Tailorable Update Policies for Distributed Object Systems

Adam Jonathan Griff and Gary J. Nutt
Department of Computer Science, Campus Box 430
University of Colorado at Boulder
Boulder, CO 80303-0430
(303)492-7906 Voice (303)492-2844 Fax
griff@cs.colorado.edu

Abstract

Distributed object applications rely heavily on distributed systems and objects. Solutions using CORBA with its location transparency are inefficient and do not scale for high-throughput networked applications. Our work addresses these inefficiencies by creating a mechanism enabling applications to influence the system's object location policies on an object-by-object basis.We also provide an analytic comparison of distributed object systems that do and do not support location transparency and caching policies. The analysis shows that by using tailorable policies it is possible to achieve significantly increased performance through the reduction of the message traffic so that it is only a fraction of a percent of the message traffic required to implement a standard CORBA policy.

1 Introduction

Distributed systems and objects have emerged as fundamental programming technologies in the last decade. The evolution of the two technologies has been relatively independent: distributed hardware and system-level software now provide an environment for supporting industrial-strength applications based on TCP/IP and on extensions such as the client-server and remote procedure call models of communication. Object technology has caused an unmistakable evolution in the way programmers produce software. Modern applications executing on a workstation are constructed using hundreds of objects, and each object can demand significant support from the hardware.

With workstations interconnected by contemporary networks with bandwidths of 100 Mbps (and greater in the near future), it is natural for object-oriented applications to attempt to take advantage of the distributed technology.

Research and commercial organizations have expended considerable effort to combine object technology with distributed systems, resulting in the creation of CORBA [OMG95] [Scot97], DCOM [Chap97], and others. CORBA is an open specification that uses location transparency and focuses on the combination of heterogeneous distributed systems and object technology. It defines an architecture in which objects, written in one language executing on one computer, can invoke methods of objects written in another language running on a different computer. Tenenbaum, Chodhry, and Hughes [TCH97] provide one of many examples of the growing acceptance of the CORBA standard in the internet market.

We have speculated that applications could have considerable influence on remote object access performance if the applications could supply *hints,* regarding the object management policy, to the distributed object system. The system would be expected to use these hints in placing specific objects at a location that maximizes application performance. These hints provide the meta-interface necessary to enable an open implementation of distributed objects. If objects are read much more often than they are written, then they could be cached at the reading locations — depending on the application semantics for the particular object. If an object is cached, and has many readers and a few writers, the application could suggest the type of memory consistency model to be used for the object, e.g., a cached object might have each write update synchronized and caches kept consistent using sequential consistency, or the cached object can use a much weaker form of consistency when the cached copy's consistency is less important.

We have designed the *Gryphon* mechanism to act as an agent between the applications and the distributed object manager (an ORB). A Gryphon analyzes hints from a set of applications, then selects a particular object location policy reflecting the application and system requirements. Based on the general design of the Gryphon, we are able to compare fundamental aspects of the performance of several different approaches for supporting distributed objects for a given set of applications.

This paper briefly describes the Gryphon architecture for a CORBA-like distributed object system where applications (or users) can tailor the locations policies. Next, the paper provides a performance model of different Gryphon configurations with a centralized remote object manager and centralized and distributed CORBA configurations. Depending on the behavior of the applications, the Gryphon approach can considerably improve the performance of applications using distributed object. The analysis demonstrates the feasibility of the Gryphon approach, which we are currently implementing.

Section 2 describes related distributed object work. Next we discuss how object location and update policies can influence performance in Section 3. Section 4 describes how Gryphon provides support for location and update policies. Section 5 presents the analytic model. Section 6 uses the models to show traffic patterns. In Section 7 we describe the *Virtual Planning Room* (VPR), our Distributed Virtual Environment (DVE) prototype. We then show workload characterization in the VPR and provide comparisons between object utilization scenarios. Section 8 discusses our conclusions.

2 Related Work

Ahamad and Smith [AS94] described a technique for detecting mutual-consistency requirements in shared objects. By determining where the object is being used and the shared repositories where the object is being stored, the object can be cached where it is frequently accessed. This reduces data access latency and the associated communication overhead. This research used causal consistency, a form of weak consistency, to regulate consistency maintenance of the object copies.

The Configurable OBjects(COBS) project at Georgia Institute of Technology is building a CORBA compliant system to run on high-performance architectures [SA97]. This project tailors the performance levels of the objects at run-time based on the requirements of the application. Tailoring is achieved using attributes to directly control system characteristics including: selecting object implementations; making objects passive, single- or multi-threaded; fragmenting and replicating object state; using reliable, unreliable, or multi-cast protocols; and selecting compression and secure transmission protocols. This project provides the application programmer a way to gain access to the high-performance features of the underlying object management system.

Project SIRAC [BAB+96] incorporates a technique for creating distributed applications for real time interaction using multiple workstations. This project uses Olan, a language designed for run time support of distributed application. Brown and Najork [BN96] have created distributed active objects, Oblets written in Obliq. The Obliq language facilitates the distribution of objects over the World Wide Web by providing distribution primitives.

There is a large research and product development community working on object management in DVEs. Some of these systems have incorporated domain specific update strategies. For example, the RING system with a centralized multi-casting subsystem incorporates knowledge about visual and auditory occlusion to reduce consistency updates for cached objects [Funk95]. The idea is to allow the server to keep track of which objects are visible and within hearing distance of an avatar at a client machine, then to only propagate changes to the client if it effects these objects. This strategy requires that the server have knowledge of the location and movement of each avatar.

The Black Sun Community Server is a DVE product that runs over the World Wide Web (WWW) [Blac97]. This product has also developed domain specific assumptions and algorithms to aid in communication reduction. For example, to aid in scalability, as the density of users increases the visible range of an object is reduced (hence reducing the number of update messages on the network). The Black Sun Community Server only propagates avatar updates to the parties that it decides are interested (using the density threshold) three times a second.

The Distributed Interactive Simulation (DIS) varies an object's update rate based on the avatar's distance from the object (this concept is based on "level of detail" notion often used in the graphics community) [Holb95].

In our research we have merged the specialized object distribution area with specialized application domains by providing mechanisms for the application developer to impart knowledge to the Gryphon system.

3 Application-Specific Object Location and Caching Policies

In Section 7.1 we describe scenarios of how objects usage varies based on the application. There is a spectrum of ways in which objects are referenced in a DVE. In some cases, an object is initially defined as a part of the world, then no part of its state ever changes (e.g., a wall in a room). In other cases an object may have its state changed rapidly due to its behavior (e.g., an avatar in a DVE, or a subject of collaborative work), or because of frequent interactions with other objects (e.g., an office form).

Figure 1 represents a set of different location policies that can be used, based on the application needs, to reduce performance bottlenecks while referencing objects. Some objects (like objects u and v) are private to an application. Other objects (like r and s in the figure) should be kept only in a server with all references to the object being remote references over the network. Cached objects (such as t and the cached copy t' in the figure) have the original object stored on the server and copies in clients. Finally, objects such as x and y in the figure are placed at a client, yet can be referenced from other clients.

