PicOS Tuples: Easing Event Based Programming in Tiny Pervasive Systems

Benny Shimony, Ioanis Nikolaidis, Pawel Gburzynski, Eleni Stroulia
Department of Computing Science
University of Alberta, Edmonton, Alberta
Canada T6G 2E8
{shimony, yannis, pawel, stroulia}@cs.ualberta.ca

ABSTRACT

The task of programming sensor-based systems comes with severe constraints on the resources, typically memory, CPU power, and energy. The challenge is usually addressed with techniques that result in poor code understandability and maintainability. In this paper, we report on a data centric language extension based on a tuple-space abstraction, akin to Linda [2], applied to PicOS [5], a programming environment for wireless sensor networks (WSN's). The extension improves state and context management in a multi-tasking environment suffering from severe memory limitations. The solution integrates tuple operations into the model – for networking, event handling, and thread contexts. We demonstrate how tuple constructs improve coding and reduce code overhead. We also show how thread's context-tuples can be used as interface arguments for extension by modular aspect constructs.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*data abstraction,domain-specific architectures, languages*; D.1.3 [**Concurrent Programming**]: Distributed programming

General Terms

Design

1. INTRODUCTION

To be practically viable, a typical wireless sensor network must be characterized by a low cost of its components combined with minimalistic energy requirements [7]. These constraints translate into some physical properties of hardware painfully perceptible by the programmer: small amount of memory (RAM), code size limitations (ROM), limited CPU power, and so on, which drastically affect the programming characteristics of building WSN applications.

From the functional point of view, the environment of a WSN node is reactive (rather than computationally-intensive), which is good news. However, it can still be quite complex, *e.g.,* requiring the node to respond in a timely fashion to complicated configurations of events, with multiple events (possibly of different types) occurring at (almost) the same time. This calls for multitasking capabilities of the node's program.

In this setting, the event-based programming paradigm has been the preferred choice of the most popular programming platforms for WSN nodes [7]. This paradigm has been demonstrated to be efficient in terms of energy usage, memory footprint, and concurrent task management. However, these advantages come at a serious detriment to system design qualities, such as code understandability, maintainability, and extensibility [13]. In particular, the tight RAM budget renders classical multi-threading methods impractical, as they tend to squander memory for multiple and largely fragmented stacks. As a result, alternative coding structures are used to handle multi-tasking. Typically, conceptual tasks are broken into *de facto* independent event-handling procedures that share a single (global) stack with no possibility of preserving there a task-specific context. Hacks mitigating this problem are known under the collective name of *manual stack management* (MSM [1]). Common techniques rely on global variables or dynamic heap memory, which are both inefficient in a RAM-tight environment. Moreover, they are bad practice from the viewpoints of encapsulation, modularity, and code readability, since they incur code overhead to co-ordinate the shared data while also being counter-intuitive and prone to errors [13].

To set the stage for our work, we review and compare two programming styles: 1) a typical event-based system akin to TinyOs [12] vs. 2) PicOS [5], a locally-developed, low-overhead operating system for sensor nodes. In both environments, limited memory imposes a non-persistent stack space. TinyOS, the most popular environment for WSN programming today, accommodates a single thread (termed task), which executes at a low priority, while interrupt handlers account for most of the dynamics. In PicOS, a lean mechanism of *cooperative threading* is used to support a finite state machine (FSM) task management abstraction.

Our solution put forward in this paper adopts a data-centric methodology relying on a tuple-space abstraction. That abstraction makes it possible to define named data structures that can be easily handled and shared (a) across multiple local tasks and (b) across a network neighborhood using mirrored tuple spaces. Tasks can coordinate their activities by accessing the (shared) tuples via associative memory operations for retrieving, removing, and adding data. The effectiveness and elegance of this paradigm have been demonstrated in Linda's distributed programs [2], and it adapts well to concurrent task programming, especially in a well-coupled (local) setting. In this paper, we show how the basic idea of Linda's context matching operations fits naturally the reactive environment of WSNs. Moreover, by implementing a local tuple repository with additional support for thread *context tuples*, we eliminate the need for global variables and (explicit) heap space, as detailed in sections 3, 4. Finally, a flexible mechanism for matching and setting the thread *context* and its event triggering conditions plays a central role in extensions/modifications of programs. We view this mechanism as an interaction point for modular *rule* refinement along the line of *aspect* based constructs. The approach is related to *cooperative aspect-oriented programming* (Co-AOP [8]), as will be detailed in sections 5 and 6.

