Resource Discovery with Evolving Tuples

Drew Stovall
Christine Julien

TR-UTEDGE-2007-006

© Copyright 2007
The University of Texas at Austin

Resource Discovery with Evolving Tuples

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

ABSTRACT

Pervasive computing environments present new challenges that hinder traditional approaches to software engineering. In this paper, we tackle one such challenge: the need of pervasive application developers to have access to constructs that enable resource discovery in dynamic environments. We first define the *evolving tuples* model, a novel extension to traditional tuple spaces that allows applications to embed context-aware adaptation directly in structures traditionally used for distributed coordination. The behavior applications embed in evolving tuples can subsequently be used by discovery queries to allow environmental characteristics to directly impact the results of discovery. Our approach minimizes the amount of infrastructure that must be available to support discovery by embedding the discovery functionality almost exclusively within the tuple. At the same time, our model retains many of the benefits of traditional tuple space approaches, namely providing content-based coordination that is easy for developers to understand and implement. The evolving tuple's inherent flexibility also allows resource discovery to adapt to the changing resources in a pervasive environment.

Keywords

tuples, evolving tuples, pervasive computing,
resource discovery

1. INTRODUCTION

The future of computing relies upon embedding computing functionality into our everyday environments. This defines the domain of pervasive computing, in which immersed users must rely on the distribution of functionality in the environment to support computing tasks. With the miniaturization of technologies and the improvements of wireless communication, pervasive computing applications are now realizable in several domains, including support for first responders [18, 26], intelligent construction sites [11, 25], aware homes [16], and many others. New characteristics of applications, such as the extreme heterogeneity of devices, unpredictability of connectivity between users and embedded functionality, the increasing scale and distribution of the network, and the need for local information contribute to significant complexity in designing, programming, and deploying pervasive computing applications.

As demonstrated by existing attempts to support pervasive computing application development [2, 6, 10, 17], one of the most important aspects that affects the quality of pervasive application support is the availability of constructs that support the discovery of and connection to resources embedded in the environment. Our experience in creating a middleware to support pervasive application development [12, 14] has demonstrated that resource discovery for pervasive environments possesses several requirements on top of the demands from traditional peer-to-peer applications. These requirements are largely tied to the characteristics of these emerging environments as overviewed above. Resource discovery in pervasive computing environments must be:

- *autonomous*: applications should be able to independently perform discovery without global coordination or even global information;
- *open*: devices are not under the control of a single administrative organization, and therefore resource discovery mechanisms must allow open interactions;
- *fully distributed*: discovery must not rely on centralized infrastructure to locate nearby useful resources;
- *localized*: because discovery should be fully distributed, it must be scoped (e.g., by physical or network distance) to prevent flooding the entire network;
- *best effort*: discovery should prevent unreasonable or exhaustive searches in lieu of strong fairness or completeness guarantees;
- *context-aware*: discovery should be influenced by environmental, network, and application information, both in the network as discoveries propagate, and at the end-points where discoveries are resolved; and
- *forward compatible*: as device capabilities and functions change (e.g., descriptions are extended to include more information) discovery queries that worked previously should continue to function.

The above stated requirements of resource discovery in pervasive computing applications intertwine specification requirements and the implementation of those specifications

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE 2007 Dubrovnik, Croatia
Copyright 2007 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

using communication protocols. This promotes awareness of network and environmental concerns at the application level (i.e., context-awareness) and awareness of application requirements at the network level (i.e., application awareness). This has the potential to complicate the interfaces between the application and communication capabilities. Support for resource discovery in pervasive computing, however, should mitigate this complexity through a specification mechanism that is implicitly coupled with the discovery implementation and allows the application maximum flexibility and expressiveness without presenting undue complexity.

To achieve the above requirements for resource discovery in pervasive computing, we define a new resource discovery mechanism using *tuple spaces* to coordinate distributed processes. The tuple space model originally introduced by Linda [9], which enables content-based access to data, has recently received significant attention in coordinating mobile and pervasive applications. Much research has focused on using tuple space derivatives to enable distribution of application data among loosely connected nodes in mobile ad hoc networks [4, 7, 13, 20, 21]. In general, the use of these systems has shown that the tuple space provides a natural abstraction that separates the application developer from explicit representations of the communication required to enable the desired coordination.

In this paper we briefly introduce a novel extension to more traditional tuple space models that allows tuples to *evolve*. This paper provides a brief introduction to the resulting *evolving tuples* model. We then focus extensively on how this model can be used to implement resource discovery in pervasive computing environments. Our resource discovery approach leverages the dynamic features provided by evolving tuples to deliver tailored functionality based on both network- and host-centric characteristics.

The novel contributions of this paper lie in two primary areas. We provide the first introduction to the *evolving tuples* model, a new application of tuples that focuses on the ability of tuples themselves to be context-aware and implement adaptive functionality. We also introduce a new approach to resource discovery in pervasive computing environments. This discovery mechanism is the first of its kind in that it is tailored to the requirements of pervasive computing environments. This general technique for resource discovery can be incorporated into existing frameworks for resource provision in pervasive computing applications (e.g., [14]).

This paper is organized as follows. Section 2 briefly describes the *evolving tuples* model which is applied to pervasive computing networks in Section 3. Section 4 then builds a resource discovery protocol for pervasive computing environments that uses this evolving tuples model. This is followed by a comparison to related work in Section 5. Section 6 concludes.

2. THE EVOLVING TUPLES MODEL

This research is founded on previous work that has demonstrated that the tuple space is a useful abstraction with respect to simplifying programming in complex distributed environments. In comparison to traditional distributed applications, however, pervasive computing applications require significant amounts of context-awareness and adaptivity. In this paper, we introduce the evolving tuples model which allows application developers to embed this context-awareness directly in the same tuples that are used for data coordination. This provides the expressiveness and flexibility applications require without adding undue complexity.

This section presents the evolving tuples model, which serves as the foundation for defining the communication and resource discovery mechanisms described in the subsequent sections. Evolving tuples are passed between connected nodes to gather and distribute information across the network. In the remainder of this section we detail the internals of these evolving tuples. We start by overviewing existing tuple approaches on which our model is based, then explore the modifications our model uses to provide context-aware behavior to pervasive applications.

