High-level abstractions for programming self-managing wireless sensor network applications

Jonathan Lewis, Alan Dearle
School of Computer Science Technical Report, University of St Andrews, Fife KY16 9SX, Scotland
{jon.lewis, alan.dearle}@st-andrews.ac.uk

Abstract

Wireless sensor networks are complex, distributed systems whose components are commonly expected to operate in the face of unforeseen changes to the environment. Sensor network applications need be self-aware to be capable of adapting in response to such changes. We address this requirement with a high-level, component-based language model and implementation. In *Insense*, a sensing system is modelled as a composition of software components that interact via channels that may be published to the network and used to identify services available to other networked components. The language permits an application to dynamically discover both network topology and channels that have been published for inter-node use. Self-configuring application components are thereby able to search for the service channels they require and access these by configuring the channel bindings. Self-healing is supported by an exception model permitting components to detect channel communication anomalies and through language mechanisms permitting applications to be dynamically reconfigured. We detail aspects of the Insense language implementation on the TMote Sky platform running Contiki and present the space requirements for sample application components. Simulation results are presented for a sample application to demonstrate its ability to autonomously configure and self-heal.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organisation**]: Computer Communication Networks – *Distributed Systems;* D.3.2 [**Programming Languages**]: Language Classification – *Specialized application languages*

General Terms

Design, Languages, Performance.

Keywords: Self-managing, Self-healing, In-network Service Discovery, Abstraction, Programming.

1. Introduction

Wireless sensor networks may be deployed in inaccessible locations where frequent administrator visits are not feasible. As such, these sensing systems are commonly expected to operate autonomously. Sensor network applications should thus be self-aware and capable of monitoring their status and taking actions to adapt and maintain system functionality in response to unforeseen changes to the environment.

Application design and implementation for the resource-constrained architectures commonly used in WSNs is traditionally very difficult. Developers are required to write distributed programs for highly resource constrained devices and deal with memory management and synchronization issues without high-level software engineering support. Furthermore, WSN applications must cater for node failure and unreliable radio communications.

Our main contribution in this paper is to present a high-level model of distributed sensing in WSNs. The model supports autonomic configuration of applications through a service-oriented approach based on inter-component communication over channels. In this approach, the components of a sensing system discover and access the required services by connecting to channels associated with other components in the network. *Self-healing* of applications is supported through an exception model that permits applications to detect channel communication anomalies indicating that a service provider is unreliable or no longer available. Language mechanisms support the healing process by permitting an application to be dynamically reconfigured. We present a language model and implementation to support these abstractions.

Insense [1] is a component-based programming language which has been developed to facilitate WSN application development by raising the abstraction level and thereby reduce the scope for programmer error. An Insense application is modelled as a composition of active, stateful components that interact via typed, directional, synchronous channels. Figure 1 depicts an Insense application in which a *TempSource* component instance *ts* sends a temperature measurement to a *Sink* component instance *sink* when it receives a timing event from a *Ticker* component.

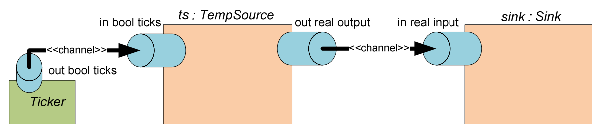

Figure 1. Timed Temperature Acquisition

The activity of the *TempSource* component is defined by repeatedly executing the following Insense code segment

```
receive tick from ticks
send tempReading() on output
```

and the activity of the *Sink* component is defined by repeatedly executing

```
receive datum from input
store(datum)
```

One of the fundamental design principles of Insense is that the complexity of distributed application design be bourne by the language implementation rather than by the programmer. In the example above, the *TempSource* component waits for a tick to arrive on its *ticks* channel before sending a temperature measurement on its *output* channel. Similarly the *Sink* component waits for a *datum* to arrive on its *input* channel before storing the *datum*. The channel construct thereby abstracts over communication and synchronization.

Communication over channels may also abstract over radio communication and routing decisions as discussed briefly below and in more detail in section 3 and section 4. The source and sink components depicted in Figure 1 may be located on the same node or on different nodes in the network and potentially separated by a number of intermediate nodes without any alteration to their program code being necessary. Insense thereby promotes the design of WSN applications comprising components that are agnostic of their deployment. As a result, a sensing system may be assembled by composing a number of software components which were written and compiled prior to their placement being decided.

Configuration of Insense applications is achieved by instantiating components and connecting and dissolving channel bindings. Component channels may be published to the network with a name that can be used to identify a *service* provided by that component. The language permits an application to dynamically discover both network topology and channels that have been published for inter-node use. Self-configuring application components are thereby able to search for the service channels they require and access these by configuring the channel bindings.

An exception model permits applications to detect channel communication anomalies which may result from unreliable radio links or node failure. Applications may thus be designed to support distributed *self-healing* in that their components may choose to switch service providers when a service channel is found to be unreliable. Alternatively, they may report errors to a monitoring component, which modifies inter-component bindings.

The remainder of the paper is organised as follows: Section 2 describes related work. The Insense language model is presented in section 3. Section 4 details some aspects of our implementation which runs on Contiki [2] and is tailored towards TMote Sky, a popular WSN node architecture with 48KB ROM and 10KB RAM. In section 5 we present the space requirements for sample application components and simulation results to demonstrate autonomous configuration and self-healing after detecting node failure. Section 6 contains our conclusions and outlines further work.

2. Related Work

This section briefly outlines approaches found in the literature for simplifying WSN application development and permitting dynamic reconfiguration of applications.

The authors of [3] present the *Pleiades* language which is a general purpose language that aims to simplify WSN application development. Pleiades extends the C language with constructs that permit developers to design a central WSN application rather than programs for each node. The central application is statically decomposed into nesC components [4] by the compiler. The compiled components may subsequently linked with the TinyOS library like any other TinyOS program [5]. Furthermore, the language permits node-local state to be named and accessed by the central program after it has been distributed over the network. As state is shared among both local components and networked components, coherency issues are to be expected. As a result, their approach requires locking and distributed deadlock detection and prevention mechanisms to guarantee a serial ordering in the face of concurrent execution.

The authors also present a user-directed concurrent loop construct which permits the central program to iterate over nodes in the network and execute the loop body concurrently on each node. This approach raises the abstraction level for WSN application development considerably from what we have observed elsewhere in the literature. It also appears to produce efficient code. However, it is not entirely clear how their approach would lend itself to applications in which different code is to be executed on a number of network nodes, for example, to support distributed database queries [6]. It is also unclear to what extent their approach deals either with unreliable radio communication or the dynamic reconfiguration of applications.

