Programming Wireless Sensor Networks with 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, Switzerland
murphy@itc.it
[4] University of Trento, Italy
picco@dit.unitn.it

Abstract. Wireless sensor networks (WSNs) are evolving to support sense-and-react applications, where actuators are physically interspersed with the sensors that trigger them. This solution maximizes localized interactions, improving resource utilization and reducing latency w.r.t. solutions with a centralized sink. Nevertheless, application development becomes more complex: the control logic must be embedded in the network, and coordination among multiple tasks is needed to achieve the application goals.

This paper presents TeenyLIME, a WSN middleware designed to address the above challenges. TeenyLIME provides programmers with the high-level abstraction of a tuple space, enabling data sharing among neighboring devices. These and other WSN-specific constructs simplify the development of a wide range of applications, including sense-and-react ones. TeenyLIME yields simpler, cleaner, and more reusable implementations, at the cost of only a very limited decrease in performance. We support these claims through a source-level, quantitative comparison between implementations based on TeenyLIME and on mainstream approaches, and by analyzing measures of processing overhead and power consumption obtained through cycle-accurate emulation.

Keywords: Wireless sensor and actuator networks, middleware, tuple spaces.

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., [2,3,4]) now include nodes hosting actuators, able to *react* to external stimuli gathered by nearby sensors and affect the environment under control.

The sense-and-react pattern has a relevant impact on application development. Appropriate programming constructs are required to deal with the increased complexity of specifying how multiple tasks *coordinate* to accomplish the desired global functionality.

R. Cerqueira and R.H. Campbell (Eds.): Middleware 2007, LNCS 4834, pp. 429–449, 2007.

Dedicated abstractions must be provided to describe the *stateful* interactions commonly present in control mechanisms. The ability to locally *react* based on external stimuli is as important as—if not more important than—the ability to gather data. These aspects are discussed in more detail in Section 2, where we both describe a paradigmatic sense-and-react application and illustrate that many of its characteristics are typical of common sense-only applications and lower-level system functionality.

To meet the requirements above we developed TeenyLIME, a WSN middleware whose foundation is the notion of distributed *tuple space* [5], a repository of elementary sequences of typed fields called tuples. This is revisited in TeenyLIME by considering WSN requirements (e.g., resource consumption and reliability) in the programming model. TeenyLIME adopts a minimalist approach: a limited number of powerful operations, with a simple and yet efficient implementation, allow for the development of both application-level and system-level functionality. An overview of TeenyLIME's base concepts and application programming interface (API) is provided in Section 3, while 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. Section 5 provides a concise account of the TeenyLIME architecture.

Section 6 evaluates quantitatively TeenyLIME along two dimensions. First, we assess the effectiveness of its *programming model* in different contexts. We examine the implementation of the reference application, whose design we sketched in Section 4, and report about uses of TeenyLIME in sense-only applications and at the system level. We derive code metrics for the TeenyLIME implementations and their counterparts, implemented using plain nesC or the higher-level support provided by Hood [6]. Results indicate that the expressive power of TeenyLIME yields cleaner, simpler, and more compact code. Second, we analyze the TeenyLIME *implementation*. We compare its overhead, in terms of processing time and energy consumption, against existing programming platforms. The results gathered using cycle-accurate emulation demonstrate that the beneficial higher level of abstraction provided by TeenyLIME comes with only a very limited overhead.

Finally, existing node-level abstractions for WSN programming are reviewed in Section 7, before our concluding remarks in Section 8.

A preliminary description of TeenyLIME appeared in a short paper [7]. Here, in addition to a more precise and exhaustive presentation, we illustrate key aspects entirely missing in [7], namely: *i)* a complete TeenyLIME-based design of a sense-and-react application; *ii)* a quantitative, source-level evaluation of the benefits to the programmer; *iii)* a quantitative, cycle-accurate evaluation of the run-time performance.

2 Scenario and Motivation

Sense-and-react applications emerge in many settings, from home automation [3] to road traffic control [4]. As a paradigmatic example, we consider *building monitoring and control*. Modern buildings typically focus on the following functionality:

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.

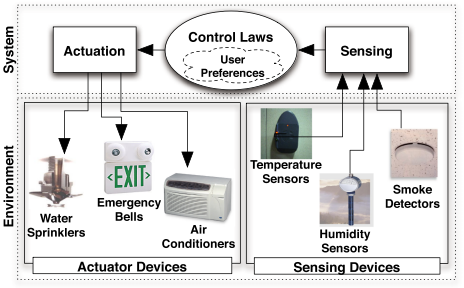


Fig. 1. High-level scheme of a building monitoring and control application

These applications, as with any other embedded control system, feature four main components, illustrated in Figure 1. The *user preferences* represent the high-level system goals, e.g., the desired temperature in the building and the need to limit fire spreading. *Sensing devices* gather data from the environment and monitor relevant 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: 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 the user preferences. In our case, a (simplified) control loop may activate air conditioners when temperature deviates significantly from the user preferences, tuning this action based on the humidity in the same location. Further, it may immediately activate emergency bells when the temperature increases above a safety threshold, but operate water sprinklers only if smoke detectors actually report the presence of fire. Oscillating behaviors must be avoided in all situations.

Application development in these scenarios is complicated not only by the peculiarities of devices, but also by the complexity of their interactions. The many requirements can be grouped into high-level challenges common to several settings:

- *Localized computations* [8] must be privileged, to keep processing close to where sensing or actuation occurs. In sense-and-react applications it is indeed unreasonable to funnel all the sensed data to a single base-station, as this may negatively affect latency and reliability, without any significant advantage [9].
- The system performs *multiple tasks* in parallel. In our example, two control laws coexist: one for air conditioning, the other for handling emergencies. These need to *share data* (e.g., temperature readings) generated by a subset of the sensing devices.
- Differently from sense-only scenarios, sense-and-react applications often require *stateful* coordination, e.g., using current shared conditions (state) to act collaboratively. This, in combination with the use of WSNs for safety critical applications, motivates an explicit account for *reliability* in the programming model.
- *Reactive interactions*, actions that automatically fire based on external conditions, assume a prominent role. In our case, a temperature reading deviating from user preferences triggers an action in both of the two application tasks. *Proactive interactions*, common in many sense-only scenarios, are still needed to gather information and

fine tune the actuation about to occur. For instance, the sprinklers in the building ask for smoke readings before taking any action.

