

# A Gaming Framework for a Transactional DSM System

M. Schoettner, M. Wende, R. Goeckelmann, U. Schmid, P. Schulthess  
Ulm University, Distributed Systems Department  
schoettner@informatik.uni-ulm.de

## Abstract

*State-of-the-art middleware like .NET or J2EE offers transparent access to distributed objects and services but lack of support for consistency between groups of objects. Distributed Shared Memory (DSM) is an interesting alternative to build distributed applications because it maintains consistency among object groups. Although DSM has been used in the past only for special number crunching programs we believe it is a good foundation to simplify the development of multi-player games and virtual reality applications. In this paper we present the relevant parts of our transactional DSM operating system (OS). Subsequently, we describe the gaming framework we have built on top of our DSM system. Finally we evaluate the framework with a sample game.*

**Keywords:** Distributed Shared Memory, Multi-player Games, Consistency.

## 1. Introduction

Many toolkits for the development of stand-alone virtual reality applications and games exist today. Typically they provide the programmer a high-level interface to represent complex geometry in a scene graph and to render the latter.

Several attempts to offer a distributed version of such toolkits have been made, e.g. Avocado [10], DIVE [14]. They provide support for network-based communication between the distributed processes that form an application. However, using these facilities typically requires significant effort from the programmer. Especially maintaining consistency of replicated objects is the burden of the developer or if supported still is considerable complex. We believe that a Distributed Shared Memory (DSM) is a good foundation to solve this problem.

Although DSM is well known in the scientific world it has been used only for special number crunching applications. There are several reasons why DSM systems are not widespread in commercial systems [9]. Of course there is a need for better interfaces to simplify the development of distributed programs and new application fields need to be explored and evaluated.

As a consequence the Plurix project implements a lean distributed operating system (OS) with an integrated Java

compiler for the PC platform. The central abstraction in the Plurix OS is a DSM providing a virtual address space shared among tasks on loosely coupled nodes. Plurix extends the traditional DSM paradigm by implementing a distributed heap storing both code and data. Extending established memory consistency models [6], we have introduced a new consistency model called transactional consistency [7]. A traditional file system is avoided by making the DSM persistent. Persistence and fault tolerance is achieved by writing checkpoints on disk. Numerous checkpointing strategies have been proposed for DSM systems [15]. The property of persistence is automatically determined by the system implementing orthogonal persistence [13]. Any object reachable from the root of a cluster-wide name service persists.

Our Java compiler is used for OS and application development, directly translating Java source texts into Intel machine instructions [16]. Native code generation is mandatory for OS and driver development in Java. Language-based OS development has been successfully demonstrated in systems like Oberon [8]. By using a popular language like Java we hope to increase the acceptance for a new OS by the programmer community.

Interesting new application fields for DSM systems are multi-player games and virtual reality applications. Both share scenes to be displayed on several machines of participants. Moving avatars or objects need to be kept consistent on each machine. This task is traditionally the burden of the programmer but not so in DSM systems. They implement transparent distribution and consistency and as a consequence simplify the development of the above applications. We have built a gaming framework prototype on top of our DSM system to study multi-player games. Currently we have only a simple teletennis game but the results are encouraging and we plan to develop more sophisticated games in the future.

In the first section of this paper we shortly review the Plurix DSM operating system. In the next section we present the gaming framework and basic components of our GUI prototype. Subsequently, we study a sample multi-player application – a teletennis game that has been built with our framework. In section five we compare our efforts to related work and finally we summarize our results and give an outlook on future work.

## 2. The Transactional DSM Operating System

### 2.1 Persistent Distributed Shared Memory

The term *distributed shared memory* was introduced by Keedy in 1985 [1]. The first prototype IVY was presented in 1988 by Li [2]. Subsequently, many DSM systems have been developed to mainly support parallel algorithms [3]. Most of the approaches were built on top of existing OS, e.g. Unix, Microsoft Windows, or using the Mach kernel.

