Alexandre Valdetaro Porto

A non-intrusive solution for distributed visualization and collaboration in a visualizer

DISSERTAÇÃO DE MESTRADO

DEPARTAMENTO DE INFORMÁTICA
Programa de Pós–graduação em Informática

Rio de Janeiro
March, 2013

Pontifícia Universidade Católica
do Rio de Janeiro

Alexandre Valdetaro Porto

A non-intrusive solution for distributed visualization and collaboration in a visualizer

DISSERTAÇÃO DE MESTRADO

Dissertation presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática, PUC-Rio as partial fulfillment of the requirements for the degree of Mestre em Informática

Advisor: Prof. Alberto Barbosa Raposo

Rio de Janeiro
March, 2013

Alexandre Valdetaro Porto

A non-intrusive solution for distributed visualization and collaboration in a visualizer

Dissertation presented to the Programa de Pós-Graduação em Informática, of the Departamento de Informática do Centro Técnico Científico da PUC-Rio, as partial fulfillment of the requirements for the degree of Mestre.

Rio de Janeiro, March 26th, 2013

Alexandre Valdetaro Porto

BSc. in Computer Engineering at PUC-Rio - 2012. Since 2009 works at Tecgraf developing virtual reality and scientific visualization systems.

Bibliographic data

Valdetaro Porto, Alexandre

A non-intrusive solution for distributed visualization and collaboration in a visualizer / Alexandre Valdetaro Porto; advisor: Alberto Barbosa Raposo. — 2013.

v., 55 f.: il. (col.) ; 29,7 cm

1. Dissertação (Mestrado em Informática) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2013.

Inclui bibliografia.

1. Informática – Teses. 2. Visualização distribuída. 3. Colaboração. 4. Orientação a componentes. I. Raposo, Alberto Barbosa. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To my brothers of the DT.

Acknowledgments

I would like to express my deep gratitude to Professor Alberto Raposo, my research advisor for his patient guidance and useful critiques of this work. I would also like to thank Mr. Thiago Bastos for his valuable friendship, advices and ideas. My grateful thanks are also extended Mr. Pablo Elias, Mr. Vinicius Rodrigues, Mr. Rodrigo Braga, Mr. Luciano Monteiro and Mr. Thiago Maioli for their friendship and support.

I would also like to thank Petrobras for funding our host project Siviep.

Resumo

Valdetaro Porto, Alexandre; Raposo, Alberto Barbosa. **Uma solução não intrusiva para visualização distribuída e colaboração em um visualizador.**. Rio de Janeiro, 2013. 55p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Neste trabalho apresentamos o design e implementação de visualização distribuída e colaboração para um visualizador 3D imersivo. Começamos apresentando, em um alto nível de abstração, nosso design de um visualizador genérico. O design segue a abordagem MVC, isolando todos os objetos de negócios na camada de baixo da aplicação para torná-la modular e extensível, permitindo assim a mais fácil prototipagem de funcionalidades e isolamento de algoritmos complexos da lógica de negócios. Este design como solução surgiu da necessidade real de um visualizador de implementação monolítica, cuja manutenção e aprimoramento se encontravam com alta complexidade devido à mistura entre a lógica de aplicação e os diversos algoritmos de visualização e distribuição. Esperamos que nosso design possa ser reutilizado como inspiração para outros visualizadores que queiram reduzir a complexidade e o custo do desenvolvimento de novas funcionalidades de negócios. Sobre este design, então, apresentamos o design e implementação detalhados de um módulo que provê visualização distribuída e colaboração para o visualizador. Este módulo é não intrusivo porque não requer qualquer mudança na arquitetura da aplicação, e esta pode se tornar distribuída apenas pela inclusão do módulo. Este módulo serve como prova de conceito para o nosso design, por solucionar um problema clássico de distribuição e sincronismo em um visualizador de maneira transparente para a lógica de negócios. Ainda implementamos um visualizador exemplo com este design e nele conectamos o módulo proposto, onde verificamos ambos o sincronismo da visualização distribuída e a consistência da colaboração entre múltiplos nós, avaliamos também o impacto no desempenho causado pela visualização distribuída.

Palavras–chave

Visualização distribuída. Colaboração. Orientação a componentes.

Abstract

Valdetaro Porto, Alexandre; Raposo, Alberto Barbosa. **A non-intrusive solution for distributed visualization and collaboration in a visualizer**. Rio de Janeiro, 2013. 55p. MSc Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

In this work, we present the design and implementation of distributed visualization and collaboration for a, immersive 3D visualizer. We start by presenting, on a high abstraction level, our design of a generic visualizer. The design follows an MVC approach, isolating all the business objects in the lowest level of the application, making it modular and extensible, therefore providing an easier prototyping of functionality and the isolation of complex business logic algorithms. This design as a solution came from the real necessity of a visualizer with a monolithic implementation, whose maintainability and improvement are impaired due to a high complexity because of the coupling between the business logic and the diverse visualization and distribution algorithms. Our hope is that our design can be reused as an inspiration for other visualizers that wish to reduce the complexity and cost of the development of new business functionality. On top of this design, then, we present the detailed design and implementation of a module that provides distributed visualization and collaboration to the visualizer. This module is non intrusive because it requires no changes to the application architecture, and the application can become distributed just by the inclusion of the module. This module serves as a proof of concept for our design as it solves a classic problem of distribution and synchronism in a visualizer in a way that is transparent to the business logic. Also, we implemented an example visualizer with our design and our proposed module, where we verified both the synchronism of the distributed visualization and the consistency of the collaboration among multiple nodes, we also evaluated the performance impact caused by the distributed visualization.

Keywords

Distributed visualization.   Collaboration.   Component oriented.

Index

Figure List

Table List

It seems that I know that I know. What I would like to see is the 'I' that knows me when I know that I know that I know

Alan Watts, .

1
Introduction

Immersive applications seek to provide an experience to the user as close to real life as possible. Such experience must re-create sensorial stimuli and perception of space in order to induce the user's brain into believing that the immersive experience is real life. Although an immersive experience can be delivered by any kind of system, Real-Time 3D visualizers may stand out as the key player in the field (Figure 1). A visualizer is an application that enables the user to visually explore a virtual scene. The success of such applications is consequence of many features combined, such as, credible visual input to the user by usage of modern computer graphics techniques, enhanced illusion of visual and auditive depth due to stereoscopy, precise and immediate control thanks to real-time reading and processing of user input and a natural and intuitive manipulation of the environment by the usage of augmented reality techniques. Therefore, the development of an application that provides an immersive experience to the user can be very complex because of these many requirements and their implementation in a real-time reactive system.

Any real-time application must be reactive, i.e., it must process input from external devices with a delay small enough not to break user immersion. Moreover, the interactive visualization requires that at least a given amount of frames be rendered every second. Thus, efficiency is the utmost requirement when developing a feature in a real time 3D visualizer.

In order to achieve maximum efficiency in a visualizer's many routines, e.g., rendering, input processing, distribution, data loading and so on, there is a natural tendency of the developers to trade abstractions for low-level APIs in order to have access to every available optimization setting. However, the exposure of every low level API to the business logic developer can greatly increase the complexity, which may lead to an increase in the lifecycle cost [10] and effort [23] required to maintain and extend a visualizer with new functionalities.

To the best of our knowledge, the majority of the 3D visualizers, where efficiency is paramount, are implemented in a monolithic way. There is no clear separation between business, distribution, rendering, architectural elements

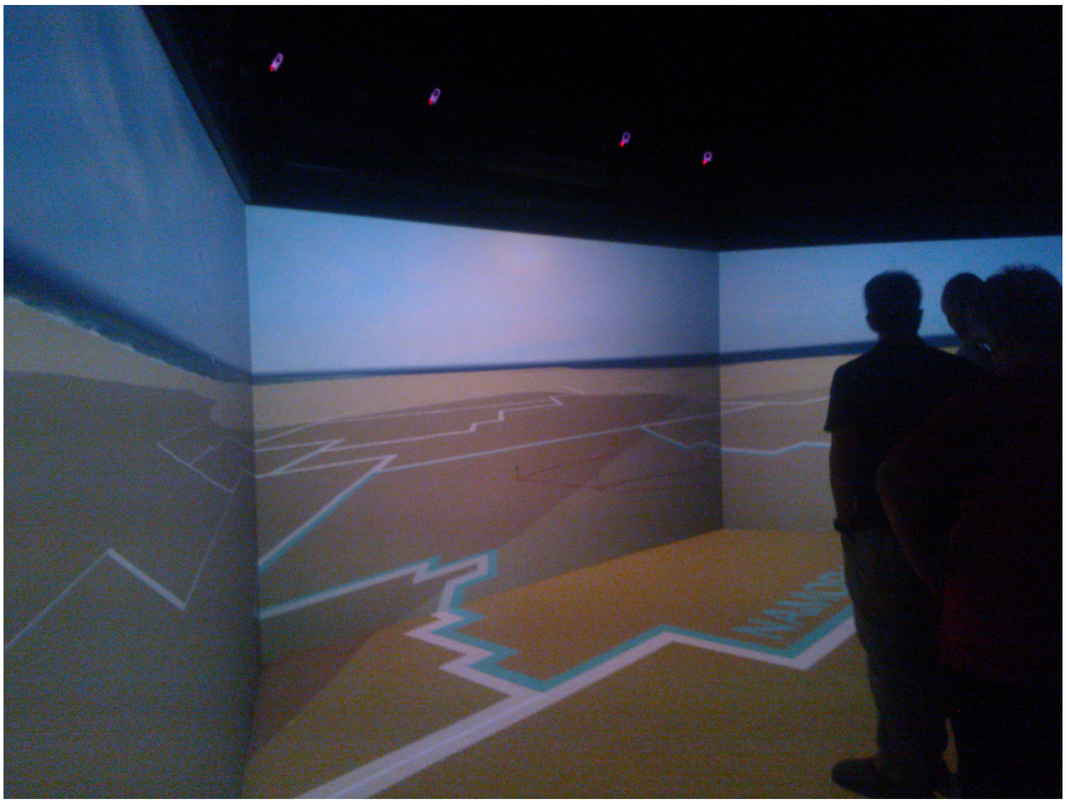

Figure 1.1: Immersive distributed visualization of an oil field

and so on. This approach can have some advantages because the development of every functionality has access to every low level system, therefore the developer can highly optimize the rendering, distribution and other consequences of the developed feature. However, from our experience (section 2.2) the constant increase in complexity for the development of simple business/application features, and the tight coupling between the logic layers can seriously impair the maintainability and extensibility of the application. Which can lead to a stall in productivity, where developers suffer with the high complexity they need to understand in order to develop a simple feature and also the risk of breaking some other feature.

The application programmer, which develops the features that aggregate real knowledge value in a software, should be focused on the domain of the application, the business logic, transactions, user interface and any other front end concern. Therefore the application programmer should work with as much high level abstractions as possible, such as frameworks, tools and *middlewares* [35]. However, one must always consider the trade-offs when abstracting low level systems, as the usage of low level optimizations may be necessary in many cases as explained above.

In the first part of this work, we propose a design for an immersive real-time 3D visualizer. With this design, we aim to reduce the complexity and increase the productivity during the development, maintenance and improvement/upgrading of the visualizer. Our main requirements for this design are modularity and extensibility, which can be very closely related and the former is a necessity for achieving the latter. Modularization of an application by itself can be of no use, there is no direct gain in separating pieces in modules. However, if we are able to separate cohesive parts of the application into modules and expose clean interfaces we may benefit with abstractions and greater flexibility. Abstractions come naturally as the modules begin to represent something more concrete and the flexibility comes from easily switching one module for another or attaching a new module to an already running system. This flexibility also implicates in testability and extensibility as we can easily switch real modules for test modules and prototype new functionality without changing the already running application.

The necessity for these design guidelines came from our witnessing of a high complexity of developing and testing new features in our host project which was also designed in a monolithic way as we explain in section 2.2. In order to achieve all of the benefits of modularity and extensibility, we first need to cohesively separate the visualizer in modules, therefore we followed the MVC concept. Our approach is to isolate the business objects in the *"business layer"*—the Model—, which is the bottom layer of the visualizer and is accessed by every other layer above it. Since the business data is the only real data that the user is concerned about, every other layer above is a mere representation of the data—the Views. Hence all the graphical representation and user interactions elements must rely *only* on the data stored in the business layer. Moreover there is ideally no side communication between these representative elements, since it can break the consistency between the representation and the real data. Such separation of data and view provides us extensibility for the application, considering that every additional element that we deploy must only guarantee a consistent representation of the data it is interested in. Also, the possibility of switching parts of the system easily. E.g., the scene graph implementation can be switched between a very efficient and licensed per station library for displaying a scene in massive immersive environments and a cheaper licensed library for common desktop usage.

