S. Bødker, M. Kyng, and K. Schmidt (eds.). *Proceedings of the Sixth European Conference on Computer-Supported Cooperative Work, 12-16 September 1999, Copenhagen, Denmark*

PSI: A Platform for Shared Interaction

Kevin Palfreyman[1], Tom Rodden & Jonathan Trevor
Computing Department, Lancaster University, UK
{kev, tom, jonathan}@comp lancs ac uk

This paper presents an infrastructure to support the dynamic sharing of information across a range of cooperative environments The infrastructure builds upon the use of shared common spaces by using a distributed tuple space to provide information sharing at its base level The platform extends existing considerations of tuple spaces by adding mechanisms to provide active support for sharing data elements. The use of a tuple space moves away from previous models of distribution in cooperative systems that focus on the propagation of events to focus on active data sharing The use of data tuples allows the sharing of information to be independent of the information model allowing a wide range of applications and environments to be supported. The paper presents the infrastructure and shows how it can be used to support information sharing across a number of different forms of cooperative system and application.

1. Introduction

The CSCW research community has seen the development of a variety of systems that support real time cooperation through some sense of shared interaction. Initial systems focused on the development of shared interface techniques such as WYSWIS (Foster, 1986), interface coupling (Dewan, 1991) and the tailoring mechanisms (Greenberg, 1991). Underpinning the development of these interface techniques was a growing acceptance of a common model of shared interaction. The general strategy agreed was to consider multi user interfaces in terms of

[1] Now at Xerox Research Centre Europe, Cambridge Laboratory, 61 Regent Street, Cambridge, UK, kevin palfreyman@xrce xerox com

different views on a shared data model (Bentley, 1992; Dewan, 1991; Hill, 1993; Lee, 1996; Mansfield, 1997). This separation allowed the interaction to be shared by sharing the underlying model while allowing its effects to be presented in a variety of different ways.

In addition to these technical developments, a number of researchers began to consider the development of "Common Information Spaces" (Bannon, 1997) and suggest that one way to support cooperation was to consider the cooperation in terms of users interaction with a pool of shared information. It is clear that both technical and conceptual agreement on the development of applications to support cooperative interaction have started to emerge. The core of this agreement is an exploitation of the concept of some form of shared space that provides a focus for interaction. These shared spaces allow resources associated with the cooperative interaction to be gathered together and presented to a community of users. These spaces all provide a number of features to promote shared interaction.

- Information and resources can be shared between users by placing them in shared spaces with the effects of interaction propagated to other users.
- Users have an awareness of others in these shared spaces as user actions are reflected through their effects on shared resources in the space.
- Some agreed model is used to manage and structure access to the shared information in the space. Essentially, two main models are used to manage interaction.

  Space based models where the space is partitioned to manage the interaction. (For example, room based systems (Roseman, 1996))

  Time based session models where the access to the space is managed by providing a series of different sessions (Edwards, 1994)

The majority of these information spaces agree on the need to allow resources and information to be shared, the provision of techniques to structure this sharing and the need to allow users to be aware of the activities of others. However, despite this broad agreement the majority of systems do not provide any mechanism for interaction with other cooperative systems. In essence, each of these cooperative systems each still behaves as closed environments with limited access to other forms of cooperative systems.

In this paper we present the development of a platform to support sharing across a heterogeneous collection of applications. The platform allows interaction to be shared between cooperative applications by providing support for the active sharing of information. This allows cooperative applications exploiting a pool of common shared information to extend this model beyond their traditional application boundaries.

Within the Platfrom for Shared Interaction (PSI) sharing is provided through an active sharing infrastructure that combines a simple data model with active sharing mechanisms. The platform is realised as an active layer on top of an extended tuple space that allows *information tuples* to be shared across a number

of distributed applications. The development of the platform required both the construction of the platform itself and the augment of an existing tuple space with notification mechanisms to make it an active tuple space. The representation of information as tuples in the active tuple space allows us to support a number of different forms of presentation and interaction of the shared information.

In the remainder of this paper we present an overview of the need to provide support for active sharing that moves beyond existing event based awareness models. We outline an overall architecture for the platform and then consider the development of the platform itself. The developed platform combines extensions to the existing model of tuple spaces to make them active and the development of a set of services to provide access to this tuple space. We outline and describe these extensions before considering how the platform is used in practices and how it may be used by existing applications.

2. The Development of Supporting Services

One reason for the lack of cooperation across cooperative systems is that a mismatch exists between the need for sharing within these systems and the provision of supporting infrastructures. The current generation of infrastructures provided to cooperative applications are motivated by the need to support awareness and focuses on the propagation of user actions. While the notion of awareness is important we feel that this focus has led to an imbalance in the provision of supporting services and we need to consider the development of active sharing services that allow interaction with different forms of heterogeneous information to be shared between cooperative systems.

The majority of supporting services and protocols have tended to focus on the propagation of awareness information in the form of events. This has included the development of protocols to augment the World Wide Web (Palfreyman, 1996) and a number of general event and awareness protocols. Although they focus on propagation of awareness some of these protocols already incorporate some model of sharing and information spaces. For example, the Corona communication service incorporates a model of shared spaces (Hall,. 1996). The use of a shared information space is also mirrored in the development of the NSTP protocol that uses "places" and "things" (Patterson, 1996).

The use of places and the move to state based services is also manifest at in applications in the way in which the shared interaction is managed. As we have said previously, two models for managing shared interaction have emerged.

