RELYonIT
dependability for the internet of things

Research by Experimentation
for Dependability on the
Internet of Things

Funded by the European
Union's Seventh Framework
Programme

D-3.2 – Report on Protocol Selection, Parameterization, and Runtime Adaptation

Grant Agreement no: 317826
www.relyonit.eu

Date: January 15, 2015

Author(s) and affiliation: Felix Jonathan Oppermann, Carlo Alberto Boano, Marcel Baunach, Kay Römer (TUG); Faisal Aslam, Marco Zúñiga, Ioannis Protonotarios, Koen Langendoen (TUD); Niclas Finne, Nicolas Tsiftes, and Thiemo Voigt (SICS)

Work package/task: WP3
Document status: Final
Dissemination level: Public
Keywords: Internet of Things, Protocol Selection, Protocol Parameterization, Run-time Adaption, Optimization

Abstract This is the final report for Tasks 3.2 "Protocol Selection and Parameterization" and 3.3 "Rumtime Adaption of Protocol Parameters". In this deliverable we present our approach and the required tools for protocol selection, parameterization, and run-time adaption. The protocol selection is supported by a decision support system, that provides hints about which protocols are most suited for a specific environment and scenarios. The selected protocols are configured by an automatic parameterization framework that employs mathematical optimization to find protocol parameters that ensure a desired performance under a given environment model. Finally, the run-time adaptation framework allows to provide best effort performance even under conditions that invalidate the original environment model.

Disclaimer

The information in this document is proprietary to the following RELYonIT consortium members: Graz University of Technology, Swedish Institute of Computer Science AB, Technische Universiteit Delft, University of Lancaster, Worldsensing, Acciona Infraestructuras S.A.

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user uses the information at its sole risk and liability. The above referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law.

Contents

1 Introduction . . . 8

2 Protocol Selection . . . 9

2.1 A cookbook for protocol selection . . . 9
2.1.1 Testbeds Used . . . 12
2.2 Description of Protocols . . . 13
2.2.1 Opportunistic Routing Protocol . . . 13
2.2.2 Backpressure Collection Protocol . . . 13
2.2.3 Collection Tree Protocol . . . 14
2.2.4 Evergreen . . . 14
2.3 Evaluation of Protocols . . . 15
2.3.1 Base Case: Static Network . . . 15
2.3.2 Network Size . . . 18
2.3.3 Offered Load . . . 18
2.3.4 Temperature . . . 19
2.3.5 Interference . . . 19
2.3.6 Mobility . . . 20
2.4 Protocol Selection Algorithm . . . 21
2.4.1 Protocol Selection for Use Cases . . . 26

3 Protocol Parameterization . . . 27

3.1 Architecture . . . 27
3.2 Protocol Parameterization . . . 29
3.2.1 Optimization Problem . . . 29
3.2.2 Optimization Strategies . . . 31
Exhaustive Search . . . 31
Simulated Annealing . . . 31
Evolution Strategies . . . 33
3.2.3 Implementation . . . 33
3.3 Run-time Interface of Protocol Parameterization . . . 37
3.3.1 Run-time Interface . . . 38
3.3.2 Representation of Configurations . . . 38
3.4 Integration of New Protocol Models . . . 41
3.5 Evaluation . . . 44

4 Runtime Adaptation . . . 47

4.1 Overview . . . 47
4.2 Reinforcement Learning of Runtime Adaptation Policies . . . 48
4.2.1 State Monitoring . . . 48
4.2.2 Configuration Policy . . . 48
4.2.3 Learning Mechanism . . . 50
4.2.4 Utility Function . . . 50

4.2.5 Simulation Framework for Off-Line Learning . . . 51
4.3 Cooja Simulator Plugin . . . 51
4.3.1 Episode Execution . . . 51
4.3.2 Policy Execution . . . 52
4.3.3 Policy Updates . . . 52
4.3.4 Generation of Environmental Parameters . . . 53
4.4 Node Runtime . . . 55
4.4.1 Runtime Adaptation Process . . . 55
4.4.2 Configuration Distribution Protocol . . . 55
4.5 Evaluation . . . 56
5 Conclusions . . . 58

List of Figures

2.1 Protocol feedback loop . . . 10
2.2 Protocol selection . . . 12
2.3 Base Case: A small static network with offered load of two packets per second. 15
2.4 Network size variations. . . . 16
2.5 Offered load variations on Indriya. . . . 16
2.6 Offered load variations on Twist. . . . 17
2.7 Temperature variations using TempLab. . . . 17
2.8 Interference on Twist. . . . 20
2.9 Mobility: Mobile root node with two different speed variations. . . . 20

3.1 Architecture of the parameterization framework. . . . 28
3.2 Overview of the system architecture of the protocol parameterization tool . . 34
3.3 Internal representation of requirement specifications . . . 35
3.4 Model interface and parameter types . . . 36
3.5 Time/quality trade off for different optimization strategies. . . . 45
3.6 Probability of finding the optimal solution within a given time. . . . 46

4.1 The illustration shows what happens when an increase of the number of neighbors causes the system state to change (left side). This change will make the policy perform an action that will change the send interval of the application (right side). . . . 49
4.2 The RL plugin for Cooja contains the reinforcement learning implementation. It is supported by a Jython module that executes the Python scripts in the simulation file. These scripts compute the utility that is used by the learning process. . . . 52
4.3 Interaction between the RL plugin and the emulated node are based both on internal state of the nodes such as the policy array and external state such as communication and log output from the nodes. . . . 53
4.4 Screenshot of Cooja running the RL plugin. The network topology is illustrated in the upper left corner. The utility graph illustrating the learning progress is shown to the right. In the bottom part of the picture are the RL-plugin control window and the serial output from the nodes. . . . 54
4.5 Learning a configuration policy in a single-hop network running the RELYonIT smart parking application. The network is disturbed by a semi-periodic interference generator to simulate a harsh environment in which the runtime assurance would have been triggered. . . . 57

List of Tables

2.1 Summary of Results: A protocol's breaking point is marked with ✗. A protocol outperforming every other protocols in a scenario is marked with ✓. A protocol is considered best for a given scenario if it outperforms other protocols and worst if it breaks in that scenario. 14
2.2 Delivery ratio: The percentage of packets delivered successfully at the root node. 22
2.3 Ratio of duty cycle and the packet delivered, i.e. $\frac{dr}{dc}$ 22
2.4 Throughput: Average number of data packets received at the root node in a second. 23
2.5 Delivery ratio preferred: Protocols ranking with $W_1 = 0.8$, $W_2 = 0.1$, and $W_3 = 0.1$ 23
2.6 Energy consumption preferred: Protocols ranking with $W_1 = 0.1$, $W_2 = 0.8$, and $W_3 = 0.1$ 24
2.7 Throughput preferred: Protocols ranking with $W_1 = 0.1$, $W_2 = 0.1$, and $W_3 = 0.8$ 25

Executive Summary

A central component of the RELYonIT approach are software tools to automatically select, parameterize, and adapt Internet of Things protocols. This deliverable describes the software artifacts created within the project, that allow the user to select protocols and to generate and parameter configurations based on models of the scenario and an abstract specification of dependability requirements. The three main tasks are addressed individually, but the interaction of the individual tools is also covered.

A first step to build a dependable Internet of Things application is the selection of suitable protocols. In RELYonIT we support this task with a decision support system that, based on a careful analysis of different protocols, provides hints about which protocols are most suited for a specific environment and scenarios. We compare three state-of-the-art data-collection protocols under six different properties: temperature, interference, mobility, high density, large scales and high offered load. Furthermore, based on the insights obtained from our extensive experiments, we propose a new data collection protocol called Evergreen, which overcomes some of the limitations of the state of the art.

To optimally meet the challenges of an environment and deliver the desired performance, most protocols require a careful tuning. The selection of good parameter settings usually requires good knowledge of the employed technology and the specifics of the environment. In RELYonIT we reduce the involved challenges by providing a tool that automatically selects near-optimal parameter values. The tool employs environment models based on previously collected environment traces, platform models that characterize the platform response, and models of the employed protocols. The selection is guided by a formal specification of the user requirements employing the format introduced in D-3.1 [18]. Output of the tool is a protocol configuration that can be deployed along the application. An run-time component handles configuration of the protocols and enables the use of several situation-dependent configurations.

Dependable operation should be ensured even under unexpected deterioration of the environment. As the environment models depend on empirically data collected prior to the deployment, it is possible that the environment changes in unpredicted ways, which invalidates the model assumptions. In this case, the previously generated configuration may be invalidated and is unable to ensure the expected performance anymore. Situations like this are reported to the user by the run-time assurance system developed in task T1.4 [6], but the user reaction may require some time. To prevent a severe deterioration of the performance and dependability of the application, a run-time adaptation framework tries to adapt the parameter values to the new environment in a best-effort fashion.

1 Introduction

This deliverable describes the protocol selection, parameterization, and adaptation that enables RELYonIT to deliver Internet of Things applications that still operate according to user-specified performance requirements even under adverse environmental conditions.

Protocol selection employs a "cook book" approach to guide the selection of suitable communication protocols based on the properties of the targeted application. Different protocols often excel in one environment or task while they deliver a poor performance under different conditions. Our selection process enables a user to make the right choices without the need for an extensive background in wireless sensing or wireless communication. This component is described in detail in Chapter 2.

To ensure an optimal performance of the selected protocols, they also need to be tuned to well match the environment of the targeted application. Our protocol parameterization provides a framework for the automatic configuration of protocols based on user-provided dependability requirements and models of the environment, the platform response and the protocol behavior. These models were developed as part of WP1 and WP2. In addition, this necessitates a specification language for the dependability requirements. However, dependability requirements may vary across the lifetime of an application, hence, these specification techniques need to be integrated with existing programming models and languages for WSAN. This was the subject of Task 3.1 and an extensive description can be found in deliverable D-3.1 [18]. The actual parameterization process and the required tools are described in Chapter 3.

Finally, the run-time assurance developed in Task 1.4 and described in deliverable D-1.3 [6] may detect changes of environmental models during the operation of the deployed system. Depending on the application requirements, it may not be possible to halt the application in order to deploy a new set of protocols. With the help of run-time adaptation, however, the parameters of the protocols may be adjusted during runtime without halting application execution. Chapter 4 is concerned with the problem of selecting appropriate protocol configurations to reflect a changed environmental model, and installing these configurations into a running network.

2 Protocol Selection

Protocol selection for Internet of Things applications is a *very* complex task. Selecting the "right" protocol among the many options available is difficult not only for the general user, but also for the expert in the domain. The reasons for this complexity are three-fold: (i) the various scenarios IoT networks are exposed to, (ii) the different requirements of the applications (some want to maximize delivery rate, while others may want to maximize throughput or minimize energy consumption) and (iii) the limited resources of the devices in terms of energy, memory, communication, and computation power. These three factors have lead to the implementation of application-specific protocols, where the methods are tailored to suit the unique triple consisting of <scenario, requirements, resources>. If resources were abundant, we could design a general-purpose protocol. Unfortunately that is not the case for Internet of Things applications, and hence, there is no one-size-fits-all protocol.

The lack of a general-purpose protocol puts the end user in a great dilemma. The user may not be able to design a new protocol to suit her needs, and she will need to choose the best option among the alternatives. As stated before, this task is not simple. In this chapter, we propose a methodology for selecting protocols, and use this methodology to provide a protocol-selection framework for three of the most popular collection protocols available in the literature. We also use our insights to design a *pseudo* general-purpose protocol called Evergreen, which will hopefully facilitate the Internet of Things deployments.

2.1 A cookbook for protocol selection

The process of selecting the "best protocol" for a particular scenario requires a *systematic approach* to construct a decision table. That is, a table (or method) that will allow us to find the protocol that has the closest performance to our needs. As described next, the systematic approach we propose for selecting a protocol requires four basic steps. The first three steps consist of identifying the properties of the environment (mobility, temperature, interference), identifying the metrics of interest (delivery rate, throughput, energy consumption), and evaluating different network parameters (offered load, scalability, density). These three steps and their interactions are depicted in Figure 2.1. With this information, the final step is to use a weighted average method to identify the best protocol for the given circumstances.

Initially, a user will provide a scenario where to deploy the network and the expected performance. For example, a user may need to monitor a factory and obtain information every second from each machine. After the scenario of interest is defined, the first step is to identify its properties.

Step 1: Identifying the properties of the environment. This step is required because environmental conditions are usually outside the user's control and they can have a dramatic

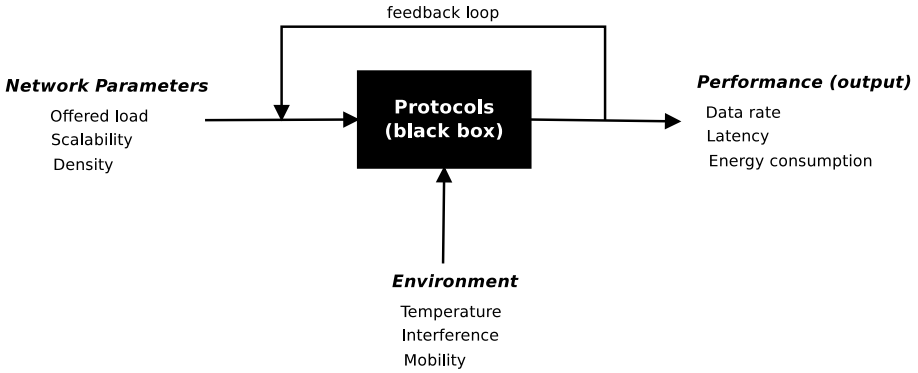


Figure 2.1: Protocol feedback loop

impact in the underlying communication structure of the network. We identified three important environmental properties affecting the dynamics of the network.

1. *Temperature*, which changes the connectivity among nodes due its negative effect on link quality.
2. *Interference* – like temperature – affects the link quality, but it does so at a much faster rate (milliseconds as compared to several tens of minutes in the case of temperature).
3. *Mobility*, which leads to the most dramatic changes in terms of the underlying communication structure.

Once these properties have been identified and quantified, the user should use a testbed or scenario that closely resembles these environmental conditions to test the protocols. Else, the resulting performance may not be the same to what will be observed in the final deployment.

Step 2: Identify the metrics of interest. In principle, we would like a network to provide as much data as possible with as high throughput as possible and lasting for as long possible. There are three de-facto metrics used to capture this behavior:

1. *Delivery ratio*, which is the fraction of packets arriving at the sink(s) compared to the total number of packets sent by the sources.
2. *Throughput*, which is the number of data packets received at the sink (root node) in a second.
3. *Energy consumption*, which is usually captured by the *duty cycle* of the radio.

More often than not, these three metrics present trade-offs. For example, a higher delivery rate or high throughput, usually requires a higher energy consumption. In this report we focus on the trade-off between delivery rate and duty cycle (energy). At the end of this chapter we show a method consisting of a single parameter that allows the user to put more "weight" on

the preferred metric.

Packet duplicates. Besides the three metrics mentioned before, in our evaluation we also measure the number of duplicated packets arriving at the sink. Packet duplicates is not really a network metric, but we use it because it gives important insights on the operation of various protocols. When duplicates increase, the bandwidth saturates. This can affect negatively the delivery ratio, throughput and energy consumption. In some cases, when the duplicates are too high, the network collapses.

Step 3: Evaluate protocols. The first and second steps of our approach provide *hard* requirements. The properties of the scenario (Step 1) are a hard input, and the desired performance (Step 2) is a hard output. We can not change them. There is however a flexible input that allows the user to have a deeper insight into the performance of the system: the properties of the network. Once the scenario of interest has been assessed (Step 1) and the metrics quantified (Step 2), the user can start testing protocols. We identified three general properties that affect the performance of protocols but that are under the control of the user.

1. *Scalability*, which is related to the size of the network in terms of number of nodes.
2. *Density*, which is defined as the average number of neighbors observed by all nodes.
3. *Offered load*, which is the total number of packets generated by the sources per unit of time.

Scalability and density are related to the diameter and average degree of the communication graph, which are known to be two central properties in graph analysis. The offered load is a general parameter that allows the user to control the amount of information flowing in the network. These flows should be high enough to provide the desired performance, but low enough to avoid saturating the network.

Step 4: Select protocol. We propose a weighted average method to select the right protocol. In this method, the users assign weights to the metrics identified in Step 2 (depending on their relative performance), and then, use the information collected in Step 3 to identify the protocol with the best performance according to the environmental parameters identified in Step 1. This method is explained in detail at the end of this Chapter.

Protocol evaluation and the exponential growth of the solution space. The most important hurdle to overcome in a protocol selection process is the exponential growth of the evaluation space. There are many protocols at the Data Link and Network Layers. Evaluating all the potential combinations of protocols at both layers under various network properties can easily become a large and time consuming problem. Consider for example a user willing to evaluate 2 protocols at the Data Link Layer, 4 protocol at Network Layer, and 3 operational levels (high, medium, low) for each of the six 'inputs': density, scalability, traffic patterns, temperature, interference and mobility. This evaluation, which may not be even considered a thorough one, would imply 144 experiments $(2 * 4 * 3 * 6)$! This could demand many months of intensive work and would not even include the potential permutations of inputs; for instance, different

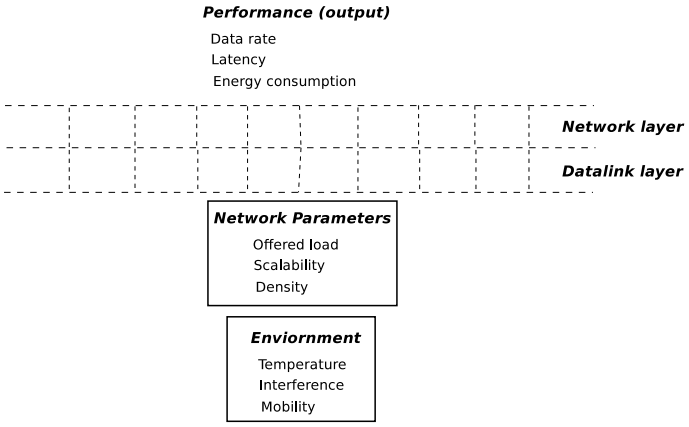


Figure 2.2: Protocol selection

combinations of temperature, interference and mobility levels. While there is no clear solution to this problem, having some expert knowledge can reduce the solution space by starting with some educated guesses. Examples of educated guesses done in our evaluation are given next.