In the second part of this work, we then propose the design and implementation of a module that transparently provides distributed visualization and collaboration for the designed visualizer. The module works by applying and observing changes to the model in our MVC design. It is non intrusive as it requires no modification in the other modules and therefore keeps the application developers indifferent of the distribution which can greatly reduce the complexity and increase productivity. The same configuration of modules can switch from local to distributed just by the addition of this module. We hope this module serves as a proof of concept for the modularity and extensibility of our design as it solves a classic problem of distribution and synchronism in a visualizer in a way that is transparent to the business logic.

We also implemented an example visualizer into which we designed and tested the distribution module. This example system became our new implementation of our host project as explained in section 2.2 . An immersive visualizer when visualizing distributively must have frame synchronism between multiple distributed views of the same scene with an acceptable loss if compared to a local rendering. Frame synchronism is the guarantee that every node displays the same frame at every rendering cycle—an thus have the same number of frames per second—and this synchronism is guaranteed by our algorithm, so our tests goal is to measure the performance loss of the distribution. As for the collaboration, which is not critical for the immersive visualizer, our test are only practical tests for our arbiter topology scheme consistency as any more complex real time manipulation with advanced collision treatment escapes the scope of this work.

This dissertation is organized as follows: In Chapter 2 we delve into the functionalities and the problems that arise when developing an immersive visualizer, as well as our background and motivation for this work. In Chapter 3 we walk through the available solutions for our distributed visualization and collaboration requirements for an immersive visualizer. In Chapter 4 we explain our design for a visualizer oriented towards modularity and extensibility. In Chapter 5 we explain our design for the module that provides distributed visualization and collaboration for the visualizer. In Chapter 6 we explain the whole implementation of the module designed in Chapter 5 and provide some results of our example system implementation. In the last chapter we present some conclusions and show possible future work.

2
Immersive Visualizers

2.1
Rendering in Immersive Applications

Rendering, in computer graphics, is the process of generating an image from a virtual scene. The generated image can then be used for multiple purposes. The most obvious purpose is to display the image in a screen to the user. However, the image can be the input to another render process, to some data analysis, to a file and to a video, among other uses.

The process of rendering is usually parallel, where there is a different execution line for each part of the output image. A simple scenario, is when a single rendering device—usually a graphics card—is rendering a full view of one scene, that is, there is only one observer of the scene, only one view of the scene and only one output image. In this scenario, a scene is processed in parallel inside the graphics card *GPU*, every pixel of the output image is delegated to a different execution thread, the final image is then output by the graphics card. In this basic scenario, there is only one layer of parallelism and it is contained inside the graphics card GPU, transparent to anyone that does not deal with *shaders* and *GPU* programming. However, in immersive applications, sometimes there is more than one view of the scene, more than one observer, more than one output image or more than one rendering unit. These usually fall into the distributed systems realm, where there are more layers of parallelism on top of the one just mentioned and there are several approaches for rendering and displaying, which are covered below.

2.1.1
Distributed Visualization

Distributed visualization is the process of displaying a single virtual scene from multiple views. Although these multiple views are usually displayed on different visualization output units, there are cases in which they are displayed in a single one, such as in stereoscopic rendering. The users observe simultaneously all of these views that are windows to the virtual world. These multiple views can be a result of a single or many observers. When there are multiple observers, each one of them is exploring the scene differently, and there is no real synchronization issues between the views considering that each view will be output to a different user of the system. When there is only one observer however, the many views are going to be output to the same user–or group of users—hence the requirement for correct synchronization. In this work, we aim to solve this synchronization for distributed visualization of a single observer with multiple views of the virtual scene. Thus whenever we mention distributed visualization from now on we are referencing such case.

When a user views a scene through multiple screens—with different views—he enjoys a greater level of immersion than a user viewing through a single screen. However, displaying a scene in different screens creates a few additional requirements. First, the user is represented by a virtual observer, which looks at one direction and is positioned somewhere inside the scene. Therefore, based on this observer and the position of the screens in the real world, the application must calculate the appropriate different views for every screen, or the immersion would be broken. This calculation, despite not being trivial, requires only the information about the displaying devices positions and orientation in the real world.

The real problem of displaying to multiple screens is the technical problem of how to render and synchronize so many views and output to different screens. A traditional approach to achieve distributed visualization is through the usage of out-of-the box systems that make use of dedicated hardware. The hardware controls all the visualization devices. Such solution presents, usually, a high cost of deployment and maintenance.

Another approach, is to use a single computer and output to multiple screens. Most of the mainstream high-end graphics cards can output to 2 different screens, and some special product lines like nVidia's Quadro Plex can output to up to 4 screens. Thus, a single machine could have 2 Quadro Plex and that way having a total of 8 outputs, which could be acceptable for many applications. Although the aforementioned setup seems to be a solution, in reality there are two serious problems that arise from that architecture. First, the price of a non mainstream graphical card can be very high. One could argue that in a multiple screen scenario, the price of the equipments are already expensive, and a specialized graphics card is acceptable. Such point of view is reasonable and true in many cases. Therefore, the real problem with a single machine with multiple screens lies in the rendering bottleneck.

The rendering speed depends roughly on the amount of geometry—the number of vertices of every object in the viewable scene—being processed and the size of the output image. Therefore rendering multiple views of the same scene gets very expensive in terms of processing, specially if the output images have to be very big for display walls. Consequently, distributing the scene to multiple nodes can be a feasible solution—despite the distribution difficulties. There are many possible designs of a visualizer that provide scene distribution as well as many tools available for it. In the related work section we make a review of the main available tools and their usefulness for our scenario.

2.1.2 Collaboration

Collaborative software is a software that enables multiple users to collaborate to achieve a common goal [29]. A collaborative visualizer lets the users explore a common scene, and possibly make changes and view other users avatars in the scene. The most common case of a collaborative visualizer is a multi-player game, where the players interact with each other in a shared virtual world. The players can talk to each other, engage non player characters and change the persistent world in many ways.

There is a considerable technical difference of achieving collaboration in the domain of games and that of scientific visualization. A game must be protected against cheating and invalid input from users, must have a low latency, must be fast to resolve inconsistencies when more than one player is changing the same part of the world and sometimes cope with a massive amount of changes at the same time. All of these problems from the gaming domain require very complex techniques to be solved. However, usually in a scientific visualizer, there is no need for checking against possible malicious input from the users, there is no need for a very low latency, there is no massive amount of users at the same time and also there is never more than one user changing the same part of the world simultaneously, as the user must first take control then change anything. Hence collaboration in the scientific visualization domain is much simpler than in games. The module that we propose in this work is aimed at scientific purposes, and provides collaboration in a simple topology, where there is a server that is responsible for persisting the world and clients communicate directly with it by two-way updates.

2.2 Background and Motivation

This work is developed as a part of SiVIEP (Integrated Visualization System of Exploration and Production) project. SiVIEP is an immersive scientific visualizer, which supports visualization of reservoirs, geological surfaces, wells, risers, platforms and many other objects from the oil field domain. All of these objects can be visualized together inside one project in a 3D multi-screen stereoscopic setup, controlled by several manipulators and tracking devices. Many supported objects pose already a challenge to be rendered in a single screen due to the number of polygons, simulation data and property visualization. Therefore, complex distribution algorithms must be employed for the system to be able to render all the required objects in a multi-screen stereoscopic setup.

SiVIEP's current production version is a monolithic *C++* application using Qt for user interface and OpenSG [5] for distributed rendering. There are some bottlenecks with the distribution provided by OpenSG [5] for our scenario, and these are discussed in the related work section. Also, the complexity of adding new business objects to the current *C++* version is too high due to the tight coupling between many non cohesive parts of the application.

We started this work to provide a solution for these SiVIEP issues with a new design that could isolate the rendering and distribution complexity from the business logic. However, we would have to provide a solution for distributed visualization in this new design of SiVIEP since it will no longer use OpenSG due to the known problems in the library's API. Moreover, collaboration has always been a milestone in the project's backlog and there has been no feasible solution for it in the previous architecture. Also, some of our clients are also programmers, and they wish develop their own business objects and tools as separate plugins without the necessity of our intervention.

In order to provide deployment of modules in a service oriented paradigm, the new SiVIEP shall sit on top of a lightweight C++ architectural middleware based on code generation. This middleware named *Coral* [1] follows a component oriented paradigm and provides some very important features for our system, which are:

- Reflection/Introspection;
- Descriptive language for specification of components, interfaces and other types;
- Toolkit for code generation and component creation;
- Module based deployment and extension;
- Full bridge for Lua scripting language [17];

Middleware has historically targeted enterprise systems, which typically involve many disparate software systems distributed across multiple company divisions [36]. Hence the mainstream middleware works at a very high level of abstraction, prioritizing flexibility over efficiency, and is of limited utility for systems with strict efficiency requisites such as real-time systems, games, natural user interface applications, and sometimes even cloud applications [26]. However, for a middleware to be useful in designing an efficient application at a fine level of granularity such as representing every entity and interface element in a 3D game—its abstractions should pose little to no overhead over the native language constructs.

Coral is a multi-paradigm C++ architectural middleware which emerged from a necessity in SiVIEP due to the lack of general C++ architectural middlewares for the niche where the language thrives most—efficiency—which had been SiVIEP's case. In our experience, most C++ applications miss out on powerful architectural abstractions due to the lack of an efficient, ready-to-use middleware.

Coral's hypothesis is that, with a C++ architectural middleware carefully designed for efficiency, even demanding applications can benefit from a diverse range of techniques such as service orientation, dynamic module deployment and component composition, model-driven engineering, scripting and rapid prototyping—among others—thus helping control complexity and reducing costs. Coral is an ongoing open-source project to develop such multi-paradigm architectural middleware, and results so far are encouraging.

With the usage of Coral, the new SiVIEP is being built as a modular application. The modules vary from time critical such as the scene graph modules (currently OSG [4] and VL [6]), which are very low level, efficient and written in C++, to the UI module, which is high level, easily readable and written in Lua.

3
Related Work

A number of generic solutions for distributed visualization have been developed, and are available commercially or open source, e.g., Equalizer [12], Chromium [16], [5] and VRJuggler [9]. Some of those are completely transparent, some require a certain level of adaptation from the programmers and some force its own programming model. In the collaboration field, most of the state of art solution are far more complex and generalist than desired for this work. However, some of these works contributed to model our solution and are going to be detailed here.

OpenSG [5] is a scene graph as much as any non-distributed scene graph. A user can develop using OpenSG without distributed visualization in mind despite being one of the scene graph's major features. The distribution strategy of OpenSG works by distributing the scene graph with all the graphical nodes containing vertices, textures, matrices and other graphical primitives. Therefore all the graphical nodes must be serializable. Making a serializable graphical-node is simple if it is a regular triangle mesh that requires no processing during runtime because the node will be serialized once during pre-processing only. However, many nodes have constantly varying set of primitives such as: large on-demand loaded files such as terrains, photo-realistic detailed meshes with continuous level-of-detail techniques, simulation data visualization and so on. These variable nodes are very expensive to distribute, as their data is constantly varying and can be of very large size. Taking our oil field visualizer scenario as an example, most of our domain is composed of simulation data, that requires different visualization modes with generated graphical data on demand. Therefore, this graphical distribution approach is not suitable for us.

Chromium [16] works by creating a powerful abstraction of the OpenGL API, that is, it intercepts every call to the API and executes a scalable rendering algorithm to distribute the call among multiple nodes. Such approach is very clean and non-intrusive, but requires the constant distribution of graphical data, which can be a bottleneck if the scene is very diverse so that the GPU memory will need to be updated frequently. Also, this approach does not consider collaboration. In order to provide collaboration with Chromium, a whole separate solution must be implemented.

