Electronic Communications of the EASST
Volume 17 (2009)

Workshops der
Wissenschaftlichen Konferenz
Kommunikation in Verteilten Systemen 2009
(WowKiVS 2009)

# Massively Multiuser Virtual Environments using Object Based Sharing

Michael Sonnenfroh, Kim-Thomas Möller, Marc-Florian Müller, Michael Schöttner and Peter Schulthess

12 pages

Guest Editors: M. Wagner, D. Hogrefe, K. Geihs, K. David
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

# Massively Multiuser Virtual Environments using Object Based Sharing

**Michael Sonnenfroh**[1], **Kim-Thomas Möller**[2], **Marc-Florian Müller**[2], **Michael Schöttner**[2] **and Peter Schulthess**[5]

[1] Michael.Sonnenfroh@uni-ulm.de
[2]http://www.cs.uni-duesseldorf.de/AG/BS
Institut für Informatik
Universität Düsseldorf

[5] Peter.Schulthess@uni-ulm.de
http://www-vs.uni-ulm.de
Institut für Verteilte Systeme
Universität Ulm

**Abstract:** Massively multiuser virtual environments (MMVEs) are becoming increasingly popular with millions of users. Commercial implementations typically rely on a traditional client/server architecture controlling the virtual world state of shared data at a central point. Message passing mechanisms are used to communicate state changes to the clients. For scalability reasons our approach creates and deploys MMVEs in a peer-to-peer (P2P) fashion. We use standard Java technology implementing only a few basic data-centric operations for the management of our distributed objects. Higher consistency models can easily be implemented using these basic operations. Currently, we have implemented transactional consistency offering convenient and consistent access to the shared scene graph. In this paper we describe our basic object model and the prototype implementation TGOS (Typed Grid Object Sharing). Furthermore, we discuss preliminary measurements with the virtual world Wissenheim executed on top of TGOS.

**Keywords:** MMVE, distributed objects, multiple consistencies

## 1 Introduction

Massively multiuser virtual environments including games and online getaways are attracting more and more people, e.g. Second Life exceeded 9 millions of users at the end of the year 2007. Most of these systems are based upon classical client-server architectures regarding the communications topology and the programming model. Providers often argue that client-server systems are superior in controlling user accounts and cheating. The installed servers are rather huge clusters which introduce latency and scalability constraints. As a consequence providers typically host different virtual worlds for each continent.

In 2004 the Wissenheim project was started to explore virtual presence applications using transactional memory for distribution. Using transactional memory to distribute and share a scene graph allowed us to deploy a simpler programming model in the field of virtual world development abstracting from the structure of the underlying network. Evolving from the original Wissenheim project we have developed a more flexible and portable object model and system, based on sharing objects. The Typed Grid Object Sharing system (TGOS) allows Wissenheim to use a data centric view to coordinate distributed access to virtual worlds. An important design aspect during the development was the requirement to cope with the challenges of wide area deployment such as high latency and potential node failures. The opportunity to use our approach for peer-to-peer deployment [13] was another aspect not only influencing the network layer but the programming model as well. Visitors may test the prototype shown in Figure 2 at the Wissenheim website [1].

This paper is organised as follows. In section 2 we describe the TGOS object model and in the following section its prototype implementation. In section 4 we present transactional consistency as a method to keep replicated objects consistent. How Wissenheim has been adapted to run on top of TGOS is described in section 5. Subsequently, we present preliminary measurement results followed by related work and conclusions.

## 2 Typed Grid Object Sharing

The object model of TGOS is divided into two different parts called the object view and the global data store as shown in Figure 1. The object view provides an object oriented and type-safe local-view of distributed objects. Object views on different nodes can show different subsets of objects with different data versions.

The global data store provides storage and distribution mechanisms for data blocks of arbitrary size. The data store is unaware of the internal structure of the data blocks it is managing. This allows different implementations for a global data store. Each data block is tagged by a unique and persistent ID provided by the global data store. In contrast to the object view where different object versions can be found the global data store provides a consistent view over all data blocks storing only one version per data block.