This work makes two important contributions. First, it integrates tuple-space operations into the existing PicOS-threads programming model – by adding language constructs and services to the event-based semantics of the system. Second, it introduces thread *context tuples* as a standard interface to manage thread state context (mitigating limitations on stack use), while providing an interface for modular extensions.

2. BACKGROUND AND EXAMPLE

PicOS [5] provides a lean multi-tasking mechanism derived from an FSM abstraction, around which elaborate rule-driven routing protocols can be developed [4]. In this section, we discuss the PicOS programming paradigm in comparison to TinyOS.

The two coding approaches are exemplified in Figures 1 and 2. The original code has been borrowed from [9]; its goal is to collect temperature samples from neighboring nodes and compare their average to a local temperature reading. Conceptually, the program at the collecting node operates in three phases: 1) sending a request, 2) waiting for replies (temperature readings), and 3) producing the result after a timeout.

2.1 The original code

As listed in Figure 1, the code includes a setup function **init_remote_compare** (lines 4-8) and two event-handling functions **message_hdl** (lines 9-13) and **timeout_hdl** (lines 14-18). In the first phase, the node broadcasts a request message to its neighboring nodes prompting them for reports of their temperature readings (line 6). A report received in response to that request triggers an invocation of **message_hdl**. The function is responsible for aggregating the received data (lines 9-13). When the timeout expires, the handler (**timeout_hdl**) compares the average of the remote temperature readings collected in the meantime to the local temperature (lines 16-18).

Note that variables **num** and **sum**, manipulated by **message_hdl** and **timeout_hdl**, realize a variant of MSM. Since the local stack is unrolled after every action, the local state is not persistent, and the programmer must store the relevant data explicitly: either globally and statically (as in the above example), or dynamically (on the heap). Both solutions are flawed. With the first one, the memory is statically and permanently locked at one task. The latter approach is ridden with various overheads and susceptible to fragmentation, which affects all heap-based allocation techniques and is particularly painful in a non-preemptive (thread-less) environment with tight RAM. This is because such a system must be prepared to deal with a (temporary) unavailability of dynamic memory. That, in turn, assumes that the requesting entity is able to block half-way through its action (no memory) and resume later (on a *memory available* event) as if "nothing happened", which gets us back to the problem of automatic and safe context preservation (for which a per-task stack appears as the most natural solution). The primary reason why TinyOS implements no heap is that its activities cannot block half-way through and thus cannot sensibly wait for memory events.

Another generic way to mitigate the complexity of detailed context preservation, known as *manual flow control* (MFC), is illustrated by the usage of variable **sampling_active**. That variable acts as a coordinating flag between the handlers **message_hdl** and **timeout_hdl** (lines 3, 5, 10, 15). Note that by setting the flag to *false*, **timeout_hdl** disables the other handler (line 10).

2.2 The PicOS version

PicOS defines a flavor of threads. A single thread can describe a number of different event-response actions selected by its dynamic *state*. Following the model of coroutines, the states tag the thread's *entry points* (the **entry** statements in Figure 2), which are the only points where the thread's execution may commence. Cooperative multi-threading takes place as threads explicitly yield the CPU, usually at the end of a state code section, using the **release** command. In order to resume a thread's flow of control, each yield is typically preceded by one or more event wait operations (called *wait requests*). A single wait request specifies an event (that the thread wants to wait for) together with the state to be assumed (entered) by the thread when the event occurs. For example the state INIT in Figure 2 includes two wait requests: one with **when** and the other with **delay** (lines 7-8) operations. With the former, the thread declares that it wants to be resumed in state COLLECT upon the occurrence of an event represented by the address of a message object (buffer). Such events can be signaled by explicit trigger operations (not shown here). The delay operation sets up an alarm clock for the specified number of milliseconds (1000). The event waking the process at state STOP will be triggered when the alarm clock goes off. The action of initiating a new data collection is captured in the invocation of **request_remote_temp** (line 6). While at first sight that statement might stand for something as simple as broadcasting a single *request message* into the neighborhood, we prefer to make the operation more elaborate. In a realistic implementation, the request message will be broadcast several times (at some intervals), and (most conveniently) by a separate thread, in order to maximize the likelihood that all neighbors have perceived the request.

```
int sum = 0;
int num = 0;
bool sampling active=FALSE;
void init_remote_compare() {
   sampling active=TRUE;
   request_remote_temp();
   register_timeout( 1000 );
}
void message_hdl( MSG msg ) {
   if(sampling active==FALSE) return;
   sum = sum + msg.value;
   num++;
}
void timeout_hdl() {
   sampling active=FALSE;
   int val = read_temp();
   int average = sum / num;
   if( average > val ) /* ... */
}
```