Equalizer [12], Chromium[16] and VRJuggler [9] are tailored for parallel and scalable rendering, that is, using multiple nodes for rendering the same scene independently of the number of screens. They all support outputting the rendered scene to multiple screens making distribute visualization feasible. One of these tools stand out as the most complete and non-invasive, the Equalizer [12]. It is a very powerful and complete tool that forces the programmer to abstract the rendering code from the rest of the application. Thus, it distributes this rendering client for the slave nodes. Such nodes perform rendering tasks controlled by a master node, which is configured by configuration files describing the available resources in the cluster as well as the desired compositing strategies. The application is unchanged for any kind of scalable setup. The rendering in the slave nodes can be configured for many different compositing strategies [28], such as sort-first or tile based, DB based or sort-last, among others [22]. After one of these strategies is used, the output image is copied to the configured output walls/screens. Such strategy of scalable rendering is very powerful, but unnecessary for a typical case of multi-screen rendering setup (our whole range of cluster examples fit the common case) where every screen is driven by an individual node of the cluster. Also, if the application is going to be used in single computer workstations to visualize the same projects, then using the scalable rendering system to achieve better system results can be pointless, as the scene needs to be processed by a single machine as well.

All of the aforementioned solutions provide the distribution on the graphical layer of the application, that is, they distribute graphical code, graphic controlling commands, graphical primitives and so on. However, for a collaborative visualization to be achieved, a distribution of the business domain (not just its graphical representation) is necessary. Therefore, in the mentioned approaches, the collaborative visualization would have to be solved by other means. In this work's solution, we provide a DSO (distributed shared objects) system for our domain data. That is, our domain data is a single shared graph among the nodes. This way, our solution provide collaboration through two-way updates of local changes for every node. Moreover, this same solution will serve as the ground for our multi-screen rendering algorithm to work with, as all the graphical representation can be loaded locally based on the received domain information.

Equalizer [12] and VRJuggler [9] provide some mechanisms for the developer to distribute generic data, but those do not solve our collaboration requirements because their distribution is push-oriented (one way) with the concept of master and slave objects and provides just limited support for a slave to update a master object (Equalizer). Nevertheless, even though their DSO approach had been more generic allowing two-way communication, such as other pure DSO approaches [18] [20] they would still be inferior to our approach due to versioning. Every DSO system needs internal versioning of objects, and they need to provide a way for the user to develop objects that are distributable, be it inheritance, description, reflection and so on. Our scenario requires distribution of objects that are instances of components, and even though it is possible to implement a bridge to a specific component system in the serialization methods provided by the DSO system, our system would have to conform to the versioning system of the library. This work's solution intends to preserve the application own versioning system.

Our approach differs from the ones above in a way that it distributes the application domain in the application layer instead of the domain graphical representation in the graphical layer. That way, we solve a collaboration and multi-screen rendering requirements at once. Although we might have a less scalable rendering system than the scalable rendering approaches because it does not support any decomposing/recomposing strategies as well as no load-balancing, which is not a problem in our case as we explained above. Every node will have the domain, they edit and share it, so any collaboration entities in the domain such as selections, avatars and so on will be distributed. From the domain, the node will create the scene and render it. If there is a multi-screen setup, then a multi-screen render controlling API should coordinate rendering based on calculated frustum and camera position by a render master. The output rendered image will simply fill its default graphical output as there is no recompositing to be done. This way, all of the component-based application domain can be extended incrementally as our whole design expects (more domain specific objects) with a simple module that distributes it and enables multi-screen immersive rendering.

For our implementation of the framework proposed in this work, there needs to be a low level, inter-machine, communicating tool. The implementation of such tool is also part of this work and all the research done for it is detailed below.

One paradigm of communication is a message oriented one, such as MPI, where a communication channel is used to write messages that shall be received by another remote(s) communication channel(s). This approach is asynchronous, not transparent and exposes the unreliability of the network for the user and although it makes the handling of network failures easier, it completely breaks encapsulation and most of current programming models (object orientation, component orientation, services among others). Implementing a distributed multi-screen rendering system on top of a raw messaging system can became very complex as there should be a separation of concerns: functionality, distribution and fault tolerance [32].

The other paradigm of communication is Remoting or RPC—Remote Procedure Call—, such as CORBA [24], RMI [2] and .NET remoting [3], where the encapsulation models are preserved, and every bit of communication among network nodes is made through the same means of local communication among entities, be it functions, methods or services. As the RPC approach abstracts the network, it can be much more complicated than messaging when it comes to dealing with network failures. Thus, in a big, unreliable and heterogeneous networking environment, in order to maintain network transparency, a messaging system associated with appropriate distributed algorithms can be a better approach [31]. The RPC paradigm can be a better option for reducing the complexity of implementation if the underlying network is reliable. Our solution is tailored for reliable networks, so we believe an RPC approach is the most appropriate solution.

In the next chapter we describe the design of a visualizer following the MVC approach, in which our distribution will be integrated.

4
Design of a Modular Visualizer

In this chapter we present an architecture for a modular and extensible visualizer, we explain the rationale behind every separation of concerns and layers and why this design works for our scenario. As explained in the previous chapters, although our main goal is to present a solution for distributed visualization and collaboration, our scope goes beyond the features, reaching the architecture of the visualizer as well. It becomes necessary to explain all the visualizer's architecture because it needs to follow a specific paradigm of modeling, which is necessary for extensibility and modularization.

4.1
Definition of the Visualizer's Building Blocks

In this section we define the main concepts and abstractions we use in our design. We analyze the main building blocks and their purposes conceptually.

Domain and Entities The domain of an application represents *business knowledge*—as the *model*—in a MVC paradigm. What we have referenced in the previous sections as *business data* or the *business objects layer*, is technically the *application domain*. It is a mirror of the real world, composed of all the real world objects that the application wants to represent. Ideally, there should be a one-to-one correspondence between the domain objects and their real world counterparts. These domain objects are represented by *Entities*. Examples of entities in an oil field visualizer application are risers, wells, reservoirs and so on.

Scenes We define a *scene* as a visualizable collection of entities with behaviors and tags. This definition however, may be considered incorrect because of our definition of domain, as the scene clearly has different properties and elements than a real world object. Moreover a more purist definition could consider the domain itself a scene, which is the case in many visualizers. Nevertheless we regard the scene as business knowledge because they represent snapshots of the same domain in different times, configurations, simulation scenarios and so on. Figure 4.1 shows the

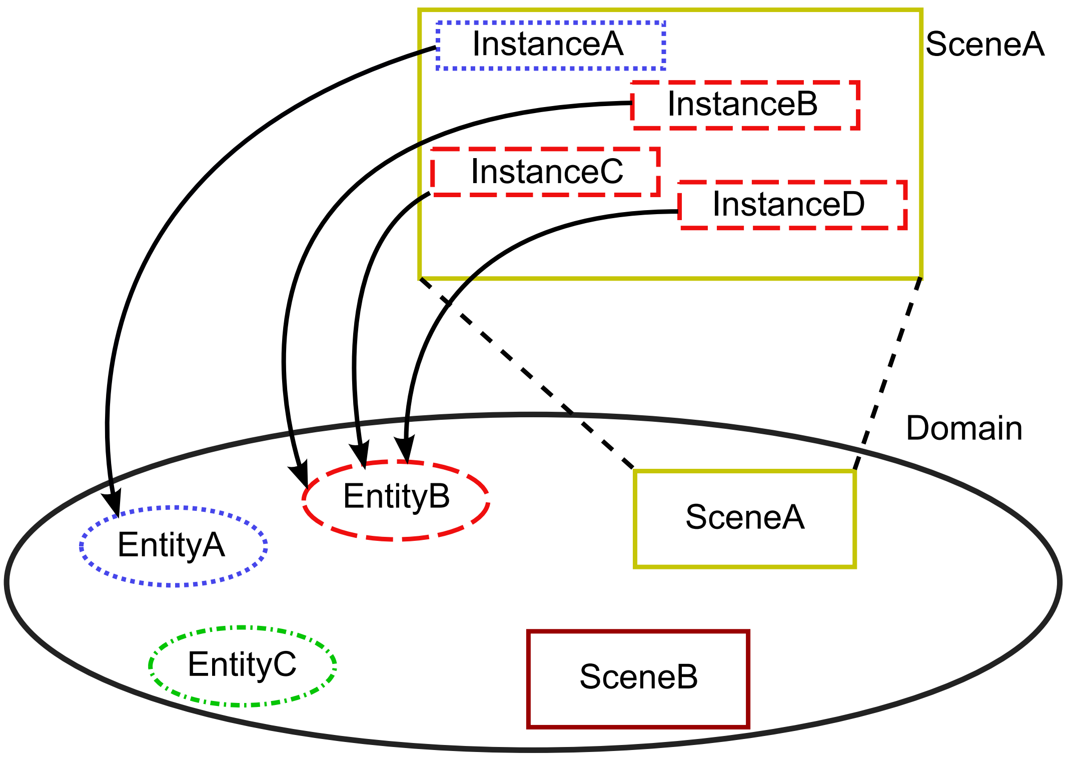

Figure 4.1: Chart showing the relation between the domain, the scenes and the entities

structuring of our domain regarding scenes. Examples of scenes in an oil field visualizer are: the oil field when exploration began, the oil field in present date, the oil field in a catastrophe simulation scenario, etc .

Every entity must have a behavior associated in the scene. There may be scenes where all the entities are static, thus behavior can be regarded as only a position where the entity is located. In some other scenes, there may be entities with attributes that vary with the time and may require a more complex behavior structure. Some changes in the behavior are non deterministic, like user input. Non deterministic changes cannot be described, and must be applied as a direct attribute change.

Roehl [30] describes the deterministic behavior of entities in different levels according to the way the attributes are modified with time. The first level, 0, is a mere position coordinate in a $\{x, y, z\}$ format. And each level above adds a new variable that describes the variation of the level below it in time, i.e. level 1 describes the variation of level 0 (the position) on time.

In this work, we are only dealing with level 0 behavior, and therefore scenes that have only static entities. The user can transform the position of the entity, but no continuous time dependent behavior can be associated.

Representations These are the Views following the MVC concept. The visualizer objective is to present the domain in many different ways to the user. There may be canvases displaying a scene from the domain and also tables and other UI pieces displaying relevant information. The user should be able to navigate through the scene, thus changing the camera position, interact with the entities, choose the relevant data to be displayed and switch between scenes, or the user may even want to switch to a different domain. These actions are a coarse example of the interaction between the user and the visualizer, but they provide a good measure of how the domain is changed and how the rest of the application should react to these changes.

4.2 Structuring of the Visualizer's Building Blocks

In this section we delve into the details of the interactions and underlying organization/structuring of the building blocks.

4.2.1 Underlying Structures

Domain - Entity Graph We defined the domain as a collection of entities and scenes. The entities are usually organized structurally as similar as possible to the real world, considering that the structure itself represents how the entities are related. Some environments may have a collection of independent entities that have no relationship among themselves, whereas some environments may have entities that are closely related and may be organized in an associative or hierarchical structure. Moreover, among with the entities, the domain contains scenes and maybe relevant user configuration data—which is not by definition part of the domain but can be included regardless. We designed the domain structure as a graph, which provides the needed structural flexibility. Furthermore the graph can be versioned, shared and persisted if needed.

Entity - Attribute Collection The entities are collections of attributes with no relational structure associated. Some are purely for the entities appearance, i.e., models, colors, materials, etc. However, there is also the possibility of associating relevant business data, such as properties, simulation data, and so on. Regardless, all of this business data has to be visualized in some way (Figure 4.2) even if their description does not directly imply in something visual.

