Developing Sensor Network Applications Using the TeenyLIME Middleware*

Paolo Costa[1], Luca Mottola[2], Amy L. Murphy[3], and Gian Pietro Picco[4]

[1]Vrije Universiteit, Amsterdam, The Netherlands, `costa@cs.vu.nl`
[2]Politecnico di Milano, Italy, `mottola@elet.polimi.it`
[3]ITC-IRST, Povo, Italy, & U. of Lugano, CH, `murphy@itc.it`
[4]University of Trento, Italy `picco@dit.unitn.it`

Abstract

One direction in the evolution of wireless sensor networks (WSNs) is in support of sense-and-react applications, where actuators are physically interspersed with the sensors that trigger them. This solution maximizes localized interactions, and therefore improves resource utilization and reduces latency w.r.t. solutions with a centralized sink. On the other hand, application development becomes more complex: the control logic must be embedded inside the network and requires appropriate coordination of the nodes' activities. Moreover, interactions tend to become stateful, and therefore reliable communication acquires an even greater importance.

This paper describes the design, implementation, and evaluation of TeenyLIME, a WSN middleware designed to address the aforementioned challenges. TeenyLIME provides application programmers with the high-level abstraction of a tuple space, shared among neighboring WSN nodes. We maintain that TeenyLIME's constructs well support sense-and-react as well as more traditional WSN applications and services, yielding a simpler, cleaner, and more reusable design. Our claims are supported by a quantitative comparison between applications developed with TeenyLIME and with existing WSN tools. Moreover, the TeenyLIME programming model is supported by an efficient middleware implementation, including a protocol that guarantees that remote tuple space operations are performed reliably. The evaluation through simulation presented in this paper shows that our protocol is able to achieve 100% reliability even in very challenging environments (e.g., up to 80% message loss), while keeping the overhead low and proportional to the error rate.

1 Introduction

Wireless sensor networks (WSNs) are a popular technology for monitoring and control applications, where they simplify deployment, maintenance, and ultimately reduce costs. Early WSN efforts were primarily concerned with *sensing* from the environment and reporting to a central data sink [1]. In contrast, an increasing number of applications (e.g., [2]) now include wireless nodes hosting actuators, and are therefore able to *react* to the external stimuli gathered by nearby sensors and affect the environment where they are deployed.

The *sense-and-react* pattern typical of sensor/actuator networks has a relevant impact on application development. Appropriate programming constructs are required to deal with the increased complexity of specifying how nodes, each with their own role, *coordinate* to accomplish a given global functionality. Reliability is fundamental, as interactions are typically stateful. The ability to locally react based on external stimuli is as important as—if not more important than—the ability to gather data provided by sense-only systems, whose interaction requirements are essentially subsumed by sense-and-react ones. These requirements are motivated in more detail in Section 2, where we describe a paradigmatic sense-and-react application, and discuss further the relationship with sense-only WSN systems.

This paper presents TeenyLIME, a middleware simplifying the development of WSN applications, and encompassing the peculiarities of sense-and-react scenarios. The foundation for TeenyLIME is the notion of *tuple space* [3], a repository of elementary sequences of typed fields, called *tuples*. This notion is revisited in an original way by

*This work is partially supported by the European Union under the IST-004536 RUNES project, and the Swiss National Science foundation under the NCCR-MICS grant number 5005-67322.

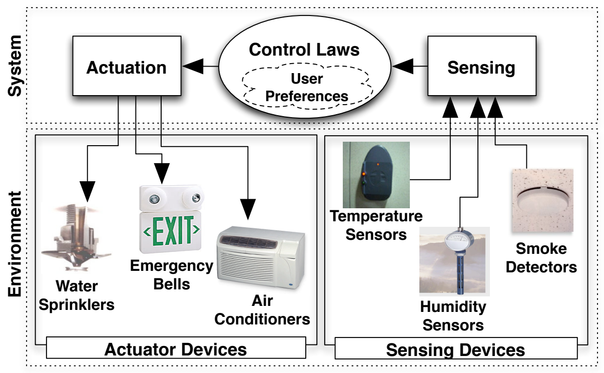


Figure 1: Schematic representation of a building monitoring and control application.

TeenyLIME by considering the WSN requirements of dynamicity, resource consumption, and reliability in the programming model, and by satisfying them concretely through an efficient middleware implementation. An overview of the TeenyLIME programming model and application programming interface (API) is provided in Section 3. Instead, Section 4 illustrates concretely the power of its WSN-specific abstractions by showing them in action in the design of the aforementioned sense-and-react application.

The development of WSN middleware must reconcile the need for expressive power and ease of programming with the reality of resource-scarce devices and unreliable (yet expensive) communication. Section 5 describes how these tradeoffs are harmonized in the TeenyLIME architecture, focusing particularly on the underlying distributed protocol ensuring that TeenyLIME's state-based interactions are performed reliably.

Model and implementation necessarily go hand-in-hand. Therefore, the evaluation of TeenyLIME in Section 6 focuses on both aspects. First, we assess quantitatively the effectiveness of the programming *model* for the sense-and-react application whose TeenyLIME design is sketched in Section 4, as well as for two alternative settings found in the literature. We derive code metrics for the TeenyLIME implementations as well as for their counterparts, implemented using plain nesC or also exploiting the higher-level support provided by Hood [4]. In both cases, results indicate that the expressive power of TeenyLIME enables cleaner, simpler, and more compact application code. Second, we evaluate quantitatively through simulation the effectiveness of the middleware *implementation*, showing that our reliable distributed protocol achieves 100% delivery even in presence of a very high error rate (80% packet loss), while still maintaining good performance.

Other research works have proposed node-level abstractions for programming WSNs: they are reviewed in Section 7. Plans for future work are reported in Section 8, along with brief concluding remarks.

A preliminary description of TeenyLIME appeared in a short paper [5]. Here, we expand it significantly by: *i)* showing practical uses of the programming model in the context of sense-and-react and other WSN applications; *ii)* introducing and evaluating a novel protocol for ensuring reliable tuple space operations; *iii)* providing an evaluation of both the model and implementation of TeenyLIME.

2 Reference Scenario and Motivation

Sense-and-react applications are being developed in many settings, from home automation [6] to road traffic control [7]. As a paradigmatic example, we consider *building monitoring and control*. Modern buildings typically focus on the following two main functionalities:

1. *heating, ventilation, and air conditioning* (HVAC [2]) systems provide fine-grained control of indoor air quality;

2. *emergency control* systems provide guidance and first response, e.g., in case of fire.

As with any other embedded control system, these applications feature four major components, illustrated in Figure 1. *User preferences* represent the high-level goal of the system, such as the desired temperature in the building, and the need to limit fire spreading. *Sensing devices* gather data from the environment and monitor the value of relevant

	Localized Computations	**Multiple Tasks**	**Reliable Interactions**	**Reactive Interactions**	**Proactive Interactions**
Applications					
Indoor Monitoring and Control [2,6]	Y	Y	Y	Y	Y
Outdoor Monitoring and Control [7]	Y	Y	Y	Y	Y
Outdoor Environmental Monitoring [1]	N	N	N	Y	Y/N
Object Tracking [11]	Y	Y	N	Y	N
Contour Finding [12]	Y	Y	Y/N	Y/N	N
Services					
Localization [13]	Y	N	N	Y	Y/N
Multi-hop Routing [10]	Y	N	Y/N	Y	Y/N
Query Processing [14]	N	N	Y/N	N	Y
Time Synchronization [15]	Y	N	N	Y	Y/N

Figure 2: Requirements for WSN applications and services.

variables. In our case, humidity and temperature sensors monitor air quality, while smoke and temperature detectors recognize the presence of a fire. *Actuator devices* perform actions affecting the environment under control. Remotely-controllable air conditioners adjust the air quality, while water sprinklers and emergency bells are used in case of fire. *Control laws* map the data *sensed* to the *actions* performed, to meet user preferences. For building monitoring and control, a (simplified) control loop may require activation of air conditioners when temperature deviates significantly from user preferences, and tune the action based on humidity in the same location. Similarly, when the temperature increases above a safety threshold, emergency bells are immediately activated, whereas water sprinklers are operated only if smoke detectors actually report the presence of fire. Oscillating behaviors must be avoided in all situations.