Interpreted languages such as TinyScript [7] for the Mate virtual machine [8] and an implementation of BASIC [9] based on a small uBasic [10] interpreter that was developed for Contiki are also gaining recognition in the WSN community. This may be due to their simplicity and ability to reduce code size. However, they would appear to

lack the expressibility for programming WSNs. For example, TinyScript does not provide support for multi-hop or unreliable radio transmissions and BASIC, in our experience, does not lend itself well to engineering complex applications. Our aim is to simplify application development by abstracting over low-level concerns while permitting the design of complex WSN applications.

Support for dynamic reconfiguration of WSN applications is, in our experience, not common place in the WSN community and provided at the operating system level rather than at the language level, if at all. Approaches in the literature centre on permitting program code to be replaced at runtime. A node's entire system image may be swapped for another [11] or smaller chunks of object code may be loaded dynamically either from a node's non-volatile store or after receiving it from the radio [12]. As sending system images or object code by radio may be costly in terms of energy consumption, some researchers promote the use of virtual machines and language interpreters to reduce code size and thereby aid reconfiguration by over-the-air programming [13][14]. While code replacement enables an application to be altered entirely, in this paper we focus on performing adaptation at the language level that is analogous to re-wiring a set of system components. However, the two approaches are complimentary.

3. Insense Language Model

In the following we present the model of the Insense language which is designed to abstract over the complexities of memory management, concurrency control, synchronisation, radio communication, routing decisions, and aims to decouple applications from the operating system and hardware.

3.1 Components and Procedures

Components are the unit of concurrent execution and the basic building blocks of Insense applications thus promoting a strong cohesion between the architectural description of a system and its implementation. Components are stateful, strongly encapsulated, and single threaded in order to prevent accidental race conditions. That is, components may contain updateable locations which can only be altered by the locus of control associated with a particular component instance.

The activity of a component is defined by its *behaviour* which is specified in a syntactic construct of the same name. A component's behaviour may be likened to a loop which starts to execute immediately after a component has been instantiated and continues until it is stopped either by the component itself or by some other component. Component instances are created by calling one of the component's constructors. Components may instantiate other components and alter the connection bindings between such components.

```
component TempSource presents ITempSource {
  constructor() {}
  behaviour {
    receive tick from ticks
    send tempReading() on output
  }
}
```

Figure 2. Temperature Source Component

The type of a component is represented by the set of channels (described in section 3.3) that it presents in its *interface* definition. The definition of the *TempSource* component depicted in Figure 1 is shown in Figure 2. The second line in the behaviour loop sends the result of calling the *tempReading*() procedure on its output *channel*. The *tempReading*() procedure is an example of a global procedure which, in this case, is provided by the runtime system as part of a standard library. Procedures may take any number of parameters and return a single value. In order to be able to determine an upper limit for the memory usage of programs, recursion is not permitted in Insense.

The interface for the component in Figure 2 may be defined as

```
type ITempSource is interface(
  in bool ticks ;
  out real output
)
```

and an instance of the component may be created by executing

```
ts = new TempSource()
```

3.2 Types

The language supports the following basic types: *boolean*, *byte*, *integer*, and *real*. An enumeration type *enum* and an immutable *string* type are also provided. The language supports the following constructed types: components, interfaces, structured record types, and arrays. Examples of component and interface declarations have been shown above. A structured record definition is shown in section 3.5.3 below. To statically determine memory requirements and simplify memory management, records may not contain other record types or references. The lack of pointers in records combined with shallow-copying when sending records over channels enforces the strong encapsulation of component locations described above.

Insense also supports an infinite union type called *any* [15]. Arbitrary values may be injected into the type *any* and a *project* operation permits values to be extracted. Arrays are the only collection type supported by the language and these must be initialized on declaration and their size must be defined using a literal to enable the required memory to be determined statically.

3.3 Channels

Channels are typed, directional, and synchronous. The type of a channel is defined by its direction, either *in* or *out,* and the type of its payload. Apart from component instances, all values in the language can be transferred over

channels of the appropriate type, including channels themselves. Arbitrary values can be sent on a channel by specifying the channel payload type to be *any*.

Attempting to send a datum on an unconnected channel results in the caller blocking until the datum has been sent to a connected receiver. Similarly, attempting to receive from an unconnected channel causes the caller to block until data can be received. These simple rules permit components to be dynamically re-wired without the need for complex synchronisation mechanisms. Thus, the semantics of the channel abstraction in Insense is akin to that of π-calculus [16].

Insense provides operations to *connect* and *disconnect* channels and to *send* and *receive* data on them. Assuming the existence of *sink,* an instance of the *Sink* component from Figure 1, we may connect *ts*, the *TempSource* component instance, to the *sink* as follows

```
connect ts.output to sink.input
```

Disconnection of the sink's *input* channel from all channels to which it is connected is achieved by

```
disconnect sink.input
```

The use of *send* and *receive* is illustrated in the behaviour loop in Figure 2 above. As the operations are synchronous the *TempSource* component in Figure 2 will block until it receives a tick on its *ticks* channel and will block until the datum sent in the *send* clause has been received by another component before continuing to execute its behaviour loop. Channels are thus used to abstract over both communication and synchronisation.

So far we have only considered the case where a channel is used to connect two component instances together. Insense permits a single channel to connect multiple component instances together. For example, a channel may be used to connect two source components to a single sink. This scenario is depicted in Figure 3 (a) in which two *TempSource* component instances *ts1* and *ts2* are both connected to the *input* channel of the *sink* component instance. While the semantics of sending and receiving data on *one-one* channels is deterministic, non-determinism is introduced for channels which connect multiple component instances. The *sink* component in Figure 3 (a) may receive a datum on its *input* channel from either one of the two sources *ts1* and *ts2*. The order in which data are received from these sources depends on their scheduling and it is not possible to determine the sender at the sink.

Figure 3 (b) depicts the scenario where a single source *output* channel is connected to multiple sinks. When *ts* sends a datum on its *output* channel it is received by one of the two sink components. If, for example, *sink1* is ready to receive a datum while *sink2* is not then the datum will be sent to *sink1*. If both sinks are ready to receive data then a fair decision is made. If neither sink is ready to receive then the source will block until one sink is ready to receive. The combination of *one-many* and *many-one* connection

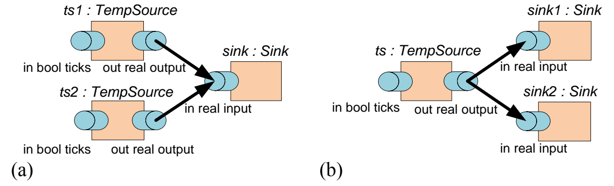

Figure 3. Many-one and one-many channel connection

patterns permits complex connection topologies to be specified in Insense.

Sometimes it is necessary to non-deterministically receive a value from one of a number of input channels. For example, a sink component may be required to receive different types of data from input channels connected to different sources. For this purpose, Insense provides a guarded, multi-channel *select* statement. The select operation tries to receive from a member of a set of input channels for which the guard conditions are met and input is available and executes a code block associated with that selection. An optional default clause is provided which permits the caller to continue execution even if no input is available or the guard conditions are not met.