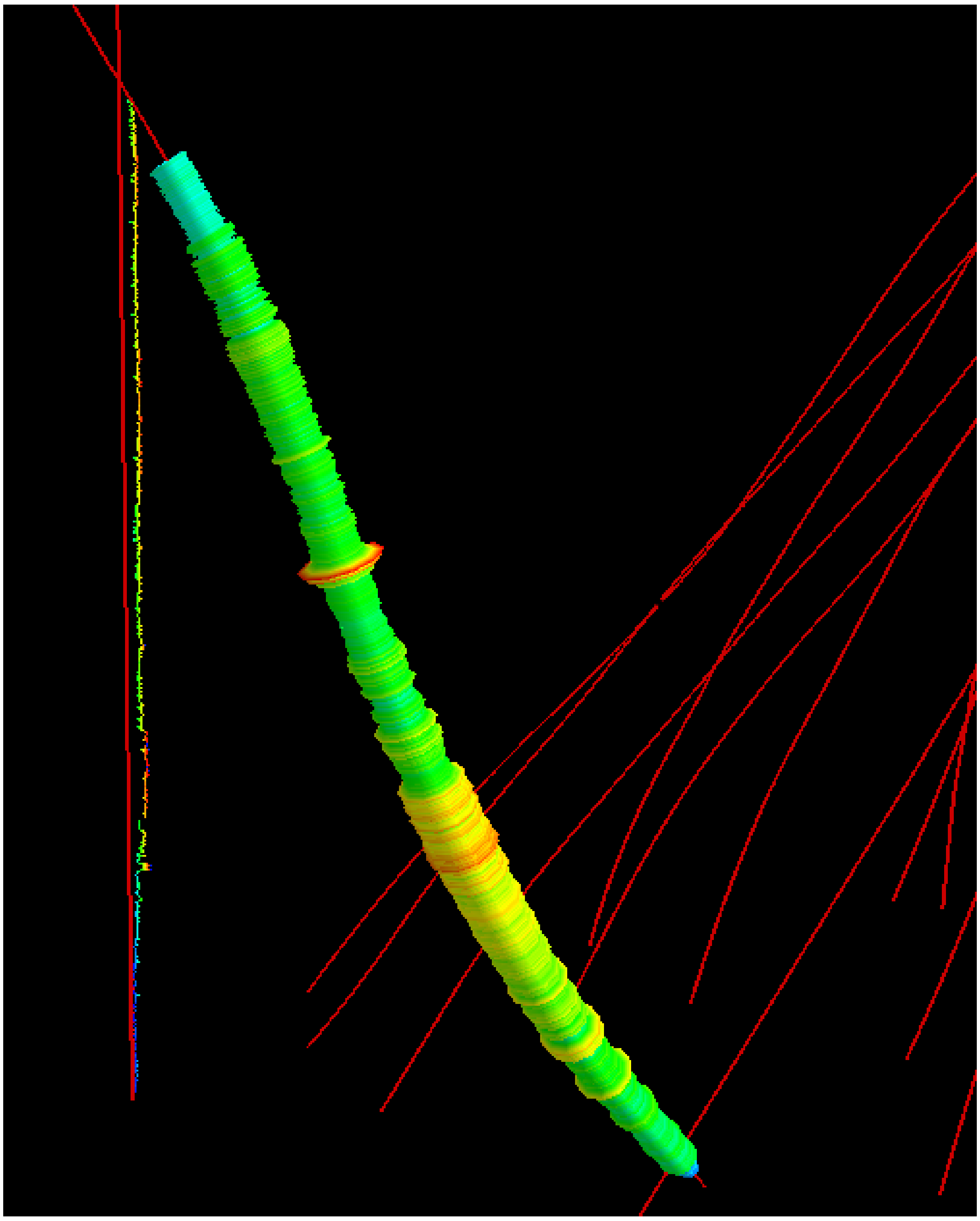

Figure 4.2: Business data visualization. A discrete property along a riser is mapped to colors and geometry.

Domain Manipulator and Versioner This is the Controller in the MVC concept. It is a module called Domain Model Framework that connects the Domain to the many representations of it to the user. It is explained in the following section when we cover the interactions among the elements.

Scene - Behaviours and References The scene is an arrangement of entities by their behaviors. This arrangement can unmistakably contain chunks of the entity graph of the domain, and in some cases, be an exact copy of it. Nevertheless we designed the scene as a graph of references to entities with behaviors attached, which are structures that can be processed directly by the rendering system.

User Interface The user interface or *UI* is the topmost layer in our system, ideally it should be able to interact with most of the application. It is comprised of a collection of interface elements, i.e., buttons, boxes, labels, fields and so on. All these elements when interacting with the user should alter the other layers below. Although we defined that there should be no sideways manipulation among layers above the domain, the UI may sometimes need to control certain functionalities that are not in the domain layer. This happens when the user wants to change how the scene is visualized, such as which property for an entity, which scene from the domain and so on. Nevertheless there should *never* be any manipulation of graphical objects by the UI.

Other Modules The other modules should provide views of the domain, domain editing functionalities, persistence and so on. These modules should only work on top of domain data and thus have no business data on them, their internal structring is irrelevant to the rest of the application as there should be no access to them from any other module except for the User Interface. Figure shows the layering overview of the design.

Figure 4.3 shows the layering of the system according to the MVC concept and an overview of the interfaces and interactions among the many modules.

4.2.2 Interactions

As mentioned above, the domain should be an object graph. However, this graph is not useful for the rest of the application only by itself. Its state needs to be propagated through the layers above it, for that a system is

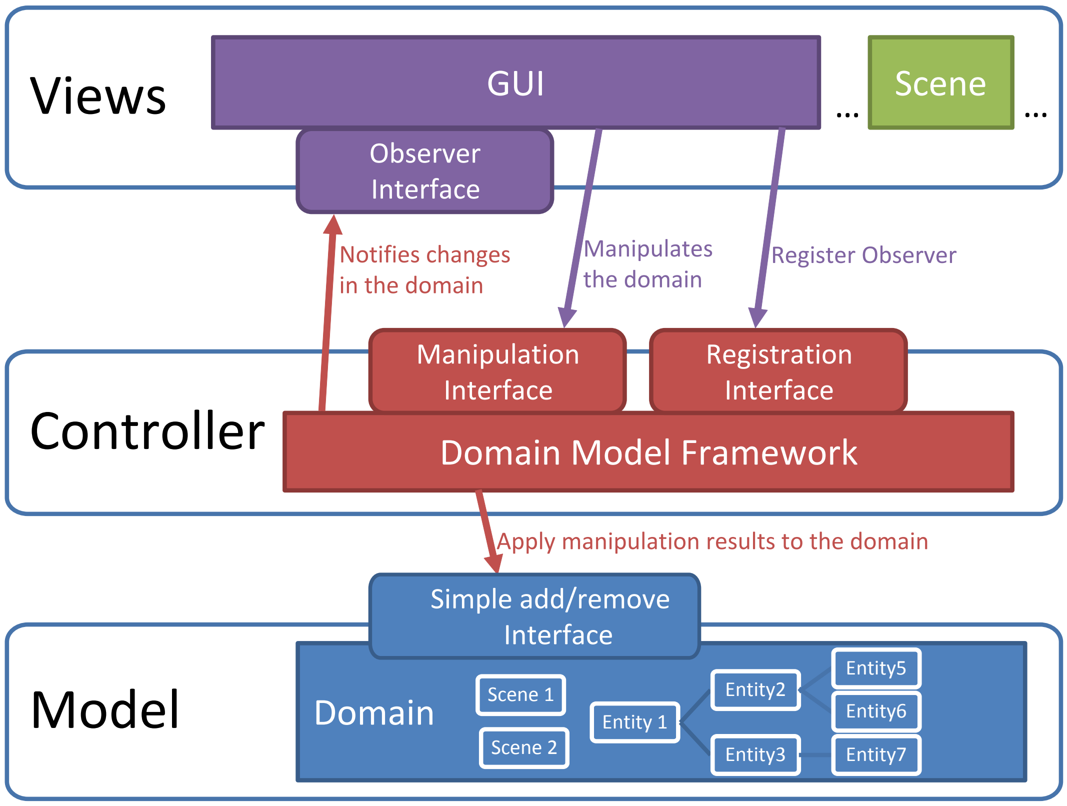

Figure 4.3: Blueprint of the the visualizer architecture

necessary. Therefore it is necessary to have a graph controlling system that provides controlled access to the domain graph and notifications to everyone that depends on it. We designed this controlling system as a non-intrusive module that keeps track of all the changes in the object graph and provides a fine grained notification system for the rest of the application. We call this system the DMF—Domain Model Framework—(Figure 4.4). Every component that applies some modification in the business object graph needs to certify that these changes will be propagated to everyone else (Figure **??**). Hence whenever a change has to be applied to the graph, this change has to go through the DMF. Consequently, every component that needs to display information about a business object needs to be connected to the notification system of the DMF.

The DMF notification system follows the *Observer* pattern [14]. Every element that needs to watch the state of a piece of the domain will be registered as an observer to that piece in the DMF notification system. Accordingly, the DMF will notify that observing element whenever a change is applied to the observed piece of the domain. The notification itself consists of the previous and the current states of the piece. The DMF also provides a global graph access system for the elements that need to change the graph. Notice that in our design, the actual objects from the business object graph does not reside inside

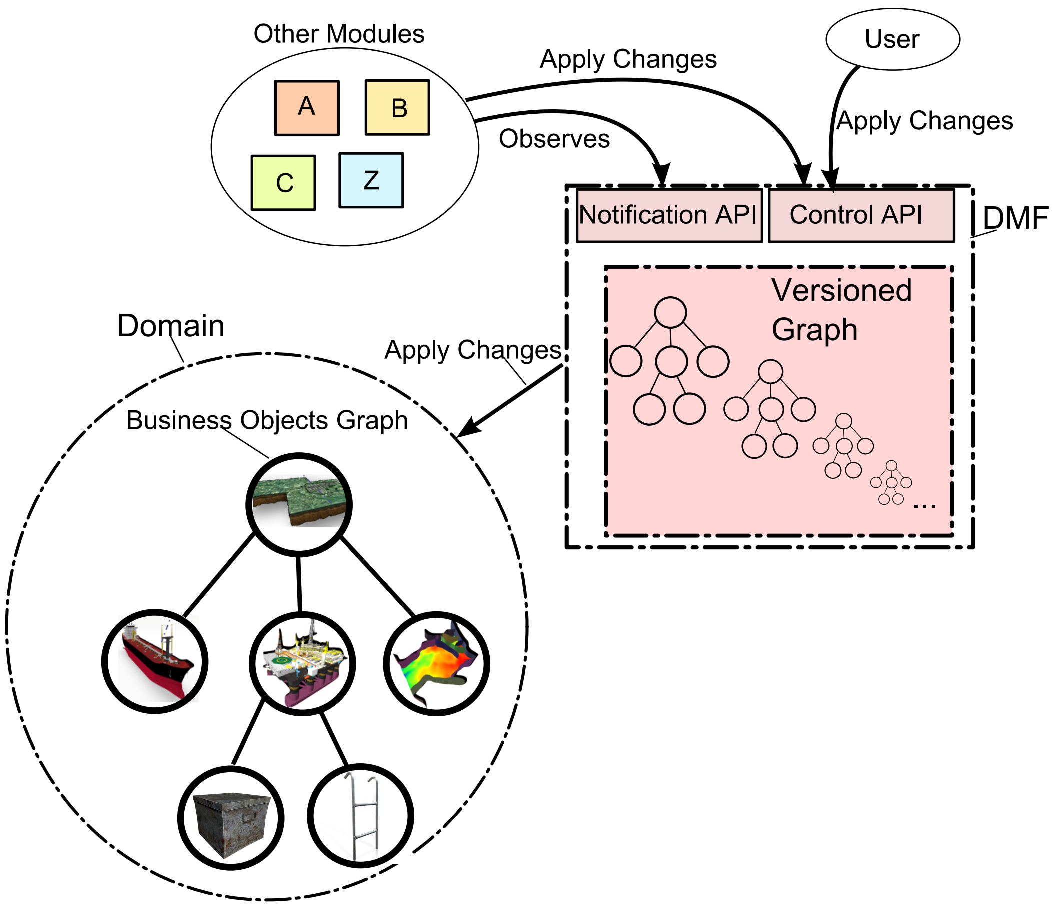

Figure 4.4: DMF architecture overview

the DMF data structures. As we mentioned before, the domain management is non-intrusive. Therefore, the DMF has an internal graph containing meta information about the actual objects and it keeps versioning information, that is necessary for undo/redo operations and is also very important for our distribution module that will be explained in the next chapter.

This non-intrusive approach makes the DMF generic and increases its usefulness, as we can use it for any graph of objects that must be versioned and shared. However, as the actual objects are not inside the DMF, there is the risk of a business object being modified from outside of the DMF interface. If this happens, then the will be an inconsistent state. Therefore, there must be strict guidelines for the developers on the usage of such kind of framework.

The internal implementation of the DMF and its versioning algorithms are not in the scope of this work.

The scene display module is obviously responsible for displaying the graphical representation of the current virtual scene. However, depending on the underlying graphical API workflow, this module's design can significantly vary. Graphical APIs can follow 2 different workflows:

Immediate Mode The graphical API is a mere set of functions to load and draw graphical objects to a buffer. Furthermore the user holds all the graphical objects and controls the rendering calls. The OpenGL, XNA and DirectX are examples of immediate mode Graphical APIs.

Retained Mode The graphical API has an internal data structure that holds all the graphical objects. The user may only add objects to this structure and leave the whole drawing workflow to be handled by the API itself. Moreover the user may register callbacks for manipulating the graphical structures after the control is handled to the API.