Our overall approach and the focus of our evaluation at the Network Layer. Figure 2.2 summarizes our approach. We consider three different *environmental properties*: temperature, interference and mobility. As mentioned before these are hard conditions, i.e. they are outside the user's control. As for the *network properties*, we evaluate the network size (diameter of graph), node density and offered load. The offered load plays a key role on the number of *packet duplicates*. At the *MAC Layer*, we use BOX-MAC, which is a widely used protocol leveraging a technique called Low Power Listening. At the *Network Layer*, we focus on three protocols that have different design principles: CTP, which targets networks with low dynamics; ORW, which is designed to overcome dynamics with a low-overhead; and BCP, which is designed to maximize the capacity of the network but at a relatively high overhead. The selections made at the MAC and Network Layers are educated guesses based on our expertise in the area.

Our decision to focus the protocol selection process at the Network Layer is twofold. First, in wireless networks, multi-hop distributed communication is not well understood from an analytical standpoint, and hence, empirical approaches are even more important (as opposed to say MAC protocols for which analytical models are better understood). The second reason is completeness. Due to the first reason, most of our new models and protocols have focused at the MAC Layer (D-2.3). To provide the user with a complete view of the network stack, we decided to apply our protocol selection approach to the Network Layer. That said, our *steps* are general enough to be applied to MAC Layer protocols as well.

2.1.1 Testbeds Used

Ideally one would test all protocols in-situ as part of real deployments, but that is practically infeasible *and* prohibits the controlled change of the environmental parameters, so instead we studied them on testbeds except in case of mobility, for which we used simulations. We have used three different testbeds: Indriya [7] with 101 TelosB motes, TempLab [4] with 17

Maxfor MTM-CM5000MSP motes (TelosB replicas), and Twist [13] with 88 TmoteSky motes[1] The later is part of the testbed available within the FIRE initiative. All of them run the TinyOS operating system. For the mobility experiments we used Avrora [23], an instruction-level accurate micro-controller simulator. For each experiment we configured all nodes to send data to the sink. Data packets carry a payload of four bytes, and are periodically injected into the network with some randomization to avoid peak loads and associated collisions at the physical/link layer. For each experiment we log the packet delivery ratio at the sink, and the average energy consumption of the nodes by means of soft metering (i.e. recording the actual duty cycle of the radio).

2.2 Description of Protocols

This section briefly describes the three protocols used in our evaluation: ORW, BCP and CTP. These three protocols represent the state-of-the-art in data collection for sensor networks.

2.2.1 Opportunistic Routing Protocol

Recently, the Opportunistic Routing Protocol for WSN (ORW) was proposed [15]. In ORW, instead of sending a data packet to a specific neighbor, a node n sends a packet to *any* neighbor that is closer to the sink that itself. This mechanism is called anycast and works as follows: n sends a packet containing its routing cost to the sink; if a neighboring node m has a lower routing cost than n, then it acknowledges the packet to n and continues the forwarding process; if on the other hand, the routing cost of the neighbor m is higher, the data packet is silently dropped.

The advantage of ORW is that the end-to-end delay is reduced significantly because node n does not have to wait for a specific neighboring node to wake-up to receive the data packet. To the contrary, the very first neighbor that wakes up from sleep and has a smaller cost to the root node, replies back with an acknowledgment. Thus, at each hop several milliseconds are saved, accumulating into a significant amount of time over the whole length of the data path. Another advantage of the ORW forwarding scheme is that data packets flow through multiple paths instead of the majority of packets taking the same path to the root node. These diverse paths increase network lifetime by distributing the forwarding workload among different nodes and also avoid congestion in the network.

2.2.2 Backpressure Collection Protocol

Typically, in collection protocols each node maintains a data queue to forward packets. Instead of forwarding a data packet immediately, packets are first placed in the forwarding queue when received from a neighboring node or from the application running on the given node. The Backpressure Collection Protocol (BCP) forwards packets to the neighbor with the shortest queue. This method maximizes the capacity of the network [17]. The method can be succinctly described in the following manner. Considering that the sink 'absorbs' all packets, the following gradient is formed by the nodes' queues: the closer the node is to the sink, the shorter its

[1] Twist normally comprises 102 nodes, but at the time of writing only 88 were available due to construction work at the facility.

	Static	Mobility	Network Size	Offered Load	Temp.	Interf.
CTP		✗				
ORW	✓			✗	✓	
BCP		✗	✗			
Evergreen		✓		✓		✓

Table 2.1: Summary of Results: A protocol's breaking point is marked with ✗. A protocol outperforming every other protocols in a scenario is marked with ✓. A protocol is considered best for a given scenario if it outperforms other protocols and worst if it breaks in that scenario.

queue should be. Thus, each node n sends information to the sink by forwarding packets to the immediate neighbor m that has a shorter queue than its own and also the shortest among its neighbors. BCP has one differentiating characteristic in contrast to typical collection protocols: nodes do not maintain end-to-end routing paths. The lack of end-to-end paths results in fast route convergence which is especially useful if nodes are mobile. BCP is shown to outperform CTP significantly when the root node is mobile [17].

2.2.3 Collection Tree Protocol

The Collection Tree Protocol (CTP) is a well-known and widely used collection protocol [12]. Unlike ORW, CTP is a deterministic protocol where a node always opts for the shortest route towards a root, based on link quality estimations. CTP uses adaptive beaconing (a.k.a Trickle timer) instead of sending periodic control messages, thus reducing protocol overhead [12, 24]. Furthermore, CTP also introduced data-path validation to avoid routing loops and uses an accurate link quality estimation (ETX) to measure the quality of each link [11]. Overall CTP works very well if the network is stable, and is shown to deliver on average more than 90% data packets while using only 3% duty cycle under moderate offered loads [12]. However, CTP's performance deteriorates when the nodes are mobile or when the network links are volatile with rapidly changing link qualities [17].

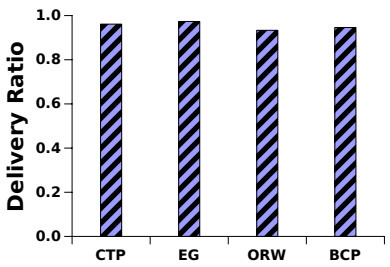
2.2.4 Evergreen

Initial experimentation with the three collection protocols just described showed that there was ample room for improvement, because each individual protocol would break, i.e. perform badly, in some specific scenario. We therefore decided to take a step further, to design the more resilient protocol Evergreen as described in D-2.1 [27]. As we will show in Section 2.3, CTP breaks when nodes become mobile, ORW has issues with increased traffic loads, and BCP's performance deteriorates when the network diameter grows. Evergreen does not strive for being the best in one situation, but to show resilience to a large number of (environmental) factors.

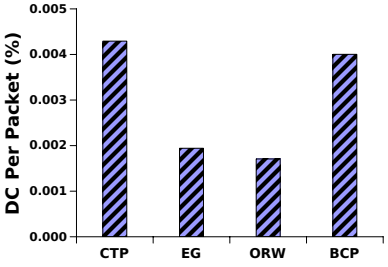
2.3 Evaluation of Protocols

This section studies how the four collection protocols (Evergreen, CTP, ORW, and BCP) perform in the five scenarios of (i) offered load, (ii) network size, (iii) temperature, (iv) interference, and (v) mobility. To this end, three testbeds are used as discussed in Section 2.1.1. Each experiment was executed for at least 40 minutes. In all the experiments only a single root node, located at one edge of the network, is used as the destination of data packets. Every non-root node of the network generates data packets in a uniformly distributed interval. To take the number of nodes in the network into account, the reported offered load is the aggregated number of packets injected into the network. That is, the offered load is calculated as the median of the interval used to send data packets by the nodes, multiplied by the total number of non-root nodes. Most of the experiments were carried out with an offered (aggregated) load of two packets per second. In all cases the duty cycle period of the default TinyOS MAC protocol was set to one second.

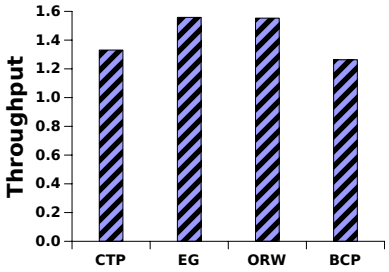
This report summarizes the results of hundreds of experiments carried out during a whole year. For these experiments, the source code of CTP and ORW was acquired from the respective public repositories, however, BCP's code was acquired directly from the authors of the original report. During this research some bugs were also fixed in the existing protocols, and reported to their original authors [14].



(a) Delivery Ratio



(b) Duty Cycle



(c) Throughput

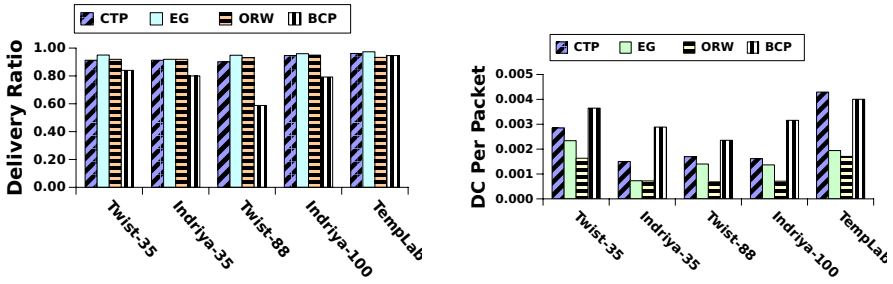
Protocol	Duplicates
CTP	1
EG	0
ORW	2
BCP	1

(d) Duplicates

Figure 2.3: Base Case: A small static network with offered load of two packets per second.

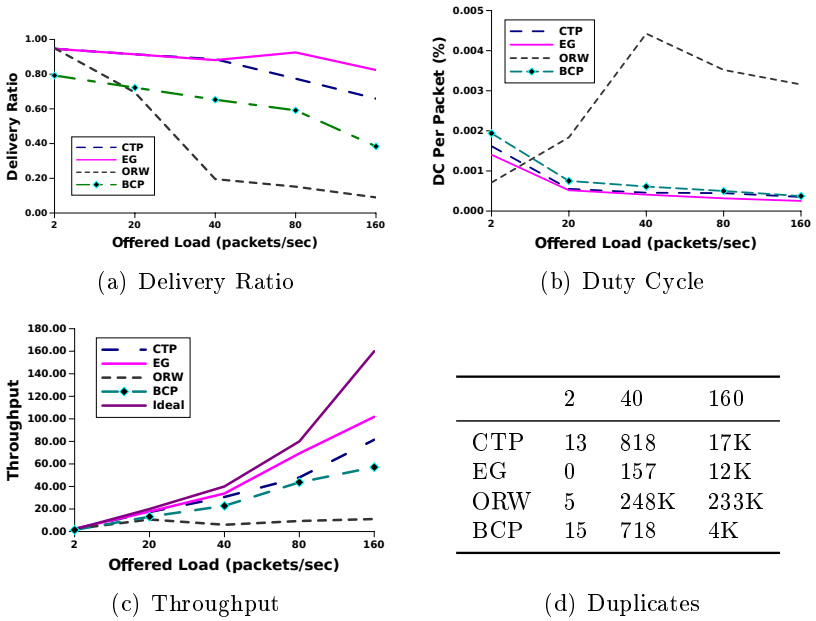
2.3.1 Base Case: Static Network

We start with a base case in which every protocol should perform reasonably well. We carefully select our base case such that it has no characteristic leading to inadequate performance of a



(a) Delivery Ratio (b) Duty cycle

Figure 2.4: Network size variations.



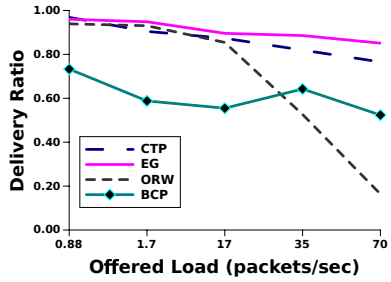
(a) Delivery Ratio (b) Duty Cycle

(c) Throughput

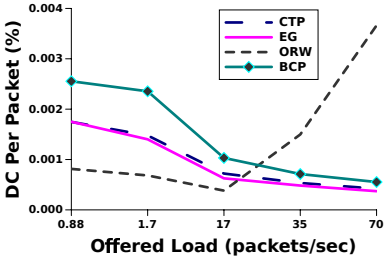
	2	40	160
CTP	13	818	17K
EG	0	157	12K
ORW	5	248K	233K
BCP	15	718	4K

(d) Duplicates

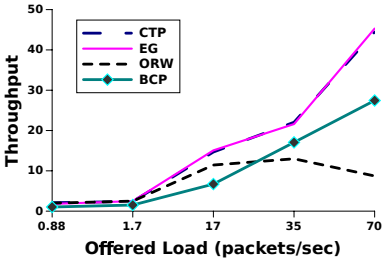
Figure 2.5: Offered load variations on Indriya.



(a) Delivery Ratio



(b) Duty Cycle

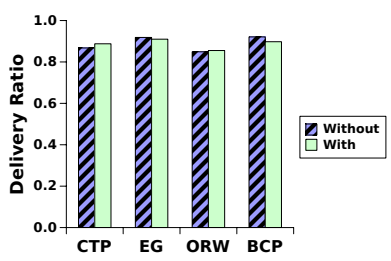


(c) Throughput

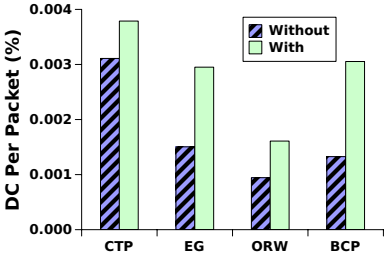
	0.88	17	70
CTP	0	104	490
EG	2	22	1400
ORW	2	8K	108K
BCP	10	759	7K

(d) Duplicates

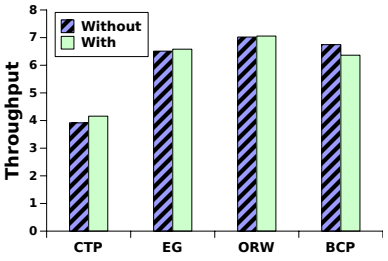
Figure 2.6: Offered load variations on Twist.



(a) Delivery Ratio



(b) Duty Cycle



(c) Throughput

	Without	With
CTP	0	2
EG	1	0
ORW	25	9
BCP	1	7

(d) Duplicates

Figure 2.7: Temperature variations using TempLab.

protocol. This is why we chose a small network, TempLab, with only 17 nodes and an offered load of two packets per second. Furthermore, the network does not experience any noticeable interference or temperature variation during the course of the experiment. It is shown in Fig. 2.3 that all four protocols achieve a delivery ratio of more than 95%. However, the major difference is in the average radio duty cycle required by nodes to forward packets to the root node. ORW outperforms all protocols, showing a duty cycle 60.2% smaller than the one of CTP. As explained in Section 2.2.1, a mote using ORW uses anycast routing thus reducing the time it needs to forward data packets. Based on these results, we marked ORW as the best performing protocol for this scenario in Table 2.1. To our surprise, Evergreen, being a deterministic protocol, has a similar energy consumption as ORW. To be precise, Evergreen consumes 54.8% less energy than CTP in terms of average duty cycle, and only 13.7% more in comparison to ORW. We determined that Evergreen's significant reduction in energy consumption as compared to CTP, is due to its bursty packet forwarding and smart control messages. Furthermore, compared to CTP, Evergreen has the highest throughput among all protocols, delivering 17% more packets in the same time. This is because Evergreen has many optimizations to reduce the time between consecutive packets. Finally, Fig. 2.3 also shows that the number of duplicate packets received at the root node during the 40 minute long experiments is also insignificant under these favorable conditions as each protocol generates very few duplicates that have no effect on the protocol performance. The subsequent section builds on this base case to stress the protocols, in order to find their breaking point.

2.3.2 Network Size

We have used three testbeds of different diameters and number of nodes (i.e., 17, 88 and 101). Furthermore, we have divided the two bigger testbeds, Twist and Indriya, into two smaller testbeds of 35 closely located nodes. Thus in total the stress testing of the network size variation is carried out on five testbeds. An offered load of 2 packets per second is used for each experiment. For Evergreen, CTP, and ORW changes in network size do not result in any significant performance deterioration as their delivery ratio remains above 90% as shown in Fig. 2.4. However, BCP experiences more than 30% reduction in delivery ratio on bigger networks as compared to the 17 node TempLab network. This is because BCP uses the number of queued packets to construct a gradient and nodes at the perimeter of networks with large diameters cannot hold enough packets; the induced queue overflows lead to random forwarding of packets in turn aggravating the problem as packets "float" longer in the network, effectively taking up more space in the forwarding queues. One solution is to increase the forwarding queue size as the network diameter increases but this solution is not generic and will not scale in a very large network of memory-constrained wireless motes. We expect BCP's performance to further deteriorate and eventually break as the network size increases.