Figure 1: TinyOS related code [9]

```
1   int sum = 0;
2   int num = 0;
3   int msg_ev;
4   thread (TempAverage)
5   entry (INIT)
6      request_remote_temp();
7      delay ( 1000, STOP);
8      when (&msg_ev, COLLECT);
9      release;
10  entry (COLLECT)
11     sum = sum + msg.value;
12     num++;
13     when (&msg_ev, COLLECT);
14     snooze (STOP);
15     release;
16  entry (STOP)
17     int val = read_temp();
18     int average = sum / num;
19     if( average > val ) /* ... */
20  end thread;
```

Figure 2: PicOS code

```
<tuple_type> get <tuple_name> ( field_x == value|'*' , ... )
<tuple_type> remove <tuple_name> ( field_x == value|'*' , ... )
List<tuple_type > get_group <tuple_name> (field_x == value|'*' , ... )
List<tuple_type > remove_group <tuple_name> (field_x == value|'*' , ... )
<tuple_type> set <tuple_name> [ field_x = value| ⊥ , ... ]
<tuple_type> tuple_send <tuple_name> [ field_x = value| ⊥ , ... ]

define context{ tuple_name, ... }
when_tuple( tuple_name, next_state, boolean (*filter_func) )
```

Figure 3: PicOS tuple operations

If multiple events are awaited by the thread, the earliest of them will wake it up. Once that happens, all the pending wait requests are erased, an so the thread has to specify them from scratch at every wake-up. For example in Figure 2 when the thread resumes in state COLLECT (after a message event trigger), it preforms aggregation of data in lines 11-12, while in lines 13-14 it re-issues the wait requests[1] before relinquishing the CPU. Finally, once a time-out event occurs, state STOP is assumed, which concludes the protocol cycle. Notice, how the task stages in the PicOS code can be explicitly organized into an FSM form, which obviates the need for state management flags (like **sampling_active** in the traditional variant).

While *flow control management* in PicOS is more convenient compared to the TinyOS model, a PicOS thread yielding the CPU also relinquishes its stack, which forces the programmer to resort to MSM schemes. Hence, the global variables **num** and **sum** have been carried over the previous solution (to be used for essentially the same purpose). Moreover, as the program evolves and its requirements (data and control structures) grow, the overhead and complexity of accommodating the new cases into the existing mess of global variables and flags become more and more difficult to manage. As noted in [12], programming can become tricky when the system needs to be extended, especially when global data requires concurrency control and locking.

[1]Operation **snooze** sets up the alarm clock for the residual time from the previous **delay**.

3. PICOS THREADS AND TUPLES

Figure 3 outlines the tuple interface that we have added to PicOS. For brevity and clarity the functions are presented in a pseudo-code at a high level. The operations are divided into two sections: the top section includes access operations to the tuple space, while the bottom part lists the *local* operations, which are thread-related. The underlying facilitator of the tuple abstraction, is the use of a *local repository*. The repository acts as the focal point of data sharing and communication in the multi-tasking environment – both locally and for the immediate radio-range neighborhood. First it stores new received tuples and manages tuple buffer allocations in the repository. Then, it propagates updates as *tuple events* to the local program, so the new data can be acted upon. In the local setting, the multi-threaded abstraction of PicOS engages the tuple space in the same fashion as Linda [2]. In the network scope, data can be shared through a *send* operation which (unreliably) mirrors/copies the tuples to neighboring nodes. A remotely received tuple is added to the local repository as if a remote **set** operation was preformed. Thus, the underlying tuple framework provides a common interface to handle shared data in the distributed setting.

Operations **get** and **remove** correspond to Linda's **rd** and **in**. They accept templates for matching tuples: each field can either provide a specific (required) value or a wildcard **'*'**. Operation **remove** is similar to **get**: they both retrieve a matching tuple from the repository, with **remove** additionally removing the tuple. If several tuples match the arguments of **get/remove**, one of them is chosen nondeterministically. If no matching tuple is found, the function returns *null*. Operations **get_group** and **remove_group** retrieve sets of matching tuples.

Operation **set** corresponds to Linda's **out**, which inserts a tuple into the repository. Each field can either provide a valid (assigned) value or be undefined '$\perp$' (by default). With **send_tuple**, which is similar to **set**, the program inserts the tuple into the local repository and in addition broadcasts the tuple as a message over the RF channel, which may update (some of) the tuple repositories at neighboring nodes. At present, this operation is asymmetric: the issuing node does not know who will receive the update (there is no guarantee of delivery).

