Cognitive Radio Experimentation World

Project Deliverable D5.3

Final report on demand-driven extensions and FIRE support actions

Contractual date of delivery:	30-09-2013
Actual date of delivery:	30-09-2013
Beneficiaries:	iMinds, IMEC, TCD, TUB, TUD, TCS, EADS, JSI
Lead beneficiary:	iMinds
Authors:	Jan Hauer (TUB), Mikolaj Chwalisz (TUB), Daan Pareit (iMinds), Wei Liu (iMinds), Ingrid Moerman (iMinds), Michael Mehari (iMinds), Lieven Hollevoet (IMEC), Somsaï Thao (TCS), Paul Sutton (TCD), Carolina Fortuna (JSI), Igor Ozimek (JSI), Matevz Vucnik (JSI), Tomaz Solc (JSI), Tomaz Javornik (JSI)
Reviewers:	Paul Sutton (TCD), Carolina Fortuna (JSI)
Workpackage:	WP5 – Demand Driven Extensions
Estimated person months:	50.9
Nature:	R
Dissemination level:	PU
Version	2.11

Abstract: This deliverable describes all the different kinds of extensions that were made. The extensions are an instrument to upgrade the CREW federation with a new set of functionality beyond the basic functionality developed in other CREW work packages. These extensions were made to support the internal use cases and the Open Call 2 experiments. Furthermore, FIRE support actions are listed.

Keywords:

cognitive radio, wireless networks, testbed, spectrum sensing, context awareness

Executive Summary

This deliverable describes how the federated CREW test facilities have been extended with a second set of new functionality beyond the basic functionality. It describes the extensions that were made to the portal, to the preceding work in other work packages (WP3 and WP4), to the Connectivity Brokerage and to the specific testbeds. The extensions are highlighted both for the internal use cases and the Open Call 2 experiments. Furthermore, FIRE support actions are listed.

We explained how the federated CREW test facilities have been extended with a second set of new functionality which has been defined in a demand-driven and open way based on the gaps identified, for both the internal use cases (WP6) and the experiments of Open Call 2 (WP7).

To this end, we extended the Connectivity Brokerage Framework for better usage in different scenarios and for improved usability in specific cognitive radio scenarios. A suitable database system was selected and a extension with some IEEE 1900.6 compatible parts was made. Implementation of the Connectivity Brokerage Framework was then also investigated and tested for the w-iLab.t and Log-a-tec testbed.

Furthermore, a framework (ProtoStack/ Crime) that allows composing communication services in a dynamic way was proposed and different optimizations were performed within the different testbeds (e.g. OMF, IRIS, GRASS-RaPlat, USRP sensing engine improvement etc.).

To conclude the document, the specific support actions required for the Open Call 2 experiments were also described, as well as a short update on the FIRE support actions.

List of Acronyms and Abbreviations

6LoWPAN	IPv6 over Low power Wireless Personal Area Networks
ACID	Atomicity, Consistency, Isolation, Durability
ADC	Analog to Digital Converter
AGRAC	Automatic Gain and Resource Activity Controller
AI	Air Interface
API	Application Programming Interface
ASIP	Application-Specific Instruction-set Processor
ATSC	Advanced Television Systems Committee
AWGN	Additive White Gaussian Noise
BAN	Body Area Network
BWRC	Berkeley Wireless Research Center
BS	Base Station
BTS	Base Transceiver Station
CAgent	Connectivity Agent
CB	Connectivity Brokerage
CBAN	Cognitive Body Area Network
CIC	Cascaded Integrator-Comb
CoAP	Constrained Application Protocol
CompNet	Composite Network Agent
CORDIC	COordinate Rotation DIgital Computer
COTS	Commercial Off-The-Shelf
CP	Cyclic Prefix
CR	Cognitive Radio
CREW	Cognitive Radio Experimentation World
CSMA	Carrier Sense Multiple Access
DC	Direct Current
DDR RAM	Double-Data-Rate Synchronous Dynamic Random Access Memory
DFE	Digital FrontEnd
DIFFS	DIgital Front end For Sensing
DVB-T	Digital Video Broadcasting - Terrestrial
ENV	Environment
EVA	Extended Vehicular A
FCC	Federal Communications Commission
FDD	Frequency Division Duplex
FER	Frame Error Ratio
FFT	Fast Fourier Transformation
FIR	Finite Impulse Response
FIRE	Future Internet Research and Experimentation Initative
FPGA	Field Programmable Gate Array
GENI	Global Environment for Network Innovations
GPS	Global Positioning System
GUI	Graphical User Interface
HID	Human Interface Device

ID	Identifier
IO	Input/Output
IPC	Interprocess Communication
I/Q	In-Phase / Quadrature-Phase
IRIS	Implementing Radio In Software
ISM	Industrial Scientific Medical
IVuC	Increase Value until Condition
LAN	Local Area Network
LTE	Long Term Evolution
MAC	Medium Access Control
MVCC	Multiversion concurrency control
NFS	Network File System
NoEE	Number of Experiment Equals
NTP	Network Time Protocol
OFDM	Orthogonal Frequency Division Multiplexing
OFDMA	Orthogonal Frequency Division Multiple Access
OS	Operating System
PA	Platform Agent
PC	Personal Computer
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PD	Probability of Detection
PFA	Probability of False Alarm
PLL	Phase Locked Loop
PMD	Probability of Missed Detection
PRB	Physical Resource Block
PSD	Power Spectral Density
PUB	Publish
RELT	Relative Error of performance is Less Than
RF	Radio Frequency
RFIC	Radio Frequency Integrated Circuit
ROC	Receiver Operating Characteristics
RSSI	Received Signal Strength Indication
RTT	Round Trip Time
Rx	Receiver
SDR	Software Defined Radio
SFTP	Secure FTP
SIMD	Single Instruction, Multiple Data
SIR	Signal Interference Ratio
SNR	Signal to Noise Ratio
SSRuC	Step Size Reduction until Condition
SUB	Subscribe
SUMO	Surrogate Modeling
SUT	System under Test
TCP	Transmission Control Protocol
TDD	Time Division Duplex

TSMC	Taiwan Semiconductor Manufacturing Company
TVWS	Television White Spaces
TWIST	TKN Wireless Indoor Sensor network Testbed
Tx	Transmitter
UDP	User Datagram Protocol
UML	Unified Modeling Language
UniNet	Unified Network Agent
US	Usage Scenario (see D2.1 for definition)
USB	Universal Serial Bus
USRP	Universal Software Radio Peripheral
VCC	Virtual Control Channel
WARP	Wireless Open Access Research Platform
WHIPP	The WiCa Heuristic Indoor Propagation Prediction
WInnF	Wireless Innovation Forum
WLAN	Wireless Local Area Network
WSAP	White Space Access Point
WSD	White Space Device
WSN	Wireless Sensor Network
ZMQ	ZeroMQ

List of Figures

Table of Contents

1 Introduction

In the CREW project demand-driven extensions are the tools to dynamically adapt the CREW federation to needs identified during the course of the project. While some of these extensions were envisioned during the project planning, not all needs were evident at that time. In addition, since external experimenters were/will be joining the project as a result of open calls during the project course (M12/M24/M36) a mechanism to extend the project with new functionality to better support these experimenters was required. This was the 'raison d'être' for Work Package 5 (WP5).

In deliverable D5.2 the extensions were described that were developed in Year 2 to support the internal use cases and the experiments of the Open Call 1 experimenters. Within this deliverable, we describe the further enhancements to support internal use cases (WP6) and we list the extensions which were needed for the Open Call 2 experiments (WP7).

We explained how the federated CREW test facilities have been extended with a second set of new functionality which has been defined in a demand-driven and open way based on the gaps identified, for both the internal use cases (WP6) and the experiments of Open Call 2 (WP7).

To this end, we extended the Connectivity Brokerage Framework for better usage in different scenarios and for improved usability in specific cognitive radio scenarios. A suitable database system was selected and a extension with some IEEE 1900.6 compatible parts was made. Implementation of the Connectivity Brokerage Framework was then also investigated and tested for the w-iLab.t and Log-a-tec testbed.

Furthermore, a framework (ProtoStack/ Crime) that allows composing communication services in a dynamic way was proposed and different optimizations were performed within the different testbeds (e.g. OMF, IRIS, GRASS-RaPlat, USRP sensing engine improvement etc.).

To conclude the document, the specific support actions required for the Open Call 2 experiments were also described, as well as a short update on the FIRE support actions.

2 Demand-driven Extensions Derived From Internal Use Cases

This section describes the extensions that were developed during the third year of the CREW project as a result of the shortcomings identified in the CREW internal use cases (WP6).

2.1 CREW Portal & CREW Repository

The portal (http://www.crew-project.eu/portal) and the repository (http://www.crew-project.eu/repository) were kept up to date during CREW's third year. Some of the notable changes are the following:

- Linking to static files for processing the CREW Common Data Format (CDF) has been replaced by linking to a github repository which has always the latest scripts available (see also 2.5.1.2).
  http://www.crew-project.eu/repository/scripts
- Information on running an experiment on iMinds' w-iLab.t testbed has been updated to reflect the new way to request an account and to use the Emulab environment
  http://www.crew-project.eu/portal/wilab/basic-tutorial-your-first-experiment-w-ilabt
- The description of the WARPs and the mobile robots in iMinds' w-iLab.t testbed has been added
  http://www.crew-project.eu/content/configuration
  http://www.crew-project.eu/content/warp-usage

2.2 Continuation of Other Work Packages

2.2.1 Creating the Federation (WP3)

2.2.1.1 Maintenance and update of the WInnF Transceiver API source code

CREW attended the 73rd Wireless Innovation Forum (WInnF) Working Meeting, from 31st October to 2nd November 2012, held by Harris Corporation in Melbourne, Florida. CREW presented there the initial software upload realized, then discussions were held on the real-time limitations of the solution based on a widely available and used platform such as Ettus Research USRP2 board. The current solution requires indeed about 3 ms of anticipation in bursts creation commands and the Absolute Timing programming mode implementation is difficult.

Mr Matt Ettus put CREW in contact with Mr Balint Seeber who is Ettus Research Expert in our issue. Direct interactions between CREW and him took place in January and February 2013, as a follow-up of the WInnF findings. Those exchanges enabled to identify promising perspectives concerning the degree of real-time performance achievable using USRP solutions, by taking advantage of digital frequency tuning, that enables, provided useful bandwidth is smaller than the digitized bandwidth, to have very fast frequency switching that can mitigate the relatively slow control reactivity from PC to USRP board. Evaluations concluded that **some 100 µs** would be a target for a minimum required anticipation for a burst creation.

2.2.1.2 Finalize the automatized support for common data format in the LOG-a-TEC testbed

To enable collaboration with other testbeds in the CREW federation, support for the CREW Common Data Format (CDF) has been developed for the LOG-a-TEC testbed. Spectrum sensing experiments that have been described in CDF files can be re-run on the LOG-a-TEC testbed without any additional programming. Results of such experiments can also be saved back to a CDF file.

As described in the deliverable D3.2 under “VESNA interfaces”, the high-level interface to wireless sensor nodes deployed in the LOG-a-TEC testbed is a HTTP-like protocol that is accessible through a gateway on the Internet.

An open source Python module library[1] called vesna-alh-tools has been developed that provides a convenient interface to the testbed through a HTTP-like protocol. By using it, a program written in the Python programming language running on the experimenter’s computer can remotely control the sensor nodes in the LOG-a-TEC testbed. At the base of this library is the vesna.alh module that provides convenience functions for communicating with the sensor nodes through the LOG-a-TEC Internet gateway. On top of this module a CDF support layer vesna.cdf provides abstraction over the raw resources that are exposed by sensor nodes in the testbed and adapts the LOG-a-TEC specific interface to the common CREW testbed schema. The vesna.cdf.xml module then provides CDF parsing and serialization utilities. Finally, a run¬_cdf_experiment executable uses this infrastructure to directly run an experiment described in a CDF file without the need for any additional programming. This layered approach allows the experimenter to choose between the level of testbed independence and LOG-a-TEC specific features that best suit his needs. Specifically, the Python interface allows for a level of automatic control over the experiment that is not directly describable within the CDF format.

Because of the specific implementation details of the LOG-a-TEC testbed, several metadata fields needed to be added to the CDF in order for an experiment description to be applicable to VESNA devices. Specifically, in order for a CDF file to be directly usable with the run_cdf_experiment tool, it must contain information about device access like the gateway URL to use and VESNA management network addresses. Due to the rigid nature of the CDF's XML schema, these metadata fields have been added as JSON-encoded strings (prefixed with “Additional VESNA metadata follows”) within free-form string sections of the CDF file. Should the CDF format be expanded to support these fields in the XML schema itself, these additional JSON-encoded fields could later be replaced in favour of native XML tags.

2.2.2 Benchmarking the Federation (WP4)

2.2.2.1 The WHIPP tool extension for REM visualization

The WiCa Heuristic Indoor Propagation Prediction (WHIPP) tool is a heuristic network planner, developed and validated for indoor environments [1]. The model has been constructed for the 1.8 - 2.6 GHz band and its performance has been validated with a large set of measurements in various buildings.

In the third year of CREW, the WHIPP tool is extended from a simple predictor to a monitor, which can be used to visualize the Radio Environment Map (REM) in the w-iLab.t testbed. The backbone of this tool relies on the distributed spectrum sensing system, as shown on the left side of Figure 1. We use several sensing devices, including USRP, IMEC sensing engines and regular Wi-Fi devices together for monitoring the w-iLab.t environment and collect the measurements into a central database. Before combining the measurements from different devices, we need to first calibrate the devices so that they don’t measure signal with different offset. Apart from that, the sensing devices have different measurement capabilities, some devices produce measurement faster than others; therefore we apply certain data fusion mechanisms to ensure the speed of data production by different sensing engines are comparable.

The REM interface is part of the original WHIPP tool, a new “Monitor Wilab” tab is added on the web interface. When clicking on it, the w-iLab.t Zwijnaarde testbed topology is loaded. The tool reads the RSSI of the selected nodes from the central database regularly. The discrete RSSI results are interpolated to have a realistic view of the REM on the testbed. A snapshot of the final result of the interpolation is presented on the right side of Figure 1.

[1] https://github.com/sensorlab/vesna-alh-tools

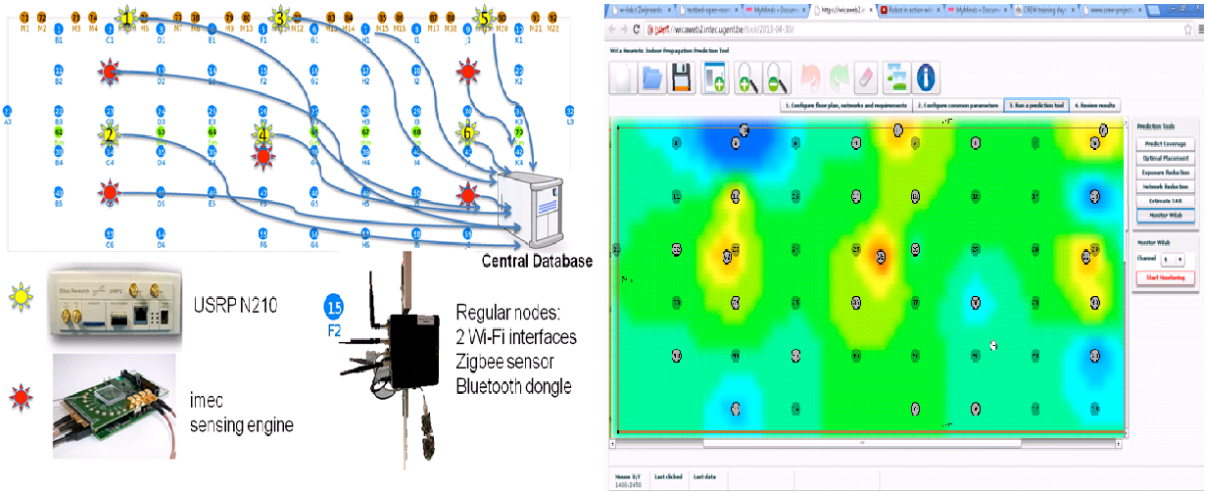


Figure 1 Distributed spectrum sensing and the monitor interface on WHIPP tool

2.2.2.2 SUMO tool

Out of the box, Surrogate Modelling SUMO toolbox [2] is used as complete multi-dimensional optimizers. It is targeted to achieve accurate models of a computationally intensive problem using reduced datasets. From the reduced datasets, the tool generates accurate Surrogate Models to evaluate design objectives.

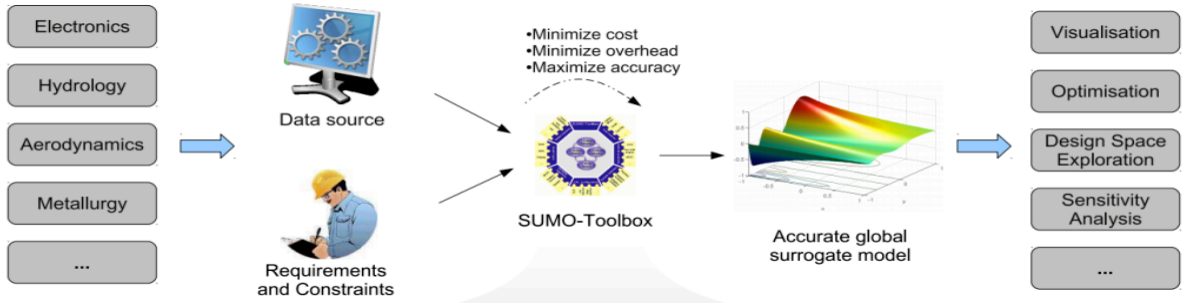


Figure 2 The process of generating accurate surrogate model

The SUMO toolbox bundles both the control and optimization functions together. The control function sitting at the highest level manages the optimization process with specific user inputs. The figure below describes SUMO tools in a nutshell highlighting the control and optimization functions together.

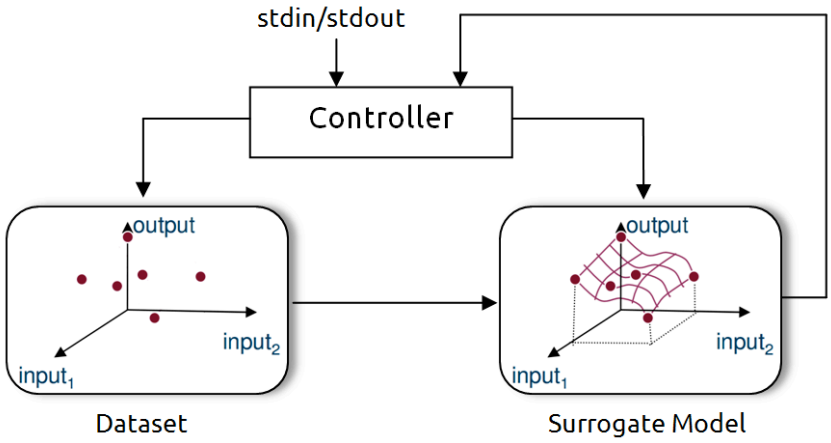


Figure 3 Out of the box SUMO tools in a nutshell view

From the figure above it can be seen that the controller manages the optimization process starting from a given dataset (i.e. initial samples + output performance) and generates a surrogate model. The surrogate model approximates the dataset over the continuous design space range. Next, the controller predicts the next design space element from the constructed Surrogate model to further meet the optimization's objective. Depending on the user's configuration, the optimization process iterates until conditions are met.

The aim is to use SUMO tools as a standalone optimizer and put them inside an experimentation framework. This means from out of the box SUMO tools, the loop is broken, the control function is removed and clear input/output interfaces are created to interact with the controlling framework. The figure below shows how modified SUMO tools are integrated in the wireless testbed.

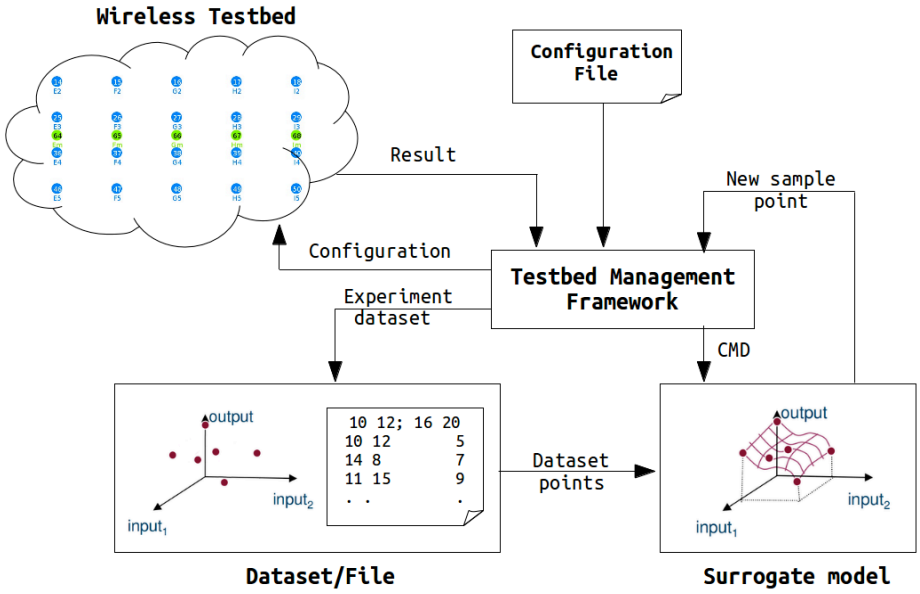


Figure 4 Integration of modified SUMO tools in the wireless testbed

The figure above shows the use of the testbed management framework instead of the default controller. Suppose we are in a context of a wireless experiment, concerning two parameters: transmit power and the node location. The goal is to find the optimum combination of those two parameters to achieve the maximum throughput. There is a format agreement between SUMO toolbox and the testbed management framework. After the initial datasets are achieved, SUMO toolbox determines the parameter set for the next experiment, and writes the parameters in the dataset file, as shown in the

figure above. The testbed framework performs the experiment, and appends the result on the same row in the data file (in this case it is the throughput result). This iteration goes on until the stopping criteria are reached. More details of this example is described in Figure 5 and the paragraph below, and a more complex experiment on the w-iLab.t testbed with SUMO optimization is described in D6.3, Section 2.3.

This framework performs the same tasks which were already implemented by the previous controller except additional tasks like experimentation on the wireless testbed, storing the dataset on a separate file, and reading experiment configuration from a file. It should be understood very well that the operation of SUMO tools has not changed at all except replacement and addition of a few working blocks. A more general pictorial presentation on how SUMO tools work is also presented on the next figure.

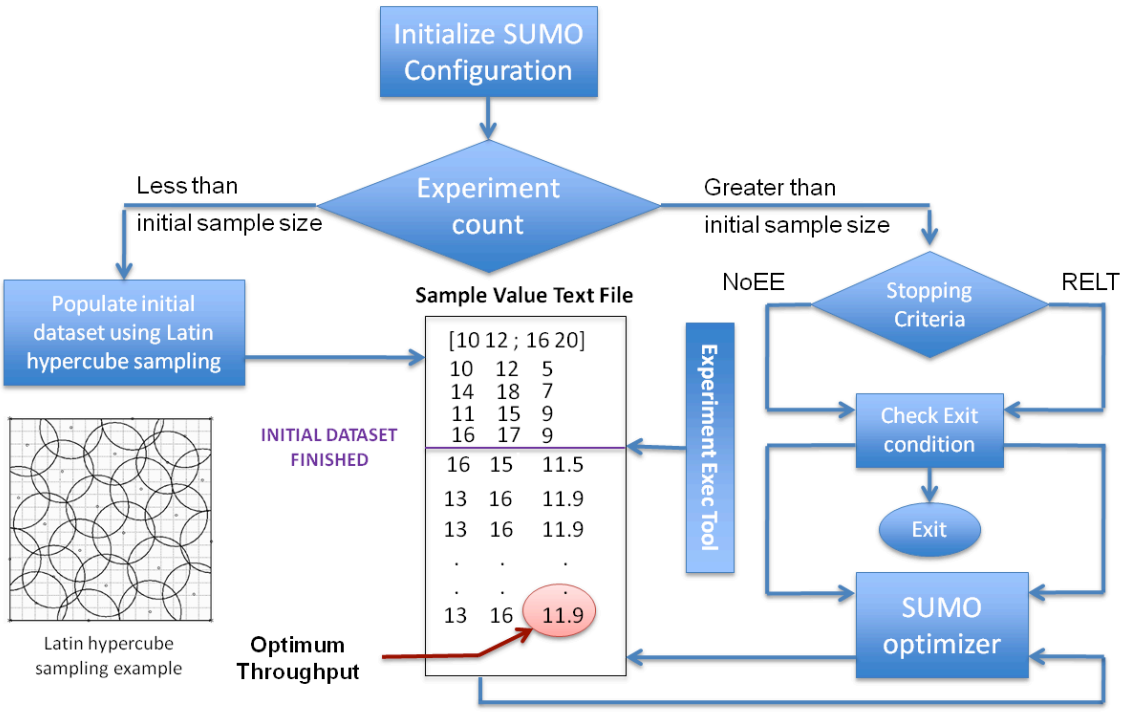


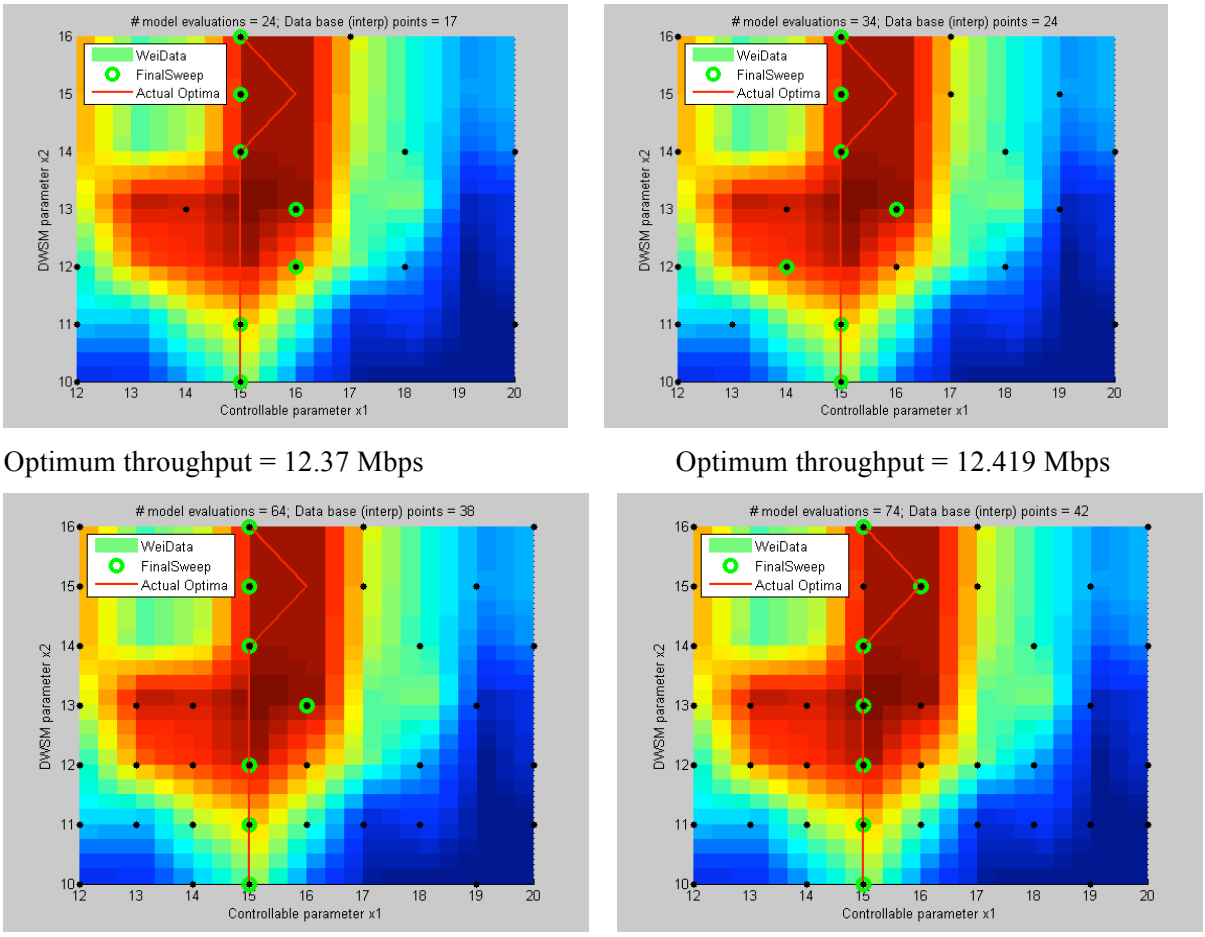
Figure 5 Complete modified SUMO toolbox optimization over two dimensional design space

In the figure above, a complete optimization using the modified SUMO tools is presented. The SUMO tools optimize a two dimensional design space (i.e. transmit power 10dbm to 16dbm and transmitter location id 12 to 20) problem.