Besides influencing the location/caching of an object, it may not be important to the application for a cached object to be especially consistent. For example in the VPR domain one client may be changing an object frequently but the other client machines need only update their cached copy of the object every few seconds or minutes. Therefore we also model the case where a client containing a cached object is only occasionally updated. Next we describe mechanisms for managing the location and caching of shared objects.

3.1 Object Location

Assume that a system provides a CORBA interface to manage shared objects for an application, i.e., remote objects are referenced using an interface definition language (IDL). CORBA explicitly addresses the possibility that the underlying object manager may be distributed, meaning that in a network environment there may be several different locations at which an object is stored. Since CORBA supports *location transparency*, a client need only use the IDL to reference the object; the client is not permitted to know where an object is located — it can only reference the object through the IDL, then the underlying ORB will locate and reference the object.

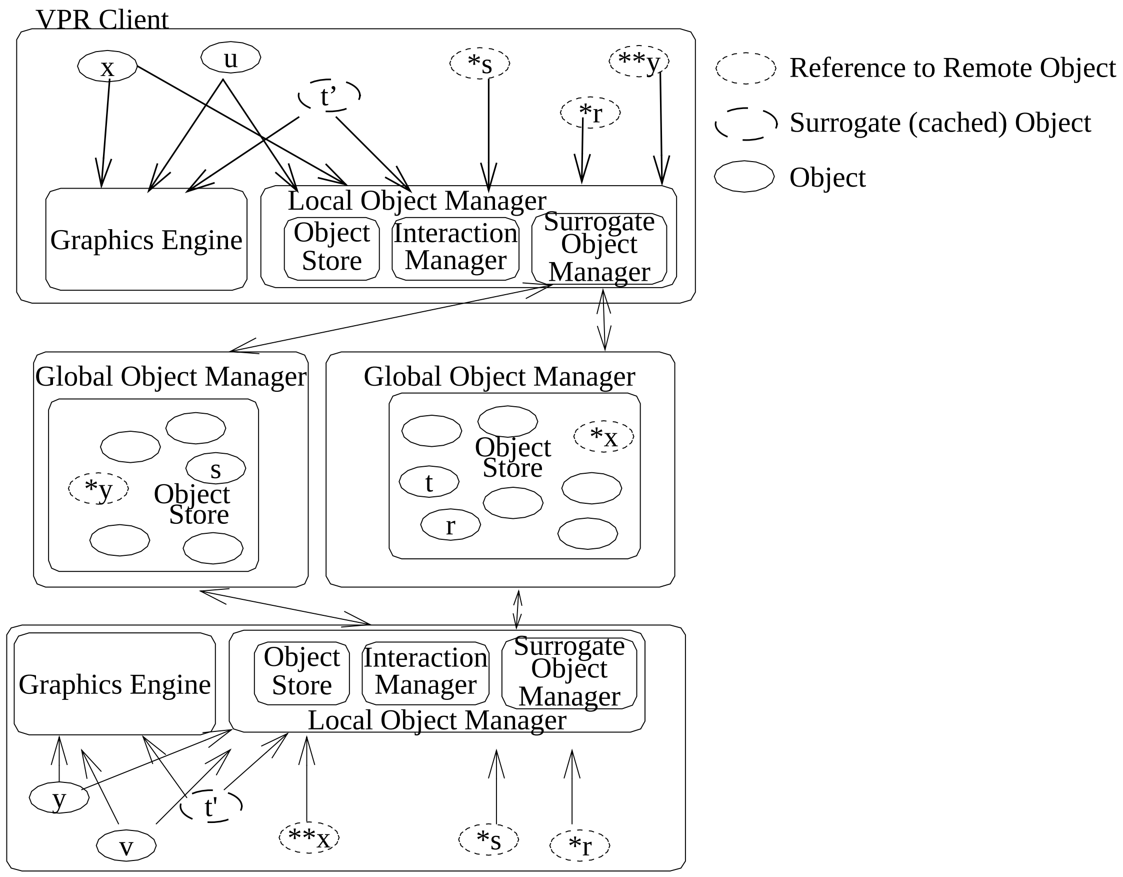


Figure 1:Object References

The system's *location policy* defines the ORB's strategy for placing objects at various locations in the network. The ORB is free to choose any location for an object, provided that it can still provide object reference. Today, remote object references dominate the performance of a distributed object application such as the VPR. If the application uses many remote objects, then its performance will be degraded to the point that remote objects are infeasible.

Since CORBA provides (and enforces) location transparency, its location policy is determined by the ORB implementation or the system administrator. Neither the ORB designer nor the system administrator is likely to know, *a priori*, what the reference pattern will be for any given object which is solely determined by the way applications use the object. In our experience, there is a diversity in the policies that should be applied to different objects depending on how the particular object is used.

The application software can provide the best information regarding the location for objects in a distributed object management system. Therefore we advocate an approach in which the application environment can *influence* the location policy by suggesting object locations to the object manager on an object-by-object basis. For example, the objects that make up a user's avatar should be located at the workstation where that user interacts with the system.

3.2 Caching and Consistency

In DVE applicationsSection 7.1 there are a large number of objects that never change state throughout their lifetime, e.g., walls and floors in a VE. If such objects are repeatedly referenced from each of the clients, e.g., to determine their VRML representation and room positioning, there will be considerable wasted network traffic and service request on the corresponding object storage location. In distributed systems, this kind of problem is classically handled by making copies of the read-only object, then distributing the copies to each client that wishes to read it, i.e., such objects are cached to the clients.

Caching becomes difficult when any copy of a cached item is updated by one or more of the clients. The other cached copies become invalid. Depending on the consistency model used by the system (the default is sequential consistency) a write operation requires that all cached copies be made consistent with one another. In the class of applications described by the scenarios, it is clear that some objects should have sequential consistency and others can have some weaker form of consistency; uniformly applying sequential consistency leads to large performance cost for small effective gain, although applying weak consistency may cause costly race conditions to occur on objects that need fine-grained sharing support.

4 The Gryphon System

In the Gryphon system, applications influence object management policy by dynamically providing *hints* and other directives regarding the location, caching, and consistency policy on a per object basis. Each object has the meta-interface and data added to handle accessing and maintaining state of the hints. These hints are analyzed by a *Gryphon*[1] (embedded in each object manager) which translates the hints and directives into object manager policy decisions for placing and caching each object. If no hints are provided, the object manager uses its default policies. The hints are evaluated at run time, allowing objects to be changed as their requirements change. It should be noted that these hints affect the object's distribution and its update rate on a global level and on a host-by-host basis. Some special distributed data (i.e. environment variables) exist solely for use by the subsystem in order to achieve global and per-user configurations. A user interacting with groupware on a modem will need different strategies than another user employing a faster 100Mb Ethernet connection.

