Tuplespaces as Coordination Infrastructure for Interactive Workspaces

Brad Johanson and Armando Fox, Stanford University
Gates 3B-376, 353 Serra Mall
Stanford, CA 94305-9035
bjohanso@graphics.stanford.edu, fox@cs.stanford.edu

Abstract. Especially given the falling cost of hardware and ubiquity of local-area wireless networking technologies, there is high current interest in programming models and software infrastructures to support "Weiserian" ubiquitous and environmental computing. In this paper, we argue from both *a priori* reasoning and our experimental experience that, with slight modifications, the *tuplespace* programming model is a natural fit for such an application. From the functionality perspective, the constraints of the problem lead directly to those put forth when the tuplespaces were originally proposed. From the systems engineering perspective, the use of a tuplespace enables improved robustness and and more resilient resource management, which are necessary if ubiquitous computing is to achieve high penetration. In the context of our implemented prototype environment, the iRoom interactive workspace, we describe the principles that led us to this design choice, the modifications we made to the basic tuplespace model to improve its suitability for ubiquitous computing, our day-to-day experience over the past year and a half putting these principles into practice, and some avenues for future research. The experience and applications described run on top of infrastructure software tools that we distributing to enable other researchers to build on our efforts.

1 Introduction

Improvements in device technologies and falling costs are rapidly enabling the original vision of ubiquitous computing [20]. Devices from large wall-sized displays to small PDAs can easily (and wirelessly) be networked together in localized areas, forming the hardware side of the ubiquitous computing environment. Once connected together, however, the problem becomes how to allow programs running on the devices to coordinate with one another in a flexible and intuitive manner, given that they were not designed to work together.

Our own project, Interactive Workspaces, investigates the systems and HCI issues that arise in room-based ubiquitous computing environments that are technology-rich and consist of interconnected large and small displays and multi-modal I/O devices, where people gather to do naturally collaborative activities such as design reviews, brainstorming, etc. Compared to other projects, some of which we list toward the end of the paper, the systems half of our project is focused on providing infrastructure for dynamic, heterogeneous and ad hoc collections of devices, applications and operating systems, all of which may be either new or legacy. Based on our experience with the *iRoom*, our prototype Interactive Workspace, the realities of these environments make most currently-used programming models incomplete or inadequate. We have the following indispensable desiderata for our software infrastructure:

1. It must tolerate a dynamic environment: portable devices entering and leaving the room's wireless network should not disrupt the experience of others.
2. The room as a whole must maintain a high degree of robustness and availability despite inevitable (usually transient) software and hardware failures. This is especially important since part of experimental research includes fast prototyping, and we do not want such experimentation to destabilize an existing system.
3. It must allow the rapid integration of new devices *and systems*. A common-language approach such as "Java everywhere" is not enough by itself: we wish to leverage new devices *and* their existing application bases, i.e. Win32 productivity applications or Palm built-in PIM applications.
4. It must be portable across installations, allowing as much as possible for the heterogeneity of specific equipment installed at each site.

In the next section we begin by defending the assertion that most applications running in such an environment can be characterized as a collection of

autonomous traditional applications and devices loosely coupled in an ensemble (in the spirit of [10]). We then explain why a tuplespace model is well suited for this scenario as well as what changes we made to the basic model, and discuss our experience with the iRoom so far.

We also propose that there are several extensions to the basic tuplespace model that are necessary in an interactive workspace, and we explain why they are needed. Specifically, the extensions are self-describing tuples, tuple types and flexible typing, tuple sequencing and tuple expiration.

2 Tuplespaces and Ubiquitous Computing

2.1 Interactive Workspace Applications Are Ensembles

We assert that most applications running in an interactive workspace will consist of traditional applications and devices composed into an ensemble. There is no single dominant OS or programming environment in our scenario. OS demands may be dictated by the experience of programmers, the availability of device drivers for experimental I/O devices, the constraints of homebuilt equipment such as our hi-res Mural [13] (which is addressable only in OpenGL), and other factors. Therefore we must make it easy to use existing OS's and their associated applications as large building blocks, and identify ways to use those applications potentially in ways the designer never intended, for example by controlling them through "puppeteering" [8].

The question, then, is what model best facilitates this composition, such that the user has the impression of using one distributed application.

