L^{2}imbo: A distributed systems platform for mobile computing

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

Distributed Multimedia Research Group, Department of Computing, Lancaster University, Bailrigg, Lancaster, LA1 4YR, U.K.

Mobile computing environments increasingly consist of a range of supporting technologies offering a diverse set of capabilities to applications and end-systems. Such environments are characterised by sudden and dramatic changes in the quality-of-service (QoS) available to applications and users. Recent work has shown that distributed systems platforms can assist applications to take advantage of these changes in QoS and, more specifically, facilitate applications to adapt to their environment. However, the current state-of-the-art in these platforms reflect their fixed network origins through their choice of synchronous connection-oriented communications paradigms. In this paper we argue that these paradigms are not well suited to operation in the emerging mobile environments. Furthermore, we offer an alternative programming paradigm based on tuple spaces which, we believe, offers a number of benefits within a mobile context. The paper presents the design, implementation and evaluation of a new platform based on this paradigm.

1. Introduction

Mobile computing environments are characterised by significant and rapid changes in their supporting infrastructure and, in particular, in the quality-of-service (QoS) available from their underlying communications channels. Previous research [1,2] has demonstrated that in order to operate effectively in mobile environments applications are required to adapt in response to these changes. Such applications are termed *adaptive applications*.

Adaptive applications require distributed systems support, and a number of platforms have recently been developed which address this requirement. Examples include Mobile DCE [3], the MOST platform [4] and the Rover Toolkit [5]. These mobile platforms attempt to provide application programmers with traditional computational models and communications semantics consistent with those normally found in platforms designed for fixed networks. In particular, the three major mobile distributed systems platforms all implement RPC semantics with (to a greater or lesser extent) additional interfaces allowing applications to monitor and adapt to changes in QoS. Clearly, procedure call semantics are difficult to provide during periods when mobile hosts are experiencing very low levels of communications QoS or during periods of disconnected operation. To address this problem the platforms all include support for buffering remote procedure calls during periods of disconnection ready for transmission when the network QoS improves and facilities for re-binding clients to new (possibly proxy) services during network partitions.

In this paper we argue that the procedure call paradigm has a number of shortcomings when used in the context of mobile environments and suggest an alternative paradigm based on *tuples* and *tuple spaces* [6]. The tuple space paradigm has been widely used in the parallel computing community but there has, to our knowledge, been no work on applying the paradigm in mobile environments. The use of tuple spaces for mobile computing was considered by Schilit in his thesis [7] but they were dismissed as being "*... not designed to manage hosts over slow links, and …not concerned with the problems of fault tolerance in a partially connected network*". We would argue that while this is clearly true of the original tuple space platform, Linda [6] (which was designed for shared memory multiprocessors), the same criticism could be levelled against any distributed systems platform which had been developed before the impact of mobile computing had been considered (e.g. DCE). In contrast, we believe that the inherent temporal and spatial de-coupling offered by the tuple space paradigm is more suited to the partial connection experienced by mobile hosts than existing synchronous communication paradigms. In the remainder of this paper, we consider the design of a tuple space based platform called

L^2imbo. The platform, API and a distributed implementation of the prototype are considered.

Section 2 presents an analysis of the characteristics of mobile environments and highlights the role of distributed systems platforms in supporting application adaptation. Section 3 considers the three foremost mobile distributed systems platforms and argues that the synchronous communications model which underpins these platforms is not well suited to use in a mobile environment. Section 4 then presents the design of our new platform, L^2imbo, which is based on an asynchronous programming paradigm and includes integrated support for QoS. A distributed implementation of the L^2imbo paradigm is described in section 5. A performance comparison of the platform prototype with respect to more well known distributed systems platforms is presented in section 6. Finally, section 7 contains our concluding remarks.

2. Supporting mobile computing

Currently, mobile hosts are able to make use of a wide variety of communications technologies to attain connectivity. These technologies include wide, metropolitan and local area wireless networks together with more traditional fixed network technologies. Increasingly, network technologies are becoming integrated to offer seamless connectivity to mobile hosts through a system of network overlays [8]. Each of these supporting technologies offer end-systems a unique signature of characteristics (QoS), the most commonly considered, and easily visualised of which, is bandwidth. In simple terms, a typical fixed network (such as Ethernet) will offer several orders of magnitude greater bandwidth than the average mobile channel (e.g. GSM). In addition, networks may be characterised in innumerable other ways including access time, bit error rates, error control strategies, orientation (connection orientation versus packet mode) and, of increasing importance, functionality. This last category reflects the fact that many networking technologies now offer a range of services to end-systems. For example, the TETRA system offers configurable variable bit rate channels with multiple levels of forward error protection which may be selected by the client. Many of these abilities are shared by cordless systems based on DECT, wired systems such as ISDN and future wireless cellular developments including the emerging GSM High Speed Circuit Switched Data (HSCSD) and General Packet Radio Services (GPRS). In addition, a number of reservation based protocols are available for fixed networks which alter the networks perceived QoS [9,10].

Given the implications for applications and end-users of the characteristics and potential services offered by such a range of networking technologies, there is an increasing requirement for QoS information and control throughout end-system software [8,11,12]. This requirement for information provision is in contrast to current distributed systems platforms which justify their existence primarily through the provision of distribution transparencies. For instance, the provision of location and access transparencies greatly simplify the development of distributed applications. Such an approach has proved highly successful for facilitating interworking in heterogeneous distributed environments and works adequately while the characteristics of the underlying system are relatively static and assumptions concerning levels of service can be made by applications and end-users. However, as stated in the previous section, change is a fundamental characteristic of mobile environments. Employing the same approach in environments such as these, i.e. hiding the environmental changes with layers of transparency, prohibits adaptation, the requirement for which has been presented in [8,11,12].

Existing research has postulated that API level QoS abstractions based on communications bindings [13,14] are sufficient to enable platforms and applications to be made aware of, and control, their supporting environment. Indeed, systems based on the concept of bindings have shown that such abstractions provide a natural and easily visualised method of managing network QoS in mobile applications [15]. However, tying system QoS management to inter-host communication is not ideal. In particular, there are a number of QoS parameters which may govern system behaviour that are not linked to communication. An example of one of the most significant of these parameters is power or, more specifically, the charge in the system's batteries and power consumption of various system components (including network interfaces, on-board hardware, the display, PC card devices and the hard disk). While the power consumption associated with using the network interface (or network interfaces) is clearly linked with communication, the remaining components' consumption is likely to be largely independent of any ongoing communications.

The close liaison between communications and QoS implies that, *at the API level*, communications must be established before QoS information can be obtained (note that this does not necessarily imply that engineering level bindings must also be created). This approach makes writing distributed applications difficult since in order to determine the QoS available to a range of services the application must first establish bindings to each service in order to gain an appropriate handle for obtaining QoS information. In our opinion, future distributed systems platforms will be required to provide more general mechanisms to facilitate adaptation. In more detail, such platforms must collate and manage QoS information from a wide range of sources (both communications and end-system based) for presentation to higher layers, enabling feedback and control throughout the architecture. Examples of such QoS information include power availability, physical location, device proximity and communications capabilities and costs.

3. Current mobile distributed systems platforms

To date there have been three significant research efforts aimed at producing general purpose distributed systems platforms for mobile environments. In the following sections we briefly review each of the resulting systems and then discuss their commonalities and shortcomings.

3.1. Mobile DCE

The Mobile DCE initiative at the University of Technology, Dresden aims to augment a standard DCE platform with new features for operation in a mobile environment.

The overall system architecture is based on the concept of domains. These are logical groupings of machines with shared resources managed by a domain manager. Mobile clients move between domains and hence have access to different resources. Manager processes on each client interact with the domain managers and utilise a matrix specifying resource characteristics in order to ensure service provision as the client changes domains. In more detail, as clients cross domain boundaries their managers decide (for each service) whether to continue remote access to a service in the original domain, to re-bind to a new service in the destination domain or, during periods of disconnection, to emulate the service and replay messages when connectivity is restored.