Note how sense-*and*-*react* scenarios essentially subsume sense-*only* ones. Therefore, the aforementioned requirements represent the most general set of application-level issues WSN developers must cope with. Also, subsets of these requirements must be accounted for at lower levels, below the application. For instance, localization algorithms [10]—often one of the many tasks of object tracking applications [11]—must rely on localized interactions, as most of the approaches in the field base the position estimation on data reported by nearby hosts. Similarly, multi-hop routing mechanisms [12] require reactive interactions to adapt to mutable network conditions, and may also exploit reliable operations to guarantee message delivery [13]. The TeenyLIME programming model, described next, supports application development without losing the ability to express system-level mechanisms.

3 TeenyLIME: Basic Concepts and API

TeenyLIME is based on the *tuple space* abstraction, originally proposed in Linda [5], and here re-elaborated in the context of WSNs. A tuple space is a repository of data represented as *tuples*, sequences of typed fields such as ⟨"foo", 29⟩. Three core Linda operations allow processes to manipulate the tuple space by creating (**out**), reading (**rd**), and removing (**in**) tuples. Tuple selection with **rd** and **in** is based on matching patterns such as ⟨"foo", ?integer⟩ against the tuple space content. Patterns may use either *actual* or *formal* values, the latter serving as a kind of "wild card" matching any data of a particular type.

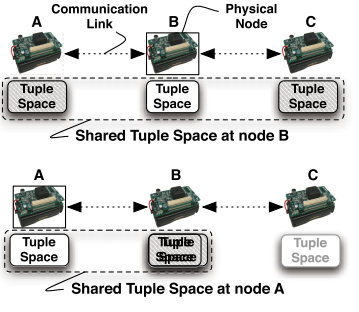


Fig. 2. Tuple space sharing in TeenyLIME

In Linda, the tuple space is assumed globally accessible to all processes—an undesirable choice in WSNs. Instead, in TeenyLIME each node hosts a tuple space, shared among nodes within direct (one-hop) communication range. *Sharing* means that a node views its local tuple space as containing its own tuples, plus those in the tuple spaces hosted by its neighbors, as shown in Figure 2. Operations span the whole shared tuple space. For instance, a query issued by a node may return a matching tuple found in any tuple space in the one-hop neighborhood—including the local one. Therefore, TeenyLIME programmers can specify interactions among nodes abstractly, by focusing on the application logic (e.g., reading temperature in the neighborhood) and leaving system configuration issues (e.g., tracking node identity and presence) to the middleware.

The choice to limit sharing to one-hop neighbors is motivated by the fact that interactions with these nodes are the most frequent in WSNs. Whitehouse et al. analyzed 16 publicly available applications to determine the node interactions, and

"All neighborhoods discovered were one-hop neighborhoods [...]" ([6], p.9)

```
interface TupleSpace {
  // Standard tuple space 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();
}
```

Fig. 3. TeenyLIME API

Interestingly, all neighborhoods were of limited size (at most ten nodes), and were used either directly at the application level to gain access to *nearby* information, or as a building block for lower-level system functionality, e.g., to implement multi-hop routing. These considerations also support our design choice, drawing the foundations for a highly-reusable programming model supported a by lightweight, scalable implementation. Furthermore, it should be noted that the applications considered in [6] were conventional sense-only ones. Sense-and-react applications exacerbate the need for localized interactions [8], and are therefore expected to benefit even more from our design. As a result, the TeenyLIME programming model can be used in many contexts, ranging from sense-and-react to sense-only, and from application-level to system-level.

Figure 3 shows the TeenyLIME API. While in principle the programming model is independent of the node platform, we present here the API in nesC, as our middleware is currently built on top of TinyOS. The interface provides the operations to manipulate TeenyLIME's shared tuple space. The first three operations correspond to the Linda operations discussed earlier, while **rdg** and **ing** are variants (as in [14]) that return all matching tuples, instead of a single match.

TeenyLIME operations are asynchronous, allowing the application to continue while the middleware completes the operation execution[1]. This approach blends well with the event-driven concurrency model of nesC. Therefore, all operations are *split-phase* [15]: the operation is issued, and later the `tupleReady` event is signaled when the operation completes. The `tupleReady` event contains an identifier allowing the application to associate the event with its earlier request. Depending on the operation, one or more tuples, indicated by the `number` parameter, may also be contained in the event.

The operations provided in the API deserve further discussion. However, instead of describing them in isolation, in the next section we discuss them "in action", i.e., hand-in-hand with the TeenyLIME-based design of our reference application.

[1] In most Linda systems **rd** and **in** are blocking, i.e., do not return until a tuple is matched.

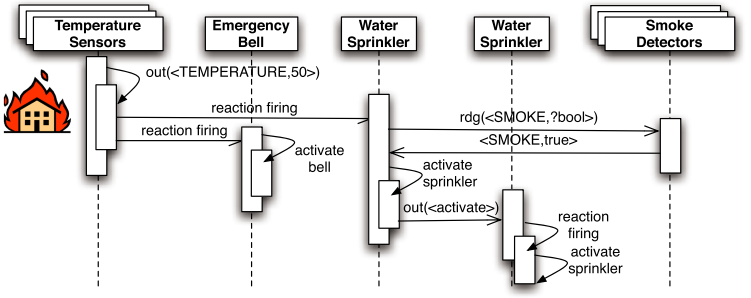


Fig. 4. Sequence of operations to handle a fire. Notified about increased temperature, a node controlling water sprinklers queries the smoke detectors to verify the presence of fire. If necessary, it sends a command activating nearby sprinklers.

4 Application Development with TeenyLIME

As discussed in Section 2, our reference application contains two sub-tasks, one managing the air conditioning system (HVAC) and the other for emergency situations such as fire. Each sub-task involves different types of nodes, e.g., humidity sensors in the HVAC sub-task, and smoke detectors to address fire emergencies. Temperature sensors are instead used in both sub-tasks. For all types of nodes, the application processing has been implemented in a single component sitting entirely on top of the `TupleSpace` interface, which masks completely TinyOS' generic communication layer. An additional component is employed to interact with the sensors/actuators attached to the node.

In the following, we explain the rest of our reference application's design and implementation. We illustrate how we exploit data sharing and related operations, and how interactions among nodes benefit from the WSN-specific API features. Throughout, the reference application is used as a motivation and source of examples for the discussion.

Sharing Application Data through Proactive and Reactive Interactions. In our design, *sensed data* and *actuating commands* take the form of tuples. These are shared across nodes (and components on the same node) to enable coordination of activities as well as data communication. Access to this data can occur *proactively*, e.g., using the **rd** and **in** operations. However, TeenyLIME supports also a notion of *reaction*, a code fragment whose execution is automatically triggered upon the appearance of a given tuple anywhere in the shared tuple space. The tuples of interest are identified through pattern matching, and the `tupleReady` event is used to signal a reaction firing. This provides an easy and yet very powerful way to monitor changes in the neighbors' data through the content of the shared tuple space.

