An Asynchronous Distributed Systems Platform For Heterogeneous Environments

Nigel Davies, Adrian Friday, Stephen P. Wade and Gordon S. Blair

Distributed Multimedia Research Group,
Computing Department,
Lancaster University,
Bailrigg,
Lancaster, UK

telephone: +44 (0)1524 594337
e-mail: nigel@comp.lancs.ac.uk

ABSTRACT

Since its introduction over a decade ago the tuple space paradigm has attracted interest from the distributed systems community. Despite being developed for shared memory parallel architectures, the simplicity and elegance of the model has led researchers to attempt to realise it in loosely coupled distributed environments. This paper argues that these attempts have largely failed due to their selection of inappropriate target domains, lack of multicast support and failure to operate in heterogeneous environments. We present the design and implementation of a new tuple space platform engineered using IP multicast. The platform is designed to support complex distributed applications such as groupware and mobile applications, operates over a range of end-systems and networks and offers performance comparable to existing RPC based platforms even in tests designed to benefit the RPC paradigm.

1. Introduction

Since its introduction in 1985 the tuple space paradigm [Gelernter,85a] has attracted interest from the distributed systems community. Despite being developed for shared memory parallel architectures the simplicity and elegance of the model has led researchers to attempt to realise it in loosely coupled distributed environments [Xu,89], [Pinakis,93], [Douglas,95].

These attempts have, in the opinion of the authors, for the most part failed. We justify such an assessment by noting that none of the major distributed systems platforms (CORBA, DCE, DCOM etc.) are based on the tuple space paradigm and that there have been relatively few papers on the topic in recent ICDCS/ICODP conferences. We attribute the failure of these initiatives to the following factors:

(i) *Performance Oriented Target Application Domains*

Tuple spaces were designed as a paradigm for parallel programming. As a result, previous platforms have typically been used to implement parallel algorithms which are often assessed largely on performance. This in turn places great significance on the performance of the tuple space platform, effectively condemning many implementations before they have had a chance to evolve in a way which would be normal for new enabling technologies.

(ii) *Reliance on Unicast Protocols*

Until the advent of multicast IP and the development of application level framing techniques in SRM [Floyd,95] distributed tuple space platforms were forced to rely on unicast protocols. This led to considerable overheads being incurred when attempting to coordinate the activities of multiple distributed users of the platform. As a consequence, most distributed tuple space implementations effectively became centralised implementations with portions of the overall state cached on different hosts (e.g. [Pinakis,93], [Douglas,95]).

(iii) *Homogeneous Network and Processing Environment*

Platforms developed in the early 1990s tended to be designed to operate in relatively homogeneous environments. In particular, most were implemented in local area environments consisting of workstations and Ethernet networking. Such environments were also used to develop RPC based platforms and proved ideal for their requirements. In particular, the low round trip times and the lack of disconnections now associated with modern mobile environments failed to fully highlight the RPC paradigm's shortcomings.

In our work to develop a tuple space based platform we have been motivated by a different set of goals and have operated in a different environment. This has, we believe, enabled us to avoid many of the pitfalls associated with previous distributed implementations.

(i) *Distributed Applications Focus*

We have designed our platform to support distributed applications with a particular emphasis on those which feature complex patterns of interaction (e.g. groupware applications) or require additional support services from their distributed systems platform (e.g. mobile applications). The suitability of the paradigm for this type of application was recently noted by members of the original Linda development team [Bjornson,97].

(ii) *Extensive Use of Multicast*

We have constructed our platform making extensive use of IP multicast. This enables us to provide a fully distributed implementation of our platform which, in environments supporting hardware multicast, provides performance comparable to RPC based platforms.

(iii) *Heterogeneous Network and Processing Environments*

We have assumed from the outset that the platform will be required to operate over a wide range of networks including high-speed fixed networks and low-bandwidth mobile networks. We have ported our platform to a wide range of end-systems including Linux, Solaris, SunOS, Windows 95 and Windows NT. Interworking between applications on these systems is supported.

