Towards Practical Debugging of Wireless Sensor Network Applications

Authors:
Matthew Bradbury (0921660)
Tim Law (0918647)
Ivan Leong (0830934)
Daniel Robertson (0910210)
Amit Shah (0904778)
Joe Yarnall (0905247)

Supervisor:
Dr. Arshad Jhumka

October 2012 - May 2013
Abstract

Debugging tools are vital for developers to produce reliable software, however traditional tools are less useful when developing software for new system paradigms such as wireless sensor networks. As wireless sensor networks (WSNs) become increasingly prevalent in our lives it will become ever more important that the software they are running works reliably and to do this debugging tools will be required. This project investigates how predicates can be specified and accurately checked throughout WSNs and how errors can be reported to a base station. We also develop a system that facilitates reporting predicate statuses to a desktop application.

Keywords - Wireless Sensor Networks; Debugging; Reliability; Predicate Checking;
# Contents

## 1 Introduction

1.1 What is a Wireless Sensor Network? ........................................... 3
1.2 The Problem - Debugging Distributed Systems .......................... 3
1.3 Related Work
   1.3.1 Classes of Distributed Predicates ........................................ 4
   1.3.2 Fault-Error-Failure Cycle .................................................... 4
   1.3.3 Existing Sensor Network Predicate Checking Tools ................. 5
   1.3.4 Practical Sensor Network Deployments ................................. 7
   1.3.5 Tools and Platforms ......................................................... 11
   1.3.6 Communication ............................................................... 12
1.4 Summary .................................................................................. 15

## 2 Problem Statement and Formalisation

2.1 Plan ....................................................................................... 17
2.2 Model ..................................................................................... 17
2.3 Algorithm Descriptions .......................................................... 18

## 3 Predicates

3.1 Types of Predicates .................................................................... 21
3.2 Predicate Evaluation
   3.2.1 Domain Specific Language ................................................... 24
   3.2.2 Example Predicates ................................................................ 26
3.3 Virtual Machine
   3.3.1 Opcodes .............................................................................. 28
   3.3.2 Bytecode .......................................................................... 30
3.4 Examples of Compiled Programs ................................................ 32
3.5 Testing ..................................................................................... 33
3.6 Data Transmission
   3.6.1 Local Event ......................................................................... 34
   3.6.2 Local Periodic ....................................................................... 35
   3.6.3 Global ................................................................................. 36
   3.6.4 Reply ................................................................................. 37
3.7 A Consideration ......................................................................... 38
3.8 A Distance Issue ........................................................................ 40
3.9 Optimising on Predicate Structure ............................................. 41

## 4 Implemented Algorithms

4.1 Container Library ...................................................................... 42
4.2 Network Library
   4.2.1 Multi-Packet Unicast ........................................................... 43
   4.2.2 N-Hop Request ..................................................................... 43
   4.2.3 N-Hop Flood ......................................................................... 46
   4.2.4 Event Update ........................................................................ 47
   4.2.5 Tree Aggregation ................................................................. 49
   4.2.6 Neighbour Detect ............................................................... 51
4.3 TDMA ....................................................................................... 52
4.4 Testing ..................................................................................... 53
4.5 Visualisation Tool ....................................................................... 53
4.6 Summary .................................................................................. 55

## 5 Results

5.1 Methodology ............................................................................. 56
5.2 Analysis ................................................................................... 57
5.3 Conclusion ................................................................................ 60

## 6 Practical Experience

6.1 Contiki Logical Channels .......................................................... 61
6.2 Power Levels ............................................................................. 61
6.3 Sensor Data Conversion ............................................................ 61
6.4 Uploading to the motes ............................................................. 61
1 Introduction

1.1 What is a Wireless Sensor Network?

A wireless sensor network, or WSN for short, is a collection of computing devices called motes; these motes are capable of short range wireless communication and they have the ability to sense their surrounding environment [60]. The network forms a distributed system that can perform a variety of distributed algorithms, usually data gathering and similar tasks. To communicate, each node is equipped with a radio that allows it to send and receive messages to neighbouring nodes within a limited range. To sense the environment motes typically have a range of embedded sensors such as heat, light, humidity and many others. They also contain a simple central processing unit (CPU), which is programmed to control the hardware on the motes. The CPU also processes events which are triggered by the hardware (such as messages being sent and received) and it also handles any other computation necessary for the operation of the system. As the platform is wireless the motes do not operate on a mains power supply, instead they are powered by energy stored in a battery. Wireless sensor networks as a platform have a wide range of practical applications that stretch from battlefield intelligence for the military [8, 95, 65] to industrial process monitoring for manufacturing companies [28, 53, 115].

A defining characteristic of designing applications for wireless sensor networks is the restricted and finite energy supply available to each node. Therefore, wireless sensor nodes tend not to use expensive broadcasting protocols such as IEEE 802.11 [60], but instead use much simpler alternatives to save energy. For example wireless protocols such as IEEE 802.15.4 ZigBee [27, 68] are designed to be used by wireless sensor networks and have a lower energy usage associated with them. Some applications rely on even lower level behaviour specified by a certain MAC layer [13, 90, 25], these applications involve a trade-off between development time and energy usage, where simplicity is often sacrificed for decreased energy usage. Using these simple protocols unfortunately has the downside that broadcasts are subject to several types of collisions and message losses; therefore it is very important that the software running on the nodes is designed to handle these cases. Being battery powered means that development of applications for Wireless Sensor Networks is fundamentally limited to maximising the system’s lifetime so that the highest utility can be achieved from the network [54].

As wireless sensor nodes operate in harsh outdoors condition [106, 113], there is a high probability of them failing. These faults can range from hardware damage caused by environmental conditions or tampering, software bugs, or simply a denial of service caused by nodes running out of power. So algorithms and software need to be designed to handle these potential failures, otherwise they risk catastrophic failure when they encounter these issues.

Wireless sensor nodes are designed with the intention for them to operate in remote and traditionally unreachable locations with no human input for the lifetime of their operation [71]. Given this, a defining characteristic of a WSN system is that any applications developed for it must be self-configuring in nature; this is, the system must be able to organize itself and the network with no external input [93]. Evidently, solutions to this issue are often considered hand in hand with the problem of network robustness and fault tolerance.

While a limited energy source, self-configuration and network robustness are the predominant characteristics of a wireless sensor node, there are numerous other traits or issues that can be considered. For instance it is possible for these nodes to be mobile (examples include an ad-hoc network of PDAs or motes built into soldiers’ helmets) [53], this can lead to very interesting behaviour in handling communication between these nodes.

1.2 The Problem - Debugging Distributed Systems

Developing a distributed system is considered a particularly challenging task, more so than a traditional application. There are several reasons for this. Firstly, within a distributed system, multiple processes must execute in parallel. The result of this is that variables may be updated independently or in response to other processes which can lead to a myriad of synchronisation and timing issues that the developer must account for. Secondly, traditional programming languages
are not well suited to develop distributed programs [81 111].

In any system developed by humans, software or otherwise, there is the potential for mistakes. Mistakes can be benign or they can cause unintended behaviour and system failures. Developing tools to detect these bugs and notify the developer so they can be corrected is an incredibly important part of any toolchain. For example the GNU toolchain has utilities such as gdb [104], which allows developers to place breakpoints in code that will halt the program’s execution at that state so that it can be examined. There are also numerous tools that look for memory issues (such as valgrind [88]) [22 85], security flaws [112 45] and many other classes of bugs.

Developing distributed systems is a difficult task; however, debugging these systems can be even more challenging [111]. When considering a distributed system if you want to examine the state of a system at a given point you cannot simply set a breakpoint in your local binary. The solution to this debugging is non-trivial, due to the difficulties that arise from distributed systems being non-deterministic as a result of the nature of their message communications [73 46]. be it when the message started transmission, how long it took, if it succeeded or in what order transmissions occurred. Every time a distributed program is run it is possible for a different result to be obtained, due to the different order of execution. This goes against one of the usual assumptions of debugging traditional applications where it is assumed that one execution with a set of inputs will execute in exactly the same way again with the same set of inputs [74, Chapter 10].

As the execution may be different each time in a distributed system, it is not suitable to wait for a bug to occur, and then try to work out where it is. Rather, the system needs to be self-evaluating its state as it executes the distributed program. If a fault is detected, then the debugging tools will report the issue. One way to do this is to test if the system satisfies some global predicate, in which area there has been much work to find and check different classes of these predicates [50 111 49]. However, of all the work that has been done, little of it has focused on wireless sensor networks where an important focus is perhaps the trade-off between accuracy and the report-ability of a predicate with the aim of reducing energy usage. In this report we will discuss our development of just such a set of tools, we intend to focus on developing a system that can accurately evaluate predicates and provide useful information about real sensor networks running outside of a simulator.

1.3 Related Work

1.3.1 Classes of Distributed Predicates

To begin with, it is important to understand what predicates are relevant to distributed systems. First we have a distinction between global and local predicates; global predicates involve taking a consistent global snapshot of the system and checking whether the snapshot satisfies the global predicate [49], and local predicates which instead work with a subset of the network [50]. These predicates also have a notion of stability – a stable predicate will remain true once it has turned true (e.g. termination), whereas unstable predicates can alternate between true and false. Finally there is a distinction between weak and strong, where a weak predicate holds if there exists an observation in which the predicate is true and a strong predicate holds if it is true for all observers of the distributed computation [50 29]. Knowing what classes of predicates there are is important because, when checking certain properties of a system, a certain class of predicate will be required and thus a certain implementation will be needed to ensure the predicate is correctly checked. An example of this is when running an algorithm using global snapshots to detect stable predicates, that same application may not be suitable to detect unstable predicates because the predicate could switch to false and then back to true before the next snapshot.

1.3.2 Fault-Error-Failure Cycle

It is evident that the types of predicates are important to consider when developing a predicate checking mechanism. However, it is also important to consider the types of errors that these predicate checking algorithms can detect. To understand this it is first important to understand the errors themselves and how they can arise. This can be done by examining the fault-error-failure cycle. This cycle says that a fault caused by either some external influence (e.g. radiation leading
to bit-flips in memory or internal influence (e.g. code bugs) may or may not lead to an error; this is the activation step. An error is the manifestation of the fault (e.g. memory holding the incorrect value due to bit-flips). An error can then lead to more errors through a step called propagation or if the error propagates outside the system boundary then it becomes a failure, the failure of the system is an observable deviation from the system’s specification (e.g. allowing doors to be opened that should remain closed). It is not always the case the faults lead to errors, or errors lead to failures, sometimes multiple faults or errors are required to cause a single error or failure, respectively.

It is also important for us to consider which step of the fault-error-failure cycle we are checking for and measuring in our predicate language, we have three choices: faults, errors and failures. Faults would be a poor choice for our predicate language for two reasons i) faults don’t necessarily activate and propagate into failures so any system that detected faults could detect a lot of false positives ii) fault prevention is traditionally achieved by strict quality control techniques employed during the design and manufacturing of hardware and software rather than by predicate checking software. This leaves just errors and failures. Failures are trivial to detect since by their definition they are errors that are detectable by the user. However, the thing that makes them so easy to detect also makes detecting them practically useless to a developer since the damage will already be done. This just leaves errors, errors can be measured if there is a dedicated program checking the state and comparing it to an expected state. For example an ECC (Error Correcting Code) such as a Hamming Code can be used to detect and correct an error (in this case a bit-flip) in some memory after a fault (such as a voltage surge).

Much of what has been discussed has involved transient faults such as those caused by environmental conditions. However, there is a class of faults that are a lot more common and much easier to resolve – faults caused by software bugs. These faults can lead to programs ending up in the wrong state and performing incorrectly. There has been a certain amount of work that looks into detecting traditional distributed system bugs (such as deadlock) in wireless sensor networks. However, there has been little work in looking into providing tools to aid in system debugging.

### 1.3.3 Existing Sensor Network Predicate Checking Tools

There are a number of existing predicate checking solutions that have already been developed which exhibit a more practical focus than the aforementioned theoretical work into global predicate detection.

**H-SEND** One of these solutions is H-SEND, which stands for Hierachichal Sensor Network Debugging. H-SEND is a framework for detecting faults in sensor networks, it was designed to minimise energy consumption and be capable of handling very large networks. As part of the implementation developer must specify invariants within their code using a custom grammar, these invariants are then semi-automatically inserted during compilation. If an invariant is violated at runtime actions are taken (such as increased logging frequency, or an error message to the base station), using these responses developers can use the information to fix the software, which possibly includes uploading a patched version of the firmware.

Of the invariants that can be specified, there are typically three different dichotomies: (i) Local vs. Multi-node, (ii) Stateless vs. Stateful and (iii) Compile-time vs. Run-time. The first indicates whether the predicate needs information about the node it is being evaluated on (Local) or other nodes in the network (Multi). The second is if the invariant depends on the node’s execution state (Stateful) or if it doesn’t (Stateless). The third indicates whether the invariant involves values that are fixed at compile time (such as integer constants), or if it compares against values obtained during execution (such as neighbouring states or previous states).

H-SEND is optimised for WSNs in a variety of ways. For example, it minimises overhead by buffering messages it needs to send, and piggybacking them on existing network traffic. Due to the hierarchical nature of the protocol, multi-node invariants can be checked efficiently at the closest parent node with all the required information.
Sympathy  One of the projects that is summarised by H-SEND’s authors [Herbert et al.] in [50] is a method for identifying and localizing failures called Sympathy [94]. Sympathy is intended to be run either in pre- or post-deployment environments where it collects data from distributed nodes at a sink. When insufficient data is received it is taken to imply that there exists a problem (insufficient data is component-defined). The idea is that by monitoring data between components (both actively and passively) the system can identify what kind of failure occurred in a certain area of the network, both of which are very useful when trying to debug a failure.

It does, however, have some downsides. The first is that there is assumed to be no traffic and thus no application traffic or network congestion. These are real issues especially when applying this kind of debugging to a high-throughput sensor network. There are also a number of spurious failure notification, which the authors are working on reducing, by applying a Bayes engine.

DAIKON  Following Sympathy’s attempts to implement a Bayes engine to allow learning to better help classify response messages, there has been work on being able to automatically detect invariants in a system. DAIKON [43] is a system that uses execution traces to produce a list of likely invariants by way of automatically inferring the invariants through the use of a prototype invariant detector.

The set of dynamically-detected invariants depend on observed values and the invariants are an indication of the quality of a test suite. DAIKON consists of two parts; a language specific front end and a language independent inference engine. The former executes the running program and accesses its runtime state to get the required information (consisting of variables and their values). This information contains a subset of relevant variables and only these are written to the trace file. Whether a variable is considered as relevant depends on the type of invariants targeted and whether they are accessible at an instrumentation point. This forms an input to the second part of the system – the invariant inference engine – which uses machine learning techniques and produces a set of detected invariants for the program.

The main challenge with these techniques is deducing the relevance of the invariants as it depends on the programmer’s experience and knowledge of the underlying system. However, these can be improved with techniques like exploiting unused polymorphism and suppressing invariants that are logically implied by other invariants. DAIKON does not require the programmer to specify invariants for the application, however it is not designed for distributed or resource-constrained systems like WSNs.

DIDUCE  One of the issues with DAIKON is that it only detects possible invariants, whereas DIDUCE [55] (Dynamic Invariant Detection ∪ Checking Engine (DIDUCE)) uses a similar methodology to detect the invariants, but it also evaluates them as they are detected. Machine learning is employed to dynamically generate hypotheses of invariants for a system at runtime. The invariants begin extremely strict, and are relaxed over time to allow for new correct behaviour. The machine learning aspect means that developers do not have to specify invariants themselves (as opposed to H-SEND), which proves beneficial since developers often supply invariants that cannot possibly pass [55]. DIDUCE checks against the invariants continually during a program’s operation and reports all violations detected at the end of the run, whereas DAIKON merely presents the user with invariants found. For all its apparent usefulness, unfortunately DIDUCE was designed for large, complex systems rather than lightweight distributed systems with constrained resources such as sensor networks.

NodeMD  An alternative to debugging compared to either specifying a predicate or an invariant, or using machine learning to learn what to check for, is to instead look for the faults that arise after a failure has occurred. This is the approach that NodeMD [70] takes; that by looking for the faults that can cause undesired behaviour, bugs in the system can be identified. NodeMD supports checking a number of fault classes: stack overflow, live-lock, deadlock and application-specific faults. By having an extensible framework, developers of a system can write their own fault detectors and plug them in to NodeMD’s framework. The authors of NodeMD point out that “human interaction is often the only reliable way to address many software issues”, therefore, NodeMD supports recording events that occur to humans can analyse them to find out how a failure occurred. To optimise this format for sensor network, the events are stored in a custom
binary format to save space and reduce the number of messages to transmit it (if it is possible to transmit).

NodeMD also has support for a number of useful features to aid in debugging. The first is a debug mode that is entered when a fault is detected – the debug mode freezes critical parts of the system to prevent the fault from leading to errors. This prevents events such as a context switch after a stack overflow that would end up being performed incorrectly. The debug mode also resets certain OS components to a safe state, so that some components (such as the radio) are usable to report the fault that was detected.

The other two useful features are support for remote debugging and an implementation of dynamic reprogramming algorithms to update the firmware across the network. The remote debugging feature allows a human to access all the available fault information of a sensor node. Parameters can be changed to expose more information when the node is queried and the potentially useful feature of telling the node to restart is also available. Overall NodeMD provides lots of insight into the low level failures in wireless sensor networks.

1.3.4 Practical Sensor Network Deployments

In order to understand how these applications may be useful it is necessary to consider how sensor networks are actually used. Overall, wireless sensor networks occupy a niche market of monitoring and reporting on vast areas. The software is very careful to minimise energy consumption to maximise the lifetime of the network [9] and it is also designed to operate on hardware with limited resources (such as memory) [95]. The following are a few examples of real-world deployments and the experiences obtained through their use.

Habitat Monitoring Habitat monitoring is widely thought to be one of the key wireless sensor network application that is driving research and adoption. The problem involves determining the location of the sensor nodes, which then allows tracking [26] and two of the primary problems in sensor networks: data aggregation and energy efficient multi-hop communications.

The research undertaken by Szewczyk et al. in [106] involves deploying sensor nodes on the Great Duck Island in order to test the long term real-world deployment of sensor networks. The importance of deploying and testing a WSN in the real world is due to the fact that challenges are encountered that are not present in indoor deployments or simulations. The problem becomes one of not just what software needs to run and how, but also consider on what hardware and in what conditions. For example the authors wished to waterproof the mote in order to ensure that it survived dew, rain and flooding. However, when enclosing the motes it was worried that (i) radio transmissions would be interfered with and (ii) sensor data could be affected.

Overall, the deployed motes performed well (logging 1.1 million readings over 123 days), however, there were a number of abnormal results that included: (i) sensor readings outside their range, (ii) “erratic packet delivery” and (iii) failure of motes. The authors point out (in section 2.5) that in the future it would be useful to augment applications with the ability to notify that failures occurred and to perform self-healing.

The performance of the network showed that initially packet loss was up to 20% but that slowly decreased, most likely due to the fact that the size of the network network was reducing over time (as nodes permanently crashed). Due to the low network utilisation (under 5%) Szewczyk et al. initially believed that collisions would not play an important role, however, their results suggested otherwise. Their results suggest that this was caused by clock drift which lead to slot assignments not preventing collisions within their period. This importance of clock drift on TDMA-based MAC protocols is something that would tend not to be found from testing in a simulator because of the slow speed of simulation and the length of time it takes for these small clock drifts to lead to an effect.

Overall, one of the major conclusions of this work is that anomalous sensor data can be used to predict mote failures with a high degree of accuracy. The authors believe that this prediction allows for a high level of pro-active maintenance, and self-organisation and self-healing of the network.
Animal Monitoring  Instead of monitoring the habitat, why not simply directly monitor the animal in question? This is the tack that a number of other wireless sensor networks have taken [64]. The traditional technique was to attach collars that emit radio waves that are used in manual triangulation or take GPS measurements and then recover the hardware attached to the animal to extract the data. Sensor networks provide a way to extract this data automatically in a much easier manner and can also allow for data to be accessed earlier. Much of what is true for habitat monitoring is also true for monitoring animals: the hardware needs to be protected against the weather, the data needs to be extracted as reliably as possible and the battery should last as long as possible to make the deployment cost effective.

ZebraNet by Juang et al. in [64] investigates these issues as well as some of the unique deployment issues related to mounting hardware on animals. For example it is very important that the hardware is light enough for the animal to handle for a long time. This adds a new type of trade-off, when considering energy usage and reliability system developers must also consider weight.

Aside from the usual issues associated with sensor networks, having sensors attached to animals presents another very important problem that needs to be solved. Many of the animals that we wish to monitor, are desired to monitor for very good reasons. Most often it is the case that they are endangered or are often the victims of poaching [100, 20]. By attaching wireless devices that broadcast information, it has been found that attackers can trace their way back to a source. There has been much research to develop ways to mitigate this problem, one example is using fake sources to lure an attacker in an alternate direction than that of the real source [88, 66, 62]. These additional problems that arise as side-effects of communicating through a wireless medium are often difficult and expensive (in terms of energy) to overcome.

Forest Fires  While habitat monitoring has been at the forefront of environmental monitoring, there is also a need to measure weather conditions in order to predict wildland fires that can occur across the globe [103, 56, 57]. Fire behaviour can change drastically, depending on different environmental factors and topological features such as elevation and aspect. It is important for such behaviour to be predicted accurately and wildland firefighters are currently using means like observing current weather conditions and weather forecasts provided by the National Weather Service to do so. While data provided by the weather forecast gives a general overview of fire behaviours, those solely collected by the firefighters (e.g. using a belt-weather kit) only provides data for a sparse number of regions, which is inadequate in making a full evaluation. With wireless sensor networks, firefighters can safely measure weather conditions over a wide range of locations and elevations anywhere within the fire environment.

FireWxNet by Hartung et al. in [56] is a robust multi-tiered portable wireless system, designed to be deployed in a rugged environment. Compared to other application deployments, FireWxNet covers a unique topology which ranges from substantial and sharp elevation differences to an extremely wide coverage area of about 160 square kilometres. Wide area communication coverage and fine-grained local weather sensing coverage are some of the main challenges that were faced by the team.

Some of the network challenges were ensuring a high probability of receiving data from the sensors, regardless of interference and asynchronous links and this was tackled by using a best-effort converge-cast protocol. Other issues included verifying the connectivity between nodes and complications were a result of the sparse nature of the deployment and large changes in elevation between nodes. In addition, a user could not receive verification when adding a node to an already-deployed network and required the resetting of a neighbour node to gain connectivity.

When testing the performance of the network, results showed that the networks which were deployed on different mountains obtained an average yield of only 40% and a 78% unique yield. This may have been the effect of timing limitations, for example some sensors required some settling time once it was activated before it could produce any accurate readings. The CSMA MAC protocol which would back-off in the presence of interference may have also contributed to the number of packets sent. To improve the success rate, they chose the option of resending the packets multiple times, but at the cost of some efficiency. In the future, the team aims at designing protocols such as hop-to-hop ACKS, in the hope of cutting down on the number of packets they send.
Volcano Monitoring

Active volcanoes need to be monitored to predict the likelihood of future eruptions so that early-warning signs can be issued for the evacuation of inhabitants near the volcano. Today, volcanoes use wired arrays of sensors such as seismometers and acoustic microphones to collect seismic and infra-red (low-frequency acoustic) signals and determine a wide range of factors such as sources of volcanic eruptions, interior structure of a volcano and differentiating eruption signals from noise. A typical study can include the placement of several stations around the volcano where each contains a low distribution of wired sensors that collects data to a hard drive. Data is then collected manually in a possibly inconvenient location.

To address the issue of high power consumption of these wired sensors, [Werner-Allen et al.] in [113] introduced the deployment of embedded wireless sensor networks consisting of low-power nodes with minimum CPU, memory and wireless communication capabilities. This allows for long distance real-time monitoring of volcanic activities and reduces the need for manual data collection. The main challenge, however, is that environmental monitoring studies sample at a frequency which is much lower than the sample rate of volcanic time series. This is due to the limited radio bandwidth of the sensors, thus presenting the need for efficient power management techniques and accurate time synchronization of the nodes.

[Werner-Allen et al.] implemented a wireless sensor network that was deployed on the Volcano Tungurahua in central Ecuador using a Mica2 sensor mote platform and three infrasonic microphone nodes. These nodes transmitted data to an aggregation node, which relayed the data over a 9km wireless link to the base station. Time synchronization for the infrasonic sensors was done using a GPS receiver which receives a GPS time signal and relays the data to the infrasound and aggregator nodes. Over a period of time, the temperature and battery voltage may change and this may affect the sampling rate of the individual nodes and the precision of the time recorded from the GPS time stamp message. These uncertainties were addressed by applying a linear regression to the logged data stream, giving an estimation of the node outputs. In addition, the sensor nodes were required to be protected against harsh environments such as rain and long exposure to sunlight by using waterproof pelican cases and 1/4-wave whip antennas.

Their first deployment on the volcano included a small network where nodes could send continuous signals to each other, however this was not feasible for a larger network deployed over longer periods of time. To solve the issue with bandwidth and energy consumption, the team implemented a distributed event detector which only transmits well-correlated signals to the base station. A local event detection algorithm is used to trigger data collection for uncorrelated signals.

- Distributed Event Detector

To measure the correlated signal, the distributed detector uses a decentralized voting system among a group of nodes. Each node samples data at a continuous rate of 102.4Hz and buffers a window of these data while running a local event detection algorithm. If an event is triggered it uses a local radio broadcast to send a vote to other nodes and a global flood to initiate global data collection from all the nodes. Radio contention is reduced using a token-based scheme for scheduling transmission, where each nodes transmits their buffer one after the other.

- Local Event Detector

The team implemented two local event detectors; a threshold-based detector and an exponentially-weighted moving average (EWMA)-based detector. The former is triggered when the signal rises above an upper threshold or falls below a lower threshold. However due to spurious signals such as wind noise, the detector may be susceptible to false positives. On the other hand, the EWMA detector calculates two moving averages with different gain parameters and is triggered if the ratio of the two averages and the new sample exceeds some threshold $T$. This method is less affected by node sensitivity and any duplicate triggers over a window of 100 samples are suppressed.

Data Collection and Performance  

During the deployment, the team managed to log 54 hours of continuous data which includes seismo-acoustic signals from several hundred events. The raw data was used to analyse the system’s performance and many challenges were discovered in the early observations. They discovered that on average only 61% of data was retrieved from the network and on several occasions the modems which transmitted data back to the station would experience short
drop-outs, causing all data aggregated from the nodes to be lost. In addition, duplicate packets were also recorded as a likely result of redundant retransmission and lost acknowledgement. Both lost and duplicate data had to be accounted for before being stored in the timebase.

Conclusion and Improvements In general, the event-triggered model was successful in detecting eruptions and other volcanic events and was able to verify the working of the local and global event detectors by examining the downloaded data. However, since the experiment was deployed over a limited area space of the volcano, further work is to be done to instrument volcanoes at a larger scale. This includes, most importantly, the management of energy and bandwidth usage of the sensors and increasing its computational power to deviate from continuous data collection to enabling the collection of well-correlated signals.

GENESI Another class of application for wireless sensor networks is the structural health monitoring (SHM) of critical infrastructure such as bridges, tunnels and dams, to estimate the state of structural health or detecting the changes in structure that may affect its performance. The conventional method uses PCs wired to piezoelectrical accelerometers and has drawbacks such as (i) use of long wires over the structure, (ii) high cost of equipment, and (iii) expensive to install and maintain. Since SHM requires high data rate, large data size, and a relatively high duty cycle, it would be more efficient to use a WSN which would provide the same functionality as conventional methods but at a much lower price and permits denser monitoring.

Benini et al. in [19] developed GENESI (Green sEnsor NEtworks for Structural monItoring), which aims to efficiently harvest and generate energy from multiple sources with radio-triggering capabilities and using new algorithms and protocols to dynamically allocate the sensing and communication of tasks to the sensors. GENESI uses a wide range of electronics such as power-scalable, high efficiency DC-DC converters and low-drop regulators as well as different harvesting and power distribution strategies to maximise the provision and collection of energy under a large range of environmental conditions. The system also uses an effective power management strategy to complement the energy harvester mechanism which improves the battery lifetime up to the theoretical limit.

Besides efficient energy harvesting methods, GENESI also focuses on reducing energy consumption for communication and uses a selective activation scheme to dynamically schedule the activation of a set of passive and active sensors. A task allocation algorithm is used alongside the scheme to decide which sensors are to be activated and considers factors like node residual energy and field coverage. Choosing the best assignment of the available sensors to task proved a great challenge for the team since this problem involving nodes with radio-triggering and harvesting capabilities had never been studied before.

Air Pollution Continuing with the theme of deploying sensor networks in environments that more directly involve humans, air pollution has been a growing concern in congested urban areas as a result of industrialisation and heavy transportation. These concerns include poor air quality and visibility and long term damage to human health [11, 67]. Traditional methods of air quality monitoring involving quality control stations are expensive and provide low resolution sensing since monitoring stations are less densely deployed. Wireless sensor networks can be used as an urban monitoring system as they have the advantages of being small, easy to set up and inexpensive with real-time monitoring capabilities. Sensors which have the ability to measure a wide range of meteorological data such as rainfall, wind speed, temperature, humidity and concentration of pollutants can be deployed in areas of high population density of vehicles and industrial areas. Collected data is transmitted back to the base station via a global system for mobile communication.

Fotue et al. in [47] proposed a Wireless Mesh Network to establish Internet connectivity and simultaneously measure air pollution in Sub-Saharan African cities. These cities experience the most problems due to the increase of industry, domestic waste and fuel combustion and the appearance of many chemical industries that have been established in the central urban areas. However, with poor infrastructure and telecommunication links in Africa, it would be difficult to deploy a sensor network that will measure air pollution since some pollution areas are out of range of the communication service. In addition, some polluted areas lack the security measures or have indus-
trial restrictions to deploy fixed sensors so they had to use mobile sensors which were attached on vehicles.

The following are a number of challenges Fotue et al. faced when developing their solution.

- **Reducing communication interference**
  
  High-gain antennae were used to reduce the probability of interference from non-city radio frequency emitters such as WiFi and access points. However, interference may also be caused from inter-device communication and was reduced by using multiple orthogonal channels. Each set of devices would use a unique channel to communicate with a different set of devices, for example the mesh network would use a channel $C_s$ with the sensors and channel $C_g$ with the gateway.

- **Limiting number of request messages each sensor can receive from the data server.**
  
  The network uses an Ad-hoc On-Demand Distance Vector (AODV) routing protocol for efficient forwarding of data packets to the data server and makes use of a series of messages to communicate through the network and to discover and maintain data routes. A request message is propagated to the network when the data server requests for a specific data item. All the sensors receive the message and check whether it is the intended destination or not. If not, it stores a copy of the message in its buffer containing previously broadcasted request messages and rebroadcasts it back to its neighbours. Once the destination node receives the message it sends back a reply message using the reverse path of the data server.

  To prevent the sensors from overloading with request messages, each message is set with a Time-To-Live (TTL) value which signals the message to be discarded after a certain period. This value is dependant on the size of the network where a larger network will have higher TTL value to ensure that the request messages reaches the desired destination.

- **Energy Usage**
  
  The wireless mesh network is energy efficient since the sensors are energy constrained. This uses an enhanced energy-conserving routing protocol which employs a routing algorithm that aims to distribute data among maximally disjoint paths towards the sink so that the lifetime of the WSN service can be prolonged. It also allows the sensors to participate in carrying the global traffic over the network.

- **Data transmission periods**
  
  To avoid high energy consumption, data from the sensors to the data server were only transmitted at specific times of the day (not in real-time). This allowed for energy and bandwidth optimization and thus increased the network’s lifetime in the long run.

- **Optimized placements of mesh router**
  
  Since the network includes mobile sensors, there are chances that it can lose connectivity with the fixed sensors. A challenge that the team faced was placing the mesh routers in such a way that they were always connected to the mobile and fixed sensors in order to receive the sensed data. This is crucial for network reliability and quality of service.

### 1.3.5 Tools and Platforms

To develop the aforementioned wireless sensor network application there are many tools that allow developers to actually develop, test and simulate the behaviour of their software before it is deployed to the physical hardware and real world situations. These pieces of software are vital for developing reliable, low-energy sensor network software.

**JProwler**  To begin with, JProwler [16] is a Java implementation of Prowler (which is implemented in MATLAB). JProwler is designed to simulate MICA motes and provides two radio models: Gaussian and Rayleigh, one of which is for use with stationary nodes and the other for use with mobile nodes. The main use case of JProwler is to simulate large networks very quickly. The
implementation of algorithms is written in Java, using the simulator as a base, so to transplant
the algorithms to run on actual hardware will require the rewriting of the program for a different
platform. The benefits of it are: that it is very fast and it is implemented using Java this enables
quick prototyping of algorithms, easy testing and fast analysis.

**NS2** A step up from JProwler is NS2, a “discrete event simulator targeted at networking research” [82] that “provides substantial support for simulation of TCP, routing, and multicast protocols
over wired and wireless” networks. Due to its initial focus on wired networks the support for
wireless networks can be below expectations. NS2 is written in C++ and uses OTcl to manage the
simulation of networks. NS2 has a very large number of features, for example many of the routing
protocols are built in and it comes with support for mobility models such as random waypoint.
Due to the number of features NS2 can be very useful to people who wish to utilise it to its full
potential, but it can be difficult to newcomers who may be overwhelmed.

**TinyOS** Moving from two simulators, we now look at two sensor network operating systems
that can be simulated. The first is TinyOS, an event driven wireless sensor network operating
system [75]. Programs are written for TinyOS in nesC, a dialect of C [51]. While TinyOS is often
described as an operating system it is better described as a framework for developing programs
for embedded systems (specifically sensor networks). Developing for TinyOS can be easier than
other implementations because the nesC language models the tasks that a developer would be pro-
gramming a sensor node to be doing better than developing programs directly in C. To simulate
applications developed for TinyOS, TOSSIM (TinyOS Simulator) [77] can be used to compile the
same nesC code that would be used to run on the hardware to an intermediate format that is
understood by the simulator. TOSSIM supports radio models where the bit error rate is config-
urable and emulates the hardware to provide an accurate simulation of what may occur in a real
life situation. Finally the simulator supports visualising the network, so developers can obtain an
intuitive understanding of what is happening.

**Contiki** Contiki is an open source event-driven operating system for wireless sensor nodes, for
which programs are written in plain C utilizing purpose-built libraries that give access to Contiki’s
core features such as the Rime communication stack and Proto-threads[3]. Contiki is designed as
an open source project to connect low-power battery operated devices to the internet of things[13],
however it can also be used as a traditional development platform as well. Cooja is a simulator
for the Contiki OS platform, in which each simulated mote is an actual compiled and executing
Contiki system which is controlled and analysed by the simulator. The compiler is built in Java
and controls a compiled Contiki system in several ways, it dispatches events to the motes such
as message received and sent as well as analysing the Contiki system memory. Cooja contains a
network visualiser, node communication log and network communication timing graph to provide
developers with a full debugging and development suite.

### 1.3.6 Communication

Within sensor networks TCP/IP is traditionally viewed as unsuitable; Estrin et al. suggested that
sensor networks have such different requirements to traditional networks that the overall structure
of these networks needs to be reconsidered [44]. Some of the problems associated with
IP in sensor networks included [44]:

- “The sheer numbers of these devices, and their unattended deployment, will preclude reliance
  on a broadcast communication or the configuration currently needed to deploy and operate
  networked devices.”

- “Unlike traditional networks, a sensor node may not need an identity (e.g., an address).”

- “Traditional networks are designed to accommodate a wide range of applications.”

Hui and Culler suggested that these problems are largely due to IPv4 and not the newer IPv6,
stating “IPv6 is better suited to the needs of WSNs than IPv4 in every dimension” [61], further
showing that an IPv6-based network architecture could be used in sensor networks, and provides
a strong foundation from which to base further work \[40\]. Simplified versions have been suggested as well as optimisations performed with headers \[33, 35\], but they still had issues with regards to their being based on IPv4. The implementation by \[Estrin et al.\] of an IPv6-based architecture still had some problems, such as large headers and the reliance on routing tables, however the implementation had many benefits when concerning applications involving neighbour discovery and routing, but provided little improvements towards applications not involving identities (such as flooding).

\[Dunkels et al.\] developed a common architecture, which allowed for much more code reuse and interoperability between protocols, which involved the Rime communication stack. The need for this new layer was brought about by applications needing to run over multiple protocols, leaving them unable to rely on specific communication mechanisms (such as retransmissions and acknowledgements) \[38\]. Previous work \[11, 91\] did not effectively solve the problem of protocols running on top of the lower layers, but did provide a basis for the Rime stack used in Contiki \[38\].

The Rime communication stack presents a solution to the cross-layer information-sharing problem of a layered communication stack by separating the protocol logic from the details of the packet headers. Packet headers need to be small, yet adaptable enough to encompass any type of communications. They solved this by using the notion of Packet Attributes. Modules would be used to transform data and the attributes into the corresponding packets (with headers and payloads). This provides the developer access to lower level information without violating layering principles, yet with similar execution performance. This also allows interoperability with all protocols, abstracting the networking layers from the main application layer. For the application layer, Rime provides many communication primitives (such as Single-Hop Broadcasting) \[38\]. These primitives were selected based on analysis on the common use cases in WSNs.

![Diagram of Rime communication primitives](image-url)

**Figure 1:** The communication primitives in the Rime network stack \[38\]
<table>
<thead>
<tr>
<th>Name</th>
<th>Header</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anonymous Broadcast</td>
<td>abc.h</td>
<td><a href="http://contiki.sf.net/docs/2.6/a01717.html">http://contiki.sf.net/docs/2.6/a01717.html</a></td>
</tr>
<tr>
<td>Broadcast</td>
<td>broadcast.h</td>
<td><a href="http://contiki.sf.net/docs/2.6/a01720.html">http://contiki.sf.net/docs/2.6/a01720.html</a></td>
</tr>
<tr>
<td>Stubborn Broadcast</td>
<td>stbroadcast.h</td>
<td><a href="http://contiki.sf.net/docs/2.6/a01739.html">http://contiki.sf.net/docs/2.6/a01739.html</a></td>
</tr>
<tr>
<td>Anonymous Polite Broadcast</td>
<td>polite.h</td>
<td><a href="http://contiki.sf.net/docs/2.6/a01730.html">http://contiki.sf.net/docs/2.6/a01730.html</a></td>
</tr>
<tr>
<td>Polite Broadcast</td>
<td>ipolite.h</td>
<td><a href="http://contiki.sf.net/docs/2.6/a01724.html">http://contiki.sf.net/docs/2.6/a01724.html</a></td>
</tr>
<tr>
<td>Unicast</td>
<td>unicast.h</td>
<td><a href="http://contiki.sf.net/docs/2.6/a01738.html">http://contiki.sf.net/docs/2.6/a01738.html</a></td>
</tr>
<tr>
<td>Stubborn Unicast</td>
<td>stunicast.h</td>
<td><a href="http://contiki.sf.net/docs/2.6/a01738.html">http://contiki.sf.net/docs/2.6/a01738.html</a></td>
</tr>
<tr>
<td>Reliable Unicast</td>
<td>runicast.h</td>
<td><a href="http://contiki.sf.net/docs/2.6/a01740.html">http://contiki.sf.net/docs/2.6/a01740.html</a></td>
</tr>
<tr>
<td>Network Flooding</td>
<td>netflood.h</td>
<td><a href="http://contiki.sf.net/docs/2.6/a01728.html">http://contiki.sf.net/docs/2.6/a01728.html</a></td>
</tr>
<tr>
<td>Multi-hop Unicast</td>
<td>multihop.h</td>
<td><a href="http://contiki.sf.net/docs/2.6/a01726.html">http://contiki.sf.net/docs/2.6/a01726.html</a></td>
</tr>
<tr>
<td>Reliable Multi-hop Unicast</td>
<td>rmh.h</td>
<td><a href="http://contiki.sf.net/docs/2.6/a01732.html">http://contiki.sf.net/docs/2.6/a01732.html</a></td>
</tr>
<tr>
<td>Reliable Unicast Bulk Transfer</td>
<td>rucb.h</td>
<td><a href="http://contiki.sf.net/docs/2.6/a00365.html">http://contiki.sf.net/docs/2.6/a00365.html</a></td>
</tr>
<tr>
<td>Mesh</td>
<td>mesh.h</td>
<td><a href="http://contiki.sf.net/docs/2.6/a01725.html">http://contiki.sf.net/docs/2.6/a01725.html</a></td>
</tr>
<tr>
<td>Collect</td>
<td>collect.h</td>
<td><a href="http://contiki.sf.net/docs/2.6/a01723.html">http://contiki.sf.net/docs/2.6/a01723.html</a></td>
</tr>
<tr>
<td>Trickle</td>
<td>trickle.h</td>
<td><a href="http://contiki.sf.net/docs/2.6/a01742.html">http://contiki.sf.net/docs/2.6/a01742.html</a></td>
</tr>
</tbody>
</table>

Table 1: Communication Primitives, headers, and documentation location

<table>
<thead>
<tr>
<th>Name</th>
<th>Reliable</th>
<th>Target</th>
<th>Sender Known</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anonymous Broadcast</td>
<td>No</td>
<td>1-hop neighbours</td>
<td>No</td>
</tr>
<tr>
<td>Broadcast</td>
<td>No</td>
<td>1-hop neighbours</td>
<td>Yes</td>
</tr>
<tr>
<td>Stubborn Broadcast</td>
<td>No</td>
<td>1-hop neighbours</td>
<td>No</td>
</tr>
<tr>
<td>Anonymous Polite Broadcast</td>
<td>No</td>
<td>1-hop neighbours</td>
<td>No</td>
</tr>
<tr>
<td>Polite Broadcast</td>
<td>No</td>
<td>1-hop neighbours</td>
<td>Yes</td>
</tr>
<tr>
<td>Unicast</td>
<td>No</td>
<td>destination</td>
<td>Yes</td>
</tr>
<tr>
<td>Stubborn Unicast</td>
<td>No</td>
<td>destination</td>
<td>Yes</td>
</tr>
<tr>
<td>Reliable Unicast</td>
<td>Yes</td>
<td>destination</td>
<td>Yes</td>
</tr>
<tr>
<td>Network Flooding</td>
<td>No</td>
<td>network</td>
<td>Yes</td>
</tr>
<tr>
<td>Multi-hop Unicast</td>
<td>No</td>
<td>destination</td>
<td>Yes</td>
</tr>
<tr>
<td>Reliable Multi-hop Unicast</td>
<td>Yes</td>
<td>destination</td>
<td>Yes</td>
</tr>
<tr>
<td>Mesh</td>
<td>No</td>
<td>destination</td>
<td>Yes</td>
</tr>
<tr>
<td>Collect</td>
<td>Yes</td>
<td>destination</td>
<td>Yes</td>
</tr>
<tr>
<td>Trickle</td>
<td>Yes</td>
<td>network</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 2: Communication Primitives behaviour

**Flooding and Gossiping** Flooding and Gossiping are arguably the most simplistic routing algorithms available to a wireless sensor network developer. In flooding, wireless sensor network nodes broadcast to all their neighbours any messages that they receive, messages only stop when either i) the packet arrives at its intended recipient or ii) the messages maximum number of node hops is reached. Gossiping is similar to flooding in many ways, the only difference being that when a node receives a message, instead of broadcasting it to every neighbour, the node randomly picks one neighbour to forward the message to. Due to the simplicity of Flooding and Gossiping there is no need for any complex routing algorithms or topology maintenance [7]. However, these two routing algorithms do suffer from several drawbacks, these are: Implosion – is when the same message is sent to the same recipient twice (Gossiping avoids this but causes delays in the propagation of messages through the network), Overlap – is when two nodes sense the same change in a local environment and both send similar messages to the same neighbour and Resource Blindness – is when the routing algorithm consumes large amounts of energy with no consideration for efficiency [7]. Sensor Protocols for Information via Negotiation (SPIN) is a family of routing algorithms designed to address the issues with traditional Flooding and Gossiping protocols[9] [7]. The family does this by using data negotiation and resource-adaptive algorithms, for example nodes running SPIN have access to the their current battery level and use this information to adapt the protocol based on how much energy is remaining. Additionally, the SPIN family of protocols uses the concept of meta-data to reduce the level of redundant data sent around the network.
Clustering  Clustering is a form of hierarchical routing that has a long history of improving the efficiency of networks, originally being used to find the optimal placement of logic gates in digital circuits \[72\]. In both these early applications and more modern variations the basic principle of clustering remains the same; to group (in a logical sense) nodes of a network according to their (typically geographic) locality \[78\]. Each node in one of these clusters has a single point of contact with the rest of the network, in a wireless sensor network this is a specific node in each cluster known as the cluster head (CH). Cluster heads communicate with each other and the base station to route packets around the network. The benefit of clustering a network is in reducing global network traffic, in a clustered network each node need only route messages to its CH rather than all the way to its destination. In some applications the CH can perform data aggregation or use a compression function \[63, 109\] on the messages it forwards to reduce traffic between clusters; the nature of these functions is always application-specific. However, the nature of clustering means that a cluster head has a significantly higher workload than any other node in its cluster, as it must listen to every message sent from the cluster and possibly forward just as many messages to the other cluster heads in the network. In domains where nodes do not have a permanent power source (such as WSNs), this leads to greater energy usage in CHs, causing their batteries to drain faster \[31\]. As a result of this, many modern clustering algorithms (e.g. \[63, 69\]) rely on or take inspiration from the Low Energy Adaptive Clustering Hierarchy (LEACH) \[58\].