Figure 4 uses the fire control sub-task to illustrate how proactive and reactive interactions are used together to trigger notifications, to perform distributed operations for gathering data from neighboring nodes, and to request actuation commands. Notably, similar patterns of interactions recur in both sub-tasks of our application.

Both emergency bells and water sprinklers have a reaction registered on their neighbors, watching for temperature tuples, as shown in the code in Figure 5. Temperature sensors periodically take a sample and pack it in a tuple, which is then stored in the local

```
command result_t StdControl.start() {
  tuple tempTemplate = newTuple(2, actualField_uint16(TEMPERATURE),
                                   formalField(TYPE_UINT16_T));
  call TS.addReaction(FALSE, TL_NEIGHBORHOOD, &tempTemplate);
  return SUCCESS;
}
 event result_t TS.tupleReady(TLOpId_t operationId,
                              tuple *tuples, uint8_t number) {
   // Notification triggered ...
}
```

Fig. 5. TeenyLIME code for an actuator node interested in temperature values

```
command result_t StdControl.start() {
  return call SensingTimer.start (TIMER_REPEAT, SENSING_TIMER);
}
event result_t SensingTimer.fired() {
  return call TemperatureSensor.getData();
}
event result_t TemperatureSensor.dataReady(uint16_t reading){
  tuple temperatureValue = newTuple(2, actualField_uint16(TEMPERATURE),
                                       actualField_uint16(reading));
  call TupleSpace.out(FALSE,TL_LOCAL,&temperatureValue);
  return SUCCESS;
}
```

Fig. 6. TeenyLIME code for a temperature node

tuple space, as shown in Figure 6. Insertion is accomplished using **out** by setting the `target` parameter to `TL_LOCAL`, which entails outputting the tuple to the local tuple space. This operation, by virtue of one-hop sharing, automatically triggers all the aforementioned reactions[2], which process the tuple contained in the event `tupleReady`. However, different types of actuator nodes behave differently when high temperatures are detected. The node hosting the emergency bell immediately activates its device. Instead, the water sprinkler node proceeds to verify the presence of fire, as shown in Figure 4. The latter behavior, specified as part of the reaction code, consists of proactively gathering the readings from nearby smoke detectors, using a **rdg** restricted (by setting `target` to `TL_NEIGHBORHOOD`) to the union of their tuple spaces. If fire is reported, the water sprinkler node requests activation of nearby sprinklers through a two-step process that relies on reactions as well. The node requesting actuation inserts a tuple representing the command on the nodes where the activation must occur, using **out** with `target` set to the sprinkler node address. The presence of this tuple triggers a locally-installed reaction delivering the activation tuple to the application, which reads the tuple fields and operates the actuator device accordingly.

Reliable Operations. Since fire detection requires the maximum degree of reliability, its implementation takes advantage of *reliable operations* for guaranteeing correct communication of reactions and query results of the **rdg** operation on smoke detectors and

[2] We assume that actuators are interested in all temperature values. We show later how notifications can be triggered only when temperature is above (or below) a given threshold.

of the **out** operations towards actuators. Furthermore, 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, achieving a sort of distributed mutual exclusion. The master node is identified as the one holding a special token tuple, periodically exchanged among co-located nodes to achieve a form of load-balancing. As a token loss implies no controller acting as the master, strong guarantees on token transfer are imperative. Therefore, the token exchange from the previous to the new master node is accomplished using a reliable **in** operation performed by the latter.

As shown in Figure 3, 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 applications (e.g., data collection), while the latter offer stronger guarantees to applications requiring stateful interactions.

Sharing System Data. 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. In our scenario, these tuples contain a field describing the (logical) location (e.g., a room) where a node is deployed, and the sensor/actuator devices attached. Which data to provide is defined by the application programmer, by specifying the body of the handler for the `reifyNodeTuple` event, shown in Figure 3. 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 on every 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 neighbors, they extract the neighbor tuple and insert it locally. This way, it is easy to query the tuple space to obtain information on neighbors with specific capabilities.

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

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

The above uses the default range matching, which the programmer can easily redefine.

Note how the issue is *not* simply one of expressive power, as it deeply affects communication. Without filtering, the programmer can only specify a *generic* pattern matching

```
command result_t StdControl.start(){
  tuple capTSmoke = newCapabilityTuple(2, actualField_uint16(SMOKE),
                                          formalField(TYPE_BOOL));
  call TupleSpace.out(FALSE,TL_LOCAL,&capTSmoke);
  return SUCCESS;
}
event result_t  TupleSpace.reifyCapabilityTuple(tuple *ct, tuple *p){
  return call SmokeDetector.getData(); // Request a reading from the sensor
}
event result_t SmokeDetector.dataReady(uint16_t reading){ // Sensor reading
  tuple smokeValue = newTuple(2, actualField_uint16(SMOKE),
                                 actualField_bool(reading));
  call TS.out(FALSE,TL_LOCAL,&smokeValue);
  return SUCCESS;
}
```

Fig. 7. TeenyLIME code for a smoke detector node

any temperature. All matching, outputted tuples would be transmitted (in our case, each time a new sample is available) and frequently discarded as out of range by the reaction code of the requester in Figure 5, wasting significant communication resources.

Dealing with Short-Lived Data. In some cases, sensor data remain useful only for a limited time after collection. For instance, an emergency bell is not interested in temperature values sensed an hour before. Instead, the same data may be of interest for a component that is periodically run 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 range matching above, 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.

Generating Data Efficiently. In our application, humidity sensors and smoke detectors need not be monitored continuously: their data is accessed only when actuation is about to occur. However, when a sensed value is requested (e.g., by issuing a **rd**) fresh-enough data must be present in the tuple space. If these data are 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.

To deal with these (frequent) situations, TeenyLIME provides the ability to output *capability tuples* indicating that a device has the capability to produce data of a given pattern. A code example for a smoke detector is shown in Figure 7. 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. The sequence of operations is depicted in Figure 8. Note how, from the perspective of the data consumer, nothing changes. Instead, on the side of the data producer, capability tuples enable considerable energy savings as the readings are taken only on-demand, without the need to maintain constantly fresh data in the tuple space.

Interestingly, capability tuples can be generalized to allow *any action* to be taken by the data producer. For example, matching a pattern to a capability tuple may invoke any application function (e.g., computing the average of all temperature tuples), whose results are inserted in the tuple space and returned to the requester.