1) Event based supporting services have tended to be based on the notion of *application sessions* as an abstraction to represent the set of destinations to which application events are delivered.
2) State based applications have seen the introduction of space based or *room based* approaches for managing shared interaction where updates

are clustered around a set of collected resources.

We wish to develop a shared interaction platform that exploits the active sharing of state. This allows us to marry both these approaches and our platform provides mechanisms to support both paradigms. Additionally the focus on state as a means of conveying interaction allows us to provide a bridge between real time and more asynchronous forms of interaction

Existing systems also offer some form of structuring model to manage shared interaction. Often these models exploit some concept of space or a spatial model. For example, the supporting services in Teamwave are articulated in terms of rooms (Roseman 1996), while Corona offers shared spaces (Hall, 1996) and ORBIT offers locales (Mansfield, 1997). Similarly, Collaborative Virtual Environments (CVEs) such as DIVE (Carlsson, 1993) and Massive (Greenhalgh, 1995) exploit 3D models of spatial arrangements as a means of structuring the shared information space.

In developing our supporting platform we wish to recognise the need for providing some form of structure and the need to provide support that allows a diverse set of structuring models to coexist. It is important that we do not impose an external structure but that we recognise the situated nature of these cooperative applications and the emergent properties of the structure used to manage interaction in shared information spaces. *Thus rather than develop another model of shared information spaces and support for this model we wish to provide support to bring these different spaces together.*

In order to allow a range of models to coexist we focus on the provision of very lightweight data model for shared state that allows users to make information available to others and to allow linkages between different forms of shared state. The service we have developed provides a number of key features

- A lightweight data model that allows support for a number of different forms of shared space
- Extensible support for sharing allowing the addition of different persistence mechanisms.
- Support for a wide variety of forms of data by providing a clear canonical representation of information
- Support for flexible forms of sharing with varied patterns of use.

In order to provide these features we have chosen to move away from more traditional communication based model of distribution This move reflects a response to the emerging demands cooperative applications place on the supporting infrastructure. However, as we build upon these alternative distribution techniques we in turn place new requirements upon them requiring changes to these underlying platforms. In the following section we show how we formed an overall architecture by building a supporting platform on top of an amended tuple based distribution platform.

3. The PSI Architecture

The PSI architecture provides an active distributed real-time sharing platform. The central approach is the cooperative sharing of arbitrary data among a heterogeneous set of distributed applications. The platform allows cooperative applications to make application information externally available by sharing it with the environment. The arrangement of the platform is shown in figure 1:

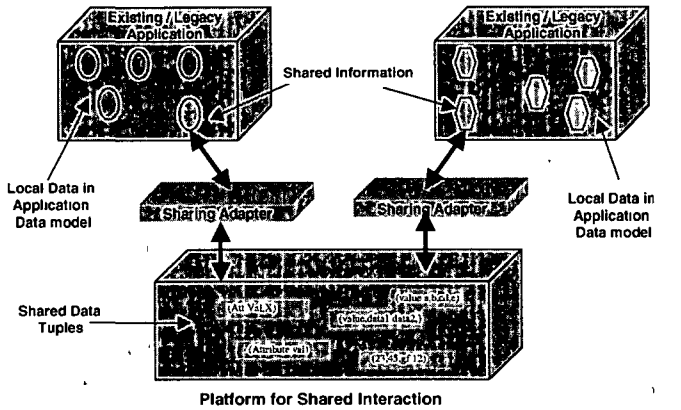

Figure 1: The general view of the Platform for Shared Interaction

To share information an application makes it available to the platform through a simple sharing adapter that provides a mapping between internal representations and the facilities provided by the PSI sharing mechanisms. Once internal application data has been made available to the platform its state information is held as a set of shared data tuples with updates and alterations to these tuples propagated to all applications sharing this information. Other applications can access this state information through *local proxies* that reconstruct the information. The platform manages alterations to the state by propagating updates through the tuples in the active tuple space.

The platform provides flexible sharing by extending current work on tuple spaces to provide an active tuple infrastructure. Traditionally tuple spaces have provided distributed access to simple data tuples and managed the issues of distribution to ensure that changes to these tuples are made available across a community of users. The platform for shared interaction extends these facilities to provide a set of cooperative shared interaction services by augmenting the existing tuple model in two ways.

- Extending the existing passive state based model of sharing to provide active support for sharing where applications are informed of alterations to shared tuples.
- Providing a simple and lightweight data model and API that supports the structuring information between cooperative applications.

What makes the PSI approach novel and advantageous over previous approaches is that the shared information space metaphor is being supported at the underlying distributed system level in a completely semantic-less manner. This is important for several reasons. First, it enables the programmer to reflect changes in the applications model of cooperation within the underlying communications structure of the distributed system (Bentley, 1995). Secondly, by moving the data sharing into the distributed system, the natural boundaries of sharing between applications can be transcended, allowing the data to be accessed and shared across applications, operating system and network.

In the following section we consider both the extensions to the underlying tuple space and the development an application interface that exploits this extended tuple space platform. The tuple space platform provides the advantage of offering a simple state representation that can be used to model a range of structures. In the following section we consider the extensions we made to the tuple space paradigm to allow the development of more active shared spaces. This is then build upon to provide the shared platform application interface.