The **define context** construct is part of the thread header. Its role is to define the categories of tuples relevant to the particular thread, which are then deemed to belong to its *context*. As we explain later, context tuples can be automatically assigned by tuple events when waking up threads.

Operation **when_tuple** is a variant of PicOS's generic **wait** request that allows the issuing thread to await a tuple event triggered when a new tuple appears in the local repository. The filter function (provided by the programmer) can describe elaborate qualifying conditions that must be met by a tuple to trigger the event.

Some notable syntactic detail is that in operations **set** and **tuple_send**, we use square brackets [...] to encapsulate associative assignments of data to tuple fields. In the remaining operations from the upper section, the arguments in round parentheses (...) describe field patterns to be matched to retrieved tuples. Individual tuple fields can be accessed through a mathematical projection notation, *e.g., field_name*(*tuple instance*).

4. THE EXAMPLE REVISITED

Figure 4 lists the temperature collection example reprogrammed using PicOS tuples. Note that the comments in lines 1 and 20 annotate for "option **b**", which will be discussed later. Basically, Figure 4 covers two slightly different versions of the code, which have been merged for brevity. We start from the simpler version (option **a**).

Note that the global variables **num** and **sum** have been replaced by a tuple, **aggData**, and the temperature readings are now represented by another tuple, **temperature** (lines 1-2). One more tuple, **reqTemp** (line 3), describes requests sent to neighboring nodes. Such a request includes a sessions identifier (line 9) used to match readings to requests.

The thread includes a context definition (line 5) describing the tuples to be handled by the thread. Such definitions can be viewed as providing placeholders for tuples with some specific layouts, each placeholder occurring in two instances (offering two slots): *current* and *new*. The *current* variant accommodates a local tuple (one residing in the node's local repository), while the *new* part is typically filled by received (retrieved) tuples, possibly triggering tuple events. Similar to the previous version, the action of initiating a new data collection (line 10) may involve multiple invocations of **tuple_send** (from a separate thread).

By calling **when_tuple** (in line 12 and subsequently in line 21) the thread declares that it wants to be resumed in state COLLECT whenever a tuple qualified by **filter_func** appears in the node's view (this may be a tuple received as a message from a neighboring node). The filter function for option **a** (Figure 5) rejects tuples tagged with the wrong session identifiers (i.e., belonging to a different collection). A typical filter function compares (some of) the attributes of the incoming (*new*) tuple to the corresponding attributes of the *current* variant. The collected temperature readings are aggregated in state COLLECT (lines 15-19).

Another point to notice is that the context's *current* instance persists across state invocations; in the present version of the system, the programmer is responsible for the integrity of current context setting as well as the specification of predicates triggering the tuple condition (which wakes up the thread in the respective state). As will be shown in option **b**, such settings can be carried out from within the **filter_func** code.

Operational model: filtering. Tuple events are generally scheduled using a FIFO model. Any tuple appearing at the head of the FIFO queue that is not explicitly waited for (does not match any filter function of a pending **wait_tuple** request) is removed from the queue and ignored (dropped). Formally, such a tuple is not relevant to any of the current contexts and thus not needed by the program. This process becomes a natural (pre)filtering mechanism whose semantics seem to well match a tight-memory implementation (as the program doesn't have to worry about storing outstanding tuples). A more lax scheme is possible whereby tuples are stored for a limited time, such that they can be accessed later (when their contexts come to life). The timeout may be explicit; alternatively, a certain dedicated amount of memory (buffer pool) can be set aside to accommodate the unclaimed tuples. The pool can be recycled as needed to accommodate new tuples, *e.g.,* by discarding those that have remained untouched for the longest time.