Before SUMO tools start the optimization process, it requires a minimum number of initial samples which is used to generate the first surrogate model. The figure above shows four selected initial dataset pairs (i.e. [10 12], [14, 18], [11, 15], [16, 17]). Latin hypercube sampling is used to generate the initial samples which verifies that the samples are equally distributed across the design space range.

Next, the experiment is conducted at each initial sample and the dataset (i.e. initial samples + output performance) is fed to the optimizer. The SUMO tools first generate the surrogate model and next calculate a new sample point to reach the global optimum. Using the calculated sample point, we do a new experiment and update the dataset. As optimization progresses, the SUMO tools approach the global optimum point. The figure below shows the different steps during SUMO optimization process.

Optimum throughput = 11.9 Mbps Optimum throughput = 12.184 Mbps



Optimum throughput = 12.37 Mbps

Optimum throughput = 12.419 Mbps

Figure 6 The different steps during SUMO optimization

From the above figures we see that as the number of iterations increases the simulation optimum coincides with the global optimum. The first figure shows 17 experiments and optimum performance of 11.9 Mbps. The second figure shows 24 experiments and optimum performance of 12.184 Mbps. The third figure shows 38 experiments and optimum performance of 12.37 Mbps. The last figure shows 38 experiments and optimum performance of 12.419 Mbps. However useful the above example was, it was not elaborate enough to fully describe SUMO tools. Thus a detail analysis on a "wireless press conference" optimization problem is presented in Deliverable 6.3 "US3-Horizontal Resource Sharing in ISM bands".

We end the optimization after a stopping criteria is met. Two possible stopping criteria are when the Number of Experiment Equals (NoEE) a given value and when the Relative Error of performance is Less Than (RELT) a given threshold and both are implemented in "experiment execution" tool.

2.2.2.3 Benchmarking framework extension

Benchmarking in the scope of wireless networks focuses on the methodology and approach of conducting the experiment. In the scope of CREW, a benchmarking framework to meet this demand is designed with a number of tools. The benchmarking tools developed are experiment definition, experiment execution, and result analysis. Each separate tool plays a role in the process of wireless network benchmarking. A brief summary of the CREW benchmarking framework is included here; more details are elaborated in section 2.2 of deliverable D4.3.

The experiment definition tool is used to define, configure, and make changes to wireless experiments that are to be conducted on the testbed. The second tool, the experiment execution tool, schedules and executes the already defined experiment. This section also defines the optimizer selection and different configurations that are also passed along. The last tool, the result analysis tool, is used to conduct

performance comparison of an experiment to a reference solution. In sum, an experimenter defines an experiment description, schedules and executes his experiment, and finally analyzes its performance.

The set of benchmarking tools that were developed in previous years are further improved with a number of functionalities.

Updates on the "experiment definition" tool

The first update made in the "experiment definition" tool is the merging of the Solution Under Test (SUT) and Environment (ENV) configuration files. Now the XML configuration file contains both the SUT and ENV configuration files together and the experiment definition tool accepts one configuration file instead of two.

Second, the project name and experiment name sections are included in the "experiment abstract" section. This modification is connected to a recent change in the w-iLab.t Zwijnaarde testbed development. This is related to the control and management framework of wireless testbeds. Most of the European research projects involving wireless testbeds have previously implemented the cOntrol and Management Framework (OMF). However, recently the OMF loading image functionality is disabled, and handled by *Emulab* [9]. The reason for this migration is that Emulab is more advanced for loading images on a node. Besides, other parts of the the iLab.t testbed (which w-iLab.t is part of) already use Emulab as its primary management framework and a unified interface internally was envisaged. Note that the normal experiment flow is still managed by OMF, as is desired. Emulab is a network testbed, giving researchers a wide range of environments in which to develop, debug, and evaluate their systems. Emulab testbed management follows a structure where each experiment is described by a unique project name and experiment name and hence the need for the changes.

The last update made on this tool is on the "Included Nodes" section. In the previous implementation, a group could only configure one node but the current implementation supports more than one. Furthermore, two important functionalities added in this section are "Send Message" and "Execute Program". The Send Message functionality gives the option to send dynamic messages to a running process. A typical example is a running process waiting for user input to get dynamic messages. The second functionality, Execute Program, lets the experimenter execute a program at a given time. The combined functionality of the two helps create more advanced scenarios.

Updates on the "experiment execution" tool

In this tool, we did the most important changes over the whole range of benchmarking tools. The first change made was the categorization of optimization problems. It is a fact that a single optimization method cannot solve all types of problems. Experimenters depend heavily on specific optimization methods for their specific problems. To this end, three optimizers are selected and plugged into the "experiment execution" tool. These are "Step Size Reduction until Condition" (SSRuC), "Increase Value until Condition" (IVuC), and "Surrogate Modeling" (SUMO) tools.

SSRuC is already covered in deliverable D4.3 and we only discuss IVuC and SUMO tools optimizers. IVuC optimizer is designed to solve problems which show either increasing or decreasing performance along the design parameter range. A typical example could be bandwidth optimization of a Wi-Fi experiment. By setting datagram error rate as a performance parameter, the highest bandwidth is searched and optimized for certain datagram error rate threshold.

The algorithm used by IVuC optimizer is similar to the SSRuC optimizer. However, the main difference between the two is that the SSRuC optimizer performs a complete experimentation cycle before locating the local optimum value whereas the IVuC optimizer performs a local optimum performance check after the end of each experiment. Later on, both approaches refine their design parameter range and restart the optimization process to further tune the design parameter. The figure below shows the different steps involved in IVuC optimizer. In the context of one-dimensional optimization, the IVuC means linear searching for an optimum area and zoom in on this area in the next iteration. Three iteration of one-dimensional optimization is illustrated in Figure 7.

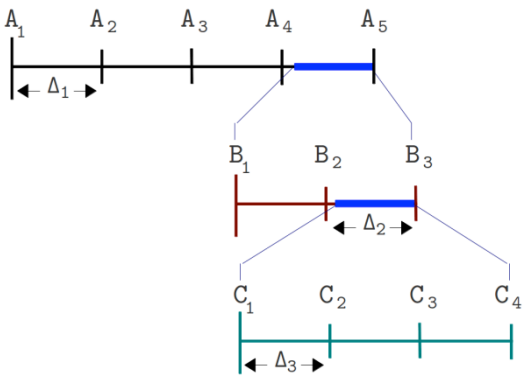


Figure 7 The different steps of an IVuC optimizer over a design parameter range A1 to A5.

The last optimizer supported is SUMO tools and unlike the previous two cases it performs multi-dimensional optimization. Usually researchers optimize problems that contain more than one parameters and their complexity urges the use of multi-parameter optimizers. SUMO tools in particular are targeted to achieve accurate models of computationally intensive problems using reduced datasets. A complete explanation of the SUMO tools, however, is vast so we devoted a separate section for it. Please refer to the previous subsection for a detailed explanation of SUMO tools.

The current needs of most researchers are covered by the three optimizer sets. However, when the need for a new set arrives, it is as simple as plugging it in and integrating it with the benchmarking tools.

Next, upon selection of a specific optimizer, unique configuration settings are automatically populated. A configuration example of the SUMO tools optimizer is shown below.

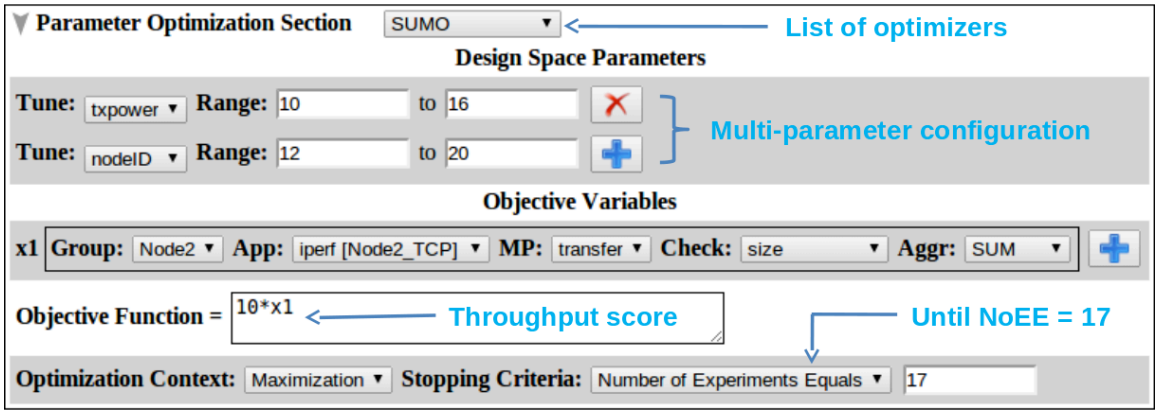


Figure 8 SUMO optimizer configuration at glance

In the figure above, a two dimensional parameter is optimized to maximize throughput of a TCP communication. The experiment is iterated within design parameter space (i.e. TxPower 10 to 16 and nodeID 12 to 20) and after NoEE equals 17, the optimization is stopped. Another addition to this section is the "objective function text to expression" interpreter. The objective function is defined using text description and parsed into an expression for objective function evaluation.

Update on the "Result Analysis" tool

The "Result Analysis" tool is a new addition, which was conceptual in previous implementation. It contains graphical analysis and score calculation and comparison sections. The graphical analysis section compares multiple experiment data, displaying it graphically and thus analyzing group performance. The figure below shows a graphical comparison of datagram vs. time for six consecutive experiments.

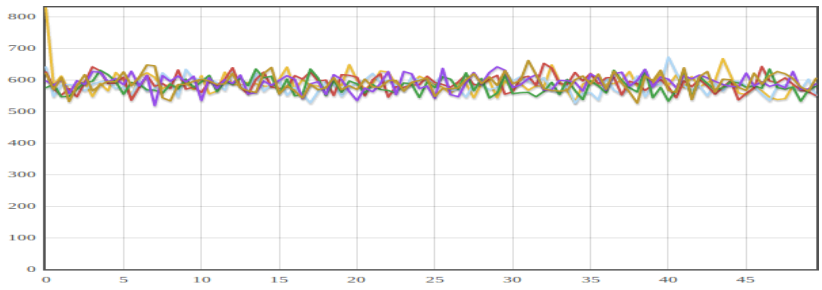


Figure 9 Datagram vs. time plot for six consecutive experiments.

Moreover, this section allows users to define a number of performance metrics such as sample average, sample standard deviation, etc. In the "Result Analysis" section of the benchmarking framework, score calculation and comparison, is used for objective comparison of experiment data. A score is defined as an arithmetic combination of performance metrics. A performance metric is a measure where a physical meaning is given to an experiment data. Mean Opinion Score (MOS), throughput and transmission exposure are examples of performance metrics. An optimization problem to maximize audio quality and minimize transmission exposure, for example, defines a score out of MOS and transmission exposure metrics.

2.3 Connectivity Brokerage Framework Extensions

During the second year of the CREW project a collaboration between CREW and UC Berkeley had been established through various bilateral discussions between CREW and Jan Rabaey from Berkeley Wireless Research Center (BWRC). The goal of this collaboration was to cooperate towards prototyping the *Connectivity Brokerage* (CB) framework [3] and applying it within the CREW infrastructure. The CB framework is a means to enable cooperation between devices in order to achieve spectrum sharing that *"enables diverse wireless technologies to exchange information and to collaborate in a seamless fashion, making a joint optimization of the scarce spectrum resources possible."* [3]. As described in D5.2, the *Connectivity Brokerage* (CB) framework allows experimenters to focus their investigations on the subset of cognitive radio approaches of interest and establishes a means for collaboration among different CR entities at different levels of abstraction (please refer to section 2.1 of deliverable D5.2 for a description of the CB framework).

The benefits for a CREW experimenter are that the CB framework describes a structured functionality set that should be supported by each entity in the framework, called Connectivity Agent (CAgent). This allows better structuring an implementation and focus on individual functionality. Furthermore, experimenters may use the CB framework to "fill up" the missing parts of their CR system-under-test (SUT) with generic CB framework software components developed in the CREW project, for example a repository that stores spectrum sensing results. In addition to providing a first implementation of the CB framework during the second year of the CREW project [3] the inter-CAgent communication via the so-called virtual control channel (VCC) and [4] the serialization of messages was specified.

In the third year, the collaboration between CREW and UC Berkeley has continued. In the following we report on the tasks that have been carried out in the third year of the CREW project, among other things, a redesign of our initial CB implementation, a performance evaluation of various database systems for their suitability to store spectrum sensing data and a new mechanism that allows CAgents to specify the type of data contained in their repositories to facilitate discovery of relevant data.

2.3.1 Revised Software Design

The first CREW implementation of the CB framework had to a large extent been used for CR experiments in the TWIST testbed. Once the implementation was applied to other platforms it became clear that it included some unwanted dependencies. For example, the Discovery class included a

function and definitions tailored to a specific sensing platform (TelosB platform); also the Repository class included a reference to a specific MySQL database to store spectrum sensing data. In order to decouple the framework from these platform- and testbed-specific functions a redesign was undertaken during the third year of the CREW project.

Our revised design tries to better separate the CB framework from the scenario and involved platforms. The revised software approach is based on two levels of abstraction, a set of generic modules and derived ones. The decoupling is realized via object-oriented inheritance. The generic modules are the CB core and the derived modules are scenario-dependent as shown in the following figure.

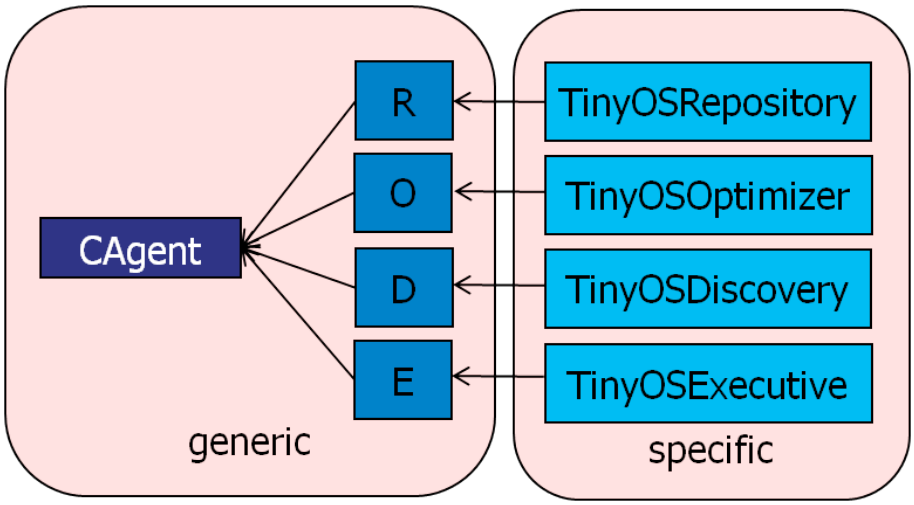


Figure 10: Revised class diagram

The generic functions are grouped in the basic classes: CAgent, Discovery, Repository, Optimizer and Executive which interact according to the following UML diagram.

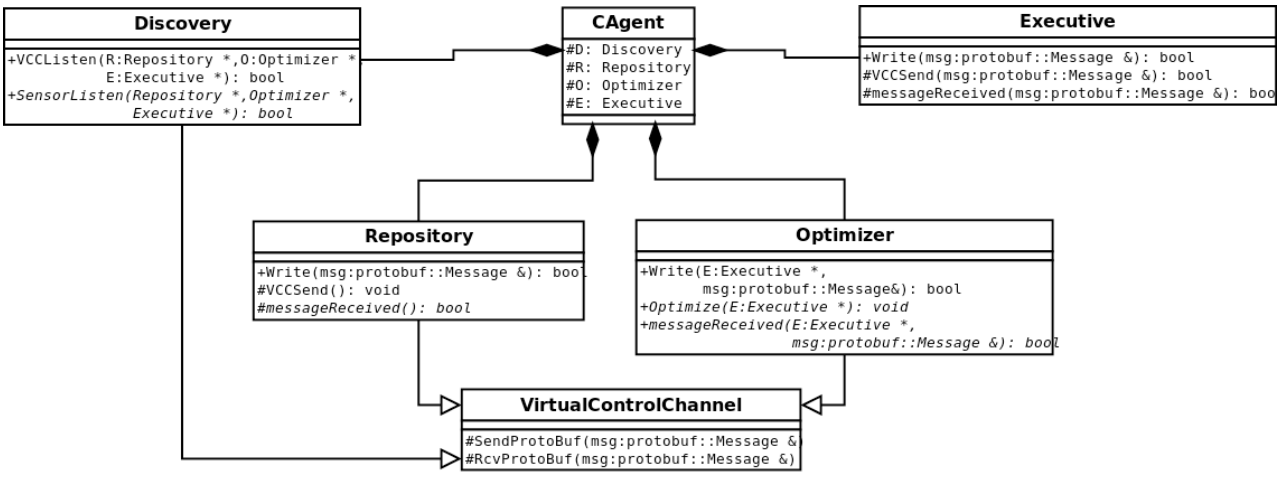


Figure 11: Revised UML diagram

The *CAgent* class is the basic block of the CB framework. It only has the reference to the four blocks that forms a CAgent now. The *Discovery* class has a similar function as a dispatcher. It has generic function in order to listen the VCC and a virtual function to listen the sensor data. This virtual function should to be implemented by the derived class, because it is technology dependent.

The *Repository* class has a method Write that receives intra-CAgent communications from other modules, and a virtual method to manage the received messages. Also, it has a method to send messages to the VCC. The new *Optimizer* class has a method that receives messages from other blocks and a virtual method to manage them. Another virtual method that has been included is Optimize(). This function should be implemented in the specific scenario because each application will need a different optimization algorithm. Like the *Optimizer* and the *Repository*, the *Executive* module has two functions to manage the received messages. Also, it provides a function for sending different kinds of messages over the VCC.

In an instantiation of this generic implementation the specific blocks are derived from the base classes. The four blocks have a derived implementation that inherit the generic methods and implement other specific functions. For example, derived classes for the TelosB platform have been developed as shown in the following figure.

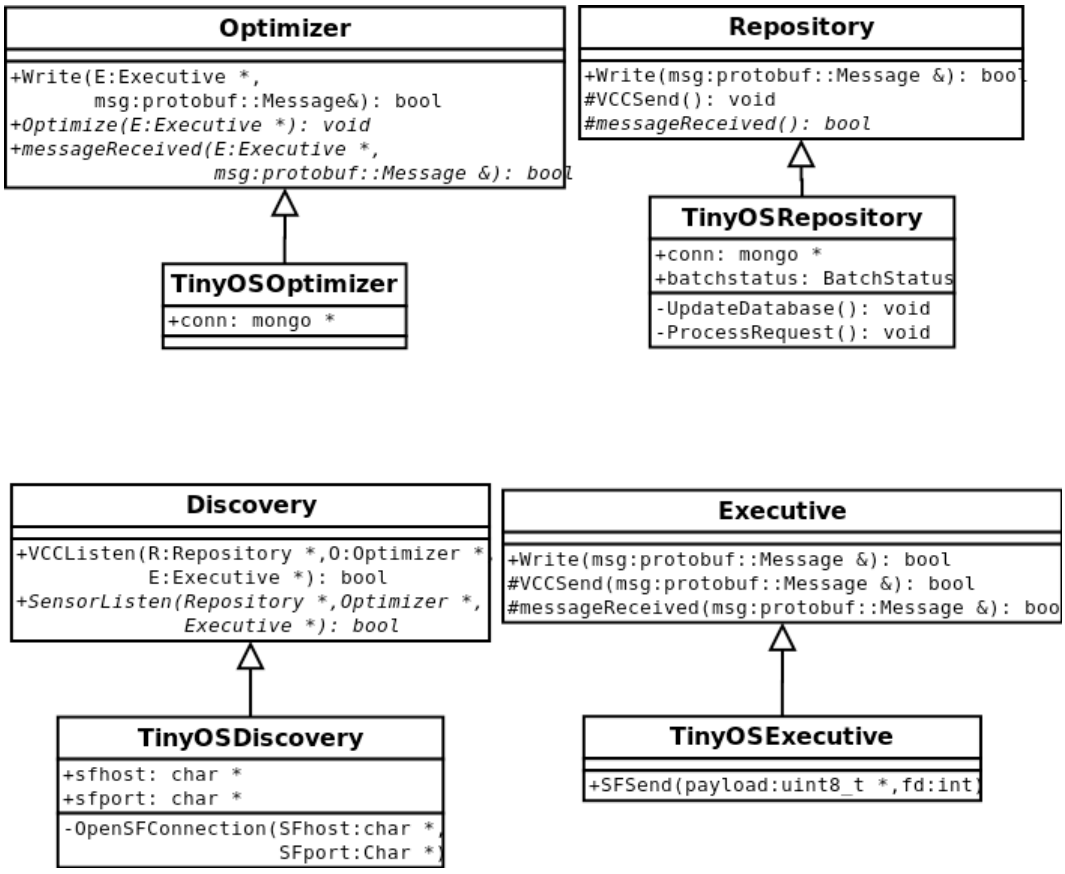


Figure 12: Revised part class diagram for TelosB platform

The *TinyOSDiscovery* class includes the data, pointers and functions to receive the spectrum sensing data from the TelosB platform over a serial port. The *TinyOSRepository* class has a function to update the database when it receives new data. For example, the revised MongoDB database introduced below can be pointed to from this class. Finally, *TinyOSRepository* has a function to manage the request message sent from others blocks or CAgents.

The *TinyOSOptimizer* class has a pointer to the database which it may need to access while executing the optimize algorithm. Finally, the *TinyOSExecutive* class maintains the function *SFSend* (c.f. previous implementation described in D5.2) that enables the communication with the TelosB nodes.

The last part of the revised design facilitates communication among CAgents. The spectrum sensing data and metadata (described below) are going to be queried by other CAgents that have no direct access to the database. Therefore, it is important that the framework supports CAgent to Cagent

communication, independently of their types. This communication is abstracted by the VCC. Now, a Broker class is inserted into the framework to improve the possibilities of the VCC: while in our previous design N (NodeCAgents) to 1 (TestbedCAgent) communication was supported, the added broker component now supports N to N communication.

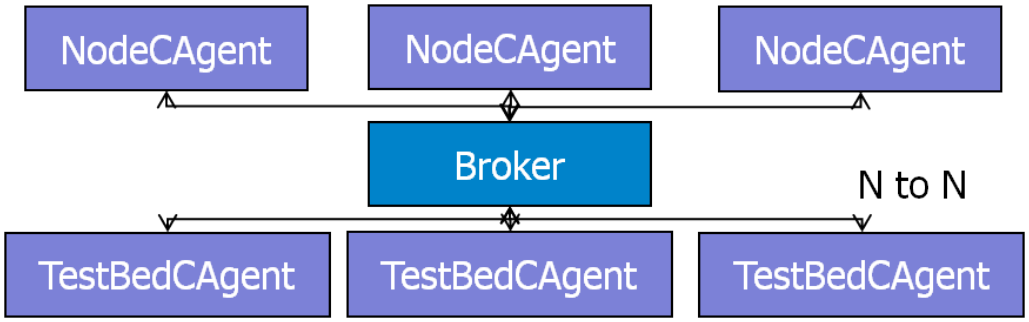


Figure 13: Broker design and CAgent communication.

The task of the *Broker* is to forward the message from the NodeCAgents to all TestbedCAgents, and in the opposite direction.

2.3.2 Comparison of Databases

Within the Cognitive Brokerage framework the task of the *Repository* module is to store the environment information collected by the sensors. It is expected that there may be a massive amount of such spectrum sensing information that needs to be stored. In our previous design (c.f. deliverable D5.2) a MySQL database was selected to store spectrum sensing information. The choice was motivated by the popularity of the database rather than its performance or suitability for our particular use cases. In the third year of the CREW project we revisited the choice of the database within the Repository component more thoroughly. We conducted a study to understand the suitability of four popular databases for our scenarios: MySQL, PostgreSQL, SQLite and MongoDB. The selection of these four databases is based on the fact that all of them are quite popular and easily available. Three of them use a compatible language, SQL. MongoDB is an NoSQL database that has been selected to test a different database approach.

One of the requirements for the database is to be accessible from everywhere and everyone. In the following we compare the local and remote access. Then, the most important programming languages used to access to the database are described.

MySQL: The database host should be installed on a mysql server. This is the only requirement in order to access the data locally or remotely. The access function is the same and supports any IP address. MySQL supports multiple transactions at the same time, multithreading and multiuser. Each user can have different privileges for different databases. The most important programming languages in order to access MySQL databases are: C, C++, C#, Pascal, Delphi (via dbExpress), Java, Lisp, Perl, PHP, Python, Ruby or Tcl.

PostgresQSL: As in MySQL, the only requirement is to install a postgreSQL server on the host. The remote and local access works in the same way. PostgreSQL supports multiple transactions at the same time, multiproccess and multiuser. Each user can have different privileges for different databases. The most important programming languages in order to access postgreSQL database are: C, C++, Java, PHP, Tcl, Python, Ruby or Perl.

