Synchronization Methods for Supporting Distributed 3D Virtual Environments in Java™

Kevin Gorman
Dept. of Electrical and Comp. Eng.
University of Vermont
Burlington, VT 05405, USA
1-802-769-0429
kgorman@emba.uvm.edu

Daneyand Singley
Dept. of Electrical and Comp. Eng.
University of Vermont
Burlington, VT 05405, USA
1-802-769-1690
dsingley@zoo.uvm.edu

Yuichi Motai
Dept. of Electrical and Comp. Eng.
University of Vermont
Burlington, VT 05405, USA
1-802-656-9660
ymotai@emba.uvm.edu

ABSTRACT
This paper presents a 3D distributed virtual environment (DVE). The DVE, created using Java, is intended to allow for the creation of applications to enable multiple users to collaboratively interact in, and communicate about, a virtual world. This DVE is used to primarily support a flight simulator application. Three client-server synchronization methods were developed and compared. The method of transparent synchronization with non-blocking I/O has been found to be the best method for maintaining synchronization among DVEs. The successful creation of the DVE and the positive results collected in the supporting experimental data, leads to the conclusion that Java, with the addition of the non-blocking I/O and 3D API's, can be successfully used to create high performance 3D DVEs.

Categories and Subject Descriptors
C.2.4 **[Computer-Communication Networks]**: Distributed Systems – *client/server;* D.3.3 **[Programming Languages]**: Language Constructs and Features – *Input/Output.*

General Terms
Design, Human Factors, Languages, Measurement, Performance.

Keywords
Java, Non-blocking I/O, Synchronization, Virtual Environments.

1. INTRODUCTION
Distributed virtual environments (DVEs) enable geographically diverse users to communicate about and collaborate on various aspects of a virtual world [3]. A 3D DVE can find many applications, and an environment that is cost effective, portable, maintainable and extensible would be of particular use in both entertainment and academia [3, 4]. However, for applications to be successful they need to be immersive and interactive, and should be founded on a DVE infrastructure that provides graphic, control, communication, and networking support [3]. Models for building a DVE with strong performance across a network have been presented [4]. A particular difficulty in dealing with network communication between DVEs is maintaining synchronization between the server and the client applications [5, 8, 9]. Multi-user applications that do not provide for relatively consistent and synchronized worldviews run the risk of losing their interactivity and immersion [9]. It would be highly desirable to utilize tools and a methodology, which, along with the attributes listed above, can also facilitate the creation of a 3D DVE that provides good network performance.

Java has been used to provide support for the creation of 3D DVEs in the past [4] and the Java Adaptive Dynamic Environment (JADE) has been proposed to provide support for dynamic extensibility for virtual environments (VEs) in run-time [6]. Those concerned with entertainment oriented DVEs readily appreciate the many benefits of using Java. Java provides many standard extensions and API's that enhance developers' abilities to create graphics, control, and communication routines for their applications [7]. However, there have been issues with using Java to support DVEs. An attempt to recreate a DVE in Java that was originally created using C++ highlighted a problem with Java's networking support. Networking support was finally created via non-blocking and polled socket I/O in C++ and Java was provided with a channel class to communicate with the C++ support code. The difficulties encountered led to the conclusion that networking services that are only capable of blocking calls are not sufficient in complex real-time applications [1].

The Java networking support issues led to the development of custom non-blocking I/O API's which seemed to address many of the deficiencies with the original Java implementation [2]. The official Java standard has since been updated to provide support for non-blocking I/O starting with the Java 1.4 release. However, limited work has been done to compare and contrast threaded (blocking) networking services with the new non-blocking I/O API's. Java SDK 1.4.1 and the new Java.NIO.channels API, which provides support for non-blocking I/O, along with the Java 3D API (version 1.3.1), together with VRML97 model support, allow a 3D DVE to be created in a cost effective manner that can support easy maintenance and extension.