Application development in sense-and-react systems is complicated not only by the peculiarities of the devices involved, but also by the complexity of their interactions. A number of requirements arise, which can be grouped into high-level challenges germane to many applications in this field:

- **Localized computations** [8] must be privileged, to keep processing close to where sensing or actuation occurs. In a sense-and-react application it is indeed unreasonable to funnel all the sensed data to a single, powerful base-station, as this may negatively affect latency and reliability, without any sensible advantage [9].
- A single system might be called on to perform **multiple tasks** in parallel, similarly to our example where two separate control laws—one controlling the air conditioning, the other handling emergencies—coexist. Note how different tasks do not merely run on the same hardware, but also *share the data* generated by a subset of the sensing devices (e.g., temperature readings in our case).
- Sense-only WSN applications focusing on data gathering are inherently *state-less*, as their core task is that of communicating sensor readings to a given collection point. Conversely, sense-and-react applications often require *stateful* coordination, e.g., using current shared conditions (state) to act collaboratively. This poses more stringent requirements on the consistency of state, and consequently on the **reliability of interactions** modifying it. This issue, in combination with the use of WSNs for safety critical applications, motivates providing reliable operations on top of the inherently unreliable, wireless network—a challenge that is often considered independently from the programming model [10].
- **Reactive interactions**, where control law actions are automatically triggered based on the value of some physical phenomena, assume a prominent role. For instance, this is the case of a temperature reading deviating from the user preference, triggering an action in either of the two application tasks. Conversely, **proactive interactions**, common in many sense-only scenarios, are still needed to gather further information and better tune the actuation about to occur. For instance, the sprinklers in the building are activated only if and when smoke is detected.

Many of the above requirements are present also in more conventional WSN scenarios, some of which are analyzed and compared in Figure 2. Note that these include both full-fledged, end-user *applications*, as well as lower-level common *services*. Space constraints prevent us from providing a detailed analysis here. However, for instance, we can note that multiple tasks must be carried out in parallel also in object tracking [11], usually implemented as the composition of a localization technique, a tracking mechanism, and a routing protocol used to report information to a pre-determined sink. Each component executes independently, but also shares data with the others. Furthermore, localization services need to perform localized interactions, as most of the approaches in the field base the position estimation on those reported by nearby hosts. Similarly, as a final example note how time synchronization algorithms require reactive interactions. Indeed, the clock skew w.r.t. an anchor point is usually monitored continuously, and the synchronization computation performed when it grows above a given threshold.

3 Tuple spaces for Wireless Sensor Networks

In the 1980's, Linda [3] emerged as a minimal, content-based shared memory model for parallel process interaction. Three basic operations allow processes to create (**out**), read (**rd**), and remove (**in**) data elements from a shared data structure. Linda data takes the form of tuples, sequences of typed fields such as ⟨"foo", 29⟩. The **out** operation stores these data elements in a *tuple space*, a multiset of tuples. **rd** and **in** query for data by specifying patterns such as ⟨"foo", ?integer⟩ that use either *actual* or *formal* values, where formals are a kind of "wild card" that match any data of a particular type.

The tuple space can be exploited straightforwardly for data sharing with processes reading or removing the data left behind by others, or to manage coordination activities. Consider a simple mutual exclusion problem where only one process at a time can use some resource. This can be handled in Linda by outputting (**out**) a token tuple that must be removed (**in**) and held by a process before it is allowed to access the critical section. When exclusive access is no longer required, the process must output the token tuple so that another process can use it to access the resource. Allowing multiple, simultaneous accesses to a resource is accomplished simply outputting multiple token tuples. In general, the Linda model supports a wide range of applications requiring interaction among loosely coupled processes.

The needs of the decentralized, sense-and-react WSN applications we target resonate with the process coordination supported by Linda. The shared tuple space can be exploited to easily make the sensed data accessible to multiple nodes, or to enable sophisticated coordination behaviors. However, the Linda model (let apart its implementations) cannot be applied directly in WSNs due to their constraints in terms of resource consumption and dynamicity.

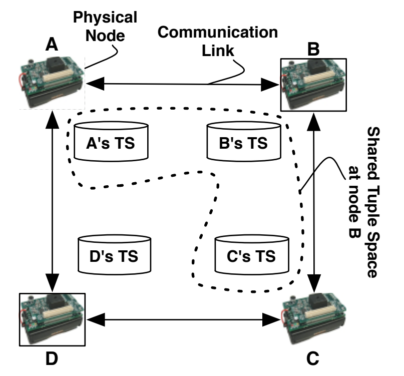
3.1 The TeenyLIME Model

The first concern when providing Linda-like coordination in WSNs is *where* to place the tuple space. Maintaining a single tuple space accessible in a client-server style is not practical as the communication overhead would outweigh the benefits of the coordination abstraction. The challenge is similar to the one posed by mobile ad hoc networks (MANETs), where it has been successfully tackled by the LIME [16, 17] model and middleware through the notion of a *transiently shared tuple space*.

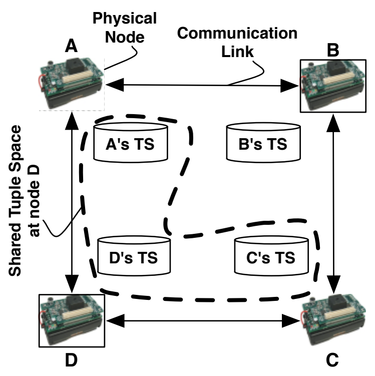
In LIME, the Linda tuple space is partitioned across hosts, but tuples are shared among connected hosts when wireless communication allows. Similarly, in TeenyLIME each node hosts a tuple space, storing tuples useful for coordination such as sensed data, pending tasks and requests, descriptions of services available to other nodes, and even system information such as location or battery level. However, tuple space sharing is enabled only among nodes that are within direct (one-hop) communication range. *Sharing* essentially means that a node views its local tuple space as containing its own tuples, plus any of the tuples in the tuple spaces hosted by its neighbors, as shown in Figure 3. As in LIME, the extent of sharing is determined dynamically by connectivity.

Restricting sharing to one-hop neighbors is beneficial for many WSN applications as it naturally provides access to *nearby* information, and is expressive enough to capture several common requirements, as we show in Section 4. Further, limiting communication to one hop allows for an efficient, broadcast-based implementation of the model, as discussed in Section 5.

The coordination operations in TeenyLIME are essentially those of Linda, namely inserting, reading, and removing tuples. The scope of an operation, however, can span the whole transiently shared tuple space. For instance, a query issued by a node may return a matching tuple found in any of tuple spaces in the one-hop neighborhood—including the local one. Therefore, TeenyLIME enables more abstract interactions among nodes, by focusing on the *data* of concern (e.g., reading a temperature tuple from the neighborhood) without the need to explicitly deal with system configuration issues (e.g., tracking the identity of nodes and their presence). Moreover, TeenyLIME also borrows from LIME the notion of *reaction*: a fragment of code whose execution is automatically triggered upon appearance of a given tuple anywhere in the shared tuple space. Reactions are also registered by specifying a pattern, that describes the conditions to trigger the execution of the reaction code. This provides an easy way for developers to monitor changes in the content of the shared tuple space.



(a) Shared tuple space at node B.



(b) Shared tuple space at node D.

Figure 3: Transiently shared tuple spaces in TeenyLIME.

```
interface TupleSpace {
  // Standard operations
  command TLOpId_t out(bool reliable, TLTarget_t target, tuple *tuple);
  command TLOpId_t rd(bool reliable, TLTarget_t target, tuple *pattern);
  command TLOpId_t in(bool reliable, TLTarget_t target, tuple *pattern);
  // Group operations
  command TLOpId_t rdg(bool reliable, TLTarget_t target, tuple *pattern);
  command TLOpId_t ing(bool reliable, TLTarget_t target, tuple *pattern);
  // Managing reactions
  command TLOpId_t addReaction(bool reliable, TLTarget_t target, tuple *pattern);
  command TLOpId_t removeReaction(TLOpId_t operationID);
  // Returning tuples
  event result_t tupleReady(TLOpId_t operationId, tuple *tuples, uint8_t number);
  // Request to reify a capability tuple
  event result_t reifyCapabilityTuple(tuple *capTuple, tuple* pattern);
}
interface NodeTuple {
  // Request to provide a tuple containing node-level system information
  event tuple* reifyNodeTuple();
}
```

Figure 4: TeenyLIME API.

Next, we describe the TeenyLIME API. While in principle the programming model is independent from the node platform, we present here the API using nesC as the host language, as our middleware is currently built on top of TinyOS.