LEACH and LEACH-C are two clustering algorithms proposed by Heinzelman et al. in their paper the authors showed that the standard LEACH protocol performed significantly better than static clustering in terms of energy efficiency, with LEACH-C performing better still. However, the inclusion of both rounds and TDMA in LEACH means that it requires nodes in a WSN to be synchronised, which comes with significant implications in terms of operational complexity. LEACH itself offers no solutions either towards ensuring that nodes maintain synchronisation, or towards detecting if some nodes (or more likely, an entire cluster) is not performing correctly due to slipping from the prescribed TDMA schedule. However, the authors themselves suggest that a possible improvement to negate the need for TDMA is some event-based method for applications in which nodes do not generate transmissible data constantly or at prescribed intervals.

Tree Aggregation  Tree aggregation is another hierarchical routing protocol, which differs from clustering in that it divides a network into a tree structure with the root node situated at the base station. Nodes that are furthest from the base station (in hops) become leaves of the tree and any intermediate nodes become branches. Leaves and branches forward messages towards the base station and around the network using the tree structure, this results in a reduced number of global network messages over other algorithms such as flooding. The message count can be further reduced in a similar way to clustering, by implementing an aggregation or compression function at each branch node of the network. The drawbacks of tree aggregation are clearly similar to clustering: i) branch nodes have a higher workload than leaf nodes ii) if a branch node’s battery is depleted and the node turns off it can disconnect an entire section of the network from the base station. There are several prominent tree aggregation algorithms (such as TAG, MFS and DCTC \[78\]). Although very similar, they all take differing approaches to organising the data aggregation process of the branch nodes in order to reduce the number of collisions – and therefore messages – in the network. For example, TAG uses the idea of a periodic traffic pattern which is divided into time intervals for each layer of the tree, which ensures that each branch receives as many messages from its children as possible before it forwards to the next layer.

1.4 Summary

In summation, we have explored many issues related to the field of wireless sensor networks with a specific focus on debugging them. Firstly, we examined the types of predicates our system might be required to check for, we aim to cover all 6 types of predicates, these are: strong, weak, stable, unstable, global and local. We also discussed the fault-error-failure cycle, with consideration to its impact on predicate language development. We decided upon attempting to detect error states and failures instead of faults, as these are best suited for a predicate language to detect. Next, we considered existing predicate checking software. We looked at 5 algorithms: H-Send, Sympathy, DIKON, DIDUCE and NodeMD. We compared and contrasted each algorithm and found that H-Send was the closest to our proposed system, in terms of its approach of defining a predicate. However, they were all very practically-oriented implementations as opposed to our
proposed system which traces its roots from theoretical work in global predicate detection. We also considered the various platforms that are available to developers of programs for wireless sensor networks. JProWler is a Java implemented simulation of MICA motes, although it allows for fast implementation and rapid prototyping, code would then to be rewritten to work on a real wireless sensor network. NS2 is another network simulator with a focus on wired networks, meaning that it was unsuitable for our purposes. TinyOS is a complete operating system for wireless sensor nodes, in which programs are written in nesC, a dialect of C. TOSSIM is a simulator designed to run TinyOS programs and the whole system is event-driven. The alternative to TinyOS is Contiki, an open source event driven operating system for wireless sensor nodes, using the standard C language (with a range of libraries). Cooja is a simulator for the Contiki OS and provides plenty of tools for developers such as a network visualiser and network communication log. We then looked at some current practical applications of wireless sensor networks, this showcased what wireless sensor technology is used for and more importantly what users of the technology might require from our system. The breadth of real-world applications was staggering, from volcano to animal habitat monitoring and everything in between. Finally, we discussed the several traditional types of routing algorithms available for distributed systems (e.g. flooding, clustering and tree aggregation). We then looked at several algorithms designed specifically for wireless sensor networks (e.g. SPIN, LEACH and TAG), we discussed the improvements made and what benefits they brought to the traditional algorithms.
2 Problem Statement and Formalisation

2.1 Plan

As stated in the introduction, we are investigating developing tools to aid in debugging the distributed programs running on wireless sensor networks. We aim to do this by implementing libraries that use different techniques to check predicates, with a focus on correctly evaluating these predicates and investigating if there are places in the network where evaluation is cheaper, in terms of energy. We also intend to visualise some of the state of the network, as part of a tool to inform system users what state the network is in.

Due to the research-led nature of this project we were unsure of how we were going to implement the desired applications as some approaches may have been infeasible. Therefore we undertook a long research period as has been reflected by our research into related work. We channelled this research into what exactly we were going to produce and experiment with. In order to test our solutions, we aimed to run simulations of them to examine how well they perform. This is obviously conditional on the libraries they use working, which will be thoroughly tested to allow the higher level predicate evaluation applications to work correctly. The remainder of our project management can be found towards the end of this report.

2.2 Model

To assist with the development of our project it is first necessary to state how we will be modelling the problem. To begin with we model a wireless sensor network as a graph $G = (V, E)$ where $V$ is a set of nodes. If we have two nodes $u$ and $v$ that can communicate $\{u, v\} \in E$, we assume that this communication is bi-directional. Every node in the network has a unique identifier that is typically referred to by $j$.

**Definition 1** m-Hop Neighbourhood: Given a node $n$ the $m$-hop neighbourhood of that node is the set of nodes that are within $m$ hops of node $n$. When we refer to this neighbourhood it will not contain the node $n$.

As we are dealing with simulators and physical hardware, there is no assumption of reliable links between nodes. Techniques will be described later on that increase the probability of a message reaching its target, but do not ensure it.

Every node in the network is assumed to have the same hardware and thus the same capabilities. As every node has the same capabilities, every node has similar properties such as: transmission range and initial power levels. We assume that the network is working in isolation to other networks and is not receiving other electromagnetic interference.

**Definition 2** $P_{all}$: A the set of predicates that are evaluated by all nodes in the network.

**Definition 3** $P_{single}$: A the set of predicates that are evaluated by a single node in the network.

We also distinguish between different types of predicates based on what parts of the network evaluate them. For our cases it is enough to simply consider predicates that are evaluated on a single node ($P_{single}$), or predicates that are evaluated on every node in the network ($P_{all}$).

When checking predicates we much deal with the notion of the correctness of evaluation of a predicate. As can be seen from the definition this involved the notion of time because the values used to evaluate the predicate can change over time.

**Definition 4** Correctly Evaluated Predicate: A predicate $P$ is correctly evaluated at time $\tau$ if the results of evaluating that predicate with global knowledge at time $\tau$ is the same as the result of predicate $P$.

To evaluate a predicate there is some predicate-evaluating function $E$ that takes a mapping from node identifiers to a user-defined structure of valued about that node and returns a boolean. By executing this function the result of a predicate is obtained. The input can be obtain through various means. The next chapter will discuss how this function is evaluated and how the required
information can be disseminated to the node that requires it to perform the evaluation.

**Definition 5** diameter: The maximum distance between any two nodes in the network.

**Definition 6** distance\((n, m)\): The length of the shortest path between the nodes \(n\) and \(m\).

### 2.3 Algorithm Descriptions

To define the algorithms developed, a custom pseudo-language is used which, for the majority, is the same as that used in [23]. Each box contains an application running on a mote or a library an application could be using (although it is possible for one of these boxes to be split up to represent the algorithm better). Within this box there are four sections parameters, variables, constants and actions. The variables section details modifiable variables that exist within the program. The constants section details network knowledge or network parameters that cannot be changed by the program, these constants will be the same for all nodes that have them. The parameters section contains variables that are passed to a library when it is initialised.

The actions section contains the methods which depending on the type will be called when a library or program starts up, when a function defined is called when a message is received or after a timer has timed out. The first type of action (receiving a message) is demonstrated in Figure 2 where ‘MessageContents...’ is the list of variables contained within the message.

```
process j
actions
  % Receiving Message
  RcvFunction:: rcv(MessageContents...) →
  % Function Contents
```

*Figure 2: Example Receive Message Algorithm*

The second type of action (timer timeout) is demonstrated in Figure 3. The function set is used to restart the timer, so it is called again once the timer times out again. If set is not called then this function will not be called again.

```
process j
variables
  period: timer init α;

constants
  % How often the message is broadcasted
  α: time;

actions
  SendFunction:: timeout(period) →
  % Function Contents
  set(rate, α);
```

*Figure 3: Example Send Message Algorithm*

The third type of action (initialisation) shown in Figure 4 is called when a library or process is started up. It is typically used to set up some initial state or fire some setup events. Once it has been run once it is never run again.

```
```

*Figure 4: Example Initialisation Algorithm*
The fourth and final type of action is a function shown in Figure 5. These are only every defined in processes intended to be libraries providing some kind of network abstraction and are typically used to expose a method that is used to send some data. Where ‘Parameters...’ is a list of arguments that can be passed to the function.

Variables and constants are given types and can optionally be initialised to a given constant. They can be initialised with ⊥ to indicate that their value has not been set. Also anything after a ‘%’ is considered a comment.

To send messages we use bcast or send and to receive a message we use recv, of which there are several varieties required as shown in Figure 6. When using the user defined syntax the user part will be replaced with the name of the library. The user defined network functions also have a user.deliver function, when this is called it will simply pass the message to the user.receive
function of the user of the library.

To help implement our algorithms we also use common programming libraries such as a map and queue. The following shows how the syntax is defined and used.

```
process j
variables
  m: map init ∅;

actions
  % Function definition
  func:: function(Parameters...) →
  % Assign a value to some key
  m[1] := 2;

  % Update a value assigned to some key
  m[1] := 4;

  % Get the set of keys in the map
  keys(m);

  % Get the set of value in the map
  values(m);

  % Remove a key and its associated value from the map
  remove(m, 2);
```

Figure 7: Map functions

```
process j
variables
  q: queue init ∅;

actions
  % Function definition
  func:: function(Parameters...) →
  % Append an item (add an item to the end of the queue)
  append(q, 1);

  % Prepend an item (add an item to the front of the queue)
  prepend(q, 2);

  % Look at the first item of the queue
  peek(q);

  % Remove the first item of the queue
  pop(q);
```

Figure 8: Queue functions
3 Predicates

When considering how to evaluate predicates there are many points that need to be considered. Depending on what the intention is, certain implementations of predicate evaluation are going to be better than other implementations. Also, when developing with considerations of the predicate evaluation such as accuracy, the considerations of sensor networks will need to be taken into account (such as the minimisation of energy usage). This section will first detail the types of predicates that we identify that we wish to be able to detect, then it will cover how we can evaluate such predicates and finally will discuss several implementations of transmitting the information to the required part of the network.

3.1 Types of Predicates

In the introduction several different classes of predicates were discussed, of which the most important one to us is the locality of the predicate. A majority of the work focuses on global predicates \[49, 111, 50\] which while useful in general for distributed systems, is perhaps not as helpful for wireless sensor networks. To begin with many of the problems that a sensor network may encounter are local problems. By local we mean that a node in the network only has access to a specific subset of the networks information, in our case we focus on the surrounding neighbours of the node evaluating the predicate. When forcing a local problem to be evaluated globally it means that investigating evaluating that predicate locally is eliminated. This is problematic as there may be energy savings when evaluating a predicate locally. So the first major decision is that instead of focusing on global predicates, local predicates are instead the focus - in order to investigate any potential energy savings.

Definition 7 Global Predicate: A predicate \(P\) that operates on some global state \(S\) where \(S\) is a mapping from a node id to some data on that node.

Definition 8 Local Predicate: A predicate \(P\) evaluated on the node \(j\), where the state \(N(j, n) \subseteq S\) available contains information on some \(n\)-hop neighbourhood of \(j\). Where \(N(j, n)\) is a function that returns the state of all nodes within \(n\) hops of \(j\).

The second decision to decide on is the stability of the predicate being detected. As discussed in the introduction there is a choice between stable predicates that remain true and unstable predicates whose truth value can alternate. This decision is important because it will affect the algorithm structure that works for checking stable predicates. An example of this is taking snapshots of global state and checking the predicate against what is expected. However, for unstable predicates detailed recording of the traces of the system is required \[17\]. Due to the limited resources of sensor nodes we focus on the simpler problem of stable predicates. This is mainly because complex programs tend to require a greater number of instructions to do more things and the size of the firmware on the motes is limited \[5\].

Definition 9 Application Predicate: A predicate that is evaluated over the state an application is in. This can involve extracting and examining sensor data or program variables.

Definition 10 Network Predicate: A predicate that is evaluated over the network interactions. Examples include detecting collisions where there should have been none, or checking that there are no loops in multi-hop communications.

So far traditional predicate properties have been focused on. However, we need to introduce two new classes of predicates. In previous work the authors dealt in the abstract notion of program traces, these traces are simply events that lead to some eventual state. When developing software for sensor networks we feared that (i) the hooks to detect the traces and (ii) recording them would be too demanding on the limited resources available. By considering what happens in the application separately from the network, the impacts may be decreased. Network predicates would involve monitoring the low level MAC layer for network events and also including extra data to packets sent from the node. Application predicates could imply be evaluated instantaneously on the data available. Due to the simplicity, but the good results that application predicates may provide we focus solely on them.
Much of the previous work exists in ideal worlds were assumptions such as “no messages are lost” are made. Unfortunately this is not the case in the real world, while the MAC layer and the application layer can do much to mitigate packet loss without energy expensive protocols such as 802.11 which use a MAC protocol based on CSMA/CA it is not possible to ensure a certain level of reliability. Therefore it must be realised that every time a predicate is evaluated it will have a certain level of accuracy. This is because some data may have been lost on the way to its destination or outdated information was used. We believe the accuracy is a very important angle to predicate evaluation as an accurate predicate evaluation that is not received often will inform the system’s user more than a predicate evaluation result that is received very frequently but is also very inaccurate.

In summary we focus on evaluating predicates that are stable and use local information. These predicates also deal exclusively with information about the application running on the mote and not the communication it is involved with. Also there should be a focus that predicates are evaluated as accurately as possible with a minimum amount of energy. The next two sections will detail how predicates are evaluated in our implementation and then how the required information is disseminated to the correct motes.

3.2 Predicate Evaluation

When deciding how we would evaluate predicates, once data was received, there were two possible solutions that were considered. The first was to simply have the system developer hardcode the checking into the code and provide a library to handle responses, the second was to implement a virtual machine that would run a script that checks the predicate. Using hard checks written in C would have provided a more efficient way of checking the predicate and would be similar to the approach used by HSend which parsed the C source code and generated C code for the predicates specified in a special comment block. A downfall of hardcoding the predicate evaluation is that it could lead to a large inflation of the firmware and if the system developers wanted to change, add or remove a predicate then they would need to update the entire firmware image across the network.

With a scripting language, instead of sending a large firmware binary across the network a small program of high level opcodes could be sent instead. The difference could be huge, for example the maximum firmware size of the CM5000 is 48KB, and it is conceivable that a program script utilising high level instructions could instead fit into the size of a single packet (128 bytes). Therefore to aid in the flexibility of our solution we implemented a simple virtual machine and language that was executed to evaluate the predicate.

Initially we looked for an already developed scripting language that we could use on the motes. Having used high level scripting languages such as Python and Lua we first looked into using them. Unfortunately even though Lua offered an embedded alternative called eLua the firmware size and the RAM usage would have been unacceptably high for the hardware at our disposal. We then started looking at a language called SScript by Dunkels who was also the author of Contiki, the wireless sensor network OS we were using. Unfortunately (in part due to the name) we were unable to find the implementation of it. Finally we looked at Antelope which provided a way to query nodes using SQL like a database. Unfortunately we were unsure as to how much control we would have on being able to optimise energy usage via different ways of gathering data. So we concluded that we would need to create our own language, parser for it, assembler and virtual machine to execute the produced code.

The predicate evaluation language that we designed needed to be fairly simple to make it (i) easy to implement and (ii) easy to execute. To do this we set out a number of desired features and then explicitly decided on features that would not be included because of their irrelevance as can be seen in Table 3. The most important feature was that we wanted to be able to test a sub-predicate over a set of neighbouring node’s data, this meant that we would need a way to represent a set of data and iterate over it. As the program is always intended to return a boolean result, there was no need to implement comprehensive iteration of the form for (INIT; CONTINUE; NEXT) we instead could just implement for-all and exists operators. To simplify further the language was designed to be functional rather than imperative as we had no need for our predicate checker to

1 http://contiki.sourceforge.net/docs/2.6/a00302.html
directly cause state changes in the application running directly on the hardware.

We managed to implement the majority of the features we desired, however there were some that were difficult to implement or had to be implemented differently. The first is feature 4 (set operations), which simply would have been too difficult to implement and would have greatly increased the code size. The next is feature 6 (targeting multiple nodes), this was not implemented because it would have required a variable amount of space in the message that disseminated the predicate throughout the network. There is also a simple work around, which is to create a new predicate using the same code for each of the nodes that wish to be targeted.

For some of the features we desired we ended up implementing them differently than expected. Initially we expected to allow users to define functions and structures that could be operated on (like in C). However, what we ended up with was slightly different. We ended up with a primitive called Neighbours(n) which allowed the predicate to ask for its n-hop neighbourhood, it may also be expressed with two parameters as Neighbours(j, n) where j refers to the node neighbour’s are being requested of, for the single argument function the node should be implied from the context it is called from. We also ended up with a single structure which the user defined themselves in C and provided functions to access data in that structure. These functions are then exposed to the predicate language. Finally information on a node is only exposed to the predicate if the user provides some way to store this data in the nodes data structure and provides functions to access it. These functions and the user data structure will be explained further on.

\[
\text{Neighbours}(j, n) = \{x | x \in V \land \text{distance}(j, x) \leq n \land x \neq j\}
\] 

Table 3: Predicate Language Features

<table>
<thead>
<tr>
<th>#</th>
<th>Feature</th>
<th>Desired</th>
<th>Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Integer ((+ - \times / \neq &lt; \leq &gt; \geq \text{casts}))</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Floating Point ((+ - \times / \neq &lt; \leq &gt; \geq \text{casts}))</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Set iteration support ((\forall 3))</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>Set operation support ((\cup \cap \setminus {}))</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>Ability to specify predicate target(s) (Single or Every node)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>Ability to specify predicate target(s) (Multiple nodes)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>Ability to get the current node information ((\textit{this}))</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>Limited function support (eg, Neighbourhood((N, H)))</td>
<td>Yes</td>
<td>Partial</td>
</tr>
<tr>
<td>9</td>
<td>Limited structure support (eg, (N.Temperature))</td>
<td>Yes</td>
<td>Partial</td>
</tr>
<tr>
<td>10</td>
<td>Ability to get sensor information on a node</td>
<td>Yes</td>
<td>Partial</td>
</tr>
<tr>
<td>11</td>
<td>Ability to get node information (Distance to Sink, . . . )</td>
<td>Yes</td>
<td>Partial</td>
</tr>
<tr>
<td>12</td>
<td>Strings</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>13</td>
<td>IO</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>14</td>
<td>Data structures (Lists, Arrays, Dictionaries, . . . )</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>15</td>
<td>Library features (Custom libraries or built-in libraries)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>16</td>
<td>Interactive mode (terminals)</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The major problem with developing our own language was that we needed a way to convert that into a compact form that could be easily executed by a virtual machine. As can be seen by the complexities of the GCC and LLVM/Clang projects, compilation of abstract code to optimal executable code is a difficult problem. So instead of trying to develop a highly optimising compiler, our target was to develop a compiler that produced good code that was capable of being hand optimised by developers before being deployed to the network.

To reach this goal, we followed good development principles and broke down the creation of this executable code into to separate phases: the compiler and the assembler. The compiler parsed the predicate language and compiled it down to a human readable intermediate representation (IR) that could be understood and modified by humans. The assembler then converted this into the raw bytecode that would be sent to the motes for evaluation. Having a human readable IR makes hand-optimisation very simple, and easy to support directly within the visualisation tool. This separation leads to the workflow of developing and deploying a predicate as shown in \[\text{Figure 9}\].

23
### 3.2.1 Domain Specific Language

The following is the definition of how to parse our predicate language. We implemented it in Java using JavaCC, using several resources[^2] to help in the development. In the language there are four main components. The first is the target of the predicate specified within square brackets, the target can be a node address (such as 1.0) or ‘all’ to target all nodes in the network. The second is the function definitions. When users initialise our library on the motes, they need to provide a function that provides the current information on the node and a mapping of function ids to the functions used to access this data. The function definition statement in the predicate language allows a human understandable name to be assigned to a function id and it also tells the compiler what id function should be called and the type that that function returns. The third is the ‘using’ statement that allows a set of neighbouring information to be assigned to a variable name. This is the only way to request node information not about the current node that is evaluating the predicate. The fourth and final part if the predicate itself, which includes expected logic operators that operate over sets, integers, floating point numbers and booleans. The logical operators that make up the predicate are described in [Table 3](#).

<table>
<thead>
<tr>
<th>Name</th>
<th>Logic</th>
<th>Symbol</th>
<th>Input Type(s)</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>For All</td>
<td>∀</td>
<td>@</td>
<td>Set of user defined data</td>
<td>Boolean</td>
</tr>
<tr>
<td>Exists</td>
<td>∃</td>
<td>#</td>
<td>Set of user defined data</td>
<td>Boolean</td>
</tr>
<tr>
<td>And</td>
<td>∧</td>
<td>&amp;</td>
<td>Booleans</td>
<td>Boolean</td>
</tr>
<tr>
<td>Or</td>
<td>∨</td>
<td></td>
<td></td>
<td>Booleans</td>
</tr>
<tr>
<td>Xor</td>
<td>⊕</td>
<td>ˆ</td>
<td>Booleans</td>
<td>Boolean</td>
</tr>
<tr>
<td>Implies</td>
<td>⇒</td>
<td>=&gt;</td>
<td>Booleans</td>
<td>Boolean</td>
</tr>
<tr>
<td>Equivalence</td>
<td>⇔</td>
<td>&lt;=&gt;</td>
<td>Booleans</td>
<td>Boolean</td>
</tr>
<tr>
<td>Not</td>
<td>¬</td>
<td>!</td>
<td>Boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>Equality</td>
<td>=</td>
<td>==</td>
<td>Integers or Floats</td>
<td>Boolean</td>
</tr>
<tr>
<td>Inequality</td>
<td>≠</td>
<td>!=</td>
<td>Integers or Floats</td>
<td>Boolean</td>
</tr>
<tr>
<td>Less Than</td>
<td>&lt;</td>
<td>&lt;</td>
<td>Integers or Floats</td>
<td>Boolean</td>
</tr>
<tr>
<td>Less Than or Equal To</td>
<td>≤</td>
<td>&lt;=</td>
<td>Integers or Floats</td>
<td>Boolean</td>
</tr>
<tr>
<td>Greater Than</td>
<td>&gt;</td>
<td>&gt;</td>
<td>Integers or Floats</td>
<td>Boolean</td>
</tr>
<tr>
<td>Greater Than or Equal To</td>
<td>≥</td>
<td>&gt;=</td>
<td>Integers or Floats</td>
<td>Boolean</td>
</tr>
<tr>
<td>Addition</td>
<td>+</td>
<td>+</td>
<td>Integers or Floats</td>
<td>Integer or Float</td>
</tr>
<tr>
<td>Subtraction</td>
<td>−</td>
<td>-</td>
<td>Integers or Floats</td>
<td>Integer or Float</td>
</tr>
<tr>
<td>Multiplication</td>
<td>×</td>
<td>*</td>
<td>Integers or Floats</td>
<td>Integer or Float</td>
</tr>
<tr>
<td>Division</td>
<td>÷</td>
<td>/</td>
<td>Integers or Floats</td>
<td>Integer or Float</td>
</tr>
</tbody>
</table>

[^2]: [JavaCC tutorial](http://cs.lmu.edu/~ray/notes/javacc/)
Figure 10: Language definition

When we were implementing the parsing for the predicate language one of the major issues encountered was that parsing infix operators (that is the operator is between two operands) was much harder to accomplish than prefix or postfix operators, was used as a guide to implement infix parsing.

Another important issue to mention is the distinction between syntax and semantics. The language definition specifies the syntax, but it fails to capture some of the semantic subtleties of valid programs. One example is that even if a variable name or a function name are syntactically valid strings, compilation should fail if they have not been defined by a statement prior to their usage. However, JavaCC provides a very capable framework for validating semantics during construction of the program’s IR.

An important part of this language is that certain parts of the program need to be written to reflect what is implemented in the actual firmware of the motes. For example the virtual machine must be provided with a function that produces the node’s current data that can be accessed through this. Calling this just puts a block of memory on the evaluation stack, the user must also tell the compiler what functions are available, how it should access them and what type of data they return. In this way the user defined data can have data extracted from it in a form that the virtual machine knows how to deal with.
### 3.2.2 Example Predicates

**Figure 11:** Check that no two neighbours have the same slot (2-hop information)

```plaintext
function 1 as slot returning int in
using Neighbours(2) as twohopn in
  @((x : twohopn ~ slot(x) != slot(this))

∀n ∈ Nodes: ∀n' ∈ Neighbours(n, 2):
  slot(n) ≠ slot(n')
```

**Figure 12:** Check that no two neighbours have the same slot (1-hop information)

```plaintext
function 0 as addr returning int in
function 1 as slot returning int in
using Neighbours(1) as onehopn in
  @((a : onehopn ~ @((b : onehopn ~ addr(a) != addr(b) => slot(a) != slot(b)) & slot(a) != slot(this))

∀n ∈ Nodes: ∀n' ∈ Neighbours(n, 1) ∪ {n}:
  ∀n'' ∈ Neighbours(n, 1) ∪ {n}:
    addr(n') ≠ addr(n'')
    ⇒ slot(n') ≠ slot(n'')
```

The predicates specified in Figure 11 and Figure 12 aim to check that slot allocation is correct and will not lead to collisions. As these are applications predicates they check that there are no clashes with the slots assigned. If we were also checking network related state, these predicates would also need to be monitoring the MAC layer to make sure that no collisions did in fact occur. Predicates similar to these appear in the problem statement of [10], which leads us to believe that an implementation of a TDMA algorithm may desire some checking of these predicates to test correctness of such an algorithm.

**Figure 13:** Check that the relative humidity of node 1.0 is less than or equal to 40%

```plaintext
function 3 as humidity returning float in
  humidity(this) <= 40.0

∀n ∈ Nodes: addr(n) = 1.0 ⇒ humidity(n) ≤ 40
```

**Figure 14:** Check that the average neighbour temperature is with 10 degrees of ours

```plaintext
function 2 as temperature returning float in
using Neighbours(2) as twohopn in
  abs(temperature(this) - mean(temperature, twohopn)) <= 10

∀n ∈ Nodes:
  \[
  \left| \frac{\sum_{n' \in \text{Neighbours}(n, 2)} \text{temperature}(n')}{|\text{Neighbours}(n, 2)|} \right| \leq 10
  \]
```

The two predicates in Figure 13 and Figure 14 capture what system users may be trying to detect. Figure 13 shows a predicate that fails when the humidity sensed by a predicate falls low and
Figure 14 detects when there is a large difference in the temperature of one node and the average of its neighbours. Both of these predicates could be used when monitoring the environment in a forest to see how likely and where forest fires may be starting.

function 0 as addr returning int in
using Neighbours(1) as onehopn in
using Neighbours(2) as twohopn in
∀n ∈ Nodes:
{addr(n′)|n′ ∈ Neighbours(n, 1)} ⊆ {addr(n′)|n′ ∈ Neighbours(n, 2)}

∀n ∈ Nodes·{addr(n′)|n′ ∈ Neighbours(n, 1)} ⊆ {addr(n′)|n′ ∈ Neighbours(n, 2)}

Figure 15: Check 1-hop neighbourhood is in 2-hop neighbourhood

This predicate is perhaps not as interesting as other predicates but it has a valid use in testing certain aspects of the information dissemination aspect of our predicate checking algorithm. It is important that we can test our mechanisms to evaluate predicates, because if we cannot rely on the correct evaluation of predicates, then we cannot rely on the results. Predicates like checking the 1-hop neighbourhood is in the 2-hop neighbourhood allows us to check that we are gathering data in the right way.

function 5 as sink-distance returning int in
using Neighbours(1) as onehopn in
sink-distance(this) == min(sink-distance, onehopn) + 1

Figure 16: We are one hop further from the base station than the closest of our neighbours. May be used to check parent in aggregation tree is actually closer to the sink.

function 0 as addr returning int in
function 4 as ch returning int in
function 6 as H returning int in
using Neighbours(H(this)) as neighbours in
∀n ∈ Nodes:
∃n′ ∈ Neighbours(n, H(n))·ch(n) = addr(n′)

Figure 17: Check cluster head is in H-hop neighbourhood

The predicate shown in Figure 16 and Figure 17 are examples of ways that we can use application predicates to check that the network communication aspect of wireless sensor network are working correctly. The first checks that a parent assigned in tree aggregation is closer to the sink than the evaluating node. Figure 17 is intended to check that there exists a cluster head within H hops of the evaluating node. The aim would be to use this when hierarchical clustering is being used to perform routing. If either of these fail then the system administrators can simply rerun the configuration step to set up the communication paths or start looking for bugs in the code that performs this setup.

Unfortunately the predicate that checks the cluster head location would not work due to the fact that the parameter to Neighbours needs to be known at compile time. This is an implementation issue and could be corrected in the future.
3.3 Virtual Machine

The virtual machine that was developed is very simple and takes inspiration from the wren language's virtual machine. At its core our virtual machine uses a stack to evaluate a program, values can be pushed onto the stack and are then popped of the stack later for use in evaluation and the results of that evaluation are pushed back onto the stack. The last value on the stack is the results of the evaluation. In the virtual machine there are three types: 16-bit integers (I), 32-bit floats (F) and user-defined size user data (U). A boolean is represented as an integer with 0 standing for false and 1 standing for true, the virtual machine relies on the compiler to have generated correct code that will be correctly evaluated, it has limited error checking and reporting to keep the firmware size down. The virtual machine also has a limited and fix stack space that is configured to be 256 bytes. If this overflows then the predicate evaluation will be terminated with an error. Due to the limited space, predicates that require more stack space than there is to evaluate themselves will fail. In testing these issues can be detected and the stack size can be increased as desired.

3.3.1 Opcodes

The program is stored as a block of memory that is a list of opcodes followed by their arguments. The following are the opcodes implemented in the virtual machine and the semantics of executing them.

For binary operations the order the operands are used is specified as x op x. When x is 0 the value on the top of the stack is used. When x is 1 the value after the one on the top of the stack is used.

Halt
Halts execution.
The stack should always have at least one integer on it when halting.

IPush (int) / FPush (float)
Pushes an int or a float on to the stack.

Ipop / Fpop
Pops the stack by sizeof(int) or sizeof(float) bytes.

IFetch (variable id - ubyte) / FFetch (variable id - ubyte)
Fetches the value of the given variable and pushes the first sizeof(int) or sizeof(float) bytes of it on to the stack.

Istore (variable id - ubyte) / Fstore (variable id - ubyte)
Stores the first sizeof(int) or sizeof(float) bytes in the variable of the given name.

AFetch (variable id - ubyte)
Fetches the value of the array named in the parameter at the integer index at the top of the stack.

ALEN (variable id - ubyte)
Pushes an integer onto the stack containing the length of the named array.

ASum (variable id - ubyte) (function id - ubyte)
AMEAN
AMax
AMin
Over the user defined array stored in the variable whose name is the first parameter, transform it using the function whose name is the second parameter. On this transformed data calculate the array operation and push the result as a float.
onto the top of the stack.

CALL (function id - ubyte)
Calls the named function on the data on top of the stack.
Pop's the data that was given to the function off the top of
the stack (sizeof(U)) and pushes the result onto the top of the stack.

ICASTF
Pops an integer off the stack, casts it to a float
then stores it back on the stack.

FCASTI
Pops a float off the stack, casts it to a integer
then stores it back on the stack.

JMP (ubyte)
Jumps to a position in the code relative to the start
of the program.

JZ (ubyte)
Pops and reads the top of the stack as an integer, if it is 0
the program jumps to the location, otherwise evaluation
continues.

JNZ (ubyte)
Pops and reads the top of the stack as an integer, if it is not 0
the program jumps to the location, otherwise evaluation
continues.

IADD / FADD 1 op 0
ISUB / FSUB 1 op 0
IMUL / FMUL 1 op 0
IDIV1 / FDIV1 0 op 1
IDIV2 / FDIV2 1 op 0
Performs a mathematical operations on the first two sizeof(int) or sizeof(float)
values on the stack. Pops both values off the stack, stores
the result in the same type back on the stack.

IINC
IDEC
Pops the top of the stack by sizeof(int) bytes, increments or decrements
the value as an integer. Pushes the result back on the stack.

IEQ / FEQ 1 op 0
INEQ / FNEQ 1 op 0
ILT / FLT 1 op 0
ILEQ / FLEQ 1 op 0
IGT / FGT 1 op 0
IGEQ / FGEQ 1 op 0
Performs a mathematical operations on the first two sizeof(int) or sizeof(float)
values on the stack. Pops both values off the stack, stores
the result in an integer on the stack.

AND 1 op 0
OR 1 op 0
XOR 1 op 0
EQUIVALENT 1 op 0
IMPLIES 1 op 0
Performs a logical operation on the first two values on the
stack. These are not bitwise operations and the expected
formats of the values are integers. 0 is false, 1 is true.
Pops both values off the stack, stores the result in an integer back on the stack.

**NOT**

Pops an integer off the stack, performs logical not on it. Pushes the result back on the stack.

**IVAR** (variable id - ubyte) / **FVAR** (variable id - ubyte)

Creates a variable of type int or float with the id set to the given unsigned byte.

**IABS / FABS**

Pops the stack by sizeof(int) or sizeof(float) bytes performs the abs function on the integer of float value and pushes sizeof(int) or sizeof(float) back onto the stack.

**VIINC** (x - variable id)

Equivalent to the following opcodes, this is opcode is included as a program size optimisation.

- IFETCH x
- IINC
- ISTORE x

**VIDEC** (x - variable id)

Equivalent to the following opcodes, this is opcode is included as a program size optimisation.

- IFETCH x
- IDEC
- ISTORE x

**VIFAPC** (x - variable id) (y - variable id) (z - function id)

Equivalent to the following opcodes, this is opcode is included as a program size optimisation.

- IFETCH x
- AFETCH y
- CALL z

**THISC** (f - function id)

This opcode gets the sizeof(U) bytes about the nodes current state, calls the function referred to by f on it and pushes that result onto the stack. This is equivalent to:

- IPUSH 0
- AFETCH 0
- CALL f

### 3.3.2 Bytecode

One of the most important aims of the predicate language’s bytecode was to be as small as possible. The reason for this was to support sending as much program information in the limited length of a network packet. To achieve this aim it meant that we needed to make the virtual machine handle more abstract representations of its components. For example, initially the bytecode for calling a function contained a byte for the opcode of `CALL` and then a string of characters which contained the name of the function to call. At minimum this would cost 2 bytes - one for the character of the function’s name and one for the NUL character. To improve this the string was replaced with a single byte, which means that the number of functions callable by the virtual machine ends up being limited to 256. The same was true for the variable ids as well and as the variable id is contained with an unsigned byte, they too are limited to 256.

Size optimizations were also done for operations we believed would be common. This is why
there exists the ASUM, AMEAN, AMIN and AMAX operations. If these were to be coded they would at least require a loop and a certain amount of setup code that loops require. By making them single operations we can eliminate the need to produce loops in the IR and can just produce these single instructions. Also as these instructions are much simpler they make adjusting the IR easier and harder to introduce bugs. The unfortunate downside is that this means that there is more implementation in the virtual machine which leads to increased firmware sizes.

The firmware size restrictions did eventually lead to a number of features being removed. For example, during development we had support for calculating the power and modulus of variables. However, due to the size of the C libraries that were pulled in to use the pow function, it needed to be disabled. The modulus function was also disabled as it has a rare use case and the firmware space was needed for actual implementation of the application.

When developing the virtual machine what also became important was the fact that the bytecode was a string of unsigned bytes, where sequences of them could end up representing a 16 bit integer or a 32 bit float. This is important because it could mean that we would try to perform unaligned reads which the CPU of our motes could potentially get wrong\(^3\). This is due to the restrictions the CPU has on the alignment of words: “Bytes are located at even or odd addresses. Words are only located at even addresses . . . . When using word instructions, only even addresses may be used. The low byte of a word is always an even address” [108, Section 1.4.5 (p. 28)].

![Alignment Issues](http://permalink.gmane.org/gmane.os.contiki.devel/1462)

As can be seen in Figure 18 if we were to align the words as the CPU wanted the size of the bytecode would massively increase as every single byte bytecode would need to be padded by a byte to (i) make sure all bytecode entries were the same length and to ensure the word parameters are correctly aligned. To solve this issue instead of accessing the memory directly in the bytecode what can be done instead is to copy the memory out byte-by-byte into a correctly aligned block of memory and use that instead. As we copy out byte-by-byte and not by word we can access the memory correctly which allows us to store the bytecode in its more compact form desired by us.

\(^3\)http://permalink.gmane.org/gmane.os.contiki.devel/1462
3.4 Examples of Compiled Programs

<table>
<thead>
<tr>
<th>[all]</th>
</tr>
</thead>
<tbody>
<tr>
<td>function 1 as slot returning int in</td>
</tr>
<tr>
<td>using Neighbours(2) as twohopn in</td>
</tr>
<tr>
<td>Θ(x : twohopn ~ slot(x) != slot(this))</td>
</tr>
<tr>
<td>)</td>
</tr>
</tbody>
</table>

//TARGETING all
//FUNC 1 AS slot
//STORING 1 IN twohopn
IVAR 1
IPUSH 1
IPUSH 0
ISTORE 1
start1: ALEN 255
INEQ
JZ end1
//slot(a[*1])
VIFAFC 1 255 1
//slot(this)
THISC 1
INEQ
AND
VIINC 1
JMP start1
endi: HALT
//VD 2 = 255

| [all] | function 0 as addr returning int in |
|-------|
| function 1 as slot returning int in |
| using Neighbours(1) as onehopn in |
| Θ(a : onehopn ~ addr(a) != addr(b) => slot(a) != slot(b)) & slot(a) != slot(this)) |
| ) |

//TARGETING all
//FUNC 0 AS addr
//FUNC 1 AS slot
//STORING 1 IN onehopn
IVAR 1
IPUSH 1
IPUSH 0
ISTORE 1
start1: ALEN 255
INEQ
JZ end1
IVAR 2
IPUSH 1
IPUSH 0
ISTORE 2
start2: ALEN 255
INEQ
JZ end2
//addr(a[*1])
VIFAFC 1 255 0
//addr(b[*2])
VIFAFC 2 255 0
INEQ
//slot(a[*1])
VIFAFC 1 255 1
//slot(b[*2])
VIFAFC 2 255 1
INEQ
IMPLIES
AND
VIINC 2
JMP start2
endi: HALT
//VD 1 = 255

Figure 20: Compiled check for slot collisions. Using 2-hop information of Figure 11 on the left and using 1-hop information of Figure 12 on the right.

Here we can see the vast difference that is caused by using a single loop versus a nested loop in a predicate. The single loop program that uses two hop information as an assembled binary size of 28 bytes, whereas the assembled size of the one hop information program has a size of 68 bytes. This program size may eventually become important when reaching the maximum allowed size of a single packet, as it is unlikely that a program like this will be able to go much over 100 bytes (without support for splitting the program up into multiple packets).