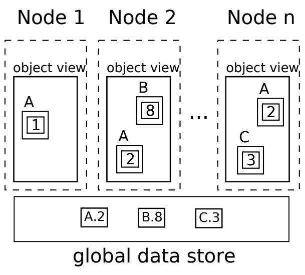


Figure 1: Object Model schematics.

Figure 2: Wissenheim Worlds screenshot.

## 2.1 Object View

Each object view is able to transform an object into a serialised form and merge serialised data into another object of the same type. Each object of a view is connected to exactly one data block in the global store identified by the global ID. Although different nodes have different views and different data versions the type system remains unchanged on all views. We are saying that an object *A* residing in object view *N* is *alike* to object *B* residing in object view *N+1* if both objects have the same ID. Two objects are called *equal* if they are alike and if they have the same version of data.

## 2.2 Global Data Store

The global data store (short: data store) can be described as a basic replication layer responsible for distributing arbitrary data blocks between nodes. It provides functions to read, write, invalidate, and lock data blocks it is managing. The granularity of a data store operation is at least one data block meaning that for example a write is updating one or multiple blocks. The data store also ensures that all write operation are performed atomically preventing, in case of a failure during block transfer, that corrupted data gets visible. All write accesses performed on the store are ordered by FIFO consistency [12]. Lock acquisitions are made on a first-come-first-served basis. Beside the replication management the data store is also responsible for generating persistent IDs for newly created data blocks. The data block size is determined by the size of the serialised object and the data format used by the object view and therefore not defined by the data store. The implementation of a global data store is not defined by our object model but for scalability reasons we recommend a peer-to-peer approach.

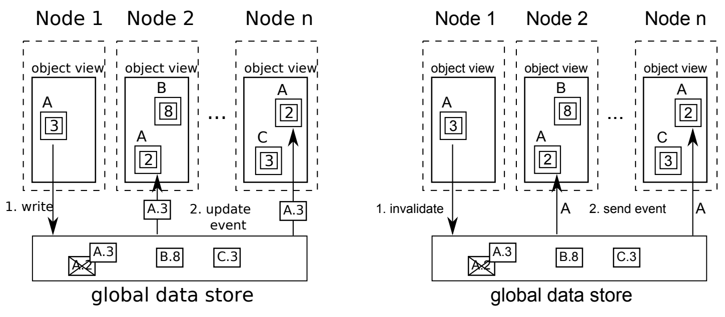
## 2.3 Events

The object views can receive asynchronous events from the global data store informing them about data changes of objects residing in their view. We have defined an *update* event and an *invalidate* event (shown in Figure 3) which are triggered through a data store write or an invalidate respectively. When the *update* event is triggered the object view receives the newest version of the written data block for an object from the data store. In case of an *invalidate* event the object view will receive only a notification that a data block corresponding to an object in the view has changed.

## 2.4 Basic Operations

The object view has a small set of basic operations which can be performed on objects to modify their corresponding data block in the global data store. With these basic operations different consistency models can easily be built inside the object model.

**Definition 1** (*push:O*) The *push:O* operation will extract the data of the given object *O* and write it back to the global object store. Views on other nodes will receive an update event. The push operation will block until the object data has been extracted and written to the data store.



(a) Update event. (b) Invalidation event.

Figure 3: Event mechanisms.

**Definition 2** (*push:O:B*) The bounded push operation extracts the data of object O but defers writing to the global data store until the bounded object *B* is pushed. All object data is then written to the data store in one data store write.
Remark: Bounded push operations are useful for creating an atomic push of multiple objects.

**Definition 3** (*inv:O, inv:O:B*) Corresponding to the push operations we have defined invalidate operations which have the same semantics beside that they will not generate an update event but an invalidate event at the other views instead.