3.2 The TeenyLIME API

The TeenyLIME API provides a `TupleSpace` interface whose operations manipulate the transiently shared tuple space, as shown in the nesC API of Figure 4. The first three operations correspond directly to the Linda operations discussed earlier, while **rdg** and **ing**, represent variants (as in [18]) that return all matching tuples, instead of a single match. Two operations are also provided to install and remove reactions.

In TeenyLIME, all operations are asynchronous, allowing the application to continue while the middleware completes a tuple space operation[1]. This approach blends well with the nesC event-driven concurrency model. Therefore, all query operations are *split-phase* [19]: first the operation is issued, then the `tupleReady` event is signaled when the operation completes. The return parameter for each operation is an identifier, or a special constant (`TL_OP_FAIL`) in case of error. The identifier and the data tuple(s) form the contents of the `tupleReady` event, allowing the application to associate the data with its earlier request. If multiple tuples are returned, the `number` parameter indicates how

[1]In most Linda implementations, the **rd** and **in** operations are blocking, meaning they do not return until a matching tuple is available.

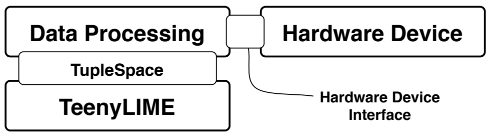


Figure 5: Component configuration of a generic node used in the building monitoring and control application.

many. Similarly, the execution of reactions is intrinsically asynchronous, with their trigger generating a `tupleReady` event.

Several other features of the API require further description. However, rather than describe them individually in isolation, in this paper we choose to discuss them "in action", i.e., hand-in-hand with the description of the Teeny-LIME-based design of the applications we outlined in Section 2. The reader can find an alternative, albeit preliminary, description of these features in [5].

4 Application Development with TeenyLIME

As discussed in Section 2, our reference example contains two sub-tasks, one that manages the air conditioning system (HVAC) and a second for emergency situations such as fire. Each of these sub-tasks involves different types of nodes, e.g., humidity sensors in the HVAC sub-task, and smoke detectors to face fire emergencies. Temperature sensors are instead used in both sub-tasks. A template for the nesC component configuration on a single node is shown in Figure 5. This is customized according to the particular node characteristics and required processing. Specifically, for each type of node, we implemented the required data processing entirely on top of the `TupleSpace` interface, masking completely the TinyOS generic communication layer. This component communicates with a dedicated one interfacing with the hardware device that performs the actual sensing or actuation. For instance, in the case of a node controlling a water sprinkler, the data processing component implements the control law run in case of emergencies, and the hardware interface is connected to the actual sprinkler device.

In the following, we explain the remainder of the building monitoring and control application design and implementation. We begin by illustrating how we leverage off sharing and describing how interactions among nodes rely on tuple space operations and reactions. We then provide additional details about how these interactions are carried out with API features dedicated to WSNs, using the application as a motivation and source of examples for the discussion.

Sharing Application Data: `TupleSpace`. In our design, all application data is represented as tuples. Specifically, both *sensed data* as well as the *commands* to operate the actuators take the form of tuples. Interestingly, data sharing occurs among the aforementioned two sub-tasks. Temperature sensors periodically output to their local tuple space a tuple containing the most recent temperature reading. Independently, the interested neighboring nodes register temperature reactions. The presence of a new temperature value triggers all matching reactions, thus delivering the data to as many applications as needed without the temperature node requiring any special code or behavior specific to the data consumers.

Sharing System Data. The coordination of activities across heterogeneous nodes sometimes relies on system information, such as the node location or capabilities. In TeenyLIME, this information is made available in the same way as application data, i.e., as tuples shared among neighboring nodes. These tuples contain a field describing the (logical) location (e.g., a room) where a node is deployed, and the sensing/actuator devices it is equipped with. Exactly which data to provide is defined by the application programmer, by specifying the body of the handler for the `reifyNodeTuple` event, shown in Figure 4. This event is signaled periodically by the TeenyLIME run-time, and the execution of the corresponding handler regenerates the tuple with new application-defined values. In our implementation, the local tuple space of each node contains tuples describing each of its neighbors. This is accomplished by appending the neighbor tuple to all outgoing messages; therefore, when the message is overheard by all neighbors, they extract the neighbor tuple and insert it locally. This way, it is easy to query the tuple space to obtain information on nodes with specific abilities.

(Reliable) Proactive and Reactive Interactions. As we mentioned, interaction among nodes occurs through standard tuple space operations, complemented by reactions. Figure 6 uses the fire control sub-task to illustrate how the two are used together to trigger notifications, perform distributed operations gathering data from neighboring nodes, and

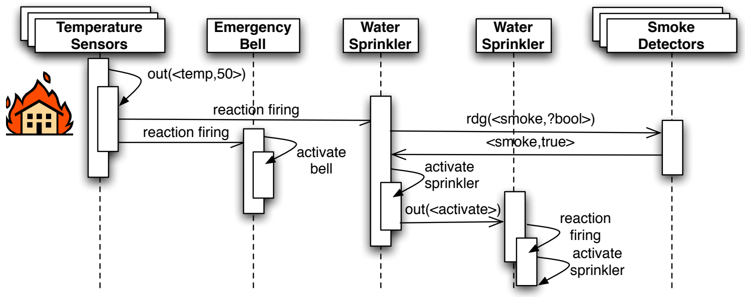


Figure 6: Sequence of operations to handle a fire. Notified about increased temperature, a node controlling water sprinklers queries the smoke detectors to verify the presence of fire. If necessary, it sends a command activating nearby sprinklers.

requesting actuation commands, in a pattern recurring in both sub-tasks of our application.

Both emergency bells and water sprinklers have a reaction registered on their neighbors, looking for temperature tuples over a given threshold. Thanks to transient sharing, the tuple space content is shared among all neighboring nodes. When a neighboring temperature sensor performs an **out** with such a tuple, the reaction fires on the two actuator nodes. As we discussed in Section 2, however, the behavior is different. The node hosting the emergency bell immediately activates its device. Instead, the water sprinkler node proceeds to verify the presence of a fire. The latter behavior, specified as part of the reaction code, consists of proactively gathering the readings from nearby smoke detectors. In case these report fire, the water sprinkler node requests activation of nearby sprinklers. Actuation is accomplished in a two-step process, and relies on reactions as well. The node requesting actuation inserts a tuple representing the command on the nodes where the activation is to occur. The presence of this tuple triggers a locally installed reaction that delivers the activation tuple to the application, which reads the various tuple fields and operates the actuator device accordingly.

Because fire detection requires the maximum degree of reliability, its implementation takes advantage of *reliable operations* for guaranteeing correct communication of reaction and query results, of the **rdg** operation on smoke detectors, and of the **out** operations towards actuators. The HVAC sub-task, instead, uses reliable operations only for actuation, but gathers data using non-reliable **rdg** operations. As shown in Figure 4, the selection between unreliable and reliable is done using a flag, available in most operations. The former offers a lightweight form of best-effort communication suitable for state-less, data collection applications, while the latter offer stronger guarantees at the price of higher resource consumption, as analyzed in Section 6.2.

Restricting the Scope of Operations. Since the tuple space is distributed across nodes, TeenyLIME allows one to explicitly define the evaluation scope for each operation invocation. These scope restrictions are specified using the `target` parameter present in most of the operations shown in Figure 4. In the processing we just described, the water sprinkler performs a **rdg** restricted (with the `TL_NEIGHBORHOOD` constant) to the union of all tuple spaces hosted by one-hop neighbor smoke detectors, and then its actuation process performs an **out** of the actuation command to a specified sprinkler, denoted by setting `target` to the node's address. Other allowed values for `target` include the local tuple space (`TL_LOCAL`), and the union of the neighbors' tuple spaces with the local one (`TL_ANY`).

Filtering the Data Being Communicated. In many WSN applications, including ours, action must be taken only when a sensed value crosses a given threshold. For example, nodes controlling water sprinklers and emergency bells use reactions to receive notifications when the temperature rises above a safety threshold. Nodes controlling air conditioners behave similarly to receive notifications when the temperature falls outside a given threshold defined by the user. These conditions require a predicate over tuple field values—something that cannot be achieved with the standard type- or value-based Linda matching semantics. In TeenyLIME, patterns are extended to support custom matching semantics on a per-field basis. For instance, the requirement concerning the safety threshold can be concisely expressed by using *range matching*, requiring the temperature field to be greater than a given parameter, as in:

```
tuple temperatureTempl = newTuple(2, actualField_uint16(TEMPERATURE_READING), greaterField(TEMPERATURE_SAFETY));
```

using one of the default range matching. The programmer is free to define her own.