Mobile DCE has been implemented under the Windows/NT operating system and a number of applications have been developed including mobile e-mail. The use of the industry standard DCE/Microsoft RPC protocol allows the platform to be integrated with existing distributed applications.

3.2. The MOST platform

Lancaster University's MOST platform provides support for adaptive mobile applications within an Open Distributed Processing (ODP) [16] based framework. The platform augments an existing ODP compatible platform called ANSAware [17] with new services, protocols and API calls. In particular, the platform incorporates a new protocol called QEX [15] which is able to adapt to changes in the QoS of its underlying communications infrastructure and pass this information on to interested client applications.

The QEX protocol is layered above a low-level service called S-UDP which provides dial-up UDP connections over GSM. S-UDP and QEX both allow messages to be tagged with deadlines and messages from the mobile host to the fixed network are sent in earliest-deadline-first order. The system uses the message deadlines to determine when to establish and break connections. Furthermore, messages can be buffered during periods of disconnection until either they are sent or their deadlines expire (in which case an exception is raised at the client).

The MOST platform has been implemented on Sun workstations and notebook PCs running a variety of flavours of UNIX and using a range of communications technologies including GSM. The platform has been used to support a wide range of mobile applications including e-mail, a collaborative geographic information system and a job dispatch application for field engineers.

3.3. Rover

The Rover toolkit from M.I.T. is designed to support the development of mobile applications. This support is based on the twin notions of relocatable data objects and queued remote procedure calls (QRPCs). In essence, the platform allows the creation of data objects with well defined interfaces which can be migrated at run-time between the mobile client and servers on the fixed network. This allows decisions regarding application configuration and the client-server computation trade-off to be made (and re-evaluated) at run-time as the network QoS and resource availability change.

Communication between objects is carried out using the toolkit's QRPC protocol. In addition to being able to re-bind to objects which have migrated, QRPC also provides support for periods of disconnection by buffering messages destined for remote sites until network connectivity is restored.

A number of applications have been ported to the Rover toolkit including a web browser and an e-mail application. However, unlike both MOST and Mobile DCE, Rover is not based on an existing standard and applications must be re-engineered to operate in a mobile environment.

3.4. Discussion

All of the above platforms offer mobile clients connection-oriented RPC-based communications with associated QoS support. The implementations of the communications services all relax the synchronous nature of RPC interactions by allowing messages to be buffered (Mobile DCE), delayed (MOST) or queued (Rover). However, the programming model presented to application writers is still essentially synchronous in nature. Indeed, all of the platforms attempt to maintain RPC semantics in the face of variations in network connectivity. Furthermore, all of the models are connection-oriented: clients select services to be used, bind to their interfaces and then invoke operations on these interfaces. As the network QoS and service availability change the platforms use a range of techniques in an attempt to maintain the illusion of connection-oriented communications. For example, in Mobile DCE the RPC protocol transparently re-binds clients to local proxy services during periods of disconnection. In all of the platforms hooks have been provided to enable application programmers to determine the QoS of the underlying network and hence construct applications which adapt to changes in this QoS.

Our experiences with developing and working with platforms of this type have led us to question the suitability of the synchronous paradigms on which they are based for use in a mobile environment. In particular, as network QoS degrades, providing a model of synchronous, connection-oriented communications becomes increasingly difficult. In addition, the emphasis placed on communications in these platforms has thus far prevented a general model of QoS monitoring and adaptation emerging (as discussed in section 2). More specifically, while explicitly modelling bindings (as in the MOST platform) provides a convenient interface for monitoring communications QoS, it does not provide a general mechanism for informing applications of changes in other QoS parameters (e.g. power availability). These changes must be propagated to clients using an alternative mechanism, e.g. operating system signals or environment variables as in [18]. However, as previously highlighted, the lack of a cohesive strategy for dealing with all forms of QoS information complicates the development of applications.

The remainder of this paper describes the design, implementation and evaluation of a new platform called L^{2}imbo which attempts to address the issues raised in this section. More specifically, the platform features an asynchronous, connectionless programming paradigm and a uniform architecture for QoS control and monitoring.

4. The L^{2}imbo architecture

4.1. The tuple space paradigm

The tuple space paradigm has been extensively researched by the parallel programming community for over a decade. Tuples are typed data structures and each tuple 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 does not. Collections of (possibly identical) tuples exist in shared data objects called tuple spaces. Tuples can be dynamically deposited in and removed from a tuple space, though they can not be altered while resident in it.

Changes can, however, be made to a tuple by withdrawing it from the tuple space, amending and then reinserting it [19]. Tuple spaces are shared between collections of processes, all of which have access to the tuples contained within.

In classic distributed environments processes communicate across virtual channels described by bindings and formed from pairs of endpoints, c.f. Chorus ports and UNIX BSD 4.3 sockets. The tuple space paradigm is fundamentally different because processes communicate exclusively through tuple space; this has been termed *generative communication* [6]. As processes no longer interact directly with one another, the implicit need for bindings is removed and inter-process communication can actually progress *anonymously*. It is, however, also possible to achieve *directed communications* whereby tuples are produced for an identified consumer process by encapsulating destination information in the tuples themselves. Several schemes have been proposed to achieve this, including an approach based on Amoeba-like ports [20]. Because tuple spaces contain persistent tuple objects, as opposed to messages, inter-process communication is supported *across time as well as space* [21].

The tuple space paradigm was conceived by researchers at Yale [6] and embodied in a coordination language called Linda. Linda is not a standalone computational language, instead Linda operators are embedded in host computational languages (e.g. C or Pascal). The original Linda model defines four basic operators :

- `out` inserts a tuple, composed of an arbitrary mix of actual and formal fields, into a tuple space. This tuple becomes visible to all processes with access to that tuple space.

- `in` extracts a tuple from a tuple space, with its argument acting as the template against which to match. Actuals match tuple fields if they are of equal type and value; formals match if their field types are equal. If all corresponding fields of a tuple match the template 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 until a suitable tuple can be found.

- `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.

- `eval` is similar to `out`, except it creates *active* rather than *passive* tuples. The tuple is active because separate processes are spawned to evaluate each of its fields. The tuple subsequently evolves into a passive tuple resident in the tuple space.

Although not proposed in the original Linda model, many implementations support two further operators, `inp` and `rdp` [22]. These are non-blocking versions of `in` and `rd` which evaluate to boolean values indicating their success and, if the operation succeeds, assigns actuals to formals as before.

4.2. Platform overview

Our new platform, L^{2}imbo, is based on the Linda model but includes a number of significant extensions which address the specific requirements necessary for operation in mobile environments. In particular, our system incorporates the following key extensions:

- multiple tuple spaces which may be specialised to meet application level requirements, e.g. for consistency, security or performance.
- an explicit tuple type hierarchy with support for dynamic sub-typing.
- tuples with QoS attributes including delivery deadlines.
- a number of system agents that provide services for QoS monitoring, the creation of new tuple spaces and the propagation of tuples between tuple spaces.

In the following sections we explain each of these extensions in detail.

4.2.1. Multiple tuple spaces

The original Linda model was designed to support parallel programming on shared-memory multi-processor systems and features a single, global tuple space. Many recent models have proposed the introduction of multiple tuple spaces to address issues of performance, partitioning and scalability [23,24,25]. In particular, supporting multiple tuple spaces removes the need to conduct all operations through a single global tuple space on all machines: important for performance in a distributed environment and critical in an environment where communications links are costly and unreliable.

We provide a class of system agent which can create new tuple spaces which can be configured to meet application specific requirements [23]. For example, in addition to general purpose tuple spaces we allow the creation of tuple spaces with support for security (user authentication), persistence and tuple logging (for accountability in safety critical systems). Crucially, it is also possible to create a range of QoS-aware tuple spaces (as discussed below).