4. Extending Tuple Spaces

A *tuple space* is a well known distributed systems mechanism originating from the work on the Linda project (Gelernter, 1985). Tuple spaces have their roots in the development of parallel processing applications and are traditionally used for (and best suited to) supporting systems that have well identified producer-consumer flows of data - where one part of the system is responsible for providing data that another part processes.

A tuple space contains zero or more *tuples*, as well as references to other tuple spaces. Each tuple consists of an ordered list of typed fields. For example, a tuple that would hold a name-value pair would be (<*String*>, <*Data*>), where <> identifies the type of the field. Note that fields are not named, making the ordering of the fields within a tuple important. Each field within a tuple is also said to be an "*actual*" or a "*formal*". An "actual" field is one that contains real data, whereas a "formal" field is a placeholder used in matching (see below). Traditional tuple spaces support three key operations, shown in Table I.

Op	Params	Description
RD	Tuple	Read any tuple from the tuple space that matches certain fields in the query tuple There is no guarantee as to which tuple will be consumed if more than one would match the query tuple Matching is performed by checking the "actual" fields have the same types and values as the tuple in the space Once all the fields have been matched, the remaining (if any) formal fields in the tuple are filled with the matching tuple's corresponding values
IN	Tuple	This operation is similar to RD but removes the matched tuple from the space Again there is no guarantee as to which tuple will be consumed if more than one would match the query tuple
OUT	Tuple	Write the tuple from this application out to the tuple space Once the tuple has been written it becomes available to all applications connected to the space All the fields in a tuple being OUTed must be "actuals".

Table I Standard tuple space operations

The PSI exploits and extends a distributed tuple space called L^2imbo that is described in detail elsewhere (Davies, 1997). The design and implementation of L^2imbo is such that it is more suited than other systems for use in distributed and mobile applications; it makes extensive use of IP multicast and supports interaction across a range of heterogeneous platforms (Davies, 1998). These characteristics differ from previous systems, which generally had more of a focus on performance for parallel processing within homogeneous networks and dedicated parallel processing environments. L^2imbo is also fully distributed, rather than being based on a client-server architecture. The system uses application-level framing techniques from the Scalable Reliable Multicast protocol, as used in a number of the MBone tools, and models each distributed tuple space as a multicast group.

The use of the L^2imbo tuple space provides us with two distinct advantages in terms of an infrastructure for real-time cooperative applications.

i. The mechanism is based on fast access to *decentralised* shared state and the infrastructure is optimised to support this arrangement. Applications sharing data with the space do not have to worry about how changes to their shared state are propagated to other applications.
ii. The use of tuples as a representation of information does not impose a particular data model and applications can externalise diverse forms of structured information across heterogeneous applications.

The L^2imbo tuple space is a passive entity which ensures that state information is available to all those who access the platform. It is the applications responsibility to continually monitor for changes to a given fixed set of tuples and act upon the changes to these tuples. This is done by making the RD operation blocking so that the execution of the application making the RD call stops until a tuple successfully matches the RD

4.1 Augmenting distributed tuple spaces for CSCW

Although the tuple space metaphor inherently supports the notion of a shared

space the traditional view of a tuple space is limited to a fixed producer-consumer model of information flow This model is problematic because the consumers are *expecting* the information to be produced. No facilities are provided for actively notifying applications of events occurring in a tuple space (such as a new tuple arriving) and only very limited facilities are available for querying, rather than consuming or extracting, the current contents of the tuple space.

To allow the more dynamic arrangements involved in cooperative settings we have extended the existing view of tuple spaces by augmenting the L^2imbo tuple space with an awareness facility offering additional operations that extend the standard set of tuple operations. These are summarised in Table II.

Operation	Params	Description
Register	Callback OUT/IN type	Register a callback in the application executed whenever a particular type (or subtypes) of tuple is added/arrives (OUTed), or removed/deleted (INed) from the tuple space. The callback receives the identifier of the tuple space where the operation occurred and a copy of the actual tuple the operation occurred on
Unregister	reg-id	Remove previously registered callback

Table II Extended tuple space access operations

The core of this extension is a simple registration based notification facility that allows applications to register an interest in particular tuples and be informed via a callback mechanism when a tuple matching the given pattern is added or removed from the tuple space. The development of this callback mechanism means that applications reading the tuple space need no longer block and are free to handle other events while waiting for particular changes to the tuple space. Other useful facilities in this particular tuple space implementation include *collect / copy_collect*, which *in / rd* all matching tuples into a new tuple space. There are also non-blocking versions of *in* and *rd*, named *inp* and *rdp*.

These extensions assist in transforming the traditional tuple space to make it an active distributed shared space. The platform is realised on top of this active tuple space and provides an interface that can be used by a range of applications. The following section provides an overview of the application interface.

5. The PSI application interface

The platform for shared interaction (PSI) basically views applications as isolated "data spaces" populated with local data in their particular information model. These applications may choose to share some or all of this data so that other applications may access this information. The platform provides support for applications to insert and extract information from the platform and allows applications to interact with this shared information. The PSI architecture is divided into three separate layers, as shown in Figure 2.