4.1 Architecture

The general organization of the Gryphon architecture is shown in Figure 2. Each application uses the CORBA IDL interface to reference objects, i.e., it is assumed to use an ORB to reference shared objects. The base ORB is extended to include a Gryphon to accommodate policy hints as specified by application. Each object provides hints (and directives and environment variable values) using additional method calls described below. These extended methods are caught by the Gryphon policy module, then analyzed in the context of the state of the system and the nature of the collective hints regarding each object.

We presume that the base ORB has its own internal mechanism for implementing object location and caching, for example in Electra/Ensemble (E/E) [Birm97] [MS97]. The Gryphon uses E/E, invoking the internal location and update features via method calls provided by the ORB implementation with the enhanced mechanisms.

The technique we are using to change an objects location generates the same amount of traffic as our other methods. All methods fit into one network data packet since the objects we are using contain a small amount of information (position information and a URL to the VRML representation). In our implementation each object, *the_object*, resides in only one physical location and the cached copies, *cached_copy*, can be distributed throughout the distributed system. The Gryphon keeps a set of pointers to *the_object* and all the associated *cached_copy*(s). In order to change the location of an object the Gryphon modifies the label that designates which is *the_object*.

Our caching implementation requires the information regarding the set of pointers for an object and the label designating the actual object. The *get_state*() and *set_state*() method calls are used to transfer object state. It should be noted that these two methods do not include the additional Gryphon information embedded in each object. This information can be distributed independently using *get_hints*() and *set_hints*(). It should be noted that distributing the entire object state can be rather costly and result in superfluous data distributed in the update. In the VPR system this problem could be many orders of magnitude, resulting from the large VRML description of an object does not change. Our implementation of the VPR does not have this problem since the object only contains a reference to the VRML file. A variety of solutions can be found to this problem including distribution of deltas but our system implementation does not address this issue.

The cache update policies that result from the hints can all be implemented by the distributed Gryphon using 3 update techniques:

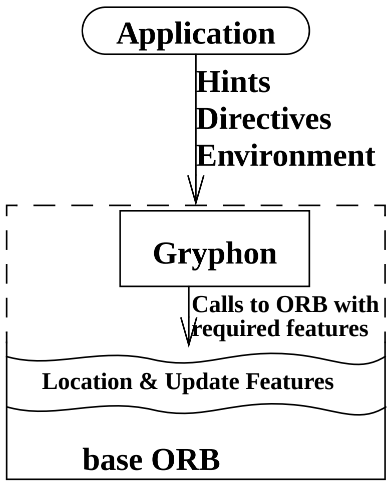
***the_object* push to *cached_copy*(s) - .** This technique is used when the Gryphon (where *the_object* is located) decides it is necessary to update all or a subset of the *cached_copy*(s). The *get_state*() method is called on the_object and then *set_state*() is called on the set of *cached_copy*(s). This results in synchronizing the state of the *cached_copy*(s) with that of *the_object*.

***cached_copy* push to *the_object* - .** The Gryphon with a *cached_copy* decides to update *the_object*. The *get_state*() method is called on *cached_copy* and then *set_state*() on *the_object*. If the other *cached_copy*(s) need to be updated then the Gryphon with *the_object* can use the push to *cached_copy*(s) technique.

1. Gryphon - A fabled animal with the body of a lion and the head and wings of an eagle. It is symbolically significant for its domination of both the earth and the sky, and its combination of intelligence and strength. The gryphon was alleged to watch over gold mines and hidden treasures. They are said to have hordes of treasure, which they guard endlessly.

***cached_copy* pull from *the_object* - .** The Gryphon with a *cached_copy* decides it needs to be synchronized with *the_object*. This Gryphon calls *get_state*() method on *the_object* and then *set_state*() on its *cached_copy.*

A fourth technique could be used with *the_object* pull from *cached_copy*(s). This technique is not used since there is a one-to-many update problem. Without additional information it must be assumed that all method calls modify the object, limiting the available implementation options of our system. The developer can label methods as *read-only* increasing the implementation options available to the Gryphon. Additional labelling specifying modification information could be supplied but is not addressed by our system.


Application
Hints
Directives
Environment
Gryphon
Calls to ORB with required features
Location & Update Features
base ORB
Figure 2:ORB with the Gryphon

Like the ORB, the Gryphon architecture is distributed. When a method is called that is specifically intended for the Gryphon, the method is processed by the portion of the Gryphon on the host where the object resides. This design eliminates the problems that could arise from having distributed Gryphons making decisions regarding conflicting requests.

4.2 Hints

Next we describe the hints, directives and environment variables used to communicate with the Gryphon. Along with the descriptions, we have provided the method calls we plan to implement in the Gryphon. A unique object containing all the environment variables exists at each client and is called the *gryphon_environment* object. It is modified using method calls and is used by the Gryphon to help in making decisions. In the descriptions that follows we will use *the_object* as the name of a distributed object in the system and use C++ like syntax.

- **Location** - Location can be considered to be more of a directive than a hint. It specifies where the object should reside. The method call *the_object.move*(host_name) is used to move *the_object* to the specified location and fix its location until another method is called to change the location. Host_name is a string representing the host name where the object should reside. The method *the_object.locate*() returns the current location of the object. Calling the method *the_object.unfix*() causes the object to use the hints we describe in deciding location. Calling the *the_object.fix*() method will cause the object to remain at its current location. This method is used to fix *the_object* at the location where the Gryphon has decided it should reside and has the same effect as calling *the_object.move*(*the_object.locate*()).
- **Users** - This hint specifies which clients are using the system and a numeric value representing the usage quantity. This numeric usage value has no absolute meaning but shows relative usage to the other clients. The Gryphon takes this information and attempts to place the object at the location with the highest utilization, also taking into account the rest of the hints and environment variables. Calling the method *the_object.usage*() returns an associative array of clients and their usage. The method *the_object.usage_clear*() removes all the usage information from the object. Calling the *usage* method with a client name returns the current usage value for that client while a call with a client name and a value as parameters sets the usage value. If the object is being cached, the usage is set to 0 for a client, the client will not have a cached copy and will need to directly access the object.
- **Relations** - Relations hints which objects should and should not be placed near each other. The method *the_object.relations*() returns an associative array of object names and their relation value. The sign and magnitude of the value represents the attraction or repulsion factor. Passing an object name as a parameter returns the association value while passing an object name and value sets the association. *the_object.relations_clear*() clears the current values.
- **Cached** - This hint is used to specify whether or not the object should be cached. The method calls include

the_object.cached() which returns true if the object is cached and false if it is not. Calling the method with the parameter true or the parameter false turns caching on and off respectively

- **Consistency** - If the object is being cached then this hint is used to specify the type of consistency to be applied. The method call *the_object.consistency*() will return the current policy while sending in a policy as a parameter will set the policy. The policies include: strong consistency, weak consistency, and domain acceptable consistency. The Gryphon will decide whether to propagate updates on read or write for weak consistency by using the information it has available. The domain acceptable consistency comes into play along with the Update Rates hint.
- **Update Rates** - This hint only has an affect when the domain acceptable consistency is selected as the consistency policy. The method *the_object.update_rates*() is used to specify the update rate for an object on a per client basis. It has the same syntax as the Users hint with *the_object.update_rates_clear*() used to clear the values. If the value is not set for a client then the Gryphon uses the other hints to decide on an update rate.
- **Environment Variables** - User resources and hardware resources can be specified to the Gryphon for each client using *gryphon_environment.resource*(). By passing in bandwidth and a value the Gryphon can be aware of how much data can be handled by the client. Other resources that can be specified include computing power, available disk space, etc. The *gryphon_environment.resource_usage*() method allows for querying about remaining resources. This feature can be used by the developer to throttle the application demands in order to remain within acceptable resource limits.