2.1 Background

In Linda [9], data elements with very little structure are used to communicate between parallel processes through a shared *tuple space*. These data structures, or *tuples*, are simply sequences of fields. An application might use the tuple ⟨“ping”, 10, 5, 3⟩ to represent a request to ping another node on a network. Given an *a priori* data format specification, another application could decode the first field as the message type, the second as the destination node, the third as the source node, and the fourth as the “time to live” (ttl).

When a process wishes to coordinate with another process, it inserts a tuple into a tuple space using the non-blocking `out(`*tuple*`)` operation. A receiving process can extract a tuple using the `in(`*template*`)` operation, where the *template* argument is a pattern that may contain a mixture of both concrete values (actuals) and value types (formals) as in ⟨“ping”, 10, ? integer, ? integer⟩. In this tuple, the first two fields are actuals and the latter two are formals.

When the `in` operation is performed on a Linda tuple space, tuples are compared with the template for exact equality to actuals and type equality to formals. If multiple tuples are found to match the template, a single tuple is selected non-deterministically and returned to the requesting process. If no tuples are found, the operation blocks until a matching tuple is inserted into the tuple space. In either case, the returned tuple is removed from the tuple space. The operation `read(`*template*`)` performs the same function as `in`, but does not remove the tuple from the tuple space.

Recent research has added more expressive operations to Linda's original suite. The non-blocking `inp` and `readp` are commonly used to ‘probe’ a tuple space for the existence of matching tuples, without suspending the calling process. The group operations `ing` and `readg` [24] are also commonly added to allow all matching tuples to be returned to the client in a single call.

ELights [13] builds upon the LighTS [3] implementation to decouple the fields of a tuple from their positions by adding an identifying *name* to each field, as in ⟨*msg_type*=“ping”, *destination*=10, *source*=5, *ttl*=3⟩. This addition allows applications to differentiate tuples that use the same value types to represent different data. It also integrates semantic information into the tuple so that it can be more easily recognized by other processes. Our evolving tuples model, described next, uses a similar approach to placing meta-information with the application data in the tuple.

While the original Linda tuple space was intended for coordinating parallel processes, tuple space models have been adapted for mobile and pervasive computing. Lime [21], for example, uses a single tuple space whose contents are globally distributed. Other approaches (e.g., MARS [4] and

TOTA [20]) use many smaller tuple spaces located on physically distributed nodes. In both cases, the approaches must be coupled with routing mechanisms that move tuples and requests for tuples through the network. Our evolving tuples model uses an approach similar to the latter case; we define several small tuple spaces and focus on how tuples move in and out of these distributed spaces.

2.2 An Overview of Evolving Tuples

The goal of this paper is to introduce a resource discovery mechanism for pervasive computing applications that maximizes application expressiveness and flexibility without sacrificing simplicity. This resource discovery model, described in Section 4, relies on a new approach to tuple space based coordination. Existing coordination mechanisms founded on tuple spaces do not completely address the requirements of pervasive computing environments as described in the previous section. Most importantly, it is difficult if not impossible to embed adaptive behavior in the tuples themselves. Instead, the above approaches rely on external processes and operations to encode adaptation. To remove such requirements for *a priori* information about adaptive behaviors, we favor an approach that combines semantic data and context adaptation into a single structure.

In this paper, we introduce evolving tuples to accomplish this adaptive coordination. Evolving tuples extend Linda tuples in two major ways. To each field in a tuple we add *name* and *formula* elements allowing us to identify each field and to specify its behavior over time. The former is similar to some of the approaches cited above in that it allows us to directly tag the data with semantic descriptions of it. In contrast, the formula allows us to embed adaptive behaviors directly in the data. Secondly we add the `evolve(...)` operation which uses the field formulas to produce the next evolution of the tuple. Evolutions are directly impacted by the context in which they occur; successive evolutionary steps of the same original tuple in different contexts can result in two completely different derivative tuples. It is from this evolution that evolving tuples acquire the ability to provide highly expressive context adaptation that is simple for application developers to specify and understand.

2.3 Decoupling Fields from their Order

As noted in ELights [13], the addition of a name element to tuple fields greatly increases the ability to share tuples between applications, especially if those tuples are developed by different organizations. Our use of field names draws directly on this work to provide tuples the flexibility required for pervasive computing. With the inclusion of the *formula* element, the format of an evolving tuple is:

$$\begin{array}{l}\langle\ (name, type, value, formula),\\ \ \ (name, type, value, formula),\\ \ \ \ldots\rangle\end{array}$$

Each field's *name* is a unique and descriptive identifier, *type* is the data type of the field's *value*. The *formula* is discussed below. With the addition of the field name, we must also alter the format of the tuple templates used by tuple space operations. The ELights system that introduces name elements to tuples also includes the ability to specify highly expressive *constraints* on field values. In this model we choose to adhere to the original Linda specifications and allow the user to match either the exact value (actual) of a field, or the type of the field's value (formal). This subset of functionality suits our application requirements but may be expanded later as we further develop the model. An actual is specified by setting a concrete value for the third element of a template field, while a formal is specified by setting this element to null (∅). In either case, the field formulas are ignored when matching tuples and templates. The format of a template is simply the name, type, and value of each of the fields that must be matched:

$$\begin{array}{ll}\langle\ (\text{name, type, value}), & \leftarrow\text{Actual}\\ \ \ (\textit{name}, \text{type}, \varnothing), & \leftarrow\text{Formal}\\ \ \ \ldots\rangle & \end{array}$$

The template's *name* and *type* elements have the same meaning as in a tuple. The third element can take either the null value (∅) or an actual. The null value effectively turns the field into a formal, indicating that the matching function should only be concerned with the type. If an actual is provided, the matching function requires the candidate field to contain the same value.

The matching function $\mathcal{M}$ used by `in` and `read` is defined for a tuple θ and a template τ as:

$$\begin{array}{rl}\mathcal{M}(\theta,\tau) \equiv \langle \forall c : c \in \tau :: \langle \exists f : f \in \theta & \\ & \wedge\, f.name = c.name\\ & \wedge\, f.type\ =\ c.type\\ & \wedge\, (c.value = f.value\ \vee\ c.value = \varnothing)\rangle^1\rangle\end{array}$$

For each field in the template, the tuple must contain a field with the same name and type. If a template field also specifies an actual, the field must have a value equal to the one specified. However, a template may also match a tuple with more fields then the template. Specifically, the fields in a template must be a subset of the fields in any matching tuple. This flexibility allows applications to use data from different sources provided a consistent naming scheme.