2.3.3 Offered Load

To measure the effect of offered load we used five offered load variations on the Indriya and Twist testbeds for each of the four collection protocols. Each protocol's performance deteriorates with the increase of offered traffic load, however, ORW is effected the most. It is shown in Fig. 2.5 and Fig. 2.6 that ORW experiences a delivery ratio reduction of 90% and 82% on Indriya and

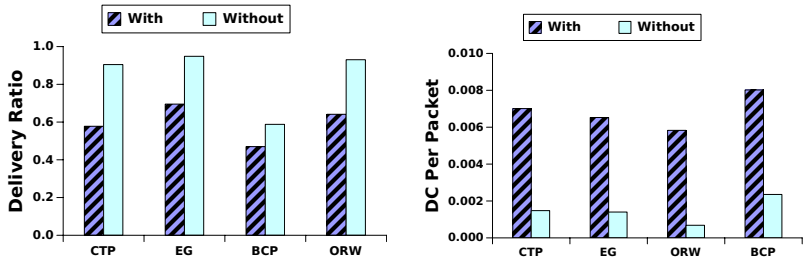
Twist testbed respectively. This is because ORW exhibits a drastic increase in packet duplicates due to the anycast routing scheme used, as explained in Section 2.2.1. These duplicate packets cause congestion in the network, resulting in dropping many data packets as well as increasing the energy consumption of the motes. As shown in Fig. 2.5, and Fig. 2.6, ORW's average duty cycle increases by up to 342%. Thus we believe that the use of ORW should be avoided in high offered load situations. In contrast, Evergreen outperforms the other protocols significantly under high offered load. This is mainly because of two reasons: Evergreen's forwarding queue management employees a strategy to avoid paths with motes having forwarding queues filled almost to their capacity. Furthermore, Evergreen is optimized to increase the packet forwarding speed, thus reducing the chances of a forwarding queue being overflowed due to high offered load. Finally, it is shown in Fig. 2.5 and Fig. 2.6, that Evergreen has a delivery ratio of more than 25% as compared to it peers under high offered load situations while at the same time consuming the least amount of energy. Based on these results, we marked Evergreen as the best performing protocol in the high offered load scenario in Table 2.1, while we marked ORW as a protocol that breaks.

2.3.4 Temperature

In this subsection we present temperature variation effects using the TempLab testbed. TempLab takes existing temperature traces recorded in outdoor environments over a long time period as input and reproduces them in a much shorter timer period in the Lab settings using infrared heating lamps [4]. For our experiment, a 40 minute long temperature trace was selected that emulates a very fast temperature variation of 1.4 °C per minute. We have recorded each protocol's performance with and without the use of the heating lamps. It is shown in Fig. 2.7, that each collection protocol copes well with temperature variations since even in the lab settings those variations take several minutes to take place, which is slow enough for a routing protocol to adopt. Thus, we have not recorded any significant changes in the delivery ratio. However, the average radio duty cycle to forward a packet to the root node increased significantly with temperature variations as shown in Fig. 2.7. This is because a node has to find new data paths to avoid heated nodes, which could lead to extra control messages being generated in the network. ORW scales better than the other protocols, because unlike CTP and EG, ORW does not use the Trickle timer or other control messages to cope with the network changes, thus it conserves energy. It is shown in Fig. 2.7 that ORW has the smallest increase in the average duty cycle as compared to the other three protocols. Therefore, we have we marked ORW as the best performing protocol in this scenario in Table 2.1.

2.3.5 Interference

We have used the JamLab[3] software to mimic Wi-Fi interference in the Twist network. Four nodes of Twist run JamLab, whereas the rest of the nodes run the collection protocol under evaluation. Instead of having continuous interference, the jammers are switched on and off for a period of five minutes, to notice how a given protocol copes with the sudden influx of interference. Two sets of experiments were carried out. In the first set, each protocol was evaluated without using JamLab whereas in the second, each protocol was evaluated with the periodic jamming. The summary of results with and without interference are shown in Fig. 2.8.

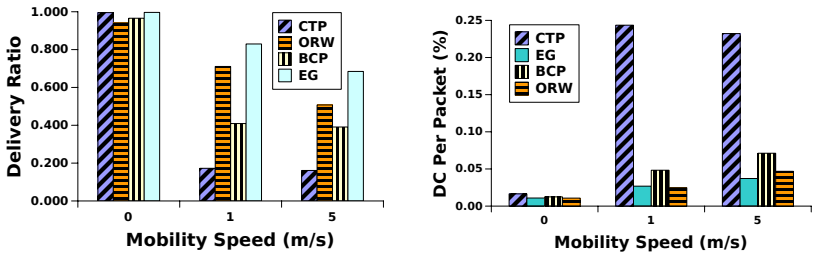


(a) Delivery Ratio (b) Duty Cycle

	Without	With
CTP	0	17
EG	2	44
ORW	2	3323
BCP	10	126

(c) Duplicates

Figure 2.8: Interference on Twist.



(a) Delivery Ratio (b) Duty Cycle

Figure 2.9: Mobility: Mobile root node with two different speed variations.

Evergreen has the highest delivery ratio in the network with interference, followed by ORW. Furthermore, Evergreen has the smallest increase in energy consumption while coping with interference as good as ORW. Therefore, we marked Evergreen as the best performing protocol in this scenario in Table 2.1.

2.3.6 Mobility

Mobility is desirable for supporting a variety of Internet of Things applications. We have used the Avrora simulator [23] to evaluate the performance of collection protocols under mobility, as the three testbeds used in this report did not have mobile nodes available for their regular users. A 6×6 grid network of 36 nodes was used, in which the root node is initially located at the center of the network. What is evaluated is how each collection protocol behaves when the root is mobile. We consider root mobility an extreme scenario to stress test collection protocols as the whole collection tree has to be readjusted whenever the root changes its position. Three sets of experiments were conducted for each protocol with root mobility speeds of 0, 1 and 5 units using the random waypoint mobility model [1]. When the root node is mobile, CTP experienced high delivery ratio reduction as well as a drastic increase in the energy consumption as compared to the scenario where the root node is static. It is shown in Fig. 2.9, that CTP has a delivery ratio reduction of 83.81% and a 11.5 times increase in the average duty cycle to transmit a single packet. This is because CTP's root node cannot detect mobility, so it does not announce itself as it moves to a new location, leading to a stale collection tree. In contrast, Evergreen has a mechanism for the root node to detect mobility and announce itself to keep the

collection tree up to date. Fig. 2.9 shows that Evergreen comprehensively outperforms all the other protocols under the mobile root node scenario. Thus we have listed Evergreen as the best performer and CTP as the protocol that breaks in this scenario in Table 2.1. To our surprise, ORW also performs reasonably well in the mobile root node scenario. ORW uses opportunistic routing, where packets are transmitted to the next available (awake) mote with the slightest route improvement. Unlike CTP a node does not have to announce its existence to receive a data packet; instead a node can acknowledge any packet that it hears during anycast. The root node takes advantage of this by accepting any packet it can hear, also when the routing information in the packet has become outdated due to movement. With CTP a root node will not receive such packets as they are explicitly addressed to some other node. Evergreen also has unicast routing, however unlike CTP, Evergreen has a mobility detection mechanism that allows a node to advertise itself as it moves in the network, keeping the routing tree always up-to-date.

2.4 Protocol Selection Algorithm

Our protocol selection method focuses on the trade-off between delivery ratio, energy consumption and throughput. In principle, we would like to get as many packets as possible as fast as possible, using very little energy.

Before proceeding it is important to discuss two points about our metrics. First, a commonly used metric to combine energy consumption and delivery ratio is to divide the delivery ratio by the duty cycle, which indicates the average amount of energy required to send a single packet to the sink. Using this metric has a problem however. Considering a scenario with 100% delivery rate and 10% duty cycle and an scenario with 30% delivery rate and 3% duty cycle during the same period of time. Both scenarios would have the same $\frac{dr}{dc}$ value, but the first option is preferable because the delivery ratio is the most important metric in data collection protocols. Second, in the project we have focused on latency as a metric for delay and not throughput. We did this because in testbeds it is easier to measure throughput in a more accurate manner. While delay and throughput are related (a lower delay leads to a higher throughput), they are not exactly the same. Our equations and normalization method are however the same and applicable in case latency is used.

To accommodate for this type of outcomes we use a weighted average methodology. Table 2.2 lists the delivery ratio (dr) of each protocol under the various environmental and networking conditions. Whereas, Table 2.3 presents the delivery ratio divided by the duty cycle ($\frac{dr}{dc}$) and Table 2.4 lists the throughput (tp) of each protocol. Our method combines the information in these three tables. At first glance it may look redundant to consider the delivery rate twice, one on its own and also as part of the $\frac{dr}{dc}$ metric (energy). But as explained before, we decided to do this because measuring energy on its own is not as relevant as connecting it to its delivery ratio.

Our method proceeds in the following manner. For each type of scenario x_i, where i= 1 (static), 2 (mobile), 3 (size), 4 (load), 5 (temperature), 6 (interference); we first normalize the dr, $\frac{dr}{dc}$ and tp metrics. For instance, for the static scenario, the delivery rate of Evergreen is the highest and we set it to be the baseline, equal to 0. The other protocols are normalized with respect to the baseline. For example, for dr in the static case, CTP would have a value of

	Static	Mobility	Network Size	Offered Load	Temp	Interference
CTP	96.1	16.1	90.1	66.0	88.8	58.0
Evergreen	97.3	68.5	94.9	82.0	91.0	70.0
ORW	93.3	50.7	93.0	9.0	85.5	64.0
BCP	94.6	39.0	58.8	38.0	89.8	47.0

Table 2.2: Delivery ratio: The percentage of packets delivered successfully at the root node.

	Static	Mobility	Network Size	Offered Load	Temp	Interference
CTP	7.77	0.49	16.36	0.99	4.39	2.74
EG	17.17	3.08	20.10	1.37	8.78	2.95
ORW	19.53	2.44	38.29	0.11	14.23	2.38
BCP	8.33	1.61	10.63	0.92	9.80	3.28

Table 2.3: Ratio of duty cycle and the packet delivered, i.e. $\frac{dr}{dc}$

$\frac{|97.3-96.1|}{97.3} \times 100 = 1.2$. Let us denote the normalized versions of dr, $\frac{dr}{dc}$, and tp with $\hat{dr}$, $\hat{\frac{dr}{dc}}$ and $\hat{tp}$ symbols respectively. Then, the relative 'rank' of each protocol p for the network settings $x_1, x_2, ..., x_n$ is calculated as follows:

$$R_p = W_1 \left\{ \sum_{y=x_i}^{x_n} \hat{dr}_y^p \right\} + W_2 \left\{ \sum_{y=x_i}^{x_n} \frac{\hat{dr}_y^p}{\hat{dc}_y^p} \right\} + W_3 \left\{ \sum_{y=x_i}^{x_n} \hat{tp}_y^p \right\} \tag{2.1}$$

where W_1, W_2 and W_3 are positive constants with a constraint that their sum should always equal to 1 [2]. The value of these constants indicates the preference between duty cycle, packet delivery ratio and throughput. Based on the above equation three tables are generated. Table 2.5, Table 2.6, and Table 2.7 such that each table gives preference to a different metric with the selection of the constants. We use abbreviations to represent combinations of network settings. S is used for Static (x_1), M for Mobility (x_2), N for Network Size (x_3), O for Offered Load (x_4), T for Temperature (x_5) and I for Interference (x_6). Table 2.5 shows the results when the delivery ratio is given higher priority over the duty cycle and throughput. In this case, Evergreen is the better choice among almost all the cases. In contrast, when duty cycling (energy) is given more preference (Table 2.6), ORW outperforms in the majority of cases as ORW consumes significantly less energy as compared to all other protocols. Finally, Table 2.7 gives results when throughput is given a higher preference than duty cycling and delivery ratio. Note that there could be scenarios which are best represented by different values of W_1, W_2 and W_3. However covering all these scenarios in this report is not possible, hence only three extreme cases are presented here to explain our ranking approach.

[2] Note that further metrics such as latency could be added in a similar way

	Static	Network Size	Offered Load	Temp	Interference
CTP	1.33	2.30	81.46	4.16	1.26
EG	1.56	2.42	101.79	6.58	1.51
ORW	1.55	2.34	11.11	7.06	1.40
BCP	1.26	1.44	57.12	6.36	1.03

Table 2.4: Throughput: Average number of data packets received at the root node in a second.

Network Settings	CTP Rank	Evergreen Rank	ORW Rank	BCP Rank	Network Settings	CTP Rank	Evergreen Rank	ORW Rank	BCP Rank
S	8.47	**1.21**	3.32	9.88	SM	78.04	**1.21**	26.14	49.08
SMN	88.24	**5.96**	27.97	90.75	SMNO	108.64	**5.96**	117.30	141.35
SMNOT	121.62	**10.46**	122.12	146.52	SMNOTI	138.65	**11.47**	132.46	175.98
SMNOI	125.68	**6.97**	127.64	170.81	SMNT	101.22	**10.46**	32.79	95.92
SMNTI	118.26	**11.47**	43.13	125.38	SMNI	105.28	**6.97**	38.31	120.20
SMO	98.44	**1.21**	115.46	99.68	SMOT	111.42	**5.71**	120.28	104.86
SMOTI	128.45	**6.72**	130.62	134.32	SMOI	115.47	**2.22**	125.80	129.14
SMT	91.02	**5.71**	30.96	54.25	SMTI	108.05	**6.72**	41.30	83.71
SMI	95.07	**2.22**	36.48	78.54	SN	18.67	5.96	**5.16**	51.54
SNO	39.07	**5.96**	94.48	102.15	SNOT	52.05	**10.46**	99.30	107.32
SNOTI	69.09	**11.47**	109.64	136.78	SNOI	56.11	**6.97**	104.82	131.61
SNT	31.65	10.46	**9.97**	56.72	SNTI	48.69	**11.47**	20.31	86.18
SNI	35.71	**6.97**	15.49	81.00	SO	28.87	**1.21**	92.65	60.48
SOT	41.85	**5.71**	97.46	65.66	SOTI	58.88	**6.72**	107.80	95.11
SOI	45.90	**2.22**	102.99	89.94	ST	21.45	**5.71**	8.14	15.05
STI	38.48	**6.72**	18.48	44.51	SI	25.50	**2.22**	13.66	39.33
M	69.57	**0**	22.82	39.20	MN	79.77	**4.75**	24.65	80.87
MNO	100.17	**4.75**	113.98	131.48	MNOT	113.15	**9.25**	118.80	136.65
MNOTI	130.19	**10.26**	129.13	166.11	MNOI	117.21	**5.76**	124.32	160.94
MNT	92.75	**9.25**	29.47	86.04	MNTI	109.79	**10.26**	39.81	115.50
MNI	96.81	**5.76**	34.99	110.33	MO	89.97	**0**	112.14	89.81
MOT	102.95	**4.50**	116.96	94.98	MOTI	119.98	**5.51**	127.30	124.44
MOI	107.00	**1.01**	122.48	119.27	MT	82.55	**4.50**	27.64	44.37
MTI	99.58	**5.51**	37.97	73.83	MI	86.60	**1.01**	33.16	68.66
N	10.20	4.75	**1.83**	41.67	NO	30.60	**4.75**	91.16	92.28
NOT	43.58	**9.25**	95.98	97.45	NOTI	60.62	**10.26**	106.32	126.91
NOI	47.64	**5.76**	101.50	121.73	NT	23.18	9.25	**6.65**	46.84
NTI	40.22	**10.26**	16.99	76.30	NI	27.24	**5.76**	12.17	71.13
O	20.40	**0**	89.33	50.61	OT	33.38	**4.50**	94.14	55.78
OTI	50.41	**5.51**	104.48	85.24	OI	37.43	**1.01**	99.66	80.07
T	12.98	**4.50**	4.82	5.17	TI	30.01	**5.51**	15.16	34.63
I	17.03	**1.01**	10.34	29.46					

Table 2.5: Delivery ratio preferred: Protocols ranking with $W_1 = 0.8$, $W_2 = 0.1$, and $W_3 = 0.1$

Network Settings	CTP Rank	Evergreen Rank	ORW Rank	BCP Rank	Network Settings	CTP Rank	Evergreen Rank	ORW Rank	BCP Rank
S	49.77	9.66	**0.44**	48.05	SM	124.61	**9.66**	19.57	90.58
SMN	171.42	47.67	**20.07**	156.20	SMNO	197.70	**47.67**	111.46	192.28
SMNOT	257.38	**78.96**	112.06	218.30	SMNOTI	273.99	**87.05**	135.63	224.76
SMNOI	214.31	**55.76**	135.03	198.74	SMNT	231.10	78.96	**20.67**	182.22
SMNTI	247.71	87.05	**44.25**	188.68	SMNI	188.04	55.76	**43.64**	162.66
SMO	150.89	**9.66**	110.96	126.66	SMOT	210.57	**40.95**	111.56	152.68
SMOTI	227.18	**49.04**	135.14	159.14	SMOI	167.50	**17.75**	134.54	133.12
SMT	184.29	40.95	**20.18**	116.60	SMTI	200.90	49.04	**43.75**	123.06
SMI	141.22	**17.75**	43.15	97.04	SN	96.58	47.67	**0.94**	113.67
SNO	122.85	**47.67**	92.33	149.75	SNOT	182.53	**78.96**	92.93	175.77
SNOTI	199.15	**87.05**	116.50	182.23	SNOI	139.47	**55.76**	115.90	156.21
SNT	156.26	78.96	**1.54**	139.69	SNTI	172.87	87.05	**25.12**	146.15
SNI	113.19	55.76	**24.51**	120.13	SO	76.04	**9.66**	91.83	84.14
SOT	135.72	**40.95**	92.43	110.16	SOTI	152.33	**49.04**	116.01	116.62
SOI	92.66	**17.75**	115.41	90.60	ST	109.45	40.95	**1.05**	74.07
STI	126.06	49.04	**24.62**	80.53	SI	66.38	**17.75**	24.02	54.51
M	74.85	**0**	19.13	42.53	MN	121.66	38.01	**19.63**	108.14
MNO	147.93	**38.01**	111.01	144.22	MNOT	207.61	**69.30**	111.61	170.24
MNOTI	224.23	**77.38**	135.19	176.70	MNOI	164.55	**46.10**	134.59	150.68
MNT	181.34	69.30	**20.23**	134.16	MNTI	197.95	77.38	**43.80**	140.62
MNI	138.27	46.10	**43.20**	114.60	MO	101.12	**0**	110.52	78.61
MOT	160.80	**31.29**	111.12	104.63	MOTI	177.41	**39.38**	134.69	111.09
MOI	117.74	**8.09**	134.09	85.07	MT	134.52	31.29	**19.73**	68.55
MTI	151.14	**39.38**	43.31	75.01	MI	91.46	**8.09**	42.70	48.99
N	46.81	38.01	**0.50**	65.62	NO	73.09	**38.01**	91.88	101.70
NOT	132.77	**69.30**	92.48	127.72	NOTI	149.38	**77.38**	116.06	134.18
NOI	89.70	**46.10**	115.46	108.16	NT	106.49	69.30	**1.10**	91.63
NTI	123.10	77.38	**24.67**	98.09	NI	63.42	46.10	**24.07**	72.07
O	26.28	**0**	91.39	36.08	OT	85.96	**31.29**	91.99	62.10
OTI	102.57	**39.38**	115.56	68.56	OI	42.89	**8.09**	114.96	42.54
T	59.68	31.29	**0.60**	26.02	TI	76.29	39.38	**24.18**	32.48
I	16.61	8.09	23.58	**6.46**					