The Plurix project implements a new PC operating system customized for paging-based DSM operation. The MMU detects non-local memory accesses and prompts the kernel to fetch the requested page from the cluster. The limitations of the 32-Bit address space will vanish with the advent of a 64 bit solution being developed for the IA64 architecture.

False sharing is a widely discussed issue in page-based DSM systems and decreases performance by page trashing [4]. Plurix alleviates this serious problem by a concurrent object relocation facility basing on a bookkeeping of references [5].

To application programmers the DSM appears as a conventional local memory heap and they are not confronted with special memory allocation functions like in other DSM systems. Implementing a DSM as a heap has also been adopted by the Murks system [17]. Heap management must be designed carefully to avoid unnecessary false-sharing situations. It is recommended that each node allocate memory in a different part of the DSM avoiding interferences. Furthermore, vital classes of the system need special protection, e.g. the page-fault handler. We have introduced different memory pools to support these requirements [18].

A central page server periodically writing consistent heap images to disc, supports persistence and fault-tolerance. Our first prototype stops the cluster and writes snapshots in an incremental fashion. A network snooping protocol reduces the amount of data to be saved during a checkpoint. In case of a critical error the cluster is restarted from the last checkpoint using the page server. We plan to implement a distributed page-server with asynchronous checkpointing and forward error recovery.

Distributed garbage collection relieves programmers from explicit memory management. Unreferenced objects can be collected very easily using the bookkeeping of references. In a DSM environment it is not a good idea for a distributed garbage collection (GC) scheme to mark heap blocks during a traversal because of the expensive network traffic and the growing collision probability with other nodes.

Since the page-server can maintain several heap images that are staggered in time it becomes interesting to use one of the older images for cyclic garbage collection. A mark & sweep variety of garbage collection can easily run on an inactive but consistent DSM image without disturbing the operation of the active cluster. Garbage

objects that have been identified off-line may be collected during some later phase of the heap.

Because all nodes of a cluster operate concurrently on the DSM adequate synchronization mechanisms are required to preserve memory consistency.

## 2.2 Transactional Consistency

Numerous consistency models were investigated in the literature [6], but for our purposes we implemented a novel memory consistency model using restartable transactions together with an optimistic synchronization scheme, which is described below.

Memory pages are distributed and replicated in the cluster and to avoid inconsistencies, memory accesses from different nodes are synchronized using our *transactional consistency model* [7]. We rely on the ability to reset operations in case of conflicting write operations on replicated memory pages. And thus create an optimistic concurrency control scheme for the shared objects in the DSM known from the database world. Optimistic concurrency control occurs in three steps: the first step is to monitor the memory access pattern of a transaction (TA). For this purpose we use the built-in facilities of the MMU.

The next step is to preserve the old state of memory pages before modifications. Backup images are created, saving the page state previous to the first write operation. These images are used to restore the memory in case of a collision, as described in the next step.

During the validation phase of a terminating TA the access patterns of all concurrent TAs in the cluster are compared. In case of a conflict the TA is rolled back using the backup images. In case of no conflict the backup copies are discarded.

To reduce the collision probability TAs should be short and working sets should remain small. Currently, we have implemented a first-wins collision resolution basing on a circulating token. A transaction wanting to commit asks for the token. If granted it broadcasts its write-set to all nodes in the Fast-Ethernet LAN. All nodes in the cluster receive that commit request and compare their running TAs to detect conflicts. If the latter is true they abort their conflicting TA. This collision resolution strategy provides for low latency of the commit requests, but is somewhat wasteful in the case of a collision.

Each network packet contains the commit number of the last committed TA. Herewith, we can detect lost network packets. If for example a node missed a commit before and is trying to commit an TA all other nodes immediately see that its commit number is out of date and can reject the request and initiate a recovery.

