A PLATFORM FOR CONSTRUCTING VIRTUAL SPACES

By Per Brand, Nils Franzen, Erik Klintskog, Seif Haridi

SICS, Sweden

perbrand, nilsf, erik, seif @sics.se

Keywords: DCE, distributed programming, distributed graphics

1 ABSTRACT

Virtual spaces (worlds) applications are among the most complex of distributed applications. They are both distributed and open. Minimizing network latency, fault-tolerance, persistence, resource control, and security are also important aspects. Designing and implementing virtual spaces is currently difficult in that the already not insignificant complexities of program functionality, distribution, openness, and efficiency are interwound and cannot be tackled separately.

We present a distributed programming language, distributed-Oz, that greatly reduces the complexity of distributed programming by clearly separating the different aspects of distributed programming. The design and implementation of distributed-Oz is ongoing work, but considerable progress has been made. The current prototype demonstrates network transparency, that computations behave the same way regardless of how the computation is partitioned between different sites. This is the basis for realizing clean separation of the functionality aspect from other aspects. Network awareness allows the programmer to predict and control network communication patterns. The current system gives the programmer the means to tackle separately the aspects of openness, efficiency (minimizing latency), distribution, and functionality.

We have extended distributed-Oz with a tool for graphics in a distributed setting. This extends the idea of network transparency and network awareness to graphics. From the programmers point of view graphics programming for a multi-user application is virtually the same as programming for a single-user application. The differences are necessary extensions for achieving network and site awareness, such as visualization control (deciding which users should see what).

Finally we consider virtual space applications, and propose a number of abstractions for use by developers of virtual spaces, relating them to the properties of distributed-Oz upon which they are based.

2 INTRODUCTION

Virtual spaces or *virtual worlds* are (or will be) characterized by large numbers of people connecting and disconnecting regularly from all over the Internet. They interact with each other, the objects of the space they're in, as well as software agents. The agents themselves may be long-lived and migratory – moving from space to space. A virtual space has an existence apart from any participant, and is continually changing in time, as the various participants interact with and change the space they're in. When a person reenters a virtual space it is rarely the same as before, not only because new participants have entered and old participants have left the space, but also because the space itself has changed.

We choose to use the word *space* rather than *world* in virtual space, as we feel that the word world has a too limited a connotation. This article focuses on the requirements for tools for constructing a wide range of graphical distributed applications, not necessary limited to the kinds of applications that come into mind on hearing the concept virtual world. For example, collaborative work environments can also be thought of as virtual space applications.

Virtual spaces as programming applications are thus both distributed and open. Efficiency, in the sense of minimizing network latency is also important. Fault-tolerance is necessary if the distributed application is to be more robust than the most failure-prone participating site. Persistence may be needed for important virtual spaces that must survive site crashes. Various security aspects may come into play. Sites must be able to control the resources used by potentially hostile migratory agents. The limitations of non-privileged participants must be maintained. There may be private information between a participant and the virtual space that is not to be shared with other participants.

For the most part we will assume that the desired consistency model of virtual space application is sequential consistency, i.e. that all participants see the same order of changes to the virtual space. It may be possible or even desirable, to relax consistency for some things. This would be done as an optimization, lowering the consistency requirements lessens the need to synchronization across the network, and hence latency. Sequential consistency is however clearly desirable in many cases and is the most difficult to achieve, which is why we focus on this model.

2.1 Organization of the paper.

This paper is organized as follows. In the rest of this introductory section, we first discuss the problems of distributed programming and identify the desired characteristics of a good distributed programming language. The described desired characteristics are important for all kinds of distributed applications, not only virtual space applications. Thereafter we briefly introduce distributed-Oz, a general-purpose distributed programming language. Distributed-Oz is still under development, but a prototype implementation exists today that demonstrate some, but not all, of the desired characteristics. As virtual space applications are often graphical, it is important to be able to easily program complex user-visualizations. We therefore, discuss the special problems with graphics in a distributed programming language. Finally, we look at the situation today with respect to the tools and languages for constructing virtual space applications.

