Supporting Ordering and Consistency in a Distributed Event Heap for Ubiquitous Computing

Oliver Storz, Adrian Friday, and Nigel Davies

Computing Department, Lancaster University, Lancaster, UK
{oliver,adrian,nigel}@comp.lancs.ac.uk

Abstract. The Stanford Event Heap has been shown to provide appropriate support for constructing interactive workspace applications. Given this success it is natural to consider the Event Heap as a platform to support other classes of Ubiquitous Computing applications. In this paper we argue that the distributed, spontaneous nature of these applications places additional demands on the Event Heap that require extensions to both the engineering and API. Suitable extensions are described and their use to support a typical Ubicomp application is discussed.

1 Introduction

Recent years have witnessed the emergence of Ubiquitous Computing as an important and energetic research topic. As new classes of application have emerged, so the search for appropriate programming abstractions and associated middleware to simplify the development and deployment of such applications has gained pace. This process has yielded a number of prototype platforms including HP's Cooltown [8], UIUC's Gaia [11], and many others.

Probably the most successful platform to date (in terms of widespread adoption) is Stanford's Event Heap [7]. The Event Heap is based on the Tuple Space paradigm [4] in which data is passed between applications through the generation and consumption of tuples of data through a shared 'data space'. The Tuple Space paradigm provides decoupling in both time and space and hence is an attractive paradigm for use in environments that consist of collections of loosely coupled cooperating processes. The Event Heap is designed to support smart room applications [6], and extends the Tuple Space paradigm in a number of important ways.

Given the success of the Event Heap API we believe it makes sense to consider this as a starting point for more general Ubiquitous Computing and mobile computing applications. In this paper we suggest a series of further API extensions that enable the Event Heap to move beyond its current target domain. These extensions include support for new semantics and disconnected operation, and borrow in part from the work of Davies et al. on creating Tuple Space platforms for mobile environments [1].

This work was supported by EPSRC project *Grid based Medical Devices for Everyday Health* (GR/R85877) and the European Union funded *Simplicity* project (IST-2004-507558).

2 Tuple-Spaces, Mobile And Ubiquitous Computing

2.1 The Tuple Space Paradigm

The Tuple Space paradigm was conceived by Gelernter et al. [4] as part of Linda. Linda augments a traditional computational language (such as C or Pascal) with new operators for process creation and inter-process communication. The Linda model initially consisted of four such operators[1] :-

1. **out** inserts a tuple, composed of an arbitrary mix of typed fields, into the tuple space. Fields are termed 'actuals' if they contain a static value and 'formals' if they map onto program variables.
2. **in** extracts a tuple from a tuple space, with its argument acting as the template, or anti-tuple, against which to match. Actuals match fields of equal type and value; formals match fields of the same type. An anti-tuple matches a tuple iff all corresponding fields match. When a match occurs the tuple is withdrawn and any actuals it contains are assigned to formals in the template. Tuples are matched non-deterministically and **in** operations block indefinitely until a suitable tuple can be found.
3. **rd** is syntactically and semantically equivalent to **in** except that a matched tuple is not withdrawn from the tuple space and hence remains visible to other processes.

Linda was subsequently extended to support additional non-blocking operators **inp** and **rdp** [9] and high performance bulk primitives [13].

2.2 The Event Heap

Targeting interactive workspaces, the *Event Heap* [7] introduced a number of extensions to both the API provided by conventional tuple spaces and the underlying semantics of many of the operations. We briefly highlight the key aspects of this functionality here, please refer to Johanson's thesis [6] for a complete description.

Extended Delivery Semantics Applications performing subsequent read operations will see events from a single source in the order that they are produced by the source (*per-source ordering*). Moreover, due to the centralised nature of the current Event Heap implementation, events originating from multiple sources are in fact delivered *totally ordered*[2].

Persistent Queries The platform supports non-destructive read operations that persist over time, matching all known events that satisfy the constraints of the read template and all further matching events as they enter the Event Heap.

[1] We have omitted discussion of 'eval' for brevity as it is not important for understanding the contribution of this paper.

[2] Note that total ordering is the result of the current centralised implementation, not a requirement of the Event Heap paradigm *per se*.

Event Notification Similar to the concept of persistent queries, applications may register to be notified whenever new events matching a certain template enter the Event Heap.
Additionally, applications performing subsequent read operations are guaranteed to see each event *at most once*, enabling applications to "iterate" through events on a server by issuing subsequent read requests. In contrast, multiple subsequent read requests issued with traditional tuple space platforms might return the same tuple over and over again, regardless of the availability of other matching tuples — the 'multiple RD problem' [12].