A more detailed explanation of Insense channels, their semantics and of the non-deterministic select operation can be found in [1] and the correctness of the channel implementation for intra-node communication with respect to the desired semantics is demonstrated in [17].

3.4 Hardware Access

Access to parts of the hardware is provided through a combination of pre-defined components and procedures. The *tempReading()* procedure called from the *behaviour* in Figure 2 returns a temperature reading to the caller. The *Ticker* component provides access to the system clock and can be instructed to send ticks on specific channels at particular intervals. The following procedure call would be used to arrange ticks to be sent to the *ticks* channel associated with the *TempSource* component instance *ts* every 10 seconds.

```
periodicEnSchedule(ts.ticks, 10.0)
```

3.5 Inter-node Channels

In addition to intra-node inter-component communication, Insense permits channels to be used for inter-component communication when components are located on different nodes in the network. When used in this manner they can abstract over data marshalling and multi-hop radio communication primitives and support a channel-based, service-oriented model of distributed sensing in WSNs. This model promotes the design of deployment agnostic code and supports self-configuring and self-healing applications. Just as with intra-node channels, all values in the language apart from components can be sent on inter-node channels of the appropriate type including channels themselves. Attempting to send or

receive on unbound inter-node channels results in the caller blocking until a connection is established.

3.5.1 Exposure

Inter-node channels may be likened to *services* which can be dynamically discovered and accessed by configuring component connections to them. Specifically, channels can be exposed to the network using a *publish* operation. An example of the *publish* operation is shown below in which an incoming channel called *input* associated with a component instance *sink* is made visible to networked components.

```
publish sink.input as "sink"
```

The channel is associated with the string "sink", which we term the local channel name (LCN), and is made public to the network along with its associated channel direction and payload type. The LCN "sink" may be seen to represent a *sink* service that is provided to components executing on network nodes and accepts values of type *real* in this case. The LCN is required to be unique to a node and an error will result if the name "sink" has already been used locally to denote a public channel. LCNs are not required to be globally unique across a set of nodes. That is, multiple nodes may make channels globally accessible using the same name.

3.5.2 Binding

To utilise a channel published using a LCN, nodes may make use of an extended version of the already familiar *connect* operation. The *connect* operation is extended to permit either of the two channel end-points to be specified as a (node-address, LCN) pair. The *disconnect* operation is extended similarly. We illustrate the use of the extended *connect* operation in an example in which there are two nodes with radio addresses A and B. The *input* channel associated with a *sink* component instance on node B is published with the LCN "sink" using the *publish* operation as outlined above. Node A contains a *TempSource* component instance *ts*. The two component channels may be bound together by executing the following on node A.

```
connect ts.output to "sink" on B
```

Once a connection has been established any output written on the channel `ts.output` on node A will be transmitted via the radio for delivery to the input channel associated with the LCN "sink" on node B.

Channel bindings may be manipulated from any node in the network. If the following were executed on node A

```
publish ts.output as "tempSource"
```

then a binding between the *TempSource* and *Sink* components may be established from any node in the network by executing

```
connect "tempSource" on A to "sink" on B
```

Inter-node channel bindings are dissolved using the *disconnect* operation.

The inter-node channel model thereby supports remote configuration and reconfiguration of WSN applications by permitting the wiring between components to be adapted at runtime even when components are distributed over a network of nodes.

3.5.3 Discovery

In order to support self-configuration in WSNs an application must be able to discover information about the network during its deployed lifetime. Two operations are provided to permit nodes to discover the addresses of neighbouring nodes. The first operation

```
Address[] getNeighbours()
```

returns an array of addresses representing the direct neighbours of the node executing the operation. The *Address* type is declared as a byte array in the language and represents the address type used under Contiki. The second operation

```
Address[] getNeighboursOf( Address node )
```

discovers the addresses of direct neighbours for the specified node. Insense also provides an operation to determine the caller's node address

```
Address getNodeAddress()
```

and an operation to determine the minimum hop distance between two nodes specified by their addresses

```
integer getNumberHops(Address src,Address dest)
```

The latter operation returns the value -1 when no route between the specified nodes can be found. Programs may use these operations to discover information about network topology.

The two following operations permit application components to discover channels that have been published for inter-node communication by particular nodes. The two operations

```
PublicChannel[] getPublicChannels()
PublicChannel[] getPublicChannelsOf(Address a)
```

return an array of s*tructs* representing the channels published by this node and the specified node respectively. The public channel type is declared as

```
type PublicChannel is struct (
  Address address ;
  ChannelDirection direction ;
  string typerep ;
  string LCN
)
```

where the channel direction is an enumeration type given by

```
type ChannelDirection is enum (Out, In, Either)
```

and the *typerep* element is a human readable string representation of the channel's payload type. The language provides a *typerepof* operator which produces the string representation for any given value in the language. For example, if we declared a variable *temp* as

```
temp = 21.0
```

then the following operation would declare a string *s* and initialise it with the value “r”, the string representation for the type *real*.

```
s = typerepof temp
```

The last discovery operation

```
Address[] findNodesPublishing(
   ChannelDirection dir ,
   string typerep ,
   string LCN
)
```

combines the functionality of the other channel operations and returns a collection of addresses of nodes that have published a channel with the given LCN, direction, and payload type. If a LCN, direction, and type descriptor are used by the programmer to indicate a particular network service then the *findNodesPublishing* operation may be seen to permit the discovery of such services in the network. The implementation of *findNodesPublishing* also supports some degree of wild-carding which we intend to extend in the future to permit arbitrary service queries to be constructed. All the above operations which return an array value return a zero-length array in the case of failure.

We demonstrate the use of in-network service discovery with an example in which a *TempSource* component (as shown in Figure 2) is bound to a remote *Sink* component (as shown in Figure 5) by a third *TempBinder* component (as shown in Figure 6).

The configured application is depicted in Figure 4. The *Sink* component definition for node B, its instantiation and channel exposure are shown in Figure 5. The sink repeatedly waits for a datum to be sent to its *input* channel and prints the datum to the serial port using a polymorphic print procedure, *printAny*. In order to permit remote components to discover and *connect* to its *input* channel, the channel is published to the network and associated with the LCN "sink". The definition, instantiation, and en-scheduling of the *TempBinder* component are shown in Figure 6. The *TempBinder* constructs a local *TempSource* component instance and initialises its *ts* location with that instance. Due to strong encapsulation in components, the *TempSource* component is not visible from outside its creator. It follows that any binding and en-scheduling of the child component must be performed by the parent. To this end, the *TempSource* component's *tick* schedule is set by the *TempBinder* component in the constructor. The tick schedule for the *TempBinder* component is set after its construction in the *main* sequence.

