TeenyLIME: Transiently Shared Tuple Space Middleware for Wireless Sensor Networks

Paolo Costa
Politecnico di Milano
Italy
costa@elet.polimi.it

Luca Mottola
Politecnico di Milano
Italy
mottola@elet.polimi.it

Amy L. Murphy
University of Lugano
Switzerland
amy.murphy@unisi.ch

Gian Pietro Picco
Politecnico di Milano
Italy
picco@elet.polimi.it

ABSTRACT

Recent developments in wireless sensor networks (WSNs) are pushing scenarios where application intelligence is no longer relegated to the fringes of the system (i.e., on a data sink running on a powerful node) rather it is distributed within the WSN itself.

To support this scenario, we propose TeenyLIME, a tuple space model and middleware supporting applications where sensing and acting devices themselves drive the network behavior. In other words, the application core is not confined to the powerful sinks, rather it is deployed on the devices embedded within the physical world. Tuple space operations are used both for data collection as well as to effect coordination among sensing and acting devices. This paper describes the TeenyLIME model and corresponding middleware implementation.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]; D.2.11 [**Software Architectures**]

General Terms

Design

Keywords

Wireless Sensor Networks, Tuple Spaces, Middleware

1. INTRODUCTION

Wireless sensor networks (WSNs) are evolving from mainstream architectures characterized by a single sink gathering data and executing the application tasks, to more decentralized architectures where the application intelligence is distributed among the devices. A prominent example of these decentralized architectures is constituted by networks composed of sensors and actuators [2], where the latter base their actions on the data gathered by the former.

Clearly, such a decentralized architecture requires *coordination* among application components running on different devices. Unfortunately, the programming support available to application developers is often very low-level, and forces them to focus on implementation details rather than on the high-level interactions. Appropriate programming abstractions are necessary to deal with the complexity of managing coordination in these dynamic and decentralized WSN environments.

In this paper, we propose TeenyLIME, a new middleware for sensor networks based on the tuple space model made popular by Linda [6]. TeenyLIME leverages off our experience in coordination technology, as it is based on the LIME [12] middleware we developed for mobile ad hoc networks (MANETs), briefly outlined in Section 2. Like LIME, TeenyLIME operates by distributing the tuple space among the devices, transiently sharing the tuple spaces contents as connectivity allows, and introducing reactive operations that fire when data matching a template appears in the tuple space. TeenyLIME is part of a broader vision concerning the application of tuple spaces to mobile and wireless sensor networks [11].

Section 3 discusses the peculiarities of the TeenyLIME model. One significant design decision in TeenyLIME is to restrict transient sharing only to the tuple spaces of one-hop neighbors. This choice is not only energy-conscious, but is also in line with similar research efforts in the field (most notably the Hood system [15]), as we discuss when examining related work in Section 7. Indeed, control of the one-hop neighborhood around a device, augmented with the powerful and expressive primitives provided by TeenyLIME, is versatile enough to enable a number of application-level uses, as we illustrate in Section 6.

A model becomes truly useful to developers only when incarnated in a real programming platform. For this reason, we designed and implemented a version of TeenyLIME for TinyOS [7]. The application programming interface (API), designed for the nesC [4] language, is illustrated in Section 4. In addition to providing the usual Linda primitives and those added by LIME, our API provides features introduced explicitly to address the specific requirements of WSNs. The internal architecture of our middleware implementation is discussed in Section 5.

2. BACKGROUND: LINDA AND LIME

In this section we provide a concise introduction to the notion of tuple space made popular by Linda, and to its adaptation to the mobile environment put forth by LIME.

2.1 Linda and Tuple Spaces

Linda [6] is a shared memory model where the data is represented as elementary data structures called *tuples* and the memory is a multiset of tuples called a *tuple space*. Each tuple is a sequence of typed fields, such as $\langle$"foo", 9, 27.5$\rangle$ and coordination among processes occurs through the writing and reading of tuples. Conceptually all processes have a handle to the tuple space and can add tuples by performing an $\mathbf{out}(t)$ operation and read and remove

MidSens'06, November 27-December 1, 2006 Melbourne, Australia

tuples using $\mathbf{rd}(p)$ and $\mathbf{in}(p)$ which specify a pattern, p, for the desired data. The pattern is a tuple whose fields contain either *actuals* or *formals*. Actuals are values; the fields of the previous tuple are all actuals, while the last two fields of ⟨"foo", ?integer, ?float⟩ are formals. Formals act like "wild cards", and are matched against actuals when selecting a tuple from the tuple space. For instance, the pattern above matches the tuple defined earlier. If multiple tuples match a pattern, the one returned by **in** or **rd** is selected nondeterministically. One typical extension is a pair of primitives, **ing** and **rdg**, used to retrieve all matching tuples.