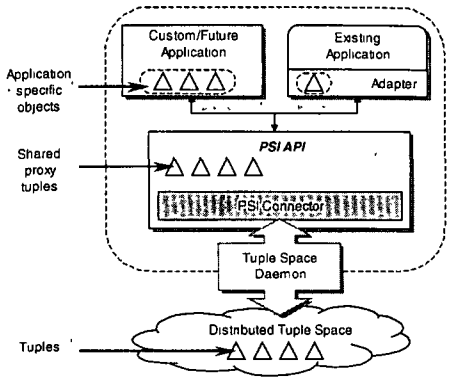

Figure 2 Architecture of the Platform for Shared Interaction

The core features of the architecture can be best described in four separate pieces. The *application layer* contains the applications themselves (or adapters for legacy applications). The application contains *application specific objects* that are native to that application. The application decides which of these objects are exported to the shared information space. The application communicates with the platform through the PSI API.

The *PSI API* maintains and decomposes the application specific data into *shared proxy tuples*. These are objects that reflect the contents of tuples in the underlying *tuple space*. They change as their tuple-space counterparts change and visa-versa.

The *PSI connector* provides the main coordination point for applications and the API, facilitating access to information in the shared spaces. This includes tasks such as adding and removing shared state from the tuple space and monitoring the tuple space for changes.

The *tuple space daemon* provides access to the tuple space and allows applications (such as the PSI connector) to add, query and remove data from the tuple space, as well as maintaining the same tuple spaces in a replicated fashion across any number of machines.

5.1 The PSI API

The API for the platform provides a simple common interface to allow applications to share and manipulate data through the tuple space. The API is currently realised in Java to provide as much cross-platform support and integration as possible. The API defines two types of tuple that are exported into the shared tuple space and carry application data: *data tuples* and *relation tuples*. These tuples are realised locally as proxies with corresponding tuples created in the shared tuple space.

In the following sections we consider the structure of the data tuples and

relation tuples that provide the application interface for the platform. A dedicated connector object that manages the connection between the proxy tuples and the tuples in the underlying tuple space supports these two tuples.

Data tuples

The Data tuple represents a single atomic piece of shared data. A Data tuple consists of the fields shown in Table III.

Field	Type	Description
Tuple ID	String	Some unique identifier that identifies this particular tuple in the tuple space It contains a tuple sequence identifier
Application	String	Identifier of the application sharing the object (typically its mime-type)
Session ID	String	Some unique identifier for this session
Object ID	String	Some unique identifier for the shared object to which this data belongs
Data Name	String	The name of the data value
Data Type	String	The type of the data value
Data Value	Data	The data itself

Table III Fields in a Data tuple

To extract tuples relevant to a particular application from the shared space, it must be possible to identify distinct pieces of application shared data Aside from the *TupleID* field which is used for state update (see later), an application may uniquely identify a particular piece of data using the following quad:

```
{Application, SessionID, Object ID, Data name}
```

The uniqueness or scope of each identifying field in the quad is the responsibility of the previous field. Thus each application with a given application identifier is responsible for maintaining and allocating the different *Session-ID*'s in that application. Each session allocates the *Object-ID*'s for objects in that session and so on.

The platform is policy free with the *Application* field and the *Session-ID* field being given equal weighting. This means that the platform supports a number of alternative arrangements allowing both session-based and space-based views of the data, even simultaneously. If an application were to query the tuple space using a specified *Application* class (resource), then it would be possible to present to the user a view of current application sessions and the data associated with them. In contrast, the *Session-ID* can be associated with a particular space (for example a room in Teamrooms(Roseman, 1996) or a region of a CVE (Greenhaulgh, 1995) and if the tuple space is queried using a specified *Session-ID* it would present a view of all of the applications available in that particular space.

Consider a particular collaborative application based on the rooms metaphor which needs to obtain all of the data for all of the resources (chat, whiteboards, shared documents, reference to audio channel, etc) in a given room, then it simply queries the space using the session identifier for the required room Alternatively, if a particular type of resource (application) is being used in many

different sessions, then querying with the application class will produce a list of the session identifiers all of the instances where it is used.

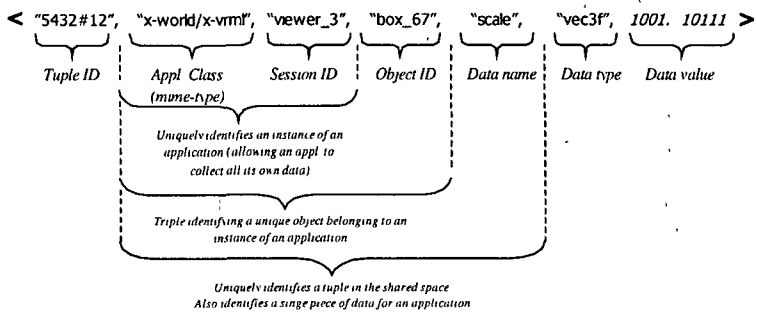

Figure 3 Composition of an example Data tuple

The value for the *Application* field of a data tuple is specific for each class of application and would normally be compiled into the application code (or its external interface). For any given application class, there are a number of *Object-ID* (and their nested arbitrary *Data-name*) values reserved for management functions. This enables an application to display standard properties for each item in a list of sessions or resources.

Once data is made available in the tuple space, it can be retrieved using the normal tuple space operations. By combining different actual and formal values in a data tuple, applications can execute template queries for different data. Some useful combinations are shown in Figure 3, above. For example, by providing an application class of "x-world/x-vrml" and leaving all other fields as formals in a data tuple, the application can obtain a list of all *Session-ID*'s for that type of application in the shared space.