The reminder of the paper describes the design, implementation and evaluation of our distributed tuple space based platform called L^{2}imbo. Section 2 provides the necessary background for the paper including a brief overview of the tuple space paradigm and a critique of other distributed implementations. Section 3 describes the design of the L^{2}imbo computational model, API and engineering infrastructure. Section 4 discusses the implementation of L^{2}imbo and presents the results of our evaluation work. Section 5 contains our concluding remarks and outlines plans for future work in this area.

2. The Tuple Space Paradigm and Distributed Systems

2.1. Paradigm Overview

Tuples are typed data structures, each of which consists of a collection of typed data fields. Each field is either termed an actual, if it contains a value, or a formal, if it is undefined. Collections of (possibly identical) tuples are placed in objects called tuple spaces which can be shared between processes. Any process using a tuple space has access to all the tuples it contains and can dynamically insert or remove tuples. Tuples are persistent and can not be altered while they reside inside a tuple space: they must be explicitly withdrawn then later re-inserted in order to effect changes [Gelernter,85b].

In tuple space systems, inter-process communications are conducted exclusively through the tuple space [Gelernter,85a]. By default, such communications are anonymous but directed communications (producing tuples for an identified consumer process) can be achieved by encapsulating destination information in tuples. Since tuple spaces contain persistent tuple objects, rather than messages, communication is supported across both space and time [Bjornson,91]. This property of the paradigm is of particular interest in environments where communications QoS is highly variable and systems may have to survive frequent periods of disconnection (e.g. mobile environments).

2.2. Existing Distributed Implementations

A number of distributed tuple space implementations have been developed over recent years. Of particular interest is the work of Pinakis [Pinakis,93] on developing a distributed operating system microkernel based on Linda. Pinakis' system uses a client-server architecture for communication between Linda client processes and the system server processes which implement tuple spaces. In more detail, each node of a distributed system which is participating in a tuple space maintains two servers, the first for tuple types and the second for tuples. Each tuple server manages a distinct portion of the recognised tuple types (hence all tuples of a given type are stored on a single node). To deposit a tuple in, read or withdraw a tuple from the distributed tuple space clients must determine which node manages tuples of the appropriate type and forward their requests to the tuple server on that host. This architecture enables Linda semantics, such as the unique withdrawal of a tuple using `in`, to be easily enforced since the system maintains only a single copy of each tuple. However, the performance and scalability of such a system is clearly limited. For instance, centralising each type at a single node can, depending on the type configuration, have the effect of serialising access to the tuple space. Furthermore, the system is unable to survive network partitions and hence is unsuitable for mobile systems. Finally, since each client must contact the server individually there are no benefits to clients of carrying out `rd` operations as compared to `in` operations despite the difference in their semantics. As a consequence, the system is likely to perform poorly when supporting applications in which the same tuple is `rd` by multiple hosts (e.g. groupware applications).

A similar approach to tuple distribution is described by Douglas et al [Douglas,95]. In their model, tuple spaces are implemented by, and distributed across, a number of tuple space managers. In common with the design proposed by Pinakis, each tuple is stored only at a single manager location. However, rather than assigning each tuple space manager a number of tuple types, a pair of hashing algorithms are used to determine the destined manager for each tuple individually. The hash algorithms ensure a reasonably balanced distribution of tuples across tuple space managers, provided tuples of differing types and initial field values are present in the system. While network partitions and server crashes are marginally less disastrous in this architecture it suffers from the same basic problems as the solution proposed by Pinakis.

Almost all other distributed Linda implementations have primarily concerned themselves with providing fault tolerance in the face of host or processor failures in networked or multi-processor environments respectively. For instance, a design for making the Linda tuple spaces stable by replicating tuples across the member nodes was proposed by researchers at MIT [Xu,89]. In this model, `out`, `rd` and `in` operations cause messages to be broadcast to all tuple space replicas; `out` distributes tuples which are cached by each replica, while `rd` and `in` transmit templates for which matches are sought. A `rd` operation blocks the client until a reply containing all matches found at a particular replica is received and, thus, a suitable tuple is available. `in` operations are less trivial since the model requires *locks* on all matching tuples to be obtained from all replicas. If a matching tuple could not be found, or the acquisition of locks for all replicas of suitable tuples could not be secured, messages are transmitted to release all locks and the matching process restarted. Where suitable matches are available, the client's node selects one, informing all replicas of the choice; they in turn delete the tuple from their cache and release any locks on those not chosen. Considering tuple space matching is non-deterministic [Gelernter,85a] this causes undue serialisation of accesses to tuples of identical type signatures. A view change algorithm adjusts the model to tackle processor failures and network partitions by restricting further tuple space accesses to clients in the majority partition. Once again, this enforces unnecessary restrictions on the use of tuples by clients in the minority partition.