2.2 LIME: Linda in a Mobile Environment

To support mobility, the LIME [12] model breaks up the Linda tuple space into multiple tuple spaces each permanently attached to a mobile component, and defines rules for the sharing of their contents when components are able to communicate. The union of all the tuple spaces, based on connectivity, yields a dynamically changing *federated tuple space*. Access to the federated tuple space remains very similar to Linda, with each application component issuing Linda operations on its own tuple space. The semantics of the operations, however, is as if they were executed over a single tuple space containing the tuples of all connected components.

Besides transient sharing, LIME adds two new notions to Linda: tuple locations and reactions. Although tuples are accessible to all connected hosts, each exists only at a single point in the system, with one of the mobile components. When a tuple is inserted, it remains in the tuple space of the outputting application component. LIME also allows tuples to be shipped to another host by extending the **out** operation to include a destination. Reactions allow an application to register a code fragment (a listener) to be executed asynchronously whenever a tuple matching a particular pattern is found anywhere in the federated tuple space. This feature is very useful in the highly dynamic mobile environment, as it frees the programmer from the burden of explicitly monitoring the system.

3. THE TEENYLIME MODEL

As previously stated, TeenyLIME targets a scenario in which all WSN components participate in the computation without relying on an external base station. Applications exhibiting such behavior include those where actuators collect information from neighboring sensors and perform some action based on the value returned [2].

This need for coordination among peer devices mimics the coordination supported by LIME, thus it is natural to adopt the transiently shared tuple space as the core abstraction of TeenyLIME. The coordination operations in TeenyLIME are essentially those of LIME, including operations to insert, read, remove, and react-to tuples. TeenyLIME tuple spaces are physically located on the devices themselves but unlike LIME they are shared only with one-hop neighbors. Because each device has a different set of one-hop neighbors, the shared tuple space view is different for each device. This is fundamentally in contrast to LIME in which the view of the transiently shared tuple space is composed of the tuple spaces of *all* connected hosts, and connectivity is assumed transitive.

In general, limiting the scope of operations to one hop is natural for many WSN applications that need access to *nearby* information. For example, a fire extinguisher can make a local decision to activate based only on readings from several sensors in its vicinity, and inform other nearby extinguishers after it activates. Activation can be effected by installing a reaction on neighboring sensors for temperature readings. When sufficiently many high readings are received, the extinguisher should be activated. Notification to the other extinguishers in the area can be handled by outputting a "notification" tuple to their tuple space.

The information about which devices are currently directly reachable is stored in a special tuple space called TeenyLIME system, providing a single unified abstraction for representing both the application and system context. This is similar to the LimeSystem tuple space of LIME that reports which hosts are currently sharing tuple spaces, although the TeenyLIME system supports weaker semantics, as explained in the next section.

4. THE TEENYLIME API

TeenyLIME is implemented on top of TinyOS [7]. Figure 1 shows the nesC interface the application *uses* (in the TinyOS sense) to access the transiently shared tuple space composed of the local tuple space and that of the one-hop neighbors. The first parameter of each nesC command requires a *target*, a specification of the tuple space repositories in the federation over which the operation should execute. Possible values restrict the scope of the operation to the local tuple space (indicated with `TL_LOCAL`), the tuple space hosted by a specific one-hop neighbor (indicated with the address of the device), the union of all tuple spaces hosted by one-hop neighbors (`TL_NEIGHBORHOOD`), or the latter plus the local tuple space (`TL_ANY`).

4.1 Concurrency

In TeenyLIME, all read/write operations are asynchronous, allowing the application to continue while the middleware completes a tuple space operation. This approach blends well with the nesC event-driven concurrency model. Therefore, all read operations are *split-phase* [4]: 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 many.

Analogously, two asynchronous commands are offered to install or remove reactions, taking a pattern as the parameter. When matching data is present, the `tupleReady` event is signaled, returning the tuple that triggered the reaction along with the corresponding operation identifier.