In section 2 an informal presentation of distributed-Oz is given. A detailed presentation of distributed-Oz is beyond the scope of this paper. The basic language constructs, which are independent of distribution, are briefly described, For the reader unfamiliar with Oz, the individual concepts will be familiar from other centralized programming languages, though not the combination. The constructs of Oz include first-class procedures, classes and objects, are give the programmer powerful mechanism for creating abstractions. Distributed-Oz was developed with the express purpose of minimizing the difference between distributed and centralized programming. Our chief purpose here is to describe the concepts and constructs that are needed for realizing distributed applications.

In section 3, graphics in distributed applications is considered. The focus here is on achieving one-to-many visualization; an event occurs on one site, and this should result in visualization changes on many other sites. There are a number of difficulties in achieving, in a controlled way, distributed visualization. These difficulties will be described in detail. Thereafter we present our solution, a multi-user graphical tool, DTk, we have developed in distributed-Oz.

In section 4 we consider the properties of virtual-space applications. We argue that the constructor of virtual spaces needs a distributed programming language and system that 1) makes distribution easy and 2) has a powerful mechanism for abstraction, and 3) provides for controlled distributed graphics and visualization. We claim that distributed-Oz with the developed graphical tool DTk meets these requirements. We exemplify with a number of abstractions that are easily built in distributed-Oz and relate the abstractions to the underlying properties and constructs of distributed-Oz that the abstractions build on.

2.2 Aspects of distributed programming

We can identify the following aspects important for the designer and implementers of virtual spaces, *functionality, distribution, efficiency (latency), openness, fault-tolerance and persistence, resource control and security, scalability*. Having to take into account all these aspects in one application greatly increases the complexity of program development, testing, and maintenance. This increase of complexity is partly inherent, as there are more aspects to consider than for centralized applications. But the biggest increase of complexity with today’s systems is due to the fact that the various aspects are interwoven with each other, and cannot for the most part be tackled separately. The resultant complexity, the tangling of the various aspects, tends to produce a combinatorial explosion in complexity.

It is well known that software development is a costly, complex, and time-consuming process even on single-user applications for centralized systems. But in the distributed perspective this activity is simple in that only the one aspect, *functionality*, is handled. Now consider the process of making the same application distributed. The distribution aspects need to be dealt with; data and subcomputations need to be partitioned between sites. To some extent we may freely distribute subcomputations and data, and to some extent this is forced because we have to allow users to participate across the net. Having decided this we must modify the program (realizing the desired functionality) to transfer data and code between sites and to synchronize between distributed but ordered subcomputations.

We now consider network latency. Latency may be unacceptable if computations need to repeatedly fetch data from other sites. Data may need to be moved to or cached at specific sites. Distribution and functionality are both affected. Synchronization and consistency must be considered.

Centralized applications fail or crash in the entirety, but distributed applications may partially fail or crash when a fault occurs on one of many sites. As the number of involved sites increases the probability of such faults grows, and error recovery mechanisms become crucial. Faults will need to be detected, analyzed, and computations and data will need to be redistributed. Persistence may be necessary for vital functions.

There are many security aspects to consider in distributed, open applications. We do not consider the question of secure network communication, as this is almost standard. More difficult is the need to restrict access to some data between sites while allowing access to other data. But even this is not always enough; it may be necessary to transfer objects (data and code) between sites where different sites are given different capabilities. For example, consider a bulletin board in a virtual space. We may want to give all users the ability to post messages, i.e. change the state of the bulletin board, without giving them the ability to remove messages which is a right restricted to the owner (an advantage that the virtual bulletin board has over a real one).

The problems of resource control and security in applications that make use of migratory agents or let users directly or indirectly inject subcomputations onto other sites are apparent. The memory and processor resources that agents can use must be limited. With foreign agents now running concurrently on the same site it is no longer enough with a site-based security mechanism. The state of agents may need to be protected from arbitrary manipulation by thread involved with other activities on the new site.

