Transaction Based Distributed Computer Graphics Systems

Daniel G. Aliaga

A paper prepared for Honor's Thesis - Brown University

Abstract

With the advent of distributed processing, the computational abilities of large systems have greatly increased. In particular, the performance of computer graphics systems can be greatly improved by exploiting the inherent distributed characteristics of such a system. The **UGA** system (a unified graphics architecture) currently being developed by the Brown University Computer Graphics Group follows an object-based framework and provides an integrated platform for a growing number of modeling and animation techniques. Distribution of the system is necessary for further growth of the functionality and computational ability of the **UGA** database. The event-based transaction system presented here provides a highly efficient implementation for distributed processing in computer graphic applications.

1.0 Introduction

1.1 Prior Work

Traditional remote procedure call interfaces (RPC) were built as a layer of software on top of an operating system. The servers defined a set of functions that were called from a remote client. A client had a stub for each server function that could be called. The RPC interface would take care of marshalling the data but was designed exclusively for point-to-point messages. When a client called an RPC stub function, the client process blocked until the remote procedure call terminated, disabling the client from performing other computations.

Higher level shared data models, such as ORCA [1] and Argus [2] provided a very good abstraction, but added unwanted overhead in order to guarantee serializability and synchronous performance. As we will see later, we do not need to guarantee synchronous performance and serializability for graphics applications in the **UGA** system.

With the introduction of networked file systems, operating systems began supporting remote communication as an integral part of the operating system (e.g. Amoeba OS, and perhaps also in this category is OSF/DCE) [3][4].

1.2 Motivation

In the area of computer graphics, where much computational power is required, distributed systems could enhance performance, bypassing the computational limitations of a single processorThe Brown Computer Graphics Group is currently designing a modeling and animation system that combines capabilities for creating very sophisticated animations, electronic books, and models into one platform [5]. The **UGA** system is a database of objects to which we can apply a list of operations or perform inquires. In distributing **UGA**, all the basic database modification and inquiry functions should be supported. Remote procedure call mechanisms do not provide an interface that is rich enough to support the database functionality. In addition, locking data fields is not a requirement since computer graphics system are not always designed to be full fledged databases that handle concurreny and race conditions. Such schemes would only slow down performance.

1.3 Overview

This paper will present a distributed environment based on a transaction model [6] used to distribute the aforementioned system. The distributed environment supports very intelligent buffering schemes to improve the performance of message passing between remote database objects. These messages can be inquiries or replies. Furthermore, since the **UGA** database server(s) and clients have an event loop as their main loop, an event triggered model is used. Additionally, the system can mimic RPC mechanisms.

The model presented here consists of two main layers:

- Transactions (TRX)
- Object Interface (OIL)

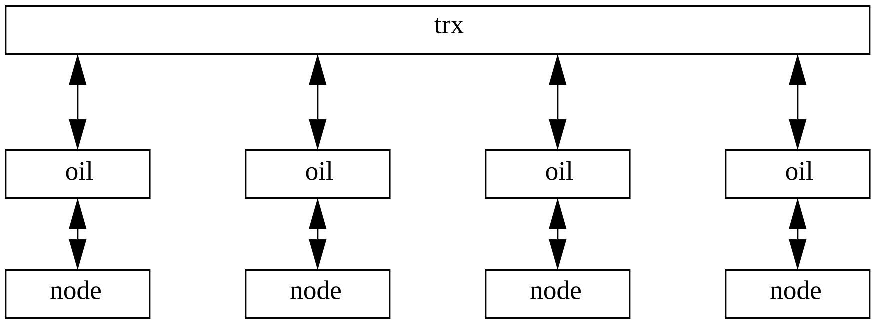

FIGURE 1 Interconnection of the Two Main Layers

The transaction layer hides the network operations and handles the opening and closing of transactions. If fact another layer, called STREAM, provides standard abstractions for basic network operations such as bind, connect, read and write. A handle is provided to represent a transaction server. Transaction servers can run on the same processor or on separate processors. The transaction layer does not differentiate between database servers and database clients since the transaction blocks look the same.