SQLite: SQLite database store the information in a simple file, that is blocked for each transaction. This characteristic makes it unsuitable for remote access. However, it can be achieved via NFS or Samba. SQLite does not support multiuser or multiaccess, only multiple reading access. This is because of the database file based architecture. Moreover the SQLite database does not have users and

passwords. The most important programming languages in order to access postgreSQL database are: C, C++, Java, PHP, Tcl or Python. The three databases selected in this benchmark are multi- platform and can be installed in Windows, Linux and OS X.

MongoDB: Instead of storing data in tables as is done in a "classical" relational database, MongoDB stores structured data as JSON-like documents, using dynamic schemas (called BSON), rather than predefined schemas. An element of data is called a document, and documents are stored in collections. One collection may have any number of documents. Compared to relational databases, we could say collections are like tables, and documents are like records. MongoDB supports local and remote connections. The installation on the host is simple. MongoDB's design philosophy doesn't care much about users and passwords but it supports them. The most important programming languages in order to access MongoDB databases are: C, C++, Java, Perl, PHP, Ruby or Python.

2.3.2.1 Buffering and Caching Concepts

Multiple agents will access the database when the system will grow. Therefore, it is important to reduce the time that the agents spend reading or writing data (concurrency). We have studied how the three databases work when the agent wants to introduce or read multiple data and if there are some buffering libraries in order to simplify this mechanism.

MySQL uses InnoDB as an ACID (Atomicity, Consistency, Isolation, Durability) compliant, transactional storage engine using Multiversion concurrency control (MVCC) technology. The InnoDB engine has an insert buffer that caches updates to secondary index entries and applies them in the background. This can significantly speed up inserts, reducing the number of physical writes required by combining many updates. If a secondary index page has outstanding updates when it is needed for a query the updates will be merged first. As of version 5.5 the insert buffer is also used as a buffer for other types of writes, improving the performance of UPDATE queries as well. Prepared statements are a method to accomplish the task of executing a query repeatedly, albeit with different parameters in each iteration. C and C++ APIs allow the use of prepare statements which improve the security of the MySQL server.

PostgreSQL supports a data storage motor, its default Postgres storage system (Postgres Storage System). It uses caching in the same way as MySQL does. The modified and written data is moved to cache in order to speed up the following queries. A C API for PostgreSQL implements Prepared Statements in the same way as MySQL. This allows to increase the security and the usability when the system executes queries repeatedly.

SQLite uses caching in order to speed up the SELECT queries. If a query only involves cached pages, it can execute much faster since no disk access is required. As in the previous two systems, a SQLite C API implements Prepared Statements. That system, described in the previous sections, provides the opportunity to insert or read multiple data sending in a more flexible and secure way. Despite implementation of prepared statements, the speed in the inserting queries is not improved. Only the security and the usability of the system are improved.

The MongoDB approach is completely different to that of SQL databases. Each document that is inserted in MongoDB could have a different structure, so, prepare statements make no sense here. MongoDB keeps all of the most recently used data in RAM. If you have created indices for your queries and your working data set fits in RAM, MongoDB serves all queries from memory.

2.3.2.2 Performance Tests

There are several benchmarks and experiments that compare between MySQL, PostgreSQL and SQLite, but the most reliable experiment is to reproduce the most common interactions between the CB system and the three kinds of database. The most common queries that will be executed in our scenarios are:

- INSERT: the Repository saves data from the sensors in the database. This query is used most often.
- SELECT by TimeStamp: some agent wants the data from a certain past time interval.
- SELECT by NodeID: the agent is interested only in a specific node.
- SELECT by Threshold: The agent only gets the data that satisfies a threshold.

In the following we report on the results of several experiments, one per each possible query and three in order to test the behavior with threads and multi-access. The tests try to mimic the usage patterns that we envision in the CREW scenarios, i.e. we utilize actual spectrum sensing data structures. The experiments have been prepared to have the same characteristics, including a database with the same data, the same host, the same host load and the same programming language, C. The database is composed of 100 rows with 19 columns in the case of SQL and 100 entries in MongoDB. Each column has an integer data. The node id and the timestamp columns are filled with controlled values, meanwhile the last 16 columns have random values between 0 and 80. Each experiment has been repeated 100 times. The experiments include the connection and the close time in each iteration.

2.3.2.2.1 Test 1: Insert

This experiment shows how a client can introduce different numbers of rows with different numbers of queries. This test measures the speed of inserting data and the efficiency of including multiple data in each query. The results are shown in the following table and visualized in the figure thereafter.

	100 rows 100 insert [us]	100 rows 1 insert [us]	1000 rows 1 insert [us]	10000 rows 1 insert [us]
mysql	840,3147	561,9536	104,44588	51,385199
sqlite	132942,772	1666,5772		
postgresql	730,7761	763,5081	123,32322	40,75623
mongodb	236,0683	48,8913	17,96169	16,304648

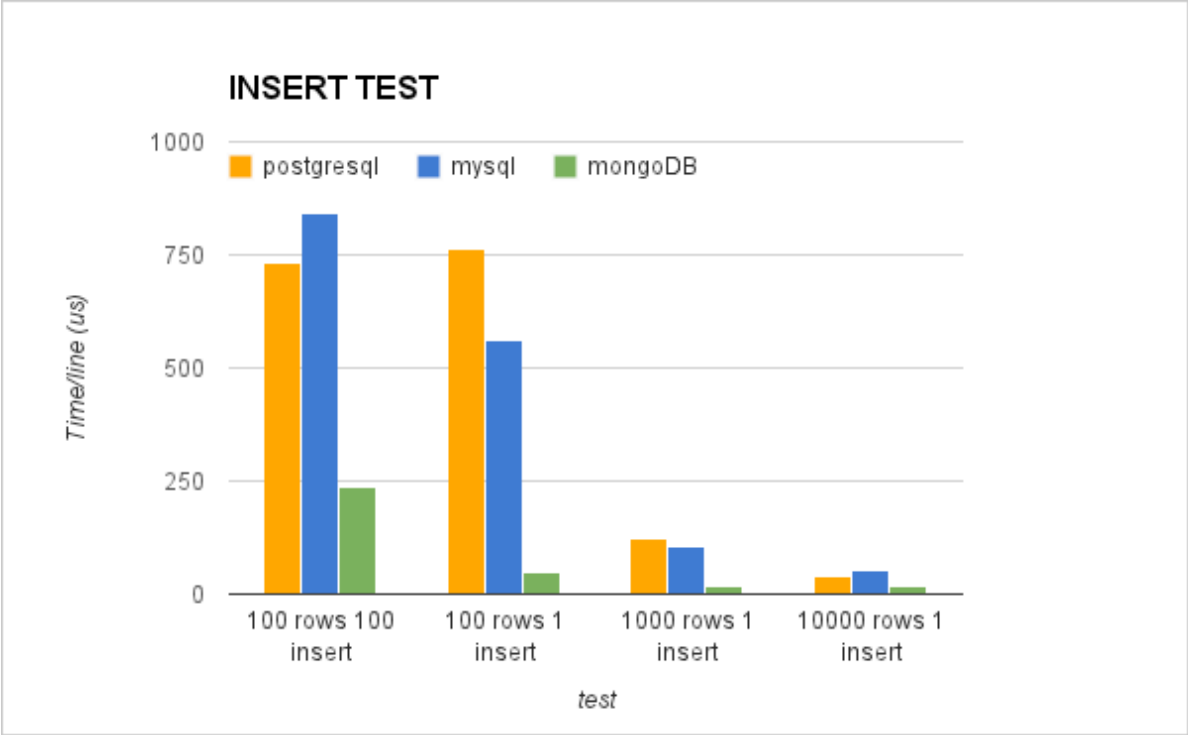


Figure 14: Insert test results. Comparison among PostgreSQL, MySQL and MongoDB

As we can see in the table, the introduction of multiple data in an INSERT sentence reduces the time for row by three when more than 1000 rows are included at the same time. However this requires more memory and a higher refresh latency. For example, if we have 100 sensors that each send data once a second, we only refresh the database once every 10 seconds. Depending on the application, the maximum buffer size should be configured in order to balance latency with the speed of insertion.

The second conclusion that can be extracted is that SQLite performs poorly compared with the rest of the databases. The performance of MySQL and PostgreSQL is quite similar and the best one is MongoDB. Inserting a batch of 100 objects in MongoDB is 10 times faster than inserting a query of 100 objects in MySQL. Three additional important factors that have been found during this test are:

1. PostgreSQL with the buffer of 10000 lines consumes lot of CPU and results in failures in the execution.
2. SQLite with the buffer of 1000 and 10000 lines results in errors. SQLite INSERT queries do not support natively more than 1 row per INSERT but it can be done using others commands, but only to 100 rows.
3. The use of prepare statements has been completely discarded because of the execution time. The insertion of 100 rows in 100 INSERT queries using prepare statements takes 41,6ms per row instead of 840us using normal string queries. This indicates that the problem with the insertion speed was caused by the prepare statements.

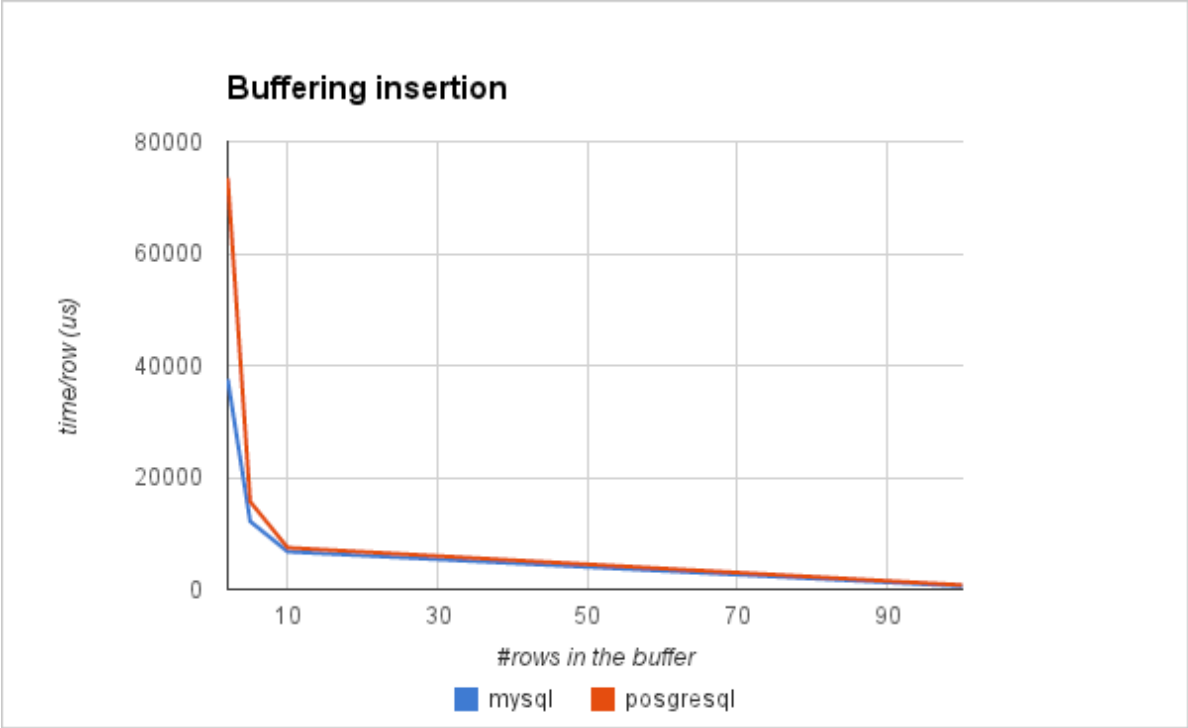


Figure 15: Buffering insertion. Comparison between PostgreSQL and MySQL

The previous figure shows the time that a client needs to insert a row when multiple rows are combined in one insert. The y axis represents the time per row in microseconds. As can be seen, with only with 5 or 10 rows in the same query the time is reduced by more than a 75%.

2.3.2.2.2 Test 2-4: Select

In order to test the performance of reading data from the database we performed the following tests:

Test 2: In this test, the clients request the data that satisfies a query of the type "Time > Threshold, i.e. reading the inserted data from an specific time. In our case the query should return 10 rows. The test consists of executing the SELECT query and accessing the data of each row. The code saves each data in a int variable. This test has been design in this way in order to compare the speed of SELECT queries but also the speed of access to the data returned.

Test 3: Select by time and node ID. The third test executes a SELECT query with two different conditions: a threshold time and a specific node ID. In this case, the database must return 2 lines. As in test 2, the application accesses the data of each row.

Test 4: Select by threshold. The difference between this example and the previous one is that the clients request data over a threshold but these data are random. This forces the database to make a more complex search, ignoring possible keys.

The results for these tests are summarized in the following table / figure.

	Test 2 [us]	Test 3 [us]	Test 4 [us]
mySQL	455,39 (148,84)	355,97 (121,96)	666,13 (293,49)

postgreSQL	70765,9 (2516,62)	65402,18 (1568,05)	66973,1 (5104,49)
SQLite	388,95 (299,01)	375,8 (273,89)	638,41 (631,80)
mongoDB	1017,36 (513,22)	1020,19 (507,89)	1135,3 (656,54)

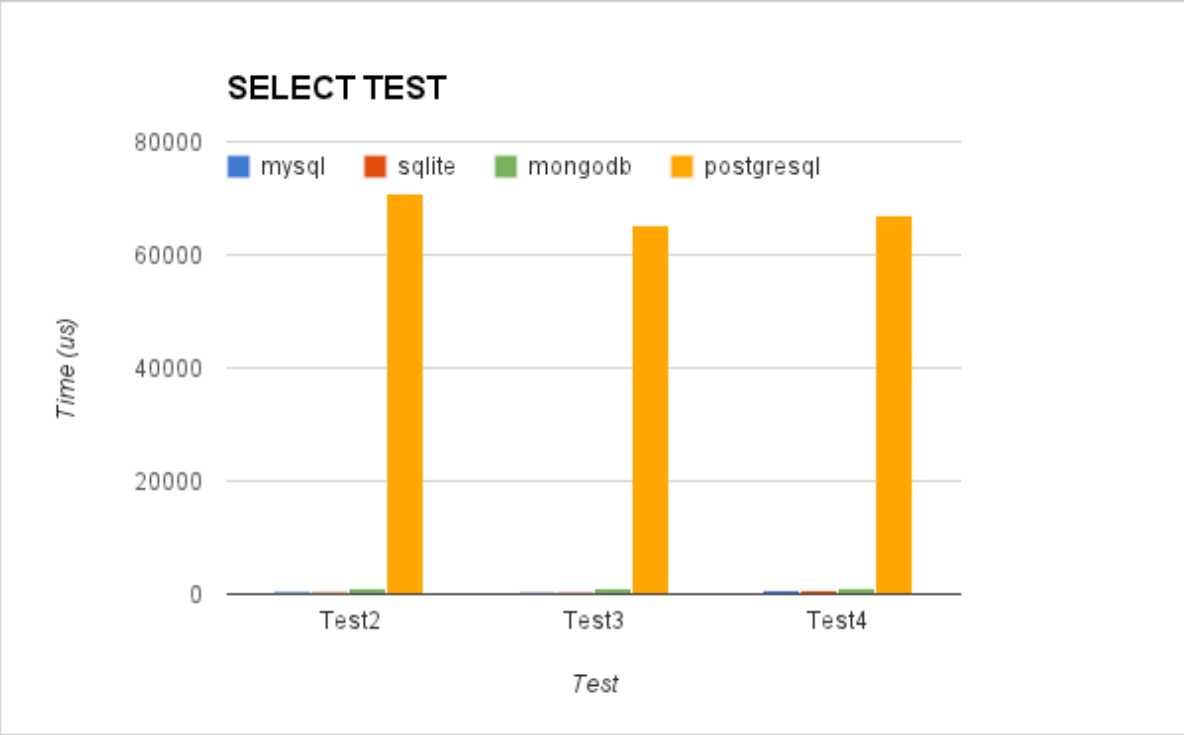


Figure 16: Select tests results. Comparison among PostgreSQL, MySQL, SQLite and MongoDB

The first figure represents the three performance results for MySQL, SQLite, PostgreSQL and MongoDB. The y axis represents the time in us to execute the SELECT sentence and to access each element of data.

The performance of PostgreSQL in all the tests is significantly worse than the performance of MySQL, SQLite or MongoDB. The reason is the overhead cost required to connect to the database. It is important to remember that each experiment includes the connection and the disconnection time. If this time is not included, the performance of PostgreSQL improves (results shown in previous table between brackets), but it still is worse than the other databases. The following figure allows to better understand the difference between the other three databases:

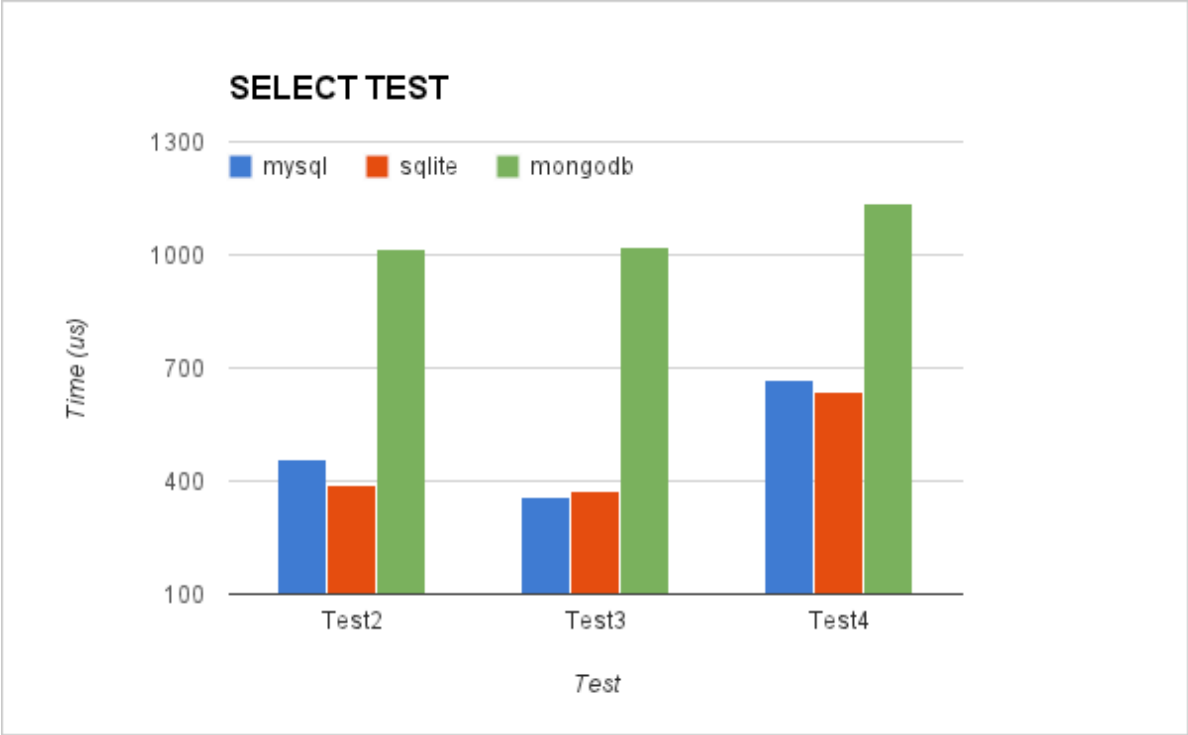


Figure 17: Select tests results. Comparison among MySQL, SQLite and MongoDB

The performance for MySQL and SQLite in all the tests is very similar. If we compare the results without including the connection time, MySQL is twice as fast as SQLite. MongoDB has good performance too, but it still needs double the time to consume data.

2.3.2.2.3 Test 5: Insert - concurrent access

These tests have been developed in order to test the performance of the database under multiple access. Each case creates different number of threads that inserts rows in the database:

- Case 1 - 10 threads and 100 rows each thread
- Case 2 - 100 threads and 100 rows each thread
- Case 3 - 100 threads and 1000 rows each thread
- Case 4 - 1000 threads and 1000 rows each thread

In the following figure the four thread-based test results are shown. The y axis represents the average time that a thread needs to insert the corresponding number of rows.

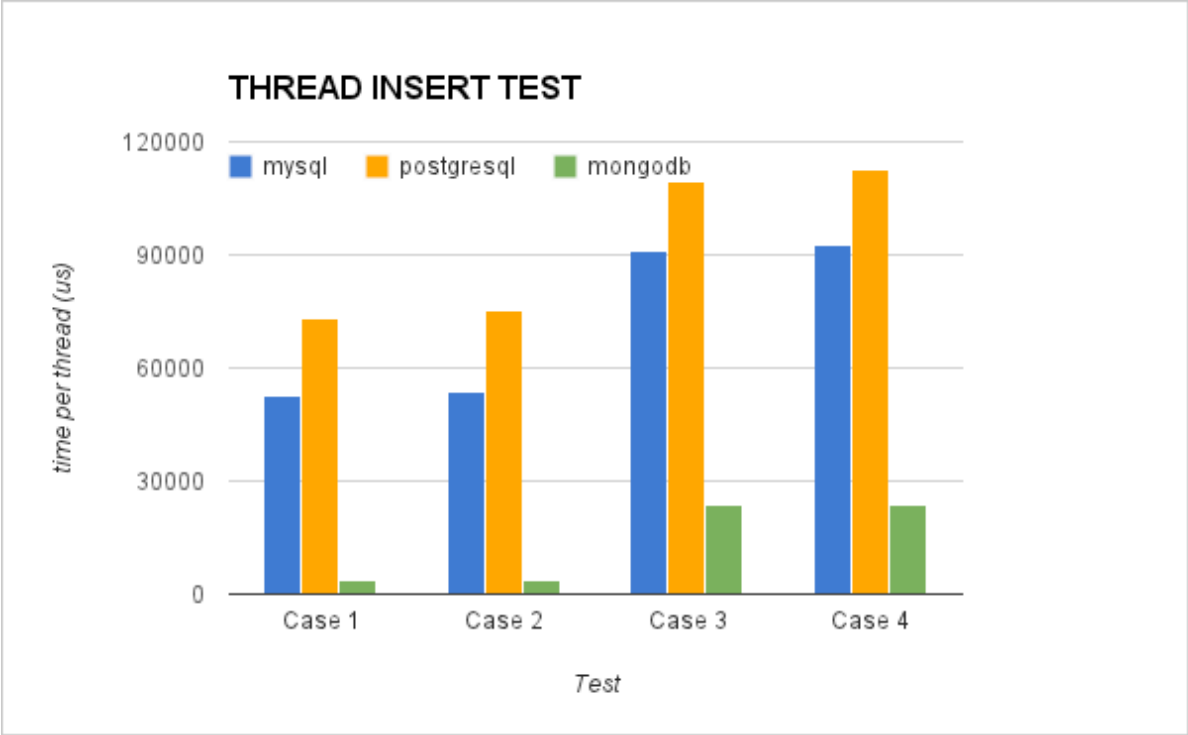


Figure 18: Insert tests with threads results. Comparison among PostgreSQL, MySQL and MongoDB

The results show that SQLite has the worst performance. They are excluded from Figure 18, as they would prevent comparison of the other solutions. Due to this reason it will also be excluded from the further investigations. The reason is again that SQLite blocks the file where the database is stored each time that a thread writes. MongoDB is the fastest database in the tests but MySQL and PostgreSQL have a good performance, too. This test shows that the number of objects in the batch affects the time more than the number of threads.

2.3.2.2.4 Test 6: Select - concurrent access

The test 6 has the same goal as the previous test, but in this case the threads are consumers. Each thread sends a query to the database in order to get the last 10 data elements saved. The results are summarized in the following figure.

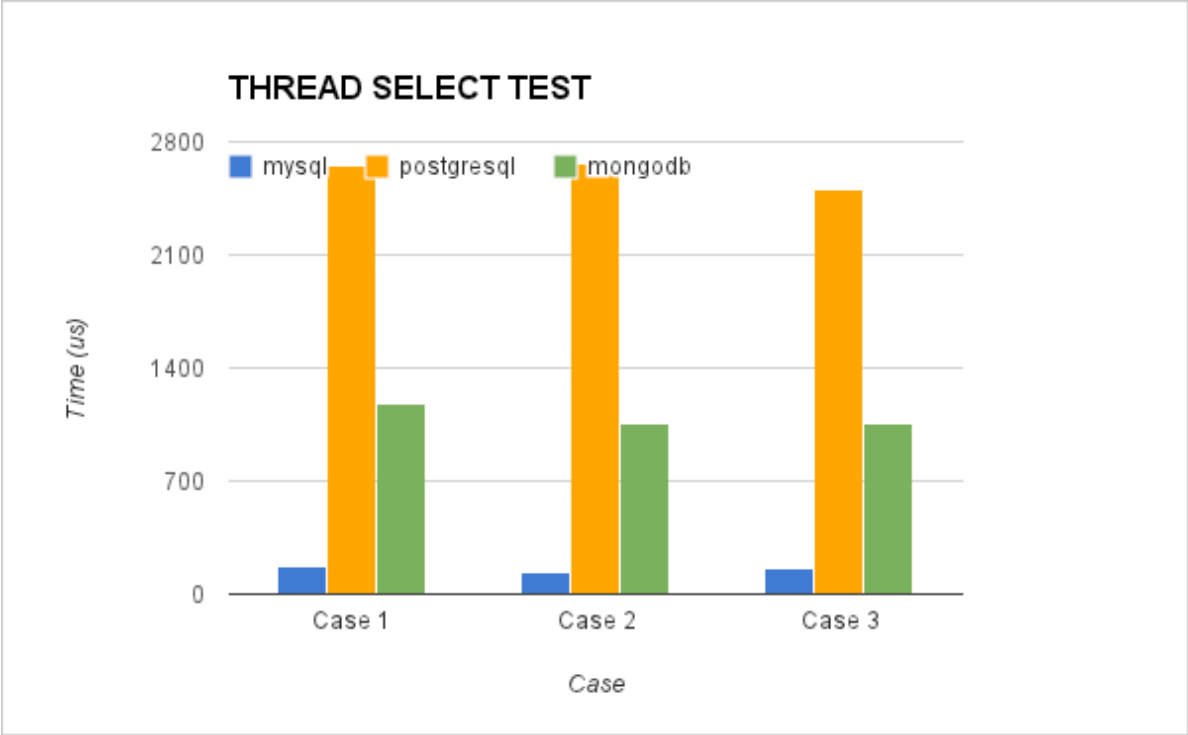


Figure 19: Select tests with threads results. Comparison among PostgreSQL, MySQL and MongoDB

MySQL is again the quickest when the application executes SELECT queries. MongoDB takes more time, but it still obtains reasonable results. Finally, PostgreSQL takes more time than the other databases.