Table 2.6: Energy consumption preferred: Protocols ranking with $W_1 = 0.1$, $W_2 = 0.8$, and $W_3 = 0.1$

Network Settings	CTP Rank	Evergreen Rank	ORW Rank	BCP Rank	Network Settings	CTP Rank	Evergreen Rank	ORW Rank	BCP Rank
S	17.82	1.21	**0.68**	21.17	SM	33.86	**1.21**	5.34	30.25
SMN	44.07	**5.96**	7.96	73.51	SMNO	64.80	**5.96**	97.33	117.27
SMNOT	104.81	**15.17**	97.93	128.37	SMNOTI	121.50	**16.18**	107.41	157.05
SMNOI	81.50	**6.97**	106.81	145.95	SMNT	84.08	15.17	**8.56**	84.60
SMNTI	100.78	**16.18**	18.04	113.28	SMNI	60.77	**6.97**	17.44	102.18
SMO	54.59	**1.21**	94.71	74.02	SMOT	94.60	**10.42**	95.31	85.11
SMOTI	111.30	**11.43**	104.79	113.79	SMOI	71.29	**2.22**	104.18	102.70
SMT	73.87	10.42	**5.94**	41.34	SMTI	90.57	**11.43**	15.42	70.02
SMI	50.56	**2.22**	14.81	58.93	SN	28.03	5.96	**3.30**	64.42
SNO	48.75	**5.96**	92.67	108.19	SNOT	88.76	**15.17**	93.27	119.29
SNOTI	105.46	**16.18**	102.75	147.97	SNOI	65.45	**6.97**	102.15	136.87
SNT	68.04	15.17	**3.90**	75.52	SNTI	84.73	16.18	**13.38**	104.20
SNI	44.72	**6.97**	12.78	93.10	SO	38.54	**1.21**	90.05	64.94
SOT	78.55	**10.42**	90.65	76.03	SOTI	95.25	**11.43**	100.13	104.71
SOI	55.24	**2.22**	99.52	93.61	ST	57.83	10.42	**1.28**	32.26
STI	74.52	11.43	**10.75**	60.94	SI	34.51	**2.22**	10.15	49.85
M	16.05	**0**	4.66	9.08	MN	26.25	**4.75**	7.28	52.34
MNO	46.98	**4.75**	96.66	96.11	MNOT	86.99	**13.96**	97.26	107.20
MNOTI	103.69	**14.97**	106.73	135.88	MNOI	63.68	**5.76**	106.13	124.78
MNT	66.26	13.96	**7.89**	63.43	MNTI	82.96	**14.97**	17.36	92.11
MNI	42.95	**5.76**	16.76	81.02	MO	36.77	**0**	94.03	52.85
MOT	76.78	**9.21**	94.63	63.95	MOTI	93.48	**10.22**	104.11	92.62
MOI	53.47	**1.01**	103.51	81.53	MT	56.05	9.21	**5.26**	20.18
MTI	72.75	**10.22**	14.74	48.85	MI	32.74	**1.01**	14.14	37.76
N	10.21	4.75	**2.62**	43.26	NO	30.93	**4.75**	91.99	87.03
NOT	70.94	**13.96**	92.60	98.12	NOTI	87.64	**14.97**	102.07	126.80
NOI	47.63	**5.76**	101.47	115.70	NT	50.22	13.96	**3.23**	54.35
NTI	66.91	14.97	**12.70**	83.03	NI	26.90	**5.76**	12.10	71.93
O	20.73	**0**	89.37	43.77	OT	60.73	**9.21**	89.97	54.86
OTI	77.43	**10.22**	99.45	83.54	OI	37.42	**1.01**	98.85	72.45
T	40.01	9.21	**0.60**	11.10	TI	56.71	10.22	**10.08**	39.77
I	16.70	**1.01**	9.48	28.68					

Table 2.7: Throughput preferred: Protocols ranking with $W_1 = 0.1$, $W_2 = 0.1$, and $W_3 = 0.8$

2.4.1 Protocol Selection for Use Cases

In this subsection we present how our approach to the protocol selection problem applies to two use cases studied by the RELYonIT consortium.

Smart Parking System As described in section 2.1, we follow three steps to select the best algorithm for a particular scenario.

- *Step 1: Identify the properties of the environment.* Based on deliverable D-4.1, we choose temperature (T) and interference (I) as the environmental properties to take under consideration.
- *Step 2: Identify the metrics of interest.* Delivery ratio is identified as the most critical metric for this use case. Imagine a driver reaching a parking spot designated by the system, only to find it has been already occupied and the user was not informed since data packets were not delivered. A scenario such as this would cause the driver to lose confidence in the system.
- *Step 3: Select protocol.* Based on Table 2.5, that favors throughput over delivery ratio and radio duty cycle, we can see that the most suitable protocol for our use case is Evergreen.

Infrastructure Monitoring As above, we follow the same three steps that will lead us to selecting the most appropriate protocol.

- *Step 1: Identify the properties of the environment.* The most important environmental property for this use case is temperature (T).
- *Step 2: Identify the metrics of interest.* In this use case, sensor nodes typically report deltas of values or their gradient, thus not losing measurements is of critical importance. We therefore select delivery ratio as the metric of interest.
- *Step 3: Select protocol.* Based on Table 2.5, that favors delivery ratio over throughput and radio duty cycle, we choose Evergreen as the most suitable for this use case.

3 Protocol Parameterization

The protocol parameterization framework is a central component of the RELYonIT software architecture, it generates protocol configurations that meet previously specified requirements. The actual parametrization component relies on mathematical optimization to determine a near optimal protocol configuration for a specific application based on a user-generated requirement specification and instances of environment, platform, and protocols models. Especially the environment model instances are adapted to the application by collecting relevant environment data before the actual deployment. The data collection employs the tools described in deliverable D-1.3 [6].

The protocol configuration generated by the parametrization component is deployed along with the application code and the implementation of the employed Internet of Things protocols. A runtime support component enables the use of different configurations based on the current situation the system is in. Additional runtime components within the RELYonIT framework monitor the model assumptions that were used while generating the configurations and alert the user in case of problems and ensure best effort operation in unanticipated environments. These additional components are described in more detail in Chapter 4.

In the following, we first give a coarse grained description of the software architecture of the parametrization component. In Section 3.2 we take a closer look at the actual parameterization component. This description includes brief coverage of the optimization techniques employed by the component. Section 3.2.3 highlights important aspects of its software implementation. In Section 3.3 the supplementary rum-time component and its interface are described. Section 3.4 provides an example of how new protocol models can be added to the framework. Finally, Section 3.5 evaluates the performance and usability of the tools.

3.1 Architecture

Figure 3.1 presents a bird's eye view of the architecture of the RELYonIT parameterization architecture. The central component of the architecture is the actual static configuration framework for protocol parameterization. This component selects a suitable parameter set to ensure the dependable performance of Internet of Things protocols. The tool receives an XML-encoded specification of the dependability requirements as input. This file employs the specification language developed in Task 3.1. A detailed description of the syntax and semantics can be found in deliverable D-3.1 [18]. In addition to a specification of the application requirements, the file also contains the selection of protocols as determined by the algorithm described in Chapter 2. A single requirement specification may contain several requirement sets for different modes of operation. Some applications have diverse modes of operation that get activated based on system usage and environmental conditions. These operation modes often have very different requirements in terms of network performance and reliability. For example, a system to detect forest fires would be typically optimized for a long system lifetime during

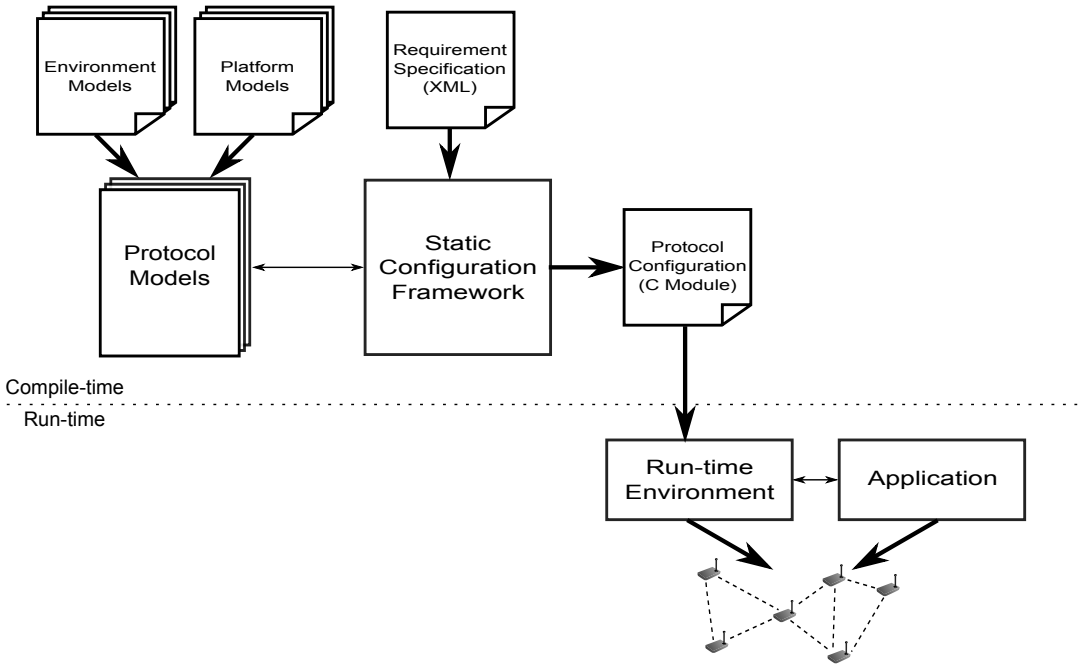


Figure 3.1: Architecture of the parameterization framework.

normal operation. As soon as a forest fire is detected, lifetime is not a primary concern anymore, but we are interested in a very fast dissemination of detailed data about the current state of the fire. To optimally support applications like this, it is necessary to allow the developer to define a set of performance states with different requirements. At run-time, the application can switch between these performance states and such select different sets of active requirements to adapt to the current situation. The selection of a performance state determines the active configuration for the employed protocols.

In addition to the requirements specification, the configuration tool has access to a repository of protocol models. Based on the protocols defined in the requirement specification, a suitable protocol model is chosen. The individual protocol models may in turn rely on application-specific instances of environment and platform models. A description of these models can be found in the deliverable D-2.2 [28] and D-1.1 [26].

Based on these inputs a near-optimal parameter configuration for the respective protocol is generated by the static configuration tool, employing mathematically optimization techniques. If the requirement specification defines different modes of operations, the process is executed individually for the requirements of each mode.

The final output of the protocol parameterization tool is a protocol configuration for each employed protocol. These configurations are static and do not change at run-time. Nevertheless, it is possible to switch between configurations associated to different performance modes. The configurations are emitted in the form of a C source file that can be compiled as an individual module and can be linked with the run-time component and the actual application program. At run-time, the information from the configuration module is passed to the respective protocols by the run-time environment. The run-time environment is also in charge of keeping track of

the different application performance states. When the application changes the performance state each protocol is informed about the update and a new set of parameters is passed.

3.2 Protocol Parameterization

The static protocol parameterization component employs mathematical optimization to generate an optimal parameter configuration for one or more Internet of Things protocols based on a user-defined dependability specification.

This dependability specification employs the XML-based language described in deliverable D-3.1 [18]. Each dependability specification essentially defines a constrained optimization problem. Within the context of RELYonIT we only consider the single objective case where a single metric is optimized, while the user may specify a number of additional constraints on the same or other metrics. Currently, the system supports three metrics, that are derived from the use cases studied in deliverable D-4.1 [5]: (1) system lifetime, (2) data yield, (3) and latency. This results in optimization problems of the following form:

$$\begin{array}{ll} \text{Maximize/Minimize} & m_0(c) \\ \text{Subject to} & m_1(c) \geq, \leq t_1 \text{with probability } p_1 \\ & m_2(c) \geq, \leq t_2 \text{with probability } p_2 \end{array} \tag{3.1}$$

The goal is to find a set of protocol configuration parameters c that optimizes a single metric m_0. In addition, a variable number of constraints need to be fulfilled by ensuring that $m_1(c)$ and $m_2(c)$ are either larger or smaller than the respective thresholds t_1 and t_2 with a probability of at least p_1 or p_2. Note that m_1 and m_2 may be the same as m_0.

3.2.1 Optimization Problem

To be suitable for automatic optimization, the optimization problem introduced in the previous section needs to be transformed into suitable input for the employed optimization strategies.

Most optimization techniques cannot directly handle constraints. Instead constraints need to be integrated in the objective function. We achieve this by normalizing the optimization goal and all constraints to the $[0, 1]$ range and by computing the weighted sum where constraints are given a higher weight than the original objective. In our case, each constraint has twice the weight as the original optimization goal. This approach corresponds to penalty functions often used in stochastic optimization [21]. It ensures that invalid solutions are unlikely to be selected, but still allows the optimization algorithm to traverse infeasible regions of the search space.

In addition, the requirement specification allows to attach probabilities to individual constraints. These constraints only need to be fulfilled with the specified probability. This is not supported by standard optimization techniques. To support this feature, we exploit the fact that most environment parameters exhibit a periodic behavior as shown in deliverable D-1.1 [26]. This enables us to sample the environment at different points of time and identify time frames with distinct environment conditions. Each time frame leads to a distinct environmental model instance. If we assume communication events will be more or less evenly spread over the different environmental settings, we can assign a probability to each time frame based on its relative length.

Without support for different probabilities, each model would provide a single cost function $f_m(c)$ for each supported metric m. The function $f_m(c)$ returns the normalized cost of configuration c. To enable the use of different probabilities, each model needs to provide a set of functions, where each of these functions $f_{m,i}(c)$ has an associated probability p_i. Each of these functions uses a different instance of the environment model that represents a distinct time frame. During the optimization instead of a single function $f_m(c)$, each function $f_{m,i}(c)$ is evaluated for the current configuration. In a second step, the associated probabilities p_i of all function that have as cost value that is above the threshold given in the requirement specification are summed up. For each constraint j, these and this probability is used instead of the original function value in the optimization process.

If we assume a weight of 2 for the constraints and a problem of the form of Equation 3.1, we end up with the following definition of the total fitness e for a given configuration c:

$$e(c) = \frac{\left(2 * \sum_{j=1}^{|M|-1} \rho\left(\sum_{i=1}^{|P|} \tau_{\mathrm{op}_j}(f_{m_j,i}(c), p_i, t_j), q_j\right) + f_{m_0}(c)\right)}{3} \tag{3.2}$$

The set M contains the metrics employed by the current specification. As a convention, we assume that the metric of the objective function is named m_0 while the constraint metrics use the indices 1 to $|M|-1$. The set P contains the probability values provided by the employed model instance. The function τ realizes the above mentioned comparison with the threshold value t. For the operators $<$ and $>$ it is defined as follows:

$$\tau_{>}(v, p, t) = \begin{cases} p & v > t \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

$$\tau_{<}(v, p, t) = \begin{cases} p & v < t \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

The function ρ is used to determine the error between the desired probability of a constraint and the current probability of a constraint being fulfilled. If the difference would be negative, it is set to 0. Consequently, the function is defined as follows:

$$\rho(p, t) = \begin{cases} p - t & p < t \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

To illustrate the conversion with an example we consider the following requirement specification:

$$\begin{array}{ll} \text{Minimize} & \text{latency}(c) \\ \text{Subject to} & \text{yield}(c) \geq 0.94 \text{ with probability } 0.95 \end{array} \tag{3.6}$$

The employed protocol model supports two metrics, the latency of a packet transmission and the packet reception rate. We assume, that our model instance has data for two time intervals with a respective probability of 0.4 and 0.6. In this example, we intend to find a configuration c that gives the smallest possible jamming period length that still supports a packet reception rate of more than 0.94 with a probability of 0.95. Converting this specification according to the previously introduced rules, yields the following adapted fitness function:

$$e(c) = \frac{(2 * \rho(\tau_{>}(f_{\text{yield},1}(c), 0.4, 0.94) + \tau_{>}(f_{\text{yield},2}(c), 0.6, 0.94), 0.95) + f_{\text{latency}}(c))}{3} \tag{3.7}$$

Our goal is now to minimize the fitness function e, which leads to the following unconstrained optimization problem:

$$\text{Minimize} \quad e(c) \tag{3.8}$$

3.2.2 Optimization Strategies

The protocol parameterization component can utilize a number of different optimization strategies to solve the optimization problem, allowing the user to choose a strategy that is most appropriate for the specification and models at hand.

The current prototype implements exhaustive search and two stochastic optimization strategies. During the course of the RELYonIT project, we also explored different approaches employing deterministic optimization strategies, but these proofed unsuitable for the task, as they required a significant amount of fine tuning for each problem instance. Deterministic strategies have the advantage to repeatably generate the same results if the input is left unchanged. In addition, they are guaranteed to find an actual minimum or maximum, even though not necessary a global one.

Stochastic optimization strategies are not able to give these guarantees, but are usually more robust to noisy data and require less fine tuning for a specific problem instance. Nevertheless, suitable strategies still possess a high probability of convergence and are usually able to find near-optimal solution. Due to their inherent randomness, stochastic strategies tend to be able to escape local minima and to approach a global optimum even in a convex search space.