2.2 Coordination-Based Programming

In [10], Gelernter and Carriero proposed that computation and coordination should be thought of as orthogonal. Computation languages express how calculations proceed, and coordination languages express the interaction between autonomous processes (standard procedure calls being a special case in which the caller process suspends pending a response from the callee). They propose the Linda [1] *tuplespace* model as an example of a general-purpose coordination language. A tuple is a set of ordered typed fields, each of which either contains a value or is undefined; a tuplespace is an abstract space containing all tuples and visible to all processes. The language primitives are 'out' (puts a tuple into an abstract space), 'in' (consume a tuple from the space), and 'read' (copy a tuple), where the 'in' and 'read' operations supply a match template that may specify explicit values or wildcards for any tuple fields. It is easy to see how other coordination types such as RPC or message passing can be implemented on top of tuplespaces if they are more appropriate for some task.

2.3 Supporting Portability and Heterogeneity

Gelernter and Carriero argue that providing a coordination mechanism separate from the computational language provides two key features: *portability*, by providing a computation language independent mechanism of coordination, and support for *heterogeneity* by allowing devices and applications to coordinate with one another even if they are based on different hardware or languages. Since tuplespaces have only three primitives, they are easy to deploy on many devices and platforms, and it is easy to add wrappers to existing programmatic interfaces when source code is unavailable (similar to "puppeteering" [8]). Since coordination state is stored in the infrastructure (in the abstract tuplespace) and not in each client, even relatively impoverished devices can easily implement the model. Finally, while marshalling and un-marshalling are still required, the tuplespace code running on the client need only implement marshaling into and un-marshaling from the basic tuple format.

2.4 Robustness to Transient Failure

Failure isolation in tuplespaces is naturally achieved since autonomous receivers and senders don't directly interact. As long as the tuplespace infrastructure itself can tolerate a client failure, the failure of one client does not directly cause failure in another. Put another way, the natural indirection in communication leads to a programming style that makes cascading failure less likely. Tuples also persist, decoupling applications in time as well as space: an application can retrieve tuples posted while it was down.

2.5 Interposability and Snooping

Tuplespaces support coordination among multiple applications not originally designed to work together. Multicast communication between disparate groups of devices and applications is easy since multiple applications can get a copy of the same tuple if they all match for it. The rendezvous mechanics for applications are also straightforward, and are aided by the following three key features:

Interposability: Since tuples are public and indirectly sent between applications, an intermediary can pick up a tuple from a source and put back one or more tuples of different types which will cause the appropriate action in a receiver or receivers. This

allows applications not originally intended to work together to coordinate.

Snooping: The tuplespace model allows one component to snoop on tuples being sent among other components without impinging on their behavior. Information in that tuple can then be used to affect the local behavior of the snooping application.

2.6 Adapting Tuplespaces for an Interactive Workspace

Our current implementation of coordination, the Event Heap, makes some changes to the basic tuplespace model to address engineering needs of interactive workspaces:

Self-describing Tuples: Since ensemble components are not necessarily designed to work together, we give each tuple field a name as well as a value, so users can figure out the intent of tuples by browsing through the tuplespace.

Flexible Matching for evolvability: In the standard tuplespace model, the number of fields in a tuple and the order of fields is significant. Our system permits the tuple matching logic to ignore these attributes and match on field names and values only. Applications can thereby extend standard message types by adding extra fields, without breaking older applications; this enables continuous evolution. (This is analogous to enhanced Web clients or servers adding new HTTP headers.) Work on flexible typing [4] describes how such a system can still preserve the desired safety properties of strict typing.

Typed Tuples for anonymous communication: We require that all tuples include a designated *type* field, whose value denotes a generic tuple class potentially intelligible to many applications, and implies the presence and semantics of certain other fields in that tuple. For example, a "pointer event" tuple is expected to have *xPos* and *yPos* fields. (Since the fields are named, ordering does not matter.) This provides a useful compromise between strong and weak typing, although name collisions on tuple types are still possible. Tuple types also facilitate anonymous communication: as long as two applications understand the same tuple types, there is no need to explicitly coordinate their behaviors.

Tuple Sequencing: Traditionally, if multiple tuples exist that match the template tuple on a 'read' or 'in' operation, any of the matching tuples can be returned on any call. Tuple sequencing means that receivers will not read the same tuple more than once. Sequencing ensures that applications requesting state change tuples will get tuples exactly once, and in order, rather than fetching the same tuple repeatedly. Nonconsuming read is still supported as a "snoop" method, which allows applications to peek at tuples without affecting sequencing.

Expiration of Tuples: If tuples intended for a particular recipient are never consumed (perhaps the recipient(s) have failed), tuples may build up in the tuplespace. We address this by giving all tuples a *TimeToLive* field that specifies how long (in wall clock time) until they are "garbage collected" by the tuplespace. In addition to eliminating the resource reclamation problem, expiration facilitates two other useful behaviors. First, it bounds the causal latency between the posting of a tuple and an action in response to reading the tuple. For example, a light should turn on within a few seconds of the remote control activation, or not at all. Second, expiration facilitates service discovery and advertisement via soft-state, announce/listen protocols [17], which are known to have excellent robustness properties (we describe our implemented service discovery and advertisement system in [16]).