4.2 Reliable Read/Write Operations

Typical WSN applications that focus on collecting sensor data are inherently *state-less*, as the core task is that of communicating sensor readings to a given collection point. Conversely, applications composed of tasks that affect the environment often require *stateful* coordination mechanisms, e.g., using current conditions (state) to act collaboratively. This poses more stringent requirements on the consistency of state, and consequently on the reliability of operations on the tuple space. To address both scenarios, the commands to perform read or write operations can be issued as either *unreliable* or *reliable*, using a flag. The former operations do not provide any guarantee on their successful completion, thus yielding a lightweight form of communication that is suited for state-less applications. Instead, *reliable* operations offer stronger guarantees at the price of higher resource consumption, allowing them to be used for coordination purposes, e.g., to implement control loops based on the representation of the current state of the environment.

4.3 Freshness

Sensor data is inherently time sensitive, a dimension that is even more important when actions must be taken based on the values themselves. For instance, a temperature reading might require dif-

```
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(TLTarget_t target, tuple *pattern);
  command TLOpId_t removeReaction(TLTarget_t target, TLOpId_t operationID);
  // Request to reify a capability tuple
  event result_t reifyCapabilityTuple(tuple* capabilityTuple);
  // Request to provide a NeighborTuple for the local device
  event result_t reifyDeviceInfo();
  // Returning tuples
  event result_t tupleReady(TLOpId_t operationId, tuple *tuples, uint8_t number); }
```

Figure 1: TeenyLIME API.

```
tuple t; TLOpId_t reactionId;
void addTemperatureReaction() {
  t = newTuple(2, // The fields of this pattern
         actualField_uint8_t(TEMPERATURE),
         greaterField(TYPE_UINT16_T, 30);
  reactionId = call TupleSpace.addReaction(
                   FALSE, TL_NEIGHBORHOOD, &t);
  if (reactionId == TL_OP_FAIL)
    dbg(DBG_ERR, "addReaction operation failed");
  else
    pendingReact[pendingReactLength++] = reactionId; }
```

Figure 2: Reacting to temperature values above 30 degrees in the one-hop neighborhood. Here, `TEMPERATURE` is a constant, and `pendingReact` is an application-defined data structure for tracking installed reactions.

ferent responses depending on how long ago it was gathered. To address time in general, TeenyLIME divides time into *epochs* of constant length, and timestamps every outputted tuple with the current epoch value. Two helper functions are offered to the application developers: `etFreshness(pattern,freshness)` and `getFreshness(tuple)`. The first customizes a pattern to impose the additional constraint to match tuples no more than `freshness` epochs old[1]. Conversely, `getFreshness(tuple)` returns the number of epochs that elapsed since the tuple was created.

4.4 Range Matching

While the standard matching semantics of LIME discussed in Section 2 suffice for basic coordination, some WSN applications require additional expressive power. For example, a fire extinguisher application only needs to obtain temperature readings *above* a safety threshold. However, using the standard matching semantics based on exact values, we must issue a reaction over the neighborhood with a pattern matching *any* temperature reading and filter the results when they arrive. This causes unnecessary communication when data is discarded upon arrival, therefore TeenyLIME extends patterns to support range matching. For example, the reaction can be issued stating that the field representing the temperature value must be *greater than* a given value.

Figure 2 illustrates how this is achieved with TeenyLIME. In addition to the usual formal and actual tuple fields, patterns can also contain customized fields whose matching semantics account for (bounded or unbounded) intervals. Figure 2 specifies that a tuple matches the pattern when the first field is equal to the constant

[1]If a pattern does not specify freshness, it matches any tuple regardless of its timestamp.

```
tuple t; TLOpId_t outOpId;
void outputCapabilityTemp() {
  t = newCapabilityTuple(2, // The fields of this tuple
         actualField_uint8_t(TEMPERATURE),
         formalField(TYPE_UINT16_T);
  outOpId = call TupleSpace.out(FALSE, TL_LOCAL, &t);
  if (outOpId == OP_FAIL)
    dbg(DBG_ERR, "out operation failed"); }
```

Figure 3: Outputting a capability tuple for temperature readings.

`TEMPERATURE`, and the second field is an integer value above 30.

This mechanism is easily extensible because the matching semantics is decoupled w.r.t. distribution mechanisms. A developer needing alternate matching semantics must only define two functions: one creating a customized field for patterns, and one defining the conditions for an actual value to match the customized field.