Also included in the programs are comments placed there by the compiler to indicate what the code is doing, the aim is that the developers will be able to manually tweak the code to try to reduce the size of it. Perhaps one of the startling differences is the benefit of some of the special
array opcodes as can be seen in Figure 21 which is only 13 bytes long. Without the AMEAN opcode there would need to be a loop in IR to calculate the mean, by shifting this responsibility to the interpreter the code size can reduced by eliminating an interpreted loop.

```plaintext
//TARGETING all
//FUNC 2 AS temp
//STORING 2 IN twohopn
//temp(this)
THISC 2

[all]
function 2 as temp returning float in
using Neighbours(2) as twohopn in

   abs(temp(this) - mean(temp, twohopn)) <= 10

AMEAN 255 2
FSUB
FABS
IPUSH 10
ICASTF
FLEQ
HALT
//VD 2 = 255
```

Figure 21: Compiled mean temperature check of Figure 14

One final optimisation that is worth mentioning is that, while we have support for it, we will often not generate code that involves pushing floats onto the stack. This is because if it is possible it we can save a byte of the program size by instead pushing an integer and then casting it to a float. When pushing a float we need 1 byte for the opcode and 4 bytes for the float being pushed giving 5 bytes of program. In the optimised case we need 1 byte for the IPUSH opcode, 2 bytes for the integer and 1 byte for the ICASTF opcode giving 4 bytes of program instead of 5.

### 3.5 Testing

As the virtual machine and its parsers can be fairly complex and difficult pieces of code to understand it was very important that we had test cases that validated that the functionality was correct. To do this three sets of tests are performed. Of them two are unit tests, one that tests that assembled bytecode can be correctly executed and the other that tests that the parser/compiler can correctly parse and generate code. After this is an integration test, that checks if the output of the parser/compiler converted into bytecode by the assembler can be correctly executed. To test the correct execution set data is given to the virtual machine, which can then be used in the predicates.

Different things are tested for each of the integration tests. When testing the assembler and virtual machine, short programs of opcodes are provided to test simple activities such as pushing onto the stack, calling functions or arithmetic. This is simply to test the functionality of the virtual machine. As we can be sure that the functionality is correct, when testing the compiler more complex predicates that we have designed the system to use are tested. We expect that for some of these tests scripts the code would be the same for real world uses.

### 3.6 Data Transmission

So far we have covered what we wish to evaluate and how that shall be evaluated, however, we have not talked about how data shall reach its destination to be evaluated. The how of evaluating data is split into two parts: ‘where’ the predicate should be evaluated and ‘when’ the data should be disseminated.

Regarding where a predicate should be evaluated there are two choices. Either there is collection of all the networks state at a single location designated as a sink (such as the base station), this is what traditional Global Predicate Evaluation algorithms would do. Or, the predicate could be evaluated in the network at some target node. We intend to investigate both of these.
Considering when the data should be disseminated there are again several options available. The first and most obvious is to simply periodically send out the information. However, periodic data transmission could potentially inefficient, so we will also investigate event-based dissemination. Finally, nodes could also request information when it is desired. We investigate all three of these.

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>Periodic</td>
<td>Periodically asks neighbours for information</td>
<td>Periodically aggregates information to sink</td>
</tr>
<tr>
<td>Event</td>
<td>Periodically checks for changes, on change floods information locally</td>
<td>Periodically checks for changes, on change aggregates data to sink</td>
</tr>
</tbody>
</table>

Table 5: Description of predicate evaluation behaviours

### 3.6.1 Local Event

When gathering data in-network using an event-based update mechanism and evaluating the predicate locally in-network there two things of importance. The first and simplest is knowing that data has changed. Because there are no ways to plug into the OS and receive notifications that a section of memory has changed it means that the local data needs to be checked periodically for a change. To do this the function that the library user has provided to return the node data will be called and then compared against some history. This history is simply the last piece of data that was sent into the network. A comparison function needed to be supplied by the user because doing a simple comparison of the memory may lead to change events being fired even though values have not changed. There are several examples where this is important. The first is that floating point calculations can produce different representations of the same results, bitwise the values may differ but logically the values are the same. Another case is that many sensors have a tolerance associated with them, by checking the values with a function this tolerance can be taken into account and events will not be fired if the data has not changed significantly. Finally, using a tolerance function allows the library user to specify the difference that is meaningful to them, even if it is different to the sensor tolerance.

As we check periodically to detect a change and then fire events, it means that there is still the possibility that we will miss some changes. For example a sensor value may spike in between measurements and this may never be detected due to how the checking period aligns with the spikes.

When an event is fired that indicates that the node’s data has changed it means that the data will need to be sent to some of the nodes in the network. Our implementation simply floods the information the maximum number of hops a predicate is asking for data. For example if there are 3 predicates in the network using 1-hop, 2-hop and 4-hop data, every node will flood their data 4-hops when it changes. This solution works because every node keeps a record of all the predicates in the network even if they are not evaluated by every node.

There is the potential for optimisations here by using knowledge of the node’s neighbours to detect if a node actually needs the current nodes data and then only flooding the data if that is the case. For example if there is one predicate in the system that is targeted to a single node that needs 3-hop information, but the node whose data has just changed is 4 hops away, that node would not need to send its information. However, this would mean that every node would need an accurate picture of the network state. This may be energy intensive to calculate, so we leave this optimisation to be investigated in later work.
3.6.2 Local Periodic

The major difference between evaluating data periodically when evaluating a predicate in-network is that instead of data being sent to nodes when it changes, data is only sent to nodes when it is requested. The idea is that when a node is evaluating its predicates it finds the maximum distance it needs information for and floods a message that many hops asking for data and each of those nodes reply back along the path that the request message came from. After a node evaluates all its predicates for that period it then forgets all that information. The reason this is done is because in the event based dissemination there is no ability to detect changes in topology and remove invalid data. For example say a cluster head that has successfully delivered information in the past crashes. As that information has been delivered and is only updated when it changes all the nodes simply assume that the data remains the same and will continue evaluating the predicate as if the cluster head had not crashed. Neighbours of the failed cluster head could detect and send this information, but as that involves network information it is not an application predicate and is out of the scope of this work. So the solution was to simply consider the information available at the node evaluating a predicate at that instant.

Instead of requesting information every node could simply periodically broadcast it. However, that has the disadvantage of causing nodes to broadcast information when they don’t need to (a problem from event-based dissemination) and also could lead to problems when nodes clocks are perfectly aligned. For example nodes could consistently receive old information from nodes if they keep sending data that was meant for a previous round such that is it only received during the current round of predicate evaluation. By asking for data these issues are mitigated.

When using the N-Hop Request library to implement this predicate evaluator there is an important issue that needs to be considered: when we have sent the data request message how long should we wait for? We cannot simply wait until all the results are returned to the evaluating node because (i) we do not necessarily know what node’s data we will be waiting for and (ii) we may not receive that data anyway. We could wait for $2 \times D \times T_{send} + \epsilon$ time units, where $D$ is the maximum number of hops we are asking for information from, $T_{send}$ is the amount of time it takes for a message to try to be sent and $\epsilon$ is a small extra amount of time to wait. However, knowing what $T_{send}$ is can be difficult, and is based on many complicated factors. For example, if the MAC protocol in use uses CSMA/CD or CSMA/CA then it is possible, when trying to send a message, to enter into an arbitrary number of back-off period while waiting for the channel to be free [107] p. 296. If there

![Figure 22: Predicate Evaluation Local Event (PELE)](image)

![Figure 23: Predicate Evaluation Local Periodic (PELP)](image)
is an upper bound on the number of retries then we can calculate a maximum $T_{send}$, otherwise we cannot. Therefore, for our implementation we chose to use a fixed large value in which it was reasonable to expect all the responses that would return, would return within that period. We expect that in the future, this wait period will need to be some function of $D$ the number of hops of information being asked for to support requesting information from deeper in the network.

### 3.6.3 Global

When considering evaluating predicates at a sink, the data can simply be aggregated to it along a tree. For both event-based and periodic data generation the way the data is generated can remain the same, the requesting of data is simply removed (for periodic) and instead of flooding the information to neighbours it is aggregated instead.

However, this change causes a few knock-on effects. For the aggregation of neighbours and data in PEGP the behaviour is as expected, data is generated at the leaves and aggregated onwards towards the tree’s root node where it is used. When aggregating data that is generated in PEGE, it is not the leaves that periodically generate data. Instead every node could potentially cause a message to being forwarded through the tree. The big change is that every time a packet is forwarded the current node’s data is aggregated to with it. This means that nodes closer to the root node of the tree have more opportunities to aggregate their data. Overall we do not believe this will affect the evaluation behaviour, as in the worst case more recent information may be available when it might not otherwise have been so.

The biggest issue with the global style of predicate evaluation is the latency that is introduced between gathering the data from the nodes in the network and the evaluation of the predicate. One problem is that the data in the network may have changed since it was gathered, a larger problem is that data will be compared at potentially very different times due to the aggregation delays. This was not as much of an issue with local in-network evaluation because the data was quickly flooded back to the target node. However, with tree aggregation every hop towards the sink needs a long wait to gather as much data as possible to reduce number of messages sent. This could potentially lead to a number of predicates being incorrectly evaluated by using data from different times.

![Diagram](image)

**Figure 24:** Neighbour Aggregation used for global predicate evaluation

Both the event-based and period versions of global predicate evaluation require the two processes of neighbour detection and aggregation running. Initially we planned to use this data to visualise the state the network is in. However, it turns out that it is a vital component of global predicate evaluation. Without this information we would be unable to work out how many hops the data is from the node that is currently being evaluated.

As every predicate is evaluated is evaluated at the root of the aggregation tree, it becomes necessary for that root node to be able to pretend that they are the node where the predicate would usually have been evaluated. This is so the relevant neighbours can be used, and also so the virtual machine will use the correct data for the this variable.
Between the two versions of global predicate evaluation there are only minor differences in how the algorithm is implemented. The first is that when sending data PEGE checks to see if the data has changed before sending it, PEGP instead checks if the node is a leaf in the tree and relies on the aggregation of intermediate nodes to gather all the network’s data. The second is a consequence of the first difference, PEGE will keep and update the information it receives. PEGP instead only used the last round of information that has been received. So after every predicate has been evaluated at the end of a predicate evaluation round the node information is cleared. This feature was added to enable support for (untested) handling failures in the network such as crashes or for handling the case when the nodes are mobile.

3.6.4 Reply

Once a node has reached the point of having a result after evaluating a predicate it needs to do something meaningful with that information so that users looking for results at the base station also end up knowing this information. The first and most obvious solution would be to inform the base station of every result, be it success or failure. Unfortunately while this is the most
comprehensive response it is would also require the most energy as every node would need to send a repose message for every predicate they were evaluating.

\[ C = |P_{all}| \times |V| + |P_{single}| \] (2)

The alternatives are to either send a message when a predicate has failed or when a predicate succeeds. This allows only a fraction of \( C \) messages to be sent. However, there are a number of differences that make deciding to send success or failure an important decision. First of all is how the result is represented. The comprehensive solution has the advantage that the result of a predicate can be put into an unknown state. Once a result message is received the result of that predicate can be moved into a failed or succeeded state. For success or failure messages you would need to assume the opposite of the messages that you are waiting for (i.e. assume predicates failed before receiving the succeeded message and vice versa) and change the state once the message is received. If messages are lost then this will lead to false notifications of success when waiting for failure messages and false failures when waiting for success messages.

The other issue to consider is what should be reported. If success messages are the chosen form of the response, then what useful information could they contain? One would expect the state that the predicate was evaluated with, but this information is not nearly as useful as the state that caused a predicate to fail. Another factor is that to create a useful description of why an error message failed is complex and would require more resources than simply evaluating the message. This means that practically it would be best for a failure message to report the state that caused the predicate to fail back to the base station and then expect further analysis on it there.

As we expect a predicate failure to be an infrequent event, our libraries have been designed to report only predicate failures. This should save energy in the form of messages. However, we expect the system to be vulnerable to not showing the full extent failures (falsely appearing successful), when messages are lost.

### 3.7 A Consideration

One of the main examples provided was the problem of assigning slots in a TDMA MAC protocol, where one would wish to ensure that no node within two hops of any given node has the same slot. Interestingly there are multiple ways to structure the predicate to perform this evaluation.

The first predicate we might consider is the one where for every node, we get the 2-hop neighbour-hood and check that none of those nodes have the same slot as the initial node. This means that we would need to send:

1. A message from the base station to the node asking it to evaluate the predicate
2. A message from the node to each of its 2-hop neighbours
3. Each 2-hop neighbour needs to send a message back to the node
4. The node would need to wait for its neighbours to send the messages, then evaluate the predicate. This would imply that the node would need to know who is in its two hop neighbourhood
5. The node would need to report to the base station the result of the predicate

So we would need at best \(|V| + |\Delta_{sink}(n)| + M(P_2)\) messages to evaluate this predicate for the single node \( n \in V \). We need \(|V|\) messages to flood the predicate to every node, \( |\Delta_{sink}(n)| \) is needed to send a failure reply back to the sink and the last component \( M(P_2) \) is the message cost of evaluating the predicate on that node. For example \( M(P_2) = 2|\text{Neighbours}(n, 2)| \) for PELP as a message has to be flooded out two hops to ask for the information and then those nodes need to respond.
∀n ∈ Nodes·
∀n' ∈ Neighbours(n, 2)·
∀n'' ∈ Neighbours(n, 1) ∪ {n}·
∀n ∈ Nodes·
∀n' ∈ Neighbours(n, 1) ∪ {n}·
∀n'' ∈ Neighbours(n, 1) ∪ {n}·

The second predicate we could consider is the one where every node checks their 1-hop neighbourhood for slot collisions. Here we model the 2-hop nature of the predicate differently, because instead of checking on the node that we want to check for slot collisions, we check on the node between two nodes that might have slot collisions.

1. A message from the base station to the nodes adjacent to that we asking to check slot collisions
2. A message from the node to each of its 1-hop neighbours
3. Each 1-hop neighbour needs to send a message back to the node
4. The node would need to wait for its neighbours to send the messages, then evaluate the predicate. This would imply that the node would need to know who is in its two hop neighbourhood
5. The node would need to report to the base station the result of the predicate

Initially one may consider that this 1-hop predicate can simply be evaluated on a node $n' \in V$ where $\{n, n'\} \in E$. However, this means that this predicate is no longer equivalent to that 2-hop predicate. For example if $n$ has another neighbour $n''$ whose 1-hop neighbourhood is not contained within $n'$’s 1-hop neighbourhood ($\text{Neighbours}(n'', 1) \setminus \text{Neighbours}(n', 1) \neq \emptyset$) then $n''$ would also need to be checked.

So the number of message that would be required to evaluate this 1-hop predicate on the required node is $|V| + \sum_{n' \in \text{Neighbours}(n, 1)} \Delta_{\text{sink}}(n') + M(P_1)$. Where $|V|$ is required for the same reasons as the 2-hop predicate and the second part is because every neighbour $n'$ of $n$ needs to send a failure message. $M(P_1) = \sum_{n' \in \text{Neighbours}(n, 1)} 2|\text{Neighbours}(n', 1)|$ for PELP because every neighbour $n'$ of $n$ needs to ask $n'$’s 1-hop neighbourhood for information.

So this introduction of the 1-hop predicate has led to a spreading out of the evaluation, but with roughly the same amount of energy usage. The energy usage is roughly the same as the number of nodes from which data is requested is the same in $M(P_1)$ as $M(P_2)$, the 1-hop predicate will likely require slightly more energy as nodes will be asked their data multiple times by other nodes. However, the 1-hop implementation has a good advantage, which is that by asking nodes neighbouring the 1-hop neighbours of $n$ to also evaluate the predicate they can take advantage that some of the predicate is already being evaluated. As shown in Figure 27 a new node can be set to evaluate a predicate and that node will take advantage of the already evaluated neighbouring information that is not necessary to repeat. Whereas the 2-hop predicate would need to repeat a lot of data request and evaluation if it were to be evaluated on multiple nodes. So Adding more and more nodes that evaluate the 1-hop predicate will scale much better than the 2-hop predicate simply because the way that the evaluation is distributed and the way that distribution can prevent duplicate requesting of information. So checking an entire network for slot collisions should be much more energy efficient when using the 1-hop predicate.
However, there is one final factor to take into account that even though using 2-hop information will require more energy it is the simpler predicate - it only has a single loop with a simpler check. This predicate requires 28 bytes instead of the 68 bytes that the predicate that required 1-hop information uses. When predicates get larger and more complicated it may be important that they can be expressed in a simpler way that may perhaps use more energy, but also are smaller and need less stack space to be evaluated. If the cost of performing a firmware update across the network is very high and would be needed to support larger predicates, then the energy cost of the simpler predicate may end up being worth it.

### 3.8 A Distance Issue

The only network primitive we have implemented is a function that gets information on nodes within a certain number of hops of the asking node. There are a number of other network primitives that might be useful for both network and application predicates. One of these is the ability to get the distance between two nodes.

An example of a predicate that uses \( \text{Neighbours}(n, h) \) is shown below. This predicate is for Hierarchical Clustering where \( H \) is a parameter that defines the distance between cluster heads. The predicate checks that for every node there is a neighbour in the \( H \)-hop neighbourhood that is a cluster head.

\[
\forall n \in \text{Nodes} \cdot \\
\exists n' \in \text{Neighbours}(n, H) \cdot \\
\text{IsClusterHead}(n)
\]

An alternative way that we might specify this predicate is to check that the distance between a node and the cluster head (that it knows the address of) is within \( H \) hops.

\[
\forall n \in \text{Nodes} \cdot \\
\text{IsClusterHead}(n) \lor \text{Distance}(n, \text{ClusterHead}(n)) \leq H
\]

However, there is an issue with this predicate. How would we implement the \( \text{Distance}(n, n') \) primitive? With \( \text{Neighbours}(n, h) \) we know how far we need to ask for information (\( h \) hops), with \( \text{Distance}(n, n') \) the maximum distance that we will need to wait for information to return from is the diameter of the network. Unfortunately we cannot assume that the diameter is known. If
instead we take a time-out approach there is no way that we can guarantee to wait for the correct amount of time. For example if we wait enough time to check if a node is at maximum $D$ hops from the given node the node will need to wait for time units. However, if the node we are trying to find the distance of is at $D + 1$ hops then we will not believe the two nodes are connected. To reliably wait to check if two nodes are connected in a network where the diameter is not known we will need to wait for $\infty$ time units. Thus making $Distance(n,n')$ impossible to implement without knowing the diameter. Due to this issue we have not implemented such a network primitive function.

### 3.9 Optimising on Predicate Structure

When evaluating predicates in-network, it is possible to consider optimisations due to the structure of the predicate to reduce the amount of information that is requested. For example take conjunctive predicates where the structure of the predicate $P$ is $P = P_1 \land P_2 \land \ldots \land P_n$. If one of the $P_i$ is false then the result of the predicate $P$ is false ($\text{False} \land P \Leftrightarrow \text{False}$). This means that the statements could potentially be evaluated in an order such that the $P_i$ that requires the fewest resources is evaluated first and the $P_j$ that requires the most resources is evaluated last.

The aim of this optimisation is to reduce energy usage by sending fewer messages. For the two types of data gathering in-network and at the sink, when gathering this data at the sink it would not be of any use as the data is sent to the sink in an event-based or periodic way. The sink doesn’t ask for this information, so it is not possible for it to not ask for certain pieces of data. This is also true of event-based in-network predicate checking. So the only time this optimisation would be of use is when gathering data for in-network predicates that are checked periodically.

For some predicates it may make obvious sense to implement such an optimisation, such as if there is a conjunctive predicate that needs 1-hop information and 15-hop information. If the 1-hop predicate is often false, then it saves asking for information from all nodes that are within 15 hops of the evaluating node. However, as many of the predicates are not likely to require information from that distance (we expect one to two hops, with the possibility of three hops) it is an optimisation that would not serve to optimise what we are targeting our system to evaluate. Another issue is that many of the predicates we have thought up are not of this form, they mostly have quantifiers as the outer structure with other logical operators inside that quantifier. It may be the case where you have a predicate of the following structure $P = P_{local} \land P_{network}$ where $P_{local}$ needs no network information, here there is the possibility to optimise the network request out if $P_{local}$ is false. However as adding this feature would increase the firmware size we decided against adding more optimisations. This was because the software began to reach the limit of the firmware size when the virtual machine was integrated with the message communication. However, investigating these kinds of optimisations is a future possibility.
4 Implemented Algorithms

4.1 Container Library

Contiki comes with a list container\(^4\), however, when we were developing with their linked list we found that it was awkward to use in some situations (for instance creating an array of lists) so we decided that we needed to find a better list library. Our problem was that other container libraries were unlikely to be optimised to the low memory requirements or support the special compile chain used by Contiki. So rather than waste time searching and integrating an external library we decided to write our own set of containers. By doing so, the main benefit we gained was that we knew how the containers worked so had a better understanding of how to use them.

Another reason to develop our own containers was that we desired a list container where the data was stored in an array, this was for because it provides lower memory overhead and the types of operations we would be performing (append and empty) meant that an array backed list would be better. The lower memory overhead comes from the fact that an array-based list doesn’t need to store a pointer to the next element as it is implicit (due to the contiguous memory) that a pointer to the current element’s pointer plus one is the next element. This means that in a list of \(N\) items our singly linked list implementation will require \((\text{sizeof}(T) + \text{sizeof}(void\ast)) \times 2\) \(\times N\) bytes of memory, whereas our array list implementation will require \((\text{sizeof}(T) + \text{sizeof}(void\ast)) \times N\) bytes of memory.

Our linked list implementation uses more memory compared to Contiki’s implementation because their list is intrusive\(^5\). What is meant by this is that the pointer to the next item in the list is contained in the structure stored in the list. Meaning their list uses the same amount of memory as our array list. However, because the list is intrusive it makes it more difficult to use as the implementation detail leaks into the structure using the library, this was why developing our own non-intrusive list made it easier to develop code.

We also developed containers that helped abstract certain concepts (such as list uniqueness or accessing elements by key) to decrease code duplication and allow us to express the code in a higher-level way making implementation easier. One thing to note is that for the containers we did not focus on implementing some of their functions to the standard complexity. For example our map implementation has requires \(O(N)\) for both insert and fetching, where these may be implemented as \(O(1)\) on average (hash tables) or \(O(\log N)\) (trees), however due to the requirements of low memory we decided to keep using the array as the backing store for the data with the aim to keep memory for at the expense of non-optimal container management functions. The increased time complexity should make little difference as the containers tend to have few elements in them (tens, possibly

---

\(^4\)Header: [https://github.com/contiki-os/contiki/blob/master/core/lib/list.h](https://github.com/contiki-os/contiki/blob/master/core/lib/list.h) Source: [https://github.com/contiki-os/contiki/blob/master/core/lib/list.c](https://github.com/contiki-os/contiki/blob/master/core/lib/list.c)

hundreds), rather than millions, where complexity would start to be important [48, p. 6].

Finally, all of these custom containers are tested by a test suite that checks for functional correctness. The test suite was also run using valgrind to check for memory leaks and corruption.

4.2 Network Library

In order to facilitate the development of our predicate checking algorithms, numerous separate libraries that encapsulated specific functionality needed to be implemented. The reason that these algorithms and protocols were implemented as libraries were so that our code remained well encapsulated, leading to easier testing and development.

4.2.1 Multi-Packet Unicast

In Contiki sending a single packet has its limits, there is only so much data you can get in a single packet. There are two C macros that are relevant to this PACKETBUF_HDR_SIZE which is set to 48 bytes and PACKETBUF_SIZE which is set to 128 bytes. This means that in general we will only be able to send a packet containing 128 bytes of information. But if a protocol called for sending a data structure that spans more than 128 bytes, then a way to send multiple packets and a receiver that can put the packets back together in the correct way to deliver the data is required.

Contiki’s data transfer primitives for their Rime protocols are fairly hidden because of their focus on uIPv6. We found ruldolph0 first and then rucb (Reliable Unicast Bulk Transfer) much later. However, both of them have problems, both APIs are very convoluted and seem more geared towards sending very large files across the network. This is very different to our aim, which is to send relatively small packets of data to a specific neighbour. So we wrote our own API that would split data of any given size and reassemble it once received, our API was not focused on processing data chunks like ruldolph0 or rucb, but simply operated on a block of memory given to it. This is a case where a simpler API made development much easier for us. Admittedly because our implementation uses dynamic memory allocation it could potentially perform worse, but because the energy of sending and receiving messages dwarfs the energy usage of the CPU, we feel that easier development tradeoff made it worth it. Also, now that code has been written to use this API, the multipacket library itself could always be rewritten to use rucb as a base to run more efficiently.

4.2.2 N-Hop Request

We developed the N-Hop Request communication layer very early in the project. Our application needed a way to ask neighbouring nodes for information, and these messages sometimes needed to travel multiple hops in the network to do so. Initially we consulted the Contiki libraries to see if any of the existing network primitives could be used for this purpose.

We first considered Mesh which sends a packet to any node on the network, this seemed like it would be a prefect fit for sending data back to the originator node. However, issues arose when we discovered that it could only send one unique packet across the network at a time. This was due to the way that Mesh was implemented, it only maintains a record of the most recently received packet on the network. This assumption is fine if you assume that you only have 1 sender like a base station but when you have multiple senders sending different packets it fails. This could only have been scaled by having a Mesh connection per node, which is not scalable at all. Next we considered Trickle. Unfortunately, like Mesh, Trickle had similar shortcomings which made it unsuitable for the task.

---

Having explored several options from the Contiki source library we realised that N-Hop Request had a very specific set of requirements. We found that no one library offered all the functionality we needed; therefore, we opted for developing our own communication layer and API to handle the routing of these messages. The layer would need to flood the network, a fixed number of hops from the source, whilst allowing multiple nodes to use the same layer, with different messages. For this we built upon Contiki Rime communication libraries.

The libraries that we used were stbroadcast (Stubborn Broadcast) and runicast (Reliable Unicast). Stubborn Broadcast is used to send the request data messages n-hops into the network. Every node that receives the request checks the time-to-live (ttl) of the message; if it is greater than 1 then, it decrements the ttl and forwards the packet on to it’s neighbours. Stubborn Broadcast was used as it broadcasts a message at a given rate, allowing this to send a few copies of the message at random intervals increases the chance that the message will be received by the neighbours. To keep energy costs down the number of message repeats is kept low.

When a node receives a request message it keeps a record of from what node it came from and which node is requesting the information, this is so that when replies are being sent they can simply follow this path instead of being broadcasted. After this information has been stored the node then waits a fixed amount of time before sending a reply. This is done so that request messages have time to propagate away to avoid collisions. Once the timer times-out Reliable Unicast is then used to send the reply to the neighbour recorded as being on the path to the node that requested this information. Reliable Unicast is used in attempt to provide reliable transmission and reduce energy usage, as in the best case it will simply involve the overhead of a single ack message.

If a node receives the reply packet and isn’t the intended target (the originator) they forward the packet onto the node that they received their request message from, eventually the packet will arrive back at the originator who can then deliver the message. In order to minimise message sends, each node maintains the most efficient node to get back to the originator. This is based on the number of hops it takes for a request message to reach the node, a message from a node with 3 hops left is a more efficient route than a message from a node with only 2 hops left. Finally, once a pre-determined wait period has expired the gathered data is passed to the predicate evaluator and any further data received is not delivered.

![Diagram](image)

Figure 28: Example of N-Hop Flood when N=2
Process $j$ - n-hop-req

variables

% The key is the node this entry is about.
% The entry contains how many hops away the node is,
% what node a reply should be forwarded to and the message id the entry is relevant to.

records: map of address to (hops: int, forwardTo: address, id: int) init \emptyset;

messageId: int init 1;

sendData: timer init $\perp$;

constants

% Send Data Period, the time between receiving a request for data, and sending it
$P_{sendData}$: time;

actions

% Sending Data Request Message

nhopreq.send:: function(hops) \rightarrow

bcast($j$, $j$, messageId, hops, 0);

messageId := messageId + 1;

% Receiving Data Request Message

receiveRequest:: recv(originator, sender, id, ttl, hops) \rightarrow

if (originator $\neq$ $j$) then

respond := False;

if (originator $\in$ keys(records)) then

(recHops, recForwardTo, recId) := records[originator];

if (hops < recHops) then

records[originator] := (hops, sender, recId);

fi;

(recHops, recForwardTo, recId) := records[originator];

if (id > recId) then

records[originator] := (recHops, recForwardTo, id);

respond := True;

fi;

else

records[originator] := (hops, sender, id);

respond := True;

fi;

if (respond)

(recHops, recForwardTo, recId) := records[originator];

unicast.send(originator, $j$, 0, NodeData()) to recForwardTo;

bcast(originator, $j$, id, hops - 1, hops + 1);

fi;

fi;

% Received Data Forward Message

forwardData:: unicast.recv(target, sender, hops, data) \rightarrow

(recHops, recForwardTo, recId) := records[target];

if ($j$ = target) then

nhopreq.deliver(sender, hops, data);

else

unicast.send(target, sender, hops + 1, data) to recForwardTo;

fi;

\fi;

Figure 29: N-Hop Request Algorithm
4.2.3 N-Hop Flood

The previously mentioned algorithm was developed so that nodes could ask for information from its N-hop neighbourhood when it is needed. This was used for when predicates were asking for data periodically, however, if nodes are sending data when said data changes then N-hop request is not as applicable. Therefore, we developed N-hop flood, which floods information the required N hops.

The N-Hop Flood protocol keeps a queue of the messages that it needs to send. Every $P_{send}$ period it will broadcast what is on the top of the queue and increment the number of times that message had been sent. Once a message reaches the maximum number of retransmits it is removed from the queue. New messages are appended to the end of the queue so they are sent after messages added to the queue earlier.

When a node receives a flooded message it checks to see what the time-to-live (TTL) of the message is. If it is not 0 then that the message is added to the queue to continue being forwarded. The node will also check the id included within the message. If the node had not received a message with that id from the sender before or it had received a message from that sender but from a route that was longer the message is delivered to the user of the library.

\begin{figure}[h]
\centering
\caption{N-Hop Flood Algorithm}
\begin{verbatim}
Process j - n-hop-flood
parameters
  \% The send period for this predicate
  $P_{send}$: time;

  \% The max number of times to re-transmit
  $maxretx$: int;

variables
  id: int init 0;

  sendData: timer init $P_{send}$;

  \% The queue of packets that need to be broadcast
  \% Tuple contains: (id, sender, ttl, hops, retx, data)
  packetQueue: queue of (int, address, int, int, int, struct) init \emptyset;

  \% The map of all messages received from the nodes N-Hop neighbours
  \% The entry contains a tuple of (packet id, hops)
  seen: map of address to (int, int) init \emptyset;
\end{verbatim}
\end{figure}
4.2.4 Event Update

The event update library was developed for use with predicates being evaluated locally in-network where information needed to be disseminated a given number of hops once the mote’s data changed. This library periodically generates the current state of the mote and then checks it against the previously sent state. If there is no previously sent state or the state has changed as specified by a differs function, then the recently generated state is flooded the required distance. For our usage this distance will be the maximum number of hops that a predicate needs information from.
Figure 32: Process of event update sending changed data

```
Process j - event-update
variables
    period: timer init P_{generate};
    previous: struct init ⊥;

constants
    P_{generate}: time;
    differs: function takes (struct, struct) returns boolean;
    data: function takes () returns struct;
    chance: real;

parameters
    distance: int;

actions
    check:: timeout(period) →
        force := RandReal(0, 1) ≤ chance;
        changed := previous = ⊥ ∨ differs(data(), previous);
        if (force ∨ changed) then
            previous := data();
            nhopflood.send(j, previous, distance);
        fi;
        set(period, P_{generate});

    receive:: nhopflood.recv(source, data, hops) →
        % Prevent delivery if being told that the current node’s data has changed
        if (j ≠ source) then
            % Inform library caller of data change
            event-update.callback(data, hops);
        fi;
```

Figure 33: Event Update Broadcast Algorithm
4.2.5 Tree Aggregation

As has been previously mentioned Tree Aggregation is a very useful technique employed in wireless sensor networks to aggregate a set of data to a single node in the network. It is very useful because of its ability to convey the same amount of data in potentially fewer messages. The algorithm works by first creating a tree structure such that nodes know of a parent node that they should unicast their messages to. Once this tree is set up the nodes (typically leaf nodes) unicast data to their parents, this can either happen when an event occurs or it can happen periodically. When a parent node receives a message it waits for a certain amount of time for more messages to arrive, every message it receives is stored and aggregated. After the time period is up the aggregated data is forwarded to its parent node. Once the data reaches the destination it is delivered to the user of the library.

The user of this library needs to implement four important functions:

- \(\aggregation\) This function aggregates stored data with data message that has been received
- \(\own\) This function aggregates the node’s own data into the stored data
- \(\read\) This function does the initial reading of data from a message and creates the local stored data
- \(\write\) This function writes the local stored data back to a packet

---

Process \(j\) - tree-aggregation

variables

- parentdetect: timer init ⊥;
- seensetup, collecting, leaf: bool init False, False, True;
- besthop: int init INTMAX;
- bestparent: address init ⊥;
- aggregation: timer init ⊥;
- collecting: bool init False;
- stored: struct init ⊥;

constants

- % How long to wait for messages to aggregate before forwarding what the node has
  \(P_{\text{aggregation}}\): time;
- % How long to wait for parents to be detected
  \(P_{\text{parentdetect}}\): time;

parameters

- % The address of the node data should be aggregated to
  sink: address;

---

Figure 34: Tree Aggregation Algorithm - variables
% Set up tree: initialise
startup:: init \rightarrow
bcast(j, \perp, 1);

% Set up tree: receive message
receive:: recv(source, parent, hops) \rightarrow
if (j \not= sink) then
  if (~seensetup) then
    seensetup := True;
    set(parentdetect, \mathit{P}_{\mathit{parentdetect}});
  fi;
  if (hops < \mathit{besthop}) then
    \mathit{bestparent}, \mathit{besthop} := source, hops;
  fi;
  if (leaf \land j = parent) then
    leaf := False;
  fi;
fi;

% Set up tree: finish detecting parents
check:: timeout(parentdetect) \rightarrow
if (\mathit{besthop} = \text{INTMAX}) then
  bcast(j, \mathit{bestparent}, \text{INTMAX});
else
  bcast(j, \mathit{bestparent}, \mathit{besthop} + 1);
fi;

actions

Figure 35: Tree Aggregation Algorithm - Setting up tree

% Send data function
treeagg.send:: function(data) \rightarrow
  multipacket.send(j, data) to \mathit{bestparent};

% Set up tree: receive message
receive:: multipacket.recv(source, data) \rightarrow
if (j = sink) then
  stored := \mathit{read}(data);
  stored := \mathit{own}(stored);
  treeagg.deliver(\mathit{write}(stored));
else
  if (collecting) then
    stored := \mathit{agg}(stored, data);
  else
    stored := \mathit{read}(data);
    collecting := True;
    set(aggregation, \mathit{P}_{\mathit{aggregation}});
  fi;
fi;

% Set up tree: finish detecting parents
finishagg:: timeout(aggregation) \rightarrow
  multipacket.send(j, \mathit{write}(stored)) to \mathit{bestparent};
collecting := False;
stored := \perp;

actions

Figure 36: Tree Aggregation Algorithm - Sending data
4.2.6 Neighbour Detect

To aid in debugging a vital piece of information that will be required about the network is its topology. Trying to analyse network state without knowing which nodes are neighbours of other nodes, makes drawing meaningful conclusions from that data much harder. This meant that we needed a way to send neighbour information back to the sink. Thankfully part of the job is already accomplished as Contiki comes with a library to perform neighbour detection\textsuperscript{11}. We extended that library in two ways.

The first was to add more features to Contiki’s neighbour detect library. As that library was very simple and supported only saying that a neighbour had been detected (with the option of providing a integer value as well). As we were aiming for this code to support changes in the network, such as nodes leaving or joining neighbourhoods, we added the concept of a round. Instead of just knowing about neighbours at that instant in time, it allowed the library to maintain some kind of history about what nodes were neighbours of other nodes in the past. Whenever new nodes are detected they are recorded, however, if a node has not been detected for a certain number of rounds then that node is removed from the record.

For this to be useful we needed a way to get this data back to the sink node. There are many ways to do this, but one of the best is aggregating along a tree because it can combine many messages into a single one and forward that aggregated message instead of lots of smaller messages \textsuperscript{[78]}. The Neighbour Aggregation part uses the Tree Aggregation library that we have developed, the function that is used to aggregate data together is $\cup$ (union), this allows data to be aggregated with other data and allows that data to be updated. The nodes own data is set to be $\text{node-data}$, where $j$ is the current node’s address and $\text{round-data}$ is the set of node addresses provided by the neighbour-detect.callback callback in Figure 37.

\begin{equation}
\text{node-data} = \{ \{ \text{other}, j \} | \text{other} \in \text{round-data} \} \tag{10}
\end{equation}

\textsuperscript{11} neighbour-discovery.h \url{http://contiki.sourceforge.net/docs/2.6/a00344.html}

(a) Example Network

(b) Logical Tree imposed on network when aggregating neighbours
Process $j$ - neighbour-detect

variables

- $period$: timer init $P_{\text{round}}$;
- $\text{previous}$: map of address to int init $\emptyset$;
- $count$: int init 0;

constants

- $P_{\text{round}}$: time;
- $\text{missed}$: int;

actions

- % Check for changes
  - $\text{round}:: \text{timeout}(period) \rightarrow$
  - $\text{previous} := \{(addr, round) \in \text{previous}| count - round \geq \text{missed}\};$
  - % Tell library caller that a round is finished
    - $\text{neighbour-detect.callback(values(\text{previous}), count)}$;
  - $count := count + 1$;
  - $\text{set}(period, P_{\text{round}})$;

- % Receiving Neighbour Discovery message
  - $\text{neighbour}:: \text{neighbour-discovery.recv(source, round)} \rightarrow$
  - % Update the round we last saw this node
  - if $(\text{source} \in \text{keys(\text{previous})})$ then
    - if $(\text{round} > \text{previous}[\text{source}])$ then
      - $\text{previous}[\text{source}] := \text{round}$;
    fi
  - else
    - $\text{previous}[\text{source}] := \text{round}$;
  fi

Figure 37: Neighbour Detect Algorithm

4.3 TDMA

To have a representative algorithm of what may be tested, TDMA (Time Division Multiple Access) was chosen to be implemented and have predicates written for it. The algorithm that was initially implemented was described by Saifullah et al. in [96, p. 4] and is outlined below:

1. Initially assign every node the smallest numbered channel
2. Every round every node broadcasts a message containing their ID and their currently assigned channel
3. When a message is received the neighbour set is updated and the node that received the message assigns its channel to be the lowest channel not assigned to any neighbour. Two neighbours are not allowed to change their channel in the same round, so a tie breaker is done and the node with the lower ID is allowed to change their channel
4. After choosing a channel the node broadcasts its ID and chosen channel
5. The procedure is repeated until every node cannot choose a smaller channel than its current channel
Interestingly enough when this was first implemented the algorithm only made sure that a node did not have the same channel as any of its one hop neighbours. It did not ensure that the one hop neighbours of any node would have unique channels (as would be required by TDMA to ensure no collisions). This is the kind of bug that running predicates would have assisted identifying.

To make sure that the channel allocation was suitable for TDMA, instead of nodes just sending their own channel assignment to its neighbours. It also included the assignment of the one hop neighbours it knows about in that message. This then allowed nodes to receive information on their two hop neighbourhoods, which they then based their decision on what channel to allocate to themselves from.

4.4 Testing

When testing our implementations there were a number of difficulties due to the differences in what was being tested, how that library needed to tested and under what circumstances. For the containers library we wrote a number of tests that simply created the relevant container and then performed an action on it. The test succeeded if the state of the container was as expected after the execution of the test. These tests were run under various conditions, including under valgrind to ensure that there were no memory leaks in these critical pieces of code.

The remaining libraries were difficult to test because they were communication libraries, so we developed applications that used them and made sure that in our tests these application performed as expected. To test Tree Aggregation without large packet support we developed a simple temperature and humidity averaging application. To test Tree Aggregation with large packet support (from our multi-packet library) we developed an application to aggregate neighbours back to the base station. This is the same application that we integrated as the neighbour detection library because it became vital for performing predicate evaluation at the base station.

N-Hop Flood, N-Hop Request and Event Update were all tested as part of the behaviour of the relevant predicate evaluation library they were used in (PELE, PELP and PELE respectively). The libraries were developed to ensure that we had correctly encapsulated the required functionality which meant that if we discovered an issue in one component we knew where we would need to look for the error.

When testing for memory leaks we couldn’t use the same approach as used to test the container libraries. This is because sensor network application never truly terminate, meaning there is no good point to check that all the memory allocated has been freed. What we instead did was run the applications we had developed in the simulators for a long period of time until the application crashed. We then used the stack trace to work out where the error was and fix that component. For memory leaks we also had to manually inspect all the allocation and freeing routines to ensure that the memory that was allocated is freed.

4.5 Visualisation Tool

In order to interface with the sensor network, we require the software running on the sink node to be capable with communicating the network’s status to a connected computer. We have developed a GUI written in Java for this purpose. It’s main functions include creating predicates to be monitored, deploying these predicates to the network, and displaying node’s neighbourhoods.

The tool has two main views, predicate view and network view, as shown in Figure 38 and Figure 39. Predicate view provides a list of predicates along with their status (satisfied (green), unsatisfied (red), unmonitored (blue) and unevaluated (yellow)), source code (written in our predicate scripting language), and more detailed evaluation results when available.