**Definition 4** (*pull:O*) To update an object in the object view we have the *pull* operation which will request the latest data block for *O* from the data store and merge it. The operation is performed synchronously meaning that it will block until object *O* has been updated.

**Definition 5** (*order:O*) For asynchronous object updates we are providing the *order* operation which requests the newest data for object *O*. The data store will transfer the new data via an update event concurrently.

**Definition 6** (*sync:O*) The *sync* operation will request an advisory lock for the data block corresponding to the object O. The operation will block until the request can be granted. The lock on *O* is removed by a *push:O* operation of the view which holds the lock.

## 2.5 Naming Service

Because the object views are accessing objects in a type-safe manner references to objects cannot be manufactured. To get an object currently not available inside a view there has to be a root object from which the desired object can be accessed. Such a root object could be created automatically by each view at startup. But this would force systems without reference tracking to fully replicate every accessible object or at least to create proxies for every accessible object. Therefore we have integrated a hierarchical naming service into our object model itself avoiding proxies or full replication. It is also useful for partitioning the accessible object graphs while still allowing access. For this purpose we have defined two basic naming service operations.

**Definition 7** (*put:O:K*) This operation stores an object *O* into the name service with a given textual key *K*.

**Definition 8** (*O:get:K*) A *get* operation retrieves the object stored under a key *K*.

## 3 TGOS Prototype

All the features defined by our theoretical model have been implemented using Sun Java 1.5. Figure 4 gives a short overview over the components implemented for the TGOS model.

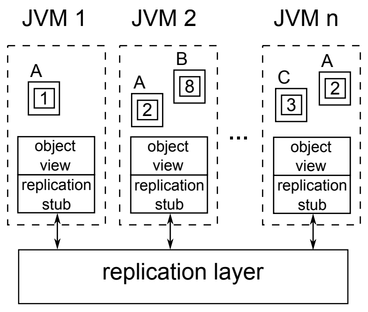


Figure 4: Prototype schematics.

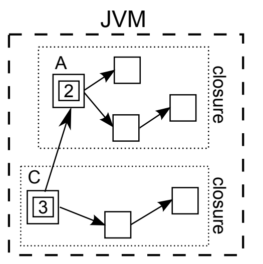


Figure 5: Closure Building.

### 3.1 Object Extraction and Merging

As mentioned before the global data store is working on binary data and is totally unaware of the object model and type system used by the application. This makes it necessary to provide a way to serialise objects to a binary form and vica versa. Java is already providing a native serialisation facility for transforming objects into a binary form. But the fact that Java creates new objects during the deserialisation process rendered it unusable for our approach. To fulfil the requirements of our object model we needed a mechanism to extract and to merge the data of an object *A* with the data of an local object *B* keeping the local references to the objects untouched. Therefore, we have created a new serialisation/merging facility using the Java Reflection API. Our implementation is capable of extracting the binary data and merging this data into another Java object of the same type in a similar way used by code versioning systems such as SVN.

### 3.2 Closure Building

The object view provides an automatic closure building mechanism by building the transitive closure over all references distinguishing between global and local objects. The solution provided with the implementation for the proposed closure building as shown in Figure 5 is based upon inheritance mechanisms provided in most type safe and object-oriented languages. All objects applicable to the basic operations must be a subtype of the class *SharedObject*. So every reference from one shared object to another shared object is treated as a global reference, every other reference is treated as local reference. When an object gets pushed the local references of a shared object are pushed using a copy by value semantic while references to shared objects use a copy by reference semantic. With this feature it is easily possible for a programmer to build object hulls with arbitrary size. Using inheritance for marking objects with globally unique references is not an optimal solution and was chosen to simplify the implementation in the early stages of development. An alternative solution would be to use interfaces for marking these shared objects.