An extension: per-node average. At first sight, it may appear that the migration to the tuple-space paradigm adds unnecessary overhead to our otherwise simple application. To see the benefits of this paradigm, consider an additional requirement: we would like to keep track of the average temperature at every reporting node, and derive a total average of averages at the end of the collection process. Notably, this kind of "enhancement" of the original solution (Figure 4) can be achieved quite easily by switching to option **b**, which involves simple changes in lines 1 and 20 and replacing the filter function. With the modification in line 1, we extend the original layout of **aggData** by an **ownerId** field, to be able to associate the data stored in the tuple with a specific node (representing the per-node average). The change in line 20 makes sure that the **set** command references the

```
define tuple aggData [ int sum, int num]  /* option b: [..., nid ownerId] */
define tuple temperature [nid owner, int sessId, int value]
define tuple reqTemp [int sessId]

thread (TempAverage)
define context{ aggData, temperature }
entry (INIT)
    set current.aggData [ sum = 0, num = 0];
    set current.temperature [ owner = nodeId, sessId =1, value = ⊥ ]
    set current.reqTemp [ sessId =1]
    request_remote_temp( current.reqTemp );
    delay ( 1000, STOP);
    when_tuple ( temperature, COLLECT,filter_func(&current, &new ) )
    release;
entry (COLLECT)
    if (owner(new.temperature) != nodeId) {
      int sum_l = sum(current.aggData) + value(new.temperature);
      int num_l = num( current.aggData)++;
      set current.aggData[ sum = sum_l, num = num_l ]  /* option b: [ ..,ownerId = owner(new.temperature)] */
      remove temperature;}
    snooze (STOP);
    when_tuple ( temperature, COLLECT, filter_func(&current, &new ) );
    release;
entry (STOP)
    current.temperature = get temperature( owner == nodeId, * ) //context pattern
    int average = sum(current.aggData) / num(current.aggData);
    if( average > ( value(temperature) ) /* ... */
end thread;
```

Figure 4: PicOS Tuples - Temperature Collection

```
boolean filter_temp (byte *current, byte *new)
{
  if ( sessId(current.temperature)!=
             sessId(new.temperature) )
        return TRUE; // drop tuple event
  else
        return FALSE; // pass event
}
```

Figure 5: a. Simple filter function

```
boolean filter_temp (byte *current, byte *new)
{
  if ( sessId(current.temperature)!=
            sessId(new.temperature) ) {
         return TRUE; // drop tuple event
  }
  current.aggData = get aggData
       ( ownerId==owner(new.temperature), *, *)
  if ( current.aggData == NULL ) {
     current.aggData = set aggData
              [ 0, 0, ownerId = owner(new.temperature)]
  }
  return FALSE;
 }
```

Figure 6: b. Extended filter function

new field in **aggData**. Finally, we have to take care of **aggData**'s context. Note that previously the node dealt with a single tuple of this layout (storing the global variables **sum** and **num**), while now a separate tuple will be needed for every neighbor. As we do not know the identities of all those neighbors in advance, we shall use a new filter function (see Figure 6). In addition to the the previous test, the new function includes a **get** command to query the (local) repository for the context of the reporting node. If it doesn't exist, then we have received the first report from this particular neighbor, and we have to set up a new entry. In any case, by the time the event is triggered (and the thread wakes up in state COLLECT) the context is set properly to the tuple of the reporting node. Despite its utmost simplicity, the example well illustrates the power of the context-setting operation: the modification at state COLLECT is extremely simple and intuitively clear, even though the action carried out at that state is now considerably different from the previous version. One more easy modification (skipped for brevity) needed to make the code complete is in state STOP, where the final calculation must now be based on the per-node averages.

Summary of benefits. The new approach has eliminated the following (displeasing) elements from the original version: ***a)*** the need to explicitly replicate global variables for each context of reporting nodes; ***b)*** conditional control instructions (or indexing) in each state and retrieval of the relevant context or, alternatively, spawning multiple threads for each context (which would be memory-expensive); ***c)*** the need to decode every message in a special thread and coordinate among the multiple threads (via explicit triggers)

As noted in [12], the major difficulty in program extension is preventing race conditions when accessing shared state. In our solution, shared state is managed through the use of a centralized local repository. Section 7 details the implementation, which in essence specializes an existing buffer management mechanism of the PicOS kernel. Moreover, almost no overhead code is needed to translate the new code constructs to basic PicOS code.

5. THREAD CONTEXT ASPECTS

As demonstrated, programs expressed in the tuple-space paradigm can be modified by changing the conditions triggering tuple events with minor, if any, modifications to the "proper" code. Consequently, a good way of integrating useful/popular features into the system must assume that those conditions amount to a powerful programming tool. In particular, it makes sense to view them as modular *rules* and provide for some natural mechanism of applying them. For example, the filtering conditions in some routing protocols for WSN [4] have the form of a chain of rules that are applied sequentially to every received packet, stopping as soon as a reason is found not to rebroadcast it. The general principle of such schemes is to utilize cached data as the knowledge base for the rules. In the absence of data, the rules will *fail* to find reasons for *not* forwarding, so the protocol will operate redundantly (erring on the side of over-eager, but otherwise harmless, collaboration). As more data (knowledge) is collected, the rules will *succeed* more often, thus trimming down the (redundant) retransmissions and improving the quality of routing.