Relation tuples

Arbitrary M-N relations can also be shared within the platform by using multiple 1-1 relationship tuples, whose definition is shown in Table IV.

Field	Type	Description
Application	String	Identifier of the application defining this relationship (typically its mime-type)
Session ID	String	Some unique identifier for this session
Object ID	String	Unique identifier for the shared object which this relationship concerns
Relation	String	Name of the relationship to which this 1-1 mapping belongs
Left	String	The LHS of the relationship
Right	String	The RHS of the relationship

Table IV Fields in a Relation tuple

This allows us to reflect and share the core connections central to the information models and data structures used internally by cooperative

applications by making the relations linking information objects externally available through the shared interface platform. Particular instances of given relations are therefore defined by the quad:

```
{Application, Session ID, Object ID, Relation}
```

An entire M-N relationship is broken down by the API into 'M x N' 1-1 tuples, to allow queries to be executed on the platform over the shared relationships. To extract an entire shared relationship, the API need only perform a single "collect" operation on the tuple space with a complete quad.

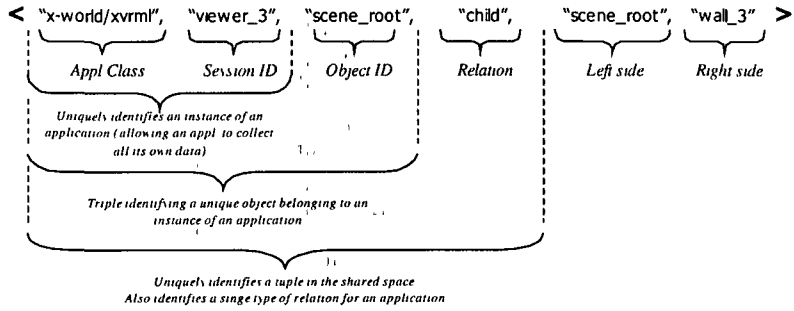

Figure 4 Composition of an example Relation tuple

Relation tuples can be used to store hierarchy, such as those used in information and organisation models. Equally they can be used to share the hierarchy in 3D geometry. In this case the geometry would typically be broken into several smaller simple parts and grouping nodes used to tie them together. To share this state across the space we need data tuples for each part of the geometry and a data tuple for the group node. We then need to create a relation tuple to link the parent to each of its children. An example of this style 1-N use of relation tuples can be seen in Figure 4 above.

The PSI Connector Object

The Connector object provides the point of communication between the PSI API and the underlying tuple space daemon, manifesting changes to the local shared data proxies as the actual values are modified in the tuple space, and visa versa. The connector provides methods that can be invoked through the API to query and manipulate the shared spaces in the platform.

The two methods *shareData( )* and *shareRelation( )* instruct the connector to share some application-defined data tuple or a MxN relationship with the space. If the shared space already contains a matching data tuple (i.e. it has the same unique quad fields) or set of 1x1 relation tuples (for the MxN relationship) then the connector makes local proxies for these existing tuples and returns their values to the application. If no such data or relation exists then new tuples are put into the shared space and local proxies are created. These API methods provide a

very simple interface for an application to either share new data with the platform or to get existing shared values back.

5.2 State update and propagation

Propagation of state and notification of state changes between the underlying tuple space and the application is performed in two distinct ways depending on the nature of the update and the demands of the cooperative application. Consider a telepointer object whose position is shared in the tuple space. Another application is displaying the co-ordinates of the point in a simple text field. If the pointer moves quickly from point A to point B several hundred potential state changes may occur. However, the most important changes are the starting position of the pointer and the final resting position of the pointer. Those in the middle tend to be less important. In essence we categorise state updates to data tuples as either *transient* (if we miss an update it doesn't matter) or *persistent* (everyone sharing this state should see the same value). A data tuple can be updated using either mechanism at any time. Which update mechanism is used for a particular data tuple is left to the actual application(s) changing the data value (which can exploit semantic knowledge about what the data *is* to decide on the most appropriate method).

Persistent updates

Because persistent events change the fundamental state of the data in the shared space, no explicit event tuple is created for these updates. Rather, the sender (the source) of the event (typically the local tuple proxy) IN's the state to be changed, modifies it, and then re-OUTs it back to the space. Two fields of the tuple are modified: the new state value containing the new data value and the tuple ID, which has its sequence component incremented.

From the tuple space perspective one tuple has been removed and a new one added. Therefore these changes to the tuple space are then distributed to the other clients of the tuple space, where the daemon will trigger all the registered callbacks in the PSI connector for the newly OUTed tuple. The connector then extracts the new state from the tuple and updates the local proxy for the data in question. Any objects, such as the applications, which have registered callbacks on the shared proxy tuple are subsequently called. Note there are two levels of callback, first from the tuple space daemon to the proxies via the connector, and secondly from those proxies to any interested applications.

Transient events

Transient events are carried by special event tuples in an event tuple space. The format is shown in Table V.

Field	Type	Description
Tuple ID	String	Identifies the data tuple containing the same Tuple ID to which this transient event applies.
Event Sequence#	Int	Current sequence number for this transient update
Data name	String	The name of the data value
Data value	Data	The data itself

Table V Fields in a Transient Event tuple

The tuple ID identifies the data tuple over which this transient update is occurring, while the event sequence number provides ordering between each transient event. This sequence number is maintained by each PSI connector, which updates its last known value whenever a new transient event is added to the event tuple space or arrives through a callback

To generate a new transient event, the source of the event (typically the proxy of the shared data tuple) creates a new transient event tuple (incrementing the local sequence number and using the last known tuple ID of the state being changed) and adds it to the tuple space. This tuple is propagated by the underlying tuple space system to all connected machines, which may issue callbacks to connected applications via the PSI connector. Each connector receiving this callback forwards the change in state (carried in the transient tuple) to the relevant PSI object and records the last event sequence number of the tuple.