In order to create a new tuple space clients communicate with the appropriate system agents via a *universal tuple space* (UTS) using the sequence of operations shown in figure 1. Clients specify the characteristics of the desired tuple space and place a `create_tuple_space` request into a common tuple space. The appropriate system agent accesses this tuple, creates a tuple space with the required characteristics and then places a tuple of type `tuple_space` into the common tuple space. The fields in this tuple denote the actual characteristics of the new tuple space (which may be different to those requested in best-effort systems) and a handle through which clients can access the new space.

```
out(create_tuple_space, QoS, characteristics, request_id)
in(tuple_space, ?QoS, ?characteristics, ?handle, request_id)
```

Figure 1. Tuple space creation.

Clients can make use of the new tuple space by means of a `use` primitive which provides access to a previously created tuple space. This primitive communicates with a *membership agent* through the universal tuple space and returns a handle if the tuple space exists and certain other criteria are met. The precise criteria vary from tuple space to tuple space and can include checks on authentication and access control functions or relevant QoS management functions. At a later time, handles can be discarded by an agent using a `discard` primitive. An appropriate tuple is then placed in the universal tuple space so that the membership agent can take appropriate steps. Tuple spaces are destroyed by placing a tuple of type `terminate` into the tuple space. These tuples are picked up by system agents within the tuple spaces themselves and invoke a system function to gracefully shut-down the tuple space.

Note that this model can be applied recursively. It is possible to access a tuple space through the universal tuple space and then find that this tuple space has system agents supporting the creation and subsequent access to tuple spaces. This recursive structure provides a means of creating private worlds offering finer grain access control.

4.2.2. Tuple type hierarchy

In our model all tuples are associated with a given type. Typed tuples can be organised to form a hierarchy by establishing sub-typing relationships between them. The benefits of sub-typing in a distributed environment have been comprehensively investigated within the ODP community as part of their work on interface trading [26]. In this model sub-typing enables added flexibility when matching service offers to client requests. We hope to accrue similar benefits by supporting sub-typing in L^{2}imbo.

This scheme has a number of advantages over the notional typing found in many tuple space implementations. In addition to the usual benefits associated with type signatures, it allows for the use of sub-typing when attempting to match tuples to `in` requests. In more detail, `in` requests for a tuple of a given type can be matched with existing tuples of an equal or sub-type. The conversion between types and sub-types (simply a matter of omitting fields when returning the matching tuple) can be handled by the tuple space. Interestingly, sub-type relationships in our model are themselves simply tuples of a predefined system type which are placed in the tuple space like any other tuple. As such, tuples and thus type relationships may be established by any authorised user of the tuple space.

As an optimisation in L^{2}imbo, a tuple of one type is only regarded as a sub-type of another tuple if the common fields are present in the same order in the initial fields of the type. This optimisation greatly reduces the complexity of the tuple matching process.

4.2.3. QoS attributes

Existing mobile distributed systems platforms such as MOST allow deadlines to be associated with messages. This enables messages to be re-ordered by the system to make optimum use of the available network connectivity or buffered during periods of disconnection. In L^{2}imbo this is achieved by associating deadlines with tuples. In the case of a tuple which is the subject of an `out` operation the deadline refers to the time the tuple is allowed to reside in the tuple space before being deleted. In the case of tuples which are used as arguments to `in` or `rd` operations the deadline refers to the time for which the requests can block before timing out. Once again, this timing information can be used by the system to re-order messages.

Note that by supporting time-outs on tuple space operations we are able to avoid having to provide special support for `inp` and `rdp`, the non-blocking forms of `in` and `rd` found in other tuple space implementations [22]. Non-blocking operations cause particular problems for implementers of distributed tuple spaces. Essentially, one must be able to satisfy the assertion that when the non-blocking operation is actioned, no matching tuple is available anywhere in the tuple space, without having to lock the entire tuple space [27]. In distributed implementations this requirement forces many tuple space operations to progress in lock step, allowing every component site to have all the available tuples simultaneously. In our model, a timed operation is subject to the weaker assertion that no tuple could be obtained within the specified time.

4.2.4. System agents

All interaction between the system and applications is via tuple spaces. In addition to the tuple space creation agents discussed in section 4.2.1, tuple spaces may be augmented more generally by further *agents* which implement system and application functions. In more detail, agents reside above tuple spaces and interact with the tuple space, carrying out a particular computation. If required, agents can be written in a language supporting the creation of mobile code (e.g. Java or TCL) and migrated as necessary. Note that mobility of agents is naturally supported by the tuple space model; agents simply stop interacting with the tuple space, re-locate and then re-start their interaction. As an enhancement, agents are also generally *stateless*; all state is assumed to be in the tuple space. As tuples are persistent and globally available, it is then trivial to support replication of agents. In other words, there is no need for a consistency algorithm; this is directly provided by tuple spaces. Agents are classified as system agents, already provided in the environment, and application agents, introduced into the environment by the programmer. This distinction is however not particularly rigid. The application programmer is free to introduce additional system agents into the environment. This overall architecture is shown in figure 2.

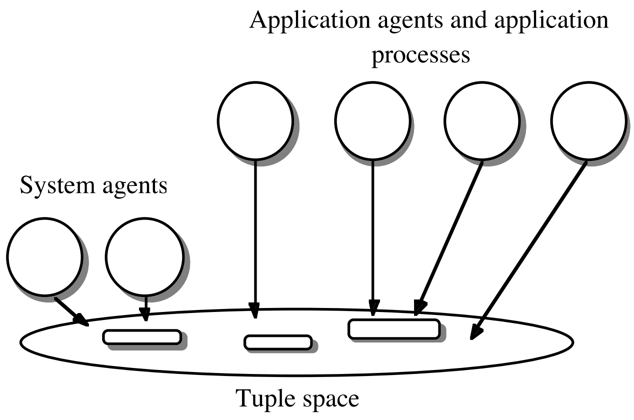

Figure 2. Tuple space agent interaction.

In the remainder of this section we consider the function of three of the most significant system agents defined in L^{2}imbo: Bridging agents, QoS monitoring agents and type management agents.

Bridging agents

Bridging agents provide the means of linking arbitrary tuple spaces and controlling the propagation of tuples between these spaces. In their simplest form bridging agents are processes which carry out repeated `rd` operations on one tuple space and then `out` the corresponding tuples into a second tuple space (with appropriate mechanisms to avoid the problems caused by the non-deterministic nature of the `rd` operation). However, bridging agents can also provide more intelligent tuple propagation based on a number of factors including tuple types and QoS parameters. For example, bridging agents can be configured to only propagate tuples or requests (generated as a result of `in`, `out` or `rd` operations) subject to a set of constraints. Bridging agents can also be used to provide gateways between specialised tuple spaces. For example, a bridging agent could be configured to carry out format conversions between homogeneous and heterogeneous tuple spaces or to act as a firewall to prevent the propagation of unauthenticated tuples to secure tuple spaces.

It is important to stress that tuple spaces usually span multiple hosts; bridging agents provide a mechanism for propagation of tuples *between tuple spaces* and are *not* usually required for the propagation of tuples between separate hosts: this functionality is provided by the protocol discussed in section 5.3.

QoS monitoring agents

Every site in L^{2}imbo has an associated local management tuple space together with a number of QoS monitoring agents. These monitoring agents monitor key aspects of the system and inject tuples representing the current state of that part of the system into the management tuple space. Some typical forms of QoS monitoring agent are outlined below:

- *Connectivity monitors*: Watch over the characteristics of the underlying communications infrastructure and make available information such as the current throughput between hosts.
- *Power monitors*: Review the availability and consumption of power on a particular host. In particular, applications can obtain power information on host peripherals and may utilise hardware power saving functionality as appropriate.
- *Cost monitors*: Determine the cost associated with the current communications links between hosts.

