The Evolving Tuples Approach to Application Development

Drew Stovall
Christine Julien

TR-UTEDGE-2008-010
The Evolving Tuples Approach to Application Development

Drew Stovall  
Mobile and Pervasive Computing Group  
Department of Electrical and Computer Engineering  
The University of Texas at Austin  
dstovall@mail.utexas.edu

Christine Julien  
Mobile and Pervasive Computing Group  
Department of Electrical and Computer Engineering  
The University of Texas at Austin  
c.julien@mail.utexas.edu

Abstract—Developing software applications for pervasive networks of resource-constrained devices is a difficult process. The domain-specific challenges are complex enough without the added complexity of unintuitive and convoluted languages required to reify and deploy applications in modern devices. Traditional techniques for pervasive application deployment are also needlessly static and brittle, impeding application maintenance and growth. We posit that easing the low-level development tasks allows developers to focus on high-level issues and encourages the rapid creation of sophisticated and robust applications. Our Evolving Tuples Model seeks to ease the cognitive workload of pervasive application development and encourage more creative, focused attention on domain specific aspects.

I. INTRODUCTION

Since Mark Weiser’s paper introducing ubiquitous computing [1], a whole community of research has been developed to study the coordination and collaboration of devices embedded in our environment. As predicted, sensing and computing devices have already been developed and deployed into a range of environments. In addition, current research is continually developing smaller, better, and longer lasting devices. The rooms, halls, cars, and even the parks where we spend our time will eventually be augmented with a plethora of devices to provide and consume all sorts of information.

However, in pervasive computing environments, the heterogeneity of these devices can be a major impediment to the creation and maintenance of applications. The wide variety of platforms that must be supported will require an enormous number of protocol and application implementations. This will inevitably lead to environments composed of incompatible, incomplete, and proprietary protocols. Additionally, any hardware or software updates that require physical access to devices will require massive effort to perform, making them impractical if not entirely impossible.

Even during development, this variety of hardware and software can become a stumbling block. Small alterations to network protocols or node behaviors can cascade into code changes in a variety of programming languages. The work required to recompile and redeploy new features can slow the rapid development of new applications to a painful crawl. By using the evolving tuples model during development and deployment, developers can make changes to applications, run different protocol versions side-by-side, and add features without redeploing code to the hosts.

In this paper we introduce the evolving tuples model, examine its use in prototyping pervasive computing protocols and applications, and briefly discuss the challenges we encountered developing an implementation. To give the reader a hands-on understanding, we present a stepwise refinement of a simple route discovery protocol into a context-aware route discovery and data aggregation mechanism. This process will show how easy prototyping pervasive computing applications can be using the evolving tuples model.

This paper is organized as follows. Section II introduces tuple spaces and developments to that model from recent literature. Section III details the extensions introduced by the evolving tuples model. Section IV steps through the process that we used to develop a context-aware source-based routing protocol for ad-hoc networks. This section includes a description of each step taken in the process of prototyping the protocol. Some of the challenges encountered in implementing the model are discussed in Section V. We examine related work in Section VI and conclude with Section VII.

II. BACKGROUND

The evolving tuples model is an extension to the large body of research on the use of tuple spaces for ubiquitous and pervasive computing. Our work descends from the Linda [2] system designed for message passing between parallel processes. In Linda, ordered lists of typed data fields are packaged as tuples. These tuples can be passed directly between processes or stored in a tuple space. The tuple space data structure is similar to a multiset, ignoring order and allowing multiple copies of a tuple. The insertion and the removal of tuples from tuple spaces are atomic operations, making the data structures a natural fit for parallel process data exchange. For example, a process monitoring a sensor can package a temperature reading as the tuple ⟨ ⟨string,”temperature”), (int, 78) ⟩ and add it to
a shared tuple space. Another process which uses temperature sensor readings can remove this tuple from the tuple space and extract the relevant data.

In the original tuple/tuple space designs, a process removing a tuple from the tuple space provides a pattern to which candidate tuples are compared. This mechanism allows a tuple space to store both temperature and wind-speed readings (for example) using the self-describing nature of the data to provide type-safety. More specifically, processes provide ordered sequences of formal or actual values which constitute a tuple pattern. A tuple matches a pattern if it has the same number of fields as the pattern and, at each position, has the same type as each formal and the same value as each actual. To continue our example, the process using temperature data would specify a pattern of \((\text{string}, \text{"temperature"}), (\text{int}, ?)\) where \text{temperature} is an actual that must be matched exactly, and \((\text{int}, ?)\) is a formal specifying only the type of the second field of a matching tuple.