2.4 Embedding Evolution

Our second distinguishing change, the field formula construct, is designed to allow a tuple author to specify incremental changes to a field's value. When evaluated, field formulas can access values associated with the tuple's context, allowing the changes to reflect the current state and history. These field formulas do not support complex programming constructs such as loops and callable functions, however they do support string concatenation and a simple if-then-else construct[2] in addition to common arithmetic and boolean operators.[3] These operators can be applied to both constant values and to values from other tuple fields which

[1]In the three-part notation: $\langle op\ quantified_variables\ :\ range :: expression\rangle$, the variables from *quantified_variables* take on all possible values permitted by *range*. Each instantiation of the variables is substituted in *expression*, producing a multiset of values to which o*p* is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies *range*, then the value of the three-part expression is the identity element for o*p*, e.g., *true* if o*p* is ∀ or ∅ when op is set.

[2]In our examples, we use the notation *if*(*condition*, *expression_1*, *expression_2*). The value of the entire expression is the value of *expression_1* if *condition* evaluates to *true*, and *expression_2* otherwise

[3]Due to length constraints, it is impossible to present the entire grammar of the allowed formulas.

are resolved when the formula is evaluated. Constant values such as *true*, 0.2, or "Hello world" appear in-line in the formulas. Values from other fields are referenced by the field's name, as in *counter + 1*.

A formula also has access to an "evolution context tuple" whose field values are referenced with the notation "*context*[*field-name*]", where *field-name* is the name of the context tuple's field. Section 3 will explore the nature of this evolution context tuple and its fields in more detail; our discussion of field formulas and the evolve operation simply assume this tuple which is supplied by an external process to provide information about the tuple's current environment. A formula's reference to a context tuple's field is replaced by the field's value when it is evaluated.

By using a field formula, a tuple can update its field value to reflect its environment. Consider, for example, the following field within a tuple:

⟨ . . .,
(*last-year*, int, 2006, *context*[*current-year*] - 1),
. . .⟩

When the field formula (*context*[*current-year*] - 1) is evaluated, it will return the value of the evolution context tuple's *current-year* field, reduced by one.

2.5 Performing Evolution

The process of evaluating field formulas is handled by the `evolve()` operation, which is responsible for choreographing the process of evolution. Using context information and the values and formulas of an existing tuple, `evolve()` generates a new tuple representing the tuple's next evolution. Since field formulas are able to reference the values of sibling fields, we must take care to evaluate formulas for the fields of a tuple in a specific order to elicit deterministic behavior. For example, if the formula for field A references a sibling field B which also contains a formula, the results of the evaluation of A's formula may differ depending on the order of the formulas' evaluation.

Determining the order of evaluation for an evolution step must be based on a universal standard and should also be intuitive to users. Various evaluation orders are available, such as "alphabetically by field name" which would yield a simple-to-implement standard. However this ordering is not intuitive to users. Instead, we logically create a dependency tree and evaluate fields that are depended upon before the fields that depend on them. In addition to being intuitive, this technique ensures that the values appearing in the resulting tuple are the same as those used to compute the other values, ensuring a consistent data structure.

This ordering does, however, impose the additional requirement that formulas do not create circular dependencies. We make one exception to this rule to allow a formula to reference itself. In this case, the value used in the evaluation is the field's previous value. We feel that the restriction on circular dependencies is more than offset by the deterministic and intuitive behavior that it provides. Given this restriction, we can formalize the `evolve()` operation using the following definitions.

First, let f refer to a field in a tuple (i.e., one name, type, value, and formula combination). Within a field, $f.formula$ refers to the code that specifies that field's evolution. Within the formalization that follows, a *formula* has three components. The first specifies the names of the sibling fields other than itself that the formula relies on. The second specifies the names of the fields of the evolution context that the formula relies on. The third specifies the executable behavior. That is, a formula, ϕ can be represented as the triple: $\phi = \langle D, E, behavior\rangle$. D and E are specified simply as sets of *names*. These dependencies are extracted from the formula when it is parsed by `evolve()` and are easy to separate based on notation. We also define the following piece of shorthand notation: $names(\theta)$, which allows us to access the set of names contained within the tuple θ.

Let θ' be the result tuple that is constructed incrementally during evolution from the original tuple θ. Fields in the tuple evolve one at a time, and as each field evolves, it is added to the result tuple θ'. Initially, θ' contains no fields. Before formally defining tuple evolution, we define what it means for a single field f in the tuple θ to be *enabled*, i.e., to be capable of being evaluated:

$$\boxed{f.enabled} \triangleq f.formula = \varnothing \;\vee\; f.formula.D \subseteq names(\theta')$$

The above states that a field's formula is enabled exactly when either no evaluation is required (the formula is $\varnothing$) or the sibling fields that a formula depends upon have been added to the new tuple θ' (i.e., they have already been evaluated for this evolution step).

We now define evolution of a tuple in terms of single steps that evolve one field at a time, ultimately generating a new tuple (θ') that has exactly the same field names and formulas as the original tuple (θ) but potentially new values:

$\boxed{\theta' := \texttt{evolve}(\theta, \varepsilon)} \triangleq$
  $\theta' = \text{newTuple}()$
  while $f := f'.(f' \in \theta \wedge f'.enabled \wedge$
      $f'.name \notin names(\theta'))^4 \neq \varnothing$ **do**
    if $(f.formula.E \in names(\varepsilon))$ **then**
      $new_value := \text{exec}(f.formula, f.value, \theta', \varepsilon)$
      $\theta'.\text{add}(\langle f.name, new_value.type, new_value, f.formula\rangle)$
    else
      $\theta'.add(f.name, f.type, f.value, f.formula)$
    endif
  od
  while $f := f'.(f' \in \theta \wedge f'.name \notin names(\theta')) \neq \varnothing$ **do**
    $\theta'.\text{add}(\langle f.name, f.type, f.value, f.formula\rangle)$
  od

where ε is the evolution context tuple provided by the calling process. The guard on the first loop in this definition requires that there exists a field in the original tuple that is enabled and has not yet been evaluated for this evolution step. As long as such a field exists (i.e., the non-deterministic selection results in a non-null value), the selected field is subjected to a second guard requiring the formula's dependencies on the evolution context are satisfied. If these dependencies are present, the field is evolved, and the result of the evolution is placed in the result tuple θ'. Since the evolution context tuple ε does not change during evolution, any formula that fails the second guard can not be evaluated during this evolution, and the field is simply copied with its