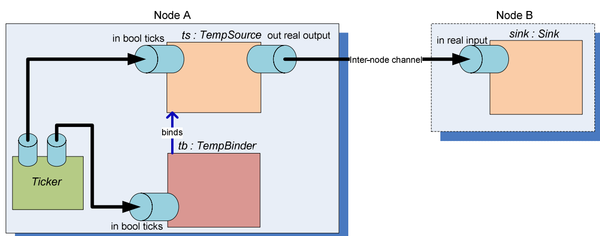

Figure 4. Self-configuring Application

```
type ISink is interface( in real input )

component Sink presents ISink {
  constructor() {}
  behaviour {
    receive data from input
    printAny(any("sink got reading "))
    printAny(any(data)) // log receipt of data
  }
}

sink = new Sink()
publish sink.input as "sink"
```

Figure 5. Passive Sink Component

```
type ITempBinder is interface( in bool ticks )

component TempBinder presents ITempBinder{
  ts = new TempSource()

  constructor(){
    periodicEnSchedule(ts.ticks, 60.0)
  }

  behaviour {
    receive tick from ticks //wait for schedule
    sinks = findNodesPublishing(In,"r","sink")
    if sinks.length > 0 then {// we found sinks
      // bind source to fist sink
      connect ts.output to "sink" on sinks[0]
      stop
    }
  }
}
tb = new TempBinder()
periodicEnSchedule(tb.ticks, 120.0)
```

Figure 6. TempBinder Program

The activity of the *TempBinder* component may be described as follows. After creation, the component waits for a tick to be received prior to searching for *sink* service providers in the network using the *findNodesPublishing* operation. Specifically, the component searches for networked components exporting an incoming channel with the LCN "sink" of payload type *real* (denoted by the string representation "r"). If no sinks are found then the next iteration of the behaviour loop is executed. The component is thereby able to periodically repeat its search until it finds a sink. If one or more sinks are found then the component connects its local *TempSource* component to the first sink in the list and stops. Meanwhile, the *TempSource* component is now connected and able to send temperature readings to the *sink* over the inter-node channel connection established by its parent.

The inter-node channel connecting the source to the sink abstracts over any multi-hop routing that may be necessary to establish and use the connection. That is, there may be any number of intermediate network nodes between the source and the sink which route both data as well as service requests and responses to their respective destinations.

3.5.4 Exception Model

The exception model in Insense permits applications to detect failing connections and deal with uncertainties while

establishing remote bindings and communicating over inter-node channels. Specifically, components can be designed to detect whether a channel operation has either succeeded, failed, or whether the outcome of the operation is unknown. For this purpose, Insense defines a fixed number of exceptions that may be thrown when operating on inter-node channels.

Four inter-node channel operations may throw exceptions in Insense. First, the *publish* operation throws the exception *DuplicateLCNException* when the specified LCN is already in use on the node executing the *publish* operation. Second, the *connect* operation throws the exceptions:

- *ChannelsUnknownException* when a channel name cannot be found on a remote node;
- *IncompatibleChannelsException* when the channel types are incompatible;
- *NodesUnreachableException* when a route from the caller to one or more nodes hosting the public channels cannot be found;
- *BindStatusUnknownException* when routes to the destinations are found, but confirmations of the channel binding status are not received.

Third, the exceptions *NodesUnreachableException* and *BindStatusUnknownException* may be thrown by the *disconnect* operation with the same semantic interpretation as for the *connect* operation. Finally, the *send* operation may throw the following exceptions when sending on an *output* channel:

- *NodesUnreachableException* when no routes to the destination channels can be found – the *output* channel is implicitly disconnected from all inputs;
- *SendStatusUnknownException* when a route can be found, and a datum is sent, but a confirmation is not received from the remote node.

Insense permits programmers to define exception handlers to handle particular exceptions. The language supports a *try-except* clause that is similar to the *try-catch* clause in Java. In contrast to exception handling in Java, the occurrence of exceptions in Insense programs is ignored unless the operation is within a *try*-block. That is, execution is allowed to continue optimistically even when one of the exceptions above occurs. Thus, for example, when knowledge of channel connectivity and reliable data transfer from one component to another is not required the programmer may simply choose to omit a *try-except* clause for the relevant *send* operation. In Insense, a *try*-block must be followed by at least one matching *except*-block. When an exception is thrown by an operation in a *try*-block execution immediately jumps to the matching *except*-block or to the instruction following the last *except*-block when no matching exception handler is defined.

The use of *try* and *except* is illustrated in Figure 7 which depicts a variation of the example from Figure 4 above in which an error channel connects the *TempSource* to the *TempBinder* on node A. A second *sink* component instance is also deployed on node C. The definition of the *TempSource* component and its interface is shown in Figure 8. The *TempSource* component in Figure 8 only differs slightly from the one shown in Figure 2. First, tracing information has been added which is later used to illustrate a Cooja simulation. The major difference is that it uses a *try-except* clause to catch exceptions when attempting to send a datum to the sink. The component is thereby capable of sending a notification on its *error* channel when it detects that its link to the *sink* is broken as indicated by the *NodesUnreachableException*. The exception *SendStatusUnknownException* is only used for trace purposes meaning that occasional data loss is ignored. The application components could trivially be adapted to permit data to be resent in the event of the exception *SendStatusUnknownException* if a greater reliability were required.

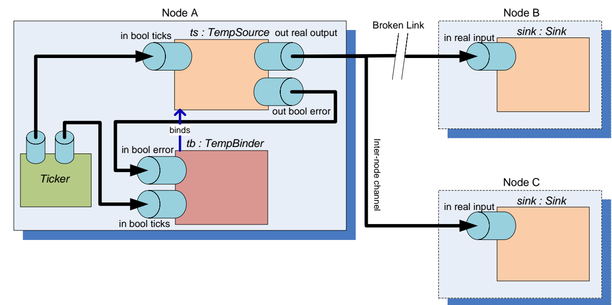

Figure 7. A Self-healing Application

```
type ITempSource is interface(
  in bool ticks ;
  out real output ;
  out bool error
)

component TempSource presents ITempSource {
  constructor() {}
  behaviour {
    receive tick from ticks
    try {
      send tempReading() on output
      printAny(any("ts: send ok"))
    } except NodesUnreachableException {
      printAny(any("ts: send unreachable"))
      send true on error
    } except SendStatusUnknownException {
      printAny(any("ts: send unknown"))
    }
  }
}
```

Figure 8. Detection of Broken Links

```
type ITempBinder is interface(
  in bool ticks ;
  in bool error
)

component TempBinder presents ITempBinder{
  ts = new TempSource()

  constructor(){
    periodicEnSchedule(ts.ticks, 60.0)
    connect ts.error to error
  }
  behaviour {
    receive tick from ticks
    sinks = findNodesPublishing(In,"r","sink")
    for i = 0 .. sinks.length-1 do {
      try {
        // try to bind the source to a sink
        connect ts.output to "sink" on sinks[i]
        printAny(any("tb: bound"))
        // bind succeeded, wait for error
        receive oops from error
        printAny(any("tb: link broken"))
      } except NodesUnreachableException {
          printAny(any("tb: connect failed"))
          // connect has failed, try next sink
      } except BindStatusUnknownException {
          printAny(any("tb: connect unknown"))
          // may be bound, still try next sink
      }
      printAny(any("tb: try next sink"))
    }
  }
}
tb = new TempBinder()
periodicEnSchedule(tb.ticks, 120.0)
```

Figure 9. Reliable Binding and Self-healing

The *TempBinder* component in Figure 9 differs in three aspects from the one in Figure 6. Similarly to Figure 8, trace information has been added to the code. Secondly, the component iterates over the *sinks* array in its *behaviour* and tries to *connect* its *TempSource* component to a sink reliably.