## 2.3 Scheduler

We have adopted the cooperative multitasking model from the Oberon system, [8]. In each station there is a central loop (the scheduler) executing a number of installed transactions. The transactions must be short to minimize collision probability and it is the task of the programmer to split up long running transactions. Each station maintains a transaction vector to keep track of active transactions. Transactions can have priorities to differentiate between important and uncritical tasks. The central event loop steps through the transaction vector attempting to execute one after the other.

## 2.4 Name Service

Any Java object residing in the heap may be registered in the global name service of the cluster and is later retrieved via directories and subdirectories. This corresponds to the directory structure of traditional file systems but the functionality of the name service is extended to include scoping in the Java compiler, to store configuration information, and to cover all naming issues occurring in the OS. Registered objects reside in the shared heap and are not serialized. Therewith we implement orthogonal persistence - any class, instance or array may persist.

The Plurix compiler automatically registers symbol information during the compilation in the name service [19]. Methods of classes or instances may be immediately invoked using this persistent symbol information. Object serialization is provided for external backups only.

## 3. The Gaming Framework

Our gaming framework provides programmers a shared scene-graph accessible from all nodes of the DSM. Replication of scene data and consistency of moving objects is automatically provided by the underlying DSM. The object-oriented framework allows the creation of application specific classes, which inherit, extend, and adapt properties and behavior of the framework.

### 3.1 The Plurix Graphical User Interface

The first GUI prototype follows the Oberon tradition offering a tiled desktop with non-overlapping windows and active texts [8]. Furthermore it implements the well-known Model View Concept introduced by Smalltalk.

Texts contain characters and objects called text elements. Any text characters may be interpreted as a command by clicking on them. Text elements are used to embed pictures, animations or whatever and may also react on user interaction. The Oberon event handling has been modified towards the Java listener concept.

Figure 1 shows classes of our GUI (on the left side) used by the gaming framework that are described in the following text. *Elements* are the basic components of the GUI. They describe the position, the color and the dimension of objects on the screen. *TextElements* extend *Elements* by mouse-handlers and keyboard-handlers to process user input. Bitmaps or textures are very important for games and are represented by the class *PicElement* that extends *TextElement*. Currently only rectangular shapes are supported.

Any Plurix application like a game must extend the given class *Application* of the window system. This class creates a new viewer and associates the application to it. All visible objects of the game are drawn in this viewer.

### 3.2 Classes of the Gaming Framework

In this subsection we briefly introduce the most important classes of the gaming framework, shown on the right side of figure 1.

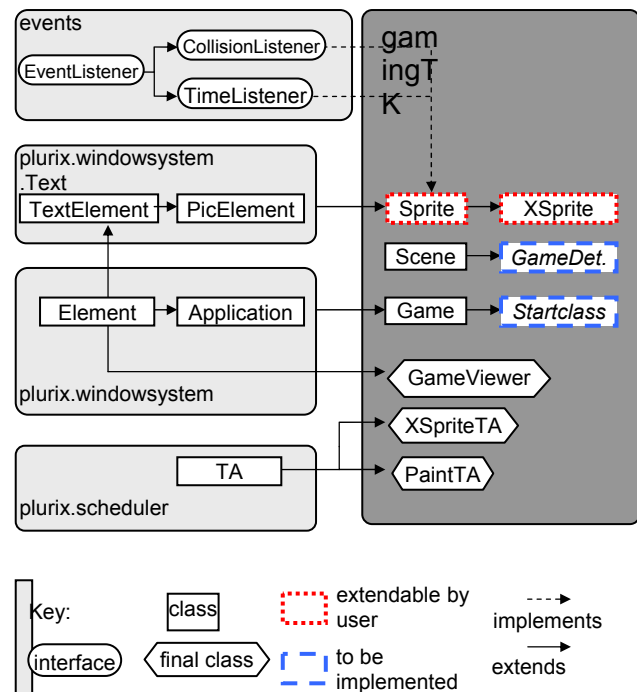


Figure 1. Plurix GUI & Gaming Framework Classes

The root element of the framework is the class *Sprite*. Sprites are unmovable objects, e.g. used for walls or background objects. They have a name and can be solid or permeable. Furthermore Sprites may have a texture or be invisible. Being immobile makes a collision detection and resolution for Sprites superfluous.

