Rapid Prototyping of Routing Protocols with Evolving Tuples

Drew Stovall
Christine Julien

TR-UTEDGE-2008-006

Rapid Prototyping of Routing Protocols with Evolving Tuples

Drew Stovall and Christine Julien

Mobile and Pervasive Computing Group
University of Texas at Austin, Austin TX 78712, USA,
{dstovall,c.julien}@mail.utexas.edu

Abstract. Developing software for dynamic pervasive computing networks can be an intimidating prospect. While much research has focused on developing and describing algorithms and protocols for these environments, the process of deploying these technologies is far from mature or streamlined. Furthermore, the heterogeneity of pervasive computing platforms can make the deployment task unapproachable. In this paper, we describe the evolving tuples model and demonstrate how a simple protocol can be quickly and easily developed. Since the evolving tuples infrastructure serves as a unifying base across heterogeneous platforms, the resulting implementation inherently supports cross-platform deployment, a common scenario for pervasive computing.

1 Introduction

Since the introduction of ubiquitous computing [14], much research has studied the coordination and collaboration of devices embedded in environments. As predicted, sensing and computing devices are been developed and deployed, and we are continually developing smaller, better, and longer lasting versions. Our rooms, halls, cars, and even parks will eventually be augmented with a plethora of devices to provide and consume all sorts of information.

However, the heterogeneity of devices can impede the creation and maintenance of applications. The variety of platforms to be supported requires an enormous number of protocol and application implementations. This inevitably leads to environments of incompatible, incomplete, and proprietary systems. Additionally, physical access to the devices to update hardware and software leads to massive efforts, making them impractical if not impossible.

Once deployed, this variety of hardware and software is a stumbling block to successful application maintenance. Small alterations to network protocols or node behaviors can cascade into significant code changes. The work required to recompile and redeploy new features can slow development. To address these issues, we introduce the *evolving tuples model* through which developers can make changes to protocols and applications, run different versions side-by-side, and add features without the cost of traditional redeployment.

In this paper we present the evolving tuples model and examine its use in prototyping behavior for pervasive computing environments. A simple route discovery protocol is described in detail to give a practical example of the work, showing the simplicity of prototyping applications using the evolving tuples model.

2 Background

Originally introduced as part of the Linda [6] system, a *tuple* is simply an ordered list of typed data fields. Tuples are collected in a bag-like data-structure called a *tuple space*. The addition and removal of a tuple from a tuple space is atomic, making it a natural mechanism for buffered communication between parallel processes.

In Linda, a process removing a tuple from the tuple space provides a pattern to which candidate tuples are compared. These patterns take the form of an ordered sequence of *actual* or *formal* values. A tuple that matches a pattern has the same number of fields as the pattern, equal values for any actuals, and the same type as any formals.

While forming a simple mechanism for passing data between processes, this design requires that data producers and consumers are maintained together. Any change in tuple format will require a similar change in the patterns used by existing processes. Since a pervasive computing environment typically consists of devices that are not under the control of a single entity, we must assume that tuple formats and tuple patterns will change independently. In our evolving tuples, as in LighTS [1] and ELights [9], fields are tagged with names, enabling us to decouple the tuple and pattern definitions.

Using tuple space systems, data can be effectively communicated between processes administrated by different organizations. However, behavior must still be specified *a priori* so that an application generating tuples can provide the right data to the consumers. Evolving tuples reduce this level of coupling by directly embedding some of the behavior we expect from nodes into the tuple itself. Specifically, rather then pre-defining the data manipulation recipe to nodes, evolving tuples allow tuple creators to stipulate this behavior at runtime.

3 The Evolving Tuples Model

In this section, we describe the evolving tuples model, consisting of three major components: the tuple format, the evolution process, and the standard deployment. A formal specification of many aspects of this model can be found in [12].

3.1 Evolving Tuple Format

In addition to the name element described in the previous section, the evolving tuples model adds a *formula* element to each tuple field. A field's formula specifies how the value is automatically updated or *evolved*. An evolving tuple is thus a set of tuple fields which comprise a *name*, a *type*, a *value*, and a *formula*.

A field's formula imparts behavior to the tuple as it passes through the network. Previous to the evolving tuples framework, tuple values were either immutable or altered only according to protocols already deployed to network nodes. Though it can be empty or null (⊘), a field's formula is nominally an arithmetic expression. A few simple logical functions are also provided [12].