3 Discussion and Experience

3.1 Design Alternatives Rejected

Publish-subscribe provides some of the same advantages as tuplespaces. However, P/S is primarily receiver-driven in that senders may not publish unless they know someone is listening; in contrast, the natural broadcast provided by tuplespaces makes anonymous rendezvous potentially easier. (Consider implementing a remote-control applet without knowing in advance how to bind to a receiver.) Also, P/S events lack persistence, making it difficult for a temporarily-down listener to retrieve an event after it comes back up. This makes it more difficult to keep he system running smoothly through a failure.

RPC/RMI suffer from well-known systems problems associated with transparency; they also lack temporal persistence in coordination, and make language independence more difficult since the method interface needs to be agreed upon ahead of time by all parties, and all parties must be notified if the interface changes. Since communication is direct, it is difficult to rendezvous programs not designed to work with each other via snooping and intermediation as described above. Both RPC/RMI and P/S can be implemented on top of tuplespaces if needed.

3.2 Drawbacks/Challenges of Tuplespaces

Scalability. Tuplespaces do not scale easily, since all participants communicate through a shared medium. We accept this tradeoff because the scale of an interactive workspace is bounded by human

interaction, i.e. it will contain perhaps tens of users and hundreds of processes. For this same reason, we anticipate that the best way to "link" multiple interactive workspaces will involve selectively filtering events to communicate among multiple independent tuplespaces. (It is unlikely that most messages controlling specific physical plant in room A are of interest to software in room B.)

Indirection. Peer-to-peer communication takes two hops. Since we are primarily interested in supporting interactions at human-scale latencies, the speed of current computers and networks diminish the importance of this issue in our domain. Also, in [6] it has been shown that a properly implemented tuplespace can adapt over time to route messages using a single hop.

Security is particularly important for multiple-space scenarios, but the "social model" for security even in a single space is not yet well-understood. In our prototype, complete trust is extended to all entities that can communicate via the tuplespace, and the entire environment is behind a firewall.

3.3 Experience To Date

Our prototype iRoom includes three SmartBoard wall-size touch-sensitive displays, a hi-res Mural display, a tabletop display, wireless LAN and wireless pointing devices, and integration with laptops and PDA's. Space does not allow a detailed description of our extensive experience with the iRoom to date. We routinely use several applications that rely on the functionality of the Event Heap, including SmartPresenter and multibrowsing [14] (exploiting multiple displays for PowerPoint presentations and Web browsing respectively), the Interface Crafter [16] and several associated "soft" remote controls for lighting, projector mux, etc. that can also be accessed handheld devices [9], and wireless buttons that can be programmed as "macros" (e.g. power up the entire room). Most of these were realized by combining a diverse array of off-the-shelf applications (e.g. PowerPoint) with tens or hundreds of semicolons of "glue". As a research lab, the room has been remarkably robust under day-to-day use, and also supports several outside groups that use it for demos and as a facility for their own (non-Computer Science) projects. We cannot prove that this combination of ease of integration and robustness would have been impossible with any other approach, but we believe we have demonstrated that our model is indeed a natural fit for such an environment.

3.4 Related Work

A large number of interesting and complex, but non-interoperable, projects [2][3][5][7][18] are investigating room or work-area based ubiquitous computing. Each has uncovered important insights in ubiquitous computing but have yet to propagate and deploy their frameworks significantly beyond the project's boundaries. Many are focused more on making the environment "smart" and responsive to users' needs, and have focused less on creating a reusable, portable and robust infrastructure. Hasha [12] proposes publish/subscribe for controlling homes filled with smart appliances, sensors and I/O devices; we believe the temporal persistence and expiration properties we have added to tuplespaces make them slightly more useful for connecting legacy components and applications and for dealing with partial failure and state corruption. Jini [19] provides lower level mechanisms by which clients and servers that understand common interfaces can interact with each other, but does not specify how coordination proceeds after the initial rendezvous. JavaSpaces [15] and TSpaces [21] are essentially direct implementations of the tuplespace model; we have extended TSpaces as described above to form the Event Heap, the basis of our coordination infrastructure.