The network view on the other hand displays a visualisation of the WSN as a graph — with vertices representing nodes and edges representing neighbour relationships. The user can scroll freely back and forth between previous rounds and view the data contained in each view for any of them.
Figure 38: Visualisation Tool Predicate View
4.6 Summary

Either we were unable to find implementations of what we needed or it simply hadn’t been implemented, this unfortunately added a lot of development time to this project. We were very shocked that libraries such as tree aggregation and hop-limited flooding seemed unavailable in Contiki as we thought that these would be fundamental libraries for wireless sensor network development. Drawing parallels from recent developments in C++ standardisation, the committees involved are actively seeking new libraries to include in the next standard due to the benefit it brings the language [23]. We believe that if Contiki supported a number of additional containers other than a linked list and circular buffer as well as other higher-level networking libraries in Rime, then Contiki could be much more useful. We will be looking to see if we can contribute back anything we have developed.
5 Results

5.1 Methodology

In our simulations we tested networks of various sizes (15, 30 and 48 nodes) aligned in a grid. The nodes were placed such that every node had four neighbours, one to the north, south, east and west, except for nodes in the edge which had three neighbours and the nodes in the corner that had two neighbours. The sink was always placed in the top left corner and was always assigned the address “1.0”. The predicate checking algorithm was started as soon as the nodes had come online, the nodes then waited for 5 minutes to allow the predicate checking algorithm time to setup and then the TDMA algorithm was started. The simulation was run for 35 minutes overall and then it was terminated. After the predicate checking algorithm had setup, the predicate was checked every 2 or 4 minutes depending on the parametrisation. When setting up the Cooja simulator to run the simulations the radio medium was set to be “UDGM: Distance Loss”, the mote startup delay was set to be 1000ms and a random seed was set to be used on every reload.

To gather metrics on the energy usage of the motes, a Contiki library called rimestats\[12\] was used. This library is built into the MAC layer and records deep statistics, we simply used the sent and received message counts. Contiki comes with a library to estimate the energy usage \[39\], however, we wish to keep things simple and focus on the number of messages being sent and received.

In order to calculate how many messages the TDMA algorithm involved we implemented our own sent and received counters that were incremented when a message was sent or received. As the TDMA protocol was implemented with simple broadcasts there is a one-to-one correspondence between the number of successful broadcasts and the number of transmissions done at the MAC layer, the same is also true for receiving a message. To calculate the message cost of the predicate evaluation algorithm, the difference between the total and the TDMA messages was taken.

To check that a predicate was successfully evaluated the TDMA algorithm printed any changes in the assigned slot as well as the time the slot was changed. When a predicate was evaluated on a node the time it was evaluated as well as the result was printed. Analysis scripts then evaluated the predicate using the most recent slot value from before or up to when the predicate was evaluated to evaluate the predicate itself. This result was then compared with the actual result of the predicate. The results were compared whether or not the predicate response message reached the sink.

We ran these experiments using two different predicates, one that required 1-hop information and one that required 2-hop information. Both were checking to see if there were slot collisions and were the same two predicate that were mentioned as examples in subsubsection 3.2.2. Below the code and logic for the two predicates is reproduced.

\[\text{[all]}\]
\begin{verbatim}
function 1 as slot returning int in using Neighbours(2) as twohopn in
\end{verbatim}

\begin{verbatim}
@\(x : \text{twohopn} \rightarrow \text{slot}(x) \neq \text{slot}(\text{this})\)\n\end{verbatim}

\begin{verbatim}
\forall n \in \text{Nodes}: 
\forall n' \in \text{Neighbours}(n, 2).
\text{slot}(n) \neq \text{slot}(n')
\end{verbatim}

Figure 40: Check that no two neighbours have the same slot (2-hop information)

\[\text{[all]}\]
\begin{verbatim}
function 0 as addr returning int in
function 1 as slot returning int in using Neighbours(1) as onehopn in
\end{verbatim}

\begin{verbatim}
@\(a : \text{onehopn} \rightarrow \text{addr}(a) \neq \text{addr}(\text{this})\)
\end{verbatim}

\begin{verbatim}
@\(b : \text{onehopn} \rightarrow \text{slot}(a) \neq \text{slot}(\text{this})\)
\end{verbatim}

\begin{verbatim}
\forall n \in \text{Nodes}: 
\forall n' \in \text{Neighbours}(n, 1) \cup \{n\}.
\forall n'' \in \text{Neighbours}(n, 1) \cup \{n\}.
\text{addr}(n') \neq \text{addr}(n'') \Rightarrow \text{slot}(n') \neq \text{slot}(n'')
\end{verbatim}

Figure 41: Check that no two neighbours have the same slot (1-hop information)

\text{\footnotesize\url{http://contiki.sourceforge.net/docs/2.6/a00357_source.html}}
5.2 Analysis

![Graphs showing analysis results](image)

Figure 42: Results when predicate period is every 4.0 minutes using a 1-hop predicate

The first point of note from the data presented in Figure 42 is that both of the global predicate evaluation algorithms (PEGE and PEGP) show a 100% delivery rate. While this may initially appear very impressive, this simply arises from the facts that these predicates are evaluated at the sink, and that the graph shows the receipt of evaluated predicates sent from the evaluation location to the sink – as no messages need to be sent, it is vacuously true that no messages will fail to arrive. The two local predicate checking algorithms, however, showed a more predictable trend of successfully delivering fewer responses as the network size increased, though the specific values of these rates are disappointingly low. The values suggest that only nodes in the immediate neighbourhood of the sink are being successful in sending the results of their predicate evaluations. As the network grows in size, the proportion of nodes within this range of the sink decreases, resulting in the trend shown. We believe that this result is due to the way the mesh protocol provided by Contiki works, as it first needs to flood the network with a route discovery message and then waits for a route reply from the target node. However, due to the high amount of traffic it is likely that either or both of these messages are simply being lost. A solution to this would be to have a setup phase, during which every node discovers their respective paths to the sink, before predicate evaluation (or any other main algorithm that is being executed) begins.

Graph (c) showing the rate of correct predicate evaluation, reveals much more useful information: locally-evaluated event-triggered predicates (PELE) show significantly higher accuracy than each of the other algorithms. Global event-based predicates (PEGE) were shown to have the second-highest accuracy, though this accuracy was only slightly higher than that of the periodic counterpart so it cannot be concluded that an event-based approach is universally better (within the scope of this metric). There may be a connection between PELE having the highest number of received messages (graph (a)) and its superior accuracy in evaluation; the premise being that more messages being received could give a node more data to use when evaluating a given predicate, in turn making it more likely to return a correct response. This notion of higher message delivery rates for PELE is somewhat corroborated by the number of transmissions shown in (b) – PELE is responsible for the second lowest number messages sent. However, the cause of PELE alone having a significantly higher success rate in delivering messages is as yet unknown, so more definitive information on this front would require further scrutiny using testing methods that may well prove infeasible in a resource-constrained system.
It is also good to note that as the network size increases the correctness of predicate evaluation remains effectively constant, with only small fluctuations. From this we can say that the predicate evaluation libraries are highly scalable with respect to the size of the network and the accurate evaluation of the predicate. However, as has been mentioned before they scale poorly with respect to the number of responses the reach the sink.

As messages sent and received use the most energy in a WSN system we will use message transmit and receive statistics to evaluate the energy usage of the algorithms. In graphs (a) and (b), local periodic predicate evaluation is the most conservative, however it also shows the lowest accuracy for its evaluations so these results show little in the way of benefits to using this algorithm. By contrast, the most accurate algorithm, PELE, has a higher (yet still moderate) level of energy consumption. An important observation is that the energy demands of global predicate evaluation – both of which showed middling accuracy – increase faster than those of local evaluation as network size increases. This is because global evaluation requires data from the entire network, and the operating of the mesh routing protocol means that nodes lying further from the sink will have to have their messages forwarded by a greater number of intermediate nodes, giving exponential growth in the number of messages sent. The local evaluation algorithms show a more linear trend as the size of a node’s 1- or 2-hop neighbourhood may not increase due to the presence of more nodes in the network – all that is guaranteed is that there will be more such neighbourhoods in which to evaluate predicates.

![Graphs showing energy consumption and accuracy of different evaluation methods.](image)

Figure 43: Results when predicate period is every 2.0 minutes using a 1-hop predicate

At first glance of figure the results seem to mirror those of figure however there is a key difference, in that the performance of the global periodic-based implementation had a lower success rate, when the predicate period is 2.0 minutes. This is likely due to the fact that messages are sent more often, but the delays in the aggregations tree remains the same. Therefore, it is likely that the aggregation tree is affected how the predicates are evaluated by limiting how fast data from the edges of the network can reach the base station for evaluation. It is possible that the aggregation tree would be tweaked to decrease its wait period before sending messages onwards to increase the accuracy of evaluation, however, this would likely increase energy usage further.

The graphs also show that there were a larger number of messages both sent and received in these tests (figures and ), although this can be attributed to the fact the predicate period is more often, requiring more messages to be passed around the network. Comparing the two sets of data we can see there were similar performance levels, regardless of the predicate period length.
It can also be suggested that increasing the predicate period would increase the evaluation success rate, as there would be fewer collisions of messages in the network. Of course, based on the above reasoning, increasing the period would also decrease the number of messages sent by the periodic implementations and would thus extend battery life.

![Graphs showing various metrics such as messages received, messages sent, percentage correctly evaluated, and percentage of responses received for different network sizes and predicate periods.](image)

**Figure 44:** Results when predicate period is every 4.0 minutes using a 2-hop predicate

The previous two sets of results in [Figure 42](#) and [Figure 43](#) were for the 1-hop variant of the TDMA slot collision predicate. In [Figure 44](#), results for the alternative 2-hop predicate are shown. There are a number of patterns that differ and some that remain the same.

To begin with we observe the same behaviour with respect to the delivery ratio of failure response messages in [Figure 44(d)](#). Predicates that are evaluated at the root node of the tree continue to have a 100% delivery rate, while nodes evaluating in-network continue to have poor delivery rates that decrease in larger networks. The issue here is the same as the problem encountered with the 1-hop predicate results, so the same solution proposed would apply here.

Next, we have very different patterns of number of messages sent in [Figure 44(b)](#). We find that PELE requires the largest number of transmissions and PELP requires the fewest, with both growing linearly with respect to network size, whereas the global predicate evaluators use energy between these two levels and there number of transmissions increase faster than linearly. The difference here is that it is no longer the case that global evaluators require the most number of message to be sent. In fact the global evaluators use almost the same number of messages when comparing the 2-hop predicate results to the 1-hop predicate results. This is because the structure of sending messages to a node for predicate evaluation is the same in both 1-hop and 2-hop predicate evaluation, it is via an aggregation tree. The local in-network predicates are where large increases in messages sent occur. These increases are down to the fact that the neighbourhood of information that needs to be disseminated has increased from 1-hop to 2-hops. We see that this affects PELE more than it does PELP, for example for size 15 networks both local predicate evaluators transmit about 1000 message when evaluating the 1-hop predicate, whereas when evaluating the 2-hop predicate PELP sends about 1500 messages and PELE sends 2500 messages. So it when increasing the size of data that needs to be requested our event-based predicate evaluator will send more messages.

There is also a very interesting difference with respect to the correctness of predicate evaluation. When evaluating the 1-hop predicate the local in-network event-based evaluator performed best. However, for the 2-hop the local periodic evaluator performed best, we would expect that usually
this would have been caused by the increase in message transmissions leading to a busier network with more collisions and message lost. Except that as the network size increases the number of messages transmitted increases, but the percentage of predicates that were correctly evaluated remains the same, so it would initially appear that collisions do not factor in. On the other hand, because these predicates are evaluated in-network they are by definition local, so increases in messages transmitted across the whole network may not paint an accurate picture. What we believe to be the case is that, compared to the 1-hop predicate, the 2-hop predicate has more local traffic and thus more local collisions which lead to event-based in-network evaluators to perform badly as lots of redundant data is transmitted. However, with in-network periodic evaluators data is only transmitted when it is asked for. So while event-based data sending may have helped for 1-hop predicates with respect to the greater number of transmissions, due to the increased number of hops data needs to travel to evaluate 2-hop predicate sending data when it changes actually leads to more local collisions leading to less current data reaching nodes, causing event-based evaluators to perform worse for 2-hop predicates than 1-hop predicates.

Another point is that for 2-hop evaluation the percentage of predicates correctly evaluated is generally lower than when evaluating 1-hop predicates. This is likely caused by the fact that more data is required, so there is a greater chance for more of it to be out-of-date by the time it reaches the evaluating node. There is also a greater chance for not all the data to reach the node evaluating the predicate (as there is simply more data being transmitted). Either of these two issues could have lead to the overall decrease in accuracy of evaluating the predicate.

5.3 Conclusion

To conclude it appears that deciding on which implementation to use when evaluating predicates depends on the size of the network and the predicate being evaluated. For predicates that involve no neighbour information or 1-hop neighbour information, evaluating them in network using PELE or PELP is preferred. PELE should most likely be used if data changes infrequently. We predict there will come a point where periodic sending of data will be better as the excess messages generated by frequent changes will lead to higher energy usage and more collisions (lower successful evaluations).

For predicates that use 2-hop information it would appear that using PELP is preferable due to the lower energy usage and high chance of a successful evaluation. We predict that as the size of the neighbour information increases, it will become more preferable to use global predicate evaluators.

For small networks there seems to be little difference between local and global predicate evaluators. However, as the network size increases it becomes less viable to use global evaluators due to the increase in energy.

So overall, deciding on where to evaluate a predicate is still a non-obvious decision. It depends on many factors, and these solutions still have the potential to be further optimised towards certain criteria.
6 Practical Experience

As part of this project we learnt a number of things that we feel would be useful to impart on anyone else that undertakes development of wireless sensor networks with Contiki and physical hardware. Many of the issues we encountered were simply down to poor documentation or it being located in difficult to find places. We hope that the information included in this section is helpful.

6.1 Contiki Logical Channels

One of the first confusing aspects of Contiki is that when creating connections you need to provide a channel number. Initially we believed that this channel caused the messages being sent by that connection to be broadcasted on a different frequency to. However, after some research we found that the channels are in fact virtual. By virtual we mean that the same radio frequency is used to broadcast and receive all messages, but when when a packet is received it is funnelled to the correct receiver by way of the channel included in the packet. If one wishes to change the frequency of the radio, the function `cc2420_set_channel` should be used. There are many other functions capable of changing settings in the cc2420 radio header\textsuperscript{13}. Another example API is `cc2420_set_txpower` which allows the transmission power of the radio to be set.

6.2 Power Levels

As mentioned in the previous section there exists an API that can be used to set the transmission power of the radio (`cc2420_set_txpower`). We experimented using the hardware in the Computer Science building at the University of Warwick at different power levels. This was done because ideally we wished to experiment deploying the applications to the hardware and then testing them in a real world environment. The reason that we needed to investigate power levels was so that we could control the neighbourhood of motes to be a small space, otherwise we would have been very spread out through the building making it difficult to update motes as necessary. Unfortunately at the low transmission power that would allow nodes to communicate when within 15cm of one another but not when further away, we found that very few messages were received by nodes. We believe that this was caused by the environment that we were testing in (DCS building), as we expect the traffic on the 2.4GHz frequency to be very busy due to the number of phones and wireless devices in the department. Therefore, we recommend to do as many other who investigate sensor networks do, and test in remote areas away from interference from commonly used devices\textsuperscript{[21]}.

6.3 Sensor Data Conversion

The sensors attached to our hardware did not provide results in a binary format such as an integer or a float. This meant that we had to convert from their custom binary representation to the binary representation desired. Equations were provided by the manufacturer to perform these conversions\textsuperscript{[7]}, however, constants in those equations are heavily dependant on the specifications of the sensor (such as the voltage levels and the number of bits of data the sensor reports). So we recommend making sure that the correct constants are used by comparing the results of the sensor to a known accurate measuring device - such as a thermometer.

6.4 Uploading to the motes

When uploading the code to the motes there is a good amount of documentation on how to do so. However, there are a certain number of other steps that need to be taken. Below is the line of code that needs to be entered into the terminal. This will make the code and it is important that a `make clean` has been run on the command line before hand. The `.upload` at the end of

\textsuperscript{[13]}\texttt{core/dev/cc2420.h} \url{http://contiki.sourceforge.net/docs/2.6/a00158_source.html}
the project name informs the name script to upload to the motes and the \texttt{MOTES} variable is used to locate the USB port the mote is connected to.

\texttt{sudo make example-unicast.upload DEFINES=DUMMY,NODE_ID=1 MOTES=/dev/ttyUSB0}

\textbf{Listing 1:} Command to upload firmware to a mote connected to a USB port

What was missing from the Contiki documentation is the importance and difference between the nodes MAC address and its Rime address. Without the following addition to the make script the MAC address of the node would not be set and the Rime address would be set to some random value. We found this difficult to debug because when loading the code into the simulator Cooja a different MAC layer is used that correctly sets up the MAC addresses. However, when running on physical hardware it is necessary for the uploader to specify the MAC address every time the firmware is uploaded. The MAC address specified will then cause the Rime address to be set to their sane counterparts of the MAC address, instead of random values.

We set the MAC address by passing some defines to the C program. We have to pass the \texttt{DUMMY} define because otherwise make would have trouble understanding the command. The define \texttt{NODE\_ID} should contain the MAC address of the node that is being uploaded to. This macro will then be handled in the source could of the application by the code given below. Without the code the MAC address will not be set. Also note that the header \texttt{sys/node\_id.h} will need to be included to provide the \texttt{node\_id\_burn} function.

\begin{verbatim}
#ifdef NODE_ID
node_id_burn(NODE_ID);
#endif
\end{verbatim}

\textbf{Listing 2:} Code that needs to be inserted into the startup process of an application

We noticed this problem because regular broadcasts would work between the motes, however, unicasts would all fail. We believe that the message was reaching the nodes correctly, but because the MAC address was default initialised to 0 it meant that the MAC address was never equal to the target address, thus the messages were never delivered.

\section{6.5 Communication Between Mote and Computer}

A very important feature that we required was the interface between an application running on the mote and an application on a desktop computer that said mote was connected to via some form of serial collection (i.e. USB). Getting data to flow from the mote to the desktop application was trivial as Contiki provided the C function \texttt{printf} which wrote to the serial output and a script called \texttt{serialdump-linux} which was wrapped in Java\textsuperscript{14} to access that output on the desktop.

Unfortunately Contiki does not implement many of the C APIs to read from \texttt{stdin}. Contiki instead has an alternative mechanism in which it buffers lines and when a newline is detected an event is sent to all processes informing them that a line has been delivered\textsuperscript{15}. This line length limitation makes designing the communication protocol from the desktop application to the mote application tricky as no more than 127 bytes can ever be sent in a single line. This means that on the mote application parsing the input can become quite intricate and difficult, we ended up coding our own automaton\textsuperscript{10} based on the first character of the line. Regarding the desktop application it can send lines of data to the mote using \texttt{serialdump-linux} if a line is sent to that tool’s \texttt{stdin} it will be forwarded to the mote.

\subsection{6.5.1 Setting up serial2pty Cooja Plugin}

When testing applications that involve serial communication between a desktop application and a real mote, it is not always possible or convenient to have the mote to hand. Therefore, we needed a way to expose one of the simulated motes in Cooja like we have done for real motes. To do this

\textsuperscript{14}http://ai.vub.ac.be/Robotics/wiki/index.php/Compiling,_uploading_and_interacting_with_Contiki:_Hello_World
\textsuperscript{15}https://github.com/contiki-os/contiki/wiki/Input-and-output#wiki-Serial_Communication
we used a plugin called serial2pty, which simply connects the mote in Cooja to a pseudo-terminal which can have data written to it and read from it.

```
# First build Cooja
cd ~/contiki/tools/cooja/apps
git clone git://i4git.informatik.uni-erlangen.de/contiki_projects.git -b serial2pty serial2pty
cd serial2pty
ant jar
```

Listing 3: Setting up and compiling the serial2pty Cooja plugin

Once you have followed the instructions in Listing 3 you can load this plugin in Cooja by going to Settings -> Cooja Extensions and selecting serial2pty. Once it has been selected click Apply for session and Save and the plugin will now be enabled. To run it simply click Tools -> Serial 2 Pty -> and select the node you want to connect. You will be told what device you can connect to, to receive the serial output from.

Initially we found that this tool would not build, so a patch was created to fix those issues, sent to the developer and it was integrated into their repository.

### 6.6 Cooja Memory Usage and Options

We found that Cooja’s default settings when run using ant run were suitable for smaller networks. However, when running more memory intensive applications on the simulator of the hardware in Cooja or when running larger networks, Cooja itself can run out of memory. The simple solution is to simply run Cooja using ant run_bigmem which supplies a flag limiting the maximum of memory to about 1536MB instead of 512MB when running normally.

As Cooja is written in Java it means that all of Java’s configuration flags are also up for modification. This however, means that the ant build script at ~/contiki/tools/cooja/build.xml will need to be modified to include the flags that need to be passed to Java. We modified the build script to add the following arguments (-XX:+OptimizeStringConcat -XX:UseSSE=3 -XX:+UseFastAccessorMethods -XX:+UseLargePages) but did not see much improvement of performance.

Finally, there is one other useful mode, and that is the ability to run Cooja without a GUI. This is done by changing the run command to ant run_nogui. Running in this mode is very useful when running many simulations to obtain results from them as fast as possible without the overhead of running the GUI.

### 6.7 Static variables

When developing libraries make sure static variables are used very carefully. For example, we found that commonly we would make callback timers static and then use that object. When the code using that timer will only be called from one place it is okay. However, when you have multiple calls to that timer (imagine broadcasting on different channels) this can cause race conditions as the memory for that timer is being shared. What should be done is the timer object should be placed in a struct and that struct should be passed to the relevant functions that need to access the timer. For each time you wish to use the library a pointer to a different struct should be passed to those functions.

However, there are times that declaring a variable as static is very important. One of these cases is when you have a process that it waiting on some event, if you want a variable to maintain the same value after the event has been waited on, then that variable needs to be static. In Listing 4 the printf statement prints out a static and non-static variable. The static counter will increase and be reliably printed out. However, when the variable x is printed, the value it contains at that point could be anything. This is due to the way that Contiki’s process are actually protothreads, so share a stack with other protothreads, if another process were to be scheduled at this point

16http://i4git.cs.fau.de/contiki_projects.git/commit/2435119800de6b233b2cddb506fadd0f8b64bb881
it could potentially modify the stack. This would lead to stack variables such as \texttt{x} potentially being changed. On the other hand, if there are no points in the program where the process may yield to other threads, then variables can be non-static. For example \texttt{y}'s value would be printed correctly every time.

\begin{verbatim}
PROCESS_THREAD(proc, ev, data)
{
    static struct etimer et;
    static unsigned int counter = 0;

    PROCESS_BEGIN();

    while (true)
    {
        unsigned int x = 1234;
        etimer_set(&et, 10 * CLOCK_SECOND);
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));

        unsigned int y = 2;
        printf("%u %u %u\n", counter, x, y);
        ++counter;
    }

    PROCESS_END();
}
\end{verbatim}

Listing 4: Contiki process static variables

6.8 Firmware Size

Reducing firmware size is a very important issue for WSN applications. As there is a small amount of space for firmware (48KB for our hardware, see \cite{Appendix A} the smaller the compiled code the more functionality that can be included in it. The Contiki wiki has a number of techniques that they have included in Contiki to optimise the code size. As these are already done, to reduce code size further it is the responsibility of the system developer to reduce the code size. The following are a number of techniques that were used to reduce the firmware size of the code that was developed:

1. Remove as many static strings as possible. Static strings take up lots of space in the ROM and are only helpful for debugging. Once debugging is finished these strings should be removed.

2. Remove as many unneeded function calls as possible.

   - printf - This is linked to removing static strings
   - memset - If you memset memory to a set value after allocating it then remove the memset call and just fill in the data structure as is appropriate
   - memcpy - If possible copy the data by hand (e.g. by a simple assignment for primitives) to save on a function call
   - Avoid string functions and other C library functions. If you do not use them it is possible that the object they are contained within will not be linked and thus space will be saved. \cite{114}

The most important thing is to make small changes at a time and then test every change that is made. This way the impact of the change on the program's size can be observed and if it made the size larger, then the change can be reverted. Doing multiple changes at once, may be faster to code, but may not lead to the desired result quicker.

\cite{114} https://github.com/contiki-os/contiki/wiki/Reducing-Contiki-OS-firmware-size
On a similar note, GCC has recently released a new type of optimisation that runs over the entire program instead of individual compilation units called LTO (Link-Time Optimisation) \[24\]. With this it should be possible for more dead code to be removed or for more size optimisations to be applied across compilation units. Unfortunately the developers of the compiler being used (MSP430) seem unwilling to support this feature\[18\].

6.9 Memory Management

When developing applications in C it is almost a given that memory management problems would be encountered. This is an inherent issue when developing with C, due to the low level control the developers are given it is very easy to “shoot yourself in the foot” (Stroustrup \[105\]), which leads us back to our initial point as mentioned in the introduction, that without good tools the quality of the system being developed will be poor. What is now clear to us is that C is not a good language for developing these kinds of applications. It would appear that developing a new language should be a priority for wireless sensor network developers. An example of this could be the design of C++ with respect to C, which added many good safety features and higher level constructs. \[2\] proposes a high level language that allows the developers to specify what they want to happen rather than how they would like it to happen.

There has been work on developing virtual machines not just for limited purposes (as was ours), but to run the entire application \[76, 82\]. With these virtual machines there are several advantages, the first is that experimenting with higher level languages becomes easier as they will all compile down to the same bytecode and the second is that memory management can be taken out of the hand of the programmer and put in control of the compiler. The problem here is that the main reason why that C is a popular language for developing sensor network applications is the control over memory that is provided. Due to the low resources of the motes, developers must be very careful as to how they allocate memory. So perhaps developments in low powered memory to allow more of it or improvements to how virtual machines can allocate memory efficiently will be required before these solutions become feasible.

6.10 Conclusion

Overall we found that developing sensor network applications using simulators and tools that meant the application could be deployed in the real world was much harder than we anticipated. We often got sidetracked from our intended task because there was a new issue facing us. It is also unfortunate that the documentation for some of these issues is very poor. We hope that this section provides a useful guide for anyone else that undertakes wireless sensor network development with little or no previous experience.

\[18\] http://comments.gmane.org/gmane.comp.hardware.texas-instruments.msp430/gcc.user/11006
7 Project Management

7.1 Methodology

The requirements of the project were not fully understood at the beginning of it and we were unsure of the efficiency of our algorithms we were developing, therefore, we used a prototyping approach as part of an evolutionary process model \cite[p. 42–44]{92}. This allowed us to start with a set of core requirements and then incorporate other extensions as the requirements are better understood. We had meetings with our supervisor to discuss the best ways to implement our system and any what additional components we should start developing. When a new idea was defined, a prototype was modelled, constructed and tested. For some parts of the project we sought out our supervisors advice as to how to implement it or what new approach we could try to take. The process was repeated until we had a sufficiently stable base of libraries and the intended code developed.

Term 1

We spent the first few weeks researching on the different aspects of wireless sensor networks as well as the appropriate tools and platform to develop our system with. Much of this involved installing and setting up the software and then developing simple applications with it to learn how the system works and how to develop for it. Much of the group was learning this from scratch, making this a slow and difficult part of the project. We focused on developing a single algorithm (PELP) and the framework that required, with the intent to extend it later. Due to the large number of algorithms we intend to implement, we made a list of such algorithms and implemented the most relevant ones first. As we progressed through the development, the list was updated by removing or adding algorithms that had been implemented or still needed to be implemented.

Term 2

At this stage, most of the algorithms and protocols were implemented independently and treated as individual components to the core system. We decided to do it this way because of the benefits of modularity where the principles of “Separation of Concerns” \cite[p. 99]{92} allows for a more manageable set of modules. The composition of these components such as predicate checking and neighbour data retrieval was done during the entire winter vacation and the beginning of the second term.

In the second term, we continued following the evolutionary process model but at a more relaxed approach which focuses on finishing the system components and preparing the gathering of results. Since the project scope was clear and we had a more directed goal, fewer meetings with the supervisor was needed as we simply needed to implement modules. We still had meetings with our supervisor, however, they were less frequent than in the first term. Ensuring that the development and debugging phase could be completes in time required the effective scheduling of tasks and prioritising work. Unfortunately, by the second half of the term, progress had fallen to a halt due the high number of coursework deadlines and this meant that team members where preoccupied with other duties. To address the issue, a meeting was held prior to the halt where we discussed our recovery plan and actions to take once members was free. This allowed team members to continue working where they left off and avoid any delays. However, it was not until during the Easter holiday that the project gained the momentum to finish development and results gathering.

7.2 Role Allocation

We decided to allocate roles in the second week after we had the first week to perform research into the problem and find out what has been done. The following were how we assigned roles, although we intend for these to be flexible:
<table>
<thead>
<tr>
<th>Name</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matthew Bradbury</td>
<td>Group Leader</td>
</tr>
<tr>
<td>Daniel Robertson</td>
<td>Project Manager</td>
</tr>
<tr>
<td>Amit Shah</td>
<td>Technical Leader</td>
</tr>
<tr>
<td>Ivan Leong</td>
<td>Developer and Tester</td>
</tr>
<tr>
<td>Joe Yarnall</td>
<td>Developer and Tester</td>
</tr>
<tr>
<td>Tim Law</td>
<td>Developer and Researcher</td>
</tr>
</tbody>
</table>

Whilst the entire team was responsible for system design and development, it was critical to have a group leader and other leading roles to facilitate and provide resources to the team. Their individual responsibilities were as follows:

**Matthew Bradbury - Group Leader**

His main task was to lead the entire project and have a clear understanding of the components that is to be implemented. He is responsible for allocating tasks to its team members and providing direction for meeting the project objectives. He also possesses strong communication skills with his team members as well as being able to explain the concepts to the supervisor concisely. Ensures that everyone is performing their roles accordingly and providing motivation when the project is slowing down.

**Daniel Robertson - Project Manager**

He supported the group leader by facilitating the management of the project and ensuring that the overall work is done on schedule. He arranges weekly meetings and provides a summary report prior to each meeting and records all discussions.

**Amit Shah - Technical Leader**

Responsible for overseeing the work done by other members and ensures that all code written by the team meets the technical specification and design requirements of the project. Also responsible for looking after the hardware and the development with it.

### 7.3 Project Planning

Our initial planning was done in the first few group meetings and most of the plan details was explained in the project specification. As we progressed through the project development, we encountered many changes from the initial plan as well as changes in task schedule. This section presents the final overall plan of the project including the revised time schedule of development activities, resources used and issues encountered.

#### 7.3.1 Work Breakdown Structure

The figure below shows the work breakdown structure as designed for the project specification.
7.3.2 Deliverables

Our project consists of 5 deliverables which includes one individual report. It was very important that these milestones were met because it allows us to monitor our progress and prove that we were working in the direction of the stated goals and requirements of the project.

1. Specification (*Due Week 4 - Term 1*)

   This was the first deliverable of the project. At the time, we were still undergoing research on the different components of the system and the project scope was not clearly defined. Nevertheless, we highlighted in the specification the main goals of our project and the different algorithms that we wished to implement. We also included a schedule of all the planned activities for the remainder of the project and an approximate time taken to complete them.

2. Progress Poster Presentation (*Due Week 10 - Term 1*)

   By the end of the first term, the scope of the project had reached a more defined level as we had decided what is to be implemented and what has been disregarded from the provisional list of algorithms. This was demonstrated in our first delivery - the poster presentation, where we had to chance to explain our project at a high level to our supervisors and other professors.

3. Final Group Report (*Due Week 1 - Term 3*)

   Although the report is due in term three we started adding content since the first term. This included the introduction, literature review, a list of algorithms and protocols to be
implemented and a brief description of their functions. In the beginning of the second term, much of the algorithms were described in the report and it was only towards the end of the term that the report was being updated on a full scale. Each member was assign a specific section to work on but with the flexibility to update other sections if it was related to them.

4. Individual Report (Due Week 1 - Term 3)

Every team member needs to write a 3000 words report to reflect on their performance and contribution towards the project as an individual.

5. Final Project Presentation (Due Week 3 - Term 3)

Demonstrate our final project to a panel of judges and our supervisor.

### 7.3.3 Schedule

Throughout the term, every member was assigned specific tasks and was given a time frame to complete them. The table below shows the time schedule allocated to each members.

<table>
<thead>
<tr>
<th>Task Description</th>
<th>Time Allocated (Weeks)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Term 1 - Developing for application predicate checking</strong></td>
<td></td>
</tr>
<tr>
<td>Research around the Problem</td>
<td>2 2 2 2 2 2</td>
</tr>
<tr>
<td>Writing Specification</td>
<td>1 1 1 1 1 1</td>
</tr>
<tr>
<td>H-SEND Implementation</td>
<td>3.5 3.5</td>
</tr>
<tr>
<td>“Send to Base” Implementation</td>
<td>2 1.5</td>
</tr>
<tr>
<td>Clustering Implementation</td>
<td>3.5 2</td>
</tr>
<tr>
<td>Aggregation Tree Implementation</td>
<td></td>
</tr>
<tr>
<td>Develop Visualisation Tool</td>
<td>1.5</td>
</tr>
<tr>
<td>Develop Predicate Language Runtime</td>
<td></td>
</tr>
<tr>
<td>Testing and Adapting to Physical Nodes</td>
<td>2 2 2 2 2 2</td>
</tr>
<tr>
<td>Message Logging</td>
<td>1</td>
</tr>
<tr>
<td>Poster Creation and Presentation preparation</td>
<td>1.5 1.5 1.5 1.5 1.5 1.5</td>
</tr>
<tr>
<td><strong>Term 2 - Developing for network predicate checking and where the predicate is checked</strong></td>
<td></td>
</tr>
<tr>
<td>Additional Research</td>
<td>1 1 1 1 1 1</td>
</tr>
<tr>
<td>Improving Dynamic Predicate Specification</td>
<td>2 2</td>
</tr>
<tr>
<td>Develop Visualisation Tool</td>
<td>2 2 2 2 2 2</td>
</tr>
<tr>
<td>Modify Algorithms to Selectively Evaluate Predicates</td>
<td>2 2 2 2 2 2</td>
</tr>
<tr>
<td>Performance Testing</td>
<td>1.5 1.5 1.5 1.5 1.5 1.5</td>
</tr>
<tr>
<td>Testing and Adapting to Physical Nodes</td>
<td>1.5 1.5 1.5 1.5 1.5 1.5</td>
</tr>
<tr>
<td>Report Writing</td>
<td>2 2 2 2 2 2</td>
</tr>
</tbody>
</table>

### 7.3.4 Gantt Chart

The project plan was adhered to as much as possible and was closely monitored as shown in the Gantt chart below. The Gantt chart was edited several times and accounted for revisions made whenever there was a change in task or time schedule. The colours of the bars are as follows:

- Green: Research and learning
- Orange: Design
- Blue: Implementation
- Yellow: Specification and report preparation
- Red: Testing
<table>
<thead>
<tr>
<th>#</th>
<th>Task Name</th>
<th>Duration</th>
<th>Start</th>
<th>Finish</th>
<th>Predictor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Research &amp; Learning</td>
<td>3.13 wks</td>
<td>Mon 01/10/12</td>
<td>Sat 20/10/12</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>OS and simulation</td>
<td>2 wks</td>
<td>Mon 01/10/12</td>
<td>Sat 13/10/12</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Algorithms and MAC Protocols</td>
<td>3 wks</td>
<td>Mon 01/10/12</td>
<td>Sat 20/10/12</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Predicate checking</td>
<td>2 wks</td>
<td>Mon 08/10/12</td>
<td>Sat 20/10/12</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Writing specification</td>
<td>1 wk</td>
<td>Fri 13/10/12</td>
<td>Thu 25/10/12</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Specification Submission</td>
<td>0 days</td>
<td>Thu 25/10/12</td>
<td>Thu 25/10/12</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Poster creation</td>
<td>1.5 wks</td>
<td>Tue 27/10/12</td>
<td>Thu 06/11/12</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Progress poster presentation</td>
<td>0 days</td>
<td>Thu 06/11/12</td>
<td>Thu 06/11/12</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Design</td>
<td>3 wks</td>
<td>Mon 01/10/12</td>
<td>Sat 20/10/12</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Algorithms, predicate and GUI</td>
<td>3 wks</td>
<td>Mon 01/10/12</td>
<td>Sat 20/10/12</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Implementation (Term 1 + Winter)</td>
<td>3.67 wks</td>
<td>Mon 22/09/12</td>
<td>Sun 06/10/13</td>
<td>9</td>
</tr>
<tr>
<td>12</td>
<td>Algorithms 1</td>
<td>6 wks</td>
<td>Mon 22/10/12</td>
<td>Thu 20/12/12</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>Predicate language runtime</td>
<td>4 wks</td>
<td>Mon 05/11/12</td>
<td>Sat 01/12/12</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>GUI</td>
<td>9 wks</td>
<td>Mon 05/11/12</td>
<td>Thu 03/12/12</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Network traffic logging and reporting</td>
<td>2 wks</td>
<td>Mon 19/11/12</td>
<td>Sat 01/12/12</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Writing-report</td>
<td>22.50 wks</td>
<td>Thu 18/12/12</td>
<td>Thu 18/04/13</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Final report</td>
<td>0 days</td>
<td>Fri 26/04/13</td>
<td>Fri 26/04/13</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Implementation (Term 2)</td>
<td>10.42 wks</td>
<td>Mon 03/06/13</td>
<td>Sat 16/06/13</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Algorithms 2</td>
<td>10.42 wks</td>
<td>Mon 03/06/13</td>
<td>Sat 16/06/13</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Integrate predicate and algorithms 1</td>
<td>3 wks</td>
<td>Mon 07/01/13</td>
<td>Sat 26/01/13</td>
<td>13, 12</td>
</tr>
<tr>
<td>21</td>
<td>Visualization tool and interface</td>
<td>5 wks</td>
<td>Mon 07/01/13</td>
<td>Fri 08/02/13</td>
<td>14</td>
</tr>
<tr>
<td>22</td>
<td>Testing software on hardware</td>
<td>2 wks</td>
<td>Mon 21/02/13</td>
<td>Sat 20/03/13</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Testing and Data Gathering</td>
<td>11.46 wks</td>
<td>Mon 28/01/13</td>
<td>Sat 13/04/13</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Optimization and algorithm testing</td>
<td>7 wks</td>
<td>Mon 28/01/13</td>
<td>Fri 15/02/13</td>
<td>20</td>
</tr>
<tr>
<td>25</td>
<td>Performance testing</td>
<td>5 wks</td>
<td>Mon 28/01/13</td>
<td>Fri 15/02/13</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Data gathering</td>
<td>4 wks</td>
<td>Mon 28/01/13</td>
<td>Sat 13/04/13</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Final presentation</td>
<td>0 days</td>
<td>Tue 07/05/13</td>
<td>Tue 07/05/13</td>
<td></td>
</tr>
</tbody>
</table>
7.4 Resources

7.4.1 Source Control Management

We signed up for a Git repository on BitBucket \[12\] where we committed all the work we produced. We initially had an issue that free private repositories hosted on BitBucket have a maximum of 5 participants, whereas we had 6 group members. Fortunately when new users sign up to the services from an invite, the person that sends the invite gets additional capacity. This meant that the person who created the repository ended up with enough capacity for all members to access the account. The use of BitBucket also gives the advantage of branching and merging codes from different users without needing to replicate any files and worry about redundancy.

7.4.2 Communication

When group members are working online, the main source of communication is via the popular social networking platform - Facebook. We created a group for our team where members could post any messages regarding the project such as issues in code development, assignment of tasks to team members and reminders of any project activities (e.g. weekly meetings).

7.4.3 Development tools

The following table summarizes all the software and hardware used to develop the wireless sensor network application.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Constraints/Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contiki</td>
<td>Operating System for WSNs</td>
<td></td>
</tr>
<tr>
<td>Cooja</td>
<td>Contiki Simulator</td>
<td></td>
</tr>
<tr>
<td>Bash and Python</td>
<td>Scripting numerous tasks</td>
<td>Some dependencies had to be manually installed</td>
</tr>
<tr>
<td>Make</td>
<td>To compile the C source code for the project</td>
<td></td>
</tr>
<tr>
<td>MSP-GCC</td>
<td>The custom compiler used</td>
<td>Did not come installed on DCS computers, needed to compile ourselves</td>
</tr>
<tr>
<td>Netbeans and Java</td>
<td>To develop the GUI</td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sensor nodes</td>
<td>See Appendix A</td>
<td>Provided by department, needed to share with PhD student who also required them</td>
</tr>
<tr>
<td>USB Hub and USB cables</td>
<td>To work with multiple sensor nodes at once</td>
<td>Hub required purchasing, cables were borrowed from DCS</td>
</tr>
</tbody>
</table>

7.5 Risk Analysis

It is almost certain that every project contains risks which could impact its progress and success if not well managed. Risk management involves two basic steps, 1) to identify the different possible risks that may occur in the project and its likelihood of occurrence and 2) state the actions to mitigate the effects of any threats. Table 6 contains a number of risks that we considered and how we planned to mitigate their effect. We also ranked them by severity and likelihood so we know which risks we needed to prioritise mitigation and which we needed to be on the watch for, where 1 represents the least severe or least likely and 10 represents the most severe or most likely.
Risk | Likelihood | Severity | Mitigation
--- | --- | --- | ---
Personal Hardware and Software failure (i.e. personal computers)  
- Report and research  
- Software code | 2 | 9 | To prevent the risk of data lost, we used a Bitbucket repository to save and store all our documents and program coding online. This also allows the concurrent development of our project and ensuring that all team members are up to date.

Team member illness and absence | 5 | 4 | There would be circumstances where team members would be unavailable to work on the project for a short period of time. To ensure that work is progressive, necessary measures were taken. For example an absent member can temporary delegate his current task to a different member. Code should also be well documented to allow others to take over work on it.

Meeting deadlines and schedules | 5 | 7 | It was important that every milestone of the project was met so that no marks were penalised. The progress of the project was monitored and we had weekly meetings to ensure that everyone was on track of their given tasks.

Obtaining desired results when deploying code (tested on simulated environment) on physical devices | 5 | 4 | Most of the code that was developed in the beginning was tested with the Cooja simulator. It would be unlikely to obtain similar results when deployed on the actual hardware due to a wide range of environmental factors such as interference and power distribution. An option is to gather data from simulated environment rather than physical devices.

Defining the scope of our project | 6 | 7 | There are many issues and research problems that revolve around the topic of wireless sensor networks. It is important to define the scope of the project to prevent scope creep and necessary expansion of the project. Need to keep track of all new ideas proposed by the supervisor and raise any change requests to the project.

Damage of the sensor nodes | 2 | 8 | Each of the sensor nodes cost 80 Euros. The sensors must be handled with care to prevent any cost of damage. Assign a group member the responsibility to store the sensors after use and keep track of who is using them.

Components of the project that could not be implemented with current tools. | 4 | 4 | Research on alternative methods or implement custom tools.