4.5 Capability Tuples

Consider a small modification of the aforementioned scenario, namely a device needing to read a single temperature value from a nearby device. For the pattern to find a match, at least one tuple must be present in the tuple space of the neighboring nodes at the time the $\mathbf{rd}(p)$ operation is issued. Therefore, because a device cannot predict when a **rd** operation will be issued, any sensor that can produce a temperature reading is forced to periodically take fresh readings, and proactively output the values in the local tuple space even if no device is currently interested in them. This clearly constitutes a waste of resources.

To manage this problem, TeenyLIME developers are given the ability to output *capability* tuples, as illustrated in Figure 3, indicating that a device has the capability to produce data of a given pattern. From the point of the view of the application performing a query (e.g., for $\langle$`TEMPERATURE`, ?integer$\rangle$) nothing changes: a matching tuple containing a temperature value is returned by raising a `tupleReady` event. Behind the scenes, however, the processing occurring at the device hosting the capability tuple is different from the normal one. A capability tuple enjoys the same matching semantics as a normal tuple, but is not returned directly as a result. Instead, it essentially works as a placeholder for the real data. A positive match triggers the event `reifyCapabilityTuple`, which reports the matched capability tuple to the application running on the device hosting it. The application can then perform the operation associated with it (e.g., reading the current temperature), build a tuple on the fly, and output it to the tuple space. TeenyLIME takes care of returning the tuple to the querying node as the result

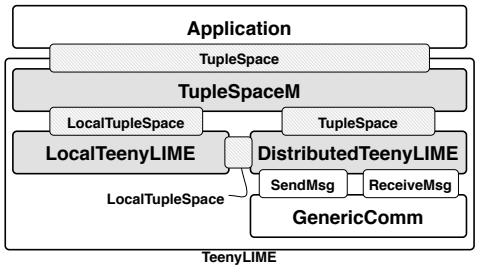

Figure 4: TeenyLIME architecture. (Timer components omitted).

of the read operation.

Our original concept for capability tuples was to act as placeholders for real data, providing a mechanism to reduce the number of sensor readings to those strictly needed to answer queries. However, the concept naturally generalizes yielding a powerful abstraction. Consider that with a capability tuple a device triggers the execution of a *set of operations* on a neighboring device merely by issuing a query matching the pattern of the capability tuple. This set of operations can be more general than simply taking a sensor reading. For example, a programmer could define a capability tuple representing its ability to average some sensor readings taken over a given time period. When a $\mathbf{rd}(p)$ operation is issued with a pattern matching this *average* capability tuple, the corresponding function is triggered and the result returned. Additionally, actual values in the pattern can be used as parameters, further customizing execution, e.g., to specify the desired time period in this example.

To exploit this general capability function feature, the developer simply defines a suitable capability tuple and the function to be called when the corresponding `reifyCapabilityTuple` event is raised. The result of the function is placed in a tuple, stored in the tuple space, and returned to the calling device.

4.6 TeenyLIME System

As previously mentioned, the system context is represented and accessible in TeenyLIME in the same way as application data, i.e., as tuples. In particular, TeenyLIME automatically provides access to the current neighbor set by storing a `NeighborTuple` for each device in range. The content of this tuple is defined by the application developer, allowing customization to include information such as the current location and/or the remaining energy. To keep the information up to date, the system periodically signals a `reifyDeviceInfo` event, which the application should handle by locally outputting a new version of its `NeighborTuple`. If the event is ignored, TeenyLIME keeps the previous tuple[2].

The mechanisms for populating the TeenyLIME System are described next, along with the design underlying the rest of the API.

5. THE TEENYLIME ARCHITECTURE

The overall architecture of TeenyLIME is depicted in Figure 4. It consists of three TinyOS components wired together in the `TeenyLIME` configuration. This is the only nesC component the application must include to access the TeenyLIME system through the aforementioned `TupleSpace` interface. In this configuration, the `TupleSpaceM` component takes care of delegating the actual operation to either the `LocalTeenyLIME` or the `DistributedTeenyLIME` component, depending on the scope of the requested operation. This supports a separation of concerns allowing independent customization of the two orthogonal aspects of TeenyLIME: local storage and distributed processing.