Note how the issue is *not* simply one of expressive power, but it has significant repercussions on communication efficiency. Indeed, without this feature the programmer can specify, in queries and reactions, only a *generic* template for *any* temperature reading. Tuples matching these generic temperature templates would then be transmitted when requested (in our case, every time a new temperature sample is available) most often only to be discarded locally as below threshold, thus wasting significant communication resources.

Dealing with Short-Lived Data. Sensor data often remain useful only for a limited time after collection, depending on the task it is being used for. For instance, the emergency bell has no interest in reacting to a temperature value that was sensed an hour before. Instead, this very same data may be of interest for a component run periodically in the HVAC application, to build a day-long analysis of temperature trends.

In TeenyLIME, time is divided into *epochs* of constant length, and every data tuple is stamped with an application-accessible field containing the current epoch value. Three helper functions allow the application developers to deal with time:

```
setFreshness(pattern,freshness)
getFreshness(tuple)
setExpireIn(tuple,expiration)
```

The first customizes a pattern, similarly to the aforementioned range matching, to impose the additional constraint to match tuples no more than `freshness` epochs old. If a pattern does not specify freshness, it matches any tuple regardless of its age. The second function returns the number of epochs elapsed since the `tuple` was created. Finally, the third specifies how many epochs the `tuple` is allowed to stay in the tuple space. When the timeout associated to the tuple expires, the tuple is automatically removed. Sample code to exploit freshness is shown in conjunction with the next feature.

Generating Data Efficiently: Capability Tuples. According to our application semantics, humidity sensors and smoke detectors need not be monitored continuously. Rather, their data is accessed only when an actuation is about to occur. In TeenyLIME, reading the sensed value is accomplished by issuing a **rd**, however this requires that fresh-enough data be present in the tuple space when the operation is issued. If data is only seldom utilized, the energy required to keep tuples fresh is mostly wasted. An alternative is to require that the programmer encodes requests to perform sensing in a way similar to actuation commands, enabling the receiving node to perform sensing on-demand and return the result. However, this solution requires extra programming effort, is error-prone, adds processing overhead, and is therefore equally undesirable.

Instead, to deal with these (frequently-occurring) situations, TeenyLIME developers are given the ability to output *capability tuples* indicating that a device has the capability to produce data of a given pattern. When a query is remotely issued with a pattern matching a capability tuple, the `reifyCapabilityTuple` event is signaled. This reports the pattern included in the query and the matching capability tuple. The application handles this event by taking a fresh reading, and outputting the actual data to the tuple space. From the perspective of the data consumer, nothing changes. Instead, on the data producer, capability tuples enable energy savings as data readings can be taken only on-demand, eliminating the requirement to maintain constantly fresh data in the tuple space and with no additional programming effort on the client side.

For this reason, in our application humidity values and smoke measurement are represented using capability tuples. The code implementing the processing is shown in Figure 7 for a smoke detector. Note this is the only processing smoke detectors and humidity sensors are required to implement. In this sense, the figure illustrates the core of the application in these cases, only initialization and error handling routines are not shown. Figure 7 also shows that when the actual value is output, the tuple is allowed to persist in the tuple space based on an expiration time associated to the tuple, as described above. Therefore, the capability tuple is eventually matched again by future queries, that trigger the sampling of a fresh reading.

Interestingly, the concept of capability tuple can be generalized to allow *any action* to be taken at the data producer side. Matching a query pattern to a capability tuple does not necessarily trigger a sensor reading, but may invoke some other application function (e.g., computing the average of all temperature tuples), whose result is still inserted in the tuple space and returned to the requester.

Avoiding Oscillating Behaviors: a Case for Reliable in. In the HVAC sub-task, the system runs the risk of oscillating behavior if multiple nodes controlling air conditioners in the same location (e.g., same floor) independently run the control algorithm. To prevent this, we designed a mechanism to assign a master role to only one of the co-located controller nodes. This is realized in TeenyLIME as part of the data processing component of the air conditioner

```
command result_t StdControl.start(){
  tuple capTSmoke = newCapabilityTuple(2,
                    actualField_uint16(SMOKE_READING), formalField(TYPE_UINT16_T));
  call TupleSpace.out(FALSE,TL_LOCAL,&capTSmoke);
  return SUCCESS;
}
event result_t  TupleSpace.reifyCapabilityTuple(tuple *ct, tuple *p){
  call SmokeDetector.getData();                      // Request a reading from the smoke detector device
  return SUCCESS;
}
event result_t SmokeDetector.dataReady(uint16_t reading){ // Reading from the smoke detector device is ready
  tuple smokeValue = newTuple(2,
                    actualField_uint16(SMOKE_READING), actualField_uint16(reading));
  setExpireIn(&smokeValue,EXPIRATION_EPOCHS);
  call TS.out(FALSE,TL_LOCAL,&smokeValue);
  return SUCCESS;
}
```

Figure 7: Application code for a smoke detector node. Initialization routines and error handling are not shown, capitalized keywords represent constant values.

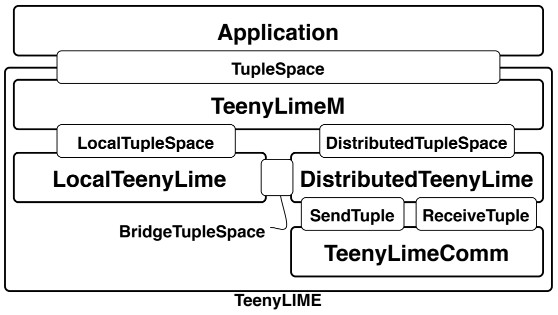


Figure 8: TeenyLIME component configuration.

controller by creating a single token tuple per location, and requiring a process to hold this tuple in order to act as the master. When the current master wishes to yield the role (e.g., to preserve its energy) it locally outputs the token tuple. This triggers a reaction previously installed on all controllers, informing them that the token, and the associated master role, is available. The reaction code at these nodes immediately issues a reliable **in** operation to remove the newly available token. Only one controller will successfully remove the token, becoming the master. Because the loss of the token implies no controller is acting as the master, reliable token transfer is imperative.

Note how this mechanism is independent from the air conditioner component. Indeed, since the TeenyLIME API exploits nesC parametric interfaces [19], the master/slave processing can be realized as a separate, encapsulated component. This turned out to be useful in other scenarios as well, as mentioned later.

5 The TeenyLIME Middleware

This section illustrates the design and implementation underlying TeenyLIME, focusing particularly on the distributed protocol enabling *reliable* operations.

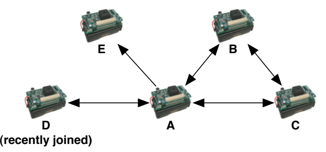
5.1 Design and Implementation Highlights

The overall design of TeenyLIME aims at enabling easy customization and extension of the different aspects involved. The **component configuration** within the middleware is illustrated in Figure 8. The `TupleSpace` interface is provided[2] by a `TeenyLimeM` component delegating the operations either to the `LocalTeenyLime` or `DistributedTeenyLime` component, depending on the operation target. The former essentially provides storage space for the local tuple space, and performs the tuple matching. The latter is responsible for distributed operations

[2]In this section, the meanings of "providing" and "using" are the ones defined the TinyOS programming model [19].

	Last Sent	*Last Received*
B	29	8
C	19	78

(a) *MsgTable* for node A.



(b) Sample network topology.

Source	Target	Neighbors Last Received	Payload
A	$\langle B, 30\rangle$, $\langle C, 20\rangle$	$\langle B, 8\rangle$, $\langle C, 78\rangle$, $\langle D, *\rangle$	

(c) Format of the next message to be broadcast by A.

Figure 9: An example illustrating the TeenyLIME protocol for reliable operations. In Figure 9(b), arrows indicate the possibility to communicate towards a node.

(e.g., maintaining the set of remotely installed reactions) and exchanges data with `LocalTeenyLime` through the `BridgeTupleSpace` interface. The actual communication is implemented in the `TeenyLimeComm` component, including our distributed protocol ensuring reliable operations, described next. The clear separation of local processing, distributed processing, and communication concerns enables customization of one aspect without impact on the rest of the system. For instance, our mechanism for reliable communication can be easily replaced with a platform- or scenario-specific solution.