[4] The construct $x.$(`condition`) non-deterministically returns any value that matches the `condition` and will return $\varnothing$ (null) immediately if no value can be found [1].

original value to the new tuple. The evolution of a particular field (either evolved or copied) may enable additional fields in the original tuple whose formulas rely on the new field. When there are no more enabled fields in the original tuple, the second loop copies any remaining unselected fields to the result tuple; their evaluation cannot be enabled in this context.

After evolution, this new tuple can be inserted into a tuple space for other processes to consume. Since the new values of the tuple have drawn from the environment, any process inspecting these fields is using context information to influence its behavior. Additionally, since the tuple can change its values and types, its formulas can use environmental information to alter the tuple itself so that it will be selected by templates used by specific processes. In the next section, we will see how tuples use context-awareness and how they can change personas as they progress through the network.

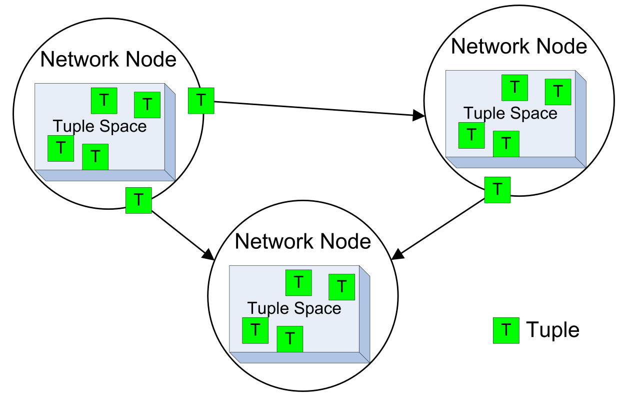

Figure 1: Ad-Hoc network using tuples to exchange application data

3. EVOLVING TUPLES IN PERVASIVE ENVIRONMENTS

The previous section defined the evolving tuples model and showed how tuples change over time in response to the contexts in which they are placed. In this section, we begin to explore the use of this model in pervasive environments in general and show how expressing of messages as tuples and processing of messages as tuple evolutions simplifies application-level understanding of communication in pervasive environments. The next section will build on this generic model to demonstrate how evolving tuples can be used to build a specific type of pervasive computing communication: resource discovery.

We envision pervasive computing environments to consist of a heterogeneous collection of distributed devices that communicate directly with each other and with users immersed in the environment. In such scenarios, it is impractical to assume that a centralized authority exists to mediate all communication; instead applications rely on *ad hoc networks* for interactions among application components and embedded capabilities. This system view is directly in line with the requirements enumerated in Section 1.

In applying evolving tuples to pervasive computing, applications perceive the data and resources available on hosts distributed throughout the network to be available within a collection of *tuple spaces*. Each participating device hosts one or more tuple spaces, and these repositories serve as mechanisms for coordinating applications' actions. This model is shown in Figure 1, which depicts a small network of coordinating devices. This representation is similar to that used in other tuple space adaptations for mobile computing [13, 20, 21], where the union of all of the tuple spaces in a connected network can be viewed as a *global virtual data structure* [22]. In our evolving tuples model, entry to and exit from tuple spaces may be guarded by tuple evolution operations, as dictated by application requirements.

The allocation and layout of the tuple spaces and the use of the `evolve` operation described in the previous section are purpose-dependent, and different applications are likely to use different collections of tuple spaces to enable their particular coordination tasks. In this section, however, we look at the tuple spaces and tuple space interactions that must be defined to allow pervasive computing applications to perform network communication actions to transmit and receive tuples. The behavior we describe here generically supports any pervasive computing behavior that may reside in a layer above the reception and distribution of tuples across the network.

An overview of a single node's participation in this communication process is shown in Figure 2. The figure shows tuples arriving from the network layer into a `receive` method. This method assumes that every tuple contains a field named *destination*. If this is not the case, the `receive` method does not know how to receive this tuple, and it will be dropped. Upon reception, incoming tuples are immediately evolved with an evolution context containing information about the network context in which the tuple was transmitted. If, after evolution, the tuple has a *destination* field with either the host's address or the broadcast address, it is placed in the *application* tuple space where local applications can consume it. Broadcast tuples and tuples not bound for the host are placed in the *outgoing* tuple space where they are forwarded through the network by the `Propagate` process. Tuples are dispatched to the network using a `send` method on the network interface. We assume that the network interface handles one-hop unicast and broadcast routing. We also assume an underlying route discovery protocol that provides the address of the next hop on the route to a given destination address (other than the broadcast address).

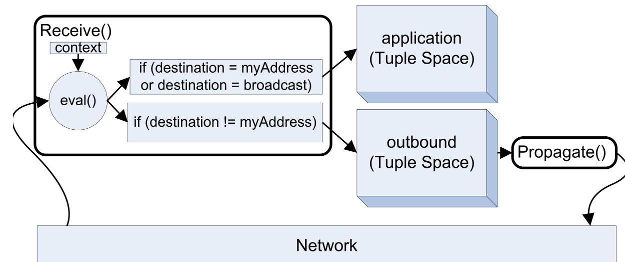

Figure 2: Tuple propagation mechanics

As indicated in the figure, the `receive` method demultiplexes tuples received from the network interface. A key component of this method's behavior is its use of the `evolve` method to potentially update the fields of the received tuple. This is important because applications may make decisions

based on the network contexts a tuple has recorded in its route to the application. As one example, real-time applications may be concerned with the amount of latency a tuple experiences between the source and the destination. A field inside the tuple can monitor this value, and the `evolve` operation in the `receive` method can update the value. Consider, for example, the following field within a tuple:

$$\langle \ldots,\ (\mathit{latency}, \text{float}, 0, \mathit{latency} + \mathit{context}[\mathit{last\text{-}link\text{-}latency}]),\ \ldots \rangle$$