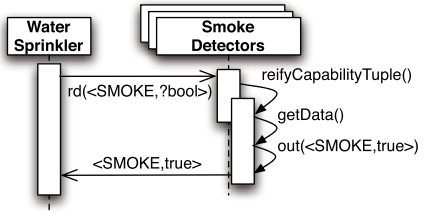


Fig. 8. Processing of capability tuples

5 The TeenyLIME Middleware

The design of TeenyLIME aims at enabling easy customization and extension of the middleware. Therefore, *local* processing, *distributed* processing, and *communication* concerns are fully decoupled, and one aspect can be changed without impact on the rest of the system. Due to space constraints, here we focus only on a few aspects of our architecture, namely, the implementation of *distributed reactions* and *capability tuples*, and the support for *reliable operations*. More details about the current prototype are reported in [16].

The implementation of *remote reactions* currently rely on a *soft-state* approach, to deal with nodes joining or failing. Each node periodically sends a message containing control data for all reactions that should be installed on its neighbors. Upon receipt of this message, a timer associated with installed reactions is refreshed. If and when a timer expires, the corresponding reaction is removed. This may happen either because the registering node became unreachable, or the application deregistered the reaction thus no longer refreshing it. Similar approaches are widely used in WSN, (e.g., in [17]), as they are sufficiently lightweight and effective.

Processing *capability tuples* requires keeping track of the source nodes whose query matched a local capability tuple so that, once the actual tuple is (locally) output by the application, it can be returned to the appropriate node. Due to nesC split-phase operations [15], this processing requires a lot of bookkeeping code. However, we noted that this processing is the same *as if* a reaction (for the same pattern as the query) were installed by a neighbor *before* the application outputs the actual tuple. Our implementation exploits this observation and installs a local reaction for the query pattern before firing the `reifyCapabilityTuple` event. When the node 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 reaction right after it fires. This solution only requires 24 nesC lines.

Finally, TeenyLIME poses only a single requirement on the communication layers: the ability to overhear messages for populating the tuple space with neighbor tuples. As a result, many existing solutions (e.g., [18, 19]) can be employed to provide *reliable operations*. Nevertheless, if reliability is only seldom required, the solutions above may be overkill, e.g., because scheduling mechanisms (as in [19]) negatively impact latency. To meet scenarios where reliable operations are rare, our current prototype includes a simple reliability scheme based on explicit acknowledgments. Messages contain a unique identifier, reported in the corresponding acknowledgment when transmission succeeds. Therefore, lost packets are easily recognized and retransmitted upon timeout expiration. Control information is piggybacked on application messages whenever possible, to reduce overhead. Our protocol is not tied to TeenyLIME, and exports the same interface as TinyOS' generic communication layer. Therefore, it can be re-used by plain TinyOS applications demanding reliable communication. More details on its internals and performance can be found in [16].

6 Evaluation

We compare quantitatively TeenyLIME against common alternatives, analyzing its impact on the application source code and on run-time performance.

6.1 Evaluating the Programming Model

Our objective is to assess the effectiveness of TeenyLIME in enabling a flexible design and clean implementations. To the best of our knowledge, there are no programming abstractions expressly designed for application scenarios such as sense-and-react. Therefore, we compare a TeenyLIME-based implementation of our reference application against one implemented directly on top of TinyOS. On the other hand, the applicability of TeenyLIME goes beyond sense-and-react applications, and reaches into system-level mechanisms, below the application layer. We substantiate this claim by reporting about implementations in both TeenyLIME and Hood [6], a programming abstraction designed around similar requirements.

Reference Application. In the TinyOS version of our reference application, each type of node (e.g., temperature sensors or air conditioners) has a component configuration similar to the one mentioned in Section 4, where however TeenyLIME is replaced by the TinyOS `GenericComm` component[3]. However, the TinyOS-based implementation is far more complex. The reader can informally verify this statement by visually comparing the *excerpt* of TinyOS code for a temperature sensor in Figure 9 against the *complete* (and *much* simpler) TeenyLIME-based equivalent shown earlier in Figure 6. The superior expressive power of TeenyLIME manifests itself in several aspects:

– Developers using plain TinyOS must keep track within the application code of all the potential data consumers. This requires several dedicated functions, such as `matchesInterest()` in Figure 9. Using TeenyLIME, the same functionality is achieved using *reactions*: no application-level bookkeeping is required.

[3] Or with our reliability component if reliable interactions, not supported by TinyOS, are required by the application. We elaborate further on reliability in Section 6.2.

```
bool pendingMsg, pendingReading;
TOS_Msg sendMsg, queueMsg[MAX_QUEUE_SIZE];
uint8_t nextQueueMsg, lastQueueMsg;
nodeInterest interests[MAX_AIR_CONDITIONERS];
void interest(uint16_t node,uint8_t t,uint16_t tShold,uint16_t tStamp){ // ... }
bool isRecipient(struct InterestMsg* msg,uint16_t nodeId) { // ... }
bool matchesInterest(uint16_t reading) { // ... }
bool enqueueMsg(TOS_Msg msg) { // ... }
bool messageWaiting() { // ... }
bool sendQueuedMsg() { // ... }
command result_t StdControl.start() {
  // ... data initialization ...
  return call SensingTimer.start(TIMER_REPEAT, SENSING_TIMER);
}
event result_t SensingTimer.fired() {
  pendingReading = TRUE;
  return call TemperatureSensor.getData();
}
event TOS_MsgPtr ReceiveInterestMsg.receive(TOS_MsgPtr m) {
  struct InterestMsg* payload = (struct InterestMsg*) m->data;
  if (!pendingReading && isRecipient(payload, TOS_LOCAL_ADDRESS))
    interest(payload->sender, payload->type,
             payload->threshold, payload->timestamp);
  return m;
}
event result_t TemperatureSensor.dataReady(uint16_t reading){
  TOS_Msg msg;
  struct DataMsg* payload = (struct DataMsg*) msg->data;
  payload->sender = TOS_LOCAL_ADDRESS;
  payload->type = TEMPERATURE;
  payload->value = reading;
  if (!pendingMsg && matchesInterest(reading)) {
    atomic {
      pendingMsg = TRUE;
      sendMsg = msg;
    }
    if (call SendDataMsg.send(TOS_BCAST_ADDR,
                              sizeof(struct AppMsg),&sendMsg)!= SUCCESS) {
        pendingMsg = FALSE;
    }
  } else if (pendingMsg)
    enqueueMsg(msg);
  pendingReading = FALSE;
  return SUCCESS;
}
event result_t SendDataMsg.sendDone(TOS_MsgPtr msg, result_t success) {
  if (msg == sendMsg) pendingMsg = FALSE;
  if (messageWaiting()) sendQueuedMsg();
  return SUCCESS;
}
```

Fig. 9. A temperature node in our reference application, using plain TinyOS. The processing above is equivalent to the TeenyLIME version in Figure 6.