In the following sections we present the design and implementation of our tuple-space based platform, L^{2}imbo, which addresses the main shortcomings identified above. In particular, L^2imbo offers comprehensive support for mobile and groupware applications and uses IP multicast to provide a good level of performance.

3. The L^2imbo Platform

3.1. Computational Model and API

Our distributed systems platform provides the same basic API and features as the original Linda model [Gelernter,85a] but includes a number of key extensions:

(i) *Extensions to the API to support asynchronous operations*

We have extended the L^2imbo API using operations based on the Bonita primitives proposed in [Rowstron,97]. These enable clients on each host to access tuple spaces asynchronously by replacing the `in` and `rd` operations by two separate operations: one to initiate the operation and one to collect the results at a later point. A further operator allows clients to poll their tuple space interface asking whether the results for a previous request are available.

(ii) *Multiple local, distributed and centralised tuple spaces which may be specialised for application level requirements such as consistency or security*

L^2imbo allows the creation of multiple tuple spaces to address issues of performance, partitioning and scale. L^2imbo supports three basic classes of tuple space for different application requirements, local (private to that host), distributed (cached at one or more hosts) and centralised (maintained on a single host but accessible from elsewhere). Tuple spaces may be linked using bridging agents which copy tuples between tuple spaces based on factors such as tuple types and QoS parameters.

•*(iii)* *System agents which provide services such as tuple space creation, tuple type management, propagating tuples between tuple spaces and QoS monitoring.*

All system operations are provided by system agents which the clients interact with using standard tuple space operations.

In addition, we have added support for deadline based operations and tuple typing to the model. Details of this work can be found in [Blair,97], [Davies,97a] and [Davies,97b].

3.2. Engineering Design and Implementation

The current prototype L^2imbo platform consists of a small stub library which is linked with each application process and a daemon process, an instance of which executes on each host.

3.2.1. The L^2imbo Daemon Process

The L^2imbo daemon process executes on each host and consists of four layers: the interface to the API, the tuple space protocols, the network scheduler and a number of network interface modules (see figure 1). The uppermost layer provides the interface to the API and is responsible for all communications between the L^2imbo daemon and client applications on the same host. By centralising application accesses to all tuple spaces through a single process on each host local matching is simplified and the platform gains an overall picture of the demands on the available network (or networks). This enables the platform to manage congestion and load balancing more effectively, but incurs a performance penalty since each message involves additional local communications and a context switch. We return to this issue in section 4. Distributed tuple spaces are serviced by the DTS protocol layer which is described in the next sub-section.

The daemon has been specially designed so that the transport services remain independent of both tuple spaces and network technology. The network scheduler accepts protocol messages from higher layers and, based on associated priority and deadline QoS parameters, determines the order in which they are transmitted. Within each priority, messages are scheduled in earliest deadline first order. Messages with the highest priority (smallest number) are considered most urgent and scheduled before those of successive priorities (even if a lower priority has an earlier deadline). This concept is based on previous work by Nieh on thread scheduling for continuous media [Nieh,95].

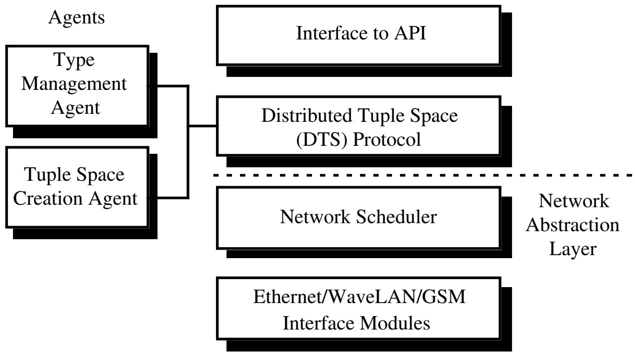

Figure 1: Structure of the L^2imbo daemon process

Each supported network has an interface module which presents a generic interface to the platform behind which details such as connection management and signalling are hidden. Packets ready for transmission are delivered to an appropriate network interface module by the network scheduler.

