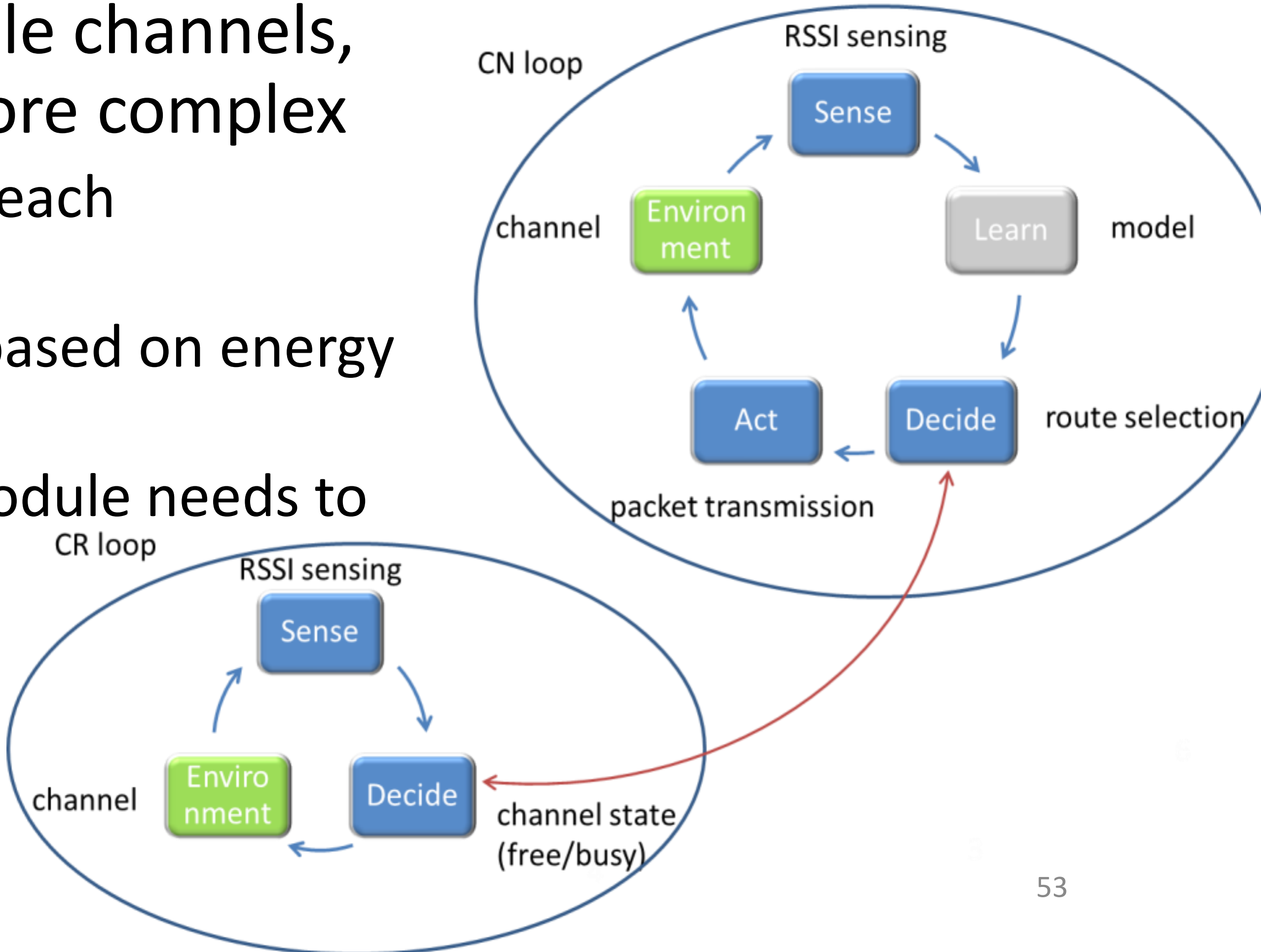
Network layer cognitive loop with cross-layer information using ProtoStack

- CRime extended with c_model source files where all the model specific logic is stored
  - The model creation and update functions are called from the c_route_discovery module.
  - This model is used to update the route costs in the routing table
  - c_multihop module then uses the resulting routing table when sending packets
- RSSI is straightforward using the Rime packet attributes which are visible to all the modules of the stack



Network layer cognitive loop with cognitive radio

- Assuming a cognitive radio device able to sense the spectrum and dynamically select available channels, the simple routing problem becomes more complex
  - the routing table has to contain entries for each potentially available channel
  - have a model for channel availability (e.g. based on energy detection)
  - At packet transmission time, the routing module needs to consult the channel availability and decide which channel is suitable for transmission.