2.3.2.2.5 Test 7: Insert/Select - concurrent access

The last test tries to generate a common situation in the database, where multiple connections (producers and consumers) access the database at the same time. The test creates 10 producers and 10 consumers. Each producer connects to the database 100 times and each time saves 100 rows or objects. The consumers connect to the database 100 times and request the last 10 rows or objects.

The test is executed in two variants ("cases"). The first case creates all the producers first and then the consumers. The second case creates alternatively (interleaved) a producer and then a consumer, etc. The results are shown in the following figure.

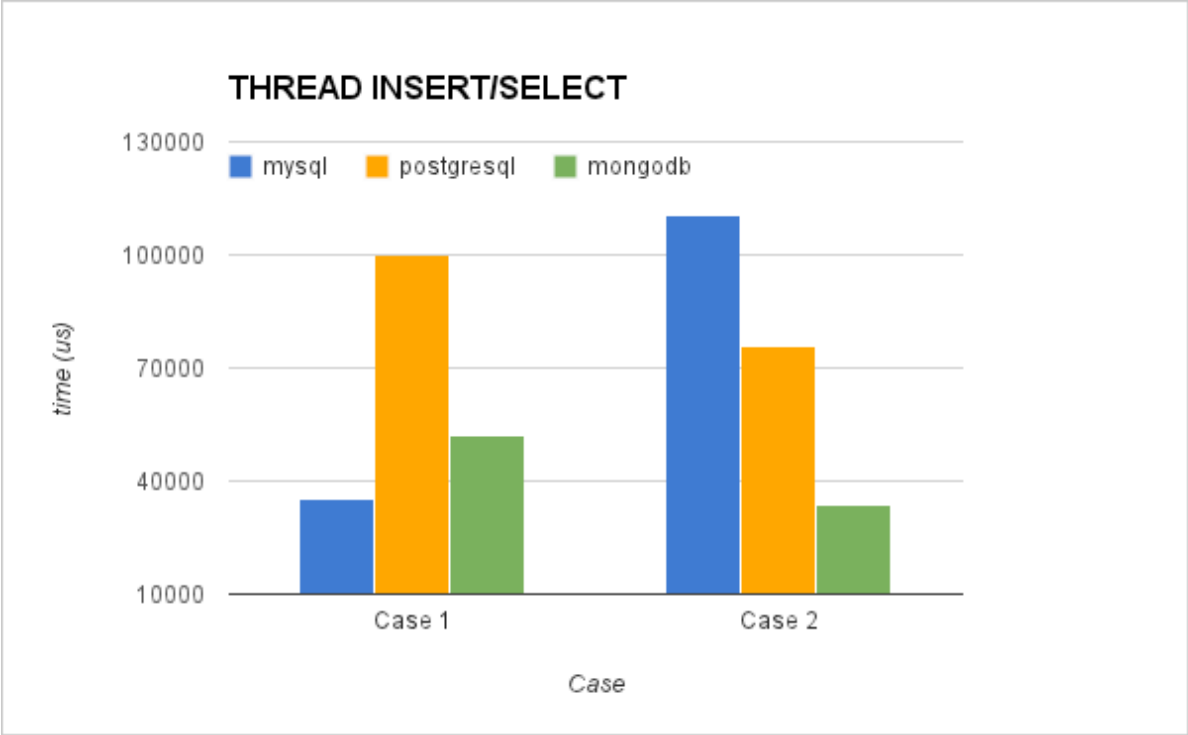


Figure 20: Multiple thread tests results. Comparison among PostgreSQL, MySQL and MongoDB

The results show that MySQL is the best option when producers and the consumers access at different times. However, when the producers and consumers operate interleaved MongoDB has the best performance.

2.3.2.2.6 Usability

The use of a C library and the installation of the three SQL technologies was simple, efficient and intuitive. The installation and the use of MongoDB are also simple, but this database uses a completely different usage model. If the user is used to SQL languages, the first steps may be slower. Our usability experiences are summarized in the following table.

	Strengths	Weaknesses
MySQL	Simple fastest SELECT queries Most used Well known by developers	Simultaneous producers and consumers few data types supported
PostgreSQL	Scalability Complex features	Speed CPU usage with high buffers
MongoDB	Speed INSERT queries Flexibility	Less used Not well known

2.3.2.2.7 Conclusions from the Performance Tests

We have presented the investigation and a benchmark among three well known databases. The test has been modeled according to the requirements of the CB system in CREW scenarios and is biased to those requirements. Other benchmarks try to assess the performance of the databases in a general and usage independent way. We have showed a pretty specific type of use where the multiple access to the newest data in the database is very important. According to the results the use of SQLite should be discarded because of many restrictions: it does not support users and passwords, does not support remote connections and does not support multiple insertions in a single query.

MySQL and PostgreSQL have a lot of features in common. The performance in many tests has also been similar, but there are a few differences in the SELECT tests, where MySQL shows better results. MongoDB often features superior results in comparison with MySQL and PostgreSQL, but the selection of "best database" is test-specific.

According to our results we can determine the scenarios where the use of each database is indicated:

1. **Scenario with more producers than consumers**. In this scenario the use of **MongoDB** is recommended, because MongoDB shows a very good performance when inserting data.
2. **Scenario with more consumers than producers**. Here, **MySQL** is the best solution. MySQL SELECT queries have had the best results and the SQL language is easier than MongoDB in order to build SELECT queries.
3. **Complex scenario**. When the scenario requires high requirements, a high number of data and complex features there is some indications that **PostgreSQL** *may* be the optimum solution. However, we have not been able to run very large-scale tests which show the threshold when PostgreSQL starts to be more efficient than MongoDB or MySQL.

Since in the CREW scenarios we envision many data producers (spectrum sensing devices) in our implementation we have opted for MongoDB. This database is scalable, high-performance, open source and NoSQL. MongoDB uses a database and collections in order to save the entries. The collection can be compared with the tables in MySQL. The basic elements of information in MongoDB are the BSON objects. The BSON objects are basically binary JSON objects. Each BSON object includes some key-value pairs, similar to a row in MySQL. The next code represents an example of BSON object showing an entry of spectrum sensing data collected with the TelosB platform:

```
{ "_id" : ObjectId("50ebf4b245c03d4f2f12a230"), "node_id" : 1,
"TimeStampSec" : 1357640882, "TimeStampMicroSec" : 297237,
"ch11" : -93, "ch12" : -92, "ch13" : -86, "ch14" : -93, "ch15"
: -92, "ch16" : -93, "ch17" : -92, "ch18" : -93, "ch19" : -94,
"ch20" : -94, "ch21" : -82, "ch22" : -93, "ch23" : -73, "ch24"
: -88, "ch25" : -70, "ch26" : -93, "capabilitiesid" : 1,
"profileid" : 1 }
```

In order to migrate our previous database implementation from MySQL to MongoDB, multiple functions had been implemented. Some are similar, for example, the function `retrieve_rssi_by_timestamp()`, which retrieves the data that fills a time restriction. In addition, other functions have been developed in order to support the metadata included in the database. The metadata are explained in the next section.

2.3.3 Metadata and Discovery Functions

Our first implementation of the CB framework lacked some important information and functions: the CAgent saved the RSSI data in a table or collection but for external clients it was not possible to query / discover the structure (data types) in another CAgent's repository. If an external CAgent wanted information from a node it would have to know the structure of the database a priori. In the third year

of the CREW project we solved this problem by introducing a well-defined set of metadata and API that CAgents can use to discover each other's repository content.

The new data and functions have been selected from the IEEE 1900.6 standard that specifies procedures, protocols and message format specifications for the exchange of sensing related data, control data and configuration data between spectrum sensors and their clients. The standard is very extensive and only two primitives have been adopted. However, these two primitives include most of the relevant information for a spectrum sensing application. The two primitives define the API to access the metadata. Moreover, IEEE 1900.6 also defines the data types of the information included in the metadata.

The first primitive is used to request information about the spectrum measurement capabilities. These capabilities are static and associated with a platform. This primitive returns information about the frequency range, the sensing mode, frequency resolution, sweep time, etc. The first primitive includes the following request() and response() function pair:

```
Get_Supported_Spectrum_Measurement_Description.request
      (MeasurementCapabilityTransID)
Get_Supported_Spectrum_Measurement_Description.response
      (MeasurementCapabilityTransID, Status, MeasuRange,
      SensingMode, DataSheet.ADDAResolution,
      DataSheet.AngleResolution, DataSheet.FrequencyResolution,
      DataSheet.LocationTimeCapability,
      DataSheet.LoggingFunctions,
      DataSheet.RecordingCapability, DataSheet.SweepTime,
      LockStatus)
```

The second primitive is related to the configuration of the platform. This type of configuration is variable, it is specific for a particular experiment but may change between different experiments. The primitive retrieves information like the confidence level, sensing mode, report rate, time stamps, lower threshold, measure bandwidth, etc. The second primitive includes the following request() and response() function pair:

```
Get_Sensor_Measurement_Profile.request
      (SensorMeasurementProfileID)
Get_Sensor_Measurement_Profile.response
      (SensorMeasurementProfileID, Status, ConfidenceLevel,
      ReportMode, SensingMode, ReportRate, TimeStamp,
      Scan.LowerThreshold, MeasuBandwidth, LockStatus)
```

For both primitives we have implemented MongoDB functions and respective protobuffers that support the request (recall that we use Google protocol buffers for serialization). The protobuffers are filled with the respective information. The google protobuffers designed for the primitives are used as in the following example:

```
message CAgentResMessage {
  optional int32 cagentsrcid = 1;
  optional int32 cagentdestid = 2;
  optional CAgentResType type = 3;
```

```
    enum CAgentResType {
     MEASUREMENT_CAPABILITIES = 0;
     MEASUREMENT_PROFILE = 1;
    }
    optional MeasurementCapabilities mcapabilities = 4;
    optional MeasurementProfile mprofile = 5;
}
```

The next message sequence diagram visualizes a common operation of a new CAgent when it wants to filter the information, i.e. a CAgent selecting the data from some platform or configuration.

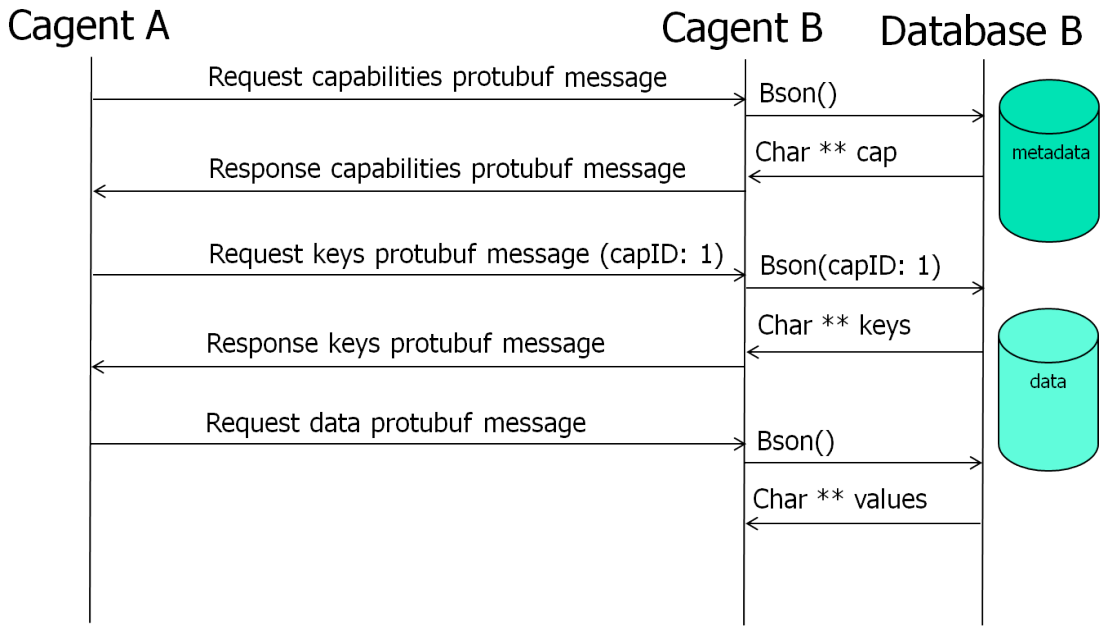


Figure 21: Example of the metadata access method.

The CAgent requests the available capabilities in the metadata. It first receives a list of different capability IDs and their properties. With this information the CAgent can find out which parameters the platform has and what the capabilities of that CAgent are. Afterwards the CAgent has all the information necessary to select the data it wants in a subsequent query (not shown).

2.3.4 Summary

In the third year of the project the collaboration with the Berkeley Wireless Research Center (BWRC) on prototyping the Connectivity Brokerage framework has continued. The main outcome is summarized in the following:

- We redesigned our software framework: the CB framework implementation is now decoupled from the scenario and can thus be instantiated more easily in different experiments.
- We have performed a detailed performance analysis of various databases to better understand their suitability for a spectrum sensing repository and extracted guidelines that help in selecting a database for a specific CR scenario.

- For our internal use case MongoDB has been selected and integrated with our software framework.
- We have adopted parts of the IEEE 1900.6 standard to extend the CB framework with metadata and provide an API that allows discovery of repository content among CAgents.

2.3.5 Hardware specific implementations

2.3.5.1 CB implementation on w-iLab.t

We further extend the CB implementation in a simplified manner for different sensing platforms, more specifically the USRP and IMEC sensing engine. Those extensions are tested on the w-iLab.t testbed. Within the context of CB, the extension focuses on the virtual control channel functionality, by utilizing Google protocol buffer and ZMQ libraries. Two types of CAgents are implemented: the USRPCAgent and the IMECCAgent. Furthermore, dedicated applications on an individual NodeCAgent are developed to query sensing information from the USRPCAgent and the IMECCAgent respectively. The USRPCAgent is directly coupled to the USRP hardware and the same goes for IMECCAgent., which produces sensing data. NodeCAgent has no direct link to the sensing hardware, it queries sensing data from the corresponding sensing engine CAgent via the virtual control channel.

2.3.5.1.1 Define the message structure

Two types of messages are defined on the virtual control channel, one contains the configuration of the sensing engine and one contains the measurement result. We defined those two types via the Google protocol buffer. First, a very concise ".proto" file is defined, which has exactly what item the message needs to contain. Secondly, based on the ".proto" file, Google protocol buffer generates more comprehensive ".cc" and ".hp" file. When opening the generated files, we see that every message we defined in the ".proto" file is a corresponding class in the ".cc" file, which not only contains all the message fields, but also the functions to prepare the message and interpret the message. The relevant part in the ".proto" file of the IMEC sensing engine and USRP are shown in Figure 22 and Figure 23 respectively. Both sensing engines have the "se_mode" in its configuration message. This field is used to specify the type of channel the sensing engine should be configured to measure, for example to choose between Wi-Fi channel and Zigbee channel. A couple of fields ("first_channel", "last_channel") are used to specify the request channel range. And a couple of fields, like the "fe_gain", "bandwidth", are used to configure the radio front-end. Apart from the comment fields, USRP sensing engine has the "DetectMode" field, to choose between maxhold, averaging or minhold; IMEC sensing engine has two output formats, you can choose to read the raw output or the output converted to dBm.

```
/*-------------------------------------------------------------------------*/

message Configuration {
  // Keep values in sync with the enumeration 'se_mode_t' defined in 'sensing.h'
  enum SeMode{
    FFT_SWEEP          = 0;
    WLAN_G             = 1;
    WLAN_A             = 2;
    BLUETOOTH          = 3;
    ZIGBEE             = 4;
    LTE                = 5;
    DVB_T              = 6;
    ISM_POWER_DETECT   = 7;
    TRANSMIT           = 97;
    ADC_LOG1           = 98;
    ADC_LOG2           = 99;
    STDBY              = 100;
  }
  enum OutputFormat {
        RAW = 0;
        DBM = 1;
  }
  required SeMode      se_mode              =   1;
  optional uint32      fe_gain              =   2;
  optional uint32      first_channel        =   3;
  optional uint32      last_channel         =   4;
  optional uint32      bandwidth            =   5;
  optional uint32      fft_points           =   6;
  optional uint32      dvb_nr_carriers      =   7;
  optional float       dvb_guard_interval   =   8;
  optional float       threshold_power      =   9 [default = 10.5];
  optional OutputFormat output_format       =   10;
}
```

Figure 22 The configuration message for IMEC sensing engine

```
message Configuration {
  enum SeMode{
//    FFT_SWEEP          = 0;
    WLAN_G             = 1;
    WLAN_A             = 2;
    BLUETOOTH          = 3;
    ZIGBEE             = 4;

  }

  enum DetectMode{
        AVERAGE = 1;
        MAXHOLD = 2;
        MINHOLD = 3;
  }
  required SeMode      se_mode              =   1;
  optional uint32      fe_gain              =   2;
  optional uint32      first_channel        =   3;
  optional uint32      last_channel         =   4;
  optional uint32      fft_points           =   5;
  optional string      usrpip               =   6;
  optional uint32      spb                  =   7;
  required DetectMode  det_mode                      =   8;
}
```

Figure 23 The configuration message for USRP

2.3.5.1.2 TestbedCAgent

When looking at the different roles of the two CAgents, it is clear that the TestbedCAgent needs to listen to the request from the NodeCAgents. Therefore the TestbedCAgent is implemented as a server, which contains a loop to wait and process incoming requests. The detailed structure of TestbedCAgent

is illustrated in Figure 24. Before entering this loop, there are a few things to set up. Upon starting up, the IMECCAgent establishes a connection with its sensing hardware. In case of the IMEC sensing engine, this means we first configure the the SPIDER platform (loading the firmware for the USB interface chip and configuring the FPGA on the device) and then load the DIFFS chip with its firmware. In case of USRP, the server tries to establish connection with USRP and allocate buffers for receiving samples. Apart from the device-specific hardware initialization, the CAgent also initialized the ZMQ libraries to ensure that the socket is successfully created and binded. For Google protocol buffer we need to do a version control to make sure that the binary version used to generate the ".cc" and ".h" files are exactly the same as the library version we are using for serializing and de-serializing the message. Finally a stop handler is registered. In case of emergency shutdown, the stop handler makes sure that the sensing engine hardware is shut down properly as well.

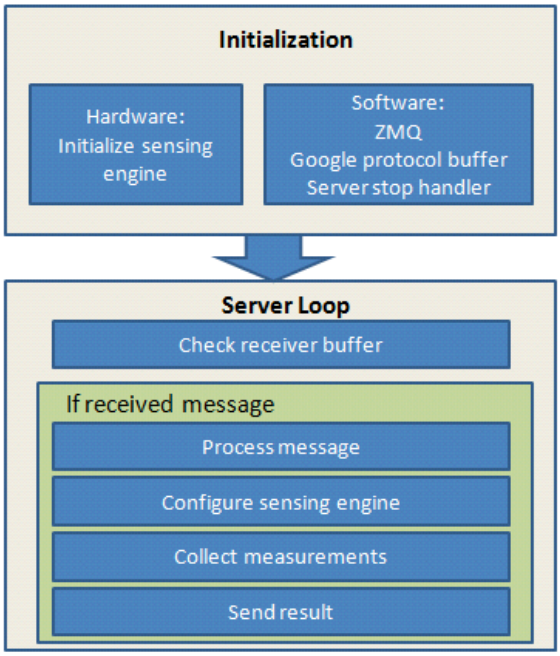


Figure 24 TestbedCAgent diagram

After the initialization phase, the program enters a loop. It keeps on polling the receiving buffer until a message is present. Then it will parse the message to understand what kind of measurement is required and configure the sensing engine to perform measurements. Finally the collected result is sent back to the client.

2.3.5.1.3 The sensing query applications on the NodeCAgent

The application to query sensing information on a regular NodeCAgent is less complex than the sensing engine CAgent. It has a couple of input options used to describe the configurations to be passed tothe sensing engine. The most important option is the "-server", to specify the address and port of the sensing engine CAgent. The rest of the options are used to configure the sensing engine, for instance, it can choose the "se_mode" among different wireless standards; it can fine tune the sensing engine's sensitivity via the "fe_gain" option, etc. As an example, the help menu of the NodeCAgent application to query from the USRP sensing engine is shown in Figure 25. The message structure used in the NodeCAgent is exactly the same as in the sensing engine CAgent, see the previous section for more information.

```
liu@craig:~/Documents/workspace/usrpcb/build$ ./usrpse_cb_client -help
linux; GNU C++ version 4.4.3; Boost_104000; UHD_003.005.003-78-g49a4929b

Usage:

        $ ./usrpse_cb_client [OPTIONS]

        Options are:
                -help
                        Print this help page.
                -level LEVEL
                        Print messages of specified and lower level.
                          Level 0: No messages, except messages from the UHD library.
                          Level 1: Panic/Error messages.
                          Level 2: Warning messages.
                          Level 3: Information messages (default).
                          Level 4: Logging messages.
                          Level 5: Verbose messages.
                -silent
                        Disable printing messages.
                -server
                        Server address of the sensing engine server.
                          ZeroMQ format: protocol://host:port
                          Default:       tcp://se-server:3607
                -output
                        Output filename storing the measurement data.
                          Default: ./data.out

                -mode MODE
                        Sensing mode. Supported modes with arguments are:
                          WLAN_G    [fe_gain,first_channel,last_channel]
                          WLAN_A    [fe_gain,first_channel,last_channel]
                          BLUETOOTH [fe_gain,first_channel,last_channel]
                          ZIGBEE    [fe_gain,first_channel,last_channel]
                -fe_gain GAIN
                -first_channel CHANNEL
                -last_channel CHANNEL
                -fft_points FFT size, must be a positive integer and a power of 2
                -detectmode DETECT_MODE
                        The detect mode:
                          1: Average
                          2: Maxhold
                          3: Minhold
```

Figure 25 The help menu to query from USRPCAgent

2.3.5.2 CB implementation on Log-a-TEC

In this section we report on our investigation about the feasibility to implement the CB framework for LOG-a-TEC.

The cognitive radio experimentation part of the LOG-a-TEC testbeds has the following main components:

- The cognitive radio experimentation hardware in the form of the VESNA node with the SNE-ISMTV extension installed outdoor on the public light poles
- The Lightweight Client Server Protocol (LCSP) which is a protocol for controlling and retrieving data from the hardware nodes.
- The Python and JS tools that are wrapped around the testbed functionality.

Additionally, all the meta-data about the nodes is managed by the Sensor Management System (SMS) that also periodically polls all the nodes for monitoring purposes. The SMS uses MongoDB and NodeJS, an event-driven I/O server-side JavaScript environment, for interacting with the database and the nodes.

Comparing this basic infrastructure and functionality offered by LOG-a-TEC with the CB framework, it can be seen that the Repository and Discovery functionality is already present, however the abstractions and implementation are different than the approach taken with the CB. For instance, the CB relies on primitives from the IEEE 1900.6 standard for discovery while the API used by LOG-a-TEC is less biased by IEEE wireless standards since it covers a broader scope of sensor descriptions (including energy consumption and air quality monitoring). Additionally, in the CREW-GENI collaboration an ontology based on the CREW common data format (CDF see CREW D3.1) that will be used for device meta-data description for spectrum sensing experiments is being developed. With respect to the Optimizer and Execution part of the CB, LOG-a-TEC already has implementations of power allocation games based on the Python tools and the LCSP protocol.

Due to several parallel developments within the CB framework, the LOG-a-TEC testbed and the CREW-GENI collaboration during the 3rd year of the CREW project it has been hard to take a final decision on whether the CB framework is suitable for LOG-a-TEC. This investigation will be continued during the 4th year of the project.

2.4 Modular protocol architecture

We have recently proposed [10] a framework that allows composing communication services in a dynamic way. The framework is based on a fine-grained modular protocol stack architecture. The reference implementation of the framework is ProtoStack[2] while the reference implementation of the modular protocol stack architecture is Composable Rime[3] (CRime).

2.4.1 The components of the proposed framework

The overall framework has four functional components: the physical testbed, the module library, the declarative language, and the workbench as depicted in Figure 26.

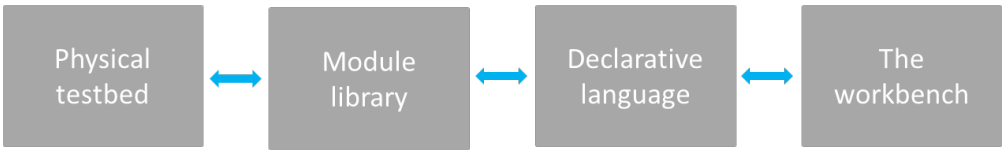


Figure 26 The four components of the framework for the composition of services.

The physical testbed

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

The module library

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

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.

[2] https://github.com/sensorlab/ProtoStack

[3] https://github.com/sensorlab/CRime

A translation tool is employed to translate from the workbench's elements to the declarative language. In some cases, the user may want to bypass the user interface and directly use the language for describing and configuring the modules in a stack prior to the experiment. As a consequence, the language typically also needs to be human readable, possibly easy to learn and should use intuitive code words.

The workbench

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

2.4.2 Requirements for the proposed framework

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

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

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

2.4.3 Reference implementation: the ProtoStack tool

In this section we briefly introduce a reference implementation of the framework for the dynamic composition of services called ProtoStack. We discuss the implementation of the framework and we identify the communities which may find such a tool useful.

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

i) an advanced user such as the component developer needs to focus on developing the component and make it work with Contiki[4] and

ii) a novice user needs only to focus on composing services in a stack using the workbench.

The physical testbed is based on VESNA sensor network platform[5] to which the Contiki OS has been ported, partly due to the adaptive Rime architecture which comes with it [12]. This physical testbed posed constraints that determined the selection of Composeable Rime (CRime) as the module library.

The CRime module library in the reference implementation is a purpose-built set of protocols influenced by and based on the Rime [12] architecture. The declarative language we selected for ProtoStack is based on the Resource Description Framework (RDF)[6], a standard semantic web language. The implementation uses the Turtle[7] syntax together with existing standardized vocabulary and a custom ontology. The workbench is tightly integrated with the language and is implemented using WireIt[8], an open source javascript library which enables the creation of full web graph editors.

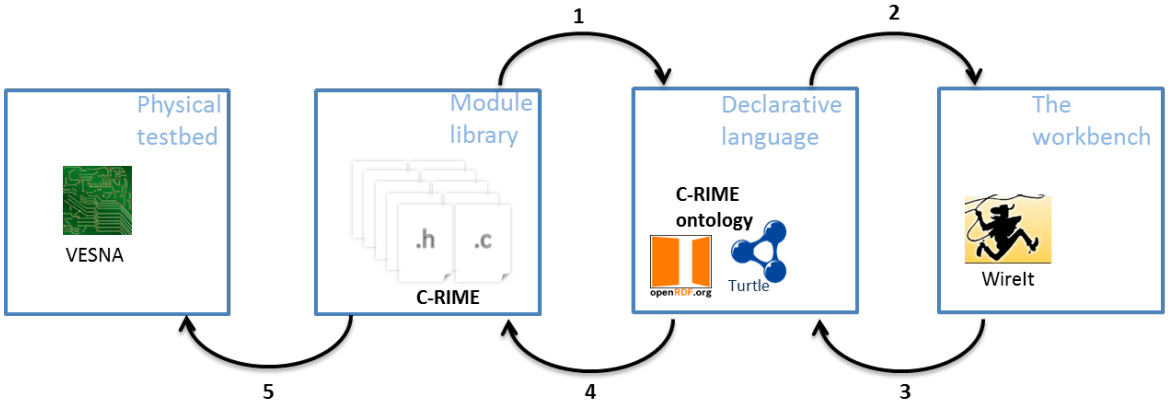


Figure 27 ProtoStack: an implementation of the framework for composing communication services. The implementation addresses the sensor networks domain.