Transient events are never IN'ed by the platform API, only RD. Consequently no negotiation has to occur in the distributed system for the ownership of this tuple, which allows transient events to send information is very fast. However, because they are never IN'ed transient events are never explicitly removed from the tuple space by the platform API or applications. Therefore each transient event has an *expires* field to enable the tuple space to remove old events during garbage collection.

Update Consistency

Consistency between the persistent events and transient events on the same data tuple is guaranteed because of the incremented tuple ID sequence value on the shared state whenever a persistent change is OUTed to the space. This means if the "tuple ID" of a transient event refers to a data tuple with a tuple ID which no longer exists, or whose sequence number is less than the last known one, these events can be safely discarded as they are "behind" some other part of the system (and are transient anyway).

Consistency between the transient events themselves is fairly loose as the sequence ordering is only guaranteed on each machine rather than across the whole tuple space. This is because no IN's are ever needed for transient events and consequently no serialisation is ever performed at the tuple space level. The advantage of doing RD's rather than IN's is a vast improvement in performance (no checking of ownership exchange needs to occur across the distributed replicas

of the tuple space). Consequently, the sequence numbering of two contradicting "actions" in a distributed application could coincide since the sequence number is added by the local PSI connector. This could cause the object to get wildly different transient state updates in quick succession. However the first machine to commit a persistent value for the data tuple to the tuple space will "win" as it will change the actual tuple ID to which these transient events apply - and persistent updates *are* serialised because they rely on IN's.

6. Using the platform

To demonstrate some of the features and operation of the platform consider the following example where two applications (in these case two different CVE systems) are modified to allow them to export and share an object through the PSI. Note these applications do not have to be the same application (one could be VRML and one could be DIVE), nor do they need to be on the same machine but are connected through the distributed tuple space under the PSI.

The first step in sharing the object between the applications is the construction of two shared adapters, one for each application. For the purpose of this example, the adapters share a common application class identifier within the PSI, "x-application/x-shared-vr".

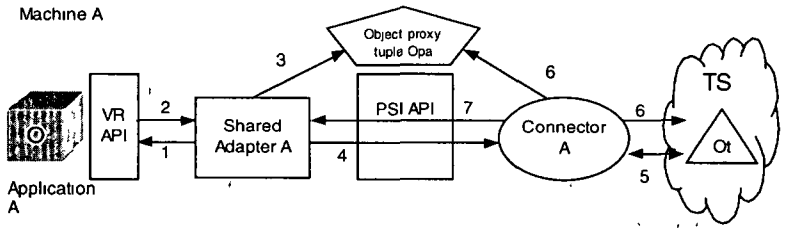

Figure 5 Exporting a VR object to the universe

The second part of making the object commonly available and shared between the applications is to export the state of the object held by the application to a common shared space in the PSI. Figure 5 illustrates this process. First, the adapter requests the object data through the applications API (such as the EAI) (1,2). The adapter proceeds to create a local "data" proxy tuple to contain this data (3), defining the various fields as appropriate - the application class "x-application/x-shared-vr", a session identifier "session1", and the name of the object "Opa" as well as value of the data just extracted from the application. The adapter performs a "shareData" call on the connector using this proxy tuple. The connector tries to locate any tuple in the tuple space that may match this (5) by making the data type and value fields in the tuple formals rather than actuals (see section 4). This will fail to match any existing tuple so the connector OUTs a new data tuple to the tuple space (Ot) (6). The original "shareData" calls returns and a

reference to the local proxy Opa is passed to the adapter by the connector (7).

During (6) the connector also registers itself and the shared adapter making the shareData call as objects to be notified whenever the proxy (Op) changes. Whenever the local proxy is updated by either the connector (in response to a change in the tuple space data tuple) or the adapter (in response to a change in the application) the adapter and connector will receive notification of the change The change in data value can be propagated down to the tuple space or up to the application depending where that change originated from.

Now that the object O is being shared in the PSI tuple space other applications may also share it by connecting to the PSI. This is accomplished in a similar manner to the initial exporting of the object, and is shown in Figure 6.

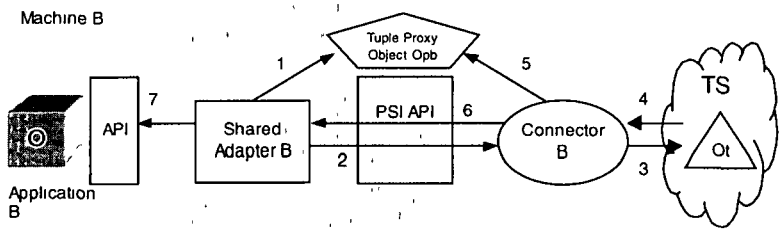

Figure 6 Importing a VR object from the universe

The adapter creates a local proxy tuple (Opb) the same application class, session ID, and object ID as was previously shared above, placing dummy values in the data value field (1). As before, the adapter performs a "shareData" request on the connector (2) that queries the tuple space for data tuples with a matching application, session, object and data name (3). However unlike before, the query succeeds and a matching tuple is RD in (4). The connector fills in the missing fields in the local proxy (Opb) (5) before registering itself and the adapter to be notified whenever the data in the proxy object is modified (5). A reference to the proxy object is returned to the shared adapter (6) and the adapter uses the VR API to add the shared object (Opb) into the applications VR world (7).