Our data-centric approach boils down to a rather simple filtering process whereby context tuples are passed as parameters to filter functions appended to the basic **wait_tuple** trigger conditions. We suggest that by refining these conditions and organizing them into configurable chains we can relatively easily build a wide range of different, very dynamic, possibly quite complex, distributed algorithms runnable on possibly large sets of resource-constrained communicating nodes. Inspired by *aspects* in the synchronous framework [6], we have found that those aspects that are categorized as *regulative advices* [10], i.e., ones that only restrict the reach space of the base system, can be effectively formed within our system.

We define an *aspect advice* as a set of additional conditions and functions at PicOS state boundaries, which, in our system, can be incorporated as part of the triggering conditions for **when_tuple** operation. Their parameters are exactly the same as the **filter_func** used within the **when_tuple** definition (i.e., **current**, **new**), which gives them the same kind of access to the context data. Needless to say, it is easy to organize them into a chain of rules for natural sequential evaluation. The additional functionality is configurable through the use of a syntactic tool applied in the compilation phase and using a certain template format for describing aspects in the FSM framework (adapted from [6]).

Temperature collection with regulative aspects. Let us revisit once again the temperature collection example and consider a modification whereby the temperature reports are solicited from a specific subset of the neighboring nodes. The selected subset is determined by a *neighbor-group service*, which sets a node group tuple in the repository. The layout of that tuple is:

$$\textbf{tuple}\ \textit{nbr\_group} = (\text{nid}\ \textit{nbr}_1,\ \text{nid}\ \textit{nbr}_2,\ \ldots,\ \text{nid}\ \textit{nbr}_n)$$

We can now describe the requested additional behavior with the aspect template definition *Temp_Group* and its corresponding **filter_advice** function listed in Figure 7 (left and right side correspondingly). The aspect defines point-cuts on transition points expressed by a **wait_tuple** system call (lines 4-5), which covers two join-points from the base program (see lines 12 and 21 in Figure 4). Note the wild card option applied to the state field. If required, we can easily add an option to restrict join-points to specific thread instances and states (or their subsets). The transformation advice (**TRNS_ADVISE** in lines 6-7) adds a conjunct filter function, **filter_advice**, to the event filter function from the original program. With the new function, if the temperature reporting source is not in the *nbr_group* tuple, the event is dropped. Note that the transition advice is actually a "do-nothing" advice, which effectively restricts the reach space of the base program. Thus, we attain the required functionality with no need to change the original code. This functionality can be compared to AspectJ's [11] **around** operation where there is no use of the **proceed** call.

Comments and summary. The regulative aspect type appears to be useful and natural in those cases when the original functionality has to be trimmed down, possibly in a complex manner. While this may be natural in many problems, especially when the base solution has been devised as reasonably general, the regulative type is rather counter-intuitive to most aspect advices (although possible with AspectJ's **around** operation), since it restricts rather than adds functionality. A point to notice is that the default **filter_func** used in the definition of **when_tuple** has a similar role to **filter_advice** functions, except for its additional side-effect of setting the current context (using **current** parameter by a reference). Thus, should the programmer be interested in exercising complete control over state-transitions through advice constructs, she can define a trivial **filter_func** in the base program and override context settings in the consecutive stages of advice. We also plan to include in the filtering advice an implication action (through a $\rightarrow$ operation) added to the aspect template, which will kick in when the advice fails (i.e., the filter passes the data). This implication action can be used to add an independent advice on transitions complementing the restrictive form.

Finally, in order to ensure that aspects do not cause faults in the base program, we note that the advice code needs to adhere to the basic requirements of PicOS threads, i.e., never block within a state and avoid hogging the CPU with extensive computations. This way the CPU never gets locked in a single task and the system appears responsive to all threads. Another responsibility of the aspect programmer is to make sure that the advice code "safely" updates the program's variables, i.e., in a way that does not affect the correctness of the base program.

6. RELATED WORK

The concepts of Linda [2] have been extensively investigated and implemented in various platforms. KLAIM [18] is a calculi based coordination language, that extends the tuple-space abstraction with localities primitives. The tuple-space is split to into multiple localities making it possible to handle data and computation in the distributed and mobile environments. A related approach can be found in LIME [16], which

```
Thread Aspect Temp_Group
VAR: // base program
 context { temperature, aggData}
POINTCUT: // transition type
 when_tuple ( temperature, *, filter_*)
TRNS_ADVISE : // add filter function
   ...and filter_advice()
```

```
boolean filter_advice (byte *current, byte *new)
{
  nbrs =get nbr_group ;
  if ( owner(new.temperature) ∉ nbrs )
     return TRUE; // drop event
  else return FALSE; // do nothing
}
```

Figure 7: Aspect template of temperature collection example.

also adjusts the global tuple space to fragmented transient local tuples. The latter sematics are modeled by state-based logic based on Mobile Unity [17]. Both platforms have been subject to several domain-specific extensions and implementations.