IBM's T Spaces [14] was initially used to underpin the Event Heap. The Event Heap was later re-engineered to remove this dependency.

2.3 L^{2}imbo

L^{2}imbo [2] is a fully decentralised distributed tuple space platform designed principally for mobile environments. Unlike approaches based on fully consistent replicas [15] or mobile agents [10], replicas in L^{2}imbo are kept consistent on-demand using an IP multicast protocol based on Scalable Reliable Multicast [3].

L^{2}imbo included a number of important extensions to the standard tuple space model, including the ability to associate types with tuples and extend types to support sub-type matching, support for multiple (possibly specialised) tuple spaces, and an extended range of matching primitives (including Rowstron's BONITA high performance bulk primitives [13] and a basic eventing API).

3 Discussion

While the Event Heap API has proved to be well suited to application development, we believe that it lacks important facilities for constructing many types of Ubiquitous Computing application, as illustrated by the following scenario:

> *Alice, Bob, Joe and Sue are researchers at the University of X. While having lunch at a café, Alice articulates some new ideas regarding project Y. The group decides to use their mobile devices to further explore these ideas using a shared whiteboard application. Each member of the group uses his/her own display and stylus to contribute to the discussion. The individual devices are connected using a wireless ad-hoc network. After lunch, Alice and Joe decide to move to their office and finalise the design. In their office, they resume the discussion from where they left off.*

Spontaneous Interaction and Mobility

The Event Heap system is based on a single server instance running within each Ubiquitous Computing "interactive work space". However, this mode of operation

is clearly not suitable in our scenario since the devices involved are operating in a peer-to-peer ad-hoc mode.

It becomes clear that if we wish to use the Event Heap to support more general mobile and ubiquitous computing applications where users and/or devices spontaneously interact there is a need to offer a more highly available and scalable solution (namely, *distributed, replicated or federated local instances*). If individual mobile nodes are able thus able to implement their own local Event Heap (or appropriate proportion thereof), then applications can operate when the node is disconnected from the network. Furthermore, if local Event Heaps can be synchronised with other mobile nodes then distributed applications can operate without the need for additional infrastructure.

Consistent Behaviour 'within the Real World'

A move to a distributed Event Heap means that we no longer have a centralised point of synchronisation (i.e. a single Event Heap instance) and hence we must take care to respect event delivery and ordering semantics. For example, the changes observable on users' displays must remain consistent with the order of their actions (instructions through the system) — the users' knowledge of the behaviour of the system, exposed through their visual senses, places a total ordering constraint on many of the underlying events (as events trigger actions and these actions are visible in the real world). This contrasts with Johanson's claim that source ordering is sufficient for most classes of ubiquitous computing application [7].

We observe that since the Event Heap offers per-source ordering, and the Event Heap is currently implemented in a centralised fashion, then events are actually totally ordered, since the platform instance provides a single point of reference. However, it is clear that future Ubiquitous Computing applications will have varying requirements for consistency and ordering of the events representing the applications' state.

4 Proposed Extensions

In order to address the two requirements raised in section 3 (i.e. distributed operation and support for ordering and consistency semantics) we have extended the Event Heap API to enable dynamic creation, destruction and interaction with multiple distributed Event Heaps (including support for propagating events between Event Heaps) and to provide more sophisticated support for event ordering and delivery semantics.

4.1 Support for Multiple Event Heaps

In the general case we do not assume a single Event Heap that is accessible by all clients. We allow multiple Event Heaps and provide an API for their creation and destruction. These operations are designed to be sufficiently lightweight as

to encourage programmers to create new Event Heaps on the fly. As in L^{2}imbo, we provide a class of system agents, called a *factory*, that can create new Event Heaps configured to meet application specific requirements [5]. For example, in future versions of our platform we plan to allow the creation of Event Heaps with support for security (user authentication), persistence and event logging (e.g. for accountability in safety critical systems).

We allow logical Event Heaps to be distributed across multiple physical hosts. Distributing Event Heaps makes our platform more highly available and improves fault tolerance, e.g. in the case of failing hosts, and mobility.

We also use the concept of L^{2}imbo bridging agents to provide the means for linking arbitrary Event Heaps and controlling the propagation of events between them. In their simplest form, bridging agents are processes that subscribe for all events in one Event Heap and generate duplicate events in the context of a second Event Heap. Bridging agents can also provide more fine grained bridging based on event type and field matching[3].