Exhaustive Search

Exhaustive search is actually not an optimization strategy, but instead all possible solutions are evaluated and the best one is chosen. The search is guaranteed to find the best solution according to the employed metric, but as all solutions need to be visited an exhaustive search in not very efficient in the general case. For more complex models it is usually not feasible at all. We still implemented exhaustive search as an alternative as it is useful for debugging purposes and might be more efficient for simple models with only a limited number of valid configurations.

Simulated Annealing

Simulated Annealing is a probabilistic metaheuristic that is usually more efficient than an exhaustive search. In contrast to exhaustive search and deterministic optimization strategies it does not guarantee that an optimal solution will be found. Still it has a high probability of finding a solution that is close to an optimum and the algorithm is often also able to find a global optimum in a search space with a large number of local optima.

The idea is based on annealing in metallurgy where controlled cooling is used to improve the crystal structure in metals. The strategy emulates the reduction of energy in the material during the cooling process. At the beginning of the process, while the temperature is high, large changes in the material state are still possible. Later at colder stages, only increasingly small changes are possible and the state of the material converge to a final configuration.

In the following, we describe the basic algorithm of simulated annealing that is also shown as pseudo code in Algorithm 1. The algorithm starts with a randomly initialized configuration

Algorithm 1 Simulated Annealing

```
c ← randconf()
e ← evaluate(c)
k ← 0
while k < k_max and e > e_min do
  t ← temperature(k/k_max)
  c_new ← neighbor(c)
  e_new ← evaluate(c)
  if p(e, e_new, t) > random() then
    c ← c_new
    e ← e_new
  end if
  k ← k + 1
end while
return c
```

c whose current energy e is evaluated by an application-specific evaluation function *evaluate*. The actual algorithm is repeated until the maximal number of iterations $k_{\max}$ is reached or the energy drops under a predefined acceptance threshold $e_{\min}$. In each step, the current temperature t is adjusted according to the temperature regime *temperature*. The exact regime depends on the problem class at hand and the function *temperature* only needs to ensure that the temperature is monotonically decreasing. Next, a new configuration is generated based on the current configuration. This employs an often problem-specific *neighbor* function. Usually, it should favor configurations that are close to the current one and only deviate slightly. The new configuration is then evaluated and compared with the current solution and a probabilistic decision is made to keep either the current configuration or replace it with the new one. This decision is based on the function p for which we used the following definition:

$$p(e, e_{\text{new}}, t) \mapsto \begin{cases} 1 & e_{\text{new}} < e \\ \exp(-(e_{\text{new}} - e)/t) & \text{otherwise} \end{cases} \tag{3.9}$$

In this case, new configurations that are superior to the current one are alway used as the new configuration to continue with. Inferior configurations have a slight chance to be chosen. This allows the algorithm to escape from local minima. The likelihood of choosing an inferior configuration decreases with temperature. Finally, the process is repeated until the maximal number of iterations is used up or the energy of the current solution drops below a predefined threshold. The algorithm returns the final configuration that is expected to be close to an optimum.

In our implementation, we adopted another common optimization of the algorithm. Instead of just storing the current configuration, we also save the best configuration seen so far. In the end, the best configuration seen is reported instead of the final configuration. This way, superior configurations are never lost and the likelihood of finding an optimal solution is increased.

Evolution Strategies

Like simulated annealing, evolution strategies are a probabilistic metaheuristic. Consequently, they also do not guarantee that an optimal solution is found, but will usually find a good solution close to the global optimum with high probability. Instead of just a single configuration per step, evolution strategies employ a population of configurations and thus are able to explore several areas of the search space at once. As a consequence, finding a good solution usually requires less iterations, but each iteration is computationally more demanding. The use of a population also makes an extension to multi-objective optimization comparatively straight forward, which could be exploited in future versions of the tool. A distinct feature of evolution strategies is a low number of tunable parameters. Instead evolution strategies are able to automatically adapt to the problem at hand. A major disadvantage of evolution strategies is the significantly higher complexity that increases the implementation effort and the risk for errors.

Evolution strategies are loosely based on concepts imitating evolution in living species. Unlike genetic algorithms, they do not model genes but instead employ a more abstract view on evolution.

The algorithm starts out with a randomized population of configurations. At each step, a predefined number of new configurations is generated by a cross-over between two existing solutions. Next, a mutation operator is applied to each of the new configurations. The mutation is typically performed by adding a normally distributed random value to each individual value in the configurations. The properties of the distribution are stored alongside the configuration. These additional parameters are implicitly also subject to the optimization and are consequently automatically adapted to the problem at hand. Finally, a subset of the new configurations is semi-randomly selected. The selection is biased by the fitness of the configuration under the problem-specific metrics. Better solutions are more likely selected to be part of the next generation. The process is repeated until a predefined number of iterations is reached or the fitness of the best solution reaches a predefined threshold.

For our implementation we employed a slightly optimized variant of the basic strategy developed by Reehuis and Bäck [20].

3.2.3 Implementation

The actual protocol parameterization tool is implemented as a standalone Python application. It was developed with extensibility at mind. Protocol models are integrated via a well-defined plug-in interface and additional models can be easily added without a need to modify the framework. In contrast, the integration of different optimization strategies does not currently use a plug-in interface, but the integration of newly implemented strategies is still rather straight forward and requires only a minimal amount of local code modifications. In addition, the implemented optimization strategies expose a number of configuration parameters that allow to adapt the strategies to different needs without any change to the code.

We will now first give an overview of the architecture of the system and will later take a closer look at some subsystems and interesting design decisions. Figure 3.2 illustrates the software architecture of the parameterization tool employing UML notation. The application is build around the central `Main` class which acts as controller for the parameterization process. It also implements the user interface and takes care of a proper initialization of the other components.

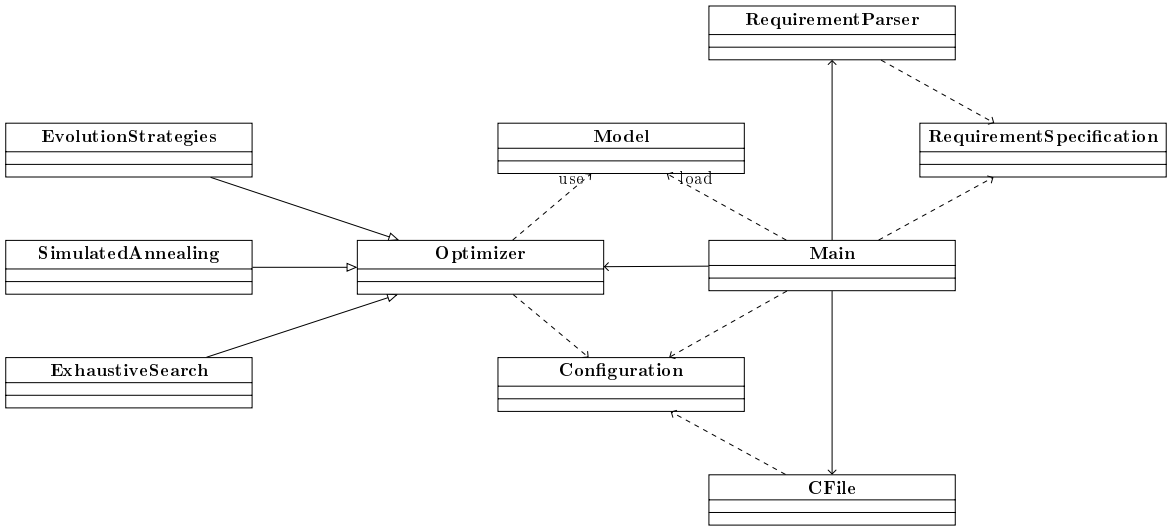


Figure 3.2: Overview of the system architecture of the protocol parameterization tool

The `Main` class is in charge of parsing user supplied command line parameters, reading the requirement specification and a supplemental configuration file and loading the model plug-ins that are required by the requirement specification at hand. After the initialization, the parsed `RequirementSpecification` and the initialized `Models` are passed to the configured optimizer. Control is passed to the `Optimizer` implementing one of the available optimization strategies as introduced in Section 3.2.2. The optimization strategies employ the different `Models` in order to evaluate specific `Configurations`. After reaching a strategy-specific termination condition, the best `Configuration` found is returned to the `Main` program. The main program passes this `Configuration` to `CFile` which generates a C representation of the configuration as output. This C representation is later used by the run-time environment to actually configure the communication protocols. The exact format of this representation and the design of the run-time environment are described in more detail in Section 3.3.

Three abstractions play a central role in the implementation of parameterization framework: the internal representation of the requirement specification, the representation of protocol configurations, and the interface to protocol model plug-ins.

A requirement specification detailing the expectations of the user for the performance of the final application is the most important input to the parameterization process. The requirement specification document employs the format introduced in deliverable D-3.1 [18]. The requirement specification is parsed at the start of execution and represented within the program with a set of objects as shown in Figure 3.3. A requirement specification can at most contain one objective, which is internally represented by an `Objective` object. Objectives can be either maximization or minimization goals. In the former case, the `criterion` attribute has the value `MAX`, in the later case, it has the value `MIN`. The objective is associated to a specific `Metric`. In addition to the objective, each specification contains one or more constraints, each also associated to a `Metric`. In addition, a constraint is defined by a threshold value. Besides, each constraint has a `probability` attribute, defining the probability with which the constraint needs to be met.

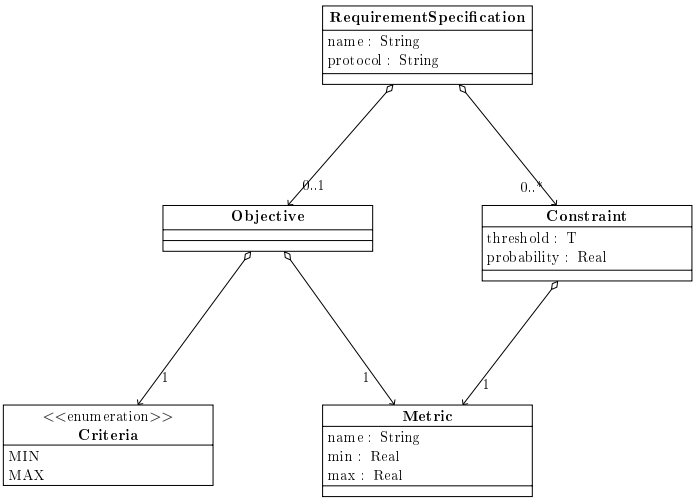


Figure 3.3: Internal representation of requirement specifications

`Configuration` objects represent possible parameter configurations during the optimization process. Its UML representation can be seen on the lower part of Figure 3.4. The basic `Configuration` interface is generic and the same for all strategies. If specific strategies require further attributes or methods they may subclass `Configuration` as needed. In the default form, each `Configuration` contains a number of values each represented by a `Value` object. Different subclasses of `Value` exist to handle different value types. Each `Value` object is associated to a specific `Parameter` which also defines the possible value range. This additional information is also the main reason to use a custom class to represent configuration values. The different value types enable the optimization strategies to easily handle the different types individually, if needed.

The most important abstraction to ensure the intended extensibility of the framework is the protocol model abstraction. While the former classes are primarily used internally, the model interface is exposed to any user implementing his own protocol models. A plug-in interface makes it comparatively easy to add new models for different protocols. A plug-in needs to provide a class that inherits from the provided `Model` class and implements its interface. Most importantly it needs to provide an implementation of the `evaluate` method which evaluates the model under a set of different metrics with a set of passed parameters. Before the model can be used, its `init` method is called to pass model-specific parameters and to initialize the model. Calls to other methods or access to properties is only valid after `init` has been called. During the optimization process, the model instance is used to evaluate different configurations. To trigger a model evaluation, its `evaluate` method is called and the relevant parameter values of the current configuration are passed as arguments. Each model specifies explicitly which parameters it supports as property `parameters`. Four basic parameter types are supported by the framework as can also be seen in Figure 3.4: (1) integers, (2) floating point numbers, (3) nominal values, and (4) Boolean values. For each type it is also possible to specify a range of possible values. Instances of parameters have unique names that identify them within the framework. The parameter values that are passed as arguments also employ a set of custom value types to be able to carry additional information. Each of these values is associated with

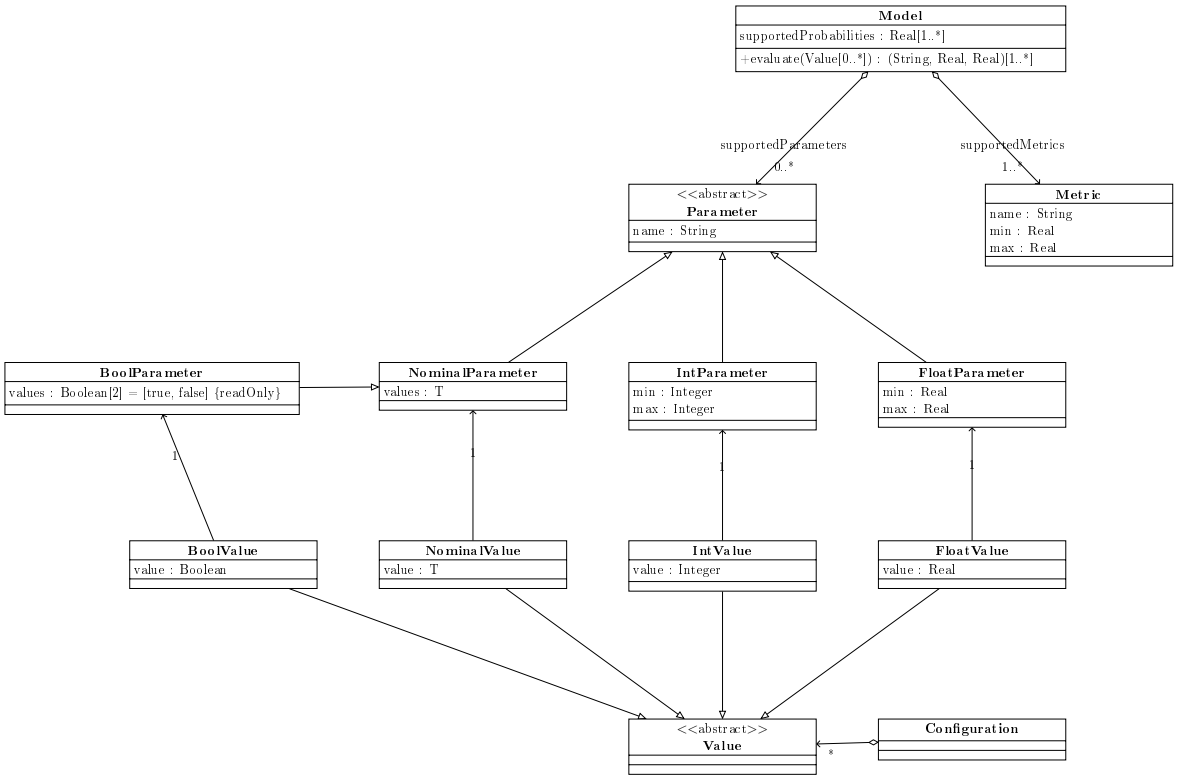


Figure 3.4: Model interface and parameter types

a parameter instance. The `Parameter` and `Value` classes are identically to the ones employed by the `Configuration` class. The return value of the `evaluate` method is a list with a fitness value for each supported metric. Like the supported parameters, all supported metrics are specified by the property `metrics`. If the model supports different time frames with associated probabilities, as described in Section 3.2.1, one fitness value per metric and time frame is returned. In this case, the plugin needs to also specify a list of supported probabilities as property `probabilities`.

In the current implementation, six protocol models are available. Additional protocol models can be easily added by the user as described in Section 3.4. The following protocol models are available:

- *energy*, a model of the effect of duty cycle modifications on the energy spending of a MAC protocol. The model can be used to find the best duty cycle for a specific radio environment. This model relies on preprocessed data generated by a tool developed within WP2.
- *jag*, a model of jamming based agreement (JAG). This model can be used to determine the jamming period length that yields an optimal agreement probability for the JAG protocol.
- *micmac*, a model of the MicMAC protocol, a multichannel MAC protocol. This model allows to find parameter settings that yield a good packet reception rate and latency.
- *packet_length*, a generic model to find a packet size that results in an optimal yield in the presence of significant radio interference. This model employs preprocessed data generated by a tool developed within WP2.
- *packet_length_closed_form*, a variant of the previous model, that does not rely on preprocessed data, but uses a closed-form mathematical model.
- *tempmac*, a model of a MAC protocol extension that adapts the CCA threshold to mitigate the impact of temperature. The model allows to determine an optimal initial CCA threshold value for a specific temperature model.

3.3 Run-time Interface of Protocol Parameterization

To be actually useful, the determined configuration parameters need to be made available to the protocols running in an actual Internet of Things application. To simplify this task, a unified configuration interface is provided by the RELYonIT run-time environment, which enables the individual protocol implementations to receive their configuration parameters at initialization.

To support different performance states for the application, the static optimization tool is feed with multiple requirements specifications. Each specification defines the requirements for a specific performance state. The static optimization tool generates an individual parameter configuration for each performance state.

The output of the static optimization tool consist of a source file containing a C representation of the different configurations. This file needs to be compiled and linked with the application. A

run-time environment provides a procedure to switch the active performance state and provides functions to enable the protocol implementations to retrieve the current parameter settings.

3.3.1 Run-time Interface

To support this functionality, the run-time interface provides a number of functions. All of these function use the `relyonit_` prefix. The current interface slightly differs from the one originally introduced in D-3.1 [18] to improve its usability.

At run-time, the active performance state can be changed by the application through the function

```
int relyonit_set_performance_state(char* name);
```

which triggers a performance state change and an update of the active protocol configurations.

To enable the run-time environment to notify the protocols about state changes, each protocol needs to register a call-back function at initialization. To register the call-back, the protocol calls the function

```
int relyonit_register_protocol(
        char* name,
        void (*callback)(struct relyonit_protocol *))
```

during initialization to pass a function pointer to its callback function. In addition to the function pointer, the protocol needs to pass its protocol name as first parameter. This name needs to be identical to the one employed in the requirement description.