Our approach to network tuple sharing resembles the Hood neighborhood abstraction [15]. In that view, local data can be mirrored to a set of neighborhood nodes through a primitive RF broadcast operation. The mechanism of group sharing corresponds directly to the asymmetric (unreliable) nature of the broadcast medium, i.e., a node is not necessarily aware of the neighbors that mirror its data (whose configuration can dynamically change). In addition, the Hood abstraction provides a data filtering mechanism which is used both for data sharing and neighborhood discovery. This approach can be paralleled to our rule based system of event-filtering. However, the multi-tasking environment of Hood, implemented on top of TinyOS as a set of language components, drastically differs from our scheme. Application design requires components to be glued (or "wired" in the language's terminology) through event interfaces reminiscent of TinyOS's event based programming, and its MFC challenges.

The TeenyLIME framework [3] preserves the semantics of tuple-space operations in the (one-hop) neighborhood. On the one hand, this relieves the programmer from the burden of managing the configuration changes (and manually tracking the neighborhoods, *e.g.,* in the face of node mobility). On the other hand, the framework comes with a hardwired (and rather complex) feature which may be superfluous in many applications. TeenyLIME is implemented on top of TinyOS as middleware that provides an API for Linda operations. The middleware is a monolithic component with a fixed set of underlying algorithms. Our approach is more low-level at its current scope, while sharing many of the motivations of the previous. We believe that our present implementation can be extended to support additional features, as configuration of reliability of tuple spaces within the neighborhood, if such features are indeed called for by the application. In fact, our experience dictates otherwise: a WSN application works at its best, if it never assumes that a sizable group of nodes can operate together as a reliable cohort for any nontrivial amount of time. In this context, a built-in reliable non-local tuple space does not appeal to us as a desirable feature.

Another approach somewhat related to ours transpires in the FACTS language [14], which is both reactive and rule-based. The high-level rules are compiled to run over a middleware layer. While our approach shares with FACTS the basic idea of tuples, our execution model is strongly integrated with PicOS threads and thus de facto becomes a component of the kernel software stack. Even though the high-level flavor of FACTS rules may have some aesthetic appeal, it also renders those rules less flexible and introduces an extra layer of complexity. Based on [14], one can suspect that the middleware rule evaluation engine of FACTS consumes a significant share of RAM. The framework only supports a primitive **send** operation for data sharing. No details regarding the underlying MAC scheme or the reliability of abstractions are provided.

Our modular extensions by regulative aspects overlaps with Co-AOP concepts [8]. This is a general framework for specifying explicit join points (EJP) with abstract interfaces to define interaction scopes of aspect advice with the base code. The authors claim that their approach improves the extensibility of code. Our system provides a fixed scheme of aspect interaction points with the base code, which are inserted at state boundaries as explicit join points. Moreover, context tuples act as exposed arguments for the aspect interaction.

7. IMPLEMENTATION STATUS

PicOS avoids layers (and, consequently, the concept of middleware) while promoting a flavor of plug-ins as the preferred way of incorporating new functionality into the kernel [5]. Plug-ins are *inserted* into a system module dubbed VNETI (Versatile NETwork Interface) depicted in Figure 8. Our tuples environment has been prototyped as a plug-in service taking advantage of the built-in buffer management mechanism of VNETI. Owing to the fact that PicOS threads (in contrast to activities in TinyOS) can comfortably block at state boundaries, PicOS can afford a lightweight dynamic memory allocator, which is used by VNETI to create flexible pools of transparent linked buffers. Those buffers serve as the basis for our repository implementation of the tuple space, both local and distributed (one-hop neighborhoods). The latter is an almost free feature stemming from the fact that selected buffers from VNETI pools can be (automatically) queued for transmission over the RF.

Concepts of modular extensions through aspects are currently at the investigation stage. This work overlaps with our recent efforts aimed at revising and refining the PicOS compiler, which allows us to think of new syntactic tools to be added to the language to facilitate those extensions.