While the tuple space design provides a simple mechanism for data passing, it also imposes the restriction that the behavior of data producers and data consumers must be maintained synchronously. Specifically, any alteration to the format or types of the generated tuples will cause them to no longer be matched by the pattern used by existing processes. To support pervasive computing deployments, we must allow tuple formats to change over time since devices are rarely under the same administrative control over their lifetime.

To address this issue, LighTS [3] and ELights [4] decouple tuple fields from their position by assigning unique names to fields. The types and values of a tuple’s fields are accessed using these names instead of fixed positions, allowing fields to appear in any order in the tuple. If a new field is added, or an unused field is removed, consumers of the tuple will be unaffected by the changes. The evolving tuples model adopts this technique to gain the same decoupling between producer and consumer implementations.

These changes allow devices to be maintained by different organizations and to independently alter the specific data format used in tuple based communication. However, the behavior of both the source and target applications must still be maintained synchronously. By embedding some of the behavior that was originally found at the target into the tuple generated by the source, we can shift the maintenance load away from the target and onto the shoulders of the source application developer.

### III. The Evolving Tuples Model

In this section, we describe the evolving tuples model. A more formal specification of many aspects of this model can be found in [5]. The evolving tuples model consists of three major components: the tuple format, the evolution process, and the standard deployment. The tuple format describes how data and behavior are specified while the evolution process describes how the behavior is applied. The standard deployment describes the instantiation of these elements that applications can expect from other nodes in the network.

#### A. Evolving Tuples

As mentioned in the previous section, the addition of a unique name to each field in a tuple decouples the implementations of the tuple producer and consumer from each other. Instead of depending on the exact format of a tuple, applications can depend on the data that the tuple should provide. For example, if one application requires tuples with fields \(A\) and \(B\) while another requires fields \(B\) and \(C\), both applications can correctly identify and consume a tuple with fields \(A\), \(B\), and \(C\). In addition, tuples with an extra field \(D\) can be introduced without affecting either application, and the removal of field \(A\) would only affect the first application.

To ensure unambiguous use of data, we constrain the names of the fields to be unique within a tuple. This allows us to unambiguously reference a field by using only the field’s name. This allows a tuple to be viewed as a look-up table of sorts; given a name, the type and value associated with that name can be returned by inspecting the associated field.

In addition to the name element, the evolving tuple model adds a formula element to each tuple field. The formula associated with a field describes how the field’s value can be automatically updated or evolved. We will discuss the formula element in detail shortly. Combined with the type and value elements of the traditional tuple model, the format of an evolving tuple takes the form:

\[
\langle \text{name}, \text{type}, \text{value}, \text{formula}, \text{name}, \text{type}, \text{value}, \text{formula}, \ldots \rangle
\]

The tuple’s producer specifies the field formulas to manage updates to the values of each field and impart some amount of behavior to the tuple as it passes through a network. Previous to the evolving tuples framework, tuple values were either immutable or altered according to strict pre-existing protocols deployed to network nodes. By moving the logic of value evolution to the tuple, its producer is free to alter his protocols without altering the behaviors already deployed to the network.

A field’s formula is normally a simple expression using traditional arithmetic operators and a few basic functions. Since fields are uniquely indexed by their names, formulas can incorporate the values of peer fields by referencing the field names. Additionally, we allow expressions to access elements of a data structure we call the evolution context. The evolution context serves as a proxy to both static and dynamic values provided by the host in a lookup-table interface. The evolution context provides sensor readings, configuration, and other contextual information. Since this data is indexed by names that are not necessarily unique from the tuple fields, formulas use the prefix “\(\text{ec}\)” to differentiate them.

As mentioned above, the traditional arithmetic operators are supplemented by a few logical functions. Table I outlines the operators that we will be using in the sections below.

#### B. Evolution

Using the contextual values provided by the host through the evolution context, a tuple’s values are updated by evaluating
the formula associated with each field. The formulas that specify new values for each field typically combine existing values with these context values. The new tuple can be viewed as the progeny of the original, the next generation in the evolution of the tuple towards its final configuration.