The precise configuration of QoS monitoring agents can vary from site to site. As above, the architecture is open in that new QoS monitoring agents can readily be added to the configuration. Monitors can observe events at various points in the system including: the rate of injection of tuples into a given tuple space, the rate of access to tuples in tuple space (through `in` or `rd` operations), the total throughput currently achieved from that node, the cost of the current channel, the level of connectivity of that node, the power availability and rate of consumption at that node, the processor load and the current physical location of that node. In this way, the architecture deals uniformly with a range of QoS parameters relating to both communications and the general environment. In addition, the architecture can provide information relating to a particular tuple space or to the node in general. This overall architecture is depicted in figure 3.

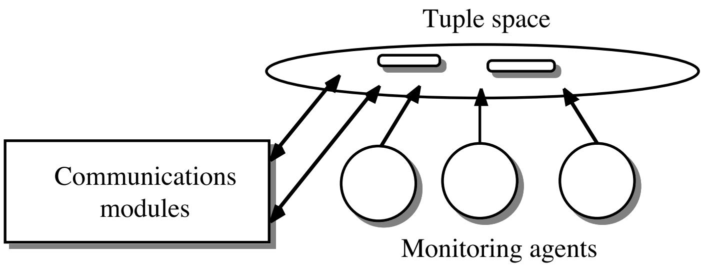

Figure 3. The L^{2}imbo monitoring architecture.

This architecture also has the advantage that information pertaining to a node can be made globally available (c.f. signals and environment variables which are only accessible by local applications). This is achieved by placing QoS information tuples in a tuple space, which other sites can access. This allows, for example, agents on different nodes to find out about the location of a particular site, its current processor load, the throughput it is currently experiencing, etc. Note however that sites can make QoS information private simply by selecting a particular membership agent that prevents access from other sites.

Type management agents

As mentioned earlier, the optional typing which may be applied to a tuple can enable sub-typing hierarchies to be established between them. The type management agent on each site is responsible for determining inter-tuple type relationships based on system tuples of the type `add_type` which are snooped from a given tuple space. The agent provides facilities to the tuple space protocols to assist them in discovering when suitable sub-types can be found based on supplied matching criteria. In addition, the type management agent is the authority through which the sub-type relationship tuples generated by local applications are ratified for validity.

4.2.5. Support for adaptation

The L^{2}imbo architecture supports a variety of mechanisms for adaptation. Such mechanisms are typically employed on detecting a significant change as a result of QoS monitoring.

One of the main techniques for achieving adaptation is that of *filtering agents*. Filtering agents are a special form of bridging agent in the architecture. As stated in the preceding section, a bridging agent is one that links arbitrary tuple spaces and controls the propagation of tuples between these spaces. Filtering agents are essentially bridging agents that perform transformations on the tuples to map between different levels of QoS and are based on the work of [28] and [29] on filtering agents for multimedia computing. They rely on typing information to identify the subset of tuples to be filtered.

Filtering agents can, for example, be used to translate between different media formats. More commonly though they are used to reduce the overall bandwidth requirements from the source to the target tuple spaces. For example, a filtering agent can act between two tuple spaces dealing with MPEG video and only propagate I-frames to the target tuple space. The filtering agent could also perform more aggressive bandwidth reduction, for example by performing colour reduction on the I-frames (as proposed in [30]).

The importance of filtering agents is that it is possible to construct a number of parallel tuple spaces offering the same service, e.g. the propagation of video frames, but at radically different levels of QoS. An agent can therefore select between the different levels depending on their level of connectivity. On detecting a drop in available bandwidth (or indeed an increase), they can switch to a different tuple space. This overall approach is illustrated in figure 4.

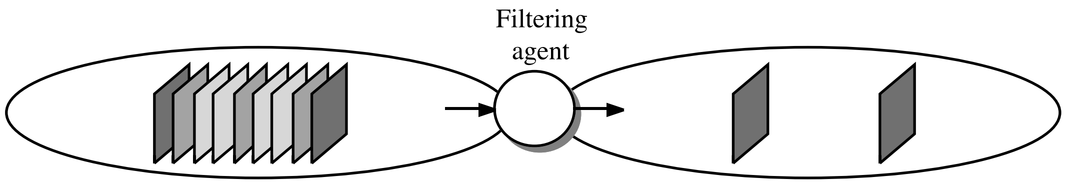

Figure 4. The use of filtering agents.

The architecture also supports a number of other forms of adaptation. For example, on detecting QoS violations, a sending agent can choose to adapt the rate at which tuples are injected into the tuple space. This is however a rather crude mechanism in an environment supporting multiple receivers with potentially different levels of connectivity. More interestingly, a receiver can selectively `in` or `rd` certain types of tuple and ignore others on detecting a drop in *their* connectivity. For example, they can select I-frames only and ignore P- and B-frames (achieving a similar effect as above). Similarly, they can select base encodings in hierarchical encoding schemes. With the appropriate allocation of priorities on these tuple types the underlying platform can discard the lower priority tuples on detecting congestion, implying that they need not be transmitted over a lower bandwidth link.

5. Prototype Implementation

5.1. Overview

We have developed a distributed systems platform based on the L^{2}imbo programming model presented in section 4. The platform consists of a small stub library which is linked with each application process and a single daemon process, an instance of which executes on each participating host. Application requests (i.e. `in`, `out` and `rd` operations) are marshalled by the stub library and passed to the daemon process. The daemon process collaborates with other instances of itself on remote hosts to provide tuple space repositories and matching functions. Hence the tuple space is implemented in a distributed fashion and the daemon processes are required to maintain consistency between copies of the tuple space on different hosts. This enables versions of the tuple space to remain accessible during periods of disconnection. The issue of consistency between copies of the tuple space is examined in detail in section 5.3. Figure 5 shows the overall platform architecture.

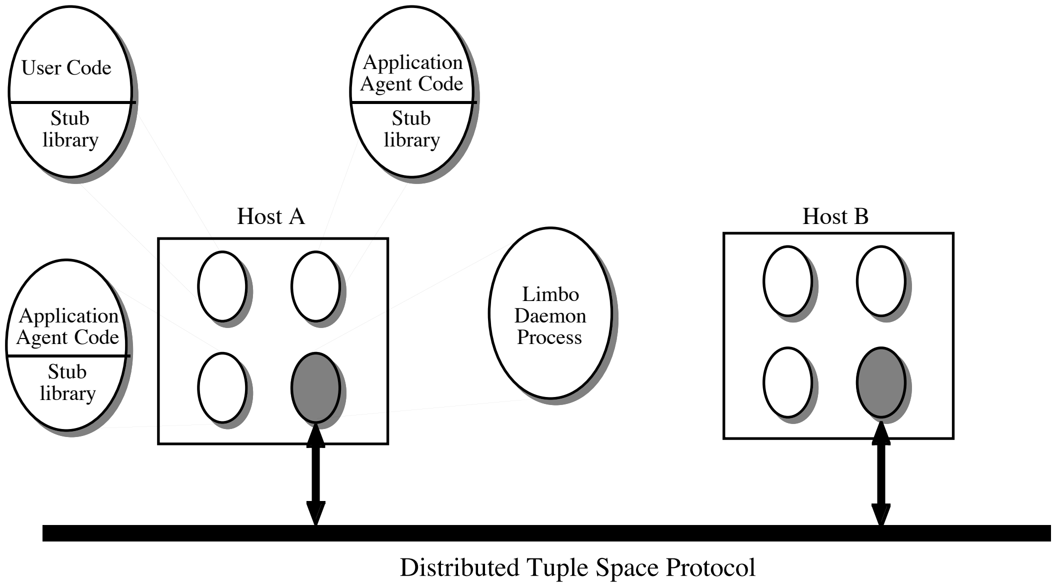

Figure 5. Overview of the L^{2}imbo platform.

Instances of the daemon process communicate using a protocol called the Distributed Tuple Space (DTS) protocol. The daemon process and the DTS protocol are described in more detail in the following sections.

5.2. The L^{2}imbo daemon process

As previously stated, all tuple space interactions are coordinated by an instance of the L^{2}imbo daemon running on each host. The structure of the L^{2}imbo daemon process can be presented as a number of layers as shown in figure 6.