The object interface layer is an abstraction that enhances the usage of the transaction layer for **UGA**. In this layer, clients can be interfaces, editors, database monitors or other tools. Servers will process database requests from clients. The system could be expanded to support multiple servers interacting. This issue will be explained in more detail in section 5.2.

2.0 The problem

2.1 What we wish to accomplish

The **UGA** modeling and animation system maintains a list of objects. Each object has methods that are used to compute and inquire data the object. Information is passed between objects through messages. We would like to be able to maintain objects on separate machines. This would allow us to distribute the computational space over multiple processors. A difficult problem that this introduces is how to determine the optimal distribution of objects. One of the principal factors that determines the best distribution is the bandwidth of message passing between sets of objects. In addition, the bandwidth of messages might change over time, which suggests the need for some sort of dynamic distribution. Furthermore, one must take into account that sometimes the distribution might actually result in more overhead than simply performing the computations on a single processor. This is mainly due to high network traffic and the lag that it causes. A scene that has many disparate parts that rarely interact lends itself well to a simple distribution of the objects in the database. Unfortunately, this is not always the case.

Transparency is another issue that we wish to address. The programmer or user should not have to worry about how many processors are computing the data or where these processors are located. This is a very difficult issue and many attempts have been made to solve it through high-level languages or explicit protocols.

In some cases we wish to compute a set of values, but we are not terribly concerned when these computations are completed. In other cases, we cannot proceed until the requested computation has completed. In such cases, our current thread cannot continue until the thread we just started completes. These situations reduce the parallizability of our system. For example, if we wish to compute the color of an object based on its position we cannot compute the color and position at the same time. On the other hand, if we wish to apply deformations to four objects and there is no interdependency, it is trivial to apply the deformations in parallel.

Other features we would like to support include remote interests and debugging/monitoring. The latter is a rather large problem: how do you debug or develop a distributed system. Standard single process debuggers are not powerful enough [7]. Not all debugging consists of stepping through the program instruction by instruction. We would like to “visualize” what the distributed system is doing in a more abstract fashion. Such a program is difficult to implement especially since the dataflow of a single processor far exceeds the data transfer rates of a typical ethernet network.

3.0 The solution

3.1 TRX: The Transaction Layer

A transaction can be defined as an atomic set of operations to be performed either in its entirety or not at all (as in the case of unexpected errors). Thus, it should be possible at all times to know what the current state of the system is since we can keep track of which transactions have occured and which have failed. In the case of **UGA**, a transaction is a set of database changes and/or inquires grouped together and identified with a unique id. A transaction is sent as a single stream of data across the network. Any return data is also sent as a single stream of data. The data stream is sent using the TCP/IP protocol which guarantees reliability of a single data stream.

In this layer, two types of transactions are distinguished:

- Message or Inquiry transaction
- Reply transaction

The first type is a transaction sent to a single destination. The destination transaction server will then decode the transaction block (see 3.2) and perform the appropriate actions.

The second type is a transaction returned to the originator of a particular transaction. For example, this allows a destination transaction server to reply to a sender.

3.1.1 Transaction Creation

A message transaction is created by a call to TRXopen. A reply transaction is created by a call to TRXreply parametrized by a message transaction. Both functions return a handle to the appropriate transaction block.

Initially a transaction block is empty. The user-level application can marshal transaction data into the transaction block. Functions exist to update and automatically resize the block according to the needs of the user-level application. The application using transaction blocks defines what the transaction data is. It can be headers for remote procedure calls, marshalled data structures or simply flags to be sent to a remote application.

Multiple transactions may be open at the same time. No network activity will occur until the transaction is closed and flushed to the destination transaction server.

3.1.2 Transaction Closing

A handle is used to identify each possible destination. A transaction is closed by a call to TRXclose parametrized by handle to a particular destination transaction server.

There are 3 ways in which a transaction can be closed:

- Synchronous
- Asynchronous
- Wait

Synchronous closing will immediately send the transaction to the destination. If a large number of transactions exist, this mode will degrade performance.