Since our implemented example system uses OSG [4] and VL [6]—both use retained mode—as graphical APIs, we explain the design of this scene display module using a retained mode graphical API. The module is a simple observer of the current scene and its referred entities, which are all contained by the domain graph. Whenever a new entity reference is added to the current scene, the module searches for a graphical representation for the object and add it to the underlying graphical API. Never should the actor be modified directly, it must always mirror the business object through observation. This way, it becomes easy to switch the graphical implementation of the objects. We may change the whole scene graph implementation without touching the rest of the application.

The UI layer sits above the rest of the application, no module should access the UI. The UI must make sure also that it does not change its own state, which is very common if no special care is taken. E.g., if a button that when pressed automatically changes its image and the button represents a state of a business object, when the state of the object is changed by any other means—be it collaboration, animation, task scheduling and so on—the button will be left in an inconsistent state. Therefore, the developers must make sure that these automatic feedbacks that usual UI frameworks have by default are not enabled.

A simple example of this system design in practice: A user clicks in the UI and wants to add a new object to the domain. The UI layer creates the object and access the global DMF interface to make sure that the new objects creation propagates. The DMF propagates the changes to every component that observes the graph. The Scene module is notified of the new object, creates an Actor for it and adds to the scene. The UI layer gets notified and also adds a new entry to a tree-view widget that describes the scene.

In this chapter, we have explained our modular and extensible design for the visualizer. In the next chapter, based on this design, we move on to describe the module that will provide distributed visualization and collaboration when attached to the visualizer.

5
Distributed Visualization and Collaboration Module

In this chapter we describe the design of the module that provides distributed visualization and collaboration for our visualizer. We call this module Distributed Visualization and Collaboration Module—DVCM— from now on.

5.1
Domain Distribution

We intend to seamlessly connect this module to our visualizer, hence transforming its visualization and workflow with only small changes in configuration files. Nevertheless switching from local to distributed may create a high risk for inconsistency if the application is not properly designed. This inconsistency can be noticed when the application state is spread across multiple elements and layers, hence making complicated for one instance of the application to propagate its state consistently to another. However, in our designed visualizer, we concentrate all the application state in a single element—the *domain*—thus alleviating our effort designing distribution. Still, in order to provide seamless integration with the application, this module must distribute the application domain in an agnostic and non-intrusive way.

We followed the same approach as with the DMF design—explained in Section 4.2.2—, we created a separate module with its own internal structures that serve only for the module purposes, the actual application domain data is controlled by the application. Hence the module *observes* and *applies* state changes to the application domain, for that purpose it has a dependency to the DMF. Therefore, our domain propagation strategy is clearly a mere distribution of the DMF notification calls, and persisted through the DMF domain control functionality.

In order to enhance the precision when dealing with dynamic entities, we can describe the entity behavior with many levels as we explained in Section 4 and distribute a higher level behavior, such as a position variation function instead of the position only. However, we need to provide some time synchronization algorithm as well as some kind of prediction algorithm for this setup to work. If the reader wants, the system is flexible enough to be adapted to support dynamic behavior distribution as our domain distribution is agnostic. Also, even if there are only static entities, but there is a very heavy user interference with the system, resulting in a network bottleneck, there is the possibility of adapting a predictive algorithm such as *Dead Reckoning*. A similar setup with dynamic entities has been developed by Ferreira [13]. His system provides distribution of dynamic entities in a visualizer by using the *SNTP* [27] time synchronization protocol and *Dead Reckoning*.

Our simple distribution strategy does not, however, contemplate dynamic entities and heavy user input with latency by itself, the distribution and redundancy algorithms are not covered here. Moreover, although there can be inconsistencies if the system is deployed in a non robust network, our desired scenario is mainly composed of static entities and our expected network setup is also a controlled robust one, there is no need to synchronize the time in every node, as well as no need for predictive algorithms. Our simple distribution technique has shown satisfactory results in our example system implementation, which contemplate effective for static scenes with a moderate amount of users making changes in the domain (Example system results are covered in Section 6.4). Nevertheless, this technique has some caveats. First of all, if the changes are to be broadcast, then all the nodes must be in the same sub-network. If they are in the same sub-network, there is still the problem that any other nodes in the sub-network will receive the changes, possibly creating conflict if two different data is being viewed by two different groups of nodes. Moreover, the required bandwidth grows proportionally to the frequency of changes in the visualization. However, in our design, we distribute only the application domain, such as behavior information and selections. All the heavy assets and data are loaded locally in each node, which alleviates the bandwidth requirements if compared to a classic graphical data distribution model.

The broadcasting requirement that every node must be in the same sub network may not be a problem at all. Such is because the utility of broadcasting information is to enable a node to quickly update multiple remote nodes with the same information, whereas using an one to one communication may take too long and put a bottleneck in the node that needs to update the others. Such bottleneck presents a problem for distributed visualization, as every frame needs to be synchronized, additionally there is one node that is responsible for updating the others and coordinating the frame synchronization, therefore a bottleneck is such node can delay the frame. However, distributed visualization setups, such as caves/display walls, are usually driven by a node cluster inside its own sub network, which allows broadcasting.

In a collaboration scenario, however, it is usually impossible to broadcast because the nodes are not in the same sub network, as a result of users working on the same data from different locations. Such limitation is actually not a problem in a collaboration scenario because broadcasting alleviates the bottleneck in the node that needs to update the others, as explained above. However, out of the distributed visualization scenario, there is no such bottleneck, or if there is, it won't compromise the experience of the users. Such is because the node that has to push the information does not need to coordinate frame synchronization, therefore not blocking every other node until all the information is pushed. Every node may receive updates to the domain asynchronously and the reception does not interfere with its frames. Moreover, the amount of bandwidth required is much smaller in collaborative environments, as there is no camera information passed to every node every frame.

5.2 Collaboration

In our shared domain scheme, whenever multiple users are making a collaborative visualization, they can simultaneously change the same data, therefore collisions are prone to happen. In online gaming realm, collisions are a very difficult problem to handle because there can be a massive amount of players interacting with the same data at the same time. Moreover there is usually no action of "take control of an entity"before altering it in any way, as the players are interacting with a simulated world, and requiring such action may break the immersion. Therefore, the collision problem needs to be handled very fast and smoothly so that all the players involved in the collision feel that the object that triggered the collision is being really shared. Such steep requirements are not the case in the scientific visualizer.

Entities in a scientific visualizer domain are not to be treated as objects in a game. It is possible to apply a system where a user must first take control of an entity in order to apply any changes to it. Such system eliminates the problem of users sharing an object. Still, even if there is no control system, the latency when solving a collision does not need to be so low as in games. Therefore, it is acceptable to have an arbiter node responsible to keep the consistency, and every change applied by every node must be verified by the arbiter first, and only the arbiter and other designated nodes by the arbiter can propagate the changes (see Figure 5.1).

In our example system, we just implemented simple collaboration with no special collision treatment, i.e., we set one node to be the arbiter and every

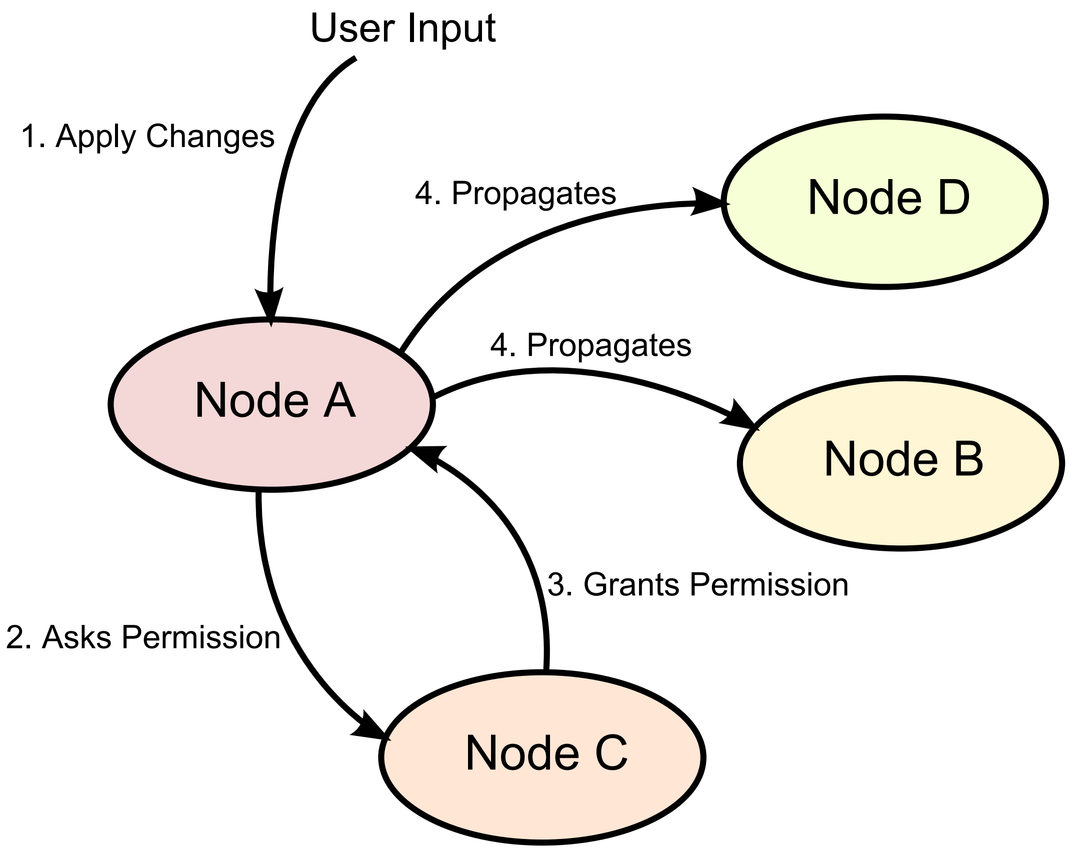

Figure 5.1: Arbiter Topology

update to be propagated must go through it.

5.3 Distributed Visualization

Considering that all the domain data is properly distributed as explained in the previous section, achieving distributed visualization becomes a simpler issue, we need to calculate the views for every display output based on a single observer and synchronize the frame displaying. These tasks must be assigned somehow to nodes in the visualization cluster, hence we need to create roles with appropriate responsibilities for the nodes. In order to accomplish distributed visualization in our design, it is imperative that we assign at least master and slave roles. There are many other roles for specific tasks that can be assigned, but we explain here only the necessary ones, and leave space for the extra in the implementation chapter. The node assuming the master role is responsible for calculating the frustums based on the virtual observer and synchronizing the time that every rendered frame is displayed across every display device. Consequently, the slave nodes passively receive what frustum they should use to render the scene and wait for the permission to display the rendered frame. Figure 5.2 shows an overview of the nodes responsibilities.