The `LocalTeenyLIME` component stores the tuples output to the local tuple space, and performs the actual matching process for query operations. This component is connected to the rest of TeenyLIME through the `LocalTupleSpace` interface, which mirrors the `TupleSpace` interface without specifying an explicit target for the operation. In addition, `LocalTeenyLIME` takes care of storing reactions installed on the local tuple space, matching reactions against newly arrived tuples, and handling capability tuples. The latter are stored within the `LocalTeenyLIME` component as any other tuple. However, when a request matching such a tuple arrives and the application outputs the real tuple, the capability tuple is not removed from the local tuple space. When further requests arrive for the same capability tuple, the `LocalTeenyLIME` returns the real tuple if it meets the freshness requirement in the pattern. Otherwise, it asks the application again for a real tuple.

The `DistributedTeenyLIME` component is in charge of implementing the operations whose scope is different from the local device. To this end, we define a suitable encoding of operations and parameters into standard TinyOS messages. For unreliable operations, no additional network-level mechanism is required above the basic functionality provided by the TinyOS `GenericComm` module. To implement the $\mathbf{out}(t)$ operation, the tuple is packed in a message and sent to a specific target. Instead, the $\mathbf{rd}(p)$ and $\mathbf{in}(p)$ operations are realized by sending a message containing the pattern. In this case, the `DistributedTeenyLIME` component on the receiving side delegates the matching process to `LocalTeenyLIME`. This returns any matching tuple to `DistributedTeenyLIME` by signalling a `tupleReady` event through the `LocalTupleSpace` interface connecting the two.

To implement reliable operations we are exploring what mechanisms can be borrowed from existing WSN proposals for reliable transport (e.g., [13]). In doing so, we are not limiting ourselves to simply reusing established solutions. The application scenarios we target—composed of localized, distributed interactions—and the specific patterns of query-reply operations of the tuple space abstraction are likely to require specific adaptations and extensions to be used effectively. An extensive study of these mechanisms and possible extensions is currently under way.

Reactions on remote devices are realized within the `DistributedTeenyLIME` component using a *soft-state* approach, i.e., they are periodically refreshed. This mechanism has been used extensively in WSN (e.g., in [8]) and naturally accounts for devices dynamically joining or leaving. Indeed, if some device joins the neighborhood after the `addReaction` command has been issued, it is likely to receive the reaction in a later refresh message. Conversely, if some node leaves, the reactions expire after a timeout.

Finally, the management of the TeenyLIME system, composed of tuples representing the devices in range, is carried out by passively listening for messages from neighboring devices. A message always contains the `NeighborTuple` of the sending device, which is used to populate the TeenyLIME system of the devices overhearing it. At start-up, a device wishing to advertise its presence and also gather information on its neighbors can simply issue a $\mathbf{rdg}$ operation for tuples of type `NeighborTuple`. This avoids periodic broadcasting of identification information, a waste of resources for devices not actively involved in any processing.

6. APPLYING TEENYLIME

Most existing middleware for WSNs target specific use cases, limiting their re-usability. As one of the primary motivations for

[2]The programmer is required at system start-up to provide a tuple identifying the local device, thus if the `reifyDeviceInfo` event is always ignored, the device will still have a `NeighborTuple`.

middleware development is the reliability that comes from reuse, this is a major limitation. TeenyLIME, instead, enjoys wide applicability because the tuple space paradigm, enhanced with reactions, supports a wide range of possible applications. To support this claim, this section outlines several ways of applying TeenyLIME, along with motivating application scenarios.

6.1 Reading Sensor Data

TeenyLIME can be naturally used as a high-level interface to gather sensor readings. The expressiveness given by range matching, together with the energy-aware mechanism of capability tuples, enables flexible, efficient data logging using TeenyLIME.

Sensed data can be locally stored as tuples, and made available for time-aware queries from neighboring devices, using the *freshness* mechanism. Alternately, sensor readings can be used for localized, collaborative tasks such as in sensing/acting applications.

6.2 Multi-hop Communication

Although TeenyLIME enables direct interactions only among neighboring devices, its model and implementation allows the implementation of multi-hop communication in terms of single-hop operations and reactions.