In Figure 27 we illustrate the steps for dynamic composition of services using the ProtoStack tool. The component developer develops a module, manually tests it and makes sure everything works as intended, and, at the end he/she needs to write few lines of Turtle statements (i.e. triples) which specify basic characteristics of the new 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 27). 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 27).

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 27). 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 27). 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 27).

[4] http://www.contiki-os.org/

[5] http://sensorlab.ijs.si/hardware.html

[6] http://www.w3.org/TR/PR-rdf-syntax/

[7] http://www.w3.org/TeamSubmission/turtle/

[8] http://neyric.github.com/wireit/

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

2.4.4 CRime abstractions

CRime is a new architecture designed to support the composition of communication services which is inspired by and built upon the Rime [12] architecture. CRime introduces three abstractions the *amodule*, the *pipe* and the *stack*.

The *amodule* (short from abstract module) is a generic building block of the CRime stack. Behind each instance of an amodule hides a communication service [13] such as broadcast or multihop. The communication service is an implementation of a network function such as protocol or algorithm and contains only the execution logic of that function. Several amodule instances can be arranged in a pipeline to form a communication stack.

The *pipe* is a vertical structure which can be accessed by any of the modules in a composed stack. The pipe contains only data structures corresponding to parameters that are used by the stack. Pipes are uniquely identified by the channel number they are assigned to, therefore a single channel can only have one associated pipe at a time. This implementation, while not the most efficient approach from a software engineer's perspective, nor the most resource efficient in terms of memory, instantiates the concept of vertical layer and is the first building block in the implementation of the knowledge plane required for experimentation with cognitive networks. The approach is also a compromise between memory footprint and the complexity required by the autogenerated C code based on user input.

The *stack* is a structure which contains a meaningful sequence of amodules and a pipe. It behaves as a container for these elements and enables the composition of more complex communication services which use more than a single channel at a time. Using the stack abstraction, an independent communication stack can reside on each channel. These stacks merge at the application layer or below it. Figure 28 depicts the three abstractions in an example of a 1 channel - 1 stack and an example of a 3 channel – 3 stack communication system.

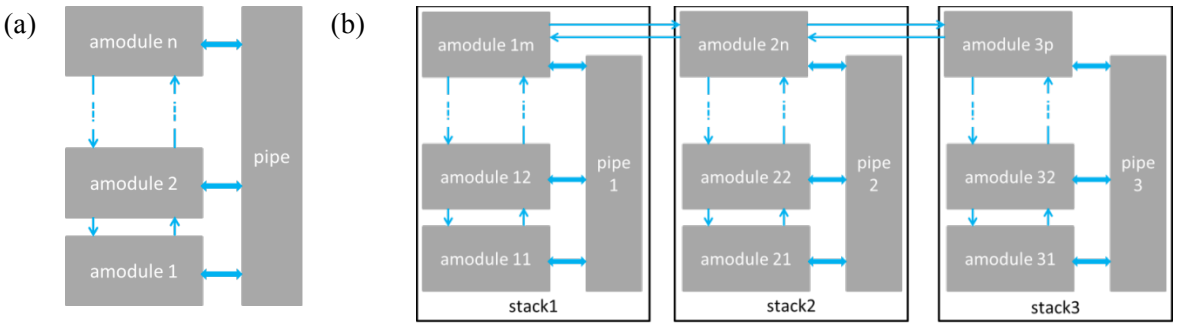


Figure 28 Example of CRime stacks: (a) 1 channel – 1 stack example and (b) 3 channel – 3 stack.

2.4.1 The cost of composeability

Rime was one of the first communication architectures for sensor networks to provide a set of abstractions in order to support adaptive communication. Compared to non-adaptive architectures, Rime incurred higher execution time [12]. CRime is, to the best of our knowledge, the first architecture that supports dynamic composition of services for sensor networks. Through dynamic composition of services, CRime helps to speed up the design, prototype and evaluation of new communication services. Compared to Rime, CRime introduces new abstractions which reflect in overhead in terms of code size, execution time and power consumption. In this section we assess the costs that CRime's overhead introduces versus Rime. In the comparison we focus on a relevant subset

of modules and the example applications that include them. The selected Rime primitives against which we compare are abc, broadcast, polite, unicast and multihop, while the Rime applications examples are example_name_of_the_application.

Experimental setup

In order to compare the components of the Rime stack against the components of the CRime stack in terms of the image size we use the VESNA sensor node[9], a hardware platform developed in our lab and used in several sensor testbeds in Slovenia. The VESNA sensor node is equipped with a ST ARM Cortex M3 32 bit microcontroller running up to 72 MHz, 1 MB of FLASH, 96 kB of SRAM and 128kB of fast (2,25 MB/s) non-volatile MRAM memory (NVRAM). The hardware has a fully modular design and can be programmed via RS-232 compatible interface or standard JTAG providing debug capabilities.

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

Image size

We first compare the components of the Rime stack against the components of the CRime stack in terms of the image size. We looked at the size of the code (.text), the size of the initialized (.data) and non-initialized (.bss) static variables and list the results in *Table 1*. It can be seen that the Rime and CRime components differ just in the size of the code.

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

Table 1: 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				

[9] http://sensorlab.ijs.si/hardware.html

ipolite	712	0	0				
stunicast	512	0	0				
				amodule	2572	0	4
				stack	3760	24	0

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

The stbroadcast, ipolite and stunicast modules from Rime have no direct equivalent in CRime. As discussed in the architectural comparison between the modules, the functionality of these is replaced by triggers and by the modular composition of the stack. The two CRime specific modules, which enable modularity, are the amodule and the stack. The code footprint of these is relatively large if we compare them to the other modules. They also use some static variables as can be seen in the corresponding .data and .bss columns of *Table 1*.

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 2 and in

Table 3, while the difference is listed in *Table 4*. It can be seen that the size of the code of the applications which use CRime stacks is about 16% larger (~13.000 bytes) while the absolute size difference of the initialized and uninitialized data sections is below 0.1%.

Table 2 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 3 The code size (.text), initialized static variables (.data) and uninitialized static variables (.bss) of five CRime example applications (the applications are compatible and do the same thing as the Rime applications from). All values are in bytes.

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

example-crime (c_multihop)	97040	94880	1664	5816

Table 4 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 2 and

Table 3. All values are in bytes.

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

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 5 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.

Table 5: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	8.50	close	67.0	5.70
Total	256.5	99.8		622	99.9

It can be seen that in both stacks, the most time consuming operation is sending a packet. Rime needs on average 104 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 consuming operation is processing the received packet throughout the stack. The CRime open, close and recv operations are relatively simple; the overhead comes mostly from the code necessary to propagate through the tree. The CRime send operation is more complex and incurs higher processing time mostly for the following two reasons. First, the operation is typically sent over

one stack. This means that some checking needs to be done so that the operation does propagate on the path of the tree corresponding to that stack and not over the entire tree. Second, this operation needs to handle triggers, for which additional instructions need to be executed.

The total amount of time needed by Rime to execute the sequence of operations open→send→recv→close is ~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 6 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.

Table 6:The cost of stack and amodule abstractions in terms of processing speed.

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

It can be clearly seen in this breakdown in

Table 6 that the overhead for the send operation where checks for walking through the tree and trigger handling are quite high, representing 80% of the total duration of the operation (304 µs out of 380 µs). The overheads for receiving a packet and closing a connection are also high in relative terms representing 87% and 61% of the total duration of the respective operations. The smallest relative overhead occurs for the open operation, representing 31% of the total duration. The differences between the overheads of the four operations are justified by the different implementations of the tree walking algorithms (for the send operation the trigger handling also has to be accounted for).

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 7.

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

Table 7: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_{CR}) [mW]	
example_abc	89.96	(c_abc)	91.84	0.020
example_broadcast	88.96	(c_broadcast)	90.21	0.014
example_polite	88.70	(c_polite)	89.58	0.010
example_unicast	90.09	(c_unicast)	91.84	0.019
example_multihop	91.84	(c_multihop)	93.59	0.019

2.4.1 Conclusions

We have recently proposed [10] a framework that allows composing communication services in a dynamic way. The framework is based on a fine-grained modular protocol stack architecture. The reference implementation of the framework is ProtoStack[10] while the reference implementation of the modular protocol stack architecture is Composable Rime[11] (CRime). In this report we briefly described ProtoStack and CRime and provided a quantitative evaluation of the overhead introduced by composeability by comparing CRime against Rime.

[10] https://github.com/sensorlab/ProtoStack

[11] https://github.com/sensorlab/CRime

More details on ProtoStack and CRime are available in [10] where we show, through feedback collection from first time users, that the ProtoStack tool can significantly speed up prototyping and testing of new stacks and is friendly to novice and advanced users. The initial feedback shows that the tool can speed up design and prototyping of new protocol stack by at least a factor of 2. The cost of increased flexibility and prototyping speed of the protocol stack is paid in terms of increased memory footprint, processing speed and energy consumption. The CRime library used by ProtoStack has a 16% larger footprint, it takes 2.4 times longer to execute an open->send->recv->close sequence and consumes 1.6% more power in doing so. Even though with ProtoStack more resources are consumed by the node, the tradeoff in terms of prototyping speed seems to pay off.

2.5 Testbed-specific Optimization

2.5.1 OMF

The TWIST testbed has very stable and well established architecture for experimenting with wireless sensor network applications. It also has a web interface for experiment control. However during the time of the project we have extended the functionalities to new devices. In this section we describe the improvements to testbed with respect to the devices other than sensor nodes.

The current TWIST web interface is sensor node oriented. That is the reason to keep it only for this purpose and install new framework of experiment control for other devices. We have tried to follow iMinds in this case and install the OMF framework for testbed control. There where however couple of issues that prevent direct clone of the OMF in the TUB testbed. Currently the iMinds testbed uses OMF in version 5.4 which is very focused on the usage with embedded PCs. This implementation was very hard to adapt to other devices that are part of the testbed such as the spectrum analyzer or signal generator. It also requires the Aggregate Manager to control the experiments. It is used to store the inventory, disk images and measurement collections. It also introduces additional overhead.

In the recently released OMF version 6.0 the whole system was greatly simplified. The new architecture is shown in Figure 29. The main focus is put at the Resource Controller (RC). In contradiction to previous versions of the, OMF it can run on the resource itself (like PC) or run separately and control other resources (like sensor nodes) [11]. This is a big improvement from the TWIST testbed point of view. It is now possible to install new RC entity that will be able to control TWIST nodes, Spectrum Analyzer or TWISTbot without major changes in the currently existing infrastructure. It is also said that OMF 6.0 is backwards compatible which means we still will be able to reuse the experiment scripts from other partners.

2.5.1.1 Spectrum sensing tools

In this section we describe the integration of spectrum sensing devices into the OMF framework. We introduce the motivation for such an improvement, then the details of implementation are explained and finally we also demonstrate the benefit for such approach.

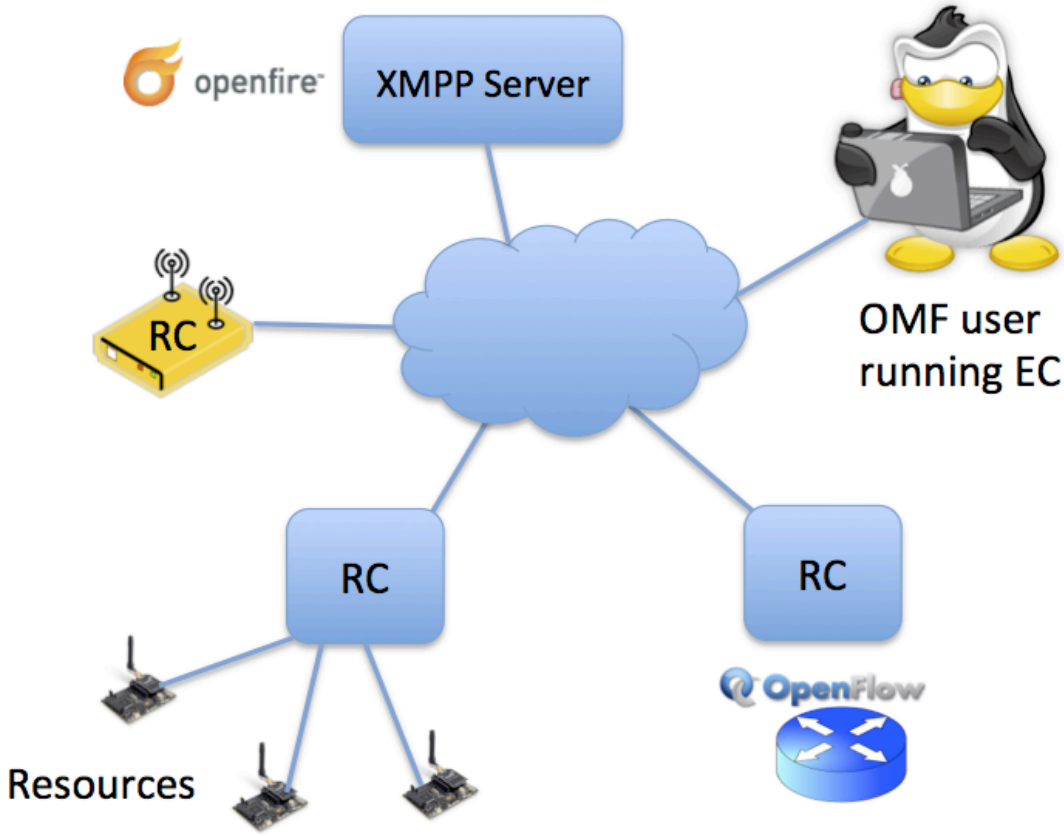


Figure 29 OMF 6.0 architecture [11]

It is possible to use different sensing devices at the TWIST testbed. They are ranging from low quality but simple TelosB nodes with sensor application, through WiSpy USB dongle, to the high performance Rohde & Schwarz (R&S) FSV7 Spectrum Analyzer. All can be used for different purposes with the great success but have different interfaces to gather the data. They also produce the results with different data formats, which also make it difficult to process the measurement data.

It would be impractical to try to uniform the basic interface to all of the devices. That is why we have developed wrappers around the standard interface of the devices. The main job of the wrapper is to correctly start measurements and store the data in the known location. The user does not have to worry about the knowledge of individual initiation and starting procedures, as they could be difficult to remember. This was especially true for TelosB devices, where it is necessary to program the node with the correct application, connect to the node over the serial interface and only then start the measurement gathering application.

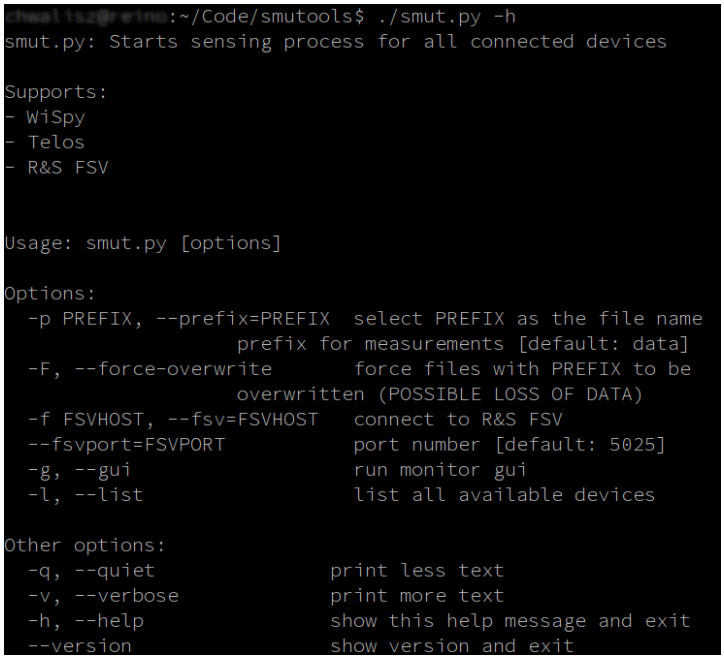


Figure 30 Spectrum sensing interface

The wrappers are implemented as the Python classes and can be started separately or with the common interface that looks for all devices connected to the PC and start measurement on all of them. The interface is shown in Figure 30. In the case of R&S FSV7 spectrum analyzer the user is responsible for the configuration of the device. It has too many options to be able to expose them in the data gathering tool that has the aim to simplify measurement procedure. The idea is to let the user set all the parameters separately and once they are correct easily and safely start the measurement.

The tool also provides graphical user interface for live preview of the measurement data. Example preview can be seen in Figure 31. It is independent to the data collection and can run without interfering with the measurements. It reads already stored data from the file and plots two graphs. The top one is the analyzer view with the current, average, minimal and maximal power per frequency. The bottom shows the spectrogram, where frequency is in X-axis, time on Y-axis and the power is represented by different colors.

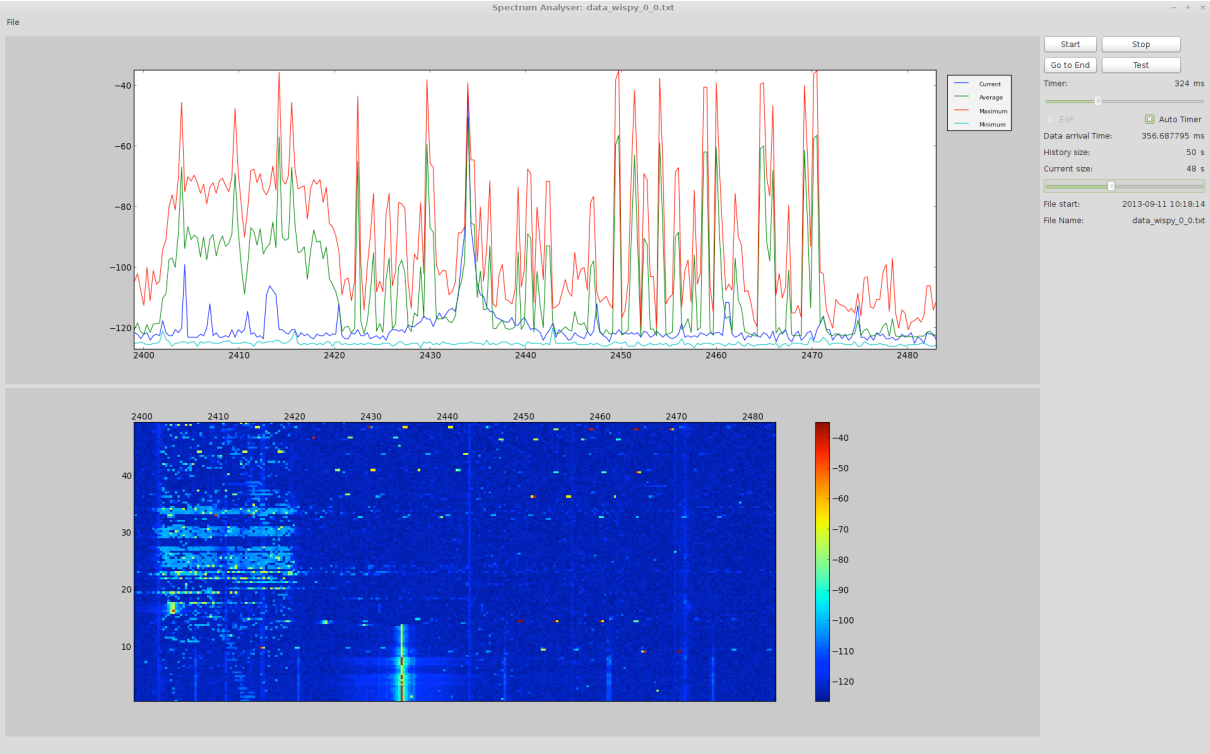


Figure 31 Measurement preview tool

2.5.1.2 CREW common data format processing scripts

All measurement tools tend to store data in its own format. It is done due to efficiency reasons. It also makes it harder for experimenter to analyse the data. In the previous years of the project we have developed set of Matlab processing scripts that aim to simplify the process of loading and working with the data. This year the scripts where migrated to the git repository for better update possibilities and are available under: http://www.crew-project.eu/repository/scripts or https://github.com/mchwalisz/crewcdf_toolbox

The Rohde & Schwarz Spectrum Analyser support has been added. It covers two types of file formats produced by custom developed measurement tools described previously and proprietary tool from Rohde & Schwarz called TraceRecorder. Both follow the same principles as for other devices where it is only necessary to provide path to the trace file and the function will load the data to the common structure as described in the following example:

```
% p = common data format structure
% p.Name         = Unique identifier of the sensing device
% p.Location     = Location of the sensing device (m) e.g [x,y,z]
% p.CenterFreq   = Array defining center frequencies
%                    of the columns of power the matrix (Hz)
% p.BW           = Bandwidth around each center frequency (Hz)
% p.Tstart       = Start time of the measurement in datestr format
%                    e.g. '24-Jan-2003 11:58:15'
% p.SampleTime   = Timestamp relative to Tstart (s)
% p.Power        = Matrix containing power measurements (dBm)
%                    row contains all frequencies for one timestamp
```

The scripts were also extended with new plotting functions. Additionally, to the previously used spectrogram showing the measurement in the time vs. frequency plot where different power levels were represented by different colours, the so called persistence plot can be produced. The sample plot can be found in Figure 32. This type is new and very interesting way of showing spectrum measurements and is available in the newest high end spectrum analysers. It is standard power vs. frequency plot but additionally the colour on the graph shows how many times given value was

recorded during the measurement. It gives a great possibility to see and recognize rare signals in the measurement. It is not possible with any other type of plot. For example it would not be possible to see the narrow band spikes that are around -75 [dBm] strong with averaging, as the noise is received so much more often or maximum hold, as there are other signals that would cover the view. To use this function it is enough to run `crewcdf_persistence(p)` with the standard common data format structure as a parameter.

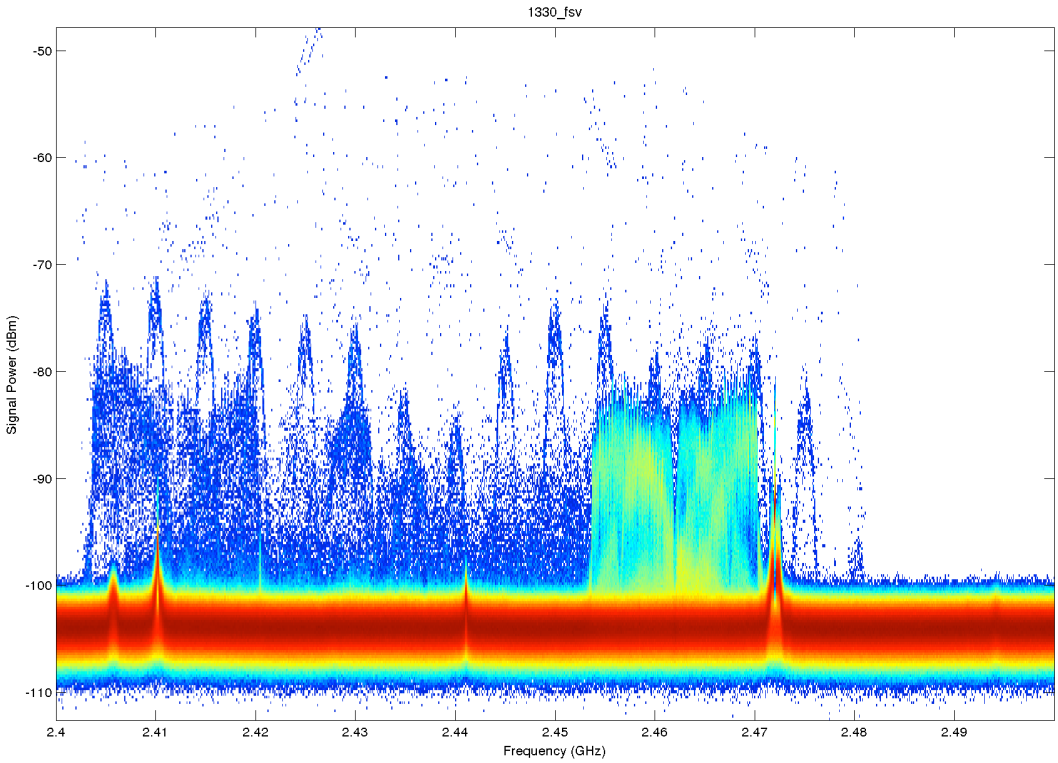


Figure 32 Sample persistence plot

2.5.2 IRIS-GUI

2.5.2.1 Graphical interface support

A key demand-driven extension for the Iris software radio architecture has been support for graphical interfaces for radio control and signal analysis. This extension has been built into the core Iris framework through support for Qt-based graphical widgets. The graphical support has been designed in such a way that graphical widgets may be easily added to any part of the Iris framework including the radio launcher, individual components and controllers. In extending Iris to support graphical widgets, it was vital that this extension should not compromise the use of Iris in non-graphical environments such as on embedded systems and headless servers. To achieve this, the CMake build environment was leveraged to ensure that graphical support is only built-in when the requisite graphical libraries are available on the system. In the absence of these libraries, Iris will successfully build and run but will not include graphics support. In addition to building graphics support into the Iris core framework, a suite of reusable graphical widgets and controllers were developed. These are described in the following sections.

2.5.2.2 Graphical interfaces for signal analysis

A suite of reusable graphical widgets have been developed to support signal display and analysis in the Iris framework. These include the following four widgets:

1. Scatter plot
2. Real plot
3. Complex plot
4. Waterfall plot

The Scatter plot widget supports the display of a vector of complex data points in two dimensions. This is useful, for example, to display the constellation of a received digital radio signal (see Figure 33).