Mobile objects are implemented by the *XSprite* class, which extends *Sprite*. XSprites have two kinds of movement control: active control by the *XSprite* autonomous, or passive control by interaction of a player.

Mobile XSprites may collide and therefore appropriate collision detection and resolution need to be supported by the framework.

The method *Collision* of the class *CollisionScanner* is called after every move of a XSprite. It compares the new position of the XSprite to all other Sprites and XSprites. A *CollisionEvent* object will be created in case of a collision and the *CollisionEventDispatcher* dispatches it to engaged objects.

XSprites and Sprites can react on a CollisionEvent by implementing the interface *CollisionListener*. The framework basing on overlapping rectangles provides a simple collision detection strategy. Enhanced algorithms can easily be introduced by overwriting methods of the implemented interface *CollisionListener*.

The class *Scene* manages a graph consisting of linked Sprites and XSprites. This class is also responsible for administration of local and remote players.

The class *GameDefinition* - an extension of class *Scene* - defines the attributes of the Scene, Sprites, and XSprites. During startup of the game the components are created as defined in the *GameDefinition*. For example the textures of Sprites are chosen and the velocity of XSprites is adjusted.

The class *GameViewer* implements local graphic output. Here the local view of the Scene is computed and local user input is dispatched. Local screen updates are performed periodically by a paint transaction (see 3.3).

The class *Game* that extends the class *Application* of the Plurix GUI controls startup of each game. If the game is started for the first time the Scene with all its Elements is instantiated and registered in the cluster-wide name service. Otherwise the existing Scene referenced by a given path and name is used. A *GameViewer* is created subsequently to display the Scene.

### 3.3 Transaction-based Graphic Output

Currently, we have a proprietary graphic API that will be extended to the OpenGL standard. Direct calls from transactions into device drivers can cause severe problems in the case of aborts. If for example a command is submitted to the display adapter there is no possibility to undo this operation in the case of an abort and the abort situation will become visible on the screen.

Modern display adapters e.g. ATI Radeon use command streams which are processed concurrently. If an abort causes a broken or erroneous stream the device may lock up and a reboot of the node may become necessary.

Implementing an undo buffer on the display adapter or in main memory solves this problem. However, the first approach is not feasible especially in the case of 3D applications where memory on the display adapter is too small. The latter solution is very time consuming and would require reading out the total visible display buffer

at the beginning of each transaction to save it in main memory. Furthermore both strategies would be visible on the screen and would cause a flickering screen in case of a high abort frequency.

Therefore we decided to implement a solution that releases device commands only in the case of a commit. Of course this is only reasonable if transactions drawing on the screen are short. If a scene is very complex the programmer is responsible to partition the drawing into several transactions.

The heart of the deferred command concept is the *smart buffer* used to bridge the gap for data streams between kernel and transaction space. Smart buffers are unidirectional and output smart buffers are able to through away input of transactions in case of an abort and make contents visible only in case of a commit. Input smart buffers are used to process input events of interrupts which are consumed only in case of a commit and which are not lost if an abort occurs.

The buffer itself is stored in kernel space to protect it against shadow copying and undo effects. The desired behavior is achieved by the proper placement of the management variables.

An output buffer has two integer variables: a write-offset (residing in DSM) and a read-offset (stored in kernel space). The write-offset only change in case of a commit and the read-offset is adjusted only within kernel space. For the processing of command smart buffers a special routine is called in case of a successful commit to flush commands to the target device. Input smart buffers are built similar with swapped variables.