Consider the following example in which a field in an evolving tuple periodically encounters new evolution contexts. In this example, the tuple maintains a field that contains the maximum temperature value from any evolution context the tuple has encountered. This tuple field may be defined as:

\[
\langle \ldots, (\text{maxTemp}, \text{int}, 0, \text{if}(\text{maxTemp} > \text{EC.temperature}, \text{maxTemp}, \text{EC.temperature})), \ldots \rangle
\]

When this tuple is evolved, the maxTemp field’s value will be assigned the greater of the value provided as temperature in the evolution context (EC.temperature) and the field’s current value (maxTemp). Each time the field is evolved in the presence of a larger temperature value, the field’s value is updated to reflect its “environment.”

In this case, the formula’s dependencies are simple. However, if the formula were to depend on another field in the same tuple, we must be able to specify which of two field’s formulas should be evaluated first. With the addition of names, fields are no longer required to appear in any specific order, eliminating the ability to use the fields’ order to resolve evaluation order. To allow for deterministic evaluation, formulas are evaluated after all the fields that they depend upon have been evolved. A formula is said to depend upon a field if it uses that field’s name. This ordering can be accomplished by selecting fields from the tuple that have satisfied dependencies until no field can be found.

This ordering of field evolution also imposes an additional restriction: formulas must not create circular dependencies. Without this restriction, deterministic evaluation order is not possible. Additionally, any formula that depends on a tuple field or evaluation context entry that is not available cannot be evaluated. However, the value of this field may still be used by other formulas. To allow these dependent formulas to be evaluated, any fields that depend on non-existent values are not updated but are made available to the formulas of other fields.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ , - , * , /</td>
<td>Arithmetic operators</td>
</tr>
<tr>
<td>&lt; , ≤ , &gt; , ≥</td>
<td>Comparison operators</td>
</tr>
<tr>
<td>= , ≠</td>
<td>Logical operators (and, or, not)</td>
</tr>
<tr>
<td>if(x, y, z)</td>
<td>Conditional statement: takes the value y when x is true, the value z otherwise.</td>
</tr>
<tr>
<td>exists(x)</td>
<td>returns the value true if a variable by the name x is present, the value false otherwise</td>
</tr>
<tr>
<td>append(x, y)</td>
<td>appends the value y to the end of the value x. The value has the list type</td>
</tr>
<tr>
<td>newUuid()</td>
<td>returns a new universally unique id</td>
</tr>
</tbody>
</table>

### C. The Deployment Model

Though evolving tuples and the evolution procedure itself can be used directly by an application, the true power of the evolving tuples model is exposed when nodes in a network provide a consistent processing scheme for messages. The model presented in this section is a reference design to conceptually represent how tuples are processed by each node in an evolving tuple network. As with any conceptual model, only the externally observable behaviors are significant, and we expect that various implementations will make appropriate trade-offs internally.

The four components of our reference model include the receive process and three tuple spaces: inbound, outbound, and application. Fig. 1 gives a high-level depiction of the components and how they interact. In the rest of this section we will describe these components in more detail. We first discuss the basics of tuple exchange. Then, because the evolving tuples model is designed for use in pervasive computing networks where broadcast transmissions are common, we discuss some additional features of the model that handle complications introduced by broadcast communication.

1) Basic Tuple Exchange: In an evolving tuples network, messages between nodes are the tuples themselves. Tuples are created and initialized by applications and deposited into the outbound tuple space. To propagate messages to another host, we require the tuple to specify a destination. In the reference model, this value is specified in the destination field. The producing application should initialize the value of this destination field to the address of a neighboring node, or to the broadcast address. The system then selects tuples from the outbound tuple space and transmits them to the node indicated by their destination fields. If this transmission fails, the tuple is redeposited into the outbound tuple space where it can be selected at a later time for another attempt.

When a tuple is received, it is inserted into the inbound tuple space. The evolving tuples deployment model’s Receive process removes tuples and performs tuple evolution on each. After the tuple is evolved, it is ready to be sent to the next node (if evolution generated a new value for the destination field), passed to a local application, or both. If, after evolution, the tuple is destined for this node (either because the destination field contains the host’s address or because the destination field contains the broadcast address), the Receive process deposits a copy of the tuple in the application tuple space. In addition,
the tuple needs to be forwarded (either because the destination field contains a different host’s address or because the destination field contains the broadcast address), the Receive process deposits a copy of the tuple in the outbound tuple space, where the system process takes over as described above.