This can communicate to the application the overall network latency that the tuple has experienced. As the tuple traverses the network, the latency experienced at each hop is aggregated and placed in the tuple. The evolution of the *latency* field defined above relies on its own previous value and a value that must be provided by the evolution context: *context*[*last-link-latency*]. Because this and other similar metrics can be measured at the network level, an evolution step on entry to the `receive` method allows the tuple to carry this network status information up to the application. We assume this network information is available as the `receive` method's evolution context in a tuple named "network-context." The behavior of the `receive` method is formalized as:

```
receive(θ) ≜
    θ' := evolve(θ, network-context)
    if (θ'[destination].value = my-address or
        θ'[destination].value = broadcast) then
            application.out(θ')
    fi
    if (θ'[destination].value ≠ my-address) then
            outbound.out(θ')
    fi
```

In this definition, after evolving the received tuple θ within the network context, the `receive` method checks θ's *destination*. If *destination* is either the node's local address (*my-address*) or the broadcast address, the tuple is delivered to the application. This is accomplished by placing the received tuple in the *application* tuple space, which serves as a communication interface between the network and the application. In addition, broadcast tuples and tuples addressed to other nodes should be propagated by placing the tuple in the *outbound* tuple space, which is accessed by the `Propagate` process, described below.

This description of the `receive` method does not handle the reception of duplicate broadcast tuples. Instead, we assume that each tuple has a unique sequence number (for example, a concatenation of the source address and a counter). As tuples evolve and are duplicated, the sequence number does not change unless explicitly reassigned (i.e., it has a $\varnothing$ formula). The above `receive` method assumes that a filter running at the network interface monitors which sequence numbers have been received and only passes up "new" tuples. More sophisticated communication methods (e.g., those specified in [15, 23]) make context-dependent decisions as to whether or not to look at a duplicate tuple (e.g., when the incoming tuple has a lower latency than the previous reception of the same tuple). In such cases, the above assumption is not valid, and a more sophisticated `receive` method is required. We ignore this issue for the remainder of this paper for the sake of clarity.

When broadcast tuples or tuples from the application need to be sent out on the network interface, they are placed in the *outbound* tuple space. A dedicated process, `Propagate`, continuously checks this tuple space for outgoing tuples.

```
Propagate ≜
    while true do
        template := ⟨(destination, int, ∅)⟩
        t := outbound.in(template)
        if t[destination].value ≠ ∅ then
            next-hop := lookup(t[destination].value)
            send(t, next-hop)
        fi
    od
```

The `Propagate` process non-deterministically removes a tuple from the *outbound* tuple space. If the tuple has a null ($\varnothing$) *destination*, the `Propagate` process ignores (i.e., drops) it. Otherwise, the process looks up the next hop on the route to the stated destination and sends the tuple there. The lookup function accesses a routing database maintained by an underlying routing protocol. The function returns *broadcast* if the destination address is broadcast and a network id of the next hop otherwise.

Throughout the `receive` method and the `Propagate()` process, tuples are handled based on their *destination* fields. Since the field's value is updated by its formula, the tuple can be viewed as providing its own routing information. This *self-routing* functionality can be leveraged by a tuple author to elicit elaborate behavior from a tuple such as waypoint based traversals of the network.

The `Propagate` process above provides simple non-deterministic behavior, and, like any non-deterministic algorithm, it lacks any fairness guarantees. For example, if enough tuples are being inserted into the *outbound* tuple space, and the `Propagate` process is slow, some tuples may never be removed and forwarded to their destination. More sophisticated `Propagate` processes could provide, for example, FIFO behavior to ensure this fairness.

As indicated above, these reception and propagation capabilities are intended to be generic across a pervasive computing system. While different implementations of `receive` or `Propagate` can provide different behavior, these behaviors are application-independent and may be tailored to a particular pervasive network's operating characteristics. Application-specific behavior is implemented on top of these communication capabilities. In the next section we describe the use of the evolving tuples model from the previous section in conjunction with the network model described here to provide a resource discovery mechanism that satisfies the requirements from Section 1 and is simple for application developers to implement and use.

4. RESOURCE DISCOVERY WITH EVOLVING TUPLES

To share its resources, a host must be able to receive resource requests and send appropriate replies. Often the process of matching resources to requests requires information from application level processes (e.g., physical location) as well as from various levels of the networking stack (e.g., link latency, buffer saturation). In addition, the context information available for evaluating a resource's suitability is often

restricted to the values originally built into the resource discovery protocol. If a certain attribute (e.g., manufacturer) was not specified during the protocol's design, a resource request may not be able to restrict resources based on its value. Evolving tuples allow us to collect context information by name from any provided context and allows the tuple author to combine these values using mathematical and logical operators to form a variety of preference behaviors.

In this section, we first show a discovery request expressed as an evolving tuple, explain the various fields, and show how the fields are essential to providing context-aware resource discovery. As resource discovery tuples are broadcast across the network using the facilities described in the previous section, they will be inserted into the *application* tuple space as described in the previous section. The `Discovery` process defined below is responsible for receiving these tuples and evolving them in the context of each available resource. After an explanation of how this process is modeled, we include a step-by-step example of a discovery tuple as it passes through a sample network.

4.1 Crafting a Resource Request

A resource discovery request, like other evolving tuples, is a self-routing structure that gathers data and changes its values as it progresses through a series of evolutions. The tuple will cross a multi-hop network and will be evolved in the context of potentially matching resources. If a suitable resource is found, the tuple switches itself from a request to a reply and sends itself back to the original source.

Figure 3 shows a tuple that a requester would create to locate a printer within 100ms network latency (0.1 seconds[5]) This request has embedded within it all of the information necessary to evaluate the request (in the context of potential resources) and to return a reply. Because this definition is under the control of the requester, many options can be changed depending on application requirements, for example, dispatching the reply to a node other than the requester itself. Below we explain each field and formula in the example, which assumes that the broadcast address is -1.