If the status of the *connect* operation is reported as having failed or as unknown, the component tries to *connect* to the next sink in its *sinks* array. Lastly, when an inter-node binding to a sink is established the component no longer *stops* its behaviour. Instead, it waits for an error signal from the *TempSource* component signifying that the inter-node connection has been lost. If such an error signal is received, the component tries to re-establish a connection to a different sink, starting with any untried sinks in the array. If the component is unsuccessful in binding to these *cached* sinks, the *behaviour* loop resumes and initiates a new service discovery and binding phase.

The use of exceptions in combination with service discovery and binding mechanisms may thereby be seen to support *self-configuration* and *self-healing*.

3.5.5 Semantic Differences

There are necessarily some differences in the semantics of sending on inter-node channels as opposed to intra-node channels. First, nodes may fail, be rebooted, and radio links may be unreliable or lost completely causing inter-node channels to fail. As a consequence messages may not be delivered to the intended receiver. In such cases an exception is thrown to the sender in order to permit applications to deal with failures of communication links.

A second difference is in inter-node channels connected to multiple inputs. The runtime system attempts to send the datum to each connected input channel in succession until it receives a message to confirm receipt (an ACK message) from a receiver. As one or more ACKs may be lost the datum may be received by multiple remote components.

3.5.6 Deployment Agnostic Code

The inter-node channel abstraction in Insense enables the composition of WSN applications from deployment-agnostic components. The agnosticism pertains to their design which does not require knowledge of their deployment at compile time. That is, Insense components may be designed much like the *Sink* and *TempSource* components, shown in Figures 5 and 8 respectively, such that they cater for both intra-node and inter-node communication during their deployment. Published channels may be used both locally and remotely. Consequently, the *TempBinder* component from Figure 9 could be deployed on the same node as the *Sink* component without requiring any alteration to the program code.

The three main benefits of such deployment agnosticism are as follows. First, the programmer can focus on higher level component logic and their interaction with other components rather than on component placement. Second, some facets of an application can initially be tested on a single node prior to being distributed over multiple nodes. Third, components may be compiled prior to their placement being decided. Insense thereby enables the composition of WSN applications from pre-defined and pre-compiled (as well as tailor-made) components. We envisage the future construction of software engineering tools for graphical composition of WSN applications by tailoring the instantiation, scheduling, placement, and wiring of pre-defined and pre-compiled components.

4. Implementation

In this section we focus on the inter-node channel implementation although parts of the implementation described in [1] and [17] are briefly presented for the reader's convenience.

The Insense compiler is written in Java and generates C source code that can be compiled and linked with the runtime system library and operating system. The code generation module is currently tailored towards the Contiki platform. The compiler generates *main.h* and *main.c* files containing the C code for global Insense definitions including procedures declared outwith components and globally defined types as well as the *main* sequence. The latter typically contains code to construct component instances and bind these together.

4.1 Components Structs and Channels

The compiler generates a source and header file for each component and *struct* declaration. When the programmer declares a component type the compiler generates a correpsonding C *struct* with members for component fields, channels, and variables and a pointer to a table of function pointers. This table contains a reference to the component's *behaviour* function and to functions that provide access to its component channels. The compiler also generates code for the component constructors and local procedures. Component constructors, procedures, and behaviour are implemented as Contiki processes so as to permit them to execute blocking calls resulting from sending and receiving data on channels. These processes each have an associated frame which is dynamically allocated on the heap when the process begins and which is used to store their state. The frame is destroyed when the process ends.

Insense *structs* are represented by C structs in the implementation with appropriate *constructor* and *copy* functions, the latter being used to support a pass-by-value scheme when sending *structs* over channels.

Every channel in the language is represented by a half-channel object in the implementation. Half-channel objects contain a number of fields as outlined in [17], including a list representing connections to other half-channels.

4.2 Memory Management

Our implementation makes use of dynamic memory allocation and a reference counting garbage collection scheme. Reference-counting garbage collection is suitable for Insense due to its simplicity and because Insense does not permit circular references. In our implementation a reference count and pointer to a destructor is stored along with every dynamically allocated object and called when the reference count reaches zero. The compiler generates a destructor for every dynamically constructed type in the language.

4.3 Inter-node Channels

The inter-node channel implementation comprises two main modules in the runtime library, neither of which are visible to an Insense programmer: 1) a radio module and 2) an inter-node channel handler (INCH) module. The INCH module provides support for all inter-node, inter-component operations. The radio module is designed to provide radio functionality to the INCH module and supports data transfer using single-hop, best-effort, unicast and best-effort, local area broadcast. The Radio module and the INCH module are described in more detail below.

4.3.1 Radio Module

The radio module is modelled as a composition of two Insense components, a *RadioIn* component and a *RadioOut* component as depicted in Figure 10.

The *RadioOut* component presents an interface containing a single incoming *send* channel and the *RadioIn* component presents an interface containing an outgoing *receive* channel. Both components use the *Unicast* and *Indentified Broadcast* layers in the Rime stack [18] as provided under Contiki. The *RadioIn* channel carries data of type *RadioPacket* which contains two fields: the destination address, and data encoded as an Insense *any* type.

The code necessary for the marshalling and un-marshalling of data is the responsibility of the Insense compiler which analyses the program and generates the appropriate functions. Pointers to these functions are stored in a map indexed by a type descriptor. When a datum is sent on an inter-node channel the appropriate serialisation function is called prior to transmission. When data is received from the radio the appropriate de-serialisation function is called after examining the type information in the payload.

Data is sent over the radio using the Rime unicast primitive if the *address* field contains a valid unicast destination address or it is *broadcast* if the field contains a zero-length array (our equivalent of a broadcast address). The *RadioIn* component receives notification of incoming radio messages via a call-back function that is registered with the Rime implementation. Once notified the component deserialises the message data and constructs a *RadioPacket* datum with the *address* field set to the source of the transmission. The packet data is then sent on the component's outgoing *receive* channel to the INCH.