- Figure 9 contains two separate execution flows: one begins when a message is received (`ReceiveInterestMsg.receive`), the other when a reading from the sensing device is ready (`TemperatureSensor.dataReady`). These two flows are not at all evident in the code, due to nesC split-phase operations [15]. Thus, maintenance and debugging are greatly complicated [20]. This problem is significantly alleviated using TeenyLIME, as only the latter execution flow is necessary.
- Distributed processing forces TinyOS programmers to delve into the details of message transmission, parsing, and buffering, therefore mixing communication aspects

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

Fig. 10. Comparing the TeenyLIME-based implementation against plain TinyOS. ML represents the maximum number of different locations the component implementing token exchange handles, NC represents the maximum number of air conditioners around a temperature sensor.

with the application semantics. Instead, the TeenyLIME component in Figure 6 contains only *application-specific* processing related to the actual *data of interest*.

– As a consequence of all the above, TinyOS programmers must manage *state variables* to deal with nearby air conditioners (`interests`), the sensing device (`pendingReading`), and the radio (`pendingMsg`). These can be the source of race conditions [15]. Conversely, in TeenyLIME these aspects are either handled by the middleware, or no longer required.

A good way to assess the complexity of implementations is to analyze them as state machines and count the number of *explicit application states*, as in [6]. These are typically stored in state variables, modified by commands and event handlers to express state transitions. The higher the number of application states, the harder it is to express state transitions [20], and the more complex and error-prone applications become.

Figure 10 reports this and other metrics for the temperature sensor and other components of our sense-and-react application, showing that the advantages of TeenyLIME hold for all the (diverse) tasks of our application. For instance, the plain-TinyOS component implementing the air conditioner control law has 8 explicit application states, whereas the TeenyLIME-based one has only 3. The reduction is due to the aforementioned ability of TeenyLIME to hide communication details, here complemented by the ability to express data filtering as patterns. The former avoids the use of several state variables, while the latter delegates most of the data processing to the middleware. Nicely, the reduction of explicit states in the application code causes the *number of lines of code* to decrease as well, as shown in the second column of Figure 10. Indeed, fewer state transitions, and therefore far less bookkeeping code, are needed.

It is worth noting that the above simplifications are *not* accomplished by *removing* application information. Doing so would indeed affect the application semantics. Rather, they are obtained by *moving* information and related processing from the application components into TeenyLIME. This is not possible using plain TinyOS, as its abstractions provide only message passing and do not explicitly represent *state*. This is instead achieved in TeenyLIME using the tuple space, as its content is *persistent*. For instance, a reading tuple output by a temperature sensor node represents its current state and remains in its tuple space until a new reading becomes available.

To quantify this aspect, the rightmost column in Figure 10 indicates the amount of information that can be moved from the application component into TeenyLIME, expressed as the percentage ratio between the TeenyLIME-based and the TinyOS-based applications. We compute it by looking at the per-component storage of *global variables*

concerned with application data. The results confirm the reasoning above, showing that a considerable portion of the application state can be managed inside the middleware. Remarkably, *all* the application data and related processing for a temperature sensor can be moved into the tuple space, as shown by comparing Figure 6 and 9.

The advantages above come at the price of a slight increase in the size of the binary code deployed on the motes. The code of a temperature node occupies 69 Kbytes using plain TinyOS and 80 Kbytes using TeenyLIME (including the middleware itself). These figures increase to 72 Kbytes and 90 Kbytes, respectively, for the air conditioner. We note, however, that the latter is a complex component, and yet it remains well within the limits imposed by commercially available sensor platforms (e.g., 128 Kbytes for MICA2).

Sense-only Applications and System-level Functionality. TeenyLIME provides relevant benefits also to the development of sense-only applications and system-level functionality. We support this statement by illustrating insights obtained by re-implementing some of the applications used in [6] to evaluate Hood, a programming abstraction geared towards sense-only applications and system mechanisms that, like TeenyLIME, focuses on one-hop interactions. Notably, by limiting ourselves to sense-only (instead of sense-and-react) applications, and comparing against Hood on the same applications used for its evaluation, we put ourselves in the most challenging situation.

Specifically, we consider the object tracking application and the multi-hop routing protocol called Mutation Routing, both described in [6]. In these applications, the evaluation using the same *quantitative* metrics considered earlier for plain-TinyOS applications shows that TeenyLIME achieves slight improvements also w.r.t. Hood. For instance, only three explicit application states are needed to implement Mutation Routing, whereas five states are required using Hood. Space constraints prevent us from an in-depth discussion of these aspects, available in [16]. Instead, we draw *qualitative* considerations showing that TeenyLIME yields cleaner and more reusable designs:

- TeenyLIME achieves a *more flexible software architecture* w.r.t. Hood. In object tracking, for instance, three components cooperate on a node to implement the desired processing: a localization algorithm, a tracking mechanism, and a geographical routing protocol. In Hood, the three need to be *wired* together using dedicated nesC interfaces. Therefore, adding a further component (e.g., to log the position of the moving object on external memory) requires modifications in several places. Instead, in TeenyLIME the three components are fully *decoupled*, and exchange data *anonymously* through the local tuple space. Thus, adding a logging component can be easily achieved without affecting the rest of the application.
- TeenyLIME fosters *code re-use* to a great extent. For instance, in Mutation Routing two nodes are appointed the role of source or destination for packets flowing along a multi-hop path. The source (destination) role must be passed between neighboring devices as some physical phenomena moves. In a TeenyLIME-based implementation, this processing can be accomplished by reusing *as is* the component implementing the token-based, mutual exclusion mechanism described in Section 4. Simply, we create a token for each role at system start-up, exchanged based on the presence of the moving target close to a given node. In Hood this functionality is interspersed with message processing, preventing its reuse.

– TeenyLIME's one-hop shared tuple space and associated operations are sufficiently powerful to express *multi-hop mechanisms*. In both Mutation Routing and the geographical routing component of object tracking, messages are easily described as tuples. At each hop, these are output to the tuple space of the next-hop node, where a previously-installed reaction delivers the tuple to the routing component. There, the subsequent forwarding to the next-hop node is determined based on the status of neighboring devices, as reflected by the information locally available in the tuple space. As a result, all the routing decisions are encapsulated in the `tupleReady` event handler. This provides an easy and clean way to implement this functionality that cannot be achieved in Hood due to the absence of abstractions to describe the node state.