The callback function is later called by the run-time environment upon a state change and the new configuration is passed as an opaque data structure. The structure contains the active parameter values for the new performance state. Three access function allow the protocol implementation to retrieve the configuration values form this opaque structure:

```
int relyonit_get_bool_parameter(
        const struct relyonit_protocol *protocol,
        char* name);
int relyonit_get_int_parameter(
        const struct relyonit_protocol *protocol,
        char* name);
float relyonit_get_float_parameter(
        const struct relyonit_protocol *protocol,
        char* name);
```

The parameters are selected by name. Different variants enable easy handling of different data types. The data types of the C representation of the parameters correspond to the respective data types specified in the protocol model.

3.3.2 Representation of Configurations

The static optimization tool generates a C representation of the parameters of the different configurations. This representation employs the following data structures to encode the information:

```
union relyonit_parameter_value {
  int bool_value;
  int int_value;
  float float_value;
};

enum relyonit_prameter_type {
  BOOL,
  INT,
  FLOAT,
};

struct relyonit_parameter {
  char *name;
  enum relyonit_parameter_type type;
  union relyonit_parameter_value value;
};

struct relyonit_protocol {
  char *name;
  unsigned parameter_count;
  struct relyonit_parameter *parameters;
};

struct relyonit_performance_state {
  char *name;
  unsigned protocol_count;
  struct relyonit_protocol *protocols;
};
```

These structures are opaque to the application and are supposed to be accessed only via the earlier introduced access functions. Like the previously introduced functions all structures employ the `relyonit_` prefix. The prefix and the exact names might still be subject to change in future versions of the interface.

The `relyonit_parameter` structure represents an individual protocol parameter. Each parameter has a unique name, a type, and a value. The actual value of the parameter is represented by the `relyonit_parameter_value` union. Currently supported are Boolean, integer, and float values. Booleans are represented as integers in C, but they are handled differently by the optimization tool. The configuration parameters are grouped by protocol, and one instance of the `relyonit_protocol` structure exists for each protocol of the specification. The `relyonit_protocol` structure contains the name of the protocol, and an array with all parameters. In addition, it contains the size of the parameters array as attribute `parameter_count`. The data structure `performance_state` groups all protocol configurations that are active in a specific performance state. Each performance state has a unique name. In addition, this structure contains the `protocols` array with a protocol configuration and the size of this array as attribute `protocoal_count`.

An instance of a complete configuration, as generated by the static configuration tool, could look like the following example:

```
const struct relyonit_performance_state
        relyonit_performance_states[] = {
  {
    "example_state_1",
    2,
    (struct relyonit_protocol []) {
      {
        "example_protocol_1",
        2,
        (struct relyonit_parameter []) {
          {
            "para_1",
            BOOL,
            1,
          },
          {
            "para_2",
            INT,
            10,
          },
        },
      },
      {
        "example_protocol_2",
        1,
        (struct relyonit_parameter []) {
          {
            "para_1",
            INT,
            200,
          },
        },
      },
    },
  },
  {
    "example_state_2",
    2,
    (struct relyonit_protocol []) {
      {
        "example_protocol_1",
        2,
        (struct relyonit_parameter []) {
          {
            "para_1",
            BOOL,
            3,
          },
          {
            "para_2",
```

```
            INT,
            12,
          },
        },
      },
      {
        "example_protocol_2",
        1,
        (struct relyonit_parameter []) {
          {
            "para_1",
            INT,
            222,
          },
        },
      },
    },
  },
};
```

3.4 Integration of New Protocol Models

In the following, we provide a brief tutorial of the steps required to add a new protocol model to the framework. This section is intended to demonstrate the extensibility of the framework.

In the following we use a model of the jamming based agreement (JAG) protocol as example. The JAG protocol is a simple yet efficient agreement protocol for wireless sensor networks and the Internet of Things. It introduces jamming as the last step of a packet handshake between two nodes to make the handshake more robust against radio interference. The detection of a jamming signal is in general more reliable than using only regular acknowledgement (ACK) packets. This way, the protocol provides a reliable way to verify whether some information was successfully shared and it is more likely for a pair of nodes to reach an agreement. In the model we will treat the agreement rate as package yield as both are strongly related. The JAG protocol has been described in detail in deliverable D-2.1 [27].

The JAG protocol model employs an existing external implementation of the model and is consequently rather simple. In addition, the model integrates the environment model and a platform model within the protocol model. Nevertheless, all aspects of framework integration are the same for more sophisticated models.

The following listing shows the full code of the model. We will frequently reference to the listing within the following text.

```
1 from model.model import Model
2 from model.parameter import *
3 from model.metric import *
4 from . import SlotStatistics
5 from . import parser
6
7 class JagModel(Model):
8
```

```
 9      def __init__(self):
10          self._interference_data = []
11          self._pkt_tx_delay = None
12          self._ack_tx_delay = None
13          self._syn_tx_delay = None
14          self._sampling_period = None
15
16      def init(self, settings):
17          try:
18              self._pkt_tx_delay = settings['pkg_tx_delay']
19          except KeyError:
20              self._pkt_tx_delay = 1000
21          try:
22              self._ack_tx_delay = settings['ack_tx_delay']
23          except KeyError:
24              self._ack_tx_delay = 750
25          try:
26              self._syn_tx_delay = settings['syn_tx_delay']
27          except KeyError:
28              self._syn_tx_delay = 1000
29          try:
30              self._sampling_period = settings['sampling_period']
31          except KeyError:
32              self._sampling_period = 25
33          try:
34              data_file = settings['data_file']
35          except KeyError as ke:
36              raise ke
37          self._interference_data = []
38          # Placeholder until datasets with probability are in place
39          self._interference_data.append((parser.parse(data_file,  26),  0.2))
40          self._interference_data.append((parser.parse(data_file,  26),  0.3))
41          self._interference_data.append((parser.parse(data_file,  26),  0.5))
42
43      def evaluate(self, parameters):
44          jam_period = parameters['jamming_period']
45          results = []
46          for (tokens, probability) in self._interference_data:
47              slot_status = SlotStatistics()
48              for token in tokens:
49                  slot_status.updateSlotStatistics(token)
50
51              slot_status.determineProbJammingSuccessful(jam_period)
52              slot_status.determineProbSelectingIdleSlot()
53              pkt_probabilities = slot_status.pktProbabilities(
54                      self._pkt_tx_delay,
55                      self._ack_tx_delay,
56                      self._syn_tx_delay,
57                      self._sampling_period)
58              prob_disagreement = pkt_probabilities[5]
```

```
59
60              results.append(
61                      ({'yield': (1 - prob_disagreement),
62                        'jamming_period': jam_period.value
63                                          / jam_period._parameter.max},
64                      probability))
65          return results
66
67      @property
68      def metrics(self):
69          return {Metric('yield'),  Metric('jamming_period')}
70
71      @property
72      def parameters(self):
73          return [IntegerParameter('jamming_period',  0, 2000,  step_size=25)]
74
75      @property
76      def probabilities(self):
77          probabilities = []
78          for (data,  probability) in self._interference_data:
79              probabilities.append(probability)
80          return probabilities
```

In a first step we need to create a directory for the model. The directory needs to be within the configured plug-in search path and needs to have a name that corresponds to the protocol name with all spaces replaced by underscores. The JAG model of the example consequently lives in a directory called "jag". Within this directory, we need to create at least on Python package with the same name. In our case this leads to a file with the name "jag.py".

Within this file, we need to create the main class of the model implementation. A new model needs to employ the model interface to interact with the framework. This is most easily ensured by creating a main class that inherits form the `Model` class provided by the framework. To make the class available, one needs to import all classes from the `model` package within the framework implementation as done in lines 1–3. The name of the call itself needs to be based on the protocol name, too, but in this case, all space characters are removed and each individual word is capitalized ("camel case") and the string "Model" is appended. In our case this yields the class name `JagModel` as can be seen in line 7.

According to the interface, the class needs to implement two methods and three properties. The latter are used to communicate a number of properties of the plug-in to the framework. The `parameters` property returns all parameters that are required by the model. This refers only to the tunable parameters that are optimized within the parameterization process. The JAG model only uses a single tunable parameter "jamming period" that represents the selected jamming period length. The second property `metrics` provides a list of metrics for which the model generates data. The JAG model employs the standard data yield metric and a custom metric "jamming period". Finally, if the model supports the use of environment data from different time frames with associated probabilities, the `probabilities` property needs to return all available probabilities. In our case, this is based on the actual data loaded. Protocol models that do not use this feature would just return a list with the value 1.0 as only element.

In addition to the properties, we need to implement two methods. The `init` method is used

to initialize the model. While the constructor of the object is only called at plug-in loading, this method is called before each optimization run. It receives a single parameter that consists of a dictionary mapping configuration parameter names to values. Protocol models can define additional parameters that can be provided by the user as part of the system configuration files. During an optimization run, these additional parameters a treated like constants and they are not modified or interpreted by the framework. Our JAG example actually uses this to receive the location of a data file with previously recorded interference traces. In addition, JAG uses four optional parameters that allow to fine-tune the model behavior. These parameters are optional and if no value is provide, a default value is used. They are stored as attributes and are later used within the `evaluate` function.

Finally, the `evaluate` method needs to be implement. This method implements the actual behavior of the model. Based on the current configuration values, which are passed as parameter `parameters` in the form of a directory mapping parameter names to values, a value for all supported metrics is calculated. If multiple time frames are supported, this needs to be done for each time frame. In this example we just employ a single method, but typically one will divide the implementation into a number of smaller methods that are called by evaluate. Here we employ a single loop spanning from line 46 to 64 that repeats the evaluation for each available time frame. Within each iteration, we first initialize the external model implementation and then employ it to determine the protocol performance based on the loaded environment data, the static model parameters, and the jamming period length value taken from the current configuration. Finally we convert the result to the required format, a tuple consisting of a dictionary mapping between metric names to their calculated values and the probability of the time frame handled in the current iteration. After evaluation of the model for all time frames, the collected results are returned as a list.

In total the integration of an existing model only required less the 80 lines of code. This integration effort does not increase for more complex models that will require a significantly larger amount of code for the actual implementation. The model implementation can benefit from the full range of Python language features and libraries.

3.5 Evaluation

In the following, we demonstrate the functionality of the system and evaluate the relative performance of the different optimization algorithms on the basis of a simple requirement specification. The evaluation employs the JAG protocol model introduced in Section 3.4. The employed data traces where recorded in the TU Graz testbed. We employ three different traces with the respective probabilities $P = \{0.2, 0.3, 0.5\}$. According to the requirement specification, the employed jamming period length is minimized while keeping the expected packet reception rate above 0.94 with a probability of one. The maximal jamming period length is set to $20\,000\,\mu$s. The step size of the jamming period length is set to $1\,\mu$s. With the given settings, a globally optimal solution can be found with a jamming period length of $701\,\mu$s. The JAG model yields a fitness value of 0.002336 for this solution. Please note that superior solutions have smaller fitness values. In addition, a number of additional less desirable local minima exists.

For the evaluation, we execute the optimization with all available optimization strategies. For

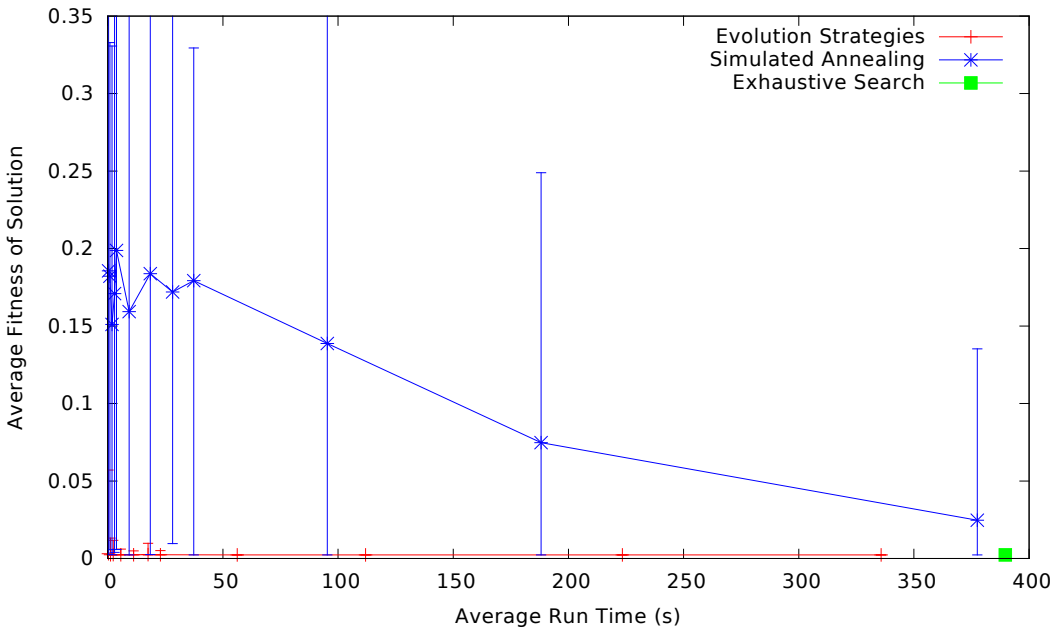


Figure 3.5: Time/quality trade off for different optimization strategies.

each strategy, we employ different numbers of iterations to evaluate the time/quality trade off. Due to obvious reasons, exhaustive search is only executed once with a single iteration. Each optimization strategy is executed one hundred times for each setting to average out random effects. This is especially important, as the employed stochastic optimization strategies rely heavily on random decisions.

The results of the evaluation can be seen in Figure 3.5. The graph represents the trade off between the runtime of the optimization algorithms and the average quality of the solutions found. The graph itself is based on the average over 100 runs, the error bars indicate the best and worst value encountered during these runs. With short run times, the optimal solution is only found in a limited number of runs and the returned solutions are often far away from the optimal one. With longer run times, the probability increases to find an optimal solution as can be seen in Figure 3.6.

As can be seen from the results, an exhaustive search requires more than 6 minutes to find the optimal solution, which establishes a baseline for the other strategies. The solution returned by an exhaustive search is always the true optimum, while with a stochastic optimization technique, the optimal solution is only returned with a high probability at best. Usually, this is offset by a significantly sorter run time of the optimization algorithms. The expected run time of the different strategies also depends on the problem at hand and some algorithms are more suitable for specific problem classes than others.

In our example it can be seen that simulated annealing performs badly. The algorithm requires more than 10 000 iterations to find suitable solutions with a sufficiently high probability. At this point it has a run time that is only slightly lower than for an exhaustive search. Nevertheless, if a close to optimal solution is sufficient, it would be possible to reduce the time required by at least 50%. The quality of the results returned by simulated annealing also varies

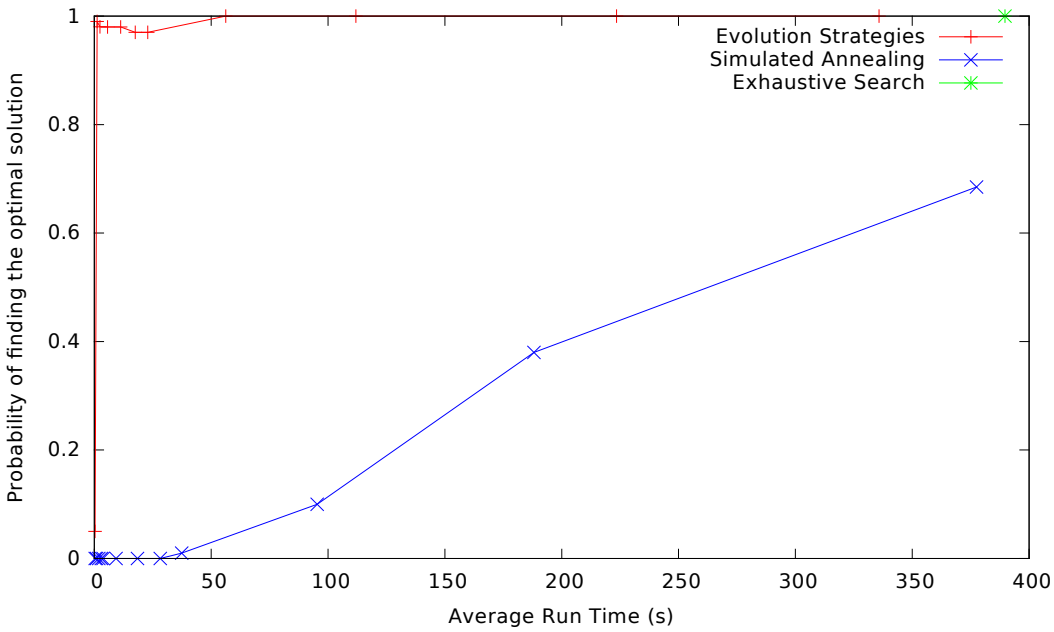


Figure 3.6: Probability of finding the optimal solution within a given time.

a lot between individual runs as indicated by very long error bars.

The evolution strategies, on the other hand, performed very good in this example. Even with only 100 iterations and the default population size of 20, this algorithm is likely able to come up with the optimal solutions. Returned inferior solutions are also of only slightly worse quality as can be seen from the error bars in Figure 3.5.

As the performance of the different algorithms strongly depends on the properties of the model, this result cannot be generalized. Nevertheless, the results demonstrate that the framework is able to generate suitable configurations within reasonable time. In addition, the evaluation underlines the benefit of providing different optimizations strategies to chose from. For simple models and small-sized data sets as employed here, exhaustive search can be a viable choice. For more complex models and larger datasets, the different optimizations strategies allow to significantly reduce the overhead.

4 Runtime Adaptation

In this chapter, we present RELYonIT's approach to runtime adaptation of protocol parameters. We begin by motivating why runtime adaptation is needed. Thereafter, we describe our simulation framework to generate suitable configurations, and lastly present the node software and communication protocol that manages the runtime adaptation.