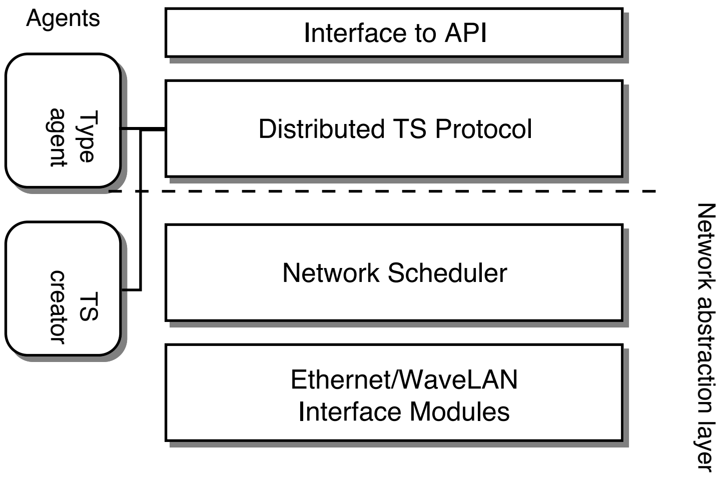

Figure 6. Structure of the L^{2}imbo daemon process.

The lowest layer of the daemon is the network interface layer. Each network supported by the L^{2}imbo distributed systems platform has a corresponding interface module. All interface modules present a generic interface 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. The network scheduler accepts protocol messages from higher layers and, based on associated QoS parameters, determines the order in which they are transmitted. The QoS structure currently contains only two fields, priority and deadline. Within each priority, messages are scheduled in earliest deadline first (EDF) order. Messages with the highest priority (smallest number) are scheduled before those of successive priorities (even if a lower priority has an earlier deadline). The priority can thus be considered to be a measure of the urgency that the associated deadline is met by the scheduler. This concept is based on previous work by Nieh on thread scheduling for continuous media [31].

The bottom two layers (consisting of the network scheduler and network interfaces) are collectively known as the network abstraction layer. The network abstraction layer has been designed to provide a set of generic transport services that are independent of both tuple spaces and network technology.

The protocol layer is responsible for the implementation of application requests (i.e. `in`, `out` and `rd`). This layer is described in detail in section 5.3. Finally, the stub layer is responsible for communicating between the L^2imbo daemon and client applications on the same host. By centralising all of the applications accesses to the tuple spaces through a single process on each host, the platform gains an overall picture of the demands on the available network (or networks) and is able to balance the load and manage congestion more effectively. However, there is of course a performance penalty associated with this approach since each message involves an additional context switch and local communications overhead. This issue is explored in more detail in section 6.

5.3. The Distributed Tuple Space protocol

A single Distributed Tuple Space protocol module on each host is responsible for providing the tuple space repository and associated matching functions for every tuple space used by client applications (in the first instance this is just the universal tuple space). The module uses the protocol to maintain the distributed local caches of tuples and requests (anti-tuples) and to ensure that they reach eventual consistency.

It is essential that such a distributed implementation does not apply locking strategies for operations which remove tuples and avoids algorithms which lead to acknowledgement implosion, both of which critically affect performance. Fortunately there are a number of features of the tuple space paradigm which greatly simplify the implementation task. Firstly, the model does not specify how long it takes for tuples to propagate to and from the tuple space: as long as tuples are matched non-deterministically and tuples can never be `in`'d more than once (tuples are thus unique) the model holds. Similarly, tuple operations are not causally ordered and total ordering does not have to be maintained. Furthermore, providing the uniqueness of tuples is maintained, it is not necessary to enforce strict consistency between the caches associated with distinct instances of the L^2imbo daemon.

Our protocol is based on IP multicast and borrows application level framing concepts from SRM the scalable multicast transport which underpins *wb* [32] and *Jetfile* [33]. Essentially, each distributed tuple space is modelled as a multicast group (see figure 7). This ensures that application level partitioning (by creating new application specific tuple spaces for a given domain) is reflected at both the platform and the communications level and hence helps L^2imbo applications to scale.

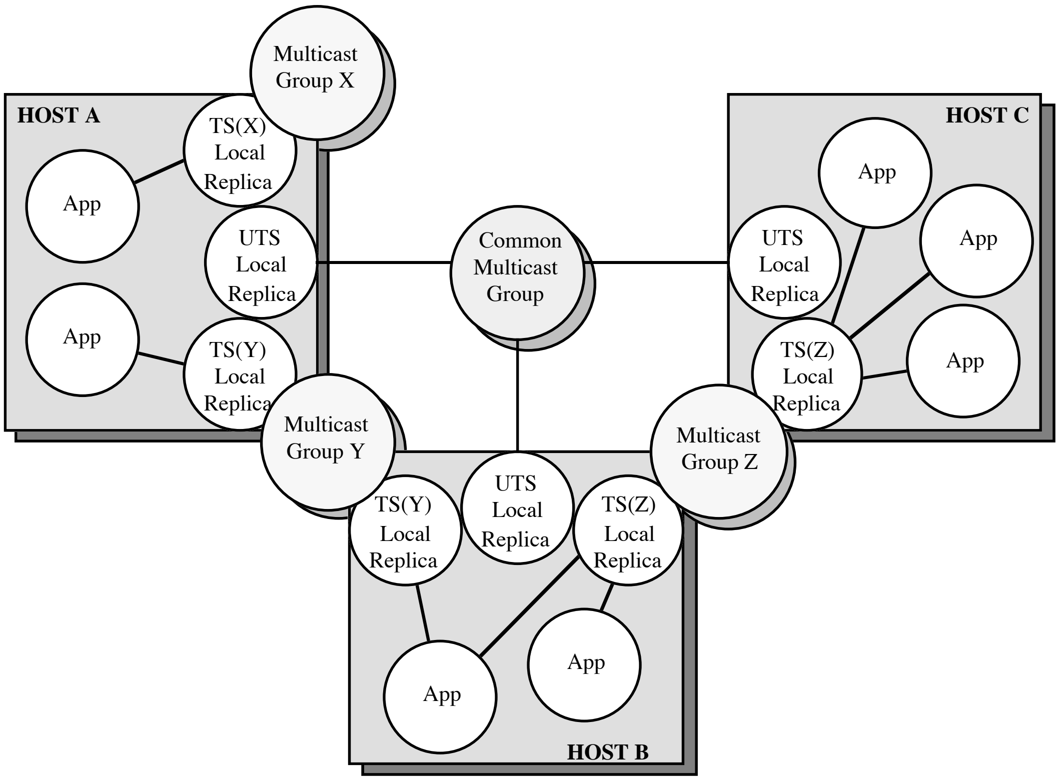

Figure 7. The decentralised architecture.

Each tuple and anti-tuple is given a unique identifier which comprises a daemon identifier and a monotonically increasing integer. All messages in the protocol are multicast and it is assumed that the messages will be snooped by all available hosts in the multicast group. The format of the nine protocol messages are shown in figure 8. Individual protocol messages may be concatenated into single multicasts for increased performance.

```
OUT         [tuple_id, owner_id, type, tuple]

IN          [client_id, request_id, type, spec]

RD          [type, spec]

CHOWN_REQ   [tuple_id, client_id]

CHOWN_ACK   [tuple_id, owner_id]

ACCESS      [tuple_id]

DELETE      [tuple_id, request_id]

REPAIR_REQ  [tuple_id]

REPAIR_ACK  [tuple_id, owner_id, type, tuple]
```

Figure 8. Distributed Tuple Space protocol messages.

The user operations `in`, `rd` and `out` map onto the messages as shown in table 1. The `out` operation propagates a tuple to all group members which then cache this locally until explicitly removed by a DELETE message. The speed of the `in` operation governs the overall performance of the tuple space and causes the most problems in distributed implementations. The key to the performance of our prototype lies in the notion of tuple ownership. Importantly, ownership can be transferred to any of the participants of the tuple space and does not solely rest with the creator of the tuple. Before a tuple can be removed (by an `in` operation) the ownership of the tuple must be transferred to the performer of the `in` operation (achieved by nomination by the current owner of the tuple or by using an explicit CHOWN_REQ message). Any user of the tuple space is free to `in` (remove) any tuples that they currently own without consulting any other user.