The considerations above confirm that TeenyLIME's benefits in terms of better design and simpler code hold not only for the development of application logic in sense-and-react scenario, but also for sense-only applications and system-level functionality.

6.2 Evaluating the Middleware Implementation

To verify that the advantages we identified do not negatively affect the system performance, we extend our evaluation beyond the programming model, into TeenyLIME's implementation. Specifically, a middleware layer may impact the *network overhead* and *execution time*, due to the additional processing w.r.t. a plain TinyOS implementation. As a consequence, the *system lifetime* may decrease as well. The latter is key in WSNs, as nodes are usually battery-powered and must operate unattended for long periods.

To investigate the above concerns, we conducted experiments using Avrora [21], an instruction-level emulator for WSNs equipped with a precise energy model. The latter is based on experimental data relative to MICA2 [22] nodes, a widespread hardware platform for WSNs. This approach allows us to gather realistic, fine-grained statistics regarding the energy consumption of arbitrary nesC code. We consider two benchmarks:

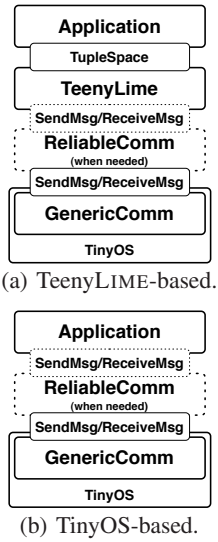
1. The *HVAC* sub-task we illustrated in Section 2, whose TeenyLIME implementation is described in Section 4. We place a variable number of temperature/humidity sensors in the same neighborhood as an air conditioner node. Every 10 seconds, each temperature sensor randomly generates a reading, whose value can deviate from the user preference with a 20% probability. This triggers actuation at the air conditioner controller, which first queries nearby humidity sensors for their most recent reading, and then decides on the specific actions to be taken.
2. A simple application using the token-based, *mutual exclusion* component illustrated in Section 4. A variable number of nodes, in the same neighborhood, express the intention to obtain the token. Every 10 seconds the token is released by the node holding it, and a different, randomly chosen node is selected as the new token holder.

Both applications above involve several TeenyLIME-specific constructs. In the first one, a temperature reading may trigger a remote *reaction* previously installed by the air conditioner, whose pattern contains a dedicated *range field* to express the user preference as a temperature interval. Moreover, humidity values are represented as *capability*

tuples. Therefore, the (unreliable) query coming from the air conditioner triggers the execution of the `reifyCapabilityTuple` event on the humidity sensors. These react by locally outputting the actual tuple[4], which is delivered by TeenyLIME to the air conditioner as the result of the initial query. Similarly, in the mutual exclusion application, releasing the token entails outputting a token tuple in the local tuple space, and possibly triggering some previously installed, remote *reaction*. Nodes receiving this notification then perform a *reliable* **in** operation to obtain the token. Among them, only one succeeds.

The processing above is the same in other scenarios where the data involved have different semantics. For instance, the processing to exchange the token (i.e., a reaction firing followed by a reliable query) is the *same* executed by a water sprinkler in the fire sub-task, shown in Figure 4: only the tuple content changes. In this sense, the meaning of our results extends beyond the benchmark applications we consider here.

For comparison, we consider a plain TinyOS implementation of the same applications. Figure 11 illustrates the component configurations in the two cases. To compare them on common ground when required, we provide TinyOS with reliable communication by using our reliable protocol, mentioned in Section 5.



(a) TeenyLIME-based.

(b) TinyOS-based.

Fig. 11. Component configurations

The emulation settings, in Figure 12, are taken from real MICA2 motes. The larger message size in TeenyLIME is due to the additional control information contained in the tuples. As independent variables, we vary the number of nodes in a neighborhood and the probability ε of losing a message, to investigate TeenyLIME's overhead w.r.t. system scale and network conditions.

Results. In our benchmark applications, TeenyLIME does not generate any increase in the *number* of messages exchanged w.r.t. a TinyOS-based implementation. Therefore, TeenyLIME's overhead in execution time is essentially due to extra *local* processing. In this respect, Figure 13 analyzes the CPU time taken to perform a set of relevant operations in our benchmark applications. The worst case accounts for a 10.08% overhead, which is reasonable given the absolute values involved. We believe these results are due to the generality of TeenyLIME's abstractions. These can capture commonly-used sequences of operations in a natural way, which allows our TeenyLIME implementation to perform close to application-specific mechanisms.

Parameter Name	**Value**
MAC Layer	standard TinyOS MAC for CC1000 chip
Initial Energy Budget	$\approx$ 2 AA batteries
Message Size	47 bytes (TinyOS), 104 bytes (TeenyLIME)

Fig. 12. Emulation parameters

[4] Gathering of physical readings from the sensor device is assumed to be instantaneous.

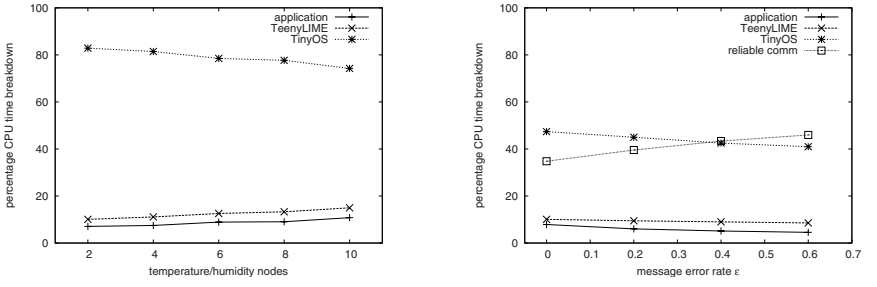
Operation	TeenyLIME	Plain TinyOS	Overhead
Notifying the Air Conditioner	2.18ms	1.99ms	9.54%
Sending a Humidity Query	1.97ms	1.85ms	6.48%
Replying to a Humidity Query	2.25ms	2.03ms	10.08%

(a) HVAC.

Operation	TeenyLIME	Plain TinyOS	Overhead
Releasing the Token	2.03ms	1.97ms	3.04%
Sending a Token Notification	2.28ms	2.07ms	8.21%
Requesting the Token	2.09ms	1.92ms	8.85%

(b) Mutual exclusion.