4.1 Overview

The parameter optimization described in this deliverable is made under the condition that the network is operating within the bounds of the environmental model specified in Deliverable 1.1—*Report on Environmental and Platform Models*. Hence, the RELYonIT application's performance goals can be met with high probability in normal circumstances. If the environmental model is violated, however, the nodes of a RELYonIT system deployment must adapt their protocol configuration at runtime to attain a best-effort delivery of data packets. This configuration is run until the off-line process of parameterization (see Chapter 3 has been re-run, which typically involves the human operator and therefore typically takes a significant amount of time.

The runtime adaptation mode is activated directly after the runtime assurance module raises an alarm that the environmental model is violated, as described in Deliverable 1.3—*Report on Runtime Assurance*. Once runtime adaptation begins, it operates according to a specific configuration policy for the network deployment. While it is active, the runtime adaptation module gathers data of how the network performance is at the moment, and accordingly adjusts protocol parameters dynamically as guided by the configuration policy.

A configuration policy is not designed to meet user-specified performance requirements. Instead, the configuration policy, in conjunction with runtime adaptation, is meant as a fallback mechanism that provides acceptable performance under a wide variety of environmental conditions outside the range of the environmental model.

Configuration policies are generated off-line for each specific deployment. For this purpose, we implement a reinforcement learning algorithm in the Cooja simulator [19], which is able to emulate a network of nodes running exactly the same system firmware as is running on the real nodes in the deployment. The reinforcement learning algorithm explores a set of protocol parameter settings, and learns which settings provide acceptable performance under various environmental conditions.

The configuration policy generation is not limited to run before the network deployment is made, however. It is possible that certain conditions change over the lifetime of a network, and the original configuration policy may become stale as a result. Thus, we also support later iterations of the policy generation to be made during deployment, guided by new performance data that has been gathered from the RELYonIT application and the Contiki OS, which it runs

on top on. The newly generated configuration policies can then be disseminated to all network nodes during runtime, where they are applied when the runtime adaptation mode is active.

4.2 Reinforcement Learning of Runtime Adaptation Policies

A RELYonIT application may have several modes of operation and will in these modes have different performance objectives. But regardless of the performance objectives, the application typically has some basic communication functionality that should be provided even when the environment model is violated.

The approach taken for adapting configurations is to have a simulation framework that can simulate the set-up of a specific RELYonIT application. The simulation will be fed relevant information such as the application's network topology, the firmwares that are used, and any collected network statistics for network connectivity.

The simulation is then run together with learning mechanisms that tune the configuration while running the application scenario. The learning mechanism will during the simulation evaluate the performance given the application requirements in different simulated environments.

We have decided to use an approach based on reinforcement learning [22] for the learning mechanism. We are using it in a black box approach that allows the learning mechanism to learn from the simulations without having any detailed knowledge about what is simulated. The learning mechanism will simply investigate the possible options and learn what are good configuration actions to take. By subjecting the nodes in the simulations to conditions that normally do not occur, the learning mechanism will find configurations that allow the network to operate under a vast range of different environmental conditions that are outside those of the environmental model.

4.2.1 State Monitoring

State monitoring is an integral part of runtime adaptation because the state serves as the input to the configuration policies. Nodes monitor their internal state as represented by various metrics collected from the operating system or from the RELYonIT application. For example, metrics such as battery level, communication performance, power consumption, and routing topology statistics can be included in the state monitoring.

The particular subset of the internal state that we monitor as part of the runtime adaptation depends on what information is relevant to the requirements of the application. The selection of metrics should be carefully tuned to achieve the best results. Including too many parameters increases the time required to learn configuration policies, and it increases the energy consumption for parameters requiring active monitoring. Including too few parameters, on the other hand, makes it difficult to find reliable configuration policies because they might have insufficient information for successful learning.

4.2.2 Configuration Policy

In its simplest form, a policy is a mapping between a state and the actions that should be performed when the application is in this state. An action in this case can be to change a

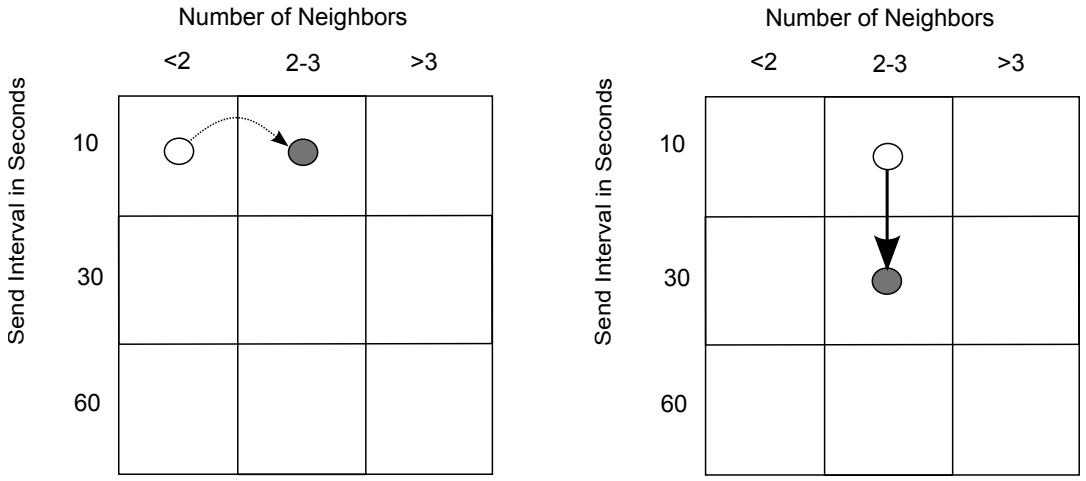


Figure 4.1: The illustration shows what happens when an increase of the number of neighbors causes the system state to change (left side). This change will make the policy perform an action that will change the send interval of the application (right side).

specific parameter in a protocol configuration to a new value.

The following code written in the C programming language is a simple example of a policy where the only state that is used is the number of neighbors for this node. It will reconfigure the send interval when the number of neighbors change. More than two neighbors will cause a configuration of a longer send interval, and fewer neighbors will cause a shorter send interval.

```
int perform_action() {
  if(number_of_neighbors > 2) {
    set_send_interval(30);
  } else {
    set_send_interval(10);
  }
}
```

In the RELYonIT project we use a table-based model for the policies. Each state corresponds to an entry in the table that contains the action to perform in that state. The action can sometimes be a *null action*, which entails that no action will be performed when entering that state.

Figure 4.1 shows the effect of performing an action that reconfigures the send interval when the number of neighbors changes. In this case it is assumed that the policy is 9 elements large and that in the second state there is an action that reconfigures the send interval.

In the approach we have taken all the nodes in a deployed sensor network use the same configuration policy, but might end up with different actual configurations as their system states might be different from each other. One node might have just one neighbor while others have several—this will cause the same policy to configure these nodes differently if the number of neighbors is included in the state. This is in contrast to approaches like pTunes where all nodes have the same system parameters [25].

4.2.3 Learning Mechanism

We are using an approach based on reinforcement learning for the process of learning policies. The learning is performed during simulation using a plug-in for the Cooja simulator [19]. The specific learning mechanism that we use is based on first visit Monte Carlo learning [22]. A utility function, which we describe below, provides the reinforcement learning with the rewards required by the learning process. These rewards are based on the application requirements as specified by the user.

4.2.4 Utility Function

As mentioned above, the learning process is based on a *utility function* that specifies the value of being in a given state and this value is what the learning process tries to maximize.

In RELYonIT the application's dependability requirements are specified as discussed in Deliverable D-3.1 with a formulation that treats all but one performance objective as constraints. The performance objectives are system lifetime T, data yield R, and latency L.

For example, the dependability requirements in the ventilation on demand use case (see Deliverable D-4.1) can be formulated as:

$$\begin{array}{llcll} \text{Maximize} & T(\mathbf{c}) & & & \\ \text{Subject to} & R(\mathbf{c}) & \geq & 75\% \quad \textit{probability} & 1 \\ & R(\mathbf{c}) & \geq & 90\% \quad \textit{probability} & 0.8 \\ & R(\mathbf{c}) & \geq & 95\% \quad \textit{probability} & 0.5 \\ & L(\mathbf{c}) & \leq & 5min \quad \textit{probability} & 1 \\ & L(\mathbf{c}) & \leq & 1min \quad \textit{probability} & 0.8 \end{array} \tag{4.1}$$

When applying any reinforcement learning approach to the problem at hand, one faces a similar issue than the one described in Section 3.2.1. The reinforcement learning mechanisms require a utility function to determine the rewards, which tends to be structurally different compared to the requirement specification in Equation 4.1. In our case, the general form of such utility function is:

$$utility = w_1 * P_1 + w_2 * P_2 + ... + w_n * P_n \tag{4.2}$$

Unfortunately, it is not possible to determine an analytical one-to-one mapping between the general form of Equation 4.1 and 4.2, and the solution space will be, at least to some extent, different between the two. To address this issue, we partly apply the same approach described in Section 3.2.1. To integrate the constraints in the objective function, we give them a higher weight than the original optimization objective. By doing so, we penalise configurations where the constraints are most likely to be violated.

However, embedding the probability appearing within the constraint specifications into the utility function as described in Section 3.2.1 is also not possible here, as simulations happen by directly running the nodes firmware and without referring to the protocol models. We therefore post-process the results of each simulation to check the probabilities that the original constraints were possibly violated during a single learning iteration. If the resulting probabilities do not match the original constraints, we further modify the weight of the corresponding performance metric in the next iterations to steer the learning process towards better configurations w.r.t.

the latter performance metric. Currently, this process occurs manually, but limited changes to the implementation would be required to embed it within the automatic learning process.

4.2.5 Simulation Framework for Off-Line Learning

We use the Cooja simulator and its possibility to accurately emulate sensor nodes such as Tmote Sky and Wismote. This makes it possible to reuse the firmwares that are executed on the real sensor network in the simulator, which will make the node behavior in simulation as close as possible to the real-world behavior.

Cooja is typically user driven and runs one simulation and then stops. This behavior is not suitable for reinforcement learning since reinforcement learning typically needs many repetitions of the same scenario. We design and implement a new extension for Cooja that is able to run multiple simulation rounds of the same scenario. This extension uses the same simulation configuration files as Cooja and after the regular simulation it restarts the scenario at fixed time intervals. Before resetting the scenario the learning process is executed.

4.3 Cooja Simulator Plugin

The learning framework in Cooja consists of a learning plugin that runs series of learning episodes, evaluates the performance and updates the policy. The plugin and its relation to other parts of Cooja is shown in Figure 4.2. The evaluation is performed via scripts in the simulation file that can be generated based on the application requirements.

The simulation file also sets the number of episodes to simulate and the length of each episode. The plugin calls an initialization script before starting each episode and then starts up all the nodes after an individual random delay. The delay is to prevent all nodes from starting at exactly the same time, something that is only possible in simulations. During the execution of the episode all nodes use the current configuration policy. Log messages from the emulated nodes can be used by a log script to calculate performance metrics such as latency for messages, parse energy consumption data, etc. When the episode is finished an evaluation script evaluates the performance of the application. This performance value, i.e., the utility, is then used for updating the policy using an RL-based learning mechanism.

The simulation plugin does not have any explicit knowledge about the utility function or the application requirements. Nor does it know what the different configuration actions will result in. This information is partly coded in the Python scripts (the utility function) and partly in the sensor node code (the configuration actions).

4.3.1 Episode Execution

When Cooja loads a simulation, it sets up the network topology, loads firmware files into emulated memory, and initializes all loaded emulator instances. This phase takes quite some time, and since learning mechanisms such as reinforcement learning need many runs, it is very important to optimize setup times. This optimization is done by keeping the same simulation running and avoid a complete reload between the episodes. Instead, we perform a reset of each node at the time they should start in the episode. In addition to the reset, each node that is not active is also removed from the radio medium in Cooja to avoid unwanted disturbance.

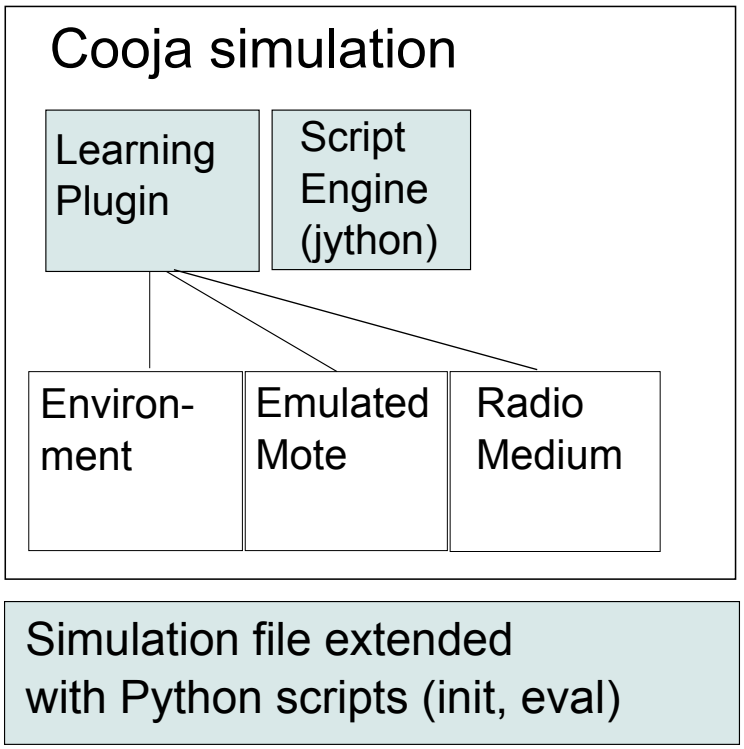


Figure 4.2: The RL plugin for Cooja contains the reinforcement learning implementation. It is supported by a Jython module that executes the Python scripts in the simulation file. These scripts compute the utility that is used by the learning process.

4.3.2 Policy Execution

In each episode, the current policy is set into each node. This is implemented by writing to a specific address that corresponds to the memory where the policy is stored in the node. The C source code in the node is mapping its state into a state index in the policy, and uses that to read and perform a configuration action from the policy.

The memory area for the policy in each node is monitored using watchpoints in the MSPSim emulator [10], which emulates the node hardware within Cooja. Each access is recorded in order to provide this information as input to the policy update.

4.3.3 Policy Updates

The policy is updated after each episode in a manner based on the total utility achieved during the episode. The learning process is based on first-visit Monte Carlo policy iteration [22]. The learning updates a Q-table, which maps all the combinations of system state and action to values. Each value represents an estimation of the expected utility when performing the action in the system state. The Q-table is then transformed to a policy by picking the action (per state) that has the highest expected utility value.

After each episode:

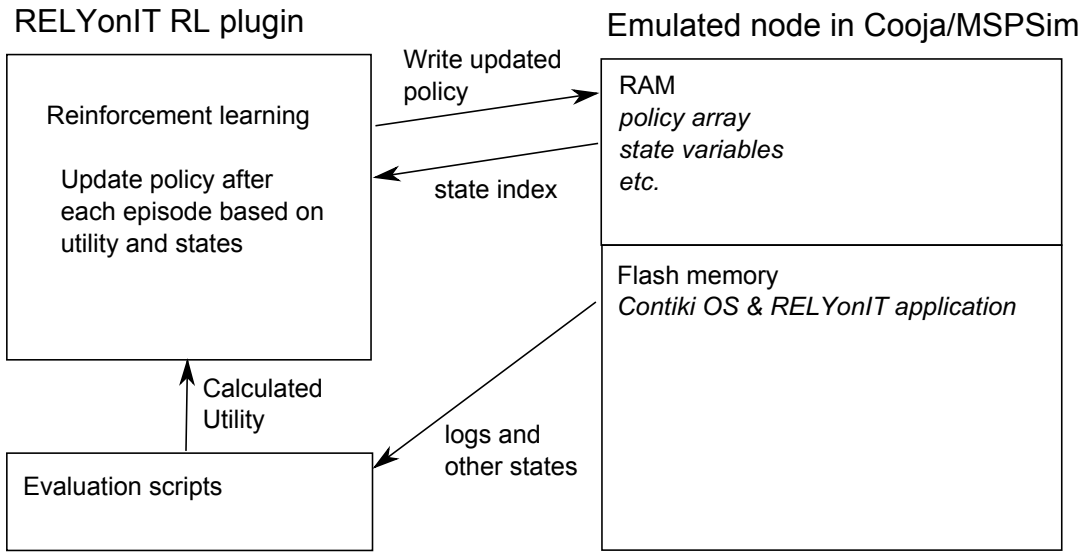


Figure 4.3: Interaction between the RL plugin and the emulated node are based both on internal state of the nodes such as the policy array and external state such as communication and log output from the nodes.

1. Call the evaluation script to compute the total utility for the episode.
2. Go through all states visited during the episode and calculate new state-action values for the Q-table based on the utility.
3. Update the policy based on the Q-table.
4. Use a random decay function to decide if a random action should be performed and possibly update the policy with a random action.
5. Store the updated policy in each node by writing directly to the node's memory in preparation for next episode.

4.3.4 Generation of Environmental Parameters

In Cooja, environmental parameters that affect the protocol performance can be simulated as well. We discuss this here for the example of interference generation but similar techniques can be used also for temperature.

Cooja has a Disturber mote type that can create interference on selectable channels. We modify this Disturber mote to generate periodical interference at a specific fraction of time. The initial Disturber mote implementation generated constant interference, but this is insufficient for the reinforcement learning to test a large variety of scenarios in which the network must operate well under. In the simulations where we run reinforcement learning, one or more Disturber motes are placed at various locations in the network, so as to ensure that the reinforcement learning tests what happens when the interference affects the network topology in different ways.