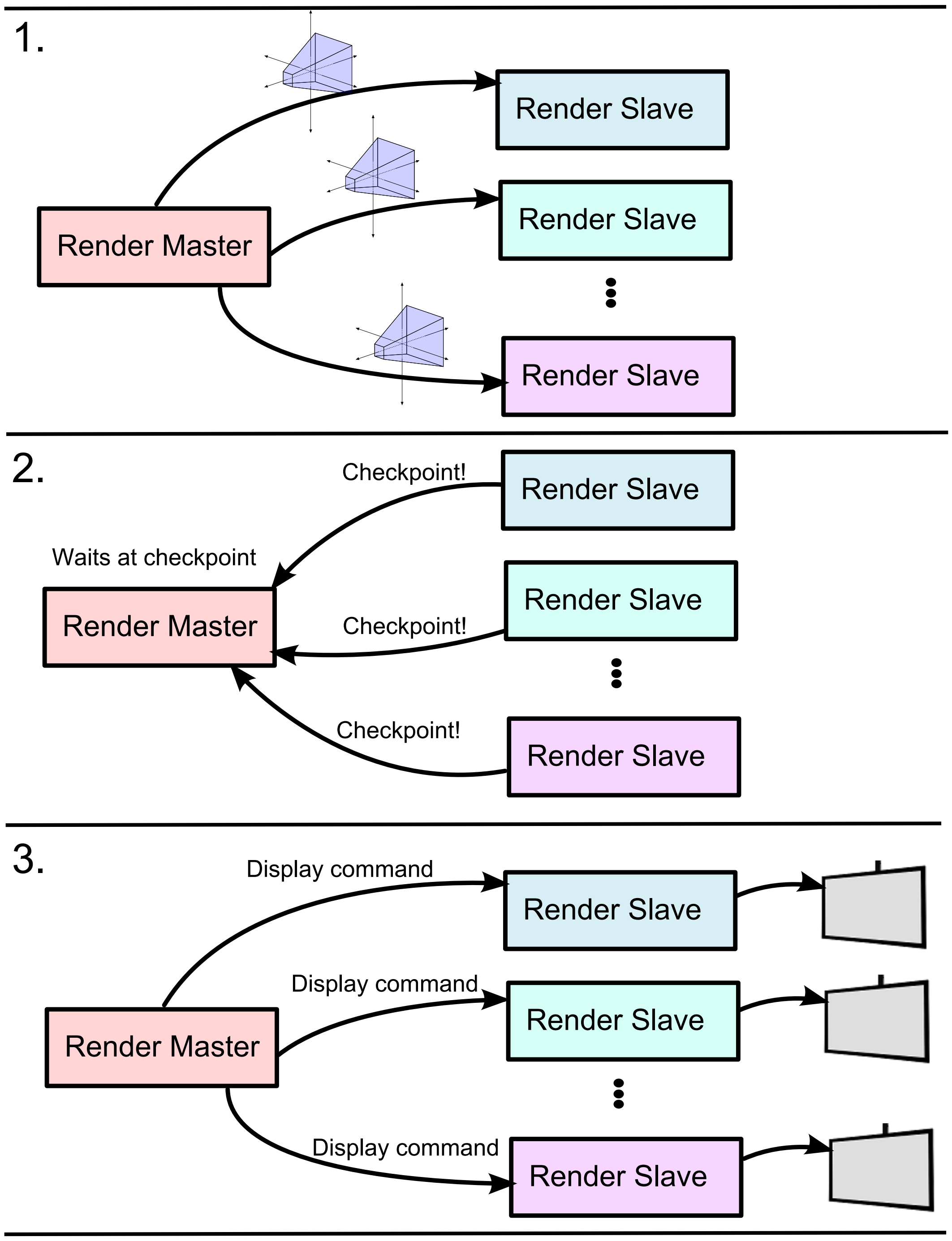

Figure 5.2: Steps towards displaying a frame with distributed visualization. 1- Master sends the frusta and commands the slaves to render. 2- Master waits at checkpoint until all Slaves reach it. 3- Master commands Slaves to display.

The frustum calculation is done based on the output screens. Every node has a number of outputs connected to it and they need to display the current view of the scene on those. The view of the scene, is based on a single observer—in a typical scenario—and therefore, this view should be split between the nodes. The master node needs to know the display wall / cave measurements and slave nodes disposition.

The frame synchronization is very simple. The master node must first command every node to render—with the appropriate frusta transferred. Then, after *every* slave node has finished rendering, the master node commands the slaves to present their rendered frames. If this synchronization strategy is not used, the user immersion can be broken, as there will be different frames being exhibited at the same time.

There are also possible variations in the network topology. E.g., in a cave setup, there may already be a cluster with one node responsible for splitting the frustum and synchronizing the frame, other nodes that only render and display, and the user can connect a notebook to the network, load projects and personal configurations and manipulate the scene through it. There may also be a dedicated node only with lots of different tracking devices configured and its role is only to receive and handle input.

In the following chapter when describing the system implementation, we will explain how to assign different roles for the nodes for with other roles other than master and slave—splitting the frustum and synchronizing the the frame, receiving and handling user input, manipulating the scene, rendering and displaying the rendered frame may be split across different roles for the nodes. There is however one case that our design is not directly created for; separating the rendering and displaying tasks among different nodes.

5.4
Integrating with the Visualizer

So the *DVCM* should be a generic DMF observer that propagates local changes to the network and network changes locally. In order to use the module, when assembling the visualizer's components through a startup script, there must be a list of available options in the *DVCM* interface in order to assign its role. These options must cover series of supported roles: local domain updates to be propagated throughout the network, incoming domain changes are going to be analyzed for collisions and confirmed or simply accepted, passively or actively render and display o the screen, among others. All of these possible setups are going to be covered and shown implementation-wise in the following chapter.

6
Implementation of the DVCM

As we already mentioned in the motivation section, our example system is an oil field visualizer implemented in C++ on top of the Coral [1] framework. In this chapter, we walk through the whole implementation of the DVCM module that currently works with our example system. We shall include some low level details like which tools and frameworks we used and how the system behaves during runtime.

6.1
Remoting Submodule

We shall start by laying the foundation for our distribution system to work - namely our networking submodule which is the low level communicating system that sits at the bottom layer of the *DVCM* module. We decided its communication paradigm to be based on an RPC approach (as explained in the related work section). As of such, an RPC based communication system needs to completely abstract the remoting of components and therefore be completely transparent to the programmer, which must treat every component as local. As explained above, the RPC communication paradigm is not suitable for massive, heterogeneous and unreliable network. However, our scenario is of small homogeneous graphics clusters. Our programming model can be preserved with RPC and the networking can be dealt with almost transparently.

Our RPC paradigm is similar to CORBA [24] by using a component based approach [15]. In other words, the user is able to remotely instantiate components and search for services and components. Any remotely acquired service or component works exactly as a local one, except for the thrown exceptions, which will have a whole new set of remoting error exceptions that will increment the actual component-thrown exceptions. Although these exceptions obviously break the transparency of the RPC, there is no such thing as perfect network transparency, and therefore some precautions must be taken.

The API consists of a single interface, an INode, which will provide Client and Server functionality. The client sub-API provides functions for

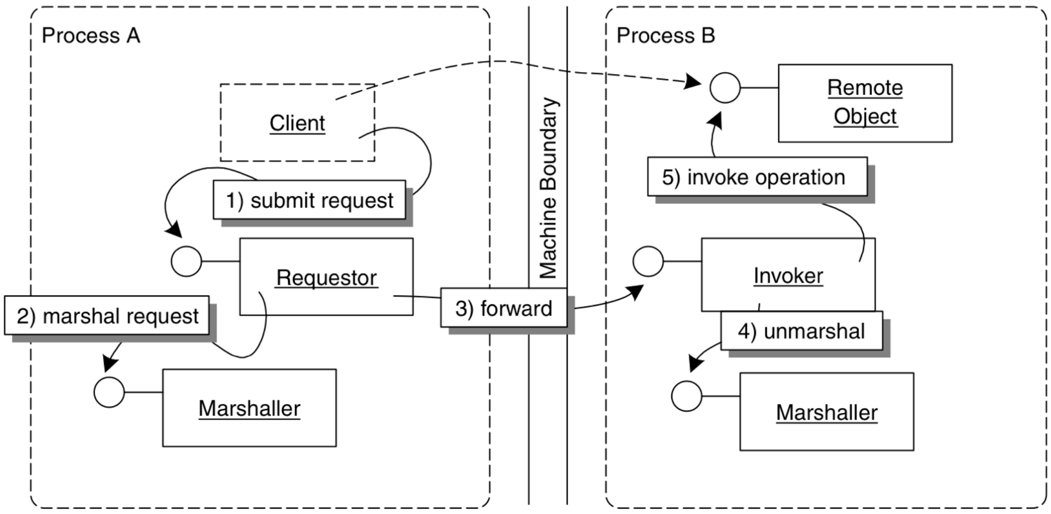

Figure 6.1: The broker pattern diagram as shown in [37]

instantiating and searching for remote components. Whereas the server sub-API should have a reactive method that handles all incoming requests and dispatches the invocations for the appropriate objects. Also, the user should be able to publish local instances through the server functionality of the INode. Therefore, in order to use the INode, the user must keep regularly calling the reactive method in order to handle remote requests, except when there is no need to heed remote requests, i.e., the node is client only. However, if the user passes any argument that is a local service, then the server functionality of the node will be necessary, as the node that receives the service will need it.

We also implemented a system for synchronization among a group of connected nodes, which we call *Checkpoints*. The system needs a master and N slaves to work. The master creates a checkpoint, broadcast it to all the slaves and stops at the checkpoint. Every slave, when reaching the checkpoint send a message to the master and waits until the clearance of the checkpoint. Eventually, when all the slaves reach the checkpoint and inform the master, the checkpoint is cleared and the master broadcasts a clear message to all the slaves so they can continue their normal workflow.

6.1.1 Internal Implementation

The internal implementation of the node is coarsely based on the broker pattern [11] [37]. The main characters of the pattern are shown in Figure 6.1 and are:

- *ClientProxy*, which is the component that provides the interface between the RPC module and the user.

– *Requestor*, which receives the function calls from the ClientProxy and communicates with remote nodes

– *Marshaller*, which serializes/desserializes the calls into messages to be sent

– *Invoker*, which receives the invocation data and invoke the method of the actual component

The interface between the *Proxy* that will be used by the user in the client and *Requestor* can be direct or transparently through a *ClientProxy* as described by [37]. In our module, we use a *ClientProxy*, and by the usage of component reflection and introspection [21] the *Requestor* is called generically by the *ClientProxy*. Any service method that has no value returned is automatically called asynchronously, whereas methods that have return types are called synchronously and thus are blocking. Notice that the asynchronous calls can fail and the caller is going to be oblivious about it, so there needs to be a protective mechanism when using them, such as callbacks and confirmation statuses.

Local instances lifecycle shall be managed through leasing [11]. Whenever a node requests a remote instance, a lease for that node is created inside the node that contains the instance. That lease can be set to expire after a determined time or left to be removed manually. If a local instance is not used locally and has no more leases it can be destroyed or returned to a pooling service.

For the transport layer, we used a library that creates abstractions for connections, messages and guarantees message delivery. From the many options researched, we decided to use ZeroMQ [7], which provides a well abstracted socket system that handles N-to-N connections transparently and delivers entire messages through various transports. It also supports multi-cast via PGM [34].

6.2 Distributed Shared Objects

With the RPC module available, we move on to the distribution of the application domain, namely the *DSO* system. We implemented the distribution in a publish-subscribe scheme, thus we created two main interfaces, unmistakably *IPublisher* and *ISubscriber*. The *IPublisher* simply has one method, which provides registration of subscribers, and consequently accepts an ISubscriber interface as a parameter. The *ISubscriber* interface has two methods, *onSubscribed* and *onPublish*. The former is a initialization method, it means that when the subscriber is registered in the publisher, the publisher must push all of its current state into the subscriber. The latter is the continuous update method that the publisher will call whenever there is a change in its state. Clearly, we implemented components that provide the functionality behind these two interfaces.

The component that provides *IPublisher* also provides an interface for observing the domain via DMF (DMF is the Domain Model Framework explained in Section 4.2). Then, whenever the DMF pushes a changeset into the Publisher component, the latter internally rearrange the way the changes are structured to make them compatible for the subscribers (if necessary) and call every registered *ISubscriber*'s with the restructured changes.

The component that provides *ISubscriber* has access to the domain and consequently the DMF control API. Then, whenever a changeset comes through the *ISubscriber* interface, the component propagate these changes into the domain, and the DMF naturally propagates them throughout the rest of the application.

These two components by themselves are not sufficient to realize the *DSO*, we need to use the *RPC* module. There can be clients being publishers and servers being subscribers and the other way around as well. E.g., if we want a Server-Publisher/Client-Subscriber set up, the node that wishes to be a Publisher-node, needs to make its *IPublisher* interface available for remote use (avoiding the term publish again for clarity's sake) through the *RPC*. Consequently, the node that wishes to be a Subscriber-node needs to search for the *IPublisher* interface of the Publisher-node and register its own *ISubscriber* on it. The *RPC* module will automatically create also a Proxy for the *ISubscriber* interface inside the Publisher-node, allowing the publisher component to call the *ISubscriber* methods as local ones.