As an example, consider a simplified location-based routing protocol that routes to a given physical location. Thanks to the TeenyLIME system, each node can include its location inside its `NeighborTuple`. Sending a packet is as simple as wrapping the data and the destination location in a tuple, and inserting it into the local tuple space. This triggers a locally-installed reaction to remove the tuple, and query the TeenyLIME system for the neighbor *closest* to the destination. Such a query can be done using a **rd** operation with a pattern matching tuples of type `NeighborTuple`. Once the closest neighbor to the destination is determined, the data tuple is written, using **out**, with the target set to that neighbor. Assuming each device installs a similar local reaction, the tuple will propagate towards the destination[3]. The latter is recognized by checking that no neighbor closer to the destination exists.

6.3 Remote Function Invocation

Capability tuples provide a way to implement rich interaction patterns beyond the simple sense-and-send functionality of traditional WSN applications. They can be used to request some processing on a set of tuples and have a single value returned, as shown by the `average` function in Section 4. Alternately, capability tuples also offer a means to request a node to perform some actions.

In a sensing/acting application, commanding actuator nodes remotely requires the application developer to implement explicitly the (distributed) processing needed to encode in messages the required operation and its parameters, and parse these messages on the receiver side to perform the corresponding operation. TeenyLIME enables the same behavior with a higher level of abstraction. Simply, an actuator node locally outputs a capability tuple for each actuator it is connected to. When this capability tuple is matched by a remote query, the application is notified through the `reifyCapabilityTuple` event, which is interpreted as a request to operate the actuator with the parameters contained in the query pattern. This way, the programmer abstracts away the distribution aspects, and concentrates on the operations.

6.4 Coordination

In general, coordination implies a separation of concerns between application functionality and communication constructs. In TeenyLIME, we can easily support localized coordination through the *data sharing* abstraction and the ability to install *reactions* on neighboring hosts. This can be useful to implement collaborative tasks and decentralized algorithms. For instance, in object tracking applications the nodes currently sensing a moving object must coordinate to elect a leader, that triangulates the position of the tracked entity [1]. This is achieved easily in TeenyLIME: each device registers a reaction on its neighbors for tuples representing a moving object. Whenever a device recognize the presence of such an object, it outputs the corresponding information in its local tuple space. This triggers the aforementioned reaction, notifying the neighbors about the detected object. Under common assumptions for object tracking applications[4] and within an application-defined timeout, the set of devices sensing the target constitutes a clique. Electing a leader from this set is trivial, e.g., selecting the device with the most recent measurement, according to the *freshness* field of the tuple.

6.5 Node and Service Discovery

The tuple space abstraction provided by TeenyLIME enables a simple and yet effective solution for node and service discovery. The set of nodes in range is automatically managed by the TeenyLIME system, and highly customizable using the appropriate `NeighborTuple`. Similarly, the tuple abstraction provides a general way to describe any service a device can offer, e.g., the list of actuators and sensors on board, or the functions available for remote invocation. Node (service) discovery is implemented by performing a **rdg** operation, discovering the nodes (services) available in the neighborhood. Alternately, the reaction mechanism can announce when a node (service) becomes available.

These mechanisms can be effectively employed to deal with the addition of new nodes and the related problem of sensing coverage [10]. Consider a WSN composed of nodes with either seismic or acoustic sensors. When new nodes are added to replace failed ones, they must discover each other and adjust their sensing ranges and periods to guarantee a minimal quality of service. For instance, each point in space must be covered by at least one seismic and one acoustic sensor at all times. An algorithm to perform this distributed adaptation is easily encoded in TeenyLIME. Simply, each node augments its `NeighborTuple` with its position, current schedule, and installed sensors. At start-up, a new node performs a **rdg** for tuples of type `NeighborTuple` to discover nodes and their services, i.e., installed sensors. In doing this, the new device also makes explicit its presence, which is automatically reflected in the TeenyLIME system of the existing devices. A previously installed reaction on the local tuple space of these devices notifies the application of the appearance of a new neighbor. At this point, each device has enough information to reschedule its sensing period and/or adjust the sensing range.

7. RELATED WORK

The work most closely related to TeenyLIME is Hood [15]. In this work, each node has access to a local cache of attributes of interest provided by neighboring nodes. Therefore, data flows proactively according to a many-to-one paradigm. The current implementation considers one-hop neighbors and is based on periodic broadcasting of all node attributes and filtering on the receiver's side. Conversely, TeenyLIME provides a more general programming model encompassing both proactive and reactive operations. In addition, the ability to express different devices as targets en-

[3]We do not solve here the local minima problem of position-based routing. Nonetheless, such mechanisms can be added easily.