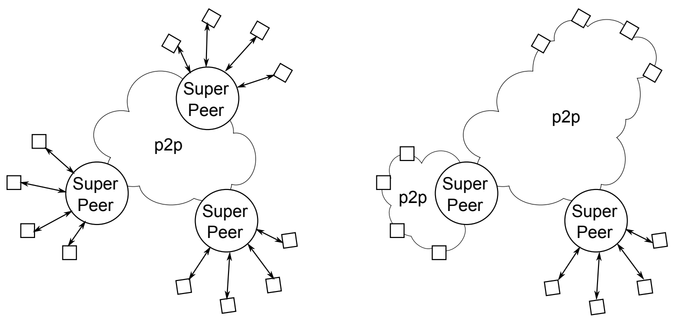
## 3.3 Basic Event Handling

Currently, the object model defines two events which have been implemented in TGOS. The update event is triggered when a node has pushed an object and the updated object data is arriving. The registered event method can now decide what to do with the updated object data: whether to integrate it fully, partial or not at all. It is also possible to store the updated information and integrate it at a later time which is used by the later described transactional consistency to keep the object views consistent at all times.

When an invalidate event is triggered the globally unique object ID of the invalidated object is transmitted. Thus the application can pull a new version of the object or delete the object from the local object view.

## 3.4 Scaleable Global Data Store

Our implementation of the global data store is using a hierarchical approach as shown in Figure 6(a). Communication between nodes on the lower hierarchy level is implemented by a client-server approach. The servers or *SuperPeers* are interconnected on a peer-to-peer basis to improve scalability. This approach was taken for simplicity and to allow rapid development in the early stages. Due to our peer-to-peer object model it is possible to change the client-server model of a *SuperPeer* to a peer-to-peer based or mixed approach without altering the application. Figure 6(b) shows a feasible setup planned for the further development of Wissenheim Worlds.



(a) current (b) planned

Figure 6: Scaleable Global Data Store Implementation.

## 4 Consistency Models

By default the object model is providing no consistency other than the FIFO consistency defined by the global data store. Stronger consistency models can be implemented using the mechanisms provided in an object-oriented fashion within the object model. The basic operations of TGOS simplify the adaption to different consistency requirements depending on the use case. Stronger consistency models can be implemented as add-on libraries, too.

### 4.1 Transactional Consistency

Transactions are well-known in data-base applications but are hardly used by highly interactive applications. A first approach for using transactions in distributed interactive applications was the original Wissenheim [4] running on top of a distributed operating system called Plurix. The Plurix OS was designed to run on standard PC hardware and was used in a local network environment. However using transactions for highly volatile data in a system with high latencies like in a wide area network will lead to a sizeable overhead and will limit scalability. Synchronised access to data which is often read but only rarely changed presents an ideal scenario for transactions. Wissenheim is using transactions for synchronising read and write accesses to the scene graph structure, virtual items, and objects which are only accessible by one person at a time. These accesses are relatively rare and can therefore be synchronised via transactions without major performance drawbacks. Because of the limited collision probability and the latency issue we are using an optimistic transactional approach [15]. Our approach to optimistic transactions is using a backup mechanism to save objects before they are modified by a transaction to be able to rollback the object state in case of an abort.

### 4.2 Transaction-based Programming Model

To use transactions in the program flow we need a way to specify a transactional block. The start of such a block is defined by the *begin* operation. The *commit* operation is used to end a started transactional block and to validate all changes. Finally, the *abort* operation is provided to voluntarily abort a transaction and thus undo all changes made within the started transaction block. Rolling back a transaction requires to save backups of objects that are modified during a transaction. For this purpose we are providing the *add2Transaction O* operation which will create a backup of object *O*. The rollback mechanism is using the same seralization mechanisms used by the object view.

```
do {
   begin();
   add2Transaction(obj);
   obj.doStuff();
   if (wrong)  abort();
} while (!commit());
```

Figure 7: Basic transactional loop.

```
class Token {
  int taCount;
  int writeSets[][] =
          new int[128][];
}
```

Figure 8: Token object.