5 Performance Comparison

The VPR establishes a baseline application domain for various scenario-specific loads for the object management system. In order to analyze the behavior of a Gryphon implementation, models have been derived to characterize traffic patterns for six different object managers.

5.1 A Model of Network Interaction

The model is based on the VPR environment. The Object state changes when the object moves (it may also change due to other behaviors, though we believe that considering only movement is sufficient for this analysis); VPR processes and external processes (such as FLOATERS) move objects in the VPR. State changes are invoked by method calls (messages). All messages are considered to be the same size since they are small and fit into one network data packet. In Section 6 these six different object distribution techniques are analyzed for message traffic using the sample scenarios. Since hints are supplied on a per-object basis it is possible to have objects with and without location, caching, and update information coexisting in a single application. For analysis purposes we assumed homogeneous configurations of all the objects in each of the six systems and systems four through six assume perfect information supplied by the application developer. The reference to perfect information presumes that the developer or the application knows where the object needs to be located and uses the location hint to place the object. The following parameters are used to model message traffic:

- N = Number of Moving Objects
- M = Number of Object being modified at each process
- U = Update rate for each of the Moving Objects
- L = Number of processes using the object
- V = Number of processes that are VPR processes
- S = Number of static (not moving) objects
- F = Update rate of display frames
- R = Ratio of updates that get propagated to total updates generated

We derive models for:

- T_{VPR} = Amount of network traffic to each VPR processes in messages per second
- T_{app} = Amount of network traffic to each non-VPR processes in messages per second
- T_{total} = Total traffic in the network in messages per second

System 1 (Centralized object manager): This object manager is the centralized object manager implemented to provide the VPR prototype with distributed objects. There is a single server that allows objects to be cached to each client location. The object manager leaves consistency entirely up to the application community. In the model, we assume that any reference to an object requires consistency, thus the reference is remote. The message traffic is represented by:

T_{VPR} = U(N+M)

T_{app} = T_{VPR}

T_{total} = UL(N+M)

System 2 (ORB Central CORBA): This object manager is a centralized ORB. There is a single server that stores all objects, so any reference to an object requires a remote reference. In addition, since the ORB has no special knowledge of the application, the remote process must send a request message and receive the response message to determine the state of an object. Note that the expressions include references due to frame updates (a DVE needs to render objects, it would implicitly read each object at the frame update rate). Because the ORB is centralized and because of the amount of traffic, the server will likely be a bottleneck. The message traffic is represented by:

T_{VPR} = 2(MU+F(N+S))

T_{app} = 2MU

T_{total} = 2(NU+FV(N+S))

System 3 (ORB Distributed CORBA): The object manager is a distributed configuration of ORBs. All objects are randomly and equally distributed among the processes. The ORB is not centralized and local objects do not result in message traffic. The problem for measurement is that accesses that would have gone to the central ORB now go to the process where the object is located. Distribution addresses the implicit bottleneck due to centralized configurations. In T_{VPR} the first part of the expression represents read operations by the local client and the second part represents reads by external clients to the data stored on the local server.

T_{VPR} = 2MU(((L-1)/L)+((L-1)/L))+2F(N+S)(((L-1)/L)+(V-1)/L))

T_{app} = 2MU(((L-1)/L)+((L-1)/L)) + 2F(N+S)((1/L)V)

T_{total} = 2NU((L-1)/L) + 2F(N+S)((L-1)/L)V

System 4 (ORB with Opaque Location): The object manager includes a Gryphon capable of acting on location hints. Like System 3, objects are evenly distributed across processes but in this case they are assumed to be located on the client making the modifications.

T_{VPR} = 2F(N+S)(((L-1)/L)+(V-1)/L))

T_{app} = 2F(N+S)((1/L)V)

T_{total} = 2F(N+S)((L-1)/L)V

System 5 (ORB with Opaque Location and Caching): The object manager includes a Gryphon capable of acting on location caching hints. Like Systems 3 and 4, objects are evenly distributed across processes but in this case they are assumed to be located on the client making the modifications. The model reflects the fact that data are pushed to the clients instead of being pulled via request messages (i.e., we remove the 2X multiplier to reflect the absence of a send message).

T_{VPR} = MU(L-1)

T_{app} = T_{VPR}

T_{total} = NU(L-1)

System 6 (ORB with Opaque Location, Caching, and Update Policies): The object manager includes a Gryphon capable of acting on all hints. Like Systems 3, 4, and 5, objects are evenly distributed across processes but in this case they are assumed to be located on the client making the modifications. In this system caches can be kept out of sync for varying lengths of time (as specified by the application). In the other models, R varies on a per object and per host basis, but for System 6, R is fixed for all objects and users.

T_{VPR} = MU(L-1)R

T_{app} = T_{VPR}

T_{total} = NU(L-1)R

6 Analyzing the Gryphon Architecture

The models for message traffic can be used with various values with the independent variables representing different scenarios in a DVE. Before comparing system performance under different scenarios, let us consider the behavior of the different systems under a fixed load somewhat similar to Scenario B (with a varying number of processes) represented by:

- N = Number of Moving Objects = 20
- M = Number of Object being modified at each process = N/L
- U = Update rate for each of the Moving Objects = 2 updates/second
- V = Number of processes that are VPR = L/2
- S = Number of static (not moving) objects = 100
- F = Update rate of display frames = 24 frames/second
- R = Ratio of updates that get propagated = 10%

Figure 3 represents message traffic (in $\log_{10}$ messages per second) for cases from 2 to 40 client processes (i.e., letting L vary). For T_{VPR} and T_{app} on Systems 1 and 2, all data passes through a central location creating a bottleneck. This bottleneck is equal to the total number of messages in the system which is shown in the equations for T_{total}. Comparing Systems 2 and 3, which overlap in T_{total}, shows the values resulting by removing the bottleneck and dispersing the message traffic over the various processes. Also from Figure 3, in the plot for T_{VPR} Systems 3 and 4 have the same curve; so Systems 2, 3, and 4 have a much higher message traffic rate than Systems 1, 5, and 6. The T_{total} in Systems 1, 5, and 6 is significantly reduced when compared to Systems 2, 3, and 4. T_{app} reflects the same results except that applications that have no visible component have reduced traffic in System 2, since they have fewer objects locally, as a result of even distribution of objects over the available processes, and they do not access the external objects. We can conclude from these two figures that using programs with these characteristics will perform very well on system 1, 5, and 6 even as the number of entities increases. This data shows that the ability to specify location is useful but in this case caching is also required. It is interesting to see that System 1 performs well but it is to be expected since it was specifically designed for this type of application.