4.2 Ordering and Delivery Semantics

We introduce two new concepts into the Event Heap API. The first is that of a *view*. Processes that share a view are guaranteed to have a consistent, ordered view of the Event Heap with the precise semantics being configurable on a per-view basis. Each view can be configured to provide no ordering of events, per-source ordering or total ordering. In the unordered case, the view simply ensures that all clients using the view will see the same (consistent) set of events without any guarantee of their relative ordering. Per-source ordering offers the same semantics as Johanson suggests for the Event Heap [7]. A totally ordered view guarantees that all operations performed within the context of the view will observe events in the order they were produced by the event source and, critically, in the same relative order across all sources. Thus two applications subscribing to events of a particular type would receive the events in exactly the same order, even if the events are produced by multiple sources. This provides total ordering as it is offered by the current centralised Event Heap.

The second concept we introduce is that of *session identifiers*. Session identifiers are used to provide at-most-once semantics for event matching (the default behaviour in the Event Heap). In the Stanford Event Heap this means that events are never returned to the same application twice, even when they match different templates: this may give rise to problems when applications wish to re-read events from the Event Heap that they have already seen.

We guarantee at-most-once semantics for non-destructive operations on a per-session identifier basis. Session identifiers can be thought of as containing a record of events that have already been seen within the context of a given session and hence are not to be retrieved a second time as a result of an application

[3] Note that bridging agents provide a mechanism for propagation of events *between Event Heaps* and are not required for the propagation of events between separate distributed instances of a single Event Heap.

request. Session identifiers can be created with either an empty record or with a record of events inherited from an existing session identifier. This provides an easy way for applications to clone session identifiers (e.g. for distributing these to other application instances). Clients are at liberty to discard sessions or maintain multiple sessions to best suit their needs. Processes can share session identifiers across hosts, enabling, for example, distributed applications to share the load of processing an event stream.

5 Implementation And Evaluation

5.1 Implementation

Views and session identifiers are created in a similar fashion to new instances of an Event Heap, i.e. using appropriate factories: applications output an appropriate creation request event into their Event Heap which is serviced by a factory on their local node. In the case of view creation these requests can be parameterised to specify the semantics that the view should provide, i.e. unordered, source or total ordering.

Our implementation is based on the protocol used in L^{2}imbo [1] to support distributed tuple spaces. We have extended this protocol to support views and session identifiers. L^{2}imbo utilises application level framing concepts based on Scalable Reliable Multicast [3] to promote scalability and avoid the need for fixed group membership. One of the key challenges in our platform is how to globally identify events and record their relative ordering when mapping them into a given session or view. We have rejected strategies based on global sequence numbers, vector clocks or atomic agreement, as these all require high (e.g. quorate) simultanous availability of all end-systems, which is not appropriate for our chosen application domain.

In our prototype, session identifiers and views are represented by distributed state that is kept consistent through the exchange of 'system events'. Each system event represents a token or 'lock' on a given session or view. Before the platform can map an event matching a template into a given session it must first obtain this lock. The lock exchange process triggers on-demand generation of a system event representing the session, causing the replicas of the session involved to synchronise. Note that any peer can snoop the current state of the session during this exchange.

Like session identifiers, views are based on single transferrable ownership of a shared token (system event) representing the current state of the view. Upon view creation all existing events in the Event Heap that are visible to the view creator are accessible via the view and according to the ordering requirements specified at creation time. Each platform instance maps any events that it snoops from the multicast channel into all views that it owns. A view must be owned (the system event for the view consumed) before operations that affect the view can take place (e.g. creation of new events). Any replica may cache the state of the view and perform matching operations without owning the view, providing

that the consistency and ordering guarantees are not violated (i.e. all earlier sequenced events in the view have been observed or are known to have been previously consumed).

5.2 Evaluation

For illustrating the qualities of our platform, we will further elaborate on the distributed whiteboard example outlined in section 3. As we will show, our platform not only supports ad-hoc interactions between entities in mobile and ubiquitous computing environments: the provision of extended, flexible delivery semantics also significantly simplifies the development of distributed applications.

During Alice's, Bob's, Joe's and Sue's lunchtime session, each device hosts an instance of the platform. These four instances form a single Distributed Event Heap. All participating whiteboard applications share a common view-identifier, specifying total ordering as target delivery semantics. The common view ensures that all applications receive all drawing events in exactly the same order, providing application behaviour consistent with the users' view of the physical world.

As the group splits and Alice and Joe resume the design meeting on their own, they are both able to restore the state of their local whiteboard applications by re-reading the set of events stored in their local instances of the Distributed Event Heap. Using the same view-specifier as before, Alice's and Joe's whiteboard applications are able to obtain a complete replay of all drawing events.