A standard transaction is shown in Figure 7. The *do..while* loop is used to restart the transaction in case the commit fails and we want to restart. To serialise our optimistic transactions we are using a token-based mechanism. Each transaction wishing to commit has to acquire the shared token. The token is implemented as ”normal” distributed object managed by TGOS. Figure 8 shows the content of the token object. The *taNumber* member is a steadily increasing number of transactions committed. Access to the token object is synchronised via the *sync* operation provided by the *object view*. The writeset information is used to check if the committing transactions is colliding with any data altered by a previous transaction. In case of a conflict a rollback is performed by restoring the object backups. With these four methods it is possible to work transactionally on any object shared by our object model.

To avoid making the backup operation for objects used inside the transactional block manually the AspectJ framework [3, 7] can be used. Creating AspectJ pointcuts which observe the access to objects which are descendants of *SharedObject* can automate the backup creation. The use of AspectJ is transparent for the implementation of the transactional consistency and can be used with any implementation.

Removing the *do..while* loop is a more difficult task because in a standard JVM we cannot use labels or access the program counter to jump back to the begin of the transactional block in case of an abort. Extending the JVM however would allow us to implement the transactional block in a more transparent fashion alike the Java *synchronized* blocks. But extending a JVM would exclude inexperienced users or users on strictly secured workstation from accessing Wissenheim due to the inability to install or launch a modified JVM.

## 5 Wissenheim Worlds

Wissenheim is designed for edutainment combining interactive teaching content with entertaining games. Currently, it is used to support lectures at the University of Ulm by providing interactive exercises. Wissenheim is running on three platforms: Plurix (the original transactional operating system), Linux (using the Object Sharing Service provided by the XtreemOS project [14]), Standard Java (using the TGOS system described in this paper). Wissenheim Worlds [1] is an extension of the Java version of Wissenheim aiming at supporting a huge number of players and many scenarios. All virtual worlds are accessible via browser applets supporting Windows Vista/XP 64-bit and 32-bit, MacOS & MacOS X, Linux 32-bit and Linux 64-bit with working Sun Java VM. By using standard Java applets we are able to launch Wissenheim without the need to install any specific client software.

### 5.1 Scene Graph

Wissenheim is using a distributed scene graph [5] to share the scene information among the users. Access to the scene graph structure is synchronised using transactional consistency. Graphical data, position information and other volatile data is accessed using weak or scene-specifing consistency constraints to keep latency issues at a minimum. Therefore, the virtual world is subdivided into disjunct scenes with their own scene graph and a separate transactional token. As a result each scene is independent from a consistency point of view and thus subdivides the network traffic on a scene basis. A user can be connected to one scene only but each node can access and modify the scene graph content and structure.

### 5.2 TGOS Integration

Wissenheim Worlds is using the TGOS service in a transparent way. The basic Wissenheim application is unaware of the different services the replication layer provides for distribution. Wissenheim is working solely on objects accessing scene and avatar data by either global references or naming service calls. The implementation of the global data store is automatically taking care of redirecting request, transparently handing over connections from one service to another. This gives the developer of a scene the opportunity to create his scene using an abstract network model.

### 5.3 Transactions in Wissenheim

Using a data centric approach for data management allows each client to directly access and modify the shared scene graph. Thus may lead to race conditions especially if the structure of the graph is modified by two or more nodes concurrently. To synchronise concurrent accesses within Wissenheim we are using the proposed transactional consistency for critical sections. Thus we are using transactions on a fine-grained basis synchronising only critical program parts, e.g. when avatars are joining to a scene or are grabbing items.

