Unreliable Transport Protocol for Commodity–Based OpenGL Distributed Visualization

Samuel Thibault, ENS Lyon
Xavier Cavin, SCI Institute
Olivier Festor*, Inria Lorraine
Eric Fleury, Inria Rhônes–Alpes

Abstract

This paper presents a way to use an unreliable transport protocol to perform distributed OpenGL visualization on a commodity network of computers. We present a performance evaluation of a prototype implementation based on the Chromium environment.

1 Introduction

Clusters of commodity computers are becoming a practical tool for visualization. However, most of the time, they are dedicated clusters with a fast (and sometimes expensive) network interconnect: gigabit Ethernet, Myrinet, *etc*. On these networks, *transfer time and packets loss are very low*. For this reason, the chosen transport protocol (IP) is often a reliable one (*i.e.* TCP).

In this paper, we consider alternative network interconnects, that are both cheaper and more commonly available: 100 Mbps, 10 Mbps or wireless networks. On these networks, transfer time is longer and packets loss happens more often (which requires to resent the packets and adds a delay in the transfer time). In those cases, it may be more interesting to use an unreliable transport protocol to get a higher framerate, if we accept a degradation of the final rendering quality (due to packets loss).

We show in this paper how to use an unreliable transport protocol (*i.e.* UDP) to perform distributed OpenGL visualization on a commodity network of computers. We present a performance evaluation of a prototype implementation based on the Chromium environment [1].

2 Network Interconnect

As opposed to shared memory parallel machines — where multiple processors share a common adress space — a cluster of computers requires to specify a way for the machines to communicate between them: a *protocol* describes "the message formats and the rules two or more machines must follow to exchange those messages" (Douglas Comer).

*Contact: mailto:Olivier.Festor@loria.fr

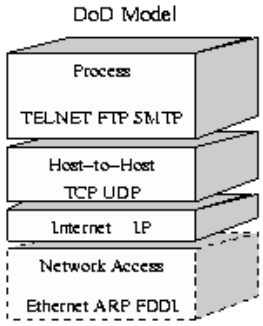


Figure 1: The four layers in the DoD model (Freesoft.org).

Protocol layering is a common technique actually used to simplify networking designs by dividing them into functional layers, and assigning protocols to perform each layer's task. The core Internet protocols rely on the Department of Defense (DoD) Four–Layer Model depicted on Figure 1. The four layers in the DoD model, from bottom to top, are:

1. The *Network Access Layer* is responsible for delivering data over the particular hardware media in use. Different protocols are implemented from this layer by a combination of hardware (*e.g.* a network adaptor) and software (*e.g.* a network device driver); for example, one might find Ethernet or Fiber Distributed Data Interface (FDDI).

2. The *Internet Layer* is responsible for delivering data across a series of different physical networks that interconnect a source and destination machine. Routing protocols are most closely associated with this layer, as is the *Internet Protocol* (IP), the Internet's fundamental protocol.

3. The *Host–to–Host Layer* handles connection rendezvous, flow control, retransmission of lost data, and other generic data flow management. The *Transfer Control Protocol* (TCP) and *User Datagram Protocol* (UDP) are this layer's most important members.

4. The *Process Layer* contains protocols that implement the user–level functions, such as mail delivery, file transfer and remote login.

Applications are usually built on the top of the process layer protocols. However, an application is free to bypass the defined transport layers and to directly use IP or one of the underlying networks.

The internet layer protocols are fundamentally datagram–oriented and unreliable. It is the responsibility of the host–to–host and process layer protocols to enhance the quality of service to that desired by a particular application. These protocols function as an intermediary between the application and network layers. The two main Internet host–to-host layer protocols are:

- TCP is a sliding–window protocol providing reliable, stream–oriented delivery. TCP includes support for *guaranteed delivery*, meaning that the recipient automatically acknowledges the sender when a message is received, and the sender waits and retries in cases where the receiver does not respond in a timely way.