User Op.	Action
out	`if (any matching local RD requests)`
	`    satisfy requests`
	`if (pending matching local IN request)`
	`    satisfy request`
	`    tx (OUT [tuple_id, our_id, type, tuple])`
	`    tx (DELETE[tuple_id, in_request_id])`
	`else`
	`    if (pending foreign IN request which matches)`
	`        tx (OUT [tuple_id, in_requester_id, type, tuple])`
	`    else`
	`        tx (OUT [tuple_id, our_id, type, tuple])`
in	`if (a matching tuple is queued)`
	`    if (we are the owner)`
	`        tx (DELETE [tuple_id, in_request_id])`
	`    else`
	`        queue the IN request`
	`        tx (CHOWN_REQ [tuple_id, client_id])`
	`else`
	`    queue the IN request`
	`    tx (IN [our_id, type, spec])`
rd	`if (a matching tuple is queued)`
	`    tx (ACCESS[tuple_id])`
	`else`
	`    queue the RD request`
	`    tx (RD[tuple, spec])`

Table 1. User operations.

Message	Action

`OUT`	`if (we already know about this tuple)`
	    `update ownership information`
	`else`
	    `queue tuple`
	    `while (any matching local RD request is queued)`
	        `satisfy requests`
	    `if (any RDs were satisfied)`
	        `tx (ACCESS[tuple_id])`
	    `else`
	        `if (a matching local IN request is queued)`
	            `if (we are the nominated owner)`
	              `tx (DELETE [tuple_id, in_request_id])`
	            `else`
	              `tx (CHOWN_REQ [tuple_id, our_id])`

`IN`	`if (a matching tuple is queued)`
	    `if (we are the owner)`
	        `tx (OUT [tuple_id, in_requester_id, type, tuple])`
	    `else`
	        `tx (OUT [tuple_id, owner_id, type, tuple])`
	`else`
	    `queue IN request`

`RD`	`if (a matching tuple is queued)`
	    `tx (OUT [tuple_id, owner_id, type, tuple])`

`CHOWN_REQ`	`if (we know about this tuple)`
	    `if (its been deleted)`
	        `tx (DELETE [tuple_id, in_request_id])`
	    `else`
	        `if (we still own tuple)`
	              `tx (CHOWN_ACK [tuple_id, client_id])`
	        `else`
	              `tx (CHOWN_ACK [tuple_id, owner_id)])`
	`else`
	    `tx (REPAIR_REQ [tuple_id])`

`CHOWN_ACK`	`if (we know about this tuple)`
	    `update tuple ownership`
	    `if (we are the new owner and we have a pending local IN)`
	        `service request`
	        `tx (DELETE [tuple_id, in_request_id])`

`DELETE`	`if (we know about the tuple)`
	    `delete tuple from store (and IN request it satisfies)`

`ACCESS`	`if (we know about the tuple)`
	    `if (we have deleted the tuple)`
	        `tx (DELETE [tuple_id, in_request_id])`
	`else`
	    `tx (REPAIR_REQ [tuple_id])`

`REPAIR_REQ`	`if (we have this tuple)`
	    `tx (REPAIR_ACK [tuple_id, owner_id, type, tuple])`

`REPAIR_ACK`	`if (we don't have this tuple)`
	    `queue tuple`

Table 2. Tuple space messaging protocol.

Optimisations can be achieved through careful assignment of the tuple ownership. For instance, consider the example where someone generates a tuple for which it knows there is a pending IN request. The generator of the tuple sends an OUT message nominating the originator of the pending IN as the owner. Upon snooping the OUT message and, assuming the IN request has remained unsatisfied in the meantime, the tuple can be immediately removed and a DELETE message generated. Moreover, based on observed interactions, a tuple which is generated in response (which is likely to be consumed by the originator of the last tuple) could nominate the originator as the owner. This mechanism allows RPC-like semantics to be modelled efficiently (if necessary).

The tuple identifier associated with every tuple enables members of the tuple space to detect tuples that are missing from their local cache. If a member detects a missing tuple, they issue a REPAIR_REQ message asking for a copy of the tuple. Members who have a copy of the tuple can multicast a REPAIR_ACK containing the missing tuple. The protocol uses a backoff proportional to the distance from the sender of the request to ensure that the closest cache which can satisfy the request responds first. If an acknowledgement is snooped, the timer governing the transmission of the acknowledgement is cancelled. Table 3 illustrates how the protocol messages are handled. Note that ACCESS and DELETE messages (generated by `rd`'s and `in`'s) do not need to be generated immediately since they are only used by other parties to either detect missing tuples or prevent the use of stale tuples. The more quickly these messages are transmitted the faster the views of the tuple space held by each node will converge but in any case the semantics will remain unchanged. This enables us to reduce transmission overhead by batching ACCESS's and DELETE's with other protocol messages.

Hosts are free to connect and disconnect from the multicast group (and/or network) at will. We assume that mobile hosts will connect through some form of mobility support gateway (such as that proposed for mobile IP and IPv6) and that the gateway will operate a cache proxy on behalf of disconnected clients. Since mobile hosts may be the nominated owners of tuples, we assume that a disconnected host may optimistically continue to access (`rd`) copies of the tuples which they have cached and do not own. However, the mobile is not allowed to remove (`in`) a tuple that has been OUT'd to the network, since the proxy may have yielded ownership while the mobile was disconnected (although, tuples generated during disconnection may be used since the rest of the network will be unaware of these until reconnection). When a host reconnects to the network, it only need inform the group of new tuples it has generated. The proxy will in turn inform the client of any cached tuples which have been deleted. Tuples missed during the disconnection can be obtained through the multicast of IN and RD messages in the usual manner.

5.4. Implementation status

We have built an implementation of the L^{2}imbo distributed systems platform which runs on Linux 2.0 (MULTICAST), SunOS 4.1.4 (MULTICAST-4.1.4) and Solaris 2.5. All these operating systems offer RFC 1112 compliant IPv4 multicast support, which is a prerequisite for running the DTS protocol. The platform currently consists of less than 14000 lines of C code with an executable size of approximately 48 Kbytes (Linux).

The platform implements the design as specified in section 5. In addition, we have implemented three key performance optimisations: piggybacking of DELETE and ACCESS messages, IN message suppression in the face of congestion and static linking of single applications with the L^{2}imbo daemon. These optimisations are considered in more detail below.

Piggybacking of DELETE and ACCESS messages

Since it is acceptable to delay the DELETE and ACCESS messages generated by `in` and `rd` operations, protocol overhead can be saved by piggybacking these messages on other protocol messages. In the current implementation a single DELETE or ACCESS message can be piggybacked on any other message (including other DELETE or ACCESS messages) by setting a flag in the message header and prepending the payload before that of the original message. This optimisation reduces the overhead due to protocol headers and, more significantly, spreads the time spent in the `select` and `recvfrom/sendto` system calls.

IN message suppression

In the protocol specification presented earlier both IN and OUT messages are propagated. On a lightly loaded network this is advantageous because it enables the creator of a tuple to nominate as an owner a process which has a currently pending IN request. However, in the face of bursts of message exchanges between processes it is advantageous to suppress the propagation of IN messages since the nominated owner can be determined based on prior interactions. In the current prototype the propagation of IN messages is controlled automatically by monitoring the interarrival time of protocol messages from a given host.