6.2.1 Configuring the Topology

With a node being only able to assume one role, it is only possible to achieve a master-slave topology, where the master is going to be a publisher and the slaves subscribers. However, we can easily set up a node to have as many publisher and subscriber components as desired. Accordingly, we can make two way updates in the master-slave topology just by adding a subscriber to the master and publishers to the slaves. However, when a node has both Publisher and Subscriber components, if no precaution is taken, an update received through the Subscriber will alter the domain state, and thus propagated again by the Publisher, which may lead to unnecessary network

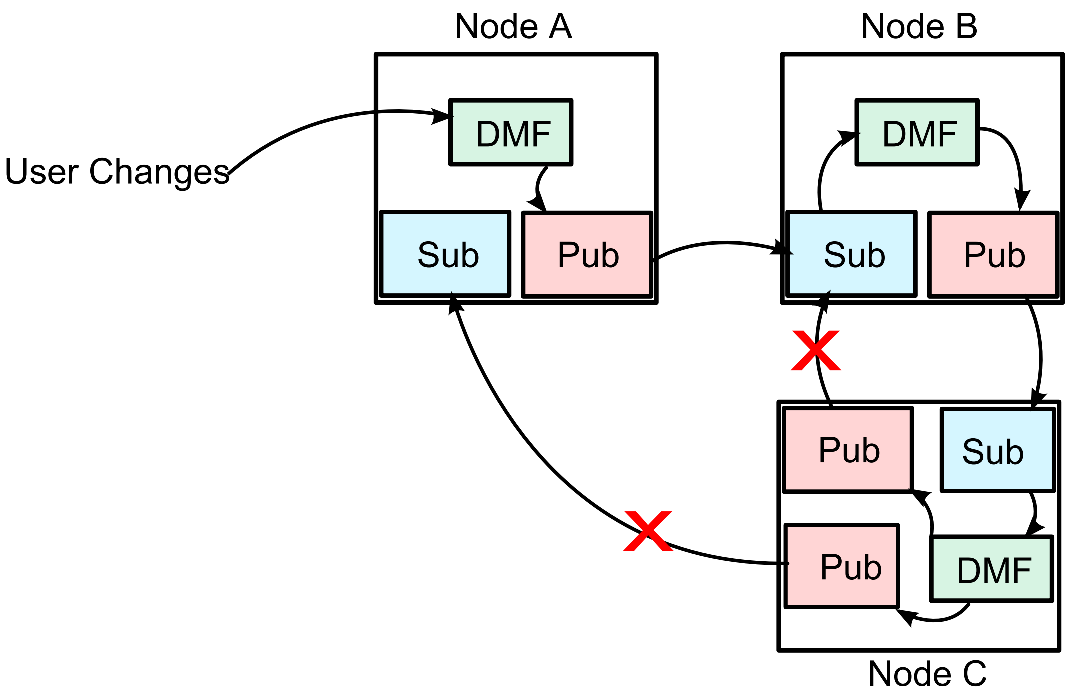

Figure 6.2: Unnecessary cycles in the network traffic

traffic depending on the scenario. E.g., a two node collaboration topology, where *NodeA* has its domain altered by user input and publishes the changes. Consequently, *NodeB* receives the updates through its Subscriber component, which changes *NodeB*'s domain. *NodeB*'s Publisher observed the changes in the domain and propagates them again to *NodeA*, which has no use for the updates as its domain is already up-to-date (Figure 6.2).

In order to avoid this unnecessary traffic overhead, we must incur in the changeset, which nodes already up-to-date with the latest version of the domain. The Subscriber that receives the change updates another component that holds a map of known hosts and their current known domain versions. The publisher always looks up in this map for publishers that already have the latest domain version and do not propagate the changes to them.

6.3 Distributed Visualization

As explained before in Section 5.3, the distributed visualization requires a simple master-slave topology. This topology is necessary for propagation of domain and frame synchronization. In this section, we explain the implementation of a distributed visualization scenario where there is a cluster of *N+1* nodes and *N* display devices. From these *N+1* nodes, *N* nodes are called *RenderSlaves* and have a display device attached. The other node is called *RenderMaster*. We reference the *ISubscriber* and *IPublisher* interfaces explained in Section 6.2.

In our sample scenario, the *RenderMaster* is responsible for distributing the domain to all *RenderSlaves* and synchronizing the frame. Additionally, it may assume different new responsibilities if the *UserMaster*, the *DeviceMaster* or both are used. However, we start with the simplest case, with no additional roles. In this simple case, the *RenderMaster* loads all the domain from local files, as well as the position of every display device in real-world and which *RenderSlave* is attached to it.

For an improved usability of the system, we make *RenderSlave*s being the servers and the *RenderMaster* being the client. That is, the *RenderSlave*s are configured to be totally passive, each one of them will start the *RPC* module, and publish two interfaces on it: an *ISubscriber* and an *IRenderSlave*, which is explained later on. After making these interfaces available for remote access, each *RenderSlave* just waits indefinitely, therefore acting as a server. Moreover, the *RenderMaster* looks up in the configuration files for the addresses of all the *RenderSlaves*, then gets the published interfaces via *RPC*, thus acting as a client.

From the retrieved interfaces, the *RenderMaster* gets all the *ISubscriber* ones and add to its own *IPublisher*. The component that provides the *IPublisher* also provides an interface that observes the DMF. Consequently, all the changes to the domain are propagated to all the *ISubscriber*s. Additionally, the component that provides the *ISubscriber* interface inside the *RenderSlave* has a receptacle for the DMF control interface, which is used to apply to its local domain any changeset received from the *ISubscriber* interface. Now the domain is shared among the nodes and consequently the same scene for visualization. Therefore, we need only to make sure the frustums are properly calculated and the exhibition of the frames is synchronized. As mentioned before, we do not delve into details of the frustum calculation as it is not in this work's scope.

When starting the scene visualization, the user will be represented by a virtual observer, which has a position and a view direction. There is an external component that calculates the frustums based on the view direction and the configuration file with all the display devices real-world positioning. Therefore, by using this component, the *RenderMaster* obtains the observer position and the frustums, namely the necessary parameters for rendering. Then, it passes these parameters to the *IRenderSlave* interfaces, which are already published by every *RenderSlave*. This interface provides functionalities to the *RenderMaster* to control the *RenderSlave*, namely the setting of frustum and position and the frame display command.

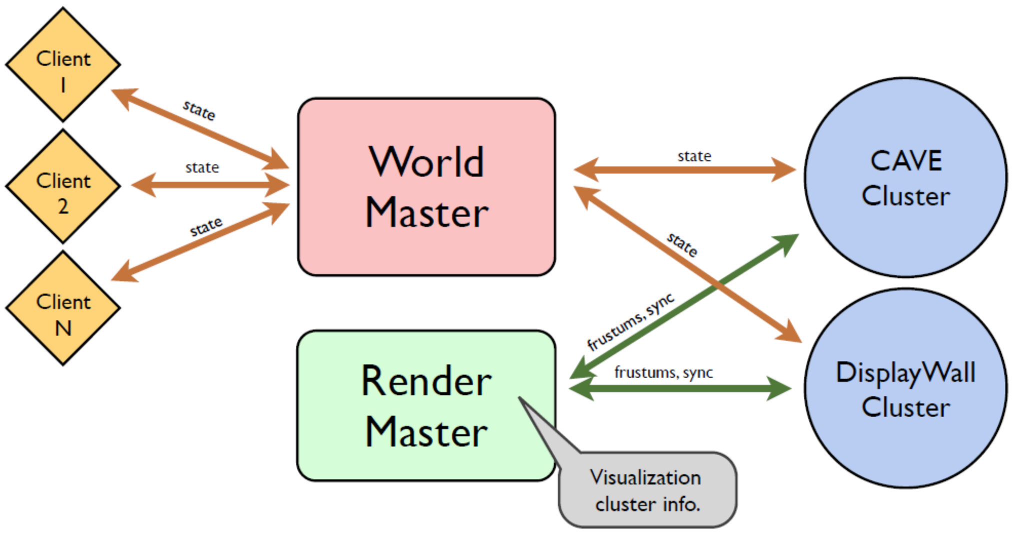

Figure 6.3: Topology of a system with Collaboration and Distributed Visualization working together

With the necessary parameters for the rendering the scene received, the *RenderSlaves* start to render. Meanwhile, the *RenderMaster* creates a *Checkpoint* (explained in Section 6.1) and waits for all the rendering to be finished. After finishing the render, every *RenderSlave* hits the checkpoint and inform the *RenderMaster*. Then, with all the rendering finished, the *RenderMaster* clears the checkpoint and calls the frame display command in the *IRenderSlave* interfaces.

If the user connects a separate node called *UserMaster* where he loads his own projects and manipulates the view, the Distributed Visualization workflow will be almost the same, except that the *RenderMaster* needs to have a *ISubscriber* interface published as well. The *UserMaster* access this interface and sets to its local *IPublisher* which works in the exact same way as the one in the *RenderMaster*. This extensible approach can work indefinitely for as many separates nodes as we want, i.e., a node for picking objects, a node for receiving input from trackers, and so on. Figure 6.3 shows how an ideal network with fine grained collaboration and distributed visualization tasks spread across multiple nodes.

6.4 Results

We show in this section how the implemented example system behaves under local and distributed visualization. Our goal is to measure how much impact the distribution and frame display checkpoint impacts the performance. Therefore we registered the frame rates of the visualization while varying the

Group	A	B	C	D	A, B	A,B,C	A,C,D	A,B,C,D
Average FPS	430	416	220	216	356	199	195	191
FPS loss	—	—	—	—	14.4%	11.3%	11.5%	11.4%

Table 6.1: Frame rates and performance comparison of different cluster configurations

number of nodes in the cluster and which nodes used. As mentioned before, the example system has no multicast implemented, thus all the tests have been executed with only unicast communication between the nodes. We used 4 nodes with similar configuration in our tests (Figure 6.4):

Our tests are consisted of the visualization of a dummy scene with 1300000 triangles with all culling disabled. This scene is just a collection of triangles in a certain position, but emulates a typical scene in our application usage scenario. During this visualization, we recorded the frames per second while the camera followed a predefined path. We execute these tests for a series of combination of nodes as shown in table.

In table 6.1, we show in the "Average FPS“ row how many frames per second each setup achieved and in the "FPS loss“ row the loss of performance of the given node group when compared to the performance of the worst local node performance of the group. Notice that the first 4 scenarios are of the nodes executing the tests locally. The results show that there is a performance loss of less than 15% with networking due to latency of the network in any of the cases. Moreover, considering that nodes C and D have a similar performance locally and since a group performance limit is equal the weakest node's local performance, the comparison of performance between groups A,B,C and A,B,C,D is important to show us the raw impact of adding another node to a group, which is a negligible loss of less than 1%.

As can be seen, although the performance suffers an expected loss with distribution, our system can easily provide interactive frame rates during the visualization of large scenes in a distributed visualization scenario 6.1, which is the usual desired cave disposition and our ultimate goal with the project.

We tested the system in the same 4 node scenario with a collaborative scene following an arbiter topology. The tests successfully worked and there had been no inconsistencies or noticeable performance hindrances as expected.

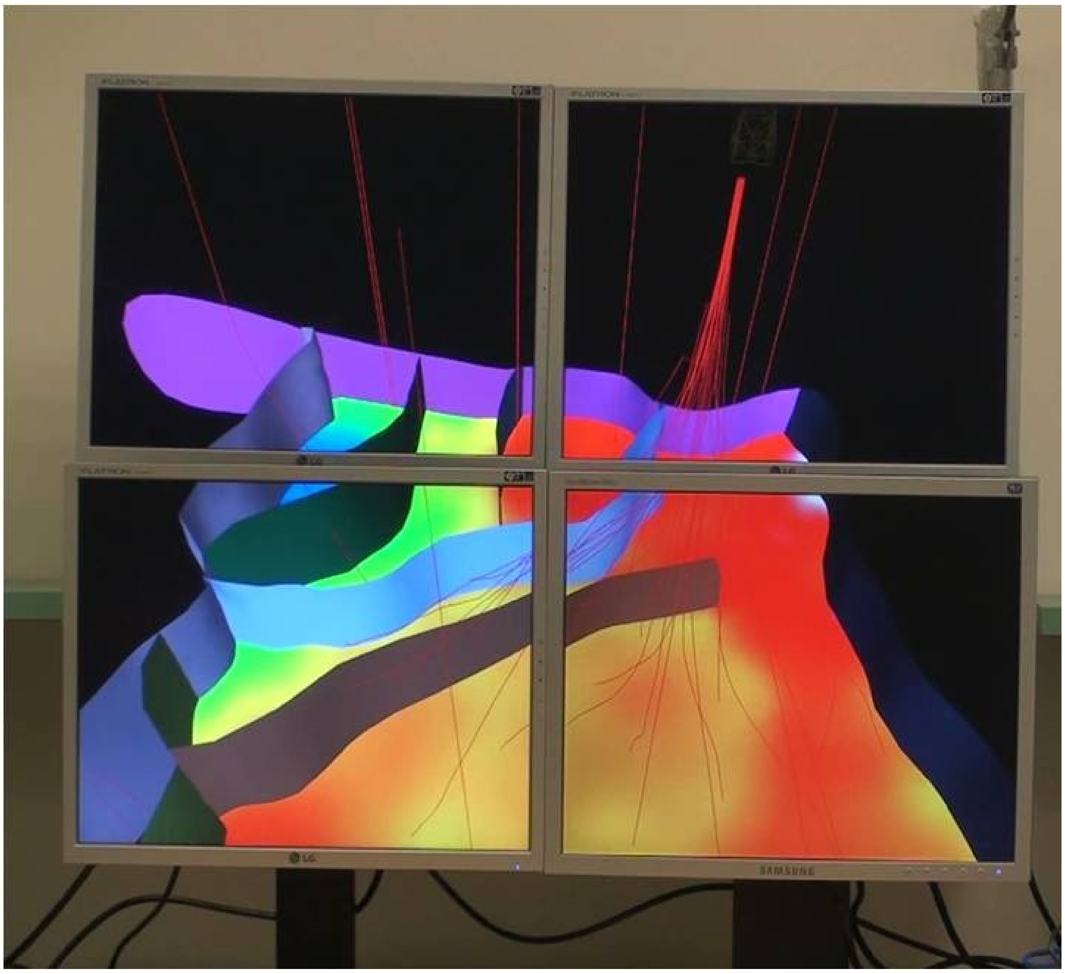

Figure 6.4: Distributed visualization with 4 nodes of a typical scene in Siviep

7
Conclusion

In this work, we presented our design of an extensible and modular visualizer as well as the design of a module that provides distributed visualization and collaboration for the visualizer. By following these designs, we implemented an example system and tested the synchronism of the distributed visualization and the consistency of the collaboration among multiple nodes, we also evaluated the impact on performance caused by the distributed visualization.