The graphics output smart buffer is filled by the transaction *PaintTA*. There is one instance of the *PaintTA* per game on each node extended the class *TA* defining the basics of each transaction. The *PaintTA* is executed periodically to refresh the game Scene in the Viewer. It forces the attached Viewer to do a repaint that subsequently calls all elements in the Scene graph to redraw them. The order of creation of Sprites and XSprites defines the level of the objects on the screen. Later instantiated Sprites respectively XSprites appear in front of earlier ones.

### 3.4 User Input

The user can control passive XSprites. In the current version of the framework there is one XSprite associated to a user, which can be controlled by the keyboard or the mouse. XSprites get their input events from the associated viewer of the GUI by implementing the *KeyListener* and *MouseListener* interfaces.

### 3.5 Sharing Scenes

As described in subsection 3.2 multi-player games can easily be implemented by publishing the Scene in the cluster-wide name service. The underlying DSM manages automatically distribution and replication of the Scene among all nodes accessing it.

The GameViewer is a node-private instance used to display a clipping of the shared Scene. Although this instance is allocated in the DSM it is not accessible by other nodes because it is not published in the name service. It can be used to store node-private information of the game that must not be shared.

All participating users are managed using an array attached to the shared Scene. The underlying DSM guarantees a consistent view on the shared Scene without any further efforts from the game developer.

### 3.6 Moving Objects

In traditional message passing systems the user input events or the new coordinates of a moved XSprite would be explicitly sent to all players.

In our system it is sufficient for the PaintTA to periodically refresh the Scene in the viewer. In case of a refresh any modified XSprites are automatically fetched via the DSM and the viewer is up to date. This approach benefits from the fact that if several node move XSprites in a small time interval all changes are fetched at once during the refresh without handling a lot of multicast message like necessary in traditional systems.

But we learned that XSprite positions must not be stored in the shared XSprites. Assume a XSprite  $XS$  is moved by a node  $n_1$  from position  $(x_1, y_1)$  to position  $(x_2, y_2)$  and immediately to  $(x_3, y_3)$ . If there is another node  $n_2$  that missed the movement to position  $(x_2, y_2)$  and is about to perform a redraw a problem occurs.  $XS$  is fetched by the DSM and node  $n_2$  erases the sprite at the most recent old position  $(x_2, y_2)$  and draws it at the new position  $(x_3, y_3)$ . Unfortunately, node  $n_2$  displays  $XS$  at position  $(x_1, y_1)$  and not at  $(x_2, y_2)$  because  $n_2$  missed one movement. Such a situation would disturb the Scene view on node  $n_2$ . Storing old XSprite positions within a list attached to the node-private game viewer solves this problem.

## 4. A Sample Multi-Player Application

In this section we present a teletennis game built with our gaming framework. Of course this is a very simple game application and we are planning to develop a more sophisticated game in future. Nevertheless, the teletennis allows an early evaluation of our gaming framework running on top of the Plurix transactional DSM.

The teletennis consumes only 256 lines of Java source code basing on our gaming framework implemented in 1'562 lines of source text.

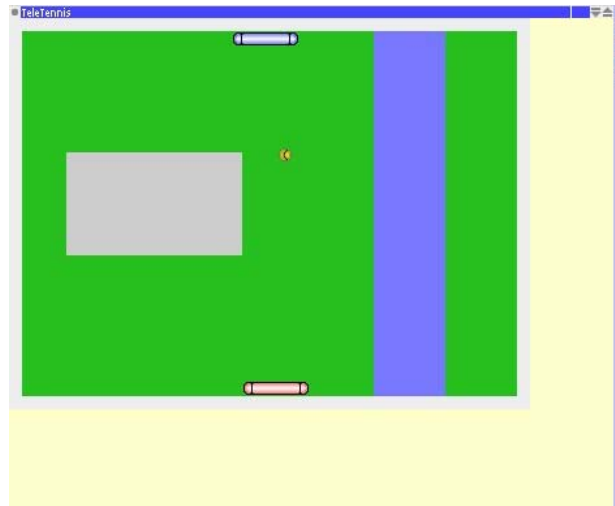


Figure 2. Teletennis Screenshot

A screenshot of the running teletennis game with two players is shown in figure-2. The teletennis allows up to four players each controlling a racket (passive XSprite). The blue rectangle is a non-solid object whereas the gray one on the left side is solid.