Fig. 13. Execution times in the components of our benchmark applications



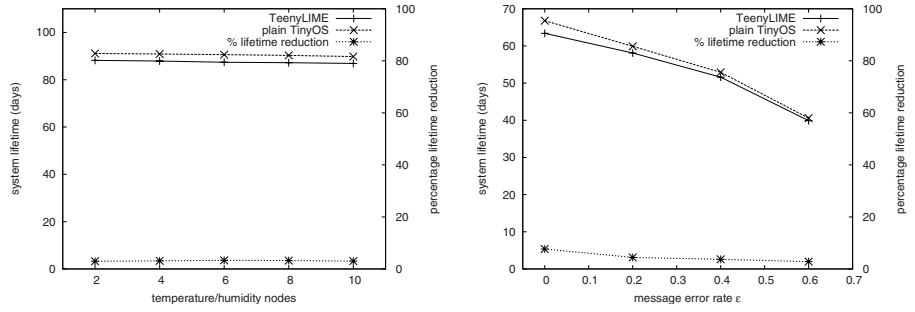
(a) HVAC. (b) Mutual exclusion.

Fig. 14. CPU time breakdown in TeenyLIME-based implementations

Figure 14 further elaborates on the timing aspects in our TeenyLIME implementations, showing the breakdown of CPU time in the different layers. Figure 14(a) illustrates the aforementioned metric for an air conditioner node in the HVAC application, against the number[5] of temperature/humidity nodes in its neighborhood. TinyOS is responsible for most of the processing, as it handles all hardware interrupts and radio-related functions, triggered quite frequently. The trend of the processing dedicated to the application and to TeenyLIME is due to the number of notifications and query replies received at the air conditioner, that grows with the number of nearby nodes. TeenyLIME engages the CPU at most 15% of the time, when 10 nodes are in reach of the air conditioner. The above metric is not directly affected by the message error rate in the HVAC application, as reliability guarantees are not required.

Conversely, when reliability is required it becomes the dominant factor, and system scale bears little effect on our metrics. Figure 14(b) analyzes the CPU time breakdown in the mutual exclusion application against a varying message error rate, with eight nodes in the neighborhood. The chart indeed shows how the reliability protocol increasingly engages the CPU as communication becomes less reliable. In fact, our reliable protocol runs periodic activities (e.g., checking whether messages not yet acknowledged need a retransmission) that take a time proportional to the number of buffered messages. In absolute values, TeenyLIME execution times remain the same regardless of mutable network conditions. Therefore, its relative contribution decreases as the

[5] Half of the nodes in the x-axis are temperature nodes, while the other half are humidity nodes.



(a) HVAC. (b) Mutual exclusion.

Fig. 15. System lifetime

reliable protocol is more stressed. This is a result of our design: TeenyLIME and the reliable communication component are fully decoupled, and the processing implemented in the former is independent from the latter.

It is interesting to look at how TeenyLIME affects the overall system lifetime. Figure 15(a) shows the time until the air conditioner node in the (unreliable) HVAC application runs out of power. This metric is only marginally affected by TeenyLIME, whose additional overhead is always under 4%. The chart also illustrates an almost constant behavior w.r.t the number of temperature/humidity nodes. This is expected: reactions and queries are issued in broadcast by the air conditioner, therefore the energy expenditures for communication are independent of the number of neighbors. Conversely, the number of temperature/humidity sensors affects the local processing, as more neighbors correspond to more replies received. Nevertheless, the extra overhead imposed by TeenyLIME has a very limited impact on the overall lifetime. Along the same lines, Figure 15(b) shows the lifetime in the (reliable) mutual exclusion application, measured as when the last node depletes its battery. The trends here are strongly tied to the message error rate: an increasing number of retransmissions are indeed required as communication becomes less reliable. TeenyLIME's overhead, however, is comparable to the HVAC application, and becomes less relevant as the probability of losing a message increases and, consequently, the reliable protocol is involved more.

Finally, we analyzed our reliable protocol, to verify that our results are not biased by an inefficient implementation. Instead, Figure 16 shows that our solution can provide 100% message delivery with a very small number of retransmissions. This performance is in line with alternative reliability mechanisms in the literature [23], and therefore confirms that our reliable protocol is a valid choice in our evaluation.

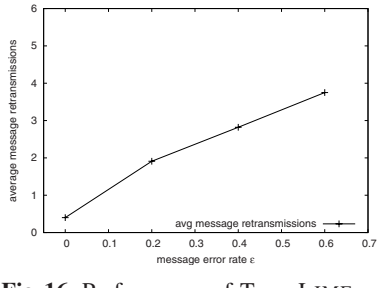


Fig. 16. Performance of TeenyLIME reliable protocol

In conclusion, the trade-offs between the benefits of the programming model and its run-time overhead are reasonable, making TeenyLIME an effective middleware for WSNs.

7 Related Work

TeenyLIME is inspired by LIME [24], which originally introduced the notion of shared tuple spaces in mobile ad hoc networks. However, not only is TeenyLIME's implementation based on entirely different technologies and mechanisms from LIME, but its model and API introduce novel concepts geared expressly towards WSNs, such as range matching, capability tuples, freshness, and explicit control over reliability. TeenyLIME follows in time another adaptation of LIME to WSNs, called TinyLIME [25]. The two, however, profoundly differ in target scenario, model, and implementation. TinyLIME focuses on mobile data collection and employs the standard LIME middleware to provide data sharing over 802.11 among mobile sinks (the data consumers) that, in turn, gather data from nearby WSN sensor nodes (the data producers). Therefore, intelligence is on sinks: the TinyLIME code deployed on sensors is "dumb" and largely application-agnostic, reporting data to external sinks (its only interlocutor) on request. Instead, TeenyLIME is expressly designed for scenarios where the application intelligence is in the network, built around node-to-node interactions inside the WSN.

The work most closely related to TeenyLIME is Hood [6], a neighborhood abstraction where nodes can share state with selected one-hop neighbors. Selection is based on attributes periodically broadcast by neighbor nodes. Neighborhoods are specified using extensions to the basic nesC constructs, precompiled into plain nesC. Therefore, unlike TeenyLIME, in Hood data sharing is decided at compile-time. Moreover, Hood provides neither the ability to affect the state of another node nor the abstractions to *react* to changes in the shared state. This hampers its use in sense-and-react applications, and in general provides a less expressive programming framework.