These formulas can reference the values of peer fields by name. We also allow expressions to access values of a dictionary-like construct called the *evolution context*. The evolution context serves as a lookup table for sensor readings, configuration information, and other context related to the tuple's current location. To access values provided by the evolution context, formulas prefix the value's name with "EC." to differentiate them from references to peer fields.

3.2 Evolution

When a tuple is *evolved*, each of its field's formulas are evaluated, and the existing value is replaced with the result. Since formulas combine both the previous value and the values provided in the evolution context, the new value is viewed as an evolution of the field's value. If a field `count` has a formula of `count + 1`, the current value of the field would be incremented during an evolution.

3.3 Standard Deployment Model

The Evolving Tuples Model includes a reference design that represents the conceptual flow of tuples through each node. While the details of any particular implementation may differ, the externally observable behaviors of each should match those of this reference design. This model contains four components: the *receive* process and three tuple spaces: *inbound*, *outbound*, and *application*. The model is depicted in Fig. 1.

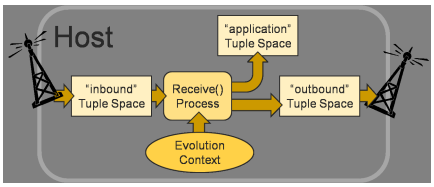

Fig. 1. Flow chart of receive process

Basic Tuple Exchange. Applications create, initialize, and deposit tuples into the *outbound* tuple space. Since messages require a destination, the reference model requires, at a minimum, a *destination* field. The field's value should be initialized to the address of a neighboring node or to the broadcast address. A system process monitors the *outbound* tuple space, removing tuples and transmitting them to their *destination*s. If a 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 host receives a tuple, it is placed in the *inbound* tuple space. The *Receive* process removes and evolves each tuple. If, after evolution, the tuple is destined for this node (via the *destination* field), a copy of the tuple is deposited in the *application* tuple space. If the tuple needs to be forwarded, the *Receive* process deposits a copy of the tuple in the *outbound* tuple space.

Broadcast and Duplicate Elimination. The use of the reserved broadcast address (typically -1) in a tuple's *destination* field designates that the tuple should be sent to every neighboring node. When using this address, the evolving tuples deployment model requires the use of a unique *id* field in the tuple to prevent the host from reprocessing tuples it has seen before. Because the tuple *id* is simply another field in the tuple, it is possible to alter the value of this id using the field's formula. When a tuple's *id* field is changed, it becomes a "new" tuple which will be processed by nodes that have already processed a previous incarnation.

4 A Routing Protocol

In this section we demonstrate how route discovery can be performed using the evolving tuples model. In a network of interconnected nodes, a route from one node to another can be found by flooding the network with a "route discovery message". If each node attaches its own address to a list of addresses in the message, a complete hop-by-hop route will be created. When the target node receives the message, a reply message is broadcast across the network to discover a route back to the source. A more complete discussion of route discovery for pervasive networks can be found in [8].

To build a route discovery tuple, we start with the *source* and *target* fields to hold the addresses of the source node and the target node respectively. No formula is specified since these are constants throughout the process. The *route* field carries the accumulated route to which each intermediate node appends its own address (EC.node). The tuple's *destination* field is used to propagate the tuple to the next node. With one exception, the tuple is always broadcasted and thus the value is usually assigned to the reserved broadcast address `-1`. When the tuple is being evolved on the source node, we assign the destination to the source's address (*source*) to prevent it from further propagation.

Name	Value	Formula
source	0	⊘
target	2	⊘
route	0	append(*route*, EC.node)
destination	-1	if (*source* == EC.node, EC.node, -1)
id	0.0	if (EC.node == *target*, `newUuid()`, *id*)

Table 1. Route Discovery Tuple

Since nodes discard tuples that contain *id*'s that have already been processed (to avoid duplicates), we must change the tuple's *id* when it arrives at the target before it is re-flooded back to the source. We accomplish this using a simple `if` statement. When the *id* formula is evaluated on the target node, the value is replaced with a new id generated by the `newUuid()` function.

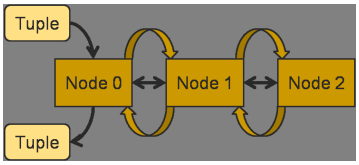

Fig. 2. Example network

The original tuple deposited by the application is shown in Table 1 (field types have been removed for brevity). In this example, the initiating application

resides on a node with address `0` and is attempting to discover a route to a node with address `2`. If nodes `0`, `1`, and `2` are interconnected as shown in Fig. 2, the evolution of the tuple's values are shown in Table 2.