3.2.2. The DTS Protocol

The development of the distributed tuple space protocol (DTS) component of our platform has been motivated by the need to maximise the scalability and availability of our tuple spaces. There are two key factors which affect the performance of such a protocol. Firstly, the protocol must not serialise operations through a single point or it will not scale. Secondly, traditional consistency mechanisms based on acknowledgements or token passing must be avoided since these will degrade unacceptably through artefacts such as acknowledgement implosion or the protracted latency of contacting all of the group members.

To tackle these problems we have taken advantage of the recent deployment of IP multicast together with application level framing concepts borrowed from work on wb [Floyd,95] and Jetfile [Grönvall,96].

The DTS protocol consists of nine distinct protocol messages which are used in conjunction with a cache of tuples and anti-tuples held on each host. Collectively, these caches represent the state of the tuple space. The messages are used to ensure timely propagation of tuples and anti-tuples between caches. An overview of the operation of the protocol is given in table 1 and more details can be found in [Davies,97b].

In order to ensure tuple uniqueness the protocol introduces the concept of tuple ownership. Tuples can only be removed from the tuple space by their owner. The initial owner of a tuple is normally the host which creates it, although if a host generates a tuple for which it knows a matching IN request has been received it can nominate the originator of that request as the owner. By observing sequences of interactions, a tuple which is likely to be consumed by the originator of the last tuple can have that host nominated as its owner. This allows RPC-like semantics to be modelled efficiently. The protocol uses CHOWN_REQ and CHOWN_ACK messages to explicitly change tuple ownership where hosts need to withdraw tuples they do not own.

Since `rd` operations are used to copy tuples non-destructively, they need not be concerned with tuple ownership and hence can be satisfied more quickly and efficiently as they require less communication. In particular, groupware applications in which the same tuple is obtained using `rd` by a number of hosts are highly efficient.

Hosts detect that tuples are missing from their local cache by observing the tuple identifiers in all messages received. REPAIR_REQ messages are automatically issued to request the retransmission of missing tuples and thus move closer to eventual global consistency. Members who have the required tuples multicast REPAIR_ACK messages subject to a backoff proportional to their distance from the REPAIR_REQ originator. This ensures the closest cache capable of satisfying the request responds first. If a host snoops an identical REPAIR_ACK message from another host, it avoids transmitting a response itself thus preventing acknowledgement implosion.

ACCESS and DELETE messages are of comparatively low priority since they are used only by other hosts to detect missing tuples or prevent the use of stale tuples by `rd` requests. The earlier these messages are transmitted, the faster the independent views of a tuple space converge. However, as their delay does not alter the semantics of the tuple space, we can batch ACCESS's and DELETE's with other protocol traffic to reduce overall communication overhead.

Message	Format and Actions
`OUT`	`[tuple_id, owner_id, type, tuple]` If we already have information about this tuple ensure that the ownership details are up-to-date. Otherwise add the tuple to our queue, satisfy any matching `RD` requests made on the local host (transmitting an `ACCESS` message for each one), then look for a matching `IN` request. If we find one, check whether we are the current owner, transmitting a `DELETE` or `CHOWN_REQ` as appropriate.
`IN`	`[client_id, request_id, type, spec]` Should we have a matching tuple, multicast an appropriate `OUT` message, otherwise add the `IN` request to our queue.
`RD`	`[type, spec]` Check if we have a matching tuple and if so multicast an `OUT` message.
`CHOWN_REQ`	`[tuple_id, client_id]` First, check to see if we know about this tuple. If we don't, transmit a `REPAIR_REQ`. Should we know the tuple has been deleted, multicast a `DELETE`. If we own the tuple, we can transmit a `CHOWN_ACK` nominating the originator of the `CHOWN_REQ` as the new owner, otherwise we send a `CHOWN_ACK` stating who we understand to be the current owner.
`CHOWN_ACK`	`[tuple_id, owner_id]` If we know about this tuple update its ownership. If we are the new owner and have a pending local `IN` which matches, service the request and multicast a `DELETE` message.
`DELETE`	`[tuple_id, request_id]` Mark the unique tuple id as having been deleted and ensure both the tuple and the `IN` request it satisfied are removed from our cache.
`ACCESS`	`[tuple_id]` If we don't know about this tuple, transmit a `REPAIR_REQ`, otherwise if we know it to have been deleted, multicast a `DELETE` message.
`REPAIR_REQ`	`[tuple_id]` If we have this tuple multicast a `REPAIR_ACK`
`REPAIR_ACK`	`[tuple_id, owner_id, type, tuple]` Queue any unknown tuples.