In Abstract Regions [26] $\langle key, value \rangle$ pairs are shared among nodes in a region (i.e., a set of topologically-related nodes), and manipulated through read/write operations. Again, there is no way to receive notifications when some given data appears in the system, unlike TeenyLIME. Moreover, although nodes in a region may leverage multi-hop communication, this and other aspects must be coded explicitly by the programmer on a per-region basis, therefore hampering generality and applicability.

Context Shadow [27] exploits multiple tuple spaces, each hosting only locally-sensed information representing a given context. Applications retrieve the data of interest by explicitly connecting to one of them. Similarly, the tuple spaces used in Agilla [28] for coordinating among mobile agents are shared only local to a node. Instead, TeenyLIME enables data sharing in a neighborhood by creating the illusion of a single address space. Moreover, these systems lack WSN-specific tuple space constructs.

8 Conclusions

Developing WSN applications is a difficult task, and sense-and-react applications are the most challenging. This paper presented and evaluated TeenyLIME, a middleware

designed for sense-and-react WSN applications, but whose programming constructs are effective in a wide range of applications. TeenyLIME yields simpler, cleaner, and more reusable designs, as we demonstrated quantitatively in non-trivial applications. Moreover, our evaluation with cycle-accurate emulation demonstrated that these benefits are supported by an efficient implementation that introduces low overhead w.r.t. plain-TinyOS implementations. The TeenyLIME middleware is freely available for download at `http://lime.sf.net/teenyLime.html`.

Acknowledgements. The work described in this paper was partially supported by the European Community under the RUNES (IST-004536) and the XtreemOS (IST-033576) projects, and by the Swiss National Science foundation NCCR-MICS (5005-67322).

References

1. Habitat Monitoring on the Great Duck Island, `www.greatisland.net`
2. Deshpande, A., Guestrin, C., Madden, S.: Resource-aware wireless sensor-actuator networks. IEEE Data Engineering 28(1) (2005)
3. Petriu, E., Georganas, N., Petriu, D., Makrakis, D., Groza, V.: Sensor-based information appliances. IEEE Instrumentation and Measurement Mag. 3, 31–35 (2000)
4. Manzie, C., Watson, H.C., Halgamuge, S.K., Lim, K.: 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)
5. Gelernter, D.: Generative communication in Linda. ACM Computing Surveys 7(1) (1985)
6. Whitehouse, K., Sharp, C., Brewer, E., Culler, D.: Hood: A neighborhood abstraction for sensor networks. In: Proc. of 2^{nd} Int. Conf. on Mobile systems, applications, and services (2004)
7. Costa, P., Mottola, L., Murphy, A.L., Picco, G.P.: 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)
8. Estrin, D., Govindan, R., Heidemann, J., Kumar, S.: Next century challenges: scalable coordination in sensor networks. In: MobiCom. Proc. of the 5^{th} Int. Conf. on Mobile computing and networking (1999)
9. Akyildiz, I.F., Kasimoglu, I.H.: Wireless sensor and actor networks: Research challenges. Ad Hoc Networks Journal 2(4), 351–367 (2004)
10. Whitehouse, K., Culler, D.: Calibration as parameter estimation in sensor networks. In: Proc. of the 1^{st} Int. Wkshp. on Wireless sensor networks and applications (2002)
11. Abdelzaher, T., et al.: Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In: ICDCS. Proc. of the 24^{th} Int. Conf. on Distributed Computing Systems (2004)
12. Al-Karaki, J.N., Kamal, A.E.: Routing techiniques in wireless sensor networks: a survey. IEEE Wireless Communications 11(6) (2004)
13. Wan, C.Y., Campbell, A.T., Krishnamurthy, L.: Reliable transport for sensor networks: PSFQ—Pump slowly fetch quickly paradigm. Wireless sensor networks (2004)
14. Rowstron, A.: WCL: A coordination language for geographically distributed agents. World Wide Web Journal 1(3), 167–179 (1998)
15. Gay, D., Levis, P., von Behren, R.: The NesC language: A holistic approach to networked embedded systems. In: Proc. of the ACM Conf. on Programming Language Design and Implementation, ACM Press, New York (2003)

16. Costa, P., Mottola, L., Murphy, A.L., Picco, G.P.: Developing Sensor Network Applications Using the TeenyLIME: Middleware. Technical Report DIT-07-059, University of Trento, Italy (2006), Available at dit.unitn.it/~picco/papers/teenylimeTR.pdf
17. Intanagonwiwat, C., et al.: Directed diffusion for wireless sensor networking. IEEE/ACM Trans. Networking 11(1) (2003)
18. van Dam, T., Langendoen, K.: An adaptive energy-efficient MAC protocol for wireless sensor networks. In: SENSYS. Proc. of the 1^{st} Conf. on Networked Sensor Systems (2003)
19. Rajendran, V., Obraczka, K., Garcia-Luna-Aceves, J.J.: Energy-efficient, collision-free medium access control for wireless sensor networks. Wirel. Netw. 12(1) (2006)
20. Kasten, O., Römer, K.: Beyond event handlers: programming wireless sensors with attributed state machines. In: Proc. of the 4^{th} Symp. on Information processing in sensor networks (2005)
21. Titzer, B., Lee, D., Palsberg, J.: Avrora: scalable sensor network simulation with precise timing. In: Proc. of the 4^{th} Int. Symp. on Information processing in sensor networks (2005)
22. Crossbow Technology Inc., http://www.xbow.com
23. Naik, P., Sivalingam, K.M.: A survey of MAC protocols for sensor networks. Wireless sensor networks, 93–107 (2004)
24. Murphy, A.L., Picco, G.P., Roman, G.C.: LIME: A coordination model and middleware supporting mobility of hosts and agents. ACM Trans. on Software Engineering and Methodology (TOSEM) 15(3), 279–328 (2006)
25. Curino, C., Giani, M., Giorgetta, M., Giusti, A., Murphy, A.L., Picco, G.P.: Mobile data collection in sensor networks: The TinyLime middleware. Elsevier Pervasive and Mobile Computing Journal 4(1), 446–469 (2005)
26. Welsh, M., Mainland, G.: Programming sensor networks using abstract regions. In: Proc. of the 1^{st} Symp. on Networked Systems Design and Implementation (2004)
27. Jonsson, M.: Supporting Context Awareness with the Context Shadow Infrastructure. In: Wkshp. on Affordable Wireless Services and Infrastructure (June 2003)
28. Fok, C.L., Roman, G.C., Lu, C.: Rapid development and flexible deployment of adaptive wireless sensor network applications. In: ICDCS. Proc. of the 25^{th} IEEE Int. Conf. on Distributed Computing Systems, IEEE Computer Society Press, Los Alamitos (2005)