2) Broadcast and duplicate elimination: A tuple can specify the reserved broadcast address (-1 in this paper) as its destination, thus indicating that the tuple should be delivered to every neighboring node. The evolving tuples deployment model contains a simple broadcast distribution protocol for ensuring delivery of these tuples across the network. While we include a simple protocol, future work will demonstrate how developers can replace this distribution protocol with a tailored one, for example implementing a gossip protocol.

The only significant complexity in our reference broadcast distribution protocol is a mechanism for suppressing duplicate broadcast messages. That is, a node that receives duplicate copies of the same broadcast message should not retransmit it. Because the evolving tuples model equates messages with tuples, this means that an application should not evolve or transmit the same tuple twice. In our model, we accomplish this by assigning each tuple a unique identifier. Our broadcast dissemination protocol then ensures that a node does not retransmit a tuple with an id identical to one received or transmitted previously. Unique tuple ids can be generated in the traditional fashion through a combination of the node’s unique id (e.g., IP or MAC address) and a node-specific counter, though any procedure that generates universally unique identifiers is acceptable.

In the evolving tuples model, the tuple id is another field in the tuple. The implication is that this id field in the tuple can carry a formula that causes the tuple’s id to change, i.e., when the tuple should be replaced by a “new” tuple which, in all respects other than the id, is identical to the old tuple.

IV. DERIVING A ROUTING PROTOCOL

The evolving tuples model described in the previous section provides a novel perspective on coordination in pervasive and mobile environments. This model enables easy expression of dynamically interacting applications by combining a traditional easy-to-use tuple space approach with a message passing model that more closely matches the network interactions that are the backbone of pervasive and mobile computing applications. In this section we expand upon our previous work [6] where we introduced the idea of route discovery with Evolving Tuples. This will demonstrate one practical use of this model: the rapid prototyping of dynamic applications. Specifically, we show the incremental derivation of a routing protocol for pervasive computing, adding functionality to the protocol at each step. This is not a new protocol, rather we intend to demonstrate that the evolving tuples model enables designers to rapidly implement and deploy functionality to validate new behavior in situ.

In this study, we use source routing, a technique that has enjoyed significant success in mobile computing due to its on-demand and autonomous nature [7]. In its simplest form, source routing requires the message sender to include the message’s route through the network by specifying the addresses of the intermediate nodes. This is typically accomplished by embedding an ordered list of network addresses in the message header. Either by cycling through the addresses in this list, or by maintaining an index in the message’s header, a relaying node can determine the next recipient in the path.

Source routing requires a sender to discover a route to the destination before sending any messages. In mobile and pervasive computing networks, this discovery usually takes the form of a simple protocol that floods a route-request packet through the network. As a node forwards the packet, it appends its own address to an accumulating ordered list of addresses. This list represents the path the route-request message has taken. When the target node receives this route-request, it sends a route-reply back to the source. If links are bidirectional, the accumulated path in the request can be reversed and used as the reply’s route. If links are unidirectional, the reply is flooded across the network and, like the request, it accumulates the path for future messages from target to source.

In this section, we start with a simple source routing protocol and demonstrate how it can be quickly and easily prototyped using the evolving tuples model. We then incrementally extend this basic protocol to include additional behaviors that eventually lead to a context-aware source routing protocol. Each derivation of the protocol requires only changes to the routing application’s tuples, without requiring changes to the logic on nodes in the network.

A. Version 1: A Basic Protocol

To implement basic source routing functionality, we introduce an evolving tuple that performs the basic functions of route discovery. By modifying field values to slightly change behavior, the same tuple is used for both the request and reply. Initially, we assume that connections are uni-directional (an assumption that we address in the next version). This requires both the route-request and the route-reply message to be flooded across the network to build the routes in both directions.