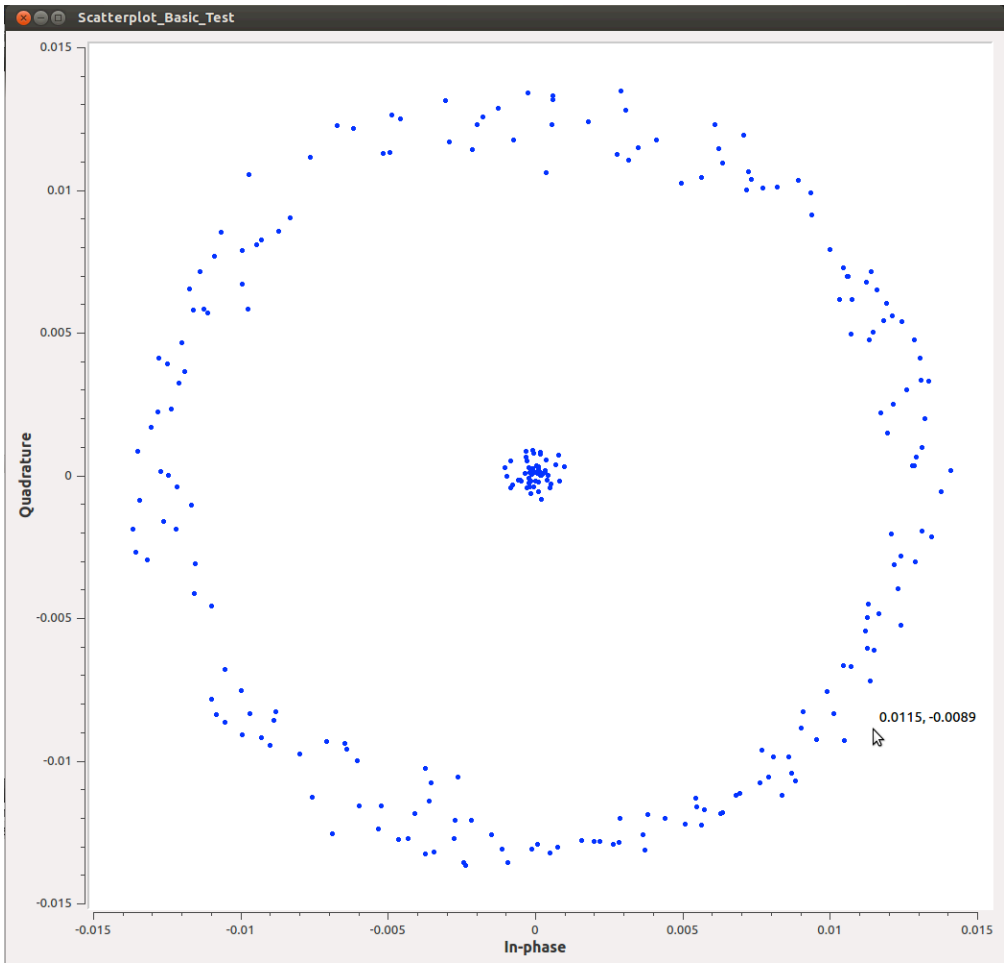


Figure 33 Scatter plot widget example

The Real plot widget can be used to display a vector of real-valued data points such as received signal magnitude. An example can be seen in Figure 34 below.

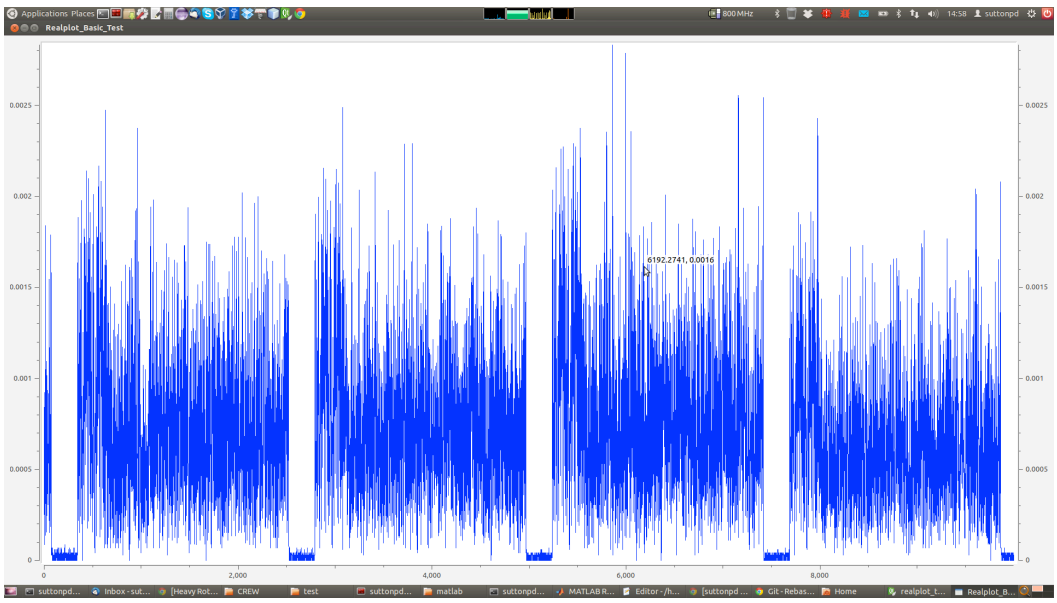


Figure 34 Real plot widget example

Similarly, the Complex plot widget can be used to display a vector of complex-valued data points. The Complex plot widget includes four subplots for in-phase, quadrature, magnitude and phase components of a complex valued vector. Again, an example can be seen in Figure 35 below.

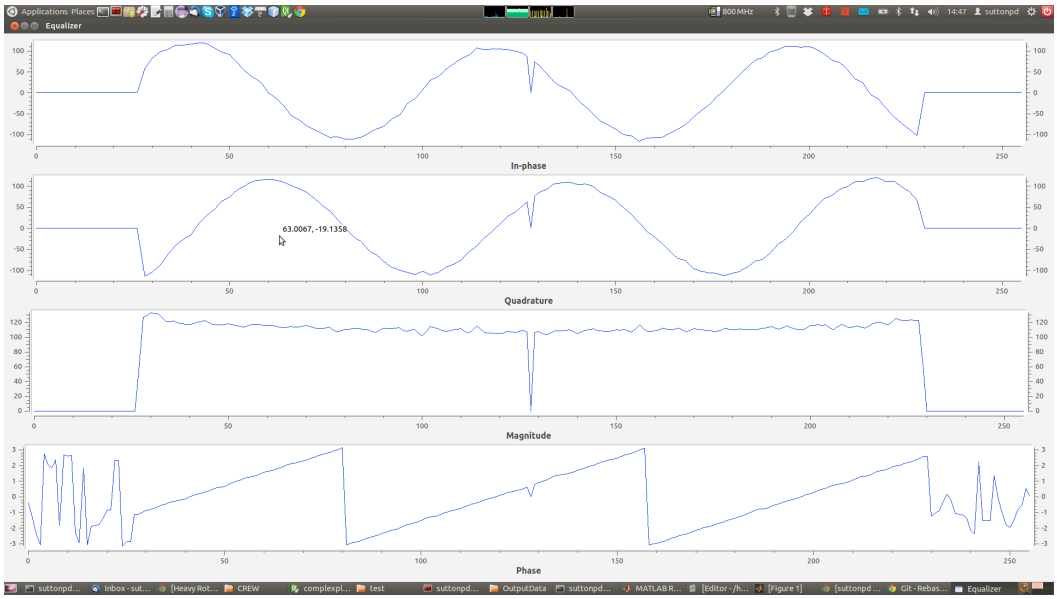


Figure 35 Complex plot widget example

Finally, the Waterfall plot widget can be used to display the historical values of a vector with colours used to indicate intensity. This widget is especially useful, for example, to monitor spectrum usage over a range of frequencies. An example can be found in Figure 36 below (here included as part of a larger graphical interface used for recording the throughput of two wireless links).

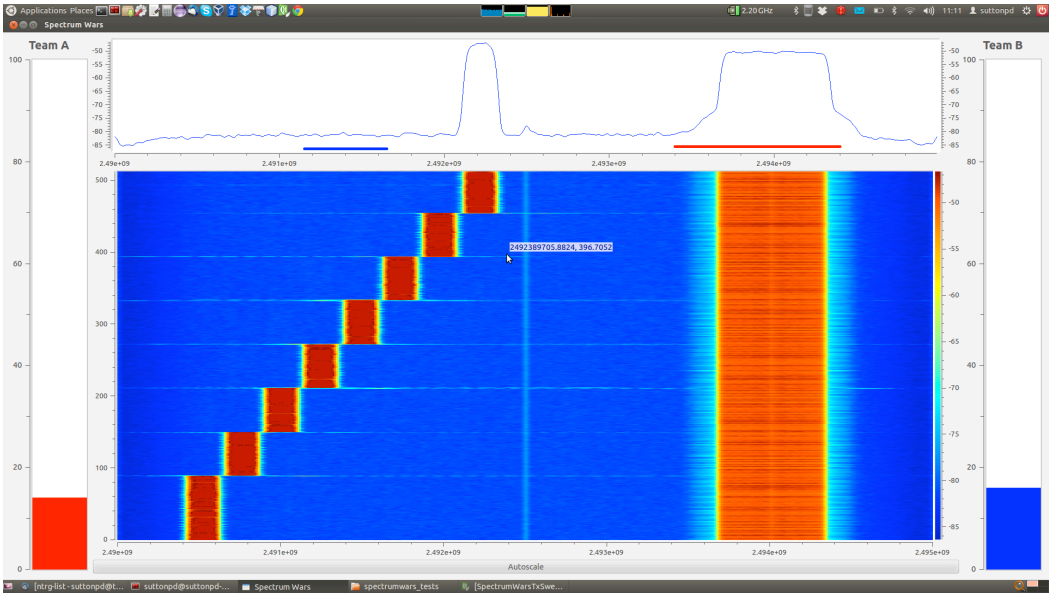


Figure 36 Waterfall plot widget example

Each of the signal display widgets is designed to support both static and time varying data. Every 10ms, each widget will check to see if new data has been provided and will replot if so. This approach ensures that the data refresh rate does not exceed 100 frames/sec (thus necessarily consuming processing resources).

2.5.2.3 Graphical interfaces for radio control

In addition to the signal analysis widgets described above, support has been built into the Iris framework for radio control widgets. These widgets can be used to manually control the performance of an Iris radio system and are often created within an Iris controller, with a global view of the running radio and the ability to reconfigure any parameter exposed within any component. Figure 37 below illustrates one such widget, provided to support reconfiguration and control of an RF front end.

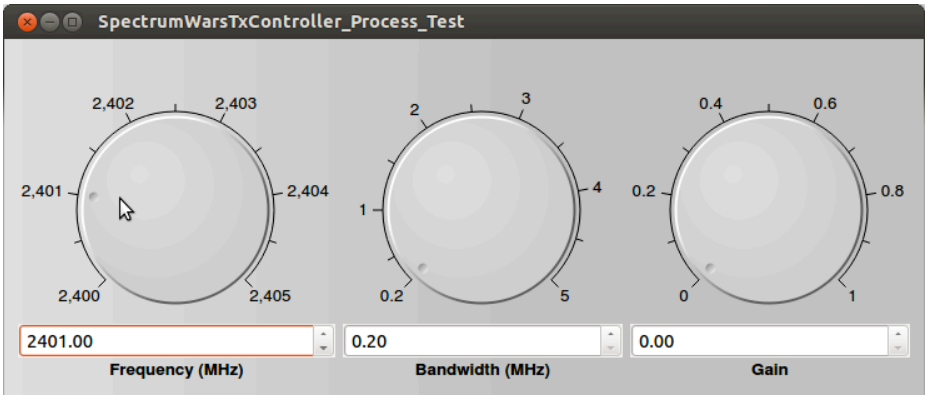


Figure 37 RF front end controller example

2.5.3 GRASS-RaPlaT

2.5.3.1 LOG-a-TEC web portal ↔ GRASS-RaPlaT – API and functions

The LOG-a-TEC web portal[12] communicates with GRASS-RaPlaT by issuing an OS-level command (implemented as a Python script) for the execution of a particular function, and providing the necessary input data with the command-line parameters and in one or more input files. GRASS-RaPlaT performs the required computation and returns results in one or more output files.

Input files are text files containing the required input data in a format specified for each function. Output files are text files containing the computed data in a format specified for each function, or a KMZ (compressed KML) format files containing a geo-referenced raster image (e.g. radio signal coverage).

The GRASS-RaPlaT command interface for the web portal includes an additional intermediate layer that temporarily creates GRASS runtime environment for each command issued by the web portal, allowing execution of GRASS commands directly from the OS-level command interface (instead of from within a running GRASS user session, as is normally the case).

The LOG-a-TEC web portal ↔ GRASS-RaPlaT communication is illustrated in Figure 38.

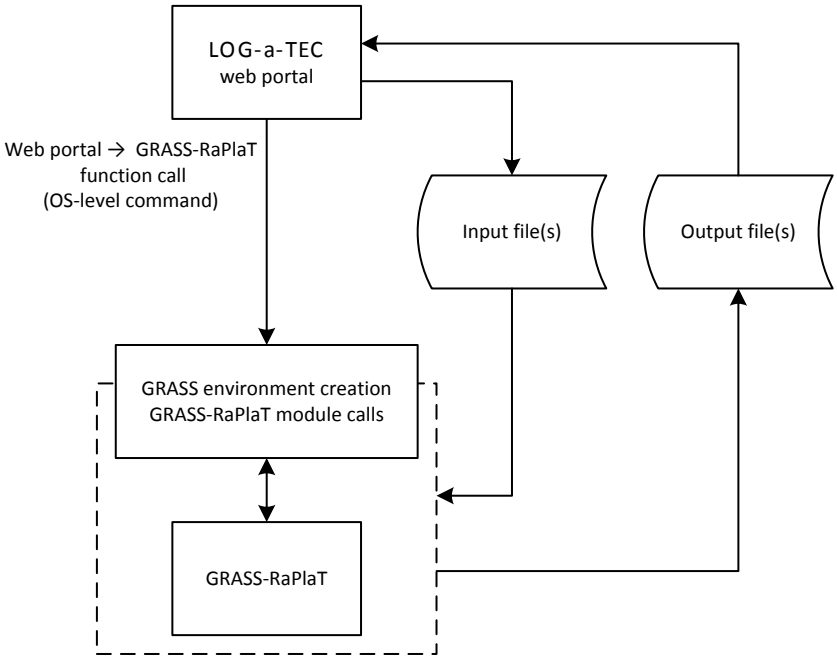


Figure 38 LOG-a-TEC web portal ↔ GRASS-RaPlaT API.

Currently, GRASS RaPlaT provides three basic functions for the CREW web portal:

- Computation of coverage or interference area. The results is a KMZ file containing a raster image of radio coverage (colour-coded received signal strength in each raster point), or interference area (single-coloured area).
- Computation of numeric received signal strengths (in dBm) for a given list of receive points (specified by their coordinates, i.e. longitude and latitude in degrees). This function is related to the coverage computation, but returns numeric values for a set of points instead of a colour-coded raster image of the entire area.
- Estimation of an unknown transmitter's location and transmission power (transmitter localisation), based on received power values from a set of receivers.

[12] http://log-a-tec.eu/

These functions can be combined by the web portal to achieve advanced functions. E.g., the transmitter localisation and interference area computations can be used in turn to compute the interference area of an unknown transmitter.

The first two of the above functions are rather straightforward and do not require further explanation (beyond the description of their use in the CREW web portal, given elsewhere). The newly implemented transmitter localisation function, however, is more specific and its internals are described in greater detail in the following section.

2.5.3.2 Unknown transmitter localisation

The transmitter localisation function returns the 2D location (longitude and latitude) and effective radiated power (ERP) of an unknown transmitter, based on the receive power measurements of at least three suitably placed receivers.

The function first computes path loss raster maps for all receivers (as if they were transmitters), which is a standard function in RaPlaT. By reinterpreting the loss values as gain values (by changing their sign), the receiver/transmitter roles are effectively interchanged. Using the standard GRASS-RaPlaT coverage computation procedures with these *gain* raster maps instead of the path-loss raster maps, and with the received power values as the transmit powers, estimated transmit power raster maps for the unknown transmitter are obtained as seen from each receiver. In this set of raster maps, the function searches for the point(s) where the estimated transmit powers agree, which would represent a possible location of the unknown transmitter. Actually, the function (more precisely the underlying RaPlaT module) first computes a *similarity* raster, where the value in each point represents the similarity between the expected transmitted powers for all receivers. The values are non-negative, with 0 (zero) meaning a complete agreement (ideal solution), and larger positive values larger disagreements between the estimated power values. The transmitter localisation function then searches this raster for a location with high similarity (low value). Since the location estimates are not exact but rounded to the raster resolution, the value of zero is generally not achieved. The current implementation calculates similarity values as the sum of absolute differences from the mean value for each raster point, and searches the similarity raster for the point with the lowest value.

The function returns two files. One is a KMZ file with a colour-coded similarity raster image. The other is a text file containing the estimated transmitter location (longitude and latitude) and its effective radiated power. Actually, there is not only one power value but a list of them, since the values estimated by each receiver are given in the file. Ideally, these values would be equal, but due to rasterisation there are generally small differences. When using more than three receivers, disagreements between them also enlarge the differences (up to the point where the transmitter position cannot be estimated any more).

The transmitter localisation function takes into account radiation patterns of the receive antennas. The transmitter antenna is unknown and an omnidirectional diagram is assumed (e.g. a vertical dipole/monopole surrounded by free space); the results can be completely wrong if this is not the case. Since the transmit antenna gain is unknown, only ERP (Effective Radiated Power) can be estimated, and not TPO (Transmitted Power Output).

At least three receivers are required for transmitter localisation, however with only three receivers two possible solutions exist. Both solutions generally differ in the estimated transmit power. The localisation function returns the position of one of them which is not necessarily the right one. By setting upper and/or lower bounds on the transmit power (optional command line parameters), the right position and power estimate can be obtained. This requires suitable positioning of the receivers relative to the transmitter, i.e. the transmitter may not be in close proximity of one of the receiver, because in such case both transmit power values would be very similar. For omnidirectional receiver antennas, the ideal positions of the receivers would be in the corners of an equilateral triangle with the transmitter in the centre. (In this idealised case, the second solution would be theoretically infinitely far away, with infinite transmit power. In practice, the second solution would be out of the computation area, or at least very far away from the receivers and with a very high transmit power.)

Figure 39 shows a simple example with the transmitter TX1 at equal distances from the receivers RX1 and RX2 and close to RX3. In this case, TX2 with about 4,6x larger transmit power then TX1 would produce the same received signal strengths at the receiver locations.

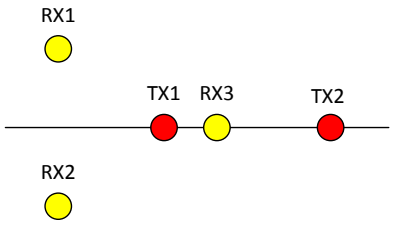
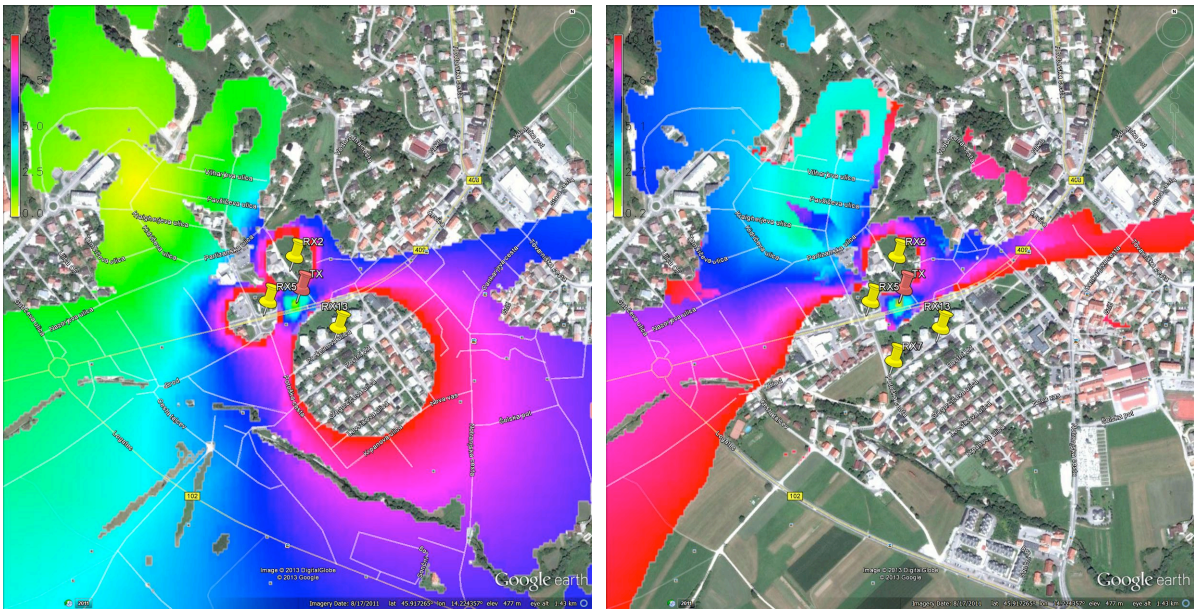


Figure 39 A simple illustration of two possible solutions with only three receivers.

A unique localisation solution can be found without providing transmit power bounds by using measurements from at least four suitably placed receivers. Figure 40 illustrates this by showing similarity raster for 3, 4, 5 and 6 receivers (yellow markers); only in the first case (3 receivers) the solution is not unique, with an alternative transmitter location in the upper left region (yellow colour) and its estimated power 13,7 dB higher than for the actual location (red marker).



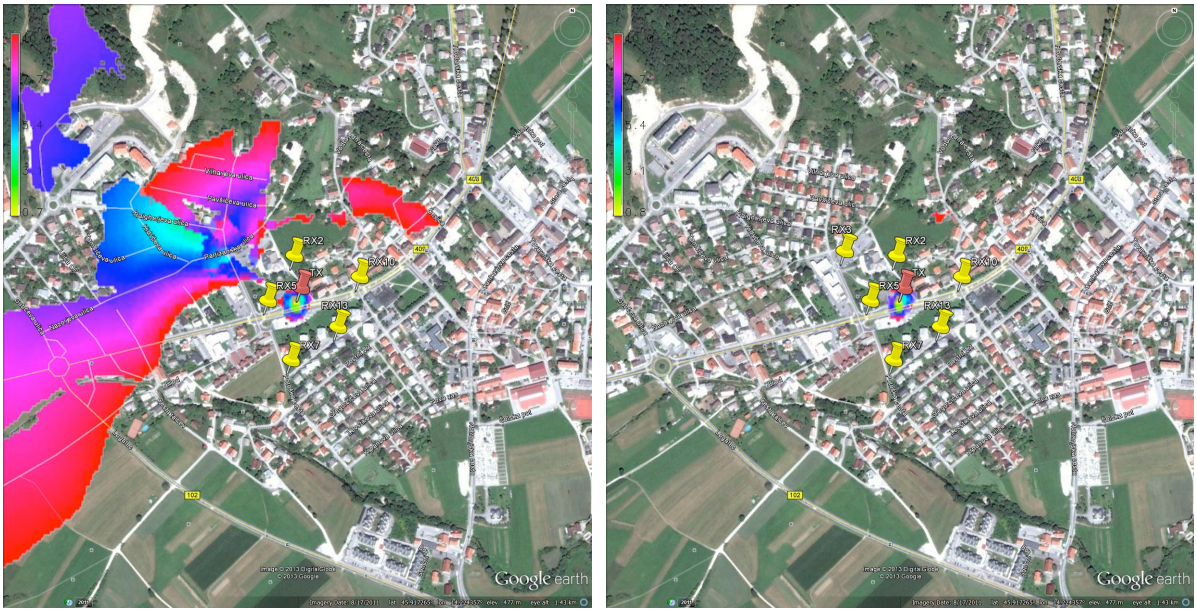


Figure 40 Probability raster images - 3, 4, 5 and 6 receivers.

The measurements from multiple receivers must be in agreement, i.e. without large individual errors, since these would prevent finding a valid solution (if necessary, an improved strategy may be implemented in the future to mitigate problems with erroneous measurements).

It should be kept in mind that the current computation raster is 5 m (limited by the available DEM raster resolution), and the distances between the receivers and the unknown transmitter should be well above this value to avoid numeric errors and problems due to rasterisation.

2.5.3.3 GRASS- RaPlaT portal integration

All features of GRASS- RaPlaT have been implemented in our web portal. The user is able to choose between 4 options. First is the calculation of coverage second is the calculation of interference region third is signal power at specified locations of the receivers and fourth is the unknown transmitter localization. The web portal supports placing transmitters anywhere on the map by double clicking on the map. The nodes that are integrated on the map are actually nodes mounted on the light pols in the city of Logatec. Each node that is marked on the map can be a receiver or a transmitter depends on the demands of the simulation.

The unknown transmitter localization is currently supported for a predefined and pre calculated example however it is going to be extended to support real measurements. Through the portal we can change the number of receivers and observe how it effects the localization result. The real measurements from the testbed where unknown node is transmitting and at least three nodes are receiving are stored in the database and used as an input for GRASS- RaPlaT transmitter localization module. When the unknown node is located it is placed on the appropriate place on the map. Figure 41 is showing the usage of LOG-a-TEC portal for coverage calculation.

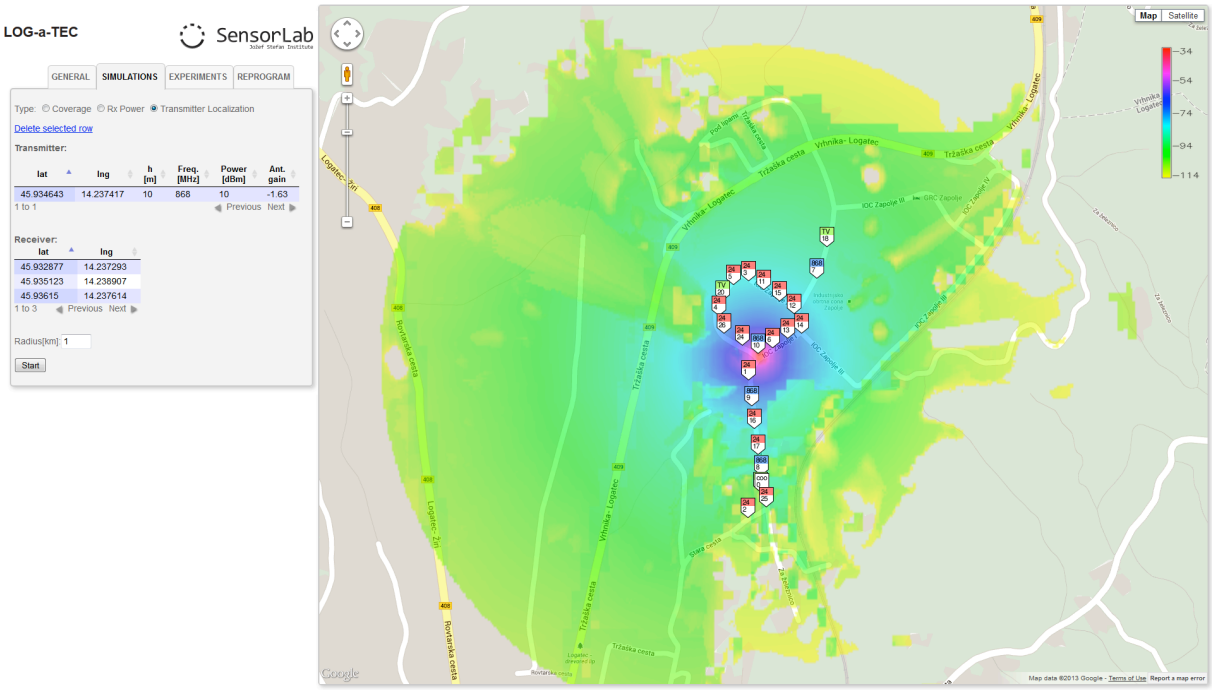


Figure 41 LOG-a-TEC Portal

2.5.4 Improvements on w-iLab.t

2.5.4.1 Improvement for USRP sensing engine

In this section, we focus on the improvement of the sensing efficiency of the USRP sensing engine. First, we introduce the background and motivation for such an improvement, then the details of the implementation are explained, finally we also demonstrate the flexibility and benefit via the list of configuration options.