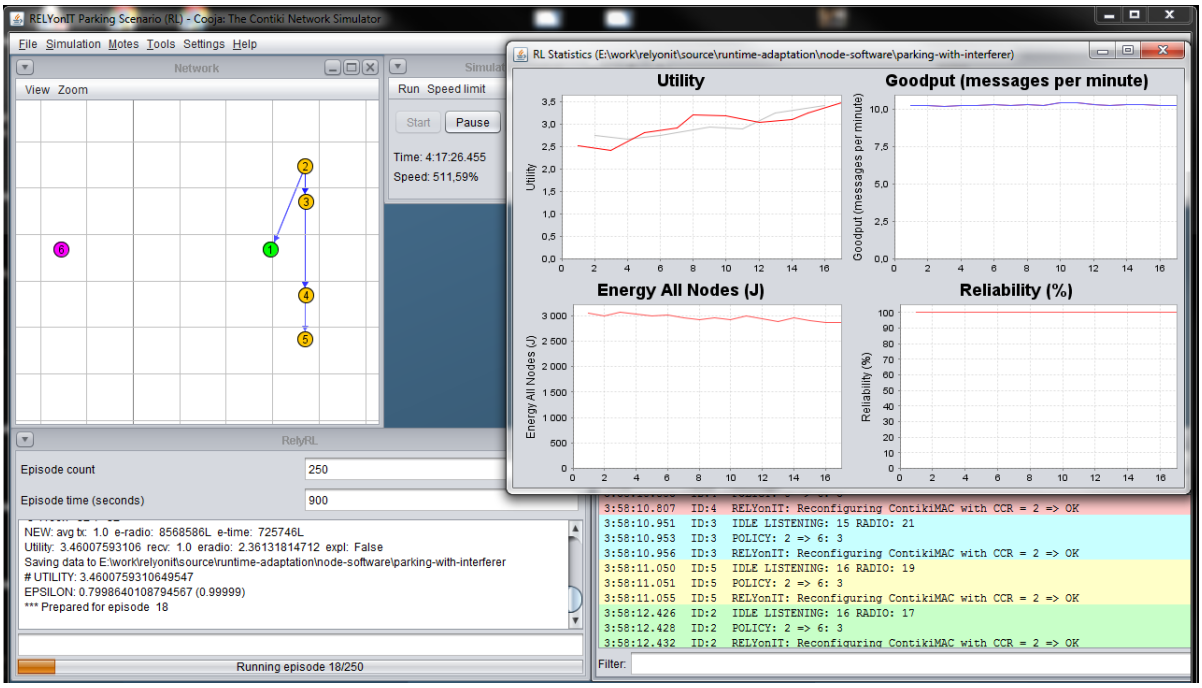


Figure 4.4: Screenshot of Cooja running the RL plugin. The network topology is illustrated in the upper left corner. The utility graph illustrating the learning progress is shown to the right. In the bottom part of the picture are the RL-plugin control window and the serial output from the nodes.

A simple approach to generate interference is to have a semi-periodic pattern of interference according to a specific rate as proposed by Boano et al. [2]. Another approach is to generate the interference based on the RELYonIT environmental model of a specific industrial deployment. Such an interference model is usually compiled into a cumulative distribution function of time intervals when the interference in the channel exceeds a certain threshold above the noise floor of the deployment.

We have chosen the first approach for our initial interference generator. Based on our experience, the simpler approach of semi-periodic interference is sufficient as long as the interference rate considerably exceeds that of the environmental model. A more detailed pattern of interference has typically been learned in environmental conditions that are within the environmental model. Once the model is violated, however, the interference pattern may look considerably different. In such circumstance, another source of interference could have entered the environment, or existing sources could have changed their interference generation pattern. Hence, generating stronger interference based on the pattern observed in normal environmental conditions is unlikely to add value over the semi-periodic interference generation.

Our reinforcement learning is based on the assumption that the interference follows an unpredictable pattern while the environmental model is violated. To take this assumption into account in our simulations, we add random jitter to each time interval when the interference generator is on and off. This jitter also ensures that nodes do not synchronize to the interference.

4.4 Node Runtime

We implement runtime adaptation as a separate application running atop the Contiki operating system in each network node. The implementation consists of two parts: 1) the runtime adaptation process, which executes its configuration policy once a runtime assurance alarm has been triggered; and 2) the configuration distribution protocol, which enables the sink to communicate updated configuration settings and to initiate switches between configuration states. In the following, we describe each of these parts of the implementation.

4.4.1 Runtime Adaptation Process

The purpose of the runtime adaption process is two-fold. First, it has a state aggregator that combines node-local information into a single value that describes the state of the node. This state information corresponds to the system state used in the Q-table during the reinforcement learning phase and is the input to the configuration policy as described in Section 4.3.3. The node-local information includes things like network statistics and power consumption. Second, the runtime adaption process uses this aggregated state value to adapt the node configuration using the pre-compiled configuration policy if needed. In this way the runtime adaption process allows the network to operate under harsh conditions that violate the environmental model.

We implemented the runtime adaptation process as a separate Contiki process that periodically evaluates the node-local information using the state monitoring mechanism as described in Section 4.2.1. This is triggered through a runtime assurance alarm which is generated when an environmental model violation is detected as described in Deliverable 1.3—*Report on Runtime Assurance*. The runtime adaptation process is notified via a callback when the runtime assurance state is changed.

We retrieve network statistics via Contiki's built-in Rime statistics API. The power consumption data is collected from Contiki's software-based power profiling tool, which provides accurate bookkeeping of the power consumption of each system component [9]. This information is then aggregated into one numeric state value representing the node's local state.

The runtime adaptation process uses the runtime interface of protocol parameterization described in Section 3.3 to adapt dynamically the configuration in the node. Unlike the constant performance states generated during the protocol optimization phase, the runtime adaption process adds another performance state in memory that can be adjusted during runtime. When the runtime assurance raises an alarm, the runtime adaption process will switch to use its own performance state. To adjust the configuration values, the runtime adaptation process has a list of available configuration actions, and the configuration policy will map the node's state to a configuration action as described in Section 4.2.2.

In addition to being used locally on the node during deployment, this information is used by the learning mechanism during simulation.

4.4.2 Configuration Distribution Protocol

Because the reinforcement learning may be executed not only before a network is deployed but multiple times thereafter, it is necessary to provide the means for a network administrator to

disseminate updated configuration policies to the network nodes. To this end, we have designed and implemented the RELYonIT Configuration Distribution Protocol (CDP).

CDP is a broadcast-based protocol that periodically transmits the current configuration, as specified by the sink node. CDP is built on top of the Rime communication stack [8]. It uses Rime's implementation of the Trickle mechanism [16] to suppress redundant messages and to increase the transmission intervals when the configuration state has been disseminated with high probability. Hence, the energy cost of these extra configuration messages is negligible.

The messages consist of a 16-byte field that specifies the performance state to switch to, and a 1-byte field to allow the sink to signal to all nodes that an alarm is raised. For testing purposes, they also include a set of fields that allow a network operator to make a request to change to a specific configuration not specified among the performance states in the network.

4.5 Evaluation

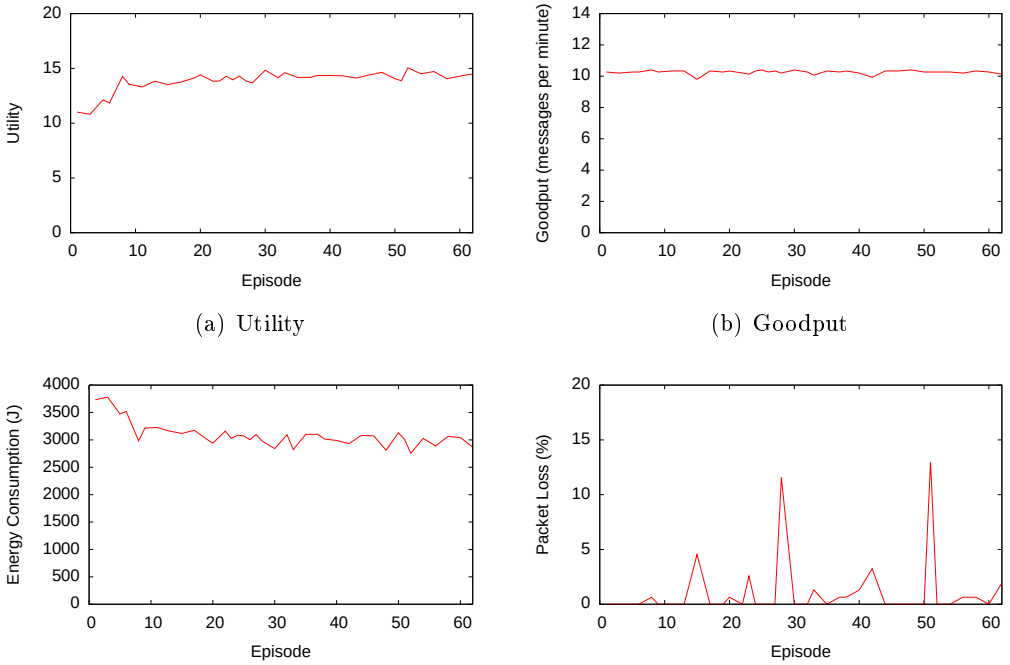
For the learning process, we setup a simulation for the experiment with one sink node, four parking sensor nodes, and one interferer node in a single-hop topology. The interferer node is a semi-periodic interferer running 33% of the time, and is placed close to the sink node. During the reinforcement learning phase, we disabled the runtime assurance, and forced all nodes into runtime adaptation mode to speed up the learning phase.

For the utility function, we use the weighted sum of the application message reception rate subtracted by the energy consumption for all nodes. For the state parameters on each individual node, we use the energy consumption and the radio idle listing time, where the latter is the time the radio wakes up and listens without receiving any data. The state also includes a Boolean parameter that determines whether the node is the sink or a client.

We use four actions to set the channel check rate (CCR) in the duty-cycling MAC protocol in four different classes ranging from low latency to low energy consumption. Since the runtime adaptation allows each node to adapt its configuration individually, the runtime adaptation sets a fixed strobe time when enabled to allow all nodes to communicate regardless of each node's CCR configuration. The strobe time is set to the lowest CCR value.

In the reinforcement learning phase, we run 60 learning episodes, 15 simulated minutes each, with an explore factor of 0.8 and a decay factor of 0.9999 to control when to do an exploration action.

Figure 4.5 shows the progress made during the reinforcement learning phase. As more episodes are run, the utility function exhibits an increasing trend, which entails that we come closer to reaching our performance objective. The energy graph shows that the initial energy consumption has been reduced, without a corresponding reduction of packet delivery rate. These results show that reinforcement learning is a promising approach to generate a configuration that leads to acceptable performance under a large variety of challenging environmental conditions.



(a) Utility (b) Goodput

(c) Energy (d) Packet Loss

Figure 4.5: Learning a configuration policy in a single-hop network running the RELYonIT smart parking application. The network is disturbed by a semi-periodic interference generator to simulate a harsh environment in which the runtime assurance would have been triggered.

5 Conclusions

This deliverable summarizes the results of work package WP3, specifically of tasks T3.2 and T3.3. It provides a detailed description of the approaches and software artifacts for protocol selection, parameterization, and adaptation that were developed within the project. In addition, it depicts the development process involving these tools. With this set of tools enables it is possible to create an environment-specific protocol selection and configuration that ensure a desired dependable performance.

Currently, the approach and tools only cover two environment factors, temperature and interference, and in our implementation they are primarily focused on the MAC layer. Nevertheless, the basic approach is also applicable for different protocol classes and could be easily extended to further environment factors.

Bibliography

[1] C. Bettstetter, G. Resta, and P. Santi, "The node distribution of the random waypoint mobility model for wireless ad hoc networks," *Mobile Computing, IEEE Transactions on*, vol. 2, no. 3, pp. 257–269, 2003.

[2] C. A. Boano, T. Voigt, N. Tsiftes, L. Mottola, K. Römer, and M. A. Zúñiga, "Making sensornet mac protocols robust against interference," in *Proceedings of the 7th European conference on Wireless Sensor Networks*, ser. EWSN'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 272–288. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11917-0_18

[3] C. A. Boano, T. Voigt, C. Noda, K. Romer, and M. Zúñiga, "Jamlab: Augmenting sensornet testbeds with realistic and controlled interference generation," in *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*. IEEE, 2011, pp. 175–186.

[4] C. A. Boano, M. Zúñiga, J. Brown, U. Roedig, C. Keppitiyagama, and K. Römer, "Templab: a testbed infrastructure to study the impact of temperature on wireless sensor networks," in *Proceedings of the 13th international symposium on Information processing in sensor networks*. IEEE Press, 2014, pp. 95–106.

[5] M. Bor, K. Langendoen, C. A. Boano, F. J. Oppermann, K. Römer, A. V. Rico, P. M. Montero, R. S. Hernández, I. Vilajosana, M. Dohler, M. Montón, U. Roedig, A. Mauthe, G. Coulson, T. Voigt, L. Mottola, Z. He, and N. Tsiftes, "D-4.1 – report on use case definition and requirements," http://www.relyonit.eu/, RELYonIT: Research by Experimentation for Dependability on the Internet of Things, Tech. Rep., Mar. 2013.

[6] J. Brown, J. Vidler, I. E. Bagci, U. Roedig, C. A. Boano, F. J. Oppermann, M. Baunach, K. Römer, M. A. Zúñiga, F. Aslam, and K. Langendoen, "D-1.3 – report on runtime assurance," http://www.relyonit.eu/, RELYonIT: Research by Experimentation for Dependability on the Internet of Things, Tech. Rep., Nov. 2014.

[7] M. Doddavenkatappa, M. C. Chan, and A. L. Ananda, "Indriya: A low-cost, 3D wireless sensor network testbed," in *Testbeds and Research Infrastructure. Development of Networks and Communities*. Springer, 2012, pp. 302–316.

[8] A. Dunkels, F. Österlind, and Z. He, "An adaptive communication architecture for wireless sensor networks," in *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys)*, Sydney, Australia, Nov. 2007.

[9] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He, "Software-based on-line energy estimation for sensor nodes," in *Proceedings of the IEEE Workshop on Embedded Networked Sensor Systems (IEEE Emnets)*, Cork, Ireland, Jun. 2007.

[10] J. Eriksson, A. Dunkels, N. Finne, F. Österlind, and T. Voigt, "Mspsim – an extensible simulator for msp430-equipped sensor boards," in *Proceedings of the European Conference on Wireless Sensor Networks (EWSN)*, Delft, The Netherlands, Jan. 2007.

[11] R. Fonseca, O. Gnawali, K. Jamieson, and P. Levis, "Four-bit wireless link estimation." in *HotNets*, 2007.

[12] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection tree protocol," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2009, pp. 1–14.

[13] V. Handziski, A. Köpke, A. Willig, and A. Wolisz, "Twist: a scalable and reconfigurable testbed for wireless indoor experiments with sensor networks," in *Proceedings of the 2nd international workshop on Multi-hop ad hoc networks: from theory to reality*. ACM, 2006, pp. 63–70.

[14] O. Landsiedel, "ORW GitHub," https://github.com/olafland/orw/commit/cefe4ec7b0807c353df29a39a823e7d44c2c8118, 2014, [Online; accessed 18-Oct-2014].

[15] O. Landsiedel, E. Ghadimi, S. Duquennoy, and M. Johansson, "Low power, low delay: opportunistic routing meets duty cycling," in *Proceedings of the 11th international conference on Information Processing in Sensor Networks*. ACM, 2012, pp. 185–196.

[16] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks," in *Proceedings of the USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, Mar. 2004.

[17] S. Moeller, A. Sridharan, B. Krishnamachari, and O. Gnawali, "Routing without routes: The backpressure collection protocol," in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. ACM, 2010, pp. 279–290.

[18] L. Mottola, T. Voigt, F. J. Oppermann, K. Römer, and K. Langendoen, "D-3.1 – report on specification language," http://www.relyonit.eu/, RELYonIT: Research by Experimentation for Dependability on the Internet of Things, Tech. Rep., Jun. 2013.

[19] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-level sensor network simulation with cooja," in *Proceedings of the First IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006)*, Tampa, Florida, USA, Nov. 2006.

[20] E. Reehuis and T. Bäck, "Mixed-integer evolution strategy using multiobjective selection applied to warehouse design optimization," in *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO)*. New York, NY, USA: ACM, 2010, pp. 1187–1194.

[21] A. E. Smith and D. W. Coit, "Penalty functions," in *Handbook of Evolutionary Computation*, T. Bäck, D. B. Fogel, and Z. Michalewicz, Eds. Bristol, UK: IOP Publishing Ltd., 1997.

[22] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[23] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: Scalable sensor network simulation with precise timing," in *Proceedings of the 4th international symposium on Information processing in sensor networks*. IEEE press, 2005, p. 67.

[24] T. ZHENG, "Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks," *Listening*, vol. 10, p. 8, 2004.

[25] M. Zimmerling, F. Ferrari, L. Mottola, T. Voigt, and L. Thiele, "ptunes: Runtime parameter adaptation for low-power mac protocols," in *Proceedings of the 11th international conference on Information Processing in Sensor Networks*, 2012, pp. 173–184.

[26] M. A. Zúñiga, C. A. Boano, J. Brown, C. Keppitiyagama, F. J. Oppermann, P. Alcock, N. Tsiftes, U. Roedig, K. Römer, T. Voigt, and K. Langendoen, "D-1.1 – report on environmental and platform models," http://www.relyonit.eu/, RELYonIT: Research by Experimentation for Dependability on the Internet of Things, Tech. Rep., Jun. 2013.

[27] M. A. Zúñiga, F. Aslam, I. Protonotoarios, K. Langendoen, C. A. Boano, K. Römer, J. Brown, U. Roedig, N. Tsiftes, and T. Voigt, "D-2.1 – report of optimized and newly designed protocols," http://www.relyonit.eu/, RELYonIT: Research by Experimentation for Dependability on the Internet of Things, Tech. Rep., May 2014.

[28] M. A. Zúñiga, I. Protonotarios, S. Li, K. Langendoen, C. A. Boano, F. J. Oppermann, K. Römer, J. Brown, U. Roedig, L. Mottola, and T. Voigt, "D-2.2 & D-2.3 – report on protocol models & validation and verification," http://www.relyonit.eu/, RELYonIT: Research by Experimentation for Dependability on the Internet of Things, Tech. Rep., Nov. 2014.