Table 6: Risk analysis

### 7.6 Work Overview

This section highlights the different tasks that was undertaken by each group member for every week of term 1 and term 2. At every weekly meetings, the project leader would discuss the progress
of the tasks that was assigned in the previous week and any new tasks for the current week would be recorded as shown in the table below.

Term 1

<table>
<thead>
<tr>
<th>Week</th>
<th>Activities</th>
<th>Task Allocation</th>
</tr>
</thead>
</table>
| 1    | 1. Meet up with Supervisor and discuss project direction  
2. Research on related work and what we can develop, before settling on main aims  
3. Investigate two OS and their corresponding simulators: TinyOS with TOSSIM and Contiki with Cooja | Amit: All  
Dan: All  
Ivan: All  
Joe: All  
Matt: All  
Tim: All |
| 2    | 1. Research if Cooja can be extended through the use of plug-ins (with the aim of extending it to replay traffic logs)  
2. Research Clustering algorithms and find implementations  
3. Develop a temperature dissemination application to learn Contiki and Cooja  
4. Research TinyOS and TinyDB and see if they could be applied to predicate checking  
5. Investigate performance of different MAC protocols  
6. Investigate the feasibility of live monitoring of the network  
7. Investigate QoS and how it may be applied to real life WSN deployments  
8. Produce a literature review on chosen topic | Amit: 1, 7, 8  
Dan: 5, 8  
Ivan: 6, 8  
Joe: 4, 8  
Matt: 3, 8  
Tim: 2, 8 |
| 3    | 1. Research DICAS  
2. Research Send to Base  
3. Research Daicon  
4. Research DIDUCE  
5. Research H-SEND  
6. Research Sympathy  
7. Write Specification | Amit: 2, 7  
Dan: 4, 7  
Ivan: 3, 7  
Joe: 1, 7  
Matt: 6, 7  
Tim: 5, 7 |
| 4    | 1. Finish Specification  
2. Work out how to get MSP430-GCC and the Cooja working on Joshua  
3. Begin developing aggregation tree example | Amit: 1, 2  
Dan: 1  
Ivan: 1  
Joe: 1  
Matt: 1, 3  
Tim: 1 |
| 5    | 1. Get used to developing in C and learn how to use the Contiki libraries  
2. Finish developing aggregation tree  
3. Start implementing clustering algorithms  
4. Start implementing H-SEND | Amit: 1, 4  
Dan: 1, 3  
Ivan: 1, 3  
Joe: 1, 4  
Matt: 1, 2  
Tim: 1 |
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 6 | 1. Start Implementing GUI  
2. Implementing clustering algorithms (Regular and Hierarchical)  
3. Implement H-SEND  
4. Implement a local predicate checker and reporter |
|   | Amit: 3  
Dan: 2  
Ivan: 2  
Joe: 3  
Matt: 4, 2  
Tim: 1 |
| 7 | 1. Library-ify Cluster implementations  
2. Implement LEACH using existing implementations as a base  
3. Work out how to interface the GUI and the base station  
4. Continue developing GUI  
5. Get in contact with Sain Saginbekov and sort out when we can use sensor nodes  
6. Contact Victor Sanchez and inform him when we are using sensor nodes  
7. Implement multi-hop predicate checking  
8. Slim down eLua enough for us to run it on the sensor nodes, to allow for dynamic predicate specification  
9. Research and start implementing network traffic logging and reporting |
|   | Amit: 7  
Dan: 1, 2  
Ivan: 9  
Joe: 5, 6, 7  
Matt: 8  
Tim: 3, 4 |
| 8 | 1. Start work on poster with the aim of finishing it by the end of week 9  
2. Detect and report neighbours for display in the visualisation tool, aim to be able to work in mobile networks, with gained and lost neighbours  
3. Log network data and report it to the network (for visualisation tool to replay traffic)  
4. Finish predicate language parser and code generation  
5. Tidy up n-hop HSEND  
6. Integrate n-hop HSEND with predicate language virtual machine  
7. Experiment with interference affecting node communications |
|   | Amit: 1, 5, 6  
Dan: 1, 3, (7)  
Ivan: 1, 2, (7)  
Joe: 1, 2, (7)  
Matt: 1, 4, 6  
Tim: 1, 4 |
| 9 | 1. Create poster for presentation |
|   | Amit: 1  
Dan: 1  
Ivan: 1  
Joe: 1  
Matt: 1  
Tim: 1 |
1. Code generation for predicate language
2. Network visualisation (node neighbours)
3. GUI to Node interface
4. Message logging
5. Neighbour Reporting
6. Turn existing code into libraries
7. Make existing code handle generic data
8. Integrate predicate evaluating VM and neighbour data retrieval
9. Write up notes on work done this term

Amit: 6, 7, 8, 9
Dan: 4, 5, 9
Ivan: 3, 5, 9
Joe: 6, 7, 8, 9
Matt: 6, 7, 8, 9
Tim: 1, 2, 9

### Term 2

<table>
<thead>
<tr>
<th>Week</th>
<th>Activities</th>
<th>Task Allocation</th>
</tr>
</thead>
</table>
| 1 - 2 | 1. Continue network visualisation  
2. Interface Base station node with desktop application  
3. Develop neighbour detection and reporting to sink  
4. Integrate predicate evaluating, predicate VM and neighbour data retrieval  
5. Finish parser and compiler of predicate language | Amit: 4  
Dan: 3  
Ivan: 2  
Joe: 4  
Matt: 4, 5  
Tim: 1 |
| 3 | 1. Continue network visualisation  
2. Interface Base station node with desktop application  
3. Develop type checker for predicate language  
4. Find bug that is causing unicast communications to fail  
5. Develop a way to handle packets larger than the maximum broadcast size | Amit: 4  
Dan: 5  
Ivan: 2  
Joe: 4  
Matt: 3  
Tim: 1 |
| 4 | 1. Continue network visualisation  
2. Interface Base station node with desktop application  
3. Find bug that is causing unicast communications to fail  
4. Develop a way to handle packets larger than the maximum broadcast size  
5. Develop some containers for use in our applications (lists and maps) | Amit: 3  
Dan: 4  
Ivan: 2  
Joe: 3  
Matt: 3, 5  
Tim: 1 |


| 6 | 1. Continue network visualisation
| 2. Interface Base station node with desktop application
| 3. Improve reliability of application on physical hardware
| 4. Refactor N-Hop-Req to split up request and reply phases
| 5. Develop a way to handle packets larger than the maximum broadcast size
| 6. Optimise and debug Tree Aggregation and Predicate Checking

| 6 - 10 | 1. Implement nhopflood
| 2. Implement Event based data sending
| 3. Implement Event-based predicate evaluation (PELE, PEGE)
| 4. Implement Global predicate evaluation (PEGP, PEGE)
| 5. Implement GUI
| 6. Get bidirectional communications between motes and desktop working
| 7. Develop a way to handle packets larger than the maximum broadcast size (multipacket)
| 8. Debug and Test
| 9. Implement failure responses for predicate evaluation
| 10. Run simulations to gather results

| 7.7 Management Issues |

| 7.7.1 Group Members Without Internet |

Unfortunately two of our group members were without internet for the first 3 weeks of term. This was an issue because they were unable to just work in the DCS labs because the computers there didn’t have the software required (such as VirtualBox or the WSN simulators). To work around this, those two members were given tasks that could be accomplished with their own machines and without internet. Once they obtained internet they were given tasks, that access allowed them to accomplish.

| 7.7.2 Society Exec Clashes |

Two members of our team are on the exec of Warwick Societies, this means that at certain parts of the year they were in high demand for those jobs. For instance during the first few weeks of term during Freshers’ Fortnight, the exec members were required to be involved in many of their events that they were running. Also during a large section of term 2, there were a number of events that required their presence such as exec elections and the Battle of the Bands. Between these busy periods the demand on their time decreased and made balancing the time been the demands of this project and their exec roles much easier.

| 7.7.3 High Number of Coursework Deadlines in Term 2 |

In the second term, there where many coursework deadlines for other course modules and some of the team members were unable to contribute towards the project for some periods of time due
to increased workload. On several occasion, the team leader had to put pressure on the team to ensure that some time was delegated to project work.

7.8 Legal and Professional Issues

This project involves working with open source code and libraries, so the use of this code must follow the requirements of the licences of that code. Ideally we would wish to contribute some of what we have developed back to the open source community as we feel that some of it may be of use. We intend to clarify the ownership of the code and will look into sorting out the legalities of contributing it back to the Contiki OS project.
8 Future Work

8.1 Improve Memory Management

Throughout the code-base malloc was used to dynamically allocate memory when needed as this simplified development. The advantage of this was that we did not have to learn alternative memory allocators used in Contiki. However, malloc is strongly advised against begin used due it potentially leading to memory fragmentation, which can decrease the overall memory space available. This can causes problems with the memory components long time life, and dynamic linking of new firmware. Using the alternate allocators to optimise the code could (i) decrease internal memory fragmentation (ii) decrease external memory fragmentation (iii) potentially increase performance and (iv) decrease memory usage.

Of one Contiki’s solutions to memory fragmentation is to use the mmem allocator. This forces pointers to be accessed through a layer of indirection, behind which memory can be compacted to avoid fragmentation. This is one of the benefits that using a virtual machine could provide but implemented in difficult to use C syntax. We feel that this is an instance where C simply starts to become difficult to use and where an alternative approach to developing distributed applications would be preferable.

8.2 Improve C Containers Developed

To aid in the development of our applications (and to ease developers from other languages such as Java to C) we developed a library of containers. These containers are very simple with the aim of having low memory usage. However, the complexity of certain operations could be improved. For example the map container’s time complexity for retrieval of an element given a key is $O(n)$ where $n$ is the number of elements in the container as the underlying container is simply an array. This could be improved to $O(\log(n))$ by sorting the elements in the underlying array, or improved further to amortized $O(1)$ by using a hash table. Hash tables were avoided simply due to the extra memory overhead that would have been required to setup the data structure. Although, whatever improved container is chosen, the importance of memory usage (including issues such as fragmentation) should be taken into account.

8.3 Stateful Predicates

In subsubsection 1.3.4 we discussed a number of real-world deployments that have been undertaken, one of these was habitat monitoring on Great Duck Island by Szewczyk et al. One of the issues that the authors found was that clock drift could lead to lots of collisions because assigned slots that should not overlap will come to do so after a large period of time. A way that our solution could have been extended to detect this is to add the notion of history to a predicate. So instead of just evaluating the predicate based on what is available at that instant, the predicate is also evaluated on what is known about that node in the past.

This becomes difficult due to the limitations of the mote hardware. As the motes have limited memory, they will not be able to hold all of the information they may wish to be evaluating over. There is also an issue where individual motes running the same firmware may have different memory usages due to memory fragmentation and the way that dynamic memory allocation works. So this rules out allowing our in-network predicate evaluation algorithms from utilising this history. However, the at-sink global evaluation algorithms can be evaluated on much more capable hardware with much larger memories. The memory size for a desktop computer is many of order of magnitude greater than that a mote (16GB vs 16KB). Finally, as the full state history is available off the sensor network, it would also allow developers to write their own analysis scripts out of the scope of testing predicates.
8.4 Mote Mobility

When considering mobility of the sensor network the problem becomes very different. We have developed our solution on the assumption that it will be used to test certain properties that have been set up in advance to ensure that energy is saved later on, or to check certain application properties and the way that they relate to their neighbours. If the neighbours are continuously changing (as they would be in a deployment such as ZebraNet [64]), then checking certain neighbour properties would be meaningless because for some of those properties there would be no reason to set them up.

8.5 Improve Failure Response

When a predicate fails it sends a failure message to the base station, this message contains the state that was used to evaluate the predicate. Currently there are two deficiencies to our implemented approach. The first as was shown by our results was that for predicates evaluated locally in-network the number of response messages reaching the sink was very poor. By switching from Contiki’s mesh communication to a more reliable networking protocol we believe that the delivery rate could improve.

The second issue is that once the failure message reaches the sink and the result is shown in the GUI, but nothing else happens. What would be useful to the users of the system would be to run that data that was used to evaluate the predicate through a virtual machine that doesn’t focus on fast evaluating with a small firmware size, but instead focuses on producing a detailed error message. This error message would contain useful messages such as the values the variable names had and why their state causes the predicate to be false.
9 Evaluation and Conclusions

To conclude we have found this a difficult project to complete. Mostly this was due to the fact that a number of the libraries that we expected to be available were not, our unfamiliarity with Contiki and the initial lack of direction for the project. We have learnt that it is much better to have a very well defined goal to being with, even in a research project such as this.

As part of this project we have produced four different predicate evaluation libraries that were analysed to find the relative performances between evaluating predicates in-network or at a sink node after gathering the network’s data. We also investigated under what circumstances should a node’s data be sent, when it changes or periodically, and how to send that data. In order to aid system administrators of sensor networks we developed a GUI that could communicate with the network, this GUI was used to send predicates into the network and visually display the network’s configuration and predicate failure responses to the user. To evaluate predicates a virtual machine and scripting language were developed so that new predicates could be deployed and old ones updated or removed. Our aim was to eliminate the need to redeploy the firmware across the networks.

From our results some libraries perform well in certain situations and other libraries in other situations, which means that it is up to the developers of systems to choose which to deploy. We believe that there is much room for improvement in the number of messages sent and the number of failure responses received. Improving the percentage of predicates evaluated successfully may be harder due to the time delays between data being generated and sent and the predicates being evaluated.

As a group we’ve had to deal with the challenges of organisation, making sure everybody knew what they had to do, and when it was needed for completion. This proved to be initially complex working around the schedules of each of the team members, but tasks were modularised to help negate this issue. Making sure each team member was contactable and was kept up-to-date with the current progress of the project ensured that everything was run as smoothly as possible.

Overall, we have learnt a lot as a group in terms of undertaking projects and how to develop for unreliable distributed systems such as wireless sensor networks. Our initial aims were to explore tools assisting in the debugging of WSNs, and our final systems makes great progress towards those goals. The tools have been shown to consistently evaluate predicates across different network sizes, subject to traditional drawbacks within WSNs (such as message loss and collisions). The visualisation tool itself allows the adjustment of these predicates, and the display of the current network status for a live deployed network, and not just a simulated network (such as COOJA). There is still a large amount of work that could be done, including expanding the abilities of the virtual machine, and the visualisation tool’s display properties. We hope that some of the work here can be contributed back to the open source community where it will help others to develop applications with Contiki.
Appendices

A Device Specifications

A.1 Sensor Board: CM5000

<table>
<thead>
<tr>
<th>Item</th>
<th>Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processor</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor Model</td>
<td>Texas Instruments® MSP430F1611</td>
<td>Texas Instruments® MSP430 family</td>
</tr>
<tr>
<td>Memory</td>
<td>48KB</td>
<td>Program Flash</td>
</tr>
<tr>
<td></td>
<td>10KB</td>
<td>Data RAM</td>
</tr>
<tr>
<td></td>
<td>1MB</td>
<td>External Flash (ST® M25P80)</td>
</tr>
<tr>
<td>ADC</td>
<td>12bit resolution</td>
<td>8 channels</td>
</tr>
<tr>
<td>Interfaces</td>
<td>UART, SPI, I2C</td>
<td>Serial Interfaces</td>
</tr>
<tr>
<td></td>
<td>USB</td>
<td>External System Interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(FTI® FT232BM)</td>
</tr>
<tr>
<td><strong>Radio</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RF Chip</td>
<td>Texas Instruments® CC2420</td>
<td>IEEE 802.15.4 2.4GHz Wireless Module</td>
</tr>
<tr>
<td>Frequency Band</td>
<td>2.4GHz ~ 2.485GHz</td>
<td>IEEE 802.15.4 compliant</td>
</tr>
<tr>
<td>Sensitivity</td>
<td>-95dBm typ</td>
<td>Receive Sensitivity</td>
</tr>
<tr>
<td>Transfer Rate</td>
<td>250Kbps</td>
<td>IEEE 802.15.4 compliant</td>
</tr>
<tr>
<td>RF Power</td>
<td>-25dBm ~ 0dBm</td>
<td>Software Configurable</td>
</tr>
<tr>
<td>Range</td>
<td>~ 120m (outdoor)</td>
<td>Longer ranges possible with optional SMA antenna attached</td>
</tr>
<tr>
<td></td>
<td>20 ~ 30m (indoor)</td>
<td></td>
</tr>
<tr>
<td>Current Draw</td>
<td>RX: 18.8mA</td>
<td>Lower RF Power Modes reduce consumption</td>
</tr>
<tr>
<td></td>
<td>TX: 17.4mA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sleep mode: 1uA</td>
<td></td>
</tr>
<tr>
<td>RF Power Supply</td>
<td>2.1V ~ 3.6V</td>
<td>CC2420 Input Power</td>
</tr>
<tr>
<td>Antenna</td>
<td>Dipole Antenna / PCB Antenna</td>
<td>Additional SMA connector available for extra antenna</td>
</tr>
<tr>
<td><strong>Sensors</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Light 1</td>
<td>Hamamatsu S1087 Series</td>
<td>Visible Range (560 nm peak sensitivity wavelength)</td>
</tr>
<tr>
<td>Light 2</td>
<td>Hamamatsu S1087 Series</td>
<td>Visible &amp; Infrared Range (960 nm peak sensitivity wavelength)</td>
</tr>
<tr>
<td>Temperature &amp; Humidity</td>
<td>Sensirion SHT11</td>
<td>Temperature Range: -40 ~ 123.8 °C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Temperature Resolution: ± 0.01 (typical)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Temperature Accuracy: ± 0.4 °C (typical)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Humidity Range: 0 ~ 100% RH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Humidity Resolution: 0.05 (typical)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Humidity Accuracy: ± 3 %RH (typical)</td>
</tr>
</tbody>
</table>

Table 9: Specifications for the CM5000 Wireless Sensor Node [5]
## A.2 Network Infrastructure: UD1000

![Image of UD1000 Sensor Network Sink](image)

<table>
<thead>
<tr>
<th>Item</th>
<th>Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processor</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor Model</td>
<td>Texas Instruments® MSP430F1611</td>
<td>Texas Instruments® MSP430 family</td>
</tr>
<tr>
<td>Memory</td>
<td>48KB</td>
<td>Program Flash</td>
</tr>
<tr>
<td></td>
<td>10KB</td>
<td>Data RAM</td>
</tr>
<tr>
<td>ADC</td>
<td>12bit resolution</td>
<td>8 channels</td>
</tr>
<tr>
<td>Interfaces</td>
<td>UART, SPI, I²C</td>
<td>Serial Interfaces</td>
</tr>
<tr>
<td></td>
<td>USB</td>
<td>External System Interface (FTI® FT232BM)</td>
</tr>
<tr>
<td><strong>Radio</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RF Chip</td>
<td>Texas Instruments® CC2420</td>
<td>IEEE 802.15.4 2.4GHz Wireless Module</td>
</tr>
<tr>
<td>Frequency Band</td>
<td>2.4GHz ~ 2.485GHz</td>
<td>IEEE 802.15.4 compliant</td>
</tr>
<tr>
<td>Sensitivity</td>
<td>-95dBm typ</td>
<td>Receive Sensitivity</td>
</tr>
<tr>
<td>Transfer Rate</td>
<td>250Kbps</td>
<td>IEEE 802.15.4 compliant</td>
</tr>
<tr>
<td>RF Power</td>
<td>-25dBm ~ 0dBm</td>
<td>Software Configurable</td>
</tr>
<tr>
<td>Range</td>
<td>~ 40m (outdoor), 15~ 20m (indoor)</td>
<td>Dongle orientation dependent</td>
</tr>
<tr>
<td>Current Draw</td>
<td>RX: 18.8mA</td>
<td>Lower RF Power Modes reduce consumption</td>
</tr>
<tr>
<td></td>
<td>TX: 17.4mA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sleep mode: 1uA</td>
<td></td>
</tr>
<tr>
<td>RF Power Supply</td>
<td>2.1V ~ 3.6V</td>
<td>CC2420 Input Power</td>
</tr>
<tr>
<td>Antenna</td>
<td>Ceramic antenna</td>
<td></td>
</tr>
<tr>
<td><strong>Electromechanical Characteristics</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dimensions</td>
<td>65mm x 22.5mm x 14mm</td>
<td>Including housing</td>
</tr>
<tr>
<td>Weight</td>
<td>15g</td>
<td></td>
</tr>
<tr>
<td>Power</td>
<td>5V</td>
<td>DC over USB</td>
</tr>
<tr>
<td>Current</td>
<td>90mA</td>
<td>Max rated current over USB</td>
</tr>
<tr>
<td>Operating Temperature</td>
<td>-25°C ~ +60°C</td>
<td></td>
</tr>
<tr>
<td>Storage Temperature</td>
<td>-40°C ~ +60°C</td>
<td></td>
</tr>
<tr>
<td>Operating Humidity</td>
<td>5% ~ 95%</td>
<td>Non condensing</td>
</tr>
<tr>
<td>Protection type</td>
<td>IP20</td>
<td>Non condensing</td>
</tr>
</tbody>
</table>

Table 10: Specifications for the UD1000 Sensor Network Sink [6]
B Algorithm Implementations

B.1 Multipacket

```c
#ifndef CS407_MULTIPACKET_H
#define CS407_MULTIPACKET_H

#include "net/rime/runicast.h"
#include "containers/linked-list.h"
#include "containers/map.h"
#include <stdbool.h>
#include <string.h>

struct multipacket_conn;

typedef struct multipacket_callbacks
{
    void (*recv)(struct multipacket_conn * conn, rimeaddr_t const * from, void * data, unsigned int length);
    void (*sent)(struct multipacket_conn * conn, rimeaddr_t const * to, void * data, unsigned int length);
} multipacket_callbacks_t;

typedef struct multipacket_conn
{
    // Keep connection in this order!
    struct runicast_conn rc;
    uint16_t id;
    linkedlist_t sending_packets; // A queue of packets to be sent
    map_t receiving_packets; // Map of recv_key_t to multipacket_receiving_packet_t
    multipacket_callbacks_t const * callbacks;
    struct ctimer ct_sender;
} multipacket_conn_t;

bool multipacket_open(multipacket_conn_t * conn, uint16_t channel, multipacket_callbacks_t const * callbacks);
void multipacket_close(multipacket_conn_t * conn);
void multipacket_send(multipacket_conn_t * conn, rimeaddr_t const * target, void * data, unsigned int length);

#endif /*CS407_MULTIPACKET_H*/
```

```c
#include "net/multipacket.h"
#include "sys/clock.h"
#include <stdbool.h>
#include <stdio.h>

#ifdef MULTIPACKET_DEBUG
#define MPDPRINTF(...) printf(__VA_ARGS__)
#else
#define MPDPRINTF(...)\
#endif
```

83
# From: http://stackoverflow.com/questions/3437404/min-and-max-in-c
#define min(a, b) \
({ __typeof__ (a) _a = (a); \ 
    __typeof__ (b) _b = (b); \ 
    _a < _b ? _a : _b; })

// The maximum number of retransmits before giving up sending a packet
#define MAX_REXMITS 4

// The time between transmits
#define SEND_PERIOD ((clock_time_t) 1 * CLOCK_SECOND)

typedef struct
{
    uint16_t id;
    rimeaddr_t target;
    rimeaddr_t source;
    unsigned int length;
    unsigned int sent;
    uint8_t seqno;

    // Data stored from here onwards
} multipacket_sending_packet_t;

// Need to use a union to not break the strict aliasing rule
typedef union
{
    uint32_t i32;
    struct {
        uint16_t id;
        rimeaddr_t originator;
    } data;
} recv_key_t;

typedef struct
{
    recv_key_t key; // Keep this key first
    unsigned int length;
    unsigned int data_received;
    uint8_t last_seqno;

    // Data stored from here onwards
} multipacket_receiving_packet_t;

// Gets the pointer to the data stored after the struct
static inline void * sending_data(multipacket_sending_packet_t * packet)
{
    return (packet + 1);
}

static inline void * receiving_data(multipacket_receiving_packet_t * packet)
{
    return (packet + 1);
}

// Equality function for two recv_key_t
static bool recv_key_equality(void const * left, void const * right)
{
    if (left == NULL || right == NULL)
        return false;
    // Your equality logic here...
}

84
recv_key_t const * l = (recv_key_t const *)left;
recv_key_t const * r = (recv_key_t const *)right;

return l->i32 == r->i32;

// Repurpose header attributes
#define PACKETBUF_ATTR_EPACKET_SEQNO PACKETBUF_ATTR_EPACKET_TYPE
#define PACKETBUF_ATTR_EPACKET_ELENGTH PACKETBUF_ATTR_TTL

static const struct packetbuf_attrlist multipacket_attributes[] = {
    { PACKETBUF_ATTR_EPACKET_ID, PACKETBUF_ATTR_BYTE * sizeof(uint16_t) }, // ID
    { PACKETBUF_ATTR_EPACKET_SEQNO, PACKETBUF_ATTR_BYTE * sizeof(uint8_t) }, // seqno
    { PACKETBUF_ATTR_EPACKET_ELENGTH, PACKETBUF_ATTR_BYTE * sizeof(unsigned int) }, // Length
    { PACKETBUF_ADDR_ESENDER, PACKETBUF_ADDRSIZE },
    RUNICAST_ATTRIBUTES
} PACKETBUF_ATTR_LAST;

static inline multipacket_conn_t * runicast_conncvt(struct runicast_conn * conn)
{
    return (multipacket_conn_t *)conn;
}

static void send_loop_callback(void * ptr)
{
    multipacket_conn_t * conn = (multipacket_conn_t *)ptr;

    // Check that we have something to send onwards and that runicast is not currently sending
    if (!linked_list_is_empty(&conn->sending_packets) && !runicast_is_transmitting(&conn->rc))
    {
        multipacket_sending_packet_t * details = linked_list_peek(&conn->sending_packets);

        unsigned int to_send = min(PACKETBUF_SIZE, details->length - details->sent);

        void * send_start = ((char *) sending_data(details)) + details->sent;

        packetbuf_clear();
        packetbuf_set_datalen(to_send);
        void * msg = packetbuf_dataptr();
        memcpy(msg, send_start, to_send);

        packetbuf_set_attr(PACKETBUF_ATTR_EPACKET_ID, details->id);
        packetbuf_set_attr(PACKETBUF_ATTR_EPACKET_SEQNO, details->seqno);
        packetbuf_set_attr(PACKETBUF_ATTR_EPACKET_ELENGTH, details->length);
        packetbuf_set_addr(PACKETBUF_ADDR_ESENDER, &details->source);

        runicast_send(&conn->rc, &details->target, MAX_REXMITS);

        // Record the number of bytes we have sent
        details->sent += to_send;
        details->seqno += 1;

        // Check to see if we have finished sending
        if (details->sent == details->length)
        {
            conn->callbacks->sent(conn, &details->target, sending_data(details), details->length);
            linked_list_pop(&conn->sending_packets);
        }
    }

    // Set the timer to call this function again
    ctimer_reset(&conn->ct_sender);
}
static void recv_from_runicast(struct runicast_conn * rc, rimeaddr_t const * from, uint8_t seqno)
{
    multipacket_conn_t * conn = runicast_conncvt(rc);

    // We have received a packet, now we need to join segmented data back together
    const uint16_t packet_id = packetbuf_attr(PACKETBUF_ATTR_EPACKET_ID);
    const uint8_t seq = packetbuf_attr(PACKETBUF_ATTR_EPACKET_SEQNO);
    const unsigned int data_length = packetbuf_attr(PACKETBUF_ATTR_EPACKET_LENGTH);
    rimeaddr_t const * source = packetbuf_addr(PACKETBUF_ADDR_ESENDER);
    void const * data_recv = packetbuf_dataptr();
    const unsigned int recv_length = packetbuf_datalen();

    recv_key_t key;
    key.data.id = packet_id;
    rimeaddr_copy(&key.data.originator, source);

    // Get the data already received about this packet
    multipacket_receiving_packet_t * details =
        (multipacket_receiving_packet_t *)map_get(&conn->receiving_packets, &key);
    void * data_to_pass_onwards = NULL;
    unsigned int length_of_data_to_pass_onwards = 0;
    bool should_remove_from_map = false;

    if (details == NULL)
    {
        // We have not received this message before!
        if (seq == 0)
        {
            // OPTIMISATION: avoid allocating memory when we will free it shortly
            // this is the case when we have an entire message in a single packet
            if (recv_length == data_length)
            {
                data_to_pass_onwards = data_recv;
                length_of_data_to_pass_onwards = recv_length;
                // No need to remove from the map as we never added it
            }
        }
        else
        {
            // Record this packet
            details = (multipacket_receiving_packet_t *)
                malloc(sizeof(multipacket_receiving_packet_t) + data_length);
            details->key = key;
            details->length = data_length;
            details->last_seqno = seq;
            details->data_received = recv_length;
            memcpy(receiving_data(details), data_recv, recv_length);
            map_put(&conn->receiving_packets, details);
        }
    }
    else
    {
        // If we do not have a record, and the sequency number is greater
        // than 0, then we have already missed the first packet and there
        // would be no point in recording any more.
    }
}

void * data_ptr = ((char *) receiving_data(details)) + details->data_received;

// Copy in the newly received data
memcpy(data_ptr, data_recv, recv_length);

// Update the data received and the last seqno
details->data_received += recv_length;
details->last_seqno = seq;

// Check if we have got everything, if so set the relevant variables
if (details->data_received == details->length)
{
    data_to_pass_onwards = receiving_data(details);
    length_of_data_to_pass_onwards = details->length;
    should_remove_from_map = true;
}

// Check to see if we have fully received this packet
if (data_to_pass_onwards != NULL)
{
    conn->callbacks->recv(
        conn, source, data_to_pass_onwards, length_of_data_to_pass_onwards);
    if (should_remove_from_map)
    {
        // This packet has been received so remove it
        map_remove(&conn->receiving_packets, &key);
    }
}

static const struct runicast_callbacks rccallbacks = {&recv_from_runicast, NULL, NULL};

bool multipacket_open(multipacket_conn_t * conn,
        uint16_t channel, multipacket_callbacks_t const * callbacks)
{
    if (conn != NULL)
    {
        runicast_open(&conn->rc, channel, &rccallbacks);
        channel_set_attributes(channel, multipacket_attributes);
        conn->id = 0;
        conn->callbacks = callbacks;
        ctimer_set(&conn->ct_sender, SEND_PERIOD, &send_loop_callback, conn);
        linked_list_init(&conn->sending_packets, &free);
        map_init(&conn->receiving_packets, &recv_key_equality, &free);
        return true;
    }
    return false;
}

void multipacket_close(multipacket_conn_t * conn)
{
    if (conn != NULL)
    {
        runicast_close(&conn->rc);
        ctimer_stop(&conn->ct_sender);
    }
void multipacket_send(multipacket_conn_t * conn, rimeaddr_t const * target, 
    void * data, unsigned int length)
{
    // Allocate the packet to send
    multipacket_sending_packet_t * details = 
        (multipacket_sending_packet_t *) 
            malloc(sizeof(multipacket_sending_packet_t) + length);
    details->id = conn->id++;
    details->length = length;
    details->sent = 0;
    details->seqno = 0;
    rimeaddr_copy(&details->target, target);
    rimeaddr_copy(&details->source, &rimeaddr_node_addr);
    
    memcpy(sending_data(details), data, length);
    
    // Add to the queue to send
    linked_list_append(&conn->sending_packets, details);
}

B.2 Tree Aggregation

#ifndef CS407_TREE_AGGREGATOR_H
#define CS407_TREE_AGGREGATOR_H

#include "net/rime/stbroadcast.h"

#include "net/multipacket.h"

struct tree_agg_conn;

typedef struct
{
    // The function called when a message is received at the sink.
    void (* recv)(struct tree_agg_conn * conn, 
        rimeaddr_t const * source, void const * packet, unsigned int length);

    // This function is called when a node has finished setting up
    void (* setup_complete)(struct tree_agg_conn * conn);

    // Add the new data to the stored data
    void (* aggregate_update)(struct tree_agg_conn * tconn, 
        void * data, void const * to_apply, unsigned int length);

    // This function is used to add a nodes own one data
    void (* aggregate_own)(struct tree_agg_conn * tconn, void * data);

    // This function is called when a node needs to save a packet
    // The arguments are: Connection, Packet and the Packet Length
    void (* store_packet)(struct tree_agg_conn * conn, void const * packet, unsigned int length);

    // This function is called to write the nodes stored data to an outward packet
    void (* write_data_to_packet)(struct tree_agg_conn * conn, void ** data, unsigned int * length);
} tree_agg_callbacks_t;

typedef struct tree_agg_conn
{
// DO NOT CHANGE CONNECTION ORDER!!!
struct stbroadcast_conn bc;
struct multipacket_conn mc;
bool has_seen_setup;
bool is_collecting;
bool is_leaf_node;
rmaddr_t best_parent;
rmaddr_t sink;
unsigned int best_hop;
void * data;
size_t data_length;
tree_agg_callbacks_t const * callbacks;

// Timers
struct ctimer ctrecv;
struct ctimer aggregate_ct;
struct ctimer ct_parent_detect;
struct ctimer ct_open;
struct ctimer ct_wait_finished;
} tree_agg_conn_t;

bool tree_agg_open(tree_agg_conn_t * conn, rmaddr_t const * sink,
        uint16_t ch1, uint16_t ch2,
        size_t data_size,
        tree_agg_callbacks_t const * callbacks);

void tree_agg_close(tree_agg_conn_t * conn);

void tree_agg_send(tree_agg_conn_t * conn, void * data, size_t length);
#endif /*CS407_TREE_AGGREGATOR_H*/

#include "net/tree-aggregator.h"
#include "contiki.h"
```c
#include "lib/random.h"
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <limits.h>
#include "random-range.h"
#include "led-helper.h"
#include "sensor-converter.h"
#include "debug-helper.h"

#ifdef TREE_AGG_DEBUG
#define TADPRINTF(...) printf(__VA_ARGS__)
#else
#define TADPRINTF(...)  
#endif

static inline tree_agg_conn_t * conncvt_stbcast(struct stbroadcast_conn * conn)
{
    return (tree_agg_conn_t *)conn;
}

static inline tree_agg_conn_t * conncvt_multipacket(struct multipacket_conn * conn)
{
    return (tree_agg_conn_t *)
    (((char *)conn) - sizeof(struct stbroadcast_conn));
}

// The amount of stubborn broadcasts to allow time for and how often for them to broadcast
#define STUBBORN_WAIT_COUNT 3u
#define MIN_SEND_TIME 1
#define MAX_SEND_TIME 3

// Time to gather aggregations over
#define AGGREGATION_WAIT (clock_time_t)(30 * CLOCK_SECOND)

// Time to wait to detect parents
#define PARENT_DETECT_WAIT (clock_time_t)((MAX_SEND_TIME * (1 + STUBBORN_WAIT_COUNT)) * CLOCK_SECOND)

static void stbroadcast_cancel_void_and_callback(void * ptr)
{
    tree_agg_conn_t * conn = conncvt_stbcast((struct stbroadcast_conn *)ptr);
    stbroadcast_cancel(&conn->bc);
    TADPRINTF("Tree Agg: Stubborn bcast canceled, setup complete\n");
    (*conn->callbacks->setup_complete)(conn);
}

typedef struct
{
    rimeaddr_t source;
    rimeaddr_t parent;
    unsigned int hop_count;
} setup_tree_msg_t;

static void parent_detect_finished(void * ptr)
{
    tree_agg_conn_t * conn = (tree_agg_conn_t *)ptr;
    TADPRINTF("Tree Agg: Timer on %s expired\n", addr2str(&rimeaddr_node_addr));
    TADPRINTF("Tree Agg: Found Parent:%s Hop:%u\n",
```
addr2str(&conn->best_parent), conn->best_hop);

// Send a message that is to be received by the children of this node.
packetbuf_clear();
packetbuf_set_datalen(sizeof(setup_tree_msg_t));
setup_tree_msg_t * msg = (setup_tree_msg_t *)packetbuf_dataptr();

// We set the parent of this node to be the best parent we heard
rimeaddr_copy(&msg->source, &rimeaddr_node_addr);
rimeaddr_copy(&msg->parent, &conn->best_parent);

// If at the max, want to keep as UINT_MAX to prevent integer overflow to 0
msg->hop_count = (conn->best_hop == UINT_MAX) ? UINT_MAX : conn->best_hop + 1;
clock_time_t send_period = random_time(MIN_SEND_TIME, MAX_SEND_TIME, 0.1);
clock_time_t wait_period = send_period * STUBBORN_WAIT_COUNT;
stbroadcast_send_stubborn(&conn->bc, send_period);

// Wait for a bit to allow a few messages to be sent
// Then close the connection and tell user that we are done
timer_set(&conn->ct_parent_detect, wait_period, &stbroadcast_cancel_void_and_callback, conn);
}

static void finish_aggregate_collect(void * ptr)
{
  tree_agg_conn_t * conn = (tree_agg_conn_t *)ptr;
  (*conn->callbacks->aggregate_own)(conn, conn->data);
  void * data;
  size_t length;

  // Copy aggregation data into the packet
  (*conn->callbacks->write_data_to_packet)(conn, &data, &length);
multipacket_send(&conn->mc, &conn->best_parent, data, length);

  // We need to free the allocated packet data
  free(data);

  TADPRINTF("Tree Agg: Send Agg\n");
toggle_led_for(LEDGS_GREEN, 1 * CLOCK_SECOND);

  // We are no longer collecting aggregation data
  conn->is_collecting = false;

  // Reset the data we have stored to nothing
  // We can do this as write_data_to_packet should have freed the memory
  // if anything was allocated.
  memset(conn->data, 0, conn->data_length);
}

static void recv_aggregate(struct multipacket_conn * ptr,
rimeaddr_t const * originator, void * msg, unsigned int length)
{
  tree_agg_conn_t const * originator, void * msg, unsigned int length)
  
  if (tree_agg_is_sink(conn))
  {
    TADPRINTF("Tree Agg: We're sink, got message from %s length:%u, sending to user\n", 
      addr2str(originator), length);
    // We need to apply the sink's data to the received data

(*conn->callbacks->store_packet)(conn, msg, length);
(*conn->callbacks->aggregate_own)(conn, conn->data);

void * data;
size_t data_length;

// Copy aggregation data into the packet
(*conn->callbacks->write_data_to_packet)(conn, &data, &data_length);

// Pass this message up to the user
(*conn->callbacks->recv)(conn, originator, data, data_length);

// Free the allocated data
free(data);
}
else
{
    // Apply some aggregation function
    if (tree_agg_is_collecting(conn))
    {
        TADPRINTF("Tree Agg: Cont Agg With: %s of length %u\n",
            addr2str(originator), length);

        (*conn->callbacks->aggregate_update)(conn, conn->data, msg, length);
    }
    else
    {
        TADPRINTF("Tree Agg: Start Agg Addr: %s of length %u\n",
            addr2str(originator), length);

        // We need to copy the user's data into our memory,
        // so we can apply future aggregations to it.
        (*conn->callbacks->store_packet)(conn, msg, length);

        // We have started collection
        conn->is_collecting = true;

        // Start aggregation timer
        ctimer_set(&conn->aggregate_ct, AGGREGATION_WAIT, &finish_aggregate_collect, conn);
    }
}

// The function that will be executed when a setup message is received
static void recv_setup(struct stbroadcast_conn * ptr)
{
    toggle_led_for(LEDS_GREEN, CLOCK_SECOND);

    tree_agg_conn_t * conn = conncvt_stbcast(ptr);

    // Store a local copy of the message
    setup_tree_msg_t msg;
    memcpy(&msg, packetbuf_dataptr(), sizeof(setup_tree_msg_t));

    TADPRINTF("Tree Agg: Got setup message from %s\n", addr2str(&msg.source));

    // If the sink received a setup message, then do nothing
    // it doesn't need a parent as it is the root.
    if (tree_agg_is_sink(conn))
    {
        TADPRINTF("Tree Agg: We are the sink node, so should not listen for parents.\n");
        return;
    }

    // If this is the first setup message that we have seen
    // Then we need to start the collect timeout
if (!conn->has_seen_setup)
    {
        conn->has_seen_setup = true;

        // Indicate that we are setting up
        leds_on(LEDS_RED);

        // Start the timer that will call a function when we are
        // done detecting parents.
        ctimer_set(&conn->ctrecv, PARENT_DETECT_WAIT, &parent_detect_finished, conn);

        TADPRINTF("Tree Agg: Not seen setup message before, so setting timer...
");
    }

    // As we have received a message we need to record the node
    // it came from, if it is closer to the sink.
    if (msg.hop_count < conn->best_hop)
    {
        // Set the best parent, and the hop count of that node
        rimeaddr_copy(&conn->best_parent, &msg.source);
        conn->best_hop = msg.hop_count;
    }

    // If the parent of the node that sent this message is this node,
    // then we are not a leaf
    if (conn->is_leaf_node && rimeaddr_cmp(&msg.parent, &rimeaddr_node_addr))
    {
        TADPRINTF("Tree Agg: Node (%s) is our child, we are not a leaf.
", addr2str(&msg.source));
        conn->is_leaf_node = false;

        leds_off(LEDS_RED);
    }

static const struct stbroadcast_callbacks callbacks_setup = { &recv_setup, NULL };
static const struct multipacket_callbacks callbacks_aggregate = { &recv_aggregate, NULL };

void tree_agg_setup_wait_finished(void * ptr)
    {
        tree_agg_conn_t * conn = (tree_agg_conn_t *)ptr;

        leds_on(LEDS_BLUE);

        // Send the first message that will be used to set up the aggregation tree
        packetbuf_clear();
        packetbuf_set_datalen(sizeof(setup_tree_msg_t));
        setup_tree_msg_t * msg = (setup_tree_msg_t *)packetbuf_dataptr();
        rimeaddr_copy(&msg->source, &rimeaddr_node_addr);
        rimeaddr_copy(&msg->parent, &rimeaddr_null);
        msg->hop_count = 1;

        clock_time_t send_period = random_time(MIN_SEND_TIME, MAX_SEND_TIME, 0.1);
        clock_time_t wait_period = send_period * STUBBORN_WAIT_COUNT;

        stbroadcast_send_stubborn(&conn->bc, send_period);