2.5.4.1.1 What is sensing efficiency and why it is important

One of the most crucial aspects of sensing engine is its efficiency. Discontinuity in spectrum sensing often leads to inaccurate assessment and missed detection of interference. Spectrum sensing generally consists of two phases: the sampling phase, in which raw samples are collected from the air; the processing phase, in which buffered samples are processed for spectrum analysis.

Depending on the processing speed, the processing phase can partially or completely happen in parallel with the sampling phase. The time used for collecting samples from the air is referred to as the sampling time, while the time required by the processing phase in addition to the sampling time is referred to as the processing time. The sensing efficiency is then defined as the ratio of the sampling time and the summation of the sampling time and the processing time. During the processing time, the sampling of the wireless medium is put onto hold, which means the sensing engine is “blind”. It is possible that a number of transient signals are missed during this period, as illustrated in Figure 42. In this figure, one can see in the upper part, where FFT processing time is longer than sampling time, discontinuous sampling and missed transient signal. In the lower part, the analyzer is capable of detecting transient signal thanks to continuous spectrum sensing. The time interval when the sampling activity is put onto hold is referred to as the blind time.

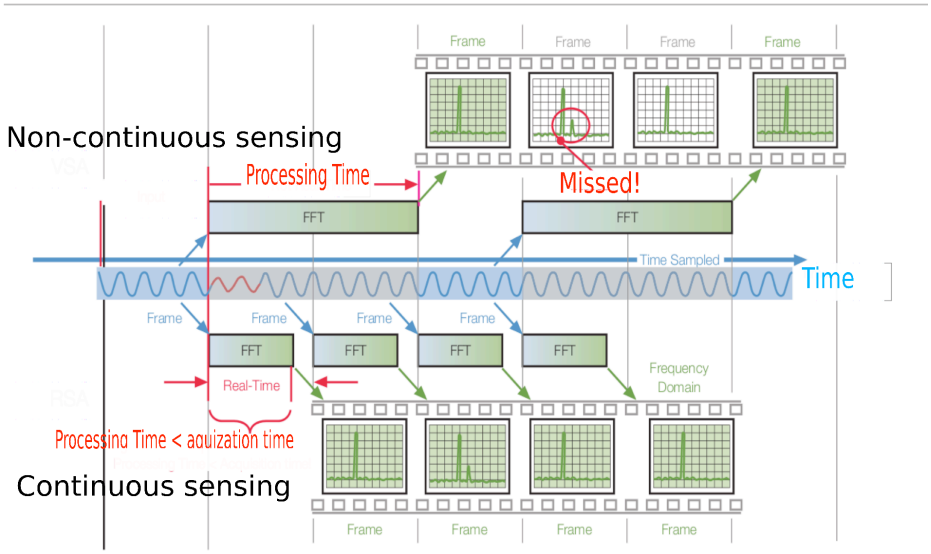


Figure 42 Continuous and non-continuous sensing. (This figure is adapted from [14].)

Ideally, the blind time should be reduced to zero, meaning 100% sensing efficiency to achieve seamless detection. There are various sensing devices on today's market. Solutions such as spectrum analyzers are capable of scanning a wide spectrum range, but are not dedicated for channel assessment and extremely costly. For instance, a spectrum analyzer usually cannot do continuous recording for a longer time period than a few seconds, and the recorded spectrum need to have high frequency resolution for visualization purpose. However, the raw spectrum information still requires further processing to obtain the energy for specified channels. On the other hand, low cost solutions are trimmed for simple and steady recording, but lack the flexibility and required performance. For instance, they are not able to achieve seamless spectrum sensing, and usually have non-configurable frequency span and resolution bandwidth.

After realizing the importance of sensing efficiency, and the lack of high efficiency in today's main stream spectrum analysis devices, we decide to rebuild the USRP sensing engine into ***an alternative*** sensing solution — a ***sensing engine*** with high efficiency and flexibility**.**

2.5.4.1.2 The software architecture

As stated previously, it is important that the processing time is shorter than the sample acquisition time in order to achieve continuous spectrum sensing. When no parallelism is present, the sample acquisition alternates with the processing phase, and as such the blind time of the sensing engine is equal to the processing time (as illustrated in plot (a) of Figure 43).

When pipelining between sample acquisition and processing is introduced, after the first batch of samples arrived, sampling the either and processing the samples obtained in the previous time frame are happening in parallel. This is the first level of parallelism. The blind time is equal to the original processing time minus the sampling time, as shown in plot (b) of Figure 43.

To further reduce the processing time, we seek to add parallel processing within the processing phase itself. The processing phase consists of splitting samples into small frames, applying FFT operation on all frames sequentially, and combining all the FFT result in one way or another. As FFT is a highly computational demanding operation, instead of having one single FFT core working sequentially, we utilize multiple FFT cores to work simultaneously: the incoming samples are divided among the multiple FFT cores for processing. Once the samples have been received, the FFT cores work independently from each other, hence ideal for parallelism. This is where the second level parallelism is introduced. We illustrate the case of two FFT cores working in parallel in the plot (c) of Figure 43. More FFT cores could be added if it is necessary to achieve continuous spectrum sensing.

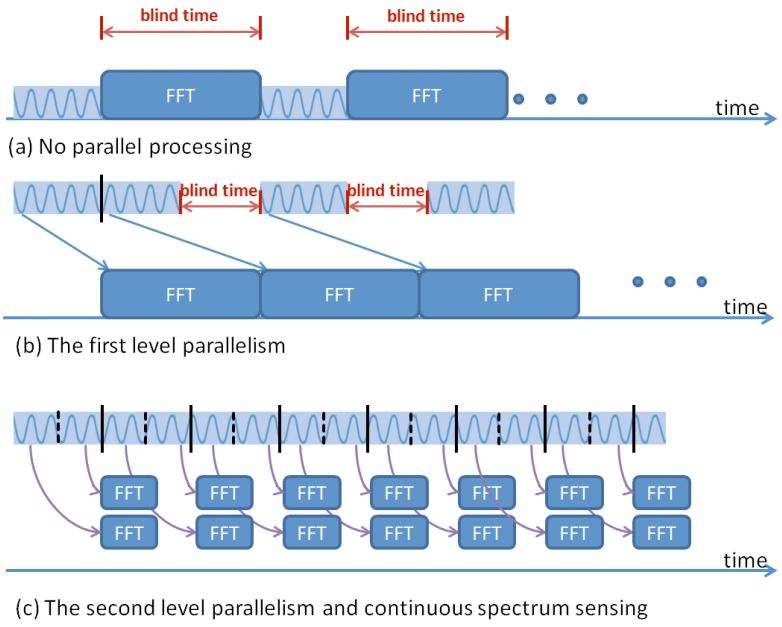


Figure 43 Parallel processing for seamless spectrum sensing

The sensing engine software relies on multi-threading to achieve parallel processing. There are two main threads running at any moment, one thread is responsible for collecting samples from the USRP (referred to as the sample-collecting thread), the other thread is responsible for processing the samples (referred to as the sample-processing thread). The sample-processing process again generates several sub threads to process the incoming samples in parallel. The sample processing in our solution calculates the FFT based power spectrum density (PSD) and the energy for specified channels. Once all sub threads finish processing, they terminate and the original sample-processing thread outputs the result to either a local file or the standard output. To simplify the collection of measurements in the wilab.t testbed, the measurements are first printed to the standard output and then piped to a predefined database using an OMF wrapper. The general structure is illustrated in Figure 44.

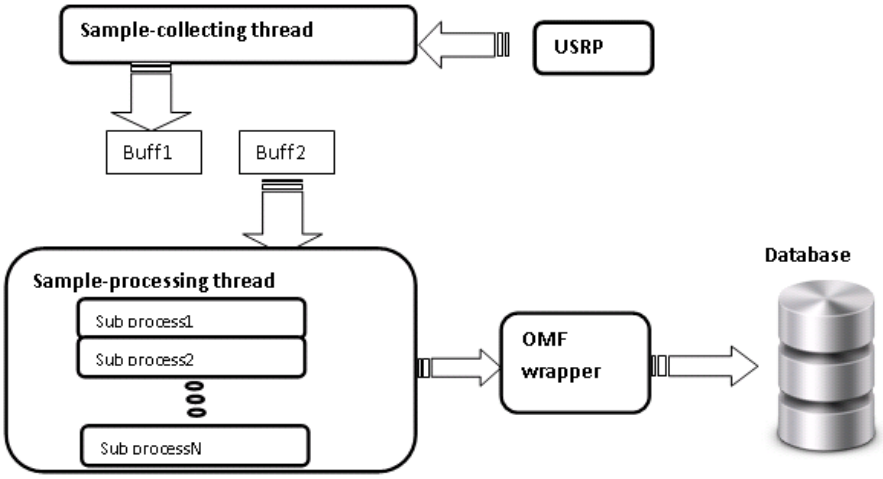


Figure 44 High level description of the software for seamless spectrum sensing

To achieve true parallel pipelining, two buffers are used to collect samples from the USRP. At any given moment, when one sample-collecting thread is writing to one buffer, the sample-processing thread will be reading from the other buffer. Therefore, once the first batch of samples have arrived, the two main threads work fully in parallel. To ensure that the two main threads do not read and write to the same buffer at the same time, the sample-processing thread needs to work faster than the

sample-collecting thread. Hence within the sample-processing thread, several sub threads are created to accelerate the processing. The number of threads that should be used to achieve best efficiency depends on how many samples the buffer contains and the FFT size. During our experiments, 8 processing-threads are sufficient to support a sample-collecting thread at the highest sample rate of the USRP (25Msps). However, for configurations in which only a small amount of samples is collected, the overhead of creating multiple threads outweighs its processing benefit, hence the sample-processing thread can no longer follow the sample-collecting thread. When this happens, the software detects the overflow of samples and returns an error message.

2.5.4.1.3 Configurations and important features

The sensing engine software can be configured using various options, which are described in detail in this section.

Continuous FFT mode vs Swept FFT mode

First of all, the sensing engine can be used in two modes: the continuous FFT mode and the swept FFT mode. For the continuous FFT mode, the USRP front-end stays at the same frequency and continuously samples the wireless medium. Similar to spectrum analyzers, users can control both the center frequency and the sample rate in order to define the spectrum range. For the swept FFT mode, the USRP will always collect samples at its maximum sample rate of 25 Msps. The samples collected at a specific RF center frequency are called a block, while the complete measurement across several RF center frequencies is called one sweep. Between two adjacent blocks, the center frequency is incremented by a step of 20 MHz. Users can specify the center frequency of the beginning block, and how many blocks one sweep should contain. As such, by adding the swept FFT mode, the frequency span is no longer limited by the sample rate.

Measurement types

The sensing engine can be configured to perform different types of measurement.

(i) The sensing engine can measure the PSD in the required frequency range, and thus calculates the amount of energy detected in each specified channel. This is referred as the PSD measurement. The PSD measurement has three variants: averaging, maxhold and minhold. Typically, the number of samples per buffer is a lot larger than the FFT size, hence each buffer contains many FFT frames. For the PSD measurements, the software does either averaging, max hold or min hold across different FFT frames, and the final FFT result is used for the power integration for the requested channels. The maxhold mode is useful to detect the signal's presence, while minhold mode can be used for estimating the noise floor.

(ii) For the continuous FFT mode, the sensing engine can also measure how much portion of time the energy of the specified channels is above a certain threshold. This is referred as the duty cycle measurement. To realize this function, the software investigates a particular channel and counts how many times its energy is above a threshold, and then divide this number by the total number of FFT frames in the buffer. When the appropriate threshold is selected (a value that is slightly above the noise floor), the duty cycle mode can be a powerful tool to detect transient signals.

Sensing efficiency

Recall that the sensing efficiency is defined as the ratio of the sampling time and the summation of sampling time and the additional processing time. In our case, the processing phase happens entirely in parallel with the sampling phase. Hence no additional time is required by the processing phase. For the continuous FFT mode, the sensing efficiency is always 100%, since the USRP never stops sampling. For the swept FFT mode, the sampling phase must be interrupted for channel switching, which is the only cause for time loss. Hence the sensing efficiency for swept FFT mode is defined as:

$$\gamma = \frac{SamplingTime}{SamplingTime + ChannelSwitchingTime}$$

Unfortunately, channel switching of the USRP is more complicated than only tuning the radio front-end's center frequency. The host machine needs to communicate with the embedded processor on the USRP over the Ethernet interface. The exact handling of channel switching depends on the firmware on the embedded processor and the driver of the host machine.

To measure the channel switching time, Wireshark was used to record the packets between the USRP and the host machine. All configuration packets have a short packet length, while the packets containing IQ samples are typically 1514 bytes long. By using a packet length based filter, only the configuration packets used for channel switching and streaming commands can be displayed. Based on this output, Wireshark can generate an IO graph, plotting the packet/ms vs the time, as shown in Figure 45. Only packets with length smaller than 1514 are displayed. The y axis is the packet rate and the x axis is the time in ms accuracy.

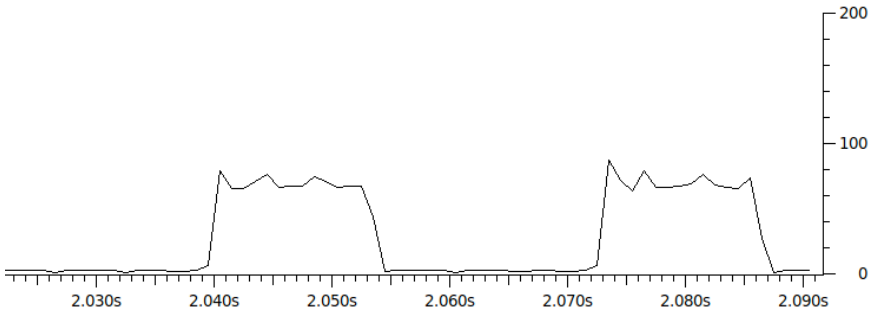


Figure 45 Wireshark IO graph derived from a packet trace between the USRP and the host machine.

This graph gives an indication on how much time is spent on sampling and how much time is spent on channel configuration. The sampling time is directly related to the requested number of samples by one stream command. This is defined by the option "—spb" in the sensing software, standing for sample per buffer. The packet trace shown in Figure 45 is generated with 524288 samples per buffer, the sampling time for each block is 524288/25Msps = 21 ms, this result corresponds with Figure 45. The configuration time for channel switching cannot be influenced by software options. It currently requires about 19 ms to switch a channel for the USRP. The channel switching time is a hardware and driver issue, it could be reduced by improving the driver and firmware.

Up till now, we have identified the channel switching time, the sensing efficiency in the swept FFT mode can be calculated by the above equation. Increasing the sampling time is an effective way to improve the sensing efficiency. When configuring the sensing engine with "spb" equal to 4194304, the sampling time is $\frac{4194304}{25000}\text{KHz} = 168\,\text{ms}$, the sensing efficiency is $\frac{168\text{ms}}{168+19}\text{ms} = 89.8\%$. With 5 blocks per sweep, the sensing engine can cover 100 MHz bandwidth and produce 1 sweep per second. Note that this configuration is very similar to the measurement capabilities of Airmagnet, however the sensing efficiency of Airmagnet is only 15% [15].

Channel Configuration

It is important to let the sensing engine know which channels to measure. Four configuration options are used to complete this target:

- numofchannel: specifies the number of channels to be measured
- firstchannel: the center frequency of the first channel
- channelwidth: the bandwidth of each channel
- channeloffset: the difference between adjacent channels center frequency

At this moment, software only allows to specify channels that are uniformly spaced and with identical bandwidth. This format is flexible enough to describe the channel specifications of the most popular wireless standard. As an example, to measure the 13 channels of Wi-Fi in the 2.4GHz range, the following settings are used:

"--numofchannels 13 --firstchannel 2412000000 --channelwidth 22000000 --channeloffset 5000000"

The above options tell the sensing engine to measure 13 channels, with the first channel starting at 2412 MHz, each channel is 22 MHz wide, the center of all the channels are 5 MHz apart. This feature makes it easy to conduct measurements for different technologies.

Output format

The output of the sensing engine contains the following components:

- timestamp: a unix timestamp in microsecond precision
- usrpid: the id of the USRP used to collect samples
- energy or duty cycle array: an array that contains either the energy (in dBm) or duty cycle (in percentage) for all channels of interest

In contrast to the raw spectrum measurements from spectrum analyzers and some USB based devices, the output format can directly be used for channel assessment.

Resolution bandwidth and FFT size

Usually, the resolution bandwidth (RBW) of the FFT based spectrum analyzer is calculated as the ratio of sample rate over FFT size. Similar to the Tektronix RSA analyzer, users can also directly specify the FFT size and sample rate independently.

The final frequency resolution that is obtained is defined by the number of channels and the channel's bandwidth. So the RBW does not directly rely on the FFT size. However, it is still necessary for the underlying RBW of FFT to be sufficiently smaller than the interested channel bandwidth, otherwise the power integration for the specified channel will be less accurate.

2.5.4.2 Extension with new wireless technologies

iMinds' w-iLab.t has been extended with cellular technologie by the addition of 4 femtocells. Two of these are commercial 3G femtocells (UMTS, HSPA), while the other two are pre-commercial 4G femtocells (LTE). They have been purchased from ip.access, a well-know reseller of small cells. iMinds has purchased an academic license for a software tool to emulate the core network (Evolved Packet Core, EPC) which interconnects these femtocells Unfortunately, this license only allows national research and cannot be used within an international context. Therefore, the available EPC emulation cannot be used within the CREW context. To mitigate this problem, iMinds is further exploring other possibilities to have EPC emulation available for international research at its testbed.

2.6 Collaborations with Other Projects

2.6.1 CREW-OpenLab

As described in D4.2, the CREW-Openlab collaboration results in the web tool CONCRETE. The main functionality of this tool is that it is able to calculate cross correlation of traces obtained in different experiment rounds. The correlation value is a fairly good indication for the stability of the experiment, high correlation value indicates that certain experiment is repeatable, low correlation value indicates there might be external interference or a particular trace is an outlier.

In the year, we aim to explore this tool further by a set of experiments. There are two different scenarios, performed on the w-iLab.t and NITOS testbed. For the experiment scenario on w-iLab.t, it involves a moving robot, connected to a fixed Wi-Fi access point. The moving robot receives packet from its access point while it is moving on a predefined route. At the same time, it tracks the RSSI

and the throughput. The route is carefully chosen to include an obstacle between the robot and the access point, as indicated in Figure 46. This route design gives sufficient variation in the RSSI trace.

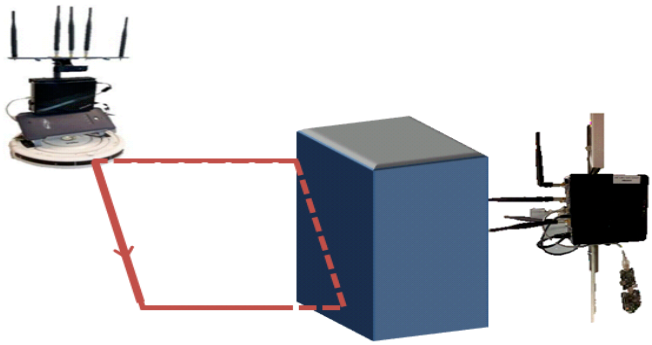


Figure 46 the route of robot in w-iLab.t experiment

The complete set of experiments is still undergoing. Therefore at this moment no comprehensive analysis is presented. But we show some initial result to illustrate how the RSSI and throughput is correlated. Figure 47 contains two typical graphs of the RSSI and throughput traces obtained during one round of experiment. The left side of Figure 47 shows the RSSI and the right side shows the throughput performance. We can see that as the robot driving close by the access point, the RSSI increases but with more fluctuation, the same trend is observed from the bandwidth trace. The fluctuation is caused by the shadowing and fading of the obstacle.

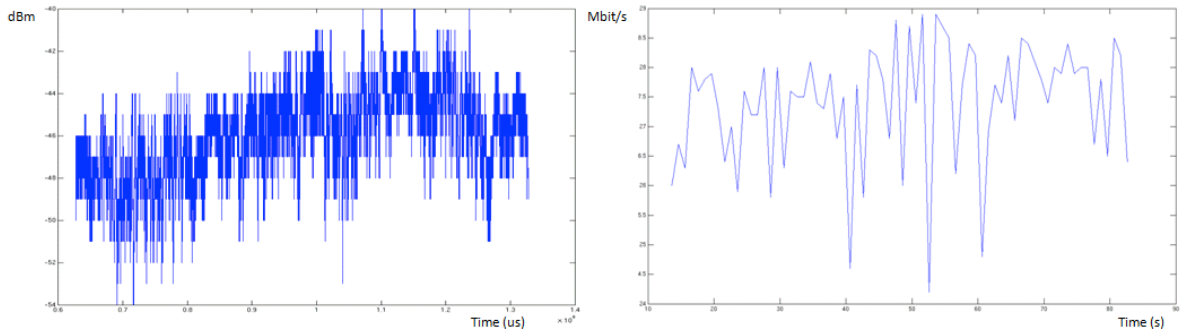


Figure 47 The traces of RSSI and throughput performance

2.6.2 CREW-Geni

The CREW-GENI collaboration consists in creating a common ontology for describing spectrum sensing and cognitive radio experiments that is sufficiently generic and expressive to be used by a large number of experimentation facilities. The work on the ontology started from the CREW common data format (CDF) that is used by the CREW federation for experiment specification as reported in D5.1.

The current version of the ontology focuses on specifying device capabilities in the scope of configuring spectrum sensing experiments and is depicted in Figure 48. The latest version of the ontology with a wiki section to which information will be added is available at https://github.com/cfortuna/CROntology

The Spectrum sensing experiment description ontology is being used by the Orbit testbed in the Orbit device inventory:

- Human readable version

http://www.orbit-lab.org:8080/tasor/#http://sensorlab.ijs.si/2013/v0/SpectrumSensingExperimentSpecification.owl%23RadioDevice

- Machine readable version http://www.orbit-lab.org:8080/tsc/resources/OrbitInventory

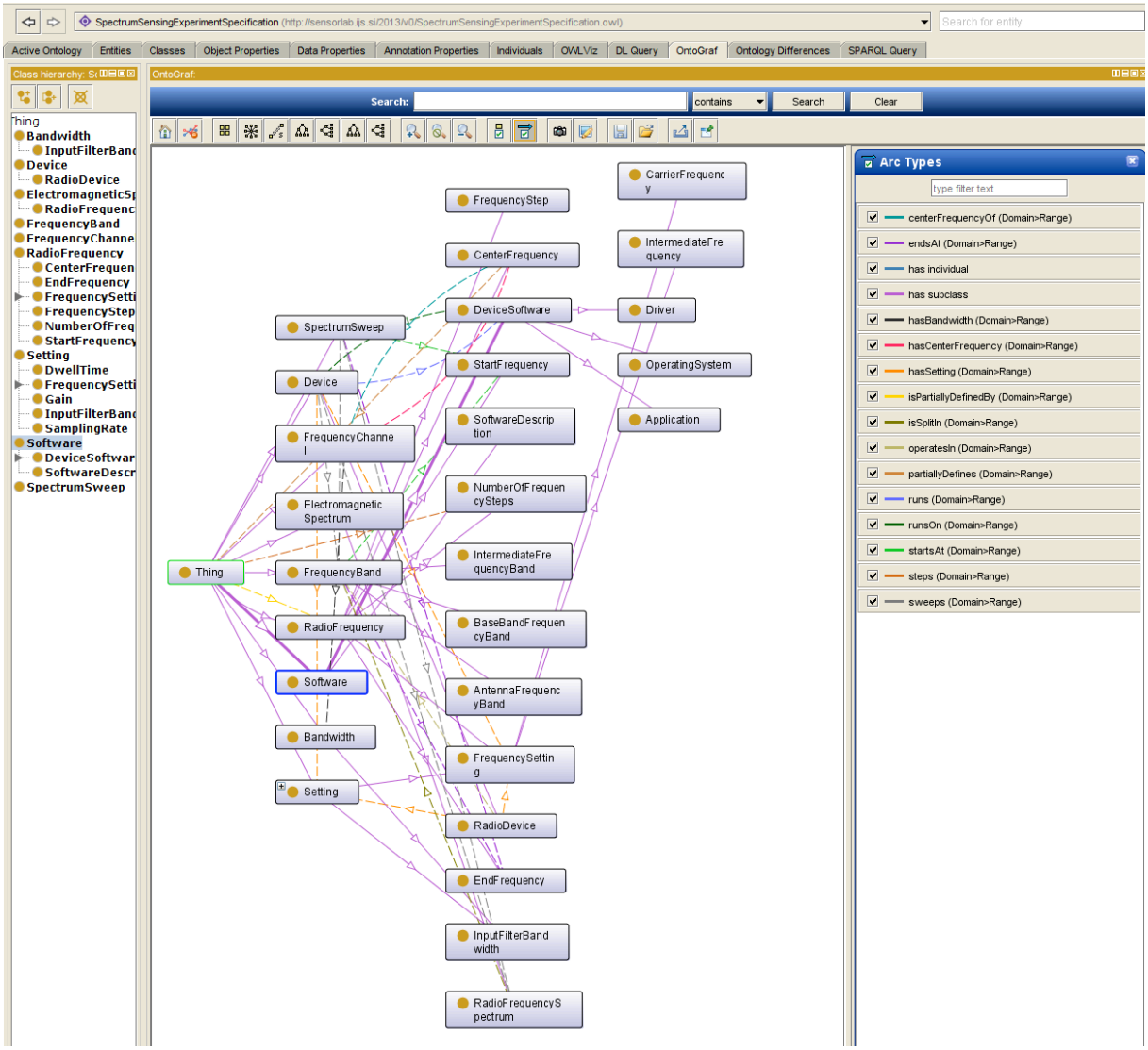


Figure 48 The Spectrum sensing experiment description ontology.

In the Spectrum sensing experiment description ontology we define three orthogonal concepts that allow the description of:

- spectrum related theoretical aspects,
- device spectrum sensing capabilities and
- ranges of values for each

This is depicted in Figure 49 where the light blue concepts represent the theoretical layer, the light green ones represent the device spectrum sensing capability layer and the dark blue individuals represent ranges that can be sets or intervals.