- UDP provides almost no additional functionality over IP. It performs fast, unreliable, datagram delivery. UDP does not implement guaranteed message delivery and provides no guarantees that the order of data delivery is preserved. UDP is called an *unreliable transport protocol* for this reason.

The *Maximum Transmission Unit* (MTU) is the biggest packet size that can be transmitted over a physical network. Different networks have different MTUs; for Ethernet, 1500 is the maximum, and recommended MTU. When a packet reaches a network whose MTU is smaller than the packet size, the Internet Protocol will "fragment" it by breaking it up into smaller ones.

TCP automatically breaks up the data to adapt to the MTU, while UDP does not: the application has to take care about the size of the packets it sends.

3 Distributed OpenGL

In this section, we assume that we have an OpenGL application that encodes its OpenGL commands, send them over a network to a rendering server that in turn decodes the commands and renders them on its own graphics hardware. This is the basic brick of a distributed OpenGL visualization system, where multiple applications can send OpenGL commands to multiple rendering servers.

Most implementations of a distributed OpenGL system rely on TCP for the transport protocol, because it is reliable by nature and rather efficient on a dedicated cluster of computers. All OpenGL commands sent over the network are guaranteed to be received, in the same order that they have been sent. Figure 2 shows a rendering of a crocodile model using TCP: it appears as if it was rendered locally.

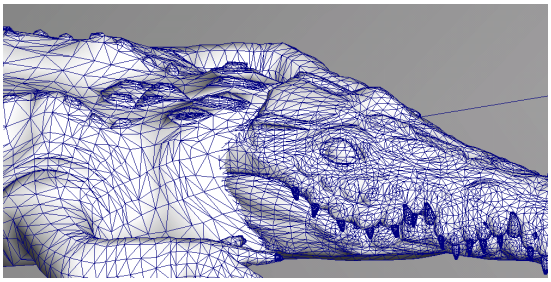


Figure 2: TCP rendering of a crocodile.

3.1 UDP *versus* TCP

Replacing TCP by UDP is equivalent to remove the acknowledgment mechanism, so that some data may not be received, or at least not in the order they were sent. It allows to optimize the use of the network, because the sender no longer have to care about the loss of packets.

Simply replacing TCP calls by UDP calls is obviously not a viable solution. Indeed, if some important OpenGL commands are lost, the execution of the received ones (possibly in a different order) often leads to a fatal crash. There is also the case of OpenGL commands that are supposed to return a value (like the `glGet*` family).

One solution is to combine the use of TCP for commands that can not be lost and UDP for all other commands. However, the two communication channels have to be synchronized so that commands sent through UDP are correctly interleaved with commands sent through TCP. We so add a sequence number to UDP packets, which is incremented each time new commands are sent through TCP, as shown by Figure 3. The receiver first tries to read a UDP packet, and if the sequence number has raised, it reads the new commands sent through TCP and can handle the UDP packet.

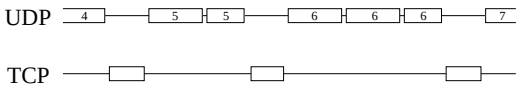


Figure 3: Interleaving UDP and TCP packets.

Rather than exhaustively listing the OpenGL commands that can not be lost, we have chosen to incrementally choose the one that can be lost and therefore be sent through UDP.

Actually, we have first decided to only send the `glVertex` commands through UDP. These commands are used within `glBegin`/`glEnd` pairs to specify point, line, and polygon vertices, and may represent up to 80% to 99% of the whole commands, depending on the type of application.

3.2 Loosing packets

If the whole content of a `glBegin`/`glEnd` pair is lost, then some holes would appear in the final rendering. Unfortunately, this is not so simple: Figure 4 shows a rendering of

the crocodile model using the interleaved UDP/TCP mechanism. We can notice that a lot of “bad”, overlapping triangles have been rendered.