4.3.2 Inter-node Channel Handler

The INCH component presents two sets of interfaces, one set to the radio module over which it abstracts, the other to provide inter-node channel services to the components running on a node. The first set contains incoming and outgoing channels on which data may be sent to and received from the radio as described in Section 4.3.1. The second is more complex and offers channels for the management and operation of:

1. Scheduling ticks;
2. Requests for connecting and disconnecting channels, and discovery operations, and;
3. Inter node, inter-component messages.

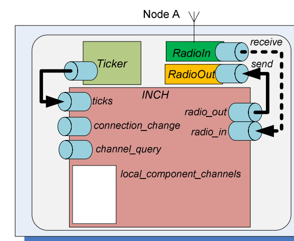

Figure 10. Radio and INCH Components

The *ticks* channel is used to schedule the dissemination of routing information to the network and is discussed in more detail in section 4.3.7. Requests for *connect*, *disconnect*, and channel discovery operations are made on the *connection_change* and *channel_query* channels shown in Figure 10. Finally, inter node, inter-component messages are sent and received on a set of channels called the *local_component_channels* set which contains channels that are connected to local components as the result of a *connect* operation. The *local_component_channels* set in Figure 10 is empty representing a scenario in which no *publish* and *connect* operations have taken place.

The INCH module is modelled as a single Insense component. The component performs a guarded, multi-channel select operation in its behaviour loop and then invokes the appropriate code block depending on the receiving channel. The channels from which INCH attempts to receive are: *connection_change*, *channel_query*, *radio_in*, *ticks*, and all incoming channels in the *local_component_channels* set.

4.3.3 Channel Exposure

As described in section 3.5.1 channels are exposed using the *publish* operation. To record such channels, each INCH maintains a *public channels table* containing local channel names, their direction, type and channel identifier which uniquely identifies local channels. This table is used whenever connection requests are made and when queries such as *getPublicChannels* or *findNodesPublishing* are performed. To illustrate its use, consider the scenario in Figure 11 in which a *Receiver* component has been declared and instantiated on node B as follows

```
r = new Receiver()
```

and its *input* channel has been published for inter-node use with the LCN "Recv" by executing

```
publish r.input as "Recv"
```

When the publish operation is executed, the INCH component examines its public channels table and throws an exception if the specified LCN is already in use. If the LCN is not in use it constructs a new channel and enters into its *local_component_channels* set after connecting it to the supplied component channel. The INCH then enters the new information into its *public channels table*.

4.3.4 Channel Binding

The INCH also maintains a *Bindings* table, which records inter-node, inter-channel connections. This table contains four fields: two node addresses and two local channel names. Entries in this table are created, whenever an inter-node connection is created and removed when a binding is dissolved.

Figure 11 depicts the scenario in which a component channel (of type out real) on node A is connected to a public incoming channel with the LCN "Recv" on node B (of type in real).

When the local INCH component receives the bind request as a result of the *connect* operation, it forwards that request via its radio to the remote INCH on node B with a message containing the four fields required to create an entry in node B's *Bindings* table.

Upon receipt of the request from the *radio_in* channel and, after finding the incoming channel with LCN "Recv" in its public channels table and creating a new binding in its *Bindings* table, the remote INCH sends an acknowledgment message to node A. Node A's radio component forwards the acknowledgement to node A's INCH component upon which it adds a new channel to its *local_component_channels* set and creates a new entry in its *Bindings* table.

Inter-node channel bindings are dissolved in a similar way by sending unbind requests instead of *bind* requests to the relevant INCH components.

Requests for connection change and the corresponding acknowledgment messages may be lost when transmitted by radio. The INCH propagates exceptions to the callers of *connect* and *disconnect* operations unless appropriate acknowledgements have been received. Since an application may choose to repeatedly execute *connect* and *disconnect* operations until no exceptions occur, the operations are idem-potent and the INCH sends an acknowledgement when it receives requests to *connect* an already connected channel or *disconnect* a channel that is not connected.

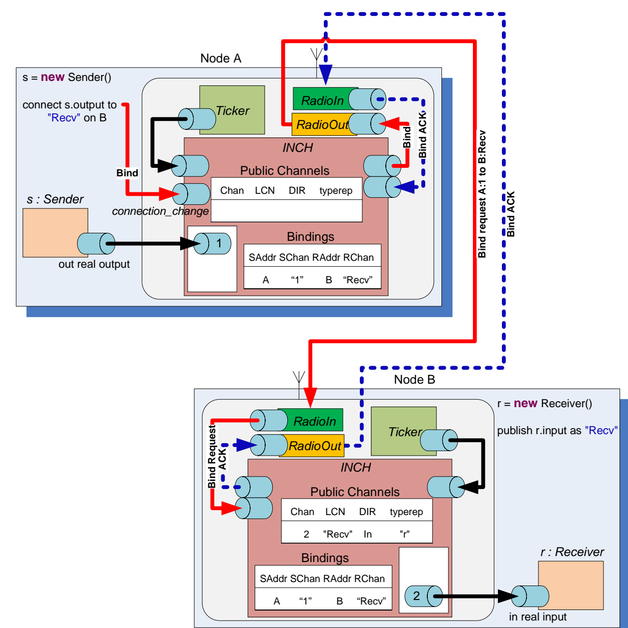

Figure 11. Channel Exposure and Binding

4.3.5 Sending Data

As described above, the INCH repeatedly performs a non deterministic select on all outgoing channels from its *local_component_channels* set each of which is connected to an internal component channel. When a datum is received from a component, the INCH iterates over the bindings for this channel in its *Bindings* table and attempts to send the datum to the appropriate destination node and channel. The INCH registers success once an acknowledgement is received from the corresponding INCH. Alternatively, the INCH registers failure by propagating an exception to the sending component if an acknowledgment is not received.

4.3.6 Remote Synchronisation

With intra-node, inter-component communication, the communicating parties handshake and exchange data contemporaneously. Such instantaneous synchronisation is not possible with inter-node and potentially multi-hop synchronisation. A mechanism is therefore required that permits a component to synchronise with a remote component via the INCH proxy components when communicating over inter-node channels. Consequently the implementation of outgoing channels associated with the INCH delay handshaking until either a timeout or acknowledgement have been received. If an acknowledgement is received the remote INCH has delivered the datum to the remote component and the handshake completes releasing the blocked sending process. Otherwise, if the timer expires prior to an acknowledgement being received, a *SendStatusUnknownException* event is propagated to the sender.

4.3.7 LSA-based Multi-hop Routing

The inter-node channel implementation incorporates multi-hop routing of data packets on minimum hop paths. Dynamic discovery of routing information is based on link-state advertisement (LSA) [19]. We chose an LSA-based approach over alternatives such as AODV [20] since communication links may not always be bi-directional.