The performance evaluation is carried out on four PCs interconnected by Fast Ethernet Hub. Each node is equipped with a RLT8139 network card and an ATI Radeon graphic adapter.

Table 1. Node configuration

Node	CPU	RAM
1	Pentium 4 2.4 GHz	256 MB DDR RAM (266 MHz)
2	Athlon XP 2.2 1.8 GHz	256 MB DDR RAM (333 MHz)
3	Athlon XP 2.0 1.66 GHz	256 MB DDR RAM (333 MHz)
4	Celeron 1.8 GHz	256 MB DDR RAM (266 MHz)

During the measurements all rackets are automatically moved every 50ms (20 times per second). The PaintTA is executed on each node periodically every 40ms (25 times per second), doing a refresh of the scene on the screen. Additionally, on the first PC participating the game the ball is controlled by an active XSpriteTA executed every 50ms (20 times per second). Furthermore, an additional timer TA changes the shape of the ball 4 times per second.

The cluster-wide measurements are integrated in the memory consistency protocol and each TA is monitored in local memory. For each TA we log: page requests, written pages, average page latency, average token latency, and TA execution time. Latency is zero for requests that do not require communication.

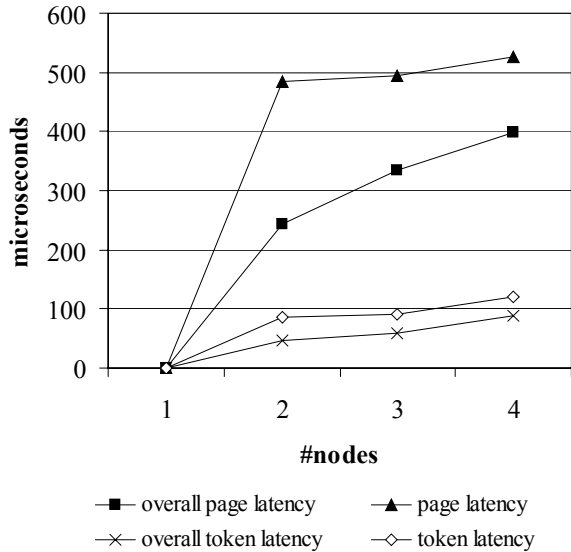


Figure 3. Cluster Latencies

Each measurement value in figure 3 is the average of all latencies of all nodes in the cluster. The overall page/token latency includes requests that can be satisfied locally whereas the other two only consider requests resulting in network transmissions. Of course latencies grow with the number of nodes in the cluster but they grow slowly and on a four-node cluster the token latency is 120μs and the page latency is 526μs.

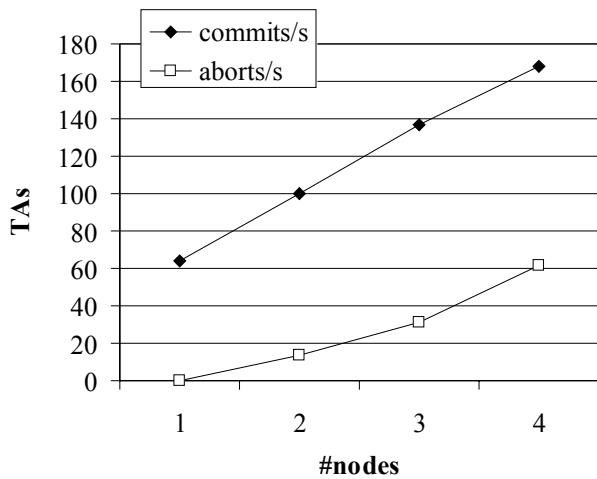


Figure 4. Cluster throughput

Figure 4 shows the total commits and aborts per second of the cluster. The first node executes more TAs than the other nodes because it processes the ball movement and the ball shape change. Collision rate grows with growing node number but the number of commits scales quite well.