As Alice and Joe continue their work, new events have to be mapped into the shared view. Complications occur if neither Alice nor Joe owned the view during the lunchtime meeting. In this case, the view will be *cloned*, i.e. a new view will be created offering the the same semantics as the old view. The new view is initialised with a complete record of the ordering information associated with the old view. If either Alice or Joe "owned" the view during the lunch-time session, they can simply continue to use that view. In either case appropriate feedback needs to be provided to the users.

6 Conclusion and Future Work

In this paper we have described how a popular distributed systems paradigm based on the Event Heap can be extended to support distributed operation and selectable ordering and consistency guarantees through the concepts of shared views and session identifiers.

Using an existing API such as that of the Event Heap is an important aspect of our work, since moving towards a common programming model is crucial to the widespread growth of middleware support for Ubiquitous Computing. The paper reports on the development of a prototype distributed systems platform that implements shared views and session identifiers without breaking the principal benefits of time and space decoupling offered by the tuple-space paradigm.

References

1. N. Davies, A. Friday, S. Wade and G. Blair: *L2imbo: A Distributed Systems Platform for Mobile Computing*. ACM Mobile Networks and Applications (MONET), Special Issue on Protocols and Software Paradigms of Mobile Networks, vol. 3(2): pp. 143–156, 1998.
2. N. Davies, S. Wade, A. Friday and G. Blair: *Limbo: A Tuple Space Based Platform for Adaptive Mobile Applications*. Joint International Conference on Open Distributed Processing and Distributed Platforms (ICODP/ICDP '97). Chapman and Hall, Toronto, Canada, 1997.
3. S. Floyd, V. Jacobson et al.: *A reliable multicast framework for light-weight sessions and application level framing*. Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication, pp. 342–356. ACM Press, 1995. ISBN 0-89791-711-1.
4. D. Gelernter: *Generative communication in Linda*. ACM Transactions on Programming Languages and Systems, vol. 7(1): pp. 80–112, 1985. ISSN 0164-0925.
5. S. Hupfer: *Melinda: Linda with Multiple Tuple Spaces*. Tech. Rep. Technical Report YALEU/DCS/RR-766, Department of Computer Science, Yale University, New Haven, Connecticut, U.S., Feb. 1990.
6. B. Johanson: Application Coordination Infrastructure for Ubiquitous Computing Rooms. Ph.D. thesis, Stanford University, Dec. 2003.
7. B. Johanson, A. Fox and T. Winograd: *The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms*. IEEE Pervasive Computing Magazine, vol. 1(2), Apr. 2002.
8. T. Kindberg, J. Barton et al.: *People, Places, Things: Web Presence for the Real World*. Proceedings of 3rd IEEE Workshop of Mobile Computing Systems and Applications (WMCSA 2000), pp. 19–30. IEEE Computer Society, Monterey, California, Dec. 2000.
9. J. S. Leichter: Shared Tuple Memories, Shared Memories, Buses and LAN's – Linda Implementations across the Spectrum of Connectivity. Ph.D. thesis, Department of Computer Science, Yale University, New Haven, Connecticut, U.S., Jul. 1989.
10. G. P. Picco, A. L. Murphy and G.-C. Roman: *LIME: Linda meets mobility*. Proceedings of the 21st international conference on Software engineering, pp. 368–377. IEEE Computer Society Press, 1999. ISBN 1-58113-074-0.
11. M. Román, C. Hess et al.: *A Middleware Infrastructure for Active Spaces*. IEEE Pervasive Computing, vol. 1(4): pp. 74–83.
12. A. I. T. Rowstron and A. M. Wood: *Solving the Linda multiple rd problem using the copy-collect primitive*. P. Ciancarini and C. Hankin, eds., Proceedings of Coordination'96, Coordination Languages and Models, vol. 1061 of *Lecture Notes in Computer Science*, pp. 357–367. Springer-Verlag, 1996.
13. A. I. T. Rowstron and A. M. Wood: *Bonita: A set of tuple space primitives for distributed coordination*. Proceedings of the 30th Hawaii International Conference on System Sciences, p. 379. IEEE Computer Society, 1997. ISBN 0-8186-7743-0.
14. P. Wyckoff, S. W. McLaughry, T. J. Lehman and D. A. Ford: *T Spaces*. IBM Systems Journal, vol. 37(3): pp. 454–474, 1998.
15. A. Xu and B. Liskov: *A design for a fault-tolerant, distributed implementation of Linda*. Nineteenth International Symposium on Fault-Tolerant Computing (FTCS-19), pp. 199–206. Jun. 1989.