Static linking of single applications with the L^{2}imbo daemon

The final optimisation we have made in the implementation relates to the case when a single application on a host is utilising the L^{2}imbo platform. In this special case, it is possible to link the application into the platform with some minor modifications. The single process version (combined platform and application) will out perform the separate process version due to the reduction in context switching and interprocess communication overhead. Although this optimisation has limited applicability in most cases, we are planning to experiment with a mechanism by which applications can inject trusted stub code into the platform to behave as a proxy for the application for certain tuple interactions. The stub code could then be executed without necessitating expensive interactions with the application. This approach has been demonstrated in systems such as Sumo [34] which allow trusted client code to be run in response to QoS upcalls from the platform. Note that such code must be trusted or suitable precautions must be taken to enable this mechanism to be safely generalised (for instance, a safe operating environment could be provided such as safe-TCL or Java).

6. Analysis

As stated earlier, the tuple space paradigm was originally designed to operate in shared memory multiprocessor architectures where the inherent time and space decoupling enables transparent service rebinding and load balancing between processors. We believe that the time and space decoupling present in the model also has relevance in the mobile computing domain and provides an effective and consistent way of dealing with the migration, failure and disconnection of mobile hosts. However, a consistent concern raised regarding the tuple space paradigm is the difficulty of developing an efficient distributed implementation. In this section we consider the performance of L^{2}imbo in a fixed network environment in order to address these concerns (the performance of L^{2}imbo over a wide range of wireless communications technologies including TETRA, GSM and WaveLAN is the subject of our current research).

The performance of tuple space implementations is usually governed by two factors, i.e. the speed with which tuples and anti-tuples can be matched and the speed with which tuples can be removed from the tuple space. As a consequence, RPC style interaction is a worst-case scenario for any tuple space implementation because the communication is directed (and hence matching must take place on each interaction) and the relevant tuples must be removed by each party in the communication. Furthermore, RPC style test programs do not typically include in their performance figures the time the client takes to locate and bind to the test service.

New applications designed for L^{2}imbo are unlikely to rely heavily on RPC style communications. However, since many existing applications do use RPC style programming models we consider it important that the L^{2}imbo platform is able to approximate RPC semantics with reasonable efficiency. In the next section we compare L^{2}imbo to a number of well known RPC based platforms.

6.1. Test configuration

The test suite consists of three separate pairs of client and server processes which carry out timed RPC interactions using raw UDP sockets, the ANSAware distributed systems platform (version 4.1) [35] and the L^{2}imbo prototype respectively. In the case of L^{2}imbo, the RPC interaction is modelled by a server process which executes an `in` operation for a tuple of type *request*. When the `in` operation has been satisfied the server `out`'s a tuple of type *reply* and continues with the next `in` operation. The client simply outputs a succession of *request* tuples, waiting for a *reply* between each one.

The test suite was compiled on two host/OS environments, i.e. PCs running Linux 2.0 and Sun Sparcstations running SunOS 4.1 both with IP multicast support compiled in. Both pairs of machines are networked with 10Mbps Ethernet and are located on the same, though separate, subnets. The versions of the sockets and L^{2}imbo test software were identical on both hardware platforms. However, while the ANSAware platform used on the Suns is a standard release, the ANSAware platform used in the Linux environment is our own port based on the SunOS distribution and contains a number of low-level performance enhancements [15].

To isolate the additional overhead we incur for splitting the L^{2}imbo platform into separate processes, we have run tests for both the optimised (linked into a single executable) and unoptimised (separate process) forms of the client and server.

6.2. Test results and analysis

The client and server processes were timed over 1000 RPCs (or equivalent). Each RPC consists of a call with the specified payload size and a null reply. Each test was run ten times and averaged to obtain the results presented in tables 3 and 4. All timing figures can be viewed dually as the time taken in seconds to complete each test or the time in milliseconds for a single interaction.

Payload (bytes)	Sockets (UDP)	ANSAware 4.1 (REX)	Limbo DTS (linked)	Limbo DTS (separate processes)
256	0.98	2.73	1.90	3.13
512	1.30	3.06	2.30	3.54
1024	1.96	3.72	3.09	4.35
2048	3.02	5.81	4.46	5.92
4096	4.95	20.12	6.93	9.26
8192	8.67	40.09	11.48	14.83

Table 3. Comparison of relative performance on Linux.

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 4. Comparison of relative performance on SunOS (vanilla ANSAware).

The figures in table 3 clearly demonstrate that in the majority of cases L^{2}imbo clients and servers outperform their ANSAware counterparts. The only exception to this is when L^{2}imbo clients and servers are running as separate processes and the packet sizes are small (i.e. less than 2K). In these cases the overhead of the additional context switching and local communication required in L^{2}imbo has a significant impact on the figures. Reducing to a minimum the overheads associated with exchanging messages between the stubs and the daemon process is clearly an important factor in improving the performance of L^{2}imbo. In all cases the integrated L^{2}imbo client/server pairs outperforms ANSAware.

Table 4 presents the test results for the Sun versions of the test suite and uses the reference version of the ANSAware platform. The figures clearly show that in the optimised form L^{2}imbo outperforms ANSAware in all cases. However, the context switch and additional local messaging exacts a heavy toll, further reinforcing the need for improved application and daemon interprocess communication.

The figures taken on Linux additionally suggest that L^{2}imbo performs comparably to other established RPC based platforms such as COOL, a CORBA based platform developed by Chorus Systèmes [36]. 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^{2}imbo 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 L^{2}imbo takes just 1.9 ms (we attribute this as being due to L^{2}imbo's use of UDP whereas COOL uses TCP as the RPC transport mechanism).

7. Concluding Remarks

Current distributed systems platforms designed for mobile environments are based on synchronous, connection-oriented communications with associated QoS monitoring and management. In this paper we have argued that such platforms are not well suited to use in emerging heterogeneous mobile environments. We have described L^{2}imbo, a new platform based on an alternative programming paradigm i.e. tuple spaces. Given the time and space decoupling inherent in the tuple space model we believe the paradigm provides an interesting approach that addresses many of the shortcomings of existing mobile support platforms. In particular, the platform offers an asynchronous programming model and an architecture for reporting and propagating QoS information relating to all aspects of the system.

We have described in detail the design and implementation of the L^{2}imbo platform and have presented the results of our initial performance analysis. This shows that L^{2}imbo is able to perform comparably to RPC based platforms when supporting bursts of intensive directed communications, the traditional worst-case scenario for a tuple space platform.

We are now developing a number of applications to exercise the L^{2}imbo platform. More specifically, we are developing a suite of applications to support collaborative work by members of the emergency services. This suite of applications will include a GIS based application which will enable users to collaboratively view and annotate geographical data as well as a number of applications which process continuous media. The network bearer for this work will initially be WaveLAN but the applications will be ported to TETRA and GSM in the long term. Through this process we hope to gain experience not only of using L^{2}imbo for large-scale application development but also of operating L^{2}imbo over a network with substantially different characteristics to our current environment.

Acknowledgements

The work described in this chapter was carried out under the auspices of the "Supporting Reactive Services in a Mobile Computing Environment" project (EPSRC Grant No. GR/K11864). The authors would like to thank the anonymous referees whose comments have helped shape this paper.

References

[1] N. Davies, S. Pink and G. S. Blair, Services to Support Distributed Applications in a Mobile Environment, in: *Proc. SDNE '94,* Prague, Czech Republic (June 1994) 84-89.

[2] R. H. Katz, Adaptation and Mobility in Wireless Information Systems, IEEE Personal Communications 1(1) (1st Quarter 1994) 6-17.

[3] A. Schill and S. Kümmel, Design and Implementation of a Support Platform for Distributed Mobile Computing, Distributed Systems Engineering Journal 2(3) (1995) 128-141.

[4] N. Davies, G. S. Blair, K. Cheverst and A. Friday, Supporting Adaptive Services in a Heterogeneous Mobile Environment, in: *Proc. MCSA '94*, Santa Cruz, California, U.S. (December 8-9, 1994) 153-157.

[5] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford and M. F. Kaashoek, Rover: A Toolkit for Mobile Information Access, in: *Proc. SOSP '95*, Copper Mountain Resort, Colorado, U.S. (December 3-6, 1995) 156-171.