The original tuple deposited by the application is shown in Table II (the types of each field have been removed for brevity). This is a variation on the tuple presented in our previous work [6]. In this table and the examples below, we will assume that the initiating application resides on a node with address 0 and is attempting to discover a route to a node with address 2. These values are stored in the source and target fields in the evolving tuple. The onSource and onTarget fields are flags to be used by other formulas in the tuple and signal that the tuple is being evolved upon its return to the source node and the target node respectively. In a final version of the tuple, these flags could be in-lined, but are useful for demonstrating the framework.

1A number of other routing techniques have also been shown to perform well, e.g., AODV [8] and DSDV [9]. We do not promote a particular protocol or style, but have selected source routing to demonstrate our approach.
The route field carries the accumulated route by appending the current node’s address, EC.node, to the end of its current value at each evolution. This includes both the trip from the source to the target, and the return trip\(^2\). The destination field, as described in Section III-C, is used to send the tuple to the next node. Since both the request and reply must be flooded across the network, the destination field is almost always the reserved broadcast address -1. In the case that the tuple is being evolved on the source node itself, we know that the tuple is a route reply and does not need to be forwarded to any other nodes. In this state we set the destination to the source node’s address which prevents it from being deposited into the outbound tuple space.

As mentioned in Section III-C, the broadcast dissemination protocol on a node in an evolving tuples network avoids duplicate transmissions by caching the unique identifiers of the tuples it has recently seen. Since the tuple is being flooded both ways across the network, we must update this id at the target node, allowing it to be transmitted again back to the source. This is accomplished by adding a formula to the tuple’s id field. When the tuple encounters the target node, the id is updated to a new globally unique id.

Fig. 2 gives an example. When the tuple in Table II is deposited in Node 0’s outbound tuple space, it is broadcast to all neighboring nodes (i.e., Node 1). As the tuple moves through this simple network and is evolved, its fields’ values change. The new values are shown in Table III as they would appear after evolution on the node at the top of the column.

We briefly walk through the tuple evolution, using the evolutions from Table III as an example. Since evaluation order is governed by dependency assurance (see Section III-B), even if the tuple fields are arranged in a different order, the evolution is deterministic, and the results would be the same. For reference, the dependency tree for the formulas in our example is shown in Fig. 3. Since formulas can be evaluated once the fields they depend upon are evolved, the order of evolution may be slightly different than described here:

- **source** and **target** have no formula and thus depend on no other fields. These values are retained.
- **onSource** and **onTarget** depend on the **source** and **target** fields and the **address** value in the evolution context. If EC.address is present, then the fields are evaluated. If the local node’s address matches the target or source field values, then the onTarget and onSource fields are set to true (respectively), otherwise the fields are set to false.
- **route** depends only on EC.address (and itself). This formula updates the field value by appending the current node’s address to the ordered list already stored there.
- **destination** depends on the **onSource** field. Once this field is evolved, the destination formula is evaluated. The field is usually set to -1 (the broadcast address) with the exception of when onSource is true. In this case we set the destination field to the current node’s address preventing it from propagating any further.
- **id** depends only on the onTarget field. When the tuple is being evolved on the target node, it needs to chose a new id to allow it to be re-flooded across the network back to the source. Here the formula calls upon the newUuid() function to generate a new globally unique identifier.

### B. Version 2: Bidirectional Links

The first derivation of our prototype protocol targets networks in which unidirectional links are present. When links are bidirectional, the target node can eliminate the use of flooding for the route-reply message by inverting the discovered path in the route-request message. As in the previous version, the route field will store a history of nodes visited. However, in this version of the tuple, there is no need to store addresses in the return trip. To make this determination easier, we introduce the isReply field to determine in which phase of route discovery the tuple is currently operating.
When the tuple is operating as a reply message, we use the addresses in the `route` field to direct the tuple back to the source. Since we are no longer broadcasting, the `destination` field must be updated on each evolution to be the address of the next node in the route. The `destination` field formula in this tuple has three cases. As in the previous version, if the tuple has returned to the source, the destination is set to the source’s address. In the case that the tuple is in the request phase (still searching for the target), the destination is set to the broadcast address (-1). In the final case, we select the next address from the previously assembled route using the value of the `routeIndex` field, which reverses the route that was received at the target. This is done by setting the index correctly at the target, and then decrementing it after each hop along the route, or more specifically, on each evolution. If the tuple is operating as a route request, the value is unimportant, so we set it to -1. These fields and formulas are shown in Table IV.