We discussed first the relevant concepts and problems of designing and implementing a real time immersive visualizer, from which we extracted our main architectural requirements—modularity and extensibility. Following we presented our design for the visualizer inspired by MVC architecture and explained how we try to fulfill the aforementioned requirements. We continued by presenting the design of the module responsible for distributed visualization and collaboration, its interactions with the visualizer and why our described design for the visualizer simplifies its implementation and usage. We concluded with the implementation of such module, followed by the example visualizer and its results.

We expected to indicate how our MVC-inspired design made possible the development of a module that transparently provided distributed visualization and collaboration to a visualizer. Also, how the design enabled the substitution of parts of the system easily. E.g., the scene graph implementation can be switched between a very efficient and licensed per station library for displaying a scene in massive immersive environments and a cheaper licensed library for common desktop usage. These reasons along with others explained in chapter 1 are what motivated this work as a solution for our real project. We try to summarize here the key points that we believe we have addressed with our design and can impact overall productivity of the development of real time and efficiency-focused applications:

Complexity The separation of a system into modules with explicit interfaces tend to isolate the low level details of implemented features and create abstractions, which enhance the productivity of the business logic development by reducing the complexity of the system. The isolation of all the distribution code inside our designed module leverages the productivity of the business and graphical developers in our visualizer.

Prototyping Designing a long development cycle for a product can be extremely complex. Therefore, designing and implementing the system iteratively can leverage productivity and also accommodate late changes in product requirements. By making the application extensible, we give the developers and easy way to prototype new functionality and implementations. Our component based design enabled our system to be distributed after its first version without any significant modifications to it.

Flexibility The possibility of switching between different functionality with no code change is important for a software that must assume different roles depending on the scenario. Our MVC design allows our system to have as many output and input endpoints for distribution of the business data in our Distributed Shared Objects module.

Testability A modular system can have a loose coupling among its modules, which makes easier to isolate a single module and test its provided functionality with mock modules connected to it. We can test our Distributed Shared Objects and Distributed Visualization submodules locally by switching the RPC module under it for a local implementation of it.

Maintainability All the items below makes the system easier to understand, extend, test and switch parts. All of these greatly reduces the time to understand a part of the system by the developers.

Our example system implementation showed that our design worked for the expect scenario. The synchronism among multiple output displays was achieved without any dedicated hardware and also without even the need to use broadcast messages. A loss of less than 15% for a 4 node setup, which is our common CAVE scenario has been a very good result, confirming our design as a proper solution. Furthermore, the consistency among the scenes when using the software collaboratively was achieved without any noticeable performance hindrances.

7.1 Future Work

Our distributed visualization strategy is not suitable for scenarios where there is no single node in the cluster with a number of output displays that creates a rendering bottleneck on it, which happens due to the amount of graphical processing on it. However, if desired, the task of rendering can be separated for the task of displaying. By creating this new layer of parallelism, the rendering task can be distributed equally among the nodes independently of the number of output displays connected to each one of them. However, it becomes necessary to recompose the final image based on a given recompositing strategy, which can adds complexity and inefficiency if not necessary.

The consistency among multiple stations when working collaboratively is currently only designed for reliable environments without simultaneous edition of the same entity. Therefore no special collision treatment of time synchronizing strategy is currently needed. However, if the visualizer needs to be deployed in a slow and non reliable network, with dynamic entity behavior, simultaneous edition of the same entity, and so on, the distribution module can be extended to support prediction algorithms, time synchronizing strategies and dynamic behavior descriptions.

Although the RPC module supports broadcasting in its current version, there is no service oriented abstraction for it. A node has to call in a reflective way the methods on other nodes, making it complex and unintuitive. Nevertheless, synchronous distributed visualization can be achieved with only unicast with some performance loss. If necessary, for efficiency in N-to-N communication, without breaking the component oriented paradigm, we intend to add the support for Collective Interfaces [8] following the implementation of [33], that use annotations [25] which are supported by Coral [1]. For ease of use and configuration, we plan to implement the Lookup pattern [19] with any node being able to assume on demand the role o service lookup directory.

Referências Bibliográficas

[1] **Coral - lightweight c++ component framework**. www.libcoral.org.

[2] **Java remote method invocation**. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html.

[3] **Microsoft .net remoting**. http://msdn.microsoft.com/en-us/library/72x4h507(v=vs.80).aspx.

[4] **Openscenegraph**. http://www.openscenegraph.org/.

[5] **Opensg**. www.opensg.org.

[6] **Visualization library**. http://www.visualizationlibrary.org//.

[7] **Zeromq**. www.zeromq.org.

[8] BAUDE, F.; CAROMEL, D.; HENRIO, L. ; MOREL, M.. **Collective interfaces for distributed components**. In: CLUSTER COMPUTING AND THE GRID, 2007. CCGRID 2007. SEVENTH IEEE INTERNATIONAL SYMPOSIUM ON, p. 599–610, May.

[9] BIERBAUM, A.; JUST, C.; HARTLING, P.; MEINERT, K.; BAKER, A. ; CRUZ-NEIRA, C.. **Vr juggler: a virtual platform for virtual reality application development**. In: VIRTUAL REALITY, 2001. PROCEEDINGS. IEEE, p. 89–96, March.

[10] BOEHM, B.. **Software Engineering Economics**. Prentice Hall, 1991.

[11] BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P. ; STAL, M.. **Pattern-Oriented Software Architecture - A System of Patterns**. John Wiley and Sons, 1996.

[12] EILEMANN, S.; MAKHINYA, M. ; PAJAROLA, R.. **Equalizer: A scalable parallel rendering framework**. IEEE Transactions on Visualization and Computer Graphics, 15:436–452, 2009.

[13] FERREIRA, A.. **Uma arquitetura para a visualização distribuída de ambientes virtuais**. Master's thesis, PUC-Rio, 1999.

[14] GAMMA, E.; HELM, R.; JOHNSON, R. ; VLISSIDES, J.. **Design Patterns - Elements of Reusable Object-Oriented Software**. Addison-Wesley, 1995.

[15] HEINEMAN, G.; COUNCILL, W.. **Component based software engineering : putting the pieces together**. Addison-Wesley, 2001.

[16] HUMPHREYS, G.; HOUSTON, M.; NG, R.; FRANK, R.; AHERN, S.; KIRCHNER, P. D. ; KLOSOWSKI, J. T.. **Chromium: a stream-processing framework for interactive rendering on clusters**. ACM Trans. Graph., 21(3):693–702, July 2002.

[17] IERUSALIMSCHY, R.; DE FIGUEIREDO, L. H. ; CELES, W.. **The evolution of lua**. In: PROCEEDINGS OF THE THIRD ACM SIGPLAN CONFERENCE ON HISTORY OF PROGRAMMING LANGUAGES, HOPL III, p. 2–1–2–26, New York, NY, USA, 2007. ACM.

[18] KELEHER, P. J.. **Decentralized replicated-object protocols**. In: PROCEEDINGS OF THE EIGHTEENTH ANNUAL ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, PODC '99, p. 143–151, New York, NY, USA, 1999. ACM.

[19] KIRCHER, M.; JAIN, P.. **Pattern-Oriented Software Architecture: Patterns for Distributed Services and Component**. John Wiley and Sons, 2004.

[20] LIU, X.; JIANG, H. ; SOH, L.-K.. **A distributed shared object model based on a hierarchical consistency protocol for heterogeneous clusters**. In: CLUSTER COMPUTING AND THE GRID, 2004. CCGRID 2004. IEEE INTERNATIONAL SYMPOSIUM ON, p. 515–522, April.

[21] MAES, P.. **Computational reflection**. The Knowledge Engineering Review, 3:1–19, 1988.

[22] MAKHINYA, M.; EILEMANN, S. ; PAJAROLA, R.. **Fast compositing for cluster-parallel rendering**. In: PROCEEDINGS OF THE 10TH EUROGRAPHICS CONFERENCE ON PARALLEL GRAPHICS AND VISUALIZATION, EG PGV'10, p. 111–120, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[23] MICROSOFT PATTERNS AND PRACTICES TEAM. **Microsoft Application Architecture Guide**. Microsoft Press, 2009.

[24] OMG. **The corba specification**. Internet Draft. `http://www.omg.org/spec/CORBA/`.

[25] SUN MICROSYSTEMS. **Annotations**. `http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html`, 2011.

[26] VISUAL C++ TEAM. **C++ and cloud computing.**

[27] MILLS, D.. **Simple network time protocol (sntp) version 4 for ipv4, ipv6 and osi**, 1996. Internet RFC 2030.

[28] MOLNAR, S.; COX, M.; ELLSWORTH, D. ; FUCHS, H.. **A sorting classification of parallel rendering**. Computer Graphics and Applications, IEEE, 14(4):23–32, 1994.

[29] PIMENTEL, M.; FUKS, H.. **Sistemas Colaborativos**. Elsevier-Campus-SBC, 2011.

[30] ROEHL, B.. **Some thoughts on behavior in vr systems**, 1995. University of Waterloo.

[31] ROY, P.. **The road to distributed programming: From network transparency to structured overlay networks and onward to self management**, 2006. invited talk at Universiteit Antwerpen,.

[32] ROY, P. V.. **On the separation of concerns in distributed programming: Application to distribution structure and fault tolerance in mozart**. In: IN INTERNATIONAL WORKSHOP ON PARALLEL AND DISTRIBUTED COMPUTING FOR SYMBOLIC AND IRREGULAR APPLICATIONS (PDSIA 99), TOHOKU, 1999.

[33] SILVEIRA, P.. **Projeto e implementação de interfaces coletivas em um middleware orientado a componentes de software**. Master's thesis, PUC-Rio, 2011.

[34] SPEAKMAN, T.; CROWCROFT, J.; GEMMELL, J.; FARINACCI, D.; LIN, S.; LESHCHINER, D.; LUBY, M.; MONTGOMERY, T.; RIZZO, L.; TWEEDLY, A.; BHASKAR, N.; EDMONSTONE, R.; SUMANASEKERA, R. ; VICISANO, L.. **Pragmatic general multicast (pgm) reliable transport protocol**. Internet Draft, 1998. CISCO Systems.

[35] TAYLOR, R. N.; MEDVIDOVIC, N. ; DASHOFY, E. M.. **Software Architecture: Foundations, Theory, and Practice.** Wiley Publishing, 2009.

[36] VINOSKI, S.. **Where is middleware?** Internet Computing, IEEE, 6(5):92–95, Sep/Oct.

[37] VOELTER, M.; KIRCHER, M. ; ZDUN, U.. **Remoting Patterns**. John Wiley and Sons, 2004.