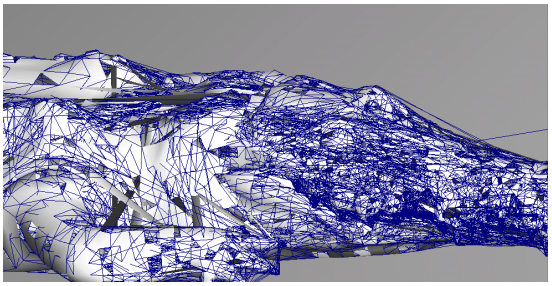
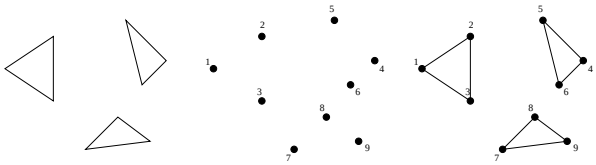


Figure 4: UDP/TCP rendering of the crocodile.

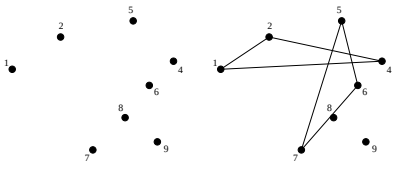
Consider the simplified example of Figure 5. Suppose that the application wants to send the three triangles depicted by Figure 5(a): it will likely use a `glBegin`/`glEnd` pair including the points of Figure 5(b) in the specified order so that they get linked correctly as on Figure 5(c).



(a) Expected. (b) Sent. (c) Rendered.

Figure 5: Rendering without loss.

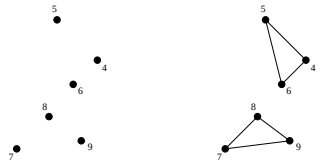
Real 3D objects (such as the crocodile model of Figure 2) are composed of many more points, and it can happen that the content of a `glBegin`/`glEnd` pair becomes larger than the MTU of the network, and then gets fragmented before being sent. If a subset of the content of a `glBegin`/`glEnd` pair is lost, we may get to a case similar to the (simplified) one depicted on Figure 6(a), where a single vertex (the third one) has been lost during the transmission. The triplets of vertices representing the individual triangles are wrong, the two last vertices are ignored, and the final rendering shown on Figure 6(b) is completely incorrect.



(a) Received. (b) Rendered.

Figure 6: Rendering with a single vertex loss.

A solution to overcome this problem is to group the points by three (in the case of our simplified example) in the UDP packets, so that the loss of vertices happen with triplets of vertices, as shown on Figure 7(a). A packet loss so leads to a coherent hole, as shown on Figure 7(b).



(a) Received. (b) Rendered.

Figure 7: Rendering with a triplet of vertices loss.

A similar attention must be given to other cases of `glBegin`/`glEnd` pairs: `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_QUADS`, `GL_QUAD_STRIP` and `GL_POLYGON`.

Figure 8 shows a better rendering of the crocodile model with the interleaved UDP/TCP mechanism and the MTU consideration. We can observe holes due to the packets loss (about 10% in this case), but the result looks much better than on Figure 4.

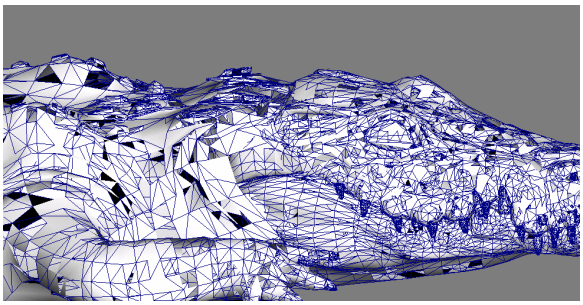
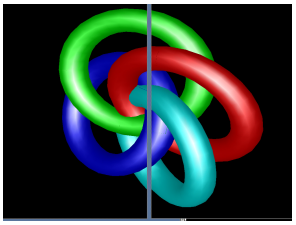


Figure 8: Fixed UDP/TCP rendering of the crocodile.