When the tuple in Table 1 is deposited into node `0`'s *outbound* tuple space, it is broadcasted to neighboring nodes (i.e., Node `1`). As the tuple moves through this simple network and is evolved, its fields' values change.

When the tuple is evolved on node `1`, only the `route` field is changed, and the tuple is broadcasted again. When the tuple is evolved in node `2`, both the `route` and `id` fields are updated. By changing the `id` field, we allow the tuple to be received again by node `1` when node 2 broadcasts it again. Node `1` again updates only the `route` field, and the tuple is passed on to node `0`. Here the `destination` field is set to `0` to prevent any further flooding of the tuple.

Tuple Field	**Node 0**	**Node 1**	**Node 2**	**Node 1**	**Node 0**
id	0.0	0.0	2.0	2.0	2.0
source	0	0	0	0	0
target	2	2	2	2	2
route	0	01	012	0121	01210
destination	-1	-1	-1	-1	0

Table 2. Tuple Values (after evolution on the node heading the column)

5 Related Work

Early tuple space designs [6] and implementations [2] for Linda targeted parallel processing environments. Specifically, the atomic insertion and removal operations on tuple spaces relied on locks provided by shared memory. LIME [11] introduced distributed tuple spaces that provided the same atomicity guarantees across a truly global tuple space spanning many devices in mobile ad hoc networks. This adaptation of tuple spaces allows for an abstract representation of the network underlying a pervasive application but requires that tuples be delivered to consumers 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 more complex representation.

Mobile agent systems also combine behavior with the data that traverses the network. In an effort to provide a wide range of functionality, undue burden is placed on either the developer or the hosts. Systems like Agilla [5] require the developer to understand very low-level programming languages, while systems like TOTA [10] and MARS [3] 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 required of the network hosts.

While rooted in different technologies, there are a number of efforts to ease development for pervasive computing applications. Visual programming techniques [13] reduce the learning curve for new developers. Other approaches [7] provide abstraction to manage complexities. Still others address recompilation and redeployment head on through more complex hardware [4].

6 Conclusions and Future Work

We have presented the evolving tuples model and shown how it can be used to implement a simple route discovery protocol. As we continue to evaluate the model, we anticipate exposing a variety of other domains to which the evolving tuples model is well suited. We feel that our model has the potential to make developing pervasive computing applications more approachable and fruitful.

7 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

1. 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 (WAIS)*, 5(2), 2007.
2. P. Butcher. A behavioural semantics for Linda-2. *Software Engineering Journal*, 6(4):196–204, 1991.
3. G. Cabri, L. Leonardi, and F. Zambonelli. MARS: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, 2000.
4. M. Dyer, J. Beutel, and T. K. et al. Deployment support network - a toolkit for the development of WSNs. In *Proc. of EWSN*, pages 195–211, Jan. 2007.
5. C.-L. Fok, G.-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proc. of ICDCS*, pages 653–662, June 2005.
6. D. Gelernter and A. J. Bernstein. Distributed communication via global buffer. In *Proc. of PODC*, pages 10–18, 1982.
7. R. Handorean, J. Payton, C. Julien, and G.-C. Roman. Coordination middleware supporting rapid deployment of ad hoc mobile systems. In *Proc. of ICDCS Workshops*, pages 362–368, May 2003.
8. D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.
9. C. Julien and G.-C. Roman. EgoSpaces: Facilitating rapid development of context-aware mobile applications. *IEEE Trans. on Soft. Eng.*, 32(5):281–298, May 2006.
10. M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In *Proc. of PerCom*, pages 263–273, 2004.
11. A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: A coordination middleware supporting mobility of hosts and agents. *ACM TOSEM*, 15(3):279–328, July 2006.
12. D. Stovall and C. Julien. Resource discovery with evolving tuples. In *Proc. of ESSPE*, pages 1–10, Sept. 2007.
13. T. Weis, M. Knoll, A. Ulbrich, G. Muhl, and A. Brandle. Rapid prototyping for pervasive applications. *IEEE Pervasive Computing*, 6(2):76–84, Apr.-June 2007.
14. M. Weiser. The computer for the 21^{st} century. *Scientific American*, Sept. 1991.