        // Wait for a bit to allow a few messages to be sent
        ctimer_set(&conn->ct_wait_finished, wait_period, &stbroadcast_cancel, &conn->bc);
    }

bool tree_agg_open(tree_agg_conn_t * conn, rimeaddr_t const * sink,
    uint16_t chi, uint16_t ch2,
    size_t data_size,
tree_agg_callbacks_t const * callbacks)
{
    if (conn != NULL && sink != NULL &&
        callbacks->recv != NULL && callbacks->setup_complete != NULL &&
        callbacks->aggregate_update != NULL && callbacks->aggregate_own != NULL &&
        callbacks->store_packet != NULL)
    {
        stbroadcast_open(&conn->bc, ch1, &callbacks_setup);
        multipacket_open(&conn->mc, ch2, &callbacks_aggregate);
        conn->has_seen_setup = false;
        conn->is_collecting = false;
        conn->is_leaf_node = true;
        rimeaddr_copy(&conn->best_parent, &rimeaddr_null);
        rimeaddr_copy(&conn->sink, sink);
        conn->best_hop = UINT_MAX;
        conn->data = malloc(data_size);
        // Make sure memory allocation was successful
        if (conn->data == NULL)
        {
            TADPRINTF("Tree Agg: MAF!\n");
            return false;
        }
        conn->data_length = data_size;
        conn->callbacks = callbacks;
        if (tree_agg_is_sink(conn))
        {
            // Wait a bit to allow processes to start up
            ctimer_set(&conn->ct_open,
                10 * CLOCK_SECOND, &tree_agg_setup_wait_finished, conn);
        }
        return true;
    }
    return false;
}

void tree_agg_close(tree_agg_conn_t * conn)
{
    TADPRINTF("Tree Agg: Closing connection.\n");
    if (conn != NULL)
    {
        stbroadcast_close(&conn->bc);
        multipacket_close(&conn->mc);
        if (conn->data != NULL)
        {
            free(conn->data);
            conn->data = NULL;
        }
        ctimer_stop(&conn->ctrecv);
        ctimer_stop(&conn->aggregate_ct);
        ctimer_stop(&conn->ct_parent_detect);
        ctimer_stop(&conn->ct_open);
        ctimer_stop(&conn->ct_wait_finished);
void tree_agg_send(tree_agg_conn_t * conn, void * data, size_t length)
{
    if (conn != NULL && data != NULL && length != 0)
    {
        TADPRINTF("Tree Agg: Sending to %s, length=%d\n",
                   addr2str(&conn->best_parent), length);
        multipacket_send(&conn->mc, &conn->best_parent, data, length);
    }
}

B.3 N-Hop Request

#ifndef CS407_NHOPREQ_H
#define CS407_NHOPREQ_H

#include "net/rime.h"
#include "net/rime/stbroadcast.h"
#include "net/rime/runicast.h"

#include "containers/map.h"
#include <stdbool.h>
#include <stdint.h>

#include <stdbool.h>
#include <stdint.h>

struct nhopreq_conn;

typedef struct
{
    // Used to get information on the node to be sent
    void (* data_fn)(struct nhopreq_conn * conn, void * data);

    // Called when requested data is received
    void (* receive_fn)(struct nhopreq_conn * conn,
                         rimeaddr_t const * from, uint8_t hops, void const * data);
} nhopreq_callbacks_t;

typedef struct nhopreq_conn
{
    // Keep connections in this order
    struct runicast_conn ru;
    struct stbroadcast_conn bc;

    uint16_t message_id;
    unsigned int data_size;
    nhopreq_callbacks_t const * callbacks;
    map_t mote_records;
    ctimer runicast_timer;
    ctimer forward_timer;
    ctimer datareq_stbroadcast_stop_timer;
} nhopreq_conn_t;

bool nhopreq_start(
    nhopreq_conn_t * conn, uint8_t ch1, uint8_t ch2,
unsigned int data_size, nhopreq_callbacks_t const * callbacks);
bool nhopreq_stop(nhopreq_conn_t * conn);
void nhopreq_request_info(nhopreq_conn_t * conn, uint8_t hops);
#endif /*CS407_NHOPREQ_H*/

#include "nhopreq.h"
#include "contiki.h"
#include "packetbuf.h"
#include "lib/random.h"
#include "dev/leds.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "net/rimeaddr-helpers.h"
#include "random-range.h"
#include "sensor-converter.h"
#include "debug-helper.h"

#ifdef NHOPREQ_DEBUG
#define NHRDPRINTF(...) printf(__VA_ARGS__)
#else
#define NHRDPRINTF(...) 
#endif

#ifndef STBROADCAST_ATTRIBUTES
#define STBROADCAST_ATTRIBUTES BROADCAST_ATTRIBUTES
#endif

// The custom headers we use
static const struct packetbuf_attrlist stbroadcast_attributes[] = {
    { PACKETBUF_ADDR_ESENDER, PACKETBUF_ADDRSIZE },
    { PACKETBUF_ADDR_SENDER, PACKETBUF_ADDRSIZE },
    { PACKETBUF_ATTR_HOPS, PACKETBUF_ATTR_BIT * 4 },
    { PACKETBUF_ATTR_TTL, PACKETBUF_ATTR_BIT * 4 },
    { PACKETBUF_ATTR_EPACKET_ID, PACKETBUF_ATTR_BIT * 16 },
    STBROADCAST_ATTRIBUTES
    PACKETBUF_ATTR_LAST
};
static const struct packetbuf_attrlist runicast_attributes[] = {
    { PACKETBUF_ADDR_ESENDER, PACKETBUF_ADDRSIZE },
    { PACKETBUF_ADDR_SENDER, PACKETBUF_ADDRSIZE },
    { PACKETBUF_ATTR_HOPS, PACKETBUF_ATTR_BIT * 4 },
    { PACKETBUF_ATTR_FREQ, PACKETBUF_ATTR_BIT * 4 },
    { PACKETBUF_ATTR_EPACKET_ID, PACKETBUF_ATTR_BIT * 16 },
    RUNICAST_ATTRIBUTES
    PACKETBUF_ATTR_LAST
};

// Number of retransmissions allowed
static const uint8_t RUNICAST_MAX_RETX = 3;
static const clock_time_t STUBBORN_SEND_REPEATS = 3;

typedef struct
{
    // Make sure about is the first entry
    rimeaddr_t about;

    // This how far we belive the node to be, we want
// the shortest path to this node, so we record how far messages have taken to come from it to the current node.
uint8_t hops;

// The node to wards messages to if you want to send to the about node.
rimeaddr_t forward_to;

// The latest id of message that we have seen
uint16_t id;

} path_record_t;

// Struct used to ask other nodes for values, this is empty, as we simply use header attributes
typedef struct { } request_data_msg_t;

static inline nhopreq_conn_t * conncvt_runicast(struct runicast_conn * conn)
{
    return (nhopreq_conn_t *)conn;
}

static inline nhopreq_conn_t * conncvt_datareq_bcast(struct stbroadcast_conn * conn)
{
    return (nhopreq_conn_t *)
        (((char *)conn) - sizeof(struct runicast_conn));
}

// Argument structs
typedef struct
{
    nhopreq_conn_t * conn;
    rimeaddr_t target;
} delayed_reply_data_params_t;

typedef struct
{
    nhopreq_conn_t * conn;
    rimeaddr_t sender;
    rimeaddr_t target;
    uint8_t hops;
} delayed_forward_reply_params_t;

// Prototypes
static void delayed_reply_data(void * ptr);

static bool send_n_hop_data_request(
    nhopreq_conn_t * conn, rimeaddr_t const * originator,
    uint16_t message_id, uint8_t hop_limit, uint8_t hops);

static void send_reply(
    nhopreq_conn_t * conn, rimeaddr_t const * sender,
    rimeaddr_t const * eventual_target, uint8_t hops, void const * data);

// STUBORN BROADCAST
static void datareq_recv(struct stbroadcast_conn * c)
{
    nhopreq_conn_t * conn = conncvt_datareq_bcast(c);

    #ifdef NHRDPRINTF
        if (packetbuf_datalen() != sizeof(request_data_msg_t))
        {
            printf("nhopreq: Packet length not as expected\n");
        }

        #endif
}

97
rimeaddr_t originator, sender;

rimeaddr_copy(&originator, packetbuf_addr(PACKETBUF_ADDR_ESENDER));
rimeaddr_copy(&sender, packetbuf_addr(PACKETBUF_ADDR_SENDER));
const uint16_t message_id = packetbuf_attr(PACKETBUF_ATTR_EPACKET_ID);
const uint8_t hop_limit = (uint8_t)packetbuf_attr(PACKETBUF_ATTR_TTL);
const uint8_t hops = (uint8_t)packetbuf_attr(PACKETBUF_ATTR_HOPS);

// We don’t want to do anything if the message we sent has got back to ourselves
if (rimeaddr_cmp(&originator, &rimeaddr_node_addr))
{
  return;
}

bool respond = false;

// Check message has not been received before
path_record_t * record = (path_record_t *)map_get(&conn->mote_records, &originator);
if (record == NULL)
{
  printf("nhopreq: Not seen message from %s with id %u before.\n",
         addr2str(&originator), message_id);
  record = (path_record_t *)malloc(sizeof(path_record_t));
  rimeaddr_copy(&record->about, &originator);
  record->hops = hops;
  rimeaddr_copy(&record->forward_to, &sender);
  record->id = message_id;
  map_put(&conn->mote_records, record);
  respond = true;
}
else
{
  // Update path back to originator
  if (hops < record->hops)
  {
    printf("nhopreq: Updating forward path of %s to ", addr2str(&originator));
    printf("%s and %u hops\n", addr2str(&sender), hops);
    record->hops = hops;
    rimeaddr_copy(&record->forward_to, &sender);
  }

  // This is a newer message, so we need to respond to it.
  if (message_id > record->id)
  {
    printf("nhopreq: Seen a newer message from %s (%u), so respond.\n",
           addr2str(&originator), message_id);
    record->id = message_id;
    respond = true;
  }
}
if (respond)
{
  // Send values back to originator
  delayed_reply_data_params_t * p =
    (delayed_reply_data_params_t *)
malloc(sizeof(delayed_reply_data_params_t));

p->conn = conn;
rimeaddr_copy(&p->target, &originator);

// In time we will need to reply to this
ctimer_set(&conn->runicast_timer,
   21 * CLOCK_SECOND, &delayed_reply_data, p);

// Forward request onwards if we have not reached hop limit
if (hop_limit > 0)
{
   // Broadcast message onwards
   send_n_hop_data_request(
      conn, &originator, message_id,
      hop_limit - 1, hops + 1);
}

// Receive reply messages
static void runicast_recv(struct runicast_conn * c, rimeaddr_t const * from, uint8_t seqno)
{
   nhopreq_conn_t * conn = conncvt_runicast(c);

   // When receive message, forward the message on to the originator
   // if the final originator, do something with the value

   // Store a copy of the message
   char tmpBuffer[PACKETBUF_SIZE];
   memcpy(tmpBuffer, packetbuf_dataptr(), packetbuf_datalen());

   rimeaddr_t sender, target;

   rimeaddr_copy(&sender, packetbuf_addr(PACKETBUF_ADDR_ESENDER));
   rimeaddr_copy(&target, packetbuf_addr(PACKETBUF_ADDR_ERECEIVER));
   const uint8_t hops = (uint8_t)packetbuf_attr(PACKETBUF_ATTR_HOPS) + 1;

   void * msgdata = tmpBuffer;

   // If this node was the one who sent the message
   if (rimeaddr_cmp(&rimeaddr_node_addr, &target))
   {
      // The target node has received the required data, so provide it to the upper layer
      conn->callbacks->receive_fn(conn, &sender, hops, msgdata);
   }
   else
   {
      printf("nhopreq: Trying to forward data to: %s\n", addr2str(&target));

      send_reply(
         conn,
         &sender, // Source
         &target, // Destination
         hops,
         msgdata
      );
   }

   // Callbacks
   static const struct runicast_callbacks runicastCallbacks = { &runicast_recv, NULL, NULL };
   static const struct stbroadcast_callbacks datareqCallbacks = { &datareq_recv, NULL };

   static void delayed_reply_data(void * ptr)
{ delayed_reply_data_params_t * p = (delayed_reply_data_params_t *)ptr;

printf("nhopreq: Starting delayed send of node data to %s\n", addr2str(&p->target));

send_reply(
   p->conn,
   &rimeaddr_node_addr, // Source
   &p->target, // Destination
   0,
   NULL // NULL specifies node data should be generated and sent
);

// Need to free allocated parameter struct
free(ptr);
}

static void delayed_forward_reply(void * ptr)
{
    delayed_forward_reply_params_t * p = (delayed_forward_reply_params_t *)ptr;
    void const * data_to_send = (void *)(p + 1);

    send_reply(p->conn, &p->sender, &p->target, p->hops, data_to_send);

    // Need to free allocated parameter struct
    free(ptr);
}

static void send_reply(
    nhopreq_conn_t * conn, rimeaddr_t const * sender,
    rimeaddr_t const * eventual_target, uint8_t hops, void const * data)
{
    if (runicast_is_transmitting(&conn->ru))
    {
        printf("nhopreq: runicast is already transmitting, trying again in a few seconds\n");

        delayed_forward_reply_params_t * p = (delayed_forward_reply_params_t *)
            malloc(sizeof(delayed_forward_reply_params_t) + conn->data_size);

        p->conn = conn;
        rimeaddr_copy(&p->sender, sender);
        rimeaddr_copy(&p->target, eventual_target);
        p->hops = hops;

        void * data_dest = (void *)(p + 1);

        if (data == NULL)
        {
            // Call data get functions and store result in outwards bound packet
            conn->callbacks->data_fn(conn, data_dest);
        }
        else
        {
            // Copy the provided data
            memcpy(data_dest, data, conn->data_size);
        }

ctimer_set(&conn->forward_timer, random_time(2, 4, 0.1), &delayed_forward_reply, p);
    }
    else
    {
        printf("nhopreq: Trying to send reply to %s\n", addr2str(eventual_target));
    }
}
unsigned int packet_size = conn->data_size;

packetbuf_clear();
packetbuf_set_datalen(packet_size);
debug_packet_size(packet_size);
void * data_dest = packetbuf_dataptr();
memset(data_dest, 0, packet_size);

packetbuf_set_addr(PACKETBUF_ADDR_ESENDER, sender);
packetbuf_set_addr(PACKETBUF_ADDR_ERECEIVER, eventual_target);
packetbuf_set_attr(PACKETBUF_ATTR_HOPS, hops);

if (data == NULL)
{
    // Call data get functions and store result in outwards bound packet
    conn->callbacks->data_fn(conn, data_dest);
}
else
{
    // Copy the provided data
    memcpy(data_dest, data, conn->data_size);
}

path_record_t * record = (path_record_t *)map_get(&conn->mote_records, eventual_target);

if (record != NULL)
{
    runicast_send(&conn->ru, &record->forward_to, RUNICAST_MAX_RETX);
}
else
{
    printf("nhopreq: Failed to find a node to forward the data to \\
%s\n",
    addr2str(eventual_target));
}

static bool send_n_hop_data_request(
    nhopreq_conn_t * conn, rimeaddr_t const * originator,
    uint16_t message_id, uint8_t hop_limit, uint8_t hops)
{
    if (conn == NULL || originator == NULL || hop_limit == 0)
    {
        return false;
    }

    packetbuf_clear();
packetbuf_set_datalen(sizeof(request_data_msg_t));
    request_data_msg_t * msg = (request_data_msg_t *)packetbuf_dataptr();

    packetbuf_set_addr(PACKETBUF_ADDR_ESENDER, originator);
    packetbuf_set_addr(PACKETBUF_ADDR_SENDER, &rimeaddr_node_addr);
    packetbuf_set_attr(PACKETBUF_ATTR_EPACKET_ID, message_id);
    packetbuf_set_attr(PACKETBUF_ATTR_TTL, hop_limit);
    packetbuf_set_attr(PACKETBUF_ATTR_HOPS, hops);

    // Generate a random number between 2 and 4 to determine how
    // often we send messages
    clock_time_t random_send_time = random_time(2, 4, 0.1);
    clock_time_t send_limit = random_send_time * STUBBORN_SEND_REPEATS;

    printf("nhopreq: Starting sbcast every %lu/%lu second(s) for %lu/%lu seconds\n",
        random_send_time, CLOCK_SECOND, send_limit, CLOCK_SECOND);
    stbroadcast_send_stubborn(&conn->bc, random_send_time);
ctimer_set(&conn->datareq_stbroadcast_stop_timer, send_limit, &stbroadcast_cancel, &conn->bc);

return true;
}

bool nhopreq_start(nhopreq_conn_t * conn, uint8_t ch1, uint8_t ch2, unsigned int data_size, nhopreq_callbacks_t const * callbacks)
{
if (conn == NULL || callbacks == NULL ||
callbacks->data_fn == NULL || ch1 == ch2 || data_size == 0 ||
callbacks->receive_fn == NULL)
{
    return false;
}

// We need to set the random number generator here
random_init(*(uint16_t*)(&rimeaddr_node_addr));

stbroadcast_open(&conn->bc, ch1, &datareqCallbacks);
channel_set_attributes(ch1, stbroadcast_attributes);

runicast_open(&conn->ru, ch2, &runicastCallbacks);
channel_set_attributes(ch2, runicast_attributes);

conn->message_id = 1;
conn->callbacks = callbacks;
conn->data_size = data_size;
map_init(&conn->mote_records, &rimeaddr_equality, &free);

return true;
}

bool nhopreq_stop(nhopreq_conn_t * conn)
{
if (conn == NULL)
{
    return false;
}

timer_stop(&conn->runicast_timer);
timer_stop(&conn->forward_timer);
timer_stop(&conn->datareq_stbroadcast_stop_timer);
runicast_close(&conn->ru);
stbroadcast_close(&conn->bc);

// Free List
map_free(&conn->mote_records);

return true;
}

void nhopreq_request_info(nhopreq_conn_t * conn, uint8_t hops)
{
    send_n_hop_data_request(conn, &rimeaddr_node_addr, conn->message_id++, hops, 0);
}
B.4 N-Hop Flood

```c
#ifndef CS407_NHOPFLOOD
#define CS407_NHOPFLOOD

#include "sys/ctimer.h"
#include "net/rime.h"
#include "net/rime/broadcast.h"
#include "linked-list.h"
#include "map.h"
#include <stdbool.h>
#include <stdint.h>

struct nhopflood_conn;

// Callback function when data is received
typedef void (*nhopflood_recv_fn)(struct nhopflood_conn * conn,
                                 rimeaddr_t const * source, uint8_t hops, uint8_t previous_hops);

typedef struct nhopflood_conn
{
    struct broadcast_conn bc;
    nhopflood_recv_fn receive_fn;
    uint8_t current_id;
    uint8_t maxrx; // Maximum number of retransmits
    clock_time_t send_period;
    struct ctimer send_timer;
    linked_list_t packet_queue;
    map_t latest_message_seen;
} nhopflood_conn_t;

// Initialise n-hop data flooding.
bool nhopflood_start(nhopflood_conn_t * conn, uint8_t ch, nhopflood_recv_fn receive_fn,
                      clock_time_t send_period, uint8_t maxrx);

// Shutdown n-hop data flooding.
void nhopflood_stop(nhopflood_conn_t * conn);

// Send an n-hop data flood.
bool nhopflood_send(nhopflood_conn_t * conn, uint8_t hops);

#endif /*CS407_NHOPFLOOD*/
```

```c
#include "nhopflood.h"
#include "contiki.h"
#include "dev/leds.h"
#include "net/packetbuf.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "net/rimeaddr-helpers.h"
```
#include "random-range.h"
#include "debug-helper.h"

#ifdef NHOP_FLOOD_DEBUG
#define NHFDPRINTF(...) printf(__VA_ARGS__)
#else
#define NHFDPRINTF(...)
#endif

static void nhopflood_delayed_start_sending(void * ptr);

// The custom headers we use
static const struct packetbuf_attrlist flood_attributes[] = {
    { PACKETBUF_ADDR_ESENDER, PACKETBUF_ADDRSIZE },
    { PACKETBUF_ATTR_HOPS, PACKETBUF_ATTR_BIT * 4 },
    { PACKETBUF_ATTR_TTL, PACKETBUF_ATTR_BIT * 4 },
    { PACKETBUF_ATTR_EPACKET_ID, PACKETBUF_ATTR_BIT * 8 },
    BROADCAST_ATTRIBUTES
    PACKETBUF_ATTR_LAST
};

// Message Structs
typedef struct
    {
        rimeaddr_t sender;
        uint8_t id;
        uint8_t ttl;
        uint8_t hops;
        uint8_t retx;
        uint8_t data_length;
    } packet_details_t;

static inline void * packet_details_data(packet_details_t * details)
    {
        return (details + 1);
    }

// Creates a packet_details_t struct from the given information
static packet_details_t * alloc_packet_details(uint8_t id, uint8_t hops)
    {
        unsigned int data_length = packetbuf_datalen();
        packet_details_t * details =
            (packet_details_t *)malloc(sizeof(packet_details_t) + data_length);
        rimeaddr_copy(&details->sender, &rimeaddr_node_addr);
        details->id = id;
        details->ttl = hops;
        details->hops = 0;
        details->retx = 0;
        details->data_length = data_length;
        memcpy(packet_details_data(details), packetbuf_dataptr(), details->data_length);
        return details;
    }

// Creates a packet_details_t struct from the data in the packetbuf
static packet_details_t * packet_details_from_packetbuf(void)
    {

```c
unsigned int data_length = packetbuf_datalen();

packet_details_t * details =
    (packet_details_t *)malloc(sizeof(packet_details_t) + data_length);

rimeaddr_copy(&details->sender, packetbuf_addr(PACKETBUF_ADDR_ESENDER));
details->id = packetbuf_attr(PACKETBUF_ATTR_EPACKET_ID);
details->ttl = packetbuf_attr(PACKETBUF_ATTR_TTL);
details->hops = packetbuf_attr(PACKETBUF_ATTR_HOPS);

details->retx = 0;
details->data_length = data_length;

memcpy(packet_details_data(details), packetbuf_dataptr(), details->data_length);

return details;
```

```c
typedef struct
{
    rimeaddr_t from;
    uint8_t id;
    uint8_t hops;
} last_seen_t;

static inline nhopflood_conn_t * conncvt_broadcast(struct broadcast_conn * conn)
{
    return (nhopflood_conn_t *)conn;
}

// We receive a message from a neighbour
static void flood_message_recv(struct broadcast_conn * c, rimeaddr_t const * sender)
{
    // Get a pointer to the nhopflood_conn_t
    nhopflood_conn_t * conn = conncvt_broadcast(c);

    rimeaddr_t const * originator = packetbuf_addr(PACKETBUF_ADDR_ESENDER);
    const uint8_t packet_id = (uint8_t)packetbuf_attr(PACKETBUF_ATTR_EPACKET_ID);
    const uint8_t hops = (uint8_t)packetbuf_attr(PACKETBUF_ATTR_HOPS);
    const uint8_t ttl = (uint8_t)packetbuf_attr(PACKETBUF_ATTR_TTL);

    // Get the last seen entry for the end-point sender
    last_seen_t * last = map_get(&conn->latest_message_seen, originator);

    bool seenbefore = true;

    // Check if we have seen this packet before
    // Not seen from this node before
    if (last == NULL)
    {
        seenbefore = false;

        // We need to record that we have seen a packet from this sender
        last = (last_seen_t *)malloc(sizeof(last_seen_t));
        rimeaddr_copy(&last->from, originator);
        last->id = packet_id;
        last->hops = hops;

        map_put(&conn->latest_message_seen, last);
    }
    // Not seen this message before, but have received from this node before
    else if (last->id < packet_id ||
        (packet_id == 0 && last->id > 240) // Handle integer overflow, TODO: Improve this
    )
    {
```

105
seenbefore = false;
last->id = packet_id;
last->hops = hops;
}
// Seen before but this is from a shorter route
else if (last->id == packet_id && last->hops > hops)
{
  // Have seen before, but re-deliver
  conn->receive_fn(
    conn,
    originator,
    hops,
    last->hops
  );

  // We now need to update the hop count we have recorded
  linked_list_elem_t elem;
  for (elem = linked_list_first(&conn->packet_queue);
    linked_list_continue(&conn->packet_queue, elem);
    elem = linked_list_next(elem))
  {
    packet_details_t * data = (packet_details_t *)
      linked_list_data(&conn->packet_queue, elem);

    if (rimeaddr_cmp(&data->sender, originator) && data->id == last->id)
    {
      const uint8_t hops_diff = data->hops - hops;
      // Update the hops
      data->hops = hops;
      // As we have updated the hops we also need to update the TTL
      data->ttl = (hops_diff > data->ttl) ? 0 : data->ttl - hops_diff;
    }
  }

  last->hops = hops;
}
if (!seenbefore)
{
  conn->receive_fn(
    conn,
    originator,
    hops,
    0
  );

  // Add this packet to the queue if it needs to be forwarded
  // and we have not seen it before.
  if (ttl != 0)
  {
    packet_details_t * details = packet_details_from_packetbuf();
    linked_list_append(&conn->packet_queue, details);
  }
}
// Setup the Stubborn Broadcast Callbacks
static const struct broadcast_callbacks broadcastCallbacks = { &flood_message_recv };

static void nhopflood_delayed_start_sending(void * ptr)
{
  // Get the conn from the pointer provided
  nhopflood_conn_t * conn = (nhopflood_conn_t *)ptr;
packet_details_t * details = (packet_details_t *) linked_list_peek(&conn->packet_queue);

if (details != NULL)
{
    // Only send if the TTL is greater than 0
    if (details->ttl > 0)
    {
        // Create the memory for the packet
        packetbuf_clear();
        packetbuf_set_datalen(details->data_length);
        debug_packet_size(details->data_length);
        void * msg = packetbuf_dataptr();

        // Copy the packet to the buffer
        memcpy(msg, packet_details_data(details), details->data_length);

        // Set the header flags
        packetbuf_set_addr(PACKETBUF_ADDR_ESENDER, &details->sender);
        packetbuf_set_attr(PACKETBUF_ATTR_HOPS, details->hops + 1);
        packetbuf_set_attr(PACKETBUF_ATTR_TTL, details->ttl - 1);
        packetbuf_set_attr(PACKETBUF_ATTR_EPACKET_ID, details->id);

        // Send the packet using normal broadcast
        if (broadcast_send(&conn->bc))
        {
            // Increment the retransmission counter
            details->retx += 1;
        }

        // Remove the current queued packet as we have sent all we intend to send
        // Or the TTL is 0
        if (details->retx >= conn->maxrx || details->ttl == 0)
        {
            linked_list_pop(&conn->packet_queue);
        }

    }

    // Restart the ctimer
    ctimer_restart(&conn->send_timer);
}

// Initialise n-hop data flooding.
bool nhopflood_start(nhopflood_conn_t * conn, uint8_t ch, nhopflood_recv_fn receive_fn,
                    clock_time_t send_period, uint8_t maxrx)
{
    if (conn == NULL || receive_fn == NULL || ch == 0)
    {
        return false;
    }

    broadcast_open(&conn->bc, ch, &broadcastCallbacks);
    channel_set_attributes(ch, flood_attributes);
    conn->receive_fn = receive_fn;
    conn->current_id = 0;
    linked_list_init(&conn->packet_queue, &free);
    map_init(&conn->latest_message_seen, &rimeaddr_equality, &free);
    conn->send_period = send_period;
    conn->maxrx = maxrx;
    ctimer_set(&conn->send_timer, conn->send_period, &nhopflood_delayed_start_sending, conn);
return true;

// Shutdown n-hop data flooding.
void nhopflood_stop(nhopflood_conn_t * conn)
{
    if (conn != NULL)
    {
        ctimer_stop(&conn->send_timer);
        map_free(&conn->latest_message_seen);
        linked_list_free(&conn->packet_queue);
        broadcast_close(&conn->bc);
    }
}

// Register a request to send this nodes data n hops next round
bool nhopflood_send(nhopflood_conn_t * conn, uint8_t hops)
{
    if (conn == NULL)
    {
        //printf("The nhopflood_conn is null!\n");
        return false;
    }

    // When the number of hops to send to are 0, we can simply
    // do nothing
    if (hops == 0)
    {
        NHFPRINTF("nhopflood: Nowhere to send data to as hops=0\n");
        return true;
    }

    // Create packet details
    packet_details_t * details = alloc_packet_details(conn->current_id++, hops);

    // Record the details to be sent
    linked_list_append(&conn->packet_queue, details);

    NHFPRINTF("nhopflood: Added a packet to be sent, now \%u packets queued.\n",
               linked_list_length(&conn->packet_queue));
    return true;
}

B.5 Event Update

/*
 */
// Called to get the node's data into a given memory location
typedef void (*data_generation_fn)(void * data);

// Checks if the node's data differs from another piece of node data
typedef bool (*data_differs_fn)(void const * data1, void const * data2);

// Called when a packet is received
typedef void (*update_fn)(struct event_update_conn * c,
                        rimeaddr_t const * from, uint8_t distance, uint8_t previous_distance);

typedef struct event_update_conn
{
    // A connection to receive and send data
    nhopflood_conn_t fc;

    // The distance we will broadcast data out to
    uint8_t distance;

    // A function that retrieves the current data state
    data_generation_fn data_fn;

    // Check if data differs
    data_differs_fn differs_fn;

    // The size of the data
    size_t data_size;

    // The data that we last broadcasted
    void * data_loc;

    // This is how often we check that a piece of information has changed
    clock_time_t generate_period;

    // This function is called whenever we receive new information from a node
    update_fn update;

    // The chance that we will send an update anyway, even if the data
    // the same from the last check
    float chance;

    // The event timer
    struct ctimer check_timer;
} event_update_conn_t;

bool event_update_start(
    event_update_conn_t * conn, uint8_t ch, data_generation_fn data_fn,
    data_differs_fn differs_fn, size_t data_size, clock_time_t generate_period,
    update_fn update, float chance);

void event_update_stop(event_update_conn_t * conn);

void event_update_set_distance(event_update_conn_t * conn, uint8_t distance);

#endif /*CS407_EVENT_UPDATE_H*/

#include "eventupdate.h"
#include <stdlib.h>
#include <stdio.h>
#include "debug-helper.h"
#include "random-range.h"
static void flood_recv(struct nhopflood_conn * c,
    rimeaddr_t const * source, uint8_t hops, uint8_t previous_hops)
{
    // Prevent delivering details about the current node (that has been received via another node)
    if (!rimeaddr_cmp(source, &rimeaddr_node_addr))
    {
        event_update_conn_t * conn = (event_update_conn_t *)c;
        // Inform the client that an update has occurred
        conn->update(conn, source, hops, previous_hops);
    }
}

static void data_check(void * p)
{
    event_update_conn_t * conn = (event_update_conn_t *)p;
    // Check if we should force sending
    double rnd = random_range_double(0, 1);
    bool force = rnd <= conn->chance;
    if (force)
    {
        printf("eup: Force up \%d <= \%d\n", (int)(rnd * 10000), (int)(conn->chance * 10000));
    }
    bool has_changed = false;
    // Check to see if we have any data currently stored
    if (conn->data_loc != NULL)
    {
        // Allocate some memory for the current data
        void * tmp = malloc(conn->data_size);
        conn->data_fn(tmp);
        has_changed = force || conn->differs_fn(conn->data_loc, tmp);
        // Data has changed, we are about to send it
        // so record the new data
        if (has_changed)
        {
            free(conn->data_loc);
            conn->data_loc = tmp;
        }
        else
        {
            free(tmp);
        }
    }
    else
    {
        // No data currently stored
        // so set the stored data to the recently gained data
        conn->data_loc = malloc(conn->data_size);
        conn->data_fn(conn->data_loc);
        has_changed = true;
    }
    // Data has changed so send update message
    if (has_changed)
    {
        unsigned int packet_size = conn->data_size;
        packetbuf_clear();
    }
packetbuf_set_datalen(packet_size);
void * msg = packetbuf_dataptr();

// Set the data to send
memcpy(msg, conn->data_loc, conn->data_size);

nhopflood_send(&conn->fc, conn->distance);
}

// Reset timer
ctimer_reset(&conn->check_timer);
}

bool event_update_start(event_update_conn_t * conn, uint8_t ch, data_generation_fn data_fn, data_differs_fn differs_fn, size_t data_size, clock_time_t generate_period, update_fn update, float chance)
{
    if (conn != NULL && data_fn != NULL && data_size != 0 && generate_period != 0 && update != NULL)
    {
        nhopflood_start(&conn->fc, ch, &flood_recv, CLOCK_SECOND * 3, 3);
        conn->distance = 0;
        conn->data_fn = data_fn;
        conn->differs_fn = differs_fn;
        conn->data_size = data_size;
        conn->data_loc = NULL;
        conn->generate_period = generate_period;
        conn->update = update;
        conn->chance = chance;
        ctimer_set(&conn->check_timer, generate_period, &data_check, conn);
        return true;
    }
    return false;
}

void event_update_stop(event_update_conn_t * conn)
{
    if (conn != NULL)
    {
        ctimer_stop(&conn->check_timer);
        free(conn->data_loc);
        conn->data_loc = NULL;
        nhopflood_stop(&conn->fc);
    }
}

void event_update_set_distance(event_update_conn_t * conn, uint8_t distance)
{
    if (conn != NULL)
    {
        printf("eup: Set hops=%d\n", distance);
        conn->distance = distance;
        // Now that the distance has changed we need to trigger an update
        // the next chance we get. This is done by forgetting about the
        // last bit of data that we sent.
        free(conn->data_loc);
        conn->data_loc = NULL;
    }
B.6 Hop Data Manager

```c
#ifndef CS407_HOP_DATA_MANAGER_H
#define CS407_HOP_DATA_MANAGER_H

#include "containers/array-list.h"
#include "containers/map.h"
#include "net/rime.h"
#include "predicate-manager.h"
#include <string.h>

// This library is a component of predicate evaluation
// it is used to manage the data structures that contain
// node information a certain number of hops away.

typedef struct hop_data
{
    array_list_t maps; // List of maps
    unsigned int max_size;
} hop_data_t;

bool hop_manager_init(hop_data_t * hop_data);
void hop_manager_free(hop_data_t * hop_data);

bool hop_manager_record(hop_data_t * hop_data,
    uint8_t hops, void const * data, size_t data_length);
void hop_manager_remove(hop_data_t * hop_data,
    uint8_t hops, rimeaddr_t const * from);

void hop_manager_reset(hop_data_t * hop_data);

// Gets a map of data on nodes that a a certain number of hops away
map_t * hop_manager_get(hop_data_t * hop_data, uint8_t hops);

unsigned int hop_manager_length(hop_data_t * hop_data, var_elem_t const * variable);

#ifndef CONTAINERS_CHECKED
#define hop_manager_max_size(hop_data) 
    (hop_data != NULL ? hop_data->max_size : 0)
#else
#define hop_manager_max_size(hop_data) 
    ((hop_data)->max_size)
#endif

#endif /*CS407_HOP_DATA_MANAGER_H*/
```

```
#include "hop-data-manager.h"
#include "net/rimeaddr-helpers.h"
#include <stddef.h>
#include <stdlib.h>

static void free_hops_data(void * voiddata)
{
    map_t * data = (map_t *)voiddata;
```
bool hop_manager_init(hop_data_t * hop_data)
{
    if (hop_data != NULL)
    {
        hop_data->max_size = 0;
        return array_list_init(&hop_data->maps, &free_hops_data);
    }
    return false;
}

void hop_manager_free(hop_data_t * hop_data)
{
    if (hop_data != NULL)
    {
        hop_data->max_size = 0;
        array_list_free(&hop_data->maps);
    }
}

bool hop_manager_record(hop_data_t * hop_data, uint8_t hops, void const * data, size_t data_length)
{
    if (hop_data == NULL || data == NULL || data_length == 0)
    {
        return false;
    }
    map_t * map = hop_manager_get(hop_data, hops);
    void * stored = map_get(map, data);
    if (stored == NULL)
    {
        stored = malloc(data_length);
        if (stored == NULL)
        {
            return false;
        }
        memcpy(stored, data, data_length);
        map_put(map, stored);
        hop_data->max_size++;
    }
    else
    {
        // Update the stored data
        memcpy(stored, data, data_length);
    }
    return true;
}

void hop_manager_remove(hop_data_t * hop_data, uint8_t hops, rimeaddr_t const * from)
{
    if (hop_data != NULL)
    {
        map_t * map = hop_manager_get(hop_data, hops);
        array_list_remove(map->maps, (void*)from);
    }
}
map_remove(map, from);
}
}

void hop_manager_reset(hop_data_t * hop_data)
{
  if (hop_data != NULL)
  {
    hop_data->max_size = 0;
    array_list_free(&hop_data->maps);
  }
}

map_t * hop_manager_get(hop_data_t * hop_data, uint8_t hop)
{
  if (hop_data == NULL || hop == 0)
  {
    return NULL;
  }
  const unsigned int length = array_list_length(&hop_data->maps);

  // Map doesn't exist so create it
  if (length < hop)
  {
    unsigned int to_add;
    for (to_add = hop - length; to_add > 0; --to_add)
    {
      map_t * map = (map_t *)malloc(sizeof(map_t));
      map_init(map, &rimeaddr_equality, &free);
      array_list_append(&hop_data->maps, map);
    }
  }

  return (map_t *)array_list_data(&hop_data->maps, hop - 1);
}

unsigned int hop_manager_length(hop_data_t * hop_data, var_elem_t const * variable)
{
  if (hop_data == NULL || variable == NULL)
  {
    return 0;
  }
  unsigned int length = 0;
  uint8_t j;
  for (j = 1; j <= variable->hops; ++j)
  {
    length += map_length(hop_manager_get(hop_data, j));
  }
  return length;
}

B.7 Predicate Manager

#ifndef CS407_PREDICATE_MANAGER_H
#define CS407_PREDICATE_MANAGER_H

#include "containers/map.h"
#include "trickle.h"

114
// This library is a component of predicate evaluation
// it encapsulates storing and evaluating predicates.