The **matching semantics** is completely decoupled from the rest of the implementation. Our implementation includes the standard value- and type-based matching as well as some TeenyLIME-specific semantics, such as range matching described in Section 4. However, developers are free to modify or add new field matching mechanisms to better meet their requirements. To do so, they need only define an additional constant to distinguish the new matching criteria, define the format of a customized tuple field if needed, and implement a boolean function that takes two fields as parameters and returns whether the former matches the latter according to the required semantics.

To implement **distributed reactions** that account for nodes joining or failing, the `DistributedTeenyLime` component takes a *soft-state* approach. Specifically, a host periodically sends a message containing the definitions of all reactions that should be installed on its neighbors. The arrival of this message effectively refreshes a timer associated with the installed reactions. If a timer expires, the reaction is removed because either it has been actively removed by the application or the issuing node has failed. This approach to periodically refresh remote state is widely used in WSN, (e.g., in [20]) as it is sufficiently lightweight to match the characteristics of WSN nodes.

Expressing the internal processing to implement *capability tuples* requires keeping track of every remote node whose query matched a local capability tuple, then returning to that node the actual tuple once (locally) output by the application. Due to the TinyOS programming model, implementing this functionality directly would require a significant amount of bookkeeping code spanning two components: `LocalTeenyLime`, where the actual tuple will be stored, and `DistributedTeenyLime`, where the query has been received. However, during implementation we observed that the processing required to manage capability tuples is exactly the same *as if* a remote reaction were installed with the same template as the query *before* the application output the actual tuple. Our implementation exploits this by having the `DistributedTeenyLime` component install an internal reaction on the local tuple space using the template contained in the query before signaling the `reifyCapabilityTuple` event. When the application actually outputs the tuple, this matches the aforementioned reaction, and is subsequently, automatically delivered to the intended recipient. The only additional processing required is to remove the internal reaction once it has fired. This way, only 24 lines of nesC code are needed to support capability tuples, and all the related processing is completely encapsulated in `DistributedTeenyLime`.

5.2 Enabling Reliable Operations

As previously observed, sense-and-react applications are often state-based, and therefore the state must be kept consistent. Further, when used in critical environments, information must be transmitted reliably, for example to activate an alarm. In these same scenarios, message order is equally important. For instance, in the fire control scenario, a sprinkler may receive an actuation command followed by a false alarm. Clearly, receiving them in the opposite order does not have the same effect. Therefore, it is important to incorporate message reliability into TeenyLIME. While reliability in WSNs has been previously studied, solutions are usually targeted to multi-hop scenarios, e.g., [21], or require knowledge on the expected message rate, as in [22]. Therefore, they are not applicable in our particular context.

Motivated by these considerations, we devised a novel protocol to guarantee reliable, in-order message delivery for both unicast and broadcast messages among neighboring nodes in a WSN. While our solution is motivated by TeenyLIME, it can also be seen as an independent component, applicable in other systems with the same reliability requirements. In fact, our implementation encapsulates the protocol into a TinyOS component that is easily reusable.

Our protocol achieves reliability by requiring the message sender to repeat transmission until all intended recipients have received the message. The main challenges are to decide *which* messages to resend and *when*. Clearly, one approach is to require per-message acknowledgments, however this introduces unacceptable communication overhead in the WSN scenario. Instead, we opt for a solution where acknowledgments are piggybacked on all outgoing messages. Because we assume a broadcast communication model in which all messages can be received by all neighbors, any message can carry information for all neighbors in communication range, thus further reducing the overhead w.r.t. an explicit acknowledgment approach.

The key to our reliability protocol is to maintain a *MsgTable* at every node to track the sequence number of the latest messages that have been sent to and received from all neighbors. Figure 9(a) shows an example for node A, assuming it has only two neighbors, B and C, as in Figure 9(b). We return later to nodes D and E. The second part of this information, namely about the latest messages received, is piggybacked on every outgoing message. On receipt, a node can determine if it has sent messages that have not been received, and retransmit if necessary. For example, if node C's MsgTable indicates that it has sent message 79 to A, the receipt of the message from A in Figure 9(c) triggers C to resend message 79.

In the remainder of this section, we explain the details of how information about messages is handled for reliability, as well as how a node knows which neighbors *should* receive a message in the presence of asymmetric and new links.

Keeping Track of Messages. As shown in Figure 9(a), a node must track messages sent to its neighbors. A complication arises because both unicast and broadcast message must be reliable, and further, the set of neighbors to receive the broadcast messages can change over time, e.g., when a node is added. Therefore, we adopt a solution that independently tracks the total number of messages sent to *each* node, and creates a unique message identifier for each message using these values. For example, when a broadcast message is sent, as in the example of Figure 9(c), the sequence number for all nodes in the MsgTable are incremented, and the message identifier contains all these new values. Instead, if the message is unicast to B, only the value for B is incremented, and only this value is used as the message identifier. In both the broadcast and unicast cases, the resulting message identifier duals as the recipient list for the message, as it contains the identities of all nodes that must acknowledge the message. This simple mechanism allows us to track only the number of messages sent to each node, ignoring if they are unicast or broadcast, and trivially allows the set of broadcast recipients to grow and shrink.

To be able to recover from a message loss, it is important for each node to buffer all outgoing messages until they have been acknowledged. When a node receives a message that contains its identifier in the *Neighbors Last Received* field, it knows that all messages in its buffer with sequence numbers up to and including the one in the message have been received by that neighbor. Therefore, that neighbor is effectively removed from the recipient list of the buffered outgoing messages. When a buffered message has no more recipients, it has been fully delivered and is deleted.

Information about sequence numbers in the MsgTable is also useful for identifying when a message arrives out of order. For example, if A receives message 10 from B, it knows message 9 was missed. Therefore, it does not update its MsgTable, but instead buffers message 10 until B retransmits 9. After 9 is received, 10 is removed from the buffer and processed as normal, incrementing the *Last Received* entry for B to 10.

Finally, we observe that the recovery process requires transmission of messages carrying the MsgTable from all recipients. If a receiver is not periodically transmitting data, this could delay feedback to the sender of the reliable message. Therefore, when a node receives a reliable message, it starts a timer. If it has not sent any messages before the timer expires, it generates a *dummy* message that simply acts as an advertisement of the messages it has received. However, it may also happen that the recipient lost the message and then no timer is started. To deal with these cases, the sender periodically checks its buffer and retransmits those messages which have not been acknowledged recently.

Keeping Track of Neighbors. To verify that all neighbors have received a message, we are required to track the identities of the neighbors of every node as this set changes due to failure and node additions. One possible solution is to maintain a potential neighbor list containing the identity of each neighbor from whom a message is overheard, associated to a specific timer to account for nodes failing. When the timer expires, the corresponding entry is removed. Unfortunately, this list would include nodes on uni-directional links, which clearly cannot be considered in the reliability protocol because even if messages can be received, acknowledgments cannot be sent in the reverse direction.

To handle this situation, when a node hears from another (e.g., A in Figure 9(b) overhears a message from D) it puts information about this node into the message (i.e., $\langle D, *\rangle$) indicating it is a potential, but unconfirmed neighbor. The

Component	Explicit states		Lines of code		% of application data in TeenyLIME
	TeenyLIME	Plain TinyOS	TeenyLIME	Plain TinyOS	
AirConditioner	3	8	93	282	72%
MasterSlave	(MR · 2)	(MR · 3) + 1	153	205	48%
TemperatureSensor	0	(NC · 2) + 1	44	107	100%

Figure 10: Comparing the TeenyLIME-based implementation against plain TinyOS. MR represents the maximum number of different regions the component implementing the master/slave mechanism is supposed to handle, NC represents the maximum number of air conditioners interested in the readings of a specific temperature sensor.

receipt of this information at D indicates that the link is symmetric, and A should become a neighbor of D. Similarly, A will eventually see its own identifier in a message from D, and will add it as a neighbor. Note that in the case of uni-directional links, potential neighbors will not be confirmed. For example, in Figure 9(b), E's outgoing messages will contain $\langle A, * \rangle$, but because A cannot receive these messages and therefore will never put any information regarding E in its outgoing messages, E will never add A to its neighbor list.

6 Evaluation

To evaluate the effectiveness of our approach, in this section we compare quantitatively the TeenyLIME design we described in Section 4 for our sense-and-react application against existing approaches, and also study the applicability of TeenyLIME beyond our target scenario. Further, we report quantitative results illustrating the performance of the distributed protocol we use to ensure reliable operations.