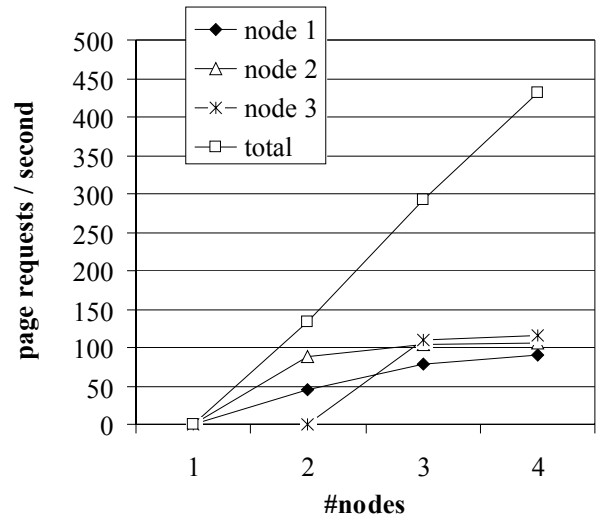


Figure 5. Page requests per second

The last chart in figure 5 shows the page request rate per second. In this chart only page requests resulting in network communications are considered. If there is only a single node there are no invalidations and no page requests are needed. With growing node numbers the page request rate grows up to ~110 req./s per node caused by invalidations. The request rate grows very slowly and the total page request rate in the cluster moves up to 431 or 1,7 MB/s in the case of four nodes.

Earlier tests (not with the gaming application) showed that in a LAN cluster of eight PCs connected by a Fast Ethernet up to 2'700 page transfers per second could be performed and 3'000 commits/s of empty TAs.

Of course our simple application is not well suited to evaluate scalability because it is limited to four players. Furthermore, the teletennis application has a high collision probability because each player displays the total scene on his machine. Multi-player games supporting a large number of concurrent users implement a lot of rooms and only smaller groups meet within a single room.

But the numbers gained during the measurements are encouraging and there are still a lot of resources left for more nodes and more players. The experiments with the teletennis with respect to the simplified development of distributed multi-player games are promising. The sharing of a scene including the consistency of moving objects is provided by the DSM and is no longer the burden of the programmer.

## 5. Related Work

There are several middleware solutions for supporting multi-player games and distributed virtual worlds that focus on providing a group communication facility. The Avocado system is such an example implementing a group communication system for state consistency [10]. It offers a framework for distributed virtual reality applications. A shared scene graph is built using predefined C++ objects that can be extended or adapted. State consistency is maintained for explicitly declared object fields that are accessed by special get and put methods. The programmer must declare field connections (allowed for fields of the same type) to propagate changes of a field. Assume a source field  $s$  is connected to a destination field  $d$ .

If  $s$  is changed the value is immediately forwarded to  $d$ . Of course inserting the correct and efficient connections is the burden of the programmer not necessary in our DSM-based solution.

To the best of our knowledge neither a gaming framework nor a game has been implemented on top of a DSM system.

## 6. Conclusions

We have presented a gaming framework built on top of a transactional DSM Operating System. Existing solutions based on message passing still require significant efforts from the programmer to implement consistency of the shared scene graph.

Using a DSM for building a gaming framework relieves the programmer from the burden of distribution and consistency management. Instead of explicitly sending modified data (e.g. coordinates of a moving avatar) they are fetched on demand by a node during redraw of the shared scene. Using DSM-based communication a slower node automatically drops missed movements of objects and simply displays the newest position during the next redraw. Herewith we gain simplicity for the development of multi-user games with respect to consistency of distributed shared scene graphs.