Asynchronous closing will not send the transaction immediately but will attach it to a queue of unsent transactions. After a given interval all currently queued transaction blocks will be flushed. The interval is user-specified; it is typically under 1 second.

Wait closing will not send any transactions closed with this mode until a specific call to TRXflush has been made. Therefore, if the user-level application knows a priori that a batch of transactions will be made, this mode is very convenient.

Note that if a reply is expected, a function may be registered to be called upon arrival of the reply transaction; or, alternatively, the reply transaction will be queued and can be retrieved later.

Since a transaction server has an event loop, when a reply transaction arrives the transaction layer can call a function specified by the user-level application. Thus, when closing a message transaction and a reply transaction is expected, a function may be specified to be called upon arrival of the reply transaction. This allows the process to continue computations that do not require the results of a computation just shipped off. If the process must block until a specific reply block arrives, the transaction layer can be told to wait for a reply block. All incoming messages and replies will be enqueued until the desired reply block arrives.

3.1.3 Transaction Queues

The transaction layer maintains 5 queues that contain all the transaction blocks for a running transaction server. A block is not in more than one queue at a time. The queues are:

- Free Queue
- Open Queue
- Closed Queue
- Inquiry Queue
- Reply Queue

The free queue maintains a list of unused blocks. It emulates a free list so that opening a transaction is efficient, since a block simply has to be moved from the free queue to the open queue. When the user-level application finishes with a transaction block and closes it, the block is put on a closed queue. It remains there until it is actually sent to the destination after which the block is put back in the free queue.

When an inquiry or reply is received, it is put in the corresponding queue. If a transaction server receiving a block is not waiting for a specific reply, the inquiry and reply queue will not get large. On the other hand, if a transaction server is waiting for a specific reply block, inquiries and other reply blocks will be enqueued until the pending reply block arrives. The inquiry or reply queue may be flushed at any time through a call to TRXflush.

3.2 OIL: The Object Interface Layer

Messages are passed between objects to resolve dependencies and compute requested data. How a message is interpreted depends on the class of the receiving object. The object interface layer, through the use of transactions, allows the user-level application to pass messages between objects in the database. The syntax by which the messages are given is the same as if the calls were made using the local function calls to the **UGA** database. Arguments to the objects are marshalled by the object interface layer (OIL) and put onto the current transaction block.

3.2.1 Messages

An application will inquire data about the objects in the **UGA** database such as their polygonal representation, color or position. Each object, in turn , will pass messages to objects it depends on to compute the requested data. These messages are passed through object class specific method calls. The object interface layer provides a mechanism by which the messages can be passed remotely. The transaction data consists of sets of method ids and method arguments, appropriately marshalled.

All functions used to perform message passing must be registered during initialization in the object interface layer function table. This table defines the argument format and the actual functions to call for each message.

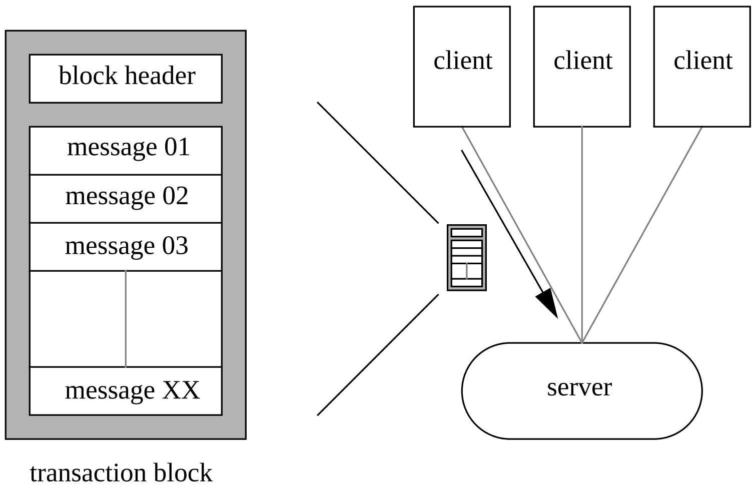

FIGURE 2 Object Interface Layer Transaction Blocks

3.2.2 Message Passing

The object interface layer differentiates between database servers and database clients. A server manages the **UGA** database and is capable of processing messages from clients in addition to messages between objects.The server waits until it receives a message from a client, the message will be processed and then the server waits for another message to process.