6.1 Evaluating the Programming Model

The objective of this section is to assess the effectiveness of TeenyLIME in enabling more flexible application design and simpler implementations. To the best of our knowledge, there are no programming abstractions expressly designed for the sense-and-react scenarios we target. For this reason, we compare the TeenyLIME-based implementation of our reference application against the implementation of the same application on top of TinyOS.

Additionally, to showcase the flexibility of TeenyLIME's programming model, we briefly report on the design and implementation of two applications belonging to some of the mainstream WSN settings outlined in Section 2. Specifically, we consider an application for tracking moving objects [11] and a multi-hop communication protocol called Mutation Routing [23]. The former is widely recognized as a good benchmark to test the flexibility of WSN programming abstractions [4,12,24]. The latter illustrates that the TeenyLIME one-hop operations are powerful enough to enable the implementation of multi-hop routing mechanisms. In both cases, we compare our TeenyLIME-based design against Hood [4], a programming abstraction specifically designed for those scenarios.

Similarly to [4] we analyze all the implementations as state machines to study their complexity. For instance, we look at the number of *explicit application states*, e.g., describing that the application is waiting for the sensor device to return a reading or a message is being sent.

Reference Application. For each type of node (e.g., temperature sensors or air conditioners), the component configuration in the TinyOS version is essentially the same as in Figure 5, replacing TeenyLIME with TinyOS' `GenericComm` component. However, the TinyOS-based implementation is far more complex, as Figure 10 illustrates for some sample nesC components. For instance, the component implementing the control law for the air conditioner has 8 explicit application states using plain TinyOS, whereas only 3 states are exposed in the corresponding TeenyLIME implementation. The gains in this case are due to the ability of TeenyLIME to express data filtering as templates, and to hide communication. The former allows to delegate most of the data processing to TeenyLIME, while the latter avoids the use of state variables to keep track of pending message send. Even more evident is the gain in application states exposed in the component implementing the processing for a temperature sensor. Using plain TinyOS, developers are required to keep track of all the potential data consumers within the application code. Conversely, using TeenyLIME there is no reference to this in the application code: the middleware handles this aspect through the use of *reactions*. As a result of a reduction of explicit states in the application code, the number of code lines decreases as well, as illustrated in the second column of Figure 10. Indeed, fewer state transitions need to be expressed, and far less bookkeeping code is needed.

It is worth noticing how the above simplifications are *not* accomplished by *removing* application information. Indeed, doing so would affect the application semantics. Rather, they are obtained by *moving* the information and the related processing from the application components to TeenyLIME. This is not possible using plain TinyOS, as the abstractions provided in this case do not provide any notion of *state*. They simply provide message passing. In TeenyLIME, the content of the tuple space can be used to represent the node state, as TeenyLIME data is *persistent*: the information is kept in the tuple space until it is consumed or replaced. For instance, the last tuple output by a temperature sensor node actually represents its current state, i.e., the current sensor reading. This remains in the node's tuple space until a more up-to-date reading is available that better reflects the node state.

To illustrate this aspect, the rightmost column in Figure 10 indicates the amount of information that can be moved from the application component into TeenyLIME. This can be computed by looking first at the per-component storage due to *global variables* concerned with application data in the non-TeenyLIME version. This measures the worst-case storage needs. Then, we repeat the same computation on the TeenyLIME-based one. The results confirm the above reasoning, illustrating how a considerable portion of the application state can be managed by TeenyLIME. Remarkably, for a temperature sensor *all* the application data and related processing can be moved to the tuple space. Furthermore, as global variables easily become a possible source of race-conditions due to the presence of split-phase operations [19] and TeenyLIME reduces their number, TeenyLIME programmers can write more reliable, and less error-prone code.

The above advantages are obtained at the price of a slight increase in the size of the binary code deployed on the motes. The compiled code occupies 69 Kbytes using plain TinyOS for a node equipped with a temperature device, whereas it accounts for 80 Kbytes using TeenyLIME (including the middleware itself). In the case of the air conditioner, the program memory required increases from about 72 Kbytes to 90 Kbytes. The air conditioner is by far the most complex component configuration we have developed so far, and is well within the limits imposed by commercially available sensor platforms, e.g., 128 Kbytes of program memory in the MICA2 platform.

Alternative Scenarios. As pointed out in Section 2, different subsets of the requirements arising in our reference application are also germane to more traditional WSNs applications and services. Here, we evaluate the flexibility of TeenyLIME in addressing these alternative settings, by focusing on the aforementioned object tracking application and Mutation Routing protocol. In doing so, we compare against Hood [4], as it is representative of the state of the art in WSN programming abstractions. In [4], object tracking was taken to motivate Hood itself, and Mutation Routing was also shown to greatly benefit from a Hood-based design. Therefore, these applications are good benchmarks to compare Hood against TeenyLIME. Note how this comparison places ourselves in the worst possible situation for comparison because Hood was designed expressly to target these applications while TeenyLIME was not.

Hood is a neighborhood abstraction where nodes can identify subsets of their physical neighbors according to various criteria and share state with them. This is expressed by specifying an interest in some *attributes* exported by neighbor nodes. The application can also provide code fragments to *filter* the attributes made available by neighboring nodes. The different neighborhoods are specified using extensions to the basic nesC constructs, translated into native nesC code by a dedicated preprocessor. With respect to TeenyLIME, in Hood data-sharing is decided at compile-time. Additionally, unlike TeenyLIME, Hood does not provide the ability to remotely affect the state of another node, and does not offer abstractions to *react* to changes in the shared state. This makes it difficult for the programmer to take actions in response to changes in the system state.

In the object tracking application described in [4], Hood enables decoupling of the three main components involved, i.e., tracking, localization, and routing. Using TeenyLIME, we can achieve the same desirable design quality entirely within the TinyOS framework, without the pre-compilation step required by Hood. In addition, Hood requires an interface between the tracking and routing components for message transmission [4], which is not necessary in TeenyLIME because the two components can exchange data anonymously through the local tuple space. As for Mutation Routing, two nodes are appointed the role of source or destination node for packets flowing along a multi-hop path. As some physical phenomena move, the source (destination) role can be passed between neighboring nodes. In the Hood-based implementation, this processing is mixed with message routing, again because Hood is not able to share state between different components on the same node. In our TeenyLIME-based implementation, we have been able to modularize this same processing so as to reuse the token-based mechanism developed for the HVAC sub-task.

Furthermore, in Hood communication must be managed explicitly by the programmer, in contrast to TeenyLIME. Therefore, explicit states must be accounted for in the application code to keep track of communication aspects, e.g., a pending message send. As this is not required in a TeenyLIME-based implementation, we have been able to obtain slight improvements also in the code complexity in both these applications. For instance, the tracking component accounts for 3 explicit states only (no object, object sensed, leader election [4]) instead of 5, as in Hood.

Indeed, these results confirm that TeenyLIME enables better design and simpler code not only for sense-and-react

Parameter	Default value
Network size	100 nodes
Transmission Range	30 ft
Neighbors	11 (on average)
rdg rate	1 msg/min
Other traffic	5 msg/min
Message size	114 byte

Figure 11: Default parameters used in the simulation.

applications, but also for conventional WSN applications and services.

6.2 Evaluating the Middleware Implementation

Given the extreme resource constraints imposed by WSN devices, our evaluation must extend beyond the programming model, into the TeenyLIME implementation. In particular, we focus on the protocol for ensuring reliable operations, introduced in Section 5.2, because network communication constitutes a significant source of energy consumption in a WSN. Our evaluation is carried out using the TOSSIM [25] simulator, and leverages off our nesC-based implementation. The results we present here fully confirm that TeenyLIME deals effectively with the characteristics of WSNs, by ensuring full reliability even in scenarios with high packet loss and at the same time keeping overhead (and therefore energy consumption) well under control.

Simulation Settings. Figure 11 reports the parameters used in the simulation. We built a grid of 100 nodes, spaced 12 ft apart. During the simulation all nodes are always active. The transmission range is set to 30 ft, which in our experiments yields about 11 neighbors per node.