// Record of the functions a user has registered
typedef struct
{
    uint8_t id;
    uint8_t return_type;
    void const * (* fn)(void const * ptr);
} function_details_t;

// The message that is sent when a predicate fails
typedef struct
{
    uint8_t predicate_id;
    uint8_t num_hops_positions;
    uint8_t data_length;
    uint8_t result;
} failure_response_t;

// Information about ‘Neighbour(x)’ information and how much data is held
typedef struct
{
    uint8_t hops;
    uint8_t var_id;
    uint8_t length;
} hops_position_t;

// Struct for the list of bytecode_variables
typedef struct
{
    uint8_t hops;
    uint8_t var_id;
} var_elem_t;

// Predicate record, predicate received over the network are stored in this
typedef struct
{
    uint8_t id; // Keep id as the first variable in the struct
    uint8_t variables_details_length;
    uint8_t bytecode_length;
    rimeaddr_t target; // rimeaddr_null indicates every node is targeted
    var_elem_t * variables_details;
    ubyte * bytecode;
} predicate_detail_entry_t;

typedef struct predicate_manager_conn;

typedef struct
{
    // Called when a predicate is added, removed or updated
    void (* updated)(struct predicate_manager_conn * conn);
}
// Called when a failure message is received.
// Only needs to be implemented by the base station.
void (*recv_response)(struct predicate_manager_conn *conn,
rimeaddr_t const *from, uint8_t hops);
} predicate_manager_callbacks_t;

typedef struct predicate_manager_conn
{
    struct trickle_conn tc;
    struct mesh_conn mc;
    map_t predicates; // Map of id to predicate_detail_entry_t
    rimeaddr_t basestation; // The target of predicate failure messages
    predicate_manager_callbacks_t const *callbacks;
} predicate_manager_conn_t;

bool predicate_manager_open(predicate_manager_conn_t *conn, uint16_t ch1, uint16_t ch2, rimeaddr_t const *basestation,
clock_time_t trickle_interval, predicate_manager_callbacks_t const *callbacks);

// Starts the process that reads serial input containing predicates provided
void predicate_manager_start_serial_input(predicate_manager_conn_t *conn);

void predicate_manager_close(predicate_manager_conn_t *conn);

bool predicate_manager_create(predicate_manager_conn_t *conn,
    uint8_t id, rimeaddr_t const *destination,
    ubyte const *bytecode, uint8_t bytecode_length,
    var_elem_t const *var_details, uint8_t var_details_length);

bool predicate_manager_cancel(predicate_manager_conn_t *conn,
    uint8_t id, rimeaddr_t const *destination);

#define predicate_manager_get_map(conn) ((conn) == NULL ? NULL : &(conn)->predicates)

uint8_t predicate_manager_max_hop(predicate_detail_entry_t const *pe);

// Actually evaluates the predicate using the provided information
bool evaluate_predicate(predicate_manager_conn_t *conn,
    node_data_fn data_fn, size_t data_size,
    function_details_t const *function_details, size_t functions_count,
    struct hop_data *hop_data,
    void const *all_neighbour_data, unsigned int nd_length,
    predicate_detail_entry_t const *pe);

// A debug helper function
void print_node_data(void const *ptr, function_details_t const *fn_details, size_t fn_count);

#endif /*CS407_PREDICATE_MANAGER_H*/
#include <limits.h>
#include <stdint.h>

#ifndef PREDICATE_MANAGER_DEBUG
    # define PMDPRINTF(...) printf(__VA_ARGS__)
#else
    # define PMDPRINTF(...)    
#endif

// Helper macro
#define UINT8_MIN 0

// Amount of time to wait for a mesh response
static const clock_time_t MESH_WAIT_PERIOD = 120 * CLOCK_SECOND;

// The process that handles serial input
PROCESS(predicate_input_process, "PredManager Read Input");

// Repurpose packetbuf attributes
#define PACKETBUF_ATTR_PREDICATE_ID PACKETBUF_ATTR_HOPS
#define PACKETBUF_ATTR_BYTECODE_LEN PACKETBUF_ATTR_ERELIABLE
#define PACKETBUF_ATTR_VAR_LEN PACKETBUF_ATTR_EPACKET_TYPE

// The custom headers we use
static const struct packetbuf_attrlist trickle_attributes[] = {
    { PACKETBUF_ATTR_PREDICATE_ID, PACKETBUF_ATTR_BYTE },
    { PACKETBUF_ATTR_BYTECODE_LEN, PACKETBUF_ATTR_BYTE },
    { PACKETBUF_ATTR_VAR_LEN, PACKETBUF_ATTR_BYTE },
    { PACKETBUF_ADDR_ERECEIVER, PACKETBUF_ADDRSIZE },
    TRICKLE_ATTRIBUTES
    PACKETBUF_ATTR_LAST
};

static void predicate_detail_entry_cleanup(void * item)
{
    predicate_detail_entry_t * entry = (predicate_detail_entry_t *)item;
    free(entry->variables_details);
    free(entry->bytecode);
    free(entry);
}

static bool predicate_id_equal(void const * left, void const * right)
{
    if (left == NULL || right == NULL)
        return false;
    uint8_t const * l = (uint8_t const *)left;
    uint8_t const * r = (uint8_t const *)right;
    return *l == *r;
}

static inline predicate_manager_conn_t * conncvt_trickle(struct trickle_conn * conn)
{
    return (predicate_manager_conn_t *)conn;
}

static inline predicate_manager_conn_t * conncvt_mesh(struct mesh_conn * conn)
{
    return (predicate_manager_conn_t *)(((char *)conn) - sizeof(struct trickle_conn));
}
static void trickle_recv(struct trickle_conn * tc)
{
    predicate_manager_conn_t * conn = conncvt_trickle(tc);

    void const * msg = packetbuf_dataptr();

    uint8_t predicate_id = (uint8_t)packetbuf_attr(PACKETBUF_ATTR_PREDICATE_ID);
    uint8_t bytecode_length = (uint8_t)packetbuf_attr(PACKETBUF_ATTR_BYTECODE_LEN);
    uint8_t variables_length = (uint8_t)packetbuf_attr(PACKETBUF_ATTR_VAR_LEN);

    // Get eventual destination from header
    rimeaddr_t const * target = packetbuf_addr(PACKETBUF_ADDR_ERECEIVER);

    if (bytecode_length == 0)
    {
        // There is no bytecode, so interpret this as a request to stop evaluating this predicate
        map_remove(&conn->predicates, &predicate_id);

        printf("PredMan: Remove %u\n", predicate_id);
    }
    else
    {
        // Add or update entry
        predicate_detail_entry_t * stored =
            (predicate_detail_entry_t *)map_get(&conn->predicates, &predicate_id);

        if (stored != NULL)
        {
            printf("PredMan: Update %u\n", predicate_id);

            // Re-allocate data structures if needed
            if (bytecode_length != stored->bytecode_length)
            {
                free(stored->bytecode);
                stored->bytecode = malloc(sizeof(ubyte) * bytecode_length);
            }

            if (variables_length != stored->variables_details_length)
            {
                free(stored->variables_details);
                stored->variables_details = malloc(sizeof(var_elem_t) * variables_length);
            }
        }
        else
        {
            printf("PredMan: Create %u\n", predicate_id);

            // Allocate memory for the data
            stored = malloc(sizeof(predicate_detail_entry_t));

            stored->id = predicate_id; // Set the key
            stored->bytecode = malloc(sizeof(ubyte) * bytecode_length);
            stored->variables_details = malloc(sizeof(var_elem_t) * variables_length);

            // Put data in the map
            map_put(&conn->predicates, stored);
        }

        // Update the target of this predicate
        rimeaddr_copy(&stored->target, target);

        // Pointer for bytecode variables
        var_elem_t const * variables = (var_elem_t const *)msg;
    }
}
// Create a pointer to the bytecode instructions stored in the message.
ubyte const * bytecode_instructions = (ubyte const *)(variables + variables_length);

// Update data
stored->bytecode_length = bytecode_length;
stored->variables_details_length = variables_length;
memcpy(stored->bytecode, bytecode_instructions, sizeof(ubyte) * stored->bytecode_length);
memcpy(stored->variables_details, variables, sizeof(var_elem_t) * stored->variables_details_length);
}

leds_off(LEDS_RED);

// Set the red led on to indicate that we are evaluating a predicate
map_elem_t elem;
for (elem = map_first(&conn->predicates);
    map_continue(&conn->predicates, elem);
    elem = map_next(elem))
{
    predicate_detail_entry_t const * pe =
        (predicate_detail_entry_t const *)map_data(&conn->predicates, elem);

    // Set the led to be red if this node will evaluate a predicate
    if (rimeaddr_cmp(&pe->target, &rimeaddr_node_addr) ||
        rimeaddr_cmp(&pe->target, &rimeaddr_null))
    {
        leds_on(LEDS_RED);
        break;
    }
}

// Call the updated callback
if (conn->callbacks->updated != NULL)
{
    conn->callbacks->updated(conn);
}

static const struct trickle_callbacks tc_callbacks = { &trickle_recv };

// Used to handle receiving predicate failure messages
static void mesh_rcv(struct mesh_conn * c, rimeaddr_t const * from, uint8_t hops)
{
    predicate_manager_conn_t * conn = conncvt_mesh(c);
    if (conn->callbacks->recv_response != NULL)
    {
        conn->callbacks->recv_response(conn, from, hops);
    }
}

static void mesh_timeout(struct mesh_conn * c)
{
    predicate_manager_conn_t * conn = conncvt_mesh(c);
    printf("PredMan: PF reply timedout\n");
}

static const struct mesh_callbacks mc_callbacks = { &mesh_rcv, NULL, &mesh_timeout };

bool predicate_manager_open(
    predicate_manager_conn_t * conn, uint16_t ch1, uint16_t ch2, rimeaddr_t const * basestation,
clock_time_t trickle_interval, predicate_manager_callbacks_t const * callbacks)
{
  if (conn != NULL && callbacks != NULL && basestation != NULL)
  {
    conn->callbacks = callbacks;
    rimeaddr_copy(&conn->basestation, basestation);
    map_init(&conn->predicates, &predicate_id_equal, &predicate_detail_entry_cleanup);
    trickle_open(&conn->tc, trickle_interval, ch1, &tc_callbacks);
    channel_set_attributes(ch1, trickle_attributes);
    mesh_open(&conn->mc, ch2, &mc_callbacks, MESH_WAIT_PERIOD);
    return true;
  }
  return false;
}

void predicate_manager_start_serial_input(predicate_manager_conn_t * conn)
{
  if (conn != NULL)
  {
    process_start(&predicate_input_process, (void *)conn);
  }
}

void predicate_manager_close(predicate_manager_conn_t * conn)
{
  if (conn != NULL)
  {
    // Shut down the serial input process if it is running
    if (process_is_running(&predicate_input_process))
    {
      process_exit(&predicate_input_process);
    }
    trickle_close(&conn->tc);
    mesh_close(&conn->mc);
    map_free(&conn->predicates);
  }
}

bool predicate_manager_create(predicate_manager_conn_t * conn, uint8_t id, rimeaddr_t const * destination,
                              ubyte const * bytecode, uint8_t bytecode_length,
                              var_elem_t const * var_details, uint8_t var_details_length)
{
  if (destination == NULL || bytecode == NULL || bytecode_length == 0 ||
      var_details == NULL || var_details_length == 0)
    return false;

  // Send the request message
  const unsigned int packet_size =
    (sizeof(ubyte) * bytecode_length) +
    (sizeof(var_elem_t) * var_details_length);
  if (packet_size > PACKETBUF_SIZE)
  {
    printf("PredMan: Packet too long\n");
    return false;
  }
packetbuf_clear();
packetbuf_set_datalen(packet_size);
void * msg = packetbuf_dataptr();

// Set eventual destination in header
packetbuf_set_addr(PACKETBUF_ADDR_ERECIEVER, destination);
packetbuf_set_attr(PACKETBUF_ATTR_PREDICATE_ID, id);
packetbuf_set_attr(PACKETBUF_ATTR_BYTECODE_LEN, bytecode_length);
packetbuf_set_attr(PACKETBUF_ATTR_VAR_LEN, var_details_length);

var_elem_t * msg_vars = (var_elem_t *)msg;
memcpy(msg_vars, var_details, sizeof(var_elem_t) * var_details_length);
ubyte * msg_bytecode = (ubyte *)(msg_vars + var_details_length);

#ifdef PREDICATE_MANAGER_DEBUG
// Debug check to make sure that we have done sane things!
if ((void *)(msg_bytecode + bytecode_length) - (void *)msg != packet_size)
{
    printf("PredMan: Failed copy got=%ld expected=%u!\n",
        (void *)(msg_bytecode + bytecode_length) - (void *)msg,
        packet_size);
}
#endif

memcpy(msg_bytecode, bytecode, sizeof(ubyte) * bytecode_length);
PMDPRINTF("PredMan: Sent %d\n", packet_size);

// We need to receive the predicate so we know of it
trickle_recv(&conn->tc);
trickle_send(&conn->tc);
return true;

/* predicate_manager_cancel */

bool predicate_manager_cancel(predicate_manager_conn_t * conn,
    uint8_t id, rimeaddr_t const * destination)
{
    if (conn == NULL || destination == NULL)
        return false;
    packetbuf_clear();
    packetbuf_set_datalen(1);

    // Set eventual destination in header
    packetbuf_set_addr(PACKETBUF_ADDR_ERECIEVER, destination);
    packetbuf_set_attr(PACKETBUF_ATTR_PREDICATE_ID, id);

    // Setting bytecode length to 0 indicates that this predicate should be removed
    packetbuf_set_attr(PACKETBUF_ATTR_BYTECODE_LEN, 0);
    packetbuf_set_attr(PACKETBUF_ATTR_VAR_LEN, 0);

    trickle_send(&conn->tc);
    return true;
}

/* predicate_manager_send_response */

bool predicate_manager_send_response(predicate_manager_conn_t * conn, hop_data_t * hop_data,
    predicate_detail_entry_t const * pe, void const * data, size_t data_size, size_t data_length,
    node_data_fn node_data, bool result)
if (conn == NULL || pe == NULL)
{
    return false;
}

const unsigned int packet_length =
    sizeof(failure_response_t) +
    sizeof(hops_position_t) * pe->variables_details_length +
    data_size * (data_length + 1);

if (packet_length > PACKETBUF_SIZE)
{
    printf("PredMan: Pred reply too long %u > %d\n", packet_length, PACKETBUF_SIZE);
    return false;
}

// TODO: Switch to using ruldolph1 or our own multipacket
// otherwise this cannot handle packets larger than 128 bytes
packetbuf_clear();
packetbuf_set_datalen(packet_length);
failure_response_t * msg = (failure_response_t *)packetbuf_dataptr();

msg->predicate_id = pe->id;
msg->num_hops_positions = pe->variables_details_length;
msg->data_length = data_length + 1;
msg->result = result ? 1 : 0;

hops_position_t * hops_details = (hops_position_t *)(msg + 1);

uint8_t i;
for (i = 0; i < msg->num_hops_positions; ++i)
{
    hops_details[i].hops = pe->variables_details[i].hops;
    hops_details[i].var_id = pe->variables_details[i].var_id;
    hops_details[i].length = hop_manager_length(hop_data, &pe->variables_details[i]);
}

void * msg_neighbour_data = (void *)(hops_details + pe->variables_details_length);

// Copy in this node’s data
node_data(msg_neighbour_data);

msg_neighbour_data = ((char *)msg_neighbour_data) + data_size;

// Copy in neighbour data
memcpy(msg_neighbour_data, data, data_size * data_length);

// Make sure we have a backup of the packet
// Just incase the receiver function clears it
char tmpBuffer[PACKETBUF_SIZE];
memcpy(tmpBuffer, msg, packet_length);

// Also have the sender receive the message
// This will cause the details to be printed out for analysis
mesh_rcv(&conn->mc, &rimeaddr_node_addr, 0);

// If the target is not the current node send the message
if (!rimeaddr_cmp(&conn->basestation, &rimeaddr_node_addr) && !result)
{
    packetbuf_clear();
    packetbuf_set_datalen(packet_length);
    msg = (failure_response_t *)packetbuf_dataptr();
}
// Copy in the packet backup
memcpy(msg, tmpBuffer, packet_length);

mesh_send(&conn->mc, &conn->basestation);

return true;

uint8_t predicate_manager_max_hop(predicate_detail_entry_t const * pe)
{
    if (pe == NULL)
    {
        return 0;
    }

    uint8_t max_hops = 0;
    uint8_t i;
    for (i = 0; i < pe->variables_details_length; ++i)
    {
        if (pe->variables_details[i].hops > max_hops)
        {
            max_hops = pe->variables_details[i].hops;
        }
    }

    return max_hops;
}

bool evaluate_predicate(predicate_manager_conn_t * conn,
        node_data_fn data_fn, size_t data_size,
        function_details_t const * function_details, size_t functions_count,
        hop_data_t * hop_data,
        void const * all_neighbour_data, unsigned int nd_length,
        predicate_detail_entry_t const * pe)
{
    unsigned int i;

    // Set up the predicate language VM
    init_pred_lang(data_fn, data_size);

    // Register the data functions
    for (i = 0; i < functions_count; ++i)
    {
        function_details_t const * fund = &function_details[i];
        register_function(fund->id, fund->fn, fund->return_type);
    }

    // Bind the variables to the VM
    for (i = 0; i < pe->variables_details_length; ++i)
    {
        // Get the length of this hop’s data
        // including all of the closer hop’s data length
        unsigned int length = hop_manager_length(hop_data, &pe->variables_details[i]);
        printf("PredMan: Binding vars: id=%d hop=%d len=%d\n",
                pe->variables_details[i].var_id, pe->variables_details[i].hops, length);

        bind_input(pe->variables_details[i].var_id, all_neighbour_data, length);
    }
bool result = evaluate(pe->bytecode, pe->bytecode_length);

predicate_manager_send_response(conn, hop_data,
pe, all_neighbour_data, data_size, nd_length, data_fn, result);

return result;
}

void print_node_data(void const * ptr, function_details_t const * fn_details, size_t fn_count)
{
    size_t i;
    for (i = 0; i != fn_count; ++i)
    {
        printf("%u=" fn_details[i].id);

        void const * data = fn_details[i].fn(ptr);

        switch (fn_details[i].return_type)
        {
        case TYPE_INTEGER:
            {
                nint x = *(nint const *)data;
                printf("%i", x);
            break;
        case TYPE_FLOATING:
            {
                double f = *(nfloat const *)data;
                printf("%f", f);
            break;
        default:
            printf("unk");
            break;
        
        if (i + 1 < fn_count)
        {
            printf(",");
        }
        }

    // Simple, no error checking to reduce firmware size
    #define HEX_CHAR_TO_NUM(ch) (((ch) >= '0' && (ch) <= '9') ? (ch) - '0' : (ch) - 'A')

    // From: http://www.techinterview.org/post/526339864/int-atoi-char-pstr
    static uint8_t myatoi(char const * str)
    {
        uint8_t result = 0;

        while (*str && *str >= '0' && *str <= '9')
        {
            result = (result * 10u) + (*str - '0');
            ++str;
        }

        return result;
    }

    PROCESS_THREAD(predicate_input_process, ev, data)
    {
        static predicate_manager_conn_t * conn;
        static predicate_detail_entry_t current;
        static unsigned int state; // The state in the automata parser

while (true)
{
    // Let others do work until we have a line to process
    // This prevents us stealing the CPU while doing no work
    PROCESS_YIELD_UNTIL(ev == serial_line_event_message);

    // Get the data from the line message
    char const * line = (char const *)data;
    const unsigned int length = strlen(line);

    PMDPRINTF("PredMan: line:'%s' (%u) in %d\n", line, length, state);

    switch (state)
    {
    // Initial state looking for start line
    case 0:
    {
        if (length == 1 && line[0] == '[' && line[1] == '\0')
        {
            PMDPRINTF("PredMan: Starting predicate input...\n");
            state = 1;
        }
    } break;

    // Read in the predicate id
    case 1:
    {
        unsigned int value = myatoi(line);

        if (value >= UINT8_MIN && value <= UINT8_MAX)
        {
            current.id = (uint8_t)value;
            state = 2;
        }
        else
        {
            PMDPRINTF("PredMan: going to error handler\n");
            state = 99;
            continue;
        }
    } break;

    // Read in the predicate target
    case 2:
    {
        char buffer[4];

        char const * position = strchr(line, '.');
        unsigned int first_length = position - line;

        memcpy(buffer, line, first_length);
        buffer[first_length] = '\0';

        // Before dot
        current.target.u8[0] = (uint8_t)myatoi(buffer);

        // After dot
        current.target.u8[1] = (uint8_t)myatoi(position + 1);
    } break;
}
state = 3;
}
break;

// Read in the variable ids or bytecode

case 3:
{
if (line[0] == 'b')
{
    PMDPRINTF("PredMan: processing bytecode\n");
    unsigned int bytecode_count = (length - 1) / 2;
    ubyte * new = malloc(sizeof(ubyte) * (bytecode_count + current.bytecode_length));
    memcpy(new, current.bytecode, sizeof(ubyte) * current.bytecode_length);
    free(current.bytecode);
    current.bytecode = new;
    ubyte * starting = current.bytecode + current.bytecode_length;
    // Start looking at characters after the first b
    char const * current_pair = line + 1;
    unsigned int i = 0;
    for (i = 0; i != bytecode_count; ++i)
    {
        starting[i] = HEX_CHAR_TO_NUM(current_pair[0]) * 16 + 
        HEX_CHAR_TO_NUM(current_pair[1]);
        current_pair += 2;
    }
    // Record the newly added bytecode
    current.bytecode_length += bytecode_count;
} else if (line[0] == 'v')
{
    PMDPRINTF("PredMan: processing variable details\n");
    var_elem_t * new = malloc(
        sizeof(var_elem_t) * (1 + current.variables_details_length));
    memcpy(new, current.variables_details,
        sizeof(var_elem_t) * current.variables_details_length);
    free(current.variables_details);
    current.variables_details = new;
    var_elem_t * to_store_at =
        current.variables_details + current.variables_details_length;
    char const * start = line + 1;
    char buffer[4];
    char const * position = strchr(start, '.');
    unsigned int first_length = position - start;
    memcpy(buffer, line, first_length);
    buffer[first_length] = '\0';
    // Before dot
    to_store_at->hops = (uint8_t)myatoi(buffer);
    // After dot
}
to_store_at->var_id = (uint8_t)myatoi(position + 1);

current.variables_details_length += 1;
} else if (length == 1 && line[0] == ']' && line[1] == '\0')
{
    if (current.bytecode_length == 0)
    {
        predicate_manager_cancel(conn, current.id, &current.target);
    } else
    {
        predicate_manager_create(conn,
            current.id, &current.target,
            current.bytecode, current.bytecode_length,
            current.variables_details, current.variables_details_length);
    }

    free(current.bytecode);
    free(current.variables_details);
    memset(&current, 0, sizeof(current));
    state = 0;
} else
{
    PMDPRINTF("PredMan: going to error handler\n");
    state = 99;
    continue;
}

} break;

// Error state
case 99:
{
    free(current.variables_details);
    free(current.bytecode);
    memset(&current, 0, sizeof(current));
    printf("PredMan: Error occured in parsing input\n");
    break;
}

default:
    printf("PredMan: Not sure what to do state=%d, line=%s\n", state, line);
    break;
}

exit:
    free(current.variables_details);
    free(current.bytecode);
    PROCESS_END();

B.8 PELP

#ifndef CS407_PELP_H
#define CS407_PELP_H

#include <stdbool.h>
#include <stdint.h>
#include "predlang.h"

#endif
#include "nhopreq.h"
#include "predicate-manager.h"
#include "hop-data-manager.h"

struct pelp_conn;

typedef void (* pelp_predicate_failed_fn)(struct pelp_conn * conn, rimeaddr_t const * from, uint8_t hops);

typedef struct pelp_conn
{
    nhopreq_conn_t nhr;
    predicate_manager_conn_t predconn;
    hop_data_t hop_data;

    rimeaddr_t const * sink;
    pelp_predicate_failed_fn predicate_failed;

    node_data_fn data_fn;
    size_t data_size;

    uint8_t max_comm_hops;
    uint8_t functions_count;

    function_details_t const * function_details;

    clock_time_t predicate_period;
} pelp_conn_t;

bool pelp_start(pelp_conn_t * conn, rimeaddr_t const * sink, node_data_fn data_fn, size_t data_size, pelp_predicate_failed_fn predicate_failed, function_details_t const * function_details, uint8_t functions_count, clock_time_t predicate_period);

void pelp_stop(pelp_conn_t * conn);

#endif /*CS407_PELP_H*/

#include "pelp.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "contiki.h"
#include "net/rime.h"
#include "lib/random.h"
#include "sys/node-id.h"

#include "dev/leds.h"
#include "dev/sht11-sensor.h"
#include "dev/light-sensor.h"

#include "containers/map.h"
#include "net/rimeaddr-helpers.h"
#include "predicate-manager.h"
#include "hop-data-manager.h"

#include "sensor-converter.h"
#include "debug-helper.h"

#ifdef PE_DEBUG
    # define PEDPRINTF(...) printf(__VA_ARGS__)
#endif
#else
  #define PEDPRINTF(...)
#endif

#define trickle_interval ((clock_time_t) 2 * CLOCK_SECOND)

#define NODE_DATA_INDEX(array, index, size) \
  (((char *)array) + ((index) * (size))

static inline pelp_conn_t * conncvt_nhopreq(nhopreq_conn_t * conn)
{
  return (pelp_conn_t *)conn;
}

static inline pelp_conn_t * conncvt_predicate_manager(predicate_manager_conn_t * conn)
{
  return (pelp_conn_t *)
    (((char *)conn) - sizeof(nhopreq_conn_t));
}

static inline pelp_conn_t * conncvt_hop_data(hop_data_t * conn)
{
  return (pelp_conn_t *)
    (((char *)conn) - sizeof(nhopreq_conn_t) - sizeof(predicate_manager_conn_t));
}

// Expose the node data function for nhopreq
static void nhopreq_data_fn(nhopreq_conn_t * conn, void * data)
{
  pelp_conn_t * pelp = conncvt_nhopreq(conn);
  pelp->data_fn(data);
}

static void receieved_data(nhopreq_conn_t * conn,
    rimeaddr_t const * from, uint8_t hops, void const * data)
{
  pelp_conn_t * pelp = conncvt_nhopreq(conn);
  hop_manager_record(&pelp->hop_data, hops, data, pelp->data_size);
}

static void pm_update_callback(predicate_manager_conn_t * conn)
{
  pelp_conn_t * pelp = conncvt_predicate_manager(conn);

  // The predicates that are being evaluated have changed,
  // we need to recalculate the maximum number of hops of
  // data that we ever need to ask for

  map_t const * predicate_map = predicate_manager_get_map(conn);
  pelp->max_comm_hops = 0;

  map_elem_t elem;
  for (elem = map_first(predicate_map);
    map_continue(predicate_map, elem);
    elem = map_next(elem))
  {
    predicate_detail_entry_t const * pe =
      predicate_detail_entry_t const *)
    map_data(predicate_map, elem);
    // We evaluate nodes targeted at us, or rimeaddr_null
if (rimeaddr_cmp(&pe->target, &rimeaddr_node_addr) ||
    rimeaddr_cmp(&pe->target, &rimeaddr_null))
{
    uint8_t local_max_hops = predicate_manager_max_hop(pe);

    if (local_max_hops > pelp->max_comm_hops)
    {
        pelp->max_comm_hops = local_max_hops;
    }
}

static void pm_predicate_failed(predicate_manager_conn_t * conn,
    rimeaddr_t const * from, uint8_t hops)
{
    pelp_conn_t * pelp = conncvt_predicate_manager(conn);

    // Pass the predicate failure message upwards
    pelp->predicate_failed(pelp, from, hops);
}

static const predicate_manager_callbacks_t pm_callbacks =
    { &pm_update_callback, &pm_predicate_failed };

static const nhopreq_callbacks_t nhopreq_callbacks =
    { &nhopreq_data_fn, &received_data };

PROCESS(pelp_process, "PELP Process");
PROCESS_THREAD(pelp_process, ev, data)
{
    static pelp_conn_t * pelp;
    static struct etimer et;
    static void * all_neighbour_data = NULL;

    PROCESS_EXITHANDLER(goto exit;)
    PROCESS_BEGIN();

    pelp = (pelp_conn_t *)data;

    // Wait for other nodes to initialize.
    etimer_set(&et, 20 * CLOCK_SECOND);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));

    while (true)
    {
        etimer_set(&et, pelp->predicate_period);
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));

        // Only ask for data if the predicate needs it
        if (pelp->max_comm_hops != 0)
        {
            PEDPRINTF("PELP: Starting request for %d hops of data...
", pelp->max_comm_hops);

            nhopreq_request_info(&pelp->nhr, pelp->max_comm_hops);

            // Get as much information as possible within a given time bound
            etimer_set(&et, 120 * CLOCK_SECOND);
            PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));

            PEDPRINTF("PELP: Finished collecting hop data.\n");

            const unsigned int max_size = hop_manager_max_size(&pelp->hop_data);

            // If we have received any data
            if (max_size > 0)
// Generate array of all the data
all_neighbour_data = malloc(pelp->data_size * max_size);

// Position in all_neighbour_data
unsigned int count = 0;

// Copy in neighbour's data into the correct location in
// the allocated block of memory
uint8_t i;
for (i = 1; i <= pelp->max_comm_hops; ++i)
{
    map_t * hop_map = hop_manager_get(&pelp->hop_data, i);
    for (elem = map_first(hop_map);
         map_continue(hop_map, elem);
         elem = map_next(elem))
    {
        void * mapdata = map_data(hop_map, elem);
        memcpy(
            NODE_DATA_INDEX(all_neighbour_data, count, pelp->data_size),
            mapdata, pelp->data_size);
        ++count;
    }
    PEDPRINTF("PELP: i=%d Count=%d/%d length=%d
",
        i, count, max_size, map_length(hop_map));
}

const unsigned int max_size = hop_manager_max_size(&pelp->hop_data);
map_t const * predicate_map = predicate_manager_get_map(&pelp->predconn);

// Evaluate every predicate targeted at this node
for (elem = map_first(predicate_map);
     map_continue(predicate_map, elem);
     elem = map_next(elem))
{
    predicate_detail_entry_t const * pe =
        (predicate_detail_entry_t const *)
        map_data(predicate_map, elem);
    if (rimeaddr_cmp(&pe->target, &rimeaddr_node_addr) ||
        rimeaddr_cmp(&pe->target, &rimeaddr_null))
    {
        evaluate_predicate(&pelp->predconn,
            pelp->data_fn, pelp->data_size,
            pelp->function_details, pelp->functions_count,
            &pelp->hop_data,
            all_neighbour_data, max_size, pe);
    }
}

// Free the allocated neighbour data
free(all_neighbour_data);
all_neighbour_data = NULL;

// We want to remove all the data we gathered,
// this is important to do so as if a node dies
// we do not want to keep using its last piece of data
224    // we want that lack of data to be picked up on.
225    hop_manager_reset(&pelp->hop_data);
226 }
227
228 exit:
229    (void)0;
230    PROCESS_END();
231 }

bool pelp_start(pelp_conn_t * conn,
                rimeaddr_t const * sink, node_data_fn data_fn, size_t data_size,
                pelp_predicate_failed_fn predicate_failed,
                function_details_t const * function_details, uint8_t functions_count,
                clock_time_t predicate_period)
{ 
    if (conn == NULL || predicate_failed == NULL || data_fn == NULL ||
        sink == NULL || data_size == 0)
    { 
        return false;
    }
    conn->sink = sink;
    conn->data_fn = data_fn;
    conn->data_size = data_size;
    conn->max_comm_hops = 0;
    conn->predicate_failed = predicate_failed;
    conn->function_details = function_details;
    conn->functions_count = functions_count;
    conn->predicate_period = predicate_period;
    hop_manager_init(&conn->hop_data);
    predicate_manager_open(&conn->predconn, 121, 126, sink, trickle_interval, &pm_callbacks);
    if (!nhopreq_start(&conn->nhr, 149, 132, conn->data_size, &nhopreq_callbacks))
    { 
        PEDPRINTF("PELP: nhopreq start function failed\n");
    }
    if (rimeaddr_cmp(sink, &rimeaddr_node_addr)) // Sink
    { 
        PEDPRINTF("PELP: Is the base station!\n");
        // As we are the base station we need to start reading the serial input
        predicate_manager_start_serial_input(&conn->predconn);
        leds_on(LEDS_BLUE);
    } 
    else 
    { 
        leds_on(LEDS_GREEN);
    }
    process_start(&pelp_process, (void *)conn);
    return true;
}

void pelp_stop(pelp_conn_t * conn)
{ 
    if (conn != NULL)
    { 
        process_exit(&pelp_process);
    }
B.9 PELE

```c
#ifndef CS407_PELE_H
#define CS407_PELE_H

#include <stdbool.h>

#include <stdint.h>

#include "predlang.h"

#include "net/eventupdate.h"

#include "predicate-manager.h"

#include "hop-data-manager.h"

struct pele_conn;

typedef bool (* pele_data_differs_fn)(void const * data1, void const * data2);

typedef void (* pele_predicate_failed_fn)(struct pele_conn * conn, rimeaddr_t const * from, uint8_t hops);

typedef struct pele_conn
{
    event_update_conn_t euc;
    predicate_manager_conn_t predconn;
    hop_data_t hop_data;
    rimeaddr_t const * sink;
    pele_predicate_failed_fn predicate_failed;
    node_data_fn data_fn;
    pele_data_differs_fn differs_fn;
    size_t data_size;
    uint8_t max_comm_hops;
    uint8_t functions_count;
    function_details_t const * function_details;
    clock_time_t predicate_period;
} pele_conn_t;

bool pele_start(pele_conn_t * conn,
                rimeaddr_t const * sink, node_data_fn data_fn, size_t data_size,
                pele_data_differs_fn differs_fn, pele_predicate_failed_fn predicate_failed,
                function_details_t const * function_details, uint8_t functions_count,
                clock_time_t predicate_period);

void pele_stop(pele_conn_t * conn);

#endif /*CS407_PELE_H*/
```

```
#include "pele.h"

#include <stdio.h>
```
#include <string.h>
#include <stdlib.h>

#include "contiki.h"
#include "lib/random.h"
#include "sys/node-id.h"

#include "dev/leds.h"
#include "dev/sht11-sensor.h"
#include "dev/light-sensor.h"

#include "containers/map.h"
#include "net/rimeaddr-helpers.h"
#include "predlang.h"
#include "sensor-converter.h"
#include "debug-helper.h"

#ifdef PE_DEBUG
#define PEDPRINTF(...) printf(__VA_ARGS__)
#else
#define PEDPRINTF(...) 
#endif

#define trickle_interval ((clock_time_t)2 * CLOCK_SECOND)
#define EVENT_CHECK_PERIOD ((clock_time_t)30 * CLOCK_SECOND)
#define CHANCE_OF_EVENT_UPDATE 0.01

#define NODE_DATA_INDEX(array, index, size) 
((char *)array) + ((index) * (size))

static inline pele_conn_t * conncvt_event_update(event_update_conn_t * conn) 
{
    return (pele_conn_t *)conn;
}

static inline pele_conn_t * conncvt_predicate_manager(predicate_manager_conn_t * conn) 
{
    return (pele_conn_t *)
    ((char *)conn) - sizeof(event_update_conn_t);
}

static inline pele_conn_t * conncvt_hop_data(hop_data_t * conn) 
{
    return (pele_conn_t *)
    ((char *)conn) - sizeof(event_update_conn_t) - sizeof(predicate_manager_conn_t);
}

static void receieved_data(event_update_conn_t * c, 
rimeaddr_t const * from, uint8_t hops, uint8_t previous_hops) 
{
    pele_conn_t * pele = conncvt_event_update(c);
    void * nd = packetbuf_dataptr();

    PEDPRINTF("PELE: Obtained information from %s hops:%u (prev:%d)\n", 
    addr2str(from), hops, previous_hops);

    // If we have previously stored data from this node at 
    // a different location, then we need to forget about that information
    if (previous_hops != 0 &
        hops != previous_hops) 
    {
        hop_manager_remove(&pele->hop_data, previous_hops, from);
    }

    hop_manager_record(&pele->hop_data, hops, nd, pele->data_size);
static void pm_update_callback(struct predicate_manager_conn * conn)
{
    pele_conn_t * pele = conncvt_predicate_manager(conn);
    map_t const * predicate_map = predicate_manager_get_map(conn);
    pele->max_comm_hops = 0;

    // We need to find and set the maximum distance of all predicates
    map_elem_t elem;
    for (elem = map_first(predicate_map);
         map_continue(predicate_map, elem);
         elem = map_next(elem))
        {
            predicate_detail_entry_t const * pe =
                (predicate_detail_entry_t const *)
                map_data(predicate_map, elem);

            uint8_t local_max_hops = predicate_manager_max_hop(pe);

            if (local_max_hops > pele->max_comm_hops)
                pele->max_comm_hops = local_max_hops;
        }

    // We need to tell event update this distance, so it knows
    // have far it must flood information when it changes
    event_update_set_distance(&pele->euc, pele->max_comm_hops);
}

static void pm_predicate_failed(predicate_manager_conn_t * conn,
                                 rimeaddr_t const * from, uint8_t hops)
{
    pele_conn_t * pele = conncvt_predicate_manager(conn);

    // Pass the failure message upwards
    pele->predicate_failed(pele, from, hops);
}

static const predicate_manager_callbacks_t pm_callbacks =
    { &pm_update_callback, &pm_predicate_failed };

PROCESS(pele_process, "PELE Process");
PROCESS_THREAD(pele_process, ev, data)
{
    static pele_conn_t * pele;
    static struct etimer et;
    static void * all_neighbour_data = NULL;

    PROCESS_EXITHANDLER(goto exit;)
    PROCESS_BEGIN();

    pele = (pele_conn_t *)data;

    PEPDPRINTF("PELE: Process Started.\n");

    // Wait for other nodes to initialize.
    etimer_set(&et, 20 * CLOCK_SECOND);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));

    while (true)
    {
    }
PEDPRINTF("PELE: Starting long wait...\n");

// This determines how often predicates are evaluated
etimer_set(&et, pele->predicate_period);
PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
PEDPRINTF("PELE: Wait finished!\n");

const unsigned int max_size = hop_manager_max_size(&pele->hop_data);

// Only ask for data if the predicate needs it
if (pele->max_comm_hops != 0 && max_size > 0) {
    // Generate array of all the data
    all_neighbour_data = malloc(pele->data_size * max_size);

    // Position in all_neighbour_data
    unsigned int count = 0;

    // Copy neighbour data into the correct location in the memory block
    uint8_t i;
    for (i = 1; i <= pele->max_comm_hops; ++i) {
        map_t * hop_map = hop_manager_get(&pele->hop_data, i);

        map_elem_t elem;
        for (elem = map_first(hop_map);
            map_continue(hop_map, elem);
            elem = map_next(elem)) {
            void * mapdata = map_data(hop_map, elem);

            memcpy(
                NODE_DATA_INDEX(all_neighbour_data, count, pele->data_size),
                mapdata, pele->data_size);
            ++count;
        }
    }

    map_t const * predicate_map = predicate_manager_get_map(&pele->predconn);

    // Evaluate all predicates targeted at this node
    map_elem_t elem;
    for (elem = map_first(predicate_map);
        map_continue(predicate_map, elem);
        elem = map_next(elem)) {
        predicate_detail_entry_t const * pe =
            (predicate_detail_entry_t const *)
            map_data(predicate_map, elem);

        if (rimeaddr_cmp(&pe->target, &rimeaddr_node_addr) ||
            rimeaddr_cmp(&pe->target, &rimeaddr_null)) {
            evaluate_predicate(&pele->predconn,
                pele->data_fn, pele->data_size,
                pele->function_details, pele->functions_count,
                &pele->hop_data,
                all_neighbour_data, max_size, pe);
        }
    }

    // Free the allocated neighbour data
    free(all_neighbour_data);
all_neighbour_data = NULL;
}

exit:
(void)0;
PROCESS_END();
}

bool pele_start(pеле_conn_t * conn,
    rimeaddr_t const * sink, node_data_fn data_fn, size_t data_size,
    pele_data_differs_fn differs_fn, pele_predicate_failed_fn predicate_failed,
    function_details_t const * function_details, uint8_t functions_count,
    clock_time_t predicate_period)
{
    if (conn == NULL || predicate_failed == NULL || data_fn == NULL ||
        sink == NULL || data_size == 0 || differs_fn == NULL)
    {
        return false;
    }

    conn->sink = sink;
    conn->data_fn = data_fn;
    conn->data_size = data_size;
    conn->differs_fn = differs_fn;
    conn->max_comm_hops = 0;
    conn->predicate_failed = predicate_failed;
    conn->function_details = function_details;
    conn->functions_count = functions_count;
    conn->predicate_period = predicate_period;
    hop_manager_init(&conn->hop_data);
    predicate_manager_open(&conn->predconn, 121, 126, sink, trickle_interval, &pm_callbacks);
    if (!event_update_start(
        &conn->euc, 149, data_fn, differs_fn,
        data_size, EVENT_CHECK_PERIOD, &received_data,
        CHANCE_OF_EVENT_UPDATE)
    )
    {
        PEDPRINTF("PELE: nhopreq start function failed\n");
    }
}

if (rimeaddr_cmp(sink, &rimeaddr_node_addr)) // Sink
{
    PEDPRINTF("PELE: Is the base station!\n");
    // As we are the base station we need to start reading the serial input
    predicate_manager_start_serial_input(&conn->predconn);
    leds_on(LEDS_BLUE);
} else
{
    leds_on(LEDS_GREEN);
}
process_start(&pele_process, (void *)conn);
return true;
}

void pele_stop(pеле_conn_t * conn)
{
if (conn != NULL)
{
    process_exit(&pele_process);
    hop_manager_free(&conn->hop_data);
    event_update_stop(&conn->euc);
    predicate_manager_close(&conn->predconn);
}

B.10 PEGP

ifndef CS407_PEGP_H
#define CS407_PEGP_H

#include <stdbool.h>
#include <stdint.h>

#include "predlang.h"
#include "predicate-manager.h"
#include "hop-data-manager.h"
#include "neighbour-aggregate.h"
#include "containers/unique-array.h"
#include "containers/map.h"

struct pegp_conn;

typedef void (* pegp_predicate_failed_fn)(struct pegp_conn * conn, rimeaddr_t const * from, uint8_t hops);

typedef struct pegp_conn
{
    tree_agg_conn_t aggconn;
    neighbour_agg_conn_t nconn;
    predicate_manager_conn_t predconn;
    hop_data_t hop_data;
    rimeaddr_t const * sink;
    pegp_predicate_failed_fn predicate_failed;
    node_data_fn data_fn;
    size_t data_size;
    //uint8_t max_comm_hops;
    struct ctimer ct_startup;

    /\_uint8_t functions_count;
    function_details_t const * function_details;
    /\ Map containing rimeaddr_pair_t
    unique_array_t neighbour_info;
    /\ Map containing node_data_t
    map_t received_data;
    /\ Used for simulating evaluating a predicate on a node
    rimeaddr_t pred_simulated_node;
    unsigned int pred_round_count;
}

struct ctimer ct_startup;
clock_time_t predicate_period;

} pegp_conn_t;

bool pegp_start(pegp_conn_t * conn,
        rimeaddr_t const * sink, node_data_fn data_fn, size_t data_size,
        pegp_predicate_failed_fn predicate_failed,
        function_details_t const * function_details, uint8_t functions_count,
        clock_time_t predicate_period);

void pegp_stop(pegp_conn_t * conn);

#endif /*CS407_PEGP_H*/

#include "pegp.h"
#include "contiki.h"
#include "net/rime.h"
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include "lib/sensors.h"
#include "dev/sht11.h"
#include "dev/sht11-sensor.h"
#include "lib/random.h"
#include "sys/node-id.h"
#include "dev/cc2420.h"
#include "led-helper.h"
#include "sensor-converter.h"
#include "debug-helper.h"

#ifdef PE_DEBUG
#define PEDPRINTF(...) printf(__VA_ARGS__)
#else
#define PEDPRINTF(...)
#endif

#define ROUND_LENGTH ((clock_time_t) 5 * 60 * CLOCK_SECOND)
#define TRICKLE_INTERVAL (clock_time_t)(2 * CLOCK_SECOND)

#define NODE_DATA_INDEX(array, index, size) 
    (((char *)array) + ((index) * (size)))
#define CNODE_DATA_INDEX(array, index, size) 
    (((char const *)array) + ((index) * (size)))

typedef struct
    {
        unsigned int round_count;
        unique_array_t list; // List of rimeaddr_t
    } aggregation_data_t;

typedef struct
    {
        uint8_t length;
        unsigned int round_count;
    } collected_data_t;

static inline pegp_conn_t * conncvt_tree_agg(tree_agg_conn_t * conn)
{
    return (pegp_conn_t *)conn;
static inline pegp_conn_t * conncvt_neighbour_agg(neighbour_agg_conn_t * conn)  
{
    return (pegp_conn_t *)
        (((char *)conn) - sizeof(tree_agg_conn_t));
}

static inline pegp_conn_t * conncvt_predicate_manager(predicate_manager_conn_t * conn)  
{
    return (pegp_conn_t *)
        (((char *)conn) - sizeof(tree_agg_conn_t) - sizeof(neighbour_agg_conn_t));
}

static void handle_neighbour_data(neighbour_agg_conn_t * conn, rimeaddr_pair_t const * pairs, unsigned int length, unsigned int round_count)  
{
    pegp_conn_t * pegp = conncvt_neighbour_agg(conn);
    for (i = 0; i < length; ++i)
    {
        unique_array_append_precheck(&pegp->neighbour_info, &pairs[i], rimeaddr_pair_clone);
    }
}

static void tree_agg_recv(tree_agg_conn_t * conn, rimeaddr_t const * source, void const * packet, unsigned int packet_length)  
{
    pegp_conn_t * pegp = conncvt_tree_agg(conn);

toggle_led_for(LEDS_GREEN, CLOCK_SECOND);

    collected_data_t const * msg = (collected_data_t const *)packet;

    unsigned int length = msg->length;

    void const * msgdata = (msg + 1); // Get the pointer after the message

    PEDPRINTF("PEGP: Adding \%u pieces of data in round \%u\n", length, msg->round_count);

    unsigned int i;
    for (i = 0; i < length; ++i)
    {
        // Get the data at the current index
        void const * item = CNODE_DATA_INDEX(msgdata, i, pegp->data_size);