In Table V we show how the values for these fields evolve as the tuple passes through the example network shown in Fig. 2. For brevity, we show only the fields that have been changed for this version of the protocol. Note that in this version, the `route` list stops growing after the tuple reaches the target node. The `destination` field only uses the broadcast address (-1) while finding the route, and we do not bother updating the `id` field.

C. Version 3: Context-Based Discovery

The first two versions of our mobile communication protocol assume that a source node knows the network address of the destination. In mobile ad hoc networks (especially those that include nodes from different administrative domains) applications may not have the benefit of knowing each other’s addresses *a priori*. In the absence of this pre-shared information, applications can use centralized or distributed service directories to locate nodes that host specific information or services [10]. Another approach to service discovery is to use context-based information at the routing layer [11].

Since it is less complex, the remaining derivations build on Version 1 of our protocol described in section IV-A. In this version, our tuple searches for the target based not on its address but on a description of a desired resource. In the evolving tuples model, the resources a node can provide are represented as part of the context that node provides. In evolving tuples language, this means descriptions of these resources are stored in the evolution context. As our route-request tuple travels through the network, the task of determining whether it has found a compatible target must now examine this context for a resource matching the one desired.

The evolving tuple that accomplishes this is shown in Table VI. We replace the formula for the `onTarget` field to examine context values rather then the node’s address. We also set the initial value of the `target` field to null (Ø) to reflect that the tuple has no concrete target. The formula for `target` is also changed to update this initial value to the network address of the current node when the `onTarget` field is true. Since we are still assuming the presence of unidirectional links, the address of this node is required to split the total route into out-bound and in-bound segments. The values of these fields as they pass through our sample network are shown in Table VII. In this example, the route-request tuple is searching for a node whose temperature value greater is than a specified threshold (65).

D. Version 4: Context Collection

Our derivations so far have generated a protocol that can find and return network routes using either unique addresses or context information to designate the route’s endpoints. Using elements from these protocol approaches, we can now use the evolving tuples model to derive additional protocols with more complex behavior. For example, applications designed for ad hoc sensor networks are commonly interested in not only the data stored at the ultimate destination but also in data from intermediate nodes. This is often addressed using protocol schemes that aggregate data along paths in a network [12].
Such approaches have been shown to have significant performance benefits when compared to contacting each node on the route individually and then locally aggregating the results.

Along these lines, our next protocol derivation adds the ability to aggregate contextual information encountered as a tuple traverses the network. In this particular example we collect enough information to determine the average temperature of the nodes along the route. Similar approaches could be used to collect other common aggregates (e.g., minimum, maximum, count, etc.). In addition, the expressiveness of the evolving tuples formula language allows the tuples to compute more complex, application-defined aggregates as well [13], [14].

In the tuple shown in Table VIII, the sum of all the temperatures encountered is maintained as the value of the $totalTemp$ field, and the number of temperature readings is maintained in the $numTemps$ field. In this example, the final calculation ($totalTemp/numTemps$) is left to the application receiving the returned tuple.

The values of our two new fields as the tuple propagates through the network are shown in Table IX. Variations of this tuple could be created to maintain separate averages for the route-discovery and route-reply phases if the application required by using the existing $isReply$ field as a guard.

While this is just a simple example of how an evolving tuple can be used to provide in-network context collection and aggregation, it demonstrates the power of the evolving tuples model. By using the evolving tuples model to prototype such sophisticated communication schemes, protocol developers can use and evaluate various context values quickly and effectively without requiring a recompilation or redeployment process. It also allows the developers to deploy and evaluate the protocols on test-bed networks in advance of creating a full-fledged low-level implementation.

E. Version 5: Context-Based Flooding Optimization

In typical implementations of flooding based protocols, we often find some sort of mechanism to limit the scope of messages to prevent the consumption of valuable resources far from the area of interest. Normally, these mechanisms are based on the number of network transmissions that the message has been passed along (i.e., hop-count or TTL). Other metrics can also be used to limit the distribution of flooded messages, for example nodes that are aware of their physical location can incorporate geographical boundaries.

Using evolving tuples, we can not only implement these simple techniques, but we can combine a variety of contextual data to precisely limit flooded messages to regions of interest on a per-message basis. We present a simple example in Table X, but much more complex variations are clearly possible. The approach here is to guard the previous tuple formula for the $destination$ field with a check to the $inFloodRegion$ field. This field is initially set to true, but is set to false if the tuple encounters a node where the temperature is below 30.