In the fire sub-task described in Section 4, controllers perform reliable **rdg** to read sensed values from smoke detectors, and reliable **out** to activate the sprinklers. Here, we focus on the **rdg** operation since it is the most interesting from a network perspective, given that it encompasses both broadcast (the query from the actuator to its neighbors) and unicast (the reply from the neighbors to the actuator) communication. In our simulated scenario, we assigned 40% of the nodes as controllers; after an initial period of 200 (virtual) seconds to discover neighbors, the controllers start issuing one **rdg** per minute on nearby nodes (both sensors and actuators), which always reply with the requested tuple. After 700 s, we stop the **rdg** generation and let the nodes run for other 800 s to recover as many lost messages as possible. In addition, to assess the impact of network collisions and further stress the protocol, all nodes transmit an additional 5 broadcast messages per minute. These messages model background traffic (e.g., reaction refresh or periodic data sampling) and are not sent reliably.

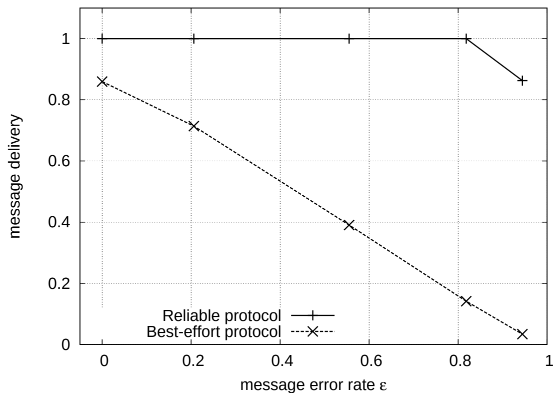
All messages, both reliable and unreliable, are fixed at the maximum size allowed by the TinyOS MAC layer, 114 bytes. This value, well beyond the needs of our application, increases the number of potential collisions and the probability of packet loss. It is worth noting that the headers added by our reliability protocol have a reasonable size of only 4 bytes per neighbor—2 for the node identifier, 1 for the message counter, and 1 for the last message received.

Finally, all results are averaged over 10 simulation runs, each with different seeds.

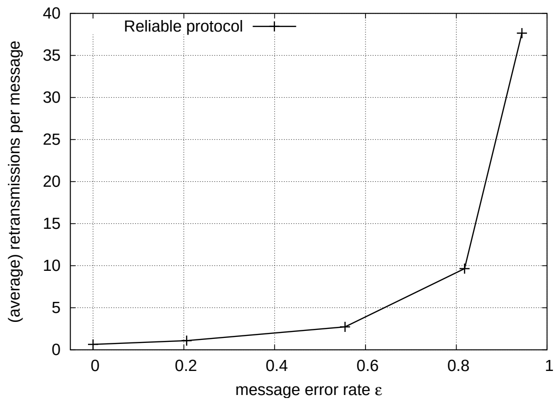
Metrics. Our evaluation focuses on *reliability* and *efficiency*, respectively in terms of message delivery and overhead. The former is defined as the ratio between the number of messages delivered (both queries and replies) and the number of messages that *should* have been delivered, determined off-line. The overhead, instead, includes the number of retransmissions required to deliver a message.

In a real WSN deployment, the rate of message loss exhibits an oscillatory behavior, often showing bursts of errors followed by stable periods. Moreover, the reliability of links vary widely, even in the context of the same application, over time. To eliminate the bias deriving from the choice of a particular pattern of unreliability, we consider an artificially constructed network where all links have the same *message error rate* ε. The performance of our protocol is then analyzed by varying ε from a value $\varepsilon = 0$, where links are reliable and network collisions are the only source of message loss, up to the extreme value of 95%.

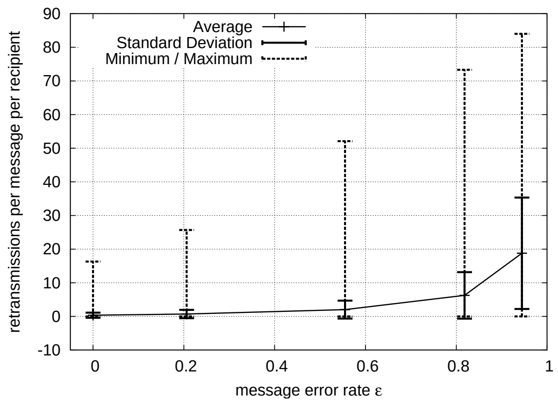
Results. The main goal of the reliability protocol is to guarantee message delivery. As shown in Figure 12(a), our solution is able to achieve this objective up to a message error rate of 80%, a quite challenging (and unrealistic) setting. The same chart also illustrates the performance of a naive solution without any delivery guarantee. The gap between this and our protocol becomes more sizable as the message error rate increases. In particular, the naive solution delivers only about 18% when $\varepsilon = 0.8$, whereas our protocol still delivers all the messages. Beyond an error rate of 80%, we



(a) Message delivery.



(b) Average number of retransmissions per message.



(c) Number of retransmissions per message per recipient.

Figure 12: Reliability protocol performance.

terminate the simulation before the protocol has a chance to recover all messages.

These very good results come at the cost of an increased number of transmissions per message. To analyze this aspect, Figure 12(b) illustrates the average number of retransmissions required against ε. Remarkably, this overhead does not grow significantly up to $\varepsilon = 0.8$. At that point, the message error rate is such that our protocol can no longer keep pace, and is forced to trigger a high number of retransmissions. Conversely, when $\varepsilon = 0$ the links are perfectly reliable, the only source of message losses are collisions at the physical level. In this case, our protocol is able to recover messages with only a small additional overhead, about 0.62 retransmissions per message.

Further insights into the cost imposed by our protocol in terms of message retransmissions can be found in Figure 12(c), where we analyze this same metric on a per-recipient basis. The chart illustrates that, although the maximum number of retransmissions for a single recipient grows significantly with ε, on average this value is almost constant, and its standard deviation is very small compared to the maximum. This illustrates that the majority of recipients receive the message after few retransmissions even for high values of ε, while only very few nodes are responsible for a high number of retransmissions.

7 Related Work

The work most closely related to TeenyLIME is Hood [4], whose abstractions have already been described in Section 6.1. It is worth noting here how the current implementation of Hood is based on periodic broadcasting of the values of all node attributes. This is not required in TeenyLIME. Rather, the programmer can trade reliability for resource consumption, according to the application needs.

Context Shadow [26] exploits multiple tuple spaces, each hosting only locally sensed information representing a given context. The application explicitly connects to one of them to retrieve information of interest. Similarly, the tuple spaces used in Agilla [27] for coordination among mobile agents are transiently shared only on the same node, and kept distinct across different hosts. Instead, data in a neighborhood is transiently shared in TeenyLIME, creating the illusion of a single address space. Furthermore, capability tuples and TeenyLIME neighborhood tuples give programmers flexible mechanisms to address specific aspects of WSNs, e.g., energy management.

Abstract Regions [12] proposes a model where $\langle key, value\rangle$ pairs are shared among the nodes in a region (i.e., a set of geographically related nodes), and manipulated through read/write operations. Differently from TeenyLIME, the programmer has no way to be notified when some particular data appears in the system. From an implementation perspective, the nodes belonging to a region can communicate despite being multiple hops away. However, each particular region requires a dedicated implementation. Therefore, their applicability is limited to specific domains.

Prior research of some of the authors has explored tuple spaces in a variety of challenging, decentralized settings, yielding a range of Linda-based middleware applicable for scenarios with varying degrees of resource restrictions [28]. TeenyLIME itself is inspired by LIME [17], which introduced *transiently shared tuple spaces* in MANETs. Compared to LIME, TeenyLIME offers features specifically targeted to WSNs, e.g., range matching and capability tuples, and explicitly addresses energy consumption due to communication. Between LIME and TeenyLIME on the scale of platform resource capabilities is TinyLIME [29], which combines comparatively resource-rich, mobile PDAs, the data consumers, with WSN sensor nodes, the data producers. In TinyLIME, tuple spaces bridge the PDAs and WSN nodes and do *not* span the sensor devices. Conversely, TeenyLIME applications are deployed directly on the WSN devices, which play an active distributed coordination role even in the absence of powerful nodes in the system.

Finally, reliable communication in WSNs is an active field of research. Solutions have been proposed both at the network and at the MAC layer. In the former case, reliability is commonly achieved by making data redundant, as in [21], or with per-hop feedback techniques, e.g., [22]. Differently from this work, these solutions target multi-hop routes leading to one or more base-stations. The requirement in TeenyLIME is instead of supporting reliable communication within a neighborhood, and in the absence of a constant data rate. At the MAC layer, reliability is usually provided by sophisticated transmission scheduling algorithms, either based on random access techniques like the one employed in [30], or on static scheduling mechanisms, e.g., as in [31]. However, these solutions often make fairly strong assumptions on constant transmission rates and the data path in the network [32]. This rules out their use in our scenarios, where the communication patterns are hard to predict due the presence of decentralized computation and reactive operations are triggered in response to application-specific events.