Now that each adapter and connector are registered to receive updates from Op and Opb, and each connector is monitoring changes to the data tuple being shared in the tuple space Ot, updates to O are automatically propagated throughout the system. For example if the second application modifies the object adapter B only needs to change the data value in the proxy tuple Opb. The connector is notified of this change which updates the data tuple Ot in the tuple space. The distributed tuple space updates all of its replicas including the one on machine A. The tuple space daemon notifies connector A of the change to the tuple which updates the local proxy tuple Opa. The tuple notifies adapter A about the change to its state that then uses the application API to send the update to the application.

If both applications A and B had been running on the same machine and had contacted the same connector then the shareData call issued by adapter B would

have been matched within the current tuples being proxied by the connector. A reference to *that* proxy would have been returned to the adapter rather than the same proxy it passed initially.

Finally, even if both applications and adapters quit, the shared object Ot remains in the active tuple space. Therefore when either of the adapters reconnected to the PSI at a later time and try to share the same data value again they will receive the current, last known, value from the tuple space.

7. Additional supporting services

In addition to providing a series of supporting mechanisms we have also realised a number of supporting services on top of PSI. These services illustrate how the semantically neutral nature of the platform has allowed us to build upon it to provide an additional layer of abstraction. In particular we present an abstraction that supports the sharing on different media and the use of existing components.

7.1. Sharing multimedia data

The original L^{2}imbo tuple space was initially designed to support streamed media, such as audio and video, across networks with variable performance and capabilities (e.g. between a wireless connection and a local area network). Consequently, the PSI architecture is not limited to facilitating the sharing of persistent (and fairly static) pieces of data.

Sharing streamed multimedia data, such as an MPEG stream, between applications is accomplished by creating a virtual buffer in the tuple space using one or more Data tuples; these are updated through a combination of persistent and transient changes. The source of the stream keeps updating the Data tuple in the tuple space while any number of recipients consume the updated "buffer" and play the media.

Although this method works reasonably well in practice the main advatange is the flexibility of the platform rather than performance. The flexibility of the platform allows different mechanisms for performing media streaming to improve performance. For example, the tuple space could be used to setup and establish a well-known but separate multicast group for the broadcast of the media between the various platform clients using protocols like RTP.

7.2. Shared JavaBeans

Building on top of the platform and API, another level of abstraction has been added to the platform that provides the ability share public properties of arbitrary JavaBean components[2]. This functionality enables the extremely simple

---

2 Considerable thanks are due to John Lamping at Xerox PARC for the initial development of a JavaBeans platform that greatly influenced the development of the shared beans layer in PSI

construction of collaborative applications from existing standard components. For example, it becomes relatively simple to construct shared graphical user interfaces, or indeed to share an underlying model that may then be represented in a number of ways in different user applications.

This mechanism works by using the reflection facilities of the Java platform and dynamically constructing listeners for all bound properties of a bean. As the value of a bound property of a bean changes, the property change events are reflected as changes to tuples in the shared tuple space. Since PSI architecture builds upon our active notification extensions to the tuple space concept, these property changes are in turn passed on to any other application that uses the same shared instance of the component.

The functionality provided means that a developer may treat PSI as a simple way of sharing Java components between applications that may either reside of the same host machine, or be spread across a distributed set of heterogeneous hosts on a network. The basic pattern of interaction for applications wishing to use the mechanism follows two simple steps.

```
Attempt to connect to a shared instance of the component.
IF instance_found THEN use that instance
        ELSE create new instance and share that with the platform
```

To provide this functionality we provide two basic methods, namely *connect()* and *share()*. The *connect()* method is overloaded to enable an application to retrieve particular instances of a component, for example by specifying property values. The *share()* method provides all of the functionality required to persist a component (recursively through all of its properties) into the shared space.

This application of PSI has been used to create a number of applications. For example, it was straightforward to take an existing 3rd-party compiled bean that provided a 3D geometry viewer and create a distributed shared viewing application (without being able to access any of the source).

The open design of the PSI API and tuple spaces makes it easy to add additional services and functionality to the platform – an application or service needs only to connect to the tuple space and can then query, add or remove the tuples within it. One of the most common types of service is the persistence service. The service simply registers a callback for all tuples being added or removed from a particular tuple space. This callback then copies or removes tuples it receives from a local database. If the tuple space crashes or fails for some reason the backup application can restore the lost information by retrieving the original persistent data and relationship tuples from the database and merging them with the current state of the tuple space.

8. Conclusions

In this paper we have presented the development of a platform that allows the active sharing of information across cooperative applications. The platform has extended an existing tuple space platform to allow applications to actively share cooperative information between heterogeneous applications. The developed PSI platform extends existing event based awareness models by promoting active sharing of a state using a lightweight data model.

The developed platform needed to augment an existing tuple space by adding mechanisms to support notification in order to allow better support for the dynamic arrangement central to cooperative work. This active tuple space allowed the development of an API centred on a set of defined tuples. These tuples provided an interface that allowed object to be placed in the platform by creating locally proxies. The platform managed the connection between these local proxies and the tuples in the active tuple space to propagate updates to state across and between applications.