4 Conclusions

Because of practicality and complexity constraints, many ubiquitous computing application scenarios will continue to be characterized as loosely-integrated ensembles of heterogeneous legacy components. We propose that this domain may be a "killer app" for the tuplespace model of coordination, because of that model's portability, extensibility, flexibility, and ability to deal with heterogeneous environments. In addition, we proposed key extensions to the basic tuplespace model for this domain: self-describing tuples, flexible typing, typed tuples, tuple sequencing, and tuple expiration. The results of using our implemented prototype (the Event Heap and the iRoom) routinely over the past year and a half have been very favorable, and we encourage interested readers to help evaluate and extend our approaching by setting up their own Interactive Workspace. The hardware can be easily simulated using a collection of PC's, handhelds and laptops, and the software described in this paper is available at **iwork.stanford.edu.**

References

[1] Ahuja, S., Carriero, N., and Gelernter, D., Linda and Friends, *IEEE Computer*, August, 1986.

[2] G.Abowd,"Classroom 2000:An Experiment with the Instrumentation of a Living Educational Environment," IBM Systems J.,Vol.38,No.4,Oct.1999,pp.508-530.

[3] Larry Arnstein et al. Ubiquitous computing in the biology laboratory. *Journal of Laboratory Automatio*n, March 2001.

[4] Begel, A. and Spreitzer, M. More Flexible Data Types. *Proceedings of The Eighth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET-ICE'99).*

[5] Brumitt, B., Meyers, B., Krumm, J., Kern, A. and Shafer, S.,. Easyliving: Technologies for intelligent environments. In *Handheld and Ubiquitous Computing 2000 (HUC2K*), Septemb er 2000.

[6] Carriero, N., Gelernter, D., Mattson, T., and Sherman, A., "The Linda alternative to message-passing systems", *Parallel Computing*, 20, 633-655, 1994.

[7] Coen, M., Phillips, B., Warshawsky, N., Weisman, L., Peters, S., and Finin, P. Meeting the Computational Needs of Intelligent Environments: The Metaglue System, *Managing Interactions in Smart Environments*,. Paddy Nixon, Gerard Lacey and Simon Dobson eds. Dublin, Ireland, 1999

[8] De Lara, E., Dan Wallach, and Willy Zwaenepoel. Puppeteer: Component-based Adaptation for Mobile Computing. In *Third USENIX Symposium on Internet Technologies and Systems*, San Francisco, CA, March 2001.

[9] Armando Fox, Brad Johanson, Pat Hanrahan, and Terry Winograd. Integrating Information Appliances into an Interactive Workspace. In *IEEE Computer Graphics & Applications*, Vol. 20, No. 3, May/June 2000.

[10] Gelernter, D., and Carriero, N., Coordination Languages and their Significance, *Communications of the ACM*, Vol. 32, Number 2, February, 1992.

[11] Robert Grimm, Tom Anderson, Brian Bershad, and David Wetherall. A system architecture for pervasive computing (PDF, 128 KB). In *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 177-182, Kolding, Denmark, September 2000.

[12] Hasha, R., Needed: A common distributed object platform, IEEE Intelligent Systems. March/April 1999.

[13] Greg Humphreys, Ian Buck, Matthew Eldridge, and Pat Hanrahan. Distributed Rendering for Scalable Displays. In Proc. *IEEE Supercomputing 2000*.

[14] Brad Johanson, Shankar R. Ponnekanti, Caesar Sengupta, Armando Fox. Multibrowsing: Moving Web Content across Multiple Displays. Technical Note in *UBICOMP 2001*, Atlanta, GA.

[15] Sun Microsystems Labs, JavaSpaces Specification, http://www.sun.com/jini/specs/js.pdf.

[16] Shankar R. Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, Terry Winograd. ICrafter: A Service Framework for Ubiquitous Computing Environments. In *Proc. UBICOMP 2001*, Atlanta, GA

[17] Suchitra Raman and Steven McCanne. A Model, Analysis, and Protocol Framework for Soft State based Communication. In Proc. *ACM SIGCOMM 99*.

[18] N.A. Streitz et al., i-LAND: An interactive Landscape for Creativity and Innovation. In Proc. ACM Conference on Human Factors in Computing Systems (CHI '99) , Pittsburgh, Pennsylvania, U.S.A., May 15-20, 1999. ACM Press, New York, 1999, pp. 120-127.

[19] Waldo, Jim, Jini Technology Architectural Overview, Sun White Paper, 1999

[20] Weiser, M., The computer for the twenty-first century. *Scientific Americ*an, pages 94–100, September 1991.

[21] P. Wyckoff, S. W. McLaughry, T. J. Lehman and D. A. Ford. TSpaces. IBM Systems Journal 37(3). Also available at http://www.almaden.ibm.com/cs/TSpaces.