In Figure 5 we fix L at 20 and vary the number of moving objects, N, between 0 and 100. The figure shows that System 2, 3, and 4 do not perform well. System 2 does perform well in the equation for T_{app} since it is not reading any objects and no objects are stored on the local process. Figure 5 shows that as the number of modifiable objects grow it is crucial to have location and caching policies.

Figure 4 shows why it is also important to have update policies and Section 7.1 will clarify this need further. For this observation, N is fixed at 20, L at 20, and we vary U (the update rate) between 0 and 100. In Figure 4 the frequency of updates to each object increase while reads remain constant which results in excessive update distribution. This is the case in System 1 and 5 (which provide a strongly consistent cache). Even though the reads are only performed at the fixed rate of F, the updates are propagated at the increasing rate represented by U. With the aid of update policies these unnecessary updates can be eliminated.

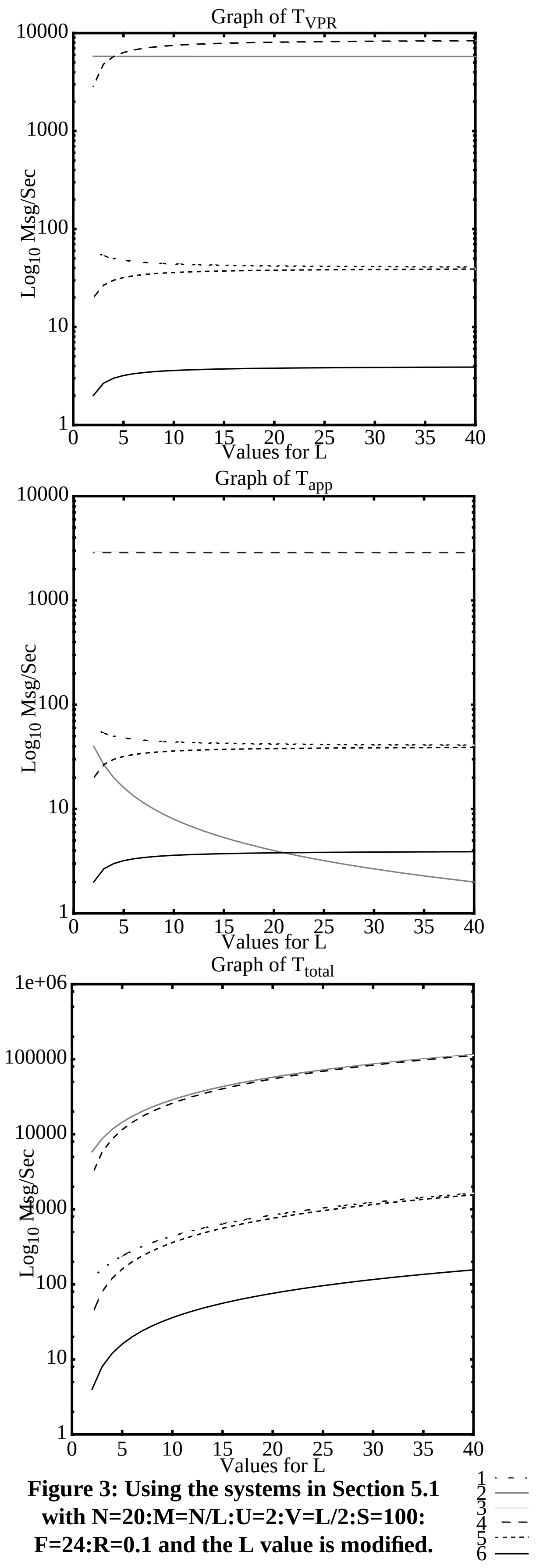


Figure 3: Using the systems in Section 5.1 with N=20:M=N/L:U=2:V=L/2:S=100: F=24:R=0.1 and the L value is modified.