Described in this paper is a DVE, complete with robust networking routines, which supports many easily modifiable

Java and Java 3D are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

applications. The DVE, dubbed Whiskey, is designed to primarily support an application that can be used in an academic setting to allow for the collaborative exploration of Digital Elevation Maps through a flight simulator-like interface. This particular application of the DVE allows for a number of users to simultaneously investigate geographic phenomena in an open ended learning environment, communicate about their impressions and adapt their viewpoint and orientation in response to other users' inputs in real time.

2. DESIGN OF THE DVE IN JAVA

The DVE consists primarily of a "Flight Engine" which can be broken into four main pieces: "Graphical Engine", "Networking Support", "Control System", and "User Communications" (see Figure 1). The flight engine communicates with "I/O Handler" sub-routines which handle the interfaces to the input and output devices, while the "Networking Interface" sub-routines handle the marshalling/un-marshalling of data and the interactions with the socket itself. The network relies on multicast communication via a TCP/IP link. The main issue in creating the DVE in Java is maintaining the synchronization between the clients and the server. The following sub-sections describe in detail three networking support/interface options.

2.1 Server Based Sync with Threads

The first synchronization method implemented to provide networking support was a server based scheme that uses threads (see Figure 2). The server periodically collects information about each user's avatar into a class used for flight data management and transmits the class directly as a packet, with no re-formatting, to a client via Java's support for object input/output streams. The frames per second setting determines the frequency with which the packets are sent. The client machine is running a thread that is periodically reading from its socket. As data arrives it is read by the thread, which returns the packet data. This data is then copied to a FIFO, or "syncQ" buffer which is filled with flight data management classes as they are filled with data from the server. The client machine then pulls the next packet from the FIFO and uses the data in the packet to update the local environment information whenever the display needs to be updated.

The issue with this method is that the thread will wait until data is available prior to returning and allowing the syncQ FIFO to update. This allows the client's environment to diverge from the server's environment over time. Synchronization is maintained by a re-sync command that the server issues. A re-sync is periodically transmitted to each client that indicates which packet they should be using to update the display. If the client is not properly synchronized the client's FIFO is emptied and the packet that they received with the re-sync command is drawn immediately. In non-ideal conditions, however, a number of packets will be queued while waiting for a re-sync command. Under some conditions the thread will be left waiting for new packets and the display will just be continually updated with the last frame that was received. These circumstances will cause the objects in the client's VE to move in a stuttering fashion when the display is updated after a poorly timed re-sync, or a long wait for data.

2.2 Server Based Sync with Non-Blocking I/O

The next synchronization method that was examined takes advantage of a new feature of Java: support for non-blocking I/O (see Figure 3, CLIENT0). Non-blocking I/O supports the ability to immediately terminate the socket read and return if there is no data available, allowing the client's environment to continue executing the application immediately afterwards. Like the previous method, synchronization is still maintained via a re-sync command. However, non-blocking I/O does not support the transmission of data in discrete packets of flight data management classes, because object input/output streams are not available. Instead the flight data must be marshaled into a sequence of bytes that can then be transmitted from the server to the client. The

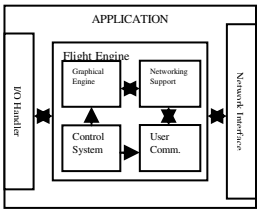

Figure 1. Architecture of the Whiskey DVE.

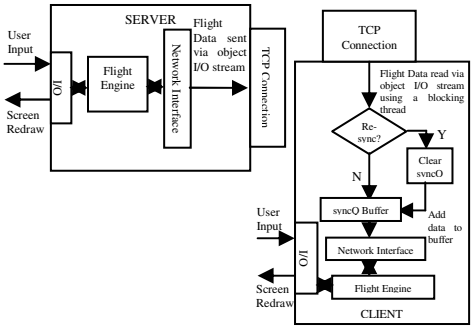

Figure 2. Server based synchronization with threads.