[4]Usually, one assumes to have the communication range at least twice the sensing range [1].

ables also the one-to-many and many-to-many communication paradigms. At the network layer, no periodic broadcasting is built into TeenyLIME. Rather, the programmer is given the freedom to explore the trade-offs between reliability and resource consumption, according to the semantics of each operation.

Context Shadow [9] exploits multiple tuple spaces, each hosting only locally sensed information. This way, the system is able to provide contextual information, at the price of increased application complexity. Indeed, the application is required to explicitly connect with the tuple space of interest to retrieve the corresponding information. Conversely, context in TeenyLIME can be managed in a more natural way using the TeenyLIME system, according to the application-defined information provided by each device.

Agilla [5] is a mobile agent platform that uses tuple spaces for local coordination of co-located agents. Agents also interact with remote tuple spaces, which are nonetheless distinct. Conversely, TeenyLIME provides a higher level of abstraction, where data in a neighborhood is transiently shared, and perceived as belonging to a single memory space. Furthermore, capability tuples and the TeenyLIME system give the programmer flexible mechanisms to address specific aspects of WSNs, e.g., energy management.

Abstract Regions [14] proposes a model reminiscent of tuple spaces to enable communication among the nodes in a region (i.e., a set of geographically related nodes). However, the model is fairly limited, as only synchronous read/write operations on $\langle key, value \rangle$ pairs are allowed. 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 rather limited.

Our prior research has explored tuple spaces in both MANET and WSN. TeenyLIME itself is inspired by LIME [12], which introduced *federated tuple spaces* in MANETs. Compared to LIME, TeenyLIME introduces novel features specifically targeted to WSNs, e.g., one hop operations, range matching, and capability tuples. In addition, the system has been completely re-engineered to fit the event-based programming model of TinyOS, in contrast to LIME's Java implementation. Instead, TinyLIME [3] targets sensor networks where mobile PDAs are the main consumers of data from their immediate vicinity. Here, tuple spaces do *not* span the sensor devices, which are instead relegated to the role of data producers. Conversely, TeenyLIME applications are deployed directly on the WSN devices, which play an active distributed coordination role even with no base stations in the system.

8. CONCLUSIONS

In this paper we outlined TeenyLIME, a model and middleware for WSNs. The abstraction available to the TeenyLIME programmer is that of a shared tuple space formed by the union of the data of the local device and the one-hop neighbors. Our future plans include refinement of the implementation to efficiently support reliable operations. Although our initial investigations clearly demonstrate the versatility of TeenyLIME to support a wide range of applications, we also plan to extensively evaluate both qualitatively and quantitatively the advantages brought by TeenyLIME to the application developers, e.g., in terms of code complexity and inter-component dependencies.

Acknowledgements. The authors wish to thank Fabrizio Bonfanti for his contribution to the first implementation of TeenyLIME. The work described in this paper was partially supported by the European Community under the IST-004536 RUNES project, by the European Science Foundation (ESF) under the MINEMA project, and by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

9. REFERENCES

[1] 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.

[2] I. F. Akyildiz and I. H. Kasimoglu. Wireless sensor and actor networks: Research challenges. *Ad Hoc Networks Journal*, 2(4):351–367, October 2004.

[3] 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*, 4(1):446–469, Dec. 2005.

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

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

[6] D. Gelernter. Generative communication in Linda. *ACM Computing Surveys*, 7(1):80–112, January 1985.

[7] J. Hill et al. System architecture directions for network sensors. In *Proc. of the* 9^{th} *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2000.

[8] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A scalable and robust communication paradigm for sensor networks. In *Proc. of the* 6^{th} *Int. Conf. on Mobile Computing and Networks (MobiCom)*, 2000.

[9] M. Jonsson. Supporting context awareness with the context shadow infrastructure. In *Wkshp. on Affordable Wireless Services and Infrastructure*, June 2003.

[10] S. Meguerduchian, F. Koushanfar, M. Potkonjak, and M. B. Srivastava. Coverage problems in wireless ad-hoc sensor networks. In *Proc. of the* 20^{th} *Int. Conf. on Computer Communications (INFOCOM)*, 2001.

[11] 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.

[12] A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: A Coordination Model and Middleware Supporting Mobility of Hosts and Agents. *ACM Transactions on Software Engineering and Methodology*, 2006.

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

[14] 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.

[15] 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.