The importance of scalability can be illustrated by the example of a virtual space manager running on one site communicating with a small number of participants. It is then found that the number of participants of the virtual space has grown too large for good performance. The key to scaling up is to distribute the virtual space managing functionality between sites, typically multi-processors or multiple computers on a LAN (local area network). This affects distribution, functionality and efficiency. Synchronization problems between the partitioned virtual space manager may arise. Efficiency would be greatly reduced if the same mutable data is repeatedly needed by one of the managers was held by another, and all access and update would require communication across the LAN.

Finally, openness requires not only that users can easily connect and disconnect to running applications but also that latecomers are not penalized, and can be given the same environment or state as users that have been connected for some time or even from the start. This is not trivial as programming languages and tools that have been designed for single users do not, as a rule, provide a means of extracting the current state of the running application. New users may need sufficient information to be able to create an environment identical to that which he would have had if he had been participating from the beginning.

2.3 Distributed Oz

Distributed-Oz is a general-purpose programming language for distributed applications that has been and is specifically designed to separate the different aspects of distributed programming [2,3].

Distributed-Oz is based on Oz-2, a programming system developed in collaboration between SICS and DFKI [1,4]. Oz is a multiple-threaded data-flow object-oriented programming language with excellent support for constraint solving and encapsulated search. It is a language with lexical-scoping, and all concepts are first-class including procedures, objects, methods, threads, and spaces.

Distributed-Oz does not yet provide support for all of the aspects mentioned above. The current design and implementation, does however, address the aspects of functionality, distribution, as well as efficiency in the sense of latency control. To a limited degree scalability and security are handled. This work in ongoing, and the issues of fault-tolerance, scalability, and resource control are currently being tackled.

The central idea of distributed-Oz is that of *network transparency*; computations behave in the same way independent of the distribution structure. The language semantics are obeyed independent of how (and even if) the computation is partitioned between sites. This cleanly separates the aspect of functionality from other aspects. Network-transparency requires a distributed shared computation space providing the illusion of a single network-wide address space for all Oz-entities (including threads, objects and procedures).

The system also provides flexible *network awareness*, giving predictability and programmability over network communication patterns. Network awareness is the basis of the way in which the programmer can partition computations between sites, i.e. dealing with the distribution aspect. With network-awareness the programmer can deal with efficiency issues. One of the most important features is the ability to control the mobility of objects.

To achieve scalability it is important to be able to partition large computations between sites which is only possible given network transparency. With network-transparency openness becomes easy, reducing to a question of establishing the initial connection between sites.

2.4 Graphics

Many virtual space applications, if not already now, will in the future make use of sophisticated graphics. The virtual space will be visualized in two or three dimensions, and the participants can interact with the space and other participants through a GUI (graphical-user interface). The interweaving of functionality, distribution, efficiency, and openness aspects is very noticeable in graphics.

The virtual space is graphically visualized and for the most part the users should see the same image (modulo network delay). Irrespective of the number of participants a state change in the virtual space needs to be communicated to all participants. The final step in the process (the back-end) for each participant is some communication with his terminal (or other I/O device). Various graphical tools are available that provide for abstractions that hide to various extents the low-level details of I/O directives from the programmer.

It is generally not acceptable that visualization change be made on the initiative of the participants, e.g. through repeated inquiries, as this increases network latency and traffic, but rather that changes are multicast to all participants by the virtual space manager. It is also desirable that the messages are as succinct as possible to minimize the size of network messages.

New users may enter the virtual space (openness), and need to be given enough information to be able to create the correct visualization on the user's terminal. Single-user graphical tools tend not to provide enough information to generate the correct visualization. In a single-user application there is generally no need for this functionality and the graphical state of the application is partly implicit. One complete characterization of the graphical state of an application is the complete sequence of graphic events since the start-up of the application, the *uncompressed history*. In many applications, including virtual spaces, the application will run for indefinite periods of time and the uncompressed history can easily become prohibitedly long. The history can, of course, be compressed with standard compression techniques but much more can be done making use of domain knowledge. Consider the case where an object enters a room, moves around from place to place, and then leaves the room. Thereafter all graphic events having to do with that object can be removed from the history, which would not be possible with standard compression techniques.