Network layer cognitive loop with cognitive radio using ProtoStack

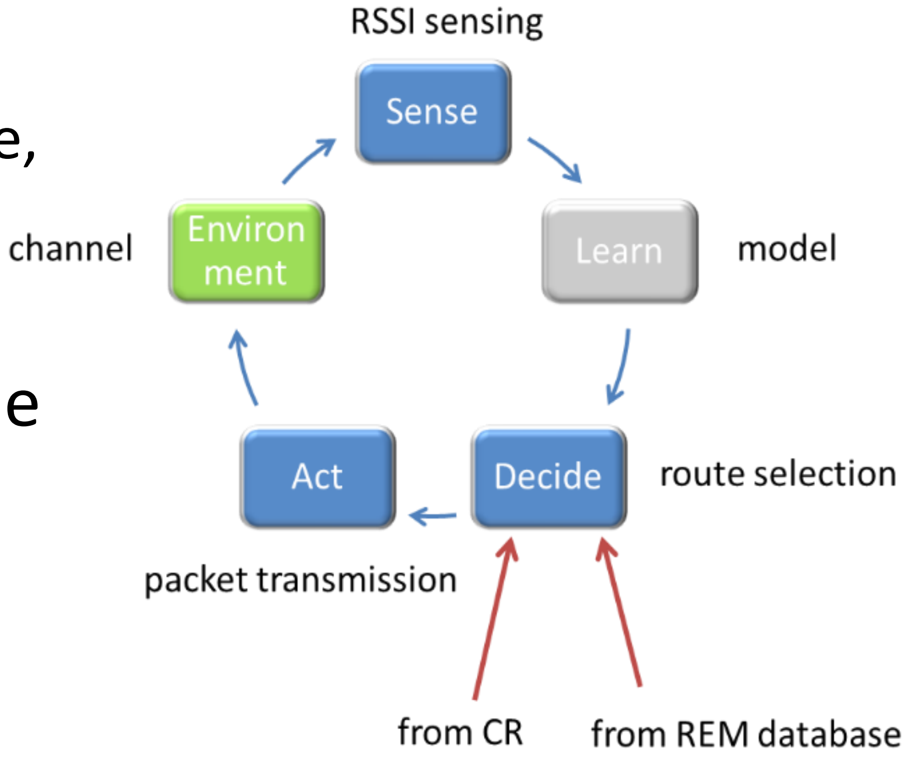
- that the CR loop is implemented independently of Crime
- the CR loop periodically updating the communication service’s pipe structure
- c_model module may need to be generalized
  - if the radio module can also switch between channels, then this has to be reflected in the network model and routing table

- CR loop is implemented using CRime modules
- c_channel_scan and c_channel_availability_model modules may need to be inserted between the c_broadcast and c_channel modules of the communication service

Soon on LOG-a-TEC

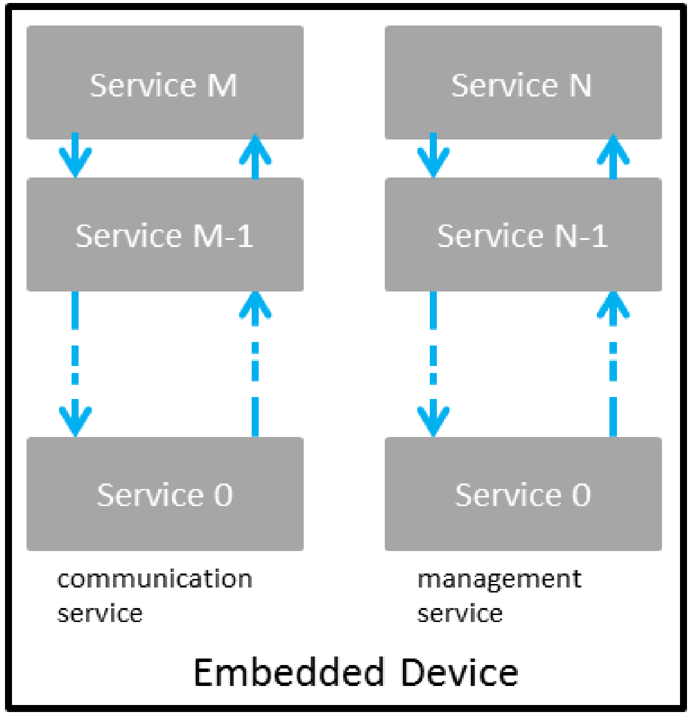
Network layer cognitive loop with radio environment maps