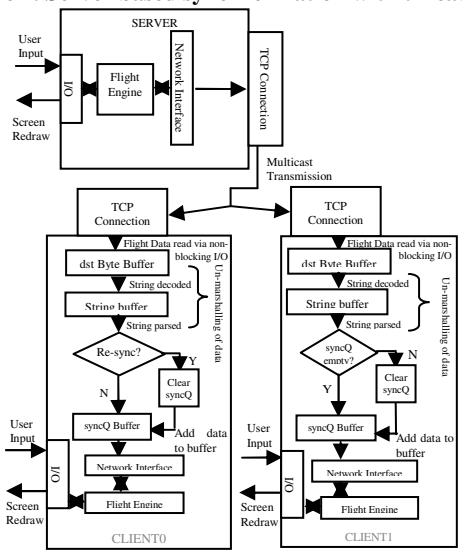

Figure 3. Network sync with non-blocking I/O. Client0 uses server based sync, while Client1 uses transparent sync.

method chosen to marshal the data is to transform the data into a string of a particular format, which is then divided into bytes. The non-blocking I/O on the client system reads the data from the socket into a byte, or "dst", buffer as soon as the data is available. The contents of the byte buffer are decoded into a string buffer, and the string buffer is parsed into a FIFO, or "syncQ" buffer, that consists of a linked list of strings. Each string in the list contains the information that was previously transmitted in a packet.

When new data is required to update the client's environment the next link in the list is accessed and the string contents are transferred back into the normal packet of the flight data management, completing the un-marshalling of the transmitted data. Occasionally, the group of bytes transmitted will only constitute a partial packet of information. In this event, the bytes are returned to the dst buffer and only after enough bytes have been stored in the dst buffer to enable re-constitution of a full packet of flight data will the data be decoded into the string buffer. Periodically, the server issues a re-sync command, which causes the syncQ to be cleared, and the next string of data to be processed immediately and used to update the display. The non-blocking I/O addresses the issue of waiting for data to arrive at the socket at the expense of needing to maintain a number of buffers for handling the marshalling and un-marshalling of data. However, this method still suffers from poorly timed re-sync commands just like the previous method. While this method does offer a distinct improvement over the threaded methods in that it addresses the issue of wasting system resources waiting for data, it is not the most ideal solution.

2.3 Transparent Sync with Non-Blocking I/O

The final synchronization method that was developed also used non-blocking I/O (see Figure 3, CLIENT1). The synchronization method did not rely upon a discrete re-sync command, but instead operated continuously to maintain synchronization between the client and server environments in a more transparent fashion. Data transmission and transformation from packet to byte and back follows the steps described in the previous non-blocking I/O method. Synchronization is maintained in this case by simply replacing all data in the syncQ string buffer with new data whenever it arrives at the socket. If no new data arrives the client application continues to update its environment based on the information contained in the syncQ buffer. As soon as new data appears an automatic re-sync-like operation takes place that enables the application to immediately begin using the new data and discard any remaining packets. This method essentially allows for re-sync to occur at optimal points in time, since it is most advantageous to re-sync whenever data has arrived before all the previous data has been used to update the display. When this is coupled with the fact that non-blocking I/O allows for socket access that avoids waiting for data to appear on the socket, the foundation for a solid networking support system is established.

3. COMPARISON OF SYNC METHODS

Quantitative experimental results were collected to determine which of the synchronization methods is best suited for creating a 3D DVE. For the methods that rely on an explicit re-sync signal, a data collection routine was established that recorded the server's currently drawn frame number the re-sync command corresponded to, along with the client's local frame that was being processed and drawn when the re-sync command was received. For the method that uses the transparent re-sync method the number of frames that were dropped when the new packet of data arrived was recorded. This data provides a quantitative means of comparing the synchronization methods by allowing for the contrasting amounts of divergence between the client and the server environments to be recorded. Each simulation on the client system consisted of a standard test that guided the user's avatar through a set sequence. The re-sync command is set to occur after every 5, 20, or 100 frames for both of the server based synchronization approaches and the amount of divergence is recorded. Then, the transparent synchronization approach is used and the amount of divergence recorded. This sequence of tests was performed using a dual processor 700MHz Intel Pentium 3 class machine with 1GB of memory as the client. The server was running on a 700MHz AMD Athlon system with 256MB of memory. Both machines were running Windows XP and communicated via a 100Mb/s LAN. Video recordings of typical test sequences used for testing the performance of the application