        // Store this data
        void * stored = map_get(&pegp->received_data, item);

        if (stored == NULL)
        {
            stored = malloc(pegp->data_size);
            map_put(&pegp->received_data, stored);
        }

        memcpy(stored, item, pegp->data_size);
    }
static void tree_agg_setup_finished(tree_agg_conn_t * conn)
{
    pegp_conn_t * pegp = conncvt_tree_agg(conn);
    PEDPRINTF("PEGP: Setup finished\n");
    if (tree_agg_is_leaf(conn))
    {
        leds_on(LEDS_RED);
    }
    process_start(&send_data_process, (void *)pegp);
}

static void tree_agg_store_packet(tree_agg_conn_t * conn, void const * packet, unsigned int length)
{
    pegp_conn_t * pegp = conncvt_tree_agg(conn);
    PEDPRINTF("PEGP: Store packet\n");
    unique_array_t * data = &(aggregation_data_t *)packet->list;
    pegp->data_fn(packet);
    unique_array_append(data, packet);
collected_data_t const * msg = (collected_data_t const *)packet;
aggregation_data_t * conn_data = (aggregation_data_t *)conn->data;
conn_data->round_count = msg->round_count;
// We need to initialise the list as this is the first packet received
unique_array_init(&conn_data->list, &rimeaddr_equality, &free);
// Store the received data
tree_aggregate_update(conn, conn_data, packet, length);
}

// Write the data structure to the outbound packet buffer
static void tree_agg_write_data_to_packet(tree_agg_conn_t * conn,
void ** data, size_t * packet_length)
{
  pegp_conn_t * pegp = conncvt_tree_agg(conn);
  // Take all data, write a struct to the buffer at the start,
  // containing the length of the packet (as the number of node_data_t)
  // write the each one to memory
toggle_led_for(LED annunciator - BLUE, CLOCK SECOND);
aggregation_data_t * conn_data = (aggregation_data_t *)conn->data;
unsigned int length = unique_array_length(&conn_data->list);
*packet_length = sizeof(collected_data_t) + (pegp->data_size * length);
*data = malloc(*packet_length);
collected_data_t * msg = (collected_data_t *)*data;
msg->length = length;
msg->round_count = conn_data->round_count;
// Get the pointer after the message
void * msgdata = (msg + 1);
unsigned int i = 0;
unique_array_elem_t elem;
for (elem = unique_array_first(&conn_data->list);
unique_array_continue(&conn_data->list, elem);
  elem = unique_array_next(elem))
{
  void * original = unique_array_data(&conn_data->list, elem);
  memcpy(NODE_DATA INDEX(msgdata, i, pegp->data_size), original, pegp->data_size);
  ++i;
}
// Free the data here
unique_array_free(&conn_data->list);
}

static void pm_predicate_failed(predicate_manager_conn_t * conn,
rimeaddr_t const * from, uint8_t hops)
{
  pegp_conn_t const * pegp = conncvt_predicate_manager(conn);
  // Pass the simulated node as the node that this reponse is from
  pegp->predicate_failed(pegp, &pegp->pred_simulated_node, hops);
}
static const predicate_manager_callbacks_t pm_callbacks = { NULL, &pm Predicate Failed };
static const tree_agg_callbacks_t callbacks = {
    &tree_agg_recv, &tree_agg_setup_finished, &tree_aggregate_update,
    &tree_aggregate_own, &tree_agg_store_packet, &tree_agg_write_data_to_packet
};

static const neighbour_agg_callbacks_t neighbour_callbacks = {&handle_neighbour_data};

PROCESS_THREAD(send_data_process, ev, data)
{
    static struct etimer et;
    static uint8_t round_count;
    static pegp_conn_t * pegp;
    static size_t data_length;
    static collected_data_t * msg;
    PROCESS_EXITHANDLER(goto exit;)
    PROCESS_BEGIN();
    pegp = (pegp_conn_t *)data;
    // Allocate once to reduce number of calls to malloc
    data_length = sizeof(collected_data_t) + pegp->data_size;
    msg = (collected_data_t *)malloc(data_length);
    round_count = 0;
    // Periodically generate and send data
    while (true)
    {
        etimer_set(&et, ROUND_LENGTH);
        // Leaf nodes start tree aggregation
        if (tree_agg_is_leaf(&pegp->aggconn))
        {
            // We should be set up by now
            // Start sending data up the tree
            msg->round_count = round_count;
            msg->length = 1;
            // Get the pointer after the message that will contain the nodes data
            void * msgdata = (msg + 1);
            pegp->data_fn(msgdata);
            tree_agg_send(&pegp->aggconn, msg, data_length);
        }
        ++round_count;
    }
    exit:
    (void)0;
    free(msg); // Never called, saves firmware size
    PROCESS_END();
}

static pegp_conn_t * global_pegp_conn;

// This function returns data on the node we are pretending to be.
// We need to pretend to be a node when evaluating a predicate
// that was targeted to that node
static void pretend_node_data(void * data)
{
if (data != NULL)
{
    void * stored_data = map_get(
        &global_pegp_conn->received_data,
        &global_pegp_conn->pred_simulated_node);

    memcpy(data, stored_data, global_pegp_conn->data_size);
}
}

static void data_evaluation(pegp_conn_t * pegp)
{
    PEDPRINTF("PEGP: Start Eval\n");

    map_t const * predicate_map = predicate_manager_get_map(&pegp->predconn);

    map_elem_t elem;
    for (elem = map_first(predicate_map);
        map_continue(predicate_map, elem);
        elem = map_next(elem))
    {
        predicate_detail_entry_t const * pred =
            (predicate_detail_entry_t const *)
        map_data(predicate_map, elem);

        unique_array_t evaluate_over;
        unique_array_init(&evaluate_over, &rimeaddr_equality, &free);

        // When null we are targeting every node
        if (rimeaddr_cmp(&pred->target, &rimeaddr_null))
        {
            // Add all nodes that we know about
            unique_array_elem_t nielem;
            for (nielem = unique_array_first(&pegp->neighbour_info);
                 unique_array_continue(&pegp->neighbour_info, nielem);
                 nielem = unique_array_next(nielem))
            {
                rimeaddr_pair_t * pair = (rimeaddr_pair_t *)
                    unique_array_data(&pegp->neighbour_info, nielem);

                unique_array_append_precheck(&evaluate_over, &pair->first, rimeaddr_clone);
                unique_array_append_precheck(&evaluate_over, &pair->second, rimeaddr_clone);
            }
        }
        else
        {
            unique_array_append(&evaluate_over, rimeaddr_clone(&pred->target));
        }

        unique_array_elem_t eoelem;
        for (eoelem = unique_array_first(&evaluate_over);
             unique_array_continue(&evaluate_over, eoelem);
             eoelem = unique_array_next(eoelem))
        {
            rimeaddr_t const * current = (rimeaddr_t const *)
                unique_array_data(&evaluate_over, eoelem);

            // Copy in the simulated node
            rimeaddr_copy(&pegp->pred_simulated_node, current);

            // Get the maximum number of hops needed for this predicate
            const uint8_t max_hops = predicate_manager_max_hop(pred);
            hop_manager_init(&pegp->hop_data);

            // Array of nodes that have been seen and checked so far
unique_array_t seen_nodes;
unique_array_init(&seen_nodes, &rimeaddr_equality, &free);

// Start with the destination node
unique_array_append(&seen_nodes, rimeaddr_clone(current));

// Array of nodes that we need the neighbours for
unique_array_t target_nodes;
unique_array_init(&target_nodes, &rimeaddr_equality, &free);

// Start with the destination node
unique_array_append(&target_nodes, rimeaddr_clone(current));

// Array of nodes, we gathered this round
unique_array_t acquired_nodes;
unique_array_init(&acquired_nodes, &rimeaddr_equality, &free);

// Get the data for each hop level (essentially a depth first search)
uint8_t hops;
for (hops = 1; hops <= max_hops; ++hops)
{
    // For each node in the target nodes, get the immediate neighbours,
    unique_array_elem_t target;
    for (target = unique_array_first(&target_nodes);
        unique_array_continue(&target_nodes, target);
        target = unique_array_next(target))
    {
        rimeaddr_t * t =
            (rimeaddr_t *)unique_array_data(&target_nodes, target);

        // Go through the neighbours for the node
        unique_array_elem_t neighbours_elem;
        for (neighbours_elem = unique_array_first(&pegp->neighbour_info);
            unique_array_continue(&pegp->neighbour_info, neighbours_elem);
            neighbours_elem = unique_array_next(neighbours_elem))
        {
            // The neighbour found
            rimeaddr_pair_t * neighbours = (rimeaddr_pair_t *)
                unique_array_data(&pegp->neighbour_info, neighbours_elem);

            rimeaddr_t * neighbour = NULL;
            if (rimeaddr_cmp(&neighbours->first, t))
            {
                neighbour = &neighbours->second;
            }
            if (rimeaddr_cmp(&neighbours->second, t))
            {
                neighbour = &neighbours->first;
            }

            if (neighbour != NULL)
            {
                PEDPRINTF("PEGP: Eval: Checking neighbour %s\n",
                    addr2str(neighbour));

                // If the neighbour hasn't been seen before
                if (!unique_array_contains(&seen_nodes, neighbour))
                {
                    void * nd = map_get(&pegp->received_data, neighbour);

                    if (nd == NULL)
                    {
                        PEDPRINTF("PEGP: ERROR no info on %s\n",
                            addr2str(neighbour));
                    }
                }
            }
    }
else {
    // Add the node to the target nodes for the next round
    unique_array_append(&acquired_nodes, rimeaddr_clone(neighbor));

    hop_manager_record(&pegp->hop_data, hops, nd, pegp->data_size);
}
}
}
}

// Been through targets add them to the seen nodes
// This call will steal the memory from target_nodes and leave it empty
unique_array_merge(&seen_nodes, &target_nodes, NULL);

// Add in the acquired nodes
unique_array_merge(&target_nodes, &acquired_nodes, &rimeaddr_clone);

// Generate array of all the data
void * all_neighbour_data = NULL;

// Number of nodes we pass to the evaluation
const unsigned int max_size = hop_manager_max_size(&pegp->hop_data);

if (max_size > 0) {
    all_neighbour_data = malloc(pegp->data_size * max_size);

    // Position in all_neighbour_data
    unsigned int count = 0;

    uint8_t i;
    for (i = 1; i <= max_hops; ++i) {
        map_t * hop_map = hop_manager_get(&pegp->hop_data, i);

        map_elem_t aelem;
        for (aelem = map_first(hop_map);
            map_continue(hop_map, aelem);
            aelem = map_next(aelem)) {
            void const * mapdata = map_data(hop_map, aelem);

            memcpy(
                NODE_DATA_INDEX(all_neighbour_data, count, pegp->data_size),
                mapdata,
                pegp->data_size);
            ++count;
        }
    }

    PEDPRINTF("PEGP: Eval: i=%u Count=%d/%d len=%d\n",
                i, count, max_size, map_length(hop_map));
}

// Need to set the global conn, so that pretend_node_data has access to it
global_pegp_conn = pegp;

evaluate_predicate(&pegp->predconn,
                    pretend_node_data, pegp->data_size,
                    pegp->function_details, pegp->functions_count,
                    &pegp->hop_data,
                    all_neighbour_data, max_size, pred);
free(all_neighbour_data);

hop_manager_reset(&pegp->hop_data);
unique_array_free(&target_nodes);
unique_array_free(&seen_nodes);
unique_array_free(&acquired_nodes);
}

unique_array_free(&evaluate_over);
}

// Empty details received and let the next round fill them up
map_free(&pegp->received_data);
unique_array_free(&pegp->neighbour_info);
PEDPRINTF("PEGP: Round=%u\n", pegp->pred_round_count);
pegp->pred_round_count += 1;
}

PROCESS_THREAD(data_evaluation_process, ev, data)
{
    static struct etimer et;
    static pegp_conn_t * pegp;
    PROCESS_EXITHANDLER(goto exit;)
    PROCESS_BEGIN();

    pegp = (pegp_conn_t *) data;
    map_init(&pegp->received_data, &rimeaddr_equality, &free);

    while (true)
    {
        etimer_set(&et, pegp->predicate_period);
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));

        data_evaluation(pegp);
    }

    exit:
    (void)0;
    //map_free(&pege->received_data); // Don’t ever expect to reach this point
    PROCESS_END();
}

void pegp_start_delayed2(pegp_conn_t * conn)
{
    PEDPRINTF("PEGP: Starting Data Aggregation\n");
    tree_agg_open(&conn->aggconn,
        conn->sink, 140, 170, sizeof(aggregation_data_t), &callbacks);

    // If sink start the evaluation process to run in the background
    if (rimeaddr_cmp(&rimeaddr_node_addr, conn->sink))
    {
        process_start(&data_evaluation_process, (void *)conn);
    }
}

void pegp_start_delayed1(pegp_conn_t * conn)
{
    neighbour_aggregate_open(&conn->nconn,
        conn->sink, 121, 110, 150, &neighbour_callbacks);
bool pegp_start(pegp_conn_t * conn, 
  rimeaddr_t const * sink, node_data_fn data_fn, size_t data_size, 
  pegp_predicate_failed_fn predicate_failed, 
  function_details_t const * function_details, uint8_t functions_count, 
  clock_time_t predicate_period)
{
  if (conn == NULL || predicate_failed == NULL || data_fn == NULL || 
      sink == NULL || data_size == 0)
  {
    return false;
  }

  conn->sink = sink;
  conn->data_fn = data_fn;
  conn->data_size = data_size;
  conn->predicate_failed = predicate_failed;
  conn->pred_round_count = 0;
  conn->function_details = function_details;
  conn->functions_count = functions_count;
  conn->predicate_period = predicate_period;
  predicate_manager_open(&conn->predconn, 135, 129, sink, TRICKLE_INTERVAL, &pm_callbacks);
  if (rimeaddr_cmp(&rimeaddr_node_addr, sink))
  {
    predicate_manager_start_serial_input(&conn->predconn);
  }

  // Setup the map
  unique_array_init(&conn->neighbour_info, &rimeaddr_pair_equality, &free);

  // Wait for some time to let process start up and perform neighbour detect
  ctimer_set(&conn->ct_startup, 10 * CLOCK_SECOND, &pegp_start_delayed1, conn);
  return true;
}

void pegp_stop(pegp_conn_t * conn)
{
  if (conn != NULL)
  {
    process_exit(&data_evaluation_process);
    process_exit(&send_data_process);
    ctimer_stop(&conn->ct_startup);
    tree_agg_close(&conn->aggconn);
    neighbour_aggregate_close(&conn->nconn);
    unique_array_free(&conn->neighbour_info);
    predicate_manager_close(&conn->predconn);
  }
}
```c
#include <stdbool.h>
#include <stdint.h>
#include "predlang.h"
#include "predicate-manager.h"
#include "hop-data-manager.h"
#include "net/tree-aggregator.h"
#include "neighbour-aggregate.h"
#include "containers/unique-array.h"
#include "containers/map.h"

struct pege_conn;

typedef bool (* pege_data_differs_fn)(void const * data1, void const * data2);
typedef void (* pege_predicate_failed_fn)(struct pege_conn * conn, rimeaddr_t const * from, uint8_t hops);

typedef struct pege_conn
{
  tree_agg_conn_t aggconn;
  neighbour_agg_conn_t nconn;
  predicate_manager_conn_t predconn;
  rimeaddr_t const * sink;
  pege_predicate_failed_fn predicate_failed;
  node_data_fn data_fn;
  pege_data_differs_fn differs_fn;
  size_t data_size;
  uint8_t functions_count;
  function_details_t const * function_details;

  // Map containing rimeaddr_pair_t
  unique_array_t neighbour_info;

  // Map containing node_data_t
  map_t received_data;

  // Used for simulating evaluating a predicate on a node
  rimeaddr_t pred_simulated_node;
  unsigned int pred_round_count;
  struct ctimer ct_startup;

  clock_time_t predicate_period;
} pege_conn_t;

bool pege_start(pege_conn_t * conn, rimeaddr_t const * sink, node_data_fn data_fn, size_t data_size, pege_data_differs_fn differs_fn, pege_predicate_failed_fn predicate_failed, function_details_t const * function_details, uint8_t functions_count, clock_time_t predicate_period);

void pege_stop(pege_conn_t * conn);

#endif /*CS407_PEGE_H*/
```
```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include "lib/sensors.h"
#include "dev/sht11.h"
#include "dev/sht11-sensor.h"
#include "lib/random.h"
#include "sys/node-id.h"
#include "dev/cc2420.h"
#include "led-helper.h"
#include "sensor-converter.h"
#include "debug-helper.h"

#ifdef PE_DEBUG
#define PEDPRINTF(...) printf(__VA_ARGS__)
#else
#define PEDPRINTF(...) 
#endif

#define ROUND_LENGTH (clock_time_t)(5 * 60 * CLOCK_SECOND)
#define TRICKLE_INTERVAL (clock_time_t)(2 * CLOCK_SECOND)

#define NODE_DATA_INDEX(array, index, size) 
    (((char *)array) + ((index) * (size)))
#define CNODE_DATA_INDEX(array, index, size) 
    (((char const *)array) + ((index) * (size)))

typedef struct
{
    unsigned int round_count;
    unique_array_t list; // List of rimeaddr_t
} aggregation_data_t;

typedef struct
{
    uint8_t length;
    unsigned int round_count;
} collected_data_t;

static inline pege_conn_t * conncvt_tree_agg(tree_agg_conn_t * conn)
{
    return (pege_conn_t *)conn;
}

static inline pege_conn_t * conncvt_neighbour_agg(neighbour_agg_conn_t * conn)
{
    return (pege_conn_t *)(((char *)conn) - sizeof(tree_agg_conn_t));
}

static inline pege_conn_t * conncvt_predicate_manager(predicate_manager_conn_t * conn)
{
    return (pege_conn_t *)(((char *)conn) - sizeof(tree_agg_conn_t) - sizeof(neighbour_agg_conn_t));
}

PROCESS(data_evaluation_process, "Data eval");
PROCESS(send_data_process, "Send data process");

static void handle_neighbour_data(neighbour_agg_conn_t * conn,
    rimeaddr_pair_t const * pairs, unsigned int length, unsigned int round_count)

```
pege_conn_t * pege = conncvt_neighbour_agg(conn);

PEDPRINTF("PEGE: Handling neighbour data round=%u length=%u\n", round_count, length);

// When receiving neighbour data at the base station
// record it into the neighbour info list
unsigned int i;
for (i = 0; i < length; ++i)
{
    unique_array_append_precheck(&pege->neighbour_info, &pairs[i], rimeaddr_pair_clone);
}

// Sink received final set of data
static void tree_agg_recv(tree_agg_conn_t * conn,
                        rimeaddr_t const * source, void const * packet, unsigned int packet_length)
{
    pege_conn_t * pege = conncvt_tree_agg(conn);
    toggle_led_for(LEDS_GREEN, CLOCK_SECOND);

    // Extract data from packet buffer
    collected_data_t const * msg = (collected_data_t const *)packet;

    unsigned int length = msg->length;
    void const * msgdata = (msg + 1); // Get the pointer after the message

    PEDPRINTF("PEGE: Adding %u pieces of data in round %u\n", length, msg->round_count);

    unsigned int i;
    for (i = 0; i < length; ++i)
    {
        // Get the data at the current index
        void const * item = CNODE_DATA_INDEX(msgdata, i, pege->data_size);

        // Store this data
        void * stored = map_get(&pege->received_data, item);

        if (stored == NULL)
        {
            stored = malloc(pege->data_size);
            map_put(&pege->received_data, stored);
        }
        memcpy(stored, item, pege->data_size);
    }
}

static void tree_agg_setup_finished(tree_agg_conn_t * conn)
{
    pege_conn_t * pege = conncvt_tree_agg(conn);
    PEDPRINTF("PEGE: Setup finished\n");

    if (tree_agg_is_leaf(conn))
    {
        PEDPRINTF("PEGE: Is leaf starting data aggregation\n");
        leds_on(LEDS_RED);
    }

    // Start sending data once setup has finished
    process_start(&send_data_process, (void *)pege);
}
static void tree_aggregate_update(tree_agg_conn_t * tconn,
void * voiddata, void const * to_apply, unsigned int to_apply_length)
{
    pege_conn_t * pege = conncvt_tree_agg(tconn);
    PEDPRINTF("PEGE: Update local data\n");
toggle_led_for(LED_RED, CLOCK_SECOND);
unique_array_t * data = &((aggregation_data_t *)voiddata)->list;
collected_data_t const * data_to_apply = (collected_data_t const *)to_apply;
void const * msgdata = (data_to_apply + 1); // Get the pointer after the message
    
    // Add the receieved data to the temporary store
    unsigned int i;
    for (i = 0; i < data_to_apply->length; ++i)
    {
        void const * item = CNODE_DATA_INDEX(msgdata, i, pegp->data_size);
        if (!unique_array_contains(data, item))
        {
            void * tmp = malloc(pege->data_size);
            memcpy(tmp, item, pege->data_size);
            unique_array_append(data, tmp);
        }
    }

    // Add our own one hop data to the list
    static void tree_aggregate_own(tree_agg_conn_t * tconn, void * ptr)
    {
        pege_conn_t * pege = conncvt_tree_agg(tconn);
        PEDPRINTF("PEGE: Update local data with own data\n");
        unique_array_t * data = &((aggregation_data_t *)ptr)->list;
        // Allocate and fill in our data
        void * msg = malloc(pege->data_size);
        pege->data_fn(msg);
        unique_array_append(data, msg);
    }

    // Store an inbound packet to the datastructure
    static void tree_aggregate_store_packet(tree_agg_conn_t * conn,
            void const * packet, unsigned int length)
    {
        pege_conn_t * pege = conncvt_tree_agg(conn);
        PEDPRINTF("PEGE: Store Packet length=%u\n", length);
        collected_data_t const * msg = (collected_data_t const *)packet;
        aggregation_data_t * conn_data = (aggregation_data_t * )conn->data;
        conn_data->round_count = msg->round_count;
        // We need to initialise the list as this is the first packet received
        unique_array_init(&conn_data->list, &rimeaddr_equality, &free);
        // Store the received data
        tree_aggregate_update(conn, conn_data, packet, length);
    }
static void tree_agg_write_data_to_packet(tree_agg_conn_t * conn, void ** data, size_t * packet_length)
{
    pega_conn_t * pege = connvt_tree_agg(conn);

    // Take all data, write a struct to the buffer at the start,
    // containing the length of the packet (as the number of node_data_t)
    // write the each one to memory
    toggle_led_for(LED_B, CLOCK.Second);

    aggregation_data_t * conn_data = (aggregation_data_t *)conn->data;
    unsigned int length = unique_array_length(&conn_data->list);
    *packet_length = sizeof(collected_data_t) + (pege->data_size * length);
    *data = malloc(*packet_length);

    collected_data_t * msg = (collected_data_t *)*data;
    msg->length = length;
    msg->round_count = conn_data->round_count;

    PEDPRINTF("PEGE: Writing len=%d dlen=%d\n", msg->length, *packet_length);

    // Get the pointer after the message
    void * msgdata = (msg + 1);

    unsigned int i = 0;
    unique_array_elem_t elem;
    for (elem = unique_array_first(&conn_data->list);
         unique_array_continue(&conn_data->list, elem);
         elem = unique_array_next(elem))
    {
        void * original = unique_array_data(&conn_data->list, elem);
        memcpy(NODE_DATA_INDEX(msgdata, i, pege->data_size), original, pege->data_size);
        ++i;
    }

    // Free the data here
    unique_array_free(&conn_data->list);
}

static void pm_predicate_failed(predicate_manager_conn_t * conn, rimeaddr_t const * from, uint8_t hops)
{
    pega_conn_t * pege = connvt_predicate_manager(conn);

    // Pass the simulated node as the node that this reponse is from
    pege->predicate_failed(pege, &pege->pred_simulated_node, hops);
}

static const predicate_manager_callbacks_t pm_callbacks = { NULL, &pm_predicate_failed };

static const tree_agg_callbacks_t callbacks = {
    &tree_agg_recv, &tree_agg_setup_finished, &tree_aggregate_update,
    &tree_aggregate_own, &tree_agg_store_packet, &tree_agg_write_data_to_packet
};

static const neighbour_agg_callbacks_t neighbour_callbacks = {&handle_neighbour_data};

PROCESS_THREAD(send_data_process, ev, data)
{
    static struct etimer et;
    static uint8_t round_count;

    // Write the data structure to the outbout packet buffer
    static void tree_agg_write_data_to_packet(tree_agg_conn_t * conn, void ** data, size_t * packet_length)
    {
        pega_conn_t * pege = connvt_tree_agg(conn);

        // Take all data, write a struct to the buffer at the start,
        // containing the length of the packet (as the number of node_data_t)
        // write the each one to memory
        toggleLedFor(LED_B, CLOCK.Second);

        aggregation_data_t * conn_data = (aggregation_data_t *)conn->data;
        unsigned int length = unique_array_length(&conn_data->list);
        *packet_length = sizeof(collected_data_t) + (pege->data_size * length);
        *data = malloc(*packet_length);

        collected_data_t * msg = (collected_data_t *)*data;
        msg->length = length;
        msg->round_count = conn_data->round_count;

        PEDPRINTF("PEGE: Writing len=%d dlen=%d\n", msg->length, *packet_length);

        // Get the pointer after the message
        void * msgdata = (msg + 1);

        unsigned int i = 0;
        unique_array_elem_t elem;
        for (elem = unique_array_first(&conn_data->list);
             unique_array_continue(&conn_data->list, elem);
             elem = unique_array_next(elem))
        {
            void * original = unique_array_data(&conn_data->list, elem);
            memcpy(NODE_DATA_INDEX(msgdata, i, pege->data_size), original, pege->data_size);
            ++i;
        }

        // Free the data here
        unique_array_free(&conn_data->list);
    }

    static void pm_predicate_failed(predicate_manager_conn_t * conn, rimeaddr_t const * from, uint8_t hops)
    {
        pega_conn_t * pege = connvt_predicate_manager(conn);

        // Pass the simulated node as the node that this reponse is from
        pege->predicate_failed(pege, &pege->pred_simulated_node, hops);
    }

    static const predicate_manager_callbacks_t pm_callbacks = { NULL, &pm_predicate_failed };

    static const tree_agg_callbacks_t callbacks = {
        &tree_agg_recv, &tree_agg_setup_finished, &tree_aggregate_update,
        &tree_aggregate_own, &tree_agg_store_packet, &tree_agg_write_data_to_packet
    };

    static const neighbour_agg_callbacks_t neighbour_callbacks = {&handle_neighbour_data};

    PROCESS_THREAD(send_data_process, ev, data)
    {
        static struct etimer et;
        static uint8_t round_count;

        // Write the data structure to the outbout packet buffer
        static void tree_agg_write_data_to_packet(tree_agg_conn_t * conn, void ** data, size_t * packet_length)
        {
            pega_conn_t * pege = connvt_tree_agg(conn);

            // Take all data, write a struct to the buffer at the start,
            // containing the length of the packet (as the number of node_data_t)
            // write the each one to memory
            toggleLedFor(LED_B, CLOCK.Second);

            aggregation_data_t * conn_data = (aggregation_data_t *)conn->data;
            unsigned int length = unique_array_length(&conn_data->list);
            *packet_length = sizeof(collected_data_t) + (pege->data_size * length);
            *data = malloc(*packet_length);

            collected_data_t * msg = (collected_data_t *)*data;
            msg->length = length;
            msg->round_count = conn_data->round_count;

            PEDPRINTF("PEGE: Writing len=%d dlen=%d\n", msg->length, *packet_length);

            // Get the pointer after the message
            void * msgdata = (msg + 1);

            unsigned int i = 0;
            unique_array_elem_t elem;
            for (elem = unique_array_first(&conn_data->list);
                 unique_array_continue(&conn_data->list, elem);
                 elem = unique_array_next(elem))
            {
                void * original = unique_array_data(&conn_data->list, elem);
                memcpy(NODE_DATA_INDEX(msgdata, i, pege->data_size), original, pege->data_size);
                ++i;
            }

            // Free the data here
            unique_array_free(&conn_data->list);
        }

        static void pm_predicate_failed(predicate_manager_conn_t * conn, rimeaddr_t const * from, uint8_t hops)
        {
            pega_conn_t * pege = connvt_predicate_manager(conn);

            // Pass the simulated node as the node that this reponse is from
            pege->predicate_failed(pege, &pege->pred_simulated_node, hops);
        }

        static const predicate_manager_callbacks_t pm_callbacks = { NULL, &pm_predicate_failed };

        static const tree_agg_callbacks_t callbacks = {
            &tree_agg_recv, &tree_agg_setup_finished, &tree_aggregate_update,
            &tree_aggregate_own, &tree_agg_store_packet, &tree_agg_write_data_to_packet
        };

        static const neighbour_agg_callbacks_t neighbour_callbacks = {&handle_neighbour_data};

        PROCESS_THREAD(send_data_process, ev, data)
        {
            static struct etimer et;
            static uint8_t round_count;

            // Write the data structure to the outbout packet buffer
            static void tree_agg_write_data_to_packet(tree_agg_conn_t * conn, void ** data, size_t * packet_length)
            {
                pega_conn_t * pege = connvt_tree_agg(conn);

                // Take all data, write a struct to the buffer at the start,
                // containing the length of the packet (as the number of node_data_t)
                // write the each one to memory
                toggleLedFor(LED_B, CLOCK.Second);

                aggregation_data_t * conn_data = (aggregation_data_t *)conn->data;
                unsigned int length = unique_array_length(&conn_data->list);
                *packet_length = sizeof(collected_data_t) + (pege->data_size * length);
                *data = malloc(*packet_length);

                collected_data_t * msg = (collected_data_t *)*data;
                msg->length = length;
                msg->round_count = conn_data->round_count;

                PEDPRINTF("PEGE: Writing len=%d dlen=%d\n", msg->length, *packet_length);

                // Get the pointer after the message
                void * msgdata = (msg + 1);

                unsigned int i = 0;
                unique_array_elem_t elem;
                for (elem = unique_array_first(&conn_data->list);
                     unique_array_continue(&conn_data->list, elem);
                     elem = unique_array_next(elem))
                {
                    void * original = unique_array_data(&conn_data->list, elem);
                    memcpy(NODE_DATA_INDEX(msgdata, i, pege->data_size), original, pege->data_size);
                    ++i;
                }

                // Free the data here
                unique_array_free(&conn_data->list);
            }

            static void pm Predicate_failed(predicate_manager_conn_t * conn, rimeaddr_t const * from, uint8_t hops)
            {
                pega_conn_t * pege = connvt_predicate_manager(conn);

                // Pass the simulated node as the node that this reponse is from
                pege->predicate_failed(pege, &pege->pred_simulated_node, hops);
            }

            static const predicate_manager_callbacks_t pm_callbacks = { NULL, &pm_predicate_failed };

            static const tree_agg_callbacks_t callbacks = {
                &tree_agg_recv, &tree_agg_setup_finished, &tree_aggregate_update,
                &tree_aggregate_own, &tree_agg_store_packet, &tree_agg_write_data_to_packet
            };

            static const neighbour_agg_callbacks_t neighbour_callbacks = {&handle_neighbour_data};

            PROCESS_THREAD(send_data_process, ev, data)
            {
                static struct etimer et;
                static uint8_t round_count;
static void * current_data;
static void * previous_data;
static pege_conn_t * pege;
static size_t data_length;
static collected_data_t * msg;

PROCESS_EXITHANDLER(goto exit;)
PROCESS_BEGIN();

pege = (pege_conn_t *)data;

// Allocate once to reduce number of calls to malloc
data_length = sizeof(collected_data_t) + pege->data_size;
msg = (collected_data_t *)malloc(data_length);

current_data = malloc(pege->data_size);
previous_data = malloc(pege->data_size);

round_count = 0;

// Send data only if it has changed
while (true)
{
    etimer_set(&et, ROUND_LENGTH);

    // Find the current data
    pege->data_fn(current_data);

    // Usually we would have leaf nodes starting sending data back up the tree
    // instead any node may do so, but only if its data has changed.
    // However, for the first round, we only let the leaves do the sending to save on
    // energy
    if ((round_count == 0 && tree_agg_is_leaf(&pege->aggconn)) ||
        (round_count > 0 && pege->differs_fn(current_data, previous_data))
    )
    {
        // We should be set up by now
        // Start sending data up the tree
        msg->round_count = round_count;
        msg->length = 1;

        // Get the pointer after the message that will contain the nodes data and fill it
        void * msgdata = (msg + 1);
        memcpy(msgdata, current_data, pege->data_size);

        // Remember the changed data
        memcpy(previous_data, current_data, pege->data_size);

        tree_agg_send(&pege->aggconn, msg, data_length);
    }

    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));

    ++round_count;
}

exit:
(void)0;
// Don't ever expect to reach this point
#if 0
    free(msg);
    free(current_data);
    free(previous_data);
#endif
PROCESS_END();

static pege_conn_t * global_pege_conn;

// This function would typically just return the current nodes data
// however because we are evaluating a predicate from a different node
// we need to return that nodes data
static void pretend_node_data(void * data)
{
    if (data != NULL)
    {
        void * stored_data = map_get(
            &global_pege_conn->received_data,
            &global_pege_conn->pred_simulated_node);
        memcpy(data, stored_data, global_pege_conn->data_size);
    }
}

static void data_evaluation(pege_conn_t * pege)
{
    PEDPRINTF("PEGE: Eval: Beginning Evaluation\n");
    map_t const * predicate_map = predicate_manager_get_map(&pege->predconn);
    map_elem_t elem;
    for (elem = map_first(predicate_map);
         map_continue(predicate_map, elem);
         elem = map_next(elem))
    {
        predicate_detail_entry_t const * pred =
            (predicate_detail_entry_t const *)map_data(predicate_map, elem);
        unique_array_t evaluate_over;
        unique_array_init(&evaluate_over, &rimeaddr_equality, &free);

        // When null we are targeting every node
        if (rimeaddr_cmp(&pred->target, &rimeaddr_null))
        {
            // Add all nodes that we know about
            unique_array_elem_t nielem;
            for (nielem = unique_array_first(&pege->neighbour_info);
                 unique_array_continue(&pege->neighbour_info, nielem);
                 nielem = unique_array_next(nielem))
            {
                rimeaddr_pair_t * pair = (rimeaddr_pair_t *)
                    unique_array_data(&pege->neighbour_info, nielem);
                unique_array_append_precheck(&evaluate_over, &pair->first, rimeaddr_clone);
                unique_array_append_precheck(&evaluate_over, &pair->second, rimeaddr_clone);
            }
        }
        else
        {
            unique_array_append(&evaluate_over, rimeaddr_clone(&pred->target));
        }
    }

    unique_array_elem_t eoelem;
    for (eoelem = unique_array_first(&evaluate_over);
         unique_array_continue(&evaluate_over, eoelem);
         eoelem = unique_array_next(eoelem))
    {
        rimeaddr_t const * current = (rimeaddr_t const *)
            unique_array_data(&evaluate_over, eoelem);
        }
// Copy in the simulated node
rimeaddr_copy(&pege->pred_simulated_node, current);

// Get the maximum number of hops needed for this predicate
const uint8_t max_hops = predicate_manager_max_hop(pred);

hop_data_t hop_data;
hop_manager_init(&hop_data);

// Get the maximum number of hops needed for this predicate
const uint8_t max_hops = predicate_manager_max_hop(pred);

hop_data_t hop_data;
hop_manager_init(&hop_data);

// Array of nodes that have been seen and checked so far
unique_array_t seen_nodes;
unique_array_init(&seen_nodes, &rimeaddr_equality, &free);

// Start with the destination node
unique_array_append(&seen_nodes, rimeaddr_clone(current));

// Array of nodes that we need the neighbours for
unique_array_t target_nodes;
unique_array_init(&target_nodes, &rimeaddr_equality, &free);

// Start with the destination node
unique_array_append(&target_nodes, rimeaddr_clone(current));

// Array of nodes, we gathered this round
unique_array_t acquired_nodes;
unique_array_init(&acquired_nodes, &rimeaddr_equality, &free);

// Get the data for each hop level (essentially a depth first search)
uint8_t hops;
for (hops = 1; hops <= max_hops; ++hops)
{

    // For each node in the target nodes, get the immediate neighbours,
    unique_array_elem_t targetelem;
    for (targetelem = unique_array_first(&target_nodes);
         unique_array_continue(&target_nodes, targetelem);
         targetelem = unique_array_next(targetelem))
    {
        rimeaddr_t * t =
            (rimeaddr_t *)unique_array_data(&target_nodes, targetelem);

        // Go through the neighbours for the node
        unique_array_elem_t neighbours_elem;
        for (neighbours_elem = unique_array_first(&pege->neighbour_info);
             unique_array_continue(&pege->neighbour_info, neighbours_elem);
             neighbours_elem = unique_array_next(neighbours_elem))
        {
            // The neighbour found
            rimeaddr_pair_t * neighbours =
                unique_array_data(&pege->neighbour_info, neighbours_elem);

            rimeaddr_t * neighbour = NULL;

            if (rimeaddr_cmp(&neighbours->first, t))
            {
                neighbour = &neighbours->second;
            }
            if (rimeaddr_cmp(&neighbours->second, t))
            {
                neighbour = &neighbours->first;
            }

            if (neighbour != NULL)
            {
                PEDPRINTF("PEGE: Eval: Checking neighbour %s\n", addr2str(neighbour));

                // If the neighbour hasn’t been seen before
            }
        }
    }
}
if (!unique_array_contains(&seen_nodes, neighbour))
{
    void * nd = map_get(&pege->received_data, neighbour);

    if (nd == NULL)
    {
        PEDPRINTF("PEGE: ERROR: no info on %s\n", addr2str(neighbour));
    }
    else
    {
        // Add the node to the target nodes for the next round
        unique_array_append(&acquired_nodes, rimeaddr_clone(neighbour));
        hop_manager_record(&hop_data, hops, nd, pege->data_size);
    }
}

// Been through targets add them to the seen nodes
// This call will steal the memory from target_nodes and leave it empty
unique_array_merge(&seen_nodes, &target_nodes, NULL);

// Add in the acquired nodes
unique_array_merge(&target_nodes, &acquired_nodes, &rimeaddr_clone);

// Generate array of all the data
void * all_neighbour_data = NULL;

// Number of nodes we pass to the evaluation
const unsigned int max_size = hop_manager_max_size(&hop_data);

if (max_size > 0)
{
    all_neighbour_data = malloc(pege->data_size * max_size);

    // Position in all_neighbour_data
    unsigned int count = 0;

    uint8_t i;
    for (i = 1; i <= max_hops; ++i)
    {
        map_t * hop_map = hop_manager_get(&hop_data, i);
        array_list_elem_t aelem;
        for (aelem = map_first(hop_map);
            map_continue(hop_map, aelem);
            aelem = map_next(aelem))
        {
            void const * mapdata = map_data(hop_map, aelem);

            memcpy(
                NODE_DATA_INDEX(all_neighbour_data, count, pege->data_size),
                mapdata, pege->data_size);
            ++count;
        }

        PEDPRINTF("PEGE: Eval: i=%d Count=%d/%d len=%d\n",
                i, count, max_size, map_length(hop_map));
    }

    // We need to set the global data needed for pretend_node_data

global_pege_conn = pege;

evaluate_predicate(&pege->predconn,
    pretend_node_data, pege->data_size,
    pege->function_details, pege->functions_count,
    &hop_data,
    all_neighbour_data, max_size, pred);

free(all_neighbour_data);

hop_manager_reset(&hop_data);
unique_array_free(&target_nodes);
unique_array_free(&seen_nodes);
unique_array_free(&acquired_nodes);
}

unique_array_free(&evaluate_over);
}

// Empty details received and let the next round fill them up
// We do not clear received_data, as that is sent only when it changes
unique_array_free(&pege->neighbour_info);

pege->pred_round_count += 1;
}

PROCESS_THREAD(data_evaluation_process, ev, data)
{
    static struct etimer et;
    static pege_conn_t * pege;

    PROCESS_EXITHANDLER(goto exit;)
    PROCESS_BEGIN();

    pege = (pege_conn_t *) data;

    map_init(&pege->received_data, &rimeaddr_equality, &free);

    while (true)
    {
        etimer_set(&et, pege->predicate_period);
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));

        data_evaluation(pege);
    }

    exit:
    (void)0;

    //map_free(&pege->received_data); // Don't ever expect to reach this point
    PROCESS_END();
}

void pege_start_delayed2(pege_conn_t * conn)
{
    PEDPRINTF("PEGE: Starting Data Aggregation\n");

    tree_agg_open(&conn->aggconn,
        conn->sink, 140, 170, sizeof(aggregation_data_t), &callbacks);

    // If sink start the evaluation process to run in the background
    if (rimeaddr_cmp(&rimeaddr_node_addr, conn->sink))
    {
        process_start(&data_evaluation_process, (void *)conn);
    }
}
void pege_start_delayed1(pege_conn_t * conn)
{
    neighbour_aggregate_open(&conn->nconn,
        conn->sink, 121, 110, 150, &neighbour_callbacks);

ctimer_set(&conn->ct_startup, 80 * CLOCK_SECOND, &pege_start_delayed2, conn);
}

bool pege_start(pege_conn_t * conn,
    rimeaddr_t const * sink, node_data_fn data_fn, size_t data_size,
    pege_data_differs_fn differs_fn, pege_predicate_failed_fn predicate_failed,
    function_details_t const * function_details, uint8_t functions_count,
    clock_time_t predicate_period)
{
    if (conn == NULL || predicate_failed == NULL || data_fn == NULL ||
       sink == NULL || data_size == 0 || differs_fn == NULL)
    {
        return false;
    }

    conn->sink = sink;
    conn->data_fn = data_fn;
    conn->data_size = data_size;
    conn->differs_fn = differs_fn;
    conn->predicate_failed = predicate_failed;
    conn->pred_round_count = 0;
    conn->function_details = function_details;
    conn->functions_count = functions_count;
    conn->predicate_period = predicate_period;

    predicate_manager_open(&conn->predconn, 135, 129, sink, TRICKLE_INTERVAL, &pm_callbacks);

    if (rimeaddr_cmp(&rimeaddr_node_addr, sink))
    {
        PEDPRINTF("PEGE: We are sink node.\n");

        predicate_manager_start_serial_input(&conn->predconn);
    }

    // Setup the map
    unique_array_init(&conn->neighbour_info, &rimeaddr_pair_equality, &free);

    // Wait for some time to let process start up and perform neighbour detect
    ctimer_set(&conn->ct_startup, 10 * CLOCK_SECOND, &pege_start_delayed1, conn);

    return true;
}

void pege_stop(pege_conn_t * conn)
{
    if (conn != NULL)
    {
        process_exit(&data_evaluation_process);
        process_exit(&send_data_process);

ctimer_stop(&conn->ct_startup);

tree_agg_close(&conn->aggconn);
    neighbour_aggregate_close(&conn->nconn);
    unique_array_free(&conn->neighbour_info);
    predicate_manager_close(&conn->predconn);
    }


C References