⟨
  (*source*, int, 5, ∅),
  (*latency*, int, 0,
        latency + *context*[*last-link-latency*]),
  (*match*, boolean, ∅,
        if (match ≠ ∅,
            match,
            (*latency* ≤ 0.1 **and**
                context[*resource-type*]="printer"))),
  (*msg_type*, string, "discovery-request",
        if (*match* = true,
            "discovery-reply",
            "discovery-request")),
  (*destination*, int, -1,
        if (*match* = ∅,
            if(*latency* < 0.1, -1, ∅),
            if(*match* = *true*, *source*, ∅))),
  (*resource-host-address*, int, ∅,
        if (*resource-host-address* = ∅ **and** *match* = true,
            context[*host-address*],
            ∅))
⟩

Figure 3: Example discovery tuple

- *source*: The network address of the node sending the resource request. In our example, this value will be used as our destination for the reply sent when the request finds a matching resource.
- *latency*: A simple counter that tracks the total network latency of the links the tuple has traversed. When this field is evolved by the `receive` method, the latency of the last transmission is added to the cumulative total. When the field is evolved by other processes, the *last-link-latency* field is not present in the evolution context, preventing the formula from being evaluated.
- *match*: Serves as a simple switch to notify other fields that a resource has (or has not) been found. The initial value is set to ∅ to indicate that no resource has been evaluated. When the formula is evaluated, the value becomes either *true* to denote a suitable resource, or *false* to denote an unsuitable resource. We use an if statement to preserve the field's value if it is non-null.
- *msg_type*: A flag used to retrieve resource discovery tuples from the *application* tuple space. Initially, and any time before the *match* field is set to true, the message type is "discovery-request". Once the tuple has found a matching resource, the message type is changed to "discovery-reply". Since this reply tuple will eventually be returned to the source, a process there can use this value to retrieve a successful resource discovery. The *msg_type* field may also be used by other application level processes, for example a print daemon might use tuples with a *msg_type* of "print-job".
- *destination*: Used by the `Propagate` process to forward the tuple. Until a resource has been evaluated (i.e., *match* = ∅) the *destination* is a function of *latency*. We set the *destination* to the broadcast address if the next hop will not exceed our scope, and set the value to null (∅) if it will. After a resource has been evaluated, the *destination* is a function of *match*. If *match* is *true*, we set our *destination* to the value of the *source* field. If *match* is *false*, we set our *destination* to null, causing the `Propagate` process to drop the tuple.
- *resource-host-address*: originally null, this value is set to the address of the host that contains the matched resource. The if statement guards the value from being updated once it has been set and only sets it when a match has been found.

As shown above, the data and logic necessary for selecting resources is completely contained within the tuple with the exception of the requisite resource attributes. This reduction frees the host system from any deep involvement in the process of discovery, and, as we will see next, the interface between the host and tuple has been reduced to just the *destination* and *msg_type* fields.

4.2 Discovery Resolution

In our model, the `Discovery` process is responsible for converting request tuples into reply tuples. As discussed in

[5]This tuple assumes that quantities are expressed in ISO units (e.g., seconds instead of milli-seconds).

Section 3, tuples addressed to either the broadcast address or a node's address are inserted into the node's *application* tuple space by the `receive` method. As shown in Figure 4, the resource discovery tuples are removed from the *application* tuple space by the `Discovery` process, using an `in` operation. Since this operation blocks when no matching tuples are available, the `Discovery` process is suspended until a request tuple is available.

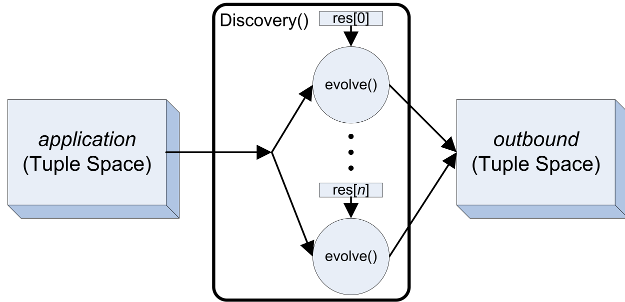

Figure 4: The discovery process

The descriptive attributes of the host's resources are accessed as the `Discovery` process's evolution context. The host maintains tuples that represent the resources available for discovery which are then used as the evolution contexts to evolve the discovery tuple for each resource. This procedure generates n new tuples that are each the result of evolving the original tuple with the ith resource where $i = 0..n$.

These new tuples are each potential matches that the `Discovery` process *could* filter based on the value of some standard field. However, by relying on the self-routing mechanism of the tuple, the `Discovery` process is relieved of any interaction with the tuples' fields beyond the template used for the `in` operation. The resulting tuples are simply deposited into the *outbound* tuple space where they are propagated to the node indicated by the tuple's *destination*. If the tuple represents a failed resource match, the *destination* will be null and the tuple dropped during propagation. We describe this formally as:

```
[Discovery()] ≜
  while true do
    template := ⟨(msg_type, string, "discovery-request")⟩
    θ = application.in(template)
    res[ ] := getResources()
    for each r in res
      θ′ := evolve(θ, r)
      outbound.out(θ′)
    rof
  od
```

This definition contains only the procedural framework for discovery, while all of the logic of matching resource attributes and determining resource satisfaction is delegated to the internals of the evolving tuple. The resource tuples used as evolution context are provided by the method `getResources()`, which may in turn retrieve its tuples from a separate tuple space (not shown). By using a 'probing group read' of this 'resources tuple space,' the operation would return all of a host's resources in a single step without blocking when no resources are available.

4.3 Example

In this section, we will step through the entire process of sending and returning resource discovery tuples. For this example we will use the network depicted in Figure 5. The discovery tuple from Figure 3 is sent by node 5 through the entire network, but we will specifically concentrate on the tuples that pass through nodes 1 and 2.