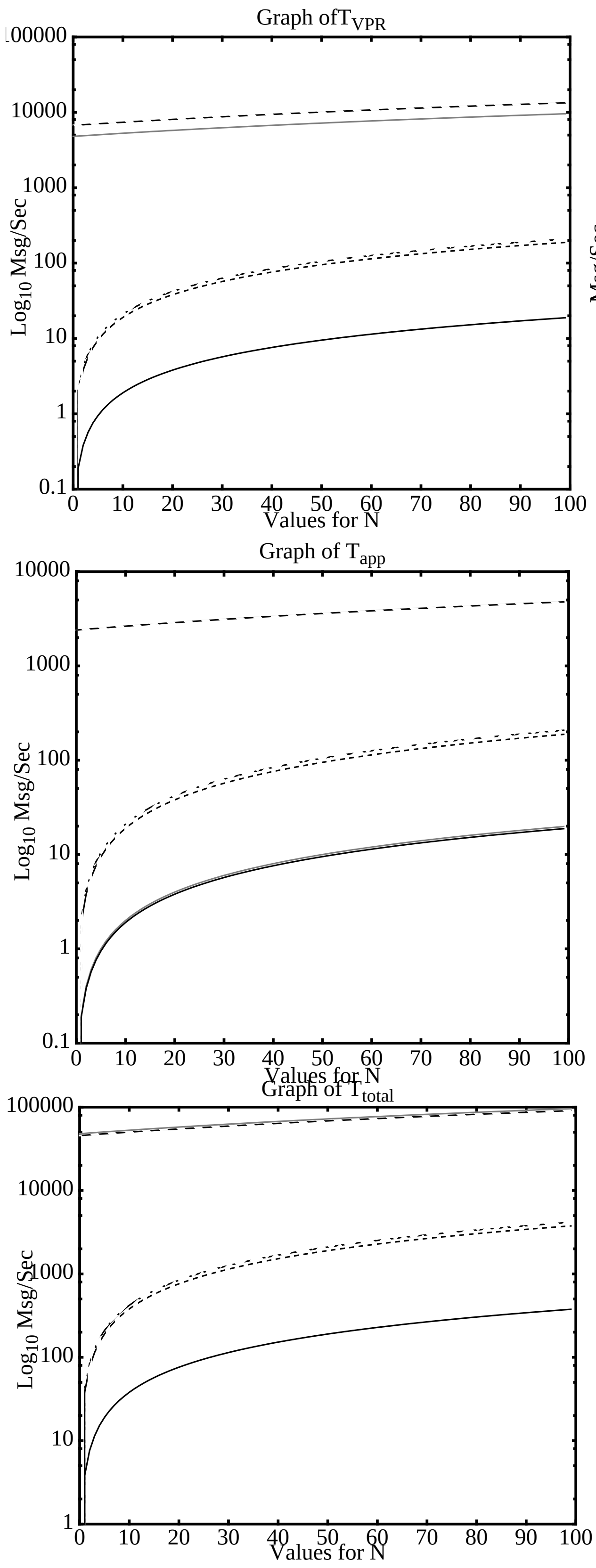
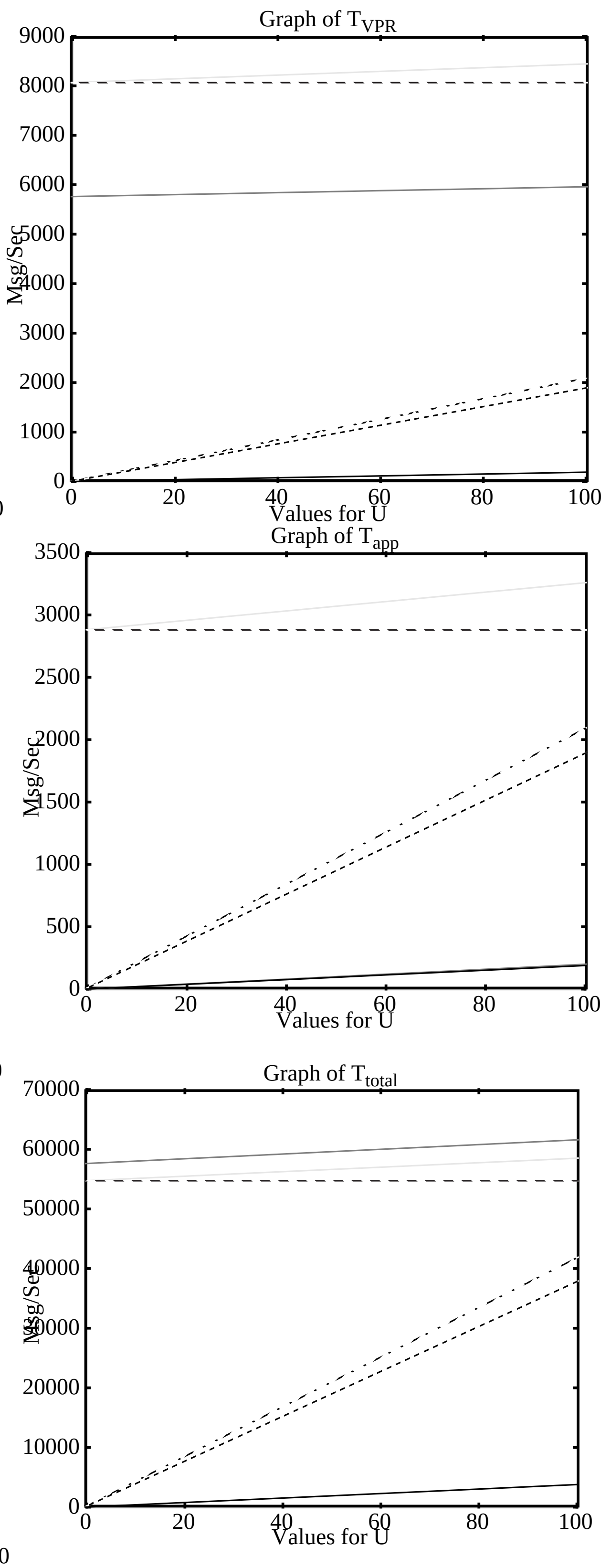


Figure 5:Using the systems in Section 5.1 with M=N/L:U=2:L=20:V=10:S=100:F=24:R=0.1 and the N value is modified.



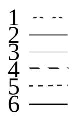


Figure 4:Using the systems in Section 5.1 with N=20:M=1:L=20:V=10:S=100:F=24:R=0.1 and the U value is modified

7 The DVE Prototype

We have constructed an extensible desktop DVE application, called the *Virtual Planning Room* (VPR), using C++ and distributed objects [NAB+97] [NBH+97]. The system environment was composed of a set of contemporary commodity workstations (200 MHz Pentium Pro PCs) interconnected with a 10 Mbps LAN. There was a spectrum of performance problems: Part of the bottleneck was in the rendering load on the graphics processor; another part was due to the complexity of generating information to be displayed; we address these issues in ongoing work described elsewhere [HBB+97]. We also suspected object communication to be a major issue and will apply the location and update policies to the VPR

The basic VPR configuration is a set of clients and servers, where each client machine implements an interface into the DVE (the VPR application) and an interface to the object system (see Figure 6). Domain-specific extensions can be added to the VPR by adding (other clients with) objects that implement the extension; domain extensions also use the object system. Thus the object management service is a global service in which shared object reside; they can be manipulated using remote method calls by objects in an instance of the VPR or in a domain extension.

The desktop DVE must represent visible/perceivable objects to its user. This is done by first defining a *world* as a collection of shared objects representing the entities that are logically significant in that world. For example, many of our VPR instances have a floor, walls, and miscellaneous other objects in the room. In particular, whenever a person enters the VPR (logs into a VPR session), that person's *avatar* is created as a collection of objects, and the corresponding objects are placed in the world.

Each VPR client implements a user interface; it determines which objects are visible to a person (based on the corresponding avatar's location and orientation), then invokes methods on the relevant objects to obtain their VRML description so they can be rendered at the client's user interface. If the VPR's user or a domain extension manipulates an object, then the appropriate method in the (shared remote) object must be called to update the object's state.

Domain extensions define VRML appearances, whenever relevant, and behaviors that represent the semantics of the extension. We have built an unoccupied air vehicle (the FLOATERS "blimp"[1]) navigation and control extension. The VPR domain extension is an interface to a blimp traveling in a large enclosed area. The blimp is controlled by VPR occupants who can see each other and a virtual representation of the blimp in the VPR. The collaborative control issues raised by FLOATERS are similar to the issues that arise with multiple researchers running simultaneous experiments on a space shuttle mission.