In our implementation time-stamped link-state information is periodically disseminated to the network by the INCH components using Rime's Identified Broadcast primitive. The INCH components update their map when newer information is received from the network. Link-failure on an incoming link is recorded when link-state broadcasts are not received from the corresponding neighbour over a certain number of link-state cycles (in the current implementation 10).

The frequency with which link-state is disseminated is set once and remains fixed, at twice per minute, in our current implementation. However, a minor change would permit the dissemination frequency to be dynamically adapted in response to apparent stability or changes in network topology. For example, the frequency may be reduced to conserve energy when the network topology appears to be stable at the expense of increasing the network's response time to link failure. When new topology information is detected, the dissemination frequency may be increased to support fast dissemination of the information to the network.

Whenever an INCH component sends data to another node, it uses Dijkstra's algorithm [21] to search its link-state map to find the next hop in a minimum hop path to the destination and sends the data to the next hop. INCH components along the minimum hop path examine a packet's destination address and forward the packet to the next hop in the network until it arrives at its final destination. The packet payload is only de-serialised by the INCH on the destination node. That way, only the source and final destination nodes for any transmission require the marshalling code for the packet payload.

When a route to the destination node cannot be found according to a node's link-state map, this information is propagated to the sending process via a *NodesUnreachableException* value as described in section 3.5.4 above.

4.3.8 Discovery Mechanisms

The *getNeighbours* and *getNeighboursOf* operations return an array of addresses representing nodes for which the node in question either has an incoming or outgoing direct link according to the local INCH component's link-state map. The operation *getNumberHops* returns the length of the minimum hop path obtained through searching the link-state map for the given source and destination nodes.

The *getPublicChannels* operation returns the content of this node's Public Channels table stored in the local INCH. Executing the *getPublicChannelsOf* operation causes the INCH to send a suitable channel query to a remote INCH component and report the returned results or an empty array if no results were obtained.

Finally, executing the *findNodesPublishing* operation causes the INCH component to send a suitable query for the specified public channel to all reachable nodes and report the addresses of nodes which made suitable replies.

5. Application Simulation

In the following we illustrate the capability for Insense applications to autonomously configure and *self-heal* using a simulation.

5.1 Deployment

A sample deployment of the components shown in Figure 7 is simulated using Cooja (the **Co**ntiki **O**s **Ja**va simulator) [22]. That is, the application components are compiled down to executable MSP430 code and linked with a Contiki system image and the Insense runtime library for the TMote Sky platform. The executable system

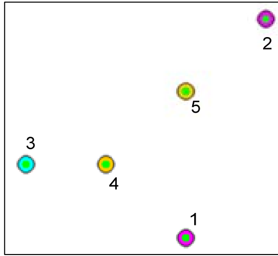

Figure 12. Node Deployment

images are then associated with simulated TMote Sky nodes running on the Cooja simulator. The sample application deployment is shown in Figure 12.

Instances of the *Sink* component (shown in Figure 5) are deployed on nodes 1 and 2. An instance of the *TempBinder* component (from Figure 9) along with its associated *TempSource* component (from Figure 8) is deployed on node 3. The spacing between nodes 1, 2, and 3 is such that intermediate nodes 4 and 5 are necessary to route both application data and data resulting from discovery and *connect* operations between the nodes. Nodes 4 and 5 do not contain any user-defined Insense components. Instead, they contain runtime components, including the Radio and INCH components discussed in sections 4.3.1 and 4.3.2 respectively, which are necessary to support the application's inter-node channel communication.

5.2 Application and System Settings

The *TempBinder* component is scheduled to conduct a search for sinks every two minutes (specified by the call to *periodicEnSchedule*) when the *TempSource* component is not connected. When connected, the *TempSource* component is set to take a temperature measurements every minute (as specified in the constructor in Figure 9) and *send* the measurements to the sink.

As described above, the INCH component is en-scheduled to disseminate link-state to the network twice a minute and the component's counter for detecting failed links is set to 10. An incoming link from node *N* will thus be deemed to have failed when no link-state advertisement is received from node *N* in 10 cycles, i.e. 5 minutes.

5.3 Self-configuration and Self-healing

In the simulation, the nodes establish a common view of the network topology after 5 minutes. When the *TempBinder* component next runs after the network becomes stable, it successfully discovers the "sink" channel exported by the *sink* on node 2. In this particular example the *TempBinder* does not discover the "sink" channel exported by the *sink* on node 1 (as it does not receive an acknowledgement). The *TempBinder* connects the *TempSource* component on node 3 to the *sink* instance on node 2, thereby completing the application configuration.

```
TIME:375774   ID:3   tb: bound
TIME:378578   ID:2   sink got reading 24.00
TIME:379280   ID:3   ts: send ok
<node 5 removed>
TIME:428757   ID:3   ts: send unknown
TIME:488342   ID:3   ts: send unknown
TIME:548736   ID:3   ts: send unknown
TIME:608663   ID:3   ts: send unknown
TIME:665047   ID:3   ts: send unreachable
TIME:665050   ID:3   tb: link broken
TIME:665052   ID:3   tb: try next sink
TIME:857628   ID:3   tb: bound
TIME:857692   ID:1   sink got reading 24.00
TIME:858196   ID:3   ts: send ok
```

Figure 13. Cooja Log File

Figure 13 is an extract of the log file produced by Cooja and shows the simulation time in milliseconds, the node number, and the output on the serial line (from the *PrintAny* statements) for all nodes in the network. After the *TempBinder* component instance *tb* establishes the inter-node channel connection, the *sink* instance on node 2 may be seen to receive a simulated temperature reading of 24.00 degrees Celsius. The synchronisation between the source and the sink may be observed in that the *TempSource* component instance *ts* reports success after the datum is received by the *sink* instance.

At time 379280, we have forcibly removed node 5 so that the *TempSource* on node 3 is no longer connected to the *sink* on node 2. The log in Figure 13 shows that no more readings are received by the *sink* on node 2 after the node is removed. Following the removal of node 5, the sending component on node 3 receives *SendStatusUnknownException* exceptions which causes the component to write "send unknown" messages to the log. After 5 minutes, the INCH throws the exception *NodesUnreachableException* to the *TempSource* instance *ts* at which point it sends a message on its error channel.

Finally, on receipt of the error notification, the *TempBinder* unblocks from the *receive oops* statement and, having reached the end of the for-loop, executes the next iteration of its behaviour loop. A connection to the *sink* on node 1 is re-established after the next discovery phase completes. Following reconnection, the log shows the *sink* on node 1 has successfully received a temperature reading.