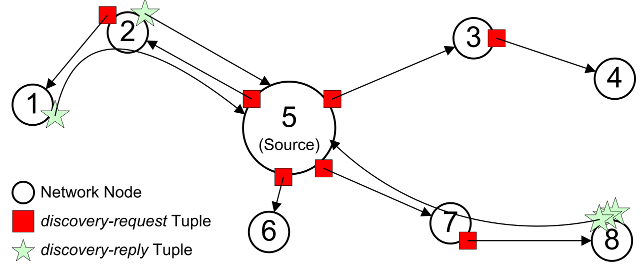

Figure 5: The dissemination of *discovery request* tuples followed by the return of *discovery reply*

The process begins when an application on node 5 creates a resource-discovery tuple. This tuple is inserted directly into the `outbound` tuple space where it is removed by the `Propagate` process described in Section 2. The `Propagate` process reads the value of the *destination* field (the broadcast address -1) and sends the tuple to each of the node's neighbors (2, 3, 6, and 7). When this tuple is received at node 2, it is handled by the `receive` method and encounters its first evolution (within the `receive` from Section 2).

This evolution is performed with a *network-context* tuple containing values describing the network. First the *latency* formula shown in Figure 3 is evaluated and increments the field's current value of 0 by *context*[*last-link-latency*] to (for example) 0.01. The *network-context* tuple does not contain a *resource-type* field, a dependency of the *match* field, so this formula is not evolved and retains its current value (∅). The other fields (*destination*, *msg_type*, and *resource-host-address*) each depend on *match*, so they are also not evaluated and retain their initial values.

After this evolution step is complete, the new tuple is inserted into both the *application* and *outbound* tuple spaces. From the *outbound* tuple space, the `Propagate` process will re-broadcast the tuple to all of the node's neighbors, where receptions of duplications of the same broadcast are handled as described in Section 3. The tuple in the *application* tuple space is removed by the `Discovery` process described in the previous subsection, which will then evolve the tuple in the context of each of the resources available on the node.

For the purposes of discussion, we will assume that node 2 contains only one resource, and the tuple describing it has a field named *resource-type* with value "printer." The `Discovery` process evolves the tuple just once with this resource as the evolution context. Since the *latency* field depends only on a context tuple field that is not available, its value is not changed. The sibling field *latency* and the context field *resource-type* are available, enabling the *match* formula. When it is evaluated, the value for the *match* field is set to *true*. This enables evolution of the remaining fields: *destination*, *msg_type*, and *resource-host-address*. The *destination* field is updated to the value of the *source* field (5), the *msg_type* field is set to "discovery-reply," and the *resource-*

host-address field is set to *context*[*host-address*] (2).

The `Discovery` process then places this evolved tuple into the *outbound* tuple space where the `Propagate` process will send it to its new destination, node 5. As mentioned previously, the `Propagate` process also sent the first evolution of our tuple to node 2's neighbors, or more specifically, to node 1. In our example, node 1 also contains a matching resource which causes the `Discovery` process to evolve a *discovery-reply* tuple and place it in the *outbound* tuple space. However, if node 1 also contains a resource with a *resource-type* field that is not equal to "printer", the `Discovery` will evolve a new tuple with a *match* value of *false* and a *destination* field of $\varnothing$. When this tuple is removed by `Propagate` from the *outbound* tuple space, it is ignored and dropped due to its null destination.

In Figure 5, node 8 similarly creates multiple discovery-reply tuples for each matching resource hosted on the node. Each of these tuples are inserted into the local *outbound* tuple space and are forwarded individually to node 5.

When the *discovery-reply* tuples arrive at node 5, the application that inserted the original tuple (or potentially another application) will retrieve the reply tuples from its *application* tuple space using an `in`($\langle\langle$*msg_type*, string, "discovery-reply"$\rangle\rangle$) operation. In our sample, the application would then use the value in the *resource-host-address* field to begin communications with the printer resources that were discovered.

5. RELATED WORK

As mentioned in previous sections, various systems exist that use tuple spaces to coordinate interactive behaviors among locally connected groups of nodes. MARS [4], for example, associates a tuple space to each host in a group of physically connected nodes. Mobile agents can roam from host to host, interacting with other agents through the locally available tuple spaces. The MARS employs reactivity in its tuple spaces to allow context information to impact some of these interactions. However, the context-awareness is embedded in these reactions, which are coded as separate entities from the data they impact. In our approach the tuples themselves carry the behavior that creates context-aware adaptation. LIME [21] is a more general model than MARS that also uses tuple spaces to simplify the development process. LIME enables mobile coordination by abstracting communication into a logically unified global tuple space that spans all connected nodes in a mobile network. At any instant, a device's perception of the world is through this tuple space which contains all of the data available on all connected devices. Like MARS, LIME relies on reactions external to the tuple space to create context-aware adaptation, while we focus on embedding this adaptation directly in the coordination model. TOTA [20, 19] has a more integrated approach to incorporating context-awareness into tuple spaces. In this system, tuples are automatically moved in a dynamic network according to contextual properties. TOTA subroutines can adapt to external properties in the environment and to the content of the tuples to make decisions regarding, for example, routing. While these subroutines can be carried within the tuple, the tuples effectively become empowered mobile agents. The evolving tuples model, on the other hand, maintains a tuple structure imposed over the data *and* the behavior in combination, maintaining the easy-to-use benefits of traditional tuple space approaches.

In general, these aforementioned tuple based systems provide complex behavior but often require rich environments to supply elaborate functionality. Our model joins the data and code into a simple structure that can elicit elaborate behavior from a very simple API. The evolving tuples model requires very little functionality (comparatively) from the host environment and thus promotes adoption across the wide variety of platforms in the mobile computing space.

An extension to LIME [10] uses tuple spaces to perform resource discovery in mobile ad hoc networks. This system is specifically designed to advertise remote services and propagate proxy objects to clients wishing to use them. The evolving tuples resource discovery model, on the other hand, does not specifically provide support for proxy-based service invocation, but these proxies could be embedded a binary data field in a future version. There are numerous other approaches to resource discovery in dynamic networks that are related to our approach in Section 4. We do not provide an exhaustive survey of them but instead highlight a few that are especially interesting. Systems such as [5] and [8] can be implemented as reactive protocols layered on top of existing routing support. However, the approaches use highly specific message formats and coordination patterns that force highly coupled interactions. By basing our resource discovery model on evolving tuples, we can leverage the simplicity of tuple-based coordination and delegate specifics of discovery decisions to the client application requesting a resource. The evolving tuples model also allows for new attributes and resources to be added to the system without requiring a global update to some *a priori* naming or coordination scheme. In addition, if a new value or field is added to a resource or a resource discovery, only the applications wishing to use this new element must be updated; remaining nodes and existing discovery requests are unaffected.