Taken together with the previous versions, our tuple now implements source-based route discovery for networks with unidirectional links to find a node whose temperature is greater than 65 without traversing any links to nodes with temperatures less than 30. Additionally, the route’s average temperature is easily calculated from the values collected as the message traversed the network.

V. IMPLEMENTATION

To validate the model and the prototyping examples described above, the evolving tuples model has been implemented for the Sun SPOT\textsuperscript{3}[15] platform. The Sun SPOT project combines the Squawk Virtual Machine [16] with wireless sensor hardware to create an ideal platform for prototyping new applications. For reference, Fig. 4 shows one of our Sun SPOT devices. In this section we will discuss the challenges that we encountered in this implementation and a qualitative analysis of the resulting system.

A. Challenges to Implementation

As if to prove the point, the first challenges in implementation of the Evolving Tuples Model were associated with

\begin{table}[h]
\centering
\caption{Route Request/Reply Tuple, Version 4}
\begin{tabular}{|c|c|c|}
\hline
Name & Value & Formula \\
\hline
numTemps & 0 & if (exists(EC.temperature), numTemps+1, numTemps) \\
\hline
totalTemp & 0 & totalTemp + EC.temperature \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\caption{Route Request/Reply Tuple, Version 5}
\begin{tabular}{|c|c|c|}
\hline
Name & Value & Formula \\
\hline
inFloodRegion & true & EC.temperature > 30 \\
\hline
destination & -4 & if (!inFloodRegion, old destination formula) \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\caption{Version 4 Values}
\begin{tabular}{|c|c|c|c|c|c|}
\hline
Name & Node 0 & Node 1 & Node 2 & Node 1 & Node 0 \\
\hline
numTemps & 0 & 1 & 2 & 3 & 4 \\
\hline
totalTemp & 0 & 60 & 130 & 180 & 230 \\
\hline
\end{tabular}
\end{table}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig4.png}
\caption{Sun SPOT Device}
\end{figure}

\textsuperscript{3}Small Programmable Object Technology
Building and deploying a working program for the device. This process requires downloading and installing development tools from the vendor, updating firmware, and a variety of configuration changes to support the new device. Additional work was required to integrate the tools into a third party development environment which was familiar to the authors. These challenges proved to be major stumbling blocks that significantly delayed the project.

The most difficult technical challenge proved to be the parsing and evaluation of mathematical expressions. To reduce errors and effort, our final solution was to port a pre-existing open-source library\(^4\) to provide most of this functionality. The porting process resulted in removal of many features due to the limited functionality available in the Squawk JVM, but the core functionality was maintained.

The most difficult engineering challenge came from developing a fast and robust unit test framework. While they provided some support for unit testing, the vendor tools assumed that the unit tests would be run on the Sun SPOT devices themselves. This assumption forces the user to deploy his unit tests to the device each time they are to be run, a process that consumes several minutes. In our experience, we have found that it is critical that unit test execution should be quick, so we developed support for unit test execution on the desktop. While this was not the most difficult challenge, this feature was clearly provided the biggest boost to productivity.

Once these engineering and development obstacles were overcome, the Evolving Tuples implementation was used to validate the protocol examples that are given in Section IV. As expected, the tuples discovered routes and aggregated data properly. The implementation has also uncovered a few optimizations that may be included in the Evolving Tuples Model in the future, but are more likely to be implementation choices.

\(B. \textit{Benefits of Rapid Prototyping}\)

While a full empirical evaluation is left for future work, in this paper, we have constructed a number of mobile computing protocol derivations, each of which turned out to be a straightforward application of the evolving tuples model. In this section, we specifically look at the context-based discovery version of our protocol because we have experience creating both the complete, deployed version of this protocol and the evolving tuples rapid prototype. The complete version of the discovery protocol is part of the Cross-Layer Discovery and Routing (CDR) protocol\([17]\). Just the route discovery and reply portion of this protocol comprise more than 900 lines of uncommented C++ code. While counting the lines of code is not necessarily the best measure of an implementation’s complexity, in the evolving tuples model, this same functionality can be implemented by defining exactly the eight-field tuple described in Section IV-C.