## 6 Measurements

Due to the fact that the implementation for the global data stores is interchangeable, latency characteristics can change as well. The remaining (constant) overhead is caused by the object view and serialiser implementation. For our measurements we have used some very common situations in Wissenheim Worlds to reproduce results as close as possible to real world situations. We have examined the serialisation overhead and the ratio of serialised data size to effective data size. For the calculations of the effective data size we are using a very strict definition counting only the size of members or array elements containing useful data. References or other structural members are ignored here. So the effective data is the most compact (uncompressed) form in which the information the object incorporates can be saved. Our measurements were made on an Intel Pentium D 2.66GHz with 1GB Ram running Windows XP SP3 using Sun Java 1.6.

```
class Transformation {
   Vect3D translation;
   Vect3D rotation;
   Vect3D scaling;
   Matrix4D transMatrix;
   Matrix4D baseMatrix;
   Matrix4D invWorld;
}
```

(a)

```
class Matrix4D {
   float [16] m;
}
```

(b)

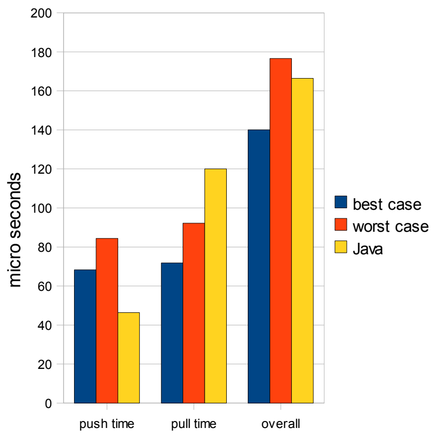
```
class Vect3D {
   float x,y,z;
}
```

(c)

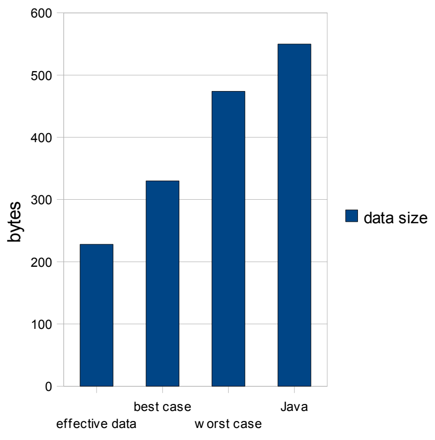
Figure 9: Classes used for measurements.

Figure 9 shows the classes used for measurements, the *Transformation* class shown in 9(a) is used by Wissenheim Worlds to position and orientate any virtual object and is therefore updated whenever a movement occurs. As one can see the *Transformation* class consists only of references which might need their class names saved during the serialisation process. Our serialiser optimises this process by omitting the class names when the type of the member variable corresponds exactly to the type of the object it is referencing. The first measurement shown in Figure 10 presents the average time to serialise and merge a transformation object and the size of the serialised data for best and worst case. Best and worst case differ in the compaction the serialiser can perform on class names needed for de/serialising references. For comparison we are presenting the performance of the native Java serialisation facility as well. In average the serialiser is at least as fast as the native Java one, in most cases even faster.

Furthermore, we have tested the optimistic transactions over wide area and WLAN connections with a round trip time of approximately *80ms*. The average number of transactions per second was about *12* which correlates exactly to the round trip time.



(a)



(b)

Figure 10: Serialisation measurements.

## 7 Related Work

Massively multiuser virtual environments such as Wissenheim Worlds refer to a wide variety of different topics. Due to limited space we are comparing our ideas only with a limited assortment including MMVEs, scene graphs, and transactional memory.

A large number of architectures and formats for scene graphs have been already proposed. Popular ones such as VRML or Java3D are unfortunately designed for single station use and lack the possibility to be used in a distributed manner. While distributed virtual environment systems such as Avocado and DIVE or distributed scene graphs such as blue-c [11] or the Distributed Open Inventor [9] present a comparable design by means of scene graph distribution, they lack support for transactional consistency. The fine-grained application-based control of the replication process and the ability to distribute scene graphs which structures and classes are completely application specific are another novelty of the TGOS approach. The requirements and motivation for a peer-to-peer architecture has been described by Schiele et al. [13] and a description of a consistency model suitable for MMVEs has been proposed by Hähner et al.[8].