5.4 Space Requirements

The space requirements of the application components used in the simulation are as follows. The code size for the *Sink* component is 800 bytes and each instance uses 64 bytes of RAM, the code size for the *TempBinder* is 1748 bytes and 104 bytes of RAM are required, and the code size for the *TempSource* component is 1150 bytes and 118 bytes of RAM are required. The sizes of the Insense Runtime library and the Contiki operating system code included by the linker for the above application are approximately 22kB each. Thus the total size for the application executing on nodes 1 and 2 is 45kB and the size of application executing on node 3 is 47kB. The code for

the INCH component, radio component, marshalling code, and half-channel implementation occupy ca. 8.5kB, 1.2kB, 3.5kB, and 2kB respectively (i.e. ca. 70% of the runtime).

It can be seen that the RAM usage of Insense programs are modest whilst the code footprint is a little large. The code generated by the Insense compiler allocates all the space required for Insense components to execute (including stack space). At first this appears to fit well with the *stackless* model used by Contiki proto-threads. However, Contiki processes are defined using macros and may not be dynamically allocated. Furthermore, if procedures or dynamic processes yield, some mechanism must be implemented to save their state. To implement Insense such mechanisms must be provided by a combination of the Insense runtime and the generated code. Combined, these contribute to the large footprint of compiled Insense components.

6. Conclusions and Further Work

The main contribution of this paper is to present a high-level component-based service-oriented model of distributed sensing in WSNs supporting inter-component communication over channels. In this model, the components of a sensing system may discover and access services by connecting to channels associated with other components in the network. The *self-healing* of applications is supported by an exception model that permits applications to detect channel communication anomalies and by language mechanisms that permit a distributed application to be dynamically re-wired.

We have demonstrated the efficacy of this model with a demonstrable implementation on the TMote Sky platform running Contiki. The Cooja simulation results demonstrate the ability of applications to autonomously configure and self-heal. The use of Contiki has permitted a running implementation of Insense. However, as described above it is not ideal. This is not to belittle the Contiki environment, which we have found to be extremely robust and well engineered – we are using it in a manner in which it was never intended.

To address the inadequacies in the runtime platform for Insense, colleagues at the University of Glasgow have initiated the development of a custom operating system for Insense in which component creation, and inter-component channel communication are factored out into the operating system [23]. We believe that this will demonstrate that Insense can be executed with a low code and data footprint given the right runtime environment.

7. Acknowledgements

Thanks to Ron Morrison and Joe Sventek who contributed to early discussions on the design of inter-node communication in Insense. This work was partially supported by the EPSRC grant "Design, Implementation and Adaptation of Sensor Networks through Multi-dimensional Co-design" (EP/C014782/1).

8. References

[1] A. Dearle, D. Balasubramaniam, J. Lewis, and R. Morrison, "A Component-Based Model and Language for Wireless Sensor Network Applications", in Proc. of 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2008) pp.1303-1308 IEEE Computer Society, 2008

[2] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors", in Proc. of The First IEEE Workshop on Embedded Networked Sensors, Tampa, Florida, 2004.

[3] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan, "Reliable and Efficient Programming Abstractions for Wireless Sensor Networks", in Proc. of PLDI'07, 2007.

[4] D. Gay, M. Welsh, P. Levis , E. Brewer, R. von Behren, D. Culler, "The nesC Language: A Holistic Approach to Networked Embedded Systems", in Proc. of PLDI'03, pp.1-11, 2003.

[5] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An operating system for sensor networks", in Ambient Intelligence, pp. 115-148, Springer Verlag 2004.

[6] I. Galpin, C. Brenninkmeijer, F. Jabeen, A. Fernandes, N. Paton, "An Architecture for Query Optimization in Sensor Networks", in Proc. of the 24th IEEE International Conference on Data Engineering (ICDE'08), pp. 1439-1441, 2008.

[7] P. Levis, D. Gay, and D. Culler, "Bridging the gap: Programming sensor networks with application specific virtual machines", Tech. rep., CSD-04-1343 UC Berkeley, Aug. 2004.

[8] P. Levis, and D. Culler, "Mate: A tiny virtual machine for sensor networks", in Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems (Oct. 2002).

[9] J. S. Miller, P. A. Dinda, and R. P. Dick, "Evaluating A BASIC Approach to Sensor Network Node Programming", in Proc. of SenSys'09, Berkeley, CA, USA, November 2009.

[10] A. Dunkels, "uBASIC: A really tiny BASIC interpreter", http://www.sics.se/~adam/ubasic/, 2007.

[11] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale", in Proc. of SenSys'04, Baltimore, Maryland, USA, November 2004.

[12] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, ”Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks”, in Proc. of SenSys'06, Boulder, Colorado, USA, November 2006.

[13] N. Brouwers, K. Langendoen, and P. Corke, “Darjeeling, A Feature-Rich VM for the Resource Poor”, in Proc. of SenSys’09, Berkeley, CA, USA, November 2009.

[14] K. Hong, J. Park, T. Kim, S. Kim, H. Kim, Y. Ko, J. Park, B. Burgstaller, and B. Scholz, “TinyVM, an Efficient Virtual Machine Infrastructure for Sensor Networks”, in Poster Session of SenSys’09, Berkeley, CA, USA, November 2009.

[15] R. Morrison, A. L. Brown, R. Carrick, R. C. H. Connor, A. Dearle, M. P. Atkinson, “Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment”, in Software Engineering Journal, pp. 199-204, December 1987.

[16] R. Milner, “Communicating and Mobile Systems: the Pi-Calculus”, Cambridge University Press; 1st edition, 1999.

[17] O. Sharma, J. Lewis, A. Miller, A. Dearle, D. Balasubramaniam, R. Morrison, and J. Sventek, “Towards verifying correctness of wireless sensor network applications using Insense and spin”, in Proc. of 16th International SPIN Workshop on Model Checking of Software (SPIN 2009), 2009.

[18] A. Dunkels, F. Osterlind, Z. He, “An Adaptive Communication Architecture for Wireless Sensor Networks”, in Proc. of SenSys’07, Sydney, Australia , November 2007.

[19] “Link-state shortest-path-first routing” in F. Halsall , “Computer Networking and the Internet”, pp. 344-352, Addison-Wesley, London, 2005.

[20] C. E. Perkins and E. M. Royer, “Ad-hoc On-Demand Distance Vector Routing”, in Proc. of the 2nd IEEE Workshop on Mobile Computing Systems and Applications, pp. 90-100, New Orleans, LA, February 1999.

[21] E. W. Dijkstra, "A note on two problems in connexion with graphs", in Numerische Mathematik 1, pp. 269-271, 1959.

[22] F. Österlind, “A Sensor Network Simulator for the Contiki OS”, Swedish Institute of Computer Science (SICS) Technical Report T2006:05, ISSN 1100-3154, February 2006.

[23] P. Harvey, “InceOS: The Insense Specific Operating System”, to be published as Computer Science Technical Report, University of Glasgow, 2010.