A client can pass any message to a server. To pass a single message or a batch of messages the client should call OILopen, followed by the appropriate OIL calls (which return immediately) which will append the message to the associated open transaction. After all messages have been specified, OILclose should be called. This function will flush the associated transaction block according to the transaction closing mode. If replies are expected, the function passed to OILclose will be called when the return data arrives.

3.2.3 A series of functions have been implemented on top of the basic open, call and close functions. These higher level functions provide an abstraction which simplifies the general functionality provided. OILwait, which can be used in place of OILclose, will block until the return data arrives. If no return data is expected, it will return immediately. OILonce is used to do a simple open, call and close. This allows the user-level application to easily pass a single message. Similarly, OILonce_wait can be used if wish to pass one message and wait for the reply to arrive (uses OILwait instead of OILclose).

3.2.4 Remote Interests

Another feature that OIL provides is remote interests. An function call is triggered when an event occurs that matches the parameters of an interest (object, change operator, change operator type, action and time). The **UGA** system already relies on an interest mechanism. OIL provides an extension that allows for interests to be registered from remote clients.

A client requests the creation of an interest on a server. The request is forwarded to the specified server. The interest is then created and a handle is returned to the client. The handle may be used to remove the interest at any time. When an interest is triggered on a server, a message is sent from the server to the appropriate client. On the client side, OIL will then call the client's registered function.

3.2.5 Marshalling

The **UGA** system uses a set of standard data structures to pass and store messages. How each type of message is passed is best determined by the actual module that implements the message. This object oriented approach allows each data structure to optimally marshal itself. If the data structure changes, the rest of the system does not have to change. A common set of data structures exist for which standard marshalling functions are provided.

In order to support our system across multiple architectures, **UGA** has its own set of common marshalling functions. They provide the mechanisms to safely marshal bytes, integers, longs, words, enumerated types and pointers.

4.0 How it can be used

The object interface layer and transaction layer allow for messages to be sent between servers and remote clients. Thus through the OIL layer, clients may control the database by editing and manipulating its contents remotely. Through remote interests, clients can be informed of events on the server. This model allows for the accumulation of many messages into one transaction block, thus optimizing performance. For example, to inquire the drawable versions of 100 objects, this model provides an interface which is very similar to performing one hundred function calls, but the inquiry gets sent in a single network operation to the server and the return data eventually arrives also in one network operation. Meanwhile the client can continue computing or inquiring other data or simply processing X events. The application can also make calls to 5 different servers and still the messages would only be sent in one network operation per server.

The following sections will briefly discuss how a distributed system could be used to enhance user interfaces, improve controllers and design an interactive monitor.

4.1 User Interfaces, Controllers

The design of a user interface for the **UGA** system is not an easy task. It is difficult to create a user interface that correctly visualizes the database for an artist or a technical user. Many different components of an interface need to interact with the database. Since multiple interfaces might operate simultaneously and since their presence should ideally not degrade performance of the system, it is appropriate for them to be remote clients that send messages to the database and are informed of other activities through further messages. For more details on the user interface see [8].

4.2 Objects, Controllers

The list of objects maintained by the system may include non-geometrical objects such as controllers. If there are multiple controllers in one scene, the system will be frequently requested to compute a time step. If each controller resides on a different processor, the total computation time will be significantly less. Controllers may have to interact, in which case the requests are over the network, but the most expensive part of controllers is their computation which can often be done in parallel. Even more parallelism is achieved when various sets of disjoint controllers operate, for each set will not affect the computation of the others.