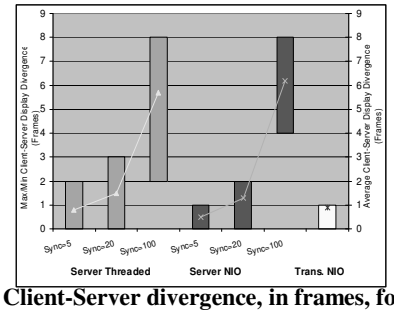

Figure 4. Client-Server divergence, in frames, for a variety of synchronization methods and re-sync intervals.

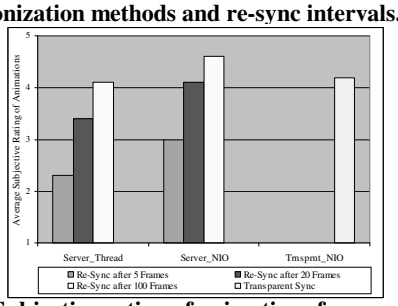

Figure 5. Subjective rating of animations for a variety of svnchronization methods and re-svnc intervals.

can be accessed from the project website [11]. The client-server frame divergence data was collected in sets of 10 samples over 10 simulation runs for each synchronization method and re-sync interval (see Figure 4). The server based synchronization method with threads shows a trend towards undesirable increasing divergence between the client and server applications as the re-sync interval increases. The same trend is seen when the server based synchronization method with non-blocking I/O is used, although the maximum divergence tends to be lower. The transparent synchronization method using non-blocking I/O shows a maximum of one frame being discarded by the client at any one point in time, which indicates that the divergence in the client and server applications is kept to a minimum. It should be noted that the only other method that provides an equivalent maximum, and slightly better average, number of frames discarded, is the server based synchronization method using non-blocking I/O with a re-sync interval of 5 frames.

In addition to the quantitative comparison, a qualitative comparison was also performed. Users of the client's VE were asked to rate the smoothness of the motion of the environment in the graphical display on a scale of 1 to 5, with 5 being the best. The subjective ratings were averaged and recorded for each synchronization method and re-sync interval in order to record the users' subjective impressions. The impressions of 15 users were recorded. The standard test sequence that was described above, and run on the same computer systems, was used again for observation and qualitative assessment. The test sequences were

repeated for each user and the users were then asked to rate the application's performance. The users' ratings were recorded and averaged together (see Figure 5). The graph clearly shows that users much preferred either the transparent synchronization method with non-blocking I/O or either of the other server based synchronization methods provided the re-sync interval was set to 100. The synchronization methods that rely upon an active re-sync command produce the most fluid and subjectively pleasing graphic animations when the re-sync interval is maximized, because fewer re-syncs result in fewer jarring updates to the objects being displayed. The use of the non-blocking I/O, even with the server based synchronization method, appears to better optimize the use of system resources due to the improved qualitative ratings when compared to the threaded approach.

The synchronization methods that performed the best both in a quantitative and qualitative sense are the transparent synchronization and the server based synchronization (with a re-sync interval of 20 or 100) with non-blocking I/O methods. It has been postulated that as the re-sync interval is decreased below 5 frames the server-based synchronization methods should start to behave similar to the transparent method. This does hold true for the amount of divergence realized, but the transparent method produces a much more subjectively pleasing experience. It is likely that inefficiencies in the server-based synchronization code are the cause, but further experimentation would be necessary. Arguments have also been made that a user's subjective experience is more important than the VE's objective reality [10]. It may be possible to relax synchronization in order to optimize a user's subjective experience, which would make the synchronization approaches that rely upon a discrete re-sync interval more viable. However, a method that provides a good subjective experience while maintaining excellent synchronization is clearly the better choice overall. The method of using transparent synchronization with non-blocking I/O provides the best subjective and objective experience in a Java based DVE.