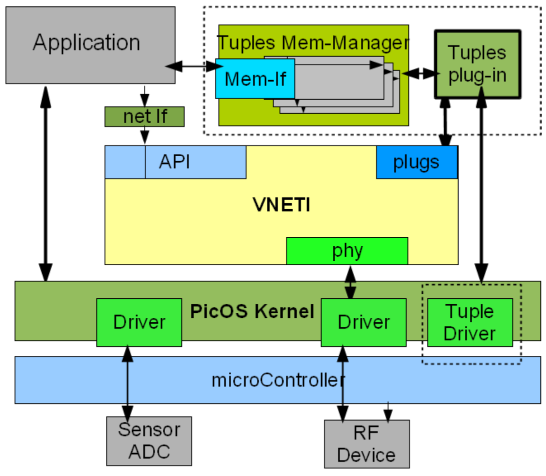

Figure 8: PicOS system architecture layout.

8. CONCLUSIONS

We have presented a tuple based extension to the PicOS threading model aimed at mitigating some of the problems caused by the frugality of the memory-constrained programming environment characteristic of a low-cost WSN node. Our extensions follow up on the elegance, simplicity, and power of Linda's tuple space concept. In contrast to Linda, we cannot get away with a single, global, and reliable tuple space; thus, our goal was not to emulate Linda in the WSN environment, but rather to study the possible ways of adapting the idea of tuples to the new (and capricious) type of distributed systems represented by WSNs. Our preliminary work demonstrates that the idea of tuples combined with thread contexts can significantly simplify programming without worsening the system's characteristic regarding its memory demands. The programs turn out to be considerably more legible and much easier to modify/extend than with the traditional approach to data structures and communication.

Our future work will include a quantitative evaluation of additional protocols and applications to provide assessment of design parameters such as code complexity, code modularity, operational efficiency. We will also focus on additional language extensions and constructs to provide a viable WSN development tool. An important issue is to find a good mechanism for extending the local and neighborhood spaces onto a global network-wide space. Inspired by TARP's approach to routing [4], we envision a fuzzy meta-routing scheme with rule driven rebroadcasts, which will collectively push the tuples towards the regions (neighborhoods) where they are needed. This calls for programming concepts and constructs that will allow the program to define special rules according to suggested distributed configurations/roles thereby mitigating the inherent unreliability of individual nodes.

9. REFERENCES

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proceedings of USENIX'02*, pp. 289–302.

[2] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[3] P. Costa, L. Mottola, Amy L. Murphy, and G. P. Picco. Teenylime: transiently shared tuple space middleware for wireless sensor networks. In *Proceedings of MidSens'06*, pp. 43–58.

[4] P. Gburzyński, B. Kaminska, and W. Olesinski. A tiny and efficient wireless ad-hoc protocol for low-cost sensor networks. In *Proceedings of DATE'07*, pp. 1562–1567.

[5] P. Gburzynski and W. Olesinski. On a practical approach to low-cost ad hoc wireless networking. *Journal of Telecommunications and Information Technology*, 2008(1):29–42, January 2008.

[6] M. Goldman and S. Katz. MAVEN: Modular aspects verification. In *Proceedings of TACAS 2007*, pp. 308–322, Springer, LNCS, volume 4424.

[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, 2000.

[8] K. Hoffman and P. Eugster. Cooperative aspect-oriented programming. *Science of Computer Programming*, 74(5-6):333–354, 2009.

[9] O. Kasten and K. Römer. Beyond event handlers: programming wireless sensors with attributed state machines. In *Proceedings of IPSN'05*, pp. 7.

[10] S. Katz. Aspect categories and classes of temporal properties. In *Transactions on Aspect-Oriented Software Development I*, pages 106–134. Springer, LNCS, volume 3880, 2006.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. pp. 327–353, In *Proceedings of ECOOP'01* Springer, LNCS, volume 2072.

[12] P. Levis. *TinyOS Programming*. Cambridge University Press, 2009.

[13] W. P. McCartney and N. Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *Proceedings of SenSys'06*, pp. 167–180.

[14] K. Terfloth, G. Wittenburg, and J. H. Schiller. FACTS - a rule-based middleware architecture for wireless sensor networks. In *Proceedings of COMSWARE'06*

[15] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of MobiSys'04*, pp. 99–110.

[16] A. L. Murphy, G. P. Picco and G. C. Roman. LIME: A coordination model and middleware supporting mobility of hosts and agents. ACM Trans. Softw. Eng. Methodol, 2006.

[17] P. J. McCann and G. C. Roman. Mobile UNITY Coordination Constructs Applied to Packet Forwarding for Mobile Hosts. In *Proceedings of COORDINATION '97*. pp. 338–354.

[18] R. De Nicola and G. L. Ferrari and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. IEEE Trans. Softw. Eng. pp. 315–330. IEEE Press, volume 24, 1998.