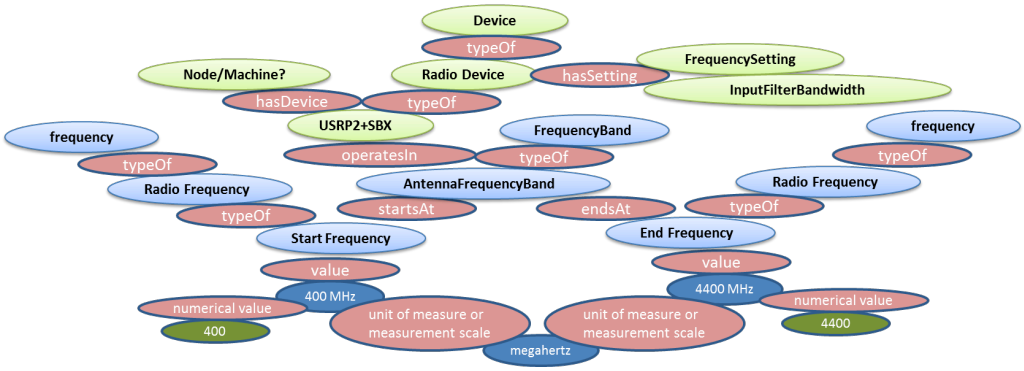


Figure 49 Conceptualization of the 3 layered approach.

3 Demand-driven Extensions Derived from Open Call 2 Experiments

In this section we describe the extensions and actions that were necessary to support the experiments carried out by the new project partners who participated in CREW via the second open call (WP7).

3.1 Support for CREW-TV

3.1.1 JavaScript library for communication with LOG-a-TEC testbed

CREW-TV involves integrating the LOG-a-TEC testbed as a spectrum sensing data provider with the geolocation TVWS database developed by Instituto de Telecomunicações. The TVWS database must be able to instruct sensor nodes in the testbed to periodically perform spectrum sensing sweeps of the UHF frequency band and retrieve channel occupancy data for integration with existing database contents.

For the high-level testbed control the preferred means of programmatic access to the LOG-a-TEC testbed is the Python module library *vesna-alh-tools*. However, because the existing TWVS database implementation is incompatible with Python, it was decided that using *vesna-alh-tools* was impractical in this case.

Because of the familiarity of Instituto de Telecomunicações with JavaScript, an open source JavaScript library analogue to *vesna-alh-tools* has been developed at JSI instead to serve as a middle layer between the TVWS database and the LOG-a-TEC testbed. This library is called *vesna-alh-js*[13]

vesna-alh-js provides JavaScript equivalents of objects from the Python module library. It can be used in a web browser or a headless server environment like node.js. Like the Python library, it provides convenient methods of accessing the HTTP-like resources exposed by the sensor nodes over the Internet gateway and the LOG-a-TEC management network. These are contained in the *ALHWeb* and *ALHProxy* objects. On top of these abstractions, a higher layer of objects is built that abstracts the spectrum sensing abilities of sensor nodes. Namely the *SpectrumSensor*, *SpectrumSensorProgram* and *SpectrumSensorResult* classes allow for programming individual sensor nodes equipped with the SNE-ISMTV-UHF receiver for energy detection spectrum sensing tasks and retrieving the results over the Internet. Currently the JavaScript implementation lacks the other abstractions present in the Python module, like the support for signal generation and firmware reprogramming, since these features have not been required for integration with CREW-TV database.

Following is a brief description of the object classes exposed by *vesna-alh-js*:

[13] https://github.com/sensorlab/vesna-alh-js

ALHWeb(base_url, cluster_id)

ALH protocol implementation through the HTTP infrastructure gateway server.

Constructor parameters:

- **base_url** Base URL of the HTTP API (e.g. https://crn.log-a-tec.eu/communicator)
- **cluster_id** Numerical cluster ID

ALHProxy(alhproxy, addr)

ALH protocol implementation through an ALH proxy.

This implementation forwards arbitrary ALH requests through the *nodes* resource on an ALH service used as a proxy.

Constructor parameters:

- **alhproxy** ALH implementation used as a proxy
- **addr** ZigBee address of the node to forward requests to

SpectrumSensor(alh)

ALH node acting as a spectrum sensor.

Constructor parameters:

- **alh** ALH implementation used to communicate with the node.

SpectrumSensorProgram(sweep_config, time_start, time_duration, slot_id)

Describes a single spectrum sensing task.

- **sweep_config** Frequency sweep configuration to use.
- **time_start** Time to start the task (UNIX timestamp)
- **time_duration** Duration of the task in seconds
- **slot_id** Numerical slot ID used for storing measurements

SpectrumSensorResult

Result of a spectrum sensing task.

Public properties:

- **program** *SpectrumSensorProgram* object that was used to obtain this result.
- **sweeps** Array of *Sweep* objects containing results of frequency sweeps.

The fact that *vesna-alh-js* can be used from a web browser greatly simplifies experimentation with the LOG-a-TEC interface since no additional software needs to be installed on the experimenter's computer. However because of the same-origin security policies of most common web browsers, the JavaScript code running in the browser cannot usually access the LOG-a-TEC gateway directly. This can be worked around by disabling the same-origin policy in the browser, or by loading the *vesna-alh-js* and accessing the LOG-a-TEC through a specially created, standalone HTTP proxy server *devserver.py* that comes with *vesna-alh-js* distribution. This proxy, written in Python, makes the *vesna-alh-js* JavaScript module and the LOG-a-TEC Internet gateway appear to come from the same originating domain from the standpoint of the browser's JavaScript interpreter.

3.1.2 Geolocation data for sensor nodes

To integrate spectrum sensing data obtained from sensor nodes in the LOG-a-TEC testbed into the CREW-TV TVWS database the geolocation (i.e. geographic coordinates, latitude and longitude) of receiver antennas need to be known.

Individual sensor nodes in the LOG-a-TEC testbed are not equipped with a GPS receiver. However at deployment of each sensor node into the testbed the mounting location has been manually noted using a hand-held GPS receiver. Since the nodes themselves are not mobile the location of any individual node can be determined from its network address and the list of coordinates compiled at testbed deployment.

To make the LOG-a-TEC integration with the TVWS database more flexible and reduce the need for hard-coded knowledge of the testbed on the Instituto de Telecomunicações side of the integration, geolocation information for individual sensor nodes has been made available through the HTTP-like protocol that is also used to measure and access the spectrum data. This involved making sensor nodes in the testbed aware of their location and exposing this knowledge through a resource that can be queried over the LOG-a-TEC management network.

Specifically, this involved modifying the *description* resource on spectrum sensing nodes so that a GET query also reports the node's geographical coordinates. For example, the *description* resource on node 19 in the industrial zone environment of the LOG-a-TEC testbed now returns the following:

```
GET nodes?19/description
id:19
mac:20
firmware:2.35
location:14.2375512,45.931374
description:node 19
```

The *location* field contains geographical longitude and latitude, separated with a comma. This information can then be directly used by the TVWS database together with spectrum data, without relying on any external database tables mapping network addresses to coordinates.

Since sensor nodes themselves cannot obtain the location data, the coordinates have to be programmed after each sensor node deployment. The *description* resource handler on the sensor node stores the location data in a section of the non-volatile MRAM present on the sensor node code (SNC) board. The contents of the MRAM can be reprogrammed by issuing a POST request for the *description* resource.

3.1.3 Direct digital synthesis support for VESNA with SNE-ISMTV-868

TVWS database in CREW-TV will use spectrum sensor receivers in the LOG-a-TEC testbed as a distributed spectrum sensor. To test this dynamic component of the database and demonstrate its abilities primary users have to be simulated. To remove the need to manually introduce wireless microphones into the testbed it has been decided to simulate wireless microphone transmissions using the remotely accessible sensor nodes in the LOG-a-TEC testbed as well.

10 nodes in the LOG-a-TEC testbed are equipped with the SNE-ISMTV-868 transceiver that is capable of transmitting a narrow-band signal at frequencies between 779 and 928 MHz. This frequency span includes the upper UHF channels that are used by licensed wireless microphone users. One of more of these nodes will be remotely triggered to act as wireless microphones during CREW-TV experiments and demonstrations.

To create as realistic environment as possible in the testbed it was decided to use the IEEE wireless microphone simulation profiles when testing the CREW-TV deployment (*Chris Clanton, Mark Kankel and Y. Tang, "Wireless Microphone Signal Simulation Method", IEEE 802.22-07/0124r0, March 2007*). This method approximates a radio signal transmitted by a legacy analogue studio wireless microphone (PMSE) using a frequency modulated continuous sine wave. Three operating conditions for the microphone are defined in literature: silent, soft speaker and loud speaker, each with its own FM parameters:

Operating mode	**F_m [kHz]**	**F_{dev} [kHz]**
Silent	32.0	5.0
Soft	3.9	15.0
Loud	13.4	32.6

Table 8: FM parameters for wireless microphone simulation profiles.

Because the SNE-ISMTV-868 transceiver is not capable of transmitting a modulated analogue signal as required by this method, an approximation has been implemented. The transceiver has been configured for a continuous digital frequency-shift keying transmission with 4 symbols (4FSK) and 200 ksymbols/s. This is equivalent to a transmission of a sampled analogue signal with 2-bit precision and 200 kHz sampling rate.

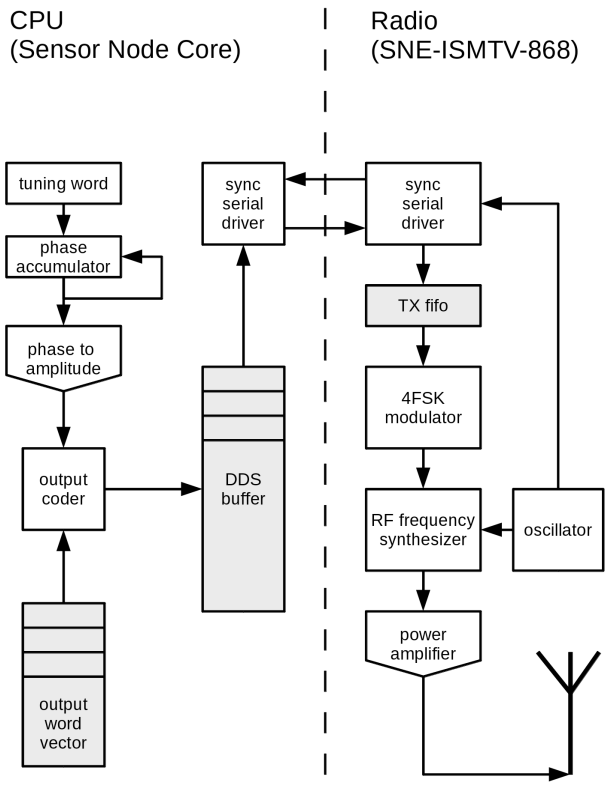


Figure 50: Block diagram of Direct Signal Synthesis on VESNA with SNE-ISMTV expansion

A direct digital synthesis (DDS) algorithm has then been implemented in software on the sensor node's CPU to continuously feed a sine waveform to the transmitter hardware through a synchronous serial bus. Modulation frequency of the modulation signal can be set by adjusting the DDS tuning word in software while the FM deviation can be set by changing the hardware FSK modulator

deviation setting. These two settings allowed us to closely match the FM parameters required for all three wireless microphone simulation profiles.

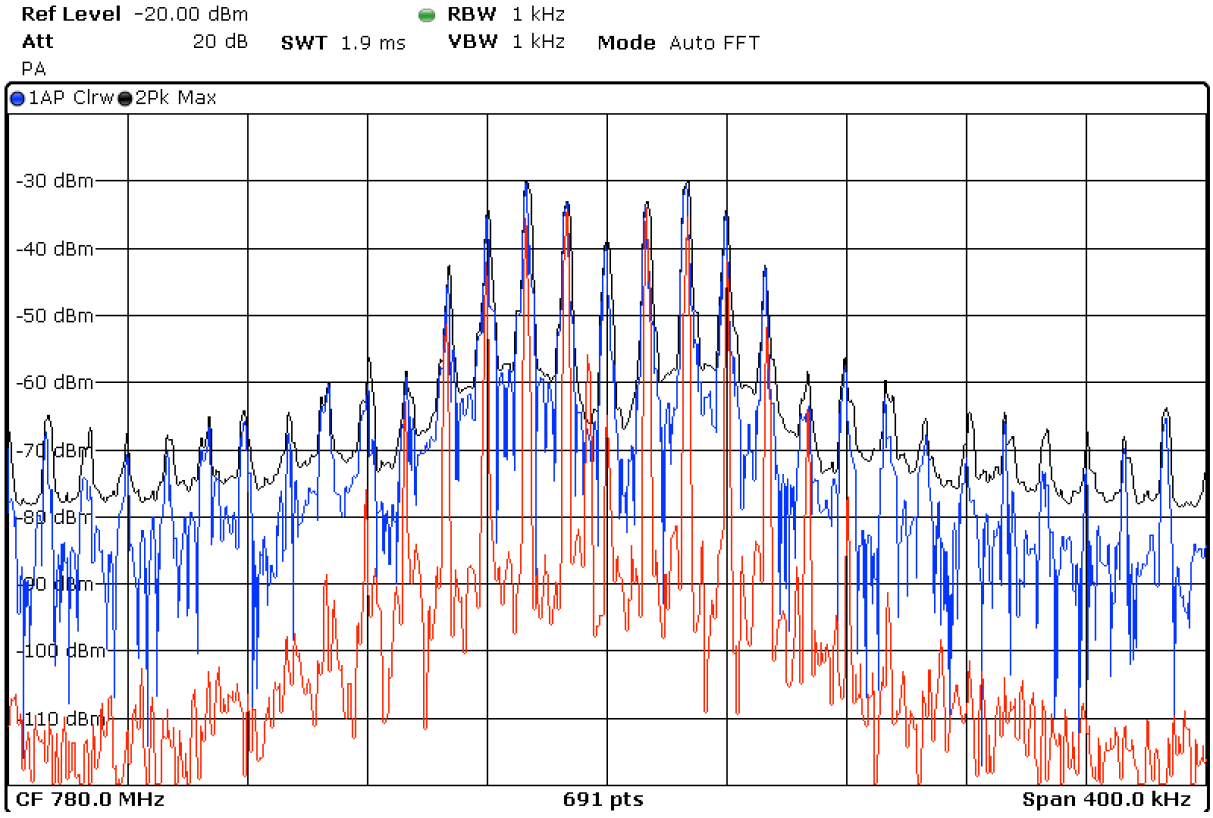


Figure 51: Spectrum of a "loud speaker" wireless microphone simulation signal produced by SNE-ISMTV (blue trace) compared to the same signal produced by a USRP device (red trace)

Wireless microphone simulation profiles have been seamlessly integrated with the existing signal generation API exposed by sensor nodes in the LOG-a-TEC testbed. All 10 nodes with the SNE-ISMTV-868 transceiver have been updated with new firmware supporting wireless microphone simulation. When queried for signal generation profiles, these nodes now report 6 additional signal generation configurations, two for each wireless microphone simulation profile (due to the limitation on the number of channels per configuration, the UHF frequency band had to be split between two different profiles)

```
GET nodes?8/generator/deviceConfigList
dev #0, CC1100, 9 configs:
        cfg #0: CC1100, FM noise, 200 kHz deviation:
                base: 779999908 Hz, spacing: 199814 Hz, bw: 197754 Hz, channels: 256, min
                power: -30 dBm, max power: 12 dBm, time: 5 ms
        cfg #1: CC1100, FM noise, 200 kHz deviation:
                base: 829999924 Hz, spacing: 199814 Hz, bw: 197754 Hz, channels: 160, min
                power: -30 dBm, max power: 12 dBm, time: 5 ms
        cfg #2: CC1100, wireless mic, silent:
                base: 779999908 Hz, spacing: 199814 Hz, bw: 197754 Hz, channels: 256, min
                power: -30 dBm, max power: 12 dBm, time: 5 ms
        cfg #3: CC1100, wireless mic, silent:
                base: 829999924 Hz, spacing: 199814 Hz, bw: 197754 Hz, channels: 160, min
                power: -30 dBm, max power: 12 dBm, time: 5 ms
        cfg #4: CC1100, wireless mic, soft speaker:
                base: 779999908 Hz, spacing: 199814 Hz, bw: 197754 Hz, channels: 256, min
                power: -30 dBm, max power: 12 dBm, time: 5 ms
        cfg #5: CC1100, wireless mic, soft speaker:
```

```
                base: 829999924 Hz, spacing: 199814 Hz, bw: 197754 Hz, channels: 160, min
                power: -30 dBm, max power: 12 dBm, time: 5 ms
        cfg #6: CC1100, wireless mic, loud speaker:
                base: 779999908 Hz, spacing: 199814 Hz, bw: 197754 Hz, channels: 256, min
                power: -30 dBm, max power: 12 dBm, time: 5 ms
        cfg #7: CC1100, wireless mic, loud speaker:
                base: 829999924 Hz, spacing: 199814 Hz, bw: 197754 Hz, channels: 160, min
                power: -30 dBm, max power: 12 dBm, time: 5 ms
        cfg #8: CC1100, 868 MHz SRD band, FM noise, 50 kHz deviation:
base: 863999695 Hz, spacing: 49953 Hz, bw: 49438 Hz, channels: 140, min power: -30 dBm, max
power: 12 dBm, time: 5 ms
```

3.2 Support for EVOLVE

The EVOLVE experiment made use of the wireless nodes in the iMinds w-iLab.t. The testbed framework was used for provisioning the nodes and executing repeatable experiments (OMF). Experiments were executed using the 802.11a/b/g/n wireless cards. Several wireless multi-hop scenarios were set up to test the signalling load of the proposed algorithm.

No additional hardware or software was installed in the w-iLab.t testbed to support the EVOLVE experiment. Technical support was provided during the experiment setup phase regarding the use of the testbed and the hardware of the wireless nodes.

In addition, the EVOLVE project made use of the Iris testbed at TCD to develop an OMF control interface for the Iris software radio framework. Two researchers from WINGS ICT Solutions visited Dublin and used the testbed onsite with technical and design support from TCD researchers.

3.3 Support for CABIN-CREW

Support to CABIN-CREW was provided for the two proposed experiments:

To support experiments with the WMP in the iMinds testbed, four Alix devices equipped with Broadcom Wi-Fi cards were installed in the iMinds testbed. These nodes were integrated in the Emulab environment and several experiments were created to use the nodes as well as one of the USRPs as a measurement device. OMF support was installed on the alix nodes and the network configuration was modified to support OMF experiments. Additional OMF support was provided in person and by e-mail.

For the planned experiments at TUB, three Alix devices equipped with Broadcom Wi-Fi cards where installed in the TUB testbed. Those nodes where integrated in the new OMF environment. It is possible to use them in the experiments.

For the WARP based experiments a temporary setup was installed at iMinds itself to evaluate remote operation capabilities. Documentation was provided to use this setup and two more WARPs were installed in the iMinds testbed and integrated in Emulab. As was the case for the alix nodes, the required experiments were created to allow remote experimentation. The WARP boards can be programmed from the connected Zotac nodes using installed Xilinx LabTools, while the ethernet connection is connected to a gigabit switch and can be connected to any of the 8 available servers where Matlab is available for processing purposes. OMF is not yet supported for the WARP.

3.4 Support for UTH + NICTA

3.4.1.1 Measurement of sensing delay in USRP sensing engine

This section focus on the support we provided for measuring the sensing delay of USRP based sensing engine. The experiment involves two types of USRP's, the USRP N210 and USRP E110.

The USRP N210 belongs to the network series in the USRP product family. It connects to a host computer via fast Ethernet connection, and relies on the host for signal processing. The USRP E110

belongs to the embedded series, it is meant to be used standalone. There is an ARM processor on board of the E110, and hence signal processing can be conducted on the device itself. In case of spectrum sensing, USRP N210 only streams raw samples to the host computer while E110 provides the final result to the host computer. It is interesting to comparing the delay of the two distinctive software radio approach, which is also the main reason that we consider to use the two distinctive types of USRP's.

For the sensing with USRP N210, we make use of the Iris platform. Several Iris components are developed for spectrum sensing. We provide those sensing components and guide lines on how to use them. In the beginning of February, one UTH researcher visited our testbed for a hands-on experience of using USRP on w-iLab.t. Iris platform provides built-in time delay measurement inside its core. For each component there is an average timing report about how much time it consumes at the end of the execution. Therefore we do not need to modify the software especially for the delay measurement.

For the experiment with USRP E110, we had an initial test of the hardware. The E110 boots from a micro SD card, which contains a disk image of embedded Linux with the UHD driver and GNU Radio installed. The first effort we made is to update the disk image, because the factory version of the disk image is out of date and somehow unstable. After that we also explored to use the default UHD driver on the E110 for simple FFT measurement. We passed the knowledge of using E110 together with the hardware to UTH for further measurement.

3.4.1.2 Measurement of the energy consumption and sensing delay of the IMEC sensing engine

The IMEC sensing engine is a compact stand-alone sensing device that combines a low-power sensing ASIC with a wideband reconfigurable analog frontend. The design goal of the IMEC sensing engine was to design a monolithic and rugged device and as a consequence the options for monitoring internal signals are limited. To cater for the requirements of this experiment the various power nets of the sensing engine (digital part and the two nets of the analog frontend) were made available. Some iterations back and forth were required to get the IMEC sensing engine up and running in the UTH lab. Next to the physical transfer of the hardware a software mapping task was required: in order to enable a comparison between the various sensing engines a new firmware mode was mapped on the IMEC sensing engine. The new firmware enables identical functionality over the various sensing engines (max-hold of band scanning sensing through FFT).

4 FIRE Support Actions

Initial contribution to FIRE were listed in deliverable D5.1. This section provides an update to these FIRE support actions. In addition to the activities listed there, following support was given to FIRE by CREW to following non-limitative list of events:

4.1 Attendance to FIRE events

Support was/will be given within the FIRE context by CREW members actively participating in the following non-limitative list of FIRE specific events:

- FIRE week in Poznan (October 2011)
- Future Internet Week in Aalborg (May 2012)
- FIRE engineering workshop in Ghent (November 2012)
- CREW training days in Brussels (February 2013)
- Future Internet Assembly in Dublin (May 2013)
- Future Network and Mobile Summit in Lisbon (July 2013)
- Workshop on Cognitive Radio in Lisbon (September 2013)
- ICT 2013 in Vilnius (November 2013)
- CREW training days 2nd edition in Ghent (January 2014)

An up-to-date overview of future and past events is available directly at the CREW website, via http://www.crew-project.eu/events or http://www.crew-project.eu/pastevents respectively.

Attendance to this FIRE related events often also coincides with presentations and/or publications at these events. They are listed in deliverable D8.4.

4.2 FIRE brochure

CREW contributed to the latest version of the FIRE brochure. The FIRE brochure is distributed during different FIRE events and holds an overview of the FIRE facility projects, and the infrastructure that is made available.

An electronic version of the brochure can be downloaded from the FIRE website: http://www.ict-fire.eu/home/publications.html .

5 Conclusions

In this document we have described the final demand-driven extensions of the CREW federation and FIRE support actions.

We explained how the federated CREW test facilities have been extended with a second set of new functionality which has been defined in a demand-driven and open way based on the gaps identified, for both the internal use cases (WP6) and the experiments of Open Call 2 (WP7).

To this end, we extended the Connectivity Brokerage Framework for better usage in different scenarios and for improved usability in specific cognitive radio scenarios. A suitable database system was selected and a extension with some IEEE 1900.6 compatible parts was made. Implementation of the Connectivity Brokerage Framework was then also investigated and tested for the w-iLab.t and Log-a-tec testbed.

Furthermore, a framework (ProtoStack/ Crime) that allows composing communication services in a dynamic way was proposed and different optimizations were performed within the different testbeds (e.g. OMF, IRIS, GRASS-RaPlat, USRP sensing engine improvement etc.).

To conclude the document, the specific support actions required for the Open Call 2 experiments were also described, as well as a short update on the FIRE support actions.

6 Bibliography

[1] Plets, D., Joseph, W., Vanhecke, K., Tanghe, E. and Martens, L., "Coverage Prediction and Optimization Algorithms for Indoor Environments," *EURASIP Journal on Wireless Communications and Networking, Special Issue on Radio Propagation, Channel Modeling, and Wireless, Channel Simulation Tools for Heterogeneous Networking Evaluation*, vol. 1, 2012.

[2] SUrrogate MOdelling Tool Box, http://sumowiki.intec.ugent.be/Main_Page, last accessed on 27th of Sep, 2013

[3] Rabaey, J et al., "Connectivity Brokerage - Enabling Seamless Cooperation" in *Wireless Networks.* 2010.

[4] Parsa, S. and Banerjee, A., "Design and Implementation Guideline for the Connectivity Brokerage Distributed Repository (CBDR)", Available: https://bitbucket.org/aparsa/connectivitybroker, 2010. Berkeley Wireless Research Center (BWRC).

[5] Polastre, J., Szewczyk, R. And Culler, D., **"**Telos: enabling ultra-low power wireless research.", *IPSN '05: Proc. of the 4th international symposium on Information processing in sensor networks*, Los Angeles, California, US.

[6] Levis, P. et al., "T2: A Second Generation OS For Embedded Sensor.", Telecommunication Networks Group, Technische Universitaet Berlin, 2005.

[7] Ingels, M. et al, **"**A 5mm2 40nm LP CMOS 0.1-to-3GHz multistandard transceiver", in *2010 IEEE International Solid-State Circuits Conference ISSCC*, 2010.

[8] Nicollet, E., Pothin, S. and Sanchez, A., **"**Transceiver Facility Specification.", Wireless Innovation Forum, 2009. "SDRF-08-S-0008-V1_0_0_Transceiver_Facility_Specification.pdf", "http://groups.winnforum.org/p/cm/ld/fid=85".

[9] Latre, S., Van de Meerssche, W., Melis, S., Papadimitriou, D., De Turck, F. and Demeester, P., "Automated management of network experiments and user behaviour emulation on large scale testbed facilities", in *Network and Service Management (CNSM)*, 2010 Niagara Falls.

[10] Fortuna, C., "Dynamic Composition of Communication Services". Ljubljana, Slovenia : Jozef Stefan International Postgraduate School, 2013.

[11] OMF 6 Documentation. [Online] http://omf.mytestbed.net/projects/omf6/wiki/Wiki

[12] Dunkels, A., Osterlind, F. and He, Z., "An Adaptive Communication Architecture for Wireless Sensor Networks", in *Proceedings of the ACM Conference on Embedded Networked Sensor Systems*, 335–349, 2007, Sydney, Australia.

[13] Fortuna, C. and Mohorcic, M., "Dynamic composition of services for end-to-end information transport", *IEEE Wireless Communications Magazine*, 2009, Vol. 16.

[14] Tektronix, Fundamentals of Real-Time Spectrum Analysis,. (Tektronix, 2009), p. 7

[15] Liu et al. EURASIP Journal on Wireless Communications and Networking 2013, 2013:228