[6] D. Gelernter, Generative Communication in Linda, ACM Transactions on Programming Languages and Systems 7(1) (January 1985) 80-112.

[7] W. N. Schilit, A System Architecture for Context-Aware Mobile Computing, *Ph.D. Thesis*, Department of Computer Science, Columbia University, New York, NY 10025, U.S. (1995).

[8] R. Katz, E. Brewer, E. Amir, H. Balakrishnan, A. Fox, S. Gribble, T. Hodes, D. Jiang, G. Nguyen, V. Padmanabhan and M. Stemm, The Bay Area Research Wireless Access Network (BARWAN), in: *Proc. IEEE COMPCON Spring '96,* Santa Clara, California, U.S. (February 25-28, 1996).

[9] A. Danthine, Y. Baguette, G. Leduc and L. Léonard, The OSI 95 Connection-mode Transport Service - The Enhanced QoS, in: *Proc. 4th IFIP Conference on High Performance Networking,* Liege, Belguim (December 14-18, 1992) 232-252.

[10] L. Delgrossi, R. G. Herrtwich, C. Vogt and L. C. Wolf, Reservation Protocols for Internetworks: A Comparison of ST-II and RSVP, in: *Proc. NOSSDAV '93,* Lancaster House, Lancaster, U.K. (November 1993) 199-207.

[11] N. Davies, G. S. Blair, K. Cheverst and A. Friday, Experiences of Using RM-ODP to Build Advanced Mobile Applications, Distributed Systems Engineering Journal 2(3) (1995) 142-151.

[12] B. D. Noble, M. Price, and M. Satyanarayanan, A Programming Interface for Application-Aware Adaptation in Mobile Computing, in: *Proc. MLIC '95,* Ann Arbor, Michigan, U.S. (April 10-11, 1995) 57-66.

[13] ISO international standard ITU-T recommendation X.903: Open Distributed Processing Reference Model Part 3: Architecture, *Standard Recommendation ISO/IEC 10746-3: 1995*, ISO WG7 Committee (January 1995).

[14] A. Friday and N. Davies, Distributed Systems Support for Mobile Applications, in: *Proc. IEE Symposium on Mobile Computing and its Applications,* Savoy Place, London (November 24, 1995) 6/1-6/3.

[15] A. J. Friday, G. S. Blair, K. W. J. Cheverst and N. Davies, Extensions to ANSAware for Advanced Mobile Applications, in: *Proc. ICDP '96,* Dresden, Germany (February 27-March 1, 1996).

[16] ISO draft recommendation X.901: Basic Reference Model of Open Distributed Processing - Part 1: Overview and Guide to Use, *Draft Report* (1992).

[17] APM Limited, ANSA: An Engineers Introduction to the Architecture, *Technical Document release TR.03.02,* APM Cambridge Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, U.K. (November 1989).

[18] B. Schilit, N. Adams and R. Want, Context-Aware Computing Applications, in: *Proc. MCSA '94,* Santa Cruz, California, U.S. (December 8-9, 1994) 85-90.

[19] D. Gelernter, N. Carriero, S. Chandran and S. Chang, Parallel Programming in Linda, in: *Proc. International Conference on Parallel Processing* (August 1985) 255-263.

[20] J. Pinakis, Providing Directed Communication in Linda, in: *Proc. 15th Australian Computer Science Conference*, Hobart, Tasmania (January 1992).

[21] 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).

[22] 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. (July 1989).

[23] S. Hupfer, Melinda: Linda with Multiple Tuple Spaces, *Technical Report YALEU/DCS/RR-766,* Department of Computer Science, Yale University, New Haven, Connecticut, U.S. (February 1990).

[24] N. Carriero, D. Gelernter and L. Zuck, Bauhaus Linda, Selected Papers from ECOOP '94, Bologna, Italy (July 1994) 66-76.

[25] N. H. Minsky and J. Leichter, Law-Governed Linda as a Coordination Model, *Selected Papers from the Workshop on Models and Languages for Coordination of Parallelism and Distribution*, Bologna, Italy (June 1994) 125-146.

[26] ISO/IEC 13235-1 | ITU recommendation X.950, Open Distributed Processing - Trading Function: Specification (March 1997).

[27] P. Butcher, A. Wood and M. Atkins, Global Synchronisation in Linda, Concurrency: Practice and Experience 6(6) (1994) 505-516.

[28] J. Pasquale, G. Polyzos, E. Anderson and V. Kompella, Filter Propagation in Dissemination Trees: Trading Off Bandwidth and Processing in Continuous Media Networks, in: *Proc. NOSSDAV '93,* Lancaster House, Lancaster, U.K. (November 1993) 269-278.

[29] N. Yeadon, Quality of Service Filters for Multimedia Communications, *Ph.D. Thesis*, Lancaster University, Lancaster, U.K. (May 1996).

[30] N. Yeadon, F. Garcia, D. Hutchison and D. Shepherd, Filters: QoS Support Mechanisms for Multipeer Communications, IEEE Journal on Selected Areas in Computing 14(7) (September 1996) 1245-1262.

[31] J. Nieh and M. Lam, Integrated Processor Scheduling for Multimedia., in: *Proc. NOSSDAV '95,* Durham, New Hampshire, U.S. (April 19-21, 1995).

[32] S. Floyd, V. Jacobson, S. McCanne, C. Liu and L. Zhang, A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing, in: *Proc. ACM SIGCOMM '95*, Cambridge, Massachusetts, U.S. (August 1995) 342-356.

[33] B. Grönvall, I. Marsh and S. Pink, A Multicast-Based Distributed File System for the Internet, in: *Proc. 7th ACM SIGOPS European Workshop*, Connemara, Ireland (September 2-4, 1996).

[34] G. Coulson, G. S. Blair, P. Robin and D. Shepherd, Extending the Chorus Micro-Kernel to Support Continuous Media Applications, in: *Proc. NOSSDAV '93,* Lancaster House, Lancaster, U.K. (November 1993) 49-60.

[35] APM Ltd., An Introduction to ANSAware 4.0, Architecture Projects Management Ltd., Cambridge, U.K. (February 1992).

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

Biographies

Nigel Davies graduated from Lancaster University in 1989 and later that year joined the Computing Department as a research associate investigating storage and management aspects of multimedia systems. As a result of his work in this area he was awarded a Ph.D. in 1994. After a spell as a visiting researcher at the Swedish Institute of Computer Science (SICS) where he worked on mobile files systems he returned to Lancaster, first as site-manager for the MOST mobile computing project and subsequently as a lecturer in the Computing Department. His current research interests include mobile computing, distributed systems platforms and systems support for multimedia communications. ACM. E-mail: nigel@comp.lancs.ac.uk

Adrian Friday graduated from the University of London in 1991. The following year he moved to Lancaster and participated in the MOST project involving Lancaster University and E.A. Technology. In 1996 he was awarded a Ph.D. for his work on "Infrastructure Support for Adaptive Mobile Applications" and is currently a research assistant in the Computing Department. E-mail: adrian@comp.lancs.ac.uk

Stephen Wade has been a research student in the Computing Department since graduating from Lancaster University in 1995. He is an active participant in the "Supporting Reactive Services in a Mobile Computing Environment" project and is working towards his Ph.D. on "Using an Asynchronous Paradigm for Mobile Distributed Computing". ACMS. E-mail: spw@comp.lancs.ac.uk

Gordon Blair is currently a Professor in the Computing Department at Lancaster University. He completed his Ph.D. in Computing at Strathclyde University in 1983. Since then, he was a SERC Research Fellow at Lancaster University before taking up a lectureship in 1986. He has been responsible for a number of research projects at Lancaster in the areas of distributed systems and multimedia support and has published over a hundred papers in his field. His current research interests include distributed multimedia computing, operating system support for continuous media, the impact of mobility on distributed systems and the use of formal methods in distributed systems development. ACM. E-mail: gordon@comp.lancs.ac.uk