4. CONCLUSION

The work on the Whiskey DVE shows that Java is now a useful tool to enable the creation of a DVE that is cost-effective, portable, maintainable, and extensible. Java can now provide adequate networking support, making effective synchronization between network and client systems feasible with the release of the official support for non-blocking I/O. Java's many benefits that make it an attractive language to develop environments for educational and entertainment purposes are now bolstered by an API that makes the promise of robust networking support an easily realized actuality. This networking support can now be used to provide real time interaction among multiple users, finally allowing Java to become a good tool with which to develop real-time networked VEs. The transparent network synchronization method, via both qualitative and quantitative measurements, was found to be the best method for maintaining client-server synchronization in a DVE. This result corresponded well with the expectations of the designers and further supported the fact that Java can be used to provide the foundation for a networked 3D VE. The source code for the Whiskey application is open source and freely available under the General Public License (GPL) from the Whiskey project website [11].

5. ACKNOWLEDGEMENTS

We would like to thank Sergei Grichine for the open source Java Flight Simulator code that served as the foundation for the graphical engine, Soji Yamakawa for his many VRML97 models, and IBM for their continuing support of our education.

6. REFERENCES

[1] Bangay, S. Experiences in porting a virtual reality system to Java. In *Proc. of the 1st Int. conf. on computer graphics, virtual reality, and visualization* (Cape Town, South Africa, 2001). ACM Press, New York, NY, 2001, 33-37.

[2] Brewer, E., Culler, D., Welsh, M. SEDA: An architecture for well-conditioned scalable Internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (Banff, Alberta, CA, October, 2001). ACM Press, New York, NY, 2001, 230-243.

[3] Burdea, G. C., and Coiffet, P. *Virtual Reality Technology*. John Wiley and Sons, Inc., Hoboken, NJ (2003).

[4] Campbell, B., Collins, B., Hadaway, H., Hedley, N. and Stoermer, M. Web3D in Ocean science learning environments: Virtual big beef creek. In *Proceedings of the 7th International Conf. on 3D Web Technology* (Tempe, AZ, 2002). ACM Press, New York, NY, 2002, 24-28.

[5] Cronin, E., Filstrup, B., Jamin, S., and Kurc, A. R. An efficient synchronization mechanism for mirrored game architectures. In *Proceedings of 1st workshop on Network and system support for games* (New York, NY, April 2002). ACM Press, New York, NY, 2002, 67-73.

[6] Crowcroft, J., Oliveira, M., Slater, M. Component framework infrastructure for virtual environments. In *Proceedings of the third international conference on collaborative virtual environments* (San Francisco, CA, 2000). ACM Press, New York, NY, 2000, 139-146.

[7] Goldberg, A., Kesselman, J., Melissinos, C., Petersen, D., Soto, J. C., Twilleager, D. Java Technologies for Games. *ACM Computers in Entertainment,* 2(2):Article 8, 2004.

[8] Ishibashi, Y. and Tasaka S.: Causality and media synchronization control for networked multimedia games: centralized versus distributed. In *Proceedings of 2nd workshop on Network and system support for games* (New York, May 2003). ACM Press, New York, NY, 2003, 42-51.

[9] Lau, R. W. H., Li, F. W. B., and Li, L. W. F.: Supporting continuous consistency in multiplayer online games. In *Proceedings of the 12th annual ACM international conference on Multimedia, Technical poster session 2: multimedia networking and system support* (New York, NY, October 2004). ACM Press, New York, NY, 2004, 388-391.

[10] Pettifer, S., West, A. Subjectivity and the relaxing of synchronization in networked virtual environments. In *Proceedings of the ACM symposium on virtual reality software and technology* (London, UK, 1999). ACM Press, New York, NY, 1999, 170-171.

[11] Whiskey Source Code, http://www.cem.uvm.edu/~medialab/Fall04/ee214/Java3D/ (accessed January 20, 2005).