We evaluated our gaming framework with a simple multi-player teletennis game. The measurement results showed that latencies grow slowly with growing cluster sizes and the overall cluster throughput scales well. Network traffic caused by page requests also grows slowly with more players.

We plan to develop a virtual world application on top of our gaming framework to study scalability for more nodes. In that application field not all participants are always located in the same area and therefore collisions are seldom. Synchronization is automatically limited to

those objects in the same area because of the implicit fetching of data during a redraw.

## 7. References

- [1] J.L. Keedy, and D. A. Abramson, "Implementing a large virtual memory in a Distributed Computing System", *Proceedings of the 18<sup>th</sup> Annual Hawaii International Conference on System Sciences*, 1985.
- [2] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing", *Proceedings of the International Conference on Parallel Processing*, 1988.
- [3] M. R. Eskicioglu, "A comprehensive bibliography of distributed shared memory", Technical Report TR96-17, Department of Computing Science, University of Alberta, 1996.
- [4] M. L. Scott, W. J. Bolosky, "False sharing and its effect on shared memory performance", Technical Report MSR-TR-93-01, Microsoft Research, One Microsoft Way, Redmond, WA 98052, 1993.
- [5] S. Traub, *Speicherverwaltung u. Kollisionsbeandlung in transaktionsbasierten verteilten Betriebssystemen*, PhD thesis, University of Ulm, Germany, Distributed Systems Department, 1996.
- [6] D. Mosberger, "Memory Consistency Models", Tech. Report, TR93/11, Department of Computer Science, University of Arizona, USA, 1993.
- [7] M. Wende, M. Schoettner, R. Goeckelmann, T. Bindhammer, P. Schulthess, "Optimistic Synchronization and Transactional Consistency", *Proceedings of the 4<sup>th</sup> International Workshop on Software Distributed Shared Memory*, Berlin, Germany, 2002.
- [8] N. Wirth, and J. Gutknecht, *Project Oberon – The Design of an Operating System and a Compiler*, Addison-Wesley, 1992.
- [9] K. Gharachorloo, "The Plight of Software Distributed Shared Memory", *Workshop on Software Distributed Shared Memory*, Rhodes, Greece, 1999.
- [10] H. Tramberend, "Avocado: A Distributed Virtual Reality Framework", *IEEE Virtual Reality*, Texas, USA, 1999.
- [13] M.P. Atkinson, "Orthogonal Persistent Object Systems", *Very Large Data Bases Journal*, 4 (3), pp. 319-340, 1995.

[14] C. Carlsson and O. Hagsand, "DIVE – A Platform for Multi-user Virtual Environments", *Computer and Graphics*, 17 (6), pp. 663-669, 1993.

[15] C. Morin and I. Puaut, "A Survey of Recoverable Distributed Shared Virtual Memory Systems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. ), September 1997.

[16] M. Schoettner, O. Marquardt, M. Wende, and P. Schulthess, "Implementation of the Java language in a persistent DSM Operating System", *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, 1999.

[17] M. Pizka and C. Rehn, "Heaps and Stacks in Distributed Shared Memory", *IEEE Parallel and Distributed Processing Symposium*, Ft. Lauderdale, USA, 2002.

[18] R. Goeckelmann, M. Schoettner, M. Wende, T. Bindhammer, and P. Schulthess, „Bootstrapping and Startup of an object-oriented Operating System”, *European Conference on Object-Oriented Programming - Workshop on Object-Oriented Programming and Operating Systems*, Malaga, Spain, 2002

[19] M. Schoettner, O. Marquardt, M. Wende, N. Link, and P. Schulthess, "Compiling in a Persistent Distributed Shared Memory Environment", *Proceedings of the 7th International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, 2001.