6. CONCLUSIONS

In general, pervasive computing applications rely on several constructs provided by underlying frameworks or middleware to achieve interactive capabilities. One of the most important of these is the ability to discover resources distributed in a dynamic network. In this paper, we have introduced the *evolving tuples* model (Section 2), which is tailored for use in pervasive computing environments. Evolving tuples embed context-aware adaptation within tuple structures, making it simple for application developers to explicitly specify how tuples should change in response to environmental conditions. We provided a formalization of this model and then shown how the model can be used to provide functionality generic to pervasive computing environments. Specifically, we show how evolving tuples can self-route through a network of connected hosts (Section 3). The approach to implementing the communication constructs is independent of the tuples' data content and depends only on the tuples' routing components. We use this foundational communication structure to provide a resource discovery framework for pervasive computing that uses decisions encoded within tuple formulas to make discovery decisions (Section 4). This reduces the requirement of *a priori* knowledge at the discovery endpoints and makes the discovery process more flexible for developers and responsive to application requirements. By using evolving tuples to implement necessary pervasive computing constructs such

as resource discovery, our approach provides a simple, extensible, flexible and expressive framework for engineering pervasive software.

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-0620245 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.

7. REFERENCES

[1] R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming*, 13(2-3):133–180, 1990.

[2] R. Bagrodia, S. Bhattacharyya, F. Cheng, S. Gerding, G. Glazer, R. Guy, Z. Ji, J. Lin, T. Phan, E. Skow, M. Varshney, and G. Zorpas. iMASH: Interactive mobile application session handoff. In *Proc. of Mobisys*, pages 259–272, May 2003.

[3] D. Balzarotti, P. Costa, and G. P. Picco. The lights tuple space framework and its customization for context-aware applications. *Int'l Journal on Web Intelligence and Agent Systems*, 2005. To appear.

[4] G. Cabri, L. Leonardi, and F. Zambonelli. MARS: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, July 2000.

[5] D. Chakraborty, A. Joshi, and Y. Yesha. Integrating service discovery with routing and session management for ad hoc networks. *Ad Hoc Networks Journal*, March 2004.

[6] A. Cole, S. Duri, J. Munson, J. Murdock, and D. Wood. Adaptive service binding middleware to support mobility. In *Proc. of ICDCS*, pages 369–374, May 2003.

[7] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. Murphy, and G. Picco. TinyLIME: Bridging mobile and sensor networks through middleware. In *Proc. of Percom*, pages 61–72, March 2005.

[8] C. Frank and H. Karl. Consistency challenges of service discovery in mobile ad hoc networks. In *Proc. of MSWiM*, pages 105–114, New York, NY, USA, 2004. ACM Press.

[9] D. Gelernter and A. J. Bernstein. Distributed communication via global buffer. In *Proc. of PODC*, pages 10–18, New York, NY, USA, 1982. ACM Press.

[10] R. Handorean and G.-C. Roman. Secure service provision in ad hoc networks. In *Proc. of ICSOC*, St. Louis, Missouri, 2003. Washington University, Department of Computer Science.

[11] E. Jaselskis and T. Elmisalami. RFID's role in a fully integrated automated project process. In *Proc. of ASCE Construction Congress 7*, 2003.

[12] C. Julien. Adaptive preference specifications for application sessions. In *Proc. of ICSOC*, volume 4294 of *Lecture Notes in Computer Science*, pages 78–89, 2006.

[13] C. Julien and G.-C. Roman. Egospaces: Facilitating rapid development of context-aware mobile applications. *IEEE Trans. on Software Engineering*, 32(5):281–298, May 2006.

[14] C. Julien and D. Stovall. Enabling ubiquitous coordination using application sessions. In *Proc. of Coordination*, pages 130–144, June 2006.

[15] S. Kabadayi and C. Julien. A local data abstraction and communication paradign for pervasive computing. In *Proc. of Percom*, 2007. (to appear).

[16] C. Kidd, R. Orr, G. Abowd, C. Atkeson, I. Essa, B. MacIntyre, E. Mynatt, T. Starner, and W. Newstetter. The aware home: A living laboratory for ubiquitous computing research. In *Proc. of the 2nd Int'l Workshop on Cooperative Buildings*, 1999.

[17] M. Klein and B. Konig-Ries. Combining query and preference: An approach to fully automize dynamic service binding. In *Proc. of the IEEE Int'l Conf. on Web Services*, pages 788–791, July 2004.

[18] D. Malan, T. Fulford-Jones, M. Welsh, and S. Moulton. CodeBlue: An ad hoc sensor network infrastructure for emergency medical care. In *Proc. of the Int'l Workshop on Wearable and Implanted Body Sensor Networks*, April 2004.

[19] M. Mamei and F. Zambonelli. Self-maintained distributed tuples for field-based coordination in dynamic networks. *Proc. of SAC*, 2004. Nicosia, Cyprus.

[20] M. Mamei, F. Zambonelli, and L. Leonardi. Tuples on the air: a middleware for context-aware computing in dynamic networks. In *Proc. of ICDCS Workshops*, pages 342–347, 19-22 May 2003.

[21] A. Murphy, G. Picco, and G.-C. Roman. Lime: A coordination middleware supporting mobility of hosts and agents. *ACM Trans. on Software Engineering and Methodology*, 15(3):279–328, July 2006.

[22] G. Picco, A. Murphy, and G.-C. Roman. On global virtual data structures. In *Process Coordination and Ubiquitous Computing*, pages 11–29, 2002.

[23] G.-C. Roman, C. Julien, and Q. Huang. Network abstractions for context-aware mobile computing. In *Proc. of ICSE*, pages 363–373, May 2002.

[24] A. Rowstron. Wcl: A co-ordination language for geographically distributed agents. *World Wide Web*, 1(3):167–179, 1998.

[25] N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *Proc. of SenSys*, pages 13–24, November 2004.

[26] G. Zussman and A. Segall. Energy efficient routing in ad hoc disaster recovery networks. In *Proc. of Infocom*, pages 682–691, March–April 2003.