We have extended distributed-Oz with a graphical interface for programming distributed graphical applications, which we call distributed-Tk, or DTk. DTk extends the ideas of network-transparency to graphics. Sites not only share a common network-wide computation space, but can also share visualizations. DTk is, as the name suggests, based on Tk, and uses Tk (single-user graphical tool) as a back-end. The methodology used is, we believe, applicable to building distributed graphics using other single-user graphical tools as back-ends.

2.5 The situation today

Traditional text-based MUDs (multi-user dungeons) can be seen as precursors to virtual spaces. They exemplify the simplest kind of server-based distributed application. The only data exchanged between sites are byte strings, and all computation is done at the server site. The application at the server is not concurrent, but a normal sequential computation that takes messages from a mailbox, processes them, and then proceeds with the next message. During the processing of a message the server may send messages to other sites. Distribution is limited to a simple message- or mail-handling system.

It was unlikely that the designers and developers of MUDs did not want sophisticated GUIs, migratory agents, and the like, but they were, and to a large extent are, limited by the technology of their day. It was necessary to keep the distribution aspects as simple as possible, in order to concentrate on the functionality of their application. This approach taken was to avoid the tangling of the various aspects of distributed programming by making very little use of the possibilities of distributed systems.

All systems for distributed programming that we know of except Emerald and Obliq, do distributed execution by adding a distribution layer on top of a centralized language. This has the disadvantage that distribution is not a seamless extension to the language, and therefore distributed extensions to language operations have to be handled by explicit programmer effort. This is not only an extra burden on the programmer, but directly entangles the functionality, distribution, and efficiency aspects of distributed programming.

Emerald [6] and Obliq [7] are concurrent languages that most closely resemble distributed-Oz. Objects can be shared across the network, and can be moved between sites. Distributed-Oz is a considerable richer programming language than Emerald, and somewhat richer than Obliq. A more detailed comparison is made in [3].

Consider graphics there are languages and tools that provide for distributed graphics. Simple broadcast visualization can be achieved with XTV [8]. Tools like Tcl-dp [9] can be used to provide distributed graphics in a more flexible manner, but are not integrated into a distributed-programming language and require the programmer to deal with distribution and broadcast visualization explicitly.

Visual-Obliq [10] is an open graphical extension to the distributed programming language Obliq, much as DTk is a graphical extension of distributed-Oz. However, Visual-Obliq provides for distributed forms, tables, buttons, etc., but is not a general-purpose graphical tool. It does not, for example, provide the necessary functionality in order to program a distributed drawing board. Much of the difficulty in combining openness with graphics, extracting the complete graphical state when new users connect, is thus avoided. For example, consider the example of two rectangles on a canvas (drawing area) where the visualization is dependent on the order of creation.

A number of commercial distributed graphical collaborative tools are available today, e.g. Lotus notes. The graphics can be quite sophisticated. Of course, they are not general-purpose graphical programming tools. We would expect that considerable time and effort was put into these applications, and that much of the effort went into explicit control of distribution, communication, etc.

Distributed virtual-reality applications, e.g. DIVE [12], are graphically the most interesting. Not only because they offer 3-D visualization, but also because of their flexibility – all kinds of graphical entities can be displayed. Once again, they are not, as a rule integrated into a general-purpose distributed programming language. Visualization is distributed, but the computations behind them are ordinary centralized applications. Also, the underlying communication mechanism is unsynchronized multicast, so they do not offer sequential consistency.

3 DISTRIBUTED OZ

3.1 The programming language Oz

The programming language Oz is a high-level concurrent programming language. It is based on a computation model combining concepts from functional programming, logic programming, object-oriented programming and concurrency theory. The most important concepts related to concurrent and distributed programming are:

1. Concurrent objects: Objects are the primary structuring concept for concurrent programming. Objects combine data abstraction and state. Objects may be locked to achieve mutual-exclusion. Locking is reentrant (like Java). Object invocation is synchronous.
2. Lexical scoping: The initial references of a program are determined by its static structure. Other references are obtained during execution.
3. Full procedural abstraction: Procedures are first-class citizens.
4. Full compositionality: Full procedural abstraction is generalized. All language features are first-class, including objects and classes. Through full compositionality it is possible to transfer arbitrary computational tasks between threads.
5. Stateless data: Oz provides records, tuples etc. as stateless data structures. The distinction between stateful and stateless data structures is clear.
6. Abstract store: Oz computes over an abstract store containing abstract entities. This is important for automatic memory management including garbage collection.
7. Threads: For concurrency, Oz provides for fine-grained thread serving as virtual processors. Threads are also first-class.
8. Logic variables: Logic variables are the basis of data-flow synchronization and communication. For example, a thread can query by sending a fresh logical variable along with the variable. The answering thread will in due course answer the query and instantiate the variable. The inquiring thread need only suspend if and when it needs the value of the variable and finds the variable still unbound.
9. Ports: asynchronous communication channels that support many-to-many communication. A port consists of a send procedure and a stream. A stream is a list whose tail is a logic variable. The sends appear at the end of the stream. The order of sends within a thread is maintained, while no guarantee is given on the order between threads. A reader can wait until the stream's tail becomes bound. Multiple readers waiting on the same tail are informed of the binding simultaneously.

With first-class procedures, objects, and ports, it is straightforward to program asynchronous objects (each invocation is executed in its own thread), active objects (each object has one thread serving it), synchronous send, etc [3].

The entities of Oz can be divided into three categories, 1) stateless, 2) stateful and 3) single-assignment. Examples of the first are records, tuples, atoms, numbers and strings, procedures, classes and methods. Examples of the second are objects, or more precisely, the object state. The third category consists exclusively of logical variables. Logical variables can also be considered stateful, but as opposed to object-state the state is changed only once upon binding the variable.

3.2 From Oz to Distributed-Oz

Distributed-Oz is a conservative extension to Oz for distributed-programming. We started from four basic requirements that are generally agreed to be important in a distributed setting.

3.2.1 Network-transparency

Network-transparency means that computations behave the same way independent of the distribution structure. Language semantics are obeyed, irrespective of how the computation is partitioned onto multiple sites. This requires a distributed shared computation space, which provides the illusion of a single network-wide address space for all entities of Oz. The distinction between local and remote references (an entity on a different site) is invisible to the programmer. Another requirement is concurrency, the language semantics must provide for multiple computational activities.

Network-transparency is essential for the clean separation of the functionality aspect of distributed programming from other aspects. For example, in order to deal with distribution and efficiency aspects it may be necessary to move computations and stateful data between sites. Without network-transparency such a move would affect functionality, and we would back to the tangling of aspects that we wish to avoid.

3.2.2 Network-awareness

Network-awareness means that network communication patterns are both predicable and programmable. The communication patterns should be simply and predictably derived from the language entities being used. The language should be flexible enough to be enable the programmer to achieve the desired communication patterns.

Network-awareness is coupled to state-awareness, and that the three kinds of basic entities of distributed-Oz are handled differently. Stateless data is replicated between sites. Once a site has referenced the data, the data is available locally, and hence there is no network latency involved in further access[1].

[1] Data is generally replicated eagerly, but a discussion of this is beyond the scope of this paper. It is easy to build abstractions to achieved lazy replication, when desired.

The most important aspect of network-awareness is the location of stateful data, e.g. an object, or more precisely an object-state. Such data must reside on exactly one site at any one time. As part of network-awareness distributed-Oz provides *mobility control* of stateful data. The programmer can choose to make an object stationary (the object resides on the creation site forever), or mobile (the object moves to the invoking site). Operations on stationary objects require a network operation for each remove invocation. Operations on mobile objects may require a network operation on the first invocation, but thereafter further invocations will be local[2]. Controlled mobility is also programmable, where some sites are given the capability of moving the object and others are not.

Network awareness is necessary for dealing with the distribution and efficiency aspects of distributed programming. The fact that objects can be given a mobility behavior independent of their definition gives clean separation of the efficiency and functionality aspect.

3.2.3 Latency tolerance

Latency tolerance means that the efficiency of computations is as little affected as possible by the latency of network operations. Concurrency provides latency tolerance between threads. When one thread on site is suspended on a data-dependency (across the net or between threads on the same site), other threads continue.