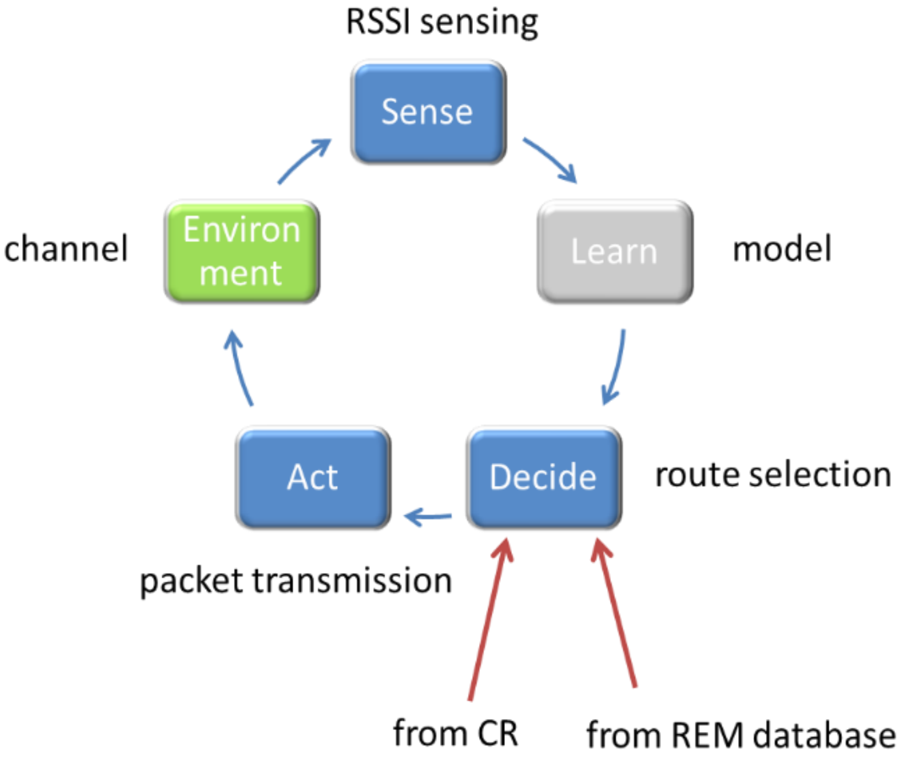
- Information related to the occupancy of the channels is most easily acquired from radio environment map services through a control channel
  - nodes can request information from the remote database,
  - such information is periodically broadcast.
- The network level cognitive loop needs to take into account spectrum occupancy information at the time of making a decision



Network layer cognitive loop with radio environment maps using ProtoStack

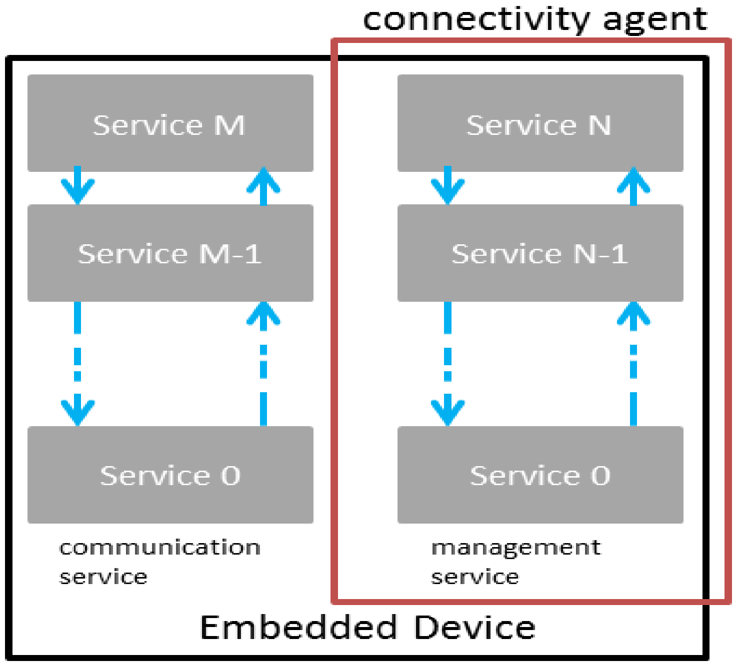
- dedicated management service communicating with the REM has to be implemented.
  - Can be done using existing CRime modules or developing additional modules that are necessary





Network layer cognitive loop with connectivity broker

- The connectivity broker
  - is a concept that provides abstractions necessary for developing large scale cognitive wireless network environments by enabling joint optimization of spectrum resources
  - operates in the control and management planes of the networks
  - the core concept behind it is the connectivity agent.
- ProtoStack tool can be used to implement node-level functionality corresponding to the connectivity agent



Conclusions

- Presented the ProtoStack tool and how it can be used in several experimentation scenarios
- The software will be soon available for download on GitHub
- The corresponding testbed will be available as soon as LOG-a-TEC port of Contiki is fully completed

Questions?