8 Conclusions and Future Work

This paper outlined TeenyLIME, a model and middleware supporting sense-and-react WSN applications. Our efforts have shown that the model is suitable for a wide range of applications, where it brings simpler, cleaner, more reusable designs supported by an efficient middleware implementation. Furthermore, the provision of reliable operations, supported by a novel protocol, further increases the applicability of TeenyLIME into domains demanding consistent state and guaranteed communication.

Our future plans lie primarily in two directions: reliability and multi-hop interactions. Although we have already successfully demonstrated our reliability protocol, there are several dimensions still to explore. For example, it is reasonable to allow nodes other than the original sender to retransmit lost data. While such a mechanism can balance the recovery communication load across nodes, coordination of retransmissions is required to ensure the overhead does not increase unnecessarily. Second, while this paper demonstrated how multi-hop protocols can be built from the one-hop sharing mechanisms of TeenyLIME, we intend to explore the actual expansion of the transiently shared space beyond one-hop neighbors. To this end, we plan to integrate the Logical Neighborhoods [33] abstraction, developed by some members of our group. This abstraction enables communication from a node to a set of others with specified characteristics (e.g., temperature ones). These nodes are not physical, but logical neighbors, and communication is therefore supported by an efficient multi-hop routing. In our context, the TeenyLIME middleware would benefit not only of this latter ability, but its model could leverage off Logical Neighborhoods also for providing scoping in the definition of data sharing.

Our implementation of the TeenyLIME middleware, including the reliable protocol described in Section 5.2, is available for download at `http://lime.sf.net/teenyLime.html`.

References

[1] Habitat Monitoring on the Great Duck Island. `www.greatisland.net`.

[2] A. Deshpande, C. Guestrin, and S. Madden, “Resource-aware wireless sensor-actuator networks,” *IEEE Data Engineering*, vol. 28, no. 1, 2005.

[3] D. Gelernter, “Generative communication in Linda,” *ACM Computing Surveys*, vol. 7, no. 1, pp. 80–112, January 1985.

[4] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler, “Hood: A neighborhood abstraction for sensor networks,” in *Proc. of* 2^{nd} *Int. Conf. on Mobile systems, applications, and services*, 2004.

[5] P. Costa, L. Mottola, A. Murphy, and G. Picco, “TeenyLIME: Transiently Shared Tuple Space Middleware for Wireless Sensor Networks,” in *Proc. of the* 1^{st} *Int. Workshop on Middleware for Sensor Networks (MidSens)*, 2006.

[6] E. Petriu, N. Georganas, D. Petriu, D. Makrakis, and V. Groza, “Sensor-based information appliances,” *IEEE Instrumentation and Measurement Mag.*, vol. 3, pp. 31–35, 2000.

[7] C. Manzie, H. C. Watson, S. K. Halgamuge, and K. Lim, “On the potential for improving fuel economy using a traffic flow sensor network,” in *Proc. of the Int. Conf. on Intelligent Sensing and Information Processing*, 2005.

[8] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar, “Next century challenges: scalable coordination in sensor networks,” in *Proc. of the* 5^{th} *Int. Conf. on Mobile computing and networking (MobiCom)*, 1999.

[9] I. F. Akyildiz and I. H. Kasimoglu, “Wireless sensor and actor networks: Research challenges,” *Ad Hoc Networks Journal*, vol. 2, no. 4, pp. 351–367, October 2004.

[10] J. Al-Karaki and A. E. Kamal, “Routing techiniques in wireless sensor networks: a survey,” *IEEE Wireless Communications*, vol. 11, no. 6, 2004.

[11] T. Abdelzaher *et al.*, “Envirotrack: Towards an environmental computing paradigm for distributed sensor networks,” in *Proc. of the* 24^{th} *Int. Conf. on Distributed Computing Systems*, 2004.

[12] M. Welsh and G. Mainland, “Programming sensor networks using abstract regions,” in *Proc. of the* 1^{st} *Symp. on Networked Systems Design and Implementation*, 2004.

[13] K. Whitehouse and D. Culler, “Calibration as parameter estimation in sensor networks,” in *Proc. of the* 1^{st} *Int. Wkshp. on Wireless sensor networks and applications*, 2002.

[14] S.R. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, “TinyDB: An acquisitional query processing system for sensor networks,” *ACM Trans. Database Syst.*, vol. 30, no. 1, 2005.

[15] J. Elson and K. Roemer, “Wireless sensor networks: a new regime for time synchronization,” *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 1, 2003.

[16] A. L. Murphy, G. P. Picco, and G.-C. Roman, “LIME: A Middleware for Physical and Logical Mobility,” in *Proc. of the* 21^{st} *Int. Conf. on Distributed Computing Systems (ICDCS)*, 2001.

[17] ——, “LIME: A coordination model and middleware supporting mobility of hosts and agents,” *ACM Trans. on Software Engineering and Methodology (TOSEM)*, vol. 15, no. 3, pp. 279–328, July 2006.

[18] A. Rowstron, “WCL: A coordination language for geographically distributed agents,” *World Wide Web Journal*, vol. 1, no. 3, pp. 167–179, 1998.

[19] D.Gay, P. Levis, and R. von Behren, “The NesC language: A holistic approach to networked embedded systems,” in *Proc. of the ACM Conf. on Programming Language Design and Implementation*, 2003.

[20] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva, “Directed diffusion for wireless sensor networking,” *IEEE/ACM Trans. Networking*, vol. 11, no. 1, 2003.

[21] B. Deb, S. Bhatnagar, and B. Nath, “ReInForM: Reliable information forwarding using multiple paths in sensor networks,” in *Proc. of the* 28th *IEEE Int. Conf. on Local Computer Networks*, 2003.

[22] C. Y. Wan, A. T. Campbell, and L. Krishnamurthy, “Reliable transport for sensor networks: PSFQ—Pump slowly fetch quickly paradigm,” *Wireless sensor networks*, 2004.

[23] TinyOS Official Source Tree. `www.tinyos.net`.

[24] R. Gummadi, O. Gnawali, and R. Govindan, “Macro-programming wireless sensor networks using Kairos,” in *Proc. of the* 1st *Int. Conf. on Distributed Computing in Sensor Systems*, 2005.

[25] P. Levis, N. Lee, M. Welsh, and D. Culler, “TOSSIM: Accurate and scalable simulation of entire TinyOS applications,” in *Proc. of the* 5th *Symp. on Operating Systems Design and Implementation*, 2002.

[26] M. Jonsson, “Supporting context awareness with the context shadow infrastructure,” in *Wkshp. on Affordable Wireless Services and Infrastructure*, June 2003.

[27] C.-L. Fok, G.-C. Roman, and C. Lu, “Rapid development and flexible deployment of adaptive wireless sensor network applications,” in *Proc. of the* 25th *IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, 2005.

[28] A. L. Murphy and G. P. Picco, “Transiently shared tuple spaces for sensor networks,” in *Proc. of the Euro-American Workshop on Middleware for Sensor Networks*, 2006.

[29] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. L. Murphy, and G. P. Picco, “Mobile data collection in sensor networks: The TinyLime middleware,” *Elsevier Pervasive and Mobile Computing Journal*, vol. 4, no. 1, pp. 446–469, Dec. 2005.

[30] T. van Dam and K. Langendoen, “An adaptive energy-efficient MAC protocol for wireless sensor networks,” in *Proc. of the* 1st *Int. Conf. on Embedded networked sensor systems (SENSYS)*, 2003.

[31] V. Rajendran, K. Obraczka, and J. J. Garcia-Luna-Aceves, “Energy-efficient, collision-free medium access control for wireless sensor networks,” *Wirel. Netw.*, vol. 12, no. 1, 2006.

[32] P. Naik and K. M. Sivalingam, “A survey of MAC protocols for sensor networks,” *Wireless sensor networks*, pp. 93–107, 2004.

[33] L. Mottola and G. P. Picco, “Logical Neighborhoods: A programming abstraction for wireless sensor networks,” in *Proc. of the* 2nd *Int. Conf. on Distributed Computing on Sensor Systems (DCOSS)*, 2006.