Logical variables also provide latency tolerance by decoupling the operations of calculating and using the value, i.e. the data dependency, from the operations of sending and receiving the value. In distributed-Oz when a variable is bound the new binding is multicast to all sites that have references to it.[3]

3.2.4 Language security

Language security guarantees integrity of computations and data. This is provided in Oz by giving the programmer the means to restrict access to data. Data is represented as references to entities in the shared computation space. This address space is abstract because it provides a well-define set of basic operations. In particular, unrestricted access to memory is forbidden. One can only access data to which one has been given an explicit reference. This is controlled through lexical scoping and first-class procedures.

3.2.5 Openness

Openness is achieved by new builtins that store and load references to ongoing computations onto a file identified with a URL. For example, a server site (Oz-process) may store a reference to a port in a file. A thread on the server reads from the associated stream. Another site, the client, may load the reference, and use that port to make a request of the server. For example, send a tuple containing an object, a port, a variable, and a procedure. The procedure is replicated. The server and client now share two ports (one in each direction), an object, and a variable.

4 DISTRIBUTED GRAPHICS

One of the main goals of distributed-Oz is to make distributed programming easy, or put in another way, only slightly more difficult than centralized programming. Distributed programming is inherently more complex, there are more aspects to deal with. What we can hope to do, however, is to minimize the added complexity by preventing the tangling of aspects.

Our goal with DTk was to extend the notions of distributed-Oz to graphics, as far as possible. We want network transparency, and clean separation of the functionality aspect with respect to other aspects. This means that programmer should be able to program and test his application, for the most part, without considering the structure of distribution. Thereafter, having fulfilled the functionality requirements there should be means via small changes to the program control distribution.

The rest of this section is organized, as follows. First, the technical problems related to distribution of graphics are presented. Thereafter, the DTk-interface programming interface is presented. This interface provides for graphical distributed programming in distributed-Oz in a network transparent and network-aware way. Thereafter the principles used in designing and implementing the DTk-module are presented.

4.1 Technical problems.

Programming languages and tools for graphics generally model graphical display in an object-oriented manner. This is very natural as both objects and graphics are stateful. For example, a *rectangle* object is created. The rectangle object has a number of attributes, color, position, height, depth, etc. The attributes are subject to change, as the user moves the rectangle, changes its color, and so on. But there is some magic here, as changes to graphical attributes are immediately reflected on the display.

Consider *ordinary objects*, i.e. non-graphical ones, in a concurrent object-oriented language. When one thread causes the state of an ordinary object to change this is ‘not known’ by other threads until they access the state of the object. Or consider objects in distributed-Oz, if one user changes the state of an ordinary object in distributed-Oz other users that have references to the object are not aware of this change until if and when they access the state. This is as it should be. It would be completely impractical that state changes to objects were automatically broadcast to all other sites with references to that object.

Graphical objects are different from ordinary objects in that changes in the object-state should be reflected in visualization changes automatically. In a distributed multi-user application we often require the update to be broadcast to all the participants. In this case it would be impractical to wait with the update visualization until the participant accesses the object. Neither is it practical to demand that the participants regularly probe the various graphical objects; this would increase the amount of network traffic enormously.

We identify four difficulties with graphics in a distributed system.

1. **Push problem**: There is a fundamental difference between graphical objects and non-graphical objects. The underlying desired pattern of communication in a distributed setting where objects are shared is different. State update is pushed rather than pulled.
2. **Sequential-consistency**. The problem with achieving sequential consistency differs with graphical *push* objects as compared to non-graphical *pull* objects.
3. **Visualization control**. There are difficulties in controlling visualization, i.e. which users see what.
4. **Openness**. The ordinary mechanism for achieving openness may not be enough.

We focus in this section on visualization, the difficult final stage in a multi-user graphical application. Given that some site initiates a visualization change (i.e. changes the object-state of graphical object), we consider how this results in a visualization

[2] Providing other sites do not invoke the object in-between.

[3] There may be references to the variable in the network that later arrive at a site that has not previously held a reference to the variable. The system guarantees, which is necessary for network transparency, that the binding will also in due course become known to the new site.