Table 1: Distributed Tuple Space (DTS) protocol messages

Hosts are free to connect and disconnect from the multicast group (and/or network) at will. Mobile hosts connect through a proxy (such as that proposed for mobile IP and IPv6) which operates a cache on behalf of disconnected clients.

4. Evaluation

We have built an initial version of the platform (approximately 3,000 lines of C for the daemon process and 500 lines for the API) which runs on SunOS 4.1.4 (MULTICAST 4.1.4), Solaris 2.5, Linux 2.0.30 and Windows NT 4.0. We have used the platform to build a number of applications including a text based conferencing tool, a collaborative geographical information system (GIS) and a group coordination service for a low bit rate video conferencing tool. We believe that the inherent time and space decoupling offered by the model permits applications a far more flexible interaction mechanism than traditional RPC semantics. Applications can transparently adopt synchronous or asynchronous styles of interaction as applications or network conditions dictate, facilitating operation in mobile environments.

We have compared the performance of our platform against both the ANSAware distributed systems platform (version 4.1) and raw BSD sockets. Our test suite consisted of three separate pairs of client and server processes which carry out timed RPC interactions consisting of an *n* byte payload and null response. An RPC is modelled in the tuple space by the exchange of two tuples with types *request* and *response* respectively. The test software was compiled on SparcStation 1 workstations networked with a moderately loaded 10Mbps Ethernet. To isolate the additional overhead we incur for splitting the L^2imbo platform into separate processes (daemon and client libraries), we have run tests for both an optimised form in which the two processes are linked into a single executable and unoptimised (separate processes) forms of the client and server. The results are summarised in table 1.

Payload (bytes)	Sockets (UDP)	ANSAware 4.1 (REX)	Limbo DTS (linked)	Limbo DTS (separate processes)
256	2.98	7.10	6.53	12.58
512	3.45	10.48	7.20	13.47
1024	3.93	11.17	8.64	15.10
2048	5.85	13.14	11.97	20.28
4096	9.46	21.14	18.06	28.26
8192	15.83	34.83	29.93	44.82

Table 1: Comparison of relative performance on SunOS

The figures show that in the optimised form Limbo outperforms ANSAware RPC in all cases. However, the overhead of the context switch and local communication required in the standard L^2imbo prototype has a significant impact on the figures. Reducing to a minimum the overheads associated with exchanging messages between the application stubs and the daemon process is clearly an important factor in improving the performance of L^2imbo.

In addition, published figures for the Chorus Systèmes COOL ORB [Chorus,96] on the Linux platform suggest that the performance of L^2imbo is comparable to other distributed systems platforms. The COOL benchmark report quotes 3.8 ms for a basic request exchange of 1000 bytes in each direction. On a similar specification Linux platform the linked version of L^2imbo takes 4.4 ms to perform this same test (averaged over 1000 interactions). Furthermore, for interactions of 100 bytes in each direction, COOL is quoted as taking 2.6 ms, whereas the optimised form of Limbo takes just 1.9 ms.

In considering these figures it is important to note that the test case demonstrates directed communication. In the conventional distributed systems platform the timing information is taken *after* an initial process of binding and thus represents the best possible case for these platforms. In the case of L^2imbo however, the test case represents a worst case scenario; the tuples are being rapidly inserted and removed from the tuple space and the overhead associated with matching is not strictly necessary since only two well known processes are communicating.

5. Concluding Remarks

The tuple-space paradigm has a number of clear advantages over the RPC paradigm for writing complex distributed applications. In particular, the paradigm is ideally suited to applications which make use of multicast (e.g. collaborative applications) and applications which communicate over both time and space (e.g. mobile applications subject to periods of disconnection). We have described a new distributed systems platform called L^2imbo which is based on the tuple-space paradigm.