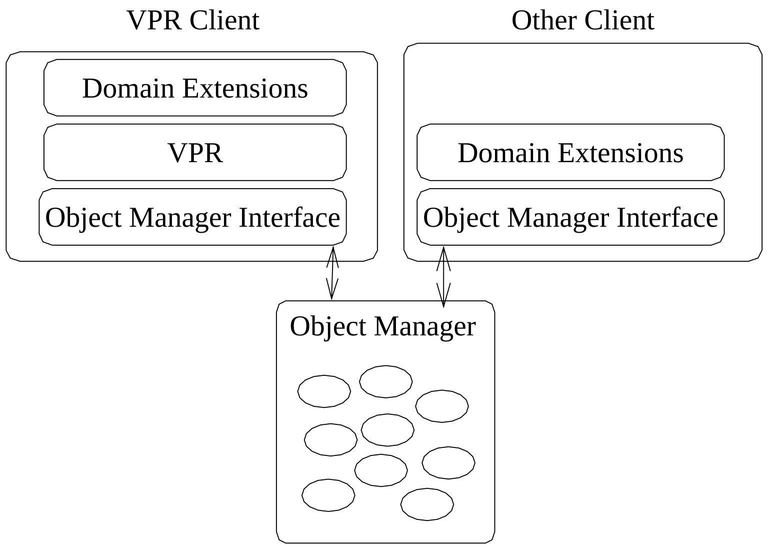


Figure 6:The VPR Organization

We believe that once DVEs become a cost-effective technology they will enable a diverse set of applications that share some of the properties of the VPR with the FLOATERS extension. Next we describe some scenarios that illustrate how the VPR and other DVEs might be used. The scenarios address varying scales of applications and varying access patterns that occur on objects. In Section 6 we will use these scenarios to represent the load on the object

1. FLOATERS was designed and built by Sam Siewert as a testbed facility for his thesis work on distributed real-time control [Siew97] [SNH97]. The FLOATERS extension is available because Siewert's work constitutes another part of the project.

management facilities.

Scenario A: Virtual Art Museum. The DVE contains a number of works of art available for viewing and discussion. A person enters the virtual museum, and may browse various works without communication. However, it is expected that the person will wish to find other people who are interested in specific works, then to discuss the work. The DVE represents a person's presence through the presence of an avatar in the environment; when one person sees a number of avatars near an interesting piece of work, then that person can join the group, view the work, and begin discussing it. The virtual art museum has a number of static objects with complex VRML specifications. Avatars move infrequently, but most other objects do not move at all.

Scenario B: An Art Gallery. An art gallery has the same essential properties as an art museum, with the difference being one of scale. The art gallery is a more intimate environment and tends to be encompassed in one room. The Art Gallery can also be viewed as a museum with domain optimizations in place to reduce the object information down to one room. Our FLOATERS application provides similar load characteristics to the art gallery.

Scenario C: Collaborative Office Work in the VPR. In a small department, people randomly visit different work areas in the office. Each worker has a set of supplementary tools that can be invoked on demand, e.g., word processors or database query interfaces. Each worker generally does not need to be intimately aware of the location of other workers in the office, except when there is collaborative work to be accomplished, e.g., a collaborative design review. The worker avatar objects change their state frequently, though other office objects do not tend to change at all.

Scenario D: Model-Based Virtual Environment. Elsewhere we have described model-based virtual environments as collaborative environments containing a model to provide context for the collaboration [Nutt95] [Nutt97]. In these systems, multiple workers interact with one another and with isolated parts of a larger artifact (a shared model of work, software methodology, etc.). In this scenario, an avatar may interact with many different components at a relatively high rate, but these pair wise interactions need not be updated at a high frequency at any workstation other than the one manipulating the objects.

Scenario E: A Weather Modeling Application. This scenario is not a DVE application but represents object usage in a highly data and computation intensive application. Each object is used as a data cell which can be thought of as one point of data in the large grid of data. In the weather modeling application, the algorithm breaks down data into small regional subsets and intense processing is performed on that data. After a large amount of processing is performed, data at the fringes of the subsets are distributed to a subset of other processes and then computation continues.

7.1 Sample Scenarios

Next we compare the performance of the various systems across various scenarios, i.e., we select a set of values for the independent variables to represent each scenario. The analysis demonstrates the advantages of using the Gryphon architecture to allow the object manager to be tuned to meet the needs of a specific application. Since our model is somewhat simplified, the representations of the scenarios are also somewhat simplified. Since our purpose is to evaluate the Gryphon architecture prior to prototyping it, we believe that the assumptions and results justify our prototyping efforts. Naturally, once the system has been implemented, we will be able to provide more precise measurements of the performance behavior.

Scenario A - A Virtual Art Museum: N = 1,000: M = 1 (only avatars move): U = 2 (averaged moves): L = 1,000: V = 1,000: S = 10,000 (includes art and building structure): R = 0.01 (only propagate 1/2 the updates for 20/1,000 of the patrons since an avatar can only see a small subset).

Scenario B - An Art Gallery: N = 20: M = 1: U = 2: L = 20: V = 20: S = 100: F=24: R = 0.5 (only propagate 1/2 the updates).

Scenario C - A Collaborative Office Work in the VPR: N = 100: M = 20: U = 5: L = 5: V = 5: S = 1,000: F = 24: R = 0.017 (only propagate 1/3 of updates and only see 5/100 of the objects).

Scenario D - A Model-Based Virtual Environment: Assume 5 participants each modify 3 objects at a time. Also assume each object in the environment controls itself, so no application is needed. N = 20: M = 4: U =5: L = 5: V = 5: S = 10,000: F = 24: R =0.1 (the client only sees some modifications and reduces the propagation frequency).

Scenario E - A Weather Modeling Application: N = 10,000: M = 1,000: U = 1,000: L = 10: V = 0: S = 0: F = 0: R

= 0.0001 (the data only need to be redistributed once every 10 seconds).

Table 1: Scenario Performance

Models	Scenario A	Scenario B	Scenario C	Scenario D	Scenario E
T_{VPR} 1	2,002	42	600	120	11,000,000
2	528,004	5,764	53,000	481,000	2,000,000
3	1,054,950	10,952	84,800	769,600	3,600,000
4	1,054,940	10,944	84,480	769,536	0
5	1,998	38	400	80	9,000,000
6	20	19	7	8	900
T_{app} 1	2,002	42	600	120	11,000,000
2	4	4	200	40	2,000,000
3	528,008	5,768	53,120	481,024	3,600,000
4	528,000	5,760	52,800	480,960	0
5	1,998	38	400	80	9,000,000
6	20	19	7	8	900
T_{total} 1	2,002,000	840	3000	600	110,000,000
2	528,004,000	115,280	265,000	2,405,000	20,000,000
3	527,476,000	109,516	212,000	1,924,000	18,000,000
4	527,472,000	109,440	211,200	1,923,840	0
5	1,998,000	760	2,000	400	90,000,000
6	19,980	380	34	40	9,000

Key:
System 1. - DOM
System 2. - Central CORBA
System 3. - Distributed CORBA
System 4. - CORBA+Location information
System 5. - System 4+Caching
System 6. System 5+Update Policies

Table 1 shows T_{VPR} (the amount of network traffic to each VPR processes in messages per second), T_{app} (the amount of network traffic to each non-VPR processes in messages per second), and T_{total} (the total traffic in the network in messages per second) that each scenario would incur using each system. Observe that System 2 (centralized CORBA) and System 3 (distributed CORBA) do not perform well for many cases due to their location transparency policy. The table illustrates the expected performance for System 4 in Scenario E, where objects are placed in an application-favored location and unnecessary caching is not used resulting in a large performance gain compared to the cached location transparent approaches. It should be noted that in Scenario E, except for System 6, the systems are not showing the extra messages that occur from the infrequent reads of small portions of the data. For many applications, caching (Systems 1 and 5) can also result in large performance gains while in some applications, Scenario A and E, caching results in unnecessary cache consistency updates. System 6 with update policies show large reduction in message traffic except in Scenario B, where System 6 shows "only" a reduction by half of the number of messages generated by System 5 without the update policies. This is a actually a good example of why update policies are important since Scenario B can be interpreted as Scenario A with update strategies already applied. The clear conclusion is that location, caching, and position policies together always generate the best performance. With input from the developer and the three application-specific policies one can create a usable system tailored to the specialized application domain.