This radical decrease in the amount of code needed to implement the same behavior is due to the fact that the evolving tuples model provides several constructs that directly assist in the definition of mobile communication protocols. First, the evolving tuples model builds in a message passing framework that is natural for prototyping mobile and pervasive communication protocols. Therefore, in the evolving tuples context-based discovery prototype this complexity is handed off to the underlying evolving tuples system and can be ignored by the protocol developer.

\(VI. \textit{Related Work}\)

Early tuple space designs\([2]\) and implementations\([18]\) for Linda targeted parallel processing environments. Specifically, the atomic insertion and removal operations on tuple spaces relied on locks provided by shared memory. The LIME\([19]\) system introduced distributed tuple spaces that provided the same atomicity guarantees across a truly global tuple space spanning many devices in a mobile network. This adaptation of tuple spaces allows for a very abstract representation of the network underlying a pervasive computing application, but requires that tuples be delivered to consumer processes without interacting with the “lower levels” of the network. We believe that exposing cross-layer information to tuples in our approach allows for more powerful applications at the cost of a slightly more complex representation.

In a similar approach, mobile agent systems also combine behavior with the data that traverses the network. Often these implementations choose to extend the tuple model as its minimal nature makes it particularly well suited for encoding for transmission. However, in an effort to provide a wide range of functionality, undue burden is placed on either the developer or the hosts. Systems like Agilla\([20]\) require the developer to understand very low-level programming languages, while systems like TOTA\([21]\),\([22]\) and MARS\([23]\) require hosts to support high-level languages (i.e., Java). We feel that the evolving tuples model strikes a balance between the skills required to use the system and the capabilities that are required of the network hosts.

Another similar technology is active networking\([24]\), in particular capsule-based systems\([25]\). The biggest difference with the evolving tuples model is the target environment. Active networking targets routers in wired networks to enhance performance and offer new services while evolving tuples are intended for resource-poor nodes in (typically) wireless networks. Additionally, the active networking community has struggled with the security trade-offs associated with allowing mobile code to modify network elements. Since the evolving tuples model only allows formulas to modify the tuple itself, the security issues are reduced significantly.

While rooted in different technologies, there are a number of designs to reduce the efforts required to develop pervasive computing applications. For example, Weis et al.\([26]\) use visual programming techniques to reduce the learning curve typically required. Other approaches such as\([27]\) and\([28]\) provide additional layers of abstraction to manage complexities that can be hidden from the developer. Still other approaches tackle the problems of recompilation and redeployment head

\(^4\)\textit{JEP-Java Math Expression Parser, version 2.4.1, http://sourceforge.net/projects/jep/}
on. For example Dyer et al. [29] deploy nodes in pairs to allow both access to running applications, but also allow new applications to be more easily deployed. We anticipate that these techniques to be complementary to the evolving tuples model, and might be combined to further ease software development for pervasive computing.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we have presented the evolving tuples model and shown how it can be used to easily prototype a complex context-aware protocol. The step-by-step development of the protocol walked the reader through a variety of issues common to this type of work and directly explained how they are addressed by elements of the model. While each development increment used simple examples, the flexibility of the model and the range of potential extensions were discussed. Furthermore, each of the steps in the routing protocol development was validated using a implementation of the Evolving Tuples Model on sensor network devices.

We have limited our presentation to focus on the use of the Evolving Tuples Model to rapidly prototype protocols before their full implementation. However, we believe that it is possible that evolving tuples may be an acceptable deployment framework for deployments that require significant degrees of flexibility and reprogrammability. In such environments the size of the messages can become prohibitive. However, resource discovery using descriptive requests has been incorporated into network communication protocols, and it has been demonstrated that the cost of sending larger (semantically rich) messages can be low enough to reap benefits in comparison to not using semantically rich requests [17]. Other approaches have explored mitigating these effects using innovative binary representations, for example to reduce the size of XML-style descriptions [30]. Ultimately, our model has the potential to make developing pervasive computing applications more approachable, exciting, and fruitful.

ACKNOWLEDGMENTS

The authors would like to thank the Center for Excellence in Distributed Global Environments for providing research facilities and the collaborative environment. This research was funded in part by NSF Grant #CNS-0626777 and AFOSR Grant #FA9550-07-1-0157. The views and conclusions herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

REFERENCES