This strategy would be very difficult to implement efficiently since there usually is frequent communication between servers. Nevertheless, controllers do seem to be able to be parallelized in some fashion. One approach is to look closely at the state traversal algorithm for the **UGA** system [9]: an object is requested to compute a certain data type. To obtain the data, the traversal goes through the list of change operators stored with the object. The object might have dependencies on other objects which will cause state traversals to start on those objects. During a state traversal we might hit an operation that is computationally intensive. We could block the current thread and spawn off a thread to another processor to handle the computation and continue the state traversal in another thread provided it does not require the results of the computation just shipped off. For example, the state traversal for the next request could be initiated. When the shipped off computation completes, the blocked thread will continue its traversal.

In order to overcome the high network bandwidth, a multi-thread environment with shared memory would be advantageous. Traditional multi-thread environments are restricted to the same processor, but current developments (e.g. Mach OS) provide multi-thread environments where each thread might be on a separate processor and still provide shared data. Such a strategy would be well suited for the **UGA** database.

Another approach would be to distribute the database among an array of processors. Each processor (workstation) would be a server with a shell database. Each server contains all the information necessary to potentially compute any requested data. Specifically, each database server will keep all the objects, change operators and control points. Consequently whenever one of these entities changes, all other servers must be updated.

Thus, transactions that make permanent changes to a database server will get propagated by a server to all other servers. The updates are grouped into transaction blocks so as to minimize network activity.

The above approach is perhaps more feasible to implement. Unfortunately, it could produce very high network bandwidth which might significantly degrade performance. Any change to the database will eventually have to get propagated to all servers - even actions as simple as modifying the translation of an object.

4.3 Interactive Monitor

The interactive monitor provides an interface that shows the current state and activity of the **UGA** database. It started as a test program, but eventually grew into a full client once its usefulness became apparent. Through remote interests which are dynamically registered and unregistered, the user may “see” the list of objects and their states. Rather than providing an abstract view as the user interface (Fugazi, currently under development) does, it provides a direct representation of the database. This monitor will track the addition, removal and editing of objects and change operators within the system, and display relevant information such as priorities, scopes, caches and other information which is of great use to a person developing in **UGA** [10].

5.0 Conclusions

The problem of adequately distributing a system is a very difficult one. The exact usage of the system is very important as well as how transparent the distribution is to a developer. Different paradigms of distribution exist. This paper presents one approach, which will hopefully promote further investigation and development of new ideas to improve the distribution of the **UGA** system.

References

[1] Henri E. Bal, Andrew S. Tanenbaum, M. Frans Kaashoek, “ORCA: A language for Distributed Programming”, Vrije Universiteit - Amsterdam, Netherlands.

[2] Barbara Liskov, “Distributed Programming in Argus”, Communications of the ACM, March 1988.

[3] Andrew S. Tanenbaum, “The Amoeba Distributed Operating System”, 1991, Vrije Universiteit, De Boelelaan1081a - Amsterdam, Netherlands.

[4] Open Software Foundation, DCE Application Development Guide, OSFTM DCE Release 1.0 S3, March 18, 1991 Revision.

[5] R. Zeleznik, B. Conner, M. Wolka, D. Aliaga, N. Huang, P. Hubbard, B. Knep, H. Kaufman, J. Hughes, A. van Dam, “An Object Oriented Framework for the Integration of Interactive Animation Techniques”, Proceedings of SIGGRAPH ‘91, August 1991.

[6] Michael Young, Dean Thompson, Elliot Jafe, “A Modular Architecture for Distributed Transaction Processing”, Transarc Corporation.

[7] Chu-Chung Lin, Richard J. LeBlanc, “Event Based Debugging of Object/Action Programs”, AT&T Bell Laboratories, Georgia Tech -Atlanta Georgia.

[8] Ken Herndon, Scott Snibbe, Brook Conner, Dan Robbins, “New Interface/Animator Interface Proposal”, 1991, Brown University, Providence RI.

[9] P. Hubbard, M. Wolka, R. Zeleznik, D. Aliaga, N. Huang, UGA: A unified graphics archectiture. Technical Report CS-91-30, Brown University, 1991.

[10] William F. Appelbe, Charles E. McDowell, “Integrating Tools for Debugging and Developing Multitasking Programs”, Georgia Institute of Technology, University of California at Santa Cruz.