Transactions are a key concept in database management systems providing significant benefit for concurrent data base access [6]. The TGOS work on optimistic transaction has been strongly influenced by the concepts proposed by H.T.Kung et al. [10]. Related work on software transactional memory (STM) has be done by Wende [15] and STMs for large scale clusters are described by Bocchino et al.[2]. Although many good ideas have been adopted from these publications the TGOS approach differs in the many properties, e.g. transactions used for an MMVE, network environments, etc.

## 8 Conclusions & Future Work

The proposed TGOS object model shows a new way of creating and sharing MMVEs by using a fine-grained data-centric approach. By allowing a peer-to-peer oriented programming model at the object level we can easily use different replication strategies customized for different distribution scenarios. The basic operations defined by the TGOS model are the building blocks for different consistency models. The Java-based TGOS implementation includes transactional consistency providing a promising platform for virtual worlds. TGOS has been tested hosting Wissenheim for supporting lectures at University of Ulm. We are encouraged by the feedback from beta users and by the preliminary measurement results presented in this paper.

Future work includes improved support of heterogeneity by allowing user-hosted scenes and the integration of more P2P like replication mechanisms. A framework for secure access to scenes and controlled avatar customization are the next steps. We plan to collect statistical data on user behaviour and load distribution when more avatars will show up in Wissenheim Worlds.

## Bibliography

[1] Wissenheim worlds, www.wissenheim.de, 2008.

[2] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 247–258, New York, NY, USA, 2008. ACM.

[3] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[4] M. Fakler, S. Frenz, R. Gockelmann, M. Schoettner, and P. Schulthess. An interactive 3d world built on a transactional operating system. *Electrical and Computer Engineering, 2005. Canadian Conference on*, pages 235–238, May 2005.

[5] Markus Fakler, S. Frenz, M. Schoettner, and P. Schulthess. A demand-driven approach for a distributed virtual environment. *Electrical and Computer Engineering, 2006. CCECE '06. Canadian Conference on*, pages 1538–1541, May 2006.

[6] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[7] Jeffrey Palmand William G. Griswold Gregor Kiczales Erik Hilsdale Jim Hugunin Mik Kersten. An overview of aspectj. In *ECOOP 2001  Object-Oriented Programming*, volume Volume 2072/-1 / 2001, pages 327–354. Springer Berlin / Heidelberg, 2001.

[8] J. Haehner, K. Rothermel, and C. Becker. Update-linearizability: a consistency concept for the chronological ordering of events in manets. *Mobile Ad-hoc and Sensor Systems, 2004 IEEE International Conference on*, pages 1–10, Oct. 2004.

[9] Gerd Hesina, Dieter Schmalstieg, Anton Furhmann, and Werner Purgathofer. Distributed open inventor: a practical approach to distributed 3d graphics. In *VRST '99: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 74–81, New York, NY, USA, 1999. ACM.

[10] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[11] Martin Naef, Edouard Lamboray, Oliver Staadt, and Markus Gross. The blue-c distributed scene graph. In *In Proceedings of the IPT/EGVE Workshop 2003*, pages 125–133. Press, 2003.

[12] J. Sandberg R. Lipton. Pram: A scalable shared memory. Technical report, Princeton, 1988.

[13] Gregor Schiele, Richard Suselbeck, Arno Wacker, Jorg Hahner, Christian Becker, and Torben Weis. Requirements of peer-to-peer-based massively multiplayer online gaming. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 773–782, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Marc-Florian Mueller Kim-Thomas Moeller Michael Sonnenfroh Michael Schoettner. Transactional data sharing in grids. In *Proceedings of the International Conference on Parallel and Distributed Computing and Systems*, Orlando, USA, 2008. IASTED Computer Society.

[15] M. Wende, M. Schoettner, R. Goeckelmann, T. Bindhammer, and P. Schulthess. Optimistic synchronization and transactional consistency. *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*, pages 331–331, May 2002.