Table 1 shows huge differences in the message traffic rate; System 6 uses all hints and directives, and is generally able to provide the highest level of performance. For T_{total} its message rate is only a fraction of a percent of centralized and distributed CORBA systems for all 5 scenarios. We conclude that the Gryphon architecture is worth prototyping for more comprehensive evaluation. (For example, the prototype system will allow us to observe the performance of a real system under real load due to objects with tailorable policies.)

8 Conclusion and Future Work

We implemented and used our VPR with the FLOATERS domain extension. As expected, the VPR was far too slow for many of the applications we wished to support (similar to the scenarios). Our research group is attacking the performance of the underlying software in various ways, all based on using application-specific knowledge to assist the system in choosing its resource management policies. This paper addresses distributed object efficiency problems which can be applied to the FLOATERS and other scenarios.

Our experience led us to believe that some of the efficiency problems could be overcome through developer and application input to the subsystem. We then hypothesized the critical performance factors for distributed objects — location, caching, and consistency — resulting in the basic design of the Gryphon.

The goal of this paper is to explore the Gryphon design to determine if tailorable policies would have a significant influence on the performance of the VPR and its applications. Rather than build the system and testing it, we have chosen to conduct the analysis described here to get an idea of the performance gains we might expect with a Gryphon-based system. Our analytic models are driven by data derived from the kind of scenarios we see driving the VPR. The model indicates that the reduced message traffic improvements of our design can be significant — it can reduce the message traffic to a fraction of a percent compared to a standard CORBA implementation (depending on the scenario). The key improvement comes from the application's ability to tailor the object management policies according to its need and use. We are convinced that this analysis justifies further consideration of the Gryphon model, so now we are prototyping the Gryphon on Electra/Ensemble.

Acknowledgments

The authors were partially supported by NSF grant number IRI-9307619. Jim Mankovich (with the help of the first author) designed and implemented the VPR.

References

[AS94] Ahamad, Mustaque, and Shawn Smith, "Detecting Mutual Consistency of Shared Objects," *Proceedings of Workshop on Mobile Computing Systems and Applications*, December, 1994

[BAB+96] Bellissard, Luc, Slim Ben Atallah, Fabienne Boyer, and Michel Riveill, "Distributed Application Configuration," *Proc. 16th International Conference on Distributed Computing Systems,* Hong-Kong - IEEE Computer Society, just did not have time to May, 1996, 579-585.

[Blac97] Black Sun, *Black Sun Community Server*, June, 1997. Computer software available URL: http://ww3.blacksun.com/launch/server.html

[Birm97] Birman, Ken, *Ensemble system web pages*, 1997. Available URL: http://simon.cs.cornell.edu/Info/Projects/Ensemble/overview.html

[BN96] Brown, Marc H. and Marc A. Najork, "Distributed Active Objects." Digital SRC Research Report, 141a. April, 1996.

[Chap97] Chappell, David, "An Introduction to ActiveX and COM," *USENIX 3rd Conference on Object-Oriented Technologies and Systems (COOTS'97),* Portland, Oregon, June, 1997

[Funk95] Funkhouser, Thomas A., "RING: A Client-Server System for Multi-User Virtual Environments," *ACM for 1995 Symposium on Interactive 3D Graphics*, Monterey CA., 1995, 85-92.

[HAB96] Hassen, Saniya Ben, Irina Athanasiu, and Henri E. Bal, "A Flexible Operation Execution Model for Shared Distributed Objects," *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '96),* San Jose, CA, October, 1996.

[HBB+97] Humphrey, Marty, Toby Berk, Scott Brandt, and Gary Nutt. "Dynamic Quality of Service Resource

Management for Multimedia Applications on General Purpose Operating Systems." *1997 IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, San Francisco, CO, December, 1997.

[Holb95] Holbrook, Hugh, "Distributed Interactive Simulation," June, 1995, Available at URL: http://www-dsg.stanford.edu/DSGHomePage.html.

[MS97] Maffeis, Silvano, and Douglas C. Schmidt, "Constructing Reliable Distributed Communication Systems with CORBA," *IEEE Communications*, Vol 14, No. 2 (February 1997).

[Nutt95] Nutt, Gary J., "Model-Based Virtual Environments for Collaboration," University of Colorado, Department of Computer Science Technical Report No. CU-CS-800-95, December, 1995.

[Nutt97] Nutt, Gary J., "The Evolution Toward Flexible Workflow Systems," *Distributed Systems Engineering*, Vol. 3, No. 4 (December 1996), 276-294.

[NBH+97] Nutt, Gary J., Toby Berk, Scott Brandt, Marty Humphrey, and Sam Siewert, "Resource Management for a Virtual Planning Room," *Proceedings of the Third International Workshop on Multimedia Information Systems*, Como, Italy, September, 1997.

[NAB+97] Nutt, Gary J., Joe Antell, Scott Brandt, Chris Gantz, Adam Griff, and Jim Mankovich, "Software Support for a Virtual Planning Room," University of Colorado, Department of Computer Science Technical Report No. CU-CS-800-95, December, 1995.

[OMG95] OMG, "The Common Object Request Broker: Architecture and Specification," *OMG Technical Document, PTC/96-03-04*, July, 1995.

[SA97] Schwan, Karsten and Mustaque Ahamad, "The COBS Project - Configurable OBjectS for High Performance Systems," May, 1997.

[Scot97] Scott, Christopher W., "ANSAware and ORBeline: a comparison," 1997. Available at URL: http://www.dcc.ufal.br/~cws/

[Siew97] Siewert, Sam, *Operating Systems Support for Parametric Control of Isochronous and Sporadic Execution Streams in Multiple Time Frames*, University of Colorado, Department of Computer Science, Ph.D. dissertation proposal, June, 1996.

[SNH97] Siewert, Sam, Gary J. Nutt, and Marty Humphrey, "Real-time Parametric Controlled In-Kernel Pipelines." *Proceedings of the Thirds IEEE Real-Time Technology and Application Symposium*, Montreal, Canada, June, 1997.

[TCH97] Tanenbaum, Jay M., Tripatinder S. Chowdhry, and Kevin Hughes, "Eco System: An Internet Commerce Architecture," *IEEE Computer,* Vol. 30, No. 5 (May 1997), 48-55.