Existing distributed platforms based on the tuple-space paradigm have suffered from a number of drawbacks. Most critically, previous systems have been unable to provide performance comparable to RPC based platforms in typical operational environments. In contrast, L^2imbo uses a combination of local tuple caches and a protocol based on IP multicast to achieve performance for directed communication in line with a number of RPC-based distributed systems platforms. For multicast and undirected communications L^2imbo provides better performance than these platforms. The L^2imbo platform will shortly be available to the research community from:

www.comp.lancs.ac.uk/computing/research/mpg/most/limbo/

The platform currently executes on Linux, Solaris, SunOS, Windows 95 and Windows NT platforms. Our future work in this area will focus on the provision of system agents to support the L^{2}imbo platform and in particular on the development of proxies to support mobile operation.

6. References

[Bjornson,91] R. Bjornson, N. Carriero, D. Gelernter, T. Mattson, D. Kaminsky and A. Sherman, "Experience with Linda", *Technical Report YALEU/DCS/TR-866*, Department of Computer Science, Yale University, New Haven, Connecticut, U.S., August 1991.

[Bjornson,97] R. Bjornson, N. Carriero and D. Gelernter, "From Weaving Threads to Untangling the Web: A View of Coordination from Linda's Perspective", *Proceedings of the 2nd International Conference on Coordination Languages and Models (Coordination '97)*, Berlin, Germany, 1-3 September, 1997, pp1-17.

[Blair,97] G. S. Blair, N. Davies, A. Friday and S. P. Wade, "Quality of Service Support in a Mobile Environment: An Approach Based on Tuple Spaces", *Proceedings of the 5th IFIP International Workshop on Quality of Service (IWQoS '97) - Building QoS into Distributed Systems*, Columbia University, New York, U.S., 21-23 May 1997, pp37-48.

[Chorus,96] Chorus Systèmes, "CHORUS/COOL-ORB Programmer's Guide", *Technical Report CS/TR-96-2.1*, Chorus Systèmes, 1996.

[Davies,97a] N. Davies, S. P. Wade, A. Friday and G. S. Blair, "Limbo: A Tuple Space Based Platform for Adaptive Mobile Applications", *Proceedings of the International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP '97)*, Toronto, Canada, 27-30 May 1997, pp291-302.

[Davies,97b] N. Davies, A. Friday, S. P. Wade and G. S. Blair, "Using Tuple Spaces for Adaptive Mobile Computing", *To appear in ACM Mobile Networks and Applications (MONET).*

[Douglas,95] A. Douglas, A. Wood and A. Rowstron, "Linda Implementation Revisited", *Transputer and Occam Developments*, IOS Press, 1995, pp125-138.

[Floyd,95] S. Floyd, V. Jacobson, S. McCanne, C. Liu and L. Zhang, "A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing", *Proceeedings of ACM SIGCOMM '95*, Cambridge, Massachusetts, U.S., August 1995, ACM Press, pp342-356.

[Gelernter,85a] D. Gelernter, "Generative Communication in Linda", *ACM Transactions on Programming Languages and Systems,* Volume 7, Number 1, January 1985, pp80-112.

[Grönvall,96] B. Grönvall, I. Marsh and S. Pink, "A Multicast-Based Distributed File System for the Internet", *Proceedings of the 7th ACM SIGOPS European Workshop*, Connemara, Ireland, 2-4 September 1996, ACM Press.

[Pinakis,93] J. Pinakis, "Using Linda as the Basis of an Operating System Microkernel", *Ph.D. Thesis*, Department of Computer Science, University of Western Australia, Nedlands, WA 6009, Australia, August 1993.

[Rowstron,97] A. I. T. Rowstron and A. M. Wood, "Bonita: A Set of Tuple Space Primitives for Distributed Coordinartion", *Proceedings of the 30th Annual Hawaii International Conference on System Sciences*, Volume 1, IEEE CS Press, 1997, pp379-388, 1997.

[Xu,89] A. Xu and B. Liskov, "A Design for a Fault-Tolerant, Distributed Implementation of Linda", *Proceedings of the 19th International Symposium on Fault-Tolerant Computing*, June 1989, pp199-206.