A number of areas are being considered for future work. One option is the possibility of migrating some of the functionality described down into the underlying tuple space implementation, thereby providing a more active platform at the level of the networking service. We are also investigating a number of other tuple space implementations, e.g. IBM's Tspaces and Sun's JavaSpaces. This is in order to determine if may be possible to provide our abstractions over these alternative base platforms, thereby providing a flexible sharing mechanism across completing platforms.

Another area of continued research that needs be addressed in the future is the area surrounding access control and authentication. Current the platform places information in a shared tuple space with little or no consideration of access control and security management We are currently considering the investigation of private tuple spaces and encrypted tuples as a means of providing this form of access management. Our final consideration is to extend our existing set of higher level services by adding even more a range of CSCW tools and management utilities as standard services.

9. Acknowledgements

This work was supported in part by the eSCAPE long term research project. Thanks are due to Adrian Friday and Nigel Davis at Lancaster University for their considerable work in the development of the original L^2imbo tuple space. Thanks are also due to John Lamping for the inspiration, structure and motivation for the development of the shared Beans facility.

10. References

Bannon, L. and Bodker, S. (1997) "Constructing Common Information Spaces", *In Proc. ECSCW97,* Lancaster, UK, Sept. 1997, pp 81-96, Kluwer

Bentley, R., Dounsh, P. (1995) "Medium versus Mechanism: Supporting Collaboration through Customisation" *In Proc. ECSCW95,* Stockholm, Sweden, Sept. 1995, pp 133-148, Kluwer.

Bentley, R., Rodden, T., Sawyer, P.; Sommerville, I (1992) "An architecture for tailoring cooperative multi-user displays", *In Proc. CSCW'92,* Toronto, Canada, Oct 1992, pp 187-194, ACM Press.

Carlsson, C , Hagsand, O (1993) "DIVE: A Multi User Virtual Reality System" *In Proc IEEE VRA1S,* Sept. 1993, pp394-400

Davies, N , Friday, A., et al. (1998). "An Asynchronous Distributed Systems Platform for Heterogeneous Environments". *Proc. 8th ACM SICOPS European Workshop: Support for Composing Distributed Applications,* Sintra, Portugal, 1998, ACM Press.

Davies, N., Wade, S. P., et al. (1997). "Limbo. A Tuple Space Based Platform for Adaptive Mobile Applications". *International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP97),* Toronto, Canada, May 1997, pp291-302

Dewan, P., Choudhary, R. (1991) "Flexible User Interface Coupling in a Collaborative System", *In Proc CHIVI,* New Orleans, LA, April 1991, pp 41-48, ACM Press.

Edwards, K.W. (1994) "Session management for collaborative applications" *In Proc. CSCW V4,* Chapel Hill, NC, Oct. 1994, pp 323-330, ACM Press.

Foster, G., Stefik, M (1986) "Cognoter: Theory and practice of a Colab-orative tool", *In Proc CSCW'86,* Austin, TX, Dec 1986, pp 7-15, ACM Press.

Gelernter, D. (1985) "Generative Communication in Linda." *ACM Transactions on Programming Languages and Systems,* 7(1), 1985, pp80-l 12, ACM Press.

Grccnberg, S (1991) "Personalizable groupware- accommodating individual roles and group differences", *In Proc. ECSCW9I,* Amsterdam, September 1991, pp 17-31.

Greenhalgh, C. M., & Bcnford, S. D.,(1995) "MASSIVE- A Virtual Reality System for Tele-conferencing". *ACM Transactions on Computer Human Interfaces (TOCHI), 2* (3), Sept 1995,' pp. 239-261, ACM Press

Hall, R.W , Mathur, A., Jahanian, F., Prakash, A. Rasmussen, C, (1996) "Corona: A Communication Service for, Scalable, Reliable Group Collaboration Systems", *In Proc. CSCW'96,* Boston, MA, Nov 1996, pp 140-149, ACM Press

Hill, R. D., Brinck, T., Patterson, J. F., Rohall S L., Wilncr, W. T. (1993) "The Rendezvous language and architecture", In *CACM,* 36(1), Jan 1993, pp 62-67.

Lee, J H, Prakash, A., Jaeger, T. (1996) "A Software Architecture to Support Open Distributed Collaboratones", *In Proc. CSCWV6,* Boston, MA, Nov. 1996, pp 344-353, ACM Press.

Mansfield, T., Kaplan, S., Fitzpatrick, G., Phelps, T., Fitzpatnck, M., Taylor, R. (1997) "Evolving Orbit: a progress report on building locales". *In Proc. Group'97,* Phoenix, Arizona, Nov. 1997, pp241-250, ACM Press

Palfreyman K A Rodden T. (1996) "A Protocol for User Awareness on the World Wide Web" *In Proc CSCW'96,* Boston, MS, USA, Nov 1996, ACM Press.

Patterson, J. F, Day, M. Kucan, J., (1996) "Notification Servers for Synchronous Groupware", *In Proc. CSCW'96,* Boston, MA, Nov. 1996, pp 122-129, ACM Press.

Roseman, M , Grcenberg, S, (1996) "TcamRooms. Network Places for Collaboration" *In Proc. CSCW96,* Boston, MA, Nov. ,1996, pp 325-333, ACM Press.