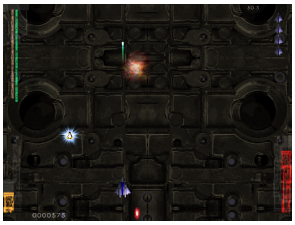
4 Experimentations

We have implemented the UDP/TCP mechanism inside the Chromium environment [1]. Chromium is a flexible framework for scalable real–time OpenGL rendering on clusters of workstations. The communication is done through a stream packing library that takes a sequence of OpenGL commands and produces a serialized encoding of the commands and their arguments. The serialized representation of the OpenGL command is sent over the network and then decoded for rendering or further processing.

Chromium provides a point–to–point connection–based networking abstraction, that abstracts the details of the underlying transport mechanism. This library is currently im-



(a) Rings.



(b) Chromium B.S.U.

Figure 9: Screenshots of two tested applications.

plemented on top of TCP/IP and Myrinet. We have extended this library so that it supports both the version 6 of the Internet Protocol (IPv6) and our UDP/TCP mechanism.[1]

We have experimented our implementation between two standard PC's on a local network with three different configurations and with the following applications :

- Crocodile: a triangulated surfaces OpenGL viewer, loaded with the crocodile model (see Figure 2).
- Rings: one of the Chromium's test programs that implements a simple parallel application displaying rings (see Figure 9(a)).
- Chromium B.S.U. [2]: a space shooter game written in OpenGL (see Figure 9(b)).

100 Mbps: our first experiment is on the 100 Mbps (100BaseTX) local network. Table 1 shows the obtained framerates with the three applications.

In this configuration, we observe a small gain with the crocodile application (22%), with few missing triangles, but the performance is the same with the rings application. Indeed, the geometry of the rings is very light, and all synchronization operations must be done through TCP.

	Crocodile	Rings	Chromium B.S.U.
TCP	2.2fps	110fps	playable
UDP/TCP	2.7fps	110fps	playable

Table 1: Framerates on the 100 Mbps local network.

10 Mbps: in order to reduce the available bandwidth, we have set the network card of the PC's to 10 Mbps (10BaseT). Table 2 shows the obtained framerates with the three applications.

In this case, we observe a gain with the crocodile application (28%) but also with the rings application (65%).

Wireless: finally, we have replaced the network card of the PC's with a 11 Mbps wireless card. In this configuration,

[1]These modifications have been applied to the patches–1–branch of the Chromium's SourceForge CVS repository.

	Crocodile	Rings	Chromium B.S.U.
TCP	0.28fps	24fps	playable
UDP/TCP	0.36fps	39fps	playable

Table 2: Framerates on the 10 Mbps local network.

a lot of packets loss happen, and the TCP performance goes dramatically down: for instance, the transfer rate for sending a file is about 200 KB/s. Table 3 shows the obtained framerates with the three applications.

We obviously observe an important gain for the crocodile application (560%), because TCP needs to resent a lot of geometry. The gain is less important with the rings application (145%), because it uses less geometry and relies more on TCP. However, the rendering quality with UDP/TCP is very bad in both cases, because the packets loss degrading the TCP performance are directly visible on the screen.

	Crocodile	Rings	Chromium B.S.U.
TCP	0.066fps	5.5fps	quite playable
UDP/TCP	0.37fps	8fps	non playable

Table 3: Framerates on the wireless network.

5 Conclusion

This paper presents a novel way to use an unreliable transport protocol for commodity–based OpenGL distributed visualization, by interleaving UDP and TCP communications. We have implemented this mechanism inside the Chromium environment, and our preliminary experiments show that it can be adapted if the geometry represents most of the traffic, if rendering quality is not the first interest, if it is animated (the eye compensates the holes), if there are a lot of packets loss, and if network is the bottleneck. However, if the application requires a lot of synchronization, or if the network is faster than what the application can handle, TCP may be better adapted. We also believe that our current implementation could benefit from some UDP specific optimizations in order to obtain better performance.

Acknowledgments

The authors would like to thank Alain Filbois for his constant technical support and Bruno Levy for his help with OpenGL related problems. This material is partially based upon work supported by the AVTC under the DOE VIEWS program.

References

[1] *Chromium*. http://chromium.sourceforge.net.

[2] *Chromium B.S.U.* http://www.reptilelabour.com/software/chromium.